GPU-Based spatially variant SR kernel modeling and projections in 3D DIRECT TOF PET Reconstruction

S. Ha[a], M. Ispiryan[b], S. Matej[b] and K. Mueller[a]

[a]*The Center for Visual Computing, Computer Science Department, Stony Brook University, NY, USA*
[b]*Medical Image Processing Group, Radiology Department, University of Pennsylvania, PA, USA*

Abstract

In this study, we develop GPU-based application to model spatially variant system response (SR) kernels and to perform forward- and back-projection operations with the SR kernels, for DIRECT iterative reconstruction approach. Modeling the spatially variant SR kernels is efficiently done and handled by introducing three kinds of look-up tables (LUTs) and storing them in different kinds of GPU memory according to their access pattern. We explore the thread-level (TLP) and instruction-level (ILP) parallelism in GPU implementation for the projection operators to efficiently hide the long latency and, eventually, to obtain the best time performance. Our GPU implementation of the space-based projection operations reveals slightly faster or approximately comparable time performance to the same (Fourier-based) operations via the FFT-based approaches using the state-of-the-art FFTW routines. Furthermore, the developed space-based GPU tools can efficiently handle any generic SR kernels such as spatially symmetric and variant, or asymmetric and variant kernels that the FFT-based approach cannot handle.

1. Introduction

The Time-Of-Flight (TOF) information has been utilized for positron emission tomography (PET) imaging. The TOF makes it possible for a point of origination of annihilation (or an event) to be more accurately predicted than conventional PET imaging. The improved localization reduces noise in imaging data, resulting in higher image quality, shorter imaging times, and lower does to the patient [1-4].

Full realization of the TOF information requires proper reconstruction tools. The DIRECT (Direct Image Reconstruction for TOF) [5] is an approach for TOF reconstruction that is more efficient alternative to traditional list-mode and binned TOF-PET reconstruction approaches [6][7]. In the binned approaches, the events are binned by their Line-Of-Response (LOR) and arrival time to form a set of *histo-projection*, one for each angular view. On the other hand, in the DIRECT approach 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 voxel resolution as the reconstructed image.

As in any iterative reconstruction algorithm, the DIRECT requires convolution-like operations (forward- and back-projection) in each corrective update using the system response (SR) kernels modeling scanner's timing (TOF) and detector (LOR) resolution functions. These operations can be performed efficiently in Fourier space when the SR kernels are spatially invariant. However, in practical applications the SR kernels are not spatially invariant but their width increases up to 40% towards the edge of the scanner. Moreover, the SR kernels are typically quite large in size, for example, 45-90 mm FWHM in TOF direction for 300-600 ps TOF resolution and 5-10 mm FWHM of LOR axial and radial directions [8][9]. Modeling spatially variant and quite large size of the SR kernels makes it difficult to implement the projection operations not only in Fourier domain but also in spatial domain with maintaining the computational efficiency, comparable to the Fourier-based approach. We have seek to overcome this challenge by GPU-acceleration [10][11], using their massively parallel computations.

In general, mapping a CPU-based algorithm to the GPU and achieving 1-2 orders of speed-up is typically not straightforward. The CPU-based algorithm often needs to be reordered or decomposed to fit in the GPU architecture and its programming model. Especially the critical component in GPU architecture 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 SR kernel, having much wider width in TOF direction than others, may traverse the histo-image space at arbitrary angle, the data access at these off-axis orientations is non-sequential. To achieve maximum utilization of on-chip GPU memory for these off-axis directions, we proposed a two-stage scheme [12]. It first resampled the data into a storage pattern aligned with the TOF direction of the SR kernels for linear access in on-chip memory. By subtracting the smoothing effects of the interpolation (sampling) kernel from the SR kernel, we were able to mitigate the blurring effects of the interpolation kernel into the SR kernel. Even if it showed good time performance, there were two limitations to apply this scheme into DIRECT framework. The first one was that the interpolation kernel had often wider than the actual detector resolution kernel. The second one was that this scheme could be only applied for symmetric SR kernels, while accurate SR kernels had some degree of asymmetry at large radii.

In this study, we chose to go a different route with fewer artifacts, if any, using a one-stage method which does not require the interpolation kernel. 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 view angle, using a dedicated addressing scheme. For this, we focused on utilizing texture memory, which is specifically designed to allow fast 2-d memory access by storing 2-d data along a space-filling curve that greatly improves access locality [13], and its cache. Along with the linear access, we explored two kinds of parallelisms, Thread-Level (TLP) and Instruction-Level (ILP) Parallelism [14], in GPU to efficiently hide long latency and to find optimal parallelism for the convolution-like operations in DIRECT framework.

The organization of this paper is as follows. In section 2, we give some more details about the SR kernel and the convolution-like operations in DIRECT. Section 3 covers some background of the GPU architecture and programming model used in this study, and two parallelisms (TLP and ILP) in the GPU. In section 4, we describe the methodology including all technical details. Experimental results are presented in section 5 and 6. We then conclude the paper in section 7.

2. Projections in DIRECT

2.1. The SR kernel in DIRECT

The characteristic of the SR kernels is determined by TOF-PET detector. In this study, the full-ring TOF-PET detector developed from the University of Pennsylvania is used [8][9]. It has 45-90 mm FWHM in TOF direction for 300-600 ps TOF resolution. This indicates the probability (as determined by the TOF measurement) an event occurs at a voxel in histo-image space in DIRECT. Furthermore, the event can be observed at multiple detector modules because of the geometry of the full-ring TOF-PET detector. The detector (LOR) blurring takes this phenomenon into account as 5-10 mm FWHM of LOR axial and radial directions. In this study, the variation in LOR axial direction is small enough to disregard over the detector. As a result, the SR kernels in DIRECT have ellipsoidal-like shape having wider width in TOF direction and are varying only in LOR radial direction.

2.2. Forward- and back-projection in DIRECT

As in any iterative reconstruction algorithm, in DIRECT, the forward-projection is performed to calculate new projection (or simulated projections) by projecting an image into histo-image space with given SR kernels. The difference between the simulated projection data and the measured data obtained from scanner is back-projected into image space to reconstruct a corrected image. This is back-projection. It is worth noting that histo-image space and image space have the same lattice configuration and the same voxel resolution in DIRECT.

In DIRECT, the use of TOF information allows to more precisely localizing an event in histo-image space (or in image space) for forward- and back-projection. Due to the localization of events in the histo-image space, projection operations become convolution-like operation with given SR kernels [7]. The forward projection can be interpreted

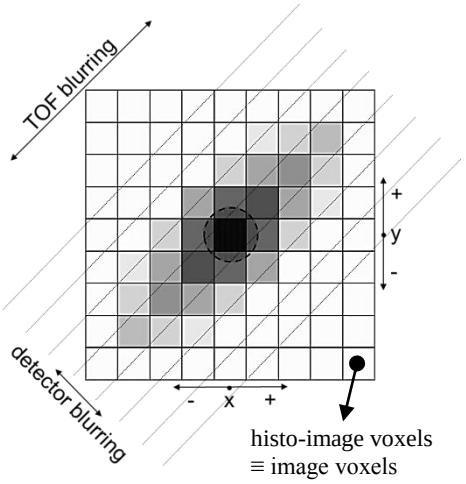


Fig. 1. TOF projection with 45 degrees of azimuthal angle. A point source is located at the centre in the histo-image and the SR kernel for the point source has symmetric resolution for TOF and detector (LOR) blurring.

as a scattering operation. In other words, each image voxel *spreads* its value to its neighbors, weighted by that voxel's SR kernel. In contrast, in the back projection, which can be described as a gathering operation, each voxel *collects* values from its neighbors, weighted by the neighbors' SR kernels. With {symmetric, invariant}[a] SR kernel, both projections are operated exactly same as convolution operation, which can be handled efficiently by means of FFT approach [7]. Fig. 1 shows this case when a point source is located at the center in the histo-image space and projection is performed for the point source with ellipsoidal-like shape of the symmetric SR kernel. In case of {asymmetric, variant} SR kernel, both projections must be strictly distinguished and handled in spatial domain because the FFT approach cannot handle such generic system resolution modeling and also two different projection operations will cause different results. The detailed comparisons and explanations are discussed in section 5.2.

3. Overview of NVIDIA GPU architecture and its programming model, CUDA

We have accelerated the (forward- and back-) projections on a NVIDIA GTX 285 GPU with 1GB DDR3 off-chip memory. This GPU has 240 CUDA cores organized into 30 streaming multiprocessors (SM) of 8 scalar processors (SP) each. Groups of SMs belong to Thread Processing Clusters (TPC). This GPU, like all modern GPUs, has off-chip memory include global, texture and constant memory and incurs hundreds of cycles of memory latency. It is often the bottleneck of a GPU application. However, texture and constant memory can be cached, replacing the hundreds of cycles of latency with only a few cycles for on-chip cache access. The CUDA (Computer Unified Device Architecture) is a C-like API used to program the NVIDIA GPUs. Execution of a CUDA kernel (or function) invokes multiple threads which are organized into thread blocks in a grid. The GTX 285 can have a maximum of 512 threads per block. Some important parameters for GTX 285 architecture and its programming model, CUDA, are listed in Table I [15][16].

Each SM containing 8 SPs uses a 24-stagte and in-order SIMD pipeline without forwarding [15]. This implies that at least 192 active threads are needed to avoid stalling for true data dependencies between consecutive instructions from a single thread. All threads in a SM are scheduled to the SIMD pipeline with a unit of warp, a collection of 32 threads, and a warp executes the same instruction with different data values over four consecutive clock cycles in all pipelines with zero overhead scheduling on a fine-grained basis [17]. To be more specific, the warp scheduler selects warps ready for execution in every four cycles and issues them to the SIMD pipelines in a loose round robin fashion that skips non-ready warps (for example, those waiting on global memory accesses). The more threads running on each SM thus give more chances to hide long latency and this tolerance to long latency allow GPUs to dedicate more of their chip area to floating-point execution resources unlike CPUs [18]. In the following subsections, we will discuss about thread occupancy and efficiency, and thread-level parallelism and

[a] For simplicity, we use the notation {a, b} to refer 'a and b', and {a; b} is for 'a or b' in this paper.

TABLE I. NVIDIA GTX 285 GPU PARAMETERS

GPU Organization	
TPCs (Thread Processing Cluster)	10 total
SMs (Streaming Multiprocessor)	3 per TPC
Shader Clock	1.48 GHz
Memory (DRAM) Clock	1.24 GHz
Memory (DRAM) Bus Width	512-bit
Memory (DRAM) Latency	400 – 600 cycles
SM Resources	
SPs (Scalar Processor)	8 per SM
SFUs (Special Function Unit)	2 per SM
DPUs (Double Precision Unit)	1 per SM
Registers	16,384 per SM
Shared Memory	16 KB per SM
Constant Cache	8 KB per SM
Texture Cache	6-8 KB per SM
Programming Model	
Warps	32 threads
Max number of threads per block	512 threads
Max sizes of each dimension of a block	512 x 512 x 64
Max sizes of each dimension of a grid	65535 x 65535 x 1
Global Memory	1 GB total
Constant Memory	64 KB total

instruction-level parallelism, in CUDA. Since they are strongly related to the usage of GPU resources and eventually give large impacts on the performance of GPU applications, it is important to have deep understanding on them.

3.1. Thread occupancy and efficiency

There are two important notions to understand GPU parallelism with CUDA: *thread occupancy* and *thread efficiency*. The *thread occupancy* is defined as the number of active threads per thread block as a percentage of the device full capacity; while the *thread efficiency* is defined as the overall computational efficiency of the individual threads [19]. Running more threads (or warps) in a thread block refers to higher thread occupancy and this helps to hide long latency thanks to fast context switching (zero-overhead scheduling) among warps; in other words, higher thread occupancy implies it will heavily rely on the off-chip memory, which has the hundreds of cycles of latency, since there are very limited amount of on-chip resources and they are allocated for the entire block all at once. On the other hand, if a thread block fully utilizes the on-chip resources to obtain higher thread efficiency, the total number of threads in the thread block will be very restricted. Therefore, it is important to finely tune the amount of on-chip resources allocated to each thread by optimizing the trade-off between thread occupancy and efficiency.

3.2. Two forms of parallelism on CUDA: TLP and ILP

There are two forms of parallelism on CUDA as in contemporary computer system (CPU): thread-level (TLP) and instruction-level (ILP) parallelism. Both forms identically identify independent instructions but in different granularities of parallelism [14]. As the CUDA models the GPU architecture as a multi-core system, it abstracts the TLP of the GPU into a hierarchy of threads [15]. The more threads in a thread block have more TLP and it means higher thread occupancy. In CUDA, the (static) ILP can be achieved by increasing independent instructions for a thread. For example, unrolling a loop in a CUDA function can be used to increase the ILP. It is obvious that

increasing ILP will cause more usages of on-chip memory and thus higher thread efficiency (or lower thread occupancy). Thus TLP and ILP should be finely tuned to obtain optimal parallelism for a CUDA function.

Providing optimal parallelism for a CUDA function means to provide proper number of operations per SM *wisely*. In this study, the term '*wisely*' refers to choose ways to hide long latency by adjusting between TLP and ILP. Assuming the maximum throughput, the needed parallelism per SM for a CUDA function can be approximately computed as following [20]:

$$Needed\ Parallelism\left[\frac{\#\ Operations}{SM}\right] = Latency\ [Clock\ Cycles] \times Throughput\ \left[\frac{\#\ SPs}{SM}\right]$$

where *latency* refers to the time required to perform an operation and it varies between different types of operations; *throughput* is the number of operations completed per cycle which is equal to the number of SPs per SM. In more precise, for example, to run an operation having *x* latency in a SM containing 8 SPs *wisely*, the operation should be executed by *8x* threads per thread block (TLP) or *8x* times of the operation should be executed by a thread (ILP) or some combinations of two strategies, TLP and ILP. Since, in practice, estimating TLP and ILP for a CUDA function based on measured latencies of a GPU device is not trivial problems and a little out of scope in this study. Instead, we explore two forms of parallelism experimentally to find optimal cross-point and the results are evaluated with time performance and corresponding thread occupancy in section 5.1.

4. Methodology

For each view angle, creating a SR kernel of a voxel based on the voxel's spatial location in histo-image space and applying projection operations to each voxel with its own SR kernel are computationally very expensive obviously. To handle this efficiently, the projection operations are followed by constructing three look-up tables (LUTs). In the following two subsections, we will discuss about those LUTs first and then a CUDA function for the projection operations.

4.1. Three look-up tables (LUTs)

The first LUT we are going to construct is to handle spatially variant SR kernels efficiently. Instead of computing the SR kernel resolutions for each voxel on-the-flight, they can be pre-computed and stored in memory before performing projection operations. In DIRECT, as discussed in section 2.1, only LOR resolution in radial direction is spatially varying. This makes it possible to build the LUT in 2-d structure having same size of width and height as image data and compute it only for a slice of the data (traverse view) in both, non-tilt and tilt cases. More specifically, the LOR resolution for a voxel is determined by the azimuthal angle at a given view and LOR distance, which is the distance from the center of the scanner to the voxel in LOR radial direction. A unique identifier, *a*, is assigned to the voxels within the range of $\{a \in Z \mid a \cdot \epsilon \leq LOR\ distance < (a+1) \cdot \epsilon\}$ with pre-defined epsilon value, ϵ. The total number of identifiers (N_{kernel}) is equal to the total number of spatially variant SR kernels, which will be used in projection operations. The identifier will be used to index spatially variant SR kernels for each voxel as well as to discretize the continuously varying LOR resolutions in radial direction. In this study, the epsilon value is set to 2 mm for 4 mm^3 voxels and the LUT is named as '*LORmap*'.

It is another time consuming task building a SR kernel on-the-flight for each voxel based on its pre-computed resolutions. Moreover, it is not trivial to perform projection operations with different size of SR kernels and this would be getting harder when it comes to {asymmetric, variant} SR kernels. To resolve these problems, ellipsoid shape of a kernel is modeled with maximum width of TOF and LOR resolutions so that it can cover {(symmetric; asymmetric), (variant; invariant)} SR kernels. We call the kernel '*mSR*' and define it as following:

$$mSR = \{v(x,y,z) \mid V_{cut} \geq -(\frac{dist_{tof}^2}{\sigma_{tof}^2} + \frac{dist_{lorr}^2}{\sigma_{lorr}^2} + \frac{dist_{lora}^2}{\sigma_{lora}^2})\}$$

$$V_{cut} = -\left(\frac{\left(k_{tof}\sigma_{tof}\right)^2}{\sigma_{tof}^2} + \frac{(k_{lorr}\sigma_{lorr})^2}{\sigma_{lorr}^2} + \frac{(k_{lora}\sigma_{lora})^2}{\sigma_{lora}^2}\right) = -(k_{tof}^2 + k_{lorr}^2 + k_{lora}^2)$$

In the two equations, $dist_{dir}$ is the distance from the center of the *mSR* kernel to a voxel, $v(x,y,z)$, in dir direction, σ_{dir} is the standard deviation of the resolution in dir direction, and k_{dir} is a constant (≥ 1) in dir direction;

$tof, lorr, and\ lora$ are the directions to TOF, LOR in radial, and LOR in axial, respectively. In short, *mSR* is a LUT containing a collection of voxels' coordinates within the ellipsoid shape kernel bounded by V_{cut}. We set those constants as 3 to keep enough long tails in all directions. Introducing *mSR* has two advantages; first of all, it does not require the interpolation kernel anymore unlike our previous work [12] and secondly it makes it much easier to implement projection operations since *mSR* itself is always {symmetric, invariant}.

The values of the *mSR* are stored in separate LUTs according to the identifiers, which are used filling in the *LORmap*. In more detail, there are N_{kernel} number of LOR resolutions varying in radial direction and we can compute and store the kernel values corresponding to each LOR resolution (σ_{lorr}^{a}) separately. In this study, the Gaussian functions are used to model {symmetric, (variant; invariant)} SR kernels with the TOF and LOR resolutions, σ_{tof}, σ_{lorr}^{a}, and σ_{lora} as followings:

$$mSRval_a(v) = \frac{1}{\sum mSRval_a} \cdot Gau_{3d}(\sigma_{tof}, \sigma_{lorr}^{a}, \sigma_{lora}, dist_{tof}, dist_{lorr}, dist_{lora})$$

The $Gau_{3d}(\cdot)$ is the 3-d Gaussian function taking six parameters; the first three are for the standard deviation of the resolutions and the rest of them are the distance from the center of the *mSR* to the voxel *v*, in three directions: TOF, LOR in radial and axial directions. This can be efficiently computed using the separable property of the Gaussian function. We can model spatially {asymmetric, (variant; invariant)} SR kernels by using double Gaussians. Indeed, this approach allows us to utilize any arbitrary SR kernels as long as there are proper models for the SR kernels or well measured SR kernel data, if any; then, it is trivial to fill in the LUTs, $mSRval_a$.

4.2. Projection operations and optimizations

With the three kinds of LUTs at any arbitrary view, the projection operations with {(asymmetric; symmetric), (variant; invariant)} SR kernels become operations with spatially {symmetric, invariant} SR kernel and this conversion makes it very straightforward to implement the operations in any programming languages. In this section, we will discuss about how the projection operations are implemented and optimized with CUDA.

In the CUDA function of the back-projection, each thread performs the gathering operations for a voxel with a given *mSR* kernel; 1) it first computes the coordinate of the voxel and selects $mSRval_a$ by fetching *LORmap* and 2) collects neighbors' values of the voxel multiplied by the selected SR kernel values ($mSRval_a$) at the location of the voxel. In contrast to the back-projection, the forward-projection is scattering operations as described in section 2.2. It is important noting that, on the GPU, gathering operations are more efficient than scattering operations because of the fact that memory reads can be cached and are therefore faster than memory writes; moreover, gathering operations can avoid write hazards (or race condition) by writing data in an orderly fashion but scattering operations require slower atomic operations to avoid such write hazards [19]. For these reasons, the scattering operation of the forward-projection is converted to gathering operations by collecting neighbors' values of a voxel weighted by their SR kernels' values.

```
/* compute a voxel coordinates */
1  col = (blockIdx.x%n)*blockDim.x + threadIdx.x;
2  row = (blockIdx.x/n)*blockDim.y + threadIdx.y;
3  dep = blockIdx.y*blockDim.z + threadIdx.z;

/* fetch LORmap and store mSRval index */
4  LORid = tex2D(LORmap, col, row);

/* gathering operations */
5  FOR i=0 to sizeof(mSR)
6    srcx = col + mSRx[i];
7    srcy = row + mSRy[i];
8    srcz = dep + mSRz[i];
9    SRval = mSRval[LORid*sizeof(mSR)+i];
10   srcval = tex3D(image, srcx, srcy, srcz);
11   projval += srcval*SRval;
12 END
```

```
/* compute a voxel coordinates */
1  col = (blockIdx.x%n)*blockDim.x + threadIdx.x;
2  row = (blockIdx.x/n)*blockDim.y + threadIdx.y;
3  dep = blockIdx.y*blockDim.z + threadIdx.z;

/* gathering operations */
4  FOR i=0 to sizeof(mSR)
5    srcx = col + mSRx[i];
6    srcy = row + mSRy[i];
7    srcz = dep + mSRz[i];

     /* fetch LORmap and store mSRval index */
8    LORid = tex2D(LORmap, srcx, srcy);
9    SRval = mSRval[LORid*sizeof(mSR)+sizeof(mSR)-i];
10   srcval = tex3D(image, srcx, srcy, srcz);
11   projval += srcval*SRval;
12 END
```

Fig. 2. Pseudo CUDA code of the projection kernels only with the TLP. [left] back-projection [right] forward-projection. The scatter operations in forward-projection are converted to gathering operations.

4.2.1. Thread-Level Parallelism (TLP)

Fig. 2 shows the pseudo CUDA code of the projection kernels at an arbitrary view with only the TLP. In the projection CUDA code, each thread performs the projection operations for a voxel in image data. The three kinds of LUTs are stored in two different kinds of device memory according to the manner how each thread access to the LUTs. The *mSR* and *mSRval* are stored in (linear) global memory since each thread accesses to them in a coalesced manner within the for-loop (in gathering operations). In contrast, the accesses to *LORmap* for each thread are spreading out but well localized as the shape of *mSR*, and texture memory allows fast access with its cache to this kind of memory access pattern (also known as 2-d cloud assessing pattern). With the same reason, the image data is also mapped to the texture memory.

As the CUDA nature (SIMT), the TLP is achieved by invoking the CUDA functions with many threads, and higher TLP helps to more hiding the long latency of the memory accesses to the LUTs and image data. The higher TLP can be obtained with higher thread occupancy. In addition, a thread block should have wider width to x- and y-dimensions than z-dimension to take into account the fact that the texture memory is specifically designed for fast 2-d memory access [13]. In this study, we experimentally choose $16 \times 16 \times 2$ (or total 512 threads) as the configuration of a thread block for NVIDIA GTX 285.

4.2.2. Instruction-Level Parallelism (ILP)

In CUDA, the TLP hides the long latency by fast switching of thread context (zero-overhead scheduling) in the unit of warp, while the long latency can be hidden by executing multiple independent instructions in each thread and this is ILP. In more detail, for example, while a thread is waiting for long latency, it executes another independent instruction instead of switching its context. As the result, it will require more on-chip memory to store intermediate results and consequently cause low thread occupancy (but high thread efficiency).

The ILP is added to the previous projection CUDA code by assigning multiple voxels for the projection operations to each thread. Fig. 3 shows the pseudo CUDA code with ILP. In the code the multiple voxels are chosen along the axial direction (z-axis) to keep the 2-d cloud accessing pattern to all voxels for *LORmap* and image data. Also, the instructions for those voxels are unrolled to hide latency among them. In this study, we select 16 voxels for

```
/* compute a voxel coordinates */
1  col = (blockIdx.x%n)*blockDim.x + threadIdx.x;
2  row = (blockIdx.x/n)*blockDim.y + threadIdx.y;
3  dep = blockIdx.y*blockDim.z + threadIdx.z;

/* fetch LORmap and store mSRval index */
4  LORid = tex2D(LORmap, col, row);

/* gathering operations */
5  FOR i=0 to sizeof(mSR)
6   srcx = col + mSRx[i];
7   srcy = row + mSRy[i];
8   SRval = mSRval[LORid*sizeof(mSR)+i];
9   FOR j=0 to 15 [#pragma unroll 16]
10   srcz = dep*16 + j + mSRz[i];
11   srcval[j] = tex3D(image, srcx, srcy, srcz);
12  END
13  FOR j=0 to 15 [#pragma unroll 16]
14   projval[j] += srcval[j]*SRval
15  END
16 END

/* output projection value */
17 FOR j=0 to 15 [#pragma unroll 16]
18  out[dep*16+j][row][col] = projval;
19 END
```

```
/* compute a voxel coordinates */
1  col = (blockIdx.x%n)*blockDim.x + threadIdx.x;
2  row = (blockIdx.x/n)*blockDim.y + threadIdx.y;
3  dep = blockIdx.y*blockDim.z + threadIdx.z;

/* gathering operations */
4  FOR i=0 to sizeof(mSR)
5   srcx = col + mSRx[i];
6   srcy = row + mSRy[i];

    /* fetch LORmap and store mSRval index */
7   LORid = tex2D(LORmap, srcx, srcy);
8   SRval = mSRval[LORid*sizeof(mSR)+sizeof(mSR)-i];
9   FOR j=0 to 15 [#pragma unroll 16]
10   srcz = dep*16 + j + mSRz[i];
11   srcval[j] = tex3D(image, srcx, srcy, srcz);
12  END
13  FOR j=0 to 15 [#pragma unroll 16]
14   projval[j] += srcval[j]*SRval;
15  END
16 END

/* output projection value */
17 FOR j=0 to 15 [#pragma unroll 16]
18  out[dep*16+j][row][col] = projval;
19 END
```

Fig. 3. Pseudo CUDA code of the projection kernels with the ILP. [left] back-projection kernel. [right] forward-projection kernel. The scatter operations in forward-projection are converted to gathering operations.

TABLE II. THE PROJECTION CUDA KERNELS' ATTRIBUTES [1] AND TIME PERFORMANCE FOR A VIEW

	Constant memory [2] (Bytes)	Local memory per thread (Bytes)	Shared memory per block (Bytes)	# threads per block	# registers per thread	Thread occupancy [3] (%)	Time (sec)	
							FP	BP
TLP	36	0	64	512	14 (13) [4]	100	4.94	4.37
ILP	36	0	64	384	42	38	2.59	2.56

Note 1) The CUDA API, cudaFuncGetAttributes(), is used to extract those parameters.
Note 2) The size in bytes of user-allocated constant memory required by projection CUDA kernel.
Note 3) The thread occupancy is calculated using the thread occupancy calculator provided by NVIDIA with the extracted parameters.
Note 4) 14 is for forward-projection CUDA kernel and 13 is for back-projection CUDA kernel.

each thread and choose $16 \times 16 \times 1$ (or total 256 threads) as the configuration of a thread block, experimentally, for NVIDIA GTX 285.

5. Analysis of projection operations

To test the CUDA code for projection operations, we use the SR kernels, if it is not specified, with Time-Of-Flight (TOF) resolution 900 ps FWHM, Line-Of-Response (LOR) resolution in radial direction varying from 10 mm to 100 mm FWHM (from the center to the edge), and LOR axial resolution 10 mm FWHM. These widths and variability are much larger than what can be observed on the real scanners, but is purposefully chosen such to better test and illustrate performance of the CUDA-based projection projections with the spatially variant kernels. Test volume dimensions are $144 \times 144 \times 48$ with 4 mm^3 voxels. Also, constructing the three kinds of LUTs is also accelerated by GPU and its computational time is trivial as less than 1% of the projection operations. Thus, we will only focus on discussing the projection operations.

5.1. Time performance analysis

Table II shows some important attributes of the projection CUDA code and average time performance for a view. As we described in section 4, the projection CUDA code with ILP more consume registers per thread and it cause lower thread occupancy. However, the ILP more efficiently handles the long latency caused by fetching the three kinds of LUTs and image data. This efficiency results in better time performance in ILP (about 2 times faster than TLP) and also almost no time differences between forward- and back-projections.

The projection CUDA code with ILP is compared with the FFT approach on CPU (Mac Pro 2.66 GHz Quad-Core Intel Xeon) to verify the correctness of the CUDA code. Since the FFT approach cannot handle {asymmetric; variant} SR kernels, the {symmetric, invariant} SR kernels are used to compare two approaches. With given image

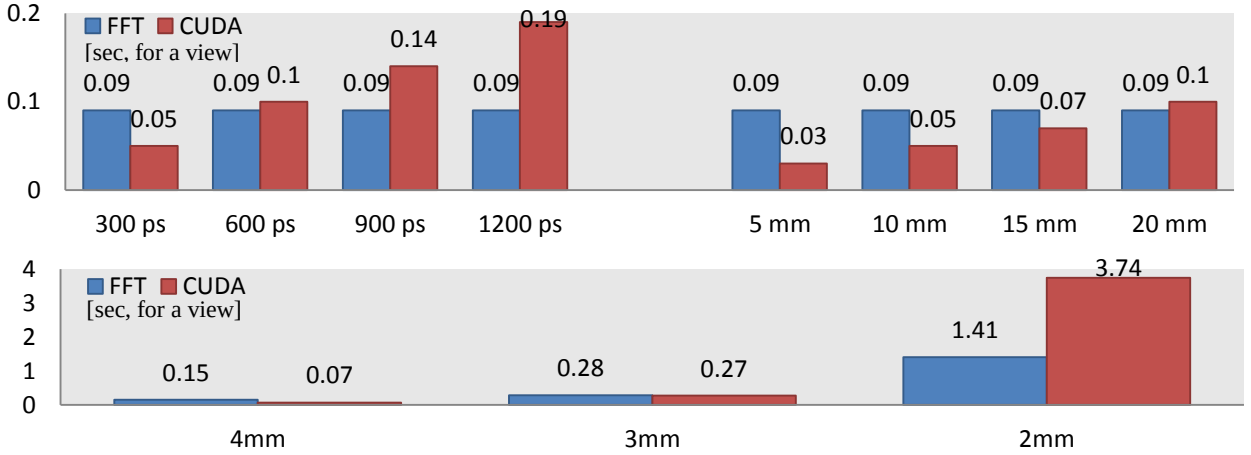


Fig. 4. Time performance comparisons between FFT and CUDA approach. [top] With different SR kernel resolutions and fixed voxel size and dimensions (4 mm, 144 x 144 x 48); varying TOF resolution (300 to 1200 ps FWHM) with fixed LOR resolution (10 mm and 5 mm radial and axial FWHM, respectively), and varying LOR resolution in radial direction (5 to 20 mm FWHM) with fixed TOF (300 ps) and axal (5 mm) resolutions. [bottom] With different voxel size (4 to 2 mm), and fixed SR kernel resolutions (375 ps FWHM for TOF and 6.5 mm for LOR) and voxel dimension (144 x 144 x 48)

data, there are two factors that we are interesting for the comparisons: size of SR kernels and voxel size. The size of SR kernels is further classified into two categories: varying TOF resolution with fixed LOR resolutions and varying LOR resolution in radial direction with fixed TOF and LOR resolutions in axial direction. Fig. 4 [top] shows the time performance comparisons with different size of the SR kernels. It is worth noting that the beauty of the FFT approach is that its time performance is not affected by the size of the SR kernels (for the fixed grid size) unlike the CUDA approach, where the projection operations are performed in the spatial domain. However, for the timing resolutions' characteristic of the modern PET scanners (600 ps and below), the CUDA approach exhibits similar or faster time performance. Fig. 4 [bottom] shows the effects of voxel size in two approaches. This test is performed for the clinical data and resolution size representative of the state-of-the-art TOF scanner based on the LaBr detector [8][9] for various voxels. Reducing voxel size means corresponding increase of the total number of voxels within the image, but at the same time also increase of the number of the voxels within the given SR kernel, both of them correspondingly increasing the computation demands of the space domain operations. For example, for reduction of the voxel size from 4 to 2 mm, the number of operations for the space based projection operations increases 2^3 x 2^3 = 64-times, while for the FFT-based approaches the number of operations increases only by a factor close to 2^3 = 8-times, because the FFT approach performance is affected only by the image dimensions and not by the SR kernel size. In practice, the CUDA approach time actually increases slightly less than predicted – by about 54.3-times – for the 4 to 2mm voxel size reduction, while the FFT approach time increases about 9.4-times; for the 4mm voxels the CUDA approach is about two times faster than the FFT approach, and for the 2mm voxels, it is about 2.7-times slower. However, it is important to note that in spite of this small increase of the total time for the 2mm voxel case in the FFT approach, the CUDA approach can efficiently handle projection operations also with {asymmetric; variant} SR kernels (as demonstrated later), which is not possible in the FFT approach.

5.2. Image quality analysis

Fig. 5 shows visual comparisons between FFT and CUDA approaches with a large and {symmetric, invariant} SR kernel (900ps of TOF resolution, 50mm and 10mm LOR resolution of FWHM in radial and axial direction, respectively). Since we model the SR kernel with the Gaussian function, which is symmetric, there is no difference between the forward and back-projections. Three point sources having the same intensity values are placed along the

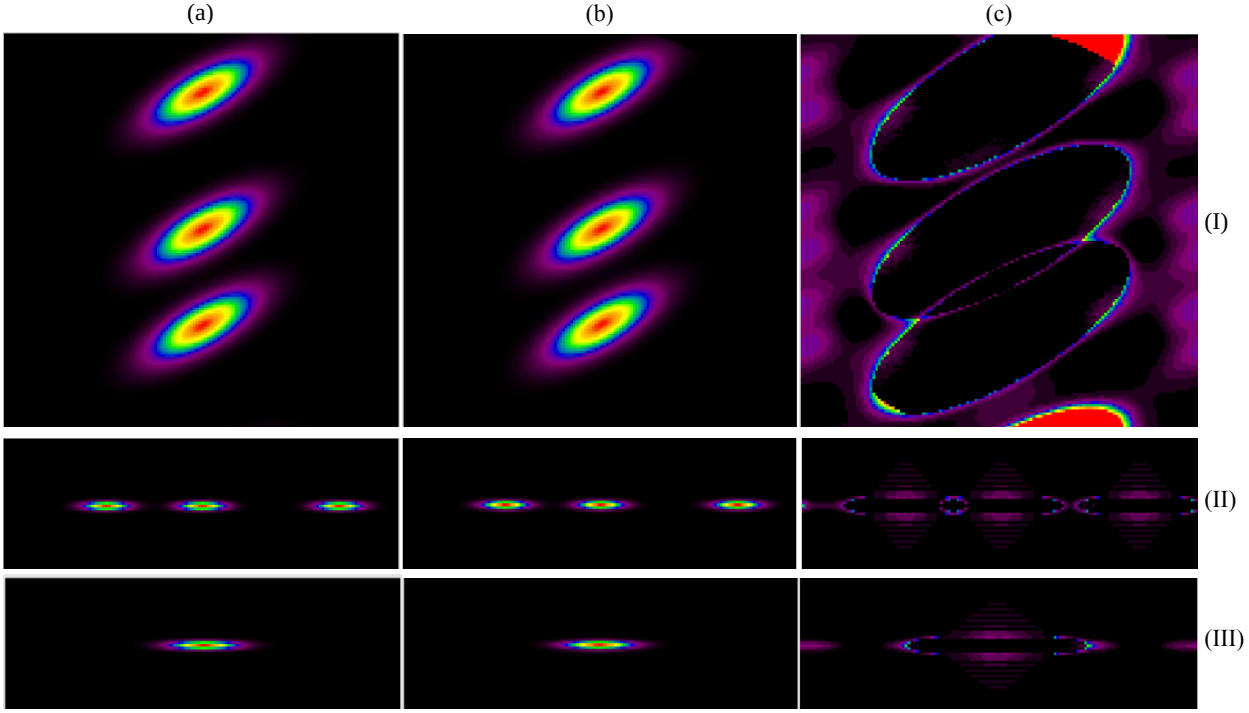


Fig. 5. Visual comparisons in FFT and CUDA. In column, (a) FFT (b) CUDA (c) difference image, and in row, (I) traverse (II) sagittal (III) coronal view. 900 ps for TOF resolution, 50 mm and 10 mm LOR resolution in radial and axial direction are used for the SR kernel. Since the SR kernel is modelled as spatially {symmetric, invariant} SR kernel, there are no differences in forward- and back-projections.

y-axis in the same image slice. In the CUDA approach, which works in the spatial domain, the SR kernel is truncated at the three times of σ_{dir} distance to minimize the SR kernel truncation errors. The maximum error between two approaches is less than 1% of the maximum intensity value.

The effects of the SR kernel truncation (at $\pm 3 \cdot \sigma_{dir}$) in the CUDA approach can be observed in the (scaled up) difference images in Fig. 5(c) – see ellipsoidal boundaries (at distances $\pm 3 \cdot \sigma_{dir}$ in TOF and LOR directions from the SR kernel centers) at which CUDA approach drops to zero while FFT approach still containes non-zero values (these truncation effects are less than 0.48% of the maximum value). On the other hand, truncation (at the Nyquist frequency) of the discretized SR kernel spectrum in the FFT approach causes small ripples (Gibbs artifacts) in the FFT generated kernels, especially in the directions of the short kernel axes (LOR radial and axial directions). It is important to note here that the color scale of the difference images in Figure 5(c) have been zoomed in to better show the structure of differences, and any differences commented on above are less than 1% of the maximum value. Furthermore, for the extra-large SR kernels used in this example, we can also observe spatial aliasing effects in the FFT approach caused by the periodic nature of the discretized image in the FFT, leading to cyclic convolution. For example, in Fig. 5(c, I), portion of the top of the SR kernel tail extending beyond (and truncated by) the top image boundary is leaking back into the bottom part of the image (from the periodic repeats of the image). To avoid such aliasing effects in the FFT approach, volume images can be padded with zeros before 3D-FFTs in x, y and z directions. The amount of needed padding depends on the SR kernel and image sizes, and eventually affects the computation times. But for the practical SR kernel and image sizes no, or only a very small amount of, zero-padding is usually needed.

In practice, the SR kernels have different LOR resolutions depending on their radial locations. In CUDA approach, such spatially varying SR kernels can be accurately modeled and applied to the projection operations. This is not possible in the FFT approach. The forward- and back-projection operations with {symmetric, variant} SR kernels are applied to three point sources as we did for {symmetric, invariant} case. With {symmetric, variant} SR kernels, the results from forward- and back-projections are different as shown in Fig. 6. This is what we can predict as discussed in section 2.2. In more detail, with {symmetric, variant} SR kernels, each voxel has a particular SR kernel based on its radial location. Thus, the forward-projection *spreads* the intensity value at each point source

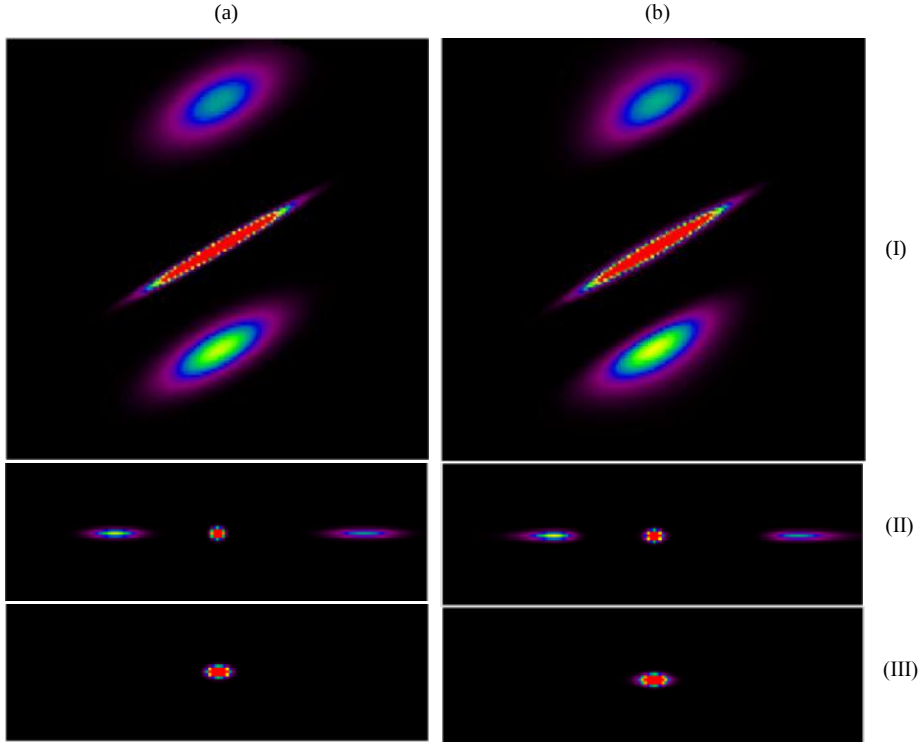


Fig. 6. Projection results with spatially {symmetric, variant} SR kernels. In column, (a) forward-projection (b) back-projection, and in row (I) transverse (II) sagittal (III) coronal view.

to its neighbors and results in symmetric ellipsoidal-like shapes, each having different level of blurring (width) according to its radial distance from the central line of the projection (Figure 6(a)). In contrast, during back-projection, voxels at various radial distances from the projection center *collect* values from the same point sources using the SR kernels of different widths (narrower at locations radially closer to the center, and wider at locations radially closer to the FOV edge). As shown in Figure 6(b), this results in an asymmetric ellipsoidal-like shapes (being wider towards the FOV edge).

Lastly, the CUDA code is tested for forward- and back-projection with spatially {(symmetric; asymmetric), (variant; invariant)} SR kernels. Since the code is designed to fetch kernel values from LUTs built at each view (or iteration), asymmetric kernel or even any arbitrary kernel can be handled by filling out the LUTs properly. To simulate asymmetric kernel and to compare it to symmetric kernel, the LOR resolutions in radial direction are generated using the sum of two Gaussians in which one has two times wider width. The Gaussian with wider width is sifted so that the sum of two Gaussians can have wider width on the side of the kernel towards the center of the FOV. The test condition is set to have 600 ps TOF resolution, 10 to 80 mm (variant) or 50 mm (invariant) of LOR resolution in radial direction, and 10 mm of LOR resolution in axial direction. Fig. 7 shows projection results with four different kinds of the SR kernels. As described in previous, with {symmetric, invariant} SR kernel, there is no difference in two projection operations; while {symmetric, variant} SR kernels result in asymmetric ellipsoidal-like shape for back-projection (Fig. 7 (a), (b)). With spatially {asymmetric, (variant; invariant)} SR kernels, we can still observe the behavior of forward- and back-projection which are represented as scattering and gathering operations, respectively. For the forward-projection, the results have elongated response toward the center of the FOV, while during the back-projection the elongated response is toward the opposite direction. Especially with spatially variant SR kernels, the elongation is getting longer (or blurring more) as it goes closer to the edge of the FOV (Fig. 7 (c), (d)). It is important to note that the SR kernels used in this test is not modeled based on scientific data. This test is to show the capability of our CUDA code that can handle any generic SR kernel resolutions and, in general, it is hard to be coped with FFT approach.

6. Analysis of DIRECT TOF PET reconstruction results

In this section, we test performance of CUDA forward- and back-projectors using both {symmetric, invariant} (for comparison to the compatible FFT case) and {symmetric, variant} SR kernels. The kernel parameters used in this section are chosen to approximate (in various degrees) characteristics of the state-of-the-art whole body TOF PET scanner.

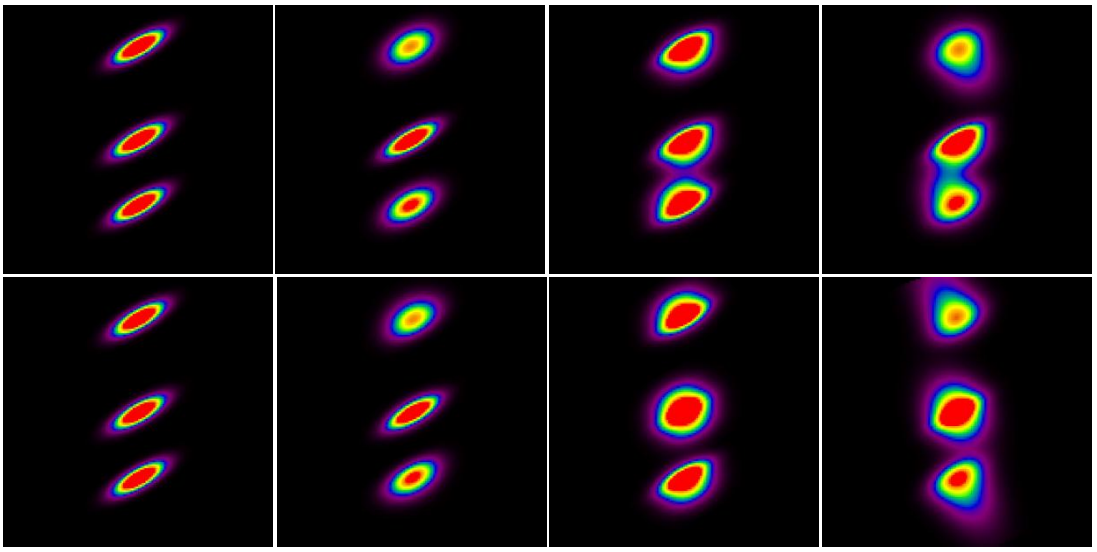


Fig. 7. Projection results with four different kinds of the SR kernels in spatial domain. The upper row is for forward-projection and the bottom one is for back-projection. In each row, the SR kernel is (a) symmetric, invariant, (b) symmetric, variant, (c) asymmetric, invariant, and (d) asymmetric, variant. All images are collected in traverse view at 30 degrees of view direction.

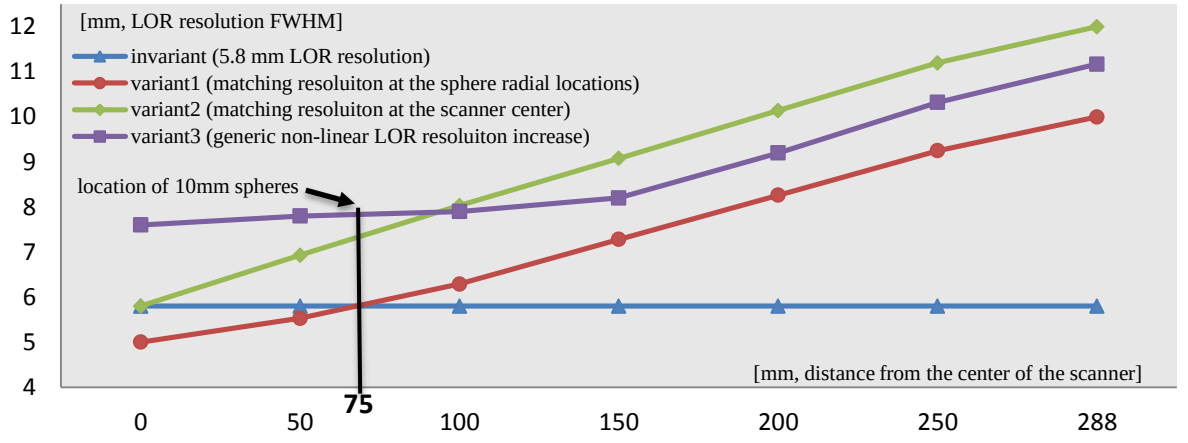


Fig. 8. Models of the spatially invariant and variant detector resolutions. The variant detector resolutions are modeled to vary along the radial direction with different slopes.

6.1. Methods

The DIRECT TOF PET reconstruction with two different projection approaches (FFT and CUDA) is tested with measured data obtained from the University of Pennsylvania prototype whole body LaBr3 TOF-PET scanner [8][9] having a 57.6 cm FOV, ±10 ° axial acceptance angle, with 4x4x30 mm^3 LaBr3 crystals (and with 4.3 mm crystal pitch). The crystals are placed within 24 detector flat modules which are placed on a cylindrical detector surface of diameter 93 cm, with a small (about 7 mm) gaps (in the azimuthal direction) between the modules. The intrinsic spatial resolution of the scanner is about 5.8 mm and timing resolution approximately 375-430 ps (depending on the count rates). The measured data include attenuation, and scatter and random events. The measured phantom is 35 cm diameter cylinder having clinically relevant volume and attenuation factors representative of a heavy patient. We acquire relatively high number of counts (approximately 430M prompts) to be able to better see any differences between the approaches. The phantom contains six uniformly distributed 10 mm diameter spheres at a radial positions about 7.5 cm (from the center) and places in central slice of the scanner.

We use a block version of RAMLA (Row-Action Maximum Likelihood Algorithm). In the DIRECT approach, we group and deposit events into 40 x 3 views: 40 intervals in azimuthal angle and 3 intervals in co-polar angle. Each view represent one block (subset) of RAMLA, giving us 120 (geometrically-ordered) updates for one pass through the data in the 40 x 3 view case. We employ TOF kernels representing 400 ps TOF resolution of measured data, and model spatially invariant and variant detector resolution kernels; the data deposition effects (image kernel) are not implicitly modeled in this particular study. The invariant LOR resolution kernel is applied within both approaches, FFT and CUDA. The variant LOR resolution kernels are applied (for obvious reasons) only in the CUDA approach. The final image array is 144 x 144 x 48 with 4 mm^3 voxels.

Fig. 8 shows the widths (FWHM) of the modeled detector LOR resolutions for invariant and variant cases. For the invariant case, we use 5.8mm FWHM LOR resolution over the entire FOV. We model also three variant kernels: first matching the LOR resolution at the center of the scanner (green line and denoted as variant2), second matching LOR resolution at the spheres' locations at 7.5 cm radius (red line and denoted as variant1), and third by choosing a generic resolution functions varying with the radius in a non-linear fashion (purple line and denoted variant3).

The behavior of variant and invariant resolution modeling in conjunction with iterative reconstruction is investigated using the contrast versus noise trade-off for a range of iterations and reconstruction parameters. Contrast recovery coefficients (CRC) are calculated for all spheres as: $CRC = (\frac{p_s - p_b}{p_b})/c$, where p_s is the mean value in a 2D circular region of interest (ROI) axially and transversely centered over the sphere, p_b is the mean background value in the 2D annular region surrounding and centered over each sphere (with the annulus inner diameter being 14 mm larger than the sphere diameter, and the annular region being 8 mm thick), and c is the ideal contrast value. The reported CRC values are average CRC values over all 10 mm spheres in the phantom. The noise

TABLE III. TIME PERFORMANCE COMPARISON IN DIRECT TOF PET RECONSTRUCTION BETWEEN FFT AND CUDA

Approach	FFT	CUDA			
LOR type	invariant	invariant	variant1	variant2	variant3
at the edge [mm]	5.8	5.8	10	12	11.17
1 iteration [sec]	21.48	15.63	22.43	25.66	24.68

TABLE IV. MEASURED TIME AT CRITICAL STAGE FOR INVARIANT AND VARIANT1 CASES WITH GPU IN DIRECT TOF PET (20 ITERATIONS)

[min:sec]	Forward-projection (GPU)			Discrepancy (CPU)	Back-projection (GPU)			Update (CPU)	Total
	H2D	FP	D2H		H2D	BP	D2H		
invariant	0:7.53	1:45.27	0:2.85	0:0.32	0:7.59	1:43.18	0:2.85	0:12.59	5:11.97
variant1	0:7.55	2:52.88	0:2.86	0:0.32	0:7.59	2:49.81	0:2.85	0:12.58	7:26.21

Note 1. For both tables, image size is 144 x 144 x 48 with 4 mm^3 voxels and single iteration consists of 40 x 3 views
Note 2. H2D: data transfer from CPU to GPU, D2H: data transfer from GPU to CPU

is evaluated as the pixel-to-pixel noise standard deviation inside a large 50 mm ROI located in the central uniform region of the phantom and normalized by the background mean value.

6.2. Analysis of reconstruction results

Table III shows time performance for the invariant and variant SR kernels with the LOR widths based on the plots shown in Fig. 8 (and as specified in the section 6.1). Similar to the evaluations of the projection operations (Fig. 4), CUDA shows slightly faster or comparable time performance to the FFT for the invariant case. For the variant cases, the required time per iteration is slightly longer compared to the FFT invariant case. Note that for the variant SR kernels, the SR kernel's physical (memory) size is defined by the longest LOR width (from all variant LOR resolutions for a given view), so the practical CUDA time performance is determined by the LOR resolution at the radial boundary of the FOV. DIRECT iterative reconstruction time include all of the reconstruction stages such as reading of the data, data transfers to and from the GPU, forward- and back-projection, and discrepancy (computing the differences between the forward-projected and measured deposited data) and update (updating the image estimation) operators, as reported in Table IV. It is worthwhile to mention that the forward- and back-projection operations still remain the bottleneck of the reconstruction process even if they are implemented on the GPU and carefully optimized. The other operations take only about 10% (or less) of the total reconstruction time, and therefore it is not necessary to optimize and/or implement them on the GPU.

Fig. 9 illustrates contrast versus noise trade-off curves for DIRECT reconstruction using variant and invariant resolution models. Fig. 10 shows representative images of individual cases for matched noise levels (about 8% of noise level in Fig. 9). It is clear from the graphs and visual image quality that the FFT and GPU approaches using

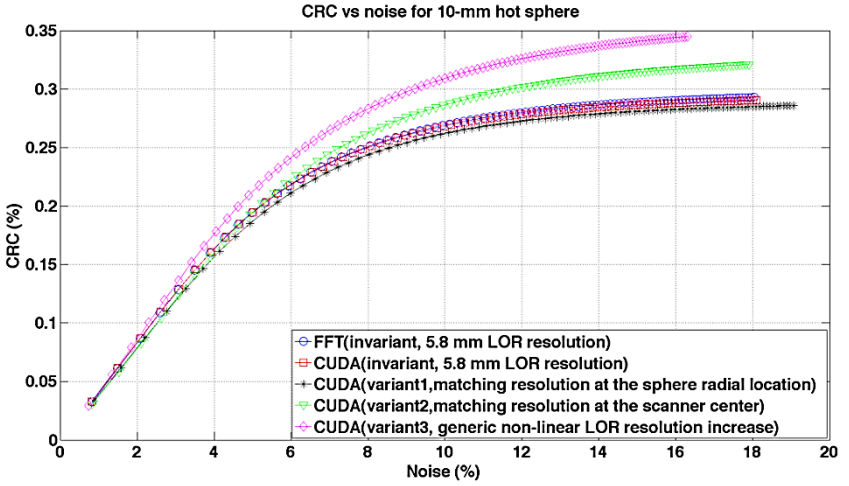
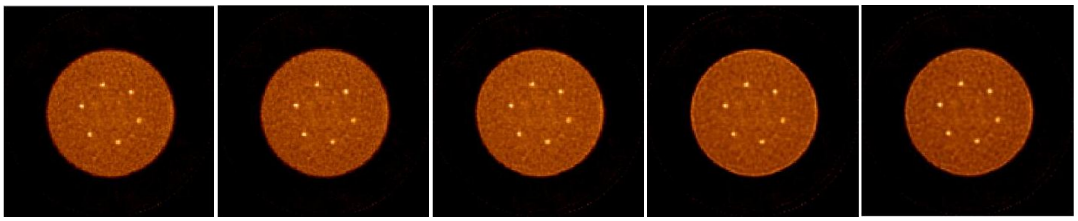


Fig. 9. CRC vs. Noise trade-off curves



(a) (b) (c) (d) (e)

Fig. 10. Reconstructed images (transverse view) at matched noise levels for DIRECT TOF PET reconstruction using different approaches and resolution models. (a) FFT with invariant SR kernel at 22 iterations (b) CUDA with invariant SR kernel at 22 iterations (c) CUDA with variant SR kernels (variant1) at 19 iterations (d) CUDA with variant SR kernels (variant2) at 21 iterations (e) CUDA with variant SR kernels (variant3) at 26 iterations

invariant resolution models provide nearly identical results. In the spatially variant resolution case (variant1 in Fig. 8) in which the LOR resolution at the spheres' locations is matched to that in to the invariant case (5.8 mm FWHM), the CRC curve converged to slightly lower values compared to the invariant case. This is due to the fact that in the variant resolution case the actually modeled LOR resolution at each particular sphere location changes with the projected view (from 4.8 mm to 5.8 mm in the considered variant1 case), based on the radial distance of the sphere from the projection central line in each view. So, although this is more accurate modeling, the average modeled resolution is actually lower than that in the invariant case, and thus leading to a lower contrast values. For the other variant cases (variant2 and variant3 in Fig. 8), having higher resolution models, the contrast converges to higher values. This is accompanied by increased overshoots (Gibbs artifacts) at the object boundaries (Fig. 10).

7. Conclusion

In this study, we implemented, optimized and evaluated GPU-based forward- and back-projection operations (of any tilt and view direction) for DIRECT iterative reconstruction approach considering generic system resolution kernels, including asymmetric invariant kernels. By careful implementation concerning memory access pattern for off-aligned axes in GPU memory and tuning-up of the projection CUDA code between the thread-level (TLP) and instruction-level (ILP) parallelism we were able to obtain slightly faster or comparable computational efficiency of the projection operations compared to the very efficient implementation of the same (Fourier-based) operations via the FFT-based approaches using state-of-the-art FFTW routines. However, the FFT-based approach cannot handle spatially invariant (and generic) resolution models, while the developed space-based GPU tools can efficiently handle any generic resolution kernel. The proposed GPU-based tools thus provide important contribution to the PET reconstruction field by allowing more accurate data modeling within DIRECT iterative reconstruction, without significant effects on the practical/clinical time performance.

References

1. S. Surti, J. S. Karp, L. M. Popescu, M. E. Daube-Witherspoon, and M. Werner, "Investigation of time-of-flight benefit for fully 3-D PET," *IEEE Trans. Med. Imag.*, vol. 25, no. 5, pp. 529-538, May 2006.
2. M. Conti, B. Bendriem, M. Casey, M. Chen, F. Kehren, C. Michel, and V. Panin, "First experimental results of time-of-flight reconstruction on an LSO PET scanner," *Phys. Med. Biol.*, vol. 50, no. 19, pp. 4507-4526, 2005.
3. C. C. Watson, "Image noise variance in 3D OSEM reconstruction of clinical time-of-flight PET," in *Conf. Rec. 2006 IEEE Nucl. Sci. Symp. Med. Imag. Conf.*, San Diego, CA, 2006.
4. J. S. Karp, S. Surti, M. E. Daube-Witherspoon, and G. Muehllehner, "Benefit of time-of-flight in PET: Experimental and clinical results," *J. Nucl. Med.*, vol. 49, no. 3, pp. 462-470, 2008.
5. S. Matej, S. Surti, S. Jayanthi, M. Daube-Witherspoon, R. Lewitt, J. Karp, "Efficient 3D TOF PET reconstruction using view-grouped histo-image: DIRECT-direct image reconstruction for TOF," *IEEE Trans. Med. Imag.*, 28(5):739-51, 2009.
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 Nucl. Sc. Symp. Med. Imag. Conf.*, M09-256, Knoxville, TN, 2010.
7. L. Popescu, S. Matej, R. Lewitt, "Iterative image reconstruction using geometrically ordered subsets with list-mode data, " *IEEE Med. Imag. Conf*, M9-211, pp. 3536-3540, 2004.
8. J. S. Karp, A. Kuhn, A. E. Perkins, S. Surti, M. E. Werner, M. E. Daube-Witherspoon, et al., "Characterization of a time-of-flight PET scanner based on lanthanum bromide," *IEEE Nucl. Sci. Symp. Med. Imag. Conf.*, M04-8, pp. 23-29, San Juan, Puerto Rico, October 2005.
9. M. E. Daube-Witherspoon, S. Surti, A. Perkins, C. C. M. Kyba, R. Wiener, M. E. Werner, et al., "The imaging performance of a LaBr3-based PET scanner," *Phys. Med. Biol.*, vol. 55, pp. 45-64, 2010.

10. F. Xu, K. Mueller, “Accelerating popular tomographic reconstruction algorithm on commodity PC graphics hardware,” *IEEE Trans. Nucl. Sci.*, 52(3):654-663, 2005.
11. F. Xu, K. Mueller, “Real-Time 3D Computed Tomographic Reconstruction Using Commodity Graphics Hardware,” *Phys. Med. Biol.*, 52:3405-3419, 2007.
12. S. Ha, Z. Zhang, S. Matej, K. Mueller, “Efficiently GPU-Accelerating Long Kernel Convoluitons in 3D DIRECT TOF PET Reconstruction via a Kernel Decomposition Scheme,” *IEEE Med. Imag. Conf.*, Knoxville, TN, October 2010.
13. Z. S. Hakura and A. Gupta, “The design and analysis of a cache architecture for texture mapping,” *Proc. 24th Int’l Symp. on Computer Architecture*, pp. 108-120, 1997.
14. J. L. Lo, J. S. Emer, H. M. Levy, R. L. Stamm, D. M. Tullsen, and SJ Eggers, “Converting thread-level parallelism to instruction-level parallelism via simultaneous multithreading,” *ACM Trans. Comp. Sys.*, vol. 15, no. 3, pp. 332-354, 1997.
15. NVIDIA Corporation, “CUDA Programming Guide Version 4.0,” 2011.
16. NVIDIA Corporation, “NVIDIA GeForce GTX 200 GPU Architectural Overview,” http://www.nvidia.com/docs/IO/55506/GeForce_GTX_200_GPU_Technical_Brief.pdf, May 2008.
17. A. Bakhoda, G. Yuan, W. W. L. Fung, H. Wong, and T. M. Aamodt, “Analyzing CUDA Workloads Using a Detailed GPU Simulator,” *IEEE ISPASS*, April 2009.
18. D. B. Kirk and W. W. Hwu, “Programming Massively Parallel Processors: A Hands-on Approach 1st ed.,” Morgan Kaufmann Publishers Inc., 2010.
19. G. Pratx and L. Xing, “GPU computing in medical physics: A review,” *Med. Phys.*, 38(5):2685-2698, 2011.
20. V. Volkov, “Better performance at lower occupancy,” *Proc. the GPU Technology Conf.*, vol. 10, 2010.