Threads and Blocks

Bedřich Beneš, Ph.D.
Purdue University
Department of Computer Graphics

Kernel

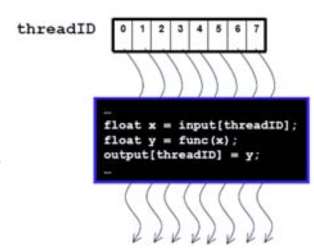
- *kernel* is a C function
- it is executed *n* times on *n* different CUDA *threads*
- defined using `__global__` keyword
- invoked by a new `<<< >>>` syntax
- each thread has unique *threadID* accessible via `threadIdx` and `blockIdx`

Threads

- kernel is executed as a grid of thread blocks
- The ID is used to index data and make decisions
- How many threads? thousands for a good performance
- 4x4 matrix multiplication is not good…

Threads

- Example:
  - eight threads
  - each runs the same code
  - each works independently
  - the `threadID` is used to access memory

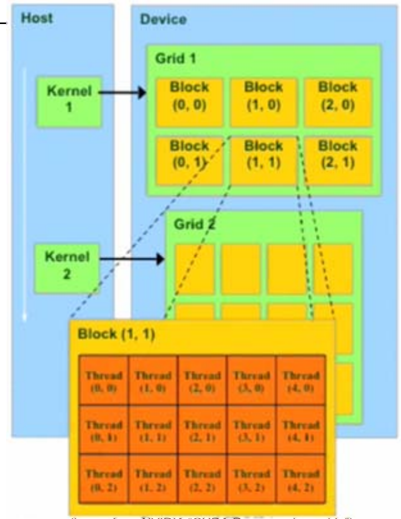


(image from NVIDIA "CUDA Programming guide")

  - threads within one block can communicate

Thread Cooperation

- kernel launches a **grid** of **thread blocks**
- threads within a single *thread block* cooperate
- memory access (SM) for sharing results
- synchronization of execution within block



(image from NVIDIA "CUDA Programming guide")

Grid of Thread Blocks

- Each thread has local private memory
- Thread block has a shared memory visible to all threads within the block very fast and very efficient
- sharing data across blocks is difficult
- threads from different blocks cannot cooperate
- All threads can see global memory
- grid is up to 2D block is up to 3D

Grid of Thread Blocks

- hw can schedule thread blocks anywhere
- thread block executes independently
- can be executed in any order (parallel, serial)
- the actual # is given by the application and the # of processor.
  Can be much higher than the # of processors, will be scheduled by CUDA

Execution Model

- Thread runs on a thread processor
- Thread Block runs on a Multiprocessor
- Grid runs on a Device

Vector Addition

- Let's sum two *n*-dimensional vectors

$$a = (a_0, a_1, \dots, a_{n-1})$$
$$b = (b_0, b_1, \dots, b_{n-1})$$
$$c = a + b = (a_0 + b_0, a_1 + b_1, \dots, a_{n-1} + b_{n-1})$$

Single CPU version

```
void VecAdd(const float* a, const float* b,
                   float* c, unsigned int n)
{
  for (unsigned int i=0;i<n;i++)
      c[i] = a[i] + b[i];
}
```


Single CPU version

Preparation:

- take two arrays a and b
- prepare array c
- call VecAdd(a,b,c,n)

GPU version

```
// Device code - kernel
//this runs on the device
__global__ void VecAdd(const float* a, const
  float* b, float* c, unsigned int N)
{
  int i = blockIdx.x;
  if (i < n) c[i] = a[i] + b[i];
}
```

GPU version

- Preparation
  - take two arrays a and b
  - prepare array c
  - **copy a and b into GPU**
  - **prepare array c on the GPU**
  - **call CUDA**
  - **copy result back**

GPU version

```
error = cudaMalloc((void**)&d_A, size);
if (error != cudaSuccess) exit(-1);
error = cudaMalloc((void**)&d_B, size);
if (error != cudaSuccess) exit(-1);
error = cudaMalloc((void**)&d_C, size);
if (error != cudaSuccess) exit(-1);
```


GPU version

```
//Copy from host memory to device memory
error = cudaMemcpy(d_A, h_A, size,
                   cudaMemcpyHostToDevice);
if (error != cudaSuccess) exit(-1);
error = cudaMemcpy(d_B, h_B, size,
                   cudaMemcpyHostToDevice);
if (error != cudaSuccess) exit(-1);

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

number of blocks

number of threads per block

GPU version

```
//getting the data back
error = cudaMemcpy(h_C, d_C, size,
                   cudaMemcpyDeviceToHost);
```

GPU version

```
//n blocks with 1 thread:
VecAdd<<<n,1>>>(d_A, d_B, d_C, n);

//works like this:
__global__ void VecAdd(const float* a, const
   float* b, float* c, unsigned int n)
{
  int i = blockIdx.x;
  if (i < n) c[i] = a[i] + b[i];
}
```


GPU version

```
//We can use 1 block with n threads
VecAdd<<<1,n>>>(d_A, d_B, d_C, n);

//and we have to rewrite:
__global__ void VecAdd(const float* a, const
   float* b, float* c, unsigned int n)
{
  int i = threadIdx.x;
  if (i < n) c[i] = a[i] + b[i];
}
```


Blocks and threads

- Current GPUs (Fermi)
- Max # of blocks 65,536
- Max # of threads 512

GPU version

```
//or
int threads=256;
int blocks=(n+threads-1)/threads;
VecAdd<<<blocks,threads>>>(d_A, d_B, d_C, n);

__global__ void VecAdd(const float* a, const
   float* b, float* c, unsigned int n)
{
  int i = threadIdx.x+blockIdx.x*blockDim.x;
  if (i < n) c[i] = a[i] + b[i];
}
```

GPU version

- Anyway, the limit will be 65536*512
- How can we sum a *very* long vector?

GPU version

```
VecAdd<<<256,256>>>(d_A, d_B, d_C, n);

__global__ void VecAdd(const float* a, const
  float* b, float* c, unsigned int n)
{ int stride=blockDim.x*gridDim.x;
  int i= threadIdx.x+blockIdx.x*blockDim.x;
  while (i<n){
    c[i] = a[i] + b[i];
    i+=stride;
  }
}
```


2D grid of blocks

- blocks can be 3D and threads can be 2D

```
dim3 blocks(MAX/16, MAX/16);
dim3 threads(16,16);
kernel<<<blocks,threads>>>()
kernel()
{ int x=threadIdx.x+blockIdx.x*blockDim.x;
  int y=threadIdx.y+blockIdx.y*blockDim.y;
...
}
```


Thread Hierarchy

- `threadIdx` is a 3D vector
- `blockIdx` is a 2D vector

```
__global__ void MatAdd(float
  A[N][N], float B[N][N], float
  C[N][N]){
  int i = threadIdx.x;
  int j = threadIdx.y;
  C[i][j] = A[i][j] + B[i][j];
}
```

Thread Hierarchy

- array mapping function (AMF) tells how to get the index
- 1D: the index itself
- 2D of dimensions (Dx,Dy) (x,y)~(x+y*Dx)
- 3D of dimensions (Dx,Dy,Dz) (x,y,z)~(x+y*Dx+z*Dx*Dy)= =(x+Dx*(y+z*Dy))

Grid of Thread Blocks

- specified by the parameters in <<<a,b>>>

```
int main(){
 dim3 dimBlock(16,16);
 dim3 dimGrid((N+dimBlock.x-1)/dimBlock.x,
              (N+dimBlock.y-1)/dimBlock.y);
 MatAdd<<<dimGrid, dimBlock>>>(A,B,C,N);
}
__global__ void MatAdd(float a[N][N], float b[N][N],
   float c[N][N],int N){
   int i=blockIdx.x*blockDim.x+threadIdx.x;
   int j=blockIdx.y*blockDim.y+threadIdx.y;
   if (i<N && j<N) c[i][j]=a[i][j]+b[i][j];
}
```


Reading

- NVIDIA, *CUDA Programming Guide*
- Sanders, J., Kandrot, E., *CUDA by Example,* Addison-Wesley