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

Entwurf eines statischen Leistungsmodells und Codegenerierung für Vektorbeschleuniger und Parallele Muster

Designing a Static Performance Model and Code Generation for Vector Accelerators and Parallel Patterns

Bachelorarbeit

Tom Hilgers
Matrikelnummer: 406581

Aachen, den 13. Januar 2023
Communicated by Prof. Matthias S. Müller

Erstgutachter:	Prof. Dr. rer. nat. Matthias S. Müller (’)
Zweitgutachter:	Dr. rer. nat. Stefan Lankes (*)
Betreuer:	Semih Burak, M. Sc. (’) Adrian Schmitz, M. Sc. (’)

(’) Lehrstuhl für Hochleistungsrechnen, RWTH Aachen University
IT Center, RWTH Aachen University
(*) Institute for Automation of Complex Power Systems,
E.ON Energy Research Center, RWTH Aachen University

Kurzfassung

Moderne Supercomputersysteme haben eine verteilte und heterogene Struktur, die verschiedene Recheneinheiten wie CPUs, GPUs und andere Beschleuniger umfasst. Die Parallel Pattern Language (PPL) ermöglicht die hardwareunabhängige Programmierung solcher Systeme mit parallelen Mustern und generiert global optimierten Code unter Verwendung des Roofline Leistungsmodells. Diese Arbeit zielt darauf ab, die PPL zu erweitern, um Vektorbeschleuniger (VB) zu unterstützen, welche eine höhere Energieeffizienz als die gängigsten Recheneinheiten bieten. Das Roofline Modell wird erweitert, indem die besonderen architektonischen Merkmale von VBs berücksichtigt werden, wodurch eine Genauigkeit von ∼80% bis 99% erreicht wird. Die PPL-Komponenten werden hinsichtlich der notwendigen Änderungen für die VB-Codegenerierung analysiert. Die Generierung von funktionalem Code erfordert nur begrenzte Änderungen an der Codegeneratorkomponente. Die Steigerung der Effizienz von Datentransfers in diesem Code erfordert Änderungen an der Frontendsprache und der Intermediate Representation. Diese Vorschläge werden zum Teil in einem Proof-of-Concept umgesetzt, das in der Lage ist, funktionalen und korrekten Code für VBs zu erzeugen. Dieser Code wird evaluiert, um mögliche Leistungsverbesserungen zu ermitteln.

Stichwörter: HPC, Vector Engine, Codegenerierung, Parallele Muster, Leistungsmodell

Abstract

Modern supercomputer systems have a distributed and heterogeneous structure that incorporates various compute units such as CPUs, GPUs, and other accelerators. The Parallel Pattern Language (PPL) enables the hardware-independent programming of such systems with parallel patterns and generates globally optimized code using the Roofline performance model. This thesis aims to extend the PPL to support Vector Accelerators (VAs), which offer a higher energy efficiency than the most common compute units. The Roofline model is extended by considering the special architectural features of VAs, which achieves an accuracy of ∼80% to 99%. The PPL components are analyzed regarding the necessary modifications for VA code generation. Generating functional code requires limited changes to the code generator component. Increasing the efficiency of data transfers in this code necessitates changes to the frontend language and intermediate representation. These proposals are partially implemented in a proof-of-concept that is capabable of generating functional and correct code for VAs. This code is evaluated to identify potential performance improvements.

Keywords: HPC, Vector Engine, Code Generation, Parallel Patterns, Performance Model

Contents

List of Figures

List of Tables

1 Introduction

The performance of the most powerful supercomputer systems continues to increase despite the slowdown of Moore's Law [85]. The exascale era in HPC has arrived with supercomputers like Frontier [74], JUPITER [30], and Aurora [9]. Such supercomputer clusters are facing new challenges due to rising energy prices. The growing gap between compute and memory performance reduces the energy efficiency of memory bound codes on the most common hardware, i.e., CPUs und GPUs. A solution to this problem can be alternative architectures realized as specialized accelerators. Vector Accelerators (VAs), in the form of NEC Vector Engines (VEs) are such an alternative due to their higher energy efficiency [47].

Todays supercomputer clusters have a distributed and heterogeneous structure. A large number of individual nodes, that contain various compute units, are connected to form a single system. Optimizing applications on such systems requires a detailed understanding of the structure and used hardware. Such applications are often not fully optimized because application developers are experts in their own field of research and rarely in HPC. Even for HPC experts, learning the system structure and programming models is a complex and time consuming task. The unique structure of each cluster also limits the portability of applications. A solution to these problems can be the automatic generation of optimized code.

The Parallel Pattern Language (PPL) is being developed to offer such a solution. Currently in a prototype development stage, it is a component based source-to-source compiler, where applications are developed in a parallel pattern based Domain Specific Language (DSL). This enables a hardware-independent expression of parallelism. From this code, an intermediate representation [83] is generated in the form of an Abstract Pattern Tree (APT), on which a global optimization [96, 95, 61] is then applied. Several algorithmic efficiencies are used to account for a given system structure. The choice of the hardware on which an application is to be executed in whole or in part is based on a performance model. Based on this decision, optimized source code is then generated [82]. The performance model and code generator component support CPUs and NVIDA GPUs with their respective execution and programming models.

The general goal of this thesis is to lay the foundations for the generation of code for VEs by the PPL. A static performance model for VEs is developed as the first step. It has to consider the special architectural features of VEs. For example, the performance of an algorithm directly depends on its vectorizability. Another major characteristic is the memory hierarchy with a single cache level between registers and main memory. The validity of the model model is examined with several benchmarks that are partially based on real-world algorithms.

As the second step of the foundational work, code generation for VEs is investigated in the context of the PPL. General objectives are specified and requirements for the individual PPL components are defined. Suggestions for the implementation, which consider the distinct programming and execution model of VEs, are given in the context of this definition.

The code generation proposals are partially implemented in a proof-of-concept extension of the PPL that can serve as a foundation for subsequent work. The generated code for VEs is evaluated in terms of correctness and performance. It is compared with handwritten VE code, as well as generated code for CPUs and GPUs. Conclusions are drawn on optimization potential in the new implementation and the PPL in general.

The validation of the performance model is performed with several benchmarks. These are K-Means clustering (K-Means), K-Nearest neighbors (NN), a Jacobi solver, and a linear shifter. With the exception of the latter, all benchmarks are based on real-world algorithms and thus enable a realistic assessment of the performance model. The assumptions made in the model about the memory architecture of VEs are investigated with the STREAM benchmark [57]. The NN benchmark is further used to evaluate the proof-of-concept code generator.

The main contributions of this thesis are:

- The definition and validation of a static performance model for VEs.
- The design of code generation for VEs in the PPL by specifiying requirements and proposing implementation approaches.
- The partial implementation and subsequent evaluation of the specified code generator.

The thesis is structured as follows. Chapter 2 provides an overview of related work. Chapter 3 introduces the relevant concepts and foundations. These are the Roofline performance model, parallel patterns, the PPL project, Vector Processors (VPs), and the used benchmarks. Chapter 4 presents and validates the developed performance model. The chapter further describes how the model can be integrated into the PPL. The specifications and requirements for the PPL to generate code for VEs are described in Chapter 5. Additionally, a proof-of-concept implementation is presented and evaluated. Chapter 6 concludes this thesis by summarizing the results and giving an outlook on possible future work.

2 Related Work

There exists a wide range of literature on performance modeling. Not only are there many different performance models, but also extensions of these for new applications. Automated code analysis tools utilize these models for hardware-specific performance predictions. Automatic code generation is also a subject of research, for example in conjunction with code optimization techniques. All of this is discussed in this chapter and the main differences to this work are highlighted.

2.1 Performance Models

Algorithm performance on specific hardware can be abstractly modeled. Such performance models are broadly divided into two categories: static and dynamic. Static models only consider information that is available before a program is executed. Dynamic models use information obtained through the execution of a program for more accurate performance predictions. Both are frequently extended to include new hardware architectures. This work extends the scope of hardware architectures covered by static models with a new extension of the Roofline model for Vector Processors (VPs) in general and the NEC SX-Aurora TSUBASA Vector Engine (VE) in particular.

2.1.1 Roofline Model

The Roofline performance model [100] statically provides an upper bound on the floating-point performance of a computational kernel on a given architecture. The model is widely used, as can be seen by the plethora of published literature [104, 52, 20, 34], some of which is presented in this section.

Many extensions have been developed because the original model is designed for regular CPU architectures and does not consider special architectural features. Ilic et al. [42] extend the model by considering the entire memory hierarchy of modern CPUs because the original model only considers the main memory. Their model has an upper bound for each level of the memory hierarchy, which they term a cache-aware Roofline model. An extension used in various publications [53, 48, 103, 104] considers the different peak performance levels achievable on modern processors in a hierarchical model. They provide multiple upper bounds, e.g., for half, single, double precision, and more. The result of both cache-aware and hierarchical Roofline models is a comprehensive abstraction of the system architecture, which enables a detailed static analysis of computational kernels. With regard to the vector architecture used in this thesis, the Roofline model represents a suitable option for performance modeling. This is because, like other architectures, they are primarily designed for floating-point performance and have a comparable memory structure. The cache-aware Roofline model is used as a starting point, as it incorporates the memory hierarchy consisting of main memory and cache. The concept of multiple compute roofs is also applied, using the vector length as the key factor for the achievable performance.

Both the original and some of the extended Roofline models have been adapted to other architectures. However, to the best of the author's knowledge, no extension specifically exists for vector architectures. Lopes et al. [52] explore the performance, power consumption, and energy efficiency limits of computational kernels on GPUs by adapting the cache-aware Roofline model. Another extension for GPUs replaces the floating-point operation-based approach with an instruction-based model to gain deeper insights into performance limits [23, 24]. To model Field Programmable Gate Array (FPGA) performance, Da Silva et al. [20] incorporate results from high-level synthesis tools. Siracusa et al. [88] use a similar approach where the Roofline model guides an automatic design optimizer to improve the performance of HPC applications on FPGAs. Extensions also exist for other architectures, such as mobile Systems on a Chip (SoCs) [36] and compute nodes with heterogeneous memory [21]. Choi et al. [16] show that the Roofline model is also useful in modeling the energy consumption of algorithms. All these works consider the unique characteristics of the respective hardware in the Roofline analysis and equation. This work applies this approach to VEs with the vector length and effective memory bandwidth extensions.

Cabezas and Püschel [13] further develop the Roofline model by including an extended set of hardware-related bottlenecks. They also describe a tool that analyzes a given computational kernel using an execution model with directed acyclic graphs, which is similar to the Abstract Pattern Tree (APT) of the Parallel Pattern Language (PPL). The Roofline model and its extensions have also proven useful in dynamic performance analysis, providing an easy-to-understand visualization of the theoretically achievable and practically achieved performance. A well-known example of a dynamic analysis tool that utilizes the Roofline model is Intel Advisor [87]. Koskela et al. [48] describe it as an integrated approach that considers cache awareness, cache blocking, and different peak performance levels. Marques et al. [53] evaluate Intel Advisor and its use of the Roofline model. Since the PPL performs a fully automated static analysis, the previously mentioned approaches are not pursued in this work.

2.1.2 Other Models and Usage

The Execution-Cache-Memory (ECM) model, proposed by Treibig and Hager [94], is based on the same fundamental idea of Roofline that the runtime of computational kernels is either determined by the peak compute power or the data transfer time. It requires an analysis of the entire cache and memory hierarchy of modern microarchitectures. The model has been applied and evaluated on various architectures and systems in a series of publications [33, 101, 89, 39, 38, 40] by authors from the University of Erlangen-Nuremberg and associated institutions. It is not used in this work based on an evaluation of the advantages and disadvantages in comparison to the Roofline model. This discussion can be found in Section 4.1.1.

The previously mentioned models are specialized for predicting the runtime time of algorithms on a single compute unit. Modern systems use a large number of different compute units in parallel, e.g., multiple CPU sockets and accelerators distributed among nodes in a cluster. Communication and network latency must be considered to model the execution of programs on multiple nodes. Culler et al. [19] introduced the LogP model in 1993, where they factor in communication latency, overhead, bandwidth, and the number of processing units. Similar to the Roofline model, several extensions and refinements have been proposed. LogGP [6] incorporates the impact of long messages into the model to better capture the benefits of large data transfers. LogCA [7] models the impact and speedup that accelerators can have on a system. An introduction of these models into the PPL for modeling network costs was already proposed and to some extent implemented in previous work [95]. As a result, a sufficient modeling is already achieved, making it unnecessary to extend it in this work.

Performance models have proven useful in automated static code analysis, where they can be used to improve performance predictions and find potential bottlenecks. A number of frameworks have been developed for this purpose. A recent example is Kerncraft, presented by Hammer et al. [34] in 2015. It focuses on the memory hierarchy and uses both the Roofline and ECM models in combination with a code analyzer. Another tool for static performance analysis and modeling is MIRA [59]. It has some internal similarities to Kerncraft, but focuses on compute, i.e. floating-point, performance and bottlenecks. The methods of these frameworks are not applied to the PPL code analysis in this work, since the performance modeling already achieves a satisfactory accuracy with the existing code analysis. Beyond Kerncraft and MIRA, there are more analysis frameworks, some of which also include dynamic or hybrid approaches. Examples include MAQAO [25], Empirical Roofline Toolkit [51], ExaSat [97], and PBound [63].

2.2 Code Generation

The current field of code generation for VPs and VEs encompasses only a limited amount of literature. The NEC compiler [69] supports C, C++ and Fortran compilation to machine code for VEs. The LLVM compiler also provides such functionality [49] and further supports OpenMP offloading to VEs [17, 18]. As in the PPL [83], both generate an intermediate representation from the source code. Unlike in the PPL, binary machine code is generated instead of new source code. In this work, an extension of the PPL to generate source code for VEs is designed and partially implemented. A comparable approach that generates VE source code is not known to the author. The code generator implementation of the PPL presented by

Schmitz [82] is extended in this work. The PPL utilizes a source-to-source approach, from a Domain Specific Language (DSL) based on parallel patterns to parallelized C++ code. A comparable pattern-based development tool is presented by Modak et al. [62]. Reiche et al. [79] present another approach that utilizes a DSL for hardware accelerator code generation. Only NVIDIA GPUs can be used as offload targets in the existing PPL implementation. This is achieved by generating parallel CUDA code. In this work, the existing code generator is extended for VEs by using VEDA [68] instead of the CUDA library. VEDA is only used in a limited number of works [11, 98].

3 Background

This chapter presents the relevant technical background to this thesis in five separate sections. Section 3.1 explains the Roofline model and its usage. Section 3.2 then introduces the concept of parallel patterns. Section 3.3 describes the Parallel Pattern Language (PPL), its concept and core components. Vector Accelerators (VAs) are described in Section 3.4 by introducing the general concept and special features of Vector Processors (VPs). The section further presents and describes the used hardware architecture and its most important features. An overview of the used benchmarks is given in Section 3.5. These are relevant for the model validation in Section 4.2 and performance evaluation of the proof-of-concept in Section 5.5

3.1 Roofline Model

Static performance models are used to estimate the performance of an algorithm before it is executed. They have a simplified view of hardware and code properties, but capture the most important factors influencing performance. This performance is typically defined as the number of floating-point operations per second. The expected runtime can be calculated if it is known how many such operations an algorithm will perform.

The Roofline model [100] statically provides an upper performance bound for a computational kernel on a given hardware. It incorporates the main code and hardware characteristics in its calculation. In this work, the cache-aware Roofline model [42] is used as a starting point to consider the entire memory hierarchy of the modeled architecture. An important advantage of the Roofline model is the possibility of visualizing the achieved performance in relation to the performance bounds.

An architecture is characterized in the Roofline model by its peak performance *Ppeak* in floating-point operations per second (Flop/s) and the peak memory bandwidth β in bytes per second (Byte/s). The cache-aware Roofline model [42] makes a distinction between β_{mem}, for the main memory, and β_{cache}, for the cache. A computational kernel is represented by its operational intensity I, i.e., the number of bytes transferred per floating-point operation. These variables are listed in Table 3.1 for a better overview. The formula of the Roofline model to calculate the upper performance bound π is defined as

$$\pi = \min(Ppeak, I \cdot \beta). \tag{3.1}$$

Variable	Description	Unit
π	Attainable performance	GFlop/s
$Ppeak$	Peak hardware performance	GFlop/s
I	Operational intensity	Flop/Byte
β	Peak memory bandwidth	GByte/s

Table 3.1: A description of the key variables used in the Roofline model and their units of measurement.

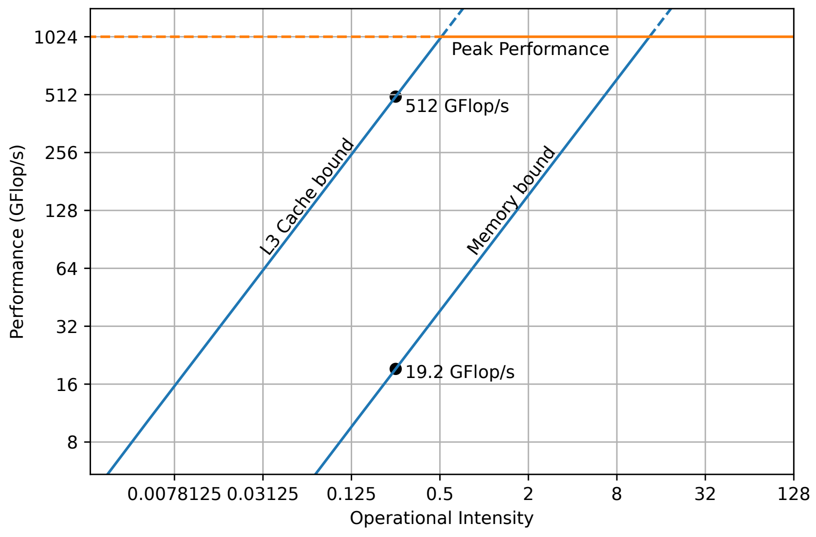


Figure 3.1: A sample visualization for the cache-aware Roofline model. Peak performance, main memory bandwidth, and L3 cache bandwidth are given as performance bounds. An algorithm with an operational intensity of 0.25 is fitted into these.

The cache-aware Roofline model is now explained in more detail with an example. Assume that a sample algorithm with an operational intensity of 0.25 is to be executed on an architecture with a peak performance of 1024 GFlop/s, 76.8 GByte/s main memory bandwidth, and 2 TByte/s L3 cache bandwidth. Other cache levels are omitted. This data is visualized in Figure 3.1 where the x-axis represents the operational intensity and the y-axis represents the achievable performance of the algorithm. It can be seen that the algorithm is memory or cache bound, therefore a distinction between the two must be made. The performance can reach 19.2 GFlop/s in case of memory boundness and 512 GFlop/s if it is cache bound. An implementation that does not achieve these values can be further optimized.

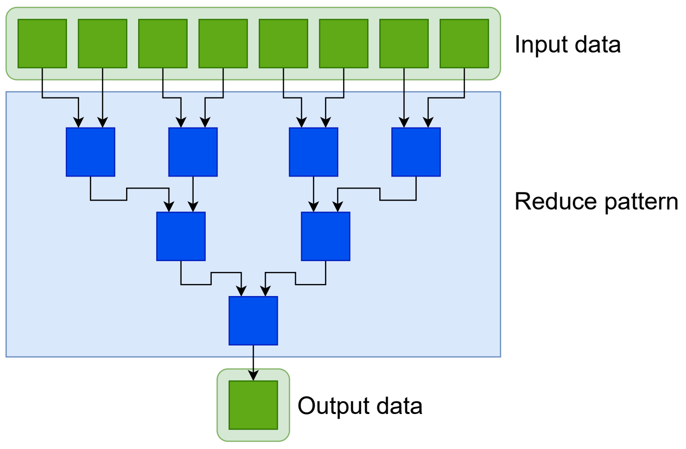


Figure 3.2: The *Reduce* parallel pattern that combines partial results in the form of a tree. Based on [58].

3.2 Parallel Patterns

Software design patterns are generalized and reusable solutions for frequently occurring problems in software development. Parallel patterns [56, 58] describe such solutions for the concurrency domain of algorithms. Two parallel patterns are presented as an example in the following.

The *Map* pattern describes the independent application of a specific operation to the elements of a dataset. In practice, this means the implementation of a loop without dependencies between the iterations. The lack of dependencies allows an arbitrary level of parallelization to take place.

Another example is the *Reduce* pattern, which reduces a dataset into a single result by combining partial results. The pattern is often used to calculate the sum, minimum or maximum of the values of a dataset. Figure 3.2 provides a visualization of this pattern. It shows the input data which is combined step by step in the form of a tree to create a single output element. Parallelization can be achieved by assigning each combination of partial results to one thread. The degree of achievable parallelism is equal to the number of combinations per reduction step and decreases in each. The first step in the figure encompasses four combinations to reduce the eight input elements to four, which allows the use of four threads. This number is halved in the second step, where only two combinations take place. The final reduction to the single output element can then only be performed by a single thread.

3.3 Parallel Pattern Language

The PPL [61, 83, 96, 82] is a framework and toolchain for the efficient development of parallel programs for heterogeneous architectures. This section introduces the concept of global optimization and gives an overview over the toolchain components.

The descriptions are partially based on and inspired by previous work [12].

3.3.1 Global Optimization

The performance of an algorithm can be optimized on instruction, routine (local) and algorithm (global) levels. Both instruction and local level optimizations are already performed by hardware, compilers and with parallel programming models such as OpenMP [75]. The PPL builds on this and performs a global optimization in the context of execution on heterogeneous and distributed systems [95]. A global optimization considers the entire structure of an algorithm, to improve, for example, the overall parallelizability. For this purpose, larger structural elements, such as functions or parallel patterns, may be reordered, splitted or fused. This is accomplished in the PPL by representing the algorithmic structure in an Abstract Pattern Tree (APT) [61] where the nodes represent the parallel patterns and structures. The control and dataflow dependencies are represented by child nodes. Based on the APT, the global optimization is performed in three steps. Each step maximizes a well-defined algorithmic efficiency:

- Synchronization: Execute as many nodes, i.e. patterns, in parallel as possible.
- Inter-processor dataflow: Runtime minimal mapping of nodes to processors.
- Intra-processor dataflow: Runtime minimal mapping of tasks to the cores of a processor.

The global optimization is statically applied, which means that it relies entirely on the structural information obtained from the APT. No dynamic, i.e. runtime information about the actual computation is considered.

A set of global steps *GSTEP* can be obtained from the APT, where each represents a set of tasks, i.e. parallel patterns, that can be executed in parallel. The optimization based on the synchronization efficiency is hardware independent and seeks to minimize the total number of *GSTEP* while maximizing the number of tasks within each. This efficiency therefore represents the structural parallelism of the program.

For the inter-processor dataflow efficiency, the set of tasks within a *GSTEP* is mapped to the target architecture, described by a set of processors. The objective is a runtime-minimal mapping of tasks to processors. A processor is a homogeneous set of cores with a locally shared cache, such as a single CPU in a multiprocessor system or a streaming multiprocessor of a GPU. The optimization considers the execution cost of a task and the network cost of the necessary data transfers in order to statically model the runtime. The execution costs are based on the Roofline performance model [100], while the network costs are based on the LogP model [6].

Optimization on the basis of the intra-processor dataflow efficiency is performed by the code generator component. It optimizes the allocation of tasks to the individual cores of a processor by parallelizing with PThreads [71].

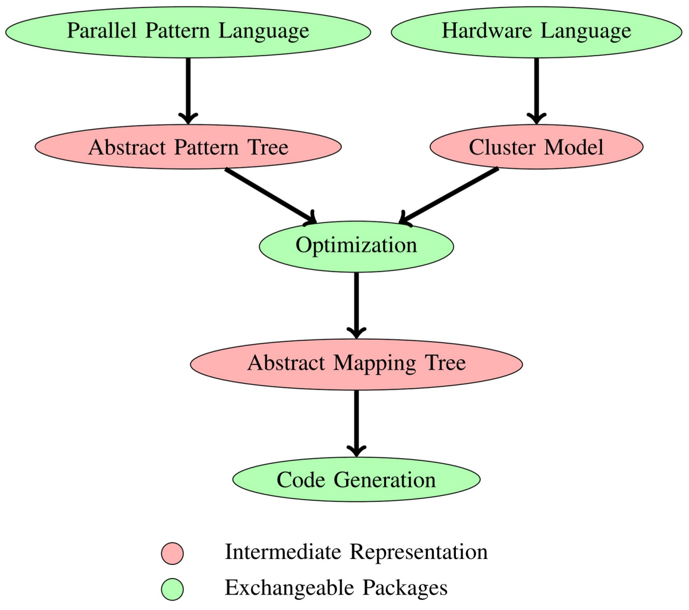


Figure 3.3: The toolchain components and workflow of the PPL. Taken from [83].

3.3.2 Toolchain Overview

The PPL implementation [83] is a component-based toolchain in which some parts are interchangeable. It acts as a source-to-source compiler to increase application portability and let regular compilers perform additional, i.e. instruction and local, optimizations. An overview of the components and workflow can be found in Figure 3.3.

The central component is the optimizer, which performs the previously described global optimization. The optimization requires two input parameters, an APT and a hardware description. The APT is generated from source code in a Domain Specific Language (DSL). This simplifies the description of parallel patterns and thus also the generation of the APT. The description of the hardware on which a code is to be executed is given by a cluster model. It is generated from the JSON based Hardware Language (HL). The DSL and HL are the frontend to the developer. The result of the global optimization is a modified structure of the APT and a mapping to the given hardware. This information is stored in an Abstract Mapping Tree (AMT), in which the nodes now also model the execution on devices like GPUs and the necessary data transfers.

The information in the AMT is used by a code generator [82] that generates optimized and parallelized code for the respective target architectures. C++ code is generated to enable further optimization at instruction and local level by compilers. The generator also optimizes at the local level by parallelizing with PThreads for CPUs and CUDA for GPUs. The generated code uses MPI to address distributed memory in a heterogeneous system. A static Makefile is generated for the compilation on the target system.

Of these components, APT, cluster model, and AMT are part of the intermediate representation and are non-interchangeable. The frontend language, HL, optimizer, and code generator are freely interchangeable. For example, a new code generator supporting a different target architecture can be used instead of the current one by using the AMT as an interface.

3.4 Vector Accelerators

Flynn's taxonomy [26] distinguishes between several methods of executing instructions on data. Basic CPUs execute one instruction on one data element. This is referred to as Single Instruction Single Data (SISD). However, other methods can be used to improve performance. VPs use a technique called Single Instruction Multiple Data (SIMD). In this method, an instruction is executed on a group of data elements, called a vector. This, when supported by a memory system with sufficient bandwidth, can dramatically increase performance [31] while maintaining a similar power requirement [47, 1, 32]. SIMD is especially beneficial in multimedia [86, 5] and scientific applications [90, 35, 99] where only a few instructions are executed on a large amount of data. This section elaborates on the concept of VPs, their differences to CPUs and GPUs, and provides a description of the VP used in this thesis.

3.4.1 Vector Processors

VPs have many distinctive features that separate them from other types of processors that might also implement SIMD. VPs usually utilize much longer vectors than regular CPUs with SIMD, thus processing more elements per instruction. Another difference is that VPs feature a fully variable vector length, meaning that a vector can take any length up to the architectural limit. Branching, i.e. conditional code, can slow down the pipelining of instructions, especially in the case of incorrect branch prediction. This is mitigated in VPs with predication [41, 78], where all possible branch paths are executed, but only on the vector elements for which the conditions are actually met. This is accomplished by using a mask vector register that contains the boolean values for each vector element and applying it to the conditional instruction. In Flynn's taxonomy, a CPU that implements predication is called *associative* [27]. To further improve performance, vector chaining [37, 93] is implemented. With this method, the results of one instruction are directly fed back into the execution units without additional memory references, which reduce performance.

Modern CPUs use instruction pipelining, where instructions are executed in several substeps. VPs apply this concept to the data itself. Figure 3.4 illustrates this concept. The left side shows how CPUs with short vectors process data; in this case with a vector length of four. The right side shows how VPs process data. Their vectors are much longer, 32 elements in this example, and are only partially processed in each

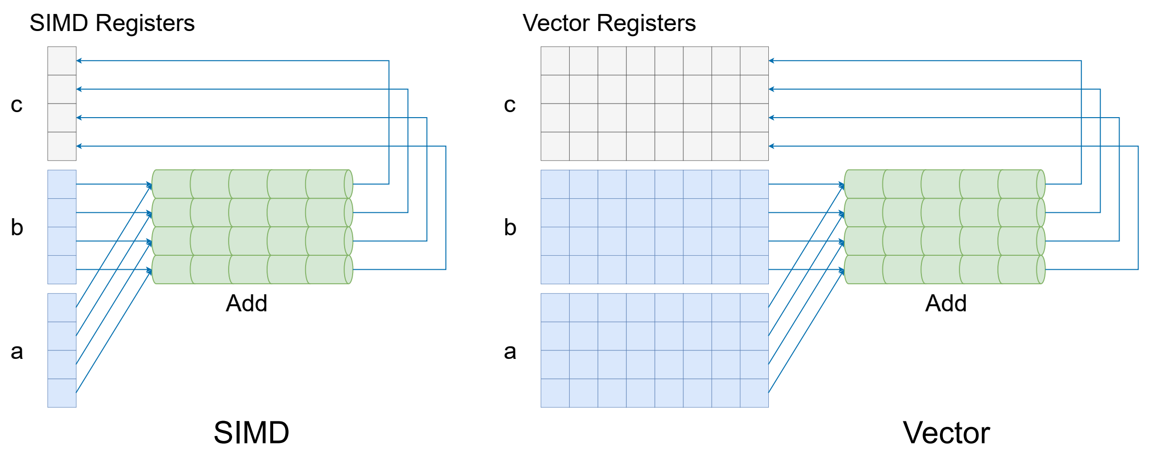


Figure 3.4: Vector Processors feature larger vector registers than CPUs with SIMD capabilities.

cycle. Meaning that a single instruction is executed over multiple cycles, eight in this case. This reduces the load and latency of the instruction fetch and decode units. In addition, implicit loops can be formed, reducing the number of instructions and thus increasing memory efficiency. The (main) advantage of VPs over traditional CPUs and GPUs is a higher compute and energy efficiency, particularly for memory-bound codes [47].

Vector Accelerators are a special form of VPs. They are no longer the main execution unit of the system, but instead act as a co-processor. An x86 CPU acts as the main processor. VAs today take the form of a VP on a PCIe expansion card. Therefore, they experience the bottleneck of the PCIe bus to the host CPU and main memory, which can significantly slow down execution. For this reason VAs usually feature a dedicated high-speed memory subsystem.

3.4.2 Comparison to CPUs and GPUs

There are many differences between CPUs, GPUs and VPs, in particular in the way they implement SIMD. In the following, these differences are discussed with an emphasis on VPs.

CPU SIMD

Almost all modern CPUs support some form of SIMD. The most recent implementation in Intel and AMD x86 CPUs is called Advanced Vector Extensions (AVX) and can process up to 512 bits of data in a single Instruction [80]. This is not a long vector in comparison to VPs, where vector lengths have reached 16384 bits. While VPs feature a fully variable vector length, most SIMD implementations support only some fixed vector lengths, such as the powers of two between one and the maximum vector length. A difference between VPs and CPUs that is no longer present in

AVX-512 and ARM SVE2 [10] is predication, where masking vectors are applied to conditional instructions. Vector chaining, on the other hand, is missing in both AVX-512 and ARM SVE2. The results of an instruction are always written back to memory. In addition, widely used CPU architectures do not currently feature the form of data pipelining present in VPs. Which means that a new instruction is executed in each cycle, imposing a higher load and latency on the instruction fetch and decode units. CPU SIMD also consumes more power and thus generates more heat than normal instructions. As a result, Intel processors, for example, usually reduce their clock frequency when using all AVX units [84].

Because of the variety of available x86-extensions, programmers have to optimize their applications specifically for each target architecture and product generation. The use of architecture-specific intrinsics is one method of accomplishing this. This workload also increases over time, since many users are still operating processors with older extensions while new ones are frequently added.

GPU SIMT

GPUs implement a different form of SIMD, Single Instruction Multiple Threads (SIMT), by combining SIMD with multithreading [58]. A modern GPU has several thousand cores, each with its own Arithmetic Logic Units (ALUs), data caches, and register files. These cores execute instructions in groups called warps. The instruction is passed to a warp by a dedicated unit with a single instruction cache, instruction decoder, and program counter. Although all cores in a warp execute the same instruction, they do so on different data sets. The purpose of SIMT is to limit the overhead of fetching and decoding instructions [81]. To achieve non-scalar execution, VPs have large cores with many compute units. The GPU approach achieves this by synchronizing many small scalar cores. Similar to VPs, GPUs are specialized for high data throughput. One drawback of the SIMT approach is poor branching performance since all threads in a group must execute all branch paths, just masked for the various conditional instructions [77]. Afanasyev et al. [3] provide a more detailed comparison between the principles of vector processing and SIMT.

3.4.3 Practical Requirements

SIMD in general and VPs in particular place special demands on compute kernels in order to achieve high performance and to use the given resources efficiently. Such a kernel must not only be parallelizable, but also vectorizable. That is, several data elements must be processable with the same instruction. The code complexity must also be minimized, especially for branching, in order to guarantee efficient execution. However, a vectorizable kernel does not guarantee good vectorization, as is visible in the case of K-Means clustering: In the algorithm of Lloyd [50], the vector length is equal to the dimensionality of the data set. This makes vectorization inefficient for small dimensionality, i.e. short vectors. This is because the initialization time of the vector pipeline exceeds the time savings from vectorization. Accordingly, the

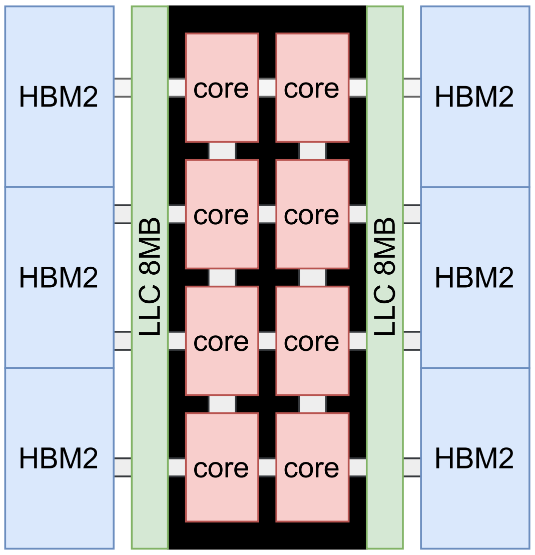


Figure 3.5: Overview of the SX-Aurora Type 10B Vector Processor.

input data must also be taken into account when evaluating the vectorizability of an algorithm.

3.4.4 NEC SX-Aurora

The SX-Aurora TSUBASA is a Vector Accelerator (also called Vector Engine (VE)) manufactured by NEC and intended to be used as a PCIe co-processor [67]. It is the successor to the SX-ACE and was presented in 2018 [102]. The architecture has its roots in the early 1980s and has been continuously developed ever since [28]. The SX-Aurora is the first VP sold as a dedicated accelerator card rather than in a complete supercomputer system. Real world applications include weather forecasting [43], sea state prediction for militaries [76], page rank algorithms [4], and other memory-intensive applications. This section provides a detailed description of the architecture, performance, system integration and other features of the VE.

Architecture & Performance

The specific SX-Aurora model used in this work is the Type 10B [65] with 8 cores clocked at 1.4GHz. Figure 3.5 gives an overview of the architecture. All cores share 16 MB of Last Level Cache (LLC). The processor is connected to six High Bandwidth Memory (HBM) modules with a capacity of 48 GB and a total bandwidth of 1.229 TByte/s. Each core has a maximum bandwidth of 409.6 GByte/s to the main memory and 3 TByte/s to the LLC. Table 3.2 provides a summary of its most important hardware properties relevant for this thesis.

Each core consists of three main components, a Scalar Processing Unit (SPU), a Vector Processing Unit (VPU), and a memory subsystem. All relevant compute capability is provided by the VPU, while the SPU is designed to provide basic

VE	Type 10B
Core Count	8
Clock Speed	1.4 GHz
Vector Length	256
Peak DP Performance	2.15 TFlop/s
Main Memory Size	48 GB
Main Mamory Bandwidth	1.229 TByte/s
LLC Size	16 MB
LLC Bandwidth	3 TByte/s

Table 3.2: The relevant hardware properties of the VE used in this thesis.

functionalities comparable to those of a CPU. A VPU hosts 64 vector registers, each with 256 entries and a width of 8 Byte, resulting in a vector length of 256 double elements. In comparison, the most recent SIMD implementations of Intel and AMD processors, e.g. AVX-512 [80], only have a vector length of 8 elements and are missing a deep pipeline. The compute capability of the VPU is provided by 32 parallel vector pipelines, each containing three Fused Multiply-Add (FMA) units, one Division/Square-Root (DIV/SQRT) unit and two ALUs. This results in a single VPU containing 96 FMA units, 32 DIV/SQRT units and 64 ALUs. All execution units are pipelined for higher efficiency. Meaning that, because there are 32 parallel vector pipelines per VPU, one vector instruction with a length of 256 is completed in 8 cycles.

The theoretical peak performance of the VE is determined by the number of cores and their number of execution units. Since each core features 96 FMA units, which achieve the highest performance compared to the other units, and each FMA unit executes two operation per cycle, the whole core can process 192 double elements per cycle. At the peak frequency of 1.4GHz, each core can achieve a peak double precision performance of 268.8 GFlop/s. The entire 8-core system can achieve 2.15 TFlop/s for double precision and 4.3 TFlop/s for single precision. In contrast, modern GPUs from NVIDIA, such as the V100 can achieve 7.066 TFlop/s for double precision and 14.13 TFlop/s for single precision [73, 92]. Besides the V100, which intended for data centers, consumer options can also outperform the VE in single precision performance. An example is the AMD Radeon RX6800 with 16.17 TFlop/s [2, 91].

An important metric for assessing the actual efficiency of hardware is machine balance. It is defined as the read and written bytes per (double precision) floating-point operation. This value has decreased in the the last decades [55], which means that fewer memory accesses are possible today than previously when reaching peak performance. Compute kernels that do not meet this requirement will therefore waste some of the potential peak performance and execute with lower efficiency. A higher machine balance means an increased efficiency. Table 3.3 lists the machine balance of various processing units, of which the SX-Aurora has the highest. This higher efficiency is also observed in practice [47].

Device	Double Precision Performance (GFlop/s)	Memory Bandwidth (GByte/s)	Machine Balance (Byte/Flop)
NEC SX-Aurora Type 10B	2150	1229	0.572
AMD Radeon RX6800	1010	512	0.507
NVIDIA V100	7066	900	0.127
Intel Xeon Platinum 8160	1612.8	128	0.079

Table 3.3: The machine balance of various processing units. Based on [65, 2, 91, 73, 92, 44].

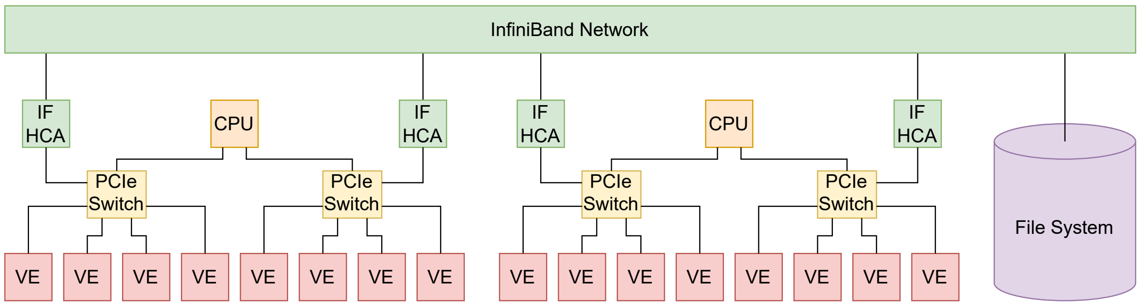


Figure 3.6: Sample cluster architecture with two nodes containing VEs that are connected by InfiniBand.

System Integration

The SX-Aurora Vector Engine is a PCIe add-in card and functions similarly to a GPU with some important differences. A VE combines vector processing with x86 architecture and programming paradigms. While a GPU has frequent data transfers between it and the host system, the VE typically only transfers data at the beginning and end of a program, or when using system calls [70]. This, along with the large main memory, avoids the PCIe bottleneck. Modern GPUs are programmed with specialized languages or APIs. Examples include CUDA [72], OpenGL [45], Vulkan [46] and DirectX [60]. In contrast, a VE executes standard C, C++ or Fortran code, which simplifies the development process. VE code is compiled by a proprietary compiler from NEC [64], which vectorizes and parallelizes the code automatically and further supports OpenMP and MPI. Due to these properties, VE programs behave similarly to any other Linux program and can use almost any Linux system call. These system calls are managed by the Vector Engine Operating System (VEOS) on the host CPU, called Vector Host (VH).

Since the SX-Aurora is designed to be used in supercomputer systems rather than personal computers or servers, it supports cross-card communication. In practice, this means that several VEs are connected via the PCIe Bus of a system. They are usually grouped together via a PCIe switch. An InfiniBand network can be used for additional connections to other nodes. Figure 3.6 shows an example of a cluster architecture. This sample architecture consists of a file system node and two identical compute nodes. These are connected by an InfiniBand network through a Host Channel Adapter (HCA). The compute nodes have eight VEs grouped using two PCIe switches connected to the CPU and the InfiniBand network.

Execution Models

VEs support several methods of executing programs [29, 66]. The first is native compilation and execution, which is done with the NEC compiler. Native execution offers the advantage of a coherent code and thus improved readability. Disadvantages are the possibly low performance of scalar code sections, I/O functions, and system calls. To address these disadvantages, other execution models are also supported.

An alternative to native code is offloading from the CPU to the accelerator. In this case, only the compute kernels that are specifically suited for VPs would be executed on the SX-Aurora. All other code remains on the main CPU of the system, which might result in runtime savings. Offloading code requires an interface to the VE in the form of either OpenMP offloading or VEDA (Vector Engine Driver API) [68]. OpenMP offloading is already implemented for GPUs [8, 22]. On the VE, it is currently only supported with an extension to LLVM, which is presented and evaluated by Cramer et al. [17, 18]. VEDA, on the other hand, is usable in common compilers as an includable library. It offers a variety of commands that provide greater control and flexibility than OpenMP offloading. It is also possible to offload parts of a program from the VE to the vector host. This is especially useful if host functions like system calls are rarely used.

This thesis uses both native compilation and VEDA to port validation benchmarks, while the presented code generation is entirely based on VEDA.

3.5 Benchmarks

In this thesis, several benchmarks are used, some of which are based on algorithms that are used in real-world applications. They possess different performance bounds, parallelizability, and vectorizability. The latter is particularly important with regard to the investigated vector architecture

The first algorithm is a K-Means clustering (K-Means) that is based on Lloyd's algorithm [50], and part of the Rodinia benchmark suite [14, 15]. It has a high degree of data parallelism, while still being memory bound. The algorithm partitions the data points on an n-dimensional grid such that each data point is assigned to the nearest of k clusters. These clusters are moved to the center of their partitioning.

This procedure is repeated in several iterations to find an optimal clustering. The vector length that can be achieved is equal to the dimensions of the grid and the data points.

The second used algorithm is K-Nearest neighbors (NN). Like K-Means, it can also be effectively parallelized and is memory bound. The used implementation is also part of the Rodinia benchmark suite. The algorithm finds the k nearest neighbors of an input point in a dataset. Unlike K-Means, this does not require multiple iterations but only needs to be performed once. Vectorizability in this algorithm does not depend on the dimensionality of the dataset, but only on the actual code design. In this implementation, a high degree of vectorization is possible.

A Jacobi solver is used, to cover cache bound algorithms. This algorithm changes array elements in a certain pattern, which is called a stencil. The exact pattern is based on the Jacobi method, which approximately solves a system of equations. Nested loops and branches permit only a partial vectorization, but that still results in significant performance improvements.

A new algorithm is implemented in order to investigate compute bound problems. It is a linear shift of a dataset by adding a fixed value to each data element. This shift can be performed in an arbitrary number of iterations to allow for a runtime that can be accurately examined. The algorithm is highly parallelizable over the iterations. A high degree of vectorization of the shift operations is possible.

The exact selection of these algorithms is based on various aspects. They have to cover the relevant cases in the Roofline and developed performance model. That is, compute, main memory, and cache boundedness. The code complexity also plays a role in the selection. It has to be relatively low in order to guarantee portability to VEs as well as ease of investigation within the time frame of this work. Table 3.4 gives an overview of the data used for the Roofline model.

Algorithm	Boundness	Loads + Stores	Flops	Operational Intensity I
K-Means clustering	Memory	23	8	0.0435
K-Nearest neighbors	Memory	3	6	0.25
Jacobi solver	Cache	9	6	$0.083\overline{3}$

Table 3.4: Operational details of the memory and cache bound benchmark algorithms.

The STREAM benchmark [57] is used for additional validation. It measures the real achievable memory bandwidth of a hardware configuration with the help of four different kernels. By processing large arrays, these kernels minimize cache effects, which would distort the performance profile of the main memory. The original benchmark is modified to allow control of the vector length. This allows the relationship between vector length and memory bandwidth to be investigated. Thus, the assumptions made in the developed model can be analyzed.

4 Performance Model

Extending the Parallel Pattern Language (PPL) for Vector Engines (VEs) requires foundational work. The first step is the development of a performance model in order to predict the performance of an algorithm on VEs. The main constraint for this model in the context of the PPL is a static usability, based on hardware and code properties. A high accuracy for a wide range of algorithms must also be achieved. Section 4.1 presents a newly developed model that meets these requirements. Especially the accuracy is validated in Section 4.2 using the benchmarks introduced in Section 3.5. Section 4.3 then investigates how the model can be integrated into the PPL. A discussion on the results and proposals of this chapter follows in Section 4.4.

4.1 Proposed Model

This section introduces the developed performance model, which is composed of two newly developed extensions to the cache-aware Roofline model [100, 42]. Using it as the base model has a number of advantages that are outlined in Section 4.1.1. Section 4.1.2 and Section 4.1.3 then present the developed extensions.

4.1.1 Concept

Modeling the performance of algorithms on VEs does not require the development of a new model. Rather, existing models can be used and extended to cover the special characteristics and behaviors of VEs. There are two main requirements for such a model:

1. Static computability, based on code and hardware characteristics.
2. Accurate performance prediction for algorithms with different types of performance limitations.

Two models that meet these requirements are the Roofline [100] and ECM [94] models.

While both fulfill the aforementioned requirements, they achieve them with different methods. The cache-aware Roofline model [42] is chosen as the base model due to the following reasons: First, the performance modeling using both code and hardware properties only requires calculation with a single, simple formula (see Equation (3.1) and Table 3.1). Meanwhile, the ECM model requires the calculation of overlap and the use of more hardware properties, which might not be known for a given architecture. For instance, the required cycles per data transfer, especially between the individual memory levels, are not entirely known for the proprietary SX-Aurora architecture. The second reason is that the Roofline formula can be quickly extended by adding new code or hardware factors. The relative complexity of the ECM model inhibits such extensibility. The third advantage of the Roofline model is its ease of visualization, which allows a programmer to quickly assess how much performance potential a code has. By using the cache-aware Roofline model, the entire architecture of the SX-Aurora is covered. Memory, cache and compute bound codes can be modeled.

4.1.2 Vector Length

The actual vector length in a computational kernel is crucial for the achievable performance on Vector Processors (VPs). For example, with a real vector length of 128 elements, only half of the available performance can be achieved on the SX-Aurora. The vector length should be integrated as a factor in the performance model to model this code characteristic. No version of the Roofline model published so far is specialized for VPs and considers the vector length as a key performance factor.

The vector length

$$V = \frac{v_{real}}{v_{max}} \tag{4.1}$$

is thus integrated into the Roofline Model as an additional factor, where $0 < v_{real} < v_{max}$ holds. Here v_{real} is the real vector length within an algorithm and v_{max} is the maximum vector length supported by the architecture. On the SX-Aurora $v_{max} := 256$ applies.

By adding the vectorization factor, the Roofline formula takes the form of

$$\pi = \min(Ppeak, I \cdot \beta) \cdot V. \tag{4.2}$$

The original components of the formula ($Ppeak$, I, and β) remain in their original form. The formula applies to both β_{HBM}, the main memory bandwidth, and β_{LLC}, the cache bandwidth. The multiplication with V ensures that the performance modeling for compute, memory, and cache bound codes depends on the vector length. The resulting performance scaling can be depicted within the established Roofline visualization, for which Section 3.1 provides an example. Figure 4.1 shows this exemplarily for the real vector lengths 256, 128, 64 and 1, where 256 is the upper and 1 the lower performance bound. However, any value between 1 and 256 can be used. The use of β_{LLC} is omitted for visual clarity. A new roof arises for each vector length.

The described extension maintains the Roofline model as a statically computable performance model. The information about the real vector length can be obtained with different (static) methods. For example, in K-Means clustering (K-Means), the vector length depends on the dimensionality of the dataset, while in other algorithms,

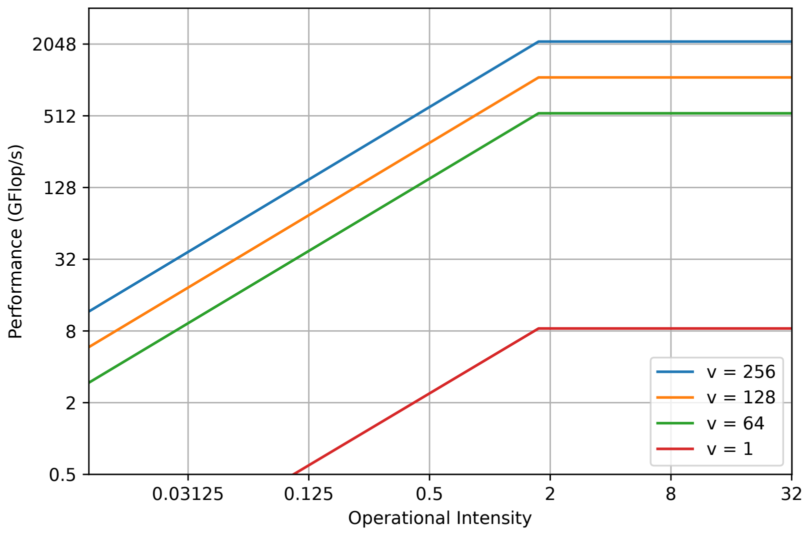


Figure 4.1: Performance roofs for different vector lengths on the SX-Aurora.

an analysis of the compute kernel is required. Consequently, static computation is possible in the context of the PPL toolchain.

4.1.3 Effective Memory Bandwidth

The initial validation of the newly added vectorization factor finds that it is not sufficiently accurate in modeling the behavior of the SX-Aurora in every case. In these measurements, some algorithms show higher performance than predicted by the model. This behavior is directly related to the vector length. For long vectors, the measured values are close to those predicted by the model. In contrast, for short vectors, the achieved performance is noticeably higher than predicted. To explain this discrepancy, the following hypothesis is put forward: *The shorter the vector length, the higher the memory bandwidth achievable due to caching and prefetching effects.*. The validity of this hypothesis cannot be investigated further due to the proprietary nature of the SX architecture and the lack of response to contact attempts with NEC. However, a second extension based on the hypothesis is added to increase the accuracy of the vector length extended Roofline model on the SX-Aurora.

This is achieved by addressing the memory bandwidth. β_{HBM} is to be scaled non-linearly based on the vector length. The extension does not apply to β_{LLC}, since the mentioned discrepancies are not observed for cache-bound algorithms. This might be because cache effects are more pronounced when the main memory is used. The modified Roofline formula should still be easy to use, i.e. it should allow a straightforward performance calculation without requiring new variables. The exact formula is acquired by curve fitting to the benchmark performance. As a result, a

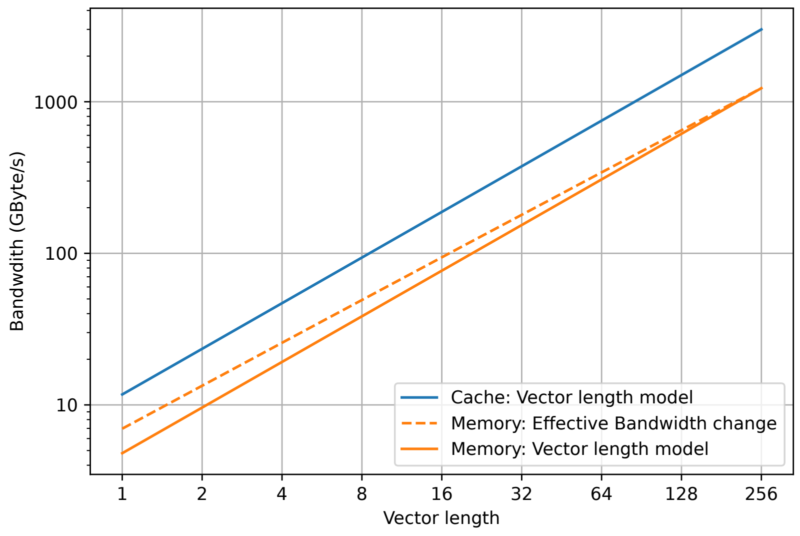


Figure 4.2: The memory bandwidths of an SX-Aurora as a function of vector length as calculated by the developed performance model.

new scaling factor

$$X = 100 \cdot \ln(\frac{1}{V}) \tag{4.3}$$

is introduced. This is added to the existing memory bandwidth. The Roofline formula for memory-bound algorithms thus takes the form of

$$\pi = \min(Ppeak, I \cdot (\beta_{HBM} + X)) \cdot V. \tag{4.4}$$

This achieves a non-linear scaling of the memory bandwidth. Figure 4.2 provides a visualization of the resulting differences in achievable bandwidth on an SX-Aurora as a function of the vector length. The cache bandwidth is represented by the blue line. The dotted orange line is the non-linearly scaled bandwidth, while the solid orange line is the originally assumed bandwidth. The distance between the two lines decreases as the vector length increases.

This second extension still allows a static performance prediction. In contrast to the first extension, no further analysis of the code is required. Only the distinction whether a code is bound by main memory or cache bandwidth becomes important. However, this is already necessary in the original cache-aware model.

4.2 Validation

An important requirement for any performance model is its accuracy when applied to real algorithms. Although such models have a simplified view of hardware and code properties, they capture the most important factors influencing performance.

The previously presented model is examined in this section in terms of its accuracy. For this purpose, the used setup and methodology are presented in Section 4.2.1. This is followed by the presentation and analysis of the results in Section 4.2.2.

4.2.1 Setup and Methodology

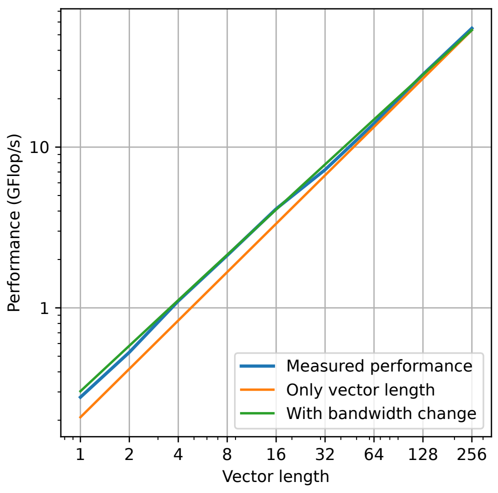
The extended model is validated using the benchmarks presented in Section 3.5. These cover all relevant cases, i.e. compute, memory, and cache boundness. The mean runtime of the compute kernels of the algorithms is measured over ten executions. It is converted into the performance in GFlop/s using the arithmetic and memory operation count from Table 3.4. An exception here is the STREAM benchmark, for which the memory bandwidth is calculated. Since the vector length is central to the extended model, the performance of each algorithm is examined for vector lengths between one and 256 elements. In order to reduce the time required for data collection while maintaining the same validity, the vector length only assumes powers of two in this range. The exact measurement results can be found in Tables A.1 and A.2.

In K-Means, the positions for five clusters are searched. In this algorithm, the vector length is equal to the dimensionality of the dataset. For this reason, a separate dataset is used for each vector length. To limit the runtime, the number of data points depends on the vector length. For a vector length of one or two, each dataset contains ten million elements. For all other lengths, it contains one million elements. In K-Nearest neighbors (NN), the $k = 50$ nearest neighbors are searched in a data set containing 32768 points. Only the kernel calculating the distances is considered in the performance measurement. The Jacobi solver is executed on a 32768 by 32786 matrix in 50 iterations. In the compute bound linear shift kernel, the data set consists of six arrays, each with a length of 256, which are used in 50 billion iterations. The STREAM benchmark uses a data set with 100 million elements. Consequently, a single double array of this length consumes 800 MByte, which prevents cache effects.

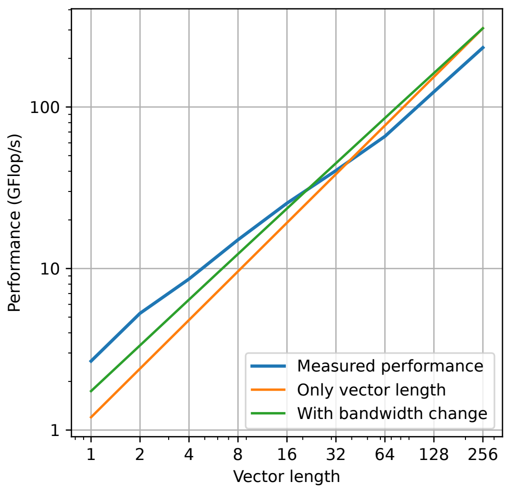
All algorithms are executed on the nca01 node of the RWTH cluster. On this node, only the Type 10B Vector Engine is relevant for the measurements. Sections that are executed on the CPU, such as file I/O, do not affect the runtime measurements. The code is compiled with the NEC compiler [69] version 3.5.1. Correctness of the results is confirmed by a comparison with the results of the original code for CPUs.

4.2.2 Results

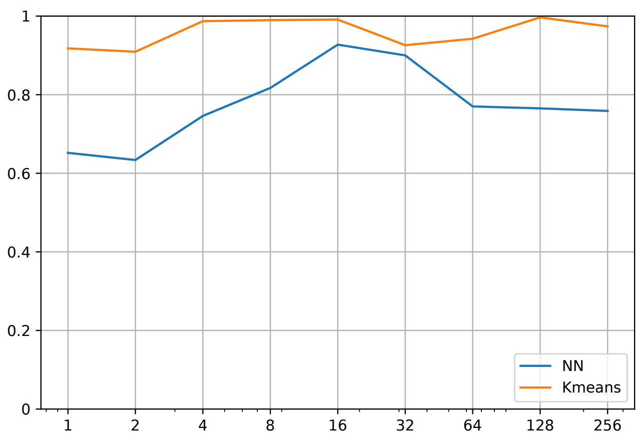
The two investigated memory bound algorithms show distinct performance profiles. The average runtime standard deviation is 0.000015% for K-Means and 0.00347% for NN. The performance of K-Means is closer to the model than that of NN. As seen in Figure 4.3a, K-Means is very close to the scaled bandwidth model for all vector lengths. In contrast, the performance observed in NN (Figure 4.3b) is higher than predicted by the scaled model for short vectors, up to length 16. This might be



(a) Measured performance of K-Means in relation to the performance predicted by the model extensions.



(b) Measured performance of NN in relation to the performance predicted by the model extensions.

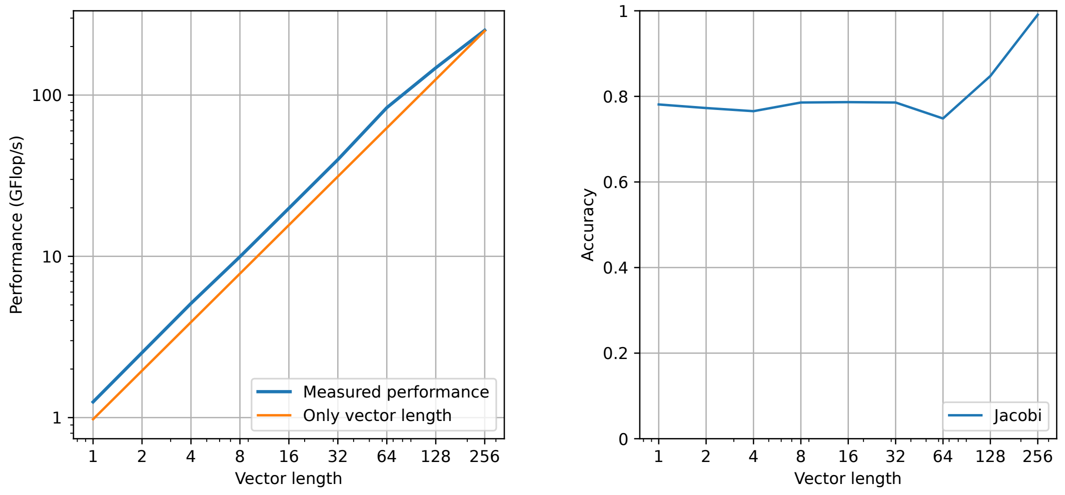


(c) Accuracy of the model for NN and K-Means over the vector lengths.

Figure 4.3: The model in relation to the performance of the memory bound benchmarks.

caused by unexpectedly strong cache effects. For all longer vectors, the performance is lower than predicted.

The accuracy of the model across all vector lengths and both algorithms is 86.7%. A breakdown across the individual vector lengths can be found in Figure 4.3c. It also shows that the performance of K-Means is closer to the predicted performance, with an average accuracy of 95.9%, than that of NN with 77.5%. Figure 4.3c also depicts how the measured and predicted performance of NN converge up to vector



(a) Measured performance of the Jacobi solver in relation to the predicted performance of the performance model extensions.

(b) Accuracy of the performance model for the Jacobi solver over the vector lengths.

Figure 4.4: The performance model predictions in relation to the observed performance of the cache bound benchmarks.

length 16 and then diverge again beyond that. K-Means remains much closer to the model over all vector lengths. The performance measured by the Jacobi solver is higher than the performance predictions of the model, which results in an average accuracy of 77.3%. Figure 4.4a visualizes this fact. A more detailed overview on the accuracy is provided by Figure 4.4b. Both figures show that the predicted and observed performance almost match at the maximum vector length of 256. This behavior can be explained by register effects that might increase the performance for shorter vectors. The average standard deviation is 0.00135%. In the case of the compute bound algorithm, the model has an average accuracy of 98.7%. An overview of the accuracy across the different vector lengths is provided by Figure 4.5. The average standard deviation is 0.017%.

The STREAM benchmark is used to investigate the hypothesis of the non-linearly scaling memory bandwidth. Namely, whether it can actually be observed in practice. The measurement results are shown in Figure 4.6, where a large discrepancy between the measured and predicted values can be seen. For example, the Copy kernel at a vector length of 32 elements achieves a bandwidth of 498.58 GByte/s. The bandwidth-scaled model predicts 217.52 GByte/s, which is less than half of the observed value. A specific reason for these remarkable differences could not be identified. The benchmark is not cache bound, since all used arrays are too large to fit into the cache. This is proven by the profiling tools of the SX-Aurora which show a cache hit ratio of less than 0.1%.

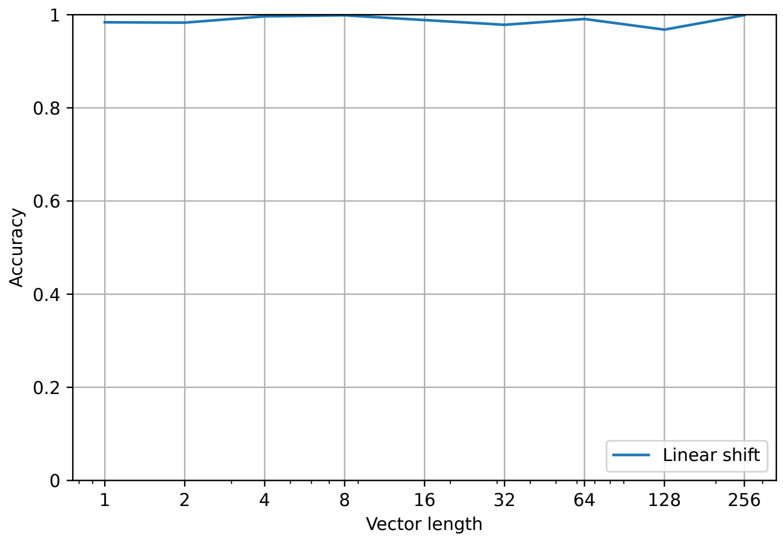


Figure 4.5: Accuracy of the performance model for the compute bound benchmark over the vector lengths.

4.3 PPL Integration

The Roofline model, including its cache-aware form, is a static performance model. The addition of the vector length does not change this. It can, for example, be determined by the used data set, a factor which is statically known in the PPL, or with other code properties like statically defined loop ranges. Although it is possible to determine the actual vector length at runtime, this should be done by static code analysis in the context of PPL.

Statically finding the vector length in a given code section is a two-step process. The first step is to determine the loops that will be vectorized by the NEC compiler. However, not only individual loops but also nested loops can be vectorized. There are two cases of nested loops. The first are the tightly nested loops where multiple loops can be merged together to form a single vectorizable loop. Tightly nested loops only have instructions in the innermost loop. The other loops that are not tightly nested might not be vectorized together. Instead, only the innermost loop may be vectorizable. An exerpt from the kernel of an n-dimensional K-Means clustering in Listing 4.1 serves as an example. Here, the two loops are not tightly nested and only the innermost, that iterates over *nfeatures*, can be vectorized. Thus, in this case the actual vector length is equal to $\min(256, \mathit{nfeatures})$. It should be noted that *nfeatures* is the dimensionality of the dataset and therefore statically available.

Other factors that can inhibit vectorization of any loop are **data dependencies**, **non-standard function calls**, **branches** (except in the case of min, max, compression, expansion, and search), **data transfers**, **I/O operations**, and **array initializations**. These general restrictions can be used to find the loop that will

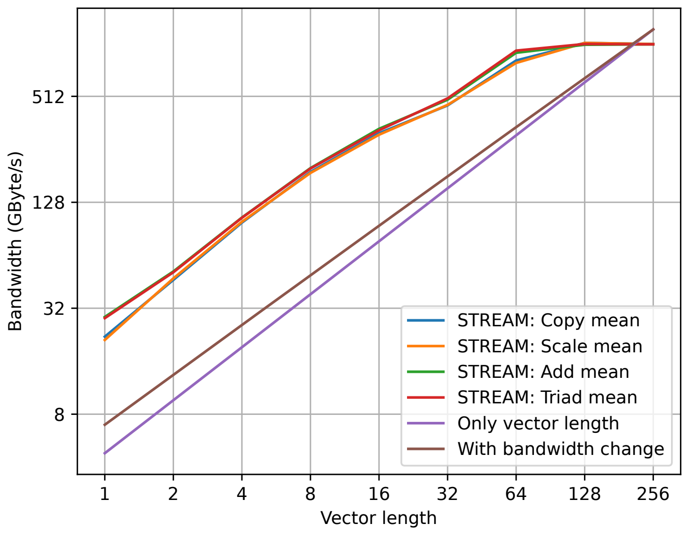


Figure 4.6: The STREAM benchmark results in comparison to the bandwidth predicted by the performance model for different vector lengths on the SX-Aurora Type 10B.

Listing 4.1: An exerpt from K-Means clustering that assigns a datapoints to clusters.

```
for (int j = 0; j < nclusters; ++j) {
    float dist = 0.0;
    for (int k = 0; k < nfeatures; ++k){
        dist += (feature[i][k]-clusters[j][k]) * (feature[i][k]-
            clusters[j][k]);
    }
    if (dist < min_dist) {
        min_dist = dist;
        index = j;
    }
}
```

be vectorized. The second step of the analysis then finds the vector length inside this loop. After performing this analysis, the real vector length is determined and can be used in the model. This further means that integrating the model with this extension into the PPL is possible.

The second extension also preserves the performance model as a static analysis. The importance of the distinction between main memory and cache bound algorithms increases. This distinction must be made statically in the PPL and is based on the analysis of the variables used in an algorithm. If the memory requirements are smaller than the cache size, then the algorithm is cache bound. This condition can be used in the PPL, since both variable types and array sizes are defined statically, and thus the required memory space can be calculated. In the case of the SX-Aurora, these variables must require less than 16 MB of memory in total. This limit can also be derived in the PPL from the cache size that is defined in the Hardware Language (HL). The model can therefore also be used with this extension in the PPL.

It is observed that some algorithms behave cache bound, although they should not be according to the common definition. An example is the Jacobi solver used in the validation where a variable does not fit in the cache, but a cache bound performance is observed. This variable can assume any size without affecting the performance. For this reason, the following proposal is made for the PPL to distinguish between cache and main memory boundness: *An algorithm is cache bound if the memory requirements of all variables used in the compute kernel is less than the cache size. A single variable may be excluded from this calculation.* This proposal is a simplistic solution to a problem that requires a more complex analysis of cache bounds. This simplicity may lead to incorrect decisions for some algorithms. But especially for the given example, does it model the performance correctly.

4.4 Discussion

Two main requirements were set for the model in Section 4.1.1. First, it must remain a static model to enable the integration into the PPL. Second, it should achieve a high accuracy for algorithms with different performance constraints. Both conditions are fulfilled by the developed model.

Proposals for the integration of the model into the PPL are made in Section 4.3. The accuracy of the proposed code analysis in determining the vectorization can only be approximated. The most important factor is the thoroughness of the implementation. However, the main constraints and influences on vectorization are considered in the proposed approach. The suggestion at the end of the Section to ignore a variable in the distinction between memory and cache boudness can be criticized. It draws general conclusions from the observation of a single algorithm. A more detailed analysis would be necessary to determine the underlying causes, which was not possible in the scope of this work. It nevertheless models the Jacobi solver accurately enough to serve as a general performance indicator, and can therefore be considered an acceptable interim solution.

The validation shows that the second condition is also satisfied. For all algorithms, the performance prediction achieves an accuracy comparable to other models [40, 54]. The accuracy for K-Means and the linear shift is noteworthy with 95.9% and 98.7%, respectively. For NN, a lower accuracy of 77.5% is achieved. The performance for longer vectors is lower than expected, suggesting the influence of additional factors that are not considered in the model. One possibility is the interaction of the different compute units in a Vector Processing Unit (VPU). The NN algorithm uses both the Fused Multiply-Add (FMA) and the Division/Square-Root (DIV/SQRT) units. This might lead to a bottleneck since there are 3 FMA units and only one DIV/SQRT unit per VPU. The mismatch between the number of FMA units, Arithmetic Logic Units (ALUs), and DIV/SQRT units and their performance is not considered in the model. The dependence of the performance on the used computing unit could be integrated into the model to improve the accuracy. This increases the complexity of the model and thus the complexity of the required code analysis. The investigation of the cache bound algorithm shows an accuracy similar to NN with 77.3%. The performance is higher than predicted for all vector lengths except 256. This indicates unexpected side effects, such as variables remaining in the registers. The fact that a variable does not fit in the cache, but the algorithm behaves cache bound, also supports this assumption.

The STREAM benchmark measures a significantly higher memory bandwidth than assumed in the model, which challenges its validity. A possible explanation is the simplicity of the four kernels of the benchmark, in the sense that no bottlenecks occur in the pipeline and that prefetching may have been used to achieve the observed bandwidth. This simplicity could also ensure that the memory controllers act partially independent of the actual vector length and thus achieve a higher performance. Since the architecture is proprietary and attempts to contact NEC via the SX-Aurora forum remain unanswered, a final answer cannot be given. However, the discrepancy observed in the STREAM benchmark only occurs there. More complex algorithms that are closer to real-world applications, like the other benchmarks, do not exhibit it.

For benchmarks that are close to real-world algorithms, the model shows an accuracy sufficient to act as a general performance indicator. In some cases, it models the performance very accurately. Since the STREAM benchmark is not close to reality, the model can be classified as valid. Section 4.3 shows that an integration of the model into the PPL can be achieved. This requires the development of new code analysis features that determine the vectorized code section and the actual vector length. The accuracy of the proposed analysis method is limited by the fact that it is performed statically. It also heavily depends on the exact implementation since many aspects of the compiler have to be taken into account. The proposed method can be improved in the future. For example, the vectorizer from compilers like LLVM or even the NEC compiler can be used. This requires developing a suitable interface or analyzing the assembler code. The NEC compiler informs the user on which code sections were parallelized and vectorized. However, this method would constitute a major departure from the current principles of the PPL, which only analyzes the frontend code. Instead, the relevant code would have to be generated, compiled and then analyzed ahead of the optimization step. This would not only greatly increase the complexity, but presumably also the runtime of the toolchain. The approach proposed in this thesis thus represents a sound balance between accuracy and complexity.

The usability of the model is not limited to VEs, because by considering the vector length, VPs can be modeled in general. The second extension, i.e. the modified bandwidth scaling, is, however, specifically tailored to the available VE. Only the first extension should be used when applying the model to other VPs. If similar observations can be made for the bandwidth, the second extension may also be adapted.

5 Code Generation

This chapter addresses the generation of code for Vector Engines (VEs) in the context of the Parallel Pattern Language (PPL). For this purpose, the various components of the toolchain need to be extended. First, Section 5.1 defines the overall goals of the extension. Section 5.2 then further specifies these in the context of a requirement analysis. A possible realization of the code generator extension is described in Section 5.3. A considerable part of the requirements is implemented in a proof-of-concept. The resulting changes are presented in Section 5.4 and subsequently evaluated in Section 5.5. Finally, the results of this chapter are discussed in Section 5.6.

5.1 Goals

The overall goal is to automatically generate globally optimized code for VEs. For this purpose the PPL shall be extended. The exact subgoals of this extension are based on both the already existing features of the PPL and on special functionalities that VEs provide. Thus, the following goals are derived:

- Generation of correct and performant code for one or more VEs in a heterogeneous system.
- Code that is offloaded to VEs must have at least the already existing GPU offloading features of the PPL:
  - Synchronization of an arbitrary subset of devices with each other.
  - Data transfers between an arbitrary pair of devices.
- The optimizer should make use of the previously presented performance model.
- Utilization of the special programming and execution model of VEs.

With the realization of these goals, an extended PPL will be able to generate code for VEs and utilize them in an optimal way. Having at least the GPU offloading functionalities ensures that interdependent computations do not cause data races and further allows flexible data transfers when necessary. The integration of the performance model ensures that VEs are only used if they can provide the best performance among the given compute units, considering the necessary data transfers. VEs may have a different execution model than CPUs and GPUs (see Section 3.4.4), which must be properly taken into account in the code generation process.

5.2 Requirements

In this section, individual components of the PPL are examined for required changes with respect to the previously defined goals. These changes are defined as specific requirements and possibilities for the implementation are discussed. Reasons why some components do not need to be changed are also given.

5.2.1 Frontend Language

The PPL frontend language already offers extensive possibilities for implementing parallel algorithms through the use of parallel patterns. However, the frontend language and its translation into standard C++ code has shortcomings with respect to the execution model of VEs. This becomes apparent when considering the realization of parallel patterns within iterator loops. This is illustrated by the example in Listing 5.1. It shows such an iterator loop containing four calls to the same map pattern with different variables.

Listing 5.1: The iterator loop of the Jacobi solver.

```
for var Int i = 0; i < 25; i++ {
    x1 = iterate<<<>>>(A,b1,x0)
    x0 = iterate<<<>>>(A,b1,x1)

    y1 = iterate<<<>>>(A,b2,y0)
    y0 = iterate<<<>>>(A,b2,y1)
}
```

When applied to the example, the current implementation of the code generator removes the loop entirely and instead generates 100 individual offloaded GPU kernels with their associated data transfers. This is due to the constraints imposed by linear programs within the optimizer. Although this approach is not ideal for GPUs, it corresponds in part to the CUDA execution model. In contrast to the model of VEs, it performs considerably more data transfers. As noted in Section 3.4.4, VEs use as few data transfers as possible. An offloaded kernel should run as long as possible and data transfers should only occur at the start and the end of the execution. Ideally, the presented code example would be completely offloaded such that the entire iteration process would take place on the VE. However, the currently generated code does not do this and thus uses VEs inefficiently. The following four different approaches present solutions for this problem:

1. Keeping the current methodology.
2. Introduction of a dedicated offload function to the frontend language and a node to the intermediate representation.
3. Introduction of a special iterator loop with a corresponding node that shall be offloaded but not be executed in parallel.

4. Automatic detection of iterator loops that would not be parallelized and their subsequent offloading.

It should also be noted that these changes are only required for VEs. A conditional code processing and optimization is therefore required. The proposed approaches are now discussed by analyzing their advantages and disadvantages.

Approach 1

Keeping the current methodology offers the advantage of reduced programming effort during the PPL extension. Furthermore, in contrast to approaches 2 and 3, no algorithms that are already implemented in the frontend language need to be modified. However as previously mentioned, this approach has disadvantages with respect to VEs. Optimal performance of a program cannot be achieved in this way due to the different execution model and the inefficiencies introduced by the PCIe bottleneck and driver overheads.

Approach 2

With the introduction of a dedicated offload function and accompanying node in the Abstract Pattern Tree (APT) and Abstract Mapping Tree (AMT), the programmer can explicitly request the use of a VE, GPU or other offloading device. However, this does not fit the programming model of the PPL, in which the programmer does not have to choose the execution device. Instead, this choice is made automatically by the optimizer, which ideally finds a mapping that maximizes performance. In addition, not all code is optimal for both VEs and GPU at the same time and could thus possibly decrease performance if it is offloaded to the wrong device. This function would also require the definition of both the input and output variables. However, this requires further changes to the code processing, since currently only a single variable can be used as output.

Approach 3

The drawbacks from the previous approaches can be overcome by introducing iterator loops. When translating the code into the APT, the iterator loop, which is initially part of a sequential node, can be split off from it into an iterator node. For this purpose, both the APT and the AMT would have to be extended. The optimizer can completely offload this node to it, if a VE in the cluster is the optimal choice. However, the currently implemented methodology can be kept if a GPU or CPU is chosen. In addition, the programmer does not have to define on which device a code section will be executed and can leave this decision to the toolchain. Another advantage is the relatively simple integration into the PPL. However, an adaptation of already existing PPL programs is necessary. Listing 5.2 applies this approach to the previous example.

Listing 5.2: The Jacobi example with an iterator loop.

```
iteratorFor var Int i = 0; i < 25; i++ {
    x1 = iterate<<<>>>(A,b1,x0)
    x0 = iterate<<<>>>(A,b1,x1)

    y1 = iterate<<<>>>(A,b2,y0)
    y0 = iterate<<<>>>(A,b2,y1)
}
```

Approach 4

The automatic detection and offloading of an iterator loop is an advanced form of approach 3. It would require similar extensions of the APT and AMT and would offer the same advantages. It would also still be neccessary to combine different calls to patterns into a single node that would be offloaded to VEs. Changes to existing programs in the frontend language would not be required. The integration into the toolchain, especially in the form of the recognition of an iterator loop, presents a greater challenge than approach 3. But this is mitigated by the fact that the toolchain already analyzes the code for loops that can be unrolled. This feature can be extended to recognize iterator loops. In total, this approach requires a significant implementation effort.

Proposal

In conclusion, approach 3 offers the greatest advantages with the fewest disadvantages. By introducing an iterator loop and accompanying node, the current programming model of the PPL is retained while a flexible assignment to devices is made possible. The introduction of the new node must not only be done in the APT, but also in the AMT. The required changes to already ported algorithms does not represent a large burden. Approach 4 can be a future goal, despite requiring significantly greater programming effort, as no changes to existing programs would be needed.

5.2.2 Hardware Language and Cluster Model

The Hardware Language (HL) offers a flexible way to represent different clusters and devices by using JSON. Devices such as CPUs and GPUs are modeled within two files. The first one models the device with its higher-level technical data, e.g. the bandwidth and size of the main memory. The first file contains a reference to the second that represents a cache group within the device. This pattern is kept for VEs. The example in Listing 5.3 shows how an SX-Aurora Type 10B can be represented as a device using the technical data from Section 3.4.4. A VE only requires a single cache group since the Last Level Cache (LLC) is shared by all 8 cores. This cache group is modeled as shown in Listing 5.4. It should be noted that the latency of the main memory and cache of the SX-Aurora are unknown. Due to the similar architecture of VEs to CPUs and GPUs, the attributes should be set to comparable values, as it is done in the examples. All relevant technical data of VEs can be mapped in the HL without structural changes. Consequently, no significant changes are required to the cluster model, which is generated from the HL without any transformations.

Listing 5.3: Modeling a VE as a device in the HL.

```json
{
    "type": "VE",
    "latency": "42.38",
    "bandwidth": "1229000.0",
    "max-bandwidth": "1229000.0",
    "size": "48000.0",
    "cache-group": [
        {
            "identifier": "1",
            "template": "ve_group.json"
        }
    ]
}
```

Listing 5.4: Modeling the cache group of a VE in the HL.

```json
{
    "cores": "8",
    "frequency": "1400",
    "arithmetic-units": "96",
    "vectorization": "256",
    "caches": [
        {
            "latency": "20.95",
            "bandwidth": "300000.0",
            "size": "16"
        }
    ]
}
```

5.2.3 Optimizer

Using the current performance model of the PPL for VEs is not an optimal approach. To effectively consider VEs and their performance, the previously defined performance model must be integrated into the optimizer. To this end, two new code analysis features must be develoepd. The first is a vector length estimator, which receives a code segment to be analyzed as an input, finds the vectorized portion, and returns the estimated vector length. The second determines whether a compute kernel is main memory or cache bound. An implementation can reuse the data size analysis from the network cost estimator. The exact methodologies for both analyses are defined in Section 4.3.

Section 5.2.1 introduces new iterator nodes in the APT and AMT, which the optimizer must be able to recognize. It must also be able to either move the entire iterator node to a VE, leave it on the CPU, or move the associated parallel pattern calls to GPUs. It therefore has to distinguish between all three cases and calculate which is the optimal choice. It also has to take into account that in the case of VEs, usually only two data transfers are required, none in the case of CPUs, but many in the case of GPUs.

5.2.4 Abstract Mapping Tree

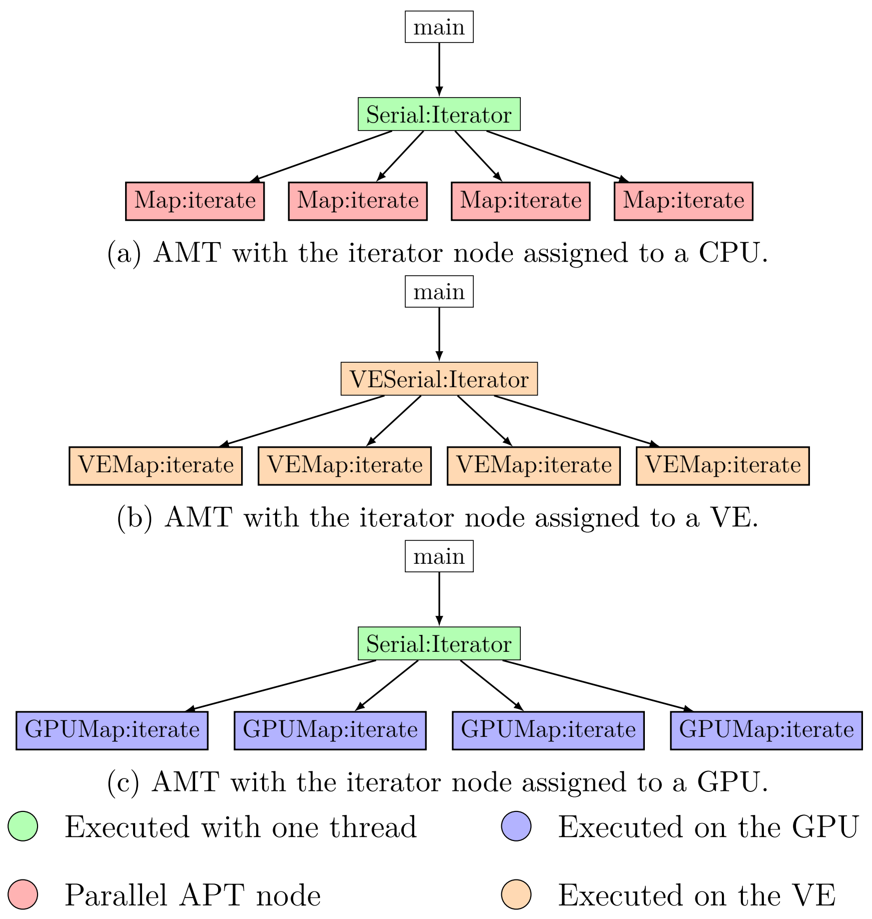
The AMT is the main abstraction level between the input code, provided cluster, and the generated code. Unlike the APT, it also models the mapping of parallel patterns to execution devices. The nodes of the AMT can be divided into four categories. **Serial**, **function**, **call**, and **data control** nodes. To ensure a correct mapping of patterns to VEs and subsequent code generation, new nodes have to be introduced in these categories. Especially the call and data control nodes must be extended.

There are currently five types of parallel **call** nodes. *Parallel* calls, *serialized parallel* calls, *reduction parallel* calls, and *fused parallel* calls, which are all executed on the CPU. The fifth type of nodes are the *GPU parallel* calls. As proposed by Schmitz [82], a new kind of node should be introduced to model parallel calls on other devices, in this case VEs. This node will have close similarities to the GPU parallel calls and would be called a *VE parallel* call. However, due to the differences in the architecture of GPUs and VEs, it is no longer necessary to merge multiple parallel patterns into one.

There are two types of **data control** nodes. Only the *data movement* nodes, which define a set of source and destination placements, are relevant for VEs. Since the current implementation only assumes GPUs as offloading devices, it must be extended to also consider VEs. An implementation can be similar to the current for GPUs, since no structural differences need to be taken into account. As a result of this and the previous extension, a successful mapping to VE will be possible.

However, both the changes to the call and the data control categories are not sufficient with respect to the VE execution model. The proposal made in Section 5.2.1, in the form of an iterator loop node should also be integrated into the AMT. This node can be categorized as a **serial** node and has parallel functions or calls as children. With this node, either the whole iterator or only its children are mapped to devices. An illustration of the different mapping possibilities of this node is given by Figure 5.1. It is based on the previously presented code example in Listing 5.2.

The implementation of the AMT in the PPL is not only limited to the tree structure but each node also includes a visitor pattern. This leads to an extension of the nodes with interfaces to simplify the traversal for visitors. This especially eases the code generation, which is realized by such a visitor. These interfaces must also



(a) AMT with the iterator node assigned to a CPU.

(b) AMT with the iterator node assigned to a VE.

(c) AMT with the iterator node assigned to a GPU.

Figure 5.1: AMT definitions based on the code in Listing 5.2 for CPUs, GPUs, and VEs.

be defined in the newly added nodes for VEs and the iterator. By meeting all of these requirements, all previously defined goals are realized, effectively extending a core component of the PPL for VEs.

5.3 Code Generator Component

The code generator implemented by Schmitz [82] holds a key position in the extension of the PPL for VEs. With the PPLs source-to-source approach, C++ code is generated instead of binary machine code. Due to the complexity and size of the generator implementation as well as the special requirements of VEs, a number of aspects have to be considered. These are investigated and discussed in this section, especially with regard to previously proposed goals and changes.

5.3.1 Programming and Execution Model

As described in Section 3.4.4, the execution model of VEs is considerably different from that of GPUs. It is partially covered by the newly introduced iterator loop and node, but changes to the code generator are still required. In the VE programming model, programs can either be natively compiled or partially offloaded using VEDA [68]. The PPL implementation should use the latter approach, since mixed execution of CPU, GPU, and VE code would otherwise not be possible. In addition, VEDA is very similar to CUDA, simplifying the implementation process. As a result of this decision, a number of changes to the generator are required.

In the VEDA programming model, two file types are used. The first is executed on the host device, is responsible for data transfers and device management, and loads and uses the VEDA library. This file is compiled with a standard C/C++ compiler. The second file type is executed on the device itself, contains the relevant code sections and is compiled by the NEC compiler. Here the main difference between GPU and VE code can be observed. Namely, a larger code section can be executed efficiently on VEs than on GPUs. This is especially important to note in the case of iterator loops. Modifications to the Makefile are necessary to include the VEDA library and to process code files with two different compilers. Both the VEDA library and NEC compiler are proprietary software that are usually only available on devices with VEs. Therefore the modifications should only be made when VEs will be used. Whether code should be executed on VEs can be directly derived from the AMT and the particular configuration of the Makefile can be made dependent on this.

Both CUDA and VEDA minimize data transfers, which results in differences in loops iterating over arrays compared to CPU code. Listing 5.5 gives an example in which a loop iterates over a subset of an array.

Listing 5.5: Example of a *for* loop that iterates over a subset of an array.

```c
int arr[4096];
for(int i = 1024; i < 2048; ++i){
    ...
}
```

If this example would be executed on a GPU or VE, only the subset of the array relevant to the loop would be allocated on and transferred to the device. Therefore, the range of the loop on the device starts at zero and extends to the length of the partial array. Listing 5.6 shows how the previous example would look in this case.

Listing 5.6: How the example in Listing 5.5 might look when executed on a device.

```c
int device_arr[1024];
//data transfer to device
for(int i = 0; i < 1024; ++i){
    ...
}
//data transfer from device
```

The current implementation of device pooling can be reused for VEs. This means that a single management thread is created for each VE. Due to this and the great similarities of VEDA to CUDA in terms of data transfers, the control of different devices can remain identical. Together with the separation of host and device code, MPI can be used in its present manner. The PPL will therefore still be able to execute programs that use VEs in a heterogeneous environment.

VEDA only provides data transfer functionality and not the possibility to parallelize code. This is instead achieved with the NEC compiler through explicit instructions. Deviating from the previous parallelization model of the PPL, OpenMP is typically used for this on VEs. This results in the introduction of OpenMP pragmas and data management clauses into the device code and Makefile. Because the use of OpenMP is limited to the device code file, there are no side effects for the rest of the code.

5.3.2 Parallel Patterns

A mandatory requirement for the extended PPL is the generation of code representing the parallel patterns for VEs. The current implementation of the PPL generates two types of parallel code. One is parallelized C++ code for CPUs and the other is a CUDA based GPU kernel. No third type is required to support VEs. This is due to the use of standard C, C++, and Fortran code by VEs. Consequently, the currently generated kernel code for CPUs can also be used for VEs. Some adjustments are needed due to the different architecture and programming model of CPUs. The necessary modifications have been examined in the previous sections and will now be illustrated with code examples.

To summarize, the three main differences between CPU and VE code are **use of OpenMP**, **embedding in the device file**, and **modified loop ranges**. These differences only result in marginal changes to the code. To illustrate this, the code generated for a map pattern as it occurs in the K-Nearest neighbors (NN) algorithm is examined. Listing 5.7 lists generated CPU code. The PPL generates it in this form, except for small adjustments made for better readability. 48 threads are used to parallelize the algorithm. The depicted loop is executed on the 48th thread and processes the corresponding part of the input arrays. This is caused by the use of the PThreads library and is described in more detail by Schmitz [82].

Listing 5.7: A loop representing the 48th part of a NN kernel parallelized with 48 threads.

```
for (size_t INDEX = 31333318; INDEX < 31333318 + 666682; ++INDEX)
    {
    dists[INDEX] = sqrt((lon[INDEX] - target_lon) * (lon[INDEX] -
         target_lon) + (lat[INDEX] - target_lat) * (lat[INDEX] -
        target_lat));
}
```

The code shown in Listing 5.8 demonstrates how the same parallelized algorithm is implemented for a VE. Since OpenMP is used instead of PThreads, splitting and manually distributing the loop is no longer necessary. Instead, this is achieved by OpenMP pragmas and clauses.

Listing 5.8: The parallelized NN kernel as it is implemented for VEs.

```
#pragma omp parallel for shared(lat, lon, dists) schedule(static)
for (size_t INDEX = 0; INDEX < 32000000; ++INDEX) {
    dists[INDEX] = sqrt((lon[INDEX] - target_lon[0]) * (lon[INDEX
       ] - target_lon[0]) + (lat[INDEX] - target_lat[0]) * (lat[
       INDEX] - target_lat[0]));
}
```

The minor differences between CPU and VE code, as shown in the examples, suggest that when the code generator is extended, a significant part of the current implementation can likely be reused with only a few adjustments. As a result, the implementation effort will be reduced.

These compute kernels can also be executed on multiple systems with distributed memory by using MPI. It acts as a wrapper for the individual code sections and ensures execution on the intended system. This approach does not require modifications, since the introduction of VE offloading takes place within these wrappers.

5.3.3 Visitor

The generator implementation utilizes the visitor pattern of the AMT nodes. Because new nodes are introduced, there must also be handler functions for each new node. Many of these nodes can be implemented by reusing previously defined functions for GPUs or CPUs. The data movement nodes can be designed like GPU nodes, with the main difference being the use of VEDA instead of CUDA. However, for the parallel call nodes, parts of both the CPU and GPU implementations can be reused. The reason for this is that within an offloaded function, the surrounding data movements and initializations are similar to GPUs, while the kernel itself is similar to CPUs.

5.3.4 Iterator Node

The proposal to add an iterator node to the APT and AMT also has consequences for the code generator. The current issue is that loops similar to those in Listing 5.1 generate unneccesary data transfers. In addition, they also generate many individual kernels, in this case 100, instead of just four, which can cause a large compilation overhead. These problems are solved with the introduction of the iterator node that is completely offloaded to VEs. However, this necessitates an entirely new implementation for processing the different variants of the iterator node and their associated code generation. For VEs this requires a new form of offloaded wrapper function. It would contain the necessary data transfers, iterator loops, and function calls.

5.4 Implementation

As part of this thesis, the PPL toolchain is extended to generate code for VEs. This extension is based on the current state of development of the PPL, as described in Section 3.3. Although the implementation does not yet meet all goals and requirements due to time constraints, a significant portion is realized. It therefore represents a proof-of-concept that generates functioning code for VEs and is intended to serve as a foundation for the future realization of the other goals. This section presents the most relevant changes undertaken, while also describing which features are not realized, and the challenges encountered during the implementation.

5.4.1 Optimizer

The optimizer is the first part of the PPL where significant changes are made. These enable the use of VEs as offload devices. The base implementation from Schmitz [82] does not consider offload devices generically, but relies explicitly on GPUs. The implementation is extended to also include VEs. Consequently, an assignment to VEs can now also take place. The code analysis features mentioned in Section 5.2.3 are not implemented, due to time constraints.

5.4.2 AMT

The changes to the AMT ensure that the nodes associated with the device mapping are generated. As a result, a number of new nodes are introduced, and existing nodes are modified. The newly introduced nodes cover allocation, deallocation, data movement, and parallel call mappings. These model data transfers as well as parallel execution on VEs. All are based on pre-existing GPU nodes. However, these changes are not sufficient to actually perform the data transfer and parallel code generation. This also requires an extension of the mapping generator associated with the AMT. It processes the mapping generated by the optimizer and creates the corresponding data transfer nodes for devices. Since this process is not designed generically, it is necessary to explicitly add VEs as a possible offload target. The visitor associated with the code generator can only initiate the generation of the correct code through this change.

Existing nodes are also modified because of the introduction of the new nodes and the fact that previously the only offloading devices were GPUs. For example, the possibility to execute on VEs is added to the nodes belonging to parallel patterns. In addition, the associated data transfers are included. The necessity of these changes is brought about by the tight coupling of nodes in the current implementation.

It is important to mention that the iterator node proposed in Section 5.2.1 is not yet implemented. While it is necessary to guarantee performant use of VEs in the future, it is not requried in order to generate functioning code. It is therefore currently still a problem that in cases of iterator loops calling parallel patterns, unnecessarily many data transfers and kernels are generated. The generated code is nevertheless functional.

5.4.3 Code Generator

The most important element for the generation of functioning VE code is the code generator itself. All other implemented changes had the goal of enabling it to perform this task. The code generator extension is based on the requirements and proposed solutions outlined in Section 5.3. There are key commonalities between GPU and VE code as well as between CPU and VE code. Thus, parts of the current generator implementation are used as a starting point.

Device Pool

Section 5.3.1 shows that the current device pool implementation is applicable to VEs. It is currently implemented within CUDA based library files. These are adapted by using VEDA functions instead of CUDA. The generated code is thus able to initialize, synchronize, and release an arbitrary number of VEs. Unlike the CUDA implementation, different contexts are created for each device. These apply locally to the thread associated with the device and work is executed within these. Furthermore, by setting an environment variable, it is ensured that all cores on the VEs are utilized by OpenMP.

Data Transfers

For the data transfer between host and GPU, two library files are generated, which contain wrapper functions for the corresponding CUDA functions. This concept is also used for the VEDA implementation. In combination with the device pool implementation it is possible to allocate and transfer data separately for each device. The usage within the main code file is as shown in Listing 5.9.

Listing 5.9: Allocation and subsequent transfer of data from the host to a VE.

```
VEDAdeviceptr Offload_Data_1;
auto f_alloc_1 = [&] () {
    veda_alloc_wrapper(&Offload_Data_1, sizeof(float) * 1000);
};
getVEPool()->at(0).addWork(f_alloc_1);
auto f_movement_1 = [&] () {
    veda_host2device_wrapper(Offload_Data_1, &data, sizeof(float)
        * 1000);
};
getVEPool()->at(0).addWork(f_movement_1);
```

In this example, a `VEDAdeviceptr` is created. It stores the device memory address of the data allocated with `veda_alloc_wrapper`. The `veda_host2device_wrapper` function transfers the data stored in the `data` array to this memory address. All used VEDA functions apply an asynchronous data transfer approach. This can theoretically reduce transfer times. Both instructions are assigned to the device managing thread by means of lambda functions. After their execution, the data array is available and usable on the device.

The execution of device code is started on the host side. This part is stored in a host library file, to allow reusability and to increase readability. The host library contains wrapper functions which have a number of tasks. An example for their design is given in Listing 5.10.

Listing 5.10: Preparation of the execution of a function on a VE.

```
void veda_wrapper_1(VEDAdeviceptr data) {
    VEDAmodule mod;
    vedaModuleLoad(&mod, "veda_lib_device.vso");
    VEDAfunction func;
    vedaModuleGetFunction(&func, mod, "kernel_veda_1");
    VEDAargs args;
    vedaArgsCreate(&args);
    vedaArgsSetVPtr(args, 0, data);
    vedaLaunchKernelEx(func, 0, args, 1, 0);
}
```

First, the device code file is loaded and the required function is defined. Then, the arguments for this function are set. These are the data required in the function in the form of a `VEDAdeviceptr`. Lastly, the function is executed.

Kernel and Patterns

The kernel that is executed on the device side accepts the input data mentioned in the previous section. However, since these are not the data itself, but only a pointer to the location of the data, they still need to be processed for use as an ordinary array. As shown in Listing 5.11, this is done using the `vedaMemPtr` function.

Listing 5.11: The device code of an algorithm. It includes the processing of input pointers as well as a parallelized *for* loop.

```
extern "C" void kernel_veda_1(VEDAdeviceptr Offload_Data_1) {
    float* data;
    vedaMemPtr((void**)&data, Offload_Data_1);

#pragma omp parallel for shared(data)
    for (size_t INDEX = 0; INDEX < 1000; ++INDEX) {
        ...
    }
}
```

This function is part of the `veda_device` library and converts the given memory address into a usable array. This is followed by the actual execution of the compute kernel. These kernels are the transformed parallel patterns. The map pattern is ported for VEs in this work. OpenMP is used for parallelization. To prevent data races, shared arrays must always be set as shared. This is done for all input data in the current implementation. This is due to the lack of analysis tools in the PPL to determine which data is actually shared. This results in an overhead for the synchronization of threads, which reduces the achievable performance. In order to use VEs optimally, such an analysis tool still has to be developed and integrated into the PPL.

Within the device code, in addition to the generated kernels, other functions are also implemented. They are also part of the CUDA library files and contain, among other things, the code for reductions and data management on the GPU. By implementing them in the device code, native compilation and execution is enabled. The addition of OpenMP clauses to the reductions also effectively parallelized them.

During the implementation process, it became apparent that the current implementation of nested patterns is not applicable or portable to VEs. For example, a nested pattern is a sequential call within a map pattern, as can be seen in Listing 5.12. In

Listing 5.12: An excerpt from the PPL code for K-Means clustering, which contains a nested pattern.

```
map determine_centroids([[Float]]points, [[Float]]centroids):[Int]
    assignment {
    assignment[INDEX] = assign_centroid(points[INDEX], centroids[
        INDEX])
}
seq assign_centroid([Float]point, [Float]centroid):Int{
    ...
}
```

the current version of the PPL this sequential function is inlined. The problem with the generated code is that lambda functions are not used for data transfer to devices as before. Instead, a transfer function is called within the device code. However, this approach is not applicable for VEs. They cannot use these transfer functions within the device code and instead have to transfer data by VEDA calls on the host side. The required changes are out of the scope of this thesis.

5.5 Evaluation

Due to the extension undertaken in the previous section, the PPL is now partially capable of generating functioning code for VEs. The quality of this code can be assessed in different ways. Maintainability and reusability are important factors, but cannot be quantified easily or quickly. Performance and correctness can be examined far better in the context of this work, given the time constraints. Therefore, the code generator developed in this thesis is evaluated based on these two criteria.

5.5.1 Setup and Methodology

As stated in Section 5.4, the presented PPL extension is capable of generating code for simple map patterns. The NN benchmark fulfills this limitation. It serves as a benchmark whose performance is subsequently assessed. In all test configurations, the algorithm uses the same data set and configuration with 32 million data points and 5000 neighbors, respectively. The data collection is done on one VE, purely on CPUs and on one GPU. Additionally, the performance data for VEs is contrasted by a handwritten and manually optimized code. Both VE codes are executed on the nca01 node of the RWTH cluster using a single device. Only the used VEs in form of the SX-Aurora Type 10B is relevant for this measurement. The CPU code is executed on a node of the CLAIX-2018 cluster. These are equipped with two 24-core Intel Xeon Platinum 8160 processors [44]. The GPU code is executed on the CLAIX-2018-GPU cluster with the same CPU and using a single NVIDIA V100 [73, 92]. The generated code is compiled using gcc 11.2.0, the NVIDIA CUDA compiler 11.6.2, and version 3.5.1 of the NEC compiler. The collected data comprises the total runtime, excluding file I/O, and the time required by the compute kernel. This enables the identification of the overhead caused by data transfers. For the CPU measurement, it must be noted that the runtime for each thread is measured seperately and only the longest one is taken as the result. This is justified by the barrier synchronization, which blocks the execution until all threads have finished their task. The correctness of the generated code is assessed by comparing the results of the different versions of the algorithm. The results obtained from the original PPL are assumed to be correct, since they were validated by Schmitz [82]. Overall, this analysis provides a comprehensive picture of the presented implementation.

5.5.2 Results

This section first examines the kernel runtime and subsequently the total runtime. Figure 5.2 shows the measured runtimes of the different architectures. A detailed overview of the values can be found in Table A.3. The measured values show a wide range, except on the GPU. It can be clearly seen that GPUs also offer the highest performance, with handwritten VE code in second place. The generated VE takes the most time, with the CPU being faster. The average runtime of the generated VE code is about ten times longer compared to the handwritten code. The two kernels differ only in a few aspects. First, unlike the handwritten kernel, the generated kernel shares all variables between the OpenMP threads. This may cause a synchronization overhead and result in a lower performance. Secondly, the two target variables in the generated code are arrays, of which the first element is always used. This might be optimized differently by the compiler and cause the lower performance.

By examining the total runtime, an overview of the overheads introduced by data transfers can be acquired. Figure 5.3 presents the total runtimes measured on the different architectures without file I/O. The handwritten code has the lowest runtime. This is expected, since it was already shown that the generated code contains many

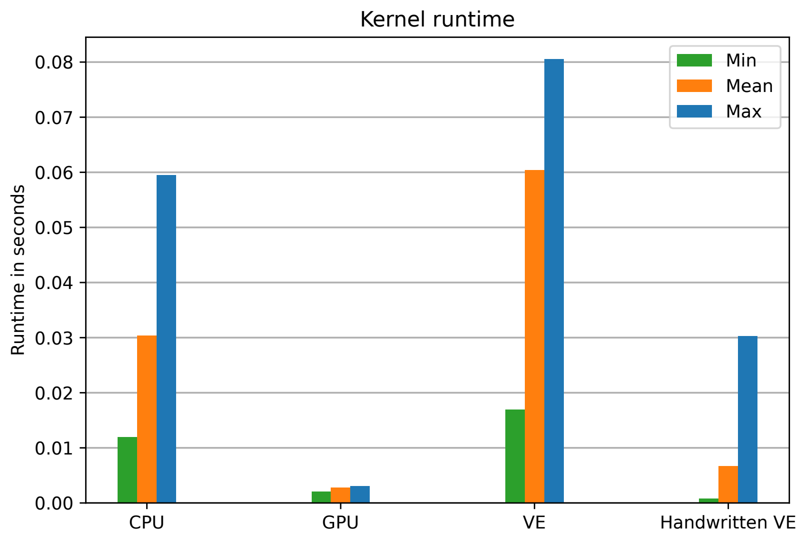


Figure 5.2: Kernel runtime of the NN algorithm on different architectures.

overheads which negatively influence the performance [82]. It can also be observed that the data transfers required for GPUs and VEs cause an increase in overall runtime for the generated code. Thus, the CPU is actually the optimal hardware choice. The influence of the time spent in the kernels is negligible, since it only represents a very small fraction of the total runtime. It is noticeable that the total runtime of the generated VE code is significantly larger than that of the handwritten code. Since both use the VEDA library, this can only be attributed to the overheads caused by the thread pool implementation.

5.6 Discussion

In this chapter, the feasibility of generating code for VEs within the framework of the PPL was investigated. The necessary changes were identified and suggestions for their implementation were given. These proposals follow a conservative approach with respect to the type of offloading support. Instead of enabling generic offloading, the current principle of explicit offloading was retained. This reduces the required development effort. Nevertheless, a generic approach to offloading would have advantages that are missing in the proposed implementation. With generic hardware support, only changes to the code generator would be required when introducing support for new devices. In the long-term, this would reduce the programming effort because it frees developers from working on the other core components such as the optimizer. This approach was not pursued, as it would require major changes and significant rewrites of the optimizer and AMT. There are also no current plans to extend the PPL to other hardware. Instead, the development focus is on improving

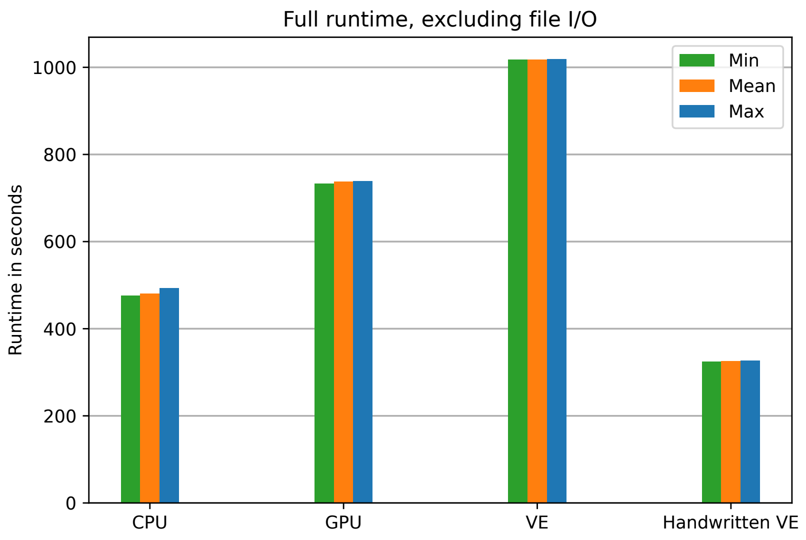


Figure 5.3: Full runtime of the NN algorithm on different architectures.

the performance of the generated code, tool usability, and bugfixing.

A consequence of the explicit offload support of VEs is the introduction of the iterator loops and nodes. It is a workaround of the previous limitations to comply with the VE programming model. It can also be used for GPUs in the future to reduce the necessary data transfers for them as well. However, the implementation could be incompatible with linear programs as the optimization model.

Some of the proposals for the extension were implemented as a proof-of-concept, that constitutes a first step and foundation for future development. The performance evaluation has shown that the achieved performance was far below its potential. This contrast became clear by comparison with the handwritten code. This code had a significantly lower runtime for the compute kernel as well as for the whole program. These differences are explained by the device pool implementation, since both codes use the same VEDA library commands. Improving the performance of the generated code, including for CPUs and GPUs, requires a significant overhaul of the way threads and devices are managed. This could also mean a departure from the use of PThreads. Overall, this would be a large, long-term project. Improvements are not only needed in the device pool. While the kernel does not have a significant impact on the total runtime in the test case, an improvement is desirable, especially in the case of larger kernels with longer runtimes. The handwritten code shows that such an improvement is possible, but it requires a more detailed bottleneck analysis. The proof-of-concept accomplished its overall goals. It showed that the generated code is correct and highlighted the potential for improvement in terms of performance.

6 Conclusion and Future Work

The first main contribution of this thesis is the development of a static performance model for Vector Engines (VEs). It extends the cache-aware Roofline model [100, 42] and therefore considers the main code and hardware properties that affect performance. The extensions incorporate the actual vector length and a modified main memory bandwidth scaling into the model. This covers the special hardware properties of VEs.

Benchmarks with different performance bounds are used to verify the model. The STREAM benchmark [57] exhibited an unexpected performance profile, which may have been caused by the simplicity of the benchmark kernels. The other benchmarks, which are largely based on real world algorithms, did not exhibit this characteristic. A high accuracy of approximately 99% is observed for compute bound algorithms. The accuracy for memory and cache bound algorithms ranges from about 80% to 96% due to untraceable memory optimizations by the hardware.

The second main contribution of this work is the design of code generation for VEs within the framework of the Parallel Pattern Language (PPL) [61]. Efficient data transfers within the constraints of the current implementation require changes to the frontend language and intermediate representation. This can be accomplished by introducing a dedicated iterator loop in the frontend language and new nodes in the Abstract Pattern Tree (APT) and Abstract Mapping Tree (AMT). Additional nodes also have to be introduced there to model a mapping to VEs. No modifications are needed in the Hardware Language (HL) and cluster model because they are already able to accurately represent VEs. New code analysis features need to be developed for the performance model in order to determine the vectorized loop, vector length, and exact memory constraint of an algorithm. The code generator should use VEDA [68] instead of CUDA for data transfers and device management. The compute kernels generated from parallel patterns should be parallelized with OpenMP.

A proof-of-concept based on the code generator design was developed and extends the PPL. Functioning and correct VE code can be generated for simple map patterns. Altough the performance of the generated code showed a significant slowdown when compared to handwritten and manually optimized VE code, the main causes could be identified. These are primarily the thread pool implementation and the current OpenMP variable scoping.

Future work might improve the accuracy of the model by cooperating with NEC. This might lead to an improved reflection of the hardware influence in the model. However, even a purely analytical examination of the architecture, without help from NEC, can lead to important insights. For example, the interaction of the different compute units of a Vector Processing Unit (VPU) can be investigated. It is also generally important to understand why an algorithm can behave cache bound when it should not, namely when a single variable is larger than the cache. An analysis of the STREAM benchmark performance profile is another way to better understand how the performance is influenced by the architecture. A more detailed curve fitting based on additional bechmarks might be performed instead of a deep hardware examination. The benchmarks must first be ported to VE code and subsequently measured in terms of performance. This process can either be performed manually or with the help of Machine Learning (ML).

The designed code generation and accompanying changes to the PPL should be implemented in future work by building on the existing proof-of-concept. The dedicated iterator loop is not needed if an automated recognition is performed instead. Future work can also implement more efficient data transfers to GPUs by using the iterator loop. Alleviating the bottleneck caused by suboptimal OpenMP variable scoping requires a new code analysis feature to determine a correct and optimal scoping. The current thread pool implementation needs to be improved to achieve a higher performance on all target architectures.

A Measurement Results

Vector length	K-Means clustering	K-Nearest neighbors	Jacobi solver	Linear shift
1	0.278	2.671	1.25	8.537
2	0.529	5.283	2.527	17.088
4	1.103	8.614	5.103	33.712
8	2.119	15.064	9.944	67.275
16	4.13	25.378	19.864	135.936
32	7.231	40.414	39.774	274.69
64	14.009	65.824	83.515	542.485
128	28.317	124.178	147.46	1110.605
256	54.873	233.105	252.31	2147.929

Table A.1: Performance (GFlop/s) of the computing benchmarks on a VE for each vector length.

Algorithm	Copy	Scale	Add	Triad
1	22.054	21.122	28.558	28.104
2	46.363	47.293	51.764	51.234
4	98.089	99.214	104.789	104.488
8	190.387	187.875	200.151	199.088
16	316.039	309.794	334.686	328.248
32	454.464	457.768	489.848	498.578
64	818.145	792.340	905.15	931.257
128	1029.355	1029.626	1005.496	1015.327
256	1015.584	1014.573	1010.534	1010.923

Table A.2: Performance (GByte/s) of the STREAM benchmarks on a VE for each vector length.

Type	CPU	GPU	VE	Handwritten VE
Mean runtime	0.030304s	0.002765s	0.060351s	0.006688s
Min runtime	0.011922s	0.002048s	0.016963s	0.000805s
Max runtime	0.059449s	0.003072s	0.080488s	0.030247s
Runtime standard deviation	0.0134s	0.0005s	0.0172s	0.0118s

Table A.3: Measurement results of generated code on various architectures. The generated code is the K-Nearest neighbors algorithm.

Bibliography

[1] A. A. Abbo, R. P. Kleihorst, V. Choudhary, and L. Sevat. Power Consumption of Performance-scaled SIMD Processors. In *International Workshop on Power and Timing Modeling, Optimization and Simulation*, pages 532–540. Springer, 2004. doi:10.1007/978-3-540-30205-6_55.

[2] Advanced Micro Devices Inc. AMD Radeon™ RX 6800 Graphics Card. `https://www.amd.com/en/products/graphics/amd-radeon-rx-6800`. Accessed: 30.11.2022.

[3] I. V. Afanasyev, V. V. Voevodin, V. V. Voevodin, K. Komatsu, and H. Kobayashi. Analysis of Relationship between SIMD-Processing Features used in NVIDIA GPUs and NEC SX-Aurora TSUBASA Vector Processors. In *International Conference on Parallel Computing Technologies*, pages 125–139. Springer, 2019. doi:10.1007/978-3-030-25636-4_10.

[4] I. V. Afanasyev, V. V. Voevodin, V. V. Voevodin, K. Komatsu, H. Kobayashi, et al. Developing Efficient Implementations of Shortest Paths and Page Rank Algorithms for NEC SX-Aurora TSUBASA Architecture. *Lobachevskii Journal of Mathematics*, 40(11):1753–1762, 2019. doi:10.1134/S1995080219110039.

[5] S. M. Al-sudany, A. S. Al-Araji, and B. M. Saeed. Architecture and Advantages of SIMD in Multimedia Applications. *Journal of Xi'an University of Architecture & Technology*, 12:1452–1459.

[6] A. Alexandrov, M. F. Ionescu, K. E. Schauser, and C. Scheiman. LogGP: Incorporating Long Messages into the LogP Model—One Step Closer Towards a Realistic Model for Parallel Computation. In *Proceedings of the seventh annual ACM symposium on Parallel algorithms and architectures*, pages 95–105, 1995. doi:10.1145/215399.215427.

[7] M. S. B. Altaf and D. A. Wood. LogCA: A Performance Codel for Hardware Accelerators. *IEEE Computer Architecture Letters*, 14(2):132–135, 2014. doi:10.1109/LCA.2014.2360182.

[8] S. F. Antao, A. Bataev, A. C. Jacob, G.-T. Bercea, A. E. Eichenberger, G. Rokos, M. Martineau, T. Jin, G. Ozen, Z. Sura, et al. Offloading Support for OpenMP in Clang and LLVM. In *2016 Third Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC)*, pages 1–11. IEEE, 2016. doi:10.1109/LLVM-HPC.2016.006.

[9] Argonne Leadership Computing Facility. Aurora. `https://www.alcf.anl.gov/aurora`. Accessed: 6.01.2023.

[10] ARM. SVE2 Architecture Fundamentals. `https://developer.arm.com/documentation/102340/0001/SVE2-architecture-fundamentals`. Accessed: 30.11.2022.

[11] K. S. Brose. Comparing and Modelling the Performance of Different ML Frameworks and Hardware Accelerators in a Coupled OpenFoam+ML Application. Master's thesis, Chair for High Performance Computing, RWTH Aachen University, 2022.

[12] S. Burak. Application of a Parallel Pattern Language to LULESH. Master's thesis, Chair for High Performance Computing, RWTH Aachen University, 2022.

[13] V. C. Cabezas and M. Püschel. Extending the Roofline Model: Bottleneck Analysis with Microarchitectural Constraints. In *2014 IEEE International Symposium on Workload Characterization (IISWC)*, pages 222–231. IEEE, 2014. doi:10.1109/IISWC.2014.6983061.

[14] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron. Rodinia: A Benchmark Suite for Heterogeneous Computing. In *2009 IEEE international symposium on workload characterization (IISWC)*, pages 44–54. IEEE, 2009. doi:10.1109/IISWC.2009.5306797.

[15] S. Che, J. W. Sheaffer, M. Boyer, L. G. Szafaryn, L. Wang, and K. Skadron. A Characterization of the Rodinia Benchmark Suite with Comparison to Contemporary CMP Workloads. In *IEEE International Symposium on Workload Characterization (IISWC'10)*, pages 1–11. IEEE, 2010. doi:10.1109/IISWC.2010.5650274.

[16] J. W. Choi, D. Bedard, R. Fowler, and R. Vuduc. A Roofline Model of Energy. In *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*, pages 661–672. IEEE, 2013. doi:10.1109/IPDPS.2013.77.

[17] T. Cramer, M. Römmer, B. Kosmynin, E. Focht, and M. S. Müller. OpenMP Target Device Offloading for the SX-Aurora TSUBASA Vector Engine. In *International Conference on Parallel Processing and Applied Mathematics*, pages 237–249. Springer, 2019. doi:10.1007/978-3-030-43229-4_21.

[18] T. Cramer, B. Kosmynin, S. Moll, M. Römmer, E. Focht, and M. S. Müller. Evaluating the Performance of OpenMP Offloading on the NEC SX-Aurora TSUBASA Vector Engine. *Supercomputing Frontiers and Innovations*, 8(2):59–74, 2021. doi:10.14529/jsfi210204.

[19] D. Culler, R. Karp, D. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, and T. Von Eicken. LogP: Towards a Realistic Model of Parallel Computation. In *Proceedings of the fourth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 1–12, 1993. doi:10.1145/155332.155333.

[20] B. Da Silva, A. Braeken, E. H. D'Hollander, and A. Touhafi. Performance Modeling for FPGAs: Extending the Roofline Model with High-level Synthesis Tools. *International Journal of Reconfigurable Computing*, 2013, 2013. doi:10.1155/2013/428078.

[21] N. Denoyelle, B. Goglin, A. Ilic, E. Jeannot, and L. Sousa. Modeling Large Compute Nodes with Heterogeneous Memories with Cache-aware Roofline Model. In *International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems*, pages 91–113. Springer, 2017. doi:10.1007/978-3-319-72971-8_5.

[22] J. M. Diaz, S. Pophale, K. Friedline, O. Hernandez, D. E. Bernholdt, and S. Chandrasekaran. Evaluating Support for OpenMP Offload Features. In *Proceedings of the 47th International Conference on Parallel Processing Companion*, pages 1–10, 2018. doi:10.1145/3229710.3229717.

[23] N. Ding and S. Williams. An Instruction Roofline Model for GPUs. In *2019 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*, pages 7–18, 2019. doi:10.1109/PMBS49563.2019.00007.

[24] N. Ding, M. Awan, and S. Williams. Instruction Roofline: An Insightful Visual Performance Model for GPUs. *Concurrency and Computation: Practice and Experience*, 34(20):e6591, 2022. doi:10.1002/cpe.6591.

[25] L. Djoudi, D. Barthou, P. Carribault, C. Lemuet, J.-T. Acquaviva, W. Jalby, et al. MAQAO: Modular Assembler Quality Analyzer and Optimizer for Itanium 2. In *The 4th Workshop on EPIC architectures and compiler technology, San Jose*, volume 200, 2005.

[26] M. J. Flynn. Very High-speed Computing Systems. *Proceedings of the IEEE*, 54(12):1901–1909, 1966. doi:10.1109/PROC.1966.5273.

[27] M. J. Flynn. Some Computer Organizations and their Effectiveness. *IEEE transactions on computers*, 100(9):948–960, 1972. doi:10.1109/TC.1972.5009071.

[28] E. Focht. Vector Evolution: The Path to the NEC SX-Aurora TSUBASA, 2 2019.

[29] E. Focht. VE Offloading. `https://blog.rwth-aachen.de/hpc_import_20210107/attachments/46401154/50790468.pdf`, 2019. Accessed: 30.11.2022.

[30] Forschungszentrum Jülich. First European Exascale Supercomputer Coming to Jülich. `https://www.fz-juelich.de/en/news/archive/press-release/2022/first-european-exascale-supercomputer-coming-to-julich`, 2022. Accessed: 29.12.2022.

[31] B. Furht, editor. *SIMD (Single Instruction Multiple Data Processing)*, pages 817–819. Springer US, Boston, MA, 2008. ISBN 978-0-387-78414-4. doi:10.1007/978-0-387-78414-4_220. URL `https://doi.org/10.1007/978-0-387-78414-4_220`.

[32] A. Guermouche and A.-C. Orgerie. Experimental Analysis of Vectorized Instructions Impact on Energy and Power Consumption under Thermal Design Power constraints. Research Report 2, Télécom Sud Paris, 2019. URL `https://hal.archives-ouvertes.fr/hal-02167083`.

[33] G. Hager, J. Treibig, J. Habich, and G. Wellein. Exploring Performance and Power Properties of Modern Multi-core Chips via Simple Machine Models. *Concurrency and computation: practice and experience*, 28(2): 189–210, 2016. doi:10.1002/cpe.3180.

[34] J. Hammer, G. Hager, J. Eitzinger, and G. Wellein. Automatic Loop Kernel Analysis and Performance Modeling with Kerncraft. In *Proceedings of the 6th International Workshop on Performance Modeling, Benchmarking, and Simulation of High Performance Computing Systems*, pages 1–11, 2015. doi:10.1145/2832087.2832092.

[35] M. Hassaballah, S. Omran, and Y. B. Mahdy. A Review of SIMD Multimedia Extensions and their usage in Scientific and Engineering Applications. *The Computer Journal*, 51(6):630–649, 2008. doi:10.1093/comjnl/bxm099.

[36] M. Hill and V. J. Reddi. Gables: A Roofline Model for Mobile SoCs. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 317–330. IEEE, 2019. doi:10.1109/HPCA.2019.00047.

[37] M. D. Hill, M. D. Hill, N. P. Jouppi, N. P. Jouppi, and G. S. Sohi. *Readings in Computer Architecture*. Gulf Professional Publishing, 2000.

[38] J. Hofmann and D. Fey. An ECM-based Energy-efficiency Optimization Approach for Bandwidth-limited Streaming Kernels on Recent Intel Xeon Processors. In *2016 4th International Workshop on Energy Efficient Supercomputing (E2SC)*, pages 31–38. IEEE, 2016. doi:10.1109/E2SC.2016.010.

[39] J. Hofmann, D. Fey, M. Riedmann, J. Eitzinger, G. Hager, and G. Wellein. Performance Analysis of the Kahan-enhanced Scalar Product on Current Multicore and Many-core Processors. *Concurrency and Computation: Practice and Experience*, 29(9):e3921, 2017. doi:10.1002/cpe.3921.

[40] J. Hofmann, G. Hager, and D. Fey. On the Accuracy and Usefulness of Analytic Energy Models for Contemporary Multicore Processors. In *International conference on high performance computing*, pages 22–43. Springer, 2018. doi:10.1007/978-3-319-92040-5_2.

[41] P. Y. Hsu and E. S. Davidson. Highly Concurrent Scalar Processing. *ACM SIGARCH Computer Architecture News*, 14(2):386–395, 1986. doi:10.1145/17356.17401.

[42] A. Ilic, F. Pratas, and L. Sousa. Cache-aware roofline model: Upgrading the loft. *IEEE Computer Architecture Letters*, 13(1):21–24, 2013. doi:10.1109/L-CA.2013.6.

[43] T. Imai. NEC Earth Simulator and the SX-Aurora TSUBASA. In *Operating Systems for Supercomputers and High Performance Computing*, pages 139–160. Springer, 2019. doi:10.1007/978-981-13-6624-6_9.

[44] Intel Corporation. Intel® Xeon® Platinum 8160 Processor. `https://www.intel.com/content/www/us/en/products/sku/120501/intel-xeon-platinum-8160-processor-33m-cache-2-10-ghz/specifications.html`. Accessed: 15.12.2022.

[45] Khronos Group. OpenGL - The Industry Standard for High Performance Graphics. `https://www.opengl.org/`, . Accessed: 10.01.2023.

[46] Khronos Group. Vulkan API. `https://www.vulkan.org/`, . Accessed: 10.01.2023.

[47] K. Komatsu, A. Onodera, E. Focht, S. Fujimoto, Y. Isobe, S. Momose, M. Sato, and H. Kobayashi. Performance and Power Analysis of a Vector Computing System. *Supercomputing Frontiers and Innovations*, 8(2):75–94, 2021. doi:10.14529/jsfi210205.

[48] T. Koskela, Z. Matveev, C. Yang, A. Adedoyin, R. Belenov, P. Thierry, Z. Zhao, R. Gayatri, H. Shan, L. Oliker, et al. A Novel Multi-level Integrated Roofline Model Approach for Performance Characterization. In *International Conference on High Performance Computing*, pages 226–245. Springer, 2018. doi:10.1007/978-3-319-92040-5_12.

[49] M. Larabel. LLVM Now Has "Official" Support For Targeting NEC's Vector Engine (VE). `https://www.phoronix.com/news/LLVM-NEC-VE-Official`, 2021.

[50] S. Lloyd. Least Squares Quantization in PCM. *IEEE transactions on information theory*, 28(2):129–137, 1982. doi:10.1109/TIT.1982.1056489.

[51] Y. J. Lo, S. Williams, B. V. Straalen, T. J. Ligocki, M. J. Cordery, N. J. Wright, M. W. Hall, and L. Oliker. Roofline Model Toolkit: A Practical Tool for Architectural and Program Analysis. In *International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems*, pages 129–148. Springer, 2014. doi:10.1007/978-3-319-17248-4_7.

[52] A. Lopes, F. Pratas, L. Sousa, and A. Ilic. Exploring GPU Performance, Power and Energy-efficiency Bounds with Cache-aware Roofline Modeling. In *2017 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 259–268. IEEE, 2017. doi:10.1109/ISPASS.2017.7975297.

[53] D. Marques, H. Duarte, A. Ilic, L. Sousa, R. Belenov, P. Thierry, and Z. A. Matveev. Performance Analysis with Cache-aware Roofline Model in Intel Advisor. In *2017 International Conference on High Performance Computing & Simulation (HPCS)*, pages 898–907. IEEE, 2017. doi:10.1109/HPCS.2017.150.

[54] D. Marques, A. Ilic, and L. Sousa. Mansard Roofline Model: Reinforcing the Accuracy of the Roofs. *ACM Transactions on Modeling and Performance Evaluation of Computing Systems*, 6(2):1–23, 2021. doi:10.1145/3475866.

[55] R. Mathur, H. Matsuoka, O. Watanabe, A. Musa, R. Egawa, and H. Kobayashi. A Memory-Efficient Implementation of a Plasmonics Simulation Application on SX-ACE. *International Journal of Networking and Computing*, 6 (2):243–262, 2016.

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

[57] J. D. McCalpin et al. Memory bandwidth and machine balance in current high performance computers. *IEEE computer society technical committee on computer architecture (TCCA) newsletter*, 2(19-25), 1995.

[58] M. McCool, J. Reinders, and A. Robison. *Structured Parallel Programming: Patterns for Efficient Computation*. Elsevier, 2012.

[59] K. Meng and B. Norris. Mira: A Framework for Static Performance Analysis. In *2017 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 103–113. IEEE, 2017. doi:10.1109/CLUSTER.2017.43.

[60] Microsoft Corporation. DirectX graphics and gaming. `https://learn.microsoft.com/en-us/windows/win32/directx`. Accessed: 10.01.2023.

[61] J. Miller, L. Trümper, C. Terboven, and M. S. Müller. A Theoretical Model for Global Optimization of Parallel Algorithms. *Mathematics*, 9(14): 1685, 2021. doi:10.3390/math9141685.

[62] V. D. Modak, D. D. Langan, and T. F. Hain. A Pattern-based Development Tool for Mobile Agents. In *Proceedings of the 36th SIGCSE technical symposium on Computer science education*, pages 72–75, 2005. doi:10.1145/1047344.1047382.

[63] S. H. K. Narayanan, B. Norris, and P. D. Hovland. Generating Performance Bounds from Source Code. In *2010 39th International Conference on Parallel Processing Workshops*, pages 197–206. IEEE, 2010. doi:10.1109/ICPPW.2010.37.

[64] NEC Corporation. Programming Languages and Compilers. `https://www.nec.com/en/global/solutions/hpc/sx/tools.html`, . Accessed: 30.11.2022.

[65] NEC Corporation. NEC Vector Engine Models. `https://www.nec.com/en/global/solutions/hpc/sx/vector_engine.html`, . Accessed: 30.11.2022.

[66] NEC Corporation. SX-Aurora TSUBASA Offloading Frameworks. `https://www.nec.com/en/global/solutions/hpc/articles/tech07.html`, . Accessed: 30.11.2022.

[67] NEC Corporation. NEC SX-Aurora TSUBASA - Vector Engine. `https://www.nec.com/en/global/solutions/hpc/sx/vector_engine.html`, . Accessed: 29.12.2022.

[68] NEC Corporation. VEDA (VE Driver API). `https://github.com/SX-Aurora/veda`, . Accessed: 30.11.2022.

[69] NEC Corporation. SX-Aurora TSUBASA C/C++ Compiler User's Guide. `https://sxauroratsubasa.sakura.ne.jp/documents/sdk/pdfs/g2af01e-C++UsersGuide-028.pdf`, 2018, 2022. Accessed: 05.12.2022.

[70] NEC Corporation. SX-Aurora Architecture Deep Dive. `https://blog.rwth-aachen.de/hpc_import_20210107/attachments/46401154/50790460.pdf`, 2019. Accessed: 30.11.2022.

[71] B. Nichols, D. Buttlar, and J. P. Farrell. *Pthreads programming-a POSIX standard for better multiprocessing*, volume 19. O'reilly Sebastopol, CA, USA, 1996.

[72] NVIDIA Corporation. CUDA Toolkit Documentation v12.0. `https://docs.nvidia.com/cuda/`, . Accessed: 10.01.2023.

[73] NVIDIA Corporation. NVIDIA V100 TENSOR CORE GPU. `https://www.nvidia.com/en-us/data-center/v100/`, . Accessed: 30.11.2022.

[74] Oak Ridge National Laboratory. Frontier supercomputer debuts as world's fastest, breaking exascale barrier. `https://www.ornl.gov/news/frontier-supercomputer-debuts-worlds-fastest-breaking-exascale-barrier`, 2022. Accessed: 29.12.2022.

[75] OpenMP Architecture Review Board. OpenMP 5.2 Specification. `https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5-2.pdf`, 11 2021. Accessed: 08.12.2022.

[76] U. Paul and T. Gunkel. 45 Jahre Seegangsmodellvorhersagen im Geoinformationsdienst der Bundeswehr für die Deutsche Marine im Einsatz. 2021. ISSN 1865-6978.

[77] M. Pharr and R. Fernando. *GPU Gems 2: Programming Techniques for High-performance Graphics and General-purpose Computation.* Addison-Wesley Professional, 2005.

[78] B. R. Rau, D. W. L. Yen, W. Yen, and R. A. Towle. The Cydra 5 Departmental Supercomputer: Design Philosophies, Decisions, and Trade-offs. *Computer*, 22(1):12–35, 1989. doi:10.1109/2.19820.

[79] O. Reiche, M. Schmid, F. Hannig, R. Membarth, and J. Teich. Code Generation from a Domain-specific Language for C-based HLS of Hardware Accelerators. In *2014 international conference on hardware/software codesign and system synthesis (CODES+ ISSS)*, pages 1–10. IEEE, 2014. doi:10.1145/2656075.2656081.

[80] J. R. Reinders. Intel® AVX-512 Instructions. `https://www.intel.com/content/www/us/en/developer/articles/technical/intel-avx-512-instructions.html`, 2013. Accessed: 30.11.2022.

[81] S. Rul, H. Vandierendonck, J. D'Haene, and K. De Bosschere. An Experimental Study on Performance Portability of OpenCL Kernels. In *2010 Symposium on Application Accelerators in High Performance Computing (SAAHPC'10)*, 2010.

[82] A. Schmitz. Hpc code generation for parallel pattern based algorithms on heterogeneous architectures. Master's thesis, Chair for High Performance Computing, RWTH Aachen University, 2021.

[83] A. Schmitz, J. Miller, L. Trümper, and M. S. Müller. PPIR: Parallel Pattern Intermediate Representation. In *2021 IEEE/ACM International Workshop on Hierarchical Parallelism for Exascale Computing (HiPar)*, pages 30–40. IEEE, 2021. doi:10.1109/HiPar54615.2021.00009.

[84] R. Schöne, T. Ilsche, M. Bielert, A. Gocht, and D. Hackenberg. Energy Efficiency Features of the Intel Skylake-SP Processor and their Impact on Performance. In *2019 International Conference on High Performance Computing & Simulation (HPCS)*, pages 399–406. IEEE, 2019. doi:10.1109/HPCS48598.2019.9188239.

[85] Semiconductor Industry Association. 2015 International Technology Roadmap for Semiconductors (ITRS). `http://www.semiconductors.org/resources/2015-international-technology-roadmap-for-semiconductors-itrs/`, 2015. Accessed: 5.01.2023.

[86] A. Shahbahrami, B. Juurlink, and S. Vassiliadis. A Comparison between Processor Architectures for Multimedia Applications. In *Proc. 15th Annual Workshop on Circuits, System and Signal Processing (ProRISC 2004), the Netherlands*, pages 138–152, 2004.

[87] A. M. Shinsel. Intel® Advisor Roofline. `https://www.intel.com/content/www/us/en/developer/articles/guide/intel-advisor-roofline.html`, 2017. Accessed: 12.12.2022.

[88] M. Siracusa, E. Del Sozzo, M. Rabozzi, L. Di Tucci, S. Williams, D. Sciuto, and M. D. Santambrogio. A Comprehensive Methodology to Optimize FPGA Designs via the Roofline Model. *IEEE Transactions on Computers*, 71(8):1903–1915, 2021. doi:10.1109/TC.2021.3111761.

[89] H. Stengel, J. Treibig, G. Hager, and G. Wellein. Quantifying Performance Bottlenecks of Stencil Computations using the Execution-Cache-Memory Model. In *Proceedings of the 29th ACM on International Conference on Supercomputing*, pages 207–216, 2015. doi:10.1145/2751205.2751240.

[90] D. Talla, L. K. John, V. Lapinskii, and B. L. Evans. Evaluating Signal Processing and Multimedia Applications on SIMD, VLIW and Superscalar Architectures. In *Proceedings 2000 International Conference on Computer Design*, pages 163–172. IEEE, 2000. doi:10.1109/ICCD.2000.878283.

[91] techpowerup.com. AMD Radeon RX 6800. `https://www.techpowerup.com/gpu-specs/radeon-rx-6800.c3713`, . Accessed: 15.12.2022.

[92] techpowerup.com. NVIDIA Tesla V100 PCIe 32 GB. `https://www.techpowerup.com/gpu-specs/tesla-v100-pcie-32-gb.c3184`, . Accessed: 15.12.2022.

[93] M. O. Tokhi, M. A. Hossain, and M. H. Shaheed. *Parallel Computing for Real-time Signal Processing and Control.* Springer Science & Business Media, 2003.

[94] J. Treibig and G. Hager. Introducing a Performance Model for Bandwidth-limited Loop Kernels. In *International Conference on Parallel Processing and Applied Mathematics*, pages 615–624. Springer, 2009. doi:10.1007/978-3-642-14390-8_64.

[95] L. Trümper. Global optimization of parallel pattern-based algorithms for heterogeneous architectures. Master's thesis, Chair for High Performance Computing, RWTH Aachen University, 2020.

[96] L. Trümper, J. Miller, C. Terboven, and M. S. Müller. Automatic Mapping of Parallel Pattern-based Algorithms on Heterogeneous Architectures. In *International Conference on Architecture of Computing Systems*, pages 53–67. Springer, 2021. doi:10.1007/978-3-030-81682-7_4.

[97] D. Unat, C. Chan, W. Zhang, S. Williams, J. Bachan, J. Bell, and J. Shalf. ExaSAT: An Exascale Co-design Tool for Performance Modeling. *The International Journal of High Performance Computing Applications*, 29(2):209–232, 2015. doi:10.1177/1094342014568690.

[98] N. Weber. SOL: Reducing the Maintenance Overhead for Integrating Hardware Support into AI Frameworks. *arXiv preprint arXiv:2205.10357*, 2022. doi:10.48550/arXiv.2205.10357.

[99] N. B. Wilding, A. Trew, K. Hawick, and G. Pawley. Scientific Modeling with Massively Parallel SIMD Computers. *Proceedings of the IEEE*, 79(4): 574–585, 1991. doi:10.1109/5.92050.

[100] S. Williams, A. Waterman, and D. Patterson. Roofline: an insightful visual performance model for multicore architectures. *Communications of the ACM*, 52(4):65–76, 2009. doi:10.1145/1498765.1498785.

[101] M. Wittmann, G. Hager, T. Zeiser, J. Treibig, and G. Wellein. Chip-level and Multi-node Analysis of Energy-optimized Lattice Boltzmann CFD Simulations. *Concurrency and Computation: Practice and Experience*, 28(7): 2295–2315, 2016. doi:10.1002/cpe.3489.

[102] Y. Yamada and S. Momose. Vector Engine Processor of NEC's Brand-new Supercomputer SX-Aurora TSUBASA. In *Proceedings of A Symposium on High Performance Chips, Hot Chips*, volume 30, pages 19–21, 2018.

[103] C. Yang, R. Gayatri, T. Kurth, P. Basu, Z. Ronaghi, A. Adetokunbo, B. Friesen, B. Cook, D. Doerfler, L. Oliker, et al. An Empirical Roofline Methodology for Quantitatively Assessing Performance Portability. In *2018 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*, pages 14–23. IEEE, 2018. doi:10.1109/P3HPC.2018.00005.

[104] C. Yang, T. Kurth, and S. Williams. Hierarchical Roofline analysis for GPUs: Accelerating performance optimization for the NERSC-9 Perlmutter system. *Concurrency and Computation: Practice and Experience*, 32 (20):e5547, 2020. doi:10.1002/cpe.5547.

List of Abbreviations

ALU	Arithmetic Logic Unit
AMT	Abstract Mapping Tree
APT	Abstract Pattern Tree
AVX	Advanced Vector Extensions
DIV/SQRT	Division/Square-Root
DSL	Domain Specific Language
ECM	Execution-Cache-Memory
FMA	Fused Multiply-Add
FPGA	Field Programmable Gate Array
HBM	High Bandwidth Memory
HCA	Host Channel Adapter
HL	Hardware Language
K-Means	K-Means clustering
LLC	Last Level Cache
ML	Machine Learning
NN	K-Nearest neighbors
PPL	Parallel Pattern Language
SIMD	Single Instruction Multiple Data
SIMT	Single Instruction Multiple Threads
SISD	Single Instruction Single Data
SoC	System on a Chip
SPU	Scalar Processing Unit
VA	Vector Accelerator
VE	Vector Engine
VEOS	Vector Engine Operating System
VH	Vector Host
VP	Vector Processor
VPU	Vector Processing Unit