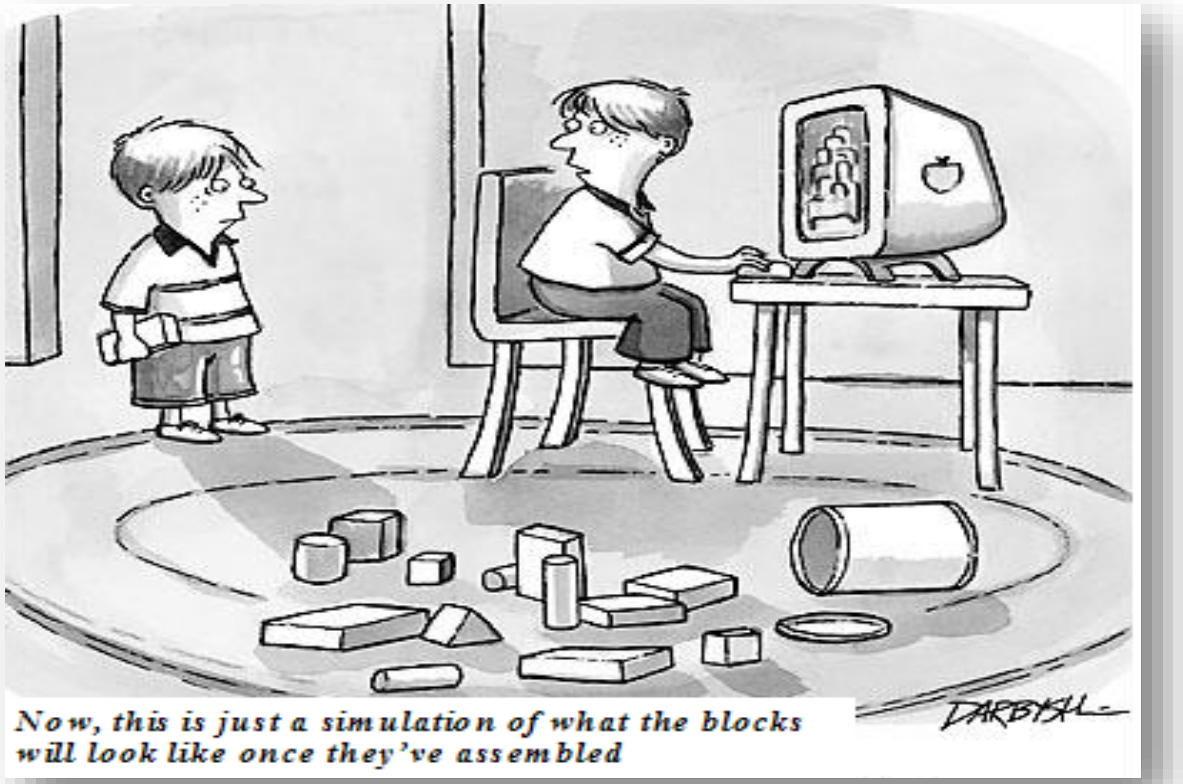
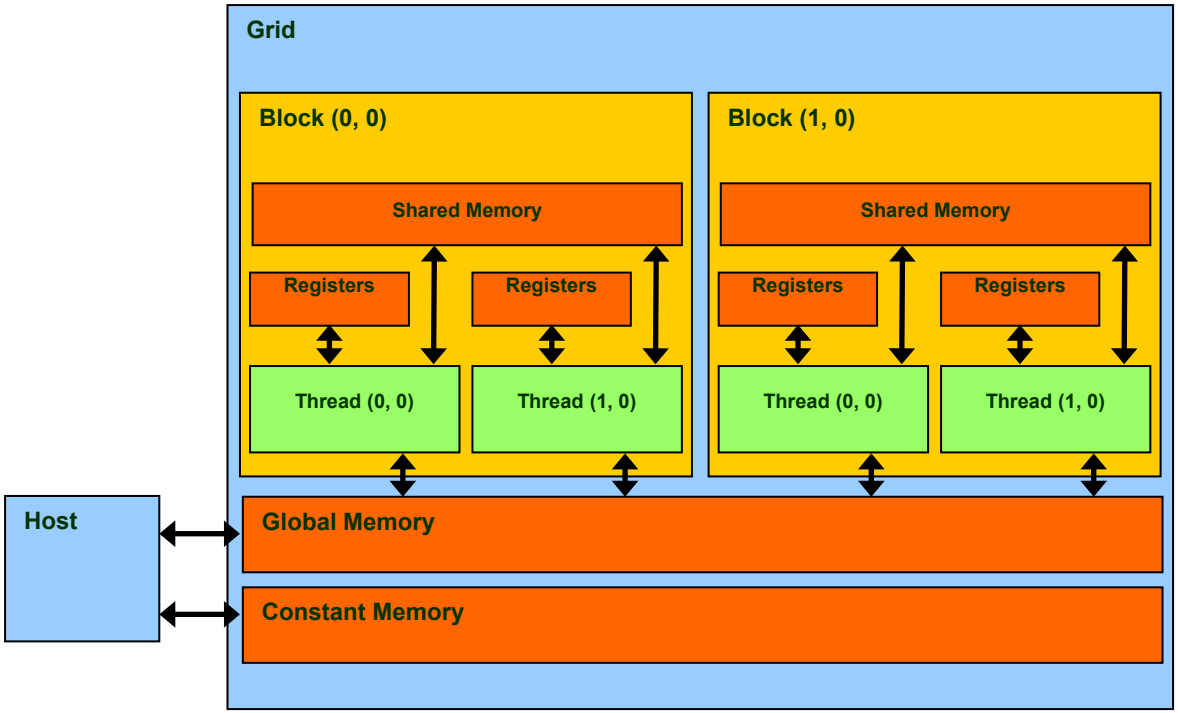
ECE569
Module 16



- CUDA Memories

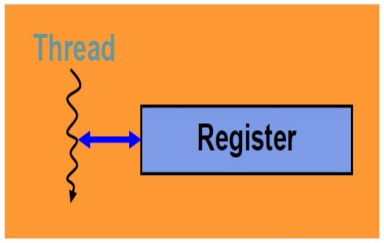
Minimize time spent on memory

- **Move frequently accessed data to fast memory**
  - local < shared << global << host
    - local is either in registers or L1 cache



Declaring CUDA Variables – Inside or Outside the Kernel

Variable declaration	Location	Memory	Cached	Scope	Lifetime
int LocalVar;	**On-chip**	**register**	**N/A**	**thread**	**thread**
__device__ __shared__ int SharedVar;	On-chip	shared	N/A	block	Threads in a block
__device__ int GlobalVar;	Off-chip	global	Yes	grid	application
__device__ __constant__ int ConstantVar;	Off-chip	constant	Yes	grid	application



- **Register:**
  - One version for every thread (private per thread)
    - Carefully selecting a few registers instead of using 50 per thread can easily double the number of concurrent blocks.

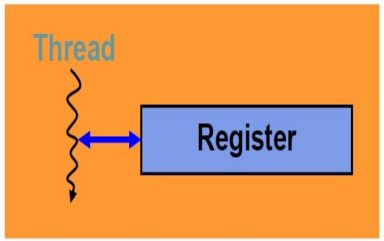
Compute Capability (Registers, Shared Memory)

Technical Specifications	3.5	3.7	5.0	5.2	5.3	6.0	6.1	6.2	7.0	7.2	7.5	8.0	8.6
Max # of resident blocks per SM	16		32								16	32	16
Max # of resident warps per SM	64										32	64	48
Max # of resident threads per SM	2048										1024	2048	1536
Number of 32-bit registers per SM	64 K	128 K						64 K					
Max # of 32-bit registers/thread block	64 K				32 K	64 K		32 K	64 K				
Max # of 32-bit registers per thread	255												
Maximum shared memory per SM	48 KB	112 KB	64 KB	96 KB	64 KB		96 KB	64 KB	96 KB		64 KB	164 KB	100 KB
Shared memory per thread block	48 KB								96 KB	48 KB	64 KB	163 KB	99 KB
Number of shared memory banks	32												
Local memory per thread	512 KB												
Constant memory size	64 KB												

- To run maximum number of threads on CUDA compute capability 6.1
  - How many registers per thread should be used?

Declaring CUDA Variables – Inside or Outside the Kernel

Variable declaration	Location	Memory	Cached	Scope	Lifetime
int LocalVar;	**On-chip**	**register**	**N/A**	**thread**	**thread**
__device__ __shared__ int SharedVar;	On-chip	shared	N/A	block	Threads in a block
__device__ int GlobalVar;	Off-chip	global	Yes	grid	application
__device__ __constant__ int ConstantVar;	Off-chip	constant	Yes	grid	application



- **Register:**
  - **Local memory** is used automatically by NVCC when we run out of registers or when registers cannot be used - **register spilling.**

- Automatic variables reside in a register
  - Except per-thread arrays that reside in global memory

Ways to minimize time spent on memory accesses

- **Move frequently accessed data to fast memory**
  - **Indicate local variables. Choose all that apply:**

```
/* using local memory */

// a __device__ or __global__ function runs on the GPU
__global__ void use_local_memory_GPU(float in, float *x)
{
  float f;
  f = x[in];
}
int main(int argc, char **argv)
{
    /*  Call a kernel that shows using local memory */
    use_local_memory_GPU<<<1, 128>>>(2.0f);
}
```

- ❑ f
- ❑ in
- ❑ x
- ❑ x[in]

Ways to minimize time spent on memory accesses

- **Move frequently accessed data to fast memory**
  - Local > Shared >> Global

```
/* using local memory */

// a __device__ or __global__ function runs on the GPU
__global__ void use_local_memory_GPU(float in, float* x)
{
  float f; //variable "f" is in local memory and private to each thread
  f = x[in]; //parameter "in" is in local memory and private to each
thread
}
int main(int argc, char **argv)
{
    /*  Call a kernel that shows using local memory */
    use_local_memory_GPU<<<1, 128>>>(2.0f);
}
```

Registers

- What happens if your application uses a little array?

```
__global__ void lap(float *u) {
float ut[3];
int tid = threadIdx.x+blockIdx.x*blockDim.x;
for (int k=0; k<3; k++)
  ut[k] = u[tid+k*gridDim.x*blockDim.x];
for (int k=0; k<3; k++)
 u[tid+k*gridDim.x*blockDim.x] =
 A[3*k]*ut[0]+A[3*k+1]*ut[1]+A[3*k+2]*ut[2];
}
```

Local Arrays

- **compiler converts to scalar registers:**

```
__global__ void lap(float *u) {
int tid = threadIdx.x + blockIdx.x*blockDim.x;
float ut0 = u[tid+0*gridDim.x*blockDim.x];
float ut1 = u[tid+1*gridDim.x*blockDim.x];
float ut2 = u[tid+2*gridDim.x*blockDim.x];

u[tid+0*gridDim.x*blockDim.x] =
A[0]*ut0 + A[1]*ut1 + A[2]*ut2;
u[tid+1*gridDim.x*blockDim.x] =
A[3]*ut0 + A[4]*ut1 + A[5]*ut2;
u[tid+2*gridDim.x*blockDim.x] =
A[6]*ut0 + A[7]*ut1 + A[8]*ut2;
}
```

Local Arrays

- **In more complicated cases, it puts the array into device memory**
  - still referred to in the documentation as a “local array” because each thread has its own private copy
  - held in L1 cache by default, may never be transferred to device memory
- **Assume 48KB of L1 cache and 1024 threads**
  - *What is the largest integer type array size we can declare without spilling over the L1 cache?*

Local Arrays

- **In more complicated cases, it puts the array into device memory**
  - still referred to in the documentation as a “local array” because each thread has its own private copy
  - held in L1 cache by default, may never be transferred to device memory
- **Assume 48KB of L1 cache and 1024 threads**
  - What is the largest integer type array size we can declare without spilling over the L1 cache?
    - equates to 12K 32-bit variables, which is only 12 per thread when using 1024 threads
    - **If L1 missed we have to go to global memory eventually**

Declaring CUDA Variables – Inside or Outside the Kernel

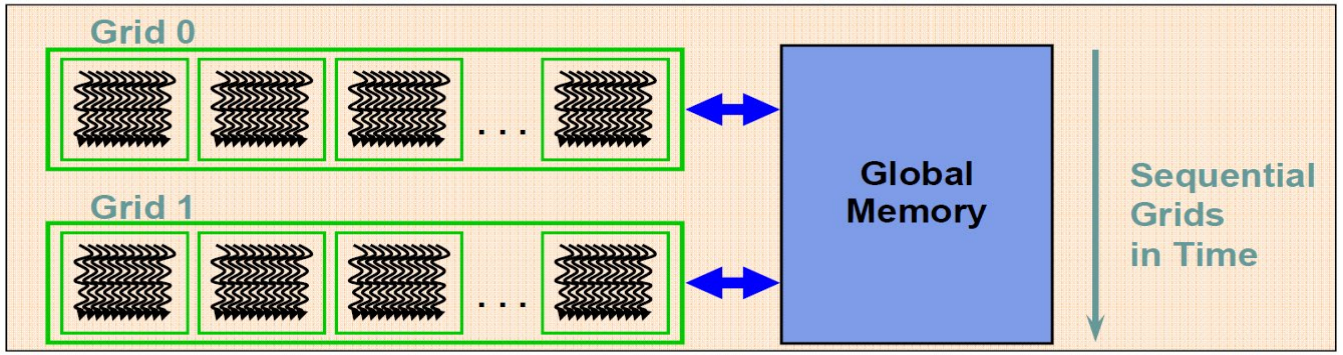
Variable declaration	Location	Memory	Cached	Scope	Lifetime
int LocalVar;	On-chip	register	N/A	thread	thread
__device__ __shared__ int SharedVar;	On-chip	shared	N/A	block	Threads in a block
__device__ int GlobalVar;	**Off-chip**	**global**	**Yes**	**grid**	**application**
__device__ __constant__ int ConstantVar;	Off-chip	constant	Yes	grid	application

- **Global:**
  - __device__ prefix indicates global variable
  - read and modify by any kernel
  - lifetime of the whole application
  - can declare arrays of fixed size
  - can read/write by host code using special routines
    - cudaMemcpyToSymbol, cudaMemcpyFromSymbol or with standard cudaMemcpy in combination with cudaGetSymbolAddress

Declaring CUDA Variables – Inside or Outside the Kernel

Variable declaration	Location	Memory	Cached	Scope	Lifetime
int LocalVar;	On-chip	register	N/A	thread	thread
__device__ __shared__ int SharedVar;	On-chip	shared	N/A	block	Threads in a block
__device__ int GlobalVar;	**Off-chip**	**global**	**Yes**	**grid**	**application**
__device__ __constant__ int ConstantVar;	Off-chip	constant	Yes	grid	application

- **Global**
  - One version of the variable, all threads in a grid will see the same version, even though they may not see the most up to date information,
  - Blocks can execute out of order (read/write)
    - requires synchronization



Declaring CUDA Variables – Inside or Outside the Kernel

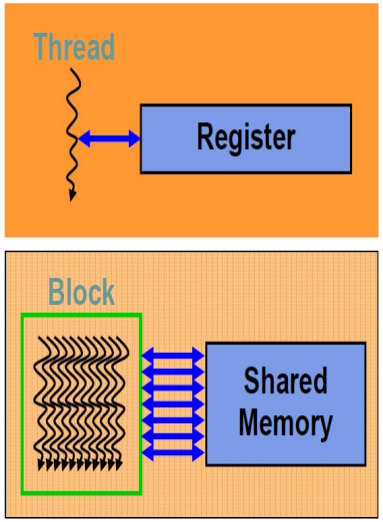
Variable declaration	Location	Memory	Cached	Scope	Lifetime
int LocalVar;	On-chip	register	N/A	thread	thread
__device__ __shared__ int SharedVar;	On-chip	shared	N/A	block	Threads in a block
__device__ int GlobalVar;	Off-chip	global	Yes	grid	application
__device__ __constant__ int ConstantVar;	**Off-chip**	**constant**	**Yes**	**grid**	**application**

- **Constant:**
  - Very similar to global variables, except that they can't be modified by kernels
  - defined using the prefix __constant__
  - initialized by the host code
    - cudaMemcpyToSymbol, cudaMemcpyFromSymbol or cudaMemcpy in combination with cudaGetSymbolAddress
  - Only 64KB of constant memory
    - big benefit is that each SM has a 8-10KB cache
    - when all threads read the same constant, almost **as fast as a register**
    - **doesn't tie up a register**, so very helpful in minimizing the total number of registers required

Declaring CUDA Variables – Inside or Outside the Kernel

Variable declaration	Location	Memory	Cached	Scope	Lifetime
int LocalVar;	On-chip	register	N/A	thread	thread
__device__ __shared__ int SharedVar;	On-chip	shared	N/A	block	Threads in a block
__device__ int GlobalVar;	Off-chip	global	Yes	grid	application
__device__ __constant__ int ConstantVar;	Off-chip	constant	Yes	grid	application

– `__device__` is optional when used with `__shared__`, or `__constant__`



- **Shared**
  - **Explicitly defined and used in the kernel code**
  - **Every block has its own version** of the shared memory variable, all threads in a block will read/write to the same version of the shared variable,
    - **may not see the most up to date** data for the variable, we will need synchronization
  - **Bank conflicts** can slow access down.
  - Fastest when
    - all threads read from different banks or
    - all threads of a warp read exactly the same value

Next

- **CUDA Memory Review**
  - Exercises