# cuMBIR: An Efficient Framework for Low-dose X-ray CT Image Reconstruction on GPUs

Xiuhong Li[1], Yun Liang[1,3,*], Wentai Zhang[1], Taide Liu[1], Haochen Li[1], Guojie Luo[1], Ming Jiang[2]
[1] Center for Energy-efficient Computing and Applications, School of EECS, Peking University, China
[2] Department of Information Science, School of Mathematical Sciences, Peking University, China
[3] State Key Laboratory of Computer Science, ICT, CAS, China
{lixiuhong,ericlyun,rchardx,tdliu,sodalee,gluo,ming-jiang}@pku.edu.cn

## ABSTRACT

Low-dose X-ray computed tomography (*XCT*) is a popular imaging technique to visualize the inside structure of object non-destructively. Model-based Iterative Reconstruction (*MBIR*) method can reconstruct high-quality image but at the cost of large computational demands. Therefore, *MBIR* often resorts to the platforms with hardware accelerators such as GPUs to speed up the reconstruction process.

For *MBIR*, the reconstruction process is to minimize an objective function by updating image iteratively. The X-ray source emits large amounts of X-rays from various views to cover the object as much as possible. Different X-rays always have complex and irregular geometric relationship. This inherent irregularity makes the minimization process of the objective function on GPUs very challenging. First, different implementations of the minimization of objective function have different impacts on the convergence and GPU resource utilization. To this end, we explore different solvers to the minimization problem and different parallelism granularities for GPU kernel design. Second, the complex and irregular geometric relationship of X-rays introduces irregular memory behaviors. Two nearby X-rays may intersect and thus incur memory collisions, while two far away X-rays may incur non-coalesced memory accesses. We design a unified thread mapping algorithm to guide the mapping from X-rays to threads, which can optimize the memory collisions and non-coalesced memory accesses together. Finally, we present a series of architecture level optimizations to fully release the horse power of GPUs. Evaluation results demonstrate that *cuMBIR* can achieve 1.48X speedup over the state-of-the-art implementation on GPUs.

## CCS CONCEPTS

• **Computing methodologies → Massively parallel algorithms**; • **Computer systems organization → Single instruction, multiple data**;

*Corresponding author.


*ICS '18, June 12–15, 2018, Beijing, China*

ACM ISBN 978-1-4503-5783-8/18/06…$15.00
https://doi.org/10.1145/3205289.3205309

## KEYWORDS

Low-dose XCT, Model-based Iterative Reconstruction, GPGPU

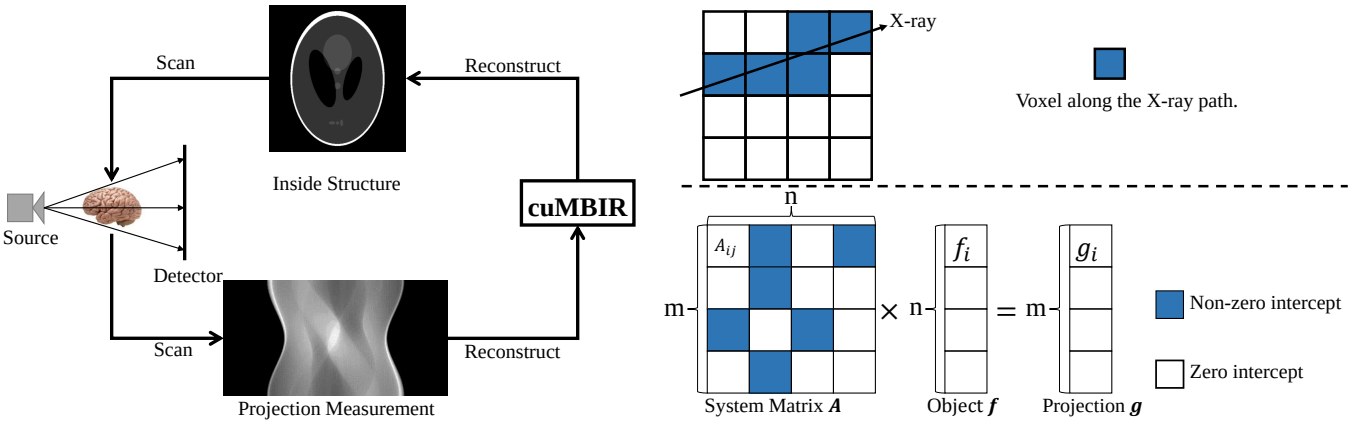
**ACM Reference Format:**
Xiuhong Li[1], Yun Liang[1,3,*], Wentai Zhang[1], Taide Liu[1], Haochen Li[1], Guojie Luo[1], Ming Jiang[2]. 2018. cuMBIR: An Efficient Framework for Low-dose X-ray CT Image Reconstruction on GPUs. In *ICS '18: 2018 International Conference on Supercomputing, June 12–15, 2018, Beijing, China.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3205289.3205309

## 1 INTRODUCTION

X-ray computed tomography (*XCT*) has been widely applied in non-destructive testing and contact-free inspection method, such as medical imaging, age determination and industrial materials testing [7]. It has received more and more attentions from high performance computing domain recently [23, 27, 28]. In general, an XCT equipment consists of an X-ray source, an array of X-ray detectors and a scanned object. The X-ray source emits large amounts of X-rays with an initial intensity. For each X-ray, its intensity will attenuate when it goes through the scanned object. The detector array will collect the attenuated intensity as the projection measurement. An X-ray corresponds to a projection measurement. XCT Image reconstruction is to obtain the inside structure of scanned object (i.e. the attenuation coefficients within the object) through large amounts of projection measurements.

For XCT image reconstruction, the research studies mainly lie on two problems: how to reduce the dosage to decrease the radiation exposure damage and how to develop efficient reconstruction algorithms to decrease the reconstruction time. In fact, the two problems are closely coupled. In the low-dose case, the lack of projection measurements, the X-ray scatter phenomenon and poly-energetic issues make the design of effective reconstruction algorithms challenging [8]. Model-based Iterative Reconstruction [26] (*MBIR*) has been proved as a good choice to generate high-quality images in the low-dose case. The prior knowledge is often used in *MBIR* in the form of regularizer to guarantee the image's smoothness within individual regions and segmentation across different regions [17, 21].

The reconstruction may take extremely long. For example, the typical reconstruction time of a 3D volume with size $512 \times 512 \times 512$ is more than half an hour on traditional multi-core CPUs [27, 36]. Thus, it is hard to meet the time requirements in the hospital where large amounts of patients are waiting for the diagnose results. To speed up the reconstruction process, XCT image reconstruction has been accelerated on GPUs [15, 16, 26, 39]. However, prior studies mainly focus on the reconstruction algorithms without the architectural insights of GPUs, and their differences mainly lie on the CT scanner geometry, such as helical XCT [26] and 3D cone-beam




(a) Illustration of XCT scan and reconstruction process.

(b) The mathematical description of XCT scan process.

**Figure 1: The preliminary of XCT image reconstruction.**

XCT [39]. [23] proposes high performance *MBIR* on GPUs. But it lacks the systematic study of solver selection, parallelism granularity, and memory optimization. Moreover, the asynchronous parallelization on GPUs will incur non-coalesced memory accesses, memory collisions and their coupling behaviors, which is ignored.

For *MBIR*, the reconstruction process is to minimize an objective function (i.e. model). How to select a GPU-friendly solver to the minimization process and determine the parallelism granularity for the GPU kernel design is the first challenge. In this work, we first explore different solvers and their impacts on GPU acceleration. We mainly consider two solvers: stochastic gradient descent *(SGD)* and iterative coordinate descent *(ICD)*. *ICD* has also been studied by some previous approaches [3, 24, 26]. Then, we explore different parallelism granularities for the GPU kernel design. We select a GPU-friendly implementation based on their impacts on the *MBIR* convergence and GPU resource utilization.

The complex and irregular geometric relationship of different X-rays leads to irregular memory access behaviors on GPUs, which is the second challenge. First, different X-rays have complex intersection relationship, which means that multiple threads may update (read or write) the same voxel (i.e. 3-D pixel) simultaneously. If multiple threads write to the same memory address simultaneously on GPUs, there is only one thread that can finish the write operation to that memory address unless using atomic operations, but atomic operations will degrade performance significantly [18]. Besides, due to complex and irregular geometric relationship, coalescing the memory accesses to reduce unnecessary memory transactions is also not trivial. To reduce memory collisions and improve memory coalescing, we design a unified thread mapping algorithm to guide the mapping from projections to threads. The thread mapping algorithm consists of two parts. We first apply graph coloring algorithm to divide the X-rays into different groups. Within each group, the X-rays have no intersections and thus no memory collisions. Then, for each group, we apply graph partitioning algorithm to map the X-rays into warps to reduce non-coalesced memory accesses.

For *MBIR*, the computations are single precision operations by default. However, modern GPUs are equipped with various *SIMT* computational resources with different precision ranges including FP16, FP32, and FP64. Besides, special functional units (*SFUs*) are also involved to support approximate computation for transcendental functions. To fully utilize these resources on GPUs, we propose to use mixed-precision computing technique within the range of allowed precision tolerance. In terms of memory resources on GPUs, read-only texture memory is designed for graphics applications where memory access patterns exhibit the spatial locality. Thus, we place read-only projection measurements into texture memory to improve memory throughput.

In summary, this paper develops *cuMBIR*, an *MBIR* solution on GPUs. The contributions are as follows:

- We first analyze two solvers (*ICD* and *SGD*) to the minimization problem and explore different parallelism granularities for GPU kernel design.
- We propose a unified thread mapping algorithm to optimize the memory collision problem and non-coalesced memory access problem together.
- We propose a series of architecture level optimizations to further improve performance including mixed-precision computing and on-chip memory optimization.

Evaluation results demonstrate *cuMBIR* can achieve 1.48X speedup on average over the state-of-the-art implementation (*GPU-ICD* [23]) on NVIDIA Volta 100 GPU with the high-quality reconstruction results.

## 2 BACKGROUND

In this section, we first introduce the preliminary of X-ray computed tomography. Then, we explain the theory of model-based iterative reconstruction method. Finally, we show the computational and memory resources on GPUs.

### 2.1 X-ray Computed Tomography

Low-dose XCT acts an important role in medical diagnosis and treatment. We illustrate the XCT using a cone-beam X-ray apparatus in Figure 1(a). The X-ray source emits an amount of X-rays

with an initial intensity. For each X-ray beam, its intensity will attenuate after it goes through the scanned object. Then, the detector will collect them as projection measurements. To fully cover the scanned object, the X-ray source and detector rotate around the scanned object to get large amounts of projection measurements from various views. For the typical XCT equipment, the number of views can be 90 to 180, and the array of detector is $512 \times 512$. The scan process from object to projection measurements with the reconstruction process from projection measurements to object are a pair of inverse processes.

*Beer-Lambert law* characterizes how the intensity decreases when an X-ray goes through an object.

$$I = I_0 \times exp(-p) \tag{1}$$

where $I_0$ is the initial intensity, $I$ is the attenuated intensity, and $p$ is the line integral of the linear attenuation coefficients along the path of the X-rays. As shown in the upper part of Figure 1(b), the scanned object is first discretized, and then the line integral can be formulated as inner production, where the first vector is intercepts of the voxels along the path and the second vector is the attenuation coefficients of these voxels.

The XCT scan process can be described as the multiplication of a system matrix and a voxel vector in the lower part of Figure 1(b).

$$\mathbf{A} \times \mathbf{f} = \mathbf{g} \tag{2}$$

where $\mathbf{A} \in \mathbb{R}^{m \times n}$ is the system matrix to simulate the path of all X-rays. $m$ is the number of X-rays, and $n$ is the number of voxels within the scanned object. Each row corresponds to an X-ray (a projection measurement). $\mathbf{A}_{i,j}$ is the intercept that the $i_{th}$ X-ray goes through the $j_{th}$ voxel. Zero represents the X-ray does not go through this voxel. It is obvious that for each X-ray, the number of voxels along its path is limited compared with the total number of voxels, and there are large amounts of zeros in each row of $\mathbf{A}$. Thus, $\mathbf{A}$ is a sparse matrix. For a given XCT equipment, we can either get the system matrix $\mathbf{A}$ offline and store it using the *CSR* format or compute it during runtime. $\mathbf{f} \in \mathbb{R}^n$ is the reconstructed image (i.e. the attenuation coefficient) and it is regarded as a column vector. $\mathbf{g} \in \mathbb{R}^m$ is the vector of projection measurements ($ln(I_0/I)$ obtained from Equation 1) of all X-rays.

## 2.2 Reconstruction Theory and Algorithm

Image reconstruction is to obtain the inside structure image $\mathbf{f}$ through the projection measurements $\mathbf{g}$. *MBIR* can generate high-quality image through the projection measurements with low sampling rate and low signal-to-noise ratio [7]. It is to minimize the following model:

$$E(\mathbf{f}) \triangleq D(\mathbf{g}, \mathbf{Af}) + \alpha R(\mathbf{f}) \tag{3}$$

where $D(\mathbf{g}, \mathbf{Af})$ is the fidelity term based on the projection measurements, $R(\mathbf{f})$ is the regularizer based on the prior knowledge, and $\alpha$ is the weight parameter of regularization. In general, different models use different $R(\mathbf{f})$.

For *MBIR*, there are various $R(\mathbf{f})$, such as Tikhonov and total variation. In this paper, we incorporate *Mumford-Shah functional* [17] as the regularizer, which can not only force the smoothness of the images within individual regions but also simultaneously guarantee segmentation across image edges[1]. Note that our technique is not restricted to the *Mumford-Shah* functional only, and can be applied to other regularizers as well.

## 2.3 Baseline GPU Architecture

A GPU is composed of several streaming multiprocessors (*SM*s) and *SM*s are connected with global memory through interconnection network. Within each *SM*, besides large amounts of registers, there are various caches and shared memory. Each *SM* is also equipped with various *SIMD* pipelines including *INT* units, *SP* units, *DP* units, and special function units (*SFU*s). The GPU global memory consists of large amounts of continuous segments with fixed size[2]. When a warp executes an instruction that accesses global memory, it coalesces the memory accesses of the threads within the warp into one or more memory transactions depending on the size of the word accessed by each thread and the distribution of the memory addresses across the threads. There are two kinds of irregular memory access behaviors, *memory collision* and *non-coalesced memory access*.

If a non-atomic instruction executed by a warp writes to the same memory address for more than one of the threads within one warp, there is only one thread that can finish the write operation to that memory address, and which thread performs the write operation is undefined [18]. This is called *memory collision*. Likewise, threads from different warps also incur memory collisions when they write to the same address simultaneously. If the memory accesses within a warp reside in multiple segments, it will cause more memory transactions with unnecessary data. This is called *non-coalesced memory access*. Because a memory transaction needs latency of hundreds of cycles, irregular memory behaviors often degrade the GPU memory throughput significantly.

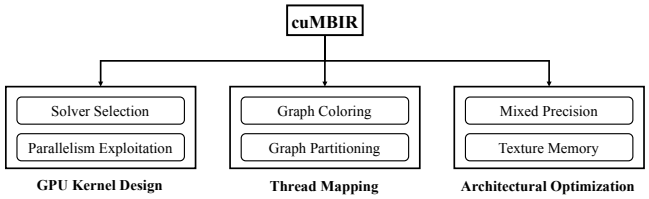



**Figure 2: The overview of *cuMBIR*.**

# 3 OVERVIEW OF *CUMBIR*

*MBIR* is an iterative method, and its evaluation criterion is the image quality and convergence speed. The convergence speed depends on the number of iterations and the time of each iteration. In this section, we present the overview of *cuMBIR*, an efficient framework for low-dose XCT image reconstruction on GPUs. As shown in Figure 2, the design of *cuMBIR* is composed of three key components including GPU kernel design, thread mapping, and architectural optimization.

For *MBIR*, the reconstruction process is to minimize an objective function by updating image iteratively. The objective function can be minimized by different solvers and different solvers have

[1] Its formula is $MS(f, K) \triangleq \beta \int_{\Omega \setminus K} |\nabla f|^2 dx + \lambda H^1(K)$. For simplicity, more details can be referred in [17].

[2] The size varies from data type of the memory access.

different impacts on the convergence and image quality. We first systematically explore two different solvers (i.e. *SGD* and *ICD*) to the objective function. Then, for each solver, we study two different granularities of parallelism on GPU kernel design. The details of GPU kernel design are in Section 4.

Different X-rays have irregular geometric relationship, and the inherent irregularity will make GPU acceleration challenging. On one hand, different X-rays may have intersections. There may exist large amounts of memory collisions when updating the image. How to reduce the number of memory collisions is critical for convergence. On the other hand, the irregular memory accesses make it hard to perform memory coalescing to reduce the number of off-chip memory transactions. Moreover, the tight coupling effect between them makes the optimization more challenging. We present an efficient thread mapping algorithm to alleviate the irregularities, which is detailed in Section 5. Finally, we propose a set of architecture level optimizations to further improve performance including mixed-precision computing and on-chip memory optimization in Section 6.

## 4 GPU KERNEL DESIGN

The key of GPU kernel design is parallelism exploitation. Different solver and parallelism granularities have significant impacts on parallelism exploitation. In this section, we first compare two popular solvers and then determine the parallelism granularity.

### 4.1 Solver Selection

There are many popular iterative optimization methods for finding the local minimum of an objective function, such as stochastic gradient descent (SGD) and iterative coordinate descent (ICD). To find a local minimum of an objective function, one takes steps proportional to the negative of the gradient of the objective function at the current point. On GPUs, parallel processes share a global address space that they read and write to asynchronously. For the synchronous parallel algorithms, multiple workers compute the gradient on a data shard in parallel and need a synchronization step before performing the next iteration. This synchronization will incur significant performance degradation [23]. On comparison, asynchronous parallel algorithms neglect these synchronization.

As the description of reconstruction theory and algorithm in Section 2, the objective function of XCT image reconstruction is sparse. [20] proves that the minimization of the sparse objective function can be parallelized without synchronization. Thus, this paper mainly considers asynchronous parallel *SGD* and *ICD* to solve the minimization problem. Both of them try to update image $\mathbf{f}$ iteratively to find the minimal of the objective function $D(\mathbf{g}, \mathbf{Af}) + \alpha R(\mathbf{f})$ in Equation 3, where $\mathbf{f}$ is n-dimension coordinate space and each dimension corresponds to a voxel. Their differences mainly lie on the update method at each step.

*SGD* uses the fact that the gradient points to the direction of local descent, and then at each step, it moves slowly towards the direction. However, for *ICD*, at each step, it only moves along one dimension, and the step size is determined by the gradient component in this dimension. Therefore, *SGD* picks a projection measurement to update image at each step, while *ICD* picks a voxel to update image at each step.

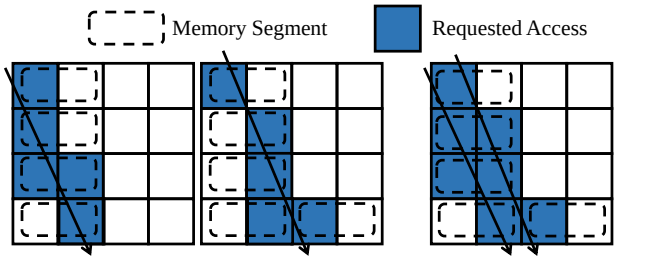
**Algorithm 1** Model-based Iterative Reconstruction.

**Input:** $f$ - image to be reconstructed, $g$ - sinogram, $\gamma$ - learning rate, $S$ - Solver
**Output:** $f$ - reconstructed image

```
1:  // SGD solver
2:  if S == SGD then
3:      while Image f has not converged do
4:          for each projection g_i from sinogram g do
5:              // Get the voxels this X-ray goes through
6:              Calculate v and w
7:              // update all voxels in v
8:              for all voxels ∈ v do
9:                  delta ← ∇(D(g, Af) + αR(f));
10:                 f ← f + γ × delta
11:             end for
12:         end for
13:     end while
14:     // ICD solver
15: else
16:     while Image f has not converged do
17:         for each voxel f_i in image f do
18:             // Get the X-rays that go through this voxel
19:             Calculate r
20:             delta ← 0
21:             for each X-ray in r do
22:                 // Get the voxels this X-ray goes through
23:                 Calculate v and w
24:                 // Get the delta for voxel f_i contributed by this X-ray
25:                 delta+ = ∇(D(g, Af) + αR(f));
26:             end for
27:             // update voxel f_i
28:             f_i ← f_i + γ × delta
29:         end for
30:     end while
31: end if
```

Algorithm 1 shows the process of *SGD* solver and *ICD* solver, respectively. The input includes the image $\mathbf{f}$, the projection measurements $\mathbf{g}$, the learning rate $\gamma$ and the selected solver $S$. We will first describe the *SGD* solver. The reconstruction process is composed of one outer loop (Line 3) and one inner loop (Line 4). An iteration of the outer loop is called as an *epoch*. In each epoch, all projections are picked to decrease Equation 3. A projection measurement corresponds to an X-ray. At each step, an X-ray is picked to perform update. For an X-ray, we first calculate its path, that is, the voxels it goes through and the intercepts ($\mathbf{v}$ and $\mathbf{w}$)(Line 6), as shown in Figure 1(b). Then, we compute the *delta* for each voxel along the path and update them (Line 9 to 10).

As for the *ICD* solver, its main difference with the *SGD* solver lies on the inner loop (Line 4 of *SGD* and Line 17 of *ICD*). At each step, one voxel is picked to perform update, instead of one projection measurement. For one voxel $f_i$ in image $\mathbf{f}$, we first get all the X-rays that go through this voxel (Line 19). Then, for each above X-ray, we get the *delta* contributed by this X-ray (Line 21 to 26). Finally, we update this voxel.




(a) A block is a worker. (b) A thread is a worker.

**Figure 3: Illustration of memory access pattern when a block or a thread acts as a worker of a projection on GPUs.**

Next, we perform the solver selection. *SGD* outperforms than *ICD* from three aspects. First, the *SGD* solver can update all the voxels along the X-ray path at each step. The latest update can be observed by other threads instantly. Second, the *ICD* solver needs large amounts of calculations of the X-ray path. Although they can be directly obtained from system matrix **A** offline, the matrix generally has very large size and it is unfeasible to reside in GPU device memory. Third, in prior works [23, 24, 26] where *ICD* have been applied, the requirement for image quality is not high. For the *Mumford-Shah* regularizer, the voxel-by-voxel update method of *ICD* has negative impacts on image smoothness and segmentation. Thus, we select *SGD* as the solver. More details will be presented in the experiment section.

## 4.2 Parallelism Granularity

For the *SGD* solver, the inner loop is designed as a kernel, and the outer loop is implemented by multiple times of kernel launch. When considering kernel design, we assign the workload of an X-ray projection to a worker. On GPUs, a worker can be a thread or a thread block. Thus, there are two different parallelism granularities. If the number of projections is not large, the latter can improve the GPU occupancy. In fact, in real XCT applications, there are large amounts of projection measurements, the former can also fully exploit the occupancy on GPUs.

When a worker is a block, there exist two disadvantages. The first disadvantage comes from the inherent serial feature of the calculation of the X-ray path (i.e. **v** and **w**). When computing them, only one thread within this thread block is utilized, leaving other threads idle. The second disadvantage lies on memory utilization. According to the coalescing mechanism on the GPU, each time a memory transaction will return a whole memory segment. We use a $4 \times 4$ object as an example, two adjacent voxels consist of a memory segment, as shown in Figure 3. We have two X-rays. The requested voxels of them are colored as blue box, and they both require to access 5 memory addresses. The top-left memory segment is notated as segment 0, and the right-down one is notated as segment 7. When a block acts as a worker shown in Figure 3(a), after we finish the update of X-ray 0 and X-ray 1 using two blocks, we access 9 memory segments in total. Thus, the memory bandwidth utilization is $\frac{5+5}{9\times2} \approx 55.56\%$. When a thread acts as a worker shown in Figure 3(b), we assume that the two threads corresponding to X-ray 0 and X-ray 1 are within the same warp. After we finish the update of X-ray 0 and X-ray 1, it needs to access segment 0, 2, 4, 4, 6, 6, 7 in order. Without consideration of locality, we access 7 memory segments with memory bandwidth utilization ($\frac{5+5}{7\times2} \approx 71.43\%$). Finally, we select the *SGD* solver with a thread as a parallel worker of a projection measurement. More performance details of different implementations will be provided in the experiment section.

# 5 THREAD MAPPING OPTIMIZATION

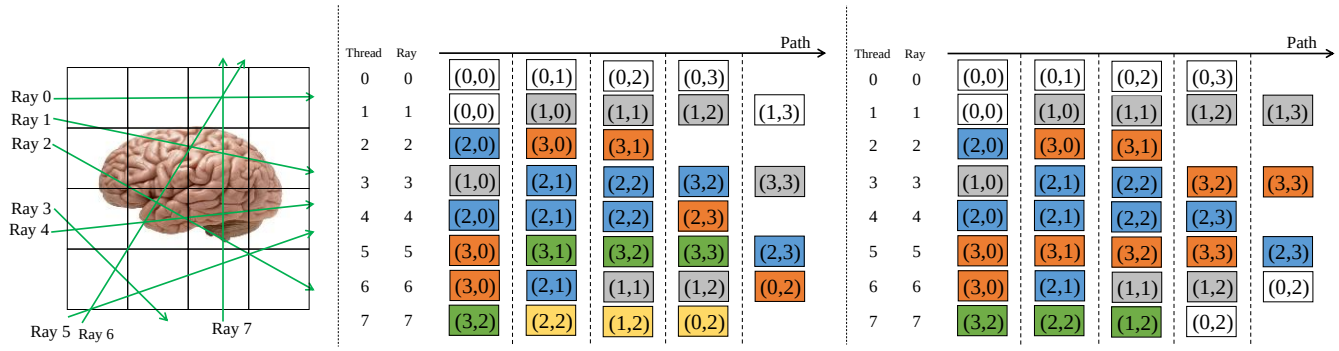
When considering GPU kernel design, we select the *SGD* solver with a thread as the worker of a projection measurement. In this section, we will present how to perform the mapping from projections to threads to alleviate the irregularities.

## 5.1 Memory Collision Problem

Each X-ray corresponds to a projection measurement (i.e. an element in **g**). It will update all of these voxels along its path. Due to the complex geometry relation, different X-rays will intersect on some voxels. As a result, during run-time, multiple threads may update the same voxles simultaneously. There are two kinds of memory collisions on GPUs: *intra-warp* collision and *inter-warp* collision. Intra-warp collisions indicate the conflicts that come from threads belonging to the same warp. Inter-warp collisions mean the conflicts that come from different warps. This can potentially be mitigated via the warp scheduler of the GPUs as long as the two conflicting warps are scheduled separately. Here, we focus on optimizing the *intra-warp* collision.

If we use non-atomic operation for image update, in each epoch, there are large amounts of updates that cannot be performed due to memory collision. Modern GPUs have provided support for atomic operations [18]. However, the overhead of atomic operation depends on the degree of memory collisions. If an atomic instruction executed by a warp writes to the same memory address for more than one of the threads of the warp, each access to that location occurs and they are all serialized, but the order in which they occur is undefined [18]. If we use atomic operation for image update, compared with non-atomic operations, it will increase the execution time of an epoch due to the expensive cost of atomic operation but reduce the number of epochs it needs to converge. Therefore, no matter we use atomic or non-atomic operations, the key issue is to reduce the memory collisions.

We use 8 X-rays to illustrate the complex geometry relationship in Figure 4(a). We list the voxels along each X-ray path at a row in Figure 4(b). The two-tuple within each voxel denotes its coordinate. When we design GPU kernel, we need to assign an X-ray to a thread. A thread will access the voxels along the path in order. For simplicity, we assume that warp size is four. By default, we assign the X-ray to the thread with the same ID, as shown in Figure 4(b). We use the same color to mark the same voxels in each column, which may be memory collisions. In this case, there is one memory collision in the first warp, and two memory collisions in the second warp. When we exchange X-ray 1 with X-ray 6, there will be one memory collision in the first warp, and no memory collisions in the second warp.




(a)X-rays with irregular geometric positions. (b) Memory collision problem. (c) Non-coalesced Memory problem.

**Figure 4: Illustration of irregular memory behaviors of *MBIR* on GPUs.**

## 5.2 Non-Coalesced Memory Access Problem

According to the coalescing memory mechanism presented in Section 2.3, memory accesses of the X-rays illustrated in Figure 4(a) cannot be coalesced together easily. For simplicity, we assume the two adjacent voxels in the same row are within the same memory segment in Figure 4(a). Similarly, we also assign the X-ray to the thread with the same ID, as shown in Figure 4(c). The voxels with the same color in the same column will reside in the same memory segment. We quantify the benefits from memory coalescing in the following way. If $N$ memory addresses can be coalesced within one memory segment, the benefit is defined as $N-1$. In this case, the score is 1 in the first warp, and 2 in the second warp. If we choose the same reorganization method as above (exchange X-ray 1 with X-ray 6), the score is 1 in the first warp, and 0 in the second warp. However, when we exchange X-ray 3 with X-ray 5, the score is 3 in the first warp, and 3 in the second warp. Therefore, a memory collision friendly scheme will alleviate the non-coalesced memory access problem and vice versa. Moreover, the coalesced memory accesses may incur memory collisions in turn. The two problems are closely coupled. How to perform thread mapping to address memory collision problem and non-coalescing memory access problem together is not trivial.

## 5.3 Optimization

Our object is to reduce memory collisions and improve memory coalescing. In general, the nearby X-rays tends to access the same memory address. Therefore, to reduce memory collisions, we prefer to choose the far-away X-rays to form a warp. On the contrary, to reduce non-coalesced memory accesses, we prefer to choose the near-by X-rays to form a warp. Thus, the optimization of non-coalesced memory accesses may introduce more memory collisions and vice versa. Prior works [30, 35, 37] only attempt to reduce the non-coalesced memory accesses, but they ignore the coupling effect with memory collisions. In this paper, we use thread mapping to unify the optimization of memory collision problem and non-coalesced memory access problem together.

When optimizing the irregular memory behaviors, our insight is to optimize the memory collision problem before non-coalesced memory access problem. This is because coalesced memory accesses will group memory accesses with the same or adjacent memory address together. However, this will aggravate the memory collision problem and in turn offset the reduction of non-coalesced memory accesses. We first incorporate graph coloring algorithm to reduce intra-warp memory collisions and then we perform graph partitioning for each color to reduce non-coalesced memory accesses.

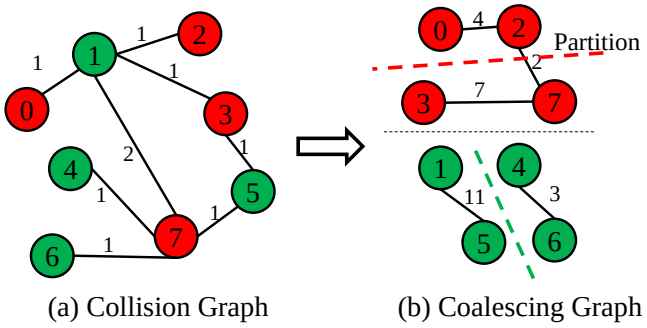



**Figure 5: The framework of thread mapping algorithm.**

Algorithm 2 presents the details. We build a memory collision graph $G(V, E)$ to represent the memory collisions shown in Figure 5(a). A node represents an X-ray, and the number is the X-ray's ID. The edge between two nodes represents that the two X-rays have memory collisions. We use the metric *collision degree* to quantify the memory collisions. In this paper, for two X-rays, *collision degree* is denoted as number of the voxels that both of them go through. The weight of edge between two nodes is the collision degree. We first perform graph coloring to eliminate intra-warp memory collisions (Line 2 in Algorithm 2). After this, the X-rays with the same color have no memory collisions. The graph coloring problem is NP-Complete, and there are no efficient polynomial algorithms so far. There have been many efficient heuristics, and we use *Welsh-Powell* algorithm [29] in this paper.

After graph coloring, we obtain a set of sub-graphs with different colors. Then, we consider to reduce non-coalesced memory accesses for each sub-graph (Line 4 to 18 in Algorithm 2). For each color, we first build a memory coalescing graph shown in Figure 5(b). The edge between two nodes represents the number of memory accesses that can be coalesced when they are grouped into the

**Algorithm 2** Thread mapping algorithm.

```
1: // Coloring process.
2: colors ← by invoking Welsh-Powell algorithm;
3: // Partitioning process.
4: for each color do
5:     build the coalescing graph G(V, E) for this color;
6:     N = ⌈(N_node/32)⌉;
7:     // partition G into N components
8:     METIS(G, N);
9:     for each component of G do
10:        if Sizeof(component) == 32 then
11:            Warps.push(component);
12:        else if Sizeof(component) > 32 then
13:            node = component.pop();
14:            vector.push(node);
15:            Warps.push(component);
16:        else
17:            node = vector.pop()
18:            component.push(node)
19:            Warp.push(component)
20:        end if
21:    end for
22: end for
23: Warps.push(vector);
```

**Output:** *Warps*

same warp. To quantify the memory coalescing, we define a metric called *coalescing degree*. For two X-rays, it means the number of voxels that can reside in the same segments. The weight of edge between two nodes represents the coalescing degree. Because the warp size is 32 in modern GPU, we need to partition the nodes within each color into sub-graphs, each of which contains 32 nodes. The number of components can be calculated as $\lfloor (N_{Node}/32) \rfloor$, where $N_{node}$ represents the number of nodes within this color. We perform graph partitioning to enhance memory coalescing as much as possible. For the graph $G(V, E)$ of each color, we use *METIS* [9] to perform graph partitioning (Line 8 in Algorithm 2).

As shown in Algorithm 2(Line 10 to 11), for the component with 32 nodes, the X-rays within this component is directly mapped as a warp. Next, let's consider the case where the number of nodes is not a multiple of 32. For the component with more than 32 nodes, we first pop the extra X-rays randomly and collect these X-rays in an independent vector. Then, the remained 32 X-rays within this component are mapped as a warp. For the component with less than 32 nodes, we select nodes from the above vector to form a warp. After all components have been processed, we map the X-rays in the vector to form warps. The output of Algorithm 2 is the thread mapping scheme. Finally, we distribute the warps into multiple thread blocks evenly to finish the thread mapping. The thread mapping algorithm is finished offline. The thread mapping scheme will be transferred to the GPU kernel as an argument. The thread mapping algorithm is only related to the geometry of the XCT scanner and object size. Thus, given an XCT equipment, for each body part, we only need to perform the off-line thread mapping one time.

# 6 ARCHITECTURE-LEVEL OPTIMIZATION

Apart from global memory optimization, further efforts on architecture level is necessary to fully exploit the horse power of GPUs. In this section, we first identify that mixed-precision computing can improve GPU computation resources and then we place the read-only projection measurements to the texture memory to further improve GPU on-chip memory resource utilization.

## 6.1 Mixed-precision Computing

For *MBIR*, the main operations are single precision. However, we find that using lower precision FP16 and higher precision FP64, it nearly has no impact on the *MBIR* convergence. Thus, to fully use the *SIMT* cores within GPUs, we propose mixed-precision *MBIR*. There are mainly four computation units in GPUs. *INT*, *SP* and *DP* units are designed for integer, FP32 (or FP16), and FP64 operations, respectively. At last, special functional units (*SFUs*) is used to accelerate the transcendental functions at the cost of accuracy. Different computation units can be used simultaneously.

*SP units.* FP16 operations are supported using half data type in modern Nvidia GPUs. Both FP16 and FP32 operations are executed in *SP* units. FP16 operations can reduce the memory traffic compared to FP32 operations with negligible accuracy loss. Thus, in this work, we use FP16 operations instead of FP32 operations.

*SFUs.* For the computation in Algorithm 1, when computing the path of an X-ray, trigonometric functions account for a large amount of time. Trigonometric functions can be implemented using *SFU*s with substantial performance improvement but tolerable accuracy loss [10]. We use $\_\_sinf(x)$ to take place of $sin(x)$ to apply *SFUs*. The performance benefits come from two-fold. Apart from the performance advantage of *SFUs*, it releases the *SP* units (otherwise occupied by the *sin* function). For *MBIR*, the precision loss caused by *SFUs* will not affect the convergence.

*DP units.* In original *MBIR*, there are no FP64 operations. However, large amounts of computations will exhaust the *SP* units on GPUs with *DP* units idle. In our implementation, we enable fine-grained FP64 operations by modifying the *PTX* code. We first claim a number of FP64 registers (*.reg .fp64 %fd<N>*), where *N* represents the number of required FP64 registers. For an arithmetic instruction with FP16 type, we first convert the operands to FP64 type (*cvt.f64.f16 %fd1, %h1;*). After type conversion, we perform FP64 operation (*add.f64 %fd3, %fd2, %fd1;*). Finally, we convert the results to FP16 again (*cvt.rn.f16.f64 %h3, %fd3;*). Considering that the ratio of *SP* units with *DP* units is 2:1, we select 1/3 instructions in an interleaved fashion and change their type from FP16 to FP64. Finally, we apply the just-in-time compilation mechanism to invoke the PTX code. In this way, we can exploit the instruction level parallelism (*ILP*) to fully utilize the *SP* units and *DP* units.

## 6.2 On-chip memory

Texture memory is designed for graphics applications where memory access patterns exhibit the spatial locality, which resides in the device memory. Besides shared memory, there are also two kinds of on-chip cache within an SM, unified Texture/L1cache. Because the projection measurements are read-only, we can place the projection

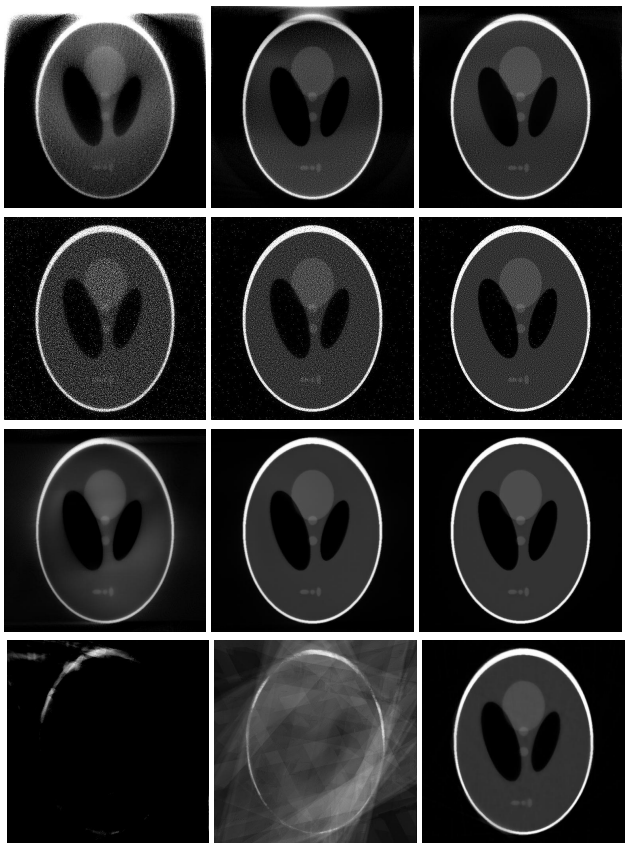


**Figure 6: The convergence process of the four different implementations. Each row shows the image reconstruction result of one implementation with number of epochs 1, 3 and 5, respectively. In top-down order, they are *ICD_Blk*, *ICD_Thd*, *SGD_Blk*, and *SGD_Thd*.**

measurements **g** into the texture memory, and it will be further cached by the unified Texture/L1 cache.

## 7 EXPERIMENT

We first introduce the methodology in Section 7.1. Then, we analyze the contribution of each component of *cuMBIR* in Section 7.2, 7.3 and 7.4, respectively. Finally, we present the overall performance and compare with the state-of-the-art work GPU-ICD [23] in Section 7.5.

**Table 1: Configuration of Platform.**

| Volta Platform | |
|---|---|
| CPU | 40-core, Intel Xeon CPU E5-2650 v3 @ 2.5GHz |
| GPU | Tesla Volta 100 (PCIe) with 84 SMs @ 1126MHz, 5376 FP32 cores, 5376 INT32 cores, 2688 FP64 cores, 672 Tensor Cores, and 336 texture units, 16 GB HBM2 memory with 900GB/s bandwidth |
| PCIe | 32 GB/sec |

**Table 2: 3D-Phantom Projection**

| Projection | Shepp-Logan | Hip | Abdomen | Jaw | Forbild |
|---|---|---|---|---|---|
| X | 256 | 512 | 512 | 128 | 512 |
| Y | 256 | 256 | 512 | 192 | 640 |
| Z | 256 | 256 | 256 | 256 | 512 |
| N | 11.8M | 18.0M | 23.6M | 11.7M | 23.6M |

### 7.1 Methodology

*Projection.* The 3D projection measurements used in this paper are generated using five different cone beam projections in Table 2, where $X$, $Y$ and $Z$ are the resolution of the 3D-phantoms, and $N$ is the number of projection measurements. Shepp-logan is a famous phantom and its description can be obtained from Wikipedia. The last four datasets can be found from CONRAD [14]. The forbild is also called head on the CONRAD website. Due to the space limit, we use *Shepp-Logan* as the representative phantom to analyze the contribution of each component of *cuMBIR*, and all five phantoms for overall evaluation. For *MBIR*, the evaluation criterion is the image quality and convergence speed. We use the root of mean-square error *RMSE* between the reconstructed image **f** with the original phantom to evaluate the image quality.

*Platform.* Table 1 shows the configuration of the used platform in this paper. The platform is composed of an Intel Xeon CPU E5-2650 v3 and an Nvidia Volta 100 GPU.

*Parameter.* There are two types of parameters: model parameters ($\alpha$ in Equation 3, $\beta, \lambda$ in *Mumford-Shah* regularizer) and learning step ($\gamma$). We explore different values of $\alpha, \beta and \lambda$. The parameters are as follows: $\alpha = 0.01, \beta = 0.001, \lambda = 0.05$. For learning rate $\gamma$, we notate the learning rate at epoch $t$ as $\gamma_t$. It monotonically decreases as the following routine:

$$\gamma_{t+1} = \frac{\gamma_t}{1 + 500\gamma_t} \tag{4}$$

where $\gamma_0$ is 0.001.

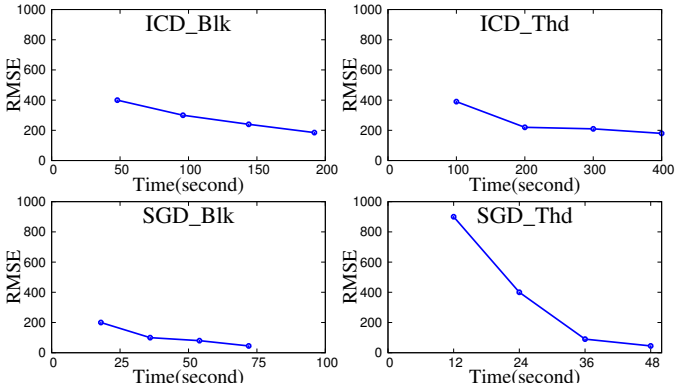
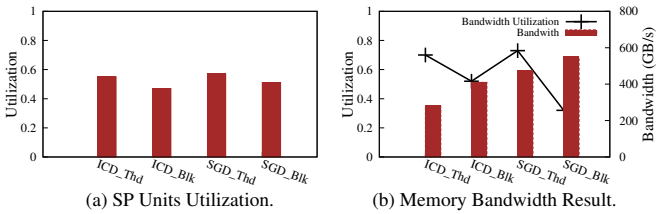



**Figure 7: Convergence process of the four different implementations.**




(a) SP Units Utilization. (b) Memory Bandwidth Result.

**Figure 8: *SP* units utilization, memory bandwidth and memory bandwidth utilization of the four different implementations.**

## 7.2 Performance Impact of Parallelism Exploitation

Figure 6 shows the the image reconstruction result of each epoch for the four implementations on GPUs:*ICD_Blk*, *ICD_Thd*, *SGD_Blk*, and *SGD_Thd* using *Shepp-Logan* phantom. We can find that *ICD*-based methods have worse image smoothness and segmentation, and they cannot converge well. This is because the coordinate-based update method cannot work well with the regularizer. The regularizer needs to update a voxel according to its neighbors. The *ICD* solver only updates a voxel at each step, and the latest values cannot be observed instantly.

Figure 7 shows the convergence process of the four different implementations. We can find *SGD_Thd* is the best implementation, it achieves 1.5X, 4.2X, and 8.3X speedup than *SGD_Blk*, *ICD_Blk* and *ICD_Thd*, respectively. *SGD_Blk* method cannot utilize the GPU resources fully because of serial process to compute the voxels an X-ray goes through. Thus, the execution time of kernel for *SGD_Blk* is longer than *SGD_Thd*. Figure 8(a) shows the *SP* units utilization, and we can find that a thread as a worker outperforms than a block as a worker. Similarly, Figure 8(b) shows the memory bandwidth and memory bandwidth utilization results. On the whole, *SGD_Thd* is the solution which is most suited for GPU acceleration.

## 7.3 Performance Impact of Thread Mapping

Figure 9 shows the convergence time of random mapping, default mapping and *cuMBIR* mapping algorithm. Random mapping means the mapping from X-rays to threads are randomly mapped, and default mapping means the thread ID equals with X-ray ID. We observe that random mapping and default mapping have nearly the same convergence process and that *cuMBIR* mapping can achieve about 1.4X convergence time speedup. The benefit comes from two aspects. The first is reduction of memory collisions. Figure 10(a) shows the memory collision results of default mapping and *cuMBIR* mapping with atomic and non-atomic, respectively. In the atomic case, the reduction of intra-warp memory collision degree can reduce the serialized memory transactions per request, and thus reduce the time of each iteration. In the non-atomic case, the reduction of intra-warp memory collision degree can reduce the number of updates that would be discarded, and thus reduce the number of iterations. In conclusion, our proposed solution can significantly reduce the degree of intra-warp memory collision, which further improve the convergence speed for both atomic and non-atomic case.

The second is the improvement of memory coalescing. The average memory transaction per request is a metric to indicate the memory coalescing. The ideal value is 1, which means perfect memory coalescing. The worst case is 32, which means that there are no any memory coalescing within a warp. Figure 10 shows the coalescing result comparison. We can find that our proposed techniques can reduce about 50% global memory transactions.

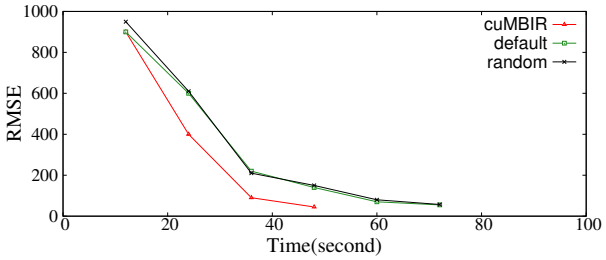
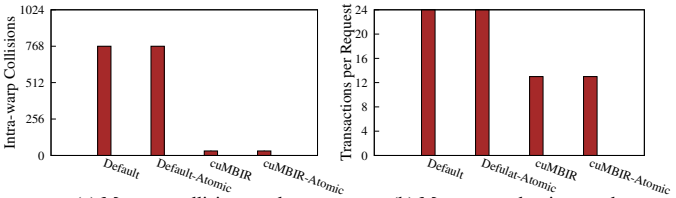



**Figure 9: Convergence process of three different mapping schemes: cuMBIR mapping, default mapping and random mapping.**




(a) Memory collision result. (b) Memory coalescing result.

**Figure 10: Improvement of irregular memory behaviors.**

## 7.4 Performance Impact of architecture level optimization

By mixed-precision computing, we can achieve about 43% convergence speedup. The special function units will decrease the overhead of triangle functions significantly with negligible precision loss. We use *DP* units through *ILP*, which can utilize both *DP* units and *SP* units. More clearly, *SFUs* can provide 30% speedup by itself and *DP* units further provide 13% convergence speedup. Unified Texture/L1 cache is also a read-only cache that is shared by all functional units and speeds up reads from the texture memory space, which resides in device memory. We can find that by on-chip memory optimization, we can achieve about 4% convergence speedup.

## 7.5 Overall Performance and Comparison

In this section, we will demonstrate that *cuMBIR* can achieve better performance than other existing XCT image reconstruction solutions on GPUs. We compare with the *GPU-ICD* solution proposed by Sabne et al. [23]. In their work, they only consider the *ICD* solver with a thread block as a parallel worker of a voxel. They compute **A** matrix offline and transfer to GPU texture memory to get rid of

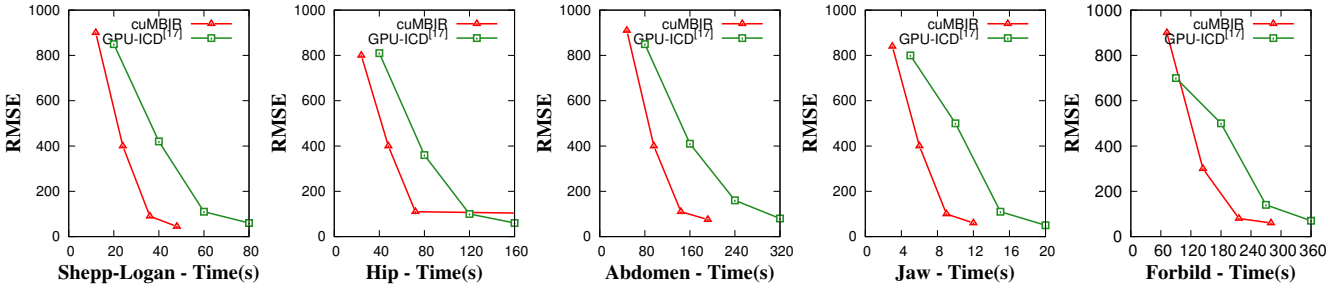



**Figure 11: Convergence process of the five popular phantom projections.**

the serial calculation of **r**. However, on one hand, it will take large amounts of time to read **A** from disk and then transfer **A** to texture memory, which will even hurt the performance. On the other hand, the size of **A** rapidly increases with the object size and number of X-rays. In terms of 3D scenario, it is unfeasible to accommodate **A** on GPU device memory. Moreover, our architectural optimizations can fully exploit the computation power of GPUs compared with *GPU-ICD*. Figure 11 shows the convergence process of five popular 3D phantom projections. We can find that our solution *cuMBIR* have 1.48X speedup for convergence time on average. Besides, the convergence time linearly increases with projection size.

## 8 RELATED WORK

*XCT Image Reconstruction on GPUs.* Low-dose XCT plays an important role in medical diagnose and treatment. The construction time and radiation exposure are the two key issues. There are two main approaches to the iterative construction [1]. The first approach comprises non-regularized methods, such as simultaneous iterative reconstruction techniques (SIRT) and algebraic reconstruction techniques (ART). The other approach is of regularized methods, which generally possess well-defined convergence criteria and generate higher quality images. The regularized methods need to minimize a cost function, which is computationally expensive. *MBIR* can reconstruct high-quality image but at the cost of large computational demands. [15, 16] discuss and demonstrate that GPU is applicable to X-ray CT image reconstruction. There are many studies that focus on the implementation of X-ray CT reconstruction on GPUs [4, 13, 23, 25]. However, these studies always lack architectural insights of GPUs.

*Asynchronous Parallel SGD.* The main characteristic of an asynchronous parallel algorithm is that its threads do not need to wait for inputs from other threads. Hogwild! [20] is an asynchronous approach for parallel SGD. [19] improves Hogwild! through some conflict groups and allocates them across cores for the multi-core CPU platform. Each core asynchronously updates the model without memory access conflicts. Moreover, there are also many studies which spend efforts on a specific application. [5, 34] propose efficient matrix factorization based on stochastic gradient descent on GPUs. However, all of these above studies either focus on the multi-core platform or a specific application. In this paper, we identify that asynchronous parallel SGD on GPUs will lead to irregular memory behaviors.

*Irregularity Optimization on GPUs.* The performance of GPUs is sensitive to the number of global memory transactions, and recent studies have focused on the non-coalesced memory accesses on GPUs. Many studies have proposed to data reorganization and job remapping to minimize non-coalesced memory accesses on GPU at compile time [2, 22, 35] and run-time [30, 37]. Apart from the global memory perspective, efficient cache management schemes have also been proposed for single kernel [32, 33] and concurrent kernel execution scenario [11, 12]. [31] focuses on the trade-off between singe thread performance and thread level parallelism through intelligent register allocation. [13, 38, 40] propose high performance implementations of *GEMM* and parallel sparse triangular solver on GPUs. However, these works cannot work well with the regularizer used in *MBIR*. Besides, memory collision on GPUs is also a critical factor of performance degradation. [6] explores the memory collision problem using atomic addition as an example and proposes to eliminate the memory collisions of atomic operations. However, those works either only study irregular memory reference or only explore memory access collisions.

## 9 ACKNOWLEDGEMENT

This work was supported by the National Natural Science Foundation China (No. 61520106004 and No. 61672048) and State Key Laboratory of Computer Architecture, Institute of Computing Technology, Chinese Academy of Sciences (No. CARCH201502).

## 10 CONCLUSION

*MBIR* can reconstruct high-quality images for low-dose X-ray computed tomography. However, it is inherently irregular due to the complex geometric relationship between X-rays. Thus, its parallelization on GPUs is a challenge. In this paper, we propose *cuMBIR*, an optimized model-based iterative reconstruction solution on GPUs. We first explore different implementations on GPUs. Then, we design a unified thread mapping scheme to optimize the memory collision problem and non-coalesced memory access problem together. Finally, we present a series of architecture level optimizations including mixed-precision computing and on-chip memory optimization. We evaluate *cuMBIR* using five famous medical phantoms, and it can achieve 1.48X speedup over the state-of-the-art GPU implementation.

## REFERENCES


[1] 2012. Iterative reconstruction methods in X-ray CT. *Physica Medica* 28, 2 (2012), 94 – 108.

[2] Muthu Manikandan Baskaran, Uday Bondhugula, Sriram Krishnamoorthy, J. Ramanujam, Atanas Rountev, and P. Sadayappan. 2008. A Compiler Framework for Optimization of Affine Loop Nests for Gpgpus. In *Proceedings of the 22Nd Annual International Conference on Supercomputing*. 225–234.

[3] Charles A Bouman and Ken Sauer. 1996. A unified approach to statistical tomography using coordinate descent optimization. *IEEE Transactions on image processing* 5, 3 (1996), 480–492.

[4] Liubov A Flores, Vicent Vidal, Patricia Mayo, Francisco Rodenas, and Gumersindo Verdú. 2014. Parallel CT image reconstruction based on GPUs. *Radiation Physics and Chemistry* 95 (2014), 247–250.

[5] Rainer Gemulla, Erik Nijkamp, Peter J. Haas, and Yannis Sismanis. 2011. Large-scale Matrix Factorization with Distributed Stochastic Gradient Descent. In *Proceedings of the 17th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. 69–77.

[6] Juan Gomez-Luna, Jose Maria Gonzalez-Linares, Jose Ignacio Benavides Benitez, and Nicolas Guil. 2013. Performance Modeling of Atomic Additions on GPU Scratchpad Memory. *IEEE Transactions on Parallel and Distributed Systems* 24, 11 (Nov 2013), 2273–2282.

[7] Willi A Kalender. 2006. X-ray computed tomography. *Physics in Medicine Biology* 51, 13 (2006), R29.

[8] Willi A Kalender. 2014. Dose in x-ray computed tomography. *Physics in Medicine Biology* 59, 3 (2014), R129.

[9] George Karypis. 1995. METIS - Unstructured Graph Partitioning and Sparse Matrix Ordering System, Version 2.0. (09 1995).

[10] Ang Li, Shuaiwen Leon Song, Mark Wijtvliet, Akash Kumar, and Henk Corporaal. 2016. SFU-Driven Transparent Approximation Acceleration on GPUs. In *Proceedings of the 2016 International Conference on Supercomputing*.

[11] Xiuhong Li and Yun Liang. 2016. Efficient Kernel Management on GPUs. In *Proceedings of the 2016 Conference on Design, Automation & Test in Europe*. 85–90.

[12] Yun Liang, Xiuhong Li, and Xiaolong Xie. 2017. Exploring Cache Bypassing and Partitioning for Multi-tasking on GPUs. In *Proceedings of the 36th International Conference on Computer-Aided Design*. 9–16.

[13] Weifeng Liu, Ang Li, Jonathan Hogg, Iain S Duff, and Brian Vinter. 2016. A Synchronization-Free Algorithm for Parallel Sparse Triangular Solves. In *Euro-Par*. 617–630.

[14] Andreas Maier, Hannes G Hofmann, Martin Berger, Peter Fischer, Chris Schwemmer, Haibo Wu, Kerstin MÃijller, Joachim Hornegger, Jang-Hwan Choi, Christian Riess, Andreas Keil, and Rebecca Fahrig. 2013. CONRAD-A software framework for cone-beam imaging in radiology. 40 (11 2013), 111914.

[15] K. Mueller and Fang Xu. 2006. Practical considerations for GPU-accelerated CT. In *3rd IEEE International Symposium on Biomedical Imaging: Nano to Macro, 2006.* 1184–1187.

[16] Klaus Mueller, Fang Xu, and Neophytos Neophytou. 2007. Why do commodity graphics hardware boards (GPUs) work so well for acceleration of computed tomography? *Computational Imaging* 6498 (2007).

[17] David Mumford and Jayant Shah. 1989. Optimal approximations by piecewise smooth functions and associated variational problems. *Communications on Pure and Applied Mathematics* 42, 5 (1989), 577–685.

[18] NVIDIA. 2017. CUDA C Programming Guide. http://docs.nvidia.com/cuda/cuda-c-programming-guide. (2017).

[19] Xinghao Pan, Maximilian Lam, Stephen Tu, Dimitris Papailiopoulos, Ce Zhang, Michael I Jordan, Kannan Ramchandran, and Christopher Ré. 2016. Cyclades: Conflict-free asynchronous machine learning. In *Advances in Neural Information Processing Systems*. 2568–2576.

[20] Benjamin Recht, Christopher Ré, Stephen J. Wright, and Feng Niu. 2011. Hogwild: A Lock-Free Approach to Parallelizing Stochastic Gradient Descent. In *25th Annual Conference on Neural Information Processing Systems*. 693–701.

[21] Ludwig Ritschl, Frank Bergner, Christof Fleischmann, and Marc Kachelriess. 2011. Improved total variation-based CT image reconstruction applied to clinical data. *Physics in Medicine Biology* 56, 6 (2011), 1545.

[22] Shane Ryoo, Christopher I. Rodrigues, Sara S. Baghsorkhi, Sam S. Stone, David B. Kirk, and Wen-mei W. Hwu. 2008. Optimization Principles and Application Performance Evaluation of a Multithreaded GPU Using CUDA. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 73–82.

[23] Amit Sabne, Xiao Wang, Sherman J. Kisner, Charles A. Bouman, Anand Raghunathan, and Samuel P. Midkiff. 2017. Model-based Iterative CT Image Reconstruction on GPUs. In *Proceedings of the 22Nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 207–220.

[24] K. Sauer and C. Bouman. 1993. A local update strategy for iterative reconstruction from projections. *IEEE Transactions on Signal Processing* 41, 2 (Feb 1993), 534–548.

[25] GC Sharp, N Kandasamy, H Singh, and Michael Folkert. 2007. GPU-based streaming architectures for fast cone-beam CT image reconstruction and demons deformable registration. *Physics in medicine and biology* 52, 19 (2007), 5771.

[26] Jean-Baptiste Thibault, Ken D Sauer, Charles A Bouman, and Jiang Hsieh. 2007. A three-dimensional statistical approach to improved image quality for multislice helical CT. *Medical physics* 34, 11 (2007), 4526–4544.

[27] Xiao Wang, Amit Sabne, Sherman Kisner, Anand Raghunathan, Charles Bouman, and Samuel Midkiff. 2016. High Performance Model Based Image Reconstruction. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 1–12.

[28] Xiao Wang, Amit Sabne, Putt Sakdhnagool, Sherman J. Kisner, Charles A. Bouman, and Samuel P. Midkiff. 2017. Massively Parallel 3D Image Reconstruction. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–12.

[29] D. J. A. Welsh and M. B. Powell. 1967. An upper bound for the chromatic number of a graph and its application to timetabling problems. *Comput. J.* 10, 1 (1967), 85–86.

[30] Bo Wu, Zhijia Zhao, Eddy Zheng Zhang, Yunlian Jiang, and Xipeng Shen. 2013. Complexity Analysis and Algorithm Design for Reorganizing Data to Minimize Non-coalesced Memory Accesses on GPU. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 57–68.

[31] Xiaolong Xie, Yun Liang, Xiuhong Li, Yudong Wu, Guangyu Sun, Tao Wang, and Dongrui Fan. 2015. Enabling Coordinated Register Allocation and Thread-level Parallelism Optimization for GPUs. In *Proceedings of the 48th International Symposium on Microarchitecture*. 395–406.

[32] Xiaolong Xie, Yun Liang, Guangyu Sun, and Deming Chen. 2013. An Efficient Compiler Framework for Cache Bypassing on GPUs. In *Proceedings of the International Conference on Computer-Aided Design*. 516–523.

[33] Xiaolong Xie, Yun Liang, Yu Wang, Guangyu Sun, and Tao Wang. 2015. Coordinated static and dynamic cache bypassing for GPUs. In *21st IEEE International Symposium on High Performance Computer Architecture*. 76–88.

[34] Xiaolong Xie, Wei Tan, Liana L. Fong, and Yun Liang. 2017. CuMF_SGD: Parallelized Stochastic Gradient Descent for Matrix Factorization on GPUs. In *Proceedings of the 26th International Symposium on High-Performance Parallel and Distributed Computing, HPDC 2017, Washington, DC, USA, June 26-30, 2017*. 79–92.

[35] Yi Yang, Ping Xiang, Jingfei Kong, and Huiyang Zhou. 2010. A GPGPU Compiler for Memory Optimization and Parallelism Management. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*. 86–97.

[36] Zhou Yu, Jean-Baptiste Thibault, Charles A Bouman, Ken D Sauer, and Jiang Hsieh. 2011. Fast model-based X-ray CT reconstruction using spatially nonhomogeneous ICD optimization. *IEEE Transactions on image processing* 20, 1 (2011), 161–175.

[37] Eddy Z. Zhang, Yunlian Jiang, Ziyu Guo, Kai Tian, and Xipeng Shen. 2011. On-the-fly Elimination of Dynamic Irregularities for GPU Computing. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*. 369–380.

[38] Xiuxia Zhang, Guangming Tan, Shuangbai Xue, Jiajia Li, Keren Zhou, and Mingyu Chen. 2017. Understanding the GPU Microarchitecture to Achieve Bare-Metal Performance Tuning. In *Proceedings of the 22Nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 31–43.

[39] Xing Zhao, Jing-jing Hu, and Peng Zhang. 2009. GPU-Based 3D Cone-Beam CT Image Reconstruction for Large Data Volume. *International Journal of Biomedical Imaging* 2009, 149079 (2009), 8.

[40] Keren Zhou, Guangming Tan, Xiuxia Zhang, Chaowei Wang, and Ninghui Sun. 2017. A Performance Analysis Framework for Exploiting GPU Microarchitectural Capability. In *Proceedings of the International Conference on Supercomputing*. 1–10.