Informatics

pSTL-Bench

Evaluating the Capabilities of ISO C++ Parallel STL Implementations on Modern Parallel Hardware using Microbenchmarking

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering und Internet Computing

eingereicht von

Diego Krupitza, BSc
Matrikelnummer 11808206

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Associate Prof. Dipl.-Inform. Dr.rer.nat. Sascha Hunold

Wien, 23. Juni 2023

Diego Krupitza

Sascha Hunold

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

Informatics

pSTL-Bench

Evaluating the Capabilities of ISO C++ Parallel STL Implementations on Modern Parallel Hardware using Microbenchmarking

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering and Internet Computing

by

Diego Krupitza, BSc
Registration Number 11808206

to the Faculty of Informatics

at the TU Wien

Advisor: Associate Prof. Dipl.-Inform. Dr.rer.nat. Sascha Hunold

Vienna, 23rd June, 2023

Diego Krupitza | Sascha Hunold

Erklärung zur Verfassung der Arbeit

Diego Krupitza, BSc

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe. Des Weiteren führe ich an, dass ich für die Korrektur und für die Verbesserung meines Textes die Tools Grammarly und ChatGPT verwendet habe. Wie genau ChatGPT verwendet wurde, kann im Appendix dieser Arbeit entnommen werden.

Wien, 23. Juni 2023

Diego Krupitza

Acknowledgements

I would like to express my heartfelt gratitude to the following individuals who have played an instrumental role in my journey, providing unwavering support and guidance.

First and foremost, I extend my deepest appreciation to my family. Their unwavering belief in me and their continuous support both emotionally and financially have been the pillars of my success. Without them, I would have never come that far! They have always stood behind me, encouraging me to pursue my goals and dreams. I am truly grateful for their love and encouragement.

I would like to give special thanks to my older sister, who courageously paved the way through the challenges of school and university before me. She courageously faced and conquered the challenges of school and university, gaining invaluable experience along the way. As a result, she was able to provide me with invaluable guidance and support, making the entire process significantly easier for me. I consider myself incredibly fortunate to have such a supportive and inspiring sibling by my side.

I am immensely grateful to my advisor, Professor Sascha Hunold, for his exceptional guidance. His dedication, expertise, and responsiveness have been remarkable. Professor Hunold's availability, even during late hours, made me feel like I had access to "24/7 premium technical support". His support has been invaluable, and contributed significantly to the completion of this thesis.

Finally, I want to extend my gratitude to all my friends and colleagues whom I had the pleasure of meeting during my time at the university. Their companionship and camaraderie have made the study experience truly memorable. Countless hours spent together, studying for exams and engaging in stimulating discussions, have not only enhanced my learning but also created a sense of community that made the journey all the more meaningful.

To everyone mentioned above, and to those who have supported me in various other ways throughout my academic pursuits, I offer my sincerest appreciation. Your belief in me, encouragement, and assistance have made a profound impact on my personal and professional growth. I am grateful for the role each of you has played in shaping my path to success.

Kurzfassung

Hochleistungscomputer bestehen heutzutage aus einer Vielzahl von Plattformen, die von Multi-Core CPUs bis zu Many-Core GPUs reichen. Folglich möchten Entwickler eine einzige Codebasis haben, die auf einer Vielzahl von Beschleunigern und Plattformen läuft und die höchstmögliche Leistung erzielt. Leistungsportabilitäts-Frameworks wie RAJA, Kokkos und SYCL ermöglichen dies den Entwicklern, allerdings mit dem Nachteil abbhängig von einem nicht standardisierten Framework zu sein. Die parallele STL von ISO C++ ermöglicht es Entwicklern, standardisierten C++ Code zu schreiben und während der Kompilierung zu entscheiden, welcher Beschleuniger verwendet werden soll. Diese Eigenschaft macht ISO C++ parallel STL zu einer potentielle Alternative zu Leistungsportabilitäts-Frameworks. Derzeit gibt es einige Implementierungen für die ISO C++ Parallel STL (auch als Backends bekannt). In dieser Arbeit haben wir die Fähigkeiten von drei ISO C++ parallelen STL-Backends untersucht, nämlich GCC mit oneTBB, NVC mit OpenMP und NVC mit CUDA. Um ihre Fähigkeiten zu bewerten, haben wir eine Mikrobenchmark-Suite namens pSTL-Bench entwickelt. Sie ermöglicht das Benchmarking von parallelen STL-Primitiven und bietet wertvolle Einblicke in die Leistungs- und Skalierungseigenschaften der Backends. Mit Hilfe dieser Benchmarks konnten wir erhebliche Unterschiede in Bezug auf Leistung und Skalierbarkeit zwischen den Backends feststellen, die vielleicht nicht auf den ersten Blick erkennbar sind. Als Nebenprodukt des umfangreichen Benchmarkings können wir die Leistungsportabilität zwischen diesen Backends quantifizieren. Während bestimmte parallele Primitive eine hohe Leistungsportabilität aufweisen, gibt es andere, bei denen ein einfacher Wechsel des Backends zu einer erheblichen Leistungssteigerung führt, was zu einer niedrigen Portabilitätsbewertung führt. Um zu zeigen, dass die ISO C++ Parallel STL tatsächlich verschiedene Beschleuniger unterstützt, haben wir die Mikrobenchmark-Suite auf einen Nvidia Tesla T4-Grafikprozessor ausgelagert und dabei das NVC mit CUDA-Backend (auch bekannt als stdpar) verwendet. Obwohl viele der Benchmarks ohne zusätzliche Änderungen auf dem Grafikprozessor ausgeführt werden können, gab es bei anderen Benchmarks eine Reihe von Herausforderungen. Diese Herausforderungen spiegeln sich in unserem abschließenden Urteil über die Fähigkeiten der ISO C++ parallel STL wider. Obwohl die ISO C++ parallel STL einen guten Ausgangspunkt für das Schreiben von leistungsfähigem ISO C++ Code bietet, gibt es noch viele Probleme, die in zukünftigen Versionen behoben werden müssen.Die parallele STL von ISO C++ ist eine einfache Schnittstelle zum Schreiben von portierbarem Hochleistungscode, aber es ist noch ein weiter Weg bis zu echter Leistungsportabilität.

Abstract

High-performance computers nowadays consist of a wide variety of architectures, ranging from multi-core CPUs to many-core GPUs. As a result, developers want to have a single code base that runs on a wide variety of accelerators and platforms and achieves the highest possible performance. Performance portability frameworks such as RAJA, Kokkos, and SYCL allow developers to do this, but at the cost of having a non-standardized third-party dependency. ISO C++ parallel STL allows developers to write standardized C++ code and decide at compile time what accelerator to target, posing as a potential alternative to those frameworks. At the moment there are several implementations (also known as Backends) for ISO C++ parallel STL. In this thesis, we evaluated the capabilities of three ISO C++ parallel STL backends, namely GCC with oneTBB, NVC with OpenMP and NVC with CUDA. To assess their capabilities, we developed a microbenchmark suite called pSTL-Bench. It allows benchmarking of parallel STL primitives and provides valuable insights into the performance and scaling characteristics of the backends. Through these benchmarks, we discovered significant differences in terms of performance and scalability among the backends, which might not be immediately apparent. As a side product of the extensive benchmarking, we can quantify performance portability between those backends. While certain parallel primitives exhibit high performance portability, there are others where simply switching the backend yields a considerable performance boost, resulting in a low-performance portability score. To demonstrate that ISO C++ parallel STL indeed supports different accelerators, we offloaded the microbenchmark suite to an Nvidia Tesla T4 GPU using the NVC with CUDA backend (also known as stdpar). Although many of the benchmarks run without any additional changes on the GPU, there were a substantial amount of challenges for other benchmarks. These challenges are reflected in our final verdict about the capabilities of ISO C++ parallel STL. Although ISO C++ parallel STL does provide a good starting point for writing performance portable ISO C++ code, there are still many hiccups and problems that need to be addressed in future releases. ISO C++ parallel STL is a simple interface for writing high-performance portable code, but it still has a long way to go to enable true performance portability.

Contents

List of Figures

List of Tables

CHAPTER 1

Introduction

Parallelization is a technique that enables programs to speedup expensive workloads. Over the years, specialized accelerators for parallel computing, such as GPUs, have been introduced, which can achieve even higher speedups. Many of the TOP500 supercomputers nowadays consist of both multi-core CPUs and many-core GPUs [2], making it desirable for high performance application to work with both targets.

The use of specialized accelerators resulted in having diverse programming models, including OpenMP for multi-core systems and CUDA for Nvidia GPUs. As a consequence, switching between accelerators or computation platforms requires rewriting the entire program, as a different programming models is needed. Furthermore, certain programming paradigms, such as CUDA, are exclusive to specific vendors, resulting in vendor lock-in, which complicates changing computation platforms in the future [3].

Developers want to write code that can be compiled and executed across multiple platforms while maintaining high performance. This concept is known as performance portability. Although the term *performance portability* has been used in various research papers, there is no concrete definition in the community [4]. From a developer's perspective, the primary objective of performance portability is to create a single program that can be deployed on different platforms, architectures, and accelerators without requiring any modifications. However, the program should still perform efficiently on each platform, without the need to tailor it to specific targets. Essentially, developers want to write a single codebase that can run efficiently on all platforms, without sacrificing performance.

From a conceptional point of view *performance portability* seems rather easy but due to the major differences between platforms, accelerators and programming paradigms, it is a rather hard problem. To overcome this problem, frameworks such as Raja [5], Kokkos [6], SYCL [7], HIP [8], and HPX [9] were introduced. These frameworks promise performance portability across various parallel architectures. However, the concepts and ideas change from one framework to another. Hence, switching between frameworks is also not straightforward, and porting large codebases from one framework to another can be a lot of work [2].

The ISO C++ parallel STL may be one possible alternative to the performance portability frameworks presented above. C++17 introduced parallel algorithms as a standardized interface for commonly used parallel primitives. Developers can write standard ISO C++ code and can decide at compile time what parallel paradigm and accelerator to use. Therefore, a developer can work with familiar ISO C++ code and easily change between various parallel backends and platforms. The parallel backend developer has the responsibility to handle programming paradigm-specific code and can optimize it to enhance efficiency and performance, which is abstracted away from the user.

There exist several parallel backends, such as the one offered by GCC, that utilizes Intel's OneAPI or the parallel backend provided by Nvidia, which uses OpenMP or CUDA.

However, there are still open questions about parallel algorithms in C++, and the backends responsible for creating the parallel implementation. These include how various parallel algorithms (and their backends) scale on modern parallel architectures, whether there is a difference between parallel backends in terms of performance (e.g., measuring performance portability), whether leveraging newer C++ concepts improves performance, and whether there are best practices when writing efficient parallel C++ code.

Being able to find answers, or even partial answers, to the questions surrounding parallel algorithms not only helps developers to create high-performance applications more easily and to achieve high performance on a given platform, but it also provides valuable performance data to the developers of parallel backends such as stdlibc++ (used by GCC) or Nvidia. This data can be used to improve the backends and make them even more efficient, ultimately leading to better performance for developers who use them.

1.1 Research Questions

Due to the driving force behind performance portability and the recent introduction of the ISO C++ parallel STL, we have identified several research questions that we aim to address in this thesis:

1. How well do the C++ parallel standard library (parallel STL) implementations scale on current parallel architectures?
2. Does a performance difference exist when C++ features exclusively available in C++-20 are enabled as compared to previous versions?
3. Which primitives should be used to achieve high-performance for typical parallel patterns? For example, when changing the contents of a vector, we could employ the `std::for_each` or `std::transform` primitive. But which one is better in which case? Can we provide simple performance guidelines?

4. Since we generally need to decide at compilation time whether we want to compile for multicore processors or for GPUs, the question becomes if we can determine the sweet-spot between these two options in terms of performance, i.e., can we predict (model) when the use of the GPU will be beneficial?

5. How can we compare the performance of parallel applications in ISO C++ on different parallel machines, i.e., how can we quantify the performance portability for these codes?

1.2 Structure of the Thesis

In Chapter 2, we provide a short overview of important background topics that have to be first addressed in order to establish the context of this thesis. This is followed by the related works in Chapter 3 where we present previous works of performance portability and how it can be quantified. In Chapter 4, we give a high level overview of ISO C++ parallel STL and the performance portability frameworks SYCL, Kokkos and RAJA. In Chapter 5, we explain our approach and environment to answer the research questions outlined in this thesis. We then introduce our novel benchmarking suite in Chapter 6. In Chapter 7, we present our findings from the experimental evaluation conducted with the benchmarking suite, and reflect on the key observations. Finally, in Chapter 8, we attempt to quantify performance portability for two parallel backends and discuss the current state of performance portability using the results gathered during the experimental evaluation. Chapter 9 provides a summary of the findings and insights gained throughout the study. We also present an outlook on future work, identifying areas where further research could be conducted to expand on the findings.

CHAPTER 2

Background

This chapter addresses several important topics that are relevant not only to the field of High Performance Computing but also to the contents of this thesis. It should be noted that the chapter is not meant to be exhaustive, as this would exceed the scope of the thesis.

2.1 Parallel Machines

The term *parallel machine* is quite ambiguous. Already in early works [10], the term parallel machine was used to describe machines with multiple processors, array-type machines, or multiprogrammed machines. Although there are a lot of types of parallel machines used in research, we limit the term parallel machine in this work to Multiprocessor systems and Graphics Processing Units (GPUs). In the following, we discuss those two types of parallel machines in more detail.

2.1.1 Multiprocessor Systems

Although personal workstation computers used by humans on a daily basis are sufficient for tasks such as writing documents or browsing the web, they are not suitable for high performance applications [11]. High-performance applications require parallelism beyond what traditional workloads have demanded in the past, making multiprocessor systems preferable due to their high level of parallelism.

Shared-Memory Multiprocessor

The Shared-Memory Multiprocessor is the most prominent type of multiprocessing system. In this system, the processing units all have direct hardware access to the whole memory region and share a common address space [12, 13]. Shared-Memory Multiprocessor systems differ mainly by the point of how memory access is organized. In general, there are two architectures which are Unified Memory Access and Non-Unified Memory Access.

Unified Memory Access Unified Memory Access (UMA) offers the significant advantage of providing identical memory access times to each processor within a computer system, as stated in [12]. However, it is important to note that in UMA systems with a single memory controller, there may be some variation in memory access times when multiple processes attempt to access the same memory bank. Despite this limitation, UMA's unique characteristics make it an ideal memory architecture for small to medium purpose machines, including personal computers and workstations [12].

Non-Uniform Memory Access Non-Uniform Memory Access (NUMA) is a technology commonly utilized in high-performance computing due to its ability to offer faster access times compared to Uniform Memory Access (UMA) [14]. NUMA divides memory and processing cores into nodes, with each node consisting of a set of cores and associated memory. Figure 2.1 depicts the NUMA topology of Nebula, a high-performance computer with eight NUMA nodes, each containing eight cores. Accessing memory within a node, also known as node-local memory access, is the fastest and is always the equal time [14, 15]. However, if a core needs to access memory from a remote NUMA node, the access time significantly increases, which leads to performance degradation since it has to access the system bus [14]. For instance, Core 0 in Nebula can access memory from *NUMA node P#0* with the fastest access time. However, if Core 0 attempts to access memory from any other NUMA node, performance will suffer. This limitation is a significant drawback, and programs running on NUMA hardware must be NUMA-aware by first minimizing remote node access and by being mindful of data placement. It is worth noting that NUMA awareness only affects program performance, not its correctness [15].

Massively Parallel Processors

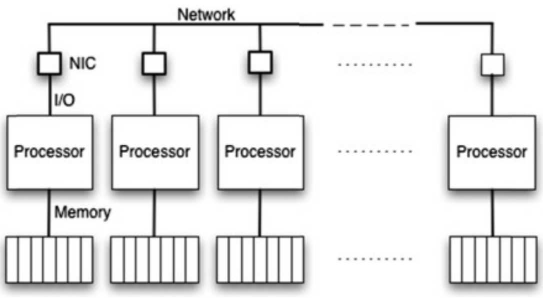


Figure 2.2: Architecture of Massively Parallel Processors [12].

Multiprocessor systems have a limit to the number of processing units that can be placed on a single board. Massively Parallel Processors provide a solution to this problem. They consist of multiple smaller machines, which can be Multiprocessor Systems or not, called

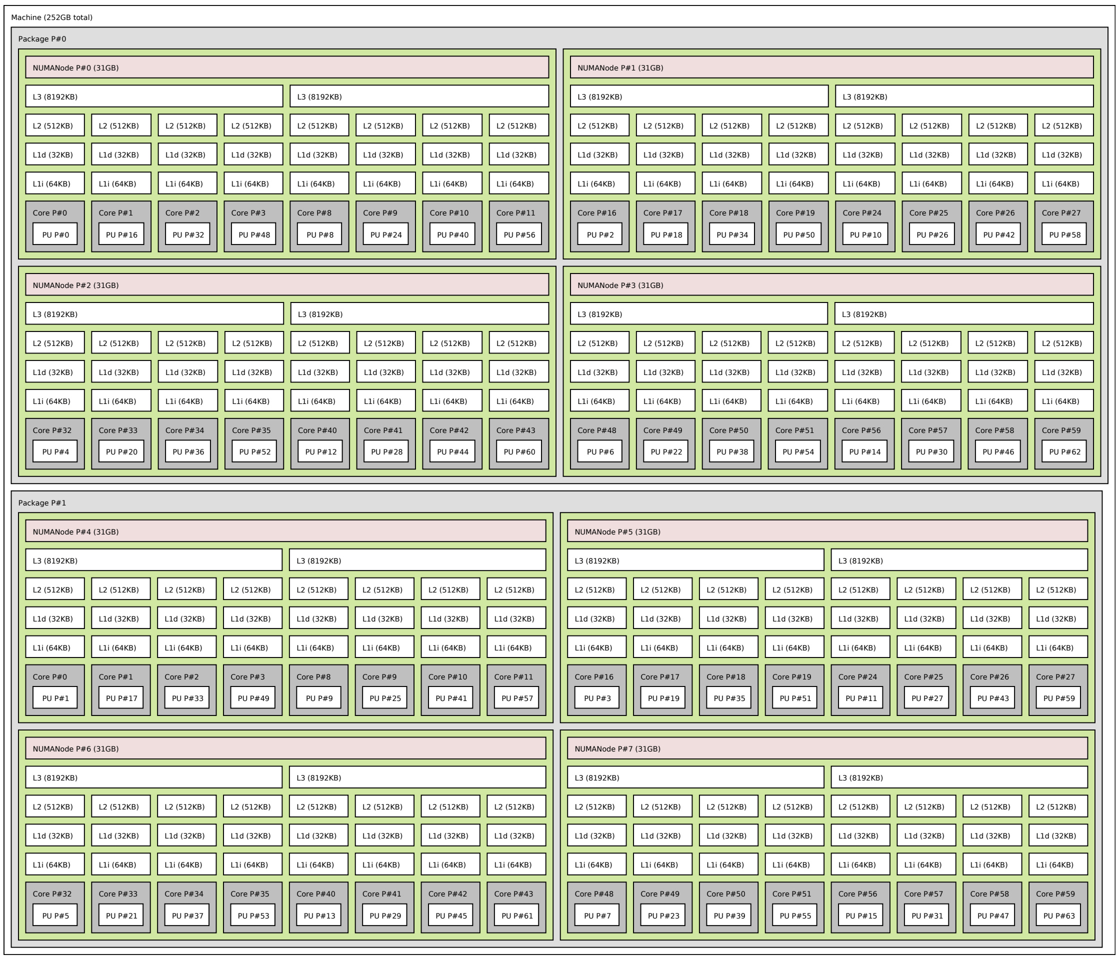

Machine (252GB total)
Package P#0
NUMANode P#0 (31GB)
L3 (8192KB) L3 (8192KB)
L2 (512KB) L2 (512KB) L2 (512KB) L2 (512KB) L2 (512KB) L2 (512KB) L2 (512KB) L2 (512KB)
L1d (32KB) L1d (32KB) L1d (32KB) L1d (32KB) L1d (32KB) L1d (32KB) L1d (32KB) L1d (32KB)
L1i (64KB) L1i (64KB) L1i (64KB) L1i (64KB) L1i (64KB) L1i (64KB) L1i (64KB) L1i (64KB)
Core P#0 Core P#1 Core P#2 Core P#3 Core P#8 Core P#9 Core P#10 Core P#11
PU P#0 PU P#16 PU P#32 PU P#48 PU P#8 PU P#24 PU P#40 PU P#56
NUMANode P#1 (31GB)
L3 (8192KB) L3 (8192KB)
L2 (512KB) L2 (512KB) L2 (512KB) L2 (512KB) L2 (512KB) L2 (512KB) L2 (512KB) L2 (512KB)
L1d (32KB) L1d (32KB) L1d (32KB) L1d (32KB) L1d (32KB) L1d (32KB) L1d (32KB) L1d (32KB)
L1i (64KB) L1i (64KB) L1i (64KB) L1i (64KB) L1i (64KB) L1i (64KB) L1i (64KB) L1i (64KB)
Core P#16 Core P#17 Core P#18 Core P#19 Core P#24 Core P#25 Core P#26 Core P#27
PU P#2 PU P#18 PU P#34 PU P#50 PU P#10 PU P#26 PU P#42 PU P#58
NUMANode P#2 (31GB)
L3 (8192KB) L3 (8192KB)
L2 (512KB) L2 (512KB) L2 (512KB) L2 (512KB) L2 (512KB) L2 (512KB) L2 (512KB) L2 (512KB)
L1d (32KB) L1d (32KB) L1d (32KB) L1d (32KB) L1d (32KB) L1d (32KB) L1d (32KB) L1d (32KB)
L1i (64KB) L1i (64KB) L1i (64KB) L1i (64KB) L1i (64KB) L1i (64KB) L1i (64KB) L1i (64KB)
Core P#32 Core P#33 Core P#34 Core P#35 Core P#40 Core P#41 Core P#42 Core P#43
PU P#4 PU P#20 PU P#36 PU P#52 PU P#12 PU P#28 PU P#44 PU P#60
NUMANode P#3 (31GB)
L3 (8192KB) L3 (8192KB)
L2 (512KB) L2 (512KB) L2 (512KB) L2 (512KB) L2 (512KB) L2 (512KB) L2 (512KB) L2 (512KB)
L1d (32KB) L1d (32KB) L1d (32KB) L1d (32KB) L1d (32KB) L1d (32KB) L1d (32KB) L1d (32KB)
L1i (64KB) L1i (64KB) L1i (64KB) L1i (64KB) L1i (64KB) L1i (64KB) L1i (64KB) L1i (64KB)
Core P#48 Core P#49 Core P#50 Core P#51 Core P#56 Core P#57 Core P#58 Core P#59
PU P#6 PU P#22 PU P#38 PU P#54 PU P#14 PU P#30 PU P#46 PU P#62
Package P#1
NUMANode P#4 (31GB)
L3 (8192KB) L3 (8192KB)
L2 (512KB) L2 (512KB) L2 (512KB) L2 (512KB) L2 (512KB) L2 (512KB) L2 (512KB) L2 (512KB)
L1d (32KB) L1d (32KB) L1d (32KB) L1d (32KB) L1d (32KB) L1d (32KB) L1d (32KB) L1d (32KB)
L1i (64KB) L1i (64KB) L1i (64KB) L1i (64KB) L1i (64KB) L1i (64KB) L1i (64KB) L1i (64KB)
Core P#0 Core P#1 Core P#2 Core P#3 Core P#8 Core P#9 Core P#10 Core P#11
PU P#1 PU P#17 PU P#33 PU P#49 PU P#9 PU P#25 PU P#41 PU P#57
NUMANode P#5 (31GB)
L3 (8192KB) L3 (8192KB)
L2 (512KB) L2 (512KB) L2 (512KB) L2 (512KB) L2 (512KB) L2 (512KB) L2 (512KB) L2 (512KB)
L1d (32KB) L1d (32KB) L1d (32KB) L1d (32KB) L1d (32KB) L1d (32KB) L1d (32KB) L1d (32KB)
L1i (64KB) L1i (64KB) L1i (64KB) L1i (64KB) L1i (64KB) L1i (64KB) L1i (64KB) L1i (64KB)
Core P#16 Core P#17 Core P#18 Core P#19 Core P#24 Core P#25 Core P#26 Core P#27
PU P#3 PU P#19 PU P#35 PU P#51 PU P#11 PU P#27 PU P#43 PU P#59
NUMANode P#6 (31GB)
L3 (8192KB) L3 (8192KB)
L2 (512KB) L2 (512KB) L2 (512KB) L2 (512KB) L2 (512KB) L2 (512KB) L2 (512KB) L2 (512KB)
L1d (32KB) L1d (32KB) L1d (32KB) L1d (32KB) L1d (32KB) L1d (32KB) L1d (32KB) L1d (32KB)
L1i (64KB) L1i (64KB) L1i (64KB) L1i (64KB) L1i (64KB) L1i (64KB) L1i (64KB) L1i (64KB)
Core P#32 Core P#33 Core P#34 Core P#35 Core P#40 Core P#41 Core P#42 Core P#43
PU P#5 PU P#21 PU P#37 PU P#53 PU P#13 PU P#29 PU P#45 PU P#61
NUMANode P#7 (31GB)
L3 (8192KB) L3 (8192KB)
L2 (512KB) L2 (512KB) L2 (512KB) L2 (512KB) L2 (512KB) L2 (512KB) L2 (512KB) L2 (512KB)
L1d (32KB) L1d (32KB) L1d (32KB) L1d (32KB) L1d (32KB) L1d (32KB) L1d (32KB) L1d (32KB)
L1i (64KB) L1i (64KB) L1i (64KB) L1i (64KB) L1i (64KB) L1i (64KB) L1i (64KB) L1i (64KB)
Core P#48 Core P#49 Core P#50 Core P#51 Core P#56 Core P#57 Core P#58 Core P#59
PU P#7 PU P#23 PU P#39 PU P#55 PU P#15 PU P#31 PU P#47 PU P#63

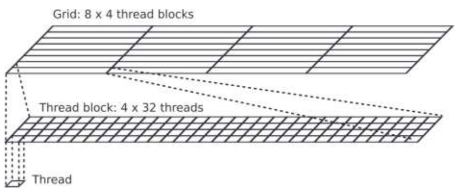
Figure 2.1: NUMA Node Topology of Nebula.

nodes. These nodes communicate with each other over a high-bandwidth network since they do not share memory [12, 16], as shown in Figure 2.2. It is worth noting that we only mention this here for completeness' sake and will not consider this type of machine in this thesis.

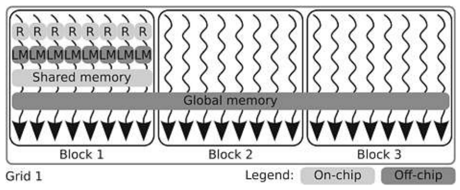
2.1.2 GPUs

Graphics Processing Units (GPUs) have revolutionized the High Performance Computing landscape [17]. GPUs were originally used for gaming or computer graphics, but over the years slowly emerged as a "versatile platform for running massively parallel computations" [18] and now have applications ranging from artificial intelligence (AI) to Internet of Things (IoT) and Autonomous driving [17]. GPUs are beneficial for high performance computing applications because of their high bandwidth, high computation throughput, and excellent support for floating point arithmetic [18, 19]. In order to run code on

GPUs, one is required to work with a programming paradigm that is understood by the target GPU. There is a wide variety of options such as Nvidia CUDA or OpenCL. Those programming paradigms allow full control over the GPU to get the most performance out of it.



(a) Thread hierarchy [17]



(b) Thread and Memory hierarchy [18]

Figure 2.3: Logical representation of GPUs.

The concrete architecture of GPUs rapidly evolved, therefore presenting a current architecture of a real GPU will be outdated in short time. To still explain the concepts and ideas behind the architecture of a GPU, Hijma et al. [17] presented a representative architecture of a GPU shown in Figure 2.3a. As one can see the smallest processing unit in a GPU is a thread. Threads are grouped into so-called thread blocks and finally grouped into grids. Current GPUs contain hundreds of such independent execution units [19].

The memory layout of GPUs reflect the processing unit hierarchy [18], as we can see in Figure 2.3b. The global memory is shared across all the threads. Then there is the shared memory, which, as the name suggests, is shared among the threads inside a block. Finally, each thread has its own local memory and registers. It is important to mention that the further away the memory the higher is the latency, meaning local memory is the fastest and global memory the slowest [17, 18].

2.2 Comparing Parallel Benchmarks

When comparing parallel benchmarks, there are quite often certain metrics involved in the discussion of the results. In this section, we highlight a selection of metrics and terminologies important for this thesis and discuss them in a more detailed way. It is important to acknowledge that the provided list is not intended to be exhaustive.

Strong Scaling is a scaling analysis where the input size of the problem stays the same and only the number of processing units (e.g., threads) is increased. Strong scaling is a reliable technique to find out how many processing elements can be efficiently utilized. A perfect strong scaling behavior would be that by doubling the number of cores the runtime decreases also by the same rate.

Weak Scaling The counterpart of strong scaling, is weak scaling. Here, the number of threads and the problems size are proportionally increased [20]. Due to the fact that both sizes increase at the same rate, the workload per processor stays the same [21].

Speedup The speedup is a metric to measure the performance of a parallel implementation. It measures the ratio between the serial and the parallel implementation [22]. The formula for speedup is $S(p) = \frac{T(1)}{T(p)}$, where p is the number of processing units and T the runtime using p processing units [16].

Efficiency The parallel efficiency is defined as the ratio between the speedup and the number of processing units used. The formula to calculate the parallel efficiency is $E(p) = \frac{S(p)}{p}$. Efficiency allows us to understand how well the processing units are used [22]. Although efficiency is usually in the range $(0, 1]$, it is possible to achieve an efficiency greater than one. This is called super linear speedup. Reasons behind this could be that the work performed by the sequential algorithm is greater than the parallel or due to hardware features (e.g., data fits into the cache perfectly) [22].

CHAPTER 3

Related Work

As the demand for high-performance computing (HPC) applications continues to grow, researchers have been actively developing and exploring various solutions and proposals to address the challenges of achieving performance portability across different hardware architectures and platforms [4].

Code portability and performance portability are critical for HPC systems. Writing code in multiple frameworks or paradigms for each new hardware platform can be time-consuming and difficult to maintain [2]. The challenge is even greater when we must manage multiple programs that accomplish the same goal but run on different hardware [4]. Additionally, the greater the heterogeneity of the architecture, the more undesirable it is to have architecture-specific implementations, since the program's lifespan becomes constrained by the frameworks or paradigms lifespan [23].

In this chapter, we aim to provide a concise yet informative overview of the related works in the field of performance portability in high-performance computing. We briefly examine how researchers have evaluated performance portability and highlight the shortcomings we have identified in current works. We start with, the main objective, performance portability using C++ in Section 3.1, followed by performance portability in other languages in Section 3.2. Finally, we present recent works covering performance portability metrics in Section 3.3.

3.1 Performance Portability with C++

Dufek et al. [2] presented several programming models, including CUDA, HIP, SYCL, and Kokkos, and developed a parallel-friendly implementation of the Milc-Dslash benchmark to analyze performance portability across different GPU platforms (Nvidia, AMD, and Intel). The Milc-Dslash benchmark, simulates the lattice quantum chromodynamics theory. The computation found in this benchmark is, as highlighted by the authors, representative of typical applications that solve partial differential equations. Because the goal is to highlight the performance portability of the frameworks Kokkos and SYCL the parallel-friendly implementation is not tailored to a specific GPU architecture or vendor. Instead, they followed commonly known best practices of GPU programming to achieve the highest possible memory bandwidth, such as minimizing the global memory access and increasing the number of Floating Point Operation (FLOPS). As performance metric, the authors used the raw speed-up in respect to the sequential implementation. The results showed that the performance portability between Kokkos and SYCL was satisfactory on all platforms, with nearly identical performance to the reference implementations in platform-specific code. The main contribution of this paper is to showcase that in fact there is performance portability between different frameworks and how to measure it explicitly for GPUs.

Lin et al. [24] compared C++17 parallel STL to other frameworks, including OpenMP, Kokkos, and TBB. To compare each paradigm, they ported a set of benchmarks to C++17 using parallel STL. The study examines three different mini-applications, which are Babelstream, CloverLeaf, and miniBUDE. Babelstream and Cloverleaf are memory-bound mini-applications, enabling them to benchmark typical memory-bound operations found in HPC programs. The application miniBUDE is a computation-bound mini-application, allowing the comparison of absolute runtime differences. The C++17 parallel STL implementations are designed to be both data-centric and index-centric, since some mini-applications used in their benchmarks require stencil operations which need indices. Since C++17 does not provide an interface for index-centric traversal, they had to implement their own iterator. The index-centric implementations are similar to the one found in the works of Olsen et al. [25] where they ported the LULESH hydrodynamics mini-app to C++ (using Nvidia's C++ parallel STL backend). The benchmarks were executed on a wide range of shared memory machines (including NUMA and UMA systems) and GPUs. The performance metrics used for memory-bound benchmarks were peak memory bandwidth, while for computational-bound benchmarks, the runtime was reported. The results indicate that there is minimal overhead from C++ parallel algorithms to their backends. Additionally, the performance between all compilers on various machines was found to be comparable.

Mietzsch and Fuerlinger [1] implemented the NAS Parallel Benchmark kernels using C++17 parallel algorithms and compared the results to OpenMP and TBB (now known as oneTBB) implementations. Their implementation was based on a serial implementation by Griebler et al. [26]. The NAS Parallel Benchmark [27] is a suite of benchmarks that includes kernels which simulate typical workloads of computational fluid dynamics and aero science applications. As seen in Table 3.1, the kernels rely heavily on the parallel building block `std::for_each`, with only one of the kernels using a wider variety of parallel algorithms. Furthermore, most of their benchmarks use an index-centric approach where `std::iota` generates the indices. As performance metric, the authors used the raw speed-up in respect to the sequential implementation. Mietzsch and Fuerlinger found that for most of the kernels, the performance difference between native implementations

Table 3.1: Time spent in each parallel algorithm adapted from [1].

	EP	MG	CG	FT	IS
`std::for_each`	100%	99.69%	99.55%	100%	100%
`std::transform_reduce`	0%	0.31%	0.08%	0%	0%
`std::transform`	0%	0%	0.21%	0%	0%
`std::copy`	0%	0%	0.12%	0%	0%
`std::fill`	0%	0%	0.04%	0%	0%

(OpenMP and TBB) and C++17 parallel STL was small. However, for one kernel, special techniques with OpenMP resulted in higher performance than what was achievable using only C++17 parallel algorithms, due to the limitations of parallel STL.

Asahi et al. [23] conducted a comprehensive analysis of performance portability using the Nvidia parallel algorithm implementation, `stdpar`, which supports both Nvidia GPUs and multicore systems. They compared its performance with that of Thrust, Kokkos, and OpenMP using a kinetic plasma simulation mini-app (*heat-functor*) as a benchmark suite. The study aims to analyze not only performance, but also the other two key aspects that developers seek, which are portability and productivity. To achieve this, they provided a tailored overview of the implementation differences between `stdpar` and the other frameworks used in the benchmark suite. The C++ parallel STL implementation uses an index-centric approach similar to the ones found in the works of Lin et al. [24] and Olsen et al [25]. Since the mini-app is rather memory intensive, the authors used peak memory bandwidth as a performance metric. The results showed that the productivity and portability were satisfactory, and the performance of `stdpar` was competitive with the reference implementations in Kokkos, Thrust, and OpenMP.

The C++ parallel STL is a powerful tool for achieving higher performance of parallel algorithms on a single machine. However, its potential is not limited to this use case. Recent research by Drocco et al. [28] has demonstrated that it is possible to extend the capabilities of the C++ parallel STL to enable the execution of distributed parallel STL programs. By adding their own execution policy, Drocco et al. were able to create a custom backend that adheres to the C++ parallel STL interface, but enables the program to be executed across multiple machines. The user experience remains seamless, as the backend operates in the same way as the standard C++ parallel STL. In addition to the convenience of a familiar interface, the performance benefits of the distributed parallel STL are significant. A quick benchmark of commonly used algorithms, such as `std::reduce`, demonstrated linear scaling of the program across multiple machines.

To our knowledge, current research limits itself to evaluating performance portability between C++ and other performance portability frameworks such as Kokkos and Friends. There is no in-depth performance portability study on C++ parallel algorithms and those backends that provide this functionality. Furthermore, current research focuses on mini-apps and kernels, which still introduces a lot of noise, not allowing to benchmark the smallest building blocks, the parallel STL primitives. It has been observed that the existing research in C++ parallel STL primarily focuses on the primitives `std::for_each` and `std::transform`, leaving a wider range of parallel primitives unexplored. This gap in research presents a challenge for analyzing the effectiveness of specific parallel algorithms such as `std::copy_if` and whether it is more advantageous to implement the same logic using alternative primitives like `std::for_each`. Furthermore, there is a need to compare different strategies for index-based access on containers to identify the most efficient approach, since every paper uses a different approach.

3.2 Performance Portability with Other Languages

As the demand for performance portability continues to grow, it is not limited to just the C++ language. Kokkos, a widely used performance portability framework written in C++, has been leveraged by Al Awar et al. [29] to create a Python framework that enables Python developers to also write performance portable code. This allows developers who prefer Python for its efficiency in rapid prototyping and research focus to still achieve performance portability. Moreover, Al Awar et al, ported a set of Kokkos applications to their new Python framework and found that the raw runtime had only a small overhead. The limitation, however, is that the Python API only supports a subset of Kokkos features, and it still requires Python developers to think like Kokkos developers.

C++ can be a challenging language to work with in production due to its undefined behavior, as noted by Lin and McIntosh-Smith (2021) [30]. In recent years, Julia has gained considerable popularity in the high-performance computing community. To demonstrate the performance portability between Julia and C++, Lin and McIntosh-Smith (2021) ported a set of mini-apps and compared the results with those achieved using OpenMP, Kokkos, CUDA, OpenCL, and SYCL. The authors found that Julia's performance is either comparable to or slightly worse than the reference implementation. To enable GPU offloading, the authors emphasized the need for third-party packages. While the interface for CPU and GPU offloading differs, performance portability to some extent exists for GPUs, with little or no code changes required. Thanks to Julia's API consistency, developers can rely on a shared interface for all major GPU offloading packages [31].

3.3 Performance Portability Metrics

Pennycook et al. introduced a metric for performance portability in their earlier work [4] and revised it in a later publication [32] in response to ongoing discussions in the research community. The performance portability Φ (Equation (3.1)) proposed is defined as the harmonic mean of an application's performance efficiency calculated for a set of platforms. H denotes a set of platforms, a an application which solves the problem p, and, $e_i(a, p)$ denotes the performance efficiency of an application a on platform i solving the problem p.

$$\Phi(a,p,H) = \begin{cases} \dfrac{|H|}{\sum_{i \in H} \frac{1}{e_i(a,p)}} & \text{, if i is supported } \forall i \in H, \\ 0 & \text{, otherwise.} \end{cases} \tag{3.1}$$

To our knowledge, there is no other performance portability metric currently proposed. Furthermore, many different performance portability studies [23, 33, 34, 35] reference or use this proposed metric. As previously highlighted in Section 3.1, the choice of the metric $e_i(a,p)$ used to evaluate the performance portability of an application depends on the context, the specific goals of the evaluation, and the nature of the benchmark or workload under consideration. For instance, for memory-bound benchmarks, one may opt to prioritize peak memory bandwidth over runtime.

CHAPTER 4

High Level Overview of C++ Parallel STL and Competitors

As seen in Section 3.1, in the literature and the applied world, a wide range of frameworks are used to achieve performance portability. In this chapter, we provide a high level overview of SYCL, followed by RAJA and Kokkos. Finally, we present C++ parallel STL and highlight how it compares to the likes of SYCL and competitors. It should be noted that the list of performance portability frameworks provided is not intended to be exhaustive. The primary motivation for selecting the frameworks outlined below is based on their widespread adoption and popularity.

4.1 SYCL

SYCL [7] is an abstraction layer that enables modern ISO C++ code to be run on a wide range of devices without considering lower-level design principles. The consistent APIs for different accelerators allows developers to target CPUs, GPUs, FPGAs and other devices in a single-source codebase [36]. To get the highest possible performance out of a SYCL program, tuning for the best performance considering the platform is still required [37]. SYCL relies on lower-level APIs, such as OpenCL or CUDA, to provide the abstraction layer to its users [38]. As seen in Figure 4.1, there exists a wide range of SYCL compilers and architectures each compiler supports.

The typical workflow when writing a SYCL program looks the following. First one has to decide what kind of SYCL compiler to use. In recent years, the SYCL community got a lot of traction, resulting in major hardware vendors, such as Intel, releasing their own implementations [36]. After writing the application using the consistent SYCL API, one uses the SYCL compiler to compile the program. At compile time it is required to specify the compilation target (e.g., CUDA or CPU).

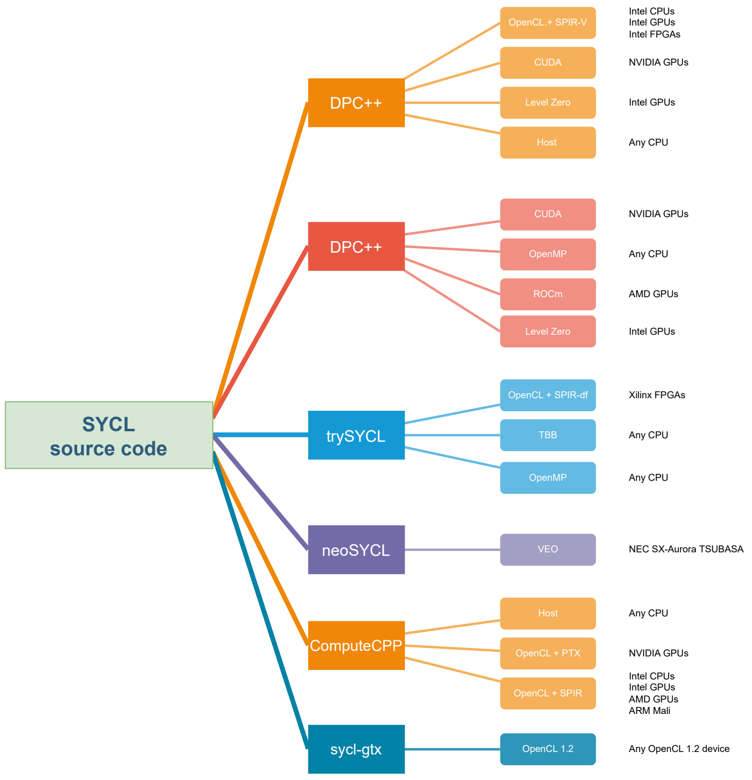


Figure 4.1: SYCL Compilers and their targets adapted from [39]

For SYCL developers, the main task is to create what are called SYCL kernels. These kernels are responsible for executing the desired logic on a device by being enqueued in a SYCL device queue. SYCL allows developers to offload computation to a wide range of devices during runtime. In other words, it is possible to write a device selector that selects a specific device depending on various criteria (e.g., a self-defined cost function). For instance, if a system has two NVIDIA GPUs, it is possible to use the first GPU for a specific task and the other GPU for a different task. Once kernels are enqueued, they are formed into a task graph. The task graph is built based on the data dependencies between kernels. After the task graph has been constructed, the kernels can be executed without any additional code required by the developer, ensuring data dependencies are satisfied.

Kernels represent the raw logic that is desired to be executed on the device. They do not typically contain any memory management logic. As Jin [40] documented, kernels are written as either lambda or named function objects (struct). Kernels are the smallest unit of a SYCL application's three-level scope. The next level up from the kernel scope is the command-group scope. This scope is responsible for specifying a unit of work and managing memory. Finally, the highest level is the application scope, which covers everything else in the SYCL application.

SYCL supports two different memory management models between devices and the host, which are Buffered and Unified Shared Memory (USM) [41]. When using Buffers, memory access is done through the SYCL API, which means that the SYCL runtime is responsible for managing, moving, and synchronizing data between the host and the device. In contrast, USM uses raw pointers. Furthermore, USM distinguishes between device-only memory and shared allocations. With device-only memory, the data is only accessible on the device where it is allocated. In contrast, shared allocations enable access from both the host and the device.

SYCL provides fine-grained control over how work is executed on a device. For example, developers can use barriers to synchronize work items within kernels. Work items are grouped into work groups, and the size of these work groups can significantly impact performance, as demonstrated by Jin [40]. SYCL has a single parallel primitive, namely `parallel_for`. However, depending on how a kernel is set up, this primitive can be used to implement a wide variety of algorithms, such as sorting, reduction, and more. When writing SYCL kernels, developers typically work with indices.

4.2 Kokkos

Kokkos [6] is a performance portability library written in modern C++, compared to directive-based approaches like OpenACC. Applications written with Kokkos can be executed on a wide variety of devices, such as CPUs and GPUs. Kokkos provides a simple API for both high performance parallel computing and data management. Kokkos works with various backends such as CUDA, HIP, SYCL, OpenMP and many more.

The main abstractions within Kokkos are Parallel Patterns, Execution Space, Execution Policy, Views, Memory Space, Memory Layout and Memory Traits [42, 43]. Only a subset of these abstractions has to be considered when writing Kokkos kernels, since the remaining ones will default to the most optimal values for the selected device [42]. Similar to SYCL, Kokkos kernels represent the raw logic that is desired to be executed on the device. They do not typically contain any memory management logic. Kernels are written as either lambda or named function objects (struct).

One key feature of Kokkos is the way how memory is managed. Kokkos provides a construct called *Views* which works similarly to a reference counter array data structure. With the help of Views, developers can define in what memory space the data is placed. The rest is dealt by Kokkos. Because views provide a fundamental building block, developers started to build fully-fledged data structures around views. Since Kokkos 3, the API provide a set of common data structures used by developers therefore it is not required by developers to build their own data structure [43].

In contrast to SYCL, where only a single parallel primitive is available, Kokkos offers a total of three distinct parallel primitives [44]:

- `parallel_for`: Can be used to implement a for loop with independent iterations;

- `parallel_reduce`: Implements a reduction operator;
- `parallel_scan`: Implements a prefix scan operator.

These primitives can be tailored to specific needs by using a custom kernel provided as lambda or named function object.

In Kokkos, the basic unit of work is referred to as a thread. Threads that are grouped share a common cache, which can be particularly advantageous for certain types of nested loops [43]. Similar to SYCL, Kokkos employs an index-centric approach to execute its parallel operations.

4.3 RAJA

RAJA [5] is a "C++ library that allows developers to write single-source applications that can target multiple hardware and programming model back-ends" [5]. RAJA has backends for a wide variety of paradigms, such as CUDA, HIP, or OpenMP.

As with SYCL and Kokkos, the smallest building blocks in RAJA are the raw kernels which represent the logic of the work to be done and can be provided as lambda or named function objects (struct) [45]. To execute kernels, RAJA provides in total three different kinds of primitives:

- `RAJA::for_all`: Simple, non-nested loops;
- `RAJA::kernel`: For nested loops and complex kernels;
- `RAJA::launch`: Creates a more sophisticated environment, allowing the execution of nested loops with `RAJA::for_each`.

`RAJA::kernel` plays a critical role as certain backends encounter performance issues or do not support nested `RAJA::for_all`. The utilization of `RAJA::kernel` offers the potential for optimization while maintaining RAJA encapsulation. Moreover, `RAJA::launch` provides the flexibility to select the device at runtime. Although there are minor differences, the remaining features of `RAJA::launch` are similar to those of `RAJA::kernel`.

RAJA does not provide a custom reduction primitive, instead, it provides a custom execution policy and access objects. Developers can write their reduction code using the three types of loops presented before. Additionally, RAJA has a custom primitive for exclusive, inclusive scan and sort. For every parallel primitive developers can specify an execution policy (CUDA, TBB, OpenMP, and many more). Changing the execution policy does not require changing the kernel logic (e.g., the lambda body).

RAJA offers an extensive selection of execution policies that enable developers to have fine-grained control over the execution of the code using the corresponding backend [46]. For instance, when using OpenMP backends, developers can enforce the use of specific directives [47], such as `schedule(static, ChunkSize) nowait` or `schedule(guided, ChunkSize)`. The extent and types of execution policies vary from one backend to another, with some providing extensive configurations and others offering only a handful. Moreover, RAJA even provides execution policies that dictate where the memory should reside, such as on the CPU stack or CUDA shared memory. Although memory management is done using execution policies, it is possible to define inside kernels "Team shared memory". Team shared memory is data that is shared across grouped threads.

RAJA uses an index-centric approach for its kernels, meaning all kernels have as a parameter only the index. What makes RAJA interesting is, it supports not only a continuous index traversal, but it is also possible to traverse indices in stride or only a subset of selected indices.

4.4 C++ Parallel STL

For a long time, C++ provided a set of algorithms that perform commonly used functionality on containers. These algorithms include sorting or counting elements matching a predicate. In C++17, the community added a new functionality to the algorithms, allowing them to include execution policies. Execution policies allow developers to hint to the compiler how the following algorithms should be executed. It is important to highlight the compiler and backends implementing those algorithms are not forced to use execution policies and always can fallback to the serial implementation. In C++17, three different execution policies were introduced:

- `std::execution::seq`: Execute the algorithm sequentially. Same as not proving an execution policy.
- `std::execution::par`: Parallel execution of the algorithm.
- `std::execution::par_unseq`: Parallel and vectorized execution.

This new feature introduced in C++17 is also known as parallel algorithms or parallel STL. Although since the release of C++17 some time has already passed, only a handful of implementations (parallel backends) exist. At the time of writing this thesis, the parallel STL is only supported by the compilers NVHPC, GCC, and MSVC. Depending on the compilers, different paradigms are used. For example, NVHPC[1] uses OpenMP for CPU parallelism and CUDA for GPU offloading, whereas GCC uses oneTBB for CPUs and provides no capability of GPU offloading.

[1]https://developer.nvidia.com/hpc-sdk

The idea of parallel STL is that for the C++ developer the interface interacting with the algorithms stays the same, no matter what kind of accelerator is used. The developers of the parallel STL backends are responsible for writing the accelerator and paradigm-specific code. This transparent interface is a key characteristic of parallel STL.

Since parallel STL is simply an extension to the already existing algorithm library [24], there exists a wide range of parallel primitives [48]. A selection of those available are:

- `std::for_each`: allows developers to run for every element of a container a specific operation defined in a kernel;
- `std::transform`: this primitive transforms each of the elements using a provided kernel and stores the result in a different container. It is important to note that due to the nature of the parallel STL the elements will be not inserted in the same order as they are present in the input container;
- `std::reduce`: by providing a reduction operation as kernel, this primitive will reduce all elements of a container to a single value;
- `std::tranform_reduce`: this primitive is a combination of `std::reduce` and `std::transform`. First, the values are transformed in place and then reduce to a single value.
- `std::merge`: this parallel primitive allows merging two containers into a result container. Developers have to ensure that the result container has at least $n + m$ space, where n and m are the numbers of elements of each input container.

Some parallel STL primitives, such as `std::for_each` allow developers to define their own kernels as lambda or named function objects (struct). For every parallel primitive, a pre-condition and post-condition is defined. As long as the pre-condition is satisfied, the post-condition is guaranteed, otherwise, undefined behavior will occur. The pre-condition and post-condition for `std::merge` can be derived from the description text[2] of the parallel primitive `std::merge` seen below:

Merges two sorted ranges [first1, last1) and [first2, last2) into one sorted range beginning at d_first.

Parallel STL is totally different to the likes of SYCL, RAJA, and Kokkos. First parallel STL does not support selecting a specific target device during run time. Depending on what backend is used, some non-standard compliant configuration options allow the selection of the offloading target at compile time. Secondly, the configuration of the C++ backends is not as extensive as for SYCL, RAJA, and Kokkos. The only configuration option provided by the ISO C++ standard during runtime is to hint how the algorithm

[2] https://en.cppreference.com/w/cpp/algorithm/merge

should be executed (i.e., execution policies). There is no option to define where and how to allocate memory. Parallel STL follows a black box approach, meaning that a developer has no control over execution, but a contract of expected results, when certain preconditions are met. Where parallel STL shines compared to the likes of SYCL, RAJA and Kokkos, is that it has an extensive selection of predefined algorithms, ranging from simple ones such as loops and sorting, to rather complex ones such as exclusive/inclusive scan or partitioning. Although, there is an extensive selection of algorithms in parallel STL, combining them is not as easy as for SYCL or Kokkos. C++17 utilizes a fork-join model in which each parallel primitive triggers parallel execution and upon completion, the next parallel primitive is executed. RAJA, Kokkos and SYCL all use an index-centric approach for their kernels, meaning the kernel parameter is an index, whereas C++17 kernels follow a data-centric approach, meaning the kernels retrieve the actual value stored in the container rather than an index.

CHAPTER 5

Formulation and Categorization of Performance Expectations for the Parallel STL

This chapter outlines our approach to addressing the research questions introduced in Section 1.1. To do so, we present a series of expectations. We aim to test these expectations and use the insights gathered, to effectively answer our research questions. It is important to highlight, that the expectations described in this chapter are implementation agnostic. This allows other researchers and developers to use these expectations as a base for implementing their custom benchmark suite as we did in Chapter 6.

In this chapter, we go through nine expectations that we test to answer our research questions. For every expectation we highlight why it is important to include it and what insights we expect from it, followed by an outline of what kind of tests have to be performed. As a side product of testing these expectations, we receive a lot of data that can be then used to evaluate and quantify performance portability between the frameworks. It is important to highlight, that performance portability is discussed in its own Chapter 8, as it should be analyzed separately.

5.1 Homogeneous Nested Parallelism

> Some parallel backends exhibit better performance and scalability when handling nested parallelism for homogeneous workloads

5.1.1 Importance

As seen in Section 4.3, the performance of a performance portability framework can vary a lot depending on how nested parallel executions are implemented. Since the C++ interface of parallel STL does not provide any configuration options on how nested parallelism should be executed at all, there might be a significant difference between parallel backends in terms of their performance and how they behave. Those differences do not only impact the performance portability between parallel STL backends but also are of interest to developers, who want to achieve the maximum possible performance. This expectation will cover homogeneous workloads (i.e., every parallel execution unit has the same amount of work).

5.1.2 Tests

For every parallel execution within the benchmarks that are associated with this expectation, the same execution policy is used. To test this expectation we first perform a strong scaling analysis and then also compare the raw runtimes using all possible threads. The expectation will be true when there is a significant difference in runtime between parallel backends and there is a visible difference in how the parallel backends scale.

5.2 Order of Parallelism

> The performance is significantly impacted by the order in which parallelism is applied, whether the outer loop is sequential and the inner loop is parallel, or the outer loop is parallel and the inner loop is sequential.

5.2.1 Importance

Section 5.1 presents the first expectation, which focuses on homogeneous nested parallelism. However, this expectation is limited in that it only employs a single execution policy for all parallel executions. It is well-known that the order of parallelism can have a significant impact on runtime. For instance, parallelizing the outer loop and keeping the inner loop sequential is likely to be the faster choice. Nevertheless, the question remains: how much faster is it? Also, it is worth investigating whether changing the order of parallelism affects the behavior of parallel backends and how it compares to other backends.

5.2.2 Tests

The benchmarks associated with this expectation are executed with two execution policies, first with the outer loop parallel and the inner loop sequential and then the outer loop sequential and the inner loop parallel. Both combinations are tested using a strong scaling analysis followed by a comparison using the raw runtimes using all available threads. The expectation will be true when there is a significant difference in runtime between the (par, seq) and (seq, par) approaches.

5.3 Heterogeneous Nested Parallelism

> Some parallel backends exhibit better performance and scalability when handling nested parallelism for heterogeneous workloads.

5.3.1 Importance

The first expectation presented in Section 5.1, highlights the importance of homogeneous workloads in nested parallelism. In this expectation, we cover heterogeneous workloads. The importance of heterogeneous workloads is justified by the fact that many algorithms, such as a parallel implementation of Depth-First-Search, provide early exit opportunities. Those early exist opportunities lead to the fact that the amount of work is not the same for all the threads. Developers would consider a backend's ability to handle heterogeneous workloads as a significant factor in their decision-making process. If one backend performs this task more effectively than another, it could strongly influence their choice.

5.3.2 Tests

For every parallel execution within the benchmarks that are associated with this expectation, the same execution policy is used. To test this expectation, we will first perform a strong scaling analysis and then also compare the raw runtimes using all possible threads. Secondly, we collect the Instructions per Second (IPS) for the parallel executions ranging for all the input sizes. The expectation will be true when there is a significant difference in runtime between parallel backends and there is a visible difference in how the parallel backends compare.

5.4 Sequential Fallback

> Different compilers/backends may fallback to sequential algorithms, leading to better performance.

5.4.1 Importance

Parallelism introduces a lot of overhead and for certain kinds of workloads this additional cost might not be feasible. There is a large number of algorithms inside the parallel STL that have some cases where parallelism solely from a conceptual standpoint does not make sense. Since the developer using the parallel STL algorithm only has an interface, which follows a pre and post condition, there is no way to control or configure specific behavior. Possible fallback mechanisms to sequential algorithms are solely in the control of the parallel STL backend developer. Because of this non-standardized behavior, users might need to introduce custom fallback checks before calling parallel STL algorithms, introducing potentially redundant work if the same checks are then performed again by the parallel STL primitive. This hidden duplicated work can have a significant impact on performance.

5.4.2 Tests

To test this expectation, we perform a strong scaling analysis and then also compare the raw runtimes using all possible threads. Furthermore, we collect the bandwidth (MBytes per second) for the parallel executions ranging for all the input sizes. The expectation will be true when there is a noticeable bump in runtime or a loss in bandwidth when changing input size or other parameters.

5.5 Specialized Parallelism Techniques

> Parallel STL backends use special parallelism techniques for (linear) algorithms, which have no clear reference implementation, leading to significant differences in terms of performance and their strong scaling properties.

5.5.1 Importance

In C++, there is no definitive parallel reference implementation for several (linear) algorithms. This implies that while adhering to the interface and producing accurate results, these algorithms can utilize various optimization techniques to enhance speed and efficiency. This allows parallel STL backend developers to optimize their code to a level such that it uses the accelerators or programming paradigms' highest possible performance. Many of the (linear) algorithms can be parallelized through chunking or other innovative approaches if the original operation permits it.

5.5.2 Tests

To test this expectation, we perform a strong scaling analysis and then also compare the raw runtimes using all possible threads. The expectation will be true when there is a significant difference in runtime between parallel backends and a visible difference in how the parallel backends behave under strong scaling.

5.6 Tailored Optimization for Parallel Scans

> Parallel STL backends leverage specialized parallelism techniques for inclusive and exclusive scans, resulting in significant variations in performance and strong scaling properties.

5.6.1 Importance

As already highlighted in Section 5.5, parallel STL backend developers can use every possible trick and performance optimization as long as they adhere to the interface description. Because parallel scans play a major role in parallel programming and many algorithms, such as sequence compaction or spare-matrix vector multiplication, used them [49], we dedicate a whole expectation to inclusive and exclusive scan.

5.6.2 Tests

To test this expectation, we perform a strong scaling analysis and then also compare the raw runtimes using all possible threads. The expectation will be true when there is a significant difference in runtime between parallel backends and a visible difference in how the parallel backends behave.

5.7 Specific vs. Custom Implementation

> Employing specific parallel algorithms tends to yield superior performance/strong scaling compared to utilizing custom implementations that rely on various other parallel algorithm functions.

5.7.1 Importance

The semantics of certain parallel STL primitives can be achieved through the use of alternative primitives, suggesting that one of the semantically equivalent implementations may outperform or scale better than the others. This understanding can be highly valuable when optimizing program performance. Additionally, it enables the development of best practices for specific parallel primitives.

5.7.2 Tests

To test this expectation we, perform a strong scaling analysis and then also compare the raw runtimes using all possible threads. Furthermore, we collect the bandwidth (MBytes per second) for the parallel executions ranging for all the input sizes. The expectation will be true when there is a significant difference in runtime between parallel primitives that have the same semantics.

5.8 Custom Index-based Iterations

Parallel STL does not inherently provide support for index-based iterations, thereby requiring the developer to devise a custom sequence for indices. The manner in which the index values are generated by the developer can significantly impact the application's performance and scalability.

5.8.1 Importance

The Parallel STL approach is data-centric, which means that the kernel parameter only receives the actual data within the container used in the parallel primitive. However, this approach can be disadvantageous when the desired algorithm requires indices. While there are various techniques to shift to an index-centric approach, there is no universally accepted "gold" standard. This means an approach that seems promising can have a drastic impact on performance.

5.8.2 Tests

To test this expectation, we perform a strong scaling analysis and then also compare the raw runtimes using all possible threads. The expectation will be true when there is a significant different in runtime between the index strategies.

5.9 Scatter Mapping vs. Compact Mapping

The performance implications of utilizing scatter mapping of threads as opposed to compact mapping can vary depending on the compiler and backend being used.

5.9.1 Importance

As highlighted in Section 2.1.1, shared memory system can be either UMA or NUMA. NUMA tend to be favored in high performance systems due to the fact of faster memory access. Since parallel STL does not provide any interface to control NUMA awareness, it is in the hand of the parallel STL backend developers.

5.9.2 Tests

To test the expectation, we conduct a strong scaling analysis, followed by comparing the raw runtimes utilizing all available threads. Additionally, we will implement thread pinning using the configuration sparse and scattered, via tools such as numactl. During runtime, we will also collect the bandwidth (MBytes per second). The expectation holds when there is a significant difference in runtime between the index strategies.

CHAPTER 6

Benchmark Suite: pSTL-Bench

In this chapter, we present the novel benchmark suite *pSTL-Bench* used in this thesis. In Section 6.1, we present the concrete benchmarks we implemented in order to test the expectations presented in Chapter 5. This is followed by an overview of the components the benchmark suite *pSTL-Bench* consists of in Section 6.2.

Due to the large number of benchmarks, we present only code snippets for a small subset. Furthermore, the code snippets presented will only include the code that will be benchmarked and not any external needed setup. A full implementation example can be seen in Listing 6.8.

6.1 Benchmarks

6.1.1 Homogeneous Nested Parallelism

For this expectation, we have created in total three different benchmarks. All of those benchmarks rely on the `std::for_each` parallel primitive. In the following sections, we will shortly explain what they do.

Homogeneous Linear

This benchmark has a time complexity of $\mathcal{O}(n)$. For every input element, the benchmark has the homogeneous workload of first calculating the sinus and cosine and then taking the minimum of both. In Listing 6.1, one can see the shortened code snippet for this benchmark.

The function seen in the listing aims to wrap abstract the code away that will be benchmarked. In order to be a flexible as possible the execution policy is not hard coded, instead it is provided as a template as one can see on Lines 1 and 3. On Line 4, the input data to the primitive is provided. Since the function call will be timed, we do not want

Listing 6.1: Implementation of the homogeneous linear benchmark (shortened).

```
 1  template<class ExecutionPolicy>
 2  void homogeneous_linear(
 3    ExecutionPolicy &policy,
 4    const std::vector<int> &in) {
 5
 6    std::for_each(policy, in.begin(), in.end(), [](const auto &entry) {
 7      std::min(std::sin(entry), std::tan(entry));
 8    });
 9
10  }
```

to create the input data inside the method. Hence, it has to be provided as reference. Finally, the most important part of the method can be found in Line 6. There the parallel primitive `std::for_each` is called. As kernel (i.e., lambda function), a rather simple code block can be found. For every element minimum of the calculated sinus and cosine is selected. The kernel has as parameter a single reference to an entry inside the input container. Using a reference is advised such that no undesired copy occurs.

Homogeneous Quadratic

This is the first benchmark where nested parallelism is in use. As one can see in Listing 6.2, the benchmark uses two parallel `std::for_each` to generate the cross join using the elements of the input vector. For example, for a vector containing two elements $[1, 2]$, we will have the following (i, j)-pairings: $[(1, 1), (1, 2), (2, 1), (2, 2)]$. The program does most of its work in the innermost part of the loops, where it calculates $tan(i) + cos(j)$ for every (i, j)-pairing. The final complexity is $\mathcal{O}(n^2)$.

Listing 6.2: Implementation of the homogeneous quadratic benchmark (shortened).

```
template<class OuterPolicy, class InnerPolicy, typename T>
void
homogeneous_quadratic(OuterPolicy outer, InnerPolicy inner, const T &in) {

  std::for_each(outer, in.begin(), in.end(), [&](const auto &e1) {
    // nested parallel loop
    std::for_each(inner, in.begin(), in.end(), [&e1](const auto &e2) {
      auto value = std::tan(e1) + std::cos(e2);
    });

  });
}
```

In contrast to the code snippet shown for the benchmark *Homogeneous Linear* (in Listing 6.1), the code shown in Listing 6.2 has three template values. The first and second template (i.e., `OuterPolicy` and `InnerPolicy`) controls the execution policy of the inner and outer parallel primitive `std::for_each`. The last template value allows users to provide as input value any type that can be traversed using the parallel STL primitive.

Homogeneous Exponential

As nested parallelism can be applied to achieve different levels of complexity, we have included a benchmark with a time complexity of $\mathcal{O}(2^n)$. This benchmark takes a list of indices starting from zero and going up to n as input. For each index, it generates a Fibonacci tree using parallel execution. For instance, if the input vector has three elements, the number of parallel calls for each of them can be seen in Figure 6.1. To add computational complexity to the benchmark, every internal node computes the sum of the tangents and cosines using its current index. This is followed by initiating a parallel execution that spans its child nodes. For instance when the computation reaches the red marked node 2 shown in Figure 6.1, it calculates the sum $tan(2) + cos(2)$ and then spawn the children 0 and 1 marked as green nodes.

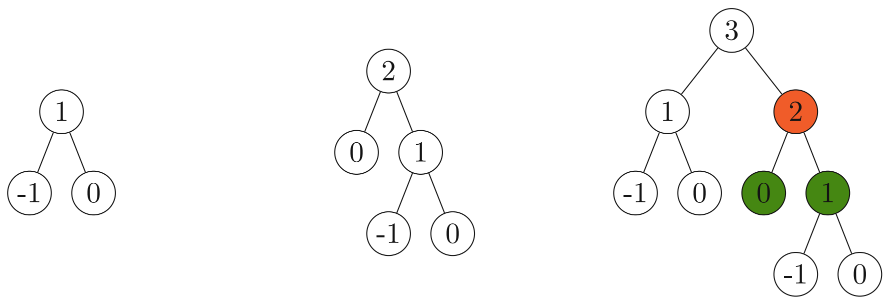


Figure 6.1: Call Tree for exponential benchmark using vector of size three.

6.1.2 Order of Parallelism

For this expectation, we have a single benchmark. The benchmark is identical to the one presented in Section 6.1.1. The only difference between those two benchmarks is in the way how we call them. For this specific expectation, the inner and outer execution policy used for the parallel primitive will not be the same. The benchmark code in Listing 6.2 is intentionally designed for easy reuse.

6.1.3 Heterogeneous Nested Parallelism

To test this expectation we will have in total two benchmarks. Both exhibit heterogeneous workloads and use the parallel primitive `std::for_each`.

Mandelbrot Linear

To simulate heterogeneous workload this benchmark calculates for a one dimensional vector the Mandelbrot set. The value inside the vector represents the X position of a pixel which Mandelbrot value should be calculated. Since we are only working with a one-dimensional vector, the Y location of the pixel calculated is fixed to a constant for all the pixels. It is important to highlight that only the loop over the one-dimensional vector is parallelized, the maximum k iterations to find the Mandelbrot value is sequential. The simplified Mandelbrot algorithm used for the benchmark has a time complexity of $\mathcal{O}(kn)$, where n is the size of the array and k the maximum number of iterations to calculate the Mandelbrot value. The code snippet for this benchmark can be seen in Listing 6.8.

Mandelbrot Quadratic

To simulate quadratic heterogeneous workload this benchmark calculates the Mandelbrot set for a picture of size n. To get the position of the pixel, two nested parallel loops are required. The loop of maximum k iterations to calculate the Mandelbrot value of a given pixel is not parallelized. The simplified Mandelbrot algorithm used for the benchmark has a time complexity of $\mathcal{O}(kn^2)$ where n is the size of the array and k is the maximum number of iterations to calculate the Mandelbrot value.

6.1.4 Sequential Fallback

For this expectation, we use in total four benchmarks, where each one uses a different parallel primitive.

Merge Logic

This benchmark is rather straightforward as one can see in Listing 6.3. We simply call the parallel primitive `std::merge`, with two, as required by the pre-condition, already sorted vectors and no additional compare lambda. Providing an additional compare lambda is not of interest since it will be only used to decide which of two elements is the smaller. The implementation of this benchmark demonstrates the ease of use and simplicity when working with the parallel STL well.

Listing 6.3: Implementation of the merge logic benchmark (shortened).

```cpp
template<class ExecutionPolicy, typename T>
void b4_1_merge(ExecutionPolicy &&policy, const T &v1, const T &v2, T &res) {
  std::merge(policy,
  v1.begin(), v1.end(), // first input container
  v2.begin(), v2.end(), // second input container
  res.begin() // where to store the results
  );
}
```

Stable Sorting

For this benchmark, we take a closer look at the parallel primitive `std::stable_sort`. This primitive as the name suggests will stable sort the input vector. The parallel primitive will be called with three different input types, which are Ascending Sorted, Descending Sorted, and Not Sorted. This wide variety of input data allows us to check if different input has an effect or always the same amount of work is used.

Set Union Logic

This benchmark focuses on the parallel primitive `std::set_union`. The algorithm will produce the union of two input vectors not including duplicates in the overlap. The parallel primitive will be called with three different input types:

- *One empty*: One of both input vectors is empty. Leading to the fact that the resulting union is a copy of the other non-empty input vector.
- *One wholly greater*: Every element of one input vector is greater than every element of the other vector. This will lead to the problem of copying both inputs into the result in a sorted manner.
- *Front overhang*: Both input vectors overlap in their values.

This wide variety of input data allows us to check if different input has an effect or always the same amount of work is used.

Set Difference Logic

For this benchmark, we explore in detail the parallel primitive `std::set_difference`. Given two sets R and S, the algorithms will produce $R \setminus S$. (In the following R will be called "Left" and S will be called "Right"). The parallel primitive will be called with four different input types:

- *Left empty*: This will allow the parallel backend to use an early exit, since the result will be always empty.
- *Right emtpy*: This will lead to a copy of the left input vector.
- *One wholly greater*: Every element of one input vector is greater than every element of the other vector. This will result in a copy of the left input vector.
- *Intersected*: Left and right intersect resulting in calculating the actual $R \setminus S$.

6.1.5 Specialized Parallelism Techniques

For this expectation, we use in total three benchmarks, where each one uses a different parallel primitive.

Find Elements

For this benchmark, we take a closer look at the parallel primitive `std::find`. The algorithm tries to find the first element in a container that matches a given predicate. We will call this parallel primitive with three different predicates to exhibit different behavior:

- *Find first entry*: the predicate provided will match the first element of the container. This provides the parallel STL backends with an early exit opportunity;
- *Find last entry*: the predicate provided will match only the last element of the container. Leading to the fact that every element might have to be touched;
- *Find non existing entry*: the predicate provided will match no element of the container. This forces the algorithm to touch every element leading to $\mathcal{O}(n)$.

Vector Partition

This benchmark focuses on the parallel primitive `std::partition`. The algorithm reorders the elements of the vector in such a way that there is a clear border between elements that match the given predicate and those that do not. Since the goal of this benchmark is to analyze the behavior of the parallel primitive, the predicate can be arbitrary.

Unique Copy

For this benchmark, we take a closer look at the parallel primitive `std::unique_copy`. As the name suggests the parallel primitive copies only the unique elements of a container into the result container. The input data used for this benchmark is a vector containing only the same element.

Min Max Elements

This benchmark focuses on the parallel primitive `std::minmax_element`. The algorithm tries to find both the minimum and maximum elements inside a container. The results are not a pair of values but rather a pair of pointers to the memory where the minimum and maximum are located. The benchmark will be called once with a vector only containing the same value (e.g., all the values in the container are 230) and once with a vector with increasing values. This allows us to check whether the implementations use different strategies for different kinds of data.

6.1.6 Tailored Optimization for Parallel Scans

Inclusive Scan

The first benchmark focuses on `std::inclusive_scan`. As input, there will be a randomly generated vector. As we can see in Listing 6.4, the benchmark code is rather simple. This is due to the ease of use and simplicity of the parallel STL interface.

Listing 6.4: Implementation of the inclusive scan benchmark (shortened).

```
template<class Policy, typename T>
void b6_1_inclusive_scan(Policy &&policy, const T &in, T &res) {
  std::inclusive_scan(policy, in.begin(), in.end(), res.begin());
}
```

Exclusive Scan

The first benchmark focuses on `std::exclusive_scan`. As input, there will be a randomly generated vector. The code for this benchmark is similar to the one presented in Listing 6.4. The only difference is that the parallel primitive is `std::exclusive_scan` instead of `std::inclusive_scan`.

6.1.7 Specific vs. Custom Implementation

The benchmarks used for this expectation are split into six groups. Each group tackles a different operation that can be replicated by alternative parallel primitives.

Group 1 - Copy logic

The first group covers how to copy the values from one container to another. There is a vast variety of possible parallel STL primitives that can be used to achieve these semantics, but we limited ourselves to two primitives a developer will most probably think of. As input, we will use a simple vector with random values.

Specific Copy Primitive This benchmark uses the parallel primitive `std::copy` to copy all the elements from one vector to another. In Listing 6.5, it becomes evident that the benchmarks' simplicity shines through as it requires only a single method call to the parallel STL to implement the copy logic.

Listing 6.5: Implementation of the Specific copy with for_each benchmark (shortened).

```
template<class Policy, typename T>
void specific_copy(Policy &&policy, const T &input, T &res) {
  std::copy(policy, input.begin(), input.end(), res.begin());
}
```

Custom copy with for_each To replicate the semantics of the parallel primitive `std::copy`, the primitive `std::for_each` with an index-centric solution is used. Meaning instead of using the contents of the input vector as a kernel parameter, an index will be provided. Hence, every execution of the kernel will copy exactly one element. As one can see in Listing 6.6, in order to copy the elements from the input container `input` to the result container `res`, a lambda reference caption is required (marked as `[&]` in Line 2). As one can see the complexity of this implementation is already rather high in contrast to the one using the specific primitive.

Listing 6.6: Implementation of the Custom copy with for_each benchmark (shortened).

```
1 template<class Policy, typename T, typename Indices>
2 void custom_copy(Policy policy, const T &input, const Indices &ind, T &res) {
3 
4   std::for_each(policy, ind.begin(), ind.end(), [&](const auto &index) {
5     res[index] = input[index];
6   });
7 
8 }
```

Group 2 - All of logic

This group covers the semantics of checking whether all the elements of a container satisfy a given predicate. For this group, we use three different predicates. The first predicate is valid for every element, the second predicate is valid for every element except the first one and the last predicate is not valid for any element at all.

Specific All_of Primitive This benchmark uses the parallel primitive `std::all_of`, to check whether a given predicate is true for every element of the container or not.

Custom All_of with transform_reduce To replicate the semantics of the specific parallel primitive `std::all_of`, the primitive `std::transform_reduce` is used. In order to replicate the semantics, the same predicate provided to `std::all_of` can be used. The challenging part of this implementation is to select the correct reduction operator and initial value. As reduction method the logical and-operator and as initial value `true` is used. Since the predicate provided to the parallel primitive will return a boolean value the logical conjunction of all those boolean values will indicate whether all the elements in the container satisfy the predicate or not. This works because the logical and-operator is commutative and associative.

Group 3 - Count if logic

This group covers the semantics of counting the number of elements in a container, that satisfy a given predicate. There are many ways to achieve this semantic using parallel STL, but we limit ourselves to two approaches which are most probably the easiest to think of. Both strategies will be called with three different predicates. The first predicate satisfies all the elements, the second satisfies only half of the elements and the number of elements that satisfy the third predicate is defined by the function $max(size - \frac{cutoff}{quantity}, 0)$.

Specific Count_if Primitive This benchmark uses the logic-specific parallel primitive `std::count_if`, to count all the elements that satisfy the given predicate.

Custom Count_if with transform_reduce To replicate the semantics of the parallel primitive `std::count_if`, the parallel primitive `std::transform_reduce` can be used. It is pretty much similar to the approach shown in Section 6.1.7, but instead of using the logic and-operator as reduction, this time we use the plus operator. The reason why the plus operator works is that the result type of the predicate is a Boolean. Booleans are represented in C++ as 0 for false and 1 for true. Meaning for every element, we want to count the predicate will return 1, and for every element we do not want to count (since the predicate evaluates to false), the predicate will return 0. The sum of those 1's and 0's is the number of elements satisfying the predicate.

Listing 6.7: Implementation of the custom count_if with transform_reduce benchmark (shortened).

```cpp
template<class Policy, typename Container,
typename Diff_Type = typename Container::difference_type,
typename Reference_Type = typename Container::reference,
typename Pred = typename std::function<bool>(Reference_Type)
>
Diff_Type
custom_count_if(Policy &&policy, const Container &in, Pred &&pred) {
  // the trick is that false = 0 and true = 1
  return std::transform_reduce(policy,
  in.begin(), in.end(),    // input values
  0, std::plus(),          // initial value and reduce operator
  pred                     // predicate provided by the user
  );
}
```

Although the idea behind the replication logic is rather complex, due to the simple interface of the parallel STL the implementation is rather simple as one can see in Listing 6.7.

Group 4 - Stencil transform logic

Many parallel algorithms require the use of stencil operations, such as the one developed by Brandvik and Pullan [50]. In their work, they designed a partial differential equation solver that heavily relied on these computations. Since in parallel STL stencil transform operations can be done with various primitives, we will compare two of them in this group. The stencil operation used for this group is not as complex as the one used by Brandvik and Pullan. Instead, we will calculate the standard deviation of three neighboring elements in a container. For both implementations, we use an index-centric approach.

Stencil with transform This benchmark uses the parallel primitive `std::transform`, to calculate the standard deviation of 3 neighboring elements.

Stencil with for_each This benchmark uses the parallel primitive `std::for_each`, to calculate the standard deviation of three neighboring elements. In contrast to the implementation using the parallel primitive `std::transform`, where the results are written into the result container implicitly, here writing the results has to be done explicitly.

Group 5 - Scalar transform logic

Scalar transformation can be achieved with a vast majority of different parallel primitives. In Group 4 we saw a transformation using indices. In this group, we perform a scalar transformation but this time not using indices.

Scalar with transform This benchmark uses the parallel primitive `std::transform`, to multiply the container values by a scalar.

Scalar with for_each This benchmark will use the parallel primitive `std::for_each`, to multiply the container values by a scalar. In contrast to `std::transform`, where the results are written into the result container implicitly, here writing the results has to be done explicitly.

Group 6 - Serial vs direct call

Parallel STL offers two essential functions, namely `std::transform` and `std::reduce`. Although, for several use cases, it is common to first transform the data and then reduce it to a single value, the parallel STL also offers a combined function called `std::transform_reduce`. However, it is worth noting that even though the parallel primitive `std::transform_reduce` exists, one can still implement the same logic by utilizing the other two functions sequentially in a serial manner.

Serial This benchmark uses the specific parallel primitive `std::transform_reduce`, to sum up the RGB-Values of all the pixels inside a container.

Direct To copy the semantics, we call the parallel primitive `std::transform`, to sum up the RGB-Values of a single pixel. Then we sum up all the single values produced by the previous step to a single value using `std::reduce`.

6.1.8 Custom Index-based Iterations

In this benchmark, we cover in total five different ways to perform a simple transform operation. One of those five uses the data-centric approach and the other four will use an index-centric approach. All the benchmarks use the same parallel primitive `std::transform`.

Data-Centric

This benchmark follows a data-centric approach. This means that the elements of the container are directly provided as a parameter to the kernel, allowing developers to directly consume their values without the need to use an index access operation on a container.

Index-Centric with iota

For this benchmark, we utilize `std::iota` to generate a vector of indices. These indices will then be used in the parallel primitive to simulate an index-centric approach, similar to the strategy used by Mietzsch and Fuerlinger [1].

Index-Centric with views::iota

In C++20, a new concept of range views was introduced. This new concept allows creating a range of numbers without actually creating the numbers and storing them in memory. This benchmark will use this new concept i.e., `std::views::iota`.

Index-Centric with Custom Iterator

In various research papers [24, 23], the authors implement their own range iterator to perform index-centric traversal. All of those implementations seem to follow the same strategy. Therefore, we selected the implementation from Lin et al. [24] in this benchmark.

Index-Centric with Boost

The boost library is widely sought-after in the C++ ecosystems. Many of the new C++ standard features were influenced by back then boost exclusive features. Boost plays still a significant role in C++ and also provides a counting iterator that can be used for an index-centric approach. Although Boost is not part of the ISO C++ standard, we still decided to include a benchmark using Boost, due to Boosts importance in C++. This benchmark will use Boosts `counting_iterator` to create the indices. In case users of the *pSTL-Bench* are not interested in the results of Boost it is possible to exclude this benchmark using the compile time flag `SKIP_BOOST`.

6.1.9 Scatter Mapping vs. Compact Mapping

In total, we use three different benchmarks to test this expectation.

Specific Copy Primitive

This is exactly the same benchmark as used in *Specific vs. Custom Implementation Group 1*, which can be seen in Section 6.1.7.

Inclusive Scan

This is the same benchmark as used in *Tailored Optimization for Parallel Scans*, which can be seen in Section 6.1.6.

Scalar with transform

This is the same benchmark as used in *Specific vs. Custom Implementation Group 5*, which can be seen in Section 6.1.7.

6.2 Implementation Details

The benchmark suite consists of four components, that play nicely together to achieve the common goal of providing a deep insight into the parallel STL. In the upcoming sections, we will provide an independent explanation of each component and its role in the bigger picture. Additionally, we will share insights into the design decisions we made, during the development.

6.2.1 CORE Component

The main goal of the benchmark suite was to decouple the execution and data collection from the actual primitive that one wants to benchmark. Hence, we introduced an independent component called *CORE* that contains all the code snippets that can be benchmarked. The main objectives for *CORE* is to hold all the benchmarks, register them with the execution engine that is within the component and expose them via a command line interface to potential users. For us, it was of utter importance that the benchmarks collected in *CORE* still can be executed independently without the need for other pSTL-Bench components. For instance, for testing or fast prototyping purposes. Furthermore, the code inside *CORE* should be backend and compiler agnostic.

Before we started with the development of *CORE*, we reflected on the requirements stated above and the expectations presented in Chapter 5. Reflecting is a critical part of this process, since designing a benchmark suite without considering what benchmarks one wants to run at a later point simply will not work. After reflecting we concluded that besides runtime other critical information (such as bandwidth or performance counters) has to be collected, ideally configurable and transparent such that the developer of the benchmark does not have to deal with it and users can decide at runtime what to include and what not.

With all those requirements we started by building our prototype using the industry standard build automation tool CMake [51] and Google Benchmark[1]. The main motivation behind utilizing Google Benchmark is multi-faced. Firstly, it offers a robust foundation for our benchmark suite, which eliminates the need for us to create one from scratch. Secondly, it enables us and of course, other developers in the future to obtain vital statistics such as bandwidth and performance counters, without writing complex code. This simple benchmarking code allows developers to concentrate on the benchmark code rather than on external factors, such as collecting metrics. Finally, this project has a large user base and has served as a benchmarking tool for numerous microbenchmarks in the community, including Chandler Carruth's CPPCon talk [52].

Because Google Benchmark gives us the possibility to configure the benchmarks during runtime, we had to decide what parameters should be configured at runtime and which at compile time. In the end, we decided that only a handful of properties should be provided at compile time and the rest can be configured during runtime. The reason for selecting the following configuration properties for compile time is twofold. Firstly, these properties are specific to the backend and should not change during runtime. Therefore, it is essential to establish them during compilation. Secondly, these properties need to remain consistent across all executions and should not be modified at runtime for the sake of consistency. Hence, it is necessary to fix them during compilation.

The following compile-time configuration can be applied using CMake:

- `BENCHMARK_PREFIX`: this allows adding a prefix to every benchmark name. This can be used to for example add the backend name at compile time in front of every benchmark, so one can later easily distinguish them.
- `CXX_COMPILER`: allows specifying what compiler to use.
- `CXX_FLAGS`: allows specifying further compile time flags needed by the backend or the select compiler.
- `USE_PARALLEL_ALLOCATOR`: allows specifying whether to use our custom memory allocator or not.
- `SKIP_BOOST`: include benchmarks using the third-party library BOOST.
- `MAX_INPUT_SIZE`: allows one to specify the maximum container size (i.e., number of elements) benchmarks should support.

During runtime, a wide variety of configurations can be applied. We will only present a selection of those, which seem to be the most important ones we identified while working on this thesis.

[1]https://github.com/google/benchmark

- `--benchmark_filter`: This feature allows users to choose which benchmarks to execute based on their name using a regular expression. It is particularly useful for benchmark suites with many tests, as it simplifies the selection of specific tests. For example, users can select benchmark A with an input size of 2^{20}.
- `--benchmark_repetitions`: allows specifying how often a benchmark should be executed.
- `--benchmark_format`: allows to specify the output format. This can be plain text, CSV or JSON.
- `--benchmark_perf_counters`: allows specifying what performance counters should be collected during runtime. The system supports a maximum of four different performance counters.

An important concept to highlight is how Google benchmark works. As shown above there is a configuration option called `benchmark_repetitons`. In Google benchmark, there are *iterations* and *repetitions*. Every benchmark registered by Google benchmark will be executed multiple times. This number of executions is called iterations. By default, the number of iterations per benchmark is determined dynamically. Meaning that faster benchmarks will have more iterations than slower ones. To ensure that the collected results are statistically stable, the system will first run the benchmark multiple times measures how much time each run took and calculates the number of iterations needed for this benchmark. Repetitions in contrast defines how often the benchmark with its iterations should be repeated. For example, benchmark A has 100 iterations. If we define ten repetitions, benchmark A will be executed ten times, each time with 100 iterations.

Since *CORE* uses under the hood Google benchmark, we have the option to collect hardware and software performance counters (using libpfm[2]) during runtime without requiring developers to interact with third-party libraries such as PAPI [53, 54]. Although if a benchmark requires the use of tools such as PAPI this is easily extendable through custom user counters.

Creating a new benchmark and including it into *pSTL-Bench* is rather straightforward. In Listing 6.8, we can see for example the implementation of the linear Mandelbrot benchmark for the expectation *Heterogeneous Nest Parallelism* shown in Section 6.1.3.

The inclusion of a benchmark in the *CORE* framework can be divided into three parts. The first part involves the creation of the code to be benchmarked. For instance, in the case of the linear Mandelbrot benchmark discussed in Section 6.1.3, this entails implementing a function that receives a vector of pixel indices and calling, using the parallel primitive `std:for_each`, the function responsible for calculating the Mandelbrot value. This operation occurs on Line 10. It is advisable to employ namespaces to prevent naming

[2]https://man7.org/linux/man-pages/man3/libpfm.3.html

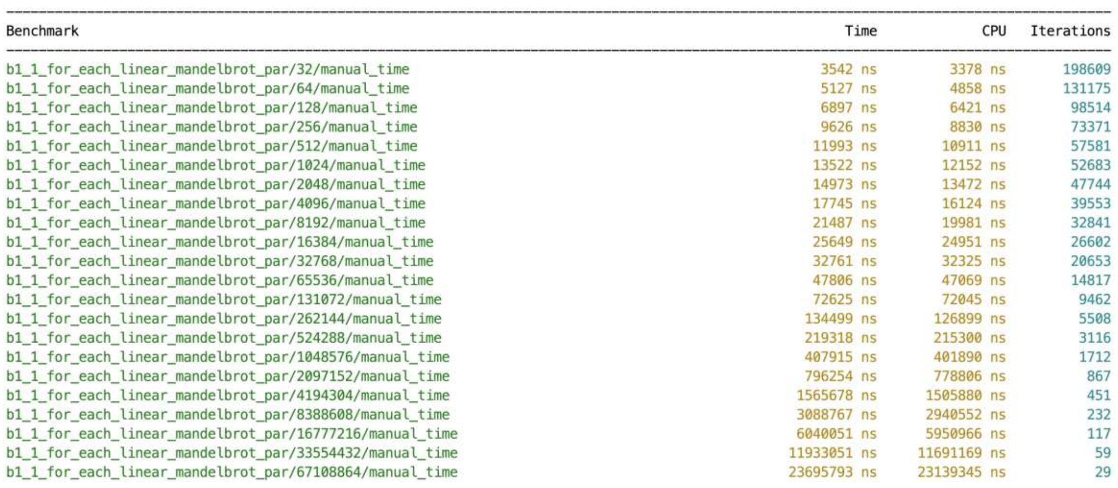
Listing 6.8: Implementation of the linear Mandelbrot benchmark (shortened).

```
 1  namespace B1 {
 2
 3    int calculate_mandelbrot(const auto &entry) {
 4      // logic to calculate mandelbrot value
 5      // ...
 6    }
 7
 8    template<class ExecutionPolicy,
 9    typename BASE_POLICY = suite::base_type<ExecutionPolicy>>
10    void b1_1_for_each_linear_mandelbrot(
11    ExecutionPolicy &policy, const suite::int_vec<BASE_POLICY> &pixel) {
12
13      std::for_each(policy, pixel.begin(), pixel.end(), [&](const auto &entry) {
14        int val = calculate_mandelbrot(entry);
15        benchmark::DoNotOptimize(val);
16      });
17    }
18
19  }
20
21  template<class Policy>
22  static void mandelbrot_wrapper(benchmark::State &state) {
23    constexpr auto execution_policy = Policy{};
24
25    const auto size = state.range(0);
26    const auto x = suite::generate_increment(execution_policy, size, 1);
27
28    for (auto _: state) {
29      WRAP_TIMING(B1::b1_1_for_each_linear_mandelbrot(execution_policy, x);)
30    }
31  }
32
33  BENCHMARK_TEMPLATE1(mandelbrot_wrapper,std::execution::parallel_policy)
34  ->Name(BENCHMARK_NAME("b1_1_for_each_linear_mandelbrot_par"))
35  ->RangeMultiplier(2)
36  ->Range(1 << 5, MAX_INPUT_SIZE);
```

conflicts. It is crucial that the benchmark code does not define the execution policy to be used. As shown in Line 8, templates are utilized to make the code as general as possible. This approach enables the code being benchmarked to remain as execution policy-agnostic as possible. The second stage involves the creation of a wrapper around the code to be benchmarked. This wrapper is responsible for setting up all the necessary inputs and utilizing the Google benchmark environment. This entire process occurs within the method shown in Line 22. For instance, in the present example, an input vector containing ascending numbers from 1 until *size* is generated, where the value of *size* is determined by the Google benchmark environment, which can be externally controlled. Lastly, in Line 29, the code to be benchmarked is invoked within the `WRAP_TIMING` macro, which measures the execution time and reports it to Google benchmark. As demonstrated in Line 21, C++ templates are once again employed to ensure the wrapper remains execution policy-agnostic.

The third and final phase of benchmark creation involves registering the benchmark with Google benchmark. This action occurs on Line 33. As our benchmark wrapper necessitates a template, we must define the execution policy to be used. In this scenario, we intend to utilize the parallel execution policy. This is followed by the name we wish to give to this specific benchmark. The last two lines of code are particularly noteworthy, as they establish the minimum and maximum number of pixels for which the benchmark may be executed. In our case, the minimum number is 2^5, and the maximum size is determined by the runtime configuration parameter `MAX_INPUT_SIZE`. It is important to note that while the minimum is 2^5 and the maximum is `MAX_INPUT_SIZE`, Line 35 restricts the selection of numbers to those that are powers of two.

After the registration is completed the benchmark will be executed with every input size by default when executing the binary, as seen in Figure 6.2. Since the default value for `MAX_INPUT_SIZE` is 2^{26} it will run until this size is reached.



```
------------------------------------------------------------------------------------------------------------------
Benchmark                                                                          Time             CPU   Iterations
------------------------------------------------------------------------------------------------------------------
b1_1_for_each_linear_mandelbrot_par/32/manual_time                              3542 ns         3378 ns       198609
b1_1_for_each_linear_mandelbrot_par/64/manual_time                              5127 ns         4858 ns       131175
b1_1_for_each_linear_mandelbrot_par/128/manual_time                             6897 ns         6421 ns        98514
b1_1_for_each_linear_mandelbrot_par/256/manual_time                             9626 ns         8830 ns        73371
b1_1_for_each_linear_mandelbrot_par/512/manual_time                            11993 ns        10911 ns        57581
b1_1_for_each_linear_mandelbrot_par/1024/manual_time                           13522 ns        12152 ns        52683
b1_1_for_each_linear_mandelbrot_par/2048/manual_time                           14973 ns        13472 ns        47744
b1_1_for_each_linear_mandelbrot_par/4096/manual_time                           17745 ns        16124 ns        39553
b1_1_for_each_linear_mandelbrot_par/8192/manual_time                           21487 ns        19981 ns        32841
b1_1_for_each_linear_mandelbrot_par/16384/manual_time                          25649 ns        24951 ns        26602
b1_1_for_each_linear_mandelbrot_par/32768/manual_time                          32761 ns        32325 ns        20653
b1_1_for_each_linear_mandelbrot_par/65536/manual_time                          47806 ns        47069 ns        14817
b1_1_for_each_linear_mandelbrot_par/131072/manual_time                         72625 ns        72045 ns         9462
b1_1_for_each_linear_mandelbrot_par/262144/manual_time                        134499 ns       126899 ns         5508
b1_1_for_each_linear_mandelbrot_par/524288/manual_time                        219318 ns       215300 ns         3116
b1_1_for_each_linear_mandelbrot_par/1048576/manual_time                       407915 ns       401890 ns         1712
b1_1_for_each_linear_mandelbrot_par/2097152/manual_time                       796254 ns       778806 ns          867
b1_1_for_each_linear_mandelbrot_par/4194304/manual_time                      1565678 ns      1505880 ns          451
b1_1_for_each_linear_mandelbrot_par/8388608/manual_time                      3088767 ns      2940552 ns          232
b1_1_for_each_linear_mandelbrot_par/16777216/manual_time                     6040051 ns      5950966 ns          117
b1_1_for_each_linear_mandelbrot_par/33554432/manual_time                    11933051 ns     11691169 ns           59
b1_1_for_each_linear_mandelbrot_par/67108864/manual_time                    23695793 ns     23139345 ns           29
```

Figure 6.2: Execution of Benchmark Mandelbrot Linear.

Due to the design decisions made at the beginning of the project, there will be for every compiler and parallel backend combination an executable of *CORE*. Each binary is by default built using the compiler flags `-O3` and `-march=native`, to ensure maximum performance.

6.2.2 Data Collector Component

All the benchmarks required to test the expectations presented in Chapter 5 are collected in *CORE*. As mentioned above the code inside *CORE* is compiler and parallel backend agnostic meaning that only at compile-time developers have to decide what backend and what compiler to use. Additionally, to ensure that our benchmarks are executed in isolation, we need to create a unique job file for each benchmark and configuration that we wish to run, and manually submit it to SLURM. For example, if we want to collect data from ten different compiler-backend-pairing, we have to manually build the ten binaries, then create all the job files calling those benchmarks in different environments (e.g., only running with 4 Threads) and then submit this manually to SLURM. This is quite tedious, hence an automation tool is required to do this. This is where *Data Collector* plays a significant part in this benchmark suite.

Based on a configuration file, the *Data Collector* will first build the correct binaries for every compiler-backend pairing. This is followed by generating the correct job files based on the benchmarks that were selected in the configuration file. This allows users to specify they want to run a specific benchmark (e.g., `b7_1_copy`) with two, eight and sixteen threads on compiler-backend-pairing A and B.

Listing B.1 provides an example configuration file. At the top of this JSON file, we specify the location to store the data collector artefacts. In Line 3, we can specify the number of repetitions to use. To conduct our experiments, we utilized a repetition count of ten. This decision was made after observing that increasing the number of repetitions did not lead to a significant change in the results. However, it substantially increased the runtime of the benchmarking tool. The most critical part of this file is the compiler and benchmark section. In the compilers section, developers can choose and name the compiler-backend pairings to use for running the benchmarks. In the benchmark section, developers can specify which benchmarks to run using a regular expression. Additionally, we can specify the environment to run these benchmarks. For example, in Line 30, we define a benchmark where only one thread is used. The data collected from this benchmark could be used, for example, to generate strong scaling plots.

Using a configuration file such as demonstrated in Listing B.1 allows the Data Collector to be a one-click solution for building the binaries and creating all the job files needed to submit to SLURM.

It is worth noting that the generation of job files based on the benchmarks defined in the configuration files is highly customizable. Therefore, if one plans to run benchmarks in a different environment, such as using numactl, there is no need to rewrite the Data Collector. Instead, one can simply extend an interface within the Data Collector, to customize the job file generation process. This feature makes it easy to modify the Data Collector to suit specific needs without the hassle of completely reworking the entire system.

6.2.3 SBatch-Trigger

Given that the Data Collector generates a broad range of SLURM job files depending on the configuration used, we developed a small and agile Python script to submit all jobs present in a provided folder and its subdirectories. For instance, on our Nebula machine, we had approximately 1030 job files distributed across three subfolders. Manually submitting that many job files using the command line would have been extremely tedious and time-consuming. Therefore, our script proved invaluable in saving us time and effort.

6.2.4 Visualization

In Chapter 5, we presented in total nine different expectations that we want to test. As one might expect to test all these expectations a large amount of data will be generated. Inside those data points, there is a lot of implicit knowledge that needs to be made explicit and easily digestible. Hence, there is a clear need for visually appealing visualizations. Although there are automation tools to visualize the results of CSV files, we realized that due to the format and distribution of the CSV files, it is still unpractical to use such tools. Hence, we had to manually analyze the data collected from the benchmarks. To do so we utilized Jupyter notebooks with industry-standard data science tools for Python such as pandas [55] and matplotlib [56].

CHAPTER 7

Experimental Evaluation

In this chapter, we present the results we got from our experiments (i.e., benchmarks) for the machines Nebula and Hydra demonstrated in Section 7.1.1. We present the results measured from our benchmarks, followed by a set of crucial details we discovered while collecting results.

7.1 Experimental Setup

As highlighted in Section 6.2.2, we used ten repetitions to make sure our results were not skewed. We utilize the minimum time measured (referred to as the fastest runtime) for visualization purposes, while for the Bytes processed, we consider the maximum value (known as the peak).

Due to the large number of graphs, we decided to only show the Nebula plots in this chapter. For completeness, one can find the Hydra plots in Appendix D. In general, both machines follow the same trends for the backends, there are only small differences that will be discussed in Section 7.11.

7.1.1 Hardware Setup

We utilize a total of three machines to execute all of our benchmarks. Two of these machines use multicore CPUs, while the third employs a GPU. All three machines are managed and owned by the Parallel Computing Research Group at TU Wien. Table 7.1 provides detailed information regarding the CPU machines. The GPU machine, Tesla, features a 16GB Nvidia Tesla T4 GPU. All machines operate on Debian Linux.

In Appendix A, the results of the command `lscpu` for both Nebula and Hydra can be seen.

Table 7.1: Hardware setup of the CPU Nodes.

Name	Cores per Socket	Sockets	Processor	L3 Cache	Numa Nodes	RAM
Hydra	16	2	Intel Xeon Gold 6130F	44 MiB	2	96 GiB
Nebula	32	2	AMD EPYC 7551	8192K	8	256 GiB

7.1.2 Software Setup

To execute our benchmarks, we employ a total of three distinct parallel backends and two different compilers. Table 7.2 provides an overview of the parallel backends and compilers utilized. The Nvidia compiler, *nvc++*, employs OpenMP as a parallel backend for multicore systems and CUDA for Nvidia GPUs. The GNU C++ compiler, *g++*, employs oneTBB as a parallel backend for multicore systems. However, it currently does not support GPUs.

Table 7.2: Compilers and parallel backends used.

Machine	Compiler	Backend	Target	Compiler Version	Backend Version
Nebula	nvc++	OMP	Multi-Core	22.5-0	4.5
	g++	oneTBB	Multi-Core	12.1.0	2021.7.0
Hydra	nvc++	OMP	Multi-Core	22.5-0	4.5
	g++	oneTBB	Multi-Core	12.1.0	2021.7.0
Tesla	nvc++	CUDA	GPU	22.11-0	11.8

All the machines that were used to run the benchmarks use SLURM [57] as a resource management tool.

In the following plots, the compiler-backend pairing Nvidia with OpenMP will be abbreviated with *NVC (OMP)* and the compiler-backend-pairing GCC with oneTBB with *GCC (TBB)*. The data used to generate the plots labeled with "runtime" was obtained without imposing any limitations on the number of threads. This granted full autonomy to the backends, allowing them to exercise complete control over resource allocation decisions.

7.2 Homogeneous Nested Parallelism

In this section, we present the results of the benchmarks we performed to test *Homogeneous Nested Parallelism*. As each of the benchmarks reveals specific characteristics, we discuss each of them separately. As a short recap all benchmarks involved use the parallel primitive `std::for_each` and have homogeneous workload. The only difference between those benchmarks is the time complexity (e.g., $\mathcal{O}(n)$, $\mathcal{O}(n^2)$, and $\mathcal{O}(2^n)$).

7.2.1 Homogeneous Linear

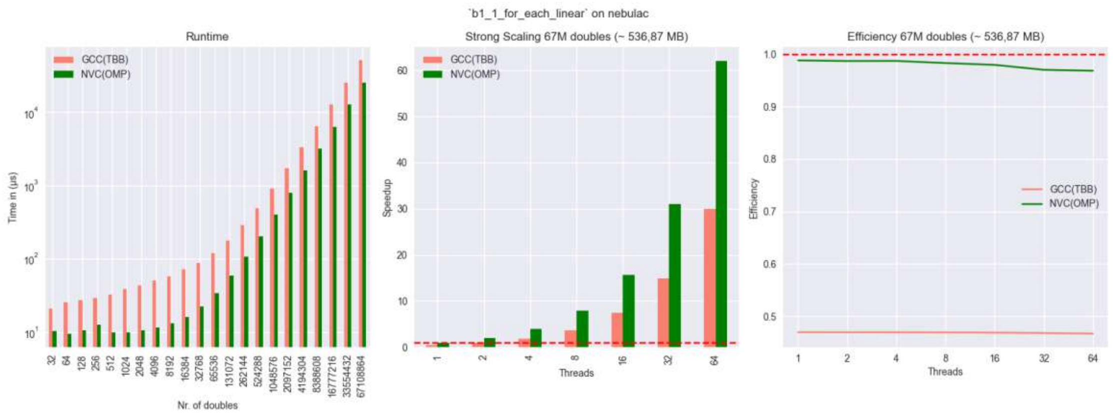


Figure 7.1: Runtime & speedup results for "Homogeneous Linear"; Nebula.

In Figure 7.1, one can see in total three different graphs. Each graph compares two implementations of parallel STL. One is *GCC (TBB)* and the other is *NVC (OMP)*. On the left, we can see the absolute runtime in nanoseconds for input scaling. For instance, the runtime of *GCC (TBB)* for a vector containing 65536 doubles is around 10^2 microseconds (i.e., 100µs). The number of threads when collecting the runtimes for the input scaling was not restricted, meaning the parallel STL backend had full control over how many threads were used. The middle plot shows strong scaling characteristics. To calculate the speedup for strong scaling, we use the formula presented in Section 2.2. For $T(1)$, we use the best achieved time of both serial implementations. For instance, if *GCC (TBB)* has a serial execution time for 67 million doubles of 300ns and *NVC (OMP)* has a serial execution time of 900ns, we take the 300ns as the base. The strong scaling plot also has a red dotted line. This red dotted line marks the point where the parallel implementation achieves speedup over the base (i.e., the best serial implementation). The right plot shows the efficiency for the same input size used for the strong scaling plot. The efficiency is calculated by dividing the speedup of the strong scaling plot by the number of threads. For instance, at 64 threads we can see in the strong scaling plot that the speedup value for *NVC (OMP)* is at around 62. Diving 62 by the number of threads (i.e., 64 threads), we get an parallel efficiency of 0.97. In this plot, we again see a red dotted line, indicating the ideal speedup.

As one can see in Figure 7.1, *NVC (OMP)* achieves faster runtimes than *GCC (TBB)* over all the input sizes when the number of threads is not constrained. Furthermore, *NVC (OMP)* tends to work especially better with smaller input sizes. The superiority of *NVC (OMP)* becomes even more clear when looking at the strong scaling plot. Although both compiler-backend-pairings do achieve a significant speedup over the sequential implementation, *NVC (OMP)* nearly does achieve linear speedup, whereas *GCC (TBB)* lacks behind at around 50% of *NVC (OMP)* efficiency. The efficiency of *NVC (OMP)* only slightly degrades insignificantly at higher thread counts.

7.2.2 Homogeneous Quadratic

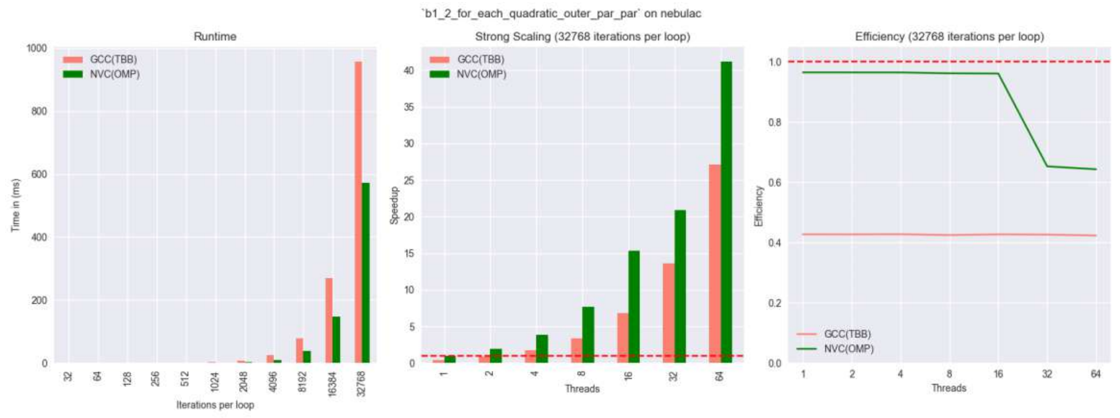


Figure 7.2: Runtime & speedup results for "Homogeneous Quadratic"; Nebula.

Although this benchmark has a quadratic time complexity, the results seen in Figure 7.2 follow the same trend we identified in Section 7.2.1. *NVC (OMP)* is superior to *GCC (TBB)*. What is interesting to see is that there is a significant loss in efficiency going from 16 to 32 threads using *NVC (OMP)*, whereas *GCC (TBB)* stays constant.

7.2.3 Homogeneous Exponential

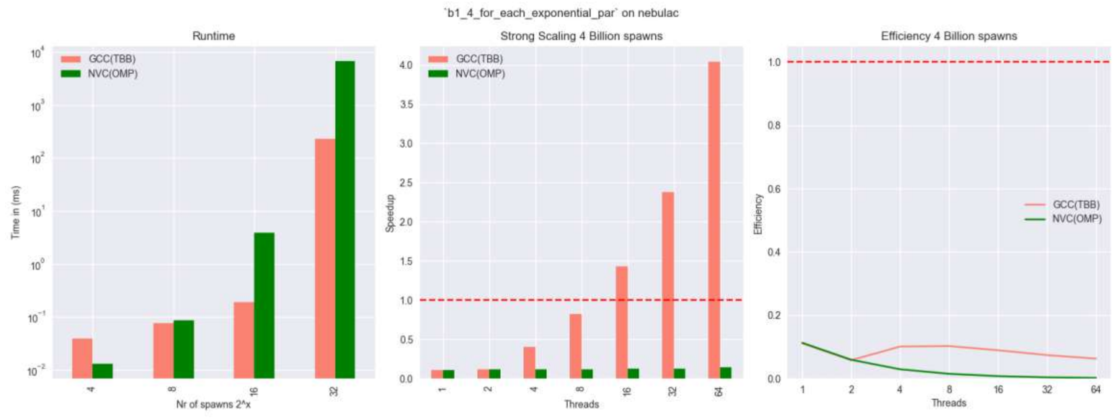


Figure 7.3: Runtime & speedup results for "Homogeneous Exponential"; Nebula.

The results gathered for this benchmark, visualized in Figure 7.3, do not follow the same trend we see in the results for the linear and quadratic time complexity results. For a small number of spawns, *NVC (OMP)* has significantly faster runtimes than *GCC (TBB)*, but as soon as the number of spawns increases the trend switches and *GCC (TBB)* becomes tremendously faster. *NVC (OMP)* struggles when the number of spawns increased, leading to the point where the parallel implementation does not even provide any speedup. In comparison, *GCC (TBB)* does provide a minimal, but still noticeable speedup with a higher number of threads utilized.

7.2.4 Discussion

As we can see from the plots above, *NVC (OMP)* tends to work better on homogeneous workloads, where the time complexity is not exponential. We tried to analyze why we see this kind of behavior, and it seems that OpenMP works fantastic with simple nested loops.

As we can see from above various parallel backends exhibit different performance and scalability characteristics on homogeneous workloads. Hence, the expectation holds that some parallel backends exhibit better performance and scalability when handling nested parallelism for homogeneous workloads.

7.3 Order of Parallelism

In order to test the given expectation, we only use a single benchmark as highlighted in Section 5.2. As a short recap, the benchmark uses two nested `std::for_each` with a homogeneous workload. Only the order of the execution policies is changed.

7.3.1 Homogeneous Quadratic

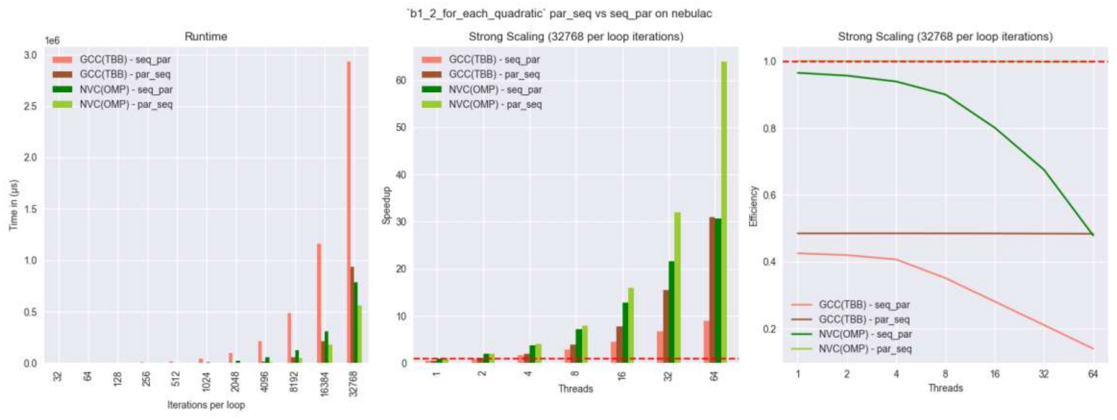


Figure 7.4: Runtime & speedup results for Homogeneous Quadratic execution order; Nebula.

As seen in Figure 7.4, there is a difference in runtime between the order of execution policies. Important to highlight is that already for small input sizes the relative difference between *par_seq* and *seq_par* for *GCC (TBB)* is significantly larger than for *NVC (OMP)*. Although all the compiler-backend-pairing achieves a speedup for every execution order, *NVC (OMP)* does dominate in raw runtime and speedup. Even the fastest execution order of *GCC (TBB)*, is not always faster than the slowest execution order of *NVC (OMP)*.

The efficiency graph shown in Figure 7.4 shows that the execution order *par_seq* for both compiler-backend-pairings keep a constant speedup, whereas the execution order *seq_par* starts to stagnate early on.

7.3.2 Discussion

As we expected, the execution policy order does make a significant performance difference and has an impact on how the parallel backends scale. Hence, the expectation holds that the performance is significantly impacted by the order in which parallelism is applied.

7.4 Heterogeneous Nested Parallelism

In this section, we present the results of the benchmarks we performed in order to test *Heterogeneous Nested Parallelism* presented in Section 5.3. During the analysis of the data gathered from the benchmarks, we identified that all the results showed common characteristics, hence we will not discuss all the results independently. Instead, we will discuss the results on a single representative benchmark namely *Mandelbrot Linear*. As a recap, the benchmark computes the Mandelbrot values for a single row of pixels. The remaining plots can be seen in the Appendix C.

7.4.1 Mandelbrot Linear

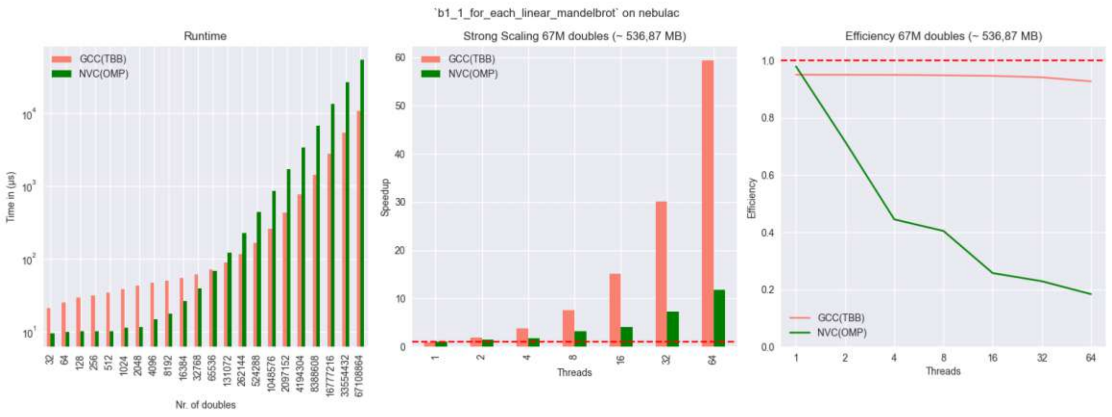


Figure 7.5: Runtime & speedup results for “Mandelbrot Linear”; Nebula.

In Figure 7.5, a comparison between *GCC (TBB)* and *NVC (OMP)* reveals distinct performance characteristics based on input size and scalability. For larger input sizes, *GCC (TBB)* demonstrates faster runtimes than *NVC (OMP)*. However, for smaller input sizes where less heterogeneous work is required, *NVC (OMP)* has superior performance.

When it comes to scalability, *GCC (TBB)* proves to have better strong scaling characteristics than *NVC (OMP)*, with nearly linear speedup. In contrast, *NVC (OMP)*'s efficiency tends to degrade quickly as more threads are added, reaching a point where the benefit of adding additional threads is minimal and not worth the cost. In contrast, the efficiency of *GCC (TBB)* remains consistent over a wide range of thread counts.

7.4.2 Discussion

We examined why *GCC (TBB)* does lack performance when having smaller input sizes. Hence, we looked at the Instructions per second (IPS). As one can see in Figure 7.6, although *NVC (OMP)* has a high number of instructions processed for smaller input sizes, *GCC (TBB)*'s number is flat until the same point (65536 doubles) where *GCC (TBB)* becomes faster than *NVC (OMP)*. This insight tells us that until a certain amount of work is reached the parallel overhead of *GCC (TBB)* is the dominant factor.

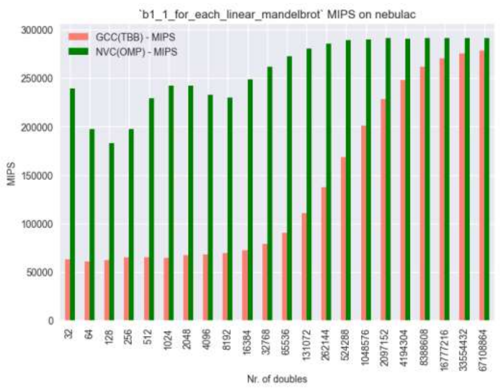


Figure 7.6: MIPS for benchmark "Mandelbrot Linear"; Nebula.

Another interesting point we investigated is that there is a clear difference between the performance and the scaling characteristics of the two backends. While *GCC (TBB)* massively benefits from more threads *NVC (OMP)* does not. After consolidating the documentation of oneTBB[1], we identified that by default oneTBB implements work stealing. In contrast, from the data collected, we see that the *NVC (OMP)* does not implement any kind of work-stealing that would benefit heterogeneous workloads.

Because there is a significant difference in runtime and scaling between the parallel STL backends, the expectation holds that some parallel backends exhibit better performance and scalability when handling nested parallelism for heterogeneous workloads.

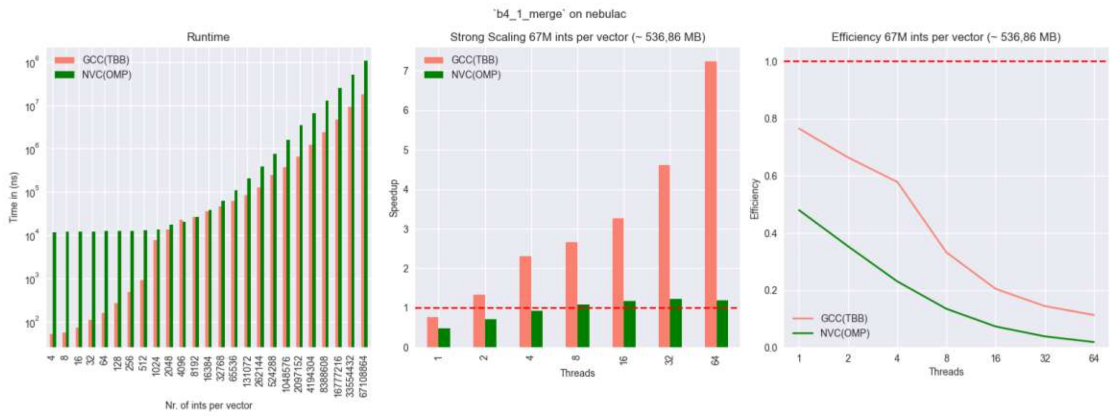
7.5 Sequential Fallback

In this section, we present the results of the benchmarks we performed in order to test *Sequential Fallback* presented in Section 5.4. During the analysis of the data gathered from the benchmarks, we identified that all the results showed common characteristics, hence we will not discuss all the results independently. Instead, we discuss the results

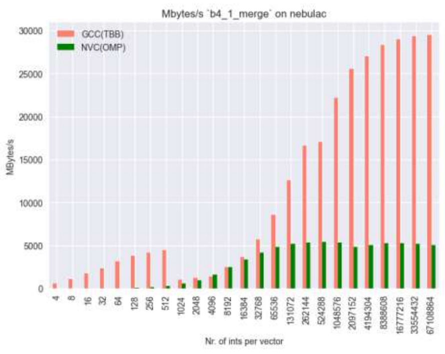
[1]https://oneapi-src.github.io/oneTBB/main/tbb_userguide/How_Task_Scheduler_Works.html

of two representative benchmarks namely *Merge Logic* and *Stable Sorting*. As a recap, the benchmark *Merge Logic* merges two sorted containers using the parallel primitive `std::merge`, whereas *Stable Sorting* stable sorts the elements of a container using the parallel primitive `std::stablesort`. The remaining plots can be seen in Appendix C.

7.5.1 Merge Logic



(a) Runtime, Strong Scaling and Efficiency

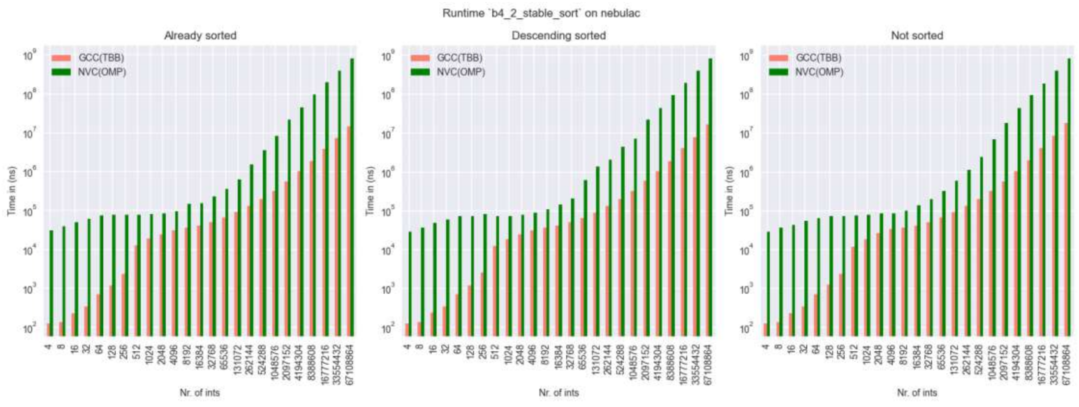


(b) MBytes/s

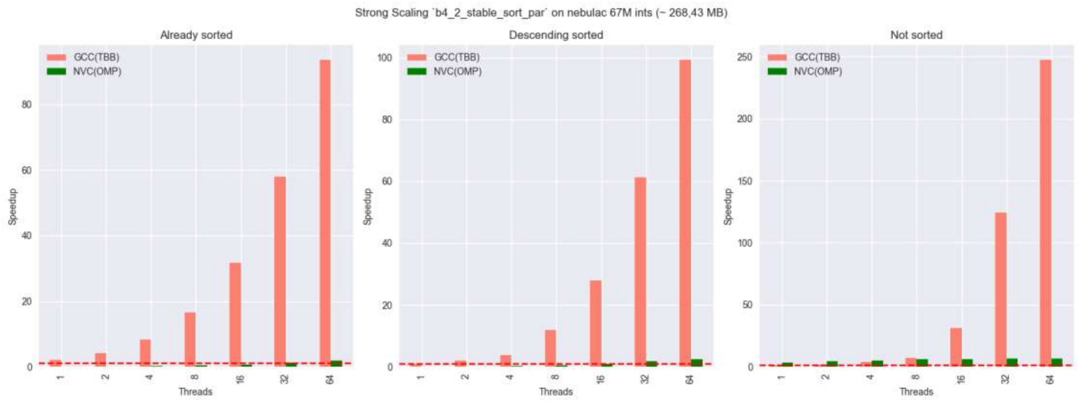
Figure 7.7: Runtime, speedup & throughput results for “Merge Logic”; Nebula.

As one can see in Figure 7.7a, for small input sizes the two backends behave differently. *NVC (OMP)* has a lot of parallelization overhead, which dominates the runtime until the input size of 4096 integers is reached. In comparison, the runtimes of *GCC (TBB)* for small input sizes look rather reasonable. At the input size of 1024, we can see a clear bump in the runtimes for *GCC (TBB)*. Because of this observation, we also looked at the bandwidth, shown in Figure 7.7b, for both compiler-backend-pairings and can see that the bandwidth for *GCC (TBB)* is rising linearly until the point of 1024 integers where it suddenly takes a significant hit. Next, we consider the strong scaling behaviors of the two compiler-backend-pairings. *GCC (TBB)* achieves a significant, although far away from perfect linear, speedup, whereas *NVC (OMP)* struggles to first achieve speedup and then does not improve at all while increasing the number of threads.

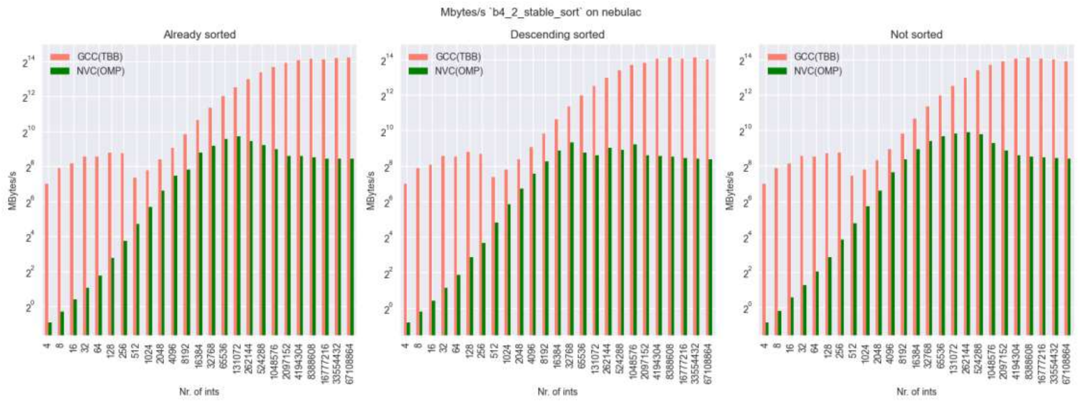
7.5.2 Stable Sorting



(a) Runtime



(b) Strong Scaling



(c) MBytes/s

Figure 7.8: Runtime, speedup & throughput results for “Stable Sorting”; Nebula.

In Figure 7.8a, we can see the same trend as we saw in Section 7.5.1 for the benchmark *Merge Logic*. *GCC (TBB)* has great runtimes until the input size of 256 integers and then at the input size of 512 integers we can see a rather significant jump in the runtime. For *Stable Sorting*, this happens across all the different input styles we tested for this benchmark and is also visible in the bandwidth shown in Figure 7.8c. Interesting to see in Figure 7.8a, the runtimes stay the same for every input style we tested, indicating that neither *GCC (TBB)* nor *NVC (OMP)* have special case treatments in their parallel implementations for `std::stabel_sort`.

Figure 7.8b demonstrates that *GCC (TBB)* has better scaling characteristics compared to *NVC (OMP)*, which we already have seen for *Merge Logic* in Section 7.5.1. What makes the strong scaling behavior noteworthy, is that *GCC (TBB)* achieves a super-linear speedup, suggesting that its parallel STL implementation employs specific parallelization techniques. Especially increasing the number of threads benefits *GCC (TBB)*.

7.5.3 Discussion

In all the benchmarks we can see there is indeed a rather significant jump in runtime whenever a certain threshold is reached. This affects only *GCC (TBB)* and not *NVC (OMP)*. Hence, we looked into the source code of *GCC (TBB)* [58] to understand why we see this strange behavior. Unfortunately, because *NVC (OMP)* is closed source we could not read into it.

Throughout the parallel STL implementation in *GCC (TBB)*, we have identified three thresholds that determine the points at which parallelization is employed. Below you can see each of the thresholds, when for what kind of operations they are used and in which file they can be found:

1. `#define _PSTL_MERGE_CUT_OFF 2000` : Used for various merge operations. Found in file parallel_backend_tbb.h on line 415;
2. `#define _PSTL_STABLE_SORT_CUT_OFF 500` : Used for the stable sort implementation. Found in file parallel_backend_tbb.h on line 1116;
3. `constexpr auto __set_algo_cut_off = 1000` : Used for all the set operations. Found in file parallel_backend_tbb.h on line 3046.

It is important to highlight that the thresholds we found are not always used. For example, the threshold we found for set operations (i.e., `__set_algo_cut_off`) is only used, as we can see in Figure C.8, when the two sets are intersected.

With the insights we gathered from the benchmarks and from reading the source code of *GCC (TBB)* we can say the expectation holds that different compilers/backends may fallback to sequential algorithms, leading to better performance.

7.6 Specialized Parallelism Techniques

In this section, we present the results of the benchmarks we performed in order to test *Specialized Parallelism Techniques* presented in Section 5.5. During the analysis of the data gathered from the benchmarks, we identified that all the results showed common characteristics, hence we will not discuss all the results independently. Instead, we discuss the results of two representative benchmarks namely *Find Elements* and *Vector Partition*. The first benchmark *Find Elements* uses the parallel primitive `std::find` to find a specific element in a container. The second benchmark *Vector Partition*, splits the input vector into two parts where in the first part every element matches the predicate and in the second part not. The remaining plots can be seen in Appendix C.

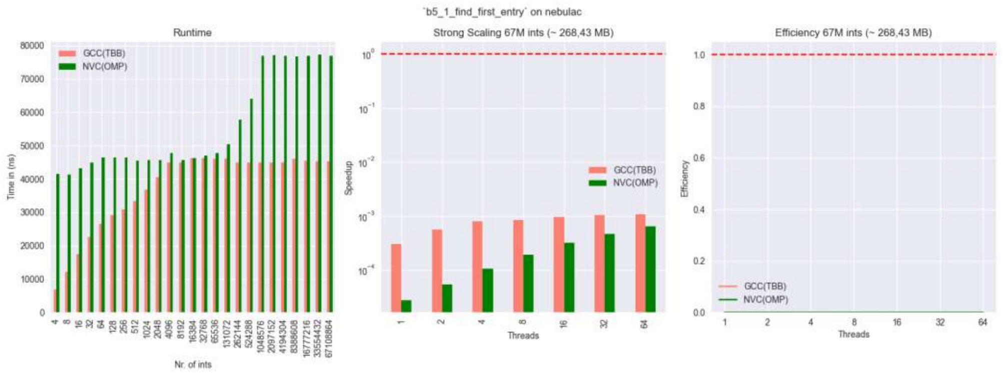
7.6.1 Find Elements

In Figure 7.9 one can see, that both parallel backends behave differently on different input types and sizes. In Figure 7.9a, we can see that the runtime for *NVC (OMP)* can be grouped into two blocks. The first block ranges from 4 to 65536, and the second block ranges from 131072 to 67108864. In contrast, the runtimes of *GCC (TBB)* start to increase and then stabilize at around 45000 ns. Since we know that the first element satisfies the predicate, we expected that a reasonable parallel implementation should be able to detect this early exit opportunity and therefore we would see a constant runtime for every input size. What is interesting to see although is that the runtimes are quite different. We assume this comes down to the fact of how the parallel backend handles scheduling.

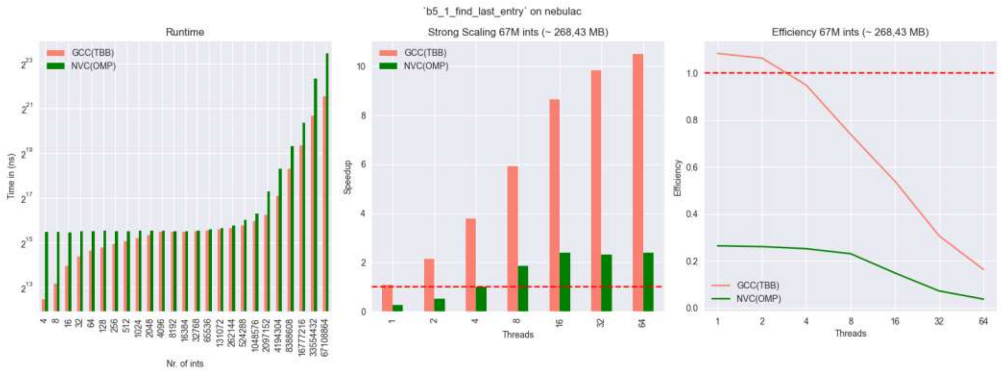
The behavior of the two other input types, shown in Figures 7.9b and 7.9c, is similar. This is interesting to see since searching for a non-existing element requires touching every element, whereas depending on the implementation of the parallel primitive `std::find`, this should not be the case for finding the last element.

7.6.2 Vector Partition

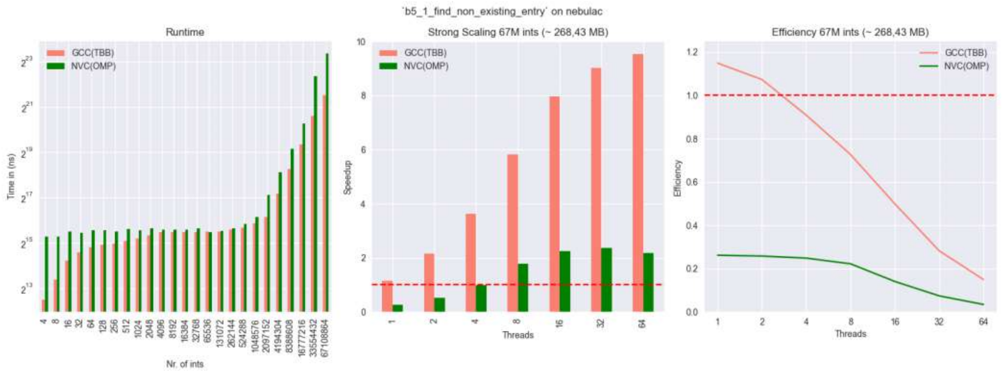
In Figure 7.10, we can see that the two parallel STL backends scale and behave differently. By analysing the runtime graph, one can see that *NVC (OMP)* has quadratic runtime, meaning by doubling the input size we also double the runtime. In contrast, *GCC (TBB)* works well on large input sizes but has a lot of parallelisation overhead for smaller input sizes. The superiority of *GCC (TBB)* continues when looking at the strong scaling characteristics. While *GCC (TBB)* achieve a significant speedup with increasing the number of threads, *NVC (OMP)* does not at all have any speedup. The strong scaling behaviour we identified for *NVC (OMP)* made us a bit suspicious therefore we looked at the efficiency and checked the data again. We collected the number of cycles run on every core using performance counters and visualized the results. As we can see in Figure 7.11, the parallel primitive `std::partition` uses only a single thread. Indicating that *NVC (OMP)* does not use any parallelization at all.



(a) Find first entry



(b) Find last entry



(c) Find non existing entry

Figure 7.9: Runtime & speedup results for “Find Elements”; Nebula.

7.6.3 Discussion

Although some parallel primitives use special techniques to achieve significant speedup over their sequential implementations, not all of the backends we tested exhibit this behavior. Specifically, we found that *GCC (TBB)* is superior for the parallel primitives we tested and the input types. It is noteworthy that the parallel backends make use

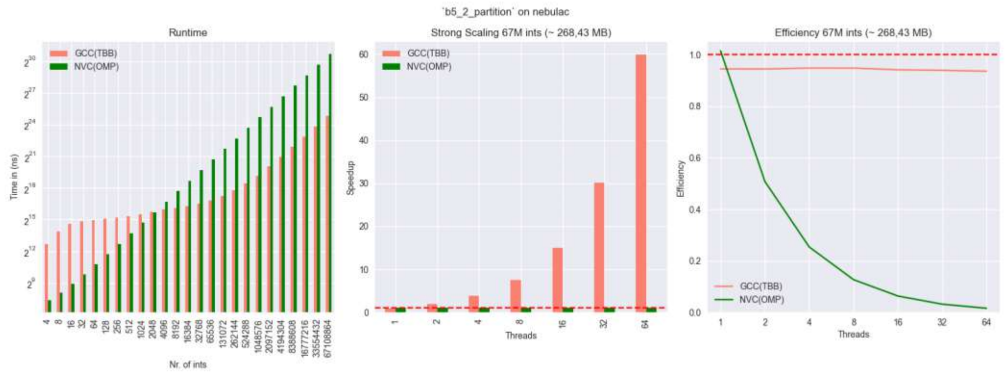


Figure 7.10: Runtime & speedup results for "Vector Partition"; Nebula.

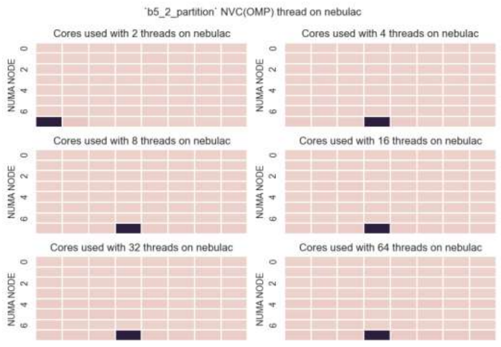


Figure 7.11: Active cores when running Vector Partition using *NVC (OMP)*. Dark spots indicate which Core is active.

of early exit opportunities, as exemplified by *Find Elements*. However, it is surprising that the parallel primitive `std::partition` does not employ any parallelization in *NVC (OMP)*.

Based on our observations, we cannot conclusively state whether the expectation, that backends use special parallelism techniques for (linear) algorithms leading to significant differences in terms of performance and their strong scaling properties, holds or not. While it holds for one of the parallel backends, it is not valid for the other.

7.7 Tailored Optimization for Parallel Scans

In this section, we present the results of the benchmarks we performed to test *Tailored Optimization for Parallel Scans* presented in Section 5.6. Because the results gathered from `b6_1_inclusive_scan` and `b6_2_exclusive_scan` show the same characteristics, we will not discuss both results independently. Instead, we will discuss only the results of *Inclusive Scan*. As a short recap, the benchmark uses the parallel primitive `std::inclusive_scan` to calculate the inclusive scan of the input container. The remaining plots can be seen in Appendix C.

7.7.1 Inclusive Scan

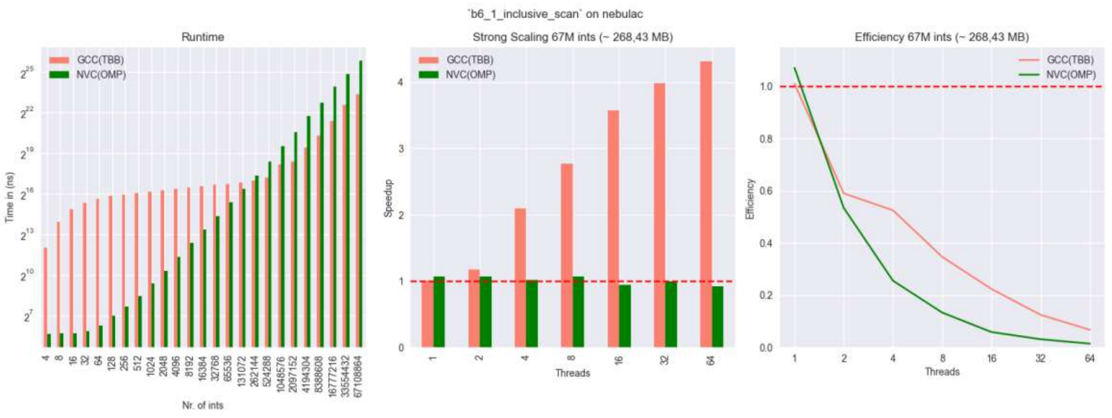


Figure 7.12: Runtime & speedup results for "Inclusive Scan"; Nebula.

As we can see in Figure 7.12, for small input sizes *NVC (OMP)* dominates the runtimes of *GCC (TBB)*. At around 262144 elements this trend turns and the runtimes for *GCC (TBB)* start to significantly improve. What is interesting to see is that the runtimes of *NVC (OMP)* are similar to the ones we saw for *Vector Partition* in Figure 7.10. The superiority of *GCC (TBB)* continues when looking at the strong scaling and efficiency plots. While *GCC (TBB)* achieves a significant speedup, *NVC (OMP)* barely achieve speedup at all.

7.7.2 Discussion

As mentioned above, the characteristics we identified for both benchmarks using *NVC (OMP)* looked quite similar to what was found in Section 7.6.2. Hence, we took a closer look at the raw data points and checked whether we had a mistake in the benchmarking code or in the calculations. After running again our tests for *NVC (OMP)* with performance counters we identified that also `std::inclusive_scan` and `std::exclusive_scan` do not use parallelization.

With that observation, we can say that there is indeed a rather significant difference between parallel STL implementation for those two parallel primitives. Although we do not see unusual high speedup numbers or efficiency for the *GCC (TBB)* implementation, there is still a significant variation in performance and strong scaling properties. Hence, the expectation holds.

7.8 Specific vs. Custom Implementation

In this section, we present the results of the benchmarks we performed in order to test whether the usage of logic-specific parallel primitives is superior to custom implementations. During the analysis of the data gathered from the benchmarks, we identified that all the results showed common characteristics, hence we will not discuss all the results independently. Instead, we will discuss the results of four representative benchmarks namely *Copy Logic*, *All_of Logic*, *Stencil Logic* and *Serial vs Direct Call*. The remaining plots can be seen in Appendix C.

7.8.1 Copy Logic

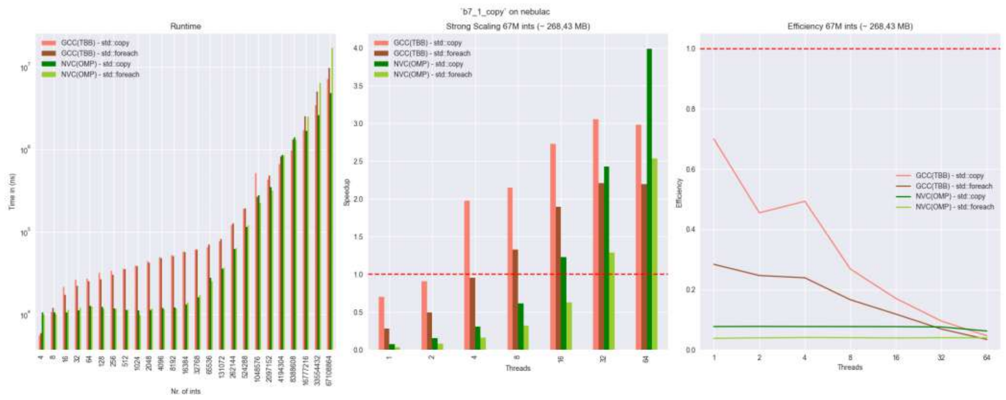


Figure 7.13: Runtime & speedup results for “Copy Logic”; Nebula.

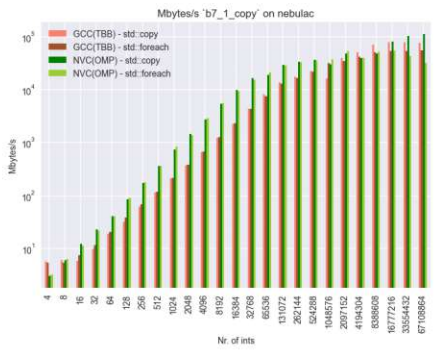
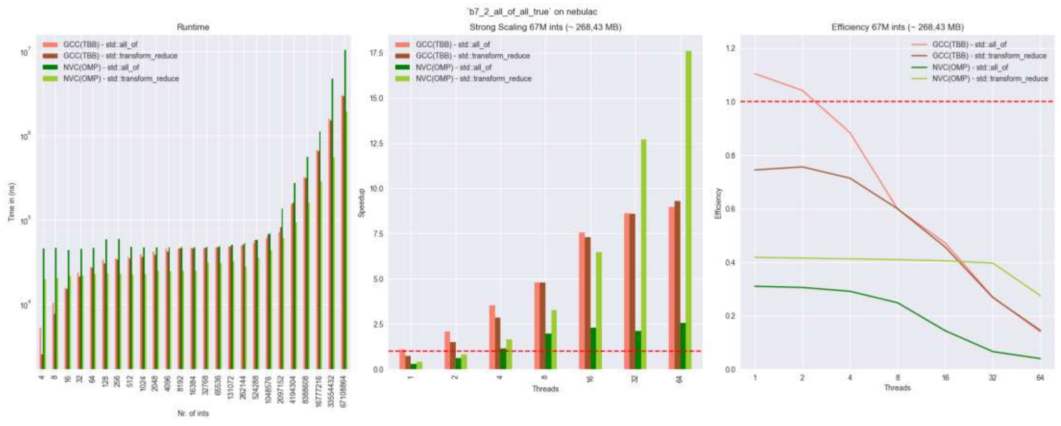


Figure 7.14: Throughput results for “Copy Logic”; Nebula.

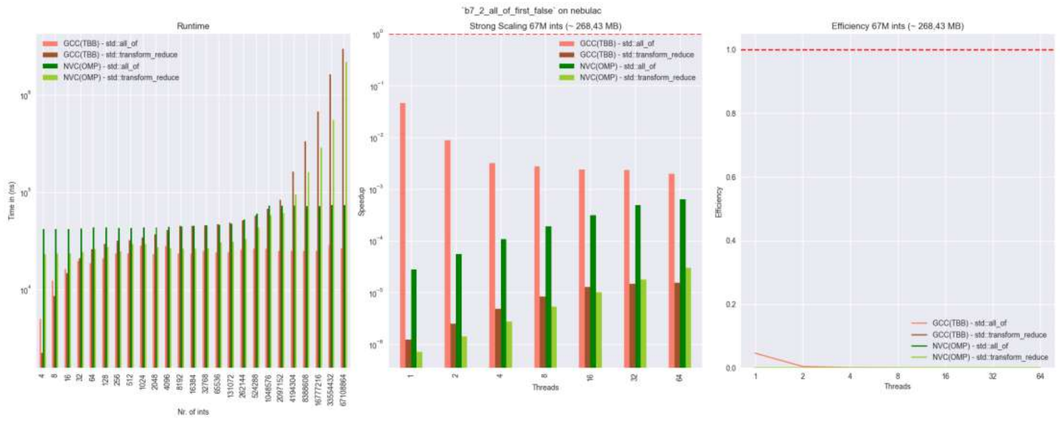
In Figure 7.13, we can see that *NVC (OMP)* works well with small input sizes, whereas *GCC (TBB)* starts to shine with larger input sizes. Due to this observation, we looked at the bytes processed by second, shown in Figure 7.14 and saw that *NVC (OMP)* can utilize the bandwidth way better for both parallel primitives compared to its *GCC (TBB)* counterparts. By looking at the strong scaling and efficiency plots, we can see that the speedup of *NVC (OMP)* increases constantly when increasing the number of threads. In comparison, *GCC (TBB)* starts to achieve a speedup rather early (at four threads) and starts to stagnate at around 32 threads.

Interesting to see is that both parallel backends show the same characteristics when comparing the two parallel primitives. For small input sizes, there is no significant difference in runtime between `std::copy` and `std::for_each`. For larger input sizes, `std::copy` dominates `std::for_each`.

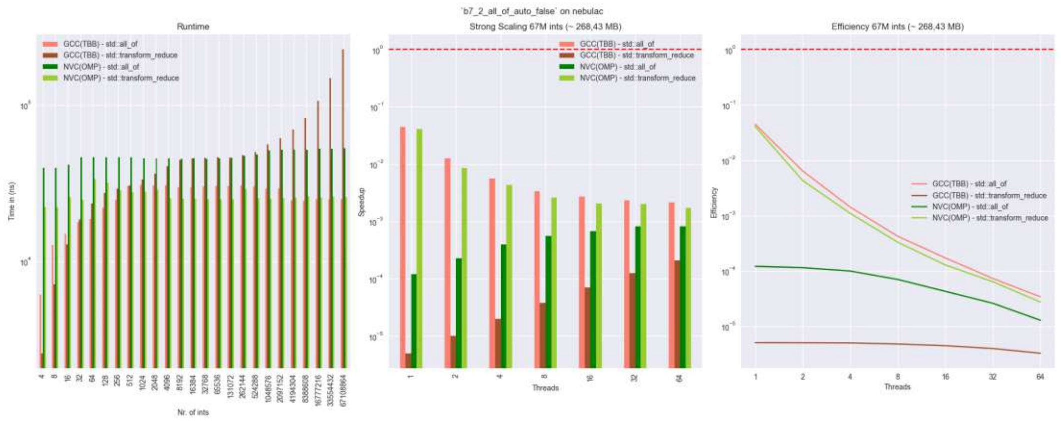
7.8.2 All_of Logic



(a) All elements true



(b) First element false



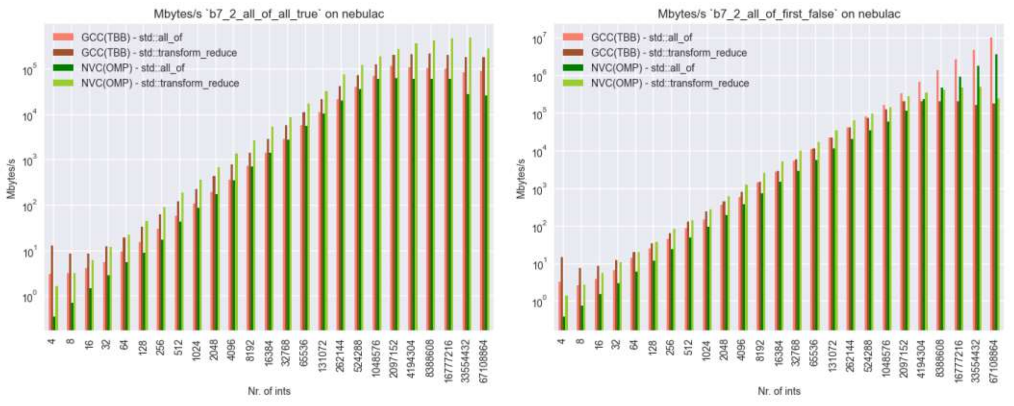
(c) All elements false

Figure 7.15: Runtime & speedup results for “`All_of` Logic”; Nebula.

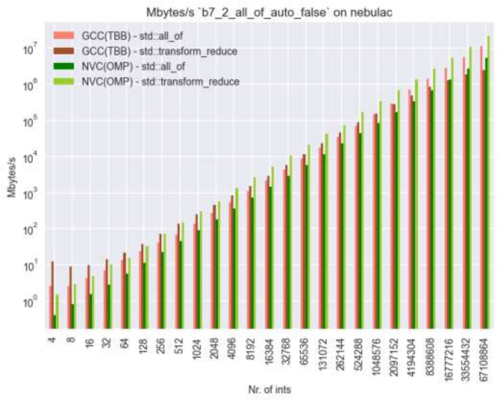
In Figure 7.15, we observe that the performance of `std::all_of` and `std::transform` varies depending on the input type. When an early exit opportunity is present, as demonstrated in Figure 7.15b, *GCC (TBB)*'s `std::all_of` outperforms `std::transform`, and this is also true for *NVC (OMP)*.

Interestingly, *NVC (OMP)*'s `std::transform` performs well with input types where all elements may need to be accessed, as shown in Figure 7.15c, in comparison to `std::all_of`. Our analysis of the bandwidth data depicted in Figure 7.16 suggests that *NVC (OMP)*'s `std::transform` can utilize the bandwidth more effectively than `std::all_of`.

The trend we see for *GCC (TBB)* is not replicated in the trends of *NVC (OMP)*.



(a) All elements true (b) First elements false



(c) All elements false

Figure 7.16: Throughput results for "`All_of` Logic"; Nebula.

7.8.3 Stencil Logic

Figure 7.17 shows that for small input sizes, the performance difference between the parallel primitives `std::transform` and `std::for_each` is negligible on both backends.

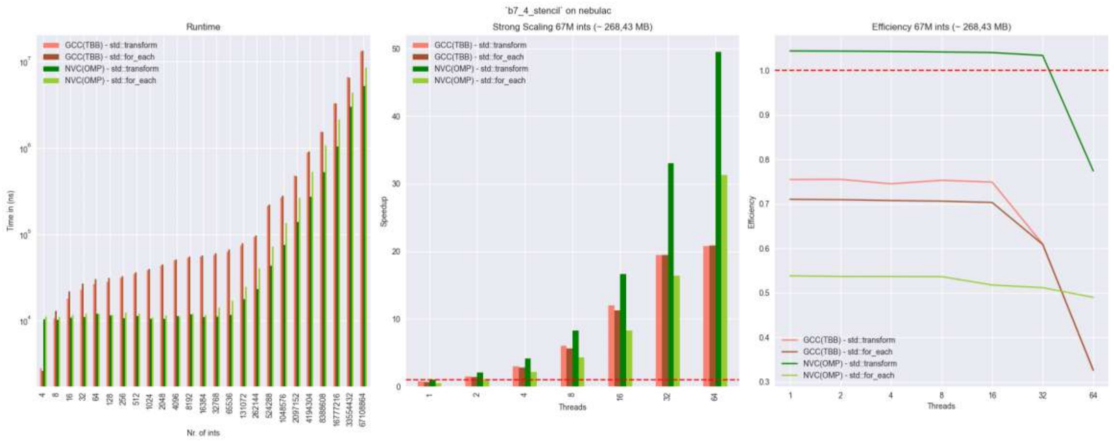


Figure 7.17: Runtime & speedup results for "Stencil Logic"; Nebula.

However, for larger input sizes, a significant difference between the two *NVC (OMP)* primitives can be observed.

It is interesting to note that both primitives scale quite well, achieving significant speedup. It is worth mentioning that, again, there is no significant difference in speedup between the *GCC (TBB)* primitives, while a rather significant difference can be observed between `std::for_each` and `std::transform` for *NVC (OMP)*. Additionally, all primitives except *NVC (OMP)*'s `std::for_each` experience a significant efficiency drop when utilizing 64 threads. If this didn't occur, *NVC (OMP)*'s `std::transform` would have achieved nearly perfect super-linear speedup.

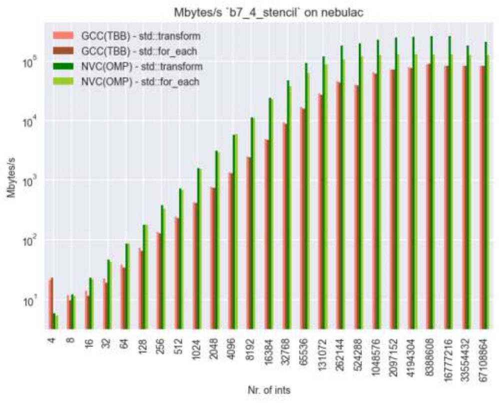


Figure 7.18: Throughput results for "Stencil Logic"; Nebula.

As we observed in Section 7.8.2 with `std::all_of`, *NVC (OMP)*'s `std::transform` outperforms its counterpart. Additionally, Figure 7.18 reveals that for large input sizes, *NVC (OMP)*'s `std::transform` utilizes bandwidth more efficiently than the parallel primitive `std::for_each`.

7.8.4 Serial vs Direct Call

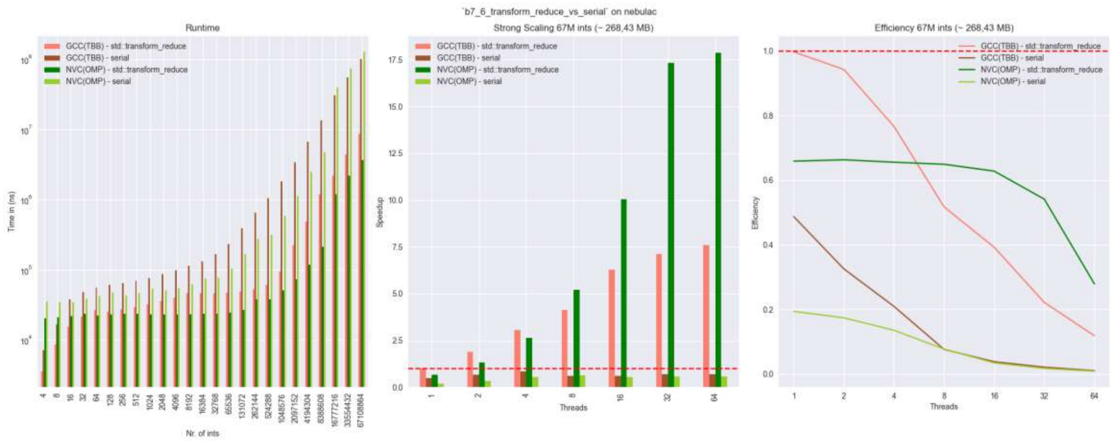


Figure 7.19: Runtime & speedup results for "Serial vs Direct Call"; Nebula.

Figure 7.19 reveals that both *NVC (OMP)* and *GCC (TBB)* exhibit similar characteristics in runtime and strong scaling. In both cases, the serial implementations are dominated by the parallel primitive `std::transform_reduce`. Even for small input sizes, the serial implementation is unable to compete with the other parallel primitive. This dominance becomes more evident when analyzing the strong scaling and efficiency plot. `std::transform_reduce` achieves a significant speedup for both parallel backends and appears to plateau at around 32 threads, whereas the serial implementation does not achieve a speedup over the sequential implementation.

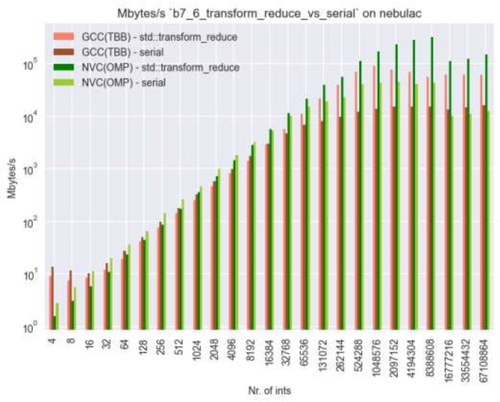


Figure 7.20: Throughput results for "Serial vs Direct Call"; Nebula.

It is noteworthy that *NVC (OMP)*'s `std::transform_reduce` achieves significantly higher speedup and lower runtimes compared to its *GCC (TBB)* counterpart. This prompted us to examine the bandwidth of the parallel primitives, and it is apparent that *NVC (OMP)* utilizes the bandwidth for large input sizes much more efficiently than *GCC (TBB)*.

7.8.5 Discussion

We can see from the benchmarks above, that for some semantics the specific parallel primitive performs and scales better than the custom implementation (for example in Section 7.8.1). But this is not always the case, sometimes implementing the same logic using different parallel primitives leads to significantly better performance. This is especially true for *NVC (OMP)* where most of the time implementing the same semantics with `std::transform` leads to a significant performance boost. From our analysis, we assume this is related to the fact that *NVC (OMP)*'s `std::transform` utilized the bandwidth better.

Hence, we neither can say the expectation, that employing specific parallel algorithms tends to yield superior performance/strong scaling compared to utilizing custom implementations, holds or not, since it heavily depends on the backend and input type.

7.9 Custom Index-based Iterations

In this section, we present the results of the benchmarks we performed in order to test various index-based iteration strategies presented in Section 5.8. During the analysis of the data gathered from the benchmarks, we identified that all the results showed common characteristics, hence we will not discuss all the results independently. Instead, we will directly discuss the results in this section. The detailed plots of all the index strategies can be seen in Appendix C. As a recap, the goal of these tests is to see whether specific index-based iteration strategies are superior to others.

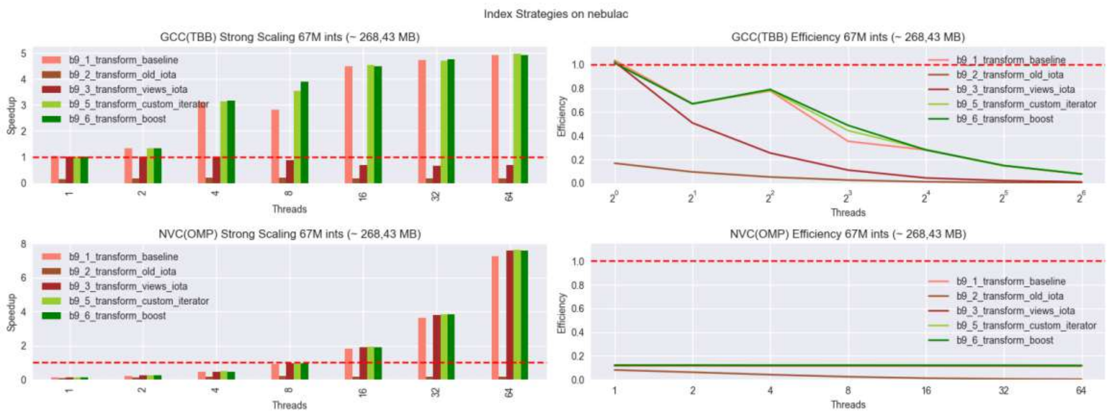


Figure 7.21: Runtime & speedup results for various index strategies; Nebula.

As depicted in Figure 7.21, the performance characteristics and strong scaling can vary drastically depending on the index strategy employed. Surprisingly, the data-centric approach, represented by the orange line in the graph, appears to be almost as efficient as the fastest index-centric approaches, indicating that switching between strategies does not yield a significant performance boost. Notably, creating an index array using `std::iota` has a substantial impact on performance. When comparing the strong scaling plots of *NVC (OMP)* and *GCC (TBB)*, we observe that *GCC (TBB)* achieves a speedup over the sequential implementation early on and stabilizes at around 16 threads, while *NVC (OMP)* takes longer to achieve speedup but does so at a consistent rate.

Furthermore, we noticed that using `std::views::iota` does not result in any speedup for *GCC (TBB)*, whereas it does for *NVC (OMP)*. Upon further investigation, we found that this is a common problem that most developers are not aware of. The `views` implementation in C++ is not compatible with the requirements specified by the parallel STL interface. As a result, depending on how the parallel STL implementation checks if the provided iterator is allowed or not, either the parallel implementation is used or not. In the case of *GCC (TBB)*, the implementation strictly follows the standard, so the implementation using `std::views::iota` does not use parallelization. *NVC (OMP)* does not implement this type of check, hence parallelization is used. This issue has already been highlighted in a proposal by David Olsen [59].

With all the insights presented above, we can say the expectations, that how the index values are generated by the developer can significantly impact the application's performance and scalability, holds.

7.10 Scatter Mapping vs. Compact Mapping

Although the mapping strategy, as presented in Section 5.9, is a crucial characteristic to test in a NUMA-aware system, we have determined that it does not make practical sense to test given the current configuration of the environment. As previously mentioned, parallel STL lacks any interface for controlling NUMA awareness, which lies solely in the hands of the parallel STL backend developers. During the development of the benchmarking suite, *pSTL-Bench*, we observed unusual results, as explained in Section 7.11.1. To mitigate these anomalies, certain steps were taken, rendering the testing of this irrelevant. Although thread pinning will affect the results, our benchmarking would not evaluate parallel STL's functionality but rather the effectiveness of the solution proposed in Section 7.11.1. Nonetheless, we believe the expectation remains significant and have retained it so that other benchmark suites following our proposed framework in Section 5 can incorporate it when using more appropriate NUMA solutions.

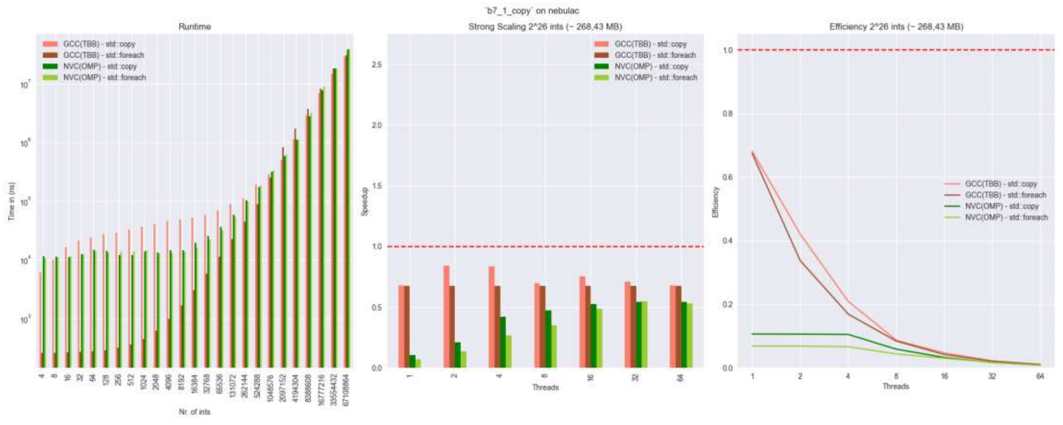
7.11 Crucial Factors for Performance

In this section, we discuss a set of crucial details we identified over the course of our experimental evaluation, that are important to highlight. First, we reflect on all the problems we experienced with the NUMA architecture on both Nebula and Hydra and how we dealt with these problems. Secondly, this is followed by how we managed to control the number of threads for both parallel STL backends. Finally, we will touch on the problem of thread pinning.

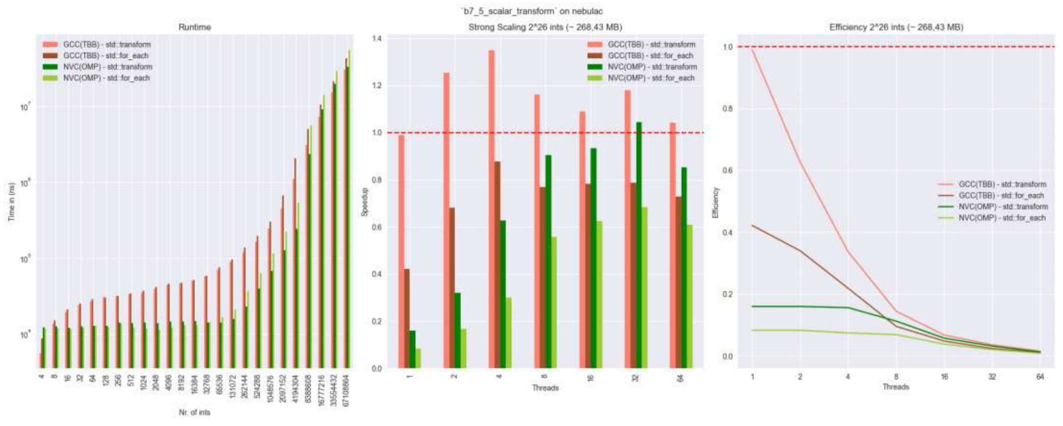
7.11.1 NUMA Aware Allocator

Both shared memory machines used for our experimental evaluation use NUMA. As we can see in Section 7.1.1, *Nebula* has in total eight NUMA nodes and *Hydra* two NUMA nodes. Furthermore, both systems run Linux, where the default memory allocation policy is the *first touch policy* [60]. The *first touch policy* has the effect that memory is allocated only when it is first touched [61]. On NUMA hardware, this implies that the allocation happens on the memory bank near the core that performed the first touch [60, 61]. This behavior can have a significant impact on performance when not considered while writing high-performance code. For example, if one thread located at NUMA node 0 allocates the memory and then another thread positioned at a different NUMA node wants to access it, the access latency will be quite high.

The prototype of *CORE* assumed that NUMA awareness will be handled by default in C++ when using parallel STL backends, but this is not the case. As one can see in Figure 7.22, both parallel backends do not scale at all and barely achieve a speedup.



(a) Benchmark Copy Logic.



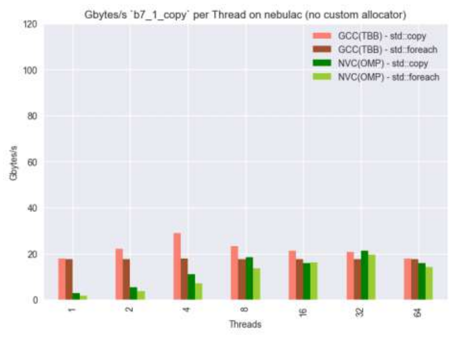
(b) Benchmark Scalar Transform Logic.

Figure 7.22: Non NUMA-aware results on Nebula.

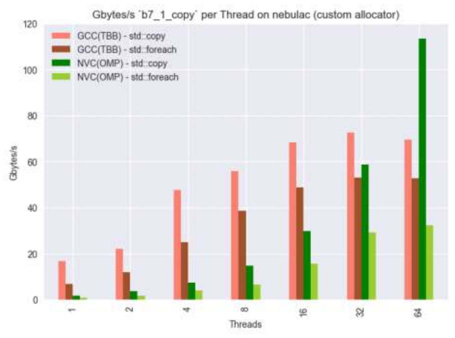
This made us suspicious, hence we took a step back to examine the implementation details of `std::vector`. By default, the implementation of `std::vector` initializes every element in the container. This means if no special allocator is provided to the container, at creation time the main thread will first touch every element of the vector. In NUMA systems, this implies that all the memory will be allocated at the memory bank of the main thread.

Modern parallelization frameworks such as HPX [9], provide custom NUMA aware allocators, but unfortunately, ISO C++ parallel STL does not provide such a feature to our knowledge. Hence, we created our own simplified NUMA aware allocator using HPX's `numa_allocator` as a reference.

The goal of our NUMA allocator is to not rely on any other third party and only use features provided by ISO C++ parallel STL. In Listing 7.1, our shortened custom NUMA Allocator can be seen. The crucial part of this code snippet lies in Line 21, where the execution policy defined earlier is used to touch the first byte of every element. This has the effect that not every element is first touched by a single thread.

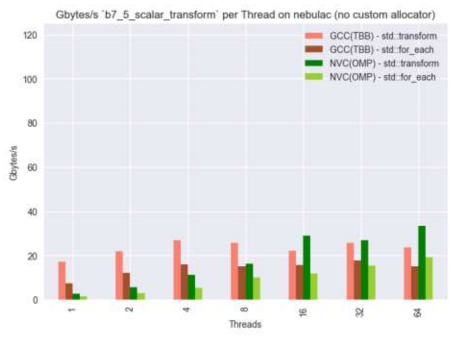


(a) Without custom NUMA Allocator

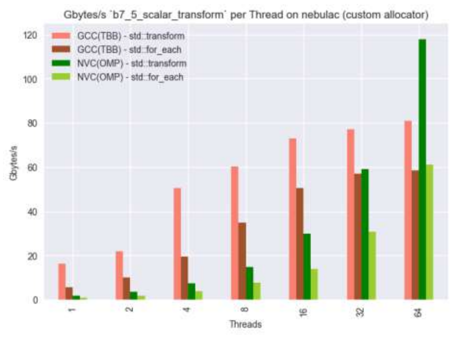


(b) With custom NUMA Allocator

Figure 7.23: Comparing throughput results of allocator for “Copy Logic”; Nebula.



(a) Without custom NUMA Allocator



(b) With custom NUMA Allocator

Figure 7.24: Comparing throughput results of allocator for “Scalar Transform Logic”; Nebula.

Listing 7.1: Implementation of the NUMA Allocator (shortened)

```
template<typename T, typename Policy>
class numa_allocator {
  // ...

  // memory allocation
  pointer allocate(size_type cnt, void const * = nullptr) {
    // allocate memory
    auto p = static_cast<pointer>(::operator new(cnt * sizeof(T)));

    // early exit when we use std::execution::seq
    // because it would be waste of time to seq touch it
    // this already happens by default since vector gets 0 initialized
    if constexpr (IS_SEQ::value) {
      return p;
    }

    pointer begin = p;
    pointer end = begin + cnt;

    // touch first byte of every object using the given strategy
    std::for_each(execution_policy, begin, end, [](T &val) {
      *reinterpret_cast<char *>(&val) = 0;
    });

    // return the overall memory block
    return p;
  }

  // ...
};
```

Figures 7.23 and 7.24 demonstrate the significant impact of the new custom allocator. It notably enhances the bandwidth utilized by parallel STL primitives, thereby leading to a drastic improvement. For instance, the bandwidth of *NVC (OMP)* for 64 threads for the benchmark *Copy Logic* increases by nearly a factor of 8. As a result of this substantial enhancement, all benchmarks used for testing were executed with the custom NUMA aware allocator.

7.11.2 Effects of Thread Pinning

Although the custom NUMA allocator we presented in Section 7.11.1 does improve the performance of the benchmarks, there is still a problem that might not be obvious at first. As we mentioned above, the custom NUMA allocator uses the execution policy defined by the user. This means if the user selects to use parallelism the first touch will be performed using the parallel STL primitive `std:for_each`. The ISO C++ parallel STL lacks fine-grained control over which cores threads should be pinned to. As a result, if two parallel primitives are launched in sequence, the first primitive may run on a different set of cores than the second. This can create a non-obvious problem when the first primitive allocates memory, as the memory will be allocated on the memory banks of the NUMA nodes those threads are running on, due to the first-touch policy. If the second primitive runs on different threads, which may potentially be on a different NUMA node, communication between the NUMA nodes will be required, resulting in a performance hit. This issue is more likely to be problematic on systems with a high number of NUMA nodes and is particularly noticeable when using a smaller number of threads.

This effect can be seen in Figures 7.24 and 7.23, where the number of threads is increased from one to two, although the bandwidth does not increase at all. These effects are more noticeable in the results of *Nebula* but are also present in *Hydra*.

There is a quite straightforward solution to this problem and many of the frameworks used in the parallel STL backends provide means to control this. The problem although is that there is no ISO C++ interface for this, meaning users have to hope for the parallel STL backend developer to use those options correctly behind the scenes or provide a non-standardized option for users to control it.

For example, oneTBB the framework used in the parallel STL backend in *GCC (TBB)*, has in total three different ways of controlling this thread affinity:

1. `affinity_partitioner`: provides the scheduler with placement hints,
2. `static_partitioner`: manually defining where to place threads,
3. Using NUMA aware task arenas.

To our knowledge, none of these three options listed above is used by the parallel STL backend of *GCC (TBB)*. Furthermore, there are no options to control this externally. The lack of this implementation or control option can be seen in Figures 7.25. By utilizing hardware counters, we gathered data on the number of cycles executed by each core during the benchmark. This information provided valuable insights into the activity status of individual cores throughout the execution. It is crucial to acknowledge that the displayed placement represents just one of the potential configurations. As one can see, the placement of the threads for 4, 8, 16 and 32 threads is not optimal. The darker the spot the more cycles were executed on this core.

Fortunately, since *NVC (OMP)* uses OpenMP as its parallelization framework, we have the option to control its behavior with non-standard environment variables. As we can see in Figure 7.26, the environment variables do have an effect on the placement leading to minimal better performance.

Because we wanted to stay conform with the parallel STL interfaces and did not want to use non-standardized interfaces as little as possible, we executed the benchmarks without

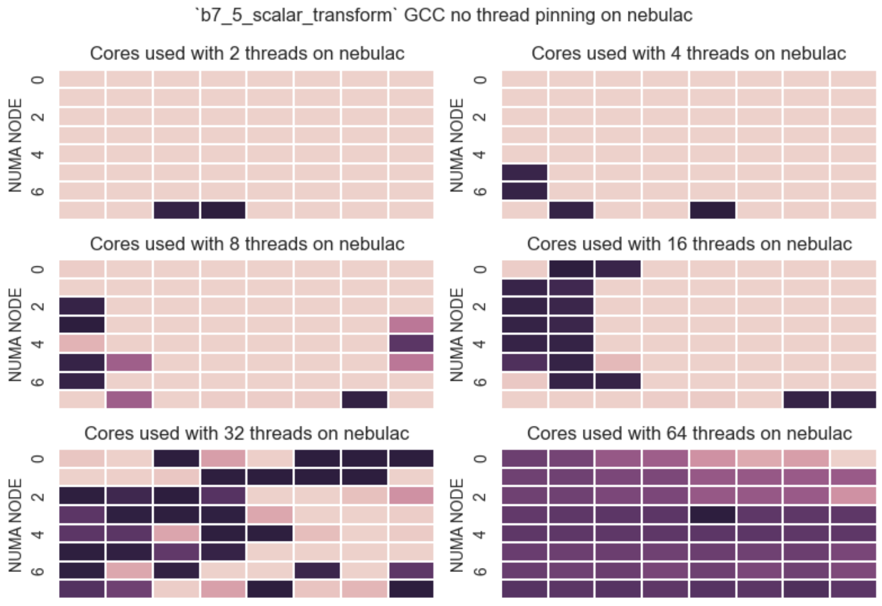
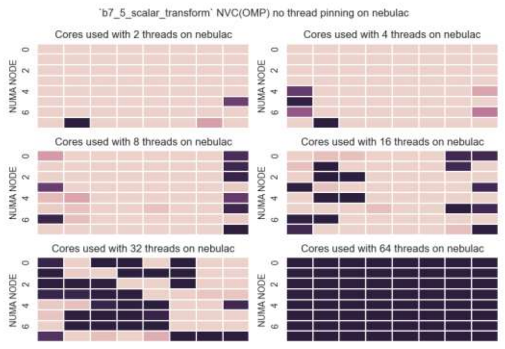
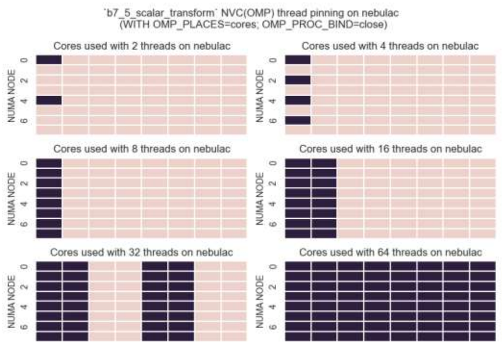


Figure 7.25: One possible GCC(TBB) Default Thread Pinning for Scalar Transform Logic on Nebula.



(a) No Thread Pinning



(b) With Thread Pinning

Figure 7.26: NVC(OMP) Thread Pinning for Scalar Transform Logic on Nebula.

interfering with thread pinning at all. Furthermore, it is still important to highlight that we still encounter major differences in strong scaling behavior between memory-bound benchmarks on *Hydra* and *Nebula* which we, unfortunately, could not explain to ourselves.

7.11.3 Controlling Number of Threads

As already mentioned in Section 7.11.2, the ISO C++ interface of parallel STL does not provide any means to control threading. Since a major part of our analysis is strong scaling, we were forced to find a way to control the number of threads used. *NVC(OMP)* provides a non-standardized way of controlling the number of threads used by using the environment variable `OMP_NUM_THREADS`. Unfortunately *GCC (TBB)* does not provide such an environment variable, meaning we had to alter our code. Luckily, oneTBB provides an interface `tbb::global_control` to control rudimentary configurations.

Not being able to control such rudimentary configuration options such as the maximum number of threads is rather problematic and does limit the usability of ISO C++ parallel STL a lot!

CHAPTER 8

Performance Portability

In this chapter, we cover the topic of performance portability between the parallel backends used during the experimental evaluation. Furthermore, we present how performance portability using ISO C++ between different architectures can be achieved.

In Section 8.1, we present how we try to quantify performance portability between the parallel backends. This is followed by Section 8.2 where we discuss how GPU offloading can be done with standard ISO C++. Finally, we discuss performance portability in ISO C++ with parallel STL in Section 8.3.

8.1 Quantifying Performance Portability

As already mentioned in Chapter 5, as a side product of testing all those expectations we gather a lot of metrics that are useful to quantify performance portability between the parallel STL backends *NVC (OMP)* and *GCC (TBB)*. While creating concrete implementations to test our expectations, we focused on creating benchmarks that are tailored to gather as much information as possible to later quantify performance portability. Furthermore, we focused on writing as much ISO C++ code as possible without using any third-party non-standardized parallelization framework interfaces. By writing C++ code as normally done, we can assure that our benchmarks were not tailored to a single parallel STL backend.

While Pennycook et al.'s definition [4] is commendable, it is necessary for the purpose of this thesis to differentiate between two distinct forms of performance portability. Hence, we classify performance portability into the following two categories:

1. *Inter-Backend Performance Portability*: Quantifies performance portability of a given application between parallel STL backends on the same architecture/platform

(e.g., performance portability between a set of parallel STL Backends that target the same shared memory machine);

2. *Cross-Machine Performance Portability*: Quantifies performance portability of a given application across a wide range of architectures/platforms (e.g., performance portability of a single backend that can target multiple GPU vendors).

In this thesis, we will only focus on *Inter-Backend Performance Portability.* Since at the time of writing this thesis, there was only a single compiler-backend-pairing that targeted Nvidia GPUs, we can only calculate *Inter-Backend Performance Portability* for the two parallel backends that target shared memory machines (i.e., *NVC (OMP)* and *GCC (TBB)*).

8.1.1 Metrics

As shown in Chapter 3, Pennycook et al. [4] contributed a significant method for quantifying performance portability (Equation (3.1)). In this thesis, we use this exact method to quantify performance portability between the two parallel backends, *NVC (OMP)* and *GCC (TBB)*, used in the experimental evaluation. Hence, it is only required to define the metric $e_i(a, p)$.

In the literature, various metrics, such as speedup [1, 2, 24] or peak memory bandwidth [23, 24], were used to quantify performance portability. This thesis uses three different metrics presented below. It is important to highlight that our thesis followed a black box testing approach, meaning we do not have in-depth knowledge of how the parallel STL backends internally implement the parallel primitives logic. This black-box testing approach had to be done since for the parallel STL backend *NVC (OMP)* the source code was not available.

Since we are focusing on *Inter-Backend Performance Portability*, it is important to highlight that for the following metrics, the set of platforms H consists only of the parallel STL Backends *NVC (OMP)* and *GCC (TBB)*, and not the machines Hydra and Nebula. This means that for both Nebula and Hydra, we get independent performance portability scores. Hence, when calculating the performance portability score for a specific benchmark on a given machine X, the values, such as maximum speedup or minimum recorded runtime, from the other machines are ignored. Getting independent results allows us to cross-check whether our results are skewed or not. Important to note is that when we talk about plattform in the context of this thesis, we always mean either *NVC (OMP)* or *GCC (TBB)*. A good way of thinking about the metrics is that there only exists one machine (e.g., imagine that only Hydra exists and there is no Nebula, and vice-versa).

Runtime (R) This metric evaluates the similarity between the runtime of the parallel STL backend and the best-recorded runtime overall, using the largest input size (which was also employed for strong scaling analysis) and without restricting the number of threads used by the backend. Hence, $e_i(a,p) = \frac{min\_time(a)}{t_i(a,p)}$, where $min\_time(a)$ denotes the minimum time across all the platforms for an application a, and $t_i(a,p)$ denotes the fastest recorded time to solve problem p with application a on platform i.

Speedup (S) This metric measures the proximity between the maximum speedup achieved using all available cores (and the biggest input size) and the highest speedup achieved across all the backends on the machine we are quantifying on (i.e., either Nebula or Hydra) with the same biggest input size. Hence, $e_i(a,p) = \frac{sp_i(a,p)}{max\_sp(a)}$, where $max\_sp(a)$ denotes the maximum achieved speedup (using all available cores with the biggest input size) across all the platforms for an application a and $sp_i(a,p)$ denotes the maximum speedup achieved to solve problem p with application a on platform i using all available cores and the biggest input size.

Bytes processed per second (Bps) This metric evaluates the bandwidth achieved by the parallel backend for the largest input size, which was also utilized for the strong scaling analysis, without any restrictions on the number of threads, and compares it with the highest recorded bandwidth across all the backends. Hence, $e_i(a,p) = \frac{bps_i(a,p)}{max\_bps(a)}$, where $max\_bps(a)$ denotes the highest recorded bandwidth across all platforms for an application a, and $bps_i(a,p)$ denotes the maximum bandwidth achieved to solve problem p with application a on platform i.

Table 8.1 displays the metrics used by each benchmark, as well as the parallel primitives utilized in the benchmark. As one can see the majority of benchmarks use the runtime metric, whereas only a small selection uses *Speedup* and *Bps*. The reason behind this is that the raw runtimes work well in a black-box approach. This metric is rather straightforward to use and does not require in-depth knowledge of the parallel backends' implementation. For a selection of benchmarks, we decided to use *Bps*. We selected this metric because we had already gathered sufficient data for it during testing. In addition, while testing, we observed significant variations in the values' behavior across different backends. For the remaining computation-bound benchmarks, we opted for *Speedup* because we noticed that it was the most effective in capturing the varying strong scaling behavior of the computation-bound benchmarks.

8.1.2 Example Calculation for "Homogeneous Linear"

To make it easier for the reader to follow how the performance portability scores were calculated, we demonstrate the calculation for the benchmark "Homogeneous Linear" on Nebula. As highlighted multiple times above we use Pennycook et al. [4] formula to quantify performance portability (Equation (3.1)). As we can see in Table 8.1 we use the metric *Speedup* for the benchmark "Homogeneous Linear". For benchmarks using the metrics *Runtime (R)* and *Bytes processed per second (Bps)* the calculation stays the same, the only difference is how $e_i(a,p)$ will be calculated.

Table 8.1: Benchmarks and metrics which were used to evaluate performance portability.

Metric	Benchmark	Parallel primitive
S	Homogeneous Linear	std::for_each
	Homogeneous Quadratic	std::for_each
	Homogeneous Exponential	std::for_each
	Mandelbrot Linear	std::for_each
	Mandelbrot Quadratic	std::for_each
Bps	Merge Logic	std::merge
	Stable Sorting	std::stable_sort
	Set Union Logic	std::set_union
	Set Difference Logic	std::set_difference
R	Find Elements	std::find
	Vector Partition	std::partition
	Unique Copy	std::unique_copy
	Min Max Elements	std::minmax_element
	Inclusive Scan	std::inclusive_scan
	Exclusive Scan	std::exclusive_scan
	Copy Logic	std::copy, std::for_each
	All_of Logic	std::all_of, std::transform_reduce
	Count_if Logic	std::count_if, std::transform_reduce
	Stencil transform Logic	std::transform, std::for_each
	Scalar transform Logic	std::transform, std::for_each
	Serial vs Direct call	std::transform, std::reduce, std::transform_reduce
	Data-Centric	std::transform data centric
	Index-Centric with iota	std::transform with std::iota
	Index-Centric with views::iota	std::transform with std::views::iota
	Index-Centric with Custom Iterator	std::transform with custom iterator
	Index-Centric with Boost	std::transform with boost

The first step is to define the platforms for which we want to calculate the performance portability $\mathbf{P}(a, p, H)$. On Nebula we have two platforms *NVC (OMP)* and *GCC (TBB)*, implying $H = \{NVC, GCC\}$ and $|H| = 2$.

The next step is to calculate $e_i(a, p)$ for each $i \in H$ (i.e., $e_{nvc}(a, p)$ and $e_{gcc}(a, p)$). For this we follow the definition of the metric *Speedup (S)* defined in Section 8.1.1:

$$e_{nvc}(a,p) = \frac{sp_{nvc}(a,p)}{max\_sp(a)} \quad (8.1) \qquad , \text{and} \qquad e_{gcc}(a,p) = \frac{sp_{gcc}(a,p)}{max\_sp(a)} \ . \quad (8.2)$$

The values $sp_{nvc}(a, p)$ and $sp_{gcc}(a, p)$ can be read from Figure 7.1. As a reminder, $sp_i(a, p)$ is the speedup achieved on platform i using all cores for the biggest input size, thus

$$sp_{nvc}(a,p) = 61.98 \quad (8.3) \qquad , \text{and} \qquad sp_{gcc}(a,p) = 29.88 \ . \quad (8.4)$$

Using the values of $sp_{nvc}(a, p)$ and $sp_{gcc}(a, p)$, one can calculate the value of $max\_sp(a)$

the following:

$$\begin{aligned} max\_sp(a) &= max(sp_{nvc}(a,p), sp_{gcc}(a,p)) \\ &= max(61.98, 29.88) \\ &= 61.98 \quad . \end{aligned} \tag{8.5}$$

The next step is to insert the values from Equations (8.5) and (8.3) into Equation (8.1) (respectively Equation (8.4) for Equation (8.2)), resulting in

$$e_{nvc}(a,p) = \frac{61.98}{61.98} = 1 \tag{8.6}$$

, and

$$e_{gcc}(a,p) = \frac{29.88}{61.98} \quad . \tag{8.7}$$

The final step is to use the values $e_{nvc}(a,p)$ and $e_{gcc}(a,p)$ to calculate the performance portability score Φ:

$$\begin{aligned} \Phi(a,p,H) &= \frac{|H|}{\sum_{i \in H} \frac{1}{e_i(a,p)}} \\ &= \frac{2}{\frac{1}{e_{nvc}(a,p)} + \frac{1}{e_{gcc}(a,p)}} \\ &= \frac{2}{\frac{1}{1} + \frac{1}{\frac{29.88}{61.98}}} \\ &= 0.65055519 \approx 65\% \quad . \end{aligned} \tag{8.8}$$

The performance portability score for the benchmark "Homogeneous Linear" on the machine Nebula is 65%. The same value can be seen in Figure 8.1.

8.1.3 Results

In Figures 8.1, 8.2, and 8.3, we can see the results of calculating the performance portability metrics for the benchmarks.

The findings from our experiments, as described in Sections 7.2 and 7.3, have been further substantiated by the results presented in Figure 8.1. Specifically, we observed that simple homogeneous workloads scale well on both parallel backends, while the scaling of complex homogeneous workloads and heterogeneous workloads depends on the parallel backend.

Figure 8.2 illustrates that performance portability is heavily dependent on how effectively the parallel backends utilize the underlying parallel architecture for memory-bound operations. The observations visible in the figure align with the metrics presented in Section 7.5.

The observations we described in the experimental evaluation in Chapter 7 are also reflected in Figure 8.3. For instance, the low-performance portability score for *Inclusive Scan* can be attributed to the fact that `std::inclusive_scan` does not use parallelization in *NVC (OMP)*. The same holds for *Vector Partition*.

It is important to highlight that for some benchmarks, such as `B4_2_decrement_sorted`, the performance portability is on one machine rather high and on another machine it is significantly lower. We tried to investigate this problem and we believe that it is closely linked to a scenario where one group of threads initially accesses the data, while another group of threads subsequently processes it. The significance of this problem becomes more pronounced with an increase in the number of NUMA nodes. We emphasized the importance of this problem in Section 7.11.2. Unfortunately, we cannot provide a fully satisfying explanation of why this problem occurs.

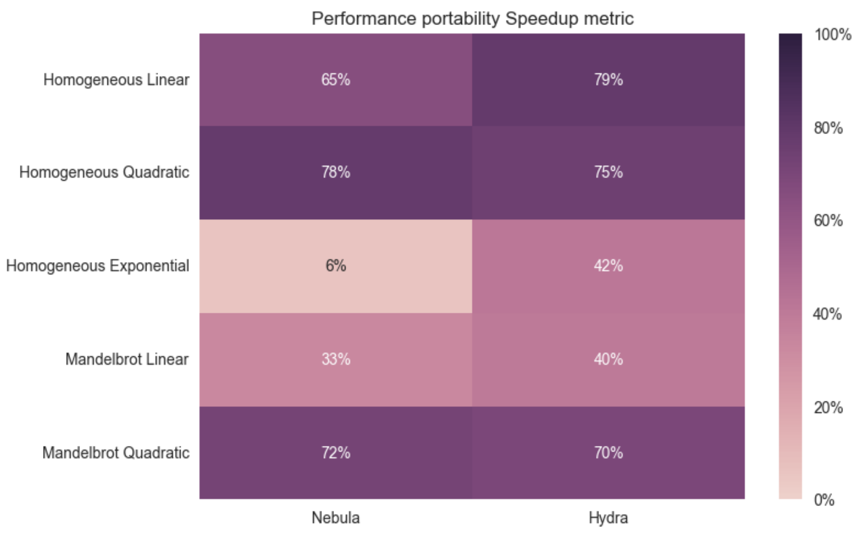


Figure 8.1: Performance Portability results for Speedup metric.

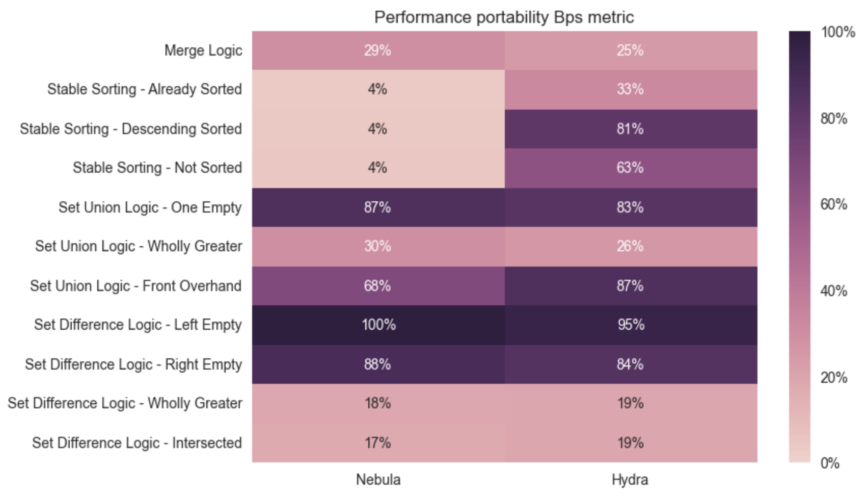


Figure 8.2: Performance Portability results for Bsp metric.

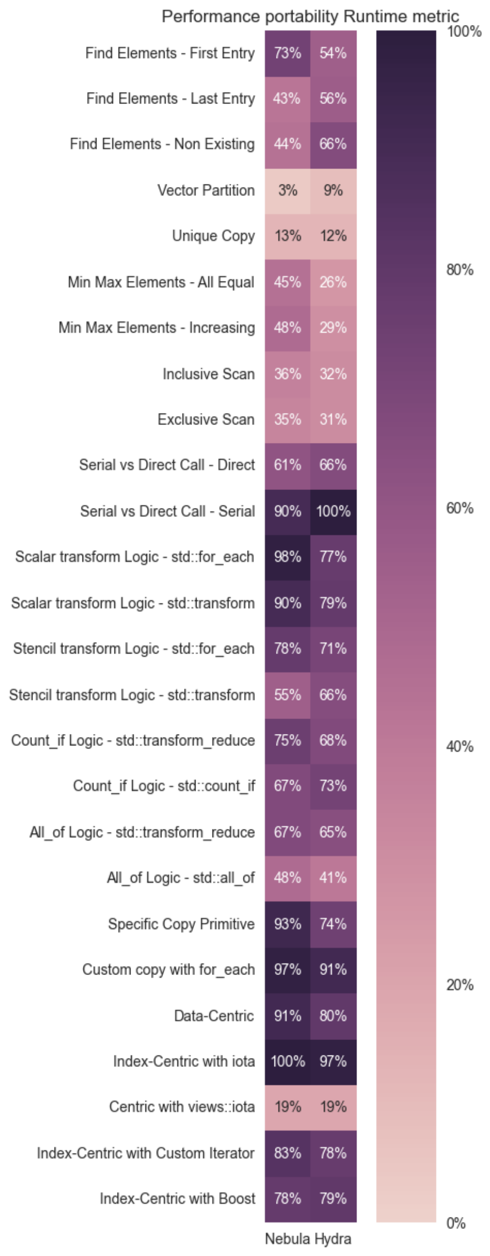


Figure 8.3: Performance Portability results for Runtime metric.

8.2 GPU Offloading

One major advantage of ISO C++ parallel STL is that developers can write standard C++ code and decide at compile time on which accelerator to execute. For instance, this allows developers to write familiar C++ code and run it on GPUs. One such parallel STL backend that does support GPU offloading is Nvidias *stdpar*. In the related work

Chapter 3, we already highlighted that *stdpar* was used in the research community to achieve performance portability for GPUs. In this section, we present the challenges we faced during GPUS offloading and try to simply model for a selection of benchmarks when it makes sense to offload to GPUs.

8.2.1 Challenges

During the modelling stage, it was discovered that working with the ISO C++ parallel STL and the parallel backend *stdpar* presents numerous challenges that significantly impact the way developers can use *stdpar*. In this section, we highlight some of the challenges we encountered.

Nested Parallelism As of writing this thesis, *stdpar*, does not support nested parallelism. This means rather complex parallelism techniques such as those used in the Mandelbrot benchmarks are not possible to implement. Although there are ways to somehow bypass this restriction by manually unfolding nested parallelism calls into a single parallel primitive, this is not a silver bullet. This restriction means that switching from *GCC (TBB)* to *stdpar* is more than just changing the compile time configuration.

Unified Memory Stdpar relies on CUDA Unified Memory [25]. Furthermore, the memory between the CPU and GPU is automatically managed by the environment, meaning data is moved depending on usage. But there is a catch, only memory that is allocated on the heap is managed, meaning any allocation happening on the GPU or CPU stack is not accessible and vice versa. Since ISO C++ parallel STL does not provide an interface to control memory management this means there is also no way to manually resolve any problems related to memory management. At first, this might not sound like a major problem, but this implies that everything has to be allocated on the heap. When writing C++ code, a developer normally tries to allocate as little as possible in the heap because this is rather expensive. Furthermore, a developer using *stdpar* has to keep the architectural constraints in mind before switching parallel STL backends. Meaning that switching parallel STL backends (e.g., from *GCC (TBB)* to *stdpar*) might imply rewriting significant parts of the program. This defeats the point of parallel STL.

Iterators Although the C++ parallel STL only requires providing forwards iterator (i.e., only the increment has to be implemented), *stdpar* does require random access iterators. Not complying with this requirement leads to compile time errors. This stricter requirement again may require developers to rewrite or adapt their existing code when switching from a different parallel STL backend to *stdpar*. This again defeats the point of parallel STL.

GPU vs CPU code Since *stdpar* allows code to be executed on GPUs the internals of the parallel STL Backend have to distinguish between CPU and GPU Code. Although from a developer's perspective, all the code looks the same, the code that will be executed on the GPU will be compiled into the target GPUs machine code. This means that at creation time developers have to be fully aware of which function and code sections will be executed on which target platform. Olsen et al. [25], advise using function objects or lambdas in parallel primitive, such that a clear separation is also visible in the code.

8.2.2 Modeling

In this section, we try to model when it would be beneficial to offload to GPUs using the parallel STL backend *stdpar*. In the benchmark suite *pSTL-Bench* we can find a wide variety of benchmarks, and it would not be feasible to model for every benchmark independently when to offload or whether it makes sense. As already highlighted in Section 8.2.1 many of the benchmarks within our suite cannot be offloaded because of various previously mentioned constraints. For modeling, in total four different benchmarks were selected which will be analyzed and discussed:

- *Mandelbrot Linear*: computes the Mandelbrot value for a single row of pixels;
- *Merge Logic*: merges two already sorted containers into a single result container;
- *Inclusive Scan*: computes the inclusive scan; and
- *Specific Copy Primitive*: copies every element of on container to another.

This collection of benchmarks includes both computation-bound and memory-bound workloads. Furthermore, our understanding is that these benchmarks are well-suited for GPUs from a theoretical standpoint, and are also common workloads that one might expect to use in parallel STL. It is important to note that these benchmarks are not indicative of typical GPU workloads, and are solely included to determine when it is beneficial to switch to *stdpar* and to evaluate the process of offloading to GPUs.

Below, we present figures that utilize CPU time. This is the minimum runtime achieved by both backends (*NVC (OMP)* and *GCC (TBB)*) on both *Nebula* and *Hydra* machines. This approach ensures that we use the best possible time within the environment we are working with.

Mandelbrot Linear

In Figure 8.4 we can see two plots. The plot on the left, shows the absolute runtimes of the Tesla T4 GPU using stdpar and the best achieved runtime on a CPU (either Hydra or Nebula). The right plot shows the speedup of the Tesla T4 GPU over the best achieved runtime on a CPU. The red dotted line indicates the point where speedup is achieved over the CPU (i.e., values above the line indicate there is a speedup and values below the line indicate there is no speedup).

As one can see in Figure 8.4, GPU offloading is already feasible for small input sizes. For this specific benchmark, at 4096 integers (representing 4096 pixels) the runtime of the

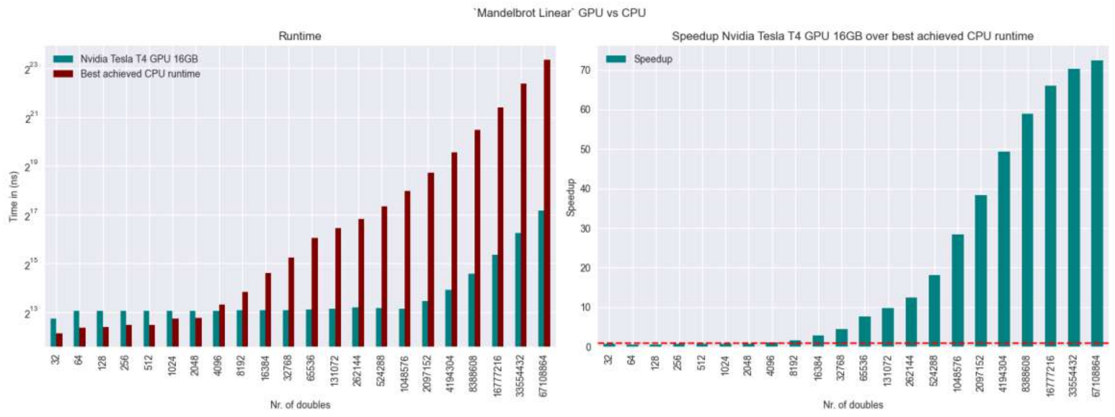


Figure 8.4: GPU Offloading comparison for Mandelbrot Linear.

GPU outperforms the best-recorded CPU time. Especially for larger input sizes, the speedup is rather significant and seems to flatten at around 70x over the best possible CPU runtimes. This kind of heterogeneous workload seems to scale fairly well on the GPU, hence similar workloads seem to be reasonable to offload at already small input sizes.

Merge Logic

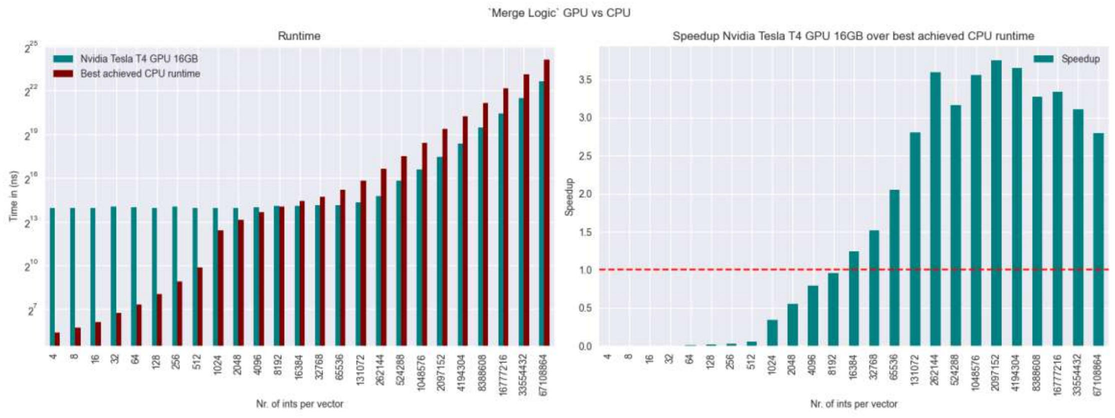


Figure 8.5: GPU Offloading comparison for Merge Logic

As presented in Figure 8.5, the performance of the GPU for smaller input sizes is rather bad. To our knowledge, this is because *stdpar* does copy both input vectors that should be merged from the CPU to the GPU. Only later at around 131072 integers per vector, the amount of work to merge the two containers surpasses the amount of work to copy the data around. Although the speedup is not as significant as the one we saw before for the previous benchmark. The GPU does indeed provide a speedup for larger input sizes.

Inclusive Scan

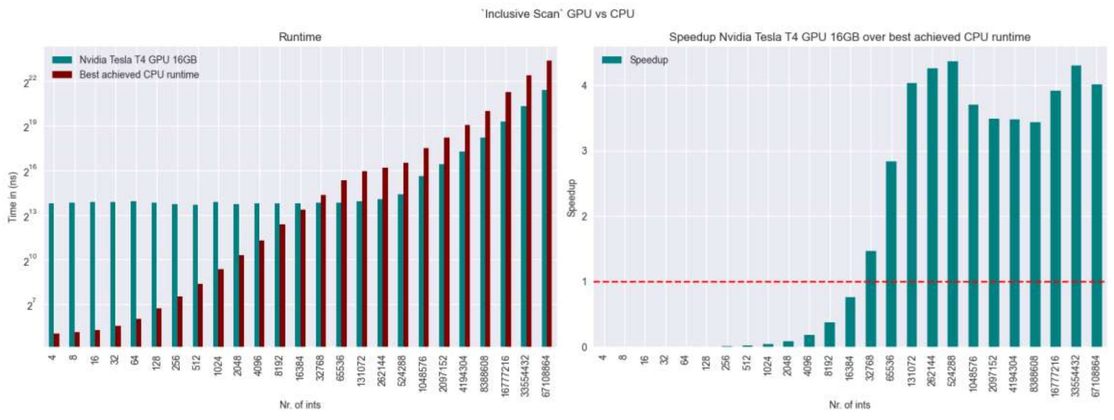


Figure 8.6: GPU Offloading comparison for Inclusive Scan.

In Figure 8.6, we can see the same trend as we saw in *Merge Logic*. For smaller input sizes the amount of work to copy the data around dominates the runtime. Only at larger input sizes, where the computation on the GPU dominates, offloading does strive.

Specific Copy Primitive

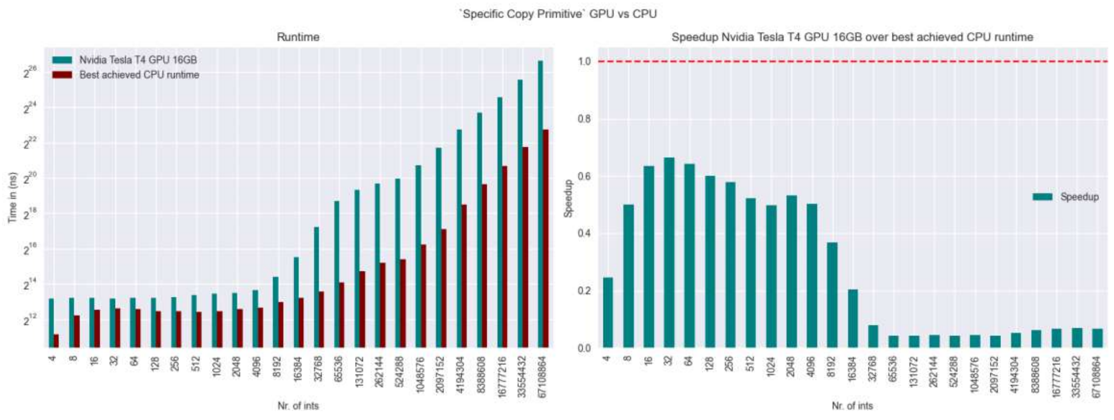


Figure 8.7: GPU Offloading comparison for Specific Copy Primitive.

The impact of the expensive operation of transferring data back and forth between the CPU and GPU is shown in Figure 8.7. It might be tempting to assume that offloading a basic parallel primitive, such as `std::copy`, could lead to a significant performance boost, but this is not the case. In reality, the amount of work required to copy the input and output data outweighs the workload executed by the GPU, making the offloading infeasible for the `std::copy` primitive.

As we can see from all the results presented above it is not possible to model when it is feasible to offload to GPUs using *stdpar*. Although the offloading for some benchmarks proves to be of tremendous advantage, this cannot be said in general. As of now, *stdpar* has a major bottleneck when it comes to copying data around. This has to be considered when working with *stdpar*. For workloads that only need to share a small portion of data between the GPU and CPUs, offloading is advisable. As soon as the parallel primitive used requires copying a large amount of data, offloading becomes rather expensive.

8.3 Discussion

Although the idea of the parallel STL interface is rather appealing to C++ developers there are still a lot of things that need to be sorted out to be competitive with third-party libraries such as Kokkos. Although some of our benchmarks did indeed result in high-performance portability, there are still some primitives that diverge massively. This discrepancy in the performance when switching parallel STL backends for the same platform is rather disturbing. One might expect that the performance between parallel backends targeting the same platform is almost the same only with minor details, but this is not the case. The expectation of true performance portability that is not reflected in reality, may trick C++ developers into switching from already established performance portability frameworks to ISO C++ parallel STL and experience major setbacks.

From a sole code writing experience, ISO C++ parallel STL is fantastic. The parallel STL interface is a rather simple interface that does indeed allow developers to work with parallel hardware rather easily. The large collection of parallel primitives makes it easy for developers to write performant code, but this large collection can be also rather daunting. Using the wrong parallel primitive can have a major impact on the performance and scaling behavior of the program. For high-performance applications where one wants to tweak every little detail to get the most out of the hardware, the C++ parallel STL interface is not the right choice.

Although one might expect that switching paradigms or hardware accelerators when using parallel STL is rather straightforward as promised, this is not the case as we saw when using GPUs with *stdpar*. Although the parallel STL interface stays the same, a lot of "hidden" changes are required. Those hidden changes impact the concept performance portability a lot. The sole purpose of performance portability is to write code abstracted away from paradigms and concepts and achieve the best possible performance. This although is not the case with ISO C++ parallel STL for non CPU accelerators. When using GPUs (with the *stdpar* parallel backend) one often has to rewrite the code. This defeats the whole purpose of using ISO C++ parallel STL. Another burden when using GPUs as a target is that developers are also required to write code with the mindset of using a GPU, meaning one is nudged into writing GPU-aware code. This has the effect that when switching platform or paradigm the code may be poorly performing or not working at all.

Although there is a clearly defined interface for the parallel STL, the parallel backends that implement this interface can do whatever they want as long as they follow the constraints given by the interface. This is desired, since it allows the parallel backends to optimize and apply tricks to achieve the highest possible performance. Despite the freedom, there is a caveat. Developers who write their code using a single parallel STL backend may be inclined to optimize their code specifically for that backend. While this may seem reasonable at first, it becomes problematic when switching to a different backend, as each backend may employ different optimization techniques of which the developer may be unaware, leading to poor performance. Therefore, instead of writing performance-portable code with parallel STL, developers end up writing code that is optimized for a specific backend, sacrificing the potential for performance portability.

CHAPTER 9

Conclusion

Although ISO C++ parallel STL is a rather new addition to the ISO C++ standard, the research community has already started to look into it. Furthermore, there are already parallel STL backends that allow developers to offload C++ code to different parallel hardware, such as Shared-Memory Multiprocessors and GPUs. One of the main objectives of ISO C++ parallel STL is that it aims to provide performance portability. Although performance portability is not something new, some frameworks and languages support it, the research community is still rather active. To answer our research questions and provide a final verdict on performance portability with ISO C++ parallel STL, we crafted a set of expectations to test. For every expectation we provided an in-depth explanation of why it is important to test and what benchmarks will be involved. The proposed expectations are rather abstract and not implementation-specific, allowing third parties to use those to develop their own benchmark suite. We implemented our own benchmark suite *pSTL-Bench* using them. Our concrete benchmark implementations to test those expectations cover a wide variety of parallel STL primitives. The main objective of our benchmark suite is to be as parallel backend agnostic as possible. This allowed us to be as flexible as possible, with what backends we want to include in our experiments. We provided an in-depth explanation behind the reasoning that went into the development of the benchmark suite. The development of the benchmark suite was an iterative approach, since during the experimental evaluation we constantly identified new challenges and problems with the parallel STL, such as NUMA awareness or the effects of thread pinning.

Our experimental evaluation was done on two parallel machines *Hydra* and *Nebula*, and the two parallel STL backends *GCC (TBB)* and *NVC (OMP)* were used. The experimental evaluation allowed us to not only gain an in-depth insight into the parallel backends but also provided us with a basis for quantifying performance portability and crafting the best practices we identified while working with the two parallel STL backends. Our experimental evaluation focused on two different parallel STL backends, which are GCC with oneTBB and Nvidia with OpenMP. Both of those backends target Shared-Memory Multiprocessors.

9.1 Best Practices

Over the course of testing the expectations, we gather a lot of knowledge of the two parallel STL Backends *GCC (TBB)* and *NVC (OMP)*. Using these insights, we can come up with best practices when working with the C++ parallel STL (especially for the two backends). Of course, it is of utter importance to note that in a high-performance computing setting, there is no one size fits all solution and one should always benchmark various approaches and solutions before committing to one. Hence, we will present in this section a set of best practices we identified while working with C++ parallel STL in our specific environment.

Nested Parallelism Implementing nested parallelism correctly can be a challenging task, and it is important to analyze whether the work within the nested parallelism remains consistent or varies. This critical characteristic can have a significant impact on the selection of the appropriate parallel backend to utilize.

Premature Optimizing Parallel primitives are known to optimize their code, which is why it is advisable to check the source code of your parallel STL backend for pre-existing optimizations targeting specific edge cases before implementing your own. If accessing the code for the backend is not possible, benchmarking is a reliable alternative to verify whether optimizations for particular edge cases are already present.

"Missing Implementations" Despite the C++ parallel STL interface offering a comprehensive selection of parallel primitives, it's important to note that not all parallel STL backends provide feature completeness. Although non-implemented "parallel" primitives may offer minimal and insignificant speed-ups, they can deceive users into believing that parallelization is being employed. Therefore, when in doubt, it is recommended to benchmark the number of threads that are being utilized to confirm the extent of parallelization.

Identify high performing primitives Through our data analysis, we have determined that on our hardware, *NVC (OMP)* exhibits exceptional performance when paired with the parallel primitive `std::transform`. This knowledge can be leveraged in a production environment to attain the last bit of additional performance. As discussed in Section 5.7, the logic behind many parallel primitives can be replicated with other primitives. Armed with the understanding that a particular parallel primitive excels in performance, one can utilize this information to develop a faster implementation with `std::transform`. However, it should be noted that there are still scenarios where a more specific parallel primitive may perform better. Thus, benchmarking remains crucial.

Stick to data centric The C++ parallel STL adopts a data-centric approach, although it is possible to follow an index-centric approach, it can be risky due to certain pitfalls that may not be immediately apparent. Our thesis highlights one such pitfall, namely that not all parallel STL backends are fully compliant with the C++ standard. This means that while certain features may perform well on one backend, they may not scale effectively on others.

9.2 Performance Portability with ISO C++ Parallel STL

To quantify performance portability we first proposed a set of metrics. Since we had to work in a black-box testing approach, our set of metrics where rather simple, but still effective. The results of quantifying performance portability reflected our results of the experimental evaluation. For example, for some parallel primitives, the performance portability is rather low because only one of the parallel STL backends provides a parallel implementation. During our experimental evaluation and while quantifying performance portability, we observed that the results obtained on one parallel machine did not always match those obtained on a different parallel machine. We attempted to investigate this issue, but unfortunately, we have not yet found a satisfactory answer. Further investigation is needed in future work to resolve this discrepancy.

In addition, we examined the feasibility of offloading parallel STL primitives to GPUs using *stdpar* and attempted to determine when it would make sense to do so. Our analysis revealed that there is no general model for offloading to GPUs, as each parallel primitive has unique characteristics. However, we did discover that due to the way *stdpar* is implemented, parallel primitives that require a lot of data copying between the GPU and CPU are not well-suited for GPU offloading, as the copy process is quite expensive.

In conclusion, we have examined the concept of performance portability with ISO C++ parallel STL. While the idea of performance portability is present, there is still a long way to go. C++ parallel STL is still in its early stages of development and has significant potential. In future versions of C++, the interface of parallel STL must be extended to give developers greater control over finer details. Additionally, it should be possible to have index-centric kernels such that developers do not have to implement their own potentially badly performing workarounds.

Currently, there are limited options for parallel STL backends, with most only offering offloading to multicore systems. To our knowledge, only one backend supports GPU offloading. In future work, it would be interesting to explore the inclusion of additional accelerators and propose new parallel STL backends that support them. Since our evaluation only covered two parallel STL backends for CPUs and one for GPU, future work should also consider including more parallel STL backends, such as Windows MSVC.

Bibliography

[1] N. Mietzsch and K. Fuerlinger, "Investigating Performance and Potential of the Parallel STL Using NAS Parallel Benchmark Kernels," in *2019 International Conference on High Performance Computing & Simulation (HPCS)*, pp. 136–144, July 2019.

[2] A. S. Dufek, R. Gayatri, N. Mehta, D. Doerfler, B. Cook, Y. Ghadar, and C. DeTar, "Case Study of Using Kokkos and SYCL as Performance-Portable Frameworks for Milc-Dslash Benchmark on NVIDIA, AMD and Intel GPUs," in *2021 International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*, (St. Louis, MO, USA), pp. 57–67, IEEE, Nov. 2021.

[3] J. Kelling, S. Bastrakov, A. Debus, T. Kluge, M. Leinhauser, R. Pausch, K. Steiniger, J. Stephan, R. Widera, J. Young, M. Bussmann, S. Chandrasekaran, and G. Juckeland, "Challenges Porting a C++ Template-Metaprogramming Abstraction Layer to Directive-Based Offloading," in *Accelerator Programming Using Directives* (S. Bhalachandra, C. Daley, and V. Melesse Vergara, eds.), Lecture Notes in Computer Science, (Cham), pp. 92–111, Springer International Publishing, 2022.

[4] S. J. Pennycook, J. D. Sewall, and V. W. Lee, "A Metric for Performance Portability," *International Workshop in Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems*, pp. 1–7, Nov. 2016. arXiv:1611.07409 [cs].

[5] D. Beckingsale, R. Hornung, T. Scogland, and A. Vargas, "Performance portable C++ programming with RAJA," in *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*, PPoPP '19, (New York, NY, USA), pp. 455–456, Association for Computing Machinery, Feb. 2019.

[6] H. Carter Edwards, C. R. Trott, and D. Sunderland, "Kokkos: Enabling manycore performance portability through polymorphic memory access patterns," *Journal of Parallel and Distributed Computing*, vol. 74, pp. 3202–3216, Dec. 2014.

[7] SYCL, "SYCL - C++ Single-source Heterogeneous Programming for Acceleration Offload," Jan. 2014. https://www.khronos.org/sycl/.

[8] HIP, "ROCm-Developer-Tools/HIP," Mar. 2023. https://github.com/ROCm-Developer-Tools/HIP.

[9] H. Kaiser, M. Simberg, B. Adelstein Lelbach, T. Heller, A. Berge, J. Biddiscombe, A. Reverdell, A. Bikineev, G. Mercer, A. Schaefer, K. Huck, A. Lemoine, T. Kwon, J. Habraken, M. Anderson, S. Brandt, M. Copik, S. Yadav, M. Stumpf, D. Bourgeois, A. Nair, D. Blank, G. Gonidelis, R. Stobaugh, N. Gupta, S. Jakobovits, V. Amatya, L. Viklund, P. Diehl, and Z. Khatami, "STEllAR-GROUP/hpx: HPX V1.9.0: The C++ Standards Library for Parallelism and Concurrency," Feb. 2023. https://doi.org/10.5281/zenodo.7672443.

[10] D. J. Kuck, "A Survey of Parallel Machine Organization and Programming," *ACM Computing Surveys*, vol. 9, pp. 29–59, Mar. 1977.

[11] S. Kurgalin and S. Borzunov, *A Practical Approach to High-Performance Computing*. Cham: Springer International Publishing, 1 ed., 2019. https://doi.org/10.1007/978-3-030-27558-7.

[12] T. Sterling, M. Anderson, and M. Brodowicz, "Chapter 2 - HPC Architecture 1: Systems and Technologies," in *High Performance Computing: Modern Systems and Practices* (T. Sterling, M. Anderson, and M. Brodowicz, eds.), pp. 43–82, Boston: Morgan Kaufmann, Jan. 2018.

[13] L. H. Ceze, "Shared-Memory Multiprocessors," in *Encyclopedia of Parallel Computing* (D. Padua, ed.), pp. 1810–1812, Boston, MA: Springer US, 2011.

[14] J. Franco and S. Drossopoulou, "Behavioural types for non-uniform memory accesses," *Electronic Proceedings in Theoretical Computer Science*, vol. 203, pp. 109–120, Feb. 2016. arXiv:1602.03599 [cs].

[15] C. Lameter, "NUMA (Non-Uniform Memory Access): An Overview: NUMA becomes more common because memory controllers get close to execution units on microprocessors.," *Queue*, vol. 11, pp. 40–51, July 2013.

[16] B. Barney, "Introduction to Parallel Computing," Sept. 2007. https://www.lrde.epita.fr/~ricou/intro_parallel_comp.pdf.

[17] P. Hijma, S. Heldens, A. Sclocco, B. van Werkhoven, and H. E. Bal, "Optimization Techniques for GPU Programming," *ACM Computing Surveys*, vol. 55, pp. 239:1–239:81, Mar. 2023.

[18] G. Pratx and L. Xing, "GPU computing in medical physics: A review," *Medical Physics*, vol. 38, no. 5, pp. 2685–2697, 2011. https://doi.org/10.1118/1.3578605.

[19] J. E. Stone, D. J. Hardy, I. S. Ufimtsev, and K. Schulten, "GPU-accelerated molecular modeling coming of age," *Journal of Molecular Graphics and Modelling*, vol. 29, pp. 116–125, Sept. 2010.

[20] X. Li, "Scalability: strong and weak scaling – PDC Blog," Nov. 2018. https://www.kth.se/blogs/pdc/2018/11/scalability-strong-and-weak-scaling/.

[21] D. Lund, X. He, X. Zhang, and D. Han, "Weak scaling of the parallel immersed-finite-element particle-in-cell (PIFE-PIC) framework with lunar plasma charging simulations," *Computational Particle Mechanics*, vol. 9, pp. 1279–1291, Nov. 2022.

[22] A. Grama, A. Gupta, G. Karypis, and V. Kumar, *Introduction to Parallel Computing*. Harlow, England ; New York: Addison-Wesley, 2 ed., Feb. 2003.

[23] Y. Asahi, T. Padioleau, G. Latu, J. Bigot, V. Grandgirard, and K. Obrejan, "Performance portable Vlasov code with C++ parallel algorithm," in *2022 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*, pp. 68–80, Nov. 2022. ISSN: 2831-3909.

[24] W.-C. Lin, T. Deakin, and S. McIntosh-Smith, "Evaluating ISO C++ Parallel Algorithms on Heterogeneous HPC Systems," in *2022 IEEE/ACM International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*, (Dallas, TX, USA), pp. 36–47, IEEE, Nov. 2022.

[25] D. Olsen, G. Lopez, and B. Adelstein Lelbach, "Accelerating Standard C++ with GPUs Using stdpar," Aug. 2020. https://developer.nvidia.com/blog/accelerating-standard-c-with-gpus-using-stdpar/.

[26] D. Griebler, J. Loff, G. Mencagli, M. Danelutto, and L. G. Fernandes, "Efficient NAS Benchmark Kernels with C++ Parallel Programming," in *2018 26th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP)*, (Cambridge), pp. 733–740, IEEE, Mar. 2018.

[27] NASA, "NAS Parallel Benchmarks."

[28] M. Drocco, V. G. Castellana, and M. Minutoli, "Practical Distributed Programming in C++," in *Proceedings of the 29th International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '20, (New York, NY, USA), pp. 35–39, Association for Computing Machinery, June 2020.

[29] N. Al Awar, S. Zhu, G. Biros, and M. Gligoric, "A performance portability framework for Python," in *Proceedings of the ACM International Conference on Supercomputing*, ICS '21, (New York, NY, USA), pp. 467–478, Association for Computing Machinery, June 2021.

[30] W.-C. Lin and S. McIntosh-Smith, "Comparing Julia to Performance Portable Parallel Programming Models for HPC," in *2021 International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*, pp. 94–105, Nov. 2021.

[31] J. M. R. Teichgraber, "Julia: A competitive high-level choice for performance portability in HPC?," *Seminar (Performance) Portable Programming of HPC Applications*, June 2022.

[32] S. J. Pennycook and J. D. Sewall, “Revisiting a Metric for Performance Portability,” in *2021 International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*, (St. Louis, MO, USA), pp. 1–9, IEEE, Nov. 2021.

[33] T. Deakin, S. McIntosh-Smith, J. Price, A. Poenaru, P. Atkinson, C. Popa, and J. Salmon, “Performance Portability across Diverse Computer Architectures,” in *2019 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*, (Denver, CO, USA), pp. 1–13, IEEE, Nov. 2019.

[34] S. Christgau and T. Steinke, “Porting a Legacy CUDA Stencil Code to oneAPI,” in *2020 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pp. 359–367, May 2020.

[35] T. Zhao, S. Williams, M. Hall, and H. Johansen, “Delivering Performance-Portable Stencil Computations on CPUs and GPUs Using Bricks,” in *2018 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*, pp. 59–70, Nov. 2018.

[36] T. Deakin and S. McIntosh-Smith, “Evaluating the performance of HPC-style SYCL applications,” in *Proceedings of the International Workshop on OpenCL*, IWOCL ’20, (New York, NY, USA), pp. 1–11, Association for Computing Machinery, Apr. 2020.

[37] Intel, “Explore SYCL with Samples from Intel,” Feb. 2023.

[38] A. Alpay and V. Heuveline, “SYCL beyond OpenCL: The architecture, current state and future direction of hipSYCL,” in *Proceedings of the International Workshop on OpenCL*, (Munich Germany), pp. 1–1, ACM, Apr. 2020.

[39] OpenSYCL. https://github.com/OpenSYCL/OpenSYCL.

[40] Z. Jin, “Improving the performance of medical imaging applications using SYCL,” Tech. Rep. ANL/ALCF-19/4-Rev.1, Argonne National Lab. (ANL), Argonne, IL (United States), May 2020.

[41] M. Mrozek, B. Ashbaugh, and J. Brodman, “Taking Memory Management to the Next Level: Unified Shared Memory in Action,” in *Proceedings of the International Workshop on OpenCL*, IWOCL ’20, (New York, NY, USA), pp. 1–3, Association for Computing Machinery, Apr. 2020.

[42] D. Sunderland, B. Peterson, J. Schmidt, A. Humphrey, J. Thornock, and M. Berzins, “An Overview of Performance Portability in the Uintah Runtime System through the Use of Kokkos,” in *2016 Second International Workshop on Extreme Scale Programming Models and Middlewar (ESPM2)*, pp. 44–47, Nov. 2016.

[43] C. R. Trott, D. Lebrun-Grandié, D. Arndt, J. Ciesko, V. Dang, N. Ellingwood, R. Gayatri, E. Harvey, D. S. Hollman, D. Ibanez, N. Liber, J. Madsen, J. Miles,

D. Poliakoff, A. Powell, S. Rajamanickam, M. Simberg, D. Sunderland, B. Turcksin, and J. Wilke, “Kokkos 3: Programming Model Extensions for the Exascale Era,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, pp. 805–817, Apr. 2022.

[44] J. R. Hammond, M. Kinsner, and J. Brodman, “A comparative analysis of Kokkos and SYCL as heterogeneous, parallel programming models for C++ applications,” in *Proceedings of the International Workshop on OpenCL*, IWOCL’19, (New York, NY, USA), pp. 1–2, Association for Computing Machinery, May 2019.

[45] R. D. Falgout, R. Li, B. Sjögreen, L. Wang, and U. M. Yang, “Porting hypre to heterogeneous computer architectures: Strategies and experiences,” *Parallel Computing*, vol. 108, p. 102840, Dec. 2021.

[46] C. Wood, G. Georgakoudis, D. Beckingsale, D. Poliakoff, A. Gimenez, K. Huck, A. Malony, and T. Gamblin, “Artemis: Automatic Runtime Tuning of Parallel Execution Parameters Using Machine Learning,” in *High Performance Computing* (B. L. Chamberlain, A.-L. Varbanescu, H. Ltaief, and P. Luszczek, eds.), Lecture Notes in Computer Science, (Cham), pp. 453–472, Springer International Publishing, 2021.

[47] W. Killian, T. Scogland, A. Kunen, and J. Cavazos, “The Design and Implementation of OpenMP 4.5 and OpenACC Backends for the RAJA C++ Performance Portability Layer,” in *Accelerator Programming Using Directives* (S. Chandrasekaran and G. Juckeland, eds.), Lecture Notes in Computer Science, (Cham), pp. 63–82, Springer International Publishing, 2018.

[48] “Algorithms library - cppreference.com.” https://en.cppreference.com/w/cpp/algorithm.

[49] S. Sengupta, M. Harris, and M. Garland, “Efficient Parallel Scan Algorithms for GPUs,” Tech. Rep. nvr-2008-003, NVIDIA, 2008.

[50] T. Brandvik and G. Pullan, “Chapter 14 - Large-Scale Gas Turbine Simulations on GPU Clusters,” in *GPU Computing Gems Jade Edition* (W.-m. W. Hwu, ed.), Applications of GPU Computing Series, pp. 157–171, Boston: Morgan Kaufmann, Jan. 2012.

[51] K. Nguyen, T. Nguyen, and Q.-S. Phan, “Analyzing the CMake build system,” in *Proceedings of the 44th International Conference on Software Engineering: Software Engineering in Practice*, (Pittsburgh Pennsylvania), pp. 27–28, ACM, May 2022.

[52] “CppCon 2015: Chandler Carruth "Tuning C++: Benchmarks, and CPUs, and Compilers! Oh My!",” Sept. 2015. https://www.youtube.com/watch?v=nXaxk27zwlk [Accessed: Apr. 25, 2014].

[53] D. Terpstra, H. Jagode, H. You, and J. Dongarra, “Collecting Performance Data with PAPI-C,” in *Tools for High Performance Computing 2009* (M. S. Müller, M. M.

Resch, A. Schulz, and W. E. Nagel, eds.), (Berlin, Heidelberg), pp. 157–173, Springer, 2010.

[54] P. Mucci, S. Moore, C. Deane, and G. Ho, "PAPI: A Portable Interface to Hardware Performance Counters," in *Proceedings of the Department of Defense HPCMP users group conference*, Jan. 1999.

[55] W. McKinney, "pandas: a foundational Python library for data analysis and statistics," *Python for high performance and scientific computing*, vol. 14, no. 9, pp. 1–9, 2011.

[56] J. D. Hunter, "Matplotlib: A 2D Graphics Environment," *Computing in Science & Engineering*, vol. 9, pp. 90–95, May 2007.

[57] A. B. Yoo, M. A. Jette, and M. Grondona, "SLURM: Simple Linux Utility for Resource Management," in *Job Scheduling Strategies for Parallel Processing* (D. Feitelson, L. Rudolph, and U. Schwiegelshohn, eds.), Lecture Notes in Computer Science, (Berlin, Heidelberg), pp. 44–60, Springer, 2003.

[58] "gcc-mirror/gcc." https://github.com/gcc-mirror/gcc.

[59] D. Olsen, "P2408R4: Ranges iterators as inputs to non-Ranges algorithms," Nov. 2021. https://wg21.link/p2408r4.

[60] F. Gaud, B. Lepers, J. Funston, M. Dashti, A. Fedorova, V. Quéma, R. Lachaize, and M. Roth, "Challenges of memory management on modern NUMA systems," *Communications of the ACM*, vol. 58, pp. 59–66, Nov. 2015.

[61] B. Goglin and N. Furmento, "Enabling high-performance memory migration for multithreaded applications on LINUX," in *2009 IEEE International Symposium on Parallel & Distributed Processing*, pp. 1–9, May 2009. ISSN: 1530-2075.

[62] ACM, "Frequently Asked Questions." Accessed 5 June 2023.

[63] D. Krupitza, "Example ChatGPT Prompts for pSTL-Bench," June 2023. https://doi.org/10.5281/zenodo.8004781.

APPENDIX A

Machine Info

Hydra

```
Architecture:                    x86_64
CPU op-mode(s):                  32-bit, 64-bit
Byte Order:                      Little Endian
Address sizes:                   46 bits physical, 48 bits virtual
CPU(s):                          32
On-line CPU(s) list:             0-31
Thread(s) per core:              1
Core(s) per socket:              16
Socket(s):                       2
NUMA node(s):                    2
Vendor ID:                       GenuineIntel
CPU family:                      6
Model:                           85
Model name:                      Intel(R) Xeon(R) Gold 6130F CPU @ 2.10GHz
Stepping:                        4
CPU MHz:                         1661.964
CPU max MHz:                     2100.0000
CPU min MHz:                     1000.0000
BogoMIPS:                        4200.00
L1d cache:                       1 MiB
L1i cache:                       1 MiB
L2 cache:                        32 MiB
L3 cache:                        44 MiB
NUMA node0 CPU(s):               0,2,4,6,8,10,12,14,16,18,20,22,24,26,28,30
NUMA node1 CPU(s):               1,3,5,7,9,11,13,15,17,19,21,23,25,27,29,31
```

Nebula

```
Architecture:        x86_64
CPU op-mode(s):      32-bit, 64-bit
Byte Order:          Little Endian
Address sizes:       43 bits physical, 48 bits virtual
CPU(s):              64
On-line CPU(s) list: 0-63
Thread(s) per core:  1
Core(s) per socket:  32
Socket(s):           2
NUMA node(s):        8
Vendor ID:           AuthenticAMD
CPU family:          23
Model:               1
Model name:          AMD EPYC 7551 32-Core Processor
Stepping:            2
CPU MHz:             2406.833
CPU max MHz:         2000.0000
CPU min MHz:         1200.0000
BogoMIPS:            3992.24
Virtualization:      AMD-V
L1d cache:           32K
L1i cache:           64K
L2 cache:            512K
L3 cache:            8192K
NUMA node0 CPU(s):   0,8,16,24,32,40,48,56
NUMA node1 CPU(s):   2,10,18,26,34,42,50,58
NUMA node2 CPU(s):   4,12,20,28,36,44,52,60
NUMA node3 CPU(s):   6,14,22,30,38,46,54,62
NUMA node4 CPU(s):   1,9,17,25,33,41,49,57
NUMA node5 CPU(s):   3,11,19,27,35,43,51,59
NUMA node6 CPU(s):   5,13,21,29,37,45,53,61
NUMA node7 CPU(s):   7,15,23,31,39,47,55,63
```

APPENDIX B

Code Snippet

Listing B.1: Example Data Collector configuration file.

```
{
  "output_dir": "./output",
  "benchmark_repetitions": 10,
  "cmake_location": "/CORE/",
  "build_artifacts_dir": "./build-artifacts",
  "binary_target": "master_benchmarks",
  "batch_file_location": "./job_files",
  "sbatch": {
    "partition": "q_thesis",
    "time": "15:00"
  },
  "compiler": [
  {
    "name": "GCC_TBB",
    "CXX_COMPILER": "g++",
    "CXX_FLAGS": "-ltbb",
    "build_location": "cmake-build-gcc",
    "description": "Benchmarks with gcc and tbb"
  },
  {
    "name": "NVHPC_Multicore",
    "CXX_COMPILER": "nvc++",
    "CXX_FLAGS": "-stdpar=multicore",
    "build_location": "cmake-build-nvc_multicore",
    "description": "Benchmarks with nvc++ with multicore"
  }
  ],
  "benchmarks": [
  {
    "name": "b1_1_for_each_linear_par_67108864_T1",
    "description": "Sample description 1",
    "type": "THREADS",
    "params": {
      "threads": 1
    },
    "regex_filter": "b1_1_for_each_linear_par/67108864"
  },
  {
    "name": "b1_1_for_each_linear_mandelbrot_seq__Default",
    "description": "Sample description 1",
    "type": "DEFAULT",
    "regex_filter": "b1_1_for_each_linear_mandelbrot_seq/"
  }
  ]
}
```

APPENDIX C

Nebula plots

Heterogeneous Nest Parallelism

Mandelbrot Quadratic

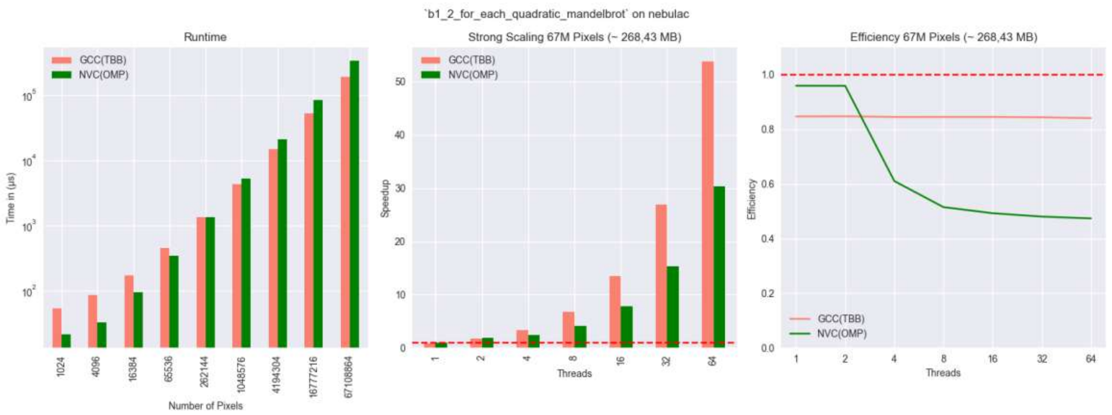


Figure C.1: Runtime & speedup results for “Mandelbrot Quadratic”; Nebula.

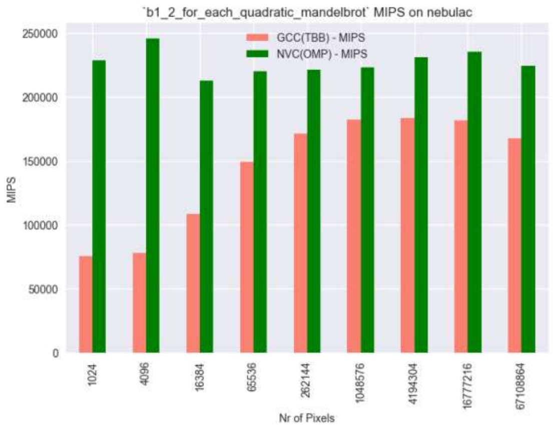


Figure C.2: MIPS for “Mandelbrot Quadratic”; Nebula.

Sequential Fallback

Stable Sorting

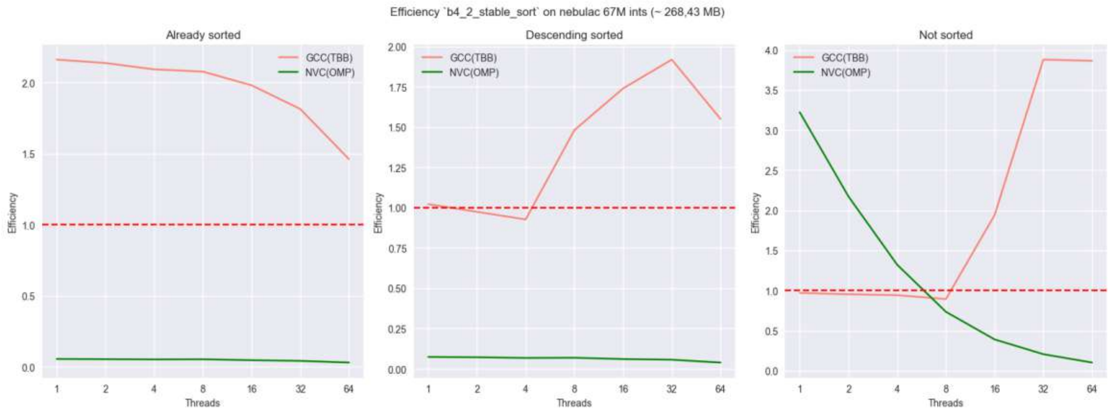


Figure C.3: Efficiency results for “Stable Sorting”; Nebula.

Set Union Logic

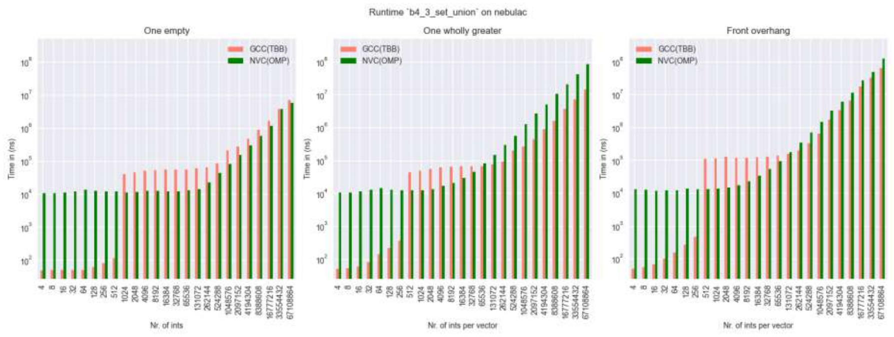


Figure C.4: Runtime results for “Set Union Logic”; Nebula.

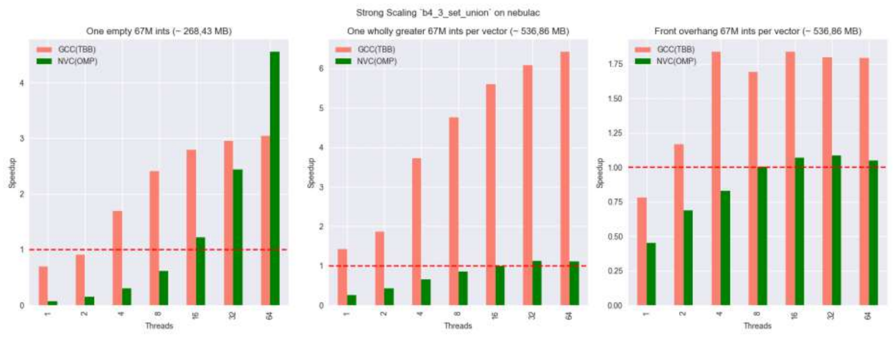


Figure C.5: Strong Scaling results for “Set Union Logic”; Nebula.

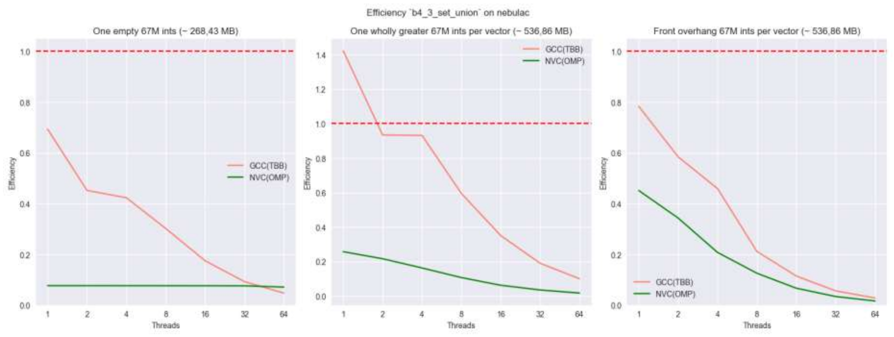


Figure C.6: Efficiency results for “Set Union Logic”; Nebula.

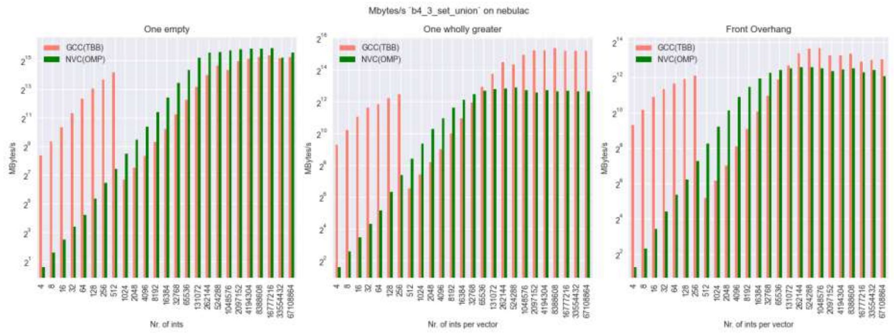


Figure C.7: Throughput results for “Set Union Logic”; Nebula.

Set Difference Logic

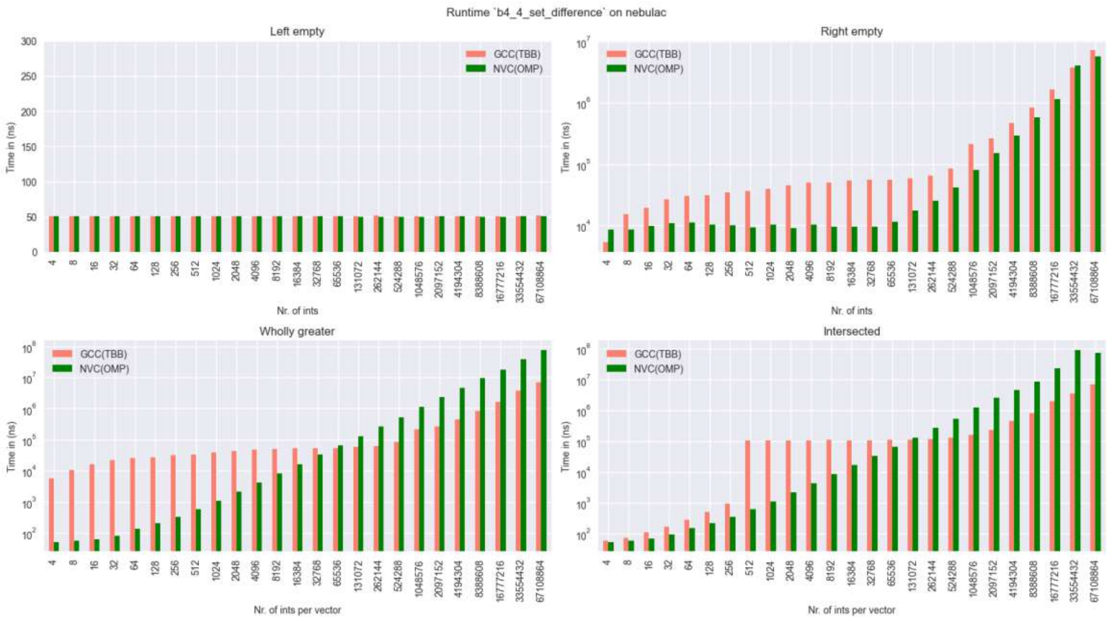


Figure C.8: Runtime results for “Set Difference Logic”; Nebula.

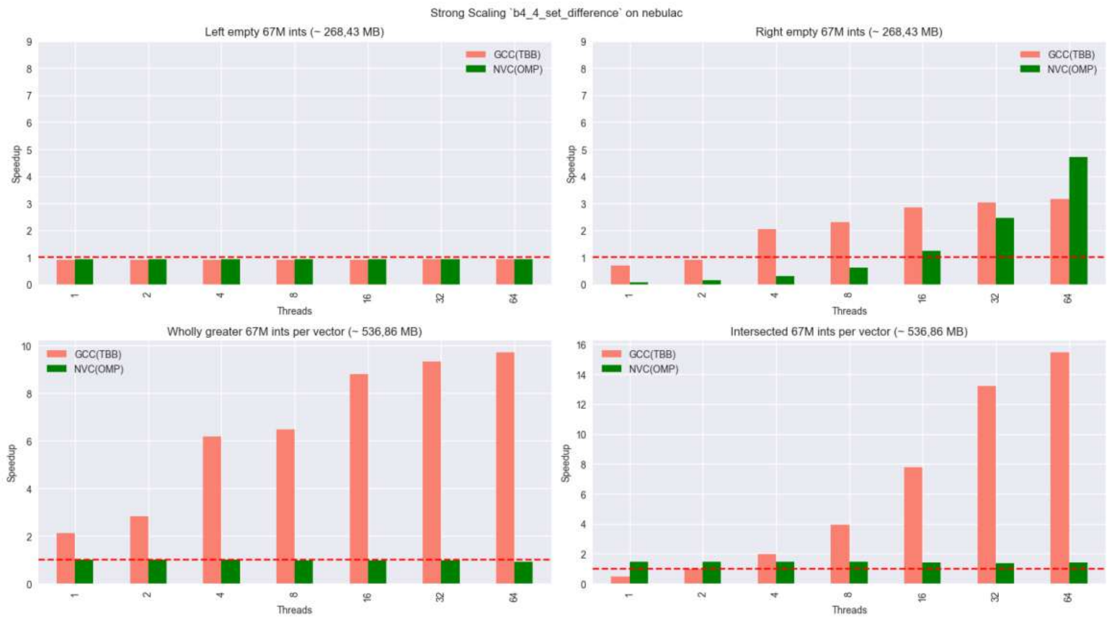


Figure C.9: Strong Scaling results for "Set Difference Logic"; Nebula.

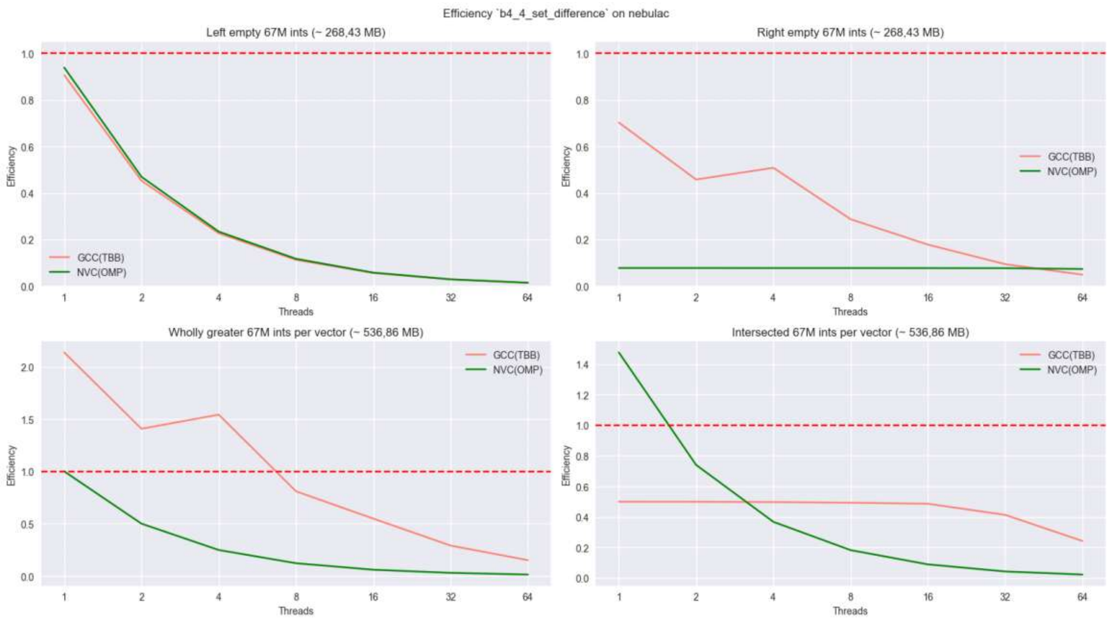


Figure C.10: Efficiency results for "Set Difference Logic"; Nebula.

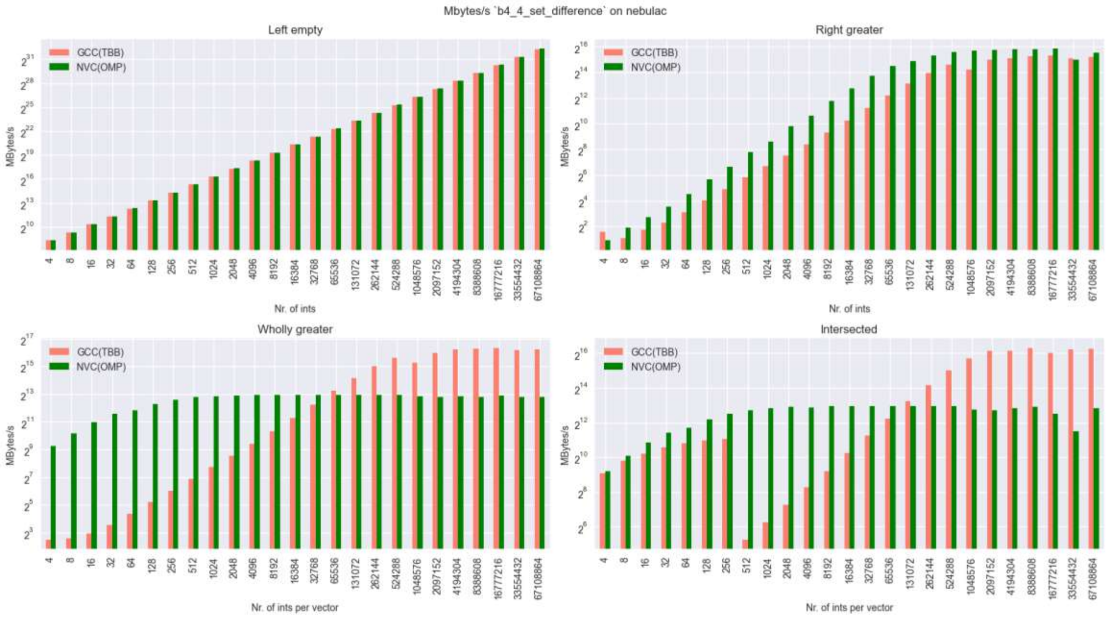


Figure C.11: Throughput results for “Set Difference Logic”; Nebula.

Specialized Parallelism Techniques

Unique Copy

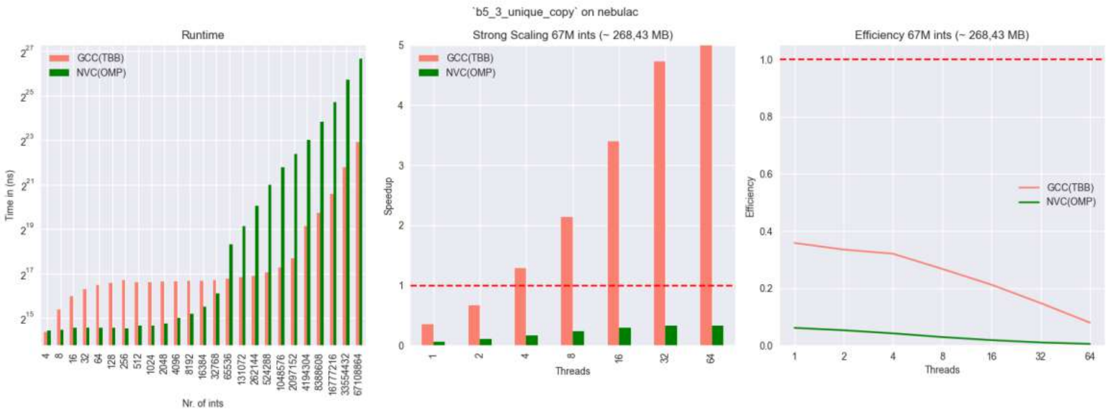


Figure C.12: Runtime & speedup results for “Unique Copy”; Nebula.

Min Max Elements

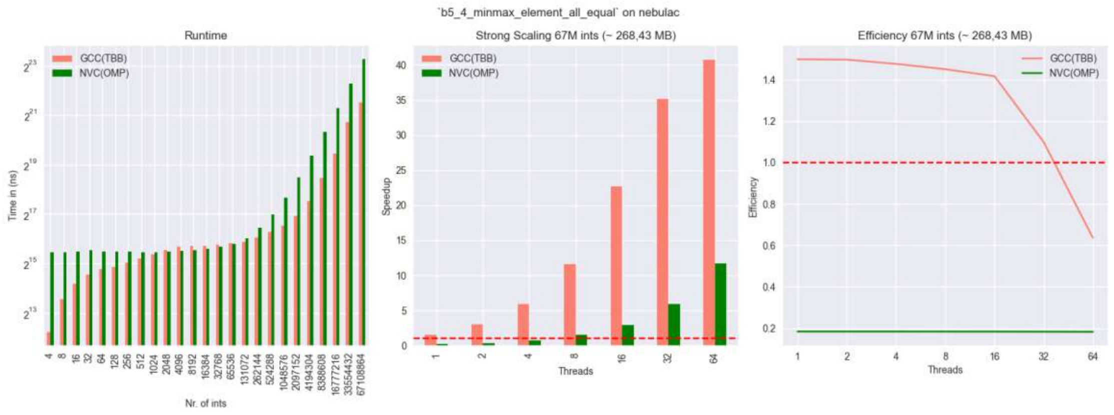


Figure C.13: Runtime & speedup results for "Min Max Elements" all elements equal; Nebula.

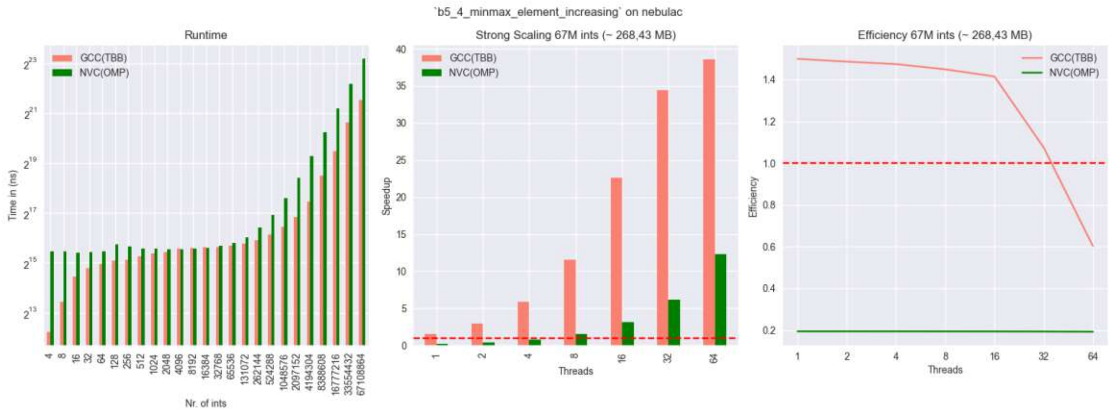


Figure C.14: Runtime & speedup results for "Min Max Elements" increasing elements; Nebula.

Tailored Optimization for Parallel Scans

Exclusive Scan

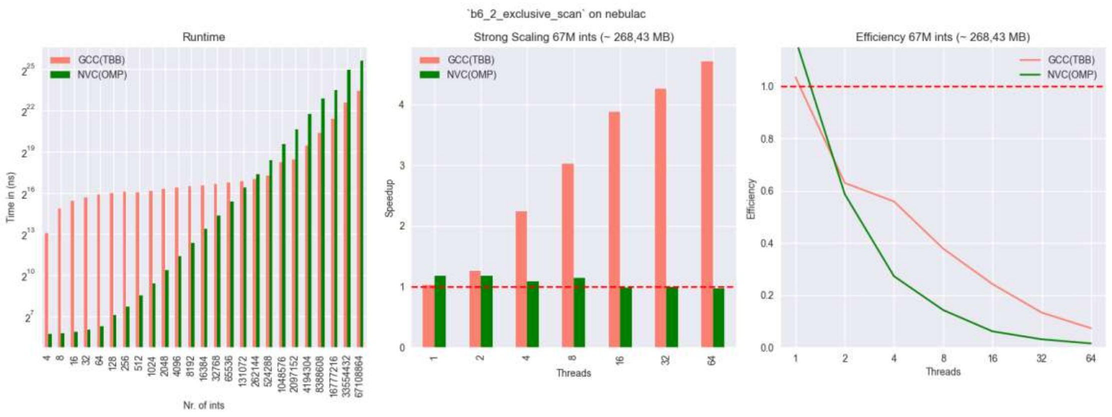


Figure C.15: Runtime & speedup results for “Exclusive Scan”; Nebula.

Specific vs. Custom Implementation

Count_if Logic

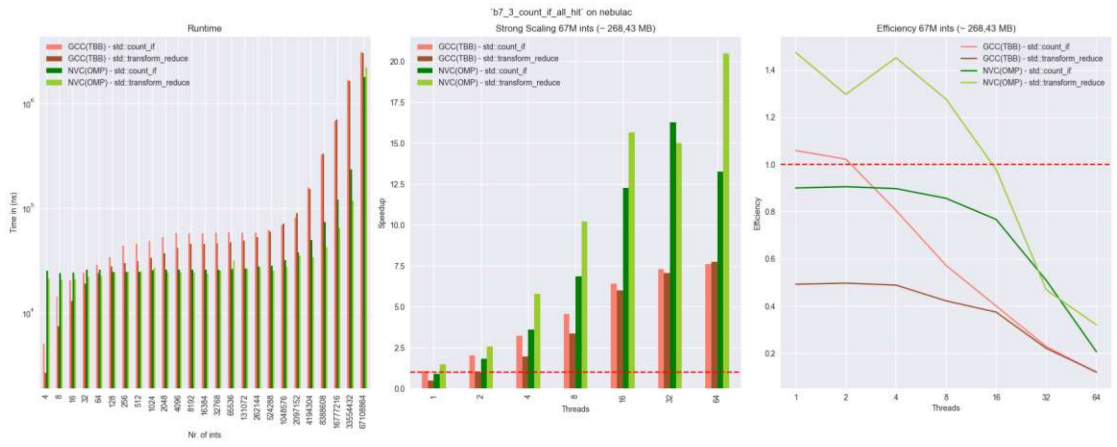


Figure C.16: Runtime & speedup results for “`Count_if` Logic” all hit; Nebula.

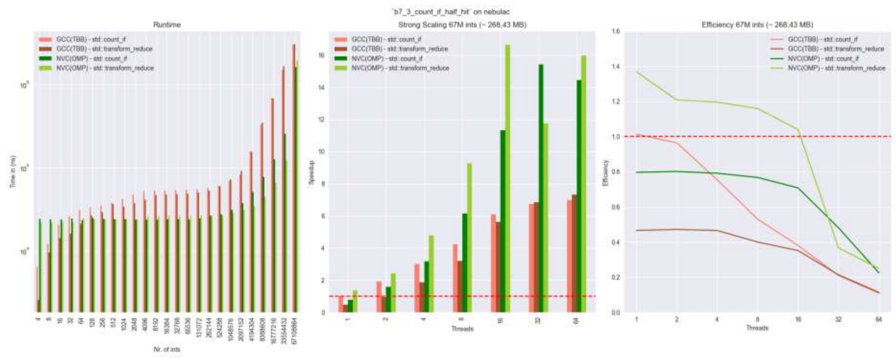


Figure C.17: Runtime & speedup results for “`Count_if` Logic” half hit; Nebula.

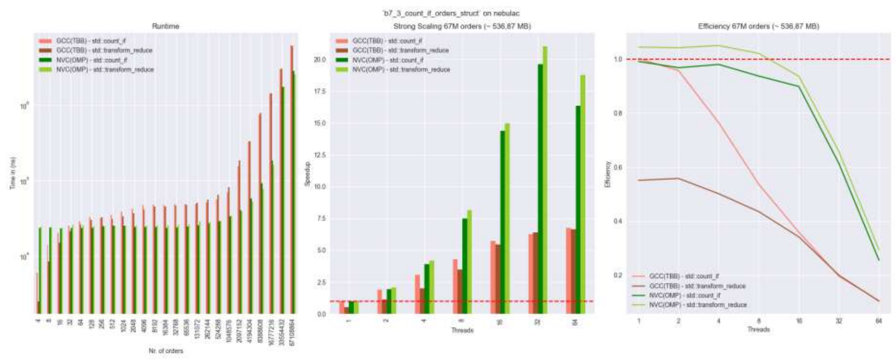
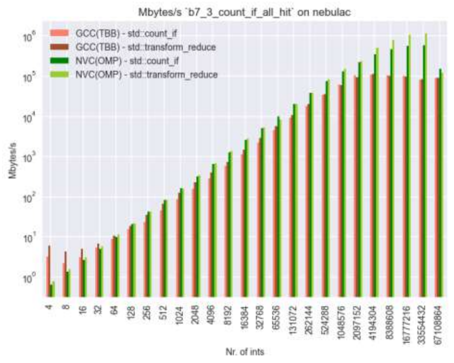
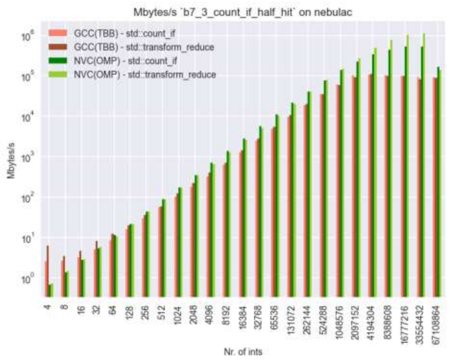


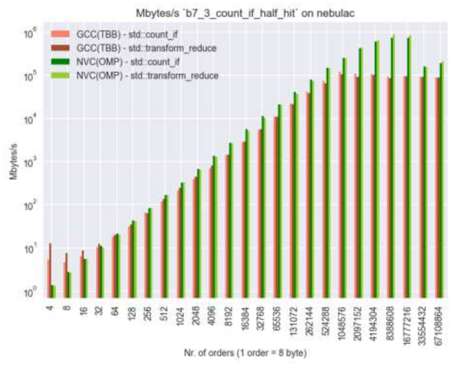
Figure C.18: Runtime & speedup results for “`Count_if` Logic” using structs; Nebula.



(a) All hit



(b) Half hit



(c) Structs

Figure C.19: Throughput results for "`Count_if` Logic"; Nebula.

Scalar Transform Logic

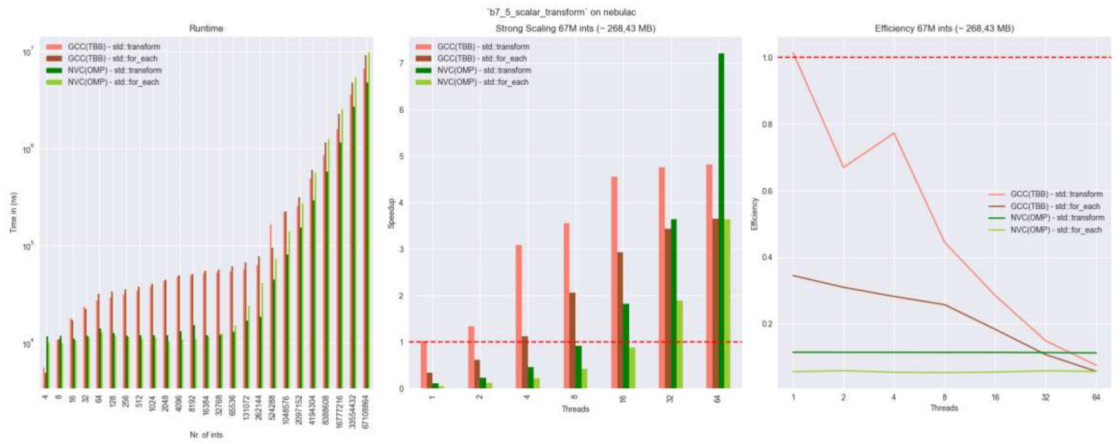


Figure C.20: Runtime & speedup results for "Scalar Transform Logic"; Nebula.

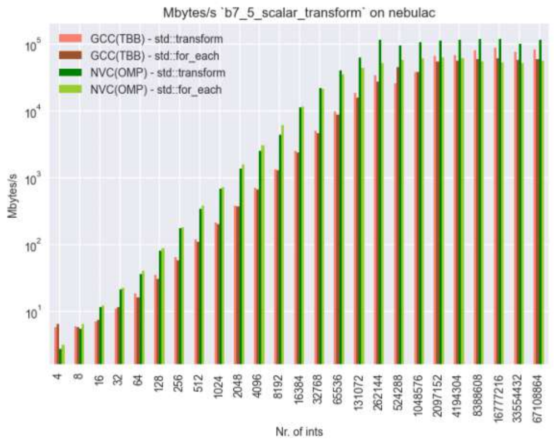


Figure C.21: Throughput results for “Scalar Transform Logic”; Nebula.

Custom Index-based Iterations

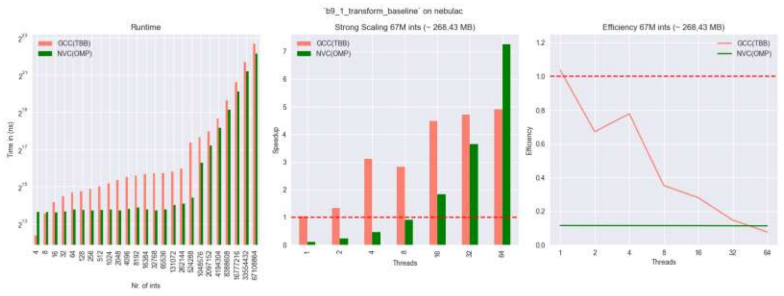


Figure C.22: Runtime & speedup results for “Data-Centric”; Nebula.

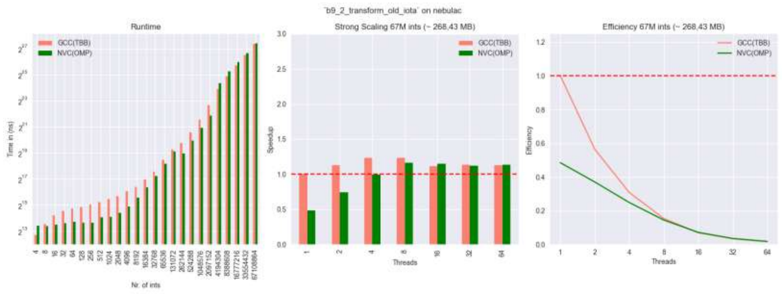


Figure C.23: Runtime & speedup results for “Index-Centric with iota”; Nebula.

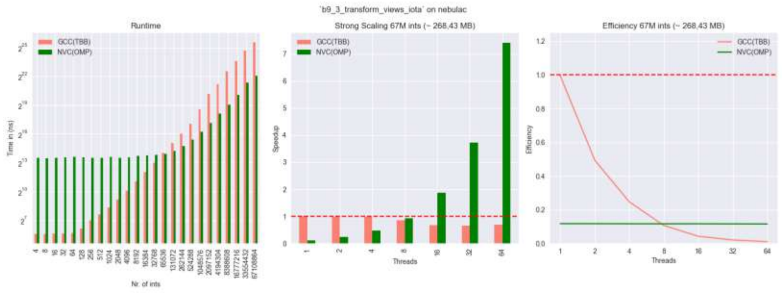


Figure C.24: Runtime & speedup results for “Index-Centric with views::iota”; Nebula.

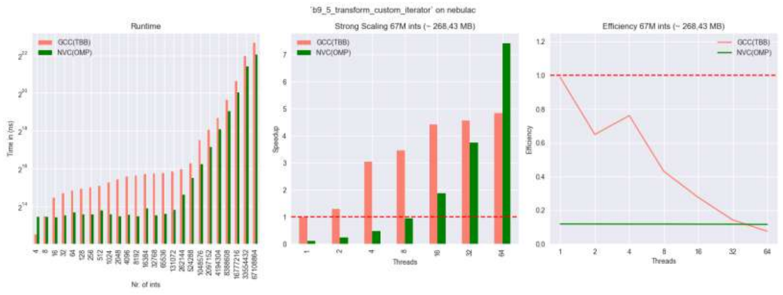


Figure C.25: Runtime & speedup results for “Index-Centric with Custom Iterator”; Nebula.

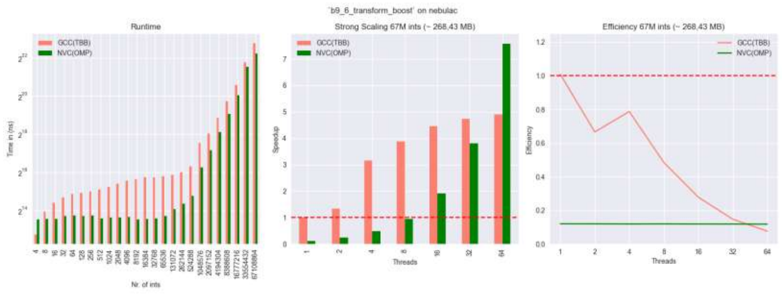


Figure C.26: Runtime & speedup results for “Index-Centric with Boost”; Nebula.

APPENDIX D

Hydra Plots

Homogeneous Nested Parallelism

Homogeneous Linear

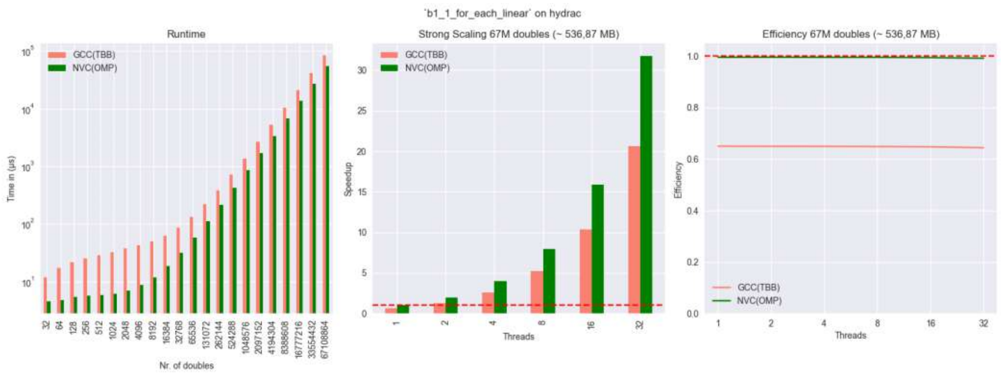


Figure D.1: Runtime & speedup results for “Homogeneous Linear”; Hydra.

Homogeneous Quadratic

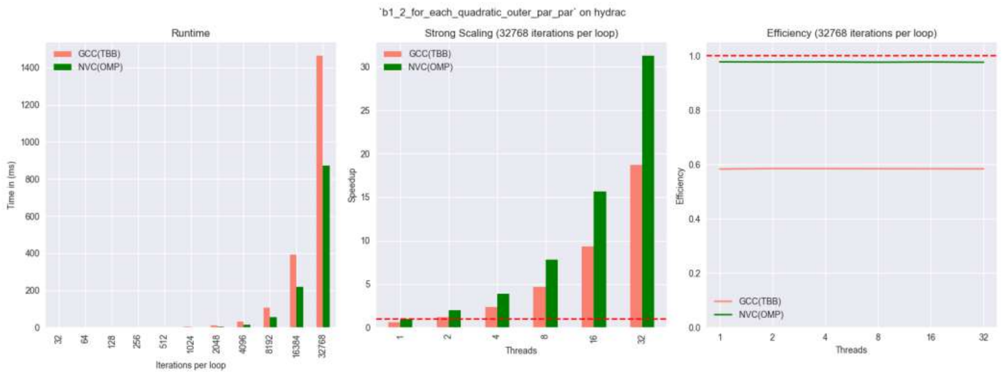


Figure D.2: Runtime & speedup results for “Homogeneous Quadratic”; Hydra.

Homogeneous Exponential

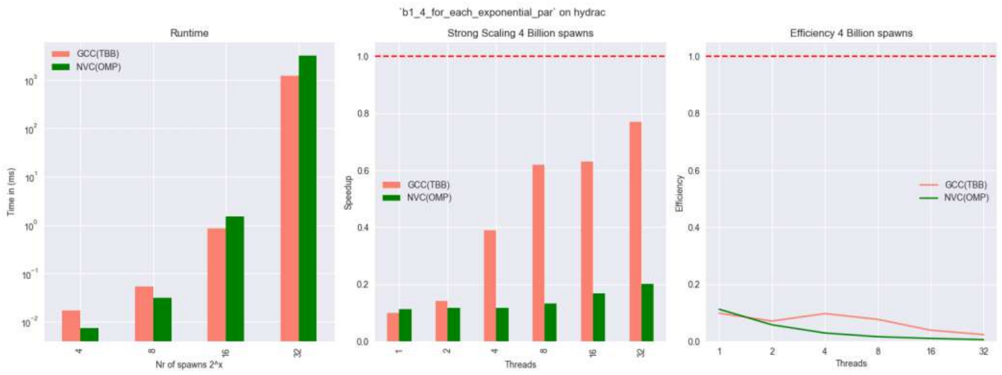


Figure D.3: Runtime & speedup results for “Homogeneous Exponential”; Hydra.

Order of Parallelism

Homogeneous Quadratic

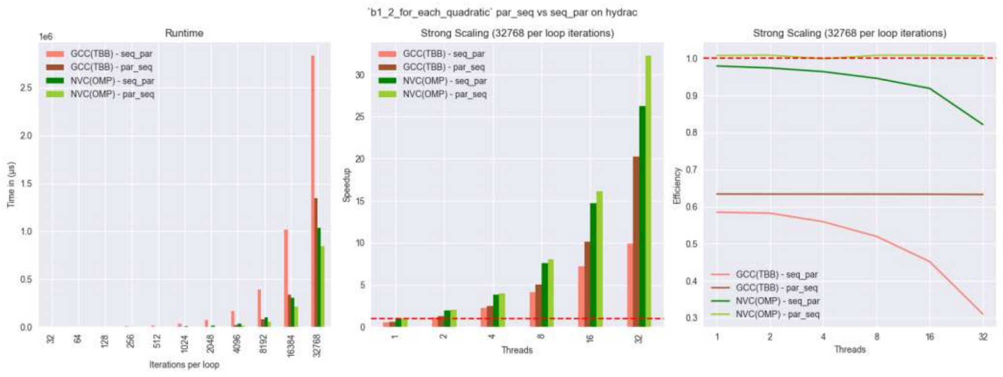


Figure D.4: Runtime & speedup results for Homogeneous Quadratic execution order; Hydra.

Heterogeneous Nest Parallelism

Mandelbrot Linear

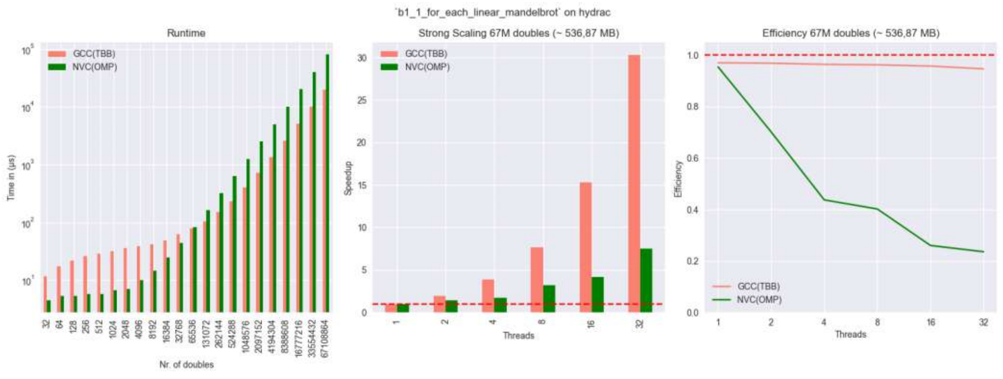


Figure D.5: Runtime & speedup results for “Mandelbrot Linear”; Hydra.

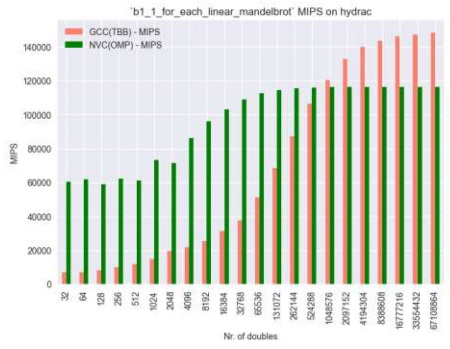


Figure D.6: MIPS for benchmark “Mandelbrot Linear”; Hydra.

Mandelbrot Quadratic

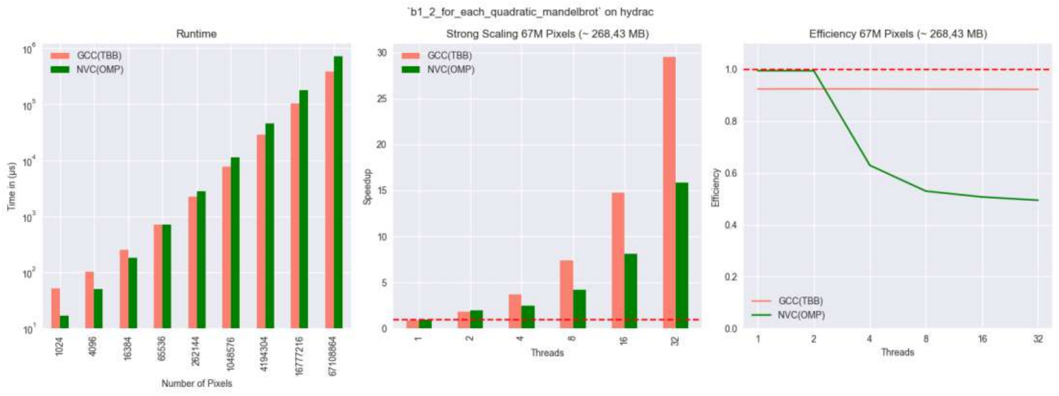


Figure D.7: Runtime & speedup results for “Mandelbrot Quadratic”; Hydra.

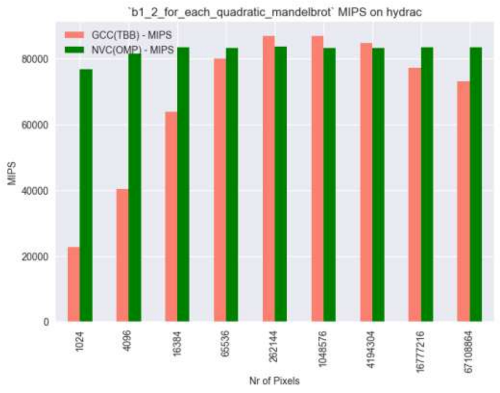
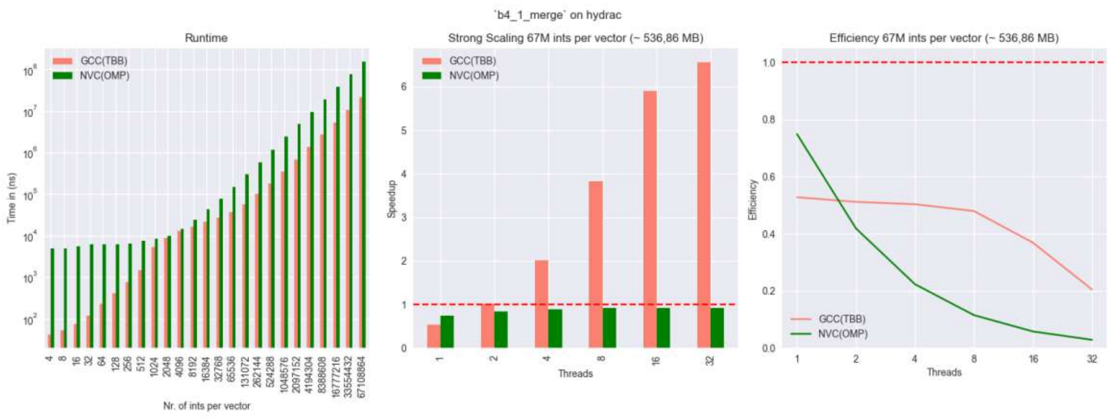


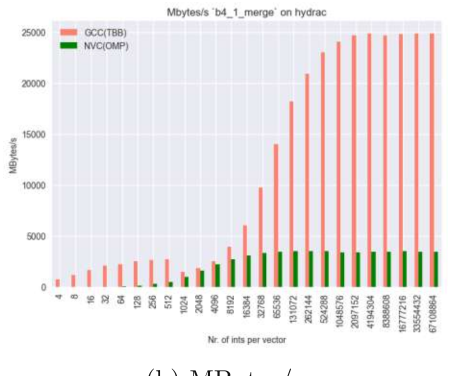
Figure D.8: MIPS for “Mandelbrot Quadratic”; Hydra.

Sequential Fallback

Merge Logic



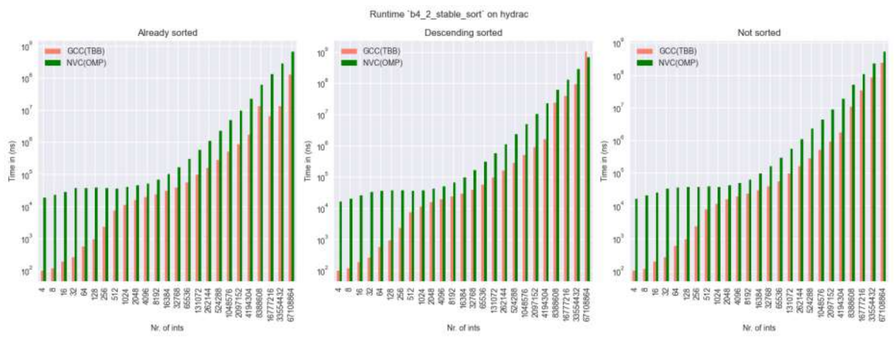
(a) Runtime, Strong Scaling and Efficiency



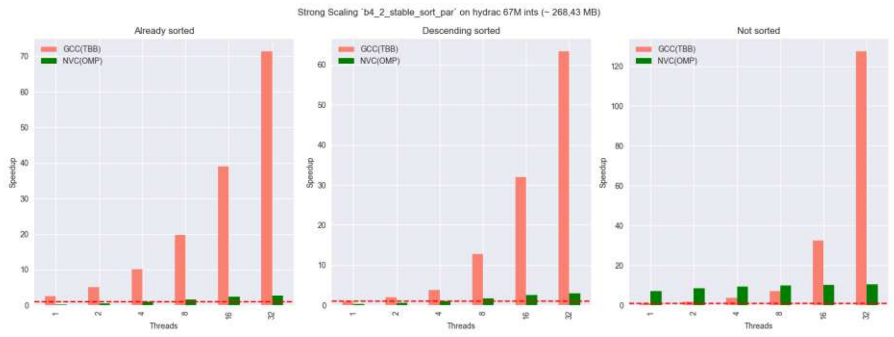
(b) MBytes/s

Figure D.9: Runtime, speedup & throughput results for "Merge Logic"; Hydra.

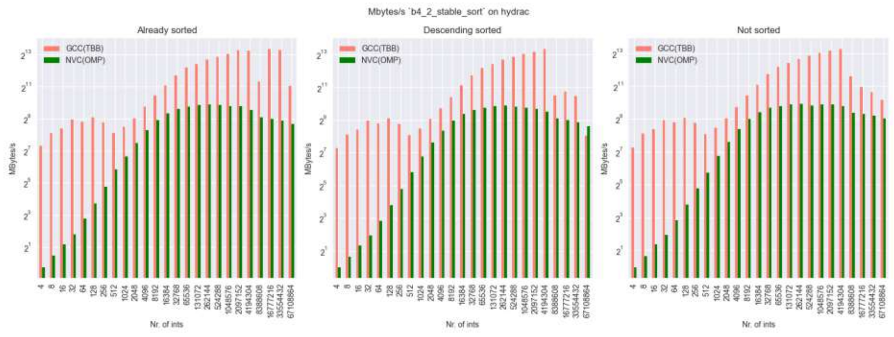
Stable Sorting



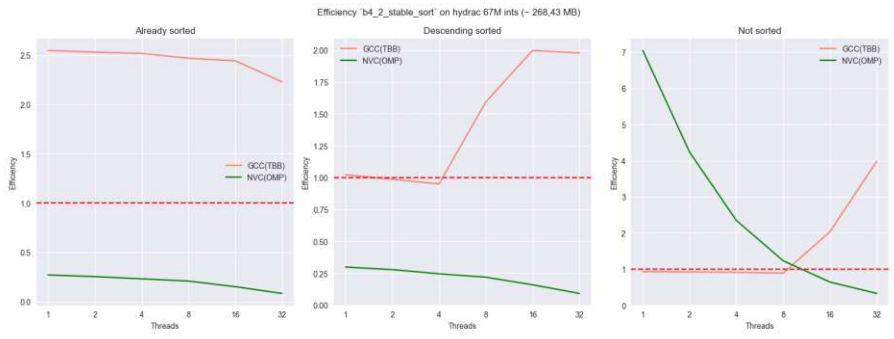
(a) Runtime.



(b) Strong Scaling.



(c) MBytes/s.



(d) Efficiency.

Figure D.10: Runtime, speedup & throughput results for “Stable Sorting”; Hydra.

Set Union Logic

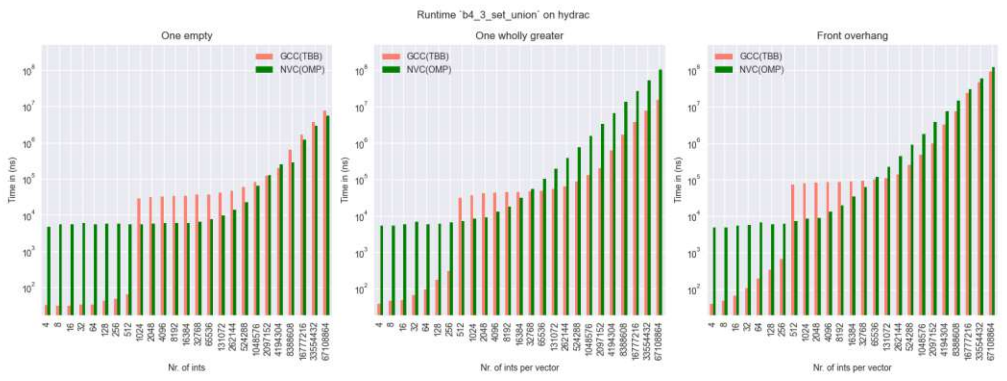


Figure D.11: Runtime results for “Set Union Logic”; Hydra.

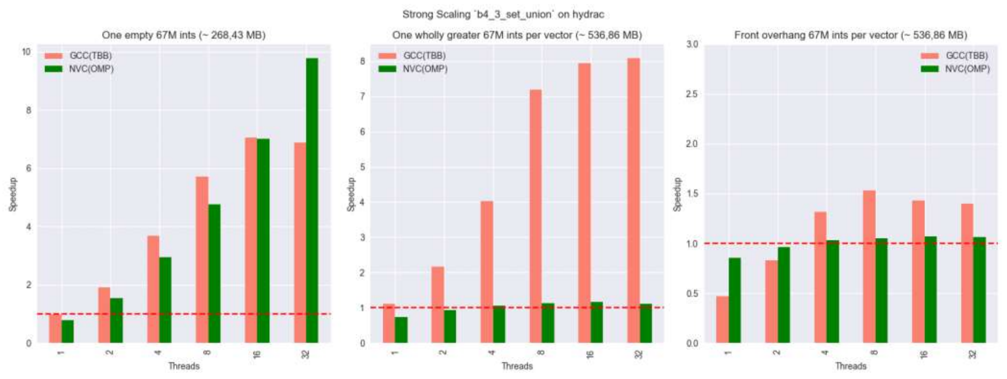


Figure D.12: Strong Scaling results for “Set Union Logic”; Hydra.

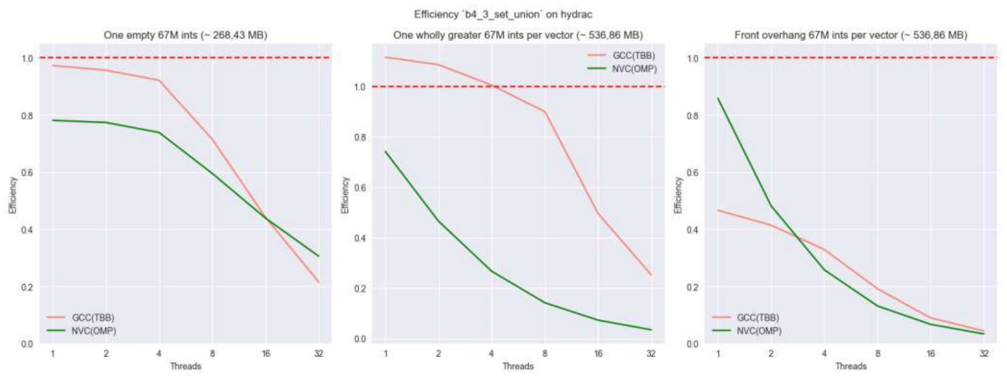


Figure D.13: Efficiency results for “Set Union Logic”; Hydra.

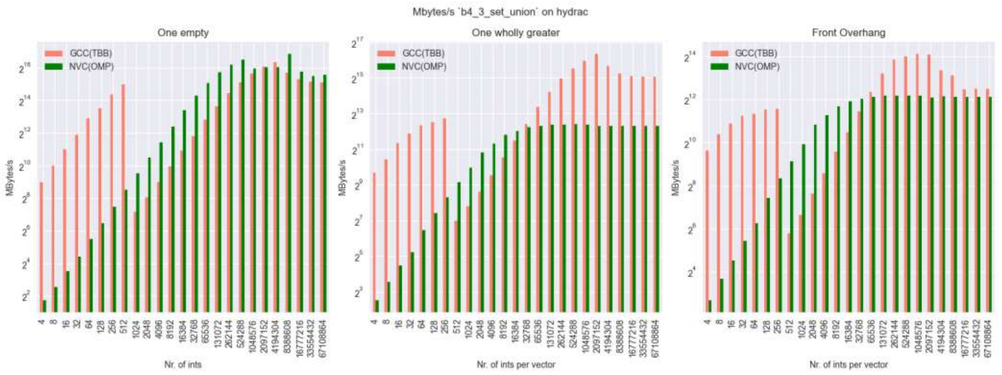


Figure D.14: Throughput results for "Set Union Logic"; Hydra.

Set Difference Logic

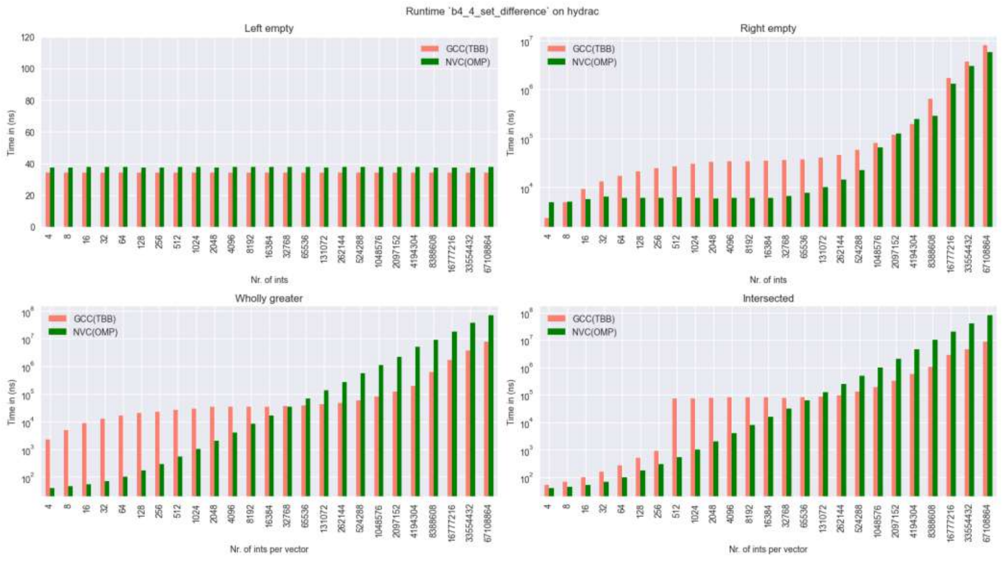


Figure D.15: Runtime results for "Set Difference Logic"; Hydra.

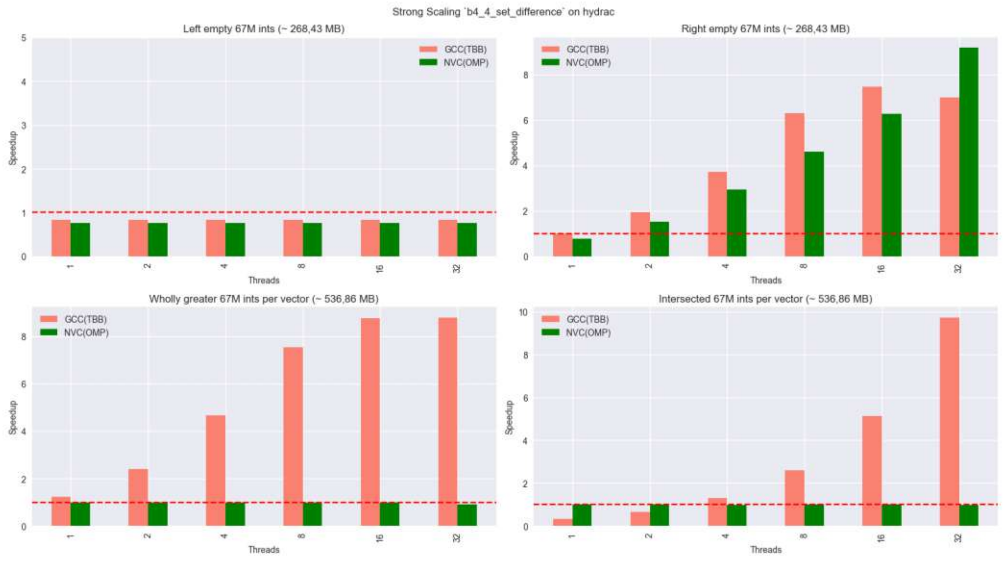


Figure D.16: Strong Scaling results for “Set Difference Logic”; Hydra.

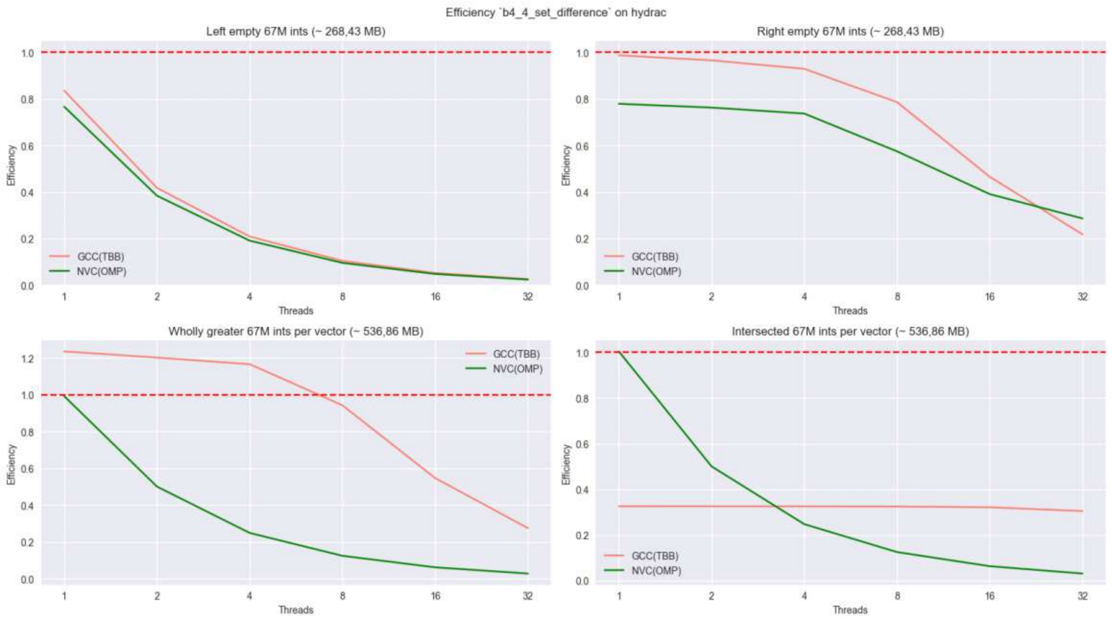


Figure D.17: Efficiency results for “Set Difference Logic”; Hydra.

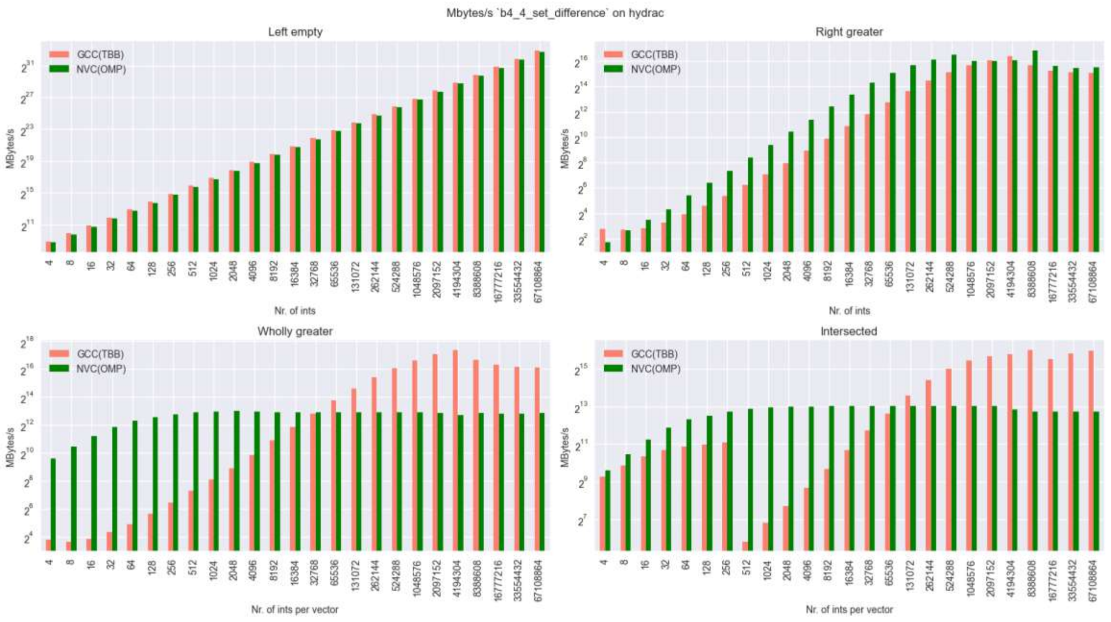
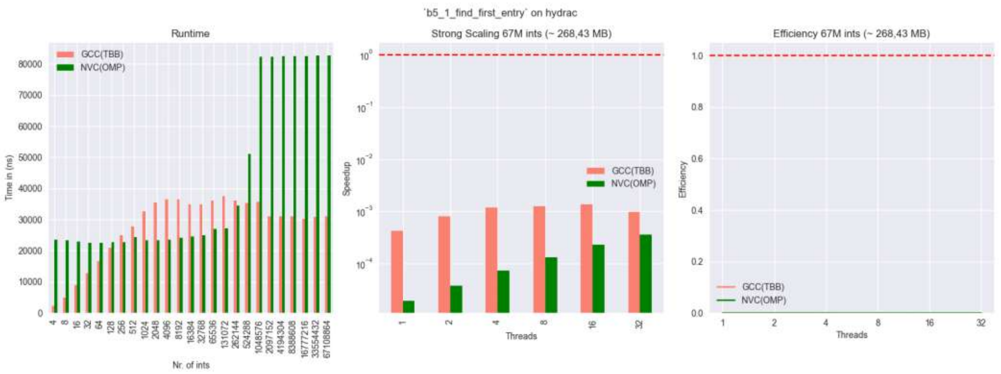


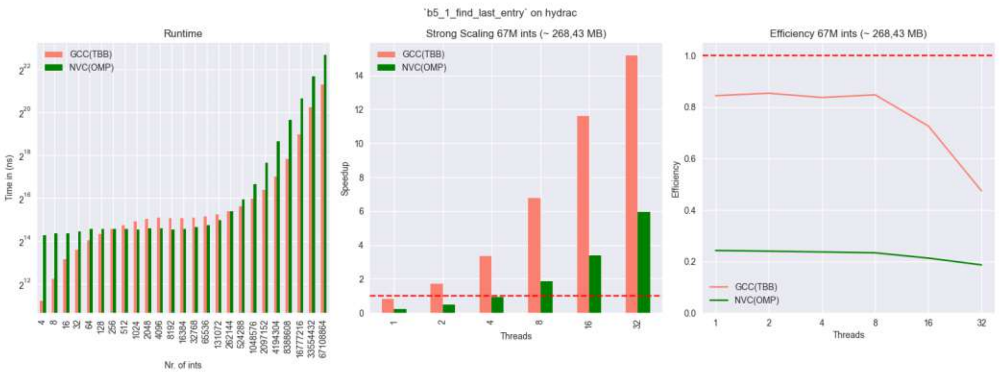
Figure D.18: Throughput results for “Set Difference Logic”; Hydra.

Specialized Parallelism Techniques

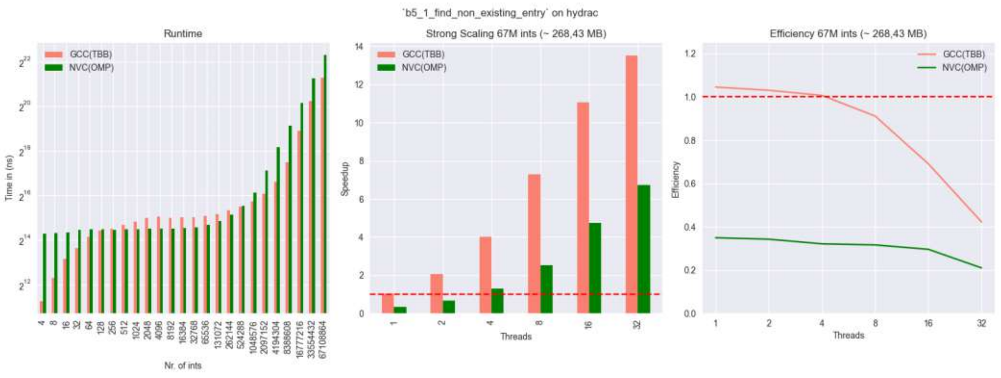
Find Elements



(a) Find first entry



(b) Find last entry



(c) Find non existing entry

Figure D.19: Runtime & speedup results for “Find Elements”; Hydra.

Vector Partition

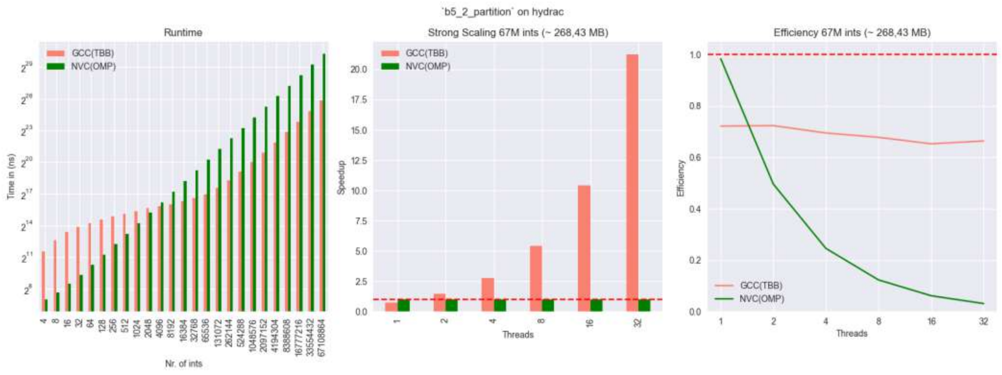


Figure D.20: Runtime & speedup results for “Vector Partition”; Hydra.

Unique Copy

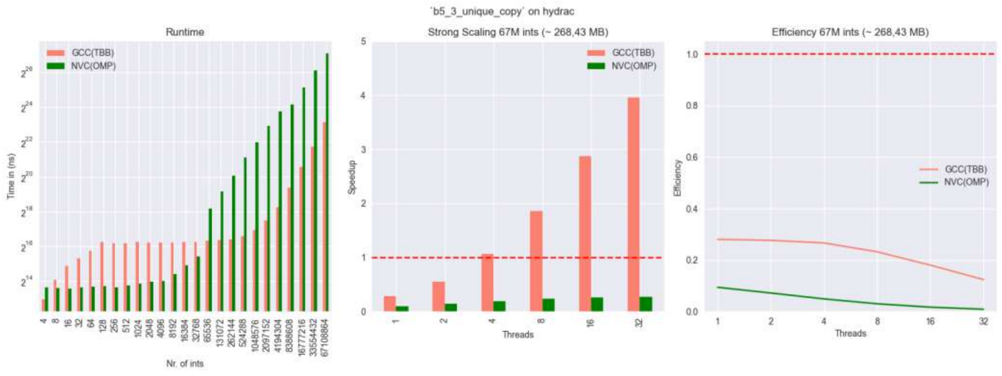


Figure D.21: Runtime & speedup results for “Unique Copy”; Hydra.

Min Max Elements

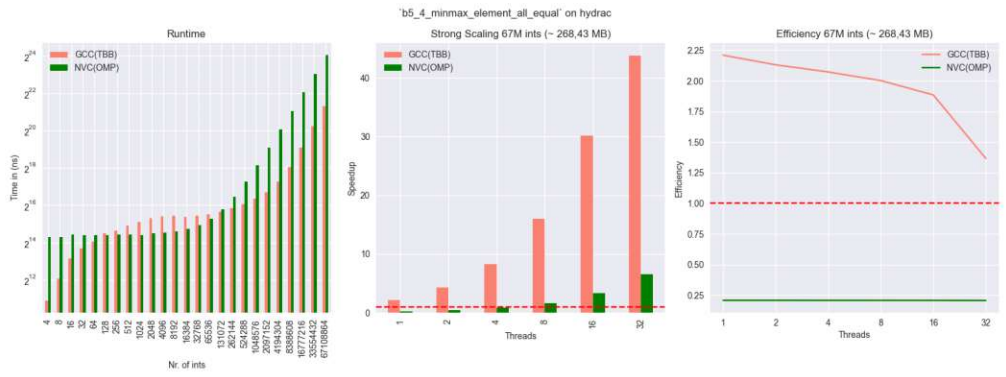


Figure D.22: Runtime & speedup results for “Min Max Elements” all elements equal; Hydra.

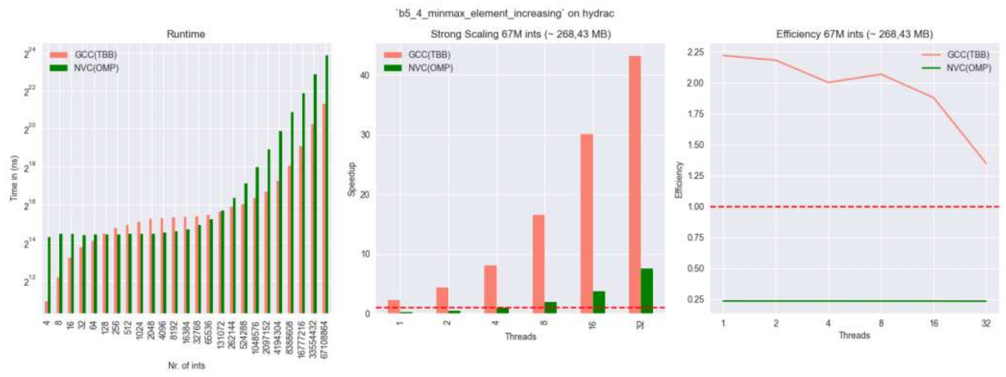


Figure D.23: Runtime & speedup results for “Min Max Elements” increasing elements; Hydra.

Tailored Optimization for Parallel Scans

Inclusive Scan

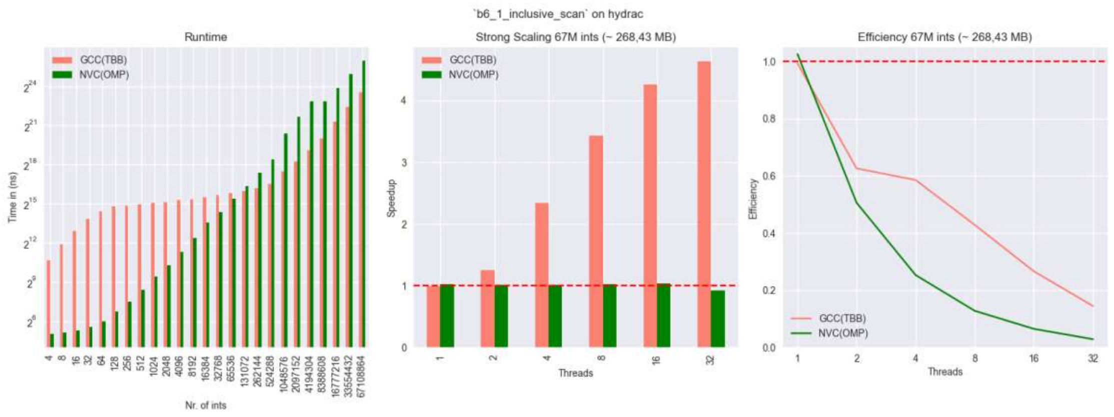


Figure D.24: Runtime & speedup results for "Inclusive Scan"; Hydra.

Exclusive Scan

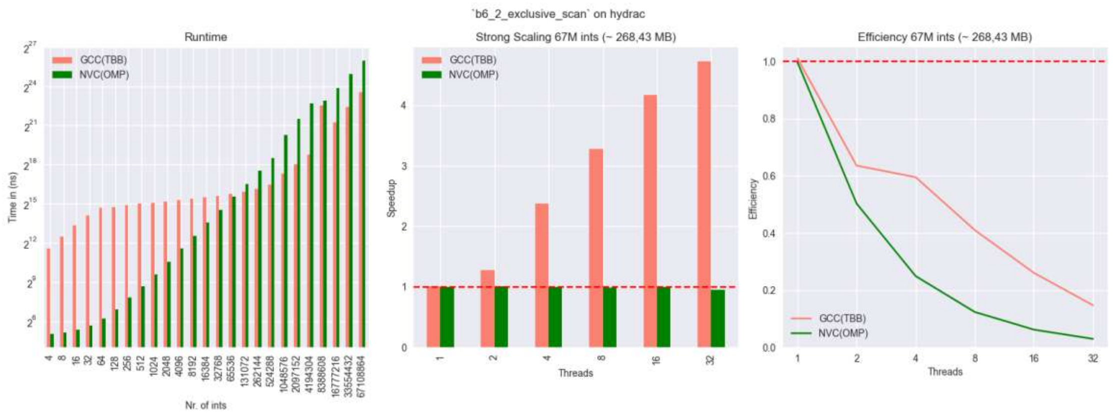


Figure D.25: Runtime & speedup results for "Exclusive Scan"; Hydra.

Specific vs. Custom Implementation

Copy Logic

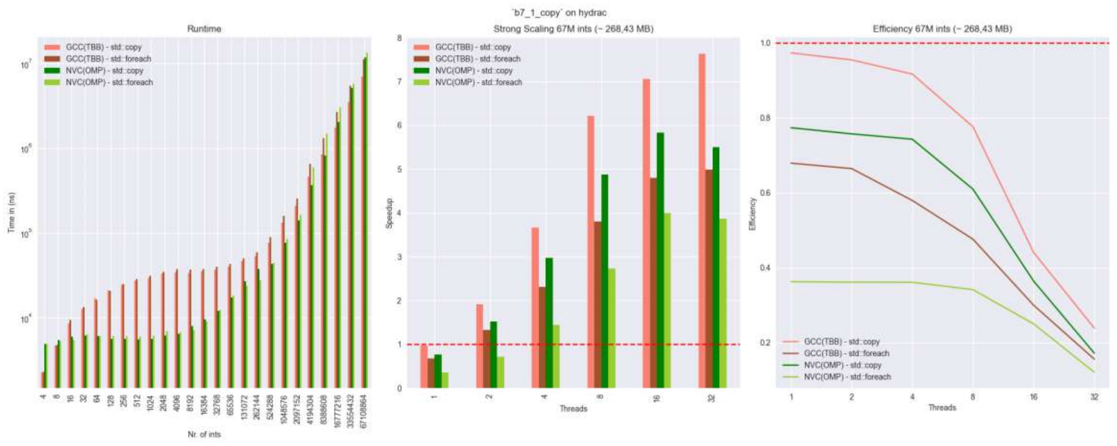


Figure D.26: Runtime & speedup results for “Copy Logic”; Hydra.

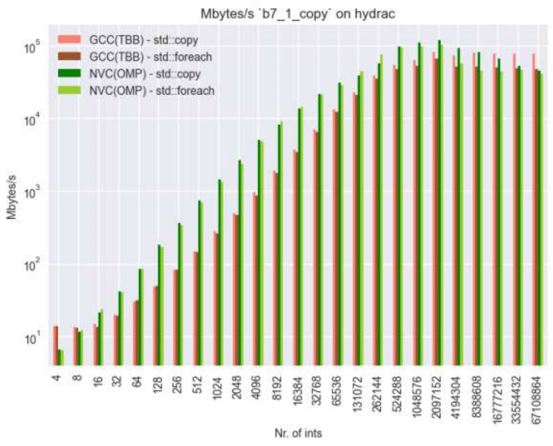
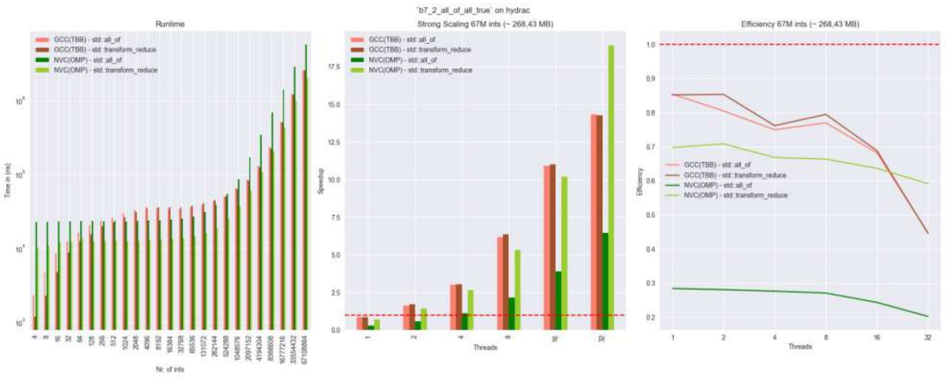
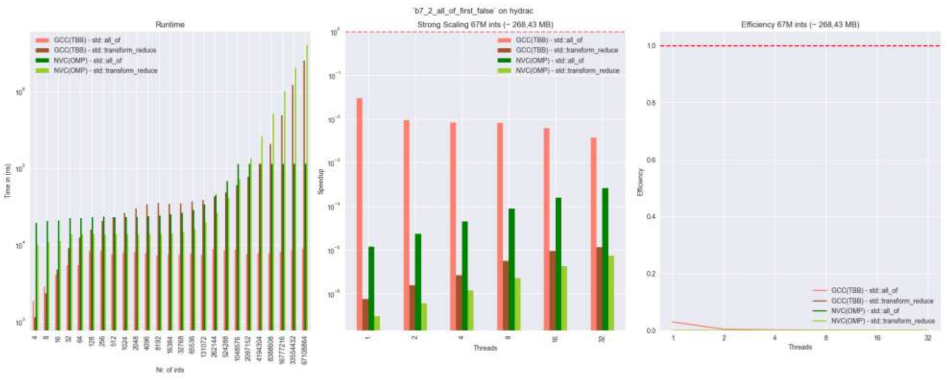


Figure D.27: Throughput results for “Copy Logic”; Hydra.

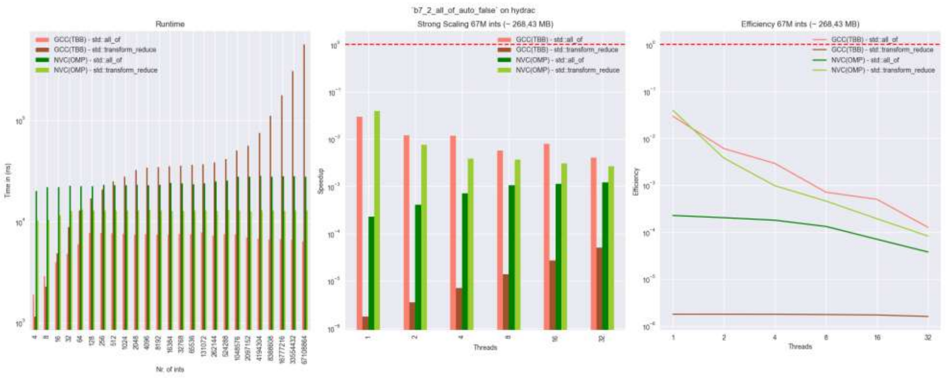
All_of Logic



(a) All elements true

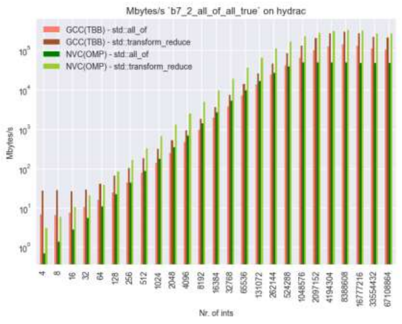


(b) First element false

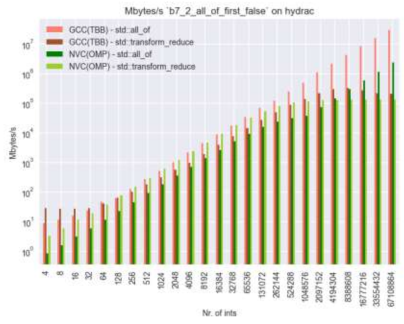


(c) All elements false

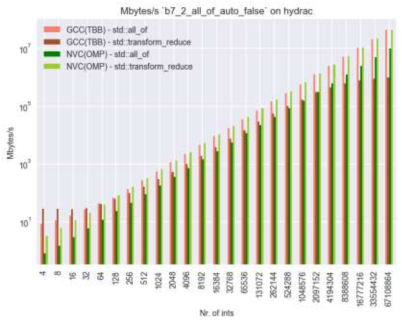
Figure D.28: Runtime & speedup results for "`All_of` Logic"; Hydra.



(a) All elements true



(b) First elements false



(c) All elements false

Figure D.29: Throughput results for "`All_of` Logic"; Hydra.

Stencil Logic

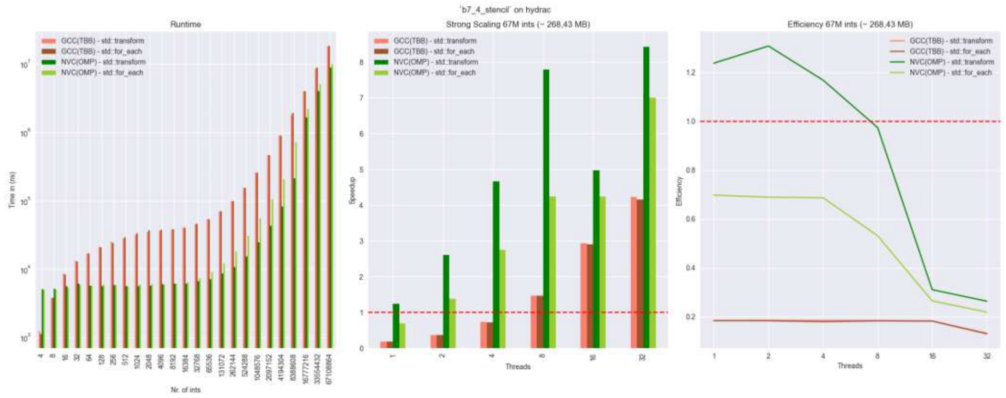


Figure D.30: Runtime & speedup results for "Stencil Logic"; Hydra.

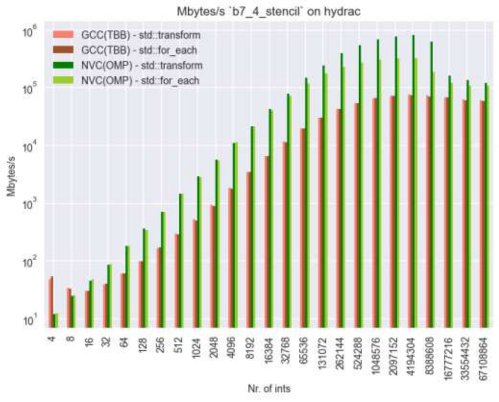


Figure D.31: Throughput results for “Stencil Logic”; Hydra.

Serial vs Direct Call

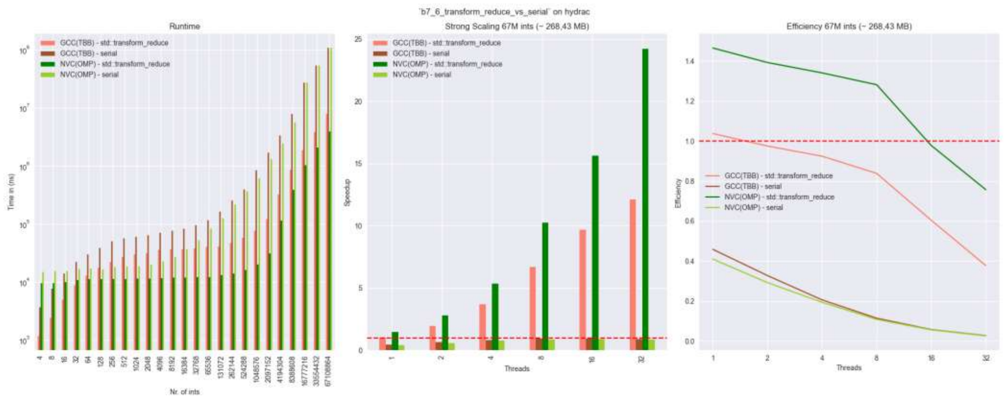


Figure D.32: Runtime & speedup results for “Serial vs Direct Call”; Hydra.

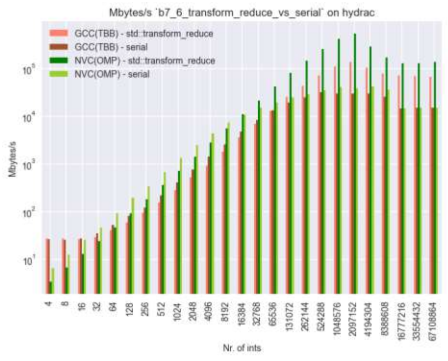


Figure D.33: Throughput results for “Serial vs Direct Call”; Hydra.

Count_if Logic

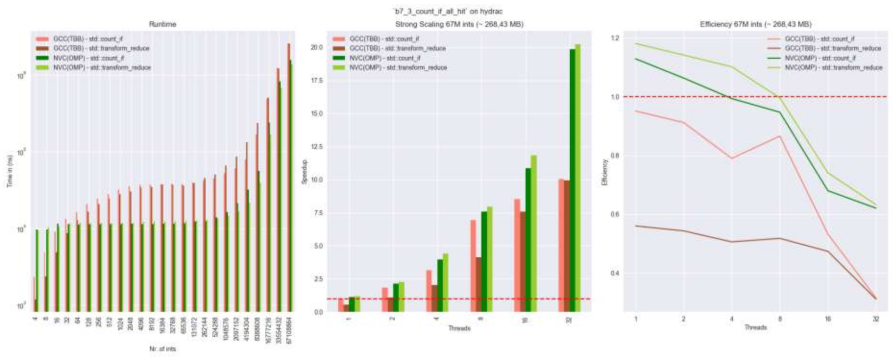


Figure D.34: Runtime & speedup results for "`Count_if` Logic" all hit; Hydra.

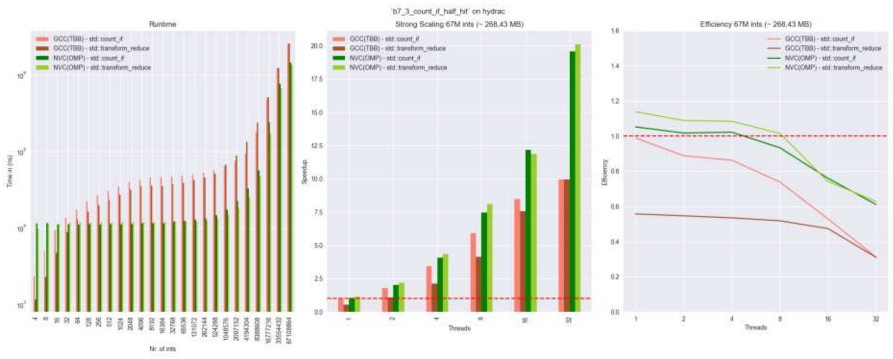


Figure D.35: Runtime & speedup results for "`Count_if` Logic" half hit; Hydra.

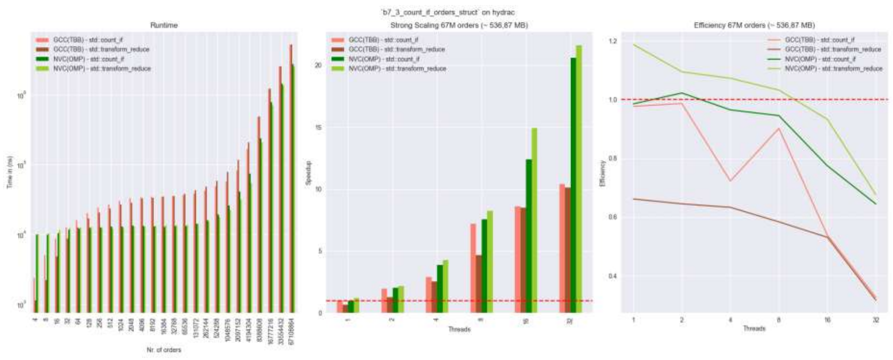
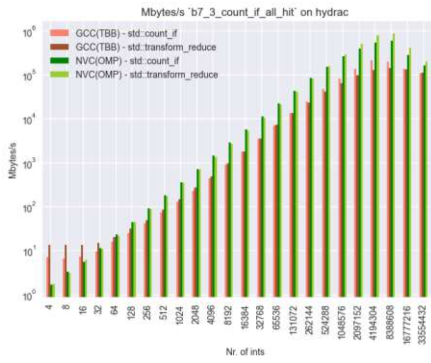
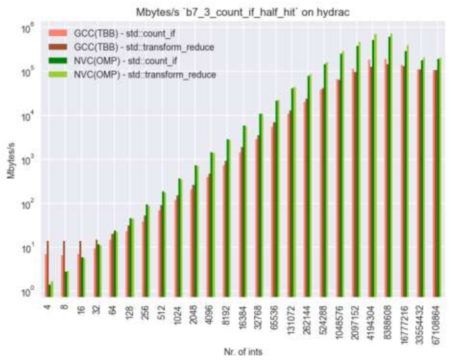


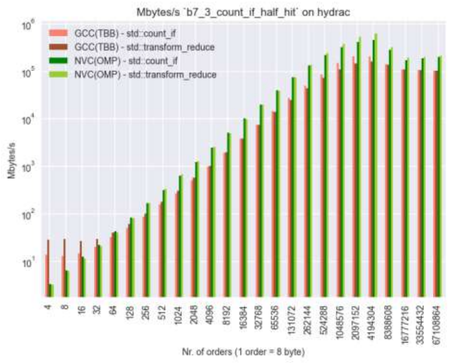
Figure D.36: Runtime & speedup results for "`Count_if` Logic" using structs; Hydra.



(a) All hit.



(b) Half hit.



(c) Structs.

Figure D.37: Throughput results for "`Count_if` Logic"; Hydra.

Scalar Transform Logic

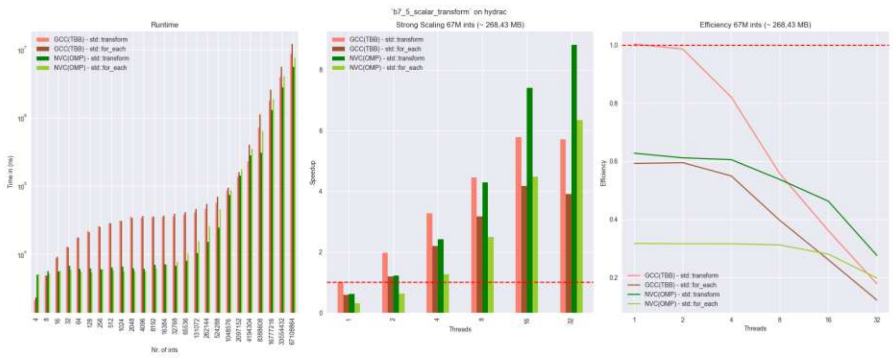


Figure D.38: Runtime & speedup results for "Scalar Transform Logic"; Hydra.

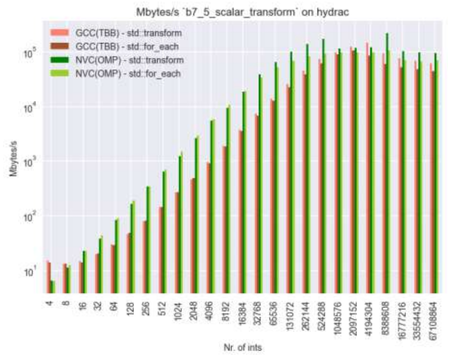


Figure D.39: Throughput results for “Scalar Transform Logic”; Hydra.

Custom Index-based Iterations

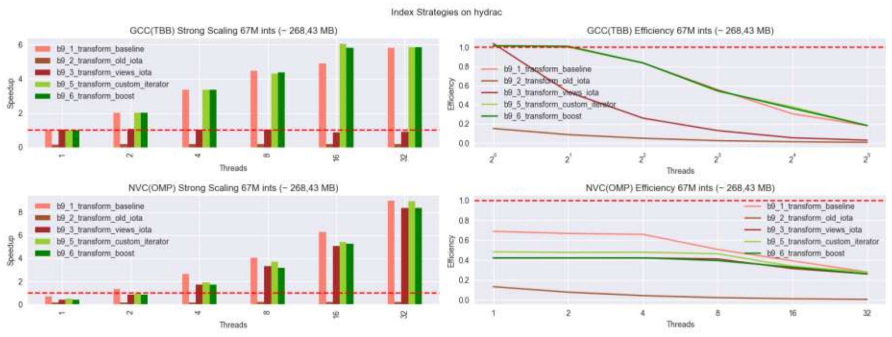


Figure D.40: Runtime & speedup results for various index strategies; Hydra.

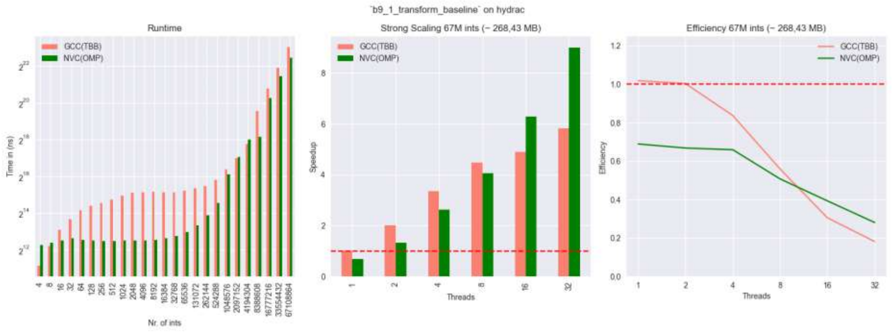


Figure D.41: Runtime & speedup results for “Data-Centric”; Hydra.

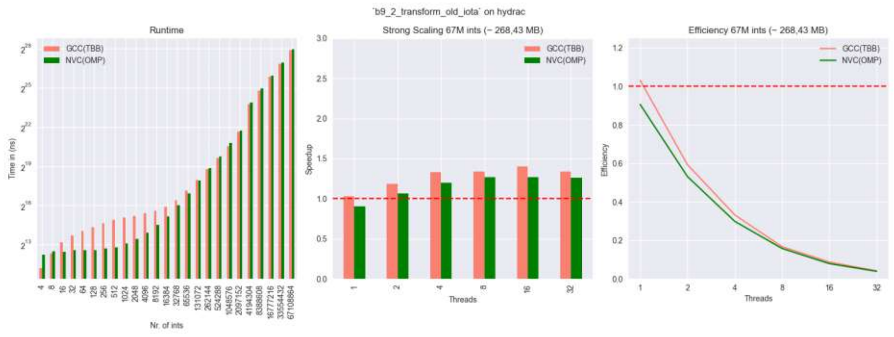


Figure D.42: Runtime & speedup results for “Index-Centric with iota”; Hydra.

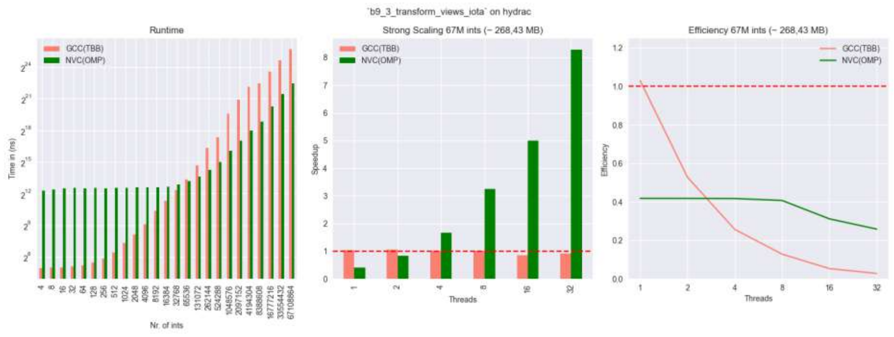


Figure D.43: Runtime & speedup results for “Index-Centric with views::iota”; Hydra.

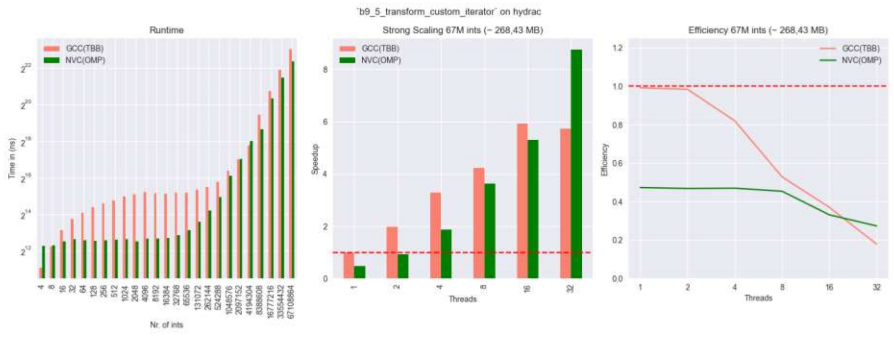


Figure D.44: Runtime & speedup results for “Index-Centric with Custom Iterator”; Hydra.

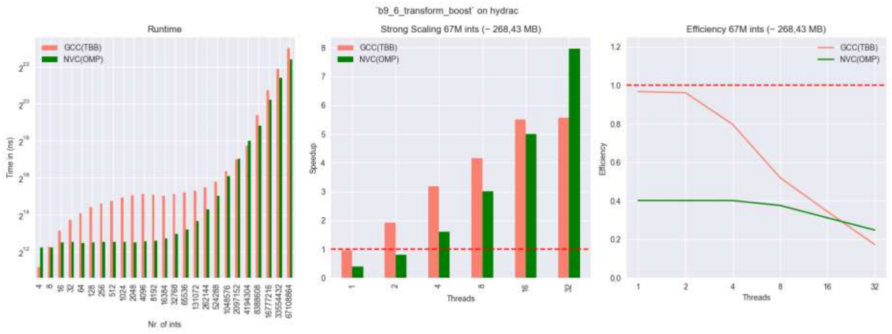


Figure D.45: Runtime & speedup results for “Index-Centric with Boost”; Hydra.

APPENDIX E

Usage of ChatGPT

While writing this thesis we used the tool ChatGPT to edit and improve the quality of our existing text. According to the "Frequently asked questions" of ACM [62] "it is not necessary to disclose such usage of these tools in your Work". Since we want to be as transparent as possible and do not have the intention to deceive others, we explicitly demonstrate how we used ChatGPT. In [63], one can see how we used the prompt *"Please write this better"* followed by our own existing text to enhance the quality. Since sometimes the results given by ChatGPT where not satisfactory, it was required to rewrite the responses or mix and match the results from different prompts. The main objective of using ChatGPT was to enhance the quality of the thesis, such that the complex construct can be delivered in easy understandable English.