Cross-Architecture Autotuning for Single-Source Heterogeneous Programming Models

Cross-Architecture Autotuning for Single-Source Heterogeneous Programming Models

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

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

Jens Domke
RIKEN
Kobe, Japan
jens.domke@riken.jp

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

Abstract

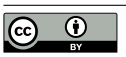
The rise of heterogeneous computing systems has intensified the need for performance-portable programming models and effective autotuning methodologies. Although compiler and runtime tuning are known to significantly influence application performance, it remains unclear how such optimizations transfer 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× 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.



ICS '26, Belfast, United Kingdom

ACM ISBN 979-8-4007-2522-7/26/07
https://doi.org/10.1145/3797905.3800519

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.

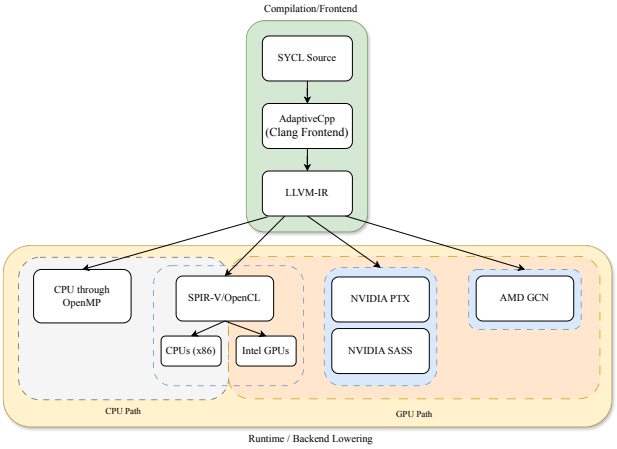


Figure 1: AdaptiveCpp flow with explicit *Compilation* vs. *Runtime* regions and CPU/GPU target paths.

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 Algorithims

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 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.

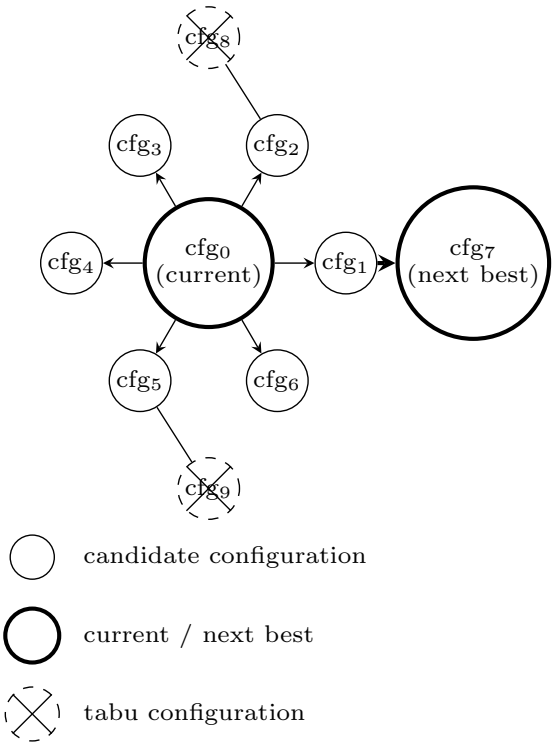


Figure 2: Each node represents a configuration (compiler flags and runtime environment variables). From the current best configuration (cfg_0), the algorithm explores its neighborhood, selects a new best configuration (cfg_7), and avoids revisiting configurations marked as tabu (cfg_8, cfg_9).

This 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.

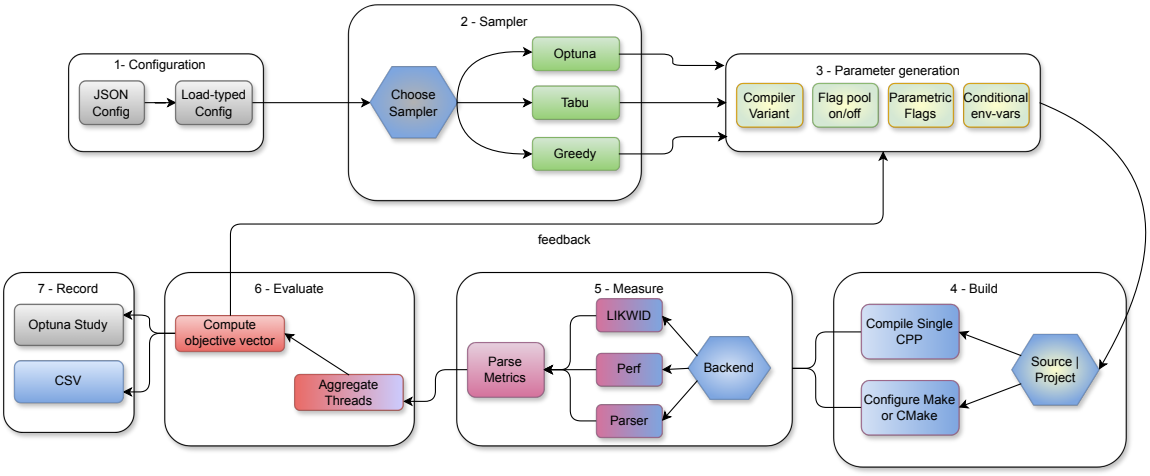


Figure 3: End-to-end workflow of SCOuT.

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

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

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, fdtd3d, 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-2addr-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	ACPP_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 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.

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

	NVIDIA A40			AMD Genoa		
Benchmark	**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
fwt	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
hwt1d	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

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× and 3.7×, 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×, indicating that the method rarely explores configurations that severely degrade performance.

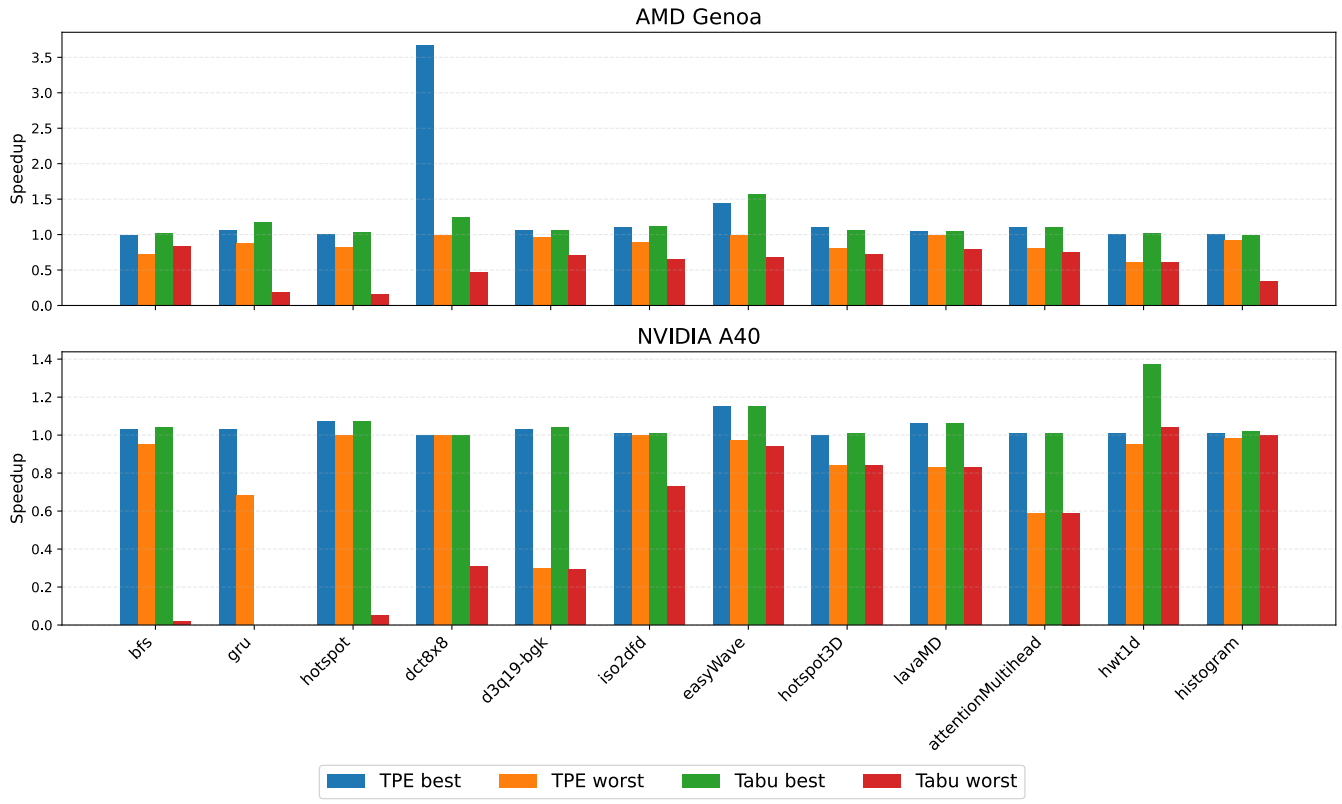


Figure 4: Per-benchmark speedups over the baseline (-O3) SYCL configuration for AMD Genoa and NVIDIA A40 (higher is better).

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×–1.4× over baseline, the worst-case configuration can degrade performance by up to 4× 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-builtin`, 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×—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-builtin` flag, which TPE explored but Tabu did not encounter during its search trajectory. Further analysis shows that `-fno-builtin` 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× 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

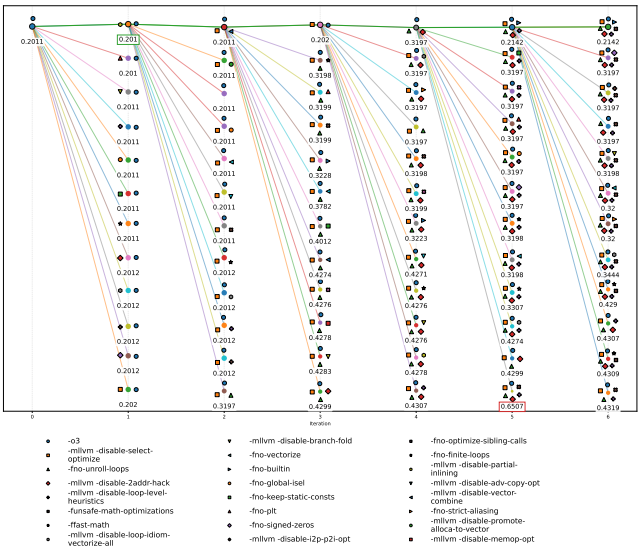


Figure 5: Tabu search path for NVIDIA A40 for benchmark `dct8x8`. Annotations in the figure correspond to the execution time for that particular configuration.

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× performance improvement, while the worst configurations degrade performance by as much as 0.684×. 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× to 1.2×, 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

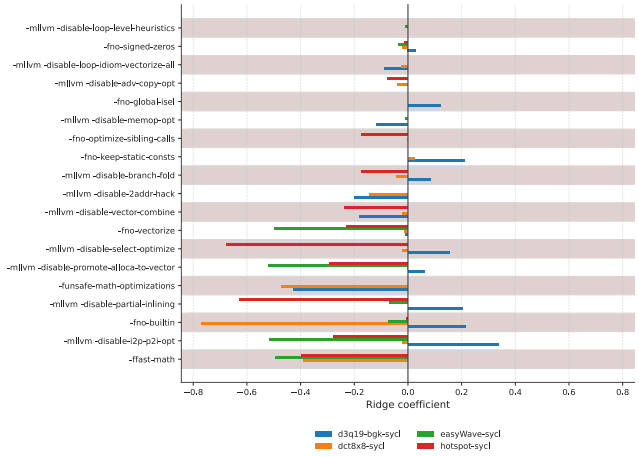


Figure 6: Ridge coefficients for the selected benchmarks on AMD Genoa platform

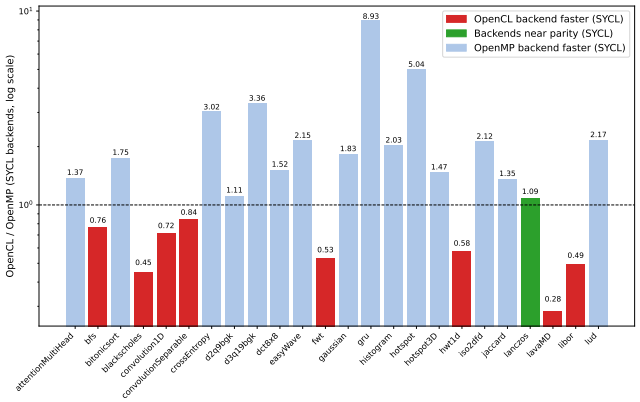


Figure 7: Performance comparison when the SYCL kernels are offloaded with OpenCL and OpenMP backend on AMD Genoa platform

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

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	$B_1, P_1, BD_3, S_1, A_1, W_1$	B_2, P_4, BD_4, S_1
bitonicsort	0.61	2.73	4.45	$B_1, P_2, BD_3, S_2, A_1, W_1$	B_2, P_4, BD_4, S_1
blackscholes	1.13	2.80	2.49	B_2, P_2, BD_3, 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_5, 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	$B_1, P_2, BD_3, S_2, A_1, W_1$	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
dct8x8	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	$B_1, P_3, BD_3, S_1, A_4, W_1$	B_2, P_4, BD_4, S_1
fdtd3d	0.51	1.50	2.95	B_2, P_1, BD_4, S_2	B_2, P_4, BD_1, S_1
fwt	1.01	5.02	4.95	B_2, P_2, BD_4, S_3	B_2, P_4, BD_2, S_1
gaussian	2.80	23.51	8.40	$B_1, P_2, BD_5, S_2, A_1, W_1$	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
hotspot3D	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	$B_1, P_3, BD_4, S_2, A_4, W_1$	B_2, P_4, BD_4, S_1
hwt1d	0.48	1.33	2.78	B_2, P_2, BD_1, S_3	B_2, P_4, BD_1, S_1
iso2dfd	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_4, S_3	$B_1, P_2, BD_5, S_1, A_1, W_1$
libor	1.89	5.00	2.65	B_2, P_3, BD_3, S_3	B_2, P_4, BD_2, 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

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	`-ffast-math`, `-funsafe-math-optimizations`	All CPUs / GPUs	Consistently improves performance across benchmarks.
		`-fno-builtin`	AMD Genoa	Enables 256–512-bit vectorization ⟶ up to 3× gains (e.g., dct8x8).
		`-mllvm -disable-loop-idion-vectorize-all`	NVIDIA GPUs	Improves certain kernels (e.g., hwt1d).
	Harmful	`-fno-unroll-loops`	NVIDIA GPUs, Intel CPUs	Causes extreme GPU slowdowns; degrades Intel CPU performance.
		`-mllvm -disable-loop-level-heuristics`, `-mllvm -disable-i2p-p2i-opt`	NVIDIA GPUs	Major slowdowns (e.g., hotspot).
		`-mllvm -disable-2addr-hack`	AMD Genoa	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., lavaMD).
	Thread Placement	Prefer threads / cores / sockets	CPU	`numa_domains` 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-builtin` 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 CHI19-0048). 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 Mullapudi. 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.10105559

[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 autotuning 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.