Jacob Chmura

Parallel Programming

The system cpu memory is called *host* memory, and the gpu memory is called the *device* memory.

We have to cudamalloc memory on the device before we do anything. Host code can cast, perform arithmetic on, and pass around the cudamalloced pointer, but it cannot read or write from its memory address. Similarly host pointers on the device cannot be dereferenced.

To free memory alocated on the device, we use cudafree.

To access the result of a computation from device, we typically perform a cudamemcpy, from device to host. The final argument is either cudaMemcpyHostToDevice, cudaMemcpyDeviceToHOst,codaMemcpyDeviceToDevice.

We can see how many gpu devices are ready by cudaGetDeviceCount. Each device has a struct of properties called cudaDeviceProp, which stores the compute capabilities of a given device id gpu.

We can build a cudaDeviceProp struct, fill in the info we need for a given application, then make a call to cudagetDevice, which will try to find a device that satisfies the critera.

Kernel Parameters If we call a device code function like kernel¡¡¡N, 1¿¿¿ then we run with N parallel *blocks* of execution.

From within the kernel code, we can get the block along an axis by calling the cuda variable blockIdx.x.

The collection of parallel blocks is called a *grid*.

We can use the qualifier *__device__* which indicates that code will run on GPU and not the host, and hence will only be callable from other *__device__* functions, or from *__global__* functions,.

Adding vectors

```

___global___ void add(int *a, int *b, int *c){
    int tid = blockIdx.x;
    if(tid < N)
        c[tid] = a[tid] + b[tid];
}


#define N 10
int main( void ) {
    int a[N], b[N], c[N];
    int *dev_a, *dev_b, *dev_c;
    // allocate the memory on the GPU
    HANDLE_ERROR( cudaMalloc( (void**)&dev_a, N * sizeof(int) ) );
    HANDLE_ERROR( cudaMalloc( (void**)&dev_b, N * sizeof(int) ) );
    HANDLE_ERROR( cudaMalloc( (void**)&dev_c, N * sizeof(int) ) );
    // fill the arrays 'a' and 'b' on the CPU
    for (int i=0; i<N; i++) {
        a[i] = -i;
        b[i] = i * i;
    }

    HANDLE_ERROR(cudaMemcpy(dev_a, a, , N*sizeof(int), cudaMemcpyHostToDevice)
    HANDLE_ERROR(cudaMemcpy(dev_d, d, , N*sizeof(int), cudaMemcpyHostToDevice)

    add<<<N, 1>>>(dev_a, dev_b, dev_c);
```

```
27
28      // copy array c back from gpu to cpu
29      HANDLE_ERROR(cudaMemcpy(c, dev_c, N * sizeof(int), cudaMemcpyDeviceToHost));
30
31      cudaFree(dev_a);
32
33
34
```

Julia Set

```
1
2   struct cuComplex {
3    float r;
4    float i;
5    cuComplex( float a, float b ) : r(a), i(b) {}
6    __device__ float magnitude2( void ) {
7    return r * r + i * i;
8    }
9    __device__ cuComplex operator*(const cuComplex& a) {
10   return cuComplex(r*a.r - i*a.i, i*a.r + r*a.i);
11   }
12   __device__ cuComplex operator+(const cuComplex& a) {
13   return cuComplex(r+a.r, i+a.i);
14   }
15  };
16  __device__ int julia( int x, int y ) {
17   const float scale = 1.5;
18   float jx = scale * (float)(DIM/2 - x)/(DIM/2);
19   float jy = scale * (float)(DIM/2 - y)/(DIM/2);
20   cuComplex c(-0.8, 0.156);
21   cuComplex a(jx, jy);
22   int i = 0;
23   for (i=0; i<200; i++) {
24       a = a * a + c;
25       if (a.magnitude2() > 1000)
26           return 0;
27       }
28      return 1;
29  }
30
31  __global__ void kernel( unsigned char *ptr ) {
32   // map from threadIdx/BlockIdx to pixel position
33   int x = blockIdx.x;
34   int y = blockIdx.y;
35   int offset = x + y * gridDim.x;
36
37   // now calculate the value at that position
38   int juliaValue = julia( x, y );
39   ptr[offset*4 + 0] = 255 * juliaValue;
40   ptr[offset*4 + 1] = 0;
41   ptr[offset*4 + 2] = 0;
42   ptr[offset*4 + 3] = 255;
43  }
```

```
44
45  int main( void ) {
46   CPUBitmap bitmap( DIM, DIM );
47   unsigned char *dev_bitmap;
48   HANDLE_ERROR( cudaMalloc( (void**)&dev_bitmap, bitmap.image_size()));
49   dim3 grid(DIM,DIM);
50   kernel<<<grid,1>>>( dev_bitmap );
51   HANDLE_ERROR( cudaMemcpy( bitmap.get_ptr(), dev_bitmap, bitmap.image_size(),
52                  cudaMemcpyDeviceToHost ) );
53   bitmap.display_and_exit();
54   HANDLE_ERROR( cudaFree( dev_bitmap ) );
55  }
56
```

Thread Cooperation

Splitting Parallel Blocks

The second parameter to the kernel call specifies the number of *threads* to launch with.

$$kernel <<< Numblocks, Numthreads >>> \tag{1}$$

Previous examples always launched one thread per block.

Vector sum using threads

```
1
2   ___global___ void add(int *a, int *b, int *c){
3       int tid = threadIdx.x;
4       if(tid < N)
5           c[tid] = a[tid] + b[tid];
6   }
7
8
9   #define N 10
10  int main( void ) {
11      int a[N], b[N], c[N];
12      int *dev_a, *dev_b, *dev_c;
13      // allocate the memory on the GPU
14      HANDLE_ERROR( cudaMalloc( (void**)&dev_a, N * sizeof(int) ) );
15      HANDLE_ERROR( cudaMalloc( (void**)&dev_b, N * sizeof(int) ) );
16      HANDLE_ERROR( cudaMalloc( (void**)&dev_c, N * sizeof(int) ) );
17      // fill the arrays 'a' and 'b' on the CPU
18      for (int i=0; i<N; i++) {
19          a[i] = -i;
20          b[i] = i * i;
21      }
22
23      HANDLE_ERROR(cudaMemcpy(dev_a, a, , N*sizeof(int), cudaMemcpyHostToDevice)
24      HANDLE_ERROR(cudaMemcpy(dev_d, d, , N*sizeof(int), cudaMemcpyHostToDevice)
25
26      add<<<1, N>>>(dev_a, dev_b, dev_c);
```

```
27
28      // copy array c back from gpu to cpu
29      HANDLE_ERROR(cudaMemcpy(c, dev_c, N * sizeof(int), cudaMemcpyDeviceToHost));
30
31      cudaFree(dev_a);
32
33
34
```

The hardware limits the number of blocks in a single launch to 65,535. The hardware limits the number of threads per block to $maxThreadsPerBlock$, a device specific parameter given in the cudadevice struct. Typically around 512.

Vector addition with long vectors

```
1  #include "../common/book.h"
2  #define N (33 * 1024)
3  __global__ void add( int *a, int *b, int *c ) {
4   int tid = threadIdx.x + blockIdx.x * blockDim.x;
5   while (tid < N) {
6       c[tid] = a[tid] + b[tid];
7       tid += blockDim.x * gridDim.x;
8   }
9  }
10
11 int main( void ) {
12  int a[N], b[N], c[N];
13  int *dev_a, *dev_b, *dev_c;
14  // allocate the memory on the GPU
15  HANDLE_ERROR( cudaMalloc( (void**)&dev_a, N * sizeof(int) ) );
16  HANDLE_ERROR( cudaMalloc( (void**)&dev_b, N * sizeof(int) ) );
17  HANDLE_ERROR( cudaMalloc( (void**)&dev_c, N * sizeof(int) ) );
18  // fill the arrays 'a' and 'b' on the CPU
19  for (int i=0; i<N; i++) {
20      a[i] = i;
21      b[i] = i * i;
22  }
23  // copy the arrays 'a' and 'b' to the GPU
24  HANDLE_ERROR( cudaMemcpy( dev_a,a,N * sizeof(int),cudaMemcpyHostToDevice ) );
25  HANDLE_ERROR( cudaMemcpy( dev_b,b,N * sizeof(int),cudaMemcpyHostToDevice ) );
26  add<<<128,128>>>( dev_a, dev_b, dev_c );
27
28  // copy the array 'c' back from the GPU to the CPU
29  HANDLE_ERROR( cudaMemcpy( c,
30  dev_c,
31  N * sizeof(int),
32  cudaMemcpyDeviceToHost ) );
33
34 // free the memory allocated on the GPU
35  cudaFree( dev_a );
36  cudaFree( dev_b );
37  cudaFree( dev_c );
38  return 0;
39 }
```

Shared Memory and Synchronization

Cuda C makes available a region of memory that is called *shared memory*. We can modify variable declaration with the keyword *__shared__*.

This variable is copied for each block that you launch on the gpu. Every thread in that block shares the memory, but threads cannot see or moridyf the copy of this variable that is seen within other blocks.

Shared memory buffers reside physically on the gpu, as opposed to residing in off-chip DRAM. Thus, the latency to access shared memory is far lower than typical buffers.

To guarentee that all the theads have finished execution into a shared memory within a block, we call *__syncthreads*();. This guarentees that every thread in the block has completed instruction prior to the *__syncthreads*() call before the hardware will execute the next instruction.

Reductions of a shared array, ex: sum after elementwise multiplication for dot product

If we make a single thread add the result, we have linear complexity. We can instead divide up the work to get log complexity.

```
1  // for reductions, threadsPerBlock must be a power of 2
2   // because of the following code
3   int i = blockDim.x/2;
4   while (i != 0) {
5       if (cacheIndex < i)
6           cache[cacheIndex] += cache[cacheIndex + i];
7       __syncthreads();
8       i /= 2;
9       }
10 }
```

We cannot put a *__syncthreads* call in a conditional that diverges for threads. This is because syncthreads waits until all threads within a block hit the instruction, which will never happen.

Constant Memory and Events

Constant Memory

Often the bottleneck is not arithmetic throughput, but memory bandwidth. It is therefore worth investigating how to reduce the amount of memory traffic required for a given problem.

Constant memory is a memory buffer for data that will not change over the course of a kernel execution. NVidia gives us 64kb of constant memory.

We declare a variable constant with the *__constant__* declaration. We do not need to cudamalloc or cudafree this memory. We do need to commit to the array size at compile time.

To put data from cpu to constant memory we do *cudaMemcpyToSymbol* instead of *cudaMemcpy*.

- A single read from constant memory can be broadcast to other nearby threads
- Constant memory is cached, so consecutive reads of the same adress will not incur any additional memory traffic

A *warp* refers to a collection of 32 threads that are woven together, and get executed in lockstep. At every line of the program, ecah thread in a warp executes the same instruction on different data. Nvidia can broadcast a single memory read from constant memory to a half-warp, a group of 16 threads.

Events

An *event* in CUDa is a Gpu time stamp that is recorded at a user-specified point in time. This removes problems in measuring GPU execution with CPU timers.

```

  cudaEvent_t start, stop;
  cudaEventCreate(&start);
  cudaEventCreate(&stop);
  cudaEventRecord(start, 0);

  // do some work on gpu

  cudaEventRecord(stop, 0);
  cudaEventSynchornize(stop); // requires the Gpu to block further instruction until the GPU has reac

  float elapsedTime = cudaEventElapsedTime(&elapsedTime, start, stop);

  // cleanup
  cudaEventDestroy(start);
  cudaEventDestroy(stop);

```

Texture Memory

Texture memory is a read-only memory that is cached on chip, designed for memory access patterns that exhibit a great deal of spatial locality: a thread is likely to read from an address near the address that nearby threads read.

We declare texture inputs like:

```

  texture<float> texConstSrc;
  cudaBindTexture(NULL, texConstSrc, dev_constSrc, size);

  // 2d
  texture<float, 2> texConstSrc2d;
  cudaChannelFormatDesc, desc = cudaCreateChannelDesc<float>();
  cudaBindTexture2d(NULL, texConstSrc2d, dev_constSrc2d, desc, DIM, DIM, size);
```

We can read texture memory from inside a kernel like:

```

  __global__ void kernel(float *dst, bool dstOut){
      // dstOut tells us which texture to fetch, if we have multiple.
      if (dstOut){
          res = tex1Dfetch(texIn, idx);
      }
      else{
          res = tex1Dfetch(texout, idx);
      }

      // 2d version
```

```
12      res = tex2D(tesConstSrc2d, xidx, yidx);
13
14  }
```

We cleanup by *unbinding* texture memory like:

```
1
2   cudaUnbindTexture(texIn);
```

Graphics Interoperability

Cuda C offers apis that allow you to do both general purpose computation, *and* rendering easily within the same application.

We need the device id so that we can tell the cuda runtime that we intent to sue the device for CUDA *and* OpenGL.

Example

```
1
2   #define GL_GLEXT_PROTOTYPES
3   #include "GL/glut.h"
4   #include "cuda.h"
5   #include "cuda_gl_interop.h"
6   #include "../common/book.h"
7   #include "../common/cpu_bitmap.h"
8   #define DIM 512
9
10  // globals containing pointer to the same data buffer.
11  GLuint bufferObj; // for opengl
12  cudaGraphicsResource *resource; // for computation on cuda
13
14  int main( int argc, char **argv ) {
15   cudaDeviceProp prop;
16   int dev;
17   memset( &prop, 0, sizeof( cudaDeviceProp ) );
18   prop.major = 1;
19   prop.minor = 0;
20   HANDLE_ERROR( cudaChooseDevice( &dev, &prop ) );
21
22   HANDLE_ERROR( cudaGLSetGLDevice(dev));
23
24   // these GLUT calls need to be made before the other GL calls
25   glutInit( &argc, argv );
26   glutInitDisplayMode( GLUT_DOUBLE | GLUT_RGBA );
27   glutInitWindowSize( DIM, DIM );
28   glutCreateWindow( "bitmap" );
29
30   // To pass data between OpenGL and CUDA, we first need to
31   // create a buffer that can be used with both APIs.
32   // We start by creating a pixel buffer object in OpenGL and
33   // storing the handle in our global variable GLuint bufferObj
```

```
34    glGenBuffers( 1, &bufferObj );
35    glBindBuffer( GL_PIXEL_UNPACK_BUFFER_ARB, bufferObj );
36    glBufferData( GL_PIXEL_UNPACK_BUFFER_ARB, DIM * DIM * 4, NULL, GL_DYNAMIC_DRAW_ARB );
37
38    // notifying CUDA runtime that we intend to share the OpenGL buffer bufferObj
39    HANDLE_ERROR(cudaGraphicsGLRegisterBuffer( &resource, bufferObj,cudaGraphicsMapFlagsNone)
40
41    // need address in device memory that can be passed to our kernel
42    uchar4* devPtr;
43    size_t size;
44    HANDLE_ERROR(cudaGraphicsMapResources( 1, &resource, NULL));
45    HANDLE_ERROR(cudaGraphicsResourceGetMappedPointer( (void**)&devPtr, &size,resource));
46
47
48    dim3 grids(DIM/16,DIM/16);
49    dim3 threads(16,16);
50    kernel<<<grids,threads>>>( devPtr );
51
52    // ensure we have synchronization between cuda and graphics
53    HANDLE_ERROR( cudaGraphicsUnmapResources( 1, &resource, NULL ) );
54
55    // set up GLUT and kick off main loop
56    glutKeyboardFunc( key_func );
57    glutDisplayFunc( draw_func );
58    glutMainLoop();
59  }
```

Atomics

Compiling with certain compute capability: nvcc -arch=sm_11 is 1.1 or greater.

An *Atomic* operation is one that cannotbe broken into smaller parts by other threads, hence ensuring well-defined behaviour when many threads are potentially competing for access.

Histogram Computation

```
1
2   #include "../common/book.h"
3   #define SIZE (100*1024*1024)
4   int main( void ) {
5    unsigned char *buffer = (unsigned char*)big_random_block( SIZE );
6    unsigned int histo[256];
7    for (int i=0; i<256; i++)
8       histo[i] = 0;
9
10   for (int i=0; i<SIZE; i++)
11      histo[buffer[i]]++;
12
13   free( buffer );
14   return 0;
15  }
```

Histogram Computation on GPU

```
1
2  __global__ void bad_histo_kernel( unsigned char *buffer, long size, unsigned int *histo ) {
3   int i = threadIdx.x + blockIdx.x * blockDim.x;
4   int stride = blockDim.x * gridDim.x;
5   while (i < size) {
6      atomicAdd( &(histo[buffer[i]]), 1 ); // super slow because alot of threads wait
7      i += stride;
8   }
9  }
10 __global__ void histo_kernel( unsigned char *buffer,long size, unsigned int *histo ) {
11      __shared__ unsigned int temp[256];
12      temp[threadIdx.x] = 0;
13      __syncthreads();
14
15      int i = threadIdx.x + blockIdx.x * blockDim.x;
16      int offset = blockDim.x * gridDim.x;
17      while (i < size) {
18          atomicAdd( &temp[buffer[i]], 1);
19          i += offset;
20      }
21      __syncthreads();
22     atomicAdd( &(histo[threadIdx.x]), temp[threadIdx.x] );
23 }
24 int main( void ) {
25  unsigned char *buffer = (unsigned char*)big_random_block( SIZE );
26  cudaEvent_t start, stop;
27  HANDLE_ERROR( cudaEventCreate( &start ) );
28  HANDLE_ERROR( cudaEventCreate( &stop ) );
29  HANDLE_ERROR( cudaEventRecord( start, 0 ) );
30
31  // allocate memory on the GPU for the file's data
32  unsigned char *dev_buffer;
33  unsigned int *dev_histo;
34  HANDLE_ERROR( cudaMalloc( (void**)&dev_buffer, SIZE ) );
35  HANDLE_ERROR( cudaMemcpy( dev_buffer, buffer, SIZE, cudaMemcpyHostToDevice ) );
36  HANDLE_ERROR( cudaMalloc( (void**)&dev_histo, 256 * sizeof( long ) ) );
37  HANDLE_ERROR( cudaMemset( dev_histo, 0, 256 * sizeof( int ) ) );
38
39
40
41  cudaDeviceProp prop;
42  HANDLE_ERROR( cudaGetDeviceProperties( &prop, 0 ) );
43  int blocks = prop.multiProcessorCount;
44  histo_kernel<<<blocks*2,256>>>( dev_buffer, SIZE, dev_histo );
45
46  unsigned int histo[256];
47  HANDLE_ERROR( cudaMemcpy( histo, dev_histo,256 * sizeof( int ), cudaMemcpyDeviceToHost ) );
48
49  // get stop time, and display the timing results
50  HANDLE_ERROR( cudaEventRecord( stop, 0 ) );
51  HANDLE_ERROR( cudaEventSynchronize( stop ) );
52  float elapsedTime;
53  HANDLE_ERROR( cudaEventElapsedTime( &elapsedTime, start, stop ) );
54  printf( "Time to generate: %3.1f ms\n", elapsedTime );
```

```
55
56    HANDLE_ERROR( cudaEventDestroy( start ) );
57    HANDLE_ERROR( cudaEventDestroy( stop ) );
58    cudaFree( dev_histo );
59    cudaFree( dev_buffer );
60    free( buffer );
61    return 0;
62  }
```

We need to pay close attention to try and alleviate contention for global memory addresses when using atomics.

Streams

cudaHostAlloc is a directive that allocates memory on the host, to be used instead of malloc.

Malloc allocates standard, pageable host memory, while cudahostalloc allocates a buffer of *page-locked* host memory, also called *pinned memory*, which guarentees that it will never page this memory out to disk, and hence ensures its residency in physical memory. Thus it becomes safe for the OS to allow an applciation access to the physical address of the memory, since the buffer will not be evicted or relocated.

Thus, the GPU can use *direct memory access (DMA)* to copy data to or from the host, which proceed without intervention from the CPU. Actually, all copies occur twice (first from pageable to pinned, then from pinned to device). Hence using pinned is roughly 2x speedup.

Downside is you don't get the nice features of virtual memory: the computer running the application needs to have available physical memory for every page-locked buffer, since the buffers can never be swapped out to disk.

Multi-gpus

Advanced Atomics