2013 IEEE 11th International Conference on Dependable, Autonomic and Secure Computing

Design and Implementation of Parallel FFT on CUDA

Xueqin Zhang
School of Electronic Engineering
University of Electronic Science and Technology of China
Chengdu, P.R.C
ishadowyeah@hotmail.com

Kai Shen
School of Electronic Engineering
University of Electronic Science and Technology of China
Chengdu, P.R.C
skkyle@hotmail.com

Chengguang Xu
School of Electronic Engineering
University of Electronic Science and Technology of China
Chengdu, P.R.C
xuxiaolan1991@gmail.com

Kaifang Wang
School of Communication and Information Engineering
University of Electronic Science and Technology of China
Chengdu, P.R.C
sk596322762@live.com

Abstract

Fast Fourier Transform (FFT) algorithm has an important role in the image processing and scientific computing, and it's a highly parallel divide-and-conquer algorithm. In this paper, we exploited the Compute Unified Device Architecture（CUDA）technology and contemporary graphics processing units *(GPUs) to achieve higher performance. We focused on two aspects to optimize the ordinary FFT algorithm, multi-threaded parallelism and memory hierarchy. We also proposed parallelism optimization strategies when the data volume occurs and predicted the possible situation when the amount of data increased further. It can be seen from the results that Parallel FFT algorithm is more efficient than the ordinary FFT algorithm.*

Key words

memory hierarchy, thread/thread block, CUDA technology

1.Introduction

The Fast Fourier Transform (FFT) is a well-known and widely used tool in many scientific and engineering fields. With the characteristics of intensive computation and being highly divide-and-conquer, FFT is proper to make parallel improvement.

With the release of the NVIDIA Compute Unified Device Architecture (CUDA), hundreds of coding applications on CUDA have succeeded in academic and industrial fields. Most of them are several times faster than commodity central processing units (CPUs). With the release of new tools, such as MCUDA[Stratton,2008][1], the CUDA parallel threads can also make high performance on multi-core CPUs, despite the slower running speed than graphics processing units (GPUs) for the low-quality floating-point operating units. Recent advances in GPU in double-precise floating-point computation fields make GPU applied in broader fields [8]. The algorithm in this paper employs the GPU accelerating computation to achieve the accelerated and optimized FFT algorithm. The contributions of the paper are as follows.

- A parallel FFT algorithm that is achieved from two aspects, multi-threaded parallelism and memory hierarchy.
- A combined algorithm that utilizes multi-core CPUs and the GPU to speed up the FFT algorithm when data volume occurs.

Our experimental results showed that we are able to achieve speedup above 50 in terms of the data volume in comparison with the ordinary FFT algorithm.

The rest of the paper is organized as follows. Sec. 2 provides background information on CUDA hardware architecture and memory hierarchy. Sec.3 presents introduction and parallelism analysis of FFT algorithm and optimization strategies. Experimental results and analysis are presented in Sec. 4 and we conclude the paper in Sec. 5.

978-1-4799-3381-5/13 $31.00 © 2013 IEEE
DOI 10.1109/DASC.2013.130

2.CUDA Computation Mode

2.1 Hardware Architecture[5]

CUDA's hierarchy of threads maps to a hierarchy of processors on the GPU; a GPU executes one or more kernel grids; a streaming multiprocessor (SM) executes one or more thread blocks and CUDA cores and other execution units in the SM execute threads. The SM executes threads in groups of 32 threads called a warp. While programmers can generally ignore warp execution for functional correctness and think of programming one thread, they can greatly improve performance by having threads in a warp execute the same code path and access memory in nearby addresses.

As Fig.1 shows, Streaming Processor (SP) is the basic operation unit in NVIDIA GPU. And there are many SP to compute simultaneously in one NVIDIA GPU. SM is comprised of numerous SPs（generally 8）together with some other units. A few SMs make up Texture Processing Clusters(TPC)

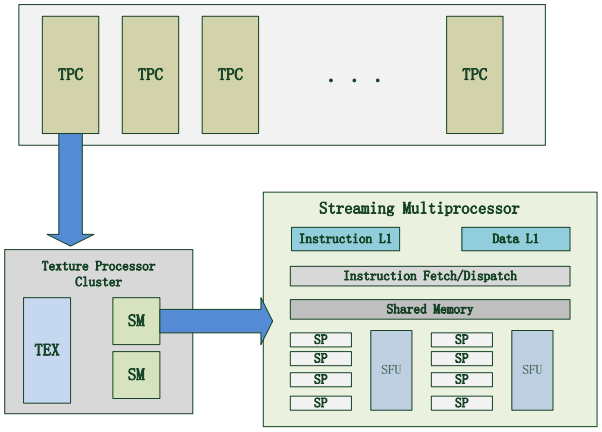

Figure1.GPU computation mode

In CUDA, there is no TPC, just adjusting according to the number of SMs and SPs. Thread blocks are distributed to SMs to compute in CUDA programming. And threads consisting of warps are divided into groups to compute.

2.2 Memory Hierarchy[1]

As shown in Fig.2, CUDA threads can access data in several memories simultaneously. A thread block is a set of concurrently executing threads that can cooperate among themselves through barrier synchronization and shared memory. A thread block has a block ID within its grid.

A grid is an array of thread blocks that execute the same kernel, read inputs from global memory, write results to global memory, and synchronize between dependent kernel calls. In the CUDA parallel programming model, each thread has a per-thread private memory space used for register spills, function calls, and C automatic array variables. Each thread block has a per-Block shared memory space used for inter-thread communication, data sharing, and result sharing in parallel algorithms. Grids of thread blocks share results in Global Memory space after kernel-wide global synchronization.

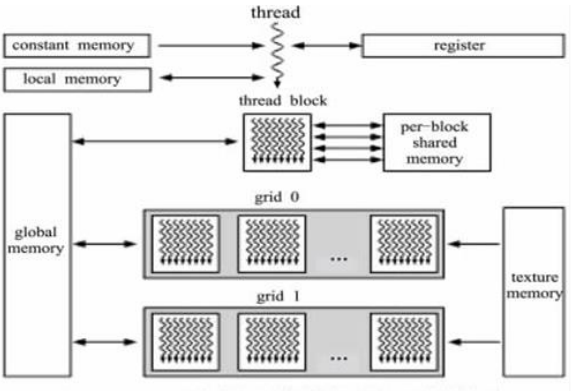

Figure2.grid and shared memory in CUDA programming architecture

3.FFT Algorithm Based on CUDA and Improvement

3.1 Introduction of FFT Algorithm [3][7]

One-dimension N-point Discrete Fourier Transform (DFT) formula is

$$X[k]=\sum_{n=0}^{N-1}x[n]e^{-j\frac{2\pi}{N}nk}\text{ , k=0,1,…N-1.} \qquad (1)$$

Suppose $W_N=e^{-j\frac{2\pi}{N}}$, namely twiddle factor. Based on Decimation-in-time FFT algorithm, we can consider

computing X[k] by separating x[n] into two (N/2)-point sequences consisting of the even-numbered points in x[n] and the odd-numbered points in x[n].Thus let

$$\begin{aligned} x[2r] &= x_1[r] \\ x[2r+1] &= x_2[r] \end{aligned} \quad , \text{r=0,1,…,N/2-1.} \qquad (2)$$

Using reducibility of the twiddle factor, namely

$W_N^{2rk} = W_{N/2}^{rk}$, yields

$$X_1[k] = \sum_{r=0}^{N/2-1} x_1[r] W_{N/2}^{rk}, X_2[k] = \sum_{r=0}^{N/2-1} x_2[r] W_{N/2}^{rk} \qquad (3)$$

Then the forward equation (1) can be written as,

$$X[k] = X_1[k] + W_N^k X_2[k] \quad \text{(k=0,1,…,N/2-1)} \qquad (4)$$

Either $X_1[k]$ or $X_2[k]$ is recognized as an (N/2)-point DFT, $X_1[k]$ being the (N/2)-point DFT of the even-numbered points of the original sequence and $X_2[k]$ being the (N/2)-point DFT of the odd-numbered points of the original sequence. After the two DFTs are computed, they are combined according to (4) to yield the N-point DFT X[k]. Figure3 depicts this computation for N=8.

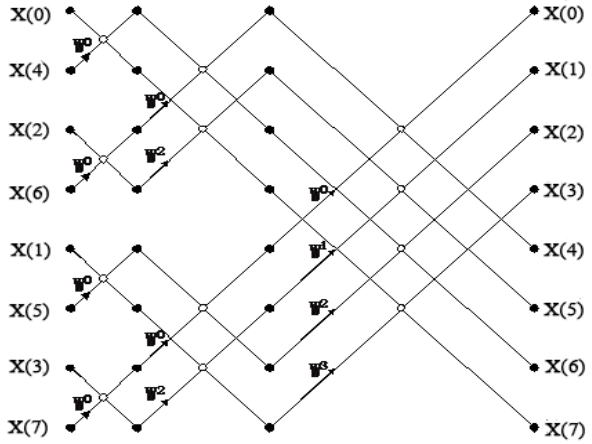

Figure3. 8-point FFT algorithm schematic

3.2 FFT Algorithm Parallelism Analysis [4][7]

As shown in Figure4, the traditional FFT algorithm employed three-tier circulation. For sampling points $N = 2^M$, each stage of the computation takes a set of N complex numbers and transforms them into another set of N complex numbers through basic butterfly computations. This process is repeated M times, resulting in the computation of the desired discrete Fourier transform. As the traditional FFT algorithm is completely serial, we can make some optimizations as follows:

- In one stage, basic butterfly computations are independent and can be operated simultaneously. Moreover, the number of basic butterfly computation in one stage is set, N/2. When implementing the computations in one stage, we can open N/2 threads to operate the corresponding butterfly computations.

- The use of twiddle factors has certain rule, namely the twiddle factors used in the proceeding stage come from ones with even exponent of W in the next stage counting down from W^0. Thus, we can get twiddle factors by method of looking up table instead of repeated computation.

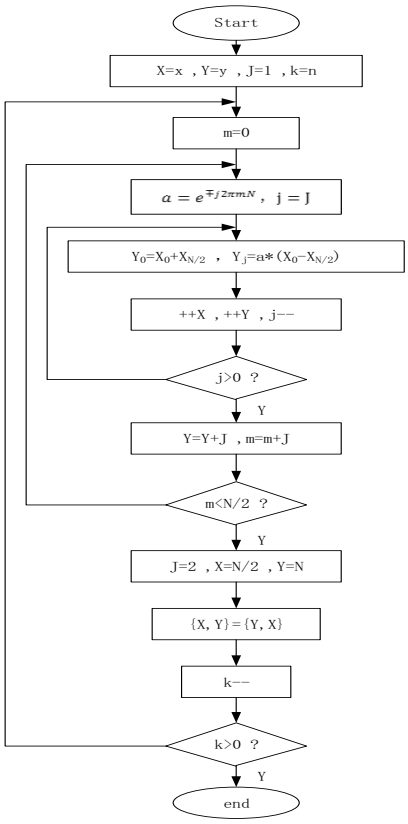

Figure4. FFT algorithm flowchart

3.3 Algorithm Optimization Based on CUDA Technology

3.3.1 Algorithm Parallelism Optimization

According to the characteristics of CUDA computation mode and FFT high parallelism, the Parallelism Optimization strategy is available. Firstly, one basic butterfly computation calls one thread, with the thread numbers adding up to N/2 in one stage. And then with new computation results covering the previous ones, each thread computation in one stage simultaneously goes on with the next stage. The whole process is repeated until the Mth stage. Finally, FFT of the original data is entirely achieved.

3.3.2 Memory Hierarchy Optimization [4]

As 2.2 Memory Hierarchy refers, there are many different types of memories in CUDA technology. Thus we need to make further-step optimization for the memories to advance the efficiency of FFT algorithm. Here are three workable means.

- As Figure3 and Figure4 shows, each stage of FFT algorithm reads and writes all the relevant data, and one basic butterfly computation is to two complex numbers, N/2 ones to N. This causes heavy data access. But luckily, shared memory has exceptional advantage of the communication between high-speed data exchange and thread. Thus, data can be stored in shared memory during algorithm operation and fetched back to the global memory after finishing operation. It is efficient to reduce access and storage latency.

- Texture memory owns such performance advantages as low access and memory latency, addressing computation achieved by efficient hardware units and two-dimensional storage. Thus, in texture memory, the twiddle factor with highly reutilizing rate can be stored in the form of two dimension.

- Thread has its own register and local memory, and the latency of register is far smaller than local memory's. But when large amount of local variable occurs and the number of register is not enough, the system will store the variable to local memory. So the use of local variable should be limited in case of access and storage latency increase.

3.3.3 the Data Volume FFT algorithm parallelism optimization

As the limitations of CUDA hardware architecture exist, such as the limited thread numbers in a thread block and size of registers, the optimization is invalid for data volume (changed by GPUs). We have to take measures as follows.

- When the data volume occurs, we need to use several thread blocks. We can divide the data into groups by bit-reversed order. For example, we use 128 threads per thread block, each block can compute 256 groups of data, and then we can divide the data into N/256 groups. Data number 0-255 uses block 0, number 256-511 uses block 1 and so forth. The data in block can be calculated by one Kernel function and after 8 stages, the data in the other M-8 stages can be done by another Kernel function. The optimized data volume FFT algorithm is shown as Figure5.

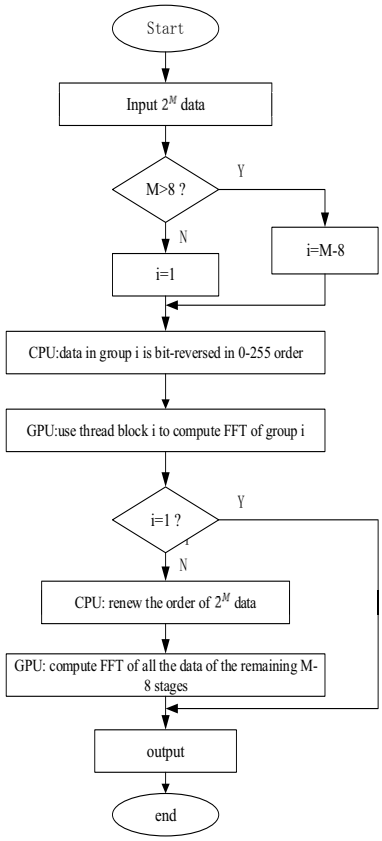

Figure5. The optimized data volume FFT algorithm flowchart

- Because the computation in the last few stages operates across threads, the shared memory is invalid. And then all the data can only be stored on the global memory to support the data volume with sacrifice of performance of this part.

4. The Test and Analysis of algorithm performance

4.1 The Test of algorithm environment

To verify the parallel FFT algorithm performance on CUDA , we use the experimental environment as shown in Table.1 to test the algorithm.

Table1 The experimental environment test

Operating system	Windows 7 Ultimate 32bit (DirectX 11)
CPU	Intel Pentium Dual Core E2200 @ 2.20GHz
Memory	2G
Graphics	Nvidia GeForce GTS 450 (256 MB / Nvidia)
Multiprocessors	12(96 Cores)
Threads Per Multiprocess	768
Threads Dimensions	512*512*64
Grid Dimensions	65535*65535*1
Share Memory Per Block	16384 byte

4.2 The results and analysis of optimized algorithm

In order to observe the improved performance of FFT algorithm better based on CUDA technology, we tested the same data using three different kinds of FFT algorithm. Parallel_FFT is optimized by CUDA technology (thread blocks and the number of threads assigned automatically based on the hardware conditions), GPU_FFT is a traditional single-threaded calculating GPU FFT algorithm. Taking into account the windows platform and GPU hardware limitations, the measured time includes only the FFT core code running time, it does not contain such as the transfer time of the host to the graphics card and function call time. The test results are shown in Table.2:

Table 2 Comparison of three kinds of FFT algorithm running time （ms）

Numbers	1	2	3	4	5	6	7	8	9
Data	128	256	512	1024	2048	4096	8192	16384	32768
Parallel_FFT	0.251	0.293	0.342	0.448	0.529	0.641	0.661	0.779	0.792
GPU_FFT	12.22	27.9	62.3	140.23	307.06	674.87	1457.2	3208	7352
CPU_FFT	<1	<1	<1	<1	4	7	16	31	62

※Due to the operating system limitation, the highest precision timekeeping function is 1 ms, in CPU_FFT 1 to 4 groups of test time, if less than 1 ms, it is displayed as <1.

Transferring table2 as Fig.6 ,

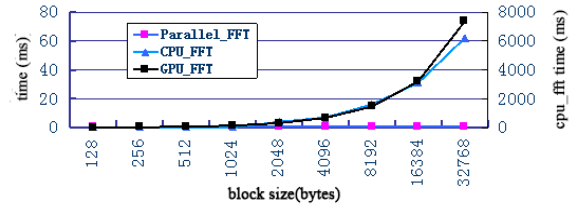

Figure6. Three FFT algorithm performance comparison

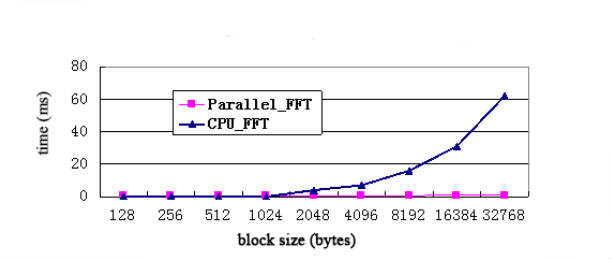

Figure7. FFT algorithm-consuming comparison between GPU parallel and the CPU serial

From the analysis of Fig.6 and Fig.7 can get:

- When the data are small (in this example, the data groups are less than 8192), CPU working time and GPU parallel computation time consume less than 1 ms, performance is not obvious, but when the data groups increase, the performance is more obvious when it reaches more than thirty thousand groups of data, performance has reached more than 50 times.

- Single-threaded GPU computing ability is much weaker than CPU's, but with the use of parallel computing technology, GPU computing ability is obviously greater than CPU.

- But when the data groups gradually increase, the working time of using traditional FFT CPU increases exponentially. While using CUDA technology it is stable and low.

4.3 The analysis of FFT algorithm parallelism degree and time-consuming

Make appropriate changes to the algorithm, the optimal allocation according to hardware conditions thread blocks and the number of threads allocated to artificial threads, respectively, we get Table 3 with 512 groups of data obtained FFT operation :

Table3 CUDA technology-based single-threaded blocks of different threads FFT calculation time

Numbers (n)	1	2	3	4	5	6	7	8	9
Threads allocated nums(p)	1	2	4	8	16	32	64	128	256
Time consuming T_p （ms）	67.4	34.1	17.2	8.7	4.4	2.3	1.2	0.6	0.4
Speed up Ratio S_p	1	1.98	3.92	7.75	15.32	29.3	56.2	112.3	168.5

※Speed up Ratio: The ratio between time-consuming of single-threaded algorithm and when p is the number of threads : $S_p = T_1 / T_p$.

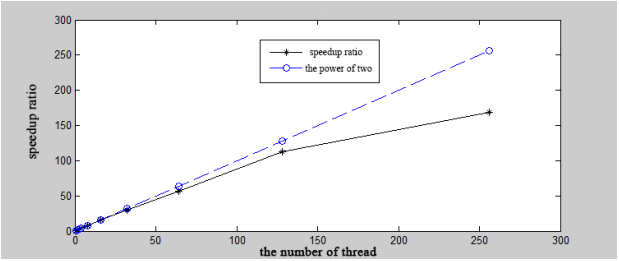

Figure8.The FFT algorithm speedup ratio with the improvement of parallelism

Analysis of Table 3 and Fig.8 we can see:

- With the exponential increase the number of the calling thread, the curve of theoretical speed up ratio should be completely overlap with the curve of a power of 2, but you can see the basic speed up ratio curve is under the curve in the power of 2, that does not increase exponentially accelerating performance, suggesting that time-consuming and does not increase proportionally with the degree of parallelism speed, but will play some discount;

- The speed up ratio growth rate increasing with the number of threads increasing, and with the further improvement of the degree of parallelism. Speed up ratio curves deviate gradually with the power of 2 curves. It is clearly that when we using 256 processes to calculating 512 groups of FFT, the actual speed up ratio is 169, with a difference of nearly 0.5 time with the theoretical value, indicating that although the overall time of when computing parallel FFT is increasing with the increased degree of parallelism, but the acceleration effect is much less obvious. It can be speculated that when the amount of data is large enough, when the degree of parallelism reaches a certain point, the memory and other aspects of the additional consumption will offset the degree of parallelism to improve time savings.

5. Conclusions

In this paper, based on CUDA technology, improved and optimized of the parallel FFT algorithm, through the parallel improvement, memory improvement and the large amount of data to improve and optimize specific aspects of the traditional serial FFT algorithm. From the results above can also be seen using CUDA technology to the traditional FFT algorithm boost hugely in performance, except of time-consuming to copy data device and the host, speaking purely from computing power, GPU-intensive data in large-scale parallel computing has unique advantages. Using the computing power is only 1.1 of the low-end graphics cards GTS450 in this paper has such a performance, showing great prospects CUDA technology. However, due to the parallel thread blocks, threads, and memory size are very different with the hardware limitations, it cannot blindly expand the size of the data, it is not as well as traditional serial FFT algorithm. Large-scale data can be considered for a particular CPU-GPU collaborative computing can be parallel to sub-blocks using the GPU to calculate the periodic results, and then the rest can easily make the CPU serial to complete. Considering the important role of FFT algorithm in engineering applications and scientific computing, this improvement will be meaningful.

6. Reference

[1] Chou， DY., GPGPU Programming Techniques—From GLSL,CUDA to OpenCl , Machinery Industry Press, 2011

[2] David B.Kirk Wen-mei W.Hwu, Actual programming massively parallel processors, Tsinghua University Press, 2010

[3] Sanjit K .Mitra, Digtal Signal Processing A Computer-Based Approach,Third Edition, Electronic Industry Press, 2010

[4]Zhao,LL. and Zhang,SB. and Zhang,M. and Yao,T., CUDA-based high-speed FFT calculation., Application Research of Computers, 2011, 1556-1559

[5]GXW., CUDA of Threading: Block and Grid settings, 2011

[6] Yuan,Q. and Guo,ZQ. and Zhan,M. and Yao.Q and Liu,CX., the parallel FFT algorithm, Scientific Technology and Engineering, 2008, 4709-4714.

[7]Oppenheim,A.V. Schafer,R.W. Buck,J.R., Discrete-time Signal Processing, Second Edition, Tsinghua University Press, 2005.1

[8] Koji Yasuda,Accelerating Density Functional Calculations with Graphics Processing Unit,Journal of Chemical Theory and Computation,2008,464-8601