MIC-GPU: High-Performance Computing for Medical Imaging on Programmable Graphics Hardware (GPUs)

CUDA API

Klaus Mueller, Ziyi Zheng, Eric Papenhausen

Stony Brook University

Computer Science

Stony Brook, NY

Function Qualifiers

Device Global, & Host

- To specify whether a function executes on the host or on the device
- __global__ must return void

Function	Exe on	Call from
__device__	GPU	GPU
__global__	GPU	CPU
__host__	CPU	CPU

```
__global__ void Func (float* parameter);
```


Variable Qualifiers

Shared, Device & Constant

- To specify the memory location on the device of a variable
- __shared__ and __constant__ are optionally used together with __device__

Variable	Memory	Scope	Lifetime
__shared__	Shared	Block	Kernel
__device__	Global	Grid	Application
__constant__	Constant	Grid	Application

```
__constant__    float ConstantArrary[16];
__shared__      float SharedArray[16];
__device__      ……
```


Execution Configuration

<<< Grids, Blocks>>>

- Kernel function must specify the number of threads for each call (dim3)

<<< Grids, Blocks, Shared>>>

- It can specify the number of bytes in shared memory that is dynamically allocated per block (size_t)

```
dim3 dimBlock(8, 8, 2);
dim3 dimGrid(10, 10, 1);
KernelFunc<<<dimGrid, dimBlock>>>(…);

KernelFunc<<< 100, 128 >>>(...);
```

Device Side Parameters

Threads get parameters from execution configuration

- Dimensions of the grid in blocks

```
dim3 gridDim;
```

- Dimensions of the block in threads

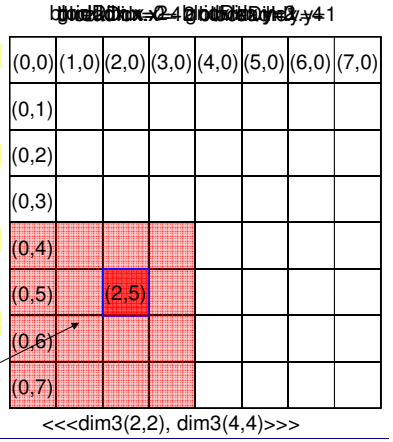
```
dim3 blockDim;
```

- Block index within the grid

```
dim3 blockIdx;
```

- Thread index within the block

```
dim3 threadIdx;
```




```
blockIdx.x *blockDim.x + threadIdx.x;
blockIdx.y *blockDim.y + threadIdx.y;
```


Anatomy of a Kernel Function Call

Define function as device kernel to be called from the host:

```
__global__ void KernelFunc(...);
```

Configuring thread layout and memory:

```
dim3   DimGrid(100,50);  // 5000 thread blocks
dim3   DimBlock(4,8,8);  // 256 threads per (3D) block
size_t SharedMemBytes = 64; // 64 bytes of shared memory
```

Launch the kernel (<<<, >>> are CUDA runtime directives)

```
KernelFunc<<< DimGrid, DimBlock, SharedMemBytes >>>(...);
```

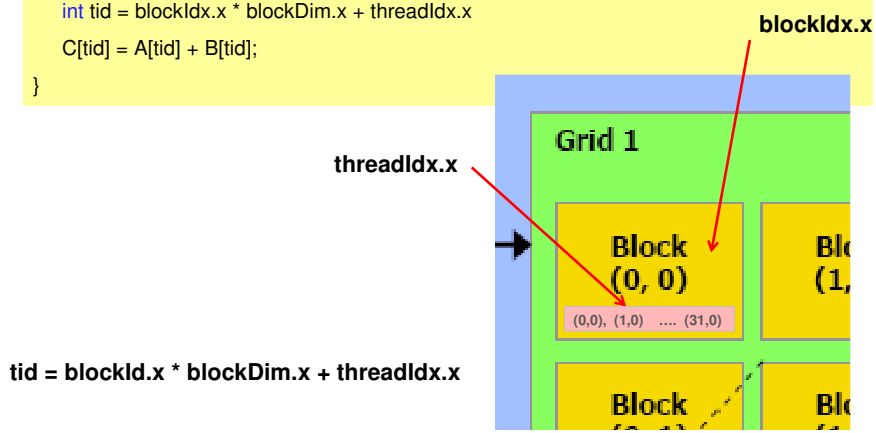
Example: Vector Add (CPU)

```
void vectorAdd(float *A, float *B, float *C, int N) {
    for(int i = 0; i < N; i++)
        C[i] = A[i] + B[i]; }

int main() {
    int N = 4096;
        // allocate and initialize memory
    float *A = (float *) malloc(sizeof(float)*N);
    float *B = (float *) malloc(sizeof(float)*N);
    float *C = (float *) malloc(sizeof(float)*N);
    init(A); init(B);
    vectorAdd(A, B, C, N);        // run function
    free(A); free(B); free(C);    // free memory

}
```

from: Dana Schaa and Byunghyun Jang, NEU

Example: Vector Add (GPU)

```
__global__ void gpuVecAdd(float *A, float *B, float *C) {
    int tid = blockIdx.x * blockDim.x + threadIdx.x
    C[tid] = A[tid] + B[tid];
}
```



Memory Management

SPIE Medical Imaging

Memory allocation on the device

- use cudaMalloc(*mem, size)
- resulting pointer may be manipulated on the host
- allocated memory cannot be accessed from the host

Data transfer from and to the device

- use cudaMemcpy(devicePtr, hostPtr, size, HtoD) for host → device
- use cudaMemcpy(hostPtr, devicePtr, size, DtoH) for device → host

Example: Vector Add (GPU)

SPIE Medical Imaging

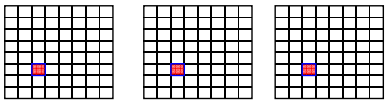
```
int N = 4096;        // allocate and initialize memory on the CPU
float *A = (float *) malloc(sizeof(float)*N);
float *B = (float *) malloc(sizeof(float)*N);
float *C = (float*) malloc(sizeof(float)*N)
init(A); init(B);
        // allocate and initialize memory on the GPU
float *d_A, *d_B, *d_C;
cudaMalloc(&d_A, sizeof(float)*N);   cudaMalloc(&d_B, sizeof(float)*N);
cudaMalloc(&d_C, sizeof(float)*N);
cudaMemcpy(d_A, A, sizeof(float)*N, HtoD);    cudaMemcpy(d_B, B, sizeof(float)*N, HtoD);
        // configure threads
dim3 dimBlock(32,1);   dim3 dimGrid(N/32,1);
        // run kernel on GPU
gpuVecAdd <<<dimGrid, dimBlock >>> (d_A, d_B, d_C);
        // copy result back to CPU
cudaMemcpy(C, d_C, sizeof(float)*N, DtoH);
        // free memory on CPU and GPU
cudaFree(d_A);   cudaFree(d_B);    cudaFree(d_C); free(A);   free(B);   free(C);
```

Example : Adding Matrix

SPIE Medical Imaging

C++

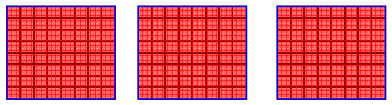
```
void AddMatrix (int *A, int *B, int *C,
                int w,  int d)
{   for ( int j = 0; j < d; j++)
      for( int i = 0; i < w; i++)
        { int index=j*w+i;
        C[index] = A[index] + B[index];
}}
```



CUDA

```
AddMatrix<<<1,dim3(w,d,1)>>>(A,B,C);
```

```
__global__ void AddMatrix (int *A,int *B,int *C)
{
        int i = threadIdx.x;
        int j = threadIdx.y;
        int id = j*blockDim.x+i;
        C[id]=A[id]+B[id];
}
```



- A simple parallel computing kernel
- It rewrites “for” loops as execution parameters
- Block ←→ multiprocessor (w*d<=1024)

Example : Adding Matrix

SPIE Medical Imaging

```
void * a,*b,*c;
cudaMalloc((void**)&a, w*d*sizeof(int));
cudaMalloc((void**)&b, w*d*sizeof(int));
cudaMalloc((void**)&c, w*d*sizeof(int));
…//load data into a, b;
cudaMemcpy(a, w*d*sizeof(int),cudaMemcpyHostToDevice);
cudaMemcpy(b, w*d*sizeof(int),cudaMemcpyHostToDevice);
dim3 dimBlock(BLOCKSIZE , BLOCKSIZE );
dim3 dimGrid( w/dimBlock.x, d/dimBlock.y );
AddMatrix <<<dimGrid, dimBlock>>> (a,b,c,w );
cudaMemcpy(c, w*d*sizeof(int),cudaMemcpyDeviceToHost);
cudaFree(a); cudaFree(b); cudaFree(c);
```

Allocate global memory (read and write)

Launch kernel

Copy result

Free memory

```
__global__ void AddMatrix (int *A, int *B, int *C, int w)
{
        int i = blockIdx.x *blockDim.x + threadIdx.x;
        int j = blockIdx.y *blockDim.y + threadIdx.y;
        int id = j* w +i;
        C[id]=A[id]+B[id];
}
```

Define kernel

Get threadIdx, blockIdx here

Parallel and Synchronize

Threads execute in asynchronous manner in general

- Threads within one block share memory and can synchronize

```
void __syncthreads();
```

- Once all threads have reached this point, execution resumes normally
- Used to avoid RAW / WAR / WAW hazards when accessing shared or global memory

CUDA Graphics API

CUDA Graphics API

- Texture (1D 2D 3D)

Texture Memory Advantages

Texture fetch versus global or constant memory read

- Cached, better performance if fetch with locality
- Not subject to the memory coalescing constraint for global and constant memory
- 2D address → (tex2D(tex, x, y))
- Filtering → (nearest neighbor or linear)
- Normalized coordinates → ([0,1] or [w, h])
- Handling boundary address → (clamp or wrap)
- Read only!

Binding textures

1. Declaring texture reference, format and cudaArray

```
texture<Type, Dim, ReadMode> texRef;
cudaArray* cu_array;
cudaChannelFormatDesc cudaCreateChannelDesc<T>();
```

2. Memory management

```
cudaMallocArray(), cudaFreeArray(), cudaMemcpyToArray()
```

3. Bind/Unbind texture before/after texture fetching

```
cudaBindTextureToArray(), cudaUnbindTexture()
```

- can also bind texture on pitched linear memory and linear memory

Example: Texture Example

```
texture<unsigned char, 2, cudaReadModeElementType> texr;
…

cudaChannelFormatDesc chDesc = cudaCreateChannelDesc<unsigned char>();
cudaArray* cuArray;
cudaMallocArray(&cuArray, &chDesc, w, h);
cudaMemcpyToArray( cuArray, 0, 0, input, data_size, cudaMemcpyHostToDevice );

cudaBindTextureToArray(texr, cuArray);
 …
//launch kernel. Inside kernel,  use tex2d(texr, idxX, idxY );
…//read result from device
cudaUnbindTexture(texr );
cudaFreeArray( cuArray );
cudaFree( data );
```

Bind texture before reference

Unbind texture after reference

Free cudaArray and memory

Thread-Level Optimization

Thread-Level Optimization

- Instruction Optimization
- Branching Overhead

Instruction Optimization

Compiling with “-usefastmath”

Single or double precision

Unrolling loops

- Overhead by loop/branching is relatively high

```
for(int k = -KERNEL_RADIUS; k <= KERNEL_RADIUS; k++)
    sum += data[sharMemPos + k] * d_Kernel[KERNEL_RADIUS - k];
```

- Results in 2-fold performance increase

```
sum = data[sharMemPos - 1] * d_Kernel[2]
      + data[sharMemPos + 0] * d_Kernel[1]
      + data[sharMemPos + 1] * d_Kernel[0];
```


Loop Overheads

Operation of original loop

```
for (int k = 0; k < BLOCK_SIZE; ++k)
    Pvalue += Ms[ty][k] * Ns[k][tx];
```

- 2 floating point arithmetic instructions
- 1 branch instruction
- 2 address arithmetic instructions
- 1 loop counter increment

→ 1/3 instructions are calculation instructions

→ limits performance to 1/3 of peak performance

Loop unrolling

- eliminates overhead
- in addition, array indexing can now be done using offsets

```
Pvalue += Ms[ty][k] * Ns[k][tx] + …
                Ms[ty][k+15] * Ns[k+15][tx];
```

Loop Unrolling

Compiler automatically unrolls small loops

Can use #pragma directive to enforce unrolling

```
#pragma unroll 5
for (int i = 0 ; i<10; i++)
```

- first 5 iterations will be unrollled
- *#pragma unroll 1* prevents compiler from unrolling
- *# pragma unroll* completely unrolls loop

Branching

If, **Switch**, **Do**, **While** statements

Can lead to diverging threads within a warp

- Threads executing different branches
- Will serialize the execution paths (disturbs parallelism)
- Rejoin when all diverging paths have been executed

Remedies

- Group warps into similar execution flows beforehand
- Unroll loops
- Remove branches algorithmically

Conclusions

- CUDA Introduction

Threads cooperate using shared memory

- CUDA Programming API

Launch parallel kernels

- CUDA Graphics API

Visualize the result

To come

- CUDA Performance Optimization

Course Schedule

1:30 – 1:45:	Introduction
1:45 – 2:00:	Parallel programming primer
2:00 – 2:15:	GPU hardware
2:15 – 3:00:	CUDA API, threads level optimization
	Coffee Break
3:30 – 4:00:	CUDA memory optimization (Eric)
4:00 – 4:15:	CUDA programming environment (Ziyi)
4:15 – 4:45:	Parallelism in medical image (Klaus)
4:45 – 5:25:	CT reconstruction examples (Eric + Ziyi)
5:25 – 5:30:	Closing remarks (Klaus)