GLOBAL MEMORY

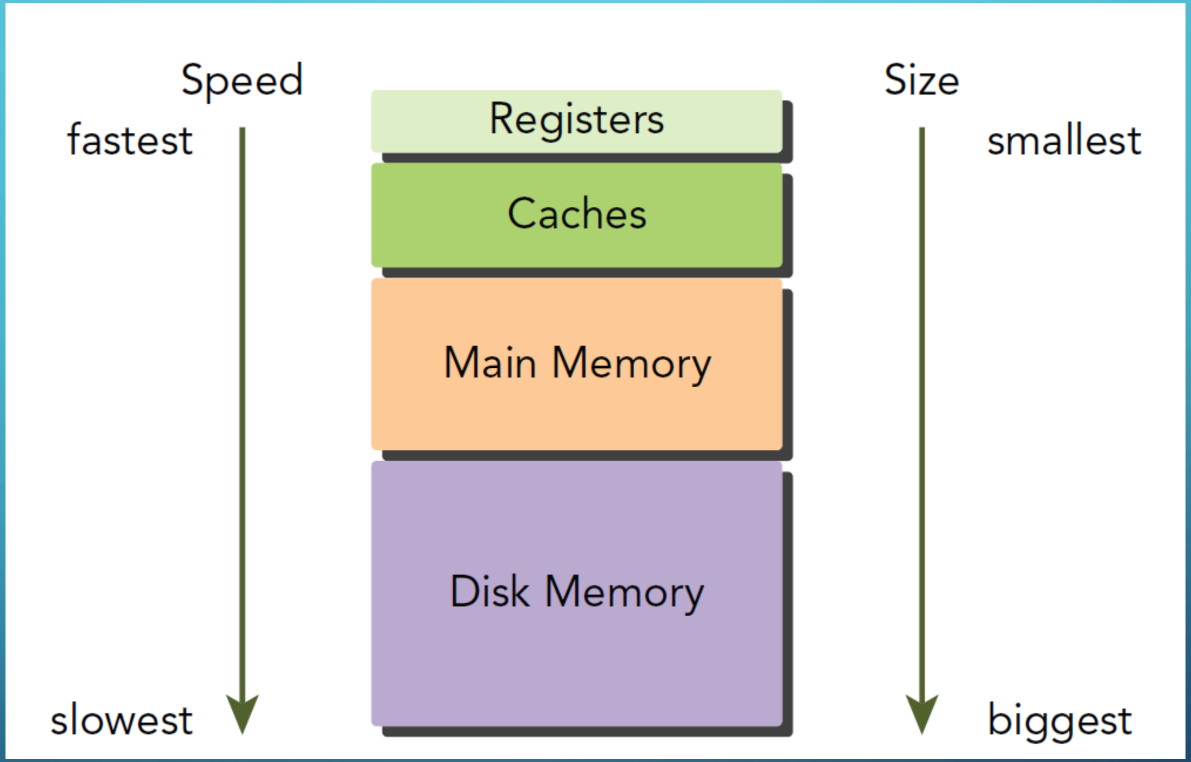
INTRODUCING THE CUDA MEMORY MODEL

- Memory access and management are important parts of any programming language.
- Memory management has a particularly large impact on high performance computing in modern accelerators.
- The CUDA memory model unifies separate host and device memory systems and exposes the full memory hierarchy so that you can explicitly control data placement for optimal performance.

BENEFITS OF A MEMORY HIERARCHY

- Applications often follow the principle of locality, which suggests that they access a relatively small and localized portion of their address space at any point-in-time.
- There are two different types of locality:
  - Temporal locality (locality in time)
  - Spatial locality (locality in space)
- Temporal locality assumes that if a data location is referenced, then it is more likely to be referenced again within a short time period and less likely to be referenced as more and more time passes.
- Spatial locality assumes that if a memory location is referenced, nearby locations are likely to be referenced as well.

BENEFITS OF A MEMORY HIERARCHY



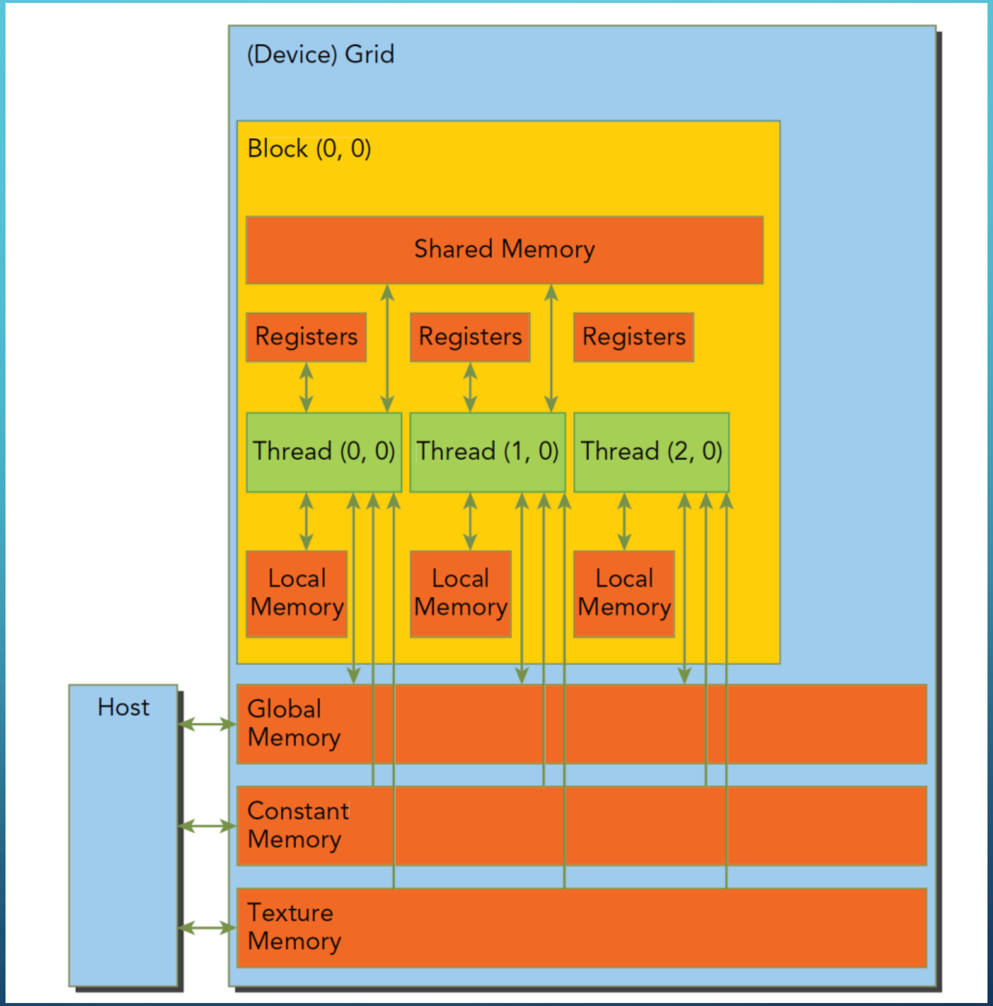
CUDA MEMORY MODEL

- To programmers, there are generally two classifications of memory:
  - Programmable: You explicitly control what data is placed in programmable memory.
  - Non-programmable: You have no control over data placement, and rely on automatic techniques to achieve good performance.
- In the CPU memory hierarchy, L1 cache and L2 cache are examples of non-programmable memory.
- On the other hand, the CUDA memory model exposes many types of programmable memory to you:
  - Registers
  - Shared memory
  - Local memory
  - Constant memory
  - Texture memory
  - Global memory

CUDA MEMORY MODEL

- A thread in a kernel has its own private local memory.
- A thread block has its own shared memory, visible to all threads in the same thread block, and whose contents persist for the lifetime of the thread block.
- All threads can access global memory.
- There are also two read-only memory spaces accessible by all threads: the constant and texture memory spaces.
- The global, constant, and texture memory spaces are optimized for different uses.
- Texture memory offers different address modes and filtering for various data layouts.
- The contents of global, constant, and texture memory have the same lifetime as an application.

CUDA MEMORY MODEL



REGISTERS

- Registers are the fastest memory space on a GPU.
- An automatic variable declared in a kernel without any other type qualifiers is generally stored in a register.
- Arrays declared in a kernel may also be stored in registers, but only if the indices used to reference the array are constant and can be determined at compile time.
- Register variables are private to each thread.
- A kernel typically uses registers to hold frequently accessed thread-private variables.
- Register variables share their lifetime with the kernel.
- Once a kernel completes execution, a register variable cannot be accessed again.

REGISTERS

```
#include <stdio.h>
#define N 5

__global__ void gpu_local_memory(int d_in)
{
  int t_local;
  t_local = d_in * threadIdx.x;
  printf("Value of Local variable in current thread is: %d \n", t_local);
}
int main(int argc, char **argv)
{
  printf("Use of Local Memory on GPU:\n");
  gpu_local_memory << <1, N >> >(5);
  cudaDeviceSynchronize();
  return 0;
}
```

LOCAL MEMORY

- Variables in a kernel that are eligible for registers but cannot fit into the register space allocated for that kernel will spill into local memory.
- Variables that the compiler is likely to place in local memory are:
  - Local arrays referenced with indices whose values cannot be determined at compile-time.
  - Large local structures or arrays that would consume too much register space.
  - Any variable that does not fit within the kernel register limit.

SHARED MEMORY

- Variables decorated with the following attribute in a kernel are stored in shared memory: **__shared__**
- Because shared memory is on-chip, it has a much higher bandwidth and much lower latency than local or global memory.
- Each SM has a limited amount of shared memory that is partitioned among thread blocks.
- Shared memory is declared in the scope of a kernel function but shares its lifetime with a thread block.
- When a thread block is finished executing, its allocation of shared memory will be released and assigned to other thread blocks.

SHARED MEMORY

```
#include <stdio.h>
__global__ void gpu_shared_memory(float *d_a)
{
  int i, index = threadIdx.x;
  float average, sum = 0.0f;
  //Defining shared memory
  __shared__ float sh_arr[10];

  sh_arr[index] = d_a[index];
 // This directive ensure all the writes to shared memory have completed

  __syncthreads();
  for (i = 0; i<= index; i++)
  {
    sum += sh_arr[i];
  }
  average = sum / (index + 1.0f);
  d_a[index] = average;

    //This statement is redundant and will have no effect on overall code execution
  sh_arr[index] = average;
}
```

CONSTANT MEMORY

- Constant memory resides in device memory and is cached in a dedicated, per-SM constant cache.
- A constant variable is decorated with the following attribute: __**constant**__
- Constant variables must be declared with global scope, outside of any kernels.
- A limited amount of constant memory can be declared — 64 KB for all compute capabilities.
- Constant memory is statically declared and visible to all kernels in the same compilation unit.

CONSTANT MEMORY

```
#include "stdio.h"
#include<iostream>
#include <cuda.h>
#include <cuda_runtime.h>

//Defining two constants
__constant__ int constant_f;
__constant__ int constant_g;
#define N 5

//Kernel function for using constant memory
__global__ void gpu_constant_memory(float *d_in, float *d_out)
{
  //Getting thread index for current kernel
  int tid = threadIdx.x;
  d_out[tid] = constant_f*d_in[tid] + constant_g;
}
```

TEXTURE MEMORY

- Texture memory is a type of global memory that is accessed through a dedicated read-only cache.
- The read-only cache includes support for hardware filtering, which can perform floating-point interpolation as part of the read process.
- This memory was originally designed for rendering graphics, but it can also be used for general purpose computing applications.
- It is very effective when applications have memory access that exhibits a great deal of spatial locality.
- It is very effective when applications have memory access that exhibits a great deal of spatial locality.
- Texture memory is optimized for 2D spatial locality, so threads in a warp that use texture memory to access 2D data will achieve the best performance.
- For some applications, this is ideal and provides a performance advantage due to the cache and the filtering hardware.
- However, for other applications using texture memory can be slower than global memory.

GLOBAL MEMORY

- Global memory is the largest, highest-latency, and most commonly used memory on a GPU.
- You can declare a global variable statically in device code using the following qualifier: **__device__**
- Global memory is allocated by the host using **cudaMalloc** and freed by the host using **cudaFree**.
- Pointers to global memory are then passed to kernel functions as parameters.
- Global memory allocations exist for the lifetime of an application and are accessible to all threads of all kernels.
- You must take care when accessing global memory from multiple threads.
- Because thread execution cannot be synchronized across thread blocks, there is a potential hazard of multiple threads in different thread blocks concurrently modifying the same location in global memory, which will lead to an undefined program behavior.

GLOBAL MEMORY

```
#include <stdio.h>
#define N 5

__global__ void gpu_global_memory(int *d_a)
{
  d_a[threadIdx.x] = threadIdx.x;
}

int main(int argc, char **argv)
{
  int h_a[N];
  int *d_a;

  cudaMalloc((void **)&d_a, sizeof(int) *N);
  cudaMemcpy((void *)d_a, (void *)h_a, sizeof(int) *N, cudaMemcpyHostToDevice);

  gpu_global_memory << <1, N >> >(d_a);
  cudaMemcpy((void *)h_a, (void *)d_a, sizeof(int) *N, cudaMemcpyDeviceToHost);

  printf("Array in Global Memory is: \n");
  for (int i = 0; i < N; i++)
  {
    printf("At Index: %d --> %d \n", i, h_a[i]);
  }
  return 0;
}
```

GPU CACHES

- Like CPU caches, GPU caches are non-programmable memory.
- There are four types of cache in GPU devices:
  - L1
  - L2
  - Read-only constant
  - Read-only texture
- There is one L1 cache per-SM and one L2 cache shared by all SMs.
- Both L1 and L2 caches are used to store data in local and global memory.

CUDA VARIABLE AND TYPE QUALIFIER

QUALIFIER	VARIABLE NAME	MEMORY	SCOPE	LIFESPAN
	`float var`	Register	Thread	Thread
	`float var[100]`	Local	Thread	Thread
`__shared__`	`float var` †	Shared	Block	Block
`__device__`	`float var` †	Global	Global	Application
`__constant__`	`float var` †	Constant	Global	Application

† Can be either scalar variable or array variable

SALIENT FEATURES OF DEVICE MEMORY

MEMORY	ON/OFF CHIP	CACHED	ACCESS	SCOPE	LIFETIME
Register	On	n/a	R/W	1 thread	Thread
Local	Off	†	R/W	1 thread	Thread
Shared	On	n/a	R/W	All threads in block	Block
Global	Off	†	R/W	All threads + host	Host allocation
Constant	Off	Yes	R	All threads + host	Host allocation
Texture	Off	Yes	R	All threads + host	Host allocation

REFERENCES

- Professional CUDA C Programming, by John Cheng, Max Grossman, Ty McKercher, Wrox