



Universiteit
Leiden
The Netherlands

Faster X-ray computed tomography in real-world dynamic applications

Graas, A.B.M.

Citation

Graas, A. B. M. (2026, February 4). *Faster X-ray computed tomography in real-world dynamic applications*. Retrieved from <https://hdl.handle.net/1887/4291923>

Version: Publisher's Version

License: [Licence agreement concerning inclusion of doctoral thesis in the Institutional Repository of the University of Leiden](#)

Downloaded from: <https://hdl.handle.net/1887/4291923>

Note: To cite this publication please use the final published version (if applicable).

Chapter 4

ASTRA KernelKit: GPU-Accelerated Projectors for Computed Tomography using CuPy

This chapter is based on the article: Adriaan Graas, Willem Jan Palenstijn, Ben van Werkhoven, and Felix Lucka. “ASTRA KernelKit: GPU-accelerated projectors for Computed Tomography using CuPy”. In: *Applied Mathematics for Modern Challenges* 2.1 (2024), pp. 70–92. DOI: [10.3934/ammc.2024004](https://doi.org/10.3934/ammc.2024004)

4.1 Introduction

Computed Tomography (CT) is an imaging technique utilized in scientific, medical and engineering disciplines to resolve the 3D interior of an object from a series of 2D projection measurements. In the most commonly used scanning geometry, *conebeam CT*, X-ray source and detector describe a circular path around the object. However, CT also exists for other imaging modalities and acquisition geometries. For instance, neutron or proton tomography can offer a different contrast than X-ray CT, and in electron microscopy, an electron beam generates projection images from nano-scale objects that are tilted over a limited range of angles inside a vacuum. The wide range of imaging scenarios leads to a vast body of tailored data-processing and reconstruction algorithms described in the literature [4].

Most CT image reconstruction algorithms make use of at least one of two common building blocks: The *forward projection* (FP) maps the object to simulated projection images by modeling the physics and geometry underlying the data acquisition. The model is typically the linear X-ray transform. The *backprojection* (BP) is the adjoint of the forward projection, and maps projection images into the reconstruction volume. The ASTRA Toolbox provides software implementations of these building blocks, and is specifically designed to facilitate the implementation and development of CT algorithms for non-standard geometries [23]. The framework is open source under the GPLv3 license, written in C++, and co-developed by *Vision Lab* at the University of Antwerp and *Centrum Wiskunde & Informatica* (CWI) in Amsterdam. ASTRA Toolbox has been used in multiple tomographic disciplines, e.g., ptychography [114] or neutron imaging [115], and serves as back-end in several frameworks [25, 26, 116, 117].

Due to their pivotal role within algorithms, projection operators – commonly referred to as “projectors” – constitute a significant research focus within computational imaging, inverse problems and scientific computing [4, 7, 118, 119]. The operators process the scan geometry, e.g., a conebeam or parallel beam source, and circular or helical orbits, to emulate X-ray (back)projection lines. The precise algorithmic formulation thereof, which entails, for instance, the discretization and interpolation in the volume, critically influences the accuracy and efficiency of reconstruction. Also computational aspects of projectors are of research interest [120]. To handle large datasets effectively, projectors need to employ memory-efficient strategies, such as partitioning reconstruction volumes into manageable chunks. Moreover, they frequently distribute tasks across multiple computing devices.

Traditionally, ASTRA Toolbox users only invoke projectors indirectly, via calls to one of the pre-implemented CT algorithms (e.g., FBP, FDK, CGLS or SIRT [4, 5, 6, 23, 121]). Figure 4.1(a) illustrates this process. In the first step, a user passes input data, such as the sinogram and geometry, to the algorithm. This is possible due to a Python-C++ interface leveraging the Python-to-C compiler called Cython. In the case of 3D data, the input is then directed to a projector based on NVIDIA CUDA. This is a parallel computing platform and C++ programming model for Graphical Processing Units (GPUs). The final computation is then performed in a CUDA *kernel*, which is a relatively small C++ function that runs on many GPU cores (further discussed in Section 4.2.1).

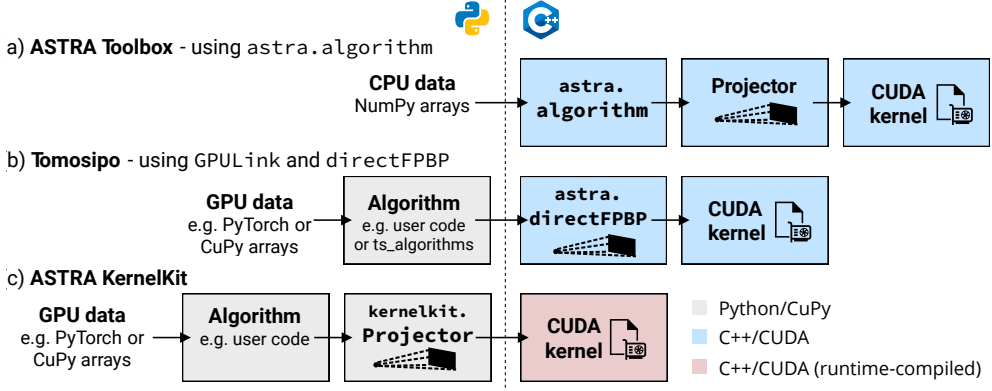


Figure 4.1: Software overview of ASTRA Toolbox, Tomosipo [26], and ASTRA KernelKit, illustrating where its components are located with respect to the Python-C++ language barrier. (a) and (b) show two methods of accessing a projector in ASTRA Toolbox. In comparison, the new ASTRA KernelKit uses a Python-based projector, as well as runtime compilation of the CUDA kernel. The figure is further explained in Section 4.3.1.

In the past decade, algorithms and data pipelines in CT applications increasingly adopted GPU acceleration for tasks beyond (back)projection. Notable are *machine learning* (ML) and *deep learning* (DL) – data-driven algorithms that are leading in the field of inverse problems in imaging [29, 30, 31]. As a consequence, there was a need to pass GPU data more efficiently between external Python frameworks (e.g., PyTorch or CuPy) and ASTRA projectors. In response, ASTRA Toolbox introduced two functionalities. The first, `GPULink`, imports *GPU arrays* (*tensors*, in machine learning terminology) from external packages into ASTRA Toolbox. GPU arrays are multi-dimensional Python arrays that are backed by GPU memory but can directly be manipulated in Python. The second functionality, `directFPBP`, executes CUDA projectors on such external data. When combined, they streamline the use of projectors, and enhances data exchange efficiency with external frameworks. Tomosipo [26] leverages this functionality in a package with a more intuitive Python front-end for geometry manipulations and visualization. Figure 4.1(b) illustrates how this approach gave Python users more flexibility.

Recent advances such as the development of more advanced detector hardware, data transfer, and storage capabilities, have enabled the acquisition of ever larger and more detailed X-ray CT data sets [122], for both laboratory X-ray systems and synchrotron light source facilities. As a result, GPUs are becoming mainstream hardware components for high-performance computing, and frequently used in data preprocessing pipelines. New algorithms, on the other hand, can no longer only be tested on small data sets or toy problems anymore, but instead require a tight integration with GPU-accelerated data processing tools. Scientists and developers working on CT algorithms would therefore benefit from a more fine-grained control over CUDA projectors. However, due to the Python-C++ language barrier, the functionality of CUDA projectors has remained difficult to access up to now.

Recent versions of CuPy and NVIDIA *CUDA Python* have addressed this difficulty, by unifying Python and CUDA into a single interface, which provides full coverage of the low-level CUDA functions. Based on these developments, we present *ASTRA KernelKit*: new high-level CUDA projectors that are fully accessible in Python. In Figure 4.1(c) it is illustrated that the fundamental operations inside the X-ray projection and backprojection, such as data operations or algorithmic formulations, have now either shifted over the language barrier or can be compiled during the Python script. This enables more flexible algorithm development as well as debugging in Python. In this paper, we will explain these advantages and showcase the approach for patch-based reconstructions for deep learning, real-time CT algorithms, and kernel tuning. The article first revisits CT and its implementation in the case of a conebeam geometry in Section 4.2, then provides an overview of KernelKit in Section 4.3, and demonstrates its use for the aforementioned applications in Section 4.5.

4.2 Tomographic reconstruction

4.2.1 The ASTRA CUDA conebeam backprojector

In ASTRA Toolbox, projectors are implemented using CUDA [123]. Their implementations involve two main tasks. Firstly, projectors manage the reconstruction volume, projections, and set-up geometry, including handling transfers between the CPU and GPU device. Secondly, they perform computations associated with the line integrals described in Eq. 1.5. For the latter, building the projector $\mathbf{A}$ as a (sparse) matrix in memory is generally inefficient or impractical. The matrix-vector products $\mathbf{A}x$ or $\mathbf{A}^T y$ that most algorithms need are instead computed *matrix-free*. This entails implementing a function that, e.g., directly computes $\mathbf{A}x$ given x as input.

Kernels are CUDA functions that execute in many parallel threads on GPU cores [123]. A GPU consists of several *streaming multiprocessors* to launch these threads. Threads are divided into *thread blocks*, and each thread computes a small portion of the parallel task. For CT, this portion corresponds to processing a small part of the volume or projection data. In ASTRA Toolbox, kernels are implemented for 2D and 3D conebeam and parallel beam geometries. In this article, we focus specifically on the 3D conebeam backprojection, using the previously mentioned voxel-driven approach, which is most commonly used in high-performance and experimental-data scenarios. However, the concepts presented can be applied to the other projectors in the same way.

The backprojector is summarized in Algorithm 1 using color-coded loops to distinguish CPU code, CUDA parallelization, and kernel code. The main CPU loop processes the projection data y by launching a kernel for each subset of $\bar{N}_\theta$ projection angles sequentially. Each launch issues a sufficient number of 2D $\bar{N}_x, \bar{N}_y$ -sized thread blocks to cover the input volume. A single thread block is responsible of a fixed-size 3D $\bar{N}_x, \bar{N}_y, \bar{N}_z$ -subvolume of x (cf. the green **for** loops). The partitioning of the entire reconstruction problem is therefore defined by four parameters $\bar{N}_x, \bar{N}_y, \bar{N}_z, \bar{N}_\theta$ (cf. line 1). In ASTRA Toolbox, they are compiled with constant values 16, 32, 6, 32 into the binary code. These values are taken after a

Algorithm 1: The ASTRA Toolbox CUDA conebeam backprojection $\mathbf{A}^T$.

Input : Volume $x \in \mathbb{R}^{N_x, N_y, N_z}$, projections $y \in \mathbb{R}^{N_\theta, N_u, N_v}$, source and detector geometries.

Output: An updated $x \in \mathbb{R}^{N_x, N_y, N_z}$.

```

1 const int  $\bar{N}_x, \bar{N}_y, \bar{N}_z, \bar{N}_\theta \leftarrow 16, 32, 6, 32$  ▷ ASTRA Toolbox defaults
2 TextureObject  $\tilde{y} \leftarrow \text{ToTexture}(y)$  ▷ Using a CUDA Array
3 for  $\theta_{\text{start}}$  in  $\{\theta_{\text{start}} \bar{N}_\theta : 0 \leq \theta_{\text{start}} \bar{N}_\theta < N_\theta\}$  do
4   for  $z_{\text{start}}$  in  $\{z_{\text{start}} \bar{N}_z : 0 \leq z_{\text{start}} \bar{N}_z < N_z\}$  parallel do ▷ Blocks
5     for  $i$  in  $[0 \dots N_x - 1]$  parallel do ▷  $\bar{N}_x$  threads
6       for  $j$  in  $[0 \dots N_y - 1]$  parallel do ▷  $\bar{N}_y$  threads
7         for  $k$  in  $[z_{\text{start}} \dots z_{\text{start}} + \bar{N}_z - 1]$  do
8           for  $l$  in  $[\theta_{\text{start}} \dots \theta_{\text{start}} + \bar{N}_\theta - 1]$  do
9              $(p, q) \leftarrow \text{ProjectRay}(\text{source}, \text{detector}, i, j, k, l)$ 
10             $x_{i,j,k} \leftarrow x_{i,j,k} + \text{Interpolate}(\tilde{y}_l, p, q)$ 

```

Algorithm 1. The ASTRA Toolbox CUDA voxel-driven conebeam backprojector algorithm $\mathbf{A}^T$. The algorithm sequentially processes subsets of $\bar{N}_\theta$ angles (line 3), and volumetric chunks of $\bar{N}_z$ vertical voxels in parallel thread blocks (line 4). The •-loops correspond to CPU code, •-loops to CUDA parallelization, and •-loops to kernel code.

bruteforce search over multiple geometries and volume sizes. In Section 4.5.3, we will explore different selections of parameters using a kernel tuning strategy.

Zooming in onto lines 7–10 of Algorithm 1 reveals the idea of the voxel-driven approach. Rather than tracing rays backward through the volume, a single kernel thread processes a stack of $\bar{N}_z$ voxels and collects contributions from $\bar{N}_\theta$ projections. Doing so, GPUs can leverage the locality of projection data by *coalescing* write operations to the reconstruction volume, i.e., by combining the writes of multiple threads to adjacent memory locations into a single memory transaction [20, 120]. Moreover, handling multiple voxels in a thread can reduce geometry computations and benefit from instruction-level parallelism [124]. For each voxel $x_{i,j,k}$ and projection angle l , a thread first computes the intersection of an X-ray at a point $(p, q) \in \mathbb{R}^2$ in the detector plane, using the spatial coordinates of the X-ray source and voxel. It then calculates a contribution to $x_{i,j,k}$ via linear interpolation of the pixel values around (p, q) in y_l . This is efficiently performed using *CUDA textures*. Textures are specific read-only structures in CUDA that enable fast and accurate bilinear or trilinear interpolation in 2D or 3D arrays. ASTRA Toolbox creates such 3D texture, before the kernel call (line 2), with a *3D CUDA Array* as the underlying resource. A CUDA Array is a data structure in which elements are stored with better spatial locality than linear memory. We will further discuss these data structures in Section 4.5.3.

4.3 Software

4.3.1 ASTRA KernelKit

KernelKit, our extension to ASTRA Toolbox, is designed to enhance the accessibility of tomographic projectors, like Algorithm 1, for scientists working within the Python ecosystem. This facilitates prototyping of new algorithms and customization for high-performance use cases. KernelKit maintains the capability to work with nonstandard geometries, a prominent feature in ASTRA Toolbox.

To illustrate its advantages, we compare KernelKit to the most flexible approach that is currently possible with ASTRA Toolbox, see Figure 4.1(b). This approach, which is the working principle of Tomosipo, consists of two parts. The first, **GPULink**, is a straightforward method to point ASTRA to a contiguous memory region on the GPU allocated by an external software (e.g., a PyTorch tensor). In contiguous memory, all bytes are stored sequentially, to allow efficient memory access. The second, **directFPBP**, offers a forward or backprojection that directly outputs into this external region. The combination enables efficient use of ASTRA Toolbox projectors, while allowing data manipulation with external frameworks. KernelKit expands and improves upon this functionality in three crucial aspects:

- *Python-based projectors*: Except for the kernels, KernelKit projectors are fully written in Python. In comparison to ASTRA Toolbox, there is no Python-C++ interfacing code, and projectors are transparently accessible. The user is therefore in full control of the lifetime of the projectors as well as the data that it stores internally. For example, KernelKit geometries can be modified without destroying and recreating the projector. Application-specific tomographic modalities, e.g., hyperspectral or dynamic CT, or algorithms with partial data, e.g., the random-batch gradient descent, often require a fine-grained control over the projectors to be most efficient.
- *High-level CUDA availability*: The linkage between KernelKit (Python) and NVIDIA CUDA (C++) is provided by CuPy. CuPy is a multi-dimensional array library for GPU-accelerated computing with Python [125]. As a drop-in for NumPy and SciPy, it implements a wide range of algorithms and linear algebra utilities using CUDA-libraries. Since KernelKit is built upon CuPy, users can leverage advanced CUDA functionality within and alongside custom algorithms and projectors. Section 4.5.1 will illustrate an example involving the use of CUDA graphs. In contrast, developing CUDA-accelerated algorithms with ASTRA Toolbox is less straightforward due to the language barrier that prevents accessing CUDA data types, such as streams or texture objects, created in the toolbox.
- *Runtime compiled kernels*: KernelKit compiles CUDA kernels during the Python script, via the NVIDIA runtime compiler (NVRTC). In contrast to ASTRA Toolbox, which uses precompiled kernels, variables such as the $\bar{N}_x$, $\bar{N}_y$, $\bar{N}_z$, $\bar{N}_\theta$, or the processing axis order, can now be decided during the Python script. With runtime compilation,

an existing kernel can be swapped out for a user-written implementation of the Radon transform, or an implementation can be optimized on basis of the input data.

4.3.2 Package overview

Since our software is written on top of numerical libraries, it benefits from concise formulations of the mathematical operations. This keeps the source code of our package to a minimum, while providing an expressive and flexible interface. The code consists of three levels:

1. *CUDA kernels*: FP and BP kernels are implemented as CUDA source files. As kernels are runtime compiled, the end-user can provide custom sources. Additionally, we use Jinja2 [126], a placeholder-style templating engine, to generate a specialized CUDA kernel code before compilation. The voxel-driven conebeam kernel, for example, is stored in `cone_bp.cu`.
2. *Kernel classes*: A Python class parses, compiles, and calls the CUDA sources, e.g., `VoxelDrivenConeB` handles the `cone_bp.cu` kernel. These classes abstract low-level CUDA tasks, such as the configuration of block sizes, precomputation of kernel parameters, and passing of arguments to the kernel. They furthermore do not perform CPU-GPU memory transfers. For custom kernels or specialized algorithms, the user can extend or modify their behavior.
3. *Projectors*: The last layer abstracts the details of the kernel and provides an interface for algorithms at the level of $\mathbf{A}$ and $\mathbf{A}^T$. A projector, such as `ForwardProjector` or `BackProjector`, compiles a kernel on initialization, and executes it multiple times on invocation. Projectors provide a detailed interface for memory management, axes conventions, and geometries, and are the recommended tool for custom algorithms.

Additionally, a few auxiliary modules provide the tools to create and manipulate geometries, e.g., `ProjectionGeometry` and `VolumeGeometry`. As examples for algorithms, we provided the Feldkamp-Davis-Kress algorithm [6] as `fdk()`, and SIRT [4] as `sirt()`. `XrayTransform` furthermore builds a linear X-ray operator on top of the projectors, in the same style as SciPy operators and the MATLAB Spot toolbox [127]. This enables a simpler interface, and is more suitable for a high-level integration with other packages.

4.3.3 Related software

In this section we position KernelKit by contrasting it with related software solutions. We restrict the comparison to packages that implement CUDA projectors, and that expose these via a Python front-end. They are ASTRA Toolbox, TIGRE [24], Pyro-NN [128], TorchRadon [129], and TomocuPy [130]. Other commonly used frameworks are Tomosipo [26], Operator Discretization Library (ODL) [117], TomoPy [25], and Core Imaging Library (CIL) [116]. Rather than implementing projectors, they use ASTRA Toolbox, TIGRE and/or Scikit-image as a back-end, and provide additional algorithms, mathematical abstractions, or data processing functionality on top.

We first acknowledge that KernelKit does not aim to provide extensive algorithmic functionality, like TomoPy, ODL, CIL, TIGRE or ASTRA Toolbox. However, its distinctive aspects, i.e., Python projectors, high-level CUDA, and runtime compilation (Section 4.3.1), are currently not offered by any other package. Another unique aspect about KernelKit is its inherited support for *vector geometries*, meaning that the source and detector do not need to describe a perfect circular orbit for (back)projection.

In terms of performance, we show that our conebeam backprojection matches or exceeds ASTRA Toolbox, depending on the configuration of our package (see Appendix 4.4). Since Tomosipo uses ASTRA Toolbox as a back-end, this also holds for Tomosipo. Several packages provide alternative backprojection or reprojection operators. For example, TorchRadon provides a multi-channel 2D backprojection kernel, and TomocuPy implements a kernel for GridRec reconstruction [131]. These kernels are understood to outperform ASTRA Toolbox counterparts for the use cases that they are suited for. However, the strength of KernelKit lies in its flexibility, as it allows users to easily add or customize kernels according to their requirements.

All compared packages are on the same level as Tomosipo (Fig. 4.1(b)), meaning that the main software features are behind the language barrier. Yet, there are also benefits to approaches with C++ projectors. In ASTRA Toolbox and TIGRE, projectors and algorithms can be shared with MATLAB. In Pyro-NN and TorchRadon, projectors are written as TensorFlow C++ and PyTorch C++ extensions, and subsequently exposed to Python as neural network layers. In these approaches, the projectors can better integrate with the low-level features of the machine learning framework. While Python projectors enable easier debugging and development, C++ is sometimes better-suited for low-latency or highly multi-threaded environments.

Among the compared packages, the design principle behind TomocuPy is closest to KernelKit. TomocuPy utilizes both Python, with CuPy, and C++, using C++-level CUDA libraries. TomocuPy does not provide Python objects for projectors or kernels, but is aimed at high-performance reconstruction pipelines for synchrotron light sources. In their approach, the two languages are connected via a thin Python-C++ interface called SWIG (Simplified Wrapper and Interface Generator) [132].

Lastly, package developers should consider the convenience of integrating projectors into their application. ASTRA Toolbox’s direct API, illustrated in Fig. 4.1(b), offers straightforward projector invocation. Tomosipo [26] enhances this with a more intuitive Python front-end for geometry manipulations and visualization. On the other hand, TIGRE provides functions in Python to invoke $\mathbf{Ax}$ or $\mathbf{A}^T y$, but lacks support for GPU arrays, making integration less straightforward. KernelKit, TomocuPy, Pyro-NN and TorchRadon all use tensor or array data structures from their host frameworks (CuPy, TensorFlow and PyTorch). These leverage DLPack, a standardized approach for exchanging GPU arrays.

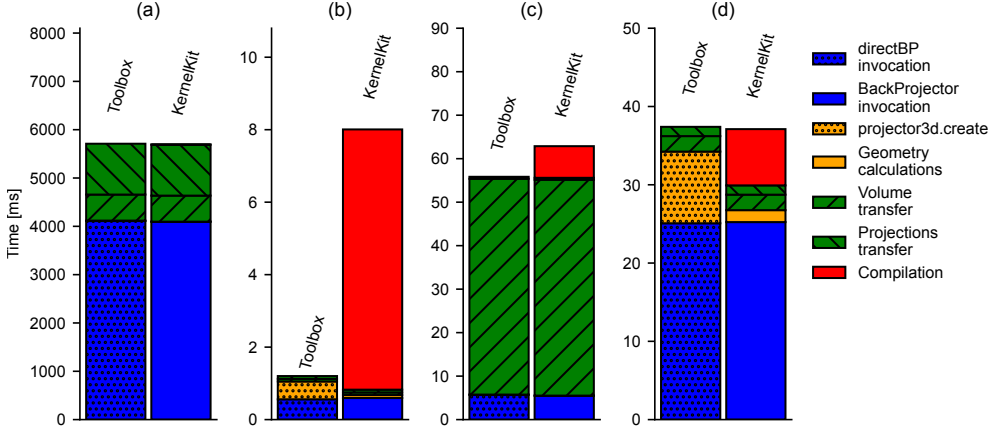


Figure 4.2: Validation of KernelKit in “Toolbox configuration”, i.e., such that it precisely matches the behavior and kernel settings of ASTRA Toolbox, on an NVIDIA GA102GL [RTX A6000]. (a) is a 1000^3 volume, (b) a 50^3 volume, (c) a $(300, 300, 1200)$ volume with $N_\theta := 3$ projections of size $(300, 1200)$, and (d) is a 2000-by-2000 slice reconstructed by $N_\theta := 1000$ images of a $(2000, 1)$ line detector.

4.4 Runtime and computational overhead analysis

In the following, we configure KernelKit in “Toolbox configuration”, i.e., such that it precisely matches the behavior and kernel settings of ASTRA Toolbox. We subsequently compare the runtimes of different components of the two software packages side-by-side. This validates that our Python implementation is correct, and allows us to detect computational overhead differences (e.g., due to the use of Python, CuPy, or the NVRTC compiler). Moreover, it verifies that the comparison between the two software packages is fair. We take the ASTRA Toolbox conebeam kernel with default parameters, and disable the use of advanced CuPy features. Software tests are used to validate the implementation numerically. Note that the comparison does not aim to demonstrate the best achievable runtime of either package.

Figure 4.2 displays a break-up of calls to ASTRA Toolbox, using `directFPBP` (cf. Section 4.3.1), and KernelKit, using `BackProjector`, for four reconstruction problems. The problems are selected such that they load the projector differently. Problem (a) and (b) are a large and small reconstruction problem. Problem (a) consists of a 1000^3 volume, 1000 angles, and a 1000-by-1000 detector. Likewise, problem (b) has a 50^3 volume, 50 angles, and a 50-by-50 detector. Problem (c) is a reconstruction from a severe sparse-angular geometry ($N_\theta := 3$), and problem (d) is a 2D slice geometry ($N_z := 1$ and $N_v := 1$).

Figure 4.2(a)-(d) show that `directFPBP` and kernel execution in our package (blue segments) have virtually equivalent runtimes, confirming that we can use ASTRA Toolbox as a

baseline measurement. In (b), the smaller reconstruction, we find that geometry processing in Python is typically slower than in C++, but that in some cases ASTRA Toolbox projector initialization is somewhat slower (orange). This overhead is in many contexts negligible, as both frameworks allow the reuse of projectors. For algorithms that use a single backprojection, such as the filtered-backprojection or FDK (Section 4.5.2), a considerable amount of time may be spent in data transfers between the host (CPU) and device (GPU). With the ASTRA Toolbox algorithms API, Fig. 4.1(a), such transfers are unavoidable. However, using Tomosipo, Fig. 4.1(b) or KernelKit, data can be kept on the GPU.

Figure 4.2(c) shows that for a set-up with three angles, only a small amount of time is spent in the kernel execution, compared to the transfer of the resulting volume. In Figure 4.2(d) the geometry is reduced to a slice: in this case ASTRA Toolbox switches to dedicated kernel with $\bar{N}_z := 1$ (cf. the standard value of $\bar{N}_z := 6$ in Algorithm 1). Also in this scenario, we observe a runtime of KernelKit that is similar to the ASTRA Toolbox precompiled binary (blue segments). This suggests that runtime compilation does not lead to observably better or worse performing binary code. Lastly, the compilation of a kernel with CuPy (red) takes a few milliseconds. This is insignificant for most applications, since the kernel is cached after its first use. Yet, it can be of importance for kernel tuning (Section 4.5.3), where each kernel configuration requires a recompilation.

4.5 Case studies

In the upcoming sections, we will demonstrate KernelKit through practical examples. While we are aware of many, often quite complex, applications that could benefit from customized projectors, particularly in dynamic imaging and machine learning domains, we have chosen three studies that showcase KernelKit’s core advantages (Section 4.3.1) while being simple to describe and easy to interpret. The scenarios use the previously-introduced conebeam backprojector in reconstruction contexts that are inspired by experiments from our FleX-ray laboratory in Amsterdam [9] and the X-ray imaging set-up for fluidized beds at Delft University of Technology [46].

The implementations of our case studies require customized projectors, high-level CUDA features, or runtime-compiled kernels. They highlight the complexity of real-world scenarios, which often pose challenges that cannot be effectively addressed from the Python interface of ASTRA Toolbox alone. As an alternative to solving these in C++ library code, which would request significantly more development time from the average Python user, we will present tailored solutions with KernelKit. In each case study, we will compare the KernelKit solution to the most efficient use of ASTRA Toolbox, which is the direct projection approach illustrated in Fig. 4.1(b). This is also equivalent to Tomosipo [26].

For a fair comparison, we have ensured that KernelKit projectors can be configured to reproduce the performance of ASTRA Toolbox (Appendix 4.4). This allows us to explain and attribute performance gains to the enhancements that we describe. In the experiments, we will use ASTRA Toolbox 2.1.2 with CUDA Toolkit 11.7; KernelKit with CuPy 12.1 and CUDA Toolkit 12.1; PyTorch 2.1.0.dev20230821 with CUDA Toolkit 12.1; and a mod-

ified version of Kernel Tuner 0.3.0. All the experiments use Python 3.10 and the NVIDIA GA102GL [RTX A6000] architecture.

4.5.1 Patch-based neural network training

In this case study, we will configure a KernelKit projector for reconstruction of image *patches*, i.e., small 2D or 3D image regions. Such a projector is particularly useful for neural network training, where CNN parameters are optimized based on a large data set consisting of pairs of input and desired output. Training on patches that are, e.g., reconstructed from random locations in a 3D volume, offers several advantages over training on high-resolution inputs. For example, it can mitigate GPU memory exhaustion and reduce training time. This approach has proven successful in various image tasks such as super-resolution and denoising [133, 134].

To avoid slowing down the training process, a large quantity of patches must be readily available. When patches are only required as network inputs, one option would be to precompute and store a vast number of patches on disk, which can then be loaded back during training. This is, however, only a practical strategy for relatively small data sets. In dynamic CT, or when neural networks use reconstruction as a layer, on-the-fly generation of the patches from projection data in parallel to the training process can be faster and more resource-efficient. During online learning, generating patches ahead of the experiment may not be feasible, and patches must be generated in parallel to the neural network training process [86].

Since patches can be as small as 20-by-20 voxels, even small computational overheads in the projector have relatively large impact on the overall backprojection time. Recognizing that a projector is executed repetitively, while the geometry and data sizes remain constant, enable an alternative implementation that is better suited to this specific use case. In the following, we will show that we can identify and reduce two overheads, and tailor KernelKit conebeam backprojectors $\mathbf{A}^T$ (Algorithm 1) to improve upon the ASTRA Toolbox baseline (Fig. 4.1(b)).

We will use Figure 4.3 to display timings from the implementations on example data. The data consists of $N_\theta := 1000$ images of an $(N, 1)$ resp. (N, N) -sized detector in the 2D, resp. 3D, case. The input size is varied from $N \in \{50, 100, \dots, 500\}$ for a 2D patch, and $N \in \{20, 30, \dots, 80\}$ for a small 3D volumetric input. We implement each backprojection algorithm as a PyTorch neural network layer, and time the execution of the forward pass through the layer. In all ASTRA Toolbox and KernelKit implementations, geometry calculations are not part of any timings, as they can be avoided by recycling an existing projector object (i.e., the orange segments of Figure 4.2 in Appendix 4.4).

ASTRA Toolbox We first show baseline implementations of ASTRA Toolbox (Fig. 4.1(a) and (b)), i.e., the ASTRA 3D backprojection algorithm, and the ASTRA Toolbox approach taken by Tomosipo. In the former, data is kept in host memory and passed back-and-forth to the GPU, while in the latter, data is kept on the GPU using **GPULink**. Figure 4.3 shows the results with green and black squares, respectively. The results emphasize that a careful

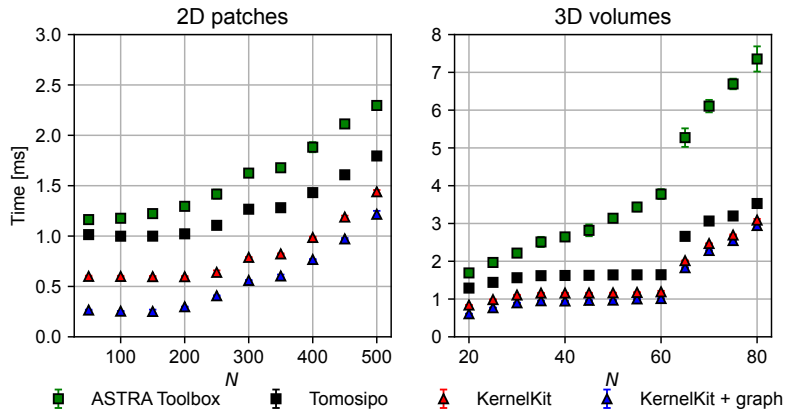


Figure 4.3: Conebeam backprojection PyTorch layers for a $(N, N, 1)$ -sized input (left) and (N, N, N) -sized volume (right), using $N_\theta := 1000$ projections of an $(N, 1)$ and (N, N) -sized detector. 2D timings use 2000 samples, and 3D timings 350 samples. Both cases use GPU warm up with an equivalent amount of burn-in samples.

integration between ASTRA Toolbox and external software is necessary to obtain good performance. When, for example, a neural network layer follows after the backprojection, e.g., a PyTorch convolution on the GPU, an unnecessary “ping-pong” occurs by a device-host-device transfer. We note that, like Tomosipo, KernelKit retains volumes and projections on the GPU, as transfers are explicit commands in the CuPy framework.

KernelKit A first KernelKit implementation, denoted by red triangles in Figure 4.3, shows the timings of a PyTorch layer based on a `BackProjector` instance. In this reconstruction context, with a fixed geometry, it is advantageous to instruct the KernelKit projector to recycle as many memory operations as possible. Firstly, previously-created CUDA textures can be recycled, meaning that they can be overwritten with new projection data when the dimensions of the reconstruction problem are unchanged. This eliminates memory reallocation and the creation of new texture objects. More precisely, in Algorithm 1 line 2, the object $\tilde{y}$ is not destroyed or recreated on every projector invocation. While this would generally come at the disadvantage of occupying additional memory in-between projector calls, this does not pose a problem due to the small memory size of patches. Secondly, a re-transfer of geometry parameters from global GPU memory into *constant memory*, i.e., a fast read-only memory that is shared by all threads, can be avoided. This is about 47 kilobyte for 1,000 angles, approximately the size of a 100-by-100 patch. Fig. 4.3 shows that these optimizations result in a consistent decrease in reconstruction time of about 0.3 ms per patch.

KernelKit + CUDA graphs In the left part of Figure 4.3, we note that at the smaller dimensions, $N < 250$, reconstruction times do not decrease further towards zero, in both KernelKit and ASTRA Toolbox. This is an indication that a part of the computation is not

yet fully explained by the cost of backprojection in the first implementation. A recent CUDA feature, called *CUDA Graphs*, provides a mechanism to launch multiple CUDA kernels with a single CPU operation. This reduces the overall launch time of kernels, and can be used to avoid the recomputation of kernel arguments on the CPU side. Using the integration between CuPy and PyTorch, we construct a CUDA Graph in PyTorch (i.e., a PyTorch Graph) by capturing all kernel launches after a warm-up iteration of the neural network. In a next iteration, with unchanged geometries, the graph replays all the kernel launches using the same parameters, but with the newly-placed data in memory. Figure 4.3 shows, with the blue triangles, that the CUDA graphs are able to remove some of the overhead, and reduce the backprojection time to approximately 0.25 ms. We conclude that patches can be generated about four times faster than the optimal implementation with ASTRA Toolbox, which does not allow an integration with CUDA Graphs due to CUDA data structures not being able to pass through the language barrier. The results from this use case can be combined with kernel customization, which we discuss in Section 4.5.3, to achieve even faster patch generation.

4.5.2 Real-time X-ray tomography

Reconstruction software generally presumes a common workflow where a set of projection images is provided by the user and where the desired output is a single reconstruction volume. Algorithms for less conventional contexts, e.g., with multiple output volumes, or with a subset of projection images as input [26], often require modifications to the low-level algorithm primitives in the software to be implemented in the most efficient way. In this section, we will demonstrate an example from *real-time tomography*, where changing the backprojector in the FDK algorithm (Eq. 1.6) improves the efficiency of reconstruction. Real-time tomography is used to visualize dynamically evolving processes in synchrotron light source facilities and X-ray microscopy laboratories [9, 34, 90]. Live feedback, provided by the reconstructions, simplifies steering of the experiment as well as the on-line optimization of acquisition settings [93]. With that it prevents experimental repetition and therefore saves valuable experiment time.

In real-time tomography, the processes from data acquisition to reconstruction are streamlined with *tomographic pipelines* [92]. Pipelines consist of sequential software components, which can be distributed over multiple machines or GPUs. After data is measured at the detector, it is first preprocessed, e.g., with dark-field and flat-field corrections, and linearization according to Beer-Lambert’s law [5]. For FDK, a filtering step is subsequently performed (i.e., the convolution $y \otimes \mathbf{f}$ in Eq. 1.6), and the final step is to apply the conebeam backprojector to the filtered data in order to reconstruct the object (Algorithm 1). The duration of data processing tasks is linear in the size of the projection data $\mathcal{O}(N_\theta N_u N_v)$, whereas reconstruction is $\mathcal{O}(N_x N_y N_z N_\theta)$ using our voxel-driven conebeam kernel, or, alternatively, $\mathcal{O}(N_x N_y N_z \log N_\theta)$ in the case of GridRec [131]. As each pipeline component processes an individual chunk of the pipeline data asynchronously, the visualization framerate follows the bottleneck component, which is often the backprojection. The performance of the real-time

FDK in a tomographic pipeline therefore depends on the GPU-accelerated backprojection component.

Although modern GPU technology has pushed the boundaries of what is achievable in parallel computing, high-resolution fully-3D reconstruction and visualization in milliseconds is still out of reach [12, 130]. While reducing the spatial resolution or visualization framerate would render the problem feasible, it is often desirable to reconstruct at the potential resolution of the set-up, i.e., such as defined by the detector resolution and exposure time, in order to follow the imaged physics. One successful approach to do so is the *quasi-3D reconstruction*, which was first introduced in the real-time reconstruction pipeline called RECAST3D [12]. The principle of a quasi-3D reconstruction is that only a few user-selected cross-sectional slices through the volume need to be reconstructed (see Figure 3.2 in chapter 2). The RECAST3D graphical user interface enables interaction with the slices, and allows for a fast interrogation of the object during the experiment. The FDK algorithm is especially well-suited in this situation, as it enables a region-of-interest reconstruction in a time that is linearly related to the number of voxels in the region. RECAST3D has been used for real-time alignment [92], explorative imaging [9], and visualization of experiments with quickly evolving dynamics [93].

A straightforward quasi-3D implementation with ASTRA Toolbox or Tomosipo, using the approach in Fig. 4.1(b), is to reconstruct the three (orthogonal) cross-sectional slices individually, and send each of the slices sequentially to the RECAST3D user interface. This requires three ASTRA Toolbox projectors, each configured to handle one slice of the quasi-3D reconstruction. However, this approach leads to unnecessary repetitions in converting projections to CUDA texture objects ($\tilde{y}$ on line 2 of Algorithm 1), which is required for fast interpolation in the sinogram (Section 4.2.1). With KernelKit projectors, on the other hand, users can capitalize on the fact that the same projections are used in each projector. First, a single texture object $\tilde{y}$ can be constructed, and kept in memory for as long as the projector is needed. This avoids memory reallocation, similar to Section 4.5.1. Then, $\tilde{y}$ can be shared with the three KernelKit projectors. As a result, only a single $\tilde{y}$ has to be updated on incoming data to RECAST3D, saving memory and computation time. The optimizations have the potential to significantly lower the quasi-3D algorithm runtime, as slice reconstructions, with complexity $\mathcal{O}(N_x N_y N_\theta)$, have the same order of complexity as data tasks, $\mathcal{O}(N_\theta N_u N_v)$.

We will compare the KernelKit projectors to the ASTRA Toolbox approach on three set-ups that are used for dynamic CT in the scientific literature. The first is our FleX-ray laboratory micro-CT scanner at CWI [9]. The second is a recent high-speed rotational set-up that achieves a half-rotation reconstruction every 10 ms [135], and the third used a high-resolution detector to study the rheology of liquid foams at the SLS TOMCAT beamline [17, 122]. We will time the KernelKit projector using an average of 400 quasi-3D reconstructions after 400 warm-up samples. Note that, in comparison to Section 4.5.1, in this section we consider the geometries (i.e., the slices) to be variable, and require the projection data to remain constant during the reconstruction of the slices. In Section 4.5.1, the geometries were

instead constant, and the projections were variable.

Table 4.1 lists the framerates that are achievable with quasi-3D reconstruction for the three set-ups. We note that for the FleX-ray micro-CT set-up, the ASTRA Toolbox baseline already achieves a quasi-3D backprojection framerate of 4.8 ms, which is faster than the framerate of the detector (12 ms). For the High-speed set-up, a reconstruction can be obtained once every 3.5 half-rotations (180°), and about one time per half-rotation for the Tomobank set-up, although in the latter case the framerate of 3 Hz is comparatively slow. Compared to ASTRA Toolbox, the optimizations with KernelKit lead to a factor 8–18 speed-up. In summary, Table 4.1 shows that customization of quasi-3D backprojection can lead to significantly faster visualization framerates and enables a reconstruction in each half-rotation in three set-ups. Yet, reconstructing at the framerates of modern detectors, for example for the purpose of automation, remains an open challenge.

	FleX-ray [9]	High-speed [135]	Tomobank [17, 122]
Detector framerate	12 ms (83 Hz)	78.125 μ s (12.8 kHz)	0.8 ms (1.3 kHz)
Detector resolution	400 $\times$ 600	1024 $\times$ 1024	2016 $\times$ 1800
Slice resolution	400 $\times$ 400	1024 $\times$ 1024	2016 $\times$ 2016
Projections per 180°	75	128	300
180° rotation time	900 ms (1.1 Hz)	10 ms (100 Hz)	210 ms (4.8 Hz)
Toolbox	4.8 ms (208 Hz)	35.8 ms (28 Hz)	291.7 ms (3 Hz)
KernelKit	0.6 ms (1.6 kHz)	1.8 ms (0.6 kHz)	15.7 ms (64 Hz)

Table 4.1: Quasi-3D backprojection for three simulated set-up configurations. Preprocessing, uploading or filtering are not part of any timings.

4.5.3 Kernel optimization using Kernel Tuner

In the last use case, we will leverage KernelKit’s ability to runtime compile kernels with CuPy and the NVIDIA runtime compiler (NVRTC), to search over CUDA parameters and kernel implementations. This is termed *kernel tuning*, and entails maximizing the performance of GPU computing, by optimizing free parameters of kernels, such as block sizes and algorithmic constants [124]. Tuned algorithms achieve better runtimes, reduced energy consumption [27, 28], or utilize less resources, in particular GPU memory. In high-throughput applications, such as in-line CT scanning, a kernel can be tuned towards a fixed measurement protocol and dedicated GPU architecture. In these situations, even a slight improvement can lead to significant energy savings over the equipment’s lifetime.

The efficiency of backprojection depends, in the first place, on the GPU architecture and the dimensions of the reconstruction problem (i.e., N_x , N_y , N_z , N_u , N_v and N_θ). Natural targets for tuning are the CUDA *thread block sizes*, i.e., free parameters that define the number of threads grouped together within a single thread block for parallel execution on the GPU. This is constrained by the capability of the CUDA architecture; commonly 1024 threads per thread block, and set to sensible defaults in ASTRA Toolbox. Another target for tuning is the backprojection code itself, known as *software tuning*. In Algorithm 1, for

example, the compiled $\bar{N}_z$ and $\bar{N}_\theta$ constants, which define how the problem is chunked, can be made variable again through the process of recompilation. In our software, CUDA/C++ kernel code is parameterized through the Jinja2 templating engine [126]. In this way, different code paths can be explored through recompilation at the program runtime.

To demonstrate the potential of kernel tuning, we demonstrate three tuning results, using KernelKit in conjunction with the Kernel Tuner software package [136]. We utilize the KernelKit projector in such a way that the sinogram and volume are retained on the GPU, allowing millisecond kernel recompilation and testing for each point in the parameter search space. Kernel tuning with ASTRA Toolbox or Tomosipo would require a scripted recompilation of ASTRA Toolbox, which would be significantly slower. Moreover, such alternative would have less flexibility in exploring templated code paths, as ASTRA Toolbox is built around fixed axis and geometry conventions. We furthermore note that our results are specific for the NVIDIA RTX A6000 architecture, and that these results do not necessarily generalize to reconstruction problems of different dimensions, e.g., with low or high numbers of angles or non-standard volume geometries.

Figure 4.4 displays the result of a brute-force search over all possible CUDA thread block sizes for the reconstruction of a large, 2000-by-2000 voxels, slice. The projection data has dimensions $N_\theta := 32$, $N_u := 2000$, $N_v := 2000$. We picked $N_\theta = \bar{N}_\theta$ to time a single kernel launch of Algorithm 1. We warm-start the GPU for every configuration with 50 samples, and average over 100 subsequent samples. For the timings, we launch CUDA graphs to eliminate CPU overhead, and eliminate the cost of all data transfers by preloading data onto the GPU. The search takes about one hour to complete, and finds a minimum at (16, 4). The associated runtime of 0.397 ms is about 8% faster than that of the uninformed standard choice of (16, 32), which yielded 0.431 ms. We confirm that this is a valid optimum by repeating the (16, 4) and (16, 32) configurations several times with 100 samples. We did, however, note a slow decrease of the speed-up with increased numbers of averages. Yet, after 40,000 averages, the new optimum nevertheless leads to a stable improvement of 3% over the uninformed default choice. Such an excessive load may, however, not be representative of real-world usage of the kernel.

To demonstrate searching over implementations, we parameterize the conebeam kernel to allow backprojection from four different texture memory back-ends (cf. line 1 in Algorithm 1), which are termed *resources* in the CUDA specification [123]. The first option is a texture object with a *3D CUDA Array* (the default used by ASTRA Toolbox, see Section 4.2.1). A 3D CUDA Array can be used with any axis order, and can therefore avoid an in-memory transposition of the data. The second option uses a *Layered CUDA Array*, which is optimized for spatial look-up in the second and third dimension, and is often more efficient when N_θ is the major array axis. The third option uses a list of N_θ two-dimensional texture objects, rather than a single texture. Compared to a layered texture, this enables a partial update of projections in-between kernel invocations, for example in a dynamic imaging scenario with a moving timewindow of projections. The last option uses N_θ texture objects backed up by linear memory, which avoids the creation of a CUDA Array. Here, the projections are stored

in *pitched* arrays, meaning that the minor array axis is padded to a multiple of 32 for faster look-up.

To compare the four options, Figure 4.5 runs Kernel Tuner for increasingly larger reconstruction problems. Figure 4.5(a) displays backprojection with an *initialization* of textures, and Figure 4.5(b) a backprojection with an *update* to existing projections. Figure 4.5(a) shows that, for algorithms that require a single invocation of $\mathbf{A}^T$, such as the FDK algorithm, layered CUDA arrays yield the best result, thanks to their fast initialization. Figure 4.5(b) shows that linear memory is only marginally slower than layered arrays, and that their main disadvantage originates from the slow initialization of N_θ texture objects. Textures that use pitched linear memory are therefore an alternative to CUDA Arrays when an algorithm requires write access to y , or when GPU memory is scarce.

Kernel tuning holds a large potential for in-line and industrial CT, and particularly for scientific imaging equipment, as optimized CUDA kernels can improve algorithm runtimes or reduce energy costs. In [46], a set-up for ultrafast imaging of bubbling fluidized beds was introduced at *Delft University of Technology*. The set-up consists of three stationary X-ray sources and flat panel detectors. In this last example, the detectors operate in (300, 1548)-pixels regions of interest at 65 Hz. Each 3-tuple of frames provides the sparse-angular projection data that is necessary to compute a (300, 300, 1200)-sized reconstruction volume. To find an optimal backprojection kernel, we use the bruteforce search strategy in Kernel Tuner with a search space consisting of texture options and the $\bar{N}_z$ parameter (Algorithm 1). Kernel Tuner finds that a configuration consisting of layered CUDA arrays and $\bar{N}_z := 2$ is optimal, and that this improves the runtime from 3.90 ms (ASTRA Toolbox default parameters) to 2.93 ms, an improvement of 25%. Using the found parameters, an optimization over block size multiples of 8 finds that (152, 1) further improves the runtime to 2.18 ms, a 44% improvement compared to the defaults. As fluidized bed experiments comprise several minutes of experimentation, and may contain several thousands of timeframes, tuned kernels realistically improve the efficiency and costs of reconstructing bubbling fluidized beds.

We note that there are many facets of kernel tuning for X-ray CT that we have not explored in this article. For large reconstruction problems, for example, searching over the parameter space takes increasingly more time, and a bruteforce search strategy may therefore not be feasible. Timing a single kernel that is representative of the entire reconstruction problem may then be able to provide a solution. Another topic of further research is to find small search spaces that can be explored quickly before the start of an iterative algorithm. This could already trade off in a faster runtime, even when the algorithm is ran once. In a follow-up study we will explore the geometry-dependence of the kernels as well as the application of the different optimization strategies in Kernel Tuner to X-ray CT.

4.6 Conclusion

ASTRA KernelKit is an all-Python CT reconstruction package that leverages the ASTRA Toolbox CUDA kernels using CuPy. KernelKit is written for user-customizable kernels, projectors and algorithms, and enables rapid prototyping of data-driven algorithms using the Python

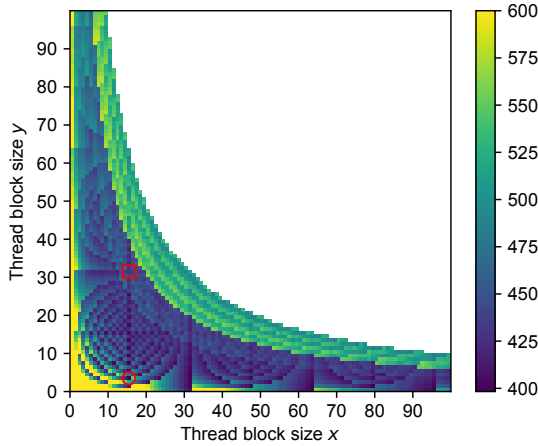


Figure 4.4: Conebeam backprojection time in microseconds for different CUDA thread block sizes on an NVIDIA GA102GL [RTX A6000] for a 2000-by-2000 slice and 32 projection angles of a 2000-by-2000 detector. The ASTRA Toolbox default thread block sizes, (16, 32), are denoted by the red square. The optimum (16, 4) is denoted by the red circle. The graphic is restricted to block sizes smaller than 100 for the purpose of visualization.

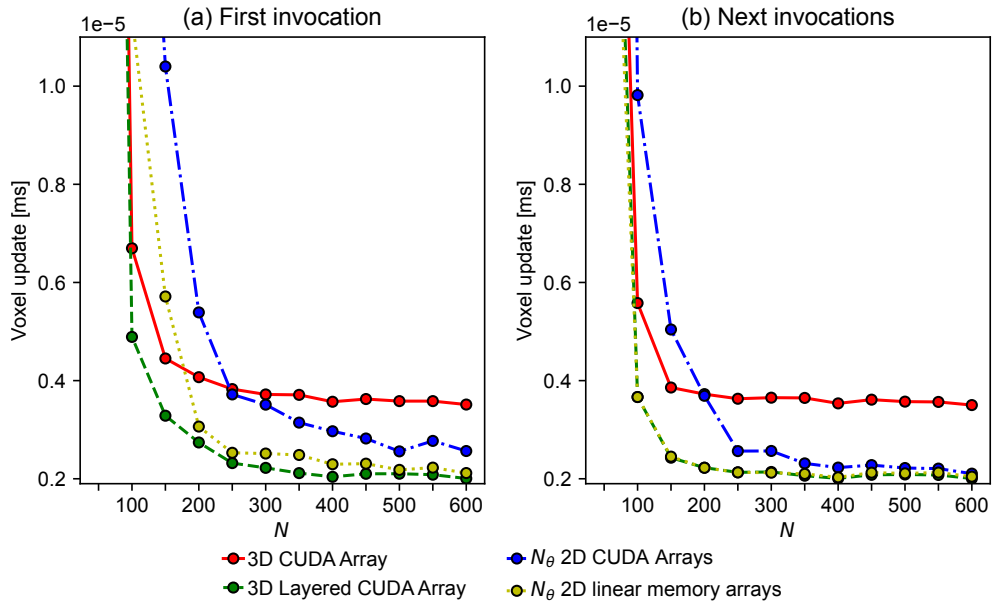


Figure 4.5: Time per *voxel update* of $\mathbf{A}^T$ on an (N, N, N) -sized volume using an NVIDIA GA102GL [RTX A6000]. (a) includes texture initialization on the first call to $\mathbf{A}^T$. (b) updates existing textures in any subsequent calls with new data of the same dimensions. Projections are $N_\theta := N$ angles of an (N, N) detector. A voxel update time is the backprojection time divided by $N_x N_y N_z N_\theta$.

ecosystem and philosophy. We envision it to serve as a minimalist back-end for high-level frameworks, as a package to develop projectors, and as a tool for high-performance applications that benefit from tuned algorithms. In this work we have focused on the voxel-driven conebeam backprojector, and demonstrated through patch-based learning, a tailored real-time algorithm, and runtime kernel compilation, that the Python ecosystem the software can now be used to implement efficient tomographic algorithms in several real-world use cases. In future work, we aim to extend the framework with new algorithm- and geometry-specific projectors [7], such as a matched forward- and backprojector, or kernels for curved detectors, for which the same principles apply. All in all, our package aims to accelerate the exploration and development of new high-performance and data-driven algorithms in CT.

4