DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Robotics, Cognition, Intelligence

Improving Efficiency in X-Ray Computed Tomography Using Compile-Time Programming and Heterogeneous Computing

Author:	Jens Petit
Supervisor:	PD Dr. Tobias Lasser
Advisor:	PD Dr. Tobias Lasser
Submission Date:	May 6th, 2020

DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Robotics, Cognition, Intelligence

Improving Efficiency in X-Ray Computed Tomography Using Compile-Time Programming and Heterogeneous Computing

Effizienzsteigerungen in Röntgen-Computertomographie mittels Compile-Time Programming und heterogenem Computing

Author:	Jens Petit
Supervisor:	PD Dr. Tobias Lasser
Advisor:	PD Dr. Tobias Lasser
Submission Date:	May 6th, 2020

I confirm that this master's thesis in robotics, cognition, intelligence is my own work and I have documented all sources and material used.

Munich, May 6th, 2020 Jens Petit

Acknowledgments

A sincere thank you to my supervisor Tobias and everyone who contributed to *elsa* in the past six months. I had a great time working with you! Furthermore, the free and open-source software community provides some of the tools I love to use everyday. I am grateful for every developer who dedicates their resources and keeps the spirit of free software alive. Finally, some persons where more or less directly involved with this thesis: Mum, Jan, Marika, Veronika and my roomies. Thank you! I am happy to have all of you in my life.

Abstract

Due to large data sizes in X-ray computed tomography, efficiency in implementing reconstruction algorithms is a core requirement. Efficiency aims to reduce both space and time complexity. This is especially important as ever-increasing resolution along with time requirements have raised the computational load further.

To mitigate these effects, the present thesis deals with improving memory and computational aspects of the tomographic reconstruction software framework named elsa. It is a free and open-source software which is developed at the Computational Imaging and Inverse Problems group at the Technical University of Munich.

The first efficiency improvements examined in this thesis stem from the application of expression templates. They are a compile-time programming technique which saves the work to be done for later lazy evaluation instead of creating temporary intermediate results.

The second main part investigates using GPUs for performing element-wise vector operations, because many computations within elsa fall into this category. Naturally, these operations are parallel problems and therefore well suited for heterogeneous computing environments. Contrary to CPUs, GPUs are very powerful in performing massively parallelized computations due to the fact that they consist of hundreds of computational cores.

The thesis starts with a general problem statement as well as motivation for X-ray computed tomography. Subsequently, a short introduction to X-ray imaging and computed tomography illustrates the context. It is followed by an overview of the elsa software framework. Then, the concept of expression templates and their application within elsa is explained. Benchmarking results conclude the chapter. Next, the utilization of GPUs for element-wise vector operations is discussed. As a solution, a custom GPU library called Quickvec is developed which internally builds on top of the previously designed expression templates. Its impact on efficiency is measured in different scenarios. The final chapter concludes the thesis with a summary and a discussion about the limitations of the presented solutions.

The main contribution of this thesis is the development of a generic element-wise vector arithmetic library utilizing the processing power of GPUs and its integration into elsa.

Contents

Abbreviations

2D two-dimensional

3D three-dimensional

API Application Programming Interface

BLAS Basic Linear Algebra Subprograms

CPU Central Processing Unit

CT Computed Tomography

CUDA Compute Unified Device Architecture

ET Expression Template

GPU Graphical Processing Unit

JIT Just-In-Time

RAII Resource Acquisition Is Initialization

saxpy single precision a times x plus y

SDK Software Development Kit

STL Standard Template Library

UML Unified Modelling Language

1. Introduction

1.1. Motivation and Research Background

X-ray Computed Tomography (CT) allows us to see inside objects where our eyes only perceive the outer structure. It does not require opening up the objects and therefore enables insights in a non-destructive way (ignoring radiation damage). What seems like a miracle is based on the close interaction between physics, mathematics, engineering and computer science. The applications are manifold: from scanning passengers at airports over obtaining information about broken bones to locating material fatigue in security critical parts like turbine blades [Her09].

Over the years, the requirements towards X-ray CT have become ever more ambitious. Two of them are considered in this thesis: First, higher scanning resolutions for more fine-grained analysis as well as localization within the objects. Second, shorter reconstruction times giving results within minutes after the scan. Both requirements contradict each other, as a higher resolution increases the computational load and therefore leads to a longer reconstruction time. Consequently, one key aspect of research concerning CT is reducing both space and time complexity of the reconstruction algorithms.

Over the last few years, a general software framework for solving reconstruction tasks in CT has been developed at the *Computational Imaging and Inverse Problems* research group of the *Technical University of Munich*. It implements state-of-the-art algorithms and is available as free and open-source software under the name of *elsa* [LHF19]. The present thesis deals with improving memory and computational aspects of it.

From a low-level perspective, most of the computations within elsa fall into the category of linear algebra or element-wise vector operations. Consequently, optimizing these operations is promising from an efficiency perspective, as it affects all reconstruction problems. Already at compile-time, some information about the computations to be done is available. This can be utilized to automatically create more efficient code via template metaprogramming, a technique which is known in this context as Expression Templates (ETs).

The element-wise vector operations within elsa are parallel problems where each computation is independent from each other. Hence, they are perfectly suited for the application of heterogeneous computing, leveraging the higher efficiency of GPUs over CPUs for parallel processing tasks [Coo12].

Developing and incorporating both ETs and GPU-based vector arithmetic into elsa are the objectives of this thesis. Ultimately, the goal of elsa is to provide a high-performance framework for developing novel algorithms and imaging techniques within the context of X-ray CT.

1.2. Thesis Outline

The present thesis starts with a brief introduction to X-ray CT, providing the necessary context. Subsequently, elsa is introduced through outlining the high-level design as well as the main classes. In the following chapter, the compile-time programming technique of ETs is explained and how it can be applied within elsa. A performance evaluation concludes the chapter. After surveying existing GPU-based vector arithmetic libraries, a custom solution called *Quickvec* is developed and benchmarked. Internally, it builds on top of the work presented for the ETs. The final chapter summarizes the added features, before discussing their limitations and providing links for future work.

2. Fundamentals of X-ray Computed Tomography

This chapter provides a short general introduction to X-ray CT, summarizing the technical progress from the discovery of X-ray radiation towards the development of high resolution X-ray CT scans.

2.1. X-ray Imaging

X-rays refer to electromagnetic radiation in the high-energy spectrum with wavelengths in the range from 0.01 to 10 nm. They were discovered in 1895 by the scientist Wilhelm Conrad Röntgen [Rön96]. Consequently, this radiation, invisible to the human eye, is also known as Röntgen radiation.

Due to their high-energy nature, X-rays are able to penetrate objects that are opaque to the human eye. However, they not only traverse the object but also interact with its matter. This interaction, also called modulation, is what enables X-ray imaging. Equivalent to photography, X-ray imaging captures the differences in modulation leading to results like the famous first radiograph of Röntgen's wife hand in Figure 2.1. The image shows the effect of stronger absorption in the bones of the hand compared to the soft tissue surrounding it.

On an atomic scale, this effect is caused by photoelectric absorption where an X-ray photon is absorbed by an electron [AM11]. From a macroscopic point of view, the individual interactions can be modelled according to the *Beer-Lambert law*

$$I_s = I_0 \exp\left(-\int_L f(r)dr\right), \tag{2.1}$$

where I_s refers to the measured intensity, I_0 to the initial intensity of the X-ray source, $f(r)$ corresponds to the *attenuation coefficient* at location $r \in \mathbb{R}^3$ and L is the traversed path [Buz11]. A large attenuation coefficient indicates dense matter causing strong absorption. For a constant $f(r) = c$, the intensity decays exponentially along the line L.

The main drawback from basic X-ray imaging is its projective nature: The measured intensity corresponds to the accumulated absorption after the ray has completely traversed the object. Therefore, all depth information is lost. This limitation is what CT overcomes.

Figure 2.1.: Radiograph of a human hand with a ring on the third finger [Rön96].

2.2. Tomographic Reconstruction

The goal of X-ray CT using absorption contrast is to reconstruct the attenuation coefficients $f(r)$ at all locations from the measured intensities.

In 1963, Allan M. Cormack formalized a method to describe the inversion of the imaging process for X-ray imaging. It requires taking measurements of the object from many different angles and then combining the information to reconstruct the attenuation coefficients [Cor63]. This setup is illustrated for the two-dimensional (2D) case in Figure 2.2. The detector is rotated together with the radiation source around the object. In the following years, Sir Godfried Hounsfield developed the first real-world apparatus for CT based on the work of Cormack [Hou73]. Nowadays, X-ray CT scanners are a staple in institutions like hospitals all around the world.

2.3. Tomographic Reconstruction as an Inverse Problem

An *inverse problem* can be characterized as trying to determine the causes from the effects [EHN96]. From a mathematical perspective, tomographic reconstruction is an example of such an inverse problem. The *forward problem* can be formulated as

$$\mathcal{A}(f) = g \tag{2.2}$$

where $\mathcal{A} : V \to W$ is the linear operator mapping from function space V to W and $f \in V$, $g \in W$ are members of their respective spaces. However, the quantity of interest

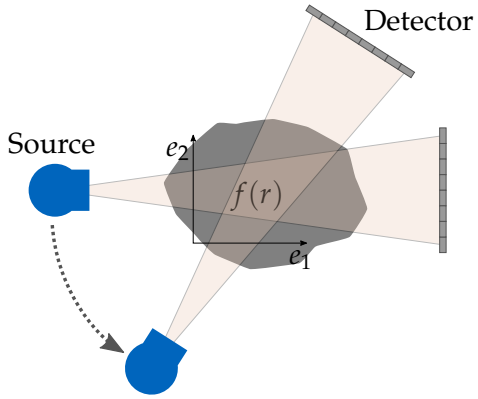


Figure 2.2.: Schematic CT setup. Multiple measurements are taken from different angles in a circular trajectory to reconstruct the attenuation coefficient $f(r)$ at all locations within the object.

is computed by the inverse problem

$$f = \mathcal{A}^{-1}(g), \tag{2.3}$$

assuming the operator $\mathcal{A}$ is invertible. Transferring these definitions to the use-case of X-ray CT using absorption contrast, $f : V \to \mathbb{R}$ is the function describing the attenuation coefficient at location $r \in V$, where $V \subset \mathbb{R}^3$. g corresponds to the measured intensity signal assuming some detector geometry.

The inverse problem of X-ray CT falls into the category of *ill-posed* problems as it does not fulfill the requirements for a *well-posed* problem as defined by Hadamard [Had02]. A well-posed problem requires the solution to be unique which might not always be the case. Another aspect is that small measurement errors can cause large deviations in the solution. An ill-posed problem is especially challenging to solve.

One approach to deal with the presented inverse problem is to directly discretize it and then apply linear algebra or optimization techniques. Analytical approaches like *filtered backprojection* are not further discussed in this thesis. They are limited to standard detector geometries and therefore not a research topic within the elsa software framework [Her09].

Discretization is achieved through the series expansion technique [Her09]. Using pixel-wise basis functions, the continuous signals f as well as g can be represented by finite dimensional vectors $x \in \mathbb{R}^n$ and $y \in \mathbb{R}^m$ respectively. Using the discretized signals and the Beer-Lambert law from (2.1), the measurement model represented by the linear operator $\mathcal{A}$ is derived as the *system matrix* $A \in \mathbb{R}^{m \times n}$. Each row of A encodes the linear combination of attenuation coefficients x_i resulting in a single detector pixel measurement of y_j. The system matrix therefore incorporates all the information about the CT geometry like source and detector positions or other source properties.

The discretized forward problem, also called *forward projection*, becomes a simple

system of linear equations

$$Ax = y. \tag{2.4}$$

Its counterpart, the backward problem, known as *backward projection*, turns into

$$x = A^T y. \tag{2.5}$$

Through optimizing the squared error

$$\arg\min_x \tfrac{1}{2}\|Ax - y\|_2^2 \tag{2.6}$$

with a gradient-based technique like gradient descent, an iterative solution can be found. Note that directly determining x in (2.5) is not possible due to discretization errors.

Another important point to consider is the size of the matrix A. A typical three-dimensional (3D) reconstruction volume has $n = 1024^3$ voxels. All of the projections taken together are in the same order of magnitude, resulting in $m = 1024^3$ measurements. This means A has $m \times n = 2^{60}$ entries, requiring multiple exa-bytes of memory for storing it. As such amounts of memory are beyond the technical possibilities in the foreseeable future, A is not stored directly but only computed when needed.

3. The elsa Software Framework

The following chapter provides a short introduction to elsa, the tomographic reconstruction software which is extended in this thesis [LHF19]. After discussing its high-level design and motivation, the core classes as well as as a 2D reconstruction example is given. The source code for elsa can be found in a public repository at `https://gitlab.lrz.de/IP/elsa`.

3.1. Motivation

The main motive for developing elsa is to provide a flexible, high performance software framework which allows to solve a wide variety of reconstruction tasks. In practice, this is achieved through an object-orientated approach, leveraging modern C++17. It traces its heritage back to the (nonpublic) *CampRecon* software which was refactored, improved and made available under a permissive license [WVL14]. The main advantage of elsa is its flexibility compared to other available reconstruction software. It follows the general mathematical concept for solving linear inverse problems developed in Section 2.3, making it suitable for a wide range of reconstruction tasks.

3.2. Architecture

As stated before, elsa follows an object-orientated style to translate the general mathematics involved in solving a linear inverse problem into code. Therefore, all main concepts from Section 2.3 have been translated into classes. The most important ones for the work presented in this thesis are:

- `DataContainer`. It handles the discrete representation of signals in a vector-like container. A `DataDescriptor` member represents meta information, allowing transformations of the linearized data into a higher dimensional representation. Member functions and operator overloading allow mathematical operations on the stored data.

- `LinearOperator`. It is the base class corresponding to the linear operator $\mathcal{A}$ introduced in Section 2.3. The `LinearOperator` can be made up of multiple operators chained together. Through the `apply` and `applyAdjoint` functions, it provides on-the-fly computations for the forward and backward projection which includes computing the system matrix A.

- `Problem`. This is the abstract base class representing a generic optimization problem, as stated through (2.6). Member functions provide access to the current solution, gradient and Hessian which can then be used in a solver. Additionally, the optimization problem can be extended through regularization terms.
- `Solver`. It serves as the abstract base class for iteratively solving an optimization problem. Instances are constructed with objects of the `Problem` class as an argument.

All classes are templated on a numerical type for switching between single and double precision floating point numbers. Note that this detail is omitted in the subsequent treatment of elsa for simplicity and single precision floats are assumed. A generic flowchart illustrates the steps for solving a linear inverse problem in Figure 3.1.

For the `LinearOperator`, two GPU-accelerated implementations exist. They are called Joseph's method and Siddon's method, referring to their original creators [Jos82] [Sid85].

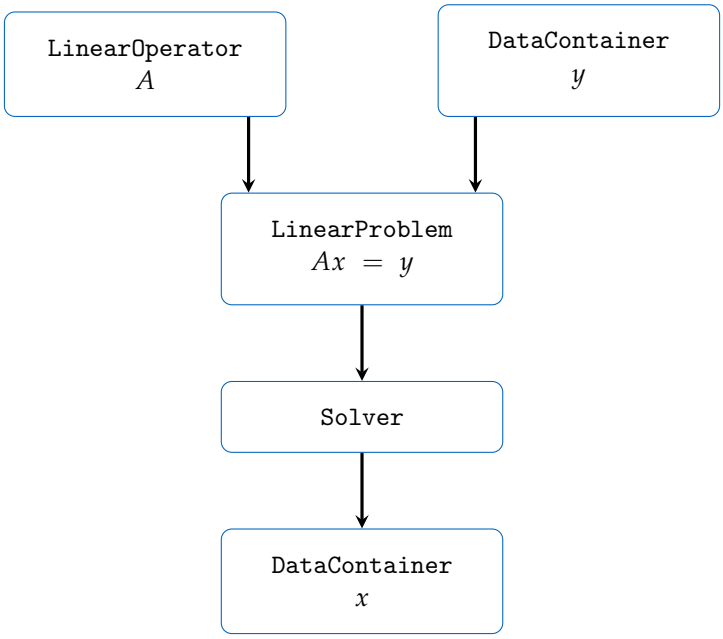


Figure 3.1.: Flowchart of a generic reconstruction task in elsa.

3.3. The `DataContainer` Class

As the `DataContainer` plays a significant role for the next chapters, it is discussed in further detail. The container itself does not directly store and manipulate data. It only delegates this work internally, using an abstract base class pointer of type `DataHandler*`. Polymorphism allows using different derived handlers. The motivation for this is to provide the user with a consistent interface through the `DataContainer` regardless of the

underlying storage mechanism implemented in the DataHandler. The member variable _dataHandlerType indicates the currently used DataHandler. Together with overloaded mathematical operators like +, -, / and *, the DataContainer interface enables intuitive (element-wise) vector arithmetic as shown in Code 3.1. Additionally, operations with scalars as well as reduction operations like different norms or the dot product between two DataContainers are available. Figure 3.2 summarizes the class in Unified Modelling Language (UML) notation.

Code 3.1: Arithmetic operations using DataContainers

```
1 DataContainer a(vector1);
2 DataContainer b(vector2);
3 DataContainer result = log(a) * 1.24 + b;
4 float norm = result.l2norm();
```

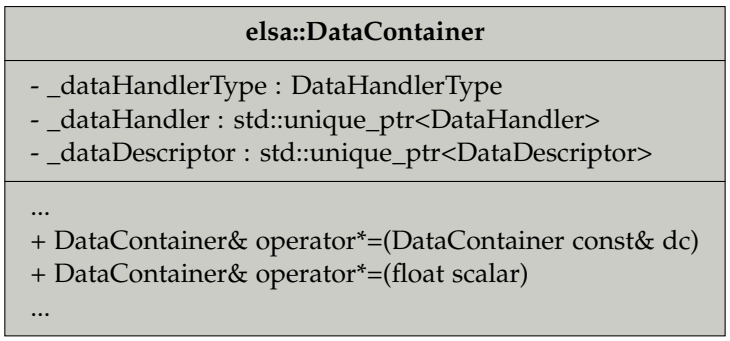


Figure 3.2.: Overview of the DataContainer class with selected members.

3.4. The DataHandler Class

This abstract base class encapsulates the handling process of the data being stored. It provides the common interface for the DataContainer via pure virtual member functions. In the current state of elsa, there are four derived child classes:

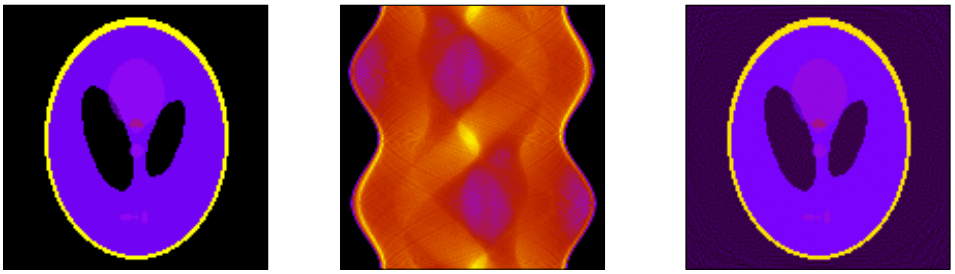
- DataHandlerCPU. Data resides in the main memory in the form of an Eigen::Matrix.
- DataHandlerMapCPU. Non-owning data mapping of main memory through an Eigen::Map.
- DataHandlerGPU. Data resides in the GPU memory using a quickvec::Vector.
- DataHandlerMapGPU. Non-owning data mapping of GPU memory with a quickvec::Vector.

The extension with the GPU-based handlers as well as the development of the Quickvec library is part of Chapter 5. The CPU handlers are wrappers around the existing linear algebra library Eigen. It provides the actual implementation of any numerical operation through highly optimized and efficient code [GJ+10].

3.5. A Reconstruction Example

A specific reconstruction example using elsa is given in Code 3.2. First, a 2D Shepp-Logan phantom is created as a test object, with a resolution of 128 by 128 pixels [SL74]. Then, the geometry of the CT setup is generated, encoding location and properties of the X-ray source, object and detectors. Having this information available, the projector corresponding to the discretized linear operator $\mathcal{A}$ is defined, using the GPU-accelerated implementation of Joseph's method. As a next step, the X-ray scan is simulated, solving the forward problem through applying the projector to the object. All projections stacked on top of each other result in the *sinogram*, which will act as DataContainer y in the algorithm flowchart shown in Figure 3.1. Finally, the reconstruction problem is created with the sinogram y and the linear operator $\mathcal{A}$ as a weighted least squares problem and solved using the method of conjugated gradients.

In Figure 3.3, the original test image (a), the intermediate sinogram (b) and the reconstructed phantom (c) is shown.



(a) Original phantom (b) Simulated sinogram (c) Reconstructed phantom

Figure 3.3.: A reconstruction process. The color scale shows the single-valued attenuation coefficient.

Code 3.2: 2D reconstruction example using elsa

```
#include "elsa.h"
using namespace elsa;

int main()
{
    // generate 2D phantom
    IndexVector_t size(2);
    size << 128, 128;
    auto phantom = PhantomGenerator::createModifiedSheppLogan(size);

    // generate full circle trajectory with 128 projections
    index_t noAngles{128}, arc{360};
    auto [geometry, sinoDescriptor] =
        CircleTrajectoryGenerator::createTrajectory(
            noAngles, phantom.getDataDescriptor(),
            arc, size(0) * 100, size(0));

    // setup operator for 2d X-ray transform
    JosephsMethodCUDA projector(phantom.getDataDescriptor(),
                                *sinoDescriptor, geometry);

    // simulate the sinogram
    auto sinogram = projector.apply(phantom);

    // setup reconstruction problem
    WLSProblem problem(projector, sinogram);

    // solve the reconstruction problem in 100 interations
    CG solver(problem);
    auto reconstruction = solver.solve(100);
}
```

4. Compile-Time Programming for elsa

This chapter explores how compile-time programming can be applied to elsa for increased efficiency. First, a short general introduction to templates and compile-time programming using C++ is given. Second, the specific programming problem of supporting numeric operations on whole array objects is discussed. As expression templates offer an elegant solution to this problem, their implementation and application to elsa is described. Benchmarks of the implemented features conclude the chapter.

4.1. Templates in C++

Templates give C++ programmers the ability to write generic code acting on many types *without* discarding type safety. They are a core feature and heavily used throughout the C++ standard library. In recent years, language extensions like *compile-time if* or *variadic templates* have further increased their usefulness [Van17]. Templates offer one possibility for compile-time programming where computations are done at compile-time as opposed to at run-time.

4.2. Numeric Operations on Whole Array Objects

The `DataContainer` class, introduced in Chapter 3, can be seen as an example of a wrapper class around array-like numerical data. Naturally, one wants to perform numeric operations with `DataContainers` such as addition, multiplication, division and subtraction.

An example of such a numerical expression is

$$y = a * x + y \tag{4.1}$$

or in element-wise notation

$$y_i = a * x_i + y_i \tag{4.2}$$

where a is a scalar, x, y are `DataContainers` and $i = 1, \ldots, n$ with n being the number of elements. This specific computation is commonly known as single precision a times x plus y (saxpy) and specified in the Basic Linear Algebra Subprograms (BLAS) [Bla+02].

There exist two obvious methods for computing saxpy. First, write a special function as shown in Code 4.1. It uses a single loop over all indices calculating the result. Second, implement overloaded operators (see Code 4.2 for an example using `operator+`). Then, the operators can be chained together to perform the desired calculation.

Code 4.1: Saxpy computation using an explicit function

```
void saxpy(DataContainer const& x, float alpha, DataContainer& y)
{
    for (int i = 0; i < y.getSize(); ++i) {
        y[i] = alpha * x[i] + y[i];
    }
}

int main() {
    ...
    saxpy(x, 1.2, y);
    ...
}
```

Code 4.2: Saxpy computation using operator overloading

```
DataContainer operator+(DataContainer left, DataContainer right) {
    DataContainer result(left);
    for (int i = 0; i < y.getSize(); ++i) {
        result[i] += right[i];
    }
    return result;
}
...

int main() {
    ...
    y = 1.2 * x + y;
    ...
}
```

At first glance, the overloaded operators seem like the better solution. They allow writing arbitrary numeric expressions without the need for explicit functions. Plus, the notation is very intuitive as it resembles the standard mathematical representation shown in (4.1). However, this solution has a major drawback. Consider the `saxpy` function from Code 4.1: It consists of a single loop covering the number of elements n only once. Using the user-defined operators from 4.2, there will be an intermediate result from the `1.2 * x` computation including traversing n elements. Then this is added to `y`, resulting in an additional loop over n elements. The allocation of memory for the temporary as well as the extra read and write operations increase both run time and memory requirements. For large n, the extra memory might not even be available.

A third solution is to use in-place operations instead, as demonstrated in Code 4.3. However, in the saxpy example, this does not improve the performance. The number of

read and write operations is the same as with operator overloading, as it also requires a temporary variable for the intermediate result. Furthermore, the notation obfuscates the intent. ETs can solve this dilemma. They allow for intuitive mathematical notation while at the same time avoiding the computational overhead.

Code 4.3: Saxpy computation using in-place operations

```
1 ...
2
3 int main() {
4     ...
5     DataContainer temp(x);
6     temp *= 1.2;
7     y += temp;
8     ...
9 }
```

4.3. Expression Templates

The technique of ETs was invented independently of each other by Todd Veldhuizen and David Vandevoorde [Vel95] [Van03]. Since then, ETs have been extensively used for array-like types and recently also in other domains like the *Boost Lambda Library* [Van17].

The idea of ETs is to save the computations to be done as a type of a lightweight `Expression` object at compile-time. This object can then be evaluated in a single pass and only when needed. In constrast, the eager evaluation taking place with operator overloading in Section 4.2 leads to a temporary intermediate result for each operator. With ETs, the saxpy computation

```
1.2 * x + y;
```

results in a type of

```
Expression<Addition,
           Expression<Multiplication,
                      float,
                      DataContainer>,
           DataContainer>.
```

The `Expression` class is declared as

```
template <typename Callable, typename... Operands>
class Expression;
```

where `Callable` represents a generic operation to be performed on the `Operands`. The type of the returned `Expression` contains all information: An addition is to be performed between a `DataContainer` and the result of an inner `Expression` which consists of multiplying a scalar with a `DataContainer`.

Using variadic template parameters for the `Operands` allows the `Expression` to accommodate computations with either one or two `Operands`. Note that `Operand` can be either a scalar, `DataContainer` or `Expression`. Consequently, nesting is possible which creates a recursive tree-like structure of the necessary computations. For saxpy such a tree is illustrated in Figure 4.1.

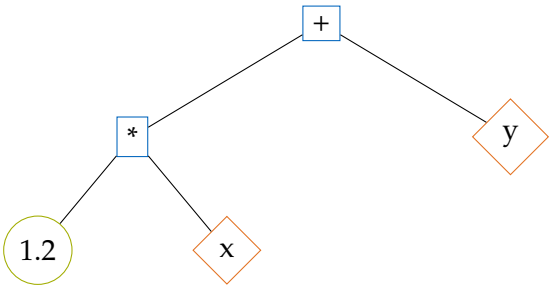


Figure 4.1.: Expression tree for saxpy. The blue rectangles represent an `Expression` object, the green circle a scalar and the orange diamonds a `DataContainer`.

The `Expression` is only evaluated when the results of the computation are needed, for example when assigning to a `DataContainer`. In the saxpy example, first the inner

```
Expression<Multiplication, float, DataContainer>
```

is computed before using this result in the calculation of the outer

```
Expression<Addition, Expression<...>, DataContainer>.
```

4.4. Expression Templates for elsa

As mentioned before, elsa internally uses the linear algebra library Eigen for doing computations based in main memory. The classes `DataHandlerCPU` and `DataHandlerMapCPU` represent wrappers around the `Eigen::Matrix type`. Eigen itself already implements the technique of ETs for efficient computations [GJ+]. Consequently, the objective is to develop a framework which allows utilizing the Eigen provided ETs within elsa.

The approach is to implement ETs for elsa which wrap around the Eigen computations. Hence, the `Expression` objects are not doing computations when evaluated but only allow the construction of Eigen internal ETs which are then immediately evaluated.

The subsequently developed ETs are based on code presented at *CppCon 2019* by Bowie Owens [Owe19]. In contrast to the elsa scenario, his implementation performs the actual computations. However, the general framework is still applicable.

4.4.1. The `Expression` Class

The `Expression` class represents temporary objects which save the computation to be done in their type information at compile-time as introduced in Section 4.3. Using UML

notation, the class is summarized in Figure 4.2. The class saves a `Callable` object as well as its `Operands` in member variables. Both are necessary when constructing an `Expression` object. Additionally, the `eval` function triggers the evaluation.

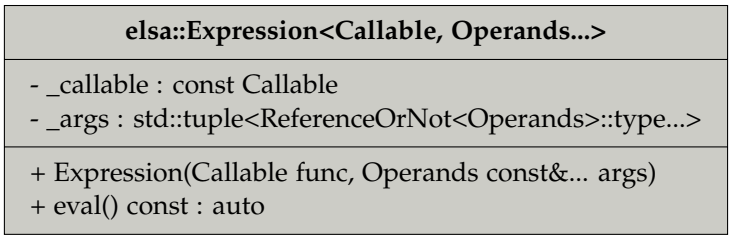


Figure 4.2.: Overview of the `Expression` class.

Of interest is the helper class `ReferenceOrNot<T>` which provides a type definition through its member `ReferenceOrNot<T>::type`. This type is either a `const` reference to `T` or `T` itself, allowing the `Expression` to refer to `DataContainers` by reference but also take scalars as well as `Expressions` by value in its `_args` member tuple. The reason for this is that `Expression` objects might be temporary to a very narrow scope. Referring to them by value extends their lifetime until needed and consequently prevents having dangling references.

`Expression` instances are instantiated in the overloaded operators. Consequently, the code presented in Code 4.2 changes to Code 4.4. Returning an `Expression` instead of the actual result as a `DataContainer` enables lazy evaluation. As a `Callable`, a generic lambda is used, which applies Eigen syntax internally because it will directly operate on `Eigen::Matrix` types.

Code 4.4: Overloaded operators returning `Expression` type

```
auto operator+(DataContainer const& lhs, DataContainer const& rhs)
{
    auto Addition = [](auto const& left, auto const& right) {
        return (left.array() + right.array()).matrix();
    };
    return Expression{addition, lhs, rhs};
}
```

4.4.2. Evaluating an `Expression`

For evaluating an `Expression`, the member function `eval` must be executed. Internally, this triggers calling the `Callable` with the `Operands` as arguments as shown in Code 4.5. Note that using the generic lambda named `callCallable` is necessary to use pack expansion again. The overloaded function `evaluateOrReturn` plays a key role by implementing different behavior based on the type of `Operand`:

- `DataContainer`. The function returns the underlying `Eigen::Matrix` through its `_dataHandler` member.
- `Expression`. The function calls `eval` of the nested `Expression`. This descends into the evaluation tree and triggers the computation from bottom up.
- Scalar. The function returns the scalar by value.

Code 4.6 shows all three functions. The overload resolution is achieved through compile-time predicates using `std::enable_if_t`.

Code 4.5: Evaluating an `Expression`

```
template<typename Callable, typename... Operands>
auto Expression<Callable, Operands>::eval() const
{
    auto const callCallable = [this](Operands const&... args) {
        return _callable(evaluateOrReturn(args)...);
    };
    return std::apply(callCallable, _args);
}
```

Code 4.6: Overloaded `evaluateOrReturn` functions

```
template <class Operand,
          std::enable_if_t<isExpression<Operand>, int> = 0>
constexpr auto evaluateOrReturn(Operand const& operand)
{
    return operand.eval();
}

template <class Operand,
          std::enable_if_t<isArithmetic<Operand>, int> = 0>
constexpr auto evaluateOrReturn(Operand const operand)
{
    return operand;
}

template <class Operand,
          std::enable_if_t<isDataContainer<Operand>, int> = 0>
constexpr auto evaluateOrReturn(Operand const& operand)
{
    return operand._dataHandler->accessData();
}
```

Calling `eval` on the `Expression` object returned from the saxpy example

```
1.2 * x + y;
```

triggers the application of `evaluateOrReturn` to the `Operands` of the outer

```
Expression<Addition, Expression<...>, DataContainer>.
```

As the first operand is the nested inner

```
Expression<Multiplication, float, DataContainer>,
```

the `eval` function is called. This causes `evaluateOrReturn` to produce the `Eigen::Matrix` and the scalar value of `1.2`. Those results are used as an input to the generic lambda

```
auto Multiplication = [](auto const left, auto const& right) {
    return (left * right.array()).matrix();
};
```

with argument `left` being the scalar and argument `right` being the `Eigen::Matrix`. However, the generic lambda does not perform the computation as Eigen itself implements ETs. Thus, the operation returns an Eigen specific temporary which is returned to the outer `Expression`. After `evaluateOrReturn` has returned the `Eigen::Matrix` from the second operand of the outer `Expression`, the generic lambda for addition

```
auto Addition = [](auto const left, auto const& right) {
    return (left.array() + right.array()).matrix();
};
```

once more creates an Eigen temporary. It represents the work to be done for both the multiplication as well as the addition. The calculation is only performed when the temporary is assigned to an `Eigen::Matrix` or used to construct a new `Eigen::Matrix`.

4.4.3. Extending the `DataContainer` Class

Instead of manually calling `eval`, the computation should be triggered automatically when:

(a) An assignment operation of an `Expression` to a `DataContainer` takes place.

(b) A new `DataContainer` is constructed with an `Expression`.

Implementing this means extending the `DataContainer` class through appropriate member functions. Code 4.7 shows the assignment operator triggering the evaluation. The result is written directly into the underlying `Eigen::Matrix` via the `_dataHandler` which in turn causes the evaluation of the Eigen internal ETs. An additional constructor evaluates an arbitrary `Expression` and then passes the result to the existing constructor taking an `Eigen::Matrix` as shown in Code 4.8. Both functions are templated to take arbitrary `Expression` types.

Code 4.7: Automatic evaluation of `Expressions` through the assignment operation

```
template <typename Expression>
DataContainer& operator=(Expression const& expression)
{
    _dataHandler->accessData() = expression.eval();
    return *this;
}
```

Code 4.8: Constructing a new `DataContainer` using an `Expression`

```
template <typename Expression>
DataContainer(Expression const& expression)
    : DataContainer(expression.eval())
{}
```

4.5. Benchmarks and Results

For benchmarking, the *Catch2* framework is used. It runs code repeatedly and measures the execution time in each run. In the end, reports are created with the average time as well as the standard deviation for each benchmarked section. The test environment is a server with the specifications summarized in Table 4.1. All runs use single precision floats as the underlying numerical type and are repeated twenty times.

Specification	Version / Value
CPU	Intel Xeon E5-2687W
GPU	Nvidia RTX 2080Ti
Main memory	128 GiB
Linux Kernel	4.15.0-88-generic
Operating System	Ubuntu 18.04.4 LTS
Eigen commit	1affbe9...
elsa commit before ETs	4497388...
elsa commit after ETs	b3227b9...

Table 4.1.: Server setup for ETs benchmarks.

4.5.1. Expression Benchmarks

The run time of simple numeric expressions like saxpy are measured using the five example terms:

(a) `x = x * y`

(b) `x = x * y + y`

(c) `x = x * y + y / z`

(d) `x = x * y + y / z + x`

(e) `x = x * y + y / z + x * z`

where `x`, `y` and `z` are array-like containers with the number of elements n varied from 256^3 to 2048^3. For all containers, pseudorandom numbers in the range of 0 to 100 are used. The expressions are computed in three different scenarios:

1. elsa before the integration of ETs using a `DataContainer` for `x,y,z`.
2. elsa after the integration of ETs using a `DataContainer` for `x,y,z`.
3. Eigen directly using the `Eigen::Matrix` type for `x,y,z`.

The absolute results for expression (a) are reported in Figure 4.3. It is clear that before ETs are integrated, a large computational overhead exists even for a simple term whereas with ETs this can be avoided. The relative results using the Eigen scenario as a baseline in Figure 4.4 confirm the effectiveness for different problem sizes: Before ETs are introduced, the run times are around 250 % slower. With ETs, they are within 15 % of the baseline case.

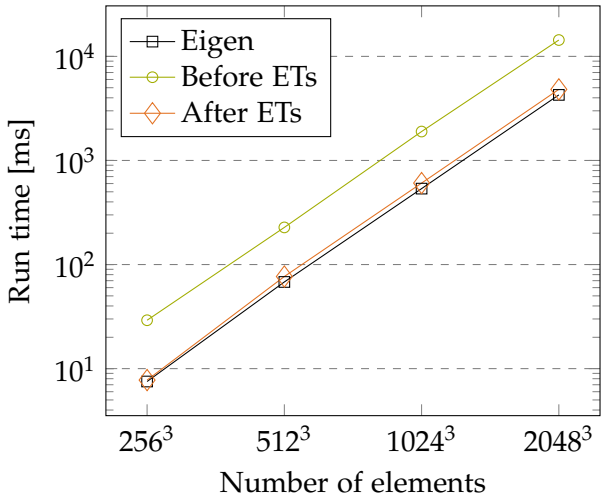


Figure 4.3.: Absolute mean run times computing `x = x * y`.

For expression (b), the relative results in Figure 4.5 display a similar picture. As the expression is more complicated than (a), the run time increases to over 600 % without ETs. Additionally, for the problem size of 2048^3, memory errors occur because there is not enough main memory for the allocation of the intermediate result. Adding the ETs solves the problem and keeps run times within 10 % of the baseline scenario.

Investigating expressions (c) to (e), confirms the findings: As the expressions get longer, the overhead without ETs grows whereas with ETs run times stay within 10 % of

the baseline scenario. This can be seen in Figure 4.6, 4.7 and 4.8. Again, memory errors occur for the largest problem size of 2048^3.

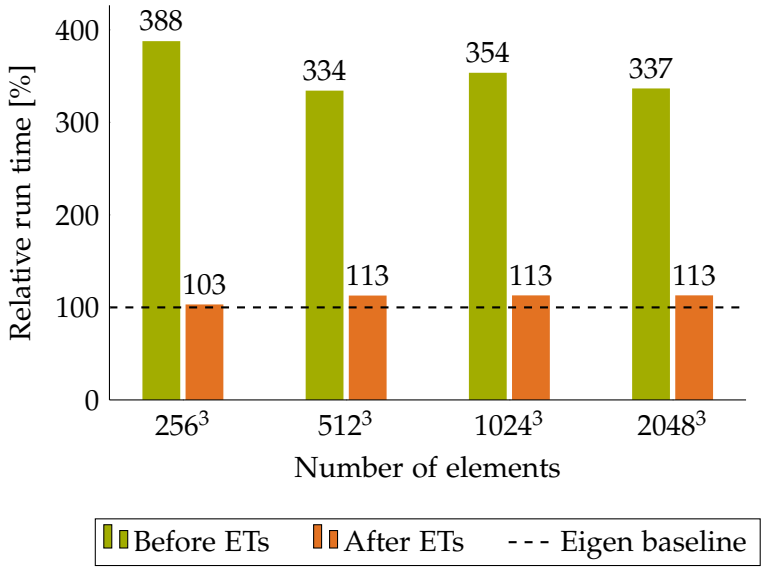


Figure 4.4.: Relative mean run times for computing `x = x * y`.

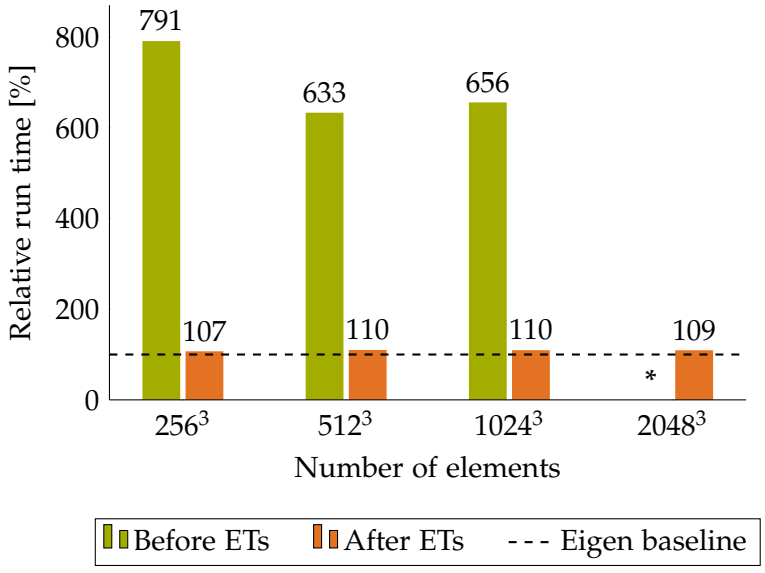


Figure 4.5.: Relative mean run times for calculating `x = x * y + y`. The * indicates a missing value due to memory errors.

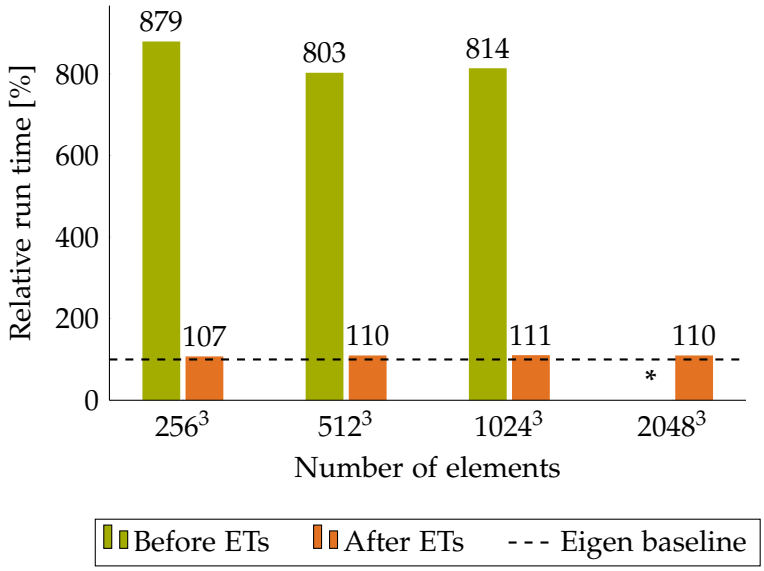


Figure 4.6.: Relative mean run times for calculating `x = x * y + y / z`. The * indicates a missing value due to memory errors.

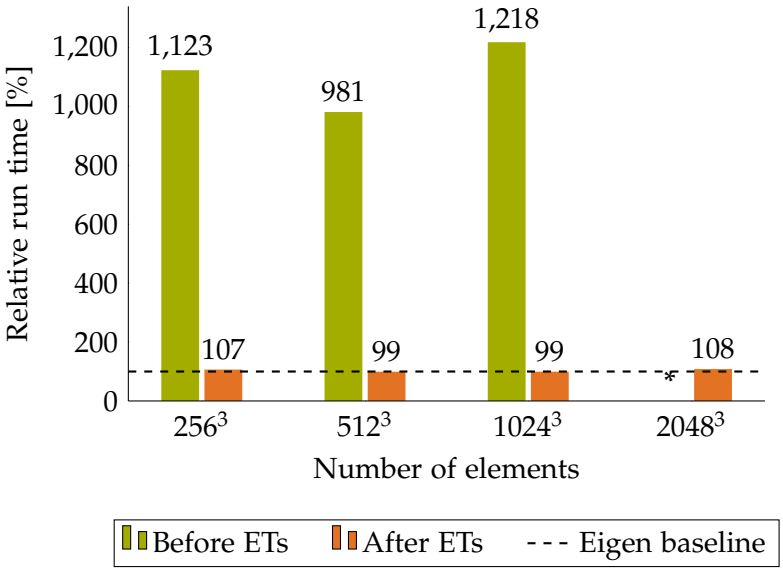


Figure 4.7.: Relative mean run times for calculating `x = x * y + y / z + x`. The * indicates a missing value due to memory errors.

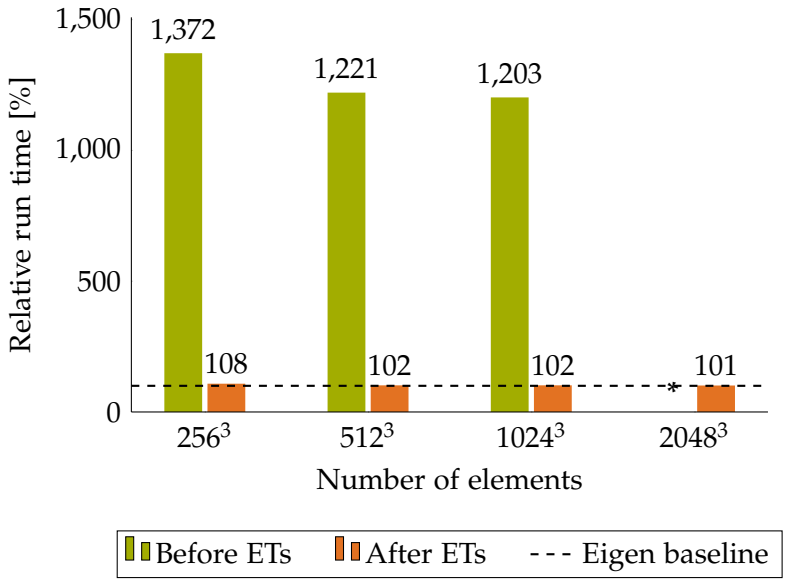


Figure 4.8.: Relative mean run times for calculating `x = x * y + y / z + x * z`. The * indicates a missing value due to memory errors.

In Figure 4.9, the connection between expression complexity and run time can be seen more clearly. The plot shows the relative run times for the single problem size of 1024^3 across all expressions. The run time grows from 354 % to 1203 % compared to the Eigen baseline scenario. The absolute mean run times including the standard deviations for all benchmarks can be found in Table A.2.

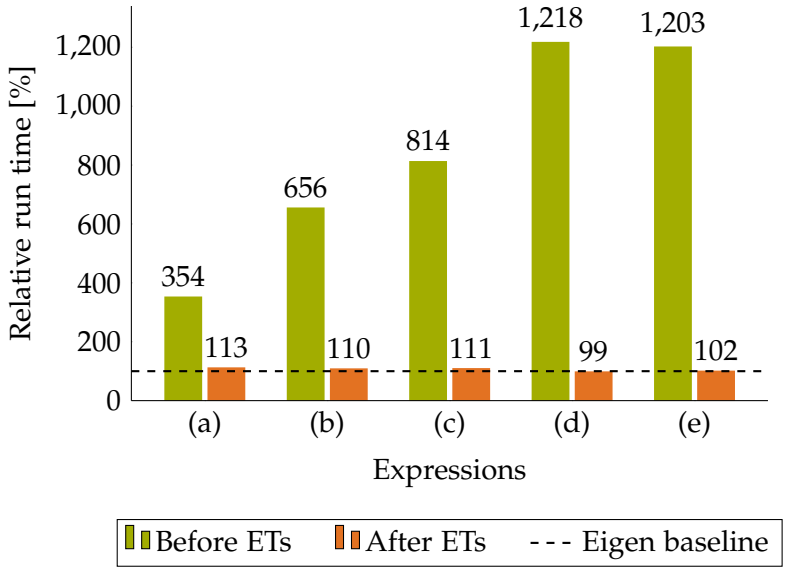


Figure 4.9.: Relative mean run times for different expressions (a) to (e) using a size of 1024^3.

4.5.2. Memory Consumption

The memory consumption is tracked using the addition example

```
x = x + y
```

where `x` and `y` are `DataContainers`. The heap allocations are measured using the profiling tool *Valgrind*. The number of floats in the `DataContainers` is set to 128^3. The results confirm the expected behavior. Before the expression templates, three large heap allocations take place, each taking up 8 MiB: Two of them when constructing the `DataContainers`, one when the operator `+` is executed for allocating the temporary resulting in a total maximum heap usage of 24 MiB.

After the introduction of the ETs, only two large heap allocations occur when the `DataContainers` are constructed. The resulting total maximum heap usage is 16 MiB. That corresponds to a reduction of more than 33 %. For longer expressions, the memory savings are increasing which directly affect the run time as shown in the previous section in Figure 4.9.

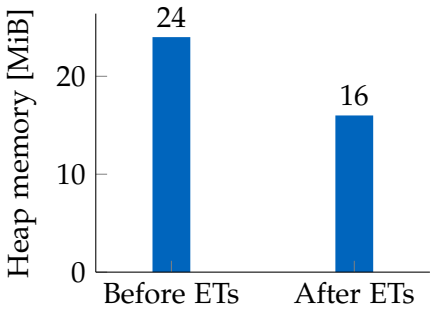


Figure 4.10.: Heap memory consumption before and after implementing ETs for calculating `x = x * y`.

4.5.3. A Full Reconstruction Task

A reconstruction example like presented in Section 3.5 is benchmarked using a Shepp-Logan phantom with 256^3 to 1024^3 elements and a matching number of projections taken in a circular trajectory. As a projector, Joseph's method is used and as a solver, the method of conjugated gradients. Averaging twenty iterations, the relative mean run times are shown in Figure 4.11. There is a speedup ranging from 41 % for the small problem size to 28 % for the large one.

The decreasing efficiency gains can be attributed to the fact that the fraction of execution time spend within the `apply` and `applyAdjoint` functions of the projector increases for larger problem sizes. Both functions consist of handwritten GPU code which does not benefit from the introduced ETs. Run time reductions are only achieved in the remaining part of the iterative solver loop. This fact is illustrated for size 1024^3 in Figure 4.12. All the absolute run times are given in Table A.1.

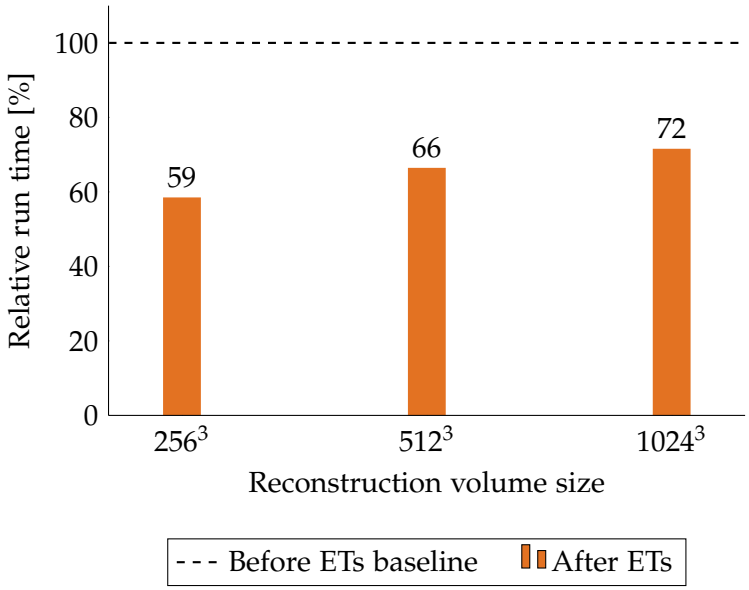


Figure 4.11.: Relative mean run times for one solver iteration in a full reconstruction task. The baseline scenario is elsa before the ETs are added.

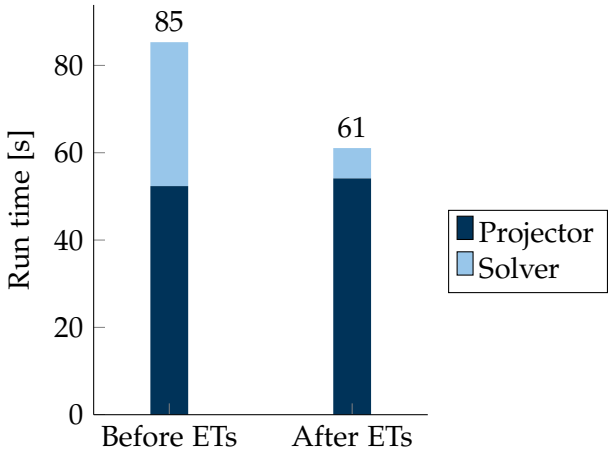


Figure 4.12.: Full reconstruction run times per iteration of the conjugated gradients solver. The time is split into the two parts: First, the amount spend within the projectors' `apply` and `aplyAdjoint` functions and second the time in the remaining solver code.

5. Heterogeneous Computing for elsa

This chapter outlines efficiency gains in elsa through leveraging the computing power of GPUs for basic numeric operations. Specifically, the `DataContainer` class is extended through a GPU-based `DataHandler`. To achieve this, a general GPU library for arbitrary vector computations called Quickvec is developed, before its integration into elsa is discussed. Run time benchmarks comparing the CPU versus the GPU `DataContainer` conclude the chapter. The source code for Quickvec can be found in a public repository at `https://gitlab.lrz.de/IP/quickvec`.

5.1. General-Purpose Computations with GPUs

The original purpose of a GPU is rendering images for displaying graphical output. As the raw computational power of GPUs started to surpass CPUs at the turn of the millennia, researchers started to look into the possibility of using those devices for general computations. In the beginning, this was achieved through existing graphics Application Programming Interfaces (APIs) like *OpenGL* or *DirectX*. However, as this was cumbersome and error-prone, it did not reach widespread adoption. That changed with the introduction of an easy to use API tailored for general-purpose computations called Compute Unified Device Architecture (CUDA) for Nvidia GPUs. Since then, many computational tasks have benefited from the additional processing power of GPUs, with one of the most recent ones being the surge in deep learning [Coo12].

A GPU is designed for parallel processing tasks. It can hide memory latencies in computations. On the other hand, a CPU is mainly tailored for serial code execution which is mirrored in its hardware. It has multiple caches and branch prediction units. A problem at hand must have a certain degree of parallelism in order to benefit from using a GPU. If not, then a CPU might be more efficient for solving it, despite possessing considerably less processing power [Coo12].

As stated before, CUDA represents one possibility to program a GPU for general-purpose computations. The only notable alternative is *OpenCL*, which can work with both AMD and Nvidia chips. However, CUDA provides better performance in many cases. As performance is key for CT and the reconstruction servers at the Computational Imaging and Inverse Problems group run on Nvidia GPUs, the following implementation details are based on CUDA.

5.1.1. Compute Kernels using CUDA

GPUs using CUDA are programmed through specialized functions called *compute kernels*

or simply *kernels*. They resemble regular functions, except that they are prefixed with a `__global__` or `__device__` qualifier and certain keywords like `threadIdx` have special meanings. For the saxpy calculation, such a kernel is given in Code 5.1. There is no need for a loop over all elements of the arrays, as the kernel is automatically executed in parallel by hundreds of threads. Each thread is identified by its thread and block ID which together map to a certain element `i` of the arrays. The calculation is performed for this element, unless it is out of the array's range.

A kernel cannot be called like a regular function, but instead it must be launched using the triple angular bracket syntax as shown in Line 11 of Code 5.1. The arguments in the angular brackets specify the number of blocks and the number of threads per block.

Code 5.1: Examplary saxpy kernel

```
1  __global__
2  void saxpy(int n, float a, float* x, float* y)
3  {
4      int i = blockIdx.x*blockDim.x + threadIdx.x;
5
6      if (i < n)
7          y[i] = a*x[i] + y[i];
8  }
9
10 ...
11 saxpy<<<4096,256>>>(N, 2.0, d_x, d_y); // Perform SAXPY
12 ...
```

5.2. Efficiency Through a GPU `DataContainer`

The efficiency of elsa is expected to increase through a GPU `DataContainer` due to the following two characteristics:

1. Many computations using the `DataContainer` are parallel problems, where each element can be computed independently of each other. Most notably, those are the operators `+`, `-`, `*` and `/`, defined between `DataContainers` and scalars. Those parallel problems are well suited for GPUs.

2. Derived classes from the `LinearOperator` base class already provide GPU-based implementations for computing the forward and backward projection. Hence, using a CPU `DataContainer` results in data transfers to and from the GPU memory in each iteration of a solver. Directly storing the data on the GPU prevents these data transfers.

In elsa, the `DataContainer` internally delegates the computations to its corresponding `DataHandler`. Consequently, the objective is to implement a `DataHandlerGPU` as a

counterpart to the existing `DataHandlerCPU`. The end user should be concerned with the decision of CPU versus GPU as little as possible, which is achieved through using the same `DataContainer` interface.

5.3. Existing Numerical GPU Libraries

The `DataHandlerCPU` represents a wrapper around the linear algebra library Eigen which is responsible for the actual computations. However, Eigen does not support GPU acceleration. Therefore, a different library needs to be used which

- supports element-wise operations between arrays,
- provides reduction operations like different norms,
- supports arbitrary expressions using ETs,
- enables intuitive mathematical notation,
- is widely used and well maintained.

Coming with the CUDA Software Development Kit (SDK) is a library called *Thrust* which mimics the behavior of the Standard Template Library (STL) on the GPU [BH12]. Its main drawback is that it does not support efficient arbitrary expressions due to its lack of ETs.

Using ETs on GPUs was first explored by Wiemann et al. [WWM11]. Their implementation generates a character string using the compile-time expression which is then at run-time compiled through the Nvidia Just-In-Time (JIT) compiler into a kernel. Their approach was further refined, resulting in the development of two notable linear algebra libraries: *VexCL* and *ViennaCL* [Dem+13] [Rup+16]. However, neither of them appears to be widespread.

Lastly, Wicht et al. created the *Expression Templates Library* for deep learning applications [WFH18]. Here, the researchers evaluate subexpression by pre-defined kernels. The drawback is that this again introduces temporaries. Moreover, the library does not seem to be widely used.

A lot of the complexity of the libraries mentioned before are caused by the fact that GPU device code only supported a small subset of C++ template metaprogramming when they were created. Luckily, that changed in the last years with the latest CUDA releases. Consequently, directly evaluating expressions in device code became possible, sidestepping string generated kernels. Breglia et al. explored this possibility through transferring expressions to the GPU and converting them to a GPU expression [Bre+13]. However, their work did not result in a stand-alone library.

Considering all the limitations of the above mentioned solutions, a custom library called Quickvec is developed in the next section. It further explores the approach of evaluating ETs directly on the GPU.

5.4. The Quickvec Library

In Chapter 4, a concise and elegant ETs implementation using modern C++ was introduced, based on the work of Owens [Owe19]. Its purpose is wrapping around the ETs provided by Eigen. As ETs are required in Quickvec for efficiency, many of the concepts developed in Chapter 4 can be adopted. However, the objective is different as the actual computations have to be performed by Quickvec.

The main idea is to transfer the `Expression` objects, which save to work to be done, to the GPU and evaluate them on the device index-wise using the available parallel processing power. In contrast to the work of Breglia et al., there is no need for conversion between the `Expression` on the CPU and the GPU.

The requirements for Quickvec do not include matrix-matrix or vector-matrix operations but only element-wise vector-vector and vector-scalar computations. That results in a much simpler treatment of complex expressions compared to a regular linear algebra library.

5.4.1. Motivation

In Code 5.1, a specific kernel for the saxpy computation is shown. Writing such a kernel for each individual expression is cumbersome. Therefore, one objective of Quickvec is to utilize ETs to generate kernels at compile-time. Evaluating an expression should then be possible through a templated kernel where each thread computes one element as demonstrated in Line 8 of Code 5.2.

Code 5.2: The kernel for evaluating generic `Expressions`

```
1  template <typename Expression>
2  __global__
3  void compute(size_t n, Expression* expression, float* result)
4  {
5      int i = blockIdx.x * blockDim.x + threadIdx.x;
6
7      if (i < n) {
8          result[i] = (*expression)[i];
9      }
10 }
```

5.4.2. The `Vector` class

The raw numerical data is stored in an array using the unified memory address space provided by CUDA. This memory space combines the main and the GPU memory, allowing the programmer to access device memory in host code. CUDA itself handles the necessary memory transfers and therefore reduces the complexity considerably.

To prevent memory errors and provide additional functionality, the raw array is wrapped into a class called `Vector` using the Resource Acquisition Is Initialization (RAII) idiom. Another important aspect is that `Vector` implements shallow copying because it enables the subsequently presented `Expression` class. Consequently, a shared pointer is needed which acts on the unified memory. As this feature is not available in the standard library, a custom shared pointer is added. It behaves like a `std::shared_ptr` with the only difference that it allocates and frees memory using the CUDA specific functions for the unified memory.

The class is summarized in Figure 5.1 using UML notation. The `eval` function evaluates an `Expression` into a `Vector` object. The process is explained in more detail in Section 5.4.4. The `operator[]` is implemented for both host and device access. For intuitive mathematical notation, the binary operators `+`, `-`, `*` and `/` and unary operators like `log` are overloaded with `Vector` types as their input. Like the overloaded operators in Chapter 4, they do not return a result but an `Expression` which stores the work to be done.

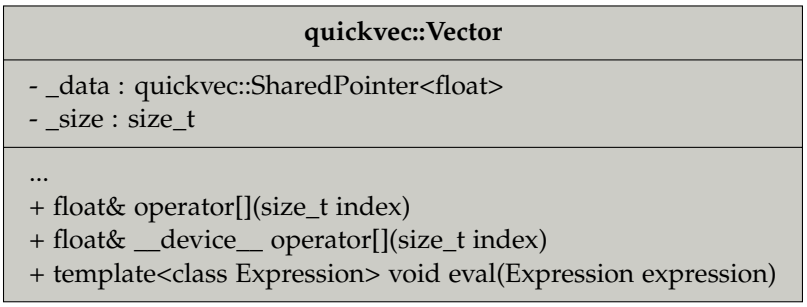


Figure 5.1.: Overview of the `Vector` class.

5.4.3. The `Expression` Class

The `quickvec::Expression` class closely resembles the `elsa::Expression` class introduced in Section 4.4. Likewise, the `Expression` saves the work to be done through its template arguments `Callable` which is executed upon the `Operands`. An `Operand` can be either a scalar, a `Vector` or an `Expression`. However, there are also differences: Instead of having an `eval` function, the Quickvec `Expression` provides element-wise evaluation using the square brackets operator. Furthermore, all of the `Operands` are saved by value in the member tuple `_args`. In the elsa case, the `DataContainer` is saved by reference to prevent copying the data.

The rationale behind this is that the Quickvec `Expression` has to be copied to the GPU in order to be evaluated. If `Vector` is saved by reference, this results in memory errors when accessing the object in GPU code as it resides in main memory. Note that this is independent of the fact that the underlying data array is allocated in unified memory. Consequently, taking `Vector` by value solves the issue but requires a shallow

copy constructor to have the copies reference the same data and avoid expensive deep copy operations. The class structure is shown in UML notation in Figure 5.2.

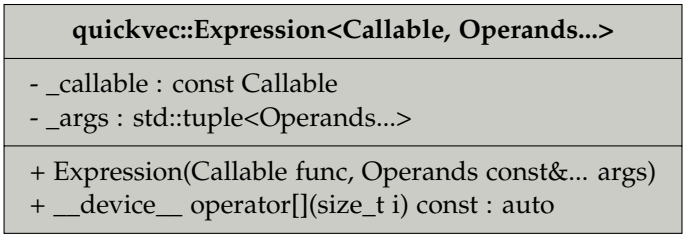


Figure 5.2.: Overview of the `quickvec::Expression` class.

`Expression` objects are created in the overloaded operators with their `Callable` argument being a generic lambda as demonstrated in Code 5.3. The lambda is a device function which performs the element-wise computation. In the elsa `Expression` case, the corresponding lambda only delegates the work to Eigen.

Code 5.3: Overloaded operator + creates an `Expression` object

```
1 auto operator+(Vector const& lhs, Vector const& rhs)
2 {
3     auto addition = [] __device__ (float l, float r) { return l + r; };
4     return Expression{addition, lhs, rhs};
5 }
```

5.4.4. Evaluating an `Expression`

Before evaluating an `Expression`, it has to be copied to the GPU. Line 10 in Code 5.4 shows this operation as part of the `Vector::eval` function which evaluates an `Expression` into the `Vector`. Copying the `Expression` is cheap as it only contains pointers to the actual data or primitive types for the scalars. Then, the `compute` kernel introduced in Code 5.2 is launched in Line 13 of Code 5.4 with a fixed number of threads per block and the number of blocks matching the problem size.

Code 5.4: Evaluating an `Expression` into a `Vector`

```
1 template <typename Expression>
2 void Vector::eval(Expresson expression)
3 {
4     unsigned int blockSize = 256;
5     auto numBlocks = static_cast<unsigned int>((_size + blockSize - 1)
6                                                / blockSize);
7
8     Expression* devExpression;
9     cudaMalloc(&devExpression, sizeof(Expression));
```

```
    cudaMemcpy(devExpression, &expression,
               sizeof(Expression), cudaMemcpyHostToDevice);

    compute<<<numBlocks, blockSize>>>(_size, devExpression, _data.get());

    cudaDeviceSynchronize();
    cudaFree(devExpression);
}
```

As a next step, the parallel processing starts with hundreds of threads running on the GPU. Each thread evaluates the `Expression` at a specific index through its square brackets operator shown in Code 5.5. Here, separate cases for unary and binary operators are created using `constexpr if`.

Code 5.5: Evaluating an `Expression` index-wise

```
__device__ float quickvec::Expression::operator[](size_t i) const
{
    if constexpr (std::tuple_size_v<decltype(_args)> == 1) {
        return _callable(evaluateOrReturn(std::get<0>(_args), i));
    } else {
        return _callable(evaluateOrReturn(std::get<0>(_args), i),
                         evaluateOrReturn(std::get<1>(_args), i));
    }
}
```

Calling `evaluateOrReturn` shown in Code 5.6 on each `Operand` at the specified index either returns the value in the case of scalars and `Vectors` or further descends into the expression tree through executing the square bracket operator of the nested `Expression` object.

Code 5.6: The `evaluateOrReturn` function

```
template <class Operand>
__device__
constexpr float evaluateOrReturn(Operand const& operand, size_t const i)
{
    if constexpr (isVectorOrExpression<Operand>) {
        return operand[i];
    } else {
        return operand;
    }
}
```

If the `Operands` have returned their values, the generic lambda stored in the `_callable` member is executed with the results from `evaluateOrReturn` as arguments. Finally, the

lambda performs the computation for the two elements.

As each index-wise computation is independent of each other, the problem can fully utilize the parallel processing power of the GPU without considering any inter-thread or inter-block synchronization. This kind of computation is categorized as an "embarrassingly" parallel problem [HS11].

5.4.5. Compiling Quickvec

Compiling Quickvec using the regular toolchain with the CUDA compiler driver `nvcc` is not possible, because `nvcc` only supports code up to C++14, but Quickvec ETs rely heavily on C++17. Instead, the compiler *Clang* is used to directly compile CUDA code. This feature is still considered experimental and therefore requires a close match between the Clang and the CUDA version [Wu+16]. Tested configurations include

- CUDA 9.2 with Clang 8 or
- CUDA 10.0 with Clang 8.

As the ETs generate custom kernels at compile-time, it is not possible to compile Quickvec once and then use it throughout the project. Instead, every target which utilizes Quickvec computations has to be compiled with the setup mentioned before.

5.5. Integrating Quickvec into elsa

The framework developed in Chapter 4 for utilizing the Eigen internal ETs within elsa can also be used for the Quickvec integration.

A new `DataHandlerGPU` is added which is used as the default handler type if the compile-time switch for Quickvec support is activated. This handler wraps around the `quickvec::Vector` class. Additionally, when constructing a `DataContainer`, the handler type can be specified using an `enum` as an additional argument.

To accommodate both Eigen and Quickvec side-by-side, the evaluation of elsa `Expressions` has to provide two different paths because the return type of all involved functions differ depending on the underlying library. Evaluating an `Expression` containing CPU `DataContainers` returns Eigen `Expression` objects. On the other hand, Quickvec `Expression` objects are returned with GPU `DataContainers`. The two paths are created at compile-time through templating all involved functions, namely `elsa::Expression::eval` and the three overloaded `elsa::evaluateOrReturn`, with a boolean GPU flag. Consequently, their definitions become

```
template <typename Operand, bool GPU>
elsa::evaluateOrReturn(Operand operand)
```

and

```
template <bool GPU>
elsa::Expression::eval();.
```

The differentiation between CPU and GPU is achieved at run-time through dynamic casts of the base class pointer `_dataHandler`. Line 6 in Code 5.7 executes the CPU path whereas Line 9 in Code 5.7 completes the GPU path when assigning an `Expression` to a `DataContainer`.

Code 5.7: Evaluating both Quickvec and Eigen depending on the handler type

```
1  template <typename Expression>
2  DataContainer<data_t>& operator=(Expression const& expression)
3  {
4      if (auto handler =
5              dynamic_cast<DataHandlerCPU<data_t>*>(_dataHandler.get())) {
6          handler->accessData() = expression.template eval<false>();
7      } else if (auto handler =
8              dynamic_cast<DataHandlerGPU<data_t>*>(_dataHandler.get())) {
9          handler->accessData().eval(expression.template eval<true>());
10     }
11     return *this;
12 }
```

In addition to using the computational power of Quickvec, the GPU projectors in elsa can be adapted to the fact that the data already resides in GPU memory with Quickvec. Expensive main to device memory transfers are avoided and replaced by GPU internal copy operations. For Joseph's method, a copy is still necessary as it uses specialized textured memory in the kernel. Using in-place manipulation with Siddon's method is possible but not explored as part of the thesis. At run-time, the `apply` and `applyAdjoint` functions of the projectors can derive the type of memory transfer from the input they receive.

5.6. Benchmarks and Results

For benchmarking, the same setup as in Section 4.5 using Catch2 with the server specified in Table 5.1 is used. Problem sizes are varied in the range of 256^3 to 1024^3 individual elements, with the upper bound set by memory limitations of available GPUs. A reconstruction volume with 1024^3 elements requires 4 GiB of memory, assuming floats with four bytes each. The corresponding full reconstruction task needs around five times as much memory due to intermediate results in the solver and projector. However, most commercially available GPUs currently provide around 10 GiB of memory which is not sufficient for tasks larger than 1024^3 elements. Each benchmark is run twenty times before computing the mean result and standard deviation.

5.6.1. Numeric Computations

The first benchmark is the saxpy computation

Specification	Version / Value
CPU	Intel Xeon E5-2687W
GPU	Nvidia RTX 2080Ti
Main memory	64 GiB
Linux Kernel	4.15.0-88-generic
Operating System	Ubuntu 18.04.4 LTS
Eigen commit	1affbe9...
Quickvec commit	a5bf6e06...
elsa commit before ETs	4497388...
elsa commit with Quickvec	2eef2f45...

Table 5.1.: Server setup for Quickvec benchmarks.

```
y = a x + y,
```

where `x` and `y` are array-like containers and `a` a floating pointer number. As a reference implementation, the Nvidia proprietary *cuBLAS* library is used which provides GPU versions of the algorithms in the BLAS library. The other scenarios are using Quickvec directly, Quickvec wrapped in elsa and Eigen directly. Figure 5.3 shows the absolute run times in comparison. The GPU-based algorithms outperform the CPU-based Eigen implementation by more than one order of magnitude.

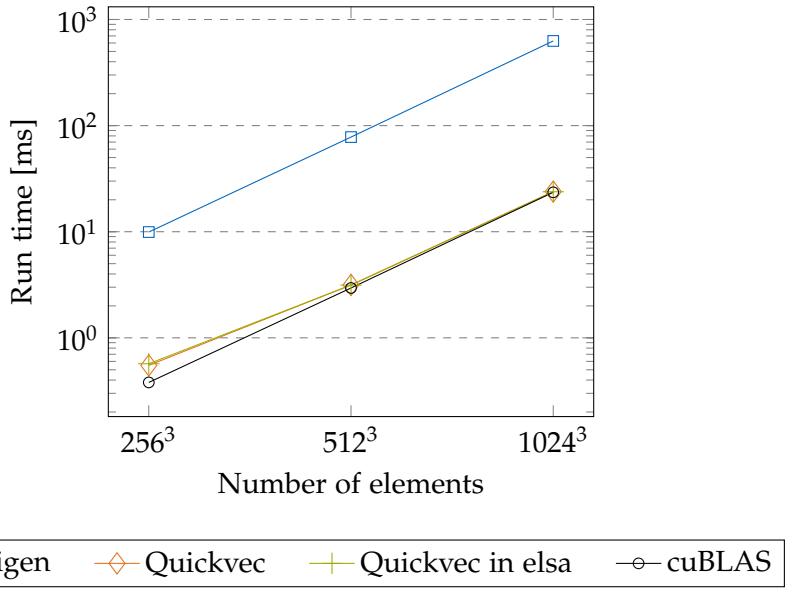


Figure 5.3.: Mean run times computing saxpy.

Relative results with cuBLAS being the baseline are presented in Figure 5.4. The plot shows that Quickvec has very minor performance deficits compared to the cuBLAS saxpy kernel for larger problem sizes. For 256^3 elements, the overhead is much bigger at

around 45 % because the necessary transfer of the `Expression` object is more relevant for the run time compared to the computation itself.

Furthermore, wrapping Quickvec into elsa adds a marginal overhead of maximum five percent. That confirms the results from Chapter 4: The elsa ETs are able to leverage internal ET implementations from both Eigen and Quickvec. All mean run times as well as their standard deviations can be found in Table B.1.

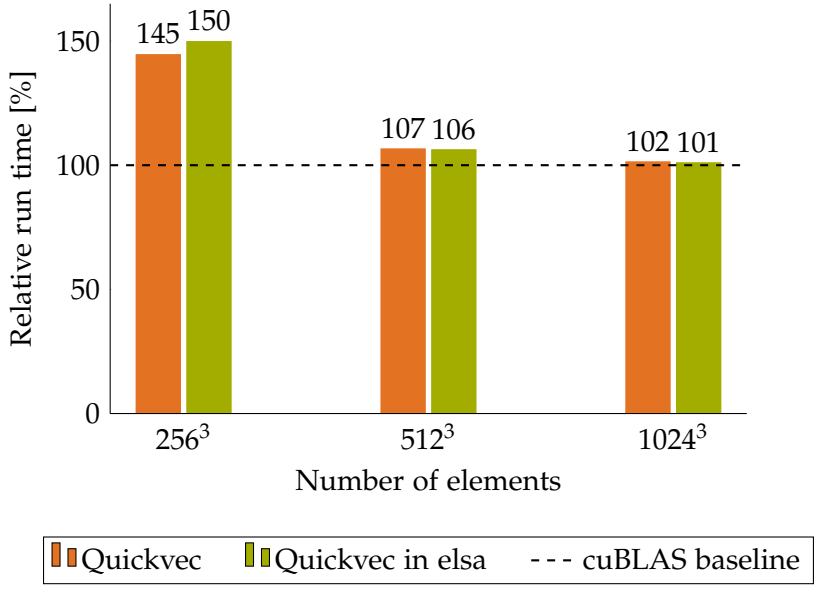


Figure 5.4.: Mean relative run times computing saxpy.

5.6.2. GPU Projectors

The second efficiency improvements through using Quickvec are reduced main memory to GPU memory transfers when using GPU projectors to compute the forward or backward projection. As previously mentioned, two implementations are available in elsa: `SiddonsMethodCUDA` and `JosephsMethodCUDA`. To measure the effect, both projectors are run using a GPU or a CPU `DataContainer` as an input. Based on the input, the algorithm creates either an internal or external copy. Figure 5.5 displays relative results with the main memory `DataContainer` as the baseline.

For smaller reconstruction volumes, there are noteworthy run time reductions in the region of 10 % to 40 %. However, for size 1024^3, those relative speed gains are diminished to under 3 %. This can be explained by the fact that the run time of the projection algorithm itself grows faster than the linearly increasing time demand for memory transfers.

As Joseph's method has shorter absolute run times than Siddon's method, the fraction of time spent with memory transfers is larger. Hence, the efficiency gains are also bigger. All the absolute measurements can be found in Appendix B in Table B.2.

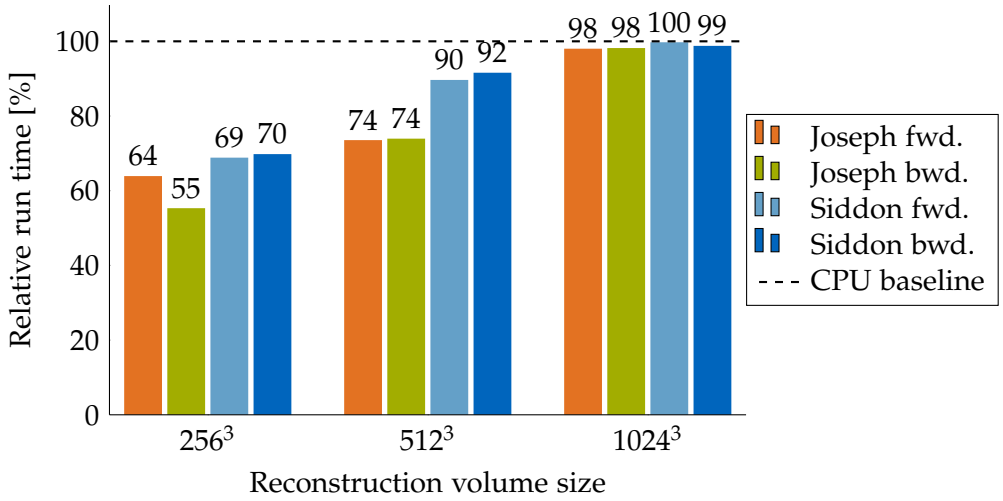


Figure 5.5.: Relative run time of projectors using a GPU `DataContainer` as input. The baseline scenario consists of a CPU `Datacontainer` as input. Fwd. abbreviates the forward projection, bwd. the backward projection.

5.6.3. Full Reconstruction

In a full reconstruction task, both the reduced memory transfers and the faster GPU computations lead to efficiency improvements. As in the results of Chapter 4, the full reconstruction task consists of a 3D Shepp-Logan phantom as a test volume, the projector `JosephsMethodCUDA` and the method of conjugated gradients as a solver. Again, different sizes from 256^3 to 768^3 elements are tested with the upper limit determined by the memory limitations of the available GPU. The number of projections matches the dimension of the volume. Two scenarios are benchmarked:

(a) elsa with a CPU `DataContainer` using Eigen. This corresponds to the state of elsa before the changes discussed in Chapter 5 are introduced. In legend entries, the scenario is abbreviated to CPU Eigen.

(b) elsa with a GPU `DataContainer` using Quickvec. This is the final state of elsa considered in this thesis. It is referred to as GPU Quickvec in legend entries.

The absolute mean run times for one solver iteration are presented in Figure 5.6. It shows that there is a considerable speedup from using Quickvec which tends to get smaller for larger problem sizes. That is again caused by the fact that the time spent within the projector's `apply` and `applyAdjoint` functions grows larger compared to the time in the remaining solver loop.

Nevertheless, for the size of 768^3 voxels, the run time is reduced by around 20 %, compared to 38 % and 51 % for sizes of 512^3 and 256^3 respectively. The relative results for all sizes are shown in Figure 5.7 and the detailed measurements are added in Table B.3.

Additionally, the full reconstruction task is run using elsa in a state before the introduction of the ETs added in Chapter 4. Using this scenario as a baseline, the relative

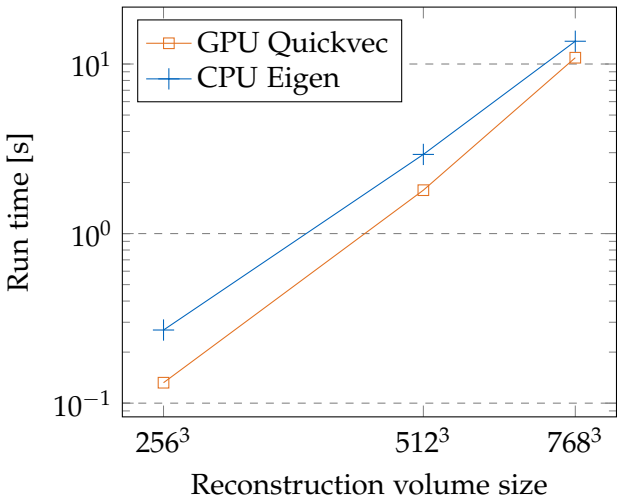


Figure 5.6.: Mean run times for one solver iteration in a full reconstruction task.

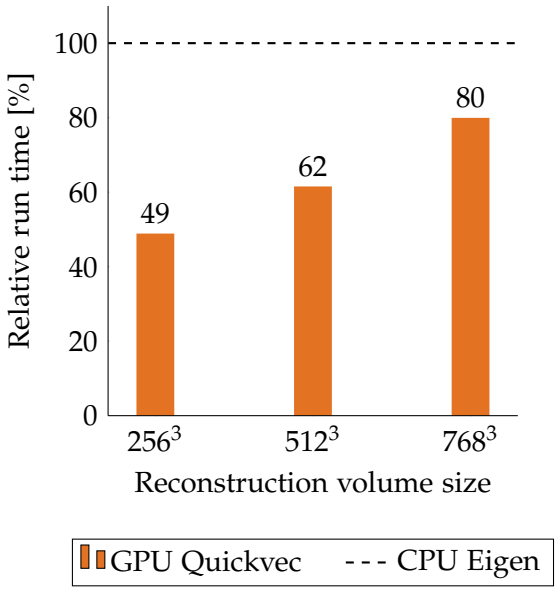


Figure 5.7.: Relative mean run times for one solver iteration in a full reconstruction task. The baseline scenario is running the example with a CPU `DataContainer`.

results are presented in Figure 5.8. The combined total run time reductions range from 66 % for size 256^3, to 54 % for size 512^3 and 42 % for the largest problem size of 768^3. Other commits that have been added to elsa between the introduction of the ETs and the GPU-based `DataContainer` also affect the measurements as indicated by the difference between the state after ETs have been added and the CPU Eigen scenario.

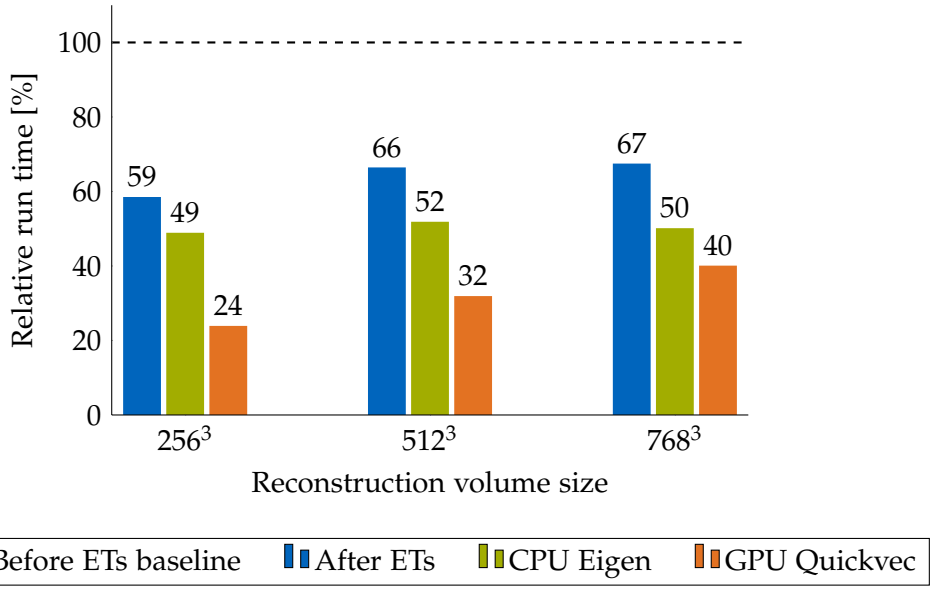


Figure 5.8.: Relative mean run times for one solver iteration in a full reconstruction task. The baseline scenario is elsa before the ETs are added.

6. Discussion

6.1. Summary

In the present thesis, efficiency improvements for the tomographic reconstruction framework elsa are presented. Those improvements are grounded in optimizing basic numeric operations while still maintaining an intuitive math-like syntax within elsa. First, the technique of ETs is integrated which allows leveraging the existing highly optimized Eigen implementations. Then, a GPU-based vector arithmetic library called Quickvec is created, harnessing the parallel processing power of GPUs. It is integrated in the same way as Eigen, therefore leveraging the internal ETs.

The results are a reduction in total reconstruction time of up to 66 % for real sized problems in CT. Additionally, larger reconstruction sizes become feasible because the peak memory consumption is reduced.

The main contribution of the thesis is the development and integration of the Quickvec library which provides general element-wise vector arithmetic using the GPU. This functionality is not limited to the scope of CT or elsa but can be used in other contexts such as tabular data processing. Consequently, Quickvec has been implemented and open-sourced in a separate repository.

6.2. Limitations and Future Directions

One limitation of the Quickvec library is that it does not support reduction operations on `quickvec::Expression` types. If such an operation is necessary, an intermediate `quickvec::Vector` which evaluates the expression has to be constructed first. Then, the reduction operation can be applied to the temporary. That requires the syntax

```
Vector temporary = 1.2 * x + y;
float result = temporary.l2norm();.
```

The consequence is a double traversal of all elements as well as a temporary allocation which both negatively affect the performance. It can be avoided through combining expression evaluation and reduction into a single kernel. Specifically that means adding a reduction step after the evaluation to the kernel given in Code 5.2, enabling concise syntax as

```
float result = (1.2 * x + y).l2norm();
```

and additionally improving efficiency.

Another limitation is that the current codebase of elsa is not fully optimized yet for the newly available ETs presented in this thesis. Many `DataContainer` operations were written without the assumption of lazy evaluation. That leads to more verbose and less intuitive code. Consequently, further work is necessary to leverage the power of lazy evaluation with the new Quickvec as well as the existing Eigen library. Furthermore, doing in-place manipulation instead of using an internal copy of the data in the GPU projectors can be explored.

Lastly, due to size limitations of GPU memory, running full reconstruction examples with volumes larger than 1024^3 voxels is currently not possible. This size already requires memory of roughly 30 GiB which is close to the maximum available memory in commercial graphics cards. Running larger tasks, for example with 2048^3 elements, is and will be above the limits of GPU memory in the foreseeable future – even until the end of the decade. The solution to this problem is splitting up the reconstruction volume into subvolumes and running reconstruction tasks either sequentially on a single GPU or in parallel on multiple GPUs. However, this requires intelligent splitting algorithms as well as extensive rework of elsa. Adding this feature can be a topic for future work.

A. Detailed Benchmarking Results for Expression Templates

Scenario \ Size	256^3		512^3		1024^3	
	Mean [s]	σ [s]	Mean [s]	σ [s]	Mean [s]	σ [s]
Before ETs solver	0.303	0.009	2.306	0.007	32.963	0.454
Before ETs projector	0.263	0.004	3.313	0.018	52.354	0.370
After ETs solver	0.062	0.007	0.436	0.012	6.951	0.120
After ETs projector	0.261	0.003	3.329	0.27	54.115	1.154

Table A.1.: Run times for one solver iteration in a full reconstruction task. Joseph's method is used as a projector and the method of conjugated gradients as a solver. The number of projections matches the size of the reconstruction volume. σ indicates the standard deviation of twenty samples.

Expression \ Size		256^3		512^3		1024^3		2048^3	
		Mean [ms]	σ [ms]	Mean [ms]	σ [ms]	Mean [ms]	σ [ms]	Mean [ms]	σ [ms]
Eigen directly	`x = x * y`	7.52	0.022	68.01	0.31	536.50	0.6	4266.01	4.8
Before ETs		29.16	0.6	227.35	3.8	1896.99	21.8	14361.81	121.25
After ETs		7.76	0.7	76.72	0.49	606.16	1.21	4821.20	11.27
Eigen directly	`x = x * y + y`	7.39	0.022	72.08	0.17	572.09	14.6	4559.64	36.9
Before ETs		58.47	0.3	456.46	1.7	3752.65	105.4	N/A	N/A
After ETs		7.91	0.47	79.35	0.25	627.06	8.92	4989.84	32.8
Eigen directly	`x = x * y + y / z`	9.93	0.015	84.90	0.24	673.37	1.1	5376.39	11.2
Before ETs		87.31	0.1	681.45	1.7	5478.87	105.1	N/A	N/A
After ETs		10.67	0.31	93.16	1.02	744.20	3.2	5903.69	42.2
Eigen directly	`x = x * y + y / z + x`	10.32	0.027	92.50	0.30	735.58	1.2	5275.39	188.8
Before ETs		155.92	0.8	907.62	4.7	8961.69	40.0	N/A	N/A
After ETs		11.00	0.02	91.79	2.1	730.92	4.21	5822.17	7.6
Eigen directly	`x = x * y + y / z + x * z`	10.50	0.032	95.90	0.74	762.46	0.6	6096.52	10.5
Before ETs		144.03	0.2	1170.94	4.0	9169.98	43.6	N/A	N/A
After ETs		11.35	0.38	97.65	0.98	777.02	5.21	6169.76	12.1

Table A.2.: Detailed benchmarking results for numeric expression from Section 4.5. `x`, `y`, `z` represent `Eigen::Matrix` ojects or `DataContainers`. The operators `+`, `-`, `*` and `/` perform element-wise computations. N/A values indicate that out of memory errors occured. For each measurement twenty samples are recorded and the mean run time as well as standard deviation σ computed.

B. Detailed Benchmarking Results for Quickvec

Scenario \ Size	256^3		512^3		1024^3	
	Mean [ms]	σ [ms]	Mean [ms]	σ [ms]	Mean [ms]	σ [ms]
Eigen	9.93	0.03	77.98	0.08	627.64	5.49
cuBLAS	0.38	0.01	2.95	0.01	23.52	0.02
Quickvec	0.55	0.01	3.15	0.02	23.89	0.37
elsa using Quickvec	0.57	0.02	3.14	0.03	23.80	0.07

Table B.1.: Run times for saxpy $y = ax + y$ computation where x and y are vectors and a is a scalar. Size refers to the number of elements in the vectors x and y. σ denotes the standard deviation of twenty sample runs.

Scenario \ Size	256^3		512^3		1024^3	
	Mean [s]	σ [s]	Mean [s]	σ [s]	Mean [s]	σ [s]
Siddon's Forward CPU	0.122	0.002	2.708	0.091	48.673	0.273
Siddon's Forward GPU	0.084	0.001	2.428	0.122	48.548	0.192
Siddon's Backward CPU	0.149	0.003	3.758	0.032	66.178	0.389
Siddon's Backward GPU	0.104	0.002	3.442	0.267	65.371	0.580
Joseph's Forward CPU	0.097	0.002	1.066	0.024	14.925	0.157
Joseph's Forward GPU	0.062	0.001	0.784	0.011	14.630	0.098
Joseph's Backward CPU	0.094	0.003	1.132	0.24	17.867	0.261
Joseph's Backward GPU	0.052	0.001	0.837	0.032	17.545	0.138

Table B.2.: Run times for projectors using a CPU or a GPU `DataContainer` as an input. σ denotes the standard deviation for twenty sample runs.

Scenario \ Size	256^3		512^3		768^3	
	Mean [s]	σ [s]	Mean [s]	σ [s]	Mean [s]	σ [s]
`DataContainer` GPU	0.132	0.002	1.803	0.017	10.892	0.083
`DataContainer` CPU	0.270	0.012	2.930	0.025	13.632	0.220
Before ETs	0.552	0.016	5.649	0.025	27.168	1.580

Table B.3.: Run times for a full reconstruction task. Joseph's method is used as a projector and the method of conjugated gradients as a solver. Twenty iterations per scenario are run. σ denotes the standard deviation of the samples.

List of Figures

List of Tables

List of Code

Bibliography

[AM11] J. Als-Nielsen and D. McMorrow. *Elements of Modern X-ray Physics*. John Wiley & Sons, Inc., Mar. 2011. DOI: 10.1002/9781119998365.

[BH12] N. Bell and J. Hoberock. "Thrust: Productivity-Oriented Library for CUDA." In: *Astrophysics Source Code Library* 7 (Dec. 2012), pp. 12014–.

[Bla+02] L. S. Blackford, A. Petitet, R. Pozo, K. Remington, R. C. Whaley, J. Demmel, J. Dongarra, I. Duff, S. Hammarling, G. Henry, et al. "An updated set of basic linear algebra subprograms (BLAS)." In: *ACM Transactions on Mathematical Software* 28.2 (2002), pp. 135–151.

[Bre+13] A. Breglia, A. Capozzoli, C. Curcio, and A. Liseno. "Achieving Natural Mathematical Expression Programming on GPUs via Expression Templates." In: *2013 European Modelling Symposium*. IEEE, Nov. 2013. DOI: 10.1109/ems.2013.84.

[Buz11] T. M. Buzug. "Computed tomography." In: *Springer Handbook of Medical Technology*. Springer, 2011, pp. 311–342.

[Coo12] S. Cook. *CUDA Programming: A Developer's Guide to Parallel Computing with GPUs*. 1st. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2012. ISBN: 9780124159334.

[Cor63] A. M. Cormack. "Representation of a Function by Its Line Integrals, with Some Radiological Applications." In: *Journal of Applied Physics* 34.9 (Sept. 1963), pp. 2722–2727. DOI: 10.1063/1.1729798.

[Dem+13] D. Demidov, K. Ahnert, K. Rupp, and P. Gottschling. "Programming CUDA and OpenCL: A case study using modern C++ libraries." In: *SIAM Journal on Scientific Computing* 35.5 (2013), pp. C453–C472.

[EHN96] H. W. Engl, M. Hanke, and A. Neubauer. *Regularization of inverse problems*. Vol. 375. Springer Science & Business Media, 1996.

[GJ+] G. Guennebaud, B. Jacob, et al. *Lazy Evaluation and Aliasing*. URL: https://eigen.tuxfamily.org/dox/TopicLazyEvaluation.html (visited on 03/25/2020).

[GJ+10] G. Guennebaud, B. Jacob, et al. *Eigen v3*. http://eigen.tuxfamily.org. 2010.

[Had02] J. Hadamard. "Sur les problèmes aux dérivées partielles et leur signification physique." In: *Princeton university bulletin* (1902), pp. 49–52.

[Her09] G. T. Herman. *Fundamentals of Computerized Tomography*. Springer London, 2009. DOI: 10.1007/978-1-84628-723-7.

[Hou73] G. N. Hounsfield. "Computerized transverse axial scanning (tomography): Part 1. Description of system." In: *The British Journal of Radiology* 46.552 (Dec. 1973), pp. 1016–1022. DOI: 10.1259/0007-1285-46-552-1016.

[HS11] M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann, 2011.

[Jos82] P. M. Joseph. "An Improved Algorithm for Reprojecting Rays through Pixel Images." In: *IEEE Transactions on Medical Imaging* 1.3 (Nov. 1982), pp. 192–196. DOI: 10.1109/tmi.1982.4307572.

[LHF19] T. Lasser, M. Hornung, and D. Frank. "elsa - an elegant framework for tomographic reconstruction." In: *Fully Three-Dimensional Image Reconstruction in Radiology and Nuclear Medicine (Fully3D)*. June 2019.

[Owe19] B. Owens. *Expression Templates for Efficient, Generic Finance Coda*. 2019. URL: https://github.com/CppCon/CppCon2019/blob/master/Presentations/expression_templatesfor_efficient_generic_finance_code/expression_templatesfor_efficient_generic_finance_code__bowie_owens__cppcon_2019.pdf (visited on 03/16/2020).

[Rön96] W. Röntgen. "On a New Kind of Rays." In: *Nature* 53.1369 (Jan. 1896), pp. 274–276. DOI: 10.1038/053274b0.

[Rup+16] K. Rupp, P. Tillet, F. Rudolf, J. Weinbub, A. Morhammer, T. Grasser, A. Jüngel, and S. Selberherr. "ViennaCL—linear algebra library for multi- and many-core architectures." In: *SIAM Journal on Scientific Computing* 38.5 (2016), S412–S439.

[Sid85] R. L. Siddon. "Fast calculation of the exact radiological path for a three-dimensional CT array." In: *Medical physics* 12.2 (1985), pp. 252–255.

[SL74] L. A. Shepp and B. F. Logan. "The Fourier reconstruction of a head section." In: *IEEE Transactions on Nuclear Science* 21.3 (June 1974), pp. 21–43. DOI: 10.1109/tns.1974.6499235.

[Van03] D. Vandevoorde. *C++ templates: The Complete Guide*. Boston: Addison-Wesley, 2003. ISBN: 0-201-73484-2.

[Van17] D. Vandevoorde. *C++ Templates: The Complete Guide (2nd Edition)*. Addison-Wesley Professional, Sept. 2017. ISBN: 0321714121.

[Vel95] T. Veldhuizen. "Expression templates." In: *C++ Report* 7.5 (1995), pp. 26–31.

[WFH18] B. Wicht, A. Fischer, and J. Hennebert. "Seamless GPU Evaluation of Smart Expression Templates." In: *2018 International Conference on High Performance Computing & Simulation (HPCS)*. IEEE, July 2018. DOI: 10.1109/hpcs.2018.00045.

[Wu+16] J. Wu, A. Belevich, E. Bendersky, M. Heffernan, C. Leary, J. Pienaar, B. Roune, R. Springer, X. Weng, and R. Hundt. "Gpucc: An Open-Source GPGPU Compiler." In: *Proceedings of the 2016 International Symposium on Code Generation and Optimization*. CGO '16. Barcelona, Spain: Association for Computing Machinery, 2016, pp. 105–116. ISBN: 9781450337786. DOI: 10.1145/2854038.2854041.

[WVL14] M. Wieczorek, J. Vogel, and T. Lasser. "CampRecon — a software framework for linear inverse problems." In: *TUM Technical Report* (2014).

[WWM11] P. Wiemann, S. Wenger, and M. Magnor. "CUDA Expression Templates." In: *WSCG Communication Papers Proceedings*. ISBN 978-80-86943-82-4. Jan. 2011, pp. 185–192.