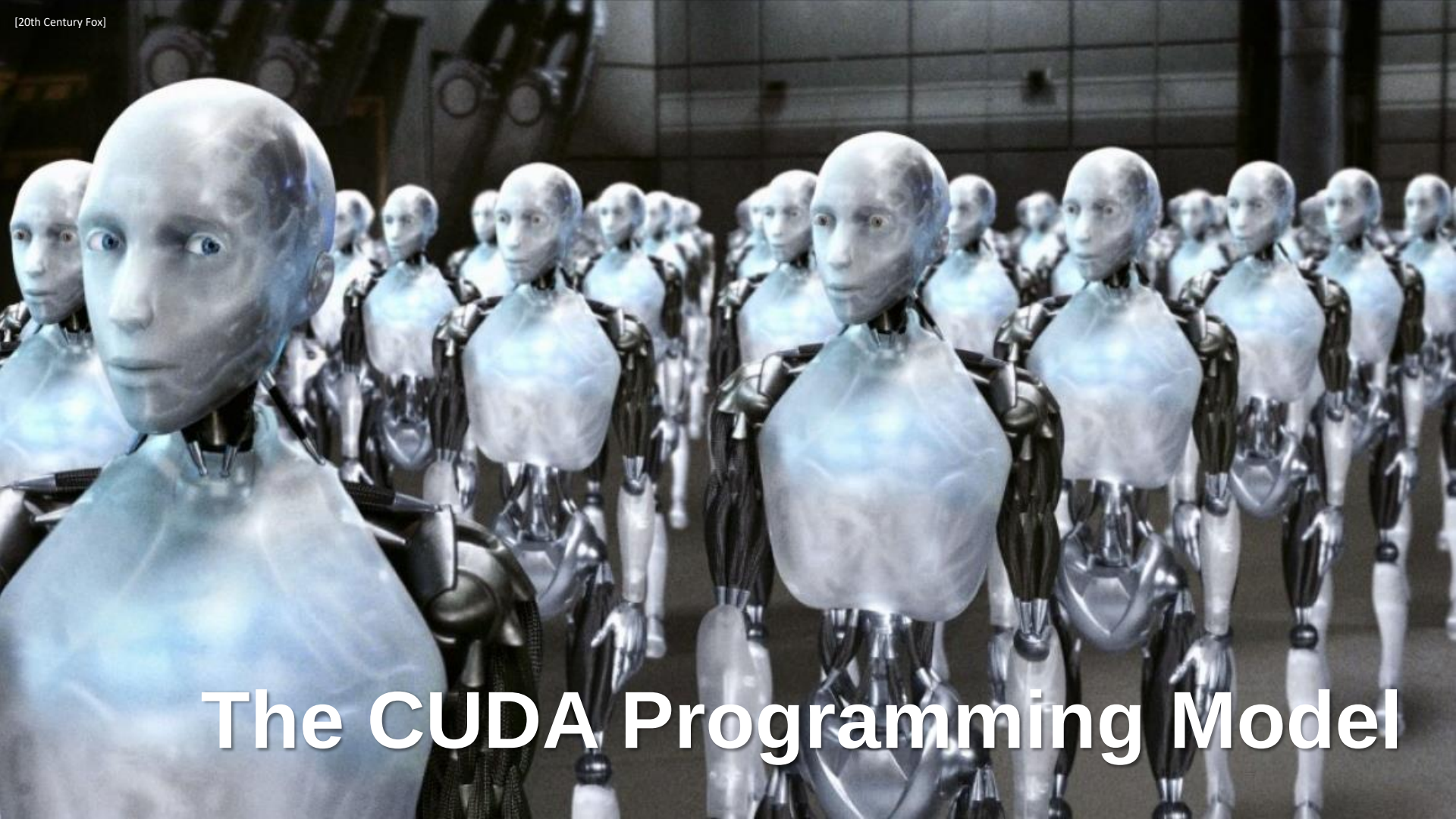
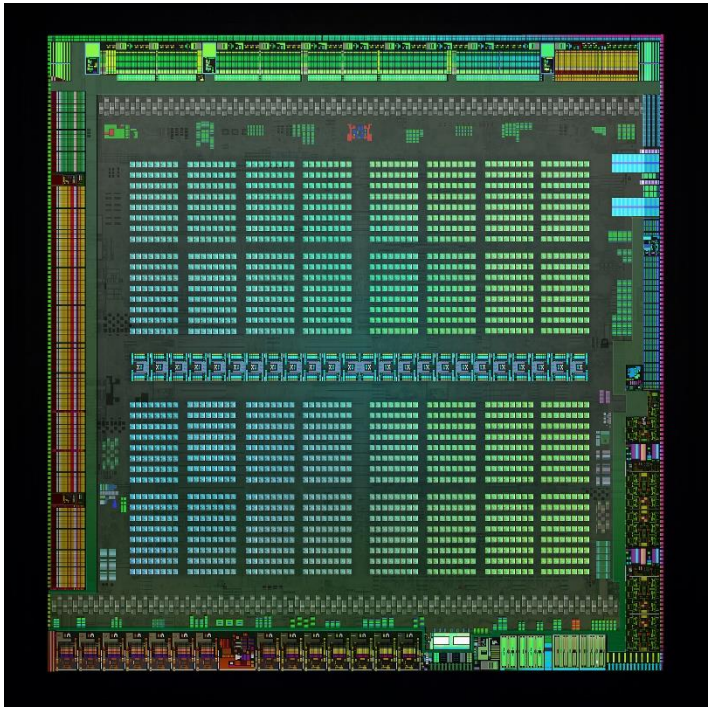
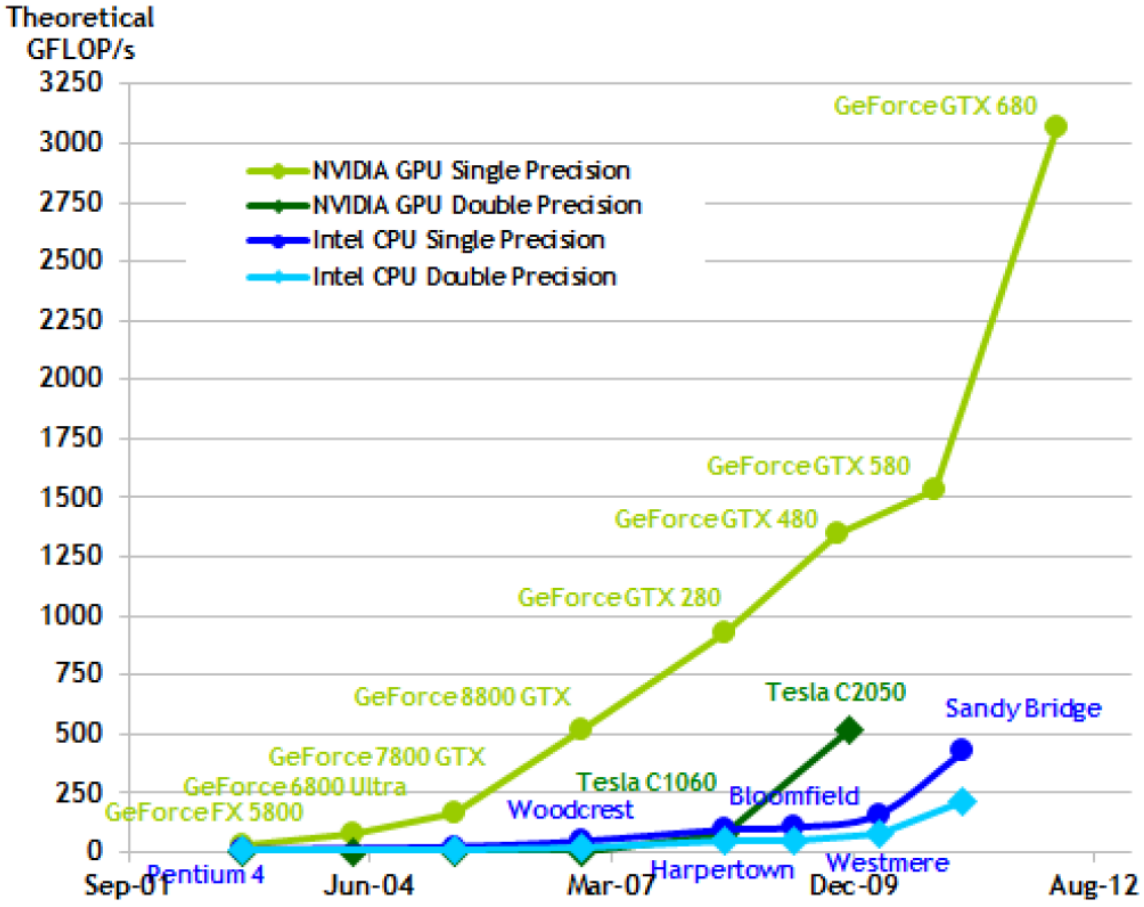


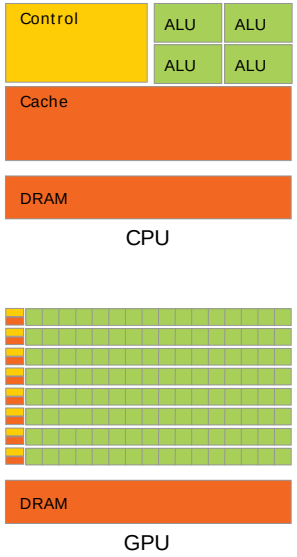
The CUDA Programming Model

NVIDIA CUDA

Compute Unified Device Architecture




Theoretical GFLOP/s
3250
3000
2750
2500
2250
2000
1750
1500
1250
1000
750
500
250
0
NVIDIA GPU Single Precision
NVIDIA GPU Double Precision
Intel CPU Single Precision
Intel CPU Double Precision
GeForce GTX 680
GeForce GTX 580
GeForce GTX 480
GeForce GTX 280
GeForce 8800 GTX
GeForce 7800 GTX
GeForce 6800 Ultra
GeForce FX 5800
Tesla C2050
Sandy Bridge
Tesla C1060
Bloomfield
Woodcrest
Pentium 4
Harpertown
Westmere
Sep-01
Jun-04
Mar-07
Dec-09
Aug-12


Control
ALU
ALU
ALU
ALU
Cache
DRAM
CPU
DRAM
GPU

Types of parallelism

Task Parallelism

- Parallelize different, independent computation
- Distribute tasks to processors
- e.g. Multitasking, Pipeline Parallelism

Data Parallelism

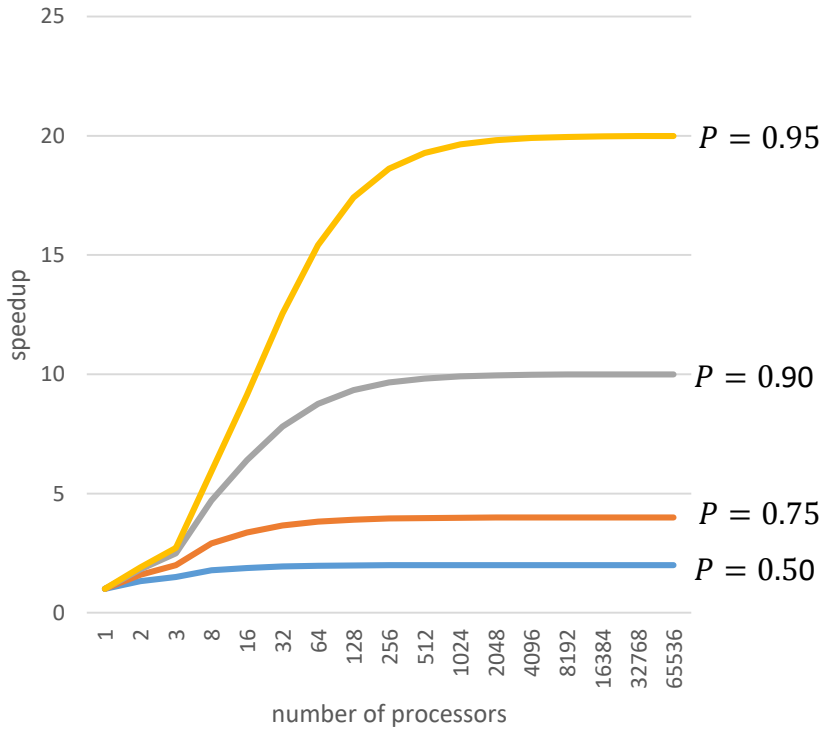
- Parallelize same computation on different, independent data
- Distribute data to processors
- e.g. Image Processing, Loop-level Parallelism, Tiling, Divide and Conquer

Amdahl's Law

- Speedup S over purely sequential implementation

$$S = \frac{1}{(1-P) + \frac{P}{N}}$$

P: parallelizable part
N: # of processors

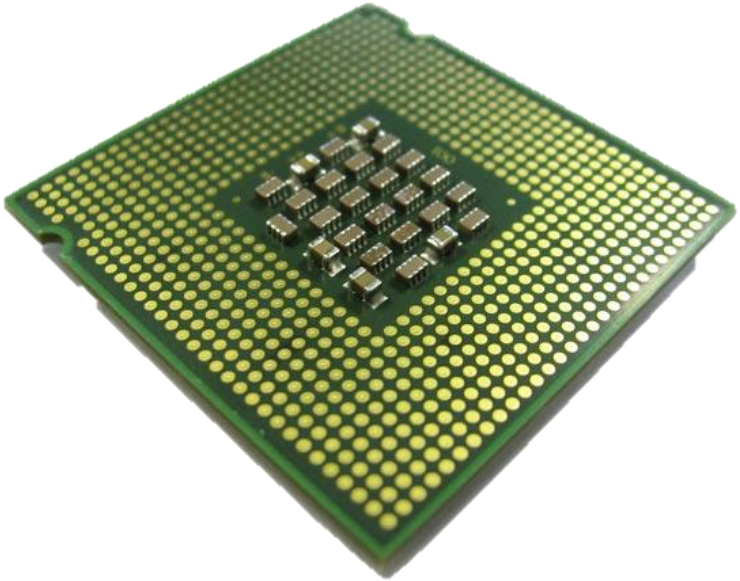


GPU Architecture

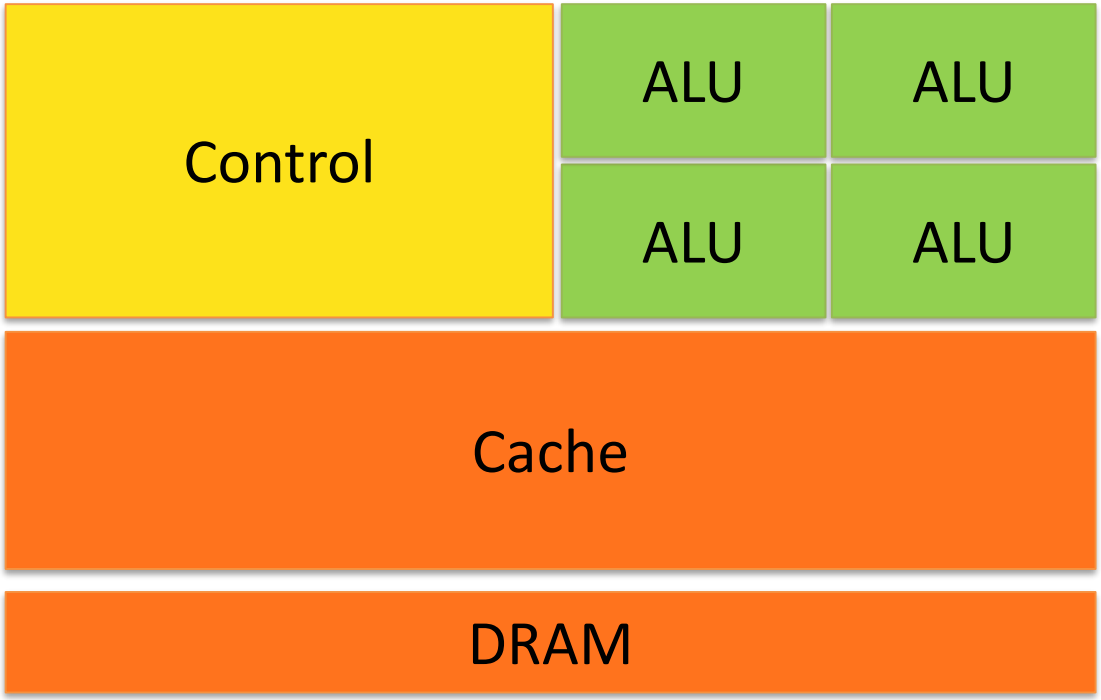
CPU

- Goal: get the best performance for a single heavy weight thread
- Big data caches
- Low latency arithmetic units
- Complex Control Logic:
  - Branch prediction
  - Out-of-order-execution

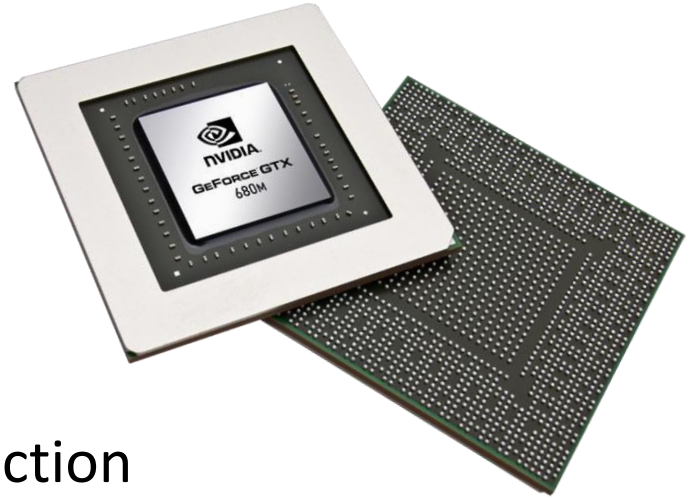
→ Latency oriented design



CPU



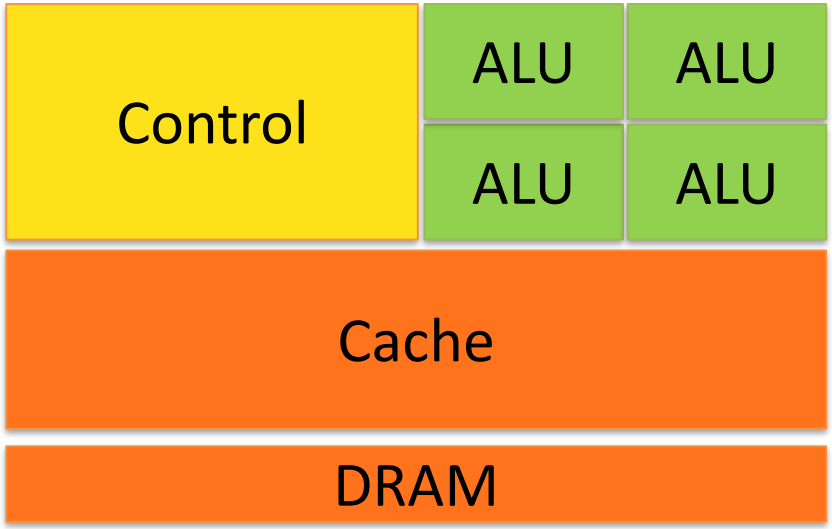
GPU



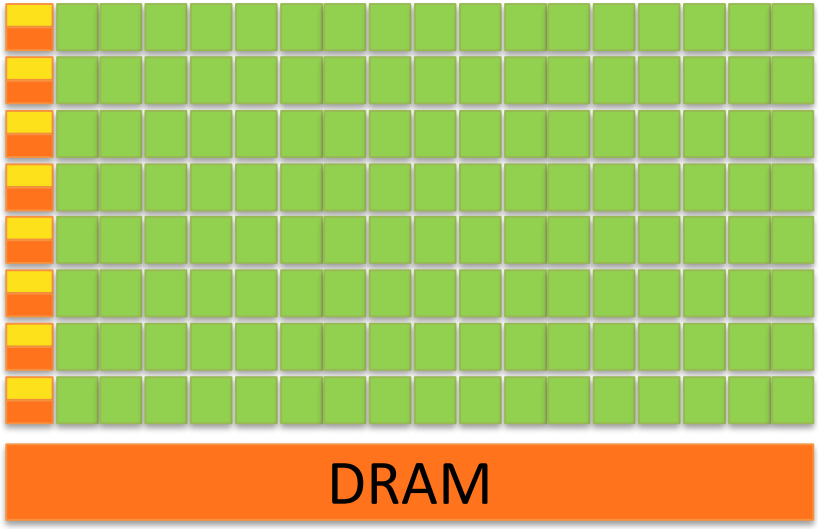
- Goal: get the best performance for thousands of simple threads
- Small caches
- Hide latency with computation
- In-order execution with no branch prediction
- Issue the same command to multiple cores

→ Throughput oriented design

CPU vs GPU



CPU

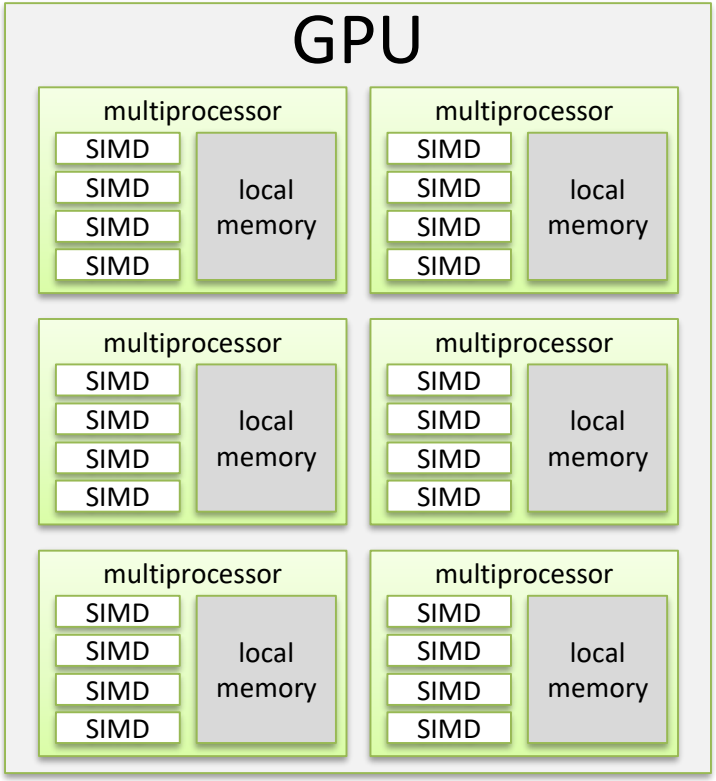


GPU

Flynn‘s Taxonomy

- SISD single-instruction, single-data (e.g. single core CPU)
- MIMD multiple-instruction, multiple-data (e.g. multi core CPU)
- SIMD single-instruction, multiple-data (e.g. data-based parallelism)
- MISD multiple-instruction, single-data (e.g. fault-tolerant computers)

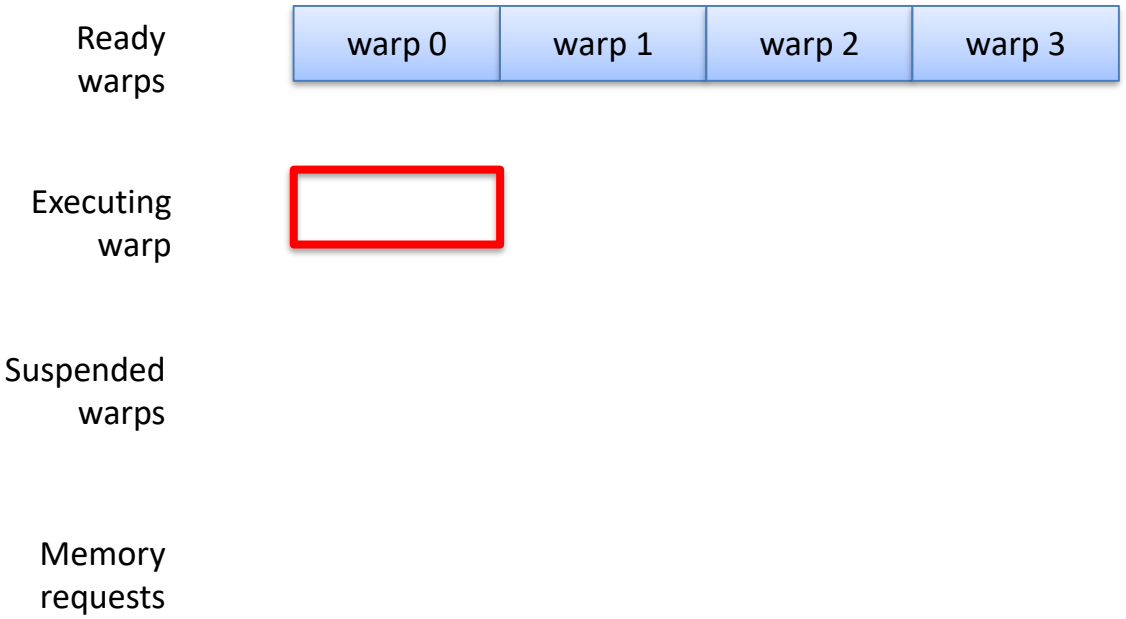
CUDA GPU (Maxwell)


GPU
multiprocessor
SIMD
SIMD
SIMD
SIMD
local memory
multiprocessor
SIMD
SIMD
SIMD
SIMD
local memory
multiprocessor
SIMD
SIMD
SIMD
SIMD
local memory
multiprocessor
SIMD
SIMD
SIMD
SIMD
local memory
multiprocessor
SIMD
SIMD
SIMD
SIMD
local memory
multiprocessor
SIMD
SIMD
SIMD
SIMD
local memory

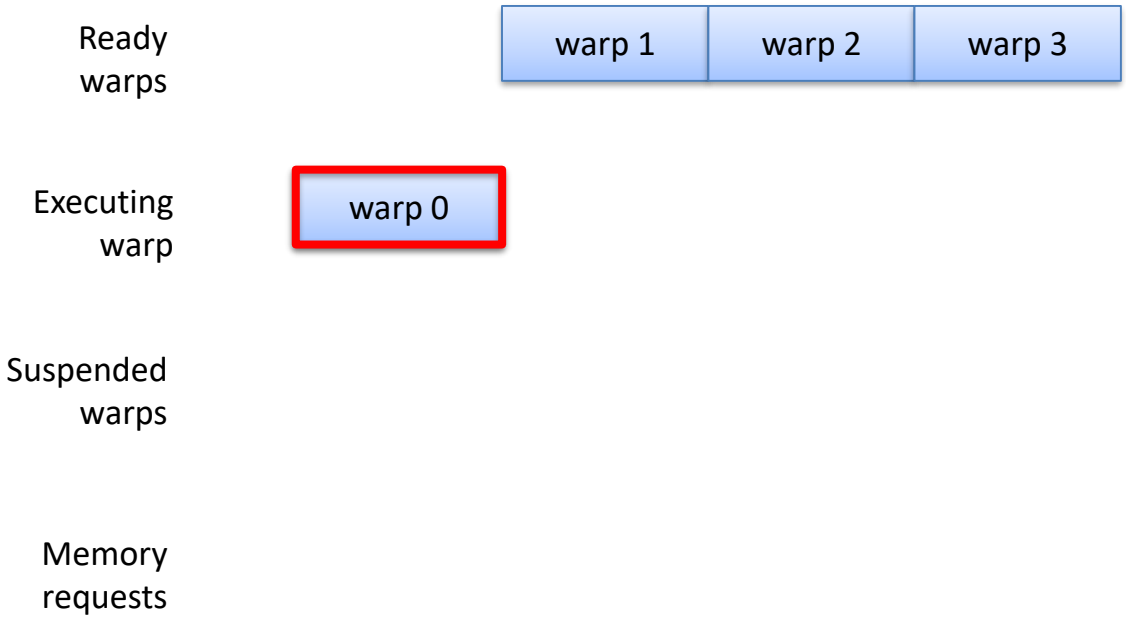
Streaming Multiprocessors (SM/SMX)

- SIMD hardware
- 32 threads form a *warp*
- Each thread within a warp must execute the same instruction (or be deactivated)
- 1 instruction → 32 values computed
- handle more warps than cores to hide latency

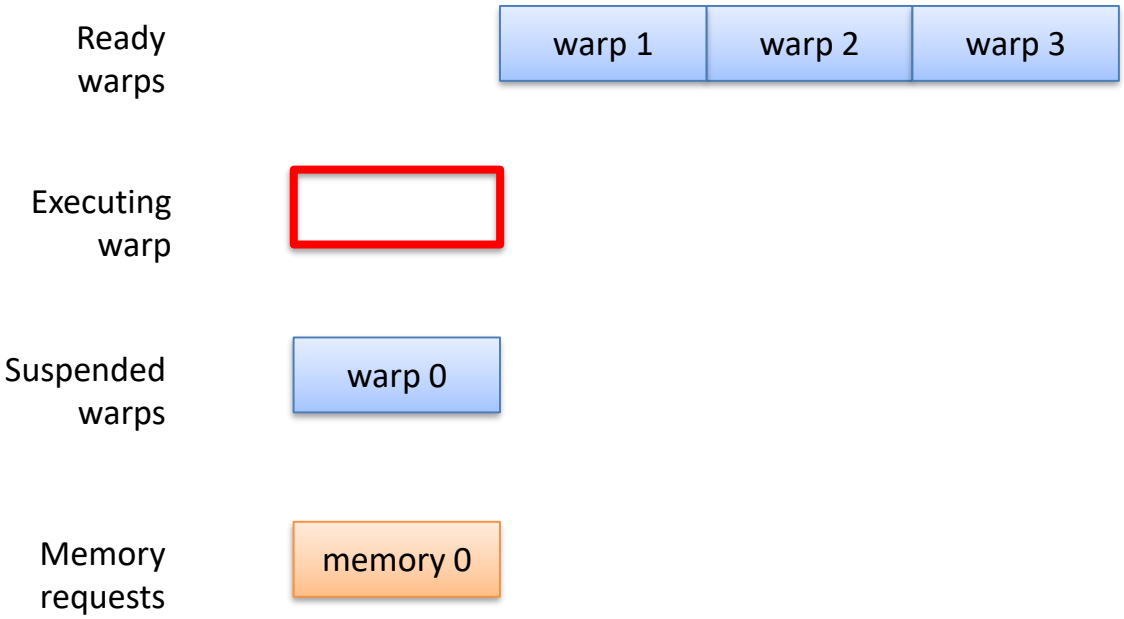
Warp scheduling



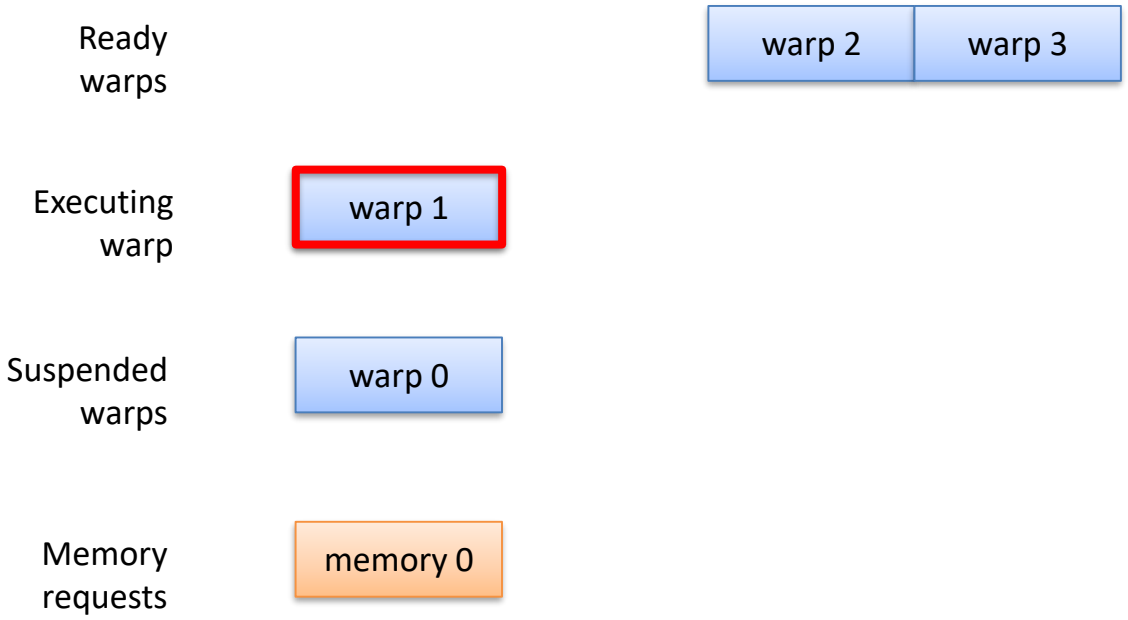
Warp scheduling



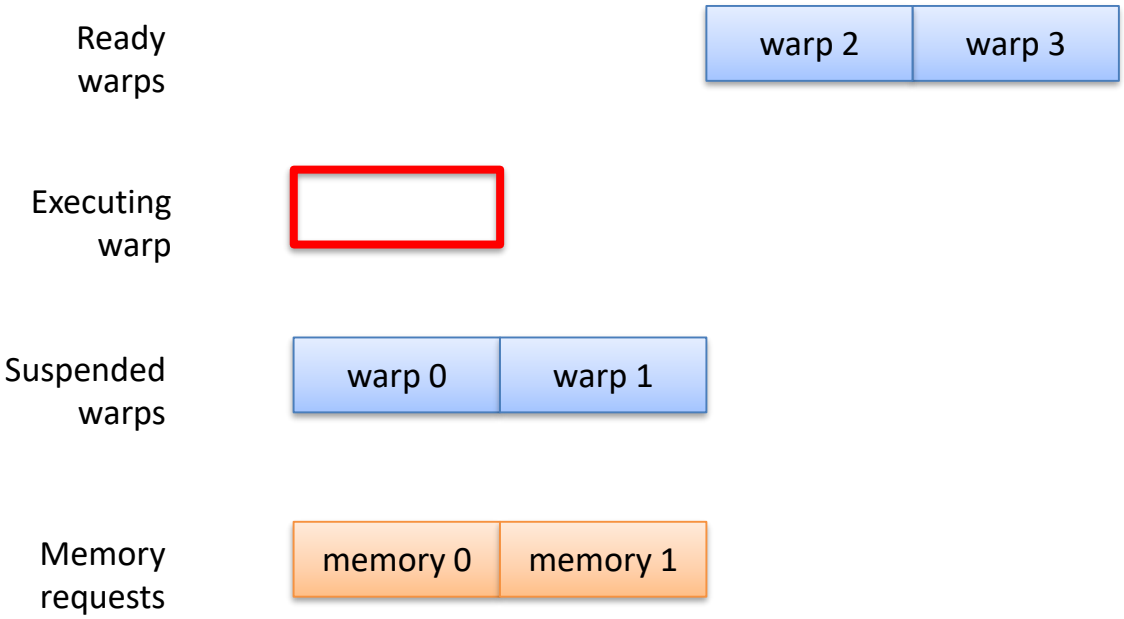
Warp scheduling



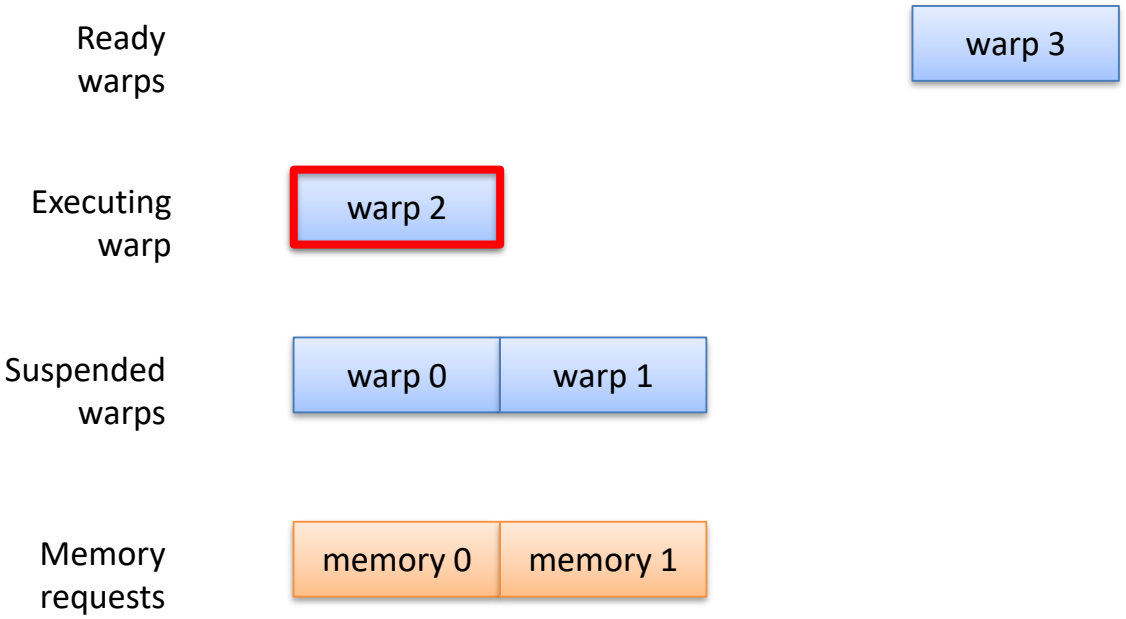
Warp scheduling



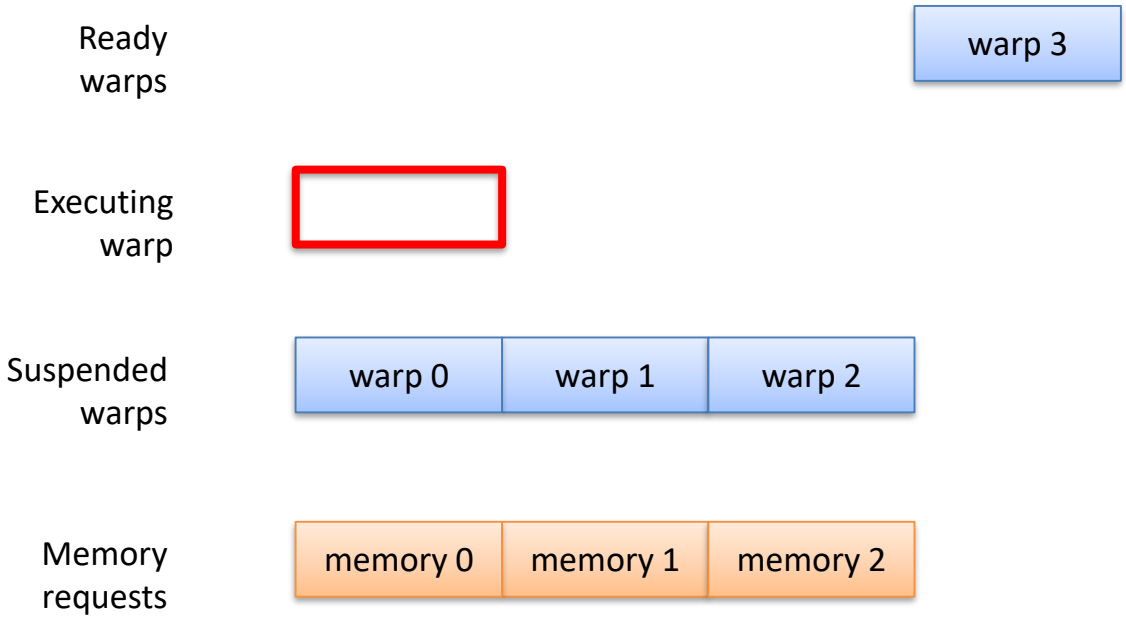
Warp scheduling



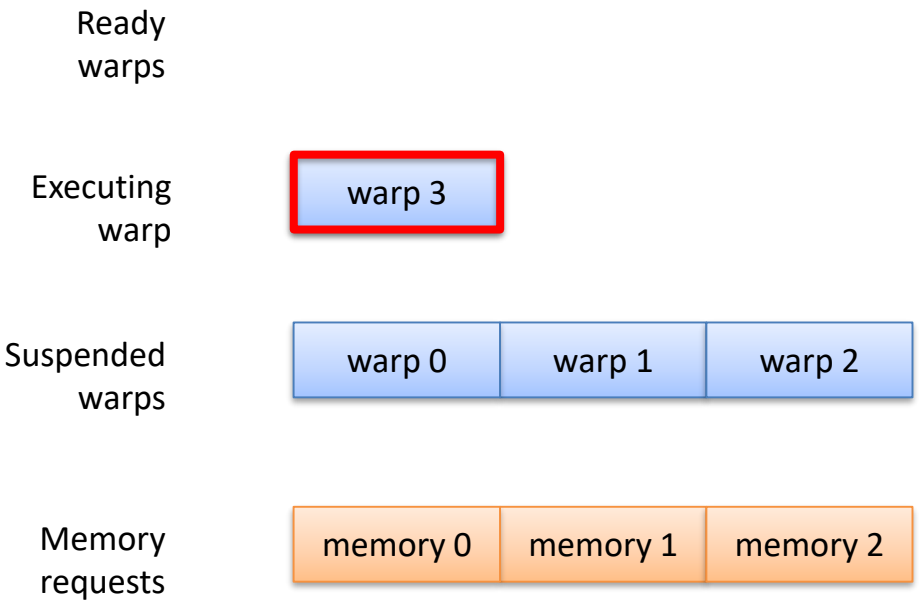
Warp scheduling



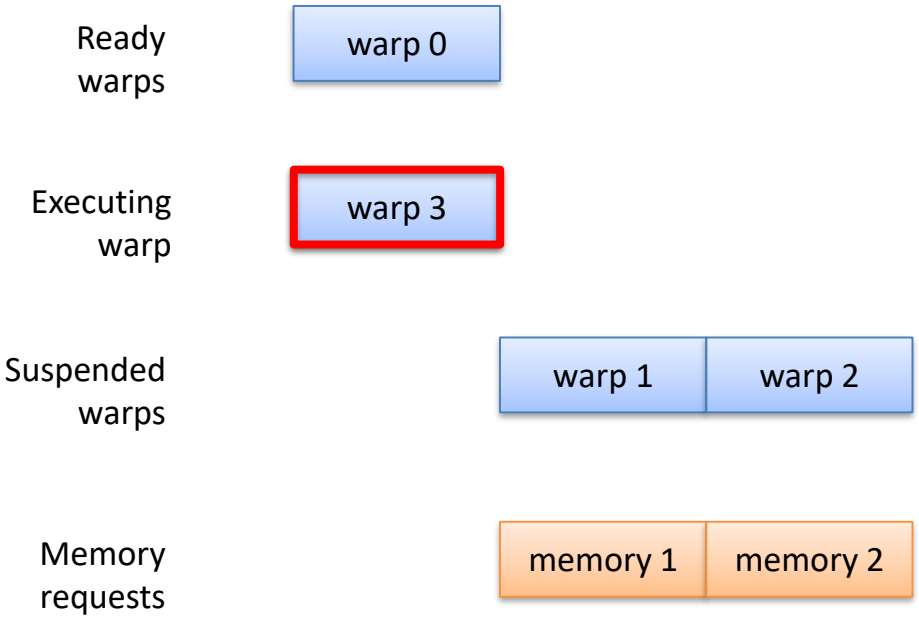
Warp scheduling



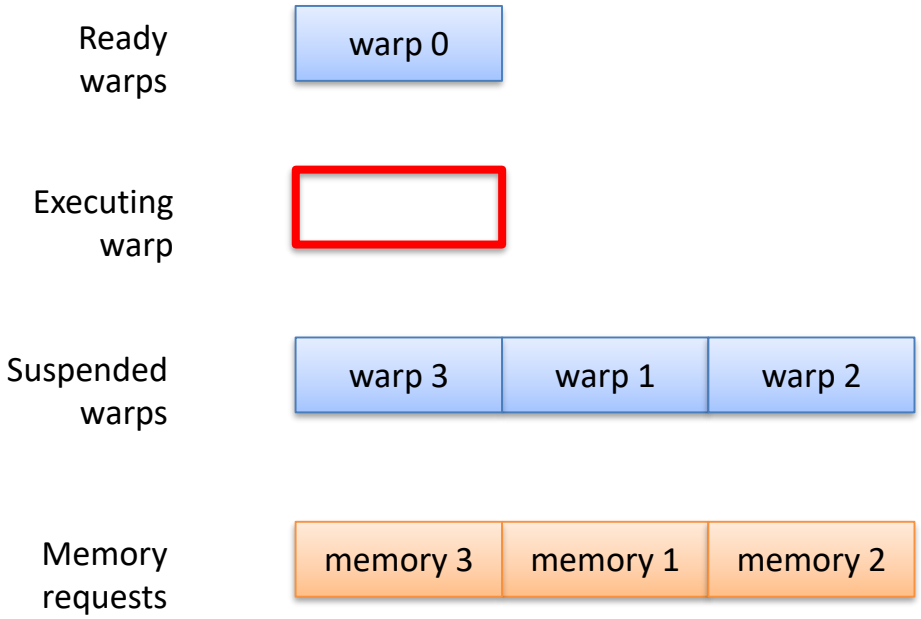
Warp scheduling



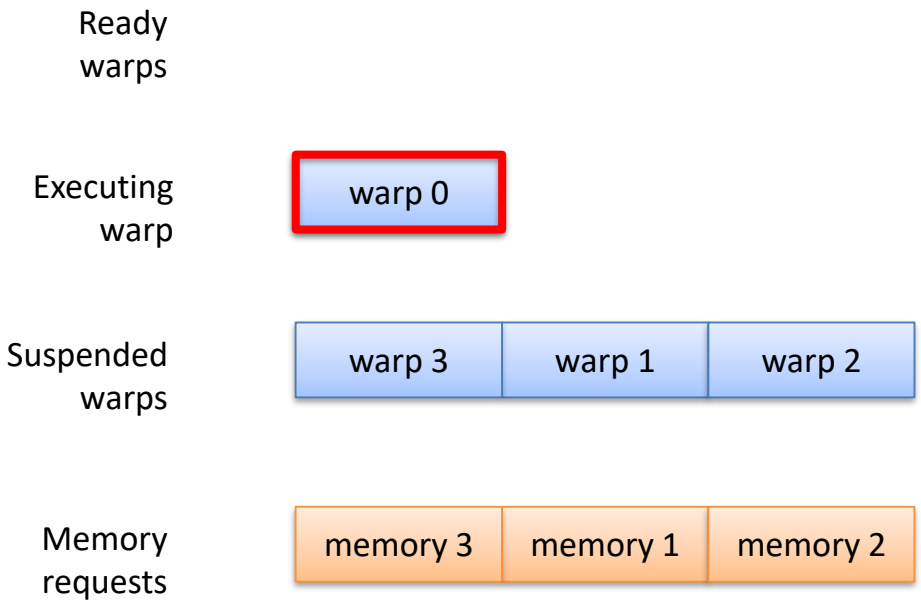
Warp scheduling



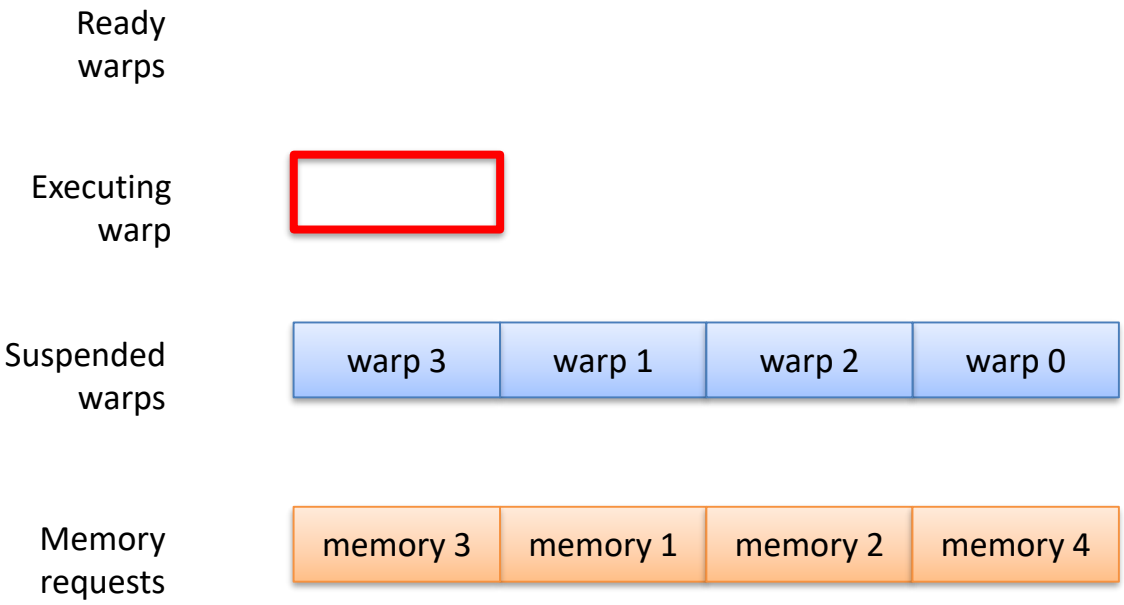
Warp scheduling



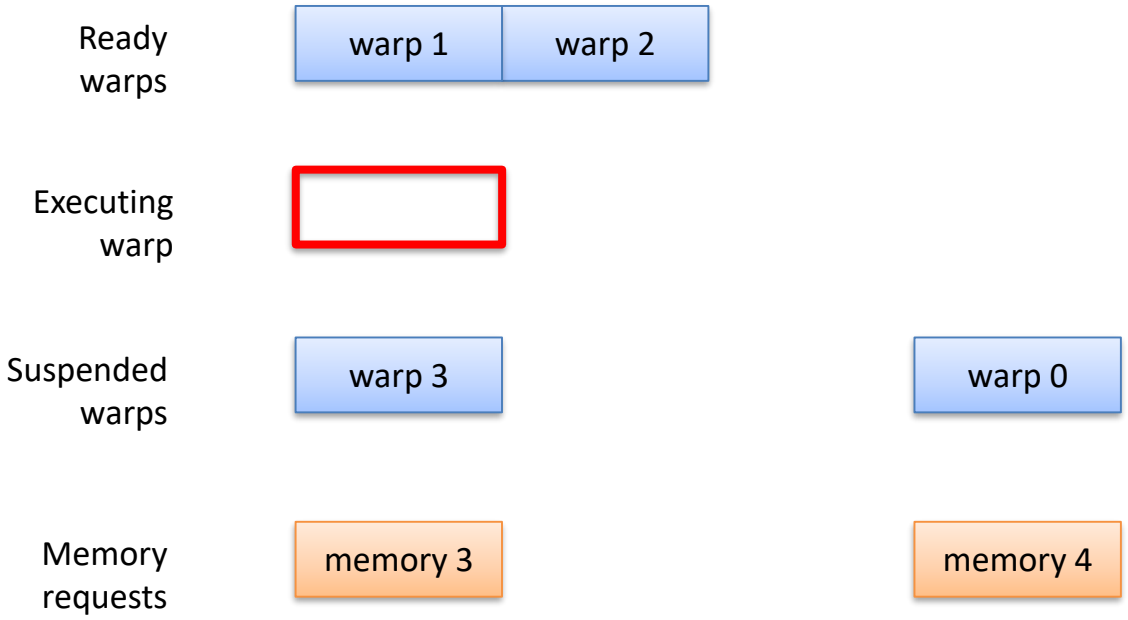
Warp scheduling



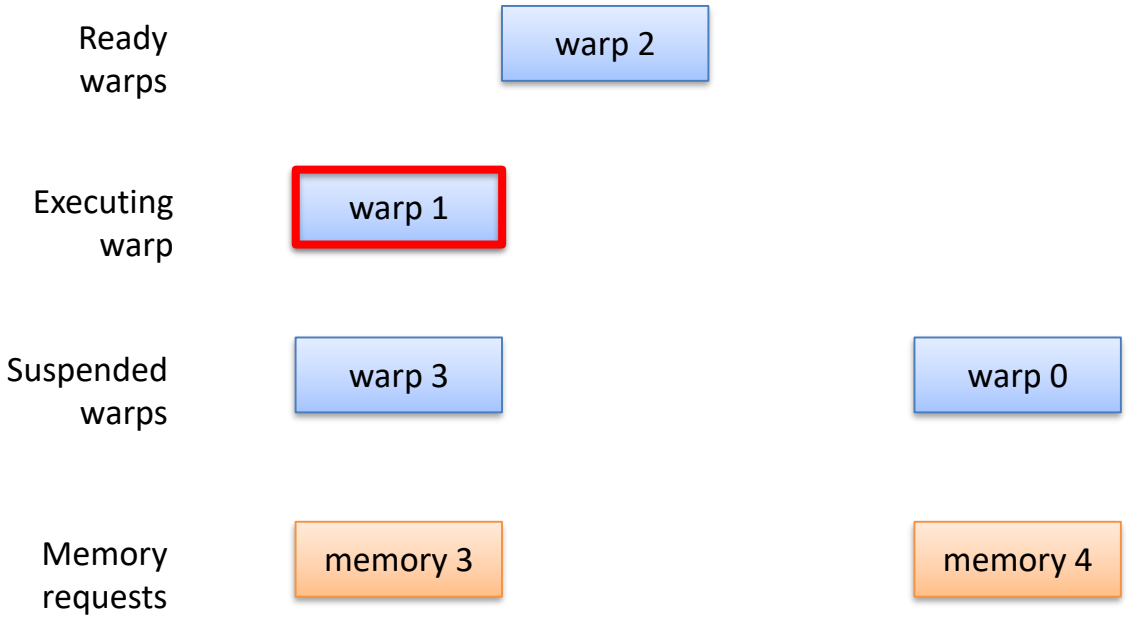
Warp scheduling



Warp scheduling



Warp scheduling



The CUDA Programming Model

Writing parallel code

- Current GPUs have 3072 cores (GTX TITAN X)
- Need more threads than cores (warp scheduler)
- Writing code for 10000 threads / 300 warps?

→Single-program, multiple-data (SPMD) model

  – Write one program that is executed by all threads

CUDA C

- CUDA C is C (C++) with additional keywords to control parallel execution

- Type qualifiers
- Builtins
- Intrinsics

- Runtime API
- GPU kernel launches

```
__device__ float x;
__global__ void func(int* mem)
{
  __shared__ int y[32];
  …
  y[threadIdx.x] = blockIdx.x;
  __syncthreads();
}
…
cudaMalloc(&d_mem, bytes);
func<<<10,10>>>(d_mem);
```

GPU code (device code)

CPU code (host code)

CUDA Runtime API

- Write Host (CPU) and Device (GPU) Code into the same file
- nvcc compiler driver takes care of
  - separating Host and Device code
  - generating additional Host code
    - e.g. for kernel launches
  - sending each part through the respective toolchain

Kernel

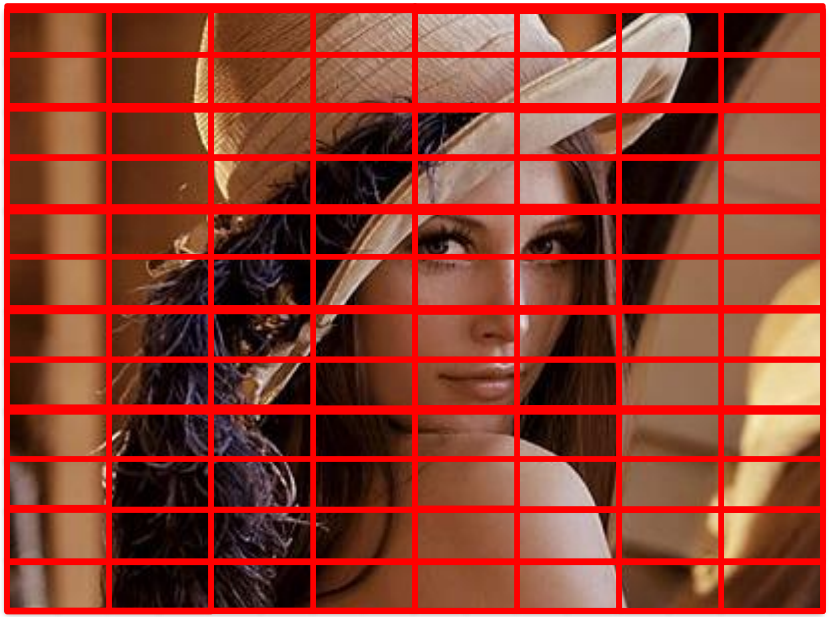
- A function that is executed on the GPU
- Each thread is executing the same function

```
__global__ void myfunction(float *input, float* output)
{
  *output = *input;
}
```

- Indicated by `__global__`
- must have return value `void`

Parallel Kernel

- Kernel is split up in blocks of threads



Launching a kernel

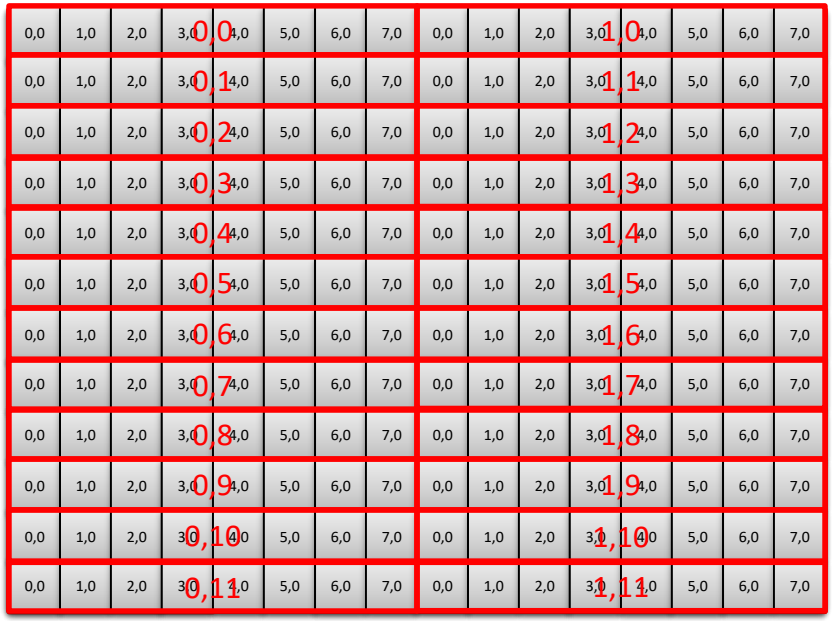
- Requires threads per block, number of blocks
- Launched is asynchronous

```
dim3 blockSize(128,1,1);
dim3 gridSize(12,1,1)
myfunction<<<gridSize,blockSize>>>(input, output);
```

- Block and thread setup specified within <<< , >>>
- Input parameters are copied to the GPU

Distinguishing between threads

- blockId and threadId

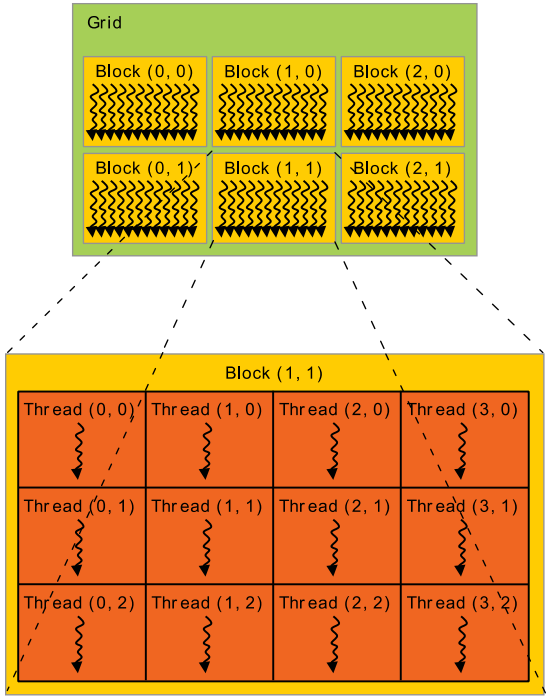


Distinguishing between threads

- using **threadIdx** and **blockIdx** execution paths are chosen
- with **blockDim** and **gridDim** number of threads can be determined

```
__global__ void myfunction(float *input, float*
output)
{
  uint id = threadIdx.x + blockIdx.x * blockDim.x;
  output[id] = input[id];
}
```

Grids, Blocks, Threads



Blocks

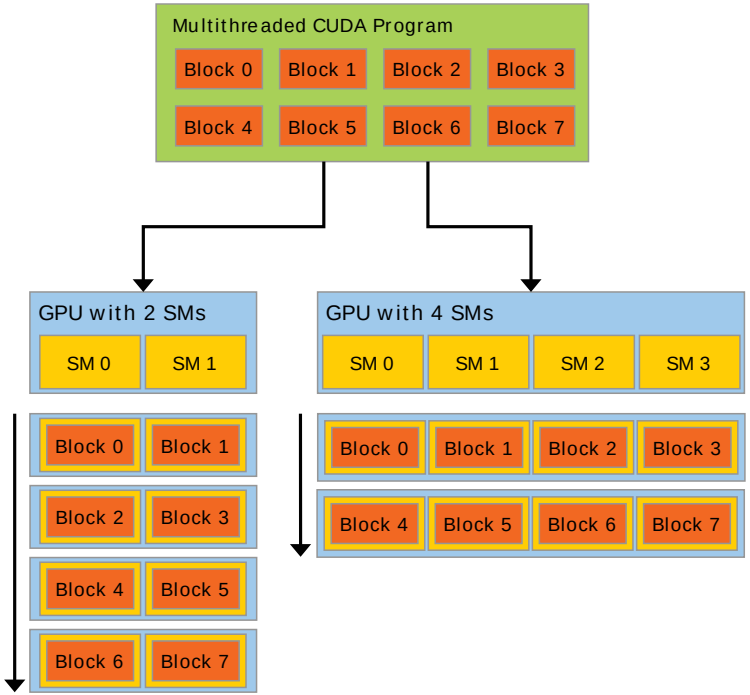
- Threads within one block…
  - are executed together on the same SM
  - can synchronize
  - can communicate efficiently
  - share the same local cache

  → can be used to cooperatively compute some result
- Threads of different blocks…
  - may be executed one after another on different SMs
  - cannot synchronize
  - can only communicate inefficiently

  → should work independently of other blocks

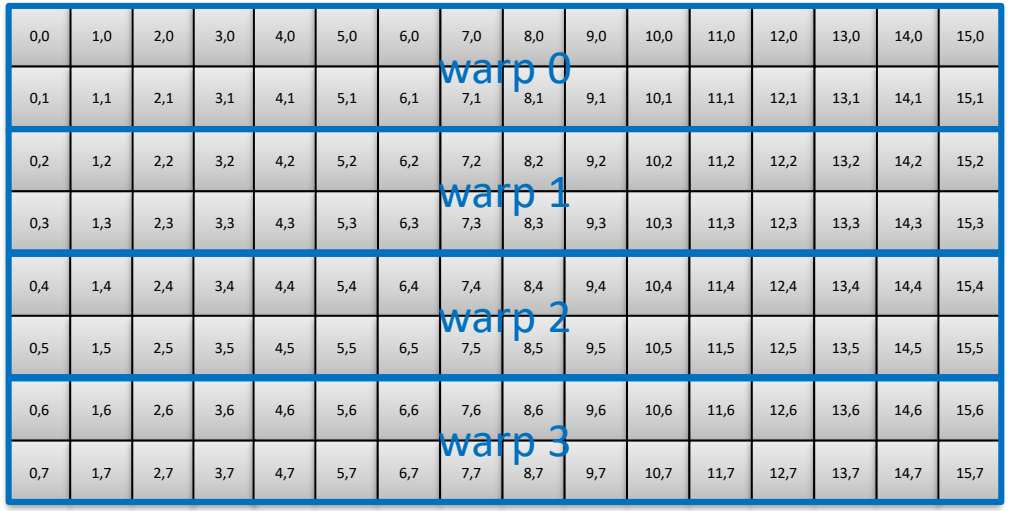
Block Scheduling

- Block queue feeds multiprocessors
- Number of available multiprocessors determines number of concurrently executed blocks



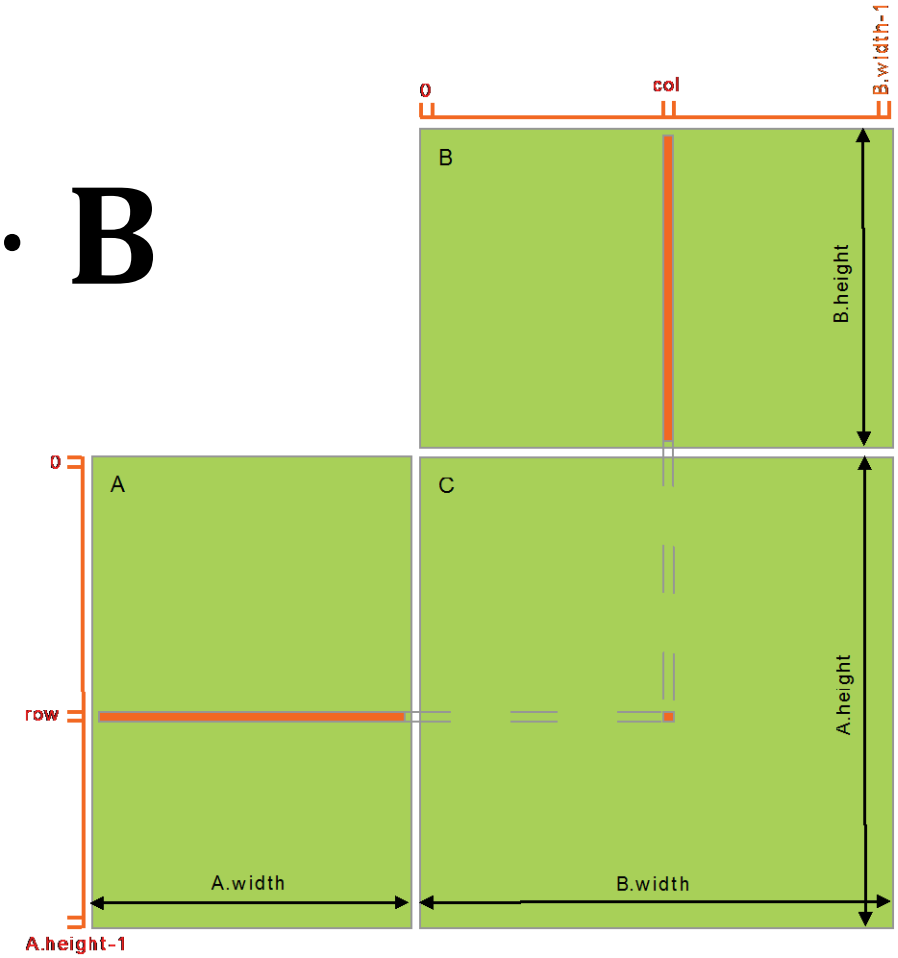
Blocks to warps

- On each multiprocessor each block is split up in warps
- Threads with the lowest id map to the first warp



Example:
Matrix Multiplication

$$\mathbf{C} = \mathbf{A} \cdot \mathbf{B}$$



```
7  std::vector<float> cpu_multiply(const std::vector<float>& A, size_t A_cols, size_t A_rows,
8    const std::vector<float>& B, size_t B_cols, size_t B_rows)
9  {
10   std::cout << "computing C on the CPU";
11   size_t C_rows = A_rows,
12          C_cols = B_cols;
13   std::vector<float> C(C_rows*C_cols);
14
15   PointInTime t0;
16   size_t elements = A_cols;
17   auto cIt = C.begin();
18   for(size_t y = 0; y < C_rows; ++y)
19     for(size_t x = 0; x < C_cols; ++x)
20     {
21       float c_xy = 0;
22       for(size_t i = 0; i < elements; ++i)
23         c_xy += A[y*A_cols + i]*B[i*B_cols + x];
24       *cIt++ = c_xy;
25     }
26   PointInTime t1;
27   std::cout << " done in " << (t1-t0) << " seconds\n";
28   return C;
29 }
```

Loop-based parallelism

```
82  __global__ void d_multiply(const float* A, int A_cols, int A_rows,
83                             const float* B, int B_cols, int B_rows,
84                             float* C)
85  {
86    int C_cols = B_cols,
87        C_rows = A_rows;
88
89    int x = blockIdx.x*blockDim.x + threadIdx.x;
90    int y = blockIdx.y*blockDim.y + threadIdx.y;
91
92
93    int elements = A_cols;
94    float c_xy = 0;
95    for(int i = 0; i < elements; ++i)
96        c_xy += A[y*A_cols + i]*B[i*B_cols + x];
97    C[y*C_cols + x] = c_xy;
98
99  }
```

```
for(size_t y = 0; y < C_rows; ++y)
  for(size_t x = 0; x < C_cols; ++x)
  {
```

Kernel call

```
for(size_t y = 0; y < C_rows; ++y)
  for(size_t x = 0; x < C_cols; ++x)
  {
```

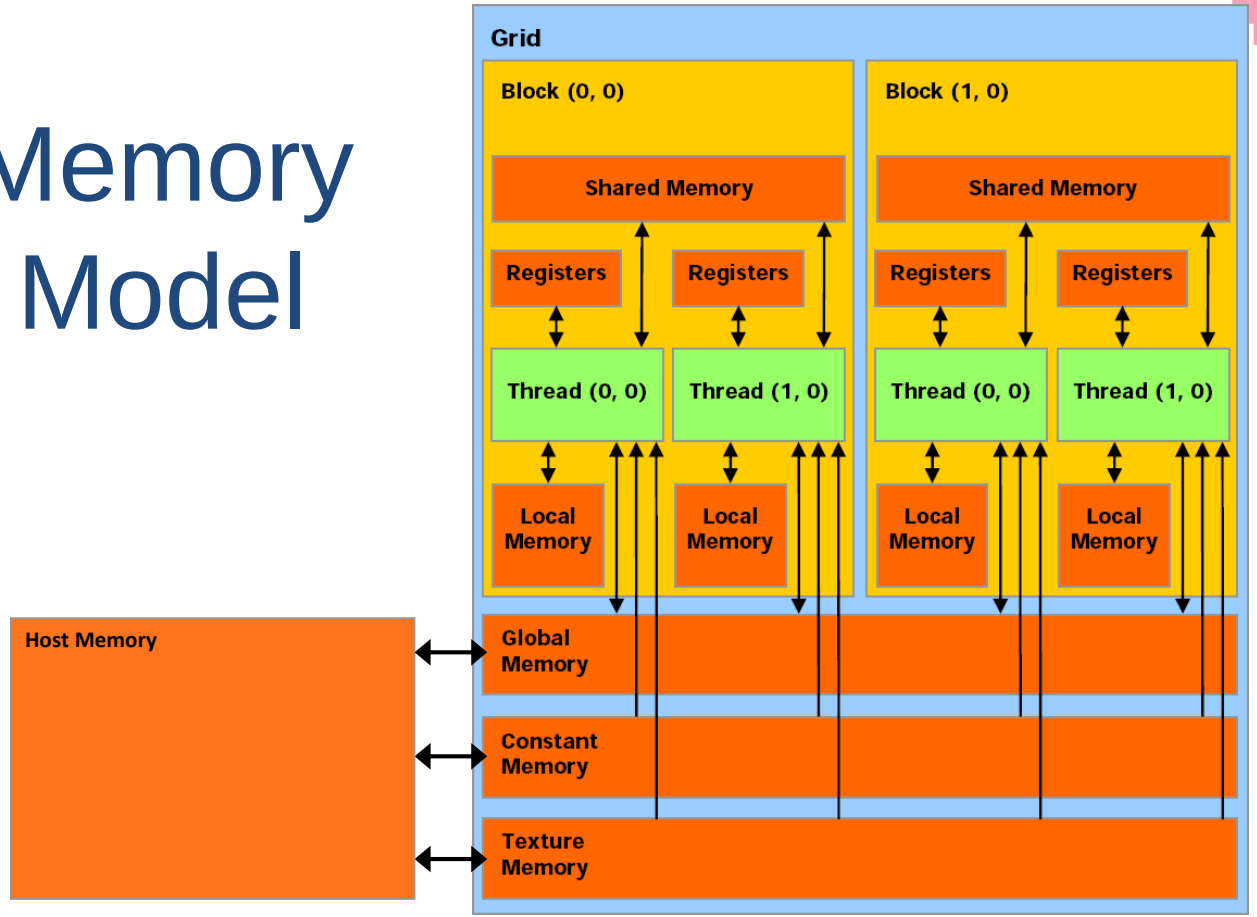
```
//THE KERNEL CALL
dim3 blockSize(16,16,1);
dim3 gridSize( C_cols/blockSize.x, C_rows/blockSize.y, 1);
d_multiply<<<gridSize, blockSize>>>(d_A, A_cols, A_rows, d_B, B_cols, B_rows, d_C);

//SYNCHRONIZE
cudaError err = cudaDeviceSynchronize();
```

E:\teaching\ezg2\documents\Cuda_lecture\matrixmultiplication\build\Release\matrixmultiplication.exe

```
going to use Tesla K20c
creating matrix A (800x1600)
creating matrix B (1600x800)
computing C on the CPU done in 16.5319 seconds
computing C on the GPU done in 0.0448455 seconds (0.0447093s on device)
computing difference
Max absolute difference: 7.62939e-005@ -107.123 -107.123
Avg rel difference: 1.77451e-006
```

Memory Spaces

Memory Model



Registers

- Private for each thread
- Fastest memory
- Automatically allocated from per SM register file (number of registers per block)
- no keyword

Global Memory

- Main GPU memory
- Accessible from all blocks/threads
- Access pattern matters
  - Older devices coalescing!
  - Newer devices: close together
- Slowest on device memory
  (up to 1000x times slower than registers)
- Cached on newer devices
- `__device__`/`cudaMalloc`

Texture Memory

- Accessible from all blocks/threads
- Cached on all devices
  - Special layout for 2D and 3D textures
- Hardware interpolation (same as OpenGL)
- Automatic handling of boundary cases
- Normalized texture coordinates
- Conversion to float
- No concurrent read and write
- `cudaMallocArray + cudaBindTextureToArray`

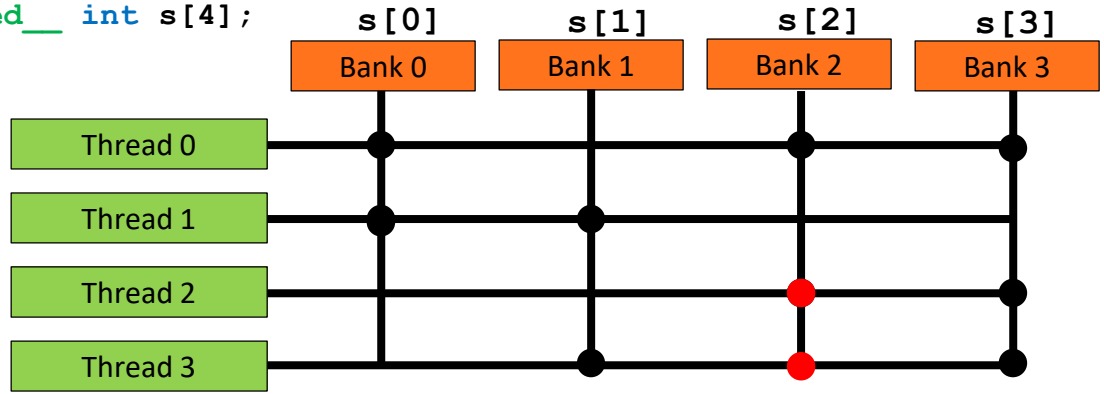
Constant Memory

- Read-only
- Ideal for coefficients and other data that is read uniformly by warps
- Cached on all devices
- `__constant__` + `cudaMemcpyToSymbol`

Shared Memory

- Shared access within one block (lifetime: block)
- Located on multiprocessor → very fast
- Limited available size on multiprocessor
- Crossbar: simultaneous access to distinct banks

```
__shared__ int s[4];
```



Local Memory

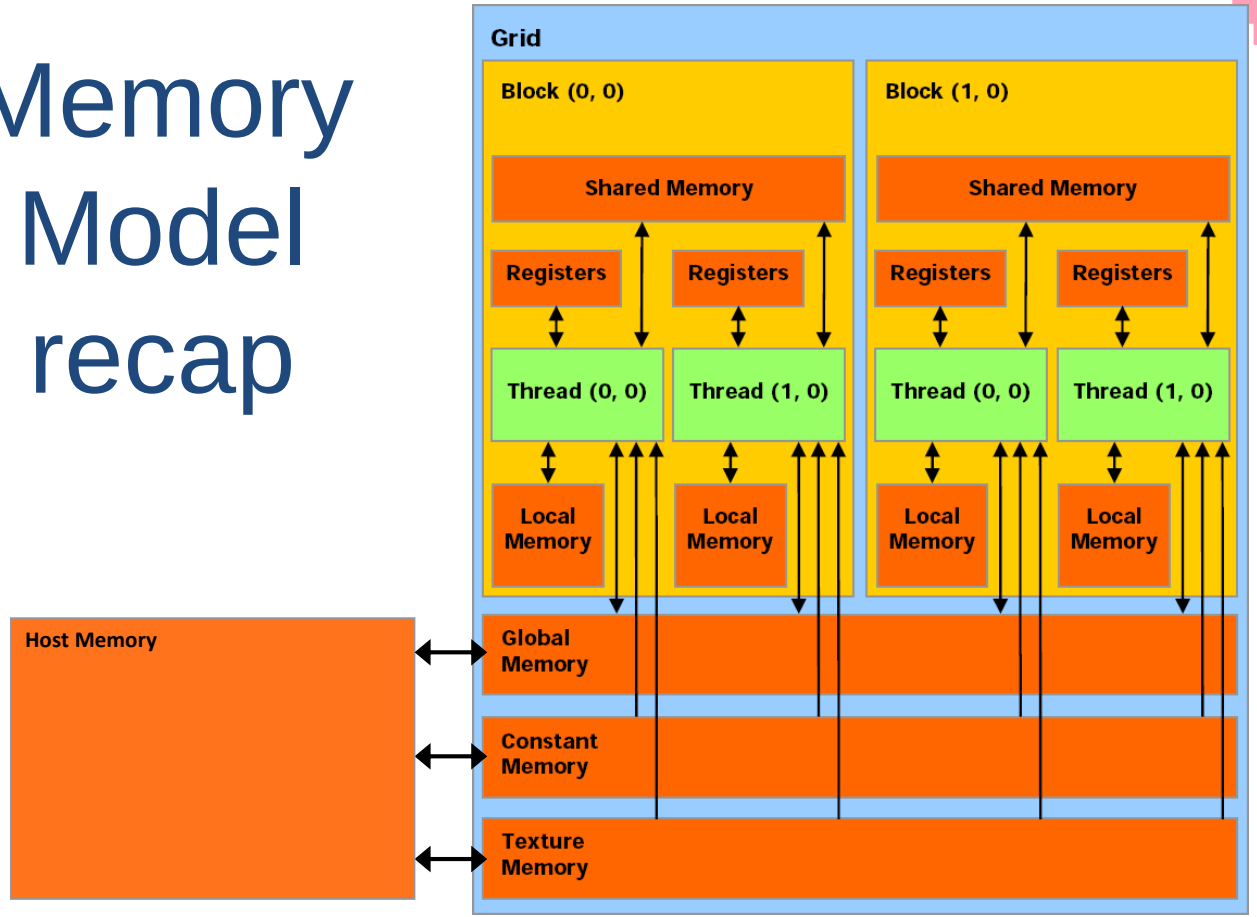
- If above register limited, registered are spilled to global
- Very slow in comparison to registers
- Besides that same behavior as registers
- Always perfect access pattern

Host Memory

- Can be mapped to device memory space
- Same data can be accessed from host and device concurrently
- Even slower than global memory
- Mapped memory:

  `cudaHostAlloc` with `cudaHostAllocMapped`

Memory Model recap



Memory Allocation

	Global	Constant	Shared	Local	Registers
Host Code	Dynamic allocation R/W	No allocation R/W	Dynamic allocation No access	No allocation No access	No allocation No access
Device Code	Static allocation (Dynamic allocation) R/W	Static allocation Read-only	Static allocation R/W	Static Allocation R/W	Static Allocation R/W

Matrix Multiplication: Memory

```
//GLOBAL MEMORY ALLOCATION
float *d_A, *d_B, *d_C;
cudaMalloc(&d_A, A_cols*A_rows*sizeof(float));
cudaMalloc(&d_B, B_cols*B_rows*sizeof(float));
cudaMalloc(&d_C, C_cols*C_rows*sizeof(float));

//MEMORY TRANSFER
cudaMemcpy(d_A, &A[0], A_cols*A_rows*sizeof(float), cudaMemcpyHostToDevice);
cudaMemcpy(d_B, &B[0], B_cols*B_rows*sizeof(float), cudaMemcpyHostToDevice);

//THE KERNEL CALL
dim3 blockSize(16,16,1);
dim3 gridSize( C_cols/blockSize.x, C_rows/blockSize.y, 1);
d_multiply<<<gridSize, blockSize>>>(d_A, A_cols, A_rows, d_B, B_cols, B_rows, d_C);

//SYNCHRONIZE
cudaError err = cudaDeviceSynchronize();

//COPY TO HOST
std::vector<float> C(C_rows*C_cols);
cudaMemcpy(&C[0], d_C, C_cols*C_rows*sizeof(float), cudaMemcpyDeviceToHost);

//MEMORY CLEAN UP
cudaFree(d_A);
cudaFree(d_B);
cudaFree(d_C);
```

Example:
Tiled Matrix Multiplication

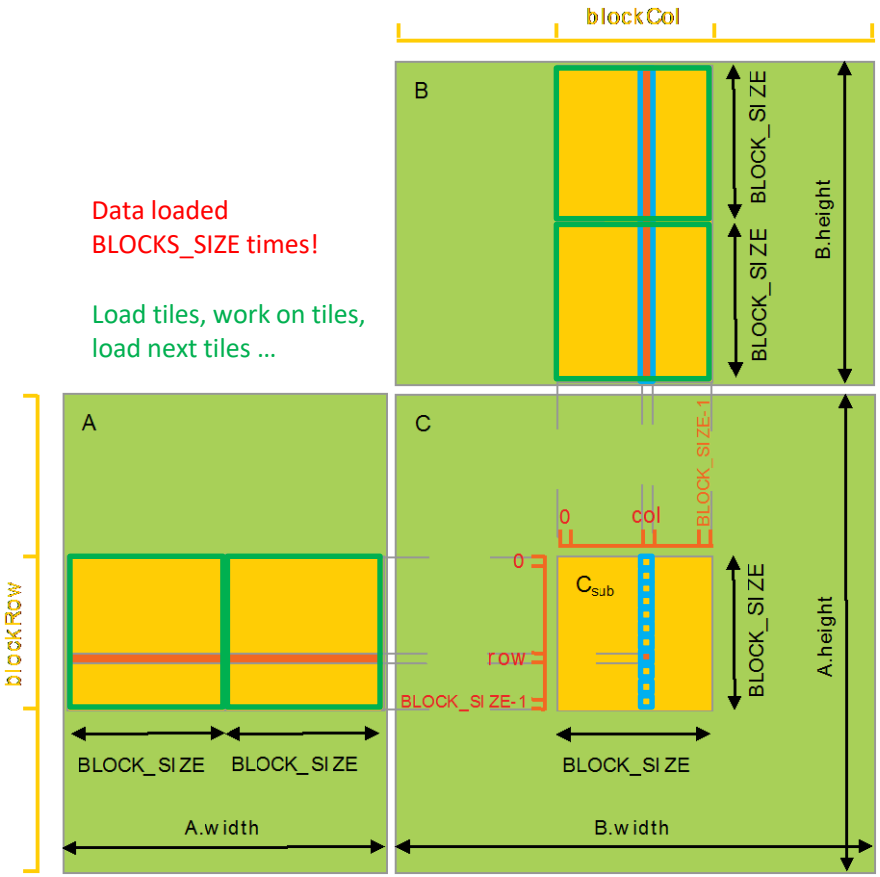
Matrix Multiplication: Problems

- A lot of memory access with little computations only
- Memory access is all going to slow global memory
- In a block the same memory is needed by multiple threads

→use shared memory to load one tile of data, consume the data together, advance to next block

Data loaded
BLOCKS_SIZE times!

Load tiles, work on tiles,
load next tiles …



```
68  __global__ void d_multiply_tiled(const float* A, int A_cols, int A_rows,
69                                   const float* B, int B_cols, int B_rows,
70                                   float* C)
71  {
72    __shared__ float s_A_TILE[TILE_WIDTH][TILE_WIDTH];
73    __shared__ float s_B_TILE[TILE_WIDTH][TILE_WIDTH];
74
75    int x = blockIdx.x*TILE_WIDTH + threadIdx.x;
76    int y = blockIdx.y*TILE_WIDTH + threadIdx.y;
77
78    float c_xy = 0;
79    int elements = A_cols;
80    int tiles = elements/TILE_WIDTH;
81    //loop over tiles
82    for(int tile = 0; tile < tiles; ++tile)
83    {
84        //load to shared
85        s_A_TILE[threadIdx.y][threadIdx.x] = A[y*A_cols + tile*TILE_WIDTH + threadIdx.x];
86        s_B_TILE[threadIdx.y][threadIdx.x] = B[(tile*TILE_WIDTH + threadIdx.y)*B_cols + x];
87
88        for(int i = 0; i < TILE_WIDTH; ++i)
89          c_xy += s_A_TILE[threadIdx.y][i] * s_B_TILE[i][threadIdx.x];
90
91    }
92    C[y*B_cols + x] = c_xy;
93  }
```

Blocksize: TILE_WIDTH x TILE_WIDTH

```
c_xy += A[y*A_cols + i]*B[i*B_cols + x];
```

E:\teaching\ezg2\documents\Cuda_lecture\matrixmultiplication\build\Release\matrixmultiplication.exe

```
going to use Tesla K20c
creating matrix A (800x1600)
creating matrix B (1600x800)
computing C on the GPU done in 0.0448499 seconds (0.0447236s on device)
computing C on the GPU in tiled mode done in 0.0204522 seconds (0.0203098s on de
vice)
computing difference
Max absolute difference: 266.558@ -148.537 118.021
Avg rel difference: 12.796
```

```
68  __global__ void d_multiply_tiled(const float* A, int A_cols, int A_rows,
69                                   const float* B, int B_cols, int B_rows,
70                                   float* C)
71  {
72    __shared__ float s_A_TILE[TILE_WIDTH][TILE_WIDTH];
73    __shared__ float s_B_TILE[TILE_WIDTH][TILE_WIDTH];
74
75    int x = blockIdx.x*TILE_WIDTH + threadIdx.x;
76    int y = blockIdx.y*TILE_WIDTH + threadIdx.y;
77
78    float c_xy = 0;
79    int elements = A_cols;
80    int tiles = elements/TILE_WIDTH;
81    //loop over tiles
82    for(int tile = 0; tile < tiles; ++tile)
83    {
84        //load to shared
85        s_A_TILE[threadIdx.y][threadIdx.x] = A[y*A_cols + tile*TILE_WIDTH + threadIdx.x];
86        s_B_TILE[threadIdx.y][threadIdx.x] = B[(tile*TILE_WIDTH + threadIdx.y)*B_cols + x];
87        __syncthreads();
88        for(int i = 0; i < TILE_WIDTH; ++i)
89          c_xy += s_A_TILE[threadIdx.y][i] * s_B_TILE[i][threadIdx.x];
90
91    }
92    C[y*B_cols + x] = c_xy;
93  }
```

Read from another thread before loaded!!

E:\teaching\ezg2\documents\Cuda_lecture\matrixmultiplication\build\Release\matrixmultiplication.exe

```
going to use Tesla K20c
creating matrix A (800x1600)
creating matrix B (1600x800)
computing C on the GPU done in 0.0448634 seconds (0.0447364s on device)
computing C on the GPU in tiled mode done in 0.0207124 seconds (0.0204656s on de
vice)
computing difference
Max absolute difference: 30.3438@ 34.7161 4.37228
Avg rel difference: 0.571838
```

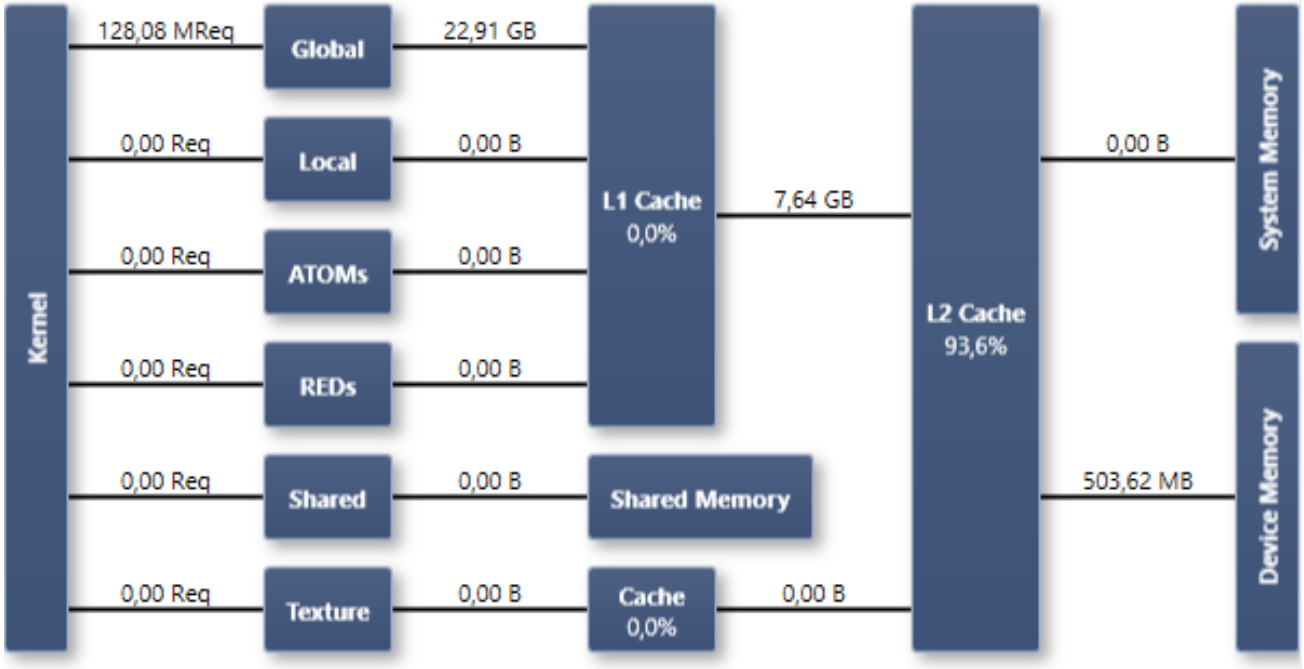
```
68  __global__ void d_multiply_tiled(const float* A, int A_cols, int A_rows,
69                                   const float* B, int B_cols, int B_rows,
70                                   float* C)
71  {
72    __shared__ float s_A_TILE[TILE_WIDTH][TILE_WIDTH];
73    __shared__ float s_B_TILE[TILE_WIDTH][TILE_WIDTH];
74
75    int x = blockIdx.x*TILE_WIDTH + threadIdx.x;
76    int y = blockIdx.y*TILE_WIDTH + threadIdx.y;
77
78    float c_xy = 0;
79    int elements = A_cols;
80    int tiles = elements/TILE_WIDTH;
81    //loop over tiles
82    for(int tile = 0; tile < tiles; ++tile)
83    {
84        //load to shared
85        s_A_TILE[threadIdx.y][threadIdx.x] = A[y*A_cols + tile*TILE_WIDTH + threadIdx.x];
86        s_B_TILE[threadIdx.y][threadIdx.x] = B[(tile*TILE_WIDTH + threadIdx.y)*B_cols + x];
87        __syncthreads();
88        for(int i = 0; i < TILE_WIDTH; ++i)
89          c_xy += s_A_TILE[threadIdx.y][i] * s_B_TILE[i][threadIdx.x];
90        __syncthreads();
91    }
92    C[y*B_cols + x] = c_xy;
93  }
```

Overwritten before read from other thread!!

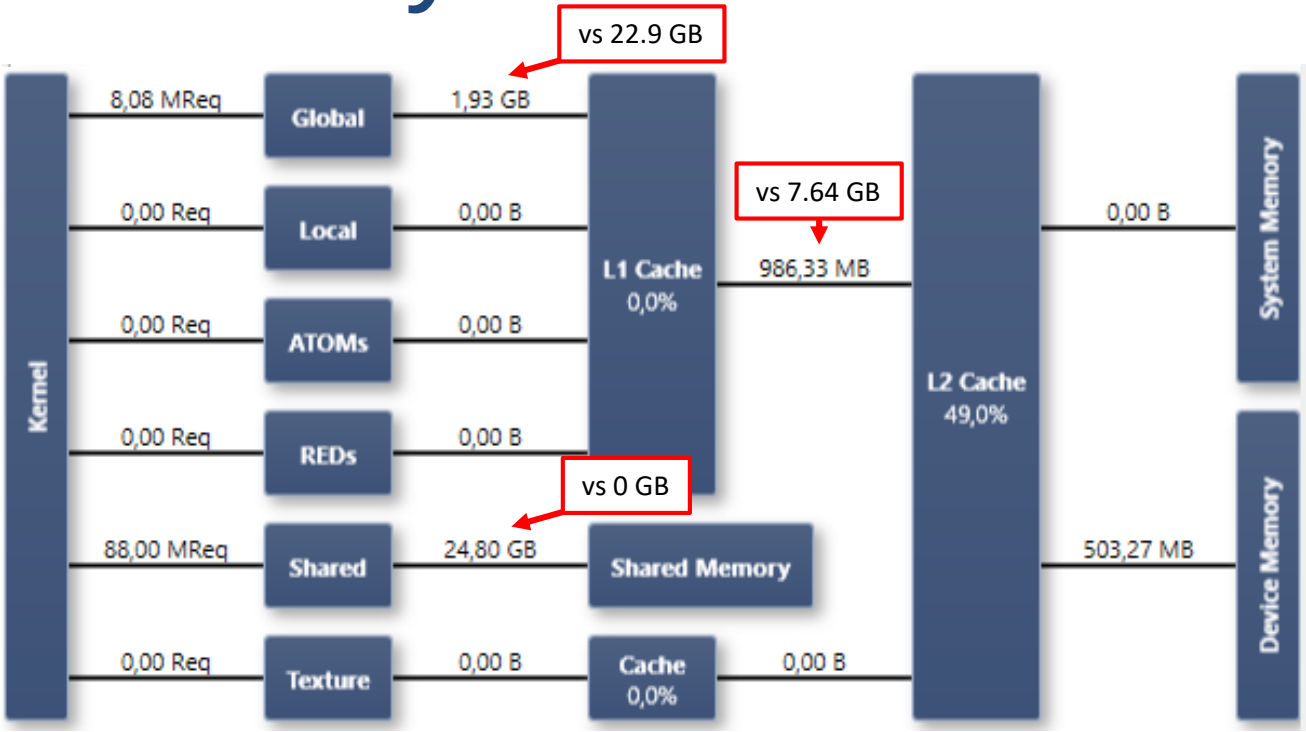
E:\teaching\ezg2\documents\Cuda_lecture\matrixmultiplication\build\Release\matrixmultiplication.exe

```
going to use Tesla K20c
creating matrix A (800x1600)
creating matrix B (1600x800)
computing C on the GPU done in 0.0448535 seconds (0.0447168s on device)
computing C on the GPU in tiled mode done in 0.0205898 seconds (0.0204427s on de
vice)
computing difference
Max absolute difference: 0@ 0 0
Avg rel difference: 0
```

Memory Statistics: Non-Tiled



Memory Statistics: Tiled



Questions