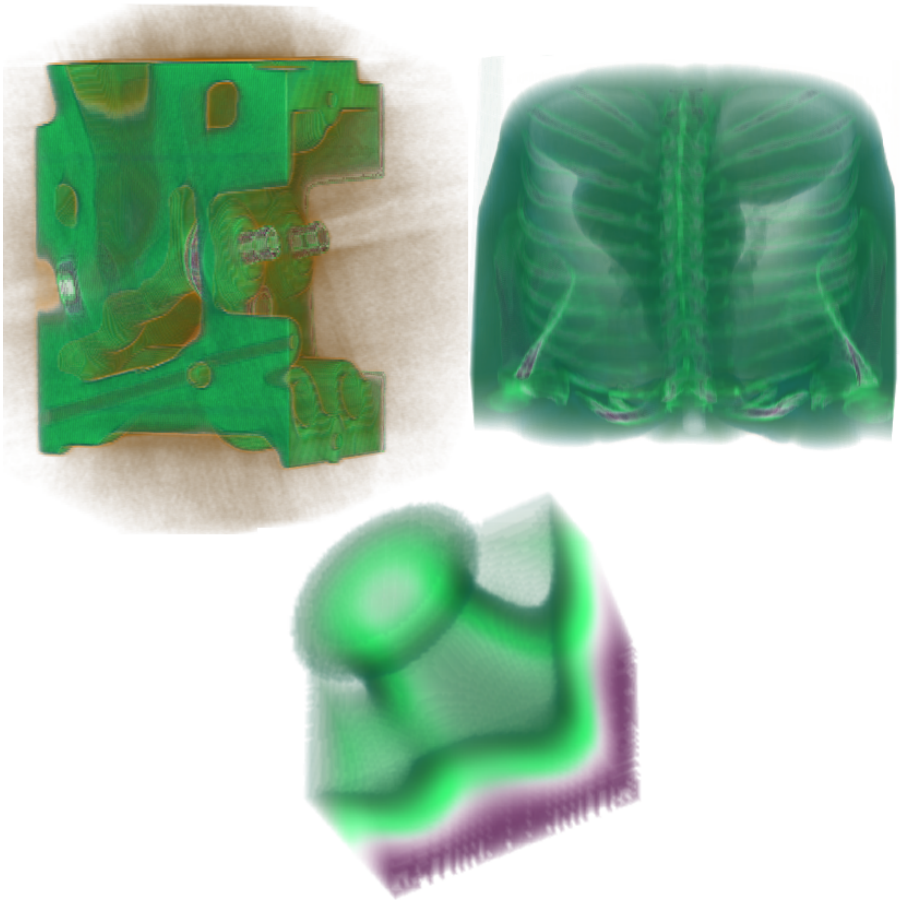
Technical University of Vienna

CUDA-based Reconstruction and Visualization of Non-uniform Point Sets

Bachelor thesis

Media Informatics

Maximilian Csuk



Supervisor: Dr. Erald Vuçini

Contents

Abstract

In this work we present a CUDA-based parallelized implementation of the reconstruction technique for 2D and 3D non-uniform point sets developed by [1] and Vuçini et al. [10]. Cubic B-splines are used as an approximation for the computationally expensive radial basis functions. As B-splines only have local support, a regularization term is added to handle larger gaps between sample points.
The main reconstruction task is the solving of a large but sparse linear system. The system's size makes the use of iterative algorithms necessary. Different solvers are implemented in order to find the best-performing approach. GPU-based algorithms require a different kind of strategy than traditional implementations. The possible degree of parallelization becomes the defining factor for the efficient realization of algorithms.
The use of B-splines also makes a multi-resolution approach possible. The system can be solved at a coarser representation and its solution acts as an initialization for the finer resolution. This further speeds up the reconstruction.
A downside of using graphics cards for computation is the relative sparsity of available memory. Most GPUs are not capable of handling big datasets directly. Therefore, the non-uniform representation is split into a number of smaller blocks. Each block is then reconstructed separately and the intermediate results are joined together resulting in the final solution.
The reconstruction results in terms of speed and quality are shown and are pitted against other CPU-based implementations.
A CUDA based interactive ray-tracer is used to visualize the 3 dimensional datasets and test the visual quality of the reconstruction.

1 Introduction

Planar and volumetric datasets are increasingly often retrieved and stored in a non-uniform representation to account for varying variance and detail, yet minimizing storage space. Therefore, the visualization and general handling of this data is increasingly important. But enhanced flexibility comes at a cost. The processing of non-uniform data is not straightforward. Several methods exist to visualize three dimensional datasets, including finite element analysis, approximation through basis functions and particle systems. All have their strengths and weaknesses, depending on the structure of the datasets to process and the expected results.
Indirect methods are known, which involve a transformation as an intermediate step, converting the data cloud into a uniform representation first. That makes it possible to continue using the existing toolsets which are already widely adopted. Several techniques exist for performing this operation. Most of them add a restriction on the maximum gap between the sample points though. This restriction is not always practical. Having no limit on the maximum distance between sample points, the reconstruction problem becomes not uniquely defined and hence, ill posed.
A way to deal with this is to convert the interpolation problem into an approximation problem. This can be done by adding a regularization term R to the equation. The weighting of R determines the smoothness of the reconstructed solution. But the smoothing term has the downside of increased errors at the input points, making the output function not follow the present data points as closely.
The, mathematically speaking, optimum solution for the approximation problem is obtained by using radial basis functions, in particular thin-plate splines. When positioning RBFs at the data points, a linear system can be set up and its solution represents the optimal weights necessary for each RBF. Even though thin-plate splines are the optimum solution, their computational cost rises drastically as the number of data points increases. Thin-plate splines have global support, meaning that each basis function is influenced by each data point across the whole dataset, leading to a huge linear system, which' size is dependent on the number of sample points. Furthermore, this system's matrix is dense, making the solving process computationally expensive.
Fortunately there are alternatives to using RBFs. B-splines represent a good approximation for RBFs but are non-zero on only a small range. The use of B-splines instead of RBFs introduces multiple advantages. Having finite support, the resulting linear system matrix is sparse, leading to a more efficient calculation. With proper restructuring, the problem can be formulated independently from the number of sample points, effectively making it dependent on the reconstruction grid size only. The system is also appropriate for a multi-resolution solving approach. Finally, no resampling step is re-

quired after obtaining the weights. A simple filter operation is sufficient.
Even though this approach outperforms other methods in terms of speed, it is still a big computational effort. GPUs offer a huge amount of processing power compared to CPUs. While CPUs have large caches and parts responsible for flow control, GPUs sacrifice these for more computing units. They are made up of thousands of processing units resulting in a massive parallel computation device.
The main reason for utilizing GPUs to do general purpose computation tasks is the gain in processing speed. But not every problem is suited to be solved with GPUs as not every algorithm and data structure can be parallelized easily or at all. A big issue when implementing GPU-based code is to occupy all or most of the GPU's processing units throughout the computation. Only utilizing a few of them results in poor performance that can then be easily matched or even outperformed by today's CPUs. This paper also addresses the difficulties and differences compared to traditional implementations that occur when developing for a massive parallel computation device like the GPU.

2 Related work

A number of different techniques have been developed in order to deal with non-uniform spatial data. A common method for visualizing non-uniform point data works by polyhedralizing the data first and rendering the new structures. This approach was used in quite a number of different works [6] [7] [11].
Another technique is to approximate the given set with the help of basis functions. The kind of basis function used plays a major role for both the resulting quality and reconstruction performance. Other works in this field use radial basis functions (RBFs) or more specifically, thin-plate splines [4] [3].
In order to lower the computational complexity that RBFs imply while still maintaining a good reconstruction quality, approximations can be used. Both Arigovindan et al. [1] and Vuçini et al. [10] use B-splines instead of thin-plate splines, improving the reconstruction speed by orders of magnitude.
This work takes their findings and translates them into a CUDA based implementation in order to use the vast amount of processing power present in recent GPUs.

3 Datasets

For testing, images and volumetric datasets are used for 2D and 3D reconstruction, respectively. Both synthetic and natural images are utilized to test different reconstruction aspects. For testing purposes, the sample points are extracted from complete datasets to be able to compare the reconstructed results with the originals. To measure the reconstruction error objectively, an error metric is utilized (see Appendix B). Images 1 and 2 each shows four of the used datasets.
If not stated otherwise, the sample points are selected by first filtering with a Laplacian filter (see Appendix A) and selecting the points with the highest absolute laplacian values. This guarantees that the points with a high gradient are selected first and less important regions are selected last.
The selected datasets range from high detail (Bridge, Natural Convection, Skull) to low detail (Head, Fuel).
Another type of 3D datasets which doesn't provide a ground truth is also tested to verify the reconstruction on real world non-uniform point clouds. These datasets can be reconstructed at almost arbitrary sizes, making another trade-off between faster reconstruction and more correct results possible. Modifying the reconstruction grid's size along the different dimensions can also be used to provide varying levels of detail. Image 3 shows four renderings from the pool of non-uniform datasets.

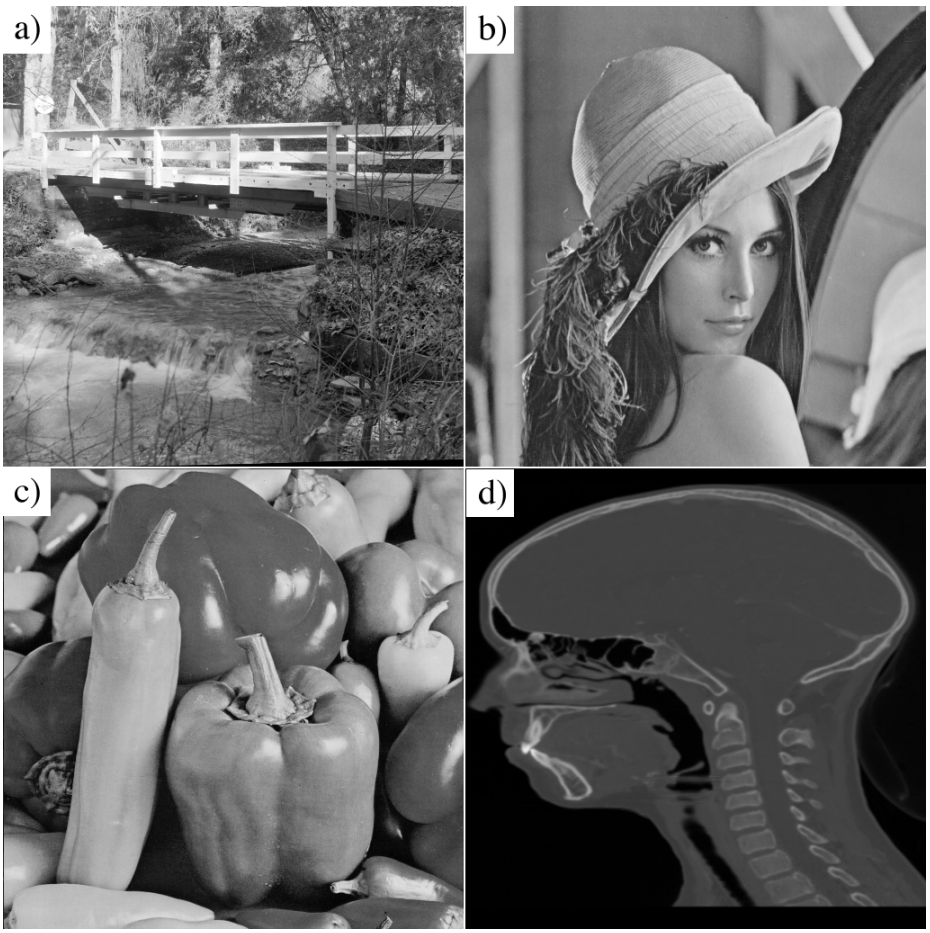


Figure 1: *Selection of used image datasets: a) Bridge, b) Lena, c) Fruits and d) Head, all 512×512*

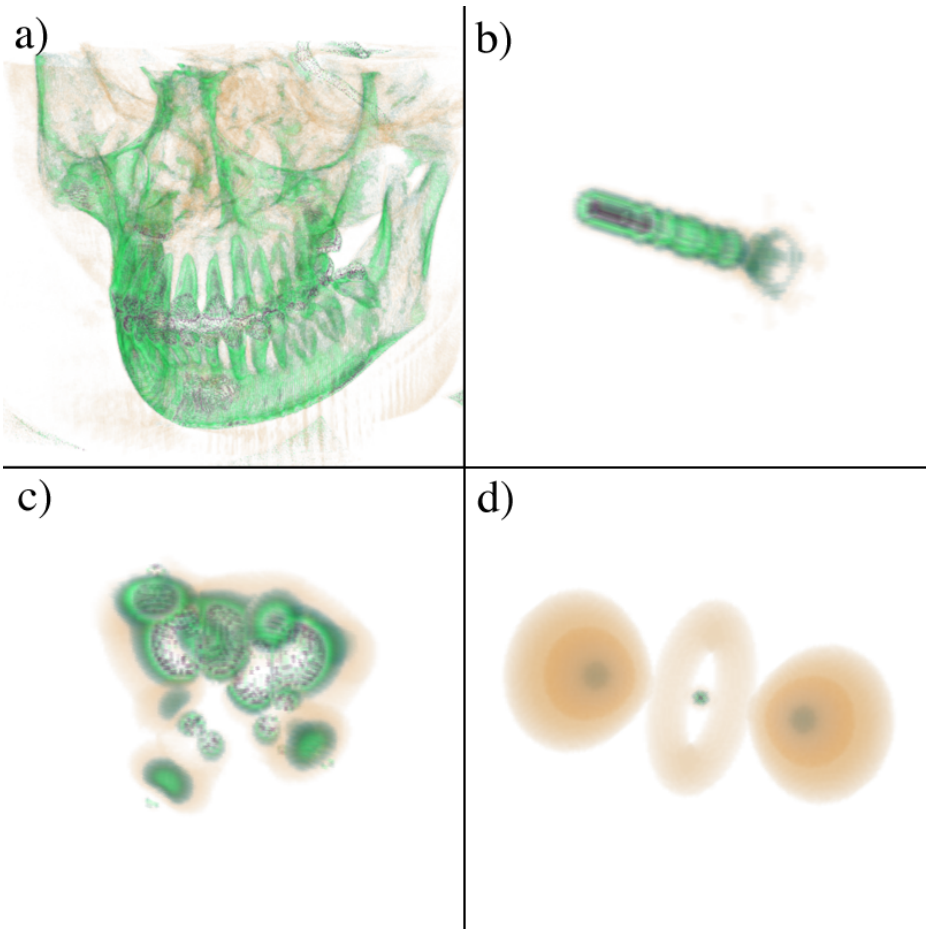


Figure 2: *Renderings of used volume datasets: a) Skull (256×256×256), b) Fuel (64×64×64), c) Neghip (64×64×64), d) Hydrogen Atom (128×128×128)*

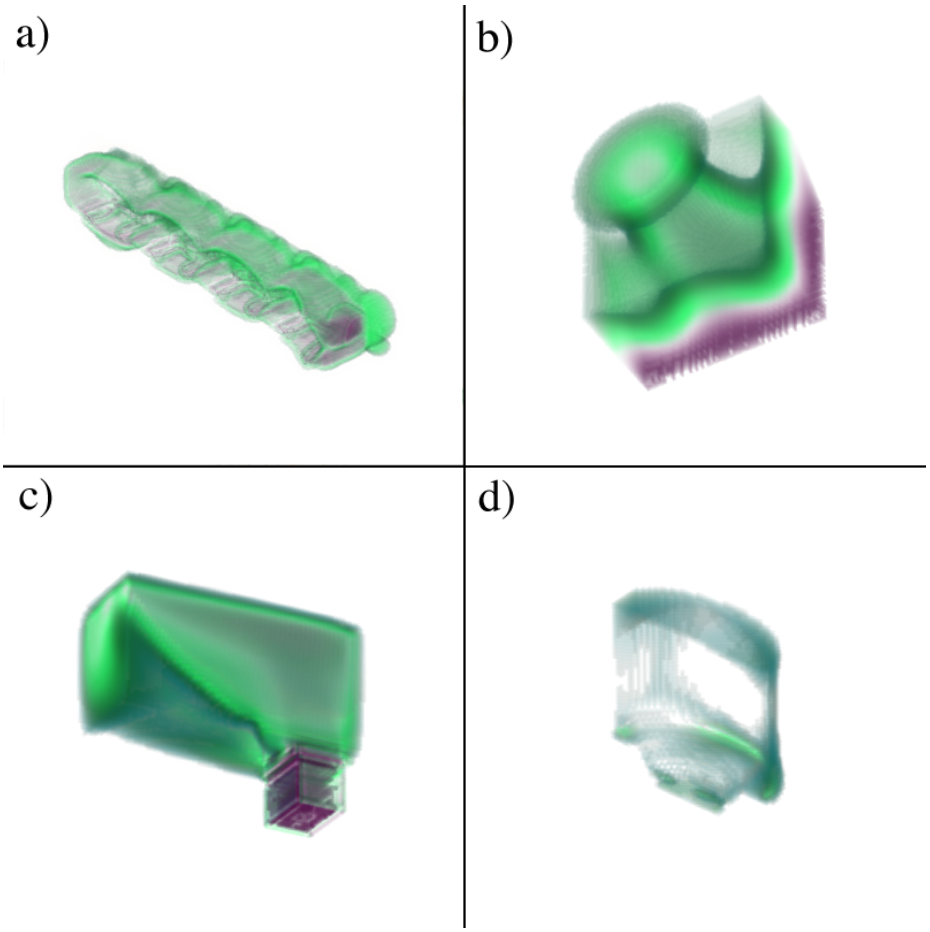


Figure 3: *Renderings of used non-uniform volume datasets: a) Cooling Jacket (222×128×122), b) Natural Convection (61×61×61), c) Flow Transport (44×58×58), d) Fuel Injection (39×51×60)*

4 Setup

The building blocks for the linear system are B-spline basis functions of the form:

$$\beta(x) = \begin{cases} \frac{x^3}{6} & 0 < x \leq 1 \\ \frac{-3x^3+12x^2-12x+4}{6} & 1 < x \leq 2 \\ \frac{3x^3-24x^2+60-44}{6} & 2 < x \leq 3 \\ \frac{(4-x)^3}{6} & 3 < x \leq 4 \\ 0 & otherwise \end{cases} \tag{1}$$

The temporary matrix F is constructed out of B-spline products. For the 3D case:

$$F_{i,N_xN_ym+N_xl+k} = \beta(x_i - k)\beta(y_i - l)\beta(z_i - m) \tag{2}$$

Therefore, F^TF is calculated like this:

$$(F^TF)_{N_xN_ym_1+N_xl_1+k_1,N_xN_ym_2+N_xl_2+k_2}$$
$$=$$
$$\sum_i \beta(x_i - k_1)\beta(y_i - l_1)\beta(z_i - m_1)\beta(x_i - k_2)\beta(y_i - l_2)\beta(z_i - m_2)$$

Note that, due to the finite support of the B-spline function β, only samples within 4 units in each dimension can influence each entry in F^TF.
F^Tf is calculated in a similar fashion:

$$(F^Tf)_{N_xN_ym+N_xl+k} = \sum_i f_i\beta(x_i - k)\beta(y_i - l)\beta(z_i - m) \tag{3}$$

In order to perform the reconstruction, a linear system of the form

$$(F^TF + \lambda R)x = F^Tf \tag{4}$$

has to be set up.
R is the regularization matrix scaled by λ, the regularization factor, which governs the smoothness of the reconstruction. Each value in R is only dependent on the position in the matrix. This can be done very efficiently (3D case):

$$\begin{aligned} R_{x,y,z} \quad = \quad & T_{2,x} * T_{0,y} * T_{0,z} + \\ & 2 * T_{1,x} * T_{1,y} * T_{0,z} + \\ & T_{0,x} * T_{2,y} * T_{0,z} + \\ & 2 * T_{1,x} * T_{0,y} * T_{1,z} + \\ & 2 * T_{0,x} * T_{1,y} * T_{1,z} + \\ & T_{0,x} * T_{0,y} * T_{2,z} \end{aligned}$$

T is a small lookup table consisting of pre-calculated integrals.
F^TF and F^Tf are dependent on the sample points f. Replacing $F^TF + \lambda R$ with A and F^Tf with b yields the typical equation:

$$Ax = b \tag{5}$$

which needs to be solved for x.
For matrix F^TF and the right-side vector F^Tf, both a shooting and a gathering algorithm were tried. The shooting algorithm is easier to implement but also requires the serialization of write-accesses to avoid race conditions. The gathering algorithm on the other hand is fully parallelized and is also independent from the number of sample points, effectively making the reconstruction process as a whole dependent on the size of the reconstruction grid only. Unfortunately, the gathering algorithm requires the reconstruction grid to be aligned with the positions of the sample points. This is a serious limitation and makes it impractical for most real-world cases. Therefore, the shooting algorithm is used throughout the report. For datasets which' point positions only differ by a multiple of the grid size, the gathering algorithm could improve reconstruction times though.

5 Single resolution linear equation solvers

5.1 Domain description

To get a better understanding, one first has to look at the problem at hand. The task is to solve a very large linear system, depending on the reconstruction grid's size. In most cases, direct solvers would be inappropriate for these systems as they aren't able to work on these amounts of data in a reasonable timeframe. Iterative approaches on the other hand can tackle such systems well and also provide more flexibility concerning the needed accuracy of the results.
An important property of the matrix A (5) is its sparsity (due to the B-spline basis functions). This makes the use of a sparse matrix storage scheme not only viable but also necessary, given its size. Furthermore, the matrix is symmetric and positive semi-definite, which affects the range of suitable algorithms as well as their convergence behavior.
Although a multi-grid approach is used (introduced in section 6), it still relies on normal iterative approaches to do the actual solving. 3 different solvers were implemented and tested out. With CUDA being a massive parallel environment, it is of most importance for the algorithms to be parallelized to achieve good performance. The following sections point out the specialities and difficulties when being implemented in CUDA and when used in a multi-grid approach. The shown algorithms relate to the 3D case. For a more detailed mathematical explanation of the linear equation solvers, refer to [2].

5.2 Building blocks for matrix-vector calculations

The whole reconstruction process mostly consists of basic matrix-vector operations. These include:

* Vector scaling
* Vector dot product
* AXPY (vector addition preceded by scalar multiplication of one vector)
* Matrix-vector multiplication.

CUBLAS, which is an add-on library to CUDA, offers most of these operations. Only the matrix-vector multiplication needs to be implemented to work with the special dense matrix structure (Algorithm 1 and 2). Note that the parts inside the outer loops can be run in parallel and hence are implemented inside a single CUDA kernel.

Algorithm 1 3D Parallel matrix-vector multiplication $multiply(A, \bar{b})$

{Outer, fully parallelizable loops}
for $i = 0$ **to** $N_x - 1$ **do** {Parallel}
  for $j = 0$ **to** $N_y - 1$ **do** {Parallel}
    for $k = 0$ **to** $N_z - 1$ **do** {Parallel}
      $multiplyKernel(A, \bar{b}, \bar{x}, i, j, k)$
    end for
  end for
end for
return $\bar{x}$

Algorithm 2 3D Multiplication kernel $multiplyKernel(A, \bar{b}, \bar{x}, i, j, k)$

$\bar{t_x} = [N_x * 3 - 6, N_x * 2 - 3, N_x - 1, 0, N_x, N_x * 2 - 1, N_x * 3 - 3]$
$\bar{t_y} = [N_y * 3 - 6, N_y * 2 - 3, N_y - 1, 0, N_y, N_y * 2 - 1, N_y * 3 - 3]$
$\bar{t_z} = [N_z * 3 - 6, N_z * 2 - 3, N_z - 1, 0, N_z, N_z * 2 - 1, N_z * 3 - 3]$
$s = 0$
for $j_x = max(-3, -i)$ **to** $min(3, N_x - 1 - i)$ **do**
  for $j_y = max(-3, -j)$ **to** $min(3, N_y - 1 - j)$ **do**
    for $j_z = max(-3, -k)$ **to** $min(3, N_z - 1 - k)$ **do**
      {Single component multiplication and summation}
      $s = s + A_{i+\bar{t_x}_{(j_x+3)}, j+\bar{t_y}_{(j_y+3)}, k+\bar{t_z}_{(j_z+3)}} * b_{i+j_x, j+j_y, k+j_z}$
    end for
  end for
end for

{Set output}
$x_{i,j,k} = s$

5.3 Conjugate Gradient

Fortunately, Conjugate Gradient is parallel in nature and only relies on operations that are not dependent on it's own partial results. Per iteration, the algorithm needs to compute the following:

* 1 matrix-vector multiplications
* 3 dot products
* 3 AXPY operations

Algorithm 3 shows the implementation of the parallel Conjugate Gradient method.

Algorithm 3 Parallel Conjugate Gradient algorithm $CG(A, \bar{b}, \bar{x})$

{Allocate space for temporary vectors}
$\bar{r} = Allocate(N_x * N_y * N_z)$ {Residual vector}
$\bar{p} = Allocate(N_x * N_y * N_z)$ {Current gradient vector}
$\bar{tmp} = Allocate(N_x * N_y * N_z)$ {Temporary vector}

{Calculate initial residuals}
$\bar{r} = multiply(A, \bar{b})$
$\bar{r} = axpy(\bar{r}, \bar{b}, -1)$

{Iteration loop}
for $i = 0$ **to** $numIterations - 1$ **do**
  {Calculate $\lambda = (\bar{r}^T \bar{r})/(\bar{p}^T A \bar{p})$}
  $rDot = dot(\bar{r}, \bar{r})$
  $\bar{tmp} = multiply(A, \bar{p})$
  $pDot = dot(\bar{p}, \bar{p})$
  $\lambda = rDot/pDot$

  $\bar{x} = axpy(\bar{p}, \bar{x}, \lambda)$ {Update $\bar{x}$}

  $\bar{r} = axpy(\bar{tmp}, \bar{r}, -\lambda)$ {Update $\bar{r}$}

  $\beta = dot(\bar{r}, \bar{r})/rDot$ {Calculate β}

  $\bar{p} = axpy(\bar{p}, \bar{r}, \beta)$ {Update $\bar{p}$}
end for

5.4 Jacobi

The Jacobi algorithm is implemented by first taking the diagonal part of the matrix and inverting it as a setup-step. This can be performed very

fast because in the dense matrix format used, the diagonal part corresponds to the first N_x, N_y and N_z values in each dimension. The inversion of the diagonal matrix is trivial and can be implemented very efficiently. Per iteration, the algorithm needs to compute the following:

* 2 matrix-vector multiplications
* 1 vector multiplication
* 1 AXPY operation

The Jacobi algorithm has the least expensive setup cost but also the slowest convergence rate of the tested algorithms.

Algorithm 4 Parallel Jacobi algorithm $Jacobi(A, \bar{b}, \bar{x})$

{Setup diagonal inverse matrix}
$\tilde{A} = inverseDiagonal(A)$

{Allocate temporary vectors}
$\bar{r} = Allocate(N_x * N_y * N_z)$
$\bar{tmp} = Allocate(N_x * N_y * N_z)$

{Iteration loop}
for $i = 0$ **to** $numIterations - 1$ **do**
  {Calculate residuls}
  $\bar{r} = multiply(A, \bar{x})$
  $\bar{r} = axpy(\bar{r}, \bar{b}, -1)$

  {Calculate tmp}
  $\bar{tmp} = multiply(\tilde{A}, \bar{r})$

  {Add to x}
  $\bar{x} = axpy(\bar{tmp}, \bar{x}, 1)$
end for

5.5 Gauss-Seidel

The Gauss-Seidel algorithm is the hardest to implement efficiently on parallel hardware. The reason is that unlike the other two algorithms, each Gauss-Seidel iteration is dependent on it's own partial results [2, chapter 2.2.2]. Without adjustments, it can only run in a completely serial fashion, making it useless for GPU calculations. Fortunately, it's possible to exploit the special structure of matrix A. Each result value is only dependent on the 7 values above it. Therefore, a modified form of red-black ordering can

be implemented [2, chapter 2.4]. By re-ordering the system, it is possible to fully parallelize 1/8th of the new partial results for each iteration. These results are then used for computing the next values. Repeating the process 8 times, the iteration is complete. Figure 4 shows this multi color ordering. Algorithms 5 shows the index conversion into the red-black ordering. Taking the original index as input, it outputs the reordered index position of a line. Algorithm 6 shows the reverse operations.

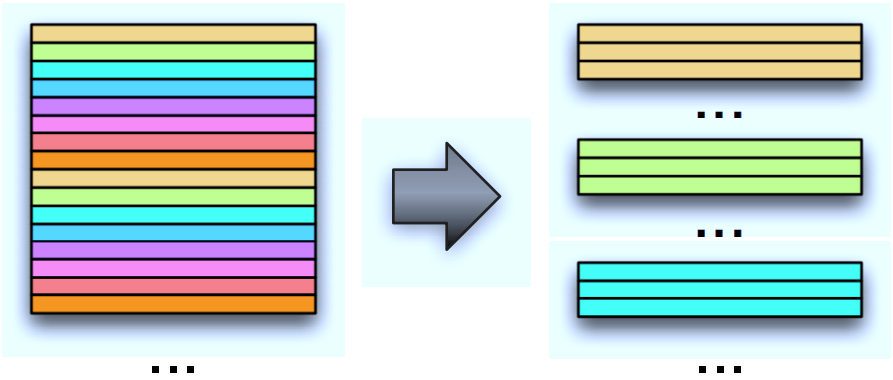


Figure 4: *Reordering of the matrix to achieve better parallelization*

The parallel calculation itself is very similar to a matrix vector multiplication plus one vector addition and one vector division afterwards. Note that the shown algorithm 8 only works on systems where $N_x \ \% \ 8 = 0$. Similar to the matrix-vector multiplication, the outer loops can be run in parallel. The Gauss-Seidel algorithm has a big setup cost because of the reordering of the linear system. Looking at the convergence rate, it performs better than Jacobi but worse than Conjugate Gradient.

Algorithm 5 Marshal $marshal(x)$

return $(x \ \% \ (N_x/8)) * 8 + (x/(N_x/8))$

Algorithm 6 Unmarshal $unmarshal(x)$

return $(x \ \% \ 8) * (N_x/8) + (x/8)$

Algorithm 7 Gauss-Seidel kernel $GSKernel(A, \bar{b}^M, \bar{x}^M, a)$

{Separate lookup table for each dimension}
$\bar{t_x} = [N_x * 3 - 6, N_x * 2 - 3, N_x - 1, 0, N_x, N_x * 2 - 1, N_x * 3 - 3]$
$\bar{t_y} = [N_y * 3 - 6, N_y * 2 - 3, N_y - 1, 0, N_y, N_y * 2 - 1, N_y * 3 - 3]$
$\bar{t_z} = [N_z * 3 - 6, N_z * 2 - 3, N_z - 1, 0, N_z, N_z * 2 - 1, N_z * 3 - 3]$

{Retrieve indices}
$i = a \ \% \ N_x$
$j = (a/N_x) \ \% \ N_y$
$k = a/(N_x * N_y)$

$i^U = unmarshal(i)$ {Marshal index along x-dimension}
$s = 0$
for $j_x^U = max(-3, -i^U)$ **to** $min(3, N_x - 1 - i^U)$ **do**
  for $j_y = max(-3, -j)$ **to** $min(3, N_y - 1 - j)$ **do**
    for $j_z = max(-3, -k)$ **to** $min(3, N_z - 1 - k)$ **do**

      {Calculate unmarshalled index for matrix A}
      $index_A = i^U + \bar{t_x}_{(j_x^U+3)}, j + \bar{t_y}_{(j_y+3)}, k + \bar{t_z}_{(j_z+3)}$

      {Calculate marshalled index for vector $\bar{x}$}
      $index_x = marshal(i^U + j_x^U), j + j_y, k + j_z$

      {Add to intermediate result}
      $s = s + A_{index_A} * \bar{x}^M_{index_x}$
    end for
  end for
end for

{Set output}
$\bar{x}^M_{i,j,k} = (\bar{b}^M_{i,j,k} - s)/A_{i^U,j,k}$

Algorithm 8 Parallel Gauss-Seidel algorithm $GS(A, \bar{b}, \bar{x})$

{Allocate marshalled temporary vectors}
$\bar{x}^M = Allocate(N_x * N_y * N_z)$
$\bar{b}^M = Allocate(N_x * N_y * N_z)$

{Marshal left- and right-hand side vector}
for $x = 0$ **to** $N_x - 1$ **do** {Parallel}
  for $y = 0$ **to** $N_y - 1$ **do** {Parallel}
    for $z = 0$ **to** $N_z - 1$ **do** {Parallel}
      $\bar{b}^M_{marshal(x),y,z} = \bar{b}_{x,y,z}$
      $\bar{x}^M_{marshal(x),y,z} = \bar{x}_{x,y,z}$
    end for
  end for
end for

{Iteration loop}
for $i = 0$ **to** $numIterations - 1$ **do**
  for $it = 0$ **to** 7 **do**

    $start = it * ({N_x}^3/8)$
    $end = (it + 1) * ({N_x}^3/8)$
    {Iteration}
    for $a = start$ **to** $end - 1$ **do** {Parallel}
      $GSKernel(A, \bar{b}^M, \bar{x}^M, a)$
    end for
  end for
end for

{Unmarshal result-vector}
for $x = 0$ **to** $N_x - 1$ **do** {Parallel}
  for $y = 0$ **to** $N_y - 1$ **do** {Parallel}
    for $z = 0$ **to** $N_z - 1$ **do** {Parallel}
      $\bar{x}_{unmarshal(x),y,z} = \bar{x}^M_{x,y,z}$
    end for
  end for
end for

6 Multigrid linear equation solver

Because of the interscale 2-scale relation of B-splines [1], which form the building blocks for the reconstruction, the linear system can be brought to a coarser and therefore smaller scale and solved there. After upsampling, the intermediate result can act as a good initializer for the system at the bigger scale. This process can be deployed recursively, greatly increasing the reconstruction process' speed. The algorithm is shown schematically in figure 5.

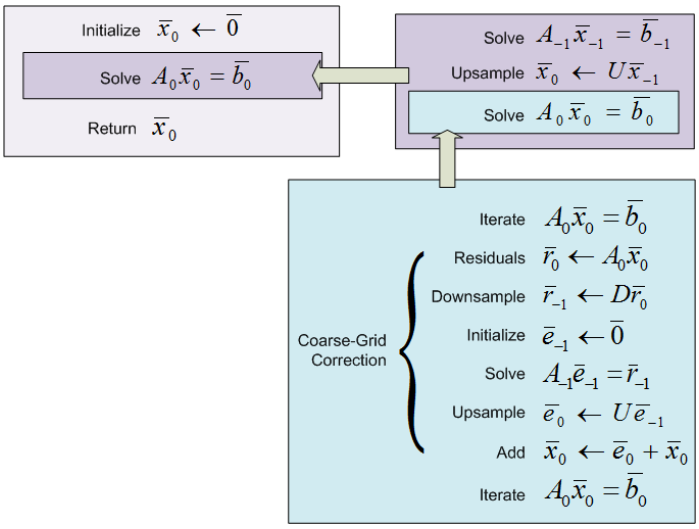


Figure 5: *The multigrid approach, shown in a modular fashion, thereby making the recursive nature of the algorithm visible. Note that every line of "Solve" can be split up into another smaller scale problem.*

To further speed up reconstruction, a mechanism called coarse-grid correction is applied. After performing the initial iterations for a given resolution (j), the residuals are calculated, downsampled to $(j+1)$ and used as the right-hand side for a smaller linear system $A_{(j+1)}e = r_{(j+1)}$. The result e, which acts as a form of error-correction, is upsampled again and added to the current solution. After coarse-grid correction, another number of iterations is carried out.
In order to implement a multi grid solver, fast vector up- and downsampling, as well as matrix downsampling is needed. These operations are composed of binomial filter operations, coarsening by dropping every second value and expanding by adding zeroes between two values, treating each dimension separately. Algorithms 9, 10, 11, 12, 13 and 14 show these operations for vectors along the X dimension. The operations for the other dimensions are implemented likewise.

Algorithms 15 and 16 are used for downsampling and filtering the dense matrix along the X dimension. Again, the other dimensions are operated on in a similar fashion. Figure 6 shows the steps necessary to downsample a matrix visually.

Algorithm 9 Kernel for filtering vector along X dimension $filterVectorXKernel(\bar{x}, \bar{x}^f, width, height, depth, x, y, z)$

$filter = [1, 4, 6, 4, 1]$
$\bar{x}^f_{x,y,z} = 0$
$denom = 0$
for $i = 0$ **to** 4 **do**
  $d_x = x + (i - 2)$
  if $0 <= d_x < width$ **then**
    $denom = denom + filter_i$
    $\bar{x}^f_{x,y,z} = \bar{x}^f_{x,y,z} + \bar{x}_{d_x,y,z}$
  end if
end for
$\bar{x}^f_{x,y,z} = 1/denom$

Algorithm 10 Filtering vector along X dimension $filterVectorX(\bar{x}, \bar{x}^f, width, height, depth)$

for $i = 0$ **to** $width - 1$ **do** {parallel}
  for $j = 0$ **to** $height - 1$ **do** {parallel}
    for $k = 0$ **to** $depth - 1$ **do** {parallel}
      $filterVectorXKernel(\bar{x}, \bar{x}^f, width, height, depth, i, j, k)$
    end for
  end for
end for

Algorithm 11 Kernel for reducing vector along X dimension $reduceVectorXKernel(\bar{x}, \bar{x}^r, width, width_{small}, height, x, y, z)$

$\bar{x}^r_{x+y*width_{small}+z*width_{small}*height} = \bar{x}^f_{x*2+y*width+z*width*height}$

Algorithm 12 Reducing vector along X dimension $reduceVectorX(\bar{x}, \bar{x}^r, width, width_{small}, height)$

for $i = 0$ **to** $width - 1$ **do** {parallel}
  for $j = 0$ **to** $height - 1$ **do** {parallel}
    for $k = 0$ **to** $depth - 1$ **do** {parallel}
      $reduceVectorXKernel(\bar{x}, \bar{x}^r, width, width_{small}, height, i, j, k)$
    end for
  end for
end for

Algorithm 13 Kernel for expanding vector along X dimension $expandVectorXKernel(\bar{x}, \bar{x}^e, width, width_{small}, height, x, y, z)$

$\bar{x}^e_{x*2+y*width+z*width*height} = \bar{x}^f_{x+y*width_{small}+z*width_{small}*height}$

Algorithm 14 Expanding vector along X dimension $expandVectorX(\bar{x}, \bar{x}^e, width, width_{small}, height)$

for $i = 0$ **to** $width - 1$ **do** {parallel}
  for $j = 0$ **to** $height - 1$ **do** {parallel}
    for $k = 0$ **to** $depth - 1$ **do** {parallel}
      $expandVectorXKernel(\bar{x}, \bar{x}^e, width, width_{small}, height, i, j, k)$
    end for
  end for
end for

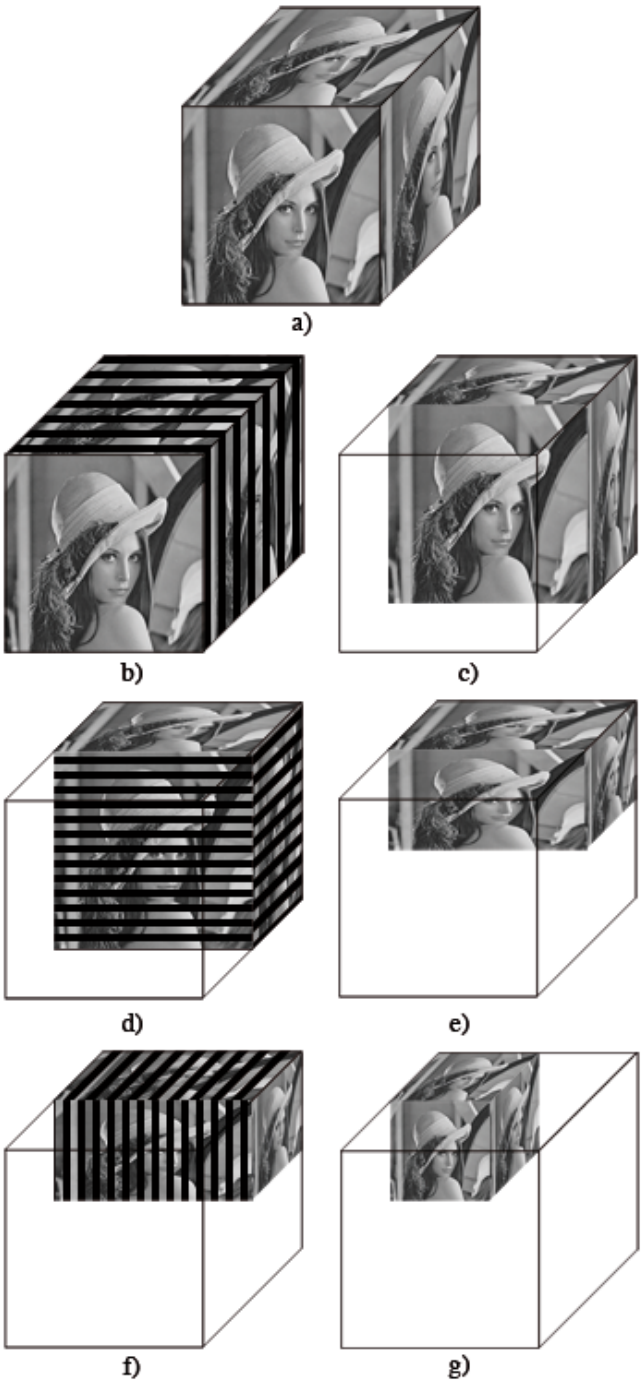


Figure 6: *Downsampling of a 3D matrix: a) Start-matrix, b) downsample along first dimension, c) filter along first dimension, d) downsample along second dimension, e) filter along second dimension, f) downsample along third dimension, g) filter along third dimension*

Algorithm 15 Kernel for filtering matrix along X dimension $filterMatrixX(A, A^f, x, y, z)$

$filter = [1, 4, 6, 4, 1,$
$4, 16, 24, 16, 4,$
$6, 24, 36, 24, 6,$
$4, 16, 24, 16, 4,$
$1, 4, 6, 4, 1]$
$d^k = [0, -1, -2, -3, -4,$
$1, 0, -1, -2, -3,$
$2, 1, 0, -1, -2,$
$3, 2, 1, 0, -1,$
$4, 3, 2, 1, 0]$
$d^j = [-2, -2, -2, -2, -2,$
$-2, -1, -1, -1, -1,$
$-2, -1, 0, 0, 0,$
$-2, -1, 0, 1, 1,$
$-2, -1, 0, 1, 2,$
$-2, -1, 0, 1, 2,$
$-2, -1, 0, 1, 2,$
$-2, -1, 0, 1, 2,$
$-2, -1, 0, 1, 2,$
$-2, -1, 0, 1, 2,$
$-2, -1, 0, 1, 2]$
$denseIndexJumpsX = [0, N_x, N_x * 2 - 1, N_x * 3 - 3, N_x * 4 - 6, N_x * 5 - 10, N_x * 6 - 15, N_x * 7 - 21]$
$DMSizesX = [N_x, N_x - 1, N_x - 2, N_x - 3, N_x - 4, N_x - 5, N_x - 6, N_x - 7]$

{Convert dense index}
$j_x^c = x, k^c = 0$
while $j_x^c >= N_x - k_c$ **do**
$j_x^c -= N_x - k_c, k_c = k_c + 1$
end while
{Filter operation}
$s = 0, denom = 0$
for $i = 0$ **to** 24 **do**
$k = |k_c + d_i^k|, j_x = j_x^c + d_{i+k_c*5}^j$
if $0 <= j_x < DMSizesX_k$ **then**
$denom = denom + filter_i$
if $k <= 4$ **then**
$s = s + A_{denseIndexJumpsX_k + j_x, y, z} * filter_i$
end if
end if
end for
$A_{x,y,z}^f = s/denom$

Algorithm 16 Kernel for reducing matrix along X dimension $reduceMatrixX(A, A^r, x, y, z)$

$denseIndexJumpsX = [0, N_x, N_x * 2 - 1, N_x * 3 - 3, N_x * 4 - 6, N_x * 5 - 10, N_x * 6 - 15, N_x * 7 - 21]$
$N_x^h = (N_x - 1)/2 + 1$
$denseIndexJumpsX^h = [0, N_x^h, N_x^h * 2 - 1, N_x^h * 3 - 3, N_x^h * 3 - 6, N_x^h * 3 - 10, N_x^h * 3 - 15, N_x^h * 3 - 21]$
$t = 0$
if $x >= denseIndexJumpsX_1^h$ **then**
  $t = t + 1$
end if
if $x >= denseIndexJumpsX_2^h$ **then**
  $t = t + 1$
end if
if $x >= denseIndexJumpsX_3^h$ **then**
  $t = t + 1$
end if
$n_x = denseIndexJumpsX_{t*2} + (x - denseIndexJumpsX_t^h) * 2$
$A^r_{x,y,z} = A_{n_x,y,z}$

7 Block-based reconstruction

The reconstruction of bigger datasets, especially volumetric ones, is very memory intense. Most of it is needed for storing and handling the dense matrix. For example, the matrix for a dataset with 64×64×64 values amounts to about 72 MB. For a dataset of size 128×128×128 even 559 MB, already too much for low-end GPUs. When downsampling, a temporary matrix of roughly twice the size is needed, taking away even more memory. Adding the other necessary elements (input samples, left- and right hand size vectors, temporary data, ...) can be too much for a GPU to handle. To be able to reconstruct these datasets without exceeding the memory limit, a block-based approach is implemented.
The input samples are grouped into blocks, thereby separating the dataset. Each block is then reconstructed independently. Finally, the partial results are assembled back into the final image or volume.
However, this results in visible seams located at the edges between the blocks. To counter that, the reconstruction area for each block is enlarged by 4 units in each direction (exactly the range of a cubic B-splines' influence) and the block-result is extracted again afterwards. These paddings of course negatively affect performance. For example: when reconstructing a 64×64×64 dataset with 2 blocks in each dimension, 8 blocks of size 38×38×38 need to be reconstructed. Extracting areas of size 32×32×32 in the center of each block, the final results are assembled again.

8 Visualization

Whereas 2D-datasets can easily be shown as images, volumetric datasets need to be visualized by projecting them onto the 2D computer screen. Therefore, an interactive volumetric ray tracer was implemented. Each single pixel value of the resulting image is calculated by one CUDA thread.
For better visual results, the scalar values are transformed into RGBA-vectors with a so-called "transfer-function". Figure 14 shows the visualization of the Skull dataset. Figure 7 shows the color map that is used by the transfer function to determine the color and opacity. The transfer function that adds RGBA values when encountering a voxel with density *input* is shown in algorithm 17.

Figure 7: *Color map used for visualization of volumetric datasets*

Algorithm 17 Transfer function $TF(input, map, r, g, b, a)$

$index = input * width(map)$
$r = r + (1 - a) * input * map_{index}.red * map_{index}.alpha$
$g = g + (1 - a) * input * map_{index}.green * map_{index}.alpha$
$b = b + (1 - a) * input * map_{index}.blue * map_{index}.alpha$
$a = a + (1 - a) * input * map_{index}.alpha$

9 Implementation details

The used code is optimized for CUDA compute capabilities 1.2. Some kernels need to perform atomic float-additions to ensure correct results without race conditions. Unfortunately, this function does not exist natively for versions < 2.0 so a (slow) manual implementation has to be deployed [9, chapter B.11.1]. With some tweaking, a version optimized for $>= 2.0$ could be set up as well.

9.1 Dense matrix format

In order to reduce the memory footprint, the matrix is stored in a dense format which eliminates empty and duplicate values present in the normal representation. The matrix is already created in this format to fully eliminate the need for the sparse representation and further speed up the process. For the 3D case, this reduces the storage requirement from ${N_x}^2 * {N_y}^2 * {N_z}^2 *$ 4 bytes to $(N_x * 4 - 6) * (N_y * 4 - 6) * (N_z * 4 - 6) * 4$ bytes. Another advantage is the improved performance when performing a matrix-vector multiplication.

9.2 Performance considerations

The most important performance improvements are achieved by coalescing global memory accesses [8, chapter 3.2.1]. While most steps can be fully coalesced, some have too unusual access patterns, which can either not be put in a better, coalesce-able order or the process was determined be too time and/or memory consuming. Unfortunately, the often-used dense matrix-vector multiplication is one of those operations (see algorithm 1). In these cases, texture memory is used to speed up fetching of spatially close-by values through the GPU's texture cache.
Another important performance factor is kernel-size. The used CUDA kernels are optimized to use as few registers as possible in order to keep all data in the (fast) registers and avoid spilling it over to (slow) local memory. To constantly occupy the GPU it is critical that enough threads can work in parallel on each kernel launch. On GPUs with CUDA compute capabilities up to 1.3, 512 threads can be spawned at the same time. Approximately half of them need to be used in order to fully occupy the GPU, depending on the kernel's size. On the other hand, too many threads increase the register pressure, which can have even higher negative effects on performance. A careful trade-off has to be chosen for each kernel call separately.
Small improvements are achieved by pre-computing values like the coefficients for the regularization matrix.
On the GPU, integer modulo-operations are very slow and are hence replaced by bit-shifts whenever possible. The same is true for divisions, which are replaced by either multiplication with the (pre-calculated) reciprocal value

or bit-shifts, whenever possible.
The end results show that for most inputs, the critical parts of the reconstruction process are the $F^T F$ matrix creation and the dense matrix-vector multiplication (table 1). Great care was taken when optimizing these components.

Component	Percentage
Create $F^T F$ matrix	47,16 %
Multiply vector matrix	37,98 %
Create $F^T f$ vector	5,46 %
Filter matrix	5,03 %
Add R matrix	2,36 %
Rest	2,01 %

Table 1: *Relative computation times of components when reconstructing the Natural Convection dataset (61×61×61).*

10 Results

Many CUDA enabled Nvidia cards only provide a subset of all possible instructions. This is reflected by the CUDA compute capability versioning. For example, GPUs with a compute capability <2.0 can only spawn 512 threads simultaneously in each kernel-call. With capability >=2.0, this upper bound is increased to 1024. Working with atomic floating-point operations inside kernels also requires at least version 2.0. Devices with a higher compute capability version feature more raw processing power as well. For a detailed explanation, see [9]. If not stated otherwise, the tests were performed on a Nvidia GT 220 with 1 GB RAM. This device has a CUDA compute capability of 1.2.

10.1 Single resolution results

10.1.1 Comparison of linear system solvers

The reconstruction process was carried out using the 3 single resolution solvers to be able to compare their performances. Each solver was iterated 100 times. The 2D dataset Fruits and the 3D dataset Natural Convection were used and the results are shown in Tables 2 and 3. Other datasets show similar results.
The metric to compare the reconstruction performance is described in Appendix B. Overall, the Conjugate Gradient solver produces the best results. Jacobi, albeit simple to implement, doesn't offer good convergence rates. Gauss-Seidel is not suited for a truly parallel implementation and therefore can't achieve a satisfying iteration speed. Conjugate Gradient achieves the best convergence rate by far and also the best iteration speed. Therefore, this method is used throughout this report.

Sample points:	**20 %**		**30 %**		**50 %**	
	Time	RMS_g	Time	RMS_g	Time	RMS_g
Jacobi	1,663	1,31	1,719	0,72	1,813	0,30
Gauss-Seidel	2,457	1,27	2,497	0,69	2,534	0,26
Conjugate Gradient	1,404	0,34	1,416	0,21	1,503	0,18

Table 2: *Comparison of linear equation solvers using the Fruits dataset ($\lambda = 1.9$).*

	Time	RMS
Jacobi	1,663	1,31
Gauss-Seidel	2,457	1,27
Conjugate Gradient	1,404	0,34

Table 3: *Comparison of linear equation solvers using the Natural Convection dataset ($\lambda = 0.001$).*

10.2 Optimal regularization

An important factor for the reconstruction quality is the regularization variable λ. Deciding the optimal value is not easy and is very much dependent on the input samples. Figures 9 and 10 show the reconstruction results for the used datasets with varying values of λ. The best results for the image datasets are obtained using a λ value in the interval [2, 3]. Higher values result in blurred images, lower values produce too many artifacts in areas where few or no sample points are present. Image 8 shows this visually. From the three presented results, the center image offers the best reconstruction quality with a λ of 2,1.

Figure 8: *Reconstructed Lena dataset with varying λ parameter. From left to right: λ: 0,1 RMS_g: 8,03; λ: 2,1 RMS_g: 5,08; λ: 4,1 RMS_g: 5,34*

The volume datasets require much smaller λ values. This is due to the decreased complexity in these datasets and the additional dimension, which increases the chance for a reconstruction point to have an input sample nearby. Especially the hydrogen dataset is almost completely defined by the 20 % of sample points and hence needs very little regularization. In general, the optimal value needs to be determined on a per dataset basis.

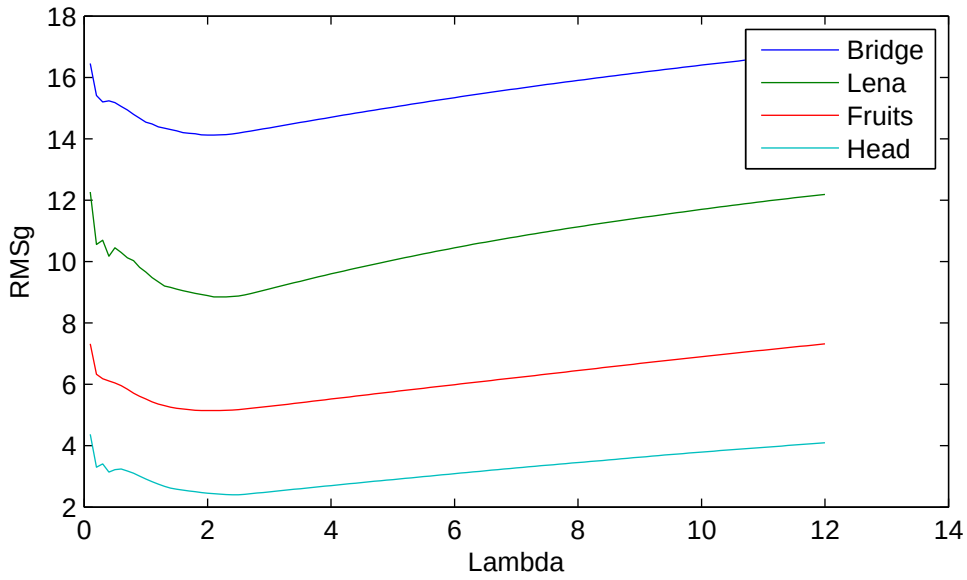


Figure 9: *2D reconstruction results for varying values of λ. 20 percent sample points are used.*

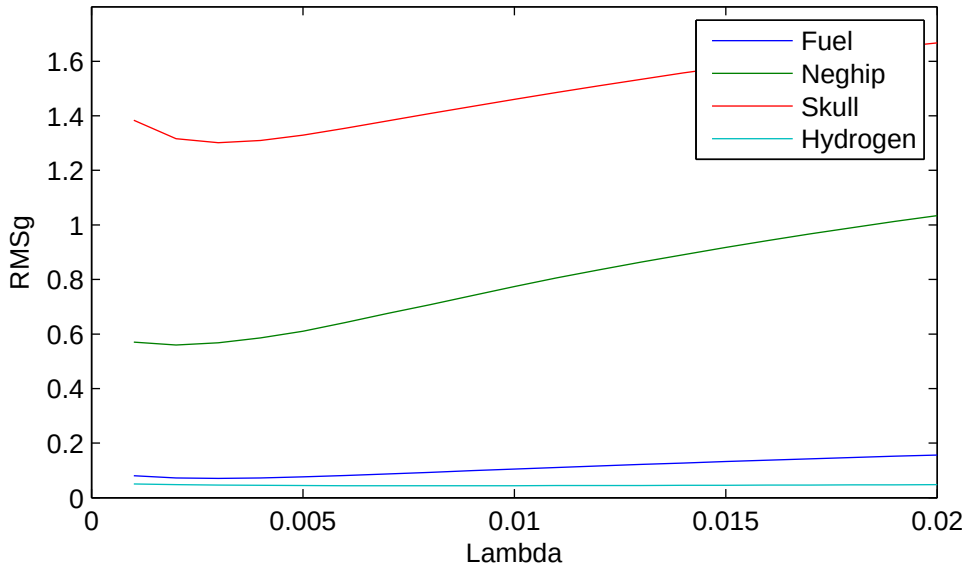


Figure 10: *3D reconstruction results for varying values of λ. 20 percent sample points are used.*

10.3 Varying sample point percentages

The more information is present about the dataset, the more precise the reconstruction can be. Figure 11 shows the reconstruction errors for the 2D datasets.
All datasets reach an almost constant error rate when increasing the number of sample points. These stationary rates are mostly affected by the used λ-value. Smaller values of λ result in smaller errors for high sample point percentages, but perform worse for percentages $< 50\%$.
Especially the complexity of the Bridge dataset can be seen here. While the other datasets reach a more or less stationary error rate at around 30 %, it takes about 50 % of the data to attain a constant rate for the Bridge dataset.

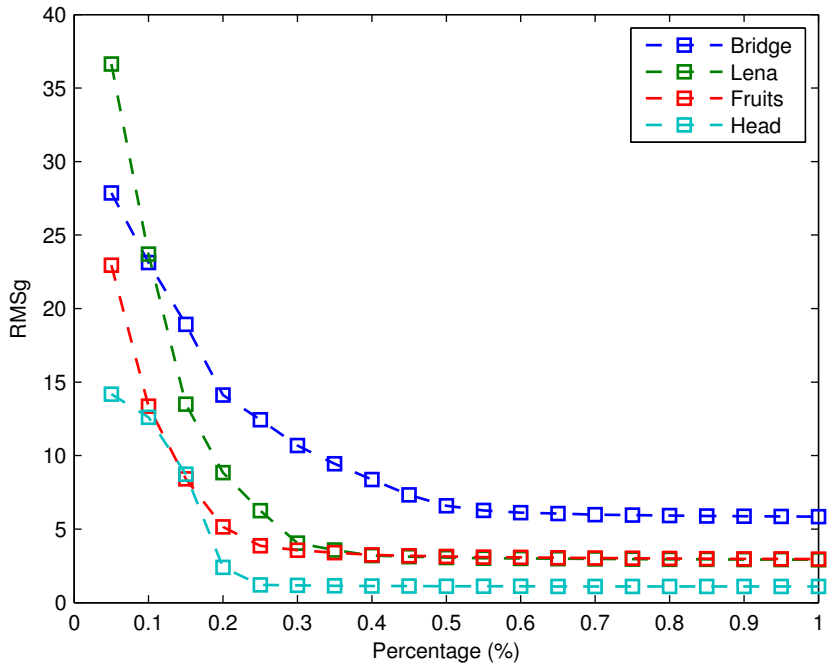


Figure 11: *Reconstruction results for varying sample point percentages.*

10.4 Multigrid results

10.4.1 Optimal multi grid parameters

There are a few parameters which affect multigrid performance, namely N, N_0, N_1, N_2, M, M_0, M_1, M_2 and P. N_0 denotes the number of initial iterations, N_1 and N_2 are the number of iterations before and after the

coarse-grid correction and N denotes the number of coarse-grid corrections. M, M_0, M_1 and M_2 are the parameters for the corresponding error system. P is the number of recursive steps and hence the size of the multigrid pyramid that needs to be set up.
Finding the optimal combination is not trivial and is mostly a tradeoff between reconstruction quality and speed. In general, the error system requires less iterations. A good value for P is around 2 to 3, depending on the reconstruction size. Figure 12 shows the reconstruction results for different values of N. It can be seen that while the reconstruction time rises rather linearly, the RMS_g error rates sink in a higher degree fashion.
Figure 13 graphs the error rates and reconstruction times for the Lena dataset with varying values of P. The reconstruction times rise about logarithmically due to the decreased complexities of the smaller resolution linear systems. RMS_g values drop quickly but then stagnate and can even get worse with increased numbers of resolution.

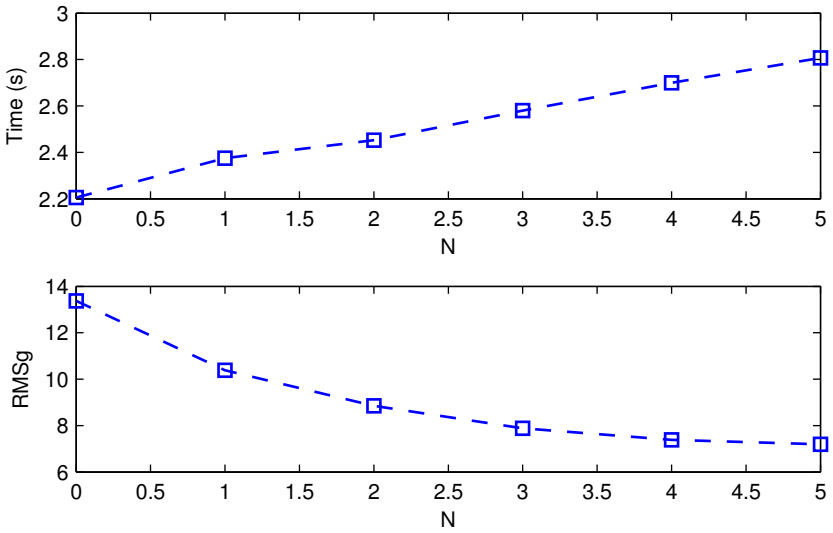


Figure 12: *Reconstruction with varying N values using the Lena dataset. Other parameters: $N_0 = 10$, $N_1 = 20$, $N_2 = 20$, $M = 2$, $M_0 = 10$, $M_1 = 5$, $M_2 = 10$ and $P = 2$*

10.4.2 Comparison with single grid

Using a multigrid approach offers great performance improvements for bigger datasets. However, because of the initial setup cost involved it is slower for small datasets. Table 4 shows the reconstruction of the Neghip dataset. The multigrid parameters were chosen to match the RMSg values of the single

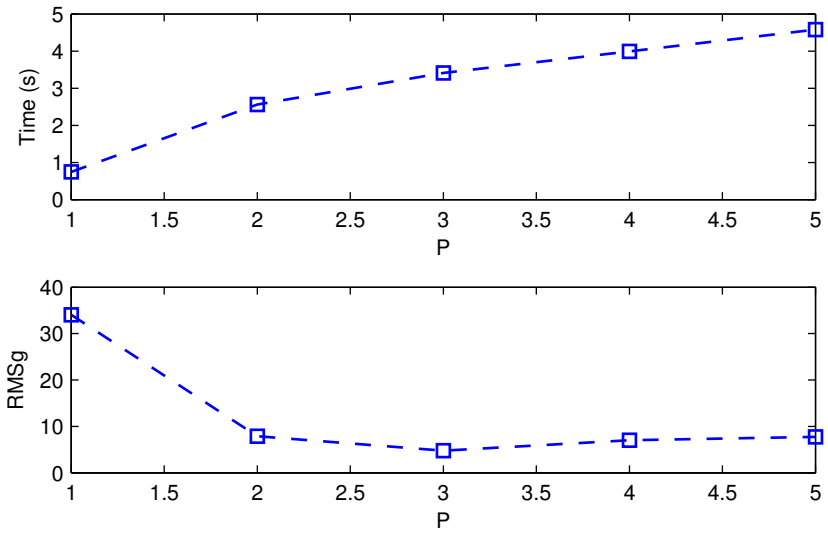


Figure 13: *Reconstruction with varying P values using the Lena dataset. Other parameters: $N = 3$, $N_0 = 10$, $N_1 = 20$, $N_2 = 20$, $M = 2$, $M_0 = 10$, $M_1 = 5$ and $M_2 = 10$*

grid reconstruction to get a better view on the reconstruction times. In this example, it is about 2,5 seconds faster.

Sample points:	**20 %**	**30 %**	**50 %**
Multigrid	2,11 s	2,12 s	2,13 s
Single grid	4,58 s	4,59 s	4,60 s

Table 4: *Comparison of reconstruction performance of single and multigrid method using the Neghip dataset. Multigrid parameters: $N = 3$, $N_0 = 10$, $N_1 = 10$, $N_2 = 10$, $M = 2$, $M_0 = 10$, $M_1 = 5$, $M_2 = 10$ and $P = 2$; number of single grid iterations: 100*

10.5 Visual results

Figure 14 shows the original Skull dataset, visualized with the implemented ray tracer. Figure 15 shows the result of the reconstruction from 20 percent of laplacian-selected data, rendered using the same settings (angle, transfer function, ...). In most parts, the reconstruction is not distinguishable from the original. However, areas like the teeth show differences. This is due to the high values located there in the original dataset. The regularization smoothes out the reconstructed data and high-valued peaks tend to get cut off.

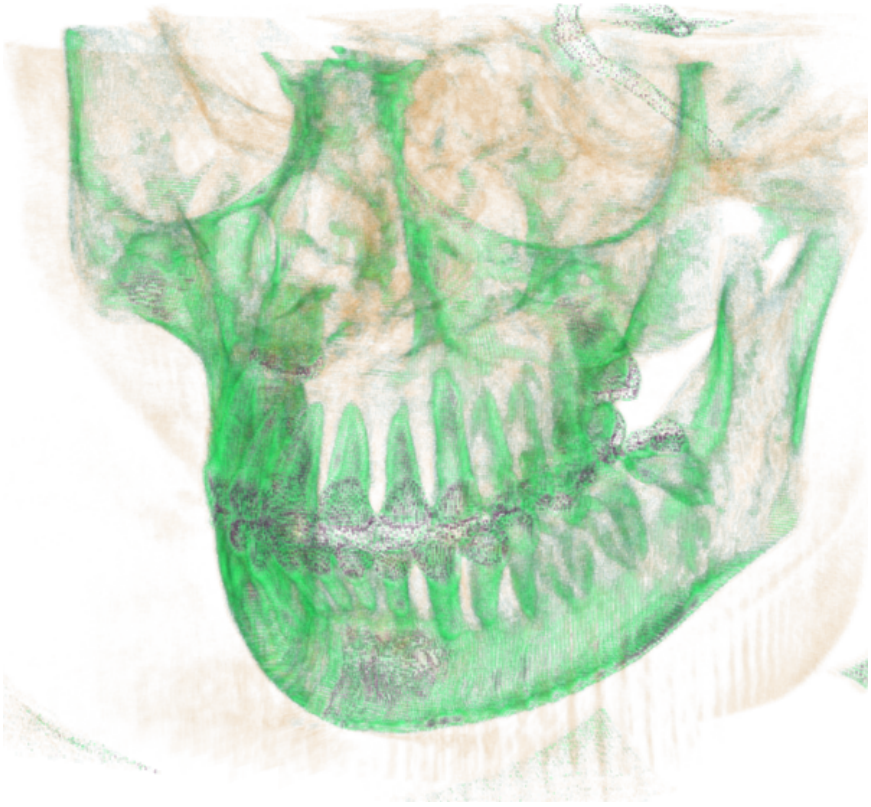


Figure 14: *Visualization of the original Skull dataset*

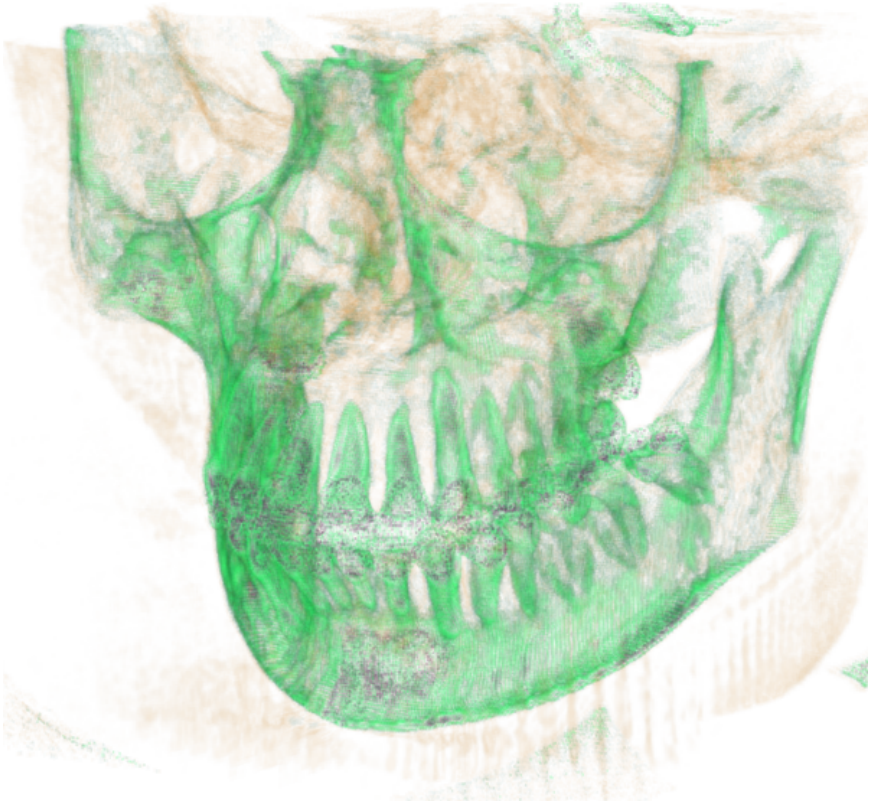


Figure 15: *Visualization of the reconstructed Skull dataset using 20 percent of the original data ($\lambda = 0.003$)*

10.6 Hardware comparisons

The GT220, which is used for all measurements until this point, is a rather low end consumer card. In this chapter, the implementation is tested out on another GPU, namely the GTX 285. Tables 5 and 6 show the differences in reconstruction time.

The GTX 285 has about 5 times the number of microprocessors of the GT 220. But other factors like the read and write bandwidths also play a role. Comparing the raw processing power of the devices and the reconstruction times, a speedup in the range of 300 % to 500 % can be witnessed, depending on the reconstruction size.

When realizing the reconstruction process, good parallelization was the main goal. The resulting implementation scales almost linearly with the amount of parallel processing power. It is expected that with even better hardware, faster reconstruction times are possible.

Sample points:	20 %	30 %	50 %
Nvidia GT 220	156,16 s	156,91 s	158,60 s
Nvidia GTX 285	28,70 s	29,00 s	29,83 s

Table 5: *Comparison of reconstruction times on different hardware using the Skull dataset. Multigrid parameters: N = 3, N0 = 2, N1 = 2, N2 = 2, M = 2, M0 = 2, M1 = 2, M2 = 2 and P = 2*

Sample points:	20 %	30 %	50 %
Nvidia GT 220	2,11 s	2,12 s	2,13 s
Nvidia GTX 285	0,85 s	0,88 s	0,89 s

Table 6: *Comparison of reconstruction times on different hardware using the Neghip dataset. Multigrid parameters:* $N = 3$, $N_0 = 10$, $N_1 = 10$, $N_2 = 10$, $M = 2$, $M_0 = 10$, $M_1 = 5$, $M_2 = 10$ *and* $P = 2$

10.7 Comparison to CPU based implementations

It is hard to compare GPU and CPU implementations of the same algorithms. They have different performance factors and sometimes even require completely different approaches to the same problem. Therefore, the following table 8 should be not be taken as a strict 1 to 1 comparison of the same algorithms. As an example, the CPU based implementations of Vuçini et al. [10] and Tech 2011 [5] both use the Gauss-Seidel technique as the preferred linear equation solver. For the GPU implementation however, this approach was determined to be impossible to parallelize up to a satisfying level and the Conjugate Gradient technique is chosen instead.

	CUDA based*		Tech2011**		Vuçini et al.***	
Dataset	**Times**	**RMS$_g$**	**Times**	**RMS$_g$**	**Times**	**RMS$_g$**
Engine 256×256×256	21,1 s	0,44	45,9 s	0,54	76,8 s	0,94
Tooth 256×256×160	25,7 s	0,45	74,0 s	0,41	112,8 s	0,57
CT Chest 384×384×240	69,4 s	0,34	192,9 s	0,39	304,8 s	0,6
Carp 256×256×512	63,2 s	0,32	219,1 s	0,38	343,8 s	0,36

Table 7: *Comparison table with CPU implementations by Vuçini et al. and Tech 2011.*
** Nvidia GTX 285, 2 GB memory*
*** Intel Core 2 Quad Q9400 CPU @ 2.67 GHz and 2 GB memory*
**** Intel Dual-core CPU @ 2.7 GHz, 6 GB memory*

Dataset	λ	**block-based**	**Times**	**RMS$_g$**
Bridge (512×512)	1,8	no	1,18 s	10,53
Lena (512×512)	2,1	no	1,20 s	5,08
Fruits (512×512)	2,1	no	1,18 s	4,42
Head (512×512)	2,3	no	0,64 s	1,99

Table 8: *Final results for the used image datasets. 20 percent of the original dataset are used, selected based on their laplacian value.*

Dataset	λ	block-based	Times	$\mathbf{RMS}_g$
Neghip (64×64×64)	0,002	no	2,56 s	0,496
Fuel (64×64×64)	0,0015	no	2,624 s	0,081
Hydrogen (128×128×128)	0,004	yes (2x1x1)	18,483 s	0,072
Skull (256×256×256)	0,002	yes (2x2x2)	156,42 s	0,264
Aneurism (256×256×256)	0,001	yes (2x2x2)	156,88 s	0,388

Table 9: *Final results for the 3D datasets extracted from the ground truth. 20 percent of the original dataset are used, selected based on their Laplacian value.*

Dataset	λ	block-based	Times	RMS
Oil (38×40×38)	0,003	no	1,037 s	0,13
Natural Convection (61×61×61)	0,001	no	3,512 s	1,136
Synthetic Chirp (64×64×64)	0,001	no	2,675 s	0,153
Flow Transport (44×58×58)	0,002	no	1,86 s	0,052
Fuel Injection (39×51×60)	0,001	no	1,745 s	0,362
Cooling Jacket (222×128×122)	0,02	yes (2x1x1)	118,64 s	1,121

Table 10: *Final results for the non-uniform 3D datasets. The sizes indicate the reconstruction grid's dimensions.*

10.8 Final results

Table 9 shows the final reconstruction times and RMS_g results for the 3D datasets created from their uniform ground truth. 20 percent of Laplacian points were selected. Most of the reconstruction parameters were set based on empirical knowledge. λ was set based on the findings in Chapter 10.2.
Table 10 presents the results for the non-uniform datasets where no ground truth is known. Instead of the RMS_g, the RMS error is evaluated.
Regarding reconstruction errors, the datasets with the least detail also show the lowest RMS_g results. Consequently, datasets with strongly oscillating or otherwise rapidly changing values show relatively high error results. These datasets also require more iterations, negatively affecting the reconstruction times.
Apart from this, the reconstruction grid's size is the most defining factor for the time it takes to reconstruct. Bigger datasets benefit from fully occupying the GPU most of the time. Smaller sets can't take advantage of the available parallel processing powers. Having to reconstruct in a block-based fashion also influences the performance in a negative way.

11 Conclusion

The methods developed by Arigovindan et al. [1] and Vuçini et al. [10] for reconstructing two- and three-dimensional non-uniform datasets and converting them to a uniform representation are carried out using traditional CPU implementation techniques. This work takes the existing knowledge of these works and transforms it into a CUDA based implementation.
By exploiting the similarity of cubic B-splines to the computationally expensive thin-plate splines, the reconstruction can be carried out much more efficiently. The interscale relation of B-splines of odd degree makes a multiresolution approach possible, resulting in further performance improvements. The linear system, which' size is dependent only on the reconstruction grid, is solved by a parallel Conjugate Gradient implementation. Bigger datasets require a block-based approach to overcome the relatively sparse amount of memory present in GPUs.
Most crucial for the performance is the amount of parallelization of the used algorithms. The performed tests show that the reconstruction method used can be implemented in an efficient manner on current graphics cards. Using the knowledge on CUDA and GPUs in general lead to a highly parallel realization that scales almost linearly with the available hardware resources (See tables 5 and 6). Keeping in mind the ever increasing amount of processing power present in current GPUs, the expected reconstruction times should further decrease as better GPUs become available.
While determining the optimal reconstruction parameters is possible on an empirical basis, this is not always feasible in practice, especially when the ground truth is not known. Some applications will require feasible parameters before performing a costly reconstruction. No methods for finding them in an efficient manner exist yet. Future works might be able to answer these questions satisfactory.

Appendix A

We define the 3D Laplacian kernel as follows:

$$Laplacian(V) = \frac{\partial^2 V}{\partial x^2} + \frac{\partial^2 V}{\partial y^2} + \frac{\partial^2 V}{\partial z^2} \tag{6}$$

where V represents the volume given as a 3D regular grid.

Appendix B

The root mean square (RMS) error is defined as follows:

$$RMS = \sqrt{\frac{\sum_i^M (F(x_i, y_i, z_i) - f_i)^2}{M}} \times \frac{100}{Max_{value}} \tag{7}$$

where F is the approximating function, f are the given values, Max_{value} is the maximum value in the given point set and M is the number of points. For the error estimation in the Laplacian datasets we will compute the global RMS error (RMS_g) in all regular points (including the points not retained in the Laplacian dataset, i.e., $M = N_x \times N_y \times N_z$).

References

[1] Muthuvel Arigovindan, Michael Shling, Patrick Hunziker, and Michael Unser. Variational image reconstruction from arbitrarily spaced samples: A fast multiresolution spline solution. *IEEE TRANSACTIONS ON IMAGE PROCESSING*, 14:450–460, 2005.

[2] R. Barrett, M. Berry, T. F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. Van der Vorst. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods, 2nd Edition*. SIAM, 1994.

[3] O. Deussen, C. Hansen, D. A. Keim, D. Saupe (editors, Yun Jang, Manfred Weiler, Matthias Hopf, Jingshu Huang, David S. Ebert, Kelly P. Gaither, and Thomas Ertl. Interactively visualizing procedurally encoded scalar fields, 2004.

[4] Yun Jang, Ralf P. Botchen, Andreas Lauser, David S. Ebert, Kelly P. Gaither, and Thomas Ertl. Enhancing the interactive visualization of procedurally encoded multifield data with ellipsoidal basis functions.

[5] Martin Kinkelin. Variational reconstruction and gpu ray-casting of non-uniform point sets using b-spline pyramids. Master's thesis, Institute of Computer Graphics and Algorithms, Vienna University of Technology, Favoritenstrasse 9-11/186, A-1040 Vienna, Austria, May 2011.

[6] Shankar Krishnan, Cludio T. Silva, and Bin Wei. A hardware-assisted visibility-ordering algorithm with applications to volume rendering. In *In Data Visualization (2001*, pages 233–242, 2000.

[7] Nelson Max, Peter Williams, Claudio Silva, and Richard Cook. Volume rendering for curvilinear and unstructured grids. In *International Journal of Imaging Systems and Technology*, page 2000, 2003.

[8] NVIDIA Corporation. *NVIDIA CUDA C Programming Best Practices Guide*, 2009.

[9] NVIDIA Corporation. *NVIDIA CUDA C Programming Guide*, 2010.

[10] Erald Vuçini, Torsten Möller, and Meister Eduard Gröller. On visualization and reconstruction from non-uniform point sets using b-splines, June 2009. 2nd Best Paper Award.

[11] Manfred Weiler, Martin Kraus, Markus Merz, and Thomas Ertl. Hardware-based ray casting for tetrahedral meshes. In *Proceedings of the 14th IEEE Visualization 2003 (VIS'03)*, VIS '03, pages 44–, Washington, DC, USA, 2003. IEEE Computer Society.