Model-based Iterative CT Image Reconstruction on GPUs

Amit Sabne
Microsoft *
amsabne@microsoft.com

Xiao Wang
School of Electrical and Computer Engineering, Purdue University
wang1698@purdue.edu

Sherman Kisner
High Performance Imaging LLC
kisner@ecn.purdue.edu

Charles Bouman
School of Electrical and Computer Engineering, Purdue University
bouman@purdue.edu

Anand Raghunathan
School of Electrical and Computer Engineering, Purdue University
raghunathan@purdue.edu

Samuel Midkiff
School of Electrical and Computer Engineering, Purdue University
smidkiff@ecn.purdue.edu

Abstract

Computed Tomography (CT) Image Reconstruction is an important technique used in a variety of domains, including medical imaging, electron microscopy, non-destructive testing and transportation security. Model-based Iterative Reconstruction (MBIR) using Iterative Coordinate Descent (ICD) is a CT algorithm that produces state-of-the-art results in terms of image quality. However, MBIR is highly computationally intensive and challenging to parallelize, and has traditionally been viewed as impractical in applications where reconstruction time is critical. We present the first GPU-based algorithm for ICD-based MBIR. The algorithm leverages the recently-proposed concept of SuperVoxels [1], and efficiently exploits the three levels of parallelism available in MBIR to better utilize the GPU hardware resources. We also explore data layout transformations to obtain more coalesced accesses and several GPU-specific optimizations for MBIR that boost performance. Across a suite of 3200 test cases, our GPU implementation obtains a geometric mean speedup of 4.43X over a state-of-the-art multi-core implementation on a 16-core iso-power CPU.

Categories and Subject Descriptors I.4.5 [*Image Processing and Computer Vision*]: Reconstruction—Transform methods; D.1.3 [*Programming Techniques*]: Concurrent Programming—Parallel Programming

Keywords Computed Tomography, Model Based Iterative Reconstruction, Iterative Coordinate Descent, Graphics Processing Units

* Author conducted this work while at Purdue University

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.

PPoPP '17, February 04-08, 2017, Austin, TX, USA
© 2017 ACM. ISBN 978-1-4503-4493-7/17/02…$15.00
DOI: http://dx.doi.org/10.1145/3018743.3018765

1. Introduction

Computed tomography (CT) is an imaging technique that makes it possible to visualize the internals of an object (including the human body) from a limited number of lower-dimensional projections. Model-based iterative reconstruction, or *MBIR* [2, 3], is a CT technique in which a model based upon the geometry of the equipment is used to perform the reconstruction. MBIR, and more generally regularized iterative reconstruction methods, are known to produce better quality images compared to the alternative class of direct methods, which are commonly referred to as *filtered back projection* (FBP) [4, 5, 6, 7, 2, 3, 8]. Unfortunately, MBIR can require up to two orders of magnitude more compute operations than FBP, and is therefore regarded as impractical in scenarios where reconstruction time is paramount.

The MBIR method we focus on in this work is based on an Iterative Coordinate Descent (ICD) algorithm [2, 9, 10]. This algorithm iteratively updates voxels (3D-pixel) in the reconstructed image. A voxel update requires reads and writes of data that are shared among multiple voxels, and so the ICD algorithm does not exhibit explicit data parallelism. Fortunately, the iterative nature of the algorithm makes it error resilient, and a careful choice of voxels allows them to be updated in parallel [11, 12].

We are unaware of any previous effort to port ICD-based MBIR to GPUs. We attribute this to the stringent requirements GPU architectures impose to obtain high performance, such as the need for a high degree of parallelism, coalesced memory accesses, and minimal synchronization and control divergence.

The first challenge in porting MBIR to GPUs is parallelism exploitation. Broadly speaking, there are two key levels of parallelism in MBIR: the computation of a single voxel and the computation of different voxels in parallel. The former involves reduction-style parallelism and the

latter requires updating shared data structures; hence, it is fair to say that MBIR lacks explicit, GPU-friendly data parallelism. The state-of-the-art parallel MBIR algorithm exploits inter-voxel parallelism on multi-core CPUs [1]. However, the CPU algorithm employs a much lower degree of parallelism than is needed to exploit the thousands of cores that GPUs offer. Restricting the synchronization cost in the presence of high thread counts is a challenge. The first contribution of this paper is a parallel algorithm that exploits multiple levels of parallelism on the GPU so as to achieve high GPU utilization.

The second challenge results from the irregular memory access pattern. MBIR is known to be a memory bandwidth intensive algorithm. Unfortunately, the memory accesses in MBIR are inherently irregular. Each voxel accesses locations within a 2D array in a sinusoidal pattern (Fig. 1), greatly limiting the caching benefits. Past research [1] has shown that grouping neighboring voxels into larger chunks, called SuperVoxels (*SV*s), and copying the data for a SuperVoxel into a dedicated buffer, called an SVB, can greatly improve cache locality and prefetching benefits on CPUs. GPUs, however, do not benefit much from automatic caching because the cache size per thread is very small. Moreover, GPUs lack hardware prefetching. The second contribution of this paper is a data layout transformation that obtains coalesced accesses for MBIR on GPUs. This transformation involves padding parts of the data, and obtains higher performance, despite performing additional computations, due to significantly improved GPU compute efficiency.

Since MBIR is a memory-bandwidth limited algorithm, the third contribution of this paper is a set of GPU-specific optimizations to improve memory bandwidth. First, a large number of active threads are required to utilize memory bandwidth. However, the MBIR GPU kernel requires a high register count per thread, limiting the number of active GPU threads. To overcome this issue, we migrate thread-local variables from registers into the GPU shared memory (on-chip programmer-managed L1 cache), and thereby obtain higher multi-threading. We also explore the use of the texture cache and improve the bandwidth of L2 cache accesses.

Although we demonstrate an ICD-based MBIR implementation on GPUs, the aforementioned issues and solutions are characteristic of ICD based optimization methods, which are used in a broader range of applications [1].

In summary, the contributions of this paper are as follows:

- This paper presents GPU-ICD, the first GPU algorithm for ICD-based MBIR, which exploits three levels of parallelism : i) inside a voxel's computation, ii) across multiple voxels inside an SV, and iii) across multiple SVs.
- The paper proposes data layout transformations that substantially increase coalesced memory accesses.
- The paper proposes a set of GPU-specific optimizations for MBIR.

With the above contributions, the GPU implementation achieves a geometric mean speedup of 4.43X over the state-of-the-art parallel implementation on an iso-power 16-core CPU [1].

The rest of this paper is organized as follows. Section 2 describes CT, the MBIR method, and the fastest-known CPU implementation. It also provides background on the architectural and programming model for GPUs. Section 3 describes our approach of mapping MBIR parallelism to GPUs, and presents the corresponding algorithm. Section 4 details our data layout transformations and GPU-specific optimizations. Section 5 describes experimental results that evaluate the proposed GPU implementation of MBIR. Section 6 discusses how GPU-ICD can be applied to a broader range of applications. Section 7 discusses related work, while Section 8 concludes the paper.

2. Background

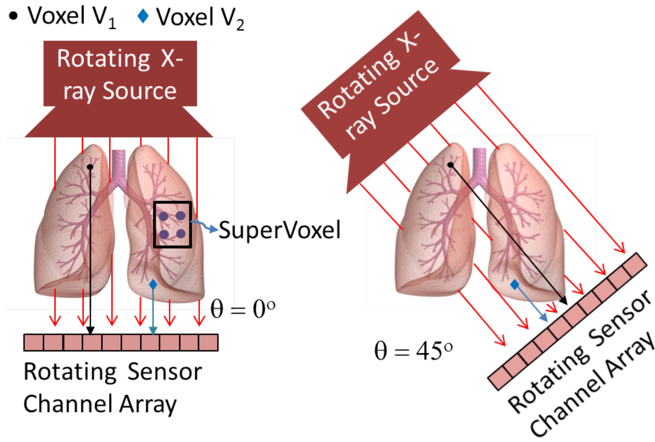
This section begins with a brief description of the CT process. Next, it presents an outline of the fastest CPU implementation. The section ends with an overview of the GPU architecture and programming.

Fig. 1a illustrates a simple representative parallel beam X-ray CT apparatus. An X-ray source and a sensor array that detects X-rays are mounted on a rotating gantry, while the object to be imaged remains stationary. For each view angle, or θ, the recorded measurements from the sensors (which represent the attenuated X-rays that have passed through the object) are stored in a single column of a data structure, called the *sinogram* (Fig. 1b). Different sensors pick up data for different sets of voxels according to the position of the projection rays terminating on each sensor at each θ.

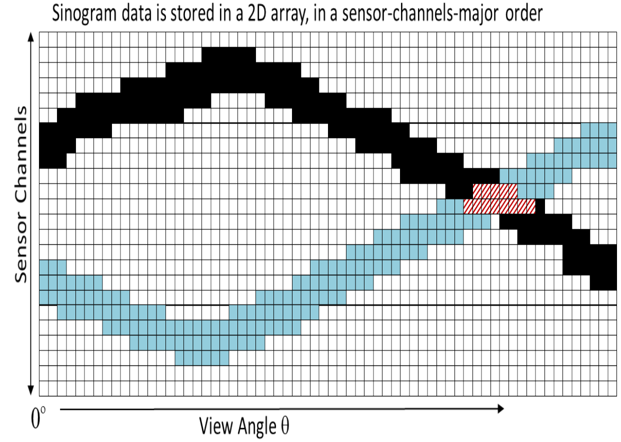
2.1 Computed Tomography and MBIR

Fig. 1b shows the captured measurements for the voxels V_1 and V_2 for different values of θ. The trajectories of the voxels in the sinogram show a sinusoidal pattern. The objective of MBIR is to reconstruct the object from the sinogram data.

Alg. 1 shows the mechanism to update a single voxel, which forms the core of ICD [2, 9, 10], the algorithm traditionally used in MBIR. The algorithm computes two values, theta1 and theta2. The error sinogram, e, represents $y - Ax$, where y is the measurement sinogram, x is the image to be reconstructed, and A represents the system matrix that encodes the CT equipment geometry. Intuitively, the A matrix entries capture the radio density incident upon a voxel, which is proportional to the distance from the X-ray source. The weighing matrix, w, contains the inverse variance of the scanner noise. The theta1 and theta2 values are then used to find the new value of the voxel v, which belongs to x. Finally, the error sinogram is updated to reflect the changed value of v. Alg. 1 repeatedly updates voxels until convergence is reached. Faster convergence is achieved by updat-



(a) CT Equipment Operation : Different sensors may collect data belonging to a given voxel, depending upon θ



***(b) Data collected (sinogram) for voxels V_1 and V_2 :** Only a partial sinogram is shown here*

Figure 1: Computed tomography geometry and measurement data organization

---

Algorithm 1: Updating Single Voxel (Foundation of all ICD-based techniques)

Input: v - the voxel value to be updated, e- error sinogram, w - weights, A - system matrix representing the CT equipment geometry

Output: v - updated voxel value, e - updated error sinogram

```
1  theta1=0;
2  theta2=0;
   // Calculate theta1, theta2
3  foreach i ∈ θ do
4  |  foreach (j ∈ sensors containing measurements for the
   |    voxel at angle i) do
5  |  |  theta1 += -w_ij × A[voxel number]_ij × e_ij;
6  |  |  theta2 += w_ij × A[voxel number]^2_ij;
   // func is computationally inexpensive
7  delta ← func(v, theta1, theta2, neighbors of v);
8  v ← v + delta; // update the voxel
   // Update error sinogram
9  foreach i ∈ θ do
10 |  foreach (j ∈ sensors containing measurements for the
   |    voxel at angle i) do
11 |  |  e_ij -= A[voxel number]_ij × delta;
```

---

ing voxels in a randomized order [13] and by *zero-skipping*, *i.e.*, skipping voxels whose values, along with all of their neighbors, are zero.

2.2 Parallel MBIR on the CPU

Several challenges arise in obtaining high performance from the ICD algorithm. We now describe the state-of-the-art adaptation of ICD, Parallel SuperVoxel ICD (PSV-ICD) [1] which was proposed to improve performance on multi-core CPUs.

The sinusoidal access pattern (Fig. 1b) makes caching and prefetching ineffective. Because neighboring voxels access access neighboring sinogram data, PSV-ICD groups together neighboring voxels into a *SuperVoxel* (SV). PSV-ICD then creates an explicit buffer, the SuperVoxel buffer (SVB), for each SV and copies the SV's sinogram data into it. This improves locality of the data, and increases the linearity of accesses, which makes prefetching more effective. Fig. 2 shows a SuperVoxel, and it also displays the corresponding sinogram accesses for the SuperVoxel. The weights also display the sinusoidal access pattern and are allocated in a similar data structure. When appropriately sized, SVBs can often fit in the CPU's L1 cache.

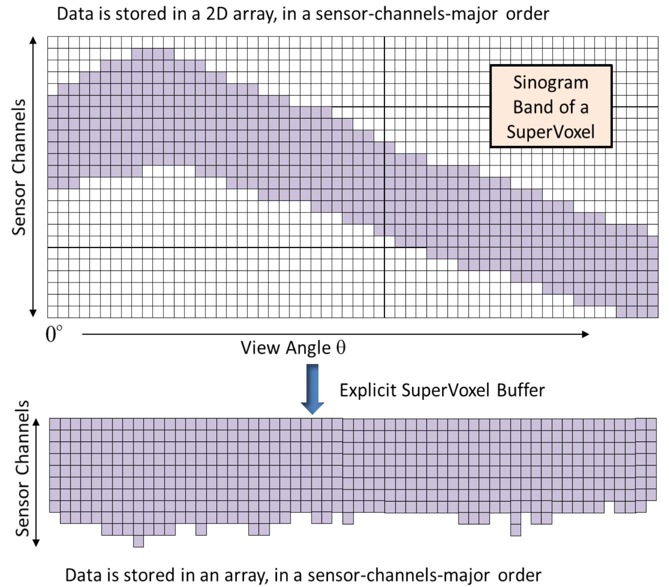


***Figure 2: SuperVoxel buffer (SVB):** linearizes data accesses to a great degree, achieving better cache locality and prefetching on CPUs*

Multiple voxels can potentially be simultaneously updated using multi-core parallelism; however, because of the sinusoidal access pattern, any two voxels are bound to par-

Algorithm 2: PSV-ICD Algorithm

```
Input: A - system geometry matrix, e - error sinogram, w -
       weight sinogram
Output: Converged reconstructed image
1  Create SVs;
2  iter = 1;
3  while image not converged do
4    if iter == 1 then
5      setOfSVsToUpdate ← all SVs;
6    else if iter is even then
7      setOfSVsToUpdate ← top 20% SVs by the update
         amount;
8    else
9      setOfSVsToUpdate ← random 20% SVs;
10   foreach sv ∈ setOfSVstoUpdate do // done in
       parallel on CPU cores
11     create weights SVB_w;
12     create error SVB_e;
13     SVB_e' ← SVB_e;
14     foreach voxel in sv do
15       updateVoxel(voxel, SVB_e, SVB_w, A);
16     SVB_Δe ← SVB_e' - SVB_e;
17     lock();
18     e ← e - SVB_Δe;
19     unlock();
20   iter++;
```

tially share error sinogram data, as shown by the fainter cells in Fig. 1b. Therefore, simultaneous updates from different voxels to the same sinogram entry must be performed in a critical section. PSV-ICD parallelizes across different SuperVoxels, while sequentially updating voxels within an SV, as shown in Algorithm 2. Since each SV has a private SVB, updating the error sinogram is deferred until all voxels in the SV are done, and then the difference between the updated error SVB and the original error SVB is computed and synchronously added back to the original error sinogram.

2.3 GPU Architecture and Programming Model

We briefly describe the GPU architecture and programming model using the example of NVIDIA's Maxwell Titan X and CUDA, which were used in our work.

The GPU's cores are organized into 24 streaming multiprocessor units (SMMs), each having 128 CUDA cores. High-performance GPU programs are typically written using the CUDA programming model, where the programmer offloads parallel sub-programs, called *kernels*, onto the GPU. The CPU and GPU have separate memory spaces among which programmers perform data transfers explicitly. Threads in a kernel are divided into *threadblocks*, which are subdivided into *warps*. A warp represents the SIMD width of the GPU. Threads of a threadblock can synchronize using the *__syncthreads()* function call. There is no such primitive available for threads belonging to different threadblocks. Each thread in the kernel has access to registers, on-chip programmer-managed cache (shared memory), and the device (global) memory. Multiple threadblocks execute on each SMM. Resources of each SMM, such as shared memory, registers, etc. are partitioned among the threadblocks executing on it. Thus, larger requirements from the threadblocks can constrain the maximum concurrent threadblocks executing on an SMM. Occupancy is the metric of the achieved degree of multi-threading. It is the ratio of coexisting GPU threads to the maximum number of threads that can reside on the GPU.

GPUs have low cache sizes per thread. In the Maxwell architecture, L1 cache and texture cache are unified, and have a size of 24KB. The unified cache is shared among the threads of an SMM. The unified cache can only possess data that remains read-only across kernels. By default, only the local data and textures are stored in the unified cache. There is a common L2 cache for all SMMs, sized 3MB. It caters to all global data accesses. If threads in a warp access neighboring memory locations, these accesses may get coalesced into only a single memory access, improving memory bandwidth.

3. Mapping MBIR Parallelism to the GPU

This section describes the proposed mapping of MBIR parallelism to the GPU. Section 3.1 describes the levels of parallelism that we exploit. Section 3.2 presents our thread-mapping scheme, and the resulting GPU-ICD algorithm.

3.1 Understanding the MBIR Parallelism

Three levels of parallelism exist in the processing of each image through the MBIR algorithm.

Intra-voxel parallelism - Updating each voxel requires computation of theta1 and theta2 (steps 3-6 of Algorithm 1). This process requires a dot-product of size (number of views) $\times$ (average channels per voxel per view), which is large in practice (>2000). Reduction-style parallelism is exploited for this operation. Updating the error sinogram after the new voxel value is computed (steps 9-11 of Algorithm 1) can also be performed in parallel.

Intra-SV parallelism - Voxels inside a given SV can be updated in parallel, as long as the updates to the error sinogram entries do not overlap. The number of voxels in an SV can be high, *e.g.*, an SV of side 30 would have 900 voxels, allowing significant parallelism.

Inter-SV parallelism - Multiple SVs can be operated on in parallel as well. However, the degree of available parallelism along this dimension is modest, *e.g.*, for an image of 512×512, an SV side of 30 would lead to 289 SVs.

The PSV-ICD algorithm only exploits the top-most level of parallelism, *i.e.*, inter-SV parallelism. As the number of SVs in a typical reconstruction is higher than the CPU core count, this approach provides sufficient parallelism. Although SVBs can be larger than the CPU L1 cache, since

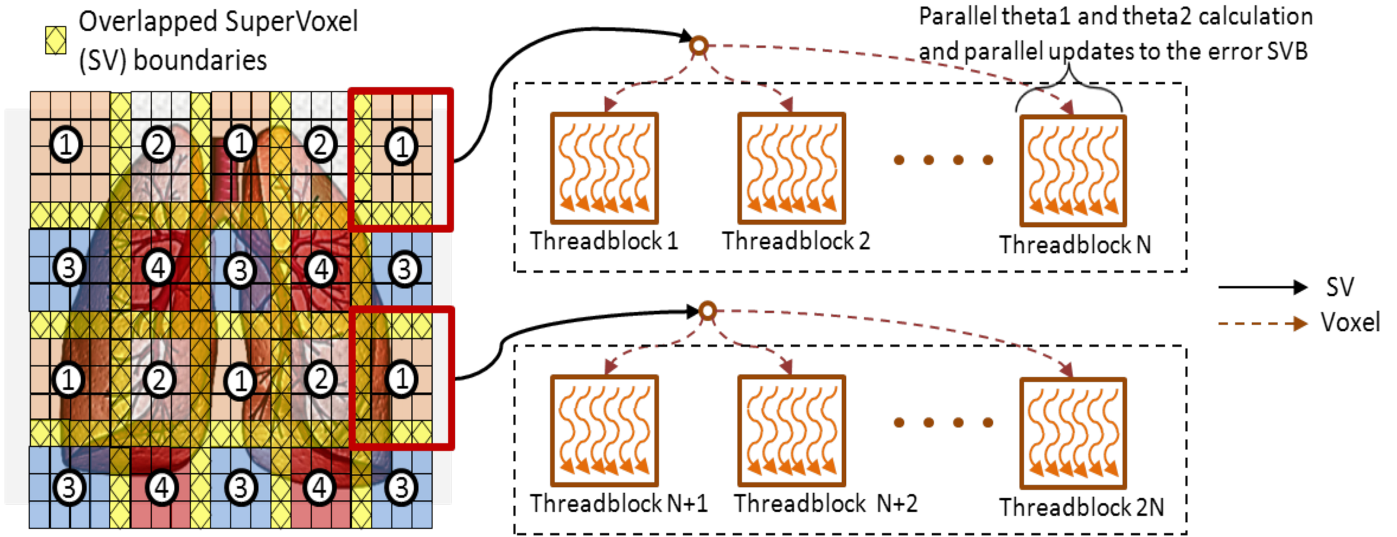


***Figure 3: GPU Parallelism Mapping -** SuperVoxels with the same color belong to the same checkerboard group (marked by circled numbers). They do not share any boundaries, and therefore can be concurrently updated. Assigning consecutive threadblocks to the same SV increases achieved L2 bandwidth.*

each CPU core has its own private L2 cache, SVBs for each SV can fit in it, and high locality can be achieved. On the other hand, intra-SV parallelism can lead to false sharing among the CPU cores. The fine-grained intra-voxel parallelism can be exploited by vector instructions, but the irregular memory access pattern hinders autovectorization.

In contrast to the CPU (where a few tens of threads suffice), orders-of-magnitude more GPU threads must be created to ensure that cores are utilized and memory latency is effectively hidden. The number of SVs ranges in the low hundreds. Hence, exploiting only inter-SV parallelism is insufficient. The L1 cache size per thread is fairly small on GPUs (making it difficult to contain an SVB), and no hardware prefetching is present. Unlike CPUs, the L2 cache latency is at least an order of magnitude higher than the L1 latency, lowering SVB caching benefits. One way to gain benefits from inter-SV parallelism alone could be for each SMM to operate on a single SV at a time, so that sufficient L1 cache is available. However, this severely limits the GPU occupancy, leading to poor performance.

Exploiting intra-SV parallelism alone faces similar issues. The voxel count per SV is of the order of a few hundreds, and fails to exploit sufficient GPU parallelism. Similarly, intra-voxel parallelism has a low degree of parallelism. One approach, therefore, would be to use both inter-SV and intra-SV parallelism. Different threadblocks then may operate on different SVs, while threads in a threadblock may update individual voxels in parallel. However, this approach eliminates any possibility of coalesced accesses to the sinograms. Our approach therefore employs all three levels of parallelism, as we describe next.

3.2 Parallelism Mapping

Fig. 3 shows our parallelism mapping scheme, and Alg. 3 describes the corresponding GPU-ICD algorithm. Unlike PSV-ICD, our mapping employs all three levels of parallelism (intra-voxel, intra-SV, and inter-SV) identified above. For each SV, multiple CUDA threadblocks are launched, partitioning the voxels in the SV among themselves. Zero-skipping (Sec. 2) can lead to load imbalance among the different threadblocks, therefore our algorithm performs dynamic scheduling of voxels among threadblocks. Adjacent SVs share boundary voxels, as in PSV-ICD, to obtain faster convergence.

Since SVB size is much larger than the L1 cache size on GPUs, achieving good cache locality via L1 cache or shared memory is impossible. Fortunately, SVB sizes are small enough to fit in the GPU L2 cache. Although GPU L2 latencies are high, our parallelism mapping scheme increases the obtained L2 bandwidth by launching multiple consecutive threadblocks per SV to make the most of the L2 temporal locality. The amount of parallelism in this dimension is limited by the bound on the exploitable intra-SV parallelism.

Each threadblock updates one voxel at a time, exploiting intra-voxel parallelism to compute the theta1 and theta2 values. The partial values computed by each thread are stored into the shared memory, and then reduced using a fast tree-style reduction [14]. Only a single thread of the threadblock updates the voxel value. Writing back to the error sinogram is again performed in parallel by the threads of a threadblock. However, as different threadblocks may be writing to the same sinogram locations, this write-back requires atomic updates.

```
Algorithm 3: GPU-ICD Algorithm
   Input: A - system geometry matrix, e - error sinogram, w -
          weight sinogram
   Output: Converged reconstructed image
1  Function MBIR_GPU_Kernel(SVBs, A)
      // Each kernel thread executes this
2     tid ← thread identifier;
3     svID ← getSVId(tid);
4     while (voxel ← atomicFetch(svId)) do // all
       threads in a threadblock get the same
       voxel
5        Calculate partial theta1 and theta2 and store them in
           shared memory;
6        __syncthreads();
7        Perform tree-style reduction for theta1 and theta2;
8        __syncthreads();
9        if tid==0 then
10          Update voxel value;
11       __syncthreads();
12       Atomically write back to the error SVB;
13       __syncthreads();

14 Create SVs;
15 iter ← 1;
16 while image not converged do
17    if iter == 1 then
18       setOfSVsToUpdate ← all SVs;
19    else if iter is even then
20       setOfSVsToUpdate ← top 25% SVs by the update
           amount;
21    else
22       setOfSVsToUpdate ← random 25% SVs;
23    cb ← makeCheckerBoardGroups(setOfSVsToUpdate);
24    for (k=0; k<4; k++) do
         // Launch GPU kernels, each having
            BATCH_SIZE SVs
25       for (i=0; i< cb[k].SVCount; i+= BATCH_SIZE) do
26          if (cb[k].SVCount - i) < threshold then
27             break;
            // done in parallel via GPU kernel
28          Create SVBs for the batch in the GPU memory;
29          MBIR_GPU_Kernel(SVBs, A);
            // done in parallel via GPU kernel
30          Atomically update error sinogram for all SVs in the
              batch;
31    iter++;
```

A key distinction of the GPU-ICD algorithm is that unlike PSV-ICD, where the error sinogram is updated with partial deltas ($\text{SVB}_{\Delta e}$ in line 18 Algo. 2), immediately after each SV is updated all error sinogram updates are performed once voxel updates across all SVs in a GPU kernel have finished. Similarly, all SVBs are created prior to the execution of the MBIR_GPU_Kernel. The GPU-ICD algorithm does this to avoid cache pollution engendered by $\text{SVB}_{\Delta e}$ and SVB'_e accesses during the execution of MBIR_GPU_Kernel. Instead, SVB generation and error sinogram write-backs are performed via separate individual GPU kernels.

Since SVs share boundaries, simultaneous updates of neighboring SVs may lead to erroneous voxel values on the boundaries. This occurs since the error sinogram and the voxel value lose the necessary correspondence. The PSV-ICD algorithm updates fewer SVs in parallel at a time (~16) and does not exploit intra-SV parallelism, making the chances of a boundary voxel being updated simultaneously in parallel negligible. However, the GPU-ICD algorithm generally updates a higher number of SVs concurrently and employs intra-SV parallelism, and therefore is more likely to perform simultaneous boundary voxel updates. To avoid this, the GPU-ICD algorithm partitions SVs into four sets in a *checkerboard* pattern, as shown in Fig. 3. SVs in a given group, marked by the same color in Fig. 3, cannot be neighbors and can be updated in parallel.

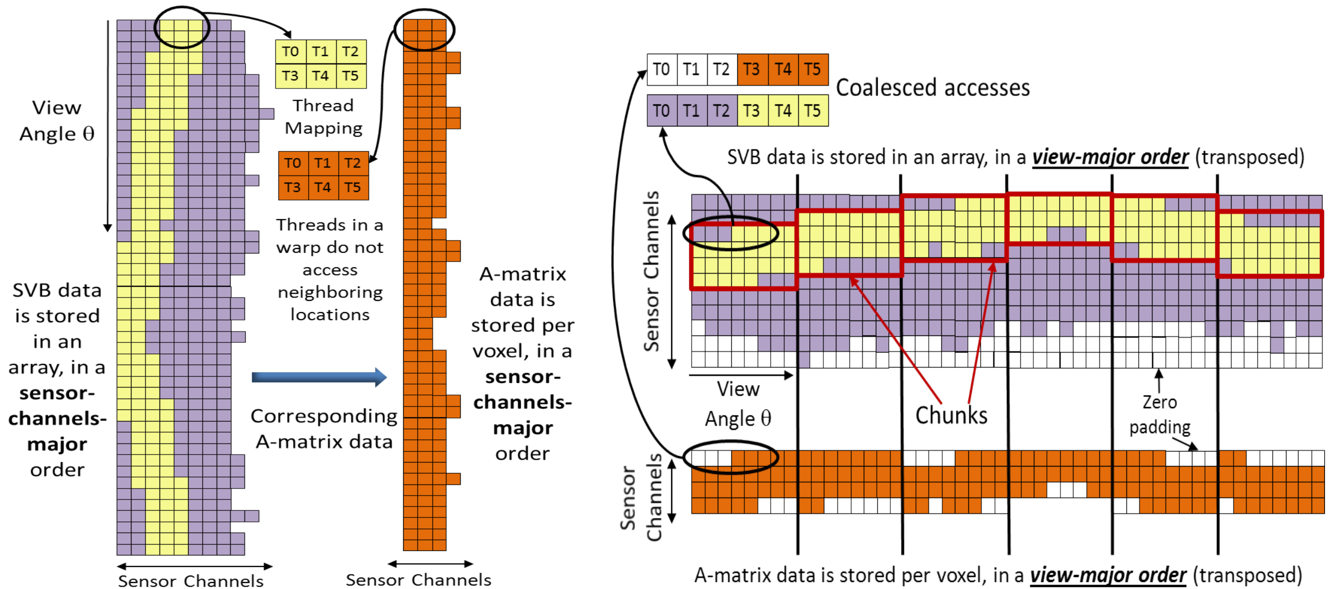
Load imbalance is the last issue overcome by the GPU-ICD algorithm. The algorithm launches up to *BATCH_SIZE* SVs at a time on the GPU. However, randomization in the SV ordering, paired with the distribution into four checkerboard sets, may result in fewer SVs than *BATCH_SIZE* getting launched in a GPU kernel. This results in GPU underutilization. To overcome this issue, the GPU-ICD algorithm increases the fraction of SVs to be updated to 25% in each iteration, compared to 20% in the PSV-ICD algorithm. This increases the SV availability in each of the four groups. Furthermore, GPU-ICD only launches a kernel if the number of SVs to be launched is above a *threshold*, which is set to be *BATCH_SIZE/4*.

4. Performance Optimizations for GPU-ICD

In this section, we describe additional optimizations that were explored to improve the performance of GPU-ICD. Section 4.1 presents a data layout transformation to achieve coalesced accesses. Section 4.2 presents a mechanism to exploit the GPU shared memory, while Section 4.3 describes our approach to increase the achieved memory bandwidth.

4.1 Data Layout Transformations to Obtain Coalesced Accesses

The accesses made per voxel to an SVB are not completely linear, as seen in Fig. 4a. For each location in the sinogram associated with a voxel, an associated A-matrix element needs to be accessed. All these A-matrix elements, across all views, are placed in memory in a contiguous fashion, using a sparse matrix format. The data in both the A-matrix and the SVBs are stored in a sensor-channels-major order. In our first naive implementation, threads in a threadblock operate on these data in a sensor-channels-major order as shown by the highlighted cells in Fig. 4a, and therefore fail to obtain coalesced accesses. Moreover, the starting location for each view for each voxel must be pre-calculated and then read before the SVB could be accessed.



(a) Original data layout - *Lighter cells contain data for a given voxel in the SVB. Accesses to both the SVB and A-matrix are in the sensor-channels order, and hence are uncoalesced.*

(b) Transformed layout - *The SVB and A-matrix are both transposed. Data accesses are now in the view order. The A-matrix is zero-padded. Each chunk in an SVB and its corresponding A-matrix chunk are now perfect rectangles.*

Figure 4: Data layout transformation to obtain coalesced accesses

To obtain coalesced accesses, we first transpose the SVB, *i.e.*, data is stored in a view-major order instead of in the sensor-channels-major order. This naturally increases the run-length (length of contiguous accesses per row). However, the number of elements in each row to be operated on is highly variable because of the sinusoidal access pattern, and distributing them fairly among threads becomes difficult. Our scheme overcomes this issue. First, we make the SVB perfectly rectangular by zero-padding, as shown in Fig. 4b, and place each row at an aligned address. Next, the SVB data required per voxel can be conceptually thought of being split into chunks of a given width. Each chunk is a rectangular block, containing few non-voxel-related elements. The next layout transformation zero-pads the A-matrix data so that each SVB chunk can be mapped to an equivalent A-matrix chunk. The computation now simply involves reading each row from the chunks of the SVB and A-matrix, and performing an element-by-element multiplication, achieving coalesced accesses. The zero-padding of the A-matrix ensures that non-voxel-related elements in the SVB do not affect the computation correctness. Although zero-padding increases the amount of computation, and requires additional data to be read, the benefits obtained by coalesced accesses outweigh these costs. In the voxel update mechanism, the chunks get distributed among the warps of the threadblock that is updating the voxel. To do so, only the starting locations and the number of rows for each chunk need to be pre-computed and accessed, overcoming the penalty of per-view starting location look-ups in the naive implementation.

4.2 Achieving High Occupancy via Register Spilling to Shared Memory

GPUs hide their longer (than CPU) memory latency by bringing in a new warp whenever a running warp is waiting on a memory access. Because MBIR performance is largely limited by memory accesses, a large number of warps must be available on each SMM.

Our initial GPU-ICD implementation used 44 registers per thread. Since registers in an SMM are partitioned among threadblocks of the SMM, occupancy is limited by the high register requirement. For a threadblock of 256 threads, occupancy was being restricted to 50%. To achieve higher occupancy, we used the `maxRegCount` flag with `nvcc` and restricted the register count to 32 per thread, achieving 100% occupancy. We observed only a 6% performance improvement by doing this. This was because restricting the register count led to increased accesses to the GPU L1 cache, and the L1 cache hit rate remained poor (30%). The spilled registers were frequently being fetched from the L2 cache, whose high latency was limiting performance.

To address this issue, we manually placed some of the variables from each thread into shared memory. This allowed the MBIR_GPU_Kernel to be compiled with 32 registers per thread. We again achieved 100% occupancy, and obtained a significant performance boost, as will be shown in Section 5.

4.3 Increasing Achieved Memory Bandwidth

Since MBIR is a memory bound algorithm, achieving high memory bandwidth is essential to obtaining high perfor-

mance. The primary way to achieve high bandwidth is to increase locality, resulting in higher cache hit rates. On the GPU, this is typically achieved by storing reusable data in the shared memory. Shared memory is partitioned across threadblocks. Hence, the larger the shared memory requirement, the smaller the number of concurrent threadblocks. In GPU-ICD, SVBs are the key data structures to be considered for placement in the shared memory owing to their high reuse. Unfortunately, the SVBs are large, and placing them in the shared memory would severely lower the obtained occupancy. *e.g.*, an SV of side 6 voxels results in an SVB of size 135KB for 720 views, which goes beyond the shared memory size. For an SV of side 4, the SVB would require 90KB. As the shared memory size on an SMM is 96KB, only a single threadblock can execute on the SMM. The maximum number of threads a threadblock could meaningfully use is limited by the number of views, i.e., 720, in our case. Hence, the achieved occupancy would only be 35%. Further, since such an SV would have only 16 voxels, intra-SV parallelism, as well as the reuse frequency would get diminished significantly. Therefore, we had to place SVBs in the global memory. To improve the achieved memory bandwidth, we next present two techniques.

4.3.1 Reading the A-Matrix through Texture Cache

The A-matrix is a read-only data structure. Therefore, it makes sense to read it via the combined Texture memory/L1 cache datapath on the Maxwell architecture. This is achieved by explicitly marking the data as read-only, and specifying the `__restrict` keyword that allows the compiler to read the data via the texture cache.

Since A-matrix accesses have little temporal locality, they consume high memory bandwidth, forming a bottleneck. The A-matrix data is stored as `floats`, requiring four bytes per entry. To reduce the memory bandwidth consumed, our scheme converts this data into `unsigned chars`, needing only a byte per entry. Each A-matrix entry is normalized to a byte-long equivalent A-matrix entry as follows:

newAEntry= (unsigned char) (AEntry/maximum of all A-matrix entries for the voxel)*255 + 0.5

The division by the maximum performs normalization, while the multiplication by 255 brings out the MSBs in the 8-bit `unsigned char`. The normalization allows for more bits in the `float` entry to get transferred to the `char`. Adding 0.5 achieves rounding instead of truncation. The maximum A-matrix entry per voxel needs to be stored and referred to when calculating the original `float` value back from the `char`, prior to the actual computation. The overhead of storing and reading this extra data gets compensated for by the reduction in the total data read, as will be shown in Section 5.

4.3.2 Obtaining Higher L2 Bandwidth

The SVBs, which are placed in the global memory, get accessed through the L2 cache. The L2 cache is large enough to hold multiple SVBs. Although the SVB accesses achieve a high hit rate on the L2 cache, it is essential to obtain high L2 bandwidth in order to gain high performance. We found that accessing data through L2 as type `float` only obtained 50% of the L2 bandwidth on the Titan X GPU. However, we found that accessing the data as type `double` can achieve 100% of the L2 bandwidth. In the GPU-ICD algorithm, we read the SVB data in this manner. The error sinogram updates, however, are read-and-write atomic operations conducted via CUDA atomic addition functions, and hence cannot be performed as `double`.

5. Experiments

5.1 Experimental Setup

Inputs: The benchmark data set used in this evaluation comprises 3200 test cases obtained from an Imatron C-300 scanner from ALERT Task Order 3 (TO3) [4]. The slices in this dataset are generated using parallel beam projection, and have reconstruction image size of 512x512. The data is generated using 720 uniformly distributed views between 0 and 180 degrees. The number of sensor array channels used was 1024.

System setup: Our GPU experiments are performed with Nvidia CUDA Titan X GPU which comprises 24 SMMs, each having 128 cores, clocked at 1127 MHz. The GPU has a device memory of 12GB. The GPU node contained an Intel i7-6700K CPU with 16GB RAM. The GPU program was compiled with `nvcc` with *-O3* option. The CPU experiments were conducted on a node with two sockets, each containing Intel Xeon E5-2670 CPU, running at 2.6GHz. The total number of cores was 16, with 64GB RAM. The CPU code was compiled with `icc` with *-O3* option. The TDP of the CPU (230W) is comparable to that of the GPU TDP (250W).

5.2 Overall Performance

Table 1 presents the execution time comparison between the CPU and GPU versions of MBIR. Owing to zero-skipping, neither PSV-ICD, nor GPU-ICD update all voxels in each iteration. Hence, we measure convergence using *equivalent iterations* or *equits*. An update of N voxels, where N is the total number of voxels in the image, is one equit. We first obtain the golden output image by executing the traditional ICD algorithm for 40 equits, by when it is known to converge. For PSV-ICD and GPU-ICD, we evaluate algorithmic convergence by measuring the root-mean-square error (RMSE) with respect to the golden image in Hounsfield Units[1]. We report execution time when the RMSE goes below 10HU, since previous work [15] has found that no visi-

[1] The Hounsfield Unit (HU): a CT measurement unit of the object's radio density compared to the radio density of distilled water.

Table 1: Comparison of PSV-ICD and GPU-ICD MBIR Performance.

	Mean Execution Time (s)	Mean Speedup over Sequential ICD	Std. Dev. of Exec. Time	SV Side Used	Average #Equits to Converge	Time per Equit (s)	Other parameter values
PSV-ICD (CPU)	138.26X	1.801	0.535	13	4.8	0.41	
GPU-ICD	0.407	611.79X (4.43X over PSV-ICD)	0.083	33	5.9	0.07	Chunk Width: 32, #Threadblocks/SV: 40, #SVs/batch: 32

ble artifacts remain at this level. Sequential ICD refers to the publicly available, single-core MBIR implementation [16]. The geometric mean speedup achieved by GPU-ICD over the sequential ICD is 611.79X, while over PSV-ICD is 4.43X. The PSV-ICD time per equit is 5.86X higher than for GPU-ICD. However, GPU-ICD requires more equits to converge. Table 1 also shows that the standard deviation in PSV-ICD runs is much higher than in the GPU-ICD ones. We suspect that GPU-ICD is being limited by the span, lowering the deviation.

Fig. 5 compares the convergence speeds of GPU-ICD and PSV-ICD on a representative image. GPU-ICD achieves convergence much rapidly compared to PSV-ICD.

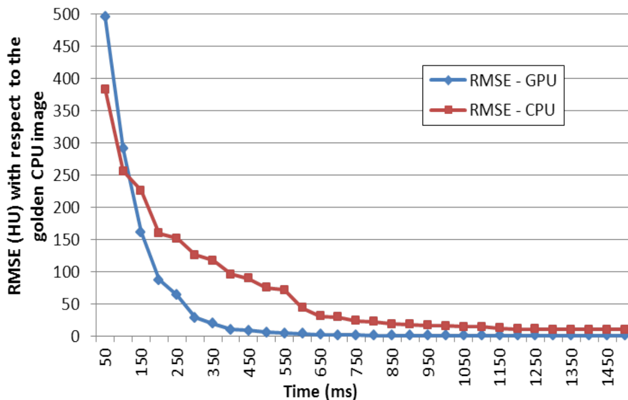


Figure 5: Convergence of PSV-ICD (CPU) and GPU-ICD Algorithms

Table 1 presented several parameters used in the GPU-ICD execution. We tuned these parameters on a representative image, and used those values for rest of the images. We observed however that the best performing parameter values differ across images, with a significant execution time variation. Therefore, the GPU performance in Table 1 can be improved with image-specific parameter selection. Next, we present performance analysis of these parameters on our representative image. To do so, we change one parameter value at a time, while retaining the other parameter values as shown in Table 1.

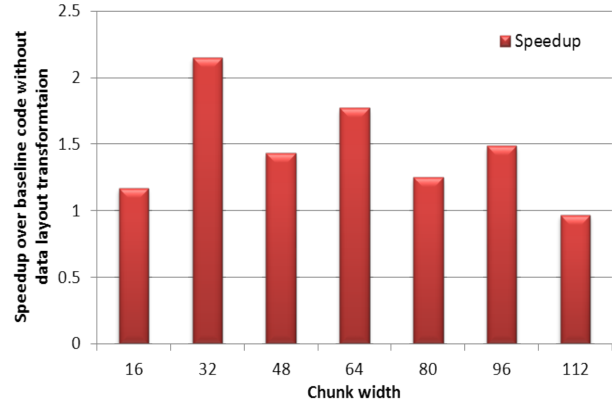


Figure 6: Speedup obtained by data layout transformed code vs. a code with the default data layout

5.3 Performance Impact of the Optimizations

Fig. 6 shows the performance benefits obtained by the proposed data layout transformation (Sec. 4.1). The speedup is calculated over the naive GPU code where the default data layout was employed. For smaller widths, data chunks for a voxel are small in size, lowering the total achieved coalesced access count. For larger chunk widths, the penalty of additional computation and memory accesses becomes prohibitive. The chunk width of 32 performs the best, obtaining a speedup of 2.1X. Widths that are multiples of warp size (i.e 32) perform better because they achieve aligned memory accesses.

Table 2: Impact of shrinking the A-matrix and reading it via Texture cache

Reading A-matrix from (memory, type)	Execution Time (s)	Achieved Bandwidth of Unified L1/Texture Cache (Hit Rate%)
(Global, `float`)	0.48	
(Texture, `float`)	0.45	519GB/s (41.78)
(Global, `char`)	0.44	
(Texture, `char`)	0.41	702GB/s (60.36)

Table 2 summarizes the benefits of compressing and reading the A-matrix via unified L1/texture cache. Shrinking the A-matrix lowers the memory footprint and achieves higher

hit rate on the unified L1/texture cache, achieving a 1.17X speedup.

Table 3: Impact of several GPU-specific optimizations

Optimization Turned Off	Slowdown
Reading Sinogram as `double`	1.053X
Placing Variables on the Shared Memory	1.124X
Exploiting Intra-SV Parallelism	6.251X
Dynamic voxel distribution	1.064X
Setting threshold for batch sizes	1.099X

Table 3 measures impact of five optimizations. The impact is measured in terms of the slowdown caused when optimizations were turned off. The first two are i) reading sinograms as `double`, and ii) spilling registers on the shared memory in order to increase occupancy. These optimizations were described in Section 4. The prior increased the achieved L2 bandwidth from 395 GB/s to 472 GB/s. The latter increased the achieved shared memory bandwidth from 398 GB/s to 456 GB/s. The achieved unified L1/Texture cache bandwidth was 702 GB/s, while for the device memory, it was 152 GB/s. Summed up, the total bandwidth achieved is 1802 GB/s, which is 5.36X that of the maximum device memory bandwidth (336 GB/s) for the Titan X GPU. Achieving such high memory bandwidth reduces the memory access bottleneck known to exist in MBIR. The third optimization in Table 3, i.e., intra-SV parallelism, has the highest performance impact measuring about 6X. Exploiting intra-SV parallelism is therefore crucial on GPUs. The last two optimizations (from Section 3) deal with load imbalance issues. The dynamic distribution of voxels of an SV among threadblocks overcomes the load imbalance that arises in static distribution owing to the zero-skipping. Setting a threshold on the number of SVs to be launched in a batch removes the GPU underutilization penalty in cases where only a few SVs are launched.

5.4 Performance Impact of Tuning Parameters

Unlike PSV-ICD, where only the SV side length impacts the achieved performance, several parameters affect the GPU-ICD performance. Fig. 7a measures execution time for different SuperVoxel side lengths. Smaller SV sides result in higher contention during the atomic updates in GPU-ICD. The intra-SV parallelism exploitation worsens this behavior compared to PSV-ICD. For larger SV sides, the SVBs become too large, reducing the L2 caching benefits. An SV side of 33 performs the best, since it achieves the highest L2 throughput. The secondary vertical axis in Fig. 7a displays the number of equits required for each SV side. This number increases with the SV side because updates to the error sinogram occur at coarser granularity, slowing down the algorithmic convergence. We also suspect that the intra-SV parallelism slows the convergence.

Fig. 7b shows the impact of exploited intra-SV granularity. The performance improves with the number of threadblocks used per SV, i.e., intra-SV parallelism. A moderately high number of threadblocks per SV achieves higher L2 temporal cache locality. The performance saturates after 32 threadblocks.

Fig. 7c evaluates performance impact of the exploited degree of intra-voxel parallelism. This degree is determined by the number of threads in a threadblock. Occupancy, which gets affected by the number of threads in a threadblock, plays an important role in the achieved performance, e.g. 384 threads per threadblock result in lower occupancy, leading to lower performance. However, occupancy is not the only factor to consider, e.g., with 64 threads per block, although the occupancy is 100%, the small threadcount per block results in larger active threadblock count. This results in more SVBs being accessed simultaneously, leading to L2 conflicts. A larger threadcount per block (e.g. 512) leads to asymmetric work distribution of the 720 views involved, and increases the reduction cost, resulting in lower performance.

Fig. 7d measures the impact of the (maximum) number of SVs updated in a batch (individual GPU kernel). The lower this number, the higher the total number of kernel launches, resulting in higher overheads occurring from kernel launches. If the number gets too high, then updates to error sinogram start taking place at coarser granularity, leading to slower algorithmic convergence.

6. Generalization

This section discusses how GPU-ICD generalizes to a broader range of applications. The algorithmic techniques, as well as the GPU-specific optimizations proposed in this paper, are applicable to these classes of applications.

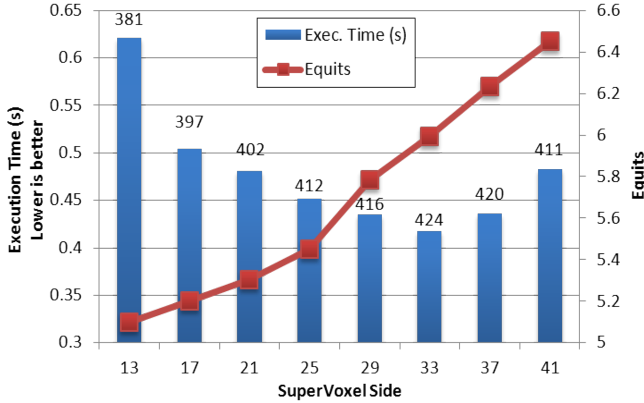
For many important sensing problems (such as synchrotron imaging [17], machine learning [18], geophysics sensing [19], radar sensing [20]), it is necessary to minimize a convex cost function given by the following general set of quadratic equations:

$$f(x) = \|y - Ax\|_\Lambda^2 = (y - Ax)^t \Lambda (y - Ax) \qquad (1)$$

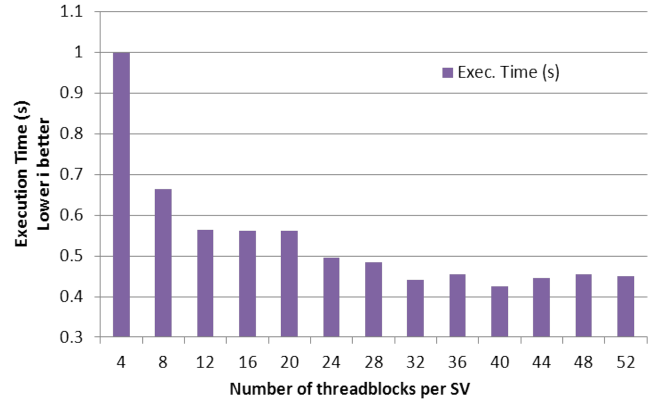
where y is a $M \times 1$ vector, A is a very large $M \times N$ matrix, Λ is a diagonal matrix of weights, and x is a $N \times 1$ vector. To find x^* such that $f(x^*)$ is minimized, we can solve Equation (1) by numerical iterative algorithms.

ICD algorithms have received increasing attention as an effective approach to compute the solution to the general optimization problem in Equation (1) because of their fast and robust convergence [2, 1, 10, 11]. In conventional ICD algorithms, every element, $x_j \in x$, is cycled through and updated exactly once in each iteration. The update of each element, x_j, is designed to decrease the value of $f(x)$, so ICD algorithms result in a monotonically decreasing sequence of cost. Under appropriate technical conditions, the ICD algorithms asymptotically reach the value $\lim_{n\to\infty} f(x^n) = f(x^*)$.

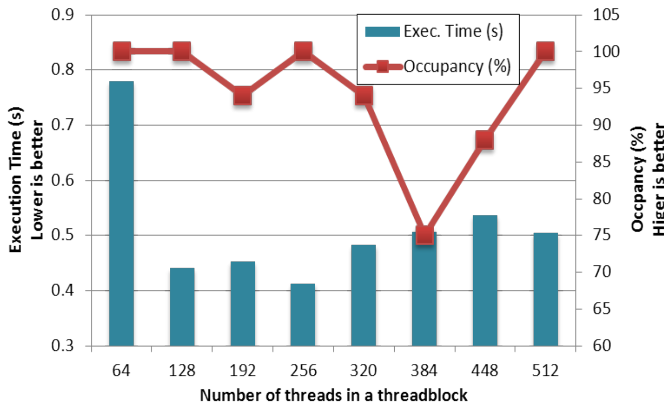
Importantly, each update generally requires the evaluation of $A_{*,j} x_j$, and this requires an access to a single col-



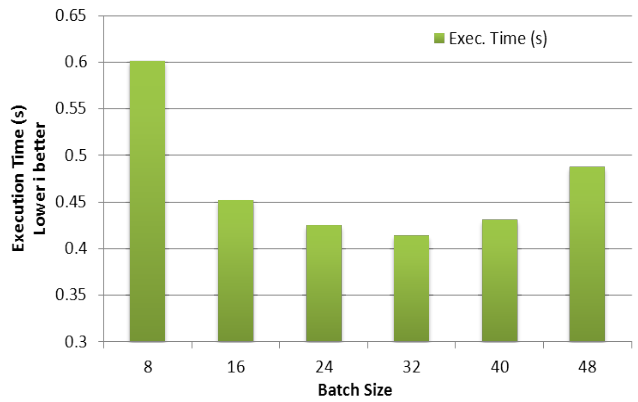
(a) *SuperVoxel side length impacts both performance and convergence. The numbers above bars indicate L2 throughput in GB/s*



(b) *Number of threadblocks used per SV determines the exploited intra-SV parallelism granularity*



(c) *Number of threads in a threadblock determine the exploited intra-voxel parallelism granularity*



(d) *Varying number of SVs processed per kernel launch (batch)*

Figure 7: Performance impact of various tuning parameters: *values of these parameters can significantly affect the achieved performance*

umn of the matrix A. While ICD algorithms have been of great theoretical interest, they have been generally assumed to be unsuited for GPUs because of what appears to be their intrinsically serial nature as well as the restriction on how columns of the A matrix are accessed. The GPU-ICD introduced in this paper demonstrates that ICD algorithms can be efficiently implemented on GPUs. Not only does the GPU-ICD show that a large number of parallel operations can run simultaneously for ICD algorithms, but the GPU-ICD algorithm also shows that ICD algorithms can efficiently reuse the GPU memory.

To map such numerical algorithms onto GPUs, not only should vector operations, such as $A_{*,j}x_j$, be parallelized, but different elements of x should be updated simultaneously. GPU-ICD can be perceived as a generalized parallel update framework for such numerical iterative algorithms.[2] Each element of x, x_j, can be perceived as a voxel, as discussed in Section 1. Therefore, the intra-voxel parallelism in GPU-ICD can be generalized for the vector multiplication between $A_{*,j}$ and x_j. A group of x_j, denoted as S, can then perceived as a SV. Voxels of the same SV are chosen to maximize their statistical correlation, so that voxels' updates can exploit cache locality. In other words, if $x_i, x_j \in S$, then $\sum_{k=1}^{M} |A_{k,i}| \cdot |A_{k,j}|$ is maximized. The intra-SV parallelism discussed in Section 3.1 describes how to map the updates for a group of x_j onto a GPU. At the same time, voxels of different SVs are chosen to minimize their statistical correlation, lowering the synchronization overhead. Namely if x_i, x_j belong to different groups, then $\sum_{k=1}^{M} |A_{k,i}| \cdot |A_{k,j}|$ is minimized. The inter-SV parallelism discussed in the paper can be generalized for mapping the updates for different groups of x_j onto a GPU.

[2] Note that if $f(x)$ is a linear system of equations, GPU-ICD is analogous to the parallel Gauss-Seidel algorithm.

7. Related Work

CT image reconstruction techniques can be classified into two major categories: direct methods, such as filtered back

projection (FBP), and iterative methods. The iterative method considered in this paper i.e., Model Based Iterative Reconstruction (MBIR) [2, 3], is known to create higher quality images than FBP methods. Past research has studied this for various domains, including medical imaging [2, 3], explosive detection systems (EDS) [4, 5, 6, 7], scientific and materials imaging [8].

There are two prominent approaches to iterative reconstruction. The first approach comprises non-regularized methods, such as Simultaneous Iterative Reconstruction Technique (SIRT) [5, 21, 22] and Algebraic Reconstruction Techniques (ART) [23]. SIRT work by iteratively projecting an entire volume to be reconstructed into the measurement, while ART work by iteratively solving a system of linear equations. However, non-regularized methods lack concrete convergence criteria and rely on stopping times. The other approach is of regularized methods, who generally possess well-defined convergence criteria and generate higher quality images than FBP. Among the regularized methods, Penalized Least Squares (PLS) [24], Penalized Maximum Likelihood (PML) [25], and Model Based Image Reconstrustion (MBIR) are popular. The regularized methods need to minimize a cost function, which is computationally expensive. Two optimization approaches exist to that end. The first is simultaneous methods, which work by using forward/backward projectors. Simultaneous methods are usually slow, and hence need pre-conditioning [26] to converge faster, which is scanner geometry dependent. Ordered subsets [26] is another approach to speed up simultaneous methods. The second approach is of coordinate decent (CD), where methods such as Iterative Coordinate Descent (ICD) [2, 9, 10] and Gradient Coordinate Descent (GCD) [11, 12] are used. The CD methods are preferable since they converge faster than the simultaneous methods, and are geometry agnostic. We consider ICD over GCD in this paper, since it has been highly optimized for multi-core CPUs [1].

Many previous GPU-based efforts of CT reconstruction [27, 28, 29, 30, 31] are based upon the FBP method, since it is easier to parallelize [32]. Cho et. al. [33] present a simultaneous method based on ordered subsets. Flores et. al. [34] show a non-regularized approach where reconstruction is formulated as a least square solver problem and is solved using standard GPU libraries. McGaffin et. al. [35] present an alternating tomography and denoising based regularized MBIR approach on the GPU. In [35], McGaffin and Fessler investigate the use of GPUs for multi-slice helical scan CT reconstruction in medical applications. Their approach uses projection based optimization, rather than ICD, for the tomographic portion of the optimization, and couples it with a separate TV denoising portion for the prior. Importantly, their approach is also based on ordered subset (OS) methods, which are not generally compatible with the sparse view tomography methods that are crucial in many scientific and NDE applications [36, 37, 7].

None of the above approaches have exploited ICD on GPUs. While ICD and its variants have been shown to be robust and rapidly converging algorithms [38] for performing the MBIR reconstruction, there is a belief that ICD can not be efficiently mapped to highly parallel GPU architectures [11, 12]. This paper shows that fast converging ICD optimization algorithms can be efficiently implemented on GPUs. Furthermore, since ICD is an optimization technique, methods employed in this work for MBIR are also applicable to PLS/PML techniques.

8. Conclusion and Future Work

Although ICD-based MBIR (Model-based Image Reconstruction) is known to produce high quality CT reconstruction images, it suffers from a heavy computational cost. This paper presented GPU-ICD, the first GPU-based highly parallel algorithm for this kind of MBIR. The algorithm exploits multiple levels of parallelism in MBIR, namely, intra-voxel parallelism, intra-SV parallelism, and inter-SV parallelism. The paper presented a data layout transformation that obtains coalesced accesses on GPUs. The paper also presented the following GPU-specific optimizations: i) increasing occupancy by spilling registers on the shared memory; ii) compressing system matrix and reading it via texture cache; and iii) obtaining higher L2 bandwidth by reading the data as type `double`. The massive parallelism exploitation combined with the above optimizations results in a mean speedup of 4.43X over the best-known parallel CPU implementation on an iso-power 16-core CPU.

Section 5 showed that various parameters can greatly impact the performance of GPU-ICD. The best values of the parameters are sensitive to the input, and hence are often not catered to by auto-tuning systems [39, 40]. In future, we plan to build a model that automatically selects input-specific high performing parameter values.

Acknowledgments

This research was supported by the U.S. Department of Homeland Security under SBIR Award D15PC00110, subcontracted through High Performance Imaging LLC, and by the Indiana Economic Development Corporation (IEDC). Additional support was provided by the DHS ALERT Center for Excellence supported by the U.S. Department of Homeland Security, Science and Technology Directorate, Office of University Programs, under Grant Award 2013-ST-061-ED0001. The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the U.S. Department of Homeland Security or the IEDC.

References

[1] X. Wang, A. Sabne, S. Kisner, A. Raghunathan, C. Bouman, and S. Midkiff, "High performance model based image reconstruction," in *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '16, (New York, NY, USA), pp. 2:1–2:12, ACM, 2016.

[2] K. Sauer and C. Bouman, "A local update strategy for iterative reconstruction from projections," *IEEE Transactions on Signal Processing*, vol. 41(2), 1993.

[3] J. B. Thibault, K. D. Sauer, C. A. Bouman, and J. Hsieh, "A three-dimensional statistical approach to improved image quality for multi-slice helical CT," *Medical Physics*, vol. 34(11), 2007.

[4] DHS/ALERT, "Research and development of reconstruction advances in CT-based object detection systems." `https://myfiles.neu.edu/groups/ALERT/strategic_studies/TO3_FinalReport.pdf`, 2009.

[5] S. Degirmenci, D. G. Politte, C. Bosch, N. Tricha, and J. A. O'Sullivan, "Acceleration of iterative image reconstruction for X-ray imaging for security applications," in *Proceedings of SPIE-IS&T Electronic Imaging*, vol. 9401, 2015.

[6] P. Jin, E. Haneda, C. A. Bouman, and K. D. Sauer, "A model-based 3D multi-slice helical CT reconstruction algorithm for transportation security application," in *Second International Conference on Image Formation in X-Ray Computed Tomography*, 2012.

[7] S. J. Kisner, E. Haneda, C. A. Bouman, S. Skatter, M. Kourinny, and S. Bedford, "Model-based CT reconstruction from sparse views," in *Second International Conference on Image Formation in X-Ray Computed Tomography*, pp. 444–447, June 2012.

[8] P. Jin, C. A. Bouman, and K. D. Sauer, "A method for simultaneous image reconstruction and beam hardening correction," in *2013 IEEE Nuclear Science Symposium and Medical Imaging Conference (NSS/MIC)*, pp. 1–5, 2013.

[9] B. D. Man, S. Basu, J.-B. Thibault, J. Hsieh, J. A. Fessler, C. A. Bouman, and K. Sauer, "A study of different minimization approaches for iterative reconstruction in X-ray CT," in *IEEE Nuclear Science Symposium*, vol. 5, pp. 2708–2710, 2005.

[10] 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 Transactions on Image Processing*, vol. 20(1), 2011.

[11] J. A. Fessler, E. Ficaro, N. Clinthorne, and K. Lange, "Grouped-coordinate ascent algorithms for penalized-likelihood transmission image reconstruction," *IEEE Transactions on Medical Imaging*, vol. 16(2), 1997.

[12] J. Zheng, S. S. Saquib, K. Sauer, and C. A. Bouman, "Parallelizable bayesian tomography algorithms with rapid, guaranteed convergence," *IEEE Transactions on Image Processing*, vol. 9(10), 2000.

[13] J. E. Bowsher, M. Smith, J.Peter, and R. J. Jaszczak, "A comparison of OSEM and ICD for iterative reconstruction of SPECT brain images," *Journal of Nuclear Medicine*, vol. 79(5), 1998.

[14] J. Luitjens, "Faster parallel reductions on Kepler." `https://devblogs.nvidia.com/parallelforall/faster-parallel-reductions-kepler/`, 2014.

[15] Z. Yu, J. Thibault, C. A. Bouman, K. D. Sauer, and J. Hsieh, "Non-homogeneous ICD optimization for targeted reconstruction of volumetric CT," in *Computational Imaging VI, part of the IS&T-SPIE Electronic Imaging Symposium, San Jose, CA, USA, January 28-29, 2008, Proceedings*, p. 681404, 2008.

[16] P. Jin, S. J. Kisner, T. Frese, and C. A. Bouman, "Model-based iterative reconstruction (MBIR) software for X-ray CT." Available from https://engineering.purdue.edu/ bouman/software/tomography/mbirct/, November 2013.

[17] X. Wang, K. A. Mohan, S. J. Kisner, C. A. Bouman, and S. P. Midkiff, "Fast voxel line update for time-space image reconstruction," in *2016 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pp. 1209–1213, March 2016.

[18] C.-J. Hsieh, K.-W. Chang, C.-J. Lin, S. S. Keerthi, and S. Sundararajan, "A dual coordinate descent method for large-scale linear SVM," in *Proceedings of the 25th International Conference on Machine Learning*, ICML '08, (New York, NY, USA), pp. 408–415, ACM, 2008.

[19] J. F. Claerbout and F. Muir, "Robust modeling with erratic data," *Geophysics*, vol. 38, no. 5, pp. 826–844, 1973.

[20] L. Wu, P. Babu, and D. P. Palomar, "Cognitive radar-based sequence design via SINR maximization," *IEEE Transactions on Signal Processing*, vol. 65(3), 2017.

[21] N. Clinthorne, T. S. Pan, P. C. Chiao, W. L. Rogers, and J. A. Stamos, "Preconditioning methods for improved convergence rates in iterative reconstructions," *IEEE Transactions on Medical Imaging*, vol. 12(1), 1993.

[22] J. Fessler and S. D. Booth, "Conjugate-gradient preconditioning methods for shift-variant PET image reconstruction," *IEEE Transactions on Image Processing*, vol. 8(5), 1999.

[23] R. Gordon, R. Bender, and G. T. Herman, "Algebraic reconstruction techniques (ART) for three-dimensional electron microscopy and x-ray photography," *Journal of Theoretical Biology*, vol. 29, pp. 471–481, 1970.

[24] J. Wang, T. Li, H. Lu, and Z. Liang, "Penalized weighted least-squares approach to sinogram noise reduction and image reconstruction for low-dose X-ray computed tomography," *IEEE Transactions on Medical Imaging*, vol. 25, pp. 1272–1283, Oct 2006.

[25] B. Brendel, M. von Teuffenbach, P. B. Noël, F. Pfeiffer, and T. Koehler, "Penalized maximum likelihood reconstruction for x-ray differential phase-contrast tomography," *Medical Physics*, vol. 43, no. 1, pp. 188–194, 2016.

[26] C. Kamphuis and F. J. Beekman, “Accelerated iterative transmission ct reconstruction using an ordered subsets convex algorithm,” *IEEE Transactions on Medical Imaging*, vol. 17, pp. 1101–1105, Dec 1998.

[27] M. Leeser, S. Mukherjee, and J. Brock, “Fast reconstruction of 3D volumes from 2D CT projection data with GPUs,” *BMC Research Notes*, vol. 7, no. 1, pp. 1–8, 2014.

[28] V. G. Nguyen, J. Jeong, and S. J. Lee, “GPU-accelerated iterative 3D CT reconstruction using exact ray-tracing method for both projection and backprojection,” in *2013 IEEE Nuclear Science Symposium and Medical Imaging Conference (2013 NSS/MIC)*, pp. 1–4, Oct 2013.

[29] G. C. Sharp, N. Kandasamy, H. Singh, and M. Folkert, “GPU-based streaming architectures for fast cone-beam CT image reconstruction and demons deformable registration,” *Physics in Medicine and Biology*, vol. 52, no. 19, p. 5771, 2007.

[30] H. Scherl, B. Keck, M. Kowarschik, and J. Hornegger, “Fast GPU-based CT reconstruction using the Common Unified Device Architecture (CUDA),” in *2007 IEEE Nuclear Science Symposium Conference Record*, vol. 6, pp. 4464–4466, Oct 2007.

[31] F. Xu and K. Mueller, “Real-time 3D computed tomographic reconstruction using commodity graphics hardware,” *Physics in Medicine and Biology*, vol. 52, no. 12, p. 3405, 2007.

[32] E. Papenhausen and K. Mueller, “Rapid rabbit: Highly optimized GPU accelerated cone-beam CT reconstruction,” in *2013 IEEE Nuclear Science Symposium and Medical Imaging Conference (2013 NSS/MIC)*, pp. 1–2, Oct 2013.

[33] J. H. Cho and J. A. Fessler, “Accelerating ordered-subsets image reconstruction for x-ray CT using double surrogates,” 2012.

[34] L. A. Flores, V. Vidal, P. Mayo, F. Rodenas, and G. Verdú, “Iterative reconstruction of CT images on GPUs,” in *2013 35th Annual International Conference of the IEEE Engineering in Medicine and Biology Society (EMBC)*, pp. 5143–5146, July 2013.

[35] M. G. McGaffin and J. A. Fessler, “Alternating Dual Updates Algorithm for X-ray CT Reconstruction on the GPU,” *IEEE Transactions on Computational Imaging*, vol. 1, pp. 186–199, Sept 2015.

[36] S. V. Venkatakrishnan, L. F. Drummy, M. Jackson, M. D. Graef, J. Simmons, and C. A. Bouman, “Model-based iterative reconstruction for bright-field electron tomography,” *IEEE Transactions on Computational Imaging*, vol. 1, pp. 1–15, March 2015.

[37] S. V. Venkatakrishnan, L. F. Drummy, M. A. Jackson, M. D. Graef, J. Simmons, and C. A. Bouman, “A model based iterative reconstruction algorithm for high angle annular dark field-scanning transmission electron microscope (haadf-stem) tomography,” *IEEE Transactions on Image Processing*, vol. 22, pp. 4532–4544, Nov 2013.

[38] Y. Nesterov, “Efficiency of coordinate descent methods on huge-scale optimization problems,” *SIAM Journal on Optimization*, vol. 22, no. 2, pp. 341–362, 2012.

[39] A. Sabne, P. Sakdhnagool, and R. Eigenmann, “Effects of compiler optimizations in openmp to cuda translation,” in *International Workshop on OpenMP*, pp. 169–181, Springer Berlin Heidelberg, 2012.

[40] A. Sabne, P. Sakdhnagool, S. Lee, and J. S. Vetter, “Evaluating performance portability of OpenACC,” in *International Workshop on Languages and Compilers for Parallel Computing*, pp. 51–66, Springer International Publishing, 2014.