CMDA 3634: cudaCircuitShell.cu

April 11, 2016

1 Introduction

This document outlines the answers to the questions with written answers in homework 5 for CMDA 3634. This includes questions 1(a-m) and possibly extra credit in this assignment. Code for the homework is listed with comments for each part.

2 Q1a-f

Here in question 1a, we will be changing the reduction sum to proper CUDA code wtih givin instructions. We take the example from the class and create a one-dimensional reduction kernel. As mentioned in the instruction, each thread-block will use **Treduction** and update the circuit loop current one by one. Then these **Treduction** values are summed up using shared memory and at the end those values will be copied to *partEpsilon* for further calculation of epsilon.

```
__global__ void partialReductionKernel(int M, double
  *c_Iold, double *c_Inew, double *c_partEpsilon){

  volatile __shared__ double s_v[Treduction];

  int t = threadIdx.x;
  int b = blockIdx.x;
  int d = blockDim.x;
  int id = t + b*d;
  s_v[t] = 0;

  while(id<M){
    s_v[t] += pow(c_Inew[id] - c_Iold[id], 2);
    id += blockDim.x*gridDim.x;
  }
  __syncthreads();

  if(t<128) s_v[t] += s_v[t+128];
  __syncthreads();
```

```
    if(t< 64) s_v[t] += s_v[t+ 64];
    __syncthreads();

    if(t< 32) s_v[t] += s_v[t+ 32];
    if(t< 16) s_v[t] += s_v[t+ 16];
    if(t<  8) s_v[t] += s_v[t+  8];
    if(t<  4) s_v[t] += s_v[t+  4];
    if(t<  2) s_v[t] += s_v[t+  2];
    if(t<  1) s_v[t] += s_v[t+  1];

    if(t==0)
      c_partEpsilon[b] = s_v[t];

}
```

3 Q1g

With given block dimension and thread-block dimension(*gDim*, *bDim*), now we have the execution configuration set up. We can then call for the *partialReductionKernel* function on GPU and we know that the function is valid configuration for the call to `__global__`.

```
double cudaCalculateEpsilon(int N, double *c_Iold,
    double *c_Inew, double *c_partEpsilon,
    double *h_partEpsilon){

    /* Cells in circuit */
    int  M = (N+2)*(N+2);

    /* Assume a one-dimensional thread array */
    dim3 gDim( (M+Treduction -1)/Treduction);
    dim3 bDim(Treduction);

    /* Q1g: call kernel to partialReductionKernel */
    partialReductionKernel <<< gDim, bDim >>> (M, c_Iold,
    c_Inew, c_partEpsilon);

    /* Q1g: end */

    ...

}
```

4 Q1h

For this part, we use variables i and j to create these linear indices into 2-D form so that we can associate with individual (i,j) cell in circuit loop. I made a mistake by not thinking of the ghost cells in the circuit loop and by adding one for both i and j eliminated the problem.

```
__global__ void cudaIterateKernel(int N, double *c_Iold,
  double *c_Inew){

  /* Q1h: compute i and j using CUDA thread indices, block
  indices, and block dimensions */
  /* Remember: use 1-indexing to match the above for loop
  in the original iterate function */
  int i,j;
  i = threadIdx.x + blockDim.x*blockIdx.x +1;
  j = threadIdx.y + blockDim.y*blockIdx.y +1;

  /* Q1h ends here */

  ...
}
```

5 Q1i

Almost identical to the given interate function, except that we take the doubly nested for loop change that into simple *if-statement*. What GPU will do is assign each thread to calculate to a single grid. Which will execute massively through parallel threading. I had similar issues as inclined in question 1h. I ignored the ghost cells in the circuit loop and had issues with going out of boundary.

```
__global__ void cudaIterateKernel(int N, double *c_Iold,
  double *c_Inew){

  ...

  /* Q1i: each thread updates one single entry of Inew
  using update formula */
  if((i>=1 && i<=N) && (j>=1 && j<=N)){
       c_Inew[c_idx(N,i,j)]
        = 0.25f*(c_Iold[c_idx(N,i+1,j)] +
          c_Iold[c_idx(N,i-1,j)] +
          c_Iold[c_idx(N,i,j+1)] +
          c_Iold[c_idx(N,i,j-1)] +
          (i==1 && j==1) + (i==N && j==N));

  }
  /* Q1i ends here */
}
```

6 Q1j

Same concept as Q1g.

```
void cudaIterate(int N, double *c_Iold, double *c_Inew){

  /* CUDA thread dimensions for two-dimensional array
  of thread-blocks */
  dim3 gDim( (N+Titerate-1)/Titerate, (N+Titerate-1)/Titerate);
  dim3 bDim( Titerate, Titerate);

  /* Q1j: invoke CUDA cudaIterateKernel */
  cudaIterateKernel <<< gDim, bDim >>> (N, c_Iold, c_Inew);
}
```

7 Q1k

cudaMalloc to allocate DEVICE storage for `c_Inew`, `c_Iold`, and `c_partEpsilon` values. Then, use cudaMemset to zero out the DEVICE vectors. Pretty much samething as what calloc does.

```
void cudaSolve(int N, double tol){

  /* use for computed epsilon */
  int N2 = (N+2)*(N+2); // number of cells
  double epsilon;

  int NpartEpsilon = (N2+Treduction-1)/Treduction;

  double *h_Inew = (double*) calloc(N2, sizeof(double));
  double *h_Iold = (double*) calloc(N2, sizeof(double));
  double *h_partEpsilon = (double*) calloc(NpartEpsilon,
                                    sizeof(double));

  double *c_Inew, *c_Iold, *c_partEpsilon;


  /* your Q1k code to build and zero out DEVICE vectors for
  c_Inew, c_Iold, c_partEpsilon starts here */
  cudaMalloc(&c_Inew, N2*sizeof(double));
  cudaMalloc(&c_Iold, N2*sizeof(double));
  cudaMalloc(&c_partEpsilon, NpartEpsilon*sizeof(double));

  cudaMemset(c_Inew, 0, N2*sizeof(double));
  cudaMemset(c_Iold, 0, N2*sizeof(double));
  cudaMemset(c_partEpsilon, 0, NpartEpsilon*sizeof(double));

  /* your Q1k code ends here */
```

```
    ...
}
```

8 Q1l

Here are the loop currents tested for 100x100, 200x200, and 400x400 circuit size with convergence tolerance of 1e-9 for all. Compare to HW01 serial circuit solver, as value of N got higher the time differences were significantly greater compare to the CUDA elapsed time.

100 1e-9:

```
epsilon = 9.99973e-10
I_{11} = 0.302347
I_{10 10} = 0.00317066
elapsed time in seconds =0.396012
```

200 1e-9:

```
epsilon = 9.99954e-10
I_{11} = 0.302347
I_{10 10} = 0.00316795
elapsed time in seconds =2.03644
```

400 1e-9:

```
epsilon = 9.99942e-10
I_{11} = 0.302347
I_{10 10} = 0.00316778
elapsed time in seconds =16.0613
```

9 Q1m

As shown on the time table below, it's obvious that the CUDA code is significantly faster and it's about 2.6 times faster at N = 4000 to a tolerance level of 1e-5. We learned that the performance is the biggest issue when writing CPU and GPU code. Avoiding too much traffic between GPU processor and the HOST is what's most important. Keep the amount of traffic on the processing unit and the memory limited by storing the data that is close to the processing element. Fetch data from DEVICE memory and save it onto shared memory. Shared memory is limited so it is important keep in mind how much we are going to store in it. Every bit of these and the interactions between kernel have great impact on both performance and timing issues.

N	OpenMP Timing	CUDA Timing
10	0.00182504s	0.0037808s
100	0.0185313s	0.0254127s
1000	2.02532s	0.575427s
2000	7.20955s	2.11434s
3000	13.519s	4.70575s
4000	21.3262s	8.23661s

10 EC

Explain what happens when you try to solve a circuit of size N = 4096 or larger using your CUDA code? Circuit size of 4096 x 4096(n=4096) gives no circuit current outputs due to the exceeding the number of memory size. There are total number of registers available per block for each individual GPU, but using 4096 as the size exceeds the maximum dimension of a grid size.

11 References

2016. URL:http://tex.stackexchange.com/questions/42144/how-to-define-macro-that-only-makes-argument-substitution

2016. URL:https://en.wikipedia.org/wiki/C$_data_types$

2016. URL:https://docs.nvidia.com/cuda/samples/6$_Advanced/reduction/doc/reduction.pdf$