EECS-4750 Assignment 4

Scan

Xinghua Sun - xs2445

Programming Problems

1. Naive serial implementation

A really naive serial implementation can be shown as the following figure.

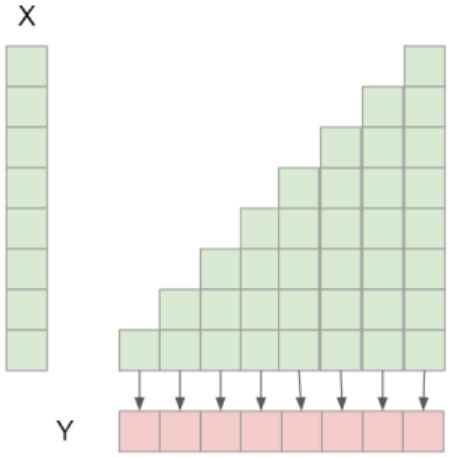

Figure 1.1 a naive serial prefix-sum (yi = x0+...+xi)

Time complexity: 1 + 2 + 3 + … + length = (1+length)length/2, which is O(n^2)
Space complexity: O(n)

Another better serial implementation can be:

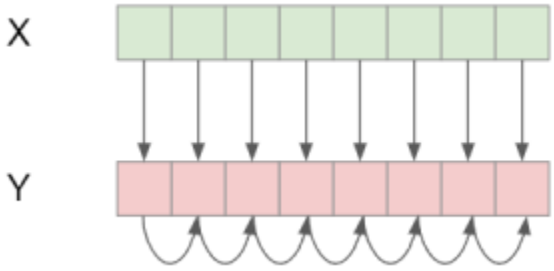

Figure 1.2 a better serial prefix-sum (yi = y(i-1)+xi)

Time complexity: 1 + 1 + 1 + … + 1 = length, which is O(n)
Space complexity: O(n)

2. Work inefficient parallel scan algorithm

The work inefficient gpu scan algorithm refers to the Koggo-Stone design as the figure below shows.

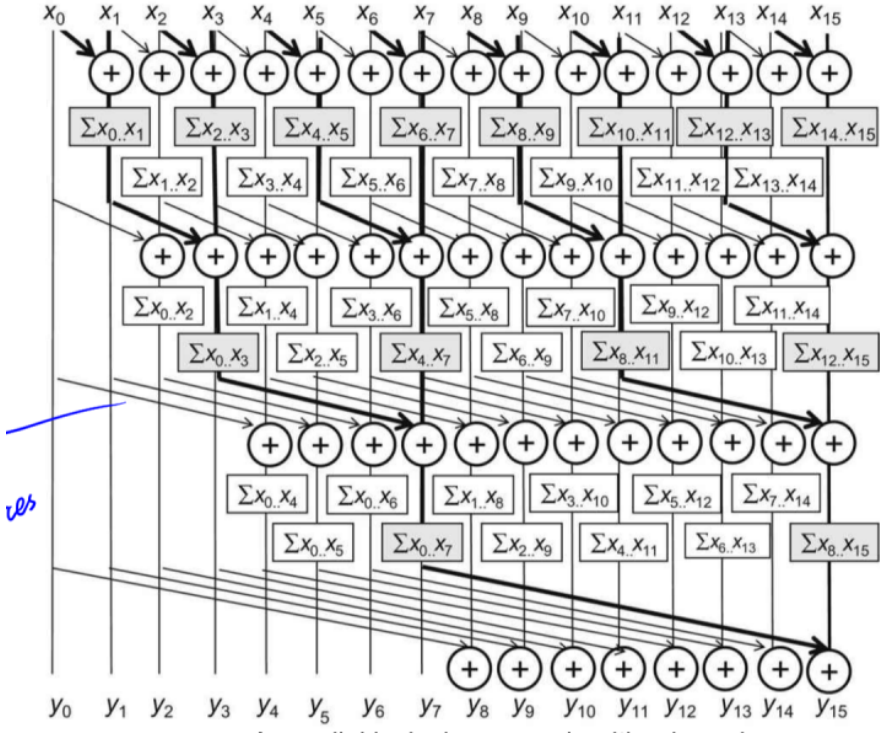

Figure 1.3 A parallel inclusive scan algorithm based on Kogge-Stone adder design

Time complexity: O(log(n))
Space complexity: O(n)

3. Work efficient parallel scan algorithm

In reality, the amount of work done by the inefficient parallel scan algorithm can be more than the theoretical number. Because many of the threads stop participating in the execution of the for-loop but still consume execution resources.
A better and work efficient parallel scan algorithm can be the Brent-Kung design as the figure shows.
If each thread processes one element of the result array, then there is always less than half the input array length of the threads calculating, so we can copy twice the size of the block dimension of the input array into the shared memory.

Time complexity: O(log(n))
Space complexity: O(n)

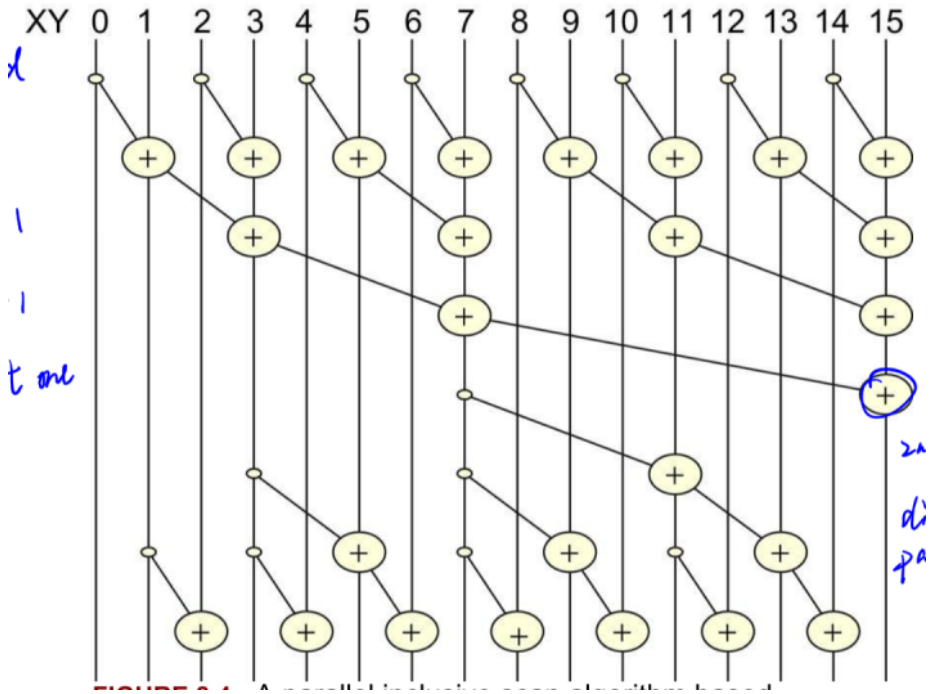

Figure 1.4 A parallel inclusive scan algorithm based on Kogge-Stone adder design

4. Prefix scan for arbitrary-length inputs

Because all the above parallel scans are implemented in a single block, so for an arbitrary-length input array we need to figure out how to scan if the length is larger than the limited maximum block size.
The maximum block size and grid size of the GPU device (Tesla T4) is 1024.

Some properties of Tesla T4:
MAX_BLOCK_DIM_X:1024
MAX_BLOCK_DIM_Y:1024
MAX_BLOCK_DIM_Z:64
MAX_GRID_DIM_X:2147483647
MAX_GRID_DIM_Y:65535
MAX_GRID_DIM_Z:65535
MAX_PITCH:2147483647

A hierarchical scan for arbitrary length inputs is shown below:

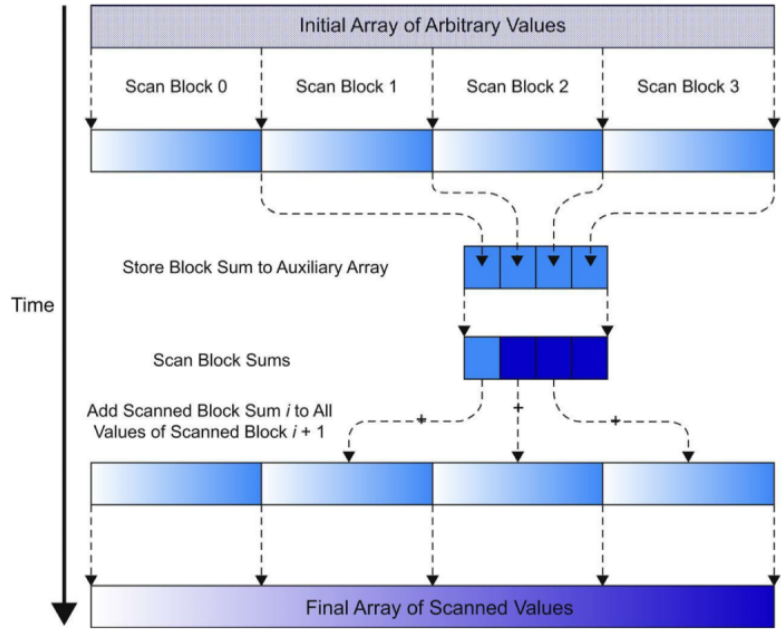

Figure 1.5 A parallel inclusive scan algorithm based on Kogge-Stone adder design

There are two phases, the figure shows 1 iteration of those 2 phases. In practice, there will be log(length, 1024(or 2048 for efficient scan)) iterations.

5. Results

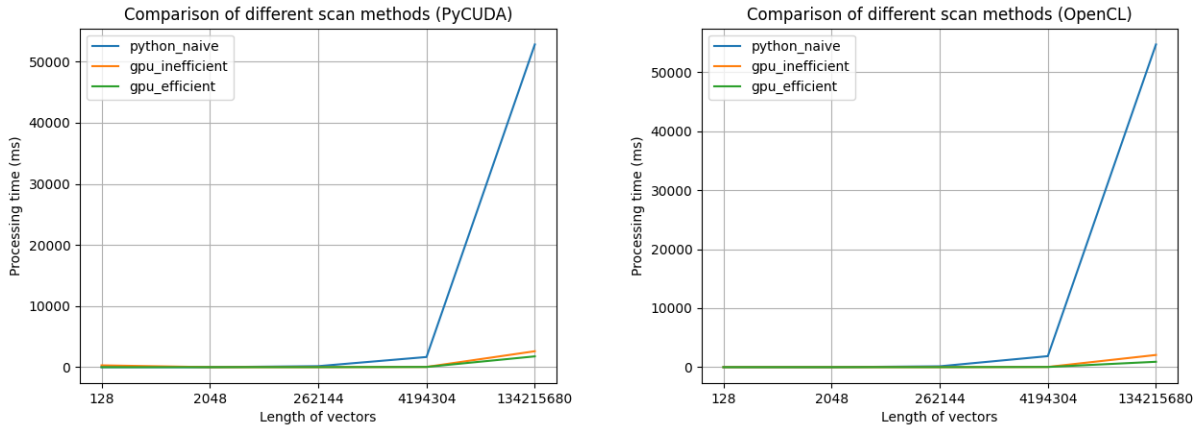

Figure 1.6 Comparison of different prefix-scan methods in PyCUDA and PyOpenCL

From figure 1.6 we can see that the parallel scan have a huge advantage compared to serial implementation.
Also, because of the reason we discussed, the work efficient scan have a better performance than work inefficient scan.

Conceptual Problem - Stream Compaction

Consider the application of scan called Stream Compaction. Sketch and describe the flow of stream compaction. Write a pseudo code for an OpenCL kernel which implements stream compaction.

Stream compaction is a common parallel primitive used to remove unwanted elements in sparse data. The main steps of performing compaction can be shown in the following figure.

Figure 2.1 Flow of stream compaction

We can see that the length of the result array is dynamic, so we need to count how many wanted elements are in the input array. So as the figure shows, 2 phases are designed for this task. The first phase, “Valid Element Flags” and “Exclusive Prefix Sum”, counts the number of wanted elements. Then the second phase “Scatter Valid” makes a new array and copies every wanted element into it.
Also the result of prefix sum is the addresses offset of wanted elements in the result array.

The OpenCL kernel:

```
__kernel void SCphase1(__global *input, __global *output, __global *lookuptable, const int length, const int length_table){

    /*
    Get the index of blocks and threads and global position.
    */
    tx=,,,;
```

```
        bx=...;
        global_id=...;

        // valid element flags
        Int count = 0.0f
        For(i=0; i<length_table; i++){
                If(input[global_id] == lookuptable[i]) {count++; break;}

        // prefix sum
        offset = prefix-sum


        // scatter wanted elements
        if(count==1) output[j] = input[global_id];

}
```

Theory Problems

1. Consider that NVIDIA GPUs execute warps of 32 parallel threads using SIMT. What's the difference between SIMD and SIMT? What is the worst choice as the number of threads per block to chose in this case among the following and why?

(A) 1

(B) 16

(C) 32

(D) 64

SIMT(Single-Instruction, Multiple-Thread) is very similar to SIMD(Single-Instruction, Multiple-Data).

In SIMD, multiple data can be processed by a single instruction, all the processors receive the same instruction from the control unit. In SIMT, multiple threads are processed by a single instruction in lock-step.

The key difference between SIMT and SIMD lanes is that each of the SIMT cores may have a completely different Stack Pointer (and thus perform computations on completely different data sets), whereas SIMD lanes simply perform computations on the same data sets. The main advantage of SIMT is that it reduces the latency that comes with instruction prefetching.

So only 1 thread per block is the worst choice because the more threads in a block makes it more efficient.

2. What is a bank conflict? Give an example for bank conflict.

Before we look into bank conflict, we need to know what a bank is.

In modern designs, DRAM systems typically employ banks and channels as form of organization. At a high level, a processor contains one or more channels and each channel is a memory controller with a bus that connects a set of DRAM banks to the processor.

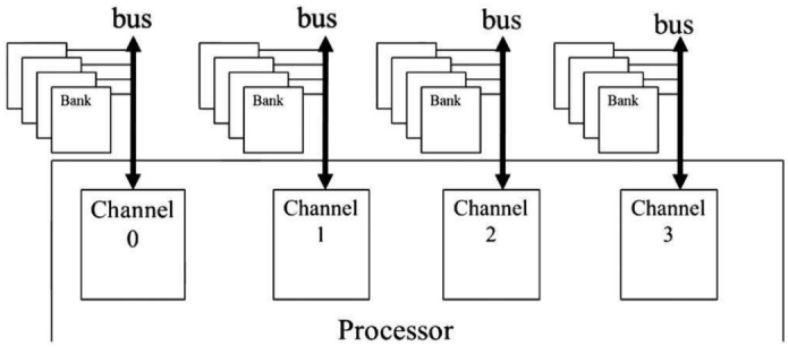

Figure 3.1 Channels and banks in DRAM systems

Modern double data rate (DDR) busses perform two data transfers per clock cycle, one at the rising edge and one at the falling edge.

For each channel, the number of banks connected to it is determined by the number of banks required to fully utilize the data transfer bandwidth of the bus. Like the figure below shows.

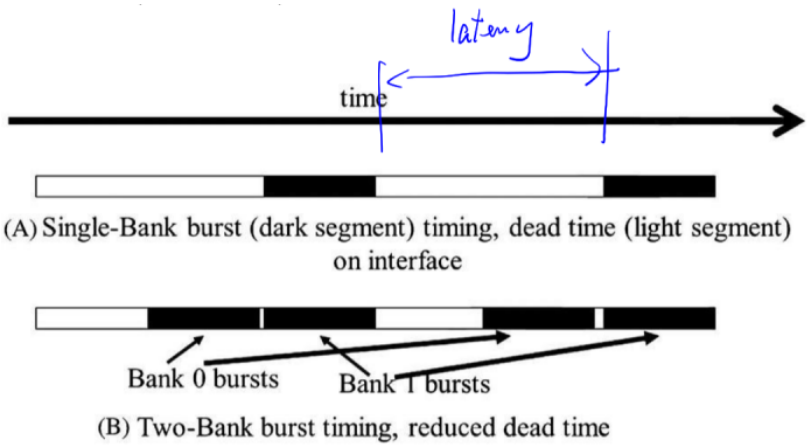

Figure 3.2 Banking improves the utilization of data transfer bandwidth of a channel

The access latency is much longer than the data transfer time. Connecting multiple banks to a channel bus to overlap the accessing latency that improves the bandwidth of a channel.

However, having more banks reduces the probability of multiple simultaneous accesses targeting the same bank. That is a phenomenon called bank conflict. Since each bank can serve only one access at a time, the cell array access latency can no longer be overlapped for these conflicting access.

For example, if we want to access two addresses in the shared memory, and those two addresses occur in the same bank, then the bank needs to access the addresses one by one and lose the advantage of parallel access, which is also known as burst.

Having a larger number of banks increases the probability that these accesses will be spread out among multiple banks.

3. Conisder the following code for finding the sum of all elements in a vector. The following code doesn't always work correctly explain why? Also suggest how to fix this? (Hint: use atomics.)

```
__global__ void vectSum(int* d_vect,size_t size, int* result){
    size_t tid = blockIdx.x * blockDim.x + threadIdx.x;
    while(tid < size){
        *result+=d_vect[tid];
        tid+=blockDim.x * gridDim.x;
    }
}
```

Race-condition could happen that multiple threads are trying to add their d_vect to result at the same time, which may lead to missing element addition in results. The execution should be sequential to avoid any conflict.

To fix the code, we can use atomicAdd() instead:

From :

*result += d_vect[tid];

To:

AtomicAdd(*result, d_vect[tid]);

4. Is there a way to dynamically allocate shared memory? Explain how?

Yes.
In PyCUDA, we need to add “extern” to the shared memory code:

extern __shared__ float shared_data[];

Then we need to specify the length of the shared memory assigned when we invoke the kernel, for example:

function(Input_0, Input_1, block=(100,1,1), grid=(1,1,1,), shared=50)