CS 6023 - GPU Programming

Full CUDA Program and Hardware Mapping

08/02/2019

Agenda

- Thread hierarchies in CUDA program
- CUDA memories
- A full CUDA program
- Mapping CUDA threads to hardware

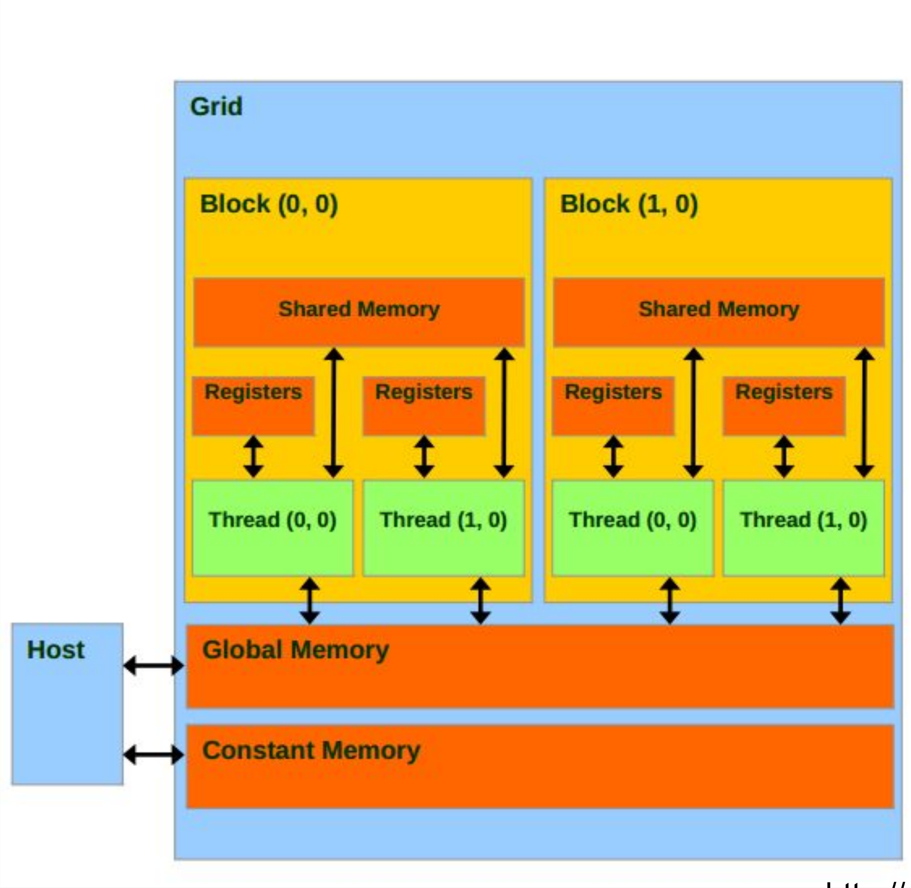
Recall: Grid/block/warp/thread

- Thread is a
- Warp is a
- Block is a
- All blocks in a grid have
- All threads in a grid share

Recall: Grid/block/warp/thread

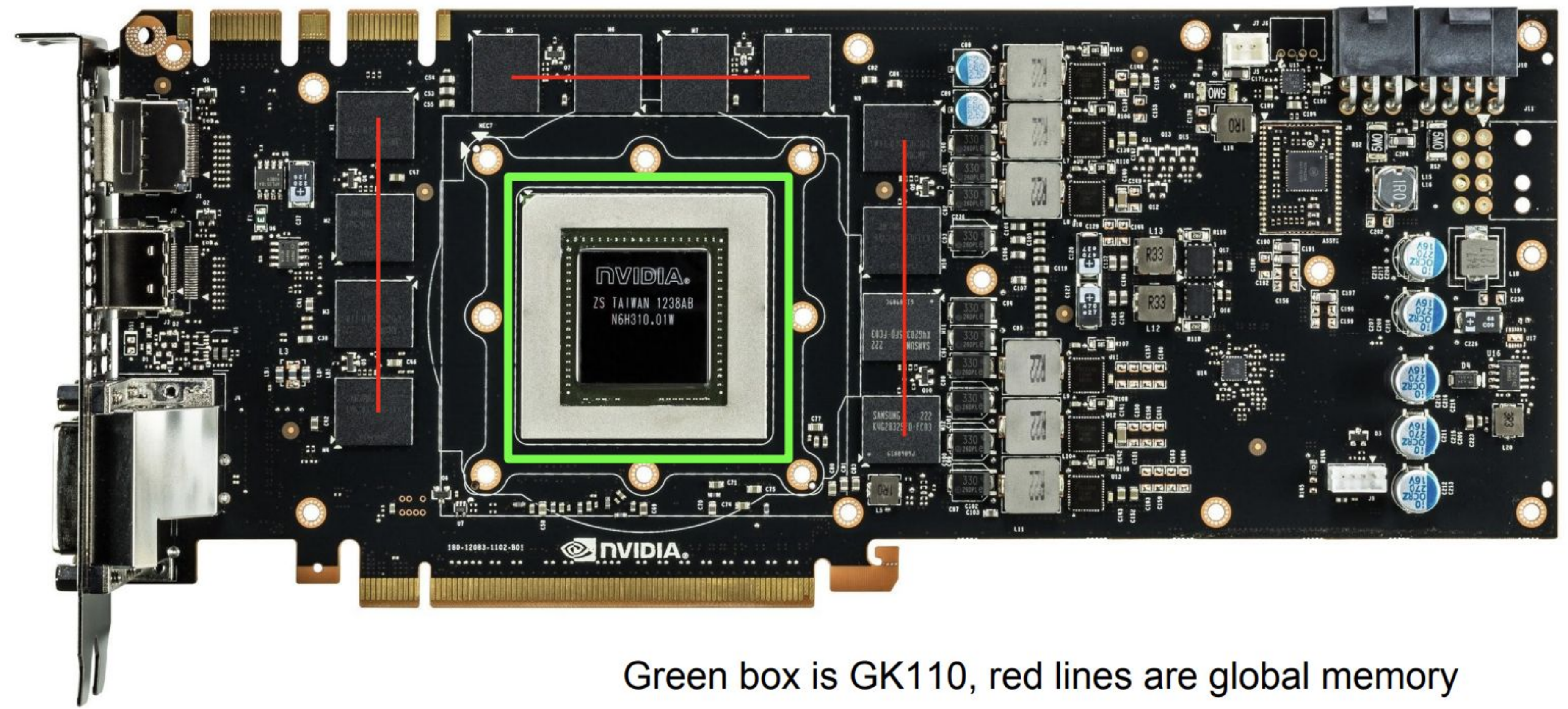
- Thread is a sequence of instructions to be executed in SIMD manner
- Warp is a collection of threads which
  - Belong to the same block
  - Execute in SIMD fashion
- Block is a collection of threads which
  - Reside on the same processor core
  - Can share memory and synchronize
- All blocks in a grid have the same number of threads
- All threads in a grid share the same kernel function

GPU memory architecture



http://courses.engr.illinois.edu/ece498

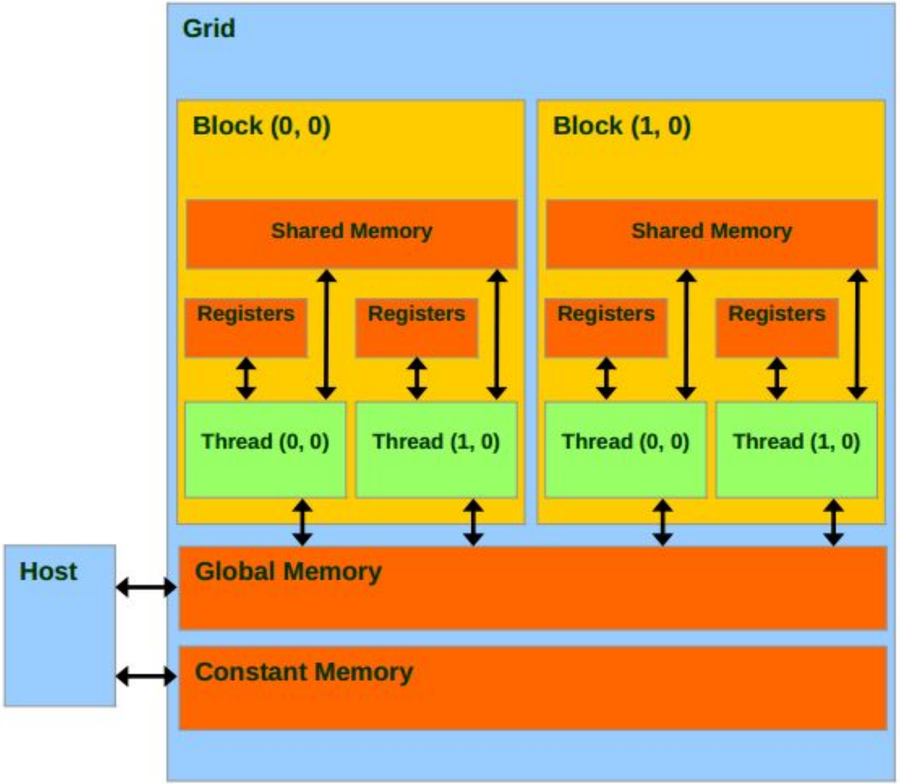
Global memory is huge



Green box is GK110, red lines are global memory

CUDA memory operations

Device	Thread	R/W registers
	Thread	R/W local memory
	Block	R/W shared memory
	Grid	R/W global memory
	Grid	Read only constant memory
Host	Grid	R/W global memory
	Grid	R/W constant memory



http://courses.engr.illinois.edu/ece498

Memory allocation and freeing

```
float *deviceData = NULL;
int size = width * height * sizeof(float);

// Allocate global memory on device
cudaMalloc((void**)&deviceData, size);

// Do work

// Free memory
cudaFree(deviceData);
```

CUDA memory copy operations

```
// host to device
cudaMemcpy(devicePtr, hostPtr, size, cudaMemcpyHostToDevice);

//device to device
cudaMemcpy(destPtr, sourcePtr, size, cudaMemcpyDeviceToDevice);

//device to host
cudaMemcpy(hostPtr, devicePtr, size, cudaMemcpyDeviceToHost);
```

Compiling programs

```
__global__ void mykernel(void) {
}

int main(void) {
    mykernel<<<1,1>>>();
    printf("Hello World!\n");
    return 0;
}
```

hello_world.cu

```
$ nvcc hello_world.cu
$ ./a.out
Hello World!
$
```

CUDA program for vector addition

```
int main() {

    // 1. Allocate and initialize vectors A, B, and result vector C on host
    // Allocate memory and transfer A, B, C to device

    // 2. Perform C = A + B on device

    // 3. Copy C from device to host
    // Free up memory on host and device

    return 0;
}
```

CUDA program for vector addition

```
// Device code
__global__ void VecAdd(float* A, float* B, float*
C, int N){
    int i = blockDim.x * blockIdx.x + threadIdx.x;
    if (i < N)
        C[i] = A[i] + B[i];
}

// Host code
int main() {
    int N = ...;
    size_t size = N * sizeof(float);

    // Allocate input vectors h_A and h_B in host
memory
    float* h_A = (float*)malloc(size);
    float* h_B = (float*)malloc(size);

    // Initialize input vectors
    ...

    // Allocate vectors in device memory
    float* d_A;   cudaMalloc(&d_A, size);
    float* d_B;   cudaMalloc(&d_B, size);
    float* d_C;   cudaMalloc(&d_C, size);
```

```
    // Copy vectors from host memory to device memory
    cudaMemcpy(d_A, h_A, size,cudaMemcpyHostToDevice);
    cudaMemcpy(d_B, h_B, size,cudaMemcpyHostToDevice);

    // Invoke kernel
    int threadsPerBlock = 256;
    int blocksPerGrid = (N + threadsPerBlock - 1) /
threadsPerBlock;
    VecAdd<<<blocksPerGrid, threadsPerBlock>>>(d_A,
d_B, d_C, N);

    // Copy result from device memory to host memory
    // h_C contains the result in host memory
    cudaMemcpy(h_C, d_C, size,cudaMemcpyDeviceToHost);

    // Free device memory
    cudaFree(d_A);
    cudaFree(d_B);
    cudaFree(d_C);

    // Free host memory
    ...
}
```

CUDA program for vector addition

```
// Device code
__global__ void VecAdd(float* A, float* B, float*
C, int N){
    int i = blockDim.x * blockIdx.x + threadIdx.x;
    if (i < N)
        C[i] = A[i] + B[i];
}

// Host code
int main() {
    int N = ...;
    size_t size = N * sizeof(float);

    // Allocate input vectors h_A and h_B in host
memory
    float* h_A = (float*)malloc(size);
    float* h_B = (float*)malloc(size);

    // Initialize input vectors
    ...

    // Allocate vectors in device memory
    float* d_A;   cudaMalloc(&d_A, size);
    float* d_B;   cudaMalloc(&d_B, size);
    float* d_C;   cudaMalloc(&d_C, size);

    // Copy vectors from host memory to device memory
    cudaMemcpy(d_A, h_A, size,cudaMemcpyHostToDevice);
    cudaMemcpy(d_B, h_B, size,cudaMemcpyHostToDevice);

    // Invoke kernel
    int threadsPerBlock = 256;
    int blocksPerGrid = (N + threadsPerBlock - 1) /
threadsPerBlock;
    VecAdd<<<blocksPerGrid, threadsPerBlock>>>(d_A,
d_B, d_C, N);

    // Copy result from device memory to host memory
    // h_C contains the result in host memory
    cudaMemcpy(h_C, d_C, size,cudaMemcpyDeviceToHost);

    // Free device memory
    cudaFree(d_A);
    cudaFree(d_B);
    cudaFree(d_C);

    // Free host memory
    ...
}
```

CUDA program for vector addition

```
// Device code
__global__ void VecAdd(float* A, float* B, float*
C, int N){
    int i = blockDim.x * blockIdx.x + threadIdx.x;
    if (i < N)
        C[i] = A[i] + B[i];
}

// Host code
int main() {
    int N = ...;
    size_t size = N * sizeof(float);

    // Allocate input vectors h_A and h_B in host
memory
    float* h_A = (float*)malloc(size);
    float* h_B = (float*)malloc(size);

    // Initialize input vectors
    ...

    // Allocate vectors in device memory
    float* d_A;   cudaMalloc(&d_A, size);
    float* d_B;   cudaMalloc(&d_B, size);
    float* d_C;   cudaMalloc(&d_C, size);
```

```
    // Copy vectors from host memory to device memory
    cudaMemcpy(d_A, h_A, size,cudaMemcpyHostToDevice);
    cudaMemcpy(d_B, h_B, size,cudaMemcpyHostToDevice);

    // Invoke kernel
    int threadsPerBlock = 256;
    int blocksPerGrid = (N + threadsPerBlock - 1) /
threadsPerBlock;
    VecAdd<<<blocksPerGrid, threadsPerBlock>>>(d_A,
d_B, d_C, N);

    // Copy result from device memory to host memory
    // h_C contains the result in host memory
    cudaMemcpy(h_C, d_C, size,cudaMemcpyDeviceToHost);

    // Free device memory
    cudaFree(d_A);
    cudaFree(d_B);
    cudaFree(d_C);

    // Free host memory
    ...
}
```

CUDA program for vector addition

```
// Device code
__global__ void VecAdd(float* A, float* B, float*
C, int N){
    int i = blockDim.x * blockIdx.x + threadIdx.x;
    if (i < N)
        C[i] = A[i] + B[i];
}

// Host code
int main() {
    int N = ...;
    size_t size = N * sizeof(float);

    // Allocate input vectors h_A and h_B in host
memory
    float* h_A = (float*)malloc(size);
    float* h_B = (float*)malloc(size);

    // Initialize input vectors
    ...

    // Allocate vectors in device memory
    float* d_A;   cudaMalloc(&d_A, size);
    float* d_B;   cudaMalloc(&d_B, size);
    float* d_C;   cudaMalloc(&d_C, size);
```

```
    // Copy vectors from host memory to device memory
    cudaMemcpy(d_A, h_A, size,cudaMemcpyHostToDevice);
    cudaMemcpy(d_B, h_B, size,cudaMemcpyHostToDevice);

    // Invoke kernel
    int threadsPerBlock = 256;
    int blocksPerGrid = (N + threadsPerBlock - 1) /
threadsPerBlock;
    VecAdd<<<blocksPerGrid, threadsPerBlock>>>(d_A,
d_B, d_C, N);

    // Copy result from device memory to host memory
    // h_C contains the result in host memory
    cudaMemcpy(h_C, d_C, size,cudaMemcpyDeviceToHost);

    // Free device memory
    cudaFree(d_A);
    cudaFree(d_B);
    cudaFree(d_C);

    // Free host memory
    ...
}
```

CUDA program for vector addition

```
// Device code
__global__ void VecAdd(float* A, float* B, float*
C, int N){
    int i = blockDim.x * blockIdx.x + threadIdx.x;
    if (i < N)
        C[i] = A[i] + B[i];
}

// Host code
int main() {
    int N = ...;
    size_t size = N * sizeof(float);

    // Allocate input vectors h_A and h_B in host
memory
    float* h_A = (float*)malloc(size);
    float* h_B = (float*)malloc(size);

    // Initialize input vectors
    ...

    // Allocate vectors in device memory
    float* d_A;   cudaMalloc(&d_A, size);
    float* d_B;   cudaMalloc(&d_B, size);
    float* d_C;   cudaMalloc(&d_C, size);
```

```
    // Copy vectors from host memory to device memory
    cudaMemcpy(d_A, h_A, size,cudaMemcpyHostToDevice);
    cudaMemcpy(d_B, h_B, size,cudaMemcpyHostToDevice);

    // Invoke kernel
    int threadsPerBlock = 256;
    int blocksPerGrid = (N + threadsPerBlock - 1) /
threadsPerBlock;
    VecAdd<<<blocksPerGrid, threadsPerBlock>>>(d_A,
d_B, d_C, N);

    // Copy result from device memory to host memory
    // h_C contains the result in host memory
    cudaMemcpy(h_C, d_C, size,cudaMemcpyDeviceToHost);

    // Free device memory
    cudaFree(d_A);
    cudaFree(d_B);
    cudaFree(d_C);

    // Free host memory
    ...
}
```

CUDA program for vector addition

```
// Device code
__global__ void VecAdd(float* A, float* B, float*
C, int N){
    int i = blockDim.x * blockIdx.x + threadIdx.x;
    if (i < N)
        C[i] = A[i] + B[i];
}

// Host code
int main() {
    int N = ...;
    size_t size = N * sizeof(float);

    // Allocate input vectors h_A and h_B in host
memory
    float* h_A = (float*)malloc(size);
    float* h_B = (float*)malloc(size);

    // Initialize input vectors
    ...

    // Allocate vectors in device memory
    float* d_A;   cudaMalloc(&d_A, size);
    float* d_B;   cudaMalloc(&d_B, size);
    float* d_C;   cudaMalloc(&d_C, size);
```

```
    // Copy vectors from host memory to device memory
    cudaMemcpy(d_A, h_A, size,cudaMemcpyHostToDevice);
    cudaMemcpy(d_B, h_B, size,cudaMemcpyHostToDevice);

    // Invoke kernel
    int threadsPerBlock = 256;
    int blocksPerGrid = (N + threadsPerBlock - 1) /
threadsPerBlock;
    VecAdd<<<blocksPerGrid, threadsPerBlock>>>(d_A,
d_B, d_C, N);

    // Copy result from device memory to host memory
    // h_C contains the result in host memory
    cudaMemcpy(h_C, d_C, size,cudaMemcpyDeviceToHost);

    // Free device memory
    cudaFree(d_A);
    cudaFree(d_B);
    cudaFree(d_C);

    // Free host memory
    ...
}
```

CUDA program for vector addition

```
// Device code
__global__ void VecAdd(float* A, float* B, float*
C, int N){
    int i = blockDim.x * blockIdx.x + threadIdx.x;
    if (i < N)
        C[i] = A[i] + B[i];
}

// Host code
int main() {
    int N = ...;
    size_t size = N * sizeof(float);

    // Allocate input vectors h_A and h_B in host
memory
    float* h_A = (float*)malloc(size);
    float* h_B = (float*)malloc(size);

    // Initialize input vectors
    ...

    // Allocate vectors in device memory
    float* d_A;   cudaMalloc(&d_A, size);
    float* d_B;   cudaMalloc(&d_B, size);
    float* d_C;   cudaMalloc(&d_C, size);
```

```
    // Copy vectors from host memory to device memory
    cudaMemcpy(d_A, h_A, size,cudaMemcpyHostToDevice);
    cudaMemcpy(d_B, h_B, size,cudaMemcpyHostToDevice);

    // Invoke kernel
    int threadsPerBlock = 256;
    int blocksPerGrid = (N + threadsPerBlock - 1) /
threadsPerBlock;
    VecAdd<<<blocksPerGrid, threadsPerBlock>>>(d_A,
d_B, d_C, N);

    // Copy result from device memory to host memory
    // h_C contains the result in host memory
    cudaMemcpy(h_C, d_C, size,cudaMemcpyDeviceToHost);

    // Free device memory
    cudaFree(d_A);
    cudaFree(d_B);
    cudaFree(d_C);

    // Free host memory
    ...
}
```

CUDA program for vector addition

```
// Device code
__global__ void VecAdd(float* A, float* B, float*
C, int N){
    int i = blockDim.x * blockIdx.x + threadIdx.x;
    if (i < N)
        C[i] = A[i] + B[i];
}

// Host code
int main() {
    int N = ...;
    size_t size = N * sizeof(float);

    // Allocate input vectors h_A and h_B in host
memory
    float* h_A = (float*)malloc(size);
    float* h_B = (float*)malloc(size);

    // Initialize input vectors
    ...

    // Allocate vectors in device memory
    float* d_A;   cudaMalloc(&d_A, size);
    float* d_B;   cudaMalloc(&d_B, size);
    float* d_C;   cudaMalloc(&d_C, size);
```

```
    // Copy vectors from host memory to device memory
    cudaMemcpy(d_A, h_A, size,cudaMemcpyHostToDevice);
    cudaMemcpy(d_B, h_B, size,cudaMemcpyHostToDevice);

    // Invoke kernel
    int threadsPerBlock = 256;
    int blocksPerGrid = (N + threadsPerBlock - 1) /
threadsPerBlock;
    VecAdd<<<blocksPerGrid, threadsPerBlock>>>(d_A,
d_B, d_C, N);

    // Copy result from device memory to host memory
    // h_C contains the result in host memory
    cudaMemcpy(h_C, d_C, size,cudaMemcpyDeviceToHost);

    // Free device memory
    cudaFree(d_A);
    cudaFree(d_B);
    cudaFree(d_C);

    // Free host memory
    ...
}
```

CUDA program for vector addition

```
// Device code
__global__ void VecAdd(float* A, float* B, float*
C, int N){
    int i = blockDim.x * blockIdx.x + threadIdx.x;
    if (i < N)
        C[i] = A[i] + B[i];
}

// Host code
int main() {
    int N = ...;
    size_t size = N * sizeof(float);

    // Allocate input vectors h_A and h_B in host
memory
    float* h_A = (float*)malloc(size);
    float* h_B = (float*)malloc(size);

    // Initialize input vectors
    ...

    // Allocate vectors in device memory
    float* d_A;   cudaMalloc(&d_A, size);
    float* d_B;   cudaMalloc(&d_B, size);
    float* d_C;   cudaMalloc(&d_C, size);
```

```
    // Copy vectors from host memory to device memory
    cudaMemcpy(d_A, h_A, size,cudaMemcpyHostToDevice);
    cudaMemcpy(d_B, h_B, size,cudaMemcpyHostToDevice);

    // Invoke kernel
    int threadsPerBlock = 256;
    int blocksPerGrid = (N + threadsPerBlock - 1) /
threadsPerBlock;
    VecAdd<<<blocksPerGrid, threadsPerBlock>>>(d_A,
d_B, d_C, N);

    // Copy result from device memory to host memory
    // h_C contains the result in host memory
    cudaMemcpy(h_C, d_C, size,cudaMemcpyDeviceToHost);

    // Free device memory
    cudaFree(d_A);
    cudaFree(d_B);
    cudaFree(d_C);

    // Free host memory
    ...
}
```

Sequence of actions

- Allocate memory on host, initialize data
- Allocate memory on device `cudaMalloc`
- Copy memory from host to device

  `cudaMemcpy(,,cudaMemcpyHostToDevice)`
- Invoke kernel to process on data on device

  `Kernel<<< >>>(args)`
- Copy output data from device to host

  `cudaMemcpy(,,cudaMemcpyDeviceToHost)`

Error handling

```
cudaError_t err = cudaMalloc((void **) &d_A, size);
if (err != cudaSuccess) {
    printf("%s in %s at line %d\n", cudaGetErrorString(err),__FILE__,__LINE__);
    exit(EXIT_FAILURE);
}
```

```
#define myCudaCheck() {
    cudaError_t err = cudaGetLastError();
    if(err != cudaSuccess) {
        printf("%s in %s at line %d\n",
                  cudaGetErrorString(err),__FILE__,__LINE__);
        exit(EXIT_FAILURE);
    }
}
```

Function types

	Executed on the:	Only callable from the:
`__global__ void KernelFunc()`	device	host
`__device__ float DeviceFunc()`	device	device
`__host__ float HostFunc()`	host	host

- `__global__` must return void
- `__device__` is inlined by default
- `__host__` and `__device__` can be used together

Blocks vs Threads

Only Threads

Blocks vs Threads

Only Threads

```
// Invoke kernel
VecAdd<<<1, N>>>(d_A, d_B, d_C, N);
```

```
// Device code
__global__ void VecAdd(float* A, float* B, float* C, int N){
    int i = blockDim.x * blockIdx.x + threadIdx.x;
    if (i < N)
        C[i] = A[i] + B[i];
}
```

Blocks vs Threads

Only Blocks

```
// Invoke kernel
```

```
// Device code
```

Blocks vs Threads

Only Blocks

```
// Invoke kernel
VecAdd<<<N, 1>>>(d_A, d_B, d_C, N);
```

```
// Device code
__global__ void VecAdd(float* A, float* B, float* C, int N){
    int i = blockIdx.x;
    if (i < N)
        C[i] = A[i] + B[i];
}
```

Blocks vs Threads

Blocks + Threads

Blocks vs Threads

Blocks + Threads

```
// Invoke kernel
VecAdd<<<(N+256-1)/256, 256>>>(d_A, d_B, d_C, N);
```

```
// Device code
__global__ void VecAdd(float* A, float* B, float* C, int N){
    int i = blockDim.x * blockIdx.x + threadIdx.x;
    if (i < N)
        C[i] = A[i] + B[i];
}
```

Blocks vs Threads

Blocks + Threads (Variable threads per block)

Blocks vs Threads

Blocks + Threads (Variable threads per block)

```
// Invoke kernel
int threadsPerBlock = 256;
int blocksPerGrid = (N + threadsPerBlock - 1) / threadsPerBlock;
VecAdd<<<blocksPerGrid, threadsPerBlock>>>(d_A, d_B, d_C, N);
```

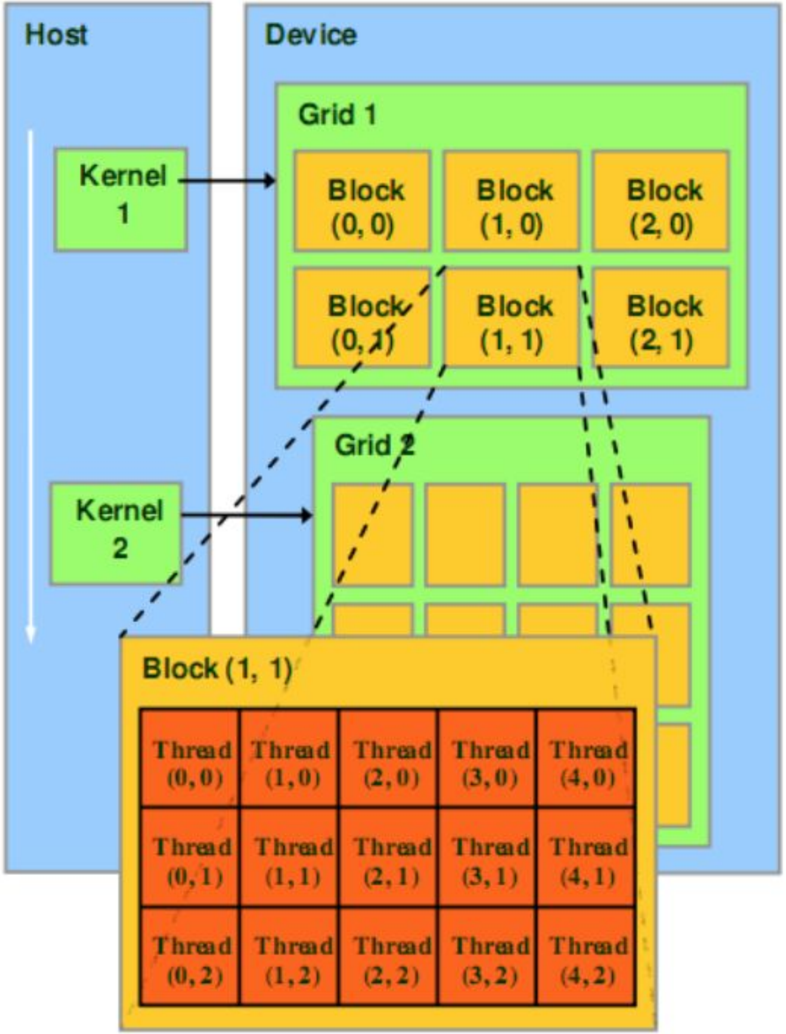
```
// Device code
__global__ void VecAdd(float* A, float* B, float* C, int N){
    int i = blockDim.x * blockIdx.x + threadIdx.x;
    if (i < N)
        C[i] = A[i] + B[i];
}
```

Recap on thread hierarchies

Kernel function corresponds to a grid which is a collection of blocks and each block is a collection of threads

Why these multiple hierarchies?

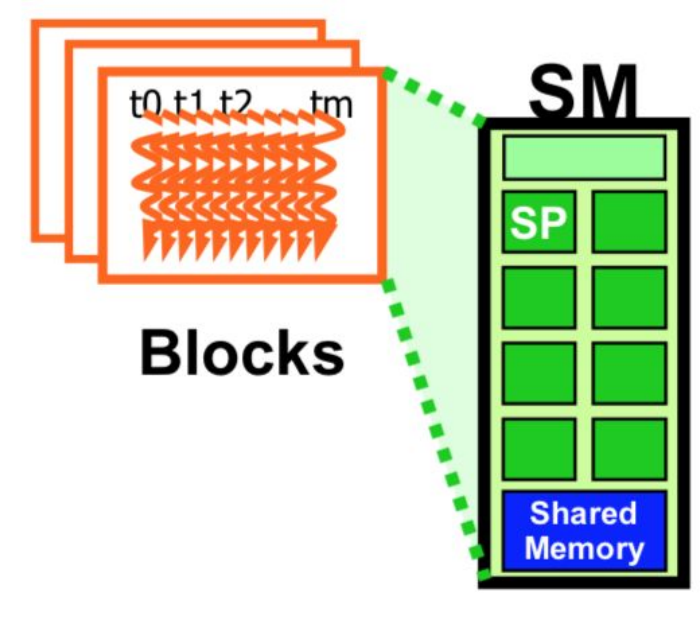
To map to typical high-dimensional data structures and multiple loops in algorithms



Allocation of threads to SMs

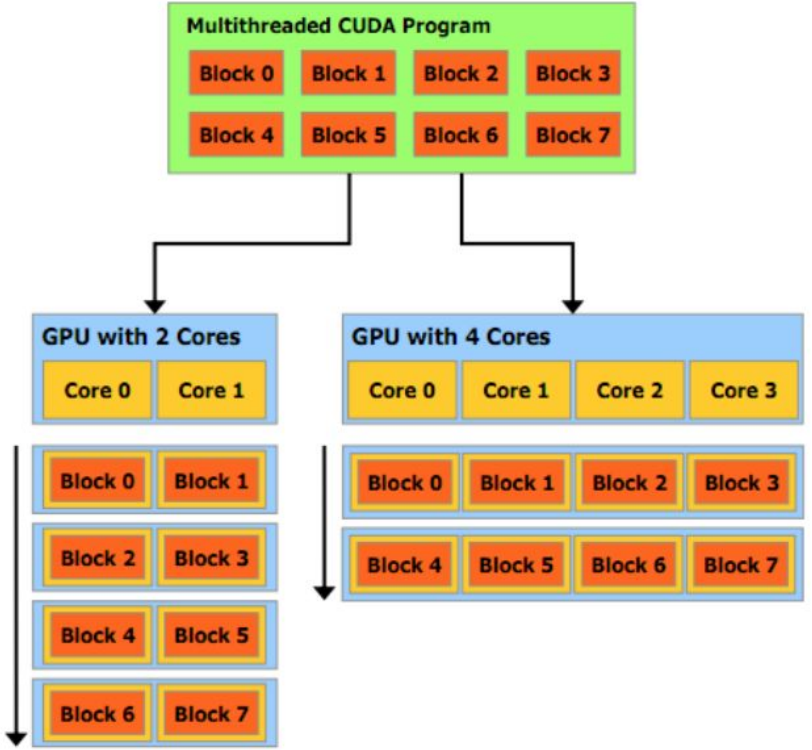
- Threads of a block execute on the same streaming multiprocessor (SM)
- Multiple blocks could be executed on the same SM
- Up to 8 blocks to each SM as resource allows
  - Fermi SM can take up to 1536 threads
  - Could be 256 (threads/block) * 6 blocks
  - Or 512 (threads/block) * 3 blocks, etc.

Map earlier cores to SMs
Map earlier ALUs to SP



Transparent or automatic scalability

- SM maintains thread/block idx #s
- SM manages/schedules thread execution
- CUDA abstracts the allocation information from the programmer to enable transparent or automatic scalability



A multithreaded program is partitioned into blocks of threads that execute independently from each other, so that a GPU with more cores will automatically execute the program in less time than a GPU with fewer cores.

Figure 1-4. Automatic Scalability

Recollect: Warp scheduling

- All threads in a warp execute the same instruction (recollect control divergence)
- To hide latency in scheduling warps, need almost zero overhead context switching between warps in SM
- Dynamic scheduling of warps: Warps whose next instruction has its operands ready for consumption are eligible for execution
- Eligible Warps are selected for execution based on a prioritized scheduling policy not exposed to the CUDA programmer

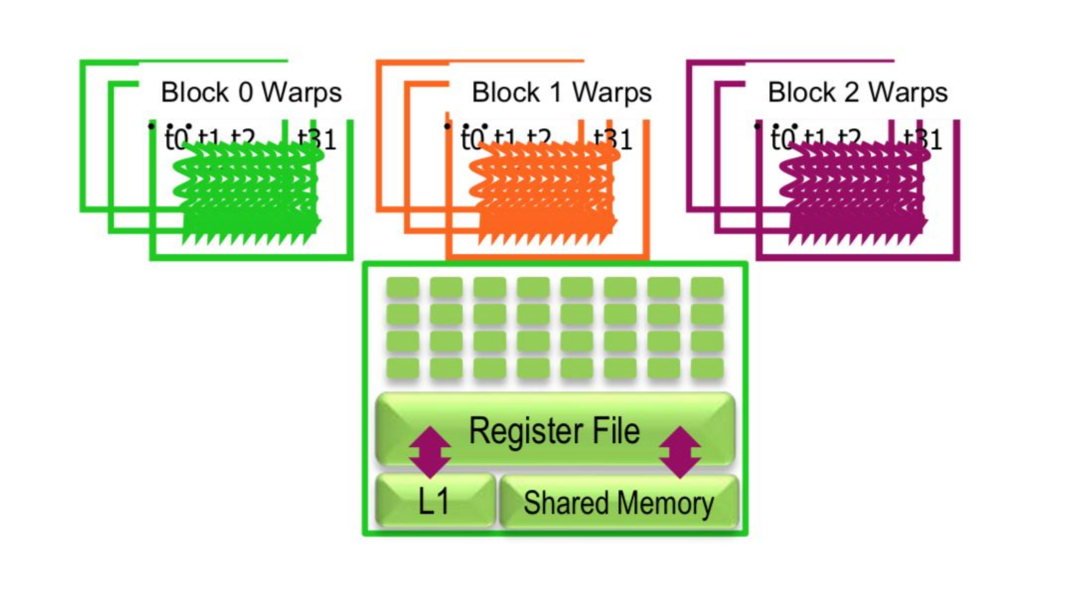
Warps in Hardware

- Each block is executed as **warps** of 32 threads
- This (32) is an implementation choice in the current GPU architectures, and is not available for configuration to a CUDA programmer
- Future GPUs may have different number of threads in each warp

Warps example

Consider 3 blocks assigned to an SM, with each block having 256 threads

How many warps are there in an SM?



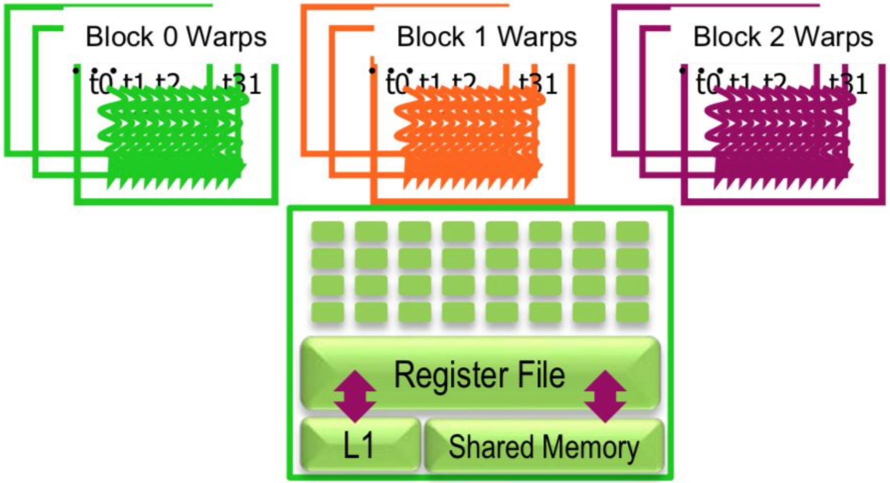
Warps example

Consider 3 blocks assigned to an SM, with each block having 256 threads

How many warps are there in an SM?

Each block has 256 threads = 256/32 = 8 warps

Each SM has 3 blocks = 3 * 8 = 24 warps



How to choose block and thread granularity

- Consider a large matrix-matrix addition operation. How to choose block size? Recollect `VecAdd<<<blocksPerGrid, threadsPerBlock>>>(d_A, d_B, d_C, N);`
- For Fermi architecture: 1536 threads per SM (48 warps of 32 threads each) and up to 8 blocks per SM
- 8x8 blocks ⇒
- 16x16 blocks ⇒
- 32x32 blocks ⇒

How to choose block and thread granularity

- Consider a large matrix-matrix addition operation. How to choose block size? Recollect `VecAdd<<<blocksPerGrid, threadsPerBlock>>>(d_A, d_B, d_C, N);`
- For Fermi architecture: 1536 threads per SM (48 warps of 32 threads each) and up to 8 blocks per SM
- 8x8 blocks ⇒ ≤64 threads per block ⇒ ≤512 threads per SM ⇒ Under-utilzation
- 16x16 blocks ⇒
- 32x32 blocks ⇒

How to choose block and thread granularity

- Consider a large matrix-matrix addition operation. How to choose block size? Recollect `VecAdd<<<blocksPerGrid, threadsPerBlock>>>(d_A, d_B, d_C, N);`
- For Fermi architecture: 1536 threads per SM (48 warps of 32 threads each) and up to 8 blocks per SM
- 8x8 blocks ⇒ ≤64 threads per block ⇒ ≤512 threads per SM ⇒ Under-utilzation
- 16x16 blocks ⇒ ≤256 threads per block ⇒ ≤6 blocks, ≤1536 threads per SM ⇒ Full utilization
- 32x32 blocks ⇒

How to choose block and thread granularity

- Consider a large matrix-matrix addition operation. How to choose block size? Recollect `VecAdd<<<blocksPerGrid, threadsPerBlock>>>(d_A, d_B, d_C, N);`
- For Fermi architecture: 1536 threads per SM (48 warps of 32 threads each) and up to 8 blocks per SM
- 8x8 blocks ⇒ ≤64 threads per block ⇒ ≤512 threads per SM ⇒ Under-utilzation
- 16x16 blocks ⇒ ≤256 threads per block ⇒ ≤6 blocks, ≤1536 threads per SM ⇒ Full utilization
- 32x32 blocks ⇒ ≤1024 threads per block ⇒ 1 block per SM ⇒ Under-utilization

Next time

- Memory and related optimization of CUDA programs