Article

Principles for an Implementation of a Complete CT Reconstruction Tool Chain for Arbitrary Sized Data Sets and Its GPU Optimization

Jürgen Hofmann *, Alexander Flisch and Robert Zboray

Center for X-ray Analytics, Empa, Swiss Federal Laboratories for Materials Science and Technology, Überlandstrasse 129, 8600 Dübendorf, Switzerland; alexander.flisch@empa.ch (A.F.); robert.zboray@empa.ch (R.Z.)
* Correspondence: juergen.hofmann@empa.ch

Abstract: This article describes the implementation of an efficient and fast in-house computed tomography (CT) reconstruction framework. The implementation principles of this cone-beam CT reconstruction tool chain are described here. The article mainly covers the core part of CT reconstruction, the filtered backprojection and its speed up on GPU hardware. Methods and implementations of tools for artifact reduction such as ring artifacts, beam hardening, algorithms for the center of rotation determination and tilted rotation axis correction are presented. The framework allows the reconstruction of CT images of arbitrary data size. Strategies on data splitting and GPU kernel optimization techniques applied for the backprojection process are illustrated by a few examples.

Keywords: computed tomography; CT reconstruction software; GPU-based reconstruction; bad center of rotation correction; data splitting techniques

1. Introduction

At Empa’s Center for X-ray Analytics, different types of Computed Tomography (CT) scanners are in use. They encompass a cone-beam CT scanner with sub-micrometer resolution, micro-CT scanners in the higher energy regime (up to 300 keV), X-ray phase contrast instruments and a high-energy CT system (up to 6 MeV) using a linear accelerator as an X-ray source. In this article, tools for the complete CT reconstruction chain are presented. They have been in use for many years for in-house CT reconstruction. The basic principles and the implementation of artefact reduction methods will be discussed. Furthermore, we will give insights into implementation and optimization strategies. We started the development of this in-house software for CT reconstruction to satisfy needs that could not be covered by commercial software. Since the CT data sets were growing over time with the increase in detector resolution from approximately $512 \times 512$ pixels to $4096 \times 4096$ pixels today by more than an order of magnitude, an according speed up in reconstruction time was also becoming mandatory. Therefore, we implemented the backprojection, the most time-consuming component, on a graphics processing unit (GPU) using NVIDIA’s CUDA toolkit [1]. This allows fast—a few minutes—backprojection for large-volume data sets ($\approx 2048^3$ voxels). In the last 10 years, we continuously integrate new modules capable of handling artefact reduction and geometry correction. Graphical user interfaces for all modules easily allow parameter configuration and the execution of the applications. Nevertheless, it is also possible to run the filtered backprojection headless, which allows batch processing. The reconstruction of phase-contrast CT will be described elsewhere. Nowadays several free or open source reconstruction tool kits are existing. TIGRE [2], ASTRA [3], RTK [4] and MuhRec [5] are prominent examples. TIGRE, ASTRA and RTK providing iterative solvers, which are capable of handling reconstruction problems that cannot be solved by filtered backprojection. These are, e.g., limited angle CT or the integration of physical models in the reconstruction procedure. Reviews for the different
methods are presented in [6,7]. Early examples for CT reconstruction implementation on the GPU are those from Fang Xu et al. [8] using GPU shaders and Holger Scherl et al. [9] using CUDA textures. However, the usability was limited due to the small GPU memory size (NVIDIA GeForce 8800 GT 512 MB). The majority of the tool kits today are using GPUs for process acceleration. Specific for our implementation is the automated determination of the center of rotation and the rotation axis tilt. These features are essential in practice for the reconstruction, especially with high magnification measurements using nano- and micro-cone-beam CT systems. Our implementation running under Windows is available as open source at: https://github.com/JueHo/CT-Recon, accessed on 15 December 2021.

2. Methods: Overview of the Reconstruction Framework

Figure 1 gives an overview of the reconstruction framework: the modules, the processes and the data flow. The next chapters focus on the method and implementation description of the two main parts of the CT reconstruction: the pre-processing of the raw data and the filtered backprojection. These include ring artefact reduction methods, automatized estimation of the rotation axis offset, tilted axis correction and a simple beam hardening correction. The basic principles for the Lak-Ram and Shepp-Logan filter design are explained. For the backprojection step we present details on the implementation of the Feldkamp–David–Kress algorithm (FDK) [10] on GPU. Furthermore a data-splitting algorithm is introduced, which enables the reconstruction of arbitrarily sized volumes.

![Figure 1. Overview: Workflow of CT reconstruction.](image-url)
2.1. Pre-Processing

Pre-processing includes several steps before executing the actual CT reconstruction with the filtered backprojection. It involves mandatory processing steps such as normalization and taking the logarithm of the normalized projection images. Others methods such as those for ring artifact suppression and beam hardening correction are optional. In the subsequent paragraphs, we explain all methods in detail.

2.1.1. Normalization and Beam Hardening Correction

Normalization:

Raw projection images have to be corrected by a reference image taken without the object, the flat-field image and a dark-field image acquired without X-ray radiation. The dark- and flat-field correction as shown in Equation (1) is normally performed during the data acquisition. Scanner systems typically provide internally an up-scaling to the original data range or a fluence correction. Therefore, we introduce the term fluence (integrated counts on the detector over the exposure time) $C_{fl}$ in Equation (1), which should make our further statements clearer:

$$I_{\phi i}^{proj}(x,y) = I_{\phi i}^{raw}(x,y) - I_{dark}(x,y)$$

with projection at angle $\phi_i \in [0, 2\pi]$.

We distinguish between two cases of up-scaling ($C_{fl}$) performed during data acquisition:

1. The data range after the normalization will be up-scaled to the level of the raw projections, e.g., in 16-bit representation, before normalization ($C_{fl}$ Equation (2)):

$$C_{fl} = \frac{mean(I_{flat,raw}(x,y)) - I_{dark}(x,y))}{I_{flat,raw}(x,y) - I_{dark}(x,y)}$$

2. An additional measurement device will gather the fluence of each projection ($C_{fl}$ Equation (3)):

$$C_{fl} = I_{\phi i}^{fl} - mean(I_{dark}(x,y))$$

In our implementation, we have three options to consider the different cases (see Figure 2, options yellow marked):

1. No correction. The projections are already fluence corrected.
2. The projections are only up-scaled, and it is possible to select a background ROI for a post-fluence correction. Then, we use Equation (4) for the correction. For that, a region of interest (ROI) of the background is taken for every projection at the same position. It is used to perform a fluence correction of the current projection by the mean value of the pixels values within the ROI. The selection of the ROI is performed interactively, as illustrated in Figure 2. The ROI should not interfere with the object, which needs to be verified by stepping through all projections:

$$I_{\phi i}^{corr}(x,y) = \frac{I_{\phi i}^{proj}(x,y)}{mean(roi_{\phi i})}$$

3. The projections are only up-scaled, but it is not possible to select a background ROI and perform a fluence correction. This might be because, e.g., projections are completely filled by the object. Then, we use Equation (5) to re-scale the projections to make the gray values comparable to those corrected by Equation (4):

$$I_{\phi i}^{corr}(x,y) = \frac{I_{\phi i}^{proj}(x,y)}{mean(I_{flat,raw}(x,y) - I_{dark}(x,y))}$$
Obviously for the in-house scanner where all the images (projection, flat and dark) are available, the user can provide those to our package and choose from the above option and use the respective equations. Our package flexibly supports all these input image options.

Beam hardening correction (BHC):

Bremsstrahlung-based X-ray sources typically used in laboratory CT systems have a polychromatic energy spectrum. Beam hardening occurs when an X-ray beam consisting of polychromatic energies passes through an object, and the sample material attenuates more pronounced lower energy photons. The mean value in the X-ray spectrum reaching the detector is shifted to higher energy. The result is a non-linearity of the attenuation versus the material thickness. In Figure 3, the artefact caused by beam hardening is visible in higher gray values in the outer region and lower gray values in the center of the object’s tomogram, as the lower energy photons preferentially become attenuated over longer path lengths. We chose a polynomial approach \[11,12\] for the correction of beam hardening, which is fast and delivers good results in practice, especially applied to cupping artifacts. The function-based correction method—e.g., with polynomials—linearizes the attenuation values in the projections. Equation (6) shows the expression used for the correction. For the correction of severe beam hardening artifacts such as photon starvation, an iterative correction algorithm [13] is often more suitable:

\[
f(x) = a \cdot x + b \cdot x^c \quad x: \text{image gray value and } a, b, c: \text{empirical coefficients}
\]

The parameters used for BHC are based on experience. Typically, values for \(c\), the most sensitive parameter, are in the range of \([2.0, 3.0]\).

2.1.2. Ring Artifact Reduction

Typical ring artifacts can be observed in the CT slice in Figure 4a. We differentiate between two main types of ring artifact sources. The first is caused by malfunctioned detector pixels creating severe artifacts. The second is caused by the nonuniform sensitivity of the detector pixels, which is less severe, but still induces significant ring artifacts in the reconstructed images. We implemented two methods to mitigate ring artifacts.
The first method is based on median filtering (MF), which reduces coarse ring artifacts induced by defect pixel values. As input for this method, we define a threshold value \( \sigma_{th} = n \times \sigma \) for the gray value outliers as a multiple of the standard deviation. In the projection stack, structures and edges vary from image to image contrary to the defect pixels, which stay on the same coordinates. Therefore, we sum up all projection images pixel-wise into an image \( I_{su} \). This reduces the influence of actual edges in \( I_{su} \). To detect the outliers, we build the difference image \( I_{diff} \) of \( I_{su} \) and the median blurred of \( I_{su} \). We build a z-score [14] image from \( I_{diff} \) and create an outlier image mask \( I_{mask} \), as described in Algorithm 1. The mask \( I_{mask} \) is used in a median filtering procedure for the X-ray projections, as shown in Algorithm 2. This MF method mitigates coarse ring artifacts caused by defect pixel values and defect detector lines (See changes in images in Figure 4a,b). The algorithm utilizes functions from the Open Source library OpenCV [15] for its implementation.

**Algorithm 1: Find outliers in X-ray image stack.**

**Input:** Image stack of X-ray projection, Threshold value \( \sigma_{th} \) of outliers, window mask win(w = width, h = height)

**Output:** Image mask \( I_{mask} \) with outlier pixels

\[
\hat{P} = \frac{1}{n} \sum_{i=1}^{n} P_i
\]

\( I_{su} = \text{MedianBlur}(\hat{P}, \text{win}(w, h)) \)

\( I_{diff} = | \hat{P} - I_{su} | \)

\( I_{z-score} = \frac{I_{diff} - m}{s} \)

\( I_{mask} = \text{threshold}(I_{z-score}, \sigma_{th}, \text{BINARY_MODE}) \)

**return** \( I_{mask} \)
Algorithm 2: Median filtering of defect pixels in X-ray projections.

**Input:** Image stack \( I_{\text{stack}} \) of X-ray projection, Image mask \( I_{\text{mask}} \) with outlier pixels, window mask \( \text{win}(w = \text{width}, h = \text{height}) \)

**Output:** Smoothed X-ray image projection stack \( I_{\text{corr}} \) with outlier correction

\[
cols \leftarrow I_{\text{mask}}.\text{column} \\
\text{rows} \leftarrow I_{\text{mask}}.\text{rows} \\
\text{Coordinate coor} \leftarrow \text{Vector outlier_list} \\
\]

// Get the coordinates of the outlier pixels
for \( y \leftarrow 0 \) to \( \text{rows}-1 \) do
  for \( x \leftarrow 0 \) to \( \text{cols}-1 \) do
    if \( I_{\text{mask}}(x, y) > 0 \) then
      Push \( \text{coor}(x, y) \) to outlier_list

// Loop over Projections \( P_i \) in X-ray image stack and // iter list of outliers and replace outlier pixels // at coordinate with median value in window win
size \leftarrow \text{count of projections in } I_{\text{stack}}
for \( i \leftarrow 0 \) to \( i = \text{size} \) do
  \( P_i \leftarrow I_{\text{stack}}[i] \)
  for \( \text{coor} \leftarrow \text{pop outlier_list to list_end} \) do
    Apply Median at \( \text{coor} \) using \( \text{win} \)
  \( I_{\text{corr}}[i] \leftarrow P_i \)
return \( I_{\text{corr}} \)

---

**Figure 4.** Tomogram of a coffee bean. Applying outlier and ring artifact reduction methods: (a) without any corrections (two to three bad pixels next to each other cause severe ring artifacts); (b) with MF method only; (c) with DLR correction only; (d) with MF and DLR method together.
The second method is proposed by Y. Kim et al. [16] and is based on ring artifact correction using detector line ratios (DLR). The objective of this method is the build-up of a sensitivity correction matrix (SCM) from the projection image stack. The SCM is used to correct the pixel values of all projections \( P(\phi) \) according to the determined detector sensitivity. The implementation in our pre-processing application is following closely the description in the paper.

The best results in ring artifact reduction can be accomplished by combining both methods, first applying the outlier correction method and afterwards the process projections detector sensitivity equalization method. An example is shown in Figure 4. Only applying the MF method (Figure 4b) leads to a reduction of ring artifacts but still leaves significant rings visible. The DLR method (Figure 4c) alone is not capable of handling defect pixels well and introduces additional artifacts. Both methods applied together (Figure 4d) show good results in ring artifact suppression.

2.2. Weighting and Filtering of Projections

Weighting: For cone-beam CT scanners, it is necessary to weight the pixel gray values in the projection images according to their geometry. We use the geometry notation of Figure 5 for the weighting shown in Equation (7). When the rotation axis is not centered, the pixel positions in \( X_p \) must be shifted by the offset value determined in Section 2.4.1:

\[
W(X_p, Y_p) = \frac{SDD}{\sqrt{X_p^2 + Y_p^2 + SDD^2}}
\]

(7)

Weighting of pixel values in a projection.

Filtering: The ramp (Ram-Lak) filter is an integral part of the filtered backprojection and is therefore mandatory. It has its mathematical origin in the Jacobian determinant for the Polar to Cartesian transformation, which has to be applied for the derivation of the backprojection algorithm [10]. The discrete Ram-Lak (Equation (8)) and Shepp and Logan filter (Equation (9)) are integrated in the reconstruction framework. \( \Delta s \) represents the sampling interval. The derivation of the discretized Ram-Lak and Shepp-Logan filters can be found in [10,17]. The discrete filters are implemented in the spatial domain to avoid constant offset (DC) errors. This is because the digital measurement of the projections is band limited due to discrete data sampling. A non-discrete filter would cause a DC offset error, which cannot be completely eliminated by zero padding [10]. The convolution of the filter kernel with the projections are processed in Fourier space due to performance reasons. Filtering is applied row wise. Filter and projections are zero padded. The padding length is at least \( N_f + N_p - 1 \), where \( N_f \) is the filter length and \( N_p \) is the projection length. This avoids interference errors are caused by acyclic convolution.

\[
H_{RL}(n\Delta s) = \begin{cases} 
1 & \text{if } n = 0, \\
0 & \text{if } n \text{ is even } (n \neq 0), \\
-\frac{1}{(n\pi\Delta s)^2} & \text{if } n \text{ is odd}
\end{cases}
\]

(8)

Discrete Ramachandran and Lakshminarayan filter kernel in spatial domain.

\[
H_{SL}(n\Delta s) = -\frac{2}{(n\pi\Delta s)^2} \frac{1}{4n^2 - 1}
\]

(9)

Discrete Shepp and Logan filter kernel in spatial domain.
2.3. Backprojection

In our framework, we implemented the voxel-driven method for backprojection. A ray is going from the X-ray source through the center of the voxel under reconstruction and is intersecting the projection plane afterwards. Code Listing A1 shows the implementation. The interpolated gray value at the intersection point is taken to sum up the voxel value. To obtain sub-pixel accuracy, bi-linear interpolation of the gray values in the neighborhood of the intersection point is performed. We provide two implementations for the bi-linear interpolation. The first is a fast hardware accelerated method that uses CUDA 2D texture fetching function tex2DLayered(). This method uses 9-bit fixed point format with 8 bits of fractional value [18], which gives the best accuracy for values near 1.0. For better accuracy in case of values far from 1.0, we implemented an interpolation based on a 32 bit floating point function—tex2DLayeredHighPrec()—running in a CUDA kernel function fdk_kernel_3DW_HA(). For details on implementation, see source code Listing A2. Using function tex2DLayeredHighPrec() decreases backprojection speed approximately by a factor of two. The backprojection instruction is shown in Equation (10). Voxels are reconstructed slice-wise $S_i$ as seen in Figure 5. The input data for the reconstruction are the weighted and filtered projections as described in Section 2.2.

$$V'(x_i, y_i, z_i) = \sum_{\varphi=0}^{2\pi} \frac{SCD^2}{(SCD - z_i(\varphi))^2} I'_\varphi(X_p, Y_p)$$

(10)

$z_i(\varphi)$: Projection on z of voxel $V'_i$ in volume coordinates and an angle $\varphi$.

$I'_\varphi(X_p, Y_p)$: Weighted and interpolated pixel value of a projection at the angle $\varphi$.

Discrete backprojection (see Figure 5 and details on implementation in Listing A3).

Figure 5. Voxel-driven backprojection.

One main feature of our framework is the capability to reconstruct tomograms from arbitrarily sized raw data sets. Neither the input data nor the reconstructed volume have to fit into CPU or GPU memory. We achieve this by splitting input data and reconstruction volume size according the available hardware resources. For that, the actual free CPU and GPU memory must be estimated and compared with the required resources. Figure 6 shows how to split the reconstruction volume so that every chunk fits into the memory. This partitioning is possible because every voxel can be reconstructed independently from each other. For the backprojection, the projection data are uploaded to 2D layered textures on the GPU. However, since the GPU memory compared to the CPU memory is typically much more limited, the complete projection dataset usually does not fit into the GPU memory. The backprojection runs over all projections from $0^\circ$ to $360^\circ$. Therefore,
the complete backprojection process of a voxel can be divided into a sequence of sub-backprojections using batches of projections which are fitting into the GPU memory (see Figure 7). The Algorithm 3 shows the workflow of the sub-divided reconstruction process in more detail.

Figure 6. Splitting reconstruction volume into chunks, which are small enough to fit into CPU memory.

Figure 7. Depending on the available GPU memory and the maximal GPU texture size, the projection stack used for the reconstruction needs to be split.

2.4. Geometry Correction

Because data acquisition with CT scanners always contains errors regarding the geometry, post corrections have to be applied for accurate reconstruction results. In our application, we integrated the correction of the horizontal offset of the rotation axis and the axis tilt. The presented algorithm can also be applied for other geometry corrections, e.g., a slanted detector.
Algorithm 3: Backprojection of arbitrary data size. Reconstruction workflow for the partitioning of input data not fitting in CPU or GPU memory.

**Input:** Total number of slices in volume $N_{stot}$, Total number of projections in volume $N_{ptot}$, Weighted and filtered projection stack: $N_{ptot} \times P_{filt}$, Size of projection stack $SZ_{ps}$, Size of reconstruction volume $SZ_{vol}$, Fraction value for CPU (host) memory usage: $CPU_{MF}$, Fraction value for GPU memory usage: $GPU_{MF}$

**Output:** Reconstructed volume

\[
\begin{align*}
M_{CPU} &= \text{GetAvailableCPUMem()} \quad // \text{Available CPU (Host) memory} \\
MC_{use} &= M_{CPU} \cdot CPU_{MF} \quad // \text{Scale memory usage} \\
N_{slice} &= \text{Calculate number of slices in chunk from memory } MC_{use} \\
\end{align*}
\]

\[
\begin{align*}
M_{GPU} &= \text{GetAvailableGPUMem()} \quad // \text{Available GPU memory} \\
MG_{use} &= M_{GPU} \cdot GPU_{MF} \quad // \text{Scale memory usage} \\
N_{proj} &= \text{Calculate number of projections in batch from memory } MG_{use} \\
\end{align*}
\]

// Calculate the number of volume chunks, // and number of the residual slices
\[
NC = \frac{N_{stot}}{N_{slice}} \\
RC = N_{stot} \mod N_{slice}
\]

if $RC > 0$ then

- hasChunkRemainder = true
- $NC = NC + 1$

// Calculate the number of projection batches, // and number of the residual projections
\[
NB = \frac{N_{ptot}}{N_{proj}} \\
RB = N_{ptot} \mod N_{proj}
\]

if $RB > 0$ then

- hasBatchRemainder = true
- $NB = NB + 1$

// Loop over CPU memory chunks
for $i_c \leftarrow 1$ to $NC$ do

- if hasChunkRemainder = true AND last memory chunk then
  \[
  N_{slice} = RC
  \]

for $i_b \leftarrow 1$ to $NB$ do

- if hasBatchRemainder = true AND last projection batch then
  \[
  N_{proj} = RB
  \]

Upload projections to GPU texture memory for $i_s \leftarrow 1$ to $N_{slice}$ do

- (Re-)Copy host memory for backprojection to device
- Run backprojection on CUDA kernel
- Copy to host memory

Write chunk to disk

return Reconstructed Volume

2.4.1. Out of Center Correction for the Rotation Axis

A rotation axis out of center leads to the artifacts seen in Figure 8a. The determination of the horizontal rotation axis shift is performed on reconstructed tomograms. For a free selectable slice in the tomogram stack (best near the central slice, because of the absence of cone-beam artifacts) a sequence of tomograms with increasing rotation axis offsets is reconstructed. The method to find the correct rotation axis offset is based on the autofocus
principle known from cameras and is adapted for X-ray CT [19]. By selecting the image with the sharpest edges, we obtain the correct axis offset. As we see in Figure 8a, an image without offset correction is blurred. The evaluation of the gray values of the gradient images is used to determine a qualifying sharpness score. For a successful application of the method in practice, some additional image processing steps are necessary, e.g., denoising. For implementation details, see the description of Algorithms 4 and 5. It is crucial to select a slice with structures and edges. In images with almost no structures or edges, the algorithm may fail, similar to autofocus cameras.

Algorithm 4: \( T_{score} \) Determine sharpness score.

**Input:** Tomo slice image \( S \), width of tomogram \( w \), margin factor \( m < 1 \)

**Output:** Sharpness score of tomo slice \( T_s \)

1. Calculate length \( L = m \cdot w \cdot 1/\sqrt{2} \) for processing window
2. Copy centred processing sub window image \( I_{sw} \) with size \( L \times L \)
3. Edge preserving smoothing of image \( I_s \) using Bilateral Filter
4. Calculate derivative image \( I_{sob} \) using Sobel Filter
5. // Determine sharpness score over all pixels
   \[ T_{score} = \sum I_{sob} \cdot I_{sob} \quad // \text{Make Pixel-Values} \geq 0 \quad (I_{sob}^2) \text{ and accumulate} \]
6. return \( T_{score} \)

Algorithm 5: \( X_m \) Find shift of rotation axis (uses Algorithm 4 “Determine sharpness score”).

**Input:** Filtered projections, offset interval in Pixel \([I_s, I_e]\), define offset iteration step \( \Delta x \)

**Output:** Offset \( X_m \) of the constructed slice with the sharpest edges

1. \( x_o \leftarrow I_s \)
2. \( S_M \leftarrow 0.0 \quad // \text{initialise maximal sharpness score} \)
3. \( n = |I_e - I_s|/\Delta x \)
4. // Reconstruct tomograms in interval range
   for \( i \leftarrow 0 \text{ to } n \) do
      \[ x_o = I_s + i \cdot \Delta x \]
      // Find rotation axis offset in interval range
      for \( i \leftarrow 0 \text{ to } n \) do
         \[ T[i] = T_i \]
         if \( S_i > S_M \) then
            \[ X_m \leftarrow x_o \]
      return \( X_m \)
Figure 8. Detailed view of a tomogram (Cheese with salt particles): (a) Left image without rotation axis correction; (b) right image rotation axis corrected with $-15.2$ pixels horizontal shift.

2.4.2. Rotation Axis Tilt Correction

If the rotation axis offset is not constant along all slices in the tomogram stack and we have a linear dependency of the offset and the position in the tomogram stack $z_i$, we speak of a tilted rotation axis. We can compensate the tilt by considering the functional dependency of the tomogram stack position and the axis offset in the slice. For that, we have to find the parameters of the equation $f(x)$:

\[ f(x) = a + m \cdot x \]  \hspace{1cm} (11)

with

\[
\begin{pmatrix}
  f(x_1) \\
  f(x_2) \\
  f(x_3) \\
  \vdots \\
  f(x_m)
\end{pmatrix},
\quad
X = \begin{pmatrix}
  1 & x_1 \\
  1 & x_2 \\
  1 & x_3 \\
  \vdots & \vdots \\
  1 & x_m
\end{pmatrix},
\quad
C = \begin{pmatrix}
  a \\
  m
\end{pmatrix}
\]

In Equation (11), the parameters $a$ and $m$ are linear with respect to the linear least square (LLS) problem. Thus, we have to solve the minimisation problem:

\[
C' = \arg \min_C O(C) \hspace{1cm} (12)
\]

where the objective function $O(C)$ is:

\[
O(C) = \| f(x) - XC \|^2 \hspace{1cm} (13)
\]

in Equation (13). The components of the vector $f(x)$ are the offsets at different positions in a projection image calculated with Algorithm 5. The least square fitting problem is solved with Eigen \cite{20}: A C++ template library for linear algebra.

2.5. Optimization

Backprojection is the most time consuming part of the CT reconstruction. Therefore, we gain the most speed-up if we focus the optimization there. Because the backprojection can be executed for all voxels independently, the process can easily be parallelized. In the past, multithreaded programs running on multiple CPUs were used to achieve high performance. For more than 10 years, programmable GPUs have been available which perfectly fit with their parallel pipeline processor architecture the needs for parallel programming. With them, it is possible to perform high-performance computing on a personal computer at low costs.
When implementing the backprojection on a GPU using NVIDIA’s CUDA programming extensions for C/C++ we are faced with three main performance bottlenecks having significant potential for optimization. These are: memory access, arithmetic operation and transfer of data from CPU to GPU memory and back. In the following, we will take a closer look at optimization strategies mitigating these bottlenecks.

Memory access: GPU’s memory model knows different types of memory. The biggest one, the global memory, allows access to the complete GPU memory from all threads of a kernel program (program running on the GPU) but has the highest latency. Only global memory provides sufficient memory to upload a stack of projection to GPU memory. The cache usage for textures can be enhanced if we have nearby spatial locality memory access by parallel executed GPU threads [21]. In our CT reconstruction, we achieve this by backprojecting a sub-region of voxels in a slice $S_i$ (see Figure 5) in a block of parallel thread. The projections into the detector plane of them laying all nearby and therefore profit from L1 caching effect. Additionally, we utilize CUDA’s high speed intrinsic bi-linear on-chip interpolation of 2D and 3D textures in our implementation.

Register memory is the fastest memory on GPU. The correct utilization of register variables is a further option to optimize the backprojection speed. Register memory is dedicated to single threads and is the scarcest memory resource on GPU. An over-usage will slow down the speed of application.

Arithmetic operations: The CUDA toolkit provides highly optimized mathematical functions. Because not all of those functions fulfill the IEEE precision standard, tests should be performed to validate the results. For the usage of intrinsic functions in our implementation, see source code Listing A1.

Memory transfer: We are using asynchronous memory copy (AMC) from CPU to GPU and vice versa in the backprojection to speed up the memory transfer. CUDA provides an Asynchronous Memory Copy API for this purpose. In conjunction with CUDA streams, memory copying from host to device and vice versa together with kernel execution can run overlapped in different streams regarding the host. The dependencies on kernel execution and memory copy within one stream will still remain. The kernel has to wait for data download and upload [22]. The usage of AMC requires the allocation of non-pageable resident memory on the CPU (CUDA function: cudaMallocHost()) and an additional non-default stream. Currently, the API does not support uploading of data to texture memory. However, we can use it for the transfer of the voxel data from CPU memory to GPU memory and back (see source code Listing A4).

3. Results and Benchmarks

Based on the results of the benchmarks, we will reveal the parameter settings dependencies regarding backprojection performance. The benchmarks were performed on a stack of images with 2048 pixel width, 2048 pixel height and 1800 projections with the following geometry parameters: SCD of 188.0 mm, SDD of 1017.34 mm and a detector pixel size of 0.2 mm. The resulting volume has the size $2048 \times 2048 \times 2048$ voxels. The configurations and results of the benchmarks are listed in Tables 1 and 2. We perform the benchmarks on two computing systems:

1. NVIDIA Quadro P4000 8 GB; Processor Intel(R) Xeon(R) W-2133 CPU 3.60 GHz, 3600 MHz, 6 cores, 128 GB;
2. NVIDIA Quadro RTX 8000 48 GB; Processor Intel(R) Xeon(R) Gold 6242 CPU 2.80 GHz, 2793 MHz, 16 Cores, 768 GB.

MAP defines the number of voxel slices processed in one backprojection kernel call. The upper limit of used register variables can be controlled with the compiler parameter “Max Used Register” RN. MAP does not only controls how many slice are copied for a kernel call, it is also used as unrolling parameter in the backprojection kernel (source
Listing A3). Because unrolling unwinds the loop body, it also increases register variables usage. Therefore, MAP, together with RN, must carefully be tuned. This is because when the register memory resource is becoming exhausted, the register variables will automatically be converted (spilled out) to local variables. Local variables are orders of magnitude slower than register variables. This will lead to a performance decrease. Furthermore, with MAP we can control the number of kernel calls and subsequently the total number of memory copies.

Table 1. Benchmarks types. Asynchronous memory copy is enabled when two streams are activated.

| Benchmark | # of Streams | # Slices Copied to Kernel | Max Used Register |
|-----------|--------------|----------------------------|-------------------|
| B1        | default      | 1                          | 31                |
| B2        | default      | 1                          | 63                |
| B3        | default      | 4                          | 31                |
| B4        | default      | 4                          | 63                |
| B5        | default      | 8                          | 31                |
| B6        | default      | 8                          | 63                |
| B7        | 2            | 1                          | 31                |
| B8        | 2            | 1                          | 63                |
| B9        | 2            | 4                          | 31                |
| B10       | 2            | 4                          | 63                |
| B11       | 2            | 8                          | 31                |
| B12       | 2            | 4                          | 63                |
| B13       | 2            | 4                          | 36                |

There is a maximum upper limit on non-pageable memory allocation, which cannot be controlled by the application software using Nvidia’s Cuda toolkit SDK function cuda-HostAlloc() and is on a Windows-based system per default roughly 25% of the CPU memory. The default maximum limit can be changed by operating-system-specific functions. This should be done carefully, not to compromise functionality of the system, especially on multi-user systems. With the default limit, we have to split for system 1 the CPU memory stack into two parts. Therefore, the projections have to be copied twice to texture memory. The projection stack itself must be split into five batches for the backprojection and the kernel is called five times more than for system 2. System 2 needs no split at all, neither for the CPU nor the GPU memory. Thus, the performance improvement potential of AMC is higher for system 1 than for those for system 2. For system 1, with AMC alone it is ≈ 38%. Together with register usage tuning, it is ≈ 48%. For system 2, with AMC alone it is ≈ 25%. With both, it is ≈ 32%. The timing results of benchmarks B4 and B10 are showing some anomalies for system 2, which are reproducible, but the exact reason is unclear.

In summary:

We studied the performance dependency of the backprojection for the parameters with the highest impact. We can conclude that the results of a parameter optimization retrieved on specific computer system cannot directly transferred to another with a different hardware configuration. Nonetheless, we can identify trends for optimal parameter configuration independent of the hardware: Enable AMC and reduce memory copy to/from GPU by uploading (downloading) multiple voxel slices. Maximize register variable usage by loop unrolling but avoid over-usage of registers. By applying the benchmarks to both available computer systems, we found optimization parameters working on both systems as a compromise quite well with AMC copy in batches of four slices to GPU memory, two streams and RN = 36.
Table 2. Processor time of backprojection for different benchmarks. Averaged over four measurements.

| Graphics Card    | Benchmark No. | Execution Time [s] |
|------------------|---------------|--------------------|
| NVIDIA Quadro P4000 | B1            | 264.6              |
| NVIDIA Quadro P4000 | B2            | 269.2              |
| NVIDIA Quadro P4000 | B3            | 200.4              |
| NVIDIA Quadro P4000 | B4            | 200.3              |
| NVIDIA Quadro P4000 | B5            | 186.4              |
| NVIDIA Quadro P4000 | B6            | 188.0              |
| NVIDIA Quadro P4000 | B7            | 223.5              |
| NVIDIA Quadro P4000 | B8            | 224.7              |
| NVIDIA Quadro P4000 | B9            | 148.4              |
| NVIDIA Quadro P4000 | B10           | 144.4              |
| NVIDIA Quadro P4000 | B11           | 139.5              |
| NVIDIA Quadro P4000 | B12           | 140.2              |
| NVIDIA Quadro P4000 | B13           | 140.1              |
| NVIDIA Quadro RTX8000 | B1          | 101.6              |
| NVIDIA Quadro RTX8000 | B2         | 97.2               |
| NVIDIA Quadro RTX8000 | B3          | 86.3               |
| NVIDIA Quadro RTX8000 | B4          | 125.4              |
| NVIDIA Quadro RTX8000 | B5          | 98.1               |
| NVIDIA Quadro RTX8000 | B6          | 89.0               |
| NVIDIA Quadro RTX8000 | B7          | 91.3               |
| NVIDIA Quadro RTX8000 | B8          | 87.3               |
| NVIDIA Quadro RTX8000 | B9          | 73.9               |
| NVIDIA Quadro RTX8000 | B10         | 115.4              |
| NVIDIA Quadro RTX8000 | B11         | 86.8               |
| NVIDIA Quadro RTX8000 | B12         | 76.5               |
| NVIDIA Quadro RTX8000 | B13         | 68.2               |

4. Discussion

In this article, we presented a tool chain which covers all necessary steps for cone-beam CT reconstruction from raw projection images. Furthermore, methods for artifact reduction and geometric error correction were introduced. The work is focused on the analytical filtered backprojection, which is a fast and reliable method for CT scans without severe artifacts for a sufficient amount of projections. Although iterative reconstruction solvers enable new opportunities to resolve problems which are hard to solve with analytical methods, the latter are still the most frequently used method in every day practice, especially if very large datasets are involved. Examples of where iterative methods are beneficial include heavy starvation artifacts, reconstruction techniques using additional models (CAD, physical, noise, . . . ), reconstruction from real time CT with few projections and non-uniform distributed projections. However, the main disadvantages of iterative reconstructions algorithms are low speed, slow or not guaranteed convergence and the complexity for parallelization of big datasets on commonly used computers with limited GPU resources (none High Performance Cluster HPC).

Our analytical CT reconstruction tool chain is very efficient and can be applied to arbitrarily large sized projection datasets. We have extensively benchmarked the tool and have shown generic trends for optimal hardware parameter configuration to boost the performance. Although not discussed explicitly, an implementation of the backprojection for a multi CPU and GPU system is straightforward. Algorithm 3 can be used as a basis.

As we have demonstrated, parameter optimization is an expensive task and hardware dependent. Therefore implementing auto-tuning of the GPU application parameters would be beneficial for a more simple adaption to different hardware systems. Examples for GPU auto-tuning can be found in [23,24]. Optimization by profiling is a further topic not discussed yet but was performed. It is summarized in Appendix A.
Author Contributions: Conceptualization, J.H.; methodology, J.H.; software, J.H.; validation, A.F. and R.Z.; writing—original draft preparation, J.H.; writing—review and editing, A.F. and R.Z. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: Not applicable.

Conflicts of Interest: The authors declare no conflict of interest.

Abbreviations
The following abbreviations are used in this manuscript:

AMC Asynchronous memory copy
BHC Beam hardening correction
CT Computed Tomography
DLR Detector line ratios
FDK Feldkamp, Davis and Kress reconstruction algorithm
GPU Graphics processing unit
MF Median Filter
ROI Region of interest
SCD Source rotation center distance
SCM sensitivity correction matrix
SDD Source detector distance SDD

Appendix A. Profiling Results

Figure A1 shows the results of profiling the backprojection GPU code using the application NVIDIA Nsight Compute Version: 2021.1.1.0 on the computer system 1 (see Section 3). To enable the profiling of benchmark B13, we had to down-sample the input data by a factor of two and reduce the reconstructed volume size to $1024 \times 1024 \times 1024$ voxels due to the computing overhead of the profiling tool Nsight Compute.

In the following, we will give a brief interpretation of the result.

As discussed in Section 2.5, crucial for fast texture memory access is the reuse of the L1 cache. The result for “L1/tex Cache compute utilization” of 94% and a “texture hit rate” of 98% shows that we achieved a high reuse rate. CUDA’s programming model defines a thread hierarchy (see details in [25]). A grid contains thread blocks which are hosts for the threads. These parameters are used for the kernel launch. We configured the parameters that the kernel runs with the maximal number of 1024 threads per block. This leads to an Occupancy value of 96%, which indicates a high computing workload of the GPU processor.
Appendix B. Source Code Listing

The essential source code of the backprojection is presented in the following. It is extracted from the original code and is written in the CUDA specific C++-language.

In Listing A3, we present the code of the default backprojection kernel function fdk_kernel_3DW() using Nvidias tex2DLayered() function for texture fetching and interpolation. For the other kernel functions, please visit the source code. MAP is the number voxel slices, which is loaded for one kernel call and is used as an unrolling parameter.

Listing A1. Function to retrieve intersection point in projection and determine voxel weight.

```c
__forceinline__ __device__ void GetInterpolationPoint(
  const VolType &voxel, Point &ptInterpol,
  const float &phi, float &weight,
  float &winOrigX, float &winOrigY, float &winOrigZ)
{
  float x, ry, z, dso, voxSz;
  dso = fdkConst.distSrcObj;
  voxSz = fdkConst.voxSize;
  // translate co-ordinates from origin left upper edge to centre of image
  x = voxel.x - fdkConst.offsetX + winOrigX;
  ry = fdkConst.offsetY - voxel.y - winOrigY;
  z = voxel.z - fdkConst.offsetX + winOrigZ;
  float co, si;
  __sincosf(phi,&si,&co);
  float rx = __fmaf_rn(x,co,z*si);
  float rz = __fmaf_rn(-x,si,z*co);
```

Figure A1. Backprojection kernel profiling using Nsight Compute on Nvidia Quadro RTX 8000.
Listing A2. Bi-linear interpolation with 32 bit float precision; the used fast intrinsic __fmaf_rn() is IEEE compliant.

void tex2DLayeredHighPrec(tex2DLayeredHighPrec, tex, &x_in, &y_in, int layer)
{
    int ix = floor(x_in);
    int iy = floor(y_in);
    float x = x_in - ix;
    float y = y_in - iy;

    float v00 = tex2DLayered(tex, ix, iy, layer);
    float v10 = tex2DLayered(tex, ix + 1, iy, layer);
    float v11 = tex2DLayered(tex, ix + 1, iy + 1, layer);
    float v01 = tex2DLayered(tex, ix, iy + 1, layer);

    float r1 = __fmaf_rn(x, v10, __fmaf_rn(-v00, x, v00));
    float r2 = __fmaf_rn(x, v11, __fmaf_rn(-v01, x, v01));

    return __fmaf_rn(y, r2, __fmaf_rn(-r1, y, r1));
}

Listing A3. Source code for default backprojection kernel function.

void fdk_kernel_3DW(float *d_backProj, // pointer to voxel slice
     int y_c, // current voxel-slice
     int cuProjBlockIdx, // current projection block index to use
     int volWidth, // width of voxel volume
     int volDepth, // depth of voxel volume
     float xOffset, // rotation axis offset
     float yOffset, // not used yet (=0)
     float winOrigX, // x pos of voxel volume
     float winOrigY, // y pos of voxel volume
     float winOrigZ) // z pos of voxel volume
{
    for (int i = 0; i < MAP; i++)
    {
        vst[i] = 0.0f;
    }

    // map from threadIdx/BlockIdx to voxel position
    int x = threadIdx.x + blockDim.x * blockIdx.x;
    int z = threadIdx.y + blockDim.y * blockIdx.y;

    Point interpolPoint;
VolType vox;

float phi; // rotation angle
// value from middle of voxel
vox.x = (float)x + 0.5f;
vox.y = (float)y_c + 0.5f;
vox.z = (float)z + 0.5f;

int volWidth_loc = volWidth;
int volDepth_loc = volDepth;
float xOffset_loc = xOffset;
float yOffset_loc = yOffset;
float discr = sqrtf((x-volWidth_loc/2.0f)*(x-volWidth_loc/2.0f)
    + (z-volWidth_loc/2.0f)*(z-volWidth_loc/2.0f));
if (x < volWidth_loc && z < volDepth_loc && discr < volWidth_loc/2.0f)
{
    float l_weight = 0.0f;
    // unroll loop to improve register variables usage
    #pragma unroll MAP
    for (int i = 0; i < fdkConst.projProcSize; i++)
    {
        phi = fdkConst.angleIncr*(cuProjBlockIdx*fdkConst.projProcSize + i);
        for (int k = 0; k < MAP; k++)
        {
            // get interpolation point on projection plane
            // and do weighting regarding voxel position
            // in reconstruction volume
            GetInterpolationPoint(vox, interpPoint, phi, l_weight, winOrigX, winOrigY, winOrigZ);
            // get gray value from projection
            // using hardware based texture interpolation
            vst[k] += tex2DLayered(texSinoLay, interpPoint.x+xOffset_loc, interpPoint.y+yOffset_loc, i)*l_weight;
            vox.y = vox.y + 1.0f;
        }
    }
    vox.y = (float)y_c + 0.5f;
    for (int k = 0; k < MAP; k++)
    {
        d_backProj[x + z*volWidth + k*volWidth+volDepth] += vst[k];
    }
}
}

Listing A4. Code snippet: Asynchronous memory copy handling and kernel calls.

// create streams
cudaStream_t *stream =
    (cudaStream_t *)malloc(STREAM_SIZE * sizeof(cudaStream_t));
for (int i = 0; i < STREAM_SIZE; i++)
{
    cudaStreamCreate(&(stream[i]));
}

int index = 0;
for (int iy = yChunkStart; iy < yChunkEnd; iy = iy + MAP*STREAM_SIZE)
{
    // Allocate and initialize an array of stream handles
    // allocate memory for MAP*backprojection slices on device
    float *d_backProj[STREAM_SIZE];
    for (int i = 0; i < STREAM_SIZE; i++)
    {
        HANDLE_ERROR(cudaMalloc((void**)&d_backProj[i],
            MAP*param.volX*param.volZ*sizeof(float)));
    }
unsigned long long offset[STREAM_SIZE];
for (int istm = 0; istm < STREAM_SIZE; istm++)
{
    // (re-)copy host memory for backprojection to device
    // using MAP slices
    offset[istm] = (unsigned long long)param.volX*(unsigned long long)param.volZ*(unsigned long long)(index + istm*MAP);
    cudaMemcpyAsync(d_backProj[istm],
                    (char *)&h_backproj_chunk[offset[istm]],
                    MAP*param.volX*param.volZ*sizeof(float),
                    cudaMemcpyHostToDevice,
                    stream[istm]);
}

// run Backprojection Kernel
int y_c = iy + istm*MAP; // current z-slice
if (param_.useVolWin)
{
    // process remaining projection blocks
    if (hasRemainder && np == proj_blocks - 1)
    {
        // switch between fast and accurate bi-linear interpolation
        if (param_.highAcc == 0)
        {
            fdk_kernel_3DW_R << <blocks, threads >> >(d_backProj[istm], y_c, cuProjBlockIdx, remainProj, param.volX, param.volZ, param.shiftX, param.shiftY, param.wVolOrigX, param.wVolOrigY, param.wVolOrigZ);
        }
        else
        {
            fdk_kernel_3DW_R_HA << <blocks, threads >> >(d_backProj[istm], y_c, cuProjBlockIdx, remainProj, param.volX, param.volZ, param.shiftX, param.shiftY, param.wVolOrigX, param.wVolOrigY, param.wVolOrigZ);
        }
    }
    else // no remaining projection blocks to process
    {
        // switch between fast and accurate bi-linear interpolation
        if (param_.highAcc == 0)
        {
            fdk_kernel_3DW << <blocks, threads >> >(d_backProj[istm], y_c, cuProjBlockIdx, param.volX, param.volZ, param.shiftX, param.shiftY, param.wVolOrigX, param.wVolOrigY, param.wVolOrigZ);
        }
        else
        {
            fdk_kernel_3DW_HA << <blocks, threads >> >(d_backProj[istm], y_c, cuProjBlockIdx, param.volX, param.volZ, param.shiftX, param.shiftY, param.wVolOrigX, param.wVolOrigY, param.wVolOrigZ);
        }
    }
}
else // no remaining projection blocks to process
d_backProj[istm], y_c, 
cuProjBlockIdx, 
param.volX, param.volZ, 
param.shiftX, param.shiftY, 
param.wVolOrigX, param.wVolOrigY, param.wVolOrigZ
); 
}
}
}

getLastCudaError("Kernel execution failed");
cudaMemcpyAsync(
(char *)&h_backproj_chunk[offset[istm]], 
d_backProj[istm], 
MAP*param.volX*param.volZ*sizeof(float), 
cudaMemcpyDeviceToHost, 
stream[istm]
);

} // streams-end

for (int i = 0; i < STREAM_SIZE; i++)
{
    HANDLE_ERROR(cudaFree(d_backProj[i]));
    index = index + MAP*STREAM_SIZE;
}

} // release all streams
for (int i = 0; i < STREAM_SIZE; i++)
{
    cudaStreamDestroy(stream[i]);
}

References
1. CUDA Toolkit. Available online: https://developer.nvidia.com/cuda-toolkit (accessed on 27 September 2021).
2. TIGRE: Tomographic Iterative GPU-Based Reconstruction Toolbox. Available online: https://github.com/CERN/TIGRE (accessed on 27 September 2021).
3. The ASTRA Toolbox. Available online: https://www.astra-toolbox.com/ (accessed on 27 September 2021).
4. Reconstruction Toolkit (RTK). Available online: https://www.openrtk.org/ (accessed on 27 September 2021).
5. Kaestner, A.P. MuhRec—A New Tomography Reconstructor. Nucl. Instrum. Methods Phys. Res. 2011, 651, 156–160. [CrossRef]
6. Beister, M.; Kolditz, D.; Kalender, W.A. Iterative reconstruction methods in X-ray ct. Phys. Med. 2012, 28, 94–108. [CrossRef] [PubMed]
7. Willemink, M.J.; Noël, P.B. The evolution of image reconstruction for CT-from filtered back projection to artificial intelligence. Eur. Radiol. 2019, 29, 2185–2195. [CrossRef] [PubMed]
8. Xu, F.; Mueller, K. Real-time 3D computed tomographic reconstruction using commodity graphics hardware. Phys. Med. Biol. 2007, 21, 3405–3419. [CrossRef] [PubMed]
9. Scherl, H.; Keck, B.; Kowarschik, M.; Hornegger, J. Fast GPU-Based CT Reconstruction using the Common Unified Device Architecture (CUDA). In Proceedings of the 2007 IEEE Nuclear Science Symposium Conference Record, Honolulu, HI, USA, 27 October–3 November 2007; pp. 4464–4466. [CrossRef]
10. Kak, A.C.; Slaney, M. Principles of Computerized Tomographic Imaging; Society of Industrial and Applied Mathematics: Philadelphia, PA, USA, 2001.
11. Dewulf, W.; Tan, Y.; Kieckens, K. Sense and non-sense of beam hardening correction in CT metrology. CIRP Ann. Manuf. Technol. 2012, 61, 495–498. [CrossRef]
12. Davis, G.; Jain, N.; Elliott, J. A modelling approach to beam hardening correction. In SPIE 7078, Developments in X-ray Tomography VI, 70781E; Society of Photo Optical: Bellingham, WA, USA, 2008. [CrossRef]
13. Kyriakou, Y.; Meyer, E.; Prell, D.; Kachelriess, M. Empirical beam hardening correction (EBHC) for CT. Med. Phys. 2010, 37, 5179–5187. [CrossRef] [PubMed]
14. Kreyszig, E. Advanced Engineering Mathematics, 4th ed.; Wiley: Hoboken, NJ, USA, 1979; p. 880, ISBN 0-471-02140-7.
15. OpenCV: Open Source Computer Vision Library. Available online: https://opencv.org/ (accessed on 27 September 2021).
16. Kim, Y.; Baek, J.; Hwang, D. Ring artifact correction using detector line-ratios in computed tomography. Opt. Express 2014, 22, 13380–13392. [CrossRef] [PubMed]
17. Buzug, T.M. *Computed Tomography: From Photon Statistics to Modern Cone-Beam CT*, 1st ed.; Springer: Berlin, Germany, 2010.

18. CUDA C++ Programming Guide. Available online: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#linear-filtering (accessed on 27 September 2021).

19. Kingston, A.; Sakellariou, A.; Varslot, T.; Myers, G.; Sheppard, A. Reliable automatic alignment of tomographic projection data by passive auto-focus. *Med. Phys.* 2011, 38, 4934–4945. [CrossRef] [PubMed]

20. Eigen: C++ Template Library for Linear Algebra: Matrices, Vectors, Numerical Solvers, and Related Algorithms. Available online: https://eigen.tuxfamily.org/index.php?title=Main_Page (accessed on 27 September 2021).

21. Farber, R. *CUDA Application Design and Development*; Morgan Kaufmann: Burlington, MA, USA, 2011.

22. Cheng, J.; Grossman, M.; McKercher, T. *Professional CUDA C Programming*; John Wiley & Sons: Hoboken, NJ, USA, 2014; p. 289.

23. Van Werkhoven, B. Kernel tuner: A search-optimizing GPU code auto-tuner. *Future Gener. Comput. Syst.* 2019, 90, 347–358. [CrossRef]

24. Grauer-Gray, S.; Xu, L.; Searles, R.; Ayalasomayajula, S.; Cavazos, J. Auto-tuning a high-level language targeted to GPU codes. In Proceedings of the 2012 Innovative Parallel Computing (InPar), San Jose, CA, USA, 13–14 May 2012; pp. 1–10. [CrossRef]

25. CUDA Programming Guide (Thread Hierarchy). Available online: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#thread-hierarchy (accessed on 27 September 2021).