Master Thesis

Evaluation of GPU based Segmented Scans

Author:
Saidi Reddy Vummadi

23rd September 2019

Advisors:
Prof. Dr. rer. nat. habil. Gunter Saake
Institute of Technical and Business Information Systems

M.Sc. Andreas Meister
Institute of Technical and Business Information Systems

M.Sc. David Broneske
Institute of Technical and Business Information Systems
Abstract

Data-parallel primitives play a significant role in a large number of applications using parallel architecture devices. One of the commonly used devices is GPU (graphical processing unit). GPUs are used from the 1970s, but the term GPU was popularized in 1999. These GPUs are used in many modern-day applications as they are more efficient than general-purpose CPUs (central processing unit). The parallel structure makes them more useful. So with development in parallel architecture devices, the pivot on utilizing the capacity of parallel devices became crucial. In the process, many methods are introduced for optimizing the parallel devices, and one of them is scan. The first naive scan was introduced by Hillis and Steele, which is not efficient, so in further years, blelloch introduced an efficient work scan. The efficient work scan is extended for solving many similar tasks as one task called blelloch segmented scan. The blelloch scan cannot solve the tasks which can be solved by other methods like the sparse matrix method, which uses fewer data storage compared to the blelloch scan. In this paper, we implemented a blelloch segmented scan and used a sparse matrix method to generate the blelloch segmented scan output on GPU using OpenCL API, and we evaluated them for small data sets. From the evaluation results, we found that both the methods are taking the same runtime for executing the same kind of data. So we have developed an equivalent method (sparse matrix segmented scan) to blelloch segmented scan. In the future, by using workgroups, we can get better results as we are getting the same results for one thread block.
Acknowledgements

I would like to thank God for giving me strength and good health throughout my thesis work. I especially thank my supervisors, M.Sc. Andreas Meister and M.Sc. David Broneske for their great support, encouragement, and valuable inputs throughout my work. I am grateful to Prof. Dr. rer. nat. habil. Gunter Saake for giving me an excellent opportunity to do my Master Thesis under his guidance.
Contents

List of Figures x

List of Tables xi

1 Introduction 1
  1.1 Aim of the thesis 2
  1.2 Structure of the thesis 2

2 Background 3
  2.1 Parallel Programming 4
  2.2 CPUs and GPUs 5
    2.2.1 CPU 5
    2.2.2 GPU 5
  2.3 Real-World Example 6
    2.3.1 Task-level parallelism 6
    2.3.2 Data-level parallelism 6
  2.4 Supercomputers 7
  2.5 OpenCL 8
  2.6 Scan 15

3 Types of Scan 17
  3.1 Sequential Scan 17
  3.2 Parallel Scan 17
    3.2.1 Naive Parallel Scan 17
    3.2.2 Blelloch Scan 18

4 Segmented scan 23
  4.1 Implementation setup 24
  4.2 Implementation of Blelloch segmented scan 24

5 Sparse Matrix 29
  5.1 Sparse Matrix-Vector Multiplication 30
  5.2 Implementation of Segmented Scan Using Sparse Matrix 32

6 Evaluation 35
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>CPU block diagram</td>
<td>3</td>
</tr>
<tr>
<td>2.2</td>
<td>concurrency</td>
<td>4</td>
</tr>
<tr>
<td>2.3</td>
<td>parallel computing</td>
<td>5</td>
</tr>
<tr>
<td>2.4</td>
<td>Core diagram</td>
<td>5</td>
</tr>
<tr>
<td>2.5</td>
<td>Task-level parallelism</td>
<td>6</td>
</tr>
<tr>
<td>2.6</td>
<td>Data-level parallelism</td>
<td>7</td>
</tr>
<tr>
<td>2.7</td>
<td>OpenCL class diagram. [1]</td>
<td>9</td>
</tr>
<tr>
<td>2.8</td>
<td>Memory model. [2]</td>
<td>11</td>
</tr>
<tr>
<td>2.9</td>
<td>comparison of serial and parallel code</td>
<td>13</td>
</tr>
<tr>
<td>2.10</td>
<td>Parallel execution with n work items</td>
<td>14</td>
</tr>
<tr>
<td>2.11</td>
<td>Flow chart for sequential execution</td>
<td>14</td>
</tr>
<tr>
<td>3.1</td>
<td>The sequential scan</td>
<td>18</td>
</tr>
<tr>
<td>3.2</td>
<td>The Naive Parallel scan</td>
<td>19</td>
</tr>
<tr>
<td>3.3</td>
<td>The reduce (up-sweep) phase scan. [3]</td>
<td>19</td>
</tr>
<tr>
<td>3.4</td>
<td>The down-sweep phase scan. [3]</td>
<td>20</td>
</tr>
<tr>
<td>3.5</td>
<td>The down-sweep inclusive scan</td>
<td>21</td>
</tr>
<tr>
<td>4.1</td>
<td>Specifications of GPU</td>
<td>24</td>
</tr>
<tr>
<td>4.2</td>
<td>Reduce phase kernel function</td>
<td>27</td>
</tr>
<tr>
<td>5.1</td>
<td>CSR format</td>
<td>30</td>
</tr>
<tr>
<td>5.2</td>
<td>kernel function of SpMV</td>
<td>31</td>
</tr>
<tr>
<td>5.3</td>
<td>kernel function</td>
<td>33</td>
</tr>
</tbody>
</table>
6.1 Line Graph ................................................................. 36
6.2 Column Graph ........................................................... 37
6.3 Data Transfer Time Graph ............................................ 37
6.4 Kernel Execution Time Graph ........................................ 38
List of Tables

2.1 Difference between OpenCL and CUDA ............... 12

4.1 Work and step complexity ............................. 24
1. Introduction

The GPUs are one of the most important parts of the computing machines in the modern world due to its parallel architecture. These powerful devices are used for large and difficult computational tasks. Initially, GPUs are used for graphics applications, where the same kind of work can be done many times individually. Nowadays, GPUs are used for running many general-purpose applications (GPGPU). From many years CUDA and OpenCL are famous in GPU programming. CUDA is designed by NVIDIA, and it is a closed source language that runs only on the Nvidia platform. OpenCL is an open-source language that can run on any platform like AMD, Intel, and Nvidia also. OpenCL was designed by Apple, and later it was taken by the Khronos group.

Due to the development of parallel computer architecture, the programming of software has developed from sequential to parallel. The parallel programming can run the code on both CPUs and GPUs. There are many algorithms developed for parallel programming as the applications based on parallel computing increased.

The scan algorithm is one of the basic algorithms for parallel programming, which is also called as prefix sum [4]. The scan was suggested by Iverson in the mid-1950s as operations for the APL language. A parallel version of scans was later suggested by Stone. A scan is used for many applications like radix sort, quick sort, removing elements, etc. [5]. The scan operations are so useful in the practical world that they are assigned as primitive instructions. A scan is of two types inclusive scan and exclusive scan, which are explained in detailed in further chapters.

In scans, there are different algorithms which are efficient than basic scan and one of them is Blelloch scan, in which the processing of task is split into two phases, i.e., upsweep phase and down sweep phase [4]. The scan can be used for solving multiple tasks all at once, which have the same instructions. This is done by combing the individual tasks and splitting them after the process is done, and the individual results are collected. This process of scan is called a segmented scan [6].
A sparse matrix is also one of the efficient algorithms used for the optimization of problems [7]. The sparse matrix optimizes the task data and uses less space for storage. There are different formats to store the sparse matrix in a database like CSR format, COO format, etc., which we see in further chapters.

1.1 Aim of the thesis

The goal of the thesis is:

- To implement the blelloch segmented scan using a sparse matrix storage method, i.e., CSR format on GPU.
- To evaluate the blelloch segmented scan and sparse matrix method by taking run time and segment length.

1.2 Structure of the thesis

The remaining structure of the thesis as follows:

- **Chapter 2** introduce the basics of parallel programming, the difference between CPU and GPU and also gives the terminology of OpenCL.
- **Chapter 3**, we discuss the scan and parallelization of the scan, blelloch scan
- **Chapter 4** introduces the segmented scan and implementation of blelloch segmented scan.
- **Chapter 5** introduces the sparse matrix, the storage formats, and the implementation of a segmented scan using a sparse matrix.
- **Chapter 6**, we evaluate the blelloch segmented scan and sparse matrix segmented scan by plotting graphs.
- **Chapter 7**, we discuss the results and conclude the thesis.
2. Background

The procedure of solving large problems by breaking them into smaller pieces and running these problems at the same period of time is called parallel computing. The present world has run into parallel that all the machines are using highly parallel devices. The modern parallel devices have thousands of ALUs that run the thousands of arithmetic operations simultaneously, thousands of processors that each can run a piece of the problem in parallel and tens of thousands of threads where each piece of work is a thread. All these futures help us to solve our problems faster but utilizing all the computing power at the same time in parallel needs a different way of programming than programming a single scalar program. This type of programming is called parallel programming.

![CPU block diagram](image)

Figure 2.1: CPU block diagram
2.1 Parallel Programming

In any computer, the instructions of a program are stored in the hard drive, and when you execute the program, the instructions from hard drive are moved to RAM then the computing unit takes the input and processes the data and gives the output. The basic model is shown in Figure 2.1. Regardless of the size of the input or the number of instructions the computing unit executes one instruction at a time sequentially but, computers can run multiple applications simultaneously even it is a single-core processor because of the concept process [8].

A process is a state of a program that is being executed, and data of the state is stored, which is called context [9]. Context holds the state of an execution process [10] and executes one process for a certain period and stops the process and executes the next process for another amount of time. This loop shifts from one process to another process during execution called context switching [8]. Each process has a main thread, which is the entry point of the process. If different tasks are running on a single processor at the same time, we call it as concurrency. The modern CPUs have multiple cores with multiple threads that can execute multiple instructions without context. This type of execution process is called parallel processing or parallel computing. Concurrency and parallel computing are similar because both run programs simultaneously in the software level, but when you go for hardware-level, the parallel code runs in a sequence if the system doesn’t have enough cores which are not useful for improving the performance of the system. Concurrency can also happen in some parallel programming. Concurrency and parallel computing process is shown in Figure 2.2 and Figure 2.3.

Figure 2.2: concurrency

To benefit from parallel computing, we need multiple core system, although modern computers are multi-core they do not have thousands of cores, which is not efficient. To overcome this, we use GPUs. The GPUs (graphical processing unit) are designed to address the demand for high-resolution graphics. So GPUs are built differently than standard CPUs.
2.2 CPUs and GPUs

2.2.1 CPU

CPU (central processing unit) has a very little number of computing cores. The CPU showed in Figure 2.4 has only 6 cores. These cores have advanced optimization hardware like branch predictors and multilevel caches. In CPU, context switching is done by software.

2.2.2 GPU

GPUs are built with thousands of computing cores, as shown in Figure 2.4, these cores accommodate thousands of instructions at once. These cores do not have branch predictors and have less clock speed compared to CPU. As GPUs have thousands of cores, we call it massively parallel architecture. In GPU context-switching is done by hardware.
2.3 Real-World Example

For a better understanding of parallel programming, we take a real-world example of building a house. For building a house, there will be many tasks like planning, foundation laying, building walls, roofing, wiring, painting, furnishing gardening. In these tasks, planning and foundation laying need to be done in sequence but the other tasks can be done simultaneously. For example, some workers can work on building sidewalls, and some can work on wiring, some on painting. So to complete any task, certain things should be done in sequential order, and some tasks can be done simultaneously/parallelly. The same process runs in computer tasks so better performance can be achieved in the combination of CPUs and GPUs. The capacity of both CPU and GPUs can be utilized by programming with parallelism concepts. There are two types of parallelism [11].

2.3.1 Task-level parallelism

Assigning different threads to perform a different task or different task on the same or different data is called task-level parallelism [11]. In the real world example if we assign multiple workers for different tasks, as shown in Figure 2.5 is task-level parallelism. For better understanding consider an array of ‘n’ elements, and if we want to calculate the maximum value, minimum value, average value and sum of all numbers of the array than we call the array of n elements for every task which means we are accessing same data four different times. This kind of process, using the same data several times for different tasks is task-level parallelism.

2.3.2 Data-level parallelism

Assigning different threads to perform the same task on different data is called data-level parallelism [11]. In the real world example if we assign multiple workers for the
2.4. Supercomputers

The development of GPUs and parallel computing led to the generation of supercomputers [12]. Supercomputers are the machines which are integrated with high compatible CPUs and a large number of GPUs which gives high performance compared to general-purpose computers [13]. Supercomputers are used in many fields like biomedical research, genetic research, physics simulations, aerospace simulation, etc. [14]. Mainly these fields because these fields need faster results and the data, attributes, and variables that to be calculated are more. For example, if we consider a rocket launch, we need to have accurate results because a small point can make a difference of thousands of miles in space. Some of the fastest computers in the world are the summit, Sierra, Sunway TaihuLight, etc. [15].

The system which works on different platforms is called a heterogeneous system [16, 17]. There are many devices that have multiple combinations of different processors like GPUs, CPUs, FPGAs, etc. To run a program across these platforms there are different interface frameworks like OpenCL [2], CUDA [18] etc.


2.5 OpenCL

OpenCL (open computing language) is a framework which facilitates parallel programming [20]. Khronos group enumerated OpenCL [21]. The hardware like Nvidia, AMD, Intel, etc. provides the SDK files with inbuilt header files and libraries which accommodate the OpenCL implementations. For writing OpenCL code for GPU devices, there are separate kernels which are included in the application called as the host application. OpenCL implementations use the host applications to communicate with in the system. The implementation of any OpenCL code starts by selecting the device platform like AMD, Nvidia, etc. which support the OpenCL drivers. For executing the instructions on a device, a context is needed which allocates the temporary storage space for different OpenCL objects such as buffers, kernels, command queue instructions which run on the Device. For compiling the program on the Device, the main code is sent to OpenCL drivers then the Kernel in the program is launched. A kernel is a function that returns void, and as a normal function, we can also give parameters to the kernel function. It is initialized by using the attribute __kernel. For transferring the data to the Kernel and collecting the results from the Kernel after data execution, we use buffer objects which are enqueued in the command queue, which gives access to the Kernel for reading and writing the data. A pointer variable can access buffers if the Buffer is set as an argument to the Kernel. OpenCL is a heterogeneous computing programming language run on CPUs, GPUs and Accelerator devices. It is used for processing a large amount of data for the best performance. It facilitates the programming languages based on C99 and C++11. The modules that make OpenCL to work on all devices are:

SDK
The tools and resources required to write a program using OpenCL are placed in the SDK(software development kit). There are different SDKs for different hardware devices. For example, Nvidia’s SDK called a CUDA Tool kit contains header files and libraries that link the basic programming languages to OpenCL, and some devices even provide the sample codes in SDK. So SDK has every tool required for writing and executing the program.

API
The application programming interface(API) assures that all devices are capable of implementing OpenCL and is defined by the Khronos Group. This API provides C header files required for a software developer, and this API set can be obtained from the Khronos website. Khronos later extended header files to C++ and other existing languages, but they are not controlled by Khronos. The class diagram of OpenCL specified by Khronos group is shown in Figure 2.7.

OpenCL C
OpenCL C language is used to write kernels in an individual source file. These files are read by applications during runtime and transferred to the compiler where the kernel
code is changed into binary. These kernels can have any number of functions, but it should have a minimum of one function which is the entry point to the Kernel and identified by __kernel.

**Driver**

In any computer, the communication between the operating system and the hardware devices is done by a bunch of files called drivers. So to implement OpenCL and communicate between functions. There is a driver called OpenCLs core which compiles kernels and communicates. Every Device has there own drivers provided by there respective hardware developing companies.

**Platform**

The SDK is installed on the system or computer (The combination of devices, context, and queue). In the modern world, every computer has a minimum of a CPU and GPU, and many GPUs are produced by different companies like Nvidia, AMD, Intel, etc. These have different OpenCL implementations that are handled by the platform. The software programmer can get the information of the number of platforms in a system by using the function clGetPlatformIDs and also the number of devices and device properties can be obtained by using the functions clGetDeviceIDs and clGetDeviceInfo respectively.
OpenCL setup
Before the execution of kernels on the Device, we need to set up memory buffers, which allocates required storage space for input and output data. This is done by context which is created using function `clCreateContext`. In this context, we can also set the properties. As context holds the data, we use one more command called command queue, which requests the actions from the Device. After all the requests are completed, we use another function `clfinish`, which synchronizes the data on the Device.

**Device:** The processor or hardware cores on which OpenCL runs like CPU, GPU, FPGA, etc.

**Kernel:** The function that does the work on Device and It is single instruction multiple data execution [22].

**Command queue:** Sends the commands to the Device from the host to execute instructions.

**Buffer:** Allocated memory space on a device is a buffer. These are created by calling `clCreateBuffer` function from the host. The memory transfer between the host and device can be done by using buffer functions `clEnqueueReadBuffer` and `clEnqueueWriteBuffer`.

Kernel creation and execution
Once the OpenCL environment is set up, the next step is to execute the device code. OpenCL code is written in the source file as normal code, but during runtime, the application reads the source file and sends it to `clCreateProgramWithSource` function. This program is built on the available devices on the system using `clBuild` function. Once the program is built, then the kernels are created by using `clCreateKernel` function. A kernel is a normal function that can be executed on a parallel architecture device. To execute the Kernel the user/programmer has to define the size of the task called NDRange(n-dimensional range) which is the size of input or output.

**Barrier** It acts as a bridge between workgroups. Every work item in a workgroup executes barrier function before interacting with other workgroups behind the barrier. It mainly helps while using if conditions and for loop conditions. In OpenCL barrier function is used by calling the barrier function with parameters `CLK_LOCAL_MEM_FENCE` and `CLK_GLOBAL_MEM_FENCE` while using local memory and global memory respectively [20]. For example, we are adding three numbers say 4, 8, and 16. In the first step, a work item adds 4 and 8 and stores the result in the previous index of 8. So, $4 + 8 = 12$ is stored in place of 8. In the second step, another work item takes this 12 and adds to 16, i.e., equal to 28 and stores the result in place of 16. If we do not put the barrier between the first and second steps then in the second step the work item takes the initial value of the 1st index, i.e., 8 and adds it to 16 which gives the wrong result.
of 24. So, in any parallel programming, the barrier function needs to be placed wherever needed to get the output.

Memory:

OpenCL standardized a memory model that suits the OpenCL API. So that all the companies that are manufacturing the parallel devices can follow the standard pattern. The specified model is shown in Figure 2.8. There are different parts of the memory model. They are:

- **Compute unit** The combination of local memory and work group.

- **Global/Constant memory** The largest memory in the device is global memory. This memory is used for copying data into and from the device. Inside a kernel, every variable directing to global memory has to be added with `__global` attribute at stating point. The buffers are issued in this memory. The constant memory is a part of global memory which stores the data that can only be read but not modified. Every variable directing to constant memory has to be added with `__constant` attribute at staring of the variable in a kernel.

- **Local memory** The memory of a single compute unit. This memory is also called shared memory as the memory is shared between work items in a workgroup. Local memory can be considered as a cache of global memory which is accessed faster. The variables directing to local memory have to be added with `__local` attribute in a kernel. Local memory stores the data required for work items in a workgroup.
Table 2.1: Difference between OpenCL and CUDA

<table>
<thead>
<tr>
<th>S.NO</th>
<th>OpenCL</th>
<th>CUDA</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Work item</td>
<td>Thread</td>
</tr>
<tr>
<td>2</td>
<td>work group</td>
<td>Thread block</td>
</tr>
<tr>
<td>3</td>
<td>private memory</td>
<td>local memory</td>
</tr>
<tr>
<td>4</td>
<td>local memory</td>
<td>shared memory</td>
</tr>
<tr>
<td>5</td>
<td>process element</td>
<td>CUDA core</td>
</tr>
<tr>
<td>6</td>
<td>compute unit</td>
<td>multiprocessor</td>
</tr>
</tbody>
</table>

- **Private memory** The memory of the individual work item. The data stored in the private memory of a work item cannot be seen by the other work items.

- **Host memory** The memory of CPU. In this memory, all the data and results are stored, and the data needed for devices are transferred from here.

Most of the above-mentioned OpenCL terms and memory patterns are similar for CUDA, but there is some difference in terminology [23] that are shown in Table 2.1.

The other differences are:

1. CUDA is built only for NVIDIA platform and it is a closed source working only on NVIDIA devices whereas OpenCL is open-source and can work on all platforms.

2. In CUDA the device is GPU so it launches GPU directly whereas in OpenCL the device can also be CPU and it is chosen and launched by the user.

CUDA has built-in libraries, and it runs faster than OpenCL, but in this paper, we use OpenCL because it is compatible with all devices. The steps in OpenCL programming are:

1. Initialize data from CPU.
2. Transfer data from CPU context to GPU context.
3. Launch the kernel with block dimensions.
4. Transfer results back to CPU from GPU.
5. Free the memory allocated.

Every OpenCL code has a sequential code and a parallel code. When we write the code, there will be two main parts of code:

1. **Host code**
   The sequential code runs on the CPU. The code which runs on CPU is called the host code. The kernel launch is also done by the host code.
2. Device code
The parallel code runs on the GPU. The code which runs on GPU is called device code.

The difference between the sequential code and parallel code is mainly allocating more number of work items for a task in place of the loop so that the execution time is faster than sequential execution time. For example, consider the addition of two vectors of length 'n'. The difference in the code is shown in Figure 2.9.

In sequential code and in Figure 2.11 the loop runs 'n' times which means a single work item executes 'n' times. If we consider the time taken for running the loop once is one second, then the total time taken is 'n' seconds. But for the parallel code, we assign 'n' work items. So all the 'n' work items work together and give the output in one second as shown in Figure 2.10 which is 'n' times faster. In the process of making efficient use of work items, the sequential code algorithms are modified for parallel programming, and one of them is scan.
Figure 2.10: Parallel execution with n work items

Figure 2.11: Flow chart for sequential execution
2.6 Scan

Scan is the prefix sum of an array. If you take a set of data elements \([a0, a1, a2, a3, \ldots]\) with binary function \(\oplus\) and apply prefix sum scan, exclusive scan output will be \([i, a0, a0 \oplus a1, a0 \oplus a1 \oplus a2, \ldots]\) and inclusive scan output will be \([a0, a0 \oplus a1, a0 \oplus a1 \oplus a2, \ldots]\). Here 'i' is the identity element. The identity element will be set to zero if \(\oplus\) is addition and it will be one for multiplication [25]. In exclusive scan, each element is the result of the reduction of all elements up to but does not include current element itself in the input array. In inclusive scan, all elements are summed including current element.

For example: If \(\oplus\) is addition, then the prefix sum(Scan) operation on the array \([1, 2, 3, 4, 5]\) would return
exclusive scan \([0, 1, 3, 6, 10]\)
inclusive scan \([1, 3, 6, 10, 15]\)

To perform the scan, the binary operator \(\oplus\) should be associate i.e \(a \oplus b = b \oplus a\).
For example consider \(a = 2, b = 4\) then
1. \(2 + 4 = 4 + 2\), '+' is associative.
2. \(2 \times 4 = 4 \times 2\), '*' is associative but,
3. \(2 / 4 \neq 4 / 2\), '/' is not associative.

In the next chapter, we see the evolution and related work of scans.
2. Background
3. Types of Scan

In this chapter, we see the progression of scans, how scans have become efficient, and also we discuss some of the scan methods in detail. The basic types of scan are sequential scan and parallel scan.

3.1 Sequential Scan

In a sequential scan, the first element of the input array will be the first element of the output array and the second element of the output is the sum of first output element and the second input element than on loop continues until the last element of the input array. For example, consider an array consisting of one to eight numbers. The process of execution goes by adding one number to the other in sequence, as shown in Figure 3.1. To compute the result of an array of length eight, it took eight steps. So to reduce the steps, we parallelize the scan based on Hillis and Steele algorithm [26] i.e., Naive Parallel Scan.

3.2 Parallel Scan

Until now, we have seen the scans which use single work item for executing the task but from now we employ more work items by parallelizing the scans.

3.2.1 Naive Parallel Scan

Naive Parallel Scan [3], in this scan in the first step, we add the adjacent elements and store them in an array. In the second step, we add the adjacent element of adjacent elements by considering the offset equal to one. i.e., now the offset is two, and this process is done until the final result is obtained. The example is shown in Figure 3.2. In this example, the length of the array is eight, and the number of steps taken to
compute is three. So to compute $2^n$ elements, it takes \( n \) steps. So the step complexity of Naive Parallel Scan is $O(\log n)$. The efficiency of any scan is mainly determined by work complexity and step complexity.

**Step complexity**: The number of steps taken to execute a task is called step complexity.

**Work complexity**: The number of operations done in executing a task is called work complexity.

In the naive parallel scan, the step complexity is less compared to the sequential scan. But the work complexity is more. In the example an array of eight elements it made 24 operations which is not efficient. When the data is large, there will be a conflict between the synchronization of work items \[3\]. To overcome this Blelloch proposed a scan called as reduce/down-sweep scan or Blelloch scan.

### 3.2.2 Blelloch Scan

Blelloch scan is also known as reduce and down sweep scan. In this, the scan of $n$ elements has two phases reduce phase and down-sweep phase. Each phase undergoes $\log(n)$ parallel steps with an overall work complexity $O(n)$ \[4\]. The algorithms for both phases are shown below. The figure of the reduce algorithm is Figure 3.3 and the figure of the down-sweep phase is Figure 3.4 \[25\]. A binary tree that has $n$ leaves has $\log_2 n$ levels, and each level has $2^d$ nodes and ‘$d$’ is the depth of the level. If you perform one add per node, then we perform $O(n)$ adds on a single phase of the tree. We use the up-sweep phase to calculate the intermediate sums of the input array.
3.2. Parallel Scan

Figure 3.2: The Naive Parallel scan

Figure 3.3: The reduce (up-sweep) phase scan.

Algorithm 1 Reduce Phase

1: for d=0 to \( \log_2 n - 1 \) do
2:   for all k=0 to n-1 by \( 2^{d+1} \) in parallel do
3:     \( x[k + 2^{d+1} - 1] \leftarrow x[k + 2^d - 1] + x[k + 2^{d+1} - 1] \)
4:   end for
5: end for
In the reduction phase, we take the input array and put in the bottom as shown in the Figure 3.3 and perform iterations upwards. Elements in the initial stage (d=3) called leaves. The purpose of the reduction phase is to find out the total sum of the elements in the array. After the reduction phase is completed we have the sum of all elements in the final index element of the array, we call this element as a root node. In the first iteration (d=2) we sum up two leaves, ith and (i-1)the leaf i.e two odd indices in the array. After the first iteration is completed we have the summation of the first two elements in index one and the element in index three is the sum of index 2 and 3 and so on. In the second iteration (d=1) the indices which have 4n-1 will perform the operation. For example, if n=1 then 4n-1 is equal to 3 so the third index will have the summation of indices 3 and 1 from the previous step. In the third iteration indices with 8n-1 will perform as we have only eight elements in the shown figure the reduction phase is completed. Now, we have the summation of all elements in the last index. Every element used for calculating the intermediate result is the child of that node. Now, we go for the down-sweep phase.

Algorithm 2 The down-sweep Phase

1: \( x[n - 1] \leftarrow 0 \)
2: for \( d = \log_2 n - 1 \) down to 0 do
3: \hspace{1cm} for all \( k=0 \) to \( n-1 \) by \( 2^{d+1} \) in parallel do
4: \hspace{2cm} \( t \leftarrow x[k + 2^d - 1] \)
5: \hspace{2cm} \( x[k + 2^d - 1] \leftarrow x[k + 2^{d+1} - 1] \)
6: \hspace{2cm} \( x[k + 2^{d+1} - 1] \leftarrow t + x[k + 2^{d+1} - 1] \)
7: \hspace{1cm} end for
8: end for

Figure 3.4: The down-sweep phase scan
In down sweep scan, we take the output array from the reduce phase and initialize the root element to zero then in the first iteration we pass the root element to its left child and set the root right child to the sum of the root element and the value of a left child in the previous step. For the second iteration, we have to do the first iteration process for two indices, as shown in Figure 3.4(d=2). In the third iteration, we repeat the process for four indices. This is where the down sweep scan ends. Now, we get the results of the scan. These are for an exclusive scan, to make it inclusive, we remove the first element and add one more element at the end of the array.

The inclusive scan also has reduce phase and down-sweep phase. The algorithm for reduce phase is the same as exclusive, but the down-sweep phase is different from the exclusive scan. As the reduction phase is equal, we take the output of the exclusive reduction phase and proceed for the down-sweep inclusive scan. In the first iteration, we add the elements which are two offset’s away from each other. In the next iteration, we add the elements which are away from half of the previous offset. Now, we have the final inclusive prefix sum of every element.
3. Types of Scan
4. Segmented scan

In some applications, we deal with many small scans rather than a large scan. In this case, it is not efficient to launch a kernel for every small scan. So we combine the small scans to form a large scan as segments. The segments are identified by the second array, which is called a flag vector or flag array. Flag vector stores the starting element of the segment as one and the remaining elements as Zero. Now the kernel is launched after the execution is done. The results are split as individual segments with the help of the second array. This type of scan is called a segmented scan [6]. For example:

\[
\begin{align*}
\text{VAL} &= (5 \ 1 \ 3 \ 4 \ 3 \ 9 \ 2 \ 6) \\
\text{FLAG} &= (1 \ 0 \ 1 \ 0 \ 0 \ 0 \ 1 \ 0) \\
\text{Inclusive-scan}(\text{VAL,FLAG}) &= (5 \ 6 \ 3 \ 7 \ 10 \ 19 \ 2 \ 8)
\end{align*}
\]

The segmented scan is efficient than individual scans because it reduces the step complexity. For example, if we have m arrays, each of which has n numbers. Independently Scanning them takes \(O(m \log n)\) step complexity. If you put them into a single array with \(m*n\) numbers and do a segmented scan, it takes only \(O(\log(m*n))\) step complexity. The segmented scan is also used in computer graphics, as it mainly works on matrix multiplication [27], machine learning. In machine learning, most methods depend on matrix operations [28]. Many engineering applications use the matrix to store data mainly in the sparse matrix.

Consider an input array of \(n\) elements that produce an output of \(n\) elements then the work complexity is equal to the number of operations done to get output from the input including copying the element and step complexity is the number of steps took to get the output. The step complexity and work complexity of different scans are shown in [Table 4.1].
4. Segmented scan

<table>
<thead>
<tr>
<th>Type of scan</th>
<th>Work complexity</th>
<th>Step complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequential scan</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>Parallel scan</td>
<td>$O(n \log n)$</td>
<td>$O(\log n)$ [considering there are $n$ processors]</td>
</tr>
<tr>
<td>Segmented scan</td>
<td>$O(n \log n)$</td>
<td>$O(\log n)$ [considering there are $n$ processors]</td>
</tr>
</tbody>
</table>

Table 4.1: work and step complexity

4.1 Implementation setup

The implementation results are generated on Intel(R) Core(TM) i5-2500 CPU with 3.30GHz and the 64-bit processor. The operating system we used is Ubuntu 16.04 version. We used the GCC compiler to compile the code and NVIDIA GPU. The properties of GPU we used are shown in Figure 4.1

Figure 4.1: Specifications of GPU

NVIDIA Corporation GK107 [GeForce GT 640] (rev a1) (prog-if 00 [VGA controller])

Flags: bus master, fast devsel, latency 0, IRQ 36
Memory at fa000000 (32-bit, non-prefetchable) [size=16M]
Memory at c0000000 (64-bit, prefetchable) [size=256M]
Memory at d0000000 (64-bit, prefetchable) [size=32M]
I/O ports at e000 [size=128]
[virtual] Expansion ROM at fb000000 [disabled] [size=512K]
Capabilities: <access denied>
Kernel driver in use: nvidia
Kernel modules: nvidiafb, nouveau, nvidia_384_drm, nvidia_384

4.2 Implementation of Blelloch segmented scan

The segment scan algorithm has a slight change from scan due to the addition of the flag vector, which identifies each segment or individual input array. The segment scan algorithms are Algorithm 3 and Algorithm 4.

Segmented scan implementation differs from scan implementation as the algorithms differ. For example, consider an array of eight numbers that have three sets of segments (1,2), (3,4), (5,6,7,8). In segmented scan implementation, segments are denoted by a head flag vector which is in equal size of the input vector.
Algorithm 3 Reduce Phase of Segmented scan

1: for \( d=1 \) to \( \log_2 n - 1 \) do
2:   for all \( k=0 \) to \( n-1 \) by \( 2^{d+1} \) in parallel do
3:     if \( f[k+2^{d+1}-1] \) is not set then
4:       \( x[k+2^{d+1}-1] \leftarrow x[k+2^d-1] + x[k+2^{d+1} - 1] \)
5:       \( f[k+2^{d+1}-1] \leftarrow f[k+2^d-1]f[k+2^{d+1} - 1] \)
6:     end for
7:   end for

<table>
<thead>
<tr>
<th>input array</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>flag vector</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>segments</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
</tbody>
</table>

For every first element in the input vector, the corresponding element in the head flag vector is 1, and the leftover elements are 0. With the help of the head flag vector, we can always distinguish the start and end of the segment when you go from left to right of the vector. Before starting the execution process, we store the copy of the flag vector as the original flag because, during the execution process, we update the flag vector, and we also need the original values. In the implementation, we start with the reduction phase, where we take the input array \( (d=2) \) and perform iterations; every element in the input array is called a leaf. In the first iteration \( (d=1) \) we sum up two leaves \( i \) th and \( (i-1) \) th leaf i.e., two odd indices in the array but for every addition, we check the flag vector so that the elements in one segment will not be added to other segment elements. After the first iteration is completed, we have the summation of the first two elements in index one and the element in index three is updated with the summation of index 2 and 3, and so on.

\( d=2 \)

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |

\( d=1 \)

| 1 | 1+2 | 3 | 3+4 | 5 | 5+6 | 7 | 7+8 |

In the second iteration, the indices which have \( 4n-1 \) will perform the operation. For example, if \( n=1 \) then \( 4n-1 \) is equal to 3, so the third iteration should have the summation of indices 3 and 1 from previous step but during the second iteration when the compiler checks the flag vector where it knows that index 1 and 3 belongs to different segments so the summation will not happen and the values in these indices remain same as previous one and In the third iteration indices with \( 8n-1 \) will perform the task and these iterations continue depending on the data size. By the end of the reduction phase, we will have the summation of all elements of the last segment in the array stored in the last index, which is called as root. Every element used for finding the intermediate
result is the child of the node where the result of these elements stored. Now, we go for the down-sweep phase.

\[
d=1 \quad 1 \quad 1+2 \quad 3 \quad 3+4 \quad 5 \quad 5+6 \quad 7 \quad 7+8
\]

\[
d=0 \quad 1 \quad 1+2 \quad 3 \quad 3+4 \quad 5 \quad 5+6 \quad 7 \quad 5+6+7+8
\]

**Algorithm 4** The down-sweep Phase of Segmented scan[@3]

1: \( x[n-1] \leftarrow 0 \)
2: \( \text{for} \) \( d = \log_2 n - 1 \) \( \text{down to} \) 0 \( \text{do} \)
3: \( \text{for} \) all \( k=0 \) to \( n-1 \) by \( 2^{d+1} \) in parallel \( \text{do} \)
4: \( t \leftarrow x[k + 2^d - 1] \)
5: \( x[k + 2^d - 1] \leftarrow x[k + 2^{d+1} - 1] \)
6: \( \text{if} \) \( f[k+2^d] \) \( \text{is set then} \)
7: \( x[k + 2^{d+1} - 1] \leftarrow 0 \)
8: \( \text{elseif} \) \( f[k+2^d-1] \) \( \text{is set then} \)
9: \( x[k + 2^{d+1} - 1] \leftarrow t \)
10: \( \text{else} \) \( x[k + 2^{d+1} - 1] \leftarrow t + x[k + 2^{d+1} - 1] \)
11: \( \text{Unset flag} \) \( f[k + 2^d - 1] \)
12: \( \text{end for} \)
13: \( \text{end for} \)

In down sweep phase, we take the output array from the reduce phase and initialize the root element to zero then In the first iteration \( (d=0 \text{ to } d=1) \) we pass the root element to its left child and set the root right child to the sum of the root element and the value of the left child in the previous step but if the child elements are in different segments then we only update the left child to Zero and the remaining elements will be the same.

\[
d=0 \quad 1 \quad 1+2 \quad 3 \quad 3+4 \quad 5 \quad 5+6 \quad 7 \quad 0
\]

\[
d=1 \quad 1 \quad 1+2 \quad 3 \quad 0 \quad 5 \quad 0 \quad 7 \quad 5+6
\]

In the second iteration \( (d=1 \text{ to } d=2) \), we do the first iteration process, and for every iteration, we check the flag vector so that elements do not interfere between segments this process runs until the final output is obtained. Now we have the prefix sum of every segment in the array \( (d=2) \).

\[
d=2 \quad 0 \quad 1 \quad 0 \quad 3 \quad 0 \quad 5 \quad 5+6 \quad 5+6+7
\]

Now we separate the result of each segment with the help of the head flag. The final result (exclusive scan) of three-segment arrays will be
4.2. Implementation of Blelloch segmented scan

1. \((1,2) \Rightarrow (0,1)\)
2. \((3,4) \Rightarrow (0,3)\)
3. \((5,6,7,8) \Rightarrow (0,5,11,18)\)

For implementing the above algorithm in OpenCL on GPU(device), We initialize the input vector and flag vector, and we create an empty vector of the size of the input vector, which stores the output. Once the initialization part is done, we create the buffers that allocate the space required for the data variables in GPU then we transfer the data from CPU to GPU using write buffer. Now, we create a kernel function as shown in Figure 4.2 which holds the instructions that to be executed on GPU. Once the kernel is created, we launch it by providing the NDRange. The kernel executes the instructions on GPU data and stores the result in GPU itself. The results are sent from GPU to CPU by using the Read buffer.

```c
std::string kernel_code =
" void kernel segmentscan_kernel(global int* n, int offset = 1; 
" if (flag[thid] == 1 || flag[bi]) 
" n
" C[bi] += C[ai];
")
" offset *= 2; 
"

Figure 4.2: Reduce phase kernel function
4. Segmented scan
5. Sparse Matrix

In this chapter, we use the sparse matrix-vector multiplication method to produce the output as the prefix sum. But to get the same output, we make some changes to the SpMV program by removing the product operation and changing it to an addition operation.

A sparse matrix is such a matrix that have the majority of their elements equal to zero. For performing a mathematical operation between two sparse matrices or one sparse matrix and a matrix, like multiplication, addition, etc., it is simpler to get the result because we can neglect the zero values as operating on them doesn’t change anything. A sparse matrix is one of the efficient and advanced methods used to implement most segmented operations because it reduces the algebraic operations and also reduces the cost in iterative methods for solving significant mathematical problems that occur in many scientific applications\[29\]. The sparse matrix is stored in a database using several techniques like CSR(Compressed sparse row), COO(Coordinate format), etc. \[30\].

**CSR format:**

CSR format consists of three arrays, in which *data array* stores non zero values row by row, *indices array* stores the column indices of every value stored in the data array and the *ptr array* stores the starting point of indices of each row \[7\]. For example, consider a matrix :

\[
A = \begin{bmatrix}
1 & 7 & 0 & 0 \\
0 & 2 & 8 & 0 \\
5 & 0 & 3 & 9 \\
0 & 6 & 0 & 4
\end{bmatrix}
\]

\[
ptr = [0 \ 2 \ 4 \ 7 \ 9]
\]
5. Sparse Matrix

Figure 5.1: CSR format

\[
\begin{align*}
\text{indices} &= [0 \ 1 \ 1 \ 2 \ 0 \ 2 \ 3 \ 1 \ 3] \\
\text{data} &= [1 \ 7 \ 2 \ 8 \ 5 \ 3 \ 9 \ 6 \ 4]
\end{align*}
\]

**COO format:**

The COO format is the same as the CSR format with a slight change. Instead of \textit{ptr array}, we have \textit{row array} which stores the row index of every value stored in the \textit{data array} \cite{7}. The data storage required is always three times the number of non zero elements in a matrix. For example, consider a matrix:

\[
A = \begin{bmatrix}
1 & 7 & 0 & 0 \\
0 & 2 & 8 & 0 \\
5 & 0 & 3 & 9 \\
0 & 6 & 0 & 4
\end{bmatrix}
\]

\[
\text{row} = [0 \ 0 \ 1 \ 1 \ 2 \ 2 \ 2 \ 3 \ 3] \\
\text{indices} = [0 \ 1 \ 1 \ 2 \ 0 \ 2 \ 3 \ 1 \ 3] \\
\text{data} = [1 \ 7 \ 2 \ 8 \ 5 \ 3 \ 9 \ 6 \ 4]
\]

From both formats, we can see that the CSR format needs less storage space than COO because \textit{ptr array} size is less compared to \textit{row array} size in COO format. The CSR format is fast in query computing in the database and always ready for execution. Due to this CSR format is used in many computations like sparse matrix-vector multiplication \cite{31}. Sparse matrix-vector multiplication(SpMV) is used in many scientific and engineering applications. The SpMV is highly capable of calculating eigenvalues, large linear systems, and implementing LAPACK \cite{32}.

5.1 Sparse Matrix-Vector Multiplication

In sparse matrix computations, matrix multiplication is the most commonly used one \cite{33}. Let us consider a sparse matrix \( A \) and a column matrix \( X \). Now we multiply \( A \) and
X and store the result in Y. i.e Y = AX. The kernel function is shown in Figure 5.2.

For example:

\[ A = \begin{bmatrix}
1 & 7 & 0 \\
0 & 2 & 8 \\
5 & 0 & 3
\end{bmatrix} \]

\[ X = \begin{bmatrix}
x_0 \\
x_1 \\
x_2
\end{bmatrix} \]

\[ Y = \begin{bmatrix}
y_0 \\
y_1 \\
y_2
\end{bmatrix} \]

\[
\begin{bmatrix}
y_0 \\
y_1 \\
y_2
\end{bmatrix} = \begin{bmatrix}
1 & 7 & 0 \\
0 & 2 & 8 \\
5 & 0 & 3
\end{bmatrix} \begin{bmatrix}
x_0 \\
x_1 \\
x_2
\end{bmatrix} \quad (5.1)
\]

\[ ptr = [0 2 4] \]

\[ indices = [0 1 1 2 0 2] \]

\[ data = [1 7 2 8 5 3] \]

```c
__kernel void
spmv_csr_scalar_kernel (__global const int num_rows ,
__global const int * ptr ,
__global const int * indices ,
__global const float * data ,
__global const float * x ,
__global const float * y)
{
int row = get_global_id(0);
if( row < num_rows ){
float dot = 0;
int row_start = ptr [ row ];
int row_end = ptr [ row +1];
for (int jj = row_start ; jj < row_end ; jj ++)
dot += data [jj] * x[ indices [ jj ]];
y[ row ] += dot ;
}
```

Figure 5.2: kernel function of SpMV
In this example we took the CSR format to store the data of matrix A. In the first step of execution the each work item is assigned to each row of matrix A. This work item takes the first row element of input matrix and multiply the element with first column element of the matrix x and stores the result in dot product then it takes the second row element of the input array and multiply with the second column element of the matrix x and add this result to the dot product and store the overall result to dot product. This process runs until the elements in the first row of the matrix is completed. As we assigned work items for all the rows, the above process is done by every work item parallelly. Once all the work items complete the execution, the resultant value stored in each work item is collected.

\[
\text{product} = [ \ x_0, 7x_1, 2x_1, 8x_2, 5x_0, 3x_2 \ ]
\]

\[
\text{product} = [ \ (x_0, 7x_1)(2x_1, 8x_2)(5x_0, 3x_2) \ ]
\]

\[
\text{product} = [ \ (x_0, x_0 + 7x_1)(2x_1, 2x_1 + 8x_2)(5x_0, 5x_0 + 3x_2) \ ]
\]

\[
Y = [ \ x_0 + 7x_1, 2x_1 + 8x_2, 5x_0 + 3x_2 \ ]
\]

The Result:

\[
y_0 = x_0 + 7x_1 \\
y_1 = 2x_1 + 8x_2 \\
y_2 = 5x_0 + 3x_2
\]

### 5.2 Implementation of Segmented Scan Using Sparse Matrix

In the implementation, we consider CSR format for storing the data of the matrix. The advantage of using CSR format is, it only stores the row start in the pointer during execution row-end takes the value of next row-start. So the memory required is less, and the other major option is we completely neglect the indices matrix because we consider each row as a segment, and we need all elements in the row. So we need only two arrays. They are data array and ptr array as we aim to evaluate the blelloch segmented scan and sparse matrix segmented scan. We take the same data which is used for implementing the blelloch segmented scan as input. Now, all the input data is in the CPU. For parallel execution, the data is transferred to the device(GPU), and the kernel is launched by allocating the work items needed (one work item for one row).

In the kernel function we assigned the index of the work items to the row attribute, and also we have provided the kernel with the number of segments in the form of the number of rows. Now, for every row value, the compiler checks whether the row value is less than the number of rows or not ( if(row< num_rows) ). If the number of rows is less than the row value then the compiler takes two more attributes row-start and
row end where row-start stores the value stored in row index of ptr array (row-start = ptr[row]) and row-end stores the next value stored in next row index (row-end = ptr[row+1]). Now every work item is executed through a for loop ((for j = row-start; j < row-end; j++)) where the initial value is taken as row-start, and the end value is taken as row-end. Every time the loop has executed the value stored in the respective index of the input array is taken and added to the previous input array value. This loop runs until the index becomes row-end. These steps are done by every work item on their respective rows and store the result in the device (GPU). Finally, we transfer the result back to the CPU and clear the data in the GPU. As we already placed each segment as a row, we can directly take the result from the row because the result of the rows is stored in their respective output rows. The kernel function is shown in Figure 5.3.

```c
if( row < num_rows ){
    float dot = 0;
    int row_start = ptr[ row ];
    int row_end = ptr[ row +1 ];
    for (int j = row_start ; j < row_end ; j++)
    {
        if(j== row_start)
        {
            y[j]=0;
        }
        else
        {
            dot += input[ j ];
            output[ j ] = dot;
        }
    }
}
```

Figure 5.3: kernel function
6. Evaluation

In this chapter, we have evaluated the blelloch segmented scan and segmented scan using sparse matrix method with respect to the segment length and run time for particular data size. We plotted the graphs based on the results obtained from implementing both the methods. For each graph, we have ten repetitions, and the average value of all the repetitions are taken. We have plotted four different graphs, line graph, column graph, data transfer graph, and kernel execution graph for better understanding. We plotted the graph by taking segment length on the x-axis and run time on the y-axis.

6.1 Line Graph

The line graph shown in Figure 6.1 is plotted for the results obtained for the data set having $2^{10}$ elements then splitting the data elements into different segment lengths. In this plot, we took the minimum length of the segment as two and the maximum has 512. As we maintained the size of data set constant when we took small length segments we got more number of segments, when we took 2 elements per segment we had 512 segments and when we took 4 elements per segment we had 256 elements. The product of the number of segments and length of segment gives the size of data. From the graph shown in Figure 6.1 we can say that all the segments have taken the same amount of time irrespective of the segment length because the size of the data set is constant and we cannot make a conclusion from this graph as the lines in the graph are overlapping. So, we go for column graph.

6.2 Column Graph

The column graph shown in Figure 6.2 is plotted for each segment where the run time for each segment length is individually obtained. In this graph, we can see some clear differences compared to the line graph that the sparse matrix is taking less time than the blelloch segment scan. As we plotted this graph for the total run time which is time
taken for initialization of data, creating the kernels, transferring the data, executing the kernel, and transferring back the results. We cannot clearly say where the time difference is caused, so we plot two more graphs only taking the data transfer and kernel execution so that we can have a better understanding.

6.3 Data Transfer Time Graph

The data transfer graph shown in the Figure 6.3 is plotted by taking the fixed segment length and increasing the number of segments. when we are increasing the number of segments that also means we are increasing the data size. In this graph, we can see the time taken for transferring the data from CPU to GPU. As the number of segments is increasing the time taken for data transfer is also increasing in both methods but the time taken by the sparse matrix method is less compared to the blelloch segmented scan until hundred segments after that the both methods are taking same amount of time for data transfer.
6.3. Data Transfer Time Graph

**Segment length vs Run time**

---

**Figure 6.2: Column Graph**

**Data Transfer Time**

---

**Figure 6.3: Data Transfer Time Graph**
6.4 Kernel Execution Time Graph

The kernel execution time graph is shown in Figure 6.4. The time taken by each method for executing the kernel function is plotted. By seeing the graph, we can say that the kernel execution time of the sparse matrix method is very less compared to the blelloch segmented scan.

![Kernel Execution Time Graph](image)

Figure 6.4: Kernel Execution Time Graph

In the thesis, we expected that the sparse matrix segmented scan would give better results because of the time taken for transferring the data will be less compared to the time taken in blelloch segmented scan. But from seeing the evaluation results of all four graphs based on segment length and run time, we found that the time taken for kernel execution of sparse matrix segmented scan is less than the blelloch segmented scan due to the instructions given for implementing the task are less in sparse matrix segmented scan.
7. Conclusion

In recent years, most of the scientific development companies are dealing with the faster results of mathematical calculations, so the companies are using highly parallel machines that indeed work on parallel computing. With the development of parallel machines, the importance of parallel computing is also growing, and the new applications based on parallel programming and GPUs are increasing. The optimization of algorithms used for parallel programming has become a major task. In this evolution process, many efficient algorithms are developed. Some algorithms are based on optimizing the memory patterns, and some are addressing the execution run time issues. In the process of optimization, we focused on using one of the efficient ways of storing data, i.e., sparse matrix storage method for generating blelloch segmented output. In this thesis, we maintained the data set size not to cross maximum of $2^{10}$ elements because the methods and algorithms we choose are for small data because we are trying to test whether the sparse matrix can generate prefix segmented scan if so we evaluate them.

We have implemented the blelloch segmented scan and segmented scan using a sparse matrix. Then we evaluated the results by plotting the graphs between run-time and segment length. From the results, we have discovered that both methods take the same time. So, we conclude this thesis by suggesting that both the methods are good enough. Finally, we suggest that we can use the sparse matrix method also for prefix scans instead of the blelloch segmented scan.
8. Future Work

As we have implemented blelloch segmented scan and sparse matrix segmented scan on small data sets. For the future work:

- We would like to extend this for large data sets. In blelloch segmented scan, we divide the data set into data blocks which can be calculated by a single thread block. In sparse matrix instead of using single thread per row, we use total wrap (a group of 32 threads) and we can use the lane techniques[31]. we implement both these and evaluate their performance.

- We want to merge both the blelloch segmented approach and segmented scan using a sparse matrix method to form a new algorithm where we try to replace the flag vector with a pointer array in the blelloch segmented scan.
9. Appendix

In this section, the code of our implementation is included, for reference.

9.1 Kernel Code for Blelloch segmented implementation

Listing 9.1: Kernel implementation for Belloch segmented scan in OpenCL

```c
std::string kernel_code =
"void kernel segmentscan_kernel (global int* n,
  global float*A,
  global float*B,
  global float*C,
  global int*flago,
  global int*flag)
{
  //Get the index of the work-item
  int thid = get_global_id(0);
  int offset = 1;
  "for (int d = n>>1; d > 0; d>>=1
  "{"barrier(CLK_GLOBAL_MEM_FENCE);
  "if (thid < d)
  "{"int ai = offset*(2*thid+1)-1;
  "int bi = offset*(2*thid+2) -
```
```
\begin{verbatim}
" if (flag[bi] == 0) 
"{  
"C[bi] += C[ai];  
"}  
"flag[bi]=flag[ai] || flag[bi];  
"}  
"offset *= 2;  
"}  
"if (thid == 0) {  C[n-1] = 0;  }  
"for (int d = 1; d < n; d *= 2)  
"{  
" offset >>= 1;  
"barrier(CLOCK_GLOBAL_MEM_FENCE );  
"if (thid < d)  
"{  
"int ai = offset*(2*thid+1)-1;  
"int bi = offset*(2*thid+2)-1;  
"int ci= ai+1;  
"float t = C[ai];  
"C[ai] = C[bi];  
" if(flago[ci] == 1)  
"{  
"C[bi]=0;  
"}  
"else  
"{  
"C[bi] += t;  
"}  
"flag[ai]=0;  
"}  
"}  
"barrier(CLOCK_GLOBAL_MEM_FENCE );  
"B[2*thid] = C[2*thid ];  
"B[2*thid+1] = C[2*thid+1];  
"}
\end{verbatim}
9.2 Kernel Code for matrix segmented implementation

Listing 9.2: kernel code implementation for segmented scan using sparse matrix in OpenCL

```c
// kernal code;
std::string kernel_code =
" void kernel_matrix_kernel( global const int* alpha,
  global float *A,
  global int *B,
  global float *C)
{
  //Get the index of the work-item
  int row = get_global_id(0);
  if(row < alpha)
  {
    float D = 0.0;
    int row_start = B[row] ;
    int row_end = B[row+1];
    for( int j= row_start ; j < row_end; j++)
    {
      D += A[j];
      C[j] = D;
    }
  }\n};
";
```
Bibliography


