GPU CUDA Contd.

High-Performance Computing

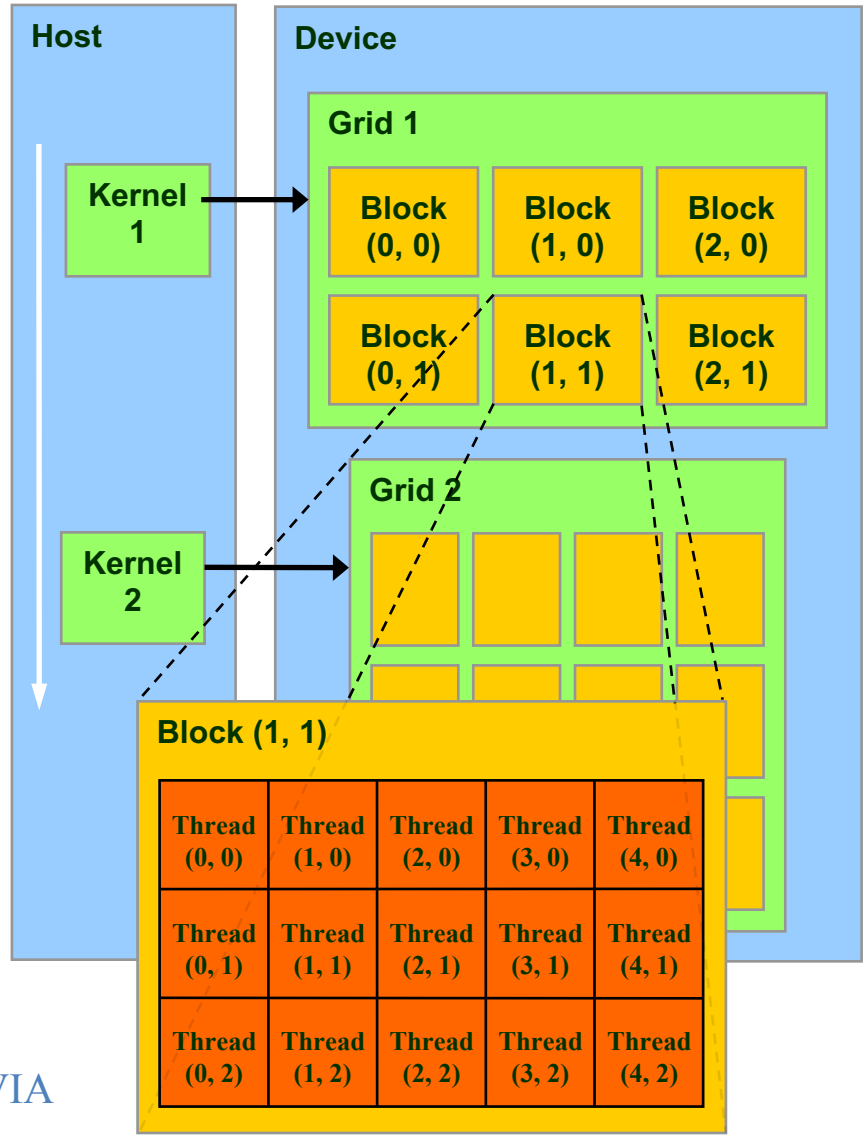
Summer 2021 at GIST

Tae-Hyuk (Ted) Ahn
Department of Computer Science
Program of Bioinformatics and Computational Biology
Saint Louis University

SAINT LOUIS
UNIVERSITY
EST. 1818

Recap: Thread Batching: Grids and Blocks

- A kernel is executed as a grid of thread blocks
  - All threads share data memory space
- A thread block is a batch of threads that can cooperate with each other by:
  - Synchronizing their execution
    - For hazard-free shared memory accesses
  - Efficiently sharing data through a low latency shared memory
- Two threads from two different blocks cannot cooperate



Courtesy: NDVIA

Important Keywords

- SM -- Streaming Multiprocessor
  - https://en.wikipedia.org/wiki/Stream_processing
- Threads – A thread of execution is the smallest sequence of programmed instructions that can be managed independently by a scheduler, which is typically a part of the operating system
  - https://en.wikipedia.org/wiki/Thread_(computing)
- Blocks – A thread block is a programming abstraction that represents a group of threads that can be executed serially or in parallel.
  - https://en.wikipedia.org/wiki/Thread_block_(CUDA_programming)
- Grids – Nvidia GRID is a family of graphics processing units (GPUs)
  - https://en.wikipedia.org/wiki/Nvidia_GRID
- Warps -- On the hardware side, a thread block is composed of 'warps'. A warp is a set of 32 threads within a thread block such that all the threads in a warp execute the same instruction. These threads are selected serially by the SM
  - https://en.wikipedia.org/wiki/Thread_block_(CUDA_programming)#Warps

NVIDIA call their parallel programming model SIMT, but

- SIMD computation – Single instruction, multiple data (SIMD) is a class of parallel computers in Flynn's taxonomy.
  - In SIMD, elements of short vectors are processed in parallel.
  - https://en.wikipedia.org/wiki/SIMD
- SIMT computation – Single instruction, multiple threads
  - SIMT is somewhere in between – an interesting hybrid between vector processing and hardware threading.
  - https://en.wikipedia.org/wiki/Single_instruction,_multiple_threads
- SMT – Simultaneous Multithreading
  - Each model exploits a different source of parallelism.
- SIMD < SIMT < SMT: parallelism in NVIDIA GPUs
  - https://yosefk.com/blog/simd-simt-smt-parallelism-in-nvidia-gpus.html (Please read this!)

deviceQuery

- Find and run

  deviceQuery

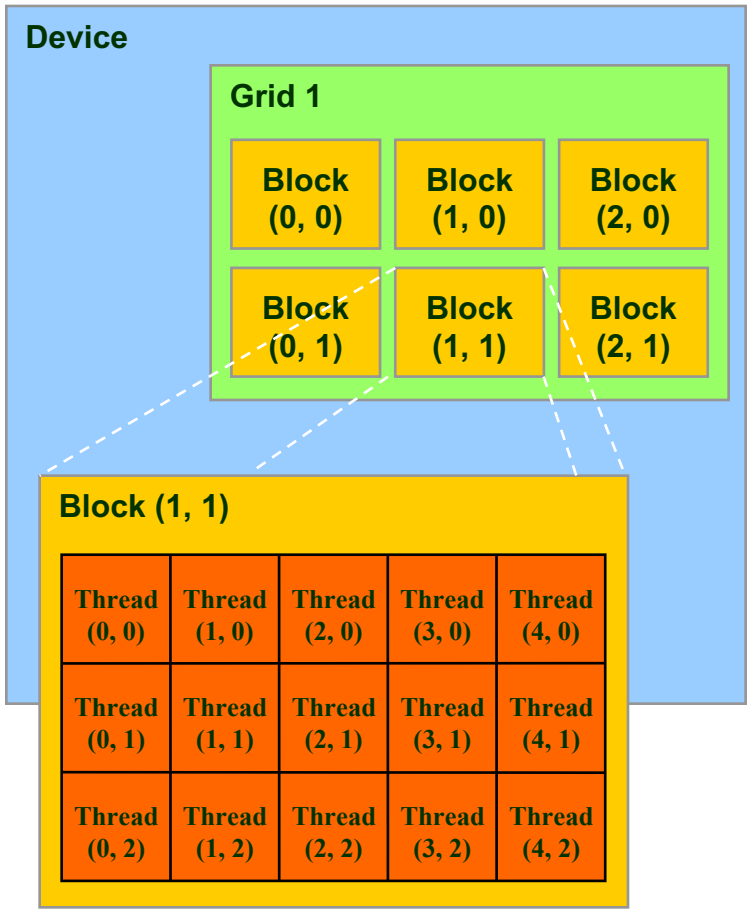
  under ~/NVIDIA_CUDA-11.X_Samples.

- Max number of threads per block is 1024!
- It shows Compute Capability!

```
aigrad1@dgx-v100-n1:/mnt/aigrad1/Lab/CUDA$ /usr/local/cuda/samples/bin/x86_64/linux/release/deviceQuery
/usr/local/cuda/samples/bin/x86_64/linux/release/deviceQuery Starting...

 CUDA Device Query (Runtime API) version (CUDART static linking)

Detected 1 CUDA Capable device(s)

Device 0: "Tesla V100-SXM2-32GB"
  CUDA Driver Version / Runtime Version          10.1 / 10.1
  CUDA Capability Major/Minor version number:    7.0
  Total amount of global memory:                 32480 MBytes (34058272768 bytes)
  (80) Multiprocessors, ( 64) CUDA Cores/MP:     5120 CUDA Cores
  GPU Max Clock rate:                            1530 MHz (1.53 GHz)
  Memory Clock rate:                             877 Mhz
  Memory Bus Width:                              4096-bit
  L2 Cache Size:                                 6291456 bytes
  Maximum Texture Dimension Size (x,y,z)         1D=(131072), 2D=(131072, 65536), 3D=(16384, 16384, 16384)
  Maximum Layered 1D Texture Size, (num) layers  1D=(32768), 2048 layers
  Maximum Layered 2D Texture Size, (num) layers  2D=(32768, 32768), 2048 layers
  Total amount of constant memory:               65536 bytes
  Total amount of shared memory per block:       49152 bytes
  Total number of registers available per block: 65536
  Warp size:                                     32
  Maximum number of threads per multiprocessor:  2048
  Maximum number of threads per block:           1024
  Max dimension size of a thread block (x,y,z): (1024, 1024, 64)
  Max dimension size of a grid size    (x,y,z): (2147483647, 65535, 65535)
  Maximum memory pitch:                          2147483647 bytes
  Texture alignment:                             512 bytes
  Concurrent copy and kernel execution:          Yes with 6 copy engine(s)
  Run time limit on kernels:                     No
  Integrated GPU sharing Host Memory:            No
  Support host page-locked memory mapping:       Yes
  Alignment requirement for Surfaces:            Yes
  Device has ECC support:                        Enabled
  Device supports Unified Addressing (UVA):      Yes
  Device supports Compute Preemption:            Yes
  Supports Cooperative Kernel Launch:            Yes
  Supports MultiDevice Co-op Kernel Launch:      Yes
  Device PCI Domain ID / Bus ID / location ID:   0 / 133 / 0
  Compute Mode:
     < Default (multiple host threads can use ::cudaSetDevice() with device simultaneously) >

deviceQuery, CUDA Driver = CUDART, CUDA Driver Version = 10.1, CUDA Runtime Version = 10.1, NumDevs = 1
Result = PASS
```

Recap: Block and Thread IDs

- Threads and blocks have IDs
  - So each thread can decide what data to work on
  - Block ID: 1D, 2D, or 3D (`blockIdx.{x,y,z}`)
  - Thread ID: 1D, 2D, or 3D (`threadIdx.{x,y,z}`)
- Simplifies memory addressing when processing multidimensional data
  - Image processing
  - Solving PDEs on volumes
  - …



Courtesy: NDVIA

Recap: Built-in Variables for Block and Thread

Built-in variables:

- `threadIdx.{x,y,z}` – thread ID within a block
- `blockIdx.{x,y,z}` – block ID within a grid
- `blockDim.{x,y,z}` – number of threads within a block
- `gridDim.{x,y,z}` – number of blocks within a grid

`kernel<<<nBlocks,nThreads>>>(args)`

Invokes a parallel kernel function on a grid of `nBlocks` where each block instantiates `nThreads` concurrent threads

Parallelizing vector addition using multithread

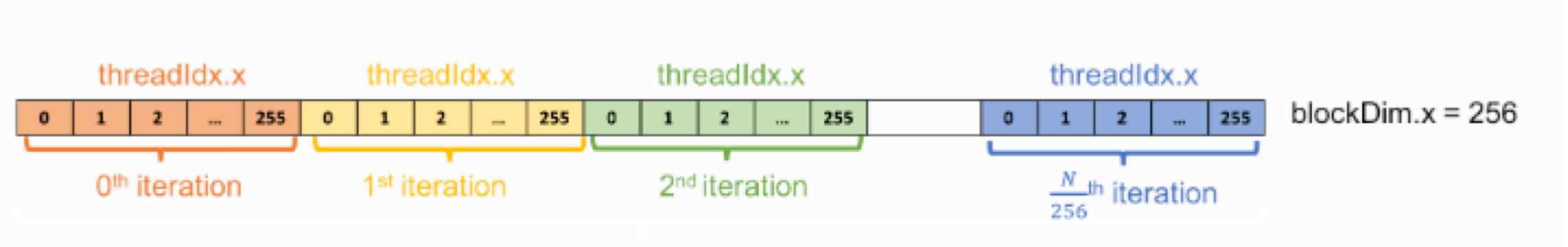
```
vector_add <<< 1 , 256 >>> (d_out, d_a, d_b, N);
```

- threadIdx.x contains the index of the thread within the block
- blockDim.x contains the size of thread block (number of threads in the thread block).
- For the vector_add() configuration, the value of threadIdx.x ranges from 0 to 255 and the value of blockDim.x is 256.
- Then, what is the maximum number of threads per block???

Parallelizing vector addition using multithread

```
vector_add <<< 1 , 256 >>> (d_out, d_a, d_b, N);
```

Parallelizing vector addition using multithread



- For the k-th thread, the loop starts from k-th element and iterates through the array with a loop stride of 256.
- For example, in the 0-th iteration, the k-th thread computes the addition of k-th element.
- In the next iteration, the k-th thread computes the addition of (k+256)-th element, and so on.

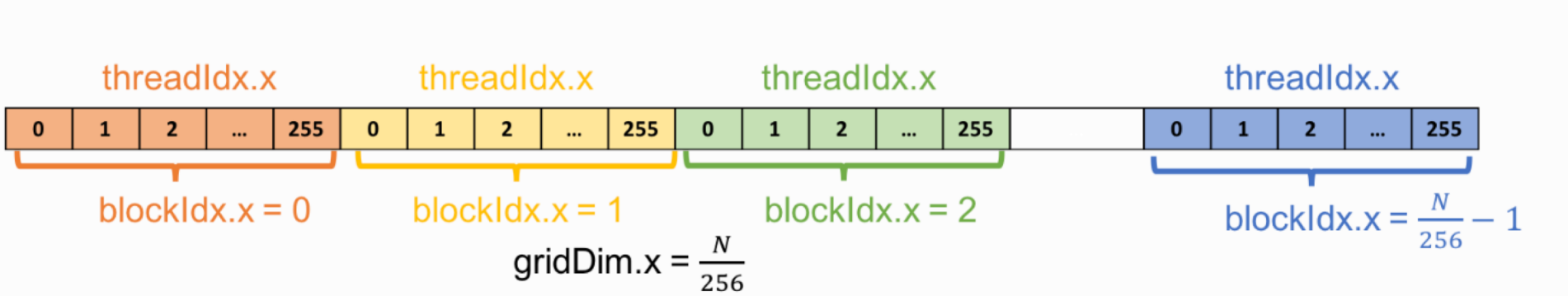
vector_add_thread.cu: You complete this using Colab

```
 8 __global__ void vector_add(float *out, float *a, float *b, int n) {
 9     int index = threadIdx.x;
10     int stride = blockDim.x;
11
12     for (int i=index; i<n; i+=stride) {
13         out[i] = a[i] + b[i];
14     }
15 }
```

```
42     // call function
43     vector_add<<<1,256>>>(d_out, d_a, d_b, N);
44
```

```
aigrad1@dgx-v100-n1:/mnt/aigrad1/Lab/CUDA$ ./vector_add_thread
out[0] = 3.000000
PASSED
```

Adding more threads blocks

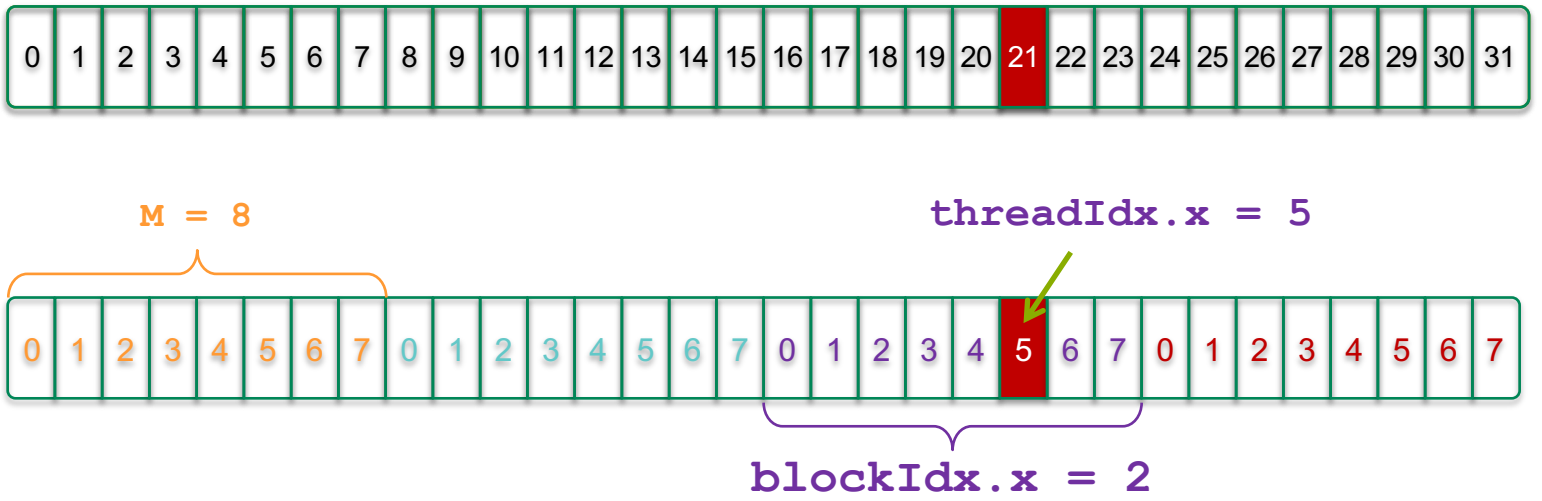


- We will use two of them: `blockIdx.x` and `gridDim.x`.
- `blockIdx.x` contains the index of the block with in the grid
- `gridDim.x` contains the size of the grid
- With 256 threads per thread block, we need at least N/256 thread blocks to have a total of Nthreads. To assign a thread to a specific element, we need to know a unique index for each thread. Such index can be computed as follow

```
int tid = blockIdx.x * M + threadIdx.x;
```

Indexing Arrays: Example

- Which thread will operate on the red element?




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

Vector Addition with Blocks and Threads

```
#define N 512
#define MAX_ERR 1e-6

__global__ void vector_add(float *out, float *a, float *b, int n) {
    int index = threadIdx.x + blockIdx.x * blockDim.x;
    out[index] = a[index] + b[index];
}
```

```
// call function
vector_add<<<N/256, 256>>>(d_out, d_a, d_b, N);
```

```
aigrad1@dgx-v100-n1:/mnt/aigrad1/Lab/CUDA$ ./vector_add_block_thread
out[0] = 3.000000
PASSED
```

Vector Addition with Blocks and Threads

```
#define N 10000000
#define MAX_ERR 1e-6

__global__ void vector_add(float *out, float *a, float *b, int n) {
    int index = threadIdx.x + blockIdx.x * blockDim.x;
    out[index] = a[index] + b[index];
}
```

```
// call function
vector_add<<<N/256, 256>>>(d_out, d_a, d_b, N);
```

```
aigrad1@dgx-v100-n1:/mnt/aigrad1/Lab/CUDA$ ./vector_add_block_thread
vector_add_block_thread: vector_add_block_thread.cu:46: int main(): Assertion `fabs(out[i] -
a[i] - b[i]) < MAX_ERR' failed.
Aborted (core dumped)
```

- Error!! Why??

Vector Addition with Blocks and Threads

- N = 10,000,000
- blockDim.x = 256
- How many blocks? Any remainder?
  - gridDim.x = (10,000,000)/256 = 39,062
  - Remainder = 128
- Therefore, you need to consider this remainder!

Vector Addition with Blocks and Threads

- Solution 1: The updated kernel also sets stride to the total number of threads in the grid (blockDim.x * gridDim.x). This type of loop in a CUDA kernel is often called a *grid-stride loop*.

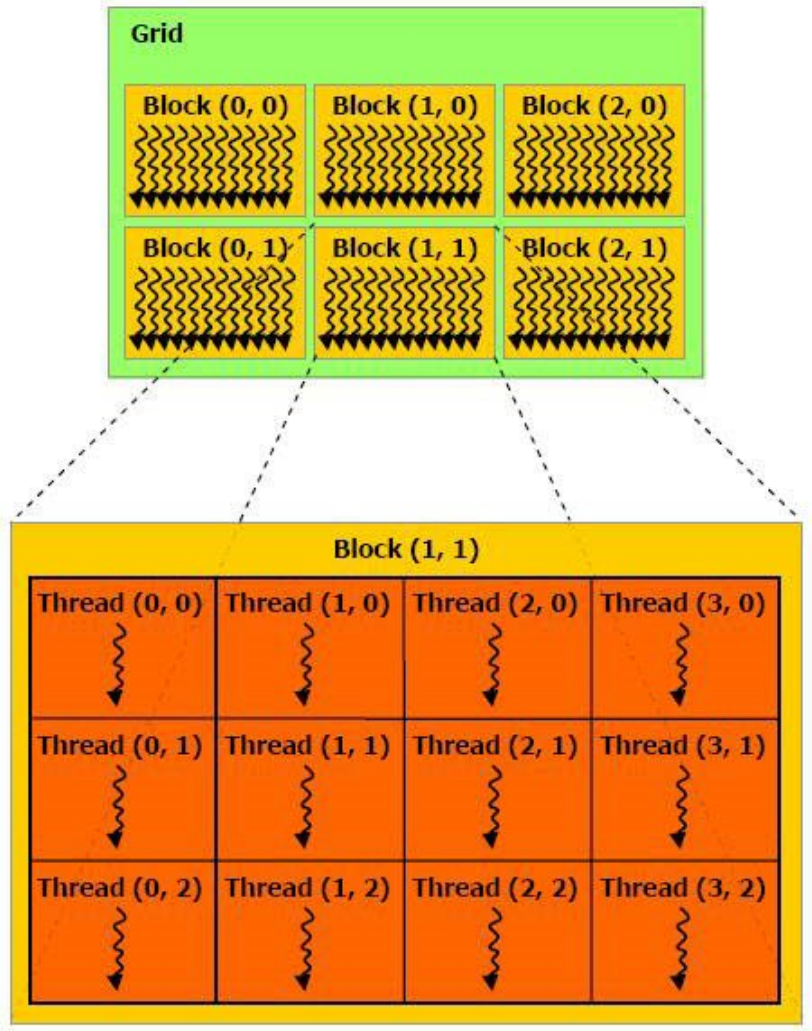
```
// get current thread's id
int index = threadIdx.x + blockIdx.x * blockDim.x;

// The updated kernel also sets stride to the total number of threads
int stride = blockDim.x * gridDim.x;
for (int i = index; i < n; i += stride)
    out[i] = a[i] + b[i];
```

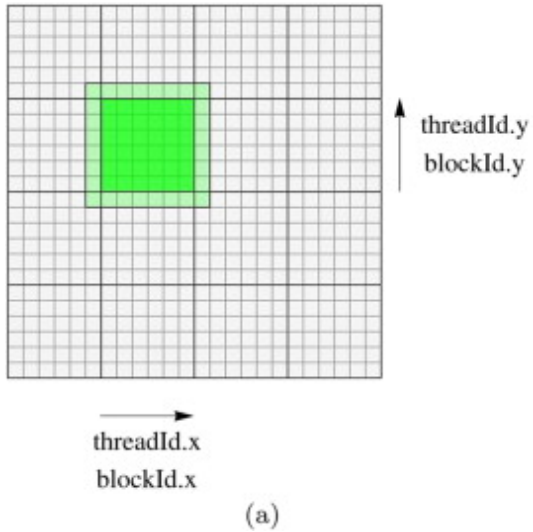
Vector Addition with Blocks and Threads

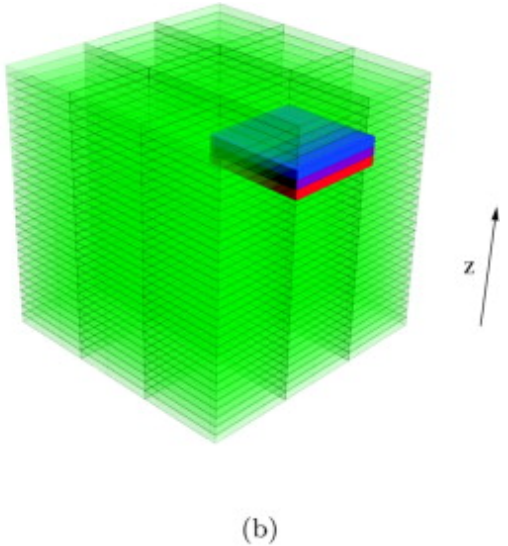
- Solution 2: The updated kernel is almost same as solution. Consider n and the total number of threads in the grid (blockDim.x * gridDim.x).

```
// get current thread's id
int index = threadIdx.x + blockIdx.x * blockDim.x;

// wile this thread is dealing with a valid index
while (index < n) {
    out[index] = a[index] + b[index];
    index += blockDim.x * gridDim.x;
}
```

Grid/Block/Thread Visualized

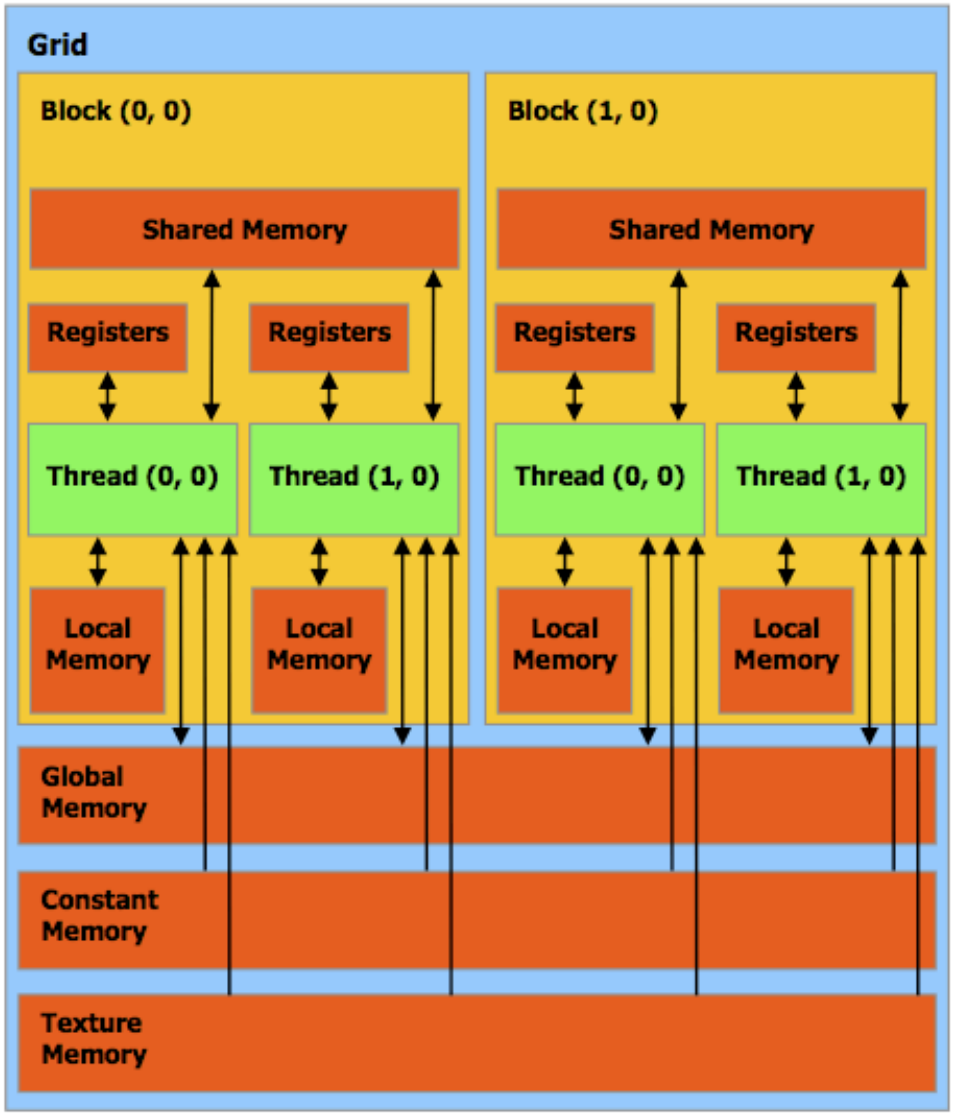






Thread Block Organization

- Keywords you MUST know to code in CUDA:
  - Thread - Distributed by the CUDA runtime (threadIdx)
  - Block - A user defined group of 1 ~ 1024 threads (blockIdx)
  - Grid - A group of one or more blocks. A grid is created for each CUDA kernel function called.
- Imagine thread organization as an array of thread indices



Careful! Max threads limit!

NVIDIA V100

```
Maximum number of threads per block:           1024
Max dimension size of a thread block (x,y,z): (1024, 1024, 64)
```

- The maximum number of threads in the block is limited to 1024. This is the product of whatever your threadblock dimensions are (xyz). For example (32,32,1) creates a block of 1024 threads. (33,32,1) is not legal, since 33*32*1 > 1024.
- The maximum x-dimension is 1024. (1024,1,1) is legal. (1025,1,1) is not legal.
- The maximum y-dimension is 1024. (1,1024,1) is legal. (1,1025,1) is not legal.
- The maximum z-dimension is 64. (1,1,64) is legal. (2,2,64) is also legal. (1,1,65) is not legal.

Define a thread hierarchy using dim3

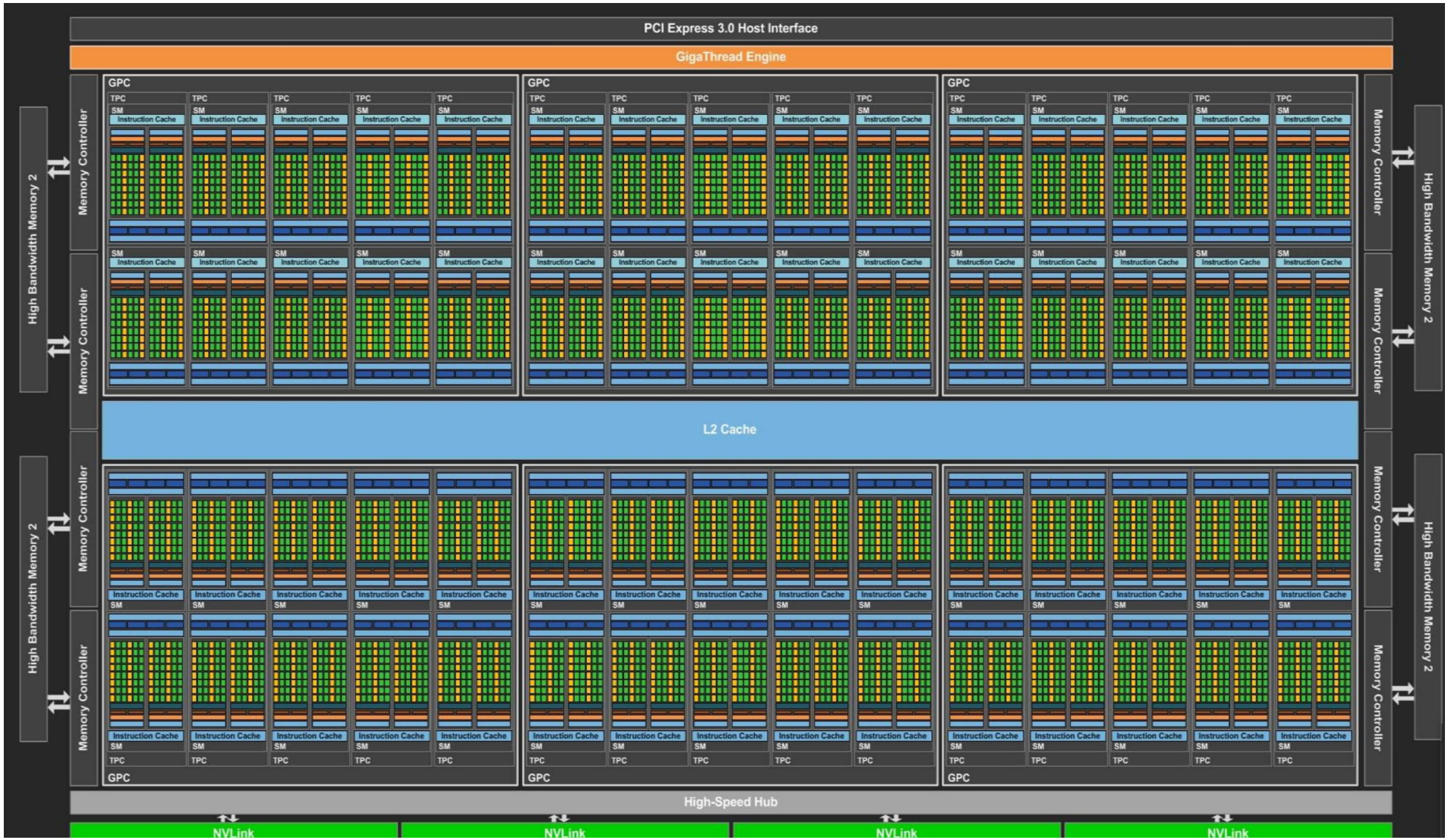
- dim3 var_name (x_dim , y_dim , z_dim)
- dim3 is a struct to define your Grid and Block dimensions.

```
// call function
//vector_add<<<N/256, 256>>>(d_out, d_a, d_b, N);
dim3 dimGrid(N/256, 1, 1);
dim3 dimBlock(256, 1, 1);
vector_add<<<dimGrid, dimBlock>>>(d_out, d_a, d_b, N);
```

Why do we use 256 threads?

- CUDA GPUs run kernels using blocks of threads that are a multiple of 32 in size, so 256 threads is a reasonable size to choose.
- Warp
  - 32 Threads
  - Basic execution unit
  - Controlled by the same instructions

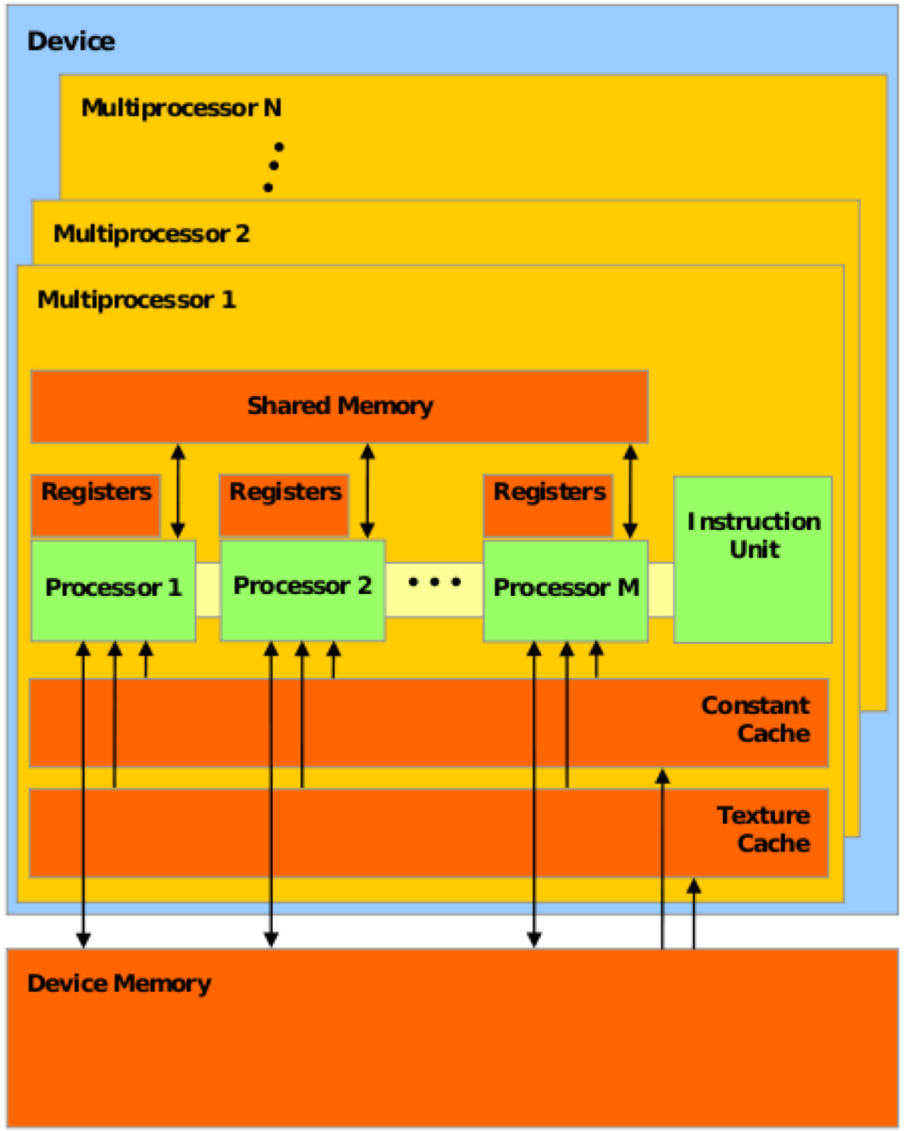
What a modern GPU looks like



Consider your GPU

```
(80) Multiprocessors, ( 64) CUDA Cores/MP:     5120 CUDA Cores
```

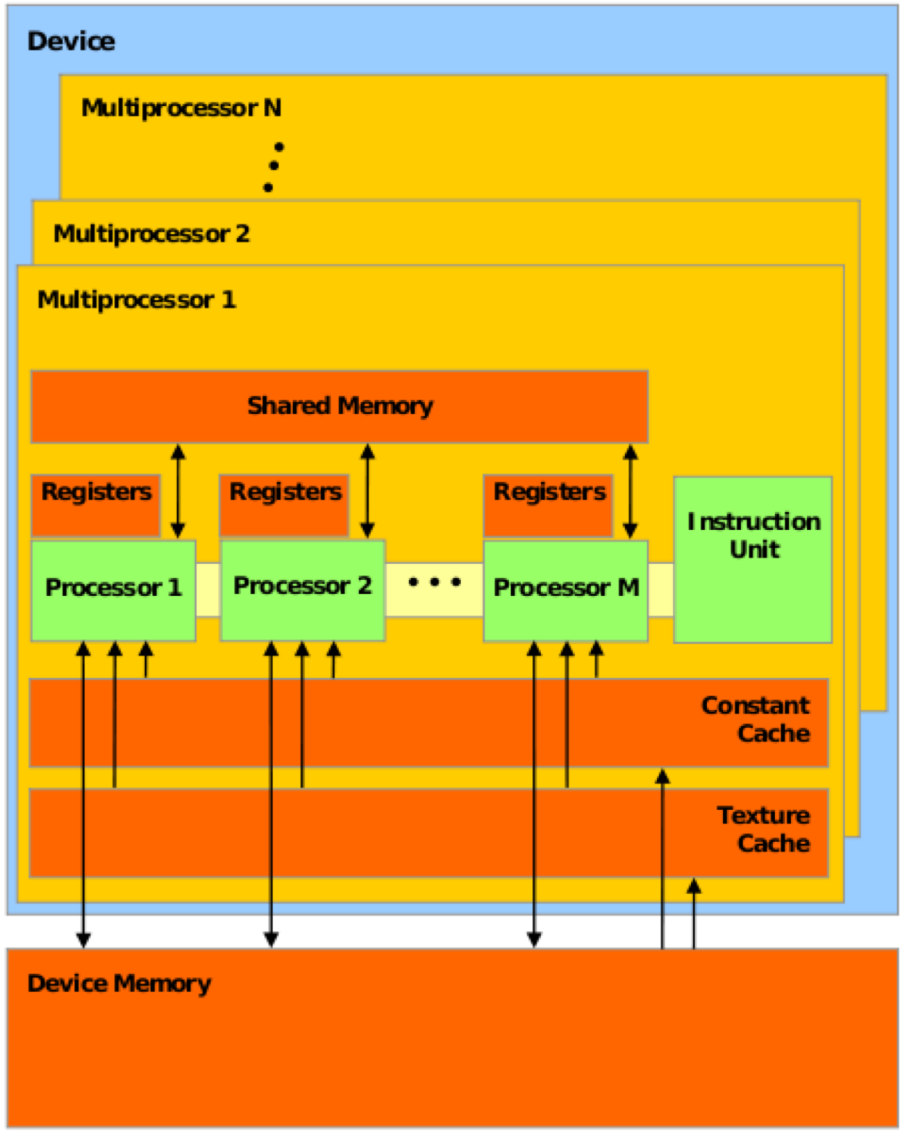
- GPUs have many Streaming Multiprocessors (SMs)
- Each SM has multiple processors but only one instruction unit (each thread shares program counter)
- Groups of processors must run the exact same set of instructions at any given time with in a single SM



Consider your GPU

```
(80) Multiprocessors, ( 64) CUDA Cores/MP:     5120 CUDA Cores
```

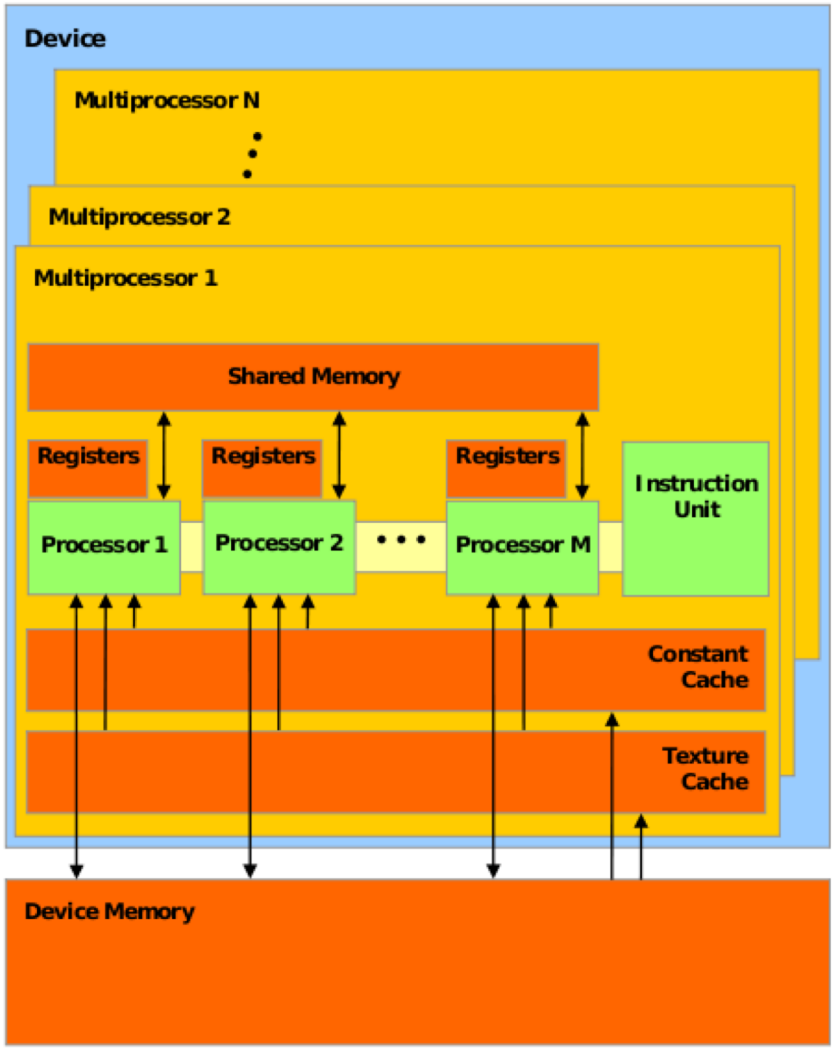
- When a kernel (the thing you define in .cu files) is called, the task is divided up into threads
  - Each thread handles a small portion of the given task
- The threads are divided into a Grid of Blocks
  - Both Grids and Blocks are 3 dimensional
  - e.g.
  - dim3 dimBlock(8, 8, 8);
  - dim3 dimGrid(100, 100, 1);



Consider your GPU

```
Max dimension size of a grid size    (x,y,z): (2147483647, 65535, 65535)
```

- Maximum number of blocks per grid is usually 65535 (but x is very large in some GPUs)
- If you go over either of these numbers your GPU will just give up or output garbage data
- Much of GPU programming is dealing with this kind of hardware limitations! Get used to it
- This limitation also means that your Kernel must compensate for the fact that you may not have enough threads to individually allocate to your data points



GPU CUDA Type of Memory

High-Performance Computing

Summer 2021 at GIST

Tae-Hyuk (Ted) Ahn
Department of Computer Science
Program of Bioinformatics and Computational Biology
Saint Louis University

SAINT LOUIS
UNIVERSITY
EST. 1818

Latency & Throughput, Properties

- Latency is the delay caused by the physical speed of the hardware
- Throughput is the maximum rate of production/processing
- CPU = low latency, low throughput
  - CPU clock = 3 GHz (3 clocks/ns)
  - CPU main memory latency: ~100+ ns
  - CPU arithmetic instruction latency: ~1+ ns
- GPU = high latency, high throughput
  - GPU clock = 1 GHz (1 clock/ns)
  - GPU main memory latency: ~300+ ns
  - GPU arithmetic instruction latency: ~10+ ns
- Above numbers were for Kepler GPUs (e.g. GTX 700 series)

Compute & IO Throughput

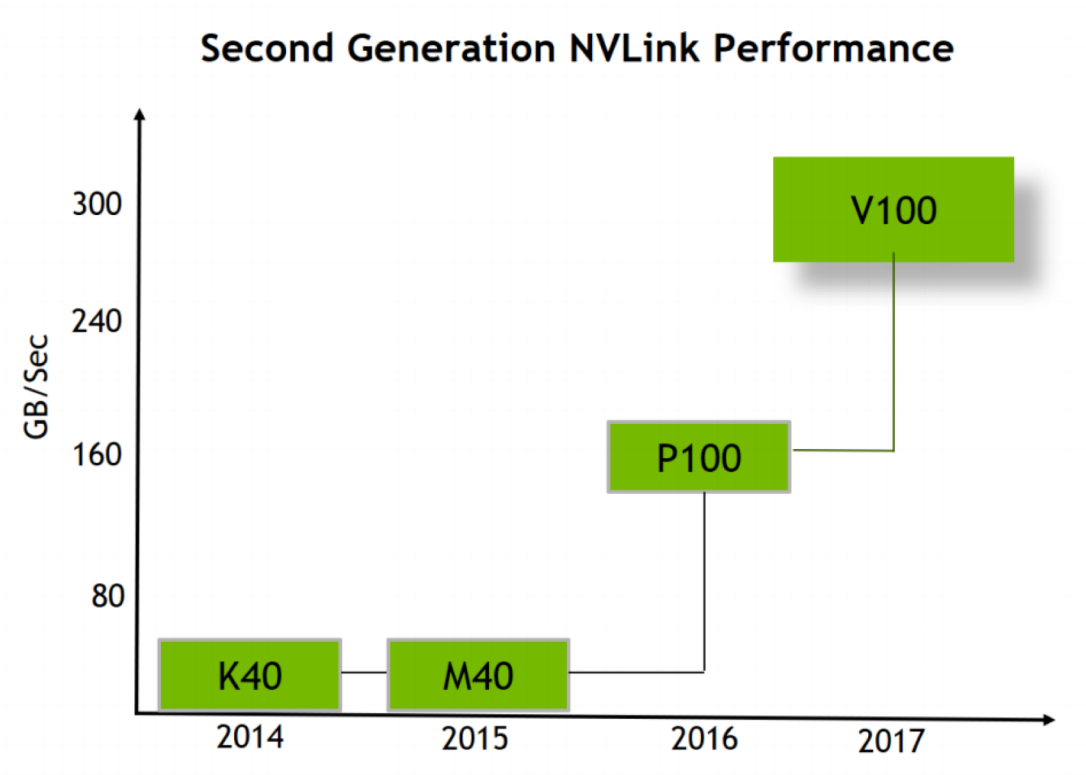


Figure 14. Second Generation NVLink Performance

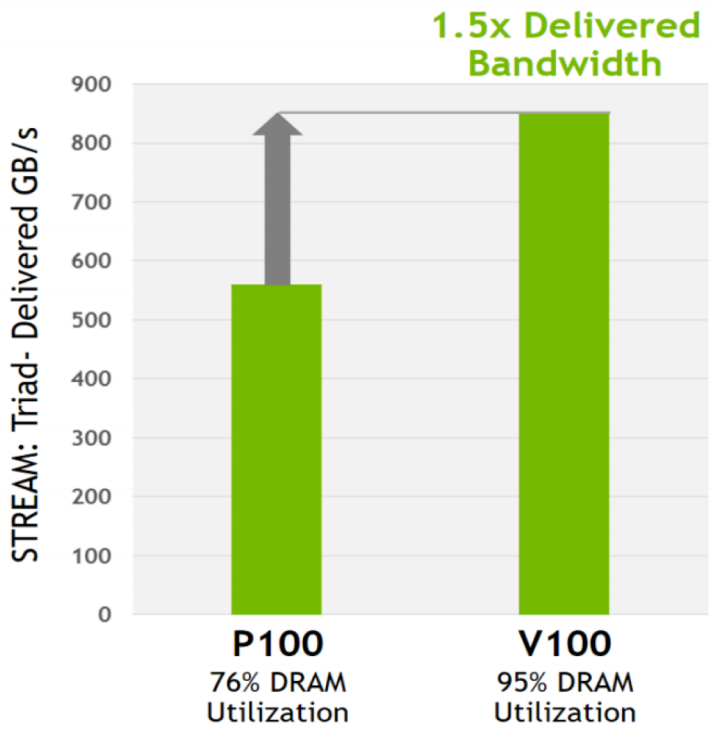


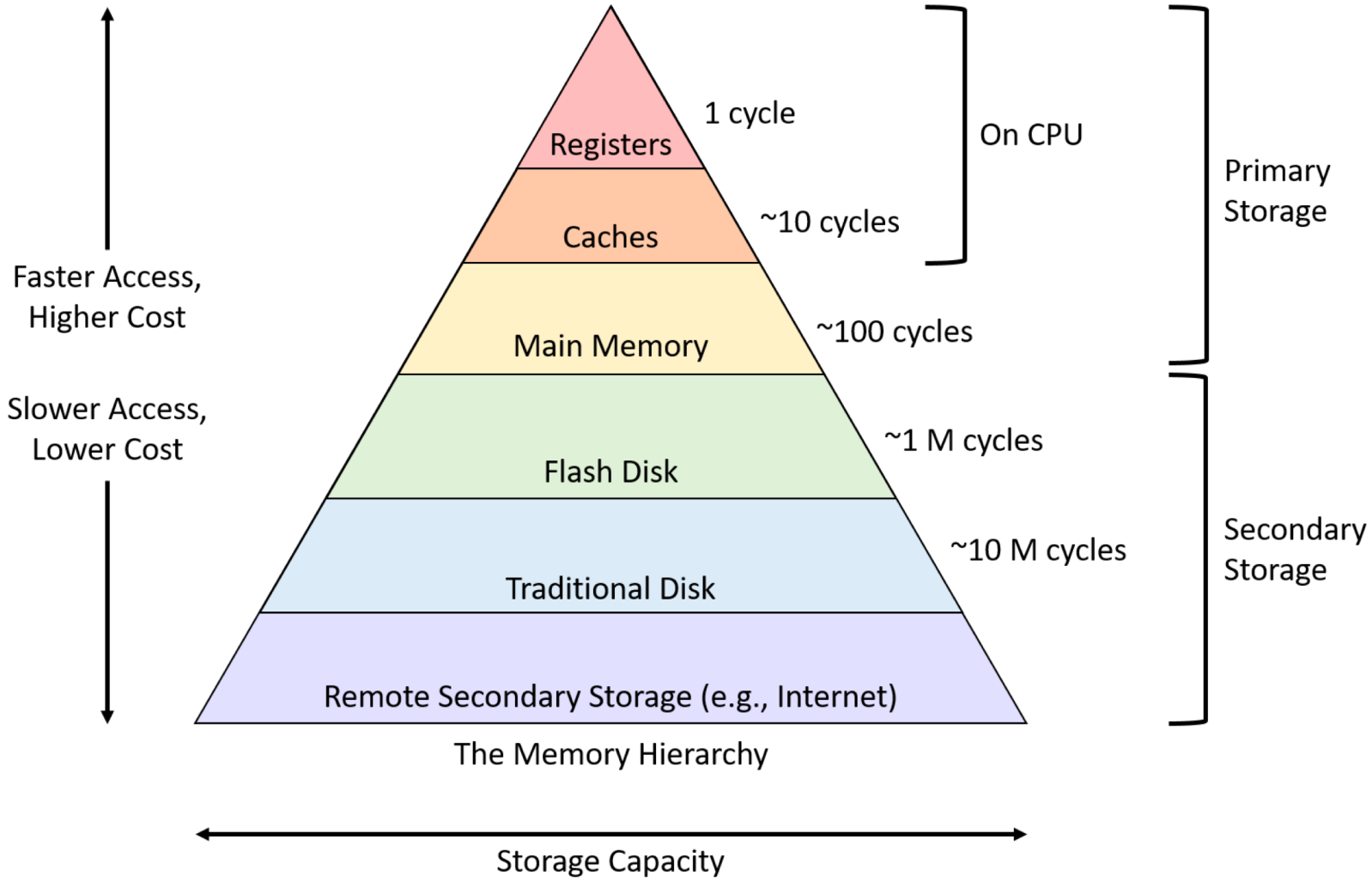
Figure 15. HBM2 Memory Speedup on V100 vs P100

GPU is very IO limited! IO is very often the throughput bottleneck, so its important to be smart about IO.

https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf

Memory Hierarchy

- The memory hierarchy
  - As we explore modern computer storage, a common pattern emerges: devices with higher capacities offer lower performance.
  - Said another way, systems use devices that are fast and devices that store a large amount of data, but no single device does both.
  - This trade-off between performance and capacity is known as the memory hierarchy, and the figure depicts the hierarchy visually.

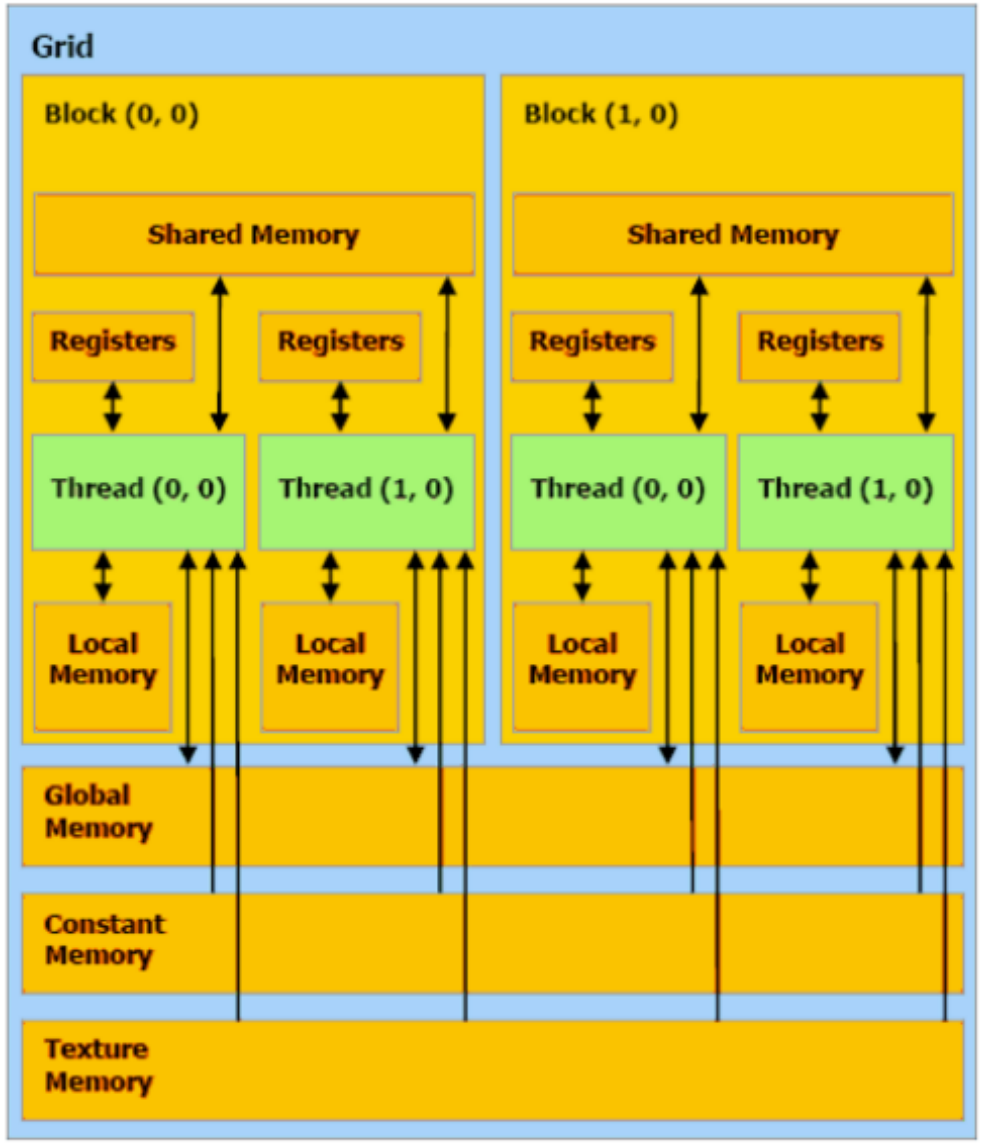


The Memory Hierarchy

https://diveintosystems.org/antora/diveintosystems/1.0/MemHierarchy/mem_hierarchy.html

GPU Memory Hierarchy

- Per Thread local memory
  - Registers
  - Local Memory
- Per Block shared memory
  - Shared Memory
- Per Grid global memory
  - Global Memory
  - Constant Memory(read only)
  - Texture Memory (read only)

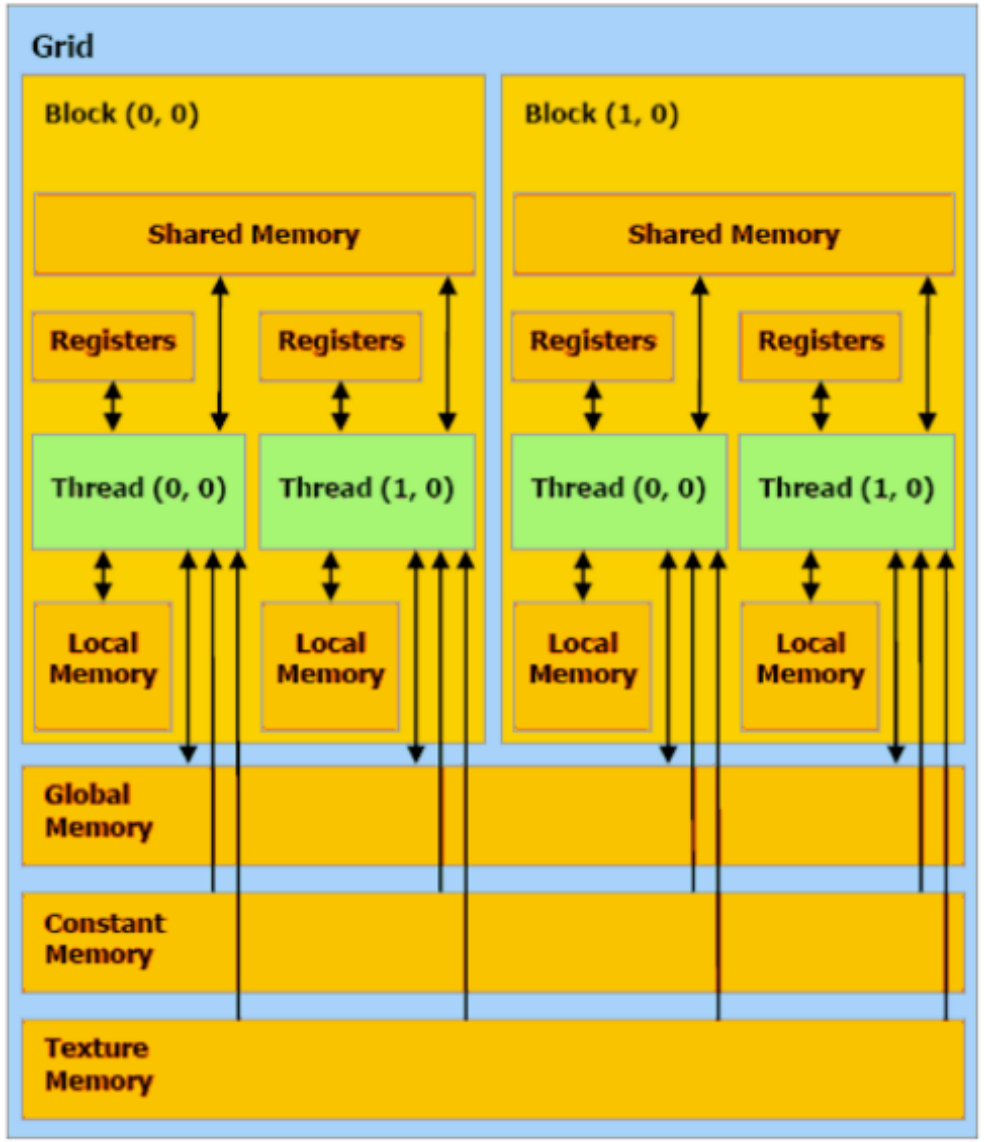


GPU Memory

- In summary,
  - Registers: The fastest form of memory on the multi-processor. Is only accessible by the thread. Has the lifetime of the thread.
  - Local memory: Resides in global memory and can be 150x slower than register or shared memory. Is only accessible by the thread. Has the lifetime of the thread. Off chip memory: Outside of a SM, reside in DRAM. Slow, but large space. Hold local variables.
  - Shared Memory: Can be as fast as a register when there are no bank conflicts or when reading from the same address. Fast, but small on chip memory. Accessible by any thread of the block from which it was created. Has the lifetime of the block: Shared by all threads in a block, so all thread in a block access the same data.
  - Global memory: Potentially 150x slower than register or shared memory -- watch out for uncoalesced reads and writes. Accessible from either the host or device. Has the lifetime of the application—that is, it persistent between kernel launches. The slowest, but largest off chip memory. Shared by all threads in a grid.
  - Constant memory: Constant memory is global memory with a special cache. Read only memory: declared with global scope, initialized by the host. Small, but Fast: reside in DRAM, but has dedicated on chip cache (64 KB per SM).
  - Texture memory: Complicated and only marginally useful for general purpose computation. Read only memory. Small, but Fast. Optimized for 2D spatial locality. Support hardware filtering such as fast interpolation.

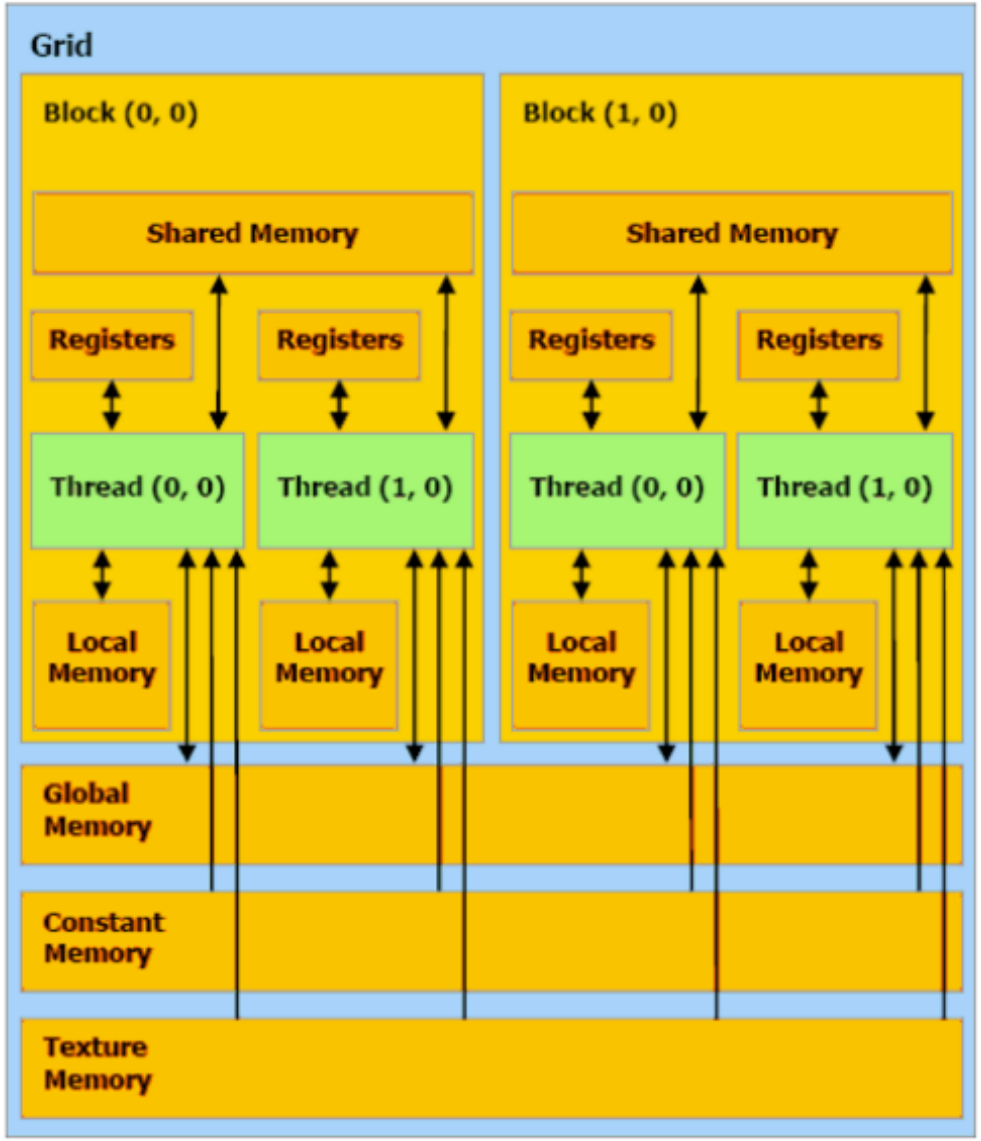
Registers

- A Register is a piece of memory used directly by the processor.
- The fastest (about 10x faster than shared memory), but smallest memory.
- There are tens of thousands of registers in each SM.
  - E.g., 8k 64k 32 bit registers per block
- For variables declared in a kernel
  - E.g., float x; (duh...)
- Automatically set by compilers



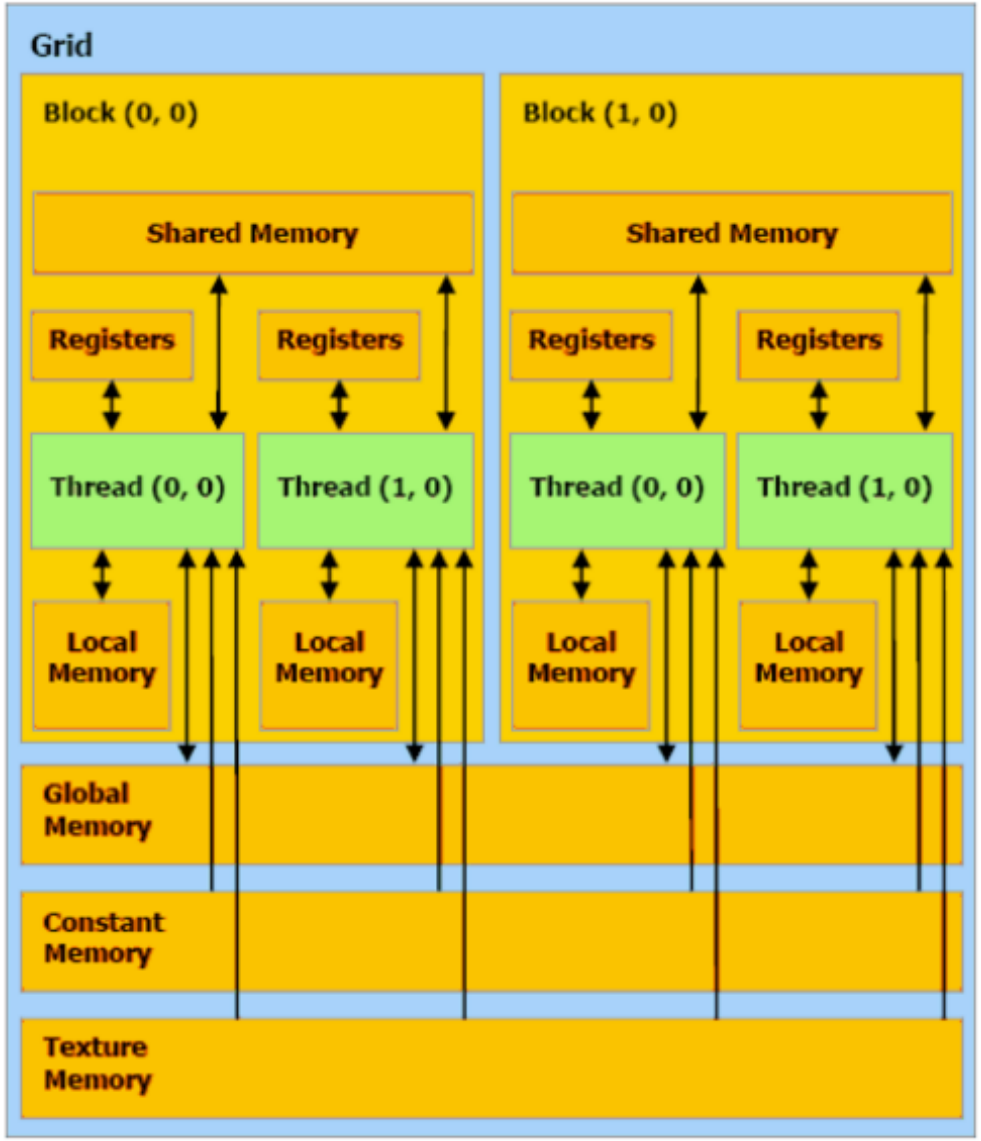
Local Memory

- Local memory is everything on the stack that can't fit in registers
  - Local arrays
  - Large local structure or arrays consume too much register space
- The scope of local memory is just the thread.
- Local memory is stored in global memory
- Much slower than registers
  - Off-chip memory
  - Outside of a SM, reside in DRAM



Shared Memory

- Very fast memory located in the SM
- Same hardware as L1 cache
  - ~5ns of latency
- 16 ~ 96 KB per block/SM (varies per GPU)
- Scope of shared memory is the block
  - Shared by all threads in a block
  - All thread in a block access the same data



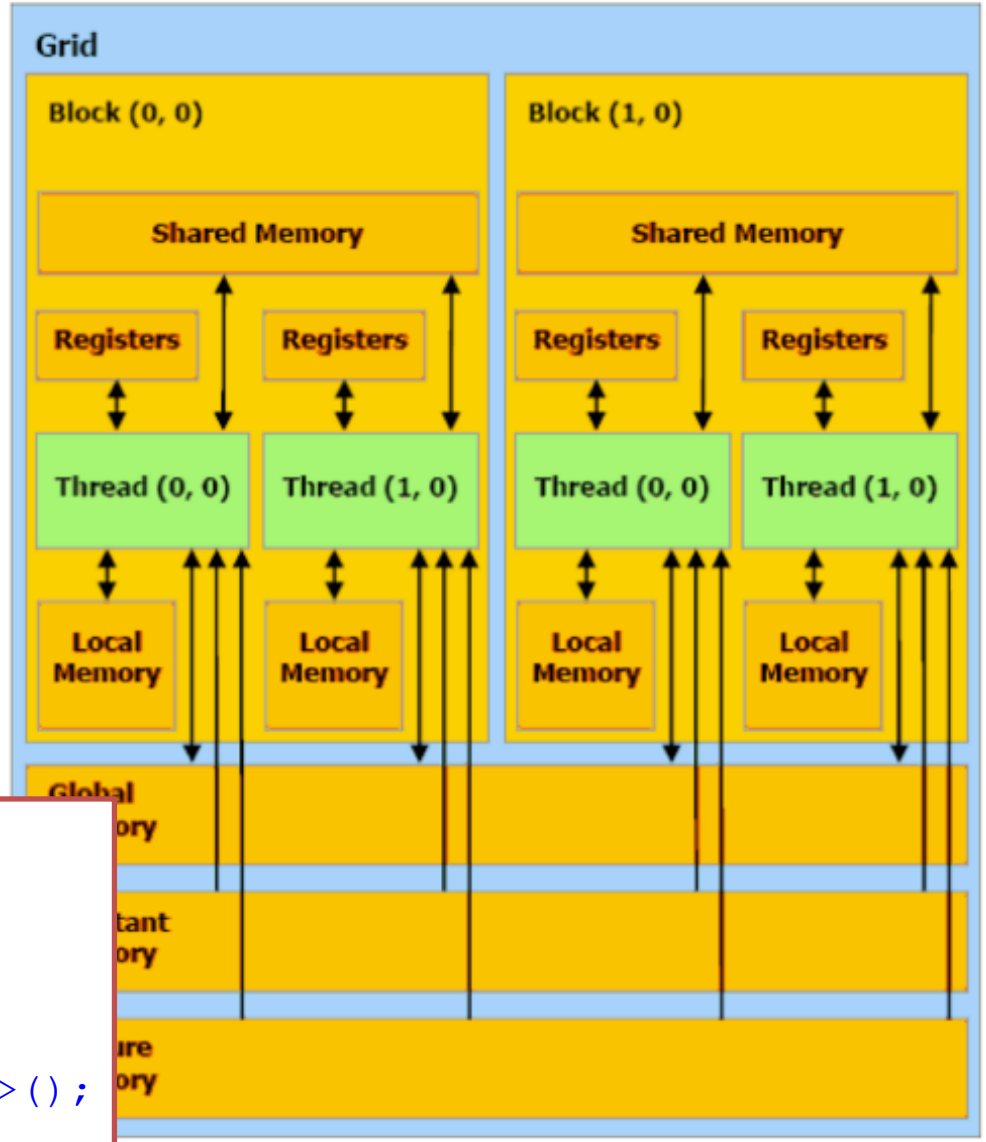
Shared Memory Syntax

- Can allocate shared memory statically (size known at compile time) or dynamically (size not known until runtime)
- Static allocation
  - Declared in the kernel, nothing in host code

```
__global__ void kernel(void)
{
    __shared__ float sharedData [1024];
}
```

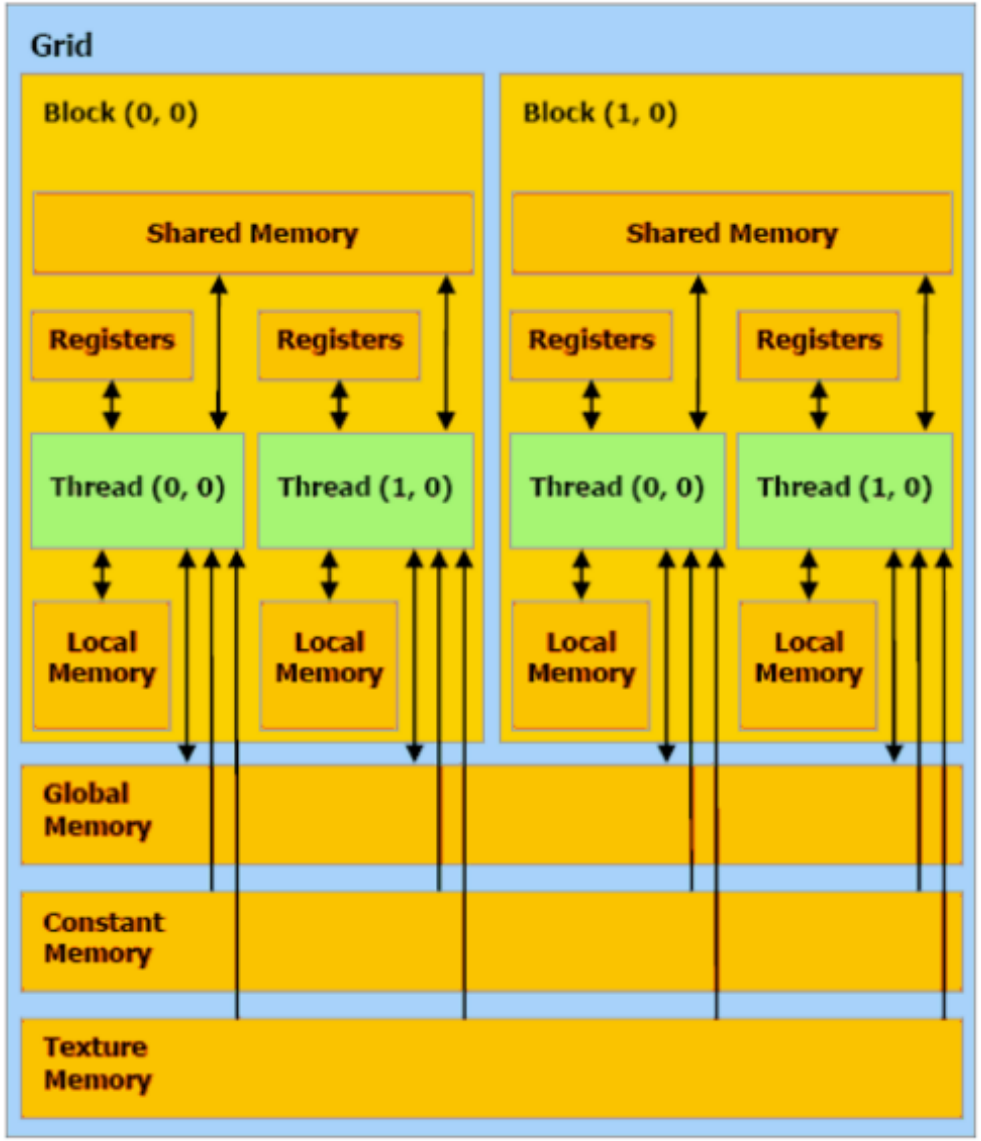
- Dynamic allocation

```
extern __shared__ float sharedData [];
__global__ void kernel(void)
{ ... }

int main () {
    kernel <<<gridDim, blockDim, sizeof(float)*1024>>>();
}
```



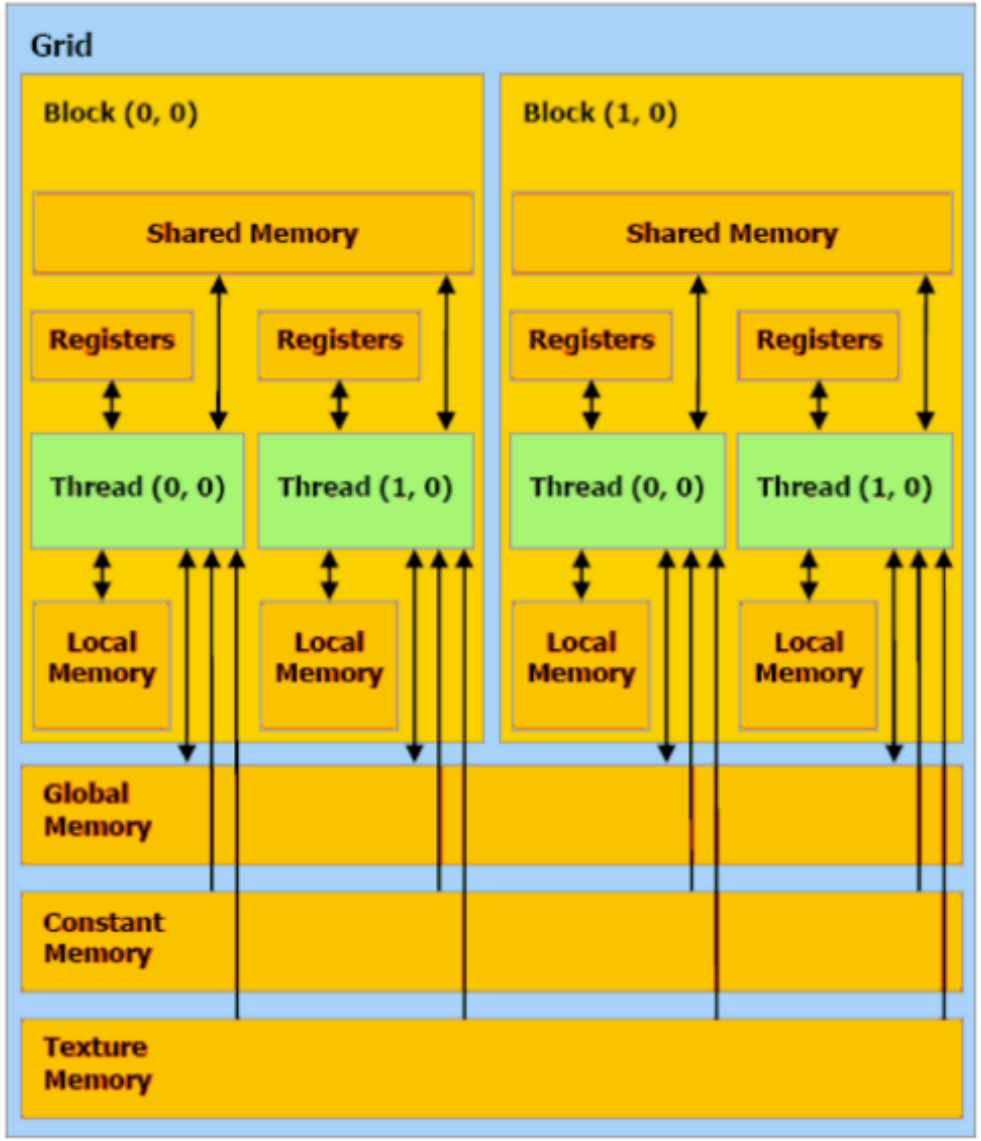
Global Memory

- The slowest, but largest off chip memoryStatic allocation
- If data doesn't fit into global memory, you are going to have process it in chunks that do fit in global memory.
- GPUs have 2 - 32GB of global memory.
- Shared by all threads in a grid



Constant Memory

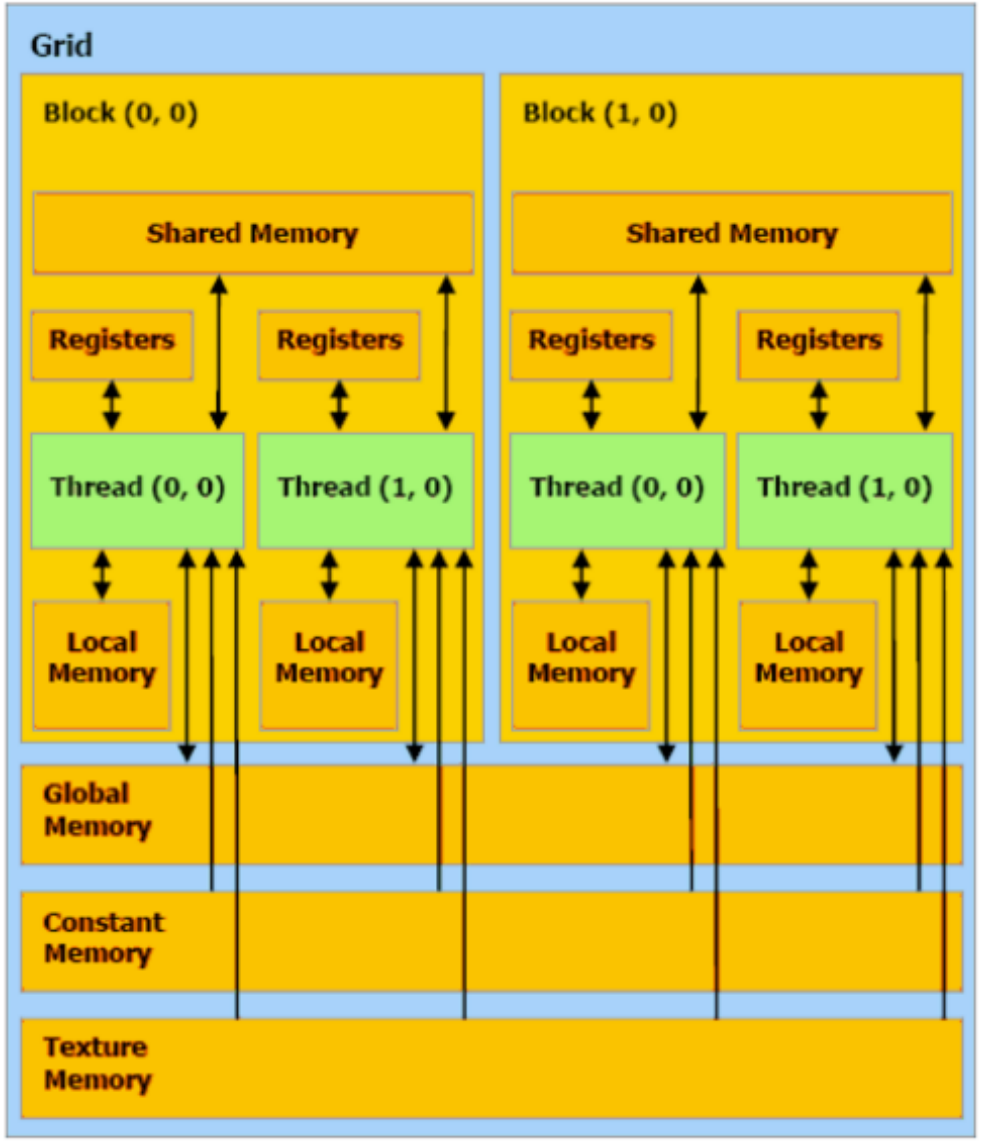
- Constant memory is global memory with a special cache
- Used for constants that cannot be compiled into program
- Constants must be set from host before running kernel.
- ~64KB for user, ~64KB for compiler
- Kernel arguments are passed through constant memory



Constant Memory Syntax

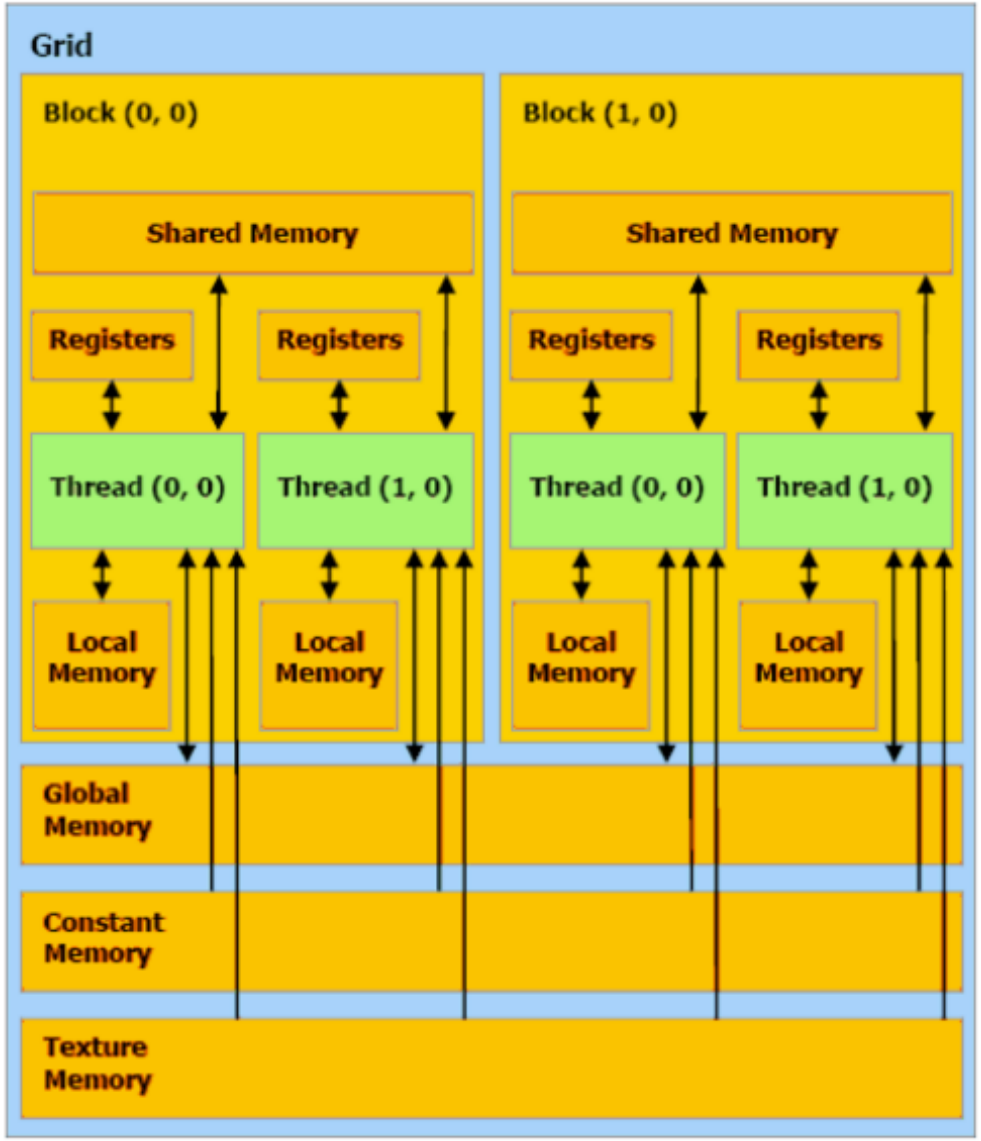
- Read only memory
- Declared with global scope, initialized by the host.
  - outside of kernel, at top level of program

```
__constant__ int constData[1024];

int main( void
{
    int data[1024] = { 0 };
    cudaMemcpyToSymbol(constData, data, sizeof(int)*1024);
    ...
    kernel <<<gridDim , blockDim>>> ();
}
```



Texture Memory

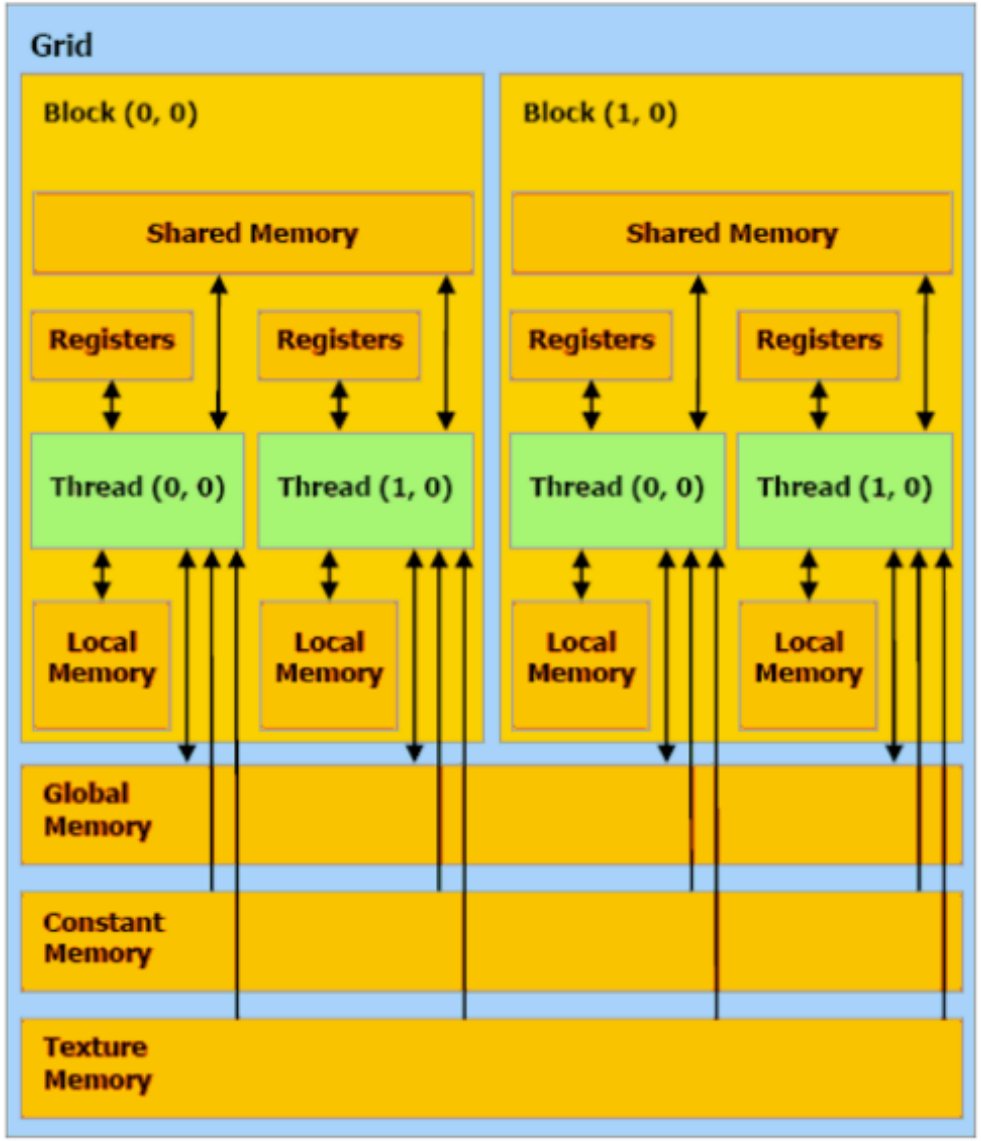
- Complicated and only marginally useful for general purpose computation
- Useful characteristics:
  - 2D or 3D data locality for caching purposes through “CUDA arrays”. Goes into special texture cache.
  - fast interpolation on 1D, 2D, or 3D array
  - converting integers to “unitized” floating point numbers
- Use cases:
  - Read input data through texture cache and CUDA array to take advantage of spatial caching. This is the most common use case.
  - Take advantage of numerical texture capabilities.
  - Interaction with OpenGL and general computer graphics



GPU Mem Hierarchy

Remember

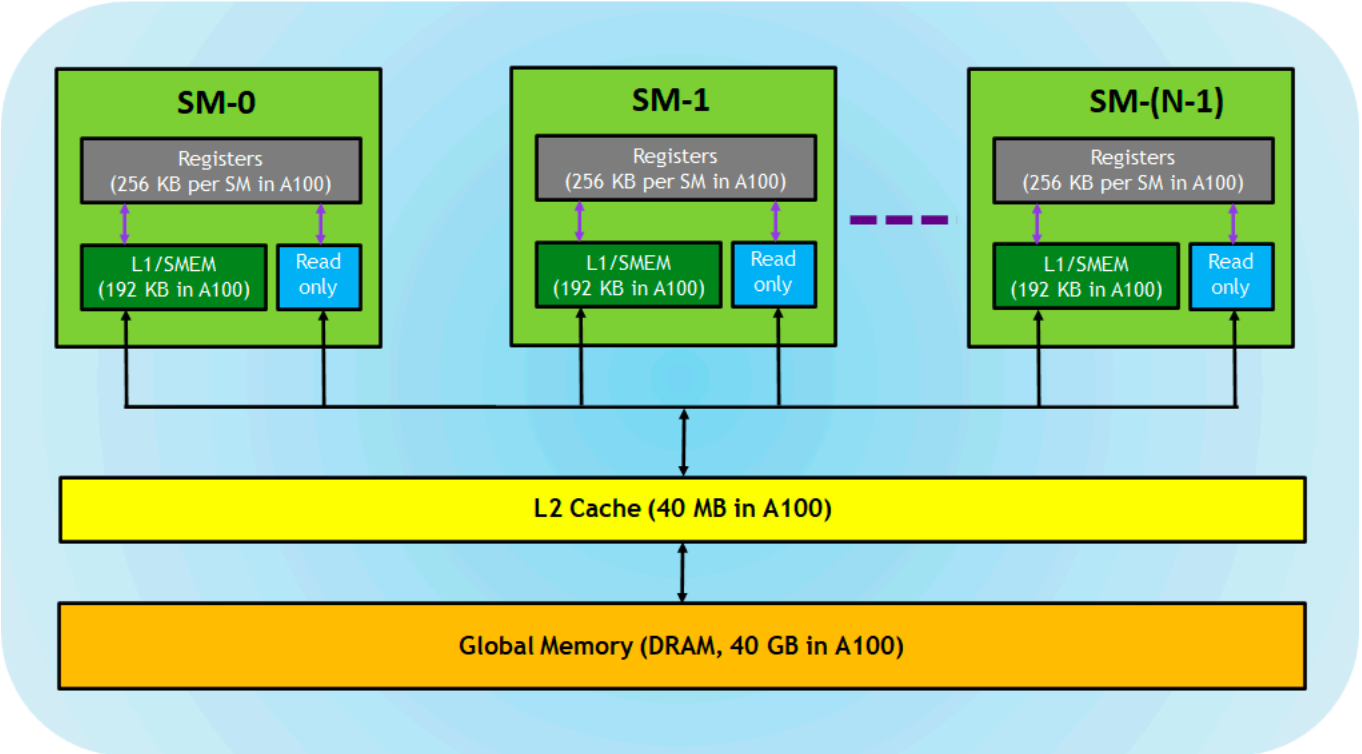
- Thread
  - Registers
  - Local Memory
- Block
  - Shared Memory
- Host Allocation
  - Global Memory
  - Constant Memory(read only)
  - Texture Memory (read only)



L1, L2, and L3 GPU Caches

In general,

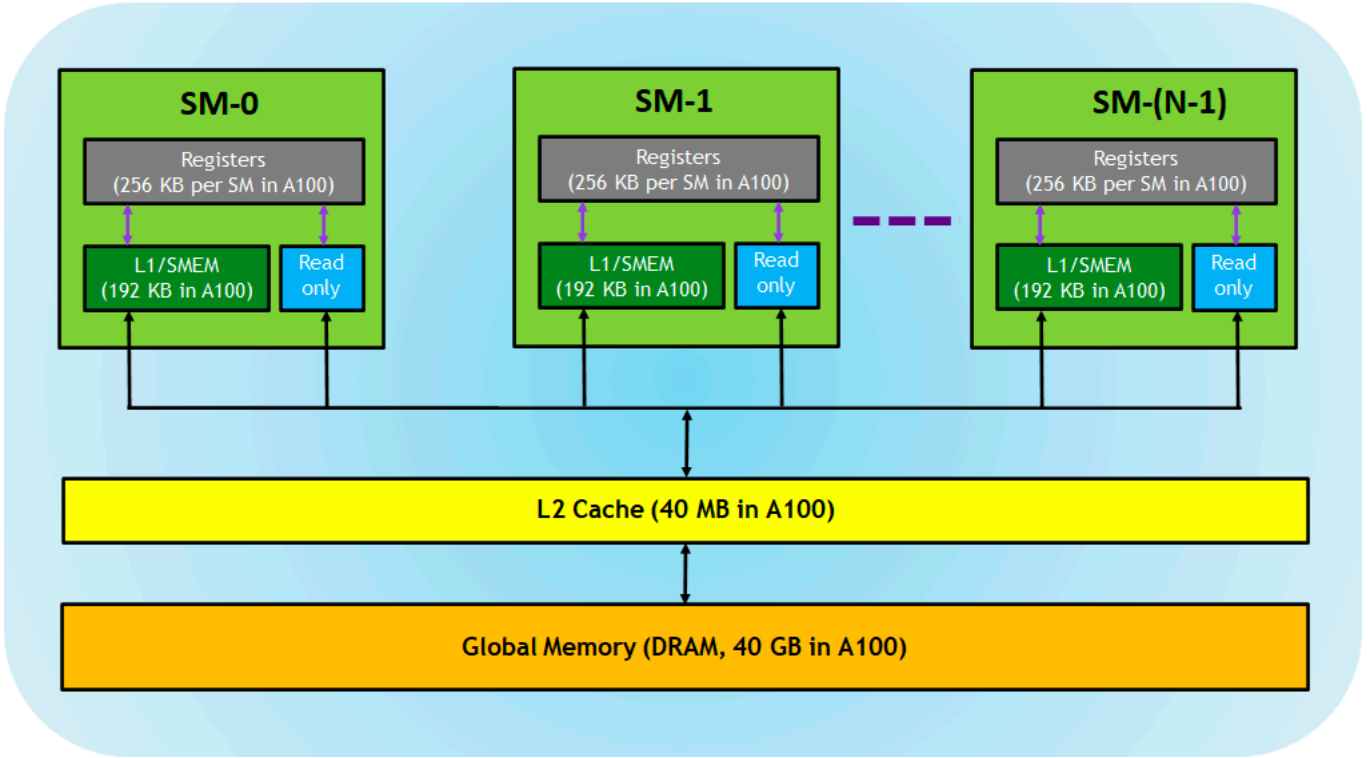
- L1 cache
  - Each SM has its own L1 cache
- L2 cache
  - Shared by all SM, so every thread in every CUDA block can access this memory.
  - Caches all global & local memory accesses
  - The NVIDIA A100 GPU has increased the L2 cache size to 40 MB as compared to 6 MB in V100 GPUs.
- L3 cache (not common)



https://developer.nvidia.com/blog/cuda-refresher-cuda-programming-model/

Physical Hardware Aspects

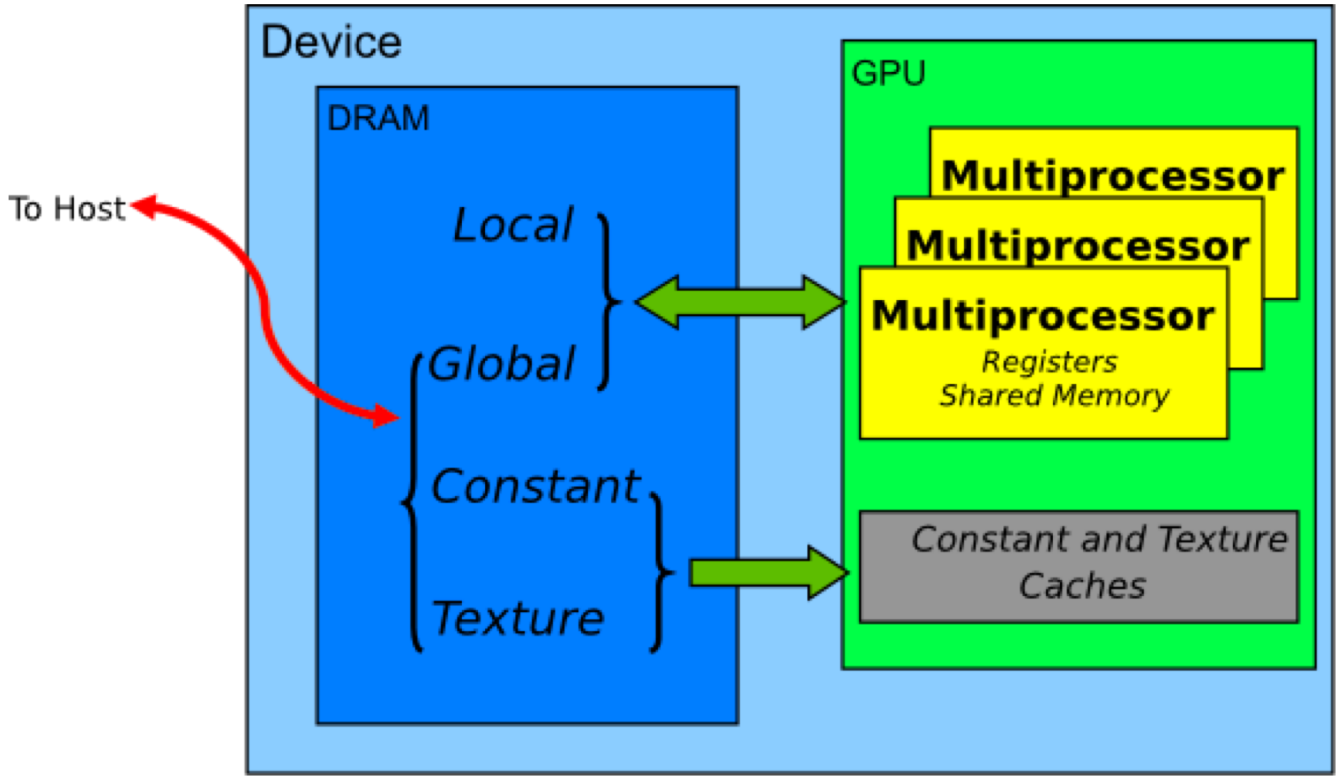
- Each SM
  - Registers
  - Shared Memory
  - Constant and Texture Memory
- Outside SM
  - Local Memory
  - Global Memory



https://developer.nvidia.com/blog/cuda-refresher-cuda-programming-model/

Summary and Recap: Memory spaces on a CUDA device

- Of these different memory spaces, global memory is the most plentiful; see Features and Technical Specifications of the CUDA C++ Programming Guide for the amounts of memory available in each memory space at each compute capability level.
- Global, local, and texture memory have the greatest access latency, followed by constant memory, shared memory, and the register file.



CUDA Memory Model & Performance

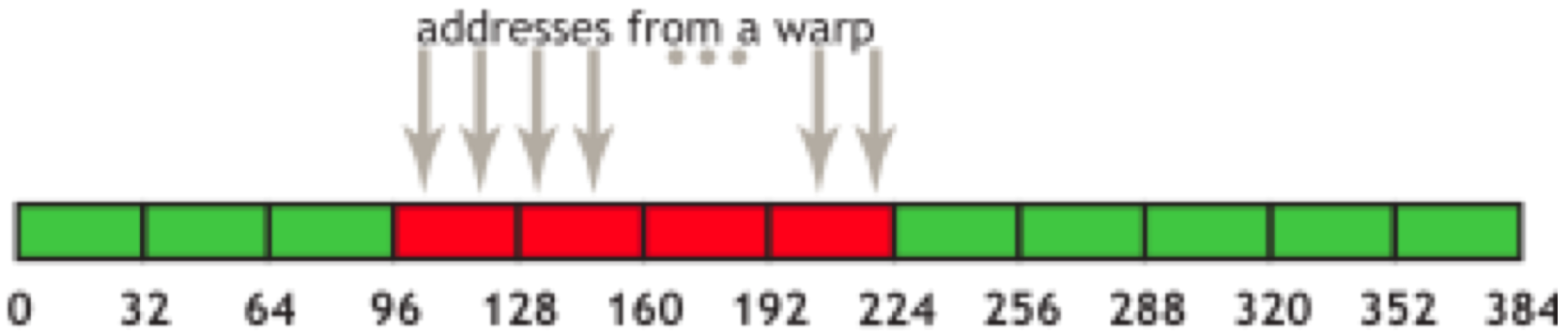
- Use appropriate memory space for your computation
  - e.g., Use shared memory for handling block local data
  - e.g., Use constant memory for read only and commonly used data
- Design your kernel and block
  - So that maximizing parallelism while considering the memory usage
- Global memory IO is the slowest form of IO on GPU
  - except for accessing host memory
- Because of this, we want to access global memory as little as possible
- Access patterns that play nicely with GPU hardware are called *coalesced memory accesses.*

Coalesced Access to Global Memory

- A very important performance consideration in programming for CUDA-capable GPU architectures is the coalescing of global memory accesses.
- Global memory loads and stores by threads of a warp are coalesced by the device into as few as possible transactions.
  - GPU cache lines are 128 bytes and are aligned
- **Note:High Priority:** Ensure global memory accesses are coalesced whenever possible.

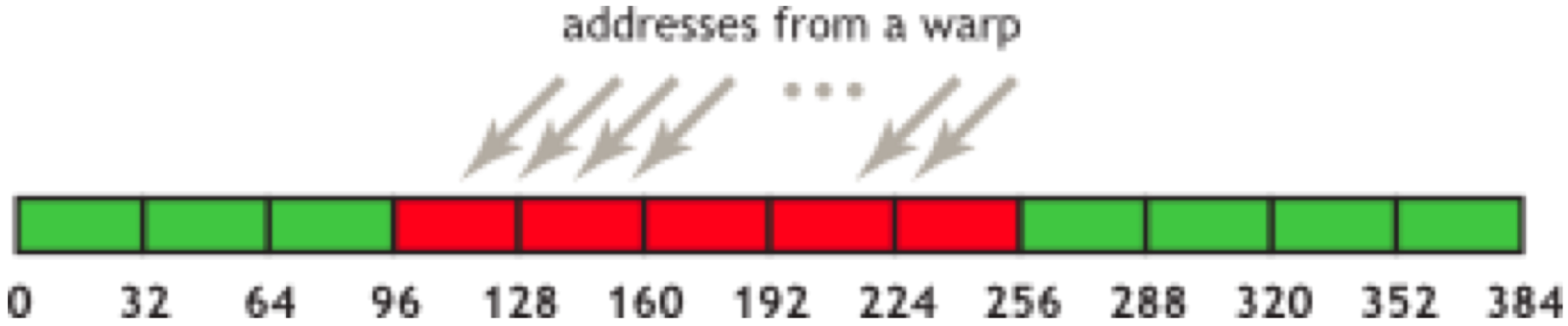
A Simple Access Pattern

- The first and simplest case of coalescing can be achieved by any CUDA-enabled device of compute capability 6.0 or higher: the k-th thread accesses the k-th word in a 32-byte aligned array. Not all threads need to participate.
- For example, if the threads of a warp access adjacent 4-byte words (e.g., adjacent float values), four coalesced 32-byte transactions will service that memory access. Such a pattern is shown in below Figure.
- This access pattern results in four 32-byte transactions, indicated by the red rectangles.



Misaligned Access Pattern

- If sequential threads in a warp access memory that is sequential but not aligned with a 32-byte segment, five 32-byte segments will be requested



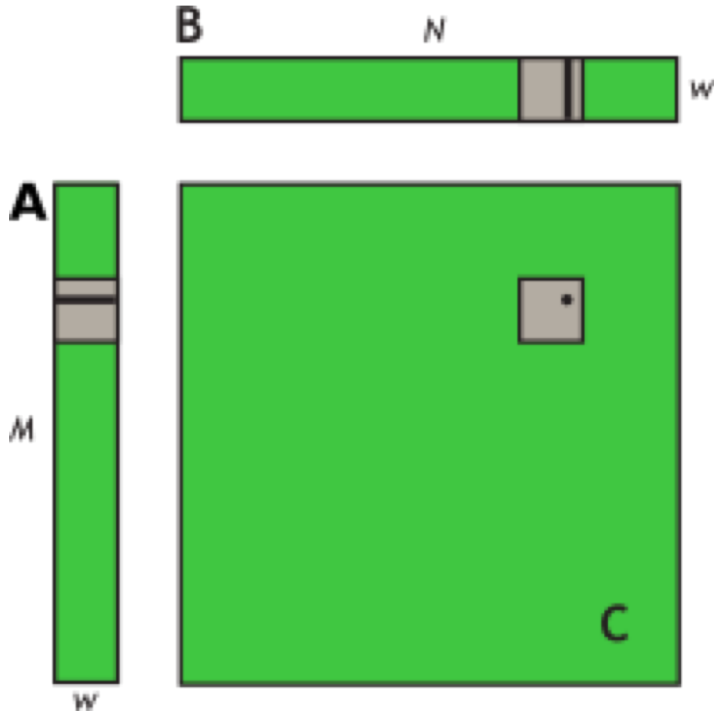
- Memory allocated through the CUDA Runtime API, such as via cudaMalloc(), is guaranteed to be aligned to at least 256 bytes. Therefore, choosing sensible thread block sizes, such as multiples of the warp size (i.e, 32 on current GPUs), facilitates memory accesses by warps that are properly aligned.
- Please read: How to Access Global Memory Efficiently in CUDA C/C++ Kernels

Example: Matrix Multiplication

- https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html#memory-optimizations

Unoptimized matrix multiplication

```
__global__ void simpleMultiply(float *a, float* b, float *c,
                               int N)
{
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;
    float sum = 0.0f;
    for (int i = 0; i < TILE_DIM; i++) {
        sum += a[row*TILE_DIM+i] * b[i*N+col];
    }
    c[row*N+col] = sum;
}
```

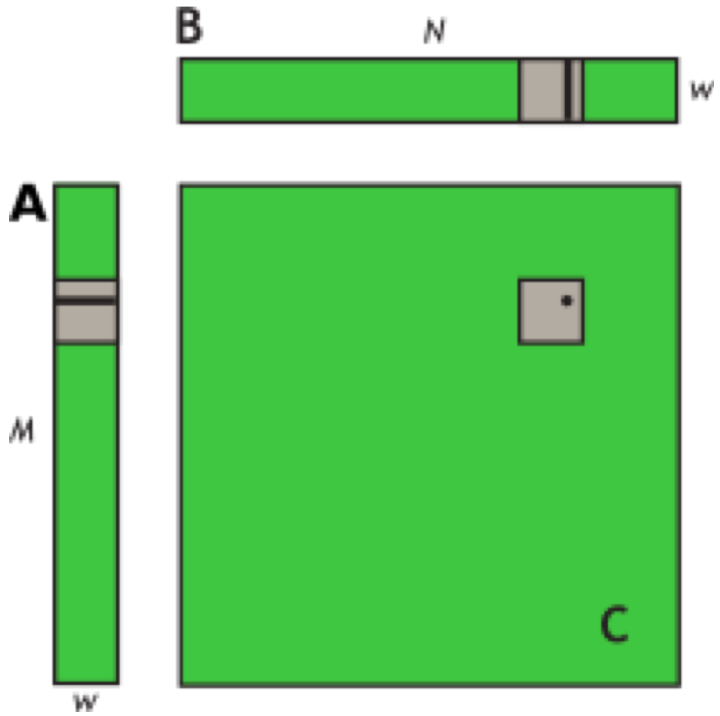


Example: Matrix Multiplication

- https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html#memory-optimizations

Using shared memory to improve the global memory load efficiency in matrix multiplication

```
__global__ void coalescedMultiply(float *a, float* b, float *c,
                                  int N)
{
    __shared__ float aTile[TILE_DIM][TILE_DIM];

    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;
    float sum = 0.0f;
    aTile[threadIdx.y][threadIdx.x] = a[row*TILE_DIM+threadIdx.x];
    __syncwarp();
    for (int i = 0; i < TILE_DIM; i++) {
        sum += aTile[threadIdx.y][i]* b[i*N+col];
    }
    c[row*N+col] = sum;
}
```



Example: Matrix Multiplication

- https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html#memory-optimizations

Improvement by reading additional data into shared memory

```
__global__ void sharedABMultiply(float *a, float* b, float *c,
                                 int N)
{
    __shared__ float aTile[TILE_DIM][TILE_DIM],
                     bTile[TILE_DIM][TILE_DIM];
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;
    float sum = 0.0f;
    aTile[threadIdx.y][threadIdx.x] = a[row*TILE_DIM+threadIdx.x];
    bTile[threadIdx.y][threadIdx.x] = b[threadIdx.y*N+col];
    __syncthreads();
    for (int i = 0; i < TILE_DIM; i++) {
        sum += aTile[threadIdx.y][i]* bTile[i][threadIdx.x];
    }
    c[row*N+col] = sum;
}
```