Multi-core Programming
Assignment 6

Ali Gholami

Department of Computer Engineering & Information Technology
Amirkabir University of Technology

https://aligholamee.github.io
aligholami7596@gmail.com

Abstract

The main purpose of this report is to analyze techniques to optimize the performance of a CUDA Kernel for *reduction* task. Reduction is to find the sum of elements of an input array. We'll start with a naive implementation and we improve the performance in latter kernels. This can be done by removing bank conflict of warps while accessing the shared memory or removing the branch divergence by purposing new perspectives of thinking about conditional statements.

Keywords. *Reduction Strategies, Bank Conflict, Idle Threads, Loop Unrolling, Branch Divergence, Performance Optimization.*

1 Memory Bandwidth Analysis

Reduction is a memory bounded task. Since the operational intensity is limited to 1 (we'll improve that in upcoming kernels), the memory bandwidth needs to be as high as possible to hide the effect of slow data access on performance. In this section, we'll compute the memory bandwidth for a *Nvidia 850m GTX*. According to the information extracted by *GPU-Z*, memory has a 128 bit interface with a clock of 1001 Mhz which yields in a bandwidth of 32 GB/s which is the peak bandwidth of this GPU.

2 A Naive Reduction

As an introduction to the reduction strategies, we'll provide a naive implementation of the reduction technique. This implementation suffers from:

- Branch Divergence

at its core logic. Note that this implementation simply does a *tree-based* reduction based on the copied input on to the shared memory. If the resulting array's size is greater than what a block can handle, it launches another kernel to compute the final results. The **problem** with this logic is that, 1 out of 2 threads will be diverged in the following *conditional-check*:

```
// Tree based reduction
for (unsigned int d = 1; d < blockDim.x; d *= 2) {
        if (tId % (2 * d) == 0)
                if (tId + d < blockDim.x)
                        sData[tId] += sData[tId + d];

        __syncthreads();
}
```

Removed Branch Divergence

The code in the previous section can be improved by doing an *index-based* condition as follows:

```
	// Tree based reduction
	for (unsigned int d = 1; d < blockDim.x; d *= 2) {
		int index = 2 * d * tId;

		if (index < blockDim.x)
			if (index + d < blockDim.x)
				sData[index] += sData[index + d];

		__syncthreads();
	}
```

This improvement introduces a new problem; *Shared Memory Bank Conflicts*. Data in shared memory is stored at *words*. Each word is a 4 byte data type that is assigned to a shared memory bank of the same index. If we assume that there are total number of 32 banks of shared memory in each thread block, words 0 to 31 are assigned to banks 0 to 31. Respectively, words 32 to 63 are assigned to banks 0 to 31 an so on. If threads of a single warp request banks with the *same index*, their request has to be processed *sequentially*. This can be a great bottleneck which results in a *memory-bound* kernel.

Removed Bank Conflicts

This problem can be solved using *sequential addressing*. If all threads in a warp request sequential shared memory indexes, the final bank conflict is at least as possible. This can be done using a reversed tree based reduction:

```
	// Tree based reduction
	for (unsigned int d = blockDim.x/2; d > 1; d >>= 1) {
		if (tId < d)
			if (tId + d < blockDim.x)
				sData[tId] += sData[tId + d];		// Sequential Addressing

		__syncthreads();
	}
```

As it illustrates, in the first iteration, there are *blockdim.x / 2* idle threads. This can turn into a huge waste. To solve the issue on the first iteration we'll provide the next improvement.

Solved Idle Threads Problem

This approach uses the initial *identification* section to perform an addition (reduction) when each thread finds its own global id.

```
	// Identification
	unsigned int tId = threadIdx.x;
	unsigned int i = blockIdx.x * (2 * blockDim.x) + threadIdx.x;

	// Tree based reduction
	for (unsigned int d = blockDim.x / 2; d > 1; d >>= 1) {
		if (tId < d)
			if (tId + d < blockDim.x)
				sData[tId] += sData[tId + d];		// Sequential Addressing

		__syncthreads();
	}
```

It is obvious that when the size of the d is less than 32, everything is being done within a warp. Thus we can remove *__syncthreads()* which saves a lot of time. The procedure of converting for loop iterations to instructions that aren't relying on each other based on a for loop, is called *loop unrolling*. This increases the operational intensity of the kernel. Also, $tId < d$ can be removed while d is less than 32.

Unrolled Warps

```
        // Identification
        unsigned int tId = threadIdx.x;
        unsigned int i = blockIdx.x * (2 * blockDim.x) + threadIdx.x;

        // Initialize
        sData[tId] = 0;

        __syncthreads();

        // Fill up the shared memory
        if (tId < blockDim.x) {
                sData[tId] = g_inData[i] + g_inData[i + blockDim.x];
        }

        __syncthreads();

        // Tree based reduction
        for (unsigned int d = blockDim.x / 2; d > 32; d >>= 1) {
                if (tId < d)
                        if (tId + d < blockDim.x)
                                sData[tId] += sData[tId + d];           // Sequential Addressing

                __syncthreads();
        }

        // Unrolled warps
        if (tId < 32) {
                sData[tId] += sData[tId + 32];
                sData[tId] += sData[tId + 16];
                sData[tId] += sData[tId + 8];
                sData[tId] += sData[tId + 4];
                sData[tId] += sData[tId + 2];
                sData[tId] += sData[tId + 1];
        }
```

Performance Analysis

Performance results for 1B element reduction is given in table 1.1. For a larger input array, GPU performs drastically better. Comparing the serial timings with computation times is not fair. It is fair to compare total time for GPU with serial time. In this case the best speed up is *1.43* compared to serial time. This is what happens in practice. Best computation speed up is *16* which is obtained using the last kernel.

Time (1B Elements)	Time (Transfer + Compute)	Grid Dimensions	Block Dimensions	Bandwidth	Step Speedup	Cumulative Speedup
39.811775	345.950745	1343	1024	1.065 GB/s		
34.879562	340.874652	1343	1024	1.330 GB/s	1.14	1.14
33.555614	338.502478	1343	1024	1.410 GB/s	1.03	1.17
31.712257	337.856598	1343	1024	1.650 GB/s	1.05	1.22
29.756232	334.745698	1343	1024	2.100 GB/s	1.06	1.3
480.923						

Figure 2.1: Illustration of Serial & Parallel Timings.