Final

By: Alex Kazantsev and Hanumant Mahida

Date: March 25th, 2017

Note: Result summaries from Q1-Q3, then Theory questions are first, followed by the code from Q1-Q3.

Theory Questions

Q1

512 threads per block would yield the most if only 3 blocks are used, as 512*3 = 1536

Q2

CUDA's current architechure has a wrap size of 32 threads, however there is no guarantee that in the future this will continue. Future changes would break the code.

There are some solutions. One could use cudaGetDeviceProperties() to get the wrap size of the GPU and always use __syncthreads() when you have shared-memory dependencies between threads not in the same warp.

One could also write code with 1 warp and not use any __syncthreads() by changing the warp_size -- but this could be limited to the application.

Since __syncthreads() is a block wide barrier, it should be used to avoid shared memory race conditions. Avoiding its use does show significant performance gains, however it does race conditions and double-bufering becomes a worry.

Q3

a) 36 FLOP/ 200 GFLOPS = 1.8e-10 s 7B*4B/100GB = 2.8e-10 s memory bound

b) 36 FLOP/ 300 GFLOPS = 1.2e-10 s 7B*4B/250GB = 1.12e-10 s compute bound

Q4

The final sum will be skewed due to floating point precision error that is an artifact of floating point addition. To correct for precision error the Kahan summation algorithm must be leveraged.

Q5

Code

Programing Question 1 (Trap)

Timing data:

Trap Size	CPU Time (s)	CUDA Time (s)	Speed Improvement
1000000	0.040476	0.000064	632.4375
10000000	0.296	0.000206	1436.8932
100000000	2.331	0.001587	1468.809

Code:

Load data to GPU

```
    float *C_on_device = NULL;

    // allocate space and copy data to device for 1 vectors

    cudaMalloc((void**)&C_on_device, GRID_SIZE*sizeof(float));
    cudaMemset(C_on_device, 0.0, GRID_SIZE*sizeof(float));
```

Setup Execution Config (Block = 1024, Grid = 30) max threads

```
    dim3 dimBlock(BLOCK_SIZE, 1, 1);
    dim3 dimGrid(GRID_SIZE, 1);
```

Launch Device

```
    integrate <<< dimGrid, dimBlock >>> (num_elements, A, B, C_on_device, mutex);
    cudaThreadSynchronize();
```

Kernel

```
    __shared__ float runningSums[BLOCK_SIZE];

    int tx = threadIdx.x;
    int threadID = blockDim.x * blockIdx.x + threadIdx.x;
    int stride = blockDim.x * gridDim.x;

    float local_thread_sum = 0.0;
    volatile float c = 0.0;
    unsigned int i = threadID+1;

    while(i < num_elements-1){
        local_thread_sum += fx(a+ h*i);
        i += stride;
```

```
    }


    runningSums[threadIdx.x] = local_thread_sum;
    __syncthreads();

    c = 0.0;

    for(int stride = blockDim.x/2; stride > 0; stride /= 2){
        if(tx < stride){
            runningSums[tx] += runningSums[tx+stride];
        }
        __syncthreads();
    }

    if(threadIdx.x == 0) {
        lock(mutex);
        result[0] += runningSums[0];
        unlock(mutex);
    }
```

Programming Question 2 (Gauss)

Timing data:

Matrix Size	CPU Time (s)	CUDA Time (s)	Speed Improvement
512	0.085277	0.112194	0.76
1024	0.5376	0.7807	0.6886
2048	3.7838	1.9009	1.9905

Code:

Load data to GPU

```
    int i, j;
    int num_elements = MATRIX_SIZE;
    for (i = 0; i < num_elements; i ++)
        for(j = 0; j < num_elements; j++)
            U.elements[num_elements * i + j] = A.elements[num_elements*i + j];

    Matrix Ud = allocate_matrix_on_gpu(U);
    copy_matrix_to_device(Ud, U);
```

Setup Execution Config (Block = 1024, Grid = 30) max threads

```
    int blockd = ((num_elements <= 1024) ? num_elements: 1024);
    int gridd = int(num_elements/1024)? int(num_elements/1024):1;

    dim3 dimBlock(blockd, 1, 1);
    dim3 dimGrid(gridd, 1);
```

Launch Device

```
    for(int k = 0; k < num_elements; k++)
    {
        gauss_eliminate_kernel <<< dimGrid, dimBlock >>> (Ud.elements, k,
num_elements);
        cudaThreadSynchronize();
    }
```

Kernel

```
    int tid = threadIdx.x + blockIdx.x*blockDim.x;
    if (tid >= k+1){
        U[num_elements * k + tid] = (float)(U[num_elements * k + tid] /
U[num_elements * k + k]);
    }
    if (tid == k)
        U[num_elements*k+k] = 1;


    __syncthreads();

    if (tid >= k+1){
        for(int j = k+1; j < num_elements; j++)
        {
            U[num_elements * tid + j] -= U[num_elements * tid + k] * U[num_elements
* k + j];
        }


        U[num_elements * tid + k] = 0;
    }
```

Programming Question 3 (Compact Stream)

The results for Compact Stream on the GPU showed about 4x-5x *slower* performance than when running compact stream on the CPU.

For example, for 1024 elements CPU time was 0.000015 seconds and GPU time was 0.000050 seconds.

Allocate vectors onto the device

```
cudaMalloc((void**)&result_device, num_elements*sizeof(float));
cudaMemcpy(result_device, result_d, num_elements*sizeof(float),
cudaMemcpyHostToDevice);

cudaMalloc((void**)&h_device, num_elements*sizeof(float));
```

```
cudaMemcpy(h_device, h_data, num_elements*sizeof(float), cudaMemcpyHostToDevice);
```

Thread Block and Grid inits

```
dim3 threads(TILE_SIZE, TILE_SIZE);
dim3 grid(num_elements);
```

Execute Kernel and Sync Threads

```
compact_stream_kernel<<<grid, threads>>>(result_device, h_device, num_elements, n);
cudaThreadSynchronize();
```

Kernel

```
for (unsigned int i = 0; i < len; i++) {
    if (idata[idx] > 0.0) {
        //reference[n++] = idata[idx];
        *n++;
}
```