GPU-CPU Multi-Core For Real-Time Signal Processing

Saraju P. Mohanty
Dept. of Computer Science and Engineering
University of North Texas, TX 76207.
Email: saraju.mohanty@unt.edu

***Abstract*—Modern graphics cards are supported with powerful computational facilities for fast computation of vertex geometry and realistic rendering of 3D graphics. The introduction of programmable pipeline in the graphics processing units (GPU) has enabled configurability. GPU which is available in every computer has a tremendous feat of highly parallel SIMD processing, but its capability is often under-utilized. In this paper, we analyze the computation of general-purpose algorithms on GPU.**

I. Introduction and Motivation

Modern GPU architectures are providing ways of configuring the graphics pipeline [1], [2]. Consequently, researchers want to harness the power of the GPUs for general-purpose computation. The GPUs are designed to perform a specific number of operations on large amounts of data [1], [2].

The performance of these GPUs is increasing at an extraordinary rate. In last decade the computing power of the GUPs is increasing much faster than Moore's law. So, performance gap of GPU and CPU is widening. It is shown that a GeForceFX 5900 processor operating at 20 GigaFlopsis equivalent to a $10GHz$ Pentium 4 processor [3]. As the GPUs are becoming faster and evolving to incorporate additional programmability, the challenge becomes to provide new functionality without sacrificing the performance advantage over conventional GPUs. GPUs use a different computational model than the classical von Neumann architecture used by the CPUs. In this context, the questions arises can the GPU and CPU of a PC be used together for high-performance and low-cost computing.

II. Graphic Processing Unit (GPU) – A Broad View

The block-diagram of a modern programmable GPU is shown in Fig. 1 [1], [4], [5]. The architecture of GPU offers a large degree of parallelism at a relatively low cost through the well known vector processing model known as Single Instruction, Multiple Data (SIMD). GPU includes 2 types of processing units: vertex and pixel (or fragment) processors. The programmable vertex and fragment processors execute a user defined assembly level program having 4-way SIMD instructions. The vertex processor performs mathematical operations that transform a vertex into a screen position. This result is then pipelined to the pixel or fragment processor, which performs the texturing operations.

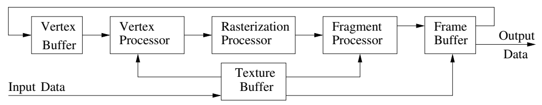

Fig. 1. The Programmable Graphics Pipeline.

GPUs are designed specifically for graphics hardware and hence are restrictive in terms of usage and programming. GPUs can only process independent vertices and fragments, but can process many of them in parallel. In this sense, GPUs are stream processors. A stream is a set of records that require similar computation and provide data parallelism. Each element in the stream are applied function using kernels. The vertices and fragments are the elements in a stream. Since GPUs process elements independently there is no way to have shared or static data. For each element one can only read from the input, perform operations on it, and write to the output. However, recent improvements in graphics cards have permitted to have multiple inputs, multiple outputs, but never a piece of memory that is both readable and writable. It is important for general-purpose applications to have high arithmetic intensity, otherwise the memory access latency will limit computation speed. Recent developments in data transfer rate through PCI Express interface to an extent addressed this issue [1], [4]. Programming the GPU in a high level language can be done using Cg from NVidia, OpenGL, etc. [6].

III. An Application Scenario – IP-TV

The Internet Protocol Television (IP-TV) scenario broadcasting live events such as sports is shown in Fig. 2. A server containing GPU and CPU performs tasks, such as MPEG compression and digital right management (DRM). As seen in Fig. 2, the video data, which arrives at shared memory, is directed by CPU and sent to GPU, then data is mapped into GPUs memory. The GPU processes the video data. After GPU finishes, the control from CPU initiates to copy the data back to CPU and the video is stored in database. This approach ensures faster video processing for vast amount of data and will not add extra hardware cost to either video providers or receivers. Data is sent to GPU as textures, which means that data is treated as group of pictures.

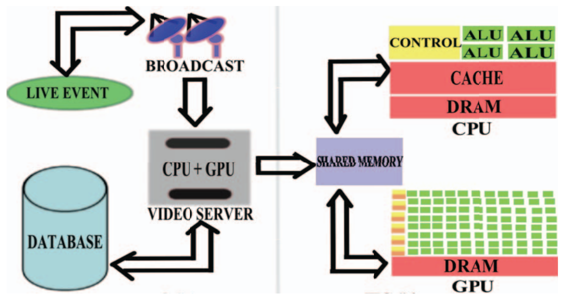

Fig. 2. The IP-TV Broadcasting Scenario Showing the Shared Architecture for GPU-CPU Multi-Core Processing in a Video Server.

IV. Computation of Discrete Cosine Transformation (DCT) – A Case Study

In applications, such as JPEG and MPEG, DCT is a resource intensive task. Thus, we consider this as a case study.

The computation of DCT on GPU is performed using one or more rendering passes. The working of a rendering pass can be divided into 2 independent stages. In the 1st stage, a number of source textures, the associated vertex streams, the render target, the vertex shader and the pixel shader are specified. The source textures hold the input data. The vertex streams consist of vertices that contain the target position and the associated texture address information. The render target is a texture that holds the resulting DCT coefficients. The 2nd stage is the rendering stage which is triggered issuing the DrawPrimitives call. The vertex shader calculates the target position and the texture address of each vertex involved in a specific primitive. Then the target texture is rasterized and the pixel shader is subsequently executed to perform pixel-wise calculations.

We used a a gray scale image data to compute the DCT using OpenGL API (Application Programming Interface). The input data are eight bit integers and the image size is 256×256. The computation involves block-wise DCT of 8×8 pixels. It also includes the facility to quantize the DCT coefficients at different levels and compute the inverse (IDCT) to recover the spatial image for verification. The display of the original and the corresponding DCT coefficient image is shown in Fig. 3.

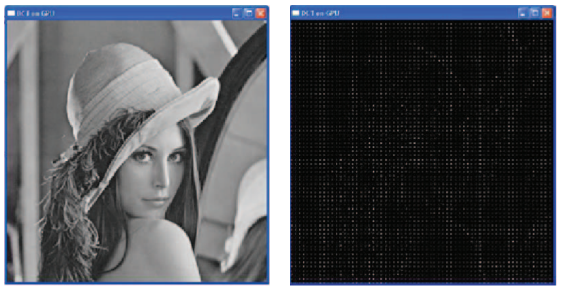
(a) The Original Image (b) The DCT Coefficient Image

Fig. 3. Computing Discrete Cosine Transform (DCT) using GPU.

We performed the experiments for the following cases: (1) using CPU only, (2) using GPU only, and (3) using both GPU and CPU. Our simulation environment computes 100 iterations of DCT. The CPU is a Pentium 4, $3.20GHz$ with RAM of $1GB$. The GPU is NVIDIA GeForce FX 5200. The simulation results are reported in Table I. The 3 parameters used to expressed results are CPU Time, GPU Time, and CPU Release Time. CPU Time refer to the total "Elapsed Time" in the system for case (1). GPU Time refer to the total "Elapsed Time" in the system for case (2). In case (3) these refer to the total "Elapsed Time" in the system when one half executed by CPU and another half executed GPU.

It can be concluded from the Table that CPU is free for more than 50% time when at least half of the computation is performed by the GPU. Although the computation power of GPU is much higher than CPU in terms of throughput, the fastest DCT technique on GPU is still slower than the implementation on CPU. The reason is that the speeds of

TABLE I
Experimental Results for Execution Time

Test Cases	CPU Time	GPU Time	CPU Release Time
CPU only	$4.376ms$	Free	Fully Occupied
GPU only	Free	$44.789ms$	Fully Free
CPU + GPU	$2.15ms$	$21.157ms$	50.87% Free

GPU is limited by memory access. However, the scenario will change with the use of more advance GPU and interface.

For best performance (i.e. both GPU and CPU work together and complete the work in same time to be free at same time for other computations) using GPU-CPU as multi-core the following relationship can be derived. Let us assume the following: T_{CPU} - CPU execution time, T_{GPU} - GPU execution time, $\alpha = \left(\frac{T_{GPU}}{T_{CPU}}\right)$, P - total number of primitive operation is an algorithm, β - a fraction, and Q - the penalty due to Amadahl's law in terms of number of primitive operations due to parallel execution of GPU and CPU. The throughput of a GPU is higher than that of CPU due to long pipelining, however, the execution time of GPU is lower than CPU, thus, typically, $\alpha < 1$. Thus, for best system performance with GPU-CPU multi-core the following relation must be satisfied:

$$\beta \times (P + Q) \times T_{GPU} = (1 - \beta) \times (P + Q) \times T_{CPU}. \quad (1)$$

Algebraically we deduce the following relation:

$$\beta = 1/\left(1 + \alpha\right). \quad (2)$$

Thus, the algorithm of an application needs to partitioned to $\left(\frac{1}{1+\alpha}\right)$ and $\left(\frac{\alpha}{1+\alpha}\right)$ primitive operations to run in GPU and CPU respectively for optimal system performance.

V. Summary and Conclusions

In this work, we have shown that GPU is an excellent candidate for performing data intensive signal processing algorithms. A direct application of this work is to perform DCT and IDCT on GPU in real-time signal processing to be used for broadcasting. Research is in full swing to develop a programming environment. The disadvantage in GPU is in the fact that the data transfer rate from the graphics hardware to the main memory is very slow. This bottleneck degrades the performance as it is needed to bring the results to the main memory in the context of general purpose computation.

VI. Acknowledgment

Author acknowledges UNT graduate Nishikanta Pati.

References

[1] K. Fatahalian and M. Houston, "GPUs: A Closer Look", *ACM Queue*, Vol. 6, No. 2, March/April 2008, pp. 18–28.
[2] C. J. Thompson, S. Hahn, and M. Oskin, "Using Modern Graphics Architectures for General-Purpose Computing: A Framework and Analysis", in *Proc. 35th International Sympo. Microarchitecture*, pp. 306-317, 2002.
[3] I. Bucks, "Data Parallel Computing on Graphics Hardware", Graphics Hardware 2003 Panel: GPUs as Stream Processors, July 27, 2003.
[4] J. D. Owens, et. al, "A Survey of General-Purpose Computation on Graphics Hardware", *Eurographics 2005*, August 2005, pp. 21-51.
[5] S. P. Mohanty, N. Pati, and E. Kougianos, "A Watermarking Co-Processor for New Generation Graphics Processing Units", in *Proc. 25th IEEE International Conference on Consumer Electronics*, pp. 303-304, 2007.
[6] General-Purpose computation on GPUs, http://www.gpgpu.org/