CS402: Lab Session 8
CUDA

1 Introduction

In the labs up until now we have been looking a mechanisms for exposing parallelism on CPUs. For this lab we will be looking at the CUDA, which is a programming model designed for GPUs. GPUs offer a finer level of data parallelism than a CPU, with which large speedups are possible if you application is well suited. The CUDA programming model is vastly different from OpenMP and MPI. This lab session will touch on kernels and how they run in the context of the thread hierarchy abstraction. For more information, you can refer to the *CUDA C Programming Guide*. The contents covered in this lab session are mainly in sections 2.1, 2.2 and 2.3.

2 Hello World in CUDA

As with the other labs, the first example we will look at is a variant of *"Hello, world!"*, which has been modified to demonstrate a minimal CUDA program. Normally, the parallel program is modified such that each thread prints out its `ID` and *"Hello, world!"*, but this is not possible as we cannot print to the screen from code running on an GPU.


```
#include <stdio.h>

const int N = 16;
const int blocksize = 16;

__global__
void hello(char *a, int *b) {
  a[threadIdx.x] += b[threadIdx.x];
}

int main(){
  char a[N] = "Hello \0\0\0\0\0\0";
  int b[N] = {15, 10, 6, 0, -11, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0};
  char *ad;
  int *bd;
  const int csize = N*sizeof(char);
  const int isize = N*sizeof(int);

  printf("%s", a);

  cudaMalloc( (void**)&ad, csize );
  cudaMalloc( (void**)&bd, isize );
  cudaMemcpy( ad, a, csize, cudaMemcpyHostToDevice );
```

```
  cudaMemcpy( bd, b, isize, cudaMemcpyHostToDevice );

  dim3 dimBlock( blocksize, 1 );
  dim3 dimGrid( 1, 1 );
  hello<<<dimGrid, dimBlock>>>(ad, bd);
  cudaMemcpy( a, ad, csize, cudaMemcpyDeviceToHost );
  cudaFree( ad );
  cudaFree( bd );

  printf("%s\n", a);
  return 0;
}
```

Listing 1: CUDA "Hello World" example

In this case we create an array, `a`, equal to {`'H','e','l','l','o',' '`}, then we use CUDA to transform this into {`'W','o','r','l','d','!'`} using the fact that we can add, for example, 15 to the ASCII value of "H" to create a "W". The difference values are stored in the array `b`. You can find this code in Listing 1 and the file named `helloworldCUDA.cu`.

CUDA code is not as easy to compile as the examples in previous labs and as such requires two steps. The first involves setting up the *environment* for the compiler, by using the `source` command on the file named `environment.sh` which is included as part of this lab.


```
source environment.sh
```


If you open this file you will see that it appends to the environmental variables `$PATH` and `$LD_LIBRARY_PATH` with the location of the CUDA compiler and CUDA libraries respectively. The second step is the compilation process, which differs from the compilation of plain C in that you must use the `nvcc` command. This is a wrapper around `gcc` which takes care of the CUDA language extensions.


```
nvcc  -o helloworldCUDA.b helloworldCUDA.cu
```

Run the compiled binary just like any other, and you will be greeted with the output below.

```
Hello World!
```

The following list contains explanations for the CUDA specific portions of the code.

- `__global__` – This annotation is used to declare a kernel.
- `cudaMalloc` – This is similar to malloc in C but allocates memory on the GPU.
- `cudaMemcpy` – This copies data from hosts memory to device memory.
- `hello<<<dimGrid, dimBlock>>>(ad,bd)` – This invokes the kernel in much the same was as a function is called in C. The main difference is the execution configuration within the triple chevrons. The first argument in the execution configuration specifies the number of thread blocks in the grid, and the second specifies the number of threads per thread block.

- `threadIdx.x` – The ID of the thread executing the kernel.

As always, if you have any questions at this point, make sure you ask one of the tutors for help.

3 Using Threads

Task 1 This task will be carried out in the context of a Single-precision A*X Plus Y (SAXPY) example written in CUDA, which can be found in Listing 2 and the file named `saxpy.cu`. Your objective is to use the knowledge gained from the lectures and the CUDA programming guide to complete the kernel; the key is in understanding the thread hierarchy and how these relate to `threadIdx`, `blockIdx` and `blockDim`. You will know if you have completed the task correctly as the reported error value will reduce from 2.0 to 0.0.

```
#include <stdio.h>

__global__
void saxpy(int n, float a, float *x, float *y){
  //TODO
}

int main(void){
  int N = 1<<20;
  float *x, *y, *d_x, *d_y;
  x = (float*)malloc(N*sizeof(float));
  y = (float*)malloc(N*sizeof(float));

  cudaMalloc(&d_x, N*sizeof(float));
  cudaMalloc(&d_y, N*sizeof(float));

  for (int i = 0; i < N; i++) {
    x[i] = 1.0f;
    y[i] = 2.0f;
  }

  cudaMemcpy(d_x, x, N*sizeof(float), cudaMemcpyHostToDevice);
  cudaMemcpy(d_y, y, N*sizeof(float), cudaMemcpyHostToDevice);
  saxpy<<<(N+255)/256, 256>>>(N, 2.0, d_x, d_y);
  cudaMemcpy(y, d_y, N*sizeof(float), cudaMemcpyDeviceToHost);

  float maxError = 0.0f;
  for (int i = 0; i < N; i++)
    maxError = max(maxError, abs(y[i]-4.0f));
  printf("Max error: %f\n", maxError);
}
```

Listing 2: CUDA SAXPY example

4 Assessing Performance

Performance analysis requires you to time the various portions of your code, but this works a little differently in CUDA than seen in previous labs. Timing CUDA code relies on creating, firing, synchronising and calculating the time difference of events using `cudaEventCreate(cudaEvent_t e)`, `cudaEventRecord(cudaEvent_t e)`, `cudaEventSynchronize(cudaEvent_t e)` and `cudaEventElapsedTime(float*`

`out, cudaEvent_t e, cudaEvent_t e)` respectively. These functions are used to time a region of code as shown in Listing 3.

```
cudaEvent_t start, stop;
cudaEventCreate(&start);
cudaEventCreate(&stop);

cudaEventRecord(start);
//Region to time.
cudaEventRecord(stop);

//blocks CPU until the stop event has been completed.
cudaEventSynchronize(stop);
float milliseconds = 0.0;
cudaEventElapsedTime(&milliseconds, start, stop);
```

Listing 3: Memory transfer example

Task 2 Often it is useful to know how long data transfers to and from the GPU take, since if the cost of sending its data outweighs the gain from parallelisation, then there is little value in trying. For Task 2, place timing calls around the `cudaAlloc` and both `cudaMemcpy` functions in Listing 4.


```
#include <stdio.h>

int main()
{
    const unsigned int N = 1048576;
    const unsigned int bytes = N * sizeof(int);
    int *h_a = (int*)malloc(bytes);
    int *d_a;

    cudaMalloc((int**)&d_a, bytes);
    memset(h_a, 0, bytes);
    cudaMemcpy(d_a, h_a, bytes, cudaMemcpyHostToDevice);
    cudaMemcpy(h_a, d_a, bytes, cudaMemcpyDeviceToHost);

    return 0;
}
```


Listing 4: Memory transfer example

5 Matrix Multiplication

Task 3 Matrix multiplication is a staple operation in scientific codes due to its use in solving linear equations. In this task we will consolidate your knowledge of the thread hierarchy, by completing the mapping from threads to rows and columns (marked by the `TODO` comments). You will know if you have done this correctly because the program will no longer print "Validation failed.". Additionally, you should determine whether it is useful, in the context of the code in Listing 5, to offload the compute to the device by timing the CPU version and the GPU version of the code (think carefully about which operations you should include in your timed regions).

```
#include <stdio.h>
#include <omp.h>
```

```
#define BLOCK_SIZE 16

__global__
void mat_mult(float *A, float *B, float *C, int N){
    int row = 0;//TODO
    int col = 0;//TODO

    float sum = 0.0f;
    for (int n = 0; n < N; ++n) {
        sum += A[row*N+n]*B[n*N+col];
    }

    C[row*N+col] = sum;
}

void mat_mult_cpu(float *A, float *B, float *C, int N) {
    #pragma omp parallel for
    for (int row=0; row<N; ++row) {
        for (int col=0; col<N; ++col) {
            float sum = 0.0f;
            for (int n = 0; n < N; ++n){
                sum += A[row*N+n]*B[n*N+col];
            }
            C[row*N+col] = sum;
        }
    }
}

int main(int argc, char *argv[]){

    int N,K;
    K = 100;
    N = K*BLOCK_SIZE;

    float *hA,*hB,*hC;
    hA = new float[N*N];
    hB = new float[N*N];
    hC = new float[N*N];

    for (int j=0; j<N; j++){
        for (int i=0; i<N; i++){
            hA[j*N+i] = 2.f*(j+i);
            hB[j*N+i] = 1.f*(j-i);
        }
    }

    int size = N*N*sizeof(float);
    float *dA,*dB,*dC;
    cudaMalloc(&dA,size);
    cudaMalloc(&dB,size);
    cudaMalloc(&dC,size);

    dim3 threadBlock(BLOCK_SIZE,BLOCK_SIZE);
    dim3 grid(K,K);

    cudaMemcpy(dA,hA,size,cudaMemcpyHostToDevice);
    cudaMemcpy(dB,hB,size,cudaMemcpyHostToDevice);

    mat_mult<<<grid,threadBlock>>>(dA,dB,dC,N);
    if (cudaPeekAtLastError() != cudaSuccess) {
```

```
            fprintf(stderr, "CUDA error detected: \"%s\"\n",
                cudaGetErrorString(cudaGetLastError()));
            return 1;
        }

        float *C;
        C = new float[N*N];
        cudaMemcpy(C,dC,size,cudaMemcpyDeviceToHost);

        mat_mult_cpu(hA, hB, hC, N);

        for (int row=0; row<N; row++) {
            for (int col=0; col<N; col++) {
                if ( C[row*N+col] != hC[row*N+col] ){
                    fprintf(stderr,"Validation failed at row=%d, col=%d
                        .\n", row, col);
                    return 1;
                }
            }
        }
}
```

Listing 5: Matrix multiplication

References

[1] NVIDIA, *CUDA C Programming Guide*, http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf, 2013.

[2] Mark Harris, *An Easy Introduction to CUDA C and C++*, http://devblogs.nvidia.com/parallelforall/easy-introduction-cuda-c-and-c/, 2011.

[3] Mark Harris, *How to Access Global Memory Efficiently in CUDA C/C++ Kernels*, http://devblogs.nvidia.com/parallelforall/how-optimize-data-transfers-cuda-cc/, 2012.

[4] Mark Harris, *How to Optimize Data Transfers in CUDA C/C++*, http://devblogs.nvidia.com/parallelforall/how-access-global-memory-efficiently-cuda-c-kernels/, 2013.

[5] Mark Harris, *How to Overlap Data Transfers in CUDA C/C++*, http://devblogs.nvidia.com/parallelforall/how-overlap-data-transfers-cuda-cc/, 2012.