Universiteit
Leiden
The Netherlands

On the optimization of imaging pipelines

Schoonhoven, R.A.

Citation
Schoonhoven, R. A. (2024, June 11). *On the optimization of imaging pipelines*. Retrieved from https://hdl.handle.net/1887/3762676

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/1887/3762676

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

On the optimization of imaging pipelines

Proefschrift

ter verkrijging van
de graad van doctor aan de Universiteit Leiden,
op gezag van rector magnificus prof.dr.ir. H. Bijl,
volgens besluit van het college voor promoties
te verdedigen op dinsdag 11 juni 2024
klokke 13:45 uur

door

Richard Arnoud Schoonhoven
geboren te Utrecht, Nederland
in 1995

Promotor:
Prof.dr. K. J. Batenburg

Co-promotores:
Dr. D. M. Pelt
Dr. B. J. C. van Werkhoven

Promotiecommissie:
Prof.dr. T. H. W. Bäck
Prof.dr. R. V. van Nieuwpoort
Prof.dr. A. Plaat
Prof.dr. ir. M. Staring
Dr. L. Cao
Dr. N. Viganò (European Synchrotron Radiation Facility)

The research presented in this dissertation was carried out at the Centrum Wiskunde & Informatica (CWI) in Amsterdam.

Financial support was provided by the Netherlands Organisation for Scientific Research (NWO) in the framework of the NWA-ORC Call (file number NWA.1160.18.316). This work is also financially supported by the Netherlands Organization for Scientific Research (NWO), project number 639.073.506 and 016.Veni.192.235. This work has made use of the experimental systems on the Dutch national e-infrastructure with the support of the SURF Cooperative. The DAS-6 cluster is funded through NWO-M and Open Competition (617.001.204) grants.

Ridderprint, the Netherlands
© 2024 Richard A. Schoonhoven,

Contents

1 Introduction **1**
- 1.1 X-ray CT pipelines . . . 2
- 1.2 Deep learning for imaging . . . 6
- 1.3 Graphics Processing Units . . . 8
- 1.4 Research questions . . . 11

2 Real-time segmentation for tomographic imaging **19**
- 2.1 Introduction . . . 19
- 2.2 Method . . . 21
- 2.3 Results and Discussion . . . 24
- 2.4 Conclusions . . . 31

3 Auto-differentiation for CT workflows **33**
- 3.1 Introduction . . . 33
- 3.2 Related work . . . 34
- 3.3 Methodology . . . 36
- 3.4 Case studies . . . 38
- 3.5 Discussion . . . 55
- 3.6 Conclusion . . . 56

4 LEAN: graph-based pruning for convolutional neural networks by extracting longest chains **59**
- 4.1 Introduction . . . 59
- 4.2 Related work . . . 60
- 4.3 Preliminaries . . . 61
- 4.4 Method . . . 63
- 4.5 Experimental setup . . . 67
- 4.6 Results . . . 69
- 4.7 Conclusion . . . 71

5 Benchmarking optimization kernels for auto-tuning GPU kernels **73**
- 5.1 Introduction . . . 73
- 5.2 Related work . . . 75
- 5.3 Method: Optimization problem . . . 77
- 5.4 Implementation . . . 83

5.5 Experimental Setup 84
5.6 Results: Benchmarking optimization algorithms on runtime 87
5.7 Quantifying GPU tuning difficulty 96
5.8 Conclusion 103

6 Going green: optimizing GPUs for energy efficiency through model-steered auto-tuning 105
6.1 Introduction 105
6.2 Related Work 106
6.3 Methodology 108
6.4 Experimental setup 111
6.5 Experimental results 114
6.6 Conclusions 127

7 Conclusion and outlook 129
7.1 Contributions and limitations 129
7.2 Outlook 131

Bibliography 133

List of publications 155

Samenvatting 157

Summary 159

Curriculum Vitae 161

Acknowledgments 163

A Appendices 165
A.1 Appendix: (LEAN) graph-based pruning for convolutional neural networks by extracting longest chains 166
A.2 Appendix: Benchmarking optimization algorithms for auto-tuning GPU kernels 168

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.

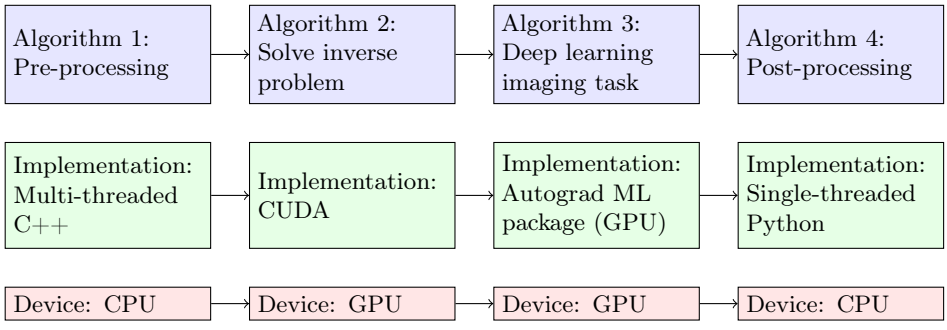


Figure 1.1: A schematic of an example imaging pipeline illustrating different implementation levels.

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.

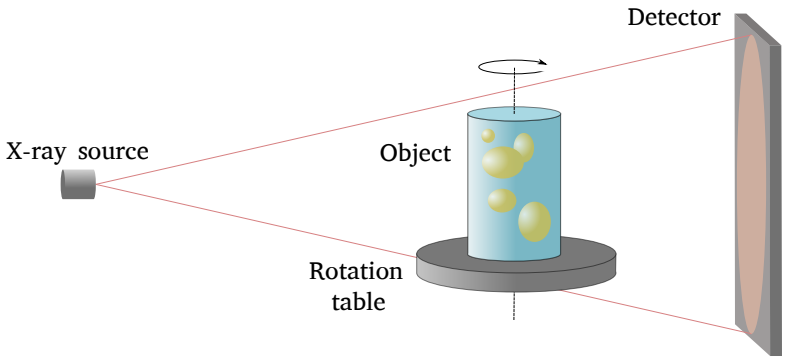


Figure 1.2: Example laboratory X-ray cone beam CT setup.

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

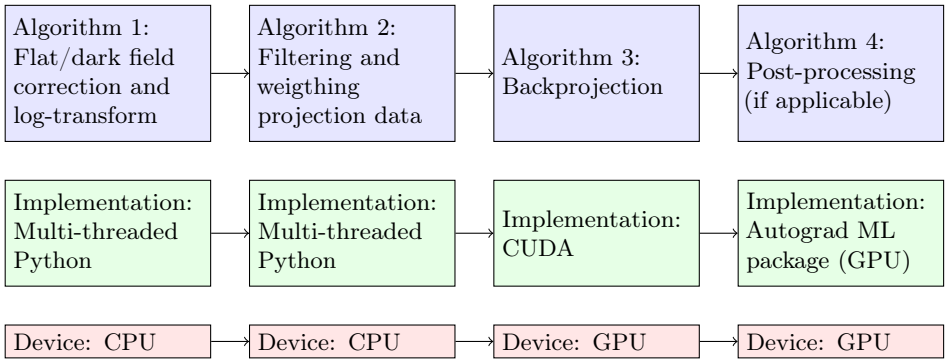


Figure 1.3: A schematic illustrating different implementation levels of an example CT imaging pipeline using FDK reconstruction.

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?*

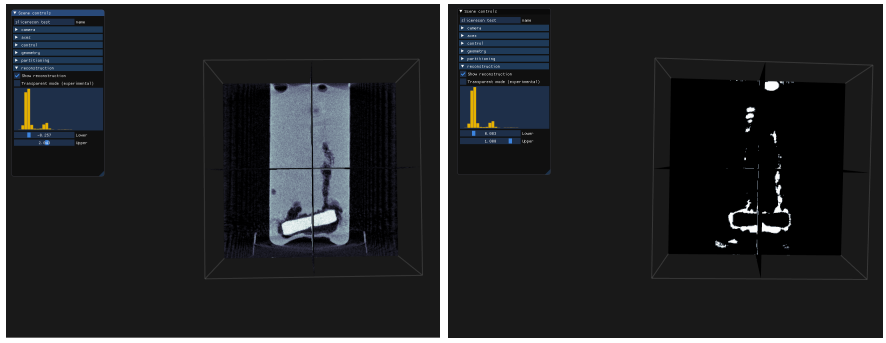
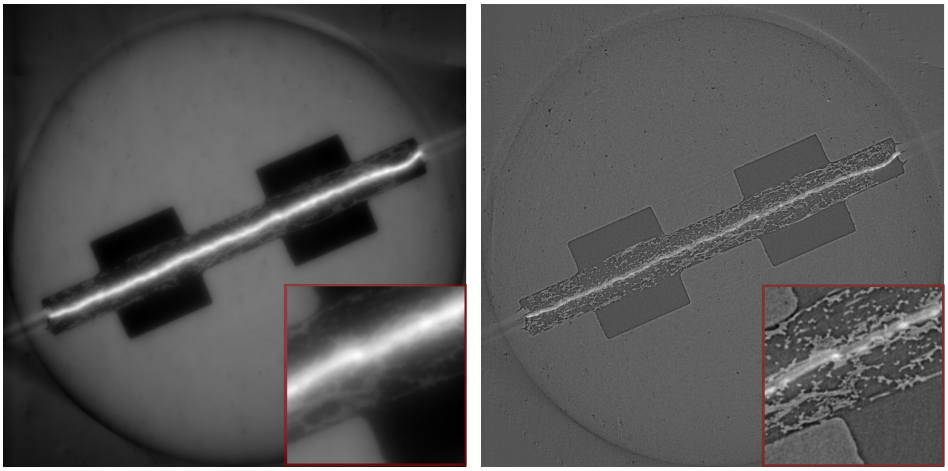


Figure 1.7: (Left) screenshot of a dissolving tablet reconstructed with FDK dynamically in RECAST3D, (right) segmented network output visualized in RECAST3D.

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?*

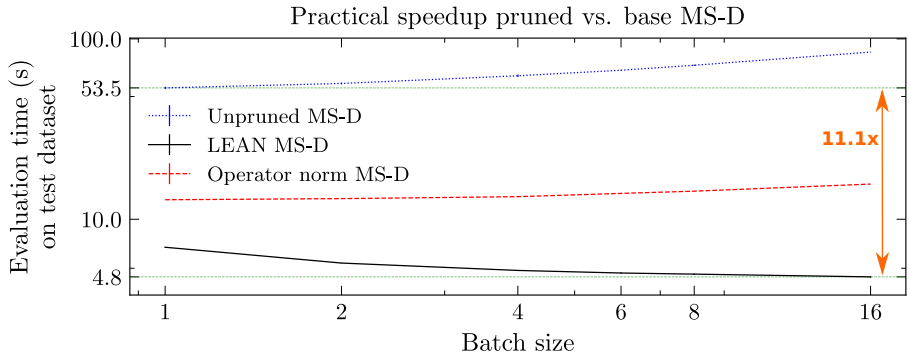


Figure 1.9: Practically realized speedup of pruned MS-D networks evaluated on the test dataset.

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?*

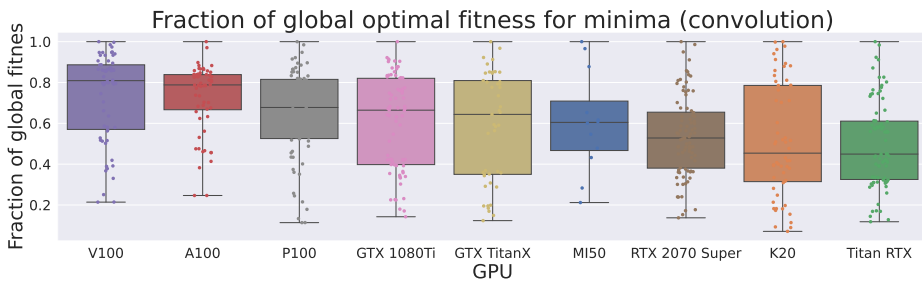


Figure 1.10: Fraction of optimal fitness of local minima for each GPU model, for the convolution kernel. The box plots shows the median line, the box designates the quartiles, and the whiskers the full extend of the distribution. Additionally, a scatter plot of the fitness for each local minimum is shown. 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?*

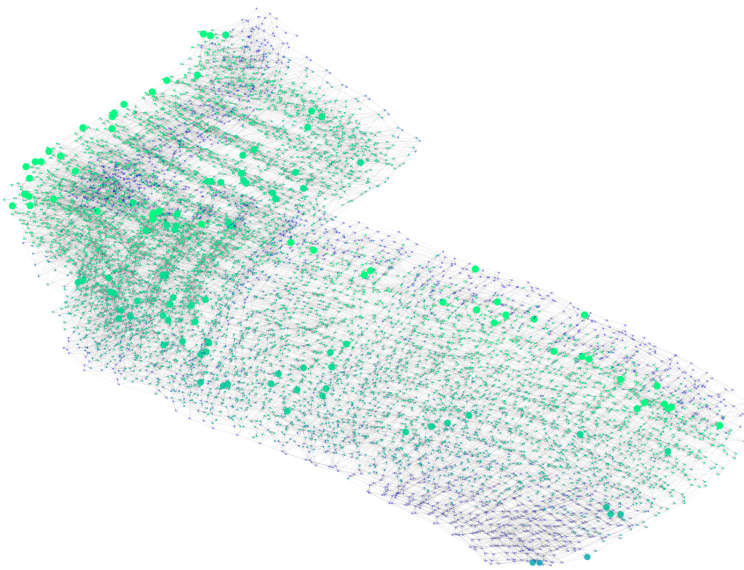


Figure 1.11: Fitness flow graph of point-in-polygon kernel search space of the Nvidia Titan RTX. Each node is a point in the search space. There is a directed edge between neighbouring points from higher to lower kernel runtime. Points are coloured within a fitness range of +25% with respect to the global 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?*

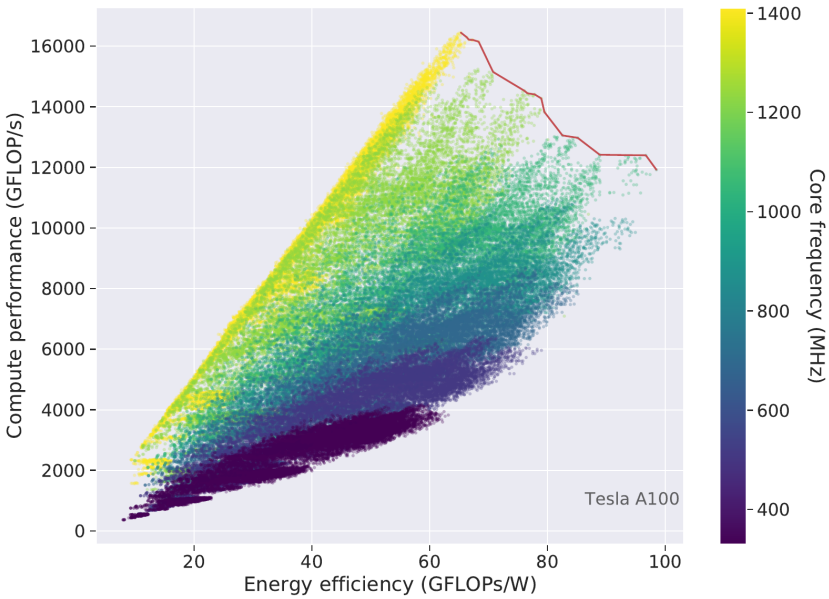


Figure 1.12: Kernel speed (GFLOP/s) over energy efficiency (GFLOPs/W) for all GEMM configurations for the Tesla A100. The red line is the Pareto front, i.e., neither performance or efficiency can be improved without decreasing the other. Points are coloured according to the core frequency.

2

Real-time segmentation for tomographic imaging

2.1 Introduction

Tomographic imaging is a widely applicable technique for studying the internal structure of objects using some form of penetrating radiation such as X-rays or an electron beam. Projection images are obtained from a range of angles and a tomographic reconstruction algorithm subsequently computes a 3D image of the internal structure of the object. Currently, reconstruction and analysis are often performed after image acquisition has completed. If processing, reconstruction, and analysis of tomographic data can be run in real time during the experiment, internal dynamic processes of the imaged object can be visualized and analyzed as they occur. Real-time feedback enables online optimization and steering of the imaging setup and experimental conditions which increases the efficiency of experiments and avoids costly repetition.

Despite advances in computationally efficient reconstruction algorithms [13, 112] and in specialized hardware such as Graphic Processing Units (GPUs) [174] and supercomputers [16], full 3D tomographic reconstructions at the rate of data acquisition remain out of reach for most applications. Recently it was shown that real-time reconstruction can be achieved for a small set of arbitrarily oriented 2D slices [27]. These slices can be adjusted on the fly, thereby giving access to a virtual full 3D volume at a fraction of the computational cost. This methodology is called quasi-3D reconstruction, and is implemented in the RECAST3D software package.

To enable adaptive imaging, where the imaging process is adjusted based on the observations, just having access to a reconstructed volume is not sufficient, as the image analysis step should also be included in the real-time processing pipeline.

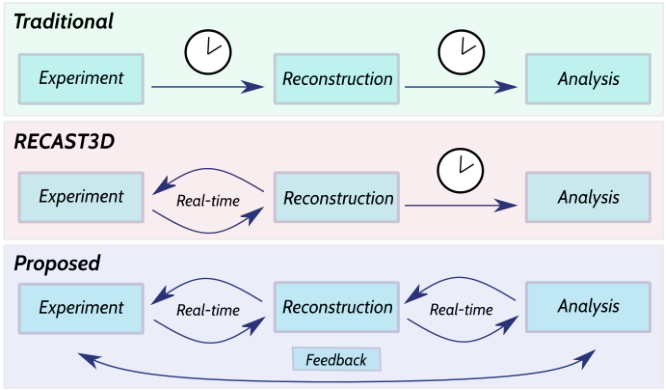


Figure 2.1: Traditional experiments (top) involving tomography require significant time for both the reconstruction phase and the offline analysis phase. With RECAST3D (middle) the reconstruction phase is performed in real time. Our method (bottom) additionally includes a real-time segmentation step.

An important step in many image analysis pipelines is segmentation, which is the problem of assigning to each pixel the appropriate class label from a finite set of classes, for example segmenting bone for calcaneal fractures in CT images [194]. In this article we introduce a real-time imaging pipeline to reconstruct, segment, and visualize quasi-3D volumes implemented as an extension of the existing RECAST3D software package. Our method adds real-time segmentation to the existing real-time reconstruction capabilities of the RECAST3D framework, as outlined in Figure 2.1.

As quasi-3D reconstruction employs direct reconstruction methods such as filtered backprojection (FBP) [106] and Feldkamp-David-Kress (FDK) [59] without additional image regularization, limited-data artefacts are typically present in the reconstructions. These artifacts limit the applicability of computationally efficient unsupervised segmentation algorithms, such as Otsu's method [170], since they are often unable to separate artifacts and noise from important features. Furthermore, because image analysis algorithms may be sensitive to noise in the segmentation [32, 150, 199], analysis based on such traditional segmentation methods may not be accurate. In addition, many unsupervised segmentation methods operate exclusively on the basis of the pixel values [124, 163, 170], limiting their applicability to general segmentation problems as they are unable to segment features that are not based on pixel values.

To overcome these issues, we propose to use a convolutional neural network (CNN) to segment the quasi-3D reconstructions in real time. To apply CNNs in a quasi-3D setting, we introduce an adapted training strategy that takes the arbitrary orientations of the slices into account. We show that a CNN is capable of achieving similar accuracy to segmentations based on computationally more expensive total variation minimization (TV-MIN) reconstructions [14] which are too slow to compute for real-time applications. In addition, we show that a CNN can

be implemented efficiently as a plugin within the existing RECAST3D framework without significantly increasing the processing time.

This article is structured as follows. In Section 2.2 we introduce the tomographic reconstruction problem and define the FDK and TV-MIN reconstruction algorithms. We introduce quasi-3D reconstructions, the segmentation problem, and provide more details on the segmentation plugin. Lastly, we outline our adapted training strategy for randomly oriented slices. In Section 2.3 we present the experimental results, and analyze the training strategies. We perform a real-world experiment on a dynamic X-ray CT dataset and two simulated experiments. Finally, in Section 2.4 we state our final conclusions.

2.2 Method

2.2.1 Prerequisites

Tomographic Reconstruction

The tomographic reconstruction problem is to recover a volume from a series of its projections. In this article we consider circular cone-beam tomography, where the object is placed in between a point source and flat-panel detector which are situated on opposite sides of a circle. The object is rotated and X-ray projections are taken at a selection of equidistant angles. The approach generalizes to other acquisition geometries (e.g. parallel beam) in a straightforward manner.

The tomographic reconstruction problem can be modelled as an inverse problem:

$$K\mathbf{u} = \mathbf{f}. \tag{2.1}$$

Here K is the forward projection operator, $\mathbf{u} \in \mathbb{R}^{N_x \times N_y \times N_z}$ represents the object, and $\mathbf{f} \in \mathbb{R}^{N_\theta \times N_a \times N_b}$ is the measured projection data, with N_θ is the number of projection angles, and N_a, N_b are the number of detector rows and columns respectively. In this article we use the FDK reconstruction algorithm, given by

$$\mathbf{u}_{\text{FDK}} = K^T(\mathbf{h} * \tilde{\mathbf{f}}). \tag{2.2}$$

Here $\tilde{\mathbf{f}}$ denotes weighted projection data, which compensates for diminishing intensity at distance from detector center, and $\mathbf{h} \in \mathbb{R}^{N_b}$ is a 1D filter. We used the Ram–Lak filter for this work.

Instead of using FDK, equation 2.1 can be solved by iteratively minimizing $\|K\mathbf{u} - \mathbf{f}\|$. In addition, we can add prior information about the gradient of the image being sparse by adding a total variation term [14] to improve reconstruction accuracy when projection data is limited or noisy:

$$\frac{1}{2}\|K\mathbf{u} - \mathbf{f}\|_2^2 + \lambda\|\nabla\mathbf{u}\|_1.$$

This function can be minimized by a range of convex optimization algorithms.

Quasi-3D Reconstruction

Quasi-3D reconstruction has recently been proposed as a method to make real-time tomographic reconstruction feasible [27]. Instead of computing a full 3D volume, only a small collection of arbitrarily oriented 2D slices is reconstructed and visualized in real-time. When these slices are translated and/or rotated by the user, they are reconstructed on the fly, so that it appears as though a full 3D reconstruction is available. This on-demand 2D reconstruction significantly reduces the total computational cost compared to full 3D reconstruction. This approach is implemented in the open source RECAST3D software package and more implementation details can be found in [27].

In RECAST3D, the filtering and weighting steps of the FDK algorithm are performed in parallel. The computation of $\mathbf{h} * \tilde{\mathbf{f}}$ is performed in real time from the incoming data. When a slice is requested, the application of K^T (called backprojection) is performed using GPU-based high-performance routines from the ASTRA toolbox [1]. In addition, a low-resolution 3D FDK reconstruction is created so that the user can preview the object. Our quasi-3D pipeline for segmentation is implemented by extending the RECAST3D software package with a computationally efficient segmentation plugin.

Segmentation

Mathematically, segmenting an image can be described as finding a function $g : \mathbb{R}^{m\times n} \to \mathbb{Z}_k^{m\times n}$, where m, n are the rows and columns of the image and k is the number of object classes to be assigned.

Classical segmentation methods (for example local and global thresholding [124, 170], watershed methods [163]) typically operate on the image greyvalues to separate classes and have the high computational efficiency that is need for real-time segmentation. As an example, Otsu's method performs a segmentation of an image by selecting a threshold that minimizes intra-class variance. In addition to the greyscale distribution, segmentation can be performed on other properties by for example clustering pixels [48] or defining edge boundaries in the image [157]. Recently, CNNs have proven successful for image segmentation [9, 202].

CNNs for segmentation

In this work we use CNNs to segment the tomographic reconstructions. In a segmentation network, the final output layer will assign one of k classes to each pixel. The CNN is defined by its architecture with weights Θ which can be altered to change the output. For a given Θ, a CNN corresponds to a function $F_\Theta : \mathbb{R}^{m\times n} \to \mathbb{R}^{k\times m\times n}$ which aims to approximate g by computing a probability vector with predictions for each class for each pixel. The highest probability class can be chosen from the network predictions to obtain a final segmentation.

The weights Θ are found in a training phase, where input samples $\mathbf{x}_1, \ldots, \mathbf{x}_N$ are processed by the network and compared to known labelled output samples $\mathbf{y}_1, \ldots, \mathbf{y}_N$. A loss function $\mathcal{J} : \mathbb{R}^{k\times m\times n} \times \mathbb{Z}_k^{m\times n} \to \mathbb{R}$, such as cross-entropy loss,

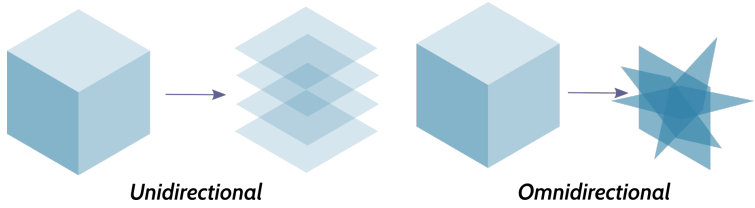


Figure 2.2: Diagram outlining unidirectional training on slices (left) and omnidirectional training (right).

measures the error of the network on the training samples. The aim of the training phase is to find a Θ that minimizes the loss on the training dataset

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

For a CNN, we can compute the partial derivatives of $\mathcal{J}$ with respect to the weights using backpropagation. The weights can be updated using gradient-based optimization algorithms [113].

2.2.2 Quasi-3D training strategies

In 3D image segmentation, neural networks are often trained on 2D slices from the volumes since full 3D networks are typically computationally too expensive [134, 183]. The 2D input slices are obtained by extracting slices in a single direction. In contrast, slices in a quasi-3D reconstruction can have an arbitrary orientation. A network trained only on unidirectional slices may not recognize object classes from a different view. Therefore, the standard training procedure has to be adapted to enable application of CNNs to arbitrary oriented slices. Here, we introduce a training strategy where arbitrarily oriented 2D slices of the tomographic volume are supplied as input for the neural network.

Let $\mathbf{X} \in \mathbb{R}^{n \times n \times n}$ be an input volume and $\mathbf{Y} \in \mathbb{Z}_k^{n \times n \times n}$ the aligned target volume. Define $E_{\alpha,\beta,\gamma} : \mathbb{Z} \times \mathbb{R}^{n \times n \times n} \to \mathbb{R}^{n \times n}$ to be a rotated extraction operation. $E_{\alpha,\beta,\gamma}(i, \mathbf{X})$ extracts the i-the slice rotated by angles α, β, γ with respect to the sagittal slice from the volume $\mathbf{X}$. For *omnidirectional training* we create a dataset of slices with pairs $(E_{\alpha,\beta,\gamma}(i, \mathbf{X}), E_{\alpha,\beta,\gamma}(i, \mathbf{Y}))$ where the angles are randomly generated. For *unidirectional training* we create pairs of sagittal slices $(E_{0,0,0}(i, \mathbf{X}), E_{0,0,0}(i, \mathbf{Y}))$ (see Figure 2.2).

2.2.3 Segmentation Plugin

To construct the pipeline for real-time segmentation we developed a plugin for RECAST3D which segments quasi-3D reconstructions. The segmented slices are then visualized in RECAST3D. The plugin is GPU-based and can be disabled,

altered and reenabled while the projection data is processed simultaneously. A command line interface allows for online tuning of parameters with immediate visual feedback, and the user can select which class is visualized. The plugin operates independently from the quasi-3D reconstruction pipeline, and can be run on a separate node.

The plugin is implemented in Python, and includes three CNNs implemented in PyTorch: the MS-D network [89, 183], U-Net [202], and ResNet [84]. In addition, the plugin includes several traditional segmentation methods, including Otsu's method [170], cross entropy thresholding [124], contour evolution [158], region based random walk segmentation [69] and the watershed algorithm [163].

2.3 Results and Discussion

2.3.1 Setup

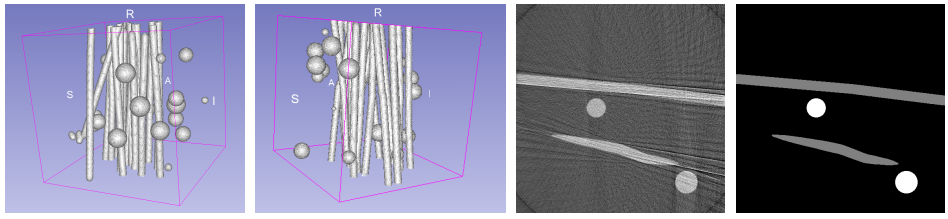
To assess the accuracy and computational efficiency of our CNN-based segmentation approach, we compare it with traditional unsupervised methods. The neural network used in this work is the MS-D network [183], chosen because of its low number of trainable parameters.

Since the MS-D network can flexibly adapt to different problems, we used the same network architecture for each experiment. We used an MS-D network, implemented in PyTorch [89], of 100 layers with a width of 1. The dilations in layer i were set to $1 + (i \mod 10)$. The networks were trained using the ADAM algorithm [113], using a batch size of 20, and the cross-entropy loss function. For each experiment, the data was split in training, validation, and test sets. The network is trained on the training set for 100 epochs. The network with the lowest validation error was selected to be evaluated on the test set. All experiments were run on a workstation with an AMD Ryzen 3800X processor and NVIDIA GeForce RTX 2070 Super GPU. To quantify our comparisons we use the F1-score (Dice coefficient) which is the harmonic mean of precision and recall, and the accuracy. For multi-class problems we report the macro F1-score (average of per-class F1-score), and the global accuracy.

2.3.2 Simulated data

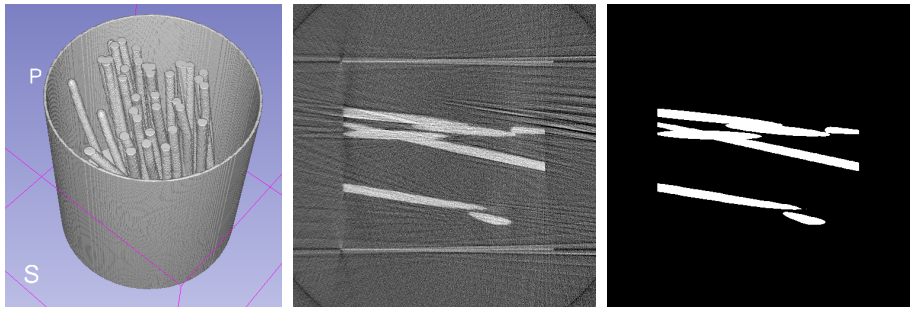
We created two sets of simulated tomographic data to investigate the difference between omni- and unidirectional training, and to quantitatively compare Otsu's method to the MS-D network. For the former we created twenty 512^3 volumes filled with fibre strands and spheres. Each volume contained 20 spheres and 20 fibres which were generated with a random shape and location. Greyscale intensities were the same for both classes. 3D renderings of the noiseless volumes are shown in Figure 2.3.

For the second simulation experiment we created twenty 512^3 volumes filled with 40 fibre strands generated with random shapes surrounded by a cylinder



(a) (b) (c) (d)

Figure 2.3: (a), (b) Example volumes of the fibre-sphere data, (c) slice of the noisy FDK reconstruction, and (d) slice of the ground truth (spheres and fibres have different labels).



(a) (b) (c)

Figure 2.4: (a) Example volume of the fibre-container data with container, (b) slice of the noisy FDK reconstruction, and (c) slice of the ground truth (the container is not to be segmented).

forming a container (see Figure 2.4). Greyscale intensities were the same for the container and the fibres to represent a segmentation problem where the fibres are to be labeled but the container is not.

Using the ASTRA toolbox [1] we simulated a cone-beam projection dataset of 60 projections for each volume. Furthermore, Poisson noise was applied to the projection dataset where the sample absorbed roughly 70% of the incoming photons. Out of the 20 phantoms, 14 were randomly selected for training, 4 for validation and 2 for testing. For each scan we obtained 500 randomly oriented, and 500 unidirectional 512×512 slices for both the ground truth labeled volume, and the noisy FDK reconstruction (see Figures 2.3 and 2.4).

Comparison of uni- and omnidirectional training slices

To investigate the importance of the slicing direction in the training strategy for CNNs, we compare unidirectional and omnidirectional training strategies. We

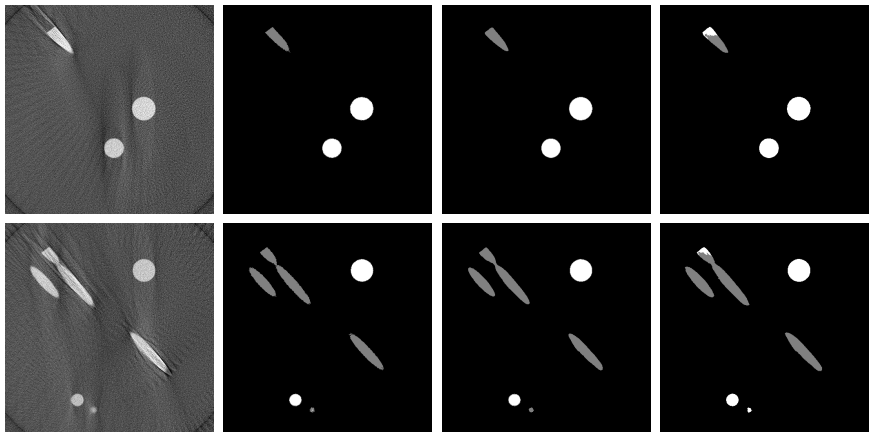


Figure 2.5: Slices of the simulated fibre-sphere test dataset and MS-D network predictions. From left to right we have the FDK reconstructions, the labeled ground truths, the MS-D network output trained on randomly oriented slices, and the MS-D network output trained on unidirectional slices.

	Random orientation		Sagittal orientation	
Training	**F1**	**Accuracy**	**F1**	**Accuracy**
Omnidirectional	**0.9989**	**0.9979**	0.9951	0.9950
Unidirectional	0.6857	0.9792	**0.9995**	**0.9995**

Table 2.1: Macro F1 and accuracy on the fibre-sphere (left) randomly oriented test set, and (right) sagittal test set, for MS-D networks trained with the omnidirectional, and unidirectional strategies.

trained one network on the slices in a single direction and the other on randomly oriented slices, using the fibre-spheres dataset (Figure 2.3). Otsu's method, and any other unsupervised method that works solely on greyvalues, are not applicable to this type of multi-class segmentation problem as they cannot categorize objects with the same greyvalue. This indicates an advantage of deep-learning approaches.

Example results from the test set are shown in Figure 2.5. Note that both networks were able to remove the Poisson noise and the tomographic artifacts. Some notable differences between both networks can be seen where the unidirectional network identifies parts of the fibre strands as spheres. In Table 2.1 we report the macro F1-score, and accuracy for both networks on both the test set with randomly oriented slices and the test set with slices in a single direction.

The MS-D omnidirectional network outperforms the unidirectional network on both quantitative measures for the randomly oriented test set, which is in

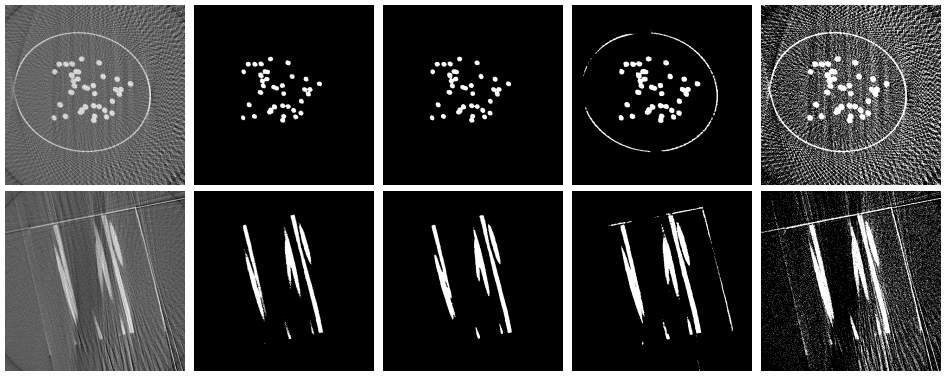


Figure 2.6: Slices of the simulated fibre-container test dataset, with (from left to right): the FDK reconstructions, the labeled ground truths, the omnidirectional MS-D network output, the unidirectional MS-D network output, and Otsu's method.

line with the qualitative comparison shown in Figure 2.5. When tested on the sagittal direction, although the unidirectional networks performs better than the omnidirectional network, the difference in performance is significantly smaller. This can be explained by the fact that the omnidirectional network has also encountered images sliced in the sagittal direction.

Comparison of MS-D segmentation and Otsu's method

Here, we compare Otsu's method to the uni- and omnidirectional MS-D networks. Both the MS-D networks and Otsu's method were tested on randomly oriented slices from the fibre data test phantoms because the user can select arbitrary slices in RECAST3D. To more accurately determine the performance of Otsu's method inside the container, we created a version of Otsu's method where a ROI-mask was applied on the segmented slices with the proper rotation. We used a cylindrical volume around the simulated container to remove misclassified background. Some example results from the test set can be seen in Figure 2.6 and in Table 2.2 we report the F1-score and accuracy.

The results show that the omnidirectional MS-D network was able to accurately segment the fibres and remove the applied Poisson noise. We see that the unidirectional MS-D network misclassified the container in the randomly oriented slices, and that Otsu's method occasionally does not remove the FDK artifacts and noise. In addition, Otsu's method classifies the container as a fibre since it is unable to distinguish it from the fibres on the basis of intensity. Even if we manually mask the region-of-interest, the interior of the container is significantly more noisy than the MS-D network segmentation.

	F1-score	**Accuracy**
MS-D omnidirectional	**0.9544**	**0.9989**
MS-D unidirectional	0.6777	0.9885
Otsu	0.0585	0.6086
ROI-Otsu	0.2568	0.9300

Table 2.2: F1 and accuracy on the fibre-container randomly oriented test set for MS-D omnidirectional, MS-D unidirectional, Otsu, and ROI Otsu.

The MS-D network outperforms Otsu and ROI-Otsu on both metrics. Otsu's method performs significantly worse on F1-score, which can be explained by the greater amount of false positives segmented by Otsu as opposed to the MS-D network.

2.3.3 Experimental data

To show the feasibility of our method in real-world applications, we applied the real-time segmentation pipeline to a real-world dynamic X-ray CT dataset of a dissolving tablet suspended in gel [38, 39]. A container with a dissolving tablet was filled with gel to create moving air bubbles which we segmented. The container was rotated at 100 deg/s and 60 projections were acquired every 180 degrees with an exposure time of 30 ms for each projection. In total 9960 projections of size 647×768 were taken and the experiment lasted 5 minutes.

In RECAST3D, the full processing step of a batch of projections takes approximately 140 ms on our workstation. The computation time to compute the backprojection for a slice is about 2 milliseconds. The segmentation with Otsu's method is about 3 milliseconds and with the MS-D network about 30 milliseconds. This means that the pipeline would be able to dynamically visualize the projection data stream every 170 milliseconds for a batch of 60 projections. In this experiment, data acquisition was at a rate of 1.8 seconds per batch of 60 projections (180 degrees), well within the computational limits of the pipeline. Figure 2.8 shows an example real-time quasi-3D reconstruction and segmentation of the data in RECAST3D.

To create training data for the neural networks we created TV-MIN reconstructions for every 60 projections with a regularization parameter $\lambda = 0.001$ for 2000 iterations. We used the Douglas–Rachford primal-dual splitting algorithm [19] to iteratively minimize the functional. Each TV-MIN reconstruction took roughly 20 hours on our workstation and is therefore infeasible to compute in real time. Next, we created 25 labeled ground truth volumes by applying Otsu's method to the TV-MIN reconstructions and masking the region outside the container. As a final processing step we removed small objects with a mass smaller than 4 pixels with the scikit-image `remove_small_objects` function from the morphology package [242]. The scans were randomly separated into 18 training scans (9216 slices), 4 validation scans (2048 slices) and 3 test scans (1536 slices) for the unidirectional

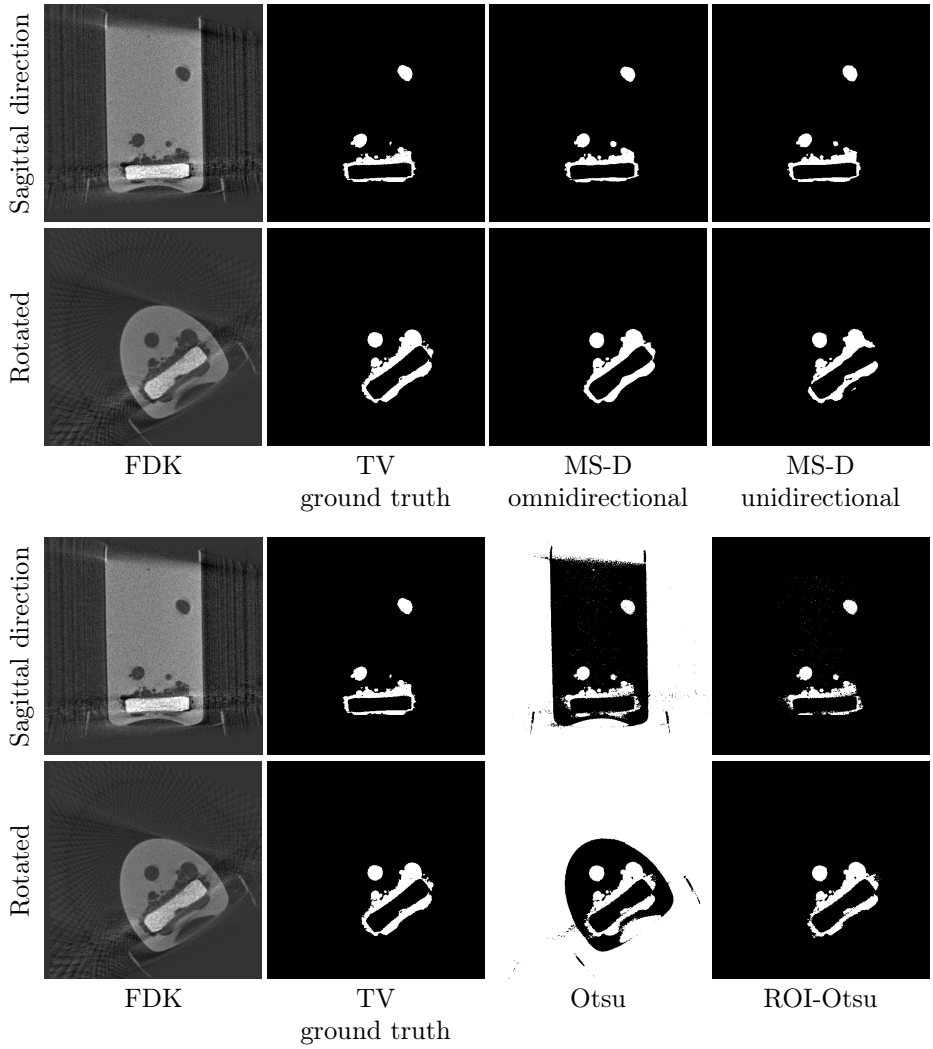


Figure 2.7: Slices of the TabletInFluid test dataset and experimental predictions. The first row is from the unidirectional dataset, the second from the randomly rotated dataset. From left to right we have the FDK reconstructions, the labeled TV-MIN ground truths, the MS-D network output trained on randomly oriented slices, the MS-D network output trained on slices in one direction, Otsu's segmentation, and Otsu's segmentation using a ROI cylindrical mask.

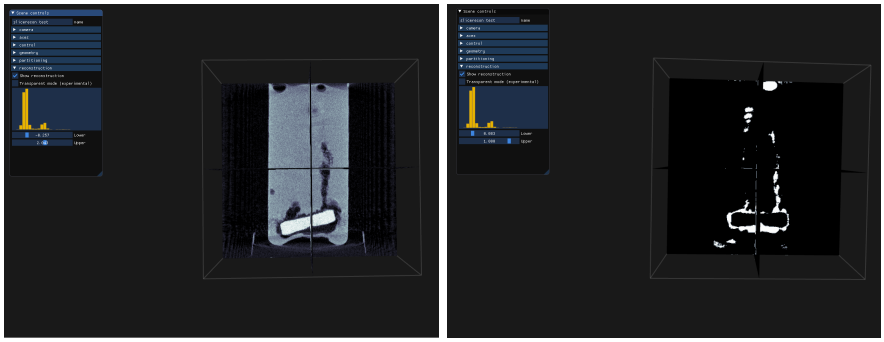


Figure 2.8: (Left) example screenshot of dissolving tablet data reconstructed dynamically in RECAST3D, (right) example of segmented network output in RECAST3D.

	F1-score	**Accuracy**
MS-D omnidirectional	**0.8816**	**0.9983**
MS-D unidirectional	0.7595	0.9968
Otsu	0.0142	0.2538
ROI-Otsu	0.8229	0.9977

Table 2.3: F1 and accuracy on the real-world TabletInFluid randomly oriented test set for MS-D omnidirectional, MS-D unidirectional, Otsu, and ROI Otsu.

network. For the arbitrarily oriented slices, we chose the same amount of slices at random 3D orientations for each scan.

To compare our method to an existing computationally efficient method, we segmented each FDK slice with Otsu's method and manually applied a cylindrical ROI-mask to remove misclassified background. The results can be seen in Figure 2.7. In Table 2.3 we report the F1-score and accuracy for the randomly oriented slices.

The MS-D network trained on randomly rotated slices is able to create an accurate segmentation of the bubbles in real time and it outperformed the other three methods on all metrics. It is able to create real-time segmentations with similar quality to the computationally expensive segmented TV-MIN reconstructions. This shows that our method can be used to perform quasi-3D reconstruction and segmentation in real time by training on randomly oriented slices of segmented TV-MIN reconstructions. Note that, in practice, acquiring training data and training the networks has to be performed offline. The results show that our method outperforms Otsu's method with masking on all metrics. Notably, the unidirectional network regularly misclassifies sections of the randomly oriented slices. The importance of the training method is highlighted when comparing the F1-scores for both MS-D networks.

2.4 Conclusions

In this paper, we introduced a real-time pipeline to process, reconstruct, and segment quasi-3D tomographic images, representing an important step for online and real-time analysis of tomographic experiments. We showed the importance of including arbitrarily oriented slices in the training dataset to achieve accurate results. We demonstrated that a deep-learning based approach can perform better than Otsu's method in terms of accuracy on both simulated data and real-world dynamic tomographic data. In addition, our deep-learning based approach is more generalizable to multi-class segmentation problems than traditional intensity-based unsupervised segmentation methods. Using our method, one can perform real-time and online segmentation of quasi-3D volumes, enabling immediate feedback and analysis during experiments.

3 Auto-differentiation for CT workflows

3.1 Introduction

In recent years, deep learning and other data-driven machine learning approaches have become increasingly popular in computed tomography. Deep neural networks have achieved strong results in X-ray CT applications by improving reconstruction quality [146], reducing metal artifacts [260], performing beam hardening correction [262], and classification [34, 98, 171]. The progress in deep learning has shown the power of data driven end-to-end optimization using auto-differentiation software, often in combination with hardware acceleration using graphical processing units (GPUs).

Despite promising results, deep learning approaches suffer from interpretability and reproducibility challenges. Furthermore, a serious issue is the behaviour of deep learning algorithms when presented with new data, and potential hallucination of object features [15]. These considerations have hampered the adoption of learned algorithms by experts in for example the medical domain where doctors can be reluctant to trust black-box learned approaches.

In contrast, classical (i.e. non-learned) algorithms often excel in interpretability and robustness. Additionally, classical algorithms often come with an intuition for their applicable input data domain, and the results they should produce. This can lead to expert users favouring such traditional methods over deep learning approaches. However, parameters of these algorithms are often either kept fixed or are adjusted by the user based on manual experimentation. Since these parameters are not learned in a data-driven way, they may not be optimally chosen for the particular dataset and application.

In this work, we apply concepts from the philosophy that made deep learning-based methods so successful, and transfer it to CT workflows. We create workflows of computed tomography algorithms for various problems and show how those problems can be solved by end-to-end optimization based on auto-differentiation. By doing so, we reconcile classical algorithms with deep learning. We perform four case studies, representative of real-world tomography problems. To do so, we create pipelines using the following core design principles:

End-to-end learning: We implement the pipelines such that all pieces facilitate gradient propagation, meaning that parameters at all steps of the pipeline can be optimized jointly.

Explicit quality criteria: All pipelines use explicit quality criteria as objectives for optimization, thereby enabling automatic optimization of parameters.

Declarative algorithm construction: Each pipeline is created by building the forward model, and we optimize its parameters using auto-differentiation and generic readily-available algorithms.

Use existing building blocks: The pipelines re-use building blocks derived from both classical algorithms and deep learning methods in a way that enables gradient propagation. This allows for seamless compatibility with deep learning-based methods.

Our portfolio of case studies sketches the outline of a new generation of powerful software toolboxes that enable users to leverage the power of auto-differentiation for advanced computational CT pipeline construction.

This work is structured as follows. In Section 3.2 we explore related work in auto-differentiation software for classical methods, and approaches for learning CT algorithms in a data-driven way. In Section 3.3 we describe the methodology. In Section 3.4 we present our results in the form of four case studies. In Section 3.5 we discuss our findings and present key potential benefits that arise from our approach. We present our final conclusions in Section 3.6.

3.2 Related work

The idea of extending auto-differentiation techniques [72] to classical algorithms is actively investigated across several domains. In the field of robotics, for non-linear optimization problems, Meta AI constructed Theseus [188] which is a library for building custom non-linear optimization layers that were shown to be useful for differentiable kinematics. In [36], robotic controllers are auto-tuned using the DiffTune package which works with forward-mode auto-differentiation. Furthermore, end-to-end differentiable optimization enabled the coupling of the prediction and planning module in autonomous vehicles [100].

In the field of cosmology, JAX-Cosmo [31] is a recently developed end-to-end GPU accelerated library for cosmological calculations. Using automatic differentiation, JAX-COSMO exposes derivatives for certain cosmological quantities, and enables previously impractical methods such as Hamiltonian Monte Carlo and Variational Inference. Furthermore, by embedding the cosmological algorithms in JAX, the algorithms can be run on accelerated hardware, and can benefit from automatic code optimizations and techniques such as just-in-time compilation.

Embedding classical operators in neural networks has been demonstrated as an effective technique for end-to-end learning based on auto-differentiation. In [144] the tomographic backprojection operator is embedded as an algorithm within a neural network. Here the backprojection parameters are not trainable themselves, but rather the algorithm introduces prior knowledge about image reconstruction in the neural network. From the field of seismic imaging, a 3-step reflective seismic imaging method is learned end-to-end by turning the Delay-And-Sum (DAS) operator into a network layer in [186]. Similarly, the wave-physics-formation algorithm is placed between two neural networks in [187] to facilitate single-plane seismic wave imaging (SFW).

Automatic differentiation techniques have already been applied to solve specific problems in computed tomography in recent years. In [231], beam hardening correction for X-ray microscopy of mouse bones is performed with a polynomial correction model that is optimized through a PyTorch-based differentiable FDK algorithm. An elaborate outline of algorithmic differentiation for phase retrieval is presented in [105]. In [108, 160], automatic differentiation is used in ptychography where the object wave function is obtained with a gradient-based method by minimizing the ptychography loss for each pixel. In [55] the 3D reconstruction problem for objects beyond depth-of-focus (DOF) is formulated as a minimization problem with a data fidelity and total variation term, which is solved with a gradient descent algorithm. How automatic differentiation techniques can be used for different imaging modalities is shown in [77], where compressive sensing, single image super resolution (SISR), and ptychography reconstructions are covered. For tomography, the reconstruction is obtained with a gradient based approach by minimizing a total variation functional, and the authors show how this is beneficial for sparse and limited angle data.

In the field of nanotomography, the auto-differentiation framework Adorym [54] for flexible reconstruction has been developed. In Adorym, a flexible forward model allows for optimization of experimental parameters such as probe position, object tilt, absorption/refraction relation coefficient, and propagation distance. The authors show improved reconstruction quality for ptychography and multi-distance holography reconstructions, by finetuning these experimental parameters. For conventional CT, they accelerate ART with gradient descent optimizers and acquire improved results over FBP.

Compared to the related work outlined above, the scope and focus of the present paper is more general. We focus on the core design principles underlying CT workflows using auto-differentiation and demonstrate the efficacy of the approach through a varied set of four case studies.

3.3 Methodology

3.3.1 Auto-differentiation

For the case studies included in this paper, we implement the CT workflows in the auto-differentiation framework PyTorch [181]. An auto-differentiation system breaks down a program in a series of primitive operations for which fixed procedures are known to compute derivatives. The series of primitive computations is collected in a computational graph. Most auto-differentiation systems, including PyTorch, trace the computational graph implicitly during the forward computation through the program. After the forward computation, the error or loss is computed. Here, we use gradient-based optimization, meaning that an auto-differentiation package needs to obtain partial derivatives of the loss with respect to the parameters.

Often each primitive function in an auto-differentiation package specifies vector-Jacobian products. Suppose two quantities are related by a primitive function $f(\mathbf{x}) = \mathbf{y}$, and $\mathbf{y}$ is used further in the computation. Denote by $\bar{\mathbf{y}}$ the derivative of a loss L with respect to $\mathbf{y}$. A vector-Jacobian product defines a way to express the derivatives of L with respect to $\mathbf{x}$, and is defined as

$$\frac{\partial L}{\partial x_j} = \sum_i \frac{\partial y_i}{\partial x_j} \frac{\partial L}{\partial y_i}, \qquad \bar{\mathbf{x}} = J^T \bar{\mathbf{y}},$$

where J is the Jacobian. For a primitive operation f, the gradient of the input $\bar{\mathbf{x}}$ can be calculated from the output gradient $\bar{\mathbf{y}}$, the input $\mathbf{x}$ and the output $\mathbf{y}$. For example, in the case of $y = f(x) = -x$ the gradient of the input can be computed as $-\bar{y}$, for $y = f(x) = e^x$ it will be $y \cdot \bar{y}$, and in the case of $y = \log(x)$ it will be $\frac{\bar{y}}{x}$. The procedure of computing gradients in reverse (starting from $\bar{L} = 1$) is called back propagation. For gradient descent, parameters are updated by

$$\mathbf{x} \leftarrow \mathbf{x} - \lambda \bar{\mathbf{x}},$$

for some step size λ (often called the learning rate in deep learning). Variants of gradient descent exist such as Nesterov's accelerated gradient descent [162]. In our experiments we will either use (stochastic) gradient descent, or Adam [113].

A drawback of auto-differentiation frameworks can be excessive memory consumption because copies of many intermediate outputs are stored for fast computation of the back propagation algorithm. This problem can be addressed by several general approaches. Gradient checkpointing stores only a subset of the intermediate outputs for gradient computation [35]. Another approach involves just-in-time (JIT) compilation where a compiler attempts to optimize the computational graph into a more memory and compute efficient set of instructions. In this work, we apply PyTorch implementations of both techniques for certain memory or computationally expensive operations.

3.3.2 Computed tomography

In computed tomography [106] a 3D image of an object is recovered from a series of projections that are taken at different angles. Tomographic reconstruction can be modelled as the problem to recover an object volume $\mathbf{x} \in \mathcal{X} := \mathbb{R}^{N_x \times N_y \times N_z}$ from the measured projection data $\mathbf{y} \in \mathbb{R}^{N_\theta \times N_u \times N_v}$. Here, N_u and N_v are the number of detector rows and columns, and N_θ is the number of projection angles. The projection process can be approximated by a linear operator A, and can be expressed as a matrix using the aforementioned discretization

$$A\mathbf{x} = \mathbf{y}, \tag{3.1}$$

where $\mathbf{x}$ and $\mathbf{y}$ are collapsed to a vector. For parallel beam tomography the object can be reconstructed by the commonly used filtered backprojection algorithm (FBP)

$$\mathbf{x}_{\mathrm{FBP}} = A^T(\mathbf{h} * \mathbf{y}). \tag{3.2}$$

Here, the projection data is convolved with a 1D filter $\mathbf{h} \in \mathbb{R}^{N_v}$ (Ram-Lak in this study), and subsequently backprojected by applying A^T. For circular cone-beam tomography, the object can be recovered by the Feldkamp-Davis-Kress (FDK) [59] algorithm, where the projection data is weighted in order to compensate for the diminishing intensity at distance from the detector center. An alternative to direct methods are iterative methods that reformulate the equation system 3.1 as an optimization problem of the form

$$\mathbf{x}^* = \underset{\mathbf{x} \in \mathcal{X}}{\arg\min} \|A\mathbf{x} - \mathbf{y}\|_2^2. \tag{3.3}$$

Variational methods additionally aim to incorporate prior knowledge in the form of a regularization term $R(\mathbf{x})$ to the functional, e.g.,

$$\mathbf{x}^* = \underset{\mathbf{x} \in \mathcal{X}}{\arg\min} \|A\mathbf{x} - \mathbf{y}\|_2^2 + R(\mathbf{x}). \tag{3.4}$$

3.3.3 CT workflows

The CT workflows that we consider here contain input data in the form of projection images, contain several data processing steps of which at least one is a reconstruction step, and contain an objective function that scores the final result. We consider potentially non-sequential workflows, i.e., the data processing graph can contain several branches, or cyclical sections. The computational blocks can contain parameters that we want to update in order to minimize the objective function (see Figure 3.1). Those parameters must be real or complex numbers in order to facilitate gradient-based optimization.

For the four case studies, we have implemented the computational blocks to be differentiable with respect to the learnable parameters. Non-differentiable steps can potentially be replaced by a differentiable approximation function. For

example, to make thresholding (and segmentation) differentiable, we implement the thresholding operation using a hyperbolic tangent

$$\tau_\gamma(\mathbf{x}, t) = \frac{1}{2}\left(1 + \tanh\left(\gamma\left(\frac{\mathbf{x}}{t} - 1\right)\right)\right), \tag{3.5}$$

where the volume $\mathbf{x}$, and threshold t have been scaled to $[0, 1]$. The parameter γ regulates the sharpness of the clipping.

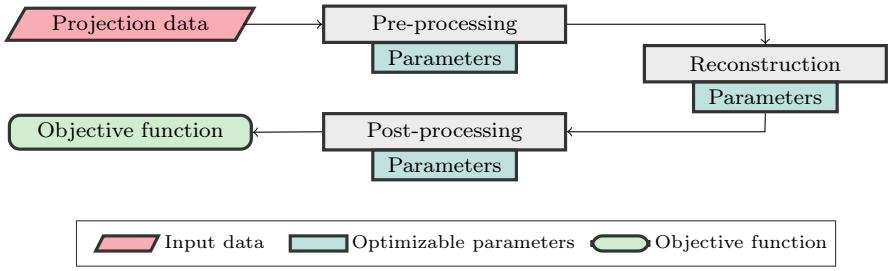


Figure 3.1: Example CT workflow diagram.

3.3.4 Software implementation

To compute gradients end-to-end for workflows that contain tomographic (back)projection operators, we will make use of the matrix identity $\nabla A\mathbf{x} = A^T\nabla\mathbf{x}$. Tomographic projection operations are implemented in the ASTRA toolbox [1] in a computationally efficient GPU accelerated manner. The tomosipo package [90] implements PyTorch support for ASTRA, and contains projection operators that propagate gradients in PyTorch using the aforementioned identity. Here, we use tomosipo projection operators in our workflows to propagate gradients end to end. In addition to reconstruction algorithms, the workflows we construct in this work contain other classical CT algorithms, such as Paganin's phase retrieval, phase contrast projection, and spectral projectors. We implemented these algorithms in PyTorch which facilitates auto-differentiation, and makes them GPU compatible, and the code is made publicly available for all case studies [216].

3.4 Case studies

3.4.1 Rotation axis alignment

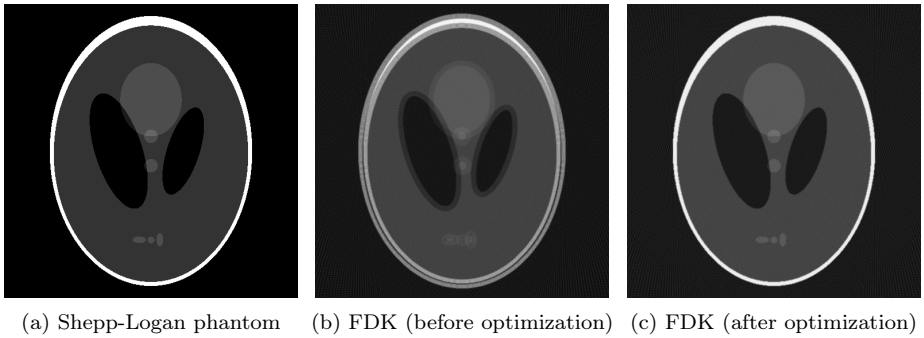
Introduction

In the first case study, we consider the problem of alignment in tomographic reconstruction [123]. Various experimental factors can introduce errors in the geometric parameters that are used to perform the reconstruction. A common occurrence of this effect in CT is a misalignment of the rotation axis position. In

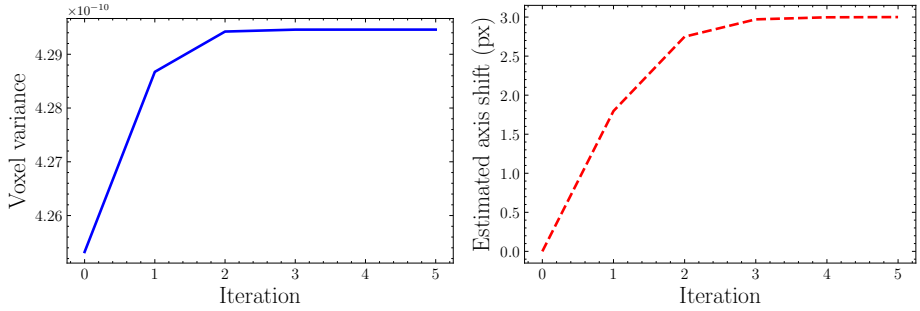
the case of a rotation axis misalignment $\boldsymbol{\delta}$, for a ray parametrized by angle $\boldsymbol{\omega}$ and position vector $\boldsymbol{a}$ from scanning set Γ, the projection p of the object function $f : \mathbb{R}^3 \to \mathbb{R}$ is given by the Radon transform as

$$p(\boldsymbol{a}, \boldsymbol{\omega}) = \int_0^\infty f(\boldsymbol{a} + \boldsymbol{\delta} + t\boldsymbol{\omega})\, dt, \quad \boldsymbol{\omega} \in \mathbb{S}^2, \quad \boldsymbol{a} \in \Gamma. \tag{3.6}$$

This discrepancy (most prominently a lateral shift) between the assumed and the true rotation axis position introduces severe artifacts in the resulting reconstruction, which appear differently depending on the acquisition geometry. In cone-beam CT this leads to blurring and "doubling" of object features, which significantly affect the sharpness of the reconstruction.



(a) Shepp-Logan phantom (b) FDK (before optimization) (c) FDK (after optimization)



(d) Contrast measure during optimization (e) Rotation axis shifts during optimization

Figure 3.2: (a) Shepp-Logan phantom and its FDK reconstructions obtained from projection data simulated using (b) shifted rotation axis and (c) rotation axis position compensated by the proposed gradient-based optimization method. (d) Image variance-based contrast measure and (e) estimated rotation axis shift during the optimization.

Experiments

Here we propose to optimize for the rotation axis position by minimizing the magnitude of misalignment-induced artefacts as measured by an appropriate measure

of reconstruction quality [50] [76]. In cone-beam CT, to account for edge blurring and "doubling" artifacts, we use a well-known image contrast measure based on voxel variance [73]. The intuition behind this metric is that a sharply focused reconstruction has a large spread in the intensity histogram, whereas upon blurring intensities concentrate around the same gray value. To enable gradient-based optimization for this task, we adjust the lateral shift of the assumed rotation axis position by shifting all projection images in the opposite direction using a bicubic interpolation-based image shift operator implemented in PyTorch. The shifted projections are then backprojected and the quality of this intermediate result is quantified. Since all the pieces of the described workflow are implemented in a differentiable manner within PyTorch, a generic gradient-based optimization algorithm (SGD) can now be used to find the shift of the rotation axis that maximizes the reconstruction quality. A diagram of the proposed pipeline is shown in Figure 3.3. Both experiments in this section were performed on a workstation with 64GB RAM, an NVidia GeForce GTX 1070 GPU, and Intel i7-7700K CPU.

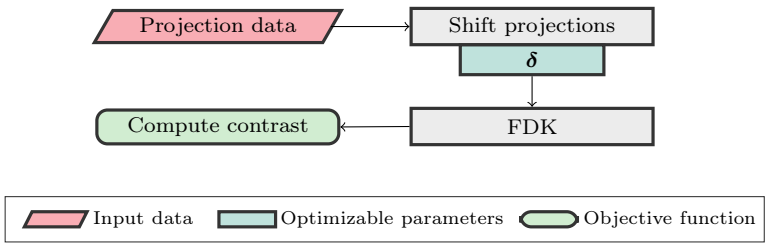


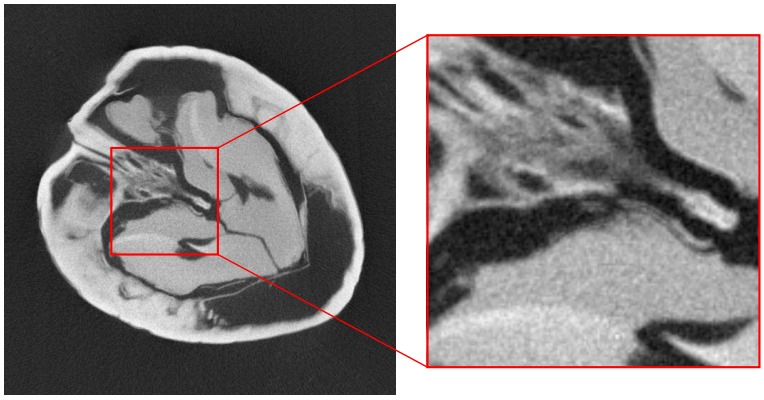
Figure 3.3: CT workflow for self-supervised rotation axis alignment.

Simulated experiment: Shepp-Logan phantom

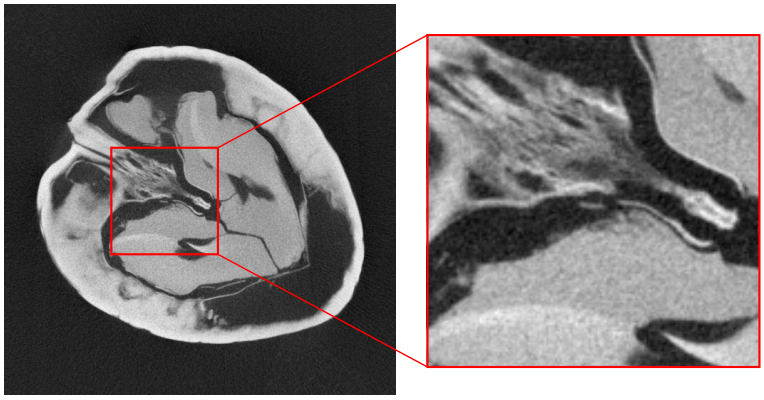
To test the proposed gradient-based rotation axis alignment method, we first employ simulated data based on a 512×512 px Shepp-Logan phantom (Figure 3.2(a)) that is projected using a geometry with a rotation axis shift of 3 px. As can be observed in Figure 3.2(b), even a minor misalignment in the axis position introduces significant blurring in the reconstruction. The proposed gradient-based contrast optimization method successfully compensates the rotation axis shift (Figure 3.2(c-e)), converging in only about 3 iterations. The proposed method is computationally efficient as running 6 iterations took 97 ms on our system.

Real-world experiment: High-resolution cone-beam CT of a walnut

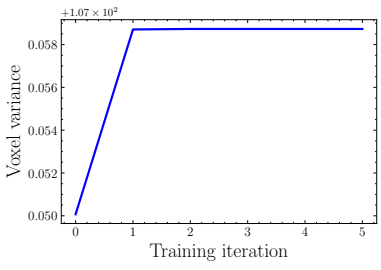
Next, we evaluate the performance of the proposed rotation axis shift compensation method on experimental data using an open dataset of high-resolution cone-beam CT of walnuts acquired at the Flex-ray lab [39, 119]. Figure 3.4(a) demonstrates an FDK reconstruction obtained with the geometry parameters specified in the dataset metadata, and Figure 3.4(b) shows the reconstruction after axis alignment method



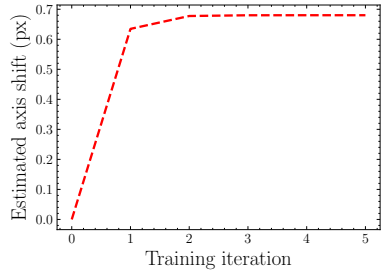
(a) FDK (before optimization)



(b) FDK (after optimization)



(c) Contrast measure during optimization



(d) Rotation axis shifts during optimization

Figure 3.4: (a) FDK reconstruction of the walnut dataset. (b) FDK reconstruction after rotation axis position was compensated by the proposed gradient-based optimization method. (d) Image variance-based contrast measure and (e) estimated rotation axis shift during the optimization.

has been applied. It can be observed that although there are no obvious artifacts present in the initial reconstruction (which makes this kind of misalignment easy to miss with manual inspection), the rotation axis position optimization significantly improves the resolution of the reconstruction. The improved resolution brings up details that were not visible before, such as pores in the central part of the walnut that can be seen in the zoom-in of Figure 3.4(b). The walnut slice is 550×550 px, with 501 projection angles, and running 6 iterations took 295 ms on our system.

To summarize, our workflow for rotation axis position alignment, implemented in an automatic differentiation framework, results in an intuitive formulation of the axis alignment problem. In addition, the resulting method enjoys a quick convergence, can retrieve sub-pixel axis shifts in a straightforward manner, and can be easily extended to multivariate optimization if other geometry parameters, such as rotation axis tilt and the cone angle of illumination, are included in the workflow.

3.4.2 Phase retrieval

Introduction

In the second case study we apply an end-to-end optimization approach to phase contrast imaging (PCI) [57]. PCI can reach nanometric resolution in tomographic imaging [251], and requires an additional reconstruction step known as phase retrieval. In PCI, the image is reconstructed based on changes to the wave front due to the material that is present along the wave path. A widely used experimental PCI setup is phase propagation-based imaging where projections are acquired from several different distances using a coherent beam. Next, a phase retrieval algorithm is used to calculate phase maps.

Here, we will consider Paganin's phase retrieval algorithm [172], which requires the material refractive index δ, and attenuation β to be known. Paganin assumes a single material object, and retrieves the projected thickness of the object $\mathbf{T}$ by

$$\mathbf{T}(\mathbf{r}_\perp) = -\frac{1}{\beta} \log \left(\mathcal{F}^{-1} \left(\frac{\beta \mathcal{F}\{I(\mathbf{r}_\perp, z = R_2)\}/I_0}{R_2 \delta \|\mathbf{k}_\perp\| + \beta} \right) \right). \tag{3.7}$$

Here I is the intensity function, I_0 the incident intensity, $\mathbf{r}_\perp$ the position vector perpendicular to the optical axis, $\mathbf{k}_\perp$ the wave vector, and R_2 the source-detector distance. A common practice is to divide both the numerator and the denominator inside the inverse Fourier transform of equation 3.7 by β. The resulting fraction δ/β is sometimes designated as α. Expert users will often pick α manually to get a good reconstruction. However, this can be a time-consuming process, it is subjective, and it can make it more difficult for other researchers to reproduce results. Therefore, in this section we will show that we can optimize both β and δ in an unsupervised way for a clear objective. We do so by constructing a pipeline of operators that propagate gradients end-to-end.

Experiments

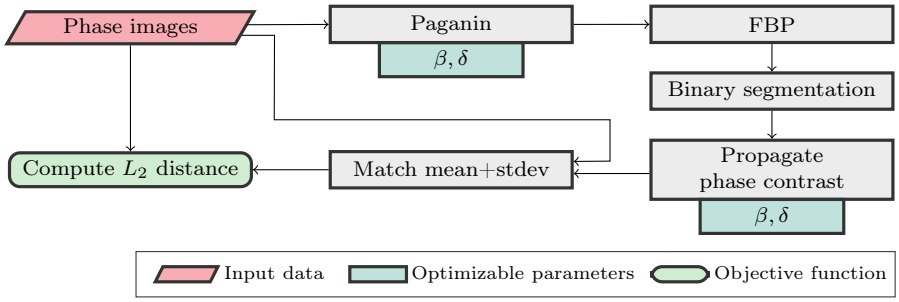
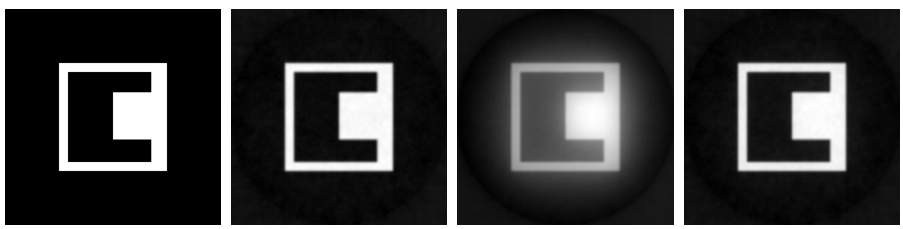


Figure 3.5: PCI pipeline for self-supervised optimization of β and δ.

In the experiments, we use a pipeline that takes raw projections as input, and performs Paganin phase retrieval to produce phase maps. A reconstruction based on the phase maps is made with filtered backprojection (FBP), and a binary segmentation is made with an implementation of Otsu's method [170] that makes use of equation 3.5. We use binary segmentation since Paganin assumes a single material.

After segmentation, using the same refraction and attenuation indices, projections are simulated based on the segmentation using a wave propagation projector. Finally, the simulated projections are scaled so that their mean and standard deviation align with the raw input. As a loss function we take the mean-squared error loss between the scaled simulated projections and the original input projections. A diagram of the pipeline is given in Figure 3.5. Both experiments in this section were performed on a server with 384GB RAM, an NVidia Titan RTX GPU, and two Intel Xeon Gold 6130 CPUs.



(a) $CaCO_3$ phantom (b) Reference FBP (c) FBP (before) (d) FBP (after)

Figure 3.6: (a) Slice through calcium carbonate simulated phantom. (b): FBP reconstruction using correct β and δ. (c): FBP of phase projections retrieved with the material indices for water that were used as initialization. (d): FBP of phase projections retrieved with learned material parameters after optimization.

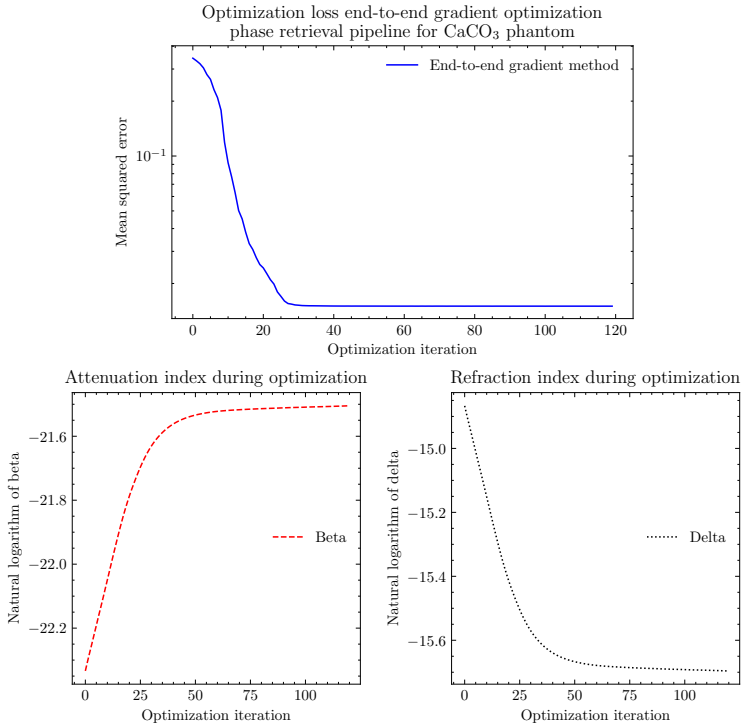
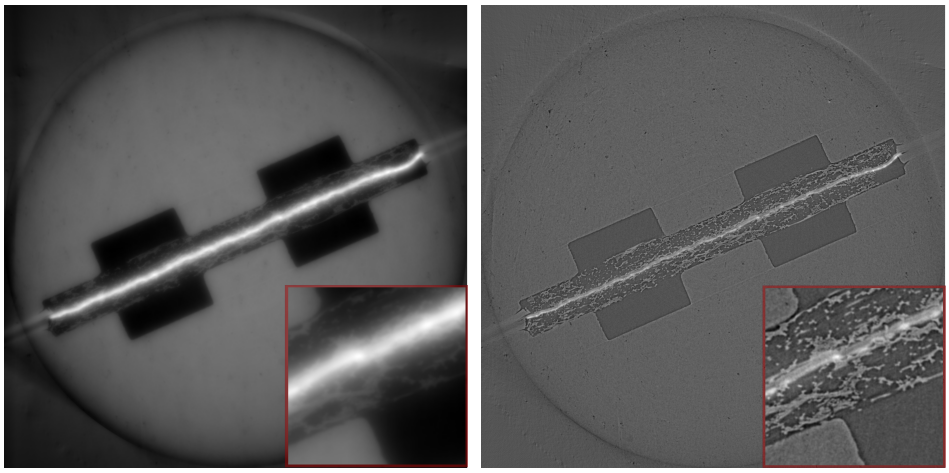


Figure 3.7: (Left): L2-loss between input phase maps and output simulated phase maps per iteration. Attenuation β (middle), and refraction δ (right) value per iteration.

Simulated experiment: Calcium carbonate cube

First, we perform a simulated experiment on a 3D phantom (192^3 volume, 572 angles) of a calcium carbonate hollow cube, with a smaller cube attached to one of its sides (see Figure 3.6(a)). Next, we simulate phase contrast images using the material parameters of calcium carbonate, and add Poisson noise. As reference, we show the FBP reconstruction of the phase maps acquired with Paganin with the correct β and δ in Figure 3.6(b). We initialize the attenuation and refraction indices to those of water. We use a gradient descent algorithm (Adam) optimization algorithm to update β and δ. For a smoother optimization we optimize these parameters on an exponential scale.

The FBP reconstructions before and after optimization are given in Figure 3.6. We see that the initial FBP reconstruction using water material indices has a halo artifact. This artifact is no longer visible after optimization. The optimization loss and the attenuation and refraction values (logarithm) per iteration are given in Figure 3.7. While the reconstruction quality has improved, the pipeline is not able to fully recover the original material index values; the final values after optimization



Before After

Figure 3.8: Hydrogen fuel cell [45] reconstructed with FBP with default material parameters for water (before), and after optimizing parameters end-to-end (after).

are $\ln(\beta) = -21.50$ and $\ln(\delta) = -15.69$, whereas the original values for calcium carbonate are $\ln(\beta) = -19.59$, $\ln(\delta) = -13.94$. However, the value of $\alpha = \delta/\beta$ is only 17.6% off the original. Running 120 iterations of optimization on this pipeline took 16 minutes and 48 seconds on our system.

Real-world experiment: Hydrogen fuel cell

We perform a real-world data experiment on a hydrogen fuel cell dataset acquired at the TOMCAT beamline of the Swiss Light Source (PaulScherrer Institut)[45]. The experiment is performed on a central slab of the full volume of size $160 \times 1001 \times 1476$ to reduce computation time. For the experiment we use the same setup as for the simulated experiment, and again initialize the attenuation and refraction indices to those of water. In Figure 3.8 we show reconstructed central slices of the fuel cell before and after optimization. We see that after optimization small-scale features such as bubbles are visible. The zoomed images show that the initial reconstruction is blurred, and the final reconstruction after optimization is much sharper. In Figure 3.9 we display the optimization loss and the attenuation and refraction values per iteration. Running 200 iterations of optimization on this pipeline took 1 hour and 38 minutes on our system.

Overall the experiments show that we can use a generic gradient based approach to optimize for reconstruction quality in an self-supervised way in phase contrast imaging. We showed on both a simulated and real-word dataset that the pipeline can be optimized for a clear objective, as opposed to manual selecting the $\alpha = \delta/\beta$ parameter. First, this reduces the work for operators as they no longer have to

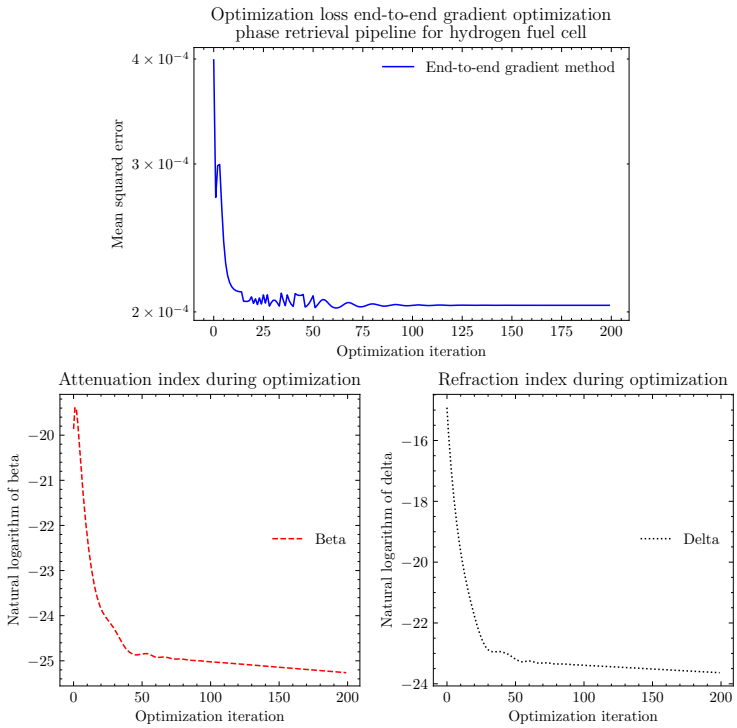


Figure 3.9: (Left): L2-loss between input phase maps and output simulated phase maps per iteration. Attenuation β (middle), and refraction δ (right) value per iteration.

tune parameters manually. Second, this makes the procedure more reproducible for different datasets.

3.4.3 Beam hardening correction

Introduction

For the third case study, we implemented self-supervised correction of artifacts introduced by beam hardening [23, 91]. The number of materials is denoted by N, with attenuation coefficients $\mu_n(E)$, and the beam spectrum has E_m energy bins with intensities I_e. Furthermore, let $l_{i,j}$ be the intersection length of ray i ($i = 1 \dots D$) with voxel j ($j = 1 \dots J$), d_j the relative density of voxel j, and $s_{n,j}$ a variable that is 1 if voxel j contains material n, and 0 otherwise. Then we can denote for a ray i and a material n the projected relative density $P_{i,n}$, and the monochromatic measured intensity by Beer-Lambert as

$$P_{i,n} = \sum_{j=1}^{J} l_{i,j} d_j s_{n,j}, I_{\text{mono},i} = I_0 e^{-\sum_{n=1}^{N} \mu_n(E_0) P_{i,n}}, \tag{3.8}$$

for a monochromatic beam with energy E_0 and intensity I_0. The measured polychromatic intensity (discrete) is given by

$$I_{\text{poly},i} = \sum_{e=1}^{E_m} I_e e^{-\sum_{n=1}^{N} \mu_{n,e} P_{i,n}}, \tag{3.9}$$

for a polychromatic beam with energies e and intensities I_e.

For a monochromatic X-ray, the attenuation coefficient μ is linearly related to the thickness of the object by Beer-Lambert's law. However, in practice X-ray beams are often polychromatic and this relation is no longer linear as lower energy photons get absorbed more than higher energy photons. Therefore, as a polychromatic beam travels through an object it "hardens", i.e., the average photon energy increases. If this effect is not taken into account by the reconstruction algorithm, it causes streaking and cupping artifacts because the lower absorbance due to higher average energy is mistakenly reconstructed as a lower density material.

Experiments

The aim of the case study is to highlight how combining an explicit forward model with a few lines of code and generic gradient-optimization can result in sophisticated algorithm construction, even for cases that used to require lengthy, hand-crafted implementations. In the experiments we will perform unsupervised beam hardening correction by learning the beam spectrum, and the energy-dependent attenuation coefficients per material. We will base the study on [66], where three different iterative unsupervised beam hardening correction algorithms are proposed and compared. We use the best performing algorithm in [66] as a comparison; the *iterative sinogram preprocessing* (ISP) method. The ISP method is an iterative scheme

with multiple steps, such as a bruteforce segmentation step, a local optimization step, and a reconstruction update step. For a detailed explanation of ISP we refer to [66]. ISP assumes that the number of materials is known beforehand, but no knowledge of the beam spectrum, or attenuation coefficients is assumed.

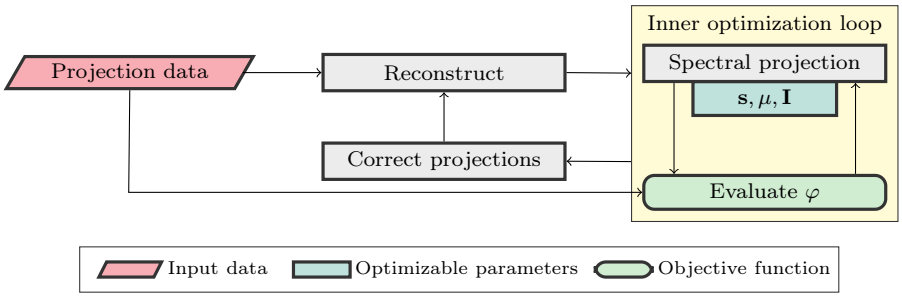


Figure 3.10: CT pipeline for self-supervised beam hardening correction.

As opposed to constructing a specialized algorithm such as ISP, we compare to a generic gradient-based approach where beam spectrum, attenuation coefficients, and thresholds are learned jointly. The generic approach is self-supervised using the same loss function as ISP

$$\varphi(\mu, \mathbf{I}, \mathbf{s}, \mathbf{d}) = \frac{1}{D} \sum_{i=1}^{D} \left(\log \left(\frac{I^{\text{meas}}_{\text{poly},i}}{I_0} \right) - \log \left(I^{\text{sim}}_{\text{poly},i} \right) \right)^2, \tag{3.10}$$

which is the L_2-loss between the simulated polychromatic projections and the original input projections. To initialize the thresholds we use Otsu's method on an initial reconstruction R_0 made with FBP or FDK. After the initialization we iterate as

1. Update segmentation thresholds, attenuation coefficients, and beam intensities,
$$\mathbf{s}^k, \mu^k, \mathbf{I}^k = \underset{\mathbf{s},\mu,\mathbf{I}}{\arg\min}\ \varphi(\mu, \mathbf{I}, g(R^{k-1}, \mathbf{s}), \mathbf{d} = 1), \tag{3.11}$$
where $g(R^{k-1}, \mathbf{s})$ is the differentiable thresholding function described before (equation 3.5), φ is the ISP loss function, and $\arg\min$ is performed with gradient descent.

2. Update the sinograms and corrected reconstruction R^k as for ISP.

A diagram of the workflow is shown in Figure 3.10. By using an implementation of the spectral projector that can propagate gradients, we can optimize the attenuation, intensity, and thresholds jointly. This improves the complexity of the algorithm as the thresholding step in ISP is exponential in the number of materials and computationally expensive. For our experiments, we created a PyTorch based implementation of ISP and our generic gradient-based approach. Both experiments

in this section were performed on a workstation with 64GB RAM, an NVidia RTX 2070 Super GPU, and AMD Ryzen 7 3800X CPU.

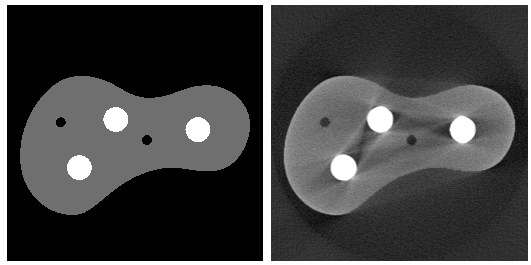


Figure 3.11: (Left): Barbapapa phantom consisting of PMMA filled with aluminium cylinders. (Right): FBP reconstructions of simulated spectral projections (with added Gaussian noise).

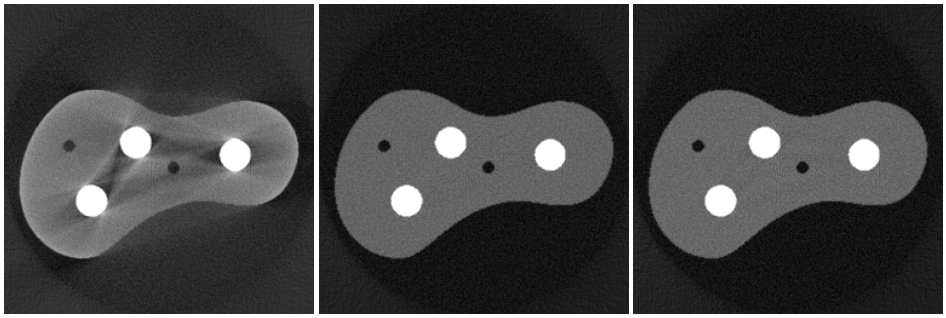
Simulated experiment

We perform a simulated experiment based on the Barbapapa phantom [66] where we created a simulated phantom (see Figure 3.11) of polymethyl methacrylate (PMMA) filled with aluminium rods with size 256×256. Next, we use a spectral projector with a simulated effective beam spectrum to simulate beam hardening artifacts. Finally, we add 2% Gaussian noise on the projections (see Figure 3.11 for reconstruction of noisy projections). To perform beam hardening correction we performed 40 steps of ISP. We ran the generic gradient-based approach for the same amount of objective function evaluations. In Figure 3.12 we show the FBP reconstructions of the corrected sinograms for both algorithms, and the accompanying material segmentations. We see that both methods reduced cupping and streaking artifacts. Furthermore, Figures 3.12(d) to 3.12(f) show a significant improvement in segmentation quality for both methods, which is close to the original phantom.

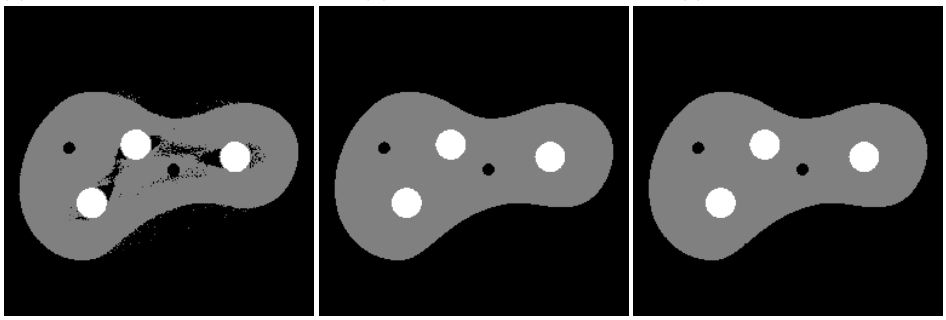
Line profiles for each of the three reconstructions are given in Figure 3.13(a). The line profiles confirm that the cupping artifacts are significantly reduced for both the PMMA material and the aluminium rods. In Figure 3.13(b) we plot the self-supervised loss for both methods. Both methods reach a similar optimum, but the combined gradient method shows slightly faster convergence. Note that the combined gradient method also runs twice as fast, which is due to the missing brute-force threshold selection step. This is also reflected in the runtimes; ISP ran for 16 minutes and 15 seconds, whereas the generic gradient-based approach ran for 8 minutes and 9 seconds.

Play-Doh foreign object X-ray CT dataset

We also evaluate both beam hardening correction methods on a real-world X-ray CT dataset of Play-Doh objects filled with stones [257, 258]. The experiment was

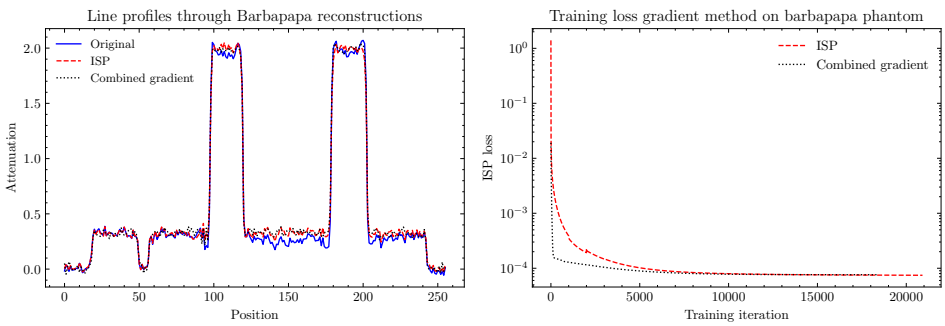


(a) Original FBP reconstruction (b) ISP reconstruction (c) CG reconstruction



(d) Original FBP segmentation (e) ISP segmentation (f) CG segmentation

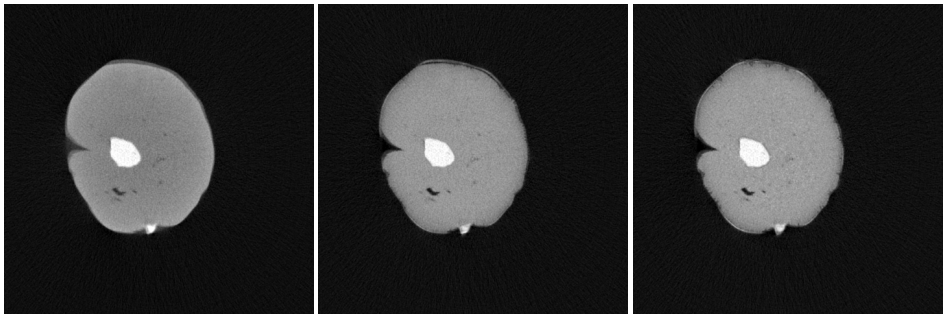
Figure 3.12: (Top): Barbapapa phantom reconstructed with the Filtered BackProjection (FBP) algorithm for ISP and a combined gradient-based (CG) approach. (Bottom): Material segmentations of corrected reconstructions.



(a) Line profiles (b) ISP loss

Figure 3.13: (a) Three line profiles through the original, ISP corrected, and combined gradient corrected Barbapapa reconstructions. (b) Optimization losses per φ evaluation.

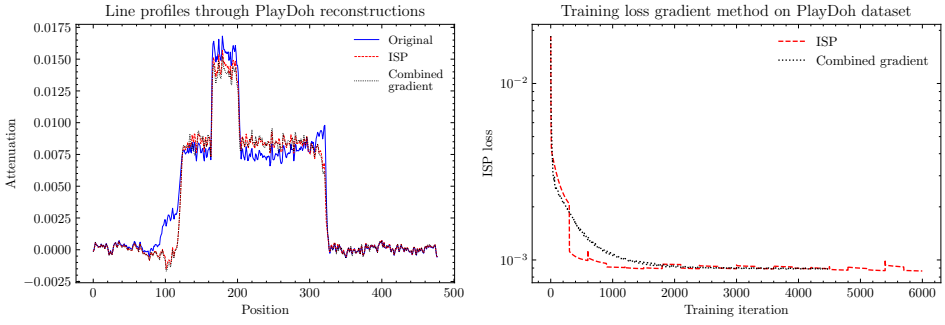
performed on a 478×478 central slice. The Play-Doh objects exhibit significant cupping artifacts. In Figure 3.14 we show the FBP reconstructions of the corrected



(a) Original FBP reconstruction (b) ISP reconstruction (c) CG reconstruction

Figure 3.14: Play-Doh and stones reconstructed with the Filtered BackProjection (FBP) algorithm for ISP and a combined gradient-based (CG) approach.

sinograms for both algorithms. We see that both methods reduced cupping, which is also clearly visible on the line profiles shown in Figure 3.15(a). In Figure 3.15(b) we plot the self-supervised loss for both methods. Both methods reach a similar optimum, but the combined gradient method shows a more stable convergence. The jumps in the loss curve for ISP happen when a new optimal set of thresholds is determined with brute-force calculation after the local minimization steps. Since the combined gradient method jointly optimizes all parameters every step, it does not suffer from such jumps. The improved computational efficiency is more prominent due to faster convergence of the local optimization step; ISP ran for 19 minutes and 51 seconds, whereas the generic gradient-based approach ran for 4 minutes and 35 seconds.



(a) Line profiles (b) ISP loss

Figure 3.15: (a) Three line profiles through the original, ISP corrected, and combined gradient corrected Play-Doh reconstructions. (b) Optimization losses per φ evaluation.

Overall the experiments show that by using a generic gradient-based approach for all parameters, we can create an algorithm that performs similar to the specialized

ISP algorithm in similar runtime. The construction of such an algorithm is more straightforward as we can create the forward model and embed the workflow in an auto-differentiation framework. This allows us to optimize end-to-end in a straight-forward manner.

3.4.4 Optimizing total variation reconstruction and neural networks end-to-end

Introduction

In the final case study we will focus on denoising CT reconstructions using convolutional neural networks (CNNs) jointly with total variation reconstruction (TV) [176, 184, 240]. Since neural networks are often large, black-box models that are hard to interpret, it can be desirable to let more computation steps be performed by interpretable algorithms and use a smaller network, rather than using simpler algorithms and a larger neural network. In the experiment we show how embedding CT algorithms in an auto-differentiation framework allows for easy interfacing with deep learning algorithms, and the pipeline can be trained end-to-end. This method allows us to design a more interpretable pipeline using variational methods with similar accuracy, while using a smaller neural network.

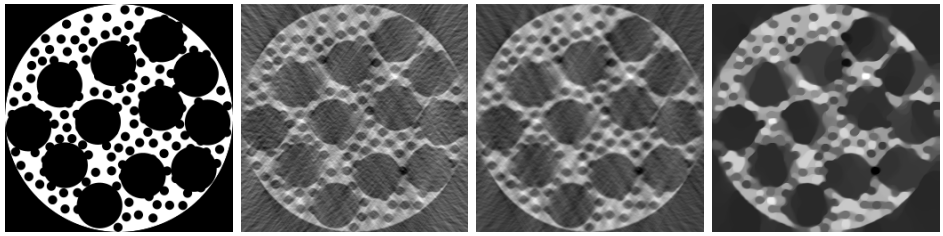
Total variation reconstruction is a commonly used variational method to incorporate prior knowledge that the gradient of the image should be sparse. For TV, the regularization term in equation 3.4 becomes the magnitude of the gradient

$$\mathbf{x}^* = \arg\min_{\mathbf{x}} \left\{ \|A\mathbf{x} - \mathbf{y}\|_2^2 + \lambda \|\mathbf{x}\|_{\mathrm{TV}} \right\} = \arg\min_{\mathbf{x}} \left\{ \|A\mathbf{x} - \mathbf{y}\|_2^2 + \lambda \|\nabla \mathbf{x}\|_1 \right\}. \tag{3.12}$$

To minimize the functional we use an implementation based on Chambolle-Pock [224]. The regularization parameter λ controls the trade-off between data fidelity and regularization term. A small λ will result in a reconstruction $\mathbf{x}^*$ that is close to the raw data, but potentially with high noise. A large λ can lead to less noise and more connected components of equal value. For more information on total variation regularization we refer to Rudin-Osher-Fatemi [203].

Experiments

We perform an experiment on simulated 2D foam-like phantoms of size 256×256. These phantoms contain both large and small scale features (see Figure 3.16(a)). We simulate parallel beam projections from an angular range of -60° to 60°, creating missing wedge artifacts, and add Poisson noise to the projection data (see Figure 3.16(b) for an example FBP reconstruction). For TV reconstruction (Figures 3.16(c) and (d)), this creates a situation where a small λ keeps smaller features, but creates a high noise reconstruction, while a larger λ removes more noise but creates connected components of voxels which removes smaller features. We created a training dataset of 100 randomly generated target phantoms, and corresponding noisy limited angle projections. In our experiments we in each case have the



(a) Phantom (b) FBP reconstruction (c) TV ($\lambda = 10^{-5}$) (d) TV ($\lambda = 10^{-2}$)

Figure 3.16: (a) Foam CT binary phantom, (b) FBP reconstruction of phantom projections with added severe noise, (c) TV reconstruction with $\lambda = 10^{-8}$, and (d) TV reconstruction with $\lambda = 10^{-2}$.

noisy projections as input data, and use the original phantom as target data. Our implementation of total variation reconstruction uses PyTorch primitives, and λ can therefore be optimized end-to-end with gradient-based approaches in conjunction with deep learning algorithms. In total, we trained four pipelines end-to-end.

1. **FBP + Small CNN:** An FBP reconstruction is input for a small CNN that denoises the image. The CNN consists of 3 layers of 3×3 kernels arranged in 1×64, 64×32, and 32×1 channels (19.393 parameters in total). Only the CNN weights are learned.

2. **Single TV + Small CNN:** A single TV reconstruction is input for a small CNN that denoises the image. The CNN consists of 3 layers of 3×3 kernels arranged in 1×64, 64×32, and 32×1 channels (19.393 parameters in total). Both λ and the CNN weights are learned jointly. λ is initialized at $\lambda = 10^{-3}$.

3. **Double TV + Small CNN:** Two TV reconstructions with different λ are input for a small CNN that uses both inputs to create a single denoised output image. The CNN consists of 3 layers of 3×3 kernels arranged in 2×64, 64×32, and 32×1 channels (19.969 parameters in total). Both λ_1, λ_2, and the CNN weights are learned jointly. The λ's are initialized as $\lambda_1 = 10^{-3}$, and $\lambda_2 = 10^{-8}$.

4. **FBP + Large CNN:** An FBP reconstruction is input for a larger CNN that densoises the image. The CNN consists of 4 layers of 3×3 kernels arranged in 1×160, 160×96, 96×64, and 64×1 channels (195.873 parameters in total). Only the CNN weights are learned.

A diagram of the Double TV + Small CNN pipeline is shown in Figure 3.17. All 4 experiments are run for an equal number of training iterations, and were performed on a workstation with 64GB RAM, an NVidia RTX 2070 Super GPU, and AMD Ryzen 7 3800X CPU. For validating the results we generated an additional random phantom that was not in the training set. The resulting denoised validation reconstructions are shown in Figure 3.18.

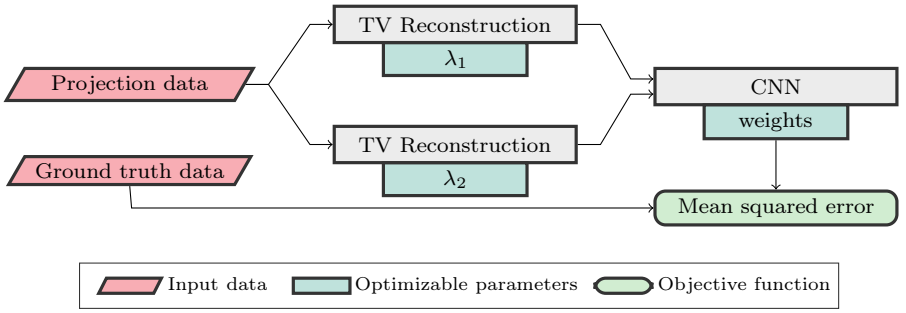


Figure 3.17: Pipeline for supervised denoising of CT data.

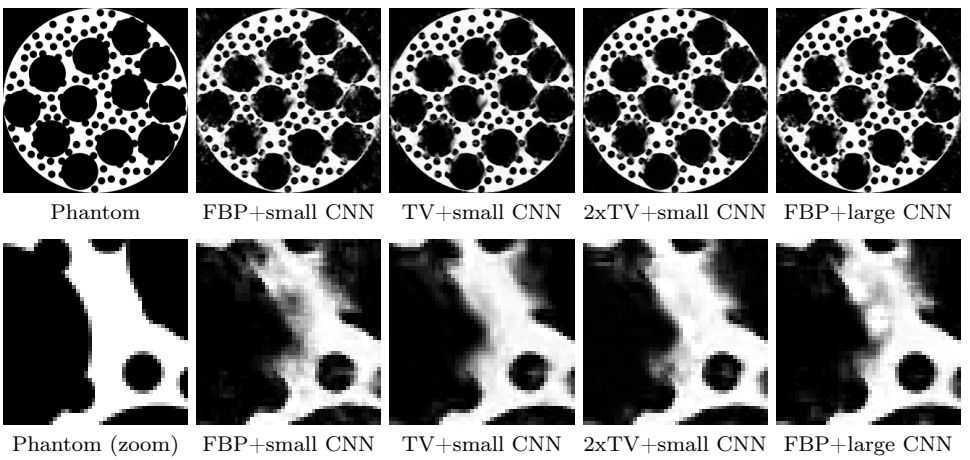
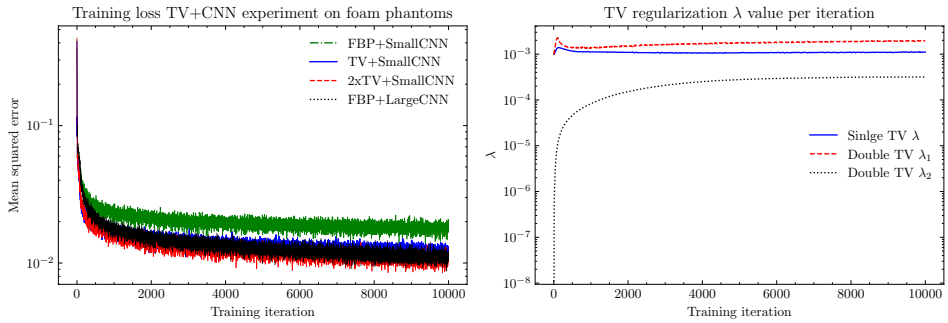


Figure 3.18: (Top): From left to right; Foam phantom, output of FBP reconstruction followed by a small CNN, output of a single total variation reconstruction followed by a CNN, output of two total variation reconstructions followed by a CNN, output of FBP reconstruction followed by a larger CNN. (Bottom): Zoom of top row.

We see that all pipelines can denoise the reconstruction significantly, but struggle to remove artifacts introduced by the missing angular information. The FBP+SmallCNN pipeline seemingly performs worse on visual inspection of the zoomed images. Arguably, both TV pipelines perform better than FBP+LargeCNN, even though the larger CNN had 10 times more parameters. This suggests that the prior knowledge incorporated by total variation makes for an easier denoising problem for the CNN. It is unclear whether the addition of a second total variation operator benefited the denoising quality. However, in validation loss the double total variation operator performs better with $1.805 \cdot 10^{-2}$ loss versus $1.864 \cdot 10^{-2}$ for the single TV operator. The validation loss for the FBP+SmallCNN pipeline is $2.322 \cdot 10^{-2}$, and for the FBP+LargeCNN pipeline $2.268 \cdot 10^{-2}$.



(a) Training losses (b) λ values

Figure 3.19: (a) Mean squared error loss during training. (b) TV regularization λ values during training.

We plot the training losses and λ values during training in Figure 3.19. An interesting observation is that single TV learned a λ with a value in between λ_1 and λ_2 of the double TV pipeline. We hypothesize that the single TV pipeline had to compromise λ between leaving small features intact, and denoising the reconstruction.

Running total variation in a training procedure comes at large computational cost; a single training step (100 images) took 99.6, and 198.3 seconds for single and double TV respectively whereas FBP + large CNN took 1.82 seconds. For 3D it could therefore be infeasible to train this pipeline. For single scan tuning of λ a surrogate TV approach can be considered during optimization, such as [118]. Alternatively, λ could be tuned on the central slice for 3D cases.

Overall the experiment shows how embedding classical CT algorithms, such as TV reconstruction, in GPU accelerated auto-differentiation frameworks allows for the easy prototyping of mixed classical and deep learning pipelines. We were able to replace a large CNN with 200 thousand parameters by a stack of two TV operators to achieve better denoising accuracy, and reduce the CNN size to 20 thousand parameters. In addition, the total variation pipelines are more interpretable as the behaviour of TV for small or large λ is well understood. In general, this easy interfacing of deep learning and classical methods enables end-to-end learning of parameters, which opens up new areas of research. Additionally, the resulting pipeline may be more interpretable since parameters of classical algorithms are often linked to physical or mathematical concepts that are better understood.

3.5 Discussion

Our four use cases and the corresponding experiments demonstrate that a broad range of CT workflows can be implemented as end-to-end optimized pipelines using auto-differentiation. Depending on the particular use case, this approach

yields several benefits, which we will now discuss in more detail. When all pieces of a data processing pipeline facilitate gradient propagation, parameters at all steps of the pipeline can be optimized jointly for a criterion calculated at any given step. In combination with explicit quality criteria this allowed us to design workflows where the learnable parameters were used in the earlier stages of the pipeline, while the objective function was more naturally defined at the end of the pipeline. For example, in both the rotation axis alignment and beam hardening correction experiments we were able to optimize parameters that are used in the projection domain for metrics computed in the volume domain. Another benefit is that this approach allows for efficient automatic optimization of parameters that may otherwise be chosen manually. This additionally improves the transferability of CT workflows when applied to new data as the parameters can be optimized in an objective manner using the same quality criterion.

Our approach made it possible to implement workflows more efficiently by using existing building blocks, and by defining the workflow in a declarative manner, i.e., implementing the forward model and then optimizing its parameters with a generic off-the-shelf optimizer. Creating workflows in this manner is typically less time consuming to develop and more flexible compared to specialized methods. For example, in the beam hardening experiment we showed that a gradient descent on the forward model of the physical effect results in a quality comparable to a specialized correction method.

Using our design principles allowed for seamless compatibility between classical and deep learning-based approaches. As classical approaches often come with an intuition of their behaviour, this combination of classical algorithms and deep learning can lead to more robust and interpretable workflows. In the last experiment we showed that using existing classical algorithms in conjunction with deep learning can create workflows that perform similarly to purely deep learning based approaches while using smaller neural networks.

Even though gradient-based end-to-end optimization has several potential benefits, our approach can create certain practical challenges. First, convergence of a gradient descent optimizer is not always guaranteed, and can potentially produce suboptimal solutions. Second, the learning rate needs to be picked manually, which is especially challenging when the involved parameters have significantly different magnitudes. Despite the disadvantages, as we have seen in the field of deep learning, this approach often still produces strong results in practice, and well-performing heuristic solutions to mathematically complicated problems were found.

3.6 Conclusion

We have shown how implementing classical CT algorithms in an auto-differentiation framework can improve their transferability and interpretability, enable efficient automatic end-to-end optimization, and allow for solving real-world problems without having to develop specialized algorithms. We have explored four use cases experimentally: rotation axis alignment, phase contrast imaging, beam hardening

correction, and end-to-end denoising with deep learning and total variation reconstruction, demonstrating that a wide range of CT workflows can be implemented in such a framework. In the future, the key benefits demonstrated in this paper can be utilized in computational toolboxes that leverage auto-differentiation for improved construction and execution of advanced CT workflows.

4

LEAN: GRAPH-BASED PRUNING FOR CONVOLUTIONAL NEURAL NETWORKS BY EXTRACTING LONGEST CHAINS

4.1 Introduction

In recent years, convolutional neural networks (CNNs) have become state-of-the-art for many image-to-image translation tasks [121], including image segmentation [202], and denoising [232]. They are increasingly used as a subcomponent of a larger system, e.g., visual odometry [254], as well as in energy-limited and real-time applications [255]. In these situations, the applicability of high-accuracy CNNs may be limited by large computational resource requirements. Small networks may be more applicable in such settings, but may lack accuracy.

Neural network pruning [109, 156] has recently gained popularity as a technique to reduce the size of neural networks [18]. Neural networks consist of learnable parameters, including the scalar components of the convolutional filters. When pruning, the neural network is reduced in size by removing such scalar parameters while trying to maintain high accuracy. We distinguish between individual parameter pruning [80], where each parameter of an operation is ranked and pruned separately, and structured pruning [125, 142], where entire convolutional filters are ranked and pruned. As convolution operators can only be removed once all scalar parameters of the filter kernel have been pruned, structured pruning is favored over

individual pruning when aiming to improve computational performance [178]. In the remainder of this paper, we focus on structured pruning.

Although structured pruning methods take into account the division of a neural network into operations, they do not take into account the fact that the output of the network is formed by a *sequence* of such operations. This has two drawbacks. First, since the relative scaling of individual convolutions may vary without changing the output of the whole chain, pruning methods that prune individual operators could potentially prune a suboptimal set of operators from the chain. Second, to significantly reduce evaluation time, a severe pruning regime must be considered, i.e., a pruning ratio (percentage of remaining parameters after pruning) of 1–10%. In this regime, pruning can result in network disjointness, i.e., the network contains sections that are not part of some path from the input to the network output. Some existing pruning methods take into account network structure to a limited degree [206]. In practice, however, these methods do not contain safeguards to avoid network disjointness.

In this paper, we present a novel pruning method called LongEst-chAiN (LEAN) pruning, which as opposed to conventional pruning approaches uses graph-based algorithms to keep or prune chains of operations collectively. In LEAN, a CNN is represented as a graph that contains all the CNN operators, with the operator norm of each operator as edge weights. We argue that strong subnetworks in a CNN can be discovered by extracting the longest (multiplicative) paths, using computationally efficient graph algorithms. The main focus of this work is to show how LEAN pruning can significantly improve the computation speed of CNNs for real-world image-to-image applications, and obtain high accuracy in the severe pruning regime that is difficult to achieve with existing approaches.

This paper is structured as follows. In Section 4.2, we explore existing pruning approaches. In Section 4.3, we outline the preliminaries on CNNs, pruning filters, and the operator norm. Next, in Section 4.4, we introduce LEAN pruning and describe how to calculate the operator norm of various convolutional operators. We discuss the setup of our experiments in Section 4.5. In Section 4.6, we demonstrate the results of the proposed pruning approach on a series of image segmentation problems and report practically realized wall time speedup. Our final conclusions are presented in Section 4.7.

4.2 Related work

Reducing the size of neural networks by removing parameters has been studied for decades [82, 109, 156]. Several works take into account the structure of the network to some degree. In [130] filters are pruned at runtime based on the feature maps [130]. Alternatively, one can prune entire channels [87], or decide which channels to keep so that the feature maps approximate the output of the unpruned network over several training examples [142]. In recent work, a graph is built for each convolutional layer, and filters are pruned based on the properties of this graph [245]. In [206] a neural network is represented as a graph and interdependencies

are determined using the Ising model.

Many pruning approaches are aimed at reducing neural network size with little accuracy drop [53, 86, 154, 261], as opposed to sacrificing accuracy in favor of computation speed. These approaches rarely exceed a pruning ratio of 12–50% [18, 131, 142]. When a high pruning ratio is used, e.g., a range of 5–10% [130, 138], a significant drop in accuracy is observed. Pruning ratios of 2–10% can be achieved with an accuracy drop of 1-3% by learning-rate rewinding [198]. However, the reduction in FLOPs was less substantial (1.5–4.8 times). In [256] severe pruning ratios of up to 1% have been considered, but the approach achieved limited improvements in terms of FLOPs reduction compared with existing pruning methods.

Criteria for deciding which elements of a neural network to prune have been extensively studied. A parameter's importance is commonly scored using its absolute value. Whether this is a reasonable metric has been questioned [122]. Singular values (which determine certain operator norms) have been used to compress network layers [46] and to prune feed-forward networks [3]. Efficient methods for the computation of singular values have been developed for convolutional layers [223]. Furthermore, a definition of ReLU singular values was proposed recently with an accompanying upper bound [49].

4.3 Preliminaries

4.3.1 CNNs for segmentation

A common image to image translation task is semantic image segmentation. The goal of semantic image segmentation is to assign a class label to each pixel in an image. A segmentation CNN computes a function $f : \mathbb{R}^{m\times n} \to [0,1]^{k\times m\times n}$, which specifies the probability of each pixel being in one of the k classes for an $m \times n$ image.

CNNs are composed of layers of operations which pass images from one layer to the next. Every operation, e.g., convolution, has an input $\mathbf{x}$ and output $\mathbf{y}$. The input and output consist of one or more images, called channels. For clarity, we distinguish throughout this paper between an **operation**, which may have several input and output channels, and an **operator**, which computes the relation between a single input channel and a single output channel. For instance, in a convolutional operation with input channels $\mathbf{x}_1, \ldots, \mathbf{x}_N$, an output channel $\mathbf{y}_j$ is computed by convolving input images with learned filters

$$\mathbf{y}_j = \left(\sum_{i=1}^{N} h_{ij} * \mathbf{x}_i \right) + b_j. \tag{4.1}$$

Here h_{ij} is the filter related to the convolution operator that acts between channel $\mathbf{x_i}$ and $\mathbf{y_j}$, and b_j is an additive bias parameter. In a similar way, every CNN operation produces an output which consists of a number of channels. The exact arrangement of operations, and connections between them, depends on the architecture.

A common operator to downsample images is the strided convolution. The stride defines the step size of the convolution kernel. A convolution with stride s defines a map $h : \mathbb{R}^{m \times n} \to \mathbb{R}^{\frac{m}{s} \times \frac{n}{s}}$. Upsampling images can be done by transposed convolutions. Transposed convolutions intersperse the input image pixels with zeroes so that the output image has larger dimensions.

In addition to convolution operators, other common operators such as pooling and batch normalization are often used. A batch normalization operator [101] normalizes the input images for convolutional layers. A batch normalization operator scales and shifts an image $\mathbf{x}_i$ by

$$\mathbf{y}_i = \gamma \frac{\mathbf{x}_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}} + \beta. \tag{4.2}$$

Here, γ and β are scaling and bias parameters which are learned during training, and μ_B and σ_B^2 are the running mean and variance of the mini-batch, i.e., the set of images used for the current training step. For an overview of CNN components we refer to [68].

4.3.2 Pruning convolution filters

Pruning techniques aim to remove extraneous parameters from a neural network. Several schemes exist to prune parameters from a network, but retraining the network after pruning is critical to avoid significantly impacting accuracy [81]. Pruning a network once after training is called one-shot pruning. Alternatively, a network can be fine-tuned, where the network is repeatedly pruned by a certain percentage and is retrained for a few epochs after every pruning step. Fine-tuning typically gives better results than one-shot pruning [198].

Generic pruning algorithm: All pruning methods used in this work make use of the fine-tuning pruning algorithm outlined in Algorithm 1. The selection criteria for determining which filters to keep for each step define the different pruning methods. The pruning ratio `pRatio` is the fraction of remaining convolutions we ultimately want to keep, and `stepRatio` is the fraction of convolutions that is pruned at each step.

Algorithm 1 Fine-tuning pruning algorithm

```
1: procedure Prune(model, pRatio, nSteps, epochs)
2:     stepRatio ← e^(ln(pRatio)/nSteps)
3:     for step ← 0 to nSteps do
4:         pruneParams ← selectPrunePars(model, stepRatio)
5:         model ← removePars(model, pruneParams)
6:         for k ← 0 to epochs do
7:             model ← trainOneEpoch(model, trainData)
       return model
```

Here, we focus on structured pruning. In structured pruning, a common approach to decide which filters to remove is *structured magnitude pruning*. When

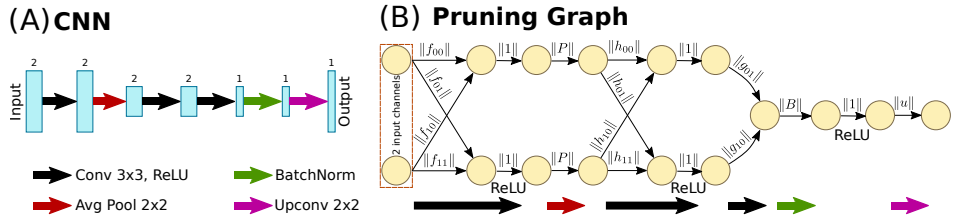


Figure 4.1: (A) Example CNN architecture with the number of channels indicated above each layer. (B) Associated pruning graph. Every channel is a node, and every operator is an edge connecting input and output nodes. The edge weights are the corresponding operator norms.

using structured magnitude pruning, a convolution filter $\mathbf{h} \in \mathbb{R}^{k \times k}$ is scored by its L_1 vector norm $||\mathbf{h}||_1$. Filters with norms below a threshold are pruned. The threshold is determined by sorting a group of filters, and removing a percentage based on the pruning ratio. Thresholds can be set per layer or globally. Setting thresholds globally can give higher accuracy than setting thresholds per layer [18].

4.3.3 Operator norm

As an alternative to the L_1-norm, one can interpret a convolution h as a linear operator which acts on the input image, and score it according to an operator norm. The (induced) operator norm $\|\cdot\|_p$ is defined as

$$\|h\|_p := \sup \left\{ \|h * \mathbf{x}\|_p \;\middle|\; \mathbf{x} \in \mathbb{R}^n, \|\mathbf{x}\|_p = 1 \right\}. \tag{4.3}$$

A common operator norm is the *spectral norm*, which is induced by the Euclidean norm ($p = 2$). The spectral norm can be obtained by calculating the largest singular value of the matrix H associated with h, $\|h\|_2 = \sigma_{max}(H)$ [148]. A property that we will use is that the spectral norm is submultiplicative, i.e., we have

$$\|AB\| \leq \|A\| \cdot \|B\|, \quad \text{for all} \quad A, B \in \mathbb{R}^{n \times n}. \tag{4.4}$$

4.4 Method

The idea behind LEAN is to construct a weighted graph structure formed by the operators of the CNN and having their respective norms as edge weights, such that the multiplicative longest paths in this graph are selected as important subnetworks. The remaining unselected operators will then be pruned. The motivation for LEAN is two-fold. The first consideration is that since convolutions are linear operators, the scaling of an individual convolution within a chain of convolutions is somewhat arbitrary. For a scalar λ and chain of linear operators $A_1 \circ \cdots \circ A_m$, we have that

any chain $(\lambda_1 A_1) \circ \cdots \circ (\lambda_m A_m)$ is equivalent if $\prod \lambda_i = 1$. Since $\|\lambda_i A_i\| = |\lambda_i| \|A_i\|$, this can lead to any arbitrary ranking of operators. However, the chain as a whole gives the same output for each input. We argue that this can lead to incorrect pruning when pruning individual operators based on norms, rather than entire chains. We hypothesize that these scaling properties still approximately hold in the presence of non-linear operators. Since LEAN ranks chains of operators, it is invariant under these scaling properties. Second, by extracting chains LEAN combats network disjointness.

4.4.1 LEAN: creating the pruning graph

Graph structure: In this section, we define the pruning graph that is the basis of the LEAN algorithm. As discussed in Section 4.3.1, we say that every CNN operation has an input $\mathbf{x}$ and an output $\mathbf{y}$, consisting of channels $\mathbf{x}_i$ and $\mathbf{y}_i$. For each channel, we add a single node in the pruning graph. An edge connects two nodes corresponding to input channel $\mathbf{x}_i$ and output channel $\mathbf{y}_i$ if channel $\mathbf{x}_i$ is used in the computation of channel $\mathbf{y}_i$. In the terminology of Section 4.3.1, each edge corresponds to an operator. For instance, a convolution operation is converted by adding an edge from each input channel $\mathbf{x}_i$ to every output channel $\mathbf{y}_j$, each corresponding to exactly one filter h_{ij}. Certain CNN operations may be performed in-place in practice, but we consider them as separate nodes in the pruning graph. A combined convolution and ReLU operation, for instance, results in a node for the output of the convolution and a separate node for the output of the ReLU, as shown in Figure 4.1.

Edge weights: To each edge, we assign as weight the operator norm of its corresponding operator. That is, we calculate the maximal scaling that an input could undergo as a result of applying the operator. In this calculation, we ignore any additive bias terms. For instance, applying a batch normalization results in a scaling of $|\gamma|/\sqrt{\sigma_B^2+\epsilon}$ (see Equation (4.2)). The scaling of non-linear operators in neural networks is sometimes bounded, as in the case of the ReLU for instance, to which we assign a weight of 1. We describe the calculation of operator norms for various convolution operators in Section 4.4.3.

Path lengths: The length of a path in the graph is determined by multiplying the edge weights. LEAN aims to model the norm of the composition of the operators corresponding to the edges. Equation (4.4) states that $\|A\| \cdot \|B\|$ is an upper bound for $\|AB\|$. For LEAN, we assume that $\|AB\| \approx \|A\| \cdot \|B\|$ is a reasonable approximation, although it may not hold generally. By defining the path length as the multiplication of the edge weights (operator norms), path lengths are invariant under scaling linear operators in a chain if the scalars multiply to 1.

There are some edge cases to consider. First, some CNNs contain operations that are meant to distribute features throughout the network, but are not implemented with learnable parameters, e.g., residual connections in ResNet [84]. We include residual connections in the pruning graph with an edge weight of 1, but label them as unprunable to prevent the residual connections from being removed from the network. Second, we do not consider CNNs with recurrent connections. Therefore,

the pruning graph is a Directed Acyclic Graph (DAG). Large pruning graphs can be reduced in size, e.g., by merging nodes that are connected by a single edge (see Appendix A.1.2).

4.4.2 LEAN: extracting chains from the graph

The LEAN method prunes chains of convolutions based on paths in the graph; we **keep** the longest paths (with the highest multiplicative operator norm). We refer to this as LongEst-chAiN (LEAN) pruning. When we perform LEAN pruning, we iteratively extract such paths from the graph until we have reached the pruning ratio. This means that the edges that are not extracted are pruned. Finding paths is done by iteratively running an all-pairs longest path algorithm [42].

Algorithm 2 LEAN

```
1: procedure LEAN(MODEL, PRUNERATIO)
2:     graph ← createPruningGraph(model)
3:     retainedConvs ← []
4:     while fractionRemainingConvs < 1 − pruneRatio do
5:         bestChain ← longestPath(graph)
6:         retainedConvs ← retainedConvs + bestChain
7:         graph ← removeFromGraph(graph, bestChain)
8:     convsToPrune ← convsInModel − retainedConvs
9:     return convsToPrune
```

LEAN pruning is incorporated in the fine-tuning procedure. For each pruning step in the fine-tuning procedure, Algorithm 2 is used to select the filters to prune (line 4 in Algorithm 1). For DAGs, the longest path in a graph can be found in $\mathcal{O}(|V|+|E|)$ time, where V is the set of nodes, and E is the set of edges [222]. For a CNN with m channels (nodes), and k operators (edges), we can extract a longest path in $\mathcal{O}(m+k)$ time. As we extract at least one operator with every execution of line 5 in Algorithm 2, the longest path algorithm is run at most $k(1-p)$ times. So the worst-case complexity of Algorithm 2 is $\mathcal{O}(k(1-p)(m+k))$ for a pruning ratio p. Unprunable edges can be part of a longest path to extract operators, but do not count towards the pruning ratio.

After every LEAN pruning step is concluded, some post-processing is performed. In some cases there are channels which receive no input data at all, or which are equal to a homogeneous constant image for all input data. We therefore remove nodes without incoming edges as well as nodes where the succeeding batch normalization has running variance below some threshold (10^{-40} by default). A low running variance can occur when the output of a convolution is always zero after applying the ReLU activation function, for instance. Second, bias terms are removed from the CNN when all associated convolution or batch normalization operations are pruned.

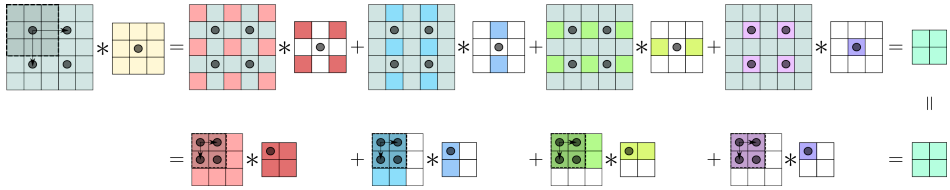


Figure 4.2: The output of a stride-2 convolution can be obtained by adding the result of 4 convolutions. Split the image and filter into the coloured sections, with white entries representing zeroes, and sum the outputs pixel-wise. The dots in the image represent the positions of the center of the filter as it moves over the image.

4.4.3 Operator norm calculation

Operator norm of convolutions: For a convolution filter h and an $n \times n$ image, h^+ is the filter padded with zeros to size $n \times n$. The singular values of h are the magnitudes of the complex entries of the 2D Fourier transform $F_{2D}(h^+)$ (Section 5 of [103])

$$\sigma_{max}(H) = \max\{|F_{2D}(h^+)|\} \tag{4.5}$$

Downsampling: operator norm for strided convolutions: A strided convolution is equal to the sum of regular convolutions on smaller input images (see Figure 4.2). A single parameter of a stride-2 convolution filter is multiplied only with every other pixel (horizontally and vertically). Similarly, filter parameters which are 2 apart are multiplied with the same pixels. Here, we calculate the operator norm for a stride-2 convolution operator h. Let $h^{[i]}$ and $X^{[i]}$ be the partitioned convolution kernels and input images, both zero-padded to the correct size. For a stride-2 convolution we have

$$h *_2 X = \sum_{i=1}^{4} h^{[i]} * X^{[i]}. \tag{4.6}$$

We can apply Equation 4.5 to obtain the singular values of $h^{[i]}$. Equation 4.6 is analogous to the equation of a convolutional layer with 4 input channels, and 1 output channel. The operator norm of a convolutional layer can be computed by means of a tensor $P \in \mathbb{R}^{4\times 1\times n\times n}$ [223]

$$P_{c_{in},c_{out},i,j} = F_{2D}(h^{[c_{in}]+})_{i,j}.$$

According to Theorem 6 of [223], the spectral norm of the convolutional layer equals the maximum of the singular values of the 4×1 matrices $P_{:,:,i,j}$. Since the matrices are single-column, their singular value equals their L_2-norm. Therefore, the spectral norm of h equals

$$\|h\| = \max_{i,j}\left\{\sum_{c_{in}} P^2_{c_{in},:,i,j}\right\}. \tag{4.7}$$

Upsampling: operator norm for transposed convolutions: The matrix of a stride-s transposed convolution is the transposed matrix of a stride-s convolution [139]. Since we have $\|A\| = \left\|A^T\right\|$, for a transposed convolution h, the operator norm can be computed by Equation 4.7.

4.5 Experimental setup

In our experiments, we compare LEAN pruning to several structured pruning methods across three image segmentation datasets and three CNN architectures: MS-D, U-Net4, and ResNet50. We assess the performance of pruned neural networks across 5 independent runs of fine-tuning (Algorithm 1). For each dataset, we have trained a single model as a starting point for pruning. In every experimental run, the same trained model was pruned.

For all pruning methods, we measure the pruning ratio as the fraction of convolutions remaining: $\sum_{h \in H} M(h) / |H|$ where H is the set of all convolutions in a network, and $M(h)$ is 0 if a convolution $h \in H$ is pruned and 1 otherwise. This means that other parameters, such as batch normalization and bias parameters, are pruned when the associated convolutions are pruned, but do not count towards the pruning ratio.

The MS-D networks were pruned to a pruning ratio of 1% in 45 steps. The U-Net4, and ResNet50 networks were pruned to a ratio of 0.1% in 30 steps. All were retrained for 5 epochs after each step. We chose relatively severe pruning ratios because we are interested in significant computational speedup. U-Net4 and ResNet50 are pruned to a lower ratio as they have orders of magnitude more convolutions than the MS-D network.

4.5.1 Structured pruning methods

We compare LEAN to two layer-wise pruning methods, and two global pruning methods. The layer-wise pruning methods are geometric median pruning (**GM**) [86], and soft filter pruning (**SFP**) [85]. The first global pruning method we compare to is **structured magnitude pruning**, i.e., pruning entire filters by their L_1-norm (see Section 4.3.2), and the second global method we compare to is **operator norm pruning**, i.e., pruning entire filters using the operator norm. Here, we choose to use the spectral norm. The difference between LEAN and structured magnitude pruning is 1) the operator norm; 2) the consideration of network structure. The comparison of LEAN to structured operator norm pruning measures the effect of incorporating the network structure.

We compare LEAN to structured magnitude pruning and operator norm pruning for all CNN architectures. Both GM and SFP contain implementations to prune ResNet50 but not MS-D or U-Net4, and hence are used only in the ResNet50 experiments.

4.5.2 CNN architectures

In this section, we describe three fully-convolutional CNN architectures that are used in the experiments: the Mixed-Scale Dense convolutional neural network (MS-D) network [183], U-Net [202], and ResNet [84]. Table 4.1 outlines which operators are present in the networks. The networks were trained using ADAM [113] with $lr = 0.001$, minimizing the negative log likelihood function.

	Convolution			
CNN	Strided	Transposed	Dilated	Pooling
MS-D	No	No	Yes	No
ResNet50	Yes	No	Yes	Yes
U-Net4	No	Yes	No	Yes
	Batch normalization	# Parameters	Edges in pruning graph	
MS-D	No	$4.57 \cdot 10^4$	$5.05 \cdot 10^3$	
ResNet50	Yes	$3.29 \cdot 10^7$	$1.44 \cdot 10^7$	
U-Net4	Yes	$1.48 \cdot 10^7$	$1.84 \cdot 10^6$	

Table 4.1: Operators present in MS-D, ResNet50, U-Net4 architectures.

In our experiments we use MS-D networks as described in [183] and implemented in [89]. Every layer has 1 channel and all convolutions have a dilated 3×3 filter, except the final 1×1-layer. The dilations for layer i were set to $1 + (i \mod 10)$. The final layer is excluded from pruning as it contributes less than 0.5% of FLOPs.

As U-Net architecture we use a fully-convolutional (FCN) U-Net4 network, i.e., a U-Net with 4 scaling operations. We used a U-Net4 architecture from the PyTorch-UNet repository [149]. As ResNet architecture, we use an FCN-ResNet50 network [84]. The ResNet50 model is adapted from PyTorch's model zoo code. We replace the max pooling layers of ResNet and U-Net with average pooling layers as average pooling is a linear operator which can be modeled as a strided convolution for which we can compute the operator norm. In some cases U-Net with average pooling can perform better than with max pooling [8].

4.5.3 Datasets

In our experiments, we consider three datasets: a high-noise, but relatively simple, segmentation dataset [183] (CS dataset); the well-known CamVid dataset [24, 25]; and a real-world X-ray CT dataset [38, 39] to test the methods in practice. The CS dataset is a 5-class segmentation dataset of 1000 training, 250 validation, and 100 test images. As a starting point for pruning, we trained a 100-layer MS-D network with an accuracy of 97.5%, ResNet50 with 95.8% accuracy, and U-Net4 with 97.6% accuracy.

The X-ray CT dataset consists of 9216 training, 2048 validation, and 1536 test images. As in [209], we use the F1-score to quantify results. As a starting point for pruning, we trained a 100-layer MS-D network with a 0.88 F1-score, ResNet50

with a 0.85 F1-score, and U-Net4 with a 0.88 F1-score. As in [180], experimental results on the CamVid dataset are quantified using mean Intersection-over-Union (mIoU). As a starting point for pruning, we trained a 150-layer MS-D with 0.52 mIoU, ResNet50 with 0.71 mIoU, and U-Net4 with 0.65 mIoU. More details on the datasets can be found in Appendix A.1.1.

4.6 Results

4.6.1 Experimental results for severe pruning

The results of the pruning experiments are displayed in Figure 4.3, showing that LEAN pruning at similar accuracy obtains networks with a lower pruning ratio than the four compared methods. The pruning ratio that can be achieved without significant loss of accuracy depends on the network architecture and the complexity of the dataset.

In the CS dataset results, we notice a drop-off point where performance decreased significantly for all networks. On average over 5 runs of pruning, LEAN achieved a 3.4%, 0.79%, and 0.79% pruning ratio for MS-D, U-Net4, and ResNet50, with an average accuracy reduction of 1.4%, 2.5%, and 2.1% respectively. On ResNet50, at an accuracy reduction of 3% compared to the unpruned network, LEAN achieves a 43% reduction in the number of convolutions compared to GM and SFP. Below 15% accuracy reduction SFP and GM perform better, but the network no longer reliably segments the data at these accuracies.

On the dynamic X-ray CT dataset, we notice large fluctuations in F1 for MS-D and U-Net4. This may be due to the F1-score which is defined as a reciprocal. In addition, on U-Net4, LEAN performs better than the structured pruning methods right from the start. On average over 5 runs, LEAN achieved a 5.1% and 6.3% pruning ratio for MS-D and U-Net4, with an average F1 drop of 5.7%, and 3.3% respectively. For ResNet50, a drop-off point is again observed, which occurs at a significantly lower pruning ratio for LEAN than for both global pruning methods. GM and SFP exhibit a more gradual reduction in F1-score.At an F1-score reduction of 3% compared to the unpruned network, LEAN achieves a 92% reduction in the number of convolutions compared to GM and SFP.

On the CamVid dataset, we observe a declining mIoU for MS-D and U-Net4 as pruning progresses on this challenging dataset. On ResNet50 we notice an initial drop in mIoU, but subsequent pruning steps increase the performance initially. These observations could indicate that 5 epochs of retraining are not sufficient for the CamVid dataset to recover performance. Interestingly for U-Net4, we notice that for two pruning ratios, structured magnitude pruning appears to perform slightly better than LEAN. Given the variance between different runs, possibly due to the limited number of retraining epochs, we suspect that this difference is not statistically significant. After the initial drop in mIoU, we notice a later drop-off pruning ratio on ResNet50. LEAN achieves a 6.3% pruning ratio with an average mIoU reduction of 14.3% on ResNet50, whereas the best performing

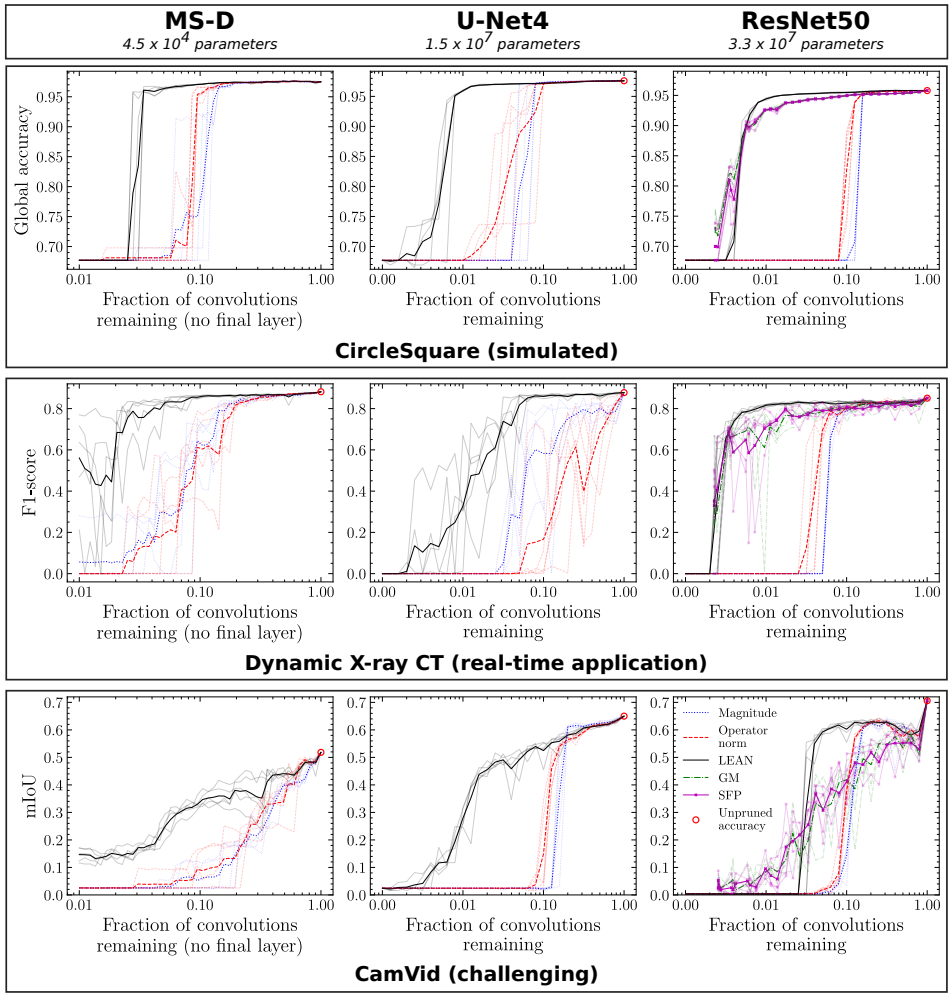


Figure 4.3: Comparison of structured pruning methods and LEAN pruning on three datasets (rows). Pruning methods are applied to MS-D, U-Net4, ResNet50 network architectures. The base model is pruned to a ratio of 1% (MS-D) or 0.1% (ResNet50, U-Net4) for all datasets. This is repeated five times (translucent lines) and the average is taken (solid lines).

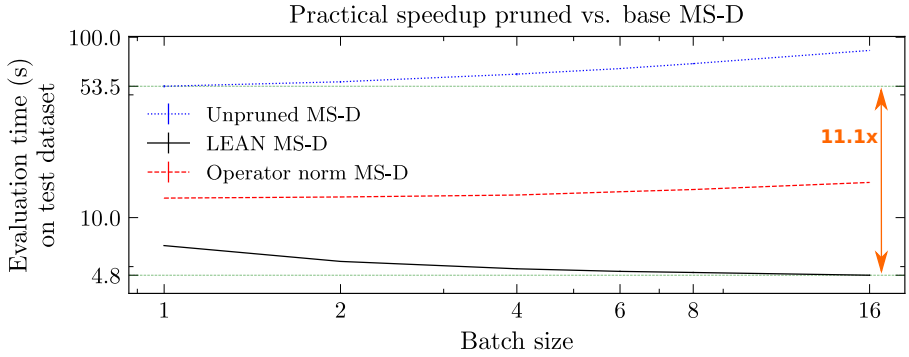


Figure 4.4: Practically realized speedup of pruned MS-D networks evaluated on the test dataset.

other method dropped-off at a 20.0% pruning ratio. Interestingly, both layer-wise pruning methods GM and SFP exhibit a sustained reduction in test mIoU rather than the drop-off we observe for the global pruning methods.

4.6.2 Speedup real-world dynamic X-ray CT segmentation

In addition to measuring the achievable pruning ratios, we measure the practically realized wall-time speedup. We tested this on the dynamic X-ray CT dataset for which a speedup has immediate benefits in practice. Existing pruning support in PyTorch only masks pruned filters, thereby not creating a faster network. Therefore, we implemented a custom MS-D model which loads only the unpruned filters. During the experiments, an MS-D network pruned to a ratio of 2.5% (40-fold reduction) with LEAN achieved an F1-score of 0.83 (drop of 5.4%). This network was 11.1 times faster than the unpruned network in practice. The speed of evaluating the entire test set is impacted by the batch size, which we take into account as shown in Figure 4.4. For comparison, we included the best performing pruned MS-D network by an other pruning method (operator norm pruning) which achieved a pruning ratio of 15.8% with an F1-score of 0.83. We show differences between MS-D networks pruned with different pruning methods in Appendix A.1.3.

4.7 Conclusion

In this paper, we have introduced a novel pruning method (LEAN) for CNN pruning by extracting the highest value paths of operators. We incorporate existing graph algorithms and computationally efficient methods for determining the operator norm. We show that LEAN pruning permits removing significantly more operators while retaining better network accuracy than several existing pruning methods. Our results show that LEAN pruning can increase the speed of the network, both in theoretical speedup (FLOPs reduction) and in practice. In conclusion, LEAN

enables severe pruning of CNNs while maintaining a high accuracy, by effectively exploiting the interdependency of network operations.

Future work could be split along several lines. First, there are more CNN operators for which methods to compute their operator norms could be developed. Notably, we have mostly disregarded non-linear operators in this work. Next, LEAN approximates the norm of a chain of operators using the submultiplicative property upper bound $\|AB\| \leq \|A\| \cdot \|B\|$. New methods for approximating the norm of a chain of composed operators could strengthen LEAN as it more accurately extracts chains with strong operator norms. In addition, new graph theoretic approaches for extracting meaningful paths from the graph could be explored. Such algorithms are already abundant in the field of graph theory, and could quite readily be carried over to neural network pruning research. Lastly, LEAN currently works by greedily extracting high-value paths. Approaches such as [75] could be considered to avoid greedy selection of operators.

5

Benchmarking optimization kernels for auto-tuning GPU kernels

5.1 Introduction

Graphics Processing Units (GPUs) have revolutionized the HPC landscape in the past decade [88], and are seen as one of enabling factors in recent breakthroughs in Artificial Intelligence (AI) [121]. GPUs originated as processors for gaming and then adapted to more general workloads as co-processors in many HPC systems. Over the past decade, GPUs have started to again penetrate new markets such as IoT devices [152] and autonomous vehicles [135]. The range of applications of GPUs as such continues to expand. Because of their relatively low cost with respect to their parallel processing power, more and more supercomputers come equipped with GPUs, and in 2020, the majority of modern supercomputers use GPUs [236] as the major source of compute power.

The sections of code that run on a GPU, called *kernels*, can be challenging to configure such that they run efficiently for a varying combinations of datasets and GPU architectures [248]. The kernel parameters can be split into those defined by the program, and those that are a consequence of the underlying architecture and models behind the GPU. The hardware-specific parameters define how the thousands of threads in a GPU are grouped. An ineffective layout can cause underutilization of GPU resources. In general, the computational efficiency can drop by an order of magnitude depending on certain implementation choices. Typically, only a small subset of the possible configurations lead to a large increase

in performance [220]. Therefore, it is vital to be able to select an efficient kernel configuration.

The search space for this problem is formed by all feasible combinations of GPU kernel parameters. This space is discrete and non-convex [185], making it hard to carry out the optimization. For most GPU kernels used in practice, the size of this search space is such that traversing the options by hand or brute-force is infeasible. An additional complication in optimizing kernel parameters is that evaluating the performance of each configuration requires costly recompilation and test runs. Furthermore, the same GPU kernel often requires re-tuning for different input data, hardware, or after changes to the code [107, 127, 166, 167]. Large throughput pipelines often rely on computationally expensive GPU kernels that consume large amount of resources [218, 221], and cannot be tuned exhaustively due to the aforementioned reasons.

Automatic performance tuning (auto-tuning) techniques rely on empirical results and feedback to optimize the kernel parameters with respect to desired performance metrics. These techniques aim to be widely applicable across architectures. For this reason, auto-tuning can be used to find configurations with increased performance for GPU programs. As the search space for the auto-tuning task depends on various aspects (kernel source, code layout, input data, GPU-architecture), the optimization framework must deal with a broad variety of search spaces and constraints. We, therefore, treat the problem as a black-box optimization task. This raises the question of which optimization algorithm is best suited to find highly efficient settings for GPU kernels, and how these optimization algorithms need to be configured to tune GPU kernels.

The main contribution of this work is to determine which optimization algorithms produce the fastest GPU kernels for different tuning-time ranges. To do so, we conduct a survey of 16 evolutionary optimization algorithms for 9 different NVidia and AMD GPUs, and run 3 real-world applicable benchmark kernels. We select our benchmark problems such that we are able (given ample time) to compute the entire search space, and make these spaces publicly available. To benchmark the GPU kernels we use the Kernel Tuner package [246]. We use the wide range of optimization algorithms present in Kernel Tuner for a large-scale comparison, and provide favourable default hyperparameters for GPU tuning for each algorithm. In addition, we extend Kernel Tuner with several highly-efficient optimization algorithms, including iterative local search (ILS) and dual annealing that cannot be found in any other generic auto-tuning framework.

Secondly, we aim to quantify tuning difficulty for these seemingly challenging and capricious search spaces. To do so, we introduce the *fitness flow graph* (FFG), which is a network of the points in the search space, with directed edges between neighbours with a better fitness. By computing the likelihood of local search walks terminating in good local minima, we use FFGs to better understand the discrepancies between optimization algorithms, and subsequently tailor them to better suit GPU tuning. In addition, FFGs can help explain the differences across GPU manufactures, architectures, and kernel programs, and such knowledge can help steer future development. To quantify tuning difficulty per kernel, we introduce

a novel metric based on Google's PageRank algorithm [21, 173].

This work is structured as follows. In section 5.2 we discuss existing GPU kernel tuning approaches. In section 5.3 we introduce the preliminaries on GPU kernels, and describe the optimization algorithms that are considered in this survey. In section 5.4, we describe certain implementation details of Kernel Tuner and our Python optimization package BlooPy. We discuss the setup of our experiments in section 5.5. In section 5.6 we tune the hyperparameters of the algorithms, and present our findings on optimization algorithm performance. In section 5.7 we introduce fitness flow graphs (FFGs) and quantify tuning difficulty for kernel search spaces. Finally, we present our conclusions in section 5.8.

5.2 Related work

5.2.1 Automated performance tuning

It is well-known that GPU tuning can yield considerable gains in computational efficiency and utilization for large-scale, high-throughput pipelines that run on compute clusters. As an example, we mention the AMBER pipeline [218, 221], which is used to detect Fast Radio Bursts (FRBs) and other single pulse radio transients in astronomy. The pipeline has a throughput of 2 TB/s, and uses a large amount of resources. Benchmarking a single configuration is expensive, and the search space consists of millions of configurations, meaning that sophisticated tuning approaches have to be developed.

Research in automated performance tuning (auto-tuning) can be grouped into two main categories: (1) auto-tuning compiler-generated code optimizations [111, 179, 193, 234], and (2) software auto-tuning [127, 259]. Ashouri et al. [7] wrote an excellent survey on machine-learning methods for compiler-based auto-tuning. In this chapter, we limit our scope to (2), i.e., optimizations methods for software auto-tuning, which is sometimes referred to as automated design space exploration [159]. Software auto-tuning allows developers to automatically optimize individual functions and allows, for example, to tune for entirely different implementations and parallelizations that solve the same problem.

As such, auto-tuning techniques are often employed to optimize the source code of high-performance libraries and applications for the CPU, e.g. ATLAS [249] or FFTW [63], as well as for GPUs [71, 127, 145, 220, 235, 247, 259].

A number of generic auto-tuning frameworks have been introduced in recent years. OpenTuner [5] was one of the first generic software auto-tuning frameworks, supporting a number of different search optimization algorithms, but with no support for tuning individual GPU kernels. GPTune [137] and HyperMapper [159] are recently proposed frameworks that both use Bayesian Optimization for auto-tuning on different platforms, but do not target GPUs.

Grauer-Gray et al. [70] have applied auto-tuning to the high-level directive-based HMPP framework, which can compile to CUDA or OpenCL code. They demonstrate significant performance improvements using auto-tuning over unopti-

mized HMPP kernels in the PolyBench benchmark suite. Wang et al. [243] take a similar compiler-based approach to automatically convert shared memory OpenMP applications into OpenCL code for GPUs. They use a machine-learning approach, which based on the number of compute operations and memory accesses in the kernels, predicts the best performing hardware platform to execute the kernels either the multi-core CPU using OpenMP, or on the GPU using OpenCL. Hou et al. [97] proposed a data-sensitive auto-tuning framework for sparse matrix vector (SpMV) multiplication that automatically finds the best parallelization strategy. They use a two-step machine learning approach in which they first determine the optimal way to group data into bins and then select the most suitable kernel to process the rows in each bin.

CLTune [166] was the first generic auto-tuning framework with specific support for directly tuning GPU kernels written in the OpenCL programming language. CLTune supports several optimization algorithms, including simulated annealing and particle swarm optimization, but these do not outperform random search [166]. Kernel Tuning Toolkit (KTT) [60] is developed specifically to support online auto-tuning and pipeline tuning, which allows for the exploration of combinations of tunable parameters over multiple kernels. An interesting feature of KTT is the support to keep track of hardware performance counters, such as L2 cache utilization, during benchmarking, which can also be used in advanced search strategies [61]. Auto-Tuning Framework (ATF) [195] implements an innovative way to generate auto-tuning search spaces, for efficient storage and fast exploration of constrained search spaces, but does not focus on introducing new optimization algorithms.

In earlier work, we have introduced Kernel Tuner [246], a generic auto-tuning framework specifically designed to be an easy-to-use and easy to extend tool for researching auto-tuning optimization algorithms. Kernel Tuner is a state-of-the-art framework that implements the largest range of search optimization strategies of all generic auto-tuning frameworks, and was the first generic framework to implement multiple search strategies that consistently outperformed random search [246].

5.2.2 Analyzing auto-tuning search spaces

In this chapter, we do not only compare the performance of different optimization algorithms on the GPU auto-tuning problem, but we also investigate the properties of the search spaces to understand why certain optimization algorithms outperform others, and gain insight into the difficulty of the optimization problem. A comprehensive introduction to GPU tuning difficulty is given in [219].

Ryoo et al. [205] were one of the first to study the properties of optimization spaces for GPU applications. They defined two performance metrics to model the efficiency and utilization of a CUDA kernel and used these to find kernel configurations on the pareto curve that maximizes the two metrics. A downside of this approach is that the performance metrics have to be constructed for each kernel individually and require manual inspection and counting instructions in assembly code. Lim et al. [128] also look into search space properties to preselect certain

parameter values using static code analysis in order to try to limit exploration of the search space by an auto-tuner.

In [169], Ochoa et al introduce the concept of *Local Optima Networks* (LONs). When constructing LONs the search space is partitioned into basins of attraction, i.e., sets of points where a local search algorithm will terminate in the same local minimum. A LON is a graph with the local optima as vertices, and a directed edge between nodes if a local search step transforms a solution from one basin of attraction to another. We build upon this idea to define *fitness flow graphs* (FFGs), which as opposed to LONs contain all the points in the search space. Due to the large number of points with "failure fitnesses" (when a configuration fails to compile) in GPU kernel spaces, defining the basin of attraction is difficult. Instead, we simplify the ideas behind LONs to the entire search space, and quantify how likely local search algorithms terminate in good local optima. To do so, we look at PageRank centrality of local optima. In [93] the idea of using PageRank centrality for LONs as predictor of performance for local-seach based heuristics was proposed, and in [92] the PageRank was used to rank space difficulty. We extend this idea to FFGs to determine GPU tuning difficulty.

5.3 Method: Optimization problem

In this section, we define the performance optimization of GPU kernels as a mathematical optimization problem, and we present the optimization algorithms that are part of our experiments.

5.3.1 GPU kernels

GPU kernels are executed by millions of threads in parallel to perform data-parallel computations on the GPU. However, the compute performance of a GPU kernel depends on how the software has been optimized for the hardware.

There are various different design choices that have an impact on the performance of GPU kernels, and this impact is challenging to accurately predict. For example, the way that a computation is parallelized and mapped on the thread blocks and individual threads affects the utilization of the GPU cores. Other design choices include what data types and data layouts to use in the various memory spaces available to GPU applications. There may also be entirely different algorithms to choose from to implement certain parts of the computation.

Other tunable parameters are introduced through code optimizations that can be enabled and may in turn introduce new parameters, such as tiling factors, vector data types, or partial loop unrolling factors. GPU kernels also have a number inherent parameters in terms of the number of thread blocks and the number of threads per block that are used to execute the kernel. The multitude of implementation choices for GPU kernels result in sizeable, non-convex, and discontinuous kernel design spaces.

To automate the kernel design space exploration process, GPU code can be parameterized, either using a kernel template or a code generator. An auto-tuner can take such a kernel template or code generator and empirically benchmark different kernel configurations, until it has found an efficient implementation. The search performed by the auto-tuner can be treated as a mathematical optimization problem of the form:

$$x^* = \arg\min_{x \in X} f(x) \tag{5.1}$$

where $f(x)$ is the performance metric to be minimized, for kernel configuration x for a combination of kernel, GPU device, and input settings. In this work the performance metric to be minimized will be the runtime of the kernel.

5.3.2 GPU kernel search spaces

The search space of possible settings for a GPU kernel can be characterized as a finite subset $X \subset \mathbb{Z}^n$ for n different parameters. Specifically, for each dimension $1 \leq i \leq n$, every entry x_i of a point $x \in X$ takes values from a finite set $S_i \subset \mathbb{Z}$. For example, the block dimension might allow values in $\{16, 32, 64, 128\}$. The total search space is the Cartesian product of these finite sets

$$X = S_1 \times S_2 \times \cdots \times S_n.$$

The local structure of a search space depends on the definition of *neighbouring points*. A common definition of the neighbours of a point are the points which differ only in one dimension, and are equal for all other dimensions. Mathematically, according to this definition the set of neighbours $N(x)$ of a point x is

$$N(x) := \bigcup_{i=1}^{n} \{y \in X \setminus \{x\} \mid y_j = x_j,\ \forall j \neq i\}. \tag{5.2}$$

Here, we will consider a more restrictive type of neighbourhood concept where we place the additional requirement that the parameter that differs from x_i should have a value adjacent to x_i in the list S_i. For example, if the block dimension is allowed to be $[16, 32, 64, 128]$, then neighbours of $x_i = 64$ would be 32 and 128, and the neighbour of 128 would only be 64. We consider this restriction because this definition gives information on whether closely related parameter values are related in performance.

Points of special interest in the search space are local minima. A point is a local minimum if all neighbouring points have a worse fitness. In other words, there are no improvements to be found in the local neighbourhood. Algorithms that scan local neighbourhoods can get stuck in local minima as there are no close points with better fitness.

5.3.3 Black-box optimization algorithms

As the search space is typically too large to iterate over all feasible points, a range of more sophisticated optimization algorithms are used in practice. In this section

we describe the optimization algorithms that are considered in the experiments. As a categorization of these algorithms, we distinguish continuous, and discrete algorithms, and algorithms that learn about the stochasticity of the problem.

Optimization - discrete algorithms

Since tuning GPU kernels involves choosing the best from a finite set of possibilities, it makes sense to consider *discrete optimization algorithms*. These algorithms are considered in this work:

- **Random sampling** randomly generates solutions and records the highest scoring one. This strategy serves as a baseline comparison to determine if optimization algorithms offer significant benefits.

We consider several local search (or hill climb) algorithms which iteratively check for a neighbouring solution of lower fitness to visit, until a local minimum is reached. For all local search algorithms we distinguish between *best-improvement* search, where we move to the best neighbour next, and *first-improvement*, where we examine neighbours in a random order and move to one when we encounter an improvement.

Local search algorithms can vary the neighbourhood function that they use to generate new candidates. The algorithms can use both the version outlined in equation 5.2 (called *Hamming*), and the more restrictive neighbourhood definition in section 5.3.2 (called *adjacent*). First-improvement variants can decide whether they continue checking the remaining variables first after finding an improvement, or if they restart the search (hyperparameter *restart search*).

- **Multi-start local search (MLS)** repeatedly generates random starting solutions, and hill climbs them until a local minimum is reached.

  Hyperparameters: neighbourhood, restart.

- **Iterative local search (ILS)** [141] is similar to MLS but inherits part of the original local minimum when generating a new starting solution. After reaching a minimum, ILS performs several random permutations to generate a new starting solution. This *perturbation size* is a tunable parameter. In addition, a tunable *exit after no improve* hyperparameter randomly restarts if no improvement is found after a that many iterations, which helps to escape basins of attraction for small perturbation sizes.

  Hyperparameters: perturbation size, exit after no improve, neighbourhood, restart.

- **Tabu search** [65] maintains a queue of previously visited solutions which the algorithm is not allowed to visit. The tunable hyperparameter *tabu size* defines the queue size, and ensures that the new solution has not been visited for *tabu size* iterations. Tabu search always picks a new solution, whether it is an improvement or not.

  Hyperparameters: tabu size, neighbourhood.

- **Simulated annealing (SA)** [114] maintains a temperature parameter that, together with the fitness values, determines the probability that we move to a (potentially worse) neighbouring solution. The temperature parameter is decreased each iteration to mimic the behaviour of cooling processes in material physics. A tunable *exploration* parameter determines the size of the mutation of the current solution that is performed at each iteration. A *hill climber* subsequently optimizes the new solution, which is accepted with a certain probability.

  Hyperparameters: exploration, hill climber, neighbourhood

We also consider two discrete population-based algorithms, which require a *population size* parameter to determine the number of solutions they maintain. Population-based methods iteratively create a next generation of solutions by mixing solutions from the previous generation. A *reproduction* operator creates new initial solutions from existing ones, e.g., with two-point crossover a section of the solution vector is swapped between two solutions. After new solutions have been created, and their fitnesses determined, a *selection* mechanism determines which solutions are kept for this generation. For example, in tournament selection a number of randomly picked solutions compete for a spot in the next generation.

- **Genetic local search (GLS)** [102] (or memetic algorithm) is a population-based method where every solution in a generation is subsequently *hill climbed*. The initial population is made up of randomly generated solutions, which after hill climbing are all local minima. Next, a number of children is created by reproduction, and a new initial starting population is selected from the batch. These solutions are subsequently hill climbed and the procedure is repeated.

  Hyperparameters: hill climber, population size, reproduction, selection.

- **Genetic algorithm (GA)** [151] is similar to GLS, but instead of hill climbing each solution, it performs a single mutation only, e.g., permuting a single parameter. Instead of a hill climbing algorithm, GA has a tunable hyperparamter *mutation* that determines the fraction of variables of a solution that are mutated each generation.

  Hyperparameters: mutation, population size, reproduction, selection.

Optimization - continuous algorithms

As an alternative to discrete algorithms, we can consider *continuous optimization algorithms* which operate on real-valued solutions. In order to apply algorithms which assume continuous variables to a discrete problem such as GPU kernel tuning, we need to define a mapping between a real-valued vector, and the discrete values in the search space. Suppose the search space allows values $x_1, x_2, \ldots, x_n$ for a

	GPU Specifications					
GPU model	Year	CUDA cores /Stream processors	Device memory	Boost clock	Band-width (GB/s)	Peak compute (GFLOP/s)
NVidia Tesla K20	2012	2496	5 GB	0.76 GHz	208	3524
NVidia GTX Titan X	2015	3072	12 GB	1.08 GHz	336	6605
NVidia Tesla P100 PCIe	2016	3584	12 GB	1.30 GHz	549	9340
NVidia GTX 1080 Ti	2017	3584	11 GB	1.58 GHz	484	11340
NVidia Tesla V100 PCIe	2017	5120	32 GB	1.37 GHz	900	14899
AMD Radeon Instinct MI50	2018	3840	16 GB	1.73 GHz	1024	13300
NVidia Titan RTX	2018	4608	24 GB	1.77 GHz	672	16312
NVidia RTX	2019	2560	8 GB	1.77 GHz	448	9060
NVidia A100 Tesla PCIe	2020	6912	40GB	1.41 GHz	1555	19500

Table 5.1: Specifications of graphical processing unints (GPUs) used to create experimental data.

particular variable, then a continuous variable $y \in [0, 1]$ gets mapped to the closest grid point $\bar{y}$:

$$B = \{\frac{1}{2n}, \frac{3}{2n}, \ldots, \frac{2n-1}{2n}\}$$
$$j^* = \underset{i=1,\ldots,n}{\arg\min}\{|B_i - y|\}$$
$$\bar{y} = x_{j^*}.$$

Effectively, this ensures that all possible discrete values are equally spaced across the interval $[0, 1]$, and the continuous variable is mapped to the closest one. The continuous optimization algorithms operate on real-valued vectors with dimensions equal to the number of parameters that are to be optimized. Each entry is bounded to the unit interval.

The mapping ensures that points close together in real-valued space can get mapped to the same point in the GPU tuning space. While this can negatively impact the performance of continuous algorithms, it does not automatically lead to poor performance, as illustrated by the strong performance of continuous algorithms in Kernel Tuner [246]. Furthermore, this mapping allows us to explore a new class of algorithms. Here, we consider two local search algorithms.

- **Basin hopping** [241] is a global stepping algorithm that chooses new starting positions for local minimization. It requires the local minimizer *method* and a *temperature* parameter to be chosen. The temperature parameter determines the accept–reject criterion. Currently supported minimization methods are the nonlinear conjugate gradient (CG) [164], simplex (Nelder-Mead) [161],

conjugate direction (Powell) [190], L-BFGS-B [28], Constrained Optimization BY Linear Approximation (COBYLA) [189], and Sequential Least Squares Programming (SLSQP) [115] methods.

Hyperparameters: minimizer method, temperature.

- **Dual annealing** [237] is an extension of generalized simulated annealing, paired with a local minimization method. It combines global and local search procedures, and it requires users to choose a local minimization *method*.

  Hyperparameters: minimizer method.

Lastly, we consider two population-based algorithms.

- **Particle swarm optimization (PSO)** [110] initializes a *number of particles* at random in the search space. Each iteration, these particles update their position and velocity. Particles transmit information to a certain *number of neighbours*, thereby influencing the movement of the other particles.

  Hyperparameters: #particles, neighbours evaluated.

- **Differential evolution** [228] is similar to a genetic algorithm, but mixing strategies are based on real-valued solutions. Typically they involve mixing the best solution with a random candidate, and accepting the result with a certain probability.

  Hyperparameters: mixing method, population size, mutation size, recombination probability.

Optimization - tuning algorithms for stochastic optimization

In this survey we consider two state-of-the-art parameter tuning algorithms;

- **Sequential Model Algorithm Configuration (SMAC)** [132] is a random forest-based Bayesian optimization method that is designed for optimization of stochastic problems. However, it can also be used to optimize deterministic problems. SMAC requires the *model type* of its Bayesian optimizer to be chosen. The *gp-mcmc* model was significantly slower and worse than *gp* in the preliminary experiments. We therefore use the *gp* model type in this work. The *acquisition function* of the BO is another tunable hyperparameter.

  Hyperparameters: acquisition function.

- **Iterated racing (irace)** [140] is a statistical approach for selecting the best configuration out of a set of candidates for stochastic optimization problems. After consulting the authors [140], we set *firstTest* and *nbConfigurations* as tunable hyperparameters for irace.

  Hyperparameters: firstTest, nbConfigurations.

5.4 Implementation

In this section we comment on certain implementation details for the software developed for this work. The algorithms and analysis tools are implemented in the BlooPy Python package, and the tuning of the GPU kernels is performed by Kernel Tuner.

5.4.1 BlooPy and SOTA packages

The algorithms evaluated in this work are implemented in the discrete optimization package **BlooPy** (BLackbOx Optimization Python) [214]. The package implements the algorithms by encoding solutions as bitstrings. BlooPy implements several functions for converting discrete solutions that are encoded as lists or arrays to bitstrings. Similarly, continuous solutions are mapped to discrete solution vectors using the mapping outlined in section 5.3.3. BlooPy requires the search space to be finite. Using the bitstring encodings, BlooPy's algorithms can make use of the computationally efficient Python module `bitarray` which implements fast low-level bitstrings in C. In addition, the algorithms are automatically applicable to benchmark bitstring-based optimization problems such as randomized Nk-landscapes [252].

Optimization algorithms in BlooPy maintain a cache of previously visited solutions. This means that a solution that has been visited before does not count as a function evaluation, and instead the cached value is returned. In addition to a variety of optimization algorithms, BlooPy implements several search space analysis tools. For example, it implements functions to determine the type of points in the search space, e.g., local minima and saddle points. Furthermore, BlooPy implements functions to compute the fitness flow graphs outlined in section 5.7.2. BlooPy can be installed from the GitHub source repository [214], or by package manager. To perform experiments with SMAC we used the Python package [133], and for irace we used the R package [143].

5.4.2 Kernel Tuner

Kernel Tuner [246] implements a wide range of optimization algorithms, and builds on top of various backends (e.g. PyOpenCL, PyCUDA, Cupy, GCC) that take care of the compilation process.

Kernel Tuner runs Python code, provided by the user, which calls the tuner function. In addition, the user needs to provide a code generator or parameterized template for the kernel they wish to optimize. An optimization algorithm then selects different kernel configurations for benchmarking.

Kernel	#points	#local minima	#variables to tune	#failed points
Convolution*	18432	89	6	12656*
GEMM	82944	64	5	64988
Point-in-polygon	8184	220	10	335

Table 5.2: Statistics (averaged across GPU models) of kernel spaces. Number of failed points refers to the average number of configurations in the kernel space that failed to compile.
*Convolution has 864 points for AMD MI50, and 450 fail points.

5.5 Experimental Setup

To analyze the structure of different kernel spaces, and find the optimization algorithm best suited to finding strong kernel settings, we run experiments on 9 different GPUs, for 3 real-world applicable kernel programs (26 kernels in total).

- **Convolution** [247] operations are an essential tool in image processing, and are often used for tasks such as edge detection, blurring, or sharpening. They also feature prominently in deep learning methods for image processing as they form the backbone of the convolutional neural network (CNN).

- **GEMM** (Generalized dense matrix–matrix multiplication) [165] is one of the most widely-used kernels across many application domains, including neural networks. Here we perform the calculation $C = \alpha A \cdot B + \beta C$ for 4096×4096 matrices A, B, C, and constants α and β.

- **PnPoly** (Point-in-Polygon) kernel is used by Goncalves et al. [67] as part of a geospatial database management system to, for example, return all objects within the outline of a specific area.

Some statistics on the kernel spaces is given in Table 5.2. For convolution and GEMM the majority of the points in the kernel space fail to compile, 68% and 78% respectively. In the case of a failed compilation we attribute a "fail" fitness of 10^{10} to this point. The exact kernel spaces can be found in the table of tunable parameters (Table A.1) in Appendix A.2.1.

We selected these kernels programs since they are tunable common subroutines in real-world applications, but also have a compact parameter space which can be fully explored, given ample computation time. Note that this is not feasible for many other kernels used in practice (see section 5.2.1). We have generated cache files of the entire search space for each kernel by brute-force calculation. This allows us to know the optimal settings for each problem, and therefore score solutions returned by algorithms. It also allows us to develop analysis metrics on the entire search space, which could at a later stage be adapted to work when sampling only small parts of the space. We supply our cache files for benchmarking optimization

	Maximum number of function evaluations (budget)						
Basin hopping	25	50	100	200	400	800	1600
method	Powell	COBYLA				SLSQP	
temperature	0.1					1.0	
Dual annealing	25	50	100	200	400	800	1600
method	COBYLA	Powell					
Differential evolution	25	50	100	200	400	800	1600
population size	1	2		4	8	16	32
method	best1bin				best2bin	best1exp	
recombination	0.5	0.7					
mutation	(0.2, 0.7)						
Particle swarm optimization	25	50	100	200	400	800	1600
Number of particles	25		10	20	40	80	160
neighbours evaluated	5			10	20	26	32
FirstILS	25	50	100	200	400	800	1600
perturbation size	1.0					0.05	
Exit after no improve	25					10	
neighbour method	Hamming					adjacent	
restart search	False				True		
BestILS	25	50	100	200	400	800	1600
perturbation size	1.0					0.05	
Exit after no improve	25						
neighbour method	adjacent		Hamming			adjacent	
FirstTabu	25	50	100	200	400	800	1600
tabu size	4		2000				
neighbour method	Hamming						
BestTabu	25	50	100	200	400	800	1600
tabu size	2000						
neighbour method	Hamming						
FirstMLS	25	50	100	200	400	800	1600
restart search	True		False		True		
neighbour method	Hamming						
BestMLS	25	50	100	200	400	800	1600
neighbour method	adjacent			Hamming			
Simulated annealing	25	50	100	200	400	800	1600
explore (p)	1.0				0.7	0.1	
hill climber	None	RandomFirst					
neighbour method	Hamming						

Table 5.3: (Part 1:) Selected hyperparameters across different budgets (25 to 1600) for the optimization algorithms used in this work. The budgets columns are given in red. These hyperparameters were optimized using the convolution, GEMM, and PnPoly kernels on the NVidia P100 GPU.

	Maximum number of function evaluations (budget)						
Genetic local search	25	50	100	200	400	800	1600
hill climber	RandomFirst						
population size	2	2	4	16	20	40	80
reproductor	uniform		2point		uniform		
selector	RTS						
neighbour method	Hamming						
Genetic algorithm	25	50	100	200	400	800	1600
mutation	0.02			0.05		0.02	
population size	8	10	20	40	80	128	320
reproductor	1point					2point	
selector	tour8					tour4	
SMAC	25	50	100	200	400	800	1600
model type	gp					NA	
acquisition function	LCB					NA	
irace	25	50	100	200	400	800	1600
firstTest	NA			2			
nbConfigurations	NA			0			

Table 5.4: (Part 2:) Selected hyperparameters across different budgets (25 to 1600) for the optimization algorithms used in this work. The budgets columns are given in red. These hyperparameters were optimized using the convolution, GEMM, and PnPoly kernels on the NVidia P100 GPU.

algorithms [215], similar to other computationally expensive applications such as neural architecture search [52].

The 9 GPUs that are used for testing are given in Table 5.1. The convolution kernel is implemented in CUDA, GEMM in OpenCL, and PnPoly is a heterogeneous kernel that runs partly on the CPU, and partly on the GPU using CUDA. The PnPoly kernel uses CUDA-specific features that are not available on AMD GPUs. We have used CUDA Version 11.2, OpenCL 1.2, Python 3.8.5, PyCUDA v2021.1, PyOpenCL v2020.3.1, and BlooPy version 0.4.2.

Experimental setup: The GPU kernels are tuned with respect to runtime (ms). The runtime of a GPU kernel is stochastic, and can vary slightly per execution. Kernel Tuner automatically benchmarks a given configuration 32 times to acquire a mean runtime per configuration. In most cases, the compilation time for a given kernel configuration significantly exceeds the time needed to benchmark 32 runs. Therefore, most of our experiments are performed in a deterministic setting where the fitness of a configuration is the mean runtime. However, we also perform a stochastic experiment where a single kernel runtime is returned for every evaluation. This means that the fitness for the same point in the search space can vary, and algorithms that learn stochastic information, such as irace and SMAC, can potentially benefit.

After discussion with the authors [140] we decided to benchmark irace only for the stochastic experiment. This was decided as it was deemed inappropriate for the deterministic setting since the point of using irace is to dynamically handle stochasticity in expensive problems.

The algorithms are evaluated based on the fraction of the optimal runtime they can find within a limited budget of evaluations. For each algorithm, we run experiments with a *maximum function evaluation* limit (budget) of 25, 50, 100, 200, 400, 800, 1600. The goal of our experiments is to benchmark algorithms when traversing only a fraction of the search space. Therefore, we set the highest budget limit at 1600 since it is already approximately 20% of the Point-in-polygon search space. Every run is performed 50 times in order to get an indication of the spread. Due to computational demand, SMAC and irace experiments are ran 20 times. SMAC is only run up to a budget of 400 evaluations due to the high tuning time. Data and scripts for the experiments and figures can be found in the GitHub repository [215].

5.6 Results: Benchmarking optimization algorithms on runtime

In this section, we first discuss how to initialize each algorithm with favourable hyperparameters. Next, we discuss which algorithms are best suited to tuning GPU kernels.

5.6.1 Setting hyperparameters

In order to compare the optimization algorithms fairly for the GPU tuning problem, we need to choose sensible hyperparameters. Which hyperparameters can be varied per algorithm is outlined in italics in section 5.3.3. We test different combinations of hyperparameters on the P100, RTX 2070 Super, and GTX 1080Ti, for all three kernels (9 out of 26 kernels). This way we select reasonable hyperparameters across various architectures and kernels, which users can use as defaults for new GPU tuning problems. We performed a bruteforce search over all combinations of parameter values, and ran each set 20 times for each algorithm. For PSO we kept the w, c_1, c_2, p parameters constant as these appeared to have little effect on algorithm performance. Note that the time required to tune hyperparameters varies greatly between algorithms due to this exhaustive search. The hyperparameters chosen are in Tables 5.3 and 5.4.

To choose hyperparameters, we first group settings which perform similarly statistically, and attempt to find one set of hyperparameters that performs well across all 9 kernels. For a budget p, let $f_{p,best}$ be the lowest average fitness achieved for a set of hyperparameters, and $\sigma_{p,best}$ the standard deviation. We perform the following selection approach:

1. For every kernel, we create a set of hyperparameter settings whose average found fitness is within $k \cdot \sigma_{p,best}$ of $f_{p,best}$.

2. For each budget, intersect the acceptable settings for convolution, GEMM, and PnPoly, across the 3 GPUs.

3. For each budget, if this intersection is non-empty, reduce k and repeat. If the intersection is empty, increase k and repeat. Repeat until only one set of hyperparameters remains in the intersection.

5.6.2 Kernel tuning algorithm comparison

To quantify which kernel tuning algorithms perform best for certain budgets, we can check whether an algorithm provided statistically significantly better results than others for a certain experiment. To do so, we use a *two-sample independent t-test* [62] with $\alpha = 0.05$. For each GPU, kernel, and budget combination, we perform the t-test to see if an algorithm A performed significantly better than algorithm B. We subsequently combine the total number of "wins" for algorithm A across all GPUs (excluding those that were used for tuning).

We split the competitions into low-range (200 evaluations or fewer), and medium-range budgets. The competition tables at different function evaluation splits can be found in the Appendix A.2.2. The full plots per GPU and algorithm can be found in Appendix A.2.3. The results of these inter-algorithm competitions are given in the heatmaps displayed in Figures 5.1, 5.2, and 5.3. The competition heatmaps display how often the column algorithm found a statistically better solution than the row algorithm in that budget range.

Kernel tuning: deterministic fitness

In this section we present the results of our deterministic experiments, i.e., the algorithms have to minimize the runtime of a GPU kernel where the runtime is fixed at the mean of 32 runs.

Low-range budget: For 200 function evaluations or fewer, for convolution dual annealing was statistically better than all other algorithms for all GPU models (see column with "DualAnnealing" for convolution $\leq$ 200 function evaluations), with simulated annealing as second. For GEMM, basin hopping and dual annealing perform equally with dual annealing beating basin hopping 5 times, and basin hopping beating dual annealing 6 times. For PnPoly dual annealing was again best, followed by SMAC. We show the total number of wins and losses across all kernels and GPUs in Table 5.5. Here we see that for low budgets dual annealing has significantly more wins, and fewer losses than all other algorithms.

Interestingly, SMAC was second for PnPoly, in the best half of algorithms for convolution, but did not even beat random sampling for GEMM. We hypothesize this is either due to the number of variables to optimize for each of the three kernels (see Table 5.2), or the increasing fraction of fail fitnesses for these kernels. We hypothesize that the Bayesian optimizer could not fit a proper surrogate for GEMM with a low budget, and many failed compilations.

Medium budget: For more than 200 function evaluations, FirstMLS and GLS performed best for convolution, followed by FirstILS. For GEMM, FirstILS and simulated annealing performed best, followed by GLS. For PnPoly, FirstMLS is the strongest algorithm, followed by simulated annealing and GLS. As can be seen from Table 5.5, FirstILS and simulated annealing have the most number of total wins, and simulated annealing and genetic local search have the least number of losses.

Additional remarks: In general, best-improvement local search algorithms performed significantly worse than the first-improvement variants. In fact, for the low range, they proved statistically worse than random sampling for PnPoly, and in general have fewer wins and more losses. This can be explained due to the fact that exploring all the neighbours before taking a step costs many evaluations, and leads to only exploring a single neighbourhood for low budgets. For the population-based methods, GLS is the best performing algorithm for medium budgets, but does worse than differential evolution and GA for low budgets. PSO performed significantly worse.

Interestingly, dual annealing, which works on real-valued solution vectors, performs well for low budgets. It seems that the mapping from $[0, 1]^n$ to discrete space does not prevent dual annealing from finding strong solutions quickly. One of the main drawbacks of using continuous algorithms is that if a continuous algorithm updates its real-valued solution vector, it could mean that it does not actually update the discrete solution vector it is mapped to. However, since our algorithms cache previously visited solutions (only for the deterministic experiments), such redundant optimization steps do not cost any budget. We think this may negatively impact gradient-based algorithms as the subroutines Powell and COBYLA, which

	Low budget		Medium budget	
	total wins	total losses	total wins	total losses
Basin hopping	403	204	131	393
Dual annealing	**680**	**65**	227	233
Differential evolution	426	192	150	347
PSO	290	327	136	368
FirstILS	352	233	**361**	77
BestILS	125	472	257	126
FirstTabu	148	430	255	183
BestTabu	35	677	220	185
FirstMLS	317	205	341	64
BestMLS	143	466	226	174
Simulated annealing	412	150	360	**59**
Genetic local search	320	216	329	**59**
Genetic algorithm	376	162	201	207
SMAC	356	344	48	145
Random sampling	222	463	7	629

Table 5.5: Total number of wins: sum of occurrences when the algorithm found statistically better solutions than other algorithms (summed over all kernels). A win (and corresponding loss for the other algorithm) is counted when 50 runs for a budget are statistically significantly better according to a two-sample independent t-test ($\alpha = 0.05$). Low budget is for ≤ 200 budgets, medium budget is for > 200 budgets. Top 3 cells are coloured green.

do not require derivatives to be known, are the selected solvers for dual annealing during hyperparameter tuning.

As a final remark, we notice that SMAC performs poorly in the medium budgets. Note that SMAC only has 1/3 as many data points in the medium budget since we do not perform the 800 and 1600 budget experiments for SMAC. Nevertheless the algorithm performs poorly on the 400 budget compared to other methods. It seems that SMAC is unsuccessful in fitting a meaningful surrogate model for kernel tuning. This could be due to the deterministic setup of this experiment, or due to the high number of fail configurations with "infinite" fitness.

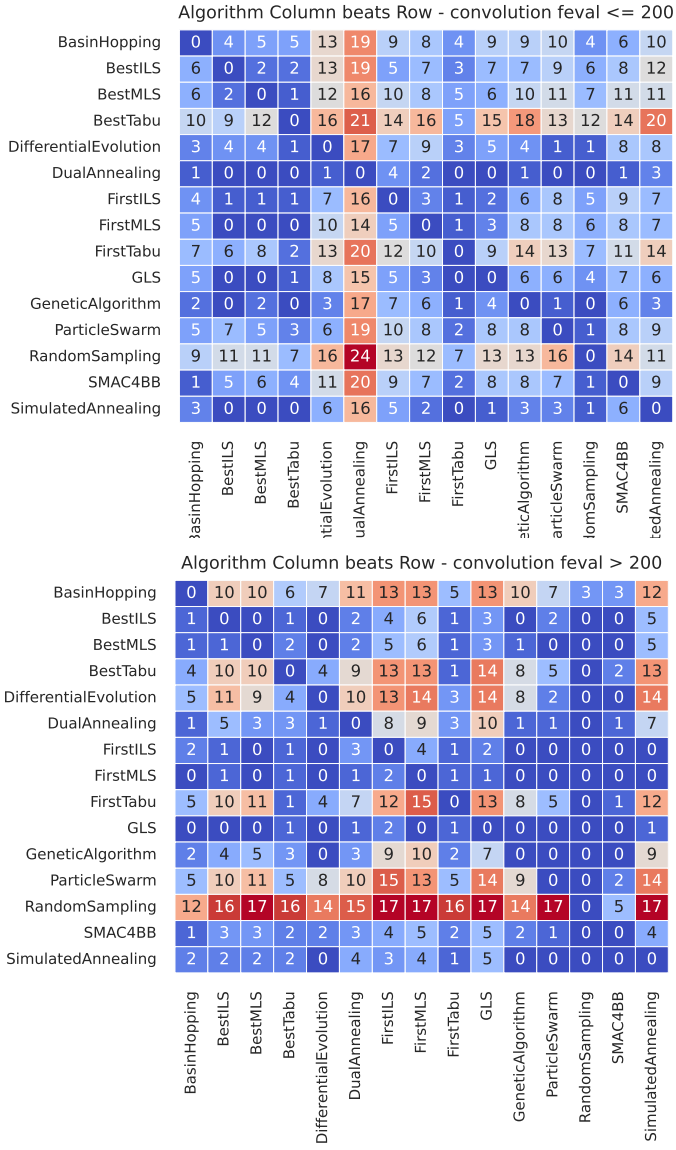


Figure 5.1: (Convolution:) Occurrences when the column algorithm found better solutions than the row algorithm. An occurrence is counted when 50 runs for a budget are statistically significantly better according to a two-sample independent t-test ($\alpha = 0.05$). (Top): Heatmap for low ≤ 200 budgets (25, 50, 100, 200). (Bottom): Heatmap for mid and high > 200 budgets (400, 800, 1600). Algorithms with low values (blue) in their rows were not often beaten for those budgets, and algorithms with high values in their column (red) often beat other algorithms.

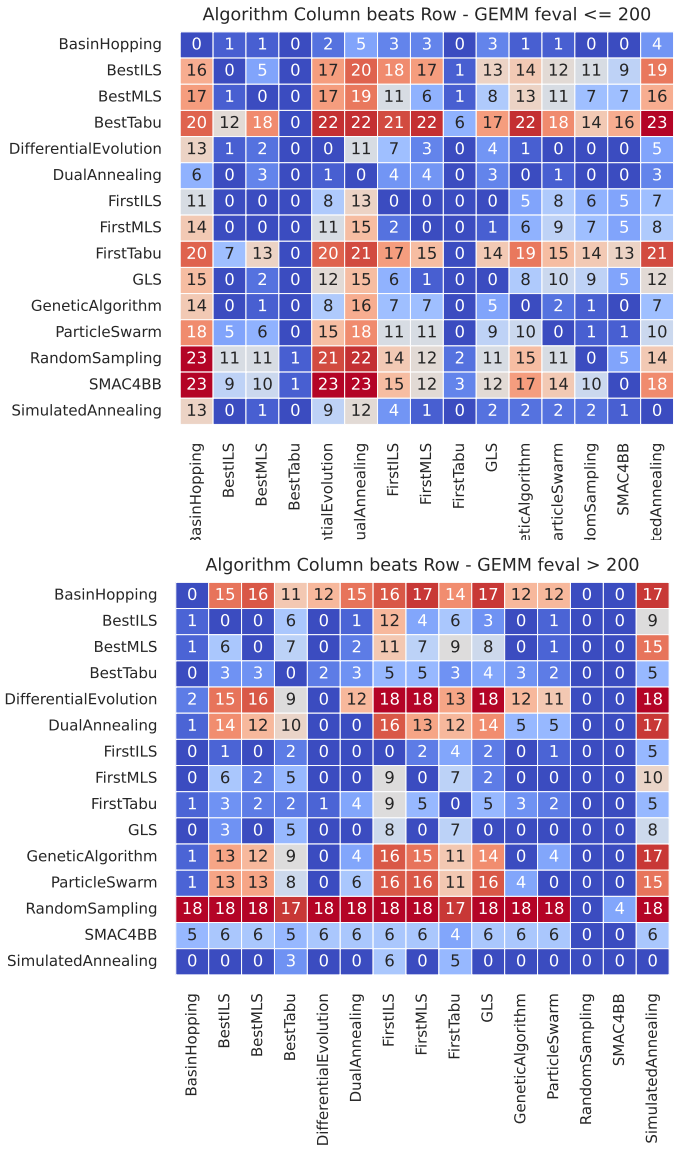


Figure 5.2: (GEMM:) Occurrences when the column algorithm found better solutions than the row algorithm. An occurrence is counted when 50 runs for a budget are statistically significantly better according to a two-sample independent t-test ($\alpha = 0.05$). (Top): Heatmap for low ≤ 200 budgets (25, 50, 100, 200). (Bottom): Heatmap for mid and high > 200 budgets (400, 800, 1600). Algorithms with low values (blue) in their rows were not often beaten for those budgets, and algorithms with high values in their column (red) often beat other algorithms.

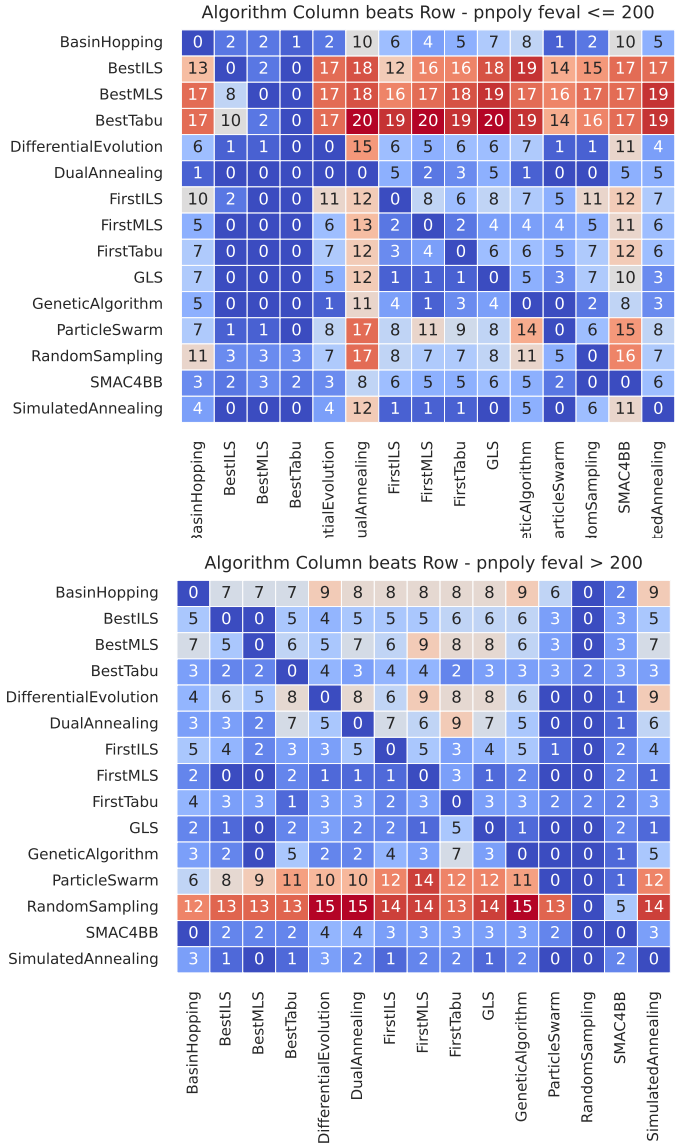


Figure 5.3: (PnPoly:) Occurrences when the column algorithm found better solutions than the row algorithm. An occurrence is counted when 50 runs for a budget are statistically significantly better according to a two-sample independent t-test ($\alpha = 0.05$). (Top): Heatmap for low ≤ 200 budgets (25, 50, 100, 200). (Bottom): Heatmap for mid and high > 200 budgets (400, 800, 1600). Algorithms with low values (blue) in their rows were not often beaten for those budgets, and algorithms with high values in their column (red) often beat other algorithms.

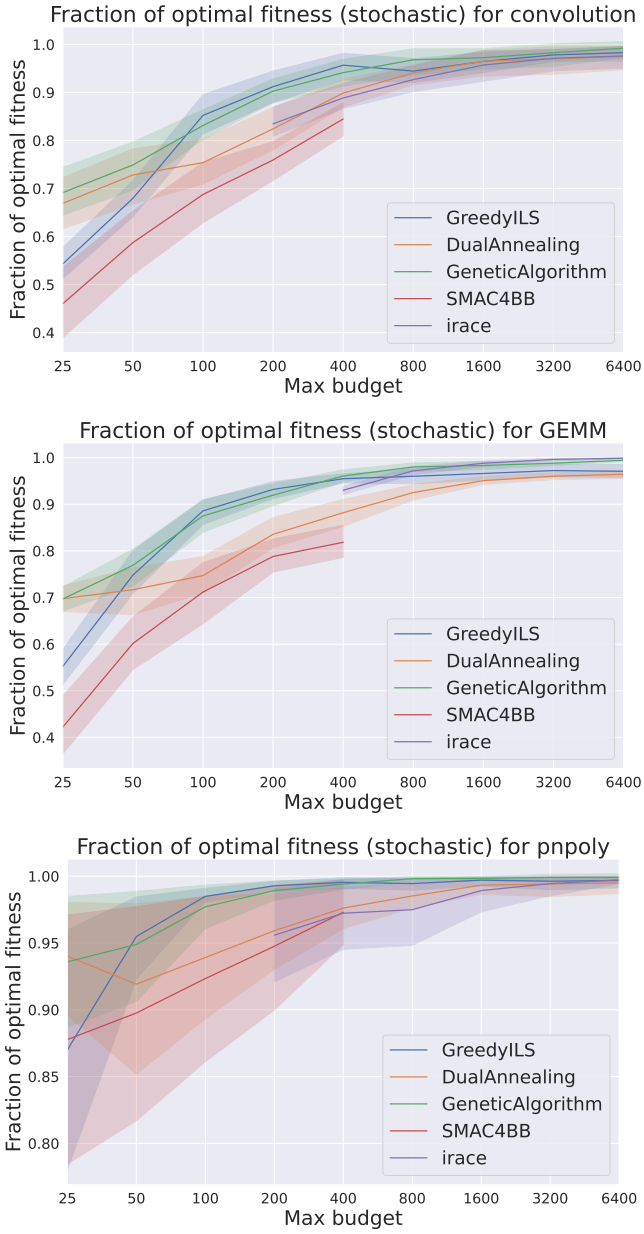


Figure 5.4: Fraction of optimal runtime for max budget supplied over all GPUs. Each point is the mean fraction of optimal runtime found (y-axis) for each budget limit (x-axis) over all GPUs, with the shaded region indicating 95% confidence interval. Left: convolution kernel. Middle: GEMM kernel. Right: PnPoly kernel (logarithmic x-axis).

Kernel tuning: stochastic fitness

For the stochastic experiments the algorithms have to minimize the runtime of a GPU kernel where the runtime is a random draw from the 32 timings. In addition to running SMAC and irace, we also run FirstILS, GA and dual annealing which did well for deterministic fitnesses. We remark that irace throws an error if the budget is too small with respect to the number of variables, and therefore starts at a budget of 200 for convolution and PnPoly, and 400 for GEMM.

The experimental results are shown in Figure 5.4. Here we aggregate the results per kernel for all GPU models by showing the mean fraction of optimum (and 95% confidence interval) for a given max budget. We see that GA and dual annealing are best for low budgets in the stochastic experiments. FirstILS does well for budgets ≥ 100. Irace is the best method for GEMM with budgets ≥ 800, but for convolution and pnpoly irace is not as good as GA, dual annealing, and FirstILS.

SMAC consistently achieves a lower fraction of optimality than the competing algorithms across kernels and budgets. Again, we hypothesize that this is because of the high number of fail fitnesses in the search spaces (see Table 5.2). This makes it hard for the Bayesian optimizer to fit a meaningful surrogate.

Stochastic or deterministic: Overall, we notice that higher budgets are necessary to find good solutions for the stochastic experiments than in the deterministic case. This leads to a higher overall tuning time. We therefore recommend to treat GPU kernel tuning as a deterministic optimization problem, with the mean runtime as fitness. The added stochastic information does not appear to allow SMAC or irace to consistently outperform conventional black-box algorithms. This could be because the runtime does not vary much; the average (normalized) runtime and standard deviation is 1.000 ± 0.011. Second, the high number of failure configurations could confuse models that try to learn stochastic information.

5.7 Quantifying GPU tuning difficulty

In this section we want to gain insight into the difficulty of the GPU kernel tuning optimization problem, and quantify kernel spaces according to tuning difficulty. When attempting to understand why certain GPU kernel spaces appear difficult to optimize we found that relatively simple metrics do not coincide with our experimental results. We outline discrepancies between an intuitive simple metric and our experimental results, and introduce a novel refined approach that does correlate with our results.

5.7.1 Naive metric: fraction of optimal fitness of local minima

Method: As an example of a simple metric that intuitively could explain the results, we consider the fraction of optimal fitness of local minima. For a minimum x_i, and optimal fitness f_{opt}, we can consider the *fraction of optimal fitness* of the minimum $f_{opt}/f(x_i)$. In this case, we divide the global minimal runtime by the runtime of the minima.

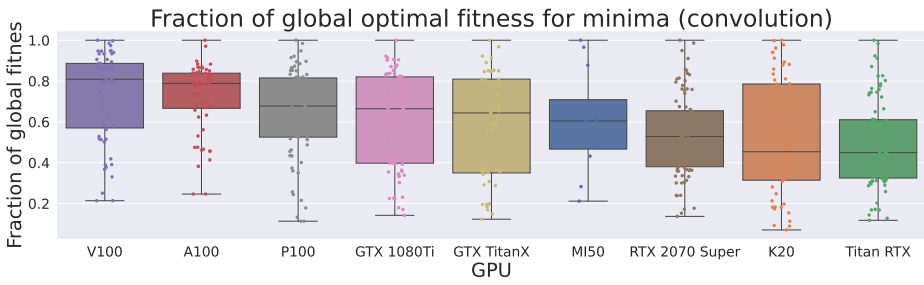


Figure 5.5: Fraction of optimal fitness of local minima for each GPU model, for the convolution kernel. The box plots shows the median line, the box designates the quartiles, and the whiskers the full extend of the distribution. Additionally, a scatter plot of the fitness for each local minimum is shown. The GPUs are ordered in descending median fraction of optimal fitness from left to right.

Results: In Figure 5.5 we show a scatter plot of the fraction of optimal fitness for the local minima for the convolution kernel (per GPU model). According to this distribution, the V100 and A100 GPUs have the closest to optimal median fitness for local minima. This means that an algorithm that randomly explores local minima with equal probability will obtain the closest to optimal runtime for these kernels.

Analysis: To empirically check how difficult the GPU kernels are to tune, we can plot the fraction of optimal fitness that optimization algorithms managed to achieve for certain budgets. If f_j is the lowest fitness found for a single run for some budget p, a point in the plot is the average over 50 runs computed as $\tilde{f}_p := (1/50) \cdot \sum_{j=1}^{50} (f_{opt}/f_{j,p})$. In Figures 5.6 and 5.7 we plot $\tilde{f}_p$ for dual annealing and FirstILS. We chose dual annealing and FirstILS as they represent the strongest algorithms for low, and medium budgets respectively.

We see that for convolution on the A100 GPU both algorithms returned solutions which were furthest away from the optimum, while for the V100 both optimizers return close to optimal solutions for few function evaluations. These observations are opposite to what would be expected on the basis of Figure 5.5. Hence, the distribution of fitness of the local optima does not properly explain tuning difficulty.

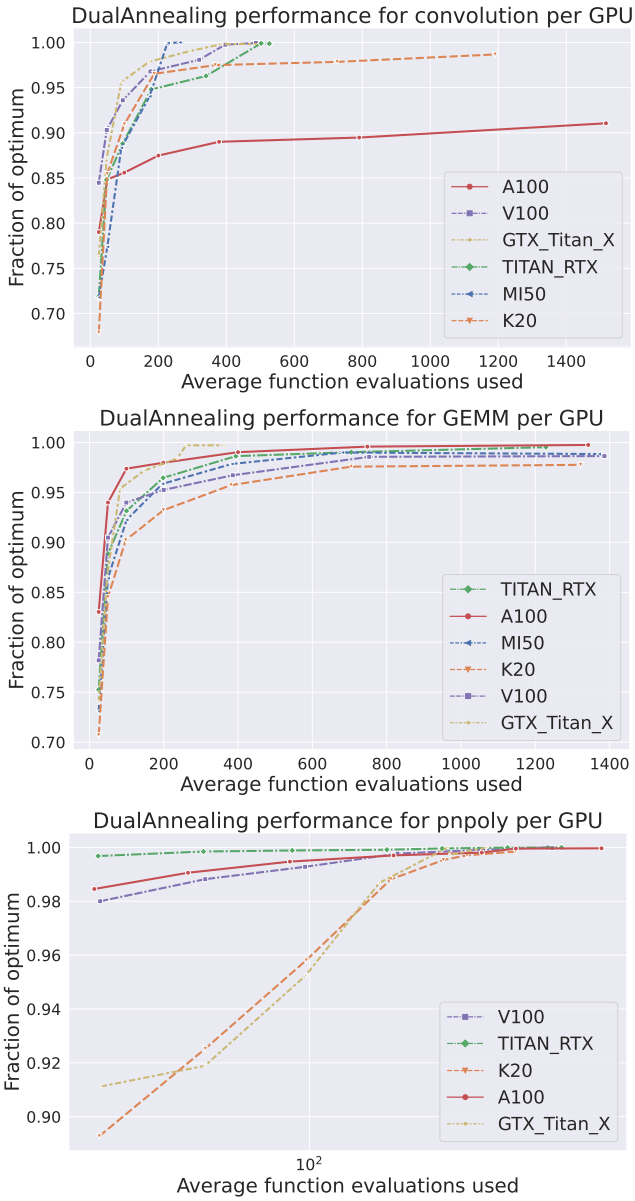


Figure 5.6: **Dual annealing:** Fraction of optimal runtime for different budgets (per GPU). Each point is the average fraction of optimal runtime found (y-axis) for each budget, with respect to the average number of evaluations actually used (x-axis) for that budget (counts only the visited unique settings). Left: convolution kernel. Middle: GEMM kernel. Right: PnPoly kernel (logarithmic x-axis).

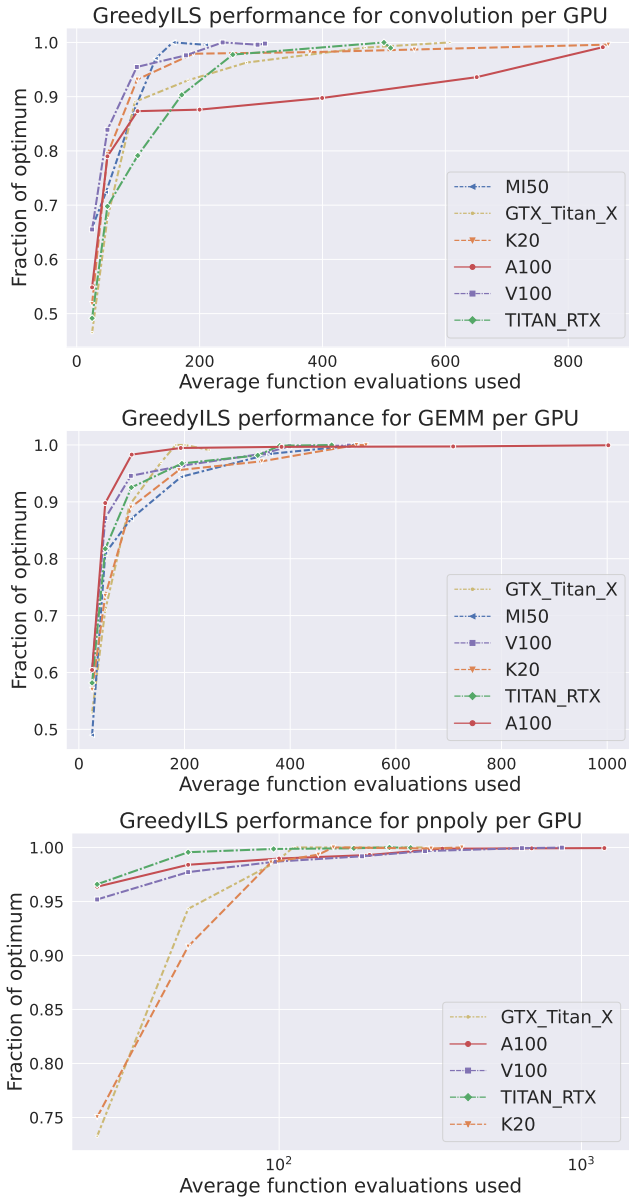


Figure 5.7: **FirstILS:** Fraction of optimal runtime for different budgets (per GPU). Each point is the average fraction of optimal runtime found (y-axis) for each budget, with respect to the average number of evaluations actually used (x-axis) for that budget (counts only the visited unique settings). Left: convolution kernel. Middle: GEMM kernel. Right: PnPoly kernel (logarithmic x-axis).

5.7.2 Refined approach: Fitness flow graphs and PageRank

Method: A more refined metric to quantify GPU tuning difficulty may be to compute how likely local search algorithms terminate in local minima. For this purpose, we introduce the *fitness flow graph* (FFG), which contains all points in the search space, and creates a directed edge to a neighbouring point if the neighbour has lower fitness. This means that a random walk across the FFG mimics the behaviour of a randomized first-improvement local search algorithm. The expected proportion of arrivals of each minimum then gives a metric for weighting reachability of each minimum. We show two example FFGs in Figure 5.8.

To compute the likelihood of arrival per local minima, we compute the PageRank *node centrality*, which was originally used to determine the relevance of a webpage [21, 173]. Let A_G be the adjacency matrix of a directed graph G, rescaled such that each column adds up to 1. Essentially, this means that for every node, the column is a probability vector of visiting adjacent nodes with equal likelihood. The PageRank values are then the values of the dominant right eigenvector of A_G. For an FFG, this means that the PageRank value of a local minimum is the probability of arriving in that minimum after a long random walk through the graph.

As a measure of difficulty we consider how likely a certain subset of "suitably good" local minima are to be visited by a local search algorithm relative to the rest. Suppose that f_{opt} is the optimal fitness, and let $L(X)$ be the set of local minima of X. Given a proportion p, we take the set of nodes $L_p(X)$ consisting of local minima with fitness less than $(1+p)f_{opt}$ (for minimization problems, otherwise $(1-p)f_{opt}$). For a centrality function c_G, we define the *p-proportion of centrality*

$$C_p(G, X) = \frac{\sum_{x \in L_p(X)} c_G(x)}{\sum_{x \in L(X)} c_G(x)}. \tag{5.3}$$

Results: The proportion of centrality for strong local minima for each FFG is shown in Figure 5.9. We calculate the proportion of centrality for different acceptance percentages with respect to the global minimum of $p = 0, 1, 2, \ldots, 15\%$.

Revisiting the A100 and V100 convolution kernel comparison, we see that the proportion of centrality matches the experimental observations for dual annealing and FirstILS. Figure 5.9 shows that the NVidia V100 has the most central local minima, whereas the A100 has the least central local minima. For the GEMM and PnPoly kernels, Figures 5.6 and 5.7 align with the expectations based on the proportion of centrality. For example, the group of PnPoly kernels with lowest proportion of centrality (P100, GTX Titan X, K20, GTX 1080Ti) are indeed the hardest to tune for both algorithms.

One exception is the K20 GEMM kernel, where proportion of centrality does not entirely reflect the perceived difficulty for dual annealing. This suggests that the proportion of centrality may correlate with GPU tuning difficulty better for certain optimization algorithms. This is to be expected since the PageRank centrality on the FFG in expectation mimics the performance of randomized first-improvement local search. Algorithms that are substantially different than first-improvement

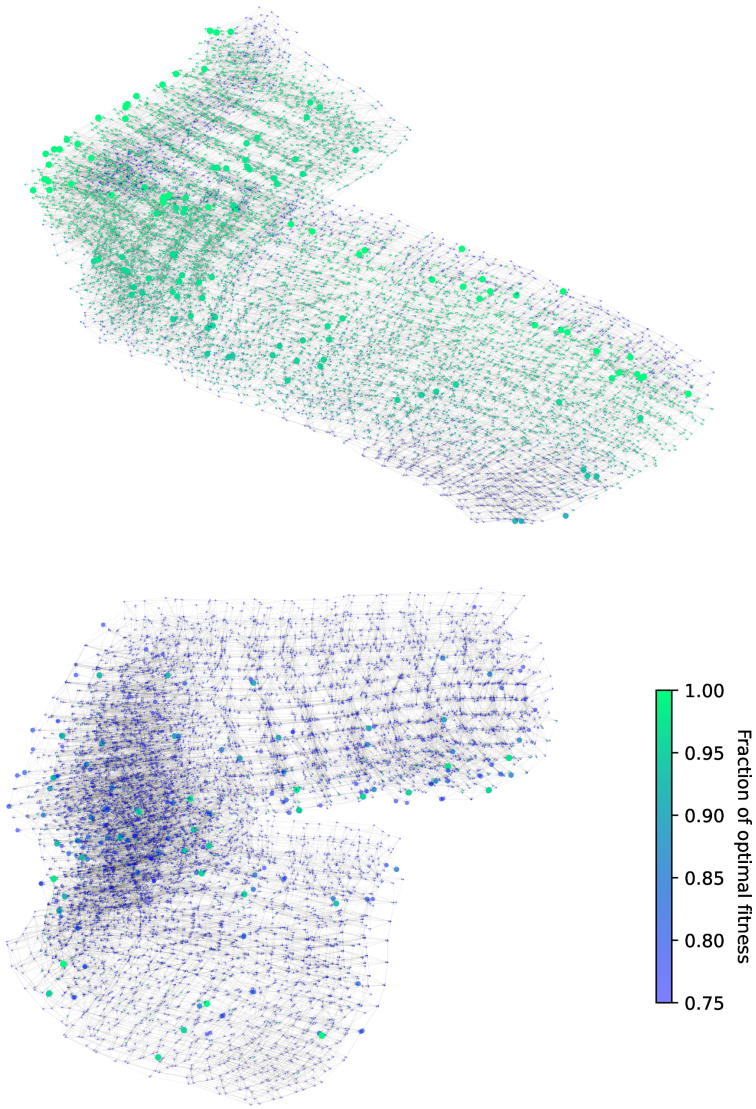


Figure 5.8: Fitness flow graphs of PnPoly kernel search spaces of the (top) NVidia Titan RTX, and (bottom) NVidia GTX 1080Ti. Each node is a point in the search space. There is a directed edge between neighbouring points from higher to lower fitness. Points are coloured within a fitness range of +25% 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.

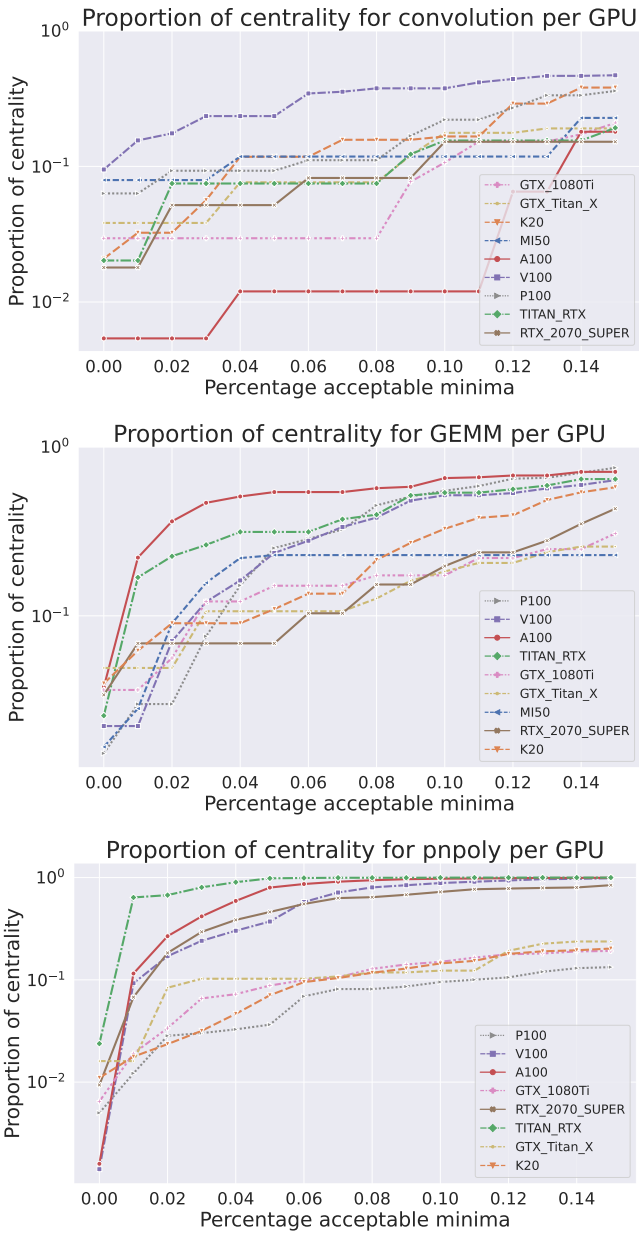


Figure 5.9: Proportion of centrality for FFGs for each GPU. The proportion of centrality is computed by taking the sum of PageRank centrality for local minima within p% of the optimal fitness, divided by the total PageRank centrality of all local minima. From top to bottom are convolution, GEMM, and PnPoly kernel.

local search will therefore also correlate less with the expected difficulty on the basis of proportion of centrality.

Analysis: Overall, the experimental results suggest that the proportion of centrality is a suitable metric for estimating tuning difficulty for GPU kernels. By using FFGs and the PageRank algorithm, we are able to observe kernel differences that were otherwise unknown. For example, both the A100 and V100 convolution kernels have few outlier minima with a close to optimal fitness. In fact, the existence of only a few kernel configurations that lead to large increases in performance is a general property of certain GPU kernels [246]. Crucially however, the likelihood of local search algorithms arriving in such minima differs greatly between the A100 and V100. The proportion of centrality of an FFG gives us a tool to quantify this likelihood. However, further research is necessary to quantitatively determine how well our proposed metric correlates with GPU tuning difficulty.

As a final remark on kernel differences, the experimental results shows that the difficulty of tuning a particular kernel can greatly differ from one GPU to the next, and that these changes do not appear to be correlated with release time of the models. The A100 is the most recent GPU in our set, while the K20 is the oldest. For GEMM and PnPoly, we can say that it has become easier to tune these kernels with more recent GPUs, but the convolution kernel has become more difficult to tune, except on the V100.

5.8 Conclusion

In this chapter, we have investigated which optimization algorithms produce the fastest GPU kernel configurations across different tuning-time ranges. To do so, we analyzed 26 GPU kernel spaces for 9 GPUs. We computed sets of optimal hyperparameters for GPU tuning for each optimization algorithm. From among the tested algorithms in this set of experiments, we conclude that dual annealing performs best as GPU kernel tuner when a limited amount of function evaluations is desirable. When more evaluations are possible, first-improvement local searchers such as FirstILS proved the best GPU kernel tuners. Using these algorithms, we are convinced that GPU programmers can reliably auto-tune GPU kernels to close to optimal runtime while requiring relatively few re-compilations of the code. Furthermore, we conclude that treating GPU tuning as a deterministic optimization problem is preferred over treating the runtime as a stochastic variable.

We showed that the basic metric of fraction of optimality of local minima is not suitable for explaining the results observed in the experimental benchmarks. To make steps towards a metric for tuning difficulty, we introduced the concept of fitness flow graphs, and proportion of centrality. Our results suggest that the proportion of centrality can be used to quantify tuning difficulty. For future work, in cases where exhaustive exploration is infeasible, perhaps a procedure to dynamically update the proportion of centrality of FFGs can be used. Such dynamic estimates of tuning difficulty could be used for automatic algorithm selection within frameworks such as Kernel Tuner. Furthermore, the pagerank centrality of strong

local minima within FFGs can be used to investigate why certain minima are unlikely to be visited, for example because neighbouring configurations fail to compile. Lastly, in this work we fully computed 26 kernel spaces, and made these publicly available. We aim to extend this to a benchmark dataset for evolutionary computation algorithms.

6

Going green: optimizing GPUs for energy efficiency through model-steered auto-tuning

6.1 Introduction

Huge amounts of compute power are powering today's industrial and scientific applications, at huge energy and environmental costs. Energy is among the largest expenses of supercomputers and data centres, and this consumption will double every four years [43]. The computational demands in deep learning (artificial intelligence) applications have been increasing at a exponential rate, 300,000× from 2012 to 2018 [217]. The carbon footprint of these applications is a great concern for the environment, as training a single large model produces as much carbon dioxide as five cars in their lifetime, including fuel [229]. In addition, many applications have stringent energy constraints; embedded and automotive systems have limited battery capacity, offshore applications where a connection to the power grid is not possible, and also large-scale scientific instruments, such as the Square Kilometre Array (SKA) built partially in the desert [47]. Graphics Processing Units (GPUs) are powering nearly all large-scale AI and HPC applications, and are in large part responsible for the total power consumption of these systems [182, 253]. For instance, 8.3 MW out of the total 13 MW by the Summit Supercomputer is consumed by its GPUs [227]. There is a clear urgency to improving the energy

efficiency of these applications.

While GPUs are relatively energy-efficient processors, energy consumption greatly depends on how well the application is optimized to efficiently use the underlying hardware [51, 127]. The optimization of GPU applications is a complex problem that requires finding the best performing combination of many implementation choices and code optimization parameters in a large and discontinuous search space [128, 166, 205, 226]. As such, auto-tuning, the process of automatically searching for the best performing configuration, is often used to optimize the compute performance of these applications [71, 145, 235, 259].

This has led to the rise of generic GPU code auto-tuners, such as CLTune [166], Kernel Tuner [246], Kernel Tuning Toolkit (KTT) [60], and Auto-Tuning Framework (ATF) [195], which facilitate the creation of auto-tuned GPU applications, and support different optimization strategies to accelerate the search process. These frameworks focus on auto-tuning user-defined code parameterizations, which is more generic and powerful than compiler-based auto-tuning [7], because it allows users to tune for entirely different ways to parallelize a computation, with different algorithms to compare, and different data layouts, loop permutations, and code optimizations. However, none of these generic GPU auto-tuners has built-in support for energy optimization, and the differences between auto-tuning for compute performance and energy efficiency have not yet been studied in detail.

In this chapter, we introduce new energy monitoring capabilities in Kernel Tuner, which allows us to use the existing frameworks to study and optimize energy efficiency. We use these capabilities to investigate how different compute performance tuning (lowest kernel runtime) is from energy tuning, and whether the tuning difficulty differs from the perspective of blind optimization algorithms. In addition, we compare two methods for tuning energy efficiency of GPUs; power capping and fixing clock frequencies. Lastly, we introduce a method to efficiently model GPU power consumption, which allows us to significantly narrow the range of clock frequencies to search for the most energy efficient configuration. All together, we provide a method and open-source tool for tuning GPU applications for both performance and/or energy efficiency. Moreover, these tools can be used for further auto-tuning and high performance computing research.

6.2 Related Work

OpenTuner [5] was one of the first generic software auto-tuning frameworks, supporting a number of different search optimization algorithms, but lacks support for tuning individual GPU kernels. CLTune [166] was one of the first of a new breed of generic auto-tuning tools with specific support for tuning GPU kernels written in OpenCL. Kernel Tuning Toolkit (KTT) [60] is developed specifically to support online auto-tuning and pipeline tuning, which allows for exploration of combinations of tunable parameters over multiple kernels. An interesting feature of KTT is its support for keeping track of hardware performance counters during benchmarking, which can also be used in advanced search strategies [61]. Auto-

Tuning Framework (ATF) [195] implements a way to generate search spaces, using a chain-of-tree search space structure for efficient storage and fast exploration of constrained search spaces. HyperMapper [159] is a tuning framework that focuses on multi-objective optimization and exploitation of user prior knowledge. Kernel Tuner [246] is specifically designed to be an easy-to-use and easy to extend tool for the development of tunable GPU kernels, and in particular supports a large selection of search optimization strategies. In this chapter, we extend Kernel Tuner [246] with functionality for auto-tuning energy efficiency, which cannot be found in any of the existing generic auto-tuning frameworks.

Research in auto-tuning GPU applications for energy efficiency is still in its infancy, despite spanning more than 12 years of research. There is no state-of-the-art method for GPU energy tuning, as comparisons between studies or even to a shared baseline are non-existent. The majority of studies only tune individual parameters, e.g. thread block dimensions [40, 95, 129, 177, 233, 244], or clock frequencies [4, 29, 58, 64, 147, 191]. Only two studies actually combine auto-tuning code optimizations with execution parameters, such as clock frequencies, but only for a single application on a single GPU [41, 153].

All generic auto-tuning frameworks use empirical performance measurements, most likely because it is difficult to create generalized performance models that capture the complex system that arises from the combination of hardware and software [30, 192, 207]. Some GPU energy tuning studies use highly-inaccurate performance models, with up to 50% error, to estimate energy consumption without evaluating the impact of these inaccuracies on the auto-tuning results [104, 129]. Therefore, most studies take an empirical approach, in particular using the GPU's internal power sensor [4, 58, 74, 83, 126, 191, 208], but also through external power sensors [44, 79, 99, 107, 197, 230] often based on custom-built measurement equipment. Internal power sensors are included in most modern GPUs and can be read by software, e.g., using the NVIDIA Management Library (NVML) for NVIDIA GPUs. Such power sensors are therefore highly accessible, but may suffer from low sampling frequencies and low accuracy [200]. Some researchers try to compensate for these limitations by measuring individual functions for long periods of time [6, 182, 191]. This approach, however, is impractical for use in auto-tuners, which often have to benchmark many configurations to find the optimum [220]. As such, Kernel Tuner supports an external power sensor, namely PowerSensor2 [200], which is accurate within 1% error and at a sampling frequency of 2.87 kHz. This means that PowerSensor2 is capable of accurately measuring the energy consumption of a kernel without the need to prolong the kernel execution time. We have used PowerSensor2 to validate the power measurements taken using NVML.

Many studies claim that there is a clear difference between the optimization objectives of compute performance and energy efficiency, and that the two require different optimization algorithms and parameters [33, 41, 58, 95, 117, 153]. However, such claims are often not experimentally verified. The relationship between performance and energy efficiency is complicated, and many authors simply optimize energy efficiency by minimizing the kernel execution time, an approach that is

sometimes referred to as race-to-idle [6]. In [37], a model for energy is proposed that predicts that energy usage differs from runtime because energy costs for memory operations cannot be hidden while the algorithm is running. Therefore, energy optimality does not depend solely on optimizing FLOPs, but also on balancing energy usage between memory and compute operations. In this chapter, we aim to experimentally verify the differences between tuning for compute performance and energy efficiency.

6.3 Methodology

6.3.1 GPU power consumption model

The energy consumed by a GPU over a time interval $[t_0, t_1]$ is related to its power usage $P(t)$ according to

$$E = \int_{t_0}^{t_1} P(t)\, dt.$$

The power consumption $P(t) = V(t)I(t)$ can be determined by measuring the current I, and voltage V. In practice, one can either approximate the integral numerically by, e.g., trapezoidal integration using the power readings, or simply multiplying the average power consumption by the elapsed time $E = \langle P \rangle (t_1 - t_0)$. We employ the latter method in this work, where we take the median power reading for $\langle P \rangle$.

The power consumption of a GPU is affected by several factors, including the workload and operating frequency of the GPU. The workload is implementation dependent, and in most cases can be optimized by tuning kernel parameters, or by changing the kernel code. Furthermore, different GPU models contain different components, such as memory and chips, that operate at certain clock frequencies which can vary at runtime. These operating frequencies are commonly taken as is.

Throughout this work, we use a variety of GPUs with distinct architectures. Moreover, even within one architecture (e.g. the Ampere architecture) we cannot assume that the energy characteristics of two different models are identical. The Tesla A100 and RTX A4000 GPUs for instance use a different chip (GA100 versus GA102), are produced at a different process size (7 nm versus 8 nm), and have a very different mix and number of execution units. Moreover, the Tesla A100 has HBM2e memory, while the RTX A4000 uses GDDR6. The NVIDIA drivers currently do not expose an option to tune the clock frequency of the HBM memory. For the RTX A4000 and a compute-bound kernel, we measured only a marginally lower energy consumption when reducing the memory clock frequency. Therefore, we consider solely the graphics clock (core) frequency in this work.

Contemporary GPUs usually operate at a base core frequency and can boost up to a certain turbo frequency to increase performance, but only when the temperature and power consumption of the device allows for it. This technique is commonly referred to as Dynamic Voltage Frequency Scaling (DVFS). Price

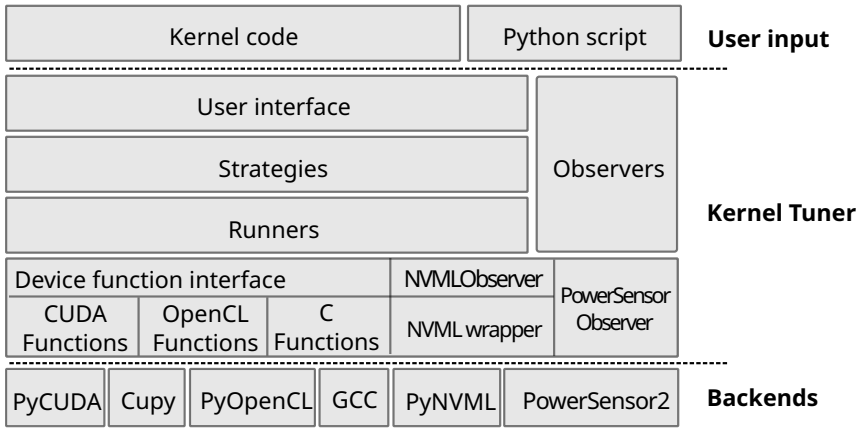


Figure 6.1: Extended software architecture of Kernel Tuner.

et al. [191] showed a relation between core frequency and the voltage required to operate on a given frequency, and a power consumption model is given by

$$P_{gpu} = P_{static} + N_c C f V^2, \tag{6.1}$$

where C is load capacitance, N_c the number of switches, f is frequency, and V is voltage. V typically increases with f. Consequently, the turbo frequency may be good for performance, but not necessarily for energy efficiency.

To steer frequency tuning, we fit a GPU power consumption model to data in section 6.5.4, using a non-linear least squares approach (Levenberg-Marquardt algorithm [155]).

6.3.2 Energy measurements in Kernel Tuner

We introduce several new features in Kernel Tuner to acquire energy measurements of GPU kernel executions, namely observers, user-defined metrics, and custom tuning objectives. The software architecture and basic functionality of Kernel Tuner is described in [246], and a diagram of software hierarchy can be found in Figure 6.1. An observer can be implemented to execute functions and can extend results obtained during benchmarking before, during and after kernel execution. For the experiments in this work, we implemented the `NVMLObserver` and `PowerSensorObserver` in Kernel Tuner.

PowerSensorObserver

To facilitate accurate energy measurements at high sampling frequency, we implemented the `PowerSensorObserver` (using PyBind11[1]) as an interface to PowerSensor2 [200]. The user can select this observer to record power and/or energy

[1] https://pybind11.readthedocs.io/en/stable/

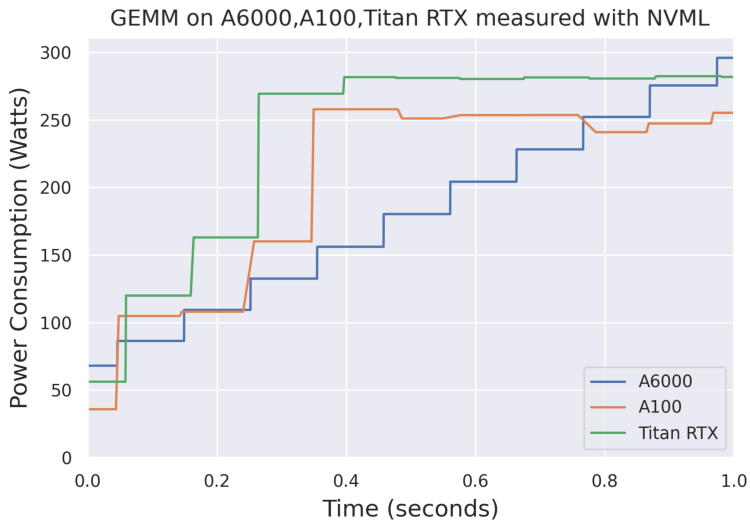


Figure 6.2: NVML power readings while executing matrix multiplication kernel (GEMM) over time on three different GPUs.

consumption of kernel configurations during auto-tuning. This allows Kernel Tuner to accurately determine the power and energy consumption of all kernel configurations it benchmarks during auto-tuning.

NVMLObserver

Measurements with the PowerSensor2 require wiring external hardware to a GPU, and the sensor is not available to most users, the bulk of our measurements will be performed using NVIDIA's internal sensors. The NVIDIA Management Library (NVML) [168] can be used for power measurements on almost all NVIDIA GPUs, so using this library is much more accessible to end-users compared to solutions that require custom hardware, such as PowerSensor2. To this end we implemented the `NVMLObserver` in Kernel Tuner, which allows the user to observe the power usage, energy consumption, core and memory frequencies, core voltage and temperature as reported by NVML.

As opposed to PowerSensor2, the power usage reported by NVML has a significantly lower temporal resolution. Furthermore, NVML only reports a time-averaged power consumption rather than instantaneous power consumption [26].

Figure 6.2 shows the GPU power consumption over time as reported by NVML, while continuously executing a matrix multiplication kernel (GEMM see section 6.4) for one second. The jumps in the graph are caused by the fact that the time-averaged value reported by NVML only refreshes at a frequency of about 10 Hz (9.75 Hz on RTX A6000, 14.5 Hz on Tesla A100, and 12.4 Hz on Titan RTX). We can see that on the Titan RTX and Tesla A100, the power consumption

GPU	Architecture	Cores	Bandwidth	Peak SP	TDP (W)
RTX A4000	Ampere (GA104)	6,144	448	19,170	140
RTX A6000	Ampere (GA104)	10,752	768	38,709	300
Tesla A100	Ampere (GA100)	6,912	1,555	19,500	250
Tesla V100	Volta (GV100)	5,120	900	14,028	250
Titan RTX	Turing (TU102)	4,608	672	16,312	320

Table 6.1: GPUs used in our experiments. Bandwidth in GB/s. Peak compute performance in GFLOP/s. TDP in Watts.

as report by NVML stabilizes after about 0.3 seconds into the run. For the RTX A6000, power consumption gradually ramps up until hitting the Thermal Design Power (TDP) right before the end of our 1-second interval.

To ensure that the NVML power measurements in Kernel Tuner more accurately reflect the power consumption of the kernel, the `NVMLObserver` executes the kernel repeatedly for a user-specified duration (1 second by default), and takes the final energy measurement, thereby ensuring a more accurate measurement with NVML. The downside of this approach is that it significantly increases benchmarking time.

6.3.3 Tunable parameters and objectives for energy tuning

Using application-specific clock frequencies is one of the most common approaches to tuning energy efficiency on GPU systems. Recently, Krzywaniak and Czarnul [117] have shown promising results with setting application-specific power limits, also called *power capping*, to optimize energy consumption. For this work, we have implemented support in Kernel Tuner for users to tune their applications under different clock frequencies and power limits. Specifically, NVML tunable parameters, such as `nvml_gr_clock`, `nvml_mem_clock`, and `nvml_pwr_limit`, can be set using Kernel Tuner. Note that changing these settings requires root privileges on most systems. As such, these features may not be available to all users on all systems.

Lastly, to perform energy tuning, we need to specify metrics that we aim to minimize or maximize. Using the aforementioned observers, we can collect power readings (in Watts) during kernel execution. Furthermore, Kernel Tuner's flexible *user-defined metrics* allows us to define other metrics such as *compute performance* in floating point operations per second (GFLOP/s). This allows us to define *energy efficiency* as GFLOPs/W (same as GFLOP/J) which is a measure of the energy used to perform a billion floating point operations.

6.4 Experimental setup

To investigate energy tuning on GPUs, we run several real-world applicable kernel programs, on a few different GPUs available in either the DAS-6 cluster (Turing and Ampere architecture) [10], or in the LOFAR COBALT-2 correlator system

(Tesla V100) [22]. Table 6.1 lists the properties of these GPUs. In addition to the widely-used GEMM kernel, we validate our results on several computationally expensive radio astronomy kernels currently processing data for the Low Frequency Array (LOFAR) radio-telescope [78]. These kernels will be used in section 6.5.5 to determine the practically obtained energy reduction for a real-world application. All kernels are compute-bound, except for the TDD kernel which is memory-bound. For the experiments in this section,

GEMM (Generalized dense matrix–matrix multiplication) is one of the most widely-used kernels across many application domains, including neural networks. Here we perform the calculation $C = \alpha A \cdot B + \beta C$ for 4096×4096 matrices A, B, C, and constants α and β. We use the highly-tunable OpenCL implementation available in CLBlast [165].

The CLBlast GEMM kernel can be tuned with many parameters, here we summarize the most important ones:

- M_{wg}, N_{wg}, and K_{wg} represent the total size of the tile processed by a single thread block in the M, N, and K matrix dimensions.
- M_{dimC} and N_{dimC} are the thread block dimensions in M and N.
- SA and SB can be used to enable or disable using shared memory as a software managed cache for matrix A and matrix B.
- M_{vec} and N_{vec} are the vector widths for loading and storing to global memory, M_{vec} is used for matrices A and C, and N_{vec} for matrix B.
- K_{wi} is the unrolling factor used for the loop over K.

While the GEMM kernel can use several code optimizations, none of the code optimizations have been introduced to optimize the kernel specifically for energy efficiency. All tunable parameters combined describe a large space, of which many portions are restricted. Using the parameters employed by CLBlast, the search space consists of 17472 valid kernel configurations, that will all be compiled and benchmarked when performing an exhaustive search. However, when we add additional tunable parameters for energy tuning, such as a power limit or clock frequency, the search space grows combinatorially from a grid search perspective. For example, if we want to tune all parameters in the search space in combination with 7 different clock frequencies, the total size of the search space becomes $17{,}472 \times 7 = 122{,}304$.

LOFAR Correlator is the correlator application used for real-time processing of LOFAR (Low Frequency Array) data [78]. It combines measurements from the radio telescope into a data product to be processed further by other (offline) processing pipelines (see other kernels). The correlator kernel was tuned by hand for the Kepler architecture, e.g. by unrolling loops and using fixed block and grid dimensions. Consequently, there is only a single tuning parameter left: `NR_STATIONS_PER_THREAD`. This parameter is used to choose between one of four different kernels.

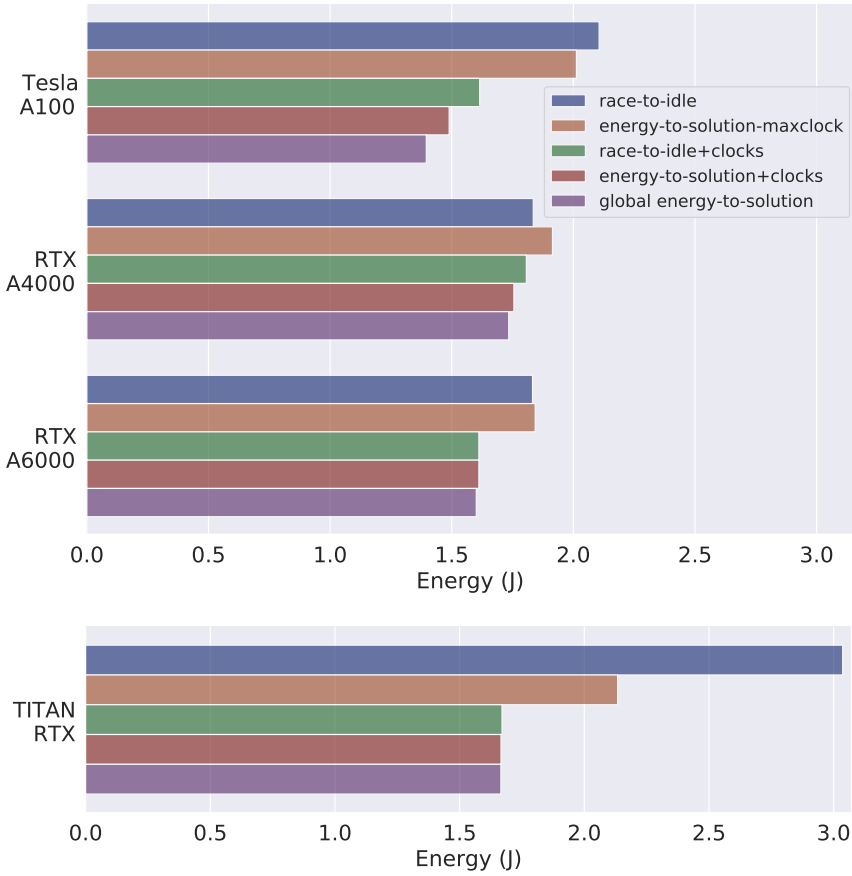


Figure 6.3: **GEMM:** Lowest energy configuration for the Tesla A100, RTX A4000, RTX A6000, and TITAN RTX GPUs for the *race-to-idle*, *energy-to-solution-maxclock*, *race-to-idle+clocks*, *energy-to-solution+clocks*, and *global energy-to-solution* tuning methods. The energy measurements for the TITAN RTX were acquired using PowerSensor2, the others using NVML.

TCC (Tensor-Core Correlator) is similar to the LOFAR correlator, leveraging the Tensor Cores of contemporary NVIDIA GPUs [201]. Tensor Cores are mixed-precision compute units that operate on matrix-like inputs. By using these compute units, the Tensor-Core correlator is both much faster and much more energy-efficient compared to previous correlators. This kernel is hand-tuned and uses fixed thread block dimensions. There is one tuning-parameter: `PORTABLE`, which determines whether the output is written using asynchronous writes (not supported on all GPUs) or via shared memory.

IDG (Image-Domain Gridding) is an algorithm for radio astronomical imaging, of which the *gridder* and *degridder* kernels are the most compute intensive. IDG moves the computation (which resembles convolution) from the *frequency domain* to the *image domain* by introducing *subgrids* and Fourier transformations for processing input data in smaller subsets [238, 239]. The GPU implementation of the gridder has the following tuning parameters: `BLOCK_SIZE_X`, the number of threads in a thread block; `UNROLL_PIXELS`, the number of pixels to process by a thread; `NUM_BLOCKS`, the number of threads blocks per SM; `USE_EXTRAPOLATE`, option to reduce the number of trigonometric operations, at the cost of having to perform more fused multiply-add operations. The degridder kernel has the same options, except for `UNROLL_PIXELS`.

Dedispersion is used in time-domain astronomy to detect transient effects (e.g. fast radio bursts) and pulsars. The signal received by the telescope is dispersed (shifted) in time of the frequency band, and dedispersion is needed to correct for this. Dedispersion can either be performed in the *time domain* (TDD), or in the *Fourier domain* (FDD) [12]. TDD has two tuning parameters: `SAMPS_PER_THREAD`, controls the number of samples to be processed per thread; `USE_TEXTURE_MEM`, whether to use texture memory as a cache when loading input data. FDD has the following tuning parameters: `NFREQ_BATCH_GRID` and `NDM_BATCH_GRID` control the number of input samples to process per kernel invocation; `NCHAN_BATCH_THREAD`, the number of input samples (in the frequency dimension) that every GPU thread processes; `USE_SHARED_MEMORY`, use shared memory as software-managed cache when reading input data; `USE_EXTRAPOLATE`, reduces the number of trigonometric operations (same as for IDG, see above.).

6.5 Experimental results

6.5.1 Impact of energy tuning versus race-to-idle

In this section, we experimentally answer whether auto-tuning for energy efficiency (global energy-to-solution) is different from auto-tuning for the lowest kernel runtime across all clock frequencies (race-to-idle). Furthermore, we report the lowest energy configuration at max clocks. We compare with a practical compromise where we first tune for time, and then select a clock frequency for the best energy efficiency. We call this last approach *race-to-idle+clocks*. Conversely, we also consider *energy-to-solution+clocks* where we fix the frequency at the base clock frequency, tune for

energy, and then select a clock frequency to further maximize energy efficiency.

In Figure 6.3, we show the lowest energy configuration in the GEMM search space with each of the aforementioned methods across several GPUs. For the TITAN RTX we used the PowerSensor2 measurements to validate the findings. We use relatively widely spaced equidistant samples from the range of supported SM clock frequencies (7-points) due to the high cost of obtaining all measurements (9 days per GPU).

First, Figure 6.3 shows that the fastest configuration returned by race-to-idle is not the most energy efficient for any of the GPUs. Second, for most GPUs, the energy usage of the configurations found by race-to-idle+clocks and energy-to-solution+clocks are close to the global lowest energy configuration, but they never have the same parameters. Note that for race-to-idle+clocks, we first tuned for time with the clock frequency fixed to the maximum, before tuning only the clock frequency for energy efficiency.

The exception is the Tesla A100, where we see a gap in energy usage between all five methods. This means that there is a particular combination of tunable parameter values that results in a configuration that is more energy-efficient than anything returned by the two-step optimization approaches. In other words, to find the global optimum in terms of energy-to-solution it is necessary to search the combined configuration space of all tunable parameters, including clock frequencies.

Our experimental results show that auto-tuning the GEMM kernel for energy efficiency does not lead to the same optimal configuration as tuning for time, as all five methods produce different configurations, with a different energy usage. This raises the question of how kernel speed and energy efficiency are related. In Figure 6.4 we plot the compute performance in GFLOP/s for every GEMM configuration over energy efficiency in GFLOPs/W, together with the Pareto front in red. By looking at the points on the Pareto front for the RTX A4000 and Tesla A100, we see that the trade-off between speed and energy efficiency differs between GPUs. For the RTX A4000, a speed reduction of 28.4% leads to an increase in energy efficiency of just 5.8%. However, for the Tesla A100, a speed reduction of 27.5.% leads to an increase in energy efficiency of 50.9%. Therefore, the trade-off between kernel runtime and energy usage is GPU specific.

Overall, our results show that, for the GEMM kernel, tuning for lowest energy leads to different configurations than tuning for lowest execution time. However, depending on the GPU, it may be sufficient to treat the optimization as a two-stage optimization problem; first optimizing for minimal energy with a fixed clock frequency, and then optimizing for the most energy efficient frequency, can result in close to optimal energy efficiency on certain GPUs.

6.5.2 Speed vs energy: tuning difficulty of optimization spaces

Tuning a kernel for energy typically requires a larger search space compared to tuning only for execution time. For energy, the search space is typically enlarged with tunable parameters such as clock frequency, or power limit, and possibly other specific optimizations that affect energy usage (e.g. the use of shared memory).

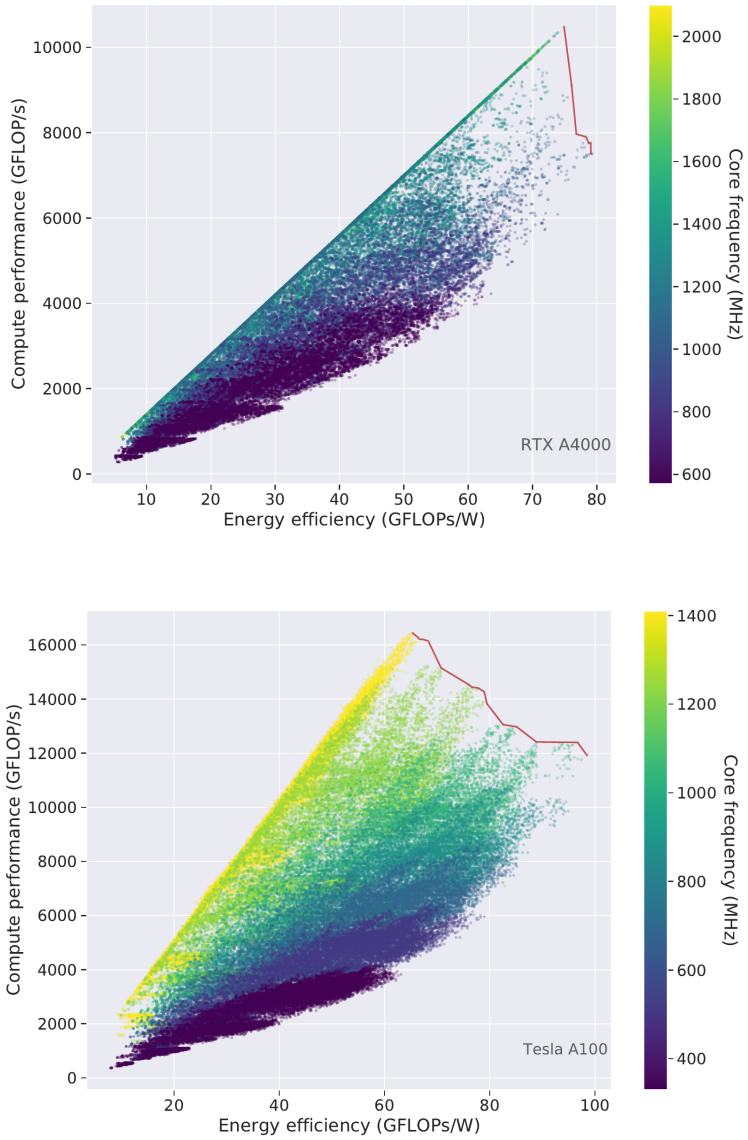


Figure 6.4: Kernel speed (GFLOP/s) over energy efficiency (GFLOPs/W) for all GEMM configurations for the RTX A4000 (top) and Tesla A100 (bottom). The red line is the Pareto front, i.e., neither performance or efficiency can be improved without decreasing the other. Points are coloured according to the core frequency.

This raises the question whether the search space for energy tuning, compared to tuning execution time, is only larger, or whether energy is actually harder to optimize with optimization algorithms.

The *proportion of PageRank centrality* [213] quantifies search difficulty for blind optimization algorithms. Here, a *fitness flow graph* (FFG) is created where all the points in the search space are represented as nodes, and a directed edge from a node to its neighbour is added if the neighbour has better fitness (energy or time). A random walk across the FFG has the property that it mimics a randomized first-improvement local search algorithm. The PageRank centrality of a local minimum in the FFG is the proportion of arrivals in that minimum for a random walk, i.e., the proportion of arrivals of a first-improvement local searcher during optimization. Since local searchers terminate in local minima, the proportion of centrality metric considers the fraction of centrality of "suitably good" local minima, among all minima in the space. In other words, it gives the expected fraction of local search terminations in "good" local minima. If near-optimal minima have high centrality, a local searcher will find a close to optimal solution in fewer evaluations. Here, "suitably good" means that the fitness of the minimum is within $p \cdot f_{optimal}$ for some $p \geq 1$.

In Figure 6.5, we plot the proportion of centrality as a function of p for GEMM, for the RTX A4000, RTX A6000, and Tesla A100 GPUs. For every GPU we plot the proportion of centrality curve for performance (time) tuning, energy tuning with clock frequency, and energy tuning with power limits. There does not appear to be a significant difference in difficulty for the RTX A4000 GPU. For the RTX A6000 GPU, the minima with more than 125% runtime of the optimum are less central. However, as these minima are already significantly worse than the near-optimal solutions, we conclude that performance tuning is not significantly harder than energy tuning for the RTX A6000. For the Tesla A100, we find that energy tuning is significantly harder than performance tuning. For minima $\leq 110\%$ of optimal fitness, a local search algorithm is 2-4$\times$ less likely to terminate in these minima when minimizing energy.

Overall, in our experiments, energy tuning is either similar in tuning difficulty or harder depending on the GPU. As such, these search spaces remain infeasibly large to traverse fully within a day, and picking many sampling clock frequencies or power limits will compound this problem.

6.5.3 Power capping versus frequency tuning

In this section, we compare two methods that frequently appear in the literature; power capping [117], which is fixing the power limit of the GPU, and frequency tuning [4, 29, 58, 64, 147, 191], which aims to find the optimal application-specific GPU clock frequency.

In Figure 6.6, we analyse the impact of both frequency tuning and power capping on GPU power consumption. At the same measured frequencies, power consumption seems a bit higher when using a fixed clock frequency compared to setting a power limit. We observe that power capping does not cover the entire

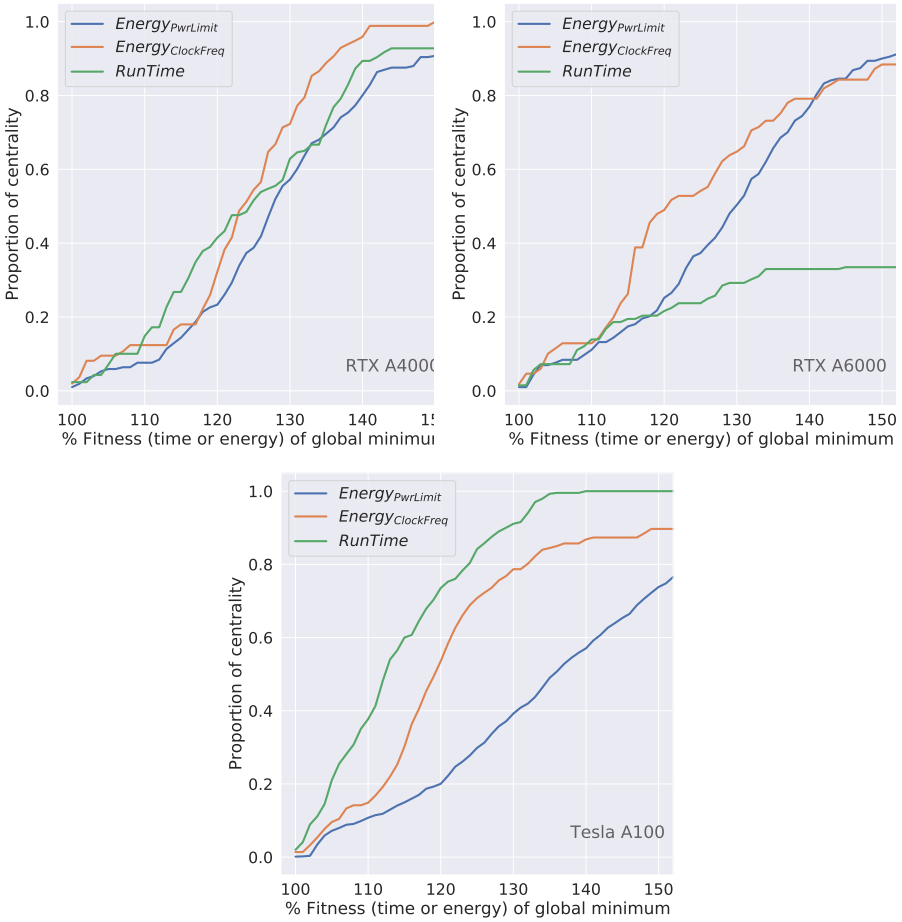


Figure 6.5: Proportion of centrality for tuning execution time, energy tuning (power limit), and energy tuning (clock frequency) for the RTX A4000, RTX A6000, and Tesla A100 GPUs.

range of clock frequencies supported by the GPU. Therefore, using frequency tuning, we can reduce the power consumption below the minimum power limit, which may be beneficial for some applications. Moreover, by operating at a fixed clock frequency (below the point where throttling may occur), GPU behaviour is more predictable.

To compare the two methods globally, we add to the existing tunable GEMM parameters either a set of power limits or clock frequencies. We take a 7-point equidistant sample from the range of power limits in case of power capping, and the range of supported SM clock frequencies in case of frequency tuning. Using these parameters, we have performed a full combined search space exploration of the GEMM application on the RTX A4000, RTX A6000, Tesla A100 and TITAN RTX GPUs. On the Titan RTX, we measured power consumption using PowerSensor2 instead of NVML.

The lowest measured energy for power capping and frequency tuning is given in Figure 6.7. For the RTX A4000 and A6000 GPUs, power capping results in a marginally lower energy configuration, but not for the Tesla A100. For the TITAN RTX, where we used 20 sampling points for frequency tuning (300 MHz to 2100 MHz in steps of 75 MHz) and 9 for power capping (100 W to 300 W in steps of 25 W), we see that frequency tuning finds a significantly more energy efficient configuration. This seems to suggest that given sufficient sampling points, due to the increased frequency range, frequency tuning can result in a more energy efficient configuration. However, this leads to an increase in search points in an already large search space. To combat this, in Section 6.5.4, we investigate the relationship between frequency and voltage, and how this can be used to steer fine-grained frequency tuning.

6.5.4 Model-steered frequency tuning

In this section, we analyse the impact of clock frequency scaling on the power consumption of the GPU, with the goal of identifying a range of suitable clock frequencies that likely results in energy-efficient configurations. The GPU core voltage can be queried by calling `NVIDIA-smi -q -d VOLTAGE`. In our experience, this option is only available with fairly recent NVIDIA drivers (510 and newer) in combination with Ampere GPUs (e.g. A100, A4000, A6000).

We plot the frequency-voltage curves for Tesla A100 and RTX A4000 in Figure 6.8. We observe that there is indeed a non-linear relation between core frequency and voltage, as discussed in Section 6.3.1. For both the Tesla A100 and RTX A4000, the voltage remains unchanged for a range of core frequencies, after which the voltage increases seemingly quadratically. We will refer to the point where this increase occurs as the *ridge point*. The RTX A4000 seems to be capped at 1875 MHz, as the core voltage does not increase beyond this point. This is likely due to its power limit of 140W. This is not observed for the Tesla A100, potentially due to its lower maximum operating frequency and higher power limit of 250W. At the ridge points, the clock frequency for the GPUs is 72% and 70% of the peak clock frequency, for the Tesla A100 and RTX A4000 respectively. Interestingly, for both

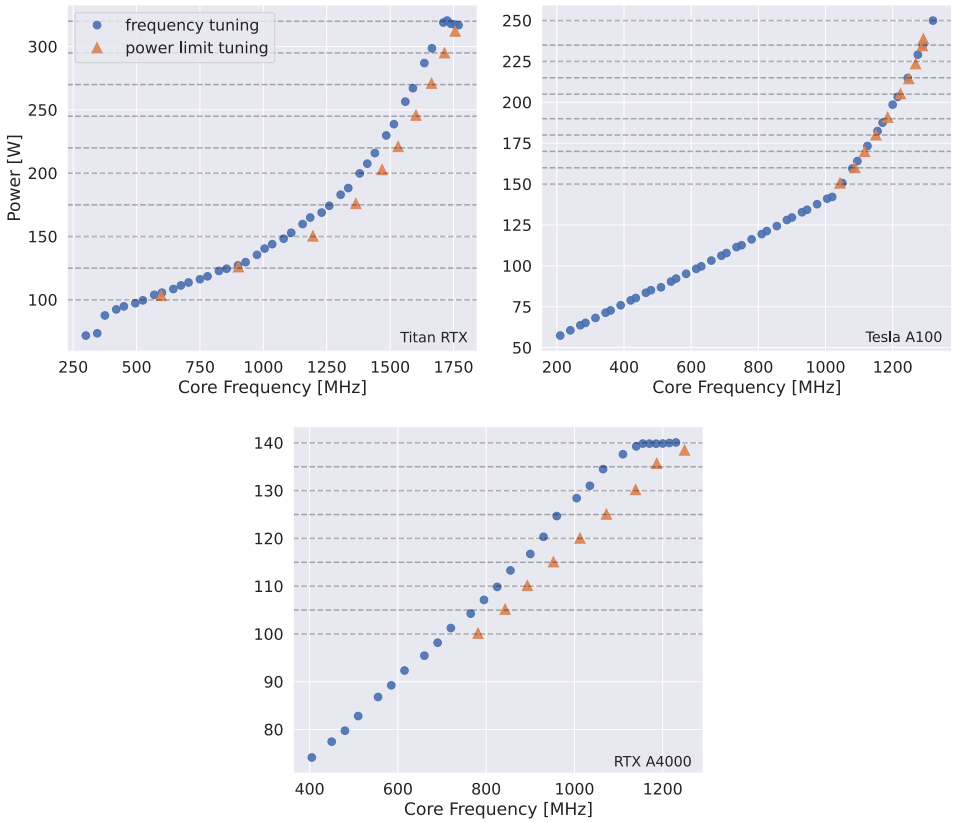


Figure 6.6: Tuning using a power limit (triangles) versus tuning using frequency (circles) for TITAN RTX (top left), Tesla A100 (top right) and RTX A4000 (bottom) for a synthetic workload that fully occupies the GPU. For all three GPUs, the power consumption coincides with the configured power limit (indicated with the dashed lines). Moreover, we observe that for this workload, the TITAN RTX and RTX A4000 can not sustain their maximum advertised turbo clock frequency of 1770 MHz and 1560 MHz, respectively.

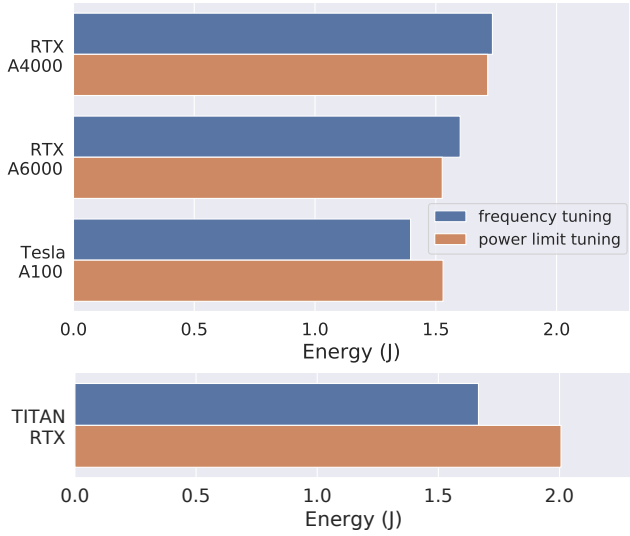


Figure 6.7: Lowest found energy for power capping or frequency tuning for GEMM, for the RTX A4000, RTX A6000, Tesla A100, and TITAN RTX GPUs. The energy measurements for the TITAN RTX were acquired using the PowerSensor2 instead of the NVML energy.

GPUs, the ridge point does not coincide with the base frequency.

Estimating GPU power consumption

Equation 6.1 shows that the power consumption of a GPU can be modelled as the sum of the idle power and the dynamic power. In our model we take the idle power consumption as a constant, and the dynamic power consumption has a linear dependence on frequency, and a quadratic dependence on voltage. Moreover, for GPUs that are prone to power-limit throttling (e.g. RTX A4000), the power consumption of the GPU is capped. The model for estimated GPU power consumption is

$$P^*_{load} = min(P_{max}, P^*_{idle} + \alpha * f * v^2). \tag{6.2}$$

P^*_{load}, P_{max}, and P^*_{idle} denote the estimated, maximum and idle power consumption of a GPU respectively. An initial value for P_{max} can be obtained by measuring the maximum power consumption observed when executing a kernel that fully loads the GPU, or simply by looking up the TDP of the device. P_{idle} can be obtained by measuring the power consumption when no kernel is being executed. α is a constant, f is the core frequency of the GPU, and v denotes the GPU core voltage.

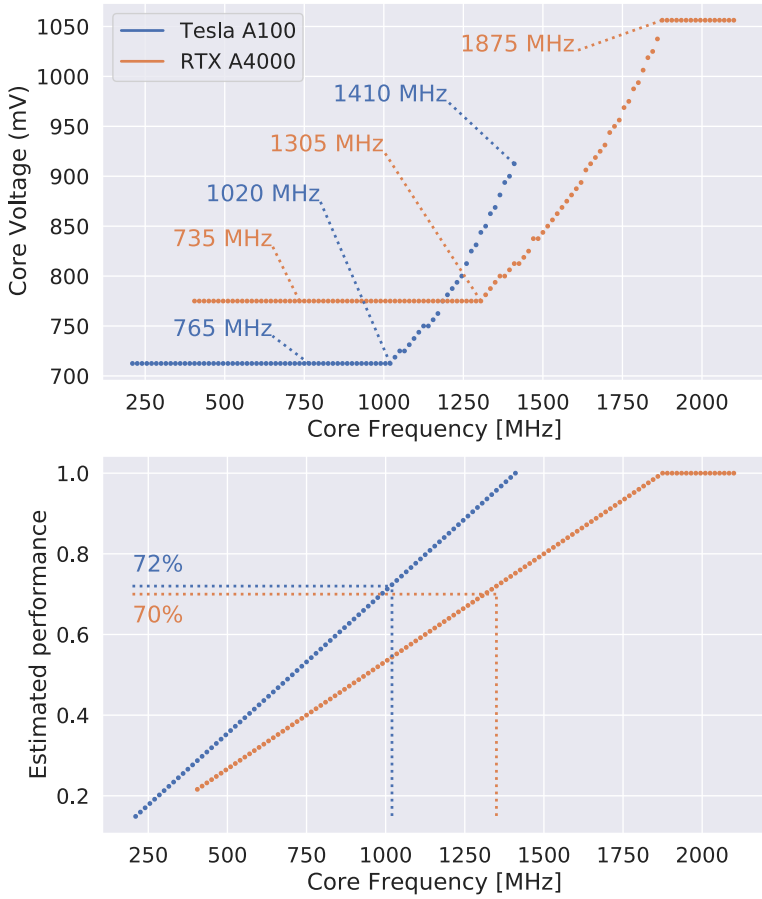


Figure 6.8: Top: GPU core frequency versus voltage curves for Tesla A100 and RTX A4000. The base clock frequency, the ridge point and peak frequency for each GPU are highlighted with a dashed line and label. Bottom: estimated performance under the assumption that GPU performance scales linearly with the clock frequency up to the point where throttling (if any) occurs. Estimated performance is normalized according to the performance for the highest possible clock frequency.

Estimating GPU core voltage

For GPUs that do not support voltage readings, such as the Tesla V100 and Titan RTX, we extend the methodology outlined above to include a voltage estimate as a function of core frequency. We assume based on our observations that for these GPUs there exists a threshold τ_{ft} after which the voltage increases with a rate β. As input, our method requires a number of power measurements for a uniform sample of all the clock frequencies that the GPU supports. These data points are used to fit equation 6.2 to estimate P_{load}, where v is substituted by:

$$v(f) = \begin{cases} 1 & f < \tau_{ft} \\ \beta * (f - \tau_{ft}) & f >= \tau_{ft} \end{cases} \tag{6.3}$$

Fitting the model

We test our model by configuring Kernel Tuner to record core frequency and power usage while running a simple synthetic kernel (array dot product) that fully loads the GPU. We only need a few samples, spaced uniformly along the supported core frequencies. Using the measurements obtained with Kernel Tuner, for every GPU, we fit equation 6.2 to the data as outlined in section 6.3.1. When fitting the model for P^*_{load}, the frequency f runs till the highest clock frequency before throttling (if any) occurs.

The left plot in Figure 6.9 illustrates that the estimated power consumption closely follows the power consumption measured using NVML. Next, the estimated power consumption is used to compute estimated energy usage as a function of absolute power (P^*_{load}) divided by clock frequency (f). For each of the GPUs, there is a core frequency that minimizes estimated energy usage, see Figure 6.9 (right). For both the Tesla A100 and RTX A4000, the predicted most energy-efficient clock frequencies (985 MHz and 1298 MHz) are close to the observed ridge points at 1025 MHz and 1290 MHz as identified in Figure 6.8.

Reducing the clock frequency beyond the ridge point does not make the GPU more energy efficient, as performance drops with f while v is constant below the ridge point. This leads to a higher total energy usage for non-zero P_{idle}. On the other hand, there is a trade-off between performance and energy when considering higher clock frequencies than the ridge point, up to the point where throttling starts to occur (at about 1700 MHz for the RTX A4000 and 2000 MHz for Titan RTX). As energy increases quadratically with voltage, and compute performance linearly with frequency, it is unnecessary to consider frequencies significantly higher than the ridge point.

To conclude, prior to energy tuning a particular GPU kernel, we recommend running a kernel that fully loads the GPU for a range of clock frequencies. Our model can then be used to fit a power consumption curve and find an estimate for the most energy-efficient frequency. Next, energy tuning can be run with a fine-grained sampling of clock frequencies around the estimated optimal frequency.

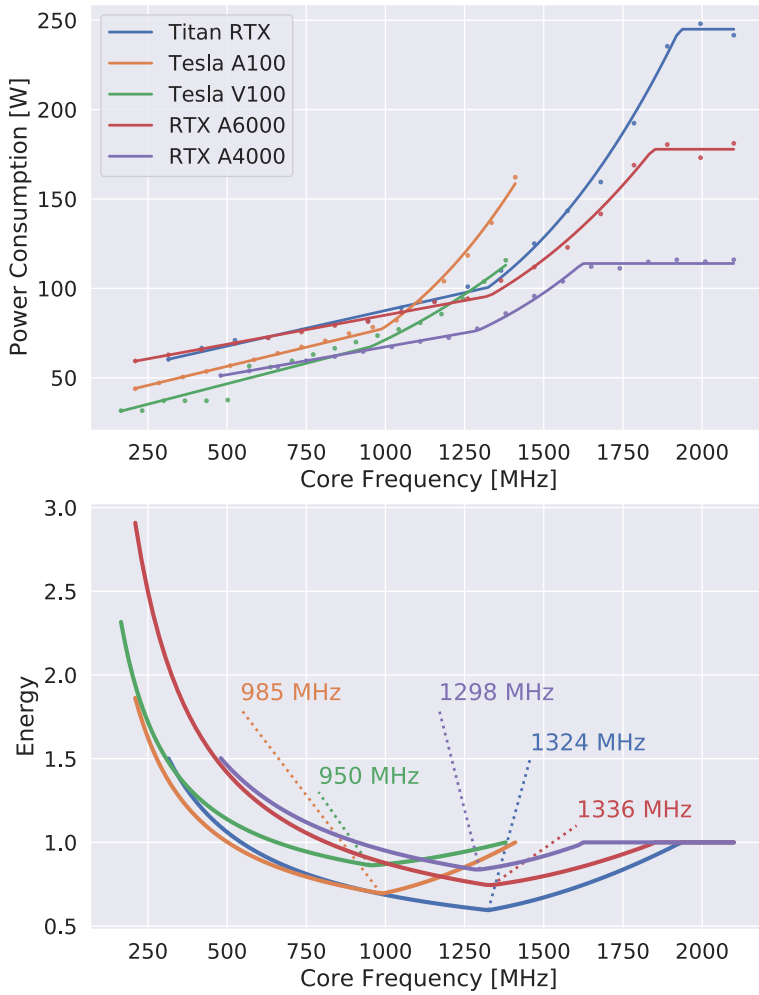


Figure 6.9: Top: Power consumption of dot product kernel that fully loads the GPU, for the Tesla A100, RTX A4000, RTX A6000, Tesla V100, and Titan RTX. The dots indicate measurements, while the lines show the modelled power consumption (equation 6.2). Bottom: Corresponding estimated energy usage, with frequency that leads to minimal energy usage.

GPU	Kernel	GOPs /W before	GOPs /W after	GOPs /W gained	TOP /s before	TOP /s after	TOP /s gained	Frequency (MHz)
Tesla A100	Gridder	64.7	102.6	58.6%	16.3	12.0	-26.5%	1035
	Degridder	59.8	97.5	63.1%	14.5	10.7	-26.2%	1035
	FD	62.2	92.8	49.1%	9.7	7.3	-24.6%	1035
	Dedispersion TD	13.3	21.5	61.3%	3.4	2.5	-26.4 %	1035
	Dedispersion Tensor-Core	684.8	1264.2	84.6%	148.4	135.2	-8.9%	1035
	Correlator							
	LOFAR Correlator	58.9	125.8	113.8%	12.2	10.7	-12.0%	1035
RTX A4000	Gridder	77.6	107.5	38.6%	11.0	8.1	-25.8%	1200
	Degridder	90.8	131.6	44.9%	10.2	9.4	-8.1%	1470
	FD	77.6	111.9	44.3%	8.3	6.7	-19.2%	1290
	Dedispersion TD	12.9	17.2	33.0%	1.5	1.1	-22.2%	1200
	Dedispersion Tensor-Core	571.2	606.8	6.2%	57.2	55.2	-3.6%	1290
	Correlator							
	LOFAR Correlator	98.9	119.3	20.6%	8.7	8.4	-4.2%	1470
TITAN RTX	Gridder	55.2	68.6	24.2%	14.3	9.0	-37.2%	1260
	Degridder	48.4	65.6	35.4%	13.7	8.2	-39.7%	1155
	FD	39.9	59.9	50.2%	10.2	5.5	-45.4%	1050
	Dedispersion TD	8.0	12.1	50.7%	2.1	1.3	-40.0%	1050
	Dedispersion Tensor-Core	140.5	209.5	49.1%	34.7	23.4	-32.6%	1155
	Correlator							
	LOFAR	51.5	78.0	51.6%	12.8	7.2	-43.4%	1155
Tesla V100	Gridder	59.6	73.6	23.6%	11.6	9.5	-18.0%	1110
	Degridder	61.7	74.2	20.2%	11.0	8.8	-19.9%	1110
	FD	58.6	69.2	18.1%	7.4	6.0	-19.2%	1110
	Dedispersion TD	11.6	15.7	34.9%	2.2	1.3	-37.8%	1110
	Dedispersion Tensor-Core	260.8	301.5	15.6%	34.2	27.7	-18.9%	1110
	Correlator							
	LOFAR Correlator	74.7	86.8	16.3%	9.9	7.6	-23.5%	1110

Table 6.2: Energy efficiency (GOPs/W) and compute performance (TOP/s) before and after model-steered frequency tuning, i.e., select the most energy-efficient frequency within ±10% MHz of the ridge points found in Figure 6.9. All kernels use floating point operations (FLOPs) except the Tensor-Core correlator, which uses 16-bit integer operations.
*__Note:__ The before measurements are already tuned for time by a domain expert.

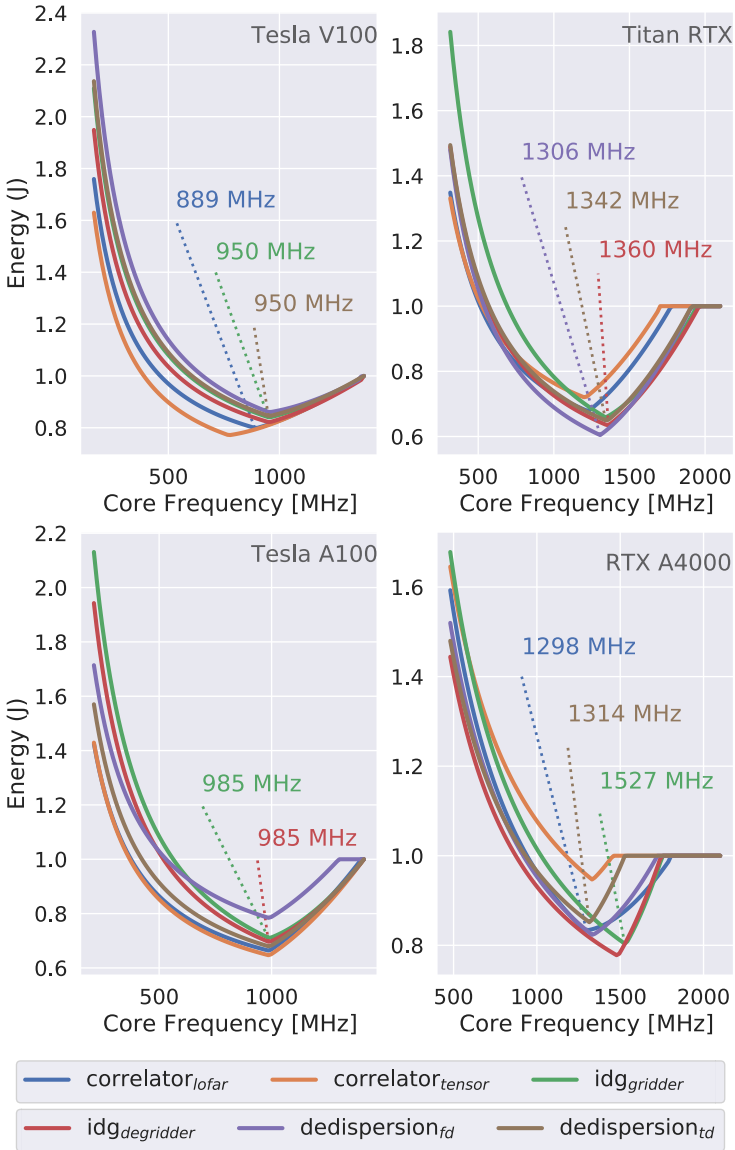


Figure 6.10: Modelled energy usage (J) with power consumption model for core clock frequencies (MHz) of LOFAR kernels for the Tesla V100, Titan RTX, Tesla A100 and RTX A4000 GPUs.

This feature is included in Kernel Tuner[2] (version 0.4.4). In this work, we use a range of ±10% of the optimal frequency estimated with the model.

6.5.5 Practical efficiency gain for radio astronomy kernels

To verify the energy gains on a real-world high-throughput pipeline, we apply our model-steered frequency tuning method to the six radio astronomy LOFAR kernels (see section 6.4) currently running on the DAS-6 system [10], and LOFAR COBALT-2 system [22] (can receive more than 1 Tbit/s). By using model-steered frequency tuning we reduce the size of the searchspaces by 82.4%, 78.9%, 77.8%, and 80.0% for the Tesla A100, RTX A4000, Titan RTX, and Tesla V100 respectively. The measured compute performance and energy efficiency before and after model-steered tuning is given in Table 6.2. Note that all six kernels have previously been optimized for compute performance, which means that the reduction in compute performance may be more severe than in most cases.

After model-steered frequency tuning, the LOFAR kernels gained between ∼15–113% in energy efficiency, while losing ∼3–45% compute performance. Gains in energy efficiency, and losses in compute performance, varied significantly between GPU models and kernels. Two notable outliers are the Tensor-Core correlator on the RTX A4000, where efficiency increased only 6%, and the LOFAR correlator on the Tesla A100, where an efficiency gain of 113.8% was achieved while losing only 12% compute performance. Overall, the mean energy efficiency gain was $42.0 \pm 24.1\%$, and the mean compute performance loss was $-24.3 \pm 12.1\%$.

The estimated energy usage curves for each application using the power consumption model are given in Figure 6.10. We can see that sometimes the estimated optimal frequency is close to the measured optimal frequency in Table 6.2, and sometimes differs more significantly. In future work, we plan to expand the model by adding memory- and temperature-dependent terms.

6.6 Conclusions

We have investigated several GPU kernel tuning approaches for improving energy efficiency, and extended Kernel Tuner's capabilities for measuring GPU power consumption and for tuning energy usage. On a commonly-used benchmark matrix multiplication kernel (GEMM) – designed for compute performance without energy-specific tunable parameters – we found that with a speed reduction of 27.5% an increase in energy efficiency of 50.9% is possible on the Tesla A100. Additionally, the combined search space of all tunable parameters (including clock frequency) contains a globally lower energy configuration, compared to tuning for performance and then tuning clock frequency separately. However, for most GPUs tuning the frequency separately did lead to a close to optimal energy usage. When investigating energy tuning methods, we found that clock frequency tuning gives

[2]https://github.com/KernelTuner/kernel_tuner

more fine-grained control over GPU power consumption than power capping, and enables a larger (and lower) range of power consumption.

Due to the prohibitively large search spaces when tuning both kernel parameters and clock frequency, we introduced a model to estimate GPU power consumption. We show that a single core clock frequency is the most energy efficient when the other tunable parameters are constant. This clock frequency can easily be estimated using our power consumption model. We verified the potential energy efficiency gains when tuning around $\pm 10\%$ of our estimated frequency on a number of real-world radio astronomy kernels, and increased energy efficiency more than two fold at a loss of 12% compute performance. Overall, the mean energy efficiency gain was $42.0 \pm 24.1\%$, and the mean compute performance loss was $-24.3 \pm 12.1\%$. Using our model-steered frequency tuning approach, we were able to dramatically reduce the size of the auto-tuning search spaces by $77.8 - 82.4\%$.

7 Conclusion and outlook

The main goal of the research presented in this thesis was to develop approaches for optimizing computationally demanding imaging pipelines from several different angles. A subordinate goal was to try and support the research questions with practical problems from real-world applications and test the techniques on real data as much as possible. In this chapter, we summarize the contributions of this thesis, discuss its limitations, and suggest directions for future research.

7.1 Contributions and limitations

The contributions of Chapter 2 can be categorized as an attempt to practically realize the computationally challenging task of doing real-time reconstruction and segmentation, and how machine learning can help in this regard. Conceptually, design choices were made in the RECAST3D framework to perform quasi-3D reconstruction, and to use an omnidirectional CNN. On a software level, RECAST3D is created as a modular system with separate servers running visualization and reconstruction separately, and communication is done via data packets. This allows the system to be extended by different plugins that can asynchronously compute processing steps, e.g., filtering of the projection data can be done while earlier reconstructed slices are being annotated. On the hardware level, this modular system also has benefits since processing steps can be split among different GPUs, or even different machines to avoid blocking the pipeline. Practically, such a computational pipeline potentially represents an important step for online and real-time analysis of tomographic experiments. A limitation of the approach is related to the generalizability of the neural network to new data. For accurate segmentation, the training data needs to be representative of the data that will be segmented during the scanning procedure, meaning similar CT data needs to be available ahead of the experiment. Nevertheless, using such a pipeline, one can perform real-time

and online segmentation of quasi-3D volumes, enabling immediate feedback and analysis during experiments.

The main contribution of Chapter 3 is that the study outlines what benefits arise from connecting traditional CT approaches with modern auto-differentiation frameworks. While the convergence of a gradient descent-based optimization method is not guaranteed, the study shows that in practice the method still produces strong results. A large benefit comes from the engineering perspective, where the user can now trial different learned CT workflows and turn optimization on or off for selected parameters at will, without having to design complicated specialized algorithms. Connecting existing building blocks to an auto-differentiation framework means that users can experiment in a plug-and-play nature to solve common image processing tasks. Despite these benefits, several limitations of the approach emerged in the study. First, the current setup requires a large effort to implement existing algorithms in PyTorch, which inhibits adoption of the approach in practice. Second, the learning rate needs to be picked manually which is challenging when multiple parameters of different magnitude are involved.

The main contribution of Chapter 4 is a method that significantly prunes CNNs which does not require hyperparameters to be set. Furthermore, the graph representation allows for efficient pruning of redundant operations, which further reduces the size of the CNN. A drawback of the method thus far is that PyTorch implements pruning by masking the model parameters with binary tensors, thereby natively not producing any computational speedup. For the MS-D network, we implemented a pruned model that loads a sparse MS-D network to measure actual acceleration, but this is a time-consuming process if it has to be done for any CNN architecture.

Chapter 5 contributes practically usable methods to the GPU auto-tuning community that are implemented in the Kernel Tuner package. It further provides a ranking of black-box optimization methods for tuning GPU kernels, and the learnings from the study contributed to the methodology paper [250] that was set up by the GPU auto-tuning community to streamline research into optimization algorithms in the future. Secondly, the study contributes a difficulty quantification (and visualization) method for discrete search spaces based on graph theory. The application of this method need not be limited to GPU auto-tuning as it generalizes to any discrete search space. Computational limitations of the approach could come into play when the graph becomes too large to store in memory. Second, the entire search space needs to be traversed before the method can be applied. This means that it is currently only useful for post-optimization analysis, and not for steering optimization of unknown GPU kernels.

Finally, Chapter 6 provides a practical method to improve the energy efficiency of GPUs. With growing environmental concerns, methods that focus not only on maximizing compute performance but also take into consideration the environmental footprint of computations are of large importance. The simplicity of the method, i.e. a handful of executions of a small kernel that is shipped with the script, means that it can easily be applied on many different systems. Steering the GPU towards energy efficiency proved potentially highly effective as for the Tesla A100 GPU we

were able to increase its energy efficiency by 113.8% while losing only 12.0% in speed. This highlights that such a trade-off may be worthwhile on many systems if it is not crucial to extract maximal performance. A drawback of the current implementation is that it only works for NVidia server-grade GPUs. While the method was also designed to work for older models (those that do not support voltage measurements), it does not work for models where setting the core clock frequency is not allowed. It appears that this functionality has been disabled on consumer-grade GPUs, something we hope to see changed with future releases.

7.2 Outlook

Several promising research directions are unfolding to optimize computational imaging pipelines that we want to discuss in turn. A promising direction to further develop the idea of optimizing legacy pipelines with modern auto-differentiation tools is to develop infrastructure that approximates the gradients instead of computing them analytically. With this approach, the traditional building blocks are registered in the auto-differentiation framework by e.g. wrapping them in a Python class. Instead of computing gradients by traversing the computational graph, which requires expensive re-implementation of the existing algorithms, we use a method to approximate the gradients such as a finite differences method. In fact, instead of wrapping the legacy code and calling it from the auto-differentiation framework, a client-server model can be designed that separates the environments. In this manner, the auto-differentiation framework can work with external C or Fortran code or even an executable. Users would need to set up a lightweight server that can run on the legacy side and run the legacy code for a set of parameters (by for example passing them as CLI arguments, or writing them to a .txt file). Hopefully, such infrastructure would allow us to extend the learnings from Chapter 3 to a larger range of legacy pipelines, many of which cannot easily be reimplemented in PyTorch.

An alternative to approximating the gradient during pipeline optimization would be to approximate the forward model. If the forward model is prohibitively expensive to run repeatedly, it may be possible to approximate the forward model by a surrogate and optimize the parameters using the surrogate model instead. For example, in [118], a surrogate to TV reconstruction is introduced which could be used to tune the regularization parameter λ. Potentially, approaches could be developed that are suitable to approximate many computational building blocks. For example, neural networks are generic surrogate models that, if given the dimensions of the input and output space of a computational block, could be used to replace that block during pipeline optimization.

Another research direction is to investigate methods that can adaptively adjust gradient-based optimization methods for traditional algorithms. When parameters have significantly different magnitudes, it is challenging to find a combination of step sizes that makes the optimization method converge. Furthermore, approaches similar to batch normalization in deep learning could be used in between traditional

building blocks to normalize input and output data to avoid vanishing gradient problems. Next, in principle for N parameters, a single finite differences approximation requires $2N$ executions ($f(x+h)$ and $f(x-h)$). If an approximation of the gradient is used such as finite differences, approaches could be applied that limit the amount of executions necessary to run a finite differences method.

To improve the usefulness of neural network pruning, more development could be done on methods that create sparse models from the current PyTorch implementation with masked parameters. Potentially, the computational graph itself could be used to construct the pruning graph that LEAN uses, and the masked tensors could be used to remove branches of the computational graph altogether.

A major issue in auto-tuning research is the lack of proper algorithm comparisons. Most studies demonstrate their optimization algorithms on a specific dataset and only compare them to selected methods using different quality metrics. As mentioned, a proposed methodology by the auto-tuning community to improve reproducibility and transferability of results is currently in preparation [250]. Another interesting direction of research could be if methods can be developed that approximate the FFG on the fly, and use it as a heuristic to guide optimization.

As mentioned in Chapter 6, a major limitation to improving the energy efficiency of GPUs is that consumer-grade GPUs do not support core clock frequency tuning. In my view, vendors must expose functionality that can help with improving the energy efficiency of as many models as possible. Furthermore, we want to extend the power consumption model to work for other vendors. The model in principle is generically applicable, but code needs to be developed that can query the relevant measurements for non-NVidia GPUs. Another dimension of the GPU auto-tuning problem that was not explored in this thesis is the incorporation of memory energy efficiency. In addition to core clock frequency, GPUs have several memory clock frequencies which influence its performance and energy consumption. Thus far, there are usually only a few available memory clock frequencies, but if future GPUs allow for a larger range of memory clock frequencies, these would need to be incorporated into the power consumption model for further gains.

Bibliography

[1] 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". *Optics Express* 24.22 (2016), pp. 25129–25147 (cit. on pp. 5, 22, 25, 38).

[2] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. Software available from tensorflow.org. 2015 (cit. on p. 8).

[3] S. Abid, F. Fnaiech, and M. Najim. "A new Neural Network pruning method based on the singular value decomposition and the weight initialisation". In: *2002 11th European Signal Processing Conference*. 2002, pp. 1–4 (cit. on p. 61).

[4] S. Akiki, Z. Yang, C. Liu, J. Tang, and S. Liu. "Energy-Aware Automatic Tuning of Many-Core Platform via Gradient Descent". In: *2018 IEEE SmartWorld, Ubiquitous Intelligence & Computing, Advanced & Trusted Computing, Scalable Computing & Communications, Cloud & Big Data Computing, Internet of People and Smart City Innovation*. 2018 (cit. on pp. 107, 117).

[5] J. Ansel, S. Kamil, K. Veeramachaneni, J. Ragan-Kelley, J. Bosboom, U.-M. O'Reilly, and S. Amarasinghe. "OpenTuner: An extensible framework for program autotuning". In: *2014 23rd International Conference on Parallel Architecture and Compilation Techniques (PACT)*. 2014, pp. 303–315 (cit. on pp. 75, 106).

[6] H. Anzt, B. Haugen, J. Kurzak, P. Luszczek, and J. Dongarra. "Experiences in autotuning matrix multiplication for energy minimization on GPUs". *Concurrency and Computation: Practice and Experience* 27.17 (2015) (cit. on pp. 107, 108).

[7] A. H. Ashouri, W. Killian, J. Cavazos, G. Palermo, and C. Silvano. "A Survey on Compiler Autotuning Using Machine Learning". *ACM Computing Surveys* 51.5 (2018) (cit. on pp. 75, 106).

[8] I. P. Astono, J. S. Welsh, S. Chalup, and P. Greer. "Optimisation of 2D U-Net Model Components for Automatic Prostate Segmentation on MRI". *Applied Sciences* 10.7 (2020) (cit. on p. 68).

[9] V. Badrinarayanan, A. Kendall, and R. Cipolla. "Segnet: A deep convolutional encoder-decoder architecture for image segmentation". *IEEE Transactions on Pattern Analysis and Machine Intelligence* 39.12 (2017), pp. 2481–2495 (cit. on pp. 22, 166).

[10] H. Bal, D. Epema, C. de Laat, R. van Nieuwpoort, J. Romein, F. Seinstra, C. Snoek, and H. Wijshoff. "A Medium-Scale Distributed System for Computer Science Research: Infrastructure for the Long Term". *IEEE Computer* 49.5 (May 2016), pp. 54–63 (cit. on pp. 111, 127).

[11] J. Ballé, V. Laparra, and E. P. Simoncelli. "End-to-end optimized image compression". In: *International Conference on Learning Representations (ICLR)*. 2017 (cit. on p. 8).

[12] Bassa, C. G., Romein, J. W., Veenboer, B., van der Vlugt, S., and Wijnholds, S. J. "Fourier-domain dedispersion". *Astronomy & Astrophysics* 657 (2022), A46 (cit. on p. 114).

[13] S. Basu and Y. Bresler. "$O(N^3 \log N)$ backprojection algorithm for the 3-D Radon transform". *IEEE Transactions on Medical Imaging* 21 (2002), pp. 76–88 (cit. on p. 19).

[14] A. Beck and M. Teboulle. "Fast Gradient-Based Algorithms for Constrained Total Variation Image Denoising and Deblurring Problems". *IEEE Transactions on Image Processing* 18.11 (2009), pp. 2419–2434 (cit. on pp. 4, 20, 21).

[15] S. Bhadra, V. A. Kelkar, F. J. Brooks, and M. A. Anastasio. "On hallucinations in tomographic image reconstruction". *IEEE Transactions on Medical Imaging* 40.11 (2021), pp. 3249–3260 (cit. on p. 33).

[16] T. Bicer, D. Gürsoy, R. Kettimuthu, F. D. Carlo, G. Agrawal, and I. T. Foster. "Rapid Tomographic Image Reconstruction via Large-Scale Parallelization". In: *European Conference on Parallel Processing*. 2015 (cit. on p. 19).

[17] C. M. Bishop and N. M. Nasrabadi. *Pattern recognition and machine learning*. Vol. 4. 4. Springer, 2006 (cit. on p. 6).

[18] D. Blalock, J. J. G. Ortiz, J. Frankle, and J. Guttag. "What is the state of neural network pruning?" *Proceedings of Machine Learning and Systems* (2020) (cit. on pp. 59, 61, 63).

[19] R. Boţ and C. Hendrich. "A Douglas–Rachford type primal-dual method for solving inclusions with mixtures of composite and parallel-sum type monotone operators". *SIAM Journal on Optimization* 24 (Jan. 2013), pp. 2541–2565 (cit. on p. 28).

[20] J. Bradbury, R. Frostig, P. Hawkins, M. J. Johnson, C. Leary, D. Maclaurin, G. Necula, A. Paszke, J. VanderPlas, S. Wanderman-Milne, and Q. Zhang. *JAX: composable transformations of Python+NumPy programs*. Version 0.3.13. 2018 (cit. on p. 8).

[21] S. Brin and L. Page. "The anatomy of a large-scale hypertextual web search engine". *Computer networks and ISDN systems* 30.1-7 (1998), pp. 107–117 (cit. on pp. 17, 75, 100).

[22] P. C. Broekema, J. J. D. Mol, R. Nijboer, A. van Amesfoort, M. Brentjens, G. M. Loose, W. Klijn, and J. Romein. "Cobalt: A GPU-based correlator and beamformer for LOFAR". *Astronomy and Computing* 23 (2018), pp. 180–192 (cit. on pp. 112, 127).

[23] R. A. Brooks and G. Di Chiro. "Beam hardening in X-ray reconstructive tomography". *Physics in medicine and biology* 21.3 (1976), p. 390 (cit. on p. 47).

[24] G. J. Brostow, J. Fauqueur, and R. Cipolla. "Semantic object classes in video: A high-definition ground truth database". *Pattern Recognition Letters* 30.2 (2009), pp. 88–97 (cit. on pp. 68, 166).

[25] G. J. Brostow, J. Shotton, J. Fauqueur, and R. Cipolla. "Segmentation and recognition using structure from motion point clouds". In: *European Conference on Computer Vision (ECCV)*. Springer. 2008, pp. 44–57 (cit. on pp. 68, 166).

[26] M. Burtscher, I. Zecena, and Z. Zong. "Measuring GPU power with the K20 built-in sensor". In: *Proceedings of Workshop on General Purpose Processing Using GPUs*. 2014 (cit. on p. 110).

[27] J.-W. Buurlage, H. Kohr, W. J. Palenstijn, and K. J. Batenburg. "Real-time quasi-3D tomographic reconstruction". *Measurement Science and Technology* 29.6 (2018), p. 064005 (cit. on pp. 19, 22).

[28] R. H. Byrd, P. Lu, J. Nocedal, and C. Zhu. "A limited memory algorithm for bound constrained optimization". *SIAM Journal on Scientific Computing* 16.5 (1995), pp. 1190–1208 (cit. on p. 82).

[29] E. Calore, S. F. Schifano, and R. Tripiccione. "Energy-performance tradeoffs for HPC applications on low power processors". In: *European Conference on Parallel Processing*. 2015 (cit. on pp. 107, 117).

[30] K. W. Cameron. "Energy Oddities, Part 2: Why Green Computing Is Odd". *IEEE Computer* 46.3 (2013) (cit. on p. 107).

[31] J.-E. Campagne, F. Lanusse, J. Zuntz, A. Boucaud, S. Casas, M. Karamanis, D. Kirkby, D. Lanzieri, Y. Li, and A. Peel. "JAX-COSMO: An End-to-End Differentiable and GPU Accelerated Cosmology Library". *arXiv preprint arXiv:2302.05163* (2023) (cit. on p. 35).

[32] S. Chakraborty, N. K. Nagwani, and L. Dey. "Performance Comparison of Incremental K-means and Incremental DBSCAN Algorithms". *International Journal of Computer Applications* 27.11 (2011), pp. 14–18 (cit. on p. 20).

[33] A. Chaparala, C. Novoa, and A. Qasem. "Autotuning GPU-Accelerated QAP Solvers for Power and Performance". In: *IEEE International Conference on High Performance Computing and Communications, International Symposium on Cyberspace Safety and Security, and International Conference on Embedded Software and Systems*. New York, NY, 2015 (cit. on p. 107).

[34] J. Chen, L. Wu, J. Zhang, L. Zhang, D. Gong, Y. Zhao, Q. Chen, S. Huang, M. Yang, X. Yang, et al. "Deep learning-based model for detecting 2019 novel coronavirus pneumonia on high-resolution computed tomography". *Scientific reports* 10.1 (2020), pp. 1–11 (cit. on p. 33).

[35] T. Chen, B. Xu, C. Zhang, and C. Guestrin. "Training deep nets with sublinear memory cost". *arXiv preprint arXiv:1604.06174* (2016) (cit. on p. 36).

[36] S. Cheng, M. Kim, L. Song, Z. Wu, S. Wang, and N. Hovakimyan. "DiffTune: Auto-Tuning through Auto-Differentiation". *arXiv preprint arXiv:2209.10021* (2022) (cit. on p. 34).

[37] J. W. Choi, D. Bedard, R. Fowler, and R. Vuduc. "A roofline model of energy". In: *IEEE 27th International Symposium on Parallel and Distributed Processing*. 2013 (cit. on p. 108).

[38] S. B. Coban and F. Lucka. *Dynamic 3D X-ray micro-CT data of a tablet dissolution in a water-based gel*. Oct. 2019 (cit. on pp. 28, 68, 167).

[39] S. B. Coban, F. Lucka, W. J. Palenstijn, D. van Loo, and K. J. Batenburg. "Explorative Imaging and Its Implementation At the Flex-Ray Laboratory". *Journal of Imaging* 6.4 (2020), p. 18 (cit. on pp. 28, 40, 68, 167).

[40] T. Connors, A. Qasem, and Q. Yi. "Modeling the Impact of Thread Configuration on Power and Performance of GPUs". *Machine Learning: Theory and Applications* (2015) (cit. on p. 107).

[41] J. Coplin and M. Burtscher. "Effects of source-code optimizations on GPU performance and energy consumption". In: *Proceedings of Workshop on General Purpose Processing Using GPUs*. 2015 (cit. on p. 107).

[42] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to algorithms*. MIT press, 2009 (cit. on p. 65).

[43] D. R. Danilak. *Why Energy Is A Big And Rapidly Growing Problem For Data Centers*. Forbes Technology Council (cit. on p. 105).

[44] K. Datta, M. Murphy, V. Volkov, S. Williams, J. Carter, L. Oliker, D. Patterson, J. Shalf, and K. Yelick. "Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures". In: *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*. 2008 (cit. on p. 107).

[45] F. De Carlo, D. Gürsoy, D. J. Ching, K. Joost Batenburg, W. Ludwig, L. Mancini, F. Marone, R. Mokso, D. M. Pelt, J. Sijbers, and M. Rivers. "TomoBank: a tomographic data repository for computational X-ray science". *Measurement Science and Technology* 29.3 (2018), p. 034004 (cit. on pp. 14, 45).

[46] E. L. Denton, W. Zaremba, J. Bruna, Y. LeCun, and R. Fergus. "Exploiting linear structure within convolutional networks for efficient evaluation". In: *Advances in Neural Information Processing Systems*. 2014, pp. 1269–1277 (cit. on p. 61).

[47] P Dewdney, W Turner, R Millenaar, R McCool, J Lazio, and T Cornwell. "SKA1 system baseline design". *Document number SKA-TEL-SKO-DD-001 Revision* (2013) (cit. on p. 105).

[48] N. Dhanachandra, K. Manglem, and Y. J. Chanu. "Image segmentation using K-means clustering algorithm and subtractive clustering algorithm". *Procedia Computer Science* 54 (2015), pp. 764–771 (cit. on p. 22).

[49] S. Dittmer, E. J., and P. Maass. "Singular Values for ReLU Layers". *IEEE Transactions on Neural Networks and Learning Systems* (2019), pp. 1–12 (cit. on p. 61).

[50] T. Donath, F. Beckmann, and A. Schreyer. "Automated determination of the center of rotation in tomography data". *Journal of the Optical Society of America A* 23.5 (2006), pp. 1048–1057 (cit. on p. 40).

[51] T. Dong, V. Dobrev, T. Kolev, R. Rieben, S. Tomov, and J. Dongarra. "A step towards energy efficient computing: Redesigning a hydrodynamic application on CPU-GPU". In: *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 2014 (cit. on p. 106).

[52] X. Dong, L. Liu, K. Musial, and B. Gabrys. "NATS-Bench: Benchmarking NAS Algorithms for Architecture Topology and Size". *IEEE Transactions on Pattern Analysis and Machine Intelligence* 44.7 (2022), pp. 3634–3646 (cit. on p. 87).

[53] X. Dong and Y. Yang. "Network pruning via transformable architecture search". In: *Advances in Neural Information Processing Systems*. 2019, pp. 760–771 (cit. on p. 61).

[54] M. Du, S. Kandel, J. Deng, X. Huang, A. Demortiere, T. T. Nguyen, R. Tucoulou, V. De Andrade, Q. Jin, and C. Jacobsen. "Adorym: A multi-platform generic X-ray image reconstruction framework based on automatic differentiation". *Optics Express* 29.7 (2021), pp. 10000–10035 (cit. on p. 35).

[55] M. Du, Y. S. G. Nashed, S. Kandel, D. Gürsoy, and C. Jacobsen. "Three dimensions, two microscopes, one code: Automatic differentiation for X-ray nanotomography beyond the depth of focus limit". *Science Advances* 6.13 (2020) (cit. on p. 35).

[56] D. Eigen and R. Fergus. "Predicting depth, surface normals and semantic labels with a common multi-scale convolutional architecture". In: *IEEE International Conference on Computer Vision (ICCV)*. 2015, pp. 2650–2658 (cit. on p. 166).

[57] M. Endrizzi. "X-ray phase-contrast imaging". *Nuclear instruments and methods in physics research section A: Accelerators, spectrometers, detectors and associated equipment* 878 (2018), pp. 88–98 (cit. on p. 42).

[58] K. Fan, B. Cosenza, and B. Juurlink. "Accurate Energy and Performance Prediction for Frequency-Scaled GPU Kernels". *Computation* 8.2 (2020) (cit. on pp. 107, 117).

[59] L. A. Feldkamp, L. C. Davis, and J. W. Kress. "Practical cone-beam algorithm". *Journal of the Optical Society of America A* 1.6 (1984), pp. 612–619 (cit. on pp. 4, 20, 37).

[60] J. Filipovič, F. Petrovič, and S. Benkner. "Autotuning of OpenCL kernels with global optimizations". In: *Proceedings of the 1st workshop on autotuning and adaptivity approaches for energy efficient HPC systems*. 2017 (cit. on pp. 76, 106).

[61] J. Filipovič, J. Hozzová, A. Nezarat, J. Ol'ha, and F. Petrovič. "Using hardware performance counters to speed up autotuning convergence on GPUs". *Journal of Parallel and Distributed Computing* 160 (2022), pp. 16–35 (cit. on pp. 76, 106).

[62] D. Freedman, R. Pisani, and R. Purves. *Statistics*. 3rd ed. W.W. Norton, 1998 (cit. on p. 88).

[63] M. Frigo and S. G. Johnson. "The Design and Implementation of FFTW3". *Proceedings of the IEEE* 93.2 (2005), pp. 216–231 (cit. on p. 75).

[64] R. Ge, R. Vogt, J. Majumder, A. Alam, M. Burtscher, and Z. Zong. "Effects of Dynamic Voltage and Frequency Scaling on a K20 GPU". In: *International Conference on Parallel Processing (ICPP)*. 2013 (cit. on pp. 107, 117).

[65] F. Glover. "Tabu search—part I". *ORSA Journal on computing* 1.3 (1989), pp. 190–206 (cit. on p. 79).

[66] G. van Gompel, K. van Slambrouck, M. Defrise, K. J. Batenburg, J. De Mey, J. Sijbers, and J. Nuyts. "Iterative correction of beam hardening artifacts in CT". *Medical physics* 38.S1 (2011), S36–S49 (cit. on pp. 47–49).

[67] R. Goncalves, T. van Tilburg, K. Kyzirakos, et al. "A Spatial Column-store to Triangulate the Netherlands on the Fly." In: *24th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*. New York, NY, USA: ACM, 2016, 80:1–80:4 (cit. on p. 84).

[68] I. Goodfellow, Y. Bengio, A. Courville, and Y. Bengio. *Deep learning*. Vol. 1. MIT Press, 2016 (cit. on pp. 7, 62).

[69] L. Grady. "Random Walks for Image Segmentation". *IEEE Transactions on Pattern Analysis and Machine Intelligence* 28.11 (2006), pp. 1768–1783 (cit. on p. 24).

[70] S. Grauer-Gray, L. Xu, R. Searles, S. Ayalasomayajula, and J. Cavazos. "Auto-tuning a high-level language targeted to GPU codes". In: *Innovative Parallel Computing (InPar)*. 2012, pp. 1–10 (cit. on p. 75).

[71] D. Grewe and A. Lokhmotov. "Automatically Generating and Tuning GPU Code for Sparse Matrix-Vector Multiplication from a High-Level Representation". In: *Proceedings of Workshop on General Purpose Processing Using GPUs*. GPGPU-4. 2011 (cit. on pp. 11, 75, 106).

[72] A. Griewank and A. Walther. *Evaluating derivatives: principles and techniques of algorithmic differentiation*. SIAM, 2008 (cit. on p. 34).

[73] F. C. Groen, I. T. Young, and G. Ligthart. "A comparison of different focus functions for use in autofocus algorithms". *Cytometry: The Journal of the International Society for Analytical Cytology* 6.2 (1985), pp. 81–91 (cit. on p. 40).

[74] J. Guerreiro, A. Ilic, N. Roma, and P. Tomás. "Multi-kernel auto-tuning on GPUs: Performance and energy-aware optimization". In: *23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*. 2015 (cit. on p. 107).

[75] Y. Guo, A. Yao, and Y. Chen. "Dynamic Network Surgery for Efficient DNNs". In: *Advances in Neural Information Processing Systems*. 2016, 1387–1395 (cit. on p. 72).

[76] D. Gürsoy, F. De Carlo, X. Xiao, and C. Jacobsen. "TomoPy: a framework for the analysis of synchrotron tomographic data". *Journal of synchrotron radiation* 21.5 (2014), pp. 1188–1193 (cit. on p. 40).

[77] F. Guzzi, A. Gianoncelli, F. Billè, S. Carrato, and G. Kourousias. "Automatic Differentiation for Inverse Problems in X-ray Imaging and Microscopy". *Life* 13.3 (2023) (cit. on p. 35).

[78] M. P. van Haarlem et al. "LOFAR: The LOw-Frequency ARray". *Astronomy & Astrophysics* 556 (2013) (cit. on pp. 11, 12, 112).

[79] D. Hackenberg, T. Ilsche, J. Schuchart, R. Schöne, W. E. Nagel, M. Simon, and Y. Georgiou. "HDEEM: High Definition Energy Efficiency Monitoring". In: *Energy Efficient Supercomputing Workshop*. 2014, pp. 1–10 (cit. on p. 107).

[80] S. Han, H. Mao, and W. J. Dally. "Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding". *International Conference on Learning Representations (ICLR)* (2016) (cit. on p. 59).

[81] S. Han, J. Pool, J. Tran, and W. Dally. "Learning both weights and connections for efficient neural network". In: *Advances in Neural Information Processing Systems*. 2015, pp. 1135–1143 (cit. on p. 62).

[82] B. Hassibi, D. G. Stork, and G. J. Wolff. "Optimal brain surgeon and general network pruning". In: *International conference on neural networks*. IEEE. 1993, pp. 293–299 (cit. on p. 60).

[83] A. B. Hayes, L. Li, D. Chavarría-Miranda, S. L. Song, and E. Z. Zhang. "Orion: A Framework for GPU Occupancy Tuning". In: *17th International Middleware Conference*. 2016 (cit. on p. 107).

[84] K. He, X. Zhang, S. Ren, and J. Sun. "Deep Residual Learning for Image Recognition". In: *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2016, pp. 770–778 (cit. on pp. 24, 64, 68).

[85] Y. He, G. Kang, X. Dong, Y. Fu, and Y. Yang. "Soft Filter Pruning for Accelerating Deep Convolutional Neural Networks". In: *27th International Joint Conference on Artificial Intelligence*. 2018 (cit. on p. 67).

[86] Y. He, P. Liu, Z. Wang, Z. Hu, and Y. Yang. "Filter Pruning via Geometric Median for Deep Convolutional Neural Networks Acceleration". In: *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2019 (cit. on pp. 61, 67).

[87] Y. He, X. Zhang, and J. Sun. "Channel pruning for accelerating very deep neural networks". In: *IEEE International Conference on Computer Vision (ICCV)*. 2017, pp. 1389–1397 (cit. on p. 60).

[88] S. Heldens, P. Hijma, B. van Werkhoven, et al. "The landscape of exascale research: a data-driven literature analysis". *ACM Computing Surveys* 53.2 (2020), pp. 1–43 (cit. on p. 73).

[89] A. A. Hendriksen. *Mixed-scale Dense Networks for PyTorch*. `https://github.com/ahendriksen/msd_pytorch`. 2020 (cit. on pp. 24, 68).

[90] A. A. Hendriksen, D. Schut, W. J. Palenstijn, N. Viganó, J. Kim, D. M. Pelt, T. van Leeuwen, and K. J. Batenburg. "Tomosipo: fast, flexible, and convenient 3D tomography for complex scanning geometries in Python". *Optics Express* 29.24 (2021), pp. 40494–40513 (cit. on p. 38).

[91] G. T. Herman. "Correction for beam hardening in computed tomography". *Physics in Medicine and Biology* 24.1 (1979), p. 81 (cit. on p. 47).

[92] S. Herrmann. "Determining the difficulty of landscapes by PageRank centrality in local optima networks". In: *Evolutionary Computation in Combinatorial Optimization*. Springer. 2016, pp. 74–87 (cit. on p. 77).

[93] S. Herrmann and F. Rothlauf. "Predicting heuristic search performance with PageRank centrality in local optima networks". In: *Annual Conference on Genetic and Evolutionary Computation*. 2015, pp. 401–408 (cit. on p. 77).

[94] P. Hijma, S. Heldens, A. Sclocco, B. van Werkhoven, and H. E. Bal. "Optimization techniques for GPU programming". *ACM Computing Surveys* 55.11 (2023), pp. 1–81 (cit. on p. 11).

[95] H. H. Holm, A. R. Brodtkorb, and M. L. Sætra. "GPU Computing with Python: Performance, Energy Efficiency and Usability". *Computation* 8.1 (2020) (cit. on p. 107).

[96] R. Horst, P. M. Pardalos, and N. Van Thoai. *Introduction to global optimization*. Springer Science & Business Media, 2000 (cit. on p. 8).

[97] K. Hou, W.-c. Feng, and S. Che. "Auto-tuning strategies for parallelizing sparse matrix-vector (spmv) multiplication on multi-and many-core processors". In: *International Symposium on Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE. 2017, pp. 713–722 (cit. on p. 76).

[98] K.-L. Hua, C.-H. Hsu, S. C. Hidayati, W.-H. Cheng, and Y.-J. Chen. "Computer-aided classification of lung nodules on computed tomography images via deep learning technique". *OncoTargets and therapy* 8 (2015) (cit. on p. 33).

[99] S. Huang, S. Xiao, and W.-c. Feng. "On the energy efficiency of graphics processing units for scientific computing". In: *IEEE 23rd International Symposium on Parallel and Distributed Processing*. 2009 (cit. on p. 107).

[100] Z. Huang, H. Liu, J. Wu, and C. Lv. "Differentiable integrated motion prediction and planning with learnable cost function for autonomous driving". *IEEE Transactions on Neural Networks and Learning Systems* (2023) (cit. on p. 34).

[101] S. Ioffe and C. Szegedy. "Batch normalization: Accelerating deep network training by reducing internal covariate shift". *International Conference on Machine Learning (ICML)* (2015) (cit. on p. 62).

[102] H. Ishibuchi and T. Murata. "A multi-objective genetic local search algorithm and its application to flowshop scheduling". *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)* 28.3 (1998), pp. 392–403 (cit. on p. 80).

[103] A. K. Jain. *Fundamentals of Digital Image Processing*. USA: Prentice-Hall, Inc., 1989 (cit. on p. 66).

[104] W. Jia, E. Garza, K. A. Shaw, and M. Martonosi. "GPU performance and power tuning using regression trees". *ACM Transactions on Architecture and Code Optimization (TACO)* 12.2 (2015) (cit. on p. 107).

[105] A. S. Jurling and J. R. Fienup. "Applications of algorithmic differentiation to phase retrieval algorithms". *Journal of the Optical Society of America A* 31.7 (2014), pp. 1348–1359 (cit. on p. 35).

[106] A. C. Kak and M. Slaney. *Principles of computerized tomographic imaging*. SIAM, 2001 (cit. on pp. 2, 3, 20, 37).

[107] S. Kamil, C. Chan, L. Oliker, J. Shalf, and S. Williams. "An auto-tuning framework for parallel multicore stencil computations". In: *IEEE International Symposium on Parallel & Distributed Processing (IPDPS)*. 2010 (cit. on pp. 74, 107).

[108] S. Kandel, S. Maddali, M. Allain, S. O. Hruszkewycz, C. Jacobsen, and Y. S. G. Nashed. "Using automatic differentiation as a general framework for ptychographic reconstruction". *Optics Express* 27.13 (2019), pp. 18653–18672 (cit. on p. 35).

[109] E. D. Karnin. "A simple procedure for pruning back-propagation trained neural networks". *IEEE Transactions on Neural Networks* 1.2 (1990), pp. 239–242 (cit. on pp. 59, 60).

[110] J. Kennedy and R. Eberhart. "Particle swarm optimization". In: *International Conference on Neural Networks*. Vol. 4. IEEE. 1995, pp. 1942–1948 (cit. on p. 82).

[111] N. Khouzami, F. Michel, P. Incardona, J. Castrillon, and I. F. Sbalzarini. "Model-based autotuning of discretization methods in numerical simulations of partial differential equations". *Journal of Computational Science* 57 (2022), p. 101489 (cit. on p. 75).

[112] D. Kim, S. Ramani, and J. A. Fessler. "Combining Ordered Subsets and Momentum for Accelerated X-Ray CT Image Reconstruction". *IEEE Transactions on Medical Imaging* 34.1 (2015), pp. 167–178 (cit. on p. 19).

[113] D. Kingma and J. Ba. "Adam: A Method for Stochastic Optimization". *International Conference on Learning Representations (ICLR)* (Dec. 2014) (cit. on pp. 7, 23, 24, 36, 68).

[114] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. "Optimization by simulated annealing". *Science* 220.4598 (1983), pp. 671–680 (cit. on p. 80).

[115] D. Kraft. *A software package for sequential quadratic programming*. Deutsche Forschungs- und Versuchsanstalt für Luft- und Raumfahrt Köln: Forschungsbericht. Wiss. Berichtswesen d. DFVLR, 1988 (cit. on p. 82).

[116] A. Krizhevsky, I. Sutskever, and G. E. Hinton. "Imagenet classification with deep convolutional neural networks". *Advances in Neural Information Processing Systems* 25 (2012) (cit. on p. 7).

[117] A. Krzywaniak and P. Czarnul. "Performance/energy aware optimization of parallel applications on GPUs under power capping". In: *International Conference on Parallel Processing and Applied Mathematics*. 2019 (cit. on pp. 107, 111, 117).

[118] M. J. Lagerwerf, W. J. Palenstijn, F. Bleichrodt, and K. J. Batenburg. "An Efficient Interpolation Approach for Exploring the Parameter Space of Regularized Tomography Algorithms". *Fundamenta Informaticae* 172.2 (2020), pp. 143–167 (cit. on pp. 55, 131).

[119] M. J. Lagerwerf, S. B. Coban, and K. J. Batenburg. *High-resolution cone-beam scan of twenty-one walnuts with two dosage levels*. Zenodo. Apr. 2020 (cit. on p. 40).

[120] A. Laugros, R. Schoonhoven, L. Pavlovic, A. Kuan, C. Bosch, A. Hendriksen, A. Diaz, M. Holler, W. C. Lee, A. Schaefer, K. J. Batenburg, P. Cloetens, N. Vigano, and A. Pacureanu. "Self-supervised image restoration in coherent X-ray neuronal microscopy". To be submitted (2024) (cit. on p. 155).

[121] Y. LeCun, Y. Bengio, and G. Hinton. "Deep learning". *Nature* 521.7553 (2015), pp. 436–444 (cit. on pp. 7, 59, 73).

[122] Y. LeCun, J. S. Denker, and S. A. Solla. "Optimal brain damage". In: *Advances in Neural Information Processing Systems*. 1990, pp. 598–605 (cit. on p. 61).

[123] T. van Leeuwen, S. Maretzke, and K. J. Batenburg. "Automatic alignment for three-dimensional tomographic reconstruction". *Inverse Problems* 34.2 (2018), p. 024004 (cit. on p. 38).

[124] C. Li and P. K.-S. Tam. "An iterative algorithm for minimum cross entropy thresholding". *Pattern Recognition Letters* 19 (1998), pp. 771–776 (cit. on pp. 20, 22, 24).

[125] H. Li, A. Kadav, I. Durdanovic, H. Samet, and H. P. Graf. "Pruning Filters for Efficient ConvNets". In: *International Conference on Learning Representations (ICLR)*. 2017 (cit. on p. 59).

[126] L. Li and C. Kessler. "MeterPU: a generic measurement abstraction API enabling energy-tuned skeleton backend selection". In: *IEEE Trustcom/BigDataSE/ISPA*. Vol. 3. 2015 (cit. on p. 107).

[127] Y. Li, J. Dongarra, and S. Tomov. "A note on auto-tuning GEMM for GPUs". In: *International Conference on Computational Science (ICCS)*. Springer. 2009, pp. 884–892 (cit. on pp. 11, 74, 75, 106).

[128] R. Lim, B. Norris, and A. Malony. "Autotuning GPU kernels via static and predictive analysis". In: *International Conference on Parallel Processing (ICPP)*. IEEE. 2017, pp. 523–532 (cit. on pp. 76, 106).

[129] C.-S. Lin, S.-M. Teng, and P.-A. Hsiung. "Auto-tuning for GPGPU applications using performance and energy model". *Journal of Systems Architecture* 62 (2016) (cit. on p. 107).

[130] J. Lin, Y. Rao, J. Lu, and J. Zhou. "Runtime neural pruning". In: *Advances in Neural Information Processing Systems*. 2017, pp. 2181–2191 (cit. on pp. 60, 61).

[131] S. Lin, R. Ji, C. Yan, B. Zhang, L. Cao, Q. Ye, F. Huang, and D. Doermann. "Towards Optimal Structured CNN Pruning via Generative Adversarial Learning". In: *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2019 (cit. on p. 61).

[132] M. Lindauer, K. Eggensperger, M. Feurer, A. Biedenkapp, D. Deng, C. Benjamins, T. Ruhkopf, R. Sass, and F. Hutter. "SMAC3: A versatile Bayesian optimization package for hyperparameter optimization". *The Journal of Machine Learning Research* 23.1 (2022), pp. 2475–2483 (cit. on p. 82).

[133] M. Lindauer, K. Eggensperger, M. Feurer, A. Biedenkapp, D. Deng, C. Benjamins, R. Sass, and F. Hutter. *Sequential Model Algorithm Configuration (SMAC)*. `https://github.com/automl/SMAC3`. 2021 (cit. on p. 83).

[134] G. Litjens, T. Kooi, B. E. Bejnordi, A. A. A. Setio, F. Ciompi, M. Ghafoorian, J. A. van der Laak, B. van Ginneken, and C. I. Sánchez. "A survey on deep learning in medical image analysis". *Medical image analysis* 42 (2017), pp. 60–88 (cit. on p. 23).

[135] S. Liu, J. Tang, Z. Zhang, and J.-L. Gaudiot. "Computer architectures for autonomous driving". *Computer* 50.8 (2017), pp. 18–25 (cit. on p. 73).

[136] W. Liu, Z. Wang, X. Liu, N. Zeng, Y. Liu, and F. E. Alsaadi. "A survey of deep neural network architectures and their applications". *Neurocomputing* 234 (2017), pp. 11–26 (cit. on p. 7).

[137] Y. Liu, W. M. Sid-Lakhdar, O. Marques, X. Zhu, C. Meng, J. W. Demmel, and X. S. Li. "GPTune: Multitask learning for autotuning exascale applications". In: *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 2021, 234–246 (cit. on p. 75).

[138] Z. Liu, M. Sun, T. Zhou, G. Huang, and T. Darrell. "Rethinking the value of network pruning". *International Conference on Learning Representations (ICLR)* (2019) (cit. on p. 61).

[139] J. Long, E. Shelhamer, and T. Darrell. "Fully convolutional networks for semantic segmentation". In: *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2015, pp. 3431–3440 (cit. on p. 67).

[140] M. López-Ibáñez, J. Dubois-Lacoste, L. P. Cáceres, M. Birattari, and T. Stützle. "The irace package: Iterated racing for automatic algorithm configuration". *Operations Research Perspectives* 3 (2016), pp. 43–58 (cit. on pp. 82, 87).

[141] H. R. Lourenço, O. C. Martin, and T. Stützle. "Iterated local search". In: *Handbook of metaheuristics*. Springer, 2003, pp. 320–353 (cit. on p. 79).

[142] J.-H. Luo, J. Wu, and W. Lin. "Thinet: A filter level pruning method for deep neural network compression". In: *IEEE International Conference on Computer Vision (ICCV)*. 2017, pp. 5058–5066 (cit. on pp. 59–61).

[143] M. López-Ibáñez, L. Pérez Cáceres, and J. Dubois-Lacoste. *irace: Iterated Racing for Automatic Algorithm Configuration*. `https://github.com/MLopez-Ibanez/irace`. 2021 (cit. on p. 83).

[144] A. K. Maier, C. Syben, B. Stimpel, T. Würfl, M. Hoffmann, F. Schebesch, W. Fu, L. Mill, L. Kling, and S. Christiansen. "Learning with known operators reduces maximum error bounds". *Nature machine intelligence* 1.8 (2019), pp. 373–380 (cit. on p. 35).

[145] A. Mametjanov, D. Lowell, C.-C. Ma, and B. Norris. "Autotuning stencil-based computations on GPUs". In: *IEEE International Conference on Cluster Computing*. 2012, pp. 266–274 (cit. on pp. 75, 106).

[146] C. McLeavy, M. Chunara, R. Gravell, A Rauf, A Cushnie, C. S. Talbot, and R. Hawkins. "The future of CT: deep learning reconstruction". *Clinical radiology* 76.6 (2021), pp. 407–415 (cit. on p. 33).

[147] X. Mei, L. S. Yung, K. Zhao, and X. Chu. "A measurement study of GPU DVFS on energy conservation". In: *Workshop on Power-Aware Computing and Systems*. 2013 (cit. on pp. 107, 117).

[148] C. D. Meyer. *Matrix analysis and applied linear algebra*. Vol. 71. Siam, 2000 (cit. on p. 63).

[149] A. Milesi. *UNet: semantic segmentation with PyTorch*. `https://github.com/milesial/Pytorch-UNet`. 2020 (cit. on p. 68).

[150] Y. Mingqiang, K. Kidiyo, and R. Joseph. "A survey of shape feature extraction techniques". *Pattern Recognition* 15.7 (2008), pp. 43–90 (cit. on p. 20).

[151] M. Mitchell. *An introduction to genetic algorithms*. MIT press, 1998 (cit. on p. 80).

[152] S. Mittal. "A Survey on optimized implementation of deep learning models on the NVIDIA Jetson platform". *Journal of Systems Architecture* 97 (2019), pp. 428–442 (cit. on p. 73).

[153] T. Miyazaki, I. Sato, and N. Shimizu. "Bayesian Optimization of HPC Systems for Energy Efficiency". In: *IEEE International Conference on High Performance Computing*. 2018 (cit. on p. 107).

[154] P. Molchanov, A. Mallya, S. Tyree, I. Frosio, and J. Kautz. "Importance Estimation for Neural Network Pruning". In: *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2019 (cit. on p. 61).

[155] J. J. Moré. "The Levenberg-Marquardt algorithm: Implementation and theory". In: *Numerical Analysis*. Vol. 630. 1978 (cit. on p. 109).

[156] M. C. Mozer and P. Smolensky. "Skeletonization: A Technique for Trimming the Fat from a Network via Relevance Assessment". In: *Advances in Neural Information Processing Systems*. Ed. by D. S. Touretzky. Morgan-Kaufmann, 1989, pp. 107–115 (cit. on pp. 59, 60).

[157] R Muthukrishnan and M. Radha. "Edge detection techniques for image segmentation". *International Journal of Computer Science and Information Technology* 3.6 (2011), p. 259 (cit. on p. 22).

[158] P. Márquez-Neila, L. Baumela, and L. Alvarez. "A Morphological Approach to Curvature-Based Evolution of Curves and Surfaces". *IEEE Transactions on Pattern Analysis and Machine Intelligence* 36.1 (2014), pp. 2–17 (cit. on p. 24).

[159] L. Nardi, A. Souza, D. Koeplinger, and K. Olukotun. "HyperMapper: a Practical Design Space Exploration Framework". In: *IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*. 2019 (cit. on pp. 75, 107).

[160] Y. S. Nashed, T. Peterka, J. Deng, and C. Jacobsen. "Distributed Automatic Differentiation for Ptychography". *Procedia Computer Science* 108 (2017), pp. 404–414 (cit. on p. 35).

[161] J. A. Nelder and R. Mead. "A simplex method for function minimization". *The computer journal* 7.4 (1965), pp. 308–313 (cit. on p. 81).

[162] Y. E. Nesterov. "A method for solving the convex programming problem with convergence rate $\mathcal{O}(1/k^2)$". In: *Doklady Akademii Nauk*. Vol. 269. Russian Academy of Sciences. 1983, pp. 543–547 (cit. on p. 36).

[163] P. Neubert and P. Protzel. "Compact Watershed and Preemptive SLIC: On Improving Trade-offs of Superpixel Segmentation Algorithms". In: *International Conference on Pattern Recognition*. 2014, pp. 996–1001 (cit. on pp. 20, 22, 24).

[164] J. Nocedal and S. J. Wright. "Conjugate gradient methods". *Numerical optimization* (2006), pp. 101–134 (cit. on p. 81).

[165] C. Nugteren. "CLBlast: A tuned OpenCL BLAS library". In: *International Workshop on OpenCL*. ACM, 2018, 5:1–5:10 (cit. on pp. 84, 112).

[166] C. Nugteren and V. Codreanu. "CLTune: A generic auto-tuner for OpenCL kernels". In: *IEEE International Symposium on Embedded Multicore/ Many-core Systems-on-Chip*. 2015, pp. 195–202 (cit. on pp. 74, 76, 106).

[167] A. Nukada and S. Matsuoka. "Auto-tuning 3-D FFT library for CUDA GPUs". In: *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*. ACM. 2009, p. 30 (cit. on p. 74).

[168] NVIDIA. *NVIDIA Management Library (NVML)*. 2011. URL: `https://developer.nvidia.com/nvidia-management-library-nvml` (cit. on p. 110).

[169] G. Ochoa, M. Tomassini, S. Vérel, and C. Darabos. "A study of NK landscapes' basins and local optima networks". In: *Annual Conference on Genetic and Evolutionary Computation*. 2008, pp. 555–562 (cit. on p. 77).

[170] N. Otsu. "A Threshold Selection Method from Gray-Level Histograms". *IEEE Transactions on Systems, Man, and Cybernetics* 9.1 (1979), pp. 62–66 (cit. on pp. 20, 22, 24, 43).

[171] T. Ozturk, M. Talo, E. A. Yildirim, U. B. Baloglu, O. Yildirim, and U. Rajendra Acharya. "Automated detection of COVID-19 cases using deep neural networks with X-ray images". *Computers in Biology and Medicine* 121 (2020), p. 103792 (cit. on p. 33).

[172] D. Paganin, S. C. Mayo, T. E. Gureyev, P. R. Miller, and S. W. Wilkins. "Simultaneous phase and amplitude extraction from a single defocused image of a homogeneous object". *Journal of microscopy* 206.1 (2002), pp. 33–40 (cit. on p. 42).

[173] L. Page, S. Brin, R. Motwani, and T. Winograd. *The PageRank citation ranking: Bringing order to the web*. Tech. rep. Stanford InfoLab, 1999 (cit. on pp. 17, 75, 100).

[174] W. Palenstijn, K. Batenburg, and J Sijbers. "Performance improvements for iterative electron tomography reconstruction using graphics processing units (GPUs)". *Journal of Structural Biology* 176.2 (2011), pp. 250–253 (cit. on p. 19).

[175] G. Pang, C. Yan, C. Shen, A. v. d. Hengel, and X. Bai. "Self-trained deep ordinal regression for end-to-end video anomaly detection". In: *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2020, pp. 12173–12182 (cit. on p. 8).

[176] V. Panin, G. Zeng, and G. Gullberg. "Total variation regulated EM algorithm [SPECT reconstruction]". *IEEE Transactions on Nuclear Science* 46.6 (1999), pp. 2202–2210 (cit. on p. 52).

[177] H. Park, Y. W. Ko, J. So, and J.-G. Lee. "Performance/Power Design Space Exploration and Analysis for GPU Based Software". *International Journal of Control and Automation* 6.6 (2013) (cit. on p. 107).

[178] J. Park, S. Li, W. Wen, P. T. P. Tang, H. Li, Y. Chen, and P. Dubey. "Faster CNNs with direct sparse convolutions and guided pruning". *International Conference on Learning Representations (ICLR)* (2017) (cit. on p. 60).

[179] S. Park, S. Latifi, Y. Park, A. Behroozi, B. Jeon, and S. Mahlke. "SRTuner: Effective Compiler Optimization Customization by Exposing Synergistic Relations". In: *IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 2022, pp. 118–130 (cit. on p. 75).

[180] A. Paszke, A. Chaurasia, S. Kim, and E. Culurciello. "ENet: A deep neural network architecture for real-time semantic segmentation". *International Conference on Learning Representations (ICLR)* (2017) (cit. on pp. 69, 166).

[181] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Köpf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala. "Pytorch: An imperative style, high-performance deep learning library". *Advances in Neural Information Processing Systems* 32 (2019) (cit. on pp. 8, 36).

[182] P. J. Pavan, M. S. Serpa, E. D. Carreño, V. Martínez, E. L. Padoin, P. O. Navaux, J. Panetta, and J.-F. Mehaut. "Improving Performance and Energy Efficiency of Geophysics Applications on GPU Architectures". In: *Latin American High Performance Computing Conference*. 2018 (cit. on pp. 105, 107).

[183] D. M. Pelt and J. A. Sethian. "A mixed-scale dense convolutional neural network for image analysis". *Proceedings of the National Academy of Sciences* 115.2 (2018), pp. 254–259 (cit. on pp. 23, 24, 68, 166).

[184] M Persson, D Bone, and H Elmqvist. "Total variation norm for three-dimensional iterative reconstruction in limited view angle tomography". *Physics in Medicine and Biology* 46.3 (2001), p. 853 (cit. on p. 52).

[185] F. Petrovič, D. Střelák, J. Hozzová, et al. "A benchmark set of highly-efficient CUDA and OpenCL kernels and its dynamic autotuning with Kernel Tuning Toolkit". *Future Generation Computer Systems* 108 (2020), pp. 161–177 (cit. on p. 74).

[186] G. Pilikos, L. Horchens, K. J. Batenburg, T. van Leeuwen, and F. Lucka. "Fast ultrasonic imaging using end-to-end deep learning". In: *IEEE International Ultrasonics Symposium (IUS)*. 2020, pp. 1–4 (cit. on p. 35).

[187] G. Pilikos, C. L. de Korte, T. van Leeuwen, and F. Lucka. "Single Plane-Wave Imaging using Physics-Based Deep Learning". In: *IEEE International Ultrasonics Symposium (IUS)*. 2021, pp. 1–4 (cit. on p. 35).

[188] L. Pineda, T. Fan, M. Monge, S. Venkataraman, P. Sodhi, R. T. Chen, J. Ortiz, D. DeTone, A. Wang, S. Anderson, et al. "Theseus: A library for differentiable nonlinear optimization". *Advances in Neural Information Processing Systems* 35 (2022), pp. 3801–3818 (cit. on p. 34).

[189] M. J. Powell. "A direct search optimization method that models the objective and constraint functions by linear interpolation". In: *Advances in optimization and numerical analysis*. Springer, 1994, pp. 51–67 (cit. on p. 82).

[190] M. J. Powell. "An efficient method for finding the minimum of a function of several variables without calculating derivatives". *The computer journal* 7.2 (1964), pp. 155–162 (cit. on p. 82).

[191] D. C. Price, M. A. Clark, B. R. Barsdell, R. Babich, and L. J. Greenhill. "Optimizing performance-per-watt on GPUs in high performance computing". *Computer Science-Research and Development* 31.4 (2016) (cit. on pp. 107, 109, 117).

[192] G. Procaccianti, P. Lago, A. Vetro, D. M. Fernandez, and R. Wieringa. "The Green Lab: Experimentation in Software Energy Efficiency". In: *IEEE/ACM International Conference on Software Engineering*. 2015 (cit. on p. 107).

[193] M. Puschel, J. M. Moura, J. R. Johnson, et al. "SPIRAL: Code generation for DSP transforms". *Proceedings of the IEEE* 93.2 (2005), pp. 232–275 (cit. on p. 75).

[194] W. Rahmaniar and W.-J. Wang. "Real-Time automated segmentation and classification of calcaneal fractures in CT images". *Applied Sciences* 9.15 (2019), p. 3011 (cit. on p. 20).

[195] A. Rasch, R. Schulze, M. Steuwer, et al. "Efficient Auto-Tuning of Parallel Programs with Interdependent Tuning Parameters via Auto-Tuning Framework (ATF)". *ACM Transactions on Architecture and Code Optimization (TACO)* 18.1 (2021) (cit. on pp. 76, 106, 107).

[196] W. Rawat and Z. Wang. "Deep convolutional neural networks for image classification: A comprehensive review". *Neural computation* 29.9 (2017), pp. 2352–2449 (cit. on p. 7).

[197] D.-Q. Ren and R. Suda. "Global optimization model on power efficiency of GPU and multicore processing element for SIMD computing with CUDA". *Computer Science-Research and Development* 27.4 (2012) (cit. on p. 107).

[198] A. Renda, J. Frankle, and M. Carbin. "Comparing rewinding and fine-tuning in neural network pruning". *International Conference on Learning Representations (ICLR)* (2020) (cit. on pp. 61, 62).

[199] A. Rodriguez, H. Zhang, K. Wiklund, T. Brodin, J. Klaminder, P. Andersson, and M. Andersson. "Refining particle positions using circular symmetry". *PLOS ONE* 12.4 (Apr. 2017), pp. 1–23 (cit. on p. 20).

[200] J. W. Romein and B. Veenboer. "PowerSensor 2: A Fast Power Measurement Tool". In: *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 2018 (cit. on pp. 107, 109).

[201] Romein, John W. "The Tensor-Core Correlator". *Astronomy & Astrophysics* 656 (2021), A52 (cit. on p. 114).

[202] O. Ronneberger, P. Fischer, and T. Brox. "U-net: Convolutional networks for biomedical image segmentation". In: *International Conference on Medical image computing and computer-assisted intervention*. Springer. 2015, pp. 234–241 (cit. on pp. 22, 24, 59, 68).

[203] L. I. Rudin, S. Osher, and E. Fatemi. "Nonlinear total variation based noise removal algorithms". *Physica D: nonlinear phenomena* 60.1-4 (1992), pp. 259–268 (cit. on p. 52).

[204] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. "Learning representations by back-propagating errors". *Nature* 323.6088 (1986), pp. 533–536 (cit. on p. 8).

[205] S. Ryoo, C. I. Rodrigues, S. S. Stone, S. S. Baghsorkhi, S.-Z. Ueng, J. A. Stratton, and W.-m. W. Hwu. "Program optimization space pruning for a multithreaded GPU". In: *IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 2008, pp. 195–204 (cit. on pp. 76, 106).

[206] H. Salehinejad and S. Valaee. "Pruning of Convolutional Neural Networks using ising Energy Model". In: *IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. 2021, pp. 3935–3939 (cit. on p. 60).

[207] E. Saxe. "Power-efficient software". *Communications of the ACM* 53.2 (2010) (cit. on p. 107).

[208] P. Schiffmann, D. Martin, G. Haase, and G. Offner. "Optimizing a RBF Interpolation Solver for Energy on Heterogeneous Systems". In: *Proceedings of the International Conference on Parallel Computing*. Vol. 32. 2017 (cit. on p. 107).

[209] R. Schoonhoven, J. W. Buurlage, D. M. Pelt, and K. J. Batenburg. "Real-time segmentation for tomographic imaging". In: *IEEE 30th International Workshop on Machine Learning for Signal Processing (MLSP)*. 2020, pp. 1–6 (cit. on pp. 68, 155, 167).

[210] R. Schoonhoven, A. A. Hendriksen, D. M. Pelt, and K. J. Batenburg. "Lean: Graph-Based Pruning for Convolutional Neural Networks By Extracting Longest Chains". *CoRR* (2020). arXiv: `2011.06923 [cs.LG]` (cit. on p. 155).

[211] R. Schoonhoven, A. Skorikov, W. J. Palenstijn, D. M. Pelt, A. A. Hendriksen, and K. J. Batenburg. "How auto-differentiation can improve CT workflows: classical algorithms in a modern framework". *Optics Express* 32.6 (2024), pp. 9019–9041 (cit. on p. 155).

[212] R. Schoonhoven, B. Veenboer, B. van Werkhoven, and K. J. Batenburg. "Going green: optimizing GPUs for energy efficiency through model-steered auto-tuning". In: *IEEE/ACM International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*. 2022, pp. 48–59 (cit. on p. 155).

[213] R. Schoonhoven, B. van Werkhoven, and K. J. Batenburg. "Benchmarking optimization algorithms for auto-tuning GPU kernels". *IEEE Transactions on Evolutionary Computation* (2022) (cit. on pp. 117, 155).

[214] R. A. Schoonhoven. *BlooPy: Black-box optimization Python for bitstring, categorical, and numerical discrete problems with local, and population-based algorithms.* `https://github.com/schoonhovenrichard/BlooPy`. 2021 (cit. on p. 83).

[215] R. A. Schoonhoven. *Data and Plotting scripts for GPU Benchmarking 2021 paper.* `https://github.com/schoonhovenrichard/GPU_benchmarking_paper`. 2021 (cit. on p. 87).

[216] R. A. Schoonhoven. *Optimizing CT workflows with auto-differentiation 2023 paper.* `https://github.com/schoonhovenrichard/AutodiffCTWorkflows`. 2023 (cit. on p. 38).

[217] R. Schwartz, J. Dodge, N. A. Smith, and O. Etzioni. "Green AI". *Communications of the ACM* 63.12 (2020), pp. 54–63 (cit. on p. 105).

[218] A. Sclocco, S. Heldens, and B. van Werkhoven. "AMBER: A real-time pipeline for the detection of single pulse astronomical transients". *SoftwareX* 12 (2020), p. 100549 (cit. on pp. 74, 75).

[219] A. Sclocco. "Accelerating Radio Astronomy with Auto-Tuning" (2017) (cit. on p. 76).

[220] A. Sclocco, H. E. Bal, J. Hessels, J. Van Leeuwen, and R. V. Van Nieuwpoort. "Auto-tuning dedispersion for many-core accelerators". In: *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE. 2014, pp. 952–961 (cit. on pp. 74, 75, 107).

[221] A. Sclocco, J. van Leeuwen, H. E. Bal, and R. V. Van Nieuwpoort. "A real-time radio transient pipeline for ARTS". In: *IEEE Global Conference on Signal and Information Processing (GlobalSIP)*. 2015, pp. 468–472 (cit. on pp. 74, 75).

[222] R. Sedgewick and K. Wayne. *Algorithms*. 4th ed. Addison-wesley professional, 2011, pp. 661–666 (cit. on p. 65).

[223] H. Sedghi, V. Gupta, and P. M. Long. "The singular values of convolutional layers". *International Conference on Learning Representations (ICLR)* (2019) (cit. on pp. 61, 66).

[224] E. Y. Sidky, J. H. Jørgensen, and X. Pan. "Convex optimization problem prototyping for image reconstruction in computed tomography with the Chambolle–Pock algorithm". *Physics in Medicine and Biology* 57.10 (2012), p. 3065 (cit. on p. 52).

[225] K. Simonyan and A. Zisserman. "Very deep convolutional networks for large-scale image recognition". *arXiv preprint arXiv:1409.1556* (2014) (cit. on p. 7).

[226] K. Spafford, J. Meredith, and J. Vetter. "Maestro: Data Orchestration and Tuning for OpenCL Devices". In: *European Conference on Parallel Processing*. 2010 (cit. on p. 106).

[227] M. Stachowski, A. Fiebig, and T. Rauber. "Autotuning based on frequency scaling toward energy efficiency of blockchain algorithms on graphics processing units". *Journal of Supercomputing* (2020) (cit. on p. 105).

[228] R. Storn and K. Price. "Differential evolution–a simple and efficient heuristic for global optimization over continuous spaces". *Journal of global optimization* 11.4 (1997), pp. 341–359 (cit. on p. 82).

[229] E. Strubell, A. Ganesh, and A. McCallum. "Energy and Policy Considerations for Deep Learning in NLP". In: *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*. 2019 (cit. on p. 105).

[230] R. Suda, L. Cheng, and T. Katagiri. "A mathematical method for online autotuning of power and energy consumption with corrected temperature effects". *Procedia Computer Science* 18 (2013) (cit. on p. 107).

[231] M. Thies, F. Wagner, Y. Huang, M. Gu, L. Kling, S. Pechmann, O. Aust, A. Grüneboom, G. Schett, S. Christiansen, and A. Maier. "Calibration by differentiation–Self-supervised calibration for X-ray microscopy using a differentiable cone-beam reconstruction operator". *Journal of Microscopy* 287.2 (2022), pp. 81–92 (cit. on p. 35).

[232] C. Tian, Y. Xu, and W. Zuo. "Image denoising using deep CNN with batch renormalization". *Neural Networks* 121 (2020), pp. 461–473 (cit. on p. 59).

[233] C. Timm, F. Weichert, P. Marwedel, and H. Müller. "Design space exploration towards a realtime and energy-aware GPGPU-based analysis of biosensor data". *Computer Science-Research and Development* 27.4 (2012) (cit. on p. 107).

[234] A. Tiwari, C. Chen, J. Chame, M. Hall, and J. K. Hollingsworth. "A scalable auto-tuning framework for compiler optimization". In: *IEEE International Symposium on Parallel and Distributed Processing (IPDPS)*. 2009, pp. 1–12 (cit. on p. 75).

[235] S. Tomov, R. Nath, H. Ltaief, and J. Dongarra. "Dense linear algebra solvers for multicore with GPU accelerators". In: *International Symposium on Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE. 2010, pp. 1–8 (cit. on pp. 75, 106).

[236] Top500. *List–November 2020*. `https://top500.org/lists/top500/2020/11/`. (accessed November 27, 2020) (cit. on p. 73).

[237] C. Tsallis and D. A. Stariolo. "Generalized simulated annealing". *Physica A* 233.1-2 (1996), pp. 395–406 (cit. on p. 82).

[238] B. Veenboer and J. W. Romein. "Radio-astronomical imaging on graphics processors". *Astronomy and Computing* 32 (2020), p. 100386 (cit. on p. 114).

[239] B. Veenboer and J. W. Romein. "Radio-Astronomical Imaging: FPGAs vs GPUs". In: *European Conference on Parallel Processing*. 2019 (cit. on p. 114).

[240] C. R. Vogel and M. E. Oman. "Iterative Methods for Total Variation Denoising". *SIAM Journal on Scientific Computing* 17.1 (1996), pp. 227–238 (cit. on p. 52).

[241] D. J. Wales and J. P. Doye. "Global optimization by basin-hopping and the lowest energy structures of Lennard-Jones clusters containing up to 110 atoms". *The Journal of Physical Chemistry A* 101.28 (1997), pp. 5111–5116 (cit. on p. 81).

[242] S. van der Walt, J. L. Schönberger, J. Nunez-Iglesias, F. Boulogne, J. D. Warner, N. Yager, E. Gouillart, T. Yu, and the scikit-image contributors. "Scikit-image: image processing in Python". *PeerJ* 2 (June 2014), e453 (cit. on p. 28).

[243] Z. Wang, D. Grewe, and M. F. O'boyle. "Automatic and portable mapping of data parallel programs to opencl for gpu-based heterogeneous systems". *ACM Transactions on Architecture and Code Optimization (TACO)* 11.4 (2014), pp. 1–26 (cit. on p. 76).

[244] Z. Wang, X. Xu, N. Xiong, L. T. Yang, and W. Zhao. "Analysis of parallel algorithms for energy conservation with GPU". In: *IEEE/ACM International Conference on Green Computing and Communications & International Conference on Cyber, Physical and Social Computing*. 2010 (cit. on p. 107).

[245] Z. Wang, C. Li, and X. Wang. "Convolutional neural network pruning with structural redundancy reduction". In: *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2021, pp. 14913–14922 (cit. on p. 60).

[246] B. van Werkhoven. "Kernel Tuner: A search-optimizing GPU code auto-tuner". *Future Generation Computer Systems* 90 (2018) (cit. on pp. 16, 74, 76, 81, 83, 103, 106, 107, 109).

[247] B. van Werkhoven, J. Maassen, H. E. Bal, et al. "Optimizing convolution operations on GPUs using adaptive tiling". *Future Generation Computer Systems* 30 (2014), pp. 14–26 (cit. on pp. 75, 84).

[248] B. van Werkhoven, W. J. Palenstijn, and A. Sclocco. "Lessons learned in a decade of research software engineering GPU applications". In: *International Conference on Computational Science (ICCS)*. Springer. 2020, pp. 399–412 (cit. on p. 73).

[249] R. C. Whaley and J. J. Dongarra. "Automatically Tuned Linear Algebra Software". In: *Proceedings of the ACM/IEEE Conference on Supercomputing*. 1998 (cit. on p. 75).

[250] F.-J. Willemsen, R. Schoonhoven, J. Filipovič, J. Tørring, R. van Nieuwpoort, and B. van Werkhoven. "A Methodology for Comparing Auto-Tuning Optimization Algorithms". *Future Generation Computer Systems* (2024) (cit. on pp. 130, 132, 155).

[251] P. J. Withers. "X-ray nanotomography". *Materials Today* 10.12 (2007), pp. 26–34 (cit. on p. 42).

[252] A. H. Wright, R. K. Thompson, and J. Zhang. "The Computational Complexity of N-K Fitness Functions". *IEEE Transactions on Evolutionary Computation* 4.4 (2000), 373–379 (cit. on p. 83).

[253] Xizhou Feng, Rong Ge, and K. Cameron. "Power and Energy Profiling of Scientific Applications on Distributed Systems". In: *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. Denver, CO, USA, 2005 (cit. on p. 105).

[254] N. Yang, L. v. Stumberg, R. Wang, and D. Cremers. "D3vo: Deep depth, deep pose and deep uncertainty for monocular visual odometry". In: *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2020, pp. 1281–1292 (cit. on p. 59).

[255] T. Yang, Y. Chen, and V. Sze. "Designing Energy-Efficient Convolutional Neural Networks Using Energy-Aware Pruning". In: *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2017, pp. 6071–6079 (cit. on p. 59).

[256] S.-K. Yeom, P. Seegerer, S. Lapuschkin, A. Binder, S. Wiedemann, K.-R. Müller, and W. Samek. "Pruning by explaining: A novel criterion for deep neural network pruning". *Pattern Recognition* 115 (2021), p. 107899 (cit. on p. 61).

[257] M. T. Zeegers, T. van Leeuwen, D. M. Pelt, S. B. Coban, R. van Liere, and K. J. Batenburg. "A tomographic workflow to enable deep learning for X-ray based foreign object detection". *Expert Systems with Applications* 206 (2022), p. 117768 (cit. on p. 49).

[258] M. T. Zeegers. *A collection of 131 CT datasets of pieces of modeling clay containing stones - Part 1 of 5*. Zenodo. Jan. 2022 (cit. on p. 49).

[259] Y. Zhang and F. Mueller. "Auto-generation and auto-tuning of 3D stencil codes on GPU clusters". In: *IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 2012, pp. 155–164 (cit. on pp. 11, 75, 106).

[260] Y. Zhang and H. Yu. "Convolutional Neural Network Based Metal Artifact Reduction in X-Ray Computed Tomography". *IEEE Transactions on Medical Imaging* 37.6 (2018), pp. 1370–1381 (cit. on p. 33).

[261] C. Zhao, B. Ni, J. Zhang, Q. Zhao, W. Zhang, and Q. Tian. "Variational Convolutional Neural Network Pruning". In: *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2019 (cit. on p. 61).

[262] A. Ziabari, S. Venkatakrishnan, M. Kirka, P. Brackman, R. Dehoff, P. Bingham, and V. Paquit. "Beam Hardening Artifact Reduction in X-Ray CT Reconstruction of 3D Printed Metal Parts Leveraging Deep Learning and CAD Models". In: *ASME International Mechanical Engineering Congress and Exposition*. Vol. 2B: Advanced Manufacturing. American Society of Mechanical Engineers. 2020 (cit. on p. 33).

List of publications

Publications that are part of this thesis:

1. R. Schoonhoven, J. W. Buurlage, D. M. Pelt, and K. J. Batenburg. "Real-time segmentation for tomographic imaging". In: *IEEE 30th International Workshop on Machine Learning for Signal Processing (MLSP)*. 2020, pp. 1–6.

2. R. Schoonhoven, A. Skorikov, W. J. Palenstijn, D. M. Pelt, A. A. Hendriksen, and K. J. Batenburg. "How auto-differentiation can improve CT workflows: classical algorithms in a modern framework". *Optics Express* 32.6 (2024), pp. 9019–9041.

3. R. Schoonhoven, A. A. Hendriksen, D. M. Pelt, and K. J. Batenburg. "Lean: Graph-Based Pruning for Convolutional Neural Networks By Extracting Longest Chains". *CoRR* (2020). arXiv: `2011.06923 [cs.LG]`.

4. R. Schoonhoven, B. van Werkhoven, and K. J. Batenburg. "Benchmarking optimization algorithms for auto-tuning GPU kernels". *IEEE Transactions on Evolutionary Computation* (2022).

5. R. Schoonhoven, B. Veenboer, B. van Werkhoven, and K. J. Batenburg. "Going green: optimizing GPUs for energy efficiency through model-steered auto-tuning". In: *IEEE/ACM International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*. 2022, pp. 48–59.

Publications that are not part of this thesis:

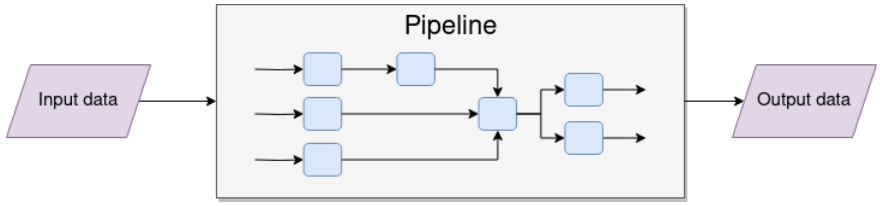
1. F.-J. Willemsen, R. Schoonhoven, J. Filipovič, J. Tørring, R. van Nieuwpoort, and B. van Werkhoven. "A Methodology for Comparing Auto-Tuning Optimization Algorithms". *Future Generation Computer Systems* (2024).

2. A. Laugros, R. Schoonhoven, L. Pavlovic, A. Kuan, C. Bosch, A. Hendriksen, A. Diaz, M. Holler, W. C. Lee, A. Schaefer, K. J. Batenburg, P. Cloetens, N. Vigano, and A. Pacureanu. "Self-supervised image restoration in coherent X-ray neuronal microscopy". To be submitted (2024).

Samenvatting

Veel van onze industriële en wetenschappelijke taken zijn tegenwoordig afhankelijk van krachtige computersystemen die enorme hoeveelheden gegevens verwerken. Deze taken worden uitgevoerd door computationele pijplijnen die vaak uit verschillende algoritmen bestaan, en die berekeningen op data uitvoeren en aan elkaar doorgeven. Conceptueel gezien bestaat het ontwerpproces van een computationele pijplijn uit verschillende fasen.

- **De ideeën fase**: Hier ontwerpen we de pijplijn, beslissen we welke algoritmen we gaan gebruiken en hoe ze op elkaar aansluiten. Figuur S1 toont een conceptueel voorbeeld van hoe we de procedure zouden kunnen ontwerpen.
- **De softwarefase**: Hier beslissen we hoe elk algoritme zijn taak moet uitvoeren. We proberen de code zo te schrijven om de algoritmes zo efficiënt en nauwkeurig mogelijk te maken.
- **De hardwarefase**: Hier wordt bepaald welke computerhardware elk algoritme tot zijn beschikking heeft en hoe de berekening efficiënt uitgevoerd kan worden op die hardware.

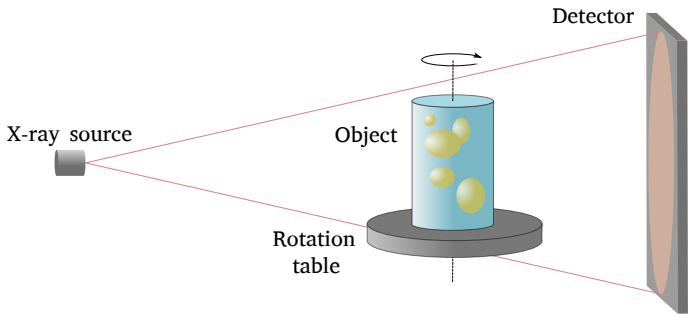


Figuur S1: Voorbeeld van een computationele pijplijn waar data door verschillende algoritmen wordt verwerkt en doorgegeven wordt totdat het uiteindelijke resultaat is berekend.

Het praktisch draaien van zo'n computationele pijplijn kan een uitdaging zijn. We moeten vaak de manier waarop de algoritmen werken aanpassen, ervoor zorgen dat ze geen tijd en energie verspillen, de hardware verbeteren en soms moeten we de hele pijplijn opnieuw ontwerpen om het beste resultaat te krijgen. Bovendien moeten we, gezien de huidige milieu overwegingen, ook proberen om onze pijplijn milieuvriendelijker te maken. Om dit te doen richten we ons in dit proefschrift op alle drie de fasen om onze computationele pijplijnen te optimaliseren.

Een van de toepassingen waar zulke uitdagende computationele pijplijnen voorkomen is bij de röntgen-computertomografie. Computertomografie (CT) is een techniek om de binnenkant van objecten te visualiseren door middel van bestraling (Figuur S2). Dit is een krachtige technologie omdat we de binnenkant

van objecten kunnen bestuderen zonder ze te beschadigen of open te maken. Één van de computationele uitdagingen komt voort uit het feit dat CT meestal in 3D wordt uitgevoerd, wat betekent dat het om grote hoeveelheden gegevens gaat en dat elke computationele stap ook een grote hoeveelheid gegevens moet werken. Bovendien bevatten CT-pijplijnen meestal verschillende verwerkingsstappen die we willen optimaliseren.



Figuur S2: Een voorbeeld CT opstelling die men in vaak in laboratoria aantreft.

Verschillende instellingen van een CT-scanner, zoals de energie van de bundel en vanuit welke positie de röntgenopnames worden opgevangen, moeten correct worden gekozen om een scherp CT-beeld met een hoog contrast te krijgen. Dit creëert een uitdaging vanuit een optimalisatie oogpunt. Vaak is het gewenste resultaat pas aan het einde van de berekening zichtbaar, terwijl het veroorzaakt kan worden door een instelling aan het begin van de CT procedure. De uitlijning van de machine kan bijvoorbeeld niet goed zijn, of we hebben onze algoritmen met de verkeerde parameters gedraaid. Het probleem wordt nog verergerd doordat men vaak extra bewerkingsstappen wil uitvoeren aan het einde van de reconstructies.

In hoofdstuk 2 proberen we met AI een zogenaamde segmentatiestap toe te voegen aan een CT procedure, zodat de berekening live kan worden uitgevoerd. In hoofdstuk 3 proberen we het optimalisatieprobleem aan te pakken om algoritmische parameters over de gehele pijplijn tegelijk te optimaliseren. In hoofdstuk 4 kijken we nog eens naar de AI die we gebruikten in hoofdstuk 2, en proberen we deze aanzienlijk te versnellen. We doen dit door een nieuwe methode te ontwikkelen die het neurale netwerk kan afslanken tot een veel kleinere versie van zichzelf, maar met een vergelijkbare nauwkeurigheid. In hoofdstuk 5 kijken we naar de hardwarefase. In het bijzonder kijken we naar de grafischekaart (Graphics Processing Unit of GPU genoemd), en bekijken we welke algoritmen het beste zijn voor het structureren van deze GPU berekeningen. In hoofdstuk 6 bekijken we GPU berekeningen vanuit een milieu- en energieverbruik perspectief, en introduceren we een model om deze energie-efficiënter te maken. Uiteindelijk bekijken we hoeveel energie onze methode bespaart voor een groot Europees netwerk van radiotelescopen genaamd de Low-frequency Array (LOFAR).

Summary

Today, many of our industrial and scientific tasks rely heavily on powerful computer systems that process huge amounts of data. These tasks are performed by computational pipelines that often consist of several algorithms that calculate the data, and pass the results on to each other. Conceptually, the process of designing a computational pipeline for these tasks has different stages.

- **The Idea Stage**: This is where we plan how the pipeline will work, deciding which algorithms to use and how they connect. Figure S3 shows a conceptual example of how we might design the workflow.
- **The Software Stage**: Here, we decide how each algorithm must perform its task. We try to write the code to make the algorithms as efficient and accurate as possible.
- **The Hardware Stage**: This is about deciding what computational hardware each algorithm will have available, and how to efficiently run the computation on that hardware.

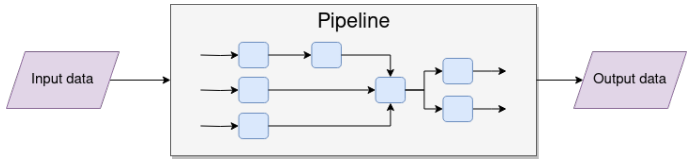


Figure S3: Example computational pipeline illustrating how input data is processed by different algorithms and passed on until the final result is computed.

Running such a pipeline can be challenging. We often need to tweak the way the algorithms work, make sure they're not wasting time and energy, upgrade their hardware, and sometimes, we need to redesign the whole pipeline to get the best result. Plus, with today's environmental challenges, we must also try to make our pipeline eco-friendly. To do this, in this thesis, we focus on all three stages to optimize our computational pipelines.

One of the applications where such challenging computational pipelines occur is in the field of X-ray computed tomography. Computed tomography (CT) is a technique for visualizing the interior of objects with some form of radiation (Figure S4). This makes it a powerful tool as we can study the internals of objects without damaging them, or opening them up. One of the computational challenges comes from the fact that CT is typically done in 3D, which means that large amounts of data are involved, and any computational step needs to work on a large batch of data. Furthermore, CT pipelines usually come with several different processing steps that we would like to optimize.

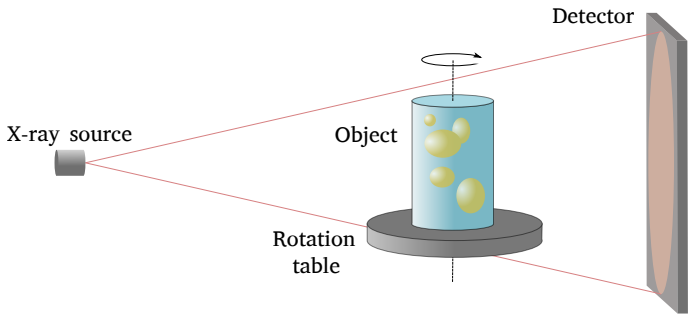


Figure S4: An example setup of an X-ray CT scanner that is common in laboratories.

Several settings of a CT scanner, such as the beam's energy and from which position it captures the X-ray photographs, need to be chosen correctly to get a sharp CT image with high contrast. This creates a challenge from an optimization point of view. Often, the desired result (or an undesirable image feature) is only visible at the very end, whereas it may be caused by a setting at the beginning of the CT workflow. For example. the alignment of the machine may be off, or we ran our algorithms with the wrong parameters. The problem is compounded because users often want to perform extra processing steps at the end of the reconstruction.

In Chapter 2, we attempt to add a so-called segmentation step to an X-ray CT workflow using AI, such that it can perform the computation in real time (meaning the operator gets a live view). In Chapter 3, we attempt to tackle the problem of optimizing algorithmic parameters along the whole pipeline for a final result. In Chapter 4, we revisit the AI that we used in Chapter 2, and attempt to speed it up significantly using a technique called "pruning". In Chapter 5, we look at the final hardware stage. In particular, we look at a type of chip called a Graphics Processing Unit (GPU), and which optimization algorithms are best for structuring these GPU computations. In Chapter 6, we look at GPU computations from an environmental and energy consumption perspective, and propose a model to make them more energy efficient. In the end, we look at how much energy our method saves for a large European network of radio telescopes called the Low-frequency Array (LOFAR).

Curriculum Vitae

Richard Schoonhoven was born in 1995 in Utrecht, the Netherlands. He completed his secondary education at RSG Broklede in Breukelen, the Netherlands. Afterwards, he completed bachelor's degrees (both cum laude) in mathematics and physics at Utrecht University in 2016, and obtained master's degrees (both cum laude) in mathematics and computer science in 2019 from Utrecht University. The master's thesis with the title "Improving cryo-ET reconstructions of ER-associated ribosomes with tomographic reconstruction methods and deep learning" was supervised by Dr. Tristan van Leeuwen. In 2019, he started as a PhD candidate at Centrum Wiskunde & Informatica (the national research institute for mathematics and computer science in Amsterdam) under the supervision of Prof.dr. K.J. Batenburg.

Acknowledgments

First, I thank my advisors, Prof. Joost Batenburg and Dr. Daniël Pelt, for the time and energy they have put into providing motivating, educational, and fun supervision of our research projects. In particular, I would like to thank them for providing me with large amounts of freedom to pursue different research directions, which has made the past four years a thoroughly enjoyable part of my career.

In addition, I would like to thank Dr. Ben van Werkhoven for his energetic and stimulating approach to research and supervision, which has made several of our projects a great pleasure to collaborate on.

I would also like to thank my colleague Dr. Alexander Skorikov for his companionable collaboration, and his tireless energy for dealing with my ceaseless stream of thoughts, and at times, tumultuous working style.

A special thank you goes out to my co-authors at the ESRF who have hosted me for several weeks on many occasions, and for their efforts to make me feel welcome and show me around their impressive facilities. I have to mention in particular Dr. Alexandra Pacureanu who was kind enough to allow me to stay in their city apartment in Grenoble during these trips.

I would like to thank my co-authors Allard Hendriksen, Bram Veenboer, and Jan-Willem Buurlage, who have greatly helped me with my research projects and spent time and effort teaching me more about new topics.

In particular, I would like to thank Willem Jan Palenstijn for his expert and patient help on countless problems I encountered.

I thank the colleagues whom I shared an office with, Vladyslav Andriiashen, Mathé Zeegers, Adriaan Graas, Poulami Ganguly, Francien Bossema, Jordi Minnema, Dirk Schut, Maximilian Kiss, Tianyuan Wang, Rien Lagerwerf for providing stimulating conversations and a pleasant work environment.

Many others contributed to a great research environment, among which Floris-Jan, Jiayang, Serban, Alex, Nicola, Henri, Maureen, Dzemila, Georgios, Sophia, Felix, Rob, Robert, Hamid, Roozbeh, Ajynkya, Jan, and Tristan.

I would like to thank my family and friends for their support, and camaraderie over the years.

Finally, I would like to thank Sasha for her love, infinite patience, and wholehearted support.

A
Appendices

A.1 Appendix: (LEAN) graph-based pruning for convolutional neural networks by extracting longest chains

A.1.1 Datasets

In this appendix we discuss some more details on the datasets used for experimentation.

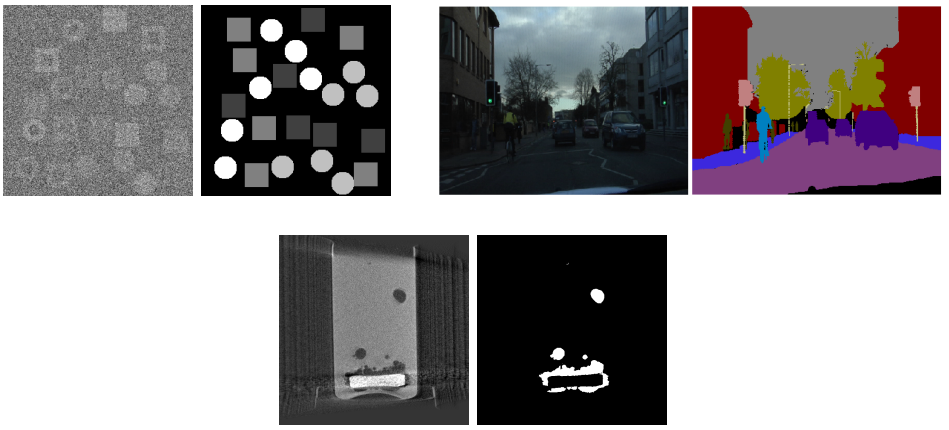
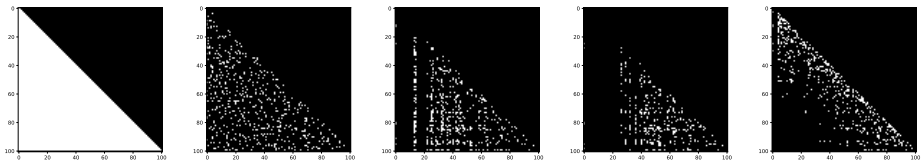


Figure A1: Example input and target images of the (top left) Circle-Square (CS), (top right) CamVid, (bottom) real-world dynamic CT datasets.

Simulated Circle-Square (CS) dataset: We used a simulated high-noise 5-class segmentation dataset containing 256×256 images of randomly placed squares and circles (CS dataset) [183] (see Figure A1). The objects were assigned a random grey value and Gaussian noise was added to the images. In total, we generated 1000 training, 250 validation, and 100 test images. Experimental results on the CS dataset are quantified using global accuracy, i.e., the ratio of correctly classified pixels, regardless of class, to the total number of pixels.

CamVid: The Cambridge-driving Labeled Video Database (CamVid) [24, 25] is a collection of videos with labels, captured from the perspective of a driving automobile. In total, 700 labeled frames are split into 367 training, 100 validation, and 233 test images. As there are few training images, we combined the training and validation datasets and trained for a fixed 500 epochs. Similar to other papers that apply CNNs to CamVid [9, 180], we use 11 classes, and a single class representing unlabeled pixels (see Figure A1).

We used median frequency balancing [56] to balance classes for training, and set the unlabeled class weights to zero. During training, we used data augmentation by cropping and (horizontally) flipping input images.



Unpruned Random Magnitude Operator norm LEAN

Figure A2: Adjacency matrices of active convolutions (in white) after pruning. All pruned network were pruned to a ratio of 10%. From left to right, we have the unpruned matrix of a 100-layer MS-D network trained on the real-world dynamic CT dataset, randomly pruned convolutions, structured magnitude pruning, structured operator norm pruning, and LEAN.

Real-world dynamic CT dataset: The real-time dynamic X-ray CT dataset contains images of a dissolving tablet suspended in gel [38, 39]. The bubbles are to be segmented within a glass container filled with gel [209] (see Figure A1). The dataset consists of 512×512 images, split into 9216 training images, 2048 validation images, and 1536 test images. As in [209], we use the F1-score because the large amount of background pixels make global accuracy an unsuitable metric.

A.1.2 Reducing the size of the pruning graph

The procedure outlined in Section 4.4 can lead to large pruning graphs, but the size of the graph can be reduced. First, according to Equation 4.3, the operator norm of ReLU is 1. Therefore, the combination of a convolution followed by a ReLU can be combined into a single edge whose weight equals the norm of the convolution.

Batch normalization often succeeds a convolution. Batch-normalization scaling is applied with different learned parameters per input channel, and output a single channel. Therefore, the input convolution edge and the following batch normalization edges can be combined. The edges can be combined into a single edge whose weight is the product of the two edge weights, preserving the path length.

A.1.3 Structure of pruned MS-D networks

To investigate the structure of pruned networks we plotted the adjacency matrices of pruned networks where an entry is 0 if it is pruned (black) and 1 if it is still active (white). Here, we show the adjacency matrices of MS-D networks pruned to a ratio of 10% in Figure A2. After pruning, LEAN retains only connections linked to nearby layers in the densely connected MS-D network. Compared to individual filter pruning, LEAN exposes a distinct structure which may suggest that LEAN could be used for architecture discovery.

A.2 Appendix: Benchmarking optimization algorithms for auto-tuning GPU kernels

A.2.1 Tunable parameters per GPU kernel

In Table A.1 we show the tunable parameters per kernel, and the values each parameter could take. For the convolution kernel, the MI50 GPU (the only AMD model) required a different problem setup due to hardware constraints.

Kernel	parameter to tune	list of values	number of possible values
Convolution (except MI50)	`block_size_x`	1, 2, 4, 8, 16, 32, 48, 64, 80, 96, 112, 128	12
	`block_size_y`	1, 2, 4, 8, 16, 32	6
	`tile_size_x`	1, 2, 3, 4, 5, 6, 7, 8	8
	`tile_size_y`	1, 2, 3, 4, 5, 6, 7, 8	8
	`use_padding`	0, 1	2
	`read_only`	0, 1	2
Convolution (MI50)	`block_size_x`	16, 32, 48, 64, 80, 96, 112, 128	8
	`block_size_y`	1, 2, 4, 8, 16, 32	6
	`tile_size_x`	1, 2, 4	3
	`tile_size_y`	1, 2, 4	3
	`use_padding`	0, 1	2
GEMM	`MWG`	16, 32, 64, 128	4
	`NWG`	16, 32, 64, 128	4
	`MDIMC`	8, 16, 32	3
	`NDIMC`	8, 16, 32	3
	`MDIMA`	8, 16, 32	3
	`NDIMB`	8, 16, 32	3
	`VWM`	1, 2, 4, 8	4
	`VWN`	1, 2, 4, 8	4
	`SA`	0, 1	2
	`SB`	0, 1	2
Point-in-polygon	`block_size_x`	32, 64, 96, 128, 160, 192, 224, 256, 288, 320, 352, 384, 416, 448, 480, 512, 544, 576, 608, 640, 672, 704, 736, 768, 800, 832, 864, 896, 928, 960, 992	31
	`tile_size`	1, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20	11
	`between_method`	0, 1, 2, 3	4
	`use_precomputed_slopes`	0, 1	2
	`use_method`	0, 1, 2	3

Table A.1: Tunable parameters per kernel, and list of possible values for each parameter.

A.2.2 Alternative splits for competition heatmaps

In Figures A1 and A6 we show the algorithm competition heatmaps such as in Figures 5.1, 5.2 and 5.3, but when split at 100 and 400 budgets instead of 200.

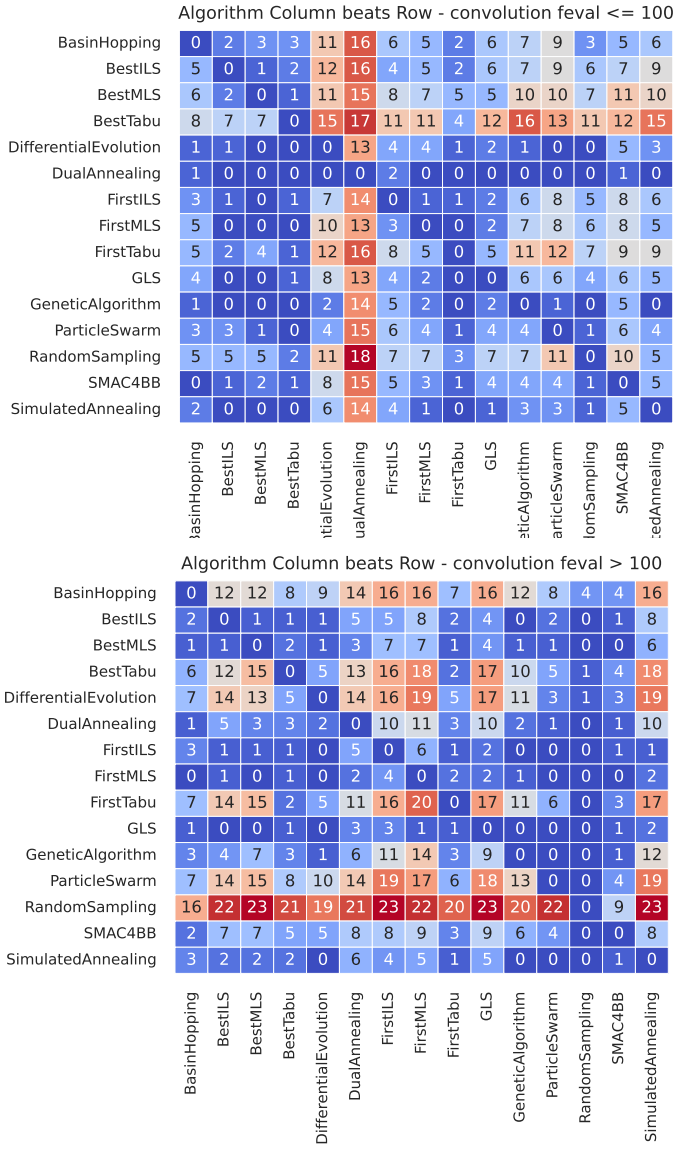


Figure A1: (Convolution:) Occurrences when the column algorithm found better solutions than the row algorithm. An occurrence is counted when 50 runs for a budget are statistically significantly better according to a two-sample independent t-test ($\alpha = 0.05$). (Top): Heatmap for low ≤ 100 budgets (25, 50, 100). (Bottom): Heatmap for mid and high > 100 budgets (200, 400, 800, 1600). Algorithms with low values (blue) in their rows were not often beaten for those budgets, and algorithms with high values in their column (red) often beat other algorithms.

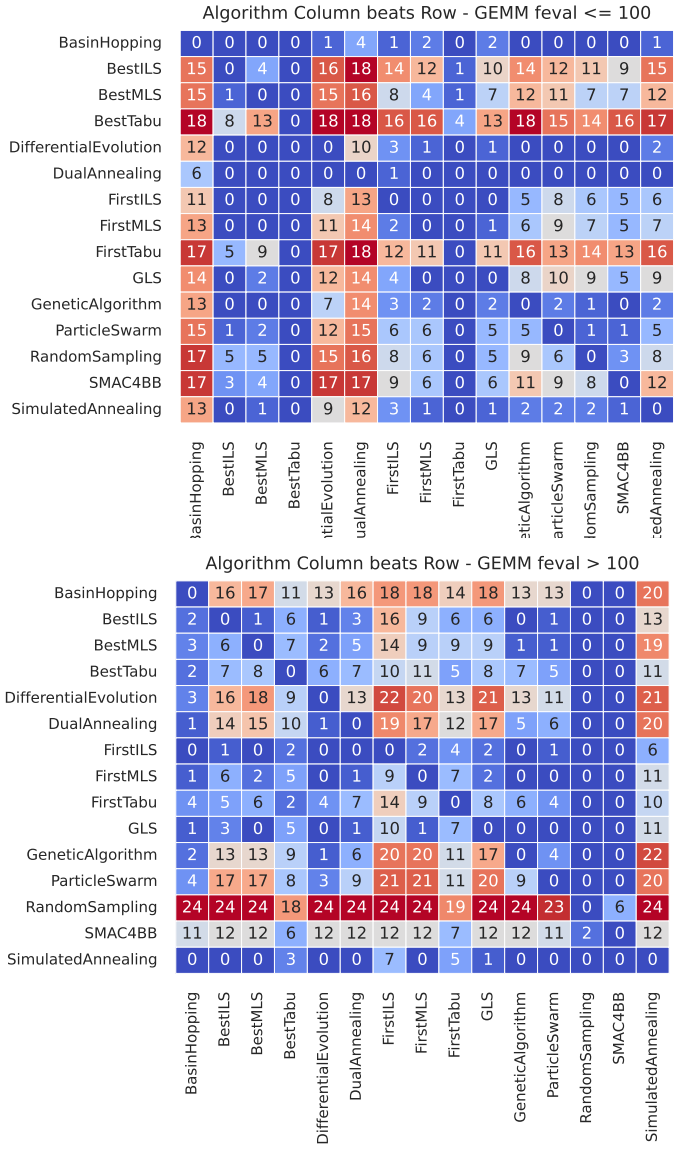


Figure A2: (GEMM:) Occurrences when the column algorithm found better solutions than the row algorithm. An occurrence is counted when 50 runs for a budget are statistically significantly better according to a two-sample independent t-test ($\alpha = 0.05$). (Top): Heatmap for low ≤ 100 budgets (25, 50, 100). (Bottom): Heatmap for mid and high > 100 budgets (200, 400, 800, 1600). Algorithms with low values (blue) in their rows were not often beaten for those budgets, and algorithms with high values in their column (red) often beat other algorithms.

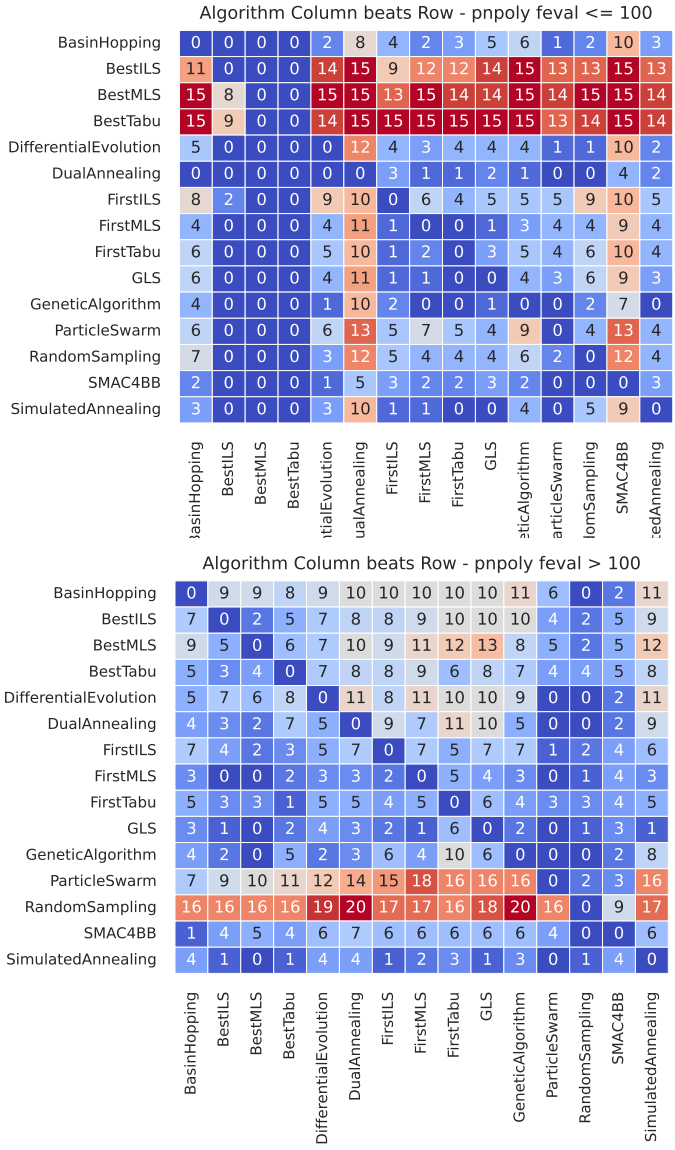


Figure A3: (PnPoly:) Occurrences when the column algorithm found better solutions than the row algorithm. An occurrence is counted when 50 runs for a budget are statistically significantly better according to a two-sample independent t-test ($\alpha = 0.05$). (Top): Heatmap for low ≤ 100 budgets (25, 50, 100). (Bottom): Heatmap for mid and high > 100 budgets (200, 400, 800, 1600). Algorithms with low values (blue) in their rows were not often beaten for those budgets, and algorithms with high values in their column (red) often beat other algorithms.

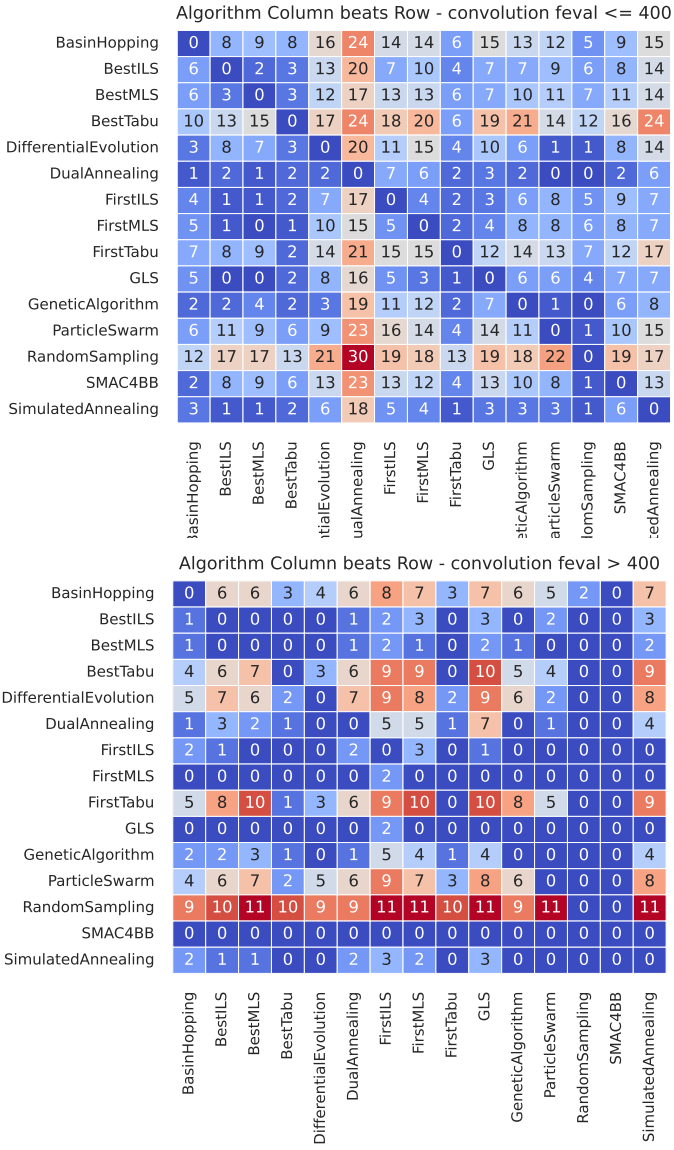


Figure A4: (Convolution:) Occurrences when the column algorithm found better solutions than the row algorithm. An occurrence is counted when 50 runs for a budget are statistically significantly better according to a two-sample independent t-test ($\alpha = 0.05$). (Top): Heatmap for low ≤ 400 budgets (25, 50, 100, 200, 400). (Bottom): Heatmap for mid and high > 400 budgets (800, 1600). Algorithms with low values (blue) in their rows were not often beaten for those budgets, and algorithms with high values in their column (red) often beat other algorithms.

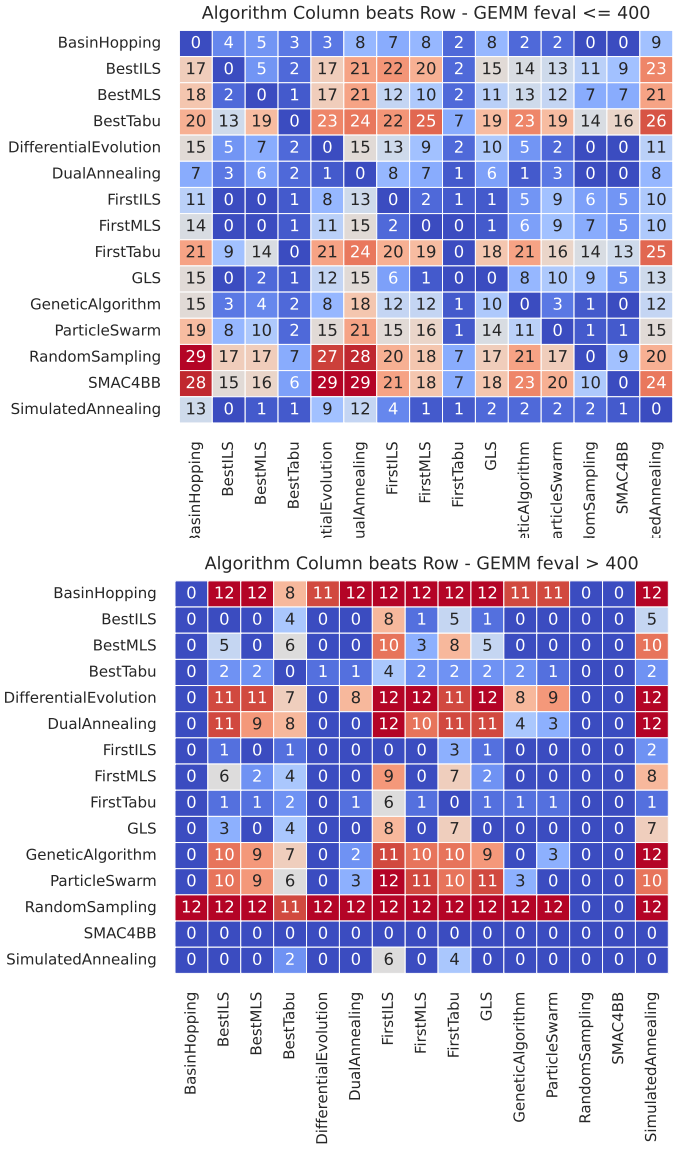


Figure A5: (GEMM:) Occurrences when the column algorithm found better solutions than the row algorithm. An occurrence is counted when 50 runs for a budget are statistically significantly better according to a two-sample independent t-test ($\alpha = 0.05$). (Top): Heatmap for low ≤ 400 budgets (25, 50, 100, 200, 400). (Bottom): Heatmap for mid and high > 400 budgets (800, 1600). Algorithms with low values (blue) in their rows were not often beaten for those budgets, and algorithms with high values in their column (red) often beat other algorithms.

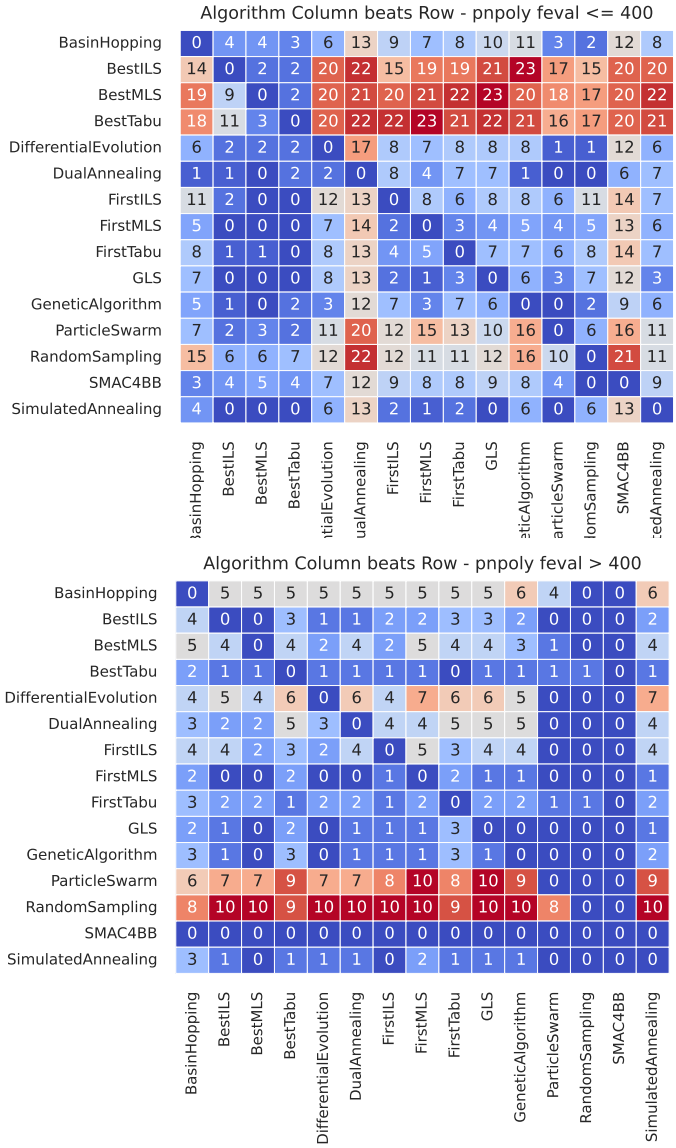


Figure A6: (PnPoly:) Occurrences when the column algorithm found better solutions than the row algorithm. An occurrence is counted when 50 runs for a budget are statistically significantly better according to a two-sample independent t-test ($\alpha = 0.05$). (Top): Heatmap for low ≤ 400 budgets (25, 50, 100, 200, 400). (Bottom): Heatmap for mid and high > 400 budgets (800, 1600). Algorithms with low values (blue) in their rows were not often beaten for those budgets, and algorithms with high values in their column (red) often beat other algorithms.

A.2.3 Per kernel graphs of experimental results

In Figures A7 to A15 we show plots of algorithm performance in terms of fraction of optimal fitness found for certain budget used (per GPU).

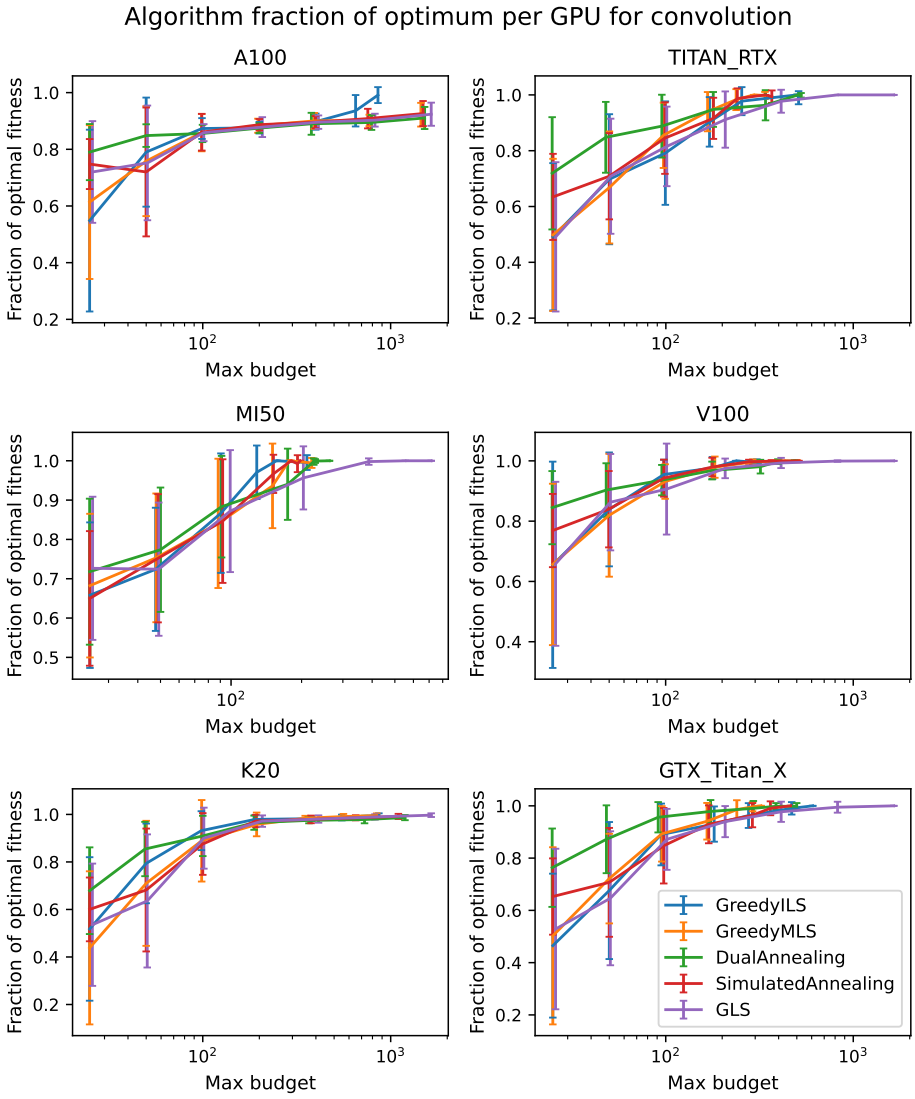


Figure A7: **Convolution:** Fraction of optimal runtime per GPU for FirstILS, FirstMLS, dual annealing, simulated annealing, and GLS over 50 runs. Each point is the mean fraction of optimal runtime found (y-axis) for mean budget used (logarithmic x-axis), with error bars indicating the standard deviation in fraction of optimum.

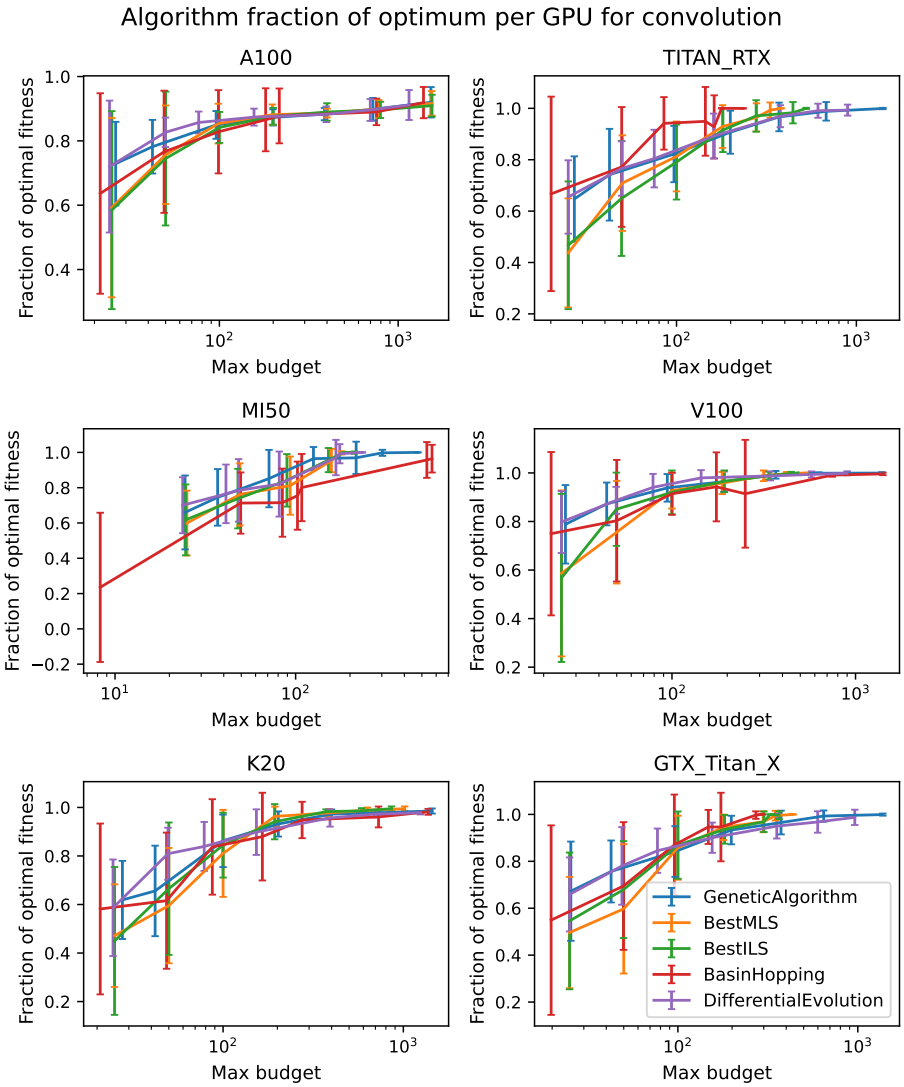


Figure A8: **Convolution:** Fraction of optimal runtime per GPU for GA, BestMLS, BestILS, basin hopping, and differential evolution over 50 runs. Each point is the mean fraction of optimal runtime found (y-axis) for mean budget used (logarithmic x-axis), with error bars indicating the standard deviation in fraction of optimum.

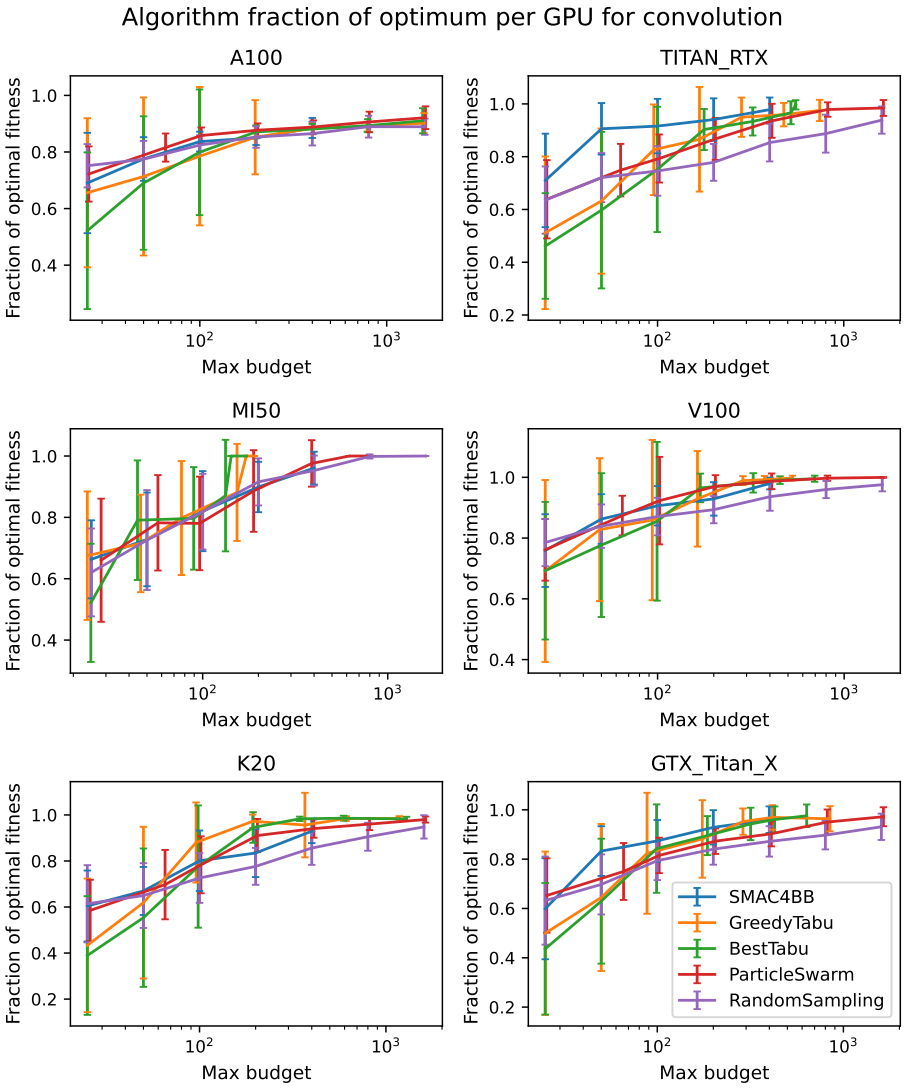


Figure A9: **Convolution:** Fraction of optimal runtime per GPU for SMAC, FirstTabu, BestTabu, PSO, and random sampling over 50 runs. Each point is the mean fraction of optimal runtime found (y-axis) for mean budget used (logarithmic x-axis), with error bars indicating the standard deviation in fraction of optimum.

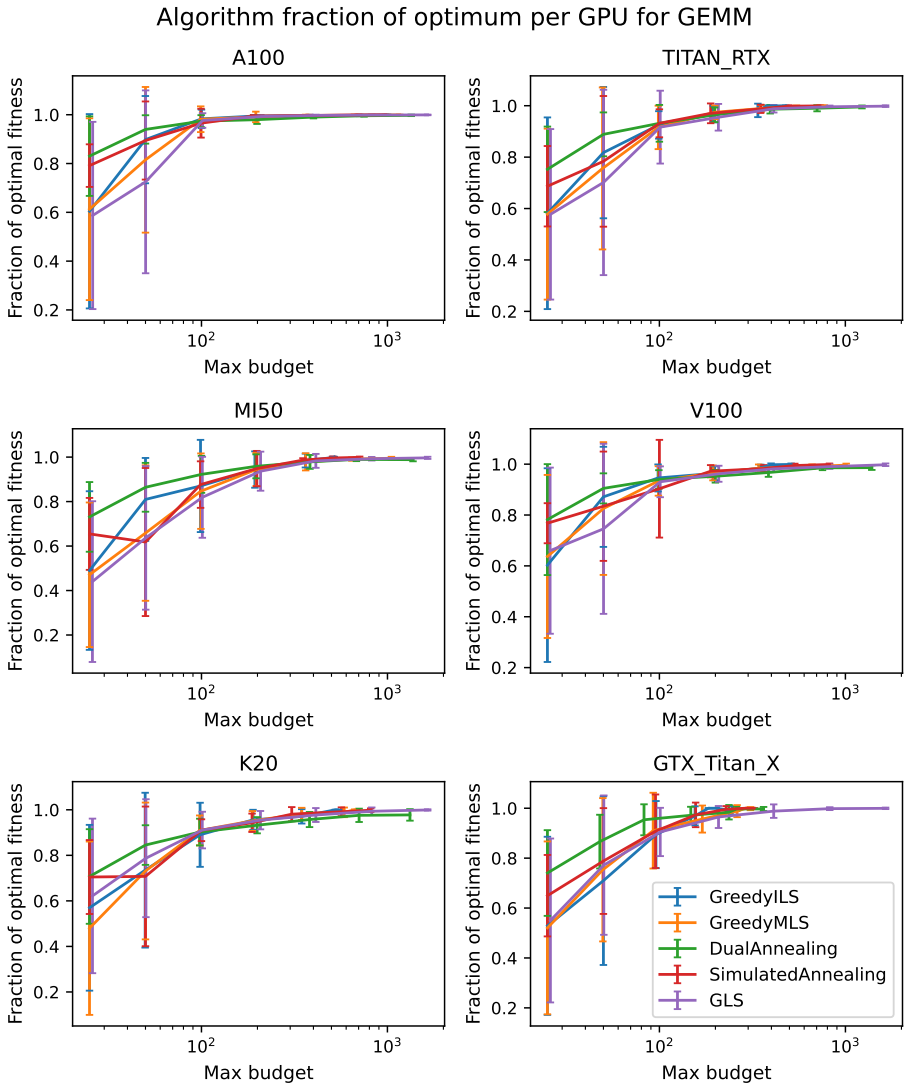


Figure A10: **GEMM:** Fraction of optimal runtime per GPU for FirstILS, FirstMLS, dual annealing, simulated annealing, and GLS over 50 runs. Each point is the mean fraction of optimal runtime found (y-axis) for mean budget used (logarithmic x-axis), with error bars indicating the standard deviation in fraction of optimum.

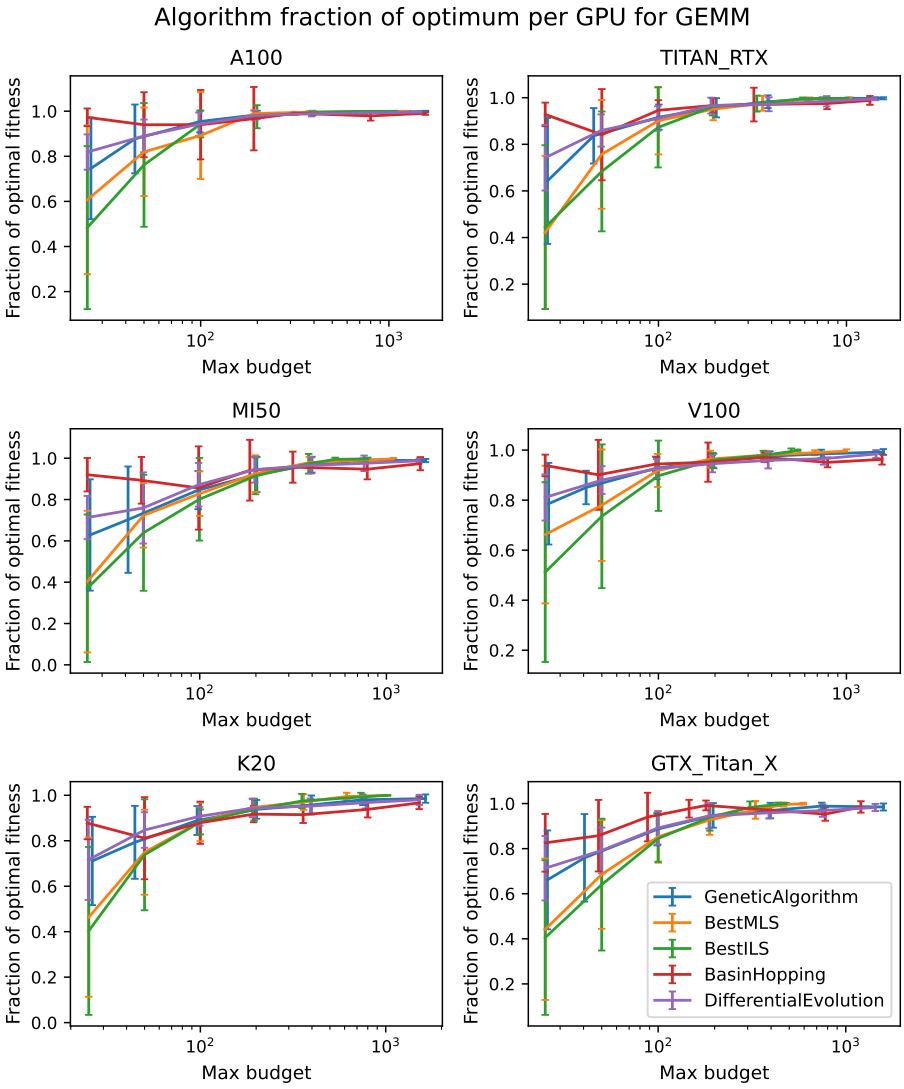


Figure A11: **GEMM:** Fraction of optimal runtime per GPU for GA, BestMLS, BestILS, basin hopping, and differential evolution over 50 runs. Each point is the mean fraction of optimal runtime found (y-axis) for mean budget used (logarithmic x-axis), with error bars indicating the standard deviation in fraction of optimum.

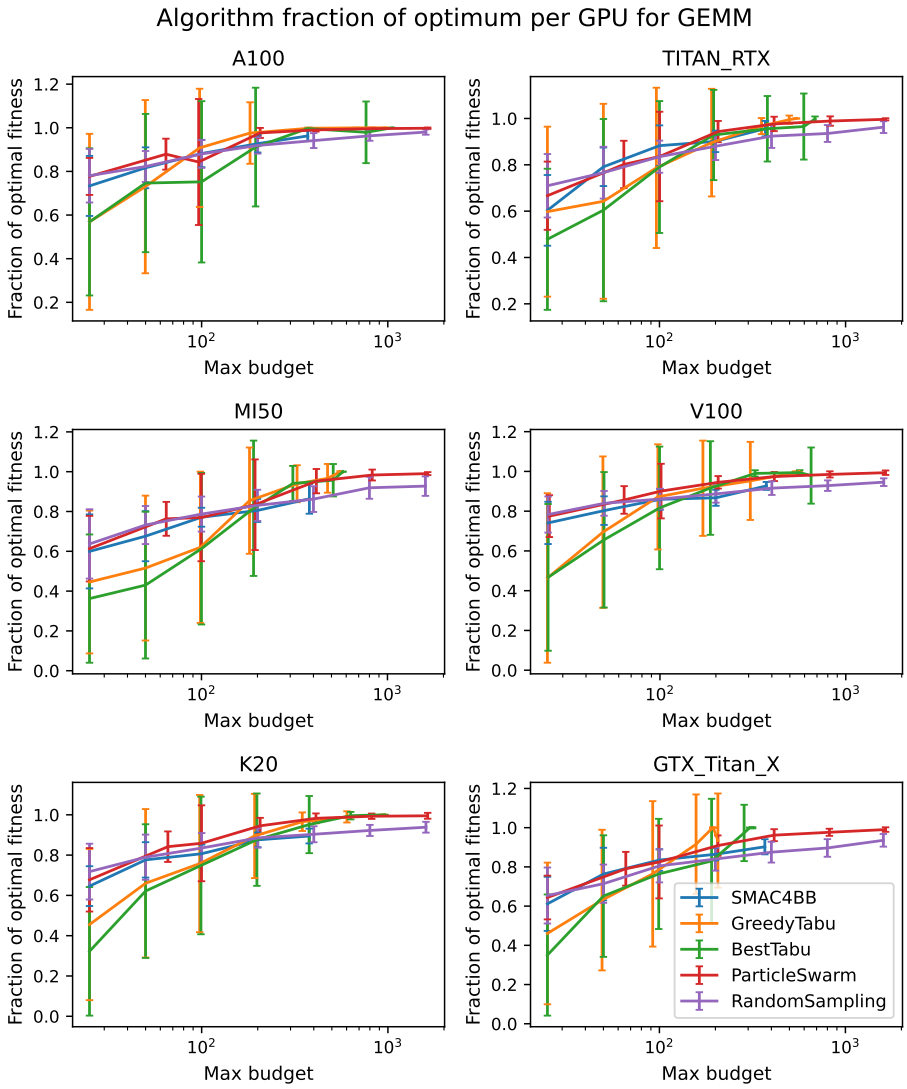


Figure A12: **GEMM:** Fraction of optimal runtime per GPU for SMAC, FirstTabu, BestTabu, PSO, and random sampling over 50 runs. Each point is the mean fraction of optimal runtime found (y-axis) for mean budget used (logarithmic x-axis), with error bars indicating the standard deviation in fraction of optimum.

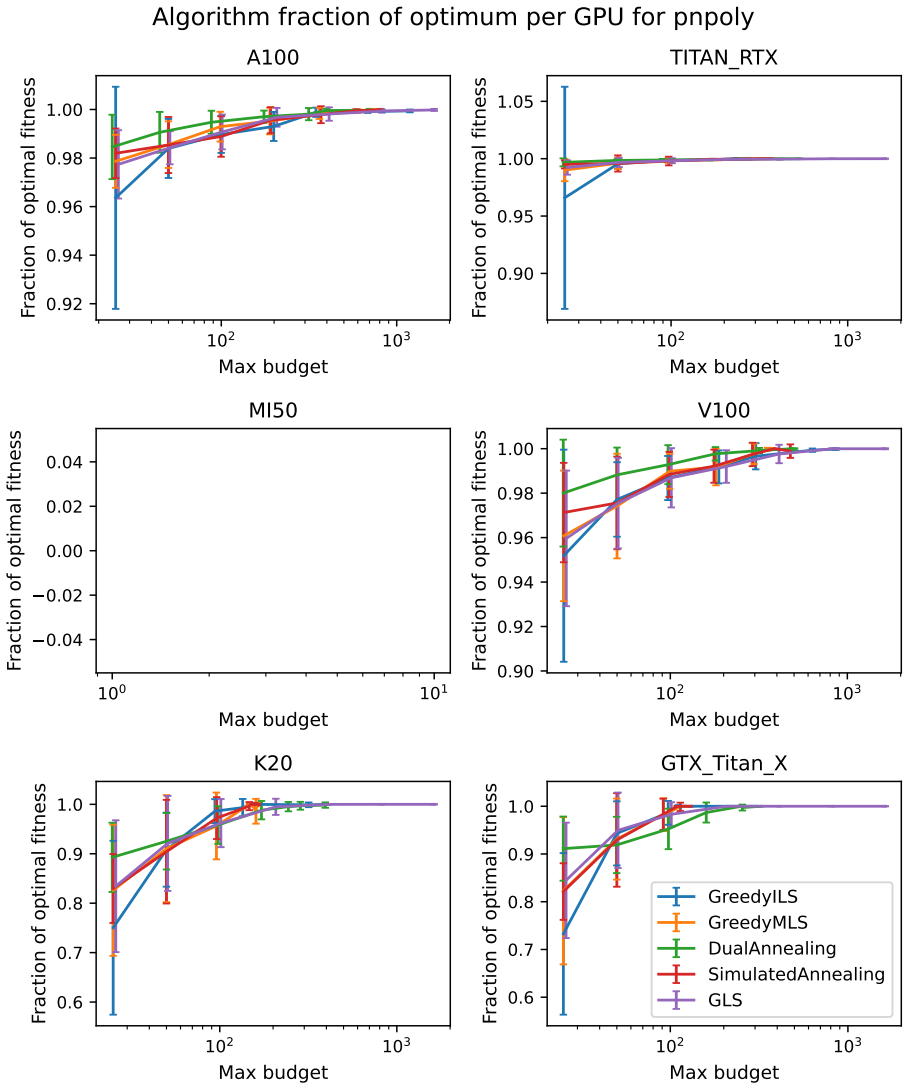


Figure A13: **Point-in-polygon:** Fraction of optimal runtime per GPU for FirstILS, FirstMLS, dual annealing, simulated annealing, and GLS over 50 runs. Each point is the mean fraction of optimal runtime found (y-axis) for mean budget used (logarithmic x-axis), with error bars indicating the standard deviation in fraction of optimum. The point-in-polygon kernel was not implemented for the MI50 GPU.

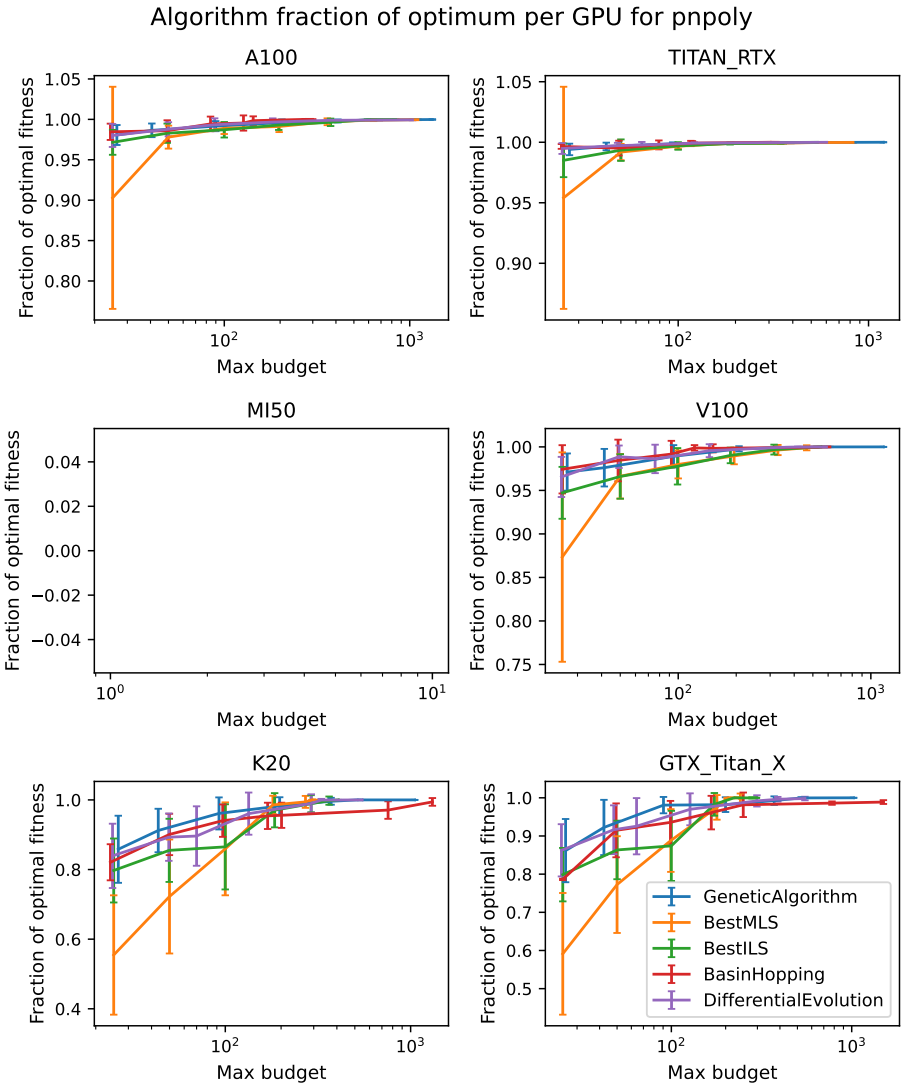


Figure A14: **Point-in-polygon:** Fraction of optimal runtime per GPU for GA, BestMLS, BestILS, basin hopping, and differential evolution over 50 runs. Each point is the mean fraction of optimal runtime found (y-axis) for mean budget used (logarithmic x-axis), with error bars indicating the standard deviation in fraction of optimum. The point-in-polygon kernel was not implemented for the MI50 GPU.

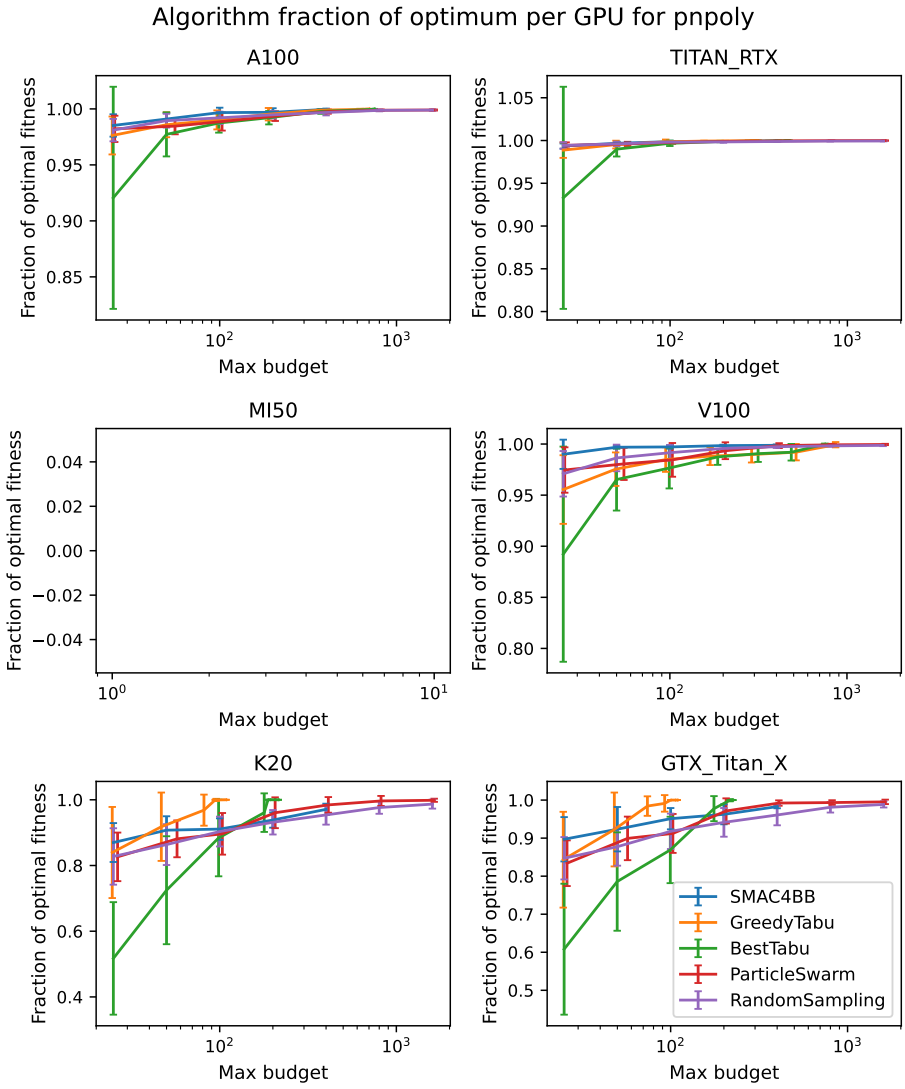


Figure A15: **Point-in-polygon:** Fraction of optimal runtime per GPU for SMAC, FirstTabu, BestTabu, PSO, and random sampling over 50 runs. Each point is the mean fraction of optimal runtime found (y-axis) for mean budget used (logarithmic x-axis), with error bars indicating the standard deviation in fraction of optimum. The point-in-polygon kernel was not implemented for the MI50 GPU.