Parallel Algorithm of IDCT with GPUs and CUDA for Large-scale Video Quality of 3G

Qingkui Chen[12] Haifeng Wang[2,3] Songlin Zhuang[1] Bocheng Liu [2]

1.School of Optical-Electrical and Computer Engineering, University of Shanghai for Science and Technology, Shanghai, China
2.University of ShangHai for Science and Technology
Shanghai, School of Management, Shanghai, China
3.Lin Yi University, Linyi, China

***Abstract*—When video is transmitted over 3G networks, the video quality might suffer from impairments caused by packet losses. Extracting video quality features is a set of algorithms and inverse discrete cosine transforms is an important algorithm in this field. To improve the performance and be suitable to apply to evaluating the 3G video quality in real-time, two different parallel algorithms with CUDA of the inverse discrete cosine transform are proposed. The parallel algorithms are exploited combined the high parallelism of GPUs with a high bandwidth in memory transfer at low cost. Moreover, the device memory accessing model is analyzed and considered the optimal use of the data cache. The experimental results show that the uberkernel algorithm with shared memory and texture memory outperforms other schemes for medium-sized tasks and the perisistent algorithm is better for large-scale tasks.**

***Index Terms*— Inverse Discrete Cosine Transform; CUDA; GPU; Persistent Thread Model; Uberkernel Thread Model**

I. INTRODUCTION

The third generation mobile communication system (3G) is mainly for high speed services rather than the traditional service of voice and low speed data. 3G has the greater capacity and more flexible transmission of high speed and offers up to 2Mbps for high-quality multimedia services[1]. Mobile video broadcasting service or mobile TV is becoming a popular application for 3G network [2]. Because watching a movie or a live TV show on their cell phones is an attractive application to many users at anytime and at any place. The H.264 standard is a newly developed video coding standard and can provide high compression performance as well as relatively robust transmission performance in error-prone wireless environments[3]. Thus it is an attractive candidate for 3G video applications. Here we consider the H.264 video over 3G wireless IP networks. The H.264 coded bitstream is packetized and the encoded bitstream is then transmitted as RTP/UDP/IP packets. Since the packet losses over wireless networks will most likely occur in bursts[2,4]. The video quality received at the receiver side might suffer from impairments caused by packet losses. Therefore, the 3G content providers pay more attentions to the quality-of-service (QOS) in order to satisfy the consumers. The video quality evaluation at the end-user terminal becomes hot research topic. The existing video quality assessment algorithms should extract the video quality features from different levels. Such as bitstreams, frames, slices, macroblocks and pixels[5,6]. Extracting quality features is an important process in the video quality assessment and has highly computational complexity. Thus the existing extracting quality features algorithm is difficult to handle the 3G video in the real-time processing due to the high computational complexity.

To solve this problem, the parallel processing algorithms for extracting video quality features should be exploited. When the video bitstreams are decoded to video files, there is an important step that is responsible for transforming the video information in frequency domain into spatial domain after the entropy decoding [3]. This specific process is inverse discrete cosine transforms (IDCT) that is intensive computation task. And the specific computational task is well suitable for Graphics Process Unit (GPU). In this work, we focus the issue how to handle the IDCT for a large-scale H.264 video bitstreams from the core network in 3G systems. The data-parallel algorithms on GPUs are designed and the performance of these algorithms are analyzed.

In this paper we study the parallel algorithms of IDCT for large scale video streams from the core network of 3G. Our motivation is to improve the performance of IDCT and allow the parallel algorithms apply to the practical projects. Two different parallel algorithms are proposed and each algorithm has three kinds of accessing mechanisms which is a key factor to the access performance of the device memory. The remainder of this paper is organized as follows. Section II gives a brief description of general computing of GPU and CUDA. Then discuss the H.264 standard. Section III describes the computational task and the two parallel algorithms. In section IV, Comparing the two parallel algorithms with three accessing mechanisms and the experimental results

doi:10.4304/jcp.7.8.1880-1886

are shown and discussed. The paper is concluded in section V.

II. PRELIMINARIES

A.GPUs and CUDA

The new generation of GPUs provide flexible programmability and computational power that exceed previous generations. GPUs is capable of providing a high computational throughput due to their large number of processor cores. GPUs consist of many multiprocessors(MP) which contains eight Scalar Processors (SPs) and additional memory. The GPUs use the SIMT(Single Intruction Multiple Thread) parallel programing paradigm. The SPs within a single MP must execute the same instruction synchronously while the MPs in the GPU can execute instructions independently of each other.

The compute unified device architecture(CUDA) is provided by the NVIDIA and is a dedicated data-paralle programming language. This vendor-specific framework allows user to directly manage the hierarchy of memories and implement their GPU programs without knowledge of computer graphics[7]. Here we give a brief summary of CUDA-specific terms used in the rest of this article. In general, CUDA-based applications consist of host code and device code, each running on the CPU and on the GPU respectively. The host code invokes the device code that implements kernels of stream applications. On the other hand, input or output data must be placed in video memory, called device memory. Since device memory is separated from host memory. Data has to be transferred between them. This implies a high memory transfer overhead.

These threads in CUDA are organized as a large grid of thread blocks. Each thread block is a 1D,2D or 3 dimension structure that can contain up to 512 threads. The threads are executed by assigning thread blocks to MPs. Each thread has a unique id that it can work out from it's thread_idx and block_idx. The block_idx identifies which block the thread belongs to and the thread_idx defines the thread's position within the thread block.

B. DCT&IDCT

H.264/AVC is well known video coding standard established by the Joint Video Team(JVT). The H.264 has many innovations compared to the earlier standards. Here we focus on the integer discrete cosine transforms(DCT). The H.264 /AVC standard use two different integer transforms which include 2D DCT and Hadamard[8]. The two dimensional DCT(2D DCT) is used to transform the information from the space domain to the frequency domain in image and video compression fields. When the information is transformed in frequency domain,the information used to represent the image or video that is less important to the human visual system may be discarded. This operation improves the compression rates with little impact to the image quality.

The 2D DCT can be represented by equation(1)

$$\mathrm{Y}=(\mathrm{C}_f\ \mathrm{X}\ \mathrm{C}_f^{\mathrm{T}})\otimes \mathrm{E}_f \quad (1)$$

$\otimes$ is indicator of multiplication.Where C_f is given by

$$C_f=\begin{bmatrix}1 & 1 & 1 & 1\\ 2 & 1 & -1 & -2\\ 1 & -1 & -1 & 1\\ 1 & -2 & 2 & -1\end{bmatrix} \quad (2)$$

and E_f is given by

$$E_f=\begin{bmatrix}a^2 & \frac{ab}{2} & a^2 & \frac{ab}{2}\\ \frac{ab}{2} & \frac{b^2}{4} & \frac{ab}{2} & \frac{b^2}{4}\\ a^2 & \frac{ab}{2} & a^2 & \frac{ab}{2}\\ ab & b^2 & ab & b^2\end{bmatrix} \quad (3)$$

Where,

$$a=1/2, b=\sqrt{1/2}\cos\left(\frac{\pi}{8}\right), c=\sqrt{1/2}\cos\left(\frac{3\pi}{8}\right), d=c/b.$$

Because multiplication by E_f is not a standard matrix multiplication but a one where only same matrix locations are multiplied. This special multiplication can be incorrporated into the quantizer in H.264 /AVC standard. Therefore the 2D DCT only consists of C_f and C_f^T. As shown in Equ.(1), it can be implemented with additions and subtractions on integer values. Due to without multiplication the cost of memory access and the computational complexity are reduced. The simplifing equation is used in JM that is a famous H.264 coder software and shown in Equ. (4)

$$Y=\begin{bmatrix}1 & 1 & 1 & 1\\ 2 & 1 & -1 & -2\\ 1 & -1 & -1 & 1\\ 1 & -2 & 2 & -1\end{bmatrix}\left[X\right]\begin{bmatrix}1 & 2 & 1 & 1\\ 1 & 1 & -1 & -2\\ 1 & -1 & -1 & 2\\ 1 & -2 & 1 & -1\end{bmatrix} \quad (4)$$

The inverse 2D DCT is inverse operation of forward DCT and has the similar principle. As shown in Equ.(5)

$$\mathrm{X}=\mathrm{C}_i^{\mathrm{T}}(\mathrm{Y}\otimes \mathrm{E}_i)\mathrm{C}_i \quad (5)$$

Where C_i is given

$$C_i=\begin{bmatrix}1 & 1 & 1 & 1\\ 1 & 1/2 & -1/2 & -1\\ 1 & -1 & -1 & 1\\ 1/2 & -1 & 1 & -1/2\end{bmatrix} \quad (6)$$

and E_f is given

$$E_f = \begin{bmatrix} a^2 & ab & a^2 & ab \\ ab & b^2 & ab & b^2 \\ a^2 & ab & a^2 & ab \\ ab & b^2 & ab & b^2 \end{bmatrix} \quad (7)$$

C. Bufferfly Algorithm

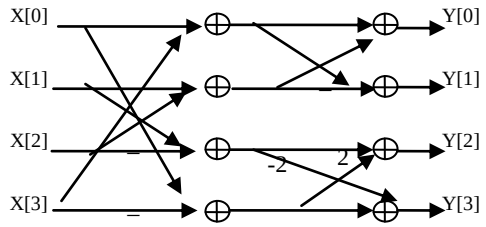

Fig1(a). 1D 4×4 Forward Integer Transform.

In this paper we focus on the optimal integer 2D DCT transformation algorithm that transforms the 2D DCT into two 1D DCT calculation. As shown in Fig.1(a), there is 4×4 2D DCT transform and X is the 4×4 residual data block. Since the 2D DCT is divided into 1D Column-Integer transform and 1D Row-Integer transform, the array X contains four elements of row or column. From the figure the optimal integer 2D DCT transformation algorithm is also called butterfly algorithm that can be implemented just using additions, subtractions and shift operation.

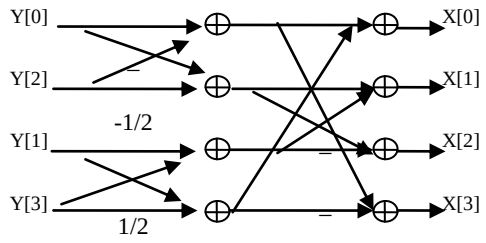

Fig1(b) 1D 4×4 Inversed Integer Transform.

As shown in Fig.1(b), the 1D 4×4 Inversed Integer transform algorithm is presented and the detailed algorithm is as follows:

Algorithm1. Butterfly for IDCT

Function idct_butterfly_device _4×4 (Y,X)

Input parameters: Y is declared as interger array which contains four elements;

Ouput parameters;X is declared as interger array which contains four elements;

Declare integer temp[4] ;

temp[0] = Y [0] + Y [2];

temp[1] = Y [0] - Y [2];

temp[2] = 1/2* Y [1] - Y [3];

temp[3] = Y [1] + 1/2* Y [3];

X[0] = temp[0] + temp[3];

X[1] = temp[1] + temp[2];

X[2] = temp[1] - temp[2];

X[3] = temp[0] - temp[3];

Return X;

4×4 and 8×8 Inversed Integer transform are the core algorithms in our project. The principle of 8×8 Inversed Integer transform is similar to 4×4 Inversed Integer transform. For simplicity, the detailed algorithm is not presented.

III. PROPOSED ALGORITHMS

A. Description of the Tasks

This section explains the computation task. In order to extract the video quality features,the inversed integer transform is important process that can transform the information in frequcency domain into spatial domain. As shown in Fig.(2), the video can be considered as a continuous task stream that has different kinds of subtasks. The subtask is 16×16 integer matrix called as Macroblock in H.264 coding standard. The integer matrix has two different type, one has four 8×8 sub-matrix and the other has sixteen 4×4 sub-matrix. These two different matrix should be handled by 4×4 and 8×8 inversed integer transform algorithms respectively. So the video is denoted as $V=\{M_1, M_2,\ldots, M_n\}$, M represents the macroblock in the video. Let t denote the type of macroblock. Suppose that t = 1 represents the 4×4 macroblock and t = 2 represents the 8×8 macroblock. So the video can be redefined as $V=\{M^t_1, M^t_2,\ldots, M^t_n, t = 1,2\}$ and it is random queue that contains different macroblocks.

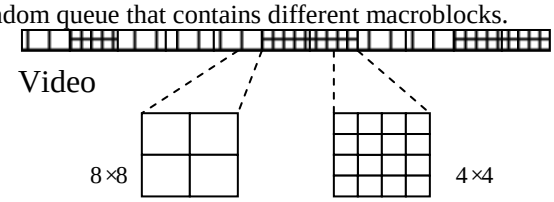

Fig2 Structure of Task

The task is highly intensive computing task that is work well on data-parallel systems. Graphics processing units or GPUs provide a high computational throughput due to their large number of processor cores. This task is very suited to divide many subtasks that can be handled by a large number of threads launched by GPUs. However, because the video is random queue that contains different macroblocks, the task has irregular parallel workloads[9]. Modern GPU programming enviroments have poor support for kernels that consume irregular amounts of work. So it is difficult to deal with this kind of computational task.

B. Persistent Thread Model

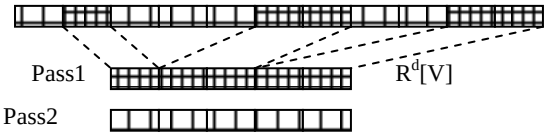

Fig3(a) the Persistent Thread Algorithm.

The persistent thread model is a thread scheduling strategy that launches only enough thread warps to fill the machine but leaves those threads alive through the entire kernel. This thread model is well-suited for efficient production and consumption of irregular parallel works[10]. Here the persistent thread algorithm is adopted based on the persistent thread model. As shown in Fig.3(a).

The algorithm includes two pass operations of inversed integer transform which are implemented by two kernel functions (see function idct_4×4_kernel and idct_8×8_kernel). Each pass has three phases. Gathering data(see function gather) is the first phase that iterates the video queue and collects the same type macroblocks. After this gathering, the array $R^d[V]$ that contains many same type macroblocks should be transformed into the device memory. Then corresponding kernel function is invoked in the second phase. In the final phase the macroblocks that has been operated by the inversed integer transform should be scattered to the original location(see function scatter). GPU launches appropriate threads corresponding to the array $R^d[V]$ size. The persistent thread algorithm pseudocode is presented as follows.

Algorithm2. persistent_thread _Host(V)
declare integer i^{type}
declare M_i integer array for temporary Macroblock
declare integer tmp [V]
declare integer $R^d[V]$ in device memory
//for the first pass executation
call gather(V, temp, 4×4)
call idct_4×4_kernel(tempd[V]);
call scatter(V, temp, 4×4, itype)
//for the second pass executation
call gather(V, temp, 8×8)
call idct_8×8_kernel(tempd[V]);
call scatter(V, temp, 8×8, itype)

Function gather(V,temp,type)
repeat
if $M_i.^{type}$ =type
tmp [V]←M_i
i^{type}←i^{type}+1
until V is end

Function scatter(V,temp,type,i^{type})
repeat
Mi←tmp $[V]^i$
if $M_i.^{type}$ =4×4
V_i←M_i
i^{type}←i^{type}-1
until i^{type}=0

These different kernel functions for inverse 2D DCT transform have the same principle and both include two 1D DCT transform operations for row or column. The inverse 1D DCT transform operation(see algorithm 1) has the same code but operates different data area.

function idct_4×4_kernel(temp [V], $R^d[V]$)
copy $R^d[V]$ ←temp [V]
read data with three different accessing strategies.
Assigned thread according to $R^d[V]$
Do in parallel on the device using $|V|$ threads:
call __device idct_butterfly_device _4×4_horizontal();
call __device idct_butterfly_device _4×4_vertical();
copy temp [V]←$R^d[V]$
function idct_8×8_kernel(temp [V], $R^d[V]$)
copy $R^d[V]$ ←temp [V]
read data with three different accessing strategies.
Assigned thread according to $R^d[V]$
read data with three different accessing strategies.
Do in parallel on the device using $|V|$ threads:
call__device idct_butterfly_device _8×8_horizontal();
call __device idct_butterfly_device _8×8_vertical();
copy temp [V]←$R^d[V]$

The persistent thread algorithm has two advantages. Because the persistent threads are alive throughout the entire kernel and the same type computational subtasks are handeled within the same kernel. The algorithm avoids the overhead of a global synchronization. The size of array $R^d[V]$ dynamiclly change. So the appropriate thread warps are launched correponding the size of array. This approach saves a large number of threads in GPUs and it is well suited for large-scale data parallel tasks. However the overhead of communication between GPUs and CPUs is increased by the gathering and scattering operations.

C.Uberkernel Thread Model

The execution process is divided into two pipeline stages in the persistent thread scheduling stratey and this thread scheduling model maps one pipeline stage to one GPU kernel. The barriers between pipeline stages hinder processor utilization. To eliminate this barriers, multiple pipeline stages can be combined into one kernel. This solution is based on the uberkernel programming model[11]. Uberkernel pack multiple different subtasks into a single physical kernel without the overhead of switching between kernels.

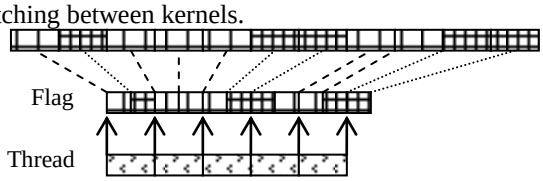

Fig3(b) the Uberkernel Thread Algorithm.

The basic idea of our uberkernel algorithm is that allows a large amount of threads to handle different type subtasks simultaneously. Note here many threads may be idle in this process. The algorithm mainly consists of two steps as follows.

1. Initiating flag array. As shown in Fig.3(b), suppose that the Video has a queue of macroblocks and the size is $|V|$. Firstly a flag array should be generated that contains $|V|$ elements. The flag element represents the type of

corresponding macroblock in the video. Then the flag array will be transferred into the GPU memory.

2. Kernel execution. The algorithm simultaneously checks the status of the flag array by using a large amount of threads. A thread is selected in a thread block and check the corresponding flag element in the flag array. This means that the others threads in thread block will be idle during this time. And then the threads are divided into two group to execute different codes. Finally the results will be gathered and transfer to host memory.

The pseudocode of the Uberkernel thread algorithm is presented as follows.

Algorithm3. Uberkernel (V,flag)
declare integer $flag^d[V]$ in device memory
declare integer $R^d[V]$ in device memory
declare M integer array for temporary Macroblock
declare integer i← thread ID from the CUDA runtime
$R^d[V] \leftarrow V$
$flag^d[V] \leftarrow flag$
Assigned $|M|$ thread according to V
Do in parallel on the device using $|V|$ threads:
If $M^i._{type} = 4 \times 4$
call __device idct_butterfly_ 4×4()
else
call __device idct_butterfly_ 8×8()
$V \leftarrow R^d[V]$

The uberkernels algorithm allows task divergence in execution routine behavior and processors to switch between executing two different stages of the pipeline at once. It eliminates the need for a global barrier between pipeline stages and reduce the overhead of synchronization. Compared by the persistent algorithm, the uberkernel algorithm reduces communication time between GPU and CPU. However, due to without fully utilizing the processor, the uberkernel algorithm is difficult to handle very large-scale data parallel tasks that need more threads.

D. Memory Access Mechanism

Memory stalls is an important factor for the overall performance of data-intensive applications. So memory optimization techniques have also been shown useful for GPU-based algorithms[12]. Current GPUs achieve a high memory bandwidth using a wide memory interface. And there is no automatic memory caching on the GPU, instead a number of optimised memory type set are made available to be explicitly used by the programmer. These memory types are shown in Fig.4

Glocal memory(GM) is the main device memory on GPU. Any input or output of the GPU must be placed in the global memory. It has the slowest access time. In order to improve the access time, a novel technique known as coalescing is proposed in CUDA. When 16 sequential threads access 16 sequential and aligned values in global memory, the 16 access operation will be automatically combined into a single access transaction.

Shared memory(SM) is stored within the MPs and allows the threads in the same block to share information. Shared memory mainly acts as an explicit cache to reduce the number of global memory transactions. As for the cache functionality, shared memory lacks hardware support and is fully managed by the application. It is divided into chucks(16KB each), each shared only amony the threads of a thread block in CUDA.

Texture memory(Tex) is a cached solution for accessing a designated piece of global memory. The texture cache is located on the MPs. Textures can be generated in different dimensions and will automatically cache values in the same spatial locality. This memory can improve the access performance when the threads operate values in some specific spatial neighbourhood area[13].

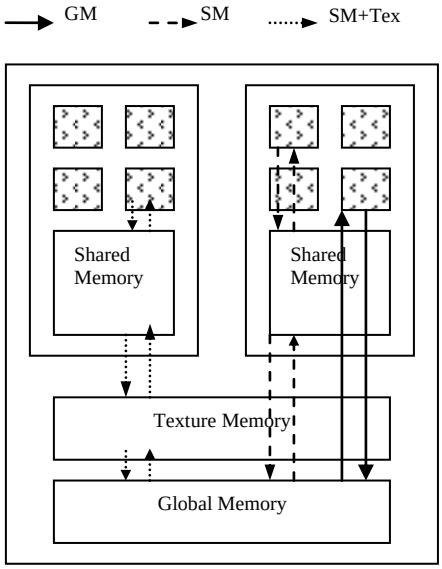

Fig4 Three Access Mechanism.

Fig.4 shows three memory accessing mechanisms. Global memory accessing mechanism denoted as GM is to access the global memory directly. This is an obvious way and easily to be implemented. However, since the global memory has the slowest access time, the performance of global memory accessing mechanism is worst in theory. To avoid bank conflict in the accessing process, GPUs should lauch fewer threads to handle the same macroblock in run time.

Shared memory accessing mechanism denoted as SM is a optimal method that mainly consists of two steps as follows. (1) Each thread block simultaneously gather the data from the global memory and place these data in the shared memory that belongs to the thread block.(2) After the kernel launching, each thread of different thread blocks access the shared memory to attain its data. The goal of SM mechanism is to achieve higher access performance by using the shared memory. The shared memory is mainly to act as an explicit access cache to the global memory.

The third accessing mechanism denoted as SM+Tex is the best scheme than the two earlier accessing

mechanisms. The SM+Tex mechanism is a improved version of the SM mechanism. When each thread block transfers data from the global memory into shared memory, the texture memory is created in 2 dimension and act as cache to map the global memory rather than directly access the global memory. This method take advantage of the spatial locality of data in the global memory to improve performance. On the other hand, this complex access pattern greatly complicates the application development.

Ⅳ. EXPERIMENTS

A. Experimental Enviroment

In experiments, we used a PCI having an AMD Athlon X2 Dual 2.1GHz and an NVIDIA GeForce GTX 280 GPU. The system is equipped with 2GB main memory and 2 GB device memory. We have installed CUDA 2.3 with Windows XP 32-bit Edition. For comparison purpose we have implemented two algorithms with three different accessing mechanisms.

B. Performance

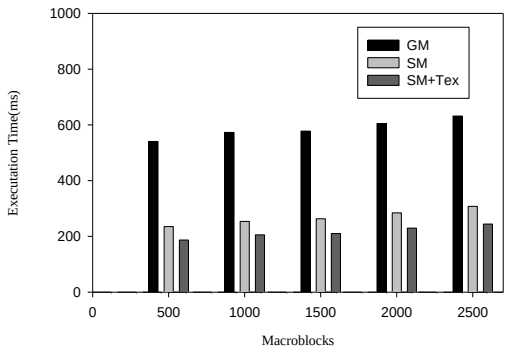

Fig5(a) the Performance of Persistent Thread Algorithm

Firstly we discusses the performance of the persistent thread algorithm using three different accessing mechanisms decribed in the previous section. Each result bar shown in the result plots given in Fig.5(a). Three different accessing mechanisms have different execution time to the same size of task. The SM+Tex mechanism outperform the two others accessing mechanisms. In contrast, it improves the performance of SM mechanism by 25% and is double than the GM mechanism. As shown in Fig.5, the execution time of each accessing mechanism increases slowly with the increasing of the number of the macroblocks. This is due to the initialization time of the kernel launching. With the increasing of the number of tasks, the initialization time overlaps by the large amount threads.

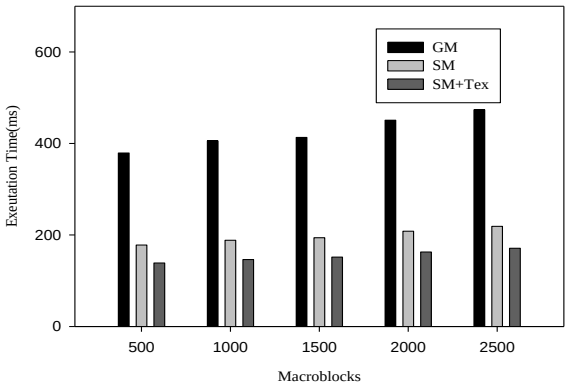

Fig5(b) the Performance of Uberkernel Thread Algorithm

Fig.5(b) illustrates the same reslut of the three different accessing mechanisms in the uberkernel algorithm. By comparing the SM+Tex mechanism with the two others, we can see that it improve the the performance of SM mechanism by 28% and is nearly double than the GM mechanism.

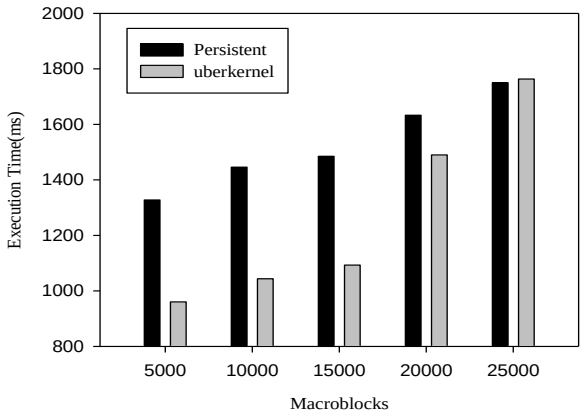

Fig5(c) Comparison of Performance of the two algorithms

By comparing the Fig5.(a) and Fig5.(b), we can conclude that the uberkernel algorithm outperforms the persistent algorithm. But in our work we should handle a large-scale video streams in real time. Then we increase the number of macroblocks to measure the performance using at most 25000 macroblocks, which require 400MB of device memory to run. Fig5.(c) illustrates that the execution time of uberkernel algorithm will be greater than the persistent one when the number of macroblocks exceeds 25000. and when the number of macroblock equal 20000, the performance of the uberkernel algorithm begin to degraded. This is due to the limitation of the uberkernel algorithm that launch a large amount of threads to execute regardless the size of computational task. So when the size of task exceed a certain value, the task should be divided into subtasks to handle and the executing time has to be increased.

V. CONCLUSIONS

We have described our data-parallel algorithms and CUDA implementations of inverse discrete cosine transform in the video evaluating projection. The uberkernel algorithm is optimal solution for the medium-sized tasks. This is due to reduce the overhead of communication between GPU and CPU and switching the kernel. Despite the high communication overhead, the persistent algorithm is well suitable to deal with large-scale video streams. Because the threads block are assigned based on the size of the task in the persistent algorithm and this way make fully use of the processors. As for accessing mechanism, GPUs allow for cached memory accesses via the GPU's texture memory. So the performance of accessing the global memory can be improved through the texture memory. The data locality is considered in shared memory accessing mechanism. Thus the SM+Tex accessing mechanism attains the best performance. In future we focus on the multiple streams technology in CUDA. This solution allows overlap kernel computation and communication between GPU and CPU. It will further improve the performance of the parallel algorithms..

ACKNOWLEDGEMENT

We would like to thank the support of National Nature Science Foundation of China (No.60970012), the Innovation Program of Shanghai Science and Technology Commission (No. 09511501000, 09220502800), and Shanghai leading academic discipline project (S30501).

REFERENCES

[1] Yunpeng Liu, Sanyuan Zhang, Shuchang Xu, et al. Research on H.264/SVC compressed video communication in 3G.[C] In the 4th International Conference on Computer Science&Education, ICCSE'09. IEEE,2009,pp:327-332

[2] Sha Hua,Yang Guo,Yong Liu,et al. Scaleable Video Multicast in Hybrid 3G/Ad-Hoc Networks[J] IEEE Transaction on Multimedia.Vol.13(2),2009.pp:402-413.

[3] Moorthy,A.K. Seshadrinathan,K. et al. Wireless Video Quality Assessment: A Study of Subjective Scores and Objective Algorithms[J] IEEE Transaction on Circuits and Systems for Video Technology. Vol.20(4),2010.pp:587-599.

[4] Qi Qu,Yong Pei, Modestino,J.W. An Adaptive Motion-Based Unequal Error Protection Approach for Real-Time Video Transport Over Wireless IP Networks[J] IEEE Transaction on Multimedia.Vol.8(5),2006.pp:1033-1044.

[5] Reibman A R, Vaishmpayan V A, Sermadevi Y. Quality monitoring of video over a packet network[J]. IEEE Trans. Multimedia, 2004, 6 (2) : 327-334

[6] Oelbaum T, Keimei C, Diepold K. Rule-based no-reference video quality evaluation using additionally coded videos[J].IEEE J. STSP, 2 009, 3 (2) : 294-303

[7] NVIDIA_Corporation, CUDA_3.0 Programming Guide, 2010, http://www.nvidia.com/ (accessed May 2010).

[8] A. K. Prasoon, K. Rajan. 4×4 2-D DCT for H.264/AVC[C] In Proceedings of the International Conference on Advances in Computing, Communication and Control. ICAC3'09,ACM.2009

[9] Stanley T. Anjul Patney and John D. Owens. Task management for irregular-paralle workloads on the GPU.In Proceedings of High Performance Graphics. Eurographics Association, Saarbruecken, Germany,June,2010,page 29-37

[10] AILA T.,LAINE S. Understanding the efficiency of ray traversal on GPUs[C].In Proceedings of High Performance Graphics 2009(Aug. 2009),pp 145-149,2,3

[11] TATARINOV A., KHARLAMOV A.: Alternative rendering pipelines using NVIDIA CUDA. Talk at SIGGRAPH 2009, http://developer.nvidia.com/object/siggraph-2009,Aug.2009.4

[12] Bingsheng He, Naga Govindaraju, Qiong Luo, Burton Smith. Efficient Gather and Scatter Operations on Graphics Processors. In:Proc. of ACM SuperComputing 2007.

[13] K. A. Hawick, A. Leist, D.P.Playne. Parallel graph component labeling with GPUs and CUDA[J]. Parallel Computing 36(2010) 655-768.

Qingkui Chen. Received the MS degree in computer science from the JILIN university, and PhD degree in University of Shanghai for Science and Technology. He is a full professor of computer science at the University of Shanghai for Science and Technology, China, where he is the head of the Network Computing Group and the Wireless Sensor Network Research Center. His research interests include GPGPU, Network Computing, and WSN. (chenqingkui@gmail.com)

Haifeng Wang. He received his diploma in computer science from the Shandong University in 1995, China. He is currently a PHD candidate in the University of Shanghai for Science and Technology. His current research interests include GPU Computing, Network Computing(gadfly7@126.com).