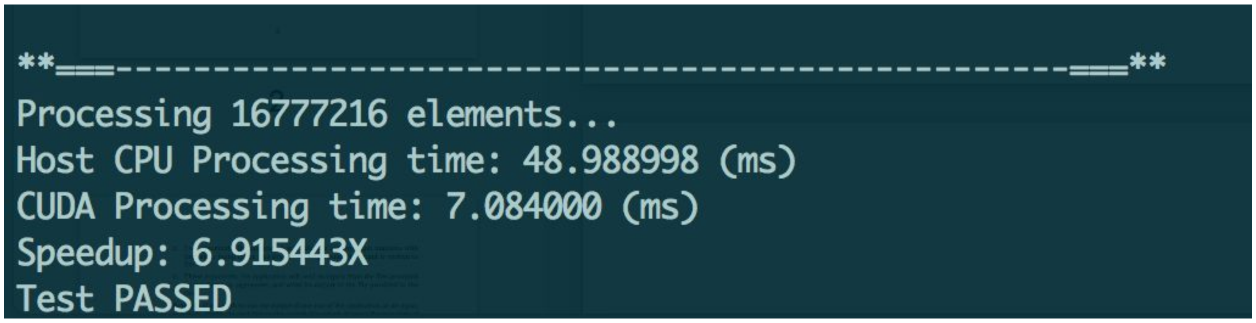
CUDA Lab4 Report

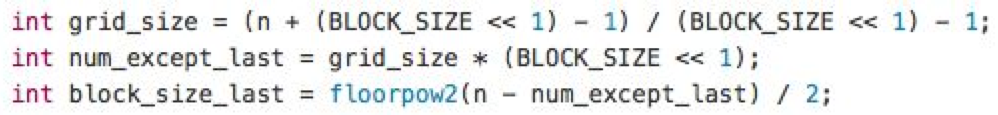
Qinglin Li
Xiangyu Ji

1. Below is our screen cur for running with DEFAULT_NUM_ELEMENT set to 16777216 and MAX_RAND to 3



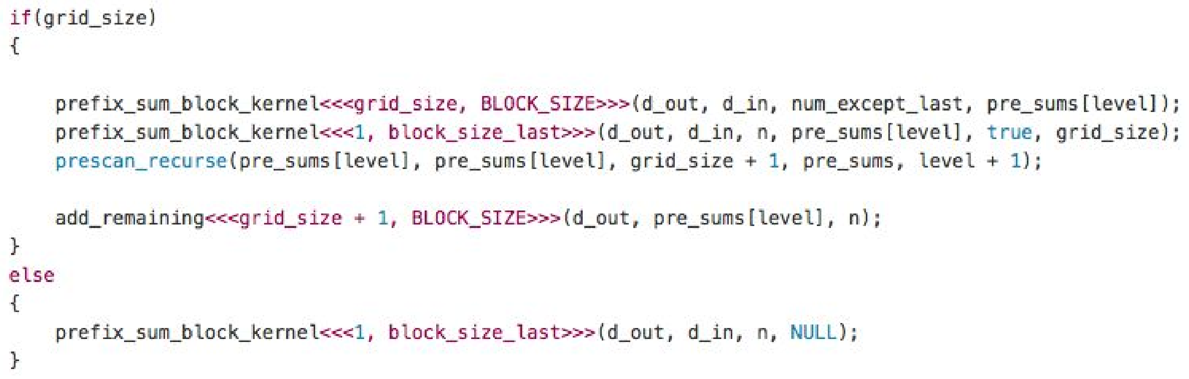
```
**===------------------------------------------------------------===**
Processing 16777216 elements...
Host CPU Processing time: 48.988998 (ms)
CUDA Processing time: 7.084000 (ms)
Speedup: 6.915443X
Test PASSED
```

2. To handle number of array elements that is not a power of two, we divide the large array into blocks that can be scanned by a single thread block.



```
int grid_size = (n + (BLOCK_SIZE << 1) - 1) / (BLOCK_SIZE << 1) - 1;
int num_except_last = grid_size * (BLOCK_SIZE << 1);
int block_size_last = floorpow2(n - num_except_last) / 2;
```

To deal with array elements number which is not the power of 2, as shown in the code shot, we set the BLOCK_SIZE to be 1024, with the number of element in the array, we first calculate the number of grid we need to use, then we calculate num_except_last, which is the maximum number of elements in the array which is the power of two, and the remaining part with is not the power of 2 array inside one block, for this block containing non-power of 2 element, we padding the shared memory use the next higher power of 2 number and fill extra memory with 0 when loading data from global memory to shared memory. Code shot can be seen below.



```
if(grid_size)
{

    prefix_sum_block_kernel<<<grid_size, BLOCK_SIZE>>>(d_out, d_in, num_except_last, pre_sums[level]);
    prefix_sum_block_kernel<<<1, block_size_last>>>(d_out, d_in, n, pre_sums[level], true, grid_size);
    prescan_recurse(pre_sums[level], pre_sums[level], grid_size + 1, pre_sums, level + 1);

    add_remaining<<<grid_size + 1, BLOCK_SIZE>>>(d_out, pre_sums[level], n);
}
else
{
    prefix_sum_block_kernel<<<1, block_size_last>>>(d_out, d_in, n, NULL);
}
```

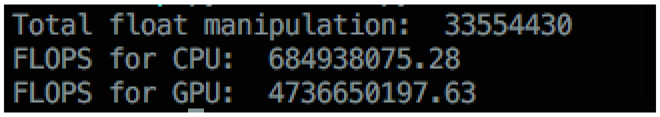
The function of prefix_sum calling is in the recursive way, in our implementation, for every recursive step, we treat the last block specially and decide whether remaining number of elements in the last block is the power of 2 to decide whether to padding memory with filling 0.

To minimize the bank conflicts, we use 2 type of implementation to achieve that. First when loading data from global to shared memory, for each thread we read 2 element that interval with blockDim.x rather than 2 neighboring element, that minimize the loading bank conflicts.

Next, as guided by Mark Harris's report, we add the index the value of the index divided by the number of shared memory banks each time we manipulate with shared memory elements. We add padding to the shared memory every NUM_BANKS elements, in our implementation, we set NUM_BANKS to be 32.

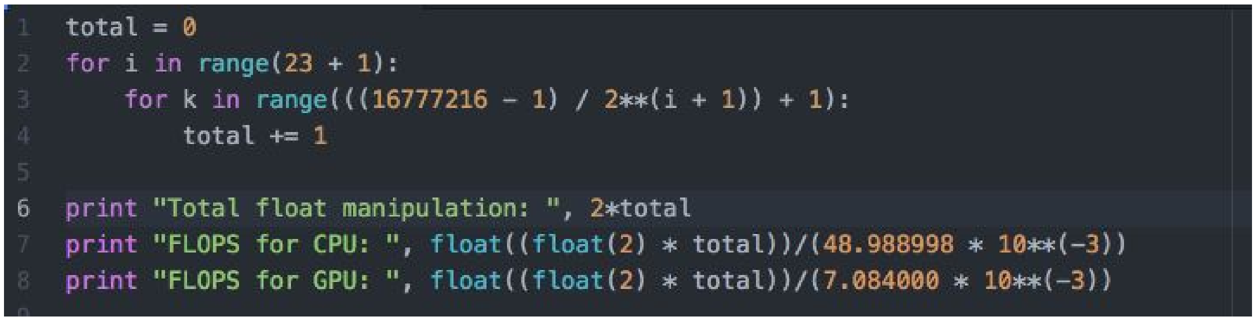
3. For the comparison of FLOPS:
In this parallel prefix sum problem, we implement the work-efficient scan, so:
In the top-down order, there are total :
For d from 0 to logn - 1, every d has (n-1) / 2**(d+1) times float add,
In the down-top order, there are total:
For d from logn - 1 to 0, every d has (n-1) / 2**(d+1) times float add,

The FLOPS for our implementation result with 16777216 NUM_ELEMENTS are:



```
Total float manipulation:  33554430
FLOPS for CPU:  684938075.28
FLOPS for GPU:  4736650197.63
```

Attached is the Python code for calculation:



```
total = 0
for i in range(23 + 1):
    for k in range(((16777216 - 1) / 2**(i + 1)) + 1):
        total += 1

print "Total float manipulation: ", 2*total
print "FLOPS for CPU: ", float((float(2) * total))/(48.988998 * 10**(-3))
print "FLOPS for GPU: ", float((float(2) * total))/(7.084000 * 10**(-3))
```

Theoretical peak FLOPS for GTX 680 in Wikinson Lab is 3.1 x 10**12.
Compare with the theoretical peak, the FLOPS of our implementation is far below the peak. The calculation reason could be our record time involves time to loading data and creating padding. The main bottleneck for the implementation could be with the Synchronize of the thread in order to remove data race.