May GeForce Be With You: A Brief Introduction to GPU Computing

Spencer Gessner
SASSin’ it up
July 25th, 2012
Outline

• CPUs vs. GPUs
• Examples with CUDA C
Why are GPUs Parallel?

• GPUs calculate pixel values.
• The value of pixel A has nothing to do with the value of pixel B.
• CPU calculates underlying object -> GPU updates all pixels.
CPU

One thread per clock cycle.
Low latency - quick read time from main memory.
Best for serial execution.

GPU

Many threads per clock cycle.
Low latency for operations on same block, high latency for reading from main memory.
Best for parallel execution.
i7 Architecture
Fermi Architecture
GPU architecture allows for stream processing!
GPGPU

• The massive parallelism of a GPU has applications to physics and math.

• For instance, linear algebra.
Vector addition is a parallel operation!

Vector addition is a parallel operation!
int main( void ) {

    int a[4] = {0,1,2,3};
    int b[4] = {4,5,6,7};
    int c[4] = {0};

    for (unsigned int n = 0; n < 4; n++) {
        c[n] = a[n] + b[n];
    }

    return 0;
}

Vector Addition on a GPU w/CUDA: The Kernel

```c
__global__ void add( int *a, int *b, int *c ) {
    int index = threadIdx.x + blockIdx.x * blockDim.x;
    c[index] = a[index] + b[index];
}
```
If we have $M$ threads/block, a unique array index for each entry given by

```
int index = threadIdx.x + blockIdx.x * M;
```

```
int index = x + y * width;
```

---

$M = 8$ threads/block

```
int index = threadIdx.x + blockIdx.x * M;
= 5 + 2 * 8;
= 21;
```
Launching a CUDA Kernel

add<<< 8192, 512 >>> (a, b, c);

- Number of blocks
- Number of threads per block
- Kernel operands
#define N (2048*2048)
#define THREADS_PER_BLOCK 512
int main( void ) {
    int *a, *b, *c;             // host copies of a, b, c
    int *dev_a, *dev_b, *dev_c; // device copies of a, b, c
    int size = N * sizeof( int );  // we need space for N integers

    // allocate device copies of a, b, c
    cudaMalloc( (void**)&dev_a, size );
    cudaMalloc( (void**)&dev_b, size );
    cudaMalloc( (void**)&dev_c, size );

    a = (int*)malloc( size );
    b = (int*)malloc( size );
    c = (int*)malloc( size );

    random_ints( a, N );
    random_ints( b, N );
Vector Addition on a GPU

```c
// copy inputs to device
cudaMemcpy( dev_a, a, size, cudaMemcpyHostToDevice );
cudaMemcpy( dev_b, b, size, cudaMemcpyHostToDevice);

// launch add() kernel with blocks and threads
add<<< N/THREADS_PER_BLOCK, THREADS_PER_BLOCK >>>( dev_a, dev_b, dev_c );

// copy device result back to host copy of c
cudaMemcpy( c, dev_c, size, cudaMemcpyDeviceToHost );

free( a ); free( b ); free( c );
cudaFree( dev_a );
cudaFree( dev_b );
cudaFree( dev_c );
return 0;
```
Host and Device

- **Host** - The CPU and its memory (host memory)
- **Device** - The GPU and its memory (device memory)
Dot Product

Multiply in parallel

Add in serial?
Dot Product Kernel

```
__global__ void dot(int *a, int *b, int *c) {
    // Each thread computes a pairwise product
    int temp = a[threadIdx.x] * b[threadIdx.x];
```
__global__ void dot( int *a, int *b, int *c )
{
    // Each thread computes a pairwise product
    int temp = a[threadIdx.x] * b[threadIdx.x];

    // Can’t compute the final sum
    // Each thread’s copy of ‘temp’ is private
}

Dot Product Kernel
Let’s Share

Block 0
- Threads
- Shared Memory

Block 1
- Threads
- Shared Memory

Block 2
- Threads
- Shared Memory

J. Sanders
Nvidia
Dot Product Kernel

```c
#define N 512
__global__ void dot( int *a, int *b, int *c ) {
    // Shared memory for results of multiplication
    __shared__ int temp[N];
    temp[threadIdx.x] = a[threadIdx.x] * b[threadIdx.x];

    // Thread 0 sums the pairwise products
    if( 0 == threadIdx.x ) {
        int sum = 0;
        for( int i = 0; i < N; i++ )
            sum += temp[i];
        *c = sum;
    }
}
```
Synching Threads

- Step 1: In parallel, each thread writes a pairwise product

  \[
  \text{\_shared\_ int temp}
  \]

- Step 2: Thread 0 reads and sums the products

  \[
  \text{\_shared\_ int temp}
  \]
Synching Threads

- Suppose thread 0 finishes its write in step 1

- Then thread 0 reads index 12 in step 2

This read returns garbage!
Dot Product Kernel

```c
__global__ void dot( int *a, int *b, int *c ) {
    __shared__ int temp[N];
    temp[threadIdx.x] = a[threadIdx.x] * b[threadIdx.x];

    __syncthreads();

    if( 0 == threadIdx.x ) {
        int sum = 0;
        for( int i = 0; i < N; i++ )
            sum += temp[i];
        *c = sum;
    }
}
```

But this will only work for a single block!
Multiblock Dot Product

Block 0

Block 1
Race Conditions and Global Memory

• Different blocks finish at different times (no __syncblocks)
• What happens when they add to ‘global’ sum?
What you want...

Block 0
\[
\text{sum} = 3
\]
\[\ast c \; \text{+= sum} \]

Read-Modify-Write
- Reads 0
- Computes 0 + 3
- Writes 3

Block 1
\[
\text{sum} = 4
\]

Read-Modify-Write
- Reads 3
- Computes 3 + 4
- Writes 7

J. Sanders
Nvidia
What you get

Block 0
sum = 3

Reads 0
0
Computes 0+3
0+3 = 3
Writes 3
3

*c += sum

Block 1
sum = 4

Reads 0
0
Computes 0+4
0+4 = 4
Writes 4
4
__global__ void dot( int *a, int *b, int *c ) {
    __shared__ int temp[THREADS_PER_BLOCK];
    int index = threadIdx.x + blockIdx.x * blockDim.x;
    temp[threadIdx.x] = a[index] * b[index];

    __syncthreads();

    if( 0 == threadIdx.x ) {
        int sum = 0;
        for( int i = 0; i < THREADS_PER_BLOCK; i++ )
            sum += temp[i];
        atomicAdd( c , sum );
    }
}
Dot Product on a GPU

```c
#define N (2048*2048)
#define THREADS_PER_BLOCK 512
int main( void ) {
    int *a, *b, *c;        // host copies of a, b, c
    int *dev_a, *dev_b, *dev_c;    // device copies of a, b, c
    int size = N * sizeof( int );    // we need space for N ints

    // allocate device copies of a, b, c
    cudaMalloc( (void**)&dev_a, size );
    cudaMalloc( (void**)&dev_b, size );
    cudaMalloc( (void**)&dev_c, sizeof( int ) );

    a = (int *)malloc( size );
    b = (int *)malloc( size );
    c = (int *)malloc( sizeof( int ) );

    random_ints( a, N );
    random_ints( b, N );
}
```
Dot Product on a GPU

```c
// copy inputs to device
cudaMemcpy( dev_a, a, size, cudaMemcpyHostToDevice );
cudaMemcpy( dev_b, b, size, cudaMemcpyHostToDevice );

// launch dot() kernel
dot<<< N/THREADS_PER_BLOCK, THREADS_PER_BLOCK >>>( dev_a, dev_b, dev_c );

// copy device result back to host copy of c
cudaMemcpy( c, dev_c, sizeof( int ) , cudaMemcpyDeviceToHost );

free( a ); free( b ); free( c );
cudaFree( dev_a );
cudaFree( dev_b );
cudaFree( dev_c );
return 0;
```
Ask Yourself...

Can my program benefit from massive parallelization? (maybe)
Do I want to deal with this CUDA BS? (maybe not)
Resources

Stanford’s GPU cluster: free if you have an SUNet ID.

NCLab.com: PyCUDA tutorials and programming space.

Amazon Work Space: pay-to-use Nvidia Tesla cards.
Bibliography

“The GPU Computing Era”

“Introduction to CUDA C”