

THESIS FOR THE DEGREE OF LICENTIATE OF ENGINEERING

Compile-time and Runtime Autotuning for Performance Portability in a Single-Source Programming Models

HARI ABRAM



Division of Computer and Network Systems
Department of Computer Science & Engineering
Chalmers University of Technology
Gothenburg, Sweden, 2026

Compile-time and Runtime Autotuning for Performance Portability in a Single-Source Programming Models

HARI ABRAM

Copyright ©2026 Hari Abram
except where otherwise stated.
All rights reserved.

Department of Computer Science & Engineering
Division of Computer and Network Systems
Chalmers University of Technology and Gothenburg University
Gothenburg, Sweden

This thesis has been prepared using L^AT_EX.
Printed by Chalmers Reproservice,
Gothenburg, Sweden 2026.

*“No person ever steps in the same river twice.”
- Heraclitus*

Abstract

Modern computing systems combine diverse CPU and GPU architectures. Single-source programming models such as SYCL aim to reduce the software-engineering burden of targeting these systems by allowing host and device code to be expressed in a common C++ program. Source-code portability, however, does not guarantee performance portability. The achieved performance depends on the mapping of the programming model to the target architecture, the compiler and compilation flow, the runtime backend, and the parameters used during compilation and execution.

This thesis investigates two complementary aspects of this problem. First, it characterizes how SYCL features perform when executing on modern vector CPUs. A suite of microbenchmarks is used to isolate memory bandwidth, kernel scheduling, synchronization, reduction, atomic operations, and vectorization. The study compares AdaptiveCpp and DPC++ SYCL implementations, ahead-of-time and just-in-time compilation, OpenMP and OpenCL backends, and four CPU architectures: Intel Ice Lake, AMD Zen 2, Fujitsu A64FX, and AWS Graviton3. The results show that no single toolchain is best for all features. Ahead-of-time compilation with an OpenMP backend provides the strongest memory-bandwidth performance, whereas just-in-time compilation is generally more effective for kernel scheduling and synchronization. The study also identifies substantial architecture-dependent differences in vectorization and in the implementation of reductions and atomic operations.

Second, the thesis studies whether best compile-time and runtime-tuned configurations are transferable across heterogeneous architectures. It introduces SCOUT, an automated framework that explores compiler flags and runtime parameters using Bayesian Tree-Structured Parzen Estimation and Tabu Search for a given objective. Ridge regression is used to estimate the influence of individual parameters. An evaluation of 26 SYCL workloads on AMD and Intel CPUs and NVIDIA and AMD GPUs shows that autotuning can provide substantial improvements, particularly on CPUs, but that the effect of individual compiler flags is rarely universal. Bayesian tuning provides the most stable improvements, while Tabu Search exposes influential parameter interactions at the cost of higher variance. Runtime choices, including the backend, thread placement, scheduling, and wait policy, can alter CPU performance by several times.

Together, the two studies show that achieving performance portability requires architecture-aware selection of compilation flows, backends, compiler optimizations, and runtime policies. The thesis therefore motivates adaptive and explainable tuning methods that preserve a portable source representation while specializing its execution to the target system.

Keywords: Heterogeneous Computing, Performance Portability, Autotuning, Search Algorithms

Acknowledgment

I would like to express my sincere gratitude to my supervisor, Miquel Pericàs, for his guidance and support throughout this work. I am also grateful to my collaborators and co-authors, Nikela Papadopoulou and Jens Domke, for their contributions, discussions, and feedback. The research presented in this thesis has benefited greatly from these collaborations and from the research environment at Chalmers University of Technology.

This work has received funding from the European High-Performance Computing Joint Undertaking (JU) under Framework Partnership Agreement No. 800928 and Specific Grant Agreement No. 101036168 (EPI SGA2), with support from the European Union’s Horizon 2020 research and innovation programme and the Swedish Research Council, among others. The work has also received funding from the Swedish Foundation for Strategic Research through the PRIDE project (reference number CHI19-0048) and from the Swedish Research Council through the P4PIM project (project ID 2020-04892).

The computations presented in this thesis were enabled by resources provided by Chalmers e-Commons and by the National Academic Infrastructure for Supercomputing in Sweden (NAISS), partially funded by the Swedish Research Council through grant agreement No. 2022-06725. I am also grateful to the Barcelona Supercomputing Center, Spain, for providing access to the MareNostrum 4 ARM-CTE partition; the University of Bristol, United Kingdom, for providing access to Isambard 2; and RIKEN Center for Computational Science (RIKEN-RCCS), Japan, for the opportunity to undertake two internships under the supervision of Jens Domke and for providing access to the A64FX systems and RCCS Cloud infrastructure.

List of Publications

Appended publications

This thesis is based on the following publications:

- [I] Hari Abram, Nikela Papadopoulou, and Miquel Pericàs
“Exploring SYCL as a portability layer for high-performance computing on CPUs”
Published in ISC High Performance 2025.
- [II] Hari Abram, Nikela Papadopoulou, Jens Domke, and Miquel Pericàs
“Cross-Architecture Autotuning for Single-Source Heterogeneous Programming Models”
Published in ICS 2026.

Contents

Abstract	v
Acknowledgement	vii
List of Publications	ix
1 Introduction	1
1.1 Background	1
1.2 Related Work	3
1.3 Problem Statements	5
1.4 Contributions	6
2 Summary of the Papers	9
2.1 Paper I - Summary	9
2.2 Paper II - Summary	11
3 Conclusions and Future Work	15
3.1 Conclusions	15
3.2 Future Work	16
Bibliography	19
Paper I	23
Paper II	37

Chapter 1

Introduction

1.1 Background

High-performance computing (HPC) systems have become increasingly heterogeneous. Contemporary systems combine processors with different instruction-set architectures, core organizations, vector widths, memory hierarchies, and accelerator interfaces. CPUs continue to increase their core counts and vector-processing capabilities, while GPUs provide large numbers of throughput-oriented execution units and high memory bandwidth. This diversity allows applications to select hardware that matches their computational characteristics, but it also increases the cost of developing and maintaining high-performance software.

Historically, applications have often been implemented separately for each class of device, for example using OpenMP for multicore CPUs and CUDA or HIP for GPUs. Such implementations can provide high performance, but maintaining multiple versions of an application introduces duplicated code, diverging optimization strategies, and substantial porting effort. Portable programming models attempt to reduce this burden by providing a common abstraction over multiple architectures. Examples include OpenCL, OpenACC, Kokkos, RAJA, and SYCL. In our work, we use SYCL as a programming model to investigate portability; SYCL is a royalty-free, standard C++ programming model for heterogeneous computing [1].

SYCL follows a single-source model in which host code and kernels are expressed in the same C++ source file. Kernels are submitted to a queue and execute over an iteration space described using constructs such as `sycl::range` and `sycl::nd_range`. Data can be managed using buffers and accessors or Unified Shared Memory (USM). These abstractions allow the same source program to target CPUs and GPUs. SYCL therefore provides functional portability: a conforming program can be compiled and executed on different supported systems without maintaining independent source versions.

Functional portability is only one part of the problem. HPC applications also require performance portability, meaning that a portable program should make effective use of the available hardware across its target platforms. A program may execute correctly on every system while achieving only a small fraction of the performance attainable by a native implementation. Performance

portability is difficult because the architectural mechanisms that provide high performance differ substantially across devices. CPUs rely on a combination of multithreading, cache locality, branch prediction, and SIMD vectorization, whereas GPUs rely on large-scale thread-level parallelism, coalesced memory accesses, and latency hiding. Consequently, an execution strategy that is beneficial on one architecture may be ineffective or harmful on another.

The implementation of the programming model forms an additional source of variation. SYCL is an open standard with multiple implementations. In this thesis, the main implementations are AdaptiveCpp [2] and DPC++ [3]. These implementations may use different compilation flows and backend runtimes. For example, a SYCL kernel targeting a CPU can be compiled ahead of time and offloaded using an OpenMP backend, or it can be represented as an intermediate form and lowered just in time through an OpenMP or OpenCL backend. The same source-level construct may therefore be implemented using different scheduling, synchronization, memory-management, and vectorization mechanisms.

This multi-stage compilation model is particularly important for single-source heterogeneous programming. With AdaptiveCpp, the compiler frontend translates the source program to LLVM intermediate representation (LLVM-IR). The device code can subsequently be lowered to CPU machine code, SPIR-V/OpenCL, NVIDIA PTX, or AMD GPU code, depending on the selected backend and target. Each stage exposes optimization choices. The frontend is affected by standard compiler flags, the backend applies architecture-specific optimization and code generation, and the runtime controls thread placement, scheduling, synchronization, and data management. The final performance is therefore determined by interactions across the complete compilation and execution pipeline.

The large number of choices makes manual optimization difficult. Compiler optimization flags interact non-linearly, and runtime parameters are often conditional on the selected backend. A configuration may improve one workload and degrade another, and a configuration found on a CPU may not transfer to a GPU even though both devices execute code generated from the same source and intermediate representation. Autotuning addresses this problem by empirically searching a parameter space and evaluating candidate configurations on the target system. Search algorithms such as Bayesian optimization can reduce the number of evaluations required, but the resulting configurations may be difficult to interpret. Conversely, local-search methods can make parameter interactions easier to observe but may explore unsafe configurations or become trapped in local regions of the search space.

This thesis studies performance portability at these two levels. Paper I first characterizes the behavior of individual SYCL features on multicore vector CPUs and determines how this behavior changes with the implementation, compilation flow, backend, and target architecture. Paper II then investigates whether compiler and runtime parameters can be tuned automatically and whether the resulting decisions transfer across CPU and GPU architectures. The first paper therefore identifies and explains sources of performance variation, while the second develops and evaluates a method for navigating that variation.

1.2 Related Work

A range of programming systems has been proposed to reduce the effort required to support heterogeneous architectures. Directive-based models such as OpenMP and OpenACC extend existing C, C++, and Fortran programs with annotations for parallel execution and device offloading [4, 5]. Library-based portability layers such as Kokkos and RAJA provide architecture-independent abstractions for parallel execution and, in the case of Kokkos, memory management [6, 7]. OpenCL exposes a comparatively low-level platform and kernel programming interface, whereas SYCL provides higher-level, single-source C++ abstractions over a heterogeneous execution model [1, 8]. These approaches differ in their abstraction level and compilation strategy, but share the objective of allowing a common source representation to target multiple classes of processors.

A common programming interface, however, does not by itself guarantee comparable efficiency across systems. Pennycook et al. formalized performance portability by considering both the efficiency attained on each supported platform and the coverage of a specified platform set [9]. Subsequent application studies have shown that portable programming models can achieve competitive performance, but that the results remain dependent on the workload, compiler implementation, runtime backend, and amount of target-specific specialization. For example, studies of matrix-free solvers and bandwidth-bound scientific applications show that SYCL performance can approach that of native programming models on some devices while falling behind on others [10, 11]. These observations motivate treating the compiler and runtime system as integral parts of the performance-portability problem.

The growing adoption of SYCL has also motivated the development of benchmark suites for comparing implementations and devices. SYCL-Bench provides benchmarks for device characterization and runtime-level evaluation [12]. The Parallel Research Kernels provide compact computational patterns for examining the behavior of architectures and parallel runtime systems [13], while HeCBench contains a broader collection of heterogeneous workloads implemented using SYCL and other programming models [14]. These suites are valuable for evaluating end-to-end behavior across representative kernels and applications.

Application-level measurements, however, combine the effects of kernel computation, memory management, task scheduling, synchronization, launch overheads, and compiler code generation. Consequently, a performance difference observed for a complete application does not directly reveal which SYCL abstraction or software layer caused that difference. Previous studies have compared the performance of HPC-style SYCL applications and different data-parallel kernel formulations, demonstrating that iteration spaces, data-management methods, and runtime implementations can substantially affect performance [11, 15, 16]. Nevertheless, the combined design space formed by Ahead-of-Time and Just-in-Time compilation, OpenMP and OpenCL backends, different iteration spaces, and multiple memory models has not been characterized systematically for multicore vector CPUs.

Paper I addresses this limitation through a set of microbenchmarks that isolate memory bandwidth, parallelization, atomic operations, reductions, barriers,

and vectorization. Its emphasis on multicore vector CPUs is important because SYCL’s GPU-oriented execution abstractions must be mapped onto CPU threads, work-sharing mechanisms, cache hierarchies, and SIMD instructions. By evaluating the same features under different compilation flows, backend runtimes, iteration spaces, and data-management methods, Paper I identifies the software layers responsible for the observed performance differences and derives practical recommendations for executing SYCL programs on CPUs.

Autotuning provides a complementary approach to performance portability by replacing manually selected optimization parameters with empirical design-space exploration. OpenTuner introduced a general framework that combines multiple search techniques for program optimization [17]. CLTune and Kernel Tuner focus primarily on accelerator kernels and expose parameters such as work-group dimensions, tiling factors, vector widths, and loop-unrolling choices [18,19]. GPTune applies Bayesian optimization and multitask learning to reduce the cost of tuning related HPC problems across inputs and platforms [20]. More recent work has jointly explored compiler, runtime, and hardware parameters, confirming that performance depends on decisions made at several layers of the execution stack [21].

Most existing autotuning frameworks nevertheless target a single programming model, a single device class, or a fixed compilation pipeline. Single-source heterogeneous programming introduces an additional complication: a compiler option selected before target-specific lowering may influence the code generated for CPUs and GPUs in different ways. Moreover, the relevant runtime parameters depend on the selected backend. The resulting search space is conditional and contains interacting categorical, Boolean, and parametric choices. Paper II addresses this gap by introducing a unified workflow for tuning compiler flags and runtime parameters in SYCL and by applying the same tuning methodology across multiple CPU and GPU architectures.

A further limitation of black-box autotuning is its restricted explainability. An optimizer may identify a high-performing configuration without revealing which parameter produced the improvement, which interactions were necessary, or whether the same effect transfers to another architecture. Paper II therefore combines complementary search and analysis methods. Bayesian Tree-Structured Parzen Estimation, implemented through Optuna, provides a probabilistic global search strategy [22]. Tabu Search provides a neighborhood-based exploration whose sequence of configuration changes can be examined directly [23]. Ridge regression is subsequently applied to the collected trials to estimate the conditional influence of individual compiler flags and runtime settings [24].

Taken together, the two papers address consecutive stages of a performance-engineering workflow. Paper I characterizes how SYCL abstractions are mapped through different compilation and runtime paths and identifies the layers that introduce performance overheads. Paper II investigates how the resulting configuration space can be searched automatically and whether the discovered optimization decisions transfer between architectures. In contrast to work that considers only application-level performance, this thesis combines feature-level characterization with application-level autotuning. In contrast to work that tunes a single target, it explicitly investigates the transferability of compiler and runtime decisions across CPUs and GPUs within a single-source heterogeneous

programming model.

1.3 Problem Statements

Context 1: SYCL execution on multicore vector CPUs

SYCL originates from a throughput-oriented execution model in which many work items execute a common kernel. Modern CPUs can execute this model, but their performance depends on how work items are mapped to CPU threads, how loops are reconstructed and vectorized, how synchronization is implemented, and how data is represented in the backend runtime. Developers can choose between multiple SYCL implementations, ahead-of-time and just-in-time compilation, OpenMP and OpenCL backends, different iteration spaces, and different data-management methods.

Problem 1. There is insufficient isolated and systematic evidence showing how these choices affect the performance of individual SYCL features on modern multicore vector CPUs. Application benchmarks reveal whether a complete program is fast or slow, but they make it difficult to attribute the result to a particular construct, compilation flow, or runtime mechanism. Without such attribution, developers cannot reliably select a CPU toolchain or understand why the same SYCL program behaves differently across architectures.

Challenges. Addressing this problem requires: (i) isolating the cost of memory access, scheduling, synchronization, and vectorization without allowing application complexity to dominate the measurement; (ii) constructing equivalent baselines for CPU-oriented comparison; (iii) evaluating different implementations, versions, compilation flows, backends, iteration spaces, and data-management methods; (iv) comparing CPUs with different ISAs, vector widths, core counts, and memory systems; and (v) distinguishing runtime overheads from the quality of generated machine code.

Research Question 1 (RQ1). *How do the SYCL implementation, compilation flow, backend runtime, and programming constructs affect the performance of SYCL on multicore vector CPUs?*

RQ1 is refined into the following sub-questions:

RQ1.1 How do these choices affect data management, kernel scheduling, synchronization, atomic operations, and reductions?

RQ1.2 How effectively do the different toolchains vectorize kernels across work items and within work items on x86 and Arm CPUs?

RQ1.3 Which toolchain choices provide robust recommendations for developers targeting CPUs with SYCL?

Context 2: Cross-architecture compiler and runtime tuning

The characterization in Paper I shows that no single compilation flow or backend is optimal for every feature and architecture. At the application level, the design

space is even larger. Compiler flags enable or disable interacting optimization passes, while runtime variables control backend selection, thread placement, scheduling, affinity, and waiting behavior. In a single-source workflow, compile-time decisions may be made before the final execution device is known, and a distributed binary may embed intermediate representation that is later specialized to the target.

Problem 2. It is unclear whether configurations identified on one architecture retain their performance effect on another architecture, particularly between CPUs and GPUs. The parameter space is too large and irregular for exhaustive search, and a configuration that is beneficial for one benchmark or platform may cause a severe regression elsewhere. Existing black-box autotuners can find good configurations, but often provide limited evidence about why a parameter in the selected configuration is beneficial or harmful.

Challenges. Addressing this problem requires: (i) a unified system that can compile, execute, and measure heterogeneous SYCL workloads; (ii) support for categorical, parametric, optional, and backend-conditional parameters; (iii) search algorithms that balance exploration, tuning cost, and robustness; (iv) repeated measurements and consistent objective evaluation across CPUs and GPUs; (v) methods for attributing performance changes to individual parameters despite interactions among them; and (vi) a cross-platform evaluation that separates portable effects from architecture-specific effects.

Research Question 2 (RQ2). *To what extent do autotuned compiler and runtime decisions transfer across CPU and GPU architectures, and how can the performance effects of individual parameters be identified?*

RQ2 is refined into the following sub-questions:

RQ2.1 Which search strategy provides the most useful balance between performance improvement, exploration, and robustness for SYCL workloads?

RQ2.2 Which compiler optimizations have consistent effects across architectures, and which effects are target-specific?

RQ2.3 How strongly do backend selection, thread placement, scheduling, binding, and wait policies influence CPU performance?

1.4 Contributions

This thesis is based on two papers. Paper I addresses RQ1 through a feature-level characterization of SYCL on multicore vector CPUs. Its main contributions are:

- We systematically evaluate the various runtimes and compilation flow toolchains of SYCL implementations when targeting modern multicore CPUs with vectorization capabilities.
- We assess the potential of SYCL for portability across different CPU architectures by evaluating its utilization of available features on various hardware platforms.

- We show that the achieved memory bandwidth is strongly influenced by the compilation flow, with AoT compilation achieving the highest possible bandwidth across all systems. In contrast, parallelization shows better performance with JIT compilation. Additionally, our findings reveal that, on certain systems, the cost model plays a significant role in the vectorization of the kernel

Paper II addresses RQ2 through automated design-space exploration of compiler and runtime parameters. Its main contributions are:

- We adapt a meta-heuristic search algorithm to the compile-time and runtime autotuning problem, improving the explainability of the tuning process by revealing how specific compiler flags and runtime settings influence performance.
- We conduct a detailed experimental study to examine how auto-tuned parameters vary across CPU and GPU architectures, encompassing devices from multiple hardware vendors.
- We employ advanced statistical analysis methods to identify and quantify the impact of individual configuration parameters on benchmark performance

Taken together, the papers contribute a progression from diagnosis to adaptation. Paper I establishes that performance variation is introduced by the mapping of SYCL abstractions through different compilation and runtime paths. Paper II shows that these choices can be searched automatically, but that the resulting knowledge must remain architecture-aware. The combined contribution is an empirical and methodological basis for designing adaptive SYCL systems that retain a portable source program while specializing execution to the target platform.

The rest of the thesis is organized as follows. Chapter 2 summarizes the two appended papers and relates their results to the research questions. Chapter 3 synthesizes the conclusions, discusses limitations, and outlines directions for future work.

Chapter 2

Summary of the Papers

2.1 Paper I - Summary

Paper I investigates how effectively SYCL maps a GPU-oriented parallel execution model to modern multicore vector CPUs. The paper is motivated by two concurrent trends. First, CPUs increasingly provide wide vector units, large core counts, and high memory bandwidth, making them suitable for workloads that have traditionally been associated with accelerators. Second, SYCL provides a single-source path to execute the same program on CPUs and GPUs. The central question is not whether a SYCL program can execute on a CPU, but which combination of programming constructs, compiler implementation, compilation flow, and runtime backend allows it to use CPU resources efficiently.

A direct application-level comparison is insufficient for answering this question because a complete application combines several sources of overhead. Therefore, in this work we utilize a micro-benchmark methodology inspired by established OpenMP microbenchmarking. The suite isolates four major aspects of SYCL execution: memory bandwidth, parallelization and scheduling, synchronization, and vectorization. Each microbenchmark is parameterized so that the same operation can be executed with different iteration spaces, dimensions, and data management strategies.

The memory-bandwidth benchmark evaluates the different forms of USM allocation and compares kernels expressed using `range` and `nd_range`. BabelStream with OpenMP is used as a reference for attainable CPU memory bandwidth. The parallelization benchmark minimizes the computation performed by each work item to expose scheduling and synchronization overheads. The atomic and reduction benchmarks implement array accumulation using `sycl::atomic_ref` and `sycl::reduction`, respectively, and compare them with OpenMP atomic and reduction constructs. Barrier benchmarks measure group and subgroup synchronization. Finally, GEMV, GEMM, outer product, and a conditional cross-product kernel are used to study vectorization across work items and within the loop body of a work item.

The experimental study covers two SYCL implementations, AdaptiveCpp and DPC++, and three principal compilation/backend paths: AdaptiveCpp ahead-of-time compilation with OpenMP, AdaptiveCpp just-in-time compilation with OpenMP or OpenCL, and DPC++ just-in-time compilation with OpenCL.

Four processors are used: Intel Xeon Platinum 8358 (Ice Lake), AMD EPYC 7742 (Zen 2), Fujitsu A64FX, and AWS Graviton3. These systems represent x86-64 and Arm architectures with different vector widths, core organizations, and memory systems. OpenMP implementations compiled with LLVM provide the native CPU baseline.

The first part of the evaluation studies historical versions of AdaptiveCpp and DPC++ on the x86 systems. The results show that implementation maturity does not improve every feature uniformly. DPC++ scheduling performance improves across the evaluated releases, particularly for one-dimensional iteration spaces, while barrier performance changes less consistently. This analysis establishes that performance depends not only on the SYCL standard but also on the implementation version and the evolution of its runtime.

The backend comparison provides the clearest evidence that the compilation and runtime path changes the nature of the generated CPU execution. For memory bandwidth, ahead-of-time compilation with the OpenMP backend is consistently the strongest configuration. On the Intel system, it approaches the OpenMP BabelStream result, whereas the OpenCL path incurs substantially more memory activity and stalls. The hardware-counter analysis reports considerably more TLB misses, loads and stores, and cycles stalled on memory for the OpenCL configuration. These results indicate that memory-management and runtime abstractions can prevent a bandwidth-bound kernel from reaching the capability of the underlying CPU even when the source-level computation is simple.

The result differs for parallelization. On Intel Ice Lake, the just-in-time OpenCL path is up to approximately 10% faster than the OpenMP path for the studied scheduling microbenchmark, except at the smallest problem size. The just-in-time OpenMP result is close to native OpenMP. USM and buffer/accessor management perform similarly in this benchmark because little data is moved. The comparison demonstrates that a backend that is unfavorable for memory bandwidth can nevertheless provide efficient kernel scheduling. This is an important result for RQ1: there is no single ranking of compilation flows that holds across all SYCL features.

Atomic operations show relatively small differences between the OpenMP and OpenCL backends on the Intel system, but reductions expose a significant gap between SYCL and native OpenMP. The USM reduction is up to five times slower than OpenMP for the evaluated problem sizes. Profiling attributes part of this overhead to a higher instruction count, cache and TLB activity, and time spent in `__sched_yield`, indicating resource contention in the runtime implementation. The relative overhead decreases for larger problems, as useful computation becomes more dominant. Barrier performance is less sensitive to the evaluated choices and shows no consistent difference across systems, backends, and compilation flows.

The cross-platform comparison demonstrates that architecture-specific behavior remains substantial even when the same SYCL implementation is used. A64FX achieves memory bandwidth close to the OpenMP reference, showing that SYCL can use its high-bandwidth memory effectively. At the same time, it performs poorly for the scheduling and atomic microbenchmarks. The atomic result is especially weak relative to the other systems, consistent with the cost of the atomic instruction sequences available on the platform. The x86

systems generally perform better than the Arm systems for the parallelization benchmark, while reduction performance is strongest on Intel Ice Lake and Graviton3 among the evaluated systems.

The vectorization analysis further shows that source portability does not imply uniform SIMD code generation. On Intel Ice Lake, the OpenCL backend executes 512-bit vector instructions for several kernels, while the evaluated OpenMP paths primarily use 128- and 256-bit instructions. On AMD Zen 2, just-in-time compilation more often generates fused multiply-add instructions, whereas the ahead-of-time path more frequently uses separate multiplication instructions. A64FX generates SIMD and SVE instructions for several regular kernels but fails to vectorize the conditional cross-product. Graviton3 exhibits the most limited behavior: the just-in-time OpenMP path does not vectorize the evaluated kernels, and the ahead-of-time path vectorizes only a subset. Compiler reports indicate that the cost model considers some transformations unprofitable. None of the SYCL toolchains vectorizes the `sycl::reduction` benchmark, while the corresponding OpenMP reduction is vectorized.

Paper I shows that CPU performance is jointly determined by the SYCL construct, the compilation flow, the backend runtime, and the target architecture. Ahead-of-time OpenMP compilation is the preferred path for bandwidth-oriented kernels, whereas just-in-time compilation is generally more competitive for scheduling and synchronization. The result is not a universal prescription but a feature-dependent decision process. The paper therefore provides a set of recommendations and establishes the need for adaptive selection.

2.2 Paper II - Summary

Paper II starts from the observation established in Paper I: the performance of a SYCL program depends on decisions made at several stages of the software stack, and the best decision varies by feature and architecture. Rather than manually selecting these decisions, Paper II studies compiler- and runtime-level autotuning. Its main objective is to determine whether parameters discovered on one architecture have similar effects on another and whether the tuning process can explain why a configuration succeeds or fails.

The paper introduces SCOuT, a Python-based framework for objective-driven design-space exploration. SCOuT is controlled by a JSON specification that describes the project, compiler, parameter space, search algorithm, measurement method, and optimization objectives. For each trial, the framework constructs a candidate configuration, builds the benchmark, applies the selected runtime environment, executes repeated measurements, computes the objective, and records the result. It supports both direct compilation of a source file and existing Make or CMake projects. Metrics can be collected through LIKWID, `perf`, or a user-provided parser.

The parameter model includes mutually exclusive compiler flags, an optional pool of on/off flags, flags with values, and runtime environment variables. Conditional generation is particularly important: OpenMP-specific variables are explored only when the OpenMP backend is selected, while OpenCL-specific variables are explored only for the OpenCL backend. This allows compiler and runtime decisions to be studied independently or jointly without generating

invalid configurations.

Two search strategies are evaluated. The first is Bayesian Tree-Structured Parzen Estimation (TPE) through Optuna. TPE builds probabilistic models of promising and non-promising regions and samples configurations that are likely to improve the objective. It provides a global search and can concentrate the trial budget on configurations similar to successful observations. The second strategy is Tabu Search, a neighborhood-based metaheuristic. Starting from a base configuration, it evaluates nearby configurations, moves to the best candidate, and records visited configurations in a tabu list to reduce cycling. Its iterative path makes the effect of adding or removing individual flags easier to inspect.

The two methods have complementary strengths. TPE is intended to find good configurations reliably under a bounded trial budget. Tabu Search deliberately explores local neighborhoods and therefore evaluates more unique configurations, but it can visit configurations with severe performance regressions. To improve the interpretability of TPE, Paper II fits a ridge regression model to the trial data. Each compiler flag or environment setting is encoded as a binary feature, and the fitted coefficient estimates its conditional association with the measured runtime. The statistical model does not replace measurement, but it provides evidence about which parameters consistently contribute to the observed result.

The evaluation uses AdaptiveCpp version 25.02.0 with LLVM 19.1.7 and 26 workloads from HeCBench. The workloads cover stencils, lattice-Boltzmann methods, signal processing, transforms, linear algebra, graph and sparse algorithms, molecular dynamics, finance, and machine learning. The target systems are AMD Genoa and Intel Ice Lake CPUs, and NVIDIA A40 and AMD MI100 GPUs. TPE uses a budget of at most 50 trials per benchmark. Tabu Search uses at most six iterations with 12 candidates per neighborhood, a tabu tenure of three, and early termination after two iterations without improvement.

The search-algorithm comparison shows that Tabu Search usually evaluates more unique configurations, because TPE may resample configurations in a promising region. More exploration does not, however, imply a better production tuning strategy. TPE produces moderate and stable improvements on both the AMD CPU and NVIDIA GPU and rarely selects configurations that are far worse than the baseline. Tabu Search occasionally finds competitive best configurations but also encounters large regressions, particularly on the GPU. The result answers RQ2.1: TPE offers the better balance for automated or production-oriented tuning, whereas iteration-limited Tabu Search is useful when the objective is to expose interactions and map hazardous regions of the parameter space.

The compile-time study shows that the influence of individual flags varies across architectures. Fast-math options are among the most consistently beneficial parameters in the evaluated workloads, subject to the numerical semantics of the application. A notable CPU-specific case occurs for `dct8x8` on AMD Genoa, where `-fno-builtin` enables wider vector instructions and produces an approximately threefold improvement. The same flag does not have a comparable effect on the GPUs. In contrast, `-fno-unroll-loops` and flags that disable loop-level heuristics often cause large slowdowns on the NVIDIA GPU and can also degrade Intel CPU performance. Other effects arise only

from particular combinations, demonstrating that compiler passes interact and that a flag cannot be classified as universally beneficial or harmful in isolation.

These observations answer RQ2.2. Some transformations have a similar direction of effect across several workloads, but the correlations are neither strong nor universal. CPU-optimal flags do not reliably transfer to GPUs, and the magnitude of compiler-level tuning is generally larger on the evaluated CPUs. The same frontend option can alter target-specific optimization and code generation in different ways after the common LLVM-IR stage. A single-source representation therefore provides a common input to the toolchain, not a common optimization landscape.

The runtime study focuses on CPUs, where the evaluated backends expose a richer set of environment variables. On AMD Genoa, the OpenMP backend outperforms OpenCL for most workloads under the default settings. The largest reported gap is the `gru` benchmark, for which OpenMP is 8.93 times faster. The result is not universal: memory-bound workloads can favor OpenCL, and `lavaMD` is reported to run 3.57 times faster with OpenCL. Profiling shows that OpenCL can provide stronger vectorization in some of these cases, while OpenMP generally has a lower memory footprint and fewer cache misses.

Thread placement and scheduling produce equally important effects. Configuring threads over `numa_domains` frequently appears in the worst configurations and can degrade runtime by an order of magnitude. Placement over cores, hardware threads, or sockets is generally safer. Compact binding improves locality for several compute-intensive workloads, while dynamic scheduling is useful for irregular workloads and static scheduling remains appropriate for uniform kernels. For the OpenMP backend, an active wait policy consistently improves several benchmarks; in `d2q9-bgk`, changing from passive to active waiting provides an improvement of up to two times. These results answer RQ2.3 by showing that runtime configuration is not a secondary detail: on CPUs it can dominate the effect of compiler flags.

Paper II concludes that autotuning configurations are not inherently portable across devices. Architecture-aware tuning is required even within a single-source programming model and a shared compiler infrastructure. The paper also shows that tuning should report more than the best configuration. A robust workflow should expose regressions, parameter interactions, and the confidence with which a parameter can be considered beneficial. TPE, Tabu Search, and ridge regression provide complementary views of this information.

Chapter 3

Conclusions and Future Work

3.1 Conclusions

This thesis investigated performance portability in SYCL from two complementary perspectives: the behavior of the programming model and its implementations on multicore vector CPUs, and the transferability of compiler/runtime tuning decisions across CPU and GPU architectures. The common conclusion is that performance portability cannot be attributed to the source language alone. It emerges from the interaction between the source program, compiler frontend, intermediate representation, target-specific optimization, backend runtime, and hardware architecture.

Paper I addressed RQ1 by isolating important SYCL features and evaluating them across implementations, compilation flows, backends, and CPUs. The experiments show that the toolchain ranking depends on the operation being performed. Ahead-of-time compilation with an OpenMP backend is effective for attaining memory bandwidth, while just-in-time compilation is generally more effective for scheduling and synchronization. Atomic operations, reductions, and vectorization expose additional platform-specific behavior. In particular, the evaluated SYCL reductions are not vectorized, and the Arm systems show less consistent vectorization than the x86 systems. The results demonstrate that mapping a throughput-oriented execution abstraction to CPUs requires careful coordination between runtime parallelization and compiler vectorization.

Paper II addressed RQ2 by treating these compiler and runtime choices as an autotuning problem. SCOUT provides a unified workflow for exploring mixed and conditional parameter spaces, while TPE, Tabu Search, and ridge regression provide complementary mechanisms for autotuning and explanation. The experiments show that tuning can yield substantial gains, particularly on CPUs, but that parameter effects do not generally transfer between CPUs and GPUs. TPE provides the most stable improvements, whereas Tabu Search reveals local interactions but encounters a much wider performance range. Runtime backend and thread-management policies can change CPU performance by several times, which confirms that runtime adaptation is as

important as compiler-flag tuning.

The two papers jointly support five broader conclusions.

- **Source portability and performance portability are distinct.** A common source program removes the need to maintain independent implementations, but the generated execution remains architecture-specific.
- **The backend is an optimization decision.** OpenMP and OpenCL can lead to substantially different CPU performance because they follow different scheduling, memory-management, and vectorization paths. As a result, the preferred backend depends on the characteristics of the workload.
- **Performance portability is feature-dependent.** A toolchain may be portable for memory bandwidth but not for reductions or vectorization. Aggregate application performance can hide this distinction.
- **Autotuning knowledge has limited transferability.** Even when CPUs and GPUs share the same source frontend and LLVM representation, the effect of a flag can change after target-specific lowering. A configuration should not be deployed on a new architecture without validation.
- **Explainability and safety matter.** The best measured configuration is insufficient if the tuning process also encounters severe regressions or if its effect depends on an unexplained interaction. Practical autotuning should quantify both improvement and risk and should identify the parameters responsible for each.

The resulting view of performance portability is therefore hierarchical. The source level provides functional portability; the compiler and intermediate representation provide opportunities for common analysis; the target backend provides architecture-specific optimization; and the runtime adapts execution to the current device and system topology. A performance-portable SYCL system must preserve the benefits of the upper layers while allowing specialization at the lower layers.

3.2 Future Work

Future work will first extend the performance characterization of SYCL across a broader range of architectures, implementations, and compilation workflows. The microbenchmark study in Paper I showed that features such as memory bandwidth, parallelization, synchronization, and vectorization can behave differently depending on the compiler, backend, and target architecture. A natural continuation is therefore to expand this analysis to additional SYCL implementations and newer CPU and accelerator architectures, while integrating the developed microbenchmarks with existing suites such as SYCL-Bench. In particular, vectorization deserves further investigation, since the results in Paper I indicate substantial differences between x86 and AArch64 systems and between Ahead-of-Time and Just-in-Time compilation. A more detailed

analysis of the generated code and compiler cost models could help explain when SYCL abstractions can be mapped efficiently to CPU SIMD execution and when architecture-specific limitations prevent this.

The autotuning framework developed in Paper II also opens several directions for extending this work beyond static, single-objective tuning. Future studies can investigate multi-objective optimization, considering metrics such as execution time, energy consumption, and memory usage simultaneously, as well as dynamic workloads where suitable compiler and runtime configurations may change during execution. Another important direction is to reduce the cost of tuning across heterogeneous systems by exploiting information collected on previously evaluated architectures. Since Paper II shows that optimization decisions do not transfer uniformly between CPUs and GPUs, such methods should be architecture-aware rather than assuming that a single configuration is universally portable. Cross-device performance models could be used to determine which tuning decisions are reusable and which must be rediscovered for each target. Together, these directions move toward a system in which performance portability is supported not only by a common programming model, but also by automated mechanisms that adapt compilation and runtime decisions to the characteristics of the target architecture.

Bibliography

- [1] Khronos SYCL Working Group, “SYCL 2020 specification,” Khronos Group, 2024. [Online]. Available: <https://registry.khronos.org/SYCL/specs/sycl-2020/html/sycl-2020.html>
- [2] A. Alpay and A. Contributors, “AdaptiveCpp: A compiler for heterogeneous programming,” <https://github.com/AdaptiveCpp/AdaptiveCpp>, 2026, gitHub repository, accessed 28 July 2026.
- [3] Intel Corporation, “Intel oneAPI DPC++/C++ Compiler,” <https://www.intel.com/content/www/us/en/developer/tools/oneapi/dpc-compiler.html>, 2026, accessed: 2026-07-29.
- [4] OpenMP Architecture Review Board, “OpenMP application programming interface, version 5.2,” OpenMP Architecture Review Board, Tech. Rep., Nov. 2021. [Online]. Available: <https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5-2.pdf>
- [5] OpenACC Standards Group, “The OpenACC application programming interface, version 2.6,” OpenACC Standards Group, Tech. Rep., 2017. [Online]. Available: <https://www.openacc.org/sites/default/files/inline-files/OpenACC.2.6.final.pdf>
- [6] H. C. 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, no. 12, pp. 3202–3216, 2014.
- [7] D. A. Beckingsale, T. R. W. Scogland, J. Burmark, R. Hornung, H. Jones, W. Killian, A. J. Kunen, O. Pearce, P. Robinson, and B. S. Ryujiin, “RAJA: Portable performance for large-scale scientific applications,” in *Proceedings of the 2019 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*, 2019, pp. 71–81.
- [8] Khronos OpenCL Working Group, “The OpenCL specification, version 3.0,” Khronos Group, 2024. [Online]. Available: https://registry.khronos.org/OpenCL/specs/3.0-unified/html/OpenCL_API.html
- [9] S. J. Pennycook, J. D. Sewall, and V. W. Lee, “A metric for performance portability,” *arXiv preprint arXiv:1611.07409*, 2016.

- [10] I. A. Baratta, C. N. Richardson, and G. N. Wells, “Performance analysis of matrix-free conjugate gradient kernels using SYCL,” in *Proceedings of the 10th International Workshop on OpenCL and SYCL*, ser. IWOCL ’22. Association for Computing Machinery, 2022, pp. 1–10.
- [11] I. Z. Reguly, “Evaluating the performance portability of SYCL across CPUs and GPUs on bandwidth-bound applications,” in *Proceedings of the SC ’23 Workshops of the International Conference on High Performance Computing, Network, Storage, and Analysis*, ser. SC-W ’23. Association for Computing Machinery, 2023, pp. 1038–1047.
- [12] S. Lal, A. Alpay, P. Salzmann, B. Cosenza, A. Hirsch, N. Stawinoga, P. Thoman, T. Fahringer, and V. Heuveline, “SYCL-Bench: A versatile cross-platform benchmark suite for heterogeneous computing,” in *Euro-Par 2020: Parallel Processing*, ser. Lecture Notes in Computer Science, vol. 12247. Springer, 2020, pp. 629–644.
- [13] E. Georganas, R. F. Van Der Wijngaart, and T. G. Mattson, “Design and implementation of a parallel research kernel for assessing dynamic load-balancing capabilities,” in *2016 IEEE International Parallel and Distributed Processing Symposium*. IEEE, 2016, pp. 73–82.
- [14] Z. Jin and J. S. Vetter, “A benchmark suite for improving performance portability of the SYCL programming model,” in *2023 IEEE International Symposium on Performance Analysis of Systems and Software*, ser. ISPASS ’23. IEEE, 2023, pp. 325–327.
- [15] T. Deakin and S. McIntosh-Smith, “Evaluating the performance of HPC-style SYCL applications,” in *Proceedings of the International Workshop on OpenCL*, ser. IWOCL ’20. Association for Computing Machinery, 2020, pp. 1–11.
- [16] M. Breyer, A. Van Craen, and D. Pflüger, “Evaluation of SYCL’s different data parallel kernels,” in *Proceedings of the 12th International Workshop on OpenCL and SYCL*, ser. IWOCL ’24. Association for Computing Machinery, 2024.
- [17] J. Ansel, S. Kamil, K. Veeramachaneni, J. Ragan-Kelley, J. Bosboom, U.-M. O’Reilly, and S. Amarasinghe, “OpenTuner: An extensible framework for program autotuning,” in *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT ’14. Association for Computing Machinery, 2014, pp. 303–316.
- [18] C. Nugteren and V. Codreanu, “CLTune: A generic auto-tuner for OpenCL kernels,” in *2015 IEEE 9th International Symposium on Embedded Multicore/Many-Core Systems-on-Chip*. IEEE, 2015, pp. 195–202.
- [19] B. van Werkhoven, “Kernel tuner: A search-optimizing GPU code auto-tuner,” *Future Generation Computer Systems*, vol. 90, pp. 347–358, 2019.
- [20] Y. Liu, W. M. Sid-Lakhdar, O. Marques, X. Zhu, C. Meng, J. W. Demmel, and X. S. Li, “GPTune: Multitask learning for autotuning exascale applications,” in *Proceedings of the 26th ACM SIGPLAN Symposium on*

- Principles and Practice of Parallel Programming*, ser. PPOPP '21. Association for Computing Machinery, 2021, pp. 234–246.
- [21] L. Scravaglieri, A. Anciaux-Sedrakian, T. Guignon, O. Aumage, and M. Popov, “Compiler, runtime, and hardware parameters design space exploration,” in *2025 IEEE International Parallel and Distributed Processing Symposium*, ser. IPDPS '25. IEEE, 2025, pp. 247–260.
- [22] T. Akiba, S. Sano, T. Yanase, T. Ohta, and M. Koyama, “Optuna: A next-generation hyperparameter optimization framework,” in *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD '19. Association for Computing Machinery, 2019, pp. 2623–2631.
- [23] F. Glover, “Future paths for integer programming and links to artificial intelligence,” *Computers & Operations Research*, vol. 13, no. 5, pp. 533–549, 1986.
- [24] A. E. Hoerl and R. W. Kennard, “Ridge regression: Biased estimation for nonorthogonal problems,” *Technometrics*, vol. 12, no. 1, pp. 55–67, 1970.

Paper I

Exploring SYCL as a portability layer for high-performance computing on CPUs

Hari Abram, Nikela Papadopoulou,
and Miquel Pericàs

Published in
ISC High Performance 2025.

Exploring SYCL as a portability layer for high-performance computing on CPUs

Hari Abram
*Chalmers University of Technology,
University of Gothenburg
Gothenburg, Sweden
hariv@chalmers.se*

Nikela Papadopoulou
*University of Glasgow
Glasgow, U.K.
nikela.papadopoulou@glasgow.ac.uk*

Miquel Pericàs
*Chalmers University of Technology,
University of Gothenburg
Gothenburg, Sweden
miquelp@chalmers.se*

Abstract—As multicore vector processors improve in computational and memory performance, running SIMT (Single Instruction Multiple Threads) programs on CPUs has become increasingly appealing, potentially eliminating the need for dedicated GPU hardware. SYCL is a royalty-free cross-platform C++ programming model for heterogeneous computing that implements the SIMT model and provides a path to run GPU programs on CPU hardware. Several SYCL implementations have been developed. To understand their performance on multicore vector CPUs, this paper systematically evaluates the two major SYCL implementations using a set of micro-benchmarks to assess key features such as memory bandwidth, kernel scheduling, synchronization, and vectorization. We test SYCL implementations across Intel Icelake, AMD Zen2, Fujitsu A64FX, and AWS Graviton3 systems, comparing Ahead-of-Time (AoT) and Just-In-Time (JIT) compilation flows, with OpenCL and OpenMP backends, and compare the results with native OpenMP implementations. Based on the results from these experiments, we generate a table of recommendations for SYCL developers. In general, our results show that AoT outperforms JIT in memory bandwidth, while JIT excels in scheduling and synchronization. SYCL on A64FX achieves near-peak memory bandwidth but underperforms in parallelization and reduction tasks. Our study also explores data management strategies, including USM and buffer & accessor (B&A), and backend configurations, revealing that these factors significantly impact performance. Our findings underscore the importance of selecting the appropriate toolchain to maximize SYCL's potential for portable and high-performance execution across diverse hardware platforms.

Index Terms—Benchmarking, CPUs, SYCL, OpenMP, Intel OneAPI, DPC++, AdaptiveCpp

I. INTRODUCTION

Recent advances in heterogeneous architectures have raised concerns about performance portability and the complexity of application porting. To address these, the Khronos Group developed the SYCL standard [1], which aims to target performance portability across different architectures. SYCL follows an SPMD (single-program, multiple-data) approach similar to OpenCL/CUDA and is primarily designed for GPU-based computing. However, current trends in CPU architectures, such as long vector widths and high memory bandwidth exemplified by processors like the Fujitsu A64FX, are being explored to achieve performance levels comparable to those of GPUs [2]–[6]. These developments underscore the need to analyze the performance of SYCL on CPUs.

The open nature of SYCL allows for various implementations, each relying on different compilation toolchains and runtimes. When targeting CPUs, Intel DPC++ [7] employs JIT compilation with an OpenCL backend. At the same time, AdaptiveCpp [8] can employ either JIT compilation, with OpenCL or OpenMP runtime, or AoT(Ahead-Of-Time) compilation with OpenMP. This differs from GPU-targeted SYCL, which typically relies on vendor-provided runtime systems, such as CUDA with NVIDIA GPUs. Achieving peak performance when executing SYCL code on CPUs is challenging due to the wide variety of available runtimes and compilation methods. This raises the critical question of which toolchain should be selected when targeting specific CPUs.

SYCL can act as a portability layer across GPUs and CPUs, but its performance on multicore vector CPUs needs careful evaluation. Inheriting its design from OpenCL, SYCL is more closely aligned with GPU-like architectures. Therefore, its effectiveness on modern CPUs requires further investigation and analysis. SYCL offers multiple ways to parallelize a kernel offloaded to a runtime, such as utilizing various iteration spaces (`sycl::range` vs `sycl::nd_range`), dimensions, and data management methods (USM (Unified Shared Memory) vs B&A (Buffer and Accessors)). Combining these diverse methods with the various toolchains available in SYCL implementations presents a challenge for developers aiming to achieve acceptable performance in their applications, considering these methods were developed to target GPU-like architectures. Motivation: Given the diverse array of toolchains available for targeting CPUs through SYCL, how can we effectively leverage these toolchains to optimize performance on multicore vector CPUs? This paper characterizes the performance of SYCL methods, focusing on the impact of these methods on vectorization, scheduling, and synchronization.

Quantifying the influence of runtimes and compilation flows on the performance of SYCL features through application-level benchmarks is challenging due to the difficulty of achieving isolation across SYCL features, given the interplay of the various SYCL features used in such benchmarks. Additionally, although similar analyses have been conducted on specific applications [9]–[17], it remains challenging to pinpoint the specific SYCL features that contribute to performance bottlenecks. To address this, we have developed a suite of micro-

benchmarks¹ that enable a bottom-up approach for assessing the performance of SYCL features across various implementations and runtimes on CPUs. These micro-benchmarks complement existing SYCL benchmark suites such as SYCL-Bench [18], HeCBench [19] and PRK [20], by enhancing the scope and depth of the benchmarks. We achieve this through the use of different methodologies and by incorporating more parametrization, allowing for a more comprehensive evaluation of SYCL performance across various configurations.

Our microbenchmarks evaluate the impact of runtime on memory bandwidth, parallelization, vectorization, and other factors on four processors with different ISAs: Intel Icelake, AMD Zen2, Fujitsu A64FX, and AWS Graviton3. These four processors are selected to showcase different vectorization and parallelization capabilities. We also compare the performance of SYCL against OpenMP by developing equivalent OpenMP benchmarks, which serve as our baseline, considering OpenMP is a standard parallel programming model for CPUs. Note that, unlike SYCL, OpenMP requires specific constructs to address GPUs, compared to the standard method when targeting CPUs, this requires an extra effort to make OpenMP single-source portable, though this effort may be minimal. Our analysis aims to identify key SYCL features and the impact of runtime and backend choices on performance, to achieve portable and acceptable performance across systems.

With our microbenchmarks, we observe the performance difference across SYCL implementation versions, compilation flows and backends while testing for performance portability across systems. Specifically, we observe that JIT with OpenCL is preferred when data movement between main memory and the processor is minimal, whereas AoT with OpenMP is preferred when there is a need for high memory bandwidth. In addition, we show that SYCL on x86 machines outperforms AArch64 systems in scheduling and synchronization tasks. When SYCL is compared to OpenMP, performance superiority varies by system and SYCL implementation, with some JIT workflows outperforming OpenMP, particularly in parallelization, where JIT compilation performs up to 10% better than OpenMP. More detailed findings are presented in Table VIII. Overall, our experiments suggest that SYCL performance on multi-core vector CPUs is influenced by factors such as compilation flows, backend choice, and system-specific hardware features.

This paper makes the following contributions:

- We systematically evaluate the various runtimes and compilation flow toolchains of SYCL implementations when targeting modern multicore CPUs with vectorization capabilities.
- We assess the potential of SYCL for portability across different CPU architectures by evaluating its utilization of available features on various hardware platforms.
- We show that the achieved memory bandwidth is strongly influenced by the compilation flow, with AoT compilation achieving the highest possible bandwidth across all

systems. In contrast, parallelization shows better performance with JIT compilation. Additionally, our findings reveal that, on certain systems, the cost model plays a significant role in the vectorization of the kernel.

The remainder of this paper is organized as follows. We discuss related work in Section II. Section III outlines our SYCL micro-benchmarks. Section IV provides details on the experimental configuration. In Section V, we present our experimental results for the developed micro-benchmarks and discuss our findings. We conclude this work in Section VI.

II. RELATED WORK

With the rising popularity of SYCL, several works have used SYCL as the programming model of their choice [9]–[16], which has led to the development of several SYCL benchmark suites [18], [20]–[22]. The SYCL-Bench suite [18], [21] offers valuable insights into the performance of SYCL. It has certain similarities to our work, particularly in trying to characterize the SYCL runtime effects, focusing on task scheduling and data management. However, the primary target of this research work is GPUs, with benchmarks specifically designed to characterize GPU runtimes and architectures. Our work aims to target multicore vector-capable CPUs by focusing on the specific aspects of SYCL performance. Still, the main reason for developing our microbenchmarks rather than using SYCL-bench is due to the degree of freedom available and the in-depth analysis we can achieve, compared to using other benchmark suites. For example, in SYCL-bench, each kernel is predominantly executed using a single method, whereas, in our micro-benchmark suite, we test multiple methods to initiate the execution of kernel for each of the micro-benchmarks.

Several research works also have benchmark suites to assess SYCL performance in the heterogeneous landscape. The PRK benchmarks [20], [22] comprise three benchmarks implemented in SYCL: stream, 2D stencil, and point-to-point synchronization. While these benchmarks are useful, they have limitations in fully characterizing SYCL implementations and their runtime behavior. The Rodinia Benchmarks [17] have also been ported to SYCL. The complexity of these benchmarks imposes constraints on characterizing the runtime effects of SYCL features. Moreover, the intricacy makes it challenging to study the effects of SYCL constructs in isolation. HeCBench [19] has been developed to test the performance portability of SYCL at the application level. These three benchmark suites are primarily designed to test the SYCL performance at the application level.

Recent work by [23] investigates the offloading of the kernel matrix assembly in PLSSVM (Parallel Least Squares Support Vector Machine) using various SYCL methods. This research shares a similar objective to our study, but differs in its approach to analyzing performance overheads. While the paper focus on the overall performance of kernels in different kernel offloading methods, our study measures the overheads incurred during the scheduling of parallel tasks using various offloading methods. This allows us to analyze task scheduling overhead independently, without the influence of kernel computation

¹<https://github.com/HariAbram/microSYCL.git>

overhead. Several research works have predominantly focused on the process of porting applications to SYCL, documenting the experiences and challenges encountered. Examples of previous research work with SYCL include the conjugate gradient [9], molecular dynamics applications [12], stellar merges simulations [13], epistasis detection [14], LU decomposition [15], cluster methods [10], bandwidth-bound applications [24], Fourier transform library [16] and AI graphs [11]. In recent work, Joubert et al. [25] implement diverse memory management strategies within SYCL to evaluate their effectiveness, this approach aims to provide a more comprehensive understanding of the intricacies surrounding the performance implications of employing these features in SYCL-based applications.

III. METHODOLOGY

This work focuses primarily on the performance characterization of various SYCL features through the utilization of microbenchmarks. To effectively understand the impact of compilation toolchains and runtimes on the various parameters through which SYCL kernels are executed on CPUs, flexibility and parametrization are essential when offloading these microkernels. This need for flexibility led to the development of our suite of microbenchmarks. The inspiration for this approach comes from the EPCC benchmark suite [26], a well-established collection of OpenMP microbenchmarks designed to evaluate the performance of parallel constructs and features in OpenMP. Within our micro-benchmark suite, we analyze a range of SYCL features, including memory bandwidth of Unified Shared Memory (USM) allocations, parallel constructs employing `range` and `nd_range` objects, various SYCL data management techniques, accumulation via atomic operations, reduction, as well as group and subgroup barriers. Additionally, our suite includes microbenchmarks to evaluate the impact of vectorization on SYCL kernels, when various SYCL features are combined. Each microbenchmark is executed 100 times, with the resulting average, median, maximum, and minimum execution times used for performance comparison. It is important to note that all variables considered in this micro-benchmark study are of type `double`. Although the experiments were also conducted using the `float` type, similar trends were observed for both types.

In the following subsections, we provide detailed descriptions of each of these features and the corresponding microbenchmarks that we have developed for their evaluation. Furthermore, we outline our definition of the baseline performance for each feature, also explaining how we obtain the baseline, to effectively measure any additional overhead introduced by the SYCL software stack.

A. Microbenchmarks for Evaluating SYCL Features

1) *Memory bandwidth*: The SYCL standard encompasses three distinct methods for USM data allocation, namely, `sycl::malloc_host`, `sycl::malloc_shared`, and `sycl::malloc_device`. The first method allocates shared data and pins it to the host memory, facilitating access by both

the host and the device, while the second method allocates a shared memory region that is similarly accessible by both the host and the device, but pins it to the device. The third method allocates device-specific memory, which can only be accessed by the device, and is pinned to the device. In our microbenchmark, we evaluate the memory bandwidth observed when employing these different memory allocation methods. Furthermore, we examine the memory bandwidth performance when using the `range` and `nd_range` iteration spaces. These different data management strategies in SYCL are used to target heterogeneous computing systems, where data has to be transferred from CPU to accelerator or GPU, it is important to note that since the experiments within our work execute their kernels on a CPU, the host and device are equivalent from the implementation perspective. We then compare this memory bandwidth with the bandwidth achieved with the BabelStream [27] benchmark, which uses OpenMP for parallelization.

2) *Parallelization*: The `parallel_for` method is a principal parallelization construct in SYCL that serves as the means to delegate computational tasks to the device. It accepts two key inputs: a kernel, represented in the form of a lambda function, and the iteration space defined using `range` and `nd_range` objects. The method proceeds to generate a distinct kernel instance for every iteration within the specified iteration space, where the kernel instance corresponds to the lambda function that is executed on individual threads created by the runtime.

Kernel instances generated with the `range` template are entirely independent of each other, and there is no communication or synchronization between these distinct instances. On the other hand, the `nd_range` class template introduces a concept of global and local range. The global range is subdivided into subgroups defined by the local range. Kernel instances associated with the local range possess shared memory, facilitating communication and synchronization of computation among these specific instances.

SYCL offers two distinct methods for offloading a kernel to a device, depending on the data management strategy. The first method involves the use of buffers and accessors, where data is initially transferred to a `sycl::buffer` and subsequently accessed through a `sycl::accessor` within the `sycl::handler`. The second approach involves Unified Shared Memory (USM); here, global variables created through the SYCL shared context can be accessed within the `sycl::handler`, unlike variables created with general memory allocations. The general approach to offloading a kernel is demonstrated in Listing 1.

The microbenchmark designed for this feature assesses the synchronization and scheduling overheads of the SYCL backend runtime. This evaluation encompasses a range of iteration spaces, each characterized by varying dimensions. Furthermore, the benchmark serves to explore the influence of both the problem size and the iteration space dimensions on performance. These parameters are determined when declaring a `range` or `nd_range` type, denoted as `range<dim>`.

```

sycl::queue Q;
sycl::buffer<TYPE, 1> buff(var, 1);
Q.submit([&](sycl::handler& cgh) {
    sycl::accessor acc = buff.get_access
        <sycl::access::mode::read_write>(cgh);
    cgh.parallel_for<>(sycl::range<1>
        (global), [=](sycl::item<1> it) {

        ///computation

    });
});

```

Listing 1: Kernel offloading for B&A data management

The aim is to scrutinize how different problem sizes and dimension choices impact the overall performance of the kernel offloading process.

To conduct a comparison between a SYCL kernel offloaded with a two-dimensional iteration space and its OpenMP counterpart, we employ the `omp parallel for collapse(2)` construct within an equivalent OpenMP microbenchmark. This approach allows for a direct comparison between SYCL and OpenMP in scenarios involving two-dimensional iteration spaces [8].

```

for(size_t i = 0; i < 1024; i++) {
    if (sum[thread_id] < 0)
    {
        break;
    }
    sum[thread_id] += 1;
}

```

Listing 2: Kernel computation

The microbenchmark for parallelization serves a twofold purpose: firstly, to identify the most effective combination of features for achieving parallelization, and secondly, to investigate whether this optimal combination remains consistent across different vendors, SYCL implementations, and backend runtimes. The kernel computation that is offloaded in the microbenchmark is outlined in Listing 2, and the `thread_id` is obtained with `sycl::item.get_thread_id(0)`. To accurately measure the overhead associated with parallelization, the computational tasks executed by each thread are kept minimal to avoid directly influencing the benchmark results. The variable `sum` is accessed either using SYCL’s unified shared memory (USM) or via a buffer and accessor, depending on whether the kernel is offloaded in a shared context or not. This approach allows us to distinctly quantify the influence of various parallelization techniques on performance.

3) *Atomics and Reduction*: To assess the performance of atomic operations across various SYCL implementations and to compare them against OpenMP atomic constructs, we introduce a sum accumulation kernel, employing atomics to perform the accumulation of an array into a variable.

For the SYCL atomics microbenchmark, we employ different data management strategies. An array is declared and managed either through USM or using a buffer and accessors. The accumulation operation (summation) is carried out on this array within the kernel. The kernel is offloaded to the device and the array is accessed in the kernel, depending on

whether the array is managed through a buffer and accessors or USM. The SYCL kernel offloaded in this microbenchmark is presented in Listing 3.

```

auto j = it.get_id(0);
auto v = sycl::atomic_ref<TYPE,
    sycl::memory_order::seq_cst,
    sycl::memory_scope::device,
    sycl::access::address_space
    ::global_space>(
    sum[0]);
v.fetch_add(m[j]);

```

Listing 3: Atomics implementation

We additionally implement reduction using the `sycl::reduction` method. We highlight that the `sycl::reduction` syntax differs for AdaptiveCpp and DPC++ when utilizing Buffer and Accessor data management, as presented in Listing 4. This is a result of changes in syntax across different versions of the SYCL standard. Between these two SYCL implementations, DPC++ conforms to the latest standard.

```

auto m_acc = m_buff.get_access<
    sycl::access::mode
    ::read>(cgh);
#ifdef (AdaptiveCpp)
auto sum_acc = sum_buff.get_access<
    sycl::access::mode
    ::read_write>(cgh);
auto sum_red = sycl::reduction(sum_acc,
    sycl::plus<TYPE>());
#else if defined (DPC++)
auto sum_red = sycl::reduction(sum_buff, cgh,
    sycl::plus<TYPE>());
#endif
cgh.parallel_for(sycl::
    nd_range<1>(global, local),
    sum_red, [=](sycl::
    nd_item<1>it, auto &sum) {

    auto j = it.get_global_id(0);
    sum += m_acc[j];
});

```

Listing 4: SYCL reduction implementation

For comparison purposes, we implement an OpenMP atomics microbenchmark that uses an `atomic` construct. We also introduce an OpenMP benchmark that utilizes a `reduction` construct. It is worth noting that the OpenMP `reduction` construct is known for offering high performance [28] when conducting reduction operations.

4) *Barriers*: In the SYCL programming model, barriers can be performed at the group level or the subgroup level. In our microbenchmark, we evaluate these two different barriers, both with USM and with buffer and accessors, but also with different dimensions. We compare the performance of these barriers with the OpenMP barriers.

B. Vectorization

This study extends its analysis to examine the vectorization capabilities of the various SYCL implementations across different machines. Additionally, we investigate the compiler’s auto-vectorization capabilities when using different iteration

spaces, such as `sycl::range` and `sycl::nd_range`, and assess the effectiveness of employing different data management strategies. The aim is to gain comprehensive insights into how these factors impact the overall auto-vectorization performance of SYCL implementations across different hardware and SYCL constructs.

To thoroughly assess the vectorization properties, we introduce four microbenchmarks in our set: GEMM (general matrix multiplication), GEMV (general matrix-vector multiplication), outer product, and cross-product. In these microbenchmarks, we focus on evaluating the auto-vectorization capabilities of SYCL implementations. The outer product benchmark analyzes vectorization across work items, while the GEMM kernel focuses on the vectorization of loops within work items for a 2-dimensional iteration space. Similarly, GEMV tests vectorization for a 1-dimensional iteration space. Additionally, we include a cross-product kernel that implements conditionals inside the kernel to analyze their impact on vectorization. Listings 5, 6, 7, and 8 present the kernels for outer-product, GEMM, GEMV, and cross-product, respectively. We also test the `sycl::reduction` construct, introduced in Listing 4, to analyse the capabilities of SYCL implementations to vectorize the standard SYCL synchronization constructs. We additionally compare this with the OpenMP reduction clause.

```
cgh.parallel_for<>(sycl::nd_range<2>(global1,local1),
    [=](sycl::nd_item<2>it){

    auto i = it.get_global_id(0);
    auto j = it.get_global_id(1);
    auto N = it.get_global_range(0);
    m1_acc[i*N+j] = v1_acc[i]*v2_acc[j];
});
```

Listing 5: Outer product Kernel

```
cgh.parallel_for<>(sycl::nd_range<2>(global1,local1),
    [=](sycl::nd_item<2>it){

    auto i = it.get_global_id(0);
    auto j = it.get_global_id(1);
    TYPE temp = 0.0;
    for (size_t k = 0; k < N; k++)
    {
        temp += m2_acc[i*N+k]*m1_acc[k*N+j];
    }
    m3_acc[i*N+j] = temp;
});
```

Listing 6: GEMM Kernel

```
cgh.parallel_for<>(sycl::nd_range<1>(global,local),
    [=](sycl::nd_item<1>it){

    const int i = it.get_global_id(0);
    const int N = it.get_global_range(0);
    TYPE temp = 0.0;
    for (size_t k = 0; k < N; k++)
    {
        temp += m1_acc[i*N+k] * v1_acc[k];
    }
    v2_acc[i] = temp;
});
```

Listing 7: GEMV Kernel

```
cgh.parallel_for<>(sycl::nd_range<2>(global1,local1),
    [=](sycl::nd_item<2>it){

    auto i = it.get_global_id(0);
    auto j = it.get_global_id(1);
    auto N = it.get_global_range(0);
    auto tmp= std::pow(-1, i+j);
    if (i == j)
    {
        m1_acc[i*N+j] = 0;
    }
    else
    {
        m1_acc[i*N+j] = v1_acc[i]*v2_acc[j]*tmp;
    }
});
```

Listing 8: Cross product Kernel

It is essential to highlight that all arrays and pointers are initialized with the `__restrict__` keyword. This keyword serves as an indication to the compiler that the memory locations of these arrays are independent, which is a critical consideration for the accurate evaluation of auto-vectorization properties. In the vectorization tests, we consider a benchmark to be vectorized if the vector instruction counters show that all cores have executed vector instructions and that the counts are comparable to the average number of vector instructions executed, calculated across all cores. This ensures that the observed vectorization is not a result of computations unrelated to the offloaded kernel but instead has a meaningful impact on performance. We also analyze the compiler reports with `-Rpass-analysis` flags, for a more thorough summary of auto-vectorization. While reporting vectorization results, we focus on whether the systems execute vector instructions and the type of vector instructions.

IV. EXPERIMENTAL SETUP

This work focuses primarily on assessing the performance of SYCL features when executed on CPUs. To this end, we select four CPUs for our experiments, an Intel Xeon Platinum 8358, an AMD EPYC™ 7742, a Fujitsu A64FX, and an AWS Graviton3. We present the details of the experimental platforms in Table I.

In this study, we use two distinct SYCL implementations, namely AdaptiveCpp and DPC++. Table II provides an overview of the SYCL implementations with different compilation flows and backends. Notably, the AdaptiveCpp implementation is available with two different compilation flows, JIT compilation and AoT compilation, each with distinct backends.

Table III presents the list of experiments conducted in this study, which were carefully selected from a larger exploration space. In the following paragraphs, we will briefly explain the setup and tools used for each experiment.

To analyze historical performance trends, we opt to use stable versions of each SYCL implementation from 2022, 2023, and 2024, conducting the comparison on x86 machines. For older versions (v2022) of AdaptiveCpp, we had to use LLVM v14.0.0 for compilation, as older versions of AdaptiveCpp

TABLE I
PLATFORMS

Machine	Architecture	Cores per Node	Threads per Core	NUMA nodes per CPU	Max Frequency (MHz)	L1 Cache	L2 Cache	L3 Cache	SIMD Width
Fujitsu A64FX	ARM-SVE 64-bit	48	1	4	2200	3 MB	32 MB	N/A	512 bit
AMD EPYC Zen2	x86-64	128	2	2	2250	4 MB	64 MB	512 MB	256 bit
Intel Xeon Platinum	x86-64	32	1	2	3700	32 KB	1 MB	22 MB	512 bit
AWS Graviton3	ARM-SVE 64-bit	64	1	1	2300	80 KB	8 MB	N/A	256 bit

TABLE II
SYCL IMPLEMENTATIONS COMPILATION FLOWS

SYCL Implementation	Compilation	Backend
AdaptiveCpp	JIT	OpenCL; OpenMP
	AoT	OpenMP
DPC++	JIT	OpenCL

depend on obsolete syntax that is no longer used in newer versions of LLVM, while newer versions of AdaptiveCpp were compiled with LLVM v18.1.8. For DPC++, we leverage the available versions provided by the package manager, specifically for this study, we utilized v2022.4 v2023.4 and v2024.2.

TABLE III
EXPERIMENTS CONDUCTED IN THIS STUDY

Experiments conducted in this study	Machines
Historical comparison across different versions for AdaptiveCpp and DPC++	Intel Icelake, AMD Zen2
Backend comparison for OpenMP and OpenCL runtimes with AdaptiveCpp JIT and AoT compilation flows.	Intel Icelake
Portability comparison of AdaptiveCpp on the systems mentioned in Table I	All the machines
Vectorization tests across the systems and SYCL implementations, for different backend and compilation flows.	All the machines

We analyze the overheads associated with different compilation workflows and backends on a single system, specifically an Intel Xeon Platinum machine based on the Icelake microarchitecture. For this analysis, we utilize the latest versions of AdaptiveCpp, compiled with LLVM v18.1.8.

To compare SYCL performance across different machines, we utilize AdaptiveCpp, as this implementation functions seamlessly across all the systems. On Fujitsu A64FX, it has been shown in the Fujitsu compiler package documentation [29] that the XOS_MMM_L_PAGING_POLICY environment variable has a significant impact on C++ code performance on A64FX. However, this variable affects only the Fujitsu compilers, which are not employed by any of the SYCL implementations. We therefore do not set this environment variable in our experiments.

We use LLVM to compile OpenMP code, which uses the respective LLVM OpenMP runtime with v18.1.8. In all experiments, we set the environment variable OMP_PROC_BIND to true, as this results in better performance for both the AdaptiveCpp (when offloaded to the OpenMP runtime) and OpenMP code.

It is important to acknowledge that the systems examined

in this study feature different numbers of cores per node. In our work, we restrict our experiments to the physical cores and do not utilize Simultaneous Multi-Threading (SMT).

V. EXPERIMENTAL RESULTS

Our analysis systematically compares the experimental results obtained with various SYCL implementations, considering that these implementations may leverage different compilation flows and backend runtimes.

Our evaluation assesses the performance of SYCL features in relation to their dependency on the backend runtimes. We evaluate each micro-benchmark with a range of problem sizes, to determine whether the performance indicators scale with increasing problem sizes. The term "problem size" varies for each micro-benchmark and may refer to parameters such as array size, the number of instances of the kernel executed, or a combination of both. We compare the different SYCL versions and implementations in terms of execution time. To compare with OpenMP, we normalize the SYCL results against the OpenMP results. We omitted confidence intervals, as the difference between mean, min and max is insignificant. On average, we see a difference of $< 1\%$, $< 5\%$ and $< 1\%$ between min, max and median, compared to the mean, respectively, across benchmarks.

In the following subsections, we first present the historical comparison of the AdaptiveCpp and DPC++ on x86 machines. Next, we compare different backends with AdaptiveCpp on a single machine. Subsequently, we evaluate the performance of AdaptiveCpp on different machines. Finally, we examine the vectorization properties of the various SYCL implementations and backends.

A. Historical Comparison

For the historical comparison of SYCL versions, we present the compilation flows used for each benchmark and version in Table IV. For memory copy operations in the latest versions of AdaptiveCpp, we employed Ahead-Of-Time (AoT) compilation with OpenMP runtime, as this compilation flow achieves higher bandwidth.

TABLE IV
BENCHMARK COMPILATION FLOWS

Microbenchmark	AdaptiveCpp			DPC++
	v22	v23	v24	all versions
Memory bandwidth	AoT	AoT	AoT	JIT
Parallelization	AoT	JIT	JIT	JIT
Barriers	AoT	JIT	JIT	JIT

Figure 1, presents the results for the various versions of SYCL implementations, specifically AdaptiveCpp (ACpp) and DPC++. For comparison, we also include the results of BabelStream, which uses OpenMP, compiled with Clang++ v18.1.8. Our observations indicate that AdaptiveCpp achieves higher bandwidth compared to DPC++, and its performance is closer to the bandwidth achieved by the BabelStream OpenMP benchmark. The difference in the maximum bandwidth achieved by the two implementations is related to runtimes, with AdaptiveCpp utilizing OpenMP runtime and DPC++ employing OpenCL runtime in this benchmark.

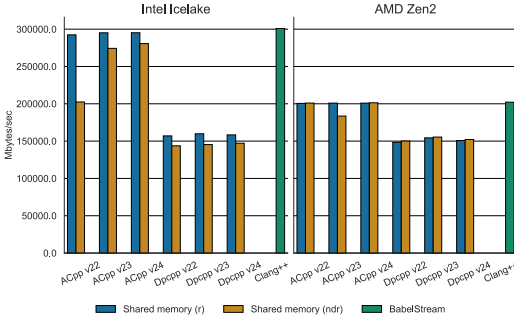


Fig. 1. Memory bandwidth across various versions, for a problem size of 2GB. ndr and r in parenthesis correspond to nd_range and range, respectively.

Figure 2, presents the results for range and nd_range, normalized against OpenMP results. For this micro-benchmark, the problem size corresponds to the size of the iteration space. We observe that DPC++ performance consistently improves year over year on both systems. Notably, on Intel systems, DPC++ demonstrates twice the performance of AdaptiveCpp for one-dimensional iteration spaces. However, our experiments indicate that, for two-dimensional iteration spaces, the performance of DPC++ and AdaptiveCpp is similarly poor. We omit these results for the sake of brevity.

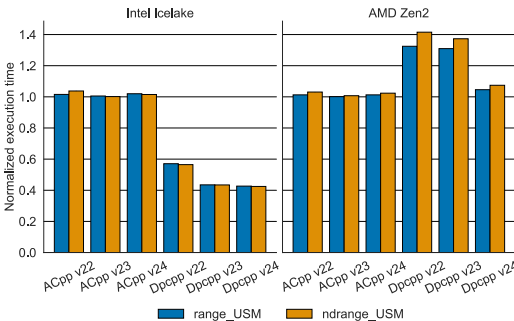


Fig. 2. Parallelization results normalized against OpenMP for a problem size of 2^{20}

Figure 3 presents the results for the barriers at group and subgroup levels, normalized against OpenMP results, compiled with Clang++ (v18.1.8). As in the parallelization benchmark, the problem size corresponds to the size of the iteration space. Overall, there is no significant difference in performance across the different SYCL versions. However, we observe that in AdaptiveCpp v23, group barrier performance is noticeably worse than subgroup barrier performance. This discrepancy coincides with the introduction of the JIT compilation in that version.

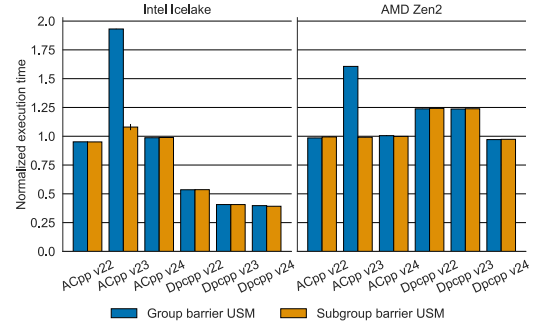


Fig. 3. Barriers results normalized against OpenMP for a problem size of 2^{20}

B. Backend Comparison

Comparing different backends used for offloading SYCL kernels provides insights into the overheads associated with each backend and compilation flow. Since AdaptiveCpp can offload to various backends via its generic JIT compilation flow, we utilized AdaptiveCpp to perform a backend comparison on the Intel Icelake machine. In Figures 4, 5, 6, 7, *JIT ocl* and *JIT omp* correspond to Just-In-Time compiled SYCL code offloaded to OpenCL and OpenMP respectively, and similarly, *AoT omp*, corresponds to Ahead-of-Time compiled SYCL code offloaded to OpenMP.

Figure 4 presents the memory bandwidth results for the various backends and compilation flows when the memory operations are performed on shared memory variables. We observe that the AoT compilation flow with the OpenMP backend achieves the highest bandwidth among the different backends compared. When comparing the bandwidth obtained by DPC++ with the OpenCL backend in Figure 1 to that of the generic OpenCL backend of AdaptiveCpp, the results are similar. This suggests that the OpenCL backend incurs significant overhead during memory operations.

Through memory counters in LIKWID, we observe that with *JIT ocl*, the TLB load and store misses were 3 and 3.7 times higher, and total loads and stores 13 and 15 times higher, respectively, compared to *AoT omp*. Similarly, cycle stalls due to memory operations during the execution of *JIT ocl* code were 18.5 times higher than those observed with *AoT omp* code. Although these findings suggest potential reasons for

the differences between *JIT ocl* and *AoT omp*, no such stark contrasts were observed between *JIT omp* and *AoT omp*. This indicates that the memory model used in OpenCL [30] may be a key factor contributing to the poorer performance observed with *JIT ocl* and *AoT omp*.

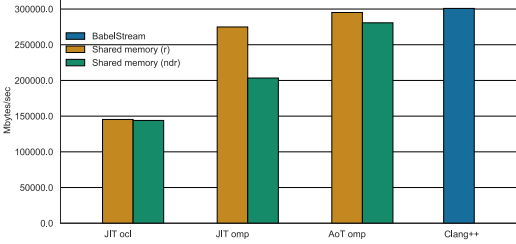


Fig. 4. Memory bandwidth across various backends, for a problem size of 2GB, on Intel Icelake machine.

We also examine the memory bandwidth when using the `range` and `nd_range` iteration spaces, where memory operations using the `range` iteration space achieve higher bandwidth. We have additionally compared the memory bandwidth with the memory allocated on the host or the device, observing similar bandwidth levels to those achieved with shared memory.

Figure 5 presents the results for our parallelization micro-benchmark, using the `range` feature with USM and Buffer & Accessor (BA) data management, normalized against OpenMP results. The results for `nd_range` are similar to those for `range`. For this micro-benchmark, the problem size corresponds to the size of the iteration space. We analyze various problem sizes, to understand if the performance scales with increasing problem sizes. We observe that the OpenCL backend is up to 10% more performant than the OpenMP backend on the Intel Icelake architecture, though, in the case of small problem size (2^{16}), this is the opposite. Furthermore, the performance of *JIT omp* and OpenMP are equivalent, and between USM and B&A we don't observe significant differences as there is very little data movement in this benchmark. We also note that *AoT omp* results are similar to *JIT omp*, though these results are not present in the figure.

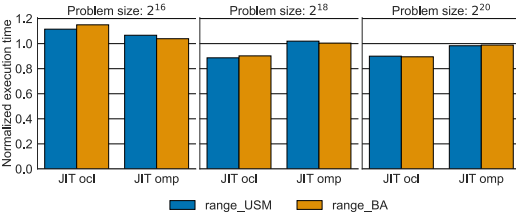


Fig. 5. Parallelization results normalized against OpenMP, with USM and BA data management, on Intel Icelake machine.

Figure 6 presents the atomics results for different backends and data management strategies normalized against OpenMP execution time. In this context, the problem size refers to the array size that is summed using atomic operations. When comparing different backends, we observe that the performance is generally similar for both OpenCL and OpenMP.

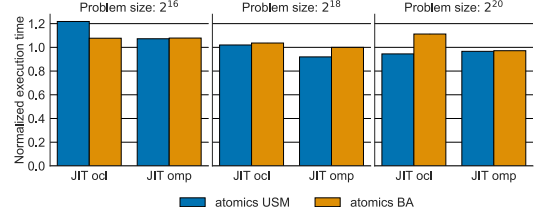


Fig. 6. Atomics results normalized with OpenMP, with USM and BA data management, on Intel Icelake machine.

Figure 7 presents the results for the reduction micro-benchmark with different backends, normalized to OpenMP execution time. We observe that the SYCL overhead compared to OpenMP is substantially higher, with USM-managed reductions performing up to 5 times worse than OpenMP reductions. More in-depth analysis of this SYCL feature through hardware counters shows that USM and B&A have a higher instruction count than OpenMP in all the counters we observed, such as cache misses, cache hits, TLB load and store misses, and cycle stalls. Analyzing the traces of these kernels, we saw that SYCL kernels spend the predominant amount of time on `__sched_yield()` function compared to OpenMP, corresponding to more time spent on resource contention. Another observation is that, for larger problem sizes, the performance difference between OpenMP and SYCL (both USM and B&A) significantly decreases.

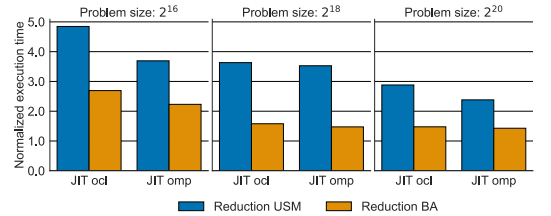


Fig. 7. Reduction results normalized against OpenMP, with USM and BA data management, on Intel Icelake machine.

C. Portability comparison

In this subsection, we present results for the micro-benchmarks on various platforms, the specifications of which are presented in Table I. These experiments focus on understanding the performance portability of various SYCL features across machines and architectures. To ensure portability across

the platforms, we utilize the AdaptiveCpp SYCL implementation.

Figure 8 presents the results for memory bandwidth on various systems. We observe that on the A64FX system, SYCL achieves bandwidth comparable to that of the OpenMP BabelStream benchmark, which we use as a yardstick for comparison. However, on other machines, there is a slight decrease in the bandwidth obtained by SYCL compared to OpenMP. We note that, on the A64FX processor, the maximum achievable bandwidth is 830 GB/s [31], but this performance can only be reached using `zfill` and software prefetch optimizations, which are available exclusively with the Fujitsu compiler.

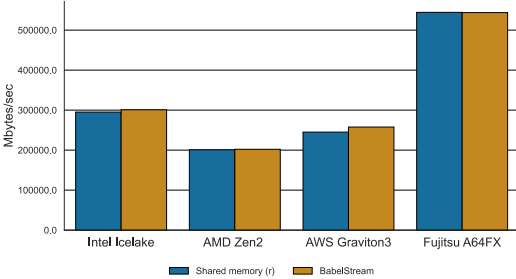


Fig. 8. Memory bandwidth across various machines, for a problem size of 2GB.

Figure 9 presents the results for the parallelization, with USM data management and `sycl::range` iteration space, atomics, and reduction benchmarks. In the parallelization micro-benchmark, AdaptiveCpp on A64FX performs worse among all the machines tested. It is well-known that A64FX performance is highly dependent on the compiler used, with the Fujitsu compiler achieving significantly better performance compared to Clang or GCC [31]. However, none of the SYCL compilers we are aware of are based on the Fujitsu compiler. AArch64 machines in general perform worse than x86 machines in the parallelization benchmark.

Similarly, we observe that the performance of atomics is 60 times worse on A64FX compared to other machines tested. This issue seems to be related to the CAS or LL-SC instructions used in A64FX, as discussed in [32]. Comparing reduction on various machines shows that Intel and Graviton3 systems perform best out of all the machines.

We summarize our observations from our micro-benchmark study across systems, backends, and compilation flows in Table VIII.

D. Vectorization

In this subsection, we present the results of the benchmarks that test for the auto-vectorization of the kernels across the work items and inside the work items. The benchmarks we consider are detailed in Section III.

Table V presents the compilation flags used for both SYCL implementations across all systems in this study. The `-Ofast`

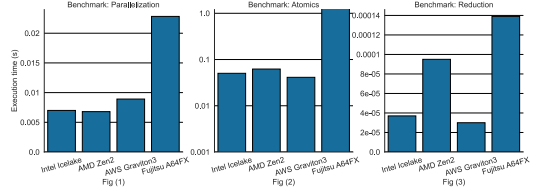


Fig. 9. Results across various systems for different benchmarks with a problem size of 2^{20}

flag is crucial for vectorizing the GEMM kernel with the AdaptiveCpp SYCL implementation. On the A64FX system, vectorization is not achieved without the `-mcpu=a64fx+sve` flag.

TABLE V
COMPILER FLAGS

Machines	Compiler flags
Fujitsu A64FX	-Ofast -mcpu=a64fx+sve
AMD EPYC Zen2	-Ofast -march=native
Intel Xeon Platinum	-Ofast -march=native
AWS Graviton3	-Ofast -mcpu=neoverse-v1

Table VII presents the vectorization results. The hardware counters used to analyze vector instructions differ across machines, and these differences will be discussed during the explanation of each machine. We used the LIKWID performance tool with `perf` events to get the instruction count from each core. The hardware counters used in this work are presented in Table VI.

TABLE VI
HARDWARE COUNTERS USED IN THIS STUDY

Machine	Hardware counters
Intel Icelake	<i>fp_arith_inst_retired_128B_packed_double</i> <i>fp_arith_inst_retired_256B_packed_double</i> <i>fp_arith_inst_retired_512B_packed_double</i>
AMD Zen2	<i>retired_sse_avx_flops_add_sub</i> <i>retired_sse_avx_flops_mult</i> <i>retired_sse_avx_flops_fma</i>
AWS Graviton3	<i>sve_inst_spec</i>
Fujitsu A64FX	<i>simd_inst_retired</i> <i>sve_inst_retired</i>

On the Intel machine, we have counters to capture 128b, 256b, and 512b vector instruction. Table VII shows that for all benchmarks using the OpenMP backend, only 128b and 256b vector instructions are retired. In contrast, when using the OpenCL backend, 512b vector instructions are also executed. This observation holds for both AdaptiveCpp and DPC++ implementations.

On the AMD machine, we have counters to capture add-sub, mult and fma vector instructions, which correspond to addition-subtraction, multiplication, and fused-

TABLE VII
VECTORIZATION RESULTS FOR THE BENCHMARKS, DIFFERENT MACHINES HAVE DIFFERENT COUNTERS TO MEASURE THE VECTOR INSTRUCTIONS, DETAILED IN TABLE VI. HERE `ocl` CORRESPONDS TO THE OPENCL BACKEND. `✗` CORRESPONDS TO NO VECTOR INSTRUCTIONS.

		Intel Icelake				AMD Zen2				A64FX		Graviton3	
		ACPP		DPC++		ACPP		DPC++		ACPP		ACPP	
		JIT	AoT	JIT	AoT	JIT	AoT	JIT	AoT	JIT	AoT	JIT	AoT
Benchmarks	omp	ocl	omp	ocl	omp	ocl	omp	ocl	omp	omp	omp	omp	omp
mat-vec	128b; 256b	512b	128b; 256b	128b; 512b	add-sub; fma	add-sub; mult	fma	simd; sve	simd; sve			✗	sve
mat-mul	128b; 256b	512b	128b; 256b	128b; 512b	add-sub; fma	add-sub; mult	fma	simd; sve	simd; sve			✗	✗
outer-product	256b	128b; 512b	256b	128b; 512b	mult	mult	mult	simd; sve	simd; sve			✗	✗
cross-product	256b	128b; 512b	256b	128b; 512b	add-sub; mult; fma	add-sub; mult; fma	add-sub; mult; fma	✗	✗			✗	✗
sycl::reduction	✗												

TABLE VIII
KEY FINDINGS

	SYCL		
	Across Systems	Across Backends	With different Compilation flows
Memory bandwidth	On A64FX we achieve bandwidth closer to BabelStream benchmark.	OpenMP backend achieves higher bandwidth compared to OpenCL	AoT achieves higher bandwidth compared to JIT compilation flow.
Parallelization	Performance on x86 systems is better than AArch64 systems.	OpenCL performs slightly better than OpenMP	JIT compilation is better than AoT
Atomics	We observe that A64FX has the worst performance across all systems.	OpenMP performs slightly better than OpenCL	Negligible difference across different compilation flows
Reduction	A64FX and AMD systems have inferior performance compared to Intel and Graviton3		
Barriers	There is a negligible difference across the systems, backends and compilation flows		
Vectorization	x86 systems perform vectorization better than AArch64 systems.	OpenCL and OpenMP perform comparably concerning vectorization.	JIT can perform complex vector instructions, such as <code>fma</code> compared to AoT.
	<code>sycl::reduction</code> feature is not vectorized by any of the toolchains on all the systems		

multiply-add instructions, respectively. We observe that with AdaptiveCpp, AoT compilation prefers `mult` instructions compared to `fma` instructions. In contrast, JIT compilation, with both AdaptiveCpp and DPC++ generally favors `fma` instructions over `mult` and `add-sub`.

On the A64FX system, we have counters to capture SIMD and SVE vector instructions. Generally, for both compilation flows, we observe the use of SIMD and SVE instructions. However, for the cross-product benchmark, no vector instructions are generated with either JIT or AoT compilation. This particular benchmark includes an `ifelse` control flow within the kernel, suggesting that the AdaptiveCpp compiler on A64FX may not be sophisticated enough to effectively compile kernels containing `ifelse` conditionals.

On the Graviton3 system, we have counters to capture SVE vector instructions. Generally, we observe that with the JIT compilation, none of the benchmarks are vectorized, while with the AoT compilation, only the GEMV and outer-product benchmarks are vectorized. The poor vectorization performance of AdaptiveCpp on Graviton3, compared to A64FX, is likely due to the absence of Graviton3-specific architecture flags and optimizations in LLVM Clang. Analyzing compiler reports on the Graviton3 machine we find out that GEMM kernel is not vectorized because the cost model suggests that vectorization is unprofitable. With cross-product kernel we observe that AdaptiveCpp is not able to vectorize `ifelse` conditionals, similar to A64FX.

With `sycl::reduction`, we observed that none of the SYCL implementations were able to vectorize this construct on any of the machines tested. In contrast, a similar OpenMP benchmark was successfully vectorized with a reduction construct.

Overall, we observe that both SYCL implementations effectively vectorize across work items on all systems, utilizing various dimensions, iteration spaces, and data management methods, except for the Graviton3 system when using JIT compilation with the OpenMP backend. Similarly, loops within work items are vectorized successfully across all systems, except Graviton3. However, the ability to vectorize conditionals within work items differs between x86 and AArch64 systems, with x86 machines demonstrating the capability to vectorize conditionals effectively.

VI. CONCLUSION

In this work, we have developed a set of micro-benchmarks to study the performance of various SYCL features on available toolchains and compilation flows of SYCL implementations. Furthermore, we analyze the performance overheads of these SYCL features over standard programming models, such as OpenMP. Our benchmarks test memory bandwidth, parallelization, synchronization, and vectorization. We evaluated two different SYCL implementations, AdaptiveCpp and DPC++, on four different platforms: Intel Icelake, AMD EPYC Zen2, Fujitsu A64FX, and AWS Graviton3.

Our experimental results demonstrate that different compilation flows offer advantages in distinct aspects of the SYCL programming model. Specifically, Ahead-Of-Time (AoT) compilation is more effective for achieving higher memory bandwidth, while Just-In-Time (JIT) compilation performs better in parallelization tasks. Comparing different backends, we observe similar trends. Across systems, SYCL on the A64FX processor exhibits generally poor performance, while other systems perform comparably to each other. The vectorization benchmarks reveal that auto-vectorization depends heavily

on the sophistication of the compiler and its cost model. Specifically, we observe that on the Graviton3 system, the compiler struggles with auto-vectorization, indicating a lack of optimization for vectorized instructions on this architecture.

For future work, we plan to integrate our benchmarks with SYCL-Bench to complement the existing benchmark suite. We also intend to conduct a more detailed study on vectorization, expanding on the initial findings in future research.

ACKNOWLEDGMENT

This project has received funding from the European High-Performance Computing Joint Undertaking (JU) under Framework Partnership Agreement No. 800928 and Specific Grant Agreement No. 101036168 (EPI SGA2). The JU receives support from the European Union's Horizon 2020 research and innovation programme, and from the Swedish Research Council, among others. Additionally, this work has received funding from the project PRIDE from the Swedish Foundation for Strategic Research with reference number CHI19-0048, and the project P4PIM from the Swedish Research Council (VR) with project ID 2020-04892. The computations were enabled by resources provided by Chalmers e-Commons at Chalmers. The computations were enabled by resources provided by the National Academic Infrastructure for Supercomputing in Sweden (NAISS), partially funded by the Swedish Research Council through grant agreement no. 2022-06725. The authors thank the Barcelona Supercomputing Center, Spain for providing access to Mare Nostrum 4 ARM-CTE partition, the University of Bristol, UK for providing access to Isambard 2; and RIKEN-RCSS, Japan for access to A64FX servers. We thank Jens Domke for interesting discussions and interpretations of our experimental data.

REFERENCES

- [1] Khronos inc. (2024). [Online]. Available: <https://registry.khronos.org/SYCL/specs/sycl-2020/html/sycl-2020.html>
- [2] S. Matsuoka, "Fugaku and a64fx: the first exascale supercomputer and its innovative arm cpu," in *2021 Symposium on VLSI Circuits*, 2021, pp. 1–3.
- [3] Intel, Inc. Speed-up memory bandwidth bound workloads with hbm. [Online]. Available: <https://www.intel.com/content/www/us/en/high-performance-computing/hpc-high-bandwidth-memory-video.html>
- [4] Advanced Micro Devices, Inc. AMD 3D V-Cache™ Technology. [Online]. Available: <https://www.amd.com/en/products/processors/technologies/3d-v-cache.html>
- [5] Nec. Nec vector engine models. [Online]. Available: https://www.nec.com/en/global/solutions/hpc/sx/vector_engine.html
- [6] D. Ditzel. (2024) Invited talk: Why i chose to build esperanto's ai and hpc accelerator around risc-v. [Online]. Available: https://sc24.supercomputing.org/proceedings/workshops/workshop_pages/misc257.html
- [7] Intel. (2022) OneAPI. [Online]. Available: <https://www.intel.com/content/www/us/en/developer/tools/oneapi/overview.html>
- [8] Aksar Alpay. (2024) AdaptiveCPP. [Online]. Available: <https://github.com/AdaptiveCpp/AdaptiveCpp.git>
- [9] I. Baratta, C. Richardson, and G. Wells, "Performance analysis of matrix-free conjugate gradient kernels using sycl," in *International Workshop on OpenCL*, ser. IWOCCL'22. New York, NY, USA: Association for Computing Machinery, 2022.
- [10] A. Bagussetty, A. Panyala, G. Brown, and J. Kirk, "Towards cross-platform portability of coupled-cluster methods with perturbative triples using sycl," in *2022 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*, 2022, pp. 81–88.
- [11] K. Narasimhan, O. E. Farouki, M. Goli, M. Tanvir, S. Georgiev, and I. Ault, "Towards performance portability of ai graphs using sycl," in *2022 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*, 2022, pp. 111–122.
- [12] A. Alekseenko and S. Páll, "Comparing the performance of sycl runtimes for molecular dynamics applications," in *Proceedings of the 2023 International Workshop on OpenCL*, ser. IWOCCL '23. New York, NY, USA: Association for Computing Machinery, 2023.
- [13] G. Daiß, P. Diehl, H. Kaiser, and D. Pflüger, "Stellar mergers with hpx-kokkos and sycl: Methods of using an asynchronous many-task runtime system with sycl," in *Proceedings of the 2023 International Workshop on OpenCL*, ser. IWOCCL '23. New York, NY, USA: Association for Computing Machinery, 2023.
- [14] Z. Jin and J. S. Vetter, "Performance portability study of epistasis detection using sycl on nvidia gpu," in *Proceedings of the 13th ACM International Conference on Bioinformatics, Computational Biology and Health Informatics*, ser. BCB '22. New York, NY, USA: Association for Computing Machinery, 2022.
- [15] D. Nasikan and V. Yaremenko, "Performance evaluation of lu matrix decomposition using the sycl standard," *Technology audit and production reserves*, vol. 3, no. 2(71), p. 6–9, Jun. 2023.
- [16] V. R. Pascuzzi and M. Goli, "Benchmarking a proof-of-concept performance portable sycl-based fast fourier transformation library," in *Proceedings of the 10th International Workshop on OpenCL*, ser. IWOCCL '22. New York, NY, USA: Association for Computing Machinery, 2022. [Online]. Available: <https://doi.org/10.1145/3529538.3529996>
- [17] Z. Jin, "The rodinia benchmark suite in sycl," 6 2020. [Online]. Available: <https://www.osti.gov/biblio/1631460>
- [18] S. Lal, A. Alpay, P. Salzmann, B. Cosenza, A. Hirsch, N. Stawinoga, P. Thoman, T. Fahringer, and V. Heuveline, "SYCL-Bench: A Versatile Cross-Platform Benchmark Suite for Heterogeneous Computing," in *Euro-Par 2020: 26th International European Conference on Parallel and Distributed Computing*, ser. Euro-Par '20. Springer International Publishing, 2020.
- [19] Z. Jin and J. S. Vetter, "A benchmark suite for improving performance portability of the sycl programming model," in *2023 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2023, pp. 325–327.
- [20] E. Georganas, R. F. Van Der Wijngaart, and T. G. Mattson, "Design and implementation of a parallel research kernel for assessing dynamic load-balancing capabilities," in *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2016, pp. 73–82.
- [21] L. Crisci, L. Carpentieri, P. Thoman, A. Alpay, V. Heuveline, and B. Cosenza, "Sycl-bench 2020: Benchmarking sycl 2020 on amd, intel, and nvidia gpus," in *Proceedings of the 12th International Workshop on OpenCL and SYCL*, ser. IWOCCL '24. New York, NY, USA: Association for Computing Machinery, 2024. [Online]. Available: <https://doi.org/10.1145/3648115.3648120>
- [22] Lal, Sohan and Alpay, Aksel and Salzmann, Philip and Cosenza, Biagio and Hirsch, Alexander and Stawinoga, Nicolai and Thoman, Peter and Fahringer, Thomas and Heuveline, Vincent. (2024) ParRes. [Online]. Available: <https://github.com/ParRes/Kernels.git>
- [23] M. Breyer, A. Van Craen, and D. Pflüger, "Evaluation of sycl's different data parallel kernels," in *Proceedings of the 12th International Workshop on OpenCL and SYCL*, ser. IWOCCL '24. New York, NY, USA: Association for Computing Machinery, 2024. [Online]. Available: <https://doi.org/10.1145/3648115.3648130>
- [24] I. Z. Regulý, "Evaluating the performance portability of sycl across cpus and gpus on bandwidth-bound applications," in *Proceedings of the SC '23 Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis*, ser. SC-W '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 1038–1047. [Online]. Available: <https://doi.org/10.1145/3624062.3624180>
- [25] S. Joubé, H. Grasland, D. Chamont, and E. Brunet, "Comparing sycl data transfer strategies for tracking use cases," *Journal of Physics: Conference Series*, vol. 2438, no. 1, p. 012018, feb 2023.
- [26] J. M. Bull, F. Reid, and N. McDonnell, "A microbenchmark suite for openmp tasks," in *OpenMP in a Heterogeneous World*, B. M. Chapman, F. Massaioli, M. S. Müller, and M. Rorro, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 271–274.
- [27] "Evaluating attainable memory bandwidth of parallel programming models via babelstream," vol. 17, no. 3, p. 247–262, jan 2018.

- [28] J. Ciesko and S. L. Olivier, "Characterizing the performance of task reductions in openmp 5.x implementations," in *OpenMP in a Modern World: From Multi-device Support to Meta Programming*, M. Klemm, B. R. de Supinski, J. Klinkenberg, and B. Neth, Eds. Cham: Springer International Publishing, 2022, pp. 35–49.
- [29] Fujitsu. (2024) Fujitsu compiler package. [Online]. Available: <https://software.fujitsu.com/jp/manual/manualindex/p22000026e.html>
- [30] Khronos inc. (2024) OpenCL Standard. [Online]. Available: https://registry.khronos.org/OpenCL/specs/3.0-unified/html/OpenCL_Env.html#validation-rules
- [31] A. Poenaru, T. Deakin, S. McIntosh-Smith, S. Hammond, and A. Younge, "An evaluation of the fujitsu a64fx for hpc applications," in *Cray User Group 2021*, May 2021, Cray User Group 2021 ; Conference date: 03-05-2021 Through 05-05-2021. [Online]. Available: <https://cug.org/cug-2021/>
- [32] R. Jesus and M. Weiland, "A study on the performance implications of aarch64 atomics," in *High Performance Computing*, A. Batele, J. Hammond, M. Baboulin, and C. Kruse, Eds. Cham: Springer Nature Switzerland, 2023, pp. 279–296.

Cross-Architecture Autotuning for Single-Source Heterogeneous Programming Models

Hari Abram, Nikela Papadopoulou, Jens Domke,
and Miquel Pericàs

Published in

ACM International Conference on Supercomputing 2026

Cross-Architecture Autotuning for Single-Source Heterogeneous Programming Models

Hari Abram

Chalmers University of Technology and University of
Gothenburg
Gothenburg, Sweden
hariv@chalmers.se

Jens Domke

RIKEN
Kobe, Japan
jens.domke@riken.jp

Nikela Papadopoulou

School of Computing Science
University of Glasgow
Glasgow, United Kingdom
nikela.papadopoulou@glasgow.ac.uk

Miquel Pericàs

Chalmers University of Technology and University of
Gothenburg
Gothenburg, Sweden
miquelp@chalmers.se

Abstract

The rise of heterogeneous computing systems has intensified the need for performance-portable programming models and effective autotuning methodologies. Although compiler and runtime tuning are known to significantly influence application performance, it remains unclear how such optimizations transfer across different hardware architectures, particularly within single-source models such as SYCL. This work investigates the *transferability* of compile-time and runtime autotuning decisions across CPUs and GPUs, focusing on AdaptiveCpp, a SYCL implementation built on LLVM.

We introduce an automated framework that jointly explores compiler flags and runtime parameters using both Bayesian optimization and a tabu-search-based strategy. The tool orchestrates compilation, execution, and measurement while also providing statistical attribution via ridge regression to quantify the impact of individual tuning parameters. Through an extensive evaluation of CPUs and GPUs from multiple vendors, we demonstrate that autotuning can deliver substantial performance gains—up to $3\times$ on CPUs—yet the influence of specific compiler flags often diverges across different architectures. For example, flags such as `-fno-builtin` yield large improvements on CPUs but have negligible effect on GPUs. We also demonstrate that runtime-level choices, such as thread-placement policies, can significantly affect performance on CPUs.

Our findings highlight the challenges and opportunities of autotuning in heterogeneous, single-source programming ecosystems. They also underline the importance of architecture-aware autotuning strategies and motivate further exploration of cross-device performance modeling.

CCS Concepts

• **Computer systems organization** → **Heterogeneous systems**;
• **Computing methodologies** → *Parallel programming languages*;
Search algorithms; • **General and reference** → Performance.

Keywords

CPUs, GPUs, Autotuning, SYCL, OpenMP, OpenCL, AdaptiveCpp, Search algorithms, TPE, Tabu

ACM Reference Format:

Hari Abram, Nikela Papadopoulou, Jens Domke, and Miquel Pericàs. 2026. Cross-Architecture Autotuning for Single-Source Heterogeneous Programming Models. In *2026 International Conference on Supercomputing (ICS '26)*, July 06–09, 2026, Belfast, United Kingdom. ACM, New York, NY, USA, 12 pages. <https://doi.org/10.1145/3797905.3800519>

1 Introduction

In modern computing systems, the growing heterogeneity of hardware platforms has driven the development and adoption of diverse parallel programming models. These models aim to address the challenges associated with performance portability, software engineering, and code optimization. Prominent examples include SYCL [17], OpenCL [18], OpenACC [24], Kokkos [11], and RAJA [6], among others. Although each programming model offers distinct advantages, the performance of even highly optimized applications remains sensitive to a variety of factors. These include compiler optimization flags, the choice of backend used for kernel offloading to CPUs or GPUs, and runtime execution parameters. Consequently, achieving optimal performance requires careful tuning across the compilation, offloading, and execution stages.

While it is well understood that compiler flag auto-tuning can significantly affect application performance [5], it remains unclear how these auto-tuned parameters translate across heterogeneous hardware platforms. Consider AdaptiveCpp [4], a SYCL implementation built on the LLVM Clang compiler. When SYCL code is compiled, the frontend generates LLVM Intermediate Representation (LLVM-IR). At runtime, this intermediate representation is further lowered depending on the target hardware: for example, to CPU assembly code when executed on a CPU, or to NVIDIA PTX and subsequently GPU machine code when executed on an NVIDIA GPU. This workflow is presented in Figure 1. In practice, SYCL developers often distribute executables that embed this intermediate representation, enabling end users to perform runtime-level autotuning on their target hardware. However, this also means that opportunities for compile-time tuning are no longer available to the user once the binary has been generated.



This work is licensed under a Creative Commons Attribution 4.0 International License.
ICS '26, Belfast, United Kingdom

© 2026 Copyright held by the owner/authors(s).
ACM ISBN 979-8-4007-2522-7/26/07
<https://doi.org/10.1145/3797905.3800519>

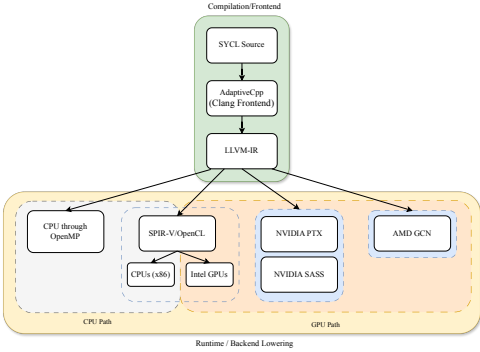


Figure 1: AdaptiveCpp flow with explicit *Compilation* vs. *Runtime* regions and CPU/GPU target paths.

This workflow raises two central questions. **First**, when a particular set of compiler flags is identified—via autotuning—as significantly influencing performance for a given benchmark on a specific CPU, does that influence persist when the same code is executed on a GPU? In other words, to what extent do compile-time optimizations transfer across heterogeneous architectures within a single-source programming model such as SYCL? **Second**, how does the choice of runtime backend affect overall performance? Even when targeting only CPUs, developers can select among multiple runtimes, each exposing different scheduling, threading, and memory-management behaviors that may lead to substantially different performance outcomes.

Previous work in the area of auto-tuning for optimization has primarily focused on traditional programming models (eg, C/C++, OpenMP) [5, 9, 25, 26] and has largely overlooked the characteristics of portable programming paradigms. These models—such as SYCL—utilize features, such as single-source programming, just-in-time (JIT) compilation, and multi-stage compilation pipelines, which fundamentally influence performance tuning and optimization strategies. Moreover, existing studies [5, 26] typically target optimization for a single platform and a single compiler toolchain. While valuable, such efforts do not address a critical question: how does the autotuning landscape change when a single compiler is responsible for generating code across multiple heterogeneous platforms?

Previous autotuning efforts commonly rely on search paradigms such as Bayesian optimization or genetic algorithms [19, 20, 26]. While these methods are effective at finding high-quality configurations for a chosen objective, they offer limited transparency regarding why specific parameters appear in the final configuration. In contrast, this work introduces a meta-heuristic approach—*Tabu Search*—whose iterative, neighborhood-based exploration naturally improves post-hoc interpretability by making the contribution of individual compiler and runtime parameters easier to trace.

In this work, we introduce an automated tuning framework that explores the search space of compiler and runtime parameters, both jointly and independently. Our approach employs a Python-based

tool that parses a JSON configuration file to automate code compilation, execution, performance measurement, and search-space exploration. We evaluate established optimization techniques—such as Bayesian optimization—as baselines and compare them against a Tabu-search-based method adapted for this study. The autotuning workflow is evaluated across both CPU and GPU architectures. Additionally, we leverage statistical analysis techniques, including ridge regression, to quantify the influence of individual compiler flags on benchmark performance for the Bayesian search algorithm.

Our experiments reveal the following trends. Bayesian TPE (Tree-Structured Parzen Estimator) delivers the most reliable performance gains, avoiding the severe regressions frequently triggered by Tabu Search, which is useful only for aggressive, non-production exploration. A small subset of compiler flags dominates behavior: `-ffast-math` and `-funsafe-math-optimizations` consistently improve performance, while `-fno-builtin` enables wider vectorization on AMD Genoa, yielding up to 3× speedups. In contrast, flags such as `-fno-unroll-loops` and loop-disabling LLVM passes often produce large slowdowns, especially on GPUs. Runtime choices further shape performance outcomes: the OpenMP backend typically outperforms OpenCL by as much as 8.9×—except on strongly memory-bound kernels, and poor thread placement (e.g., over NUMA domains) can degrade performance by an order of magnitude. Favorable settings include compact thread binding, dynamic scheduling for irregular workloads, and active wait policies.

The main contributions of this paper are as follows:

- We adapt a meta-heuristic search algorithm to the compile-time and runtime autotuning problem, improving the explainability of the tuning process by revealing how specific compiler flags and runtime settings influence performance.
- We conduct a detailed experimental study to examine how auto-tuned parameters vary across CPU and GPU architectures, encompassing devices from multiple hardware vendors.
- We employ advanced statistical analysis methods to identify and quantify the impact of individual configuration parameters on benchmark performance.

Section 2 delves into the design of our auto-tuning setup. Section 3 describes the experimental configuration and the benchmark suite underpinning our evaluation, while Section 4 reports quantitative results. Section 5 surveys related auto-tuning systems. Finally, Section 6 summarizes our findings.

2 Methodology

To navigate the large design space induced by compiler flags and runtime environment variables, we introduce **SCOuT**, a Python-based framework for *objective-driven* design-space exploration (DSE). SCoUT automates compilation, execution, and performance measurement within a closed-loop workflow that guides search algorithms toward high-performing parameter configurations while continuously exploring new candidates. This section describes the search algorithms employed and implemented in this work, as well as the overall control flow of the tool.

2.1 Search Algorithms

Autotuning frameworks rely heavily on the choice of search algorithm—ranging from established approaches such as Bayesian optimization to more specialized methods like particle swarm optimization. Fundamentally, these algorithms aim to optimize a chosen objective (e.g., application runtime, energy consumption, or binary size). However, the parameter spaces that influence these objectives are often large, nonlinear, and highly irregular, making exhaustive exploration impractical. Consequently, effective autotuning requires guided search strategies that can efficiently identify high-performing configurations without enumerating the entire design space.

In this work, we focus on two such strategies: the Bayesian Tree-Structured Parzen Estimator [30] (TPE) and Tabu Search [13]. We explain how these two search algorithms are incorporated into our search of the parameter space.

2.1.1 Bayesian Tree-Structured Parzen Estimator (TPE). We employ the Python-based library OPTUNA [3] to perform Bayesian Tree-Structured Parzen Estimator (TPE) optimization. In our workflow, Optuna is supplied with the compiler flags and environment variables to be explored. Based on this configuration and the user-defined objective (e.g., minimizing runtime or energy), Optuna conducts a search to identify high-performing or low-performing configurations.

Search Procedure. The Bayesian TPE search within Optuna operates as follows:

- (1) **Trial specification:** The user specifies the number of trials to be executed.
- (2) **Configuration sampling:** For each trial, the TPE sampler selects a combination of compiler flags and environment variables from the declared search space.
- (3) **Evaluation:** Our tool builds and executes the benchmark using the sampled configuration, collecting the metric specified for optimization.
- (4) **Guided refinement:** Based on outcomes from previous trials, Optuna updates its model and proposes new configurations for subsequent trials.

Bayesian Model. These models distinguish between *promising* and *non-promising* regions and guide the search toward areas likely to yield better objective values, thereby improving efficiency compared to random or exhaustive exploration.

2.1.2 Tabu Search. This algorithm is a meta-heuristic, neighborhood-based optimization method that iteratively improves a candidate solution by exploring its local neighborhood while avoiding cycles through the use of a tabu list. In our framework, Tabu Search offers an interpretable, iterative mechanism for exploring compiler flags and environment variable combinations. Figure 2 presents the visual representation of tabu search.

Implementation in Our Work. We implement Tabu Search in SCOUT as follows:

- (1) **Initialization:** The search begins from a base configuration (e.g., -O3). In iteration 1, we generate the neighborhood of

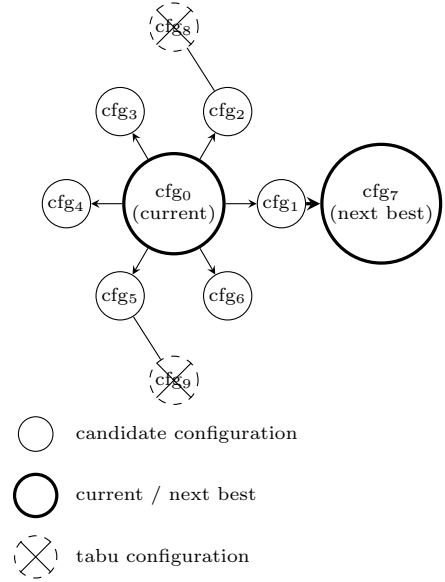


Figure 2: Each node represents a configuration (compiler flags and runtime environment variables). From the current best configuration (cfg_0), the algorithm explores its neighborhood, selects a new best configuration (cfg_7), and avoids revisiting configurations marked as tabu (cfg_8 , cfg_9).

this base configuration. The number of flags in each configuration grows with the iteration count.

- (2) **Neighborhood size:** The user specifies the neighborhood size, which determines how many candidate configurations are generated per iteration.
- (3) **Evaluation:** For each candidate configuration in the neighborhood, our tool builds and executes the benchmark and records the objective metric.
- (4) **Selection:** The best-performing configuration in the neighborhood becomes the starting point for the next iteration.

This process repeats until convergence, defined as observing no meaningful improvement in the best configuration across successive iterations.

Tabu Tenure. To prevent cycling, previously visited configurations are stored in a *tabu list*. A configuration remains “tabu” for a fixed number of iterations, known as the *tabu tenure*. During this period, the algorithm is prohibited from revisiting that configuration unless an aspiration criterion is met (e.g., it yields a globally best solution).

Table 1 presents the differences between *Tabu* and *Bayesian TPE* across the various aspects.

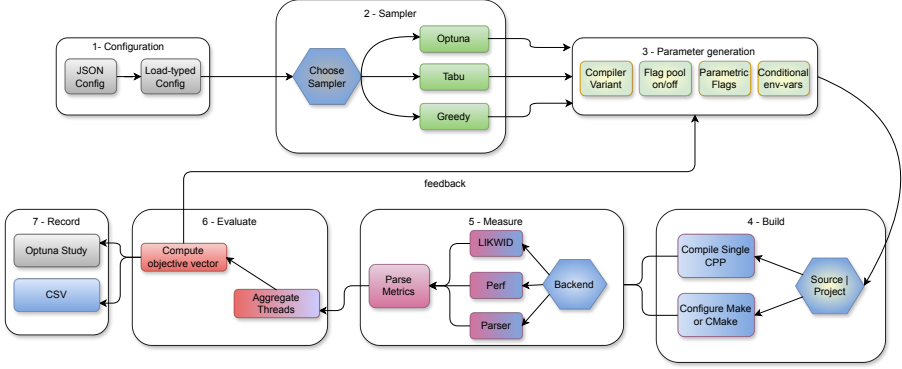


Figure 3: End-to-end workflow of SCOUT.

Table 1: Comparison of Bayesian TPE and Tabu Search in our work.

Aspect	TPE	Tabu
Search type	Global	Local
Sampling	Probabilistic	Neighborhood
User input	Trials	Base config. Neighborhood sizes Max. iterations
Config. generation	Random draws	Neighbor sets
Evaluation	Per trial	Per neighborhood
Update mechanism	Bayesian model	Tabu list
Stopping rule	Trials	No improvement
Strength	Broad search	Interpretable steps

2.2 SCOUT Overview

Figure 3 illustrates the end-to-end workflow of **SCOUT**, which consists of configuration, parameter generation, building, measurement, and evaluation.

2.2.1 Configuration. SCOUT is driven by a single JSON specification that declares the project path, the chosen search algorithm and study, the compiler/runtime parameters to explore, and the backend tool used to collect metrics.

2.2.2 Search Algorithm. Python-based search algorithms are instantiated through the **OPTUNA** [3] sampler interface. **OPTUNA** supports samplers, including TPE, NSGA-III, CMA-ES, and a random sampler. The tabu search implemented by the authors can be selected by changing the study key to `tabu`.

Listing 1: Sampler selection in JSON

```
{ "search": { "study" : "optuna",
```

```
"sampler": "tpe | nsga3 | cmaes | rs" } }
```

2.2.3 Parameter Generation. For each trial, SCOUT constructs a build command from the parameter space defined in the configuration file. The space includes:

- **Mutually exclusive compiler flags**
- **Optional flag pool** (subset selected per trial)
- **Parametric flags** drawn from user-provided value lists
- **Environment variables** influencing runtime behavior

Example specification is provided in Listing 2

Listing 2: Example parameter-space elements.

```
"compiler_flags" : [ "-O2", "-O3" ],
"compiler_flag_pool" : [ "-fvectorize", "-mllvm --vplan" ],
"compiler_params" : { " -march": [ "native", "skylake" ] },
"env" : { "OMP_SCHEDULE": [ "static", "dynamic" ] }
```

2.2.4 Build. SCOUT invokes either the project's existing make/CMake system or compiles a single source file directly. The configuration specifies the build directory, compiler, and optional variables.

2.2.5 Measurement. Metrics are obtained via **LIKWID** [14], **perf** [1], or a user-specified parser, for example, parsing the execution times from the output printed to the terminal. SCOUT applies the environment variables selected for the current trial, runs the benchmark, and aggregates repeated executions:

Listing 3: Measurement backends.

```
"likwid": { "group": "SCOUT" },
"perf" : { "events": [ "cycles", "instructions" ] },
"parser": { "label": "avg", "aggregate": "sum", "warmup_runs": 1 },
"runs" : 5
```

2.2.6 Evaluation. The measured metrics are mapped to user-defined objectives (e.g., minimize runtime). The search algorithm then uses this feedback to guide the selection of parameters for subsequent trials.

Listing 4: Objective definition.

```

"objectives": [
  { "metric": "Runtime", "goal": "min" },
  { "metric": "kernel_time_s", "goal": "min" },
  { "metric": "Runtime (RDTSC) [s]", "goal": "min" }
]

```

2.2.7 Record. SCOUT records all results in a CSV file and reports the best (or Pareto-optimal) configuration at the end of the search.

SCOUT provides a unified, extensible, and automated pipeline for compiler- and runtime-level autotuning, supporting both single- and multi-objective searches while remaining agnostic to the underlying search algorithm.

2.3 Statistical methods

The Bayesian TPE search algorithm performs the design space search by selecting a combination of flags and environment variables and optimizing for the given objective. While this is a valid idea to find the best combination of compiler flags or runtime variables, it does not suggest which individual flags or runtime variables have the highest impact on the performance. To overcome this limitation, we perform ridge regression on the data we obtain from the trials.

Ridge regression [16]. To estimate the conditional effects of individual flags and environment variables, we fit a linear model with an ℓ_2 (ridge) penalty. Specifically, we solve

$$\min_{\beta_0, \beta} \sum_i (y_i - \beta_0 - X_i \beta)^2 + \alpha \|\beta\|_2^2,$$

where X is a binary design matrix with $X_{ij} = 1$ if the token (flag or environment setting) corresponding to column j is present in trial i , and 0 otherwise, β is the ridge coefficient, and y_i is the chosen metric, α is set to $1e-3$; the intercept β_0 is left unpenalised. The ridge coefficients present the importance of each flag. For example, with a runtime as a metric (y_i , where a lower value is better), a negative ridge coefficient has a positive impact on the performance.

3 Experimental Setup

This section details the setup for the experiments, where we discuss the platforms used to conduct the experiments and our software setup.

For this study, we utilized AMD EPYC 9354 Zen4 (codenamed "Genoa") and Intel Xeon Platinum 8358 (codenamed "Icelake") CPUs, as well as NVIDIA A40 and AMD MI100 GPUs. Table 2 presents the specifications of the systems used in this study.

Table 2: Compute platforms used in this study.

Feature	Genoa	Ice Lake	A40	MI100
Vendor	AMD	Intel	Nvidia	AMD
Micro Arch.	Zen 4	Ice Lake	Ampere	CDNA
ISA	x86-64	x86-64	CUDA/PTX	GCN
Cores / SMs	64	64	84 SMs	120 CUs
NUMA	8	2	–	–
Freq.	3.8 GHz	3.7 GHz	1.74 GHz	1.50 GHz
L1	32 KB	32 KB	128 KB/SM	16 KB per CU
L2	1 MB	1 MB	6 MB	8 MB
L3	512 MB	22 MB	–	–

We focus on portable programming models and use SYCL as the primary testbed for evaluating our autotuning approach. We adopt the AdaptiveCpp implementation of SYCL because of its open-source nature and support multiple backends for kernel offloading. All experiments use AdaptiveCpp v25.02.0, built on LLVM v19.1.7.

For the Bayesian TPE search algorithm, SCOUT interfaces with OPTUNA v4.0.0 and performs a maximum of $N = 50$ trials per benchmark. For Tabu search, we limit the search to 6 iterations, with each iteration exploring a *neighbourhood* of 12 configurations. The *tabu tenure* is fixed at 3. A Tabu search run terminates early if no improvement in the objective is observed for two consecutive iterations.

Performance metrics were collected using a custom parser designed to process the benchmark's terminal output. Each benchmark reports the execution time of its constituent kernels, which our parser extracts and aggregates. When available, we rely on SYCL's event-timing interface to record kernel durations, providing a consistent and portable timing mechanism across both CPU and GPU devices.

The benchmark suite consists of 26 workloads drawn from HeCBench. Except for inserting a lightweight macro for kernel timing, we made no modifications to the benchmarks. The macro records the execution time of all kernels in each benchmark and reports both per-offload and average timings. The full set of benchmarks used in this study is listed in Table 3.

Table 3: List of HeCBench benchmarks used in this paper, grouped per category.

Category	Benchmarks
Stencil / PDE	hotspot, hotspot3D, fdt3d, iso2dfd
Lattice Boltzmann	d2q9-bgk, d3q19-bgk
Vision / Signal Proc.	convolutionSeparable, convolution1D
Transforms	dct8x8, fwt, hwt1d
Linear Algebra / Solvers	gaussian, lud
Reduction / Sorting	histogram, bitonic-sort
Graph / Sparse	bfs, lanczos, jaccard
MD/CFD	lavaMD, easyWave
Finance	black-scholes, libor
Machine Learning	attentionMultiHead, gru, crossEntropy

All benchmarks were compiled once per generated flag set. For each trial, the tool executes the binary with arguments provided by the user three times and aggregates the metric values at the end of the trial. For the search objective, we test both minimizing and maximizing the runtime of the benchmark.

Table 4 summarizes the parameter space explored in this study. These parameters define the search domain for both compile-time and runtime tuning. The selection is informed by prior experience and observations from the compiler-optimization literature [7]. We adopt -O3 as the baseline optimization level and extend the search space with a curated set of LLVM/Clang flags and SYCL-related environment variables. In particular, the DPCPP_CPU_* variables control thread placement and scheduling when using the Intel OpenCL driver.

Most of the compiler flags included in the search space explicitly disable individual optimization passes. We start from the most aggressive optimization level (-O3) and progressively disable specific optimizations to observe their individual and combined effects on

Table 4: Parameter space explored in this study

Configuration	Values
compiler_flags	-O3
compiler_flag_pool	-mllvm -disable-memop-opt -mllvm -disable-partial-inlining -mllvm -disable-vector-combine -mllvm -disable-adv-copy-opt -mllvm -disable-zaddr-hack -mllvm -disable-branch-fold -mllvm -disable-loop-idiom-vectorize-all -mllvm -disable-loop-level-heuristics -mllvm -disable-select-optimize -mllvm -disable-i2p-p2i-opt -mllvm -disable-promote-alloca-to-vector -fno-unroll-loops -fno-optimize-sibling-calls -fno-plt -fno-global-isel -fno-signed-zeros -fno-strict-aliasing -fno-keep-static-consts -fno-builtin -fno-finite-loops -fno-vectorize -ffast-math -funsafe-math-optimizations
Environment variables	ACPD_VISIBILITY_MASK: {omp, ocl} OMP_PLACES: {sockets, cores, threads, numa_domains} OMP_PROC_BIND: {false, true, close, spread, master} OMP_SCHEDULE: {static, dynamic} KMP_AFFINITY: {compact, scatter, balanced} OMP_WAIT_POLICY: {active, passive} DPCPP_CPU_PLACES: {sockets, cores, threads, numa_domains} DPCPP_CPU_SCHEDULE: {static, dynamic} DPCPP_CPU_CU_AFFINITY: {false, true, close, spread, master}

performance. This approach is preferable to starting from a lower baseline such as -O1, since several optimizations present in -O3 are not exposed as independent user flags and would therefore be inaccessible when building upward from a lower optimization level.

4 Results

This section presents our experimental results, organized into three parts. The first subsection, *Search Algorithm Comparison*, evaluates how the different search strategies perform across platforms. The second subsection, *Auto-Tuning on CPUs and GPUs*, analyzes the tuning impact across the architectures. The final subsection, *Runtime Tuning*, examines the impact of runtime-level parameters on both CPU and GPU execution.

4.1 Search Algorithm Comparison

As described in Section 2, we evaluate the two search strategies—Bayesian TPE and Tabu Search—by comparing the number of unique configurations tested for each benchmark. In addition, we assess the quality of the configurations they produce by examining the best and worst performance achieved relative to the baseline configuration (which is -O3) for every benchmark.

Table 5 reports the number of unique configurations evaluated during compile-time tuning—including the baseline execution—using both search algorithms under the parameter settings

Table 5: Number of unique configurations tested by TPE and Tabu Search, with baseline (-O3) runtimes for each platform.

Benchmark	NVIDIA A40			AMD Genoa		
	TPE	Tabu	Base [s]	TPE	Tabu	Base [s]
attentionMultiHead	40	68	2.082	21	69	5.102
bfs	27	46	0.002	24	36	0.026
bitonic-sort	22	68	0.113	19	36	0.620
black-scholes	26	69	0.391	39	69	2.707
convolution1D	30	35	90.761	18	-	487.365
convolutionSeparable	26	35	2.246	34	68	12.017
crossEntropy	24	46	1.152	17	34	2.738
d2q9-bgk	17	72	1.717	25	36	7.825
d3q19-bgk	38	57	27.283	36	69	64.135
dct8x8	26	68	0.201	41	69	1.029
easyWave	23	35	1.046	9	34	0.577
fwf	29	35	0.309	22	47	2.003
gaussian	24	71	0.598	24	58	2.843
gru	25	46	0.003	21	68	0.031
histogram	18	35	0.058	23	46	0.251
hotspot3D	19	36	0.315	41	69	2.261
hotspot	9	35	0.003	21	34	0.024
hwtid	20	35	0.999	38	60	0.698
iso2dfd	13	35	0.135	34	35	0.522
jaccard	22	35	24.073	26	-	123.031
lanczos	9	45	0.908	17	48	3.790
lavaMD	33	72	0.015	9	35	0.698
libor	26	24	1.721	24	35	3.944
lud	28	70	0.251	17	60	0.646

described in Section 3. Results are shown for all benchmarks in this study. Across both platforms, Tabu Search consistently explores a larger number of unique configurations than TPE. Although TPE was configured to run 50 trials, the actual number of *unique* configurations observed is often no more than 40. This occurs because, once TPE identifies a promising region of the search space, it frequently resamples the same configuration when no further improvements are predicted, thereby limiting the number of distinct builds tested. In contrast, Tabu Search systematically explores new neighborhoods until its stopping condition is met—two consecutive iterations without improvement in execution time—which naturally results in a larger set of unique configurations. For some benchmarks, the table contains no Tabu Search data. This omission is due to the long execution times of these workloads.

Tabu Search evaluates more configurations than TPE due to its exploratory nature, whereas TPE revisits promising regions once identified. The total tuning time reflects compilation, search, and benchmark execution costs. In practice, users can reduce autotuning overhead by lowering benchmark iteration counts or tuning on smaller problem sizes before scaling up.

Figure 4 demonstrates the effect of the two search strategies, TPE and Tabu search, for the AMD Genoa CPU and the NVIDIA A40 GPU. We only present a subset of benchmarks as we observe similar differences in other benchmarks. All values are normalized to the untuned (-O3) baseline runtime of each benchmark.

On the AMD Genoa system, TPE consistently delivers moderate but reliable improvements. Across all benchmarks, the best TPE configurations achieve speedups between roughly 1.0 $\times$ and 3.7 $\times$, with the largest gain observed for dct8x8. The corresponding worst-case TPE configurations remain close to the baseline: for most kernels, TPE_{worst} stays within 1.0–1.3 $\times$, indicating that the method rarely explores configurations that severely degrade performance.

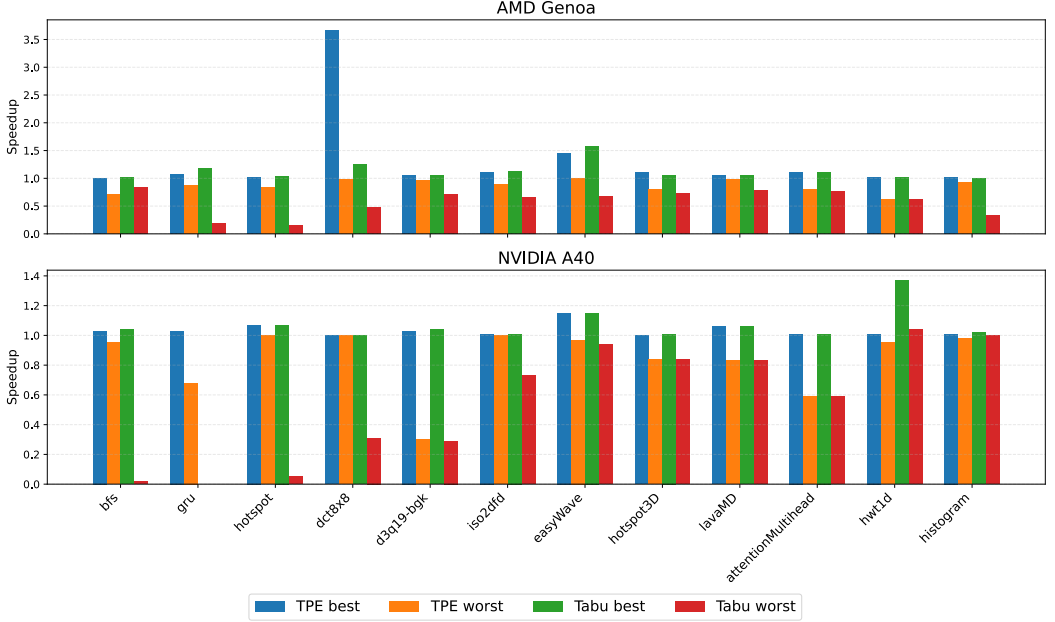


Figure 4: Per-benchmark speedups over the baseline (-O3) SYCL configuration for AMD Genoa and NVIDIA A40 (higher is better).

Tabu search, in contrast, exhibits higher variance. While its best results are occasionally competitive with or slightly better than TPE (e.g., *easyWave*), its worst-case outcomes can be substantially worse, with slowdowns up to about 0.2–0.167× on *gru*, *hotspot*, and *histogram*. This suggests that on the CPU, Tabu search is more aggressive but also more fragile: it can converge to good configurations but frequently visits very poor ones.

The trends are even more pronounced on the NVIDIA A40 GPU. TPE again yields modest and stable improvements, typically around 1.2–1.4×, and only rarely produces configurations that regress meaningfully relative to the baseline. Tabu search, however, suffers from extreme worst-case slowdowns on several benchmarks. For instance, *bfs* and *hotspot* experience slowdowns of 0.02× and 0.05×, respectively, in the worst Tabu configurations, and *d3q19-bgk* incurs a more than 0.33× slowdown. Even where Tabu finds reasonable best configurations, these dramatic outliers highlight a substantially less predictable search behavior on the GPU.

Overall, these results indicate that TPE offers a more favorable balance between performance and robustness for our SYCL autotuning workload, whereas Tabu Search exhibits a more exploratory behavior. It is worth noting, however, that the experiments shown here target an objective of **maximizing** runtime. When minimizing runtime, the search often becomes trapped in local minima, reducing the diversity of configurations explored and limiting the ability to uncover more informative optimization patterns.

Both search algorithms systematically improve over the baseline on both architectures and avoid catastrophic regressions, whereas Tabu occasionally discovers comparable or better configurations but at the cost of much larger performance variance and severe worst-case slowdowns, particularly on the GPU. In practice, this makes TPE a safer choice for automated tuning, especially in scenarios where long-running or production workloads cannot tolerate large auto-tuning times and tuning algorithms that focus on exploration, rather than exploitation.

4.2 Auto-Tuning on CPUs and GPUs

While tuning reveals substantial differences between the best and worst configurations for each benchmark, it is also important to understand how CPU and GPU platforms diverge when using a single-source programming model. To this end, in this section, we analyze which compiler flags meaningfully affect performance on CPUs versus GPUs, and whether consistent correlations emerge across benchmarks.

To evaluate the transferability of autotuning across CPUs and GPUs, we focus on the four benchmarks that exhibit the largest performance differences across the platforms. Specifically, we analyze *hotspot*, *dct8x8*, *d3q19-bgk*, and *easyWave*.

hotspot: From Figure 4, we observe that although the best configuration yields only marginal performance improvement for this

benchmark, the worst configuration leads to a substantial degradation. A similar pattern appears on Intel Ice Lake and AMD MI100: while the best-case performance improves by roughly $1.2\times$ – $1.4\times$ over baseline, the worst-case configuration can degrade performance by up to $4\times$ on Intel Ice Lake.

Using Tabu Search on the A40, we observe substantial performance degradation when loop-level optimizations are disabled. In particular, adding any of the following flags leads to noticeably poor performance: `-mllvm -disable-loop-level-heuristics`, `-mllvm -disable-i2p-p2i-opt`, and `-fno-unroll-loops`. However, this effect is not universal. When combined with certain other flags—such as `-ffast-math`—the performance penalty is mitigated or even reversed, suggesting that loop-level optimizations interact strongly with other optimization passes on this platform.

For AMD Genoa, the flag that produces the largest performance degradation is `-mllvm -disable-2addr-hack`. Notably, this degradation persists only when the flag is used in isolation: combining it with any other flag—except `-mllvm -disable-i2p-p2i-opt`—mitigates the slowdown. This suggests that the performance penalty introduced by disabling the two-address optimization is not influenced by `-mllvm -disable-i2p-p2i-opt`, whereas interactions with other flags appear to offset its negative impact.

For Intel Ice Lake, we again observe that a distinct set of flags triggers performance degradation. In this case, the combination of `-mllvm -disable-select-optimize`, `-fno-built-in`, and `-funsafe-math-optimizations` leads to a substantial slowdown. Interestingly, this degradation occurs only when these three flags are used together; adding any additional flag restores performance, indicating that the negative interaction is specific to this exact trio.

dct8x8: For this benchmark, we observe a substantial performance improvement—up to $3\times$ —on AMD Genoa when using the TPE search algorithm, an effect not reproduced by Tabu Search. This gain is attributable to the inclusion of the `-fno-built-in` flag, which TPE explored but Tabu did not encounter during its search trajectory. Further analysis shows that `-fno-built-in` enables the compiler to generate wider vector instructions: without the flag, the assembly contains primarily 128-bit operations, whereas enabling it produces 256-bit and even 512-bit vector instructions. This suggests that certain built-in functions inhibit vectorization on this platform. Notably, this behavior appears specific to AMD Genoa.

For GPUs, the performance variation on AMD hardware is smaller than what we observe on NVIDIA GPUs. As in the hotspot benchmark, similar flags lead to gradual performance degradation on NVIDIA GPUs. This trend is illustrated in Figure 5.

On Intel Ice Lake, the primary source of performance degradation is the `-fno-unroll-loops` flag.

d3q19-bgk: This benchmark implements a Lattice–Boltzmann method, for which the performance variation on both AMD Genoa and Intel Ice Lake CPUs is minimal. A similar trend holds for AMD GPUs. In contrast, NVIDIA GPUs exhibit a much wider performance spread, with the worst-performing configurations running up to $0.33\times$ slower than the baseline.

A Tabu Search analysis of this benchmark reveals a pattern similar to the previous case: the `-fno-unroll-loops` flag is the primary source of performance degradation. Whenever the search algorithms explore configurations that include this flag—whether

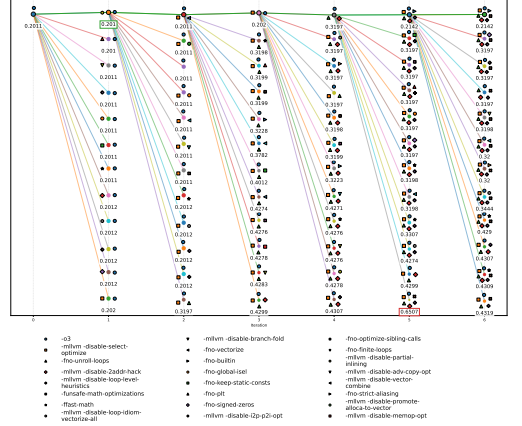


Figure 5: Tabu search path for NVIDIA A40 for benchmark `dct8x8`. Annotations in the figure correspond to the execution time for that particular configuration.

alone or in combination with others—the resulting performance consistently falls into the worst-performing category.

easyWave: This benchmark models shallow-water wave dynamics by solving simplified tsunami and surface-wave propagation equations. On AMD Genoa, autotuning yields up to a $1.57\times$ performance improvement, while the worst configurations degrade performance by as much as $0.684\times$. Both improvements and degradations arise from specific combinations of compiler flags, indicating that this benchmark is particularly sensitive to interactions among optimization passes.

On the remaining platforms, performance improvements are more modest, typically ranging from $1.1\times$ to $1.2\times$, with Intel CPUs exhibiting the largest gains within this range.

The preceding analyses rely primarily on Tabu Search results, as its iterative structure makes it straightforward to identify which individual flags contribute to performance gains or losses, as can be seen in Figure 5. In contrast, TPE—implemented via the Optuna library—samples configurations probabilistically, making it more challenging to isolate the influence of specific flags. To address this, we apply ridge regression, as described in Section 2, to estimate the correlation between individual flags and runtime.

Figure 6 shows the resulting regression coefficients, where smaller values correspond to a stronger reduction in runtime. The trends largely align with the flag effects identified earlier. Notably, `-ffast-math` consistently exhibits a beneficial impact across benchmarks.

4.3 Runtime-tuning

In the previous subsections, our analysis focused on compile-time autotuning and the performance variations arising from different compiler flag configurations. In this subsection, we shift focus to the

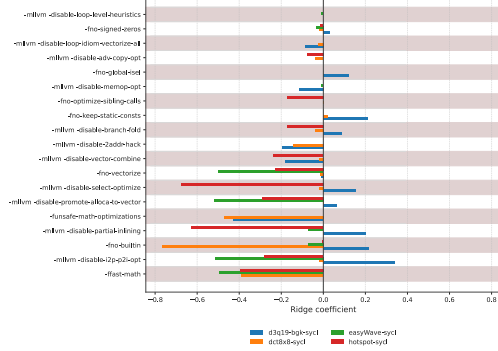


Figure 6: Ridge coefficients for the selected benchmarks on AMD Genoa platform

runtime layer, examining how performance is influenced by selecting different runtime parameters—specifically, the backend, scheduling policy, and thread-placement strategy. Our study concentrates on CPU platforms, since the runtime-level design space exposed through environment variables for NVIDIA and AMD GPUs is minimal, and thus offers limited opportunity for meaningful tuning.

Figure 7 shows the normalized performance results for the AMD Genoa CPU when SYCL kernels are executed using two different backends: OpenCL and OpenMP. These results are obtained for default runtime environment variables. All values are normalized to the SYCL kernel performance under the OpenMP backend. Overall, the OpenMP backend delivers consistently better performance than OpenCL, with the most pronounced advantage observed in the gru benchmark, where OpenMP achieves an 8.93× speedup over OpenCL. However, this trend is not uniform across all benchmarks. In a few cases, OpenCL outperforms OpenMP; the most notable example is lavaMD, where OpenCL achieves a 3.57× improvement relative to OpenMP. Only one benchmark, lanczos, exhibits less than a 10% difference between the two backends.

Examining the overall trends across benchmarks that perform best with the OpenMP backend, we observe that they span a broad spectrum—from compute-intensive to memory-intensive workloads. In contrast, the benchmarks where OpenCL outperforms OpenMP tend to be predominantly memory-bound. Profiling further reveals that, for these cases, OpenCL generally achieves superior auto-vectorization compared to OpenMP. However, when OpenMP is able to auto-vectorize the SYCL kernels to a degree comparable with OpenCL, it consistently delivers higher performance.

We additionally find that the OpenCL backend typically exhibits a larger memory footprint than OpenMP. This increased memory pressure leads to higher cache-miss rates, which in turn contribute to the slower overall runtime of OpenCL for most benchmarks.

We evaluate the runtime environment parameters listed in Table 4. It is important to note that our tool explores the runtime parameter space conditionally: OpenMP-specific runtime variables are considered only when `ACPP_VISIBILITY_MASK = omp`, and

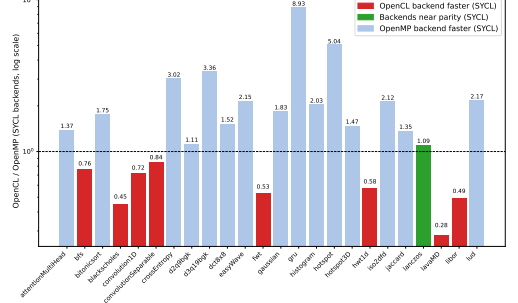


Figure 7: Performance comparison when the SYCL kernels are offloaded with OpenCL and OpenMP backend on AMD Genoa platform

Table 6: Best and worst configurations for SYCL workloads on an AMD Genoa (runtimes in seconds; values rounded to two decimals). Symbolic configuration codes map to actual runtime parameters as follows: B_i encodes `ACPP_VISIBILITY_MASK` ($B_1=omp$, $B_2=ocl$); P_i encodes CPU resource placement variables (`OMP_PLACES` or `DPCPP_CPU_PLACES`: $P_1=cores$, $P_2=sockets$, $P_3=threads$, $P_4=numa_domains$); BD_i encodes thread binding and affinity (`OMP_PROC_BIND` or `DPCPP_CPU_CU_AFFINITY`: $BD_1=false$, $BD_2=true$, $BD_3=close$, $BD_4=master$, $BD_5=spread$); S_i encodes the scheduling policy (`OMP_SCHEDULE` or `DPCPP_CPU_SCHEDULE`: $S_1=static$, $S_2=dynamic$, $S_3=affinity$); A_i denotes affinity policies (`KMP_AFFINITY`: $A_1=compact$, $A_2=scatter$, $A_3=balanced$, $A_4=granularity-fine,compact$, $A_5=granularity-fine,scatter$); W_i encodes the OpenMP waiting policy (`OMP_WAIT_POLICY`: $W_1=active$, $W_2=passive$).

Benchmark	Best (B)	Worst (W)	W/B	Best configuration	Worst configuration
attentionMultiHead	3.97	123.31	31.03	$B_1, P_1, BD_1, S_2, A_2, W_1$	B_2, P_4, BD_4, S_1
bfs	0.02	0.18	8.03	$B_1, P_1, BD_1, S_1, A_1, W_1$	B_2, P_4, BD_4, S_1
bitonicSort	0.61	2.73	4.45	$B_1, P_2, BD_1, S_2, A_1, W_1$	B_2, P_4, BD_4, S_1
blackscholes	1.13	2.80	2.49	B_2, P_2, BD_1, S_1	$B_1, P_4, BD_1, S_1, A_3, W_2$
convolution1D	362.13	897.68	2.48	B_2, P_1, BD_1, S_3	B_2, P_4, BD_4, S_1
convolutionSeparable	11.47	28.50	2.48	$B_1, P_4, BD_1, S_2, A_3, W_1$	B_2, P_1, BD_1, S_1
crossEntropy	2.55	10.78	4.22	$B_2, P_1, BD_1, S_2, A_1, W_1$	B_2, P_4, BD_4, S_1
d2q9bgk	7.74	101.32	13.08	$B_1, P_3, BD_4, S_1, A_3, W_1$	B_2, P_4, BD_4, S_3
d3q19bgk	63.23	330.22	5.22	$B_1, P_3, BD_2, S_2, A_1, W_1$	B_2, P_4, BD_4, S_1
dct68	1.03	2.27	2.19	$B_1, P_4, BD_2, S_1, A_4, W_2$	B_2, P_2, BD_2, S_1
easyWave	0.38	4.06	10.79	$B_1, P_3, BD_1, S_1, A_4, W_1$	B_2, P_4, BD_4, S_1
fftd3d	0.51	1.50	2.95	B_2, P_1, BD_4, S_2	B_2, P_4, BD_4, S_1
fwt	1.01	5.02	4.95	B_2, P_1, BD_4, S_3	B_2, P_4, BD_4, S_1
gaussian	2.80	23.51	8.40	$B_1, P_2, BD_1, S_2, A_1, W_1$	B_2, P_4, BD_4, S_1
gru	0.02	0.32	13.25	$B_1, P_4, BD_1, S_1, A_1, W_1$	B_2, P_4, BD_4, S_1
hotspots3D	2.18	15.01	6.88	$B_1, P_4, BD_4, S_2, A_1, W_1$	B_2, P_4, BD_4, S_1
hotspot	0.02	0.27	14.50	$B_1, P_1, BD_4, S_1, A_3, W_1$	B_2, P_4, BD_4, S_1
hwtid	0.48	1.33	2.78	B_2, P_1, BD_1, S_3	B_2, P_4, BD_4, S_1
iso2ddid	0.46	2.88	6.29	$B_1, P_2, BD_4, S_1, A_3, W_1$	B_2, P_4, BD_4, S_1
jaccard	122.35	291.54	2.38	$B_1, P_4, BD_4, S_1, A_4, W_1$	B_2, P_4, BD_4, S_1
lanczos	3.71	17.45	4.71	$B_1, P_1, BD_2, S_1, A_5, W_1$	B_2, P_4, BD_4, S_1
lavaMD	0.19	1.18	6.13	B_2, P_3, BD_1, S_3	$B_1, P_2, BD_2, S_1, A_1, W_1$
libor	1.89	5.00	2.65	B_2, P_3, BD_1, S_3	B_2, P_4, BD_4, S_1
lud	0.63	3.23	5.13	$B_1, P_2, BD_2, S_2, A_3, W_1$	B_2, P_4, BD_4, S_1

likewise, OpenCL-specific variables are included in the search only when `ACPP_VISIBILITY_MASK = ocl`.

Table 7: Summary of observations on search algorithms, compiler flags, and runtime parameters for SYCL performance optimization

Category	Subcategory	Recommendation	Platform / Context	Rationale
Search Algorithm	–	Use TPE for stable, low-variance improvements	CPU / GPU	Avoids catastrophic regressions; strong exploitation; safer overall.
	–	Use Tabu Search for aggressive exploration (iteration-capped)	Large search spaces; non-production tuning	Finds strong configs but frequently hits very poor ones, especially on GPUs.
Compiler Flags	Beneficial	<code>-ffast-math, -funsafe-math-optimizations</code>	All CPUs / GPUs	Consistently improves performance across benchmarks.
		<code>-fno-builtin</code>	AMD Genoa	Enables 256–512-bit vectorization → up to 3× gains (e.g., <code>dct8x8</code>).
	Harmful	<code>-mllvm -disable-loop-idion-vectorize-all</code>	NVIDIA GPUs	Improves certain kernels (e.g., <code>hwtid</code>).
		<code>-fno-unroll-loops</code>	NVIDIA GPUs, Intel CPUs	Causes extreme GPU slowdowns; degrades Intel CPU performance.
		<code>-mllvm -disable-loop-level-heuristics, -mllvm -disable-i2p-p2i-opt -mllvm -disable-2addr-hack</code>	NVIDIA GPUs AMD Genoa	Major slowdowns (e.g., <code>hotspot</code>). Large degradation when used alone.
Runtime	Backend	Prefer OpenMP backend	CPU	Better auto-vectorization; lower memory; up to 8.9× faster (GRU).
		Use OpenCL backend for memory-bound kernels	CPU	Provides 3–5× improvement (e.g., <code>lavaMD</code>).
	Thread Placement	Prefer threads / cores / sockets	CPU	<code>numa_domains</code> often causes 5–14× regressions.
	Scheduling Policy	Dynamic for irregular workloads; static for uniform kernels	CPU	–
	Thread Binding	compact binding	CPU	Improves cache locality; strong for compute-heavy workloads.
	Wait Policy	Active OpenMP wait policy	CPU	Consistently improves performance; passive often worse than the OpenCL backend.

From our analysis, we observe that when SYCL kernels are offloaded using the OpenMP backend, the choice of wait policy has a substantial impact on performance. In particular, switching from a *passive* wait policy to an *active* wait policy consistently improves runtime across multiple benchmarks. For example, in the `d2q9-bgk` benchmark, we measure up to a 2× speedup when using the active wait policy. Notably, the OpenMP passive wait policy performs worse than the OpenCL backend, whereas the active wait policy surpasses OpenCL performance.

Table 6 reports the best and worst runtime configurations identified during our runtime-level exploration, along with their corresponding execution times and the ratio between the worst and best cases. A notable observation is that for some benchmarks, for example, `fwt`, both the best and worst configurations occur when kernels are offloaded through the OpenCL backend. This behavior is primarily attributable to the use of `numa_domains` as the thread-placement policy, which leads to severe performance degradation.

We summarise our observations from this section in Table 7.

5 Related Work

5.1 General-Purpose Autotuning Frameworks

Early autotuning systems, such as OpenTuner [5], pioneered the idea of exposing a *meta-search space* of search strategies, allowing users to combine genetic algorithms, hill climbing, and Bayesian optimization within a single driver to generate program-specific autotuners. OpenTuner demonstrated strong results, yet its interface remains focused primarily on compiler flags and does not explicitly address the tunability of runtime parameters within programming models. More recently, Scravaglieri *et al.* introduced a holistic design-space exploration framework that spans compiler, runtime, and hardware parameters [26]. Their workflow targets

CPU applications on a single node and does not investigate how autotuning parameters vary across CPUs and GPUs. In contrast, our work focuses on heterogeneous devices through the SYCL programming model and emphasizes the transferability of autotuned parameters across CPU and GPU architectures.

Beyond these, several general-purpose autotuners provide complementary capabilities. *CLTune* targets OpenCL kernels with search over kernel-level parameters (e.g., workgroup size, tiling, unrolling) and supports multiple meta-heuristics [23]. *Kernel Tuner* offers an easy-to-use, search-optimizing autotuner for CUDA/OpenCL/C kernels with a broad set of optimization algorithms and energy-aware options [28]. *Orio* provides annotation-based empirical tuning across C/Fortran/CUDA/OpenCL through code generation and search [15]. Finally, *Active Harmony* focuses on runtime parameter tuning via a client–server model, enabling on-the-fly adaptation [27].

5.2 Multi-Objective Optimization

Several studies have argued that a single optimization objective (e.g., execution time) is insufficient when factors such as energy consumption and memory footprint are also critical. Durillo *et al.* introduced *region-aware* multi-objective tuning for OpenMP programs, employing evolutionary algorithms to trade off energy and performance across code regions [9]. Popov *et al.* extended this concept to NUMA-aware thread, page, and task decompositions [25]. In our work, multi-objective search engines—including Pareto-front Bayesian TPE optimization—are implemented as interchangeable plug-ins within the proposed optimizer layer, enabling flexible exploration of performance trade-offs.

Related multi-objective DSE frameworks include *HyperMapper 2.0*, which handles categorical/ordinal variables, unknown constraints, and user priors for practical design-space exploration [22].

For HPC applications, *GPTune* combines Bayesian optimization with multitask learning to reduce tuning cost across problem sizes and targets [19]. These systems motivate our design but differ in scope: our framework unifies the multi-objective optimization in the context of cross-platform offloading.

5.3 Machine Learning for Application Optimization

The survey by Wang and O’Boyle [29] summarizes a decade of machine-learning-driven compiler optimizations, including pass ordering, phase selection, and code generation. Menon *et al.* [20] applied Bayesian optimization to tune communication parameters in large-scale HPC applications, while Dutta *et al.* [10] employed heterogeneous graph neural networks to model complex performance surfaces. Our search methodology draws upon these insights but emphasizes *explainability*: the ability to attribute performance changes to individual compiler or runtime parameters. While existing methods often obscure the specific contribution of each flag due to the stochastic nature of the search, our tabu search approach provides interpretable insights owing to its iterative exploration mechanism.

Closely related auto-scheduling/auto-tuning systems in compilers include *Milepost GCC*, which learns optimization heuristics for GCC from program features [12]; the *TVM* stack with *AutoTVM* (learning-based cost models) and *Ansor* (auto-scheduler) for tensor program optimization across CPUs/GPUs [8]; and the *Halide* auto-schedulers for imaging pipelines [2, 21]. These systems underscore the importance of portable performance and search-guided optimization. Our work differs in targeting SYCL’s single-source heterogeneous model, jointly tuning compiler flags and runtime parameters, and emphasizing per-parameter attribution.

5.4 Positioning of Our Work

Where prior efforts have predominantly focused on fixed programming models (e.g., OpenMP or CUDA) or isolated levels of the optimization stack, our work unifies *compiler flags*, *runtime parameters*, and *portable programming models* within a single framework. This unified approach facilitates the analysis of autotuning behavior across heterogeneous architectures in a single-source programming context. By integrating state-of-the-art search strategies and emphasizing reproducibility and explainability, our work complements and extends prior autotuning frameworks to address the emerging challenges of portable, heterogeneous computing.

6 Conclusion

This work presented a unified autotuning framework for portable programming models that jointly explores compiler flags and runtime parameters across heterogeneous CPU and GPU architectures. By integrating both Bayesian TPE and a novel Tabu Search implementation into a single workflow, the study demonstrated how different search strategies navigate complex optimization spaces and how their behaviors differ with respect to robustness, variance, and interpretability. Using a broad benchmark suite and multiple backend/device combinations, we systematically quantified the extent to which auto-tuned configurations transfer across hardware platforms.

Our experimental results reveal three key insights. First, compiler flag autotuning yields substantial gains on CPUs—up to 4× on AMD Genoa—while GPUs exhibit far smaller sensitivity to compiler-level changes. Second, although several flags influence performance similarly across CPUs and GPUs, these correlations are neither strong nor universal: flags such as `-fno-built-in` exert a pronounced effect on CPUs but minimal influence on GPUs, while loop-related flags (`-fno-unroll-loops`, `-disable-loop-level-heuristics`) consistently degrade GPU performance. Third, runtime-level parameters can meaningfully affect CPU performance, especially thread placement and scheduling, whereas GPU runtime behavior shows greater stability.

Beyond raw performance, the framework emphasizes explainability. Tabu Search’s iterative neighborhood structure makes it possible to attribute performance changes to individual flags, while ridge regression provides a statistical lens for analyzing TPE’s probabilistic sampling. Together, these methods offer complementary insights into why certain configurations succeed or fail.

Overall, this study underscores that autotuning results in SYCL are not inherently portable across devices. CPU-optimal configurations do not reliably translate to GPU performance, and vice versa. These findings suggest the future of autotuning for portable programming models is still a relatively unknown field. Extending the framework to multi-objective optimization, dynamic workloads, and additional SYCL implementations represents promising avenues for future work.

Acknowledgments

This work has received funding from the Swedish Foundation for Strategic Research (reference number CH19-0048). Resources provided by Chalmers e-Commons enabled the computations. The computations were enabled by resources provided by the National Academic Infrastructure for Supercomputing in Sweden (NAISS), partially funded by the Swedish Research Council through grant agreement no. 2022-06725. The authors thank RIKEN-RCCS, Japan, for access to the RCCS-cloud infrastructure.

References

- [1] 2025. *perf: Linux profiling with performance counters*. <https://perf.wiki.kernel.org/>
- [2] Andrew Adams, Dillon Sharlet, Jonathan Ragan-Kelley, Kayvon Fatahalian, and Ravi Teja Mullanpudi. 2016. Automatically Scheduling Halide Image Processing Pipelines. *SIGGRAPH 2016* (2016).
- [3] Takuya Akiba, Shotaro Sano, Toshihiko Yanase, Takeru Ohta, and Masanori Koyama. 2019. Optuna: A Next-generation Hyperparameter Optimization Framework (KDD ’19). Association for Computing Machinery, New York, NY, USA, 2623–2631. doi:10.1145/3292500.3330701
- [4] Aksar Alpay. 2025. *AdaptiveCPP*. <https://github.com/AdaptiveCpp/AdaptiveCpp.git>
- [5] J. Ansel, S. Kamil, K. Veeramachaneni, J. Ragan-Kelley, J. Bosboom, U.-M. O’Reilly, and S. Amarasinghe. 2014. OpenTuner: An Extensible Framework for Program Autotuning. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation Techniques (PACT ’14)*. Edmonton, AB, Canada, 303–316. doi:10.1145/2628071.2628092
- [6] Darryl A. Beckingsale, Jordan Burmark, Rich Hornung, Terry Jones, William Killian, Casey McKinsey, Rob Neely, and Peter Robinson. 2018. RAJA: Portable Performance for Large-Scale Scientific Applications. In *2018 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*. IEEE, 71–81. doi:10.1109/P3HPC.2018.00011
- [7] Junjie Chen, Ningxin Xu, Peiqi Chen, and Hongyu Zhang. 2021. Efficient Compiler Autotuning via Bayesian Optimization. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. 1198–1209. doi:10.1109/ICSE43902.2021.00110

- [8] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Meghan Cowan, Haichen Shen, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. TVM: an automated end-to-end optimizing compiler for deep learning. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation* (Carlsbad, CA, USA) (OSDI'18). USENIX Association, USA, 579–594.
- [9] J. J. Durillo, P. Gschwandtner, K. Kofler, and T. Fahringer. 2019. Multi-Objective Region-Aware Optimization of Parallel Programs. *Parallel Comput.* 83 (2019), 3–21. doi:10.1016/j.parco.2018.03.010
- [10] A. Dutta, J. Alcaraz, A. Tehrani Jamsaz, E. Cesar, A. Sikora, and A. Jannesari. 2023. Performance Optimization using Multimodal Modeling and Heterogeneous GNN. In *Proceedings of the 32nd International Symposium on High-Performance Parallel and Distributed Computing (HPDC '23)*. Orlando, FL, USA, 45–57. doi:10.1145/3588195.3592984
- [11] H. Carter Edwards, Christian Sunderland, Daniel Porter, and Christopher Amsler. 2014. Kokkos: Enabling Performance Portability Across Manycore Architectures. In *2014 Extreme Scaling Workshop (XSW)*. IEEE, 18–24. doi:10.1109/XSW.2014.5
- [12] Grigori Fursin, Yevgeny Kashnikov, A. W. Memon, et al. 2011. Milepost GCC: Machine Learning Enabled Self-tuning Compiler. *International Journal of Parallel Programming* 39, 3 (2011), 296–327. doi:10.1007/s10766-010-0161-2
- [13] Fred Glover. 1986. Future paths for integer programming and links to artificial intelligence. *Computers & Operations Research* 13, 5 (1986), 533–549. doi:10.1016/0305-0548(86)90048-1 Applications of Integer Programming.
- [14] Thomas Gruber, Jan Eitzinger, Georg Hager, and Gerhard Wellein. 2023. *LIKWID*. doi:10.5281/zenodo.1010559
- [15] Albert Hartono, Boyana Norris, and P. Sadayappan. 2009. Annotation-based empirical performance tuning using Orio. In *Proceedings of the 2009 IEEE International Symposium on Parallel & Distributed Processing (IPDPS '09)*. IEEE Computer Society, USA, 1–11. doi:10.1109/IPDPS.2009.5161004
- [16] Donald E. Hilt, Donald W. Seegrist, United States. Forest Service., and Pa.) Northeastern Forest Experiment Station (Radnor. 1977. *Ridge, a computer program for calculating ridge regression estimates*. Vol. no.236. Upper Darby, Pa. Dept. of Agriculture, Forest Service, Northeastern Forest Experiment Station, 1977. 10 pages. <https://www.biodiversitylibrary.org/item/137258> <https://www.biodiversitylibrary.org/bibliography/68934> — Caption title.
- [17] Khronos inc. 2024. . Retrieved April 18, 2023 from <https://registry.khronos.org/SYCL/specs/sycl-2020/html/sycl-2020.html>
- [18] Khronos inc. 2024. *OpenCL Standard*. Retrieved April 18, 2023 from https://registry.khronos.org/OpenCL/specs/3.0-unified/html/OpenCL_Env.html#validation-rules
- [19] Yang Liu, Wissam M. Sid-Lakhdar, Osni Marques, Xinran Zhu, Chang Meng, James W. Demmel, and Xiaoye S. Li. 2021. GPTune: multitask learning for auto-tuning exascale applications. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Virtual Event, Republic of Korea) (PPoPP '21). Association for Computing Machinery, New York, NY, USA, 234–246. doi:10.1145/3437801.3441621
- [20] H. Menon, A. Bhatele, and T. Gamblin. 2020. Auto-tuning Parameter Choices in HPC Applications using Bayesian Optimization. In *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. New Orleans, LA, USA, 831–840. doi:10.1109/IPDPS47924.2020.00090
- [21] Ravi Teja Mullapudi, Andrew Adams, Dillon Sharlet, Jonathan Ragan-Kelley, and Kayvon Fatahalian. 2016. Automatically scheduling halide image processing pipelines. *ACM Trans. Graph.* 35, 4, Article 83 (July 2016), 11 pages. doi:10.1145/2897824.2925952
- [22] Luigi Nardi, Artur Souza, David Koeplinger, and Kunle Olukotun. 2019. HyperMapper: a Practical Design Space Exploration Framework. In *2019 IEEE 27th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*. 425–426. doi:10.1109/MASCOTS.2019.00053
- [23] Cedric Nugteren and Valeriu Codreanu. 2015. CLTune: A Generic Auto-Tuner for OpenCL Kernels. In *Proceedings of the 2015 IEEE 9th International Symposium on Embedded Multicore/Many-Core Systems-on-Chip (MCSOC '15)*. IEEE Computer Society, USA, 195–202. doi:10.1109/MCSOC.2015.10
- [24] OpenACC Standards Group 2017. *The OpenACC Application Programming Interface* (version 2.6 ed.). OpenACC Standards Group. <https://www.openacc.org/specification>
- [25] M. Popov, A. Jimborean, and D. Black-Schaffer. 2019. Efficient Thread/Page-Parallelism Autotuning for NUMA Systems. In *Proceedings of the ACM International Conference on Supercomputing (ICS '19)*. Phoenix, AZ, USA, 342–353. doi:10.1145/3330345.3330376
- [26] L. Scravaglieri, A. Anciaux-Sedrakian, O. Aumage, T. Guignon, and M. Popov. 2025. Compiler, Runtime, and Hardware Parameters Design Space Exploration. In *2025 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. Milan, Italy, 247–260. doi:10.1109/IPDPS64566.2025.00030
- [27] C. Tapus, I-Hsin Chung, and J.K. Hollingsworth. 2002. Active Harmony: Towards Automated Performance Tuning. In *SC '02: Proceedings of the 2002 ACM/IEEE Conference on Supercomputing*. 44–44. doi:10.1109/SC.2002.10062
- [28] Ben van Werkhoven. 2019. Kernel Tuner: A search-optimizing GPU code auto-tuner. *Future Generation Computer Systems* 90 (2019), 347–358. doi:10.1016/j.future.2018.08.004
- [29] Z. Wang and M. O'Boyle. 2018. Machine Learning in Compiler Optimization. *Proc. IEEE* 106, 11 (2018), 1879–1901. doi:10.1109/JPROC.2018.2817118
- [30] S. Watanabe. 2023. Tree-Structured Parzen Estimator: Understanding Its Algorithm Components and Their Roles for Better Empirical Performance. arXiv:2304.11127. accessed 4 Aug. 2025.