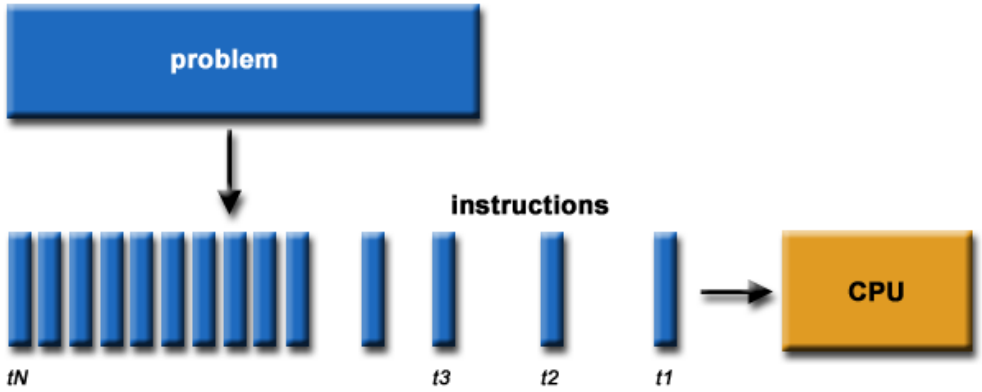
Parallel Programming using CUDA

Traditional Computing

Von Neumann architecture: instructions are sent from memory to the CPU

Serial execution: Instructions are executed one after another on a single Central Processing Unit (CPU)



Problems:

- More expensive to produce
- More expensive to run
- Bus speed limitation

Parallel Computing

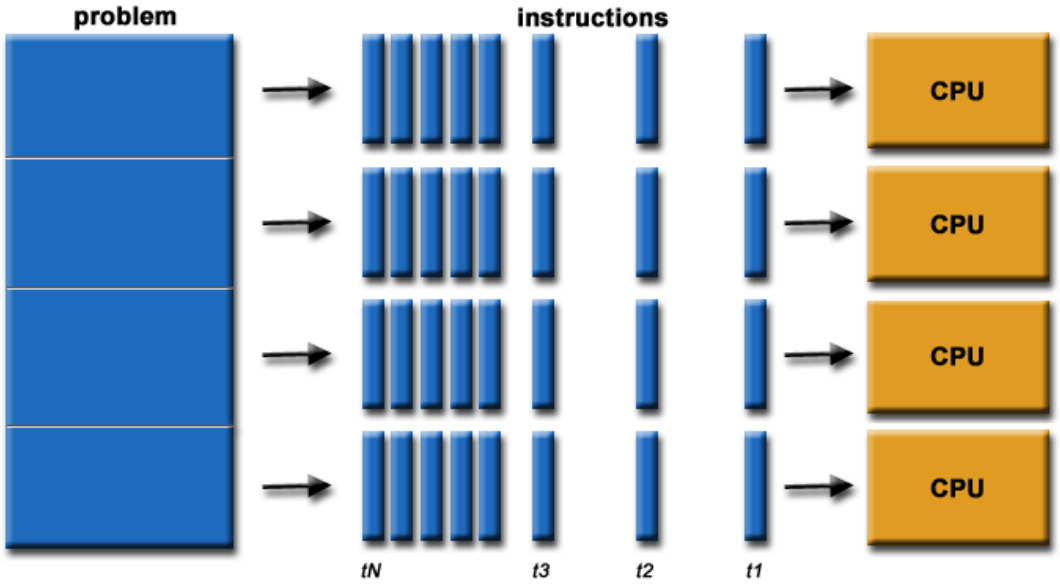
Official-sounding definition: The simultaneous use of multiple compute resources to solve a computational problem.

Benefits:

- Economical – requires less power and cheaper to produce
- Better performance – bus/bottleneck issue

Limitations:

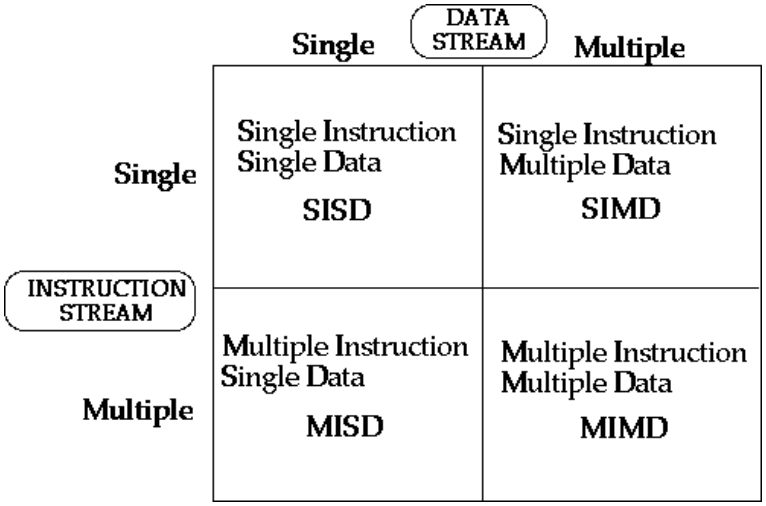
- New architecture – Von Neumann is all we know!
- New debugging difficulties – cache consistency issue



Flynn's Taxonomy

Classification of computer architectures, proposed by Michael J. Flynn

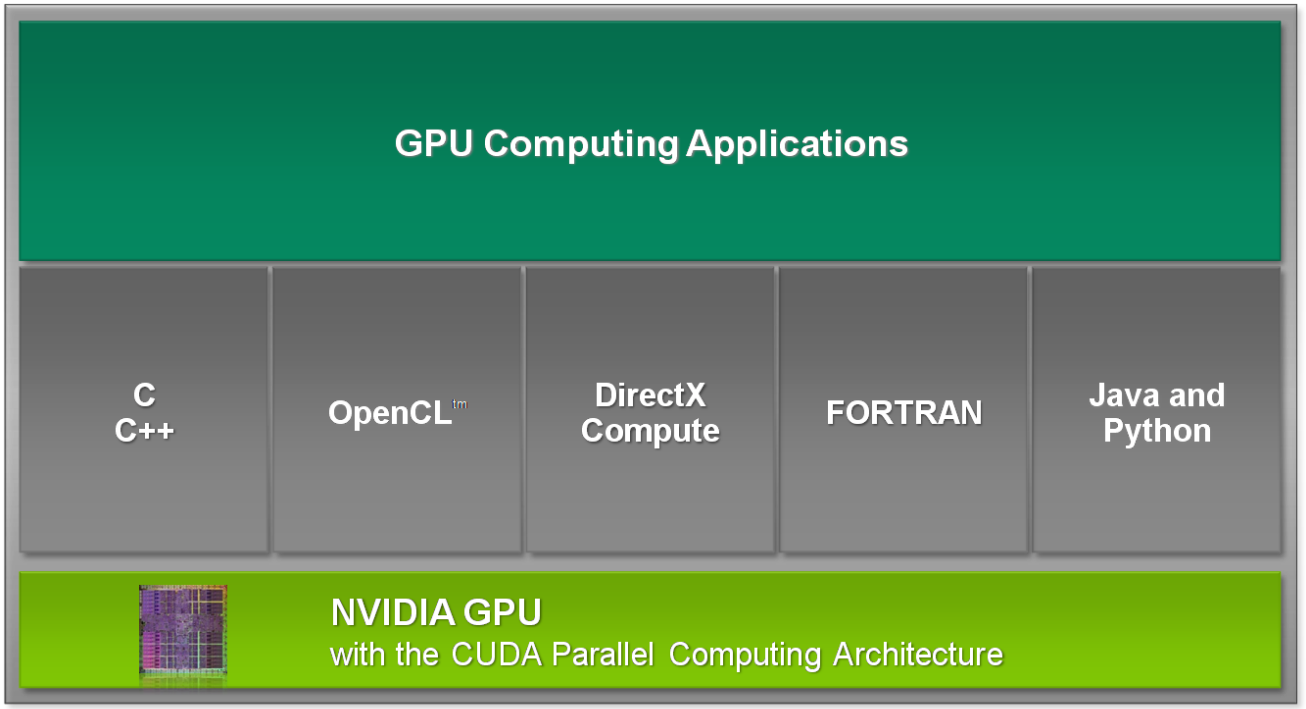
- SISD – traditional serial architecture in computers.
- SIMD – parallel computer. One instruction is executed many times with different data (think of a for loop indexing through an array)
- MISD - Each processing unit operates on the data independently via independent instruction streams. Not really used in parallel
- MIMD – Fully parallel and the most common form of parallel computing.



Enter CUDA

CUDA is NVIDIA's general purpose parallel computing architecture .

- designed for calculation-intensive computation on GPU hardware
- CUDA is not a language, it is an API
- we will mostly concentrate on the C implementation of CUDA

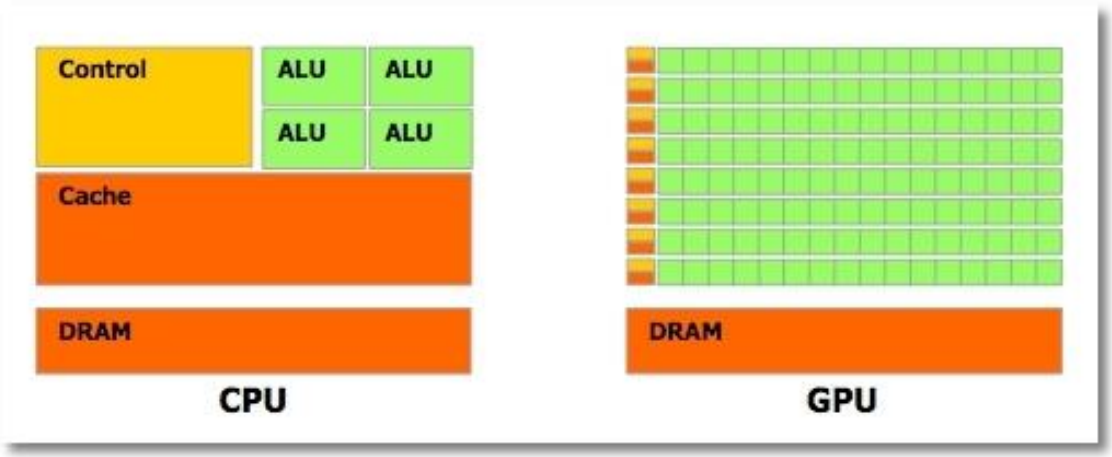


What is GPGPU ?

- General Purpose computation using GPU in applications other than 3D graphics
  - GPU accelerates critical path of application
- Data parallel algorithms leverage GPU attributes
  - Large data arrays, streaming throughput
  - Fine-grain SIMD parallelism
  - Low-latency floating point (FP) computation
- Applications – see //GPGPU.org
  - Game effects (FX) physics, image processing
  - Physical modeling, computational engineering, matrix algebra, convolution, correlation, sorting

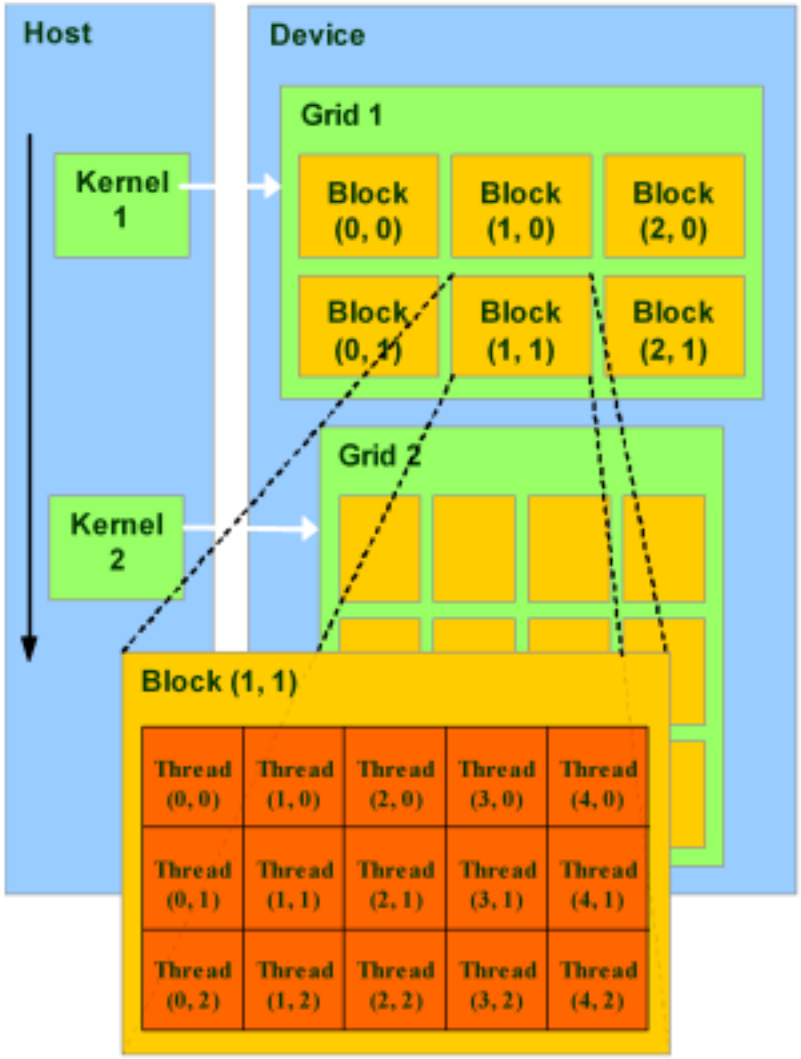
CUDA Goals

- Scale code to hundreds of cores running thousands of threads
- The task runs on the gpu independently from the cpu



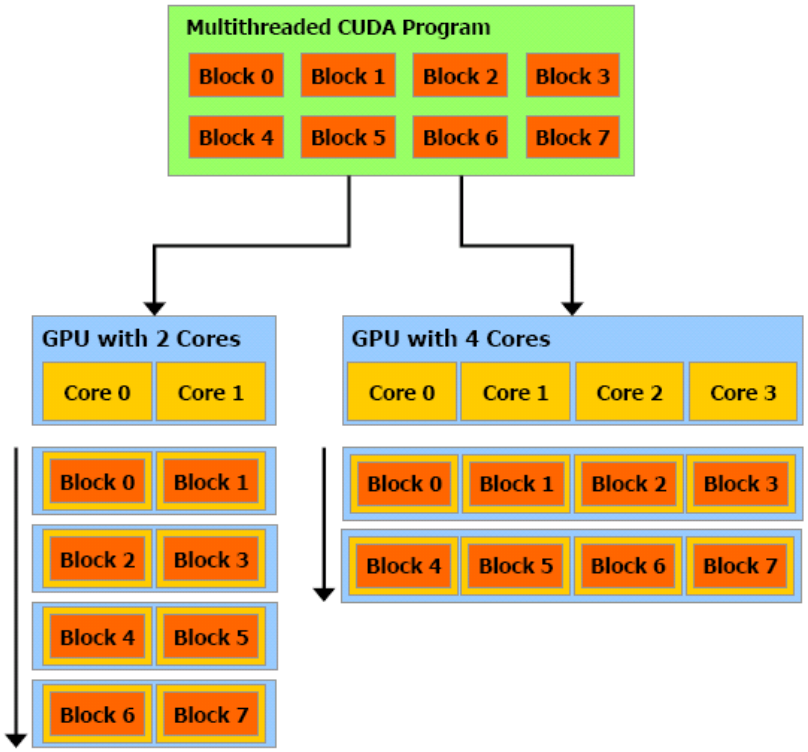
CUDA Structure

- Threads are grouped into thread blocks
- Blocks are grouped into a single grid
- The grid is executed on the GPU as a kernel



Scalability

- Blocks map to cores on the GPU
- Allows for portability when changing hardware



Terms and Concepts

Each block and thread has a unique id within a block.

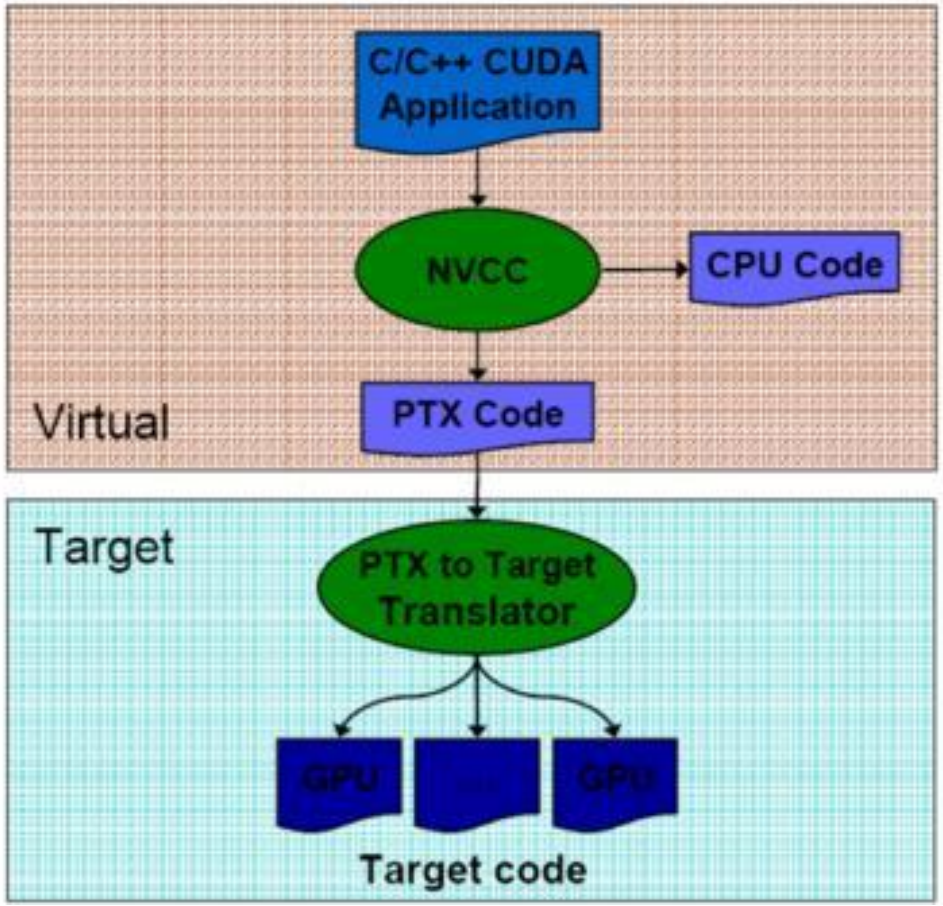
- threadIdx – identifier for a thread
- blockIdx – identifier for a block
- blockDim – size of the block

Unique thread id:

(blockIdx*blockDim)+threadIdx

NVCC compiler

- Compiles C or PTX code (CUDA instruction set architecture)
- Compiles to either PTX code or binary (cubin object)



Development: Basic Idea

1. Allocate equal size of memory for both host and device
2. Transfer data from host to device
3. Execute kernel to compute on data
4. Transfer data back to host

Kernel Function Qualifiers

- __device__
- __global__
- __host__

Example in C:

CPU program

void increment_cpu(float *a, float b, int N)

CUDA program

__global__ void increment_gpu(float *a, float b, int N)

Variable Type Qualifiers

- Specify how a variable is stored in memory
- __device__
- __shared__
- __constant__

Example:

```
__global__ void increment_gpu(float *a, float b, int N)
{
    __shared__ float shared[];
}
```

Calling the Kernel

- Calling a kernel function is much different from calling a regular function

```
Void main(){
Int blocks = 256;
Int threadsperblock = 512;
    mycudafunc<<<blocks,threadsperblock>>>(some parameter);
}
```

GPU Memory Allocation / Release

Host (CPU) manages GPU memory:

- cudaMalloc (void ** pointer, size_t nbytes)
- cudaMemset (void * pointer, int value, size_t count);
- cudaFree (void* pointer)

```
Void main(){
    int n = 1024;
    int nbytes = 1024*sizeof(int);
    int * d_a = 0;
    cudaMalloc( (void**)&d_a,  nbytes );
    cudaMemset( d_a, 0, nbytes);
    cudaFree(d_a);
}
```

Memory Transfer

cudaMemcpy(void *dst, void *src, size_t nbytes, enum cudaMemcpyKind direction);

- returns after the copy is complete blocks CPU
- thread doesn't start copying until previous CUDA calls complete

enum cudaMemcpyKind

- cudaMemcpyHostToDevice
- cudaMemcpyDeviceToHost
- cudaMemcpyDeviceToDevice

Host Synchronization

All kernel launches are asynchronous

- control returns to CPU immediately
- kernel starts executing once all previous CUDA calls have completed

Memcopies are synchronous

- control returns to CPU once the copy is complete
- copy starts once all previous CUDA calls have completed

cudaThreadSynchronize()

- blocks until all previous CUDA calls complete

Asynchronous CUDA calls provide:

- non-blocking memcopies
- ability to overlap memcopies and kernel execution

The Big Difference

CPU program

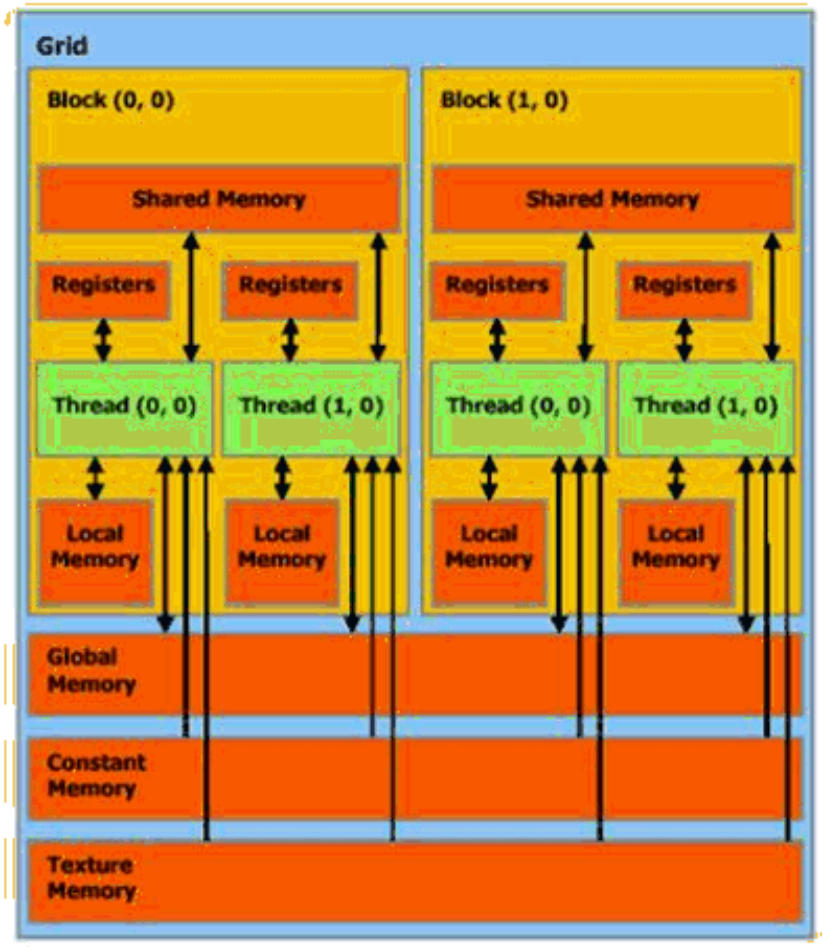
```
void increment_cpu(float *a, float b, int N)
{
 for (int idx = 0; idx<N; idx++)
a[idx] = a[idx] + b;
}

void main()
{
…
increment_cpu(a, b, N);
}
```

GPU program

```
__global__ void increment_gpu(float *a, float b, int
     N)
 {
int idx = blockIdx.x * blockDim.x + threadIdx.x;
     if( idx < N)              a[idx] = a[idx] + b;
}
void main() {
…..
dim3 dimBlock (blocksize);
 dim3 dimGrid( ceil( N / (float)blocksize)  )
     increment_gpu<<<dimGrid, dimBlock>>>(a, b,
     N);
}
```

Memory Hierarchy

- Shared memory much much faster than global
- Don’t trust local memory
- Global, Constant, and Texture memory available to both host and cpu



Global and Shared Memory

Global memory not cached on G8x GPUs

- **High latency, but launching more threads hides latency**
- **Important to minimize accesses**
- **Coalesce global memory accesses (more later)**

Shared memory is on-chip, very high bandwidth

- **Low latency (100-150times faster than global memory)**
- **Like a user-managed per-multiprocessor cache**
- **Try to minimize or avoid bank conflicts (more later)**

Texture and Constant Memory

Texture partition is cached

- **Uses the texture cache also used for graphics**
- **Optimized for 2D spatial locality**
- **Best performance when threads of a warp read locations that are close together in 2D**

Constant memory is cached

- **4 cycles per address read within a single warp**
- **Total cost 4 cycles if all threads in a warp read same address**
- **Total cost 64 cycles if all threads read different addresses**

Coalescing

- A coordinated read by a half-warp (16threads)
- A contiguous region of global memory:
  - 64bytes - each thread reads a word: int, float, …
  - 128bytes - each thread reads a double-word: int2, float2, …
  - 256bytes - each thread reads a quad-word: int4, float4, …

Additional restrictions:

- Starting address for a region must be a multiple of region size
- The kth thread in a half-warp must access the kth element in a block being read

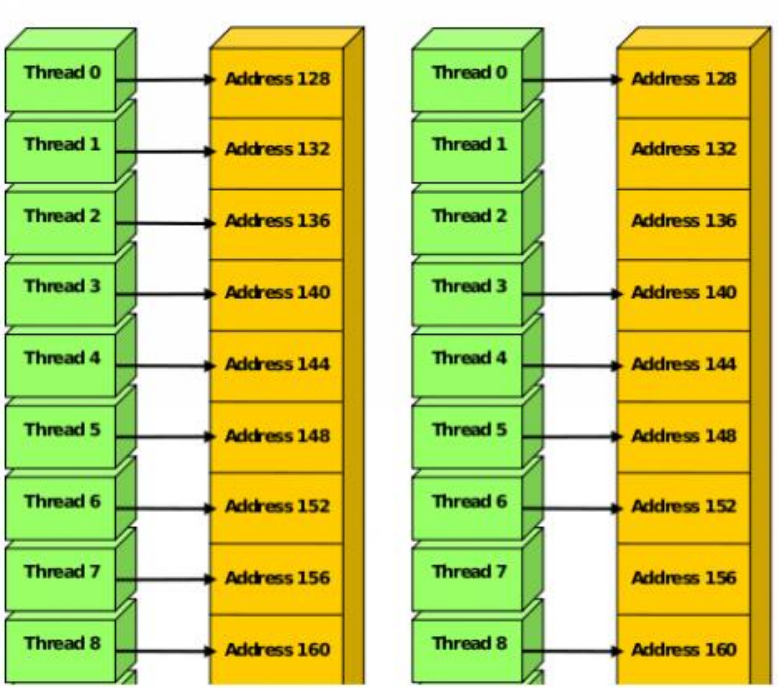
Exception: not all threads must be participating

- Predicated access, divergence within a halfwarp

See Nvidia Cuda volume2.pdf tutorial

Coalescing (cont.)

Coalesced memory accesses



Uncoalesced memory accesses

Mechanics of Using Shared Memory

- __shared__ type qualifier required
- Must be allocated from global/device function, or as "extern"
- Examples:

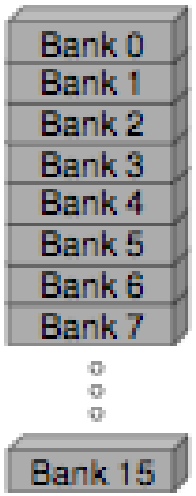
```
extern __shared__ float  d_s_array[];

/* a form of dynamic allocation */
/* MEMSIZE is size of per-block  */
/* shared memory*/
__host__ void outerCompute() {
   compute<<<gs,bs,MEMSIZE>>>();
}
__global__ void compute() {
    d_s_array[i] = …;
}
```

```
__global__ void compute2() {
   __shared__ float d_s_array[M];

   /* create or copy from global memory */
   d_s_array[j] = …;

  /* write result back to global memory */
   d_g_array[j] =  d_s_array[j];
}
```

Optimization using Shared Memory

Parallel Memory Architecture

- Many threads accessing memory
  - Therefore, memory is divided into banks
  - Essential to achieve high bandwidth
- Each bank can service one address per cycle
  - A memory can service as many simultaneous accesses as it has banks
- Multiple simultaneous accesses to a bank result in a bank conflict
  - Conflicting accesses are serialized



Bank Conflicts

- Shared memory is divided into banks
- Each bank has serial read/write access
- Bank addresses are striped
- If more than one thread attempts to access the same bank at the same time, they're accesses are serialized
- This is a bank conflict