Balancing load of GPU subsystems to accelerate image reconstruction in parallel beam tomography

Suren Chilingaryan
Karlsruhe Institute of Technology
chilingaryan@kit.edu

Evelina Ametova
KU Leuven
evelina.ametova@kuleuven.be

Andreas Kopmann
Karlsruhe Institute of Technology
kopmann@kit.edu

Alessandro Mirone
ESRF
mirone@esrf.fr

***Abstract*—Synchrotron X-ray imaging is a powerful method to investigate internal structures down to the micro- and nanoscopic scale. Fast cameras recording thousands of frames per second allow time-resolved studies with a high temporal resolution. Fast image reconstruction is essential to provide the synchrotron instrumentation with the imaging information required to track and control the process under study. Traditionally Filtered Back Projection algorithm is used for tomographic reconstruction. In this article, we discuss how to implement the algorithm on nowadays GPGPU architectures efficiently. The key is to achieve balanced utilization of available GPU subsystems. We present two highly optimized algorithms to perform back projection on parallel hardware. One is relying on the texture engine to perform reconstruction, while another one utilizes the *Core* computational units of the GPU. Both methods outperform current state-of-the-art techniques found in the standard reconstructions codes significantly. Finally, we propose a hybrid approach combining both algorithms to better balance load between GPU subsystems. It further boosts the performance by about 30% on NVIDIA Pascal micro-architecture.**

I. Introduction

Recent advances in X-ray optics and detector technology have paved the way for a variety of new X-ray imaging experiments aiming to study dynamic processes in materials and to analyze small organisms in vivo [1], [2], [3]. The instrumentation used at imaging beamlines has also recently undergone a major update. The installed streaming cameras are able to deliver thousands of frames per second with a continuous data rate of up to 8 GB/s [4]. Newly developed control systems use the acquired imaging information to track the processes under study and adjust the instrumentation accordingly [4], [5]. In order to achieve higher temporal resolution and prolong experiment duration, advanced and compute intensive methods are developed [6], [7]. These methods are able to produce high quality images from undersampled and underexposed measurements by incorporating existing a priori knowledge and solving the reconstruction as an iterative optimization problem. Consequently, the amount of data generated at imaging beamlines quickly grows and the computational demands are on a steep rise.

To tackle the performance challenge several reconstruction frameworks have been developed and optimized to use parallel capabilities of nowadays computing architectures [8], [9], [10], [11]. For online monitoring and control, normally fast analytical methods are used to reconstruct 3D images. There are two main approaches: Filtered Back Projection (FBP) and methods based on the Fourier Slice Theorem [12]. The later methods are asymptotically faster, but due to the involved interpolation in the Fourier domain are more sensitive to the quality of the available projections. Recent study suggests to implement back projection as convolution in log-polar coordinates in order to gain high reconstruction speed with interpolation in the image domain [13]. However, this new method has not yet been adopted in production environments. Still, Filtered Back Projection is the method of choice, largely due to its simplicity and robustness.

To provide high performance, GPU architectures include multiple components that are operating independently. Texture fetches, memory operations, several types of arithmetic instructions are executed by the different blocks of a GPU in parallel. Hence, the execution time is not determined by the sum of all operations, but rather by the slowest execution pipeline. The strategy to implement an efficient algorithm is to balance operations between the available GPU blocks and to minimize the time required to execute the slowest pipeline. While many imaging frameworks use GPUs to speed-up the reconstruction, the known implementations solely use the texture engine to implement the back-projection. Furthermore, specific features of the GPU architecture are not considered and, consequently, the full performance of the texture engine is not utilized. Only a few papers discuss GPU architectures in details, but are mostly addressing conic beam geometry [14], [15]. To our knowledge, there are no papers available on the NVIDIA Pascal and newer architectures yet.

In this article, we present an optimized implementation of the FBP algorithm. The focus is on the back projection step of the algorithm which dominates the reconstruction time. We discuss two approaches to adapt back projection to GPU architecture and show how both algorithms can be executed in parallel to better balance load between available hardware units. While multiple tasks are often scheduled to the same GPU to achieve better hardware utilization, it is a new technique to execute two algorithms optimized to utilize a different set of hardware units in order to solve a single problem. Up to our knowledge, it was not studied in the literature yet. The article is organized as follows. Parallel beam tomography and the data-flow are discussed in section II. Two modifications of the back-projection algorithm are developed in section III. The performance achieved on NVIDIA Pascal micro-architecture is discussed in section IV.

II. Parallel Beam Tomography

At synchrotron imaging beamlines, samples are typically placed at a rotating stage in front of a pixel detector. It registers a series of 2D projections while the sample is turning. Information about X-ray attenuation or/and phase changes in the sample are used to reconstruct its internal structure. Due to the rather large source-to-sample distance, imaging at synchrotron light sources is well described by a parallel-beam geometry. The beam direction is perpendicular to the rotation axis and to the lines of the pixel detector. Therefore, the 3D reconstruction problem can be split into a stack of 2D reconstructions performed with cross-sectional slices. The coordinate system is defined so that the center of rotation is located at the center of the sample coordinate system and the sample is rotating about the vertical axis. To reconstruct a slice, the projection values are smeared back over the 2D cross section and are integrated over all projection angles, see Fig. 1. To compensate blurring effects inherent to back-projection, high-pass filtering of the projection data is performed prior to back projection [12].

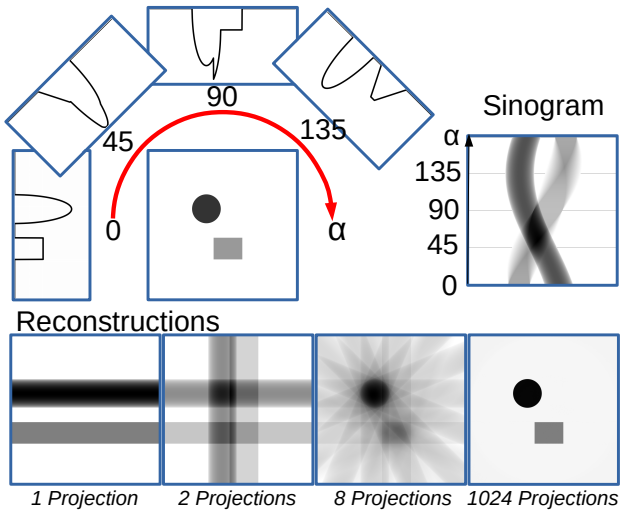

Fig. 1: Reconstruction of 2D slice. A sample is rotated in front of a pixel detector which measures the attenuation of X-rays in the sample under different rotation angles (top-left). The recorded projections are arranged in sinogram (top-right). For image reconstruction, all projections are smeared back onto the cross section along the direction of incidence yielding an accumulated image (bottom).

To perform reconstruction, first the projections are rearranged to group together chunks of data required to reconstruct each slice. A single row of pixels is extracted from each projection and all such rows are stacked together in a 2D image, i.e. y-coordinates correspond to projections and x-coordinates to detector bins (pixels) in a row. These images are called sinograms and are reconstructed independently. To enable multi-GPU support, sinograms are distributed between the parallel accelerators available in the system in a round-robin fashion. The FBP algorithm is executed in 4 steps. First the sinogram is transferred to GPU memory. Next, it is convolved with the configured high-pass filter and the result is stored in 2D texture. Then, the back projection is performed and finally the reconstructed slice is transferred back to system memory.

To allow overlapping of memory transfers with kernel executions, the page-locked (pinned) system memory is used to store the data and all steps of FBP algorithm are pipelined. The sinograms are transferred to GPU memory in groups of 4. While one group is transferred to GPU, the sinograms already residing in the GPU memory are reconstructed, and the results of a third group are transferred back to the system memory. A double-buffering technique and a stream-based asynchronous API are used to allow parallel execution. Two pairs of buffers are used to store input and output data in the GPU memory. The workload is executed in 3 CUDA streams. One stream is used to download data, one for processing, and one for upload. In each iteration, the *download* stream is synchronized to ensure that data is ready for filtering. Then, the input buffers are switched and the new transfer is scheduled asynchronously. In parallel filtering and back-projection are started in another CUDA stream. Upon completion, also the *upload* stream is synchronized to ensure that previous transfers have been finished. Then, the data upload is scheduled asynchronously, the output buffers are switched, and the next iteration is started.

The convolution with a high-pass filter is performed as multiplication in Fourier domain using standard FFT libraries. For optimal performance, the sinogram rows are padded to the next power of 2 and all rows are converted to and from Fourier domain together using a single batched transformation. Most available FFT libraries do not support *real-to-complex* and *complex-to-real* transforms. While NVIDIA *cuFFT* does, the performance improvement over the *complex-to-complex* transform is minimal. Therefore, we convert each pair of sinograms into a single complex vector and execute a single complex convolution instead of two convolutions with real numbers [16]. The first sinogram is stored in the real components of the complex vector and the imaginary components contain the second sinogram. Two *complex-to-complex* transformations are executed to perform convolution and the resulting sinograms are similarly interleaved in the memory. The approach results in a 65-75% performance increase compared to the standard convolution with real numbers.

III. Back-projection

To determine a function of the sample object at a given position with coordinates (x, y), $\sum_p I(x \cdot \cos(\alpha_p) - y \cdot \sin(\alpha_p), p)$ is computed over all projections, where I is a sinogram and α_p is rotational angle of p-th projection. The widely adopted parallel implementation is rather straightforward. Each GPU thread is responsible for a single pixel of the output slice and iterates over all projections to sum up the contribution of each one. At each iteration, a projection is performed to find the coordinate where the ray passing through the pixel hits the detector. The value at the corresponding position in the sinogram is fetched using the texture engine and summed

up with the contributions from other projections. The texture engine is configured to perform either nearest-neighbor or linear interpolation as desired. Since trigonometric functions are relatively slow in GPU architectures, sine and cosine of all projection angles are pre-computed at the CPU and are stored in the GPU constant memory during the initialization phase. The performance of this approach is dominated by the throughput of the texture engine, but keeps all other components of the GPU architecture rather under-utilized. Furthermore, it even does not make use of the full potential of the texture engine available on the newer GPUs. In this section we first discuss how the standard implementation can be optimized to use throughput and cache of the texture engine more efficiently. In addition we will introduce an alternative algorithm which caches data in shared memory and performs the interpolation using GPU *Core* units. Finally, we will discuss a hybrid method that combines both approaches and is able to balance the computational load between all available GPU resources.

A. Notation

We use mixture of a mathematical and a *C*-style notation to describe the algorithms. All variables used across the algorithms are listed in TABLE I and use the following naming scheme to simplify reading. We group related variables together. The same letter is used to refer all variables of the group and the actual variable is specified using subscript. The superscript indicates a memory domain where the variable is located: $\cdot^S$ refers variables in shared memory, $\cdot^C$ - constant memory, and $\cdot^G$ - global memory. The local variable is referenced if superscript is not used. For instance, c_s^S points to the sine of the projection angle stored in the shared memory. We use $\vec{\cdot}$ symbol to denote all vector variables. The assignment between vector variable and scalars are shown using curly braces. Furthermore, all proposed algorithms are capable to reconstruct 1, 2, or 4 slices in parallel. $\tilde{\cdot}$ is used to indicate variables which have a variable size depending on the number of reconstructed slices. All arithmetic operations in this case are performed in vector form and affect all slices. The vector multiplication is performed element wise. We also use the *__shfl_xor* operation to perform reduction on a vector data. In fact, the vector types are not supported. The operation is implemented as several calls to the corresponding function using all vector components one after another. We use integer division and modulo operations across the code listings. These operations are very slow on GPUs, but the optimizing compiler will replace them by the faster bit-mangling instructions automatically. Therefore, we use the notation which is easier to read. There are a few other cases where the optimization is left to the compiler.

B. Multi-slice reconstruction

The reconstruction performance in the standard approach is bound to the filtering rate of the texture engine. Up to a certain limit, the filter rate is independent from the actually used data type. The same number of texels is returned per second if either 8, 16, or 32-bit format is used to encode the texel values. All modern GPUs are capable to filter 64-bit data at the full speed including 64-bit vector types, like *float2*. The tomographic reconstruction is typically performed using 32-bit single-precision numbers. In parallel tomography, however, exactly the same operations are performed for all the reconstructed slices. Therefore, it is possible to reconstruct multiple slices in parallel if the back projection operator is applied to a compound sinogram which encodes bins from multiple simple sinograms as a vector value. Particularly, it is possible to construct such sinogram using *float2* vector type and interleave values from one sinogram as x components and from another as y. The bandwidth of the texture engine is fully utilized and two slices are reconstructed in parallel if the appropriate *float2*-typed texture is mapped on this sinogram. The interleaving is done as an additional data preparation step before the back projection kernel is started. The kernel is adjusted to use the *float2* type and writes the x component of the result into the first output slice and the y component into the second.

The proposed approach can be further scaled to 4 slices if 16-bit half-precision floating-point data is used. While the reduced precision might affect the quality of reconstruction, the majority of cameras has only a dynamic range of 16 bits or bellow. High-speed cameras actually used for time-resolved synchrotron tomography have even a lower resolution of 10-12 bits only. Therefore, using a half-precision representation to store the input data should have a limited impact on the resulting image quality if all further arithmetic operations are performed in single-precision. The half-precision textures are not supported in the latest available version of CUDA yet (*CUDA 8.0*). While one can store the half-precision numbers in the GPU memory, it is impossible to map the corresponding

TABLE I: List of variables used in code snippets

Var	Type	Description
n_p	int	Number of projections
$\vec{n}_t$	int2	Dimensions of thread block
n_q	int	Number of pixels assigned per GPU thread
n_s	int	Size of an area assigned to a thread block
s_d	int	Size of data cache in projections
s_t	int	Caching threads per projection row
$\vec{v}_a$	float2	The position of rotation axis
c_c	float[]	Cosine values of the projection angles
c_s	float[]	Sine values of the projection angles
c_a	float[]	Coordinates of the rotational axis
c_m	float[]	Coefficients to find bins required by a block
m_p	int	Projection index in a group
m_d	int	Mapping to select an offset in the cache
$\vec{m}_b$	int2	Index of a thread block within the grid
$\vec{m}_t$	int2	Index of a thread within the block
$\vec{m}_g$	int2	Index of a thread within the grid
$\vec{m}'_*$	int2	Re-mapped index
$\vec{f}_*^*$	float2	Pixel coordinate according to the mapping
d	float[][]	The cache storing a subset of sinogram
$\tilde{s}$	float[]	Accumulator of a resulting pixel value
$\tilde{r}$	float[][]	The reconstructed slice
p_*	int	Projection number; iterators (p_b, p_i)
q_*	int	Pixel block iterators
h_*	various	Positions in cache/sinogram

texture. Still, it is possible to speed-up the reconstruction if the nearest-neighbor interpolation mode is utilized. The texture-mapping is created using the *float2* data type. Upon request the texture engine returns the nearest value without performing any operations on it. Therefore, the appropriate data is returned even if an incompatible format is configured. It is important that the data size is correct. To avoid further penalty to the precision, the half-precision numbers are immediately cast to single-precision and all further operations are performed in single-precision as usual. Using the standard Shepp Logan phantom [17], the penalty on quality due to the described optimization is negligible. The difference in gray values between reconstructions is bellow 1% for all pixels. The achieved speed-up depends on a performance of half- to single-precision type conversions. While significant speed-up is measured on NVIDIA Pascal, no performance improvements are reported for NVIDIA Kepler architecture and for all AMD GPUs lacking support of half-float OpenCL extension.

C. Texture-based back-projection kernel

Only 70% of the theoretical throughput of the texture engine is actually reached on NVIDIA Pascal if 4 slices are reconstructed in parallel. The performance is limited by the bad locality of the texture fetches and an excessive load of the Special Function Units (SFU) units. The CUDA C Programming Guide states that the SFUs are used to compute approximates of transcendent functions [18]. In fact, they are also used to perform bit-mangling, type-conversion, and integer multiplication on Pascal GPUs. In this case, the SFUs are busy resolving indexes of the array with the geometrical constants and converting data between half-precision and single-precision representations.

While it is not possible to reduce the number of type casts, the constants can be re-used multiple times if each GPU thread reconstructs several pixels. As different pixels are reconstructed completely independently, the thread processing multiple pixels would also benefit from a flow of independent operations allowing the GPU scheduler to dual-issue instructions. There are two approaches to do it. The first option is to reduce the size of the computational grid and to assign to each thread the corresponding amount of output pixels. Alternatively, the number of threads is kept unchanged, but several projections are processed in parallel. Then, each thread contributes to multiple resulting pixels, but iterates over only a subset of all projections while another thread contributes to the same group of pixels, but from a different set of projections. Both methods perform similarly if properly optimized. However, the second approach allows to keep the dimensions of a computational grid unchanged and only re-defines how the work is performed by the threads of the grid. Therefore, it has an advantage for reconstructing small images or if only a region-of-interest (ROI) is required.

Further it is necessary to ensure a good locality of the texture fetches. The locality of fetches within a block, a warp, and also within a group of 4 consecutive threads is important to keep the texture engine running at full speed. Fig. 2 illustrates the proposed mapping. Blocks of 256 threads are responsible for an output area of 16x16 pixels and this area is further subdivided in 4x4 pixels squares. 64 threads are assigned to process each square and 4 such squares are reconstructed in parallel. A full set of 16 squares requires 4 iterations to complete. At each iteration, the squares on the same line are reconstructed. Compared to the other possible arrangements, this mapping results in a slightly better cache hit rate and reduces the register usage as only a single pixel coordinate is incremented for each thread. Each pixel is reconstructed using 4 threads. Each thread is responsible to compute the contribution to the pixel value from a quarter of all available projections. To avoid costly atomic operations, the contributions of the projection subsets are summed completely independently. Then, the threads are re-assigned to perform reduction in the shared memory and to compute the resulting pixel value. To avoid serialization of the warps due to unaligned constant memory requests, all threads of a warp are always used to process the same projection. Consequently, the lowest 4 bits of the thread number in a block define the mapping within pixel square, next 2 bits define a square, and the top 2 bits define the processed projection. To ensure good fetch locality, the threads within the square are mapped along Z-order curve.

The pseudo-code for the proposed approach is presented in Algorithm 1. Two distinct processing stages are executed. First the partial sums are computed in an 4-element array. The outer loop starts from the first projection assigned to the current thread and steps over the projections which are processed in parallel. At each iteration the constants are loaded and inner loop is executed to process 4 pixels. After completion of all projections, the reduction loop is started. The partial sums are written into the shared memory and reduction is performed using the *shuffle* operation. On architectures without *shuffle* instruction, a standard reduction in shared memory can be executed instead.

D. ALU-based back-projection kernel

The previously described approach is based on the texture engine to perform the interpolation, but the *Core* floating-point units can be used instead. The loads from global memory limit performance severely. The L1 cache is small and suffers from cache poisoning. Therefore, the shared memory is used instead as explicit cache. Due to the caching overhead the performance improves if a large number of pixel is reconstructed in each block. The actual size is determined by the amount of registers provided by the hardware. Either 32x32 or 64x64 pixel per thread block are suitable for Pascal GPUs.

A subset of a single sinogram row is required to reconstruct a square region of an output slice with side N. The smallest bin (h_m) is always accessed while reconstructing one of the corners. The actual corner is only depending on the projection angle and is the same across all squares in a slice. Consequently, the h_m can be computed as $floor(h_b + c_m)$, where h_b is the bin accessed by the first thread of a block and $c_m = N \cdot \max(0, \cos(\alpha), -\sin(\alpha), \cos(\alpha) - \sin(\alpha))$.

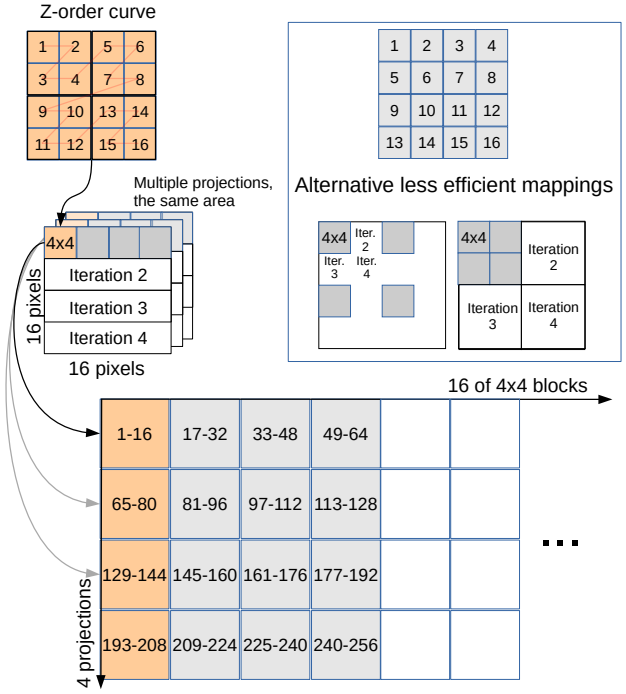

Fig. 2: GPU thread to pixel/projection mapping. A block with 256 threads is assigned to each square of 16x16 pixels. The threads within each small 4x4 square are mapped along Z-order curve (top). A group of 64 consecutive threads is responsible to process a rectangular area of 16x4 pixels (middle). Multiple consecutive projections are processed in parallel using 4 such groups (bottom). The complete square is reconstructed over 4 iterations. Alternative possible mappings are less efficient due to worse cache utilization or higher register usage (right). For each output pixel or block of pixels, the assigned range of threads is stated in the corresponding cells. The area processed by the threads of a block in parallel are shown in gray/orange.

Input: Texture, projection constants (c_*^C), dimensions (n_*), cache sizes (s_*), and other parameters (v_*)
Shared: $\tilde{s}^S[64][4]$, $\tilde{r}^S[16][16]$
Output: Reconstructed slice $\tilde{r}^G$
begin
  /* Computing sequential numbers of 4x4 square, quadrant, and pixel within quadrant */
  $square = m_t.y \,\%\, 4$
  $quadrant = m_t.x \,/\, 4$
  $pixel = m_t.x \,\%\, 4$
  /* Computing projection and pixel offsets */
  $m_p = m_t.y \,/\, 4$
  $m'_t.x = 4 * square + 2 * (quadrant \,\%\, 2) + (pixel \,\%\, 2)$
  $m'_t.y = 2 * (quadrant \,/\, 2) + (pixel \,/\, 2)$
  /* Computing pixel coordinates */
  $\vec{m}'_g = \vec{m}_b * \vec{n}_t + \vec{m}'_t$
  $\vec{f}'_g = \vec{m}'_g - \vec{v}_a$
  /* Computing partial sums */
  $\tilde{s}[4] = \{0\}$
  for $(p = m_p;\ p < n_p;\ p \mathrel{+}= 4)$
    $c_s = c_s^C[p].y$
    $h = c_a^C[p] + f'_g.x * c_c^C[p] - f'_g.y * c_s^C[p] + 0.5f$
    for $(q = 0;\ q < 4;\ q \mathrel{+}= 1)$
      $\tilde{s}[q] \mathrel{+}=$ **tex2d**$(h - 4 * q * c_s,\ p + 0.5f)$
    end
  end
  /* Reduction */
  $\vec{m}''_t = \{m_t.x \,\%\, 4, 4 * m_t.y + m_t.x \,/\, 4\}$
  for $(q = 0;\ q < 4;\ q \mathrel{+}= 1)$
    /* Moving partial sums to shared memory */
    $\tilde{s}^S[n_t.x * m'_t.y + m'_t.x][m_p] = \tilde{s}[q]$
    syncthreads
    /* Performing reduction */
    $\tilde{r} = \tilde{s}^S[m'_t.y][m'_t.x]$
    for $(i = 2;\ i \geq 1;\ i \mathrel{/}= 2)$
      $\tilde{r} \mathrel{+}=$ **shfl_xor**$(\tilde{r},\ i,\ 4)$
    end
    /* Grouping results in shared memory to coalesce global memory writes */
    if $m''_t.x == 0$ **then**
      $\tilde{r}^S[4 * q + m''_t.y \,/\, 16][m''_t.y \,\%\, 16] = \tilde{r}$
    end
    syncthreads
  end
  $\tilde{r}^G[m_g.y][m_g.x] = \tilde{r}^S[m_t.y][m_t.x]$
end

Algorithm 1: Optimized implementation of the back-projection kernel relaying on the texture engine to perform interpolation

This value is pre-computed during initialization stage and is stored in the constant memory along with other per-projection constants. The number of required bins is equal to $(x_0 - x_1) \cdot \cos(\alpha) - (y_0 - y_1) \cdot \sin(\alpha)$, where $(x_0,\ y_0)$ and $(x_1,\ y_1)$ are some coordinates in the region. For some angle β, it is equivalent to: $\sqrt{(x_0 - x_1)^2 + (y_0 - y_1)^2} \cdot \cos(\alpha + \beta)$ and does not exceed $N \cdot \sqrt{2}$. One extra bin is required to perform linear interpolations. For sake of simplicity, we always cache $\frac{3}{2}N$ bins starting at offset h_m. Furthermore, the value of h_m is required by both steps of the algorithm. First it is needed to perform caching and, then, to locate the required value in the cache. The rounding operation is performed to find h_m. On NVIDIA architectures it adds significantly to the computational balance of SFU units which are also heavily loaded by rounding and type-mangling operations required to resolve cache indexes. To reduce the load, the h_m is cached in shared memory during the first stage of algorithm and re-used in the second. The complete scheme is illustrated in Fig. 3.

Since no interpolation is required while the data is read from the global memory, it is possible to access the sinograms directly rather than using texture fetches. However, NVIDIA relays on the same LD units to perform the shared and global memory operations. I.e. either a shared memory or a global memory instruction can be executed at each clock. On other hand, texture fetches and shared memory loads are performed in parallel. The corresponding pseudo-code is presented in Algorithm 2. A block of 256 threads is used and each thread is responsible to reconstruct 4 to 16 pixels. The reconstruction is performed in two stages. At first, the bins required by a thread block are determined and cached for a set of projections. Afterwards, the reconstruction is performed using the data in the cache. To perform caching, the threads of a block are split into several groups. Each group is responsible for caching bins for a single projection. The number of threads in the group is selected to avoid bank conflicts in shared memory and a sufficient number of projections is cached to utilize

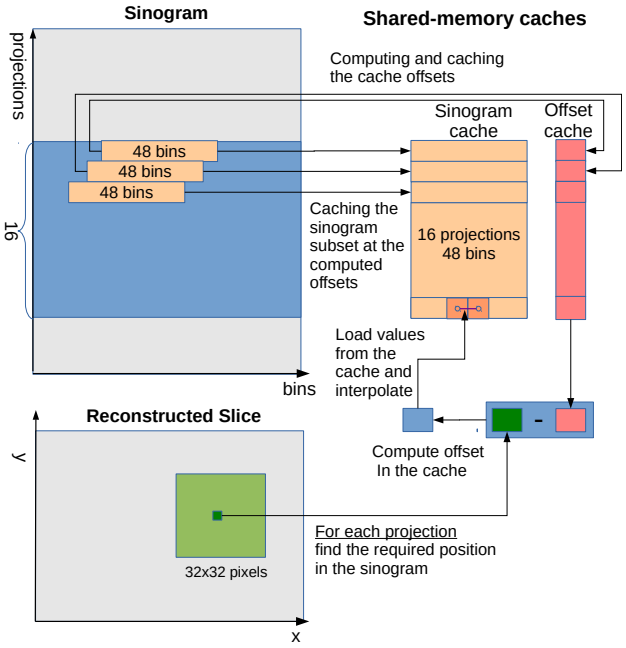

Fig. 3: Reconstruction using GPU *Core* units. The projections are processed in groups of 16 iteratively. For each square of 32x32 pixels, a thread block extracts 48 bins from a set of projections (top) and stores them in the shared memory along with the corresponding offsets. Then, the thread mapping is changed and projections are processed in a loop one after another. Each thread is responsible for several pixels of output slice. For each pixel, the corresponding position in a sinogram is computed and the required offset in the cache is determined by subtracting the cache offset (bottom). Then, one or two values are loaded from the cache and the configured interpolation is performed.

all available threads of a block. The group consisting of $\frac{N}{2}$ threads causes no bank conflicts on the Pascal architecture. Consequently, either 16 or 8 projections are processed per single step of the algorithm. It takes 3 iterations to cache all required bins. The threads of a block are then re-assigned to match the output pixels and process the contributions from the cached projections in a loop. The threads determine a position where the ray passing through the reconstructed pixel hits the detector row. The corresponding bin in a sinogram is computed and the offset in the cache is found. Typically the offset is not an integer and falls in between of two cached values. Depending on the configured interpolation mode either the offset is rounded to the nearest integer and a single value is loaded from the shared memory or both neighboring values are loaded and a linear interpolation is performed to compute the impact of the projection.

According to the documentation there is no difference in which order the threads of a half warp are accessing the shared memory. However, in practice we found that the performance of 64- and 128-bit shared memory loads is slightly improved if only 1-2 different addresses are accessed by groups of 4 consecutive threads. Therefore, the half-warps are mapped to the pixel squares of 4x4 pixels and arranged along a Z-order curve similarly to the texture fetches. The 256 threads of the block are mapped to 16 such squares and the squares are arranged linearly along x-axis. Two rows of 4x4 squares are processed in parallel if a small 32x32 area is reconstructed. A single row is covered for the bigger area. The remaining rows are processed over 4-16 iterations. The threads accumulate the sums for each pixel in a register-bound array and dump it to the global memory once processing of all projections is completed.

Input: Texture, projection constants (c^C_*), dimensions (n_*), cache sizes (s_*), and other parameters (v_*)
Assume: $n_s = 32$; $n_q = 4$, $s_t = 16$, $s_d = 16$
Shared: $\tilde{d}^S[s_d][\frac{3}{2} * n_s]$, $\tilde{h}^S_m[s_d]$
Output: Reconstructed slice $\tilde{r}^G$
begin
  /* Caching mapping for $s_t = 16$ and $s_d = 16$ */
  $\{m_d, m_p\} = \vec{m_t}$
  $\vec{f_b} = \vec{m}_b - \vec{v}_a$
  /* Reconstruction mapping for $n_s = 32$ */
  $quadrant = m_t.x \;/\; 4$
  $pixel = m_t.x \,\%\, 4$
  $m'_t.x = 4 * (m_t.y \,\%\, 8) + 2 * (quadrant \,\%\, 2) + (pixel \,\%\, 2)$
  $m'_t.y = 4 * (m_t.y \;/\; 8) + 2 * (quadrant \;/\; 2) + (pixel \;/\; 2)$
  $\vec{m}'_g = n_s * \vec{m}_b + \vec{m}'_t$
  $\vec{f}'_g = \vec{m}'_g - \vec{v}_a$
  /* Set accumulators to 0 and run projection loop */
  $\tilde{s}[n_q] = \{0\}$
  for *($p_b = 0$; $p_b < n_p$; $p_b \mathrel{+}= s_d$)*
    /* Compute the minimal required bin */
    $h_b = c^C_a[p_b + m_p] + f_b.x * c^C_c[p_b + m_p] - f_b.y * c^C_s[p_b + m_p]$
    $h_m = \textbf{floor}(h_b + c^C_m[p_b + m_p])$
    /* Cache it in the shared memory */
    if $m_d == 0$ **then**
      $\tilde{h}^S_m[m_p] = c^C_a[p_b + m_p] - h_m$
    end
    /* Cache the data in the shared memory */
    for *($i = 0$; $i < 3$; $i \mathrel{+}= 1$)*
      $h = i * s_t + m_d$
      $\tilde{d}^S[m_p][h] = \textbf{tex2d}(h_m + h + 0.5f,\; p + m_p + 0.5f)$
    end
    syncthreads
    for *($p_i = 0$; $p_i < s_d$; $p_i \mathrel{+}= 1$)*
      $p = p_b + p_i$
      $c_s = c^C_s[p]$
      $h = \tilde{h}^S_m[p_i] + f'_g.x * c^C_c[p] - f'_g.y * c^C_s[p]$
      for *($q = 0$; $q < n_q$; $q \mathrel{+}= 1$)*
        /* Compute the offset in cache */
        $h_i = \textbf{floor}(h)$
        $h_l = h - h_i$
        /* Iterpolate */
        $\tilde{d}_1 = \tilde{d}^S[p_i][h_i]$
        $\tilde{d}_2 = \tilde{d}^S[p_i][h_i + 1] - \tilde{d}_1$
        $\tilde{s}[q] \mathrel{+}= \tilde{d}_1 + h_l * \tilde{d}_2$
        /* Move to the next position */
        $h \mathrel{-}= (n_s \;/\; n_q) * c_s$
      end
    end
    syncthreads
  end
  /* Save the results to global memory */
  for *($q = 0$; $q < n_q$; $q \mathrel{+}= 1$)*
    $\tilde{r}^G[m'_g.y + 8 * q][m'_g.x] = \tilde{r}[q]$
  end
end

Algorithm 2: ALU-based implementation of the back-projection kernel

E. Hybrid Approach

We have proposed two algorithms to perform back-projection. One relies on the texture engine and is bound to its performance. The second is using shared memory and GPU *Core* units, with only a small load on the texture engine. Therefore, it is possible to run the texture-based kernel for one part of the blocks and ALU-based kernel for another part. NVIDIA allows to detect the *Streaming Multiprocessor (SM)* executing the block. Consequently, it is possible to ensure that the desired ratio between threads running texture- and ALU-based kernels is achieved. An array is statically defined in the global GPU memory space. The first thread of a block is resolving the SM number using $get\_smid()$ instruction and increments the corresponding cell of the array using an atomic operation. The block number within a cell is obtained and depending on the requested ratio one of the two algorithms is executed. The code snippet is shown bellow.

```
__device__ uint smblocks[128] = {0};
__global__ static void reconstruct_hybrid() {
    __shared__ uint block;
    if ((threadIdx.x == 0)&&(threadIdx.y == 0)) {
        uint smid = get_smid();
        block = atomicAdd(&smblocks[smid], 1);
    }
    __syncthreads();
    if (block&1) reconstruct_tex(...);
    else reconstruct_alu(...);
}
```

In section III-C we proposed an advanced thread mapping scheme for the texture-based kernel. The goal is to keep the pixel to block assignment minimal in order to preserve the performance for the small images. The ALU kernel, however, aims for larger image sizes and works with 32-by-32 pixel area at minimum. Therefore, an alternative simpler mapping is utilized for the texture-based kernel if it is executed as part of the hybrid approach. Each thread is responsible for 4 to 16 pixels and processes them in a loop. The same texture is used to perform linear interpolation in blocks running texture-based algorithm and to cache data in the blocks executing ALU-based reconstruction.

The preferred algorithm depends on the relative performance of different GPU subsystems. The ALU-based algorithm is the fastest on NVIDIA Fermi architecture because of relatively fast shared memory. Vice versa the texture engine of the Kepler architecture got a significantly higher performance boost compared to other components [18]. Consequently, the texture-based version performs better on the Kepler GPUs. On Maxwell and Pascal architectures it is possible to efficiently balance performance using the described hybrid approach. The highest measured performance is achieved if 2-slices are reconstructed in parallel and 32x32 pixels are assigned to each thread block. The blocks are uniformly split between ALU- and texture-based kernels in this case. Extra 20% of speed is gained if the 100% occupancy is targeted using *__launch_bounds__* annotation. If the reconstruction is limited to a single-slice only, a larger square of 64x64 pixels is assigned to each block and only 3 blocks out of each 8 are executing the texture-based reconstruction. The approach is only useful in cases when the linear interpolation is performed. In the nearest-neighbor mode, the ALU-based kernel outperforms the texture-based variant significantly unless 4-slice reconstruction is performed using the half-float data. Consequently, there is a little effect if they are executed in parallel. In the last case, the SFU performance becomes the bottleneck as both Texture- and ALU-based kernels compete for SFUs to perform rounding and type-mangling operations.

IV. Performance Evaluation

We are not aiming to precisely characterize the performance, but rather try to validate the efficiency of the proposed optimizations. In most tests, we use a data set consisting of 2048 projections with dimensions of 2048 by 2048 pixels each. 512 slices with the same dimensions are reconstructed and the median reconstruction time is used as estimate for the performance. The performance is shown in Giga-updates per second (GU/s) indicating the rate at which the contribution from projections are computed and used to update voxels values. The complete reconstruction time can be estimated by dividing the number of required updates, i.e. total number of voxels multiplied by a number of projections, by the given number in GU/s. Before the tests a heat-up procedure is executed to avoid significant performance discrepancies due to the GPUBoost technology employed by NVIDIA to adjust the GPU clock based on the current load and chip temperature. However, we do not wait until the performance completely stabilizes, but rather avoid a steep performance spike in the beginning, see Fig. 5. The actual hardware clock is compared before and after measurements and the experiment is re-run if the difference is significant. The I/O is completely excluded. The reconstructions are executed using dummy data. The results are dropped without transferring them back to the system memory if the back-projection only benchmark is executed. All tests were executed on NVIDIA GeForce GTX Titan X GPU (Pascal-based) installed in a Supermicro 7047GT server. The server was also equipped with dual Intel Xeon E5-2640 and running OpenSuSE 13.1 along with NVIDIA CUDA SDK 8.0.61 and GPU driver 375.39.

TABLE II shows the effect of optimized thread mapping for the texture-based back projection kernel. According to the profiling results, the number of queries to the texture cache is significantly reduced in the optimized version. While there is no difference in single-slice reconstruction mode, a speed-up of 25% and 35% is measured if 2- or 4-slices are reconstructed in parallel. In the last case, the SFU units limit the performance due to a large number of type-mangling operations used to convert the data between half- and single-precision formats. While we can't eliminate this load, the proposed optimizations reduce the usage of SFU due to other operations 3-fold, and about 90% of the theoretical throughput of the texture engine is achieved in all modes. Unlike the desktop card used in this benchmark, the professional Pascal-based Tesla GPUs have a higher throughput of the half-precision arithmetic instructions. Thus, the back-projection can be performed in half-precision arithmetic completely reducing the load on SFU. The quality

TABLE II: Efficiency of texture fetches using the standard and the optimized texture-based algorithms

Setup		Cache Queries & Hits			Utilization		
#	Alg.	Q[a]	L1[b]	L2[c]	Tex[d]	SFU[e]	Perf.[f]
1	Std.	0.43	95.7%	89.0%	100%	30%	382
	Opt.	0.39	93.2%	89.1%	100%	10%	389
2	Std.	0.61	91.5%	91.8%	100%	30%	534
	Opt.	0.53	90.6%	92.4%	100%	10%	739
4	Std.	0.49	86.2%	83.1%	80%	90%	1128
	Opt.	0.41	90.5%	87.0%	90%	100%	1422

The measurements are obtained with the NVIDIA profiler for the 1-, 2- and 4-slice reconstruction modes. The nearest-neighbor interpolation is performed in 4-slice mode and linear interpolation is used otherwise. The table lists: [a] number of 32-byte queries issued to texture cache per fetch, [b] hit rate of the texture cache, [c] L2 cache hit rate, [d] utilization of texture units, [e] utilization of SFU units, and [f] the achieved reconstruction performance.

of reconstruction, however, might further be penalized in this case.

The performance of all developed algorithms using linear-interpolation is summarized in TABLE III. Although the ALU-based algorithm does not reach the performance of the texture-based kernel, the hybrid approach outperforms the texture-based version by 35%. But neither the texture engine nor the GPU *Core* units are saturated in hybrid mode. Using the Pascal architecture, the performance of both texture- and ALU-based kernels is limited by the available memory throughput. Using the hybrid approach, the data is loaded from the texture engine and the shared memory simultaneously. A combined utilization of 140% is achieved for the two slice reconstruction mode. This matches the achieved speed-up well. The higher utilization of the combined bandwidth is prevented by the high latencies associated with memory load operations. The hybrid kernel is executed at full occupancy. However, as only half of the threads are accessing each type of memory it is efficiently equivalent to running at 50% occupancy. Consequently, there is not enough parallelism for the GPU scheduler to hide the latency completely.

TABLE III: Performance and utilization of the GPU subsystems by the proposed back-projection algorithms

Setup		Utilization				
Slices	Alg.	Texture	Shared	Core	SFU	Perf.
1	Texture	100%	10%	20%	10%	389
	ALU	10%	80%	60%	90%	606
	Hybrid	90%	70%	60%	70%	735
2	Texture	100%	20%	40%	10%	739
	ALU	10%	90%	60%	50%	693
	Hybrid	70%	70%	70%	40%	995

Fig. 4 evaluates the performance depending on the size of reconstructed image. The size is increased from 128 up to 4096 pixels in steps of 128 pixels. The size defines the dimensions of the reconstructed cubic volume, the number of projections used to reconstruct the volume, and the dimensions of each projection. The texture-based approach is faster for small images. Already for 512 pixels the hybrid approach outperforms both other methods. The performance slightly drops due to reduced cache efficiency for large images with the side lengths above 2048 pixels. The right side of the figure shows the effect of pipelining data transfers and reconstruction. The transfer time is significant compared to the reconstruction time for moderate image sizes. Fortunately this time is mostly hidden if NVIDIA scheduler is able to overlap data transfer with kernel execution. The effect of NVIDIA GPUBoost technology is evaluated on Fig. 5. The reconstruction is started with very low clock, but quite fast accelerated. After initial boost of 20%, the performance is reduced and fluctuates around 3-5% above the nominal. It stabilizes only after about 40,000 frames.

The proposed back-projection algorithms outperform the state-of-the-art method substantially. For images with dimensions above 1024 pixels, the standard algorithm has throughput in the range of 350 - 400 GU/s on NVIDIA GeForce Titan X GPU, see Fig. 4. The hybrid reconstruction method performs 2.6 times better and achieves about 970 - 1070 GU/s. The throughput is even faster and reaches 1420 GU/s if the nearest-neighbor interpolation is performed, see TABLE II. This gives a boost of 3.5 times over the performance of the standard implementation. Due to pipelining and optimization of filtering, the impact of back-projection dominates the performance also for moderate image sizes. The new algorithms allow to reduce the complete reconstruction time 2 - 3 times depending on the image size and selected interpolation method. For instance, only 21 seconds are required to reconstruct a cubic volume of 2048^3 voxels from 2048 projections.

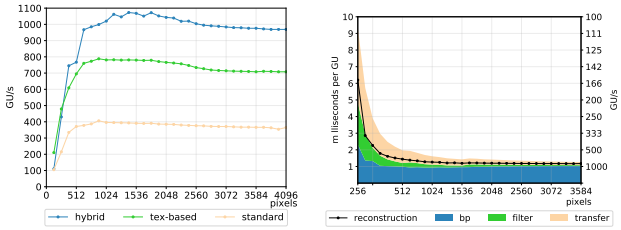

Fig. 4: The performance of back-projection kernels (left) and of complete FBP algorithm using the hybrid kernel (right). In the right, the stacked chart sums times required for each step of the algorithm independently and the black line indicates the actual reconstruction time achieved by overlapping of computation and data transfer. To allow stacking of the times, the main axis is expressed in milliseconds per GU and the corresponding number of GU/s is shown on the secondary axis.

V. Conclusion

In this paper, we have demonstrated that a significant speed-up is possible if low-level details of GPU architecture are taken into the consideration. A higher utilization of the texture engine is achieved if the data can be re-arranged in larger vector types. Such vectors are streamed by the texture engine at the same rate as simple floating-point numbers provided that the high locality of the texture fetches can be ensured across half-wraps and also within groups of 4-consecutive threads. Even if half-precision floating point numbers are not directly

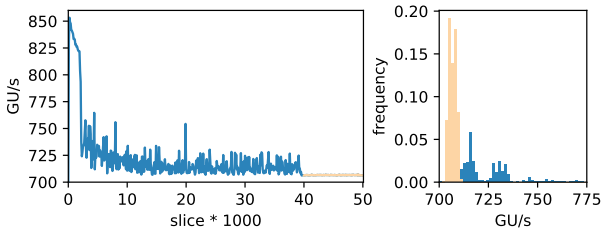

Fig. 5: The influence of GPUboost on the performance of reconstruction using hybrid algorithm. The performance is averaged over groups of 100 slices and is shown versus consecutive slice number (left). The distribution of reconstruction speed is given for each slice (right). The distribution after the performance stabilization is shown in orange.

supported by the texture engine, we shown that they still can be efficiently utilized by binding a texture with the forged data type. We further explained how to improve hardware utilization by launching several concurring algorithms which solve the same problem but target different GPU subsystems. The optimal ratio between GPU threads executing each of the algorithms can be ensured within *NVIDIA Streaming Multiprocessors* allowing the balanced utilization of all execution pipelines in a fashion similar to *Intel Hyper-Threading* technology. We demonstrated that this approach is feasible and brings a significant performance boost of up to 35%.

We developed a highly optimized implementation of Filtered Back Projection algorithm and evaluated it on NVIDIA Pascal micro-architecture. The proposed back-projection kernel is able to reach more than 90% of the theoretical throughput of the texture engine. We further developed a second implementation relying on a different set of hardware units. While the alternative algorithm is slightly slower on Pascal GPUs, running both algorithms in parallel results in a more balanced load of available hardware components. We measure an additional speed-up of 35% relative to the optimized texture-only version. Compared to the state-of-the-art implementation the hybrid reconstruction method performs 2.6 times better in the linear interpolation mode. The performance is boosted by 3.5 times if nearest-neighbor interpolation is performed. With minor modifications, both presented algorithms are also applicable to a wide range of other architectures from AMD and NVIDIA. The measured speed-ups range from 2 to 7 times depending on the considered architecture. While the proposed hybrid approach is only suited for recent NVIDIA architectures starting with Maxwell, different strategies to balance load are available for other GPU types. The high-speed reconstruction is of a significant importance for imaging at synchrotron facilities and allows to improve spatial and temporal resolutions of beam-line instrumentation. The back-projection algorithm is also utilized in slow iterative reconstruction techniques aimed at high-quality reconstruction. Therefore, the faster implementation lowers the computational demands for high-quality offline reconstruction as well.

VI. Acknowledgments

This work was partially supported by the German-Russian BMBF funding program, grant numbers 05K10CKB and 05K10VKE. The authors would like to thank to EXTREMA COST Action MP1207 for providing the networking support.

References

[1] R. Mokso, D. Schwyn, S. Walker, M. Doube, M. Wicklein, T. Müller, M. Stampanoni, G. Taylor, and H. Krapp, "Four-dimensional in vivo x-ray microscopy with projection-guided gating," *Scientific Reports*, vol. 5, p. 8727, 2015.

[2] E. Maire, C. Bourlot, J. Adrien, A. Mortensen, and R. Mokso, "20 hz x-ray tomography during an in situ tensile test," *Int. J. Fract.*, vol. 200, 2016.

[3] T. dos Santos Rolo, A. Ershov, T. van de Kamp, and T. Baumbach, "In vivo x-ray cine-tomography for tracking morphological dynamics," *Proceedings of the National Academy of Sciences*, vol. 111, no. 11, pp. 3921–3926, 2014.

[4] F. Marone, A. Studer, H. Billich, L. Sala, and M. Stampanoni, "Towards on-the-fly data post-processing for real-time tomographic imaging at tomcat," *Advanced Structural and Chemical Imaging*, vol. 3, no. 1, p. 1, 2017.

[5] M. Vogelgesang, T. Farago, T. F. Morgeneyer, L. Helfen, T. dos Santos Rolo, A. Myagotin, and T. Baumbach, "Real-time image-content-based beamline control for smart 4d x-ray imaging," *Journal of Synchrotron Radiation*, vol. 23, no. 5, pp. 1254–1263, 2016.

[6] A. Mirone, E. Brun, and P. Coan, "A dictionary learning approach with overlap for the low dose computed tomography reconstruction and its vectorial application to differential phase tomography," *PLOS ONE*, vol. 9, no. 12, pp. 1–18, 2014.

[7] G. V. Eyndhoven, K. J. Batenburg, D. Kazantsev, V. V. Nieuwenhove, P. D. Lee, K. J. Dobson, and J. Sijbers, "An iterative ct reconstruction algorithm for fast fluid flow imaging," *IEEE Transactions on Image Processing*, vol. 24, no. 11, pp. 4446–4458, 2015.

[8] F. Marone and M. Stampanoni, "Regridding reconstruction algorithm for real-time tomographic imaging," *Journal of Synchrotron Radiation*, vol. 19, pp. 1029–1037, 2012.

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

[10] M. Vogelgesang, S. Chilingaryan, T. dos Santos Rolo, and A. Kopmann, "Ufo: A scalable gpu-based image processing framework for on-line monitoring," in *Proceedings of The 14th IEEE Conference on High Performance Computing and Communication & The 9th IEEE International Conference on Embedded Software and Systems (HPCC-ICESS)*, ser. HPCC '12. IEEE Computer Society, 6 2012, pp. 824–829.

[11] W. van Aarle, W. J. Palenstijn, J. Cant, E. Janssens, F. Bleichrodt, A. Dabravolski, J. D. Beenhouwer, K. J. Batenburg, and J. Sijbers, "Fast and flexible x-ray tomography using the astra toolbox," *Opt. Express*, vol. 24, no. 22, pp. 25 129–25 147, 2016.

[12] F. Natterer and F. Wübbeling, *Mathematical Methods in Image Reconstruction*, ser. Mathematical Modeling and Computation. Society for Industrial and Applied Mathematics, 2001.

[13] F. Andersson, M. Carlsson, and V. V. Nikitin, "Fast algorithms and efficient gpu implementations for the radon transform and the back-projection operator represented as convolution operators," *SIAM Journal on Imaging Sciences*, vol. 9, no. 2, pp. 637–664, 2016.

[14] T. Zinsser and B. Keck, "Systematic performance optimization of cone-beam back-projection on the kepler architecture," in *Proceedings of the 12th Fully Three-Dimensional Image Reconstruction in Radiology and Nuclear Medicine*, 2013, pp. 225–228.

[15] E. Papenhausen and K. Mueller, "Rapid rabbit: Highly optimized gpu accelerated cone-beam ct reconstruction," in *IEEE Nuclear Science Symposium and Medical Imaging Conference (NSS/MIC)*, 2013.

[16] A. Hey, "The fft demystified," 1999. [Online]. Available: http://www.engineeringproductivitytools.com/stuff/T0001/PT10.HTM

[17] L. Shepp and B. Logan, "The fourier reconstruction of a head section," *IEEE Transactions on Nuclear Science*, vol. 21, 1974.

[18] "Cuda c programming guide," Manual, NVIDIA, 2017.