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

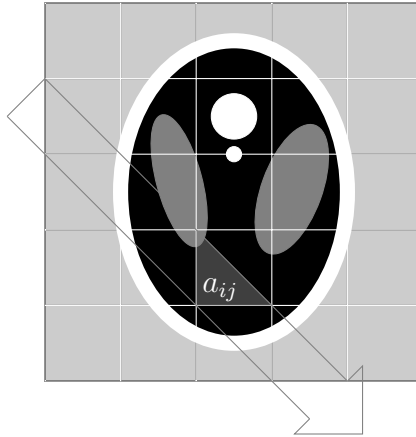Mette Bjerg (zgb585) and Lærke Pedersen (crj405)

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), namely forward projection and back projection and investigate how we may use the high level data-parallel language Futhark for implememting a fast version of this algorithm. The reason for choosing 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 discretizing 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:

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

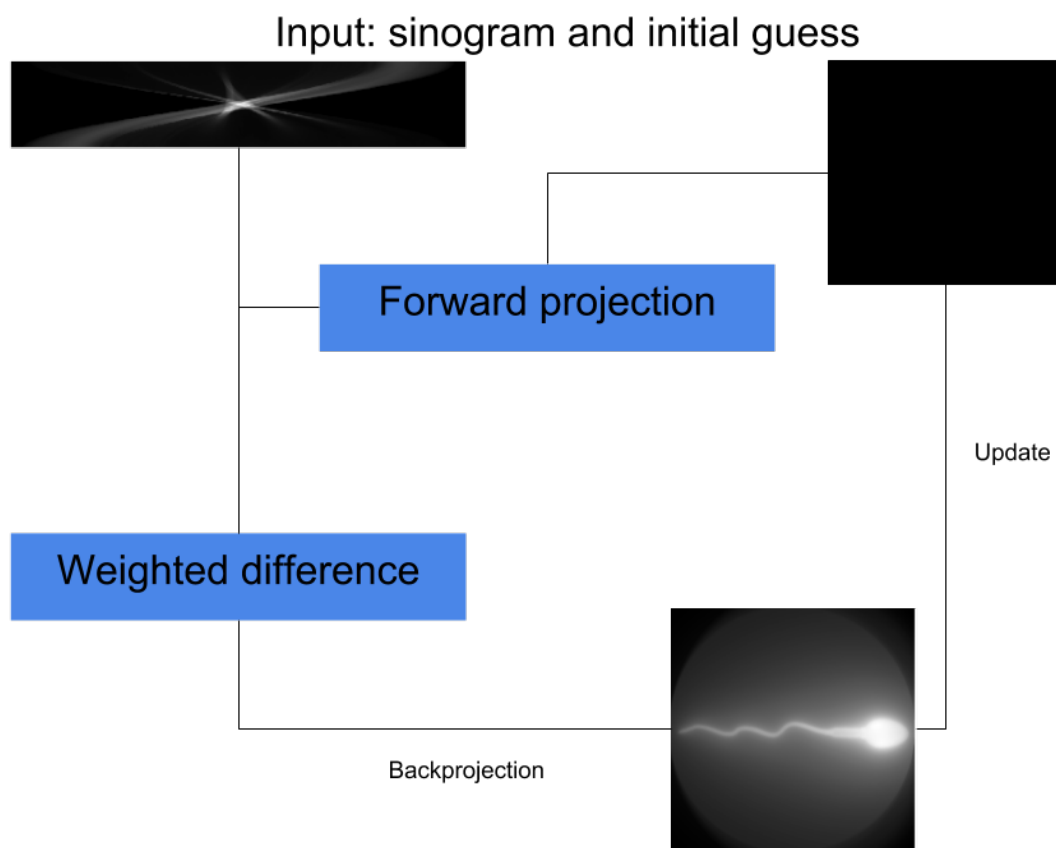where $\|\boldsymbol{x}\|_{\boldsymbol{R}} = \boldsymbol{x}^T\boldsymbol{R}\boldsymbol{x}$.

Figure 2: An illustration of the SIRT algorithm.

The backprojection and foward projection operations are standard operations in many iterative algebraic reconstruction methods and are the bottle necks of the algorithms [2]. 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 $4 \cdot 2 \cdot \lceil \frac{\pi \cdot 4000}{2} \rceil \cdot 4000 \cdot (2 \cdot 4000 - 1) \approx 1 TB$. 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 $3 - 4 GB$. Besides the matrix we also need to store the sinogram data which will take approximately $420 MB$, and the reconstruction taking up around $270 MB$. Hence even for relatively small problems we're looking at an estimated $5 GB$ 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 consumer card, and large problems won't fit on any cards. When we tested our algorithms, we were not able to compute the system matrix on the universitys 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.

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

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

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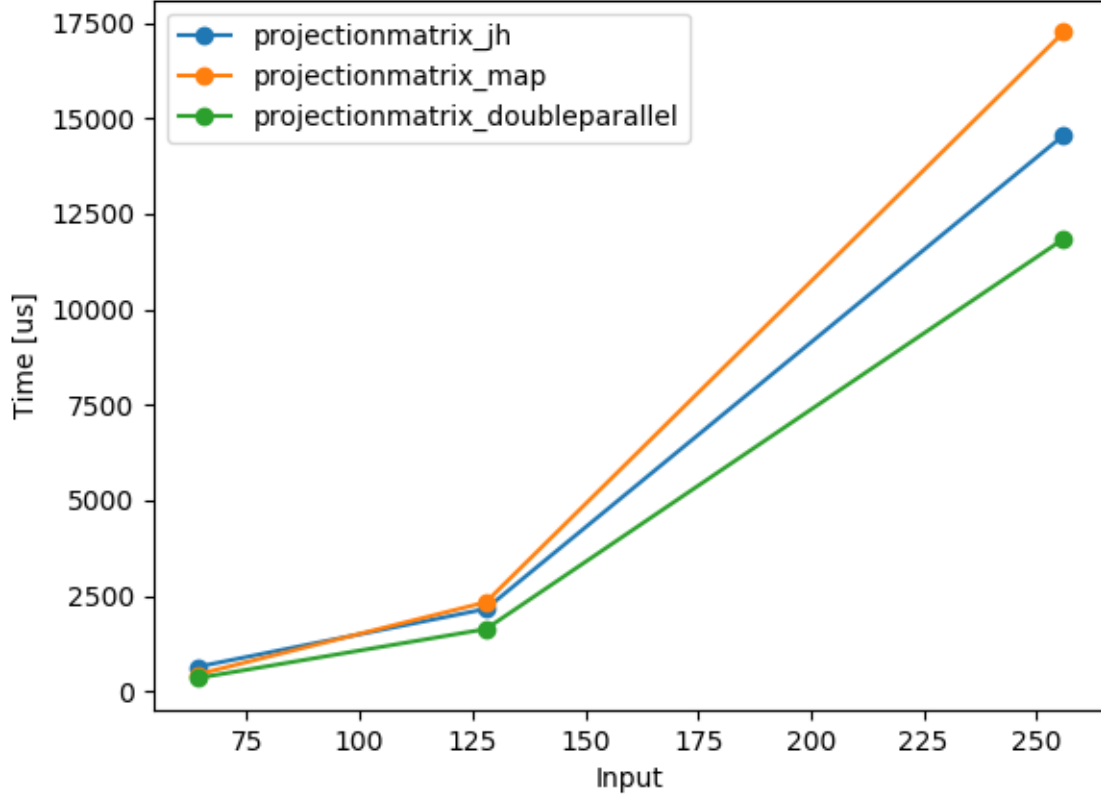
Figure 4: A lcomparison of the projection matrix algorithms run for gridsizes 64 to 256. Since the universities GPUs only have about 3GB of memory this was the largest size we could run for without chunking up the matrix.

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:

```
1  for step = 0; step < steps; step++
2    A = getRays(raysperstep)
3    for ray = 0; ray < raysperstep; row++
4      acc = 0.0
5      for p = 0; p<numpixels; p++
6        acc+= A[ray][p]*image[p]
7      FP[step*raysperstep+ray] = acc
8
9  this can be written in futhark like pseudo code as:
10
11 loop (output, step, raysperstep)
12   let A = getRays raysperstep step
13   let partresult = map (\row -> reduce (+) 0 <| map ( \i -> row[i]*
        vector[i] ) (iota (length row)) ) A
14   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.

```
1  for step = 0; step < steps; step++
2    A = getRays(raysperstep)
3    AT = A.transpose()
4    for p = 0; p<numpixels; p++
5      acc = 0.0
6      for ray = 0; ray<raysperstep; ray++
7        acc+= AT[p][ray]*sinogram[ray]
8      BP[p] += acc
9
10 this can be written in futhark like pseudo code as:
11
12 loop (output, step, raysperstep)
13   let A = getRays raysperstep step
14   let AT = transpose A
15   let partresult = map (\row -> reduce (+) 0 <| map2 (*) row vect )
        AT
16   map2 (+) partresult output
17   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.
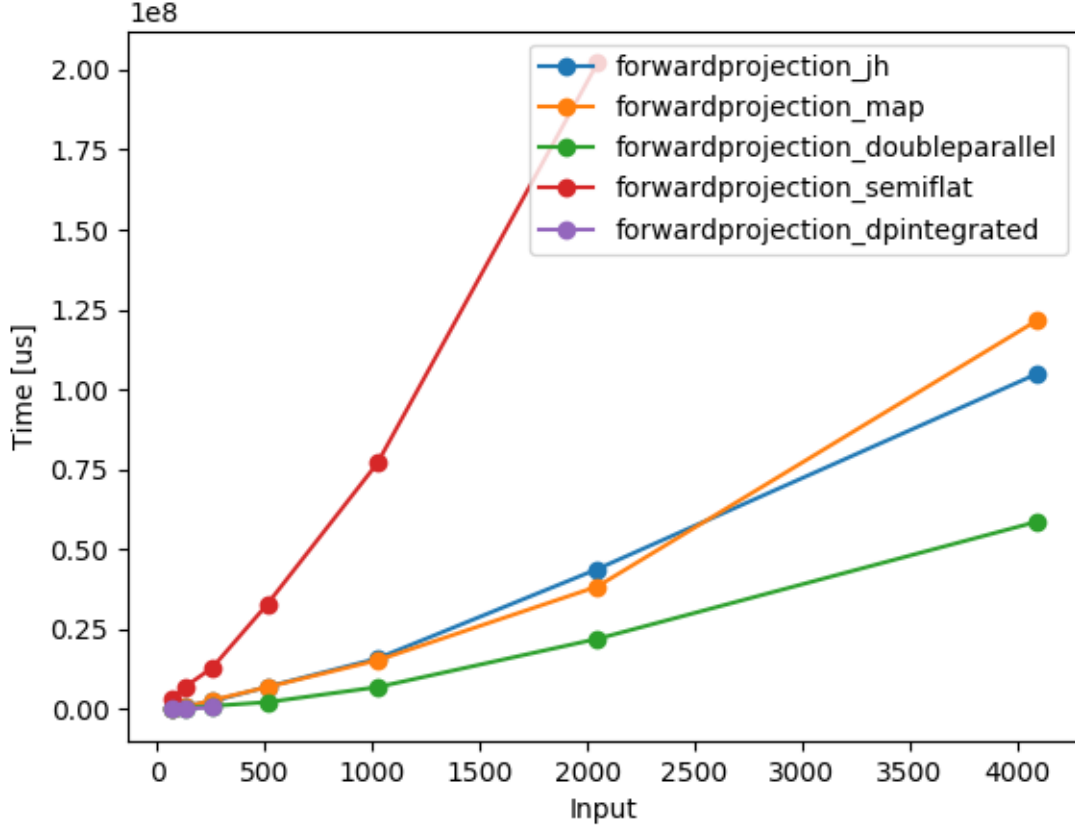
Figure 7: A comparison of the forward projections run with different matrix implementations and the version where the forward projection is integrated in projectiomatrix_doubleparallel. The chunk size was 32. We chose this number to fit a CUDA warp. As expected the semiflat version performed rather badly. The implementation using the double parallel version performed best. The rest of the algorithms are nested, using different projection matrix algorithms inside. We had to run our algorithms for only 30 angles as we got memory errors otherwise. This requires further investigation. In our initial tests our stripmined algorithms ran for all sizes, but perhaps we were competing for space with other groups.
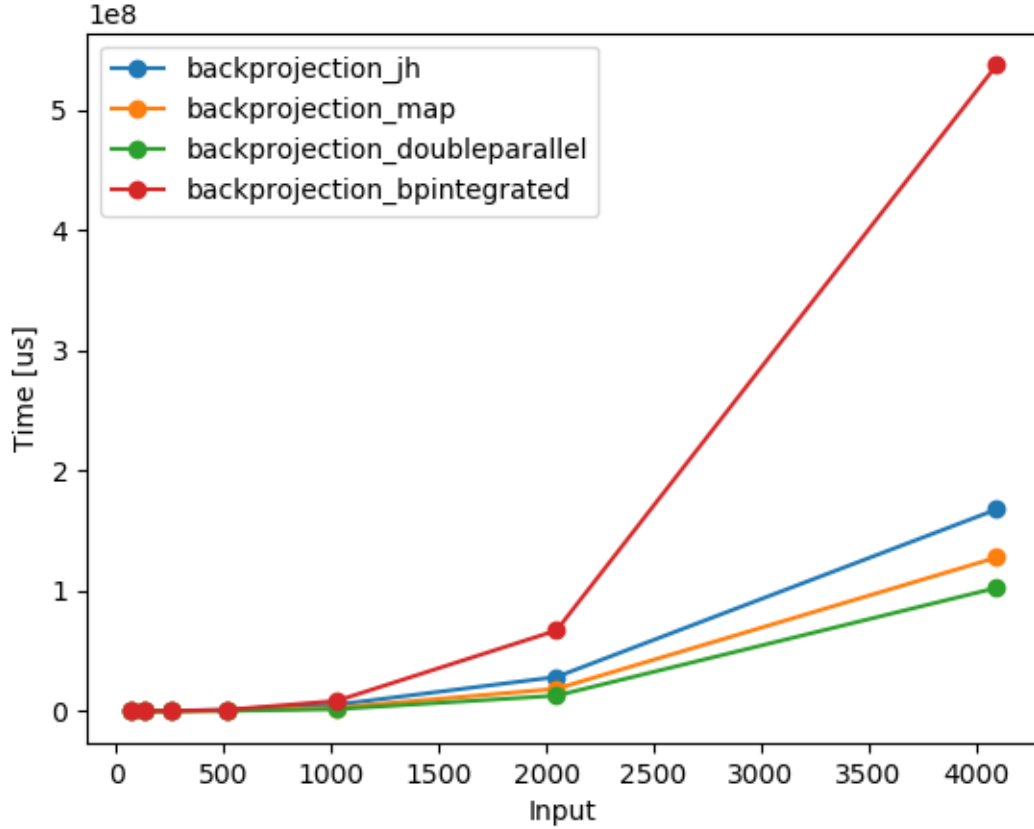
Figure 8: A comparison of the backprojections projections run with different matrix implementations and the version where the forward projection is integrated in projectiomatrix_doubleparallel. The chunk size was 32. The implementation using the double parallel version of the matrix performed best again as expected. The integrated version performed very badly, and it would be interesting to investigate why. We had to run our algorithms for only 30 angles as we got memory errors otherwise.
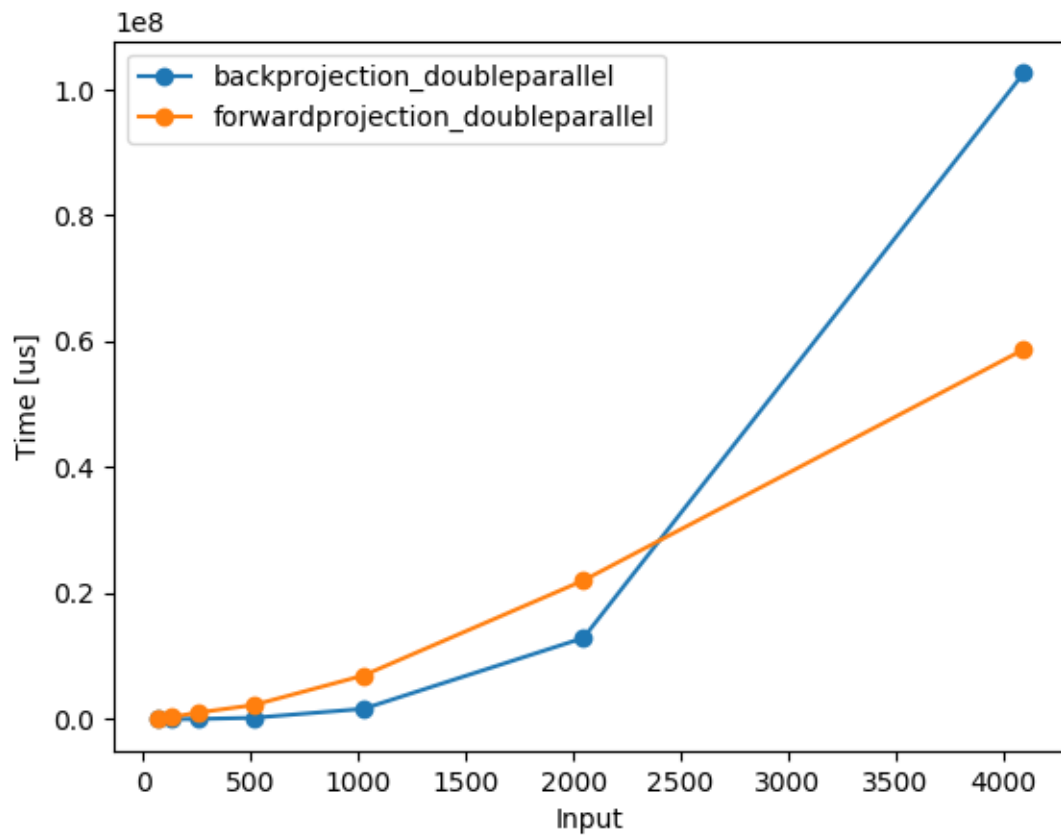
Figure 9: A comparison of backprojection and forward projection. Backprojection seems to be faster for small problem sizes, but scale worse.

5 Comparison to a CUDA implementation

We compared our implementation of forward projection and back projection to implementations from the astra toolbox. However we only compared the algorithms through python, by compiling our algorithms with pyopencl and running them through python and doing the same with the forward and backward rpojection from the atsra toolbox. Ideally the functions should have been compared without the memory overhead.
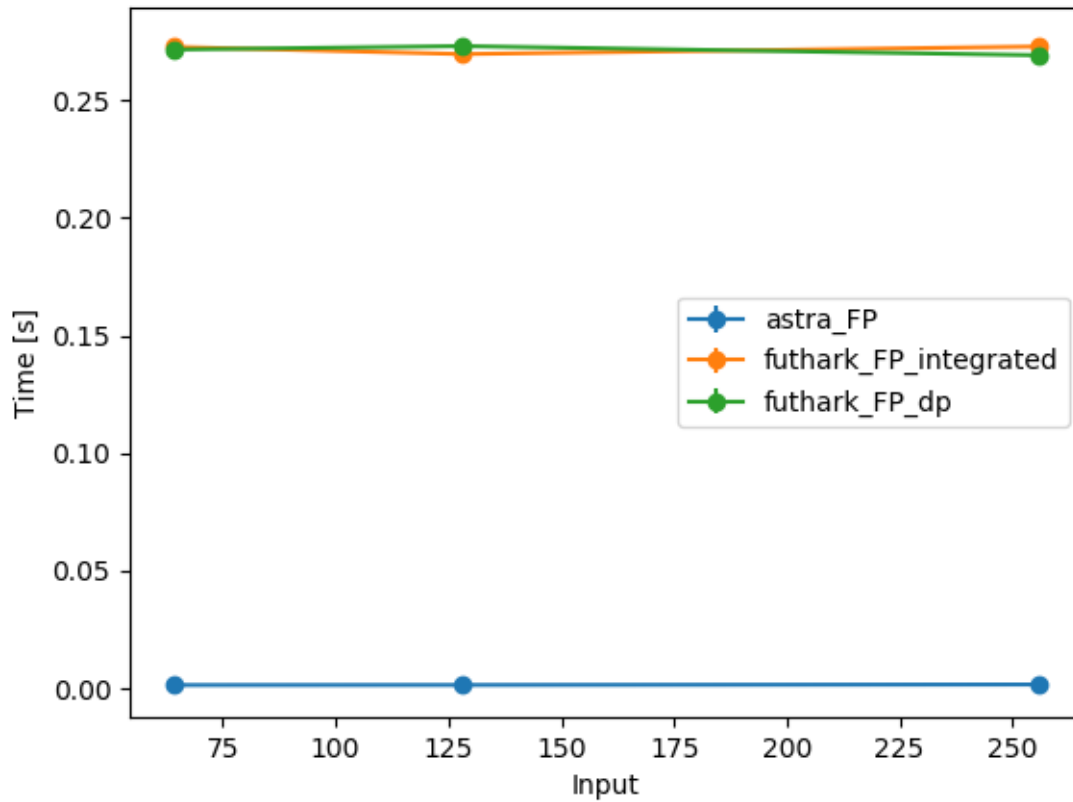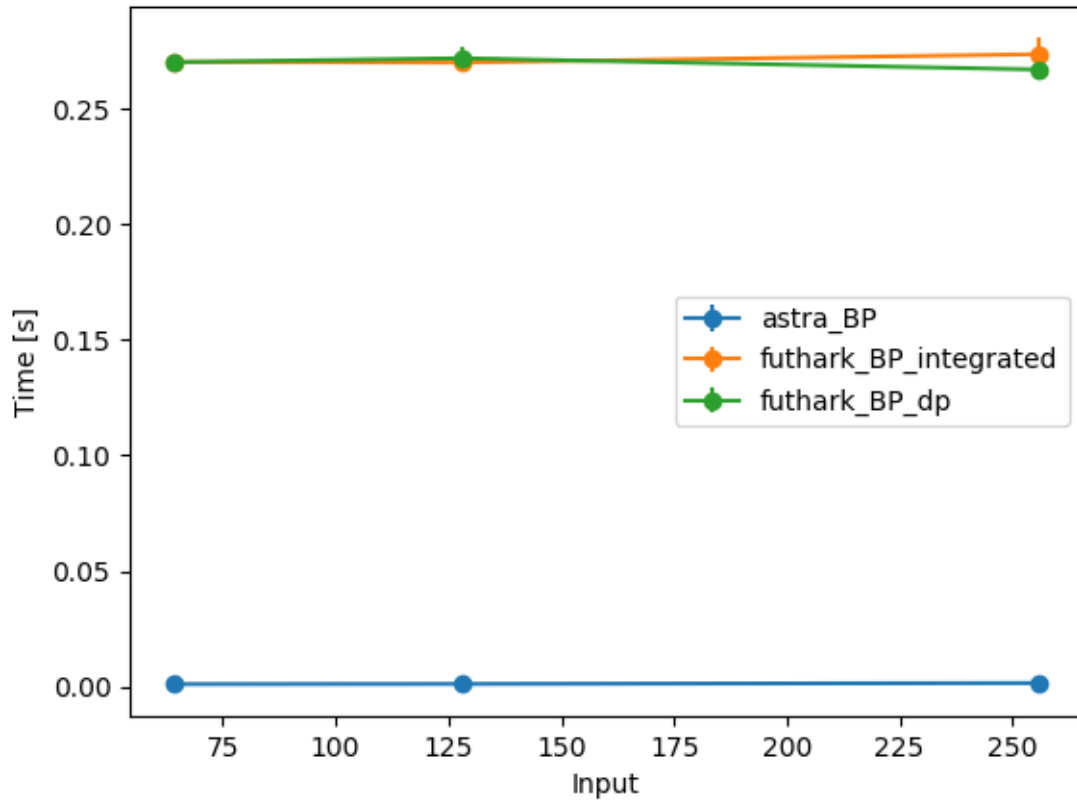
Figure 10: A comparison of the back projections from astra and our fastest futhark algorithms. The futhark algorithms perform much worse, approximately 128 times worse for both forward and backprojection. It would be worth comparing our code with the implemented version from the astra toolbox, to see how they get such a good running time.

6 Code Overview

Our code is primarily comprised of Futhark code, alongside a few python script for data generation, and shell scripts.

6.1 folder structure

- futhark contains the source code files, as well as the files generated during compilation.

- runscripts contains the shell scripts used to run tests and benchmarks

- root folder - the python scripts are located in the root folder, which makes it easier to navigate the folder structure from the python code.

- data and output - data is an, so far, empty folder to store the generated input data, while the output is the same for output data

6.2 Code structure

The actual Futhark code is located in the files projectionlib.fut, matrixlib.fut and linelib.fut. All the forwardprojection and backprojection files contains main functions to call the libraries for testing. The projectionlib contains the actual code for doing the forwardprojections and backprojections, as well as a few limited helper functions. The matrixlib is focused on generating the system matrices, with code for each way of doing this. Finally the linelib contains helper functions for the matrixlib.

7 Conclusion

We set out to make a parallel implementation of forward and back projection for algebraic tomographic reconstruction. We found that most of the time spend in our initial nested versions, using code from a bachelor project for generating the system matrix, was spend on calculating the matrix. We therefore concentrated on improving the code for calcualting the matrix. We managed to get a slight speedup of about 1.3 times compared to the code from the bachelor project by exploiting a second level of parallelism. When we compare a c compiled version of our futhark code with an opencl compiled version we get a speedup of about 1.3. Strangely enough our c compiled code actually runs slightly faster than the opencl compiled version of the code from the bachelor project. It therefore seems that our new version is better both sequentially and in parallel.
We attempted to compare our implementations with those in a python library called astra toolbox with a backend implemented in CUDA. However, we did not time the kernels only, but timed the call from python which has significant overhead. Therefore, it is difficult to compare. The results we got differed on which function was run first, and generally seemed

to vary a lot. To properly time the functions, we would need to go to the astra code an time the kernel only. It seems to perform better than our versions though, and it would be nice to compare the CUDA code with the opencl compiled versions from futhark.

Some of our code may benefit from upcoming block and register tiling in futhark, or streaming possibilities as memory seems to be the biggest limitation. Other possibilities for optimizations could be to consider other datastructures for more sparse storage, or investigate how much precision is needed for good reconstructions. It will also be worth it to look at the branching in the kernels. Flattening does not seem very promising since we already have memory issues and our semiflat versions performed rather badly. Perhaps if the computation of the system matrix was also sparse and flat we would see an improvement.

Overall we were happy to get a slight speedup for the matrix computations, and there is still lots of work that could be done on this project. One thing that needs investigation is why chunking up the matrix does not seem to work as expected as we still ran into memory problems. We look forward to keep working on it with new upcoming features in futhark and to have time to scour the compiled code for further optimizations.

8 Code

8.1 projection lib

```
1  import "futlib/array"
2  import "matrix_lib"
3  open Matrix
4
5  module Projection = {
6
7      --segmented scan with (+) on floats:
8      let sgmSumF32 [n]
9          (flg : [n]i32)
10         (arr : [n]f32) : [n]f32 =
11             let flgs_vals =
12              scan ( \ (f1, x1) (f2,x2) ->
13                      let f = f1 | f2 in
14                      if f2 > 0 then (f, x2)
15                      else (f, x1 + x2) )
16                  (0,0.0f32) (zip flg arr)
17             let (_, vals) = unzip flgs_vals
18             in vals
19
20     -- segmented scan with (+) on ints:
21     let sgmSumI32 [n]
22         (flg : [n]i32)
23         (arr : [n]i32) : [n]i32 =
24             let flgs_vals =
```

```
25                   scan ( \ (i1, x1) (i2,x2) ->
26                          let i = i1 | i2 in
27                          if i2 > 0 then (i, x2)
28                          else (i, x1 + x2) )
29                       (0,0i32) (zip flg arr)
30               let (_, vals) = unzip flgs_vals
31               in vals
32
33       -- unzip for 2D arrays
34       let unzip_d2 (xs: [][](f32,i32)): ([][]f32,[][]i32) =
35           xs |> map unzip
36              |> unzip
37
38       -- step in radix sort
39       let rsort_step [n] (xs: [n](f32, i32, i32), bitn: i32): [n](f32,
           i32, i32) =
40               let (data,rays,pixels) = unzip3 xs
41               let unsigned = map(\p -> u32.i32 p) pixels
42               let bits1 = map (\x -> (i32.u32 (x >> u32.i32 bitn)) &
                    1) unsigned
43               let bits0 = map (1-) bits1
44               let idxs0 = map2 (*) bits0 (scan (+) 0 bits0)
45               let idxs1 = scan (+) 0 bits1
46               let offs  = reduce (+) 0 bits0
47               let idxs1 = map2 (*) bits1 (map (+offs) idxs1)
48               let idxs  = map2 (+) idxs0 idxs1
49               let idxs  = map (\x->x-1) idxs
50               in scatter (copy xs) idxs xs
51
52       -- Radix sort algorithm, ascending
53       let rsort [n] (xs: [n](f32, i32, i32)): [n](f32,i32,i32) =
54               loop (xs) for i < 32 do rsort_step(xs,i)
55
56       -- sparse matrix vector multiplication
57       let spMatVctMult [num_elms] [vct_len] [num_rows]
58           (mat_val : [num_elms](i32,f32))
59           (shp_scn : [num_rows]i32)
60           (vct : [vct_len]f32) : [num_rows]f32 =
61               let len = shp_scn[num_rows-1]
62               let shp_inds =
63                   map (\i -> if i==0 then 0
64                       else unsafe shp_scn[i-1]
65                   ) (iota num_rows)
66               let flags = scatter ( replicate len 0)
67                       shp_inds ( replicate num_rows 1)
68               let prods = map (\(i,x) -> x*(unsafe vct[i])) mat_val
69               let sums = sgmSumF32 flags prods
70               let mat_inds = map (\i -> i-1) shp_scn
71               in map (\i -> unsafe sums[i]) mat_inds
72
73       let notSparseMatMult [num_rows] [num_cols]
```

```
                            (mat_vals : [num_rows][num_cols](f32,i32))
                            (vect : []f32) : [num_rows]f32 =
        map (\row -> reduce (+) 0 <| map (\(v, ind) -> unsafe (if ind
            == -1 then 0.0 else v*vect[ind]) ) row ) mat_vals

    -- not sparse, nested, matrix vector multiplication for
        backprojection, uses a padded version of the matrix
    let notSparseMatMult_back [num_rows] [num_cols]
                            (mat_vals : [num_rows][num_cols]f32)
                            (vect : []f32) : [num_rows]f32 =
        map (\row -> reduce (+) 0 <| map2 (*) row vect ) mat_vals

    -- gets the shape of a matrix - i.e number of entries pr. row
        when mat is in the format [[(d,i),(d,i)...]] where d is data
    -- and i is column index and -1 means no data
    let getshp (matrix: [][](f32,i32)) : []i32 =
            let dp = unzip_d2 matrix
            let flagsforindexes = map(\r -> map(\p -> if p == -1
                then 0 else 1)r)dp.2
            in map(\f -> reduce (+) 0 f)flagsforindexes

    -- helper function to determine if we entered new segment of
        consecutive numbers in an array.
    let isnewsegment (i: i32) (arr: []i32) : bool =
                        i!=0 && (unsafe arr[i]) != (unsafe arr[i-1])

    let backprojection_semiflat  (rays : []f32)
                        (angles : []f32)
                        (projections : []f32)
                        (gridsize: i32) : []f32=
            let halfsize = r32(gridsize)/2
            let entrypoints = convert2entry angles rays halfsize
            let intersections = map (\(p,sc) -> (lengths gridsize sc
                .1 sc.2 p)) entrypoints
            --convert to triples (data,ray,pixel)
            let triples_tmp = flatten(map(\i -> map(\v -> (v.1, i, v
                .2))(unsafe intersections[i])) (iota (length
                intersections)))
            -- remove none values
            let triples = filter (\x -> x.3 != -1) triples_tmp
            -- sort by pixel indexes
            let pixelsorted = rsort triples
            -- slplit int three arrays in order to use pixels only
                for shp
            let (data,rays,pixels) = unzip3 pixelsorted
            let num_pixels = length pixels
            -- contains sum of values where a row ends since columns
                 will be rows.
            let shp_scn_tmp = map (\i -> if (i == num_pixels || (
                isnewsegment i pixels)) then i else 0) (iota (
                num_pixels+1))
```

```
113                    let shp_scn = filter (\p -> p != 0) shp_scn_tmp
114                    let values = map(\x-> (x.2,x.1))pixelsorted
115                    in spMatVctMult values shp_scn projections
116
117        -- pads and transposes the matrix, nested, will perform better
               when tiling is improved in futhark
118        let trans_map [m] [n]
119                    (matrix : [m][n](f32, i32)) (gridsize: i32): [][]
                           f32 =
120                    let rs = gridsize*gridsize
121                    let padded = map (\row -> let (vals, inds) = (unzip row)
                           in scatter (replicate rs 0.0f32) inds vals ) matrix
122                    in transpose padded
123
124        -- backprojection nested map version.
125        let backprojection_map  (angles : []f32)
126                           (rays : []f32)
127                           (projections : []f32)
128                           (gridsize: i32)
129                           (stepSize : i32) : []f32=
130                    let halfsize = r32(gridsize)/2
131                    let entrypoints = convert2entry angles rays halfsize
132                    let totalLen = (length entrypoints)
133                    let runLen = (totalLen/stepSize)
134                    -- result array
135                    let backmat = replicate (gridsize*gridsize) 0.0f32
136                    -- stripmined, sequential outer loop, mapped inner
137                    let (backmat, _, _, _, _, _, _) =
138                        loop (output, run, runLen, stepSize, gridsize,
                               entrypoints, totalLen) = (backmat, 0, runLen,
                               stepSize, gridsize, entrypoints, totalLen)
139                        while ( run < runLen ) do
140                            -- if the number of entrypoints doesn't line
                                   perfectly up with the stepsize
141                            let step = if (run+1)*stepSize >= totalLen then
                                   totalLen - run*stepSize else stepSize
142                            -- calc part of matrix, stepSize rows
143                            let partmatrix = map (\s -> unsafe (lengths_map
                                   gridsize (entrypoints[run*stepSize + s].2).1
                                   (entrypoints[run*stepSize + s].2).2
                                   entrypoints[run*stepSize + s].1 )) (iota step
                                   )
144                            -- transpose
145                            let transmat = trans_map partmatrix gridsize
146                            -- mult
147                            let partresult = (notSparseMatMult_back transmat
                                   projections[(run*stepSize) : (run*stepSize +
                                   step)])
148                            -- add
149                            let result = (map2 (+) partresult output)
```

```
150                      in (result, run+1, runLen, stepSize, gridsize,
                             entrypoints, totalLen)
151              in backmat
152
153      let backprojection_jh (angles : []f32)
154                      (rays : []f32)
155                      (projections : []f32)
156                      (gridsize: i32)
157                      (stepSize : i32) : []f32=
158              let halfsize = r32(gridsize)/2
159              let entrypoints = convert2entry angles rays halfsize
160              let totalLen = (length entrypoints)
161              let runLen = (totalLen/stepSize)
162              -- result array
163              let backmat = replicate (gridsize*gridsize) 0.0f32
164              -- stripmined, sequential outer loop, mapped inner
165              let (backmat, _, _, _, _, _, _) =
166                  loop (output, run, runLen, stepSize, gridsize,
                         entrypoints, totalLen) = (backmat, 0, runLen,
                         stepSize, gridsize, entrypoints, totalLen)
167                  while ( run < runLen ) do
168                      -- if the number of entrypoints doesn't line
                             perfectly up with the stepsize
169                      let step = if (run+1)*stepSize >= totalLen then
                             totalLen - run*stepSize else stepSize
170                      -- calc part of matrix, stepSize rows
171                      let partmatrix = map (\s -> unsafe (lengths
                             gridsize (entrypoints[run*stepSize + s].2).1
                             (entrypoints[run*stepSize + s].2).2
                             entrypoints[run*stepSize + s].1 )) (iota step
                             )
172                      -- transpose
173                      let transmat = trans_map partmatrix gridsize
174                      -- mult
175                      let partresult = (notSparseMatMult_back transmat
                             projections[(run*stepSize) : (run*stepSize +
                             step)])
176                      -- add
177                      let result = (map2 (+) partresult output)
178                      in (result, run+1, runLen, stepSize, gridsize,
                             entrypoints, totalLen)
179              in backmat
180
181      let backprojection_doubleparallel  (angles : []f32)
182                      (rays : []f32)
183                      (projections : []f32)
184                      (gridsize: i32)
185                      (stepSize : i32) : []f32=
186              let halfsize = r32(gridsize)/2
187              let entryexitpoints =  convert2entryexit angles rays
                     halfsize
```

21

```
188        let totalLen = (length entryexitpoints)
189        let runLen = (totalLen/stepSize)
190        -- result array
191        let backmat = replicate (gridsize*gridsize) 0.0f32
192        -- stripmined, sequential outer loop, mapped inner
193        let (backmat, _, _, _, _, _, _) =
194            loop (output, run, runLen, stepSize, gridsize,
                entryexitpoints, totalLen) = (backmat, 0, runLen,
                stepSize, gridsize, entryexitpoints, totalLen)
195        while ( run < runLen ) do
196            -- if the number of entrypoints doesn't line
                perfectly up with the stepsize
197            let step = if (run+1)*stepSize >= totalLen then
                totalLen - run*stepSize else stepSize
198            -- calc part of matrix, stepSize rows
199            let halfgridsize = gridsize/2
200
201            let partmatrix = map(\(ent,ext) -> (flatten(map
                (\i ->
202                    calculate_weight ent ext i gridsize
203                )((-halfgridsize)...(halfgridsize-1))))) (
                    entryexitpoints[(run*stepSize) : (run*
                    stepSize + step)])
204            -- transpose
205            let transmat = trans_map partmatrix gridsize
206            -- mult
207            let partresult = (notSparseMatMult_back transmat
                projections[(run*stepSize) : (run*stepSize +
                step)])
208            -- add
209            let result = (map2 (+) partresult output)
210            in (result, run+1, runLen, stepSize, gridsize,
                entryexitpoints, totalLen)
211        in backmat
212
213    let forwardprojection_doubleparallel (angles : []f32)
214                    (rays : []f32)
215                    (voxels : []f32)
216                    (stepSize : i32) : []f32 =
217        let gridsize = t32(f32.sqrt(r32((length voxels))))
218        let halfgridsize = gridsize/2
219        let entryexitpoints =  convert2entryexit angles rays (
            r32(halfgridsize))
220        let totalLen =  (length entryexitpoints)
221        let runLen = (totalLen/stepSize)
222        let testmat = [0f32]
223        let (testmat, _, _, _, _, _, _) =
224            loop (output, run, runLen, stepSize, gridsize,
                entryexitpoints, totalLen) = (testmat, 0, runLen,
                stepSize, gridsize, entryexitpoints, totalLen)
225        while ( run < runLen ) do
```

```
226              let step = if (run+1)*stepSize >= totalLen then
                    totalLen - run*stepSize else stepSize
227              let partmatrix = map(\(ent,ext) -> (flatten(map
                    (\i ->
228                      calculate_weight ent ext i gridsize
229                  )(-halfgridsize...halfgridsize-1)))) (
                        entryexitpoints[run*stepSize:run*
                        stepSize+step])
230              let partresult = notSparseMatMult partmatrix
                    voxels
231              in (output++partresult, run+1, runLen, stepSize,
                    gridsize, entryexitpoints, totalLen)
232          in (tail testmat)
233
234    let forwardprojection_jh [r][a][n] (angles : [a]f32)
235                    (rays : [r]f32)
236                    (voxels : [n]f32)
237                    (stepSize : i32) : []f32 =
238          let gridsize = t32(f32.sqrt(r32((length voxels))))
239          let halfsize = r32(gridsize)/2
240          let entrypoints = convert2entry angles rays halfsize
241          let totalLen = (length entrypoints)
242          -- let runLen = if (totalLen/stepSize) == 0 then 1 else
                (totalLen/stepSize)
243          let runLen = (totalLen/stepSize)
244          let testmat = [0f32]
245          let (testmat, _, _, _, _, _, _) =
246              loop (output, run, runLen, stepSize, gridsize,
                    entrypoints, totalLen) = (testmat, 0, runLen,
                    stepSize, gridsize, entrypoints, totalLen)
247              while ( run < runLen ) do
248                  let step = if (run+1)*stepSize >= totalLen then
                        totalLen - run*stepSize else stepSize
249                  let partmatrix = map (\s -> unsafe (lengths
                        gridsize (entrypoints[run*stepSize + s].2).1
                        (entrypoints[run*stepSize + s].2).2
                        entrypoints[run*stepSize + s].1 )) (iota step
                        )
250                  let partresult = notSparseMatMult partmatrix
                        voxels
251                  in (output++partresult, run+1, runLen, stepSize,
                        gridsize, entrypoints, totalLen)
252              in (tail testmat)
253
254    let forwardprojection_map [r][a][n] (angles : [a]f32)
255                    (rays : [r]f32)
256                    (voxels : [n]f32)
257                    (stepSize : i32) : []f32 =
258          let gridsize = t32(f32.sqrt(r32((length voxels))))
259          let halfsize = r32(gridsize)/2
260          let entrypoints = convert2entry angles rays halfsize
```

```
261              let totalLen = (length entrypoints)
262              let runLen = (totalLen/stepSize)
263              let testmat = [0f32]
264              -- stripmined, sequential outer loop, mapped inner
265              let (testmat, _, _, _, _, _, _) =
266                  loop (output, run, runLen, stepSize, gridsize,
                          entrypoints, totalLen) = (testmat, 0, runLen,
                          stepSize, gridsize, entrypoints, totalLen)
267                  while ( run < runLen ) do
268                      -- if the number of entrypoints doesn't line
                          perfectly up with the stepsize
269                      let step = if (run+1)*stepSize >= totalLen then
                              totalLen - run*stepSize else stepSize
270                      -- calc part of matrix, stepSize rows
271                      let partmatrix = map (\s -> unsafe (lengths_map
                              gridsize (entrypoints[run*stepSize + s].2).1
                              (entrypoints[run*stepSize + s].2).2
                              entrypoints[run*stepSize + s].1 )) (iota step
                              )
272                      let partresult = notSparseMatMult partmatrix
                              voxels
273                      in (output++partresult, run+1, runLen, stepSize,
                              gridsize, entrypoints, totalLen)
274              in (tail testmat)
275
276     let forwardprojection_semiflat [r][a][n] (angles : [a]f32)
277                      (rays : [r]f32)
278                      (voxels : [n]f32)
279                      (stepSize : i32) : []f32 =
280              let gridsize = t32(f32.sqrt(r32((length voxels))))
281              let halfsize = r32(gridsize)/2
282              let entrypoints = convert2entry angles rays halfsize
283              let totalLen = (length entrypoints)
284              -- let runLen = if (totalLen/stepSize) == 0 then 1 else
                  (totalLen/stepSize)
285              let runLen = (totalLen/stepSize)
286              let testmat = [0f32]
287              let (testmat, _, _, _, _, _, _) =
288                  loop (output, run, runLen, stepSize, gridsize,
                          entrypoints, totalLen) = (testmat, 0, runLen,
                          stepSize, gridsize, entrypoints, totalLen)
289                  while ( run < runLen ) do
290                      let step = if (run+1)*stepSize >= totalLen then
                              totalLen - run*stepSize else stepSize
291                      let intersections = map (\(p,sc) -> (lengths
                              gridsize sc.1 sc.2 p)) entrypoints[(run*
                              stepSize) : (run*stepSize + step)]
292                      let shp = getshp intersections
293                      let shp_scn = scan (+) 0 shp
294                      let values_tmp = flatten(map(\r -> map(\(d,p)->(
                              p,d))r)intersections)
```

```
295          let values = filter (\x -> x.1 != -1) values_tmp
296          let partresult = spMatVctMult values shp_scn
                 voxels
297          in (output++partresult, run+1, runLen, stepSize,
                 gridsize, entrypoints, totalLen)
298      in (tail testmat)
299
300  let forwardprojection_integrated [r][a][n] (angles : [a]f32)
301                    (rays : [r]f32)
302                    (voxels : [n]f32)
303                    (stepSize : i32) : []f32 =
304      let gridsize = t32(f32.sqrt(r32((length voxels))))
305      let halfsize = gridsize/2
306      let entryexitpoints =  convert2entryexit angles rays (
             r32(halfsize))
307      let totalLen = (length entryexitpoints)
308      -- let runLen = if (totalLen/stepSize) == 0 then 1 else
             (totalLen/stepSize)
309      let runLen = (totalLen/stepSize)
310      let testmat = [0f32]
311      let (testmat, _, _, _, _, _, _) =
312          loop (output, run, runLen, stepSize, gridsize,
                 entryexitpoints, totalLen) = (testmat, 0, runLen,
                  stepSize, gridsize, entryexitpoints, totalLen)
313          while ( run < runLen ) do
314              let step = if (run+1)*stepSize >= totalLen then
                     totalLen - run*stepSize else stepSize
315              let partresult = map(\(ent,ext) -> (reduce (+) 0
                     (flatten(map (\i ->
316                          calculate_fp_val ent ext i gridsize
                             voxels
317                      )((-halfsize)...(halfsize-1)))))) 
                         entryexitpoints[run*stepSize:run*
                         stepSize+step]
318              in (output++partresult, run+1, runLen, stepSize,
                     gridsize, entryexitpoints, totalLen)
319      in (tail testmat)
320
321
322  let backprojection_integrated  (angles : []f32)
323                    (rays : []f32)
324                    (projections : []f32)
325                    (gridsize: i32)
326                    (stepSize : i32) : []f32=
327      let halfsize = gridsize/2
328      let entryexitpoints =  convert2entryexit angles rays (
             r32(halfsize))
329      let totalLen = (length entryexitpoints)
330      let runLen = (totalLen/stepSize)
331      -- result array
332      let backmat = replicate (gridsize*gridsize) 0.0f32
```

```
333                      -- stripmined, sequential outer loop, mapped inner
334                      let (backmat, _, _, _, _, _, _) =
335                          loop (output, run, runLen, stepSize, gridsize,
                                entryexitpoints, totalLen) = (backmat, 0, runLen,
                                stepSize, gridsize, entryexitpoints, totalLen)
336                          while ( run < runLen ) do
337                              -- if the number of entrypoints doesn't line
                                    perfectly up with the stepsize
338                              let step = if (run+1)*stepSize >= totalLen then
                                    totalLen - run*stepSize else stepSize
339                              let partmatresult = map(\j ->
340                                  (flatten(map (\i ->
341                                      calculate_bp_val (unsafe
                                            entryexitpoints[run*stepSize+j]).1 (
                                            unsafe entryexitpoints[run*stepSize+
                                            j]).2 i gridsize (unsafe projections
                                            [j])
342                                  )((-halfsize)...(halfsize-1))))) (iota step)
343                              let transp = trans_map partmatresult gridsize
344                              let partresult = map (\row -> reduce (+) 0 row)
                                    transp
345                              -- add
346                              let result = (map2 (+) partresult output)
347                              in (result, run+1, runLen, stepSize, gridsize,
                                    entryexitpoints, totalLen)
348                      in backmat
349  }
```

8.2   matrix lib

```
1  import "line_lib"
2  open Lines
3  module Matrix =
4  {
5      let calculate_fp_val(ent: point)
6                  (ext: point)
7                  (i: i32)
8                  (N: i32)
9                  (pixels: []f32) : []f32 =
10         let Nhalf = N/2
11         -- handle all lines as slope < 1 reverse the others
12         let slope = (ext.2 - ent.2)/(ext.1 - ent.1)
13         let reverse = f32.abs(slope) > 1
14         let gridentry = if reverse then (if slope < 0 then (-ent.2,
               ent.1) else (-ext.2,ext.1)) else ent
15         let k = if reverse then (-1/slope) else slope
16
17         --- calculate stuff
18         let ymin = k*(r32(i) - gridentry.1) + gridentry.2 + r32(
               Nhalf)
19         let yplus = k*(r32(i) + 1 - gridentry.1) + gridentry.2 +
```

```
                r32(Nhalf)
20          let Ypixmin = t32(f32.floor(ymin))
21          let Ypixplus = t32(f32.floor(yplus))
22          let baselength = f32.sqrt(1+k*k)
23          -- in [(baselength,Ypixmin),(baselength,Ypixplus)]
24          let Ypixmax = i32.max Ypixmin Ypixplus
25          let ydiff = yplus - ymin
26          let yminfact = (r32(Ypixmax) - ymin)/ydiff
27          let yplusfact = (yplus - r32(Ypixmax))/ydiff
28          let iindex = i+Nhalf
29          -- index calculated wrong for reversed lines i think
30          let pixmin = if reverse then (N-iindex-1)*N+Ypixmin else
                iindex+Ypixmin*N
31          let pixplus = if reverse then (N-iindex-1)*N+Ypixplus else
                iindex+Ypixplus*N
32          let lymin = yminfact*baselength
33          let lyplus = yplusfact*baselength
34          let min = if (pixmin >= 0 && pixmin < N ** 2) then
35              (if Ypixmin == Ypixplus then (unsafe baselength*pixels
                    [pixmin]) else (unsafe lymin*pixels[pixmin]))
36              else 0
37          let plus = if (pixplus >= 0 && pixplus < N ** 2) then
38              (if Ypixmin == Ypixplus then 0 else (unsafe lyplus*
                    pixels[pixmin]))
39              else 0
40          in [min,plus]
41
42      let calculate_bp_val(ent: point)
43                  (ext: point)
44                  (i: i32)
45                  (N: i32)
46                  (proj_val: f32) : [](f32,i32) =
47          let Nhalf = N/2
48          -- handle all lines as slope < 1 reverse the others
49          let slope = (ext.2 - ent.2)/(ext.1 - ent.1)
50          let reverse = f32.abs(slope) > 1
51          let gridentry = if reverse then (if slope < 0 then (-ent.2,
                ent.1) else (-ext.2,ext.1)) else ent
52          let k = if reverse then (-1/slope) else slope
53
54          --- calculate stuff
55          let ymin = k*(r32(i) - gridentry.1) + gridentry.2 + r32(
                Nhalf)
56          let yplus = k*(r32(i) + 1 - gridentry.1) + gridentry.2 +
                r32(Nhalf)
57          let Ypixmin = t32(f32.floor(ymin))
58          let Ypixplus = t32(f32.floor(yplus))
59          let baselength = f32.sqrt(1+k*k)
60          -- in [(baselength,Ypixmin),(baselength,Ypixplus)]
61          let Ypixmax = i32.max Ypixmin Ypixplus
62          let ydiff = yplus - ymin
```

27

```
63            let yminfact = (r32(Ypixmax) - ymin)/ydiff
64            let yplusfact = (yplus - r32(Ypixmax))/ydiff
65            let iindex = i+Nhalf
66            -- index calculated wrong for reversed lines i think
67            let pixmin = if reverse then (N-iindex-1)*N+Ypixmin else
                  iindex+Ypixmin*N
68            let pixplus = if reverse then (N-iindex-1)*N+Ypixplus else
                  iindex+Ypixplus*N
69            let lymin = yminfact*baselength
70            let lyplus = yplusfact*baselength
71            let min = if (pixmin >= 0 && pixmin < N ** 2) then
72                  (if Ypixmin == Ypixplus then ((baselength*proj_val),
                      pixmin) else (lymin*proj_val,pixmin))
73                  else (-1f32,-1i32)
74            let plus = if (pixplus >= 0 && pixplus < N ** 2) then
75                  (if Ypixmin == Ypixplus then (-1f32,-1i32) else ((
                      lyplus*proj_val), pixmin))
76                  else (-1f32,-1i32)
77          in [min,plus]
78
79
80      --- DOUBLE PARALLEL
81      -- function which computes the weight of pixels in grid_column
          for ray with entry/exit p
82      let calculate_weight(ent: point)
83              (ext: point)
84              (i: i32)
85              (N: i32) : [](f32,i32) =
86          let Nhalf = N/2
87          -- handle all lines as slope < 1 reverse the others
88          let slope = (ext.2 - ent.2)/(ext.1 - ent.1)
89          let reverse = f32.abs(slope) > 1
90          let gridentry = if reverse then (if slope < 0 then (-ent.2,
                  ent.1) else (-ext.2,ext.1)) else ent
91          let k = if reverse then (-1/slope) else slope
92
93          --- calculate stuff
94          let ymin = k*(r32(i) - gridentry.1) + gridentry.2 + r32(
                  Nhalf)
95          let yplus = k*(r32(i) + 1 - gridentry.1) + gridentry.2 +
                  r32(Nhalf)
96          let Ypixmin = t32(f32.floor(ymin))
97          let Ypixplus = t32(f32.floor(yplus))
98          let baselength = f32.sqrt(1+k*k)
99          -- in [(baselength,Ypixmin),(baselength,Ypixplus)]
100         let Ypixmax = i32.max Ypixmin Ypixplus
101         let ydiff = yplus - ymin
102         let yminfact = (r32(Ypixmax) - ymin)/ydiff
103         let yplusfact = (yplus - r32(Ypixmax))/ydiff
104         let lymin = yminfact*baselength
105         let lyplus = yplusfact*baselength
```

```
106            let iindex = i+Nhalf
107            -- index calculated wrong for reversed lines i think
108            let pixmin = if reverse then (N-iindex-1)*N+Ypixmin else
                   iindex+Ypixmin*N
109            let pixplus = if reverse then (N-iindex-1)*N+Ypixplus else
                   iindex+Ypixplus*N
110            let min = if (pixmin >= 0 && pixmin < N ** 2) then
111                (if Ypixmin == Ypixplus then (baselength,pixmin) else
                     (lymin,pixmin))
112                else (-1f32,-1i32)
113            let plus = if (pixplus >= 0 && pixplus < N ** 2) then
114                (if Ypixmin == Ypixplus then (-1f32,-1i32) else (
                     lyplus,pixplus))
115                else (-1f32,-1i32)
116            in [min,plus]
117
118        -- assuming  gridsize even
119        let weights_doublepar(angles: []f32) (rays: []f32) (gridsize:
            i32): [][](f32,i32) =
120            let halfgridsize = gridsize/2
121            let entryexitpoints =  convert2entryexit angles rays (r32(
                   halfgridsize))
122            in map(\(ent,ext) -> (flatten(map (\i ->
123                       calculate_weight ent ext i gridsize
124                  )((-halfgridsize)...(halfgridsize-1)))))
                       entryexitpoints
125
126        --- JH VERSION
127        let lengths     (grid_size: i32)
128                        (sint: f32)
129                        (cost: f32)
130                        (entry_point: point): [](f32, i32) =
131
132            let horizontal = cost == 0
133            let vertical = f32.abs(cost) == 1
134            let slope = cost/(-sint) -- tan(x+90) = -cot(x) = slope
                   since the angles ar ethe normals of the line
135
136            let size = r32(grid_size)
137            let halfsize = size/2.0f32
138
139            let A = replicate (t32(size*2f32-1f32)) (-1f32, -1)
140
141            let y_step_dir = if slope < 0f32 then -1f32 else 1f32
142            let anchorX = f32.floor(entry_point.1) + 1f32
143            let anchorY = if y_step_dir == -1f32
144                then f32.ceil(entry_point.2) - 1f32
145                else f32.floor(entry_point.2) + 1f32
146
147          let (A, _, _, _, _) =
148            loop (A, focusPoint, anchorX, anchorY, write_index) = (A,
```

```
                        entry_point , anchorX , anchorY , 0)
149             while ( isInGrid halfsize y_step_dir focusPoint ) do
150                --compute index of pixel in array by computing x
                     component and y component if
151                --center was at bottom left corner (add halfsize), and
                     add them multiplying y_comp by size
152                let y_floor = f32.floor(halfsize+focusPoint.2)
153                let y_comp =
154                  if (y_step_dir == -1f32 && focusPoint.2 - f32.floor(
                       focusPoint.2) == 0f32)
155                  then y_floor - 1f32
156                  else y_floor
157                let x_comp= f32.floor(halfsize+focusPoint.1)
158                let index = t32(x_comp+size*y_comp)
159
160                --compute the distances using the difference travelled
                     along an axis to the
161                --next whole number and the slope or inverse slope
162                let dy = if vertical then 1f32 else if horizontal then 0
                     f32 else (anchorX-focusPoint.1)*slope
163                let dx = if vertical then 0f32 else if horizontal then 1
                     f32 else (anchorY-focusPoint.2)*(1/slope)
164                let p_anchor_x = (anchorX, focusPoint.2+dy)
165                let p_anchor_y = (focusPoint.1+dx, anchorY)
166
167                let dist_p_x = distance focusPoint p_anchor_x
168                let dist_p_y = distance focusPoint p_anchor_y
169
170                in
171                  if horizontal then
172                    unsafe let A[write_index] = (dist_p_x, index)
173                    in (A, p_anchor_x, anchorX + 1f32, anchorY,
                         write_index+1)
174                  else if vertical then
175                    unsafe let A[write_index] = (dist_p_y, index)
176                    in (A, p_anchor_y, anchorX, anchorY + y_step_dir,
                         write_index+1)
177                  else
178                  if (f32.abs(dist_p_x - dist_p_y) > 0.000000001f32)
179                  then
180                    if ( dist_p_x < dist_p_y )
181                    then
182                      unsafe let A[write_index] = (dist_p_x, index)
183                      in (A, p_anchor_x, anchorX + 1f32, anchorY,
                           write_index+1)
184                    else
185                      unsafe let A[write_index] = (dist_p_y, index)
186                      in (A, p_anchor_y, anchorX, anchorY + y_step_dir,
                           write_index+1)
187                  else
188                      unsafe let A[write_index] = (dist_p_x, index)
```

```
189                    in (A, p_anchor_x, anchorX + 1f32, anchorY +
                           y_step_dir, write_index+1)
190          in A
191
192      let weights_jh   (angles: []f32)
193                       (rays: []f32)
194                       (gridsize: i32) : [][](f32,i32) =
195          let halfsize = r32(gridsize)/2
196          let entrypoints = convert2entry angles rays halfsize
197          in map (\(p,sc) -> (lengths gridsize sc.1 sc.2 p))
                 entrypoints
198
199      ----------------------MAP
             ------------------------------------------------------------
200      let index (focusPoint: point) (halfsize: f32) (y_step_dir): i32
            =
201          let y_floor = f32.floor(halfsize+focusPoint.2)
202          let y_comp =
203              if (y_step_dir == -1f32 && focusPoint.2 - f32.floor(
                     focusPoint.2) == 0f32)
204              then y_floor - 1f32
205              else y_floor
206          let x_comp= f32.floor(halfsize+focusPoint.1)
207          in t32(x_comp+(halfsize*2f32)*y_comp)
208
209       -- let nextpointonline (vertical: bool) (horizontal: bool) (
             anchorX: point) (anchorY: point) (slope: f32) (focusPoint:
             point): point or array of points and lengths
210      let nextpointonline (slope: f32) (vertical: bool) (focusPoint:
            point): point =
211          let y_step_dir = if slope < 0f32 then -1f32 else 1f32
212          let anchorX = if vertical then focusPoint.1 else f32.floor(
                 focusPoint.1) + 1f32
213          let anchorY = if slope == 0 then focusPoint.2 else if
                 y_step_dir == -1f32
214              then f32.ceil(focusPoint.2) - 1f32
215              else f32.floor(focusPoint.2) + 1f32
216          let dy = if slope == 1 then 1f32 else if slope == 0 then 0
                 f32 else (anchorX-focusPoint.1)*slope
217          let dx = if slope == 1 then 0f32 else if slope == 0 then 1
                 f32 else (anchorY-focusPoint.2)*(1/slope)
218          let p_anchor_x = (anchorX, focusPoint.2+dy)
219          let p_anchor_y = (focusPoint.1+dx, anchorY)
220          in if p_anchor_x.1 < p_anchor_y.1 then p_anchor_x else
                 p_anchor_y
221
222      let getFocusPoints (entryPoint: point) slope vertical halfsize
            y_step_dir =
223          let A = replicate (t32(2f32*halfsize*2f32-1f32)) (f32.
                 lowest, f32.lowest)
224          let (A, _, _) =
```

31

```
225            loop (A, focusPoint, write_index) = (A, entryPoint, 0)
226                while ( isInGrid halfsize y_step_dir focusPoint ) do
227                let nextpoint = (nextpointonline slope vertical
                        focusPoint)
228                in unsafe let A[write_index] = focusPoint
229                in (A, nextpoint, write_index+1)
230            in A
231
232     let lengths_map
233            (grid_size: i32)
234            (sint: f32)
235            (cost: f32)
236            (entryPoint: point): [](f32, i32) =
237
238            let vertical = f32.abs(cost) == 1
239            let slope =  cost/(-sint)
240
241            let size = r32(grid_size)
242            let halfsize = size/2.0f32
243
244            let y_step_dir = if slope < 0f32 then -1f32 else 1f32
245            let focuspoints = (getFocusPoints entryPoint slope vertical
                        halfsize y_step_dir)
246            --for all focuspoints, save index and distance
247            let mf = map(\i ->
248                let ind = if !(isInGrid halfsize y_step_dir (unsafe
                        focuspoints[i])) then -1 else index (unsafe
                        focuspoints[i]) halfsize y_step_dir
249                let dist = if isInGrid halfsize y_step_dir (unsafe
                        focuspoints[i+1]) then (unsafe (distance
                        focuspoints[i] focuspoints[i+1])) else 0.0f32
250                in (dist, ind)
251            ) (iota ((length focuspoints)-1))
252            in mf
253
254     let weights_map   (angles: []f32)
255                    (rays: []f32)
256                    (gridsize: i32) : [][](f32,i32) =
257            let halfsize = r32(gridsize)/2
258            let entrypoints = convert2entry angles rays halfsize
259            in map (\(p,sc) -> (lengths_map gridsize sc.1 sc.2 p))
                        entrypoints
260  }
```

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.