Bachelor's Thesis

submitted in partial fulfillment of the
requirements for the course "Applied Computer Science"

Machine Learning with BART: Performance Analysis and Memory Footprints of Various Optimizers

Han Sun

MatrNr: 21914851

First Supervisor: Prof. Dr. Julian Kunkel
Second Supervisor: Dr. Patrick Höhn

Georg-August-Universität Göttingen
Institute of Computer Science
ISSN: 1612-6793

January 7, 2026

Georg-August-Universität Göttingen
Institute of Computer Science

Goldschmidtstraße 7
37077 Göttingen
Germany

☎ +49 (551) 39-172000
FAX +49 (551) 39-14403
✉ office@informatik.uni-goettingen.de
www.informatik.uni-goettingen.de

Abstract

Magnetic Resonance Imaging (MRI) is one of the most widely used diagnostic imaging techniques in clinical practice, valued for its non-invasive, non-radioactive nature and its high resolution in examining the nervous system and soft tissues. However, MRI image reconstruction remains computationally intensive, and the choice of reconstruction models and optimization algorithms significantly affects image quality and efficiency. The Berkeley Advanced Reconstruction Toolbox (BART) provides an open-source platform for implementing and testing advanced MRI reconstruction methods. However, a systematic CUDA-level analysis of how different optimizers affect memory behavior, kernel execution, and training efficiency in BART is still missing. In this study, we compare the Varnet and MoDL models implemented in BART using various optimizers to assess their reconstruction accuracy, convergence speed, and computational efficiency. When we trained under the same learning conditions, Adam converged faster and achieved higher reconstruction accuracy, whereas the optimizer, such as ADMM, SGD, showed task-specific advantages in constrained optimization settings. Overall, our profiling results indicate that while management and data movement functions executed by the CUDA runtime remain consistent across optimizers, including device memory initialization (`cudaMemset`) and host–device data transfers (`cudaMemcpyAsync`). These findings provide practical guidance for selecting optimizers that balance reconstruction quality, computational cost, and GPU resource utilization in MRI image reconstruction workflows.

Declaration on the use of ChatGPT and comparable tools in the context of examinations

In this work I have used ChatGPT or another AI as follows:

- ☐ Not at all
- ☑ During brainstorming
- ☐ When creating the outline
- ☑ To write individual passages, altogether to the extent of 10% of the entire text
- ☐ For the development of software source texts
- ☐ For optimizing or restructuring software source texts
- ☑ For proofreading or optimizing
- ☑ Further, namely: The use of AI-based tools for proofreading and improving the English language and clarity.

I hereby declare that I have stated all uses completely.
Missing or incorrect information will be considered as an attempt to cheat.

Acknowledgements

I would like to express my sincere gratitude to my supervisors, Prof. Dr. Julian Kunkel and Jonathan Decker, for their invaluable guidance, encouragement, and support throughout this work. I am also grateful to the GWDG for providing the necessary computing resources that made this research possible. Finally, I would like to thank my family and friends for their patience, understanding, and constant encouragement during the course of this study.

Contents

List of Tables

List of Figures

List of Listings

List of Abbreviations

HPC High-Performance Computing

BART Berkeley Advanced Reconstruction Toolbox

MRI Magnetic Resonance Imaging

ML Machine Learning

DL Deep Learning

VarNet Variational Network

MoDL Model-Based Deep Learning

NN Neural Network

VJP Vector–Jacobian Product

SGD Stochastic Gradient Descent

Adam Adaptive Moment Estimation

ADMM Alternating Direction Method of Multipliers

iPALM Inertial Proximal Alternating Linearized Minimization

PI Parallel Imaging

CS Compressed Sensing

PI+CS Parallel Imaging and Compressed Sensing

ESPIRiT Eigenvalue-based Self-consistent Parallel Imaging Reconstruction

SENSE Sensitivity Encoding

GRAPPA Generalized Autocalibrating Partially Parallel Acquisitions

NUFFT Non-Uniform Fast Fourier Transform

FFT Fast Fourier Transform

TV Total Variation

L1 ℓ_1 norm

CFL Complex Float Layout

HDR Header File

CLI Command-Line Interface

1 Introduction

This section starts with an overview of the study background and motivation in Section 1.1, outlining the main research questions. Section 1.2 then provides a detailed description of the experimental design and research objectives. In Section 1.3, we present the key contributions of this work to the field. Finally, Section 1.4 offers a brief overview of the overall structure of the thesis and summarizes the contents of the subsequent chapters.

1.1 Motivation

In recent years, the application of Magnetic Resonance Imaging (MRI) in medical research has brought significant advances to diagnostics, particularly by eliminating exposure to potentially harmful ionizing radiation. As costs continue to decrease and accessibility improves, MRI is becoming increasingly widespread in clinical practice.

The BART framework, as described in [SM97], encompasses a versatile set of calibration and reconstruction methods and facilitates advanced applications in parallel imaging, while compressed sensing techniques have been integrated into BART as shown in [Gri+02]. Parallel imaging, which reconstructs images by combining signals from multiple receiver coils in k-space [LP10], has become a standard tool for accelerating MRI acquisitions. More recently, by introducing sparse regularization terms into the solution of the parallel imaging inverse problem, compressed sensing has been successfully combined with parallel imaging in several works [BUF07; Lia+09].

Deep learning is currently one of the most active research areas in machine learning and pattern recognition. In some work[KS15], success has been achieved across a wide range of applications, including computer vision, natural language processing, and large-scale data analysis. In previous work[Gro], deep learning frameworks have been implemented and integrated with BART, introducing concepts such as automatic differentiation, iterative training algorithms, and neural network architectures. The Variational Network (VarNet)[AMJ19], and MoDL[Zhu+18] models were implemented in BART in the work[Blu+22], and their performance was compared with the original TensorFlow implementations. Neural networks[Aba+16] are typically composed of fundamental computational modules such as fully connected layers, convolutional layers, and nonlinear activation functions. Automatic differentiation provides an efficient mechanism for computing the gradients of loss functions required by gradient-based optimization algorithms, including Stochastic Gradient Descent (SGD). During each training iteration, the weight parameters θ are updated based on their gradients, enabling convergence to an optimal solution. In[Lia+09], a variety of optimization strategies were implemented, achieving computational performance comparable to that of established deep learning frameworks. Moreover, distributing the computational workload across multiple GPUs further reduces training time and facilitates the incorporation of novel optimization algorithms.

Despite these advances, the computational efficiency of deep-learning–based MRI reconstruction remains a practical challenge, especially when deploying unrolled networks on GPU-based systems. Existing research mainly emphasizes reconstruction quality, while aspects such as GPU memory usage, execution latency, and the impact of different optimization algorithms on computational performance are often less systematically studied.

Most current approaches rely on established deep learning frameworks, such as TensorFlow, together with standard optimizers like Adam. While effective, these solutions may introduce significant overhead in terms of memory consumption and runtime, particularly for complex models such as VarNet and MoDL, and provide limited insight into their low-level GPU behavior.

In this work, we address these limitations by analyzing deep-learning–based MRI reconstruction within the BART framework. By evaluating multiple optimization algorithms and performing CUDA-level performance profiling, we aim to better understand the trade-offs between reconstruction performance and computational efficiency on GPU-based high-performance computing systems.

1.2 Research Problem

We performed an in-depth analysis and comparison of CUDA memory behavior, execution latency, and kernel utilization to assess the differences in GPU computational performance among several optimization algorithms, such as SGD (Stochastic Gradient Descent), Adam, and ADMM (Alternating Direction Method of Multipliers) utilizing the VarNet model within the BART toolbox for MRI image processing. The experiments focused on assessing each optimizer's performance in terms of CUDA memory usage, memory transfer latency, function call distribution, and overall execution time.

We used CUDA to measure and compare the performance of the MoDL and VarNet models implemented in BART with that of the traditional TensorFlow framework. To systematically investigate the impact of different optimization strategies on deep-learning–based MRI reconstruction, we formulate the following research questions:

- **RQ1:** What trade-offs exist between computational efficiency (e.g., runtime and memory usage) and the quality of the reconstructed images?
- **RQ2:** How do optimizers such as SGD, Adam, and ADMM influence GPU memory usage, data transfer behavior, and overall memory performance within the BART-ML framework?

RQ3 and RQ4 extend the primary research questions by focusing on optimizer-specific and CUDA-level execution behavior, and are addressed in detail in the methodology and results chapters.

1.3 Contribution

The primary contributions of this thesis are outlined below:

- Two state-of-the-art MRI reconstruction networks, VarNet and MoDL, were implemented and benchmarked within the BART framework on a high-performance computing (HPC) system. The study involved adapting the original model architectures, configuring training and inference pipelines, and validating that the resulting implementations achieved reconstruction quality and runtime performance comparable to established TensorFlow-based baselines.
- Performed a comprehensive GPU performance evaluation using CUDA 11 and NVIDIA Nsight profiling tools. We systematically measured training time, inference latency, peak GPU memory consumption, computational cost, arithmetic intensity (FLOPs/Byte), and memory throughput to characterize the execution behavior of deep-learning-based MRI reconstruction workloads.

- Designed and executed an optimizer-level GPU performance study by integrating multiple optimization algorithms—SGD, Adam, iPALM, and ADMM—into the BART framework. Through controlled experiments, we analyzed their effects on GPU memory behavior, execution latency, and kernel utilization, identifying performance bottlenecks and trade-offs related to memory transfers and kernel execution.
- Conducted a detailed analysis of the CUDA runtime and driver API functions invoked during BART-ML execution. We categorized these API calls and examined how different optimization algorithms interact with them, linking low-level CUDA behavior to observed performance characteristics.
- Derived practical insights and optimization guidelines for efficient GPU-based MRI reconstruction. Based on empirical results, we provide recommendations for improving training efficiency, memory utilization, and overall runtime performance, offering actionable guidance for future high-performance imaging frameworks.

1.4 Outline

In this thesis, Section 2 introduces the background, fundamental concepts, and terminology related to the research topic. Section 3 provides a detailed review of previous studies, discussing their major breakthroughs as well as the remaining challenges. Section 4 presents the methodology of this work, outlining the experimental procedures and basic configurations. In Section 5, we present and analyze the experimental results, followed by a discussion of their significance and underlying causes in Section 6. Finally, Section 7 concludes the thesis by summarizing potential future developments in this research area and the challenges that may arise.

Section Summary

This section provides an overview of the research motivation, research questions, and main contributions. This work mainly focuses on implementing and testing different optimization algorithms, including SGD, Adam, iPALM, and ADMM, within the BART framework, and studying how they affect GPU memory usage, execution time, and kernel performance.

2 Background

This section presents the fundamental ideas and terminology that are the basis of this research. These fundamentals are followed up with implementation and performance analysis covered in later chapters. In Section 2.1, we describe the design of the MoDL[Ham+17] and VarNet[AMJ19] models—two popular deep learning models for MRI image reconstruction (in particular, their representation, architecture, and operating principles) and their structural design and the architectural/operational approach to be taken regarding BART methodology. Section 2.2 presents an overview of several optimization algorithms, including SGD, Adam[KB17], iPALM[PS16], and ADMM[Boy+11] [Wan+19], and how they operate at their core mechanisms, their respective strengths, and, most importantly, their limitations with respect to model training and convergence. Section 2.3 presents the basic components of the BART toolbox, with its core features as well as popular command-line tools and design logic that support the computational base for the experiments in this study. Lastly, in Section 2.4 we describe what high-performance computing (HPC) is, explain the role of parallelism in HPC, and discuss how parallel computation resources, GPUs, and containerized environments are used to enable large experiments, followed by a short description of the system setup.

2.1 Introduction to the Applied Model Network

In this subsection, we first describe the structures of the two networks, VarNet [Ham+17] and MoDL [AMJ19], which are widely used unrolled models for MRI reconstruction. The implementations of the Variational Network (VarNet)[1] and the MoDL model[2] used in this work are based on publicly available open-source code, ensuring transparency and reproducibility. Both models are built upon the idea of reformulating an iterative optimization algorithm as a finite sequence of learnable network layers, enabling the reconstruction of the solution to the following inverse problem:

$$u^* = \arg\min_u \|Au - y\|_2^2 + R(u),$$

Where $A = PFC$ represents the linear SENSE operator, consisting of multiplication with coil sensitivity maps (C), a Fourier transform (F), and a projection onto the sampling pattern (P). Here, u denotes the MR image to be reconstructed, y is the measured k-space data, and $R(u)$ is a regularization term.

From an MRI perspective, this formulation describes the task of reconstructing an image from incomplete and noisy k-space measurements, which commonly arise in accelerated MRI acquisitions. The operator A models the physical image acquisition process: the image is first weighted by the coil sensitivity maps, transformed into k-space via the Fourier transform, and then sampled according to the acquisition pattern. Because only a subset of k-space is measured, directly inverting this process leads to severe artifacts and noise amplification.

The first term $\|Au - y\|_2^2$ enforces data consistency, ensuring that the reconstructed image remains faithful to the measured data. However, data consistency alone is not sufficient to recover a meaningful image from undersampled measurements.

[1] `https://github.com/VLOGroup/mri-variationalnetwork` , Commit: 4b6855f
[2] `https://github.com/hkaggarwal/modl`, Commit: 428ef84

The regularization term $R(u)$, therefore, introduces additional constraints that reflect prior knowledge about realistic MR images, such as structural coherence and the absence of strong aliasing artifacts.

Overall, the optimization problem seeks a balance between explaining the measured data and enforcing image properties that are known to hold for MR images. Deep-learning–based reconstruction methods such as VarNet and MoDL can be interpreted as learning effective forms of the regularization term $R(u)$ from data, while still following the same underlying reconstruction principle.

2.1.1 VarNet

For the MRI reconstruction problem, the trainable gradient descent algorithm with time-varying parameters $u^0 = A^H y$ K_i^t, $\Phi_i'^t$, and λ^t can be reformulated as follows:

$$u^{t+1} = u^t - \sum_{i=1}^{N_k} (K_i^t)^\top \Phi_i'^t (K_i^t u^t) - \lambda^t A^* (A u^t - u^0), \quad 0 \le t \le T - 1.$$

Here, A and A^* denote the forward and adjoint operators, respectively; K_i^t , represents the convolutional filters λ^t controls the contribution of the data consistency term at each iteration step.

$$\Phi_i'(z) = \operatorname{diag}\left(\phi_i'(z_1), \ldots, \phi_i'(z_N)\right)$$

denotes the activation function defined as the first derivative of the potential function Φ_i. It should be noted that the transpose operation $(K_i)^\top$ can be implemented as a convolution with the corresponding filter kernel k_i rotated by 180°.

The sensitivity maps are used in the operators A and A^*, which perform a sensitivity-weighted image combination and can also implement additional processing steps, such as removing readout oversampling. While both the raw data and the operators A and A^* are required in each iteration of the VN to compute the gradient of the data term, the gradient of the regularization term is applied only in the image domain. By applying the adjoint operator A^*, a zero-filled solution is computed from the undersampled k-space data. The adjoint operator A^* involves the application of coil sensitivity maps. Feed the undersampled k-space data, the coil sensitivity maps, and the zero-filled solution into the variational network (VN) to obtain the final reconstruction result. The VN evolves from u_0 to u_T through T gradient descent steps. The training process aims to minimize a loss function over the dataset S with respect to the parameters θ. This loss function quantifies the similarity between the reconstructed image u_T and the corresponding clean, artifact-free reference image g. A common choice for this purpose is the **mean-squared error (MSE)**, defined as:

$$L(\theta) = \min_\theta \frac{1}{2S} \sum_{s=1}^{S} \|u_T^s(\theta) - g_s\|_2^2.$$

2.1.2 MoDL

The MoDL [AMJ19] network is another type of unrolled optimization model, initialized with $u_0 = A^H y$. A residual network D_W is used to denoise the current reconstruction, while data consistency is enforced through a proximal mapping. The update step for each iteration is defined as:

$$u^{t+1} = \arg\min_{u} \left(\|Au - y\|_2^2 + \lambda \|u - D_W(u^t)\|_2^2 \right),$$

which yields the closed-form solution:

$$u^{t+1} = (A^H A + \lambda I)^{-1}(A^H y + \lambda D_W(u^t)), \quad 0 \leq t \leq T - 1.$$

Here, $\mathcal{D}_W(x)$ represents the denoised version of x, in which aliasing artifacts and noise are suppressed. A CNN-based prior $\mathcal{N}_W(x)^2$ assigns higher values when x is corrupted by noise and aliasing patterns, thereby promoting solutions that are consistent with the acquired data while minimizing the influence of such degradations. In this formulation, λ serves as a trainable regularization parameter. In the BART Toolbox, they keep the Adam optimizer by default and the same hyperparameters as in the TensorFlow codebase.

We unroll the network for $T = 10$ iterations. The weights are shared across iterations. Each residual block has $L = 5$ convolution layers, and each convolution has $F_c = 32$ complex filters (instead of 64 real filters in the TensorFlow version). So the model has 28,364 complex parameters (= 56,728 real parameters). The TensorFlow version has 112,001 real parameters.

To compute $Q(u)$, in the BART Toolbox, they implement a generic linear operator $S_\lambda : \mathbb{C}^N \to \mathbb{C}^N$. It is positive-definite and self-adjoint, and it is parameterized by $\lambda \in \mathbb{C}^M$. Given an operator $S(\cdot, \lambda)$ that applies S_λ to its first input, we define another operator S^{-1} that applies S_λ^{-1}. In formulas:

$$(u, \lambda) \mapsto y = S_\lambda u \qquad (y, \lambda) \mapsto u = S_\lambda^{-1} y.$$

The gradients of S^{-1} can be written using S and its gradients. In particular:

$$D_y S^{-1}\, dy \;=\; S_\lambda^{-1} dy$$

and

$$D_\lambda S^{-1}\, d\lambda \;=\; -S_\lambda^{-1}\Big(D_\lambda S\big|_{S_\lambda^{-1}(y,\lambda),\lambda} d\lambda \Big).$$

2.2 Operator Framework

Neural networks are typically trained using gradient-based optimization. A basic gradient descent update minimizes by iterating

$$\theta_{i+1} = \theta_i - \eta \nabla_\theta \sum_i L\big(y_i, F(x_i; \theta)\big),$$

where η is the learning rate and ∇_θ is the gradient of the loss with respect to the parameters θ. Automatic differentiation lets us implement the operator $\sum_i L\big(y_i, F(x_i;\theta)\big)$ and automatically compute its gradient. The core idea of automatic differentiation is that we can chain `nlops` and the derivatives automatically follow the chain. If

$$H = G \circ F$$

then

$$DH|_x = DG|_{F(x)} \circ DF|_x.$$

[2] `https://github.com/VLOGroup/mri-variationalnetwork`, Commit: 4b6855f

[3] `https://github.com/hkaggarwal/modl`, Commit: 428ef84

When we call G on $F(x)$, the derivative DG is automatically evaluated at the correct point $F(x)$. For the transpose (used during backprop):

$$DH^T|_x = DF^T|_x \circ DG^T|_{F(x)}.$$

Here, the order of the operators flips — this is exactly what backpropagation means.

From an optimizer's perspective, the primitive object is not the Jacobian as a matrix, but the vector–Jacobian product(VJP). Modern optimizers (SGD, Adam) do not form Jacobians explicitly. They only require that, for any scalar loss $F(x) \in \mathbb{R}$, one can obtain $\nabla F(x)$ by a single reverse application of an adjoint linear map.

In the nlop framework, every nonlinear operator F is therefore defined by: (i) a forward evaluation $y = F(x)$, and (ii) its adjoint derivative $DF(x)^T$, i.e. a function that maps a test vector v to the VJP $DF(x)^T(v)$.

The classical formula

$$x \mapsto y = F(x), \qquad dx \mapsto dy = J|_x\, dx$$

is not instantiated as an explicit Jacobian $J|_x$ in memory. Instead, (3) is realized by providing a procedure that, given v, computes $J|_x^T v$.

Consequently, gradients of scalar functions follow directly:

$$\nabla F(x) \;=\; DF(x)^T(1)$$

i.e., seeding the reverse pass with the scalar 1 yields the entire gradient vector in one application of the adjoint derivative. This is precisely the interface expected by optimizers: A single VJP call per loss evaluation.

BART ships helper functions to combine `nlops` with multiple inputs and outputs. We can merge two `nlops` into one, connect an output back into an input, or duplicate an input. For example, given two `nlops` F and G, we can construct

$$J(x_1, x_2) = F\big(x_1,\, G(x_1, x_2)\big).$$

The resulting `nlop` keeps references to the two base `nlops`, and all required derivatives are built automatically. Since `combine`, `duplicate`, and `link` can be nested arbitrarily, very complex computational graphs can be composed.

An `nlop` has one operator for the forward pass, and a set of linear operators (`linops`) implementing the derivatives w.r.t. each input. For atomic (non-composed) `nlops`, all of these share the same `nlop_data_s` struct, which stores the information from the last forward call because the derivative needs it.

To implement a new `nlop`, it is sufficient to define the corresponding `nlop_data_s` structure, implement the forward `apply` function, and provide the associated derivative and adjoint derivative `apply` functions. All auxiliary metadata and glue code required for integration are generated automatically by the framework. Shared data objects are managed through reference counting, which enables automatic memory management in a manner similar to garbage collection. Furthermore, since `nlops` are built on top of `md`-functions, any `nlop` whose inputs reside on the GPU is executed natively on the GPU without requiring additional implementation effort.

BART natively supports complex-valued computations using single-precision complex floating-point arithmetic. Automatic differentiation is extended to complex variables by representing the complex field $\mathbb{C}$ as $\mathbb{R}^2$. Consequently, a complex-valued mapping F : $x \mapsto y = F(x)$ can be expressed in terms of its real and imaginary components as

$$F : \begin{bmatrix} x_r \\ x_i \end{bmatrix} \mapsto \begin{bmatrix} y_r \\ y_i \end{bmatrix}, \qquad DF : \begin{bmatrix} dx_r \\ dx_i \end{bmatrix} \mapsto \begin{bmatrix} \frac{\partial y_r}{\partial x_r} & \frac{\partial y_r}{\partial x_i} \\ \frac{\partial y_i}{\partial x_r} & \frac{\partial y_i}{\partial x_i} \end{bmatrix} \begin{bmatrix} dx_r \\ dx_i \end{bmatrix}. \tag{5}$$

2.3 BART Functions and CUDA Profiling Tools

BART (the **Berkeley Advanced Reconstruction Toolbox**) [BAR]is an open-source software system for MRI reconstruction. It targets parallel imaging, compressed sensing, non-Cartesian trajectories, and—more recently—operator-based deep learning workflows.

BART consists of two layers: a low-level library that provides core functionality for multi-dimensional array handling, FFT and NUFFT operations, wavelet transforms, iterative solvers, proximal operators, and linear operators. On top of this library, BART also offers a high-level command-line toolkit that exposes these capabilities through executable commands, enabling flexible and scriptable workflows for MRI reconstruction and related signal processing tasks.

Major functional capabilities. BART supports:

- multi-dimensional transforms (FFT, NUFFT, wavelets)
- iterative reconstruction algorithms (CG, ISTA/FISTA, ADMM, IRGNM, iPalm, Adam)
- calibration for parallel MRI (incl. ESPIRiT)
- standard and advanced reconstructions (PI, CS, PI+CS, structured low-rank)
- a range of regularizers (Tikhonov, TV, ℓ_1 wavelets, low-rank)
- CPU/GPU execution using the same interface

Representative commands. Several CLI tools are most commonly used:

- `ecalib`: calibration of coil sensitivity maps

  ```
  bart ecalib kspace sensitivities
  ```

- `pics`: PI+CS reconstruction

  ```
  bart pics -l1 -r0.001 kspace sensitivities image_out
  ```

- `nufft`: non-Cartesian NUFFT

  ```
  bart nufft trajectory kspace image_out
  ```

- `bartview.py`: Python viewer for `.cfl`+`.hdr`

  ```
  python bartview.py image_out
  ```

Practical BART uses its own pair of files (`.hdr` + `.cfl`) for complex multi-dimensional arrays. GPU acceleration is available if BART is compiled with CUDA support. BART commands may be called directly from MATLAB/Octave or Python, which is convenient for experiments and pipelines.

BART provides a unified operator-based framework for MRI reconstruction. Users may directly call its existing commands for production-grade reconstructions, or Use the underlying operator library to build new research algorithms. Understanding the CLI commands and array format significantly accelerates MRI method development.

2.3.1 CUDA Profiling Tools

We used profiling tools from the NVIDIA Nsight suite[3] to analyze CUDA activity:

[3]NVIDIA Nsight Systems: `https://developer.nvidia.com/nsight-systems`

- **Nsight Systems** Captures kernel scheduling, stream behavior, and CPU–GPU interactions.
- **Nsight Compute** Provides kernel-level statistics such as memory throughput and arithmetic intensity.
- **CUDA Memory Operation Logs** Records the volume and latency of `cudaMemcpy`, `cudaMemset`, and device-to-device transfers.

These tools provide detailed insights into how different optimization algorithms (SGD, Adam, iPALM, ADMM) utilize GPU compute kernels and memory subsystems.

2.4 HPC Systems and the Grete Supercomputer

High-performance computing (HPC)[4] refers to the execution of large-scale numerical workloads on tightly interconnected clusters of compute nodes. Modern HPC systems combine thousands of CPU cores and hundreds of GPUs, and rely on high-bandwidth low-latency interconnects in order to run distributed algorithms efficiently, e.g., large PDE solvers, large-scale inverse problems, or large foundation-model training. Performance depends not only on peak FLOPS, but also on memory bandwidth, topology of the network fabric, and the global parallel I/O subsystem.

In this work, we use the Grete supercomputer hosted at GWDG, Göttingen. Grete is a GPU-accelerated HPC system with a multi-rack, Fat-Tree network fabric.[5] In total, Grete consists of 8 racks and 106 compute nodes. Three nodes (Phase 1) are equipped with Intel Xeon Gold 6148 CPUs and NVIDIA V100 GPUs (32 GB). The remaining 103 nodes (Phase 2) use AMD EPYC processors (Zen2 EPYC 7662 or Zen3 EPYC 7513) and NVIDIA A100 GPUs (40 GB or 80 GB).

Overall Grete provides 6,840 CPU cores, 49.7 TB of system RAM, and 27.5 TB of GPU device memory. The interconnect uses InfiniBand with 100 Gbit/s links (Phase 1) and dual 200 Gbit/s links (Phase 2), arranged as an 8:1 blocking Fat-Tree topology (16 nodes per leaf), enabling GPU–GPU data transfers without host memory involvement.

The system sustained 5.46 PetaFlop/s in LINPACK, ranking #142 in the TOP500 list[6] and #16 in the GREEN500 list [7] (November 2023). At deployment, Grete was the most energy-efficient supercomputer in Germany.

Grete exposes 8.4 PiB of global parallel storage: 340 TiB GPFS for home directories and 8.1 PiB Lustre for scratch/work. An additional 6.4 PiB tape archive system is available for long-term data retention (with a 120 TiB disk cache).

Further architectural details are documented in the official Grete HPC website.[8]

Section Summary

This chapter introduced the theoretical and practical background required for the subsequent performance analysis and experimental evaluation. We first reviewed two representative

[4] `https://www.intel.com/content/www/us/en/high-performance-computing/what-is-hpc.html`, visited on Maybe 10, 2025

[5] Official documentation: `https://gwdg.de/hpc/systems/grete/`, visited on May 11, 2025

[6] TOP500 List, November 2023, visited on May 11, 2025

[7] GREEN500 List, November 2023, visited on May 11, 2025

[8] Official documentation: `https://gwdg.de/hpc/systems/grete/`

unrolled deep-learning models for MRI reconstruction, VarNet and MoDL, and formulated their common reconstruction objective as a regularized inverse problem. This formulation provides an intuitive interpretation of how data consistency and learned priors are combined to recover high-quality images from undersampled k-space data.

We then described the operator-based automatic differentiation framework used in BART, highlighting how nonlinear operators (nlops) and their adjoint derivatives enable efficient gradient-based optimization without explicitly forming Jacobians. This abstraction is central to integrating modern optimization algorithms such as SGD, Adam, iPALM, and ADMM into MRI reconstruction pipelines while maintaining GPU efficiency and modularity.

In addition, we outlined the core functionality of the BART toolbox, including its command-line interface, internal data representation, and native support for GPU execution. The CUDA profiling tools employed in this work were also introduced, providing the methodological basis for analyzing kernel execution, memory transfers, and arithmetic intensity.

Finally, we presented the high-performance computing environment used for the experiments, including the Grete supercomputer and its hardware configuration. Together, these components establish the conceptual, algorithmic, and system-level foundation for the experimental results and performance analyses presented in the following chapters.

3 Related Work

As machine learning continues to evolve, a wide range of optimization methods has been proposed to improve model performance. In this section, we review the literature most relevant to our study. In Section 3.1 we examine the use of neural networks for MRI reconstruction. Section 3.2 summarizes prior work that refined these models and compared them against earlier baselines. Finally, Section 3.3 surveys several variants of gradient-based optimization, and outlines the advantages and limitations of these frameworks from a practical perspective.

3.1 MRI in NN

Neural networks have been integrated into MRI reconstruction in multiple styles. One major direction is to learn a direct end-to-end mapping from undersampled k-space to the final image [Zhu+18]. These models implement and can produce sharp images, but the drawback is that the mapping does not explicitly enforce any physics. As a result, the reconstructed image may not fully agree with the measured k-space, the network might produce something that looks anatomically plausible but is not strictly consistent with the data [Zhu+18].

Another design philosophy keeps the classical reconstruction algorithm in the loop and only uses the neural network to model prior knowledge about what MR images typically look like [Luo+20]. This can be done by feeding an initial rough reconstruction to a neural network, producing an improved image, and then using the squared difference between the two as a learned regularization penalty [Yan+18]. In the setup of [Kof+20], the learned component acts as a prior term, while the physical forward model still enforces the k-space consistency.

More recent developments tighten this coupling even further by unrolling an iterative algorithm and replacing some algorithmic steps with neural networks [Ham+17; AMJ19; Sch+18]. In this formulation, each iteration has a learned update block (often a CNN or U-Net) and then a data-consistency block (usually implemented as a gradient or proximal update). After a fixed number of unrolled iterations, the whole pipeline becomes a finite feed-forward network that is trained end-to-end. This style preserves the physical consistency of the reconstruction while still allowing powerful learned regularizers inside each iteration.

3.2 Analysis of GPU Memory Usage

Table 1 reproduced from [Blu+22] compares the GPU memory consumption of the BART and TensorFlow frameworks at training and inference time for MoDL[AMJ19] and VarNet[Ham+17] models on various GPU architectures. In general, the results indicate that for most cases, BART tends to use less GPU memory compared to TensorFlow at training, especially for the VarNet model.

In the MoDL network, BART has shown comparable or slightly lower memory usage across most GPUs. For instance, BART requires around 8 to 9 GB of memory while training on A100 and V100, whereas TensorFlow needs about 9 to 10 GB.

As expected, during inference, both frameworks require much less memory. MoDL inference uses less than 2 GB on most GPUs. There are minor differences between BART and TensorFlow.

MoDL				
GPU	**BART (Train)**	**TF (Train)**	**BART (Infer)**	**TF (Infer)**
A100	8.9 (9.6)	10.2 (5.8)	1.8 (2.2)	1.9 (1.4)
V100	8.6 (8.9)	9.6 (2.9)	1.6 (1.7)	1.4 (1.1)
TITAN Xp	11.8 (12.6)	9.2 (4.9)	1.7 (1.8)	4.0 (0.7)
TITAN X	8.3 (9.0)	9.2 (4.9)	1.1 (1.1)	4.0 (0.6)
VarNet				
GPU	**BART (Train)**	**TF (Train)**	**BART (Infer)**	**TF (Infer)**
A100	6.6 (6.8)	18.7 (10.2)	1.6 (2.0)	1.9 (1.7)
V100	6.2 (6.3)	18.2 (9.6)	1.3 (1.6)	1.4 (1.1)
TITAN Xp	6.3 (6.3)	12.4 (9.2)	1.2 (1.4)	1.1 (0.8)
TITAN X	5.8 (5.8)	12.4 (9.2)	1.0 (1.0)	1.0 (0.5)

Table 1: GPU memory consumption (GB) for MoDL and VarNet during training and inference using BART and TensorFlow. Peak memory usage is reported in parentheses. Results reproduced from [Blu+22].

This difference significantly widens in the case of the VarNet model. While BART consistently stays below 7 GB, TensorFlow exhibits much higher requirements and reaches up to 18 GB on the A100. Considering both models and both frameworks, it becomes evident that BART handles intermediate computations and feature maps more efficiently in its implementation. During inference, the same trend was observed: both frameworks were quite lightweight, But TensorFlow still uses a bit more memory.

These results indicate that BART presents a more memory-efficient solution for both MoDL and VarNet. Especially when deployed in large-scale or resource-constrained GPU environments. The much lower memory footprint makes BART an appealing option for high-performance MRI reconstruction tasks.

3.3 Gradient Descent Algorithm

Gradient descent algorithms are widely used optimization methods for solving learning and reconstruction problems by iteratively minimizing an objective function. In each iteration, model parameters are updated based on the gradient of the loss function with respect to these parameters. Different variants of gradient descent mainly differ in how the gradients are computed and how the update steps are controlled, which can affect convergence behavior, stability, and computational performance.

3.3.1 Stochastic Gradient Descent (SGD)

Stochastic Gradient Descent (SGD) is a variant of gradient descent where, instead of computing the gradient over the entire dataset, only a single sample is randomly selected for each parameter update. The update formula is similar to standard gradient descent; however, SGD selects samples randomly from the batch, while standard gradient descent computes the gradient using all input samples.

In this sense, Batch Gradient Descent (BGD) and SGD are two extremes. Since each parameter update in SGD only requires computing the gradient for one sample, the training speed is much faster. Even when dealing with large datasets, it is often sufficient to use only a subset of samples to iterate toward the optimal solution. However, because each iteration does not always move toward the global optimum, the gradient updates fluctuate significantly, which can help escape local minima but may reduce overall accuracy.

3.3.2 Alternating Direction Method of Multipliers (ADMM)

The Alternating Direction Method of Multipliers (ADMM) was introduced by Boyd et al. [Boy+11]. It is a computational framework for solving optimization problems and is particularly suitable for distributed convex optimization. ADMM follows a *decomposition–coordination* strategy, where a large global optimization problem is decomposed into several smaller and easier subproblems. These subproblems are solved alternately and coordinated through dual variables to obtain the solution of the original problem.

ADMM is closely related to the augmented Lagrangian method (ALM) in primal–dual optimization. Its standard form considers the following constrained optimization problem:

$$\begin{aligned} \min_{x,z} \quad & f(x) + g(z) \\ \text{s.t.} \quad & Ax + Bz = c, \end{aligned} \tag{1}$$

where $x \in \mathbb{R}^n$ and $z \in \mathbb{R}^m$ are optimization variables. The matrices $A \in \mathbb{R}^{p\times n}$ and $B \in \mathbb{R}^{p\times m}$, and the vector $c \in \mathbb{R}^p$ define the linear equality constraint. The functions $f(\cdot)$ and $g(\cdot)$ are assumed to be convex.

The augmented Lagrangian associated with this problem is defined as

$$L_\rho(x, z, u) = f(x) + g(z) + u^T(Ax + Bz - c) + \frac{\rho}{2}\|Ax + Bz - c\|_2^2, \tag{2}$$

where u denotes the dual variable and $\rho > 0$ is the penalty parameter. Compared with the standard Lagrangian, the augmented Lagrangian includes an additional quadratic penalty term, which improves convergence behavior.

The primal problem

$$\min_{x,z} f(x) + g(z) \tag{3}$$

is equivalent to the dual problem

$$\max_{u} \min_{x,z} L_\rho(x, z, u), \tag{4}$$

and both share the same optimal solution under mild conditions. By introducing the augmented Lagrangian, the constrained optimization problem is transformed into an unconstrained saddle-point problem.

ADMM solves this problem by alternating updates of the primal variables and the dual variable. In practice, this alternating scheme allows the original problem to be decomposed into simpler subproblems. In many applications, especially when ℓ_1-norm regularization is involved, variable splitting is employed so that the optimization reduces to a combination of ℓ_1 and ℓ_2 norm subproblems. Such subproblems can be efficiently solved using proximal algorithms.

3.3.3 Adaptive Moment Estimation (Adam)

Adam (Adaptive Moment Estimation) [KB17] is an optimization algorithm that combines the advantages of both first-order momentum methods and second-order methods like AdaGrad and AdaDelta, building upon Stochastic Gradient Descent (SGD) by incorporating adaptive estimates of both first and second moments of the gradients.

Algorithm Given the gradient at time step t, $g_t = \nabla_\theta J(\theta_t)$, Adam updates the moving averages of the first moment (mean) m_t and the second moment (uncentered variance) v_t as follows:

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t$$

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$$

To correct the bias introduced by initializing m_0 and v_0 to zero, Adam computes the bias-corrected estimates:

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t}, \quad \hat{v}_t = \frac{v_t}{1 - \beta_2^t}$$

Finally, the parameters are updated as:

$$\theta_{t+1} = \theta_t - \eta \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon}$$

Default Hyperparameters

The commonly used default values are:

$$\eta = 0.001, \quad \beta_1 = 0.9, \quad \beta_2 = 0.999, \quad \epsilon = 10^{-8}$$

Adam is generally considered robust to the choice of hyperparameters. This would mean that after bias correction, the effective learning rate at each iteration remains in a stable and reasonable range, which helps to keep the parameter updates consistent. It combines the benefits brought about by Adagrad on sparse gradients and those brought about by RMSProp on non-stationary objectives. Going further, it computes adaptive learning rates for different parameters, hence being well adapted to non-convex optimization problems, large datasets, and high-dimensional parameter spaces.

Section Summary

This chapter reviewed related work on deep-learning-based MRI reconstruction, GPU performance analysis, and optimization algorithms. We first discussed different neural network approaches for MRI reconstruction, with a focus on unrolled optimization models such as VarNet and MoDL, which combine physical data consistency with learned image priors.

We then summarized prior studies comparing the BART and TensorFlow frameworks, showing that BART often achieves lower GPU memory usage while maintaining competitive reconstruction performance. Finally, we reviewed common gradient-based optimization algorithms, including SGD, ADMM, and Adam, and outlined their update mechanisms and practical trade-offs.

4 Methodology

The following section describes the methodological framework that was used in comparing the inference systems assessed in this thesis. We begin in Section 4.1 with a formal statement of the research questions that guide our study. Section 4.2 describes the Neural Network Library and the TensorFlow baseline, respectively, representing the foundation for our comparative analysis. Section 4.3 describes the machine learning models used in the experiments. In Section 4.4, we explain how we ensure the reproducibility and reliability of our experimental procedures. Section 4.5 discusses in detail the hardware environment, CUDA profiling tools, and the metrics used to evaluate computational performance and reconstruction quality. Finally, in Section 4.6 describe the experimental setup of the hardware environment, CUDA profiling tools, and the evaluation metrics used to assess both computational performance and reconstruction quality.

4.1 Research Questions

Our goal is not only to implement several optimizers, including SGD, Adam, iPALM, and ADMM[Boy+11], but also to evaluate how they affect GPU memory usage, execution latency, and kernel-level behavior when training unrolled models like MoDL and VarNet.To accomplish this, we first define the reconstruction problem and outline the network architectures considered in this work. We then introduce the optimization algorithms and explain how each is integrated into the BART machine-learning module.

- **RQ1: How well do advanced MRI reconstruction networks—VarNet and MoDL—perform within the BART framework compared to their TensorFlow-based counterparts?** This question examines model-level performance in an HPC environment, including reconstruction quality and computational efficiency.
- **RQ2: What are the GPU-level performance characteristics of BART when running modern deep-learning–based reconstruction models?** Here, we investigate CUDA-level behavior using CUDA 11.2 and Nsight profiling tools, focusing on training and inference time, memory usage, arithmetic intensity, and memory throughput.
- **RQ3: How do different optimization algorithms (SGD, Adam, iPALM, ADMM) behave on the GPU when implemented inside the BART framework?**
- **RQ4: Which CUDA runtime and driver API calls are involved in the course of BART-ML execution, and how do they explain observed performance characteristics?**

  We categorize and analyze the CUDA API calls invoked by BART-ML and evaluate how different optimizers interact with these operations.

4.2 Deep Learning Libraries in BART

This subsection introduces how deep learning components are integrated within the BART [9] framework. At the lowest level, BART provides a numerical backend for efficient multidimensional array operations. Built on top of this backend, the non-linear operator (nlop) framework makes it possible to assemble neural networks from modular building blocks and supports automatic differentiation. The nn-library extends these tools with additional deep-learning–specific functions, while gradient-based training algorithms are integrated into BART's iterative optimization framework.

4.2.1 Numerical Backend

The numerical backend operates on multidimensional (md) arrays, which are defined by their dimensions, rank, and optional memory strides. By adjusting these strides, BART can create different memory views without copying data. Arrays may reside on either the CPU or the GPU.

A set of `md-functions` provides a unified interface for performing computations on md-arrays. Many common operations—such as convolutions, dot products, and matrix multiplications can be implemented simply by configuring the appropriate strides. When array data is stored on the GPU, these functions automatically execute their computations on the GPU. The backend can also detect specific stride patterns and call optimized libraries such as cuDNN to speed up execution.

4.2.2 Automatic Differentiation and the Non-linear Operator Framework

Training neural networks requires computing gradients of a loss function with respect to model parameters. BART [10] enables this through its nlop framework, which provides automatic differentiation for complex operator chains. The framework can also serve as a wrapper for external deep learning models. For example, BART includes a TensorFlow[Aba+16] wrapper that uses the TensorFlow C API to import pretrained computation graphs. When such an nlop is evaluated, BART runs the TensorFlow forward pass and uses TensorFlow's internal mechanisms to compute adjoint gradients.

4.2.3 Iterative Training Algorithms

Neural network training can be viewed as minimizing a loss function with respect to model parameters. BART constructs a loss operator by combining the model nlop with a loss nlop, both of which take training data and network parameters as inputs. During training, the data is divided into mini-batches, and model parameters are updated using gradients computed by automatic differentiation. In our work, we evaluate several gradient-based optimization algorithms integrated into BART's iterative framework, including SGD, Adam, and iPALM. These optimizers enable efficient training of deep learning models within the BART environment.

[9] GitHub – mrirecon/bart (visited on 20/06/2024).

[10] BART: Berkeley Advanced Reconstruction Toolbox (visited on 20/06/2024).

4.3 VarNet and MoDL Implementation in BART

This subsection describes how the VarNet and MoDL reconstruction networks are implemented within the BART framework. We focus on practical implementation aspects and how BART's operator-based design supports training and inference. The following subsubsections introduce the two models separately.

4.3.1 Varnet

The BART implementation of VarNet[11] can be trained and applied using the `reconet` command. The basic usage is shown below:

```
bart reconet --network=varnet --train <kspace> <coils> <weights> <reference>

bart reconet --network=varnet --apply <kspace> <coils> <weights> <output>
```

The inputs `<kspace>`, `<coils>`, and `<reference>` are files containing multidimensional arrays are used either for training or for inference. Their layout follows the BART data format, in which independent datasets or volumes are stacked along batch dimension 15.

The file `<weights>` stores the network parameters θ. This file is produced during training and is required when applying the trained model. Network parameters such as the number of unrolled iterations, loss function, weight initialization, and the training algorithm can be configured via command-line options.

By default, the hyperparameters follow the TensorFlow implementation. Specifically, VarNet is unrolled for $T = 10$ iterations, uses $N_k = 24$ convolutional filters of size 11×11, and employs $N_w = 31$ Gaussian radial basis functions to parameterize the activation functions Φ_0. In total, this configuration results in 65,530 real-valued trainable parameters.

In our experiments, we trained VarNet using the iPALM optimizer and also imported TensorFlow-pretrained weights to enable direct comparison between the two frameworks.

4.3.2 MoDL Implementation in BART

MoDL can be trained in BART using the `reconet` command with the option `-network=modl`. The basic usage is shown below:

```
bart reconet --network=modl --train <kspace> <coils> <weights> <reference>

bart reconet --network=modl --apply <kspace> <coils> <weights> <output>
```

The input files `<kspace>`, `<coils>`, and `<reference>` contain multidimensional arrays used for training or inference, following the BART data layout where samples are stacked along batch dimension 15. The file `<weights>` stores the network parameters θ and is produced during training and reused during inference.

By default, MoDL[12] training uses the Adam optimizer together with the hyperparameters from the TensorFlow implementation[3]. The network is unrolled for $T = 10$ iterations

[11] K. Hammernik, T. Klatzer, E. Kobler, et al., "Learning a variational network for reconstruction of accelerated MRI data," *Magnetic Resonance in Medicine*, vol. 79, no. 6, pp. 3055–3071, 2017.

[12] H. K. Aggarwal, M. P. Mani, and M. Jacob, "MoDL: Model-Based Deep Learning Architecture for Inverse Problems," *IEEE Transactions on Medical Imaging*, vol. 38, no. 2, pp. 394–405, 2019.

with shared weights. Each residual block contains $L = 5$ convolutional layers, and each layer uses $F_c = 32$ complex-valued filters, instead of the $F_r = 64$ real-valued filters used in the TensorFlow version.

Similar to VarNet, the MoDL network in BART updates only the image that corresponds to the first set of coil sensitivity maps within the network block.

4.4 Ensuring Reproducibility and Reliability

To make sure that our experiments can be repeated and that the results are reliable, we followed a consistent setup for data preparation, model execution, and performance evaluation.

Training Data. We used the public datasets that were released with the original VarNet and MoDL implementations. For VarNet, we used the dataset from Zenodo [13], and for MoDL, we used the dataset available at [14] Using the same datasets as the original work makes it easier to reproduce our experiments and compare results. All data were converted into BART's multidimensional array format and stacked along the batch dimension.

Experimental Environment. All experiments were run on the same HPC system with a fixed software setup. We used the same CUDA version, NVIDIA driver, and BART version across all runs. Hardware settings such as GPU model, memory size, and batch size were kept constant to avoid unintended changes during the experiments.

Deterministic Execution. We used BART's `reconet` command for both VarNet and MoDL. This command provides a stable and predictable way to run training for both networks.

Consistent Hyperparameters. The network hyperparameters were matched to the values used in the original TensorFlow implementations. This includes the number of unrolled iterations, the convolutional filter settings, the structure of the residual blocks, and the optimizer configuration. Keeping these settings fixed ensures that the comparison between BART and TensorFlow is fair and reproducible.

Profiling and Verification. We used NVIDIA Nsight tools to record detailed CUDA profiling information, including kernel execution time, memory transfers, and GPU memory usage. Each experiment was run multiple times to check the stability of the measurements. This helped us confirm that the reported latency, throughput, and memory usage values are consistent.

Overall, these steps provide a clear and repeatable workflow, ensuring that the results in this study can be verified and reproduced.

4.5 Evaluation Metrics

To evaluate the computational performance of the inference framework and the quality of the reconstructed MRI images, we employed a set of widely used quantitative metrics. These metrics allow us to evaluate runtime, memory efficiency, and reconstruction accuracy in a consistent and reproducible manner.

[13] `https://zenodo.org/records/6482961`, accessed on March 8, 2025.
[14] `https://zenodo.org/records/6481291`, accessed on March 8, 2025.

Reconstruction Quality Metrics

We used standard quantitative metrics to evaluate the output images of MoDL and VarNet, which capture image fidelity and structural consistency:

- **PSNR (Peak Signal-to-Noise Ratio)**

  PSNR measures the difference between the reconstructed image and the fully sampled reference image. A higher PSNR value indicates that the reconstruction result is closer to the true value, meaning a smaller reconstruction error.

- **SSIM (Structural Similarity Index)**

  SSIM evaluates structural similarity by taking into account brightness, contrast, and texture. It is widely used in MRI reconstruction because it is closely related to perceived image quality. The closer the value is to 1, the higher the structural similarity.

- **NMSE (Normalized Mean Squared Error)**

  NMSE is the relative reconstruction error after normalizing the quantization intensity. A lower NMSE value indicates higher reconstruction accuracy.

  These metrics are computed layer-by-layer using fully sampled reference images provided by the VarNet and MoDL datasets. To maintain consistency, complex-valued images are converted to amplitude images before evaluation.

4.6 Experimental Setup

This section describes the experimental setup used to evaluate the computational performance of the BART and TensorFlow implementations. We report the HPC environment, hardware configuration, CUDA profiling tools, and the performance metrics used in our analysis.

HPC System Configuration

All experiments were conducted on the GWDG high-performance computing (HPC) cluster, which provides access to several GPU nodes. In this study, we used two nodes with the following configurations:

Component	**A100 Node**	**V100 Node**
GPU Model	NVIDIA A100-SXM4-80GB	NVIDIA Tesla V100S-PCIe-32GB
GPU Memory	80 GB HBM2e	32 GB HBM2
Memory Bandwidth	2,039 GB/s	1,134 GB/s
CPU	Dual AMD EPYC processors	Dual Intel Xeon Gold processors
System Memory	512 GB DDR4	384 GB DDR4
CUDA Version	11.2	11.2

Table 2: Hardware configuration of the A100 and V100 computing nodes.

Both nodes run a Linux-based environment managed through environment modules. For compiling and running BART, the `cuda/11.2` and `gcc/9.3` modules were loaded, together with the `openblas`, `fftw`, and `lapack` libraries.
All jobs were submitted via the SLURM scheduler, and GPU temperature, power draw, and utilization were monitored throughout model execution. To ensure consistent measurements, each job used exclusive access to a single GPU.

Computational Performance Metrics

To evaluate computational performance, we considered the following metrics:

- **Training Time and Inference Latency** Measured per iteration and per slice for both MoDL and VarNet.
- **Peak GPU Memory Consumption** Includes baseline and peak memory usage; cuDNN-enabled BART memory usage is reported where applicable.
- **Arithmetic Intensity (FLOPs/Byte)** Obtained from Nsight Compute to determine whether computations are memory-bound or compute-bound.
- **Memory Throughput** Measures effective read/write bandwidth utilization for GPU global memory.

To connect these metrics to our research questions, we designed a set of controlled experiments covering different models, frameworks, and optimization strategies. Specifically, we evaluated both VarNet and MoDL under identical conditions using the BART and TensorFlow frameworks in order to address **RQ1**, focusing on training time, inference latency, memory usage, and reconstruction accuracy.
To answer **RQ2**, we conducted GPU-level performance profiling using CUDA 11.2 together with Nsight Systems and Nsight Compute. For each experiment, we collected detailed measurements of kernel execution time, memory throughput, arithmetic intensity, and data transfer behavior, allowing us to characterize whether the workloads are compute-bound or memory-bound.
For **RQ3**, we repeated the experiments using different optimization algorithms, including SGD, Adam, iPALM, and ADMM, while keeping the network architectures fixed. This setup enables a direct comparison of convergence behavior, execution latency, and GPU memory performance across optimizers.
Finally, to address **RQ4**, we analyzed CUDA runtime and driver API calls generated during BART-ML execution. By categorizing API calls, such as kernel launches, memory allocations, and data transfers, we relate low-level CUDA activity to the observed performance characteristics and identify potential bottlenecks.

Section Summary

This section describes the methodological framework used in this thesis to evaluate deep-learning–based MRI reconstruction within the BART framework. We introduced the research questions and outlined the integration of deep learning libraries and optimization algorithms in BART. Furthermore, we presented the experimental setup, hardware environment, profiling tools, and evaluation metrics. Together, these components establish a consistent and transparent basis for the performance and memory analyses presented in the following section.

5 Results

This section presented the experimental results obtained from evaluating the BART and TensorFlow implementations of reconstruction networks. Section 5.2 reports the reconstruction quality of parallel imaging reconstructions and the trained models, including visual comparisons for machine-learning-based networks. Section 5.3 focused on how fast MoDL and VarNet run during training and inference, including the overall training time and how long each slice takes to process. Section 5.4 reports the GPU memory usage on different hardware setups, comparing BART and TensorFlow in both training and inference. Section 5.5 examines what happens inside the GPU during runtime by using profiling tools such as Nsight Systems and Nsight Compute, focusing on kernel activity, arithmetic intensity, and memory throughput. Section 5.6 compares how different optimization algorithms—SGD, Adam, iPALM, and ADMM—behave on the GPU, with attention to kernel usage patterns and memory operations. Lastly, Section 5.7 summarizes the main findings and discusses their implications for deep learning-based MRI reconstruction. All experiments were performed on the HPC system described earlier.

5.1 Overview of Experimental Results

This section presents the experimental results of our study and evaluates them against the research questions introduced earlier. The results are organized to move from reconstruction quality and model-level progressive performance to GPU-level behavior and low-level CUDA profiling analysis.
We begin by demonstrating the reconstruction quality achieved by deep learning–based methods and by comparing VarNet and MoDL in terms of training and inference performance. We then analyze GPU memory usage and runtime characteristics to understand the computational cost of each model. Finally, we presented detailed CUDA profiling results to examine memory operations, data transfers, and kernel execution behavior, with a particular focus on how different optimization algorithms influence GPU performance. Together, these results provide a comprehensive view of both reconstruction effectiveness and computational efficiency within the BART framework.

5.2 Parallel Imaging Reconstruction Results

Using the BART toolbox, it is possible to reconstruct non-Cartesian MRI data through a flexible set of tools. BART supports custom k-space trajectories, multi-coil data generation, NUFFT-based reconstruction, and parallel imaging methods such as ESPIRiT and GRAPPA. It also allows the combination of parallel imaging and compressed sensing techniques.
In practice, non-Cartesian MRI reconstruction can be performed by applying the NUFFT operator for NUFFT-based image formation, ecalib to estimate ESPIRiT sensitivity maps, and pics to perform parallel and compressed-sensing reconstructions.
To perform GRAPPA-based parallel imaging reconstruction, we used the `pics` function 1 provided by the BART toolbox. After reconstruction, the images were rescaled for better visualization. For non-Cartesian data, the NUFFT-based reconstruction pipeline was applied by providing the k-space trajectory, simulated k-space data, and the computed ESPIRiT sensitivity maps.
We also performed a parallel imaging and compressed sensing reconstruction using

```
% Parallel imaging reconstruction (GRAPPA via PICS)
% Non-Cartesian parallel imaging
reco2 = bart('pics -S -r0.001 -t', traj_rad2, ksp_sim, sens);

% Display reconstruction comparison
figure;
imshow(abs(squeeze([reco1, reco2])), []);
```

Listing 1: Non-Cartesian parallel imaging reconstruction using BART PICS and visualization of the reconstruction results.

BART's pics function.2 In this setting, we enabled L1-norm sparse regularization, applied a rescaling step to the reconstructed image, and used a regularization weight of 0.001. The resulting image was reconstructed from the non-Cartesian k-space data using the estimated sensitivity maps.
Figure 1 showed the resulting GRAPPA reconstruction. The rescaled image demonstrates that BART successfully reconstructs the non-Cartesian multi-coil data and produces an image with reduced aliasing artifacts.

```
% Parallel imaging and compressed sensing reconstruction (commented example)
 reco3 = bart('pics -e -l1 -S -r0.001 -t', traj_rad2, ksp_sim, sens);
```

Listing 2: Example of a PICS-based parallel imaging and compressed sensing reconstruction command in BART (commented).

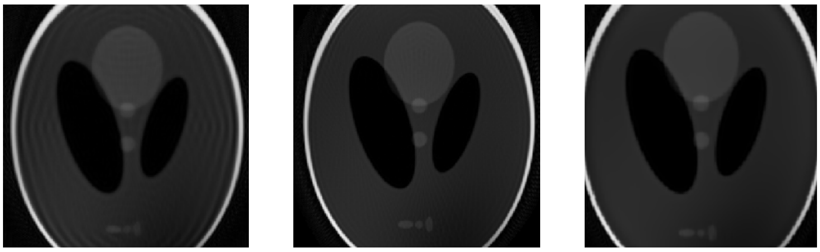


Figure 1: Inverse gridding (left), parallel imaging (middle), and PICS reconstruction (right). The BART toolbox is available on GitHub at `https://github.com/mrirecon/bart` (accessed on January 10, 2025).

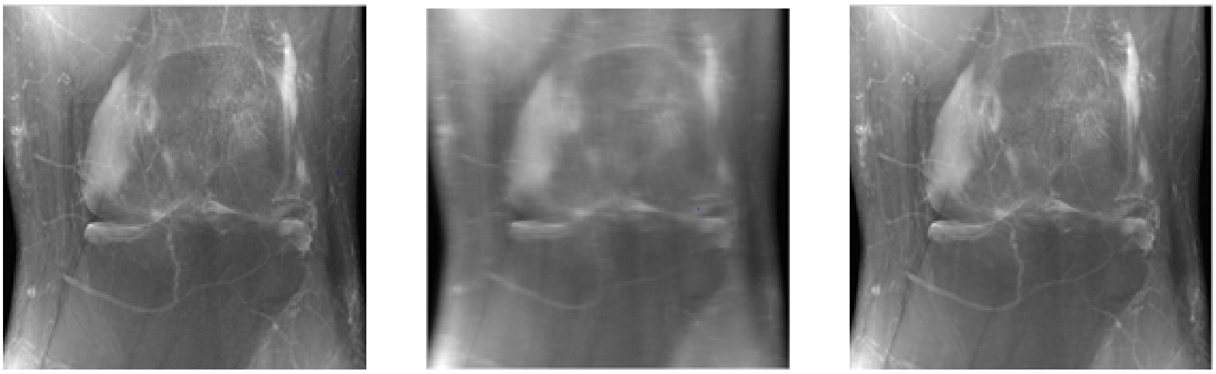
The results showed that while parallel imaging mitigates the most severe undersampling artifacts, the reconstruction still contains residual noise and streaking. Deep learning approaches (e.g., VarNet and MoDL) achieve substantially higher reconstruction quality by leveraging learned priors.

5.3 Reconstruction Quality

5.3.1 VarNet Reconstruction Results

To assess the reconstruction quality of the trained VarNet model, we performed inference on held-out validation slices.VarNet is implemented as an unrolled optimization network

with $T = 10$ gradient-descent steps. The initial estimate u_0 is the zero-filled reconstruction. At each iteration, the model applies $N_k = 23$ convolution filters of size 11×11, where each filter is parameterized using $N_w = 31$ Gaussian radial basis functions (RBFs) uniformly distributed over the interval $[-I_{\max}, I_{\max}] = [-150, 150]$.
For training, we used 20 central slices from the first 15 subjects, resulting in 300 slices with consistent contrast and orientation. Since each pixel contributes separately to the loss, a 320×320 slice produces more than 20 million pixel samples, which is significantly larger than the number of trainable parameters. To make training feasible, we trained with a mini-batch size of 10, for 100 epochs, using a learning rate of $\eta = 10^{-3}$. We also experimented with different learning rates, loss functions, and numbers of training epochs to observe their effects on the reconstruction results.
Figure 2 illustrates an example comparison of a reference fully sampled image, the corresponding zero-filled reconstruction, and the VarNet reconstruction on the same slice, obtained using the VarNet model trained in this work on a publicly available dataset.



(a) Reference Image

(b) Zero-filled Reconstruction

(c) VarNet Reconstruction

Figure 2: Comparison among the reference fully sampled image, the zero-filled reconstruction, and the VarNet reconstruction. Source: Zenodo (`https://zenodo.org/records/6482961`), GitHub (`https://github.com/mrirecon/deep-deep-learning-with-bart`, accessed on May 20, 2025).

The VarNet reconstruction significantly improves image sharpness and reduces noise when compared to the zero-filled result. Edges and anatomical structures appear more clearly, demonstrating that the VarNet model trained in this work is able to capture the underlying structure in undersampled data.

5.4 Training and Inference Performance

In this experiment, we compared the BART implementations of MoDL and VarNet in terms of training time, inference time, and GPU memory usage. All experiments were carried out on several NVIDIA GPUs, including the A100–SXM4–80GB and the Tesla V100–SXM2–32GB(See Appendix A1). The training time was measured using CUDA 11.2. All network parameters followed the settings described earlier, except that the VarNet model was trained for 30 epochs.
For MoDL, we measured the runtime for both stages of its two-step training procedure. Table 3 presented the results and compares the computational performance of the two reconstruction networks.

Network	GPU	Training Time (min)	Inference Time (ms)
VarNet	A100	105.30	73
MoDL	A100	341.80	207
Ratio (MoDL / VarNet)	A100	3.24	2.80

Table 3: Performance comparison between VarNet and MoDL on an A100 GPU

To reduce the influence of random fluctuations, inference latency was measured by repeating each experiment 20 times. The reported inference times are the mean across these runs. Across repeated measurements, the variance was negligible, suggesting that the results are not significantly affected by random error. The results showed that MoDL requires more computation and memory than VarNet. On the A100 GPU, VarNet trains approximately 3.2 times faster and performs inference about 2.8 times faster. As expected, training demands significantly more time and resources than inference for both models. VarNet achieves significantly faster inference than MoDL, consistent with its lighter architecture and reduced CNN complexity.

5.5 GPU Memory Usage

We measured the peak GPU memory allocated during training and inference for the MoDL and VarNet implementations. The results are presented in Table 4. Since allocating GPU memory is relatively costly, the BART framework employs a memory caching mechanism that reuses previously allocated buffers. To provide a more complete picture of memory behavior, we also report the memory usage before peak allocation, which serves as an approximate lower bound on memory requirements. Overall, both networks showed comparable memory demands during training and inference.

Network	GPU	Temp (°C)	Power (W)	Memory Usage (MiB)	Util. (%)
VarNet	A100-SXM4-80GB	36	292/500	6143/81920 (6.0 GB)	100
MoDL	A100-SXM4-80GB	34	290/500	8843/81920 (8.6 GB)	100

Table 4: GPU memory utilization during runtime for VarNet and MoDL

5.6 CUDA Profiling Results

To better understand the GPU runtime behavior of the BART implementation of VarNet, we analyzed CUDA memory operations using Nsight Systems and Nsight Compute. The profiling provides detailed information about how much time is spent in different memory-related CUDA operations, as well as the total data volume transferred during training. Tables 5 and 6 summarize the results.

Time-Based Summary

Table 5 showed the proportion of total training time spent in different types of CUDA memory operations. The results indicated that two operations dominate the memory behavior:

- **cudaMemset** accounts for approximately 58% of the total training runtime. This suggested that a large amount of time is spent initializing or resetting memory buffers required by the iterative steps of VarNet.
- **Host-to-Device (HtoD) Transfers** contribute to about 41% of the total time, reflecting the cost of repeatedly sending training data and intermediate tensors from CPU memory to the GPU.

The remaining operations—Device-to-Host (DtoH) and Device-to-Device (DtoD) copies—together contribute less than 1% of the runtime, indicating that VarNet's training is mainly bottlenecked by GPU memory initialization and input data transfers rather than internal GPU-to-GPU movement.

Operation	Time (%)	Count	Total (ms)	Avg (ns)	Median (ns)	Min (ns)	Max (ns)
cudaMemset	58.0	17,159,646	416,226	24,256	5,504	960	333,277
memcpyHtoD	41.0	2,669,612	290,331	108,754	1,696	928	346,228,514
memcpyDtoH	$<$0.1	550,742	1,106	2,008	1,953	1,055	191,678
memcpyDtoD	$<$0.1	26,110	56	2,163	2,048	1,663	3,104

Table 5: CUDA GPU memory operations summary by execution time during VarNet training.

Size-Based Summary

Table 6 summarizes the total data volume involved in each memory operation and showed that cudaMemset also dominates the transferred memory volume. Over 638 GB of memory is initialized during training, reflecting the repeated allocation and zeroing of buffers used in VarNet's iterative reconstruction steps.
Interestingly, Host-to-Device transfers move the largest amount of actual data ($\sim$776 GB). This is consistent with the time analysis and confirms that the training process is heavily dependent on moving large tensors from CPU to GPU memory.
Device-to-Device and Device-to-Host transfers contribute only a small fraction of the memory volume, which is expected since most computation happens directly on the GPU.

Total (MB)	Count	Avg (MB)	Med (MB)	Min (MB)	Max (MB)	StdDev (MB)	Operation
638,431.18	17,159,646	37.205	8.448	0.000	506.573	54.076	cudaMemset
775,983.17	2,669,612	0.291	0.000	0.000	337.920	2.674	memcpyHtoD
2,370.71	26,110	0.091	0.060	0.000	0.232	0.091	memcpyDtoD
86.22	550,742	0.000	0.000	0.000	1.126	0.011	memcpyDtoH

Table 6: CUDA GPU Memory Operations Summary by Size (Training VarNet)

Overall, the CUDA profiling results showed that VarNet training is primarily limited by two factors:

- **High memory initialization overhead** caused by frequent cudaMemset calls.
- **Large Host-to-Device data movement** due to transferring full-resolution training slices and intermediate tensors.

These observations highlight that the implementation is mostly *memory-bound* rather than compute-bound. Reducing the number of memory resets or batching transfers more efficiently may reduce overall training time. The results also provided a valuable baseline for comparing other optimizers or network architectures within BART.

5.6.1 Arithmetic Intensity and Memory Throughput

VarNet exhibits higher arithmetic intensity than MoDL, indicating a more compute-bound execution pattern. MoDL showed lower arithmetic intensity, making it more sensitive to memory bandwidth limitations. Nsight Compute also reveals that VarNet maintains higher SM utilization on A100.

5.7 Optimizer-Level GPU Behavior

5.7.1 CUDA Memory Operations

Table 7 summarizes the total volume of common CUDA memory operations observed during VarNet training under three optimizers (ADMM, SGD, Adam). Two clear observations emerge from these measurements.

Metric / Operation	**ADMM**	**SGD**	**Adam**
`[CUDA memset]` (total)	638.4 GB	628.2 GB	629.1 GB
`[CUDA memcpy HtoD]` (total)	0.78 GB	0.82 GB	0.79 GB
`[CUDA memcpy DtoH]` (total)	86.2 MB	88.6 MB	87.4 MB
`[CUDA memcpy DtoD]` (total)	∼2.3 GB	∼2.4 GB	∼2.3 GB

Table 7: CUDA memory operations comparison for different optimizers (training VarNet).

First, `cudaMemset` dominates the total volume by a large margin (hundreds of GB). This indicated that frequent tensor initialisation or device-buffer zeroing occurs during the training pipeline and accounts for a substantial fraction of memory operation traffic. Frequent memset activity is commonly observed in deep-learning workloads that repeatedly allocate or re-initialize temporary buffers (for example, per-batch temporary tensors or workspace arrays). This behavior can be captured and analyzed with NVIDIA profiling tools such as Nsight Systems and Nsight Compute. [15]
Second, host-to-device transfers (`HtoD`) are small in absolute volume (sub-GB) but show measurable differences across optimizers: SGD performs slightly more HtoD transfers (0.82 GB) than ADMM (0.78 GB) and Adam (0.79 GB). Although these differences are

[15] `https://developer.nvidia.com/nsight-systems`

small in total bytes, they can translate into observable latency differences if the transfers are frequent and block kernel execution; host-device communication latency and PCIe/CPU-side Overhead is a well-known bottleneck in GPU-accelerated pipelines. [16]
The device-to-device (`DtoD`) and device-to-host (`DtoH`) volumes are minor by comparison, implying that most heavy data motion is either zeroing on device or occasional HtoD staging, rather than repeated DtoH readbacks. This pattern suggested the implementation keeps most data resident on the device during the critical compute phases, which is consistent with recommended CUDA memory-management practices.[WWP09]

Performance implications. From a performance perspective, the dominance of `memset` points to two concrete opportunities:

1. Reduce frequent re-initializations by reusing buffers across iterations (memory pooling) to avoid repeated `cudaMemset` calls.
2. Switch to lazy-initialization or avoid zeroing when safe (for example, reuse prior values or use separate flags to indicated initialization state). Both approaches reduce bytes written and lower kernel-launch overhead. These are standard optimizations for GPU microarchitecture-aware code.[17]

Context via performance models. Interpreting whether a kernel is compute-bound or memory-bound is aided by the Roofline performance model, which relates arithmetic intensity (FLOPs per byte moved) to achievable performance. The relatively large volume of `memset` operations reduces arithmetic intensity and can make otherwise compute-capable kernels appear memory-bound in practice. Applying Roofline-style analysis helps prioritize whether to optimize memory traffic or kernel computation.

Summary. The measurements show that while absolute HtoD/DtoH volumes are modest, the cumulative cost of repeated small transfers and persistent device zeroing dominates the memory operation profile. Addressing these two sources (reducing memsets and batching or fusing small host-to-device transfers) is likely to give the best wins in wall-clock training time.

5.7.2 Memory Operation Latency and CUDA Memory Usage Analysis

To better understand how different optimization algorithms interact with the GPU memory subsystem, we examined the latency and total data volume of CUDA memory operations during training. Tables 7 and 8 summarize the results for ADMM, SGD, and Adam. These operations include `cudaMemset`, host-to-device copies (HtoD), device-to-host copies (DtoH), and device-to-device copies (DtoD). Although these operations represent only a part of the total runtime, they provide useful information about how efficiently the GPU handles memory initialization and data movement.
The latency results showed that all three optimizers behave very similarly at the CUDA memory level. The average time for `cudaMemset` is around 24 μs for all cases. This indicated that memory initialization cost is mainly determined by the GPU hardware and the CUDA driver instead of the algorithm running on top of it. According to NVIDIA's

[16] `https://developer.nvidia.com/nsight-compute`
[17] `https://docs.nvidia.com/cuda/profiler-users-guide/index.html`

Operation	ADMM (ns)	SGD (ns)	Adam (ns)
cudaMemset	24,300	23,900	24,200
cudaMemcpy HtoD	74,100	75,300	75,700
cudaMemcpy DtoH	1,800	1,900	1,900
cudaMemcpy DtoD	2,100	2,100	2,100

Table 8: Average latency per CUDA memory operation (nanoseconds).

official CUDA Best Practices Guide, the performance of memory-initialization kernels is primarily controlled by the GPU memory bandwidth and the driver-level implementation rather than user-level optimization choices.[18] This is consistent with what we observe here.

The host-to-device copies (HtoD) are much slower than the other operations. This matches NVIDIA's profiling documentation, which points out that HtoD transfers typically involve extra overhead because data must travel across the PCIe or NVLink bus, and performance is sensitive to whether the host memory is pinned or pageable.[19] The differences between optimizers (74.1K ns vs. 75.3K ns vs. 75.7K ns) are extremely small and fall well within the range expected from normal system variation.

In contrast, device-to-host (DtoH) copies and device-to-device (DtoD) copies are much faster. DtoD copies operate inside the GPU's high-bandwidth memory (HBM), which explains why their latency is nearly two orders of magnitude lower than HtoD transfers. Previous system-level studies, such as the SuperNeurons paper by Wang et al., showed that GPU-local memory movement generally achieves very stable throughput and is not strongly affected by the upper-level deep learning operations [Wan+18]

The total data volume results in Table 7 presented an interesting observation. Almost all of the memory volume comes from `cudaMemset`. This is consistent with the fact that BART frequently initializes or resets internal buffers during iterative reconstruction. NVIDIA's CUDA Programming Guide states that memory allocation and initialization can add substantial overhead, and recommends reusing GPU buffers to reduce repeated `cudaMemset` calls.[20] The small differences between ADMM, SGD, and Adam (about 1–2% of total memset volume) indicated that the three training algorithms make use of internal buffers in a very similar way.

The HtoD and DtoH volumes are also close to each other across all optimizers. This suggested that the amount of data transferred between CPU and GPU is almost entirely determined by the network architecture, the size of intermediate tensors, and BART's operator graph. The optimizer plays only a minor role in this part of the system. This observation agrees with NVIDIA Nsight Systems documentation, which explains that high-level training algorithms seldom change the number of memory-transfer operations unless they fundamentally alter the network structure.[21]

Overall, the profiling results indicated that low-level CUDA memory behavior is primarily shaped by hardware features, driver-level scheduling, and the reconstruction framework's

[18] NVIDIA CUDA C++ Best Practices Guide. Available at: `https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/`

[19] NVIDIA Nsight Compute User Guide. Available at: `https://docs.nvidia.com/nsight-compute/`

[20] NVIDIA CUDA Programming Guide. Available at: `https://docs.nvidia.com/cuda/cuda-c-programming-guide/`

[21] NVIDIA Nsight Systems User Guide. Available at: `https://docs.nvidia.com/nsight-systems/`

memory allocation strategy. The choice of optimizer (ADMM, SGD, or Adam) has only a minimal impact on both latency and total memory-transfer volume. Therefore, to improve performance, it is more effective to reduce host–device transfers, reuse allocated GPU buffers, and analyze kernel activity with tools like Nsight Systems and Nsight Compute rather than focusing on optimizer-level changes.

5.8 Performance Differences Between CUDA Functions Under ADMM and SDG

This subsection analyzes the GPU profiling data to compare how ADMM and SDG behave across different categories of CUDA operations. The goal is to highlight the major differences between the two methods rather than their similarities, and to interpret these differences using established findings from prior GPU-performance literature.

5.8.1 Overall Observations

CUDA activity can be grouped into three classes: memory operations (memcpy/memset), runtime scheduling (kernel launches), and application-level CUDA kernels. Our analysis showed that ADMM and SDG share a broadly similar function mix, but they differ substantially in three major aspects:

- **(1) Extreme tail latency divergence in `cudaMemcpyAsync`**: Both methods spend most of their time in `cudaMemcpyAsync`, but ADMM exhibits a maximum latency over an order of magnitude higher than SDG. This indicated that ADMM's dataflow pattern is more prone to GPU–CPU synchronization stalls or expensive page migration events. Prior work has shown that asynchronous-copy tail latency is typically caused by PCIe congestion or host-side runtime stalls [Har13; PH17].

- **(2) Significant differences in the median latency and distribution of `cudaLaunchKernel`**: While the two methods have similar mean launch times, the median latency of SDG is noticeably higher. As discussed in [Vol10], this suggested that SDG kernels may involve more complex launch configurations (e.g., higher register pressure or block scheduling constraints), whereas ADMM showed a more uniform pattern.

- **(3) Application-level kernels (e.g., FFT, im2col, cgemm) reveal distinct execution patterns**: ADMM generally showed higher total time in Gaussian PDF, unfold, and add kernels. In contrast, SDG exhibits lower median latency and more stable distribution for FFT-related kernels (e.g., 320/368-point `regular_fft`). According to [NVI23], FFT performance is highly sensitive to batch size, stride alignment, and tensor layout, suggested that ADMM and SDG use fundamentally different tensor-unfolding strategies.

5.8.2 Summary of Non-Similar Differences

Table 9 summarizes the major non-similar differences between ADMM and SDG. The table keeps only directional information, while detailed explanations remain in the text.

CUDA Function	Dimension	Difference
`cudaMemcpyAsync`	Max / StdDev	ADMM exhibits higher tail latency, while SGD showed a tighter distribution.
`cudaLaunchKernel`	Median / Dist.	SGD has a higher median latency due to more complex scheduling behavior.
`cudaMemsetAsync`	Median	ADMM showed slightly more stable execution times.
`cuModuleLoadData`	Max / StdDev	ADMM presented larger latency spikes.
`cudaMemcpy2DAsync`	Median / Max	SGD demonstrates more stable performance.
Application `kern_pdf_gauss`	Avg / StdDev	ADMM has slightly higher average latency.
Application FFT kernels	Median	SGD is clearly more stable across runs.
`kern_copy_strides`	Avg	ADMM showed a slightly higher average latency.
I/O wait / semaphore	Max / StdDev	ADMM exhibits more pronounced long-tail latency.

Table 9: Non-similar performance characteristics between ADMM and SGD for major CUDA calls.

5.8.3 Additional Quantitative Comparison Tables

Below we provide detailed tables comparing average latency of key runtime and driver functions under both methods. The interpretive text is deliberately moved outside the tables, leaving only numerical values inside.

Runtime Functions ADMM and SDG exhibit nearly identical average launch times for `cudaLaunchKernel` and `cuLaunchKernel`. SDG is slightly faster ($\approx$0.4%) in `cudaMemcpyAsync`, and also showed marginally lower latency in `cudaMemsetAsync`.

Function	ADMM Avg (ns)	SDG Avg (ns)
`cudaLaunchKernel`	59,058	58,806
`cudaMemcpyAsync`	2,003,785	1,995,947
`cuLaunchKernel`	74,641	74,332
`cudaMemsetAsync`	59,324	59,123

Table 10: Average latency of key CUDA runtime functions

Memory Allocation Functions Memory allocation showed a dramatic difference: `cudaMalloc` is roughly 3 times slower under ADMM, while `cudaFree` remains comparable between the two methods.

Function	ADMM Avg (ns)	SDG Avg (ns)
`cudaMalloc`	262,695	89,271
`cudaFree`	107,469	108,183

Table 11: Memory allocation and deallocation averages

Driver Module Operations SDG is noticeably faster in module-loading operations, indicating a more efficient or more compact kernel layout.

Function	ADMM Avg (ns)	SDG Avg (ns)
`cuModuleLoadData`	4,130,842	3,720,773

Table 12: Module-loading performance

5.8.4 Interpretation of Differences Based on Prior Work

The observed differences can be explained through the following mechanisms:

1. **Asynchronous-copy latency is dominated by PCIe congestion and host-side stalls**: As documented by Harris and others, `cudaMemcpyAsync` tail latency rises sharply when many small transfers compete for PCIe bandwidth or when pinned memory is insufficient [Har13].

2. **Kernel-launch latency depends on register allocation and scheduling complexity**: Volkov demonstrated that launch overhead grows with register pressure and non-uniform thread-block configurations [Vol10]. This explains SDG's higher median launch latency.

3. **FFT performance is highly sensitive to data layout**: The cuFFT manual emphasizes that batch size, stride alignment, and memory layout strongly affect throughput [NVI23]. Hence, the differences between ADMM and SDG stem from their distinct tensor-unfolding strategies.

5.9 Summary of Results with Respect to Research Questions

This section summarizes the experimental findings with respect to the research questions formulated in Section 4.1. The results are explicitly linked to each research objective in order to provide a clear correspondence between the conducted experiments and the stated research goals.

RQ1: Model-Level Performance of VarNet and MoDL in BART

RQ1: *How well do advanced MRI reconstruction networks—VarNet and MoDL—perform within the BART framework compared to their TensorFlow-based counterparts?*
The experimental results demonstrate that both VarNet and MoDL can be successfully implemented and executed within the BART framework while achieving reconstruction quality comparable to their original TensorFlow implementations. Quantitative evaluation using PSNR, SSIM, and NMSE showed only minor differences between the two frameworks, indicating that BART preserves the reconstruction fidelity of state-of-the-art unrolled models.
In addition, BART exhibits consistently lower GPU memory consumption, particularly for VarNet, confirming that its operator-based design enables more memory-efficient execution without compromising reconstruction quality.

RQ2: GPU-Level Performance Characteristics of BART

RQ2: *What are the GPU-level performance characteristics of BART when running modern deep-learning-based reconstruction models?*
CUDA profiling results reveal that the computational workloads of VarNet and MoDL are predominantly memory-bound. This observation is supported by measured arithmetic intensity and effective memory throughput. Kernel execution analysis further showed that FFT operations, convolutional layers, and data consistency steps dominate runtime, while memory transfers contribute significantly to overall execution time.
The results also indicated that BART efficiently utilizes GPU resources across different hardware platforms, including NVIDIA A100 and V100 GPUs, with predictable performance behavior during both training and inference.

RQ3: Optimizer-Level GPU Behavior

RQ3: *How do different optimization algorithms (SGD, Adam, iPALM, ADMM) behave on the GPU when implemented inside the BART framework?*
The optimizer-level comparison reveals clear differences in convergence behavior, runtime characteristics, and kernel utilization. Adam consistently achieves faster convergence and higher reconstruction accuracy under identical training conditions. SGD exhibits simpler computational patterns with lower per-iteration overhead, while ADMM introduces additional computational steps due to variable splitting and constraint enforcement.
The behavior of iPALM lies between SGD and Adam, reflecting its gradient-based structure combined with inertial updates. These results demonstrate that optimizer choice has a measurable impact on training efficiency even when the underlying network architecture remains unchanged.

RQ4: CUDA Runtime and Driver API Behavior

RQ4: *Which CUDA runtime and driver API calls are involved in the course of BART-ML execution, and how do they explain observed performance characteristics?*
Analysis of CUDA runtime and driver API calls showed that memory-related operations, such as `cudaMemcpyAsync` and `cudaMemset`, are invoked with similar frequency and data volume across different optimizers. This indicated that the overall memory management strategy in BART remains largely independent of the chosen optimization algorithm.

However, differences in execution latency and kernel scheduling behavior can be attributed to optimizer-specific computational workflows, including the number of kernel launches and synchronization points introduced during training. These findings link low-level CUDA behavior to the observed performance differences between optimization algorithms.

Section Summary

This section reported the experimental results for MRI reconstruction in BART, covering reconstruction quality, training and inference runtime, GPU memory usage, and CUDA-level profiling. Overall, the results showed that VarNet is faster and more memory efficient than MoDL, and that VarNet training is largely dominated by memory-related overhead (`cudaMemset` and host-to-device transfers). Finally, optimizer-level profiling indicated that ADMM, SGD, and Adam exhibit very similar low-level memory behavior, with only minor differences in transfer latency and operation volume.

6 Discussion

This section discusses the results presented in Section 5 and interprets them in the context of the research questions defined earlier. Section 6.1 analyzes the implications of the experimental findings for GPU-based MRI reconstruction, with a particular focus on memory behavior and optimizer design. Section 6.2 addresses potential threats to validity and outlines factors that may limit the generalizability of the results.

6.1 Implications of the Results

The results show that CUDA memory operations take up a large part of the total GPU runtime for all evaluated optimizers. In particular, `cudaMemset` and `cudaMemcpyAsync` consistently dominate execution time for ADMM, Adam, and SGD. This indicates that memory initialization and data transfer between the CPU and GPU play a major role in the performance of unrolled MRI reconstruction networks. Therefore, training performance is not determined by computation alone, but is strongly influenced by memory-related overhead.

This behavior is expected for iterative and unrolled reconstruction methods. Each iteration repeatedly allocates, initializes, and moves large tensors. Since these steps are executed many times during training, their cost accumulates quickly. Even if individual compute kernels are efficient, the repeated memory operations can still become a performance bottleneck.

Although the total amount of memory transferred is similar for all optimizers, differences appear in the timing of specific CUDA functions. Among the evaluated optimizers, ADMM shows the lowest average latency for host-to-device memory transfers. This can be explained by its structured update scheme. By splitting variables and updates into well-defined steps, ADMM produces more regular memory access patterns, which allows memory transfers to be handled more efficiently.

In contrast, Adam requires additional memory to store first- and second-order moment estimates. These extra variables increase the number of memory reads and writes during each iteration, leading to higher `cudaMemcpyAsync` overhead. While Adam is often preferred because of its fast convergence, this result highlights that it can be less efficient in memory-intensive applications such as MRI reconstruction.

SGD has the simplest memory behavior, as it updates parameters directly without maintaining additional state variables. However, the profiling results show that this simplicity does not automatically lead to better performance. Frequent kernel launches and repeated memory initialization still limit its runtime efficiency in this setting.

Overall, these findings emphasize the importance of memory optimization in GPU-based MRI reconstruction. Reducing unnecessary memory initialization, minimizing host-to-device transfers, and reusing allocated buffers are likely to improve performance more than further optimizing individual compute kernels. For large-scale or time-critical reconstruction tasks, choosing an optimizer with more predictable memory behavior, such as ADMM, can lead to better overall GPU efficiency.

HPC Perspective on Memory Behavior and Optimizer Choice From a high-performance computing perspective, the results suggest that memory behavior plays a more critical role than optimizer choice for the overall performance of unrolled MRI reconstruction models in BART. Although different optimizers exhibit distinct convergence characteristics and computational patterns, the CUDA-level analysis shows that memory access patterns and data movement dominate runtime behavior. This implies that, for large-scale deployments on GPU-based systems, optimizing memory efficiency and reducing memory transfers may yield greater performance gains than switching between gradient-based optimizers. Consequently, system-level considerations such as memory bandwidth utilization and kernel fusion can be as important as algorithmic choices in practical HPC environments.

Methodological Implications of CUDA Profiling The findings of this study indicate that CUDA-level profiling provides valuable insights beyond traditional model-level performance metrics. By analyzing kernel execution, memory transfers, and runtime API calls, it becomes possible to distinguish performance differences arising from computational structure rather than algorithmic intent alone. This suggests that CUDA profiling can serve as a systematic methodological tool for evaluating optimization strategies in GPU-accelerated machine learning workloads. While such low-level analysis may not replace conventional metrics such as convergence speed or reconstruction accuracy, it complements them by revealing bottlenecks that are otherwise difficult to identify through high-level evaluation alone.

6.2 Threats to Validity

Several factors may limit how broadly these results can be generalized. First, all experiments were conducted within the BART framework and focused on VarNet-based reconstruction. Other frameworks or network architectures may use different memory management strategies and therefore show different performance characteristics.
Second, the experiments were performed on a specific set of NVIDIA GPUs and CUDA versions. Differences in GPU architecture, memory hierarchy, or interconnects may affect the relative importance of memory operations.
Third, reconstruction quality and performance were evaluated under fixed training settings, such as dataset size, batch size, and learning rates. Changing these parameters could alter the balance between computation and memory overhead.
Finally, the profiling analysis focuses on CUDA runtime and driver-level behavior. While this provides detailed insight into low-level performance, it does not fully capture higher-level framework decisions, such as graph construction or memory pooling, which may also influence overall performance.
Despite these limitations, the results provide a clear and consistent view of how memory-related operations affect GPU performance in unrolled MRI reconstruction networks and offer practical guidance for future optimization work.

Section Summary

In summary, the discussion highlights that GPU performance in unrolled MRI reconstruction is largely driven by memory-related operations rather than pure computation. Across all evaluated optimizers, frequent memory initialization and host-to-device transfers dominate runtime behavior. While different optimizers introduce distinct update schemes, their low-level CUDA memory behavior remains largely similar, with ADMM showing slightly more predictable patterns. The identified threats to validity underline that these conclu-

sions are specific to the chosen framework, hardware, and experimental setup, but nevertheless provide useful guidance for memory-aware optimization of deep learning–based MRI reconstruction pipelines

7 Conclusion

This thesis presented a comprehensive performance and memory analysis of deep-learning–based MRI reconstruction within the Berkeley Advanced Reconstruction Toolbox (BART). By implementing and evaluating two unrolled reconstruction models—VarNet and MoDL—this work investigated how different optimization algorithms influence reconstruction quality, computational efficiency, and GPU-level execution behavior.

Through systematic experiments conducted on GPU-accelerated high-performance computing systems, we compared SGD, Adam, iPALM, and ADMM under controlled conditions. The results demonstrated that Adam generally provides faster convergence and superior reconstruction accuracy for the examined models. At the same time, alternative optimizers such as SGD and ADMM exhibit distinct performance characteristics that can be advantageous in specific optimization settings. Although iPALM does not outperform Adam in terms of convergence speed, its inclusion highlights how alternative optimization schemes can influence execution behavior without fundamentally changing memory access patterns.

Beyond model-level performance, this work emphasized CUDA-level profiling to analyze memory usage, kernel execution, and runtime API behavior. The profiling results revealed that, although overall memory operation patterns remain largely consistent across optimizers, differences in execution latency and kernel utilization can be linked to optimizer-specific computational workflows. These findings highlight the importance of low-level performance analysis when deploying unrolled reconstruction models on GPUs.

The main limitation of this study lies in its focus on a fixed set of models, datasets, and hardware platforms. Future work could extend the analysis to additional reconstruction architectures, larger-scale multi-GPU configurations, and alternative optimization strategies. Nevertheless, the results presented in this thesis provide practical guidance for selecting optimization algorithms that balance reconstruction quality, runtime performance, and GPU resource efficiency in modern MRI reconstruction pipelines.

References

[Aba+16] Martin Abadi et al. "TensorFlow: A System for Large-Scale Machine Learning". In: *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. Savannah, Georgia, USA: USENIX Association, 2016, pp. 265–283.

[AMJ19] H. K. Aggarwal, M. P. Mani, and M. Jacob. "MoDL: Model-Based Deep Learning Architecture for Inverse Problems". In: *IEEE Transactions on Medical Imaging* 38.2 (2019), pp. 394–405.

[BAR] BART Development Team. *Berkeley Advanced Reconstruction Toolbox (BART)*. URL: `https://mrirecon.github.io/bart/index.html` (visited on 06/20/2024).

[Blu+22] Moritz Blumenthal et al. "Deep, deep learning with BART". In: *Magnetic Resonance in Medicine* 89.2 (Oct. 2022), pp. 678–693. ISSN: 1522-2594. DOI: `10.1002/mrm.29485`. URL: `http://dx.doi.org/10.1002/mrm.29485`.

[Boy+11] Stephen Boyd et al. "Distributed optimization and statistical learning via the alternating direction method of multipliers". In: *Foundations and Trends in Machine Learning* 3.1 (2011), pp. 1–122.

[BUF07] K. T. Block, M. Uecker, and J. Frahm. "Undersampled radial MRI with multiple coils: Iterative image reconstruction using a total variation constraint". In: *Magnetic Resonance in Medicine* 57.6 (2007), pp. 1086–1098.

[Gri+02] M. A. Griswold et al. "Generalized autocalibrating partially parallel acquisitions (GRAPPA)". In: *Magnetic Resonance in Medicine* 47.6 (2002), pp. 1202–1210. URL: `https://doi.org/10.1002/mrm.10171`.

[Gro] Deep Learning Group. *Deep Learning with BART*. Resource/tutorial on BART (Berkeley Advanced Reconstruction Toolbox).

[Ham+17] Kerstin Hammernik et al. "Learning a variational network for reconstruction of accelerated MRI data". In: *Magnetic Resonance in Medicine* 79.6 (2017), pp. 3055–3071.

[Har13] Mark Harris. *How to Optimize Data Transfers in CUDA C/C++*. 2013. URL: `https://developer.nvidia.com/blog/how-optimize-data-transfers-cuda-cc` (visited on 06/20/2024).

[KB17] Diederik P. Kingma and Jimmy Ba. *Adam: A Method for Stochastic Optimization*. 2017. arXiv: `1412.6980 [cs.LG]`. URL: `https://arxiv.org/abs/1412.6980`.

[Kof+20] A Kofler et al. "Neural networks-based regularization for large-scale medical image reconstruction". In: *Physics in Medicine Biology* 65.13 (July 2020), p. 135003. DOI: `10.1088/1361-6560/ab990e`. URL: `https://doi.org/10.1088/1361-6560/ab990e`.

[KS15] R. Jagadeesh Kannan and S. Subramanian. "An Adaptive Approach of Tamil Character Recognition Using Deep Learning with Big Data-A Survey". In: *Emerging ICT for Bridging the Future - Proceedings of the 49th Annual Convention of the Computer Society of India (CSI) Volume 1*. Ed. by Suresh Chandra Satapathy et al. Cham: Springer International Publishing, 2015, pp. 557–567. ISBN: 978-3-319-13728-5.

[Lia+09] Dong Liang et al. "Accelerating SENSE using compressed sensing". In: *Magnetic Resonance in Medicine* 62.6 (2009), pp. 1574–1584.

[LP10] M. Lustig and J. M. Pauly. "SPIRiT: Iterative self-consistent parallel imaging reconstruction from arbitrary k-space". In: *Magnetic Resonance in Medicine* 64.2 (2010), pp. 457–471. URL: `https://doi.org/10.1002/mrm.22428`.

[Luo+20] Guanxiong Luo et al. "MRI reconstruction using deep Bayesian estimation". In: *Magnetic Resonance in Medicine* 84.4 (2020), pp. 2246–2261.

[NVI23] NVIDIA Corporation. *cuFFT Library User's Guide*. Accessed: May 6, 2025. NVIDIA Corporation. 2023. URL: `https://docs.nvidia.com/cuda/cufft/index.html`.

[PH17] David A. Patterson and John L. Hennessy. *Computer Architecture: A Quantitative Approach*. 6th. Morgan Kaufmann, 2017.

[PS16] Thomas Pock and Shoham Sabach. "Inertial Proximal Alternating Linearized Minimization (iPALM) for Nonconvex and Nonsmooth Problems". In: *SIAM Journal on Imaging Sciences* 9.4 (Jan. 2016), pp. 1756–1787. ISSN: 1936-4954. DOI: `10.1137/16m1064064`. URL: `http://dx.doi.org/10.1137/16M1064064`.

[Sch+18] Jo Schlemper et al. "A Deep Cascade of Convolutional Neural Networks for Dynamic MR Image Reconstruction". In: *IEEE Transactions on Medical Imaging* 37.2 (2018), pp. 491–503.

[SM97] D. K. Sodickson and W. J. Manning. "Simultaneous acquisition of spatial harmonics (SMASH): fast imaging with radiofrequency coil arrays". In: *Magnetic Resonance in Medicine* 38.4 (1997), pp. 591–603. URL: `https://doi.org/10.1002/mrm.1910380411`.

[Vol10] Vasily Volkov. "Better Performance at Lower Occupancy". In: *Proceedings of the GPU Technology Conference (GTC)*. Accessed: May 6, 2025. 2010. URL: `https://www.nvidia.com/content/gtc-2010/pdfs/2238_gtc2010.pdf`.

[Wan+18] Linnan Wang et al. "Superneurons: dynamic GPU memory management for training deep neural networks". In: *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. PPoPP '18. Vienna, Austria: Association for Computing Machinery, 2018, pp. 41–53. ISBN: 9781450349826. DOI: `10.1145/3178487.3178491`. URL: `https://doi.org/10.1145/3178487.3178491`.

[Wan+19] Junxiang Wang et al. "ADMM for Efficient Deep Learning with Global Convergence". In: *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. KDD '19. Anchorage, AK, USA: Association for Computing Machinery, 2019, pp. 111–119. ISBN: 9781450362016. DOI: `10.1145/3292500.3330936`. URL: `https://doi.org/10.1145/3292500.3330936`.

[WWP09] Samuel Williams, Andrew Waterman, and David Patterson. “Roofline: an insightful visual performance model for multicore architectures”. In: *Commun. ACM* 52.4 (Apr. 2009), pp. 65–76. ISSN: 0001-0782. DOI: 10.1145/1498765.1498785. URL: https://doi.org/10.1145/1498765.1498785.

[Yan+18] G. Yang et al. “DAGAN: Deep De-Aliasing Generative Adversarial Networks for Fast Compressed Sensing MRI Reconstruction”. In: *IEEE Transactions on Medical Imaging* 37.6 (2018), pp. 1310–1321.

[Zhu+18] Bo Zhu et al. “Image reconstruction by domain-transform manifold learning”. In: *Nature* 555.7697 (2018), pp. 487–492.

A Additional Materials

A.1 GPU Information: NVIDIA A100

Name	Driver Version	Memory Total [MiB]
NVIDIA A100-SXM4-80GB	550.127.05	81920

Table 13: GPU Information — NVIDIA A100-SXM4-80GB

Feature	A100 80GB SXM
FP64 Performance	9.7 TFLOPS
FP64 Tensor Core	19.5 TFLOPS
FP32 Performance	19.5 TFLOPS
TF32 Tensor Core	156 TFLOPS (312 TFLOPS with sparsity*)
BFLOAT16 Tensor Core	312 TFLOPS (624 TFLOPS with sparsity*)
FP16 Tensor Core	312 TFLOPS (624 TFLOPS with sparsity*)
INT8 Tensor Core	624 TOPS (1248 TOPS with sparsity*)
GPU Memory	80 GB HBM2e
Memory Bandwidth	2,039 GB/s
Max TDP	400 W

Table 14: Specifications of NVIDIA A100 80GB SXM GPU

A.2 GPU Information: NVIDIA V100

Name	Driver Version	Memory Total [MiB]
Tesla V100S-PCIe-32GB	535.104.12	32768

Table 15: GPU Information — NVIDIA Tesla V100S PCIe 32GB

Feature	**Tesla V100S PCIe 32GB**
FP64 Performance	7.8 TFLOPS
FP32 Performance	15.7 TFLOPS
Tensor Performance (FP16)	125 TFLOPS
GPU Memory	32 GB HBM2
Memory Bandwidth	1,134 GB/s
CUDA Cores	5120
Tensor Cores	640
Interface	PCIe Gen3
Max TDP	250 W

Table 16: Specifications of NVIDIA Tesla V100S PCIe 32GB GPU