Arbitrarily large tomography with iterative algorithms on multiple GPUs using the TIGRE toolbox

Ander Biguri Reuben Lindroos Robert Bryll Hossein Towsyfyan
Hans Deyhle Ibrahim El khalil Harrane Richard Boardman
Mark Mavrogordato Manjit Dosanjh Steven Hancock
Thomas Blumensath

July 16, 2020

Abstract

3D tomographic imaging requires the computation of a solutions to very large inverse problems. In many applications, iterative algorithms provide superior results, however, memory limits in available computing hardware restrict the size of problems that can be solved. For this reason, iterative methods are not normally used to reconstruct typical data sets acquired with lab based CT systems. We thus use state of the art techniques such as dual buffering to develop an efficient strategy to compute the required operations for iterative reconstruction. This allows the iterative reconstruction of volumetric images of arbitrary size using any number of GPUs, each with arbitrarily small memory. Strategies for both the forward and backprojection operators are presented, along with two regularization approaches that are easily generalized to other projection types or regularisers. The proposed improvement also accelerates reconstruction of smaller images on single or multiple GPU systems, providing faster code for time-critical applications. The resulting algorithm has been added to the TIGRE toolbox, a repository for iterative reconstruction algorithms for general CT, but this memory-saving and problem-splitting strategy can be easily adapted for use with other GPU-based tomographic reconstruction code.

1 Introduction

Modern computed tomographic imaging (CT) systems can produce very large data-sets. For example, X-ray computed tomography systems that can cover a detector area of 2 m $\times$ 3 m are already available commercially, with detector pixel sizes in the order of 0.2 mm, resulting in very high resolution projections and consequently very high resolution volumetric images. To reconstruct high resolution three dimensional volumetric images from this data remains challenging. Whilst efficient filtered backprojection methods work well with high quality data, many applications increasingly require more advanced, iterative reconstruction methods. Typical computing hardware is however currently unable to efficiently compute iterative solutions in reasonable timeframes. The current limiting factor for these problems is the memory size available on Graphic Processing Units (GPUs), which are typically used to accelerate computation. Furthermore, industrial and medical applications of CT are also pushing for faster reconstruction. CT for quality control and metrology of manufactured components requires fast scan times as increased throughput of samples is desired [1, 2]. In medical applications, especially with adaptive and image guided radiation therapy, short scan and reconstruction times are fundamental for the correct targeting of tumours [3, 4].

Particularly in the medical field, but also in other applications, iterative algorithms are becoming more common, as they can provide higher quality images in low data quality and quantity scenarios, reducing scan times and radiation dose [5, 6, 7]. However, iterative algorithms are significantly slower than standard filtered backprojection based reconstruction methods.

Mathematically speaking, iterative algorithms solve the linearized model

$$Ax = b + \tilde{e} \tag{1}$$

where $x \in \mathbb{R}^{N_e}$ is a vector of image elements, $b \in \mathbb{R}^{N_p}$ a vector of absorption measurements and A a matrix that describes intersection lengths between rays and elements. There are multiple approaches in the literature to solve equation 1 using iterative methods, however all of them have one aspect in common: at least one of each, Ax (forward projection) and $A^T b$ (backprojection) operations are required in each iteration.

For small problems, where the system matrix A can be pre-computed and stored, iterative reconstruction can be easily parallelized on multi-GPUs using sparse matrix vector (SpMV) multiplication techniques [8, 9, 10]. As these methods rely on storage of the system matrix, they are unsuitable for tomographic reconstruction of volumetric images that are typically computed from lab-based systems. Even in the most recent work on the use of SpMV for tomographic reconstruction [11] approximately 1.5GB of memory is required for sparse matrix storage for each 1024^2 slice of the reconstruction. This is over a terabyte of memory for a "standard small" lab-based image of 1024^3 voxels. Achieving this memory efficiency furthermore imposes significant symmetry assumptions on the acquisition process which is not always applicable in lab-based systems.

For large volumetric tomographic reconstruction, the system matrix in the inverse problem is too large and so is never stored. Instead, methods compute matrix vector products involving the system matrix (the so called projection and backprojection operations) using ray tracing type methods, which can be done efficiently using GPUs. There is a large literature on parallel computation for large tomographic problems, especially for the Filtered Backprojection (FBP) algorithm and its cone-beam corrected modification, the Feldkamp Davis Kress (FDK) algorithm. For parallel beam systems, the problem can be easily split into independant 2D probelms which can be solved in parallel on different GPUs or High Performance Computers (HPCs) [12, 13, 14, 15]. For cone beam tomography this no longer works. CPU-based methods are the most common approach taken for parallelization in this case, as they require no change to the reconstruction code. This works as long as the code is executed with a sufficiently large CPU-based computer cluster, such as an HPC cluster [16, 17, 18]. However, GPUs have significant advantages for CT reconstruction and several single and multi-GPU approaches have also been proposed [19, 20, 21], often using a message passing protocol (MPI) [22] to allow execution in large clusters of nodes with a few GPUs each. This method is currently the best approach for large volumetric CT reconstruction as it can be highly efficient computationally if the partitions of the image are optimized to minimize memory transfer [23, 24].

Most multi-GPU tomographic reconstruction code parallelizes the computation of the projection and backprojection operators. With this approach, no change to the iterative reconstruction algorithms is needed. An alternative approach would be to split the problem mathematically instead [25, 26, 27], proposing different, distributed optimization methods that use multiple locally converging independent computations that ensure global convergence of the reconstruction. Each of these threads can be executed in a separate machine with limited memory transfer between nodes. These methods can be executed in arbitrarily large computer networks, but use different algorithms to solve the reconstruction problem.

Current literature and open source software have significant limitations when reconstructing large images on multiple GPUs: hitherto, the entire set of projections and images had to fit inside the GPU's internal RAM. To reconstruct an image volume with 2000^3 voxels requires around 8 of the highest memory GPUs, whilst for 4000^3 voxel volumes over 50 GPUs are required, depending on the projection size. Access to a computer with these resources is not common and would be extremely expensive. Most industrial, scientific and medical centres do not have access to such multi-GPU HPC clusters and instead have dedicated single-node machines for CT data processing, with a few GPUs each (with numbers ranging from 2 to 8 GPUs at best). Consequently, recent GPU based implementations have tried to overcome these limitations. For example, recently published GPU implementations of the FDK algorithm can overcome these issues by dividing the image into separate pieces and treating each piece independently [28, 29]. The Python version of the ASTRA toolbox [19] now also provides functionality to reconstruct large images, however, this is achieved by simply splitting the reconstruction volume into smaller pieces and by then calling dedicated GPU-based forward and backwards operators from CPU, where the results are combined. This method allows the the reconstruction of problems whose size is independent from the available GPU memory, but the approach is inefficient. We propose a more efficient method to compute arbitrarily large iterative reconstructions on arbitrarily small GPUs.

We here propose a method for large scale tomography that

- splits the forward and backprojection operators into multiple pieces that can be computed in parallel or in series on a single-node multi-GPU machine;
- uses a splitting method that reduces GPU memory consumption and is faster thanks to the introduction of memory management approaches;
- allows the reconstruction of images whose size is limited only by CPU RAM (as opposed to the common GPU RAM limitation);
- is provided freely under a permissive license (BSD 3-clause).

The proposed improvements are achieved by using recent advances in memory handling in GPU technology.

The rest of this paper is structured as follows. Firstly, in the methods section, an overview of TIGRE and iterative algorithms is given, together with a review of GPU terminology. The section continues to explain the proposed approach for the forward and backprojection operations and finishes with a description of how to accelerate regularization operations that are implemented on GPUs. We then present results on execution times for different systems with varying image and detector sizes, together with two examples reconstructed with measured data.

2 Methods

The strategies for efficient multi-GPU tomographic reconstruction are primarily based on rearranging computation and memory transfer operations in such a way that the amount of computation is highly reduced while the overlap between memory transfer and computation execution is increased.

The Tomographic Iterative GPU-based Reconstruction toolbox (TIGRE) is a MATLAB and Python based toolbox for GPU accelerated tomographic reconstruction [30]. It is especially focused on iterative algorithms and provides implementations of more than ten of them, together with the FBP and FDK algorithms.

The forward projection and backprojection are the computationally heavy operations that take the majority of the execution time. Some algorithms also contain other computationally heavy operations, such as regularization operations. These operations are implemented in TIGRE using GPU accelerated code for NVIDIA devices using the CUDA language. These core operations are the ones needing adaptation if arbitrarily large images are to be reconstructed. Inefficient strategies in splitting and gathering of the partial results can lead to unnecessary computations and memory transfers. Proposing a splitting strategy for the general case is the goal of this article.

TIGRE's architecture is modular, thus all of the GPU code is independent from the algorithm that uses it. Thus, by adapting the GPU code to be able to handle arbitrarily large reconstruction, TIGRE will also automatically handle such images. The modular design however has a drawback: To allow the higher level abstraction required for the design of new algorithms by turning the computationally heavy operations into "black box" functions, sacrifices are needed in memory management and algorithm specific optimizations. Specifically, TIGRE will move memory from CPU to GPU and vice-versa in each GPU call. This allows for fast prototyping of new methods, but means algorithms may perform slightly slower than in fully optimized GPU code. The architecture also brings another limitation, especially for multi-GPU code: as memory is allocated and deallocated in the higher level languages (MATLAB and Python), it is stored in pageable memory. In brief, this is memory managed by the operating system (OS) that may be split, moved around, or not fully stored in RAM if the OS deems it necessary. An alternative would be page-locked or pinned memory,

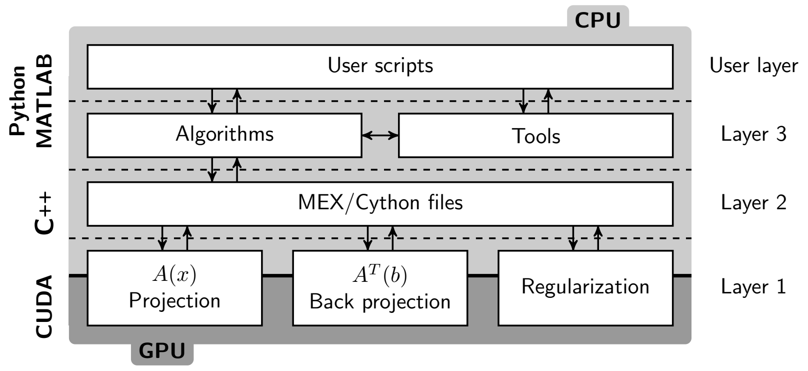


Figure 1: Modular architecture of the TIGRE toolbox.

memory that as its name suggest, is locked into a certain RAM location. This type of memory has not only faster transfer speeds (as it can be managed by the GPU without CPU intervention), but also allows for an important feature in speeding up code, asynchronous memory transfers.

CPU calls to GPU instructions can either be synchronous or asynchronous. If the call is synchronous, the CPU waits until the completion of the instruction before continuing, while if asynchronous, the CPU will keep executing further instructions, regardless if the GPU has terminated its task or not. Queueing operations but ensuring none of them are executed before its requirements are met (e.g. some computation finished, specific memory transfers ended) is an important feature when improving total computational time on multi-GPU systems, as it allows queuing simultaneous memory reads or writes and simultaneous memory management and computation execution. As TIGRE relies on Python and MATLAB to allocate this memory, memory will initially be paged, thus only transferable synchronously, unless explicitly pinned beforehand, which is a costly operation. While the aforementioned limitations will cause the code presented here to be slightly less efficient, discussion on how to implement the multi-GPU strategy when the limitations imposed by TIGRE are not present is also given.

Before introducing the acceleration strategies used, it is worth mentioning that this work focuses on how to organize computation and memory transfer, but does not describe (nor constrain) the kernels (computational functions) themselves. There are several algorithms available in the literature [31, 32, 33, 34] to implement forward and back-projectors efficiently on GPUs and fair comparison between codes is a challenging task, as code is often not available and execution times are hardware specific. We do not claim that the kernels used in this work are the fastest, but our multi-GPU strategy presented is applicable to most, if not all, the algorithms for forward and backprojections in the literature.

2.1 Forward projection

The forward projection operation, Ax, simulates ideal X-ray attenuation with a given image estimate, x. There are several published algorithms to approximate the operation, two of which are implemented in TIGRE: interpolated projection [31] and ray-driven intersection projection with Siddon's method [35]. While the integral over the X-ray paths is computed differently in each methods, the kernels are organized and executed using the same structure.

The kernels are queued and executed on GPUs using 3D blocks of threads of size $N_u \times N_v \times N_{angles}$[1], where each thread processes the integral of an individual detector pixel, as seen in Figure 2. By using the neighbouring memory caching capabilities of the texture in the GPU, this kernel structure allows for faster executions due to higher memory read latency. This is because memory

[1]For NVIDIA GTX 10XX GPUs, values of $N_u = 9$, $N_v = 9$, $N_{angles} = 9$ have been found to be the fastest empirically. Different values do not alter the algorithms proposed in this work.

reads needs hundreds of cycles to complete, but storing the image in texture memory allows caching in 3D. Therefore concurrent threads reading in the same or neighbouring elements cause an increased cache hit rate, directly translated into lower memory read time and faster execution. On top of the faster memory reads, texture memory also allows for hardware implemented trilinear interpolation. For more information on the projection algorithm, the reader is reffered to the technical note by Palenstijn *et al* [36]. Each time the forward projection kernel is launched enough thread-blocks to cover the entire height and width of the projection are queued, but only one in depth. Thus, each kernel launch will compute N_{angles} entire projections.

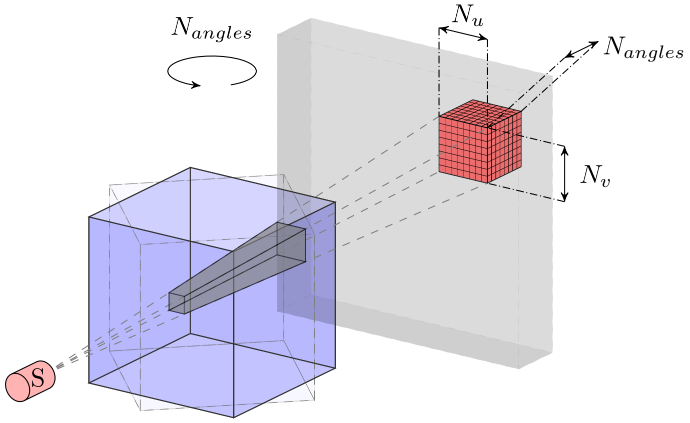


Figure 2: The forward projection block execution kernel structure. The detector is divided in $N_u \times N_v \times N_{angles}$ sized blocks to maximize cache hit rates. Each kernel launch executes enough blocks to fully compute N_{angles} entire projections.

To execute the projection operation in cone beam geometries (such as circular, C-arm or helical scans, among others) however, the entire image is required, as due to the shape of the X-rays different image partitions will have influence in overlapping areas in the detector. Thus if the image does not fit in the GPU, projections will need to be separately computed for each image partition and later accumulated. The accumulation procedure is ultra-fast, executing in approximately 0.01% of the time that a projection kernel launch needs to finish.

A common approach for a projection operation on multiple GPUs is to allocate the memory of the projections along with the image size, then execute the kernels. Then a gathering step is performed to accumulate the overlapping detector pieces. This is a good approach if the image and projections fit in the available GPU memory, but for large images this is not the case thus the variables needs to be partially stored in the host RAM. Once this becomes necessary, then it is important to reduce the amount of unnecessary memory on the GPUs, in order to increase the size of each individual image partition (i.e. reduce the amount of splits to the image).

The approach we propose can be summarised as follows: Only the memory for two projection kernel launches will be allocated in each device ($2 \times N_{angles}$ individual projections) and the rest of the GPU memory will be free for the image. These two projection pieces will act as a buffer to store the computed projections while they are being copied out to the CPU during the kernel execution. As one piece is being used to store the projections that are being computed at that moment, the other will be copying memory to the CPU RAM. This not only allows us to reduce the number of splits needed, it also reduces execution time, as no extra time is needed to copy the results out of the GPU. This technique is called double-buffering and it is common in several GPU applications [37, 38]. On multi-GPU systems, each GPU will compute a set of independent projections, using the entire image. If the image does not fit entirely in each of the GPUs, the image is partitioned into equal size volumetric axial slices stacks, as big as possible. Additionally, an extra projection buffer will be allocated with the same size. These will be used after the first image slice has been forward projected, as they will load previously computed partial set of projection data into GPU RAM, to be accumulated on the GPU when each of the projection kernels are finished.

The instruction queueing algorithm is as follows. Once the image slice is already in the GPU, first the kernel launch for the projection is executed asynchronously. Then, if needed, the partial projections from previously computed image slices are copied from CPU to each of the GPUs. After the completion of the copy, the CPU queues asynchronously the accumulation kernel and finally copies the result from previous computation to CPU. Each of these individual instructions is queued in all available GPUs simultaneously. A timeline of the process can be seen in figure 3 and Algorithm 1 shows the pseudocode.

In the one and two GPU case, empirical tests show that if the image needs to be partitioned due to lack of memory, page-locking the image memory in the beginning of the process results in a faster overall execution. This is true even with the significant time it requires to page-lock, thanks to faster CPU-GPU transfer speeds (from approximately 4GB/s to 12GB/s on a PCI-e Gen3). On more than two GPUs, page locking always brings an improvement, due to the simultaneous copy that it enables.

Algorithm 1 Projection operation kernel launch procedure

```
 1: Check GPU memory and properties
 2: Split projections among GPUs
 3: if Image does not fit on GPU then
 4:     Page-lock image memory                                  ▷ If the device allows it
 5: end if
 6: Initialize required synchronous operations (texture, buffers, auxiliary variables)
 7: for $N_{sp}$ image splits do
 8:     Copy current image split CPU→GPUs                        ▷ Asynchronous
 9:     Synchronize()
10:     for (Total angles)/$N_{GPU}$/$N_{angles}$ amount of kernel calls do
11:         ForwardProject<<<Launch>>>                           ▷ Asynchronous
12:         if Not first image split then
13:             Copy already computed partial projections CPU→GPUs   ▷ Synchronous
14:             Synchronize(Memory)                  ▷ Wait until memory copy completion
15:             AccumulateProjections<<<Launch>>>                ▷ Asynchronous
16:         end if
17:         if Not first projection then
18:             Copy previous kernel projections GPUs→CPU        ▷ Synchronous
19:         end if
20:         Synchronize(Compute)
21:     end for
22:     Copy last kernel projections GPUs→CPU                    ▷ Synchronous
23:     Synchronize()
24: end for
25: Free GPU resources
```

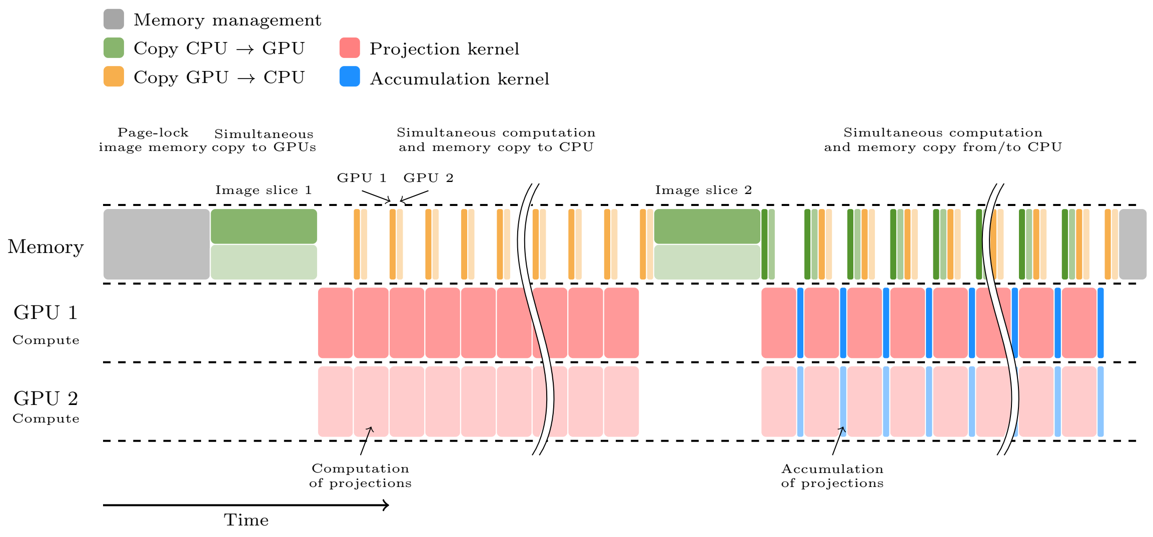

Memory management
Copy CPU → GPU
Copy GPU → CPU
Projection kernel
Accumulation kernel
Page-lock image memory
Simultaneous copy to GPUs
Simultaneous computation and memory copy to CPU
Simultaneous computation and memory copy from/to CPU
Image slice 1
GPU 1
GPU 2
Image slice 2
Memory
GPU 1 Compute
GPU 2 Compute
Computation of projections
Accumulation of projections
Time

Figure 3: Forward projection timeline for 2 GPUs and an image that requires 2 splits per GPU (not to scale). Each GPU handles half of the projections required. Using buffers, projection results are asynchronously copied to CPU while the next kernel launch is being computed. On image partitions other than the first, copies of the partial projections are copied while the current set is being computed for accumulation.

2.2 Backprojection

The backprojection operation, $A^T b$, smears detector values, b along the X-ray paths towards the source. TIGRE uses a voxel-based backprojection, with two optional weights, FDK weights and "pseudo-matched" weights that approximate the adjoint of the ray-driven intersection projector [39]. Both backprojectors execute $N_x \times N_y \times N_{angles}$ block of threads, where each thread computes N_z voxel updates[2], as described in detail in articles [40] and [41] and shown in figure 4. Similar to the projection operators, this execution order increases occupancy of the processors and optimizes cache hit rates, reducing the overall computational times. Therefore the projections need to be stored in texture memory to enable 3D caching.

Similarly to the projection, the common approach for back-projecting in the literature requires the allocation of the entire set of projections on GPU RAM before or during the computation of the backprojection voxel updates. A more efficient approach is possible.

The proposed algorithm for multi-GPU splitting is presented in algorithm 2. Not unlike in the projection operation, two buffers of size N_{angles} are allocated in each GPU for the projections. Before computation starts, the image is split into equal sized volumetric axial slices stacks and allocated among GPUs. If the total image (plus buffers) does not fit in the total amount of GPU RAM among devices a queue of image pieces is added and each GPU is thus responsible of back-projecting more than one image slice. Each GPU uses all the projections, but performs the CPU-GPU memory transfer simultaneously with the voxel update computation. Unless the image size is considerably smaller than the detector size (which does not happen in real datasets) the memory transfer should complete sufficiently fast. The execution timeline can be seen in figure 5. Empirical tests show that page-locking the image memory is faster than using pageable memory when a single GPU needs to compute multiple image pieces.

[2]In TIGRE $N_x = 16$, $N_y = 32$, $N_{angles} = 32$ and $N_z = 8$. Different values do not alter the algorithms proposed in this work.

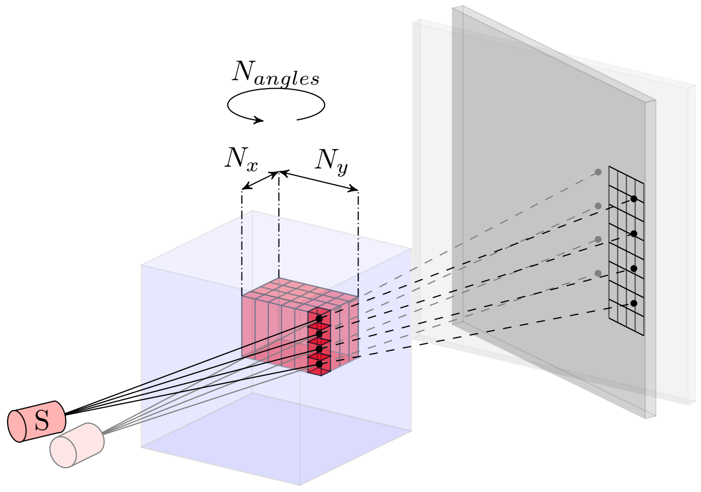


Figure 4: Backprojection kernel structure for a single thread-block. Each kernel launch executes a set of blocks covering N_{angles} projections using the entire image.

Algorithm 2 Backprojection operation kernel launch procedure

```
 1: Check GPU memory and properties
 2: Split projections among GPUs
 3: **if** Image does not fit on GPU **then**
 4:     Page-lock image memory                              ▷ If the device allows it
 5: **end if**
 6: Initialize required synchronous operations (texture, buffers, auxiliary variables)
 7: **for** $N_{sp}$ image splits **do**
 8:     **for** (Total angles)/$N_{angles}$ amount of kernel calls **do**
 9:         Copy projection set CPU→GPUs
10:         **Synchronize**()
11:         Backproject<<<**Launch**>>>                     ▷ Asynchronous, it gets queued
12:     **end for**
13:     Copy image piece GPUs→CPU
14: **end for**
15: Free GPU resources
```

2.3 Regularization

Tomographic reconstruction algorithms may include prior information to enforce a user-defined constraint. While a variety of regularizers are available in the literature [42, 43, 44], the total variation (TV) regularization is arguably the most common in CT algorithms [45, 46]. Therefore we focus here on TV type constraints. The logic used to speed up and split the problems however can be applied to other forms of neighbourhood based regularizers.

TV regularization adds an additional constraint to the image reconstruction problem: sparseness of the gradient, enforcing piecewise flat images. Two distinct formats of minimizing the TV constraint are used in TIGRE that appear in different algorithms in the literature: The gradient descend minimization approach and the Rudin-Osher-Fatemi (ROF) model minimization approach [47, 48]. Both of the codes use different minimization algorithms to reduce the total variation, but have two things in common. First, both methods take an image volume as input and return an image volume of the same size, i.e. the algorithms are not specific to CT. Secondly, similar to some other regularizers, they require knowledge of adjacent voxels to update each voxel value, thus are coupled operations. This makes them considerably harder to parallelise on multiple devices, as

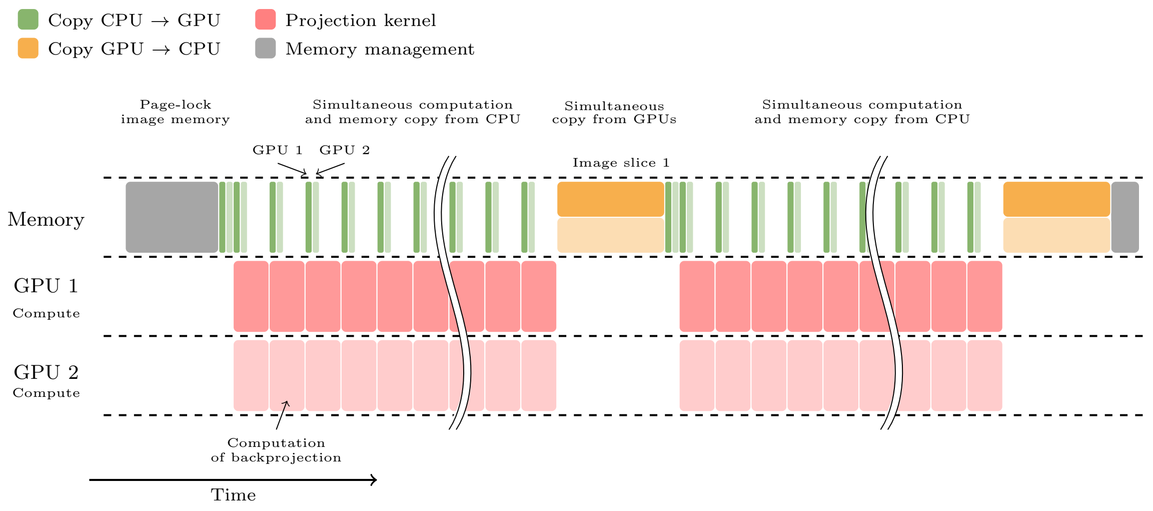

Copy CPU → GPU
Projection kernel
Copy GPU → CPU
Memory management
Page-lock image memory
Simultaneous computation and memory copy from CPU
Simultaneous copy from GPUs
Simultaneous computation and memory copy from CPU
GPU 1
GPU 2
Image slice 1
Memory
GPU 1 Compute
GPU 2 Compute
Computation of backprojection
Time

Figure 5: Backprojection timeline for 2 GPUs and an image size that requires 2 splits per GPU (not to scale). Each GPU will update each corresponding image slice using the entire set of projections. These are copied to GPU while the backprojection is being computed to reduce transfer times.

synchronization of the voxel values is required in each iteration, requiring communication between GPUs. Both methods often need tens or hundreds of iterations to converge.

While optimization of each of these regularizers is algorithm specific, the splitting method is the same as described in figure 6. An interesting property of this type of single voxel neighbouring is that if an overlapping buffer is created on the boundaries of the image containing voxel values of adjacent image splits, the depth of the buffer is equal to the amount of independent iterations that image piece can perform. This means that one can run stacks of N_{in} amount of independent iterations of the minimization of the regularization function to then update the buffer values with the newly computed voxel values on other GPUs. However, the depth on the buffer directly translates to higher computational demand, as each GPU will need to repeat operations in the same slices. In our work, a depth value of $N_{in} = 60$ in each split boundary has been found to have the best balance between reducing memory transfer steps and computational time. Additionally, the algorithms may need a norm in each iteration. If an operation where the entire image is required is present, algorithm specific decisions need to be made. In the TV case, the norm of the image update is required in each iteration. Empirical measurements suggest that not synchronizing in each iteration and approximating the norm assuming uniform distribution along the image samples has negligible effect in the convergence and result of the algorithm.

This approach however suffers heavily whenever the image and auxiliary variables do not fit entirely on the total amount of GPU RAM across devices. As quite often the regularization optimizers need several copies of the input image (e.g. the ROF minimizer in TIGRE requires 5 copies) the upper bound of the GPU RAM is easily reached. If this is the case, image splits need to be continuously read and written into CPU RAM, which heavily hinders overall performance. To increase the data transfer speeds in these cases the memory is allocated and pinned in the CPU RAM. In the case where the copies fit in GPU memory, the buffers are also allocated and pinned in the host, but are of much smaller size.

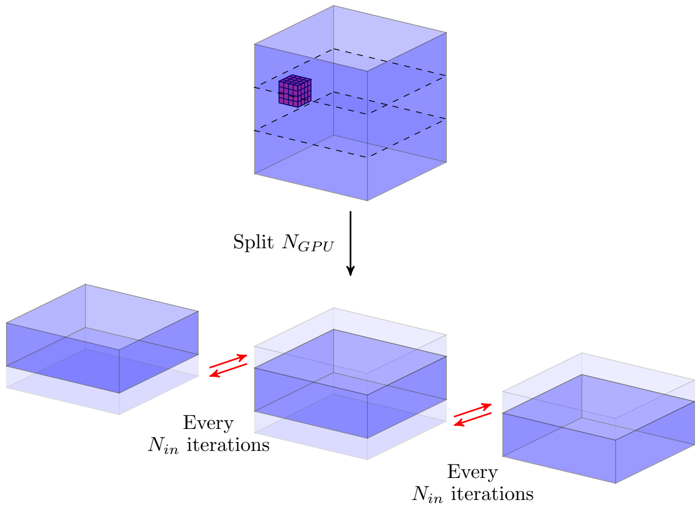


Figure 6: Regularization kernel diagram and splitting procedure for $N_{GPU} = 3$. Each kernel launch updates the entire image with multiple blocks. Each split contains N_{in} deep buffers in its boundaries, allowing for N_{in} independent inner iterations, before synchronizing the buffers.

3 Results

Two experiments are conducted to evaluate the proposed multi-GPU code. First, a detailed experiment is conducted on the projection and backprojection operations themselves, using varying problem sizes and numbers of GPUs. Analysis of the speed-up and performance achieved in the different situations is given. Second, a comparison of the presented method against code that parallelizes from CPU is given. Lastly, two large scans at different scales and from different X-ray machines are reconstructed using the multi-GPU code and the TIGRE toolbox[3].

3.1 GPU call performance analysis

To test the performance of the multi-GPU code, the CUDA code is executed with varying image sizes on up to 4 GPUs. Two machines are used for the experiments with very similar specifications. For the 1 and 2 GPU nodes, a workstation with an AMD EPYC 7551P processor with 256GB of RAM and 2 NVIDIA GTX 1080Ti with 11GiB of RAM each is used, connected with independent PCI-e Gen3 x16 ports. For the 3 and 4 GPU tests, a GPU node on an Iridis 5 HPC cluster has been used, with 2 Intel Xeon E5 2680 processors, 128GB of RAM and 4 NVIDIA GTX 1080 Ti connected with independent PCE-e Gen3 x16 ports. Different machines were used simply due to the larger RAM on the 2 GPU workstation. While the processors are different, empirical measurements showed no significant time difference when executing the 1 and 2 GPU code in the HPC GPU node, thus results are comparable.

Figure 7 presents the total time (computational plus memory transfer) for different sizes (N) and number of GPUs. The experiments used N^3 image volumes with N^2 detector pixels and N projection angles each. Figure 8 shows the percentage of time each amount of GPUs required compared to the single-GPU code. TIGRE has two projection and backprojection types, but here we only time the default ray-voxel intersection algorithm for projection and FDK weighted backprojection. The interpolated projector is slower than the ray-voxel version, and is thus not used in TIGRE's algorithms. It was included in TIGRE for completeness. The backprojection with matched weights

[3]In the following section the memory sizes are reported sometimes in decimal and sometimes binary units. We have kept the same unit as the manufacturer of the particular hardware in order to avoid confusion by having, for example, 238.419GiB of CPU RAM or a GTX 1080Ti with 11.811GB of RAM.

is 10%-20% slower than the FDK weighted ones and it is only used when a matched backprojection is fundamentally required, for example when using the CGLS or FISTA [49] algorithms. Apart from the slower kernels, the multi-GPU implementation is the same and thus their performance results are not shown.

The results show the expected performance gains. Computationally, duplicating the amount of GPUs should halve the kernel time. This is mostly true, the only difference is due to the last computational block in each kernel having less than their corresponding value of N_{angles} and the extra computational time required for projection accumulation. The other reason for not reaching the theoretical speed increase is due to memory management and transfer times. At small problem sizes, the kernels are fast enough so that the time is dominated by GPU property checking and memory transfer. The computation kernels are small enough as to hide any improvement that multiple GPUs could provide. The special case of the backprojection parallelizing a size of $N = 128$ results in a slower execution, but the overall time is smaller than 20 milliseconds. As N gets larger, the computational cost of both operations increases, therefore the measured speedup ratios approach the theoretical ones (50%, 33% and 25% for 2,3 and 4 GPUs respectively). This can be seen more clearly for the forward projection. Adding more GPUs is generally expected to keep the ratio closer to the theoretical limit until there are enough GPUs with each having too little computational load compared to the memory transfer required, as happens with the smallest sizes on this experiment[4].

The backprojection doesn't scale as well as the projection for two main reasons. Firstly, it is faster. This means that there is a significantly larger influence of memory management on the overall execution time. Additionally, the backprojection requires pinning image memory which adds significant overhead. In the projection operation, the memory is reserved before execution, but the OS allocates it as it gets written.

As an example of the splitting procedure, for the size $N = 3072$, the single GPU node required 11 image partitions while the 2 GPU version required 6 partitions for the backprojection. The projection just needed 10 and 5 partitions each. The difference is due to the smaller value of N_{angles} that the projection uses.

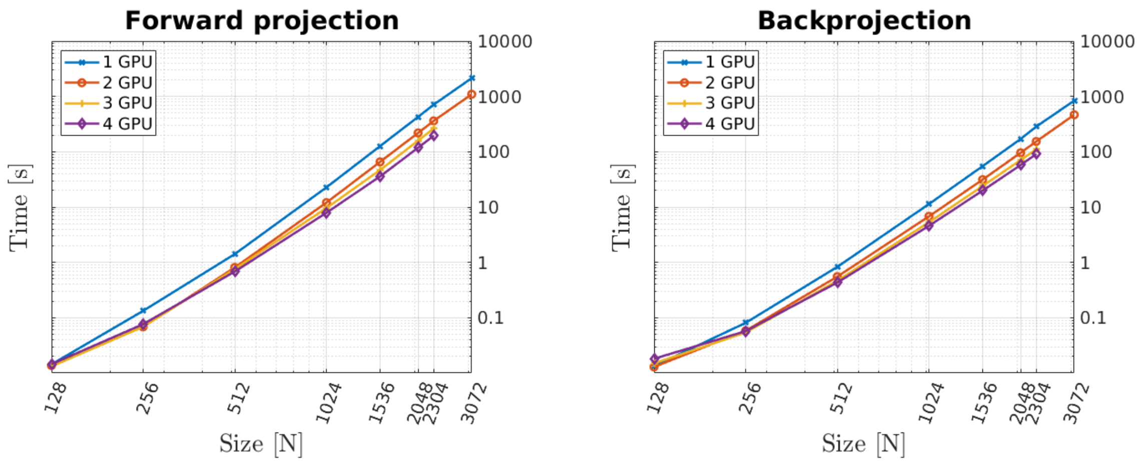
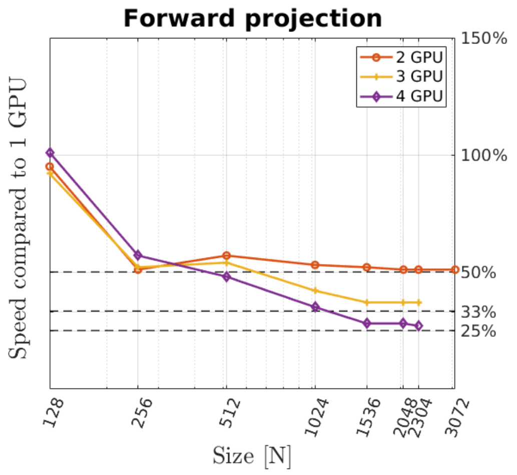


Figure 7: Projection and backprojection speeds at different sizes (N^3 voxel volumes with N^2 projections at N angles) for different numbers of GPUs. The time includes memory transfer speeds from and to the GPU. The missing points are due to lack of CPU RAM in the 4 GPU machine.

As this work proposes memory handling optimizations for GPU acceleration, measuring the impact that the memory has on the total execution of the code is of particular interest. Figure 9 shows the result for different image sizes and number of GPUs, for both the projection and backprojection. The operations are binned into three categories. Computing contains the time for

[4]Assuming a unique PCI-e Gen3 for each, otherwise the slower memory transfer time would impact the speed-up.



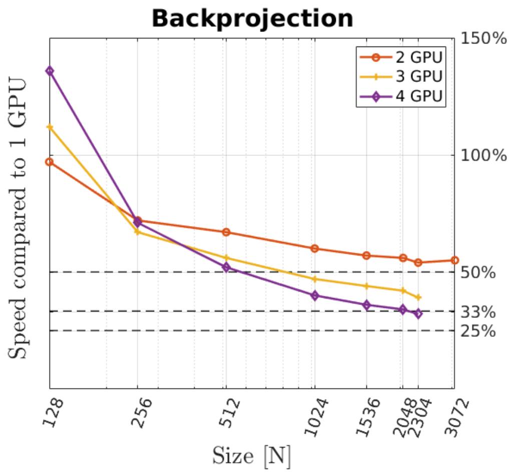


Figure 8: Projection and backprojection computational time percentages compared to the 1 GPU execution time at different sizes (N^3 voxel volumes with N^2 projections at N angles) for different numbers of GPUs. The time includes memory transfer speeds from and to the GPU. The missing points are due to lack of CPU RAM in the 4 GPU machine.

kernel launches, which includes simultaneous memory copies as they happen concurrently. Then the time to pin and unpin the CPU memory is added. In the backprojection operation, this time is a bigger part of the total because it forces the CPU to allocate the memory, while in the forward projection the memory is already allocated, as the image volume already exists. Some problem sizes do not have this section due to page-locking being a slow operation and thus not being performed as it brings less improvement than its cost. Finally, there are other memory operations. These include the time when the image is being copied in or out of the GPU and no computations are happening. It also includes memory freeing and other minor memory operations.

This analysis also helps explain better Figure 9. One can see that in the forward projection the computing time dominates most of the execution times even for very small images, while in the backprojection, even at 512^3 voxel volumes, the computation takes less than half of the time with more than 1 GPU.

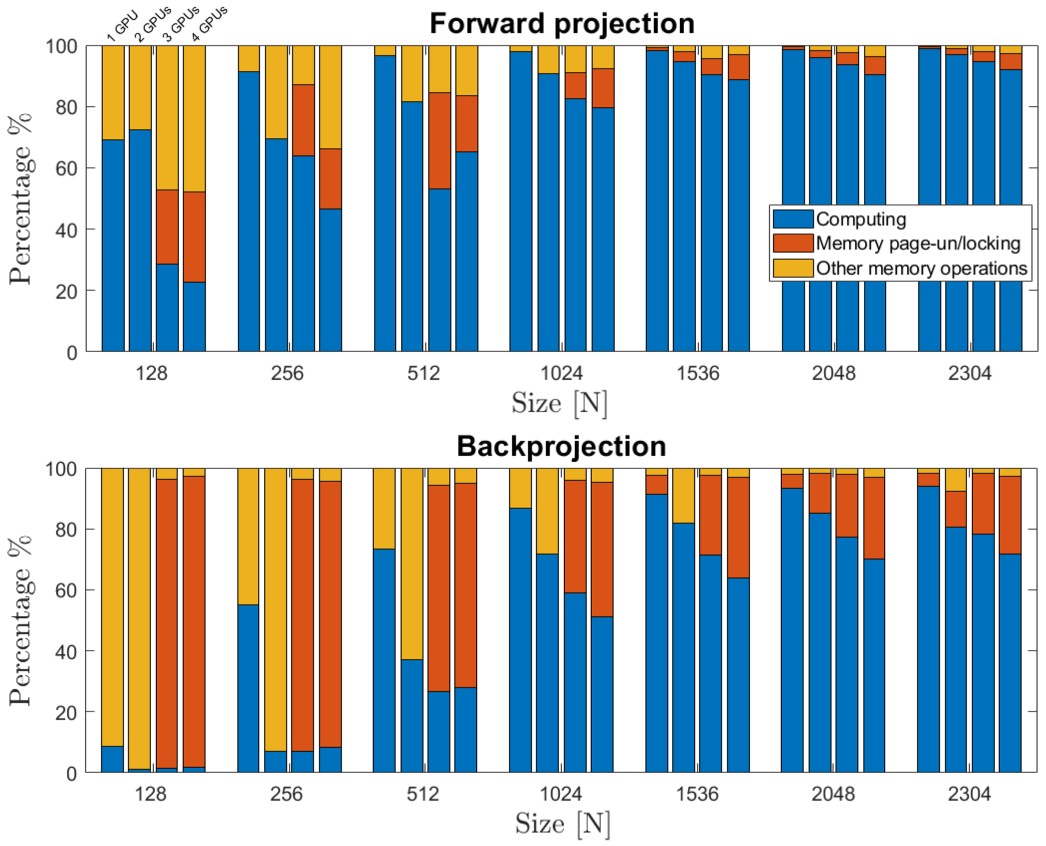


Figure 9: Percentage of the total execution times by different operations compared to the image size on different numbers of GPUs. Operations are grouped into three labels: Computing, memory page-locking and unlocking and other memory operations, such as allocating, freeing and memory transfer operations that are not concurrent with execution.

3.2 Comparison against CPU implementation

As previously discussed, an alternative to GPU-memory unbounded CT reconstruction is to split the image in the CPU and to perform smaller forward and backprojections on the GPUs, with the results gathered on the CPU. This is the approach taken by the Python implementation of the ASTRA toolbox. To demonstrate the advantages of our method, we here compare our new version of TIGRE with ASTRA v1.9.9[5].

We run both methods on the same dual GPU system introduced above, using 1 and 2 GPUs for the same range of problem sizes used previously in Figure 7. To show relative performance differences, we plot the ratio between computation times achieved by ASTRA and TIGRE in Figure 10, showing both forward and backprojection results.

Whilst TIGRE is slower for smaller problems, for larger problems, where data size becomes an issue, it is clear that our new efficient implementation in TIGRE is significantly faster than ASTRA. Starting at sizes of 512, TIGRE starts to outperform ASTRA whenever splitting is used (2 GPUs). For larger problems, the improvements become significant, both in ratio and wall time (where the difference can be counted in minutes). It is noteworthy that due to its memory management strategy, TIGRE can fit a much larger image size into a single GPU call. Also note that sizes bigger than

[5]Given the constant development in the ASTRA and TIGRE codebase, this experiment should not be seen as an ASTRA-versus-TIGRE comparison in general, but as a comparison between a method that uses the efficient memory managing and problem splitting strategies presented here and a method that does not currently use this strategy.

2048 have not been compared as ASTRA runs out of RAM at this point, which is likely due to the gathering operation requiring at least 2 image copies.

TIGRE's slower performance on the small problems shown here stems from two factors. Firstly, ASTRA is slightly more optimised than TIGRE, so that its GPU kernels are slightly faster. Secondly, our proposed method can have an influence on memory transfer as pinning memory can add additional workload to the procedure, as seen in Figure 9. However, computation times in this small problem regime are measured in milliseconds, thus while the difference in ratio is large, the difference in total wall time is quite small.

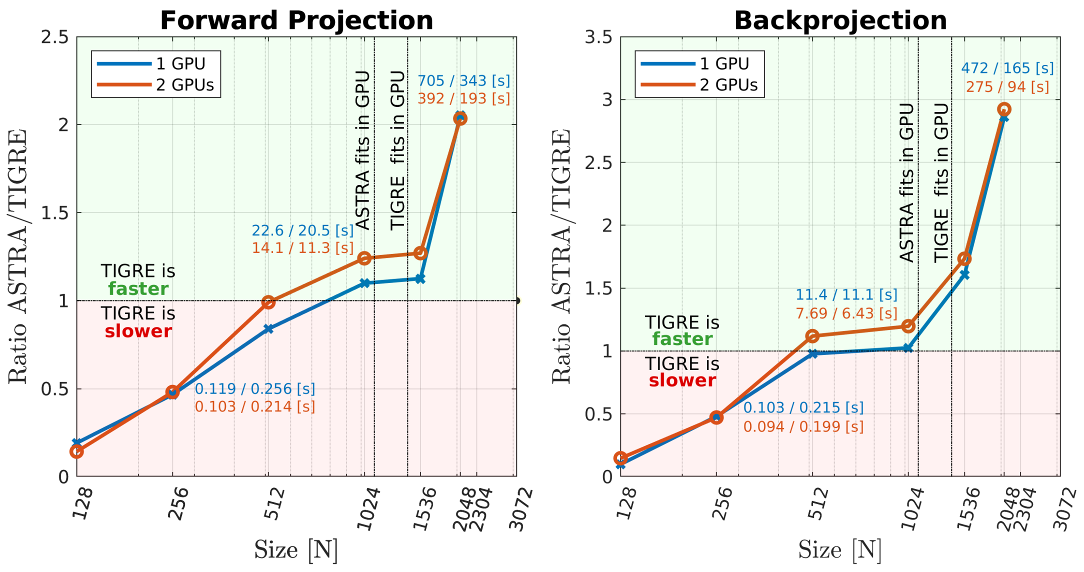


Figure 10: Ratio between the ASTRA and TIGRE projector types in execution time, for one and two GPUs. This figure showcases the improvements that the splitting and memory management methods provide. Some sizes showcase total measured execution time, for scale.

3.3 CT Image reconstruction

In order to showcase the capabilities of the toolbox with the multi-GPU acceleration two images from two different scanners and at different resolutions are reconstructed. The computer used is the previously described dual-GPU machine with 256GB of RAM. The first scan is of a roasted coffee bean and the second of a piece of a fossilized species of Ichthyosaur.

Coffee Bean

A roasted coffee bean was scanned on a Zeiss Xradia Versa 510. Projections of size 900 $\times$ 3780 where acquired at 2134 angles. This projection set requires 29 GB of memory. An image of 3340 $\times$ 3340 $\times$ 900 voxels was reconstructed of size 12.2 $\times$ 12.2 $\times$ 3.28 mm^3, requiring 40 GB of memory. As previously mentioned the 2 GPUs on the reconstructed machine have 22 GiB of RAM between them.

Figure 11 shows a slice of the reconstruction using the FDK algorithm (left) and the CGLS algorithm after 30 iterations. The CGLS reconstruction required 4 hours and 21 minutes. While at full angular sampling the FDK algorithm provides a high quality image, whereas using a third of the angles it starts showing noise artefacts over the reconstructed images. The CGLS reconstruction is more robust against this noise.

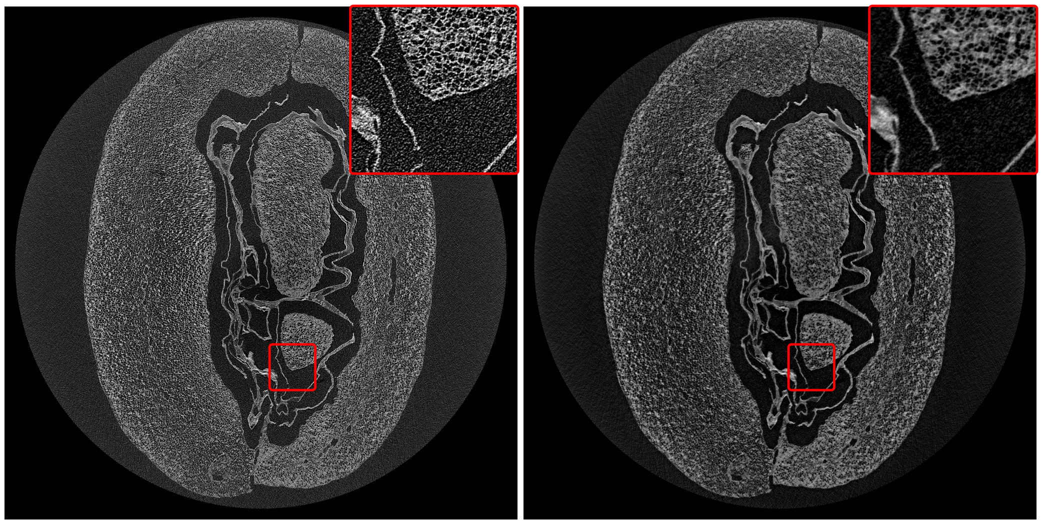


Figure 11: FDK (left) and CGLS with 30 iterations (right) reconstructed slices of image size 3340×3340×900 voxels of a roasted coffee bean. The reconstruction was performed in a 2 GPU machine.

Ichthyosaur fossil on a Nikon/Metris 225kVp/450kVp custom bay

The dataset used for this large image reconstruction is a small part of a fossilized Ichthyosaur species [50], the tip of the fins specifically.

The reconstruction was performed using the OS-SART algorithm, a variation of the classic SART algorithm that updates the volume using partial projection subsets instead of updating it for each projection. The subset size was 200 projections and it was executed using 50 iterations, taking 6 hours and 40 minutes of execution time. 2000 uniformly sampled projections were used of 4000 × 2000 pixels each. The image is 14.5 GB and the projections used 62 GB. Figure 12 shows the specimen and slices from the reconstructed image.

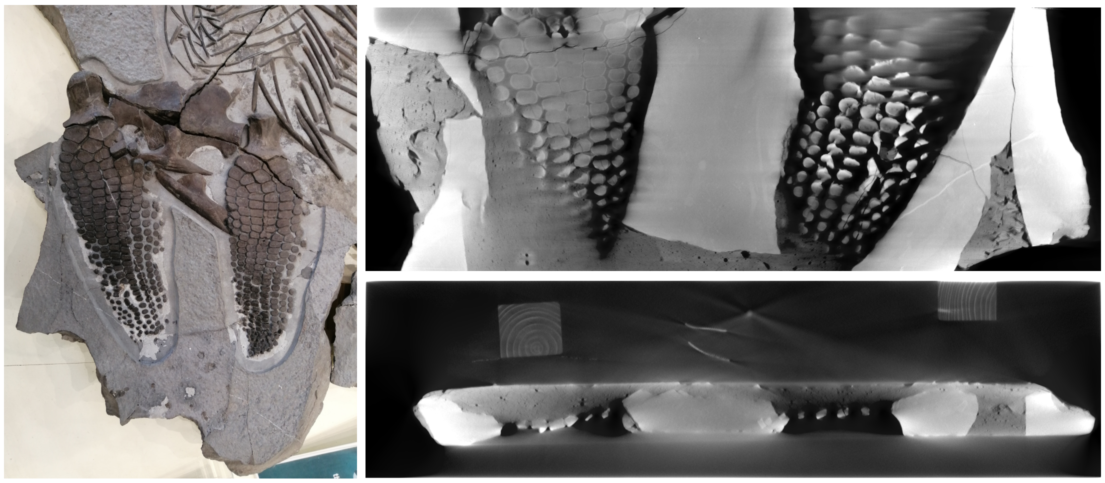


Figure 12: Photography of the Ichthyosaur fossil (left) and OS-SART with 50 iterations (right) reconstructed slices of image size 3360×900×2000 voxels. The reconstruction was performed in a 2 GPU machine.

4 Discussion

The results present a clear picture: with the proposed kernel launch structure very large images can be easily reconstructed in parallel using multiple GPUs, reducing the amount of image partitions needed. They also show how using multiple GPUs accelerates the reconstruction almost linearly, especially for bigger images. There are, however, some possible improvements that can be added into the presented structure and code.

The simultaneous memory transfer and computation queueing strategy is the improvement that allows the multi-GPU code in this work to achieve higher parallelism with little overhead, as there is no need to wait for projections to be copied in or out of the GPUs before or after the computation. However, in the work by Zinsser *et al* [41] a strategy that can be applied to overlap the image memory transfer is presented. In their work, the projections are entirely kept in the GPU memory but copied partially in pieces of incremental size, while their computation is happening. Similarly, they divide the image into two slices in a single GPU, and start copying one of the pieces out to the CPU while the other one finishes computing. The splitting of the image into various slices per GPU can also be applied in the projection operation. This would allow in both operators to slightly decrease the memory transfer times. However, to ensure simultaneous compute and memory transfer, the image needs to be split into very computer- and problem-specific sizes. Memory transfer speeds and different projection/image sizes are the main variables that influence how many splits the image would require. As TIGRE attempts to be as flexible as possible, this improvement has not been implemented, but it is noted here, as application specific implementation of CT algorithms may want to consider it. The problem of memory operations possibly taking longer than kernels does not exist when handling projections, as the memory transfer times are always significantly faster than the computation of the kernels, unless an unrealistically big difference exists between the projection and image sizes.

Some improvement can be achieved in the GPU code, especially in the backprojection, if an already preallocated memory piece can be given to the code. Due to the modular design of the toolbox, each kernel call allocates memory for its output, but due to the nature of iterative algorithms, it is likely that a variable with memory to overwrite already exists. We decided not to include this improvement and keep the modular design of TIGRE as it was.

For some scan geometries (such as standard rotational tomography), certain partitions of the reconstruction volume automatically lead to natural splits of the projections, as parts of the projections do not influence parts of the reconstruction volume and thus would not require being loaded into GPU memory. Whilst exploiting this could lead to further efficiency improvements, we do not do this here: firstly, as it would restrict TIGRE to limited scan geometries; and secondly, this has minimal impact on the overall computational and memory transfer times as projections are loaded simultaneously with computation (which dominates the cost) and as the additional gathering operation takes less than 0.01% of the total time.

The upper bound for memory is set by the CPU RAM available, however the GPU codes have their own limitation. If the values for the fastest execution of kernels are respected (N_{angles} in the projection operation, N_{angles} and N_z in the backprojection), and using a 11 GiB device RAM for the GPUs as available memory with an image of N^3 size with N^2 pixel N projections, the GPUs could handle $N = 17000$ before running out of memory for the projection and $N = 8500$ for the backprojection. By relaxing the limits and requiring a single image slice and projection in each GPU the limit can be increased to $N = 27000$. The first limit would require over 2TiB of CPU RAM to store the image, while the more permissive GPU limits would require 74 TiB of CPU RAM. We conclude that images of any of these sizes are unlikely to be used in the near future and that this code can safely be claimed to be able to reconstruct arbitrarily large images on GPUs, even if an upper limit exists.

The corollary is that in any realistic application the upper bound of the possible image size will be set by the CPU RAM. In addition to the limitations of storing the projections and image entirely on the CPU RAM, most iterative algorithms require auxiliary variables of the same sizes. Therefore the realistic upper bound will be set by the chosen algorithm. Using ROM as swap memory or even just writing and reading to file may increase the image size beyond the available RAM, but will result in a severe performance bottleneck due to continuous reading and writing of the auxiliary variables to ROM. We have decided not to implement this explicitly[6].

In any case, it is possible to accelerate the general tomography code available in TIGRE provided constraints are set on the sizes it would run. As Figure 9 shows, different problem sizes would require different optimization strategies. However we present a code that works well in a generic setting, for any size and amount of GPUs. Accelerating the code for a given set of GPUs with specific architecture, with a specific problem size is left to the users.

On the multi-GPU side, the presented splitting method shows how single-node machines can work with multiple GPUs, but it is limited to this computer topology. Fortunately the strategy itself can be combined with a higher level parallelization for multiple nodes, similar to what the ASTRA toolbox provides [22]. Using the MPI protocol the code could be adapted to multiple node topologies, such as in an HPC.

Computational results show that not only very large images can be successfully reconstructed regardless of the GPU, but that current image sizes can be reconstructed very fast. For a medical image size (generally smaller than 512^3), iterative reconstruction can be achieved in less than 1 second per iteration on properly chosen hardware. As a rough comparison, the original TIGRE article suggests that a reconstruction of 512^3 medical image using 15 iterations of CGLS can be achieved in 4min41s. With the proposed implementation the same problem can be solved in 1min01s on a single GTX 1080 Ti.

Results in Figure 9 show further possible speed-ups. For a size of $N = 512$, the memory operations take a visible amount of the total time in the forward projection and are the main operation in the backprojection. However, most of these operations only exist due to the modular design of TIGRE. If the algorithms were to be written directly in C++/CUDA, these operations would only be needed once in the entire algorithm instead of being used in each projection and backprojection call. Such an implementation would therefore speed up the algorithm even more. Nevertheless, the presented code is ready to be used for time-critical medical cases, as reconstructions in less than a minute are more than possible using it.

[6]Anyone using the code can set up their machine to use swap memory.

5 Conclusions

In this work we propose a projection and backprojection operation splitting strategy that allows seamlessly splitting the computation over multiple GPUs in a single node with little to no overhead. The method operates with arbitrarily large images removing the common limit of GPU RAM size, pushing the upper bound of the maximum size that can be reconstructed in CT in single machines and allowing significant speed-up for multiple GPU machines. Similarly, it allows reconstruction of images in machines with small GPUs. Additionally, the method allows for faster reconstruction for images that fit on GPUs. This has a clear application in medical CT, where in some cases reconstruction time is critical. With the current implementation, iterative reconstruction of medical sized images can be performed in less than a minute. An implementation of the proposed method is freely available integrated into the latest version of the TIGRE toolbox at github.com/CERN/TIGRE.

Acknowledgements

We gratefully acknowledge the support of NVIDIA Corporation with the donation of the Titan Xp GPU used for this research. This research was supported by EPSRC grant EP/R002495/1 and the European Metrology Research Programme through grant 17IND08. The authors acknowledge the use of the IRIDIS High Performance Computing Facility, and associated support services at the University of Southampton, in the completion of this work. We thank Chris Moore Fossils and Sally Thompson for sharing the Ichthyosaur fossil data.

References

[1] J. Kruth, M. Bartscher, S. Carmignato, R. Schmitt, L. D. Chiffre, and A. Weckenmann, "Computed tomography for dimensional metrology," *CIRP Annals - Manufacturing Technology*, vol. 60, no. 2, pp. 821 – 842, 2011.

[2] J. M. Warnett, V. Titarenko, E. Kiraci, A. Attridge, W. R. Lionheart, P. J. Withers, and M. A. Williams, "Towards in-process x-ray CT for dimensional metrology," *Measurement Science and Technology*, vol. 27, no. 3, p. 035401, 2016.

[3] D. Létourneau, J. W. Wong, M. Oldham, M. Gulam, L. Watt, D. A. Jaffray, J. H. Siewerdsen, and A. A. Martinez, "Cone-beam-CT guided radiation therapy: technical implementation," *Radiotherapy and Oncology*, vol. 75, no. 3, pp. 279–286, 2005.

[4] D. C. Hansen and T. S. Sørensen, "Fast 4D cone-beam CT from 60s acquisitions," *Physics and Imaging in Radiation Oncology*, vol. 5, pp. 69–75, 2018.

[5] G. S. Desai, R. N. Uppot, E. W. Yu, A. R. Kambadakone, and D. V. Sahani, "Impact of iterative reconstruction on image quality and radiation dose in multidetector CT of large body size adults," *European Radiology*, vol. 22, no. 8, pp. 1631–1640, 2012.

[6] W. Mao, C. Liu, S. J. Gardner, F. Siddiqui, K. C. Snyder, A. Kumarasiri, B. Zhao, J. Kim, N. W. Wen, B. Movsas, and I. J. Chetty, "Evaluation and clinical application of a commercially available iterative reconstruction algorithm for CBCT-based IGRT," *Technology in Cancer Research & Treatment*, vol. 18, 2019. PMID: 30803367.

[7] B. Kataria, J. N. Althén, Ö. Smedby, A. Persson, H. Sökjer, and M. Sandborg, "Assessment of image quality in abdominal CT: potential dose reduction with model-based iterative reconstruction," *European Radiology*, vol. 28, p. 2464–2473, 2018.

[8] X. Yu, H. Wang, W.-c. Feng, H. Gong, and G. Cao, "cuart: Fine-grained algebraic reconstruction technique for computed tomography images on gpus," in *2016 16th IEEE/ACM*

International Symposium on Cluster, Cloud and Grid Computing (CCGrid), pp. 165–168, IEEE, 2016.

[9] E. Romero, A. Tomás, A. Soriano, and I. Blanquer, "A fast sparse block circulant matrix vector product," in *Euro-Par 2014 Parallel Processing* (F. Silva, I. Dutra, and V. Santos Costa, eds.), (Cham), pp. 548–559, Springer International Publishing, 2014.

[10] X. Yu, H. Wang, W.-c. Feng, H. Gong, and G. Cao, "An enhanced image reconstruction tool for computed tomography on gpus," in *Proceedings of the Computing Frontiers Conference*, pp. 97–106, ACM, 2017.

[11] X. Yu, H. Wang, W.-c. Feng, H. Gong, and G. Cao, "GPU-based iterative medical CT image reconstructions," *Journal of Signal Processing Systems*, vol. 91, no. 3-4, pp. 321–338, 2019.

[12] R. C. Atwood, A. J. Bodey, S. W. Price, M. Basham, and M. Drakopoulos, "A high-throughput system for high-quality tomographic reconstruction of large datasets at Diamond Light Source," *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, vol. 373, no. 2043, p. 20140398, 2015.

[13] C. Melvin, M. Xu, and P. Thulasiraman, "HPC for iterative image reconstruction in CT," in *Proceedings of the 2008 C 3 S 2 E conference*, pp. 61–68, ACM, 2008.

[14] A. Mirone, E. Brun, E. Gouillart, P. Tafforeau, and J. Kieffer, "The PyHST2 hybrid distributed code for high speed tomographic reconstruction with iterative reconstruction and a priori knowledge capabilities," *Nuclear Instruments and Methods in Physics Research Section B: Beam Interactions with Materials and Atoms*, vol. 324, pp. 41–48, 2014.

[15] D. M. Pelt, D. Gürsoy, W. J. Palenstijn, J. Sijbers, F. De Carlo, and K. J. Batenburg, "Integration of TomoPy and the ASTRA toolbox for advanced processing and reconstruction of tomographic synchrotron data," *Journal of Synchrotron Radiation*, vol. 23, no. 3, pp. 842–849, 2016.

[16] T. Benson and J. Gregor, "Framework for iterative cone-beam micro-CT reconstruction," in *IEEE Symposium Conference Record Nuclear Science 2004.*, vol. 5, pp. 3253–3257 Vol. 5, 2004.

[17] J. M. Rosen and J. Wu, "Iterative helical CT reconstruction in the cloud for ten dollars in five minutes," in *Proceedings of the 12th Fully Three-Dimensional Image Reconstruction in Radiology and Nuclear Medicine 2013*, 2013.

[18] M. Kachelrieß, M. Knaup, and O. Bockenbach, "Hyperfast perspective cone–beam backprojection," in *2006 IEEE Nuclear Science Symposium Conference Record*, vol. 3, pp. 1679–1683, 2006.

[19] W. J. Palenstijn, J. Bédorf, and K. J. Batenburg, "A distributed SIRT implementation for the ASTRA toolbox," in *Scientific Computing*, 2015.

[20] A. Fehringer, T. Lasser, I. Zanette, P. B. Noël, and F. Pfeiffer, "A versatile tomographic forward-and back-projection approach on multi-GPUs," in *Medical Imaging 2014: Image Processing*, vol. 9034, p. 90344F, 2014.

[21] Y. Zhu, Y. Zhao, and X. Zhao, "A multi-thread scheduling method for 3D CT image reconstruction using multi-GPU," *Journal of X-ray Science and Technology*, vol. 20, no. 2, pp. 187–197, 2012.

[22] W. J. Palenstijn, J. Bédorf, J. Sijbers, and K. J. Batenburg, "A distributed ASTRA toolbox," *Advanced structural and chemical imaging*, vol. 2, no. 1, p. 19, 2017.

[23] E. S. Jimenez, L. J. Orr, and K. R. Thompson, "An irregular approach to large-scale computed tomography on multiple graphics processors improves voxel processing throughput," in *2012 SC Companion: High Performance Computing, Networking Storage and Analysis*, pp. 254–260, 2012.

[24] J. W. Buurlage, R. H. Bisseling, and K. J. Batenburg, "A geometric partitioning method for distributed tomographic reconstruction," *Parallel Computing*, vol. 81, pp. 104–121, 2019.

[25] K. Hämäläinen, L. Harhanen, A. Hauptmann, A. Kallonen, E. Niemi, and S. Siltanen, "Total variation regularization for large-scale X-ray tomography," *International Journal of Tomography and Simulation*, vol. 25, no. 1, pp. 1–25, 2014.

[26] H. H. B. Sørensen and P. C. Hansen, "Multicore performance of block algebraic iterative reconstruction methods," *SIAM Journal on Scientific Computing*, vol. 36, no. 5, pp. C524–C546, 2014.

[27] Y. Gao and T. Blumensath, "A joint row and column action method for cone-beam computed tomography," *IEEE Transactions on Computational Imaging*, vol. 4, no. 4, pp. 599–608, 2018.

[28] M. Käseberg, S. Melnik, and E. Keeve, "OpenCL accelerated multi-GPU cone-beam reconstruction," in *Proceedings of the 12th International Meeting on Fully Three-Dimensional Image Reconstruction in Radiology and Nuclear Medicine*, pp. 477–480, 2013.

[29] J. Bardino, M. Rehr, and B. Vinter, "Cph CT toolbox: A performance evaluation," in *2015 International Conference on High Performance Computing Simulation (HPCS)*, pp. 35–46, 2015.

[30] A. Biguri, M. Dosanjh, S. Hancock, and M. Soleimani, "TIGRE: a MATLAB-GPU toolbox for CBCT image reconstruction," *Biomedical Physics & Engineering Express*, vol. 2, no. 5, p. 055010, 2016.

[31] C.-Y. Chou, Y.-Y. Chuo, Y. Hung, and W. Wang, "A fast forward projection using multithreads for multirays on GPUs in medical image reconstruction," *Medical Physics*, vol. 38, no. 7, pp. 4052–4065, 2011.

[32] D. Schlifske and H. Medeiros, "A fast GPU-based approach to branchless distance-driven projection and back-projection in cone beam CT," in *SPIE Medical Imaging*, pp. 97832W–97832W, 2016.

[33] Y. Okitsu, F. Ino, and K. Hagihara, "High-performance cone beam reconstruction using CUDA compatible GPUs," *Parallel Computing*, vol. 36, no. 2, pp. 129–141, 2010.

[34] Y. Long, J. A. Fessler, and J. M. Balter, "3D forward and back-projection for X-ray CT using separable footprints," *IEEE Transactions on Medical Imaging*, vol. 29, no. 11, pp. 1839–1850, 2010.

[35] R. L. Siddon, "Fast calculation of the exact radiological path for a three-dimensional CT array," *Medical Physics*, vol. 12, no. 2, pp. 252–255, 1985.

[36] W. Palenstijn, K. Batenburg, and J. Sijbers, "Performance improvements for iterative electron tomography reconstruction using graphics processing units (GPUs)," *Journal of Structural Biology*, vol. 176, no. 2, pp. 250–253, 2011.

[37] X. Yu and M. Becchi, "Gpu acceleration of regular expression matching for large datasets: exploring the implementation space," in *Proceedings of the ACM International Conference on Computing Frontiers*, p. 18, ACM, 2013.

[38] Y. Hu, H. Liu, and H. H. Huang, "Tricore: Parallel triangle counting on gpus," in *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 171–182, IEEE, 2018.

[39] X. Jia, Y. Lou, J. Lewis, R. Li, X. Gu, C. Men, W. Y. Song, and S. B. Jiang, "GPU-based fast low-dose cone beam CT reconstruction via total variation," *Journal of X-ray Science and Technology*, vol. 19, no. 2, pp. 139–154, 2011.

[40] E. Papenhausen, Z. Zheng, and K. Mueller, "GPU-accelerated back-projection revisited: squeezing performance by careful tuning," in *Fully Three-Dimensional Image Reconstruction in Radiology and Nuclear Medicine*, 2011.

[41] T. Zinsser and B. Keck, "Systematic performance optimization of cone-beam back-projection on the Kepler architecture," in *Proceedings of the 12th Fully Three-Dimensional Image Reconstruction in Radiology and Nuclear Medicine*, 2013.

[42] G.-H. Chen, J. Tang, and S. Leng, "Prior image constrained compressed sensing (PICCS): a method to accurately reconstruct dynamic CT images from highly undersampled projection data sets," *Medical Physics*, vol. 35, no. 2, pp. 660–663, 2008.

[43] O. Semerci, N. Hao, M. E. Kilmer, and E. L. Miller, "Tensor-based formulation and nuclear norm regularization for multienergy computed tomography," *IEEE Transactions on Image Processing*, vol. 23, no. 4, pp. 1678–1693, 2014.

[44] E. L. Piccolomini and F. Zama, "The conjugate gradient regularization method in computed tomography problems," *Applied Mathematics and Computation*, vol. 102, no. 1, pp. 87–99, 1999.

[45] E. Y. Sidky and X. Pan, "Image reconstruction in circular cone-beam computed tomography by constrained, total-variation minimization," *Physics in Medicine & Biology*, vol. 53, no. 17, p. 4777, 2008.

[46] M. Lohvithee, A. Biguri, and M. Soleimani, "Parameter selection in limited data cone-beam CT reconstruction using edge-preserving total variation algorithms," *Physics in Medicine & Biology*, vol. 62, no. 24, p. 9295, 2017.

[47] L. I. Rudin, S. Osher, and E. Fatemi, "Nonlinear total variation based noise removal algorithms," *Physica D: Nonlinear Phenomena*, vol. 60, no. 1-4, pp. 259–268, 1992.

[48] F. Knoll, M. Unger, C. Diwoky, C. Clason, T. Pock, and R. Stollberger, "Fast reduction of undersampling artifacts in radial MR angiography with 3D total variation on graphics hardware," *Magnetic Resonance Materials in Physics, Biology and Medicine*, vol. 23, no. 2, pp. 103–114, 2010.

[49] A. Beck and M. Teboulle, "A fast iterative shrinkage-thresholding algorithm for linear inverse problems," *SIAM journal on imaging sciences*, vol. 2, no. 1, pp. 183–202, 2009.

[50] S. Thomson, "Attenborough and the sea dragon," *BBC*, 2018.