Institutionen för systemteknik
Department of Electrical Engineering

Examensarbete

Implementations of the FFT algorithm on GPU

Examensarbete utfört i Elektroniksystem
vid Tekniska högskolan vid Linköpings universitet
av

Sreehari Ambuluri

LiTH-ISY-EX--12/4649--SE

Linköping 2012

Linköpings universitet
TEKNISKA HÖGSKOLAN

Department of Electrical Engineering
Linköpings universitet
SE-581 83 Linköping, Sweden

Linköpings tekniska högskola
Linköpings universitet
581 83 Linköping

Implementations of the FFT algorithm on GPU

Examensarbete utfört i Elektroniksystem
vid Tekniska högskolan i Linköping
av

Sreehari Ambuluri

LiTH-ISY-EX--12/4649--SE

Handledare: **Dr. Mario Garrido**
ISY, Linköpings universitet

Examinator: **Dr. Oscar Gustafsson**
ISY, Linköpings universitet

Linköping, 21 December, 2012

Avdelning, Institution
Division, Department

Division of Electronic Systems
Department of Electrical Engineering
Linköpings universitet
SE-581 83 Linköping, Sweden

Datum
Date

2012-12-21

Språk
Language

☐ Svenska/Swedish
☒ Engelska/English
☐ _____

Rapporttyp
Report category

☐ Licentiatavhandling
☒ Examensarbete
☐ C-uppsats
☐ D-uppsats
☐ Övrig rapport
☐ _____

ISBN
—

ISRN
LiTH-ISY-EX--12/4649--SE

Serietitel och serienummer **ISSN**
Title of series, numbering —

URL för elektronisk version
http://www.es.isy.liu.se/
http://www.es.isy.liu.se/

Titel
Title Implementations of the FFT algorithm on GPU

Författare Sreehari Ambuluri
Author

Sammanfattning
Abstract

The fast Fourier transform (FFT) plays an important role in digital signal processing (DSP) applications, and its implementation involves a large number of computations. Many DSP designers have been working on implementations of the FFT algorithms on different devices, such as central processing unit (CPU), Field programmable gate array (FPGA), and graphical processing unit (GPU), in order to accelerate the performance.

We selected the GPU device for the implementations of the FFT algorithm because the hardware of GPU is designed with highly parallel structure. It consists of many hundreds of small parallel processing units. The programming of such a parallel device, can be done by a parallel programming language CUDA (Compute Unified Device Architecture).

In this thesis, we propose different implementations of the FFT algorithm on the NVIDIA GPU using CUDA programming language. We study and analyze the different approaches, and use different techniques to accelerate the computations of the FFT. We also discuss the results and compare different approaches and techniques. Finally, we compare our best cases of results with the CUFFT library, which is a specific library to compute the FFT on NVIDIA GPUs.

Nyckelord
Keywords GPU, CUDA, FFT.

Abstract

The fast Fourier transform (FFT) plays an important role in digital signal processing (DSP) applications, and its implementation involves a large number of computations. Many DSP designers have been working on implementations of the FFT algorithms on different devices, such as central processing unit (CPU), Field programmable gate array (FPGA), and graphical processing unit (GPU), in order to accelerate the performance.

We selected the GPU device for the implementations of the FFT algorithm because the hardware of GPU is designed with highly parallel structure. It consists of many hundreds of small parallel processing units. The programming of such a parallel device, can be done by a parallel programming language CUDA (Compute Unified Device Architecture).

In this thesis, we propose different implementations of the FFT algorithm on the NVIDIA GPU using CUDA programming language. We study and analyze the different approaches, and use different techniques to accelerate the computations of the FFT. We also discuss the results and compare different approaches and techniques. Finally, we compare our best cases of results with the CUFFT library, which is a specific library to compute the FFT on NVIDIA GPUs.

Acknowledgments

Firstly, I would like to express my sincere gratitude to Dr. Oscar Gustafsson for accepting me to do this work in the Department of Electrical Engineering. I would like to thank my supervisor Dr. Mario Garrido for giving me this opportunity and for his support, time, and concern. Under his guidance, I have learned to think in righteous way while solving the problems.

Secondly, I would like to thank Dr. Kent Palmkvist for directing me to find this opportunity. I would like to thank Gabriel Caffarena for giving tips at every stage of my work. I would also thank to Dr. Ingemar Ragnemalm and Jens Ogniewski for their suggestions and to clarify my doubts in GPUs and CUDA through out my thesis.

Thirdly, I would like to thank Peter Johonsson and Thomas Johonsson for helping me to make an experimental setup for my work.

Finally but not least, thanks to my parents, cousins, and friends to give me the moral support during my Master Studies.

Contents

List of Acronyms

ALU	Arithmetic Logic Unit
API	Application Programmable Interface
CG	Constant Geometry
CPU	Central Processing Unit
CUDA	Compute Unified Device Architecture
CUFFT	CUDA FFT library
DFT	Discrete Fourier Transform
DIF	Decimation In Frequency
DIT	Decimation In Time
DRAM	Dynamic Random Access Memory
DSP	Digital Signal Processing
FFT	Fast Fourier Transform
FPU	Floating Point Unit
FSM	Finite State Machine
GPU	Graphical Processing Unit
PCI	Peripheral Component Interconnect
RF	Register File
SFU	Special Function Unit
SIMD	Single Instruction Multiple Data
SIMT	Single Instruction Multiple Threads
SM	Streaming Multiprocessor

Chapter 1

Introduction

A graphical processing unit (GPU) is a processing unit that helps to accelerate the graphical computations and few other non graphical (general) computations in a modern day computers. It is used as a co-processor to a conventional CPU to speed up the computations [26], [27], [29]. A GPU may have hundreds of processing cores.

A CPU takes more computation time to process certain programs or tasks which have large number of iterations. Instead, a GPU can complete such tasks much faster. The reason behind the improved computational time is the larger number of cores or computational area compared to the CPU. The comparison of a CPU and a GPU is shown in Figure 1.1.

The fast Fourier transform (FFT) is an efficient algorithm to compute the discrete Fourier transform (DFT) and its inverse [1]. It involves a large number of computations. Many hardware designers have been working on implementing different implementations of the FFT algorithms on different devices, in order to accelerate the performance.

As there are a large number of computations in the implementation of the FFT algorithm and there is a large computational area in the GPU, we experiment the implementations of the FFT algorithm in a GPU device in this thesis work.

In this work, we propose different implementations of the FFT algorithm on NVIDIA GPU using Compute Unified Device Architecture (CUDA) parallel programming language. We study and analyze the radix-2, radix-2^2 algorithms in different approaches such as word groups, constant geometry, and further study those approaches with different possible schedules in order to gain performance. We use different techniques to accelerate the computations of the FFT, such as shared memory utilization and instruction optimizations. We also discuss the results and comparisons of different approaches. Finally, we compare our best cases of results with CUFFT, which is a specific library to compute the FFT on NVIDIA GPUs.

Apart from Introduction chapter, this document is organized as follows,

- Chapter 2 gives the introduction to GPUs, an overview of GPU architectures, CUDA programming language, and important guidelines to improve

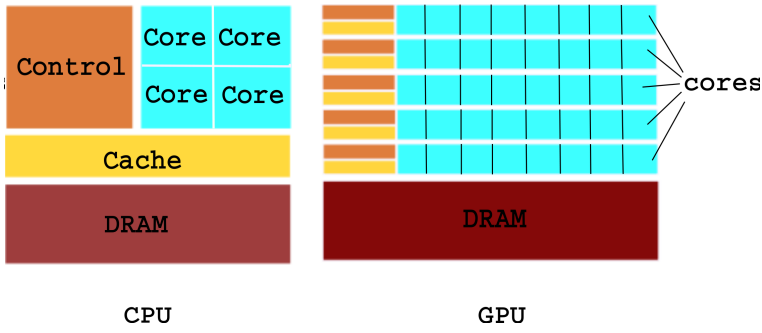

Figure 1.1. CPU and GPU Comparison

the performance of the CUDA applications.

- Chapter 3 gives a brief introduction to the FFT algorithm. This chapter also presents the radix-2, radix-2^2 algorithms, and further alternate arrangements in these algorithms such as word groups, constant geometry, that yields an equally useful algorithms in the desired hardware.
- Chapter 4 discusses the different experiments using radix-2 and radix-2^2 algorithms conducted on GPU using CUDA, with different approaches, such as word groups, constant geometry, and further implemented these approaches with different schedules, in order to gain the performance.
- Chapter 5 discusses about the comparisons and results of the implementations, discussed in Chapter 4. We also compare our best results with CUFFT.

Chapter 2

GPUs

In this Chapter, Section 2.2 provides a short introduction to graphical processing, Section 2.3 provides a brief history of GPUs, Section 2.4 explains the usage of NVIDIA GPUs using CUDA.

2.1 Introduction

GPUs [5] have been used successfully for visual-processing (graphical-processing) in the last three decades. Initially, GPUs were designed specifically to accelerate the graphical processing. A short description of graphical processing and a brief history of the GPUs gives an idea about the fundamental reasons behind the design of modern GPU architecture.

2.2 Graphical processing

Many of the graphical images are more pleasant for the human eyes, though they are absolutely a human imagination. The reason for such acceptability is the shape of an object and its color in the images. In graphical processing, the shape of the certain object is formed with only triangles and that object is bit mapped with the texture (colored image), which result in the effective graphical image.

2.2.1 Graphics pipeline

The stages in the graphical processing are arranged as a "graphics pipeline", which is shown as Figure 2.1.

The first stage of the graphics pipeline is vertex operations. Any surface for a required object can be formed with triangles. This stage translates the input data into vertices of a triangles. As there might be hundreds of thousands of vertices, and each vertex has to be computed independently, a high number of processing units are required in order to accelerate the processing of this stage.

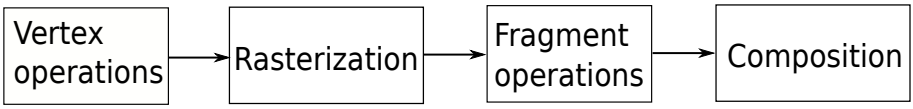

Figure 2.1. Graphics pipeline

The second stage is rasterization, which is the process of determining which screen pixels are covered by each triangle.

In the third stage fragment operations, the data is fetched in the form of textures (images) and mapped on to surfaces (which is framed in vertex operations stage) through fragment (pixel) processing. For an effective scene, different colors are shaded to each pixel. As there are a million number of pixels, this stage is also required a high number of processing units to accelerate the fragments processing.

In the final stage composition, all the fragments are assembled and stored into a frame buffer.

2.3 A brief history of the GPUs

The hardware of the early GPUs was the fixed function graphics pipeline and those were popular for rendering images during 1980 to 2000. Following subsection explains briefly about the fixed function pipilines.

2.3.1 Fixed function graphics pipelines hardware

Fixed function pipelines consisted of hardware which is only configurable, but not programmable. It is operated with Finite State Machines (FSM) from Application Programmable Interfaces (API). An API is a standardized layer that allows applications (such as games) to use software or hardware services and functionality. For two decades, APIs became more popular and each generation of the hardware and its corresponding generation of API brought incremental improvements for the rendering of images.

2.3.2 Programmable graphics hardware

The next generation of GPUs consisted of hardware with both vertex processors and fragment processors and which were programmable. The vertex processors were used for the vertex operations and the fragment processors were used for the fragment operations. Both of these processors have separate instruction sets [3]. The programming for this hardware was more flexible compared to the fixed graphics pipelines and the acceleration of the graphical processing was further increased. It also enabled more different effects for the rendering images. Still, the graphics experts were not satisfied because of the following problem in the hardware utilization.

On the one hand, in a few of the program tasks, all the vertex processors might be busy with a high number of vertex operations, where as the many of the

fragment processors might be free, computing only very few fragment operations. On the other hand, in a few of the program tasks, many of the vertex processors might be free, computing only very few operations, where as all the fragment processors are busy with a greater number of the fragment operations. The result is that one part of the hardware is overloaded with more operations, and the other part is free with less operations.

The next generation of GPUs consisted of the hardware with "unified shader processors", in which the same processors were used for both the vertex operations and the fragment operations to increase the hardware utilization. The Microsoft Xbox360 is using the first GPU with unified shader processors, but still the separate instruction sets for both the vertex operations and the fragment operations. It was also difficult for the general purpose (non graphics) programming.

In 2006, NVIDIA introduced two key technologies, one being the Geforce8800 and the other is the CUDA programming language. The Geforce8800 hardware consists of an architecture with unified shader processors [4]. CUDA is a parallel programming language used to Compute the Unified shader Device Architecture. General programmers were also flexible with CUDA programming language.

From 2006 to the present, the NVIDIA GPUs were more popular than others such as ATI or Intel GPUs. During the same period, NVIDIA released several generations of GPUs, each generation brought incremental improvements in the hardware such as number of cores, data bus size (data type), shared memory size (data cache), DRAM size, etc.

2.3.3 Fermi: A modern GPU architecture

The Fermi architecture is the major leap in GPU architecture since Geforce8800 [6].

The first Fermi-based GPU, implemented with 3 billion transistors, features up to 512 CUDA cores. Each CUDA core is a small processing unit which has a fully pipelined integer Arithmetic Logic Unit (ALU) and Floating Point Unit (FPU). 512 CUDA cores are organized in 16 small groups of 32 cores each. Each small group called as Streaming Multiprocessor (SM).

Figure 2.2 shows the Fermi architecture. The 16 SMs are positioned around a common L2 cache. Each SM is a vertical rectangular strip as shown in Figure 2.2. Figure 2.3 shows a better look of an SM. It contains:

- 32 processing cores (green portion), working together Single Instruction Multiple Data (SIMD), with minimal control logic.
- register files and 64 KB of on-chip memory that can be configured as 48 KB of Shared memory with 16 KB of L1 cache or as 16 KB of Shared memory with 48 KB of L1 cache (light blue portion).
- 16 load/store units, allowing source and destination addresses to be calculated. Supporting units load and store the data to cache or DRAM.
- 4 Special Function Units (SFU) execute transcendental instructions such as sin, cosine, reciprocal, and square root. Each SFU execute one instruction

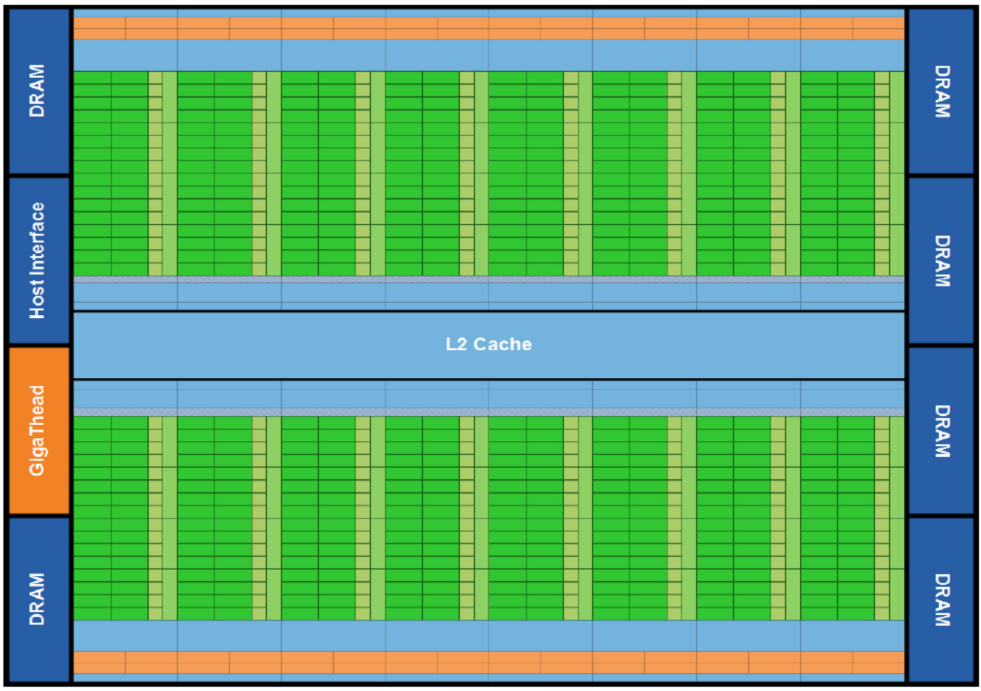

Figure 2.2. Fermi Architecture (Source: NVIDIA)

per clock.

- a dual instruction dispatch unit and warp scheduler (orange portion), and more details about this will be given this in the following section (Execution model).

Apart from these, the GPU has six 64-bit memory partitions, supporting up to a total of 6 GB of DRAM memory. The DRAM also called device memory, which includes constant memory, texture memory, local memory, and global memory. The access latencies from the processing cores are in increasing order as followed by register files, shared memory, constant memory, texture memory, local memory, and global memory.

A host interface connects the GPU to the CPU via a PCI-Express. The GigaThread global scheduler distributes thread blocks to the schedulers of each SM. The description of threads and thread blocks are given in the following sections.

2.4 CUDA

CUDA is a high level programming language released by NVIDIA in 2006. It is “C“ with NVIDIA extensions and some certain restrictions. It is also called as

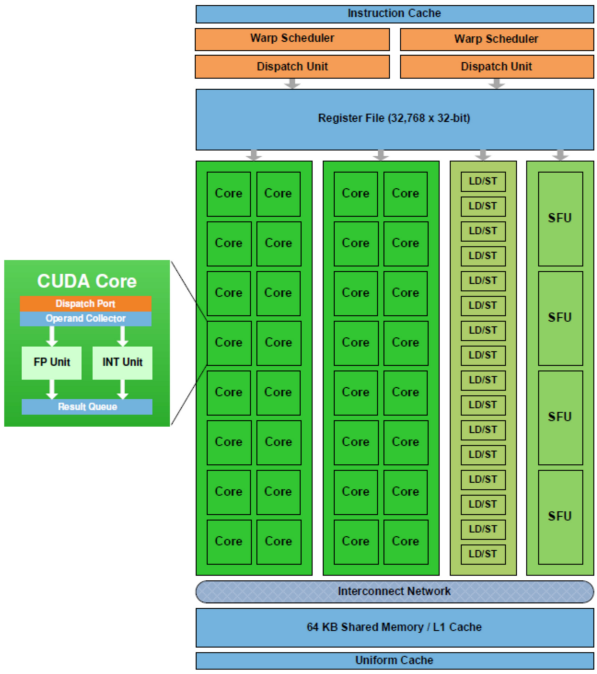

Figure 2.3. Streaming Multi processor (Source: NVIDIA)

computing engine of GPU. CUDA was designed for programming only NVIDIA GPUs.

This section describes the usage of NVIDIA GPUs using CUDA, through programming model, memory model, execution model and some guidelines to improve the performance.

2.4.1 Programming model

CUDA is a heterogeneous computing language. The structure of CUDA program consists of both "host" and "device" parts, the host part is executed on the CPU and the device part is executed on the GPU. The host part exhibits little or no parallelism and the device part exhibits a larger portion of parallelism [2].

A CUDA program is a unified source code on the host, which consists of both host code and device code. NVIDIA C compiler separates host code and device code during the compilation process. The host code is a straight C code, which is compiled further by regular C compilers, and the device code is NVIDIA C extensions, for device parallelism, which can be launched by special device functions called "kernels". The device code further compiled by NVCC (NVIDIA C Compiler) and executed on the GPU (device).

The further discussion in this subsection describes the CUDA kernels, threads, and thread blocks.

```
//kernel definition
__global__ void  VecAdd (float* X, float* Y, float* Z)
{
     int i = threadIdx.x;
     Z[i] = X[i] + Y[i];
}
int main()
{
      .....
      //kernel invocation
       VecAdd<<<1,9>>>(X, Y, Z);
}
```

Figure 2.4. Simple CUDA Program: Vector addition

CUDA kernel and threads

Generally, a CUDA program calls parallel kernels. A kernel executes in parallel across a set of parallel threads. The programmer or compiler organizes these threads in thread blocks and grids of thread blocks.

The kernel invocation specifies the "number of threads" by using execution syntax as shown in the following equation, the detailed explanation about the syntax of kernel definition are given in the following section.

$$\text{Kernel name} <<< \text{number of threads} >>> (\text{parameters}) \tag{2.1}$$

Each thread has a unique "thread index", which can be accessible using "threadIdx".

As an illustration, the code in Figure 2.4 is a simple CUDA program, which adds two vectors X and Y of size "9" and the calculated result is saved into vector Z. There are two functions in program, those are $main()$ and $VecAdd()$.

The CUDA program compilation starts from the $main()$ function by the C compiler, same as the regular C program. When the C compiler finds an NVIDIA extension function such as $VecAdd$ by " <<<>>> ", it gives the responsibility to the NVCC to compile the kernel code.

The execution of the device code can be done using an array of threads as shown in Figure 2.5. All the threads execute the same instructions. Each thread has a unique thread index, that it used to compute the memory addresses and the control decisions.

Thread blocks

A group of threads is called "thread block" and a group of such thread blocks is known as a "grid". These are organized by a programmer or compiler.

The threads in a thread block are aligned in one-dimensional, two-dimensional, or three-dimensional form. In a similar way, thread blocks in a grid are aligned

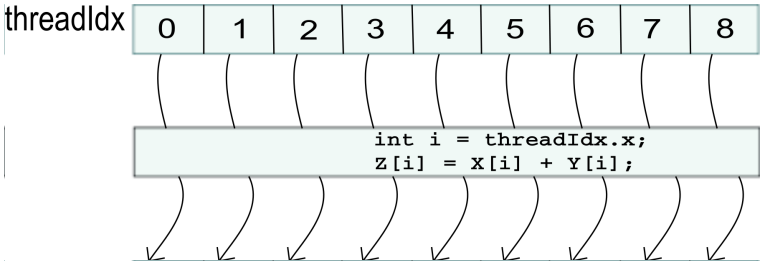

Figure 2.5. Execution of Threads

in a one-dimensional, two-dimensional or three-dimensional form. This provides a natural way to invoke computations across the elements in a domain such as a vector, matrix, or volume.

The thread ID can be calculated based on its index of thread. For a one dimensional block they are the same. For a two dimensional block of size (Dx, Dy), if the thread index is (x, y) then its thread ID is $(x+y \cdot Dx)$. For a three dimensional block of size (Dx, Dy, Dz), if the thread index is (x, y, z) then its thread ID is $(x + y \cdot Dx + z \cdot Dx \cdot Dy)$ [17].

The number of blocks per grid (nB) and the number of threads per block (nT) are specified in a kernel launch as

$$\text{Kernel name} <<< nB,\ nT >>> (\text{parameters}) \tag{2.2}$$

Figure 2.6 gives the illustration of a thread hierarchy. The nB and nT are organized in its kernel as

$$\text{Kernel name} <<< nB,\ 10 >>> (\text{parameters})$$

where $nB = (4,2)$, represents a two dimensional blocks by 4 columns and 2 rows and a total of $4 \cdot 2 = 8$ blocks. Each block has 10 number of threads. Each thread in a thread block and each block in a grid has unique thread index and block index respectively as shown in Figure 2.6.

2.4.2 Memory model

The data transfer from the host memory to the device memory happens before the kernel is launched. There are three different memory levels on the device. They are register files, shared memory and device memory (includes global memory, local memory, texture memory, and constant memory). Figure 2.7 illustrates how CUDA threads can access the three memory levels during the execution.

Each thread has separate Register files (RF), also called private memory, which has less latency of memory access. A thread block has a shared memory which is accessible by all threads in each thread block. The latency access is almost the same as the register files. All threads in the grid can access the global memory. The latency access of global memory is approximately 100x slower than shared memory.

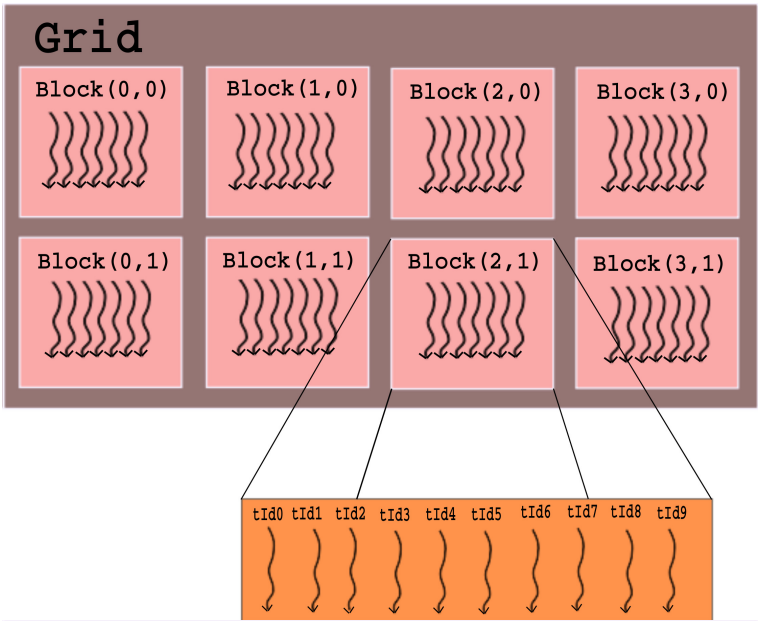

Figure 2.6. Thread Heirarchy

The data can be transferred from global memory to shared memory by using the device code. Accessing the data in shared memory rather than global memory lead to better performance. Different strategies to improve the performance of the memory accesses will be discussed in the following sections .

2.4.3 Execution model

This section describes about the scheduling of CUDA threads and thread blocks in the cores and the SMs of a GPU, during the execution.

A GPU executes one or more kernel grids. However, a GPU executes one grid at a time, which is scheduled by a programmer or compiler. Similarly, a streaming multiprocessor (SM) executes one or more thread blocks. One SM executes one thread block at a time, which is scheduled automatically by a "giga thread global scheduler" using scalable programming model.

Scalable programming model

A multi threaded program is partitioned into blocks of threads by programmer or compiler. Those execute independently from each other.

Let us consider that a multi threaded program is partitioned in 8 thread blocks, as shown in Figure 2.8, which have to be executed in 2 different GPUs, one consisted of 2 SMs and other consisted of 4 SMs. Let us assume that the number of cores in the SMs are the same in both GPUs. On the one hand, all the thread

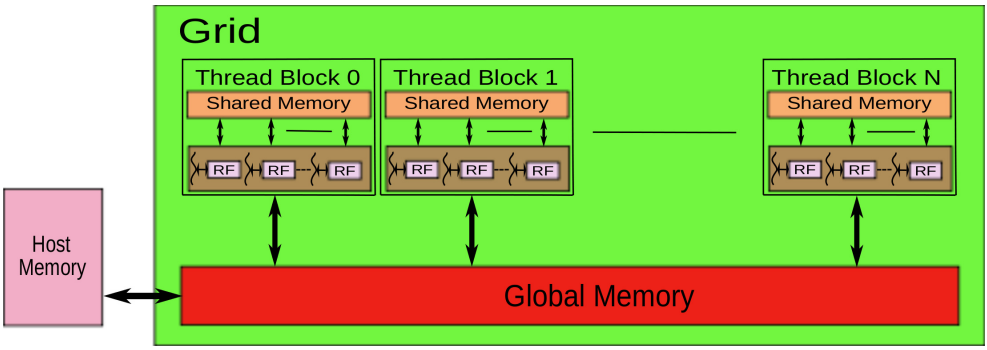

Figure 2.7. Memory Heirarchy

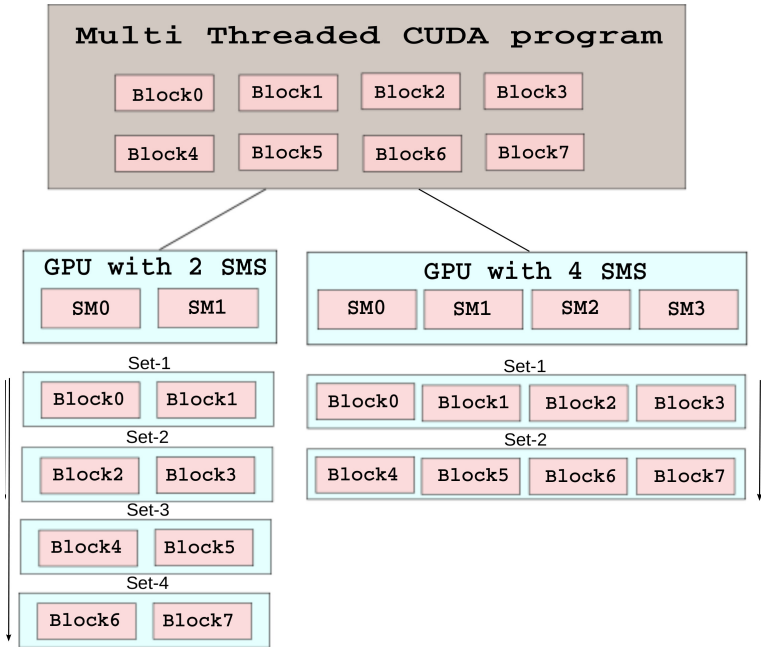

Figure 2.8. Scalable programming model

blocks are scheduled on a GPU which has 2 SMs in four sets. Each set having two thread blocks are executed on parallel. On the other hand, the same thread blocks are executed on a GPU which has 4 SMs in two sets. Each set having four thread blocks are executed on parallel. Here, the number of thread blocks in a set are automatically scheduled by the giga thread global scheduler, depends on the number of the thread blocks in the CUDA program and the number of SMs in the GPU.

Warp Scheduler

As explained earlier, each block is executed by one SM. It means that the cores in a SM execute all the threads in a thread block. These threads are scheduled automatically by a dual warp scheduler. It schedules threads in groups of 32 parallel threads called warps. The dual warp scheduler selects two warps and the dual instruction dispatch units selects particular instructions to be executed concurrently as shown in Figure 2.9. All the warps execute independently and therefore the warp scheduler does not need to check for dependencies from within the instruction stream [6].

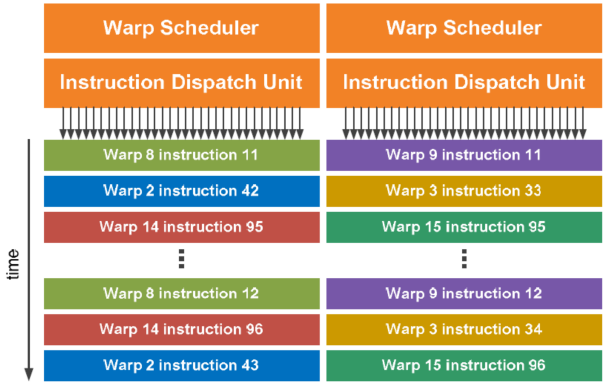

Figure 2.9. Dual warp scheduler (Source: NVIDIA)

The threads in a thread block cooperate with each other while execution as the following.

- Registers in one SM are partitioned among all threads in a thread block.
- Threads in a thread block can only access data in shared memory of its mapped SM, which means that executing threads in one SM can not access data from other SM.
- Threads in a thread block can only be synchronized, while execution by calling ___syncthread().

2.4.4 Performance guidelines

These performance guidelines are given by the CUDA programming guide [7] and the CUDA C best practices guide [8] to improve the performance of CUDA program on the GPU architecture. The most important ones for our application are discussed as below.

Maximize the parallelism

It is very important to find as many ways as possible to parallelize the sequential code in order to obtain the optimal performance.

Amdahl's law [16] specifies the maximum speed-up that can be expected by parallelizing the sequential code. The maximum speed-up (S) of a program can be calculated by

$$S = \frac{1}{(1-P) + \frac{P}{N}} \tag{2.3}$$

where P is ratio of the execution time taken by the part of the code that can be parallelized to the total sequential code. N is the number of the processors, those can be used to run the parallel portion of code.

For example, if P is 80%, which is the proportion of a program that can be made parallelized, then $(1-P)$ is 20%, is the proportion that can not be parallelized (remains sequential), then the maximum speed up that can be acheived by

$$S = \frac{1}{0.2 + \frac{0.8}{N}} \tag{2.4}$$

here, if the number of processors N tends to infinity, then the maximum speed up that can be acheived is a factor of 20. From the observation of the graph shown in Figure 2.10, the speed up factor (S) has been increased, upto the number of processors 1024, and then it is constant. This means that the speed up can not be increased more than this, no matter how many processors, unless there are ways to increase the proportion of parallelized portion.

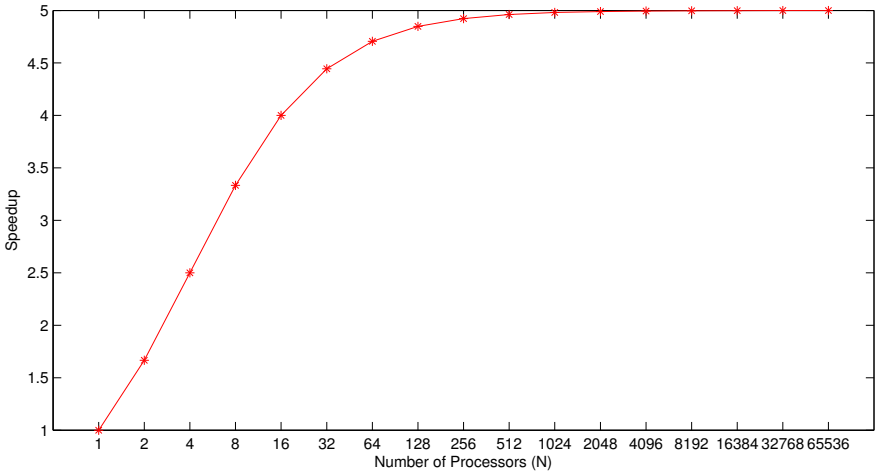

Figure 2.10. Amdahl's Law

Data transfer between host and device

It is of high priority to minimize the data transfer between the host and the device, because the bandwidth between host memory and device memory is very low. In

some programs, this can be reduced by creating the intermediate data structure in the device memory. This can be done with the kernel code, by generating the required data, and is stored in the device memory. Finally, this data is destroyed after uses, without data transfer from device memory to host memory.

For example, the code in Listing 2.1 computes the addition of two vectors A and B, and stored into vector C. The input data A and B are generated on the CPU and transferred from host to Device (see the code in Listing 2.1 from line 15 to 28). Here, there is a possibility to generate the same data from the kernel code as shown in Listing 2.2. The same data is generated from the kernel code on GPU part (see the code in Listing 2.2, lines 6 and 7). This helps to reduce or avoid the data transfer from host to device.

Generating the data from kernel code, may not be possible in all the cases, such as the present input data is the output of previous program, etc.

Listing 2.1. Vector addition: Input data from host to device

```
1  #define  SIZE 16
2  //GPU Part
3  __global__  void vecadd(float* A, float* B, float* C)
4  {
5   short tid = threadIdx.x;
6   C[tid] = A[tid] + B[tid];
7  }
8  //CPU Part
9  #include  <stdio.h>
10 int  main()
11 {
12  float A[SIZE], B[SIZE], C[SIZE];
13  float *Ad, *Bd, *Cd;
14  int memsize= SIZE*sizeof(float);
15  for(int x=0; x < SIZE; ++x)
16  {
17   A[x]= x;
18  }
19  for(int y=0; y < SIZE; ++y)
20  {
21   B[y]= SIZE+y;
22  }
23  cudaMalloc((void**)&Ad, memsize);
24  cudaMalloc((void**)&Bd, memsize);
25  cudaMalloc((void**)&Cd, memsize);
26  //Data transfer from Host to Device.
27  cudaMemcpy(Ad, A, memsize,  cudaMemcpyHostToDevice);
28  cudaMemcpy(Bd, B, memsize,  cudaMemcpyHostToDevice);
29  dim3 gridDim(1,1);
30  dim3 blockDim(SIZE,1);
31  vecadd<<<gridDim , blockDim>>>(Ad , Bd , Cd);
32  //Data tansfer from Device to Host.
33  cudaMemcpy(C, Cd, memsize,  cudaMemcpyDeviceToHost);
34  printf("The outputs are: \n");
35  for (int z=0; z<SIZE; z++)
36  printf("C[%d]=%f \t B[%d] = %f  \t A[%d] = %f
37         \n ",z,C[z],z,B[z],z,A[z]);
38 }
```

Listing 2.2. Vector addition: Input data is generated from kernel code (GPU Part)

```
1  #define  SIZE 16
2  __global__  void vecadd(float* C)
3  {
4   __device__ float A[SIZE], B[SIZE];
5   short tid = threadIdx.x;
6   A[tid] = tid;
7   B[tid] = SIZE+tid;
8   C[tid] = A[tid] + B[tid];
9  }
10 #include  <stdio.h>
11 int  main()
12 {
13  float C[SIZE];
14  float  *Cd;
15  int memsize= SIZE*sizeof(float);
16  cudaMalloc((void**)&Cd, memsize);
```

```
17  //Data transfer from Host to Device.
18  //-----Nothing to transfer as Data will generate
19  //from kernel code-----.
20  dim3 gridDim(1,1);
21  dim3 blockDim(SIZE,1);
22  vecadd<<<gridDim , blockDim>>>(Cd);
23  //Data tansfer from Device to Host.
24  cudaMemcpy(C, Cd, memsize,  cudaMemcpyDeviceToHost);
25  printf("The outputs are: \n");
26  for (int z=0; z<SIZE; z++)
27  printf("C[%d]=%f \n ",z,C[z]);
28 }
```

Global memory accesses

It is also important to minimize the global memory accesses by maximizing the use of on-chip memory that is shared memory. There is a latency of 400 to 600 clock cycles to read data from global memory. Shared memory is equivalent to the cache. It can be done by taking the advantage of parallelism, for data transactions between global memory and shared memory using SIMD by many threads, in other words SIMT (Single Instruction Multi Threading). Accessing the data from shared memory is 100x faster than that of global memory. The following process explains how to use the shared memory.

1. Load data from device memory to shared memory by Single Instruction Multiple Threads.
2. Synchronize all the threads in a block by ___syncthreads() function which makes them safe to read. This assures that all threads loaded the data from global memory to the shared memory.
3. Process or compute the data by accessing from shared memory.
4. Synchronize again if data in shared memory has modified which makes safe to write.
5. Write the results back to global memory.

The concept of coalescence

The data transaction from/to two or more consequent memory blocks is called "coalescence". It plays an important role in order to improve the performance. The concept of coalescence explains about the importance of utilizing most of the bandwidth, while accessing the global memory. The requirements of accessing global memory in order to fulfill coalescence depends on the compute capability of the device, and can be found in [7] (Appendix F.3.2 for compute capability 1.x and Appendix F.5.2 for compute capability 2.x). Here, the coalescence requirements for compute capability 2.x are explained as latest GPUs are of compute capability 2.x, and the the device used for our work is also the same.

The bandwidth of the latest GPUs is 128 bytes. This means that the cache line (bus) carries 128 bytes of data for every global memory request. The cache line is aligned at addresses

$$A[x] = 128 \cdot x, x = 0, 1, ..., \frac{G}{128} \tag{2.5}$$

where G is the total global memory size and x is an integer from 0 to $\frac{G}{128}$.

For every global memory request, bus carries 128 bytes of data at addresses from $A[x]$ to $A[x+1]-1$, no matter of how much data has been requested between $A[x]$ and $A[x+1]-1$.

If a warp has requested the data at addresses from $A[x]$ to $A[x+1]-1$, it is called "perfectly aligned".

Let us consider an example, that each thread in a warp (32 threads) requests to fetch 4 bytes of data, and the starting address from $A[1] = 128$, as shown in Figure 2.11. Therefore, all the threads in the warp request $32 \cdot 4 = 128$ bytes of data. The cache line fetches 128 bytes of data, at addresses $A[1] = 128$ to $A[2] - 1 = 255$, as shown in Figure 2.11. The data is aligned perfectly, to get the required data in a single transaction.

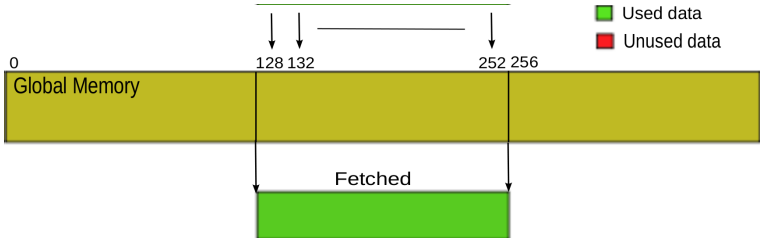

Figure 2.11. Illustration of Coalescence: Perfectly aligned

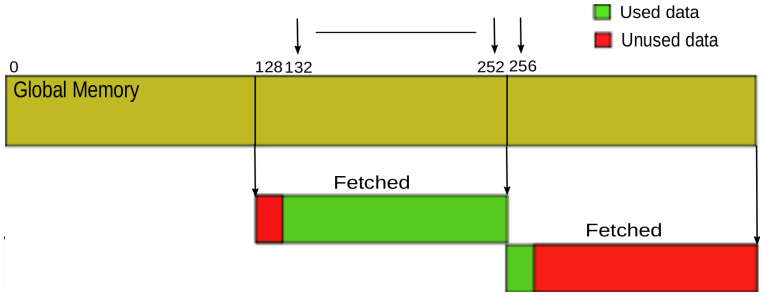

Figure 2.12. Illustration of Coalescence: Missaligned

If a warp has requested a data, which is neither aligned at address $A[x]$ nor with range from $A[x]$ to $A[p \cdot x + 1] - 1$, it is called "missaligned". Where p is an positive integer from 1 to $\frac{G}{128}$.

Let us consider another example, that each thread in a warp requested to fetch 4 bytes of data like previous example, but the starting address is 132 as shown in Figure 2.12. So, here also the warp requests the 128 bytes of data, but the data is not aligned. The cache line fetches two times with 128 bytes of data for every data transaction. The data at addresses $A[1] = 128$ to $A[2] - 1 = 255$ are fetched in the first transaction, and the data at addresses $A[2] = 256$ to $A[3] - 1 = 383$ are fetched in the second transaction. Even though 256 bytes of data are fetched, only

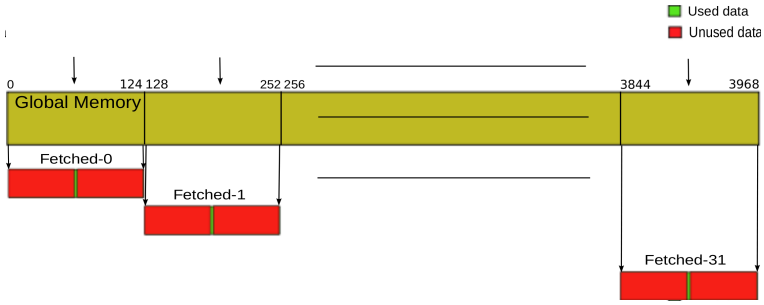

Figure 2.13. Illustration of Coalescence: Missaligned very bad

128 bytes of data are used as shown in figure. This is because of missalignment of data.

Let us consider one more example, to explain the worst case of missalignment. Assume that each thread in a warp requests 4 bytes of data as in previous examples, but each location requested is addressed with a difference of 128 bytes to other location, as shown in Figure 2.13. For every 4 bytes of data, 128 bytes of data is fetched, in each transaction, in which 4 bytes of data is only used, and rest is unused. Totally $128 \cdot 32 = 4096$ bytes of data are fetched in 32 transactions, in which 128 bytes of data is only used. All the examples discussed above are also applied for storing into global memory. Therefore, special care should be taken when accessing from/into global memory.

Maximize instruction throughput

The following points explain about maximize instruction throughput.

1. Reduce the intrinsic (in-built) functions. Instead use the regular operations if possible, because intrinsic functions are of high latency. These functions can be replaced with regular operations as follows,

$$mod(x, y) = x - floor(x/y). \tag{2.6}$$

2. Replace the possible expensive operations with simpler operations. Let us consider the expensive operations such as multiplication or division having two operands, and one of the operand is a power of 2, then the same operations can be replaced with data shifts, which are very simple operations. This can be illustrated as follows,

$$mul(x, 8) = x << 3 \tag{2.7}$$

here, 8 is 2^3, so it can be replaced with shifters. The result of $x \cdot 8$ is same as left shift of x 3 times.

$$div(y, 16) = y >> 4 \tag{2.8}$$

here, 16 is 2^4, so it can be also replaced with shifters. The result of y/16 is same as right shift of y 4 times.

3. Optimize the synchronization points when it is possible because synchronization functions make the program wait until all the threads have completed their execution.

4. Avoid different execution paths (if, switch, do, for, while) within a warp.

   Since the execution of threads will be done in terms of warps, the different execution paths in a warp makes to consume high execution time. So, it is of high priority to keep no divergences within a warp.

   To obtain the best performance, the controlling condition should be written so as to minimize the number of divergences in a warp. The controlling condition should be aligned at (thread ID / WSIZE), where WSIZE is the warp size. If this is the controlling condition, then all the threads in a warp execute the same instructions or no divergences within a warp.

5. Replace the possible double precision operations with single precision operations, because double precision operations are of high latency compared to single precision operations.

   For example, instead of using floating point functions such as cosf(), sinf(), tanf(), use single precision functions such as sincosf(), sinsinf(), sintanf().

6. Prefer faster, more specialized math functions over slower, more general ones when possible.

   For example, incase of exponentiation using base 2 or 10, use the functions exp2() or expf2() and exp10() or expf10() rather than the functions pow() or powf().

   The functions exp2(), exp2f(), exp10(), and exp10f(), on the other hand, are similar to exp() and expf() in terms of performance, and can be as much as ten times faster than their pow()/powf() equivalents.

Chapter 3

Introduction to the FFT

The discrete Fourier transform (DFT) plays an important role in the Digital Signal Processing (DSP). The direct computation of the DFT involves a large number of complex multiplications and additions, which cause the consumption of high computational time. This problem was discussed for a long time, leading to the development of algorithms reducing the number of complex multiplications and additions called Fast Fourier Transform (FFT) [9], [10], [11], [12]. This chapter gives a short introduction to the FFT algorithms using radix-2 and radix-2^2, briefly about the FFT implementations on GPUs, and a couple of fast computing techniques of FFT in processors.

3.1 FFT

The DFT transforms a signal in the time domain into the frequency domain. The DFT of a finite sequence of length N is defined as

$$X[k] = \sum_{n=0}^{N-1} x[n] {W_N}^{kn}, k = 0, 1, ..., N-1 \tag{3.1}$$

where $W_N = e^{-j(\frac{2\pi}{N})}$.

The inverse discrete Fourier transform, which is used to recover the original signal in time domain is given by

$$x[n] = \frac{1}{N} \sum_{k=0}^{N-1} X[k] W_N^{-kn}, n = 0, 1, ...N-1 \tag{3.2}$$

where, $x[n]$ and $X[k]$ may be the complex numbers in both equations.

The direct computation of the DFT requires $O(N^2)$ complex multiplications and complex additions. With the goal of fast computing the DFT, The FFT (Fast Fourier Transform) includes a collection of algorithms that reduce the number of operations of the DFT. The Cooly-Tukey algorithm is the most commonly used as

the FFT. It has a total number of complex multiplications and complex additions of order $O(N \cdot \log_2 N)$. It is a divide and conquer algorithm, which decomposes the DFT into $s = \log_r N$ stages. The N-point DFT is built with r-point small DFTs in s stages such that $N = 2^s$. Here, r-point DFT is called radix-r butterfly. For example radix-2 butterfly is shown in Figure 3.1. A radix-2 butterfly has only one addition and one subtraction.

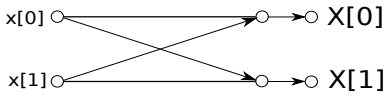

Figure 3.1. Radix-2 butterfly

Figure 3.2 shows the flow graph of the 16-point radix-2 DIF FFT. The input sequences in the 16-point radix-2 DIF FFT are in natural order whereas the outputs are in bit reversed order [15]. There are $\log_2 N$ stages in the algorithm, where N is the FFT size equal to 16. There are $N/2$ butterflies in each stage. Each butterfly has one addition, one subtraction, and followed by a multiplication with a twiddle factor, $e^{-j\frac{2\pi}{N}\phi}$. Figure 3.2 assumes that the lower edge of each butterfly is always multiplied by -1 and the number ϕ after each stage represents a multiplication with the corresponding twiddle factor, $e^{-j\frac{2\pi}{N}\phi}$. On the other hand, Figure 3.3 represents a 16-point radix-2 DIT FFT. In this, the inputs are in bit reversed order and the outputs are in natural order. The twiddle factors of first stage in DIT are the same as the last stage of DIF and viceversa.

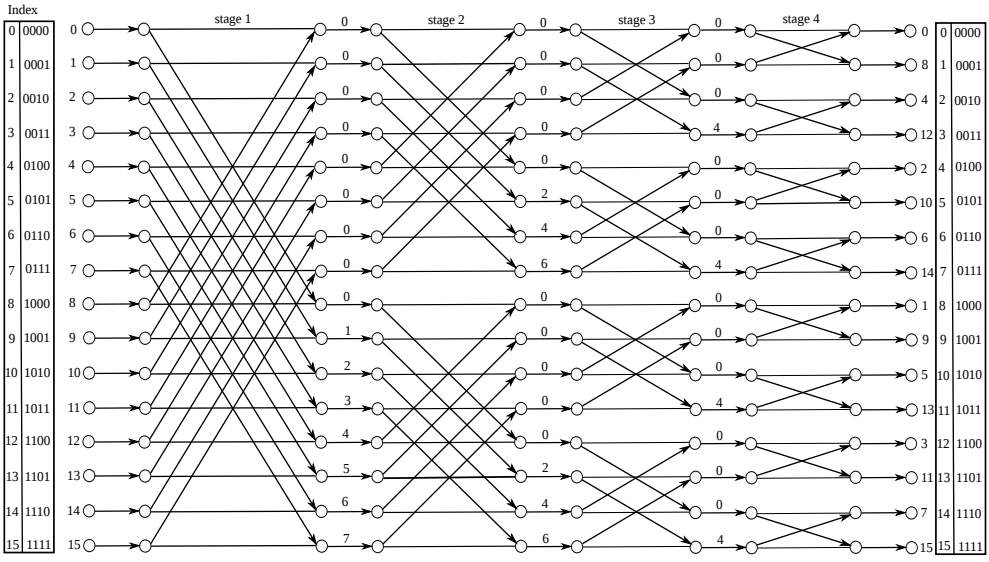

Figure 3.2. Flow graph of the 16-point radix-2 DIF FFT

Radix-2^2 [14] is an alternative to radix-2. The flow graph of the 16-point radix-2^2 DIF FFT is shown in Figure 3.4. By comparing fig 3.2 and fig 3.4, noticed that

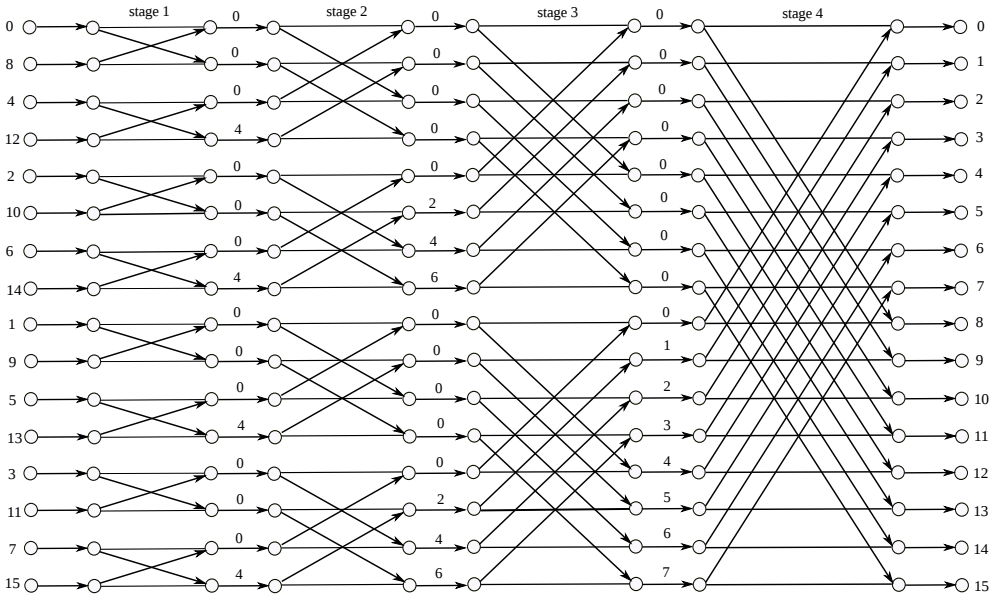

Figure 3.3. Flow graph of the 16-point radix-2 DIT FFT

the only difference is in twiddle factors. It is further noticed that ϕ is 0 or 4 in odd stages, which results in the twiddle factor being 1 or -j. While implementation, the multiplication with 1 or -j is operated by moving real and imaginary words and/or changing signs, and such a stage is called trivial stage. There are $n/2$ trivial stages in the N-point radix-2^2 FFT, where $n = log_2 N$. This means that the multiplication of the non trivial factors is required only for a non trivial stage (i.e. even stage).

3.2 FFT implementations on GPU

This section explains about the related work on GPUs.

In [20], the mixed radix FFT implementations were implemented on the FPGA. The performance and the energy efficiency compared with the FFTW 3.2 FFT library, which is executed on the CPU and the CUFFT 3.2 FFT library, which is executed on the GPU. They concluded that the GPUs are more suitable as the FFT accelerators and the FPGAs are the energy efficient than the CPU and the GPU.

In [21], proposed a mixed precision method of the FFT algorithm on the CPU-GPU heterogeneous platform to increase the accuracy. The twiddle factors are calculated in double precision on the CPU, and are transferred into the GPU in single precision using a single float array. The GPU calculate the butterfly operations in single precision. They compared the error rate and the performance with the CUFFT 1.1 and CUFFT 3.2. The error rate of the mixed precision FFT

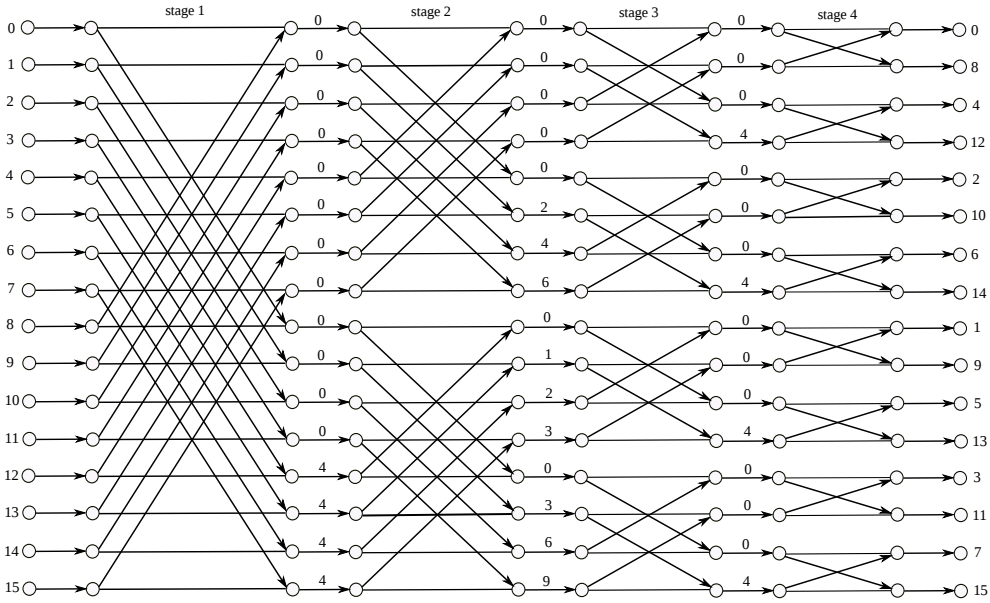

Figure 3.4. Flow graph of the 16-point radix-2^2 DIF FFT

algorithm is less than the CUFFT. The performance of mixed precision FFT is not good as the CUFFT 3.2, and still better than the CUFFT 1.1.

In [22], proposed a OpenCL based FFT and evaluated its performance on different OpenCL programming platforms, such as NVIDIA CUDA, ATI Stream (GPU), ATI Stream (CPU), and Intel OpenCL. The experimental results concluded that the performance of the NVIDIA CUDA is better than the others. The ATI Stream (GPU) is faster than that on the CPUs. The intel OpenCL outperforms the ATI Stream (CPU). They also concluded that the OpenCL based applications can not run only on heterogeneous platforms, but also achieve relatively high performance.

In [23] and [24], presented the fast computation of the Fourier transforms on the GPUs. They used the Stockham formulations to eliminate the bit reversal in the implementation. They compared the results with CUFFT and shown that the performance is better than CUFFT.

In [25], proposed a asynchronously streaming FFT on the GPU. They optimized the data transfer time between the CPU and the GPU using asynchronous communication and the GPU stream programming model. They showed that the speedup is 18 percent higher than the original one.

3.3 Fast computing techniques of FFT in processors

Two techniques are explained in this section to accelerate the FFT implementation in the processors.

3.3.1 FFT in cached memory architecture

M. Baas proposed a FFT data flow graph to implement the FFT algorithm in a cached memory architecture [13]. A small group of data words are loaded from the main memory into a cache, processed for a few intermediate steps, and then written back to the main memory. This increases the speed since the smaller memories (cache) are faster than the larger ones (main memory).

An *epoch* is the portion of the FFT algorithm, where all N data words are loaded into a cache, processed, and written back to the main memory once.

A *word group* is a small portion of the FFT algorithm (or an epoch), where a few of data words less than or equal to the cache size are loaded in to a cache, processed for a few intermediate steps, and written back to the main memory.

For an example, Let us consider that the cache size is equal to the size of 4 data words. Figure 3.5 represents a cached FFT data flow graph for a 16-point radix-2 DIF FFT. As the cache size is equal to 4 data words, 4 data words are loaded from main memory to cache and processed for a portion of the 4-word group in two intermediate steps. The butterflies with thicker lines (colored) represents a 4-word group in Figure 3.5. The stages in the box represents an epoch. All the word groups in the epoch are processed in sequence to share the cache and written back to the main memory once. This increases the speedup since the main memory accesses are done for every epoch rather than every stage.

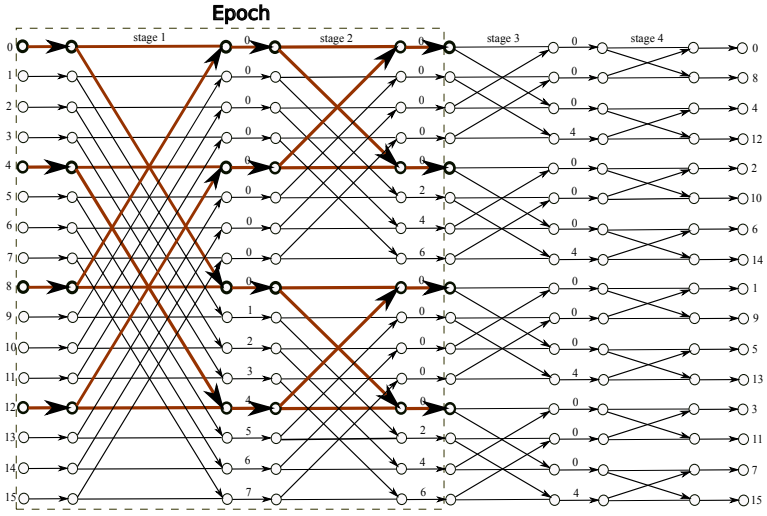

Figure 3.5. Cached FFT data flow graph

3.3.2 FFT using constant geometry

Figure 3.6 shows the 16-point radix-2 DIF FFT flow graph using Constant Geometry (CG) [9]. It is the same as the radix-2 DIF FFT, but the structure of the flow graph is changed in order to keep the indexing same in all the stages. Since

the indexing is kept constant in all the stages, this creates simplification in the indexing. Which in turn increases the speedup, as the calculation of the indexing is required only once for the FFT algorithm rather than for every stage.

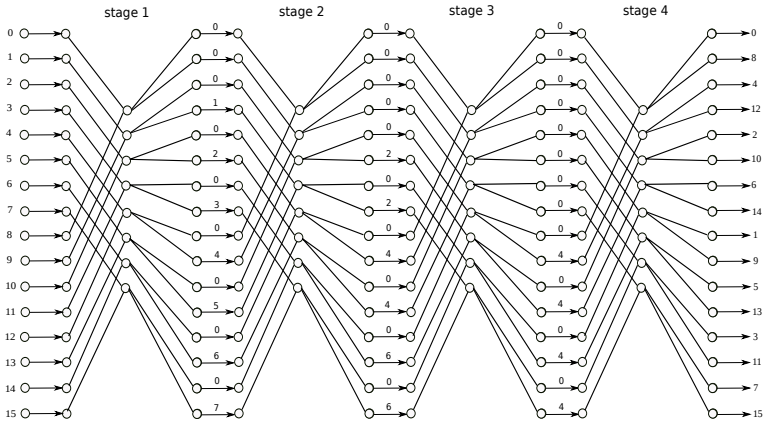

Figure 3.6. 16-point radix-2 constant geometry DIF FFT flow graph

Chapter 4

Implementations of the FFT on GPUs

In this chapter, we propose in Section 4.1 how to parallelise the radix-2 FFT algorithm to implement in the GPUs. We analyze the implementation in order to increase the performance and focus to optimize the synchronization and the index updates from each stage to a group of stages by introducing the word groups and epochs. We propose the different schedules for different word groups , which are explained in Section 4.2. We also propose in Sections 4.3 and 4.5 the flow graphs for the radix-2 and radix-2^2 FFT using constant geometry, in which index is constant in all the stages. Therefore, no index updates are required in any stage by using the constant geometry flow graph. We also propose the approaches in Section 4.7 to implement the FFT in multiple SMs using constant geometry.

4.1 Parallelization of the FFT on GPU

This section explains about how to implement the radix-2 FFT algorithm in parallel.

As explained in Section 3.1, there are $\log_2 N$ stages in a N-point radix-2 FFT algorithm. Each stage has $N/2$ butterflies. Each butterfly consists of an addition, a subtraction, followed by a multiplication by a twiddle factor, as shown in Figure 4.1.

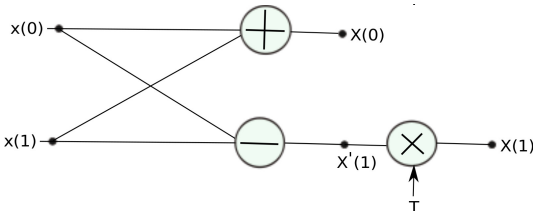

Figure 4.1. Butterfly operations

The data elements for the FFT implementation are stored in the memory. Those are addressed by the indexing before each butterfly operations, which is used for both reading and writing before and after the operations of each butterfly, respectively.

Generally, it is important to find the parallelism in the implementation, to implement a task in multiple cores parallely. In a N-point radix-2 FFT algorithm, there are $N/2$ independent butterflies in each stage. These independent butterflies in each stage are chosen as the parallel processing elements (PE) in order to implement the FFT algorithm in multiple cores. Therefore, each processing element consists of a butterfly operations preceded by indexing to address the data elements required for the particular butterfly operations. The computations of all the butterfly operations in each stage have to be finished before starting the process of the next stage. This means that the parallel processing elements, which process each stage are synchronized by a barrier to confirm that all the operations in that particular stage are completed.

Figure 4.2 represents the parallel implementation of the 16-point radix-2 FFT algorithm. Each vertical processing elements processes the butterfly operations of each stage in parallel, and followed by a synchronization. Each row of tied processing elements represents a thread. The processing elements in each thread are processed concurrently. In each processing element, the data elements are read and written from/into memory, addressed by the indexing before and after the operations of each butterfly respectively. The indexing is calculated in each processing element before the operations of the butterfly.

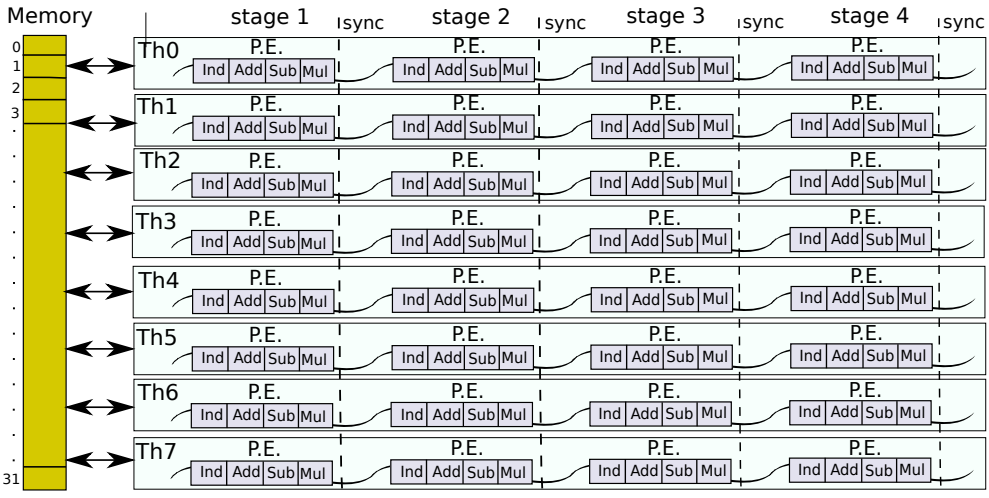

Figure 4.2. Parallel implementation of the 16-point radix-2 FFT

4.2 Word groups and scheduling

This section explains about the implementation of the N-point radix-2 DIF FFT with different word groups, and further with different schedules of each word group.

4.2.1 FFT implementation using word groups

In section 3.3.1, it was explained that the speed up of the N-point radix-2 FFT algorithm can be increased by using word groups in a cached memory architecture.

In the implementation of the FFT on GPUs, we use the concepts of word groups. This subsection explains how to parallelise the FFT implementation on the GPU using word groups. It also explains the computable optimizations by the usage of word groups.

If N is the size of the FFT and x is a power of 2, then an x-word group processes the x-data elements (words) in y intermediate steps, such that $N = x^z$ and $y = \log_2 x$, where $z = \log_x N$. In each intermediate step, the x-word group processes $x/2$ butterflies.

An epoch is a group of few stages, that covers the y intermediate steps processed by the x-word groups.

2-word group

The 2-word group processes 2 data elements for one intermediate step ($\because y = \log_2 2$). Therefore, a 2-word group is the same as the butterfly and an epoch is the same as one stage. So, there are $N/2$ number of independent 2-word groups in each epoch. These 2-word groups in each epoch are chosen as the parallel processing elements to implement the N-point radix-2 FFT in a GPU.

In Figure 4.3, the first stage in the box represents one of the epochs. In which, the butterfly in the thicker lines (colored) represents a 2-word group. Similarly, there are eight 2-word groups in each stage. Its parallel implementation is the same as the explanation in the previous section (refer Figure 4.2). It can be noticed that the indexing is updated and synchronized for each stage.

4-word group

The 4-word group processes the 4-word data elements for two intermediate steps ($\because y = \log_2 4$). An epoch is a group of two stages. So, there are $N/4$ number of 4-word groups in each epoch. These 4-word groups are chosen as the parallel processing elements to implement the N-point radix-2 FFT in a GPU.

In Figure 4.4, the stages in the box represent an epoch. The thicker lines represent a 4-word group. The 4-word group processes 4-data elements in two intermediate steps. Each intermediate step processes 2 butterflies. The advantage of 4-word group is that indexing and synchronization is only required every two stages or epoch. The indexing calculated for the first intermediate step of the 4-word group is used the same for the second intermediate step, which can be noticed from Figure 4.4. Similarly, since the second intermediate step is processed

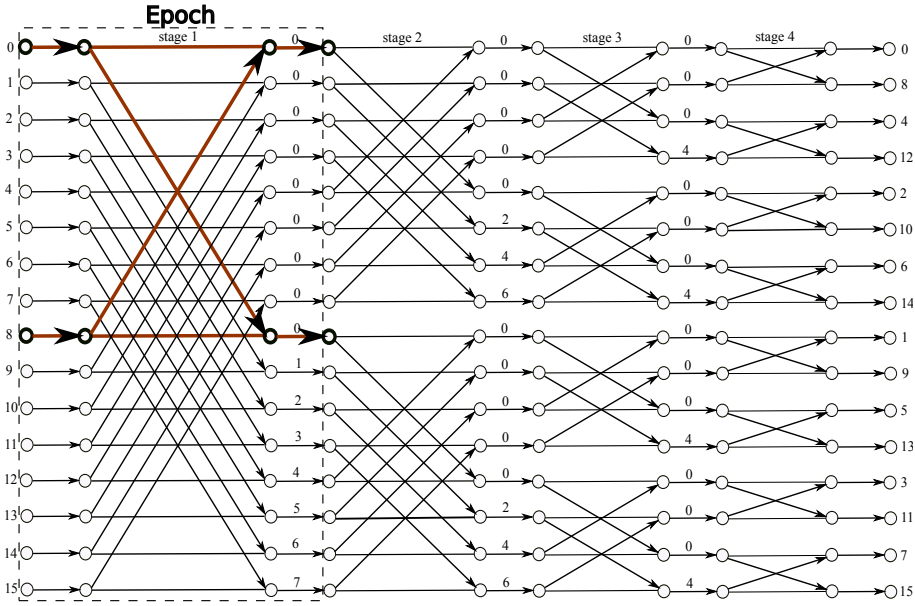

Figure 4.3. 16-point radix-2 DIF FFT using 2-word groups

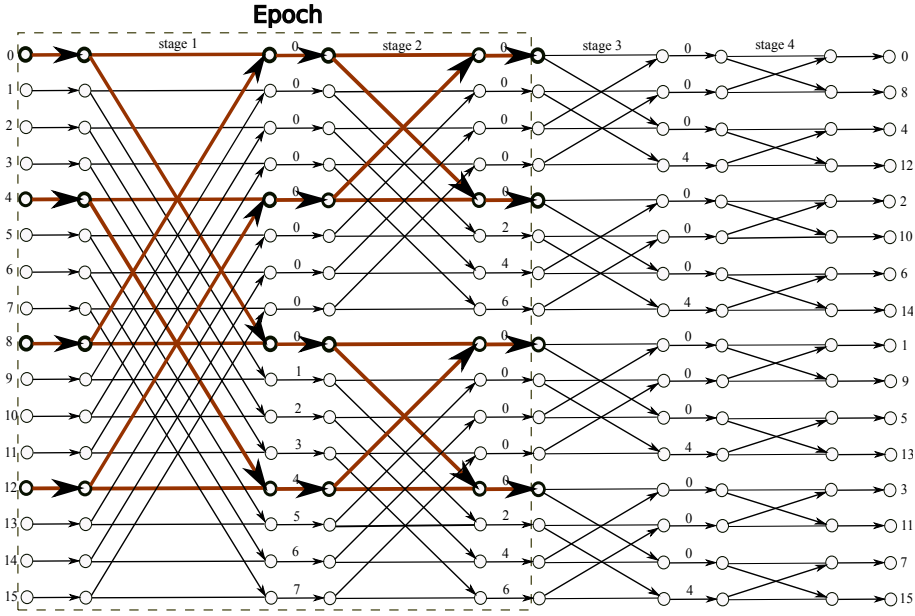

Figure 4.4. 16-point radix-2 DIF FFT using 4-word groups

after the completion of the first intermediate step, the data elements are readily available for the second intermediate step without need of any synchronization.

Similarly, the N-point radix-2 FFT can be implemented using 8-word groups. In this case, the indexing and synchronization is required for each epoch (i.e. every three stages). It means that the larger the size of the word group, the lesser the index update and the synchronizations.

The following subsections, 4.2.2 and 4.2.3 explain about the scheduling of the operations in a 2-word group and 4-word group using different number of threads.

4.2.2 Scheduling of the 2-word group

The data inputs of each butterfly are complex numbers. Therefore, all the operations in the FFT algorithm are complex operations. The upper edge of the 2-word group (or butterfly) has a complex addition. The lower edge of the 2-word group has a complex subtraction followed by a complex multiplication with the twiddle factor (T). The complex operations of a 2-word group are shown in Figure 4.5. It shows that there are two real word additions (i.e. one complex addition) in the upper edge and two real word subtractions (i.e. one complex subtraction) followed by four real word multiplications, one real word addition, and one real word subtraction (i.e. one complex multiplication) in the lower edge.

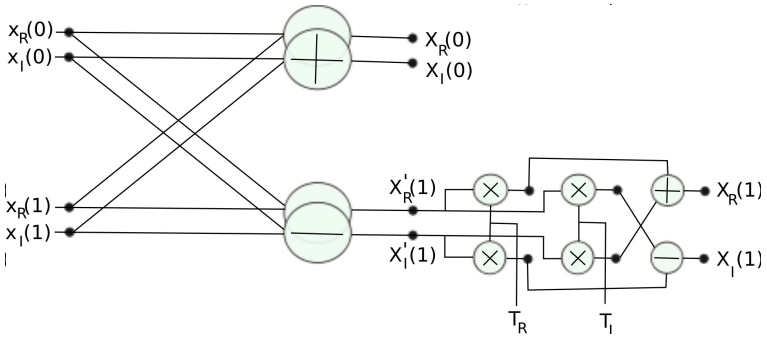

Figure 4.5. Operations of the butterfly

Unbalanced schedule

Let us consider that two threads are used to implement a 2-word group as shown in Figure 4.6. The operations of the upper edge are shared by thread $Th0$ and the operations of the lower edge are shared by the thread $Th1$ directly. The critical path CP is

$$CP = 4 \cdot t_{add} + 4 \cdot t_{mult} \tag{4.1}$$

The thread $Th1$ has to calculate more computations than the thread $Th0$. Therefore, the thread $Th0$ have to wait while thread $Th1$ is operating, for a time (t_{wait}),

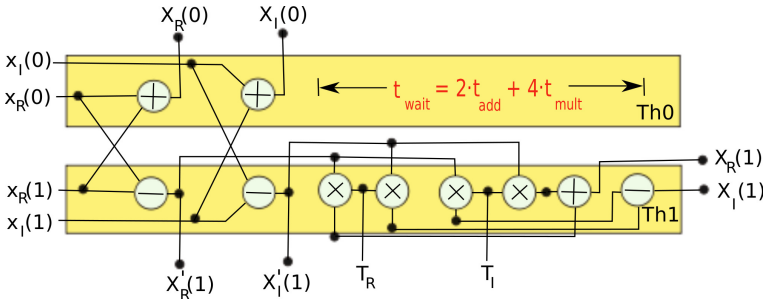

Figure 4.6. Schedule: Unbalanced

$$t_{wait} = 2 \cdot t_{add} + 4 \cdot t_{mult} \tag{4.2}$$

It is an unbalanced scheduling as the operations are not shared equally in both of the threads. The critical path and t_{wait} can be reduced by equal share of the operations in different threads.

2-word group using one thread

In this schedule, one thread is used for all the operations of a 2-word group as shown in Figure 4.7. The the critical path, CP, of this schedule is

$$CP = 6 \cdot t_{add} + 4 \cdot t_{mult} \tag{4.3}$$

There is no waiting time in this schedule, though the CP is increased than the unbalanced approach. The device code of this schedule is given in Appendix A.2.

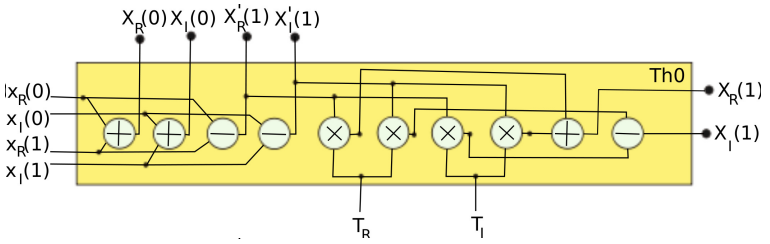

Figure 4.7. Schedule: 2-word group using 1 thread

2-word group using two threads

In this schedule, two threads are used for the operations of a 2-word group as shown in Figure 4.8. The operations of the lower edge such as two multiplications and one addition are shifted to the $Th0$, unlike unbalanced approach in order to reduce t_{wait} and CP. The CP of this schedule is

$$CP = 3 \cdot t_{add} + 2 \cdot t_{mult} \tag{4.4}$$

There is no waiting time in this schedule. The CP of this schedule is reduced to the half compared to the previous schedule. The device code of this schedule is given in Appendix A.3.

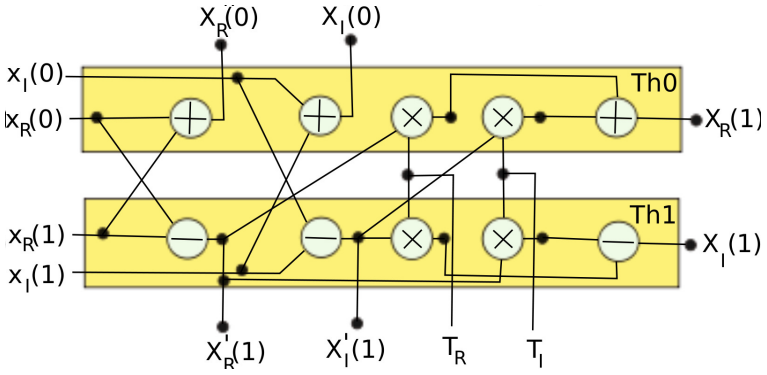

Figure 4.8. Schedule: 2-word group using 2 thread

2-word group using four threads

In this schedule, four threads are used for the operations of a 2-word group as shown in Figure 4.9. Each thread shares one addition or subtraction and one multiplication, and the threads $Th1$ and $Th3$ shares one addition and one subtraction respectively. The CP of this schedule is

$$CP = 2 \cdot t_{add} + 1 \cdot t_{mult} \tag{4.5}$$

Though the $t_{wait} = 1 \cdot t_{add}$, the CP is still almost reduced to the half of the previous schedule. The device code of this schedule is given in Appendix A.4.

4.2.3 Scheduling of the 4-word group

In this section, the operations of the 4-word group are scheduled using different threads.

4-word group using one thread

In this schedule, one thread is used for all the operations of the 4-word group (i.e. four 2-word groups) as shown in Figure 4.10. The critical path CP of this schedule is

$$CP = 4 \cdot (6 \cdot t_{add} + 4 \cdot t_{mult}) \tag{4.6}$$

There is no waiting time in this schedule, though the CP is high. The device code of this schedule is given in Appendix A.5.

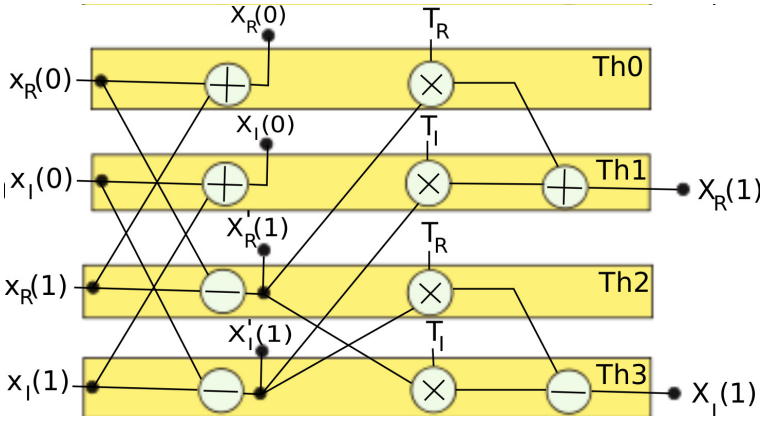

Figure 4.9. Schedule: 2-word group using 4 thread

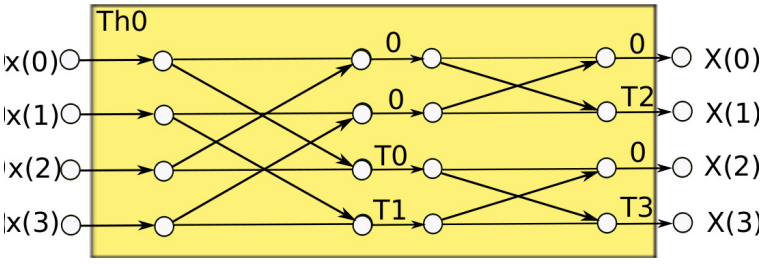

Figure 4.10. Schedule: 4-word group using one thread

4-word group using two threads

In this schedule, two threads are used for the operations of the 4-word group. Each thread shares two 2-word groups equally as shown in Figure 4.11. Here, the thicker lines represents the $Th0$ and the thinner lines represents the $Th1$. In this, the way that each thread shares the operations of the 2-word group is similar to the 2- word group using one thread (Section 4.2.2). Although, in this schedule each thread processes two 2-word groups sequentially. The critical path CP of this schedule is

$$CP = 2 \cdot (6 \cdot t_{add} + 4 \cdot t_{mult}) \tag{4.7}$$

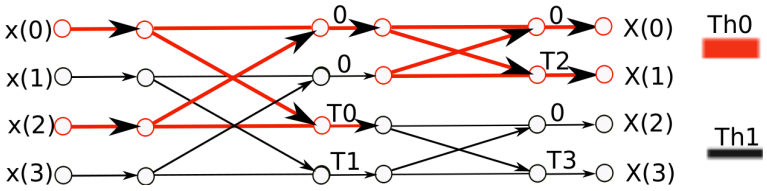

Figure 4.11. Schedule: 4-word group using two threads

The CP of this schedule is reduced to the half of the previous schedule. The device code of this schedule is given in Appendix A.6.

4-word group using four threads

In this schedule, four threads are used for the operations of the 4-word group. The four threads decomposes into two sets, as two threads in each set. Each set of threads shares the operations of the two 2-word groups as shown in Figure 4.12. In this, the way that each set of threads shares the operations of the 2-word group is similar to the 2- word group using two threads (Section 4.2.2). Yet, in this schedule each set of threads processes two 2-word groups sequentially. The critical path CP of this schedule is

$$CP = 6 \cdot t_{add} + 4 \cdot t_{mult} \tag{4.8}$$

The CP of this schedule is reduced still to the half of the previous schedule. The device code of this schedule is given in Appendix A.7.

4-word group using eight threads

In this schedule, eight threads are used for the operations of the 4-word group. In this, the eight threads decomposes into two sets, as four threads in each set. Each set of threads shares the operations of the two 2-word groups as shown in Figure 4.13. In this, the way that each set of threads shares the operations of the 2-word group is similar to the 2- word group using four threads (Section 4.2.2). However, in this schedule each set of threads processes two 2-word groups sequentially. The critical path of this schedule is

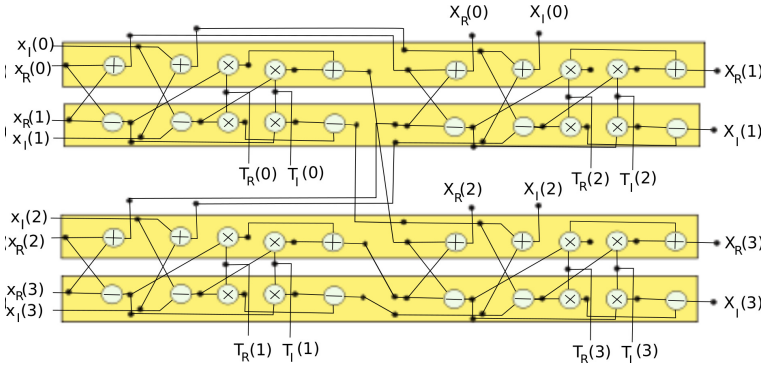

Figure 4.12. Schedule: 4-word group using four threads

$$CP = 4 \cdot t_{add} + 2 \cdot t_{mult} \tag{4.9}$$

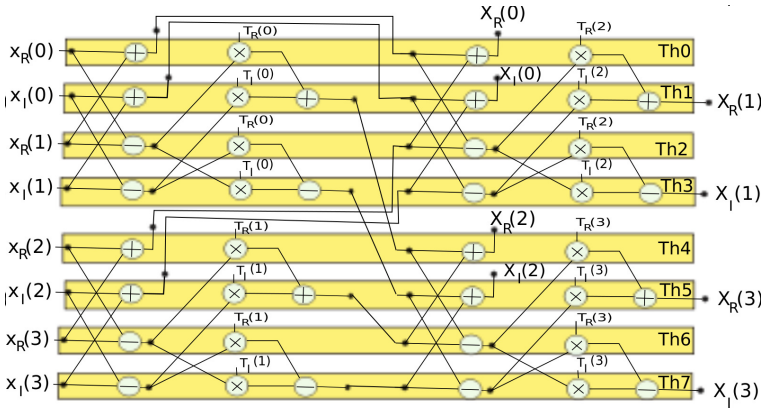

Figure 4.13. Schedule: 4-word group using eight threads

The CP of this schedule is reduced still almost to the half of the previous schedule. The device code of this schedule is given in Appendix A.8.

Similarly, the operations in the 8-word group can be scheduled using 1, 2, 4, 8, and 16 threads.

4.3 Implementation of the radix-2 and constant geometry

Section 3.3.2 explains that the flow graph of the FFT using constant geometry increases the speed up. The indexing is constant in all the stages (refer Figure 3.6). Therefore, the indexing is calculated once for the FFT algorithm rather than for every stage.

The implementation of this flow graph can be implemented similar to the FFT implementation using 2-word group (Section 4.2.1). The only difference is that not in place transformations are used in this implementation (i.e. the output elements of the 2-word group are not written at the locations, where the inputs of the 2-word group are read). However, it is also possible to implement this approach with different schedules of the 2-word group. One of the schedule, 2-word group using one thread is explained in this section with the device code.

Listing 4.1 shows the device code for the N-point radix-2 DIF FFT with constant geometry. In the implementation, one thread for each 2-word group is used. The indexing $x0$, $x1$, and $x2$ are calculated once before the loop, and used in all the stages. It resembles the constant geometry. The lines from 8 to 11, shows that the input elements of the 2-word group are read at locations $x0$, $x0+1$, $x1$ and $x1+1$. The lines from 13 to 16, show that the output elements of that 2-word group are written at locations $x2$, $x2+1$, $x2+2$, and $x2+3$. It means that the not in place transformations are used in this implementation.

Listing 4.1. N-point radix-2 DIF FFT with constant geometry

```
1  short j  = threadIdx.x;
2  short x0 = j<<1;
3  short x1 = (j<<1) + N;
4  short x2 = (j<<2);
5
6  for(short s=1; s<=n; s++)
7  {
8   SB[x2] = SA[x0] + SA[x1];
9   SB[x2+1] = SA[x0+1] + SA[x1+1];
10  SB[x2+2] = SA[x0] - SA[x1];
11  SB[x2+3] = SA[x0+1] - SA[x1+1];
12  short r0 = (j>>(s-1))*(1<<(s-1));
13  SA[x2] = SB[x2];
14  SA[x2+1] = SB[x2+1];
15  SA[x2+2] = SB[x2+2]*SROT[2*r0] + SB[x2+3]*SROT[2*r0+1];
16  SA[x2+3] = -SB[x2+2]*SROT[2*r0+1] + SB[x2+3]*SROT[2*r0];
17  __syncthreads();
18 }
```

4.4 Implementation of the radix-2^2

Section 3.1 explained that there are $n/2$ trivial stages in the N-point radix-2^2 FFT, where $n = log_2 N$. In the implementation of the radix-2^2, the multiplication by twiddle factors on even stages can be avoided by interchanging the real and imaginary parts and/or changing signs [14]. Therefore, the non trivial stages (i.e. odd stages) only have multiplications of the twiddle factors.

The radix-2^2 FFT can be implemented similar to the FFT implementation of the 4-word groups, so that the synchronization can be skipped after the trivial stages. Therefore, all the schedules used for 4-word group are applicable to implement the implementation of radix-2^2.

As there are many operations in each 4-word group (four 2-word groups), we focused to implement the 4-word group of radix-2^2 using high number of paralleled processing schedules, such as 4-word group using 4 threads and 8 threads (refer Sections 4.2.3). One of the schedules, the 4-word group using 4 threads is explained in this section with the device code.

Listing 4.2 shows the device code for the N-point radix-2^2 DIF FFT. In the implementation, 4 threads for each 4-word group are used. The lines 18 and 19 processes the first intermediate step of the 4-word group (followed by no synchronization). It can be noticed that there are no multiplications by the twiddle factors, as it is a trivial stage. The lines from 27 to 30 process the second intermediate stage of the 4-word group. It has multiplications with twiddle factors as it is a non-trivial stage (see lines 29 and 30).

Listing 4.2. N-point radix-2^2 DIF FFT

```
1
2  short i = j/2;
3  short k = j%2;
4  short h = i/2;
5  short l = j%4;
6  short m = i%2;
7  short w = m*k;
8
9  s = 1;
10 p0 = M/(1<<(2*s0));
11 x = 2*(h+(h/(1<<2*(n-s0)))*(1<<2*(n-s0))*(n-1));
12 x0 = x+l*p0;
13 //Iteration -1
14 sign1 = m*(-2)+1;
15 signw = w*(-2)+1;
16 x1 = x0+sign1*2*p0;
17 //step 1:
18 SB[x0+w] = signw*(SA[x1] + sign1*SA[x0]);
19 SB[x0+!w] = SA[x1+1] + sign1*SA[x0+1];
20
21 s = 2;
22 r1 = (1<<(s0-2))*((((ind1>>(n1-s0+1))%2)<<1)+
23       ((ind1>>(n1-s0+2)))%2)*((ind1>>1)%(1<<(n1-s0))));
24 x2 = x0+sign2*p0;
25 sign2 = k*(-2)+1;
26 //step 2:
27 SA[x0] = SB[x2] + sign2*SB[x0];
28 SA[x0+1] = SB[x2+1] + sign2*SB[x0+1];
29 SB[x0] = SA[x0]*SROT[2*r1] + SA[x0+1]*SROT[2*r1+1];
30 SB[x0+1] = -SA[x0]*SROT[2*r1+1] + SA[x0+1]*SROT[2*r1];
31 __syncthreads();
```

4.5 Implementation of the radix-2^2 and constant geometry

It was explained that the structure of the flow graphs of the radix- 2 and the radix-2^2 are the same. It was also explained that the radix-2 with constant geometry is the only change in the structure of the flow graph of the radix-2. It means that the flow graph of the radix-2^2 with constant geometry is the same as the radix-2 with constant geometry. The 16-point radix-2^2 with constant geometry is shown in Figure 4.14. The structure is the same as Figure 3.6.

We implemented this using 4-word groups similar to the previous section. In this section, the indexing of the 4-word groups in all the epochs is the constant, unlike the previous section. It can be noticed in Figure 4.14, that the indexing of the 4-word group in the *Epoch*0 (thicker or red colored lines) is the same as that in the *Epoch*1 (thicker or green colored lines).

We implemented the implementation of the N-point radix-2^2 constant geometry DIF FFT using the schedules, 4-word groups using 4 threads and 8 threads, similar to the previous section. Listing 4.3 shows the device code for the N-point

radix-2^2 DIF FFT with constant geometry. The schedule 4-word group using 4 threads is used in this listing.

In Listing 4.3, it can be noticed that the indexing $x0$, $x1$, and $x2$ are calculated once before the operations of the 4-word groups, which resembles the behavior of the constant geometry. There are no multiplications with the twiddle factors in the first intermediate step of the 4-word group, which resembles the radix-2^2.

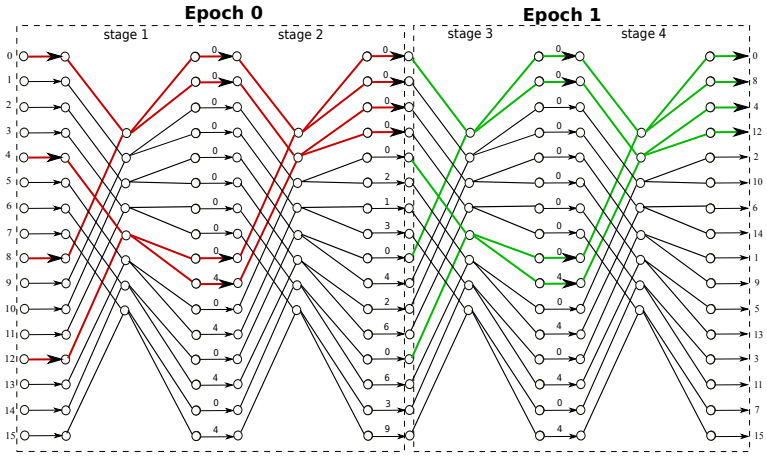

Figure 4.14. 4-word groups in the 16-point radix-2^2 constant geometry DIF FFT flow graph

Listing 4.3. N-point radix-2^2 DIF FFT with constant geometry

```
short j   = threadIdx.x;
short i = j>>1;
short k = j%2;
short x0 = i<<1;
short x1 = (i<<1) + N;
short x2 = j<<1;
short signk =-(k<<1) + 1;
short s;

//Iteration-1

r0 = k*(j>>(n-1));
signr0 = -(r0<<1) + 1;
//step 1:
SB[x2 + r0] = signr0*(SA[x0] + signk*SA[x1]);
SB[x2 + (!r0)] = SA[x0+1] + signk*SA[x1+1];

s=2;
short r1 = (((j%2)<<1) + ((j>>1)%2))*(j>>s);
//step 2:
SA[x2] = SB[x0] + signk*SB[x1];
SA[x2+1] = SB[x0+1] + signk*SB[x1+1];
SB[x2] = SA[x2]*SROT[r1<<1]+SA[x2+1]*SROT[(r1<<1)+1];
SB[x2+1] = -SA[x2]*SROT[(r1<<1)+1]+SA[x2+1]*SROT[r1<<1];
```

4.6 Pre-computation of the twiddle factors

The calculations of the twiddle factors are of high latency in the GPU. Instead, calculating them in the CPU, and storing in the shared memory leads to gain in the performance of the GPU code. However, this makes the performance loss of

the CPU code. This technique is applicable in order to gain the performance of the GPU code.

For an example, in Listing 4.4 the twiddle factors are calculated in GPU code (see lines 12 and 13), where as in Listing 4.5, the twiddle factors are calculated in the CPU code (see lines 6 and 7). Those results are shown in Table 4.1. On the one hand, the performance of GPU part is better in the first case compared to the later one. On the other hand, the performance of CPU is better in the later one, compared to the first one.

Listing 4.4. Twiddle factors calculations in the GPU

```
1  #define N 16
2  CPU Part
3  {
4  ----------------------------
5  ----------------------------
6  }
7
8  GPU Part
9  {
10 __shared__ SROT[N];
11 short tid = threadIdx.x;
12 SROT[2*tid] = cosf(tid*6.8/N);
13 SROT[2*tid+1] = sinf(tid*6.8/N);
14 ----------------------------
15 ----------------------------
16 }
```

Listing 4.5. Twiddle factors calculations in the CPU

```
1  #define N 16
2  CPU Part
3  {
4   for(short j=0; j<(N/2); j++)
5   {
6   ROT[2*j] = cosf(j*6.8/N);
7   ROT[2*j+1] = sinf(j*6.8/N);
8   }
9  }
10
11 GPU Part
12 {
13 __shared__ SROT[N];
14 short tid = threadIdx.x;
15 SROT[2*tid] = ROT[2*tid];
16 SROT[2*tid+1] = ROT[2*tid+1];
17 ----------------------------
18 ----------------------------
19 }
```

16-point DIF FFT	GPU part (μs)	CPU part (μs)	Tranfer time CPU to GPU (μs)	Total time (μs)
FFT: case-1	7.8	12	-	19.8
FFT: case-2	6.624	17	1	26.624

Table 4.1. Comparisons of the FFT results, where FFT: case-1 represents the twiddle factors calculated in GPU for the FFT implementation and FFT: case-2 represents the twiddle factors calculated in CPU for the FFT implementation.

4.7 Implementations of the FFT in multiple SMs

Section 2.4.2 explains that the access latency of the shared memory is very less as compared to the global memory. Therefore, it is very important to reduce the

usage of the global memory accesses as much as possible and increase the usage of the shared memory.

The memory space is required for data elements and twiddle factors for the implementation of the FFT. There are also different memory levels available in the GPU. We selected the shared memory as it is of low access latency. We used it as explained in Section 2.4.4. However, the limited shared memory in one SM limits the FFT size. One needs to use multiple SMs to implement larger sizes of the FFT. While processing the implementation of the FFT in multiple SMs, the data elements have to be moved from one SM to another SM. These data elements are moved through the global memory, as there is no internal communication between the shared memories of any two SMs. We designed the different flow graphs to use the FFT implementation in two SMs and four SMs, in order to use the least global memory.

4.7.1 Implementation of the FFT in two SMs

In the implementation of the N-point radix-2 FFT in two SMs, each SM processes with $N/2$ data elements. Each SM computes the $N/4$ butterflies of each stage. The shared memory of each SM is used to store the data elements and the twiddle factors required for the corresponding butterfly operations in each SM.

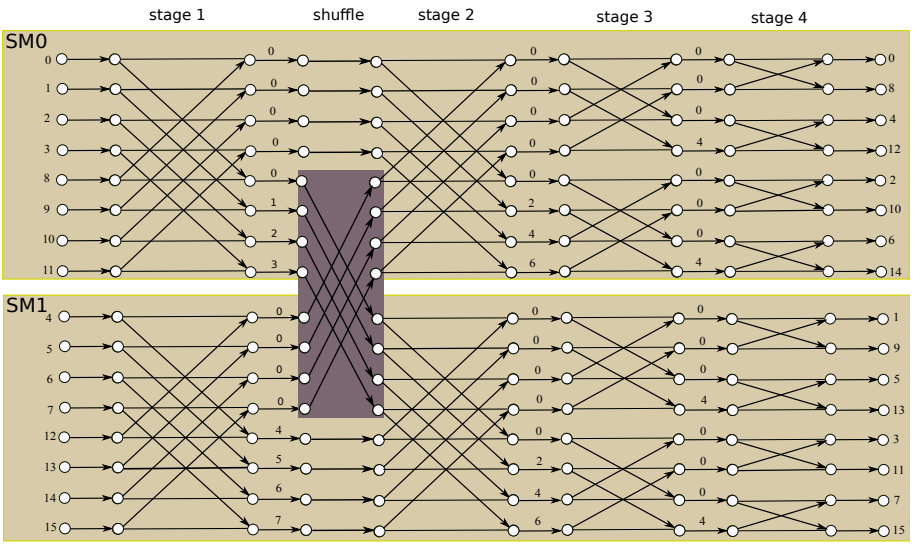

Figure 4.15. Flow graph of the 16-point radix-2 DIF FFT to implement in two SMs

The flow graph of the radix-2 DIF FFT algorithm to implement in two SMs is shown in Figure 4.15. The flow graph is divided into two parts, as each part for each SM. These two are connected by a shuffle, which is after the first stage of the FFT algorithm as shown in figure. This shuffling is nothing but the exchanging of data between two SMs. This is done through the global memory. The device code for this shuffling is given in Listing 4.6.

Listing 4.6. Data shuffle in two SMS

```
short j = threadIdx.x;
short b = blockIdx.x;
D = (N>>2) <-- number of threads.
//First stage.
.....................
.....................
__syncthreads(); //end of first stage.
//move data from shared memory to global memory.
A[j+b*(N>>1)] = SA[j+(!b)*(N>>1)];
A[j+b*(N>>1)+D] = SA[j+(!b)*(N>>1)+D];
__syncthreads();
//move data from global memory to shared memory.
SA[j+(!b)*(N>>1)] = A[j+(!b)*(N>>1)];
SA[j+(!b)*(N>>1)+D] = A[j+(!b)*(N>>1)+D];
__syncthreads();
```

4.7.2 Implementations of the FFT in four SMs

In the implementation of a N-point radix-2 FFT in four SMs, each SM processes with $N/4$ data elements. Each SM computes the $N/8$ butterflies of each stage. The shared memory of each stage is used to store the data elements and the twiddle factors required for the corresponding butterfly operations in each SM.

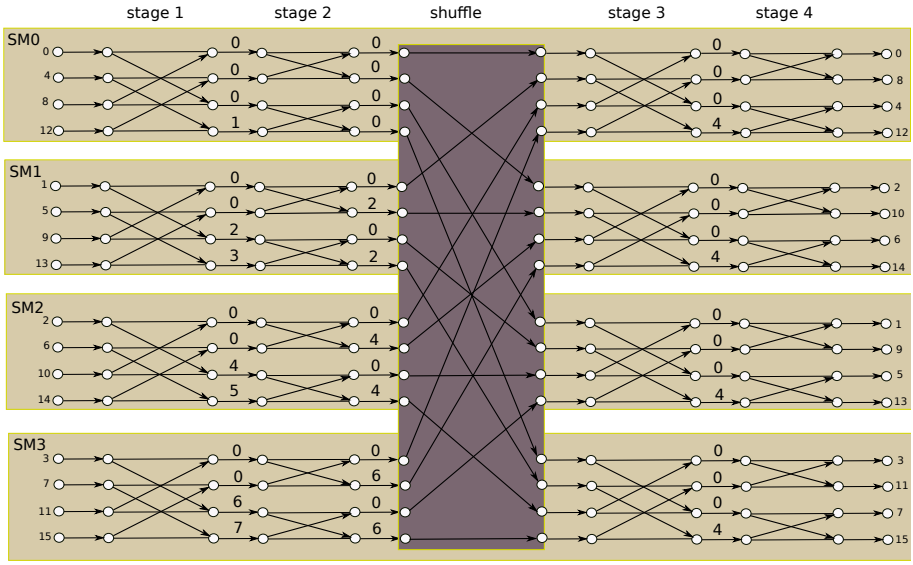

Figure 4.16. Flow graph of the 16-point radix-2 DIF FFT to implement in four SMs

The flow graph of the radix-2 DIF FFT algorithm to implement in four SMs is shown in Figure 4.16. The flow graph is divided into four parts, as each part for each SM, similar to the previous section. All these four parts are connected by a shuffle. This shuffle makes to move the data from one SM to another SM. This is done through global memory. The device code for this shuffling is given in Listing 4.7.

Listing 4.7. Data shuffle in four SMs

```
short j = threadIdx.x;
short b = blockIdx.x;
D = (N>>2) <-- number of threads.
//First stage.
.....................
.....................
__syncthreads(); //end of first stage.
//Second stage.
.....................
.....................
__syncthreads(); //end of second stage.
//move data from shared memory to global memory.
A[j+b*(D)]          = SA[j];
A[j+b*(D)+(N/2)]    = SA[j+D];
A[j+b*(D)+N]        = SA[j+2*D];
A[j+b*(D)+N+(N/2)] = SA[j+3*D];
__syncthreads();
//move data from global memory to shared memory.
SA[j] = A[j+b*(N>>1)];
SA[j+D] = A[j+b*(N>>1) + D];
SA[j+2*D] = A[j+b*(N>>1) + 2*D];
SA[j+3*D] = A[j+b*(N>>1) + 3*D];
__syncthreads();
```

4.7.3 The data synchronization in multiple SMs

The threads cooperate with each other in each thread block as explained in Section 2.4.3. Therefore, the synchronization of the data using ___syncthreads can also be done in each thread block (i.e. in one SM while execution).

The global transactions in the shuffling have to be completed before the processing of stages after the shuffling. However, the ___syncthreads() is valid with in each SM, which is not assured that all the global transactions by different SMs are completed before the processing of the stages after the shuffling.

Therefore, we used two kernels, the one for the processing of the stages before the shuffling and the global transactions in the shuffling and the other one for the processing of the stages after the shuffling. As the execution of the different kernels does in sequential, the completion of the first kernel assures that the global transactions in the shuffling is done. It can be shown from Listing 4.8.

Listing 4.8. The data synchronization using multiple kernels

```
//GPU Part
__global__ void kernel0(float* A ,float* ROT)
{
 Processes the stages before shuffling and does
 the transactions of data elements from shared
 memory to global memory in the shuffling.
 __syncthreads();
}

__global__ void kernel1(float* A ,float* ROT)
{
 Does the transactions of data elements from
 global memory to shared memory and processes
 the stages after the shuffling.
}

//CPU Part
{
 ........................
 ........................

 dim3 gridDim(4,1);
 dim3 blockDim(D,1);

 kernel0<<<gridDim , blockDim>>>(Ad,ROTd);

 kernel1<<<gridDim , blockDim>>>(Ad,ROTd);
 cudaMemcpy(A, Ad, memsize, cudaMemcpyDeviceToHost);
}
```

4.7.4 Constant geometry in the implementations of the FFT in multiple SMs

The constant geometry can also be used in the implementations of the FFT in multiple SMs. Figures 4.17 and 4.18 show the flow graph of the 16-point radix-2 DIF FFT using constant geometry to implement in two SMs and four SMs respectively. The structure of the stages after the shuffle are constant, which resembles the constant geometry. The stages before the shuffle, are not changed from Figure 4.15 and Figure 4.16 in order to not make the complexity in the shuffling.

Similarly, the constant geometry can be used in the implementations of the radix-2^2 in two SMs and four SMs.

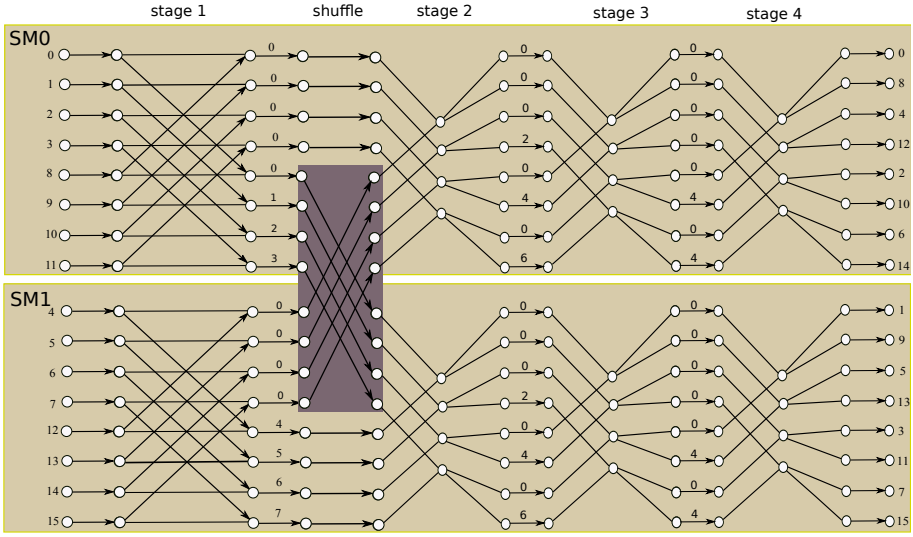

Figure 4.17. Flow graph of the 16-point radix-2 DIF FFT to implement in two SMs using constant geometry

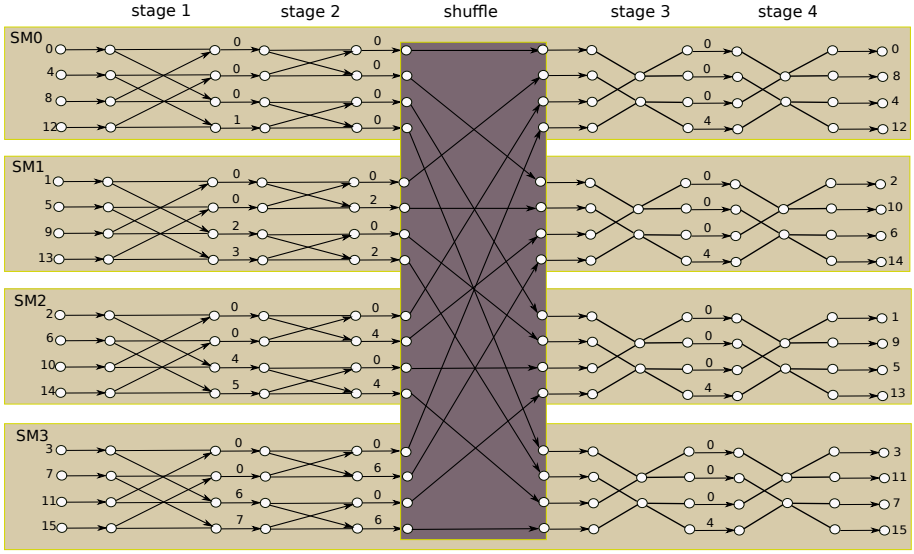

Figure 4.18. Flow graph of the 16-point radix-2 DIF FFT to implement in four SMs using constant geometry

Chapter 5

Results and Comparison

In this chapter, we report and discuss the experimental results conducted on the GPU, NVIDIA GeForce GTX 560 for all the implementations discussed in Chapter 4. The specifications of the GeForce GTX 560 are given in the Appendix A.1.

5.1 Experimental setup

The FFT alternatives proposed in Chapter 4 have been implemented and executed on a NVIDIA Geforce GTX 560, which consists of 7 SMs and a total of 336 processing elements running at 1620-1900 MHz. The GPU is connected to the host system. CUDA toolkit v4.0.17 is used, which mainly consists of compiler (nvcc), the required libraries, the visual profiler and the debugger.

We used the visual profiler for calculating the results, which provides the information such as the GPU execution time, the CPU execution time, and the transfer time between the CPU and the GPU, etc. We have taken the raw GPU execution time for our implementation and reported in this chapter.

5.2 Analysis of the GPU resources and influence on the FFT

The limited resources such as shared memory, number of threads, and registers limits the size of FFT. The resources in each SM in the Geforce GTX 560 are as the following,

- The available shared memory : 49152 bytes.
- The maximum number of threads : 1024.
- The available registers : 32768 bytes.

5.2.1 The influence of shared memory

As shown in Section 4.1, the data elements are read and write from/into memory before and after each stage of the radix-2 FFT. The two independent arrays are used in the implementation, one for reading and another for writing the data elements. As there are N complex data elements in the N-point radix-2 FFT, the size of each array is $2 \cdot N$.

The memory is also required for storing twiddle factors as explained in Section 4.6. As $N/2$ number of complex twiddle factors are used in the N-point radix-2 FFT, the array size of N. Therefore, the shared memory required for the N-point radix-2 FFT is $(5 \cdot N) \cdot WL_F$, where WL_F is data word length in floating point representation (i.e. 4 bytes).

If the radix-2 FFT is implemented in nSM number of SMs, then the maximum size of FFT (N_{max}) supportable in the limited shared memory is

$$(5 \cdot N_{max}) \cdot (4) \leq 49152 \cdot nSF \tag{5.1}$$

$$N_{max} \leq 2457.6 \cdot nSF \tag{5.2}$$

N number of complex twiddle factors are used in the N-point radix-2^2 FFT. In this, $2 \cdot N$ size of array is used for twiddle factors. Therefore, the shared memory required for the N-point radix-2^2 FFT is $(6 \cdot N) \cdot WL_F$.

If the radix-2^2 FFT is implemented in nSM number of SMs, then the maximum size of FFT (N_{max}) supportable in the limited shared memory is

$$(6 \cdot N_{max}) \cdot (4) \leq 49152 \cdot nSF \tag{5.3}$$

$$N_{max} \leq 2048 \cdot nSF \tag{5.4}$$

5.2.2 The influence of the number of the threads

If the N-point radix-2 (or radix-2^2) FFT is implemented using nT number of threads in nSM number of SMs, then the following condition should satisfy to execute the implementation in the Geforce GTX 560 in order to use the limited number of threads,

$$nT \leq 1024 \cdot nSM \tag{5.5}$$

If this condition fails the implementation is not executable in the Geforce GTX 560, which is called threads exceeds than limit (TE).

The number of threads used in the FFT implementation are dependent on the size of the FFT, because the threads are allocated for the butterfly operations as explained in Section 4.2.2. If we define $k = \frac{nT}{N}$ and the condition in the equation 5.5 is satisfied, then the size of the FFT is

$$N_{max} \cdot k \leq 1024 \cdot nSM \tag{5.6}$$

$$N_{max} \leq \frac{1024 \cdot nSM}{k} \tag{5.7}$$

For example, if $k = \frac{1}{2}$ (or $nT = \frac{N}{2}$) and $nSM = 1$, then $N_{max} \leqslant 2048$

5.2.3 The influence of the registers

The number of registers is dependent on the number of threads in each implementation. As the CUDA code executes on each thread, the registers are shared per each thread.

If the N-point radix-2 (or radix-2^2) FFT is implemented using nT number of threads in nSM number of SMs, such that the registers per thread is RT, then the condition to use the limited registers is

$$nT \cdot RT \leq 32768 \cdot nSM \tag{5.8}$$

If this condition fails then the implementation is not executable in the GPU, which is called register spilling (RS).

Once, if this condition is satisfied, then the maximum size of the FFT is

$$N_{max} \cdot k \cdot RT \leq 32768 \cdot nSM \tag{5.9}$$

$$N_{max} \leq \frac{32768 \cdot nSM}{k \cdot RT} \tag{5.10}$$

5.3 Results and comparisons

The implementations explained in the previous chapter are experimented in one, two and four SMs. Therefore, the results to be reported and compared will be done in the following manner,

- The implementations in one SM.
- The implementations in two SMs.
- The implementations in four SMs.

5.3.1 The implementations in one SM

The implementations in Sections 4.2, 4.3, 4.4, and 4.5 are experimented in one SM. Those results of the N-point radix-2 DIF FFT are reported in Table 5.1.

Few of the implementations are not supportable, specifically on NVIDIA Geforce GTX560, for such the results are mentioned with notations TE, RS, and NP. The notations used in Table 5.1 are abbreviated in Table 5.2. Each SM consists of limited resources.

Few of the implementations are only supportable with the specific sizes of the FFT. For example, the size of radix-2^2 FFT should be power of four. The rest of the sizes of radix-2^2 FFT are notified with NP, refers not portable.

Implementation	Size of FFT (N)							
	16	**32**	**64**	**128**	**256**	**512**	**1024**	**2048**
2 WG using 1 T	6.58	8.1	12.06	13.44	14.98	17.41	23.78	43.2
2 WG using 2 T	6.05	7.52	8.45	9.66	11.23	14.18	26.72	TE
2 WG using 4 T	6.08	6.43	7.17	8.67	10.7	19.01	TE	TE
4 WG using 1 T	6.62	NP	14.91	NP	27.97	NP	34.46	NP
4 WG using 2 T	4.99	NP	10.21	NP	12.80	NP	19.94	NP
4 WG using 4 T	5.02	NP	8	NP	11.46	NP	RS	NP
4 WG using 8 T	4.7	NP	6.99	NP	10.43	NP	TE	NP
8 WG using 1 T	NP	NP	24.42	NP	NP	86.56	NP	NP
8 WG using 2 T	NP	NP	13.47	NP	NP	26.53	NP	NP
8 WG using 4 T	NP	NP	9.92	NP	NP	15.33	NP	NP
8 WG using 8 T	NP	NP	7.74	NP	NP	14.59	NP	NP
8 WG using 16 T	NP	NP	6.75	NP	NP	RS	NP	NP
2 WG using 1 T and CG	4.7	5.76	7.39	8.19	9.04	10.4	14.34	25.18
2 WG using 2 T and CG	4.61	5.6	6.2	6.91	7.84	9.95	17.66	TE
2 WG using 4 T and CG	5.53	5.4	5.9	6.82	8.78	12.96	TE	TE

Table 5.1. The GPU time (μs) for different implementations of the N-point radix-2 DIF FFT in one SM

T : Thread/Theads.
WG: Work group.
BF: Butterfly.
TE: Threads exceeds than limit.
RS: Registers spilling.
NP: Not portable.
CG: Constant Geometry
P : Performance

Table 5.2. Notation used in Table 5.1

The results of the different implementations in Table 5.1 are compared, mainly "GPU time (μs) versus the number of threads and the size of FFT".

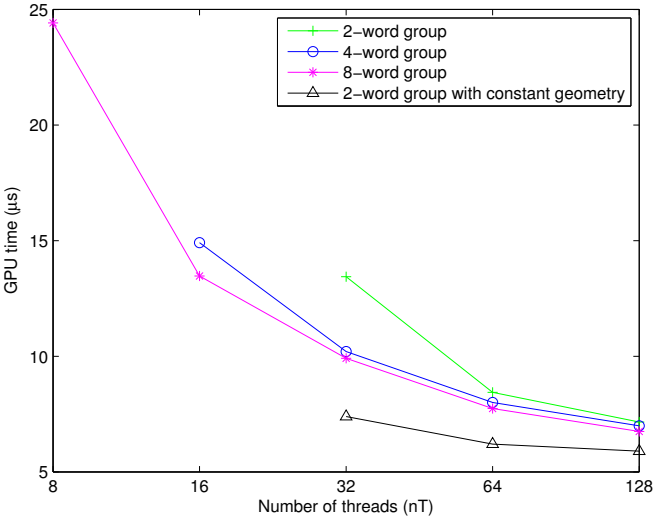

Figure 5.1. 64-point radix-2 DIF FFT with different threads

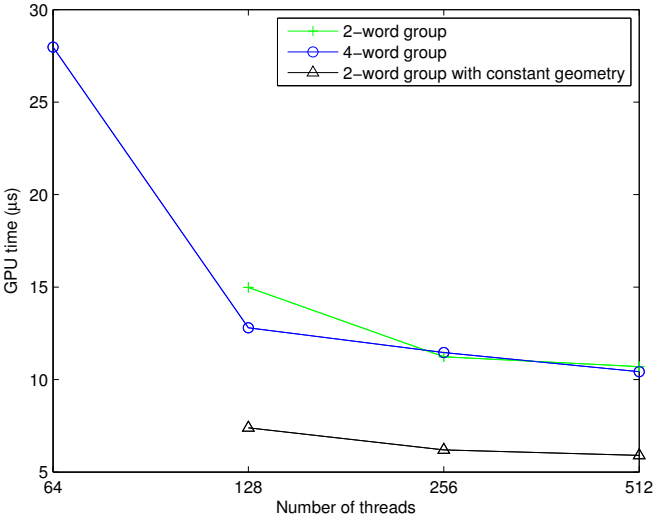

Figure 5.2. 256-point radix-2 DIF FFT with different threads

The results reported in Table 5.1 are compared in different ways.

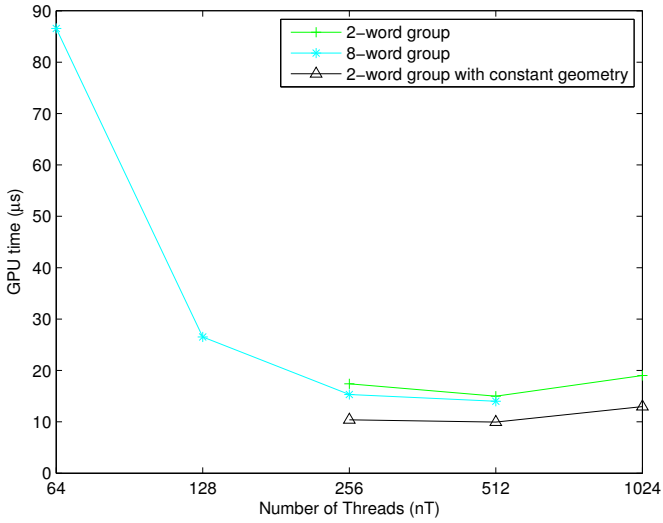

Figure 5.3. 512-point radix-2 DIF FFT with different threads

The sizes of the radix-2 DIF FFT are fixed to 64, 256, and 512, and then compared their performance with different number of threads, those are represented from graphs shown in Figures 5.1, 5.2, and 5.3. In each graph, different implementations of the radix-2 DIF FFT, such as 2-word group, 4-word group, 8-word group, and constant geometry are compared, with respect to the influence of the threads. The observations from these graphs are as follows:

- The performance of the constant geometry is the best. It is because of the the constant index in all the stages.
- At a glance, the performance is getting better as the number of the threads increases. Therefore, it is better to experiment with different number of threads in each implementation.
- If N is the size of the FFT and nT is the number of threads, then we can notice in these (three) graphs that it is a very less performance difference between the cases $nT = N$ and $nT = 2 \cdot N$. It means that the performance does not increase with higher number of threads, specifically after $nT = N$.

As the implementations with constant geometry showing the interesting performance, it is further compared with all the sizes of the radix-2 DIF FFT, which implemented with different number of threads. The graph is shown in Figure 5.4. The conclusions are as follows:

- The GPU time is the higher, the larger the size of the FFT. This is because of the number of computations increases as the larger the size of the FFT.

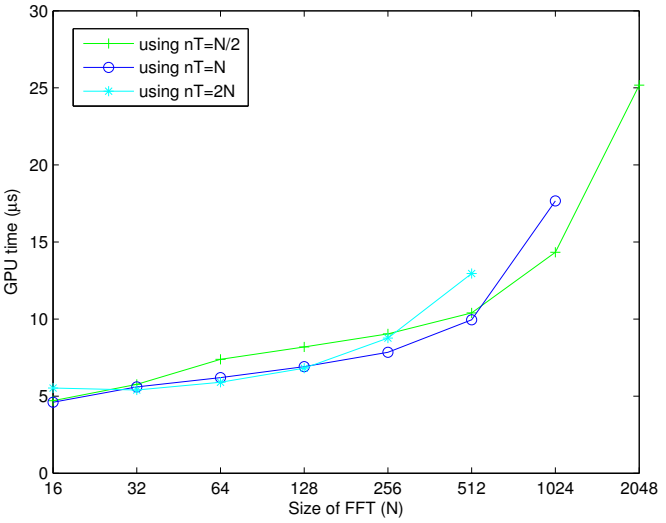

Figure 5.4. N-point radix-2 DIF FFT with constant geometry

- As the larger the size of FFT, the performance is better with lesser number of threads.
- The performance of the implementations with the case, number of threads $nT = N$ is better than others up to the size of FFT, $N = 512$.
- The performance of the implementations with the case, number of threads $nT = N$ and $nT = 2 \cdot N$ is almost the same in the smaller FFTs.

The results of the N-point radix-2^2 DIF FFT with and with out Constant Geometry (CG) are reported in Table 5.3, and their comparisons represented by the graph shown in Figure 5.5. At a glance, the plots of the implementations, with and with out CG looks similar. The performance with the case CG is better than the case without CG.

Implementation	**Size of FFT (N)**			
	16	**64**	**256**	**1024**
4 WG using 4 T	4.19	5.82	8.42	22.98
4 WG using 8 T	4	5.38	9.15	TE
4 WG using 4 T and CG	3.55	4.22	5.12	11.1
4 WG using 8 T and CG	3.55	3.9	5.79	TE

Table 5.3. The GPU time (μs) of the N-point radix-2^2 DIF FFT in one SM

The best cases of radix-2 and radix-2^2 with CG in one SM are reported in Table 5.4. The comparison of radix-2 and radix-2^2 with CG, and CUFFT is represented

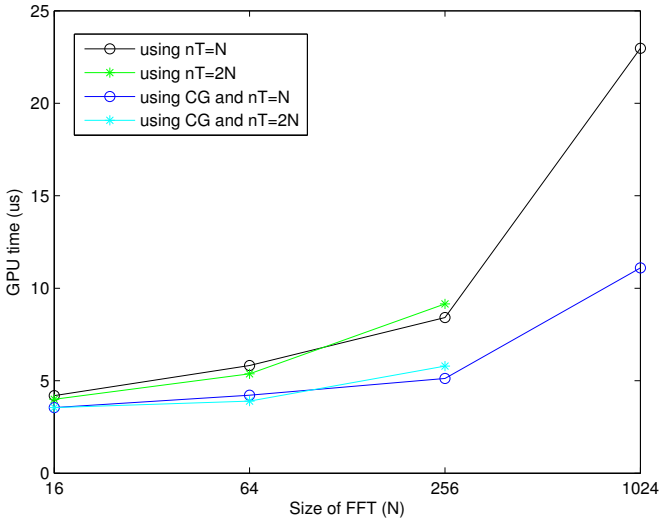

Figure 5.5. N-point radix-2^2 4-word group DIF FFT

by the graph shown in Figure 5.6. The CUFFT is the specific library to compute FFT on NVIDIA GPUs. The CUFFT library provides efficient computing parallel FFTs on an NVIDIA GPU [17].

The conclusions for the graph as the following,

- The performance of the implementations with radix-2^2 are better than radix-2.
- The performance for the sizes of FFT, 16, 32, 64, 128, 256 are better than the CUFFT. Further, the performance gain is more for the sizes of FFT, 16, 64,and 256 (power of 2^2), than sizes 32 and 128 (power of 2). This is because of the radix-2^2, which has multiplications on only even stages.

Size of FFT (N)	**radix-2**		**radix-2^2**	
	nT	**P (μs)**	**nT**	**P (μs)**
16	16	4.61	32	3.55
32	64	5.4	-	NP
64	128	5.9	128	3.9
128	256	6.82	-	NP
256	256	7.84	256	5.12
512	512	9.95	-	NP
1024	512	14.34	1024	11.04
2048	1024	25.18	-	NP

Table 5.4. Best cases of radix-2 and radix-2^2 with CG in one SM

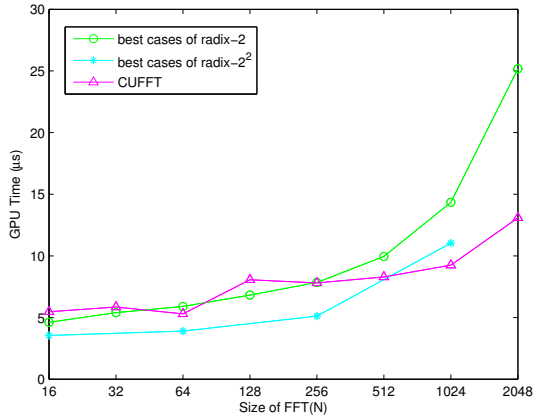

Figure 5.6. The comparison of the best cases of radix-2 and radix-2^2 with CG in one SM and CUFFT.

5.3.2 The implementations in two SMs

The results of the N-point radix-2 and radix-2^2 with CG are reported in Table 5.5 and these results are compared from the graphs in Figures 5.7 and 5.8 respectively. The information about the notations used in Table 5.5 is the same as given in Section 5.3.1.

Implementation	Size of FFT (N)								
	16	**32**	**64**	**128**	**256**	**512**	**1024**	**2048**	**4096**
radix-2									
2 WG using 1 T	5.88	6.36	7.90	10.88	12.9	13.5	16.48	21.47	37.31
2 WG using 2 T	5.6	5.79	7.23	7.80	8.60	9.88	12.74	21.73	TE
2 WG using 4 T	5.18	5.22	5.69	5.88	7.26	8.06	14.05	TE	TE
radix-2^2									
4 WG using 4 T	4.35	NP	5.25	NP	5.426	NP	9.24	NP	TE
4 WG using 8 T	4.29	NP	4.8	NP	5.2	NP	11.616	NP	TE

Table 5.5. The GPU time (μs) for the N-point radix-2 and radix-2^2 DIF FFT with constant geometry in two SMs

The conclusions for the N-point radix-2 with CG in two SMs are as follows:

- At a glance, the performance is better, the higher the number of threads. Therefore, it is the better to implement the FFT with the higher number of threads in two SMs also. This is because of the global memory transactions in the implementation, while shuffling. Therefore, the higher the number of threads, the faster the global memory transactions.
- The larger the size of FFT, the better the performance with the lesser number of threads.

The conclusions for the N-point radix-2^2 with CG in two SMs are the same as that of radix-2.

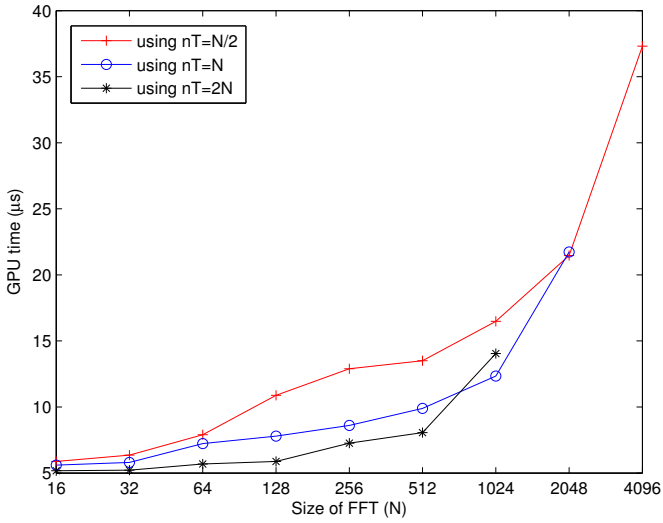

Figure 5.7. N-point radix-2 DIF FFT with CG in two SMs

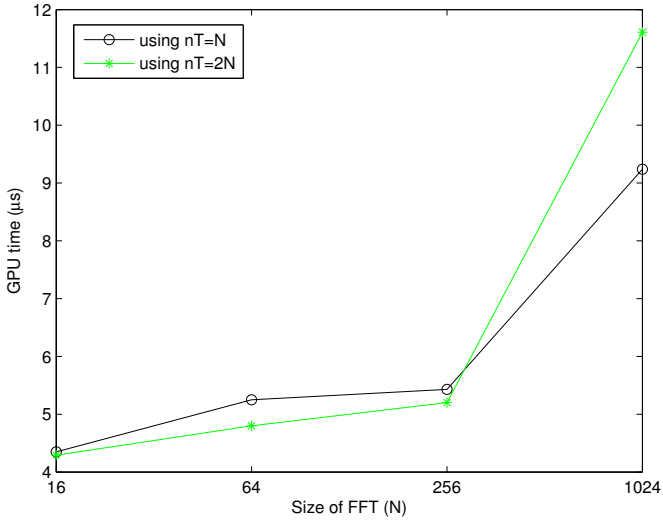

Figure 5.8. N-point radix-2^2 DIF FFT with CG in two SMs

5.3.3 The implementations in four SMs

The results of the N-point radix-2 and radix-2^2 with CG are reported in Table 5.6. The information about the notations used in Table 5.6 is also the same as given in Section 5.3.1. The results of the N-point radix-2 with CG and radix-2^2 are shown from the graphs in Figure 5.9.

Implementation	Size of FFT (N)								
	16	**32**	**64**	**128**	**256**	**512**	**1024**	**2048**	**4096**
radix-2									
2 WG using 1 T	5.94	7.86	8.79	10.5	13.23	14.1	15.7	19.2	24.7
2 WG using 2 T	5.6	6.2	6.62	9.09	9.73	10.69	12.13	15.3	25.82
radix-2^2									
4 WG using 4 T	5.7	NP	6.5	NP	7.4	NP	9.2	NP	20.7

Table 5.6. The GPU time (μs) for the N-point radix-2 and radix-2^2 DIF FFT with constant geometry in four SMs

The conclusions for the N-point radix-2 with CG in four SMs are also the same as that of radix-2 with CG in two SMs.

As shown in Figure 5.9, there is only one plot for representing the results of the N-point radix-2^2 with CG, which is with the case $nT = N$. We do not have the results for the case $nT = 2 \cdot N$, as it is left for the future work. However, the conclusions of this plot is that the larger the size of the FFT, the higher the performance.

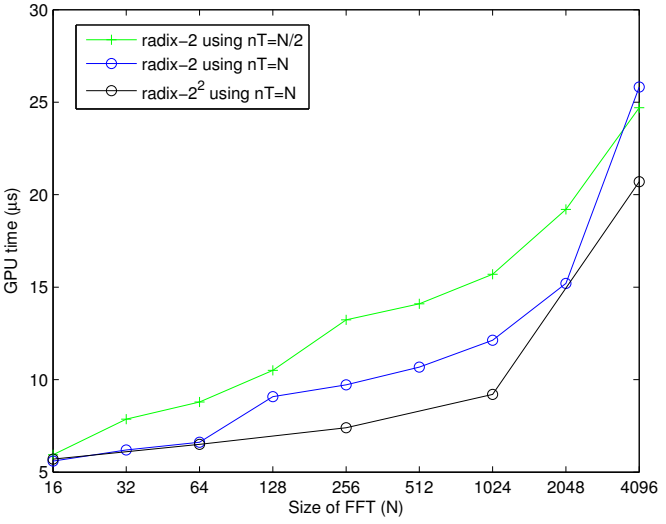

Figure 5.9. N-point radix-2 and radix-2^2 DIF FFT with CG in four SMs

The best cases in one SM, two SMs, and four SMs are reported in Table 5.7 and it is plotted in the graph shown in Figure 5.10. The same is compared with

CUFFT in the graph shown in Figure 5.11. The conclusions of these two graphs are as the following,

- As the smaller the sizes of FFT, the performance is the better in one SM and two SMs than that of four SMs. It is because of no global transactions in the implementations in one SM.
- As the larger the size of the FFT, the better the performance in the higher number of SMs. It is because of more computations in the larger sized FFTs and also the more computational area as the higher the number of SMs.
- At a glance, all the plots looks like zigzag. The performance of the sizes with even powers of 2 are more interesting than those of odd powers of 2. This is because of the implementations with even powers of 2 are radix-2^2, where as odd powers of are radix-2.
- It can be noticed in Figure 5.11 that the performance of our implementations is interesting compared to the CUFFT, up to the FFT size 1024.

	One SM			Two SMs			Four SMs		
N	Algorithm	nT	P (μs)	Algorithm	nT	P (μs)	Algorithm	nT	P (μs)
16	radix-2^2	16/32	3.55	radix-2^2	32	4.9	radix-2	32	5.6
32	radix-2	64	5.4	radix-2	64	5.22	radix-2	64	6.2
64	radix-2^2	128	3.9	radix-2^2	128	4.8	radix-2^2	128	6.5
128	radix-2	256	6.82	radix-2	256	5.88	radix-2	256	9.09
256	radix-2^2	512	5.12	radix-2^2	512	5.2	radix-2^2	512	7.4
512	radix-2	512	9.95	radix-2	1024	8.06	radix-2	1024	10.69
1024	radix-2^2	1024	11.04	radix-2^2	1024	9.24	radix-2^2	2048	9.2
2048	radix-2	1024	25.18	radix-2	1024	21.47	radix-2	4096	15.3
4096	-	-	-	radix-2	2048	37.31	radix-2^2	8192	20.7

Table 5.7. Best cases in all SMs

Our performance is better than CUFFT performance, from the size of FFT, 16 to 1024. The performance gain of "our FFT" than "CUFFT" is reported in Table 5.8, and it's graph is represented from Figure 5.12.

The performance gain is calculated as,

$$Gain = \frac{(t_{CUFFT} - t_{PROPOSED}) \cdot 100}{t_{CUFFT}} \tag{5.11}$$

where t_{CUFFT} is the GPU time for the CUDA FFT, and $t_{PROPOSED}$ is the GPU time for our FFT.

The peak performance gain is the 35% for the FFT sizes 16 and 256.

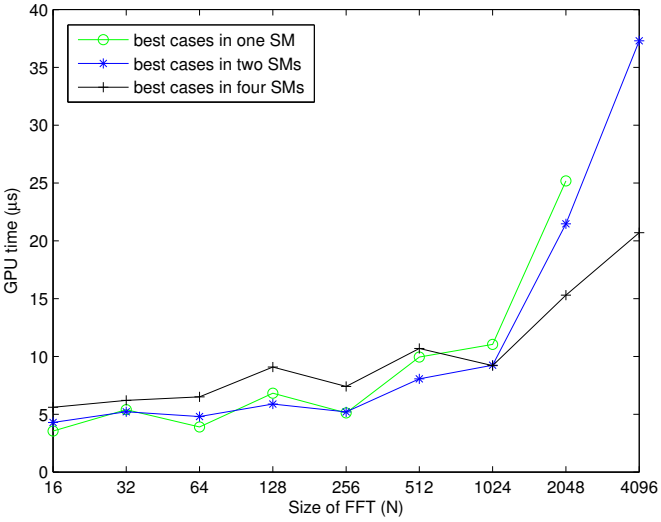

Figure 5.10. Best cases in all SMs comparison

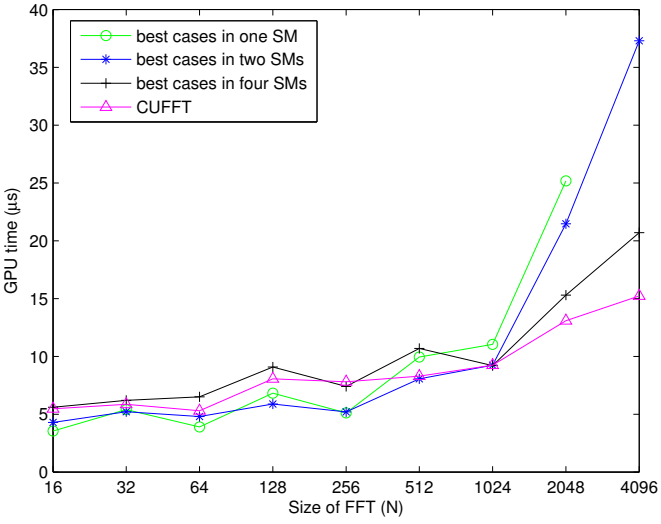

Figure 5.11. Best cases in all SMs comparison with CUFFT

Size of FFT (N)	Gain (%)
16	35.12
32	10.87
64	26.41
128	27.08
256	34.42
512	2.75
1024	0.52

Table 5.8. The gain of our FFT from CUFFT

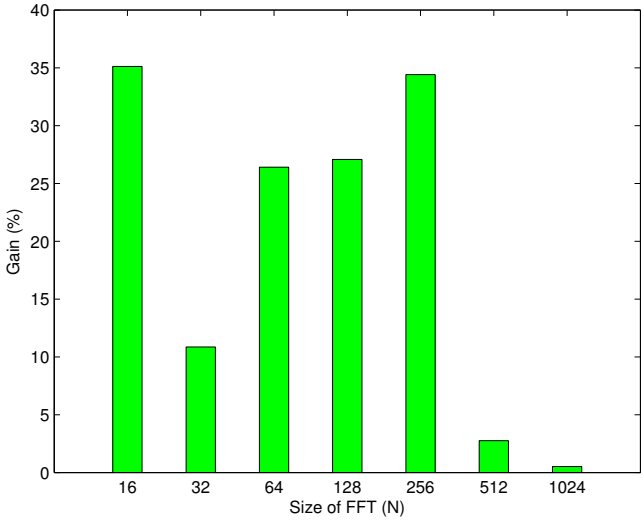

Figure 5.12. The gain of our FFT from CUFFT

Chapter 6

Conclusion and Future work

This chapter discusses the conclusions drawn from the proposed work and the directions to the future work. The summary of thesis work is

- Analysis of the FFT algorithm, the GPU hardware, and the CUDA programming language.
- General controlling of the data for the implementation of the FFT algorithm using parallel and independent threads.
- Design and implementation of the radix-2 and radix-2^2 FFT algorithm using word groups with different schedules.
- Design and implementation of the radix-2 and radix-2^2 FFT with and with out constant geometry using word groups and schedules.
- Design and implementation of the radix-2 and radix-2^2 FFT with and with out constant geometry in multiple SMs, with less global transactions.

6.1 Conclusion

A very high performance FFT are implemented on a modern GPU. The performance gain of our algorithms over CUFFT is due to the following optimizations:

- Usage of maximal shared memory for accessing both twiddle factors and data elements.
- Usage of the low latency instructions and no execution paths in the implementations.
- Usage of the constant geometry FFT algorithm which simplifies index.
- Usage of radix-2^2 constant geometry to simplify certain stages.
- Usage of word groups with balanced scheduling, which distributes related calculations among several threads while reducing the synchronization points.

- Usage of least global transactions in the implementation of FFTs in several SMs

6.2 Future work

The future work in order to explore the proposed work are:

- Implementation of the bit reversal step to the output of the DIF FFT.
- Usage the constant memory for the twiddle factors.

  The constant memory in the GPU has read only access. In the proposed work we have used the shared memory for the storage of data elements and twiddle factors. While execution, the data elements are updated, where as the twiddle factors are used only for read and are not updated. Therefore, the usage of the constant memory for the twiddle factors is perfectly supported.
- Computing twiddle factors with low latency on the GPU. The calculation of the twiddle factors on the GPU is of high latency. The computations of the twiddle factors with less latency will lead to performance improvement.
- Different implementations for large size FFTs by the usage of multiple SMs (i.e. more than four SMs).
- The implementations using 16-word groups.

  In the proposed work, the FFT algorithms are implemented using 2-word group, 4-word group, and 8-word group. This can be extended using 16-word group.
- OpenCL (Open Computing Language) [28] is a parallel programming language similar to CUDA. In addition, it can be used to execute over the cross platforms such as CPU and GPU. Recently, a new version of the OpenCL standard for FPGAs has been released [31]. In order to take advantage of this new release, the experiments with the implementation of the FFT algorithm using OpenCL in FPGAs are:

  1. The implementation of the FFT algorithm in FPGA and GPU using OpenCL.

  In this, different implementations of the FFT algorithm will be implemented using OpenCL in order to optimize the power and the performance. The design will be executed on both the devices FPGA and GPU. The power and performance results will be compared in both the devices.

  2. The implementation of the FFT algorithm in FPGA using OpenCL and VHDL.

  In this, the FPGA designs of the different implementations of the FFT algorithm are implemented using both OpenCL and VHDL. These two FPGA designs will be tested in the FPGA and will be compared in terms of area, throughput, and power consumption. The purpose is to compare the trade off between time to market and performance of VHDL and OpenCL [30].

Appendix A

A.1 GeForce GTX 560 specifications

Table 5.6 shows the specifications of the GeForce GTX 560.

Number of SMs	7
Total CUDA cores	336
Processor clock rate (MHz)	1620-1900
Device memory size	1024 MB
Device memory bandwidth (GB/sec)	128
Memory clock (MHz)	2002-2200
Number of registers availble per SM	32768
Amount of shared memory per SM	49152 bytes
Number of threads per warp	32
Maximum size of threads allowed per thread block	1024

Table A.1. GeForce GTX 560 technical specifications

A.2 2-word group using one thread

Listing A.1. 2-word group using one thread

```
1  short j = threadIdx.x;
2  short n = logf(N)/logf(2);
3  ........................
4  ........................
5  for(short s= 1; s<= n ; s++)
6  {
7   short p = (2*N)>>s;
8   short x = 2*j + ((j>>(n-s))<<(n-s+1));
9   short x1 = x + p;
10  SB[x] = SA[x] + SA[x1];
11  SB[x+1] = SA[x+1] + SA[x1+1];
12  SB[x1] = SA[x] - SA[x1];
13  SB[x1+1] = SA[x+1] - SA[x1+1];
14  short r = (j%(1<<(n-s)))*(1<<(s-1));
15  SA[x] = SB[x];
16  SA[x+1] = SB[x+1];
17  SA[x1] = SB[x1]*SROT[2*r] + SB[x1+1]*SROT[2*r+1];
18  SA[x1+1] = -SB[x1]*SROT[2*r+1] + SB[x1+1]*SROT[2*r];
19 }
```

In Listing A.1, each iteration of the loop performs each epoch (stage). If one thread is used for each 2-word group, then the total number of threads used for a N-point radix- 2 DIF FFT is equal to $N/2$. It can be noticed that two additions, two subtractions are calculated by lines from 10 to 13. The 2 sets of two multiplications and one addition or subtraction are calculated from lines 17 and 18. The data in temporary array SB is moved to the array SA in order to use SA as input data in the next stage (see lines 15 and 16).

A.3 2-word group using two threads

Listing A.2. 2-word group using two threads

```
1  short j   = threadIdx.x;
2  short i = j/2;
3  short k = j%2;
4  ..........................
5  ..........................
6  for(short s= 1; s<= n ; s++)
7  {
8   short p = (2*N)<<s;
9   short sign = (-2)*k+1;
10  short x_tmp = 2*i+ (i>>(n-s))*(1<<(n-s+1));
11  short x = x_tmp + k*p;
12  short x1 = x+sign*p;
13  SB[x] = SA[x1] + sign*SA[x];
14  SB[x+1] = SA[x1+1] + sign*SA[x+1];
15  short r0 = (i%(1<<(n-s)))*(1<<(s-1));
16  short inx1 = x -k*p +k;
17  short inx2 = x +(!k)*p;
18  SA[inx1] = SB[inx1];
19  SA[inx2+k] = sign*SB[inx2]*SROT[2*r0+k] +
20                  SB[inx2+1]*SROT[2*r0+(!k)];
21 }
```

In Listing A.2, the variable k differentiates even and odd of the two threads, those sharing any of the 2-word group operations. The variable *sign* is the positive for even threads, where as the negative for odd threads (see line 9). The even threads share the operations two additions, where as odd threads share two subtractions (see lines 13 and 14). Each thread also shares a set of two multiplications and one addition or subtraction (see lines 19 and 20).

A.4 2-word group using four threads

Listing A.3. 4 threads sharing operations of one butterfly

```
1  short j = threadIdx.x;
2  short i = j/2;
3  short k = j%2;
4  short l = i/2;
5  short m = i%2;
6  for(short s= 1; s<= n; s++)
7  {
8   short p = (2*N)/(1<<s);
9   short sign0 = k*(-2)+1;
10  short sign1 = m*(-2)+1;
11  short x_tmp = 2*l + (l/(1<<(n-s)))*(1<<(n-s+1));
12  short x = x_tmp + m*p;
13  short r0 = (l%(1<<(n-s)))*(1<<(s-1));
14  short x1 = x + sign1*p;
15  short inx_tmp0 = x + (!m)*p;
16  SB[x+k] = SA[x1+k] + sign1*SA[x+k];
17  SC[x+k] = SB[inx_tmp0+k]*SROT[2*r0+m];
18  SA[x+k] = (!m)*SB[x+k] + m*(SC[x1+k]+sign0*SC[x+!k]);
19 }
```

Listing A.3 shows the implementation of the N-point radix-2 DIF FFT using the 2-word groups, each using four threads. As four threads are sharing the operations of each 2-word group, the total number of threads used for the N-point radix-2 DIF FFT is $2 \cdot N$. The variables i, k, l,and m are used to differentiate the four threads(see lines from 1 to 5).

Each thread shares one addition or subtraction (see line 16). The result is stored in a temporary variable SB. Each thread also shares one multiplication and stores in the temporary variable SC (see line 17). Two of four threads share *move* operation from SB to SA, and the rest of the two processes an addition (see line 18).

A.5 4-word group using one thread

Listing A.4 used one thread for each 4-word group, so total number of threads used for the implementation of the N-point radix-2 DIF FFT is equal to $N/4$. One can understand that there are two butterfly computations in each stage (lines from 18 to 36 and from 39 to 57).

Listing A.4. 4-word group using one threa

```
1  short j   = threadIdx.x;
2  short n = logf(N)/logf(4);
3
4  for(short s=1; s<=n; s++)
5  {
6   short p = M/(1<<(2*s));
7   x0 = 2*(j+(j/(1<<2*(n-s)))*(1<<2*(n-s))*3);
8   x1 = x0+p;
9   x2 = x1+p;//x0+2p
10  x3 = x2+p;//x0+3p
11
12  r0 = (j%(1<<2*(n-s)))*(1<<2*(s-1));
13  r1 = r0+(N/4);
14  r2 = 2*r0;
15  r3 =r2;
16
17  //step 1
18  SB[x0]   =  SA[x0] + SA[x2];
19  SB[x0+1] =  SA[x0+1] + SA[x2+1];
20  SB[x2]   =  SA[x0] - SA[x2];
21  SB[x2+1] =  SA[x0+1] - SA[x2+1];
22
23  SA[x0]   =  SB[x0];
24  SA[x0+1] =  SB[x0+1];
25  SA[x2]   =  SB[x2]*SROT[2*r0] + SB[x2+1]*SROT[2*r0+1];
26  SA[x2+1] =  -SB[x2]*SROT[2*r0+1] + SB[x2+1]*SROT[2*r0];
27
28  SB[x1]   =  SA[x1] + SA[x3];
29  SB[x1+1] =  SA[x1+1] + SA[x3+1];
30  SB[x3]   =  SA[x1] - SA[x3];
31  SB[x3+1] =  SA[x1+1] - SA[x3+1];
32
33  SA[x1]   =  SB[x1];
34  SA[x1+1] =  SB[x1+1];
35  SA[x3]   =  SB[x3]*SROT[2*r1] + SB[x3+1]*SROT[2*r1+1];
36  SA[x3+1] =  -SB[x3]*SROT[2*r1+1] + SB[x3+1]*SROT[2*r1];
37
38  //step 2
39  SB[x0]   =  SA[x0] + SA[x1];
40  SB[x0+1] =  SA[x0+1] + SA[x1+1];
41  SB[x1]   =  SA[x0] - SA[x1];
42  SB[x1+1] =  SA[x0+1] - SA[x1+1];
43
44  SA[x0]   =  SB[x0];
45  SA[x0+1] =  SB[x0+1];
46  SA[x1]   =  SB[x1]*SROT[2*r2] + SB[x1+1]*SROT[2*r2+1];
47  SA[x1+1] =  -SB[x1]*SROT[2*r2+1] + SB[x1+1]*SROT[2*r2];
48
49  SB[x2]   =  SA[x2] + SA[x3];
50  SB[x2+1] =  SA[x2+1] + SA[x3+1];
51  SB[x3]   =  SA[x2] - SA[x3];
52  SB[x3+1] =  SA[x2+1] - SA[x3+1];
```

```
53
54  SA[x2]   =  SB[x2];
55  SA[x2+1] =  SB[x2+1];
56  SA[x3]   =  SB[x3]*SROT[2*r3] + SB[x3+1]*SROT[2*r3+1];
57  SA[x3+1] =  -SB[x3]*SROT[2*r3+1] + SB[x3+1]*SROT[2*r3];
58 }
```

A.6 4-word group using two threads

Listing A.5 used two threads for each 4-word group, so the total number of threads used for the implementation of the N-point radix-2 DIF FFT is equal to $N/2$. Each thread processing the operations of the two 2-word groups (see lines from 18 to 26 and from 31 to 39). Each 2-word group is processed by one thread similar to Listing A.1.

Listing A.5. N-point radix-2 DIF FFT using 4-word group

```
1  short j = threadIdx.x
2  short i = j/2;
3  short k = j%2;
4  for(short s=1; s<=n; s++)
5  {
6
7   short p = M/(1<<(2*s));
8   short x = 2*(i+(i/(1<<2*(n-s)))*(1<<2*(n-s))*3);
9   short r = (i%(1<<2*(n-s)))*(1<<2*(s-1));
10
11  x0 = x + k*p;
12  x1 = x0+(p<<1);
13  x2 = x0 + k*p;
14  r0 = r + k*(N>>2);
15  r1 = r<<1;
16
17  //step 1:
18  SB[x0]   =  SA[x0] + SA[x1];
19  SB[x0+1] =  SA[x0+1] + SA[x1+1];
20  SB[x1]   =  SA[x0] - SA[x1];
21  SB[x1+1] =  SA[x0+1] - SA[x1+1];
22
23  SA[x0]   =  SB[x0];
24  SA[x0+1] =  SB[x0+1];
25  SA[x1]   =  SB[x1]*SROT[2*r0] + SB[x1+1]*SROT[2*r0+1];
26  SA[x1+1] =  -SB[x1]*SROT[2*r0+1] + SB[x1+1]*SROT[2*r0];
27
28  __syncthreads();
29  //step 2:
30
31  SB[x2]   =  SA[x2] + SA[x2+p];
32  SB[x2+1] =  SA[x2+1] + SA[x2+p+1];
33  SB[x2+p] =  SA[x2] - SA[x2+p];
34  SB[x2+p+1] =  SA[x2+1] - SA[x2+p+1];
35
36  SA[x2]   =  SB[x2];
37  SA[x2+1] =  SB[x2+1];
38  SA[x2+p] = SB[x2+p]*SROT[2*r1]+SB[x2+p+1]*SROT[2*r1+1];
39  SA[x2+p+1]=-SB[x2+p]*SROT[2*r1+1]+SB[x2+p+1]*SROT[2*r1];
40
41 }
```

A.7 4-word group using four threads

Listing A.6 used four threads for each 4-word group, so the total number of threads used for the implementation of the N-point radix-2 DIF FFT is equal to N. Each set of threads processes the operations of the two 2-word groups (see lines from 19 to 25, from 32 to 38). Each 2-word group is processed by a set of threads similar to Listing A.2.

Listing A.6. 4-word group using four threads

```
 short j = threadIdx.x
short i = j/2;
short k = j%2;
short h = i/2;
short l = j%4;
short m = i%2;

for(short s=1; s<=n; s++)
{
 p = M/(1<<(2*s));
 x = 2*(h+(h/(1<<2*(n-s)))*(1<<2*(n-s))*3);
 r_tmp = (h%(1<<2*(n-s)))*(1<<2*(s-1));
 x0 = x+l*p;

 //step 1:
 sign1= m*(-2)+1;
 r0 = r_tmp + k*(N/4);
 x1 = x0+sign1*2*p;
 SB[x0] = SA[x1] + sign1*SA[x0];
 SB[x0+1] = SA[x1+1] + sign1*SA[x0+1];
 short inx1 = x0 - m*2*p + m;
 short inx2 = x0 + (!m)*2*p ;
 SA[inx1]   = SB[inx1];
 SA[inx2+m] = sign1*SB[inx2]*SROT[2*r0+m] +
                          SB[inx2+1]*SROT[2*r0+(!m)];
 __syncthreads();

 //step 2:
 sign2 = k*(-2)+1;
 r1 = r_tmp*2;
 x2 = x0+sign2*p;
 SB[x0] = SA[x2] + sign2*SA[x0];
 SB[x0+1] = SA[x2+1] + sign2*SA[x0+1];
 short inx3 = x0 - k*p + k;
 short inx4 = x0 + (!k)*p ;
 SA[inx3] = SB[inx3];
 SA[inx4+k] = sign2*SB[inx4]*SROT[2*r1+k] +
                          SB[inx4+1]*SROT[2*r1+(!k)];
 __syncthreads();
}
```

A.8 4-word group using eight threads

In Listing A.7, used eight threads for each 4-word group. So, the total number of threads used for the implementation of the N-point radix-2 DIF FFT is equal to $2 \cdot N$. Each set of threads processes the operations of the two 2-word groups (see lines from 23 to 25 and from 31 to 33). Each 2-word group is processed by a set of threads similar to Listing A.3.

Listing A.7. 4-word group using eight threads

```
short j = threadIdx.x;
short i = j/2;
short k = j%2;
short l = i%4;
short h = i/2;
short m = i%2;
short g = h/2;
short u = h%2;

for(short s=1; s<=n; s++)
{
short q = (1<<2*(n-s));
short p = M/(1<<(2*s));
short x = 2*(g+(g/q)*q*3);
short r_tmp = (g%q)*(1<<2*(s-1));
x0 = x + l*p;
short sign0 = k*(-2)+1;
//step 1:
sign1= u*(-2)+1;
r0 = r_tmp + m*(N/4);
x1 = x0+sign1*2*p;
short inx0 = x0 + (!u)*2*p;
SB[x0+k] = SA[x1+k] + sign1*SA[x0+k];
SC[x0+k] = SB[inx0+k]*SROT[2*r0+u];
```

```
25 SA[x0+k] = (!u)*SB[x0+k] + u*(SC[x1+k]+sign0*SC[x0+!k]);
26 //step 2:
27 sign2= m*(-2)+1;
28 r1 = 2*r_tmp;
29 x2 = x0+sign2*p;
30 short inx1 = x0 + (!m)*p;
31 SB[x0+k] = SA[x2+k] + sign2*SA[x0+k];
32 SC[x0+k] = SB[inx1+k]*SROT[2*r1+m];
33 SA[x0+k] = (!m)*SB[x0+k] + m*(SC[x2+k]+sign0*SC[x0+!k]);
34 }
```

Bibliography

[1] S. Bertram, "On the Derivation of the Fast Fourier Transform," *Audio and Electroacoustics, IEEE Transactions on*, vol. 18, Mar. 1970,.

[2] D. B. Kirk and W. W. Hwu, "Programming massively parallel processors:a hands-on approach," Morgan kaufmann publications, ISBN: 978-0-12-381472-2, 2010.

[3] NVIDIA, "GPU Gems2," Chapter 30, ISBN 0-321-33559-7, http://http.developer.nvidia.com/GPUGems2/gpugems2_chapter30.html, 2005.

[4] NVIDIA, "NVIDIA GeForce 8800 Architecture Technical Brief," Nov. 2006.

[5] http://www.tomshardware.com/reviews/graphics-beginners-3,1297.html.

[6] NVIDIA, "Whitepaper NVIDIA's Next Generation CUDA Compute Architecture:Fermi," 2009.

[7] Nvidia, "NVIDIA CUDA C Programming Guide," version 4.2, http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_C_Programming_Guide.pdf, 2012.

[8] Nvidia, "CUDA C Best Practices Guide," version 4.1, http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_C_Best_Practices_Guide.pdf, 2012.

[9] L. Rabiner and B. Gold, "Theory and Application of Digital Signal Processing," Prentice-Hall, pp. 356-435, pp. 573-626, 1975.

[10] A. V. Oppenheim and R. W. Schafer, "Discrete-time signal processing", Prentice-Hall, Second edition, pp. 629-669, 1999.

[11] E. C. Ifeachor and B. W. Jervis, "Digital Signal Processing", Addison-Wesley, pp. 47-102, 1993.

[12] E. O. Brigham, "The Fast Fourier Transform," Prentice-Hall, 1974.

[13] B. M. Baas, "A Low-Power, High-Performance, 1024-Point FFT Processor," *IEEE Journal of solid-state circuits*, vol. 34, pp. 380-387, Mar. 1999.

[14] M. Garrido, J. Grajal, M. A. Sanchez, and O. Gustafsson, "Pipelined Radix-2^k Feedforward FFT Architectures," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, pp. 1-10, 2011.

[15] M. Garrido, "Efficient hardware architectures for the computation of the FFT and other related signal processing algorithms in real time," Ph.D. dissertation, Universidad Politécnica de Madrid, Madrid, Spain, 2009.

[16] http://en.wikipedia.org/wiki/Amdahl%27s_law.

[17] Nvidia, "CUDA Toolkit 4.2 CUFFT Library Programming Guide," http://developer.download.nvidia.com/compute/DevZone/docs/html/ CUDALibraries/doc/CUFFT_Library.pdf, Mar. 2012.

[18] D. Brandon Lloyd, C. Boyd, and N. Govindaraju, "Fast computation of general Fourier Transforms on GPUs," *Multimedia and Expo, 2008 IEEE International Conference on*, pp. 5-8, Apr. 2008.

[19] J. W. Cooley and J. W. Tukey, "An algorithm for the machine calculation of complex Fourier series," *Math. Computation.*, vol. 19, pp. 297-301, Apr. 1965.

[20] B. Duan, W. Wang, X. Li, C. Zhang, P. Zhang, and N. Sun, "Floating-point mixed-radix FFT core generation for FPGA and comparison with GPU and CPU," *Field-Programmable Technology (FPT), 2011 International Conference on*, Dec. 2011.

[21] S. Qi, X. Wang, and S. Shi, "Mixed Precision Method for GPU-based FFT," *Computational Science and Engineering (CSE), 2011 IEEE 14th International Conference on*, pp.580-586, Aug. 2011.

[22] X. Li, Y. Gao, and Y. Liu, "Performance Evaluation of Fast Fourier Transform Application on Heterogeneous Platforms," *Cyber-Enabled Distributed Computing and Knowledge Discovery (CyberC), 2011 International Conference on*, pp.241-249, Oct. 2011.

[23] D. Brandon Lloyd, C. Boyd, and N. Govindaraju, "Fast computation of general Fourier Transforms on GPUS," *Multimedia and Expo, 2008 IEEE International Conference on*, pp.5-8, 2008.

[24] N. Govindaraju, B. Lloyd, Y. Dotsenko, B. Smith, and J. Manferdelli, "High performance discrete Fourier transforms on graphics processors," *High Performance Computing, Networking, Storage and Analysis, 2008. SC 2008. International Conference for*, pp.1-12, Nov. 2008.

[25] Z. Lili, S. Zhang, M. Zhang, and Z. Yi, "Streaming FFT Asynchronously on Graphics Processor Units," *Information Technology and Applications (IFITA), 2010 International Forum on*, vol. 1, pp.308-312, Jul. 2010.

[26] A. Brodtkorb, C. Dyken, T. Hagen, J. Hjelmervik, and O. Storaasli, "State-of-the-Art in heterogeneous computing," *Scientific Programming*, 18(1), pp. 1-33, 2010.

[27] B. Cope, P. Y. K. Cheung, W. Luk, and L. Howes, "Performance comparison of graphics processors to reconfigurable logic: A case study," *IEEE Trans Computers*, 59(4), pp. 433-448, 2010.

[28] Khronos OpenCL Working Group, "The OpenCL specification 1.0," http://www.khronos.org/registry/cl/, 2008.

[29] J. Nickolls and W. J. Dally, "The GPU computing era," *IEEE Micro*, 30(2), pp. 56-69, 2010.

[30] K. H. Tsoi and W. Luk, "Axel: a heterogeneous cluster with FPGAs and GPUs," *ACM/SIGDA Int. Symp. FPGA*, pp. 115-124, 2010.

[31] Altera Corporation, "Implementing FPGA Design with the OpenCL Standard,", pp. 1-9, Nov, 2012. http://www.altera.com/literature/wp/wp-01173-opencl.pdf