GPU Acceleration of 3D Forward and Backward Projection Using Separable Footprints for X-ray CT Image Reconstruction

Meng Wu and Jeffrey A. Fessler

***Abstract*—Iterative 3D image reconstruction methods can improve image quality over conventional filtered back projection (FBP) in X-ray computed tomography. However, high computational costs deter the routine use of iterative reconstruction clinically. The separable footprint method [1] for forward and back-projection simplifies the integrals over a detector cell in a way that is quite accurate and also has a relatively efficient CPU implementation. In this project, we implemented the separable footprints method for both forward and backward projection on a graphics processing unit (GPU) with NVDIA's parallel computing architecture (CUDA). This paper describes our GPU kernels for the separable footprint method and simulation results.**

I. Introduction

Iterative statistical methods for 3D tomographic image reconstruction offer the potential for improved image quality and reduced dose compared to conventional methods such as filtered back-projection (FBP). The main disadvantage of iterative reconstruction methods is the longer computation time. Most iterative reconstruction methods require one forward projection and one back-projection per iteration. These operations are the primary computational bottleneck.

The study of fast and efficient reconstruction algorithms for large 3D images and their implementation on hardware and in software is important both theoretically and practically [2]. The separable footprint (SF) projectors approximate the voxel footprint functions as 2D separable functions [1]. This approximation is reasonable for typical axial or helical cone-beam CT geometries. The separability of these footprint functions greatly simplifies calculating their integrals over a detector cell and allows efficient implementation [1]. GPUs provide high performance for highly parallel computations [3], [4]. This paper describes how we adapted the SF projector algorithm to a high-end multi-GPU platform using the compute unified device architecture (CUDA) API from NVDIA. There are many ways to adapt the algorithms to GPU. We investigated several different kernel and memory structures for GPU implementation to optimize speed. We also compared GPU and CPU run times by simulating a typical helical CT scan.

Dept. of Electrical Engineering and Computer Science, University of Michigan, 1301 Beal Ave., Ann Arbor, MI 48109-2122, U.S.A. Email: {febmeng, fessler} @umich.edu. Supported in part by NIH grant R01-HL-098686.

II. Separable footprint method

Mathematically, any 3D projector and back-projector for helical or axial CT can be represented in the general form:

$$\begin{aligned} g(s,t,\beta) &= \sum_{x,y,z} a(s,t,\beta;x,y,z) f(x,y,z), \\ b(x,y,z) &= \sum_{s,t,\beta} a(s,t,\beta;x,y,z) g(s,t,\beta), \end{aligned} \quad (1)$$

where $f(x,y,z)$ and $b(x,y,z)$ denote the image voxel values at 3D spatial location x,y,z, and $g(s,t,\beta)$ denotes the measured projection views. The parameter β indexes the projection view angles and s,t denote the transaxial and axial coordinates of a 2D projection view. We assume that the row coordinate t on the detector is aligned with the axial coordinate z within the object. The function $a(s,t,\beta;x,y,z)$ is the system model and denotes the footprints of the voxel centered at x,y,z blurred by the detector element size.

The SF method [1] models the (blurred) footprint function as follows:

$$\begin{aligned} a(s,t,\beta;x,y,z) = &\, v(s,t,\beta)\, u(\beta;x,y) \\ &\cdot F_1(s,\beta;x,y) F_2(t,\beta;x,y,z), \end{aligned} \quad (2)$$

where the footprint functions F_1 and F_2 approximate the shape of the true footprint for small cone angles. F_1 is a trapezoid function in the transaxial direction and F_2 is a rectangular function in the axial direction, called the SF-TR method. The factors $u(\beta;x,y)$ and $v(s,t,\beta)$ are simple amplitude functions described in [1] as the "A2" method, and they require minimal computation time. The main computational work is related to F_1 and F_2. Because F_1 depends only on s (detector column) and F_2 depends only on t (detector row), the SF-TR-A2 method has a particularly good trade-off between accuracy and computation time so we focused on its GPU implementation. The units of a are cm so that the reconstructed image f has linear attenuation units 1/cm.

A. Forward projection parallelization

For a single-core system it is natural to work on one projection view at a time. Likewise, a simple approach to parallelization is to have each thread work simultaneously on a distinct single projection view, so that n threads produce n projection views concurrently. However, parallelizing only over projection views would be a suboptimal GPU implementation, because GPUs usually have many more thread processors than

a multi-core CPU. The thread processors within a GPU are more suitable for running massively parallel single instructions than for the complex computations required for an entire projection view. Thus it can be preferable to use a GPU's many threads in parallel to compute a single projection view.

For threads working on a single value of β (a single projection view), we first initialize an accumulation array: $g(s,t,\beta) = 0$. The key to efficient implementation is to rewrite (1) using the distributive property of addition and multiplication and the separability (2) as follows:

$$g(s,t,\beta) = v(s,t,\beta) \sum_{x,y} F_1'(s,\beta;x,y) \cdot \left[\sum_z F_2(t,\beta;x,y,z) f(x,y,z) \right], \quad (3)$$

where we define the modified transaxial footprint function:

$$F_1'(s,\beta;x,y) = u(\beta;x,y) F_1(s,\beta;x,y). \quad (4)$$

The inner sum over z (within square brackets in (3)) does not depend on s, *i.e.*, it involves only the axial direction. For a single view, β is a fixed constant, so one natural approach to parallelization is to perform the axial summation over z for several (x,y) locations concurrently, yielding 1D arrays in t:

$$h(t,\beta;x,y) = \sum_z F_2(t,\beta;x,y,z) f(x,y,z). \quad (5)$$

Then, for each detector row (each t value) we add

$$F_1'(s,\beta;x,y)\, h(t,\beta;x,y)$$

to the running accumulator, *i.e.*,

$$g(s,t,\beta) \mathrel{+}= F_1'(s,\beta;x,y)\, h(t,\beta;x,y), \quad (6)$$
$$\forall s \in [s_{\min}(\beta;x,y), s_{\max}(\beta;x,y)],$$

where $s_{\min}$ and $s_{\max}$ define the transaxial support of the footprint for each voxel column (each x,y value): $\{s : F_1'(s,\beta;x,y) \neq 0\} \subseteq [s_{\min}(\beta;x,y), s_{\max}(\beta;x,y)]$.

After computing (5) for some (or all) of the (x,y) locations, threads can then compute (in parallel) the footprint values F_1' in (4) for the same set of (x,y) locations. Alternatively the threads could compute these footprints before computing (5); this approach might seem slightly suboptimal, because the transaxial footprints F_1' for a given (x,y) location are not needed until after (5) is computed. However, some of the geometric factors needed for computing F_2 are also relevant to computing F_1', and F_1' requires very little storage, so this alternative approach may be the most efficient. Having computed the axial sums (5) and the footprint values F_1' in (4), the next step is to perform the accumulation (6).

For a thread working on a single value of β, we can loop over (x,y), and do the accumulation (6) for each (x,y) location sequentially. However, in a GPU implementation, there is a subtlety that in general the footprints of different voxels overlap, so parallelization across arbitrary (x,y) values (or arbitrary s values) would cause read-write errors. One standard way of preventing such errors is to use a `mutex` or a conditional variable to ensure that only one thread at a time is allowed to update the view at given s value. The CUDA environment has no efficient mechanism for avoiding such read-write errors. Using shared memory and/or synchronizing after each update could prevent the errors, but would significantly increase computation time.

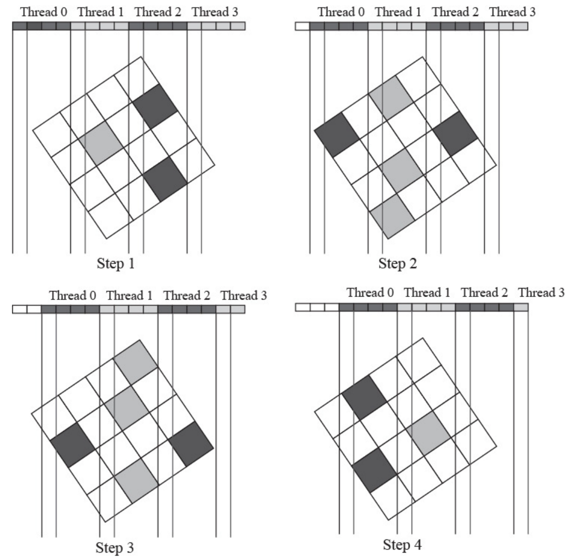

Fig. 1. Parallelization of grouped s locations with disjoint footprints. Because the maximum of footprints along s direction is 4, we put every 4 consecutive detector cells into a group. One thread can update the shaded voxels, having $s_{\min}$ at the first detector cell of each group, simultaneously to its own s-group without memory conflicts. After four steps, all voxels have contributed to the projection view.

To overcome this problem, we choose the set of (x,y) values that are parallelized during the accumulation (6) such that their corresponding footprints do not overlap. Instead of parallelizing over all (x,y) values, we let each GPU thread processor update only a specific group of detector cells. For example, Fig. 1 illustrates a simple case where 4 threads are running for a 16×1 detector. Because typically N_{s} is many hundreds, and the maximum footprint size along s usually is small, there is opportunity for substantial parallelization across s in this way. For example, $N_{\mathrm{s}} = 888$ for a typical GE CT scanner and if $s_{\max} - s_{\min} < 10$ then more than 80 threads can work simultaneously with no read-write errors. It is easy to identify suitable collections of (x,y) locations because we must compute $s_{\min}(\beta,x,y)$ anyway and we can group together (x,y) locations that have the same value of $s_{\min}(\beta,x,y)$. We found that the upper bound on the size of a set is twice the image transaxial size ($2 \times N_{\mathrm{x}}$). In addition, because each detector row (t value) is incremented independently in (6), this accumulation is amenable to parallelization over t. For a helical scan, the number of z values (slices) is much larger than the number of t values (detector rows), so we can further parallelize the method by computing the projection views having same projection angle β concurrently.

B. Back-projection parallelization

The back-projection operation for SF-TR-A2 has the form:

$$b(x,y,z) = \sum_x \sum_y F_2(t,\beta;x,y,z) \cdot \left[\sum_s F_1'(s,\beta;x,y)(v(s,t,\beta)g(s,t,\beta)) \right] \quad (7)$$

To implement this operation it is again natural to work on one view at a time (one value of β). Before starting the loop over β, we initialize the back-projection array $b(x,y,z) = 0$. The first step is to multiply the projection view $g(s,t,\beta)$ by the ray-dependent amplitude factors:

$$g'(s,t,\beta) = v(s,t,\beta)g(s,t,\beta). \quad (8)$$

This multiplication can be easily parallelized over (s,t) values. The CPU can do this step very quickly and it has very small influence on total computational time. The next step is to compute the inner products along s in (7):

$$h(t,\beta;x,y) = \sum_s F_1'(t,\beta;x,y)g'(s,t,\beta). \quad (9)$$

Each GPU thread can perform this summation for several (x,y) locations and t values. For this step each thread needs to perform up to 10 multiplies and adds, and then store just one $h(t;x,y,\beta)$ for each (x,y) and t value. This process yields a data structure with size $N_x \times N_y \times N_t$ in GPU global memory, which can be acceptable storage. After a thread computes $h(t;x,y,\beta)$, it increments the back-projection accumulation array along z as follows:

$$b(x,y,z) \mathrel{+}= \sum_t F_2(t,\beta;x,y,z)h(t,\beta;x,y). \quad (10)$$

Because each thread works on disjoint (x,y) locations, read-write problems are prevented by parallelizing over all (x,y,z) locations of the 3D image. By analogy with (5), for a given z value typically there will be only a very small number of t values for which F_2 is nonzero; in fact typically at most 3 values for a multi-slice CT scanner geometry with the natural slice thickness. For a helical scan, the number of z values (N_z slices) is much larger than the number of t values (N_t detector rows), so each projection view will modify only a small part of the 3D image. Thus, we need only $3 \times N_x \times N_y \times N_t$ threads rather than $N_x \times N_y \times N_z$ to implement (10) efficiently.

III. GPU implementation and simulation results

We implemented the forward and backward projectors on a GPU with NVDIA's CUDA environment. We separated the algorithms into several kernels and optimized the number of declared threads for each kernel to help ensure all GPU threads work efficiently. We simulated the geometry of a GE LightSpeed X-ray CT system with an arc detector of $N_s = 888$ detector channels for $N_t = 64$ detector rows with $N_\beta = 984$ views over 360° for a 3D object of size $512 \times 512 \times 640$. We evaluated the elapsed time using the average of 5 projector runs on a 12-core, 4-GPU server with two 2.66 GHz Xeon X5650 processors and four 1.15 GHz NVDIA Tesla C2050 graphic cards. Because of the "hyperthreading" of the Intel Nehalem architecture, for the CPU version we used 24 POSIX threads where each thread computes a set of projection views.

A. Forward projection

The proposed forward projector uses the following steps. The indented kernel steps following each "parfor" execute in parallel.

- Initialize projection view array to zero: $g(s,t,\beta) = 0$.
- CPU loop: for each projection angle β:
  1) **GPU kernel 1**: parfor each x and y ($N_x \times N_y$):
     - Compute and store $F_1'(s,\beta;x,y)$.
     - Compute and store $s_{\min}(\beta;x,y)$.
  2) **CPU function**: For each x and y:
     - Construct voxel sets according to $s_{\min}$.
  3) CPU loop: for each detector row group (loop from 0 to 9 for GE CT scan):
     - **GPU kernel 2**: parfor each s and t ($\lceil \frac{N_s}{10} \rceil \times N_t$).
       * For each voxel set (loop over $2 \times N_x$):
         · Compute $h(t,\beta;x,y,z)$ in (5).
         · Run accumulation (6) into local array.
       * Increment projection view using local arrays
     - **GPU kernel 3**: parfor each s and t ($N_s \times N_t$):
       * Scale the projection view by $u(s,t,\beta)$.

Table 1 shows the computation time of the GPU kernels and the CPU function for a single 360° turn. The CPU function only reads $s_{\min}$ with a loop over (x,y) and then constructs voxel sets, but it consumes the most time for forward projection. It is unclear if it can be parallelized. That means in this algorithm, voxel set construction time in CPU is one of the largest limitations to GPU acceleration.

TABLE I
FORWARD PROJECTION COMPUTATION TIME FOR 1 HELICAL TURN.

GPU kern. 1	CPU func.	GPU kern. 2	GPU kern. 3	Total
7.8 s	9.9 s	2.1 s	1.1 s	20.9 s

Fortunately, most helical CT scan use multiple turns around the object. Projection views at same projection angle β have exactly same values of F_1', F_2, u, v. Only the relevant image slices $f(x,y,z)$ differ when GPU kernel 2 computes (5). Therefore, we can use the results from GPU kernel 1 and the CPU function for every projection view having the same projection angle β, saving considerable redundant computation. In addition, we also further parallelized over the projection views in GPU kernels 2 and 3. There is also a limitation in this parallelization. Because a CUDA block has at most 512 threads, and we want to put threads with same t value in one block to use shared memory, we can parallelize at most $8(512/N_t)$ projection views in GPU kernel 2. In GPU kernel 2, threads that have same s values use the same voxel sets. Because these threads use the same $F_1'(s,\beta;x,y)$ and $s_{\min}$ values, storing these values in shared memory can significantly

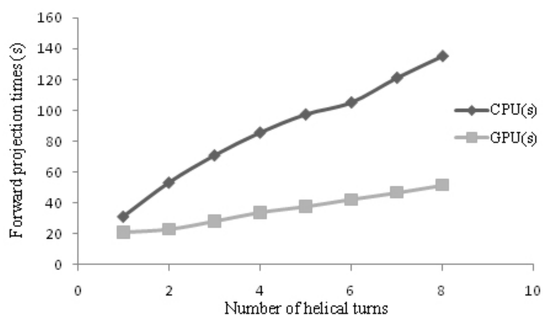

Fig. 2. Computation times for CPU and GPU implementations for different numbers of helical turns when parallelizing over projection views at same angle.

reduce the time spent reading data from the global memory. Fig. 2 shows that, in general, more helical turns leads to larger accelerations in our GPU implementation.

Our test case with a GE CT scan geometry uses about 640 MB for the 3D image, 2 MB for 8 projection views, 10 MB for F_1', F_2, u, v and other values. The total GPU global memory used is less than 1 GB. Thus, the Tesla C2050 with 3 GB total memory can easily support our GPU implementation. The global memory accesses are about $12N_\text{x}N_\text{y}$ in kernel 1, and $22N_\text{s}N_\text{t}$ in kernels 2 and 3.

B. Back projection

The proposed back projector uses the following steps:

- Initialize image data array to zero, *i.e.*, $b(x, y, z) = 0$.
- Scale the projection views with $u(s, t, \beta)$ per (8).
- For each projection view at angle β:
  1) **GPU kernel 1**: parfor each x and y ($N_\text{x} \cdot N_\text{y}$):
     - Compute and store $F_1'(s, \beta; x, y)$.
     - Compute and store s_min and s_max.
  2) **GPU kernel 2**: parfor each x, y and t ($N_\text{x} \cdot N_\text{y} \cdot N_\text{t}$):
     - Compute and store $h(t; x, y, \beta)$ in (9).
  3) **GPU kernel 3**: parfor each x, y and relevant z ($N_\text{x} \cdot N_\text{y} \cdot 3N_\text{t}$):
     - Compute accumulation (10) array along z.

After computing kernel 1, we loop over projection views having the same angle β in kernels 2 and 3, as in forward projection, to avoid redundant computation. The global memory used in our test case is about 640 MB for the image, 2 MB for projection views, 64 MB for $h(t; x, y, \beta)$ and 10 MB for F_1', F_2, u, v and other variables. The total global memory is still less than 1 GB. The global memory accesses are about $12N_\text{x}N_\text{y}$ in kernel 1, $N_\text{t}N_\text{x}N_\text{y}$ in kernel 2 and $3N_\text{x}N_\text{y}N_\text{t}$ in kernel 3. Shared memories are used in kernel 2 to store F_1', s_min, and s_max and the step size along z direction in kernel 3. Because projection data could be read up to $3N_\text{x}$ times in kernel 2, we used the GPU texture memory to store the projection view, so that kernel 2 reads them from cached texture memory. This may save much computation time compared to using global memory.

TABLE II
FORWARD AND BACK PROJECTION COMPUTATION TIME OF 24-THREAD CPU VERSION AND OF GPU VERSIONS FOR 8 HELICAL TURNS.

	CPU	single GPU	dual GPU	quad GPU
Forward projection	145 s	52 s	45 s	45 s
Back projection	156 s	114 s	71 s	50 s

C. Using multiple GPUs

Using multiple GPUs in parallel is another acceleration method. Our sever has 4 Tesla GPUs. Ideally, the computational time would be reduced by the number of GPU devices. We used POSIX threads to implement multiple CPU threads, each of which is assigned to control one GPU device. We used different strategies to distribute the computation for forward and backward projection in SF methods. For forward projection, we divided the projection views by projection angle. For back projection, we divided the image into several parts along x or y direction. Table II summarizes the computation times. Although multiple GPUs provided some acceleration, the reduction is less than ideal, particularly for more than two GPUs, presumably due to memory bandwidth limits. More investigation is needed for this part.

IV. Summary

We presented GPU accelerated implementations of both the forward and backward projection algorithms for the separable footprints method. The dual GPU version runs 2-3 times faster than a 24-thread CPU version on a 12-core Nehalem system, due to the GPU's fast massively parallel computational ability and quick memory access times. Further improvement of the multi-GPU version is needed. Some of the ideas developed for the GPU implementation may also benefit a revised CPU version and we plan to investigate that next to better understand the relative benefits of CPUs and GPUs for iterative reconstruction in computed tomography. Either choice of hardware will involve substantial parallelism and adapting the algorithms to such architectures is important.

Acknowledgment

The authors would like to thank Yong Long for her availability to answer questions, and Ryan James, Matt Lauer and Esther Wei for their former work on this project.

References

[1] Y. Long, J. A. Fessler, and J. M. Balter, "3D forward and back-projection for X-ray CT using separable footprints," *IEEE Trans. Med. Imag.*, vol. 29, no. 11, pp. 1839–50, Nov. 2010. [Online]. Available: http://dx.doi.org/10.1109/TMI.2010.2050898
[2] K. Mueller, F. Xu, and N. Neophytou, "Why do commodity graphics hardware boards (GPUs) work so well for acceleration of computed tomography?" in *SPIE Electronic Imaging*, vol. 6498, 2007, p. 64980N. [Online]. Available: http://dx.doi.org/10.1117/12.716797
[3] D. B. Kirk and W. W. Hwu, *Programming massively parallel processors: A hands-on approach.* Morgan Kaufmann, 2010. [Online]. Available: http://www.elsevierdirect.com/companion.jsp?ISBN=9780123814722
[4] J. Sanders and E. Kandrot, *CUDA by example: An introduction to general-purpose GPU programming*. Addison-Wesley, 2010.