GPU Architecture

National Tsing-Hua University

2019, Summer Semester

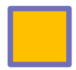
Outline

- Thread execution
  - Execution model
  - Warp
  - Warp Divergence
- Memory hierarchy

Execution Model

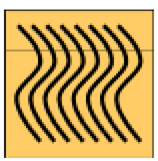
Software | Hardware

Thread

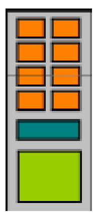


Scalar processor

Threads are executed by scalar processor



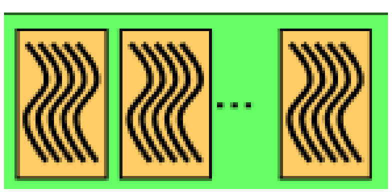
Thread block



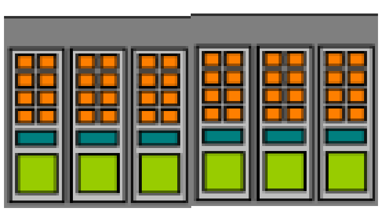
Stream Processor (SM)

Thread blocks are executed on SM

Several concurrent thread block can reside on one SM



Grid



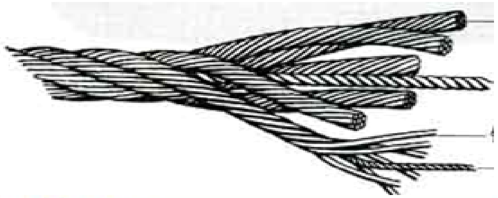
GPU device

A kernel is launched as a grid of thread blocks

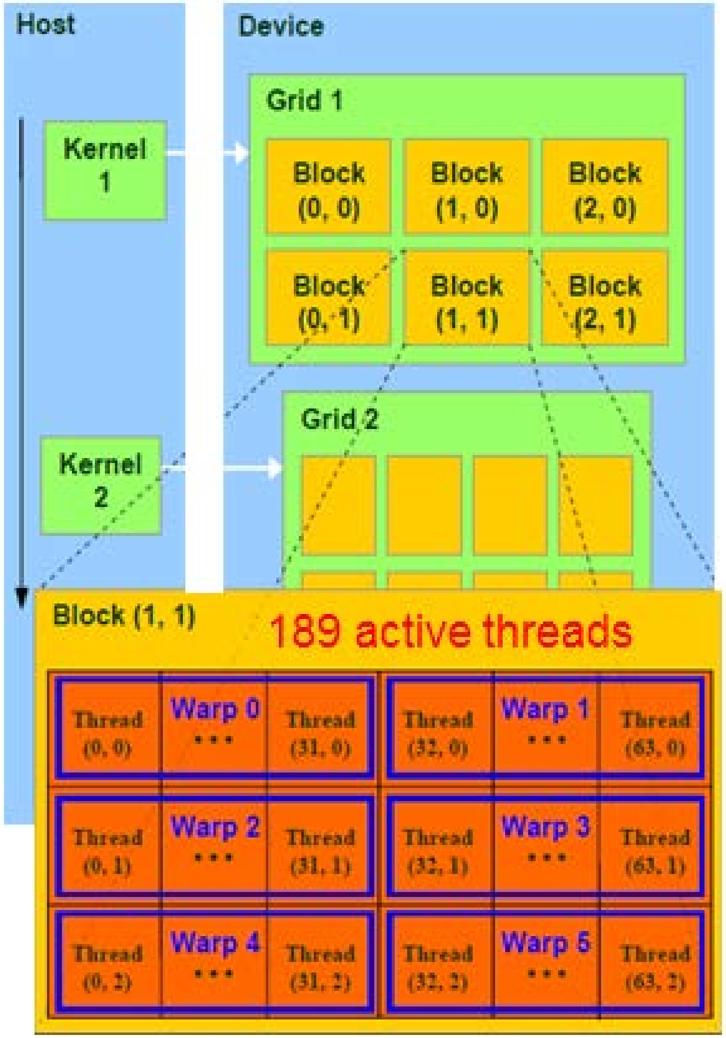
Thread Execution

- CUDA threads are grouped into blocks
  - All threads of the same block are executed in an SM
  - SMs have shared memories, where threads within a block can communicate
  - The entire threads of a block must be executed completely before there is space to schedule another thread block
- Hardware schedules thread blocks onto available SMs
  - No guarantee of order of execution
  - If an SM has more resources, the hardware can schedule more blocks

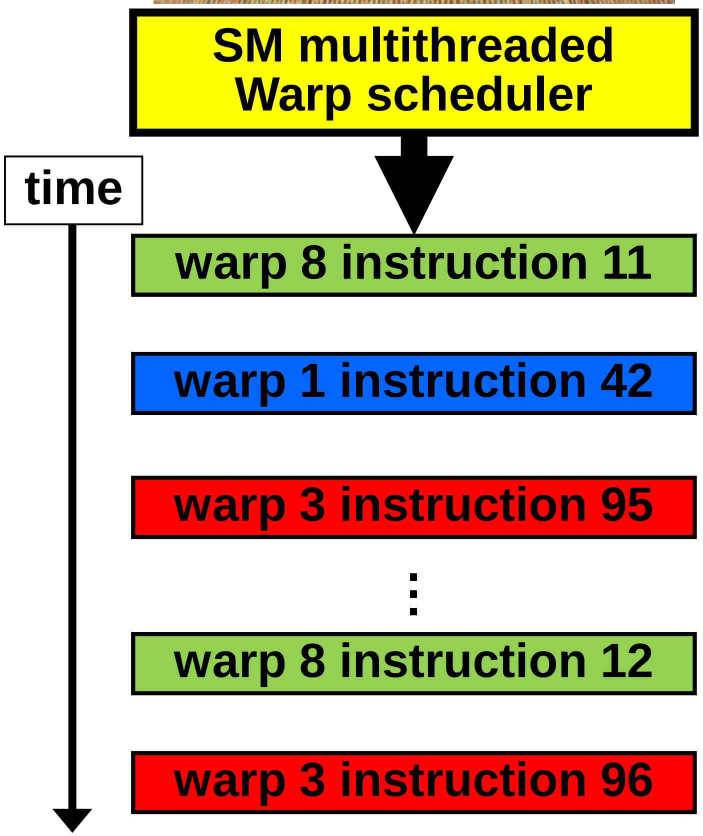
Warp



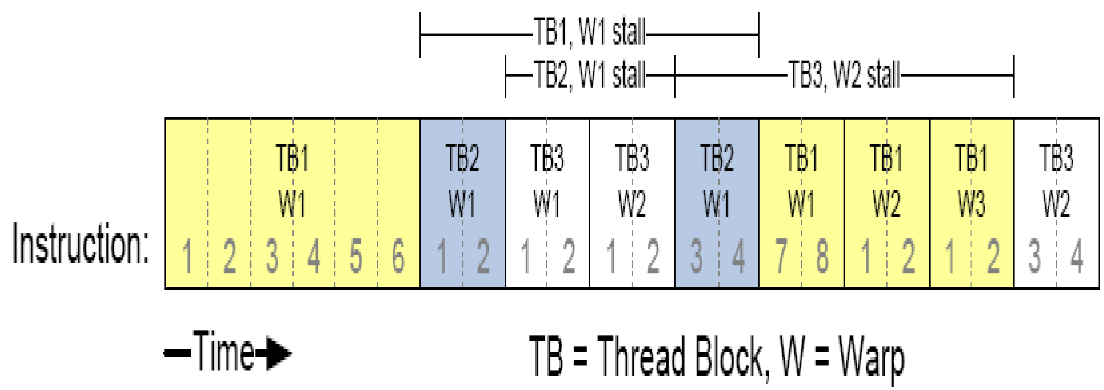
- Inside the SM, threads are launched in groups of 32, called warps
  - Warps share the control part (warp scheduler)
  - At any time, only one warp is executed per SM
  - Threads in a warp will be executing the same instruction (SIMD)
- In other words …
  - Threads in a wrap execute **physically** in parallel
  - Warps and blocks execute **logically** in parallel
  - ➔ Kernel needs to sync threads within a block
- For Fermi:
  - Maximum number of active blocks per SM is 8
  - Maximum number of active warps per SM is 48
  - Maximum number of active threads per SM is 48*32=1,536



Warp Scheduler



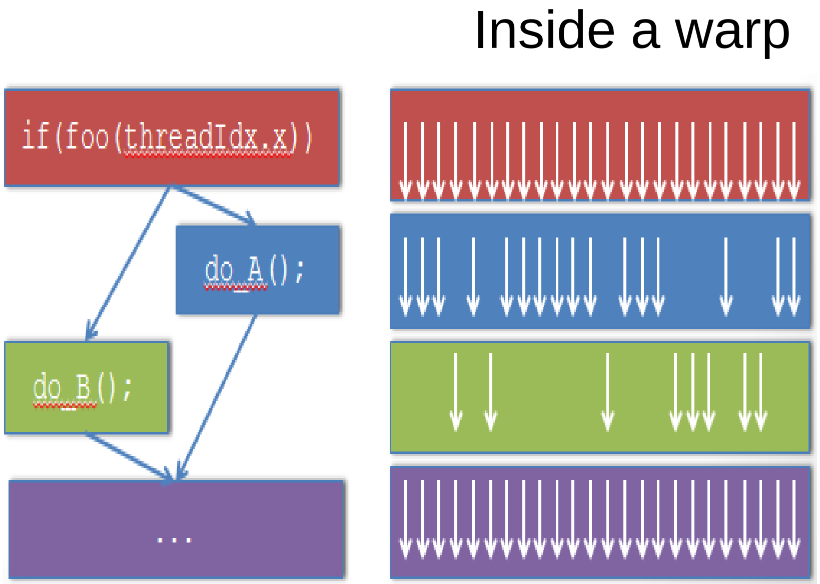
- SM hardware implements zero-overhead Warp scheduling
  - **Warps whose next instruction has its operands ready for consumption are eligible for execution**
  - **Wraps are switched when memory stalls**
  - **Eligible Warps are selected for execution on prioritized scheduling**
  - **All threads in a Warp execute the same instruction when selected**



Warp Divergence

- What if different threads in a warp need to do different things:
  - Including any flow control instruction (**if, switch, do, for, while**)

```
if(foo(threadIdx.x)){
        do_A();
} else {
        do_B();
}
```



- Different execution paths within a warp are serialized
  - Predicated instructions which are carried out only if logical flag is true
  - All threads compute the logical predicate and two predicated instructions/statements
  - ➔ **Potential large lost of performance**

Avoid Diverging in a Warp

- Example with divergence:

```
if (threadIdx.x > 2) {...}
else {...}
```

  - Branch granularity < warp size

- Example without divergence:

```
if (threadIdx.x / WARP_SIZE > 2) {...}
else {...}
```

  - Different warps can execute different code with no impact on performance
  - Branch granularity is a whole multiple of warp size

Iteration Divergence

- A single thread can drag a whole warp with it for a long time
- Know your data patterns
- If data is unpredictable, try to flatten peaks by letting threads work on multiple data items

Unroll the for-loop

- Unroll the statements can reduce the branches and increase the pipeline
- Example:

```
for (i=0;i<n;i++) {
    a = a + i;
}
```

  - Unrolled 3 times

```
for (i=0;i<n;i+=3) {
    a = a + i;
    a = a + i+1;
    a = a + i+2;
}
```

#pragma unroll

- The `#pragma unroll` directive can be used to control unrolling of any given loop.
- must be placed immediately before the loop and only applies to that loop
- Example:

```
#pragma unroll 5
for (int i = 0; i < n; ++i)
```

  - the loop will be unrolled 5 times.
  - The compiler will also insert code to ensure correctness
- `#pragma unroll 1` will prevent the compiler from ever unrolling a loop.

Atomic Operations

- Occasionally, an application may need threads to update a counter in **shared** or **global** memory

```
__shared__ int count;
……
if (……) count++;
```

- Synchronization problem: if two (or more) threads execute this statement at the same time
- Solution: use atomic instructions supported by GPU
  - addition / subtraction
  - max / min
  - increment / decrement
  - compare-and-swap

Example: Histogram

```
/* Determine frequency of colors in a picture
colors have already been converted into ints. Each
thread looks at one pixel and increments a counter
atomically*/

__global__ void hist(int* color, int* bin){
   int i = threadIdx.x + blockDim.x *
                         blockIdx.x;
   int c = colors[i];
   atomicAdd(&bin[c], 1);
}
```

Example: Global Min/Max

```
__global__ void global_max(int* values,
int* gl_max){
    int i = threadIdx.x + blockDim.x *
                          blockIdx.x;
    int val = values[i];
    atomicMax(gl_max,val);
}
```

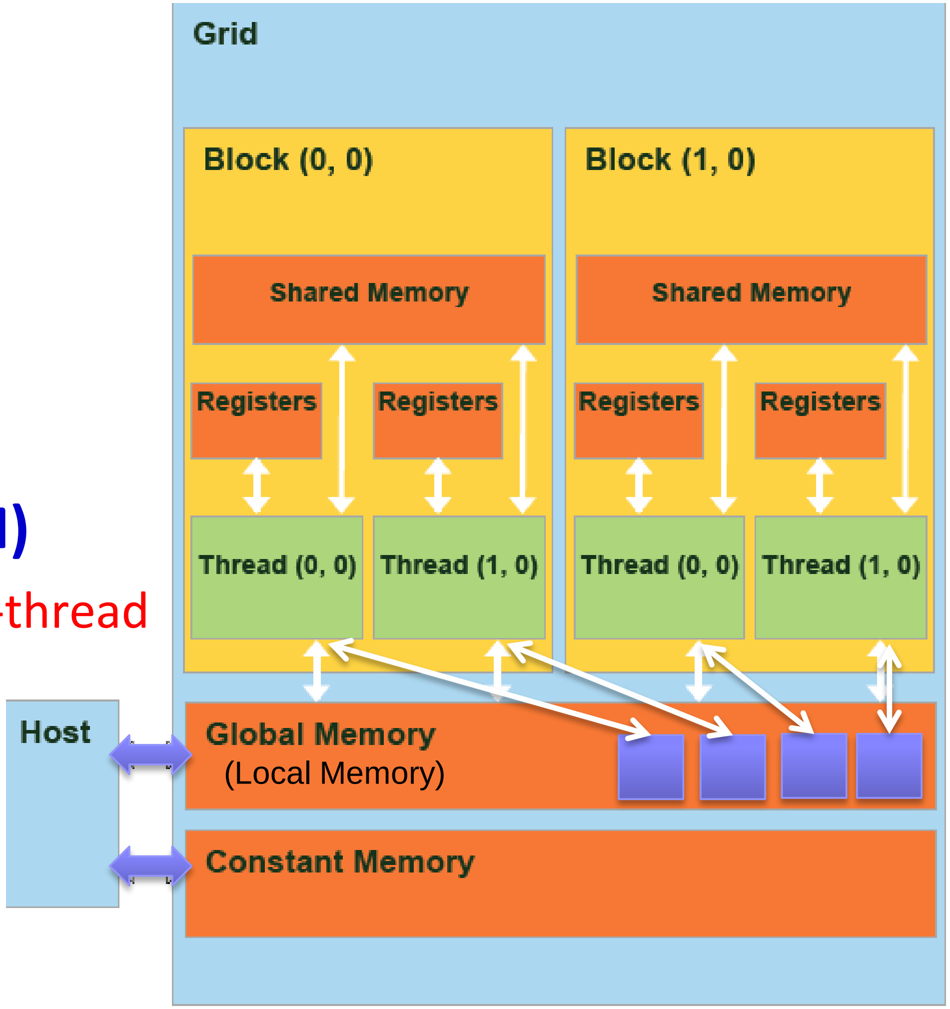
- Not very fast for data in shared memory
- Only slightly slower for data in device memory

Outline

- Thread execution
- Memory hierarchy
  - Register & Local memory
  - Shared memory
  - Global & Constant memory

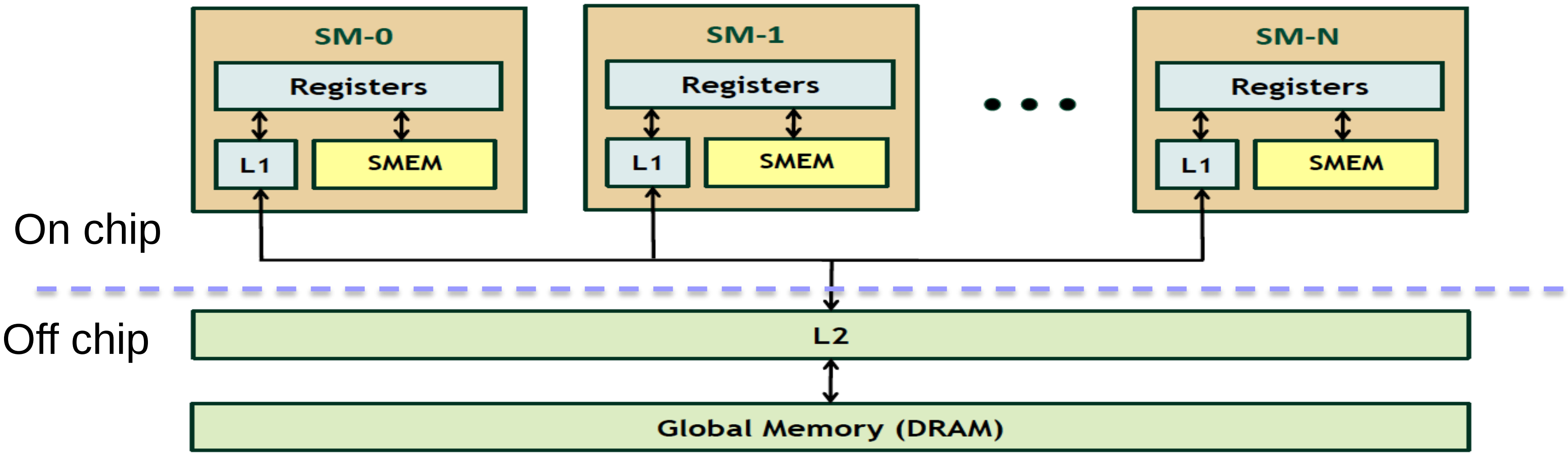
GPU Memory Hierarchy

- **Registers**
  - Read/write per-thread
  - Low latency & High BW
- **Shared memory**
  - Read/write per-block
  - Similar to register performance
- **Global/Local memory (DRAM)**
  - Global is per-grid & Local is per-thread
  - High latency & Low BW
  - Not cached
- **Constant memory**
  - Read only per-grid
  - Cached



Memory Access

- **A store writes a line to L1**
  - If evicted, that line is written to L2
  - The line could also be evicted from L2 and written to DRAM (global mem.)
- **A load requests the line from L1**
  - If a hit, operation is complete
  - If a miss, then requests the line from L2
    - If a miss, then requests the line from DRAM (global memory)
- **Only GPU threads can access local memory addresses**



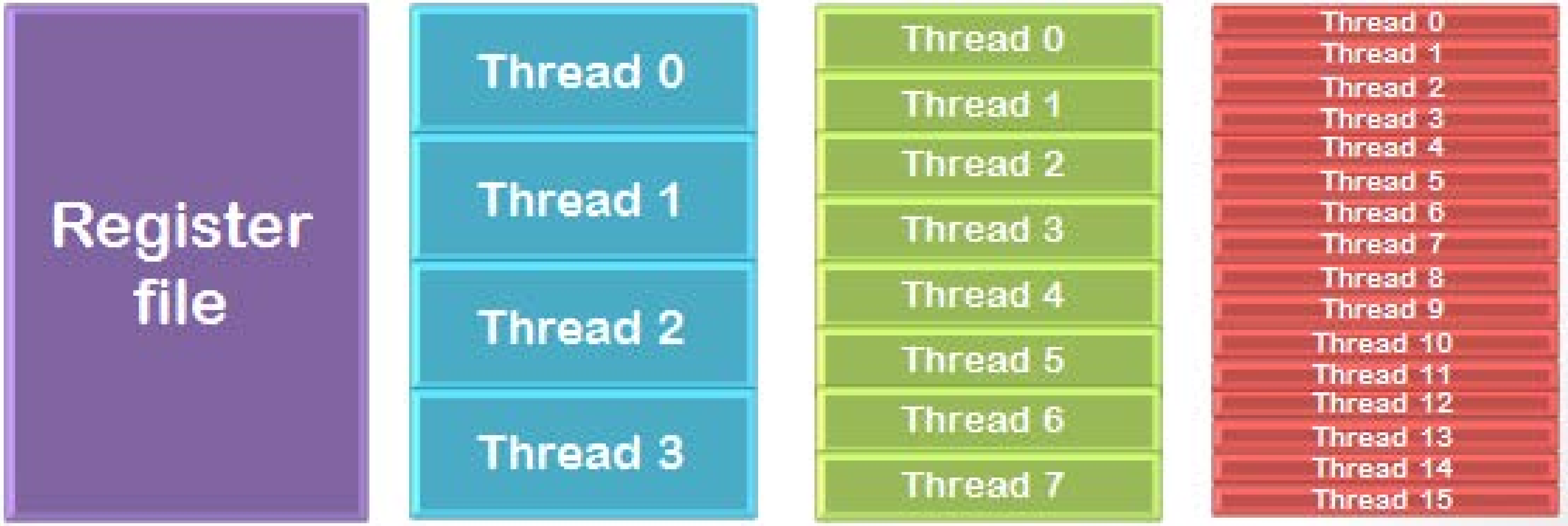
Register

- Register consumes zero extra clock cycles per instruction, except
  - Register read-after-write dependencies (24 cycles) and
  - Register memory bank conflicts

- Register spilling
  - Max number of register per threads is 63
  - Local memory is used if the register limit is met
  - **Array variables** always are allocated in **local memory (DRAM)**
  - Max amount of local memory per thread is 512K

Register Pressure

- Too few threads
  - can't hide pipeline / memory access latency
- Too many threads
  - register pressure
  - Limited number of registers among concurrent threads
  - Limited shared memory among concurrent blocks



Local Memory

- Name refers to memory where registers and other thread-data is spilled
  - Usually when one runs out of SM resources
  - "Local" because each thread has its own private area
- Details:
  - Not really a "memory" – bytes are stored in **global memory (DRAM)**
- Differences from global memory:
  - Addressing is resolved by the compiler
  - Stores are cached in L1

Example

```
__device__ void distance(int m, int n, int *V){
    int i, j, k;
    int a[10], b[10], c[10];
    ...
}
```

- Variables i, j, k, a, b, c are called “local variables”.
- It is likely that variable i, j, k are stored in registers, and variable a, b, c are stored in “local memory” (off-chip DRAM).
  - Compiler decides which memory space to use.
  - Registers aren’t indexable, so arrays have to be placed in local memory.
  - If not enough registers, local memory will be used.
- Only allowed static array!! ➔ No `int a[m];`

Outline

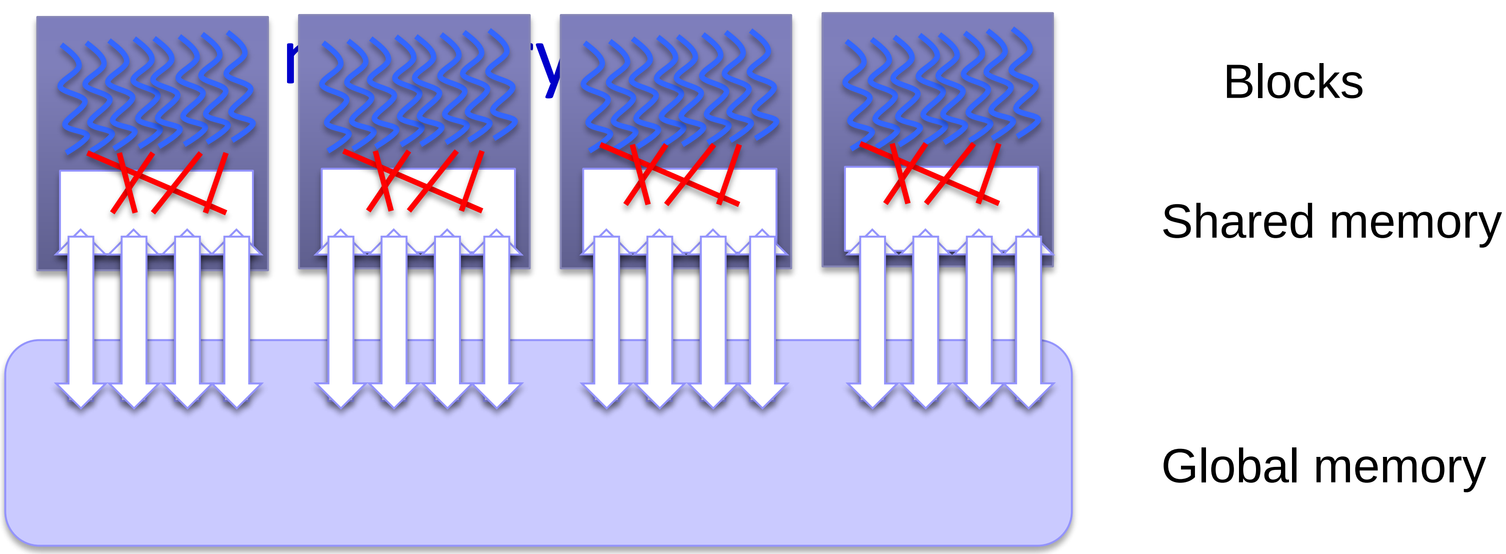
- Thread execution
- Memory hierarchy
  - Register & Local memory
  - Shared memory
  - Global & Constant memory
- Occupancy

Shared Memory

- Programmable cache!!
  - **Almost as fast as registers**
- Scope: shared by all the threads in a block.
  - The threads in the same block can communicate with each other through the shared memory.
  - Threads in different blocks can only communicate with each other through global memory.
- Size: at most 48K per block
  - On Fermi/Kepler GPU, shared memory and L1 cache use the same memory hardware (64K). Programmers can decide the ratio of shared memory and L1 cache:
  - The ratio (shared:L1) can be (3:1) or (1:1) or (1:3).

General Strategy

1. Load data from global memory to shared memory
2. Process data in the shared memory
3. Write data back from shared memory to



APSP Parallel Implementation Revisit

- Use **n*n threads**.
- Each updates the shortest path of one pair vertices
- Use **global memory** to store the matrix D.

```
__global__ void FW_APSP(int k, int D[n][n]) {
     int i = threadIdx.x;
     int j = threadIdx.y;
      if (D[i][j]>D [i][k]+D[k][j])
        D[i][j]= D[i][k]+D[k][j];
}
int main() { ...
    dim3 threadsPerBlock(n, n);
    for (int k = 0; k<n, k++)
        FW_APSP<<<1, threadsPerBlock >>>(k, D);
}
```

6 global memory access

Using Shared Memory

- This way of using shared memory is called dynamic allocation of shared memory, whose size is specified in the kernel launcher.

  `FW_APSP<<<1,n*n, n*n*sizeof(int)>>>(…);`

  - The third parameter is the size of shared memory.

```
extern __shared__ int S[][];
__global__ void FW_APSP(int k, int D[n][n]) {
    int i = threadIdx.x;
     int j = threadIdx.y;
    S[i][j]=D[i][j]; // move data to shared memory
    __syncthreads();
    // do computation
     if (S[i][j]>S[i][k]+S[k][j])
          D[i][j]= S[i][k]+S[k][j];
}
```

ONLY 2
global mem
access

Limit of Dynamic Allocation

- If you have multiple extern declaration of shared:

```
extern __shared__ float As[];
extern __shared__ float Bs[];
```

  this will lead to As pointing to the same address as Bs.

- Solution: keep `As` and `Bs` inside the 1D-array.

```
extern __shared__ float smem[];
```

- Need to do the memory management yourself
  - When calling kernel, launch it with size of `sAs+sBs`, where `sAs` and `sBs` are the size of `As` and `Bs` respectively.
  - When indexing elements in `As`, use `smem[0:sAs-1]`; when indexing elements in `Bs`, use `smem[sAs:sAs+sBs]`.

Using Shared Memory

- `FW_APSP<<<1,n*n, n*n*sizeof(int)>>>(…);`
  - The third parameter is the size of shared memory.

```
extern __shared__ int S[];
__global__ void FW_APSP(int* k, int* D,int* n) {
    int i = threadIdx.x;
     int j = threadIdx.y;
    S[i*(*n)+j]=D[i*(*n)+j]; //move data to shared memory
    __syncthreads();
    // do computation
     if (S[i*(*n)+j]>S[i*(*n)+k]+S[k*(*n)+j])
          D[i*(*n)+j]= S[i*(*n)+k]+S[k*(*n)+j];
}
```

Static Shared Memory Allocation

- If the size of shared memory is known in compilation time, shared memory can be allocated statically.

```
__global__ void FW_APSP(int k, int D[][]){
    __shared__ int DS[10*10];
    ...
}
```

Must know n=10 at compile time

Outline

- Thread execution
- Memory hierarchy
  - Register & Local memory
  - Shared memory
  - Global & Constant memory

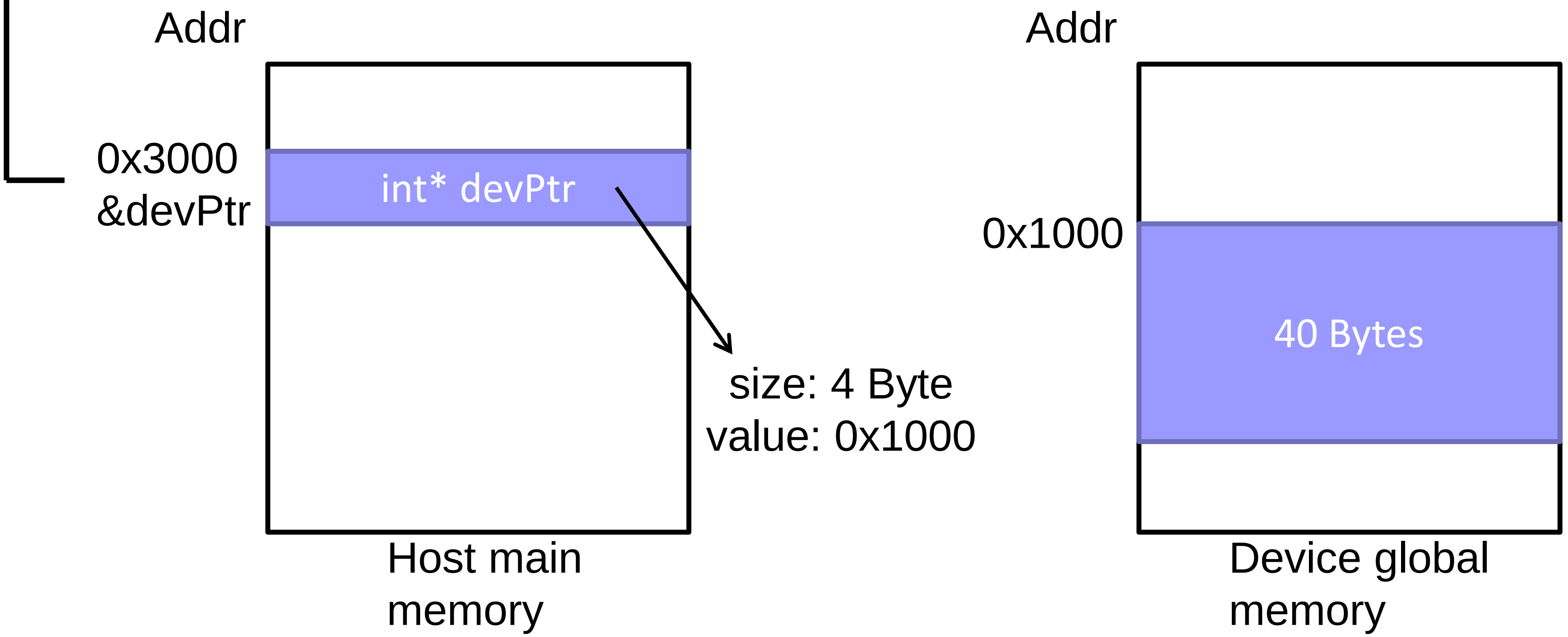
Global Memory in Kernel

- Through the kernel launcher arguments
  - Need to use `cudaMalloc` to allocate memory and use `cudaMemcpy` to set their values.
  - This method is what we used in previous examples.

```
cudaMemcpy( void *dst, const void
*src, size_t count, enum
cudaMemcpyKind kind)
```

How to Allocate Device (Global) Memory

1. `cudaMalloc(void **devPtr, size_t size)`
   - `devPtr`: return the address of the allocated memory on **device**
   - `size`: the allocated memory size (**bytes**)
2. `cudaFree (void *devPtr)`



Constant Memory

- Same usage and scope as the global memory except
  - **Read only**
  - Declare by variable qualifier `__constant__`
  - Move by `cudaMemcpyToSymbol()` & `cudaMemcpyFromSymbol()`
- Each SM has its own constant memory
  - The constant memory on each SM is of size 64K, and has a separated cache, of size 8K.

```
__constant__ int constData[100];
int main(void) {
    int A[100];
    cudaMemcpyToSymbol(constData, A, sizeof(A));
    add<<<grid_size,blk_size>>>();
    cudaMemcpyFromSymbol(A, constData, sizeof(A));
}
__global__ kernel() {
    int v = constData[threadIdx];
}
```

CUDA Variables within a Kernel

Variable declaration	Memory	Scope	Lifetime
`int var`	Register	Thread	Thread
`int array_var[10]`	Local	Thread	Thread
`__shared__ int shared_var`	Shared	Block	Block
`__device__ int global_var`	Global	Grid	App
`__constant__ int constant_var`	Constant	Grid	App

- Scalar variables without qualifier reside in a register
  - Compiler will spill to thread local memory
- Array variables without qualifier reside in thread-local memory

Memory Speed

Variable declaration	Memory	Speed
int var	Register	1x
int array_var[10]	Local	100x
__shared__ int shared_var	Shared	1x
__device__ int global_var	Global	100x
__constant__ int constant_var	Constant	1x

- Scalar variables reside in fast, on-chip registers
- Shared variables reside in fast, on-chip memories
- Thread-local arrays & global variables reside in uncached off-chip memory
- Constant variables reside in cached off-chip memory

Memory Scale

Variable declaration	Total no. of variables	Visible by no. of threads
`int var`	100,000	1
`int array_var[10]`	100,000	1
`__shared__ int shared_var`	100	100
`__device__ int global_var`	1	100,000
`__constant__ int constant_var`	1	100,000

- 100Ks per-thread variables, R/W by 1 thread
- 100s shared variables, each R/W by 100s of threads
- Global variable is R/W by 100Ks threads
- 1 constant variable is readable by 100Ks threads

Quiz

CUDA Variables within a Kernel

Local variable

Shared variable

Global variable

Register

Local Memory

Shared Memory

Global Memory

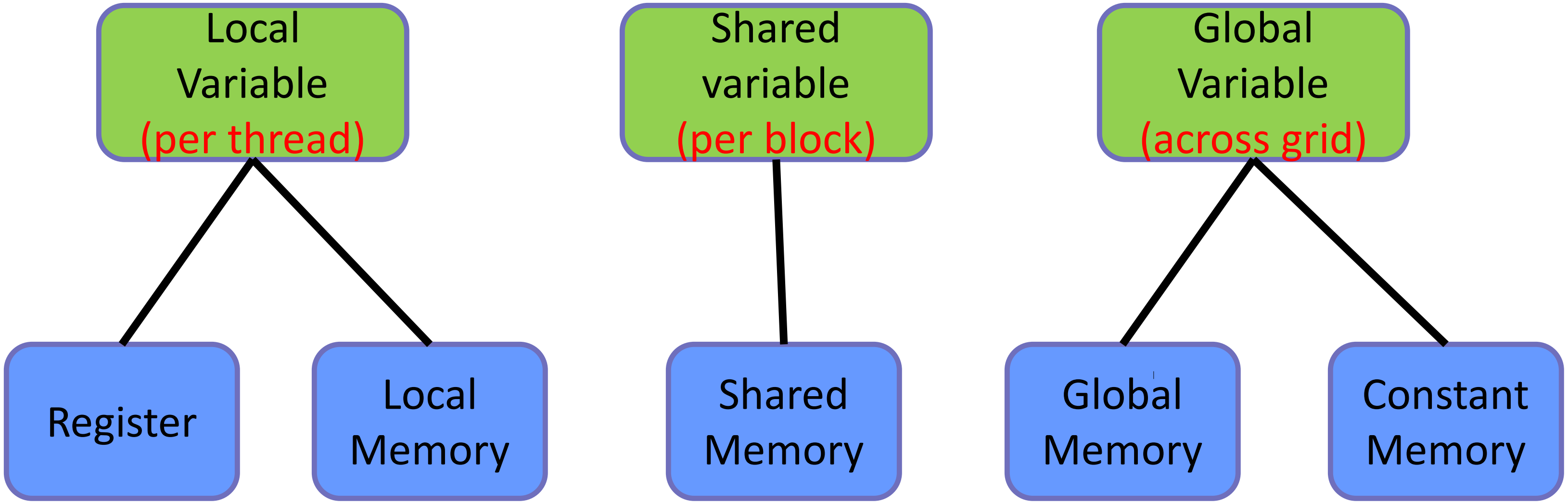
Constant Memory

GPU Memory Hierarchy

Q: What is the data scope of each variable, and what is the mapping between software and hardware?

Quiz

CUDA Variables within a Kernel



GPU Memory Hierarchy

Q: What is the data scope of each variable, and what is the mapping between software and hardware?

Reference

- **NVIDIA CUDA Library Documentation**
  - http://developer.download.nvidia.com/compute/cuda/4_1/rel/toolkit/docs/online/index.html
- **NVIDIA CUDA Warps and Occupancy**
  - http://on-demand.gputechconf.com/gtc-express/2011/presentations/cuda_webinars_WarpsAndOccupancy.pdf
- **Heterogeneous computing course slides from Prof. Che-Rung Lee**