A GPU-Accelerated Multivoxel Update Scheme for Iterative Coordinate Descent (ICD) Optimization in Statistical Iterative CT Reconstruction (SIR)

Sungsoo Ha and Klaus Mueller, *Senior Member, IEEE*

***Abstract*—Statistical iterative reconstruction (SIR) algorithms have shown great potential for improving image quality in reduced and low dose X-ray computed tomography (CT). However, high computational cost and long reconstruction times have so far prevented the use of SIR in practical applications. Various optimization algorithms have been proposed to make SIR parallelizable for execution on multicore computational platforms, whereas others have sought to improve its convergence rate. Parallelizing on a set of decoupled voxels within an iterative coordinate descent (ICD) optimization framework has shown good promise to achieve both of these premises. However, so far these types of frameworks come at the price of additional complexities or are limited to parallel beam geometry only. We improve on this prior research and present a framework, which also achieves parallelism by processing sets of independent voxels, but does not introduce additional complexities and has no restrictions on beam geometry. Our method uses a novel multivoxel update (MVU) scheme within a general ICD framework fully optimized for acceleration on commodity GPUs. We also investigate different GPU memory access patterns to increase cache hit-rates that result in improved time performance in the ICD framework. Experiments demonstrate speedups of two orders of magnitude for clinical datasets in cone-beam CT geometry, compared to the single-voxel update scheme native to conventional ICD-based SIR. Finally, since our MVU scheme operates on fully independent voxels, it maintains the fast convergence properties of ICD-based SIR. Consequently, the speedups achieved by parallel computing are not diminished by slower convergence of the iterative updates or by any additional overhead to decouple conflicting voxels.**

***Index Terms*—Statistical iterative CT reconstruction, SIR, iterative coordinate descent optimization, ICD, multi-voxel update, GPU, CUDA.**

I. Introduction

THE statistical iterative reconstruction (SIR) algorithm for computed tomography (CT) has shown great potential to generate high quality images with less artifacts and noise even in reduced X-ray settings. This capability mainly results from the statistical noise modeling that puts higher weight on reliable measurements while deemphasizing noisy measurements. It solves the following weighted least-squares (WLS) cost function:

$$\hat{\mathbf{x}} = \arg\min_{\mathbf{x}\geq 0} \left\{ \frac{1}{2}(\mathbf{y}-\mathbf{A}\mathbf{x})^T \mathbf{W}(\mathbf{y}-\mathbf{A}\mathbf{x}) + R(\mathbf{x}) \right\}, \tag{1}$$

where $\mathbf{y} = (y_1, \ldots, y_M)^T$ is the vector of CT measurements and $\mathbf{x} = (x_1, \ldots, x_N)^T$ is the vector representing the unknown object subject to reconstruction. M is the total number of CT measurements computed by multiplying the number of detector bins with the number of projection views, while N is the number of voxels in the 3D volumetric object. $\mathbf{A}$ is the system matrix of size $M \times N$ in which each element, a_{ij}, represents the contribution of the j-th voxel to the i-th CT measurement [1]–[4]. W is a $M \times M$ diagonal matrix of $(\lambda_1, \ldots, \lambda_M)$ in which λ_i represents the photon count for the i-th measurement [5]. The first term in Eq.(1) is the data-fidelity term where the simulated forward projections of the estimated object, $\mathbf{A}\mathbf{x}$, is compared with the CT measurement. The squared difference term is weighted by W to put more weights on the data less affected by Compton scattering and photoelectric absorption of the associated X-ray. $R(\mathbf{x})$ is the regularizing prior function [6]. It only depends on the object and controls noise while attempting to preserve spatial resolution [6]–[10]. We use q-generalized Gaussian Markov random fields (q-GGMRF) for the prior function throughout this work (see [11] for mathematical details), but mainly focus on parallelizing the data-fidelity term using modern parallel processors, such as the GPU.

Broadly speaking, there are two approaches to solve the WLS minimization problems in Eq.(1). The first approach is updating the entire volume simultaneously and it includes conjugate gradient (CG) methods [12], and ordered-subsets (OS) algorithms based on separable quadratic surrogates (SQS) [13]. Although these approaches can readily take advantage of the massive compute resources on the GPU [14], [15], it usually requires tens to hundreds of iterations to converge [16]. To gain a better convergence rate, preconditioning methods [12], [17] that are necessary customized for the given CT image system and geometry are utilized for CG methods. For OS-SQS, Kim *et al.* [18], [19] explored non-uniform update scheme [18] and momentum techniques [19] for the convergence acceleration. Recently, Nien *et al.* [20], [21] proposed OS algorithms based

Manuscript received July 26, 2017; revised December 16, 2017 and March 30, 2018; accepted April 29, 2018. Date of publication May 9, 2018; date of current version August 13, 2018. This work was supported in part by NSF under Grant IIS 1527200 and in part by the Ministry of Science, ICT and Future Planning, South Korea, under the IT Consilience Creative Program (ITCCP) supervised by NIPA. The associate editor coordinating the review of this manuscript and approving it for publication was Prof. Jong Chul Ye. *(Corresponding author: Sungsoo Ha.)*

The authors are with the Visual Analytics and Imaging Laboratory, Computer Science Department, Stony Brook University, Stony Brook, NY 11794-4400 USA (e-mail: sunha@cs.sunysb.edu; mueller@cs.sunysb.edu).

Digital Object Identifier 10.1109/TCI.2018.2833622

on a linearized augmented Lagrangian method (LALM) that can converge even in a few iterations.

The second approach is based on iterative coordinate descent (ICD) optimization which greedily optimizes a single voxel at a time [6], [22]. Due to the tight feedback from each voxel during the optimization, this approach converges to a robust solution quite fast, within 10 to 20 iterations [16]. Even faster are non-homogeneous ICD update algorithms which use selective voxel update schemes where convergence can be achieved in less than 5 iterations [11]. However, the inherent single voxel update requirement makes ICD-based SIR much more difficult to parallelize than its counter parts.

The strive to find a middle ground between two extreme cases, updating one voxel versus entire ones, led to the development of the group coordinate descent (GCD) [23] and the block-iterative coordinate descent (B-ICD) [24] algorithms. Both aim at finding parallel voxels within a transaxial (xy) plane where, however, in most cases the coupling among those voxels remains high. And so, due to this lack of voxel independence the net acceleration tends to be not overly dramatic. Another challenge in GCD and B-ICD is to strike a good balance between cache locality and parallelism. More parallelism can be found when the number of blocks increases (i.e. the size of the each block becomes smaller). However, with smaller blocks the number of cache faults increases due to the sinusoidal in-plane memory access pattern inherent to CT. Recently, Wang *et al*. proposed the concept of super-voxel (SV) and super-voxel buffer (SVB) to find this balance. They were able to achieve speed-ups of two levels of magnitude on multi-core CPUs ($\times$187 on 20 cores vs. single core) [25]. The concept was successfully extended to a single GPU implementation for further speed-ups ($\times$4) [26] and was applied to synchrotron based X-ray CT reconstruction [27]. However, this work was still limited to parallel beam geometry. Conversely, the axial block coordinate descent (ABCD) algorithm by Fessler and Kim [28] looks for parallel voxels along the axial (z) direction where the amount of coupling is relatively small, compared to the transaxial plane. Their study, however, only compared the SQS-based and conventional ICD-based approaches in terms of convergence.

In our own recent work [29] we attempted to algorithmically find maximal groups of parallelizable voxels that were fully decoupled from each other. This approach revealed some rather complex patterns very different from the regular groupings presented in other work [23]–[28]. The patterns we found offered a theoretical speed-up of two orders of magnitude (compared to conventional ICD-based SIR) for any view geometry. However, while the voxels groups were quite large, which would favor parallelism, the voxels were distributed throughout the volume slices in pseudo-random arrangements leading to unfavorable cache behavior. For this reason, we abandoned this approach in favor of a more regular arrangement of parallelizable voxels along the axial (z) direction which we obtain via a dedicated algorithm.

Our work differs from the ABCD algorithm[28] in that in our case the parallelizable voxels are fully decoupled from each other and so no additional computational overhead is needed to undo any overlap effects among voxels processed in parallel. In addition, most of existing research [23]–[25], [28] has focused on multi-core high-end CPUs systems, while our work uses a single GPU which is significantly more economical. For example, Wang *et al*. [25] used two high-end Intel processors at a combined cost of $3,239, while the single GPU we use in our work costs a mere $509 which is less than 1/6 of that. Finally, compared to the recent GPU-based ICD-SIR approach by Sabne *et al*. [26] which was focused on parallel beam geometry, our work finds the parallelism across the axial direction which is more amenable to cone-beam or helical CT.

The contributions in this work are in two folders. First, we propose multi-voxel update (MVU) scheme by finding an optimal set of fully decoupled voxels along the axial direction. The full independence of these parallel voxels ensures that the rapid convergence rate of the conventional single-voxel update (SVU) scheme is preserved, and so is image quality. Second, we provide details of an effective CUDA implementation strategy for the three core functions, forward- and back-projections and ICD update. We find that this implementation gives excellent GPU utilization even with relatively small number of voxels that can be processed concurrently compared to CG- or OS-based algorithms.

The remainder of this paper is organized as follows. Section II reviews the single-voxel update scheme of conventional ICD-based SIR and explains the proposed multi-voxel update schemes. Section III presents our framework and implementation details. Section IV reports on experimental results we obtained with our ICD-based SIR with the new multi-voxel update scheme. Section V concludes this paper with discussion and future works.

II. Methodology

A. Single-Voxel Update in ICD-SIR

The conventional ICD-based statistical iterative CT reconstruction (ICD-SIR) algorithm operates in a sequential fashion and updates a *single voxel at a time*. This minimizes the cost function of Eq.(1) for the chosen voxel $\mathbf{x}_j$ while keeping all remaining voxels fixed. Formally, the update is:

$$\mathbf{x}_j = \underset{\mathbf{x}_j \geq 0}{\arg\min} \left\{ \frac{1}{2}(\mathbf{y} - \mathbf{A}\mathbf{x})^T \mathbf{W}(\mathbf{y} - \mathbf{A}\mathbf{x}) + R(\mathbf{x}_j) \right\} . \quad (2)$$

This update can be computed efficiently by tracking the residual projections defined by $\mathbf{e} = \mathbf{A}\mathbf{x} - \mathbf{y}$. To do this, the first (θ_1) and second (θ_2) derivatives of the data-fidelity term in Eq.(2) are computed as follows:

$$\theta_1 = \sum_{i=1}^{M} \lambda_i \mathbf{A}_{ij} \mathbf{e}_i \quad (3)$$

$$\theta_2 = \sum_{i=1}^{M} \lambda_i \mathbf{A}_{ij}^2 , \quad (4)$$

where $\mathbf{e}_i$ is the i-th residual projection value; and λ_i is the i-th element of the diagonal weight matrix, $\mathbf{W}$. Then, the minimization

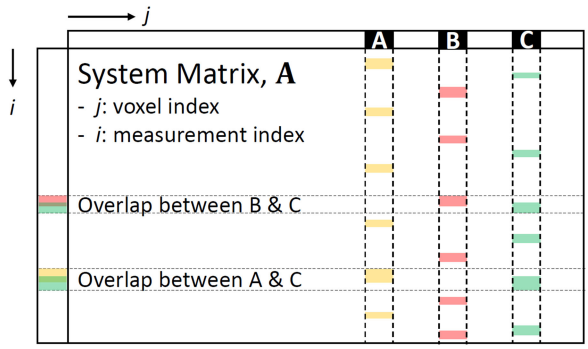

Fig. 1. Example of correlation among voxels in CT system matrix view.

Algorithm 1: Single Voxel Update (SVU).

Function SVU(A,W,x,e,j)
- $\bar{\mathbf{x}}_j \leftarrow \mathbf{x}_j$;
- $\theta_1 \leftarrow Eq.(3)$;
- $\theta_2 \leftarrow Eq.(4)$;
- $\mathbf{x}_j \leftarrow Eq.(5)$;
- $\mathbf{e} \leftarrow \mathbf{e} + \mathbf{A}_{*j}(\mathbf{x}_j - \bar{\mathbf{x}}_j)$;

end

of the 1D objective function for $\mathbf{x}_j$ is derived from Eq.(2) as follows [30]:

$$\mathbf{x}_j \leftarrow \underset{r \geq 0}{\arg\min} \left\{ \theta_1 r + \frac{\theta_2 (r - \bar{\mathbf{x}}_j)^2}{2} + f(r, \mathbf{x}_{\partial j}) \right\} , \quad (5)$$

where $\bar{\mathbf{x}}_j$ is the attenuation of the j-th voxel before the update and $f(r, \mathbf{x}_{\partial j})$ is a prior model that typically takes 26 neighbors of the j-th voxel into account. The residual projections are updated by forward projecting the voxel update amount, $\mathbf{x}_j - \bar{\mathbf{x}}_j$. The overall procedure of the single voxel update (SVU) scheme is listed in Algorithm.1. Note that we explicitly follow the mathematical details of the SVU scheme outlined in [22].

In the SVU scheme each selected voxel, $\mathbf{x}_j$, first back-projects the error terms *corresponding to this voxel* into image space, and after solving the 1-D object function for this voxel the error terms *corresponding to this voxel* are updated by forward projecting the updated amount. In the following, we will describe our proposed multi-voxel update (MVU) scheme. It is different from SVU in that it identifies not one voxel, but a set of voxels, which however like sequential SUV have non-overlapped correspondences with the error terms. This enables a parallelized sequential SVU without adverse side effects.

B. Multi-Voxel Update in ICD-SIR

The multi-voxel update (MVU) scheme is achieved by first identifying voxels that do not have shared correspondences with projection data. More specifically, we say that two voxels, i and j, are parallelizable if the correlation is zero ($cor(i,j) = 0$). We define the correlation as follows [25], [29]:

$$cor(i,j) = \sum_{k=1}^{M} |a_{ki}| \cdot |a_{kj}| . \quad (6)$$

For example, in Fig. 1, voxel **C** has a non-zero correlation with both voxel **A** and **B** due to several shared (detector bin) data correspondences and hence cannot be simultaneously updated with others. On the other hand, voxel **A** and **B** do not have shared data correspondences (i.e. zero correlation), and hence they are good candidates for parallel updates. Essentially, with these voxels having zero correlation, the MVU is simply a parallel execution of traditional SVU.

Another important aspect to consider when choosing the set of parallelizable voxels is the memory access patterns for the CT projection data. This pattern changes per given viewing geometry, such as cone beam, helical beam, and so on. Let us denote the s- and t-axis the local coordinate system of a flat X-ray detector in which the t-axis is parallel to the z-axis. Also, let us assume that the X-ray source-detector pair is rotating about the rotation axis, which is aligned with the z-axis, following a helical path. At a projection angle (β), the central positions of the X-ray source ($\mathbf{P}_{src}$) and the detector ($\mathbf{P}_{det}$) can be formulated as follows:

$$\mathbf{P}_{src} = [d_{sad} \cdot \sin\beta,\ -d_{sad} \cdot \cos\beta,\ h\beta] , \quad (7)$$

$$\mathbf{P}_{det} = [-d_{add} \cdot \sin\beta,\ d_{add} \cdot \cos\beta,\ h\beta] , \quad (8)$$

where d_{sad} and d_{add} are srouce-axis and axis-detector distance, respectively. Its pitch is defined as $2\pi h$ where $h \in \mathbb{R}$ and $h > 0$ for a helical path and $h = 0$ for a circular path. Then, the projected position, (s,t), of a point, $\mathbf{P}(x,y,z)$, onto the detector is computed as follows:

$$s = \kappa \times (\hat{\mathbf{s}} \cdot \mathbf{P}) , \quad (9)$$

$$t = \kappa \times (z - h\beta) , \quad (10)$$

where κ is computed as

$$\kappa = \frac{\mathbf{d}_{sad} + \mathbf{d}_{add}}{\mathbf{d}_{sad} + \hat{\mathbf{r}} \cdot \mathbf{P}} . \quad (11)$$

Here, $\hat{\mathbf{s}}$ is the direction unit vector of the s-axis and $\hat{\mathbf{r}}$ is the orthogonal unit vector connecting the X-ray source to the detector plane at a projection view, β. For a path rotating in counter clockwise direction, they are computed as

$$\hat{\mathbf{s}} = [\cos\beta,\ \sin\beta,\ 0] \quad (12)$$

$$\hat{\mathbf{r}} = [-\sin\beta,\ \cos\beta,\ 0] . \quad (13)$$

For a circular path ($h = 0$), projected positions of voxels located within the same slice exhibit a large amount of variation in the s-axis, resulting in a sinusoidal pattern over different views (see Eq.(9)). Not only does this pattern make it difficult to find fully decoupled voxels within the same slice, it also results in poor cache locality since it requires a voxel to access different entries of the CT projection data for different views [25]. On the other hand, voxels in different slices but at the same xy-position typically are only loosely coupled or even fully decoupled. In addition, their projected positions have less variations along the t-axis (see Eq.(10)). Hence, it seems better to find parallelizable voxels along the z-direction, as this results in better cache locality when fetching the CT projection data (provided they are stored with the leading dimension being the t-axis).

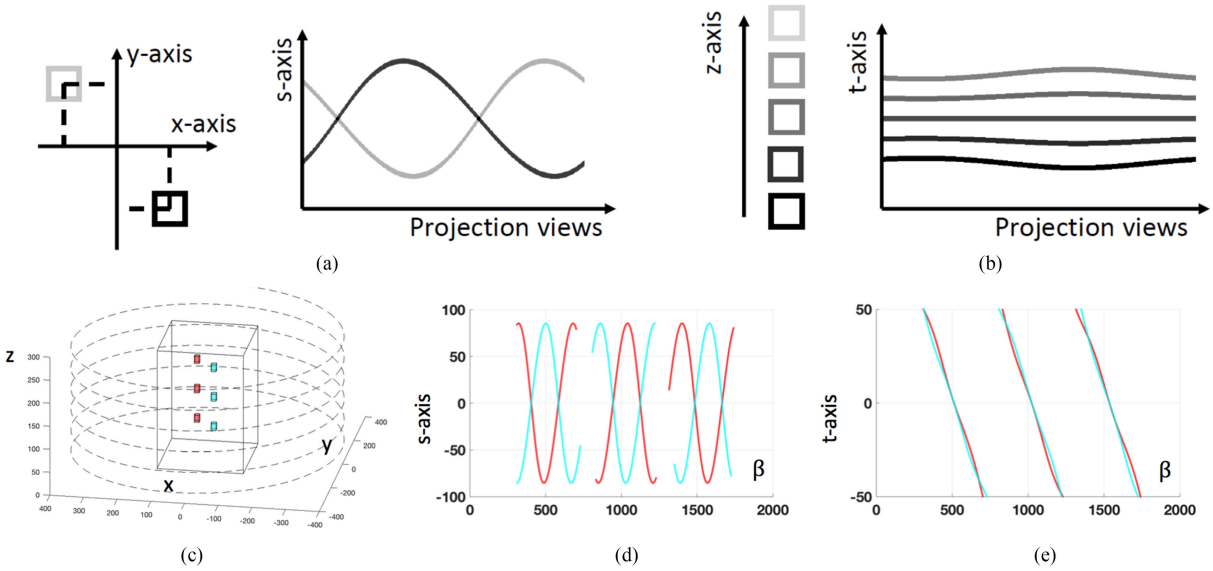

Fig. 2. Voxel correlations and memory access patterns. For a circular path, (a) when voxels are in the same slice, there are at least one or two intersections that give rise to non-zero correlations among the voxels. Also, memory access patterns for CT projection data widely varies along the s-axis. (b) For the well grouped voxels along the z-direction, zero correlations can occur already with small variation along the t-axis, resulting in better cache locality along the t-direction as well as good parallelism along the z-direction. Similarly in a helical path, (c) for voxels sampled along the z-direction (either red or blue), they have zero correlations as they do not interfere with each other in (d) s-axis and (e) t-axis over all views.

Fig. 2 illustrates voxel correlations and memory access patterns when voxels are in the same slice but at different xy positions [Fig. 2(a)], or in different slices but at the same xy position [Fig. 2(b)]. Similar patterns are also observed in a helical path ($h > 0$). For example, in Fig. 2(c), there are two sets of voxels sampled in two different z-positions. While voxels in a set have zero-correlation and thus can be processed concurrently, voxels in a same slice interfere with each other in the corresponding projection space (along s- and t-axis).

We modified the algorithm we presented in [29] to find sets of fully decoupled (and therefore parallelizable) voxels along the z-direction, for a given CT geometry and set of projection views. Let us assume a 3D volume object of size $N_x \times N_y \times N_z$ with voxel size $\Delta_x \times \Delta_y \times \Delta_z$ mm^3 which we wish to reconstruct. Then, due to the sinusoidal projection pattern there are $N_x \times N_y$ SVUs which cannot be parallelized when N_z is equal to 1. On the other hand, for $N_z > 1$ and for each voxel in the xy-plane, a group of parallelizable voxels can be found by looping over all voxels at that slice position along the z-direction. For each such voxel and view we compute the range $[t_{\min,\beta}, t_{\max,\beta}]$ of projected positions. Given a view, β, for example, $t_{\min,\beta}$ is the minimum value of t computed using Eq.(10) over the 4 projected voxel corners, $(x \pm \Delta_x/2, y \pm \Delta_y/2, z - \Delta_z/2)$; in similar way, $t_{\max,\beta}$ is the maximum value of t using the 4 projected voxel corners, $(x \pm \Delta_x/2, y \pm \Delta_y/2, z + \Delta_z/2)$. Next, these ranges are compared with the union of all ranges of the parallelizable voxels aggregated in the group so far. If and only if there are no overlaps in all views, the voxel is added to the current group. These two steps are repeated until all voxels are checked for the current group. We then repeat this process until all voxels along the z-direction have been assigned to any group. For example, in Fig. 2(b), the five voxels along the z-direction are updated simultaneously because there are no overlaps in all projection views, and same for voxels colored in either blue or red in Fig. 2(c).

III. CUDA Implementation

For the ICD-SIR framework we seek to accelerate, the back projection's task is to compute θ_1 in Eq.(3) and θ_2 in Eq.(4) while the forward projection operation updates the error terms, $\mathbf{e}$, for each voxel, as $\bar{\mathbf{x}}_j - \mathbf{x}_j$. The 1D minimization problem in Eq.(5) can vary according to the prior model, $R(\mathbf{x})$.

Our proposed multi-voxel update (MVU) scheme for ICD-SIR (see Section II-B) aims to parallelize both the forward- and back-projection via dedicated CUDA kernels over K parallelizable voxels within a group, g, and N_β projection views. Both kernels are built on top of the separable footprint (SF) projector where we use the trapezoid-rectangle function for and the A1 amplitude method [3] for efficiency. Algorithm 2 presents an overview of our MVU-based ICD-SIR method.

In the following, Section III-A describes an efficient implementation of the SF projection kernel, while Section III-B discusses our GPU memory management. The actual GPU implementation of the forward- and back-projection operations is covered in Sections III-C and III-D. We also describe an efficient CUDA implementation strategy for Eq.(5) with a prior model including a 26 neighborhood for each of the K voxels. It is worth mentioning that the efficient CUDA implementation of the three core functions, **FP**, **BP**, and **VoxelUpdate** in Algorithm 2 is accomplished by maximizing GPU utilization in terms of CUDA occupancy [31] along with efficient CUDA memory utilization to handle relatively small number of voxels that can be processed in parallel compared to conjugate gradient-based approach [12], [32], [33].

Algorithm 2: MVU-accelerated ICD-SIR.

G: parallelizable voxel groups (See Section.II-B)
g : a group of parallelizable voxels (See Section.II-B)
K: the number of voxels in a group

```
Procedure MVU_ICD_SIR
    Initialize transaxial footprint (Section. III-A) ;
    for g ∈ G do
        K ← length(g) ;
        /* compute θ1s and θ2s for all K voxels */
        BP(g, K) ;
        /* solve 1D minimization object function of Eq.(5) */
        VoxelUpdate(g, K) ;
        /* update projection error with the new voxel value */
        FP(g, K) ;
    end
end
```

A. Separable Footprint CT Projector

We used the separable footprint (SF) projector with the trapezoid-rectangle function and the A1 amplitude method [3]. A convenient property of the SF is that it is separable into a transaxial- and an axial-footprint as well as the amplitude term. Hence, for an efficient GPU implementation we can pre-compute the transaxial footprint for the given CT geometry to avoid redundant computations during the forward- and back-projections. To do this, we define the transaxial footprint size of a voxel at a given view as follows:

$$\text{foot}_{xy} = 2 \cdot \max_{i=1\ldots4} |s_c - s_i| \,, \tag{14}$$

where s_c is the projected location of a voxel center in the center slice, $(x, y, 0)$, onto the s-axis, using Eq.(9). In a similar fashion, with Δ_x and Δ_y denoting the voxel's size in the x- and y-directions, s_i is one of the four projected voxel corner locations, $(x \pm \Delta_x/2, y \pm \Delta_y/2, 0)$, on the s-axis. The maximum transaxial footprint size, $\text{foot}_{xy}^{\max}$, encompassing the transaxial footprint sizes of all voxels is then determined by applying Eq.(14) for the four corner voxels in the field-of-view for all projection views.

Having fixed the size of the transaxial footprint as $\text{foot}_{xy}^{\max}$ for each voxel, we allocate $(N_\beta \times N_f) \times (N_x \times N_y)$ of global memory to store all of the precomputed transaxial footprint information. Here, N_β is the number of projection views, $N_x \times N_y$ is the number of voxels in a volume slice, and N_f is the length of the footprint information array dedicated for each voxel in the slice. Note that we only need to store the information for one slice since the other slices have identical information. $N_f (= \text{foot}_{xy} + 3)$ is the size of a vector array that contains the transaxial footprint values and three additional pieces of information required for the efficient execution in the projection kernels. This information includes (1) the start index of the transaxial footprint, (2) the value for κ in Eq.(11) that is used for computing the detector bin index along the t-direction, and (3) the part of the A1 amplitude term related to the azimuthal (projection) angle, β (see Eq.(37) in [3]). Note that one can easily adapt this part, and the forward- and back-projection kernels listed in Algorithm 3 and 4 to support distance-driven projector [2] as it also utilizes the separable properties for the efficient GPU implementation [34].

Algorithm 3: BP kernel to compute θ_1 and θ_2.

gF_{xy}: transaxial footprint at (x, y) position $(N_\beta \times N_f)$
idx : voxel index to be updated $(K \times 1)$

Result: $K \times 1$ of $g\theta_1$ and $g\theta_2$ in Eq.(3) and Eq.(4)

```
__global__ void BP_kernel(...)
    __shared__ sF[Blk_y × N_f] ;
    __shared__ sθ1[Blk_x × Blk_y] ;
    __shared__ sθ2[Blk_x × Blk_y] ;
    tx ← threadIdx.x ;
    ty ← threadIdx.y ;
    ivox ← blockIdx.x · Blk_x + tx ;
    iview ← blockIdx.y · Blk_y + ty ;
    /* Step 1: pre-fetch transaxial footprint */
    if (tx < N_f and iview < N_β) then
        sF[tx · Blk_y + ty] ← gF_xy[iview · N_f + tx] ;
    __syncthreads() ;
    /* Step 2: back-project error terms in Eq.(3) and Eq.(4) */
    [θ_1,β, θ_2,β] ← 0 ;
    iz ← N_z ;
    if (ivox < K and iview < N_β) then
        iz ← idx[ivox] ;
        Compute θ_1,β and θ_2,β with SF [3] ;
    sidx ← tx · Blk_y + ty ;
    sθ1[sidx] = θ_1,β ;
    sθ2[sidx] = θ_2,β ;
    __syncthreads() ;
    /* Step 3: parallel reduction within a thread block */
    if ty < 4 then
        sθ1[sidx] = θ_1,β = θ_1,β + sθ1[sidx + 4] ;
        sθ2[sidx] = θ_2,β = θ_2,β + sθ2[sidx + 4] ;
    __syncthreads() ;
    /* repeat above 4 lines until ty < 1 */
    ...
    /* sequential reduction over blocks using atomic operations */
    if (ty = 0 and iz < N_z) then
        atomicAdd (gθ1 + ivox, θ_1,β) ;
        atomicAdd (gθ2 + ivox, θ_2,β) ;
```

B. GPU Memory Management

We store the $N_x \times N_y \times N_z$ volume data in z-major order, that is, the leading dimension is the z-direction. This is advantageous for two reasons: (1) the K voxels in a group (see Algorithm 2) are aligned along the z-axis, and (2) there is less indexing variation in the t-direction (which is aligned with the z-axis) than in the s-direction. Since the volume data are stored in global memory, this storage order will ensure a coalesced memory access pattern and also increase the probability that accesses of voxels in the same group will hit the L2 cache when their values are read or written. For similar reasons, the $N_s \times N_t \times N_\beta$ CT projection data are stored in global memory along the t-direction.

C. Back Projection Kernel

Let us denote Blk_x and Blk_y as the number of voxels and views, respectively, considered in a certain thread block. Hence there are $Blk_x \times Blk_y$ threads in such a block. Note that voxels in each thread block are chosen at appropriate locations along the z-direction since they are independent and have good as cache coherency, as discussed earlier.

Since the voxels in a block all have the same (x, y) coordinate they only require one footprint information array per view. And so, the array portion dedicated for the thread block's Blk_y views can be shared among its Blk_x voxels. To avoid redundant computations, the footprint information is precomputed (see Section III-A) and stored in global memory. To minimize any expensive global memory accesses, the footprint information is first pre-fetched from global memory and then stored in shared memory. Shared memory is as fast as local registers provided there are no bank conflicts [35].

Once the footprint information has been stored in shared memory and is available to all threads in a thread block, each thread computes a part of θ_1 in Eq.(3) and θ_2 in Eq.(4) by back-projecting the projection error, $\mathbf{e}$, weighted by $\mathbf{W}$, onto the thread's target voxel using the separable footprint projector. A thread only computes $\theta_{1,\beta}$ and $\theta_{2,\beta}$ corresponding to a specific view, β, and the results are stored in shared memory. They are then added in a subsequent parallel reduction step (within the thread block) [36] in which only a subset of the block's threads participate. Lastly, once the within-block aggregation phase has completed, the reduction results of the different blocks are sequentially added to global memory using atomic operations.

Algorithm 3 shows the pseudo CUDA code for the back-projection kernel, where we specifically focus on shared memory utilization. When using shared memory it is important to ensure that any updates are visible to all threads in a thread block before the data stored there is actually utilized. This is accomplished by using the CUDA API, **__syncthreads()**, to set a barrier at which all threads in a thread block must wait until all of them reach it. Only then the threads are allowed to proceed [35] to the next task. We note that in our application the branch for parallel reduction in line 22 to 25 starts when $threadIdx.y$ is less than 4 assuming $Blk_y = 8$.

D. Forward Projection Kernel

The CUDA implementation of the forward projection kernel is similar to that of the back projection kernel due to the symmetry of volumetric CT projectors [2]–[4]. The main difference is that while the back projection accumulates the corresponding projection values into a voxel, which is a so-called *gathering operation*, in the forward projection, the voxel value is spread out to the corresponding projections, which is the so-called *scattering operation* [31].

Algorithm 4 shows the pseudo code for the forward projection kernel. In this kernel, the parameters Blk_x and Blk_y are set to the same values as in the back projection kernel. The shared memory stores the K pre-fetched, updated voxel values from global memory as well as the transaxial footprint information. As noted above, the voxels are chosen to have no overlaps in the projection data for all views (i.e. have zero correlation in Eq.(6)), Atomic operations are not needed while projecting the updated voxel values to the error projections.

Algorithm 4: FP kernel to update the error term, $\mathbf{e}$.

$gAtt$: the attenuation to be updated ($K \times 1$)
idx : voxel index to be updated ($K \times 1$)

Result: Updated error projection term, $\mathbf{e}$

```
1  __global__ void FP_kernel(...)
2      __shared__ sF[Blk_y × N_f] ;
3      __shared__ sA[Blk_x];
       /* same as FP for thread preparation                    */
4      ...
       /* same as FP for pre-fetching transaxial footprint     */
5      ...
       /* pre-fetch attenuation                                */
6      if (ivox < K and ty == 0) then
7          sA[tx] = gAtt[ivox];
8      __syncthreads() ;
       /* update projection error terms, e                     */
9      if (ivox ≥ K  and  iview ≥ N_view) then
10         return ;
11     iz ← idx[ivox] ;
12     att ← sA[tx] ;
13     Projecting att with the SF [3] ;
```

E. Voxel Update Kernel

We assume a prior model that operates on a neighborhood of 27 voxels (one center voxel and its 26-connected direct neighbors). Given K voxels that can be updated in parallel, an efficient CUDA kernel implementation for solving Eq.(5) needs to fulfill two key properties. First, given that these voxels will be stored in global memory, we must ensure that their memory access pattern is well coalesced to maximize L2 cache hit rate (see Section III-B). The second requirement is that there should be a sufficient number of warps in the thread blocks in order to hide the long latency incurred for fetching the voxel values from global memory [31]. Algorithm 5 shows the corresponding CUDA pseudo-code we designed to satisfy both of these conditions.

Let us denote Blk_x and Blk_y as the number of threads in a 2D thread block, with $Blk_x \leq K$ being the number of voxels processed in parallel and $Blk_y = 8$ being the number of neighborhood voxels in the xy-plane. Each thread takes responsibility for one of these 8 neighbors (at a given xy-position). It fetches the associated 3 voxels in the z-direction, computes and sums their influences, and adds the result to the overall voxel update using an atomic operation. A thread's x and y offset from the target voxel in the center of its neighborhood is obtained from an array stored in constant memory as $cStepx = [-1, 0, 1, -1, 1, -1, 0, 1]$ and $cStepy = [-1, -1, -1, 0, 0, 1, 1, 1]$, respectively.

After computing the target voxel's index, iz (line $5 - 8$), each thread fetches the three voxel values in the z-direction at its designated xy-offset and stores them in three private

Algorithm 5: Voxel update kernel.

ix, iy : common x and y index of K parallelizable voxels
f_1 : prior model computing influence of neighbor voxels
θ_1, θ_2: the first and second derivative of Eq.(2) ($K \times 1$)
idx : voxel index to be updated ($K \times 1$)
d : updated voxel amount, $\bar{\mathbf{x}}_j - \mathbf{x}_j$ ($K \times 1$)

Result: Updated the object, **x**, and the updated amount, **d**

```
1  __global__ void voxelUpdate_kernel(...)
2      __shared__ att_ctr[Blk_x][3] ;
3      __shared__ influ[Blk_x] ;
4      att_nei[3] ;                          /* private register per thread */
5      tx ← threadIdx.x ;
6      ty ← threadIdx.y ;
7      ivox ← blockIdx.x · Blk_x + tx ;
8      iz ← idx[ivox] ;
       /* pre-fetching attenuation                                          */
9      sx ← cStepx[ty] ;        /* cStepx(y) is in constant memory */
10     sy ← cStepy[ty] ;
11     att_nei[1] ← x[iz + 1][ix + sx][iy + sy] ;
12     att_nei[2] ← x[iz + 0][ix + sx][iy + sy] ;
13     att_nei[3] ← x[iz − 1][ix + sx][iy + sy] ;
14     if (ty == 0) then
15         att_ctr[tx][1] ← x[iz + 1][ix][iy] ;
16         att_ctr[tx][2] ← x[iz + 0][ix][iy] ;
17         att_ctr[tx][3] ← x[iz − 1][ix][iy] ;
18     __syncthreads() ;
19     x_j ← att_ctr[tx][2] ;
       /* compute influence of three neighbors                             */
20     for i = 1 to 3 do
21         δ ← x_j − att_nei[i] ;
           /* atomic operations in shared memory                           */
22         atomicAdd (influ + tx, f_1(δ)) ;
23     if (ty == 0) then
           /* same as above for two neighbors at (ix, iy, iz ± 1)          */
24     __syncthreads() ;
       /* update attenuation                                               */
25     if (ty == 0) then
26         x̄_j ← max(0, influ[tx] + θ_2[ivox] · x_j − θ_1[ivox]) ;
27         x[iz][ix][iy] ← x̄_j ;
28         d[ivox] ← x̄_j − x_j ;
```

registers, att_nei (line 9 - 13). The central voxel and its two z-neighbors are taken care of by threads with $threadIdx.y = 0$ and stored as a $3D$-array (for each voxel) in shared memory (line 14 - 17). The center of that array is the target voxel, x_j (line 19).

With all required data fetched from global memory and put into the right locations, the thread computes and adds the influences of its three z-connected voxels using the prior model f_1 (line $20 - 23$). After all 27 neighbor (and own) influence contributions have been collaboratively added by the 9 threads associated with the target voxel (barrier in line 24), the $ty = 0$ thread solves the 1D minimization problem in Eq.(5) with the θ_1 (Eq.(3) and θ_2 (Eq.(4) factors stored in the corresponding arrays filled in the back-projection kernel. We note that although we use a 3-D indexing scheme for the volume data, **x**, in Algorithm 5 for brevity, in the actual implementation the data is flattened and stored in a 1-D array. A detailed 1-D indexing scheme is discussed in Section IV.

IV. Results

We begin by comparing the reconstruction quality achieved with conventional filtered back-projection (FBP) and with our ICD-SIR using the multi-voxel update (MVU) scheme. For these reconstructions, we obtained 391 X-Ray projections over 360° by scanning a lumbar region with 20 cm field-of-view using a Medtronic O-arm O2 surgical imaging system under low-dose conditions (120 kVp, 40 mA). The system has a source-detector distance of 1147.7 mm and a source-axis distance of 647.7 mm. Its X-ray flat detector has $N_x \times N_t = 1024 \times 386$ detector bins with a bin size of $\Delta_s \times \Delta_t = 0.384 \times 0.755$ mm^2. All reconstructions had a resolution of $N_x \times N_y \times N_z = 512 \times 512 \times 512$ with a voxel size of $\Delta_x \times \Delta_y \times \Delta_z = 0.415 \times 0.415 \times 0.3$ mm^3. Fig. 3 shows an example slice of the 3-D volumes we reconstructed using FBP and ICD-SIR with both the SVU and the MVU schemes. For ICD-SIR, the reconstructed volume was obtained after 4 iterations which took about 2.5 minutes for MVU and about 10.5 hours for SVU, in total. We found that the MVU-based version exhibited a similar convergence rate than the conventional ICD-SIR method using SVU scheme. We also observe that the reconstruction obtained with the SVU scheme is virtually indistinguishable from the one generated with our MVU scheme. Fig. 3 confirms what has already been amply demonstrated in the literature [6], [8], [11], [22], [37], namely that ICD-SIR achieves significantly better reconstruction quality with less noise and CT artifacts (including streaks and beam-hardening effect) than FBP.

We also observe that the reconstruction obtained with the conventional SVU scheme is virtually indistinguishable from the one generated with our MVU scheme.

In the following, we will present results that are related to the main premise of this paper – accelerating ICD-SIR. We will compare the time performance of the proposed multi-voxel update scheme with that of the conventional single-voxel update scheme within a GPU-accelerated ICD-SIR framework. To gain further insight we will also present an analysis of the three CUDA kernels listed in Algorithm 3 to 5.

A. Analysis of CUDA Kernels in ICD-SIR

We first conducted an experiment to see how data arrangement affects the time performance of the three CUDA kernels listed in Algorithm 3–5. First, for both the forward- and back-projection kernels, the 2-D projection data were arranged in a 1-D array in two different ways: (1) with the s-axis as the leading dimension (called ST order), and (2) in the transposed arrangement (called TS order). In ST order, the 1-D index corresponding to the location (is, it) in the 2-D projections is:

$$\text{index}_{\text{ST}} = it \times N_s + is \,, \tag{15}$$

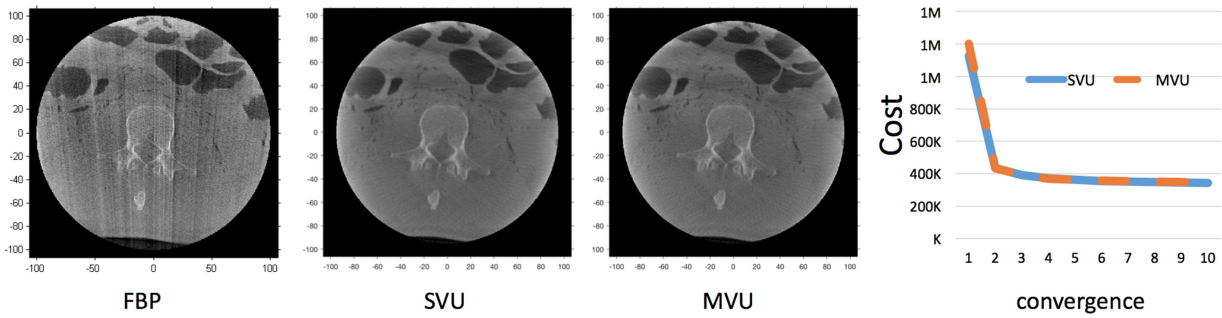

Fig. 3. Comparison of a corresponding slice taken from volumes reconstructed with FBP as well as with the two ICD-SIR schemes - the conventional SVU and our MVU each after 4 iterations. The visual quality obtained with our MVU scheme is virtually identical to that obtained with the conventional SVU scheme and the convergence curves are also very similar.

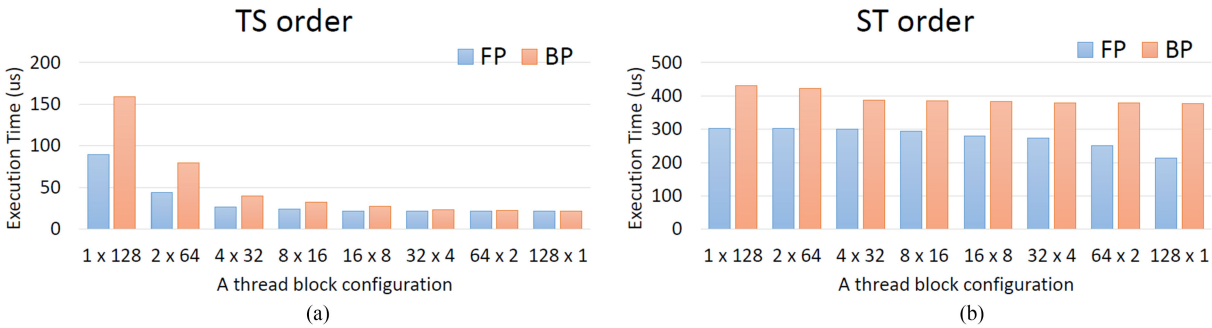

Fig. 4. Projection kernel performance with different thread block configurations and projection data arrangements. (a) TS order. (b) ST order.

and, for the TS order, the 1-D index becomes

$$\text{index}_{\text{TS}} = is \times N_t + it\,. \tag{16}$$

Here, N_s and N_t are the dimension sizes of the projection data on the s- and t-axis, respectively. In addition to the data arrangement, we also investigated the thread block size for the projection kernels. Let us assume a threat block size of $Blk_x \times Blk_y$. where Blk_x is the number of voxels and Blk_y is the number of views assigned to the thread block.

On recent NVIDIA GPU Pascal architectures [38], a multiprocessor (MP) can have at most 64 concurrent warps (2,048 threads/MP = 64 warps × 32 threads/warp) which can be distributed in maximal 16 concurrent thread blocks. To fully utilize the compute resources in a MP, a preferred CUDA kernel configuration would be 16 blocks with 128 threads each (2,048 threads/MP = 16 blocks × 128 threads/block). Specifically, in order to fully utilize the GTX 1070 processors, one would want at least 240 blocks with 128 threads each (240 blocks = 16 blocks/MP × 15 MPs). Therefore, throughout our experiments we fixed the total number of threads in a thread block to 128 in order to maximize the computational occupancy of all multiprocessors on the NVIDIA GTX 1070 GPU we used.

Fig. 4 shows the experimental results we obtained by measuring kernel execution time averaged over 1000 samples. In each sample, 256 voxels aligned in the z-direction were simultaneously forward- and back-projected over 391 projection views and the xy coordinates of these voxel locations were randomly chosen. The trans-axial footprint size in Eq.(14) was 5. We

TABLE I
DATA ARRANGEMENT AND TIME PERFORMANCE

	order	Time(μs)	speed-up
FP	ST	212.841	n/a
	TS	21.412	9.94
BP	ST	377.162	n/a
	TS	21.322	17.68
VU	XYZ	3.899	n/a
	ZXY	2.838	1.37

readily observe that the TS order offers better performance than the ST-order, which confirms our initial thoughts. The better time performance occurs for both forward- and back-projection and for all thread block configurations. For both ordering schemes, the best time performance is observed when all threads in a block process voxels for the same view (i.e. 128×1). On the thread block level, this is due to (1) the reduced amount of global memory accessing for fetching pre-computed trans-axial footprint information, (2) the increased L2 cache hit rate for the projection data, and (3) a smaller number of thread barriers (only for the back-projection kernel). This results in about 10 times better performance for forward projection and about 17.6 times for back-projection than the TS order. Table I summarizes the time performance for the forward- and back-projection kernels with a 128×1 thread block size. It is worth noting that for all experiment settings in Fig. 4 the achieved thread occupancy (measured using NVIDIA Nsight [39]) was between 90 to 98%.

For the voxel update (VU) kernel listed in Algorithm 5 we used a fixed thread block size of 16×8 such that 8 threads in a

TABLE II
TIME PERFORMANCE COMPARISONS BETWEEN MVU AND SVU

# slices	16	32	64	128	256	512
K	8	16	32	64	128	256
MVU (min/iter)	0.36	0.36	0.36	0.37	0.57	0.63
SVU (min/iter)	2.8	5.7	11.5	23.1	46.1	90.8
Actual gain	7.9	16.0	31.9	61.3	80.4	143.4
Ideal gain	8	16	32	64	128	256

thread block collaborated to solve the 1-D minimization problem in Eq.(5) for the given voxel. We measured the average execution time for the VU kernel in a similar fashion than for the projection kernels, by varying the 3-D volume data arrangement in two different ways. One way was to store the volume data in a 1-D array with the leading dimension being the x-axis (called XYZ order), such that the indexing for a voxel at (ix, iy, iz) becomes:

$$\text{index}_{\text{XYZ}} = ix + N_x \times (iy + N_y \times iz)\,, \tag{17}$$

The other ordering was with the z-axis being the leading dimension (called ZXY order). Here the indexing is:

$$\text{index}_{\text{ZXY}} = iz + N_z \times (ix + N_x \times iy)\,. \tag{18}$$

As shown in Table I, the ZXY order shows about 1.37 times better performance than the XYZ order since it has a slightly better cache locality for the voxels along the z-direction.

B. Execution Time Comparison Between SVU and MVU

We compared the time performance of the multi-voxel update (MVU) scheme with that of the single-voxel update (SVU) scheme by varying the number of volume slices from 16 to 512 while keeping the number of voxels in the xy-plane fixed as 512×512. For each case, we found that the number of parallelizable voxels, K, was about half of the number of z-slices. We note that the GPU-accelerated SVU is just a special case of MVU when $K = 1$. The execution times of MVU and SVU were measured by averaging over 10 iterations in the ICD-SIR framework. The speedup was computed by dividing SVU execution time by MVU execution time. Table II shows the results. We observe that MVU shows speedups of one and two orders of magnitude compared to SVU and we also observe that the speedup increases as the number of slices grows, which can be expected. However, the real speedup drifts apart from the ideal speedup, which is the number of parallelizable voxels in each case. This is largely because of the increased cache miss rate when reading and writing the projection data during forward- and back-projection operations. With an increasing number of slices, the projection data corresponding to the K parallelizable voxels is increasingly distributed more irregularly due to the increasing spread and divergence of the cone-beam. This results in the gap with respect to the ideal speedup.

On the other hand, MVU shows a much slower growth in execution time than SVU as the number of slices increases. To observe this behavior, we measured the average execution time from 50 slices to 500 slices, each over 50 intervals, and normalized the execution time by that obtained with 50 slices (i.e. the smallest execution time). Fig. 5 shows the normalized execution time for both MVU and SVU. While both schemes grow linearly, the approximated slopes estimated using the two end points of the curves are $0.0015 \approx (1.68 - 1)/(500 - 50)$ for MVU and $0.02 \approx (10.13 - 1)/(500 - 50)$ for SVU. We observe that SIR-ICD execution time with MVU grows about $13.3 \approx 0.02/0.0015$ times slower than SVU.

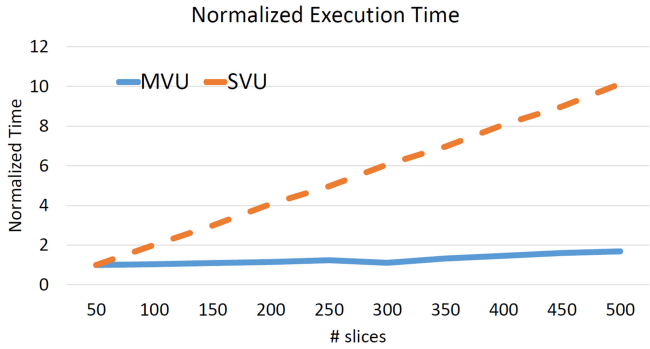

Fig. 5. Normalized execution time comparison.

V. CONCLUSION

In this paper, we have introduced a conflict-free multi-voxel update (MVU) scheme to accelerate the ICD-based SIR framework for general viewing geometries – we focused on cone-beam with a circular trajectory in this work. Unlike conventional ICD-based SIR which updates one voxel at a time, the proposed method updates multiple voxels in parallel. Since the parallelizable voxels are selected to have zero correlation with each other, our method does not introduce any additional complexities within the conventional ICD-based SIR framework and so does not compromise robustness and rate of convergence. Using a modern commodity GPU platform, we show that our MVU scheme can gain speedups of one to two orders of magnitude, compared to a traditional single-voxel update (SVU) scheme.

To achieve these high speedups, for example, about x140 for reconstructing a clinically relevant $512 \times 512 \times 512$ volume from 391 cone-beam projections, we have carefully tuned the CUDA code, the memory access patterns for both projection and volume data, and the CUDA thread block configurations for the maximum occupancy. However, despite these efforts the achieved gain is about x0.56 ($\approx 143.4/256$) behind the ideal gain. There are two reasons for this. First, it is due to the accumulated overhead incurred by kernel launch time which can prevent achieving the ideal gain for small number of slices. Secondly, more importantly, we found that the overall arithmetic pipe utilization (measured by NVIDIA Nsight) show around 60 % indicating the CUDA kernels are inherently compute-bound [39].

To get speed-up gain close to the ideal one, we will first extend our present single GPU approach to support a multi-GPUs platform such that the computational burden can be shared among multiple GPUs. One straightforward way would be to divide 256 parallelizable voxels into four groups (i.e. 64 voxels per group) and processing them on 4 GPUs in parallel because 64 voxels seem a good number to process in a single GPU as shown in

Table II. We will also explore possible ways to include loosely coupled voxels in the z-direction as presented in the ABCD algorithm [28] (but adding extract computational complexity) or in the xy-plane in [25], [27], [40] (but tested only for parallel beam geometry). Furthermore, we would also like to explore our own more general voxel set identification scheme [29] in a helical path with different parameter settings including pitch, $\mathbf{d}_{sad}$, and $\mathbf{d}_{add}$ which might have a non-negligible impact on the number of voxels that can be processed in parallel. Finally, in future research we would also like to find a effective method that includes a non-homogeneous update scheme[11]. The adaptive voxel selection and update might lead to an unbalanced GPU workload in each iteration step within the MVU framework. We suspect that there will be trade-offs similar to those appearing when GPU-accelerating ordered subsets methods [41]. A solution will hence need to address two concurrent objectives: (1) improving convergence rate, and (2) maintaining fast reconstruction performance with the GPU.

ACKNOWLEDGMENT

The authors would like to thank Medtronic, Inc., for the data and support.

REFERENCES

[1] R. L. Siddon, "Fast calculation of the exact radiological path for a three-dimensional CT array," *Med. Phys.*, vol. 12, no. 2, pp. 252–255, 1985.
[2] B. De Man and S. Basu, "Distance-driven projection and backprojection in three dimensions," *Phys. Med. Biol.*, vol. 49, no. 11, pp. 2463–2475, 2004.
[3] Y. Long, J. A. Fessler, and J. M. Balter, "3-D forward and back-projection for X-ray CT using separable footprints," *IEEE Trans. Med. Imag.*, vol. 29, no. 11, pp. 1839–1850, Nov. 2010.
[4] S. Ha, H. Li, and K. Mueller, "Efficient area-based ray integration using summed area tables and regression models," in *Proc. 4th Int. Meeting Image Formation X-ray CT*, 2016, pp. 507–510.
[5] J.-B. Thibault, C. A. Bouman, K. D. Sauer, and J. Hsieh, "A recursive filter for noise reduction in statistical iterative tomographic imaging," in *Proc. Electron. Imag.*, 2006, pp. 60650X-1–60650X-10.
[6] J.-B. Thibault, K. D. Sauer, C. A. Bouman, and J. Hsieh, "A three-dimensional statistical approach to improved image quality for multislice helical CT," *Med. Phys.*, vol. 34, no. 11, pp. 4526–4544, 2007.
[7] H. Zhang, J. Ma, J. Wang, Y. Liu, H. Lu, and Z. Liang, "Statistical image reconstruction for low-dose CT using nonlocal means-based regularization," *Comput. Med. Imag. Graph.*, vol. 38, no. 6, pp. 423–435, 2014.
[8] C. Bouman and K. Sauer, "A generalized Gaussian image model for edge-preserving map estimation," *IEEE Trans. Image Process.*, vol. 2, no. 3, pp. 296–310, Jul. 1993.
[9] V. Panin, G. L. Zeng, and G. Gullberg, "Total variation regulated EM algorithm [SPECT reconstruction]," *IEEE Trans. Nucl. Sci.*, vol. 46, no. 6, pp. 2202–2210, Dec. 1999.
[10] D. Geman and G. Reynolds, "Constrained restoration and the recovery of discontinuities," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 14, no. 3, pp. 367–383, Mar. 1992.
[11] Z. Yu, J.-B. Thibault, C. A. Bouman, K. D. Sauer, and J. Hsieh, "Fast model-based X-ray CT reconstruction using spatially nonhomogeneous ICD optimization," *IEEE Trans. Image Process.*, vol. 20, no. 1, pp. 161–175, Jan. 2011.
[12] J. A. Fessler and S. D. Booth, "Conjugate-gradient preconditioning methods for shift-variant pet image reconstruction," *IEEE Trans. Image Process.*, vol. 8, no. 5, pp. 688–699, May 1999.
[13] H. Erdogan and J. A. Fessler, "Ordered subsets algorithms for transmission tomography," *Phys. Med. Biol.*, vol. 44, no. 11, pp. 2835–2851, 1999.
[14] F. Xu and K. Mueller, "Accelerating popular tomographic reconstruction algorithms on commodity PC graphics hardware," *IEEE Trans. Nucl. Sci.*, vol. 52, no. 3, pp. 654–663, Jun. 2005.
[15] F. Xu and K. Mueller, "Real-time 3-D computed tomographic reconstruction using commodity graphics hardware," *Phys. Med. Biol.*, vol. 52, no. 12, pp. 3405–3419, May 2007.
[16] B. De Man *et al.*, "A study of four minimization approaches for iterative reconstruction in X-ray CT," in *Proc. IEEE Nucl. Sci. Symp. Conf. Rec.*, 2005, vol. 5, pp. 2708–2710.
[17] C. Kamphuis and F. J. Beekman, "Accelerated iterative transmission CT reconstruction using an ordered subsets convex algorithm," *IEEE Trans. Med. Imag.*, vol. 17, no. 6, pp. 1101–1105, Dec. 1998.
[18] D. Kim, D. Pal, J.-B. Thibault, and J. A. Fessler, "Accelerating ordered subsets image reconstruction for X-ray CT using spatially nonuniform optimization transfer," *IEEE Trans. Med. Imag.*, vol. 32, no. 11, pp. 1965–1978, Nov. 2013.
[19] D. Kim, S. Ramani, and J. A. Fessler, "Combining ordered subsets and momentum for accelerated X-ray CT image reconstruction," *IEEE Trans. Med. Imag.*, vol. 34, no. 1, pp. 167–178, Jan. 2015.
[20] H. Nien and J. A. Fessler, "Fast X-ray CT image reconstruction using a linearized augmented Lagrangian method with ordered subsets," *IEEE Trans. Med. Imag.*, vol. 34, no. 2, pp. 388–399, Feb. 2015.
[21] H. Nien and J. A. Fessler, "Relaxed linearized algorithms for faster X-ray CT image reconstruction," *IEEE Trans. Med. Imag.*, vol. 35, no. 4, pp. 1090–1098, Apr. 2016.
[22] K. Sauer and C. Bouman, "A local update strategy for iterative reconstruction from projections," *IEEE Trans. Signal Process.*, vol. 41, no. 2, pp. 534–548, Feb. 1993.
[23] J. A. Fessler, E. P. Ficaro, N. H. Clinthorne, and K. Lange, "Grouped-coordinate ascent algorithms for penalized-likelihood transmission image reconstruction," *IEEE Trans. Med. Imag.*, vol. 16, no. 2, pp. 166–175, Apr. 1997.
[24] T. M. Benson, B. K. De Man, L. Fu, and J.-B. Thibault, "Block-based iterative coordinate descent," in *Proc. IEEE Nucl. Sci. Symp. Conf. Rec.*, 2010, pp. 2856–2859.
[25] X. Wang, A. Sabne, S. Kisner, A. Raghunathan, C. Bouman, and S. Midkiff, "High performance model based image reconstruction," in *Proc. 21st ACM SIGPLAN Symp. Princ. Pract. Parallel Program.*, 2016, pp. 1–12.
[26] A. Sabne, X. Wang, S. J. Kisner, C. A. Bouman, A. Raghunathan, and S. P. Midkiff, "Model-based iterative CT image reconstruction on GPUs," in *Proc. 22nd ACM SIGPLAN Symp. Princ. Pract. Parallel Program.*, 2017, pp. 207–220.
[27] X. Wang, K. A. Mohan, S. J. Kisner, C. Bouman, and S. Midkiff, "Fast voxel line update for time-space image reconstruction," in *Proc. IEEE Int. Conf. Acoust., Speech, Signal Process.*, 2016, pp. 1209–1213.
[28] J. A. Fessler and D. Kim, "Axial block coordinate descent (ABCD) algorithm for X-ray CT image reconstruction," in *Proc. Fully 3D Image Reconstruction Radiol. Nucl. Med.*, 2011, pp. 262–265.
[29] S. Ha and K. Mueller, "An algorithm to compute independent sets of voxels for parallelization of ICD-based statistical iterative reconstruction," in *Proc. 13th Int. Meeting Fully 3D Image Reconstruction Radiol. Nucl. Med.*, 2015, pp. 1–4.
[30] C. A. Bouman and K. Sauer, "A unified approach to statistical tomography using coordinate descent optimization," *IEEE Trans. Image Process.*, vol. 5, no. 3, pp. 480–492, Mar. 1996.
[31] S. Ha, S. Matej, M. Ispiryan, and K. Mueller, "GPU-accelerated forward and back-projections with spatially varying kernels for 3D direct TOF pet reconstruction," *IEEE Trans. Nucl. Sci.*, vol. 60, no. 1, pp. 166–173, Feb. 2013.
[32] N. H. Clinthorne, T.-S. Pan, P.-C. Chiao, W. L. Rogers, and J. A. Stamos, "Preconditioning methods for improved convergence rates in iterative reconstructions," *IEEE Trans. Med. Imag.*, vol. 12, no. 1, pp. 78–83, Mar. 1993.
[33] E. U. Mumcuoglu, R. Leahy, S. R. Cherry, and Z. Zhou, "Fast gradient-based methods for Bayesian reconstruction of transmission and emission PET images," *IEEE Trans. Med. Imag.*, vol. 13, no. 4, pp. 687–701, Dec. 1994.
[34] R. Liu, L. Fu, B. De Man, and H. Yu, "GPU-based branchless distance-driven projection and backprojection," *IEEE Trans. Comput. Imag.*, vol. 3, no. 4, pp. 617–632, Dec. 2017.
[35] "CUDA C programming guide," NVIDIA Corporation, 2017. [Online]. Available: https://docs.nvidia.com/cuda/cuda-c-programming-guide
[36] H. Mark, "Optimizing parallel reduction in CUDA," NVIDIA CUDA SDK, NVIDIA Santa Clara, CA, USA, Tech. Rep., 2008, vol. 2.

[37] H. Zhang *et al.*, "Statistical image reconstruction for low-dose CT using nonlocal means-based regularization. Part II: An adaptive approach," *Comput. Med. Imag. Graph.*, vol. 43, pp. 26–35, 2015.

[38] "Tuning application for Pascal," Wikipedia, 2017. [Online]. Available: http://docs.nvidia.com/cuda/pascal-tuning-guide/index.html#axzz4gUxXGfzn

[39] C. Angerer and J. Progsch, "CUDA optimization with NVIDIA Nsight visual studio edition," 2016. [Online]. Available: http://on-demand.gputechconf.com/gtc/2016/presentation/s6112-angerer-cuda-nsight-vse.pdf

[40] M. Wu and J. A. Fessler, "GPU acceleration of 3D forward and backward projection using separable footprints for X-ray CT image reconstruction," in *Proc. Fully 3D Image Reconstruction Radiol. Nucl. Med.*, 2011, vol. 6, Art. no. 021911.

[41] F. Xu *et al.*, "On the efficiency of iterative ordered subset reconstruction algorithms for acceleration on GPUs," *Comput. Methods Programs Biomed.*, vol. 98, no. 3, pp. 261–270, 2010.

Authors' photographs and biographies not available at the time of publication.