1 Introduction & Background

1.1 Introduction

In this assignment, we'll gonna implement parallel reduction using **CUDA** program. We'll start from the most straight forward way and develop further optimization based on underlying hardware characteristics. Also by profiling the GPU kernel, we'll analyze the kernel behavior and optimize them more to achieve the ideal performance. Before diving into the parallel reduction, we'll briefly review how the GPU works. The primary purpose of this report is of course implementing parallel reduction and analyze them, establishing semantic understanding to GPU architecture is also desirable. But if you're already familiar with the concept of GPU and GPU programming, you can skip this section and start from [2. Parallel Reduction].

1.2 Background

A **GPU, or Graphics Processing Unit**, is a specialized processor designed to compute massively parallel data. We'll start by exploring how the GPU operates within the overall computer system and then delve into the details of the GPU's architecture. At [1.2.2 GPU Architecture] part, we'll briefly review some important point of GPU operation.

1.2.1 Modern Computing System Architecture

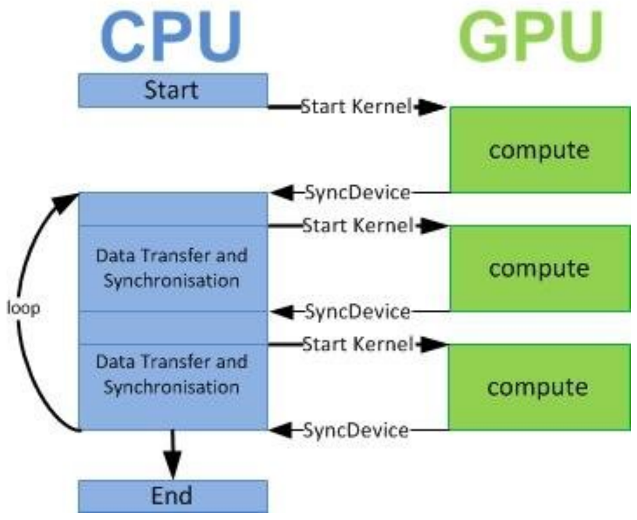


Figure 1: Control Flow between CPU and GPU

CPUs and GPUs have fundamentally different hardware architectures and philosophy. While a CPU consists of a few high-performance cores, a GPU is composed of a large number of low-performance cores. CPU operates at relatively high clock speed and achieves its performance by primarily exploiting ILP with the aid of caching, pre-fetching, branch prediction and etc. In contrast, GPU achieves its performance by primarily exploiting DLP with the aid of massive numbers of threads and fine-grained multi-threading. Due to this distinction, CPUs are good at performing complex, control extensive operation while GPUs are good at performing massively parallel processing. To capture both advantages, modern computer system adapts both different types of processors.

Since CPU has its advantage at performing control extensive workloads, it orchestrates the overall computer systems and take the role of host. In the contrast, GPU has its benefit when performing the massively parallel workloads and take the role of device. As a result, GPU applications are typically divided into two parts: CPU(Host)code and GPU(Device) code. Figure 1 decently depicts the control flow between CPU and GPU in GPU application. At CPU sides, it prepares the data or synchronizesß the execution. Whenever it needs GPU's computing power, it explicitly launch the GPU kernel. Then GPUs, as a device, execute the kernel and do the job. It is noteworthy that typically utilizing the GPU is significantly expensive since it needs to transfer the data from host to device and then receive the computation results back from device to host.

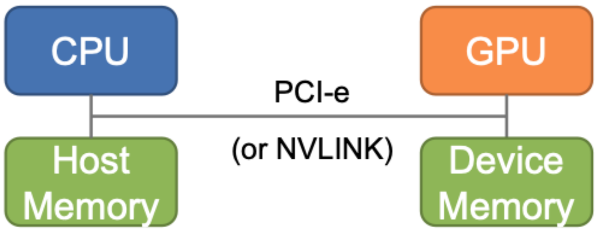
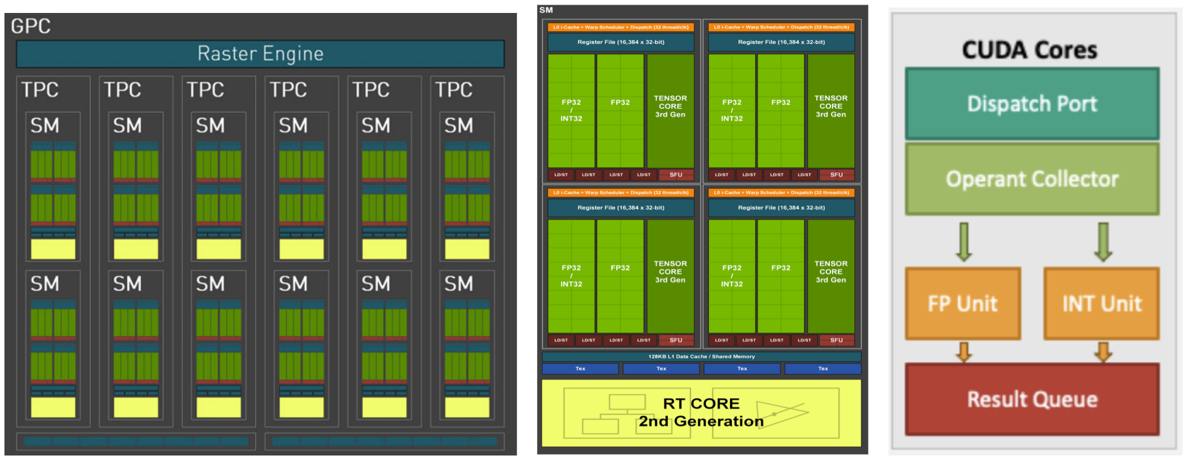


Figure 2: Modern Computer System Architecture

Figure 2 represents the modern computer system architecture with external GPU. As you can see, CPU and GPU are connected by PCI-e and have their own memory. Since they have their own memory space, data must be transferred before and after the kernel launch. Due to this data transfer through PCI-e(or NVLINK), significantly large overheads occur. Therefore when utilizing the GPU, programmers need to consider the size of workloads whether the benefit can sufficiently amortize such overhead.

1.2.2 GPU Architecture



(a) GPC (b) SM (c) CUDA core

Figure 3: Hierarchial View of GPU components

The key concept of GPUs is **massive parallelism**. The performance might be worse for a single or few computations. However, if the workloads become highly parallelizable with massive number of computations, it becomes beneficial compared to CPU. GPU exploits data-level parallelism operating under the **SIMT** (Single-Instruction, Multiple-Threads) model. The SIMT model is similar to the **SIMD** (Single-Instruction, Multiple-Data) model but introduces better flexibility. While vector (or array) processors execute the same instruction on multiple data using multiple processing elements (PEs) and vector registers, GPUs execute the same instruction across multiple threads. Each thread in a GPU can maintain its own context, thereby extending flexibility.

GPU hardware is directly related to the key concept of GPUs. To support such massive parallelism, it requires two hardware support: **1. Compute Throughput** and **2. Memory Throughput**. GPUs fundamentally consist of large numbers of cores, which is called CUDA core in Nvidia, on-chip memory and DRAM.

CUDA core, itself is a processing element, rather than complete core. It consists of FP unit and INT unit that can receive the operand and compute on them. **SM, streaming multiprocessor** is one of the fundamental building blocks of GPUs. It is partitioned by four computing blocks consists of warp selector, L0 I-Cache and CUDA cores. 4 partitions share the L1/Shared memory. Therefore, each SM can execute 4 warps simultaneously increasing the parallelism. With the large numbers of CUDA cores organized consecutively into SM, TPC, GPC, GPU can provide massively large computational throughput. We'll talk again about those organizations later.

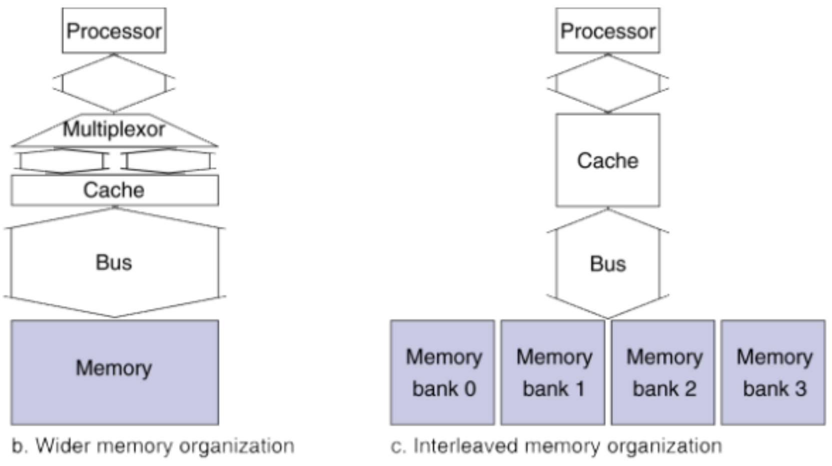
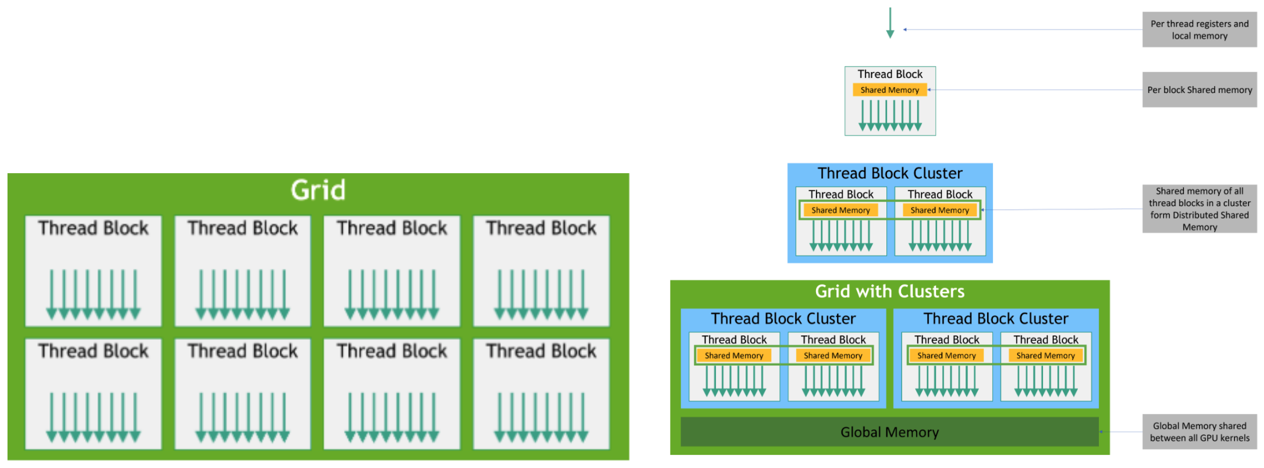


Figure 4: Bank Interleaved Memory Organization

Since large numbers of threads are executed simultaneously, large numbers of memory requests are queried at once. GPU's memory system must support extensive memory access by providing extremely large memory bandwidth. Memory bank interleaving is one of the technique to provide such wide memory bandwidth. Main memory which is seen to be a one logical memory is actually divided into multiple memory bank, which operates independently. Addresses are calculated by modulation operation as follows. $B = Addr\ (mod)\ N$ It is important to note that maximum bandwidth provided by GPU memory system is only achievable when the whole bank is utilized without conflicts. We will look closer about **Bank Conflict** later.

GPUs also utilize memory hierarchy to capture temporal and spatial locality. L1 Cache and shared memory is dedicated for each SM and L2 cache is shared among all SMs. Unlike CPU, since there's massive numbers of active threads and context switching occurs frequently, it is hard to capture temporal locality with default hardware managed cache. To capture temporal locality and utilize cache efficiently, shared memory exists. **Shared memory** is software manged, cache-like memory. It means that programmer has responsibility to utilize and manage the shared memory. By explicitly load frequently used data into shared memory, GPU can capture the temporal locality and effectively reduce the memory access latency. Also there's a large register files (256KB !) per SM. It can hold at most contexts of 64 warps.

Now lets look closer how those large numbers of threads are actually executed in GPU. A **warp**, consisting of 32 threads, is a unit of execution. It means that 32 threads are packed into a warp and executed in SIMT manner. Threads within the same warp executes the same instruction(it means same PC value) simultaneously and in lock-step, meaning their execution is synchronized. However, there's no order of execution between warps. Therefore, if synchronization between different warps is required, programmers need to specify it explicitly. Fine-grained multi-threading is one of the key concept of improving GPU performance. It enables GPU to keep resource utilization full by effectively hiding the latency when warp stalls. Whenever warp stalls for some reason, warp selector selects other ready warp and execute the warp. For example, if cache miss occurs for warp A and it needs to access main memory, other enabled warp B is scheduled and executed. To eliminate the overheads occur in frequent context switching, GPU adopts large size of register files which can hold contexts of 64 warps at maximum. It doesn't need to restore the context, just use the context already stored in register file which leads to zero overhead context switching. Also, there's a warp selector which is dedicated hardware for scheduling unlike CPU. Since frequent context switching occurs in GPU, scheduling specialized hardware, warp selector minimize the idle time between switching the warps. For these reasons, it is very important to launch GPU with sufficient number of threads.


Grid
Thread Block
Per thread registers and local memory
Shared Memory
Per block Shared memory
Thread Block Cluster
Shared memory of all thread blocks in a cluster form Distributed Shared Memory
Grid with Clusters
Global Memory
Global Memory shared between all GPU kernels

(a) Thread Hierarchy (b) Memory Hierarchy

Figure 5: Abstactions of CUDA programming language

Now, lets move on to the **CUDA programming model**. CUDA programming model provides an abstraction of GPU operation. From Nvidia's document, CUDA provides three core abstractions which provides different levels of parallelism: A hierarchy of thread groups, shared memories and barrier synchronization. These abstractions guide programmers to divide a problem into different levels of sub-problems and solve them cooperatively.

It provides two different hierarchies: **Thread Hierarchy** and **Memory Hierarchy**. Figure 5. (a) represents that multiple threads packed into a thread block and multiple thread blocks packed into a grid. In the execution, each thread block is divided into multiple warps and allocated in the **same streaming multiprocessor (SM)**. It simplifies and enhances the communication between threads in the same thread block. Also, in CUDA, threads block can consist of at most 1024 threads.

Figure 5 (b) decently depicts the memory hierarchy with the thread hierarchy. Each thread owns local memory, and thread block owns shared memory shared by all threads in the block. Consecutively, a grid owns their global memory which is shared by all threads in the grid. **Shared memory** primitive provides abstraction of shared memory which is shared among threads in the same thread block. Since it helps to effectively reduce the memory traffic and access latency, programmers need to utilize this primitive wisely. **Barrier synchronization** provides abstractions of communication between threads in the same thread blocks.

As mentioned earlier, there is no order of execution between multiple warps. When programmers want to ensure that execution of threads is synchronized, __syncthreads() primitive must be held explicitly. When __syncthreads() primitive is held, warp waits for all the warps in the same thread block. Since it makes multiple warps idle, it is very expensive primitives and therefore must be utilized appropriately.

Also, it is very interesting connecting the GPU architecture and the abstraction provided in CUDA programming language. For example, to support communication between threads in the thread blocks, warps from the same thread block are allocated in the same SM, and shared memory is dedicated per SM. If you're interested in understanding CUDA C programming model, **NVIDIA Corporation, *CUDA C Programming Guide*, 2023.** might be useful.

2 Parallel Reduction

Reduction refers to reducing the elements of matrix or arrays. By adding the elements of matrices of arrays, we can reduce the size of them. Since addition of elements can't be done at once, it has serial characteristics. To parallelize the reduction, we'll divide entire arrays into multiple sub-arrays and assign them to each thread block. Each thread block will reduce the sub-arrays into scalar value by adding the elements. As a result arrays with number of thread blocks elements will be produced. Since we want to reduce entire arrays into one element, reduction should be continued until the array size fit into one thread block.

2.1 Workload Characteristics

Before diving into actual implementations, let's analyze the workload characteristics of reduction. Assume the size of arrays is N and output result is scalar value. $\log_2 N$ stages are required and size of n stage is $N/2^{i-1}$. Therefore we can conclude that $\sum \frac{N}{2^{i-1}}$ = **2N-2** loads $\sum \frac{N}{2^i}$ = **N-1** adds and $\sum \frac{N}{2^i}$ = **N-1** stores. Therefore, reduction with size N arrays contains $\frac{3N}{2}$ memory accesses and N compute operations. RTX 3090 supports at maximum 935.8GB/s memory bandwith and 35.6TIOPS (INT 32). As a result, we can induce that reduction kernel is actually memory bounded, and therefore we should focus on **memory optimization**.

2.2 Kernel 1: Interleaved Addressing

```
__global__ void reduce_1(const int* d_idata, int* const d_odata, const int n) {
  extern __shared__ volatile int sdata[]; //shared between threads in a block
  unsigned int tid = threadIdx.x;
  unsigned int i   = blockIdx.x * blockDim.x + threadIdx.x;

  sdata[tid]       = d_idata[i];
  __syncthreads();

  for(unsigned int s=1; s<blockDim.x; s *= 2){
    if(tid%(2*s)==0){
      sdata[tid] +=  sdata[tid+s];
    }
    __syncthreads();
  }
  if(tid==0)
    d_odata[blockIdx.x] = sdata[0];
}
```

The above code shows the direct implementation of reduction kernel. It first loads the elements from main memory to shared memory. After that, it adds elements with stride s. Since number of elements become half for every loop iteration, number of active threads also become half correspondingly. In this kernel, thread id is directly assigned to index the elements. Since it accesses data elements with stride s, the thread id which is active is also

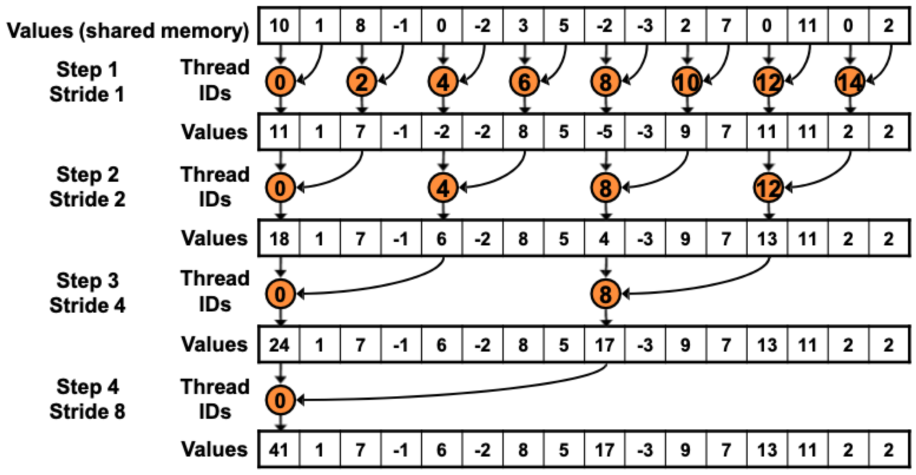


Figure 6: Kernel 1: Interleaved Addressing

interleaved with stride s. It means that active threads in the same warp become half for every stage, which leads to branch divergence and potentially leads to resource underutilization. We'll gonna look closer later.

```
void reduce_optimize(const int* const g_idata, int* const g_odata, const int*
    const d_idata, int* const d_odata, const int n) {
    // TODO: Implement your CUDA code
    // Reduction result must be stored in d_odata[0]
    // You should run the best kernel in here but you must remain other
    kernels as evidence.
    for(unsigned int i=n; i>1; i>>=10){
      dim3 Dimgrid((i+1024-1)>>10,1,1);
      dim3 Dimblock(i>=1024 ? 1024 : i, 1, 1);
      const int* const input = (i==n) ? (d_idata) : (d_odata);
      reduce_1<<<Dimgrid, Dimblock, sizeof(int)*Dimblock.x>>>(input, d_odata,
    i);
    }
}
```

The above code shows the kernel invocation part. As mentioned earlier, it iterates until the size of arrays fits into a single thread block. Since the maximum number of threads per thread block is 1024, each thread operates on arrays with 1024 elements. Therefore, at every reduction, size of arrays become 1/1024.

I'd conducted GPU profiling for the first launched kernel with input array size 2^{24} using Nsight Compute profile tool. Since it wasn't available on server environment, profiling was conducted on local environment with RTX 2080 ti. Since target device is different, the result might be not accurate, however it would be still useful for comparing different versions of reduction kernels.

Metrics	Value
Duration [µs]	1110
Memory [%]	33.82
DRAM Throughput [%]	9.45
Compute (SM) [%]	40.69
Executed Instructions [inst]	166,805,504
Achieved Occupancy [%]	98.47
Achieved Active Warps Per SM	31.51

Table 1: Kernel1: Profiling Results

Above table represents the part of profile results. First of all, we can observe that portion of memory access is only about 33%. It means that for some other reasons, not the memory access, warps are stalled and performance degradation occurs. Also we can see that DRAM throughput is only utilized 9.45 % compared to maximum bandwidth. But it is suffering from other factor, we need to address this factor first. This factor might be **branch divergence**. Since the threads activated are strided with variable s, the activated threads in the same warp become a half for every loop iterations. We'll address branch divergence in kernel 2. The other metrics such as achieved occupancy and achieved active warps per SM is convincing and reasonable.

2.3 Kernel 2: Interleaved Addressing w/o Branch Divergence

We'll change the distribution of active threads to be sequential. By doing so, we can ensure that all of the threads in the same warp activated or deactivated, and therefore avoid the branch divergence. Also, GPU doesn't actually execute instructions for deactivated threads, we might gonna reduce the instructions executed as well. However it's important to note that the addressing of data remains interleaved. We'll discuss about this part later.

```
__global__ void reduce_2(const int* d_idata, int* const d_odata, const int n) {
  extern __shared__ volatile int sdata[]; //shared between threads in a block
  unsigned int tid = threadIdx.x;
  unsigned int i   = blockIdx.x * blockDim.x + threadIdx.x;

  sdata[tid]       = d_idata[i];
  __syncthreads();
  for(unsigned int s=1; s<blockDim.x; s *= 2){
    int index = 2*s*tid;
    if((index< blockDim.x))
      sdata[index] +=  sdata[index+s];
    __syncthreads();
  }
  if(tid==0)
    d_odata[blockIdx.x] = sdata[0];
}
```

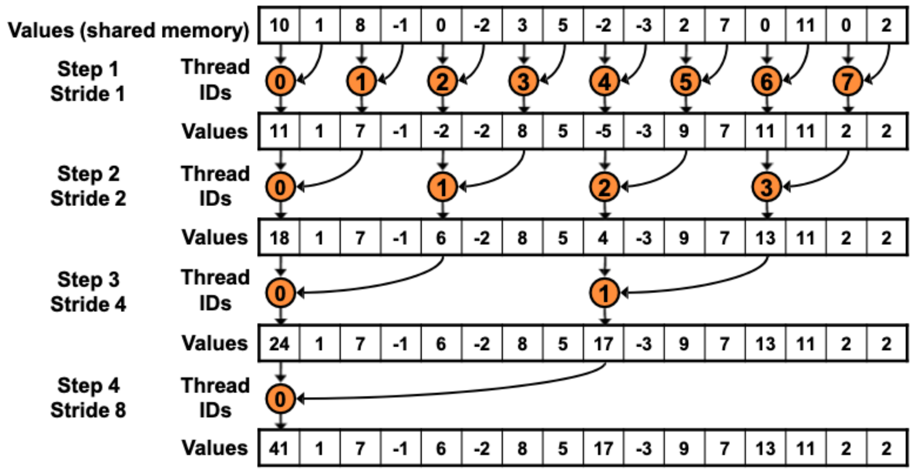


Figure 7: Kernel 2: Interleaved addressing w/o Branch Divergence

Above code shows the reduction kernel with sequential addressing. Instead of mod operation for activating the threads and directly use tid as an index, I'd calculated index based on tid, and activated the threads by comparing the index and block dimensions. Now, we can induce that tids of activated threads will be continuous and therefore might not gonna suffer from branch divergence. Since the kernel invocation part is the same, I'd omitted the code.

Metrics	Value
Duration [µs]	1110
Memory [%]	33.82
DRAM Throughput [%]	9.45
Compute (SM) [%]	40.69
Executed Instructions [inst]	166,805,504
Achieved Occupancy [%]	98.47
Achieved Active Warps Per SM	31.51

Table 2: Kernel 1

Metrics	Value
Duration [µs]	744.7
Memory [%]	50.23
DRAM Throughput [%]	14
Compute (SM) [%]	50.23
Executed Instructions [inst]	70,860,800
Achieved Occupancy [%]	97.64
Achieved Active Warps Per SM	31.24

Table 3: Kernel 2

First, we achieved $\times 1.5$ speedup. Since the the overall runtime has been decreased, memory utilization and DRAM throughput increased with corresponding factor. Also, we can observe that the number of executed instructions decreased by about 2.36 times. As mentioned earlier, data memory addressing remains interleaved. Therefore the memory accesses which are done parallel by threads in the warp are distributed to overall shared memory region. Since the wide-spread address distribution, bank conflicts occur on shared memory which leads to memory underutilization.

2.4 Kernel 3: Sequential Addressing

In kernel 3, by changing the memory access pattern to be sequential, we will address the shared memory bank conflicts.

```
__global__ void reduce_3(const int* d_idata, int* const d_odata, const int n) {

  extern __shared__ volatile int sdata[]; //shared between threads in a block
  unsigned int tid = threadIdx.x;
  unsigned int i   = blockIdx.x * blockDim.x + threadIdx.x;

  sdata[tid]       = d_idata[i];
  __syncthreads();

  for(unsigned int s=blockDim.x>>1; s>0; s >>= 1){
    if(tid<s)
      sdata[tid] +=  sdata[tid+s];
    __syncthreads();
  }
  if(tid==0)
    d_odata[blockIdx.x] = sdata[0];
}
```

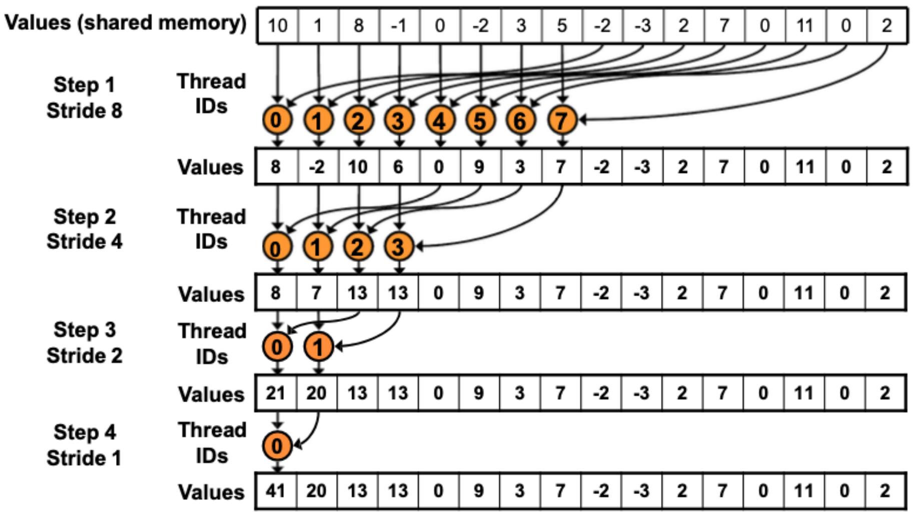


Figure 8: Kernel 3: Sequential Addressing

The above code shows sequential addressing reduction kernel. tid instead of index is used to indexing the elements of array. Since thread IDs are sequential within the threads in the same warp, it guarantees access to data sequentially. The kernel invocation part is the same, so I omitted it.

In profiling, I observed that kernel 3 achieved ×1.15 speed up, but no other significant differences. (Actually I failed to profile shared memory behavior.) By the way, we can

Metrics	Value
Duration [µs]	648.77
Memory [%]	57.84
DRAM Throughput [%]	15.95
Compute (SM) [%]	57.84
Executed Instructions [inst]	61,423,616
Achieved Occupancy [%]	97.26
Achieved Active Warps Per SM	31.12

Table 4: Kernel 3: Profiling Results

observe that DRAM is still underutilized. The cause of this underutilization is due to the workloads assigned to each thread doesn't contain enough memory accesses. I think this might not be a problem if all of the threads in the thread block can be actually executed at once. However in reality, due to the hardware constraints, SM cannot execute all of the threads in the thread block simultaneously. Therefore, memory accesses become sequential and potentially leads to memory underutilization.

2.5 Kernel 4: First Add During Load

The main idea of kernel 4 is to increase the number of memory access per thread. Of course the total memory accesses remain the same, but it can bring data more efficiently since there's a less overhead compared to single memory access per thread.

```
__global__ void reduce_4(const int* d_idata, int* const d_odata, const int n) {
  extern __shared__ volatile int sdata[]; //shared between threads in a block
  unsigned int tid = threadIdx.x;
  unsigned int i   = blockIdx.x * (blockDim.x*2) + threadIdx.x;

  sdata[tid]       = d_idata[i] + d_idata[i+blockDim.x];
  __syncthreads();

  for(unsigned int s=blockDim.x>>1; s>0; s >>= 1){
    if(tid<s)
      sdata[tid] +=  sdata[tid+s];
    __syncthreads();
  }
  if(tid==0)
    d_odata[blockIdx.x] = sdata[0];
}
```

The above code shows the behavior of kernel 4. Instead of just loading the single elements of array, it loads and add two data elements and store it to the shared memory.

```
void reduce_optimize(const int* const g_idata, int* const g_odata, const int*
    const d_idata, int* const d_odata, const int n) {
```

```
    // TODO: Implement your CUDA code
    // Reduction result must be stored in d_odata[0]
    // You should run the best kernel in here but you must remain other
    kernels as evidence.
    for(unsigned int i=n; i>1; i>>=10){

      dim3 Dimgrid((i+1024-1)>>10,1,1);
      dim3 Dimblock(i>=1024 ? 512 : (i+2-1)>>1, 1, 1);

      const int* const input = (i==n) ? (d_idata) : (d_odata);
      reduce_4<<<Dimgrid, Dimblock, sizeof(int)*Dimblock.x>>>(input, d_odata,
    i);
    }
}
```

Above code represents the kernel invocation part of the parallel reduction. Since we replaced a single load into two loads and one addition, the size of thread block becomes a half correspondingly.

Metrics	Value
Duration [µs]	648.77
Memory [%]	57.84
DRAM Throughput [%]	15.95
Compute (SM) [%]	57.84
Executed Instructions [inst]	61,423,616
Achieved Occupancy [%]	97.26
Achieved Active Warps Per SM	31.12

Table 5: Kernel 3

Metrics	Value
Duration [µs]	254.62
Memory [%]	69.59
DRAM Throughput [%]	40.78
Compute (SM) [%]	69.59
Executed Instructions [inst]	29,442,048
Achieved Occupancy [%]	89.77
Achieved Active Warps Per SM	28.73

Table 6: Kernel 4

We can observe that it achieves about $\times 2.55$ speed up compared to previous kernel. Also, as expected, DRAM throughput also increased to 40.70 %. Executed instructions become a half since the number of total threads become a half. Also, while profiling, I got following message: "LSU is the highest-utilized pipeline (70.1%). It executes load/store memory operations. The pipeline is well-utilized and might become a bottleneck if more work is added." Thus we can induce that memory utilization entered to convincing region. Regarding to occupancy, we can observe that achieved occupancy is only 89.77%. Theoretically, if sufficient amount of threads are launched, occupancy should be 100%. I guess this is because we reduce the thread block size by half. We'll gonna discuss it later. __syncthreads() in the for loop was to synchronize the execution of threads(warp) in the same thread block. However there is no need to synchronize the execution in the case that active threads are in the same warp. Next kernel will remove unnecessary synchronization and unroll the loop effectively.

2.6 Kernel 5 & Kernel 6: Loop Unrolling

Since threads in the same warp is already synchronized, __syncthreads() keyword is useless for that case thus we'll remove unnecessary synchronization and effectively reduce the overheads.

```
__global__ void reduce_5(const int* d_idata, int* const d_odata, const int n) {
  extern __shared__ volatile int sdata[]; //shared between threads in a block
  unsigned int tid = threadIdx.x;
  unsigned int i   = blockIdx.x * (blockDim.x*2) + threadIdx.x;

  sdata[tid]       = d_idata[i] + d_idata[i+blockDim.x];
  __syncthreads();

  for(unsigned int s=blockDim.x/2; s>32; s >>= 1){
    if(tid<s)
      sdata[tid] +=  sdata[tid+s];
    __syncthreads();
  }

  if(tid<32 &&blockDim.x >= 64) sdata[tid] += sdata[tid+32];
  if(tid<16 &&blockDim.x >= 32) sdata[tid] += sdata[tid+16];
  if(tid<8  &&blockDim.x >= 16) sdata[tid] += sdata[tid+8];
  if(tid<4  &&blockDim.x >= 8)  sdata[tid] += sdata[tid+4];
  if(tid<2  &&blockDim.x >= 4)  sdata[tid] += sdata[tid+2];
  if(tid<1  &&blockDim.x >= 2)  sdata[tid] += sdata[tid+1];
  if(tid==0)
    d_odata[blockIdx.x] = sdata[0];
}
```

When active threads are in the same warp, __syncthreads() keyword is removed and loop is also unrolled. Using template we can unroll the remaining loop as follows.

```
template <unsigned int blockSize>
__global__ void reduce_6(const int* d_idata, int* const d_odata, const int n) {
  extern __shared__ volatile int sdata[]; //shared between threads in a block
  unsigned int tid = threadIdx.x;
  unsigned int i   = blockIdx.x * (blockDim.x*2) + threadIdx.x;

  sdata[tid]       = d_idata[i] + d_idata[i+blockDim.x];
  __syncthreads();

  if(blockSize >= 1024){
    if(tid < 512) sdata[tid] += sdata[tid+512];
    __syncthreads();
  }
  if(blockSize >= 512){
    if(tid < 256) sdata[tid] += sdata[tid+256];
    __syncthreads();
  }
  if(blockSize >= 256){
    if(tid < 128) sdata[tid] += sdata[tid+128];
    __syncthreads();
  }
  if(blockSize >= 128){
    if(tid < 64)  sdata[tid] += sdata[tid+64];
    __syncthreads();
  }

  if(tid<32){
    if(blockDim.x >= 64) sdata[tid] += sdata[tid+32];
    if(blockDim.x >= 32) sdata[tid] += sdata[tid+16];
    if(blockDim.x >= 16) sdata[tid] += sdata[tid+8];
    if(blockDim.x >= 8)  sdata[tid] += sdata[tid+4];
    if(blockDim.x >= 4)  sdata[tid] += sdata[tid+2];
    if(blockDim.x >= 2)  sdata[tid] += sdata[tid+1];
  }
  if(tid==0)
    d_odata[blockIdx.x] = sdata[0];
}
```

Kernel 5 can be invoked same as previous kernel. Above code represents the kernel 6 invocation part using template.

```
void reduce_optimize(const int* const g_idata, int* const g_odata, const int*
    const d_idata, int* const d_odata, const int n) {
    // TODO: Implement your CUDA code
    // Reduction result must be stored in d_odata[0]
    // You should run the best kernel in here but you must remain other
    kernels as evidence.
    for(unsigned int i=n; i>1; i>>=10){

      dim3 Dimgrid((i+1024-1)>>10,1,1);
      dim3 Dimblock(i>=1024 ? 512 : (i+1)>>1, 1, 1);

      const int* const input = (i==n) ? (d_idata) : (d_odata);
      switch(Dimblock.x){
        case 1024:  reduce_6<1024> <<<Dimgrid, Dimblock,
    sizeof(int)*Dimblock.x>>>(input, d_odata, i); break;
        case 512:   reduce_6<512> <<<Dimgrid, Dimblock,
    sizeof(int)*Dimblock.x>>>(input, d_odata, i); break;
        case 256:   reduce_6<256> <<<Dimgrid, Dimblock,
    sizeof(int)*Dimblock.x>>>(input, d_odata, i); break;
        case 128:   reduce_6<128> <<<Dimgrid, Dimblock,
    sizeof(int)*Dimblock.x>>>(input, d_odata, i); break;
```

```
        case 64:    reduce_6<64>  <<<Dimgrid, Dimblock,
    sizeof(int)*Dimblock.x>>>(input, d_odata, i); break;
        case 32:    reduce_6<32>  <<<Dimgrid, Dimblock,
    sizeof(int)*Dimblock.x>>>(input, d_odata, i); break;
        case 16:    reduce_6<16>  <<<Dimgrid, Dimblock,
    sizeof(int)*Dimblock.x>>>(input, d_odata, i); break;
        case 8:     reduce_6<8>   <<<Dimgrid, Dimblock,
    sizeof(int)*Dimblock.x>>>(input, d_odata, i); break;
        case 4:     reduce_6<4>   <<<Dimgrid, Dimblock,
    sizeof(int)*Dimblock.x>>>(input, d_odata, i); break;
        case 2:     reduce_6<2>   <<<Dimgrid, Dimblock,
    sizeof(int)*Dimblock.x>>>(input, d_odata, i); break;
        case 1:     reduce_6<1>   <<<Dimgrid, Dimblock,
    sizeof(int)*Dimblock.x>>>(input, d_odata, i); break;
      }
    }
}
```

Metrics	Value
Duration [µs]	254.62
Memory [%]	69.59
DRAM Throughput [%]	40.78
Compute (SM) [%]	69.59
Executed Instructions [inst]	29,442,048
Achieved Occupancy [%]	89.77
Achieved Active Warps Per SM	28.73

Table 7: Kernel 4

Metrics	Value
Duration [µs]	221.89
Memory [%]	80.09
DRAM Throughput [%]	46.92
Compute (SM) [%]	80.09
Executed Instructions [inst]	23,964,288
Achieved Occupancy [%]	88.01
Achieved Active Warps Per SM	28.16

Table 8: Kernel 5

Metrics	Value
Duration [µs]	161.18
Memory [%]	65.18
DRAM Throughput [%]	65.18
Compute (SM) [%]	50.91
Executed Instructions [inst]	11,042,816
Achieved Occupancy [%]	70.49
Achieved Active Warps Per SM	22.56

Table 9: Kernel 6

Kernel 5 and 6 achieves $\times 1.148$ and $\times 1.58$ Speed Up each. We can observe that executed instructions decreased by factor of 1.23 and 2.66 each. Actually I can't fully understand why unrolling the loop of remaining part at kernel 6 affects the results that much. I think it not only removes the loop overhead, but compiler also optimizes the branches and eliminate them. I'd disassembled and get PTX file from source code. Of course, PTX isn't actual ISA that directly executed in GPU, but the disassemble result is still valid. As expected, it compiled **11 versions of kernel 6** depending on the template variable, blockSize. So, in run time, if template variable is determined, it runs the corresponding version of kernels which is actually free from branch instructions.

Also in kernel 6, we can confirm that occupancy is significantly decreased. I guess in previous kernel, there's lots of active kernels in fact in idle state, but int kernel 6 it allevated for some reasons.

2.7 Kernel 7: Multiple Adds Per Thread

The main idea of kernel 7 is just extending the optimization technique conducted on kernel 4. As talked earlier, since all of the threads in the same thread block cannot be fully parallelized, assigning more workloads can be valid optimization. However I think the degree of extending this technique must be adjusted based on target hardware.

```
template <unsigned int blockSize>
__global__ void reduce_7(const int* d_idata, int* const d_odata, const int n) {
  extern __shared__ volatile int sdata[]; //shared between threads in a block
  unsigned int tid = threadIdx.x;
  unsigned int i   = blockIdx.x * (blockSize*2) + threadIdx.x;
  unsigned int gridSize = blockSize*2*gridDim.x;

  sdata[tid] = 0;
  __syncthreads();

  while(i < n){
    sdata[tid]     += d_idata[i] + d_idata[i+blockSize];
    i += gridSize;
  }
  __syncthreads();

  if(blockSize >= 1024){
    if(tid < 512) sdata[tid] += sdata[tid+512];
    __syncthreads();
  }
  if(blockSize >= 512){
    if(tid < 256) sdata[tid] += sdata[tid+256];
    __syncthreads();
  }
  if(blockSize >= 256){
    if(tid < 128) sdata[tid] += sdata[tid+128];
    __syncthreads();
  }
  if(blockSize >= 128){
    if(tid < 64)  sdata[tid] += sdata[tid+64];
    __syncthreads();
  }
  if(tid<32){
    if(blockSize >= 64) sdata[tid] += sdata[tid+32]; __syncthreads();
    if(blockSize >= 32) sdata[tid] += sdata[tid+16]; __syncthreads();
    if(blockSize >= 16) sdata[tid] += sdata[tid+8];  __syncthreads();
    if(blockSize >= 8)  sdata[tid] += sdata[tid+4];  __syncthreads();
    if(blockSize >= 4)  sdata[tid] += sdata[tid+2];  __syncthreads();
    if(blockSize >= 2)  sdata[tid] += sdata[tid+1];  __syncthreads();
  }
  if(tid==0)
    d_odata[blockIdx.x] = sdata[0];
}
```

It divides total array into multiple sub-arrays and make them one sub arrays by adding each element with another. And one sub arrays are further divided into smaller arrays, which is assigned to each thread block, and reduced into one elements. In while loop, at every iteration, gridSize is added to index the total array. Strided with gridSize helps to maintain the coalescing when accessing the main memory.

```
void reduce_optimize(const int* const g_idata, int* const g_odata, const int*
    const d_idata, int* const d_odata, const int n) {
    // TODO: Implement your CUDA code
    // Reduction result must be stored in d_odata[0]
    // You should run the best kernel in here but you must remain other
    kernels as evidence.
      for(unsigned int i=n; i>1; i>>=13){

        dim3 Dimgrid((i+8192-1)>>13,1,1);
        dim3 Dimblock(i>=8192 ? 512 : (i+15)>>4, 1, 1);

        const int* const input = (i==n) ? (d_idata) : (d_odata);
        switch(Dimblock.x){
          case 1024:  reduce_7<1024> <<<Dimgrid, Dimblock,
    sizeof(int)*Dimblock.x>>>(input, d_odata, i); break;
          case 512:   reduce_7<512> <<<Dimgrid, Dimblock,
    sizeof(int)*Dimblock.x>>>(input, d_odata, i); break;
          case 256:   reduce_7<256> <<<Dimgrid, Dimblock,
    sizeof(int)*Dimblock.x>>>(input, d_odata, i); break;
          case 128:   reduce_7<128> <<<Dimgrid, Dimblock,
    sizeof(int)*Dimblock.x>>>(input, d_odata, i); break;
          case 64:    reduce_7<64>  <<<Dimgrid, Dimblock,
    sizeof(int)*Dimblock.x>>>(input, d_odata, i); break;
          case 32:    reduce_7<32>  <<<Dimgrid, Dimblock,
    sizeof(int)*Dimblock.x>>>(input, d_odata, i); break;
          case 16:    reduce_7<16>  <<<Dimgrid, Dimblock,
    sizeof(int)*Dimblock.x>>>(input, d_odata, i); break;
```

```
            case 8:     reduce_7<8>   <<<Dimgrid, Dimblock,
    sizeof(int)*Dimblock.x>>>(input, d_odata, i); break;
            case 4:     reduce_7<4>   <<<Dimgrid, Dimblock,
    sizeof(int)*Dimblock.x>>>(input, d_odata, i); break;
            case 2:     reduce_7<2>   <<<Dimgrid, Dimblock,
    sizeof(int)*Dimblock.x>>>(input, d_odata, i); break;
            case 1:     reduce_7<1>   <<<Dimgrid, Dimblock,
    sizeof(int)*Dimblock.x>>>(input, d_odata, i); break;
        }
    }
```

When launching the code, kernel configuration can vary depending on the target device.

Metrics	Value
Duration [µs]	161.18
Memory [%]	65.18
DRAM Throughput [%]	65.18
Compute (SM) [%]	50.91
Executed Instructions [inst]	11,042,816
Achieved Occupancy [%]	70.49
Achieved Active Warps Per SM	22.56

Table 10: Kernel 6

Metrics	Value
Duration [µs]	123.87
Memory [%]	85.55
DRAM Throughput [%]	85.55
Compute (SM) [%]	25.96
Executed Instructions [inst]	4,694,016
Achieved Occupancy [%]	93.34
Achieved Active Warps Per SM	29.87

Table 11: Kernel 7

By apply multiple adds per thread, we achieved $\times 1.30$ Speed Up compared to kernel 6. Correspondingly, memory utilization and DRAM throughput also increased with the similar factor. It means that in previous kernel, DRAM was actually underutilized. Since total number of threads decreased, total executed instructions also decreased. Also we should consider the kernel configuration, I'd conducted experiment to analyze the results depending on kernel block and grid size.

Block Size	Grid Size			
	512	1024	2048	4096
128	122.72	123.42	122.66	122.59
256	122.82	122.91	122.53	122.66
512	123.26	123.94	123.87	123.01
1024	126.05	126.08	126.27	147.84

Table 12: Result depends on Block Size and Grid Size

It is important that the workload depends on the Grid Size. Input data size is the same, but output size is same to grid size. For example, kernel with grid size 1024 perform reductions on 2^{24} size of array and produce 2^{10} elements array. Therefore, we can conclude that block size with 128 and grid size 512 is the most fast parallel reduction in the given range.

3 Reference

[1] "Nsight Compute Command Line Interface (CLI)," NVIDIA Documentation, 2024. [Online]. Available: https://docs.nvidia.com/nsight-compute/NsightComputeCli/index.html. [Accessed: 02-Jun-2024].

[2] "CUDA C Programming Guide," NVIDIA Documentation, 2024. [Online]. Available: https://docs.nvidia.com/cuda/cudacprogramming-guide/index.html. [Accessed: 02Jun2024].

[3] M. Harris, "Optimizing Parallel Reduction in CUDA," NVIDIA Developer Blog, 2007. [Online]. Available: https://developer.nvidia.com/blog/optimizingparallelreductioncuda/. [Accessed: 02-Jun-2024].

[4] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym, "NVIDIA TESLA: A Unified Graphics and Computing Architecture," IEEE Micro, vol. 28, no. 2, pp. 3955, Mar.-Apr. 2008. [Online]. Available: https://research.nvidia.com/publication/2008-04_nvidia-teslaunifiedgraphicsandcomputing-architecture. [Accessed: 02-Jun-2024].