A Software-Hardware Co-design Solution for 3D Inner Structure Reconstruction

Xingchen Li[1,3], Zhe Zhou[1,2,3], Qilin Zheng[1,3], Guangyu Sun[1,3*], Qiankun Wang[1,3], Chenhao Xue[1,3]
[1]School of Integrated Circuits, Peking University. [2]School of Computer Science, Peking University. [3]Beijing Advanced Innovation Center for Integrated Circuits.

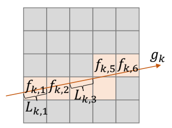
Abstract

Volume imaging (3D model with inner structure) is widely applied to various areas, such as medical diagnosis and archaeology. Especially during the COVID-19 pandemic, there is a great demand for lung CT. However, it is quite time-consuming to generate a 3D model by reconstructing the internal structure of an object. To make things worse, due to the **poor data locality** of the reconstruction algorithm, researchers are pessimistic about accelerating it with ASIC. Besides the locality issue, we find that the **complex synchronization** is also a major obstacle for 3D reconstruction. To overcome the problems, we propose a holistic solution using software-hardware co-design. We first provide a unified programming model to cover various 3D reconstruction tasks. Then, we redesign the dataflow of the reconstruction algorithm to improve data locality. In addition, we remove unnecessary synchronizations by carefully analyzing the data dependency. After that, we propose a novel near-memory acceleration architecture, called Waffle, for further improvement. Experiment results show that Waffle in a package can achieve 3.51× ∼ 3.96× speedup over a cluster of 10 GPUs with 9.35× ∼ 10.97× energy efficiency.

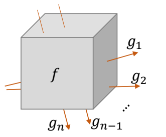
ACM Reference Format:
Xingchen Li[1,3], Zhe Zhou[1,2,3], Qilin Zheng[1,3], Guangyu Sun[1,3*], Qiankun Wang[1,3], Chenhao Xue[1,3]. 2024. A Software-Hardware Co-design Solution for 3D Inner Structure Reconstruction. In *61st ACM/IEEE Design Automation Conference (DAC '24), June 23–27, 2024, San Francisco, CA, USA.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3649329.3656522

1 Introduction

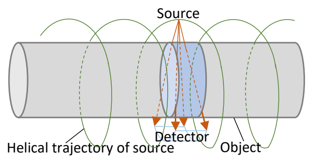
3D volume imaging has been widely applied to various areas, such as medical diagnosis (e.g. CT and PET) [11, 12], microscopic analysis [9], non-destructive inspection (e.g. customs inspection) [2], archaeology [5], and geological exploration [6]. Especially during the COVID-19 pandemic, there is a great demand for lung CT. Meanwhile, low-dose CT is recommended for its reduced cumulative radiation dose, which is beneficial for human health. However, reconstructing the 3D model with low-dose CT is extremely slow, which cannot match the increasing number of patients nowadays. For example, the CT reconstruction process of an 874×874×161 image consumes 6506s on a 16-core Xeon CPU. Considering the wide range of applications and the recent real problems, it is necessary to accelerate 3D inner-structure reconstruction.



(a) 2D illustration.



(b) 3D schematic diagram.



(c) Helical CT example.

Figure 1: Diagram of reconstruction problem.

Unfortunately, due to the extremely poor locality of 3D inner reconstruction, researchers are pessimistic about accelerating it with ASIC. Therefore, unlike 2D and 3D-surface reconstruction, which have many ASIC realizations, we find only FPGA and GPU realizations [11, 12] targeted at CT reconstruction, which still cannot solve the locality issues, and are not general for 3D inner-structure reconstruction.

To further understand the dilemma, we now briefly introduce the 3D inner reconstruction problem. A 3D inner structure is normally represented as a 3D array, each element representing the density information of the corresponding voxel. The common method is to irradiate the target object with rays from different directions. And a detector is used to measure the attenuation of the rays after penetrating the object. Then, the internal structure is reconstructed so that the attenuation of this reconstructed model can match the detected results. The reconstruction process is often realized by the iterative gradient descent approach, and synchronization is required in each iteration[11, 12].Because the rays pass through the object from various angles, the locality is extremely bad.

Besides the (1) poor locality, we find that the (2) significant synchronization overhead also limits the performance gain. Moreover, previous works had to (3) split the task into many parts because of these two issues. Each part is treated with gradient descent individually. The lack of global information greatly slows down the convergence speed of the gradient descent algorithm.

In this paper, we propose a software-hardware co-design solution, which solves all the above problems. We first formulate the 3D reconstruction problem and abstract a universal programming model for it. With this programming model, we analyse why the existing algorithm is not suitable for acceleration. After that, we propose the dataflow optimizations, including a voxel-driven manner to improve locality, some pre-computation to reduce redundant run-time calculations, a simplified synchronization scheme, etc.

After optimizing the algorithm, we further propose a hardware accelerator, called Waffle, to accelerate it. It can be realized by customizing the computation logic of existing near-memorey-computing techniques, such as HMC and HBM. The contributions of this paper are listed as follows.

[*]Corresponding author: Guangyu Sun, gsun@pku.edu.cn.

- Different from previous works for specific scenarios such as CT, we formulate the general 3D inner reconstruction problem, and present a unified programming model.
- We analyse the issues of the existing algorithm and redesign a hardware-friendly one, making ASIC acceleration practical.
- We propose a novel near-memory-computing architecture to further accelerate our new algorithm.
- Experimental results show that the holistic solution within a package can achieve 3.51× ∼ 3.96× speedup over a cluster of 10 GPUs with 9.35× ∼ 10.97× energy efficiency.

2 Problem Formulation

In this section, we formulate the 3D ray-driven iterative reconstruction problem. We begin with a simple 2D case for demonstration, as Figure 1a shows. The aim of reconstruction is to get the attenuation coefficient f of each pixel (voxel in 3D case). When a beam k penetrates the object (e.g. the arrow in Figure 1a), its attenuation can be represented as $\Sigma_i f_{k,i} L_{k,i}$, where $f_{k,i}$ means the attenuation coefficient of the i-th pixel that beam k goes through, and $L_{k,i}$ indicates the intersection length of beam k and the i-th pixel. This process is called *forward projection*. After the beam goes out of the object, its attenuation degree g_k will be detected. Then, $f_{k,i}$ will be updated to minimize $(\Sigma_i f_{k,i} L_{k,i} - g_k)^2$. This process is called *backward projection*, which is realized by the gradient descent algorithm, i.e., $f_{k,i}$ is updated with $2(\Sigma_i f_{k,i} L_{k,i} - g_k) L_{k,i}$.

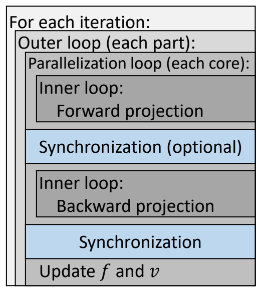
However, in practice, the simple backward projection above usually causes non-smoothness of f and oscillating edges of reconstructed objects. This conflicts with the smooth feature of natural objects, thus decreasing reconstruction quality. Therefore, Mumford-Shah functional, which measures the smoothness of both f and edges, is commonly introduced to the objective function [7, 11], i.e., we need to minimize $(\Sigma_i f_{k,i} L_{k,i} - g_k)^2 + MS$, where MS denotes the Mumford-Shah functional. It introduces an extra tensor v, which represents the probability of each voxel belonging to the edge of the reconstructed object. Tensor v and f are optimized together to minimize the objective function.

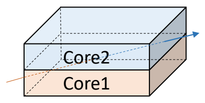
Although the Mumford-Shah functional is very complex, we do not need to solve it. To minimize it with the gradient descent algorithm, we only need to calculate its gradient using Equation 1. Here, α, β and ϵ are hyper-parameters. $\boldsymbol{\nabla}\cdot$, ∇, and Δ are 2D operators[1] of divergence, Nabla, and Laplacian, respectively.

$$\begin{cases} \dfrac{\partial MS}{\partial f} = -2\alpha \boldsymbol{\nabla} \cdot \left(v^2 \nabla f \right) \\ \dfrac{\partial MS}{\partial v} = 2\alpha |\nabla f|^2 v + \dfrac{\beta}{2\epsilon}(v-1) - 2\beta\epsilon\Delta v \end{cases} \tag{1}$$

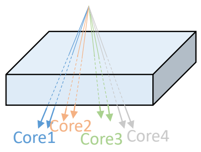
To get a good reconstruction result, multiple beams from many directions that cover the whole object are needed, as Figure 1b depicts. There are many kinds of projection geometry that can be used, e.g. parallel beams, fan beams, and cone beams. Figure 1c gives an example of helical CT with cone beams. The X-ray source moves in a circle while the patient moves forward. Thus, the trajectory of the X-ray source is a spiral relative to the patient. The source launches a cone of beams at each position. A detector rotates simultaneously with the source, and detects the attenuation degree of each beam.

[1]In theory, 3D operators should be used, which causes unacceptable poor locality. Therefore, all prior works use 2D operators in practice.





(b) Divide volumes for cores.



(a) Algorithm overview. (c) Divide beams for cores.

Figure 2: (a)Programming model of the iterative reconstruction algorithm. The first synchronization is required for the method in (b), but not required for the method in (c).

3 Programming Model

After formulating the 3D reconstruction problem, we now abstract the programming model to solve it. The programming model is summarized as three nested loops, as Figure 2a describes.

- The outer loop divides the task into parts according to voxels or rays. For a heavy task, the locality and synchronization issues both become severe. Thus previous works had to divide it. However, the division losses global information, thus the algorithm becomes harder to converge. We aim to improve the locality and synchronization issues so as to eliminate this loop.
- The parallelization loop is distributed to each core. A core first conducts the forward projection of all its corresponded beams. Then, synchronization may be required according to how tasks are divided to each core. For example, in Figure 2b, each of Core1 and Core2 processes part of the forward projection of the beam. Thus synchronization is needed to obtain the integral projection. Next, the core conducts backward projection. Finally, there is synchronization among all the cores in a part before f and v are updated.
- The inner loop is the forward and backward projection of the beams processed by a core.

4 Software Optimizations

In this section, we analyse the drawbacks of the existing algorithm and give corresponding optimizations, which will finally lead to our new architecture in Section 5.

4.1 Ray-driven or Voxel-driven?

The two most important elements in 3D inner reconstruction are **rays** (beams) and **voxels**. Straightforwardly, there are two ways to conduct forward and backward projection: (1) Traverse the beams. For each beam, find all the voxels it passes through. We call this method "ray-driven" reconstruction. (2) Traverse the voxels. For each voxel, find all the beams that pass through it. We call this method "voxel-driven" reconstruction.

When traversing beams, the accesses to voxels are irregular, and vice versa. That is the reason for the bad locality of 3D inner reconstruction problems. The bad locality in turn induces ineffectiveness in utilizing the cache or on-chip BRAM. For example, in our realization of a previous FPGA design [11], 58.5% of time is consumed by pipeline stall due to BRAM misses.

Previous works [11, 12] usually adopt the ray-driven method. Their logic is as follows. The bad locality and synchronization overhead make it necessary to split the whole task into parts. In this case, the voxel-driven method causes many redundant calculations. Take the helical CT in Figure 1c as an example. Previous works [11, 12] regard voxels and beams related to each source position as a part. The related volumes are shown in blue in Figure 1c. In fact, many voxels in the blue cylindrical region are not penetrated by the cone beams in the part. Traverse on these voxels is wasted. Besides, adjacent parts share a large number of voxels. Still take Figure 1c as an example, when the source position moves a little following the helical trajectory, its related volumes (the blue cylindrical region) also move a little. Thus many voxels are shared between these two source positions (two parts). A shared voxel needs to be processed redundantly in each part. Therefore, they use the ray-driven method to avoid these redundant calculations.

However, we find that, with some modifications, the voxel-driven method deals well with the locality problem. We will further mitigate the synchronization overhead in Section 4.2. Then, we do not need to divide the task into small parts, thus the voxel-driven method does not have the redundant calculation issue. In this way, it can beat the previous ray-driven method with a better locality.

Take the forward projection as an example. The key point of our new method is that, for each voxel, we store its attenuation coefficient f, and the id of all the beams that pass through it. We also pre-compute the intersection length "L" of these beams with the voxel, which are also stored together in the voxel data structure. The detailed data structure will be shown later in Section 5.2. In this way, when traversing the voxels, we can easily obtain f and L with good spatial locality, and calculate fL. Then, we update the attenuation degree of the related beam by adding fL. Although the updates to the beam structure still present a bad locality, it is not on the critical path. The traverse on the voxels does not need to stall to wait for the updates to beams. Besides, we design an *Exclusive Updater*, which will be introduced in Section 5, to buffer, integrate, and schedule the update commands. This design can further reduce the overhead of the updates.

4.2 Real data dependency

We reduce the synchronization overhead by carefully analysing the data dependency, and removing unnecessary synchronizations. We summarize the data dependency in Figure 3. Our key observations are as follows:

- Backward projection can begin simultaneously with forward projection. Previous works begin backward projection after forward projection finishes. However, recall that the backward projection can be divided into three parts: calculating $2(\Sigma fL - g)L$, $\frac{\partial MS}{\partial f}$, and $\frac{\partial MS}{\partial v}$. As shown in Figure 3, the last two have no dependency on the forward projection. Therefore, they can be started simultaneously with forward projection.
- Unlike the ray-driven method, our voxel-driven method does not require the strict synchronization in Figure 2a, which split the stage of "backward projection" and "updating f and v". This is because both f and v are features of voxels. In voxel-driven method, after a voxel is processed, its f and v can be updated directly, which will not influence other voxels.

The above two observations remove the most expensive synchronizations in previous works. However, there are still some other necessary synchronizations, which we will solve by our hardware architecture designs. We list these synchronizations as follows.

- The optional synchronization between forward and backward projection in Figure 2a becomes necessary. This is because our voxel-driven method divides volumes for cores as in Figure 2b, thus beams may go across different cores. We will realize this synchronization scalably and efficiently using asynchronous communication between adjacent cores in Section 5.
- Updating $\Sigma fL - g$ (beam feature) in Figure 3 in forward projection may cause potential write conflicts, because different voxels may be penetrated by the same beam, and we process the voxels in parallel.
- Updating f (voxel feature) using $2(\Sigma fL - g)L$ (depending on beam feature $\Sigma fL - g$) in Figure 3 in backward projection may cause potential write conflicts, because one voxel can be penetrated by multiple beams, and we process the beams in parallel.

As will be shown in Section 5, for both two kinds of write conflict issues, we can design an exclusive updater to ensure the exclusive updates to voxels and beams.

4.3 New Dataflow

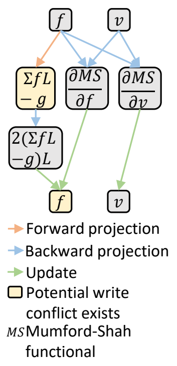
With the above algorithm optimizations, our new dataflow changes to Figure 4. The main difference from Figure 2a and the advantages are summarized as follows.

- The inner loop becomes voxel-driven to improve locality.
- We calculate $\frac{\partial MS}{\partial v}$ simultaneously with forward projection. The forward projection and the calculation of $2(\Sigma fL - g)L$ share multiplication logic, during the calculation of $\frac{\partial MS}{\partial v}$ and $\frac{\partial MS}{\partial f}$ share gradient logic. Therefore, we can reuse hardware in the two stages of the inner loop.
- The first synchronization becomes necessary, while the second one is removed. Note that the second one means extremely high overhead, because all the intermediate results should be stored before updating f and v. This also induces scalability and locality issues. However, the first synchronization can be realized scalably and efficiently, as will be shown in Section 5.
- The updates to f and v are directly done in the calculation process, thus the time is overlapped.
- Now that locality and synchronization issues are solved, we can remove the outer loop, thus the algorithm can converge faster.

5 Accelerator Design

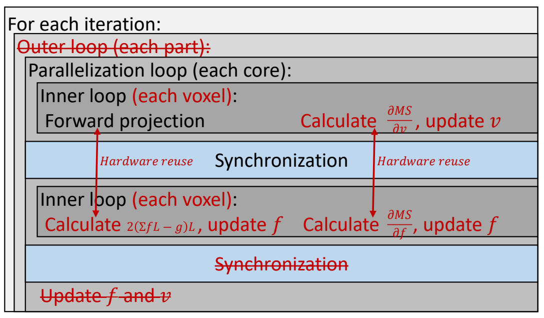
5.1 Overview

We design a hardware accelerator, called Waffle, whose overall architecture is shown in Figure 5. Note that, although we use HMC-based Waffle as an example for legibility, it is similar to extend other near-memory-computing techniques like HBM to support Waffle. Recall that the parallelization loop in our algorithm is distributed to multi-cores. The structure of each core is presented at the upper left of Figure 5, which will be introduced later. Each core is fabricated on a logical die, which is connected to several memory dies using through-silicon-via (TSV). The logical die and the memory dies constitute a cube. 16 cubes are connected as a mesh, which constitutes a package. The system can be scaled up by using more packages with PCIe connection.



f
v
$\Sigma fL - g$
$\frac{\partial MS}{\partial f}$
$\frac{\partial MS}{\partial v}$
$2(\Sigma fL - g)L$
f
v
Forward projection
Backward projection
Update
Potential write conflict exists
MS Mumford-Shah functional

Figure 3: Data dependency.



For each iteration:
~~Outer loop (each part):~~
Parallelization loop (each core):
Inner loop (each voxel):
Forward projection
Calculate $\frac{\partial MS}{\partial v}$, update v
Hardware reuse
Synchronization
Hardware reuse
Inner loop (each voxel):
Calculate $2(\Sigma fL - g)L$, update f
Calculate $\frac{\partial MS}{\partial f}$, update f
~~Synchronization~~
~~Update f and v~~

Figure 4: Our optimized dataflow.

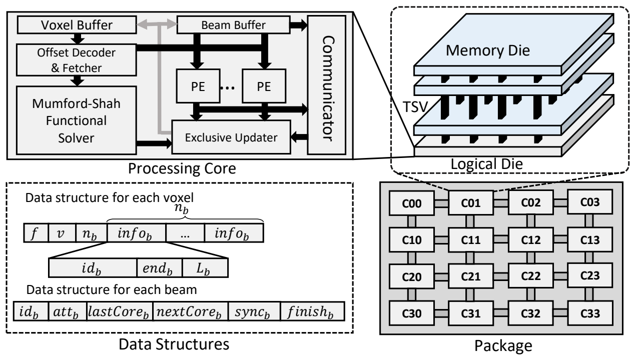
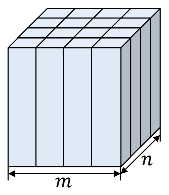
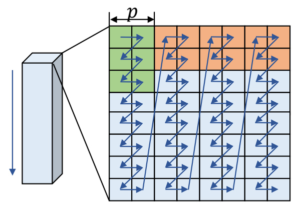


Figure 5: Waffle architecture overview.

Assume we have $m \times n$ cores (or cubes) in total. To reconstruct an object, we divide it into $m \times n$ parts as Figure 6a depicts. Each part and its related beam information are stored in a cube, and processed by the core in the cube in a voxel-driven manner.



(a) Object partition.



(b) The order of storing voxels in each cube.

Figure 6: Partition the reconstruction object for each cube, and store it in a snakelike manner.

5.2 Data Structure and Layout

Each cube needs to store the information of voxels assigned to it, and the beams that pass through these voxels. The data structures for each voxel and beam are shown in Figure 5.

For each voxel, besides its basic features f and v, some information of beams that penetrate it is stored. n_b indicates how many beams pass through this voxel. For each beam, its information $info_b$ includes three contents. id_b is the unique id of the beam. end_b indicates whether this is the last voxel processed in this core, among the voxels this beam passes through. L_b represents the intersection length of the beam and the voxel.

Here, the fields of voxel data structure except end_b have been introduced in previous sections. As for end_b, its purpose is explained as follows. Because the traversing order of voxels is fixed in the voxel-driven manner, we can know which voxel among all the voxels a beam penetrates is processed at last. We store this information in the end_b field. In forward projection, if end_b is true, all the related voxels of this beam in this core (cube) have been processed. Therefore, it is an indication that we may start communication with another core to get the integral forward projection result of this beam.

Now we give the storage layout of voxels in a cube. The voxels are stored in the order shown in Figure 6b, where p is the parallelism degree of Mumford-Shah Functional Solver, which will be introduced soon in Section 5.3. We also traverse the voxels in this order. The reason is that the main operation of both $\frac{\partial MS}{\partial f}$ and $\frac{\partial MS}{\partial v}$ is 2D second-order derivative. To conduct 2D second-order derivative, at least three rows of data are required. Therefore, if we use the traverse order in Figure 6b, the size of the green region is enough for the buffer. However, if we traverse the voxels row by row, the buffer size rapidly increases to the green region plus the orange region. Therefore, using the snakelike manner in Figure 6b can greatly reduce the buffer size demand.

For each beam, the data structure includes the following fields. Its unique id id_b and the attenuation degree att_b (ΣfL) are of course needed. Besides, $lastCore_b$ and $nextCore_b$ record the last/next core the beam passes through. With these two fields, we can know which core should be communicated within synchronization.

To introduce the last two fields $sync_b$ and $finish_b$, we first give the main idea of our synchronization here. Recall that the synchronization in Figure 4 is to deal with beams that go across multiple cores. Assume a beam passes $core_0, core_1, \cdots, core_n$, successively. For $core_0$, when it finishes its partial forward projection of this beam, it gives this partial result to $core_1$. For $core_m, 0 < m < n$, when it finishes its partial forward projection of this beam, and also receives the partial result from $core_{(m-1)}$, it sums them and deliver the new result to $core_{(m+1)}$. For $core_n$, when it finishes its partial forward projection of this beam, and also receives the partial result from $core_{(n-1)}$, it can sum up the total attenuation degree (ΣfL) of this beam. Then, it can calculate ($\Sigma fL - g$) of this beam, and deliver it to $core_{(n-1)}$. Then, each core can deliver it to the previous core successively. In this way, all the cores can obtain ($\Sigma fL - g$) of this beam. Such a synchronization method only requires communication between adjacent cores, and the related operation is only an add operation. It can also be conducted asynchronously with the voxel-driven forward projection, which is very efficient.

Now it is straightforward to introduce $sync_b$ and $finish_b$ of the beam data structure. $sync_b$ indicates whether the previous core has passed its partial result to this core. $finish_b$ indicates whether this core has finished its partial forward projection of this beam.

5.3 Dataflow and Logical Die Designs

5.3.1 Dataflow The logical die is responsible for the voxel-driven inner loop in Figure 4, and also responsible for the synchronization. The dataflow is divided into two stages which can reuse hardware resources. Details about these two stages are as follows.

The first stage conducts forward projection, and calculates $\frac{\partial MS}{\partial v}$ to update v. Because the voxel data structure is of variable length due to n_b, the *offset decoder & fetcher* is needed to decode the address offset of each voxel according to n_b, and fetch the data to subsequent components. Concretely, it fetches f and v for the *Mumford-Shah functional solver*, and fetches f and $info_b$ for the *PEs*.

Then, the *Mumford-Shah functional solver* calculates $\frac{\partial MS}{\partial v}$ according to Equation 1, and updates v. Each *PE* calculates fL_b, passing the beam id and fL_b to the *Exclusive Updater*, which will update att_b in the *beam buffer* with no potential conflict. If end_b is true, the *PE* will fetch the related beam from the *beam buffer*, change $finish_b$ to true, and check whether $sync_b$ is true. Both $finish_b$ and $sync_b$ equal to true means that the forward projection till this core has finished, thus the *PE* will notify the *communicator* for further communication.

The *communicator* waits for three cycles when notified by the *PE*. This is to avoid the possible hazard that the *Exclusive Updater* has not updated att_b in the *beam buffer*. Then, it realizes the synchronization introduced in Section 5.2. To be concrete, (1) if this is the last core the beam passes through, the *communicator* will calculate $att_b - g$ as the forward projection result. This result is still stored in the att_b field to save space, and pass it to the previous core. The *communicator* in the previous core will also update the forward projection result and delivers it further. (2) If this is not the last core the beam passes through, the *communicator* will deliver att_b to the next core. When the next core receives att_b, its *communicator* passes it to the *Exclusive Updater*, which will add it to the corresponded att_b in the beam buffer. Its *communicator* also sets $sync_b$ to be true, and checks whether $finish_b$ is true. If $finish_b$ is true, it will deliver att_b to its next core.

The second stage calculates $2(\Sigma fL - g)L$ and $\frac{\partial MS}{\partial f}$ to update f. This stage can reuse the hardware resources in the first stage. The *offset decoder & fetcher* can be directly reused to traversing the voxels. Because the main overhead of calculating both $\frac{\partial MS}{\partial v}$ and $\frac{\partial MS}{\partial f}$ is 2D first-order and second-order derivative, we can also reuse the *Mumford-Shah functional solver* to calculate $\frac{\partial MS}{\partial f}$. The *Exclusive Updater* is still responsible for exclusive updates to the voxel buffer. Moreover, the *PEs* can also be reused as follows.

In first stage, the *PEs* conduct multiplication for fL_b. In the second stage, they are used to calculate $2(\Sigma fL - g)L$. Recall that when the first stage finishes, the forward projection result $\Sigma fL - g$ has been stored in att_b domain of each beam. Therefore, in the second stage, the *PEs* actually conduct multiplication $2att_bL_b$. Therefore, these two stages both mainly conduct multiplication applications. The inputs are both fetched in a voxel-driven manner, and the outputs are both sent to the *Exclusive Updater*.

5.3.2 Mumford-Shah functional solver This unit is to calculate $\frac{\partial MS}{\partial v}$ and $\frac{\partial MS}{\partial f}$. Because we need three rows of data to calculate second-order derivative, we store and traverse the voxels as shown in Figure 6b to improve the data reuse rate, where p is the parallelism degree. In our experiments, p is $0.1\times$ of the number of *PEs* to keep their execution time balanced.

5.3.3 Exclusive updater This unit is to update the buffers without conflict. It receives a pair of an address and a value, and adds the value to the original value in the address exclusively. The realization of the exclusive mechanism is very simple and efficient. We maintain a small content-addressable memory (CAM) in the *Exclusive Updater*. All the update requests are buffered in the CAM. Each entry of the CAM and its state machine are shown in Figure 7. When a new request arrives, if its address matches one request

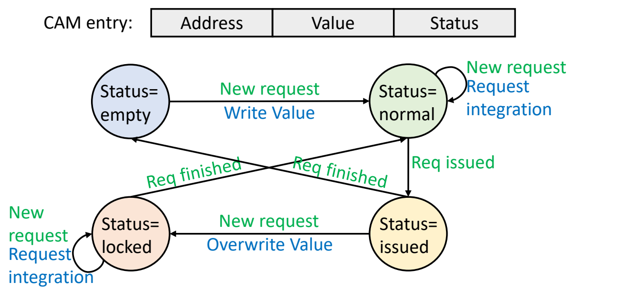


Figure 7: The fields of a CAM entry in the Exclusive Updater, and the state machine of an entry.

Table 1: Waffle parameters.

Cube parameters (32nm)			
Capacity	8.0GB	Area	$226mm^2$
Bandwidth	320GB/s	Power	0.029W/(GB/s)
Core parameters (32nm)			
Frequency	600MHz	Area	$23mm^2$
Voxel Buffer	256KB	Beam Buffer	1MB
PEs	2048	Power	5.6W

in the CAM, its value is added to the value of the matched one. We call this operation "request integration". When the *Exclusive Updater* issues one request *req*, it marks the related entry in the CAM as "issued". New request matching an "issued" entry cannot do the request integration, but writes its value into the entry instead, and changes the status of the entry from "issued" to "locked". A "locked" entry supports request integration, but cannot be issued. Only when *req* finishes, this entry becomes a "normal" entry.

6 Evaluation

6.1 Experiment Setup

Since there is no previous work using ASIC for 3D inner reconstruction, we compare Waffle with two previous works on FPGA [11] and multi-GPU [12] platforms, respectively. We choose the CT reconstruction task as the benchmark. We realize the FPGA-based accelerator on Xilinx ZCU102 FPGAs. The multi-GPU-based accelerator uses 10 GPUs of 1080Ti. For Waffle, we simulate it with configurations shown in Table 1. The memory parameters are extracted from previous works [1, 3, 8]. The logic core is synthesized using Synopsys Design Compiler. Its power density satisfies the thermal constraint [4].

The way we choose the parameters for Waffle is as follows. Before core design, we first select memory parameters from industrial SPEC [3] and previous works [1, 8]. Then, the core is carefully configured so that its throughput can match the memory bandwidth. The voxel and beam buffer sizes are chosen empirically in our experiments to cover the working set, i.e., when buffer size increases, the hit ratio cannot further increase.

We use three configurations of Waffle in our evaluation to show its scalability: (1) a package with 2×2 cubes, (2) a package with 4×4 cubes, and (3) two packages, each with 4×4 cubes. For numerical simulation, we use a modified FORBILD head phantom [10]. For the CT system, we use the same parameters as the previous work [12].

6.2 Reconstruction Quality and Convergence Speed

Peak signal-noise ratio (PSNR) is the most commonly used metric for the quality of 3D inner reconstruction. To compare the reconstruction quality, and the speed for convergence to a stable

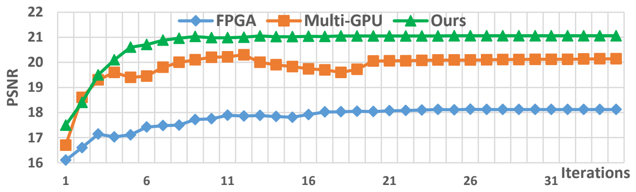
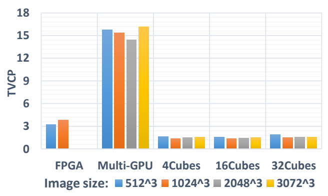


Figure 8: Convergence iteration comparison.



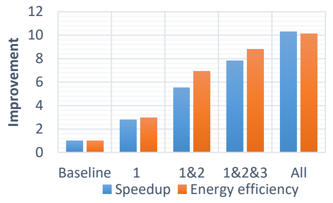


Figure 9: Time to reconstruct a Voxel under normalized Computation Power (TVCP).

Figure 10: Breakdown of the improvements over multi-GPU baseline.

PSNR, we test how PSNR changes with more iterations, as Figure 8 shows. Here we use 720 views, 1000 photons, and the image size is 1024×1024. We can see that, Waffle converges to a better PNSR with fewer iterations. This is because both previous works cannot conduct global synchronization. The FPGA work gives up synchronization and adopts an asynchronous method. The multi-GPU work divides the task and conducts local gradient descent.

6.3 Reconstruction Time and Scalability

Table 2 shows the time for the three works to converge. The asynchronized FPGA accelerator has many hyper-parameters to tune, and we fail to make it converge on large images.

Note that the computation power of the Xilinx ZCU102 FPGA (200MHz [11]), the 10-GPU cluster, and one cube of Waffle can ideally achieve 0.71TOPs/s, 73TFLOPs/s and 1.6TOPs/s, respectively. To compare the results more clearly, we design a new metric called TVCP, which means Time to reconstruct a Voxel under normalized Computation Power. Results are shown in Figure 9.

We can see that, when normalized to the same computation power, Waffle is $1.70\times \sim 2.82\times$ faster than the FPGA solution, and $8.26\times \sim 11.29\times$ faster than the multi-GPU solution. The FPGA performance is okay on small images, because the harm on the convergence of their asynchronous method is insignificant on small images. But their method fails when image size increases. Why Waffle is much faster than the multi-GPU work will be analysed in Section 6.5.

Besides, as shown in Figure 9, TVCP of Waffle changes little when the number of cubes or image size scales up. This indicates the great scalability of Waffle.

6.4 Energy Efficiency

Table 3 shows the energy for reconstruction. Waffle (16Cubes) has $8.24\times \sim 10.67\times$ energy efficiency of the FPGA solution, and $9.35\times \sim 10.97\times$ of the multi-GPU one. The breakdown of improvement sources is shown in Section 6.5.

6.5 Breakdown of Improvements

In this subsection, we present how much performance gain can be obtained from each of our optimizations. We divide our optimizations into four categories: (1) our voxel-driven algorithm and

Table 2: Reconstruction time comparison (unit: second).

Image Size	512^3	1024^3	2048^3	3072^3
FPGA	612	5811	not converge	
Multi-GPU	29	226	1701	6427
Ours(4Cubes)	35	235	2027	7213
Ours(16Cubes)	8	57	485	1662
Ours(32Cubes)	5	31	263	867

Table 3: Reconstruction energy comparison (unit:kJ).

Image Size	512^3	1024^3	2048^3	3072^3
FPGA	70	672	not converge	
Multi-GPU	86	691	5098	19287
Ours(4Cubes)	8.8	67	615	2265
Ours(16Cubes)	8.5	63	545	1906
Ours(32Cubes)	9.3	76	678	2521

the specialized hardware support for it; (2) the data structure design (Especially introducing $info_b$ into the voxel data structure); (3) the optimized dataflow considering the real data dependency, and the related hardware designs such as the exclusive updater; (4) some free-lunch offline optimizations, such as the layout and traverse order of the voxels.

We use 512^3 image size and 16-cube Waffle in this experiment. Baseline is the multi-GPU design [12] normalized to the same computation power. The improvements of both speed and energy efficiency are shown in Figure 10, which indicates that all our optimizations are significant to the performance of Waffle.

7 Conclusion

In this work, we formulate the problem and present a unified programming model for 3D inner reconstruction. Then, we redesign the algorithm to be hardware-friendly with many optimizations, such as the voxel-driven traverse, removal of unnecessary synchronizations, etc. We further propose Waffle, a near-memory-computing architecture for our optimized algorithm. Our software-hardware co-design work can fully utilize the advantages of near-memory-computing, and gains $8.24\times \sim 10.97\times$ energy efficiency over previous works. We hope this paper can introduce the 3D reconstruction problem to the community and inspire more future works.

Acknowledgments

This work is supported by National Natural Science Foundation of China (Grant No. 62032001) and 111 Project (B18001).

References

[1] J. Ahn, S. Hong, et al. 2015. A scalable processing-in-memory accelerator for parallel graph processing. In *ISCA*.
[2] Margherita Capriotti, Hyungsuk E Kim, et al. 2017. Non-Destructive inspection of impact damage in composite aircraft panels by ultrasonic guided waves and statistical processing. *Materials* (2017).
[3] Hybrid Memory Cube Consortium et al. 2014. Hybrid memory cube specification 2.1. *hybridmemorycube. org* (2014).
[4] Yasuko Eckert, Nuwan Jayasena, and Gabriel H Loh. 2014. Thermal feasibility of die-stacked processing in memory. (2014).
[5] Yu-Hang He, Ai-Xin Zhang, et al. 2020. Deep learning based high-resolution incoherent x-ray imaging with a single-pixel detector. In *ISA*.
[6] Weichang Li. 2018. Classifying geological structure elements from seismic images using deep learning. In *SEG Technical Program Expanded Abstracts 2018*.
[7] Chaoyue Liu, Libin Zhu, and Mikhail Belkin. 2020. Toward a theory of optimization for over-parameterized systems of non-linear equations: the lessons of deep learning. *arXiv preprint arXiv:2003.00307* (2020).
[8] Seth H Pugsley, Jeffrey Jestes, et al. 2014. NDC: Analyzing the impact of 3D-stacked memory+ logic devices on MapReduce workloads. In *ISPASS*.
[9] Yuki Takechi-Haraya, Kumiko Sakai-Kato, et al. 2016. Atomic force microscopic analysis of the effect of lipid composition on liposome membrane rigidity. *Langmuir* (2016).
[10] Zhicong Yu, Frédéric Noo, et al. 2012. Simulation tools for two-dimensional experiments in x-ray computed tomography using the FORBILD head phantom. *Physics in Medicine & Biology* (2012).
[11] Wentai Zhang, Linjun Qiao, et al. 2020. FPGA Acceleration for 3D Low-Dose Tomographic Reconstruction. *TCAD* (2020).
[12] Yining Zhu, Qian Wang, et al. 2019. Image reconstruction by Mumford–Shah regularization for low-dose CT with multi-GPU acceleration. (2019).