Using Futhark for a fast, parallel implementation of forward and back projection in algebraic reconstruction methods - A pre-study

Mette Bjerg and Lærke Pedersen

November 2, 2018

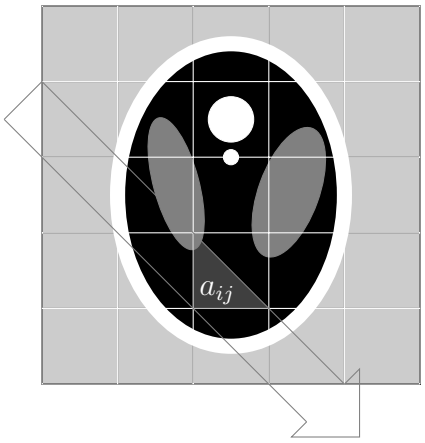

Figure 1: Example of a weighting a_{ij}.

1 Introduction

Computed tomography is the inverse problem of reconstructing an image or volume from its x-ray projections. The x-ray source spins around the object to be analyzed and sends x-rays that hit a detector on the oposite side. The detector shows how much of each x-ray was attenuated when passing through the volume. In this report we will focus on the bottlenecks of an algebraic reconstruction algorithm called the Simulataneous Iterative Reconstruction Algorithm (SIRT) and investigate how we may use the high level data-parallel language Futhark for implememting a fast version of this algorithm. The reason for chosing this algorithm is that it provides good reconstruction quality under non-optimal circumstances, but unfortunately has very poor performance compared to techniques such as filtered backprojection.
In algebraic reconstruction we solve the problem as a linear system of equations. The main idea is that the process can be modelled as a linear transformation by discretizng the object to be reconstructed into N pixels. First we place a coordinate system with origo at the center of the object to be reconstructed and denote by $\boldsymbol{\theta}$ the vector of angles between the positions of the source and origo. For each angle several x-rays are cast from the source. We denote by the vector $\boldsymbol{\rho}$ the signed distances from each line to origo. The data produced by the process is called the sinogram. Then the sinogram values p_i for each (θ_k, ρ_l) are a weighted sum of the attenuations at each pixel f_j that the (θ_k, ρ_l) ray passes through:

$$\sum_{j=1}^{N} a_{ij} f_j = p_i \tag{1}$$

Where a_{ij} are the weights, corresponding to the fraction of the pixel j that the ray i covers.

Writing all the projections as a column vector $\boldsymbol{p}$ and the attenuation values to be reconstructed as a column vector $\boldsymbol{f}$ the weightings are represented as an $M \times N$ matrix

$\boldsymbol{A}$, we obtain a linear system of the form:

$$\boldsymbol{p} = \boldsymbol{A}\boldsymbol{f} \tag{2}$$

These systems of equations may easily be solved under the right circumstances, where $M = N$. However this is rarely the case. In most real cases $M > N$, i.e. the number of projections is larger than the number of pixels to be reconstructed and the size of the matrix is very large - more about this in the next section.
However, the algebraic reconstruction methods also have some advantages. Since the model closely relates to the real world scenario the weightings can be refactored to take irregularities in the setup, such as differences in beam energies or irregular geometries and missing data into account. Furthermore these methods generally give better image quality than analytic methods when the data is sparse.

A linear system like this is typically solved by minimizing some norm:

$$\|\boldsymbol{A}\boldsymbol{f} - \boldsymbol{p}\| \tag{3}$$

An example is the SIRT algorithm. The action of the matrix $\boldsymbol{A}$ is called the *forward projection*, and the matrix itself is called the *system matrix*,*projection matrix* or *radon matrix*. Each row of $\boldsymbol{A}$ represents the coefficients of the equation for one ray. The action of the transposed projection matrix $\boldsymbol{A}^T$ on the sinogram vector is called the *backprojection*, and can be vizualized as smearing the projection values across the reconstruction in the direction of the ray, or equivalently summing up the contrubitions of each ray for a given pixel. The idea behind the SIRT algorithm is to forward project the current reconstruction, then subtract this from the original projection data and do a weighted backprojection resulting in a correction factor which can be added to the current reconstruction. The update equeation is:

$$\boldsymbol{f}^n = \boldsymbol{f}^{(n-1)} + \boldsymbol{C}\boldsymbol{A}^T\boldsymbol{R}(\boldsymbol{p} - \boldsymbol{A}\boldsymbol{f}^{(n-1)}), \tag{4}$$

where $\boldsymbol{C}$ and $\boldsymbol{R}$ are the diagonal matrices containing the inverse column and row sums of the system matrix respectively.

It can be shown that this iterative scheme solves the problem:

$$\boldsymbol{f}^* = argmin_{\boldsymbol{f}} \|\boldsymbol{p} - \boldsymbol{A}\boldsymbol{f}\|_{\boldsymbol{R}}, \tag{5}$$

where $\|\boldsymbol{x}\|_{\boldsymbol{R}} = \boldsymbol{x}^T\boldsymbol{R}\boldsymbol{x}$.
The backprojection and foward projection operations are standard operations in many iterative algebraic reconstruction methods and are the bottle necks of the algorithms [2].

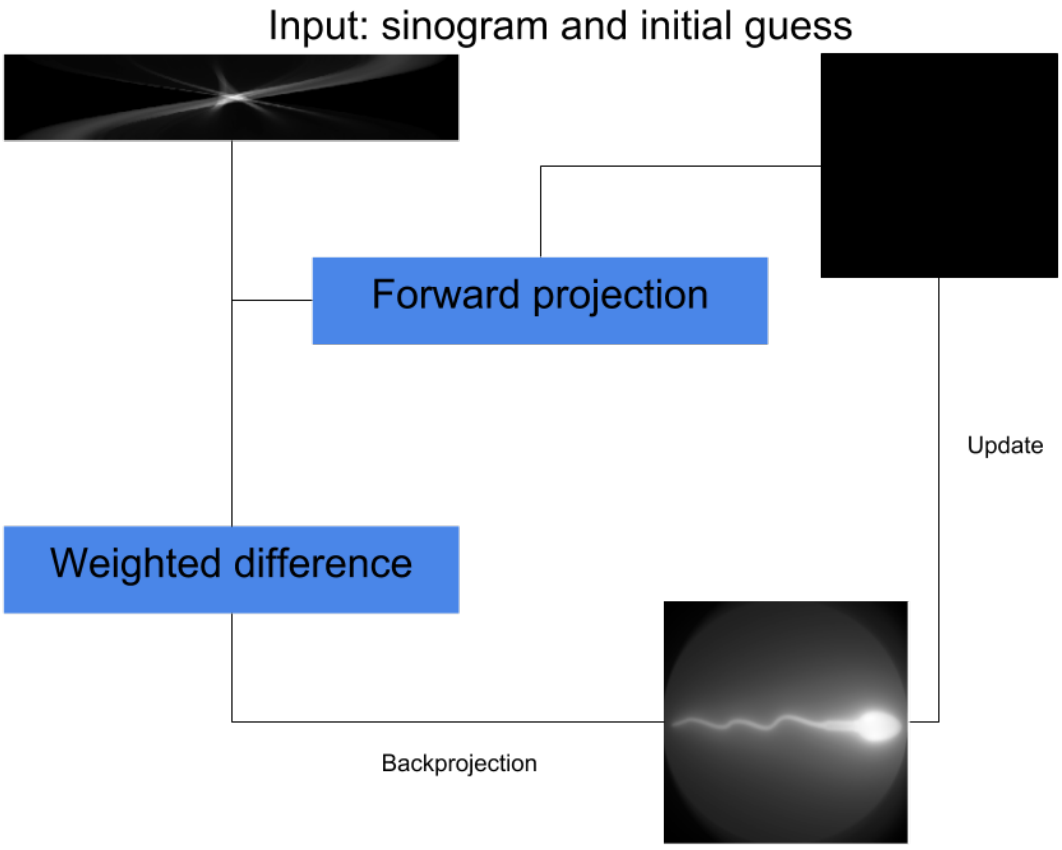

Figure 2: An illustration of the SIRT algorithm.

Therefore, our main focus has been on optimizing these operations.
Different beam geometries exist, such as parallel beams, fan beams and cone beams. The geometry must be considered when constructing the system matrix. However as this study is intended for applications with synchrotron data only parallel beam geometries will be considered. Furthermore, we will only consider reconstructions of 2D images, since this is simply a pre-study and it makes the problem conceptually easier to follow.

2 Computing the projection matrix

One of the main problems when parallelizing the algorithm is that the amount of data in real applications is huge. Images from synchrotrons are generated with detectors of sizes up to about 4000×4000, with a typical detector size being about 2048×2048. To accurately generate 3D reconstructions it has been shown that approximately $\frac{\pi \cdot N}{2}$ vieweing angles are needed, where N is the size of the detector in one direction [3].
We benchmarked our algorithms by using sizes N ranging from 128 to 4096 and using $\lceil \frac{\pi \cdot N}{2} \rceil$ angles and N lines for each of these sizes to make the problem sizes realistic. However, in the future it might be worth investigating how the reconstructions look when using fewer angles, since one of the strengths of the SIRT algorithm is that it produces good results with few angles.
To solve the large problems it is not possible to store the whole system matrix on the GPU. Lets take as an example the amount of space used to store the matrix for a 3D problem with a $N = 4000$ detector, with $\lceil \frac{\pi \cdot N}{2} \rceil$ angles, and 4000 rays per angle. The matrix is sparse having at most $2 \cdot N - 1$ entries in each row, so instead of storing the whole matrix with all the zeroes, we will consider a semi-sparse representation with one 2 dimensional array containing the data as floats in arrays on each row of size $2 \cdot N - 1$, and one matching 2 dimensional array containing the column indexes of the datapoints. This is also the representation we have used in our implementations. For the example sketched above the number of rows of each matrix will be $\lceil \frac{\pi \cdot 4000}{2} \rceil \cdot 4000$ and the number of columns is $2 \cdot 4000 - 1$. Assuming ints and floats both take up 4 bytes each then storing the matrix requires $4 \cdot 2 \cdot \lceil \frac{\pi \cdot 4000}{2} \rceil \cdot 4000 \cdot (2 \cdot 4000 - 1) \approx 1TB$. This is of course one of the largest problems, but even for a problem with $N = 512$ we're looking at the matrix taking up around $3 - 4GB$. Besides the matrix we also need to store the sinogram data which will take approximately $420MB$, and the reconstruction taking up around $270MB$. Hence even for relatively small problems we're looking at an estimated $5GB$ of data. GPU memory is expensive, and is typically in the range 2-8GB for consumer end cards, and 10-48GB for high end cards. Thus, the small problems are not able to fit in a standard comsumer card, and large problems won't fit on any cards. When we tested our algorithms, we were not able to copute the system matrix on the universities GPU's for sizes larger than 256×256. There are several ways of getting around this problem. Some problems exhibit symmetries in the system matrix so that we only have to store part of the matrix. This is already somewhat accounted for in the above calculations. In the example we only considered

the matrix for one slice of a 3D reconstruction since it will be equivalent for other slices when using parallel beam geometry. Thus for cone beam geometries, the memory needed would be approximately N times greater. Other symetries involve angle symmetries, where for example a scan from a 45° or 135° angle would contain the same values for mirrored pixels. However these are obviously not generally applicable since they will depend on which angles are used in the reconstruction.
Another more flexible approach is to calculate the system matrix in chunks which is what we have done. For this we started by using the code from a bachelor project in which they did an implementation of finding the system matrix using futhark. We had two different parallel implementations in futhark and a sequential version in python which we compared. The sequential version in python was too slow (in the order of hours) for realistic problem sizes to be worth considering. When we compared the two bachelor projects one seemed to outperform the other both in terms of memory usage and computation time. Since one of the implementations ran in to problems with memory for some of the smallest problems, we decided to keep working with the most promising of the two. We call this algorithm *projectionmatrix_jh*.
The system matrix may be adequately approximated by considering the x-rays as being thin lines and calculating the length of intersections with each pixel in a grid. For simplicity we have only considered square grids with isotropic pixels.
The computation of the system matrix contains two levels of possible parallelization. The

```
1 for ray = 0; ray < numberofrays; ray++ //parallel
2         for pixel = 0; pixel < pixels; pixel++ //parallel
3                 if ray intersects pixel:
4         (p1,p2) = intersectionpoints pixel ray
5         A[ray][pixel] = distance p1 p2
```

Figure 3: A looped version of the projection matrix algorithm.

implementation from the bachelor project we used had parallelized over the outer loop - i.e for each ray. To compute the intersection lengths a sequential loop was run following each line. The number of rays is very high (6.586.368 for a grid of size $2048 * 2048$), and larger than the number of pixels, this seems a good choice. By following each line, the inner loop only runs $2 * N - 1$ times. From their report it seems they tried a naive implementation of computing the intersection with each pixel in parallel (akin to the pseudocode in 2) but found it to be too slow to consider.
However, when we had an implementation of the foward projection and analyzed the compiled code, we could clearly see that the most time was spend in the kernel consisting of the sequential loop doing grid line intersections. Therefore we decided to spend some more time on the system matrix computations. Futhark does a lot of optimizations, but can not merge code across a loop in a map so our first approach was to attempt to change the inner loop to a map. However, since the inner loop is inherently sequential because of cross loop dependencies, we did not completely succeed. We seperated out the part of the code which computes the intersection point, and did the distance calculations as a map.

We call this algorithm *projectionmatrix_ map*. Another issue with the projectionmatrix_jh is a lot of branches in the inner loop. We also tried to limit the computations in these branches as much as possible in projectionmatrix_map.
Since we did not manage to convert the inner loop to a map in projectionmatrix_jh, we also attemted a different approach, explained in [1]. It is based on the observation that each line with slope at most 1 intersects at most two rows of the grid in each column of the grid, and opposite for the lines with slope greater than 1. Hence there is potential for exploiting a second level of parallelism by parrallelizing over the columns or rows of the grid and finding the nontrivial intersections. They show a method for doing this in time $O(1)$. Unfortunately there is a small bug somewhere our implementation which seems to be related to the transition between lines with slope > 1 and those with slope < 1. We did not manage to find the error, but have decided to include the algorithm in the benchmarking anyway, since we believe it is a small error somewhere with no significant influence of the speed of the algorithm. We call this algorithm *projectionmatrix_ doubleparallel*.

We benchmarked the algorithms by generating the input, saving it to files and running with futhatk-bench. We used the opencl compiler, and did 10 runs. From the plot 2 we can see that the version where we exploit inner parallelism is indeed fastest. For the maximum size we were able to run, i.e 256 we got a speedup of about 1.2 compared to projectionmatrix_jh.

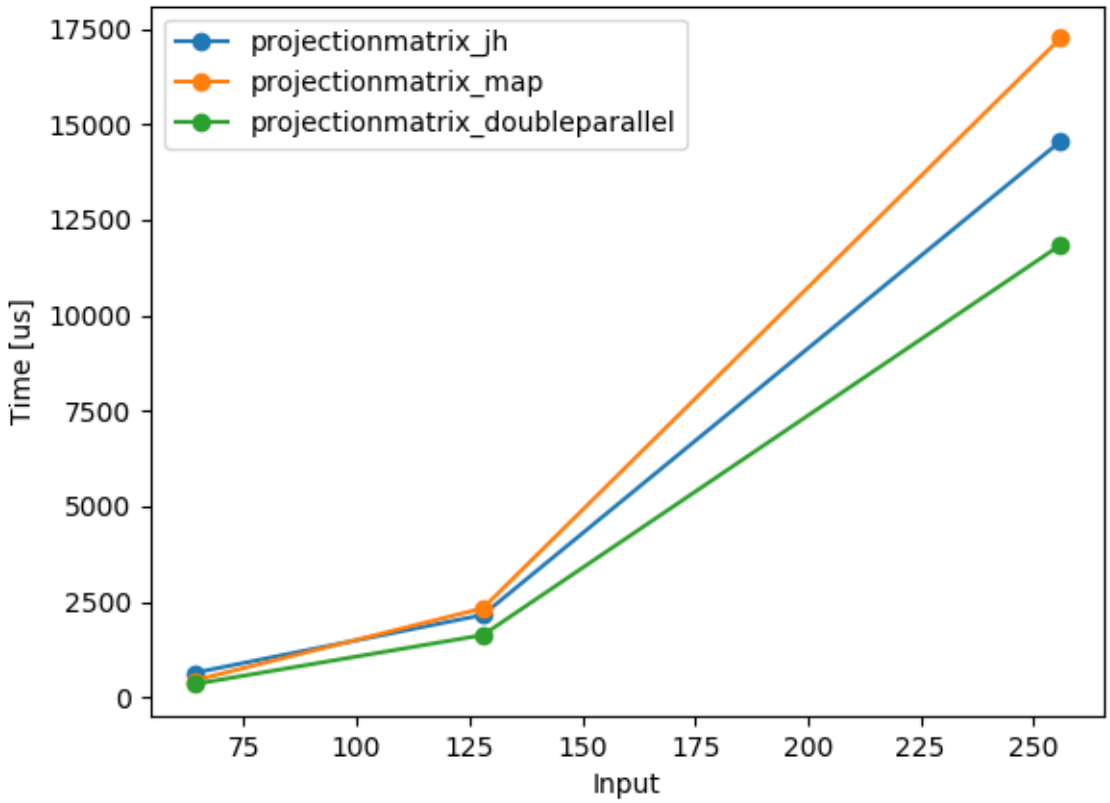

Figure 4: A lcomparison of the projection matrix algorithms run for gridsizes 64 to 256. Since the universities GPUs only have about 3GB of memory this was the largest size we could run for without chunking up the matrix.

3 Nested parallel forward and back projctions

A first approach at forward and backprojection was to do a nested parallel version and using computed chunks of the system matrix.
A looped version of the forward projection with the system matrix cut in *steps* chunks looks like this:

```
for step = 0; step < steps; step++
        A = getRays(raysperstep)
        for ray = 0; ray < raysperstep; row++
                acc = 0.0
                for p = 0; p<numpixels; p++
                        acc+= A[ray][p]*image[p]
                FP[step*raysperstep+ray] = acc

this can be written in futhark like pseudo code as:

loop (output, step, raysperstep)
        let A = getRays raysperstep step
        let partresult = map (\row -> reduce (+) 0 <| map ( \i -> row[i]*
            vector[i] ) (iota (length row)) ) A
        in (output++partresult, step, raysperstep)
```

Figure 5: A looped version of the forward projection, where the raysperstep should be chosen such that the computations fit in the memory. $step * raysperstep$ should equal the total number of rows.

```
for step = 0; step < steps; step++
        A = getRays(raysperstep)
        AT = A.transpose()
        for p = 0; p<numpixels; p++
                acc = 0.0
                for ray = 0; ray<raysperstep; ray++
                        acc+= AT[p][ray]*sinogram[ray]
                BP[p] += acc

this can be written in futhark like pseudo code as:

loop (output, step, raysperstep)
        let A = getRays raysperstep step
        let AT = transpose A
        let partresult = map (\row -> reduce (+) 0 <| map2 (*) row vect ) AT
        map2 (+) partresult output
        in (output, step, raysperstep)
```

Figure 6: A looped version of the back projection, where the raysperstep should be the largest number possible such that the computations fit in the memory. $step * raysperstep$ should equal the total number of rows.

4 Flattening and other optimization attempts

Exploiting nested parallelism, is difficult on GPU hardware since the hardware is organized on one, or maybe two, levels that allow threads to comminucate via shared scratchpad memory. Hence mapping the application level parallelism to the GPU requires a choice of which level to paralellize, since both level cannot be directly mapped.
One way to get around this problem is to use a flattening transformation. The problem with this is that it will require even more memory usage, and may prevent opportunities for locality optimizations. We tried to do a flattening of our forawrd and backward rpoejctions, since we have multiple levels of paralellism. The idea was to compute the matrix, the transform it to a sparse and flat version and use our sparse matrix vector multiplication from a previous assignment. However, it turned out to be quite complex, and a lot of code was required to transform the matrix to the correct sparse format. Since it later turned out that most of the time spend during the computations was during the matrix computation and we had many issues with running out of memory, this probably wasn't the best approach and we did not pursue it to the end. We report result of semiflat versions.
A much more promising approach seems to be feeding the data directly to the matrix computations, and not have to save the matrix data at all but only the end result. We managed to finish an implementation of this approach for forward projection by feeding the data directly to the projectiomatrix_doubleparallel version. Unfortunately, it ran out of resources so we need to investigate this further.

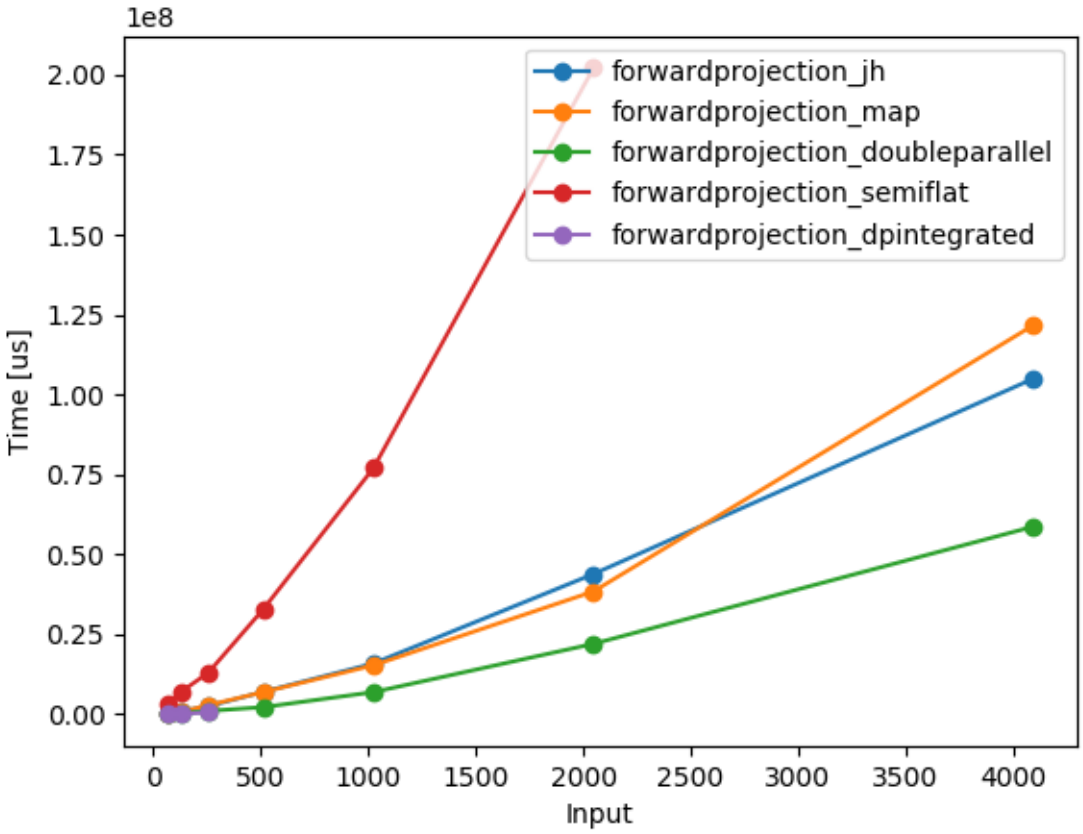

Figure 7: A comparison of the forward projections run with different matrix implementations and the version where the forward projection is integrated in projectiomatrix_doubleparallel. The chunk size was 32. As expected the semiflat version performed rather badly. The implementation using the double parallel version performed best. The rest of the algorithms are nested, using different projection matrix algorithms inside.

5 Comparison to a CUDA implementation

We compared our implementation of forward projection and back projection to implementations from the astra toolbox.

cite article about astra and write about how these are implemented - look into code/read articles from mat mat mult assignment it is clear that memory coalescence is important. How may we use it in this project?

6 Memory coalescence and other optimization ideas

How may we flatten the calculations of the system matrix Consider if the problem is suited for streaming or calculations on a cluster

References

[1] Hao Gao. Fast parallel algorithms for the x-ray transform and its adjoint. *Medical physics*, 39(23127102):7110–7120, November 2012.

[2] 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*, 29(11):1839–1850, Nov 2010.

[3] F. Natterer. *The Mathematics of Computerized Tomography*. Society for Industrial and Applied Mathematics, 2001.

[4] Z. Xue, L. Zhang, and J. Pan. A new algorithm for calculating the radiological path in ct image reconstruction. In *Proceedings of 2011 International Conference on Electronic Mechanical Engineering and Information Technology*, volume 9, pages 4527–4530, Aug 2011.