GPGPU Programming Fundamentals — 2

Salvatore Filippone, PhD

DICII
salvatore.filippone@uniroma2.it

Synchronisation

In this section you will hear about...

- How the kernels are dispatched and processed on the GPU
- Synchronisation between the host and the device
- Timing the GPU code execution
- Synchronisation between CUDA threads

Kernel Dispatch

Kernel invocation is a form of *Remote Procedure Call* (RPC). Hence:

- CUDA kernel calls are asynchronous
- CPU can run its own computation while waiting for the GPU
- `cudaDeviceSynchronize()` call blocks the CPU until the GPU has finished scheduled work

Kernel Dispatch

Kernel invocation is a form of *Remote Procedure Call* (RPC). Hence:

- CUDA kernel calls are asynchronous
- CPU can run its own computation while waiting for the GPU
- `cudaDeviceSynchronize()` call blocks the CPU until the GPU has finished scheduled work

```
Kernel<<< ... >>>(...);
// Kernel is scheduled for execution on the GPU
// Control is returned immediately to the CPU
do_something(); // CPU can run its own computation
// Computation on the GPU may have not even started yet!
do_something_more();
```

Multiple Kernels Execution

```
Kernel1<<< ... >>>(...);
Kernel2<<< ... >>>(...);
Kernel3<<< ... >>>(...);
// Three kernel calls are scheduled on the GPU
// They will be executed one at a time in FIFO order
// Control is returned immediately to the CPU
```

Implicit Kernel Serialisation

- All three kernel calls will be executed, but not (necessarily) concurrently
- Kernels sent to the same CUDA stream (default stream in this case) are executed in FIFO order
- It is possible to execute kernels concurrently using multiple CUDA streams

Timing & benchmarks

How do you time a GPU application?

Timing & benchmarks

How do you time a GPU application?

Simple approach

```
sdkStartTimer(&timer);
Kernel<<< ... >>>(...);
sdkStopTimer(&timer);
cout << sdkGetTimerValue(&timer);
```

What have you actually measured?

Example: Timing GPU Execution (1/2): Simple Solution

Easy solution for the GPU

```
sdkStartTimer(&timer);
Kernel<<< ... >>>(...);
cudaDeviceSynchronize(); // Blocks the CPU
// until the GPU has finished computation
// Introduces overhead
sdkStopTimer(&timer);
cout << sdkGetTimerValue(&timer); // Almost correct
```

Example: Timing GPU Execution (2/2): Advanced Solution

Accurate solution using CUDA events

```
cudaEvent_t start, stop;
cudaEventCreate(&start);
cudaEventCreate(&stop);
cudaEventRecord(start, 0); // 0 - the default stream
Kernel<<< ... >>>(...);
cudaEventRecord(stop, 0); // 0 - the default stream
cudaEventSynchronize(stop);
float time;
cudaEventElapsedTime(&time, start, stop);
cout << time; // Very accurate
cudaEventDestroy(start);
cudaEventDestroy(stop);
```

Thread Synchronisation

- Threads can co-operate by exchanging data via shared memory
- To avoid race conditions synchronisation is essential
- `__syncthreads()` is a low-overhead barrier synchronisation
- All threads in a block must reach barrier before they continue
- There is no (direct) way of applying synchronisation among the blocks!

$\Rightarrow$ Blocks should be designed to execute independently

Thread Synchronisation

Imagine two threads executing this code:

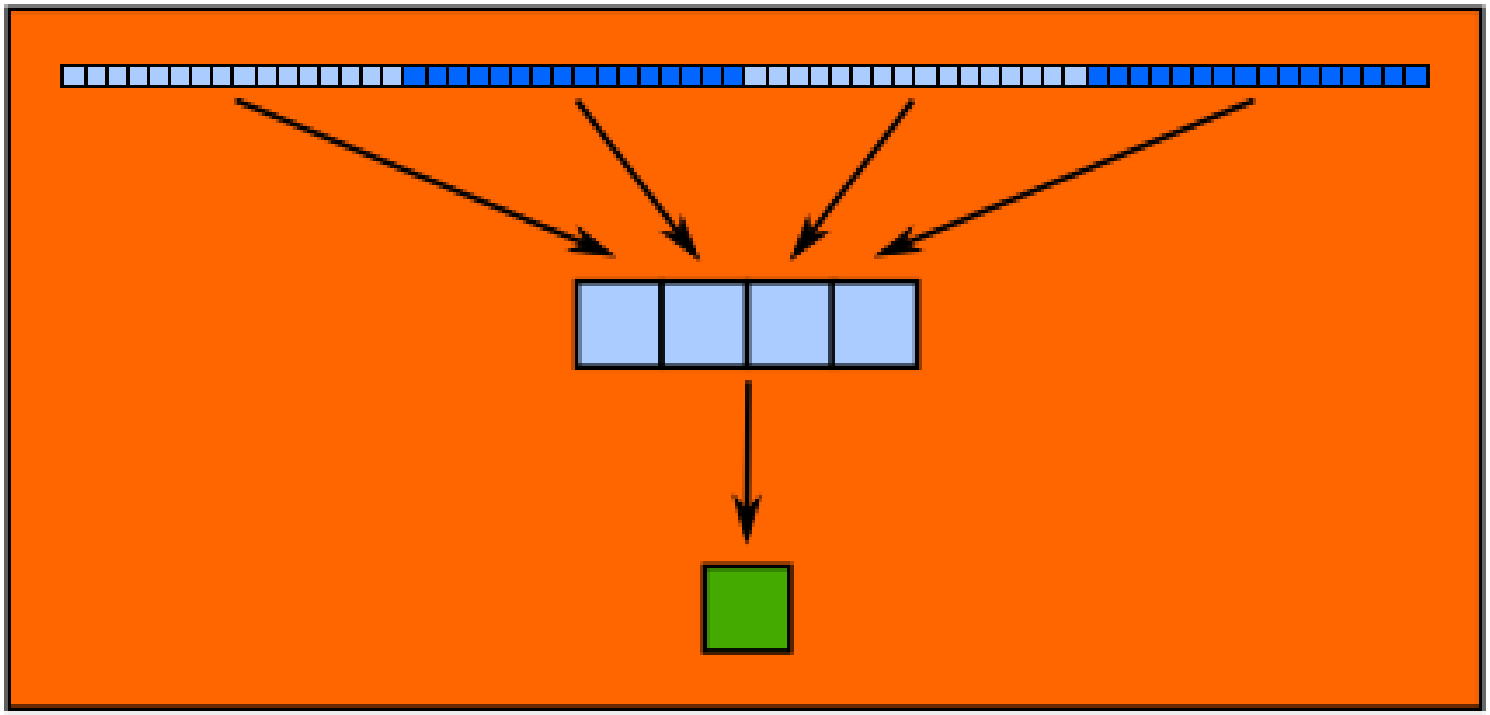
```
count = 4; // Variable count is shared by two threads
count = count + 1;
```

What is the value of count?

Race Conditions

- Difficult to detect (even with the unit tests)
- Even more difficult to identify and fix

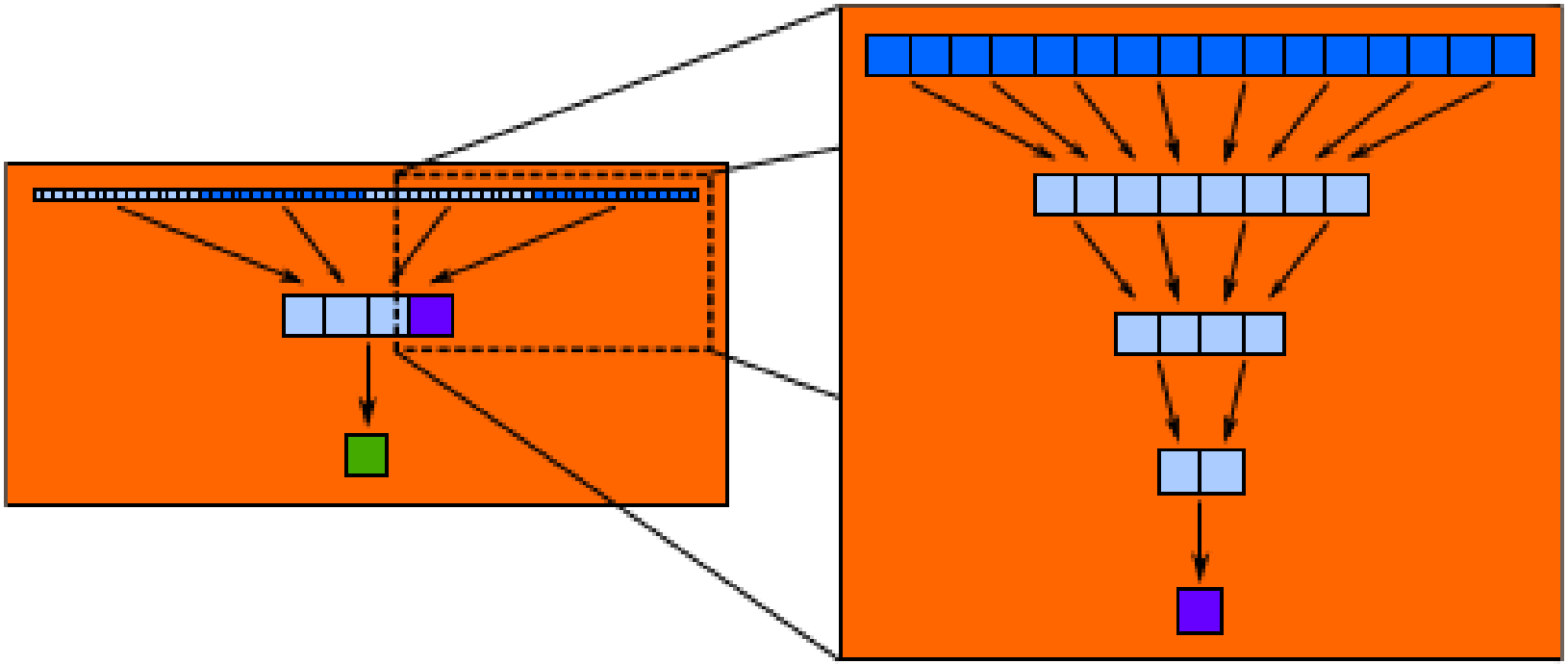
Example: Parallel Reduction



Procedure

- One thread per value, each block performs a local reduction
- Procedure is repeated on the partially reduced data until only one value is left (multiple kernel calls)

Example: Parallel Reduction



Synchronisation

- In intra-block reduction `__syncthreads()` is used
- All blocks must finish before the next level is processed (implicit kernel serialisation)

Summary

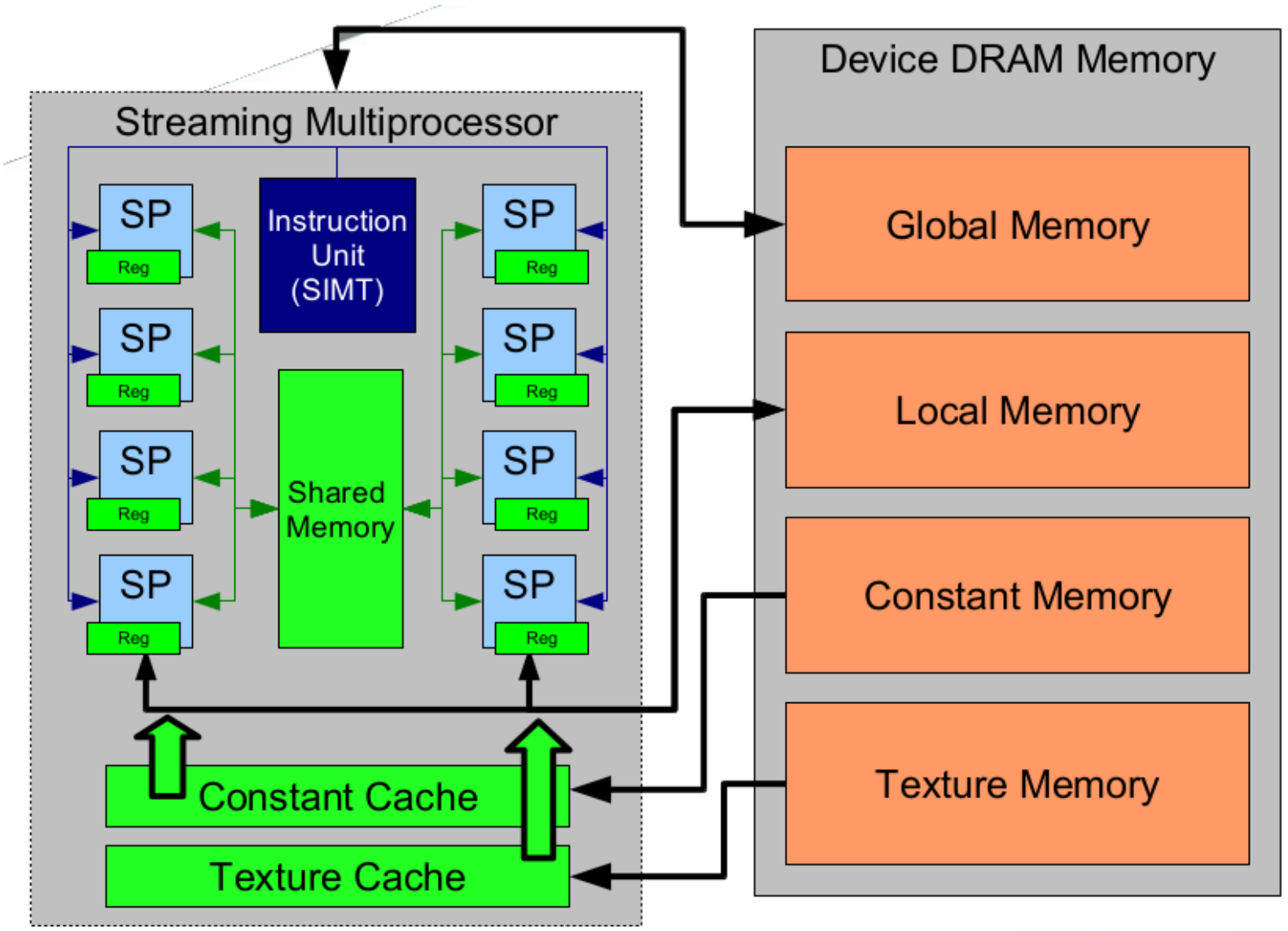
- Kernel calls are asynchronous with respect to the host
- The CPU can wait for the GPU to complete computation by calling `cudaDeviceSynchronize()`
- Multiple kernel calls dispatched to the same CUDA stream are serialised and executed on the GPU in FIFO order
- `__syncthreads()` instruction provides a low-overhead barrier synchronisation for threads within a block
- There is no (direct) way of applying synchronisation among the blocks!
- Blocks should be designed to execute independently

Memory Model

In this section you will hear about...

- Memory types available on the GPU and their characteristics
- Communication between the threads using shared memory
- Communication between the host and the device

GPU Memory


Streaming Multiprocessor
SP
Reg
Instruction Unit (SIMT)
Shared Memory
Constant Cache
Texture Cache
Device DRAM Memory
Global Memory
Local Memory
Constant Memory
Texture Memory

Basic Memory Types (1/4)

Global memory

- ✓ Used for data exchange between host and device
- ✓ The largest memory space — typically several GBs
- ✓ Accessible directly from the GPU by all threads and blocks
- ✓ Supports reading and writing
- ✗ Relatively slow
- ✗ High latency (400–600 GPU cycles per operation)

From CC 2.x, global memory accesses are handled through a Level-2 cache.

Device Memory: Global Memory

To create a variable in global shared memory, use the qualifier

`__device__`

- Has the lifetime of the application
- Is accessible to
  - all threads in the grid
  - the host, using runtime library functions

Use CUDA API functions to allocate memory on the device

```
cudaMalloc((void**) &ptrMem, size)
```

- `PtrMem` is a pointer to device memory
- `size` is the amount of memory to be allocated
- Memory freed using `cudaFree(ptrMem)` function

Device Memory

Once device memory has been allocated, data can be transferred between host and device

`cudaMemcpy(dest, src, size, <dirn>)`

- `dest` pointer to destination location
- `src` pointer to source location
- `size` size of data to be transferred
- `<dirn>` direction of transfer
  - `cudaMemcpyHostToDevice`
  - `cudaMemcpyDeviceToHost`

So Where Is My Data?

- The CPU does not access the GPU (global) memory directly
- CUDA kernels do not have access to the main memory

CUDA offers an API for:

- GPU memory allocation and deallocation
- CPU-GPU transfers

Recent versions of CUDA provide UVA: Unified Virtual Address space, but need to understand the basics first.

GPU Memory Allocation

```
float* dev_float = 0;
cudaMalloc((void**) &dev_float, N * sizeof(float));
// Allocated an array with N float elements
// Template version of cudaMalloc is available
int* dev_int = 0;
cudaMalloc<int>(&dev_int, N * sizeof(int));
// Size is always specified in bytes, not elements!
...
cudaFree(dev_float);
cudaFree(dev_int); // Never forget to free memory
```

Do not mix the pointers!

- Pointers to the CPU and the GPU memory are of the same type and cannot be easily distinguished
- Using the wrong pointer will result in segmentation fault
- Documentation should specify what kind of pointer is expected

CPU-GPU Memory Transfer

```
float* host = new float[N];
float* device = 0;
cudaMalloc((void**) &device, N * sizeof(float));
// Copying data from main memory to device memory
cudaMemcpy(device, host, N * sizeof(float),
                    cudaMemcpyHostToDevice);
// Copying data from device memory to host memory
cudaMemcpy(host, device, N * sizeof(float),
                    cudaMemcpyDeviceToHost);
delete[] host;
cudaFree(device);
```

- `cudaMemcpy(destination, source, size, direction)`
- Again: Do not mix the pointers!

A Note on CUDA API Calls

There are many types of CUDA API calls: device, memory, stream and event management (and more)

Check for CUDA API errors

- All CUDA API calls return `cudaError_t` value
- It is a good practice to check if cudaSuccess was returned
- CUDA SDK provides macro `checkCudaErrors()`

Synchronisation

- Unlike kernel calls, many API calls are synchronous with respect to the host
- This includes `cudaMemcpy()`
- However, an API call may return an error from previous asynchronous call!

Error checking macro

```
#include <helper_cuda.h>

checkCudaErrors(cudaMalloc((void **) &gpu_I,
                           N*sizeof(int)));
checkCudaErrors(cudaMalloc((void **) &gpu_V,
                           K*sizeof(float)));
```

Always use it! Silent CUDA errors would otherwise show up as unreasonably short execution times!

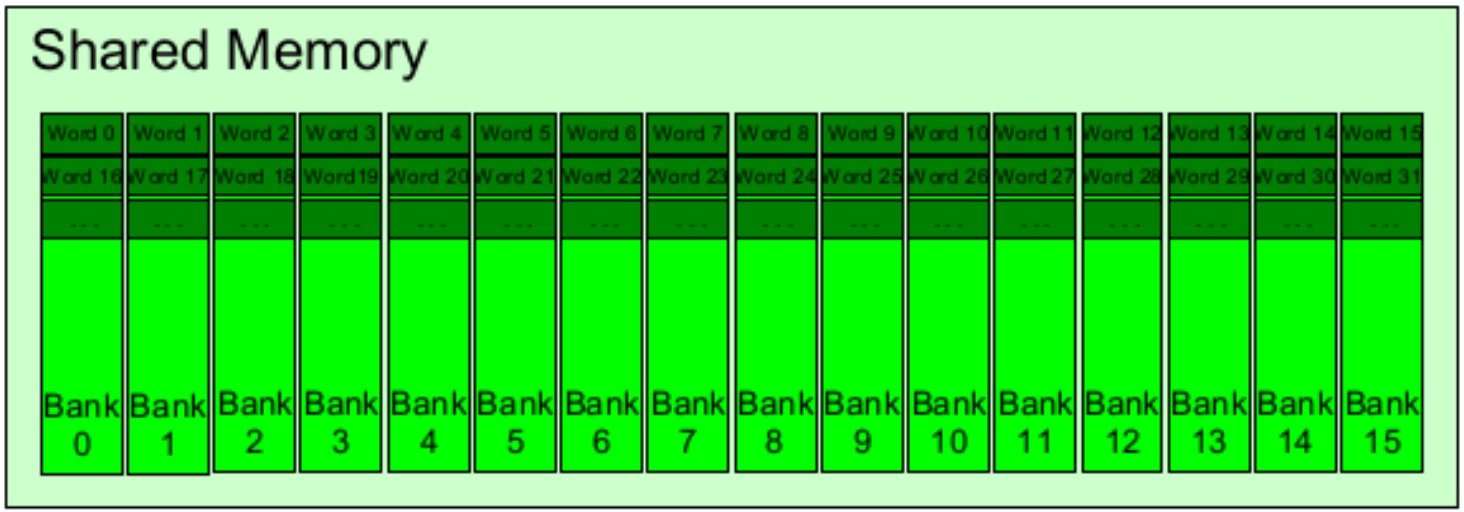
Basic Memory Types (2/4): Shared Memory

Shared memory

- ✓ Provides the way of communication between the threads within a block
- ✓ Relatively fast
- ✓ Low latency (a few GPU cycles per operation)
- ✓ Supports reading and writing
- ✗ Very small (16–248 KB per SM)
- ✗ Has a lifetime of a block
- ✗ Only accessible by threads within a block

On-chip Memory: Local Shared

- Available to all Scalar Processors on the Streaming Multiprocessor
- Available to all active threads in a block
- This is the key memory for performance; care needed to avoid unforeseen data dependencies
- Memory arranged as banks to increase parallel access
- Accessing a memory location with multiple threads may cause a bank conflict $\Rightarrow$ serialization and performance loss



Local Shared

To declare a variable in local shared memory, use the qualifier

`__shared__`

- Has the lifetime of the block
- Can only be accessed by the threads in the block
- When a thread updates a variable in shared memory, other threads may not immediately see the change
- Requires a `__syncthreads()` function call

Basic Memory Types (3/4): Registers

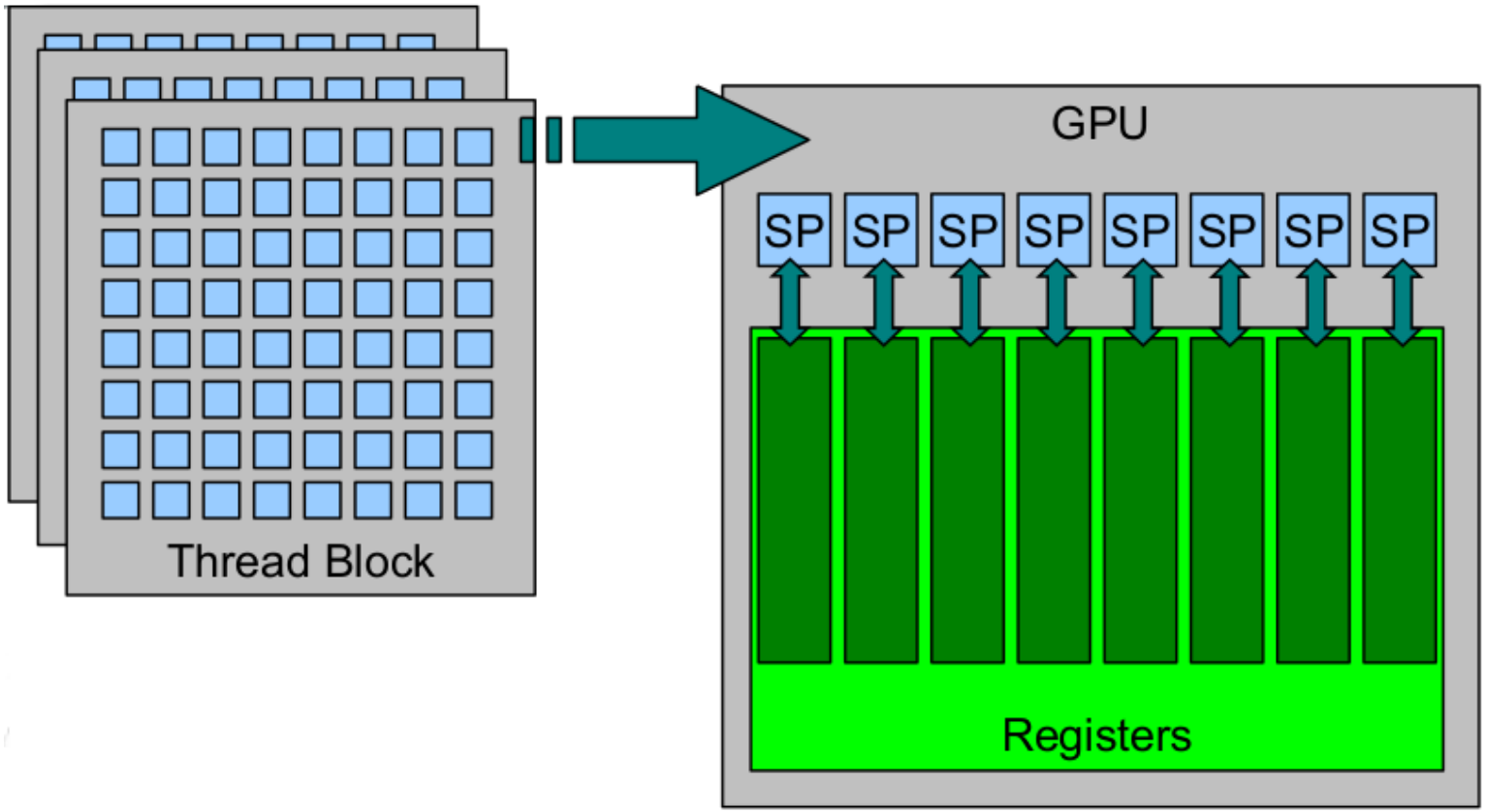
Registers

- ✓ The fastest memory used for thread automatic variables
- ✓ Supports reading and writing
- ✗ Very small (typically several 4-byte registers per thread)
- ✗ Only accessible by a thread

Use sparingly!

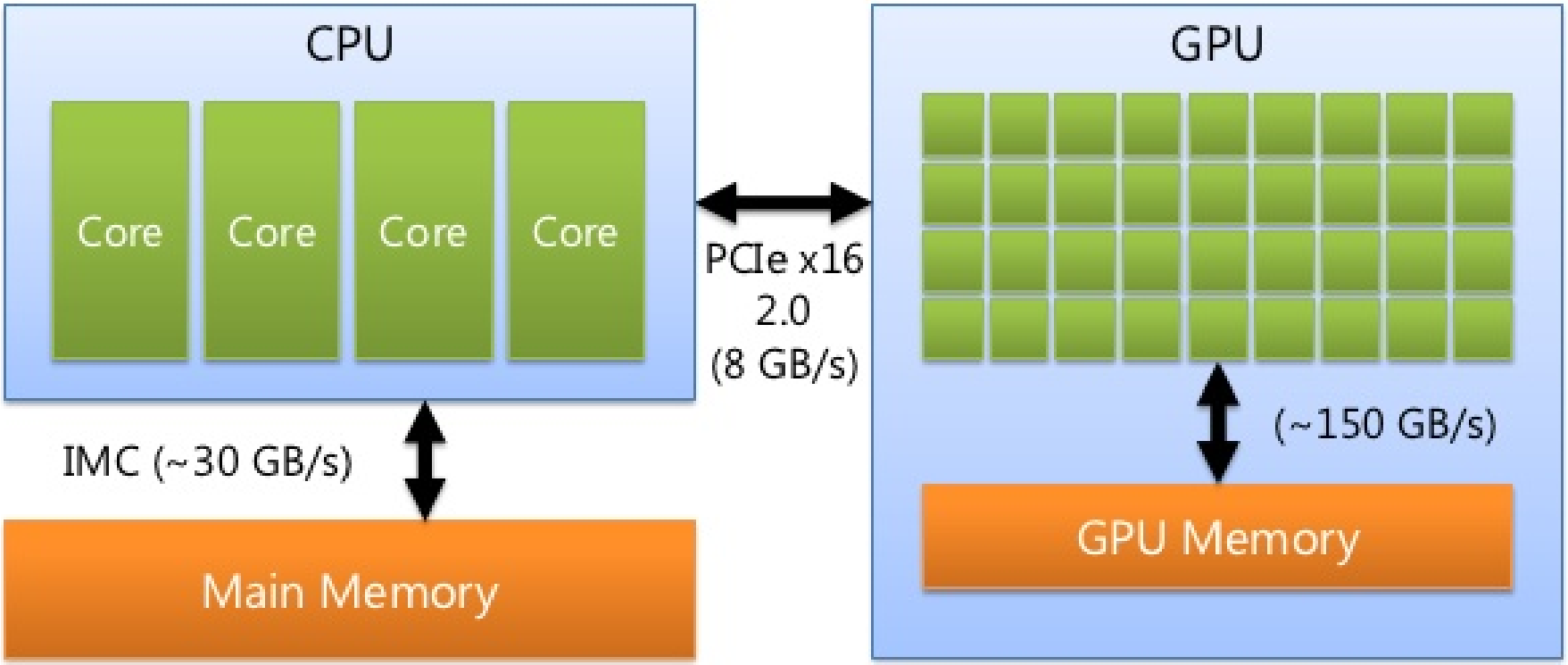
- If you declare too many automatic variables (arrays, structures) the compiler may decide to put them in global memory (register spill)!
- This may lead to very poor performance!

Scalar Processors: Registers



- Registers distributed between the active threads
- Limits the maximum number of active threads

Basic Memory Types (4/4)



Shared memory and registers reside on streaming multiprocessors

Advanced Memory Types (1/2)

Constant memory

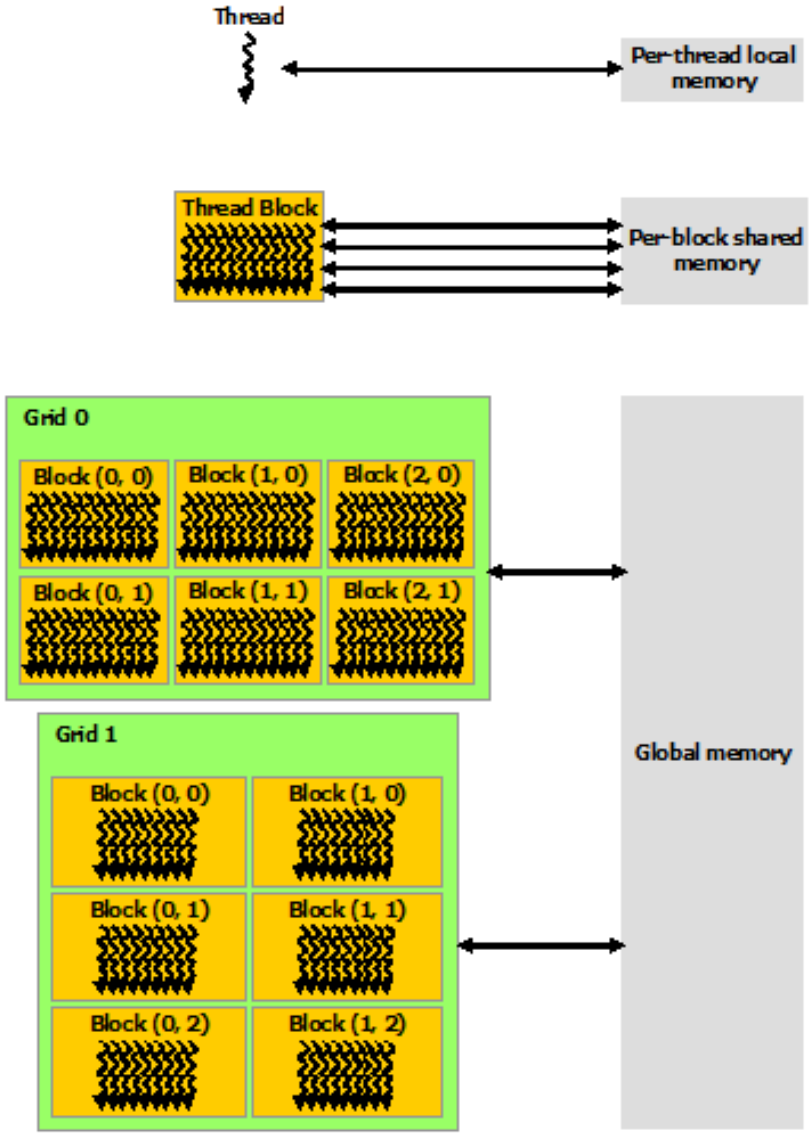
- ✓ Cached memory accessible by all threads
- ✓ Fast alternative to global memory for small constant data
- ✓ Implicitly used for transferring kernel calls parameters
- ✗ Read-only
- ✗ Very small (64 KB)

Advanced Memory Types (2/2)

Texture Memory

- ✓ Provides cached access to global memory
- ✓ Optimised for 2D spatial locality (e.g. five-point stencil)
- ✓ Offers different addressing modes and data filtering
- ✗ Read-only
- ✗ Has to be accessed indirectly through an API
- ✗ Difficult to understand performance

GPU Memory Hierarchy



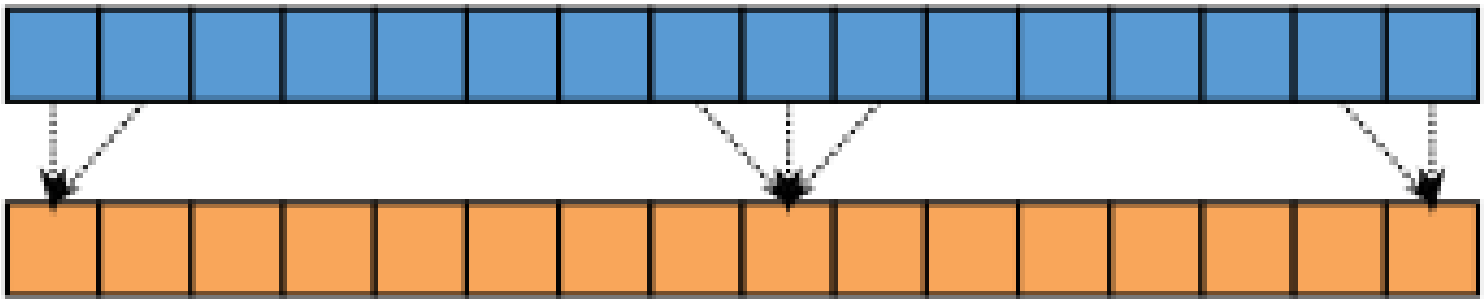
GPU Memory Comparison

Memory	Speed	Size	Access	Visibility	Persistent
Global	slow	very large	RW	global	yes
Shared	fast	small	RW	block	no
Registers	very fast	very small	RW	thread	no
Constant	medium	small	RO	global	yes
Texture	fast	large	RO	global	yes

Shared Memory Example

```
__global__ void Stencil(int n, int iters, float* x) {
  int tid = threadIdx.x;
  int gid = tid + blockIdx.x * blockDim.x;
  if (gid >= n) return;
  extern __shared__ float aux[];
  aux[tid] = x[gid]; // Load input data to shared memory
  __syncthreads(); // Wait for all threads to load data
  float left, right; // These reside in registers
  for (int iter = 0; iter < iters; ++iter) {
    left = (tid > 0 ? aux[tid-1] : 0.0f);
    right = (tid + 1 < blockDim.x ? aux[tid+1] : 0.0f);
    __syncthreads();
    aux[tid] = left + aux[tid] + right;
    __syncthreads();
  }
  x[gid] = aux[tid]; // Write results to global memory
}
```

Shared memory Example



Data dependencies in three-point stencil

What did just happen?!

- `extern __shared__ float aux[]` declares shared memory with dynamic size (determined at run-time)
- If the amount of shared memory is known at the compile time, it can be declared like `__shared__ float aux[10]`
- Different indexing is used by global $(0, \ldots, n-1)$ and shared memory $(0, \ldots, blockDim.x - 1)$
- `__syncthreads()` synchronises all threads within a block
- Synchronisation is essential to avoid race condition

Dynamic Shared Memory (1/2)

```
int main(int argc, char** argv) {
    ... // Allocate and initialise array d_x
    size_t shmem_size = 256 * sizeof(float); // In bytes!
    Stencil<<<N / 256, 256, shmem_size>>>(N, 10, d_x);
}
```

How does the GPU know how much memory to allocate?

- Shared memory size is declared using <<<...>>> syntax
- Only one dynamic shared memory block can be allocated

Dynamic Shared Memory (2/2): Multiple arrays in shared memory

What if I need more than one shared array?

```
__device__ void Kernel(...) {
    // Each thread gets one element in float and int
    // vectors and two elements in short vector
    extern __shared__ float float_aux[];
    short* short_aux = (short*) (float_aux + blockDim.x);
    int* int_aux = (int*) (short_aux + 2 * blockDim.x);
    // All three variables point to shared memory
}
int main(int argc, char** argv) {
    size_t shmem_size = 256 * // In bytes!
            (sizeof(float) + 2 * sizeof(short)
             + sizeof(int));
    Kernel<<<N / 256, 256, shmem_size>>>(...);
}
```

Summary

- There are three main memory spaces: registers (per thread), shared (per block), global
- Data can be transferred between host and device using CUDA API: `cudaMalloc()`, `cudaFree()`, `cudaMemcpy()`
- CUDA API calls return value should be checked for errors
- Two advanced read-only memory spaces: constant, texture
- Shared memory can be allocated statically (in the kernel) or dynamically (using `<<<...>>>` syntax)
- Thread synchronisation with `__syncthreads()` is essential for correct behaviour when using shared memory