Lab 5

George Onwubuya

November 6, 2018

1 Reduction Sum

1.1 Output

```
Setting up the problem...0.000019 s
    Input size = 1000
Allocating device variables...0.203124 s
Copying data from host to device...0.000065 s
Launching kernel...0.000103 s
Copying data from device to host...0.000052 s
Verifying results...TEST PASSED


Setting up the problem...0.000057 s
    Input size = 2000
Allocating device variables...0.176309 s
Copying data from host to device...0.000059 s
Launching kernel...0.000123 s
Copying data from device to host...0.000035 s
Verifying results...TEST PASSED


Setting up the problem...0.000053 s
    Input size = 4000
Allocating device variables...0.179124 s
Copying data from host to device...0.000068 s
Launching kernel...0.000105 s
Copying data from device to host...0.000026 s
Verifying results...TEST PASSED


Setting up the problem...0.000095 s
    Input size = 8000
Allocating device variables...0.177338 s
```

```
Copying data from host to device...0.000086 s
Launching kernel...0.000115 s
Copying data from device to host...0.000026 s
Verifying results...TEST PASSED


Setting up the problem...0.000184 s
    Input size = 16000
Allocating device variables...0.140137 s
Copying data from host to device...0.000087 s
Launching kernel...0.000099 s
Copying data from device to host...0.000023 s
Verifying results...TEST PASSED


Setting up the problem...0.000385 s
    Input size = 32000
Allocating device variables...0.155782 s
Copying data from host to device...0.000166 s
Launching kernel...0.000108 s
Copying data from device to host...0.000026 s
Verifying results...TEST PASSED


Setting up the problem...0.000803 s
    Input size = 64000
Allocating device variables...0.141736 s
Copying data from host to device...0.000190 s
Launching kernel...0.000108 s
Copying data from device to host...0.000026 s
Verifying results...TEST PASSED


Setting up the problem...0.001595 s
    Input size = 128000
Allocating device variables...0.139249 s
Copying data from host to device...0.000328 s
Launching kernel...0.000107 s
Copying data from device to host...0.000036 s
Verifying results...TEST PASSED


Setting up the problem...0.010696 s
    Input size = 1000000
Allocating device variables...0.158959 s
Copying data from host to device...0.002293 s
```

```
Launching kernel...0.000158 s
Copying data from device to host...0.000028 s
Verifying results...TEST PASSED


Setting up the problem...0.019773 s
    Input size = 2000000
Allocating device variables...0.187990 s
Copying data from host to device...0.004141 s
Launching kernel...0.000205 s
Copying data from device to host...0.000039 s
Verifying results...TEST PASSED


Setting up the problem...0.038021 s
    Input size = 4000000
Allocating device variables...0.181643 s
Copying data from host to device...0.007546 s
Launching kernel...0.000308 s
Copying data from device to host...0.000046 s
Verifying results...TEST PASSED
```

1.2 Performance Analysis

1.2.1 Array Size

Execution Time (seconds) for Each Process					
Elements(m)	Setting Up	DeviceVar	HostToDevice	Kernel	DeviceToHost
1000	0.000019	0.203124	0.000065	0.000103	0.000052
2000	0.000057	0.176309	0.000059	0.000123	0.000035
4000	0.000053	0.179124	0.000068	0.000105	0.000026
8000	0.000095	0.177338	0.000086	0.000115	0.000026
16000	0.000184	0.140137	0.000087	0.000099	0.000023
32000	0.000385	0.155782	0.000166	0.000108	0.000026
64000	0.000803	0.141736	0.000190	0.000108	0.000026
128000	0.001595	0.139249	0.000328	0.000107	0.000036
1000000	0.010696	0.158959	0.002293	0.000158	0.000028
2000000	0.019773	0.187990	0.004141	0.000205	0.000039
4000000	0.038021	0.181643	0.007546	0.000308	0.000046

1.2.2 Comments

The different execution times relate to the number of elements in different ways. The execution times for allocating device variables generally are similar because the same device variables will be allocated regardless of the size of the array.

The execution times for setting up the problem and copying data from host to device are directly proportional to the number of elements. I assumed that the time taken to launch the kernel would follow the same trend but the results do not show this trend. This maybe due to the fact that a noticeable difference can be observed with large element array sizes only. The time it takes to copy data from the device to the host is generally the same because we one value is copied to host.

1.3 Answers

1.3.1 a

A single thread block will synchronize about $\log_2(BlockSize)$.

1.3.2 b

Every thread should minimally perform one operation and that is the loading of the elements into shared memory. The maximum number of 'real' operations would $1 + \log_2(BlockSize)$. The average number of 'real' operations would be $(1 + \log_2(BlockSize))/BlockSize$

1.4 Kernel

```
/***************************************************************************
 *cr
 *cr            (C) Copyright 2010 The Board of Trustees of the
 *cr                        University of Illinois
 *cr                         All Rights Reserved
 *cr
 ***************************************************************************/

#define BLOCK_SIZE 512
//#define SIMPLE

__global__ void reduction(float *out, float *in, unsigned size)
{
    /*********************************************************************
    Load a segment of the input vector into shared memory
    Traverse the reduction tree
    Write the computed sum to the output vector at the correct index
    *********************************************************************/


```

```
#ifdef SIMPLE
    __shared__ float in_s[2*BLOCK_SIZE];
    int idx = 2 * blockIdx.x * blockDim.x + threadIdx.x;

    in_s[threadIdx.x]            = ((idx              < size)? in[idx]:            0.0f);
    in_s[threadIdx.x+BLOCK_SIZE] = ((idx + BLOCK_SIZE < size)? in[idx+BLOCK_SIZE]: 0.0f);

    for(int stride = 1; stride < BLOCK_SIZE<<1; stride <<= 1) {
        __syncthreads();
        if(threadIdx.x % stride == 0)
                in_s[2*threadIdx.x] += in_s[2*threadIdx.x + stride];
    }

#else

    // INSERT KERNEL CODE HERE
    __shared__ float in_s[BLOCK_SIZE];
    int idx =2*blockIdx.x * blockDim.x + threadIdx.x;

    in_s[threadIdx.x] = ((idx < size) ? in[idx] : 0.0f) + ((idx + BLOCK_SIZE < size) ? in[idx + BLOCK_SIZE]: 0.0f);

    for(int stride = BLOCK_SIZE/2; stride >= 1; stride >>= 1){

    __syncthreads();

        if(threadIdx.x < stride)

                in_s[threadIdx.x] += in_s[threadIdx.x + stride];
    }




#endif

    if(threadIdx.x == 0)
        out[blockIdx.x] = in_s[0];
}
```

2 Prefix Scan

2.1 Output

```
Setting up the problem...0.000021 s
    Input size = 1000
Allocating device variables...0.160909 s
Copying data from host to device...0.000054 s
Launching kernel...0.000112 s
Copying data from device to host...0.000029 s
Verifying results...TEST PASSED


Setting up the problem...0.000032 s
    Input size = 2000
Allocating device variables...0.182564 s
Copying data from host to device...0.000061 s
Launching kernel...0.000342 s
Copying data from device to host...0.000037 s
Verifying results...TEST PASSED


Setting up the problem...0.000062 s
    Input size = 4000
Allocating device variables...0.179288 s
Copying data from host to device...0.000069 s
Launching kernel...0.000363 s
Copying data from device to host...0.000040 s
Verifying results...TEST PASSED


Setting up the problem...0.000132 s
    Input size = 8000
Allocating device variables...0.139090 s
Copying data from host to device...0.000067 s
Launching kernel...0.000324 s
Copying data from device to host...0.000045 s
Verifying results...TEST PASSED


Setting up the problem...0.000205 s
    Input size = 16000
Allocating device variables...0.163738 s
Copying data from host to device...0.000106 s
Launching kernel...0.000346 s
```

```
Copying data from device to host...0.000056 s
Verifying results...TEST PASSED


Setting up the problem...0.000459 s
    Input size = 32000
Allocating device variables...0.154225 s
Copying data from host to device...0.000128 s
Launching kernel...0.000349 s
Copying data from device to host...0.000154 s
Verifying results...TEST PASSED


Setting up the problem...0.000772 s
    Input size = 64000
Allocating device variables...0.137835 s
Copying data from host to device...0.000227 s
Launching kernel...0.000356 s
Copying data from device to host...0.000261 s
Verifying results...TEST PASSED


Setting up the problem...0.001514 s
    Input size = 128000
Allocating device variables...0.140842 s
Copying data from host to device...0.000374 s
Launching kernel...0.000365 s
Copying data from device to host...0.000512 s
Verifying results...TEST PASSED


Setting up the problem...0.002899 s
    Input size = 256000
Allocating device variables...0.141219 s
Copying data from host to device...0.000667 s
Launching kernel...0.000377 s
Copying data from device to host...0.001167 s
Verifying results...TEST PASSED


Setting up the problem...0.010954 s
    Input size = 1000000
Allocating device variables...0.140619 s
Copying data from host to device...0.002301 s
Launching kernel...0.000527 s
Copying data from device to host...0.002632 s
```

```
Verifying results...TEST PASSED


Setting up the problem...0.020591 s
    Input size = 2000000
Allocating device variables...0.156192 s
Copying data from host to device...0.004212 s
Launching kernel...0.000766 s
Copying data from device to host...0.005400 s
Verifying results...TEST PASSED
```

2.2 Performance Analysis

2.2.1 Array Size

Execution Time (seconds) for Each Process					
Elements(m)	Setting Up	DeviceVar	HostToDevice	Kernel	DeviceToHost
1000	0.000021	0.160909	0.000054	0.000112	0.000029
2000	0.000032	0.182564	0.000061	0.000342	0.000037
4000	0.000062	0.179288	0.000069	0.000363	0.000040
8000	0.000132	0.139090	0.000067	0.000324	0.000045
16000	0.000205	0.163738	0.000106	0.000346	0.000056
32000	0.000459	0.154225	0.000128	0.000349	0.000154
64000	0.000772	0.137835	0.000227	0.000356	0.000261
128000	0.001514	0.140842	0.000374	0.000365	0.000512
256000	0.002899	0.141219	0.000667	0.000377	0.001167
1000000	0.010954	0.140619	0.002301	0.000527	0.002632
2000000	0.020591	0.156192	0.004212	0.000766	0.005400

2.2.2 Comments

The different execution times relate to the number of elements in different ways. The execution times for allocating device variables generally are similar because the same device variables will be allocated regardless of the size of the array. The execution times for setting up the problem, copying data from host to device and vice-versa are directly proportional to the number of elements. I assumed that the time taken to launch the kernel would follow the same trend but the results do not support this assumption. Maybe it has to do with the fact that a noticeable difference in launch time can only be observed with large elements. The code 'fails' after two million elements because of the floating point limitations that produce inaccurate results which exceed the relative error.

2.3 Answers

2.3.1 a

In the code, a thread block or a block size is defined as 512 which is a multiple of 2. There is a check that ensures when the global index exceeds the size of the input array, the remaining threads in the thread block load zeroes. To improve the speed up performance of the code the input elements were loaded into shared memory which is faster than global memory. Mathematical operations found in the up sweep and down sweep portions of the kernel such as multiplying or dividing by two were defined using binary shift which is faster than its arithmetic counterpart. To improve the efficient use of the memory banks, a memory bank offset was calculated and added to the different points in the shared memory in order to avoid memory banking conflicts.

2.4 Kernel

```
/********************************************************************************
 *cr
 *cr            (C) Copyright 2010 The Board of Trustees of the
 *cr                        University of Illinois
 *cr                         All Rights Reserved
 *cr
 ********************************************************************************/

#define BLOCK_SIZE 512

#define NUM_BANKS 32
#define LOG_NUM_BANKS 5

#ifdef ZERO_BANK_CONFLICTS
#define CONFLICT_FREE_OFFSET(n) ((n) >> NUM_BANKS + (n) >> (2 *
    LOG_NUM_BANKS))
#else
#define CONFLICT_FREE_OFFSET(n) ((n) >> LOG_NUM_BANKS)
#endif



// Define your kernels in this file you may use more than one
    kernel if you
// need to

// INSERT KERNEL(S) HERE

```

```
__global__ void preScanKernel(float *out, float *in, unsigned size, float *sum){
    // INSERT CODE HERE
        __shared__ float a_s[(2 * BLOCK_SIZE) + CONFLICT_FREE_OFFSET(2 * BLOCK_SIZE)];
        int idx = 2 * blockIdx.x * blockDim.x + threadIdx.x;

        int thid = threadIdx.x;
        thid += CONFLICT_FREE_OFFSET(thid);
        int thid_BS = threadIdx.x + BLOCK_SIZE;
        thid_BS += CONFLICT_FREE_OFFSET(thid_BS);

        a_s[thid]    = ((idx              < size)? in[idx]: 0.0f);
        a_s[thid_BS] = ((idx + BLOCK_SIZE < size)? in[idx+BLOCK_SIZE]: 0.0f);


        unsigned int ai, bi;
        unsigned int numThreads, stride;

        for(numThreads = BLOCK_SIZE, stride = 1; numThreads > 0; numThreads >>= 1, stride <<= 1){

                ai = (2 * threadIdx.x * stride + stride - 1);
                bi = (2 * threadIdx.x * stride + 2 * stride - 1);

                ai += CONFLICT_FREE_OFFSET(ai);
                bi += CONFLICT_FREE_OFFSET(bi);

        __syncthreads();

                if(threadIdx.x < numThreads)
                        a_s[bi] += a_s[ai];
        }

        if(threadIdx.x == 0){
                int last_elem = 2 * BLOCK_SIZE - 1;
                last_elem += CONFLICT_FREE_OFFSET(last_elem);
                if(sum != NULL){
                        sum[blockIdx.x] = a_s[last_elem];
                }
                a_s[last_elem] = 0;
        }


```

```
        for(numThreads = 1, stride = BLOCK_SIZE; numThreads <= BLOCK_SIZE; numThreads <<= 1, stride >>= 1){

                ai = (2 * threadIdx.x * stride + stride - 1);
                bi = (2 * threadIdx.x * stride + 2 * stride - 1);

                ai += CONFLICT_FREE_OFFSET(ai);
                bi += CONFLICT_FREE_OFFSET(bi);

                __syncthreads();

                if(threadIdx.x < numThreads){
                float temp = a_s[bi];
                a_s[bi] += a_s[ai];
                a_s[ai] = temp;
                }
                __syncthreads();
        }
        if(idx < size)
        out[idx] = a_s[thid];

        if(idx + BLOCK_SIZE < size)
        out[idx + BLOCK_SIZE] = a_s[thid_BS];

}


__global__ void addKernel(float *out, float *sum, unsigned size)
{
    // INSERT CODE HERE
        int idx = 2 * blockIdx.x * blockDim.x + threadIdx.x;

        if(idx < size)
        out[idx] += sum[blockIdx.x];

        if(idx + BLOCK_SIZE < size)
        out[idx + BLOCK_SIZE] += sum[blockIdx.x];

}

/******************************************************************************
Setup and invoke your kernel(s) in this function. You may also allocate more
GPU memory if you need to
*******************************************************************************/
void preScan(float *out, float *in, unsigned in_size)
```

```
{
	float *sum;
	unsigned num_blocks;
	cudaError_t cuda_ret;
	dim3 dim_grid, dim_block;

	num_blocks = in_size/(BLOCK_SIZE*2);
	if(in_size%(BLOCK_SIZE*2) !=0) num_blocks++;

	dim_block.x = BLOCK_SIZE; dim_block.y = 1; dim_block.z = 1;
	dim_grid.x = num_blocks; dim_grid.y = 1; dim_grid.z = 1;

	if(num_blocks > 1) {
		cuda_ret = cudaMalloc((void**)&sum, num_blocks*sizeof(float));
		if(cuda_ret != cudaSuccess) FATAL("Unable to allocate device memory");

		preScanKernel<<<dim_grid, dim_block>>>(out, in, in_size, sum);
		preScan(sum, sum, num_blocks);
		addKernel<<<dim_grid, dim_block>>>(out, sum, in_size);

		cudaFree(sum);
	}
	else
		preScanKernel<<<dim_grid, dim_block>>>(out, in, in_size, NULL);
}
```