Accelerated Cone Beam CT Reconstruction Based on OpenCL

Bo Wang[#1], Lei Zhu[*2], Kebin Jia[#3], Jie Zheng[*4]

[#]*College of Electronic Information and Control Engineering, Beijing University of Technology*
Beijing, China
[1]sindywb@emails.bjut.edu.cn
[3]kebinj@bjut.edu.cn

[*]*Corporate Technology, Siemens Ltd. China*
Beijing, China
[2]z.lei@siemens.com
[4]zheng.jie@siemens.com

Abstract— **Open Computing Language (OpenCL) is a fundamental technology for cross-platform parallel programming. The emerging of OpenCL provides portable and efficient access to the power of modern processors. This revolutionary new technology is applied to accelerate the reconstruction of cone beam computed tomography (CBCT) on Graphics Processing Unit (GPU) in this paper. An OpenCL-based implementation of the Feldkamp-Davis-Kress (FDK) algorithm is presented. The required transformations to parallelize the algorithm for the OpenCL architecture are also explained. Comparing to the conventional CPU-based implementation, the proposed method reaches an over 57 times speedup. Experimental results show a great performance boost, which can pave the way for widespread application and new conceptual innovation of CBCT. Besides, the feasibility and potential of OpenCL-based implementation are also indicated.**

Keywords— **image reconstruction; cone beam computed tomography; FDK; parallel computing; OpenCL**

I. INTRODUCTION

Computed tomography (CT) is widely applied in the clinical arena. The use of cone beam computed tomography (CBCT) is growing rapidly due to its ability to provide 3-D information with short scanning time. But in many situations, the short scanning time is followed by a time consuming 3-D reconstruction, which prohibits the use of CBCT for many clinical applications. On the other hand, the highly parallel nature of CBCT reconstruction algorithms and great power of modern processors enable the accelerated computing solutions to gain a significant performance boost [1]. Consequently, several hardware accelerated approaches [2-3] have been developed based on various accelerators, such as multi-core CPU, Field Programmable Gate Array (FPGA), Cell processor and Graphics Processing Unit (GPU).

However, traditional parallel programming models are usually platform-, vendor- or hardware-specific which may be a barrier to the growth of parallel computing. Taking the Compute Unified Device Architecture (CUDA) for example, it is a powerful GPU programming interface for general- purpose computation. But it is not device agnostic. Currently, CUDA can only work on NVIDIA GPUs. If we want to adapt a CUDA implementation to an AMD GPU or multi-core CPU, the programming effort will be demanding. Thus a generic developing model is in urgent need and that is just what the Open Computing Language (OpenCL) is designed for. As the first truly open and royalty-free programming standard for general-purpose computations on heterogeneous systems, OpenCL allows programmers to preserve their expensive source code investment and easily target multi-core CPUs, latest GPUs and other modern processors. Meanwhile, among those multi-cores, GPU has become a compelling platform due to its rapid increases in performance and recent improvements in programmability. Thus, implementing the acceleration of CBCT reconstruction on GPU using OpenCL can bring us a significant performance boost while not losing much versatility and scalability. And from experimental results we can see, with an over 57 times speedup, it is really a promising solution to use OpenCL in computationally demanding tasks like CBCT reconstruction.

The rest of this paper is organized as follows. In Section II background information on the OpenCL programming model and the CBCT reconstruction method is reviewed. In Section III, details of our implementation are presented. Then how the algorithm proposed is evaluated and corresponding results are shown in Section IV. Finally, a conclusion with some ideas for future work is given in Section V.

II. BACKGROUND

A. Open Computing Language

Open Computing Language [4] (OpenCL) is an open royalty-free standard for general purpose parallel programming across CPUs, GPUs and other processors. It was publicly released by Khronos Group in December 2008 and has been developing very fast with the cooperation of industry leaders, such as Apple, NVIDIA and AMD.

In contrast to other parallel programming models, OpenCL is platform-, vendor- and hardware-independent, consisting of an API for coordinating parallel computation across heterogeneous processors and a cross-platform programming language with a well specified computation environment. The following hierarchy of models is used:

978-1-4244-5555-3/10/$26.00 ©2010 IEEE

- Platform Model: OpenCL platform model consists of a host connected to one or more OpenCL devices.
- Execution Model: Execution of an OpenCL program includes two parts: kernels execute on OpenCL devices and a host program executes on the host.
- Memory Model: Four distinct memory regions are defined: global memory, constant memory, local memory and private memory.
- Programming Model: Both data parallel and task parallel programming models are supported.

Basic OpenCL program structure [5] is shown in Fig.1. There are two parts: host program and kernels. Host program queries available compute devices and creates corresponding running contexts. Through issuing commands to command-queues by the host, kernels are invoked and computations will be performed on OpenCL devices.

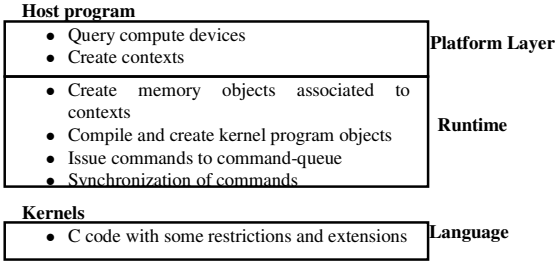


Fig. 1 Basic OpenCL program structure

B. Feldkamp-Davis-Kress Method

Feldkamp-Davis-Kress (FDK) [6] developed first practical algorithm for CBCT reconstruction in 1984. It is an approximate reconstruction algorithm for circular cone beam tomography and has been commonly used as a standard CBCT reconstruction approach [7].

As the scanning geometer shown in Fig.2, the FDK method can be expressed as [8] [9]:

$$f(x,y,z)=\int_0^{2\pi}\frac{D^2}{U^2}\tilde{p}(\beta,u,v)d\beta \tag{1}$$

Where

$$U=D+x\cos\beta+y\sin\beta \tag{2}$$

$$u=(D/U)(-x\sin\beta+y\cos\beta) \tag{3}$$

$$v=z(D/U) \tag{4}$$

$$\tilde{p}(\beta,u,v)=\left(\frac{D}{\sqrt{D^2+u^2+v^2}}p(\beta,u,v)\right)*h(u) \tag{5}$$

And (x,y,z) are voxel coordinates, (u,v) are detector coordinates, D is the gantry radius, β is the projection angle, $p(\beta,u,v)$ represents the raw data acquainted by detector , $h(u)$ represents the ramp filter.

The FDK method can be performed in three steps: pre-weight projection data, ramp filter the projection images row wise, and back-project the filtered projection data into reconstructed volume. Each row of the projection images can be processed independently in most time. Even each pixel can be processed independently in some cases. Thus, there is a highly parallel nature in the FDK method, which enables a lot of parallel accelerated solutions utilizing the power of multi-cores such as GPU [1-2], [7] and Cell processors [3]. Similarly, the FDK method can be fitted into data parallel model under OpenCL architecture.

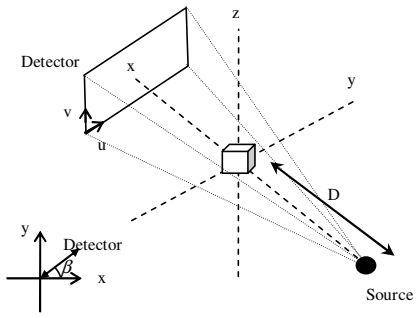


Fig. 2 Geometer of cone-beam apparatus

III. Our Implementation

A. OpenCL Initialization

According to OpenCL architecture, there is a host connected to one or more OpenCL devices. And host program manages the whole process to perform computations on devices.

In our program, CPU is the host and GPU is the OpenCL device. As there is only one OpenCL device in our system, to create one command-queue is a natural choice. Actually two command-queues cost more execution time according to our experiments. Maybe resource limitation and invocation overhead are the main reasons.

After setting up platform layer, the runtime is to be dealt with. Computations are performed through kernel executions. Kernel objects need to be created before invocation. OpenCL supports both online and offline build. Of course, for released version, offline build should be applied to reduce initialization time, but in debugging stage, online build is more convenient. After building up all runtime objects, computations can be performed.

B. Parallelization Strategy

How to parallelize the FDK method is the next issue.

As an initial thinking, we just focused on employing the computational intensive parts on GPU. Other operations, such as data rearrangement, were executed on CPU. It's to some extent justified, but not optimal here. Because it caused a lot of data transfer between the host and the device which is every expensive. Thus besides computationally demanding parts, some kernels, which may not gain much performance improvement but can contribute to minimizing data transfer, are also performed. Further, due to the limited memory capacity of GPU, not all the input and output data can be stored in device memory. Data transfer is inevitable. To alleviate this problem, page-locked host memory, instead of non-pinned memory, is used to store projections. We also take the advantage of GPU on chip memory (local memory) to save the memory bandwidth.

Thus our projection wise scenario is presented in Fig.3. Host program manages the whole processing loop for each projection angles, during which the computations are executed on GPU (OpenCL device). Basic FDK

processing method is mapped to six successive OpenCL kernel executions: pre-weighting, filtering, transpose, grids mapping, weighting and accumulation. As shown in Fig.4, kernels are executed under data parallel model in an N-dimensional index space (N is one, two or three). Each point in the index space is a single kernel instance, called a work-item. And work-items are further organized into work-groups. Kernel executions are managed by host program and coordinated through command-queue.

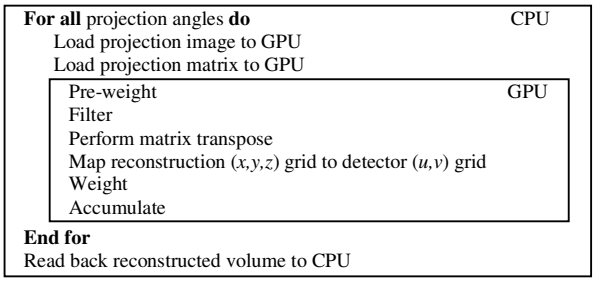


Fig.3 OpenCL-based FDK implementation

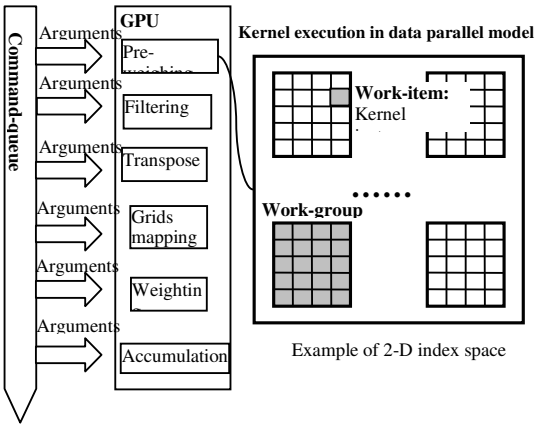


Fig.4 Illustration of kernel execution

C. Kernel Design

Kernels are the basic functions executed on OpenCL devices. As mentioned above, six kernels are designed to perform the parallel implementation of FDK method.

Pre-weighting kernel performs a multiplication between the projection and pre-weighting factor. A 2-D index space of N_row*N_col is defined to execute parallel pre-weighting. Thus theoretically there are N_row*N_col concurrent kernel instances. Where N_row is the height of projection image and N_col is the width.

Filtering kernel applies the Shepp-Logan filter [10] to each projection, aiming at minimizing noise propagation at the back projection stage. This can be computed by a complex 1-D FFT followed by the point-wise multiplication of the filter kernel and then the computation of the inverse FFT of the respective point-wise product. Parallel filtering is executed in a 2-D index space of (N_col/2)*N_row. Each work-group is used to perform the filtering of one image row. Thus N_row work-groups are needed. And every work-group consists of N_col/2 work-items; each performs one butterfly operation of the FFT (or inverse FFT). Besides, local memory is used to reduce memory access latency. So synchronization between work-items is necessary. Additionally, some constant tables are used, including bit inversion table, trigonometric coefficient table and Shepp-Logan filter kernel table. They all have been set up in initialization stage.

Transpose kernel performs a parallel matrix transpose by splitting matrix into small tiles with the usage of local memory to reduce transfer latency [11].

Grids mapping kernel uses projection matrix to map reconstruction (x,y,z) grid to detector (u,v) grid. Actually, a matrix multiplication is performed, in which big matrix is split into small tiles to implement parallel multiplication.

Weighting kernel performs a multiplication between weighting factor D^2/U^2 and the projections, during which a nearest neighbor interpolation is used to determine discrete detector coordinates (u,v) of projection data. Just a 1-D index space of N_X*N_Y*N_Z is used, where N_X, N_Y and N_Z describes the interested region of volume. Actually, only Region-Of-Interest (ROI) is reconstructed in our implementation.

Lastly, accumulation kernel adds increments to reconstructed ROI. Necessary matrix transpose is also performed and local memory is used.

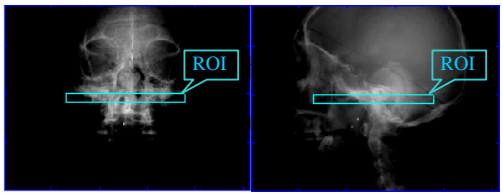
IV. Results

Our algorithm is evaluated on a head phantom (sample data in OSCaR-02 [12]), in which 320 projections of size 256*192 were acquired. The gantry has a source to axis distance of 100cm, a source-to-detector distance of 155 cm. Both the input projections and the final reconstructed volume are stored in 32-bit floating point format.

Two ROIs of (-15cm~15cm, -15cm~15cm,-1cm~1cm) and (-30.5 cm~30.7cm, -30.5cm~30.7cm, -15. 6cm~14.9cm) are selected, corresponding to 126*126*9 and 256*256*128 reconstructed volume. Evaluations are based on the following hardware configurations:

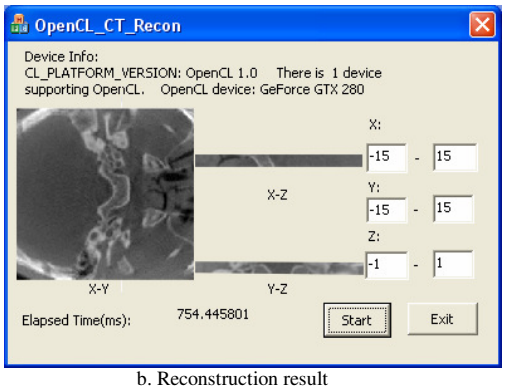
- Dell Precision T5400 with Intel Xeon CPU E5405 @ 2.00GHz, 4GB of RAM; NVIDIA GeForce GTX280 with GPU driver v190.89 and NVIDIA gpucomputingsdk_2.3a.

One of the experimental results is shown in Fig.5. To evaluate the performance improvements driven by the OpenCL-based GPU implementation, each kernel execution time and the total time are determined (See Table 1), where transfer and other overhead are included in overall time but not in core time. The whole program speedup is shown in Fig.6. Separate speedup of each kernel is shown in Fig.7. One thing should be noted that we did not pay much attention to optimize the serial code which the parallel code was based on. Thus the actual execution time can be further reduced through algorithm optimizations where our parallelization strategy may also drive similar performance improvements.



0 degree Projection 90 degree Projection

a. Projections and ROI



b. Reconstruction result

Fig.5 Experimental result

TABLE I
COMPUTATION TIME

	126*126*9			256*256*128		
	CPU (ms)	GPU (ms)		CPU (ms)	GPU (ms)	
		over-all	core		over-all	core
pre-weighting	1750	19	17	1754	19	17
filtering	15332	231	229	15326	227	225
transpose	146	21	20	147	21	20
grids mapping	7144	174	174	474477	10204	10199
weighting	3087	33	31	280581	1149	1130
accumu-lation	707	174	172	48278	2637	2636
transfer	N/A	53	N/A	N/A	85	N/A
total	28166	705	643	820562	14343	14227

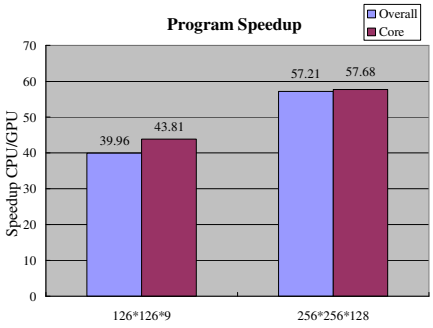
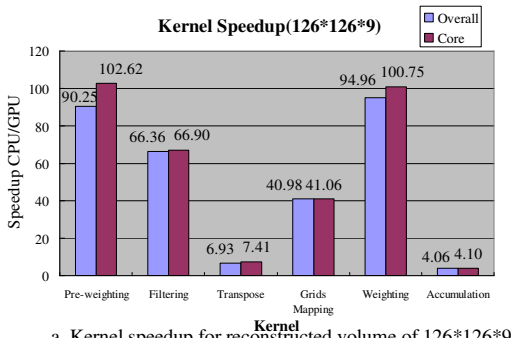
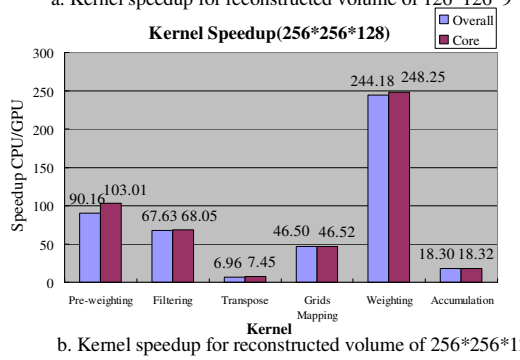


Fig.6 Program speedup



a. Kernel speedup for reconstructed volume of 126*126*9



b. Kernel speedup for reconstructed volume of 256*256*128

Fig.7 Kernel speedup

From results above, we can see that OpenCL-based parallel implementation drives significant performance improvements over conventional CPU implementation (using C language). For a reconstructed volume of 256*256*128, the total program speedup is over 57 times (See Fig.6) and the highest kernel speedup is nearly 250 times (See Fig.7). Moreover, it seems that, as reconstructed volume gets larger, the speedup gets higher and the proportion of overhead turns smaller, which may benefit us more in real clinical applications.

V. CONCLUSIONS

As a fast developing technology, OpenCL enables software developers to take full advantage of heterogeneous processing platforms. This revolutionary new technology is applied to the FDK method in cone beam CT reconstruction in this paper. For a reconstructed volume of 256*256*128, the overall speedup is over 57 times (See Fig.6). Experimental results indicate significant performance improvements, which may break the application barrier for CBCT and move it to a more advanced state. The feasibility and potential of OpenCL-based implementation are also demonstrated. Although regarding the technology maturity, OpenCL is just a baby (For the similar implementations on GPU, it may be a little slower than CUDA currently), undoubtedly as it's thriving, OpenCL will bring about higher performance and more innovative usage of the hardware.

For future work, further optimizations of parallelization strategy and memory management are in the plan. Evaluating OpenCL on other hardware platforms is also being considered. The similar CUDA implementation and other more complicated reconstruction algorithms may also be performed for further comparison.

REFERENCES

[1] D. Riabkov, X. Xue, D. Tubbs, and A. Cheryauka., "Accelerated cone-beam backprojection using GPU-CPU hardware," *Proceedings of 9th International Meeting Fully Three-Dimensional Image Reconstruction in Radiology and Nuclear Medicine*, July 2007, pp. 68–71.

[2] X. Xue, A. Cheryauka, and D. Tubbs, "Acceleration of fluoro-CT reconstruction for a mobile C-Arm on GPU and FPGA hardware: A simulation study," *Proceedings of SPIE Medical Imaging*, San Diego, Calif, USA, vol.6142, February 2006, pp.1494-1501.

[3] O. Bockenbach, S. Schuberth, M. Knaup, and M. Kachelrieß, "High performance 3D image reconstruction platforms; state of the art, implications and compromises," *Proceedings of 9th International Meeting on Fully Three-Dimensional Image Reconstruction in Radiology and Nuclear Medicine*, vol. 9, 2007, pp. 17-20.

[4] *OpenCL 1.0 Specification*, Khronos OpenCL Working Group, 2009.

[5] (2009) OpenCL Overview Multicore Expo Mar09, Khronos. [Online]. Available:http://www.khronos.org/library/detail/multicore_expo_2009/

[6] L. A. Feldkamp, L. C. Davis, and J. W. Kress, "Practical cone-beam algorithm," *J. Opt. Soc. Am. A*, vol. 1, pp.612–619, 1984.

[7] H. Scherl, B. Keck, M. Kowarschik, and J. Hornegger, "Fast GPU-based CT reconstruction using the common unified device architecture (CUDA)," *Nuclear Science Symposium, Medical Imaging Conference*, 2007, Hawaii ,USA, vol. 6, 2007, pp. 4464-4466.

[8] H. Turbell, "Cone-beam reconstruction using filtered back projection," Ph.D. thesis, Linköping University, Linköping, Sweden, Feb. 2001.

[9] N. Rezvani, D. Aruliah, K. Jackson, D.Moseley, and J. Siewerdsen, (2009) "OSCaR: an open source cone-beam CT reconstruction tool for imaging research", The Huangguoshu International Interdisciplinary Conference on Biomedical Mathematics, 2008, Huangguoshu, China. [Online]. Available: http://www.cs.toronto.edu/%7Enrezvani/

[10] L. A. Shepp and B. F. Logan, "The fourier reconstruction of a head section," *IEEE Trans. Nuclear Science*, vol.21, pp. 21–43, 1974.

[11] NVIDIA, (2009) NVIDIA OpenCL Best Practices Guide Version 1.0. [Online]. Available: http://www.nvidia.com/object/cuda_opencl.html

[12] N. Rezvani, (2009) OSCaR: An Open-Source Cone-Beam CT Reconstruction Tool for Imaging Research. [Online]. Available: http://www.cs.utoronto.ca/~nrezvani/OSCaR.html