Diese Arbeit wurde vorgelegt am
Lehrstuhl für Hochleistungsrechnen (Informatik 12), IT Center.

HPC Code Generation for Parallel Pattern Based Algorithms on Heterogeneous Architectures

HPC Code Generierung für Parallele Musterbasierte Algorithmen auf Heterogenen Architekturen

Masterarbeit

Adrian Schmitz
Matrikelnummer: 357201

Aachen, den 01.03.2021
Communicated by Prof. Matthias S. Müller

Erstgutachter:	Prof. Dr. rer. nat. Matthias S. Müller (')
Zweitgutachter:	Prof. Dr. Bernhard Rumpe (*)
Betreuer:	Julian Miller, M.Sc. (')

(') Lehrstuhl für Hochleistungsrechnen, RWTH Aachen University
IT Center, RWTH Aachen University
(*) Lehrstuhl für Software Engineering, RWTH Aachen University

Ich versichere hiermit, dass ich die vorliegende Arbeit selbständig und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe. Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten und nicht veröffentlichten Schriften entnommen sind, sind als solche kenntlich gemacht. Die Arbeit ist in gleicher oder ähnlicher Form noch nicht als Prüfungsarbeit eingereicht worden.

Aachen, den 01.03.2021

Kurzfassung

Parallele Programmierung erfreut sich einer immer grösseren Vielfalt in verschiedenen Bereichen von Wissenschaft und Wirtschaft. Insbesondere in Bezug auf Simulationen und maschinellem Lernen wird immer mehr Rechenleistung benötigt. Zudem steigen Spezialisierung und Heterogenität im Aufbau der Systeme.
Zur besseren Ausnutzung der Resourcen werden von der Hardware abhängige Optimierungen benötigt. Viele der Anwendungen für solche Systeme werden nicht von High Performance Computing (HPC)-Experten entwickelt, sondern von Experten in ihren jeweiligen Fachbereichen.
Die Anwendung Hardware-spezifischer Optimierungen erfordert ein hohes Verständnis der Systemarchitekturen und nimmt daher viel Zeit für die Entwicklung in Anspruch. Darüber hinaus, schränken Hardware-spezifische Optimierungen die Portabilität des Codes ein, da diese spezifischen Optimierungen nicht zwingend performant auf anderen Architekturen sind.
Um die Nutzer komplexer Systeme bei der Anwendungsentwicklung zu unterstützen, ist die Verwendung automatischer Optimierungen mit Hilfe eines Compilers sinnvoll. Diese Optimierungen sind für gewöhnlich sehr abstrakt und benötigen eine globale Sicht auf das zugrunde liegende Modell. Die theoretischen Ansätze für diese globalen Optimierungen wurden bereits von Miller et al. [**?**] eingeführt und von Trümper [**?**] implementiert. Die derzeitige Implementierung des zugrundeliegenden Systems kann die Optimierungen derzeit allerdings noch nicht in ausführbaren Code überführen. Um die Optimierungen effizient nutzen zu können, sollen diese automatisch in Binärdateien transformiert werden können.
Die vorliegende Arbeit befasst sich mit der Generierung von optimiertem und ausführbarem Code für heterogene Systeme. Zur Verknüpfung des Codegenerators mit der Optimierung, wird eine zusätzliche Abstraktionsebene eingeführt. Diese erlaubt es, die globalen Optimierungen im Kontext der Anwendung zu spezifizieren. So kann der Codegenerator die abstrakte Darstellung verwenden, um parallelen Code zu generieren, ohne von der allgemeinen Implementierung der Optimierung abhängig zu sein.
Der Abstract Mapping Tree (AMT), welcher in dieser Arbeit eingefürt wird, erweitert die bereits existierende Intermediate Representation (IR) des übergreifenden Projektes. Der implementierte Codegenerator transformiert den entsprechenden AMT in optimierten Quelltext.
Zur Evaluaierung der Qualität wird der generierte Code mit handgeschriebenen Optimierungen verglichen. Die Kriterien der Evaluierung umfassen die Korrektheit des Codes, den erzeugten Laufzeitoverhead und die Laufzeit des generierten Codes im Vergleich mit äquivalent optimiertem, handgeschriebenem Code.

Zur Sicherstellung der Korrektheit des generierten Codes, wird die Fehlerklassifikation von Schmitz et al. [**?**] erweitert. Diese Erweiterungen umfassen Fehler auf verteilten Systemen und Fehler im Bezug auf die Codegenerierung. Basierend auf dieser erweiterten Klassifikation wird eine Testumgebung eingeführt, welche als Teil dieser Arbeit entwickelt wurde. Die Testumgebung stellt sicher, dass grundlegende Konstrukte im generierten Code korrekt umgesetzt werden.
Der generierte und optimierte Code wird auf Basis eines Vergleichs mit handgeschriebenem Code evaluiert. Diese Evaluierung umfasst typische parallele Algorithmen, wie ein forward pass in einem Neuronalen Netz oder Bildverarbeitung. Als Grundlage für den Vergleich mit handgeschriebenem Code werden die Korrektheit, sowie der Laufzeitoverhead und die Laufzeit des Algorithmus herangezogen. Auf Basis dieser Evaluierung werden Verbesserungvorschläge für den Generator vorgeschlagen und diskutiert.

Stichwörter: HPC, Code Generation, MPI, CUDA, PThreads, Cluster

Abstract

Parallel programming is widely used in many different domains in science and engineering. Especially, simulations and machine learning applications require large amounts of computing power. To solve this demand, large-scale systems use spezialized and heterogeneous hardware architectures. To properly utilize the capabilities of these specialized systems, hardware-dependant optimizations are necessary. Utilizing these optimizations by hand is a time consuming task which requires deep understanding of the target system. Many applications are developed by domain experts and not performance engineers and, thus, may not be able to fully utilize these systems. Furthermore, applications optimized for one system may not perform well on a different system restricting the portability of the code.
Automatic optimizations based on an unified and high-level base code could support the users of such systems. These optimizations are rather abstract and require a global model of the hardware. Recently, a theoretical approach for such global optimizations was introduced by Miller et al. [**?**] and implemented by Trümper [**?**]. However, they cannot be used to automatically generate optimized executable code yet. To utilize the generated optimizations, they need to be transformed into an executable.
In this thesis, optimizations for heterogeneous systems are translated into executable code. An additional layer of abstraction is introduced to define an interface between the optimization and the code generator. This layer of abstraction is used to specify the optimizations in the context of the application. The abstract mapping tree (AMT), introduced in this thesis, extends the intermediate representation (IR) of the existing implementation of the overarching project. As a result, the implemented code generator can produce optimized code based on the abstraction without directly depending on the implementation of the optimizer. The implemented code generator transforms the AMT into optimized source code. Therefore, the implementation follows a source-to-source compilation approach.
To ensure the correctness of the generated code, the error classification introduced by Schmitz et al. [**?**] is expanded to cover defects on distributed-memory systems and generated code. Based on the extended error classification a test suite is introduced in this thesis. The test suite covers the correct generation of constructs utilized in the generated code.
The generated optimized code is evaluated in comparison to handwritten code. The evaluation covers optimizations on typical parallel algorithms like a neural network forward pass or image processing. As key indicators for the evaluation the correctness, runtime overhead and runtime in comparison to handwritten code are used, e.g., for the Monte Carlo estimation of π the generated code takes *16.5826* sec-

onds, while the handwritten optimized code takes *21.9670* seconds. Based on the evaluation, improvements for the implementation of the generator are proposed and discussed.

Keywords: HPC, Code Generation, MPI, CUDA, PThreads, Cluster

Contents

List of Figures

List of Listings

List of Tables

1 Introduction

The importance of large computer clusters is increasing for scientific and engineering purposes. To improve the performance, these systems utilize more dedicated hardware [**?**]. The increasing complexity of HPC-systems is mainly caused by the limitation in the energy consumption [**?**] and the memory wall [**?**]. The current Top500 list [**?**] is composed of different cluster architectures which contain GPUs from different vendors, co-processors like the Intel Xeon Phi, specialized accelerators like the Matrix-2000 or ARM processors like the current top system of the Top500 list. In total ten different accelerator architectures and 24 different CPU architectures are used within the Top500.
The optimization of computer clusters and corresponding applications requires a deep understanding of the used hardware architecture. To handle the increasing complexity of HPC-systems, especially regarding exa-scale computing (10^{18} floating point operations per second), new programming paradigms are necessary [**?**].
The developers of applications are experts in their domain, but rarely in HPC, which may result in code which is not fully optimized. Although, even for HPC experts, writing optimizations for a specific system is a time consuming and complex task. On the other hand, utilizing architecture based optimizations reduces the portability of parallel applications. One approach to handle these portability issues is the automatic generation of optimized code.
The automatic generation of global optimizations targeting arbitrary systems was introduced by Miller et al. [**?**]. Their work is based on the data-centric optimization of parallel patterns for a given architecture. The current implementation of Millers work by Trümper [**?**] provides the generation of automatic optimization strategies. These strategies are yet to be transformed into executable binaries.
The generation of optimized code is often focussed on specific algorithms or hardware architectures [**?**] [**?**]. Although, there are already optimization and generation approaches present, the implementation of global optimizations by Trümper [**?**] were shown to provide suitable automatic optimizations for many algorithms on different architectures.
In this thesis, a generator and an interface for automatically transforming the optimization introduced by Miller et al. [**?**] and implemented by Trümper [**?**] into binaries are developed and implemented. The user written code is transformed into an abstract tree based data structure. The abstract pattern tree (APT) is used as the intermediate representation (IR) of the user written code. The APT and the details of the optimization are transformed into an abstract mapping tree (AMT), as part of this thesis. The AMT is a tree based data structure providing an abstract specification of an optimized algorithm. Based on the definition of the AMT, syn-

chronization points and data transfers are generated.
In the next steps, the implemented generator transforms the AMT into optimized source code. The generated code supports shared-memory and distributed-memory parallelism as well as GPU offloading.
The optimized source code is interpreted and compiled by already existing source compilers.
To ensure the correctness of the generated code, a test suite was developed which covers a wide range of optimizations representing key features of the generated code.
In order to evaluate the generation of specific optimizations, a debug mode was designed and implemented extending the AMT.
Further, the performance of the generated code is evaluated in comparison to handwritten optimizations implementing the same strategies defined by the optimization packages.
For the performance evaluation realistic algorithms are used. These algorithms include image processing, a forward pass in a neural network, an ensemble based classification, solving a linear equation system and the estimation of pi.
All evaluations were performed on two nodes of the RWTH Aachen University CLAIX 2018 cluster [**?**]. As part of the current Top500 [**?**] this system represents a modern architecture. The nodes consist of two Intel Xeon Platinum 8160 and two NVIDIA Volta 100 GPUs and are connected via Intel Omnipath.
The main contributions of this thesis are the following:

- Definition and implementation of the abstract mapping tree (AMT) as an abstract representation of optimized parallel pattern based applications. The AMT extends the existing IR and provides an interface for optimization and code generation packages.

- The derivation of an automatic process of synchronization and data transfers based on the AMT.

- The specification and implementation of a code generator transforming the AMT into correct and compilable source code. The generation of a Makefile and libraries ensures the correct transformation of the source code into binaries.

- The categorization of defects for generated code targeting shared-memory, distributed memory and GPU architectures.

- The definition of a test suite based on the defect classification. The test suite ensures the correctness of the generated source code.

- The evaluation of the performance of the generated code in comparison to handwritten code implementing the same optimization strategies on typical parallel algorithms.

The structure of this thesis is the following:
Chapter 2 provides an overview of related work. Chapter 3 introduces the overarching project this thesis contributes to. Furthermore, it gives an overview of the overarching project to discuss the foundations and the relation of this thesis to the project. Chapter 4 introduces the AMT and explains how it can be generated. Additionally, the synchronization and data transfer generation are discussed. In Chapter 5, the design choices and implementation details for the source code generator are presented. The error classification with the test suite and a performance evaluation of the generated code are explained in Chapter 6. The results of this thesis are discussed in Chapter 7. In Chapter 8, conclusions of this thesis are drawn and the results of this thesis are summarized and an overview on future steps is given.

2 Related Work

The generation and intermediate representation of optimized parallel code is a complex task. Approaches like autotuners [?, ?] and the parallel code generation for specific architectures and algorithms provide extensive literature. In this chapter, the related literature is discussed and the significant differences to this work are highlighted.

Parallel Intermediate Representation

The intermediate representation is used in compilers to represent the application and to perform optimizations. The definition of a parallel IR is often strongly connected to specific programming models [?, ?, ?] or to specific algorithms [?, ?, ?]. The abstract mapping tree representation introduced in this thesis follows a parallel pattern based programming model and extends the IR of the overarching project. This allows the representation of different typical algorithms as shown in Chapter 6. The use of a parallel pattern based representation allows for the utilization of different existing programming models when generating code. IR implementations targeting specific system architectures [?, ?] are developed as well, but much like TensorFlow [?], LIFT [?] or KORALI [?] the AMT can represent parallel code for modern heterogeneous cluster systems.
TensorFlow [?] supports a similar graph based IR approach as the AMT defined in this work. In contrast to this work, TensorFlow is targeted at the optimization of machine learning systems and especially the training of these systems. In contrast to the APT (Abstract Pattern Tree), the definition of the LIFT IR is based on OpenCL constructs and focuses on the generation of GPU code. The KORALI frame work targets stochastic methods on heterogeneous cluster systems, while the approach in this thesis covers parallel patterns in order to target a wider range of applications.
There are already IR based approaches targeting the optimization of parallel patterns like LIFT. LIFT is a high-level parallel IR developed by Steuwer et al. [?]. The IR was designed focusing on parallel patterns, similar to APT utilized in this work. The results of LIFT show a performance on par with manually optimized code. Thus, revealing that the task of code generation is applicable for data driven HPC applications.
Low level IR extensions like Tapir [?] are unable by design to properly represent distributed-memory parallelism. The AMT is designed to represent shared-memory parallelism, distributed memory parallelism and GPU offloading. While the generation of FPGA code is currently not supported, an appropriate generator could generate FPGA, GPU or shared-memory code depending on the optimization.

An FPGA generator can be implemented by extending the generator and AMT proposed in this work.

Parallel Code Generation

The generation of source code is a topic which is already used in many fields by introducing DSLs (Domain Specific Languages) to improve portability and readability for code produced in specific domains. The generation of efficient source code to divide the domain knowledge from the optimizations source-to-source compilation is beneficial. The generation source code for specific target architectures [**?**, **?**, **?**, **?**] is already well developed, similarly the code generation targeting specific algorithms [**?**, **?**, **?**, **?**] is also already well understood. The code generator proposed in this thesis can generate serial, shared-memory parallel, distributed-memory parallel und GPU code.
The source-to-source transformer by Verdoolaege et al. [**?**] generates parallel CUDA code based on polyhedral loops in C code. In contrast, this work generates code based on parallel patterns supporting, shared-memory parallelism, distributed-memory parallelism and GPU offloading.
The work by Yount et al. [**?**] generates and tunes stencil patterns for shared-memory parallelism with a focus on Intel Xeon architectures like the Intel Xeon Phi. The generated code is written in C++ like in this thesis. Further, the focus on the stencil pattern which is also supported by this work and the on focus specific architecture can improve the quality of the generated code. The goal of this thesis is to support a wide range of parallel patterns including the stencil and the reduction pattern on different architectures like GPUs, distributed-memory and shared-memory architectures.

Compiler Approaches

There already exists excessive literature on compiler optimizations [**?**, **?**, **?**, **?**, **?**, **?**] For example, the Intel Array Building Blocks [**?**] focus on the dynamic mapping of parallel code on different multi-core and heterogeneous many-core architecture. The dynamic approach interoperates with different programming models to improve the performance. The utilization of dynamic approaches introduces additional overhead during runtime, but generally contains more information on the program and is able to achieve potentially better optimizations. The approach of the code generator in this thesis is the static generation of optimized source code.
The compiler based auto-parallelization like in the work of Tournavitis et al. [**?**] or Kombi et al. [**?**] define compiler based optimizations. These optimizations are based on the detection of potentially parallel operations and replacing them with parallel operations. These compiler-based approaches directly generate binary code, which is limited to the optimizations performed by their compiler. The approach in this thesis is the implementation of a source-to-source compiler.

3 Background

This master thesis is part of a larger project. In order to define the contribution of this thesis, it is necessary to understand the overarching project. This chapter presents an overview on the project and the most relevant information provided by other packages within the overall project in relation to this master thesis.

3.1 Overview

In the overarching project, a modular compiler targeting automatic global optimizations on algorithms defined via parallel patterns is developed.

- **Parallel pattern:** A parallel pattern is a set of similar algorithmic building blocks, originally introduced by Mattson et al. [**?**] and McCool et al. [**?**]. In the past, approaches like MapReduce [**?**] are successfully utilized, e.g., a map and a reduction for parallel programming models. These patterns have concurrent potential, e.g., a map pattern processes each element without any dependencies on other elements of a given input array. Typically, all elements can be processed in parallel.

- **Global optimization:** A global optimization describes changes within the structure of the algorithm. Global optimizations include reordering of statements/expression, splitting/combining parallel patterns etc. Thus, creating a more favourable algorithmic structure for parallel programming on modern architectures. Contrarily, local optimizations only affect single statements or expressions , e.g., loop optimization.

Figure 3.1 provides an overview on how the compiler is structured. It demonstrates the differentiation into core and exchangeable packages and the classification into front-end, middle-ware and back-end.
The core packages, depicted in red, build the foundation of the compiler. They define the interfaces and the implementation of abstraction levels. The exchangeable packages, depicted in green, are necessary as well for the compiler to work. But different implementations can be mixed and matched to allow for a wider choice concerning input and output languages. Thus, it is especially useful for the parallel pattern language (PPL) and the code generation. Currently, only the prototype PPL for testing purposes is supported. For future versions of the compiler, it is intended to add support for other languages, e.g., with annotations like OpenMP [**?**]. This would allow existing codes to be reused.

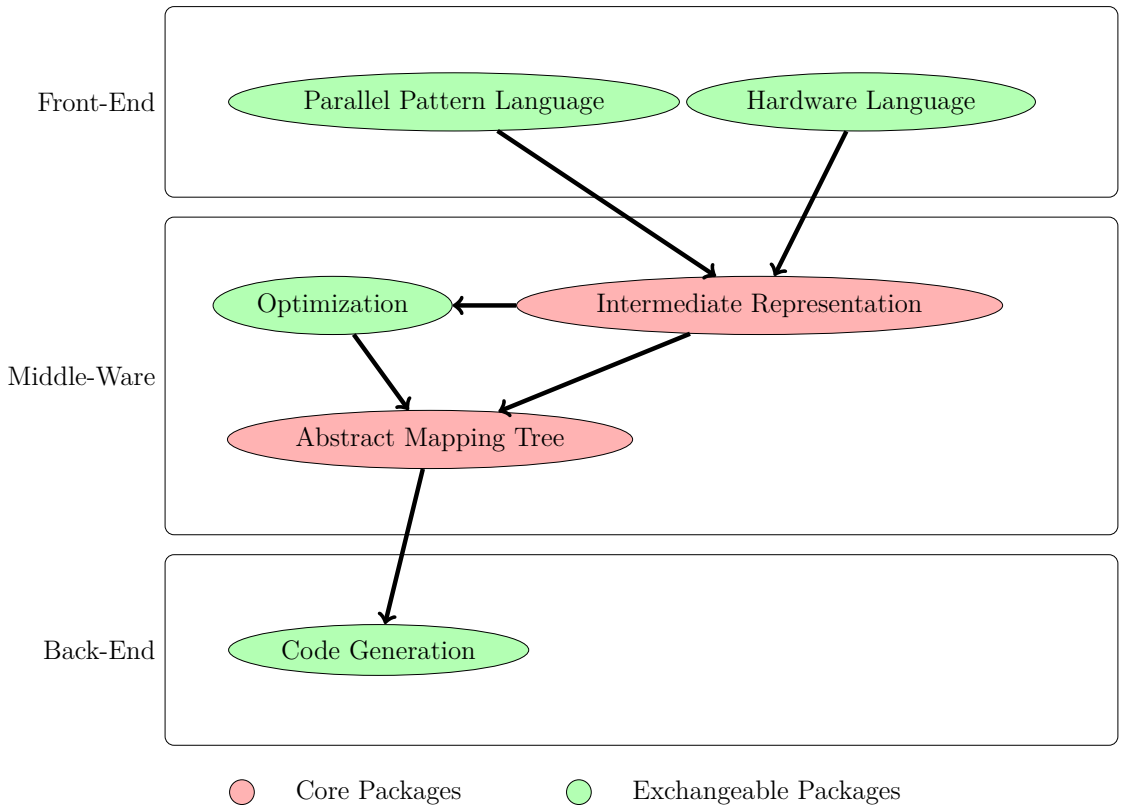


Figure 3.1: Dependencies and information flow within the parallel pattern DSL.

More different output formats are also applicable, since this work generates C++ code combined with PThreads, MPI and CUDA which can be lowered by production compilers.
The front-end packages provide raw information on the algorithmic structure of a given source code via the PPL. The environment description is provided via the hardware language (HL). The HL is crucial for the compiler, since global optimizations strongly depend on the targeted system.
Within the middle-ware the intermediate representation (IR) processes the information provided by the front-end packages to generate further information. Afterwards, this newly generated information is processed by the optimization package into an abstracted result, for instance, a mapping of APT nodes to machines. This master thesis translates the result first into an abstract mapping tree (AMT). The IR is extended by the AMT which generates additional information from the result, namely, data transfer and synchronization steps.
The back-end implementation covers the code generation. The code generation interprets the AMT and generates the output. The code generator referred to in this thesis generates parallel C++ code.

Listing 3.1: Example of a simple map in the PPL

```
MapTest{
// patter definition
map increment([Int] input): [Int] res{
    res[INDEX] = input[INDEX] + 1
}
// program entry point
seq main() : Int {
    // initialize values
    var [Int] initial = init_List([200],1)
    var [Int] result = init_List([200])
    var [Int] result_seq = init_List([200])

    // execute the pattern
    result = increment<<<>>>(initial)

    // checking result
    for var Int i = 0; i < 200; i++ {
        result_seq[i] = initial[i] + 1
        if result_seq[i] != result[i] {
            print("value at element " {i} " is wrong!")
        }
    }

return 0
}}
```

3.2 Parallel Pattern Language

The parallel pattern language is the prototype programming language of this project. It is designed to directly integrate parallel pattern programming. To simplify the analysis and increase the number of potential optimizations, two restrictions were introduced:

1. The data size for all arrays (number of elements) must be statically provided by the programmer.

2. The execution is pure, which means global variables and function parameters can only be read, but not be written to.

The first restriction allows to generate data flow for partial arrays in order to reduce the number of write conflicts and allowing for automatic partitioning approaches of parallel patterns. The second restriction removes most of the occurring parallel write conflicts.

Listing 3.1 provides an example code in the PPL. Lines 3-5 define a map as stated by the keyword `map`. The pattern takes `input` as an input argument and `res` as an output argument. Since `increment` implements a map pattern, the keyword `INDEX` covers every possible value, which does not cause an out of bounds access in line 4. The set of possible index values will be called index range from now on. In line 4 the access is not shifted or scaled, thus the index range only depends on the length of `input` and `res`. The pattern is executed in line 14, where the pattern call statement takes `initial` as a read argument and `result` as a write argument. The size of both arrays is 200 elements, therefore resulting in an index range of 0 to 199 covering all elements in both arrays. The size of all elements is defined and initialized in lines 9-11. The `init_List` function is predefined and returns an array with the shape given by the first argument. Lines 17-22 evaluate and print the parallel result calculated by the map execution in line 14.
The PPL is implemented and transformed to an abstract syntax tree (AST) using the MontiCore language workbench [**?**]. With MontiCore a syntax for the PPL can be defined based on ANTLR [**?**]. Additionally, context conditions can be specified to further restrict the language on a semantic level. Based on the generated AST multiple other features are generated by MontiCore, e.g., a visitor pattern. These additional features allow for a rapid prototype development for the PPL and following steps.

3.2.1 Hardware Language

Another important source of information provided by the front-end is the hardware language. The HL is aimed to describe the target architecture for the optimized code. All usable nodes in a cluster should be specified, since unknown nodes are utilized by neither the optimization nor the code generation. To retain the hierarchy within a cluster, the following components are necessary:

1. **Network**: A network is a collection of nodes specifying a connectivity matrix with latency and bandwidth between all nodes.

2. **Node**: A node corresponds to a singular compute node within a cluster. It contains a set of devices and a connectivity matrix specifying the latency and bandwidth between all devices. Additionally, some node specific information is defined in form of the address and the name.

3. **Device**: A device defines a processing unit like a CPU, GPU or an FPGA and specifies some device specific information like latency and bandwidth to the main memory, main memory size and the type of device it models.

4. **Processor**: A processor is a sub group of execution units on a device that form a Non-uniform memory architecture (NUMA) node. NUMA nodes describe a sub-clustering of execution units with localized main memory. Non-local memory can still be accessed by other processors with an increase in latency

and bandwidth.
Every execution unit may only be part of exactly one processor. E.g. for a given GPU, each streaming multiprocessor can be modelled by a processor. Though, the modelling of processors is defined by the user.

For the implementation of the HL, we utilized the Java script object notation (JSON) format with a slightly modified interpreter to allow the combination of multiple JSON files. This combination allows for identical nodes, devices and processors to be reused.

3.3 Abstract Pattern Trees

The abstract pattern tree (APT) is the intermediate representation of the compiler. In this section, the theoretical concept of the APT will be dicussed. Furthermore, a brief overview of the implementation for the APT will be presented.

3.3.1 Model of Parallel Algorithms

The APT is a tree based abstraction defining the algorithmic structure of a given pattern based algorithm. The concept of the APT was first introduced by Miller et al. [**?**] and implemented by Trümper [**?**]. The APT is defined by its nodes, most notably by the difference between serial and parallel pattern nodes. For serial nodes, the only important property for the optimization is the data flow, defining which data is read or written to by the node. Since serial nodes are not optimized explicitly it suffices to store the data flow. For parallel nodes, the data flow can be used to derive data dependencies between all nodes. After generating the information on data dependencies, the APT is restructured to globally optimize the data flow.
Each node defines a set of child nodes defining algorithmic structure of a single node. These child nodes have to be executed in their defined order. This order can be optimized as mentioned above. The root of the APT is defined by the program-entry-point. The program-entry-point in the PPL is defined by the `main` function.
Listing 3.2 defines a matrix-square function (see lines 14-18) sequentially. The `MatrixSquare` function executes the `MxM` pattern in line 16. The matrix matrix multiplication defined by `MxM` is based on the nesting of two other patterns. The lowest level pattern necessary for the matrix multiplication is the scalar product, defined in lines 2-4. The scalar product is executed within the matrix vector product `MxV` defined in lines 6-8. The `MxV` pattern is again executed by the matrix multiplication `MxM` defined in lines 10-12.
Transforming the example from Listing 3.2 into an APT results in the APT presented in Figure 3.2. The root of the APT corresponds to the main function defined in lines 20-24. The parallel patterns depicted in red specify both the name and the type of pattern used for the implementation of the algorithm. The theory captures sequential execution as a serial node.
In reality structures like an expression and a for-loop cannot be stored and handled

Listing 3.2: Matrix square function implementing MxM with parallel patterns.

```
1  MatrixSquare{
2  reduction scalar([Int] input1, [Int] input2): Int res {
3      res += input1[INDEX] * input2[INDEX]
4  }
5
6  map MxV([[Int]] input1, [Int] input2): [Int] res {
7      res[INDEX] = scalar<<<>>>(input1[INDEX], input2)
8  }
9
10 map MxM([[Int]] input1, [[Int]] input2): [[Int]] res {
11     res[INDEX] = MxV<<<>>>(input1, input2[INDEX])
12 }
13
14 seq MatrixSquare([[Int]] input): [[Int]] {
15     var [[Int]] result = [[0,0,0,0], [0,0,0,0],
           [0,0,0,0], [0,0,0,0]]
16     result = MxM<<<>>>(input, input)
17     return result
18 }
19
20 seq main():Int {
21     var [[Int]] M1 = init_List([4,4],1)
22     M2 = MatrixSquare(M1)
23     return 0
24 }}
```

in the same way, since the behaviour and implications by control flow structures like loops are vastly different from expressions like `a=b+c`.

3.3.2 Implementation

The implementation of APTs solves three major challenges which are adapted by the AMT introduced in this thesis:

1. Differences in serial nodes like we depicted in Figure 3.2 cannot be ignored. As stated before different serial patterns need to be supported in order to properly define loops, branches etc.

2. The index range for each pattern call needs to be derived. According to the index range the input and output data sizes need to be computed.

3. A subset of all parallel patterns supported by the intermediate representation needs to be defined.

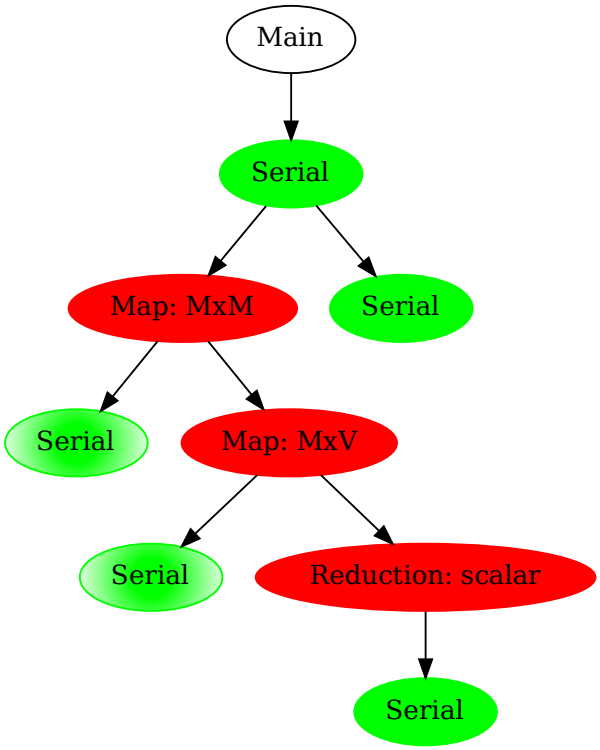


Figure 3.2: APT of a matrix square function, implementing MxM with parallel patterns. The serial child nodes of MxM and MxV define only the parallel pattern call expression evaluation. Thus, they can be disregarded.

All three challenges were solved by the implementation of the intermediate representation. This implementation defines the APT interface this thesis builds upon. The APT implementation differentiates between two types of serial nodes:

- **Expressions** cover the evaluation of them self, e.g., `a = b + 1` is an expression.
- **Statements** define the control-flow of the algorithm. Branches and loops mostly cover this class, but the parallel call is also a statement.

If the expressions are handled separately from the APT, the number of applicable serial nodes is limited. Thus, the serial nodes defined include expression wrapper, control statement classes and function definitions. Please refer to the wiki page in the supplemental material for more details.
The second challenge is directly dependent on the data flow of the source algorithm which is generated within the expression wrappers and in another step communicated to their ancestors. After the communication between each node and their

child nodes, all nodes store their incoming and outgoing data as well as how it is accessed. In the context of parallel pattern definition this analysis regards each individual elements of an array. The details for this analysis are out of scope for this work, but the general idea is discussed in Section 3.2. The analysis mentioned above will provide the information necessary to generate the index range for each pattern call, as well as the input and output array slices. For each data element all accessing expressions are stored. Furthermore, for each node all incoming and outgouing data elements are stored. The stored data elements in combination with the data accesses depict a partial data flow. By combining this partial data flow from all node the complete data flow of the application can be reconstructed.
To efficiently optimize and implement the parallel patterns, restrictions on the supported parallel patterns are necessary. Currently the APT implementation focuses on the four parallel patterns:

- **Map**: The map pattern iterates over every index within the index range and can potentially execute each index in parallel. The example in Listing 3.3 computes an element wise multiplication of two one dimensional arrays.

Listing 3.3: Element wise multiplication of two one dimensional arrays.

```
1 map matrixAdd([Int] input1, [Int] input2): [Int]
    res {
2     res[INDEX] = input1[INDEX] * input2[INDEX]
3 }
```

- **Reduction**: The reduction pattern first executes a map pattern. In the second step, the result of each index within the index range is combined into one. The example in Listing 3.4 describes a reduction over the value `res`. The preceding theoretical map covers the element wise multiplication.

Listing 3.4: Scalar product using a reduction.

```
1 reduction scalar([Int] input1, [Int] input2): Int
    res {
2     res += input1[INDEX] * input2[INDEX]
3 }
```

- **Stencil**: The stencil pattern defines a multidimensional pattern iterating over one index range per dimension. The multidimensional iteration provides a simpler and more flexible way of accessing two-dimensional, three-dimensional arrays than nesting map patterns. The example in Listing 3.5 defines an element wise matrix addition, where `INDEX0` defines the current row and `INDEX1` the current column.

Listing 3.5: Element wise matrix addition.

```
1 stencil matrixAdd([[Int]] input1, [[Int]] input2):
    [[Int]] res {
```

```
2     res[INDEX0][INDEX1] = input1[INDEX0][INDEX1] +
          input2[INDEX0][INDEX1]
3 }
```

- **Dynamic Programming**: The dynamic programming pattern executes a simplified map over a fixed number of time steps. The result of the previous time step is used as the input for the current time step. The example in Listing 3.6 defines the computation of the Fibonacci numbers. The implementation computes the Fibonacci numbers n and $n+1$ for a number of time steps n, defined only in the pattern call. The Fibonacci computation assumes an initial input of `[1,1]`.

Listing 3.6: Dynamic programming Fibonacci.

```
1 dp fibonacci([Int] input):[Int] res {
2     res[0] = input[1]
3     res[1] = input[0] + input[1]
4 }
```

These four patterns cover a wide range of algorithms, e.g., neural network training, computer graphics and matrix operations.
To avoid data races, only local variables and the return variable can be written to. Furthermore, the return variable should be written to exactly once. The correctness of the generated code must not be broken. Therefore, these restriction are defined in order to ensure generation of correct synchronization and data transfers. An evaluation of the correctness of the generated code will be discussed in Chapter 6.

3.4 Optimization

The Optimization proposed by Miller et al. [**?**] and implemented by Trümper [**?**] is focussed on global optimizations. To perform the global optimizations, the APT is first transformed into a FlatAPT.

3.4.1 Optimizations

The optimizations implemented by Trümper [**?**] are realized in the FlatAPT. Currently the optimizations include re-ordering, pipelining, non-uniform memory architecture (NUMA) awareness and a hardware mapping.

- **Re-ordering**: Creating the FlatAPT from the basic APT by sorting the APT nodes in relation to the data flow. The nodes are ordered to be executed as soon as possible. The generated order maximizes the potential parallelism by creating "steps" where all nodes are independent from each other, similar to bulk-synchronous programming [**?**], where the complete program synchronizes at predefined steps.

- **Pipelining**: This optimization fuses parallel patterns, if the output of the predecessor is identical to the input of the current node. The indivdual parallel patterns stay unchanged in this step. This reduces the number of potential mapping results and signals the code generation, that no synchronization needs to be generated between these parallel patterns.

- **Hardware mapping**: The hardware mapping generated by this optimization minimizes the total amount of data transfers between devices and/or nodes. In this step, parallel patterns may be divided to increase the level of parallelism. To reduce the complexity of this optimization, minimal split sizes for parallel patterns and data elements are defined. The optimal placement of the parallel patterns on the hardware is based on a simplified machine model based on the hardware language. Currently, the optimization is realized by utilizing a roofline model [**?**] and solves a mixed integer linear problem (MILP) by using the Gurobi solver [**?**].

- **NUMA-awareness**: A NUMA system is a system with a non-uniform memory architecture. The NUMA-awareness is optimized, by regarding the distance between the different processors defined in the hardware language.

3.4.2 FlatAPT

The FlatAPT is a linearized version of the APT based on the root node of the APT. The FlatAPT in combination with the mapping table, which stores the optimized hardware mapping for the individual parallel patterns, define the interface to generate the resulting code. The FlatAPT itself stores a sorted list of steps generated by the reordering optimization. The mapping table contains the results from the hardware mapping in form of a look-up table. Each parallel pattern is assigned to a processor in this mapping table. For the parallel patterns a set of start-point and iteration counts are defined. The size of the set is defined by the dimensionality of the pattern, e.g., a map pattern may only have a single start and iteration count, while a stencil pattern will have n elements for an n-dimensional input array. The input and output data is also defined as data-splits referencing the corresponding data element and defining the a fixed slice, if the data element is an array.
The pipeline optimizations are realized by introducing two types of parallel pattern mappings:

- **Pattern mapping**: The normal pattern mapping corresponds to a single parallel pattern which will be executed on the processor defined in the mapping table.

- **Fused-pattern mapping**: A pattern mapping containing an ordered set of parallel patterns combined by the pipelining optimization. The patterns use the output of the direct predecessor based on the data splits.

The transformation of the FlatAPT into the abstract mapping tree (AMT) will be discussed in Chapter 4.

3.4.3 Target Architectures

The system targets three different forms of parallelism which all need to be handled appropriately:

- **Shared-memory parallelism** defines the utilization of multiple threads on a CPU, all threads can access the same memory space. Data accesses without proper synchronization can induce data races. The CPU can directly access the main memory and loads small chunks of the data into their local caches. Changes made by one thread are directly visible by all other threads on cache coherent systems. Systems without cache coherency do not communicate changes for cache local data. NUMA architectures define a sub-clustering of memory and cores. These sub-clusters are typically aligned improving the performance by utilizing the distances between sub-clusters of cores and memory.

- **Distributed-memory parallelism** defines the utilization of multiple nodes via an interconnect. Since these nodes do not share the same memory space, necessary data and changes in data need to be communicated by the user. Thus, besides synchronization it is necessary to send and receive data between different nodes. These nodes might not be identical and need to be differentiated. The individual nodes act independent from each other, besides the communication.

- **GPU offloading** defines the utilization of GPU architectures. A compute node can incorporate multiple GPUs. GPUs have a reduced control logic in comparison to CPU architectures. Data transfers and memory allocations concerning the GPU must be handled by the CPU. GPUs are optimized to perform highly parallel computations with identical data access patterns, e.g., matrix operation for image processing or neural network training. Modern GPUs are structured into multiple streaming multiprocessors. Modern architectures like the Nvidia V100 [**?**] consist of 80 streaming multiprocessors with 64 FP32, 32 FP64, 64 INT32 and 8 Tensor cores per streaming multiprocessor. The Intel Xeon Platinum 8160 [**?**] CPU in comparison supports 24 cores and 48 threads with hyper-threading. GPUs execute the code in subsets of threads called warps. For Nvidia GPUs a warp is typically of size 32, all threads within a warp execute the same code and can share a program counter.

4 Abstract Mapping Tree

The abstract mapping tree (AMT) is an additional abstraction level between the optimization of the abstract pattern tree (APT) and the actual generation of (source) code. This additional layer of abstraction helps integrating the global optimizations, while also restructuring existing information. The structural changes are made in order to adjust the AMT to the task of code generation, while the APT is used to provide different information to the optimization. The AMT is the first contribution of this thesis to the overall project.
This chapter lines out the structure of the AMT. It explains how the implemented global optimizations [**?**] is transformed into the AMT structure. A debug transformation which directly transforms the APT into an AMT, to help testing the generated code, is also covered. Finally, the approach of generating synchronization and data transfer mechanisms used in this thesis is discussed.

4.1 AMT Structure

The AMT is directly related to the APT in terms of structure. Therefore, the AMT is implemented in a tree structure which defines the targeted application. The root node specifies the program entry point. The different nodes can be distinguished into four different categories:

1. **Serial nodes** cover the behaviour of simple sequential pattern such as: loops, branches and expressions. This kind of node is mostly identical to their APT node counter parts.

2. **Function nodes** define a static sub-tree, which can either be a sequential function definition or a description of a parallel pattern:

   - **Sequential function nodes** define the sub-tree executed when encountering a serial call node.
   - **Parallel pattern nodes** specify the sub-tree covering the individual parallel pattern implementations. Since the different parallel patterns differ in their behaviour, for each pattern a parallel pattern node type is defined, e.g., a stencil node is different from a reduction node (See Section 3.3.2).

   The function nodes are defined in order to specify templates for the execution of call nodes. This reduces the memory usage and complexity of the AMT. Function nodes in the AMT are similar to function nodes in the APT.

3. **Call nodes** are the entry point for function nodes. They specify the data flow into and out of their corresponding function node. Sequential function nodes are self explanatory. In this context, the parallel pattern calls are important, since they combine the information from the global optimizations with the original APT. They define a target processor and an index range regarding the original computation. To differentiate the behaviour of the parallel calls additional variants are used extending the basic definition of a parallel call. The following types of parallel pattern calls further define specific variants:
    - **Parallel calls** are the basic definition of a parallel pattern call. Besides an index range covering multi-dimensional patterns, synchronization and data transfer schemes for dynamic programming patterns are stored within the data structure.
    - **Serialized parallel calls** define parallel pattern calls, that should be executed sequentially. They extend the basic parallel call to signal the sequential execution of a parallel pattern.
    - **Reduction parallel calls** specifically define the execution of the reduction pattern as explained in Section 3.3.2. The reduction pattern combines the results of every iteration into a single result. Therefore, the recombination of partial results is also subject to optimization. By defining temporary input and output data to express the recombination hierarchy between multiple reduction parallel calls, the globally optimized reduction can be expressed. The temporary data represent the partial results computed by different reduction parallel call nodes.
    - The **fused parallel call** is special, since it is not a single parallel call node. The fused parallel call is an ordered set of other parallel call nodes, which are executed via pipelining. Thus, specifying that no synchronization or explicit data transfers are necessary between these nodes.
    - **GPU parallel calls** are parallel pattern calls to be executed on a GPU. As introduced in Section 3.4.3 the GPU architecture is different from the CPU. Due to the difference in architecture parallel patterns executed on the GPU are combined into GPU parallel pattern calls. The combination of multiple GPU calls allows for the utilization of more streaming multiprocessors increasing the performance of the GPU. Therefore, this node is mainly used to improve the performance on a GPU architecture.
      For other different architectures like FPGAs it might also be applicable to introduce new nodes.

4. **Data control nodes** specify the synchronization and data transfer mechanisms. They are generated within the AMT and can be differentiated into as follows:
    - **Barrier nodes** describe a synchronization scheme defined by a set of processors on different machines, that need to be synchronized.

- **Data movement nodes** define a set of destination placements and a set of source placements. Thus, determining the data needed to be transferred before the next AMT node. A placement in this context is a data element and a set of start and length values specifying the subsets of the data to be transferred.

AMT nodes are executed from left to right to emphasize the order of execution of child nodes. An example for an AMT is depicted in Figure 4.1b, where the green nodes define serial nodes and the blue node depicts a GPU parallel call node.
In addition a visitor implementation generating the shape of array data, for traversing the AMT is defined. This visitor strongly simplifies the actual code generation process defined in Chapter 5. The implementation of the visitor allows for a standardized traversal. Therefore, the generator can be implemented by defining how the AMT nodes are transformed into source code.

4.1.1 Visitor Pattern

To properly traverse the AMT, while optimizing the maintainability of the code the use of design patterns is applicable. The notion of design patterns including the visitor pattern, discussed by Gamma et al. [**?**] is especially useful to support maintainability. For a traversal of a graph or tree based structure the visitor pattern is the most suitable [**?**]. The implementation of the visitor pattern is based on the AST visitor generated by MontiCore [**?**], by defining an interface where each AMT node implements the following five functions by overloading the signature:

- `accept`: This function defines the entry point for an AMT node. The function needs to be implemented within every AMT node and accepts a given visitor implementation. When this function is called the current node will be traversed.

- `visit`: The visit function is called when the given node is first encountered. Thus, the function is computed before the child nodes of the given node are encountered. The default implementation for this function is empty.

- `endVisit`: The implementation of the endVisit function is executed just before the visitor leaves the node. All descendants of this node are encountered before this function is called for the current node. The default implementation for this function is empty

- `traverse`: This function defines the traversal strategy of the current node. By calling `accept` for each child node the current node possesses, the sub-AMT spanned by the current node can be traversed. The traverse function can be used to specify the order in which the child nodes are traversed or if they should be traversed at all. The default implementation will first cover nodes explicitly defined in the AMT definition and then iterate over the list of all remaining child nodes.

- `handle`: The handle function is implemented as a part of the visitor definition. It takes the current node as an argument and ensures the correct order of the `visit`, `traverse` and `endVisit` functions. The default implementation only refers to the argument of those specific functions.

This interface for the visitor pattern allows the implementation of specific visitors for each node type and can be extended for new AMT nodes by implementing the five functions described above.
Since the generator needs information on the shape of a variable at all times, which is not ensured within function and pattern definitions, the shape needs to be computed within the visitor.
To further support the implementation of the generator an extended visitor was implemented. For the generation of source code it is currently necessary to know the shape of every data element. The utilization of function nodes and call nodes decouples the data flow from the definition of function templates. Therefore, the shape of parameters needs to be replaced by the shape of the argument during traversal. This allows for the reuse of existing algorithms for the AMT within function nodes. Recursive calls are not covered by this, since the visitor is incapable of branch prediction. This would lead to indefinite repetitions in the traversal of the same node. The extended visitor extends the basic visitor definition and overwrites the `handle` function for function and parallel call nodes.
The implementation of these `handle` functions evaluates the shape of the given arguments and replaces the dummy shape defined within the function parameters with the current shape of the arguments. Thus, the information on the shape of all data elements is clearly defined before encountering them. To ensure the correct behaviour of this extension, the handle function may not be overwritten when implementing an instance of the extended visitor.

4.2 Transformation to AMT

The automatic generation of the AMT is a complex task which can be split into two smaller tasks:

1. The generation of the basic AMT without synchronization and data transfers.
2. The subsequent generation of synchronization and data transfers, as explained in Section 4.3.

To support a decent evaluation of correctness of the generated code, without considering potential implications induced by the global optimizations, two different transformation processes were developed and implemented over the course of this thesis.
The first process defines the transformation of the global optimizations into an AMT. This transformation can be used for a complete compiler.
The second process implementation realizes a debug mode for the code generator.

This debug mode evaluates additional parameters from the front-end to create an artificial mapping. The artificial mapping can be used to evaluate the correctness of the generated code in different circumstances. In Chapter 6 this feature is utilized for such an evaluation.
In the following two subsections the transformations into an AMT are explained in more detail.

4.2.1 Optimization to AMT

The first transformation process converts the global optimizations and the APT into an AMT. This transformation is strongly dependent on the implemented optimization package. At the point of writing this thesis only the implementation by Trümper [**?**] was available providing the necessary information for the transformation. This implementation of the global optimizations introduces two important data structures:

- The **FlatAPT** is an optimized version of the original APT. To improve the global data flow of the application the direct descendant of the APT root is abstracted into an ordered list of steps. A step is an unordered set of APT nodes that are pair-wise data independent. Thus, the nodes within a step can be executed in an arbitrary order. However, they must be finished before the start of the following step.

- The **mapping** is a set of hash maps assigning different parallel pattern splits to execution units. The different hash maps are also separated into consecutive steps of parallel assignments.
  - **Pattern splits** are wrapper for the partial execution of a parallel pattern.
  - **Fused pattern splits** define a sequence of pattern splits which should be executed in direct sequence without synchronization.

The AMT is generated by traversing the FlatAPT. As mentioned in Section 4.1 serial nodes and function nodes can easily be transformed by copying important information and by transforming their children.
Parallel call nodes are mostly generated from the parallel pattern splits. In the following passage, it is briefly explained how the different parallel call types for the AMT are generated:

- Serialized parallel calls are generated when a parallel execution is not appropriate.

- Parallel calls are created when a pattern should be executed in parallel on the CPU.

- GPU parallel calls are used when a pattern should be executed on a GPU in parallel.

- Reduction parallel calls are utilized for reduction pattern calls regardless of their execution device.

- Fused parallel calls are generated for fused pattern splits.

Two major challenges must be handled in order to implement the AMT generation. The first challenge is that the steps of the mapping and the FlatAPT are not aligned, since the FlatAPT contains steps with serial nodes which are skipped for the mapping. This can be handled by searching the parallel patterns splits from the FlatAPT within the mapping.
The second challenge is induced by computation on GPUs. To properly utilize the performance of the GPUs architecture, pattern splits targeting the GPU should be fused. The fusion of GPU nodes is recommended, since the global optimizations map a pattern split only to a single streaming multiprocessor. Thus, a lot more compute power can be utilized when fusing GPU nodes.
The APT only supports linearly aligned data accesses within patterns. Therefore, input data which is aligned in their placement, results in linearly aligned output data. Thus, the combination of multiple pattern splits considering the following three conditions internally, enlarges the index range of a pattern and allows to utilize architectural benefits of the GPU, e.g., the large amount of threads. Furthermore, the restriction to linear data accesses enables the horizontal fusion of fused parallel call nodes. This horizontal fusion can only be performed, if and only if these three conditions are met:

1. The pattern splits must target the same GPU, since the utilization of a single GPU should be improved.

2. The pattern splits must be based on the same APT node, because only identical definitions can result in identical GPU kernels.

3. The pattern splits must be "connected". Which means that the index range of the pattern splits must be combinable without creating any gaps, e.g. "0,1,2" and "3,4" are combinable, but "0,1,2" and "5,6" are not, since values 3 and 4 would be missing in the index range. This condition must hold, since the generation of GPU kernels cannot handle gaps in the index range, yet. The condition could be omitted if GPU call nodes support multiple index ranges and the generator could statically shift the data accesses based on the thread_id. The implementation of a multi index range GPU call node is complicated and could lead in the worst case to large memory overheads during the source generation.

By identifying these three conditions an algorithm was designed which meets all three conditions in the above presented order. The algorithm performs the following steps:

Listing 4.1: A simple Map.

```
1  TransformationSample{
2  map add1([Int] input) : [Int] res {
3      res[INDEX] = input[INDEX] + 1
4  }
5  seq main():Int {
6      var [Int] initial = init_List([LARGE_SIZE],1)
7      var [Int] result = init_List([LARGE_SIZE],1)
8      result = add1<<<>>>(initial)
9      return 0
10 }}
```

1. Splitting the set of all pattern splits available in the current step until the three conditions for each subset are satisfied.

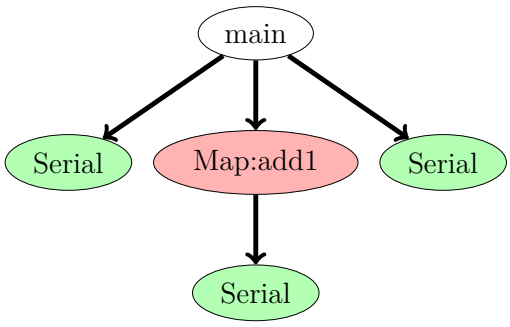
2. Ordering the subset based on the index range when solving the third condition. This maximizes the size of the subsets.

3. Combine the pattern splits targeting the same GPU, if they are "connected".

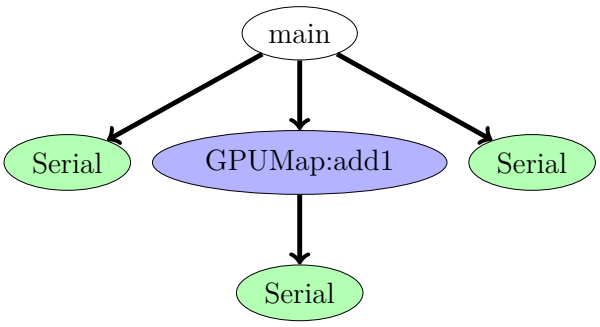
4. Create a GPU pattern call based on the combined pattern splits and the optimized call.

The algorithm is implemented during the generation of the AMT and can therefore directly generate the fusing of GPU call nodes without considering the parallel pattern splits individually.

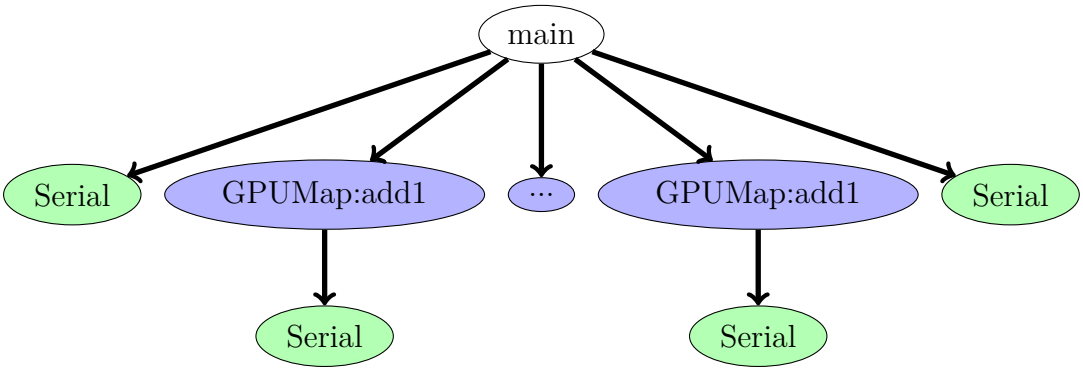
As an example, to depict the AMT generation, in Listing 4.1 a large map is defined which increments the values of `initial` and stores the resulting array in `result`. The corresponding APT is depicted in Figure 4.1a, where the left most serial node defines the code in lines 6 and 7, the map in line 8 and the right most serial node line 9. The serial node below the map node is defined by the expression in line 3. Considering a system with a single GPU, the global optimizations would utilize it as much as possible, if the arrays are large enough. Assuming that the `LARGE_SIZE` place holder is sufficiently large for GPU execution, the global optimizations would map the execution to the GPU. Thus, creating a FlatAPT similar to the one depicted in Figure 4.1c. The FlatAPT presents a set of map patterns mapped to the GPU. To avoid a too large depiction, additional GPU maps are combined into the (...) node. Based on the FlatAPT a basic AMT without data transfers and synchronization is generated. The AMT in Figure 4.1b illustrates this basic AMT based on the FlatAPT. Since our target architecture only comprises a single GPU, all GPU nodes from the FlatAPT are combined into a single node within this basic AMT. The generation of the synchronization and data transfers will be further explained in Section 4.3.



(a) APT based on Listing 4.1.



(b) Basic AMT based on Figure 4.1c.



(c) Optimized APT based on Figure 4.1a.

Executed with one thread

Parallel APT node

Executed on the GPU

Figure 4.1: APT, FlatAPT and AMT definitions based on the on the code in Listing 4.1.

4.2.2 Debug Mode

The second realization of the AMT generation is the debug mode defined and implemented in the context of this thesis. The goal of this generator is to artificially create scenarios, where the synchronization model and the code generator must create certain global optimizations. This includes the coverage of data transfers and synchronization between different nodes and device. The debug mode is independent from the global optimizations. Thus, different machine models and optimization strategies do not influence the AMT generated with the debug mode. The quality of the generated code in terms of correctness can then be evaluated without depending on global optimizations. To reduce the complexity of this transformation only the hardware mapping is recreated artificially. Reordering and NUMA-awareness optimizations (see Section 3.4.1) are omitted, since they do not impact the correctness of the application. Pipeline optimization can affect the generation of synchronization points, but generally only affect the performance of the execution.

Listing 4.2: A simple Map transformed by the debug mode.

```
1  TransformationSample{
2  map add1([Int] input) : [Int] res {
3      res[INDEX] = input[INDEX] + 1
4  }
5  seq main():Int {
6      var [Int] initial = init_List([LARGE_SIZE],1)
7      var [Int] result = init_List([LARGE_SIZE],1)
8      result = add1<<<[0,0,0],[1,0,0]>>>(initial)
9      return 0
10 }}
```

Therefore, pipeline optimizations are omitted as well.
An individual mapping is defined as a triple `[N,D,P]`:

- **N** defines the node the call should utilize.
- **D** defines the device the call should utilize.
- **P** defines the processor the call should utilize.

The additional arguments are utilized to artificially assign these mappings to a parallel pattern. The additional arguments are defined within the `"<<< >>>"` caption and can contain arbitrarily many mappings. If more than one mapping is defined the index range is divided between the different mappings. Listing 4.2 defines the same basic example as Listing 4.1. The codes differ in line 8, which defines the additional arguments for Listing 4.2. The example in Listing 4.2 defines two mappings: `[0,0,0]`, `[1,0,0]`. Thus, the debug mode splits the index range into two equal parts. The first part, defined by `[0,0,0]`, executes on the first processor of the first device on node 0. The exact definition of the hardware depends on the input provided by the HL (see Section 3.2.1). The second part, defined by `[1,0,0]`, executes on the first processor of the first device on node 1. This pair of mappings enforces the generation of data transfers and synchronizations between two nodes by the synchronization model.
The creation of an artificial mapping replaces the transformations required from the optimized code. The generation of the structure of the AMT and the node definitions is the same for the debug mode and the optimization transformation.

4.3 Synchronization Model

The last step of the AMT generation is the generation of synchronization points and data transfers. As part of this thesis a modular synchronization package was implemented to generate synchronization and data transfer nodes.

The synchronization model is implemented as a standalone tool utilizing the AMT implementation. The AMT does not specify the implementation of the synchronization model, as such the synchronization model can be exchanged by different implementations.
In this section the implementation developed over the course of this thesis will be discussed.
To generate a synchronized AMT based on a given basic AMT, two requirements need to be satisfied:

1. The basic AMT needs to be flat, e.g., all parallel calls must either be direct descendants of the root node or be executed sequentially. This requirement may not be necessary for other implementations of a synchronization model. For the approach implemented in this thesis it strongly reduces the complexity of the task without restricting the application programmer. A flattening of the APT is both implemented for the APT and the global optimizations. Thus, this requirement is sufficiently fulfilled by previous steps.

2. Each node needs to exactly define the incoming and the outgoing data flow. The information on the data flows is generated within each node of the AMT based on the data accesses within the node and the index range, e.g., for the example in Listing 4.2 the synchronization model would define `initial` from 0 to `LARGE_SIZE - 1` as the incoming data flow for the complete pattern call in line 8, disregarding the pattern split by the additional arguments.

The optimization of synchronization and data transfer nodes is out of scope for this thesis. Thus, the model generates the synchronization and data transfer nodes as soon as possible avoiding potential deadlocks and data races. The execution of serial nodes must be done on the default device. The default device is generally defined to be the first CPU of the node with rank = 0.
The code generation is defined by the following execution steps:

1. **Parallel Groups**: In the first step the parallel pattern calls within the AMT are grouped according to their former APT node. These parallel groups are used to avoid early synchronization and subsequent data races or deadlocks. The parallel group associated with a parallel call node is always known by the node. For each parallel group it is also known if a member is the first to be handled or the last. Therefore, data transfers and synchronization can be generated for each parallel group. The data needed by all members of the group can be transferred before the first member begins computation and synchronization steps affecting all members can only be done once each member has started the computation.

2. **Definition of the Initial Flow**: The second step creates initial values for the current placement of all variables available in the scope of the root node. The initial placement for each variable is defined to be in the main memory of the default device.

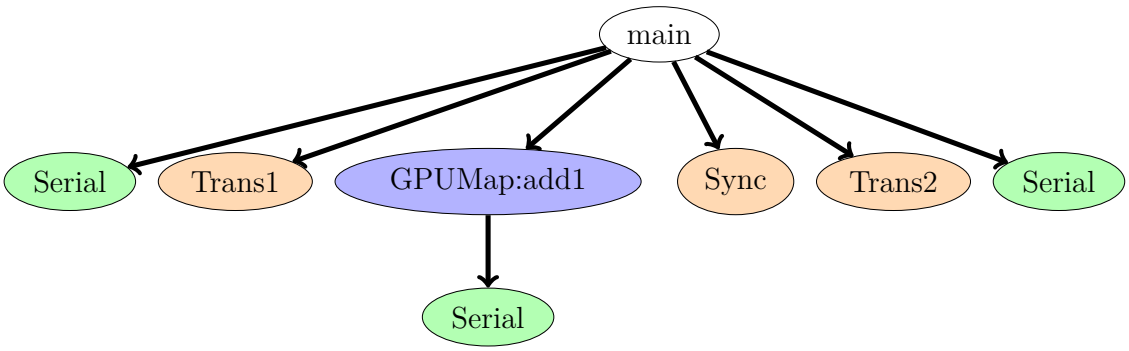


Figure 4.2: AMT with Synchronization and data transfers based on Figure 4.1b.

3. **Node Traversal**: The following steps are repeated for each child node of the root. The order of the traversal is defined by the order of the nodes within the basic AMT.

4. **Dynamic Programming Handling:** If the current node is a parallel pattern call referencing a dynamic programming pattern, data transfers and synchronization within the time step loop are generated. The synchronization specifies all execution units taking part in the dynamic programming recursion. The data transfer sends all written data to all participating execution units. The generated nodes are added to the specification of the parallel call node.

5. **Synchronization Point**: The current data placements are compared with the data flow of the current node. If the current node is a parallel pattern call and all other members of the same group were already traversed, all execution units used by the parallel group are synchronized. This step creates a barrier node after the current node.

6. **Data Transfers**: The current data placements and the data flow of the next child node are compared. The incoming data flow of the next node and the current data placement detects overlapping data slices on different devices. In this case a data movement is generated after the current node specifying the data do be transferred. Since data placements can be arbitrarily small they are maximized in order to reduce the overall latency.

7. **Update Placements**: The current data placements are updated in relation to the data transfers occurring and write accesses discovered in the previous steps. The new data placements are maximized in order to reduce the runtime of the algorithm. Data placements written to in the current step are erased from all other devices.

Figure 4.2 depicts a synchronized AMT based on the AMT in Figure 4.1b. The AMT was generated by an implementation of the algorithm above. The first data movement node in Figure 4.2, named *Trans1*, defines the data transfer from the CPU to the GPU. The added barrier node *Sync* defines the synchronization of the CPU

and the GPU. The synchronization is necessary to ensure that the computation on the GPU finished before utilizing the result. Afterwards, the second data movement node *Trans2* defines the data transfer from the GPU to the CPU in order to evaluate the results on the CPU.

5 Code Generation

The code generator designed and implemented in this thesis, defines and executes the transformation from the AMT into a code which can be interpreted by a machine. As such, it is possible to generate binary files which can be executed directly by the machine. Another possible approach is the definition and implementation of a source-to-source compiler. For a source-to-source approach the generator transforms the AMT into optimized source code.
In this chapter, the requirements for the implemented code generator are defined. Afterwards, the design decisions and the corresponding implementations are discussed. Finally, the structure of the generator is explained.

5.1 Requirements

The concept of the code generator is dependant on the AMT extending the IR. The AMT itself is designed to cover features from the optimizations implemented by Trümper [**?**] and the available hardware. This leads to a set of requirements induced by the optimization package, the targeted hardware and the IR.

5.1.1 Hardware

The hardware currently targeted by the overarching project covers CPUs and GPUs on different machines. Therefore, it is required to also generate code targeting those systems.
Regarding the architectural differences between the targeted systems the following requirements for the generated code can be derived:

- The generated code should at least support shared-memory parallelism, distributed-memory parallelism and GPU offloading.
- The generated code needs to be able to synchronize an arbitrary subset of devices with each other.
- The generated code must be able to transfer data between an arbitrary pair of devices.

These three requirements define the minimal capabilities necessary to correctly utilize the targeted architectures. The synchronization of multiple device is necessary in order to avoid data races between computations depending on each other.

At this point it is assumed that each computation always fully utilizes the device. The data transfers between devices are necessary because the GPU typically cannot access the main memory. The CPU has to handle the data transfers to and from the GPU. Additionally, machines do not share a memory space in general. Therefore, the necessary data needs to be actively communicated between machines.

5.1.2 Optimization Representation

The optimizations implemented by Trümper [**?**] are represented by the AMT, e.g., pipeline optimizations are defined in fused parallel call nodes and the mapping is represented by an assignment to a processor within the HL (see Section 3.2.1).
The following requirements are derived, in order to correctly represent the optimizations in the generated code:

- **Shared-memory parallelism:**
  Each processor defined in the HL must be accessible. This requirement is necessary in order to correctly implement the hardware mapping, since tasks are assigned to these processors.
  It has to be clarified on which physical core a thread is executed. In order to correctly generate the optimized hardware mapping, since the costs for moving data from one processor to another might be different. For example, NUMA nodes can have different latencies and bandwidths between different pairs of sub-NUMA clusters.

- **Distributed-memory parallelism:**
  Each node must be accessible individually in the order defined in the HL. Therefore, it has to be explicitly clarified during the generation and optimization which node handles which tasks. This has an impact on the communication cost between nodes and can influence the correctness of the computation, if the used nodes are not identical.

- **GPU offloading:**
  The generated code must have access to each device specified in the HL and can be used individually in order to realize the defined hardware mappings.

- **Pipelining:**
  Parallel calls within a pipeline should not be synchronized with each other, since the optimization already discovered, that the structure is synchronized implicitly. Further, the parallel calls are capable of implementing a loop fusion behaviour.

These requirements are represented within the AMT, but are originally derived from the optimization.

5.1.3 Intermediate Representation

Most requirements are indirectly dependant on either the targeted hardware or the optimizations. Although, some design decisions for the IR directly affect the generated code. These requirements are derived from the APT and AMT definitions and are defined as follows:

- **Data structures**: The IR is currently only capable of utilizing multi-dimensional concatenated data structure, e.g., arrays. Therefore, it is sufficient to support only these data structures within the generated code.
- **Memory management**: The memory management is not explicitly stated by the IR. In order to generated correct code, the code generator must be capable of handling the memory allocation and deallocation during the generation.

5.1.4 Design Decisions

Based on the requirements stated in the previous sections, the following design decisions for the generated code must be made. These design decisions focus on both the code generator and the generated code:

1. Selection of a target language.
2. Selection of programming models for the individual target architectures.
3. Selection of data structures utilized by the generated code.
4. Generation of the parallel patterns based on the target architecture and the programming model.
5. Definition of the memory management for the different architectures.
6. Selection of strategies ensuring correctness of the generated code.

These design decisions are discussed in the following sections as well as the implementation of the selection.

5.2 Target Language

The target language is the most basic decision made in this thesis, since it affects all following decisions. By considering the performance of the generated output the following three alternatives are the most applicable for the generated code:

- **Binary Code**: The first alternative, is the creation of executable binary files. This approach is highly complex and time consuming to implement. The generation of binary files would only benefit from the global optimizations performed by the optimization package. Existing local optimizations like

vectorization would have to be implemented specifically for a generator. The implementation of these local optimizations would be necessary in order to achieve performance results comparable to existing production compilers and frameworks such as the GNU compiler, the Intel compiler or LLVM compiler frameworks.

- **Fortran**: The generation of Fortran code is another alternative. This approach makes use of source-to-source compilation. With source-to-source compilation the generator produces parallel Fortran code which can be compiled by production compilers to utilize local optimizations. The generation of Fortran code allows to reuse most of the more well-known parallel programming models while also creating fast code with production compilers.

- **C/C++**: C code supports a similar set of parallel programming models as Fortran and is another source-to-source compilation approach. Furthermore, C code also performs similar to Fortran. C++ code supports the same concepts as C code, while providing a much larger variety of standard functionalities.

C++ provides a lot of functionalities, which greatly simplify the realization of IO and array functions, especially the utilization of C++ templates was a deciding factor.
Therefore, C++ as a target language is utilized for the code generator to be applied in this thesis.

5.3 Shared-Memory Parallelism

5.3.1 Programming Model

As stated in Section 5.1, the generated parallel CPU code must fulfil two requirements defined by the optimization. Firstly, the individual threads must be mapped to the hardware in order for the optimizations to target the hardware correctly. Secondly, each thread must be accessible individually (or at the very least in small groups defining a processor in the HL) in order to correctly realize the optimization with the generated code.
Hardware dependant programming models like the Cray XMT [**?**] are not applicable for the code generator, since they cannot support different architectures. Programming models like MapReduce [**?**] or fork-join [**?**] are not capable of defining different parallel patterns efficiently, in terms of performance or implementation complexity. The two most applicable parallel programming models for the generated code are OpenMP and PThreads.

- **OpenMP** [**?**] is a very prominent CPU parallel programming model. The OpenMP programming model introduces compiler directives to automatically parallelize structured blocks. Thread pinning with OpenMP is also possible. But assignments of tasks to individual threads is not possible with OpenMP.

- **PThreads** [**?**] is defined by the C standard for UNIX systems and provides an interface for using the underlying hardware threads. By utilizing OS-callbacks like `pthread_setaffinity_np` the newly created threads can be pinned to specific cores. PThreads only supports a single task per thread. A threads pool implementation would be necessary to reuse existing threads avoiding additional overhead. Existing thread pool implementations mostly support a single task queue shared by all threads. To assign the tasks to an explicit thread a new thread pool implementation is necessary.

It is necessary to assign a task to a specific subset of threads which is not possible with OpenMP. It is possible to define the mapping of threads to specific cores and utilize a subset of all possible threads to handle a given task with OpenMP. Although, the arbitrary assignment of these subsets is not possible with OpenMP. While generating OpenMP code would definitely be the simpler task, the goal of this thesis is to generate code based on the given optimizations. Thus, a programming model which cannot express the assignment of tasks to threads on the same level as the optimization is not desired. The ability to reuse dynamic optimizations and other features from OpenMP are beneficial. To meet the requirements and model the generated code as closely to the optimization as possible PThreads was chosen.

5.3.2 Implementation

The thread pool implementation is important for the CPU parallelization. As mentioned in Section 5.3.1, PThreads is utilized to implement a thread pool and CPU parallelization. The thread pool is designed to run each thread individually. The approach followed in the implementation provided by this thesis defines a work queue for each thread where only the main thread adds tasks. Therefore, dynamic parallelization approaches are not implemented for this thread pool. It is possible to improve the implementation by utilizing the HL structure of processors. Since, a single processor contains multiple cores, a hybrid implementation could be used. This hybrid implementation would define a task queue for each processor. Thus, the parallel CPU code would benefit from both dynamic and static optimizations. To properly utilize the thread pool it needs to be initialized before the parallel execution and destroyed when the computation is finished. The thread affinity is defined during the initialization and pins thread 0 to core 0, thread 1 to core 1 etc. Currently, multi threading is not supported, but could be added by introducing an additional parameter defining the number of hyper threads per core.
The thread pool uses a C++ standard vector to define the individual threads. Each thread consists of a set of attributes wrapped in a C++ class:

- **thread**: The actual pthreads thread object executing the tasks.
- **workBuffer**: A vector of C++ standard functions defining the task queue of the thread. The queue is implemented as a ring buffer with a fixed size.
- **head**: The index of the current first element of the ring buffer.

- **tail**: The index of the current last item of the ring buffer.
- **size**: The number of tasks within the ring buffer. It is used to avoid corner cases and error messages on an empty or full queue.
- **result**: The result of the thread creation, used to handle errors from pthreads.
- **concluding**: A boolean value indicating if the pool should get destroyed once the task queue is empty.
- **queueLock**: A mutex to avoid data races when accessing the task queue. A lock free implementation would be preferable.

In order to avoid the synchronization of more threads than necessary, only the threads inducing a data race need to be synchronized. To realize this task, the bit mask is implemented to define synchronization. The bit mask data structure defines a pthread barrier and a bit mask defining the synchronization partners. The synchronization is then realized by adding a task executing the barrier to the task queue. If necessary, the threads can be synchronized just with each other or with the main thread.
The thread pool can be initialized by specifying the number of threads and a vector containing the threads. Once the pool is created the execution behaviour needs to be initialized via a function call. The function defines the thread affinity by setting an affinity mask for each threads and enforcing the affinity with the `pthread_setaffinity_np` OS callback. The function further initializes the queue lock and handles errors during the initialization. The destruction of the thread pool first tells all threads to finish their task queue. Once, a thread has finished its tasks it joins the main thread again. The thread pool is designed to support an individual thread pool for each node specified in the HL.
In Listing 5.1 the utilization and initialization of a thread pool is defined. The numbers are based on two nodes from the CLAIX 2018 cluster [**?**] equipped with two Intel Xeon Platinum 8160 with 24 cores each. Thus, resulting in 48 CPU cores per node. Listing 5.1 is extracted from generated code. In order to simplify the code some details are omitted. In line 1 the variable `NUM_CORES` is defined. It is used to define the number of cores for each individual MPI rank. Lines 3 - 7 initialize the MPI environment. When the MPI environment is initialized the value of `NUM_CORES` is set for each node (see lines 8 - 12). Since, two identical nodes are used the values do not differ. When utilizing nodes with different architectures, the thread pools can have different sizes. In line 13 a vector with `NUM_CORES` elements is created. The thread pool becomes the utilized thread pool when the `setPool` function is used. This function takes a reference to the thread pool and indicates that the memory for the thread pool is allocated. The `startExecution` call handles the initialization and thread pinning. Once this function is called the thread pool can be utilized for parallel execution similar to `MPI_Init`. The individual tasks are defined as lambda functions capturing the original input and output (see line 18 -

Listing 5.1: Example usage of the thread pool.

```
1  int NUM_CORES;
2
3  int rank, nprocs;
4  MPI_Status Stat;
5  MPI_Init(&argc, &argv);
6  MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
7  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
8  if ( rank == 0 ) {
9    NUM_CORES = 48;
10 } else if ( rank == 1 ) {
11   NUM_CORES = 48;
12 }
13 std::vector<Thread> pool(NUM_CORES);
14 setPool(&pool);
15 startExecution();
16 // variable initialization
17 if ( rank == 0) {
18   auto f_0 = [initial, &result] () {
19     for (int INDEX = 0; INDEX < 0 + 8; ++INDEX) {
20       result[(INDEX)] = initial[(INDEX)] + 1;
21     }
22   };
23   getPool()->at(0).addWork(f_0);
24   // partial operations for threads 1 to 23
25 }
```

22). The lambda function computes a subset of the workload which is computed during the generation. The workload for the task in lines 18-22 handles the index range from zero to seven, as specified in the loop in line 19. The task is added to the task queue of thread zero in line 23. The lambda functions defining the tasks for threads one through 23 were omitted. The last task for every processor handles all iteration that could not be assigned to the previous threads. Thus, thread 23 might have a slightly larger workload compared to the other threads. This load imbalance is necessary in order to ensure the correctness of the computation.
In some cases, it is not possible to perfectly balance the work for a given number of threads. For example, a pattern with three iterations cannot be perfectly partitioned for two threads. The result of the last computation is still necessary for the correctness of the computation and the number of threads cannot be changed. Therefore, one thread has to handle a larger workload than the other. For the current implementation it was decided, that the last participating thread has to handle this increased workload. Thus, thread one handles the first iteration and thread two

Listing 5.2: Example usage of the thread pool for synchronization.

```
1 if ( rank == 0) {
2   Bit_Mask * mask_ptr = new Bit_Mask(48,true);
3   for (int i = 0; i < 24; ++i) {
4     mask_ptr->setBarrier(i);
5   }
6   boost::shared_ptr<Bit_Mask>boost_mask_ptr (mask_ptr);
7   self_barrier(boost_mask_ptr);
8 }
9 finishExecution();
```

handles the second and third iterations. This implementation can still be optimized. The remaining workload, in some scenarios, with more than two threads can still be assigned more equally.

Listing 5.2 defines the use of the bit mask for synchronization. In line 2 the bit mask object is created. The first argument defines the total number of threads within the thread pool. The second argument defines whether all threads are used for synchronization initially or none. `true` means none are used initially, while `false` indicates that all threads should be synchronized. The code in Listing 5.2 is executed directly after the example in Listing 5.1. Thus, the barrier is set for the first 24 threads as seen in the loop in lines 3 - 5. In order to preserve the pointer to the barrier the object pointer is wrapped in a shared pointer provided by the boost library [**?**]. Boost is widely known and accepted C++ library. The library supports header-only implementations for many of their applications, such as the shared pointers. The header-only libraries adopt the functions used during compilation minimizing the induced overhead during computation.

The `self_barrier` function in line 7 then synchronizes the threads defined in the shared pointer with the main thread. `barrier` would only synchronize the threads with each other. The function `finishExecution` in line 9 destroys the thread pool as explained before.

5.4 Distributed-Memory Parallelism

5.4.1 Programming Model

The communication between nodes is generated utilizing MPI [**?**] in the generated code. MPI is a message passing interface well known within the HPC community. The MPI standard supports data transfer between nodes with different communication schemes,e.g., point-to-point and collective operations like a reduction. Especially, the support for collective operations like the before mentioned reduction can be utilized to efficiently implement data transfers and parallel patterns.

There are other programming models for distributed-memory programming such as

GASPI [**?**] which could be considered, but do not provide the same level of support MPI does. Therefore, MPI was chosen for the implementation of the code generator for this thesis.

5.4.2 Implementation

The generation of parallel code across multiple nodes is realized with MPI in this thesis. The programming model introduced by MPI is based on the direct communication between nodes via messages. In this thesis mostly the point-to-point communication is utilized to transfer the data between nodes. Although, the structure to analyse the communication behaviour and make use of collective communication is provided. The use of collective communication patterns is out of scope for this work.
During the generation of the AMT the data transfer nodes are grouped by data elements and all necessary data as well as the current placement of data is provided. The generation of data transfers can be adopted to enable faster communication patterns. Therefore, data transfer nodes need to be more specialized in order to realize more complex communication patterns.
To correctly generate code for multiple nodes two problems needs to be solved:

1. Handling of sequential execution.

2. Handling of the memory management.

Sequential code should, per definition, be executed exactly once. Thus only a single node should compute sequential results. To implement a strategy solving the first challenge a single node must be chosen. There are two applicable strategies when choosing a node for the sequential computation:

- **Global**: One possibility is to select a node from the beginning, e.g., sequential code must always execute on rank zero. By analysing the amount of potential data transfers for each node, the choice could be optimized in order to minimize the amount of data transfers.

- **Local**: The other possibility is to select the optimal node for each serial block individually, optimizing the data transfers locally. Therefore, it would be possible to change the node computing sequential code every time. The implementation of this strategy is dependant on the optimization results and is much more complex to implement compared to a global selection.

The implementation of the code generator currently utilizes a fixed global selection for simplicity. Sequential computations are all executed on the first node (rank zero). The performance benefits from improving the global selection or utilizing a local selection. This can be analysed in the future.

Secondly, the memory model of generated distributed-memory parallelism needs to be defined.
Three general strategies for the allocation of memory are possible:

- **Complete allocation**: The memory for all data elements is fully allocated on every node. The data transfers only need to move the most recently changed chunks of data between the nodes.
- **Dependency complete allocation**: Only the memory for data elements which are used on the respective node are allocated completely. The data can be transferred similar to the complete allocation strategy.
- **Partial allocation**: Only the memory currently used on the node is allocated. When the data is changed on a another node, it has to be deallocated. The data transfers need to handle the allocation and deallocation of the transferred data in order to minimize the memory overhead. Further, the generated code needs to be adapted in order to handle the offsets induced by this strategy, e.g., for a map which computes half of its workload on a different node, memory is allocated for half of its in- and outputs. Since the data accesses for the second half also starts from zero, all accesses to this data need to be adapted.

The partial allocation strategy would need to be tested, since the allocation and deallocation will result in a runtime overhead for both the generated code and the compiler. Potentially, a partitioned global address space (PGAS) model can be utilized in the future. A PGAS model would allow direct memory access on remote memory. Approaches are already implemented by GASPI [**?**] and could be utilized to generate distributed memory parallelism. MPI does also provide support for direct memory accesses. By specifically optimizing data transfers and utilizing existing features could further improve the performance of distributed memory programming. One example that already utilizes collective operations provided by MPI is the parallel reduction pattern. MPI supports the reduction of arrays over multiple nodes for a predefined set of combiner functions: Maximum, minimum, summation and multiplication. The generator utilizes the collective `MPI_Reduce` function in order to combine the temporary results of partial reductions into a single result. Similar optimizations could be performed in the future, when splitting data for computation on multiple nodes, e.g., the `MPI_Scatter` and `MPI_Gather` functions could be used. Due to the limited time available for this thesis the complete allocation strategy is implemented. The dependency complete allocation would be a reasonable approach for the next steps.

5.5 GPU Offloading

5.5.1 Programming Model

The restrictions for GPU programming models are met by most of the well-known GPU programming models, since multi GPU programming is an important feature

for many applications. Thus, the more important aspect of the chosen GPU programming model is the potential performance. The following three programming models are the most suitable for the generated code:

- **OpenMP** [**?**] does also support GPU programming. The additional compiler directives allow the choice of a specific GPU with the device clause. The behaviour of the GPU computation is handled by additional compiler directives. The compiler directives further include the definition of data transfers. Generating CPU and GPU code with OpenMP could greatly simplify code generation, since two types of parallelism can be handled by a single programming model.
- **OpenACC** [**?**] similar to OpenMP parallelizes code with compiler directives. The compiler directives provided by OpenACC provide the same capabilities as OpenMP. There are frameworks which transform OpenACC and OpenMP code into one another, such as CCAMP by Lambert et al. [**?**]. OpenACC does not support CPU parallelism.
- **CUDA** [**?**] in contrast to OpenMP and OpenACC, directly utilizes the driver callbacks for the GPU. Since CUDA is designed by NVIDIA, the level of freedom and in turn the potential performance is high. Additionally, CUDA supports many runtime libraries which can be reused. When generating CUDA code the generator has to handle data mapping and transfers automatically. A simple CUDA generation can be used as a baseline and be further improved in the future.

Using CUDA, especially in combination with PThreads, it is possible to create threads specifically tasked with handling the GPU offloading asynchronously. Since the AMT defines synchronization points for individual execution units asynchronous offloading does not cause data races. For the generation of GPU code, CUDA was chosen, due to its performance potential.

5.5.2 Implementation

For the generation of GPU code it was decided to utilize CUDA as discussed above. The direct combination of CUDA code and MPI code is not fully possible, due to the existing compilers. There are CUDA-aware implementations of MPI, these support the CUDA runtime during compilation. But the CUDA kernel generation is not supported by the CUDA-aware MPI. Thus, at least a kernel wrapper needs to be generated in an additional file. This file can be compiled by the NVIDIA CUDA compiler and linked with the compiled main code.
Since, the goal of this thesis is to generate code compatible with many architectures and compilers a CUDA-aware implementation of MPI should not be assumed. Thus, the complete CUDA runtime must be compiled in a separate file, which must be generated. In order to avoid additional overhead by wrapping every single callback, two functions are defined and generated.

Listing 5.3: A GPU Kernel defining a map which increments each element.

```
1  __global__
2  void kernel_cuda_wrapper_increment(int32_t* initial,
       int32_t* result ) {
3    int tid = blockIdx.x * blockDim.x + threadIdx.x;
4    int exec_range = 0;
5    if (tid < 100) {
6      exec_range++;
7    }
8    for ( int range_iterator = 0; range_iterator <
         exec_range + 0; range_iterator++) {
9      int INDEX = tid * exec_range + range_iterator +
           100;
10     if (tid < 100) {
11       INDEX -= 100;
12     }
13     result[(INDEX)] = initial[(INDEX)] + 1;
14   }
15 }
```

- **GPU Wrapper**: This function defines the wrapper which is called by the main file. The function handles the memory allocation on the GPU as well as the data transfers between CPU and GPU. Further, the GPU kernel is executed in this function.
- **GPU Kernel**: The GPU kernel implements the logic defined by the application programmer. In order to ensure the correctness of the call additional control structures are generated.

The GPU kernel in Listing 5.3 and the GPU wrapper in Listing 5.4 implement a map which increments the value of `initial` and store it in `result`. Both listings contain generated code which removed randomly generated suffices to increase readability. These suffices are utilized to differentiate variables and function calls that are similar. This helps by avoiding unintentional multi-specifications during the code generation.
To simplify the generation of the wrapper and the kernel they are executed exactly once with the corresponding input and output. The current example defines the second half of a map with 200 elements. The last argument of the GPU wrapper call (Listing 5.4) always defines the output. At first the device pointers are declared (see lines 2 and 3). In line 4 the device pointers are used to allocate memory on the GPU. Since, this wrapper computes half of the 200 element map, 100 elements are allocated. If the offset between multiple input elements or the output element is different, more memory is allocated to maintain the correct proportions of the

Listing 5.4: A GPU wrapper defining the data transfers and allocations for the kernel in Listing 5.3.

```
1  void cuda_wrapper_increment(int32_t* initial, int32_t*
       result) {
2    int32_t* d_initial;
3    int32_t* d_result;
4    cudaMalloc(&d_initial, sizeof(int32_t) * 100);
5    cudaMemcpy(&d_initial[0], &initial[100], sizeof(
       int32_t) * 100, cudaMemcpyHostToDevice);
6    cudaMalloc(&d_result, sizeof(int32_t) * 100);
7    kernel_cuda_wrapper_increment<<<1, 168>>> (d_initial,
        d_result );
8    cudaMemcpy(&result[100], &d_result[0], sizeof(int32_t
       ) * 100, cudaMemcpyDeviceToHost);
9    cudaFree(d_result);
10   cudaFree(d_initial);
11 }
```

parallel pattern. The data transfer in line 5 copies the second 100 elements of the input to the device. If the proportions need to be handled, the start value of the device pointer is increased. Thus, the halo data is only allocated but not transferred. Since the wrapper only handles the second half of the map the data transfer starts at the 101st element of the input. The allocation and transfers are generated for each input parameter. The allocation of output memory on the device in line 6 is handled in the same way as the allocation of input variables. For reductions the output is replaced with a temporary array which stores the result of each block and is reduced in an additional step to a single value on the GPU.
Line 7 defines the kernel call with CUDA. In this case there were no kernels fused thus only one block is used. Further, each block utilizes 168 threads. The threads are composed of the 64 FP32 cores, the 32 FP64 cores, 64 INT32 cores and 8 Tensor Cores [**?**]. To execute a task with 100 iterations on the GPU is not applicable. Especially, when the number of threads is higher than the potential parallelism. For example, checking the correctness of the generated code is applicable. In line 8 the result computed on the GPU is transferred back to the CPU, since only the second half is computed the data transfer starts at index 100 of the CPU array. The data transfer and the kernel execution both execute on the default CUDA stream, thus they are synchronized implicitly and explicit synchronization is not necessary. In the end (lines 9 and 10) the memory on the GPU is deallocated to avoid memory leaks on the GPU.
The GPU kernel in Listing 5.3 defines the computation on the GPU. As mentioned before, in order to ensure the correctness of the kernel regardless of the number of threads, additional branching is introduced. In line 3 the id of the current thread is

computed. This is necessary to assign an index range for each thread. Line 4 defines the size of the range per thread. Since the number of threads (168) is larger than the number of iterations (100) `exec_range` becomes 0. The first 100 threads have their workload increased by one iteration in lines 5 - 7 because not all iterations are covered by the `exec_range`.
The first threads are chosen in order to optimize the branching behaviour of a warp. A **warp** is a group of threads. For NVIDIA GPUs the warps generally have size 32. During the execution, all threads within a warp compute the exact same steps which are chosen by the first thread within the warp. Thus, for branches used only by a subset of a warp, the performance is worse, since some threads need to completely redo the computation.
Thus, only a single warp contains diverging threads minimizing the potential performance loss. In lines 8 and 9 the `INDEX` variable is defined according to the index range of each thread. Potential offsets are handled by the loop header in line 8. The value of `exec_range` can be different for different threads. Hence, the complete offset must be handled for the threads with a smaller index range. In the example in Listing 5.3 100 threads get an additional increased range leading to an offset of 100 for the threads with a smaller range. This holds since $r * m + m = r * n \forall r \in \mathbb{N}$ holds for $\forall n, m \in \mathbb{N}$ with $n = m + 1$. This shows, that for an arbitrary amount of threads r can have their workload increased by one and the correct ranges can still be computed. To ensure an optimal branching behaviour the offset added in line 9 must be subtracted for all threads with the increased range (see lines 10 - 12). In line 13 the logic for each iteration is generated.
For the generation of reduction patterns the recombination is generated according to the examples provided by NVIDIA [**?**]. However the examples do not cover newer architectures like the NVIDIA Volta GPUs [**?**]. Volta GPUs have a program counter for each thread unlike older architectures with a program counter per warp. Thus, the threads within a single warp can diverge leading to data races during the warp wide reduction. Therefore, additional synchronization steps within the warp were introduced in order to avoid data races.
The utilization of the wrapper function forces the executing thread to remain in the function until the corresponding kernel and data transfers are completed. Thus, the execution on the main thread is not desirable. The management of asynchronous behaviour in the limited scope of the wrapper is highly complex and introduces a substantial overhead in terms of time and memory usage. The solution to this issue, implemented in this thesis, makes use of the thread pool implementation discussed in Section 5.3.2.
By introducing a second smaller thread pool, the GPU wrapper can be parallelized. The GPU pool is handled exactly as the normal thread pool. To explicitly handle the GPU pool the known functions are extended. The thread pinning is not necessary for the GPU pool but it could improve the performance if the core is physically close to the GPU. The current implementation utilizes one thread per GPU.
By utilizing the existing thread pool GPU code only needs to call the wrapper in a lambda function and assign the lambda function to a GPU. Further, the data

transfers between GPU and CPU are already covered by the wrapper. Synchronizing a GPU is realized by generating a barrier task for the GPU similar to the CPU parallelization.

5.6 Data Structures

Data structures are an essential part of parallel programming. Currently the APT only supports arrays as data structure. Thus, it is sufficient to support simple concatenated data structures. In the context of C++ as the target language, the following data structures are the most suitable to be used alongside the chosen programming models:

- **C Arrays** are supported by all programming models. Though they do not support arbitrary nesting to create multi-dimensional arrays. Thus, the dimensionality of an array must always be known for accesses, needing an inline implementation by hand. Further, the memory management needs to be considered, adding further complexity.
- **C++ Standard Vectors** are a data structure defined by the C++ standard. They can be nested into each other arbitrarily and the memory management is handled automatically by an allocator. The standard vectors are supported neither by CUDA nor by MPI. Even if it is possible to directly access the underlying C array, for nested vectors the underlying array must be accessed arbitrarily often to access the values. Thus, for CUDA and MPI operations the data would need to be transformed every time.
- **Thrust vectors** are a data structure introduced by the thrust library for CUDA [**?**]. The thrust vectors cannot be nested, but multi-dimensional data structures can be emulated in the same way as C arrays. While thrust vectors would allow an integration of the thrust library and corresponding performance improvements, it is incompatible with MPI. Thus, needing to transform the data structure for every data transfer.

The transformation of data structures depending on the parallel context is costly during the runtime. Furthermore, a context dependant generation of data is highly complex and error prone. The inlining process is slightly complicated, but it does not strongly impact the performance of the generated code. A drawback of the inlining is the continuous generation of local variables in the root node, increasing the memory usage of the application. The transformation of data structures are assumed to create a greater overhead in either runtime or memory. Therefore, the data structure chosen, in the context of C++, for the implementation in this thesis were C arrays.
The memory management for different nodes is solved by utilizing the structure of the AMT. Both the variable scope of the current node and the parent are known. Thus, before generating any child nodes the variables can be initialized. Further,

Figure 5.1: Nested order of the File Generator, the AMT to Source generator and the Expression Printer.

upon reaching the last child or a return statement, the memory allocated by the local variables can be freed.

5.7 Code Generator Tool

The generator is designed to only depend on the AMT. As a result of this dependency, the generator can be implemented by utilizing a visitor pattern for the AMT, as discussed in Section 4.1.1. The use of the visitor simplifies the addition of new nodes in the AMT, since only the respective visitor functions for the new node needs to be defined.
The code generator designed in this thesis can be separated into three different parts:

1. **File generation:** The creation of output files utilizing FreeMarker [**?**] templates and the MontiCore [**?**] template engine.
   - FreeMarker is a template engine used for the creation of template based text files with changing data.
   - The MontiCore template engine utilizes FreeMarker and builds upon it with their generator functions.
2. **AMT to source generation:** The traversal of the AMT and the transformation of AMT nodes to a string.
3. **Expression generation**: The generation of expressions, since they do not follow the tree structure of the AMT.

Figure 5.1 depicts the hierarchy in which the File Generator calls the AMT to Source Generator and the Expression Printer.

5.7.1 File Generation

As mentioned before, the generation of files utilizes the template engine provided by the MontiCore language workbench [**?**]. With the template engine a set of templates for different output files can be generated with FreeMarker templates. By defining a generator helper, contents based on the AMT generation can be added dynamically to the defined templates. This allows a static and dynamic definition of different output files,e.g., runtime libraries, program entry points and Makefiles. The generator and the generator helper utilize a given AMT to fill the templates. In order to simplify the generation of the output, multiple files are generated. The generated files include a program entry point, multiple libraries, header files and utility files

such as a Makefile and a machine-file as described in Section 5.8. The dynamically generated contents of these files are all gathered in the generator helper and retrieved via callback in the template definitions. In order to improve the runtime of the generator, the generator helper executes the generation of the AMT exactly once during the initialization and stores all the necessary information. Another advantage of the generator helper is that it can be reused for the definition of each template.

5.7.2 AMT Generation

The AMT generator provides a 4-tupel of strings. The individual elements are defined as follows and define a part of the dynamically generated contents of the templates:

1. The **Program entry point** is the first element of this 4-tupel. It defines the string generated from the root node and contains the parallel execution of all patterns targeting shared and distributed memory. This element does not define sub-AMTs which did not need to be inlined. The information on these function templates is provided by the AMT generator statically.

2. **GPU sources** define the second element of the 4-tupel. As explained in Section 5.5 the generated code targeting the GPU must be defined in a different file. The contents of this string contain the definition of both the GPU wrapper and kernel.

3. The third element defines the **GPU header**. This string defines the function headers for the GPU wrapper functions. The generation of such a header file is necessary in order to define the existence of the GPU wrapper, while the GPU code is compiled by a different compiler from the program entry point.

4. The **GPU kernel header** defines the GPU kernel and it is used to simplify the generation of the GPU sources. With the target language (C++) functions cannot be used before they are defined. In order to utilize functions defined afterwards the header file can be used to declare the function.

The generator utilizes an AMT visitor for the implementation of the generator. The visitor always starts from the root node. When encountering a call node which does not need to be inlined, the visitor traverses the corresponding sub-AMT and stores the generated code in a hash map with the function identifier as a key. Thus, it can be checked whether the sub-AMT has already been generated and redundant traversals can be skipped.
In order to reduce the overhead of generated parallel patterns, the input and output elements are connected to the parameters of the function node. This connection is defined as a variable inlining table which indicates how data elements are replaced within a scope, e.g., Listing 5.5 defines a map pattern in lines 2 - 4 which copies the contents of `input` and stores them in `res`. The pattern is executed in line 8 with `initial[1]` and `result` as its input and output arguments. The call copies the

Listing 5.5: A map pattern which copies the input values into the output.

```
1  Pattern_Inlining{
2  map copy([Int] input):[Int] res {
3      res[INDEX] = input[INDEX]
4  }
5  seq main():Int {
6  var [[Int]] initial = init_List([2,200],3)
7  var [Int] result = init_List([200],0)
8  result = copy<<<>>>(initial[1])
9  return 0
10 }}
```

values of the second line of the matrix `initial` into the array `result`. The data elements defined in the sub-AMT during the generation of the pattern need to be replaced. The replacement is indicated by an entry in the data replacement table. In the example in Listing 5.5 the table contains two entries. Each entry has the original data element as its key and a replacing expression as its value. The first entry defines `input` as the key and the expression `initial[1]` as its value. The second, defines `res` as the key and `result` as its value.
The complete AMT is depicted in Figure 5.2 including the synchronization step necessary after the parallel call. The generated code is depicted in Listing 5.6. Initialization, finalization and multiple thread specific task definitions are omitted for readability. Inlining the `initial[1]` expression in Listing 5.5 results in the expression `initial[200 * (1)+ (INDEX)]` in Listing 5.6 line 10. This allows for a reduction of memory and runtime overhead for the generated code, since local variables are not necessary to replace parallel call arguments.

The expression printer can then replace the data elements of the pattern template with the corresponding values. The structure of the data replacement table allows the nesting and replacement of arbitrarily many parallel patterns.

5.7.3 Expression Printer

The structure defined for expressions is different from the tree structure defined for the AMT. The expressions are originally defined in the APT. Changes to the APT are out of scope for this work. Therefore, either the definition of a new expression structure for the AMT needs to be implemented and generated or a generator which prints the expressions directly into the defined output structure.
The expressions are integrated into the AMT with expression nodes which can hold one or more expressions. The expressions are structured into two different classes:

- **Operation expression**: The operation expression defines an expression which computes a value, like `a + b * c` or `a < b`. The data structure is based on two lists:

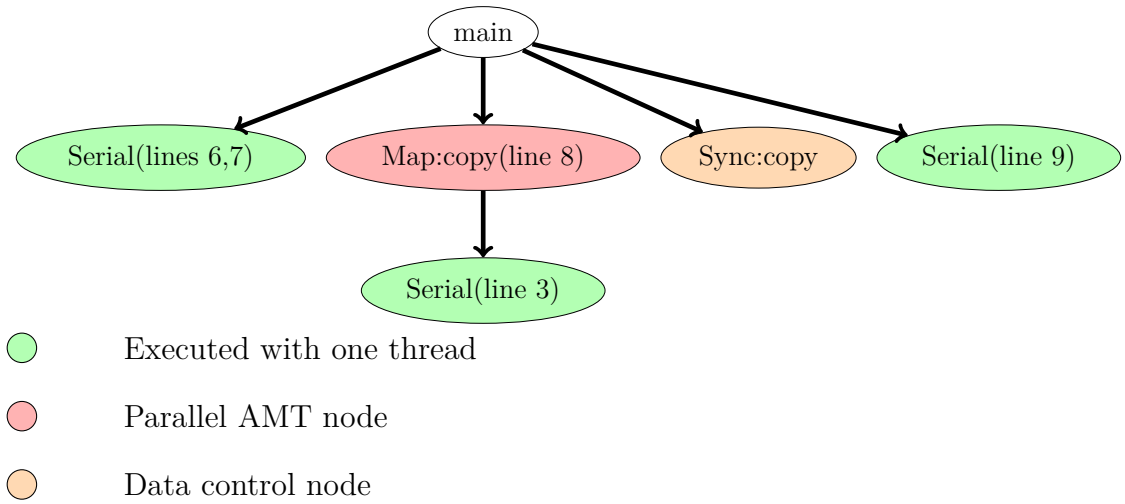


Figure 5.2: AMT based on Listing 5.5.

  - **Operands**: A list of data elements in the order they are used in the expression.
  - **Operators**: A list of operations in the order they are performed. Parenthesis, arithmetic operation like plus and boolean operations are completely included in this list.

- **Assignment expression**: An assignment expression defines an assignment of a data element based on an operation expression. Further, the assignment specifies an access scheme which defines via a set of operation expressions how a (multi-dimensional) array is accessed.

To construct a well-formed C++ expression, based on operation expressions, the arity and the necessity if a left context is defined for all operators, e.g., the plus operator has an arity of two and needs a left element. With this information a simplified algorithm was designed to implement the expression printer for operation expression (see Listing 5.7). The right context can be computed by iterating the operator list and keeping track of the number of operators with an arity of two, which need a left element. It defines the index of the next operator in the operator list. The algorithm iterates over all operands and tests if an operator follows the current operand. When there are none, all remaining operators are printed before the operand (see line 4 - 6), otherwise only the operators between the current and next operator (see line 7 - 10) are printed. Afterwards, the current operand is printed in line 11. When all operands are handled and operators remain, the remaining operators are printed at the end.
For example the expression `(a + b)* c` is defined as the operand list `[a,b,c]` and the operator list `[(,+,),*]`. The C++ expression is generated as follows:

1. `a` is the first operand with `+` as its right context.

Listing 5.6: Generated code based on the AMT in Figure 5.2.

```
int main(int argc, char** argv) {
  int32_t* result;
  int32_t* initial;
  //init
  initial = Init_List(3, initial, 2LL * 200LL * 1LL);
  result = Init_List(0, result, 200LL * 1LL);
  if ( rank == 0) {// Map:copy
    auto f_0 = [initial, &result] () {
      for (int INDEX = 0; INDEX < 0 + 8; ++INDEX) {
        result[(INDEX)] =  initial[200 * (1) + (INDEX)
           ];
      }
    };
    getPool()->at(0).addWork(f_0);
  }// work for threads 1 to 23
  if ( rank == 0) {//sync:copy
    Bit_Mask * mask_ptr = new Bit_Mask(48,true);
    for (int i = 0; i < 24; ++i) {
      mask_ptr->setBarrier(i);
    }
    boost::shared_ptr<Bit_Mask>boost_mask_ptr (mask_ptr
       );
    self_barrier(boost_mask_ptr);
  }
  // finalize
  return 0;
}
```

2. The operator `(` is printed since it precedes `+`.
3. The current operand is printed resulting in `(a` as the current result
4. The next operand is `b` with right context `)`
5. `+` is printed, since it precedes `)`
6. The current operand is printed resulting in `(a + b`
7. The next operand is `c` with no right context.
8. Since no right context is present the remaining operators `)` and `*` are printed.
9. The last operand is printed resulting in `(a + b)* c` as the output.

The actual implementation of the algorithm in Listing 5.7 is much more complex than the example provided in Listing 5.7. The additional complexity is based on the context of array access parenthesis, where the left context dependency changes whether it is the first array access or not.

5.8 Utility Files

The code generator also provides a set of utility files which are not part of the executed sources. The files specify the behaviour of the sources. First, the generated Makefile defines the tool chain of different compilers necessary to transform the generated source files into a single executable. The second utility file defines the machine file or host file for MPI. The machine file defines the order in which the machines are used by MPI, e.g., the first address takes rank zero, the second address rank one etc. This ensures that the order of the nodes in the HL is the same as the order used during the computation.

5.8.1 Makefile

The Makefile defines the compiler and compiler flags for each source file. With `make` all main source files are compiled into executables, with their respective dependencies. Thus, the thread pool and GPU pool implementations can be compiled without compiler optimizations. The combination of C++ classes with PThreads

Listing 5.7: A simplified algorithm which creates a well formed C++ expression based on an operation expression.

```
1  current operator = 0
2  for int i = 0; i < operandList.size() ; i++
3    int nextOperator = getRightContext(i);
4    if nextOperator == -1
5      for int j = currentOperator; j < OperatorList.size
           (); j++
6        generateOperator(j)
7    if nextOperator > currentOperator
8      for int j = currentOperator; j < nextOperator; j++
9        generateOperator(j)
10     currentOperator = nextOperator
11   generateOperand(i)
12 if currentOperator < OperatorList.size() - 1
13   for int j = currentOperator; j < OperatorList.size();
         j++
14     generateOperator(j)
```

is interpreted as constant callback by the GNU and Intel compiler. During the optimization step of those libraries, the queue dependant functions are replaced with constants by the compiler. Additionally, the program entry point and the GPU code are compiled with the MPI and NVIDIA CUDA compiler respectively. In order to create the executable, all files are compiled into object files. Afterwards all dependant object files are compiled into an executable binary file. The Makefile is defined statically.

5.8.2 Machine File

The machine file defines the order of the machines used during the computation. The definition of a consistent order during optimization and computation is important. When all nodes are identical, but the distance between nodes is different, the performance cannot be modelled correctly resulting in a loss of performance. Another problem occurs when the nodes are equipped differently. The computation optimized for a node with GPU devices might crash on a node without GPUs. Therefore, the machine file is necessary to ensure the correctness of distributed-memory programming. The machine file is passed to the MPI execution handler which ensures the order of the defined nodes. To generate the machine file the HL is traversed and the address defined by all nodes is returned in the order of their definition.

6 Evaluation

The quality of the generated code is important when evaluating a code generator. In this thesis parallel code is generated. An important factor for quality evaluation of the generator is the performance of the generated code. The correctness of the generated code defines another quality assessment discussed in this work. Other indicators for the quality of a generator and the generated code like maintainability and reusability is not easily quantifiable and is as such omitted from the evaluation in this thesis.
The correctness is evaluated by a set of test cases defined for the generator. The verification suite and the corresponding test results are discussed in Section 6.1. The results of the test suite can be measured by counting the failed and successful test cases.
For evaluating the performance of the generated code, the test cases introduced in the work of Trümper [**?**] are utilized. The test cases define algorithms that are optimized using the optimizer implemented in their work. The algorithms were originally defined in the PPL to get the results from the optimizer.
Since the code generator was not implemented during the implementation of the optimization package, the optimizations are carried out by hand. To evaluate the performance of the generated code the runtime of the handwritten and generated code are compared. The performance evaluation is discussed in detail in Section 6.2.
All test cases are executed on two nodes of the CLAIX-2018-GPU cluster [**?**]. The nodes are equipped with two Intel Xeon Platinum 8160 and two NVIDIA Volta 100 GPUs. They are connected via Intel Omnipath. The generated code is compiled with gcc 9.3.0, openMPI 4.0.3 and the Nvidia CUDA compiler for CUDA 11.0.

6.1 Code Generator Verification

The correctness of the generated code is important. Formal verification is a complex and time consuming task especially when both the generator and the generated code need to be verified. Furthermore, when verifying the generated code an additional test suite is necessary in order to at least verify the correctness of some representative code examples. Due to time limits relevant for this master thesis a formal verification is not possible. To still have the opportunity evaluating the correctness of the generated source, the error classification according to Schmitz et al. [**?**] will be extended to cover syntactical errors. In addition, a test suite was created in this thesis based on the extended error classification. The test suite comprises 85 individual test cases designed covering all error classes. The test cases are written in the

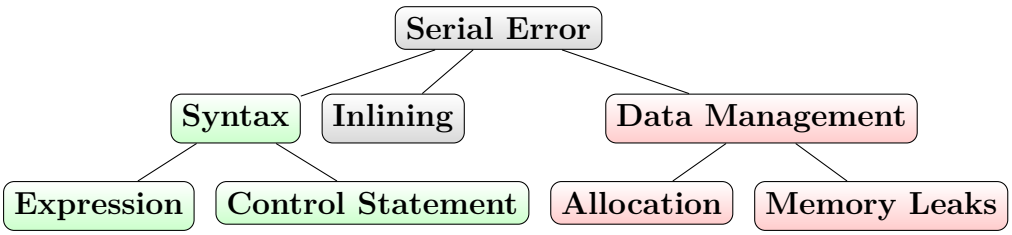


Figure 6.1: Serial error classification.

PPL and are transformed to an AMT with the "Debug Mode" creator introduced in Section 4.2.2 to enforce the generation of specific scenarios.

6.1.1 Error Classification

The error classes are separated into two distinct sub-classes. The serial errors which may occur at any point in the source and the parallel errors which will only be induced when parallel computation is involved.

Serial Error Classes

The serial error classes are mostly related to the syntactic correctness and the memory allocation strategies of the generated code. The inlining of function calls with array arguments is also covered by serial correctness. In Figure 6.1 the serial error classification is depicted. The individual classes and corresponding test cases will be elaborated in the following:

- **Syntax**: The syntax is an essential part of the generated code. While it is not necessary that the generated code is readable, it must be correctly interpreted by a following production compiler. Thus, it is significant that the generated code meets the syntactic standards defined by the target language. The test cases following this class must compile with a production compiler and represent the semantic defined in the front-end language. The following two classes further explain the syntax of certain constructs:
  - **Expression**: The expression class represents the correct generation of expressions utilizing the expression printer. The expression printer transforms expressions into character strings in the target language. Therefore, test cases implementing this class cover multiple combinations of different expressions.
  - **Control Statement**: The class of control statements defines the correct generation of control statements utilizing the AMT generator. This class covers errors regarding for-loops, while-loops and branches.
- **Inlining**: Due to the choice of data structures, as stated in Section 5.6 certain serial function calls need to be inlined during the generation.

To ensure the correct generation of inlined functions two test cases are designed. The first, accepts an array as its input. The second, does not accept an array. Due to the unknown size of pointer based arrays in C/C++ the first function must be inlined to retain correctness. This is especially important for multi-dimensional arrays, because of the nesting strategy implemented, which skips dimensions using a polynomial access pattern. For the second function this is not necessary, since data accesses do not depend on the size of a scalar value. Therefore, it is checked whether the first test case is properly inlined while the second is not.

- **Data Management**: This class covers the correctness of allocation and deallocation in a serial context. To further specify this class the following error cases are defined:
  - **Allocation**: Errors with this classification do not allocate the memory specified by the user. Depending on the definition of a variable in the front-end, the corresponding data element must be allocated on the heap or the stack. Therefore, initializations need to be handled properly.
  - **Memory leak**: This class of errors relates to the correct deallocation of memory space. Thus, it is checked whether for each scope all local memory on the heap is freed at the end, e.g., when allocating an array within a loop it must be deallocated at the end of the loop iteration.

Parallel Error Classes

The classification of parallel correctness errors depicted in Figure 6.2 is based on the classification presented by Schmitz et al. [**?**]. To utilize the classification in the context of the code generated in this thesis differences between devices are disregarded. Further, a node defining the correctness of performance was added. The error classes are defined as follows:

- The **On Device** error class defines concurrency defects on a single device, e.g., a data race.
  - **Concurrency Defect**: A concurrency defect is an error which is caused by a concurrent execution of source code. The failure (Error manifestation) does not necessarily occur always, e.g., a data race can potentially provide the right result even if the race conditions are met.

  The differentiation between inter and intra region is especially apparent for GPU code [**?**]. The definition of processors in the HL as subsets of cores within a CPU supports the application of a differentiation. Inter region concurrency defects on the CPU are induced by the hardware mapping provided by the optimization. Intra region concurrency defects are induced during the source code generation. Thus, the source of an error is dependent on different packages and a differentiation is applicable.

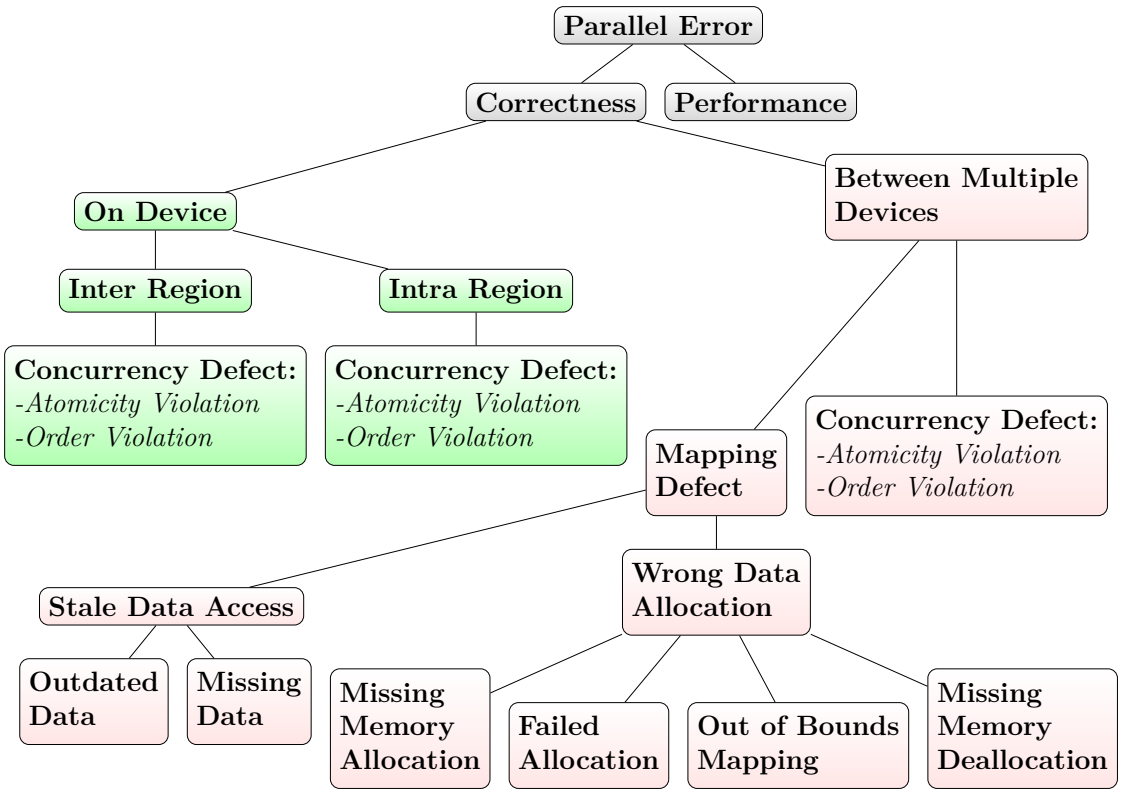


Figure 6.2: Parallel error classification based on the work of Schmitz et al. [**?**]

- Errors **Between Multiple Devices** define defects which occur when utilizing multiple devices. Concurrency defects are only possible if the devices support direct memory access or have a shared memory space, e.g., partitioned global address space (PGAS) models can induce such concurrency defects. Mapping defects on the other hand are related to the data transfers between devices:
  - **Stale Data Accesses** depict errors where current data is missing. The two subcategories are defined as follows:
    - **Outdated Data** errors are cases where the data was changed on another device, but not updated on the current device.
    - **Missing Data** errors define defects related to data never has been copied to the device.
  - **Wrong Data Allocation** defects are categorized by a faulty allocation of memory on a communicating device. The four following classes defined by Schmitz et al. [**?**] further specify different defects:
    - **Missing Memory Allocation** errors occur when memory is not allocated before transferring data.
    - A **Failed Allocation** defines cases referring to memory allocation fails which had not been handled properly before use.

    - **Out of Bounds Mapping** errors define defects induced insufficient memory allocations for the transferred data.
    - **Missing Memory Deallocation** classify defects related to allocated memory that is not freed at the end of an execution. Hence, creating a memory leak.

- **Performance** errors appear if the optimizations are generated correctly. Thus, if the optimization package wants to execute on the GPU, then the code generator should generate GPU code.

6.1.2 Test Suite Implementation

The test suite contains 85 test cases implemented in the PPL. The test cases cover all error classes in both the serial and parallel correctness classification. The test cases utilize the debug mode introduced in Section 4.2.2 to avoid dependencies on the optimization package. Further, it is assumed that the APT is generated correctly. The test suite utilizes a Makefile generated by the code generator to be compiled and executed. The code generation supports the generation of parallel CPU, GPU and MPI code. Thus, the target system must contain at least two nodes with two GPUs each.
All test cases should be transformed into compilable and executable C++ code. Since the automatic evaluation of serial test cases is dependant on the same serial constructs, the serial test cases need to be evaluated by hand. The test cases covering parallel correctness errors are designed to be evaluated by the result of the computation. This is achieved by defining both a sequential and a parallel execution of the same algorithm. When both implementations are finished the results of both runs are compared, if the results are not identical the test case fails and an error message is printed. The parallel test cases assume the correctness of the generated sequential code. Hence, errors in both a sequential and a parallel test case may not imply an error during the generation of parallel code. Therefore, the test suite should always be evaluated completely in order to avoid false implications.

Listing 6.1 introduces a test case used in the test suite, which will be discussed for further clarification. The test case implements a map pattern which increments `input` and stores the result in `res` (see lines 2-4). The pattern is executed in line 10 with `initial` as input and `result` as its output. `result_seq` is used to store the result of the sequential execution. All three arrays are initialized in lines 6-8 and can store 200 elements each. `initial` takes `1` as a starting value for each element. As indicated by the additional argument `[0,0,0]` the pattern call in line 10 executes on the first processor of the first device of the first node. The defined device is a CPU according to the HL used by the test suite. Lines 11-16 define the sequential error check. In line 12 the sequential result is computed. If the branch condition in line 13 evaluates to `true`, the sequential and parallel results are not equal. Thus, the predefined `print` function prints an error message. Test cases like the one in Listing 6.1 which are dependant on the execution of a specific pattern on a specific

Listing 6.1: A test case implementing a map pattern covering intra region concurrency errors on the CPU.

```
1  MapTest{
2  map increment([Int] input): [Int] res{
3      res[INDEX] = input[INDEX] + 1
4  }
5  seq main() : Int {
6      var [Int] initial = init_List([200],1)
7      var [Int] result = init_List([200])
8      var [Int] result_seq = init_List([200])
9      /*execute the pattern on node 0 device 0 */
10     result = increment<<<[0,0,0]>>>(initial)
11     for var Int i = 0; i < 200; i++ {
12         result_seq[i] = initial[i] + 1
13         if result_seq[i] != result[i] {
14             print("value at element " {i} " is wrong!")
15         }
16     }
17 return 0
18 }}
```

device are implemented for each covered parallel pattern (see Section 3.3.2) and target device architecture. Specific MPI test are not necessary, since errors can as well be covered by the test cases utilizing multiple devices on different nodes. Further, MPI is mostly utilized for transferring data between nodes.

6.1.3 Test Cases

Table 6.1 depicts the coverage of the test suite. The serial tests include 14 of the 85 test cases. The eight test cases covering the syntax errors combine multiple scenarios in one. The expression tests combine related expressions into the same test case, e.g., the arithmetic expressions are all cover in a single test case. The test cases covering data management errors cannot be further differentiated into the allocation and memory leak errors, because the data management is handled implicitly by the AMT and code generator. Therefore, test cases covering data management error include both sub-classes, e.g., a defect in the code generator can induce allocation error, memory leaks or both. The utilization of mapping defects in the parallel context is similar.
The implicit data management during the AMT and code generator stages of the program compilation causes these types of errors depending on the correctness of the compilation steps. Thus, the same test cases comprise all sub-classes with the

Serial			
Syntax		Inlining	Data Management
Expression	Control Statement	2	4
4	4		
Parallel			
On Device			Between Multiple Devices
Inter Region	Intra Region	Mapping Defect	Concurrency Defect
10	20	16	25

Table 6.1: The test case coverage of the test suite.

same test case. The mapping defect test cases implement examples of two map patterns, with data dependencies between architectures, e.g., the first map performs on the CPU, while the second map pattern performs on the GPU with the output of the first map. These test cases contain multiple different interaction patterns on different device combinations.

Concurrency defects between multiple devices make up 25 of the 85 test cases. To cover different scenarios the test cases implement each of the four parallel patterns and recursion on multiple combinations of two or more devices on a set of two nodes with two GPUs each. Therefore, combinations like a Stencil on two GPUs on two different nodes is included. The combinations covered are:

- Two accelerators on the same node
- Two accelerators on different nodes
- The CPU and an accelerator on the same node
- Two CPUs on different nodes
- Two CPUs and two accelerators on different nodes

Each combination implements every parallel pattern supported and recursion. The implementation of inter region concurrency defects utilizes two different processors on the same device. The ten test cases are split into five implementations for the CPU and five for an accelerator. The five individual test cases implement the four parallel patterns and recursion.

Similarly for the intra region test cases, half of the 20 tests target an accelerator and half the CPU.

Furthermore, for each pattern including the recursion a test case is defined implementing the nesting of the pattern within a parallel map pattern.

6.1.4 Results

The implementation of the code generator provided by this thesis successfully completes all 85 test cases. While the test suite aims to cover as many errors cases as

possible, the test suite may not include all potential error classes. Further, implementations of the test suite might need to extend the classification and test cases when extending the generator or implementing a new generator. For example, errors on FPGAs can be different from errors on CPU or GPU, similar to the *on Device* error class in the work of Schmitz et al. [**?**]. As such, the test suite is used to prove the correctness of a minimal capability of the code generator and should be extended upon in the future.
Predefined functions are not specifically covered in the test suite and are assumed to be tested before hand. The integration of the predefined functions, is tested as part of the syntactical expression test cases which are part of the serial correctness classification. Additionally, the test cases should be extended to cover errors induced by the optimization and various transformations until generation. The correctness evaluation of other components is out of scope for this work, but the test suite and error classification can be utilized as a basis for a group of automatic tests checking the correctness of the complete project.

6.2 Performance of Generated Code

The performance of the generated code is an important criteria for evaluating parallel code. One part of the performance assessment is the evaluation of the thread pool implementation. The thread pool was developed in this thesis, as such no performance and overhead evaluations exist for the thread pool implementation. The second part of the performance evaluation covers the runtime differences between the benchmarks designed by Trümper [**?**] and the generated code. The test cases consist of different algorithms implemented as an unoptimized baseline in C, an optimized C implementation and a PPL implementation of the algorithm. The PPL implementation is optimized and generated into the C++ code as discussed in Chapter 5.

6.2.1 Thread Pool Overhead Measurements

The thread pool implementation introduced in Section 5.3 for shared-memory programming utilizes additional data structures and functionalities. These are added in order to satisfy the requirements for the generation of shared-memory parallelism. Although, the introduction of such a library introduces overhead in terms of runtime and memory. The focus of this section will be on the runtime overhead. The memory overhead is static with a dependency on the size of the ring buffers and the number of threads. To measure the runtime overhead induced by the thread pool, the EPCC benchmark by Bull et al. [**?**] is utilized.
The syncbench micro benchmarks introduced in EPCC were adapted in order to measure the overhead of the thread pool. The syncbench micro benchmarks measure the work sharing and synchronization overhead for OpenMP. The results of the adapted EPCC and the original EPCC for OpenMP are compared. Therefore, a

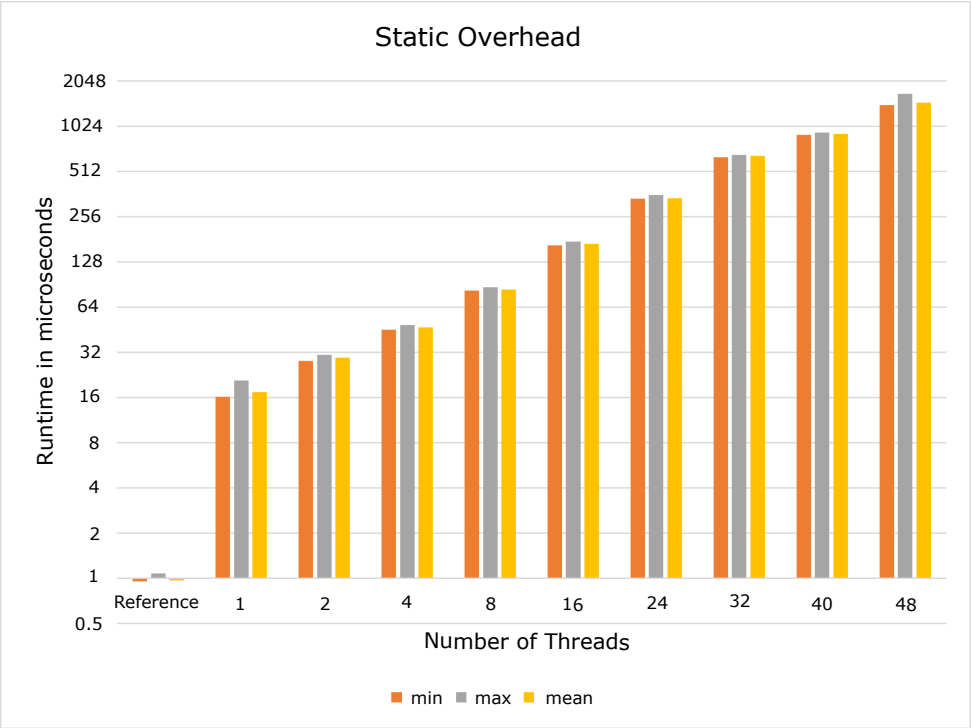


Figure 6.3: Static overhead of the thread pool with different thread sizes on two Intel Xeon Platinum 8160.

conclusion on the implementation in comparison to a well established programming model can be drawn. The EPCC benchmark is implemented in C and has portability issues with C++.
Further, the high contention of test tasks can induce deadlocks for the barrier and the initialization of the thread pool, since it is not designed to handle arbitrarily small tasks. To still be able to generate results comparable to the EPCC implementation the idea of the executor was implemented to run a fixed set of outer iteration. The outer loop defines how many results are evaluated. The time is measured for each outer loop iteration. The number of inner loop iterations is constant in contrast to the EPCC implementation to avoid an overflow of the task queues. The task performed in the inner loop is defined by a delay function adapted from EPCC benchmark and the operation from which the overhead is measured. The total runtime of the inner loop is divided by the number of inner loop iterations to avoid inaccuracies induced by the time measurements. For each function the minimum, maximum and average runtime are calculated. Both implementation were executed with a delay of one microsecond and 64 outer iteration. The size if the inner loop for the original EPCC is computed on demand, the version for the thread pool utilizes 128 iterations for the inner loop.

In contrast to the OpenMP implementation, where the threads are available implicitly, the thread pool needs to be initialized and destroyed explicitly. This induces measurable overhead described in Figure 6.3. The runtime is shown in a log scale

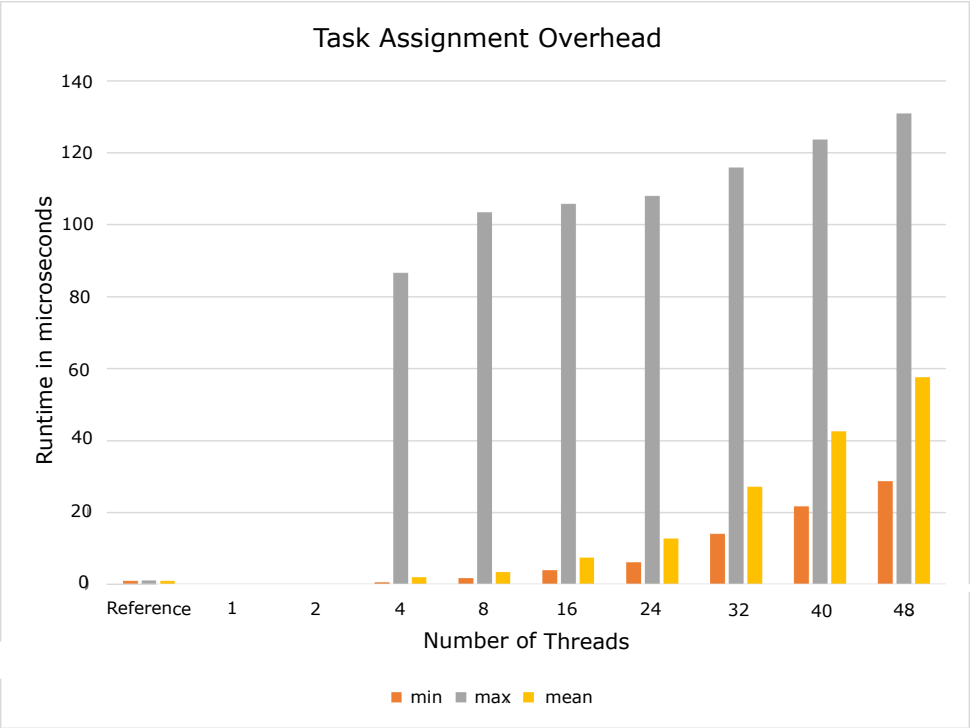


Figure 6.4: Overhead induced by the assignment of work to the task queue in the thread pool on two Intel Xeon Platinum 8160.

with base two. The total overhead induced by the initialization increases exponentially based on the number of threads used.
The exponential development of the overhead is dominated by the migration of threads using OS callbacks, since the initialization of data structures can induce a linearly growing overhead at most. Especially, the context switch necessary to correctly map the thread id to the core becomes more expensive with increasing distance between the current and new core. Since the amount of threads for shared-memory programming is limited and the static overhead happens exactly once per application, the generated overhead is manageable with 1.6 milliseconds of overhead for 48 threads. The reference time of 1 microsecond is 1600 times smaller than the generated overhead.
The main thread of the thread pool implementation does not take part in shared-memory parallelization. In contrast to OpenMP, where the main thread also handles a part of the workload. Therefore, the barrier overhead measured defines the complete overhead, while the OpenMP defines the overhead as a difference between the reference time and the measured time.
Another source for runtime overhead is the assignment of work to the corresponding threads. Although, the OpenMP `task` constructs are closer in behaviour and runtime overhead to the implementation of the thread pool, they are compared to the `parallel for` construct. The map, stencil and reduction pattern describe mostly regular access patterns which do not benefit from dynamic scheduling to improve

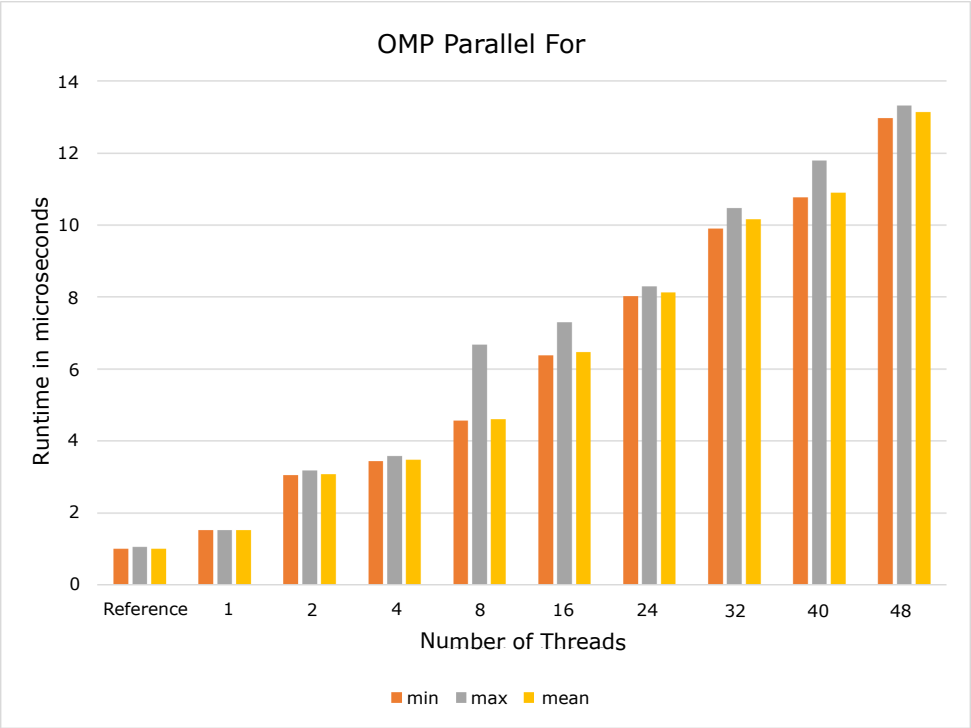


Figure 6.5: Overhead induced by the assignment of work with the parallel for construct in OpenMP on two Intel Xeon Platinum 8160.

the load balance. Thus, their implementation in handwritten code would rather utilize the `parallel for` construct than the `task` constructs. The runtime of the `parallel for` construct is depicted in Figure 6.5. The reference delay is one microsecond like in all other measurements as well. The time difference between the reference and the runtime with n threads defines the overhead. The overhead becomes maximum 12 microseconds when utilizing 48 threads. The measurements for the thread pool implementation (see. Figure 6.4) are much more unstable leading to more outliers and a larger difference between the maximum overhead and the average overhead. These outliers are most likely caused by the critical section used to limit the access to the task queue. The additional threads can start their work once the task is added to the task queue. Therefore, it is possible that the overall duration of the assignment is smaller than the reference time, e.g., this is the case for computations with one or two threads.
The overhead induced by the thread pool increases much faster than the OpenMP overhead which is also most likely due to the critical section enclosing the task queue accesses. The overhead when utilizing 24 threads is roughly the same in both OpenMP and the thread pool. For more than 24 threads used, the overhead becomes much larger in the thread pool where the overhead induced with 48 threads is almost five times the overhead induced by OpenMP. Since the two Intel Xeon Platinum 8160 used for this measurement are modelled as two processors in the HL, there are maximum 24 threads used per task assignment. Thus, the two program-

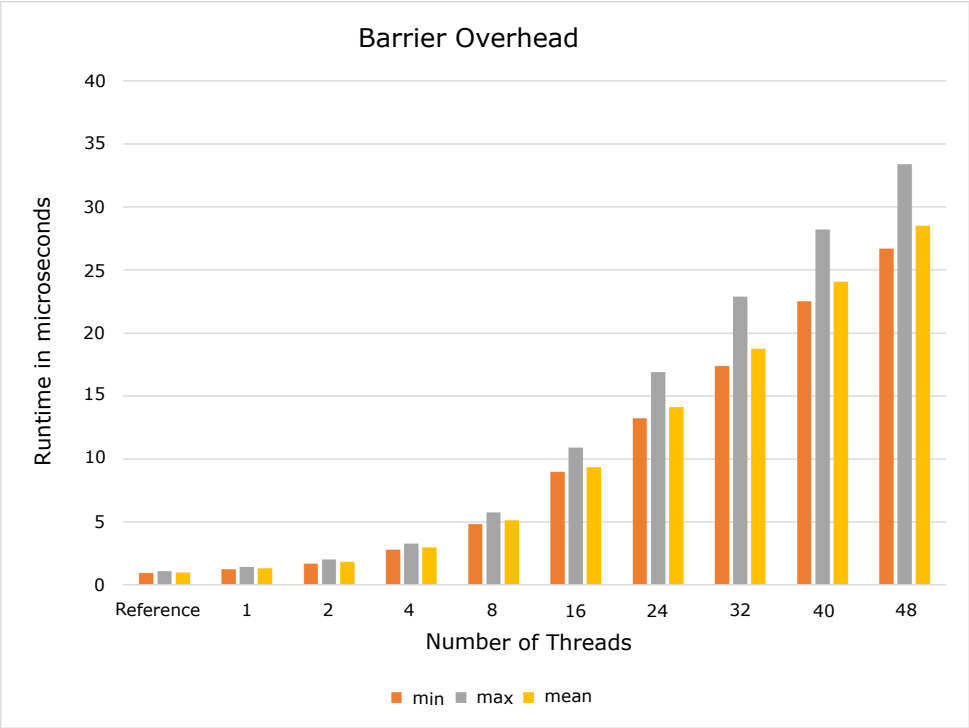


Figure 6.6: Overhead induced by a barrier in the thread pool on two Intel Xeon Platinum 8160.

ming models are comparable for the task assignment. The volatility of the thread pool overhead is a problem to be solved in the future. To approach this task detailed profiling results are necessary.

As introduced in Section 5.3, the implementation of a barrier in the thread pool implementation is realized by defining a task for each node. The task contains the pthread barrier used to synchronize the individual threads. The graph in Figure 6.6 depicts the scaling behaviour for the runtime overhead induced by a barrier over the given amount of threads. The development of the overhead is similar to the task assignment, but slightly higher with about one or two microseconds difference. Further, the runtime of the barrier is much more stable in comparison to the assignment. Since every barrier is also a task, the barrier induces the overhead of both the assignment and the actual synchronization of threads. The overhead of the assignment is almost the same as the overhead from the synchronization. Thus, the assignment of the barrier task is most notably the dominating factor. The barrier tested does not synchronize the main thread with the parallel threads, only the creation and assignment of the barrier is measured.
The impact on the computation is measured in Figure 6.7. With a reference time of one microsecond the self barrier creates an overhead of roughly eleven milliseconds. This overhead does not increase for a higher number of threads which is the expected behaviour seen in Figure 6.6 and Figure 6.8. Therefore, it is assumed, that this constant overhead is caused by the logic surrounding the actual synchro-

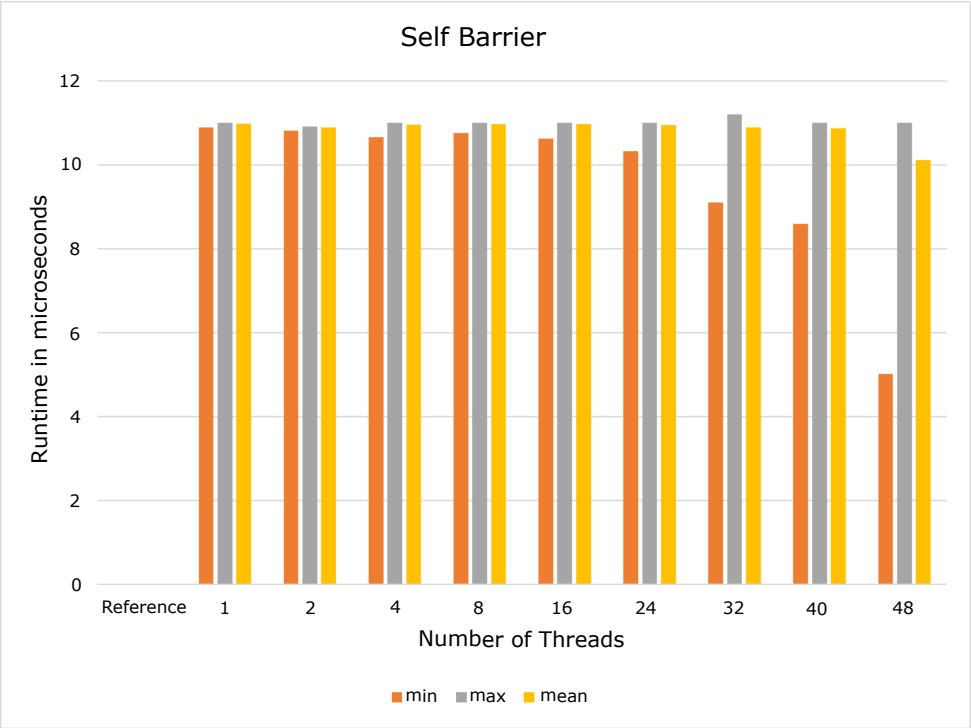


Figure 6.7: Overhead induced by a barrier synchronizing all threads in the thread pool with the main thread on two Intel Xeon Platinum 8160.

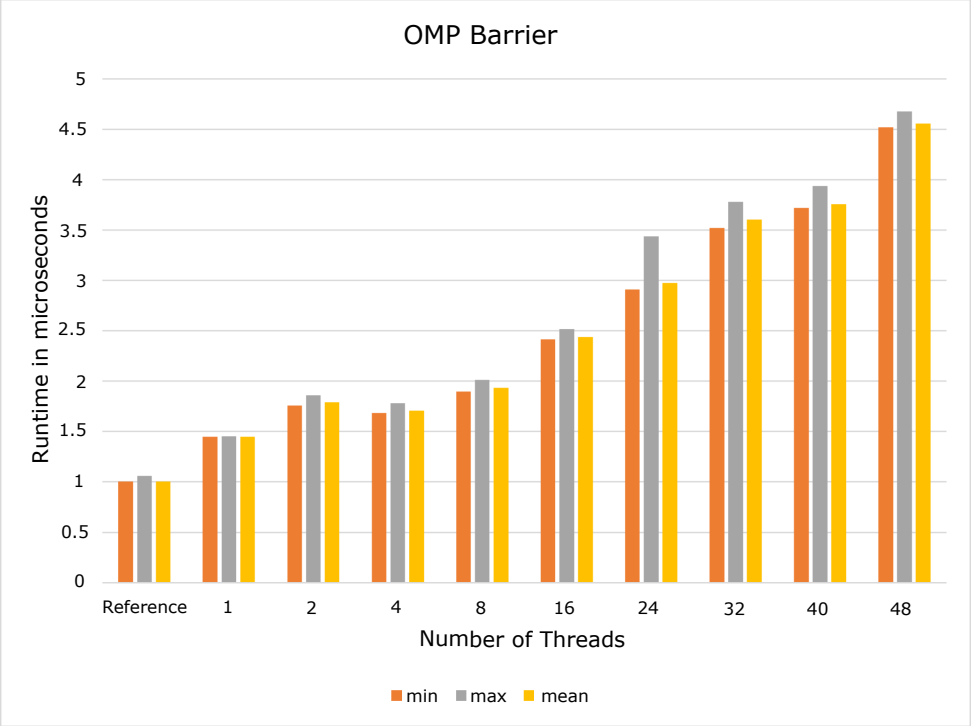


Figure 6.8: Overhead induced by a barrier in OpenMP on two Intel Xeon Platinum 8160.

nization, performed by PThreads. To properly detect the cause for the overhead, a hotspot analysis is necessary, e.g., with the Intel VTune perfomance analyzer [**?**] such a hotspot analysis can be performed. Based on the results, the overhead can be improved, e.g., if the overhead is caused by the boost pointers, global data structures can be used to ensure the availability of the bit mask. Removing the bit mask data structure is another approach.
Nevertheless, barriers in the thread pool including the main thread take at least 2000 times more time than the OpenMP implementation. The cost for synchronization without the main threads cannot be measured accurately due to deadlocks induced by repeated initialization and destruction of the thread pool.
The OpenMP barrier overhead depicted in Figure 6.8 is generally smaller than the overhead induced by the thread pool. The cost of a barrier increases logarithmic. The barrier implemented in OpenMP does not require a task assignment resulting in better scaling behaviour, than the thread pool implementation. The runtime overhead measured for OpenMP barriers is always smaller than the overhead measured thread pool barriers. The overhead induced by the thread pool is twice as high as the overhead induced by OpenMP for a single thread. The difference increases for 48 threads to about 8.5 times the overhead induced by OpenMP. When including the main thread into the synchronization becomes 2000 times more costly.

6.2.2 Algorithmic Case Study

To evaluate the performance of the generated code in comparison to handwritten code, a small case study covering five different algorithms is evaluated. The algorithms were utilized and implemented in the work of Trümper [**?**]. The evaluation of the algorithms is based on the average duration of the kernel of the application for 50 repetition. The baseline implementation is based on a naive parallelization of the algorithm written by hand. The optimized C code is also written by hand and optimized in accordance to the optimization package. The baseline and optimized C code were both implemented and defined by Trümper [**?**]. The generated code is based on an implementation of the algorithms in the PPL by Trümper. The PPL implementation was defined to generate the optimizations utilized in the optimized C code. The generated code is based on the same PPL and HL code used for optimizing the handwritten C code. The speed up is defined as: $\frac{\text{Baseline runtime}}{\text{Optimized runtime}}$. Therefore, the speed up is a positive number and a speed up between *0* and *1* defines an increase in runtime.

Batch Classification Test

The batch classification task defines a three stage classification algorithm over a $2^{19} \times 4096$ input matrix. The goal for this test is to evaluate the loop fusion and pipeline optimizations.

1. Normalize the 2^{19} input elements

Batch class	Baseline	Optimized C	Generated	Generated_loop
Mean Runtime	1.1703s	0.8823s	1.8690s	1.7984s
Standard Dev.	0.0540	0.0703	0.0216	0.0353
Speed up	1	1.3264	0.6262	0.6507

Table 6.2: Representation of the batch classification performance for generated and handwritten code. (Runtime in seconds.)

2. Extract the feature of each element
3. Classify each element based on a majority vote from an ensemble of 4096 classifiers.

The goal defined in the work of Trümper [**?**] is to generate a pipeline optimization on the CPU, since the workload is to small for utilizing a GPU. The pipeline optimization for the three steps is an optimal solution according to the optimization, because each iteration only depends on a single iteration of the preceding step. The handwritten optimization is compiled with gcc 9.3.0 with the flags *-fopenmp -std=c99 -O2*. For this test case two versions of the generated code are evaluated. The first one utilizes nested parallel patterns to define internal loops. The other implementation utilizes for-loops and is extended by the *_loop* suffix.
The results for the batch classification algorithm can be observed in Table 6.2. The baseline implementation takes about *1.1703* seconds to compute the evaluation of all 2^{19} input elements. The loop fusing performed by the optimization yields a speed up of *1.3264* in the handwritten code C code reducing the runtime to about *0.8823* seconds. The runtime of the generated code yields a speed up of *0.6262* indicating that the generated codes takes longer to compute with *1.8690* seconds. This increase in runtime is induced by the higher overhead of the thread pool implementation and the missing implementation of the fused nodes. The complexity of the reduction pattern used to implement the majority vote in the third step in most definitely inducing a large overhead. The overhead induced by the reduction is composed of the following elements:

- The task assignment for the local results
- A barrier to ensure that all local results are computed
- A reduction over all local results as a task creating a processor local result
- A critical section to compute the global results based on all processor local results

Since the overhead of each of these components is already much higher in the thread pool implementation than with OpenMP, an increase in runtime is induced. The missing implementation of the fused nodes during generation further increases the runtime, because the optimal optimization strategy cannot be generated yet. The

runtime of the loop based implementation of algorithm performs slightly better. The estimated cost for loops is more accurate than the estimated cost for nested parallel patterns. Thus, the loop based version can perform the optimizations much more accurate reducing the runtime of the generated code.

Jacobi Test

Jacobi	**Baseline**	**Optimized C**	**Generated**	**Generated_old**
Mean Runtime	1.3197s	0.5883s	4.6822s	4.4027s
Standard Dev.	0.0599	0.0743	0.1840	0.2025
Speed up	1	2.2432	0.2829	0.2998

Table 6.3: Representation of the Jacobi solver performance for generated and handwritten code. (Runtime in seconds.)

The Jacobi test case implements a linear equation solver utilizing the Jacobi method. The test case solves three linear equation system on the shared matrix A. The number of Jacobi iterations is fixed to 50 steps with 8192 unknown variables. The optimization generated for this algorithm computes the solution for the three equation systems in parallel by reordering the individual Jacobi iterations from the three equation systems. The handwritten code is compiled with gcc 9.3.0 with the flags *-fopenmp -std=c99 -O2*. For this algorithm two implementations are generated. The first, replacing inner loops with parallel patterns. The second, denoted by the suffix *_old* defines the generated version using inner loops.
The results for the jacobi solver can be observed in Table 6.3. The baseline application kernel computes all three linear equations in sequence. The optimization deems a computation of all three equation systems in parallel more efficient. This reordering results in a speed up of *2.2432*, reaching a runtime of *0.5883* seconds for the computation. Neither of the generated versions can improve the performance of the baseline implementation. Most notably, the inner loop based implementation is nearly *0.3* seconds faster than the PPL implementation utilizing nested patterns. This difference is induced by the difference in accuracy when analysing the cost for loops and patterns, the cost for patterns in the optimization is significantly less than the estimated loop cost. The actual cost should be equivalent.
The increase in runtime is dependant on two factors:

- **Overhead**: The first reason for the elongated runtime of the generated code in comparison to the handwritten is the higher overhead induced by the thread pool implementation. The overhead induced by the assignments and the barrier should be at most $3 * 50 * (90 + 10) = 15000$ microseconds for 50 iterations of three Jacobi iteration and a maximum difference in overhead around 90 microseconds for the assignment and 10 microseconds for the barrier definition. These numbers were taken for the utilization of 24 threads per iteration per

equation system. When considering the self barrier used for the synchronization, with eleven milliseconds of overhead and 50 iterations the synchronization creates an overhead of *0.55* seconds.

- **Horizontal fusing**: The optimization defines the execution of the generated code to be divided among 2 processors on a single device. The handwritten code with OpenMP allows an explicit loop fusion for each step which is currently not supported by the code generator. This further increases the amount of necessary synchronization steps threefold.
  Since the optimization can only assign a task to a processor and not a thread a lot of potential parallelism is lost. This can also increase the load imbalance between the different processors, one processor has effectively twice the workload. This problem can be solved by increasing the granularity of the HL definition. By specifying more processor on the same amount of threads the compilation time increases, but the quality of the generated code should increase as well. Measurements regarding this behaviour are independent from the code generation and are therefore out of scope for this work.

Monte Carlo Test

Monte Carlo	**Baseline**	**Optimized C**	**Generated**
Mean Runtime	43.3359s	21.9670s	16.5826s
Standard Deviation	1.4012	0.7351	0.9719
Speed up	1	1.9728	2.6133

Table 6.4: Representation of the π approximation performance with monte carlo for generated and handwritten code. (Runtime in seconds.)

The Monte Carlo test implements the estimation of π using a Monte Carlo approximation. The value is computed based on the average of 96 estimations with 10^9 random draws each. The code is optimized to perform on two nodes with 48 threads each. The handwritten code is compiled with OpenMPI 3.1.3 and gcc 9.3.0 with the flags *-fopenmp -std=c99 -O2*. With the handwritten code only MPI is utilized for parallelization.
The result for the Monte Carlo algorithm are depicted in Table 6.4. The baseline computation on a single node is optimized to utilize a second node which reduces the original runtime of *43.3359* seconds to *21.9670* seconds achieving a speed up of *1.9728*. Due to additional overhead an ideal speed up of *2* is impossible only by doubling the number of node/threads. The generated code achieves a speed up of *2.6133* with a runtime of *16.5826* seconds. The generated code utilizes a hybrid programming model which performs shared-memory computations with the thread pool and distributed memory computations with MPI. This reduces the amount of overhead induced during the shared-memory computation.

Further improvements like thread affinity and memory locality are not considered in the handwritten C code.

Multi Filter Convolution Test

Multi Filter	Baseline	Optimized C	Generated
Mean Runtime	0.0371s	0.0336s	0.1212s
Standard Deviation	0.0016	0.0048	0.0178
Speed up	1	1.1050	0.3063

Table 6.5: Representation of the multi filter convolution performance for generated and handwritten code. (Runtime in seconds.)

The multi filter convolution implements image processing with three different filters on a 8194×8194 image. During the first step a Sobel filter operates on the upper half of the image and a Prewitt filter on the lower half. Additionally, a Laplace filter is performed on the whole image. The optimized code performs the Sobel filter and the first half of the Laplace filter on the same processor and the Prewitt, the second half of the Laplace filter on another processor. This optimizes the memory accesses for the different filter operation. The handwritten optimization is compiled with gcc 9.3.0 with the flags *-fopenmp -std=c99 -O2*.
Table 6.5 shows the results of the multi filter convolution. The baseline takes about *0.0371* seconds to compute the three individual filters. The handwritten optimization improves this runtime with a speed up of *1.105* to *0.0336* seconds, by improving the memory accesses. The generated code takes about three times as long as the optimized handwritten code. This increase is also implicated by a speed up of *0.3063* and a runtime of *0.1212* seconds for the generated code. The increase in runtime is caused by a combination of the higher overhead induced by the thread pool and an implicit imbalance caused by the assignment of the tasks to the task queue of the individual threads. The assignment of each task is handled individually for each thread during runtime. Thus, threads with a higher id get their work assigned after all tasks to preceding threads are assigned.

Neural Network Test

Neural Network	Baseline	Optimized C	Generated
Mean Runtime	0.3431s	0.1739s	18.5476s
Standard Deviation	0.0458	0.0010	0.9883
Speed up	1	1.9728	0.0185

Table 6.6: Representation of the feed forward neural network performance for generated and handwritten code. (Runtime in seconds.)

The neural network implementation defines a forward pass in an eight layer neural network. The individual layers of the neural network are composed of 64 neurons. The algorithm computes a batch with 2^{18} elements on the GPU with a block size of 512, as defined by the optimization. The handwritten code is compiled with the Nvidia CUDA compiler for CUDA 10.2 and the gcc 8.2.0 with the flags *-fopenmp -std=c99*.
The results of the measurements are depicted in Table 6.6. The baseline implementation is implemented in C and parallelized on the CPU using OpenMP. This version achieves a runtime of *0.3431* seconds. The optimization deems the algorithm large enough for GPU offloading. Thus, the optimized code computed on a GPU and achieves a speed up of *1.9728* with a runtime of *0.1739* seconds. The generated optimization takes *18.5476* seconds to compute the results of the forward pass, achieving a speed up of *0.0185*. This increase in runtime is caused by three issues:

- **No fusing**: The optimization does not define a fused node for the eight layers. Thus, after each layer the GPU is synchronized with the CPU which is not necessary. The handwritten code does not synchronize the kernel explicitly.
- **Wrapper definition**: The wrapper function used to implement GPU offloading is defined to always transfer the input data to the GPU and the output data to the CPU. Since the output from the previous layer is to be reused in the current layer, most of the data transfers generated are redundant.
- **Overhead**: Due to the additional control logic implemented a small runtime overhead is induced. The amount can be neglected in comparison to the cost of redundant data movement.

7 Discussion

The generation of parallel code is a complex task. The generation of efficient parallel patterns for different architectures increases the complexity of the generated code even more. As such, there are many challenges to be solved when implementing the generation of optimized code. The optimization defines where and how the abstraction is to be computed. Furthermore, the results are all computed statically and the generation has to define an application based on limited data. To properly replicate the semantics of the abstract optimization certain design choices are necessary. Based on the design choices and the corresponding implementation the following five major challenges are identified:

- **Correctness**: This thesis is the implementation of the back-end for code generation. The correctness of the generated output is more important than the performance itself, since this is the first implementation of this back-end. This implementation can be used as a baseline for further optimizations and extensions of the generator.

- **Performance**: The runtime of the application is an essential property of HPC applications. The scalability is mostly covered by the optimization package and can, therefore, be disregarded partially in this work. In order to improve the runtime and scalability of the generated code the defined optimization should be implemented and the runtime overhead needs to be minimized.

- **Reusability**: The outcomes of this thesis contribute to the last remaining packages of the prototype implementation of the overarching project. Thus, the resulting implementation should be (partially-) reusable when extending the prototype or implementing a release version of the project.

- **Integration**: As part of a larger project, the results in this thesis have to properly work within the whole project, by integrating existing packages like optimizing and providing interfaces for future implementations of the generator and optimization packages.

- **Synchronization and Data Transfers**: The optimization packages are enforced to define synchronization and data transfer behaviour. The availability of the nodes is still necessary for the generation of parallel code, targeting shared-memory parallelism, distributed-memory parallelism and GPU offloading. Furthermore, the integration of barriers and data transfers impacts the runtime and scalability of the generated code.

7.1 Correctness

The generated code should compute the results intended by the programmer. If the correctness of the code is not sufficiently reached non-functional requirements like performance have a reduced significance. The test suite discussed in Section 6.1 defines the minimum requirements in terms of correctness. To proof the correctness of the generator and the AMT a formal verification of the tool would yield the safest results. This verification can be similar in structure to the verification of existing compilers [**?**, **?**]. Further, the verification of parallel code will still be necessary in order to avoid conceptional errors. The work by Zheng el al. [**?**] provides a baseline to verify the correctness of the generated code. The combination of the two above mentioned approaches very particularly increases the complexity of this task.
A test suite is comparably much easier to define than the verification. Especially, if targeting only a part of the system. The test suite introduced in Section 6.1 targets only the code generator. To properly cover the correctness of the code generation the test suite should be extended towards covering the correctness of AMT, the AMT transformation as well as the correctness of the optimization and the APT representation. Since the project is designed to be modular with the AMT and APT as fixed interfaces, a test suite covering the individual modules will be a suitable approach for the future. A test suite independent from the front-end package would be beneficial as well. This can be realized by defining an interface to directly define an APT with an integrated language. In such a language, the nodes of the APT could be specified and transformed into the actual APT specification.

7.2 Performance

A key aspect of this thesis is the correctness of the generated code. Although, the performance is still an important aspect. As discussed in Section 6.2 the performance can be improved significantly in future implementations.
The implementation of the thread pool is inefficient in comparison to programming models like OpenMP. Although, the generation of hybrid MPI + PThreads code does perform better than pure MPI applications, as seen in Section 6.2.2.
By defining a lock free implementation of the task queues the generated overhead can be reduced in the future. By further defining task queues for the entire processor instead of individual threads, dynamic scheduling improvements with less overhead are realized. The work stealing concept of such an implementation could handle performance differences not covered by the performance model defined in the optimization.
The generation of fused nodes in order to generate pipelines is not yet supported. This is especially obvious in Table 6.6 illustrating additional data transfers and synchronization steps between GPU kernels that increased the runtime drastically. The generation of the fused nodes becomes a complex problem utilizing a basic visitor, since it does not provide sufficient context information. A possibility to achieve the

generation of fused nodes would be the implementation and utilization of a parent aware visitor. This type of visitor would provide knowledge on the parent node of the current parallel node. Thus, a context dependant implementation of the fused nodes can be implemented by generating the fused tasks for shared-memory parallelism (see Section 5.3.2) into the same lambda function. For the GPU implementation (see Section 5.5.2) the individual kernel wrapper can be fused, in a way that the kernel calls happen in succession without data movement and synchronization. These optimizations would greatly impact the performance of the generated code.
The implementation of GPU offloading can also be improved significantly in the future. Currently, the GPU wrapper handles allocation, data movement and kernel execution in a single call. By better analysing the required and changed data during the generation, redundant data transfers can be avoided. By splitting the individual allocations and data transfers from the kernel call a more diverse generation of GPU code would be possible. To avoid losing the information on data pointers when splitting the wrapper, a global data structure utilizing hash maps can be implemented. The data structure would dynamically store the device pointer with a given hash which depends on the parallel pattern call. Thus, the allocation and data transfers could be defined statically with a dynamic data structure during runtime.
The benefits of implicit synchronization during the generation of barrier and data transfer nodes is still disregarded. Thus, as long as two nodes access the same data and the first node has a parallel write access to the data, a barrier is generated. An exception to this are fused nodes, where no synchronization is happening within. To reduce the number of barriers while maintaining the correctness of the code, data accesses regarding the barrier generation can be analysed more accurately. When the two concatenated data accesses follow the same access pattern on the same processor, the tasks for each thread are assigned to the same threads. Therefore, the data necessary for the computation is already on the thread and synchronization is not necessary, since the a task within the thread queue cannot be computed prior to its predecessor.

7.3 Reusability

The results of this thesis are directly used within the prototype implementation of the overarching project. The implementation of the generator module is closely related to the choice of the target language. Thus, for other target languages it is necessary to develop a new code generator. Extensions to the AMT can be directly integrated into the generator implementation, since the utilization of the visitor pattern, as explained in Section 4.1.1 allows the addition of new nodes as well as the extension of existing nodes.
The AMT is designed to provide an interface for the optimization output and the generator input. Therefore, the data structure can and should be reused in combination with other packages. The implementation of the optimization has to make sure that the interface defined by the AMT is met.

The implementation of the AMT only covers the optimizations implemented by Trümper [**?**], since it is the only optimization package available at the time. In order to define a different optimization, the AMT will need to be extended. These extensions can be integrated directly, if they exclusively add new nodes or parameters. Structural changes can also be integrated and adopted by the visitor implementation.
Essential changes to the interface induce essential changes in the optimization and generator packages. Thus, the optimization and generator need to be reworked to ensure a correct functionality.

7.4 Integration

The integration of the results of this thesis into the overarching project depend on the implementation of the middle-ware. The AMT is integrated into the existing code by implementing the two transformations discussed in Section 4.2. The instantiation of serial AMT nodes is realized by extracting the necessary information from the corresponding APT nodes. This direct dependency does simplify the implementation of both transformations. Although, an extraction where only the transformer is dependant on the APT would be the better solution. By removing the dependency between the APT and AMT the data structures could be updated regardless of each other improving the maintainability. Future implementation of the optimization packages will need to implement an adaptor to transform internal data structures into an AMT. The adaptor has to make sure that necessary information remains available in the AMT.
The integration of the generator is directly dependant on the AMT. The generator will get the AMT as an argument and can handle the generation arbitrarily. The implementation of the visitor pattern helps to integrate the generator. The limitations of the visitor are highlighted by the inability to utilize the context of specific nodes. Most notably, the implementation of fused nodes becomes more complicated, since depending on the context, the current node has to be generated differently. This problem can be solved by introducing context dependant visitor implementations like the extended visitor introduced in Section 4.1.1. This would provide more flexibility for future generator implementations.

7.5 Synchronization and Data Transfers

The generation of synchronization and data transfers is currently handled by the AMT. Whenever data is used on a machine, on which the data is available, the current placement of the data and the new placement of the data is gathered in the AMT to create a data transfer node. Such nodes contain a set of input placements and a set of output placements for the same data element. This information on input and output data placements is used by the generator to define point-to-point

communication between the different devices/nodes.

7.5.1 Data Management

The global optimization of communication patterns influences the performance model and cost estimation of the optimization package. Although, a local optimization which can detect communication patterns within the data transfer nodes can improve the performance in the future.
By analysing the structure of these data transfer nodes, they could be optimized in order to utilize collective communication approaches. The utilization of efficient collective communication patterns improves the performance of the code [**?**], e.g., it is more efficient to utilize MPI_Broadcast then communicating with every node individually. For example, broad casting data to 16 different nodes takes 16 steps in a sequential unoptimized case, while the utilization of appropriately build binary trees could reduce the number of steps to five. In this scenario, nodes that already have the data also participate in the distribution. Such strategies are applicable for many collective communication patterns. The optimization of the communication within the AMT should be better handled by the optimization package.
The data allocation strategy is part of the data transfer and synchronization implementation and enforces a specific memory model. In Section 5.4, three different allocation strategies are introduced. The *complete allocation* strategy limits the possibilities of data-centric distributed memory optimizations, since it assumes, that a sequential implementation of the code would be computable in any case. This is not true for extremely larger applications where the data does not fit into the main memory of a single machine. This also holds for the *dependency complete allocation* strategy. Therefore, the *partial allocation* strategy or an abbreviation should extend the data transfers in the future.

7.5.2 Synchronization Overhead

Barriers are created with a focus on the correctness of the generated code, to avoid data races and deadlocks. As already mentioned in the discussion on performance, the amount of synchronization points can be reduced in future enhancements. By increasing the accuracy when analysing data accesses, implicit synchronization schemes can be utilized. The order of the tasks within the task queue of each thread is fixed. Therefore, tasks that only depend on their predecessor are synchronized implicitly, if and only if they are performed on the same thread. The utilization of such an implicit synchronization scheme cannot be utilized when the task queues are shared for multiple threads to utilize dynamic load balancing.
The implementation of the synchronization is by a factor of 2000 to 11000 more expensive compared to OpenMP. Therefore, it is necessary to improve the generated overhead in the future. In some cases, it is not possible to avoid synchronization. To be able to achieve good performance in these scenarios, the barrier implementa-

tion must be improved. To improve the synchronization implementation, overhead measurements are necessary in order to derive the cause for the elongated runtime. The average synchronization overhead of about eleven milliseconds measured in Section 6.2 indicates, that a large constant overhead dominates the synchronization in the thread pool. By detecting and improving upon the cause of the constant overhead, the generated code will be greatly improved. The cause for the overhead can be detected with the Intel VTune perfomance analyzer [**?**] in a hotspot analysis. Then the implementation can be improved to reduce the runtime overhead.

8 Conclusion and Future Work

The goal of the overarching project is to automatically apply global optimizations targeting heterogeneous cluster systems. The goal of this work is to provide the code generation for the overarching project. This thesis extended the IR defined in the overarching project by introducing the AMT. The AMT is a tree based structure which describes an algorithm after being globally optimized. It provides an interface for the optimization and the code generation. The structure contains information on the targeted architectures as well as the data flow between different nodes.
In this thesis, a code generator was developed and implemented based on the AMT. The code generator is the first implementation of the back-end for the overarching project. As such, the main focus of its development is primarily the correctness of the generated code in order to provide a suitable baseline for future implementations. The code generator supports shared-memory parallelism, distributed-memory parallelism and GPU offloading.
To measure the correctness of generated code, a test suite is developed. This test suite adopts the error classification defined by Schmitz et al. [**?**] in order to cover most of the potential error patterns within the generated code. The code generator developed in this thesis successfully generated correct code for all test cases.
The performance of the generated code is another important factor once the correctness is satisfactorily covered. The first results of the performance measurement are strongly influenced by the overhead of the thread pool implementation. The unoptimized thread pool implementation leads to multiplying the runtime overhead in comparison to OpenMP, e.g., the synchronization with a barrier takes multiple times as long compared to OpenMP. The complete synchronization takes about 2000 times longer than the OpenMP synchronization.
The performance evaluation against handwritten optimizations, as discussed in Section 6.2.2, shows that the performance is not on par with handwritten code yet. The generated code shows slowdowns between 2 and 100. Nevertheless, the result of the Monte Carlo test case shows that the generated MPI+PThreads hybrid code performs much better in comparison to the handwritten MPI code, because the shared-memory parallelism with PThreads scales much better than MPI. This indicates that the utilization of hybrid code is an appropriate step towards efficient generated code.
Towards real-world usage, the code generator can be improved in many areas in the future. Especially, performance improvements for shared-memory and GPU offloading are required. Reducing the overhead of synchronization and improving the use of fused nodes will further improve the performance of the generated code.
This is demonstrated in the Monte Carlo test case. The generated code already has

the potential to perform faster than handwritten optimizations, since hybrid code in general performs better on shared memory than by using MPI by itself. Therefore, algorithms targeting multiple nodes will benefit from the generated MPI+PThreads hybrid code. Furthermore, the global optimization greatly increases the portability of the original code, since the compiler package can automatically target different systems and optimize the code accordingly.
The next steps should include, the reduction of synchronization overhead in shared memory.
First, to achieve performance comparable to handwritten OpenMP code, the cost of barriers needs to be reduced drastically. Currently, a shared-memory synchronization is about 2000 to 11000 times more expansive than an OpenMP barrier.
Secondly, the generator needs to improve the generated code based on fused nodes, by generating combined tasks. This enables the production compilers to perform loop fusion optimizations. Especially, the generated GPU code benefits from these improvements, since additional data transfers and synchronizations will be avoided.
Dynamic scheduling strategies and reducing the number of synchronization and communication steps also benefits the efficiency of the generated code.

Bibliography

[1] FreeMarker tamplate engine. https://freemarker.apache.org/. Accessed: 2021-01-23.

[2] Intel Xeon Platinum 8160 Prozessor . https://ark.intel.com/content/www/de/de/ark/products/120501/intel-xeon-platinum-8160-processor-33m-cache-2-10-ghz.html. Accessed: 2021-02-15.

[3] I. C. R. Aachen. Hardware of the rwth compute cluster. https://help.itc.rwth-aachen.de/service/rhr4fjjutttf/article/e018f684c5624ae6b9bf7f0994d399f2. Accessed: 2021-02-15.

[4] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, et al. Tensorflow: A system for large-scale machine learning. In *12th {USENIX} symposium on operating systems design and implementation ({OSDI} 16)*, pages 265–283, 2016.

[5] H. Arabnejad, J. Bispo, J. M. Cardoso, and J. G. Barbosa. Source-to-source compilation targeting openmp-based automatic parallelization of c applications. *The Journal of Supercomputing*, pages 1–33, 2019.

[6] N. Bell and J. Hoberock. Thrust: A productivity-oriented library for cuda. In *GPU computing gems Jade edition*, pages 359–371. Elsevier, 2012.

[7] T. Ben-Nun, J. de Fine Licht, A. N. Ziogas, T. Schneider, and T. Hoefler. Stateful dataflow multigraphs: A data-centric model for performance portability on heterogeneous architectures. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–14, 2019.

[8] S. Blazy, Z. Dargaye, and X. Leroy. Formal verification of a c compiler front-end. In *International Symposium on Formal Methods*, pages 460–475. Springer, 2006.

[9] B. Blume, R. Eigenmann, K. Faigin, J. Grout, J. Hoeflinger, D. Padua, P. Petersen, B. Pottenger, L. Rauchwerger, P. Tu, et al. Polaris: The next generation in parallelizing compilers. In *Proceedings of the Seventh Workshop on Languages and Compilers for Parallel Computing*, pages 141–154. Citeseer, 1994.

[10] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan. Pluto: A practical and fully automatic polyhedral program optimization system. In *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation (PLDI 08), Tucson, AZ (June 2008)*. Citeseer, 2008.

[11] Boost Community. Boost C++ Library V. 1.75.0. `https://www.boost.org/doc/libs/1_75_0/`. Accessed: 2021-02-15.

[12] P. Bose. *Power Wall*, pages 1593–1608. Springer US, Boston, MA, 2011.

[13] J. M. Bull, F. Reid, and N. McDonnell. A microbenchmark suite for openmp tasks. In *International Workshop on OpenMP*, pages 271–274. Springer, 2012.

[14] M. Christen, O. Schenk, and H. Burkhart. Patus: A code generation and autotuning framework for parallel iterative stencil computations on modern microarchitectures. In *2011 IEEE International Parallel & Distributed Processing Symposium*, pages 676–687. IEEE, 2011.

[15] G. D. Costa, T. Fahringer, J. A. R. Gallego, I. Grasso, A. Hristov, H. Karatza, A. Lastovetsky, F. Marozzo, D. Petcu, G. Stavrinides, D. Talia, P. Trunfio, and H. Astsatryan. Exascale Machines Require New Programming Paradigms and Runtimes. *Supercomputing Frontiers and Innovations*, 2(2), 2015.

[16] C. Dave, H. Bae, S.-J. Min, S. Lee, R. Eigenmann, and S. Midkiff. Cetus: A source-to-source compiler infrastructure for multicores. *Computer*, 42(12):36–42, 2009.

[17] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[18] J. Dongarra, H. Meuer, E. Strohmeier, H. Simon, and M. Meuer. Top 500 Supercomputer Sites. November 2020. `https://www.top500.org/lists/top500/2020/11/`. Accessed: 2021-02-15.

[19] D. Etiemble. 45-year cpu evolution: one law and two equations. *arXiv preprint arXiv:1803.00254*, 2018.

[20] E. Gamma. *Design patterns: elements of reusable object-oriented software*. Pearson Education India, 1995.

[21] A. V. Gerbessiotis and L. G. Valiant. Direct bulk-synchronous parallel algorithms. *Journal of parallel and distributed computing*, 22(2):251–267, 1994.

[22] T. Grosser, A. Groesslinger, and C. Lengauer. Polly: performing polyhedral optimizations on a low-level intermediate representation. *Parallel Processing Letters*, 22(04):1250010, 2012.

[23] Z. Guo, B. Buyukkurt, J. Cortes, A. Mitra, and W. Najjar. A compiler intermediate representation for reconfigurable fabrics. *International Journal of Parallel Programming*, 36(5):493–520, 2008.

[24] L. Gurobi Optimization. Gurobi Optimizer Reference Manual. `http://www.gurobi.com`. Accessed: 2021-02-15.

[25] T. Gysi, C. Müller, O. Zinenko, S. Herhut, E. Davis, T. Wicky, O. Fuhrer, T. Hoefler, and T. Grosser. Domain-specific multi-level ir rewriting for gpu. *arXiv preprint arXiv:2005.13014*, 2020.

[26] M. Harris et al. Optimizing parallel reduction in cuda. `https://developer.download.nvidia.com/assets/cuda/files/reduction.pdf`. Accessed: 2021-02-15.

[27] J. Holewinski, L.-N. Pouchet, and P. Sadayappan. High-performance code generation for stencil computations on gpu architectures. In *Proceedings of the 26th ACM international conference on Supercomputing*, pages 311–320, 2012.

[28] S. Jeanmart, Y.-G. Gueheneuc, H. Sahraoui, and N. Habra. Impact of the visitor pattern on program comprehension and maintenance. In *2009 3rd International Symposium on Empirical Software Engineering and Measurement*, pages 69–78. IEEE, 2009.

[29] H. Jordan, S. Pellegrini, P. Thoman, K. Kofler, and T. Fahringer. Inspire: The insieme parallel intermediate representation. In *Proceedings of the 22nd international conference on Parallel architectures and compilation techniques*, pages 7–17. IEEE, 2013.

[30] S. Kamil, C. Chan, L. Oliker, J. Shalf, and S. Williams. An auto-tuning framework for parallel multicore stencil computations. In *2010 IEEE international symposium on parallel & distributed processing (IPDPS)*, pages 1–12. IEEE, 2010.

[31] D. Khaldi, P. Jouvelot, F. Irigoin, C. Ancourt, and B. Chapman. Llvm parallel intermediate representation: Design and evaluation using openshmem communications. In *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*, pages 1–8, 2015.

[32] A. Klöckner, N. Pinto, Y. Lee, B. Catanzaro, P. Ivanov, and A. Fasih. Pycuda and pyopencl: A scripting-based approach to gpu run-time code generation. *Parallel Computing*, 38(3):157–174, 2012.

[33] R. K. Kombi, N. Lumineau, and P. Lamarre. A preventive auto-parallelization approach for elastic stream processing. In *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*, pages 1532–1542. IEEE, 2017.

[34] P. Konecny. Introducing the cray xmt. In *Proc. Cray User Group meeting (CUG 2007). Seattle, WA: CUG Proceedings*, 2007.

[35] J. Lambert, S. Lee, A. Malony, and J. S. Vetter. Ccamp: Openmp and openacc interoperable framework. In *European Conference on Parallel Processing*, pages 357–369. Springer, 2019.

[36] X. Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, 2009.

[37] A. R. Mamidala, R. Kumar, D. De, and D. K. Panda. Mpi collectives on modern multicore clusters: Performance optimizations and communication characteristics. In *2008 Eighth IEEE International Symposium on Cluster Computing and the Grid (CCGRID)*, pages 130–137. IEEE, 2008.

[38] S. M. Martin, D. Wälchli, G. Arampatzis, and P. Koumoutsakos. Korali: a high-performance computing framework for stochastic optimization and bayesian uncertainty quantification. *arXiv preprint arXiv:2005.13457*, 2020.

[39] T. G. Mattson, B. Sanders, and B. Massingill. *Patterns for Parallel Programming*. Pearson Education, Amsterdam, 2004.

[40] M. D. McCool, A. D. Robison, and J. Reinders. *Structured Parallel Programming - Patterns for Efficient Computation*. Elsevier, Amsterdam, 2012.

[41] S. A. McKee and R. W. Wisniewski. *Memory Wall*, pages 1110–1116. Springer US, Boston, MA, 2011.

[42] J. Miller, L. Trümper, C. Terboven, and M. S. Müller. Poster: Efficiency of algorithmic structures. In *International Conference on High Performance Computing, Networking, Storage and Analysis (SC19)*. IEEE/ACM, november 2019.

[43] MPI Forum. MPI: A Message-Passing Interface Standard, Version 3.1. `https://www.mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf`. Accessed: 2021-02-15.

[44] C. J. Newburn, B. So, Z. Liu, M. McCool, A. Ghuloum, S. D. Toit, Z. G. Wang, Z. H. Du, Y. Chen, G. Wu, P. Guo, Z. Liu, and D. Zhang. Intel's array building blocks: A retargetable, dynamic compiler and embedded language. In *International Symposium on Code Generation and Optimization (CGO 2011)*, pages 224–235, 2011.

[45] B. Nichols, D. Buttlar, and J. P. Farrell. *Pthreads programming - a POSIX standard for better multiprocessing*. O'Reilly, 1996.

[46] Nvidia. Cuda 11 documentation. `https://docs.nvidia.com/cuda/`. Accessed: 2021-02-15.

[47] NVIDIA V100 GPU ARCHITECTURE. `https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf`. Accessed: 2021-02-15.

[48] OpenACC User Group. OpenACC 3.0 Specification. `https://www.openacc.org/sites/default/files/inline-images/Specification/OpenACC.3.0.pdf`. Accessed: 2021-01-23.

[49] OpenMP Architecture Review Board. OpenMP 5.1 Specification. `https://www.openmp.org/wp-content/uploads/openmp-5-1.pdf`. Accessed: 2021-02-01.

[50] T. J. Parr and R. W. Quong. Antlr: A predicated-ll (k) parser generator. *Software: Practice and Experience*, 25(7):789–810, 1995.

[51] M. J. Quinn. Parallel programming. *TMH CSE*, 526:105, 2003.

[52] J. Reinders. Vtune performance analyzer essentials. *Intel Press*, 2005.

[53] B. Rumpe and K. Hölldobler. *Monticore 5 language workbench. edition 2017*. Shaker Verlag, 2017.

[54] C. A. Schaefer, V. Pankratius, and W. F. Tichy. Atune-il: An instrumentation language for auto-tuning parallel applications. In *European Conference on Parallel Processing*, pages 9–20. Springer, 2009.

[55] T. B. Schardl, W. S. Moses, and C. E. Leiserson. Tapir: Embedding fork-join parallelism into llvm's intermediate representation. In *Proceedings of the 22Nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 249–265, 2017.

[56] A. Schmitz, J. Protze, L. Yu, S. Schwitanski, and M. S. Müller. Dataraceonaccelerator a micro-benchmark suite for evaluating correctness tools targeting accelerators. In *European Conference on Parallel Processing*, pages 245–257. Springer, 2019.

[57] C. Silvano, G. Agosta, S. Cherubin, D. Gadioli, G. Palermo, A. Bartolini, L. Benini, J. Martinovič, M. Palkovič, K. Slaninová, et al. The antarex approach to autotuning and adaptivity for energy efficient hpc systems. In *Proceedings of the ACM International Conference on Computing Frontiers*, pages 288–293, 2016.

[58] C. Simmendinger, M. Rahn, and D. Gruenewald. The GASPI API: A failure tolerant PGAS API for asynchronous dataflow on heterogeneous architectures. In *Sustained Simulation Performance 2014*, pages 17–32. Springer, 2015.

[59] M. Steuwer, T. Remmelg, and C. Dubach. Lift: a functional data-parallel ir for high-performance gpu code generation. In *2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 74–85. IEEE, 2017.

[60] G. Tournavitis, Z. Wang, B. Franke, and M. F. O'Boyle. Towards a holistic approach to auto-parallelization: integrating profile-driven parallelism detection and machine-learning based mapping. *ACM Sigplan notices*, 44(6):177–187, 2009.

[61] L. Trümper. Global optimization of parallel pattern-based algorithms for heterogeneous architectures. In *Master thesis, RWTH Aachen University*, 2020.

[62] S. Verdoolaege, J. Carlos Juega, A. Cohen, J. Ignacio Gomez, C. Tenllado, and F. Catthoor. Polyhedral parallel code generation for cuda. *ACM Transactions on Architecture and Code Optimization (TACO)*, 9(4):1–23, 2013.

[63] J. Vitay, H. Ü. Dinkelbach, and F. H. Hamker. Annarchy: a code generation approach to neural simulations on parallel hardware. *Frontiers in neuroinformatics*, 9:19, 2015.

[64] D. Wälchli, S. M. Martin, A. Economides, L. Amoudruz, G. Arampatzis, X. Bian, and P. Koumoutsakos. Load balancing in large scale bayesian inference. In *Proceedings of the Platform for Advanced Scientific Computing Conference*, PASC '20, New York, NY, USA, 2020. Association for Computing Machinery.

[65] S. Williams, A. Waterman, and D. Patterson. Roofline: An Insightful Visual Performance Model for Multicore Architectures. *Commun. ACM*, 52(4):65–76, Apr. 2009.

[66] C. Yount, J. Tobin, A. Breuer, and A. Duran. Yask:yet another stencil kernel: A framework for hpc stencil code-generation and tuning. In *2016 Sixth International Workshop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing (WOLFHPC)*, pages 30–39. IEEE, 2016.

[67] J. Zhao and V. Sarkar. Intermediate language extensions for parallelism. In *Proceedings of the compilation of the co-located workshops on DSM'11, TMC'11, AGERE! 2011, AOOPES'11, NEAT'11, & VMIL'11*, pages 329–340, 2011.

[68] M. Zheng, M. S. Rogers, Z. Luo, M. B. Dwyer, and S. F. Siegel. Civl: formal verification of parallel programs. In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 830–835. IEEE, 2015.