Introduction to CUDA C

Hands on

S. Sundar and M. Panchatcharam

August 9, 2014

Outline

Introduction to CUDA C
- First Program

Matrix Matrix Multiplication
- MatrixMul in GPU
- Tiled Matrices
- Shared Memory

CUDA Streams

Simple C Program

At this point let us look at the first CUDA program

```
#include<stdio.h>
int main()
{
  printf("Hello World\n");
  return 0;
}
```

It is a simple C program, there is nothing to do with CUDA

Kernel Call

Let us call an empty kernel from GPU

```
#include<stdio.h>
__global__ void kernel(void){}
#include<stdio.h>
int main()
{
  printf("Hello World\n");
  return 0;
}
```

Again there is nothing much. It calls a CUDA kernel but that does nothing

How to compile

- In Ubuntu, go to terminal and type `nvcc kernel.cu`, where kernel is the name you have named for your program.
- It produces a default output as a.out, now run ./a.out
- In case of Windows, press F7 or build solution using visual studion 2010/13/05/08
- Run the program or F5

Add two numbers

Let us add two numbers

```
#include<stdio.h>
__global__ void add(int a, int b, int *c)
{
  *c=a+b;
}
int main()
{
  int c,*devc;
  cudaMalloc(void**)&devc,sizeof(int));
  add<<<1,1>>>(2,5,dev  c);
  cudaMemcpy(&c,devc,sizeof(int),cudaMemcpyDeviceToHost);
  printf("2+5 =  d\n",c);
  cudaFree(devc);
  return 0;
}
```

Again there is nothing much. It calls a CUDA kernel but that does nothing

Add two numbers

- `cudaMalloc()` works similar to `malloc()` in CPU
- `cudaMalloc()` allocates the memory in GPU
- `cudaMemcpy()` copies data between host and device
- `cudaFree()` frees the allocated memory of GPU
- Once `add` kernel is launched from CPU, it transfers the value 2 and 5 from CPU to GPU
- GPU adds the two values and stores them in c
- cudaMemcpy copies data from GPU to CPU

cudaMemcpy

- `cudaMemcpy()` copies data between host and device
- Syntax `cudaMemcpy(destination, source, sizeof(source),TransferType);`
- Mention the address of the destination and source
- Data transfer is depending on the following for keywords
  - `cudaMemcpyDeviceToHost` transfers from GPU to CPU
  - `cudaMemcpyDeviceToDevice` transfers from GPU to GPU
  - `cudaMemcpyHostToHost` transfers from CPU to CPU
  - `cudaMemcpyHostToDevice` transfers from CPU to GPU

Add two Vectors

A	3	6	2	0	-2	...
+						
B	2	3	1	1	2	...
=						
C	5	9	3	1	0	...

Add two Vectors

Let us add two vectors

```
#include <stdio.h>
int main()
{
  const int a[size] = { 1, 2, 3, 4, 5 };
  const int b[size] = { 10, 20, 30, 40, 50 };
  int c[size] = { 0 };int *da,*db,*dc
  cudaMalloc((void**)& dc, size * sizeof(int));
  cudaMalloc((void**)& da, size * sizeof(int));
  cudaMalloc((void**)& db, size * sizeof(int));
  cudaMemcpy( da,a,size*sizeof(int),cudaMemcpyHostToDevice);
  cudaMemcpy( db,b,size*sizeof(int),cudaMemcpyHostToDevice);
  addvec<<<1, size>>>( dc,  da,  db);
  cudaMemcpy(c, dc,size*sizeof(int),cudaMemcpyDeviceToHost);
  printf("{1,2,3,4,5} + {10,20,30,40,50} = {%d,%d,%d,%d,%d}\
      n",c[0], c[1], c[2], c[3], c[4]);
  cudaThreadSynchronize();
  cudaFree( dc);  cudaFree( da);  cudaFree( db);
  return 0;
```

Add two Vectors

Vector Kernel

```
__global__ void addvec(int *c, const int *a, const int *b)
{
  int i = threadIdx.x;
  c[i] = a[i] + b[i];
}
```

Add two Long Vectors

Vector Addition

```
#include <stdio.h>
int main()
{
  int N=2048;
  int a[N],b[N],c[N],*da,*db,*dc;
  for(int i=0; i<N;i++)
  { a[i]=-i;b[i] = i*i;}
  cudaMalloc((void**)& dc, N*sizeof(int));
  cudaMalloc((void**)& da, N*sizeof(int));
  cudaMalloc((void**)& db, N*sizeof(int));
  cudaMemcpy( da,a,N*sizeof(int),cudaMemcpyHostToDevice);
  cudaMemcpy( db,b,N*sizeof(int),cudaMemcpyHostToDevice);
  addvec<<<(N+127)/128, 128>>>( dc,  da,  db);
  cudaMemcpy(c, dc,N*sizeof(int),cudaMemcpyDeviceToHost);
  cudaFree( dc);  cudaFree( da);  cudaFree( db);
  return 0;
}
```

Add two Long Vectors

addvec kernel

```
__global__ void addvec(int *dc, const int *da, const int *db
    )
{
  int tid = blockIdx.x * blockDim.x + threadIdx.x ;
  while ( tid < N )
  {
    dc [tid] = da[tid] + db[tid] ;
    tid+= blockDim.x * gridDim.x ;
  }
}
```

Matrix Multiplication CPU

```
// Multiply two matrices A * B = C

#include <stdlib.h>
#include <stdio.h>
#include <math.h>

// Allocates a matrix with random float entries.
void randomInit(float* data, int size)
{
    for (int i = 0; i < size; ++i)
        data[i] = rand() / (float)RAND_MAX;
}
```

Matrix Multiplication CPU

```
// Program main
int main(int argc, char** argv)
{
     // set seed for rand()
    srand(2006);

    // 1. allocate host memory for matrices A and B
    unsigned int size_A = WA * HA;
    unsigned int mem_size_A = sizeof(float) * size_A;
    float* h_A = (float*) malloc(mem_size_A);

    unsigned int size_B = WB * HB;
    unsigned int mem_size_B = sizeof(float) * size_B;
    float* h_B = (float*) malloc(mem_size_B);

    // 2. initialize host memory
    randomInit(h_A, size_A);
    randomInit(h_B, size_B);
```

Matrix Multiplication CPU

```
// 3. print out A and B
printf("\n\nMatrix A\n");
for(int i = 0; i < size_A; i++)
{
   printf("%f ", h_A[i]);
   if(((i + 1) % WA) == 0)
      printf("\n");
}
    printf("\n\nMatrix B\n");
for(int i = 0; i < size_B; i++)
{
   printf("%f ", h_B[i]);
   if(((i + 1) % WB) == 0)
      printf("\n");
}
// 4. allocate host memory for the result C
unsigned int size_C = WC * HC;
unsigned int mem_size_C = sizeof(float) * size_C;
float* h_C = (float*) malloc(mem_size_C);
```

Matrix Multiplication CPU

```
    // 5. perform the calculation

     // 6. print out the results
    printf("\n\nMatrix C (Results)\n");
    for(int i = 0; i < size_C; i++)
    {
        printf("%f ", h_C[i]);
        if(((i + 1) % WC) == 0)
            printf("\n");
    }
    printf("\n");
     // 7. clean up memory
    free(h_A);
    free(h_B);
    free(h_C);
}
```

Matrix Multiplication GPU

In between 4 and 5 modify as follows

```
// 4. allocate host memory for the result C
unsigned int size_C = WC * HC;
unsigned int mem_size_C = sizeof(float) * size_C;
float* h_C = (float*) malloc(mem_size_C);
// 8. allocate device memory
float* d_A;
float* d_B;
cudaMalloc((void**) &d_A, mem_size_A);
cudaMalloc((void**) &d_B, mem_size_B);

// 9. copy host memory to device
cudaMemcpy(d_A, h_A,mem_size_A,cudaMemcpyHostToDevice);
cudaMemcpy(d_B, h_B,mem_size_B,cudaMemcpyHostToDevice);

// 10. allocate device memory for the result
float* d_C;
cudaMalloc((void**) &d_C, mem_size_C);
// 5. perform the calculation
```

Matrix Multiplication GPU

After the calculation performance, that is calling CUDA kernel, do the following

```
  // 5. perform the calculation
     ...

   // 11. copy result from device to host
   cudaMemcpy(h_C, d_C, mem_size_C,
              cudaMemcpyDeviceToHost);
```

Finally free the memory

```
   cudaFree(d_A);
   cudaFree(d_B);
   cudaFree(d_C);
```

Matrix Multiplication Kernel

Look at the following header files (matrixMulkernel.cu)

```
#ifndef _MATRIXMUL_KERNEL_H_
#define _MATRIXMUL_KERNEL_H_

#include <stdio.h>

// Thread block size
#define BLOCK_SIZE 3

#define WA 3   // Matrix A width
#define HA 3   // Matrix A height
#define WB 3   // Matrix B width
#define HB WA  // Matrix B height
#define WC WB  // Matrix C width
#define HC HA  // Matrix C height
```

Matrix Multiplication Kernel

Kernel

```
// CUDA Kernel
__global__ void matrixMul( float* C, float* A, float* B, int
    wA, int wB)
{    // 2D Thread ID
  int tx = threadIdx.x;   int ty = threadIdx.y;
  // value stores the element that is computed by the
      thread
  float value = 0;
  for (int i = 0; i < wA; ++i)
  {
    float elementA = A[ty * wA + i];
    float elementB = B[i * wB + tx];
    value += elementA * elementB;
  }
   // Write the matrix to device memory each
  // thread writes one element
  C[ty * wA + tx] = value;
}
```

Matrix Multiplication Kernel

Include the matrixMukKernel.cu at the top

```
// Multiply two matrices A * B = C

#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <matrixMulkernel.cu>
```

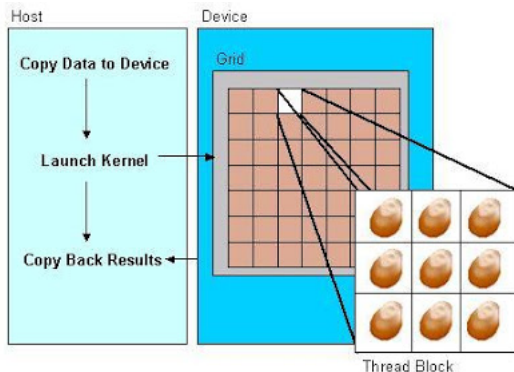
Matrix Multiplication Kernel

Finally include the following codes at perform calculations

```
// 5. perform the calculation
//     setup execution parameters
dim3 threads(BLOCK_SIZE, BLOCK_SIZE);
dim3 grid(WC / threads.x, HC / threads.y);

//    execute the kernel
matrixMul<<< grid, threads >>>(d_C, d_A,
                                d_B, WA, WB);
```

Matrix Multiplication Kernel



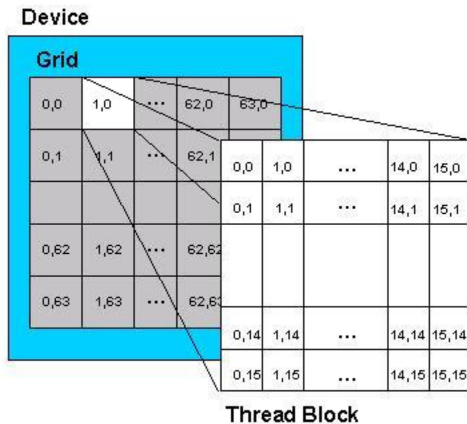
Matrix Multiplication

- When our matrixMul( ) kernel is launched multiple threads will be created
- Each invocation of our kernel uses the two thread indices
- It Identifies the row of A and the column of B to perform dot product operation
- Calculation based on their unique thread indices (threaded.x, threaded.y)
- Iterates through a loop to calculate the dot product
- Determines where to write the result

How does MatrixMul Work in CPU

How does MatrixMul Work in GPU

Let us change the matrix multiplication using tiled format



How does MatrixMul Work in GPU

Kernel Changes

Finally include the following codes at perform calculations

```
#ifndef _MATRIXMUL_KERNEL_H_
#define _MATRIXMUL_KERNEL_H_

#include <stdio.h>

// Thread block size
#define BLOCK_SIZE 16
#define TILE_SIZE  16

#define WA 1024   // Matrix A width
#define HA 1024   // Matrix A height
#define WB 1024   // Matrix B width
#define HB WA     // Matrix B height
#define WC WB     // Matrix C width
#define HC HA     // Matrix C height
```

Kernel Changes

Finally include the following codes at perform calculations

```
// CUDA Kernel
__global__ void
matrixMul( float* C, float* A, float* B, int wA, int wB)
{  // 1. 2D Thread ID
   int tx = blockIdx.x * TILE_SIZE + threadIdx.x;
   int ty = blockIdx.y * TILE_SIZE + threadIdx.y;

   // value stores the element that is
   // computed by the thread
   float value = 0;
   for (int i = 0; i < wA; ++i)
   {
      float elementA = A[ty * wA + i];
      float elementB = B[i * wB + tx];
      value += elementA * elementB;
   }
```

Kernel Changes

Finally include the following codes at perform calculations

```
    // Write the matrix to device memory each
    // thread writes one element
    C[ty * wA + tx] = value;
}

#endif // #ifndef _MATRIXMUL_KERNEL_H_
```

- The code in section 1 has been changed to take the index of the block (and the size of the tile) into consideration.
- Didn't change the code for pulling out elementA and elementB from our input matrices
- both input marices are based on tx and ty

Matrix Multiplication using shared memory

Let us change the kernel by further tiling

```
/* Matrix multiplication: C = A * B.
 * Device code.
 */
#ifndef _MATRIXMUL_KERNEL_H_
#define _MATRIXMUL_KERNEL_H_
// Thread block size
#define BLOCK_SIZE 16
#define TILE_SIZE  16
#define WA 1024   // Matrix A width
#define HA 1024   // Matrix A height
#define WB 1024   // Matrix B width
#define HB WA     // Matrix B height
#define WC WB     // Matrix C width
#define HC HA     // Matrix C height
```

Kernel Changes

```
//! Matrix multiplication on the device: C = A * B
//! wA is A's width and wB is B's width
__global__ void
matrixMul( float* C, float* A, float* B, int wA, int wB)
{
    // Block index
    int bx = blockIdx.x;
    int by = blockIdx.y;
     // Thread index
    int tx = threadIdx.x;
    int ty = threadIdx.y;
     // Index of the first sub-matrix of A processed
    // by the block
    int aBegin = wA * BLOCK_SIZE * by;
     // Index of the last sub-matrix of A processed
    // by the block
    int aEnd    = aBegin + wA - 1;
```

Kernel Changes

```
// Step size used to iterate through the
 // sub-matrices of A
 int aStep  = BLOCK_SIZE;
 // Index of the first sub-matrix of B processed
 // by the block
 int bBegin = BLOCK_SIZE * bx;
 // Step size used to iterate through the
 // sub-matrices of B
 int bStep  = BLOCK_SIZE * wB;
 // Loop over all the sub-matrices of A and B
 // required to compute the block sub-matrix
 for (int a = aBegin, b = bBegin;
          a <= aEnd;
          a += aStep, b += bStep)
 {
     // Declaration of the shared memory array As
     // used to store the sub-matrix of A
     __shared__ float As[BLOCK_SIZE][BLOCK_SIZE];
     // Declaration of the shared memory array Bs
     // used to store the sub-matrix of B
```

Kernel Changes

```
__shared__ float Bs[BLOCK_SIZE][BLOCK_SIZE];
 // Load the matrices from global memory
// to shared memory; each thread loads
// one element of each matrix
As[ty][tx] = A[a + wA * ty + tx];
Bs[ty][tx] = B[b + wB * ty + tx];
// Synchronize to make sure the matrices
// are loaded
__syncthreads();
// Multiply the two matrices together;
// each thread computes one element
// of the block sub-matrix
for (int k = 0; k < BLOCK_SIZE; ++k)
    Csub += As[ty][k] * Bs[k][tx];
// Synchronize to make sure that the preceding
// computation is done before loading two new
// sub-matrices of A and B in the next iteration
__syncthreads();
```

Kernel Changes

```
    }
    // Write the block sub-matrix to device memory;
    // each thread writes one element
    int c = wB * BLOCK_SIZE * by + BLOCK_SIZE * bx;
    C[c + wB * ty + tx] = Csub;
}
```

Matrix Multiplication using shared memory

- The kernel code above uses blockIdx to determine the start and end location of our two sub matrices.
- Loops through our input matrices and has each thread in the thread block copy one cell of the A and B sub matrices into shared memory.
- Keep in mind that shared memory has thread block scope so each thread in the thread block can see the data copied over by the other threads.

CUDA streams

CUDA stream

A stream is a sequence of operations that are performed in order on the device. Streams allows independent concurrent in-order queues of execution.

- Operations in different streams can be interleaved and overlapped, which can be used to hide data transfers between host and device.
- Use cudaStreamCreate() (runtime API) or cuStreamCreate() (Driver API) to create a stream of type cudaStream_t
- The default stream (ID=0) need not be create.
- Multiple streams exist within a single context, they share memory and other resources.
- Copies & Kernel launches with the same stream parameter execute in-order

CUDA stream

C2050 Execution Time Lines

Sequential Version

Asynchronous Versions 1and 3

Asynchronous Version 2

THANK YOU