Reviewing GPU architectures to build efficient back projection for parallel geometries

Suren Chilingaryan[a], Evelina Ametova[b], Anreas Kopmann[a], Alessandro Mirone[c]

[a] *Karlsruhe Institute of Technology, Germany*
[b] *KU Leuven, Belgium*
[c] *ESRF, France*

Abstract

Back-Projection is the major algorithm in Computed Tomography to reconstruct images from a set of recorded projections. It is used for both fast analytical methods and high-quality iterative techniques. X-ray imaging facilities rely on Back-Projection to reconstruct internal structures in material samples and living organisms with high spatial and temporal resolution. Fast image reconstruction is also essential to track and control processes under study in real-time. In this article, we present efficient implementations of the Back-Projection algorithm for parallel hardware. We survey a range of parallel architectures presented by the major hardware vendors during the last 10 years. Similarities and differences between these architectures are analyzed and we highlight how specific features can be used to enhance the reconstruction performance. In particular, we build a performance model to find hardware hotspots and propose several optimizations to balance the load between texture engine, computational and special function units, as well as different types of memory maximizing the utilization of all GPU subsystems in parallel. We further show that targeting architecture-specific features allows one to boost the performance 2-7 times compared to the current state-of-the-art algorithms used in standard reconstructions codes.

Keywords: parallel algorithms, hardware architecture, GPU computing, synchrotron tomography, back-projection, CUDA, OpenCL

Email addresses: `chilingaryan@kit.edu` (Suren Chilingaryan), `evelina.ametova@kuleuven.be` (Evelina Ametova), `kopmann@kit.edu` (Anreas Kopmann), `mirone@esrf.fr` (Alessandro Mirone)

1. Introduction

X-ray tomography is a powerful tool to investigate materials and small animals at the micro- and nano-scale [1]. Information about X-ray attenuation or/and phase changes in the sample is used to reconstruct its internal structure. 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. At the Swiss Light Source (SLS) scientists were able to take high quality 3D snapshots of 150 Hz oscillations of a blowfly flight motor [2]. A temporal resolution of 20 milliseconds was achieved during a stencil test performed at SLS [3] and also in the analysis of morphological dynamics of fast-moving weevils at the ANKA synchrotron at KIT [4].

To achieve these results, the instrumentation used at imaging beamlines has recently undergone a major update. The installed streaming cameras are able to deliver up to hundreds of thousands of frames per second with a continuous data rate up to 8 GB/s [5]. Newly developed control systems at ANKA [6], SLS [5], and other synchrotron facilities use the acquired imaging information to track the processes under study and adjust the instrumentation accordingly. These control systems rely highly on the performance of the integrated image processing frameworks. Faster acquisition and a high level of automation is essential to study dynamic phenomena and at the same time enables experiments with significantly increased sample throughput. For example, in 2015 Diamond Light Source (DLS) has reported that typically about 3000 scans are recorded during 5 days of operation at a single imaging beamline [7]. Consequently, the amount of data generated at imaging beamlines quickly grows and results in a steep rise of the required computing power. In order to achieve higher temporal resolution and to prolong the duration of experiments, advanced methods are developed that incorporate a priori knowledge in the reconstruction procedure. These methods are able to produce high-quality images from undersampled and underexposed measurements, as demonstrated by [8, 9]. Unfortunately these methods are computationally significantly more demanding then traditional reconstruction algorithms and further increase the load on the computing infrastructure [10].

To tackle the performance challenge several reconstruction frameworks have been developed and optimized to utilize the parallel capabilities of nowadays computing architectures. At SLS *GridRec*, a fast reconstruction approach optimized for conventional CPU technology, has been adopted [11]. The reconstruction is scheduled across a dedicated cluster and reconstructs a 3D image within a couple of minutes [5]. Other frameworks use GPUs to accelerate the computation and are able to achieve minute-scale reconstructions at a single node equipped with multiple GPU adapters. PyHST is developed at ESRF and uses the CUDA framework to offload image reconstruction to NVIDIA GPUs [12]. The second version of PyHST provides also a number of iterative reconstruction techniques [13]. The UFO parallel computing framework is used at ANKA synchrotron to realize in-vivo tomography and laminography experiments [14, 15]. It constructs a data processing workflow by combining basic building blocks in a graph structure. OpenCL is used to execute the reconstruction at parallel accelerators with a primary focus on NVIDIA and AMD GPUs. ASTRA is a fast and flexible development platform for tomographic algorithms with MATLAB and python interfaces [16, 17]. It is implemented in C++ and uses CUDA to offload computations to GPU. Several other frameworks are based on the ASTRA libraries to provide GPU-accelerated reconstruction, for instance the Savu framework at DLS [7] or TomoPy at the Advanced Photon Source (APS) [18]. Recent versions of TomoPy also support *UFO* and *GridRec* as backends. All of the GPU-accelerated frameworks are capable to distribute the computation to a GPU cluster as well.

While most of the nowadays imaging frameworks rely heavily on parallel hardware to speed-up the reconstruction, specific features of the GPU architecture are rarely considered. On other hand, the hardware architectures differ significantly [19]. Organization of memory and cache hierarchies, performance balance between different types of operations, and even the type of parallelism varies. A significant speed-up is possible if details of the specific architecture are taken into account as illustrated in [20]. Fast execution is especially important if the reconstruction is embedded in a control workflow. Minimal latency is essential to track faster processes and to improve the achieved spatial and temporal resolutions. Due to unavoidable communication overhead, it is not always possible to reduce the latencies by scaling the reconstruction cluster.

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 [21].

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. For typical geometries Fourier-based methods are several times faster using the same computing hardware [22] and should be preferred if the computing infrastructure is limited to general-purpose processors only [5]. 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 [23]. However, this new method has not yet been adopted in production environments. Still, Filtered Back Projection is the method of choice, largely due to it simplicity and robustness. Therefore, the efficiency of the FBP implementation is still crucial for the operated monitoring and control systems. Furthermore, methods used for low dose tomography normally consist out of sequences of forward and back projections. And, thus, a faster implementation of the back projection lowers also the computational demands for high-quality offline reconstruction and might reduce the required hardware investments.

While there are several articles aiming at optimization of Back Projection for general-purpose processors and Intel Xeon-Phi accelerators [24], up to our knowledge there are no publications considering the variety of GPU architectures. A number of papers addresses specific GPU architectures [25, 26]. Multiple papers perform a general analysis of a range of GPU architectures, reveal undisclosed details trough micro-benchmarking, and propose guidelines for performance optimization [27, 28, 29]. This information is invaluable to understand factors limiting performance on a specific architecture and to find an alternative approach to achieve a better performance. Several papers propose methods to auto-tune computation kernels [30]. However, the tuning is limited to finding optimal configuration of pre-defined parameters like desired occupancy, dimensions of execution blocks, etc. For instance, there are no automated solutions to tune the balance between the texture engine and the computational cores.

In [31], we presented two highly-optimized back-projection algorithms for NVIDIA Pascal GPUs and a hybrid approach to balance the load between different GPU subsystems using both in parallel. While the algorithms can be used on different hardware, multiple modifications are required to address the differences in the architectures efficiently. Furthermore, the proposed hybrid approach is only suitable for the NVIDIA GPUs of a few latest generations. A different scheme to balance load is required for AMD, Intel, and older NVIDIA GPUs. In this paper, we review a variety of parallel architectures presented in last 10 years and establish a methodology to expand the original work to different parallel hardware. We discuss hardware differences in detail, build performance model, and demonstrate how the differences can be addressed to optimize the performance of the FBP algorithm on the existing parallel hardware. We also propose two new methods to balance the load between different GPU subsystems. One targets NVIDIA Kepler architecture and another can be applied universally but with a minor penalty to the quality. We will focus on the optimizations of the back-projection algorithm and will only briefly mention the organization of data flow as it is already explained in literature [12, 15]. We also do not cover scaling issues since the proposed optimizations can be easily integrated in existing frameworks like ASTRA, PyHST, or UFO which provide multi-GPU and GPU-cluster support already.

The article is organized as follows. The hardware setup, software configuration, and pseudo-code conventions are listed in section 2. A short introduction to parallel architectures that is required to understand the proposed optimizations is given in section 3. In this section we also highlight the differences between considered parallel architectures. The Filtered Back Projection algorithm and the state-of-the-art implementation are presented in section 4. A number of optimizations to the state-of-the-art implementation of the back-projection algorithm are proposed in section 5. An alternative implementation relaying on a different set of hardware resources is developed in section 6. A hybrid approach combining both approaches to fully utilize all hardware resources is presented in section 7. The achieved performance improvements are finally discussed in section 8.

2. Setup, Methodology, and Conventions

2.1. Hardware Platform

To evaluate the performance of the proposed methods, we have selected 9 AMD and NVIDIA GPUs with varying micro-architectures. Table 1 summarizes the considered GPUs. These GPUs were assembled into the 3 GPU servers. The newer NVIDIA cards with Maxwell and Pascal architectures were installed in a Supermicro 7047GT based server specified in Table 2. The older NVIDIA cards and all AMD cards were installed in two identical systems based on the Supermicro 7046GT platform. The full specification is given in the table 3. Additionally, we have tested how the developed code is performing on Intel Xeon Phi 5110P accelerator. The accelerator was installed in the first platform along with the newer NVIDIA cards.

Table 1: List of selected GPU architectures

Vendor	GPU	Arch.	Code	Release
NVIDIA	GeForce GTX 295	GT200	GT200	2009
NVIDIA	GeForce GTX 580	Fermi	GF110	2010
NVIDIA	GeForce GTX 680	Kepler	GK104	2012
NVIDIA	GeForce GTX Titan	Kepler	GK110	2013
NVIDIA	GeForce GTX 980	Maxwell	GM204	2014
NVIDIA	GeForce GTX Titan X	Pascal	GP102	2016
AMD	Radeon HD-5970	VLIW5	Cypress	2009
AMD	Radeon HD-7970	GCN1	Tahiti	2012
AMD	Radeon R9-290	GCN2	Hawaii	2013

Table 2: Server for newer NVIDIA cards

Platform	Supermicro 7047GT GPU Server
Motherboard	Supermicro X9DRG-QF with Intel C602 chipset
Memory	256 GB DDR3-133 Memory
Processor	Dual Intel Xeon E5-2640 (24 cores at 2.5 GHz)

Table 3: Servers for AMD and older NVIDIA cards

Platform	Supermicro 7046GT GPU Server
Motherboard	Supermicro X8DTG-QF with Intel 5520 chipset
Memory	96 GB DDR3-1066 Memory
Processor	Dual Intel Xeon X5650 (12 cores at 2.67 GHz)

2.2. Software Setup

All described systems were running OpenSuSE 13.1. The code for the NVIDIA cards was developed using the CUDA framework. As newer versions of the framework have dropped support for older GPUs, we have used CUDA 6.5 for the NVIDIA GeForce GTX295 card and CUDA 8.0 for other NVIDIA GPUs. The AMD version of the code is based on OpenCL and was compiled using AMD APPSDK 3.0. Additionally, we have tested the performance of Xeon CPUs and a Xeon Phi accelerator using Intel SDK for OpenCL. Since the latest version of Intel OpenCL SDK does not support Xeon Phi processors any more, again we needed to use two different SDK versions. The newer one was used to evaluate the performance of the Xeon processors while the older one served to execute the developed methods on the Xeon Phi accelerator. All installed software components are summarized in Table 4.

Table 4: Software components

Operating System	OpenSuSE 13.1
System Configuration	kernel 3.11.10, glibc 2.18, gcc 4.8.1
CUDA Platform	CUDA SDK 8.0.61, driver 375.39
CUDA Platform (GT200)	CUDA SDK 6.5.14, driver 340.102
AMD Platform	APP SDK 3.0.130.136, driver 15.12
Intel Platform	OpenCL SDK 2017 v. 7.0.0.2511
Intel Platform (Xeon Phi)	MPSS 3.5.1, OpenCL SDK 4.5.0.8

2.3. Benchmarking Strategy

In this article we are not aiming to precisely characterize the performance of the graphics cards, but rather validate the efficiency of the proposed optimizations. For this reason we take a relatively lax approach to the performance measurements. In most tests, we use a data set consisting of 2048 projections with dimensions of 2048 by 2048 pixels each. 512 slices with same dimensions are reconstructed and the median reconstruction time is used to estimate the performance.

Starting with the Kepler architecture, NVIDIA introduces the GPUBoost technology to adapt the clock speed according to the current load and the processor temperature [32]. To avoid significant performance discrepancies, we run a heat-up procedure until the performance stabilizes. Furthermore, we verify that the actual hardware clock measured before start of measurements (but after the heat-up procedure) does not significantly differ from the clock measured after the measurements. Otherwise, we re-run the test. Finally, we exclude all I/O operations in the benchmarks. The reconstructions are executed using dummy data and the results are discarded without transferring them back to the system memory.

2.4. Quality Evaluation

Some of the suggested optimizations alter the resulting reconstruction. To assess the effect on quality, we compare the obtained results with the standard reconstruction in such cases. The standard Shepp Logan Head Phantom with resoltion of 1024x1024 pixels is used for the evaluation [33]. As the changes are typically small and are hardly visible in the 2D image, we show a profile along a vertical line crossing most of the features in the phantom, see Figure 1.

2.5. Pseudo-code Conventions

To avoid long code listings we use pseudo-code to describe the algorithms. We use mixture of a mathematical and a *C*-style notation to keep it minimalistic and easy to follow. C syntax is mostly adapted for operations, loops, and conditionals. We use / to denote integer division and % for modulo operation. No floating point division is performed in any of algorithms. The division is always executed on positive integer arguments and produces integer number which is rounded towards zero. The standard naming scheme for variables is used across all presented algorithms. 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. Furthermore, some algorithms use shared memory to cache the data stored in global or constant memory. In such cases, we keep the variable name, but add superscript indicating the memory domain. For instance, c_s^S points to the sine of the projection angle stored in the shared memory. c is a group of variables storing the projection constants. c_s refers specifically the sine of the projection angle and the superscript $\cdot^S$ indicates that the copy in shared memory is accessed. All variables used across the algorithms are listed in Table 5, 6, 7. The superscripts used to indicate memory segment are specified in Table 8.

Table 5: List of parameters used in code snippets

Var	Type	Description
n_p	int	Number of projections
n_v	int	Number of slices reconstructed in parallel
n_q	int	Number of pixels assigned per GPU thread
n_s	int	The side of a pixel square reconstructed by a thread block
$\vec{n}_t$	int2	Dimensions of thread block
s_p	int	Size of the larger projection block, indicates the size of caches holding projection constants and h_m values
s_d	int	Size of data cache, specifies how many projection lines are cached
s_t	int	Number of threads assigned to cache a projection row, see section 6.3 and Table 14
s_i	int	Iterations required to completely cache a projection row (determined based $\vec{n}_t$, s_t, and the used caching optimizations as explained in section 6.3)
$\vec{v}_a$	float2	The position of rotation axis
c_c	float[]	Constant array storing cosine values of the projection angles
c_s	float[]	Constant array storing sine values of the projection angles
$\vec{c}_{cs}$	float2[]	Constant array storing (cosine, sine) pairs for each projection angle
c_a	float[]	Constant array storing coordinate of the rotational axis with applied per projection correction to compensate for possible mechanical displacements
c_m	float[]	Constant array storing coefficients required to quickly compute h_m

We use $\vec{\cdot}$ symbol to denote all vector variables, i.e. $float2$, $float4$, etc. Furthermore, all proposed algorithms are capable to reconstruct 1, 2, or 4 slices in parallel. If more than 1 slice is reconstructed, the accumulator and a few other temporary variables use the floating-point vector format to store values for multiple slices. These variables are marked with $\tilde{\cdot}$. All arithmetic operations in this case are performed in vector form and affect

Table 6: List of indexes used in code snippets

Var	Type	Description
$\vec{m}_b$	int2	The index of a thread block within the computation grid. Referred as *blockIdx* in CUDA or *get_group_id()* in OpenCL
$\vec{m}_t$	int2	The index of a thread with the thread block. Referred as *threadIdx* in CUDA or *get_local_id()* in OpenCL
$\vec{m}_g$	int2	The index of a thread within the computation grid, i.e. $\vec{m}_b * \vec{n}_t + \vec{m}_t$
$\vec{m}'_*$	int2	The re-mapped index, the number is specified in superscript if multiple mappings are used
$\vec{f}^*_g$	float2	The absolute coordinates of the reconstructed pixel according to the selected mapping, usually: $\vec{f}'_g = \vec{m}'_g - \vec{v}_a$
$\vec{f}_b$	float2	The absolute coordinates of a pixel block (i.e. coordinates of the pixel processed by the first thread of the block)
m_p	int	For algorithms processing multiple projections in parallel, it defines a projection index in a group
m_d	int	For algorithms caching the sinogram in shared memory, this is a mapping selecting offset in the cache
m_l	int	Linear addressing of threads in the thread block ($m_t.y{*}n_t.x{+}m_t.x$). It is another mapping used for caching constants.

Table 7: List of variables used in code snippets

Var	Type	Description
h	float	The required projection bin (including offset from the center)
h_i	int	The position of the required projection bin in the cache
h_f	float	The floating-point representation of h_i
h_l	float	The offset from the center of bin (i.e. coefficient for linear interpolation)
h_b	float	The bin required by the first thread of the block
h_m	float[]	The smallest bin required by a thread block in the selected projection row
h_x	float[][]	The cache storing the value of $c_a + f_g.x * c_c - h_m$ for each column of pixels processed by a thread block (and for each of s_d cached projections)
p_*	int	Projection number (p) and projection iterators (p_b, p_i)
q_*	int	Pixel block iterators
$\tilde{d}$	float[][]	The cache storing a subset of sinogram required to process s_d projections for the current thread block
$\tilde{s}$	float[]	Variable accumulating the impact of the projections. Defined as array if the thread is responsible for multiple pixels.
$\tilde{r}$	float[][]	The reconstructed slice

all slices. The vector multiplication is performed element wise as it would be in CUDA and OpenCL. We use the standard C notation to refer array indexes and components of the vector variables. The arrays are indexed from 0. For instance $\tilde{s}[0].x$ refers to the first component of the accumulator. The assignment between vector variable and scalars are shown using curly braces, like $\{x, y\} = \tilde{s}[0]$. The floating point constants are shown without C type specification. However, it is of utmost importance to qualify all floating-point constants as single precision in the C code, i.e. using $0.5f$ in place of 0.5. Otherwise the double-precision arithmetic will be executed severely penalizing performance on majority of consumer-grade GPUs.

Table 8: Memory Domains

Superscript	Domain
$.^G$	Variable in global GPU memory
$.^C$	Variable in constant memory
$.^S$	Variable in shared memory

To perform thread synchronization and to access the texture engine, the algorithms rely on a few functions provided by CUDA SDK or defined in the OpenCL specifications. To preserve neutrality of notation, we use abbreviated keywords to reference this functions. This list of used abbreviations along with the corresponding CUDA and OpenCL functions are listed in Table 9. Actually, the syntax of OpenCL and CUDA kernels is very closely related. Only a few language keywords are named differently. It is a trivial task to generate both CUDA and OpenCL kernels based on the provided pseudo-code.

Table 9: CUDA/OpenCL functions

Function	Description
sync	Denotes a synchronization point. The further execution is blocked until all threads of the block reach this point. It is implemented with *__syncthreads*() command in CUDA and *barrier*() with the *CLK_LOCAL_MEM_FENCE* type in OpenCL
fence	Enforces ordering of loads and stores. Equivalent to *__threadfence_block*() in CUDA and *mem_fence*() in OpenCL
tex2d	2D fetch from the texture mapped to the sinogram. It is implemented with *tex2D*() function in CUDA and *read_imagef*() in OpenCL.
shfl∗	A group of CUDA functions (*__shfl*, *__shfl_up*, *__shfl_down*, *__shfl_xor*) used to exchange data between the threads of a warp [34]. The vector types are not supported by CUDA functions. If *shfl* is applied to vector data, it is actually implemented as several calls to the corresponding function using all vector components one after another. There is no AMD counterpart of these functions.
floor	Rounding towards negative infinity

We use integer division and modulo operations across the code listings. These operations are very slow on GPUs and actually should be performed as bit mangling operations instead. However, the optimizing compilers can replace them automatically by the faster bit-mangling instructions. So, we are free to use notation which is easier to read. There are a few other cases where the optimization is left to the compiler.

3. Parallel Architectures

The architectures of nowadays GPUs are rather heterogeneous and includes multiple types of computational elements. The performance balance between these elements is shifting with each release of a new GPU architecture. To feed the fast computational units with data, a complex hierarchy of memories and caches is introduced. But the memories are very sensitive to the access patterns and the optimal patterns also differ between the hardware generations [34]. In this section we briefly explain the GPU architecture and elaborate differences between the considered GPUs with a focus on the aspects important to implement back projection efficiently. To simplify reading for a broader audience, we use the more common CUDA terminology across this paper.

3.1. Hardware Architecture

The typical GPU consists of several semi-independent Streaming Multiprocessors (SM) which share global GPU memory and L2 cache [35]. Several Direct Memory Access (DMA) engines are included to move data to and from system memory. Each SM includes a task scheduler, computing units, a large register file, a fast on-chip (*shared*) memory, and several different caches. There are a few types of computing units. The number crunching capabilities are provided by a large number of Arithmetic Units (ALU) also called *Core* units by NVIDIA. ALUs are aimed on single-precision floating point and integer arithmetic. Some GPUs also include specialized *half precision* and *double precision* units to perform operations with these types faster. There are also architecture-specific units. All NVIDIA devices include Special Function Units (SFU) which are used to quickly compute approximates of transcendent operations. The latest Volta architecture includes *Tensor* units aimed on fast multiplication of small matrices to accelerate deep learning workloads [36]. AMD architectures adapt scalar units to track loop counters, etc [37]. The memory operations are executed by Load/Store (LD/ST) units. The memory is either accessed directly or *Texture* units are used to perform a fast linear interpolation between the neighboring data elements while loading the data.

The computing units are not operating independently, but grouped in multiple sets which are operating in a Single Instruction Multiple Data (SIMD) fashion. Each set is able to execute the same instruction on multiple data elements simultaneously. Several such sets are included in SM and, often, can be utilized in parallel. The *SM scheduler* employs data- and instruction-level parallelism to distribute the work-load between all available sets of units. However, it is architecture depended which combination of instructions can be executed in parallel. The simplified and generalized scheme of GPU architecture is presented in Figure 2 and is further explained in the next subsections.

3.2. Execution Model

The GPU architectures rely on SIMT (Single Instruction Multiple Threads) processing model [34]. The problem is represented as a 3D *grid* of tasks or *threads* in CUDA terminology. All threads are executing the same code which is called *kernel*. The actual work of a thread is defined by its index (x, y, z) within the grid. Typically, a *mapping* between a thread index and image coordinates is established and each GPU thread processes the associated pixel or a group of pixels. Since memory access patterns matter, finding a suitable mapping has a very significant impact on the performance. In many practical applications, multiple mappings are used during the execution of a kernel. Particularly, all presented algorithms use 2 to 4 different mappings during the kernel execution.

The grid is split in multiple *blocks* of the same size. The blocks are assigned to a specific SM and are executed on this SM exclusively. Consequently, the information between threads of the same block can be exchanged using the fast shared memory local to SM. When a block is scheduled, all threads belonging to this block are made resident on the selected SM and all required hardware resources are allocated. A dedicated set of registers is assigned to each of the threads. However, not all threads of the block are executed simultaneously. The SM distributes resident threads between computational units in portions of 32/64 threads which are called *warps*. All threads of a warp are always executed simultaneously using one of available sets of units. If the execution flow within the warp diverges, it is executed sequentially: first all threads of the first branch are executed while others are kept idle and, then, vice-versa. To achieve optimal performance it is important to keep all threads of a warp synchronized, but the execution of complete warps may diverge if necessary. Similarly, the memory access patterns and locality are extremely important within a warp, less important within a block, but rather irrelevant between different blocks. GPUs always assign threads with consecutive indexes to the same warp and the thread mappings are always constructed with these considerations in mind.

At each given moment, the SM executes a few warps while several others are idle, either waiting for memory transaction to complete or for a set of units to become available. This is one of the mechanisms used to hide latencies associated with long memory operations. While one warp is set aside waiting for the requested data, the computational units are kept busy executing other resident warps. As the registers are assigned to all threads permanently and are not saved/restored during scheduling, the switching of the running warp inflicts no penalty.

3.3. Memory hierarchy

Compared to a general-purpose processor the ratio between computational power and throughput of the memory subsystem is significantly higher on GPUs. To feed the computation units with data, the GPU architectures rely on multiple types of implicit and explicit caches which are optimized for different use cases. Furthermore, the maximum bandwidth of GPU memory is only achieved if all threads of a warp are accessing neighboring locations in memory. For optimal performance some architectures may require even stricter access patterns.

There are 3 types of general-purpose memory available in the GPU. A large amount of *global memory* is accessible to all threads of the task grid. Much smaller, but significantly faster *shared memory* is local to a thread block. The thread-specific local variables are normally hold in registers. If there is not enough register space, a part of variables may be offloaded to the *local memory.* The thread-specific, but dynamically addressed arrays are always stored in the local memory (i.e. if array addresses can't be statically resolved during the compilation stage). In fact, the local memory is a special area of the global memory. But the data will be actually written and read to/from L1 or L2 cache unless an extreme amount of local memory is required. Even then, access to variables in the local memory inflicts a severe performance penalty compared to the variables kept in the registers and should be avoided if possible.

To reduce the load on the memory subsystem, GPUs try to coalesce the global memory accesses into as few transactions as possible. This can only be realized if the threads of a warp are addressing adjacent locations in the memory. The memory controller aggregates the addresses requested by all threads of a warp and issues a minimal possible amount of 32- to 128-byte wide transactions. These transactions are subject to alignment requirements as well. It does not matter in which order the addresses are requested by the threads of a warp. The maximum bandwidth is achieved if as few as possible of such transactions are issued to satisfy the data request of the complete warp. This was different in older hardware when the stricter access patterns had to be followed. If it is not possible to implement coalesced access strategy, the shared memory is often used as explicit cache to streamline accesses to the global memory [38].

The shared memory is composed out of multiple data banks. The banks are 32- or 64-bit wide and are organized in a such way that successive words are mapped to successive banks. The shared memory bank conflict occurs if the threads of a warp are accessing multiple memory locations belonging to the same bank. The conflicts causes warp serialization and may inflict a significant penalty to the shared memory bandwidth. Furthermore, the achieved bandwidth depends on a bit-width of the accessed data. The Kepler GPUs are equipped with 64-bit shared memory and only deliver full bandwidth if 64-bit data is accessed. While the AMD Cypress and Tahiti GPUs are equipped 32-bit shared memory, the performance is still considerably improved if 64-bit operations are performed. Increasing the data size beyond 64-bit has a negative impact on the performance on some architectures. 128-bit loads from shared memory always cause bank conflicts on NVIDIA GT200, NVIDIA Fermi, and all AMD architectures. We tackle the differences between shared memory organization in sections 6.3 and 6.4.

Most of the GPU architectures provide both L1 and L2 caches. However, the amount of the cache per compute element is quite low. On NVIDIA Fermi and Kepler GPUs, both L1 cache and shared memory are provided using the same hardware unit and the ratio between the size of L1 cache and the shared memory is configured at compilation stage [35, 39]. Only buffers that are read-only during a complete execution of a kernel are usually cached in L1. This property is not always detected by the compiler and should be either hinted in the code or enforced using a special CUDA intrinsic instruction [39]. There are two additional caches optimized for specific use-cases. The *constant memory* is used to store parameters which are broadcasted to all threads of the grid. For optimal performance 64-bit or 128-bit access is required [40]. The texture engine provides a cache optimized for spatial access. While the line of L1 cache is typically 128-byte long, the texture cache operates with lines of 32-bytes allowing to fetch the data from multiple rows of an image as required to perform bi-linear interpolation.

3.4. Texture Engine

The texture engine associates a dimensional information with buffers in the global GPU memory [41]. By doing so, it is able to interpret the memory as a multi-dimensional object and perform implicit interpolation if a texel with fractional coordinates is requested. Nearest-neighbor or linear interpolation modes are supported. The texture engines are able to work with a variety of data types. Besides simple integer and floating-point numbers, they are also capable to interpolate and return the values encoded in standard vector types. The performance is defined by the number of texels processed per time unit and is called *texture filter rate*. Up to a threshold, 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-bits are used to encode the texel values. For the larger vector types the theoretical filter rate, however, is not actually reached. Depending on the GPU architecture, a maximum 32-, 64-, or 128-bit values are processed at a full speed.

To achieve maximum performance it is also necessary to ensure the spatial locality of the texture fetches. The locality is important at several levels. At a block level it results in a high level of texture cache utilization. A more dense access layout within a warp reduces the number of required transactions to the texture cache. While it is not documented, the distribution of the fetch locations between groups of 4 consecutive threads impacts performance significantly if a bi-linear interpolation is performed. To verify it, we developed a small benchmark using the techniques proposed by Konstantinidis and Cotronis for *gpumembench* and *mixbench* suites [40, 42]. Figure 3 shows two different thread assignments to fetch 16 texels from a 4-by-4 pixel square. The fetched coordinates are always slightly shifted from the pixel centers to ensure that the bi-linear interpolation is actually executed. There is a little difference if 32-bit data is accessed. For 64-bit data, however, the thread assignments following Z-order curve reach almost 100% of the theoretical maximum while only 50% is achieved if simple linear layout is used. Section 5.5 discusses the effect of the optimized fetch locality on a performance of tomographic reconstruction.

We also used the developed benchmark to find the maximum size of fetched data which is still filtered at full speed. Our results show that all NVIDIA GPUs starting with Fermi benefit from the 64-bit texture fetches if requests are properly localized. It is also supported by the latest of the considered chips from AMD. However, the OpenCL kernel must be compiled with OpenCL 2.0 support enabled. It is done by passing *-cl-std=CL2.0* flag to *clBuildProgram*() call. Otherwise, the full performance is only achieved if the nearest-neighbor interpolation is performed. This is always the case for older AMD devices. If the texture engine is configured to perform linear-interpolation on 64-bit data, only the half of throughput is delivered on these

AMD architectures. On other hand, all AMD devices are able to deliver the full performance using the 128-bit data if the nearest-neighbor interpolation is utilized. The NVIDIA devices are limited by 64-bit in both cases.

3.5. Task partitioning

The number of resident threads directly affects the ability of the SM to hide memory latencies. Each architecture limits the maximum number of resident warps per SM. Since SM has only a limited amount of registers and shared memory, the actual number of resident warps could be bellow this limit. The ratio between the actual and the maximum number of resident warps is called *occupancy* and has a significant impact on the performance. The complexity of the kernel dictates how many registers is required per thread and, hence, restricts the maximum amount of resident threads on the SM. It is possible to target occupancy on NVIDIA platform. If a higher occupancy is requested, the CUDA compiler either reduces the number of used registers in a price of repeating some computations or offloads part of the used variables in the local memory. Vice-versa, the compiler may perform more aggressive caching and pre-fetching if lower occupancy is targeted. Both approaches may significantly improve the performance under different conditions. The optimal occupancy depends on both, work-load and hardware capabilities. On one hand, it should be high enough to ensure that the SM scheduler always has warps ready to execute. On other hand, prefetching may significantly improve performance of memory bound applications. Furthermore, offloading variables to local memory will not necessarily harm the performance if the local memory is fully backed by L1 cache. Consequently, more registers can be made available for prefetching also without decreasing occupancy. However, the shared memory available to applications is reduced on Fermi and Kepler platforms if a large amount of L1 cache is dedicated to the local memory. A very detailed study of the optimal occupancy under different workloads is performed in the PhD thesis of Vasiliy Volkov [27]. We study the effect of occupancy tuning on the performance of the back projection kernel in sections 5.7 and 6.9. Both reduced and increased occupancy are found practically useful in different circumstances.

GPUs have varying limits on a number of threads allowed per block. To achieve a higher occupancy, multiple thread blocks can be scheduled on the same SM simultaneously. The maximum number of resident blocks is architecture dependent and is further restricted by the requested amount of shared memory. The required shared memory is not always proportional to the size of a thread block. The larger blocks may require less shared memory per thread. As the block is always made resident as a whole, some configurations are better mapped to available resources while other leave part of the memory unused.

3.6. Code Generation

Even the fast shared memory has a relatively high latency [28]. Consequently, GPU vendors prvide several mechanisms to hide this latency and preserve the high memory bandwidth. The thread is not stalled until the executed memory operation is finished. The GPU scheduler launches the operation, but proceeds issuing independent instructions from the execution flow of the thread until the requested data is actually required. If the next instruction in the flow depends on the result of the memory operation which is not completed yet, the SM puts the thread aside and schedules another resident thread as stipulated by SIMT execution model. For compute-bound applications, the optimizing compiler re-arranges instructions to interleave memory operations with independent arithmetic instructions and uses both described mechanisms to avoid performance penalties due to memory latencies [27].

If an application is memory bound, the compiler vice-versa groups multiple load operations together to benefit from streaming. The latency, then, has to be hidden only a single time for all load operations which are streamed together. This mechanism is of a great importance to perform texture fetches as a texture cache hit reduces usage of memory bandwidth, but not the fetch latency [34]. Furthermore, several 32-bit loads from the consecutive addresses may be re-combined by a compiler in a single 64- or 128-bit memory instruction. It reduces the number of issued instructions and gives the warp scheduler an opportunity to increase the Instruction Level Parallelism (ILP) by launching additional instructions in the vacated execution slots. With the Kepler architecture, this scheme may even double the shared memory bandwidth by utilizing 64-bit memory banks more efficiently. Several papers show a significant performance improvement also on other architectures [29].

The described optimizations are performed automatically by the compilers from AMD and NVIDIA. The loops are unrolled and instructions are re-arranged as necessary to increase the hardware utilization. The loop unrolling not only allows the compiler to optimize the instruction flow, but also reduces the load on the ALUs. In particular, the computation of array indexes is replaced by static offsets at compilation stage. In some cases, however, it is possible to further improve the generated code by enforcing the desired unrolling factors and by targeting the occupancy. This is discussed in section 6.9. Furthermore, the data layout may be adjusted in order to give compiler more options in optimizing the code flow. The algorithm described in section 5.6 relies on a large number of independent operations to compensate the low occupancy. In section 6.8 we optimize the data layout to enable the re-combination of memory instructions.

3.7. Scheduling

To provide high performance, the GPU architectures include multiple components operating independently. Texture fetches, memory operations, several types of arithmetic instructions are executed by different blocks of GPU in parallel. Hence, the kernel execution time is not determined as a sum of all operations, but rather is given by the slowest execution pipeline. One strategy to implement an efficient algorithm is to balance operations between available GPU blocks uniformly and minimize the time required to execute the slowest pipeline. Using this methodology we were able to gain significant performance improvements. Section 6.6 discusses balancing of SFU and ALU operations to speed-up the linear interpolation on the Kepler architecture. Two different back-projection algorithms are combined in section 7.1 to balance the load across all major GPU subsystems. As result the proposed hybrid approach outperforms the fastest of the algorithms by 40% on Pascal and Maxwell architectures.

Each SM includes one or more *warp schedulers* which execute instructions of resident warps. Each scheduler is able to issue either a single instruction per-clock or at each clock to *dual-issue* two independent instructions from the same warp. On most architectures the number of warp schedulers is synchronized with the number of independent ALU units. All available units are fully utilized if a single ALU instruction is scheduled by each warp scheduler at every clock cycle. The SM processor on Kepler, however, includes 6 sets of ALUs, but only 4 warp schedulers [39]. To achieve 100% utilization all SM schedulers are expected at each second clock cycle to select two independent instructions from the execution flow and dispatch them to 2 different sets of ALU units. The VLIW architecture adopted by the older AMD GPUs requires 4 to 5 independent instructions in the flow for optimal performance [43]. The flow of independent instructions and dual-issue capabilities are also required to utilize multiple functional blocks of GPU in parallel.

Only a little official information is available about instructions which can be schedulled in parallel. The CUDA C Programming Guide states that SFUs are used to compute approximates of transcendent functions [34]. In fact, they are also used to perform bit-mangling, type-conversions, and integer multiplication on the NVIDIA Kepler, Maxwell, and Pascal GPUs. We developed a micro-benchmark to verify if certain instructions can be dual-issued. The idea is to measure the throughput of each individual instruction and, then, compare it to throughput of their combination. The instructions are assumed to be executed by the same function unit if the combination runs slower than the slowest of the individual instructions. In particular we found out that NVIDIA GPUs starting with Kepler execute rounding, type conversion, and bit-shift operations in parallel with ALU instructions, but slow down the computation of sine and cosine approximates. Consequently, we assume that SFUs are used to execute these operations. On Maxwell and Pascal, the bit-wise operations also slow down ALU instructions slightly. Both SFUs and ALUs are used in this case. However, the decrease is small and additional ALU-operations are still possible to execute in parallel. There is no parallelism of these operations on the AMD platform.

3.8. Synchronization

The GPU memory hierarchy and a few synchronization primitives are used to efficiently coordinate work between threads. The fast shared memory is used to exchange information between threads of the same block. An even faster shuffle instruction is available on NVIDIA GPUs since the Kepler generation. It allows to exchange data stored in the registers of multiple threads belonging to the same warp [39]. Both CUDA and OpenCL provide a fast synchronization instruction which ensures that all threads of a block have completed the assigned part of the work and reached the synchronization point. This allows to split execution of a kernel in multiple phases with different thread mappings. For example in the algorithm described in section 6.2, the threads are first mapped to the elements of a cache and are used to prefetch data from global memory. After synchronization the threads are re-assigned to the pixels of output image and use the cached information to compute their intensities.

The synchronization may restrict the ability of the SM schedulers to benefit from the ILP parallelism if the groups of instructions aimed on different functional units are separated by a synchronization primitive. Partial remedy is to allocate more resident blocks to SM by increasing occupancy or by using smaller blocks. Still, a well composed code usually results in better performance if it allows the compiler to re-arrange execution flow and dual-issue instructions.

3.9. Communication

Most GPUs include a pair of DMA engines and are able to perform data transfers over the PCIe bus in both directions in parallel with kernel executions. This, however, requires page-locked (non swappable) host memory. While OpenCL does not define how the page-locked memory can be obtained, in practice it can be done by allocating a host-mapped GPU buffer. This is realized by calling *clCreateBuffer* with *CL_MEM_ALLOC_HOST_PTR* flag. While only the host buffer is required in this case, the command allocates also the GPU buffer. The memory overcommitting is, however, supported on NVIDIA platform. Consequently, only the host memory is actually reserved. The corresponding GPU buffer is never accessed and, correspondingly, the GPU memory is not reserved. On the AMD platform, however, the memory is actually set aside for both buffers immediately. Consequently, the amount of GPU memory available to application is reduced. To enable parallel data transfer and computations, double buffering technique along with asynchronous CUDA/OpenCL API are typically used. The CUDA/OpenCL events are used for synchronization.

In addition to the DMA engines used for communication with the host memory, the professional series of GPUs also support a slave mode of DMA operation. In this mode the other devices on the PCIe bus are able to write data directly into the GPU memory. Starting with the Kepler micro-architecture, this feature is supported by the NVIDIA Tesla cards using the GPUDirect technology [44]. AMD provides the DirectGMA technology to enable the feature on the GCN-based AMD FirePro cards [45]. The GPUDirect technology is already used in several MPI frameworks to speed-up communication in Infiniband networks [46].

3.10. Summary

We summarize the properties of target GPUs in Table 10. Besides the hardware specification available in the vendor white papers, we present architecture-specific information obtained using micro-benchmarking and further investigate the performance balance of different operations. Only characteristics important to implement fast back-projection kernel are included. For this reason, we only report throughput of the floating-point, bit-mangling, and type-conversion instructions.

Compared to the GT200, the Fermi architecture significantly improved the arithmetic capabilities, but the texture filter rate has not changed. Instead, the texture units got the ability to fetch 64-bit data at full speed. The Fermi GPUs also lost the capability to dual-issue instructions from the same warp and are the most restricted architecture of the considered ones concerning the ability to schedule instructions to different execution pipelines in parallel. Consequently, the Fermi performance is likely improved if the number of the required instructions is reduced. One option is to organize the data in a way allowing wider 64/128-bit memory operations and texture fetches. The Kepler architecture massively improved the performance of the texture engine. But the throughput of integer, bit-mangling, and type-conversion operations has actually slowed down compared to the Fermi devices. Furthermore, the ILP become a necessity for optimal performance. On Pascal, the amount and performance of the shared memory has doubled. While the amount of available registers has not changed, the generated code is typically requires less registers. Consequently, it is either possible to achieve higher occupancy or execute more sophisticated kernels at the same occupancy.

There is a few important differences between NVIDIA and AMD platforms. AMD provides less control over the code-generation. The NVIDIA compiler can be parametrized to use less registers for generated code. This option is not available for AMD. Neither of the considered AMD devices support the half-precision extension of the OpenCL specification. While we can use the smaller data representation to reduce texture and shared memory bandwidth on NVIDIA platform, it is not possible to achieve it with AMD. On the other hand, the AMD devices are capable to perform full-speed texture filtering also using 128-bit data if the nearest-neighbor interpolation is selected. Furthermore, the ratio between the shared memory throughput and the performance of the texture engine is 2 - 4 times higher on AMD devices. Consequently, it is more likely that caching of the fetched data in the shared memory will result in performance improvements. The organization of AMD Cypress GPUs differs from the other considered architectures significantly. It has very slow constant memory and relies on ILP parallelism extensively. Five instructions has to be scheduled at each clock cycle for optimal performance. Vice-versa the GCN-based devices do not provide ILP. There is also no parallelism between floating-point and bit-mangling/type-conversion instructions. The throughput of arithmetic operations is comparatively slow and is bottleneck for the proposed algorithms. There are also minor difference between two generation of GCN platform. The first generation of GCN chips performs better if 64-bit operations are performed on the shared memory. This is not required in the second generation of the architecture anymore. Starting with GCN2, the AMD devices are capable to perform 64-bit texture fetches at full pace also if bi-linear interpolation is employed.

To build an efficient implementation of the algorithm it is important to account for the described architectural differences. Across all architectures a good locality of the texture fetches has to be ensured and optimal access patterns to global and shared memory has to be followed. It is necessary to adjust the algorithm flow to balance the load between different execution pipelines according to their hardware capabilities. Finally, also the right balance between ILP, streaming memory operations, and achieved occupancy has to be found.

4. Tomographic Reconstruction

At synchrotron imaging beamlines, information about X-ray attenuation or/and phase changes in the sample is used to reconstruct its internal structure. The objects are placed on a rotation stage in front of a pixel detector and rotated in equiangular steps. As the object rotates, the pixel detector registers a series of two-dimensional intensity images of the incident X-rays. Typically the X-rays are not detected directly, but converted to visible light using a scintillator placed between the sample and pixel detector. Then, the conventional CCD cameras are used to record intensities which actually correspond to projections of the sample volume. Due to the rather large source-to-sample distance, imaging at synchrotron light sources is usually 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 series of 2D reconstructions performed with cross-sectional slices. An origin of coordinate system coincides with center of sample rotation stage and rotation axis is anti-parallel to gravity. To reconstruct a slice, the projection values are "smeared" back over the 2D cross section along the direction of incidence and are accumulated over all projection angles. To compensate blurring effects, high-pass filtering of the projection data is performed prior to back projection [21].

Table 10: List and specification of considered GPU architectures

	NVIDIA GeForce [34]						AMD Radeon [47]		
	GTX295[1]	GTX580	GTX680	Titan	GTX980	Titan X	HD5970[1]	HD7970	R9-29
Architecture	GT200	Fermi	Kepler	Kepler	Maxwell	Pascal	Cypress	Tahiti	Hawa
Architecture Code Name	GT200	GF110	GK104	GK110	GM204	GP104	VLIW5	GCN1	GCN
Release Year	2009	2010	2012	2013	2015	2016	2009	2012	2013
Reference	[48]	[35]	[49]	[39]	[50]	[51]	[52]	[37]	[37]
Global Memory									
Global Memory (GB)	0.9	1.5	2	6	4	12	1	3	4
Memory Bandwidth (GB/s)	112	192	192	288	224	480	128	264	320
L2 Cache (KB)	-	768	512	1536	208	3072	512	768	1024
L2 Bandwidth (GB/s)	-	296	515	763	641	1351	371	710	970
Execution Units									
Number of SM	30	16	8	14	16	28	20	32	40
ALU Reference Clock (MHz)	1242	1544	1006	837	1126	1417	725	925	947
ALU Max Turbo Clock (Mhz)	-	-	1110	1202	1392	1911	-	-	-
ALU Benchmark Clock (MHz)[3]	1242	1544	1006	993	1252	1759	725	925	947
Warp Schedulers (per SM)	1	2	4	4	4	4	1	5	5
Max Instructions per Warp	2	1	2	2	2	2	5	1	1
ALU Units (per SM)	8	2x16	6x32	6x32	4x32	4x32	16x4	4x16	4x16
SFU Units (per SM)	2	4	32	32	32	32	16	-	-
Texture Units (per SM)	2.66[2]	4	16	16	8	8	4	4	4
ILP Required for Peak GFlops	Yes	No	Yes	Yes	No	No	Yes	No	No
Hardware resources									
Warp Size	32	32	32	32	32	32	64	64	64
Max Resident Warps (per SM)	32	48	64	64	64	64	24	40	40
Shared Memory (KB/SM)	16	16-48	16-48	16-48	96	96	32	64	64
Registers (KB/SM)	64	128	256	256	256	256	256	256	256
Max 32-bit regs. per thread	128	63	63	255	255	255	248	256	256
Regs. Per Thread at Full Occupancy	16	21	32	32	32	32	40	25	25
Shared & Constant Memory									
Shared Memory Banks	16	32	32	32	32	32	32	32	32
Sh.Mem Bank Width (bits)	32	32	64	64	32	32	32	32	32
Sh.Mem Bank Broadcasts	Yes	Yes	Yes	Yes	Yes	Yes	No	Yes	Yes
Speed-up using 64-bit Loads[4]	-	-	100%	100%	-	-	15%[4]	40%	-
Conflict-free Loads (up to, bits)	32	64	128	128	128	128	64	64	64
Sh.Mem Max Bandiwdth (GB/s)	1324	1581	2060	3559	2564	6304	1856	3789	4849
C.Mem. Max Bandwidth (GB/s)[5]	875	1511	1980	3120	4186	11500	928	7578	9697
Instruction throughput									
Units executing FP-insructions	ALU,SFU	ALU	ALU	ALU	ALU	ALU	ALU,SFU	ALU	ALU
Units executing bit-shifts[6]	ALU	ALU	SFU	SFU	ALU,SFU	ALU,SFU	SFU	ALU	ALU
Units executing type-conversions[6]	ALU	ALU	SFU	SFU	SFU	SFU	SFU	ALU	ALU
FP Performance (GFlops)[7]	994[8]	1581	3090	5338	5128	12608	2320	3789	4849
Bit-shift Performance (G-ops)	331	395	258	444	1282	3152	232	1894	2424
Type-mangling performance (G-ops)[9]	331	395	258	444	641	1576	232	1894[10]	2424
Performane of Texture Engine									
Texture Engine (GT/s)	51	49	129	222	160	394	58	118	152
TE, 64-bit Data, Bi-linear (GT/s)[6]	25	49	123	204	156	398	26	55	113
TE, 64-bit Data, Nearest (GT/s)[6]	25	50	132	212	156	400	52	103	131
TE, 128-bit Data, Nearest (GT/s)[6]	12	25	70	114	79	200	49	116	147
Performance Ratios									
Constant to Shared Memories	1	1	1	1	2	2	0.5	2	2
C.Mem to Texture (Words/Texels)	6.5	8	4	4	8	8	4	16	16
Sh.Mem to Texture (Words/Texels)	6.5	8	4	4	4	4	8	8	8
Type-conv to Texture (Ops/Texels)	6.5	8	2	2	4	4	4	16	16
GFlops to Texture (Ops/Texels)	19.4	32	24	24	32	32	40	32	32
GFlops to Sh.Mem (Ops/Words)	3	4	6	6	8	8	5	4	4
GFlops to Type-conversion	3	4	12	12	8	8	10	2	2

The presented numbers are either taken from the referenced programming guide and specifications or computed based on the oth
presented values. All exceptions which are obtained using micro-benchmarking are indicated with footnotes.

1 The characteristics for a single GPU core are given

2 On GT200 the texture units are not included in SM, but are part of Texture Clusters which includes several SM

3 GPUBoost technology adjusts clock according to load and temperature. In this row we specify the approximate clock rate during t
benchmarks

4 Using 64-bit loads are only faster if two shared memory operations can't be combined in a single VLIW instruction

5 On NVIDIA platform the bandwidth of constant memory is obtained with benchmarking

6 Measured using micro-benchmarking

7 MAD/FMA are counted as two operations

[illegible]

The typical reconstruction data flow using parallel accelerators is represented on Figure 4. The projections are loaded into the system memory either from a storage facility or directly from a camera and, then, transferred into the GPU memory before executing pre-processing or reconstruction steps. From cameras equipped with PCIe-interface it is also possible to transfer the projections directly into the GPU memory using GPUDirect or DirectGMA technologies. The later is supported by UFO framework [15]. The loaded projections are pre-processed with a chain of filters to compensate the defects of optical system. Then, the projections are rearranged in order to group together the chunks of data required to reconstruct each slice. These chunks are called *sinograms* and are distributed between parallel accelerators available in the system in a round-robin fashion. Filtering and back-projection on each slice are performed on each GPU independently, the results are transferred back, and are either stored or passed further for online processing and visualization. To efficiently utilize the system resources, usually all described steps are pipelined. The output volume is divided into multiple subvolumes, each encompassing multiple slices. The data required to reconstruct each subvolume is loaded and send further trough the pipeline. While next portion of the data is loaded, the already loaded data is preprocessed, assembled in sinograms, and reconstructed. The preprocessing is significantly less compute-intensive compared to the reconstruction and is often, but not always, performed on CPUs. OpenCL, OpenMP, or POSIX threads are used to utilize all CPU cores. The pre-processed sinograms are, then, distributed between GPUs for reconstruction. For each GPU a new data pipeline is started. While one sinogram is transferred into the GPU memory, the sinograms already residing in GPU memory are first filtered, then back projected to the resulting slice, and finally transferred back to the system memory. Event-based asynchronous API and double-buffering are utilized to execute data transfer in parallel with reconstruction. Basically, such approach allows to use all system resources including Disk/Network I/O, PCIe bus, CPUs, and GPUs in parallel.

A single row from each of the projections is required to reconstruct a slice of output 3D volume. These rows are grouped together in a sinogram. For the sake of simplicity, we refer to these rows as *projections* while discussing reconstruction from sinograms. Each slice is reconstructed independently. First, each sinogram row is convolved with a high-pass filter to reduce blurring - an effect inherent to back-projection. The convolution is normally performed as multiplication in Fourier domain. The implementation is based on available FFT libraries. NVIDIA *cuFFT* is used on CUDA platform and either AMD *clFFT* or Apple *oclFFT* is utilized for OpenCL. For optimal FFT performance, multiple sinogram rows are converted to and from FFT domain together using batched transformation mode. After filtering, the buffer with filtered sinograms is either bind to texture on CUDA platform or copied into the texture if OpenCL is used. The pixel-driven approach is used to compute back-projection. For each pixel (x, y) of the resulting slice, the impact of all projections is summed. This is done by computing the positions r_p where the corresponding back projection rays are originated and interpolating the values of projection bins around this position.

If α is an angle between consecutive projections, the positions are computed as follows:

$$r_p(x, y) = x \cdot \cos(p\alpha) - y \cdot \sin(p\alpha) \tag{1}$$

As computation of trigonometric function is relatively slow on all GPU architectures, the values of $cos(p\alpha)$ and $sin(p\alpha)$ are normally pre-computed on CPU for all projections, transferred to GPU constant memory, and, then, re-used for each slice. Assuming this optimization, the back projection performance is basically determined by how fast the interpolation could be made. Two interpolation modes are generally used. The nearest neighbor interpolation is faster and better at preserving the edges while the linear interpolation reconstructs the texture better. While more sophisticated interpolation algorithms can be used as well, they are significantly slower and are rarely if ever used. All reviewed reconstruction frameworks rely on GPU texture engine to perform interpolation. This technique was first proposed in the beginning of the nineties for the *SGI RealityEngine* [53].

5. Back-projection based on Texture engine

The standard implementation described in previous section performs fairly good. The compilers included in the CUDA Framework and AMD APPSDK are optimize the execution flow automatically. The loops are unrolled and the operations are re-arranged to allow streaming texture loads as explained in the section 3.6. Still, the default implementation does not utilize all capabilities of texture engine and significant improvement can be achieved on all architectures.

5.1. Standard version

First, we will detail how the standard implementation works. Each GPU thread is responsible for a single pixel of output slice and iterates over all projections to sum the contribution from each one. At each iteration, a projection is performed to find a coordinate where the ray passing through the reconstructed image pixel hits the detector. The value at the corresponding position in the sinogram row 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. The projection is computed according to equation 1. To align the coordinate system with rotational axis, the position of the rotational axis is first subtracted from the pixel coordinates and, then, added to the computed detector coordinate to find the required position in the sinogram. To compensate for possible distortions of imaging system during the experiment, the rotational center is not constant, but may include per-projection corrections. Sine and cosine of each projection angle as well as the corrected position of the rotation axis are read from a buffer in the constant memory which is generated during the initialization phase. The computation grid is split in square blocks of 16-by-16 threads. It results in optimal occupancy across all considered platforms. The corresponding pseudo-code is presented in Algorithm 1.

```
Input: Texture and the projection constants $c_*^C$. Dimensions ($n_*$) and parameters
       ($v_*$) as specified in Table 5. The indexes ($m_*$) and other used variables are
       described in Table 6 and 7
Output: Reconstructed slice $\tilde{r}^G$
begin
    $\tilde{r} = 0$
    $\vec{f}_g = \vec{m}_g - \vec{v}_a$
    for ($p = 0$; $p < n_p$; $p$ += 1)
        $h = c_a^C[p] + f_g.x * c_c^C[p] - f_g.y * c_s^C[p]$
        $\tilde{r}$ += tex2d($h + 0.5$, $p + 0.5$)
    end
    $\tilde{r}^G[m_g.y, m_g.x] = \tilde{r}$
end
```

Algorithm 1: Standard implementation of the back-projection kernel

The CUDA platform supports two slightly different approaches to manage textures: the texture reference API and the texture object API [34]. The texture reference API is universal and is supported by all devices. The texture object API is only supported since Kepler architecture. While the reference API can be used on all devices, as we found out the object API outperforms it on the devices with compute compatibility 3.5 and later. Therefore, we use the reference API for GT200, Fermi, and the first generation of Kepler devices and the object API for all newer architectures.

5.2. Multi-slice reconstruction

The texture engines integrated in all recent generations of GPUs are capable filter 8-byte data at the full pace, see section 3.4. The standard reconstruction algorithm can benefit from this feature only if changed to double-precision for better accuracy. But this have a little use in practice. 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 the several individual sinograms as vector data. 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, see Figure 5. With *float2*-typed texture mapped on this interleaved sinogram, it is possible to fully utilize the bandwidth of the texture engine and reconstruct two slices in parallel. The interleaving is done as an additional data preparation step between filtering and back projection steps. The back projection kernel is, then, 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. There is a considerable speed-up on all relevant architectures as can be seen on Figure 6.

5.3. Using half-precision data representation

Since the NVIDIA texture engine is currently limited to 8-byte vectors, the proposed approach can't be scaled to 4 slices if the single-precision input is used. However, CUDA supports half-precision data type which encodes each floating-point number using 16 bits only. While 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. Unfortunately, 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 texture. Still, it is possible to speed-up the reconstruction if the nearest-neighbor interpolation mode is selected. After filtering, the sinograms are down-sampled to the half-precision format and interleaved. 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 casted to single-precision using *__half22float2* instruction and all further operations are performed in single-precision as usual.

The Figure 6 indicates a significant speed-up on all NVIDIA architectures except Kepler. As can be seen from Table 10, the type casting is very slow on Kepler and caps the performance gains. The proposed method is also not viable on AMD platform. Neither of the considered AMD GPUs support half-precision extension of OpenCL specification. Without this extension, no hardware instruction is available to convert between half-precision and single-precision. While such conversion can be performed using several bit mangling operations, it would cap the possible performance gain as well.

The penalty to the quality of the reconstruction induced by reduced precision is evaluated in Figure 7. For synthetic Shepp Logan phantom it is negligible. However, the behavior for the real-world measurements may be different, especially if projections are obtained using a camera with high dynamic range. As the optimization proposed in this subsection changes the reconstruction results, it is important to verify that the achieved quality is still satisfactory for the considered application.

5.4. Efficiency of the standard algorithm

The Figure 6 evaluates efficiency of texture engine utilization. While performance in a single-slice processing mode is close to theoretical maximum on a majority of the considered architectures, the efficiency drops significantly if multiple slices are reconstructed in parallel. The AMD cards and the cards based on the NVIDIA Kepler architecture show sub-optimal performance also in a single-slice reconstruction mode.

As was discussed in section 3.7, GPU architectures include multiple functional blocks operating independently. The performance of the GPU application is typically restricted by the slowest and/or most loaded of these blocks.

Secondly, complex algorithms require a large amount of hardware resources like registers and shared memory. Large footprint on resources may constrain parallelism and, consequently, limit an GPU ability to hide memory latencies and schedule load across all available functional units. The discussed algorithm relies on:

- Texture engine to fetch and interpolate data
- ALUs to find the ray incidence point
- Constant memory to load projection constants
- The SFU units are used for type conversions and integer multiplication on the recent NVIDIA devices. The major load is from conversion between half- and single-precision formats in 4-slice reconstruction mode. The SFUs are also used for addressing constant memory arrays and to convert a loop index to the texture-coordinate along the projection axis.

The standard algorithm has a small register footprint and all GPUs provide enough computing power to find incidence points. The performance of the texture engine, however, is sub-optimal across all architectures if multi-slice reconstruction is performed. The reason is the bad locality of the texture fetches. The AMD GPUs are also restricted by the performance of the texture cache if only a single-slice is reconstructed. On top of that, the Kepler and AMD VLIW systems have comparatively slow constant memory which also bounds the performance bellow the theoretical throughput. Finally, the low SFU performance on the Kepler GPUs restricts the reconstruction if half-float format is used to store the sinograms. More information about GPU capabilities and the relative performance of GPU components is given in Table 10.

5.5. Optimizing locality of texture fetches

The standard algorithm maps each GPU thread to a single pixel of output slice. The default mapping is linear: the thread with coordinates (x, y) in a computational grid is used to reconstruct the pixel with coordinates (x, y) in a slice. Since every thread in a wrap reconstructs consecutive pixel along x axis, a large range of sinogram bins is always accessed. Up to 16 different locations is fetched by a warp if 16-by-16 thread blocks are utilized. As it

Table 11: Queries to texture cache with standard and optimized mapping on NVIDIA GeForce Titan X (Pascal)

Slices	Approach	Queries[a]	Tex. hits[b]	L2 hits[c]	Perf.[d]
1	Standard	0.43	96.0%	89.0%	381 GU/s
	Remapped	0.39	95.5%	89.4%	376 GU/s
2	Standard	0.61	91.5%	88.6%	534 GU/s
	Remapped	0.53	93.8%	88.3%	724 GU/s

The table compares efficiency of the texture fetches using standard linear mapping scheme and the new scheme with improved locality. The measurements obtained using NVIDIA profiler for the 1- and 2-slice reconstruction modes. 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]** achieved reconstruction performance in giga-updates per second.

was discussed in the section 3.4, 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.

To improve the locality of the texture fetches, a new thread-to-pixel mapping is proposed. The thread blocks assignments are kept exactly the same as in the standard version. I.e. each block of 256 threads is responsible for an output area of 16-by-16 pixels. However, this area is further subdivided into 4-by-4 pixel squares. Within each square, the threads are mapped along Z-order curve as illustrated in Figure 9, left. Then, a group of 4 threads fetches positions in a sinogram row which are maximum 3 bins apart. And only up to 5 elements are required to perform corresponding linear interpolations. The data required for 16 threads is limited to 8 bins only. Table 11 shows the effect of remapping for the 1- and 2-slice reconstruction on NVIDIA Titan X GPU. According to Figure 6 a significant speed-up is also achieved on other architectures unless the performance is also capped by other factors.

The pseudo-code to compute the new thread indexes is given in Algorithm 2. The only required modification in Algorithm 1 is to use the updated indexes m'_t in place of ones reported by CUDA/OpenCL.

5.6. Optimizing memory bandwidth

Even though the new thread mapping gives a significant speed-up on a majority of considered architectures, the performance on Kepler and AMD

```
Input: $\vec{m}_t$ is the original mapping as reported by CUDA/OpenCL
Output: $\vec{m}_t^1$ is a new mapping proposed in section 5.5 to improve locality of the
        texture fetches. $m_p$ and $\vec{m}_t^2$ define an alternative mapping allowing also
        to reduce the load on constant memory as explained in section 5.6.
begin
    /* Each thread is responsible for one of 4 pixels laying within a small 2x2 pixel
       square which is in its own right is one of 4 squares composing the larger 4x4
       pixel block.  Here we determine the sequential number of pixel in small
       square, the sequential number of the small square in the larger pixel block,
       and the sequential number of these block.  */
    $block_n = m_t.y$
    $square_n = m_t.x\ /\ 4$
    $pixel_n = m_t.x\ \%\ 4$
    /* Converting the sequential number to $x$,$y$ coordinates.  */
    $\vec{block} = \{block_n\ \%\ 4, block_n\ /\ 4\}$
    $\vec{square} = \{square_n\ \%\ 2, square_n\ /\ 2\}$
    $\vec{pixel} = \{pixel_n\ \%\ 2, pixel_n\ /\ 2\}$
    /* Compute the actual pixel offset for the first mapping */
    $\vec{m}_t^1 = 4 * \vec{block} + 2 * \vec{square} + \vec{pixel}$
    /* Compute the projection and pixel offset for the second mapping */
    $\vec{m}_t^2 = 2 * \vec{square} + \vec{pixel}$
    $m_t^2.x \mathrel{+}= 4 * block.x$
    $m_p = block.y$
end
```

Algorithm 2: Optimizing thread mapping for the better cache locality and reduced load on constant memory

VLIW GPUs is still bound by the slow constant memory. To process a projection, GPU threads load several geometric constants to locate point of incidence as defined in equation 1. These constants can be re-used multiple times if each GPU thread would reconstruct several pixels. Since pixels are reconstructed independently, it will also increase the number of independent instructions in the execution flow and improve a scheduler ability to hide memory latencies and to issue multiple instructions per clock. There are two approaches how to adapt thread-to-pixel mapping. Either the number of threads in a computational grid is reduced proportionally or a new mapping scheme is constructed in a way that the same amount of threads is running but each thread contributes to multiple resulting pixels. The later can be achieved by processing several projections in parallel. Then, each thread is responsible for a group of pixels but iterates over a subset of all projections only. Another thread would contribute to the same group of pixels but from a different subset of projections.

Both methods perform similarly if properly optimized for the target GPU. Using the second approach, however, the dimensions of computational grid stay unchanged. Consequently, it has advantage for region of interest (ROI) and small-scale reconstructions. For this reason, we focus on this method and elaborate how it is implemented and tuned to run efficiently across platforms. To preserve good locality of texture fetches, the mapping described in previous section is adapted with small changes. The thread blocks assignments are kept the same. Each block is responsible for an output area of 16-by-16 pixels and this area is further subdivided into 4-by-4 pixel squares. In contrast to original mapping, however, 64 threads are assigned per square. Each thread is responsible to compute a contribution to the pixel value from a quarter of all available projections. Hence, each thread processes 4 pixels and each pixel is reconstructed using 4 threads. To avoid costly atomic operations, the contributions of the projection subsets are summed independently. Then, the threads are re-assigned to perform reduction in the shared memory and compute the final value of a pixel. To preserve a good spatial locality of the texture fetches, 4 neighboring projections are processed in parallel and the threads step over 4 projections at each iteration step.

There are 256 threads in a block and 64 threads are assigned to reconstruct each 4-by-4 pixel square. Therefore, 4 such squares are processed in parallel and a complete set of 16 squares requires 4 iterations. Figure 8 shows several possible sequences to serialize processing. The first mapping is sparse and results in a reduced cache hit rate as compared to the other options. Since only a single pixel coordinate has to be incremented in a pixel loop, the third option requires less registers compared to the second. While the second mapping has a better access locality within the 64-thread warps of the AMD platform, it does not affect performance in practice. On other hand, the register usage is very high in multi-slice reconstruction modes and the extra registers cause reduced occupancy or the spillage of registers into the local memory. Therefore, the third approach is preferred.

A request to multiple locations in the constant memory by a warp is serialized on NVIDIA platform. To avoid such serialization, all threads of a warp are always assigned to the same projection. The following mapping scheme is adopted. The lowest 4 bits of the thread number in a block define the mapping within a 4-by-4 pixel square. A group of 16 threads follows Z-curve as explained in the section 5.5. Next 2 bits define a square and the top 2 bits define the processed projection. Figure 9 illustrates the proposed mapping and Algorithm 2 provides the corresponding pseudo-code.

The pseudo-code for the complete approach is presented in Algorithm 3. There are two distinct processing steps. First the partial sums are computed in an 4-element array. It is declared as a local variable and both NVIDIA and AMD compilers are able to back it with registers because of the fixed size. The outer loop starts from the first projection assigned to a thread and steps over the projections which are processed in parallel. The large loop-unrolling factor requested with *pragma* preprocessor directive has a positive impact on performance, especially on Kepler architecture. At each iteration constants are loaded and inner loop is executed to process 4 pixels the thread is responsible for. After completion of all projections, the reduction loop is executed. The partial sums are written into shared memory and reduction is performed. To avoid non-coalesced global memory writes, first all results are stored in a shared memory buffer $\tilde{r}^S$ and, then, written in the coalesced manner. The synchronization is needed when switching different mapping modes. Since each reduction is performed by a single warp only, it is sufficient to prevent compiler from reordering read and write operations in-between of reduction steps using *fence* operation. Alternatively, the *shuffle* operation may be utilized to perform reduction on Kepler and newer NVIDIA architectures. Then, neither *fence* nor *if*-condition are required. The reduction loop using the *shuffle* instruction is shown in Algorithm 4.

On GTX295 using CUDA6, there are a few glitches significantly affecting performance. The *fence* instruction prevents unrolling of the reduction loop. Consequently, the array with partial sums is referenced indirectly using the loop index. This forces the compiler to allocate array in the local memory instead of using registers and causes enormous penalty to the performance. Therefore, a standard *__syncthreads* is used instead. The loop is also not unrolled if the inner reduction loop is implemented directly as written in Algorithm 3. The following formulation causes no issues:

```
for(j = 0; j < 2; j++) {
    i = 2 >> j;
    ...
}
```

The GPU constant memory is optimized with assumption that always the same constants are accessed by all threads of a computational grid. Since the new algorithm goes over several projections in parallel, this assumption is not valid any more. While the proposed mapping avoids major penalty due to warp serialization, slow constant memory is still a bottleneck on older

AMD devices. To avoid performance penalty, faster and larger shared memory is used instead in this case. The projection constants are initially stored in global GPU memory and, then, are cached in shared memory. The Algorithm 5 contains alternative implementation of the accumulation step for Algorithm 3. Shared memory is additionally configured to store constants for up to 256 projections. In fact, the same shared memory buffer may be used in the both steps of algorithm, first for caching constants and later for a data exchange while performing reduction. An outer loop iterating over blocks of 256 projections is introduced. At each iteration, the threads of a block are, first, used to read the constants from global memory and fill the cache. To allow 64-bit loads, we use a *float2* variable to store values of both trigonometric functions. After synchronization, the inner projection loop is started to compute partial sums. The inner loop is implemented as in Algorithm 3 with only difference that constants are loaded from shared memory. This method, however, cannot be used across all platforms. While majority of NVIDIA GPUs showed similar performance for both implementations, Kepler-based GPUs perform better if constant memory is utilized.

5.7. Optimizing occupancy

Similarly to the standard algorithm, the optimized version can be easily adapted to process 2- and 4-slices in parallel. Only accumulators and intermediate buffers have to be declared with the appropriate vector type. However, the usage of hardware resources grows significantly if multiple slices are processed in parallel. In a 4-slice mode, 16 registers (32-bit each) are required only to accumulate the partial sums. The large register footprint reduces occupancy and may result in a sub-optimal performance unless treated properly.

The register allocation is completely out of developer control on AMD platform. NVIDIA allows to target the desired number of blocks executed by each SM in parallel. It is done using *__launch_bounds__* keyword. The CUDA optimizer, then, changes the code generation algorithm to meet the target. It prevents data pre-fetching and also may result in an increased computational load and/or in a more intensive usage of L1 caches as a part of local variables is offloaded to local memory. On Fermi and Kepler architectures, 64 KB of on-chip memory is split between L1-cache and shared memory according to the user-specified configuration. By default 48 KB is assigned to shared memory and only 16 KB is left for L1 cache. If the shared memory consumption is low enough, it is possible to re-balance this ratio and achieve a high occupancy on one hand and ensure that there is enough L1 cache to back all the required local memory on the other.

The Table 12 summarizes resource consumption, theoretical occupancy, and achieved performance on the NVIDIA GTX Titan with and without resource restriction. The results show that improved occupancy may bring a considerable speed-up also if significant number of variables has to be offloaded to local memory, provided it is backed by L1 cache. Without restriction the generated code requires 38 registers if 2-slice reconstruction mode is enabled. This limits the number of resident threads to 1724 or 6 blocks and results in 75% occupancy. The performance is improved by 15% if CUDA compiler is instructed to allow execution of 8 blocks, i.e. running at full occupancy. To fulfill this requirement the compiler puts 6 variables in the local memory. However, 16 KB of L1 cache is not enough to assure backing of the required local memory for 8 resident blocks. On other hand, only 4 KB of shared memory is required per block for temporary buffers or 32 KB for all 8 blocks. Therefore, the ratio between L1 cache and shared memory is shifted to allow 32 KB of L1 cache. This is done using *cudaFuncSetCacheConfig* command with *cudaFuncCachePrefer* argument to specify preference for L1 cache. The recommended restrictions for other architectures are summarized in Table 13

Both shared and constant memories are comparatively slow on Kepler with respect to the performance of the texture engine, see Table 10. Furthermore, 64-bit access is required to fully utilize the available bandwidth of shared memory. This is given in the multi-slice reconstruction mode. However, 64-bit operation should be also enabled in CUDA using *cudaDeviceSetSharedMemConfig* command with *cudaSharedMemBankSizeEightByte* argument. The constant memory also performs better if 64- or 128-bit wide access is performed. A speed-up is achieved if all projection constants are grouped together and stored as a single *float4* vector. Even if only 3 components of the vector are actually required (i.e. one quarter of bandwidth is actually wasted), the performance is considerably better.

5.8. Summary

We have introduced a new cache-aware algorithm which is able to reconstruct up to 4 slices in parallel. Several modifications are proposed to improve performance on specific GPU architectures. The optimal configuration and the corresponding performance are summarized in Table 13. The achieved efficiency is further analyzed on Figure 6. For a single slice reconstruction mode, the performance is above 90% of the theoretical maximum across all considered platforms. Depending on the architecture, it corresponds to a speed-up of up to 90% as compared to the original algorithm. Using the multi-slice reconstruction and half-float data representation, it is possible to quadruple performance on Fermi and the latest AMD and NVIDIA architectures. Efficiency of about 80-90% is achieved if the optimized reconstruction kernel is utilized. The efficiency of Kepler GPUs is restricted due to comparatively slow on-chip memory and low performance of SFU units which are used to perform type mangling operations. Nevertheless, 2 to 3 times speed-up over the standard single-slice algorithm is achieved due to the proposed optimizations.

6. Alternative algorithm based on ALUs

While it is possible to reach a very high reconstruction speed by processing multiple slices in parallel, this option is not available on all GPUs. Furthermore, the ability to re-combine slices for parallel reconstruction may be limited due to architecture of data processing pipeline or by the latency requirements. According to specifications, the majority of GPUs are able to perform over 32 floating-point operations during a single texture fetch, see Table 10. Only 9 floating-point operations are required to perform a single update of back projection algorithm [53]. Therefore, an alternative implementation using the algebraic units to perform interpolation may outperform the texture-based kernel by 3-times if executed on a single slice. The challenge is to feed the data into the floating-point units at the required rate. The L1 cache integrated in SM is small with low associativity and, consequently, is susceptible of cache poisoning. As result, the loads from global memory limit performance severely. In this section we present a back projection algorithm based on ALU to perform interpolation and using shared-memory as an explicit cache. First, we will explain the concept and present a base version of the algorithm. Then, we build a simplified performance model and analyze that limits the performance on each of the hardware platforms. Finally, multiple adjustments are discussed to slightly shift balance between memory operations and different types of computations and to address the capabilities of a specific architecture better.

6.1. The Concept

The proposed approach is illustrated on Figure 10. To avoid the penalties associated with global memory loads, shared memory is used to cache all bins required for reconstruction by a block of threads. To reserve a large enough buffer for the cache, it is necessary to find an upper bound of bins (b) required to reconstruct a rectangular block of pixels (S) with dimensions n by m. It is defined as

$$b_p = \max_{(x,y)\in S} r_p(x,y) - \min_{(x,y)\in S} r_p(x,y)$$

where $r_p(x,y)$ is the incident offset in a projection row which is computed as defined in equation 1. If $(x_0,\, y_0)$ is the coordinates of maximum of r_p and $(x_1,\, y_1)$ - of minimum, the equation can be reformulated as

$$b_p = (x_0 - x_1)\cdot\cos(p\alpha) - (y_0 - y_1)\cdot\sin(p\alpha)$$

or

$$b_p = \sqrt{(x_0-x_1)^2 + (y_0-y_1)^2}\cdot\cos(\alpha+\beta)$$

where β is some angle. Then, b_p can be estimated as:

$$b_p \leq \sqrt{(n)^2 + (m)^2}\cdot\cos(\alpha+\beta)$$

It is independent of processed projection and is minimal if the area S is square. In this case the value of b does not exceed $n\cdot\sqrt{2}$. For practical purposes we assume that $\frac{3}{2}n$ bins are required per projection to reconstruct a full pixel square with side n. To perform caching, it is necessary to find the minimal required bin (h_m) for each projection. Then, the reconstruction is performed in two stages. First, the required bins are 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 in several groups. Each group is responsible to cache bins for a single projection. A subset of a sinogram row consisting of $\frac{3}{2}n$ bins is extracted starting at an offset equal to the h_m. Based on a thread index in a group, the offset in a sinogram is computed and the corresponding bin is cached in a shared memory array. If necessary, a few bins with a stride equal to a number of threads in a group are cached by the same thread. 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. As usual, 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 by each thread and an offset from the h_m value is found. Typically the offset is not 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 shared memory or both neighboring values are loaded and the linear interpolation is performed to compute the impact of a projection.

Both steps depend on the h_m to perform caching and to locate the required value in the cache. This operation is costly and would add significantly to computation balance if executed by each thread and for each projection. To reduce amount of required operations, the h_m values are cached in the shared memory during the first stage of algorithm and, then, re-used in the second. Furthermore, the minimal bin is always accessed while reconstructing one of the corners of the pixel square. The actual corner is only depending on the projection angle and is the same across all squares of the reconstructed slice. Therefore, a single value is required for each projection to compute minimal bin. This value (c_m) can be defined as the difference between the position accessed to compute a top-left pixel of a square and the minimal position accessed across this square. Then, it is computed as:

$$c_m = n \cdot \max(0, \cos(\alpha_p), -\sin(\alpha_p), \cos(\alpha_p) - \sin(\alpha_p))$$

Using c_m, the minimal required bin (h_m) is computed as: $h_m = floor(h_b + c_m)$, where h_b is the bin accessed by the first thread of a block. It is computed based on a index of a thread block in the computational grid as described in Table 7. The c_m is computed during the initialization stage and is stored along with other projection constants in the GPU constant memory.

Multiple auxiliary operations are required to perform reconstruction. The sinogram values are fetched from the cache, interpolated, and summed up. On top of that, the h_m is computed for each projection, the selected parts of sinogram are cached, and the corresponding positions in the projection cache are determined for each pixel. These operations add an additional load on GPU and significantly reduce the performance. While it is impossible to eliminate the auxiliary operations entirely, there are two major ways to scale down their proportion. Either several slices are reconstructed in parallel or a larger pixel area is assigned to a thread block for reconstruction. First option allows to reduce proportion of computations needed to determine which data is cached for each projection and to find the required offset in the shared memory array. Since the reconstruction is not bound by a performance of the texture engine any more, there is no restriction on a number of slices processed in parallel. It is possible to reconstruct 4 or more slices together provided there is enough hardware resources to handle the data. Proportionally less data has to be cached if a larger area is assigned to a thread block. Consequently, the load on global and shared memories is reduced. This is achieved either by increasing a number of threads in a block or by assigning multiple output pixels to each thread. Since the constants can be stored in the registers and re-used to process multiple pixels, the load on constant memory is reduced in the last case as well T So, the first option cuts down the amount of computations significantly. The second method cuts down computations to lesser extent, but also reduces an utilization of shared memory slightly. Both ways, however, increase the use of hardware resources significantly. More shared memory and more registers are required. The optimal compromise between these options has to be found for each targeted platform. Furthermore, there are multiple ways to implement the described operations. Each variant will put more load on one GPU subsystem or another. The additional shared-memory caches can be utilized to shift the balance between computations and memory operations. In the next subsection we present a base implementation and will target the specific architectures across the rest of the section.

6.2. Base Implementation

Processing only a single pixel per thread is sub-optimal across all targeted platforms. The optimal load is between 4 and 16 pixels depending on the available hardware resources. Since square areas are most efficient to cache, we target areas of either 32-by-32 or 64-by-64 pixels per a thread block. While intermediate sizes can be used as well, for power of two sizes it is easy to design thread mappings suitable for both caching and accumulation stages of the reconstruction process. For sake of simplicity, in Algorithm 6 we present a simple version processing 4 pixels per a thread. A block of 256 threads is used to reconstruct a square of 32-by-32 pixels. The maximum number of bins accessed per projection, then, is equal to $32 \cdot \sqrt{2}$ or 46. If the linear interpolation is used, up to 47 elements in a sinogram array are actually accessed for each projection. Therefore, 16 threads cache all required values in 3 iterations. To avoid conditionals all 48 values are always cached. This ratio keeps if 64-by-64 area is reconstructed. The 96 values has to be cached. The number of projections processed in parallel is limited by the available shared memory and the size of a single projection row in the cache. A group of 16 projections may be cached at once if only a single slice is reconstructed. For 4-slice mode or if a 64-by-64 area is reconstructed, only 8 projections are typically processed in parallel. Reducing this number further may have negative impact on the performance as many threads would need to wait at the synchronization point reducing the effective occupancy.

As was already explained, the h_m is computed during the caching stage and also stored in shared memory. Instead of repeated computation, the value is loaded from shared memory during the reconstruction stage. To find the required offset in the cache, a difference between the position in a sinogram row (h) and h_m is computed. The equation for h includes the projection-corrected position of the rotational axis (c_a) which is constant for all pixels. It can be integrated into the h_m already during the caching step of the algorithm. So the value of $c_a - h_m$ is stored in shared memory instead of h_m. Then, only the pixel-dependent part of the projection equation is computed inside of the main loop.

Since no interpolation is required while the data is read from global memory, it is possible to access the sinograms directly rather than using texture fetches. The loads are always coalesced and thread blocks read each value only once. 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 will be executed by SM at each clock. On other hand, texture loads are performed using the specialized units on all architectures. Therefore, it is possible to load data from global and shared memory simultaneously if global memory is accessed using the texture engine. It makes the texture engine a preferred option to get the data in the shared memory cache. To avoid unnecessary interpolations, the texture engine is configured to use nearest neighbor interpolation.

6.3. Optimizing the thread mapping to avoid shared memory bank conflicts

Like for the texture-based reconstruction kernel, the thread-to-pixel mapping is important to achieve a good performance. The main goal is to reduce shared memory transactions and avoid shared memory bank conflicts during the both stages of reconstruction. On all architectures, the warps need to avoid accessing multiple rows of the same shared memory bank in a single instruction. While the warp consists of 64 threads on the AMD platform, maximum 32 shared memory banks are supported on the reviewed GPUs. To prevent bank conflicts, it is only necessary to avoid accessing the same bank across a group of 32 threads [47]. Therefore, there is no need to tackle the larger warp size on AMD while discussing the shared memory access. Furthermore, there are several architecture specific restrictions. The Fermi and AMD devices are not capable to handle 128-bit data efficiently [34, 47]. Using 64-bit wide operations is extremely important on the Kepler architecture to utilize the full performance of shared memory. Only half of the bandwidth is available if 32-bit access is performed. While not as significant as on the Kepler architecture, 64-bit loads are about 20% faster on AMD Cypress and Tahiti [47].

No changes are required to benefit from the 64-bit shared memory in the multi-slice reconstruction modes. A 64-bit access can be easily facilitated in the caching step of the algorithm also if a single-slice reconstruction is performed. Each thread is made responsible to cache 2 bins per iteration. First, 2 texture fetches are performed to extract values of the neighboring bins. Then, both values are assembled in a 64-bit *float2* vector and are written into the shared memory using a single operation, see Algorithm 7. This approach, however, reduces the locality of texture fetches. Since h_m may have an odd value, switching to *float2* textures is not an option. However the load on the texture engine is quite low and in contrast to shared memory has little impact on overall reconstruction speed. This optimization is relevant on NVIDIA Kepler and both older AMD GPUs.

Only the half of the available shared memory banks are utilized on the NVIDIA Fermi and all AMD GPUs if 128-bit data is accessed. To circumvent the problem, it is possible to split the *float4* vectors in two parts, store them in the two buffers in shared memory separately, and re-combine back before performing interpolation, see Algorithm 8.

In the first stage of algorithm, the number of threads assigned to cache each projection is adjusted to optimize access to the shared memory. If a large 64-by-64 area is reconstructed, a full warp of 32 threads can be assigned for each projection row avoiding any possible bank conflicts. Unfortunately, it is not completely optimal on the Kepler architecture as, then, it is impossible to re-combine two bins into a single 64-bit wide write as explained above. It is also not possible to assign 32 threads per row for a smaller 32-by-32 area because only 48 bins has to be cached per projection in this case. And it is a bad idea to keep the half of threads idling. Therefore, several projection rows are processed by each warp in the described cases. This potentially may cause bank conflicts. If only a single slice is reconstructed, however, the banks are shifted from one projection row to another as illustrated on Figure 11. The caching is performed without bank conflicts if either 16 threads are assigned per projection row on the platforms with 32-bit shared memory or 8/16 threads are used on the Kepler devices. Only 8 threads are used to allow bin re-combination if a small area is reconstructed. 16 threads per projection are optimal on all platforms if multiple slices are reconstructed. The 64-bit banks storing *float2*-sinogram are shifted across projection rows exactly the same way as 32-bit banks do if a simple *float*-sinogram is reconstructed. And on platforms with 32-bit shared memory it is enough to prevent bank conflicts within a group of 16 threads while dealing with 64-/128-bit data. The optimal settings for each reconstruction mode are summarized in Table 14.

According to the documentation it does not matter how the threads of a half warp are accessing shared memory. In practice, however, we found that on recent NVIDIA devices the performance of 64- and 128-bit loads is slightly improved if only 1-2 different memory locations are accessed by groups of 4 consecutive threads. The locality of shared memory loads is improved if each half-warp is mapped to a square consisting of 4-by-4 pixels and the threads are arranged along Z-order curve similarly to the texture fetches. All 256 threads of a block are mapped to 16 such squares. For the reasons explained in section 5.6, the squares are arranged linearly along x-axis. Two rows of 4x4 squares are processed in parallel if a small 32-by-32 area is reconstructed. A single row is covered for the bigger area or if only 128 threads are assigned per a block. 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 global memory once the processing of all projections is completed. The complete mapping scheme is illustrated on Figure 12. The performance of NVIDIA Titan X is increased by 3% if the described mapping is utilized.

6.4. Advanced Caching Mode

For linear interpolation two neighboring bins are always loaded, but it is impossible to perform 64-bit load due to the alignment requirements. Consequently, only a half of the available bandwidth is used on the Kepler architecture in the single-slice processing mode. To allow 64-bit access, both values required to perform linear interpolation are stored as *float2* vector in the corresponding bin of the cache. The size of cache is doubled, but also the achieved bandwidth is increased by factor of two on the Kepler platform and is considerably improved on the AMD devices which are optimized for 64-bit loads. The required amount of shared memory is still adequately low and does not limit occupancy if the single-slice reconstruction is performed.

Furthermore, one floating-point operation is eliminated in the interpolation step of algorithm if the second component of cached vector actually stores the difference between the values of consecutive bins in a sinogram as shown on Figure 13.

The caching procedure is modified as shown in Algorithm 9. To reduce required inter-thread communication, each thread caches several consecutive bins. The communication is, then, only required to compute the second part of the last bin which is assigned to a thread. The shuffle instruction is used on Kepler and the newer NVIDIA architectures. A read from shared memory is performed on the NVIDIA Fermi and all AMD GPUs after the *fence*-style synchronization. In this case the shared memory cache is also padded by one extra column to allow an unconditional read by the last thread in a group assigned to a projection row.

In case of a 32-by-32 pixel area, 16 threads per projection row are used on all platforms independent of the width of a shared memory bank. The banks are shifted between projection rows on the 64-bit platforms as explained in section 6.3. And for 32-bit architectures it is enough to avoid bank conflicts within a half-warp only. Furthermore, there is also no bank conflicts between the threads of a half-warp as the stride is not a multiple of 4, see illustration in Figure 14. A full warp is used per row if a thread block is assigned to process larger 64-by-64 pixel area. The same number of iterations is, then, required to process the complete projection row and, consequently, shared memory is accessed with the same stride without bank conflicts.

6.5. Modeling

The proposed method is relatively complex and utilizes multiple GPU subsystems. There are many ways to tune the proposed algorithm to address the capabilities of a targeted architecture better. It is important to understand the limiting factors in each case. Here, we try to build a simplified performance model. First, we identify several distinct operations required to perform back projection:

1. The projection constants are loaded from memory. And the minimal required bin is computed to decide which data has to be cached.
2. The sinogram subsets are fetched from the texture and cached in shared memory.
3. For each reconstructed pixel, the corresponding position in a sinogram is determined.

4. The offset in the shared memory array is computed.
5. Depending on the requested interpolation type, one or two values are fetched and the contribution of a projection is added to the accumulator.

These operations rely on several hardware components:

- **Constant memory** is used to retrieve projection constants.
- **Texture Engine** is used to retrieve the sinogram values.
- **Shared memory** is used while caching the data and retrieving the cached values to perform interpolation.
- **ALUs** are used for general-purpose computations, particularly to perform projection and interpolation.
- **SFUs** are used on Kepler, Maxwell, and Pascal architectures to perform rounding operations, to convert data between floating point and integer representation, and to perform bit-shifts. These instructions are used to compute offsets in the cache and to perform interpolations. While the bit-shifts are not used directly in pseudo-code, they are implicitly utilized to resolve addresses in the constant and shared memory arrays.

Each of these components may limit the performance if its resource is exceeded. Furthermore, there is also a limit on a number of instructions which SM is able to schedule per a clock cycle. Particularly, the warp scheduler on Fermi is limited to a single instruction per clock. If a memory instruction is launched, the half of ALUs are kept idle. To estimate the performance we assess the number of required operations according to the presented pseudo-code. We assume that the performance is either capped by the slowest of the components or by a total number of instructions. It is a very rough estimate. The developed kernels are resource intensive and are executed at a significantly reduced occupancy. It is difficult to predict how the compiler will generate the code to manage the available resources. Furthermore, some variables are moved to the slower local memory. The local memory is backed by L1 cache which shares the hardware with shared memory on the Fermi and Kepler based GPUs. Consequently, the operations with such variables are not only increasing latency, but also may penalize the shared memory performance. Nevertheless, the obtained estimates allow us to choose the required optimization strategy for each architecture.

Instructions required to perform a single update on a pixel value are summarized bellow for the reconstruction using the linear interpolation. Rounding/type-conversions (TC) and bit-shift (BS) operations are counted separately because they are scheduled differently on Maxwell/Pascal and Kepler GPUs. For each operation the normalization coefficient, i.e. the number of updates performed per the specified number of instructions, is indicated. Fused-Multiply-Add (FMA) is counted as a single instruction.

1. Computing and caching of h_m (per $n_t * n_q * n_v$ updates)
   - **Constant Memory** (128-bit): s_t
   - **Shared Memory** (32-bit): s_t (because a full warp is executed anyway)
   - **FP**: $4s_t$ (to compute h_b and h_m^S)
   - **TC**: s_t (rounding)
   - **BS**: $2s_t$ (resolving addresses in constant and shared-memory arrays)
2. Caching (per $n_t * n_q * n_v$ updates)
   - **Texture Fetches**: $\frac{3}{2}\sqrt{n_t * n_q}$
   - **Shared Memory** (type-dependent, but always 64-bit in Advanced Caching Mode): $\frac{3}{2}\sqrt{n_t * n_q}$
   - **FP**: $4s_t$ (3 if Advanced Caching Mode is not used)
   - **TC**: s_t (integer to float conversion of projection number to perform texture fetch)
   - **BS**: $2s_t$ (resolving addresses in the shared-memory array)
3. Setting inner-projection loop and evaluating required position in sinogram (per $n_q * n_v$ updates):
   - **Constant Memory** (64-bit): 1 (only cosine and sine of the angle are required here)
   - **Shared Memory** (32-bit): $0.25 - 1$ (offsets for 4 projections can be loaded at once using a single 128-bit load if the loop is unrolled)
   - **TC**: $2 - 3$ (computing h, updating loop index unless unrolled)
   - **BS**: $1 - 3$ (resolving addresses in the constant array and also in both shared memory caches unless the loop is fully unrolled)

4. Computing an offset in the cache and the coefficient for linear interpolation (per n_v):
    - **FP**: 2 (update to the next offset; computation of interpolation coefficient unless nearest neighbor mode is selected)
    - **TC**: 2 (rounding and float-to-integer type conversion; only a single operation is required if nearest neighbor interpolation is performed)
    - **BS**: 1 (resolving the address in the shared memory array)
5. Linear Interpolation (for each update):
    - **Shared Memory** (type dependent): 1 (2 if Advanced Caching Mode is not used)
    - **FP**: 2 (interpolation and update; 3 operations if Advanced Caching Mode is not used and only 1 if nearest neighbor interpolation is performed)

Further, a single-slice reconstruction ($n_v = 1$) using advanced caching mode is evaluated. Blocks of 256 threads ($n_t = 256$) are assigned to process a 32-by-32 pixel square ($n_q = 4$). For sake of simplicity we assume that 16 threads are used to cache a single projection row and that the inner projection loop is fully unrolled. We skip the texture fetches as load is very low and certainly is not a limiting factor here. Then, the following number of operations is estimated per a single update:

- **Constant Memory**: 2.3 bytes (0.3 instructions)
- **Shared Memory**: 8.7 bytes (1.1 instructions)
- **FP**: 4.6 (counting FMA as a single operation)
- **TC**: 2.0
- **BS**: 1.3
- **Instructions**: 8.2

To verify these estimates, the prototype implementation was executed under CUDA profiler. The number of estimated and measured operations is compared in Table 15. There is a difference, but the error is within 10%.

Table 16 evaluates the maximum performance according to each execution unit. The throughput is taken from Table 10 and the load is computed according to the list above using the following assumptions which are explained in the section 3.7. NVIDIA Maxwell and Pascal are not restricted to the SFU to perform bit-shifts, but are able to use also ALU units. On this devices we do not include the integer multiplications in the SFU balance. On NVIDIA Kepler we do. SFUs are either not available or not used on AMD GCN and NVIDIA Fermi. So, all types of operations are counted together in the ALU balance.

As can be seen, the performance bottleneck is architecture dependent. The AMD VLIW and NVIDIA Kepler GPUs are bound by ability to perform rounding operations and to convert variables between integer and floatint-point representation. While not limiting performance in the modeled configuration, this still sets a quite low threshold on Maxwell and Pascal GPUs. However, the main limiting factor on these architectures is the shared memory bandwidth. The Fermi GPU is only capable to dispatch a single instruction per warp and, consequently, bound by the instruction throughput. Finally, AMD GCN based devices are restricted by the performance of algebraic units.

6.6. Rounding Using Floating-Point Arithmetic

The Kepler performance is severely limited because due to rounding and type conversion operations. The reason is the slow performance of SFU units on the Kepler platform. Total 3 SFU operations are required to compute offset in shared memory and to perform linear interpolation.

`float` $h_f = floor(h);$
`int` $h_i = (int)h_f;$
`float` $d = d^S[h_i];$

Each of the listed instructions uses SFU. The first instruction performs rounding and the second converts floating-point number to integer. The last operation involves a bit-shift to resolve the address of an array element. The array index is multiplied by the size of a data type, but the bit shift is actually performed in place of multiplication because the data size is always power of two. Instead, it is possible to perform multiplication using the floating point numbers and operate with pointers directly, like:

`float` $h_f = floor(h);$
`int` $h_i = (int)(4.f * h_f);$
`float` $d = *(float*)((void*)d^S + h_i);$

Then, one of the 3 SFU instructions is replaced with 2 floating-point operations. However, it is possible to eliminate the SFU instructions entirely. Since the offsets are always small positive numbers, rounding and type-conversion operations can be implemented using the floating-point arithmetic only. IEEE754 specification defines the format of a single-precision floating point number [54]. It is illustrated on Figure 15. For positive numbers, the representation is defined as:

$$f = 2^{e-127} \cdot (1 + \sum f_i \cdot 2^{i-23}) \tag{2}$$

Consequently all fractional components are eliminated if 2^{23} is added to a number.

$$f + 2^{23} = 2^{23} \cdot (1 + \sum f_i \cdot 2^{i-23}) \tag{3}$$

The rounded number is obtained if e^{23} is subtracted back afterwards. To compute *floor()* it is necessary to subtract 0.5 before these operations. I.e. the following implementation is suggested:

```
float $e_{23} = exp2(23.f)$;
float $h' = h - 0.499999f$;
float $h_{tmp} = h' + e_{23}$;
float $h_f = h_{tmp} - e_{23}$;
```

The proposed method replaces a single SFU-based rounding instruction with 3 floating-point operations. The e_{23} constant is computed only once in the beginning of a kernel and does not add much to the computation balance. It is further possible to make a float-to-integer conversion using a simple integer subtraction which is performed by ALU. The small integer numbers are fully encoded by the fraction portion of an IEEE 754 number. There are still some significant bits representing exponent, but they can be easily eliminated as illustrated on Figure 15.

```
int $h_i$ = __float_as_int($h_{tmp}$) - 0x4B000001;
```

The *__float_as_int* is a simple cast (re-interpretation) of a floating point number as an integer. Using the pointer arithmetic notation, it is equivalent to $*(int*)\&h_{tmp}$. Still there is an index computation left. It is often reasonable to keep some load on SFUs as well. If indexing is left unchanged, the $d[\_\_float\_as\_int(h_{tmp} - 0x4B000001)]$ is replaced with a single *iSCADD* operation combining multiplication and integer subtraction. It is executed on SFU in a single clock cycle. Consequently, 2 SFU instructions are replaced with 3 floating point operations and a single SFU instruction is left. The other option is to eliminate SFU instructions entirely. It is possible with

```
$h_{tmp}$ = 4 * $h_{tmp}$ - (4 - 1) * $e_{23}$;
void *addr = (void*)$d^S$ + __float_as_int($h_{tmp}$);
float $d$ = *(float*)(addr);
```

In this case 3 SFU instructions are replaced with 5 floating-point operations. The method to use depends on the expected operation balance. It can be estimated using the performance model which was proposed in section 6.5. Either way the result is exact and there is no penalty to quality.

To perform nearest-neighbor interpolation, 2 SFU instructions are required on Kepler. One instruction can be easily replaced with floating-point operation by performing multiplication before type conversion as explained above. Otherwise, the SFU instructions are completely replaced with 3 floating-point operations.

The performance along with a number of instructions issued per update is shown in Table 17 for NVIDIA GTX Titan. The speed-up of 20% is achieved if rounding is implemented using floating point instructions, but index computation is left on SFU. The complete elimination of SFU instructions puts unnecessary load on ALUs and keeps SFU units idle. This method has a little impact on the VLIW architecture. While the performance is limited by the throughput of integer instructions, the difference between performance of floating-point and special units is not as high as on Kepler. Consequently, the performance is limited approximately at the same level.

6.7. Half-float cache

Like in the texture-based reconstruction algorithm, the half-float representation may be used to speed-up reconstruction. While the texture engine is not a performance-limiting factor here, the bottleneck in shared memory is lifted on the Maxwell and Pascal architectures if half-float values are cached in shared memory. The values are converted to a single-precision format just before computing interpolation. Since the texture units are always used in the nearest neighbor mode, it possible to use the half-float data representation also to speed-up reconstruction performing linear interpolations. About 10% performance increase is measured on Pascal and Maxwell if 4 slices are reconstructed in parallel and the linear interpolation is performed. High load on SFU units to convert between half and floating-point representation prevents larger speed-ups. It is also the reason why no performance improvements are reported on other platforms. On professional series of Tesla cards with Pascal architecture it could be possible to achieve higher performance by keeping the computations in half-precision all way through the end. The results, however, are expected to suffer from additional performance degradation. In any case this is not feasible on Titan X because of significantly lower throughput of high-float arithmetic.

6.8. Additional Caches

The Fermi performance is limited by the throughput of ALU units and also by a number of instructions it is able to dispatch per clock cycle. Using the advanced caching mode, the interpolation footprint is reduced by a single instruction. Advanced caching is used across multiple platforms in the single-slice reconstruction mode. On Fermi, however, it also improves performance if multi slices are reconstructed in parallel. The vectors fetched from the texture engine are split into the components and are cached using 2 or 4 independent caches. Furthermore, there is an additional option to slightly reduce the number of operations. Two *FMA* instructions are required to find the required offset in the cache.

$$h = h_m^S[p] + f_g'.x * c_c^C[p] - f_g'.y * c_s^C[p]$$

This computation is performed once per so many pixels the thread is responsible for. If 16 pixels are assigned to each thread, the impact is negligible in the overall computational balance. Fermi is, however, limited by amount of available registers and unlike newer architectures is restricted to process only 4 pixels per a thread. Therefore, it is relevant to reduce a number of instructions required to compute h. When h_m^S value is cached, only a single thread in every 16 is actually used to perform the caching. Instead the cached value may include the x component as well and utilize all threads with a minimal extra load. I.e. the following value is cached in shared memory instead of h_m^S:

$$h_x^S[m_p][m_t.x] = c_a^C[p] + f_g.x * c_c^C[p] - h_m$$

Then, 32 values are cached per projection row, but only one *FMA* is used to compute the offset:

$$h = h_x^S[p][m_t'.x] - f_g'.y * c_s^C[p]$$

The amount of required shared memory is significantly increased, but there is no additional memory traffic. A warp is either loading the same value which is broadcasted from a single shared memory bank or up to 32 values are loaded and all banks are utilized. Furthermore, the cosine of a projection angle is not loaded any more from the constant memory. Extra instructions, however, are dispatched unless special care is taken. As was mentioned in section 6.5, the 64- or 128-bit loads are performed to load h_m^S if the projection loop is unrolled. This is possible because the values for consecutive projections are stored next to each other. Technically it is possible to organize the new cache to keep such arrangement, but there is a better option which is independent of loop-unrolling. The threads of the block are assigned to process a 16-by-16 pixel square at each iteration instead of the mapping proposed in Figure 12. In section 5.6 the linear mapping scheme is reasoned by ability to maintain only a single index because, then, each thread needs to increment an y-coordinate only. This is given using the new caching scheme as the x component is already included in the value loaded from the cache. Each thread processes 4 pixels with coordinates (x, y), $(x+16, y)$, $(x, y+16)$, and $(x+16, y+16)$. It loads $h_x^S[p_i][x]$ and $h_x^S[p_i][x+16]$ using a single 64-bit instruction and only need to increment the y-coordinate. The utilization of the shared memory bandwidth is increased as each thread needs to load 64 bits per projection instead of 32. But the total memory bandwidth is still exactly the same as in the base implementation due to reduced requests to constant memory. About 5% speed-up is achieved on the NVIDIA Fermi and AMD Tahiti architectures.

Few other values can be cached to slightly shift the balance of operations. In some cases, it is beneficial to cache also trigonometric constants in shared memory. This is slightly improves the performance across NVIDIA architectures. The h_m value is normally computed multiple times by all threads responsible to cache a specific projection row. The extra load is not very high, but can be avoided for a price of several additional registers required to introduce a third stage in the reconstruction process. At first, threads of a block are assigned to compute h_m values for 256 projections and cache it in shared memory. Then, the values are just loaded at each iteration. It was found useful on the AMD VLIW architecture. Vice-versa the caching of h_m can be disabled altogether on the systems with fast ALUs, but slow shared memory. The suggested cache settings are summarized in Table 18.

6.9. Managing Occupancy

The number of pixels processed per block and per thread is one of the main parameters affecting the performance. The optimal configuration depends on the available GPU resources, but also on the size of the reconstructed image. The number of executed blocks could be insufficient to load GPU evenly if large pixel blocks are used in conjunction with a small image. However, the texture-based approach is expected to perform better for small images in any case. To summarize the performance and optimal configuration we assume that the sufficiently large image is reconstructed and focus on the hardware capabilities only. Depending on the available GPU resources 4, 8, or 16 pixels are assigned per thread. In the last case each thread block is responsible to reconstruct an area of 64x64 pixels. Otherwise, only 32x32 pixels are processed. The block of 128 threads is used to allow processing of 8 pixels per thread.

If a number of concurrently processed slices is given, there are still multiple factors affecting the performance. The optimal implementation of the proposed algorithm should ensure that:

- The full occupancy is achieved to hide latencies efficiently.
- A large reconstruction area is assigned to each thread block to reduce amount of caching operations per reconstructed pixel.
- As many pixels as possible are assigned to each GPU thread. It allows to reduce a proportion of the auxiliary operations required to compute offsets in the cache and also ensures that a large amount of independent instructions is in execution flow as required by the architectures relaying on the instruction level parallelism (ILP).
- The number of threads assigned to cache each projection row is in accordance with the requirements specified in Table 14. Then, no shared memory bank conflicts occur and the shared memory writes are executed optimally.
- The number of projections cached at each iteration step is enough to utilize all threads in the block. Otherwise, the threads idling at the synchronization point reduce the efficiently achieved occupation.
- The projection loop is completely unrolled to provide additional ILP parallelism and ensure that multiple 32-bit memory operations can be combined into a single 64-/128-bit instruction.
- The generated code is able to issue multiple load operations in a streaming fashion as explained in section 3.6. It allows to reduce penalty inflicted by the memory access latencies if other mechanism fail to hide them entirely.

- All appropriate optimizations discussed through this section are implemented.

Due to hardware limitations it is impossible to achieve all these goals simultaneously. The number of required registers is steeply increased with a number of pixels assigned per thread and restricts the achieved occupancy. Using the multi-slice reconstruction mode, either a high occupancy or a high number of pixels per thread is possible to achieve. The amount of available shared memory restricts how many projections could be cached at the desired occupancy level. If this restriction is low, the high effective occupancy is still possible to achieve in the caching stage if more threads are used to cache each projection or smaller 128-thread blocks are in use. The first option is only available if 16 pixels are reconstructed per thread. Consequently, a high number of registers is required in both cases. Furthermore, the number of threads assigned per projection is in turn restricted if shared memory is optimized for 64-bit writes. Most of the proposed optimizations increase the usage of registers or/and shared memory. The use of additional caches could have a negative general impact if the increased shared memory footprint results in a lower number of cached projections or reduces the achieved occupancy. The streaming loads cause a significant increase of consumed registers and definitively reduce the occupancy. Hints to compiler reducing the unrolling of inner projection loop are used to prevent this. Furthermore, the desired occupancy can be targeted on NVIDIA platform. Forcing the higher occupancy may result in additional computational load may and cause the compiler to back part of the local variables with slower local memory instead of registers. Vice-versa under low occupancy, the compiler may be able to increase ILP parallelism and perform stream-loading more efficiently.

The importance of the described aspects differs between architectures and an optimal compromise has to be found for each targeted platform. We found out that targeting 50% occupancy is optimal across majority of architectures. On the platforms with a larger register bank, the occupancy above 50% is achieved by default. If 50% is targeted, more registers are available for data streaming which has often a positive impact on the performance. Vice-versa, on the systems with a small register bank the default occupancy is typically low and restricting the amount of used registers to ensure 50% occupancy results in a faster code. We also have found that it is important to keep at least 50% of threads busy in the caching stage. Above this threshold the under-utilization has an impact, but relatively insignificant. Therefore, to cope with shortage of shared memory, the number of cached projections is decreased in steps of 4.

The Maxwell and Pascal GPUs have a large amount of both shared memory and registers, but are bound by the shared memory bandwidth. An area of 64-by-64 pixels are processed by each thread block on these platforms in order to reduce amount of shared memory writes. In the linear interpolation mode the amount of shared memory operations is well balanced with ALU throughput. The streaming of memory reads is not required if the shared memory loads are interleaved with ALU- and SFU-bound interpolation instructions. Thus, the 100% occupancy is targeted and a speed-up of 15% is measured. This is not the case in the nearest neighbor interpolation mode. The shared memory bandwidth is the bottleneck in this case and the performance is improved if multiple shared memory loads are streamed together. Consequently, significantly more registers are required. By default the CUDA compiler does not utilize the streaming capabilities fully, but runs at 62% occupancy. Requesting occupancy to 50% allows to stream more loads together and improves performance by 7%. The impact of occupancy on the performance for both linear and nearest-neighbor interpolation modes are reviewed Table 19.

The Kepler GPUs has the same amount of registers as Maxwell and Pascal. However, a more aggressive unrolling is required and is performed by the CUDA compiler to ensure the wide memory accesses and to enable the longer flow of independent instructions. The ILP parallelism is required to allow 4 warp scheduler to utilize all 6 ALU blocks integrated in the Kepler SM. Consequently, an increased number of registers is used to execute the same code. For instance, the reconstruction based on the linear interpolation as discussed in the previous paragraph would use 55 registers if compiled for the Kepler architecture (compute capability 3.5) instead of only 40 registers which are required if Pascal architecture is targeted. The performance at 100% occupancy is sub-optimal if linear interpolation is performed. On other hand, the 64 registers available at 50% occupancy are not enough to enable efficient streaming of the shared memory loads. Therefore, a small pixel area of 32-by-32 pixels is reconstructed per block and the block is reduced to 128 threads only. The last point is important to keep a high level of ILP and also to achieve a full thread utilization in the caching stage as the number of cached projections is limited due to low amount of available shared memory. Using nearest neighbor interpolation, there is enough registers to organize stream-loading at 50% occupancy also for the larger pixel area. The

Fermi architecture includes only a half of the Kepler registers and is bound to 32-by-32 pixel area in all interpolation modes. While the amount of the shared memory is the same as on Kepler, fewer blocks are required to achieve full occupancy here. Consequently, it is possible to cache more projections at 50% occupancy. On GT200 the amount of registers is even lower and it is not suitable to implement the proposed scheme with sufficiently high performance.

While there is no option to instruct compiler on the desired occupancy on the AMD GCN devices, the used caches are aimed to ensure that at least 50% occupancy can be achieved. The VLIW architecture needs to issue 4-5 independent instructions at each clock. Therefore, it is important to ensure a very large ILP parallelism even in price of significantly reduced occupancy. The larger area is assigned to a thread block for a single-slice reconstruction and a smaller thread block is used to process 8 pixels per thread in all other cases. The algorithm is running at about 35% of the maximum occupancy.

Table 20 summarizes the proposed configuration and gives the measured performance. If only a single slice is available for reconstruction, the new algorithm outperforms the texture-based version across all considered architectures. The maximum speed is better on Fermi and on all AMD architectures if the linear interpolation is performed. Using the nearest-neighbor interpolation the performance is improved on Kepler GPUs and also across all target architectures in the case if the quality is not compromised by a half-float data representation.

The GPU-specific tuning has a major positive effect on the performance. However, new architectures are announced yearly. A throughout study would be required to adjust parameters accordingly. Furthermore, the generated code varies significantly for the devices of different compute capabilities even within the same architecture family. While we had not studied it in detail, there are also differences depending on the CUDA version. To avoid manual work, the actual configuration can be parametrized and a quick search in the parameters space executed to find the optimal settings. The parameters may include numeric options like the targeted occupancy, number of cached projections, unrolling factor, etc. But also switching on and off specific optimizations and caches is feasible. Automated approach would not deliver the optimal performance if the new functional blocks are introduced like Tensor Units on a recently announced NVIDIA Volta architecture. However, it can address the shifts in the operation balance.

6.10. CPU and Xeon Phi

While we are not aiming on the CPU-based architectures, the OpenCL code developed for AMD platform is easy to modify to run also on general-purpose processors and we have evaluated CPU performance for sake of completeness. The texture-engine is not provided by the general-purpose processors. While the recent versions of OpenCL frameworks emulate the missing functionality, better performance is achieved by targeting the algebraic units of CPU directly. We adapted both standard and the ALU-based algorithms to load data directly from system memory instead of fetching it using texture engine. The standard algorithm is additionally modified to perform linear interpolation explicitly. The main difference between two methods is that the ALU algorithm caches data in shared memory while the adapted standard method loads data directly from system memory relaying on CPU caches. In fact, however, there is no a special hardware component backing shared memory. The appropriate blocking is enough to utilize CPU caches and the intermediate caching step is not necessarily required. On other hand, the amount of required computations is reduced if the second term for linear interpolation and a few other intermediate values are pre-computed and cached in shared memory as proposed in sections 6.4 and 6.8. In either case, the performance is improved if multiple slices are reconstructed in parallel and a larger pixel area is assigned to a thread block. Actually, on newer systems supporting 256-bit AVX instructions it makes sense to scale up processing to at least 8 slices in parallel. Allocating a larger amount of pixels per block is relevant to use the cache efficiently. The optimal number is determined by the size of L2 cache available per CPU core.

There are two major OpenCL frameworks supporting general-purpose processors. AMD and Intel deliver their own SDKs, but the processors by both vendors are supported in either case. The AMD framework is not capable to run ALU algorithm efficiently without further adaption. A faster reconstruction is possible if the simpler standard algorithm is used instead. Still, it is significantly slower compared to the performance delivered by the Intel SDK running the same OpenCL code on the same hardware. The speed is even faster if Intel is running the ALU variant with advanced caching mode and h_x caching enabled. The best performance is measured in a 4-slice reconstruction mode and with 64x64 regions assigned per a thread block. To evaluate performance we compared the reconstruction speed against CPU-version of PyHST [12]. It implements multi-thread and cache-aware reconstruction, but does not perform implicit vectorization. Each thread processes a subset of all slices. The compound sinograms for simultaneous reconstruction of several slices are not supported. The performance is summarized in Table 21. PyHST outperforms the OpenCL prototype if it is executed in the single-slice mode, but it is slower if multi-slices are reconstructed at once. The performance of 33 GU/s is measured if a newer server with dual Xeon E5-2680 v.3 is used. Even then the achieved reconstruction speed is inferior to the performance delivered by the slowest of considered GPUs.

There are several architectural differences on a CPU platform which are not considered in our implementation which is optimized for a range of GPU architectures. On GPUs, the fastest available memory is used to cache the data. In case of CPUs, the proposed algorithm is able to utilize L2 cache efficiently, but does not consider the faster L1 cache. The size of L1 cache is significantly smaller compared to the amount of shared memory on GPUs and is limited to only a few kilobytes per CPU core. It is enough to cache the data required to process a single projection. Unfortunately, the context switches are significantly more expensive on CPU platform. When a thread block is scheduled to SM, the SM permanently assigns registers to all threads of the block and can switch executed threads without significant penalty. It is not the case for general-purpose processors. The used registers and the stack pointer have to be saved and restored at each context switch [55]. To avoid an associated performance penalty, the threads on CPU platform are usually execute a large amount instructions before switching. Particularly, for the proposed back-projection algorithm this means that a thread will process multiple projections before giving a way to other threads of a block. Consequently, the data cached from the first projection of a block is already evicted from the L1 cache when the next thread is started. While it can be prevented by synchronizing block threads at each projection iteration, the performance will be penalized just other way due to expensive context switches. The penalty due to context switches is actually playing a significant role in the performance difference between AMD and Intel SDKs. Using Intel SDK, the performance of the standard algorithm is slightly improved if the synchronization is performed before moving to a next projection. On AMD, this penalizes performance drastically as it is shown in Table 21.

The latest versions of Intel OpenCL SDK does not include support of Xeon Phi processors any more. For this reason we had to resort to much older version from 2014. This version perform significantly worse on general-purpose CPUs. The delivered performance is on pair with SDK from AMD. Consequently, the measured performance is barely above the speed of a pair of old Xeon processors. The higher performance probably can be achieved if a way to consider L1 cache is found. However, it is much simpler to target general-purpose architectures using a simple C code. No context switches are required if CPU cores are made responsible for different subsets of slices. And both L1 and L2 caches can be targeted with the appropriate blocking directly.

7. Hybrid Approaches

We have proposed two algorithms to perform back-projection. One relays on the texture engine and is bound to its performance. The second is using shared memory and ALUs with only a small load on the texture engine. In this section we propose two methods to balance the load across all hardware components.

7.1. Combined Approach for Pascal Architecture

On Maxwell and Pascal architectures shared memory and SFU performance are the main limiting factor for the ALU-based algorithm. Both of these resources are very lightly utilized by the texture-based kernel. Therefore, it is possible to run the texture-based kernel for one part of the blocks and ALU-based kernel for another. NVIDIA allows to detect which SM executes the block. Consequently, it is possible to ensure that the desired ratio between texture- and ALU-based kernels is achieved.

An array is statically defined in the global 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 5.6 we proposed an advanced thread mapping scheme for the texture-based kernel. The intention was to keep pixel-to-block assignments minimal and preserve the performance also for small images. The ALU kernel, however, aims for larger image sizes and works with 32-by-32 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. The block-to-pixel assignments are kept in sync with the ALU-based kernel. At each iteration a standard region of 16-by-16 pixels is processed. The thread to pixel assignments follow the mapping described in section 5.5. 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 the texture-based algorithm and to cache data if the blocks execute the ALU-based reconstruction. The performance and utilization of GPU subsystems using the different reconstruction modes is reviewed in Table 22.

In 2-slice reconstruction mode, the performance of the texture-based and ALU-based kernels is very close. Therefore, half of the blocks run the ALU-based reconstruction and the other half uses the texture engine. The hybrid approach outperforms the optimized texture-based method by 30% in this case. The ALU-based reconstruction is significantly faster if only a single slice is reconstructed. The SM on Pascal and Maxwell runs up to 8 blocks with 256 threads each. The ALU reconstruction is executed for 5 blocks and the texture based reconstruction is performed for other 3. It is possible to secure 20% higher throughput over the plain ALU-based reconstruction. Too many registers are required if 4-slices are reconstructed in parallel. Consequently, a low occupancy penalizes the performance of the texture-based kernel significantly.

While it is possible to use the described approach using the nearest-neighbor interpolation, in practice there is a little speed-up. The ALU kernel outperforms the texture-based kernel significantly unless 4-slice reconstruction is performed using the half-float data. Consequently, there is a little effect if they are executed in parallel. The proposed method is only suitable for Maxwell and Pascal architectures. All other devices are bound by the performance of the ALU units. While the Kepler architecture has a very high ALU performance, ALUs are also utilized to perform rounding operations to overcome the slow SFU performance. Since the texure-based kernel also uses ALUs intensively, no performance gains are measured. The used configuration and the achieved performance on Maxwell and Pascal platforms are presented in Table 23.

7.2. Oversampling

There is an alternative approach to improve the utilization of the texture engine using the ALU-based reconstruction. The idea is to sample several values for each bin of a sinogram and use the nearest-neighbor instead of linear interpolation, see Figure 16. While more shared memory is required, the number of computations and memory transactions is reduced in this case. A significant speed-up is achieved compared to linear interpolation if 4 values are sampled for each bin at offsets .00, .25, .50, and .75. Figure 17 compares the described approach against the reconstructions performed using the nearest neighbor and linear interpolation. The reconstruction in oversampling mode is similar to the results obtained using the linear interpolation.

Implementation-wise a few modifications are required for the optimal performance. The amount of used shared memory is quadrupled. To achieve reasonable occupancy the number of cached projections has to be reduced. Typically only 4 - 8 projections are processed in parallel. The amount of available shared memory on Kepler still does not allow to reach 50% occupancy if multiple slices are reconstructed. The performance is significantly penalized if only 2 projections are cached per iteration. Therefore, the Kepler GPUs are running with occupancy under 50%. On the Kepler-based Titan card the actual occupancy allowed by shared memory is hinted and 72 registers are used per thread. The GeForce GTX680 is restricted to 63 registers per thread and hinting occupancy bellow 50% is not useful as extra registers can't be assigned.

To avoid idling at synchronization point, 32 to 64 threads are used to cache each projection row. The texture engine is expected to perform linear interpolation to deliver data at fractional offsets. Consequently, the half-float data representation can't be used together with the oversampling. If the caching of h_x is enabled as explained in section 6.8, the first 16 threads of each warp are assigned to cache the first component of h_x vector and the second half-warp stores the second half of the value. It allows to cache all required data using a single 32-bit instruction and reduces the required shared memory bandwidth.

The data locality is significantly worse if the oversampling approach is used. Up to 4 times more values are accessed by each warp. Consequently, there is a high possibility of shared memory bank conflicts. To reduce the amount of conflicts, the data vectors are split if multiple slices are reconstructed in parallel. The vector components used to represent each sinogram are extracted after texture fetch and are stored into the 2 - 4 separate caches.

On the systems with 32-bit shared memory, a dedicated buffer is allocated for each sinogram component. On the platforms preferring 64-bit over 32-bit loads, the data is split only if 4-slice reconstruction is performed. Two buffers are used in this case, each storing the sinogram components for a pair of reconstructed slices. During the accumulation step, the values are extracted from all caches using the same index and re-combined into the appropriate vector again. To allow 64-bit writes also during a single-slice reconstruction, the re-combination of shared memory writes is performed as explained in section 6.3 and 2 bins are cached per thread at each iteration on the Kepler platform. The used configuration and achieved performance are summarized in the table 24.

8. Conclusion

We have surveyed a range of GPGPU architectures presented by the major hardware vendors in the last 10 years. Table 10 lists architecture details and summarizes rather considerable shifts of the performance balance between different hardware pipelines. The throughput ratio between the floating point and type-conversion instructions has fluctuated 8-fold. The type-conversions are executed at a half rate of the peak floating-point performance on AMD GCN GPUs, but only a single type-conversion instruction can be executed per 12 floating-point operations on NVIDIA Kepler GPUs (considering the peak rates). On the other hand, the type-conversion instructions can be executed in parallel with floating-point operations on NVIDIA Kepler, but not on AMD GCN. The ratio between the theoretical throughput of floating-point instructions and the shared memory bandwidth has changed 2.6 times among the reviewed architectures. 2-fold change is reported between the throughput of floating point operations and the filtering rate of the texture engine. Furthermore, we found that even more considerable architectural changes are introduced in some products. AMD has replaced VLIW architecture with GCN effectively moving from instruction level parallelism to SIMT-only model. On NVIDIA platforms, execution of bit mangling and type conversion operations was shifted between ALU and SFU units. On recent architectures half-float and Tensor Units have been introduced to accelerate machine learning algorithms. The study demonstrates that these changes are highly relevant to the performance of developed algorithms and a significant speed-up is possible if low-level details of GPU architecture are taken into the consideration. In addition to GPU architectures, we also reviewed Intel Xeon-Phi technology in section 6.10. We show that the OpenCL algorithms developed for GPUs are barely suited for this architecture due to the different scheduling model. The standard threaded code seems easier to implement and is a better fitting approach for the general-purpose CPUs and Intel accelerators based on Xeon Phi technology.

We present two algorithms to perform fast back-projection on the variety of GPU architectures. The first utilizes the texture engine to interpolate the data. The second algorithm relies on ALU units and shared memory. Furthermore, we proposed two hybrid approaches to combine these methods and achieved an even higher performance by balancing the load across the GPU subsystems. In section 5.2 we show that a higher utilization of the texture engine can be achieved if the data is 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. On some architectures we can further double performance by switching to a half-precision data representation in price of some penalty to the image quality. The only requirement is the ability of the hardware to perform high-speed transformation between half- and single-precision formats of floating point numbers. Even if half-precision floating point numbers are not directly supported by the texture engine, in section 5.3 we demonstrated that they still can be efficiently utilized by binding a texture with the forged data type. To reach the theoretical rate of the texture engine, the performance bottleneck caused by the low throughput of constant memory and SFU units is resolved by re-assigning work between GPU threads as explained in section 5.6. While this approach results in a lower occupancy on the AMD platform, the resulting performance is considerably improved especially on AMD VLIW-based GPUs. On the NVIDIA platform we are able to enforce 100% occupancy instead, see section 5.7. Consequently, a relatively large amount of local memory is used, but it is completely backed by the L1 cache and the performance is improved significantly on most NVIDIA architectures as well. As can be seen from Figure 6, a high utilization of the texture engine is achieved across all hardware platforms. The algorithm is highly portable and only a minor tuning to the specific hardware is required. In contrast, the ALU-based algorithm requires significant adaptions for some of the considered architectures. As we have shown in section 6.5, different functional blocks may limit the algorithm performance depending on the underlying hardware. Consequently, we were able to significantly boost its performance by re-balancing load of these functional blocks. For Maxwell and Pascal micro-architectures, we run both algorithms in parallel efficiently redistributing load between texture engine, shared memory, and ALUs. This approach is explained in section 7.1. Because of slow throughput of Keplers SFU units, in section 6.6 we proposed an alternative method to perform rounding and type-conversion operations using ALUs instead of SFUs. Consequently, part of SFU load is shifted to ALUs and a higher performance is achieved. In section 6.8, we introduce additional caches for Fermi architecture to reduce the total number of issued instructions. For AMD VLIW architecture, we significantly increase an amount of work per GPU thread. Consequently, the kernel runs at a very low occupancy but utilizes the instruction level parallelism better. In section 6.9 we also discuss the optimal occupancy across other architectures. It depends on the amount of available hardware registers, kernel complexity, and also the ratio between memory and ALU/SFU instructions. We show that targeting both higher and lower occupancy may bring a considerable speedup under different conditions.

Different algorithms can be used to better target a varying balance of subsystem performances in each GPU architecture. We have also shown that it is viable to utilize multiple algorithms in parallel if they are primarily aimed at the different hardware units. The optimal ratio between algorithms can be ensured on NVIDIA platform allowing the balanced usage of all GPU components. The recommended algorithms for each platform are summarized in Table 25. The nearest-neighbor interpolation performs significantly faster on the majority of the considered platforms if the ALU-based algorithm is used. Except on Kepler, the linear interpolation is also accelerated if ALU variant is used either alone or in combination with the texture-based algorithm. If the exact agreement with the standard algorithm is not required, an additional speed-up can be achieved by using the half-float data representation or by replacing the linear interpolation with a combination of the oversampling and the nearest-neighbor approach as explained in section 7.2. There is still rapid progress in parallel hardware and new architectures are announced yearly. To port the algorithms to new devices, the algorithm configuration can be parametrized and a quick search in the parameters space be executed to find the optimal settings. This approach would not deliver the optimal performance if new functional blocks are introduced in the architecture. However, it can address the shifts in the operation balance.

Figure 18 illustrates the history of NVIDIA platform from 2009 to 2016. While the performance of the standard algorithm has grown on the pair with the hardware improvements, the optimized algorithms got an additional boost from utilizing parallelism between GPU subsystems. The speed-up of the optimized back-projection algorithms significantly outperform the respective grow of the hardware performance. Particularly, using new ALU-based algorithm we boosted performance by 3 - 5 times on the Fermi architecture. In the same time, the peak throughput of the floating-point instruction has been only been improved by 50%. The balance of operations has changed on the Kepler architecture significantly. The throughput of bit-mangling and type-conversion operators has been even reduced on GTX680 if compared to GTX580. We still were able to preserve steady grow of the performance by optimizing usage of the texture engine and re-balancing load between SFUs and ALUs. Due to ability to utilize texture engine in parallel with ALUs, on Maxwell and Pascal architectures the algorithm performance again increased above the improvements of the hardware.

NVIDIA Titan X is the newest of the evaluated GPUs. Here, we were able to accelerate the code by 2.5 times using the linear interpolation without loss of image quality. The proposed algorithm is 3.5 times faster if the nearest-neighbor interpolation is required. Even if the reconstruction chain is only able to process a single-slice at a time, the proposed hybrid approach is 2 times faster then the standard algorithm. The achieved speed-up across all platforms is presented in Figure 19. Some architectures can be accelerated as much as 7 times against the state-of-the-art method. 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 iterative reconstruction techniques aiming for high-quality reconstruction. Therefore, the faster implementation lowers the computational demands for high-quality offline reconstruction as well. Furthermore, the general concept of balancing the load among the computational units of the GPU is not limited the presented tomographic reconstruction but rather suggested for any computational intense task.

9. 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.

[1] P. J. Withers, “X-ray nanotomography,” *Materials Today*, vol. 10, no. 12, pp. 26–34, 2007.

[2] 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.

[3] 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.

[4] 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.

[5] 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.

[6] 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.

[7] R. C. Atwood, A. J. Bodey, S. W. T. Price, M. Basham, and M. Drakopoulos, “A high-throughput system for high-quality tomographic reconstruction of large datasets at diamond light source,” *Philosophical Transactions of the Royal Society of London A: Mathematical, Physical and Engineering Sciences*, vol. 373, no. 2043, 2015.

[8] 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.

[9] 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.

[10] A. Shkarin, E. Ametova, S. Chilingaryan, T. Dritschler, A. Kopmann, M. Vogelgesang, R. Shkarin, and S. Tsapko, "An open source gpu accelerated framework for flexible algebraic reconstruction at synchrotron light sources," *Fundam. Inform.*, vol. 141, no. 2-3, pp. 259–274, 2015.

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

[12] S. Chilingaryan, A. Mirone, A. Hammersley, C. Ferrero, L. Helfen, A. Kopmann, T. dos Santos Rolo, and P. Vagovič, "A gpu-based architecture for real-time data assessment at synchrotron experiments," *Nuclear Science, IEEE Transactions on*, vol. 58, no. 4, pp. 1447–1455, 2011.

[13] 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.

[14] 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.

[15] M. Vogelgesang, L. Rota, L. E. Ardila Perez, M. Caselle, S. Chilingaryan, and A. Kopmann, "High-throughput data acquisition and processing for real-time x-ray imaging," in *Proc. SPIE*, vol. 9967, 2016, pp. 996 715–996 715–9.

[16] 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.

[17] W. J. Palenstijn, J. Bédorf, J. Sijbers, and K. J. Batenburg, “A distributed astra toolbox,” *Advanced Structural and Chemical Imaging*, vol. 2, no. 1, p. 18, 2017.

[18] D. Gürsoy, F. De Carlo, X. Xiao, and C. Jacobsen, “Tomopy: a framework for the analysis of synchrotron tomographic data,” *Journal of Synchrotron Radiation*, vol. 21, no. 5, pp. 1188–1193, 2014.

[19] Y. Zhang, L. Peng, B. Li, J.-K. Peir, and J. Chen, “Performance and power comparisons between nvidia and ati gpus,” *International Journal of Computer Science & Information Technology*, vol. 6, no. 6, 2014.

[20] S. Chilingaryan, A. Kopmann, A. Mirone, T. dos Santos Rolo, and M. Vogelgesang, “A gpu-based architecture for real-time data assessment at synchrotron experiments,” in *Proceedings of the 2011 Companion on High Performance Computing Networking, Storage and Analysis Companion*, ser. SC ’11 Companion, 2011, pp. 51–52.

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

[22] R. Shkarin, E. Ametova, S. Chilingaryan, T. Dritschler, A. Kopmann, A. Mirone, A. Shkarin, M. Vogelgesang, and S. Tsapko, “Gpu-optimized direct fourier method for on-line tomography,” *Fundam. Inform.*, vol. 141, no. 2-3, pp. 245–258, 2015.

[23] 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.

[24] J. Treibig, G. Hager, H. G. Hofmann, J. Hornegger, and G. Wellein, “Pushing the limits for medical image reconstruction on recent standard multicore processors,” *The International Journal of High Performance Computing Applications*, vol. 27, no. 2, pp. 162–177, 2013.

[25] 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.

[26] 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.

[27] V. Volkov, "Understanding latency hiding on gpus," Ph.D. dissertation, EECS Department, University of California, Berkeley, 2016. [Online]. Available: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-143.html

[28] X. Mei and X. Chu, "Dissecting gpu memory hierarchy through microbenchmarking," *IEEE Trans. Parallel Distrib. Syst.*, vol. 28, no. 1, pp. 72–86, 2017.

[29] X. Zhang, G. Tan, S. Xue, J. Li, K. Zhou, and M. Chen, "Understanding the gpu microarchitecture to achieve bare-metal performance tuning," in *Proceedings of the 22Nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '17. ACM, 2017, pp. 31–43.

[30] R. V. Lim, B. Norris, and A. D. Malony, "Autotuning GPU kernels via static and predictive analysis," *CoRR*, vol. abs/1701.08547, 2017. [Online]. Available: http://arxiv.org/abs/1701.08547

[31] S. Chilingaryan, E. Ametova, A. Kopmann, and A. Mirone, "Balancing load of gpu subsystems to accelerate image reconstruction in parallel beam tomography," in *Proceedings of the 30th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, 2018.

[32] R. Smith, "The nvidia geforce gtx 1080 & gtx 1070 founders editions review: Kicking off the finfet generation," 2016. [Online]. Available: https://www.anandtech.com/show/10325/

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

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

[35] "Nvidia's next generation cuda compute architecture: Fermi," White Paper, NVIDIA, 2009.

[36] "Nvidia tesla v100 gpu architecture," White Paper, NVIDIA, 2017.

[37] "Amd graphics core next (gcn) architecture," White Paper, AMD, 2012.

[38] G. Ruetsch, P. Micikevicius, and T. Scudiero, "Optimizing matrix transpose in cuda," Manual, NVIDIA, 2014.

[39] "Nvidia's next generation cuda compute architecture: Kepler gk110," White Paper, NVIDIA, 2012.

[40] E. Konstantinidis and Y. Cotronis, "A quantitative performance evaluation of fast on-chip memories of gpus," in *24th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP)*, 2016, pp. 448–455.

[41] M. Doggett, "Texture caches," *IEEE Micro*, vol. 32, no. 3, pp. 136–141, 2012.

[42] E. Konstantinidis and Y. Cotronis, "A quantitative roofline model for gpu kernel performance estimation using micro-benchmarks and hardware metric profiling," *Journal of Parallel and Distributed Computing*, vol. 107, pp. 37 – 56, 2017.

[43] Y. Zhang, Y. Hu, B. Li, and L. Peng, "Performance and power analysis of ati gpu: A statistical approach," in *Networking, Architecture and Storage (NAS), 6th IEEE International Conference on*, 2011, pp. 149–158.

[44] "Developing a linux kernel module using rdma for gpudirect," Manual, NVIDIA, 2017.

[45] B. Sumner, "Opencl extension: Amd bus addressable memory," Manual, AMD, 2011. [Online]. Available: https://www.khronos.org/registry/OpenCL/extensions/amd/cl_amd_bus_addressable_memory.txt

[46] J. Kraus, "An introduction to cuda-aware mpi," Blog post, NVIDIA, 2013. [Online]. Available: https://devblogs.nvidia.com/parallelforall/introduction-cuda-aware-mpi/

[47] "Amd accelerated parallel processing opencl programming guide," Manual, AMD, 2013.

[48] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym, “Nvidia tesla: A unified graphics and computing architecture,” *Hot Chips*, vol. 19, pp. 39–55, 2008.

[49] “Nvidia geforce gtx 680,” White Paper, NVIDIA, 2012.

[50] “Nvidia geforce gtx 980,” White Paper, NVIDIA, 2014.

[51] “Nvidia geforce gtx 1080,” White Paper, NVIDIA, 2016.

[52] “Anatomy of amd’s terascale graphics engine,” White Paper, AMD, 2008.

[53] B. Cabral, N. Cam, and J. Foran, “Accelerated volume rendering and tomographic reconstruction using texture mapping hardware,” in *Proc. of Symp. on Volume Visualization, Tysons Corner, Virginia, USA*, 1994, pp. 91–98.

[54] I. T. P754, *IEEE standard for binary floating-point arithmetic.* New York: Institute of Electrical and Electronics Engineers, 1985, note: Standard 754–1985. [Online]. Available: http://ieeexplore.ieee.org/iel1/2355/1316/00030711.pd

[55] “Writing optimal opencl code with intel opencl sdk: Performance guide,” Manual, Intel, 2011.

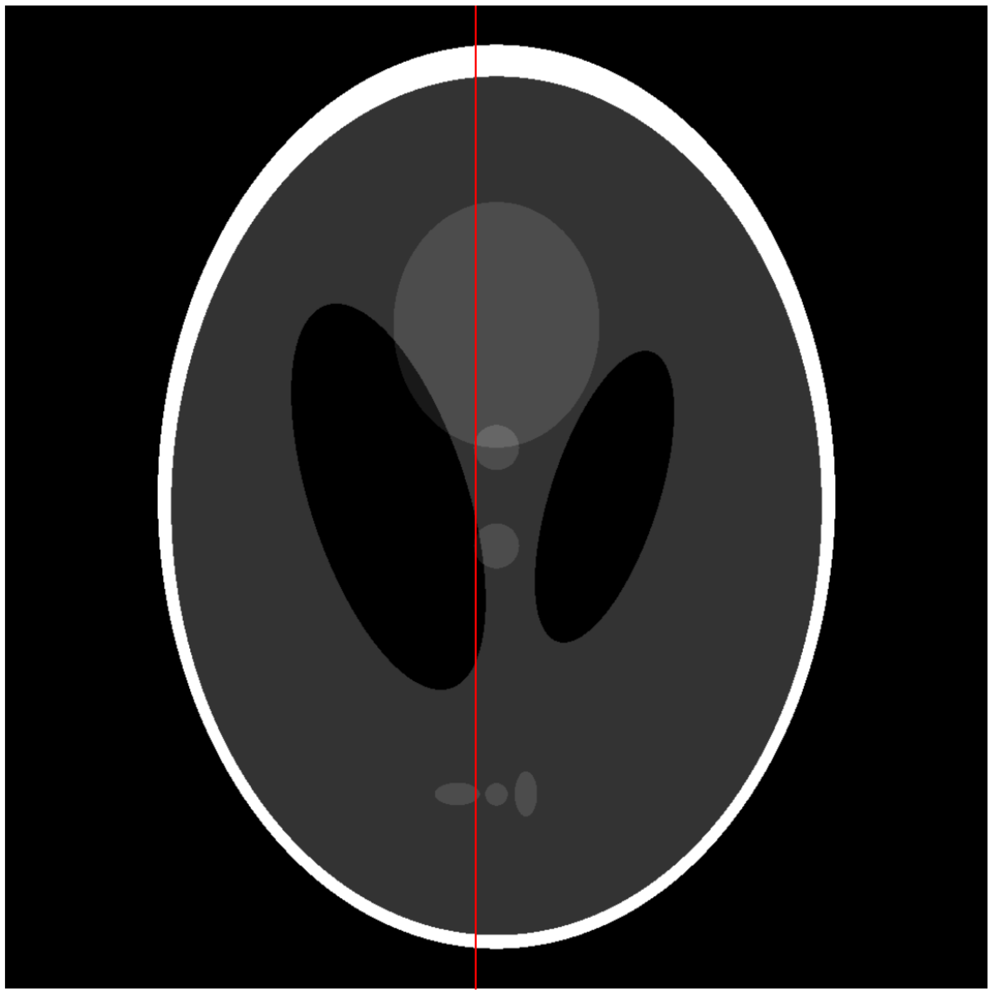


Figure 1: Shepp-Logan phantom used for quality evaluation. All profile plots in the article are shown along the red vertical line.

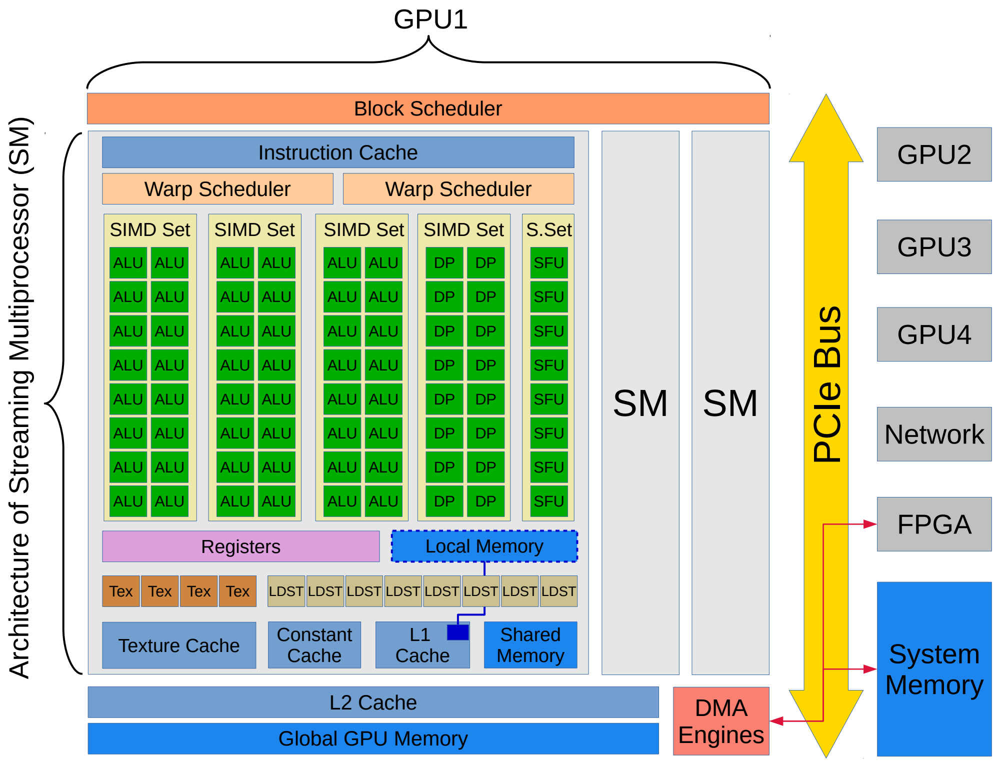


Figure 2: Generalized scheme of GPU architecture. A typical GPU includes DMA engines, Global GPU memory, L2 cache, and multiple Streaming Multiprocessors (SM). The integrated DMA engines are primarily used to exchange data between GPU and system memory over PCIe bus, but also can be utilized to communicate with other devices at the PCIe bus (right). Each SM includes several types of caches and computational units (left).

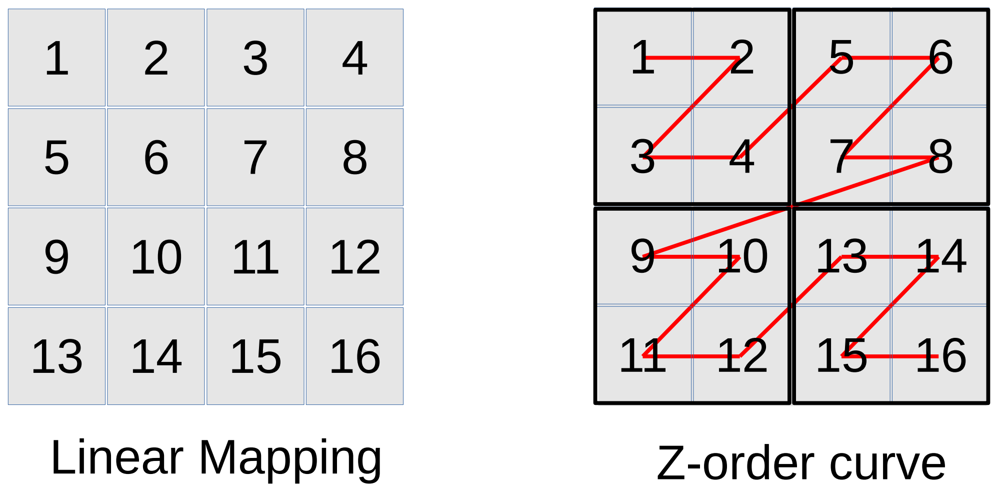


Figure 3: Two ways to exploit spatial locality while fetching 16 texels from a 4-by-4 pixel square. A simple linear mapping is used to assign a group of 16 threads to the square (left). Alternative mapping along Z-order curve improves the spatial locality withing groups of 4 consecutive threads (right)

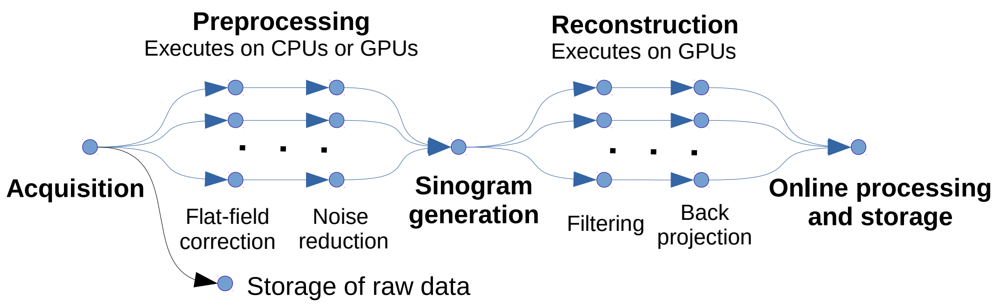


Figure 4: The data flow in image reconstruction framework. The data is split in blocks and processed using pipelined architecture to efficiently use all system resources.

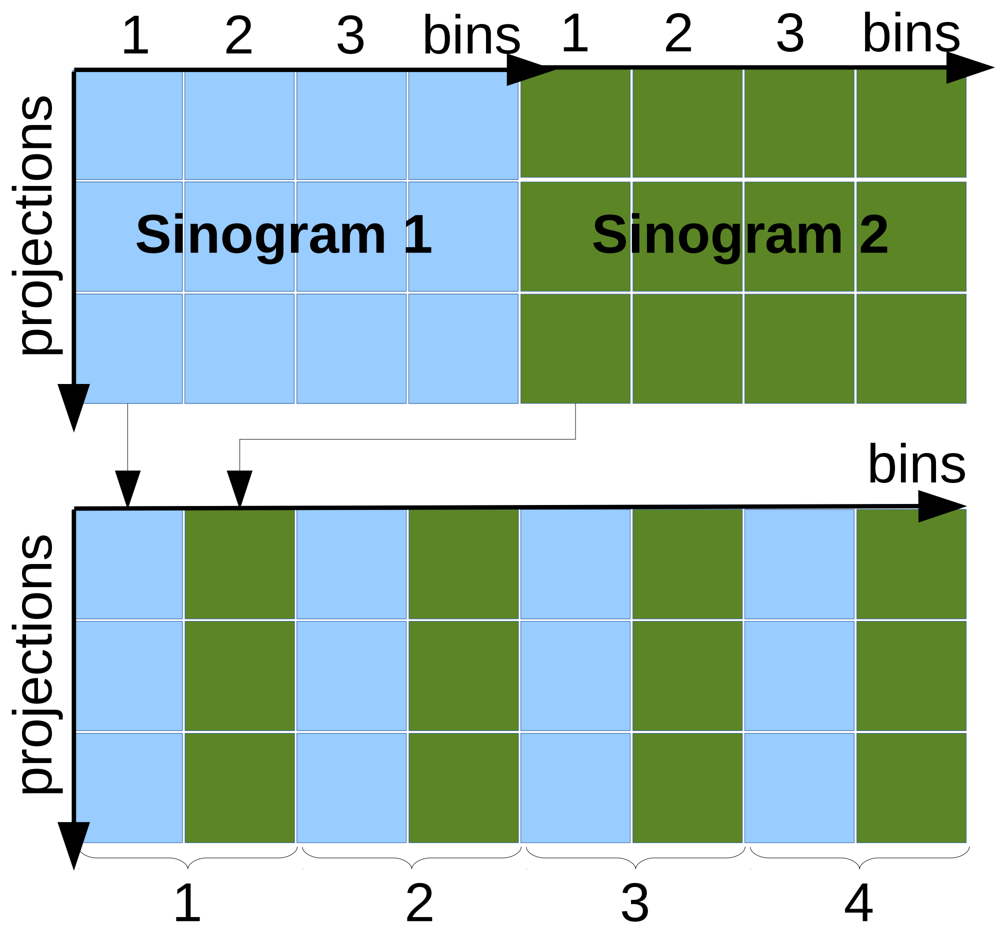


Figure 5: Interleaving two sinograms to allow utilization of full 8-byte filtering bandwidth on post Fermi NVIDIA GPUs.

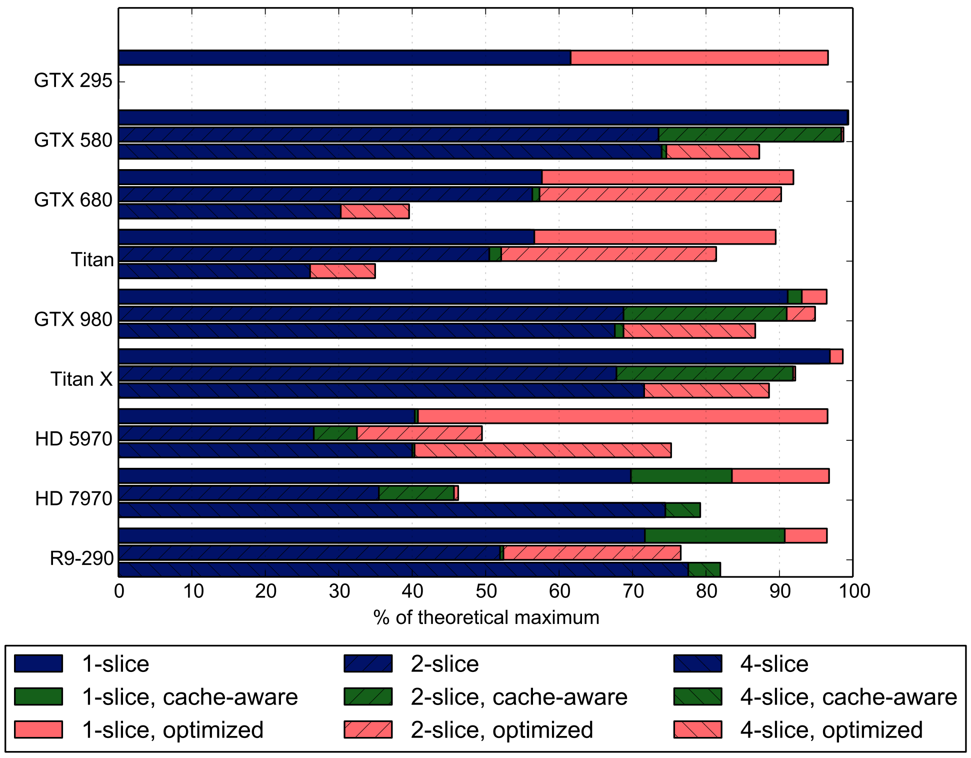


Figure 6: The figure evaluates efficiency of optimizations proposed for texture-based back-projection kernel. The measured throughput is compared to the maximum filter rate of a texture engine and the performance is reported as a percent of achieved utilization. The results are reported also for processing multiple slices in parallel. The nearest-neighbour interpolation is used to measure performance if 4 slices are reconstructed in parallel. On NVIDIA platform the data is also stored in the half float data representation in this case. For single- and dual-slice reconstruction, the performance is measured for bi-linear interpolation mode and the sinogram is stored in the single-precision floating point format on all platforms. The blue bars show performance of the standard Algorithm 1 just modified to process multiple slices in parallel. The green bars show improvements due to a better fetch locality. The red bars show the maximum performance achieved by using Algorithm 3 with optimal combination of tweaks explained through section 5. Table 13 summarizes the architecture-specific parameters used at each GPU. The utilization is reported according to the supported filter rate, not the bandwidth. While the lower utilization is achieved for multi-slices reconstruction modes, the actual performance is higher as 2-/4-slices are processed using a single texture access.

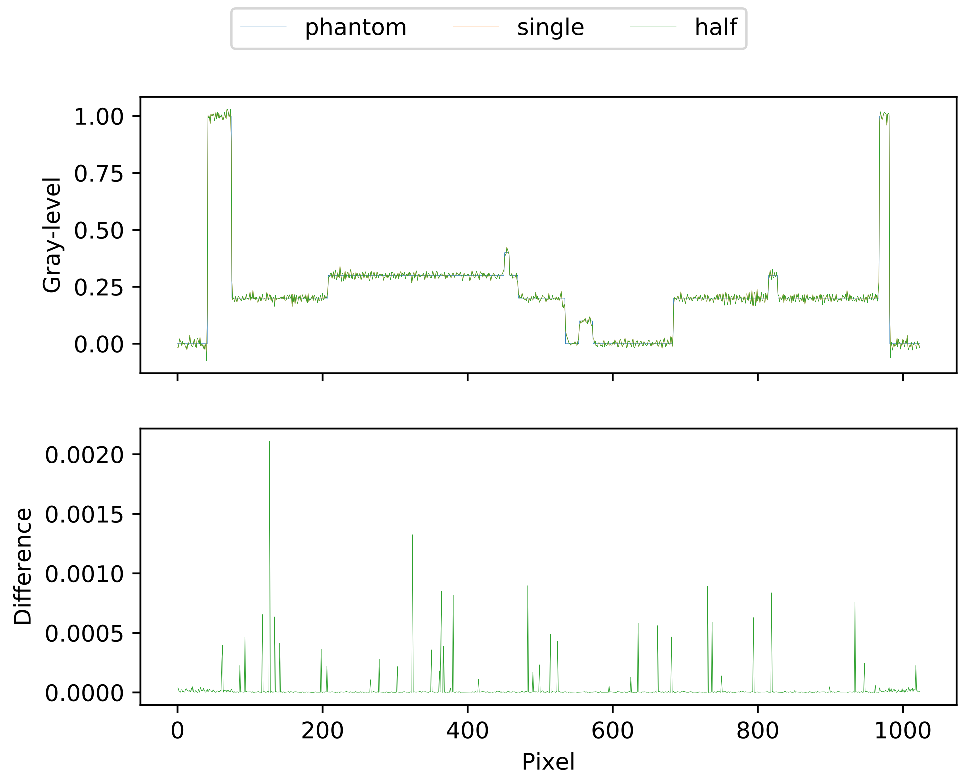


Figure 7: Comparison of two reconstructions of the Shepp Logan Head Phantom using a single-precision (red) or half-precision (gren) input. A profile (top) and absolute difference between reconstructions (bottom) are shown along the line crossing maximum of features on the phantom.

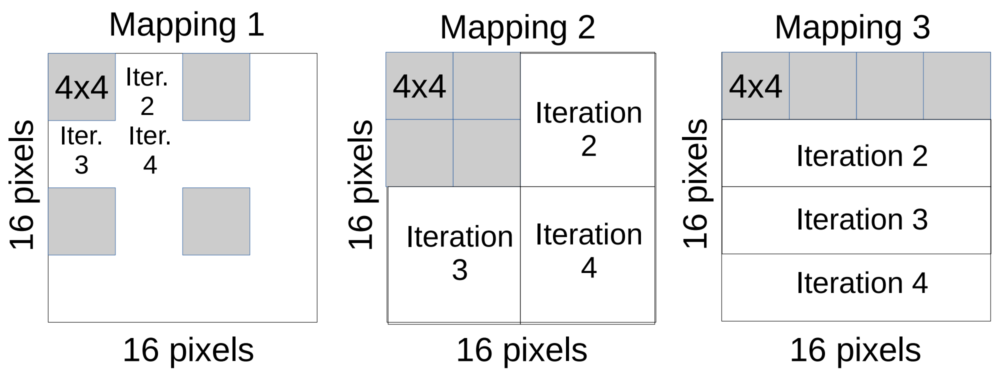


Figure 8: The figure illustrates several ways to assign a block of GPU threads to an area of 16-by-16 pixels. Since 4 projections are processed at once, only 64 threads are available for entire area and it take 4 iterations to process it completely. For each possible scheme in gray are shown all pixels which are processed during the first iteration in parallel. The first mapping (left) is sparse and results in increased cache misses. The second mapping (center) requires more registers and may cause reduced occupancy. So, the third mapping (right) is preferred.

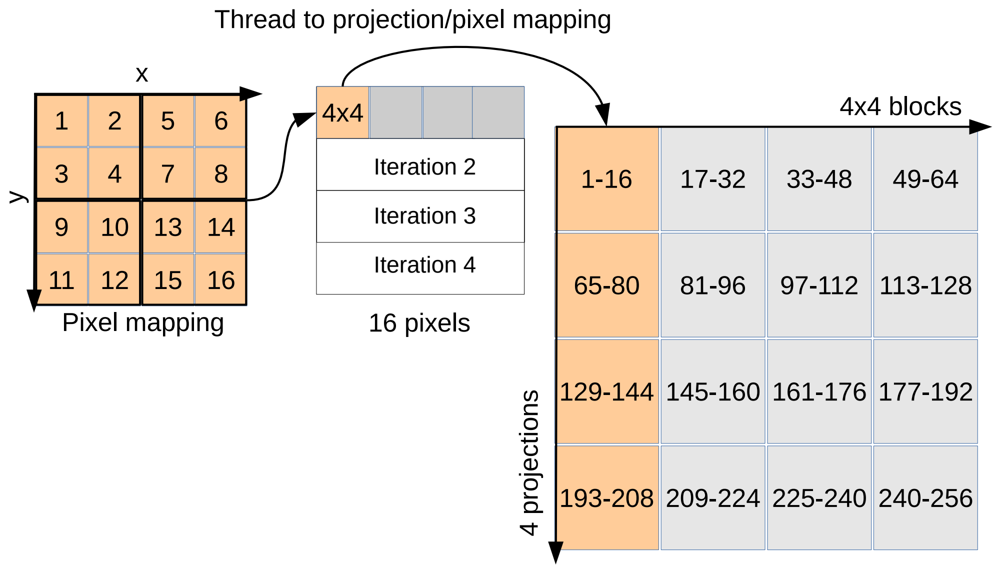


Figure 9: Mapping of a block with 256 threads to reconstruct a square of 16-by-16 pixels along 4 projections. 4 iterations are required to process all pixels. A group of 64 consecutive threads is responsible to process a rectangular area of 16 by 4 pixels (middle). 4 projections are processed in parallel using 4 such groups (right). Each 4-by-4 pixel square is reconstructed by 16 threads arranged along Z-order curve (left). For each output pixel or block of pixels, the assigned range of threads is shown in the figure.

```
Input: Texture and the projection constants $c^C_*$. Dimensions ($n_*$) and parameters
       ($v_*$) as specified in Table 5. The indexes ($m_*$) and other used variables are
       described in Table 6 and 7. Mappings $\vec{m}^2_t$ and $m_p$ are computed as
       explained in Algorithm 2.
Shared: $\tilde{s}^S[64][4]$, $\tilde{r}^S[16][16]$
Output: Reconstructed slice $\tilde{r}^G$
begin
    /* Computing pixel coordinates using the new mapping */
    $\vec{m}^2_g = \vec{m}_b * \vec{n}_t + \vec{m}^2_t$
    $\vec{f}'_g = \vec{m}^2_g - \vec{v}_a$
    /* Computing partial sums */
    $\tilde{s}[4] = \{0\}$
    for ($p = m_p$; $p < n_p$; $p$ += 4)
        $c_s = c^C_s[p].y$
        $h = c^C_a[p] + f'_g.x * c^C_c[p] - f'_g.y * c^C_s[p] + 0.5$
        for ($q = 0$; $q < 4$; $q$ += 1)
            $\tilde{s}[q]$ += tex2d($h$, $p + 0.5$)
            $h$ −= $4 * c_s$
        end
    end
    /* Reduction */
    $\vec{m}^3_t = \{m_t.x \,\%\, 4, 4 * m_t.y + m_t.x \,/\, 4\}$
    for ($q = 0$; $q < 4$; $q$ += 1)
        /* Moving partial sums to shared memory */
        $\tilde{s}^S[n_t.x * m^2_t.y + m^2_t.x][m_p] = \tilde{s}[q]$
        ____________________ sync ____________________
        /* Performing reduction */
        for ($i = 2$; $i \geq 1$; $i$ /= 2)
            if $m^3_t.x < i$ then
                $\tilde{s}^S[m^3_t.y][m^3_t.x]$ += $\tilde{s}^S[m^3_t.y][m^3_t.x + i]$
            end
            ____________________ fence ____________________
        end
        /* To coalesce global memory writes, results are grouped in shared memory */
        if $m^3_t.x == 0$ then
            $\tilde{r}^S[4 * q + m^3_t.y \,/\, 16][m^3_t.y \,\%\, 16] = \tilde{s}^S[m^3_t.y][0]$
        end
        ____________________ sync ____________________
    end
    $\tilde{r}^G[m_g.y][m_g.x] = \tilde{r}^S[m_t.y][m_t.x]$
end
```

Algorithm 3: Optimized implementation of the back-projection kernel relaying on the texture engine to perform interpolation

```
for ($q = 0; q < 4; q += 1$)
    /* Moving partial sums to shared memory */
    $\tilde{s}^S[n_t.x * m_t^2.y + m_t^2.x][m_p] = \tilde{s}[q]$
    ———————— sync ————————
    /* Performing reduction */
    $\tilde{r} = \tilde{s}^S[m_t^2.y][m_t^2.x]$
    for ($i = 2; i \geq 1; i /= 2$)
        $\tilde{r}$ += shfl_xor($\tilde{r}$, $i$, 4)
    end
    /* To coalesce global memory writes, the results are grouped in shared memory */
    if $m_t^3.x == 0$ then
        $\tilde{r}^S[4 * q + m_t^3.y \,/\, 16][m_t^3.y \,\%\, 16] = \tilde{r}$
    end
    ———————— sync ————————
end
```

Algorithm 4: The reduction loop of Algorithm 3 using shuffle instruction

```
Input: Similar to Algorithm 3, but projection constants $c_*^G$ are provided in global
       GPU memory
Shared: $\vec{c}_{cs}^S[s_p]$, $c_a^S[s_p]$
for ($p_b = 0; p_b < n_p; p_b += s_p$)
    /* Caching projection constants in shared memory */
    $m_l = m_t.y * n_t.x + m_t.x$
    $\vec{c}_{cs}^S[m_l] = \{c_c^G[p_b + m_l], c_s^G[p_b + m_l]\}$
    $c_a^S[m_l] = c_a^G[p_b + m_l]$
    ———————— sync ————————
    /* Computing partial sums */
    for ($p = m_p; p < min(s_p, n_p - p_b); p += 4$)
        $c_s = c_{cs}^S[p].y$
        $h = c_a^S[p] + f_g'.x * c_{cs}^S[p].x - f_g'.y * c_{cs}^S[p].y + 0.5$
        for ($q = 0; q < 4; q += 1$)
            $\tilde{s}[q]$ += tex2d($h$, $p_b + p + 0.5$)
            $h -= 4 * c_s$
        end
    end
    ———————— sync ————————
end
```

Algorithm 5: The main loop of Algorithm 3 modified to cache geometrical constants in the shared memory

Table 12: Occupancy and performance of NVIDIA GeForce GTX Titan (Kepler) using 2-slice reconstruction mode

Restricted	Registers	Local Mem.	Occupancy	Performance
No	38	-	75%	320 GU/s
Yes	32	24 bytes	100%	368 GU/s

Table 13: Performance and configuration of cache-aware texture-based back-projection kernel

			Configuration		
GPU	n_v	**Perf.**	**Occupancy**	**L1/ShMem**	**Cache**
GTX295	1	49 GU/s	75%	-	-
GTX580	1	49 GU/s	50%	16/48	-
	2	97 GU/s	50%	16/48	-
	4	172 GU/s	50%	16/48	-
GTX680	1	118 GU/s	100%	16/48	-
	2	232 GU/s	100%	32/32	-
Titan	1	200 GU/s	100%	16/48	-
	2	362 GU/s	100%	32/32	-
GTX980	1	155 GU/s	100%	-	-
	2	304 GU/s	100%	-	-
	4	555 GU/s	75%	-	-
Titan X	1	389 GU/s	100%	-	-
	2	726 GU/s	100%	-	-
	4	1396 GU/s	75%	-	-
HD5970	1	56 GU/s	-	-	256
HD7970	1	115 GU/s	-	-	256
R9-290	1	146 GU/s	-	-	256

The table summarizes the performance and optimal configuration for the texture-based back-projection kernel. Information is provided for all supported slice-modes.

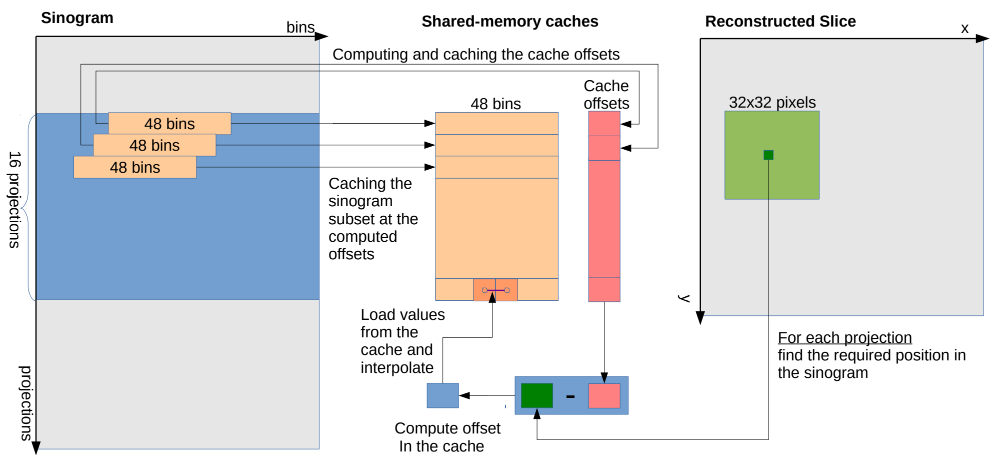

Sinogram
bins
projections
16 projections
48 bins
48 bins
48 bins
Shared-memory caches
Computing and caching the cache offsets
48 bins
Cache offsets
Caching the sinogram subset at the computed offsets
Load values from the cache and interpolate
Compute offset In the cache
Reconstructed Slice
x
y
32x32 pixels
For each projection find the required position in the sinogram

Figure 10: The figure illustrates reconstruction process relaying on the shared memory cache and the algebraic units to perform back-projection. To reconstruct 32x32 square, a thread block caches 48 bins from each projection row. The projections are processed in groups moderated by the size of available shared memory. At first, the required subset of bins in each projection is determined (left). The selected subsets along with their offsets in the projection rows are cached in the shared memory (center). Then, the reconstruction is performed and projections are processed in a loop one after another (right). Each thread is responsible for several pixels of output slice. For each pixel the required position in the sinogram is computed. The cache offset for the considered projection row is subtracted from this position and the offset in the cache is determined (bottom). As the offset is typically not integer, two array elements are loaded from the cache and interpolation is performed.

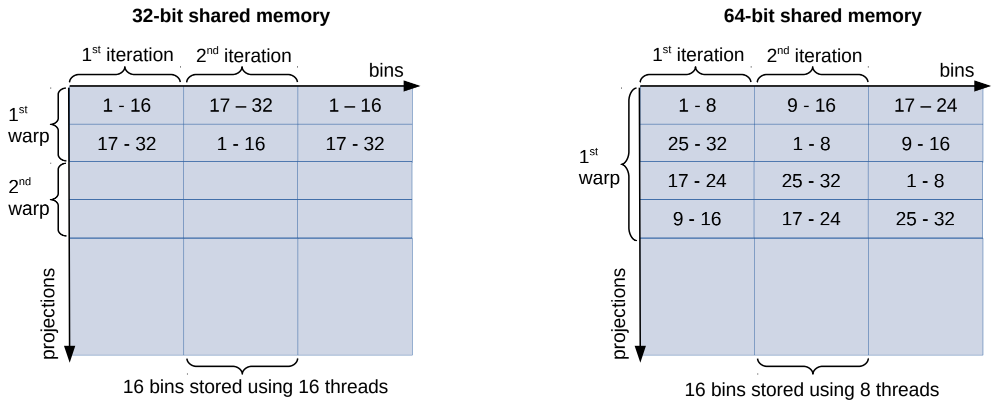


Figure 11: The figure illustrates how the warps are assigned to cache a subset of a sinogram on the systems with 32-bit and 64-bit shared memory. For each projection 48 bins which are required to reconstruct area of 32-by-32 pixels are cached. The shared memory banks used to back each group of 16 bins are specified considering that 32-bit data format is used.

```
Input: Texture and the projection constants $c^C_*$. Dimensions ($n_*$), cache sizes
       ($s_*$), and parameters ($v_*$) as specified in Table 5. The used variables are
       described in Table 6 and 7.
Assume: $n_s = 32$, $n_q = 4$, $s_t = 16$, $s_i = 3$
Shared: $\tilde{d}^S[s_d][\frac{3}{2} * n_s]$, $\tilde{h}^S_m[s_d]$
Output: Reconstructed slice $\tilde{r}^G$
begin
    /* Simplified mapping */
    $\{m_d, m_p\} = \vec{m_t}$
    $m'_t = \{n_t * (m_t.y \,\%\, 2) + m_t.x, m_t.y \,/\, 2\}$
    $m'_g = \{n_s * m_b.x + m'_t.x, n_s * m_b.y + m'_t.y\}$
    /* Set accumulators to 0 and run projection loop */
    $\tilde{s}[n_q] = \{0\}$
    for ($p_b = 0$; $p_b < n_p$; $p_b$ += $s_d$)
        if $m_p < s_d$ then
            /* Compute the minimal required bin */
            $p = p_b + m_p$
            $h_b = c^C_a[p] + f_b.x * c^C_c[p] - f_b.y * c^C_s[p]$
            $h_m = \mathbf{floor}(h_b + c^C_m[p])$
            /* Cache it in the shared memory */
            if $m_d == 0$ then
                $h^S_m[m_p] = c^C_a[p] - h_m$
            end
            /* Cache the data in the shared memory */
            for ($i = 0$; $i < s_i$; $i$ += 1)
                $h = i * s_t + m_d$
                $\tilde{d}^S[m_p][h] = \mathbf{tex2d}(h_m + h + 0.5,\ p + 0.5)$
            end
        end
        ———————————— sync ————————————
        for ($p_i = 0$; $p_i < s_d$; $p_i$ += 1)
            $p = p_b + p_i$
            $c_s = c^C_s[p]$
            $h = h^S_m[p_i] + f'_g.x * c^C_c[p] - f'_g.y * c^C_s[p]$
            for ($q = 0$; $q < n_q$; $q$ += 1)
                /* Compute the offset in cache */
                $h_i = \mathbf{floor}(h)$
                $h_l = h - h_i$
                /* Iterpolate */
                $\tilde{d}_1 = \tilde{d}^S[p_i][h_i]$
                $\tilde{d}_2 = \tilde{d}^S[p_i][h_i + 1] - \tilde{d}_1$
                $\tilde{s}[q]$ += $\tilde{d}_1 + h_l * \tilde{d}_2$
                /* Move to the next position */
                $h$ −= $(n_s \,/\, n_q) * c_s$
            end
        end
        ———————————— sync ————————————
    end
    /* Save the results to global memory */
    for ($q = 0$; $q < n_q$; $q$ += 1)
        $\tilde{r}^G[m'_g.y + 8 * q][m'_g.x] = \tilde{r}[q]$
    end
end
```

Algorithm 6: ALU-based implementation of the back-projection kernel

```
for $(i = 0;\ i < s_i;\ i \mathrel{+}= 1)$
    $h = 2 * (i * s_t + m_d)$
    $d_1 = \textbf{tex2d}(h_m + h + 0.5,\ p + 0.5)$
    $d_2 = \textbf{tex2d}(h_m + h + 1.5,\ p + 0.5)$
    $\textbf{*(float2)(\&}\tilde{d}^S[m_p][h]\textbf{)} = (float2)\{d_1, d_2\}$
end
```

Algorithm 7: The caching stage of Algorithm 6 optimized for architectures with 64-bit shared memory

```
begin
    for $(i = 0;\ i < s_i;\ i \mathrel{+}= 1)$
        $h = i * s_t + m_d$
        $\tilde{d} = \textbf{tex2d}(h_m + h + 0.5,\ p + 0.5)$
        $\vec{d}_1^S[m_p][h] = (float2)\{d.x, d.y\}$
        $\vec{d}_2^S[m_p][h] = (float2)\{d.z, d.w\}$
    end
    ...
        $\tilde{d}_1 = (float4)\{\vec{d}_1^S[p_i][h_i], \vec{d}_2^S[p_i][h_i]\}$
        $\tilde{d}_2 = (float4)\{\vec{d}_1^S[p_i][h_i + 1], \vec{d}_2^S[p_i][h_i + 1]\}$
        $\tilde{d}_2 = \tilde{d}_2 - \tilde{d}_1$
    ...
end
```

Algorithm 8: Modification of Algorithm 6 to split the 4-slice cache as required on Fermi and AMD architectures

```
begin
    $h = s_i * m_d$
    $d_1 = \textbf{tex2d}(h_m + h + 0.5,\ p + 0.5)$
    $d = d_1$
    for $(i = 0;\ i < (s_i - 1);\ i \mathrel{+}= 1)$
        $d_n = \textbf{tex2d}(h_m + i + 1.5,\ p + 0.5)$
        $\vec{d}^S[m_p][h + i] = (float2)\{d, d_n - d\}$
        $d = d_n$
    end
    $d_1 = \textbf{shfl\_down}\ (d_1, 1, s_t)$
    $\vec{d}^S[m_p][h + (s_i - 1)] = (float2)\{d, d_1 - d\}$
end
```

Algorithm 9: Advanced Caching Mode for Algorithm 6

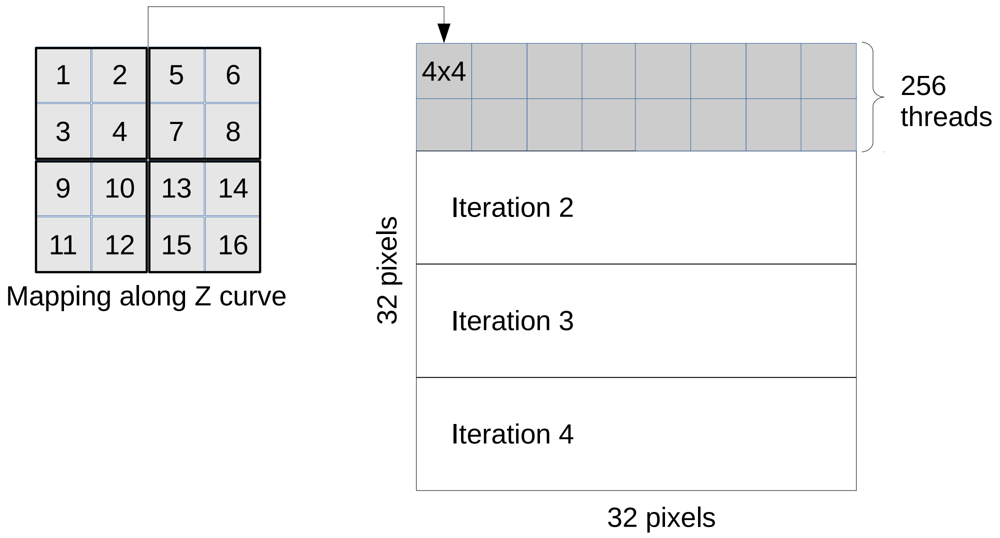


Figure 12: The assignment of block threads to pixels as proposed for ALU-based reconstruction

Table 14: The optimal parameters to prevent shared memory bank conflicts in ALU-based reconstruction kernel

Standard Caching Mode (see section 6.3)

Area	n_v	Platform	Threads	Optimizations
32x32	1	32-bit	16	-
		64-bit	8	write64
	2	32-bit	16	-
		64-bit	16	-
	4	AMD & Fermi	16	double-buffer
		Kepler+	16	-
64x64	1	32-bit	32	-
		64-bit	16	write64
	2	32-bit	32	-
		64-bit	32	-
	4	AMD & Fermi	32	double-buffer
		Kepler+	32	-

Advanced Caching Mode (see section 6.4)

Area	n_v	Platform	Threads	Optimizations
32x32	1	All	16	-
64x64	1	All	32	-

For each considered configuration, the number of threads per projection row and the required optimizations are specified. The *double-buffer* optimization splits the shared memory cache in 2 parts to prevent bank conflicts on the NVIDIA Fermi and all considered AMD architectures. The *write64* optimization combines two writes to shared memory to use full bandwidth of Kepler GPUs.

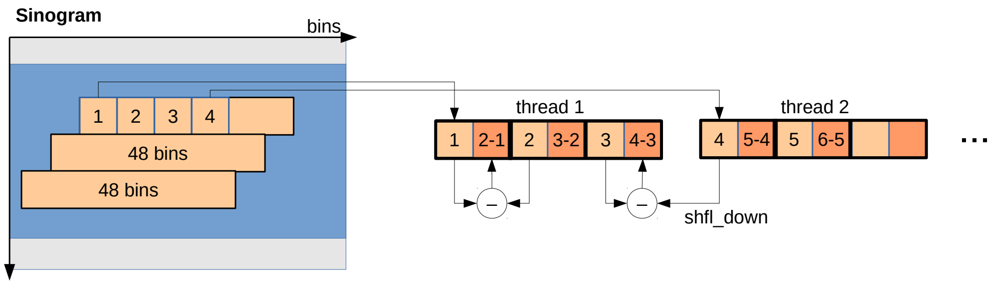


Figure 13: Advanced Caching Mode. Two values are cached for each bin of a sinogram. The second value stores the difference between neighboring bins to allow faster interpolation. The shuffle instruction is used to get values from the bins cached by a different GPU thread.

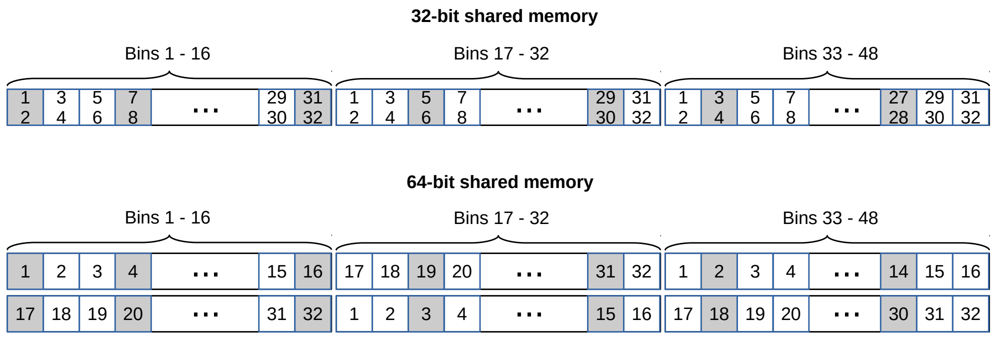


Figure 14: The figure illustrates how the shared memory banks are accessed if advanced caching mode is used, see section 6.4. The presented layout is employed to reconstruct area of 32x32 pixels. The grayed boxes indicate the banks accessed by a warp during the first caching iteration. Two values are cached for each bin and, consequently, each bin spans over two memory banks on the platforms with 32-bit shared memory. On these platform it is also only necessary to avoid conflicts within a half-warp. So, the accesses for second half-warp are not shown.

Table 15: Estimated and measured number of different operations required to perform back-projection using linear interpolation

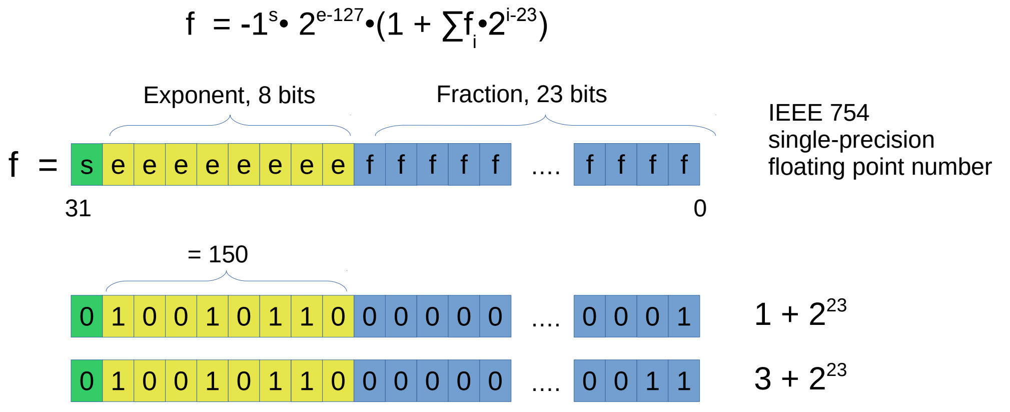
	SFU	Int	FP	Shared	Constant
Estimated	2.03125	1.3125	4.625	1.125	0.265625
Measured	2.032125	5.87		1.265625	0.297875

The table gives the number of operations required to perform back-projection of a single slice using linear-interpolation, processing 4 pixels per GPU thread, and with the advanced caching mode enabled. The measured values are obtained on NVIDIA GeForce Titan X (Pascal) using *nvprof*. The SFU usage is represented by value of *inst_bit_convert* metric. It is impossible integer multiplications from other instructions executed on ALU. Therefore, a common number is given based on the sum of *inst_fp_32* and *inst_integer* metrics. The shared memory operations are given as a sum of counts for *shared_store* and *shared_load* events. To estimate number of constant memory operations, from the number of executed load/store instructions obtained using *ldst_executed* metric we have subtracted all other memory operations which are reported as *shared_store*, *shared_load*, and *global_store* events and all texture transactions which are counted in *tex_cache_transactions* metric.

Table 16: Performance estimates according to model

GPU	Mem	ALU	SFU	OPS	Limit
GTX580	145	**99**	-	**97**	Instructions
GTX680	188	334	**77**	252	SFU
Titan	325	577	**133**	436	SFU
GTX980	**234**	432	315	628	Memory
Titan X	**576**	1062	776	1545	Memory
HD5970	169	145	**114**	114	SFU
HD7970	346	**238**	-	1160	ALU
R9-290	443	**304**	-	1485	ALU

The table gives the estimates for maximum performance of back-projection kernel and reports the performance bottleneck for each considered GPU. The numbers are given in giga-updates per second. The performance limit for each execution unit is evaluated separately and the minimum throughput bounding the kernel performance is highlighted. The estimation is made for a kernel configured to run linear-interpolation and process 4 pixels per GPU thread and running in the single slice reconstruction mode with advanced caching enabled.



$f = -1^s \cdot 2^{e-127} \cdot (1 + \sum f_i \cdot 2^{i-23})$
Exponent, 8 bits
Fraction, 23 bits
IEEE 754 single-precision floating point number
f = s e e e e e e e e f f f f f .... f f f f
31
0
= 150
0 1 0 0 1 0 1 1 0 0 0 0 0 0 .... 0 0 0 1
1 + 2^23
0 1 0 0 1 0 1 1 0 0 0 0 0 0 .... 0 0 1 1
3 + 2^23

Figure 15: IEEE 754 representation of single-precision floating-point number (top) and an example how to get the standard integer representation in fraction part by adding 2^{23} (bottom)

Table 17: Different interpolation modes on Titan (Kepler) GPU

		Instructions		
Method	Performance	FP	Integer	Bit-convert
Stanard	165 GU/s	4.5	1.4	2.03
FP round	197 GU/s	7.5	1.4	0.03
FP round & index	182 GU/s	8.5	1.4	0.03

The table compares performance of 3 different rounding modes described in section 6.6. The performance is measured on the Kepler-based Titan GPU and the number of issued instructions is obtained using NVIDIA profiler. The number of floating point and integer operations is reported by *inst_fp_32* and *inst_integer* metrics correspondingly. The integer counter includes both additions/subtractions executed on ALU and *iSCADD* operations executed on SFU. Consequently, the number of integer operations is constant because the *iSCADD* instruction is just replaced with integer addition. The bit-convert instructions are reported as *inst_bit_convert* and actually represent the rounding and type-mangling operations.

Table 18: Suggested cache settings for ALU-based reconstruction kernel

		Caches			
GPU	n_v	$\tilde{d}$	h_m/h_x	c_s	s_p
Fermi	1, 2	Adv.	h_x	c_s	-
	4	Adv.	h_m	c_s	-
Kepler, Maxwell, Pascal	1	Adv.	h_m	c_s	-
	2, 4	Std.	h_m	-	-
VLIW	1	Adv.	h_m	-	256
	2, 4	Std.	h_m	-	256
GCN	1	Adv.	h_x	-	256
	2, 4	Std.	h_x	-	256
GCN2	1	Adv.	h_m	-	-
	2, 4	Std.	h_m	-	-

Table 19: The effect of occupancy-targeting on the performance for NVIDIA Titan X GPU

Linear Interpolation Mode

Target	Registers	Local Memory	Occupancy	Performance
-	40	-	75%	565 GU/s
50%	64	-	50%	570 GU/s
100%	32	8 bytes	100%	**620 GU/s**

Nearest Neighbour Interpolation Mode

Target	Registers	Local Memory	Occupancy	Performance
-	48	-	62%	1082 GU/s
50%	64	-	50%	**1158 GU/s**
100%	32	40 bytes	100%	954 GU/s

A single slice reconstruction is executed with the settings configured according to Table 20 with the only exception of occupancy which is set as specified in the *Target* column.

Table 20: Performance and configuration of ALU-based back-projection kernel

		Perf. (GU/s)		Configuration				
GPU	n_v	**Lin**	**NN**	n_q	s_d	**U**	**R**	**O**
GTX580	1	80	120	4	16	-	SFU	75%
	2	113	188	4	16	-	SFU	50%
	4	142	247	4	8 [2]	-	SFU	50%
GTX680	1	123	195	8[3]	8[4]	4	ALU	50%
	2	160	290	8	8	2	ALU	50%
	4	165	306	4	8	2	SFU	50%
Titan	1	195	268	8[3]	8[4]	4	ALU	50%
	2	237	429	8	8	2	ALU	50%
	4	278	471	4	8	2	SFU	50%
GTX980	1	218	452	16	8	-	SFU	100%[5]
	2	269	510	16	8	-	SFU	50%
	4[1]	292	567	4	16	-	ALU	50%
Titan X	1	606	1161	16	8	-	SFU	100%[5]
	2	692	1328	16	8	-	SFU	50%
	4[1]	743	1405	4	16	-	ALU	50%
HD5970	1	63	116	16	8[3]	-	-	-
	2	71	146	8	16	-	-	-
	4	73	160	8	8	-	-	-
HD7970	1	178	290	16	8[3]	-	-	-
	2	221	430	4[3]	16[6]	-	-	-
	4	233	450	4	8	-	-	-
R9-290	1	219	341	16	8	-	-	-
	2	298	582	4[3]	16[6]	-	-	-
	4	383	635	4	16	-	-	-

The table summarizes the performance and optimal configuration for the ALU-based back-projection kernel. The performance is reported for the linear and nearest neighbor interpolation modes. The configuration specifies: n_q - a number of pixels per thread, s_d - a number of cached projections, **U** - unrolling hint for inner projection loop, **R** - the units to perform rounding and type conversions (index is always computed using SFU), **O** - the desired occupancy. The caches are configured as specified in Table 18. The number of threads to cache a projection row is determined according to guidelines in Table 14.

1 The configuration and performance are specified for half-float data representation. The half-float values are also cached in the shared memory.

2 Because of the reduced shared memory requirements, 16 projections are cached in the nearest neighbor interpolation mode.

3 A larger 64x64 area is reconstructed if nearest neighbour interpolation is performed. The 16 pixels are assigned to each GPU thread.

4 Each GPU thread caches 2 values per iteration to enable 64-bit writes if nearest neighbor interpolation is used. Consequently, only 16 threads are used per projection row and 16 projections are cached to utilize all threads.

5 The 50% occupancy is targeted in nearest-neighbor interpolation mode.

6 Since 64x64 blocks are assigned to the thread block in the nearest-neighbor interpolation mode, the 32 threads are used per projection row and only 8 projections are cached.

Table 21: Performance using general-purpose processors

Method	n_v	2x Xeon X5650 **AMD**	2x Xeon X5650 **Intel**	Xeon Phi 5110P **Intel**
PyHST	12	9.3 GU/s		-
Standard	1	1.2 GU/s	3.6 GU/s	16.2 GU/s
	4	4.2 GU/s	10.2 GU/s	12.1 GU/s
Synchronized	1	0.9 GU/s	3.9 GU/s	
	4	3.2 GU/s	10.6 GU/s	
ALU algorithm	1	0.9 GU/s	6.1 GU/s	2.7 GU/s
	4	3.7 GU/s	14.1 GU/s	0.2 GU/s

Table 22: Utilization of functional units in hybrid reconstruction mode

Method	Texture	Shared	ALU	SFU	Perf.
Tex-based	100%	20%	40%	10%	726 GU/s
ALU-based	10%	90%	60%	50%	693 GU/s
Hybrid	70%	70%	70%	40%	995 GU/s
Oversampling	20%	90%	50%	40%	1107 GU/s

Utilization of NVIDIA GeForce Titan X (Pascal) subsystems using different reconstruction algorithms. Two slices are reconstructed in parallel according to the configuration given in tables 13 and 20. The utilization is obtained using *nvprof* based on the following metrics: *tex_fu_utilization*, *shared_utilization*, *single_precision_fu_utilization*, *special_fu_utilization*.

Table 23: Performance and configuration of hybrid back-projection kernel

GPU	n_v	**Perf** GU/s	Configuration **T/A**	n_q	s_d	**U**	**R**	**O**
GTX980	1	266	3/5	16	8	-	SFU	100%
	2	389	1/1	4	16	-	SFU	100%
Titan	1	734	3/5	16	8	-	SFU	100%
	2	995	1/1	4	16	-	SFU	100%

The table summarizes the performance and optimal configuration for the hybrid back-projection kernel. Both texture engine and ALUs are used to perform interpolation. The configuration specifies: **T/A** - is a ratio between the blocks executing **T**exture-based reconstruction and the blocks running **A**LU-based algorithm, n_q - a number of pixels per thread, s_d - a number of cached projections, **U** - unrolling hint for inner projection loop, **R** - the units to perform rounding and type conversions (index is always computed using SFU), **O** - the requested occupancy. The caches are configured as specified in Table 18. The number of threads to cache a projection row is determined according to guidelines in Table 14.

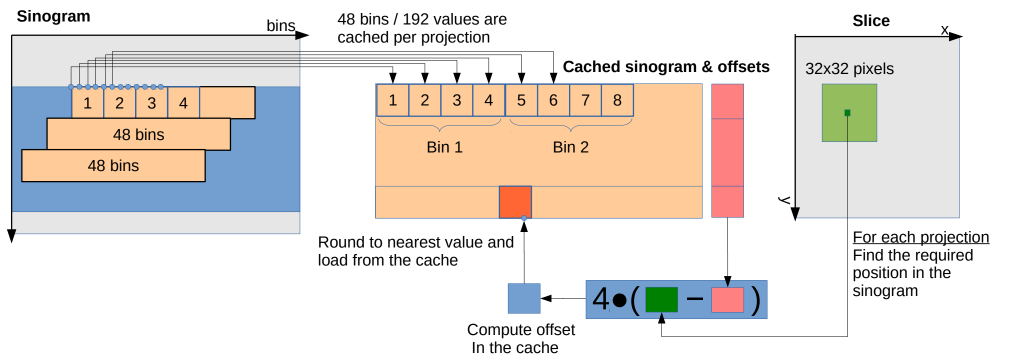

Sinogram
bins
48 bins / 192 values are cached per projection
1
2
3
4
48 bins
48 bins
Cached sinogram & offsets
1
2
3
4
5
6
7
8
Bin 1
Bin 2
Round to nearest value and load from the cache
Compute offset In the cache
4•(−)
Slice
x
y
32x32 pixels
For each projection
Find the required position in the sinogram

Figure 16: The figure illustrates the oversampling reconstruction approach. To reconstruct a 32x32 pixel square, a thread block caches 192 values per projection (left). The values are fetched from 48 bins at uniform intervals using the texture engine. Then, the reconstruction is performed and projections are processed in a loop one after another (right). To determine required position in the cache, the offset from the first bin of the cache is multiplied by 4 and the result is rounded to the nearest integer. The value at this position is loaded from the array and used to update pixel value.

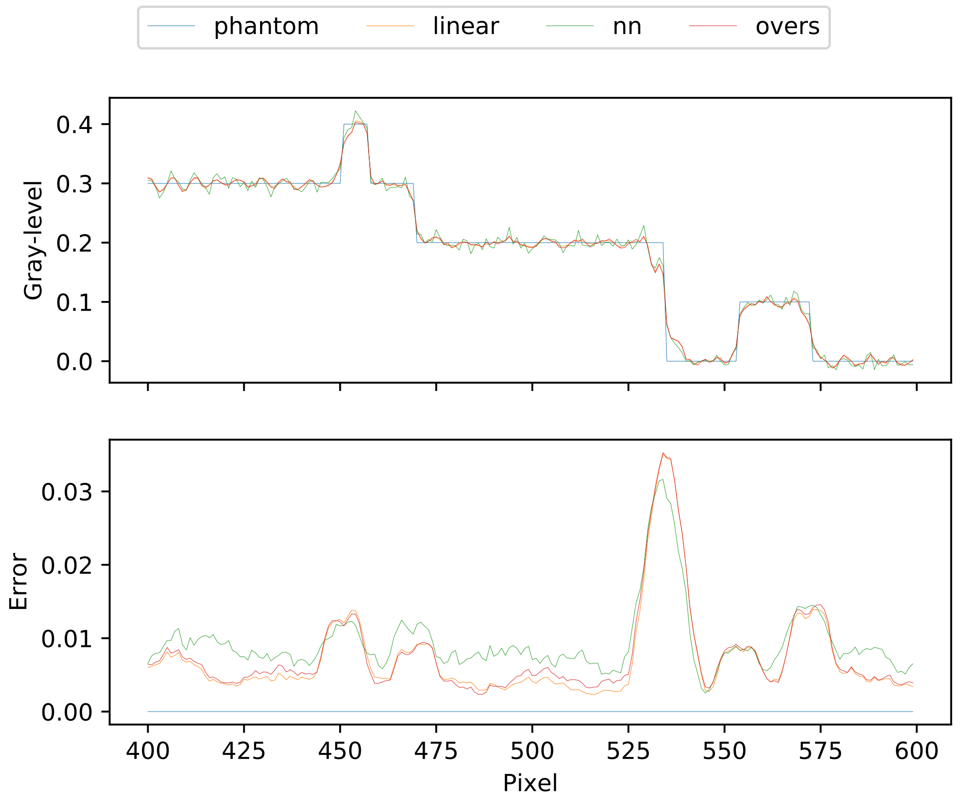


Figure 17: Comparison of the reconstructions performed using nearest neighbor (NN) and linear interpolation with the hybrid oversampling approach ($overs$). The profile plot along the selected line is shown in the top part of the figure for the phantom and all reconstruction methods. The absolute difference from precise phantom image is shown along the same line in the bottom part.

Table 24: Performance and configuration of ALU-based back-projection kernel performing oversampling-based interpolation

		Perf	Configuration					
GPU	n_v	GU/s	n_q	**C**	s_t/s_d	**U**	**R**	**O**
GTX580	1	80	4	1	32 / 8	-	SFU	75%
	2	116	4	2	32 / 8	-	SFU	50%
	4	142	4	4	64 / 4	2	SFU	50%
GTX680	1	123	16	1	32 / 4[1]	4	ALU[2]	50%
	2	160	8	1	32 / 4	2	ALU	50%
	4	165	4	2	64 / 4	2	SFU	50%
Titan	1	195	16	1	32 / 4[1]	4	ALU[2]	50%
	2	237	8	1	32 / 4	2	ALU	43%
	4	279	4	2	64 / 4	2	SFU	37%
GTX980	1	218	16	1	32 / 8	-	SFU	50%
	2	269	16	2	64 / 4	-	SFU	50%
	4	292	4	4	64 / 4	2	SFU	50%
Titan X	1	606	16	1	32 / 8	-	SFU	50%
	2	693	16	2	64 / 4	-	SFU	50%
	4	743	4	4	64 / 4	2	SFU	50%
HD5970	1	63	16	1	32 / 8[1]	-	-	-
	2	71	8	1	32 / 4	-	-	-
	4	73	8	2	32 / 4	2	-	-
HD7970	1	178	16	1	32 / 8[1]	-	-	-
	2	222	4	1	32 / 8	-	-	-
	4	233	4	2	64 / 4	2	-	-
R9-290	1	219	16	1	32 / 8	-	-	-
	2	298	4	2	32 / 8	-	-	-
	4	384	4	4	64 / 4	2	-	-

The table summarizes the performance and optimal configuration for the ALU-based back-projection kernel if oversampling and nearest neighbor interpolation are used to update values of reconstructed pixels. The configuration specifies: n_q - a number of pixels per thread, **C** - a number of separate arrays used to cache singoram (either a dedicated array is used to store each component of sinogram vector or two components are stored together to allow 64-bit writes), s_t/s_d - a number of threads used to cache projection row and a number cached projections, **U** - unrolling hint for inner projection loop, **R** - the units to perform rounding and type conversions (index is always computed using SFU), **O** - the desired occupancy. The caches are configured as specified in Table 18.

1 Each GPU thread caches 2 values per iteration to enable 64-bit writes.

2 The use of SFU is also avoided while resolving array addresses, see section 6.6.

Table 25: Suggested algorithms

GPU	Mode	S	Linear Precise	Linear Appr.	S	Nearest Neighbor Precise	Nearest Neighbor Appr.
GT200	Single	1	TEX	TEX	1	TEX	
Fermi	*	4	ALU	Overs.	4	ALU	
Kepler	Single	1	TEX	Overs.	1	ALU	
	Multi	2	TEX	TEX	4	ALU	
Mxwl+	Single	1	Hybrid	Overs.	1	ALU	
	Multi	2	Hybrid	Overs.	4	ALU	TEX/Half
VLIW	Single	4	ALU	Overs.	1	ALU	
	Multi	4	ALU	Overs.	4	TEX	
GCN	*	4	ALU	Overs.	4	ALU	

The table specifies the fastest algorithms to implement back-projection kernel with linear or nearest-neighbor interpolation at each platform. Individual recommendations are given for the single-slice and multi-slice reconstruction modes. The recommended number of slices is given in column **S**. The options for precise and approximate reconstructions are proposed. In precise mode, the obtained reconstruction is exactly the same as one produced by the standard reconstruction method. In approximate mode, either a half-float data representation is used to accelerate nearest-neighbor interpolation or the oversampling approach is combined with nearest neighbor interpolation to substitute linear interpolation. The performance and optimal configuration for the texture-based algorithm is listed in Table 13. The ALU-based algorithm is described in Table 20 and its oversampling modification is given in Table 24. The hybrid approach is defined in Table 23.

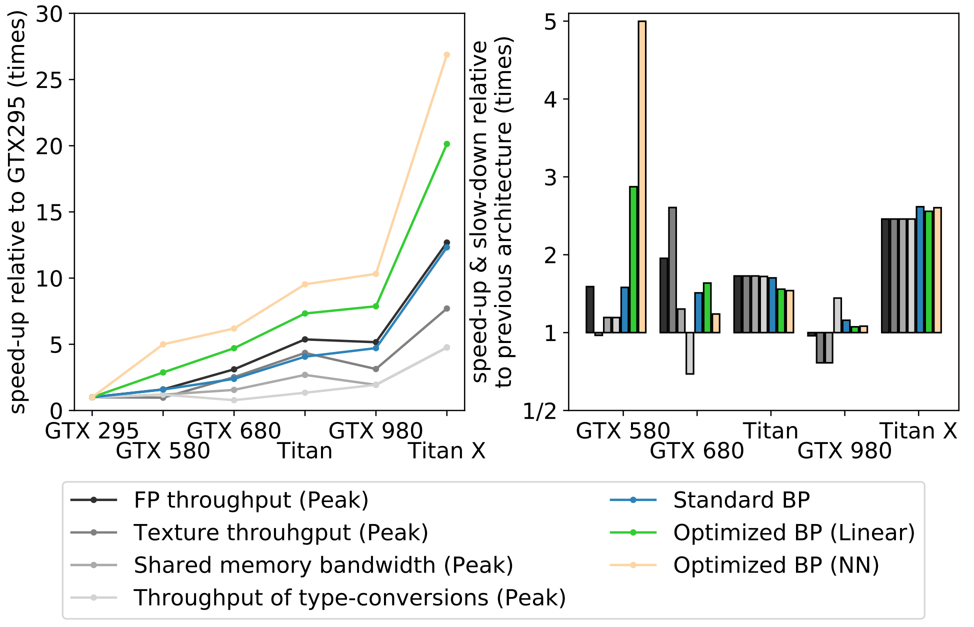


Figure 18: The figure evaluates the theoretical peak throughput of GPU subsystems and the measured performance of standard and optimized back-projection algorithms. The speed-up against NVIDIA GeForce GTX295 is shown in the left part of the figure. The relative speed-up between consecutive architectures is shown in the right part.

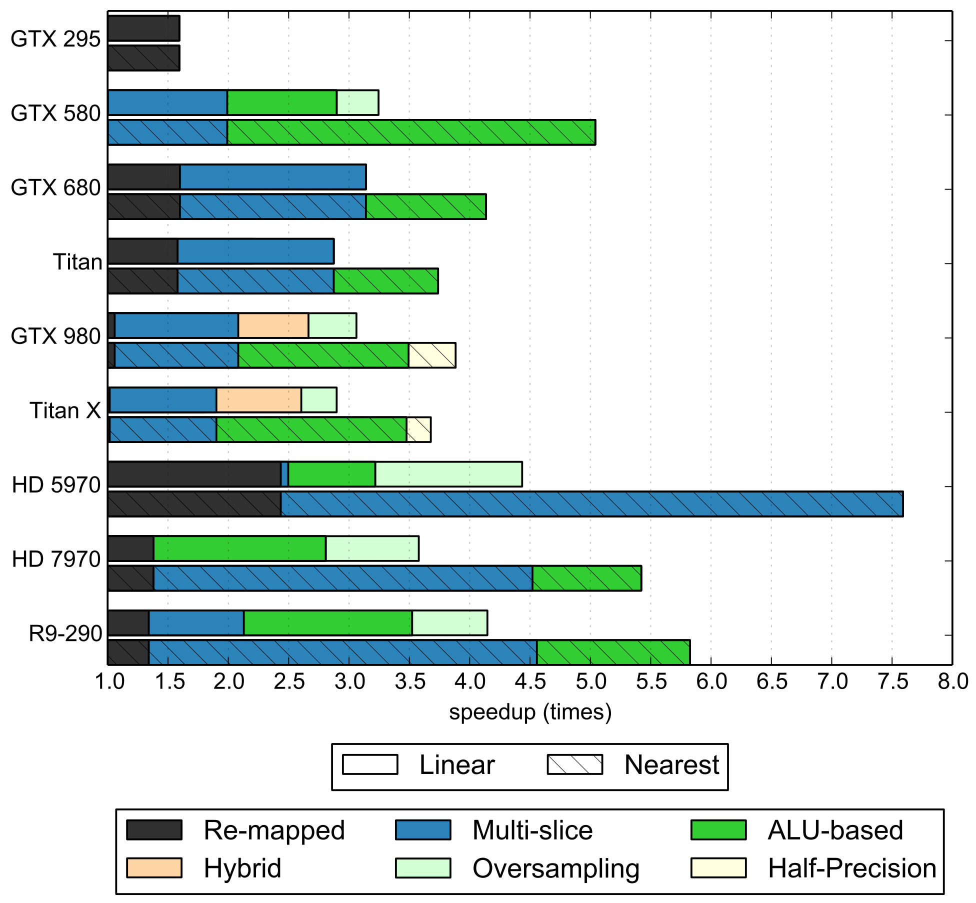


Figure 19: The figure lists the performance improvements of the proposed algorithms using the linear and nearest-neighbor interpolation modes. The speed-up against the standard implementation is measured across all architectures. The black bars show the improvements of a single slice reconstruction performance achieved using the new texture-based kernel due to the optimized fetch locality and reduced load on the constant memory. The blue bars show the increased speed-up using the multi-slice reconstruction. The green bars indicate if the alternative ALU-based kernel outperforms the texture based approach and the achieved gains. The performance of the hybrid approach is shown using the orange color. The last two bars show additional speed-up with approximate methods which do not replicate results of the standard method exactly.