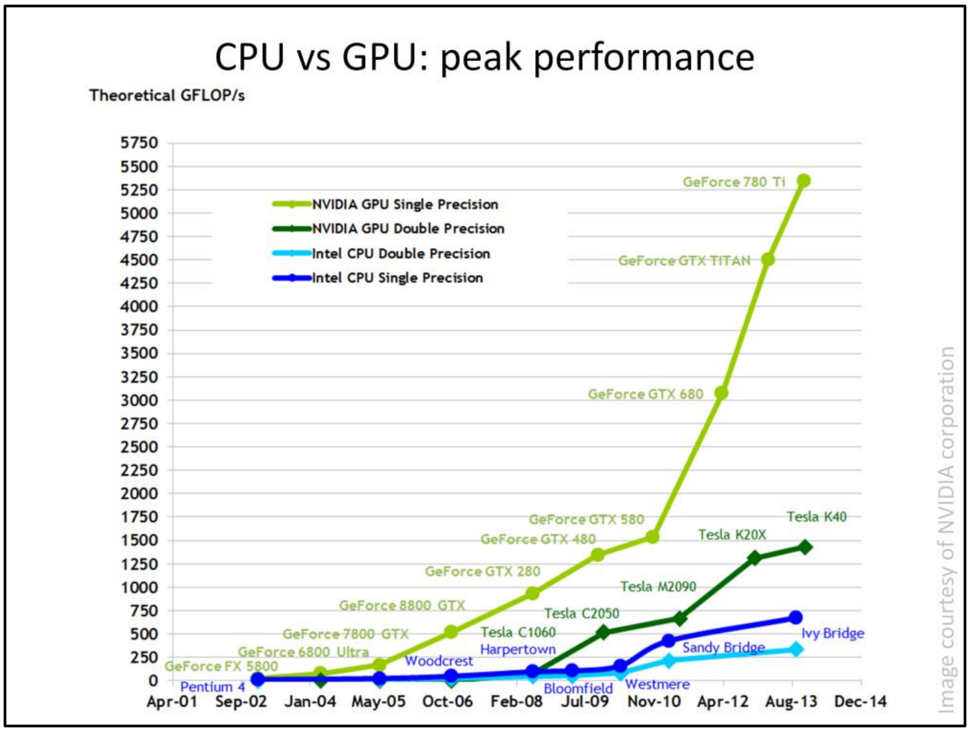
Chapter 13: Graphics Processing Units (GPUs)

Chapter 13: outline

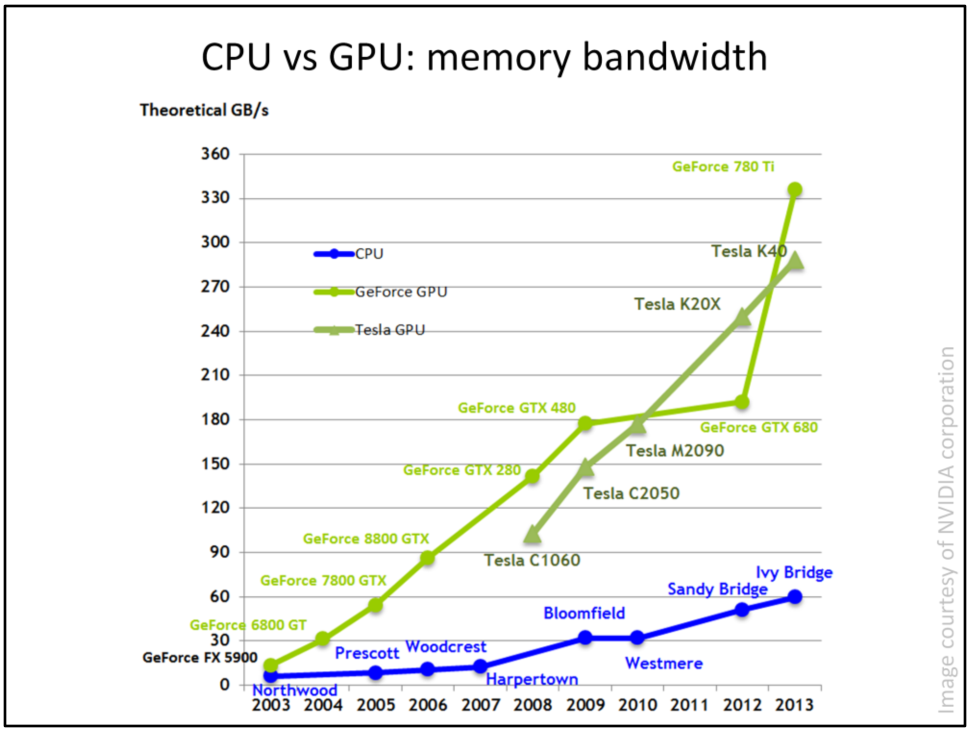
- CPUs versus GPUs: performance
    - FLOPS/s
    - Memory bandwidth
- Programming model
    - Blocks
    - Threads
    - Kernels
- Hardware architecture
    - Warps – SIMT
    - Memory hierarchy
        - Global, local, shared memory
        - Bank conflicts
- Examples (live demos)



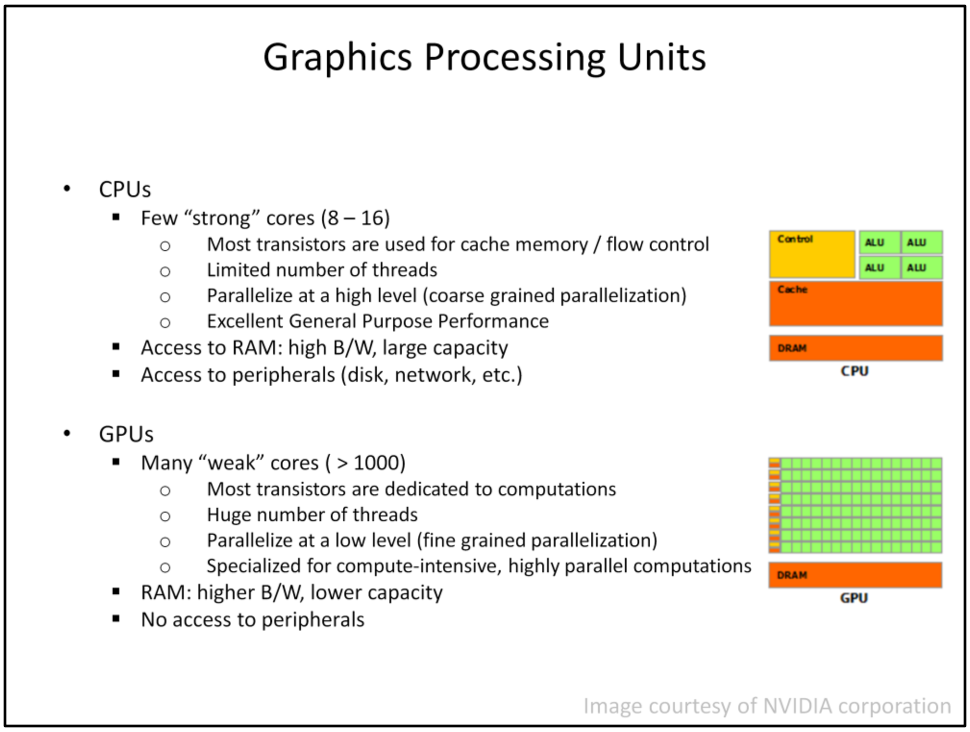
How are these numbers obtained?

For CPUs, an Ivy Bridge architecture can produce 16 single precision (SP) or 8 double precision (DP) results per cycle per core, using AVX-256 instructions and registers. The most powerful Ivy Bridge CPU on the market is (at the time of writing) the Intel Xeon E7-2890 v2 featuring 15 CPU cores running at 2.8 GHz. In total, this amounts to a peak performance of 672 GFLOPS/s for SP and 336 GFLOPS/s for DP.

For GPUs, a top model Tesla K40 has 15 Streaming Multiprocessors (SM), each containing 64 DP ALU (Arithmetic Logic Unit) that are each capable of delivering two DP results per cycle (in fact: one fused multiply add (FMA) instruction). At 745 MHz one obtains 1430.4 GFLOPS/s.

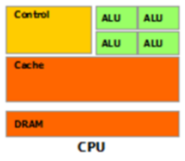


As an important note: the amount of memory on a GPU (a few GByte) is typically much lower than the available RAM on a worker node (tens of GByte for a modern workstation).



Graphics Processing Units

- CPUs
  - Few “strong” cores (8 – 16)
    - Most transistors are used for cache memory / flow control
    - Limited number of threads
    - Parallelize at a high level (coarse grained parallelization)
    - Excellent General Purpose Performance
  - Access to RAM: high B/W, large capacity
  - Access to peripherals (disk, network, etc.)



- GPUs
  - Many “weak” cores (> 1000)
    - Most transistors are dedicated to computations
    - Huge number of threads
    - Parallelize at a low level (fine grained parallelization)
    - Specialized for compute-intensive, highly parallel computations
  - RAM: higher B/W, lower capacity
  - No access to peripherals

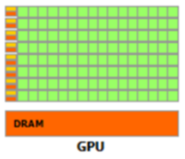
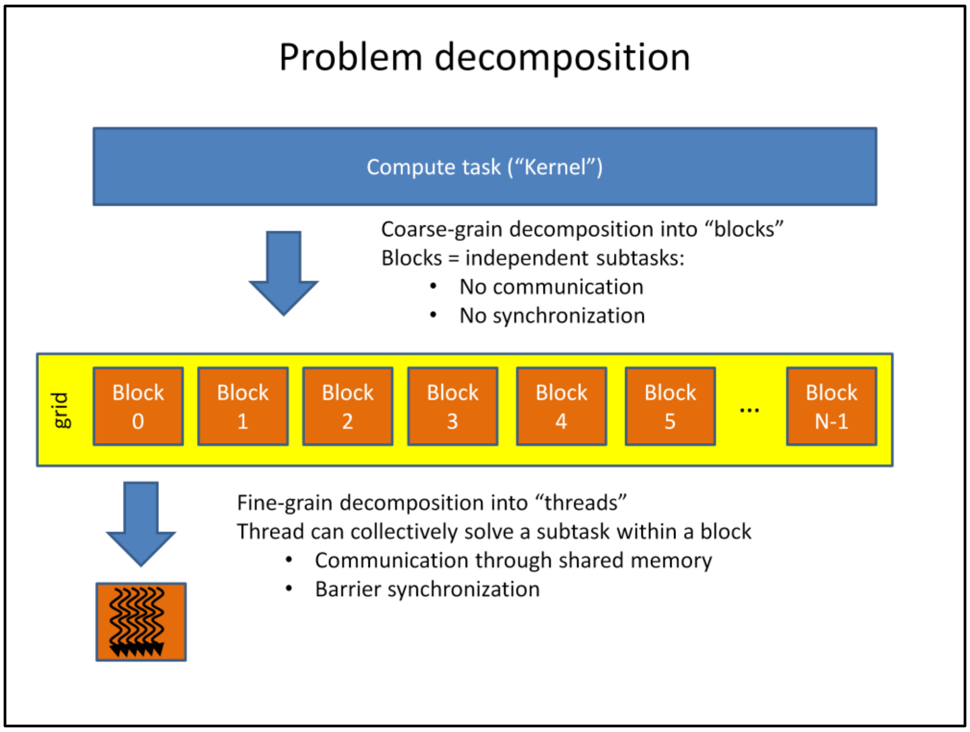


Image courtesy of NVIDIA corporation

GPUs are designed for 3D graphics to do TCL (Transformation, Clipping, Lighting) and rendering (texture mapping). These are highly parallel computations as the transformation of vertices (triangles) is parallel by vertex (triangle) for the TCL part and because the computation of the color value for each pixel on the screen is parallel by pixel for the rendering step.

Starting from the year 2000, GPUs featured programmable shaders (= rendering technique to control the exact value of a pixel, e.g. lighting and depth effects and so on) which were massive parallel, but really simple compute cores on the a GPU. At a certain point in time, people started to “abuse” these programmable shaders in order to accelerate highly parallel computations. Companies like NVIDIA realized this extra potential of their GPUs and introduced CUDA (Compute Unified Device Architecture) as a language extension to C in order to instruct GPUs to perform certain computations. The “Tesla” series of GPUs are not intended for gaming, but rather for scientific computing. These modern GPUs support the IEEE 745 floating point format, including double precision (DP) computations and special function (sin, cos, exp, and many more) evaluation.



Problem decomposition

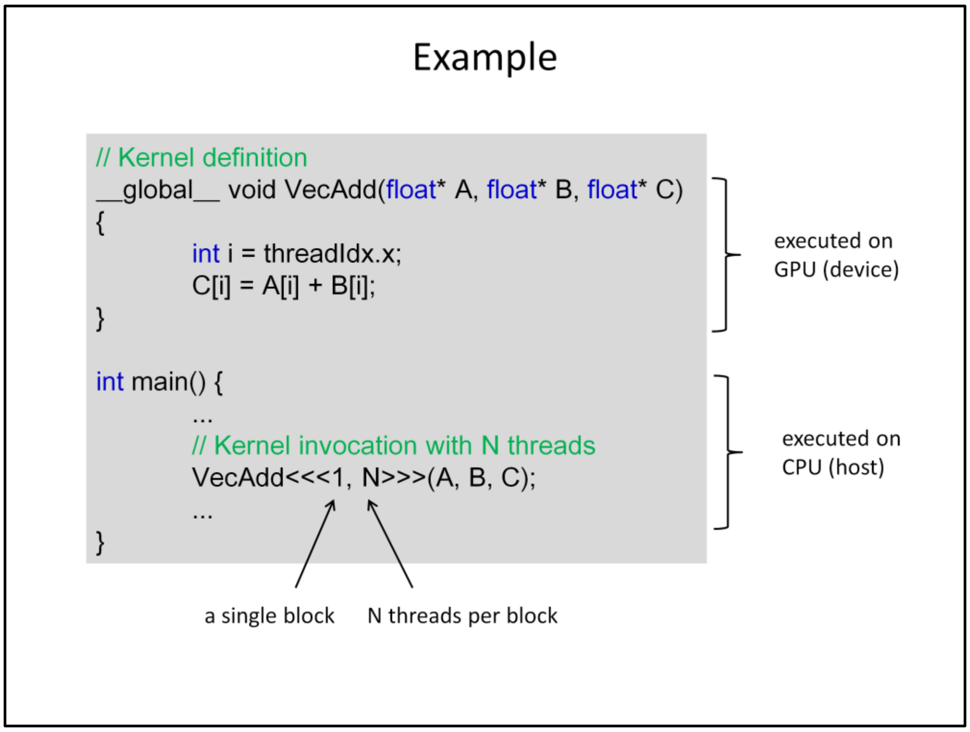
Coarse-grain decomposition into "blocks"
Blocks = independent subtasks:

- No communication
- No synchronization

Fine-grain decomposition into "threads"
Thread can collectively solve a subtask within a block

- Communication through shared memory
- Barrier synchronization

A computational problem that must be solved using GPUs should first be partitioned into coarse sub-problems that can be solved independently in parallel by blocks of threads. Each sub-problem can be further parallelized into finer pieces that can be solved cooperatively in parallel by all threads within the block.



Example

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

int main() {
        ...
        // Kernel invocation with N threads
        VecAdd<<<1, N>>>(A, B, C);
        ...
}
```

executed on
GPU (device)
executed on
CPU (host)
a single block
N threads per block

In this first CUDA example, two vectors A and B of size N are added. The result is stored in vector C. Note that pointers A, B and C refer to memory that is allocated on the GPU (code not shown). Trying to dereference A, B or C on the host is meaningless and will likely result in a segmentation fault.

Workload to be executed on the GPU is described in “Kernels”, declared with the __global__ declaration specifier. Kernels define functions to be executed by 100’s or 1000’s of threads at a time. Kernels return “void”, they cannot take a variable number of input parameters and they cannot access host memory. They are invoked by the <<<…>>> execution configuration syntax. The first argument is the number of blocks, the second argument the number of threads per block. In this case, there is only a single block with N threads. Note that the number of threads per block is limited (typically 1024 -- see further). The number of blocks can be very large (see example next slide). This also means that in this example, N is limited to 1024.

Each thread in a block gets a unique **thread identifier** between [0… N-1], with N the number of threads per block. The built-in variable threadIdx (**thread index**) can be used for this purpose. In general, threadIdx is a “dim3” structure with three fields; x, y and z that can be used for one-dimensional (as in this example), two-dimensional (see next slide) and three-dimensional indexing.

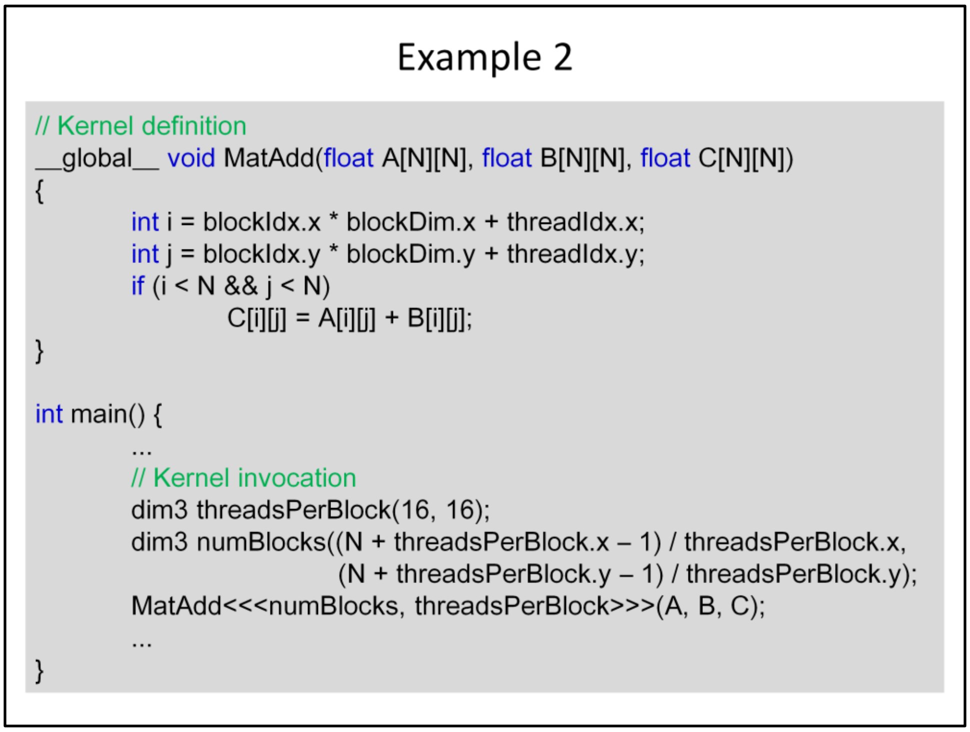
The relation between the thread identifier and thread index is straightforward:

a) 1D indexing: thread identifier = threadIdx.x
b) 2D indexing: thread identifier = threadIdx.x + blockDim.x * threadIdx.y
c) 3D indexing: thread identifier = threadIdx.x + blockDim.x * threadIdx.y + blockDim.x * blockDim.y * threadIdx.z

Where blockDim is again a built-in dim3 structure that contains the dimensions of the thread indices within a block (see next slide)

The __global__ function qualifier denotes kernels that run on the device, but which are called by the host
The __device__ function qualifier denotes kernels that run on the device and are called by the device (a.k.a. kernel helper functions)
The __host__ function qualifier denotes functions that run on the host. In case no qualifier is used, __host__ is assumed by the compiler.

Memory is managed on the device with
- cudaMalloc(void **devPtr, size_t sizeInBytes)
- cudaFree(void **devPtr, size_t sizeInBytes)
- cudaMemcpy(void *dest, void *src, size_t sizeInBytes, enum direction)
    - Where direction can be {cudaMemcpyDeviceToHost, cudaMemcpyHostToDevice}



Example 2

```
// Kernel definition
__global__ void MatAdd(float A[N][N], float B[N][N], float C[N][N])
{
        int i = blockIdx.x * blockDim.x + threadIdx.x;
        int j = blockIdx.y * blockDim.y + threadIdx.y;
        if (i < N && j < N)
                C[i][j] = A[i][j] + B[i][j];
}

int main() {
        ...
        // Kernel invocation
        dim3 threadsPerBlock(16, 16);
        dim3 numBlocks((N + threadsPerBlock.x – 1) / threadsPerBlock.x,
                       (N + threadsPerBlock.y – 1) / threadsPerBlock.y);
        MatAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
        ...
}
```

In this example, NxN matrices A and B are added and the result is stored in C. Note that in this case, we use two-dimensional indexing of A, B and C to illustrate the convenience of two-dimensional thread indices within a block.

In this example, and in contrast to the previous slide, we execute multiple blocks to allow arbitrary dimensions of N. An NxN matrix is partitioned into a two-dimensional “grid” of thread blocks, where each block itself executes as a two-dimensional array of 16 x 16 threads.

To summarize:

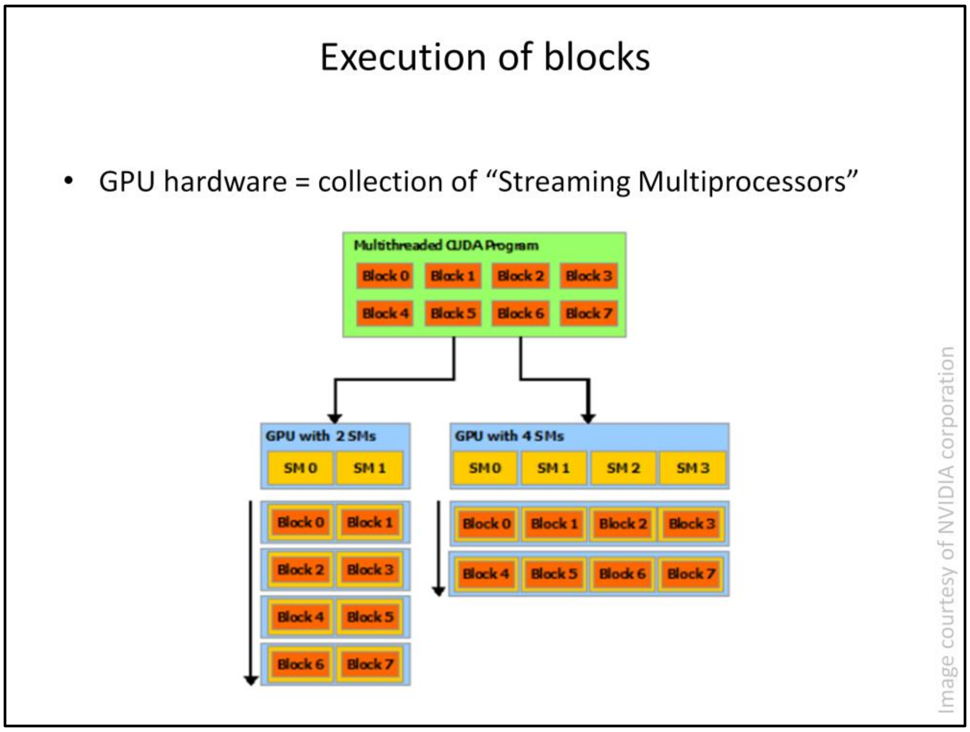
- Threads: single execution units that run kernels on a GPU.
- Thread Blocks: collection of threads (up to 1024) that are scheduled and executed together by definition on a single Streaming Multiprocessor. These threads *can* synchronize, communicate and are organized in a 1D, 2D or 3D structure.
- Grid: a collection of thread blocks that should be independent (no communication / synchronization). Thread block can be executed sequentially or in parallel, depending on the available number of Streaming Multiprocessors in a GPU and the capacity of a Streaming Multiprocessor (modern SMs can handle several thread blocks in parallel). The execution of thread blocks however is completely scheduled by the hardware and the

user should assume no specific execution order. Thread blocks are again organized in a 1D, 2D or 3D structure.

Built-in variables that one can use within a kernel:

- threadIdx: dim3 structure that contains the thread index within a block
- blockIdx: dim3 structure that contains the block index within the grid
- blockDim: dim3 structure that contains the block dimensions (expressed in number of threads)
- gridDim: dim3 structure that contains the grid dimensions (expressed in number of blocks)

The pattern “blockIdx.x * blockDim.x + threadIdx.x” (and similarly for y and z) is very commonly used within a kernel to compute in the index of an element to work on. Additionally, the pattern “if (index < N)” is commonly used within a kernel in case the workload size N is not perfectly dividable by the number of threadsPerBlock.



Execution of blocks

- GPU hardware = collection of “Streaming Multiprocessors”

A single block is run on a single “Streaming Multiprocessor” (SM), a scalable unit of compute hardware on a GPU that can execute a number of threads simultaneously. SMs are sometimes seen as the equivalent of CPU cores. This is true in that sense that GPU manufacturers can flexibly chose the number of SMs on a GPU chip, just as CPU vendors can chose the number of cores on a CPU chip. There are important differences however, for example, a SM itself consists of many small cores (> 100 for high-end hardware). Hence, there is no real 1 to 1 analogy between SMs and CPU hardware.

The example shows a problem parallelized into 8 blocks that is run on a GPU with 2 SMs and 4 SMs, respectively.

Heterogeneous computing

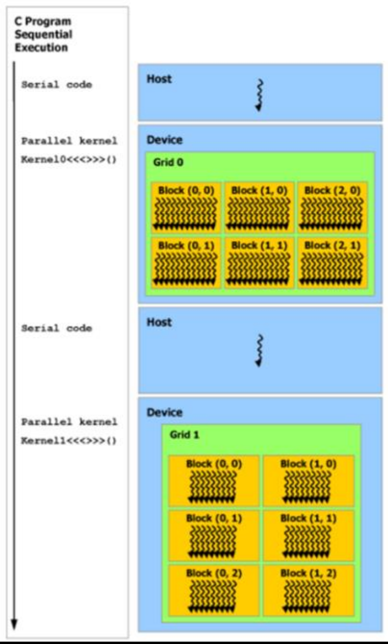
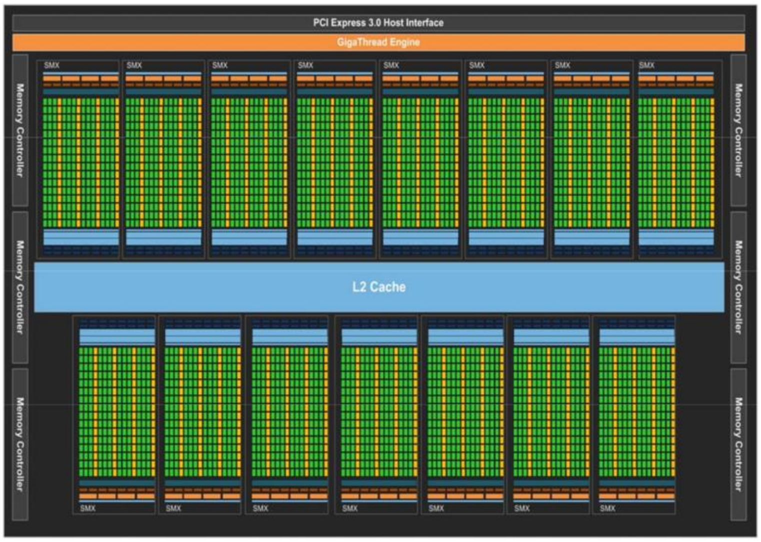


Image courtesy of NVIDIA corporation

A CUDA program typically consists of a number of Kernel calls executed on a GPU with “regular” CPU code in between. GPU and CPU computations can overlap.

GK110/GK210 Kepler Architecture

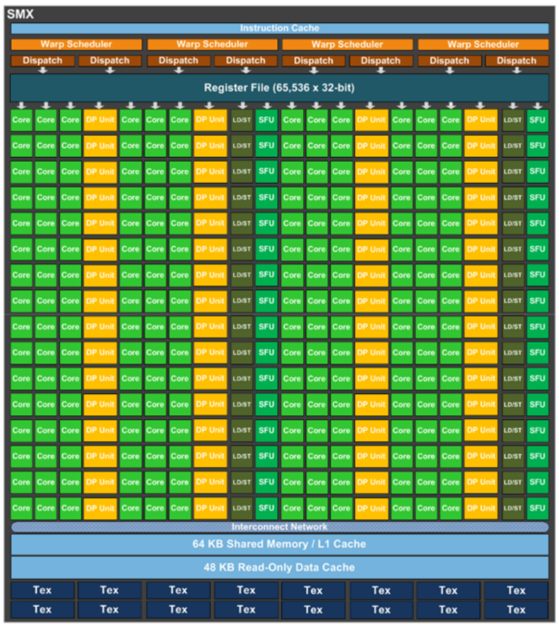


"**Streaming Multiprocessor**" (SM): executes "**thread blocks**"

Image courtesy of NVIDIA corporation

The latest Kepler architecture by Nvidia is designed primarily for the Tesla series of accelerators (scientific computing). The Kepler GK110 and GK210 implementation features 15 SM per chip and six 64-bit memory controllers to access the card's memory with very high bandwidth. Memory access is cached in L2 cache which is shared by all SMs.

GK110/GK210 Streaming Multiprocessor


SMX
Instruction Cache
Warp Scheduler
Dispatch
Register File (65,536 x 32-bit)
Core
DP Unit
LD/ST
SFU
Interconnect Network
64 KB Shared Memory / L1 Cache
48 KB Read-Only Data Cache
Tex

Per Streaming Multiprocessor:

- 192 SP Cuda Cores
- 64 DP Units
- 32 Special function units
- 32 Load/store units
- 64 kB L1 cache/shared memory

Per Streaming Multiprocessor:

- Can handle up to 2048 threads
- Can handle up to 16 blocks
- Can issue max. 2 instructions per warps, for four warps simultaneously

Each SM contains a large number of cores, arithmetic units, special function units, etc. Additionally, it contains 64 kB of L1 cache that can also be used a shared memory to communicate between threads (see further). A streaming multiprocessor can **handle** up to 16 thread blocks or 2048 threads. For example, it can handle e.g. two thread blocks with 1024 threads each, or 16 thread blocks with 128 threads each. In reality, the number of thread blocks that are handled by a SM can be lower, in case the thread block require e.g. more shared memory than can delivered by a SM. The hardware automatically computes for the user how blocks are scheduled and executed on SMs.

Note that it is not true that up to 2048 threads can execute concurrently. Internally, threads within a block are grouped in “warps”, i.e., a collection of 32 threads which will all perform exactly the same instruction (SIMT, see further). Per clock cycle, this SM **can** select four warps and two independent instructions **can** be dispatched per warp (8 independent instructions per cycle). In principle, 4 warps x 32 threads/warp x 2 instructions = 256 instructions can be processed per cycle. Note that this assumes that there are sufficient hardware units available on the SM. For example, a SM will never be able to process 256 double precision DP instructions simultaneously, as there are only 64 DP hardware units. It is possible however, in principle, to process 64 DP instructions and 192 non-DP (logic / SP / integer) instructions simultaneously.

The reason why a SM handled more warps than it can actually schedule per cycle is to hide the latency. When e.g. four warps load data from the global memory, the latency can be overlapped by executing instructions from other warps. This principle is very similar to Simultaneous Multithreading(SMT) (see Chapter 3) used in CPUs (Intel name: Hyperthreading).

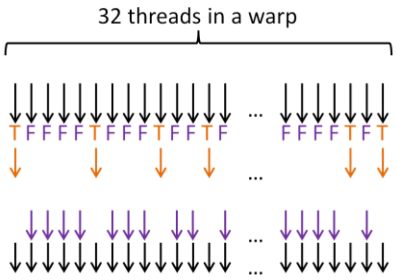
SIMT architecture

Streaming Multiprocessor (SM)

- Multiple thread blocks can be executed concurrently per SM
  - Depending on required resources per thread block
- Threads within a block will execute concurrently per SM
- Designed to execute hundreds of threads concurrently
- Uses the **SIMT (Single-Instruction, Multiple-Thread)** architecture
  - Threads are scheduled, managed & created per warp (32 threads)
  - All threads per warp execute a common instruction at a time
  - Brand divergence occurs in case of data-dependent conditional branches (see example next slide).

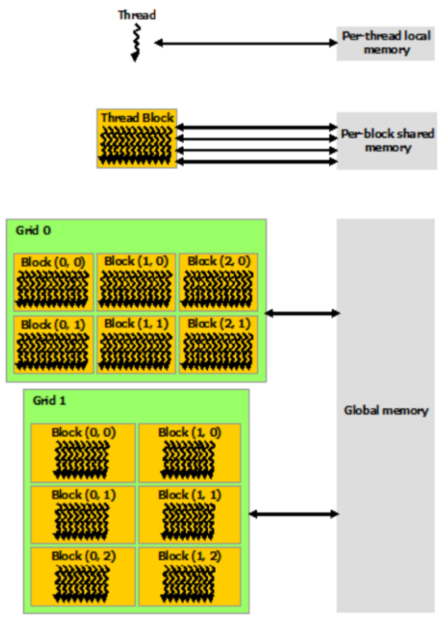
Branch divergence example

```
// Kernel definition
__global__ void Kernel(float *a)
{
        int i = threadIdx.x;
        if (a[i] > 0)
                // do something
        else
                // do something else
        …
}
```



Branch diversion causes threads in a warp to be serialized in case they take different execution paths.

Memory hierarchy (logical view)



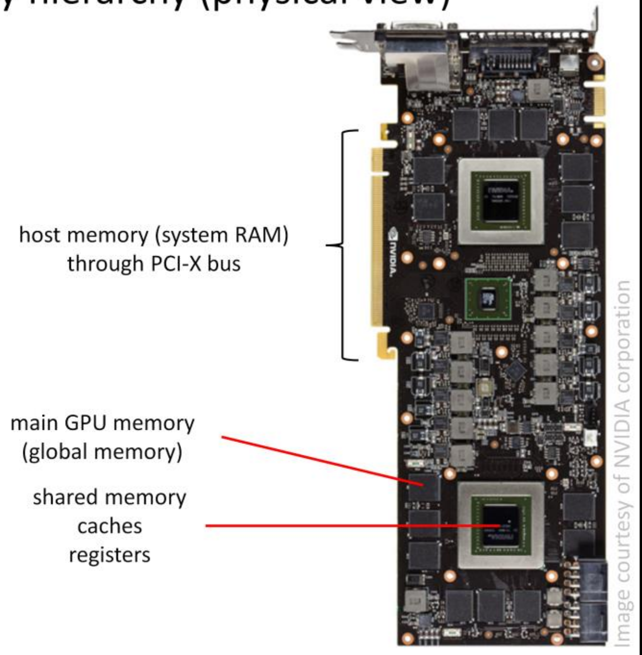
private local memory + registers
accessible per thread
lifetime: duration of that thread

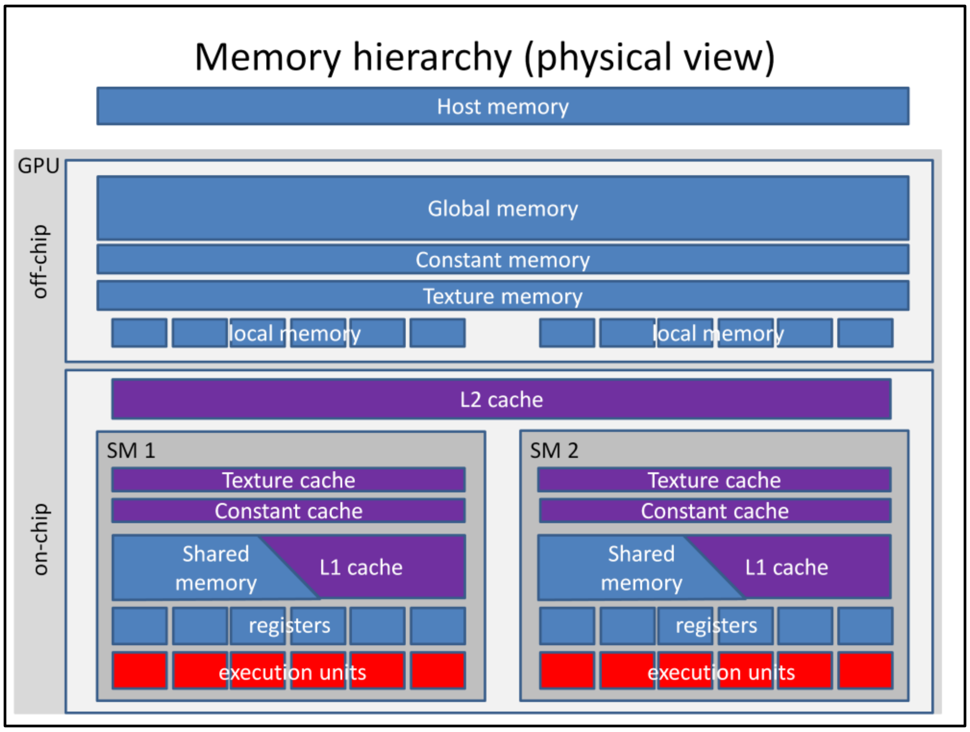
shared memory
accessible by all threads in a block
lifetime: duration of that block

global, constant and texture memory
accessible by all threads in all blocks
lifetime: duration of the application

This diagram shows the logical memory hierarchy. Note that this does not entirely correspond to a physical layout (see next slides)

Memory hierarchy (physical view)



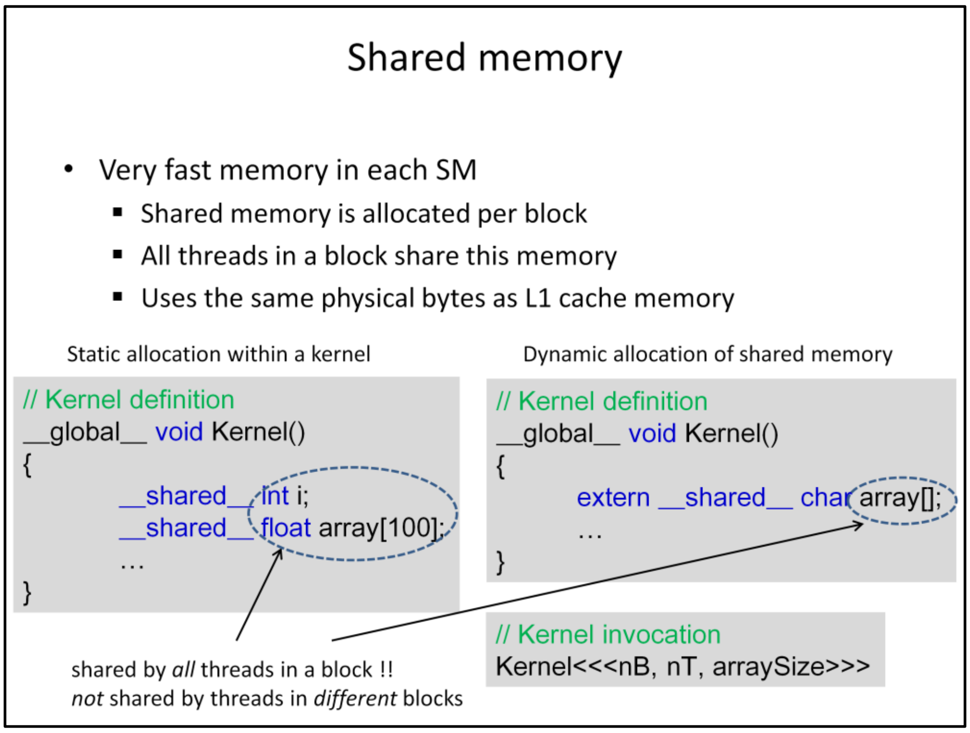


Note that the names “global”, “local”, “shared” refer to the scope of the memory and not to the proximity of the CUDA cores.

- **Global memory** is part of the main memory of the GPU (off-chip memory) and can be accessed by all threads running on the GPU and CPU. Memory is managed from the host side with cudaMalloc, cudaFree and cudaMemcpy, or from the device side using malloc() and free(). It is byte addressable and persistent across kernel calls (when allocated from the host-side)
- **Constant memory** is read-only memory that is part of the main memory that is not cached by the L2 and L1 caches, but rather by specific constant caches per SM. It is small (typically 64 kB) and contains constant values set by the host. It is designed for very fast broadcast operations, that is, running threads in a warp can read exactly the same address with no delays.
- **Texture memory** is read-only memory that is part of the device memory with again its own cache. It is designed for fetching textures during 3D rendering. It has very fast 2D addressing capabilities and automatic interpolation in case a non-integer (floating point) index is fetched with no delay.
- The L2 cache is on-chip memory that is shared by all multiprocessors to cache local and global memory access, including those by the CPU. Texture and constant memory have their own separate caches per Streaming Multiprocessor that is optimized for 2D spatial locality.
- **Local memory** is part of the main memory that is used automatically by the CUDA

compiler in case insufficient registers are available on the Streaming Multiprocessors (register spilling). The scope is per thread (that’s why it is drawn in boxes).

- The L1 cache is typically 64 KB per SM that is very fast (register speed) and represents exactly the same bytes as the shared memory. It can be configured to be e.g. 48 kB of shared memory and 16 kB of L1 or 16 kB of shared memory and 48 kB of L1. This feature does not exist in CPUs.
- **Shared memory** is designed to allow for very fast communication between threads, if needed. When it is not needed, the available 64 kB are used as L1 cache. The programmer can chose to use either 0 bytes, 16 kB, 48 kB or 64 kB as shared memory. It is organized in dwords (4 bytes) that reside in 32 different banks. Different threads in a warp cannot access the same bank at the same time (see further).
- Registers are the fastest memory. They contain variables declared in a kernel, unless there are too many of them in which case local memory will be used. One should always try to use as few registers as possible, because this determines the ultimate number of block that will run concurrently on a SM (up to 16).



Shared memory

- Very fast memory in each SM
  - Shared memory is allocated per block
  - All threads in a block share this memory
  - Uses the same physical bytes as L1 cache memory

Static allocation within a kernel

```
// Kernel definition
__global__ void Kernel()
{
        __shared__ int i;
        __shared__ float array[100];
        …
}
```

Dynamic allocation of shared memory

```
// Kernel definition
__global__ void Kernel()
{
        extern __shared__ char array[];
        …
}
```

```
// Kernel invocation
Kernel<<<nB, nT, arraySize>>>
```

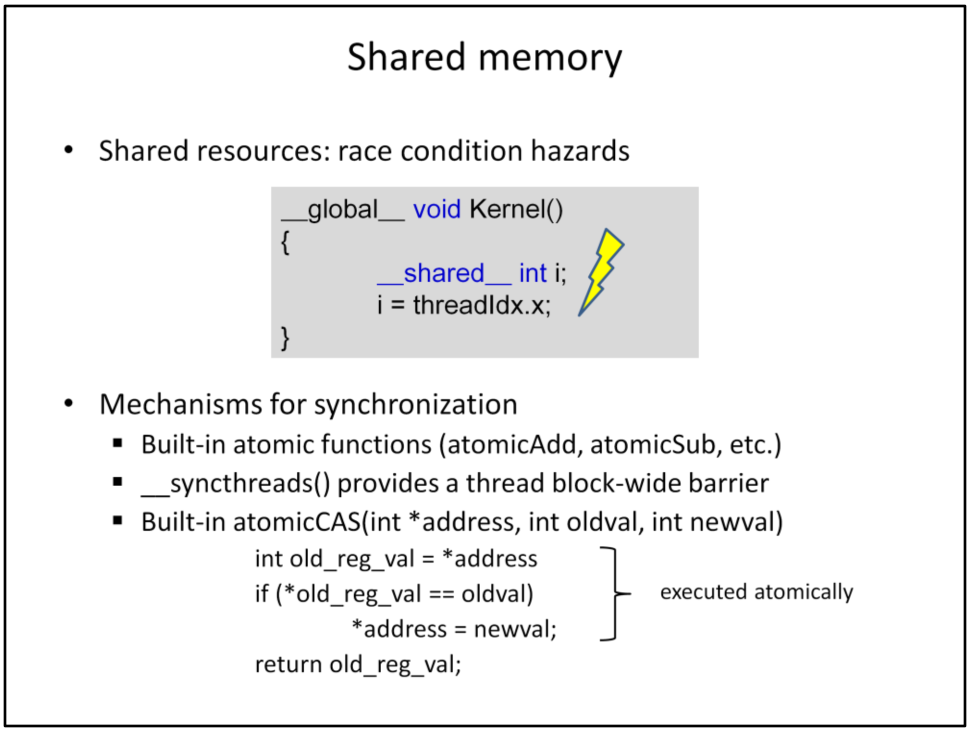
shared by *all* threads in a block !!
not shared by threads in *different* blocks

One can set the amount of shared and L1 cache as follows:

cudaFuncSetCacheConfig(kernelName, enum cudaFuncCache);

where cudaFuncCache is one of the following values:

CudaFuncCachePreferNone	0	(Default)
CudaFuncCachePreferShared	1	48 kB of shared memory and 16 kB of L1 cache
CudaFuncCachePreferL1	2	16 kB of shared memory and 48 kB of L1 cache
CudaFuncCachePreferEqual	3	32 kB of shared memory and 32 kB of L1 cache



Shared memory

- Shared resources: race condition hazards

```
__global__ void Kernel()
{
        __shared__ int i;
        i = threadIdx.x;
}
```

- Mechanisms for synchronization
  - Built-in atomic functions (atomicAdd, atomicSub, etc.)
  - __syncthreads() provides a thread block-wide barrier
  - Built-in atomicCAS(int *address, int oldval, int newval)

```
int old_reg_val = *address          ⎫
if (*old_reg_val == oldval)         ⎬ executed atomically
        *address = newval;          ⎭
return old_reg_val;
```

Note that race conditions might also arise in global memory, when different threads are writing to the same memory location, e.g.:

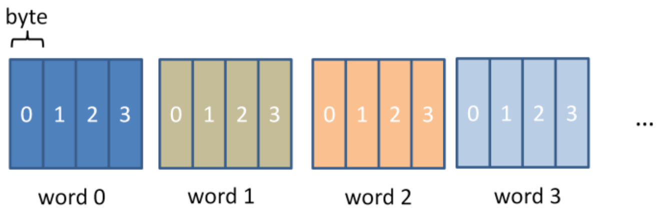
```
__global__ void Kernel(int *a)
{
        *a += 1;        // race condition
}

int main()
{
        int a = 0, *a_d;

        cudaMalloc((void**) &a_d, sizeof(int));
        cudaMemcpy(a_d, &a, sizeof(int), cudaMemcpyHostToDevice);

        Kernel<<<1000, 1000>>(a_d);

        …
}
```

Atomic CAS (Compare and Swap) operations have many applications, both on CPUs and on GPUs. They write a certain value “val” to a memory location “address”, but only if the value at “address” is still equal to an earlier read value “compare”. It is typically used in the following context:

```
do {
                compare = *address;    // read value at address
                val = computeNewValue(compare);   // compute new value, based on
the old value
} while (atomicCAS(address, compare, val) != compare)   // write the new value to
*address, but only if no other thread changed
```

CAS operations can be used to implement Mutexes and Semaphores on GPUs, however, per-thread synchronization is extremely costly for GPUs (see slides on the SIMT model). CAS operations are also frequently used on CPUs, to implement semaphores, mutexes and lock-free datastructures.

Shared memory organization

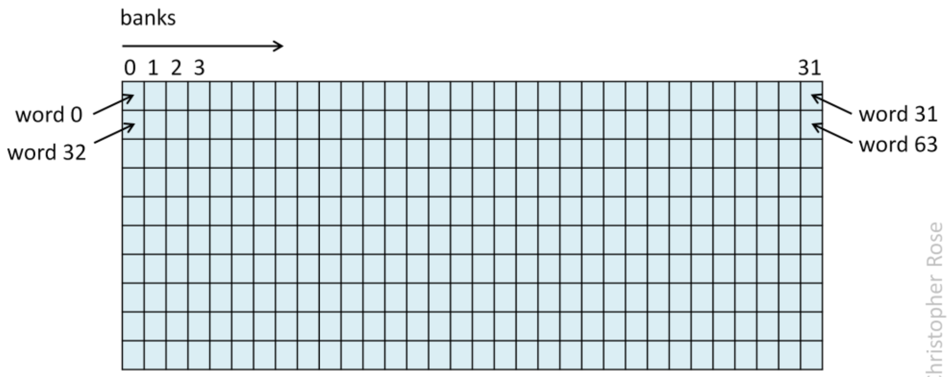


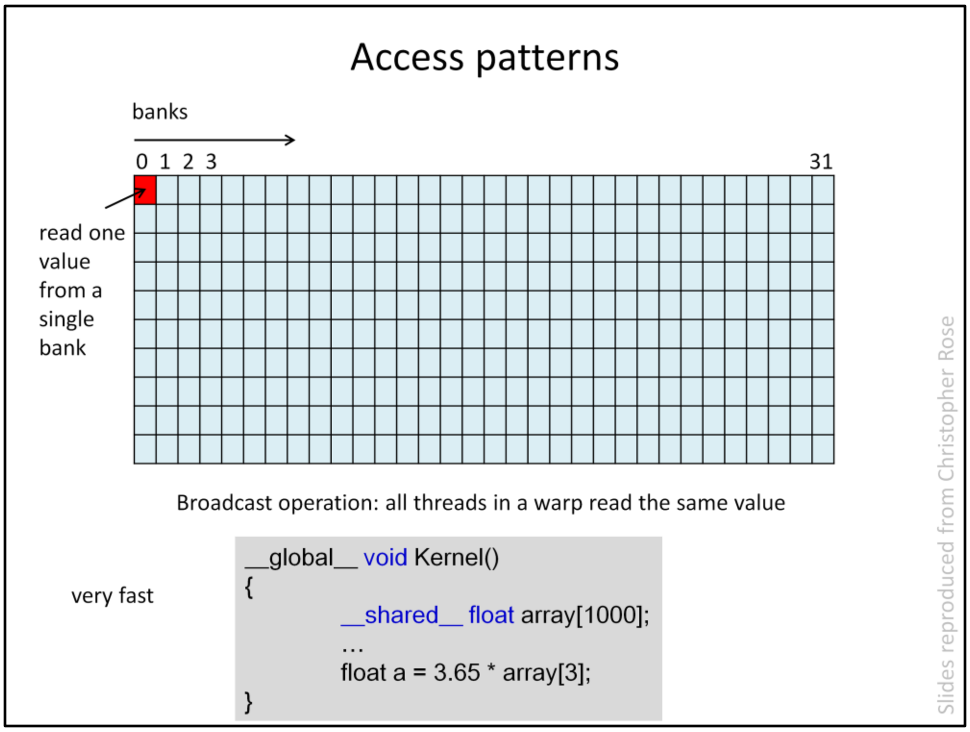
- Shared memory is organized and accessed in 4 byte words
  - A word can be an integer, float, 4 bytes, half a double, etc.
- Words are organized in 32 banks [0 … 31]
  - Words 0, 32, 64, 96 belong to bank 0
  - Words 1, 33, 65, 97 belong to bank 1
  - etc.

Shared memory access

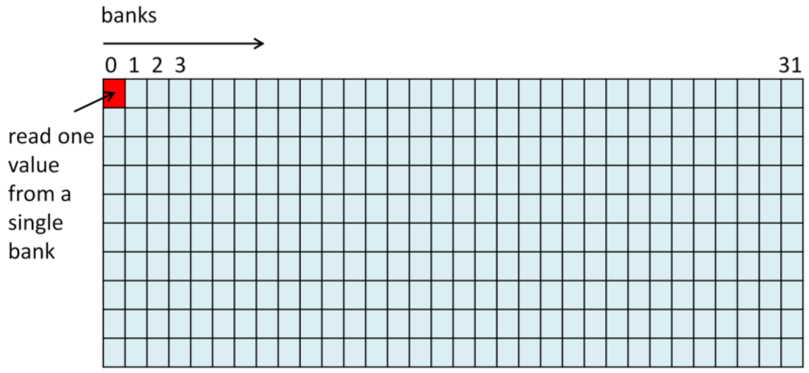
- Threads in a warp execute the same instruction (or are idle)
  - This also means they might concurrently access shared memory
- Several possibilities:
  - All threads in a warp read the same word: “broadcast”
    - Memory is accessed once, followed by a broadcast to the threads
  - Some threads in a warp read the same word: “multicast”
    - Memory is accessed once and copies to a subset of the threads
  - Threads in a warp that read different words
    - Access patterns where different threads access different banks can happen concurrently
    - Otherwise: bank conflicts (danger of running slower)

Access patterns





Access patterns

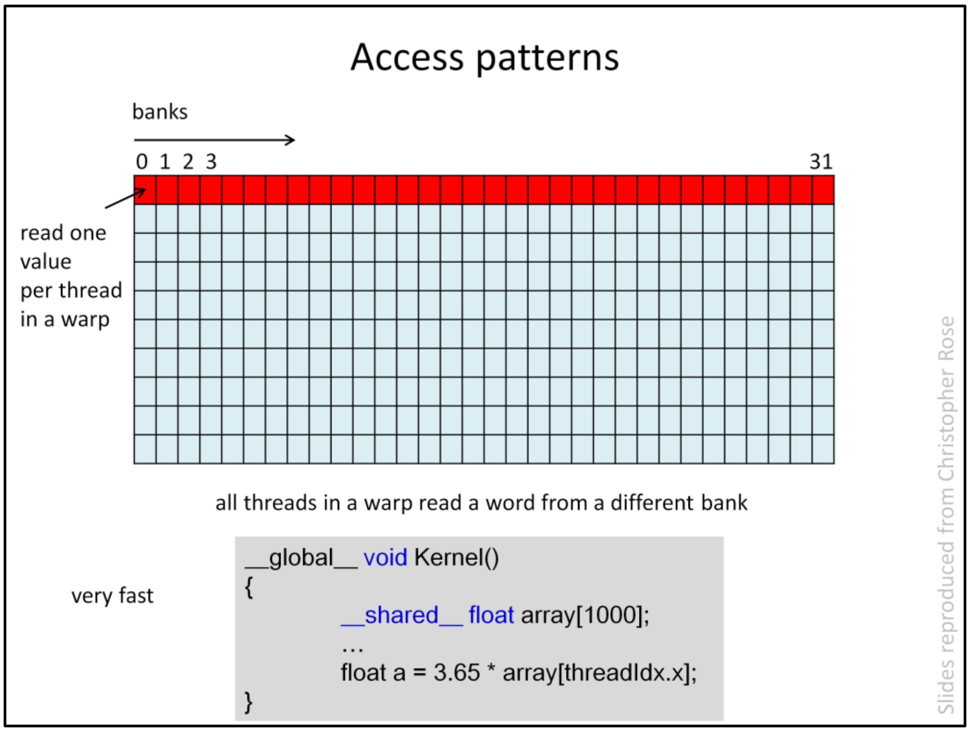


Broadcast operation: all threads in a warp read the same value

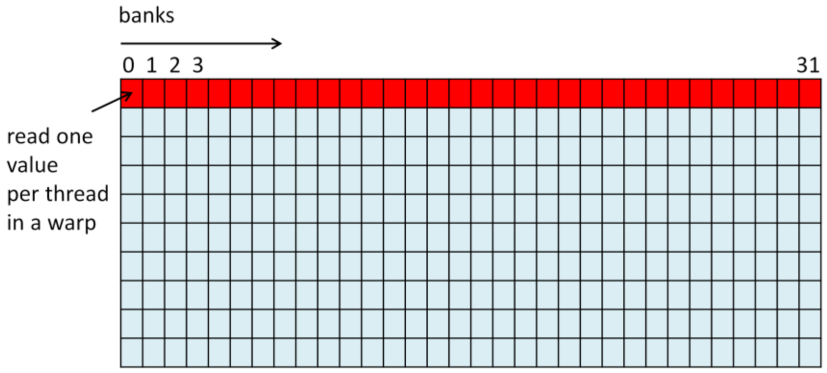
very fast

```
__global__ void Kernel()
{
        __shared__ float array[1000];
        …
        float a = 3.65 * array[3];
}
```

In case all threads in a warp access exactly the same element, this element is read from memory once and broadcasted to all threads in a very rapid manner.



Access patterns

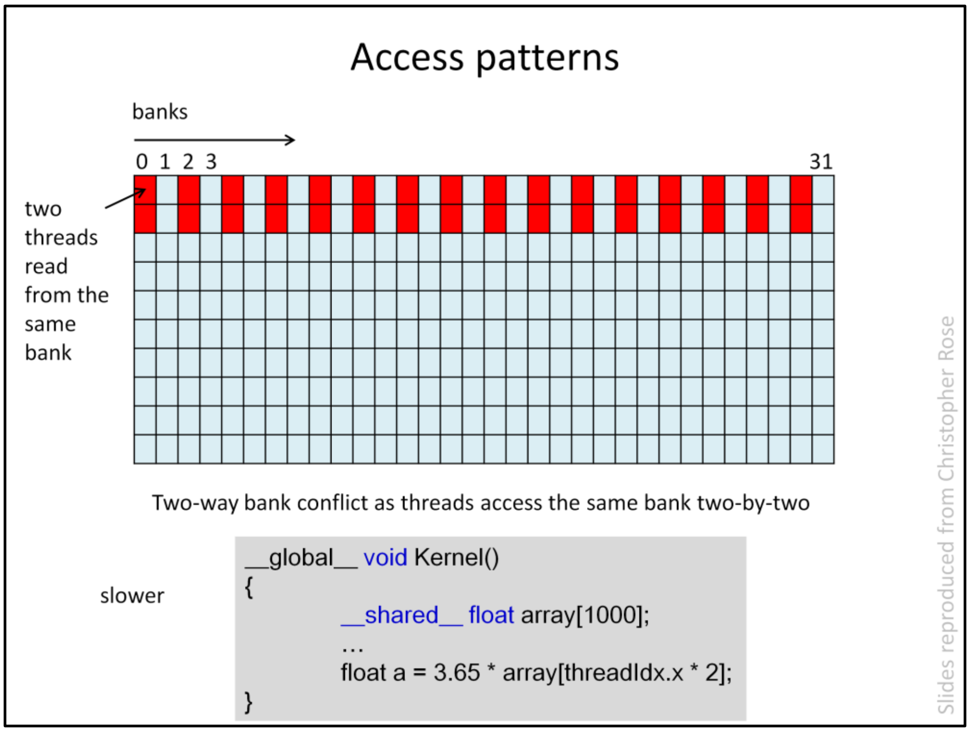


all threads in a warp read a word from a different bank

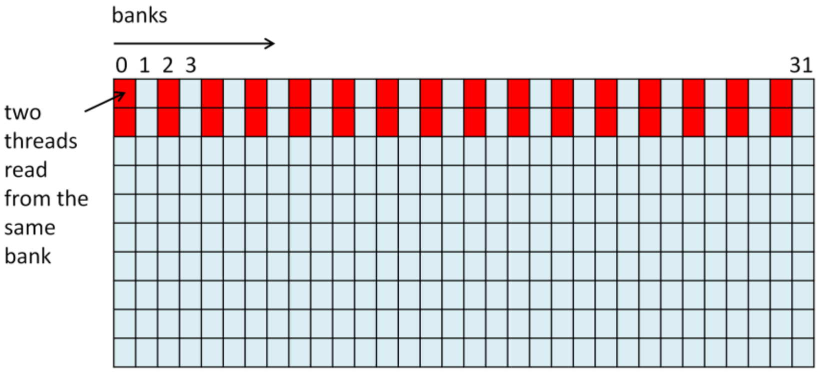
very fast

```
__global__ void Kernel()
{
            __shared__ float array[1000];
            …
            float a = 3.65 * array[threadIdx.x];
}
```

In case all threads in a warp access a different word, but all these words are stored in different banks, the access can be parallelized efficiently resulting in a very low access time.



Access patterns

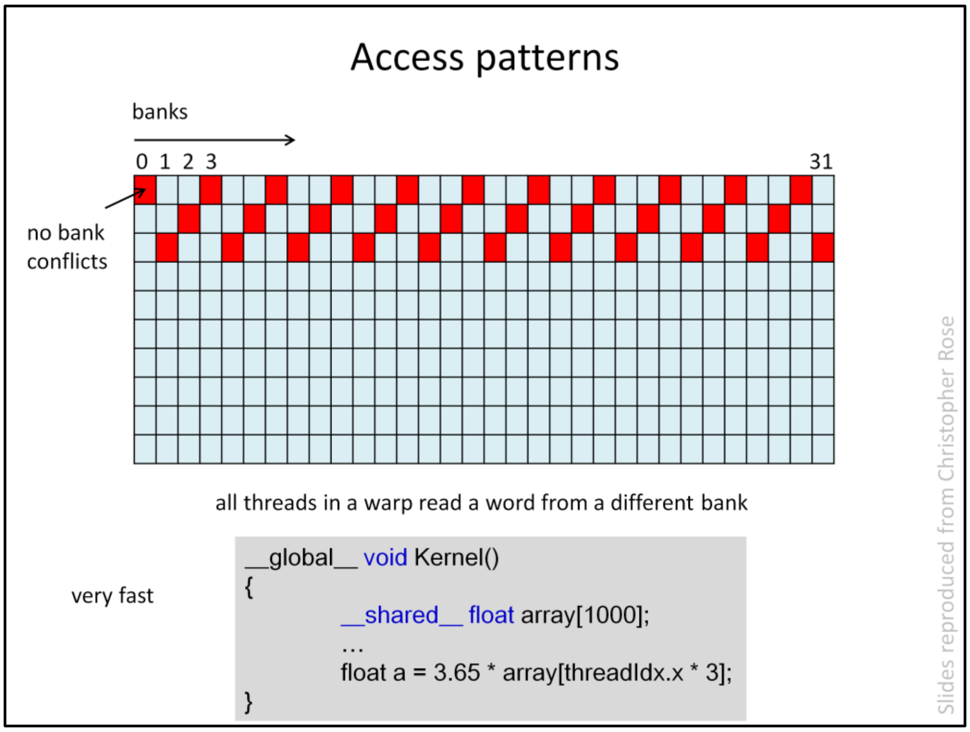


Two-way bank conflict as threads access the same bank two-by-two

slower

```
__global__ void Kernel()
{
        __shared__ float array[1000];
        …
        float a = 3.65 * array[threadIdx.x * 2];
}
```

A two-way bank conflict is also typically encountered when threads read consecutive elements from an array of doubles (two words). As different words belonging to the same bank cannot be retrieved simultaneously, this access pattern is somewhat slower.



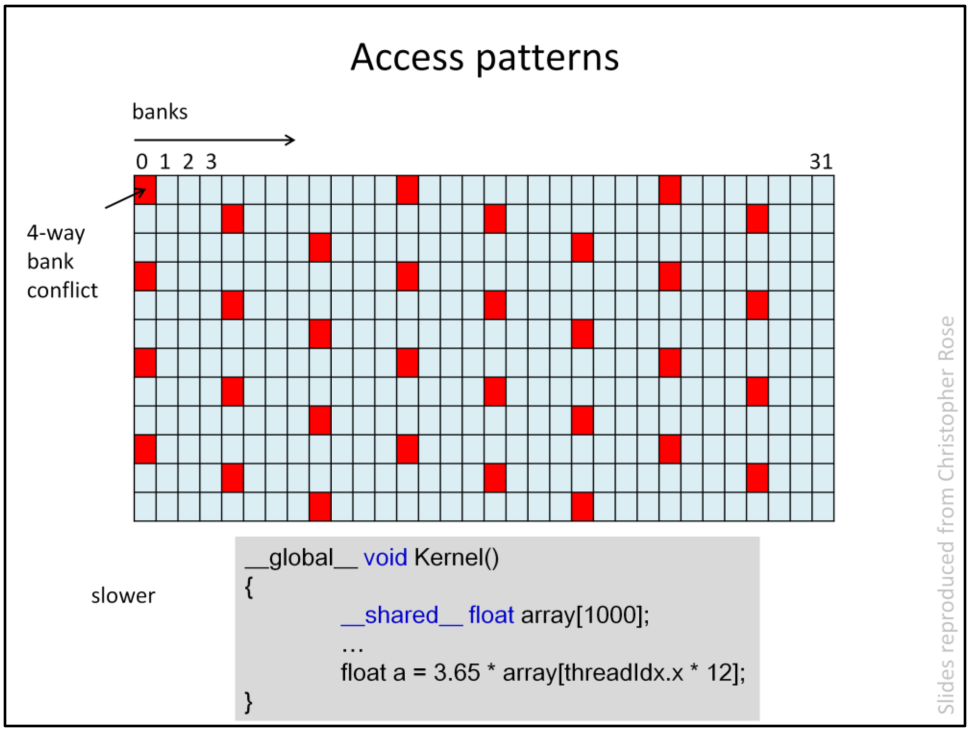
Access patterns

all threads in a warp read a word from a different bank

very fast

```
__global__ void Kernel()
{
        __shared__ float array[1000];
        …
        float a = 3.65 * array[threadIdx.x * 3];
}
```

A more complex access pattern, however, since all threads access different banks, the access pattern is very fast.



Access patterns

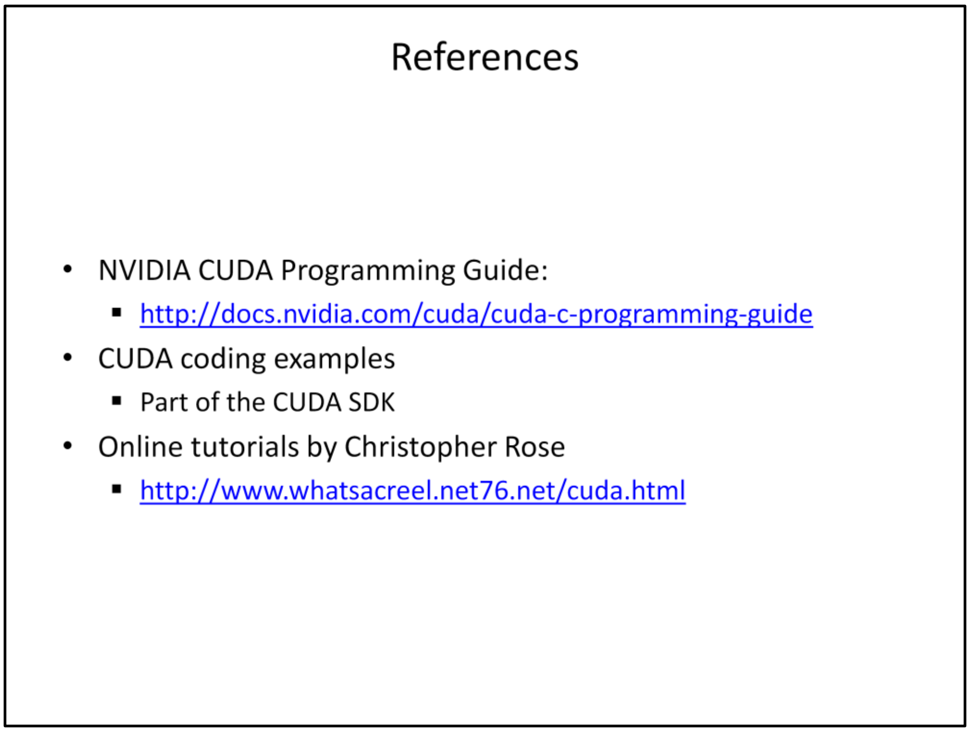
slower

```
__global__ void Kernel()
{
        __shared__ float array[1000];
        …
        float a = 3.65 * array[threadIdx.x * 12];
}
```

A four-way bank conflict, resulting in the serialization of four accesses. Note that the worst-case scenario is a 32-way bank conflict, if all threads in a warp try to access the same bank simultaneously.

Live examples

- Class-room demonstration
  - VecAdd
  - DeviceQuery
  - MatMul
  - Bank conflict example
  - MatMulBlocked (shared memory example)
  - MatMulCUBlas



References

- NVIDIA CUDA Programming Guide:
  - http://docs.nvidia.com/cuda/cuda-c-programming-guide
- CUDA coding examples
  - Part of the CUDA SDK
- Online tutorials by Christopher Rose
  - http://www.whatsacreel.net76.net/cuda.html

The end