On the optimization of imaging pipelines

Schoonhoven, R.A.

1 Introduction

Today's industrial and scientific processes are powered by large amounts of computing power in the form of complex hardware systems running computational pipelines. These pipelines process vast amounts of data, sometimes in real time. Often such processing pipelines consist of several different algorithms connected end-to-end. Furthermore, computational pipelines have different levels of implementation, as illustrated in Figure 1.1. On one hand, there's the conceptual level: the design of the algorithm and the adjustable parameters that influence its accuracy. Second, there is the software level of the algorithms and how computationally efficient they are implemented. Third, there is the hardware aspect of efficiently configuring the different hardware systems that run algorithms. Altogether, such pipelines come with challenges when it comes to extracting better accuracy, running the computations at higher speed, or bringing down energy and environmental costs.

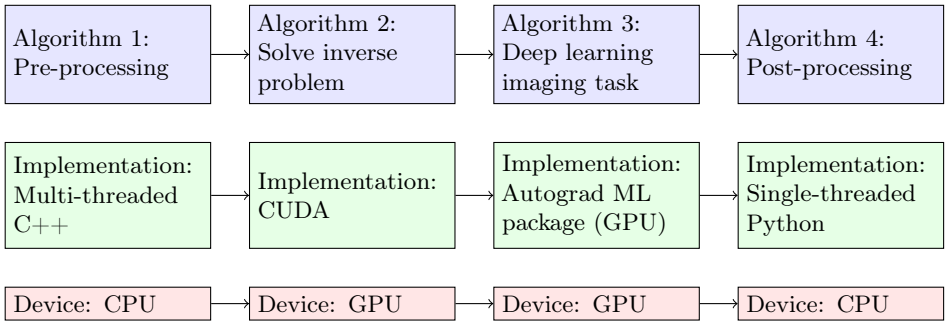


Figure 1.1: A schematic of an example imaging pipeline illustrating different implementation levels.

Addressing these challenges requires a multi-faceted approach. First, these pipelines often consist of several algorithms, each with parameters that typically remain fixed or are manually set. Optimizing these parameters can lead to improved output quality. Second, it may be possible to accelerate algorithms while minimizing the reduction in output quality. If the speed-up is significant, it can enable previously unfeasible applications. Third, equipping the computer systems with the right hardware is crucial, and efficiently running algorithms on this hardware presents its own set of challenges. An added layer of complexity is that hardware efficiency should encompass not just computation speed but also energy efficiency, especially in light of growing environmental concerns. Lastly, the most efficient optimization can come from an end-to-end approach rather than analyzing each computational block in isolation. Enabling end-to-end optimization may require integration with advanced software that, for example, propagates gradients through computational blocks.

This thesis is based on a range of results in optimizing computational imaging pipelines at different levels of implementation. Jointly, these results encompass a multi-faceted perspective on pipeline optimization. Imaging pipelines that are used for X-ray CT serve as a leading application topic for many results in this thesis, but different computational tasks such as radio astronomy are also considered. In this introductory chapter, we first introduce X-ray CT as an application field in Section 1.1. Next, we introduce deep learning techniques for computational imaging tasks in Section 1.2. Lastly, we introduce the basics of Graphics Processing Units (GPUs) and GPU programming in Section 1.3.

1.1 X-ray CT pipelines

1.1.1 Introduction to tomography

Tomographic imaging [106] is a technique for examining the internal structure of objects using penetrating radiation such as X-rays or electron beams. This process involves acquiring projection images from various angles to subsequently compute a 3D image of the object's internal structure (see Figure 1.2). In most laboratory CT setups an object is placed on a rotation table, and a source illuminates an object with an X-ray cone beam. The beam is attenuated by the material along its path, and the beam intensity, after passing through the object, is measured on a flat panel detector. The detector counts the photons hitting the detector for a certain exposure time. The image acquired by the detector in this manner is called a projection. During a CT scan, projections are acquired for different angles by rotating the object on the rotation table.

The raw projections represent the intensity of X-rays after passing through the object. To represent the 3D interior of an object we are interested in its density and composition, which are directly related to the object's attenuation coefficients. These attenuation coefficients are reconstructed during the tomographic reconstruction process.

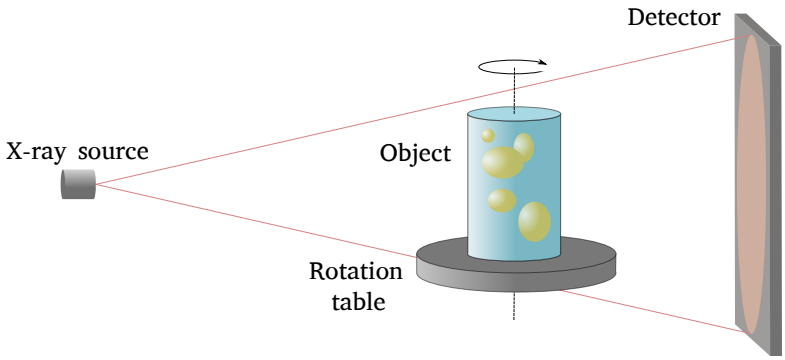


Figure 1.2: Example laboratory X-ray cone beam CT setup.

Mathematically, the attenuation function can be expressed as a continuous function $u : \Omega \to \mathbb{R}$ on a bounded set $\Omega \subset \mathbb{R}^3$. For a given ray $L_{\mathbf{x}}$ passing through a point $\mathbf{x} \in \Omega$, the detected X-ray intensity $I(\mathbf{x})$ is related to the object's attenuation coefficients according to the Beer-Lambert law [106]

$$I(\mathbf{x}) = I_0 \exp\left(-\int_{L_{\mathbf{x}}} u(\mathbf{z})\, d\mathbf{z}\right), \tag{1.1}$$

where I_0 is the intensity at the X-ray source. To extract the attenuation coefficients from the raw projections, a log-transformation of the raw projections is performed in the pre-processing phase

$$\int_{L_{\mathbf{x}}} u(\mathbf{z})\, d\mathbf{z} = -\log\left(\frac{I(\mathbf{x})}{I_0}\right).$$

After pre-processing, a tomographic reconstruction algorithm computes a 3D volume of the object using the geometry of the setup and the acquired projections.

1.1.2 Tomography as an inverse problem

The tomographic reconstruction problem is to recover an object's volume from a series of its projections. In the discrete setting, it can be modeled as the problem to recover a volume $\mathbf{x} \in \mathbb{R}^{N_x \times N_y \times N_z}$ from the measured projection data $\mathbf{y} \in \mathbb{R}^{N_\theta \times N_u \times N_v}$. Here, N_θ is the number of projection angles, and N_u and N_v are the number of detector rows and columns. The process of passing X-rays through an object from various angles, and capturing the resulting 2D images on a detector is called projection. The projection process can be approximated by a linear operator A, and the tomographic reconstruction problem can be modeled as an inverse problem

$$A\mathbf{x} = \mathbf{y}. \tag{1.2}$$

To provide solutions to the inverse problem we run tomographic reconstruction algorithms to compute $\mathbf{x}$ for measured projection data $\mathbf{y}$. A widely used reconstruction algorithm is filtered backprojection (FBP)

$$\mathbf{x}_{\text{FBP}} = A^T(\mathbf{h} * \mathbf{y}). \tag{1.3}$$

Here, $\mathbf{h} \in \mathbb{R}^{N_v}$ is a 1D filter that is convolved with the projection data to amplify the high spatial frequencies. For example, the commonly used Ram-Lak filter is essentially a ramp filter in the frequency domain that linearly weights the frequencies. In the context of circular cone-beam tomography, a variant of FBP the Feldkamp-Davis-Kress (FDK) [59] algorithm is used

$$\mathbf{x}_{\text{FDK}} = A^T(\mathbf{h} * \tilde{\mathbf{y}}). \tag{1.4}$$

Here $\tilde{\mathbf{y}}$ denotes weighted projection data, adjusting for diminishing intensity further from the detector center.

Alternative algorithms are iterative algorithms that formulate equation 1.2 as a minimization problem to iteratively reduce the difference between the estimated projections and the measured projection data

$$\mathbf{x}^* = \underset{\mathbf{x}\in\mathbb{R}^{N_x\times N_y\times N_z}}{\arg\min} \|A\mathbf{x} - \mathbf{y}\|_2^2. \tag{1.5}$$

Moreover, variational methods incorporate additional prior knowledge via a regularization term $R(x)$

$$\mathbf{x}^* = \underset{\mathbf{x}\in\mathbb{R}^{N_x\times N_y\times N_z}}{\arg\min} \|A\mathbf{x} - \mathbf{y}\|_2^2 + R(\mathbf{x}). \tag{1.6}$$

These methods can enhance reconstruction accuracy when working with for example limited or noisy projection data.

A commonly used variational method is total variation reconstruction where we add prior information about the gradient of the image being sparse by adding a total variation term [14]

$$\|A\mathbf{x} - \mathbf{y}\|_2^2 + \lambda\|\nabla\mathbf{x}\|_1. \tag{1.7}$$

The trade-off between prior knowledge and data fidelity is regulated by the parameter λ, which has to be set by the user.

1.1.3 Computational pipeline

Acquiring a CT reconstruction of an object requires several processing steps. These steps can contain parameters that are potentially optimizable or learnable. To illustrate, an example FDK reconstruction pipeline is presented in Figure 1.3.

1. First, there can be inconsistencies or non-uniformities in the X-ray beam and detector response. To correct these, an image with no object is taken called

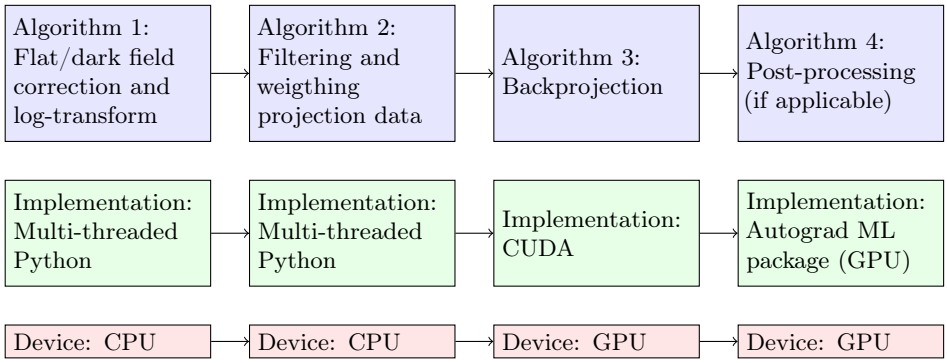


Figure 1.3: A schematic illustrating different implementation levels of an example CT imaging pipeline using FDK reconstruction.

the flat-field image. Another image, called the dark-field image, is taken with the X-ray source turned off. It captures the electronic background of the detector and the after-glow of the detector sensor material. As mentioned, a log-transformation is applied to the raw projection data in the pre-processing step. The full pre-processing equation is

$$\mathbf{y}_{\text{corrected}} = -\log\left(\frac{\mathbf{y}_{\text{raw}} - \mathbf{y}_{\text{dark-field}}}{\mathbf{y}_{\text{flat-field}} - \mathbf{y}_{\text{dark-field}}}\right).$$

In the example pipeline of Figure 1.3 the projections are for example loaded as NumPy arrays, and the equation is computed on the CPU using multi-threaded Python code.

Learnable parameters: None. However, in this stage for example beam hardening correction could be performed in which case the beam spectrum, and the energy-dependent attenuation coefficients could be learned.

2. Second, as explained in the previous section, the first step of FDK is to apply a weighting depending on the distance to the center of the beam and convolve the result with a 1D filter. In this example, both steps are still performed on the CPU with multi-threaded Python code. However, convolution is an operation that can efficiently be performed on the GPU so it may be more efficient to run this step on the GPU.

   Learnable parameters: Filter coefficients.

3. Third, the filtered projection data is backprojected to compute the 3D reconstruction of the object. For backprojection, we can use efficient high-performance libraries such as the ASTRA toolbox [1]. In the example pipeline, ASTRA is run on the GPU (CUDA is a GPU programming language for Nvidia GPUs) to efficiently perform the backprojection.

Learnable parameters: None. However, if a different reconstruction methods was used, e.g., total variation reconstruction, then the regularization parameter could be learned.

4. Fourth, many imaging tasks require a post-processing step to be performed after reconstruction. In this example, a post-processing step using neural networks is performed that creates a segmented volume, or denoises the reconstruction to improve the quality.

   Learnable parameters: Neural network parameters.

In addition to the steps outlined in Figure 1.3, CT pipelines often contain processing steps such as ring artifact reduction, beam hardening correction, and rotation axis alignment (beam hardening correction and rotation axis alignment are considered in Chapter 3). Efficient CT pipelines need to consider not only the different processing steps (conceptual level) but also the implementation and hardware levels.

1.2 Deep learning for imaging

1.2.1 Machine learning for imaging pipelines

Machine learning involves training algorithms to recognize patterns and output predictions based on data [17]. A machine learning model typically has a set of parameters Θ. For supervised learning, the training process attempts to adjust the parameters to minimize the difference between the model predictions, and the true values. For given Θ, the machine learning model defines a function $F_\Theta : \mathcal{X} \to \mathcal{Y}$ for input data space $\mathcal{X}$ and desired output data space $\mathcal{Y}$. Digital images are typically represented as arrays of pixel values $\mathbb{R}^{c_{in} \times m \times n}$. The number of channels c_{in} signifies the number of input channels an image possesses. Typically, natural images have $c_{in} = 3$ corresponding to RGB channels, while X-ray CT images have $c_{in} = 1$. However, spectral CT images might have as many input channels as there are spectral bins.

A machine learning model for imaging typically has input space $\mathcal{X} = \mathbb{R}^{c_{in} \times m \times n}$. The shape of the output $\mathcal{Y}$ is dependent on the specific imaging task. For instance:

- In a 1-D regression task, $\mathcal{Y} = \mathbb{R}$.
- For classification tasks with k classes, $\mathcal{Y} = \mathbb{R}^k$. In this case, the machine learning model computes a probability vector with predictions for each class. The highest probability class is then chosen from the network output to obtain a final prediction.
- For image denoising we have $\mathcal{Y} = \mathbb{R}^{c_{in} \times m \times n}$ for an image of size $m \times n$.
- For segmentation tasks, we have $\mathcal{Y} = \mathbb{R}^{k \times m \times n}$ for k classes. Here, a probability vector with predictions for each class and each pixel is computed.

1.2.2 Convolutional neural networks

In recent years, convolutional neural networks (CNNs) have been state-of-the-art machine learning algorithms for solving many imaging tasks [116, 121, 225]. A CNN is a machine learning model where layers of convolutional filters are arranged hierarchically, allowing for the extraction of features from image data [68]. A CNN consists of consecutive layers of operations that sequentially process image data. Each operation, such as convolution, has an input $\mathbf{x}$, and an output $\mathbf{y}$. These inputs and outputs are multi-dimensional arrays that consist of one or more images, referred to as channels. For example, let's consider a convolutional operation with input channels $\mathbf{x}_1, \ldots, \mathbf{x}_N$. An output channel, $\mathbf{y}_j$, is computed by convolving the input channels with corresponding learned filters:

$$\mathbf{y}_j = \left(\sum_{i=1}^{N} h_{ij} * \mathbf{x}_i \right) + b_j.$$

In this equation, $h_{ij} \in \Theta$ represents the filter related to the convolution operator that acts between channels $\mathbf{x}_i$ and $\mathbf{y}_j$, and $b_j \in \Theta$ is an additive bias term. The specific sequence and interconnections of operations in a CNN are dependent on the chosen network architecture [136, 196].

The weights Θ are optimized during the training phase, where input samples $\mathbf{x}_1, \ldots, \mathbf{x}_N$ are processed by the network. In supervised learning, the output of the network is compared against known desired output samples $\mathbf{y}_1, \ldots, \mathbf{y}_N$. An appropriate loss function denoted as $\mathcal{J} : \mathcal{Y} \times \mathcal{Y} \to \mathbb{R}$, quantifies the error between the network's predictions and the desired samples (for instance, mean squared error loss). The objective of the training phase is to determine a Θ that minimizes this loss over the training data:

$$\Theta^* = \arg\min_{\Theta} \left\{ \sum_{i=1}^{N} \mathcal{J}(F_\Theta(\mathbf{x}_i), \mathbf{y}_i) \right\}.$$

The partial derivatives of $\mathcal{J}$ with respect to the weights can be computed through backpropagation [68]. The weights can then be updated using a gradient-based optimization strategy such as Adam [113].

The purpose of the training phase is to tune the CNN to produce accurate predictions for a certain distribution of images. However, it's possible that during the training phase, while Θ is optimized to yield the correct results for the given input samples $\mathbf{x}_i$, it may not generalize effectively to unseen samples. This process is called overfitting. To combat this, a set of samples independent of the training set can be created, called the validation set. In that case, the optimization step of the training phase remains the same, but the final objective becomes to determine Θ that minimizes the loss over the validation data.

1.2.3 End-to-end learning

End-to-end learning refers to the process of optimizing all components of a computational pipeline jointly to achieve a specific objective [11, 96, 175]. Instead of optimizing each component of the pipeline separately, this approach tunes the entire system together to better optimize for the final objective. Consider a pipeline $g = f_{\theta_n}^{[n]} \circ \cdots \circ f_{\theta_2}^{[2]} \circ f_{\theta_1}^{[1]}$ defined by functions $f_{\theta_i}^{[i]}$ parametrized by parameters θ_i. For some loss function $\mathcal{J}$, traditionally this could involve solving separate optimization problems $\arg\min_{\theta_i} \left\{ \sum_{j=1}^{N} \mathcal{J}(g(\mathbf{x}_j)) \right\}$ for input data $\{\mathbf{x}_j\}_{j=1,\ldots,N}$. End-to-end learning instead solves one joint optimization problem over all parameters

$$\underset{\theta_1,\ldots,\theta_n}{\arg\min} \left\{ \sum_{j=1}^{N} \mathcal{J}(g(\mathbf{x}_j)) \right\}.$$

In general, this can be a challenging optimization problem. Neural networks, with their multilayer architectures, are inherently well-suited for this kind of end-to-end optimization [204]. This is due to techniques that propagate gradients end-to-end in neural networks, which allows for gradient-based optimization strategies to be used.

Neural networks are commonly trained using backpropagation, a method that employs the chain rule to compute partial derivatives of the loss function with respect to the model parameters, beginning from the last layer. To facilitate this, neural networks are often implemented in auto-differentiation frameworks such as PyTorch [181], TensorFlow [2], or JAX [20]. When running code in an auto-differentiation framework, a program is decomposed into a series of primitive operations for which predefined procedures exist to compute derivatives. These series of computations are structured into a computational graph. In many auto-differentiation frameworks, like PyTorch, the computational graph is implicitly traced during the forward pass through the program. This graph is then utilized in the backward pass (when backpropagation is performed) to efficiently compute gradients for the model parameters.

End-to-end optimization, aided by auto-differentiation frameworks, is not limited to deep learning techniques. Exposing traditional algorithms to auto-differentiation frameworks can lead to numerous benefits, such as allowing for efficient automatic optimization of parameters that may otherwise be chosen manually, seamless compatibility with deep learning, and potentially improved accuracy of the optimized computational pipeline with respect to a final objective.

1.3 Graphics Processing Units

Graphics Processing Units, or GPUs, originally emerged as specialized hardware to accelerate graphics rendering tasks. Their design consists of a large number of simple, data-parallel processors which makes them an attractive option for computations that are parallel in nature. While CPUs typically have a handful to

dozens of threads, GPUs are built to handle thousands of threads in parallel. This design is particularly efficient for tasks that involve applying the same operation to every element in large data sets.

Over the years, GPUs have seen increased usage beyond graphics rendering. Modern computational pipelines, especially deep learning, simulation, and imaging pipelines, have benefited from GPU acceleration. In imaging pipelines, for instance, the parallel processing capability of GPUs facilitates rapid transformations, filtering, and other manipulations of pixel data. In deep learning, GPUs play an important role in accelerating the training of neural networks, making GPUs significantly responsible for the recent advancements in AI capabilities. As algorithms become more complex and data sets grow in size, the demand for high-throughput computational resources has surged. In this context, GPUs have emerged as an important processing unit in modern high-performance computing infrastructures. Therefore, optimizing the performance of GPUs, for both speed and energy efficiency, is relevant to extract the most out of existing computational pipelines, and reduce their environmental impact. In the remainder of this section, we give an introduction to GPU architectures, and programming on the GPU.

In a GPU architecture, threads are the smallest execution units. These threads are grouped into warps (usually 32 threads per warp), and warps are further grouped into blocks. A collection of blocks forms a grid. A block of threads is a unit that runs on a single streaming multiprocessor (SM) on the GPU. Each SM has a shared memory (that a block can use) and can synchronize its threads.

In addition to the hierarchy of threads, there is a memory hierarchy on the GPU. First, global memory is the main memory pool and is accessible by all threads, but it has higher latency than other types of memory. Second, shared memory is shared between threads in a block. It is faster than global memory but is limited in size. Third, local memory is used for thread-private variables and is only visible and accessible by the thread that allocated it. This memory typically is not accessed directly by the programmer but is rather used by the system as an overflow for registers. Lastly, a thread has a register memory that can exclusively be accessed by it. In general, scalar variables defined in GPU code are stored in registers. Additionally, GPUs can have texture and constant memory. These are special memory types optimized for specific use cases, such as storing constants for computations.

1.3.1 GPU Programming

A GPU program has both a CPU and a GPU part. The CPU is responsible for memory management, and for starting the functions that are executed on the GPU. GPUs are typically programmed using specific frameworks designed to exploit their parallel processing capabilities. In this work, we mainly focus on CUDA (Compute Unified Device Architecture), which is a parallel computing platform and API created by Nvidia that allows developers to use C, C++, and Fortran to code algorithms for execution on the GPU. As opposed to most code written in languages such as Python, writing GPU code presents challenges in designing

```
def add_vector(A, B, C, size):
    for i in range(0, size):
        C[i] = A[i] + B[i]
```

Figure 1.4: Example vector addition function in Python.

```
__global__ void vector_add(const float * A, const float * B, float * C,
                           const int size)
{
   int i = threadIdx.x;
   if ( i < size )
   {
      C[i] = A[i] + B[i];
   }
}

...
vector_add<<<1, N>>>(a, b, c, N);
```

Figure 1.5: Example vector addition function written in CUDA.

thread layouts and applying GPU-specific optimizations. To illustrate this, we will transform a simple vector addition function in Python (see Figure 1.4) to a CUDA program (called a GPU kernel):

When programming using CUDA, one of the key decisions a developer has to make is regarding the block size (number of threads per block) and grid dimensions (number of blocks). These parameters directly affect the distribution of tasks across the GPU's SMs. For example, a block size of 256 means that each block has 256 threads. The total number of threads launched for a kernel is equal to block size times grid size. A CUDA implementation of the aforementioned Python function could be (see Figure 1.5):

Here the keyword `__global__` specifies that the kernel is run on the GPU, and can be called from both the CPU and GPU (as opposed to `__device__` which runs on the GPU and can only be called from the GPU). The keyword `__host__` specifies that the function can be run and called only from the CPU. Apart from the type declarations, the main difference with the Python code is the omission of the for-loop, and the addition of the variable `threadIdx.x`. In CUDA, the `threadIdx` of a thread is a triplet that denotes the coordinate of the thread within a three-dimensional block. In this example, we are adding 1D vectors so we are only interested in the x coordinate. Instead of a loop running on a single CPU thread for the Python code snippet, the above CUDA code will be run on a GPU and every thread will execute a single addition in parallel. The final line (which is executed on the CPU portion of the code) indicates that the kernel will be launched on a grid with one block of N threads.

If we have one grid, and one block of threads, the above GPU kernel will

```
__global__ void vector_add(const float * A, const float * B, float * C,
                           const int size)
{
   int i = (blockIdx.x * blockDim.x) + threadIdx.x;
   if ( i < size )
   {
      C[i] = A[i] + B[i];
   }
}

...
vector_add<<<N/256, 256>>>(a, b, c, N);
```

Figure 1.6: Improved vector addition function written in CUDA.

typically only work for vectors of size 1024 or smaller. This is because most GPUs will not allow the execution of a block with more than 1024 threads. Instead, we should organize our computation in blocks. The keyword `blockDim` defines a triplet with the number of threads per dimension, and the keyword `gridDim` defines a triplet with the number of blocks per dimension. Using the `blockIdx` keyword we can modify the kernel to run on different blocks (see Figure 1.6).

Here we distribute the computation over $N/256$ blocks and 256 threads. Other design choices include what data types and data layouts to use in the various memory spaces available to GPU applications. Furthermore, there are other code optimizations available such as varying the amount of work per thread. An overview of many of the available optimization techniques for GPU programming is contained in [94].

The execution speed, and energy efficiency, of the kernel will depend greatly (potentially an order of magnitude or more) on choosing the correct block and grid sizes and applying useful code optimizations. However, this usually depends on the input size and the GPU model that the code is executed on. This means that a GPU kernel often needs to be (re)tuned to a specific situation. For this reason, auto-tuners have been developed to tune GPU kernels automatically [71, 127, 259]. In Chapter 5 we auto-tune several real-world kernels on many different GPU models, and benchmark different black-box optimization algorithms on the auto-tuning search space. Additionally, we look into a new method for quantifying the difficulty of GPU kernel search spaces. In Chapter 6 we use auto-tuners in combination with a data-driven power consumption model to improve the energy efficiency of GPU kernels. In the end, we look at the high-throughput LOFAR pipeline [78] and compute potential energy efficiency gains on different GPU models.

1.4 Research questions

In this thesis, we explore techniques to optimize different aspects of imaging pipelines used in contemporary industrial and scientific processes on different implementation

levels. On the whole, the thesis shows how concepts like parameter optimization, algorithm acceleration, hardware efficiency, and end-to-end optimization combine to allow for high-performance, energy efficient and potentially novel pipelines to modern computational imaging tasks. We mostly focus on X-ray computed tomography (CT) as an application field, but also consider radio astronomy pipelines that process data for the LOFAR telescope [78].

The chapters in this thesis deal with the following research questions.

In Chapter 2, we build upon RECAST3D to enable real-time quasi-3D segmentation of X-ray CT images. RECAST3D is a software package for real-time reconstruction and visualization of X-ray CT data. It uses a system of servers that in parallel process data, and reconstructs slices in real time that are requested by the user as they interact with the visualization. Here we intercept reconstruction data packets, use a convolutional neural network to segment the reconstructed slices, and let the visualization server show the segmentation to the user (see Figure 1.7).

Research question 1. *Can deep learning be used to perform segmentation of X-ray images in real time?*

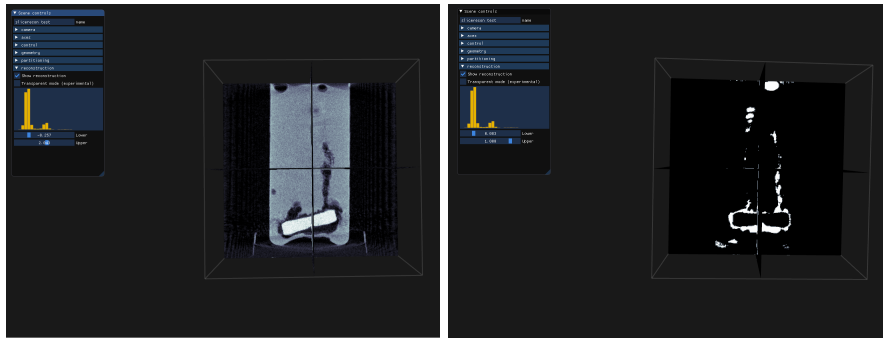
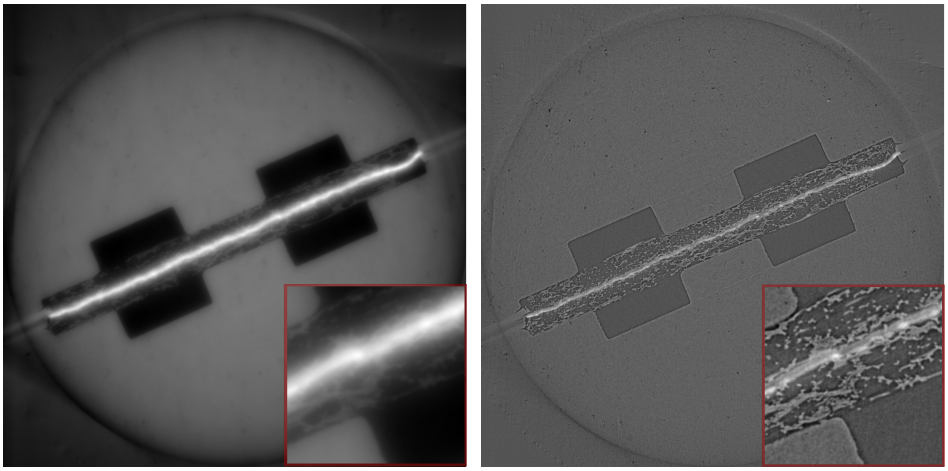


Figure 1.7: (Left) screenshot of a dissolving tablet reconstructed with FDK dynamically in RECAST3D, (right) segmented network output visualized in RECAST3D.

In Chapter 3, we perform four case studies on four CT workflows embedded in the PyTorch auto-differentiation framework. We expose various parameters of traditional CT algorithms to end-to-end optimization with gradient descent-based methods. The first workflow we consider is rotation axis alignment where we optimize the rotation axis shift applied in the projection domain for a quality criterion computed in the volume domain. Second, we optimize the refraction and attenuation indices for phase retrieval imaging using a self-consistent pipeline, i.e. we minimize the error between the raw projection data, and the simulated projection data acquired from forward propagating the segmented reconstruction. In Figure 1.8 we show a reconstructed slice before and after optimization. Third, we correct beam hardening artifacts by learning the beam spectrum and attenuation coefficients. Fourth, we denoise simulated CT images by joint optimization of the TV regularization parameter and CNN weights.

Research question 2. *To what extent can auto-differentiation be generalized to work effectively on a wide range of CT workflows?*



Before After

Figure 1.8: Hydrogen fuel cell [45] reconstructed with FBP with default material parameters for water (before), and after self-supervised optimization of parameters (after).

In Chapter 4, we accelerate convolutional neural networks by introducing a new pruning technique called *longest-chain pruning (LEAN)*. LEAN creates a directed graph representation where the operations are edges on the graph, and the weights of the edges are the operator norm. Next, the path through the graph with the highest multiplicative weight is iteratively selected to remain in the pruned network. Using LEAN we were able to significantly accelerate several neural networks in practice. Specifically, we measured an 11.1× speedup for the CNN used for real-time segmentation in Chapter 2 (see Figure 1.9).

Research question 3. *Can convolutional neural networks be accelerated significantly to enable real-time applications?*

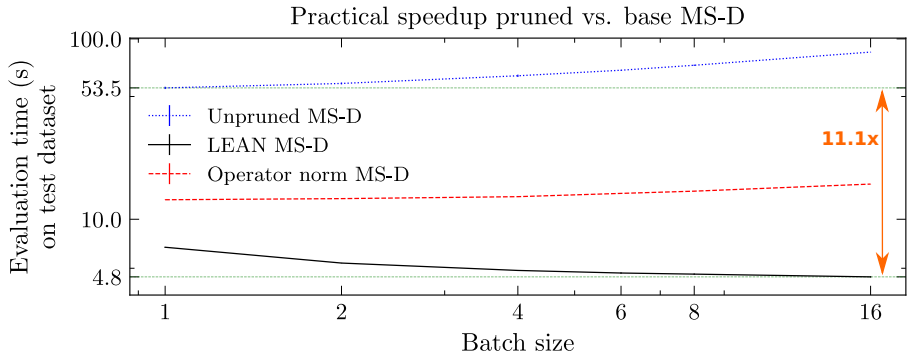


Figure 1.9: Practically realized speedup of pruned MS-D networks evaluated on the test dataset.

In Chapter 5, we perform a benchmarking study of 16 black-box optimization algorithms for 3 real-world GPU kernels on 9 different GPU models. The GPU kernels define a discrete search space with potentially many discontinuities (combinations of parameters that result in an invalid kernel) which can be challenging to traverse for traditional methods. In Figure 1.10 we show the varying distributions of local minima for different GPU models. We solve the optimization problem both as a deterministic and stochastic optimization problem and select the best-performing optimization algorithms using a statistical inter-algorithm competition. Additionally, as part of the study we contribute these optimization algorithms to the auto-tuning software package Kernel Tuner [246].

Research question 4A. *To what extent can black-box optimization algorithms be used to accelerate the process of auto-tuning GPU kernels?*

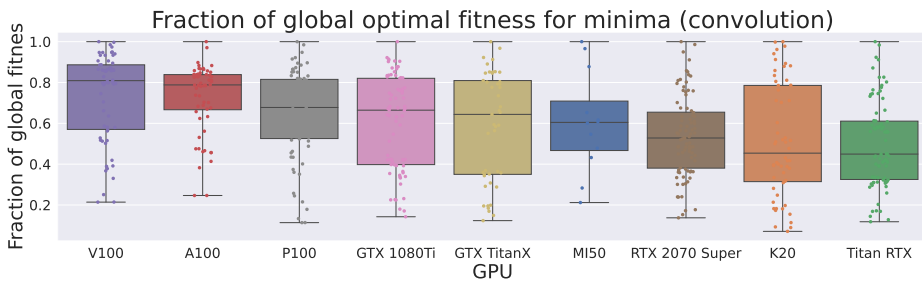


Figure 1.10: Fraction of optimal fitness of local minima for each GPU model, for the convolution kernel. The box plots shows the median line, the box designates the quartiles, and the whiskers the full extend of the distribution. Additionally, a scatter plot of the fitness for each local minimum is shown. The GPUs are ordered in descending median fraction of optimal fitness from left to right.

In Chapter 5, we introduce a graph-based method for quantifying the difficulty of discrete optimization search spaces. We introduce the *fitness flow graph (FFG)* (see Figure 1.11) where each point in the search space is a node, and a directed edge is drawn to neighbours with better fitness. We then use PageRank centrality [21, 173] to quantify the likelihood of a random walk ending in certain local minima (this mimics the behaviour of greedy local search). Next, we define a metric that quantifies how likely such random walks end in suitably good local minima to rank search space difficulty.

Research question 4B. *Can a graph representation of the GPU kernel search space be used to quantify the difficulty of the optimization problem?*

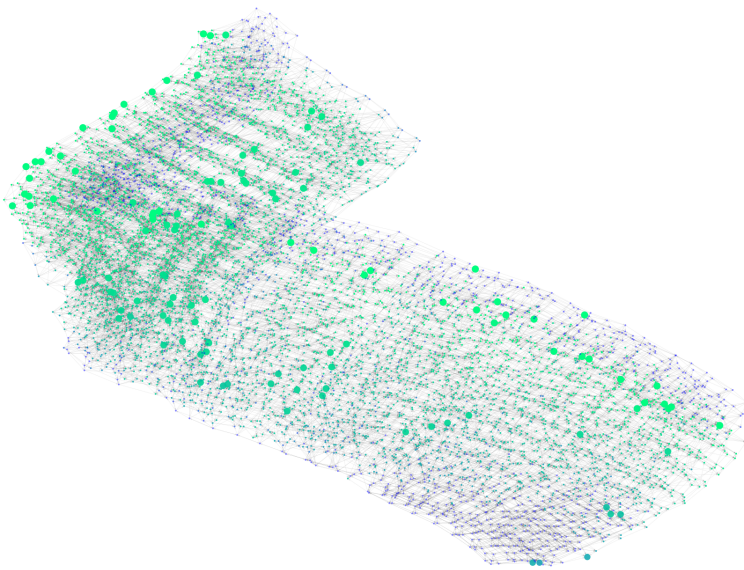


Figure 1.11: Fitness flow graph of point-in-polygon kernel search space of the Nvidia Titan RTX. Each node is a point in the search space. There is a directed edge between neighbouring points from higher to lower kernel runtime. Points are coloured within a fitness range of +25% with respect to the global minimal fitness (global minimum in green), i.e., each point is coloured by its fraction of optimal fitness, and points with a fraction below 0.75 are given the same colour. Local minima are represented as larger nodes.

In Chapter 6, we develop a model for GPU power consumption that greatly reduces the large tuning search space that is formed by adding core clock frequency as a tunable parameter (see Figure 1.12 for an example scatter plot). We use the model to provide clock frequencies for which a GPU is likely most energy efficient. We experiment on 6 kernels currently running in production for the low-frequency array (LOFAR) for 4 different GPU models and achieve measured improvements in energy efficiency. The power consumption model can be fitted using a few executions of a small example kernel. As part of the study, we contribute the power consumption model to Kernel Tuner as a script that can be executed on Nvidia GPUs to suggest the most energy-efficient clock frequency in under a minute.

Research question 5. *To what extent can we model the power consumption of GPUs and steer the auto-tuning process to improve the energy efficiency of GPUs?*

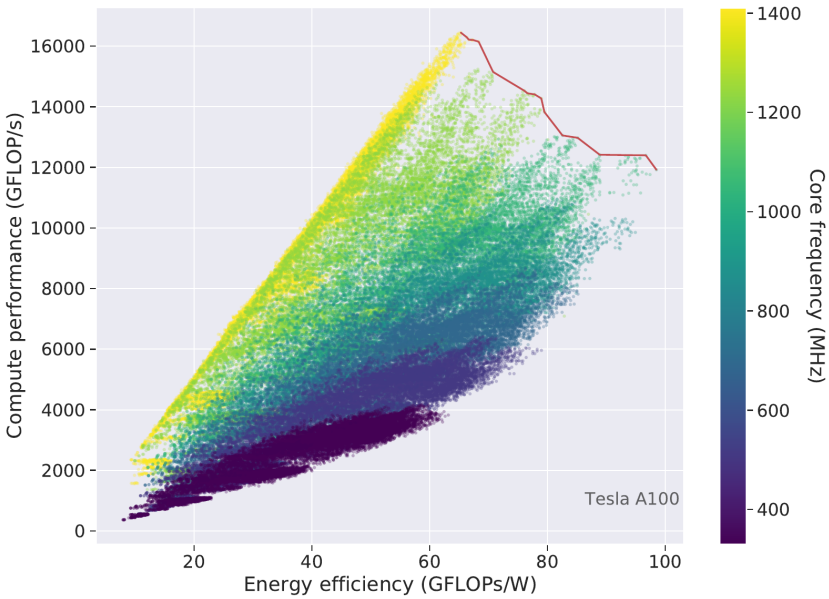


Figure 1.12: Kernel speed (GFLOP/s) over energy efficiency (GFLOPs/W) for all GEMM configurations for the Tesla A100. The red line is the Pareto front, i.e., neither performance or efficiency can be improved without decreasing the other. Points are coloured according to the core frequency.