Efficiently GPU-Accelerating Long Kernel Convolutions in 3-D DIRECT TOF PET Reconstruction via Memory Cache Optimization

Sungsoo Ha, Samuel Matej and Klaus Mueller

***Abstract*--The DIRECT represents a novel approach for 3-D Time-of-Flight (TOF) PET reconstruction. Its novelty stems from the fact that it performs all iterative predictor-corrector operations directly in image space. The projection operations now amount to convolutions in image space, using long TOF (resolution) kernels. While for spatially invariant kernels the computational complexity can be algorithmically overcome by replacing spatial convolution with multiplication in Fourier space, spatially variant kernels cannot use this shortcut. Therefore in this paper, we describe a GPU-accelerated approach for this task. However, the intricate parallel architecture of GPUs poses its own challenges, and careful memory and thread management is the key to obtaining optimal results. As convolution is mainly memory-bound we focus on the former, proposing two types of memory caching schemes that warrant best cache memory re-use by the parallel threads. In contrast to our previous two-stage algorithm [1], the schemes presented here are both single-stage which is more accurate.**

***Index Terms*—CUDA, DIRECT, GPU, TOF**

I. Introduction

DIRECT (Direct Image Reconstruction for TOF) [2] is an approach for TOF reconstruction that is a more efficient alternative to traditional list-mode and binned TOF PET reconstruction approaches [3]. In these latter approaches, the events are binned by their LOR (Line of Response) and arrival time to form a set of *histo-projections*, one for each angular view. In DIRECT, on the other hand, the events are first sorted into a (sub)set of angular views and then deposited for each view into a dedicated *histo-image*, each having the same lattice configuration and the same resolution as the reconstructed image. Here, each corrective update involves simple 3D convolutions using the system response (SR) kernel, which can be performed efficiently in Fourier space when the SR kernel is spatially invariant. However, in practical applications the SR kernel is not spatially invariant – its width increases up-to 40% towards the edge of the scanner. This prohibits the use of efficient Fourier-space methods to accelerate the convolution operations. Since the SR kernel is typically quite large (having FWHM 45mm in TOF direction for 300ps resolution and 5 - 6mm in LOR direction) a spatial convolution within a 144×144×62 matrix and 120 views can be prohibitively expensive for clinical application. We seek to overcome this challenge by GPU-acceleration [4][5], using their massively parallel computations to meet this challenge.

In general, mapping a CPU-based algorithm to the GPU and achieving 1-2 orders of speed-up is typically not straightforward. An especially critical component in GPUs is the memory, which is organized into a hierarchy, with some of it on-chip but the majority of it off-chip (but on-board). The former is orders of magnitudes faster. As it takes 100s of clock cycles to bring off-chip data into on-chip memory, it is of utmost importance to re-use these data among the parallel threads as much as possible. Also, since on-chip memory is quite small, on the order of KB, careful occupancy planning of this limited resource is equally important.

Since the long DIRECT kernels may traverse the image space at arbitrary angles, data access at these off-axis orientations is non-sequential. In [1] we presented a two-stage scheme that, for these off-axis directions, first resampled the data into a storage pattern aligned with the convolution direction of the long kernels. This allowed linear access in on-chip (shared) memory. By subtracting the smoothing effects of the interpolation (sampling) kernel from the convolution kernel, we were able to mitigate the blurring effects of the interpolation kernel into the convolution. This 2-stage scheme was about 10 times faster than the 1-stage scheme and only a small amount of artifacts could be observed.

In the current paper, we chose to go a different route with fewer artifacts, if any, using a 1-stage method that does not require resampling. Here we aimed for a method that loads the data into on-chip memory in such a way that it allows linear access at any angle, using a dedicated addressing scheme. For this, we investigated two types of on-chip memory: (i) shared memory and (ii) texture memory cache.

Our paper is organized as follows. Section 2 discusses relevant GPU details, Section 3 outlines our scheme, and Section 4 presents results and conclusions.

Sungsoo Ha and Klaus Mueller are with the Center for Visual Computing, Computer Science Department, Stony Brook University, NY 11794 USA (e-mail: {sunha, mueller}@cs.sunysb.edu).

Samuel Matej is with the Department of Radiology, University of Pennsylvania, Philadelphia, PA 19104 USA (e-mail: matej@mail.med.upenn.edu).

II. Some Notes on GPU Architecture

We have implemented all methods on a NVIDIA 480 GTX GPU with 1.5GB off-chip memory. This GPU has 480 CUDA cores organized into 15 Streaming Multiprocessors (SM) of 32 processors each. Important for our purposes are the size and access of the memory. Each SM has 64KB of on-chip memory that can be configured as 48KB of shared memory with 16KB of L1 cache, or as 16KB of shared memory with 48KB of L1 cache. Each SM also has a dedicated 768KB L2 texture cache. An important texture memory feature is that it is specifically designed to allow fast 2D memory access. It achieves this by storing 2D data along a space-filling curve which greatly improves access locality. All other GPU memories simply use linear addressing and so do not support fast 2D access.

With regards to shared memory bandwidth, a somewhat theoretical estimate is to assume that every clock-tick (the GTX 480 operates at 1,401 MHz) produces 16 (a half-warp) 4-byte data (from 32 banks) per SM and then multiply this number by 15 (the total number of SMs). This yields 1.344TB/s. Less detailed information is available to make such estimates for the texture cache. The texture fill rate is rated as 34 (bi-linearly interpolated) Gtextels/s, but this number likely includes off-chip memory as well. Nevertheless, there are many factors that can limit memory bandwidth in practice, both for shared and for texture memory, such as bank conflicts and the aforementioned optimized 2D access for texture memory and possibly cache. We therefore chose to implement and optimize for the application at hand methods that use shared memory as well as texture memory.

III. Methods

Fig.1 shows the pipeline of our scheme for forward- and backward-projection in DIRECT TOF PET. For each iteration step from a given view direction, there are four stages: (i) create the 2D projection mask; (ii) create the 3D point cloud, (iii) build the look-up table, and (iv) load the data and perform the forward or backward projection.

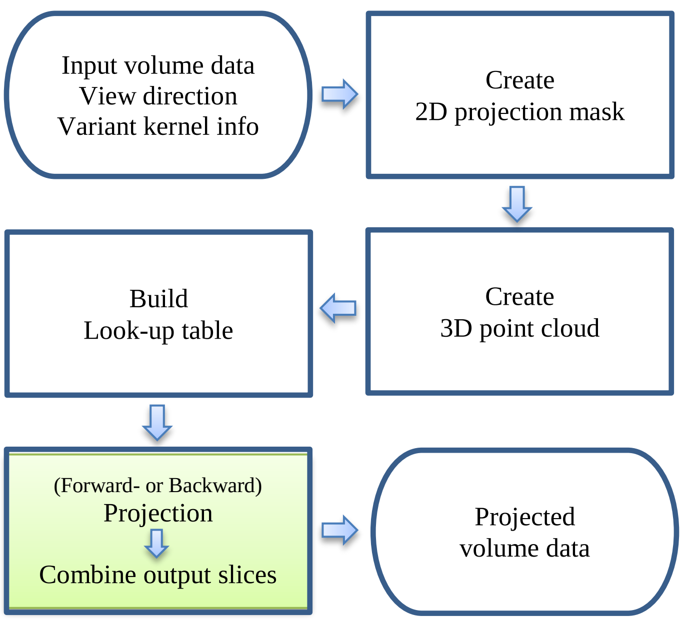


Fig. 1. Pipeline for proposed forward- and backward-projection scheme.

The *projection mask* is used to index and retrieve, from global memory, the list of voxels located along an approximate linear path that aligns with the long axis of the convolution kernel at the current view direction (Fig. 2a). The width of the kernel (LOR) determines the thickness of the path of retrieved voxels (Fig. 2b), which are then stored into on-chip memory. The computation targets are the voxels connected by the line in Fig. 2a. Each such voxel becomes a parallel thread, processing the data within the kernel's *point cloud* (Fig. 2b) and weighing each by the Gaussian kernel represented by the *lookup table*. We do this for every slice in parallel and also compute the z-extent of the 3D-kernel in parallel as well, which we then accumulate in a second step (Fig. 2c). Since we can store data for only 1-2 such lines at a time, due to limitations of the memory, we end up, in our case, with $62 \cdot 3=186$ CUDA blocks, with 168 threads each.

We note that both point cloud and lookup table are simple masks and do not require interpolation. Therefore, when stored in 1D texture memory, they can be retrieved with fast 1D texture fetches during the processing. In the following we motivate and describe each of these components in detail.

A. Create 2D Projection Mask

As mentioned, for general view directions, it is typical to have poor memory locality for forward- and backward-projections. Moreover, computing variant kernel resolutions along LOR directions will increase the overall computational burden exponentially. To solve these problems, we introduce a 2D projection mask of the same size as the input volume slice at each view direction. The mask consists of several lists based on the LOR distance so that elements in each list can share the same radial resolution with less error. The error is determined by the distance between lists. The distance is chosen to have less than or equal to 1 pixel distance within elements in a list. For example, for 0 and 90 degrees, the distance between elements is 1 pixel, which is the typically the grid size. To improve memory locality, the elements are sorted in the TOF distance ascending order.

B. Create 3D Cloud for variant kernel

The variant kernel for DIRECT TOF PET has non-symmetric (in radial direction), ellipsoidal shape. Constructing each kernel for a position will increase the overall computational burden exponentially. To solve this problem, we introduce a 3D cloud, which can fit in all possible non-symmetric, ellipsoidal, variant kernels. The value for a point within a cloud is determined by the distance from the center of the cloud to the point in TOF, LOR, and z-axis directions. The distance can be pre-computed and used as index to fetch a corresponding look up table. The cloud points are sorted in the TOF distance ascending order, layer by layer, to minimize the number of access to volume data in projection stage.

C. Build Look-up Table for variant kernel

There are three look-up tables (LUTs) according to the TOF, LOR, and z-axis directions. Since we know the exact TOF, LOR and z distance from the created cloud points, we only need a finite number of kernel values in each direction. In the TOF- and z-axis directions, their resolutions are invariant and

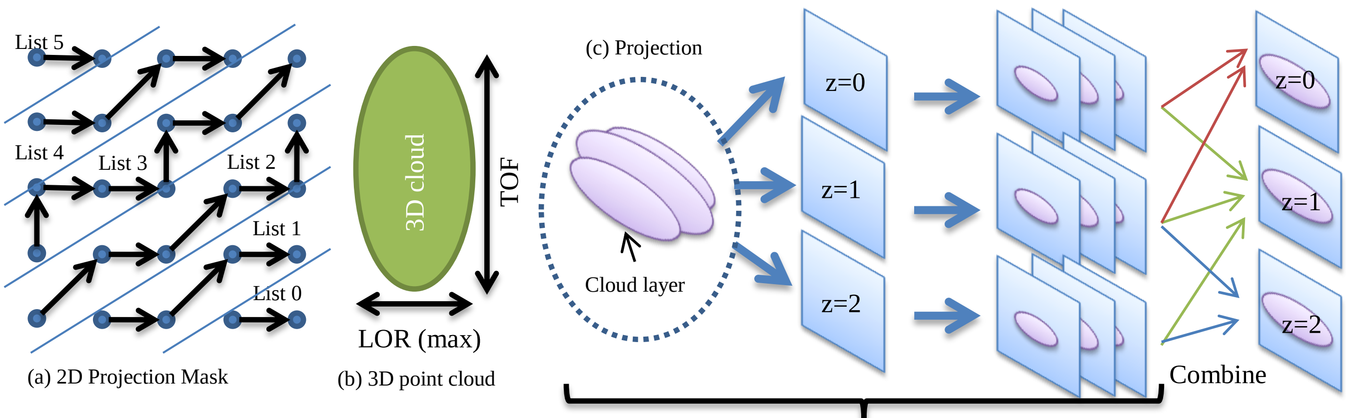


Fig. 2. Three critical stages

thus their kernel values can be stored in a 1D array. The length of the array is determined by the number of cloud points. For the LOR direction, there are as many different resolutions as the number of lists in a projection mask. Thus, the kernel values for radial direction can be stored in a 2D array having same length with TOF (and z-axis) LUT.

D. Projection

To improve memory locality, (forward- and backward-) projection for a voxel proceeds list by list in a projection mask. Also, to minimize the number of accesses to volume data, the projection proceeds cloud layer by layer and its results are stored in different places; after completing the projection, the results are combined into a volume, which have the same size than the input volume (Fig. 2c).

IV. IMPLEMENTATION

As mentioned, we have explored our method with regards to the on-chip cache/memory used: shared memory and texture cache. The former allows for better user management.

Fig. 3a shows the pseudo code for the shared memory implementation. The advantage of using shared memory is that we can manipulate caches of very low latency, as low as registers. This allows us to achieve minimum access to the input volume data by fetching the data first and then updating the data after completing the projection for a list. Moreover, accessing volume data stored in shared memory can be as fast as accessing to registers as long as there are no bank conflicts. To maximize the ability of the shared memory, in our scheme each CUDA block takes a volume slice and a cloud layer, and all threads in a block run several lists and its elements in parallel but the projection for each such element proceeds in sequence (along the line) to avoid bank conflicts. Because we store the list elements in TOF distance-ascending order, all threads are guaranteed to access different banks in shared memory.

Fig. 3b shows the pseudo code for the texture memory implementation. The advantage of using texture cache is that we are free from indexing problems when fetching volume data into texture memory. Also, we can use the fastest cache (shared memory) for other purposes. It allows this method to have a higher degree of parallelism, on the granularity of cloud points. Each thread computes one cloud point multiplication, which are then summed via fast parallel reduction to achieve the projection result for a target voxel along the line.

There is a difference in computational overhead for forward and backward projection. In the forward projection, the kernel resolution in the LOR direction needs to be computed for each cloud point. We can do this by moving the 'compute radial

```
Run each block (take a volume slice)
  Read volume data from global memory &
  Store in shared memory
  __sync()
  Run lists do projection
    Run list elements
    Compute radial resolution
    Set output to zero
    For cloud points // to avoid bank conflicts
      Fetch look up table from texture memory
      Compute shared memory index
      Read a volume point from shared memory
      Accumulate output
    End cloud points
    Write output to global memory
  End list elements
  __sync()
  Update shared memory for next lists
  __sync()
  End lists
```

(a) Method 1: Using shared memory

```
Run each block (take a list)
  Run list elements
    Compute radial resolution
    Set output to zero
    Run cloud points
      Fetch look up table from texture memory
      Read volume data from texture memory
      Accumulate output
    End cloud points
    Write output to shared memory
    __sync()
    Parallel sum reduction
    __sync()
  Write output to global memory
  End list elements
End block
```

(b) Method 2: Using texture cache

Fig. 3. Pseudo code for two different implementation methods in projection. 'Run' is for parallel loop. 'For' is for sequential loop.

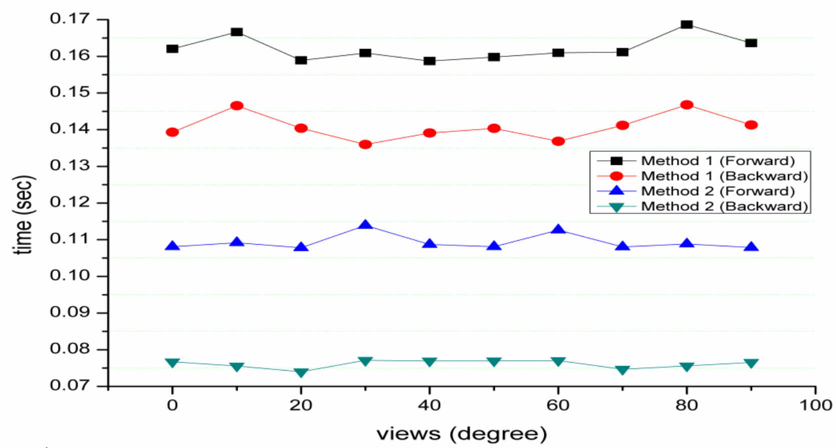


Fig. 4. Time performance comparison

resolution' line inside the cloud point loop in Fig.3. Because of this, it is obvious that forward projection is slower than backward projection in DIRECT TOF PET.

Table 1. Two different methods for projection stage

Method 1	Method 2
Shared memory	Texture cache
Need to compute index	No need to compute index
Partial parallelized	Fully parallelized

V. Results and Conclusion

As mentioned in Section 2, we tested our algorithm on a NVIDIA GTX 480 GPU. The volume size of the test was 144× 144×62. The projection kernel was 45mm along the TOF direction, 5-6mm along the radial direction, and 5mm along the *z*-direction of the Gaussian FWHM (the corresponding cloud size has at most 29×5×3 voxels). The most time-consuming stage is the projection stage. All other stages took about 0.00003 seconds each, which is a trivial time compared to the time for the projection stage.

In our experiments the forward projection was slower than backward projection by factors of about 0.9 and 0.7 for method 1 and method 2, respectively. Further, method 2 was faster than method 1 by factors of about 1.5 and 1.8 for forward projection and backward-projection, respectively.

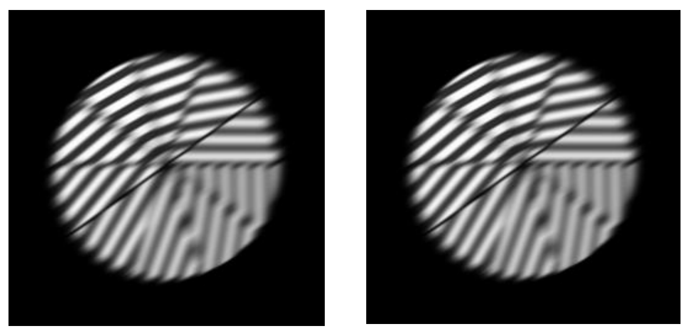
In our test case, the number of threads in the projection stage was about 23,000 for method 1 and 6,856,000 for method 2. Thus, there were about 300 times as many threads for the latter due to its better memory cache usage. In addition, the native 2D indexing of the texture caches alleviated the need for extra index computing to fetch the volume data. All of these aspects combined favors method 2 over method 1.

If we run a DIRECT reconstruction with 120 views, a 144× 144×62 volume, method 2 will require approximately 24 seconds per iteration, while the FFT based approach on a 2.8GHz Dell Precision T5500 single processor takes about 40 seconds per iteration for the considered 62 slices (31 seconds for the 48 slices reported in [6]). It is important to note, however, that the FFT based approach assumes a spatially invariant kernel, while our GPU implementation can widen the kernel width towards the edges of the detector and hence is spatially variant which prohibits the log(n) acceleration of FFTs.

In current work, we are working on expanding our algorithm to also incorporate the tilt case and we are incorporating the presented implementations into the DIRECT reconstruction framework.

Table 2. Time performance

Time [sec]	Forward	Backward
Method 1	0.16213	0.14077
Method 2	0.10932	0.07612



(a) Forward (b) Backward

Fig. 5. Selected sample: projected volume slice at 30 degrees view

References

[1] S. Ha, Z. Zhang, S. Matej, K. Mueller, "Efficiently GPU-Accelerating Long Kernel Convolutions in 3-D DIRECT TOF PET Reconstruction via a Kernel Decomposition Scheme," IEEE Medical Imaging Conference, Knoxville, TN, October, 2010.

[2] S. Matej, S. Surti, S. Jayanthi. M. Daube-Witherspoon. R. Lewitt, J. Karp, "Efficient 3-D TOF PET reconstruction using view-grouped histo-image: DIRECT-direct image reconstruction for TOF," *IEEE Trans Med Imaging*, 28(5):739-51, 2009.

[3] L. Popescu, S. Matej, R. Lewitt, "Iterative image reconstruction using geometrically ordered subsets with list-mode data," *IEEE Medical Imaging Conference*, M9-211, pp. 3536-3540, 2004.

[4] F. Xu, K. Mueller, "Accelerating popular tomographic reconstruction algorithms on commodity PC graphics hardware." *IEEE Trans. On Nuclear Science*, 52(3):654-663, 2005

[5] F. Xu, K. Mueller, "Real-Time 3D Computed Tomographic Reconstruction Using Commodity Graphics Hardware," *Physics in Medicine and Biology*, 52:3405-3419, 2007.

[6] M. E. Daube-Witherspoon, S. Matej, M. E. Werner, S. Surti, and J. S. Karp, "Comparison of listmode and DIRECT approaches for time-of-flight PET reconstruction," *IEEE Nuclear Science Symposium and Medical Imaging Conference*, M09-256, Knoxville, TN, 2010