HybridSIMD: A Super C++ SIMD Library with Integrated Auto-tuning Capabilities

Haolin Pan[1,2,3], Xulin Zhou[1,3], Mingjie Xing[1*], Yanjun Wu[1,3]
[1]*Institute of Software, Chinese Academy of Sciences, Beijing, China*
[2]*Hangzhou Institute for Advanced Study at University of Chinese Academy of Sciences, Hangzhou, China*
[3]*University of Chinese Academy of Sciences, Beijing, China*
panhaolin21@mails.ucas.ac.cn, {zhouxulin2023, mingjie, yanjun}@iscas.ac.cn

***Abstract*—Single Instruction, Multiple Data (SIMD) technology is crucial for enhancing computational efficiency in High-Performance Computing (HPC). While C++ SIMD libraries abstract away low-level complexities, their proliferation has led to a fragmented set of libraries, creating significant challenges in both performance and usability for developers. To overcome these library-level limitations, this paper introduces a new collaborative concept for SIMD library design. We present HybridSIMD, a C++ library to embody this principle, resolving fragmentation through a unified interface and an operator-level collaborative back-end that leverages the collective strengths of existing libraries. A built-in auto-tuning engine, featuring a hierarchical search strategy, automatically navigates the rich optimization space created by this collaborative approach to deliver maximum performance without manual intervention. Experimental results across six real-world HPC benchmarks on AVX2, AVX512, and NEON architectures demonstrate HybridSIMD's superiority. Notably, the highest speedups achieved are 185.34× on AVX2, 97.80× on AVX512, and 71.32× on NEON, showcasing its effectiveness in resolving fragmentation while delivering state-of-the-art performance. Our artifact is available at https://github.com/Panhaolin2001/HybridSIMD.**

***Index Terms*—SIMD Library, Vectorization, Auto-tuning, High-performance, Library Fragmentation**

I. Introduction

Single Instruction, Multiple Data (SIMD) technology is critical for enhancing computational efficiency through parallel data processing. SIMD finds widespread application in machine learning, data mining, image/video processing, and scientific simulations. SIMD program vectorization employs two main strategies: automatic vectorization [1]–[3] and manual vectorization [4], [5]. Automatic vectorization leverages the compiler's capability to identify and transform scalar operations into vector operations using predefined patterns. In contrast, manual vectorization allows developers to explicitly write low-level assembly code or use intrinsics for fine-grained performance optimization. Both automatic and manual SIMD vectorization approaches have inherent limitations. Automatic vectorization, constrained by the static analysis capabilities of compilers, often fails to fully exploit underlying hardware potential, especially with complex code structures. Manual vectorization, using low-level intrinsics for manual vectorization, directly introduces issues like high development complexity and poor portability.

To mitigate these limitations and address the complexities of direct intrinsic programming, several C++ SIMD libraries have been developed, including VCL [6], HIGHWAY [7], XSIMD [8], TSIMD [9], STD_SIMD [10], and EVE [11]. (Hereafter, SIMD library refers exclusively to these C++ SIMD libraries.) These libraries aim to improve portability and usability by abstracting hardware-specific intrinsics or assembly instructions through a high-level template interface.

However, the evolution of these libraries has also given rise to a notable challenge across these libraries: fragmentation. This fragmentation manifests in two forms that significantly influence developer productivity and application performance: **(1)Performance fragmentation:** No single library achieves consistently high performance across all computational patterns. For example, one library may be advantageous for arithmetic kernels, while another may handle memory operations more efficiently. developers are therefore often required to benchmark and choose libraries on a case-by- case basis to avoid sub-optimal results. **(2)Usability fragmentation:** Each library exposes its own API style and design choices, resulting in a steep learning curve and additional engineering work when comparing, integrating, or migrating between them. Due to performance fragmentation, developers are often forced to adopt multiple SIMD libraries to implement the same kernel so they can select the best-performing version. At the same time, usability fragmentation requires developers to invest significant time learning the unique APIs of each SIMD library. These challenges substantially hinder developer productivity in the pursuit of higher performance.

Addressing this problem requires more than simply publishing yet another standalone library, which would only compound the issue. Instead, it calls for a new design concept: rather than attempting to replace individual libraries, our approach enables them to collaborate with a unified interface. By using this interface, users not only save the cost of learning the unique APIs of each individual SIMD library, but are also able to write a single implementation that leverages the complementary strengths of existing optimized implementations under a unified abstraction.

*Corresponding author

This work was supported by the Strategic Priority Research Program of the Chinese Academy of Sciences, Grant No. XDA0320000 and XDA0320200.

2643-1572/25/$31.00 ©2025 IEEE
DOI 10.1109/ASE63991.2025.00147

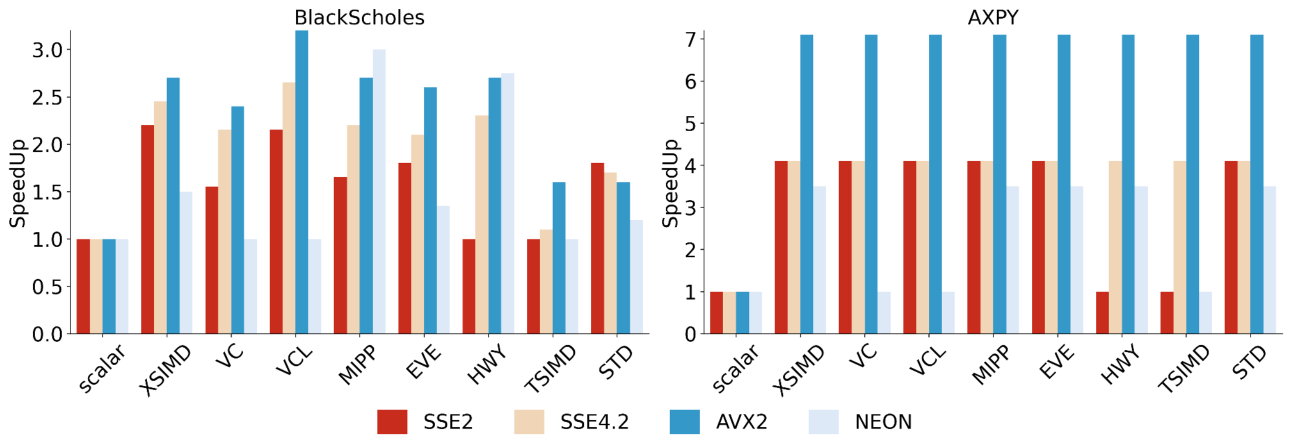


Fig. 1: Comparison of speedup of different SIMD libraries for BlackScholes and AXPY.

TABLE I: Interface differences across various SIMD libraries. Each row represents the interface form of different SIMD libraries.

XSIMD	TSIMD	HIGHWAY	EVE	VCL	STD_SIMD	MIPP
load_aligned	load	Load	load	load_a	copy_from	load
load_unaligned	*N/A*	LoadU	load	load	copy_from	loadu
store_aligned	store	Store	store	store_a	copy_to	store
store_unaligned	*N/A*	StoreU	store	store	copy_to	storeu
select	select	IfThenElse	if_else	select	where	mask
all	all	AllTrue	all	horizontal_and	all_of	Reduction
!none	any	!AllFalse	any	horizontal_or	!none_of	testz
none	none	AllFalse	none	horizontal_none	none_of	!testz
...	...	...	...	...	...	...

To this end, we introduce HybridSIMD, a C++ library that embodys this collaborative concept. HybridSIMD addresses the two forms of fragmentation by offering a unified interface that reduces API diversity for developers, and an operator-level back-end that can integrate multiple SIMD libraries (including others like TSIMD [9], STD_SIMD [10], and EVE [12]) to build a broader search space. This design makes it possible to select the most suitable implementation for each operation and collaborate them into a hybrid solution adapted to the application context.

To explore this enlarged optimization space, HybridSIMD incorporates a built-in auto-tuning engine. This engine applies a hierarchical search strategy to identify effective configurations automatically, thereby reducing the need for manual benchmarking. The collaborative design also enables secondary optimizations, such as flexible handling of vector lengths, which in turn allows more aggressive loop unrolling than what would be feasible with hardware registers alone.

We evaluate HybridSIMD on six representative HPC benchmarks across AVX2, AVX512, and NEON architectures [13]–[16]. Results show that our collaborative approach delivers competitive performance, with speedups of up to 185.34×, 97.80×, and 71.32×, while at the same time addressing usability and integration issues. The contributions of this paper are summarized as follows:

- We introduce a new collaborative concept for SIMD library design to address the systemic fragmentation problem.
- We present HybridSIMD, a novel C++ library that implements this concept through a unified interface and an operator-level collaborative back-end.
- We design a hierarchical auto-tuning strategy that efficiently searches the expanded optimization space created by our collaborative approach.
- We conduct an evaluation showing that our collaborative system mitigates fragmentation and achieves strong performance across diverse architectures and workloads.

II. Motivation

While C++ SIMD libraries offer a valuable abstraction over low-level intrinsics, their proliferation has led to a fragmentation. This fragmentation creates significant, interconnected hurdles for developers, stemming directly from the fundamental pursuit of high performance.

A. Performance Fragmentation

The primary driver for using SIMD is the pursuit of maximum computational performance. However, in this pursuit, developers quickly encounter the challenge of **performance**

fragmentation: no single SIMD library is universally optimal. As depicted in Figure 1, different libraries exhibit vastly different performance characteristics depending on the computational kernel. For a complex program like BlackScholes, the choice of library can lead to dramatic performance swings, whereas for a simpler task such as AXPY, the differences are less pronounced.

This inconsistency forces developers into a dilemma. To guarantee optimal performance, they must engage in a tedious and time-consuming process of benchmarking multiple libraries for each performance-critical section of their application. Choosing a sub-optimal library can mean leaving significant performance on the table, while the process of finding the optimal one introduces substantial engineering overhead.

B. *Usability Fragmentation*

The quest to resolve performance fragmentation directly leads to a second, equally challenging problem: **usability fragmentation**. As developers attempt to compare and contrast different libraries to find the best performer for their specific kernels, they are confronted with a diverse landscape of APIs and design philosophies.

As illustrated in Table I, fundamental operations like memory loads or conditional selections are expressed with different function names, parameter orders, and semantics across each library. This lack of standardization creates a steep learning curve, forcing developers to master the unique interface of each candidate library. The process of writing functionally equivalent code snippets for benchmarking becomes costly and error-prone. Consequently, what begins as a search for performance quickly devolves into a struggle with disparate APIs, severely hindering developer productivity and code maintainability. This tight coupling between performance and usability fragmentation forms a vicious cycle that plagues the current SIMD libraries.

III. Overview of HybridSIMD

HybridSIMD's library is divided into two main components: the front-end and the back-end, as shown in Figure 2. Users first vectorize a program using HybridSIMD's unified interface, which then serves as the input to the library. The overall flow of HybridSIMD is thus divided into these two parts: front-end and back-end.

The front-end begins by conducting static analysis on the user's kernel program, written using HybridSIMD's unified interface, to extract important static information. Based on this analysis, HybridSIMD generates an optimization search space. This space is then pruned by the front-end to eliminate suboptimal configurations, thereby facilitating a more efficient exploration. Subsequently, the back-end explores this refined search space using a hierarchical search method to identify optimal configurations. Upon completion of this exploration, HybridSIMD generates an optimized SIMD executable file tailored for the target device.

IV. HybridSIMD Unified Interface Design

Existing SIMD libraries abstract intrinsic vector types and functions using C++ templates, providing common operations like `load` and `store`. These libraries effectively streamline the use of intrinsics, enhancing developer productivity by offering an abstraction layer. As discussed in Section II-A, they typically provide native vector types (optimized for underlying instruction sets) and customizable types (allowing explicit element counts).

Our design introduces two core vector abstractions, as illustrated in Figure 3, ensuring both scalability and compatibility across different SIMD libraries. The first, `Vector<ElemType>`, encapsulates an intrinsic vector type. For each integrated SIMD library, a dedicated adapter converts HybridSIMD's intrinsic vector type to the library-specific type, enabling operations through that library's interface, which are ultimately executed on the intrinsic vector. The second abstraction, `Vectors<ElemType, N>`, leverages Clang Builtin Vector Types, allowing users to specify the number of elements `N`. This type aggregates multiple SIMD registers into a contiguous memory space, with each sub-register handled by a library-specific adapter. Operations on `Vectors<ElemType, N>` simulate inner loop unrolling by iteratively applying underlying SIMD library interfaces, mirroring the behavior of automatic loop unrolling. This

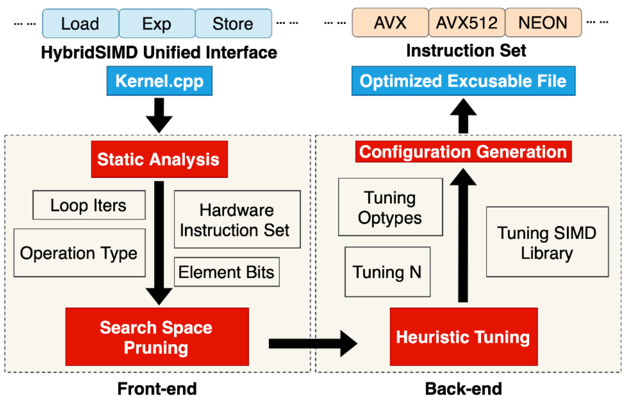


Fig. 2: Overview of HybridSIMD library.

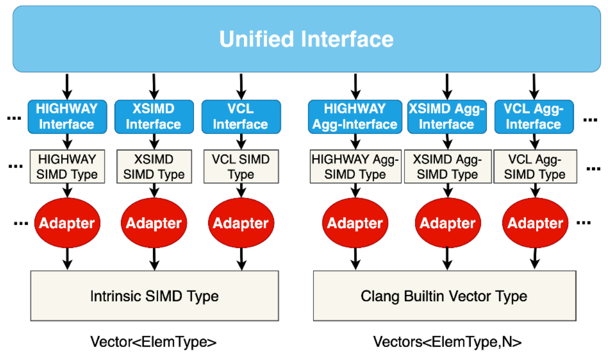


Fig. 3: Design structure of HybridSIMD unified interface.

scalable and compatible design seamlessly integrates with various SIMD libraries while adapting to different vectorization strategies, directly addressing the interface fragmentation issue from Section II-B.

Our adapter mechanism is crucial for enabling interoperability between different SIMD libraries and for extending their capabilities, particularly for those that do not natively support variable vector element counts. For libraries like VCL and XSIMD, which traditionally require manual selection of vector types based on register length (e.g., VCL's `VecXf` for 128, 256, or 512-bit registers, or XSIMD's instruction set tags), our adapter automates this selection. It chooses the optimal library-specific vector type based on the machine's instruction set, eliminating manual specification. Furthermore, when the user-specified `N` for `Vectors<ElemType, N>` exceeds the element capacity of the current optimal instruction set, the adapter seamlessly combines multiple smaller vector types to form a contiguous vector type that matches the required `N`. For other SIMD libraries that already provide vector types with customizable element counts, HybridSIMD seamlessly integrates them. This adapter-based encapsulation is fundamental to HybridSIMD's ability to pick and choose the most performant implementations from different libraries at a fine-grained level, a capability not possible by merely extending a single library.

Figure 4 provides an example that illustrates the typical usage of our `Vectors<ElemType, N>` type compared to STD_SIMD's `fixed_size_simd<ElemType, N>`. Although both share similar underlying logic, HybridSIMD's interface employs three template parameters for each operation: element type, vector element count, and the SIMD library type (denoted by `AUTO` for auto-tuning). This unified interface supports a comprehensive range of SIMD operations, including element extraction, arithmetic operations, logical and bitwise operations, comparison and masking, memory operations, and standard operator overloading.

```
for(int i=0; i<I; i++){
  for(int j=0; j<J; j+=ElementCount){
    fixed_size_simd<ElemType, N> a,b,c;
    a.copy_from(&A[i][j],element_aligned_tag());
    b.copy_from(&B[i][j],element_aligned_tag());
    c = a * b;
    c.copy_to(&C[i][j],element_aligned_tag());
  }
}
```

(a) STD_SIMD

```
for(int i=0; i<I; i++){
  for(int j=0; j<J; j+=ElementCount){
    Vectors<ElemType, N> a,b,c;
    LoadAligned<ElemType, N, AUTO>(a,&A[i][j]);
    LoadAligned<ElemType, N, AUTO>(b,&B[i][j]);
    c = Mul<ElemType, N, AUTO>(a,b);
    StoreAligned<ElemType, N, AUTO>(c,C[i][j]);
  }
}
```

(b) HybridSIMD

Fig. 4: Code examples of STD_SIMD and HybridSIMD interface.

To illustrate the underlying mechanism for our `Vector<ElemType>` type, the `LoadAligned` operation uses the optimal instruction set for the current machine, as shown in Equation (1):

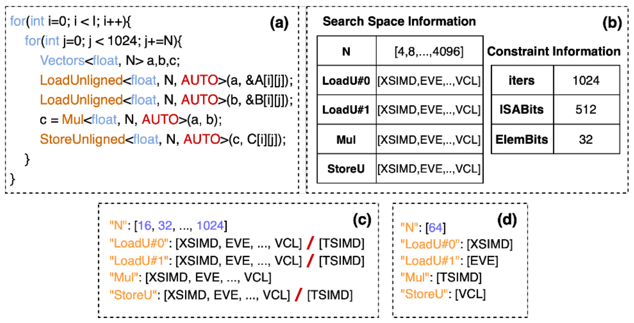
$$\text{Register}^{\mathit{Impl}}[i] = \text{LoadAligned}^{\mathit{Impl}}(\text{Mem}[\mathit{addr} + i \cdot s]) \\ (\mathit{addr} \pmod{a} = 0) \tag{1}$$

For `Vectors<ElemType, N>`, where `N` specifies the number of elements, this type allows using interfaces from various SIMD libraries while simulating automatic loop unrolling, crucial for computationally intensive tasks. When `N` multiplied by `sizeof(ElemType)` exceeds the register size of the current machine's optimal instruction set, loop unrolling is activated to fully utilize available SIMD registers. The implementation for this `LoadAligned` operation is described by Equation (2):

$$\text{Register}^{\mathit{Impl}}[i] = \text{Aggregate}_{j=0}^{N/N'-1} \\ \Big(\text{LoadAligned}^{Impl,N'}(\text{Mem}[\mathit{addr} + (j \cdot N' + i) \cdot s])\Big) \\ \text{where } (\mathit{addr} \pmod{a} = 0), \text{ for } 0 \le i < N, \text{ if } N > N' \tag{2}$$

Here, `N'` represents the longest SIMD register size for the current machine, and the operation aggregates multiple smaller `LoadAligned` operations of size `N` to process the larger data vector of size `N'`.

V. Front-end Analysis and Search Space


(a)

```
for(int i=0; i < I; i++){
  for(int j=0; j < 1024; j+=N){
    Vectors<float, N> a,b,c;
    LoadUnligned<float, N, AUTO>(a, &A[i][j]);
    LoadUnligned<float, N, AUTO>(b, &B[i][j]);
    c = Mul<float, N, AUTO>(a, b);
    StoreUnligned<float, N, AUTO>(c, C[i][j]);
  }
}
```

(b)

Search Space Information

N	[4,8,...,4096]
LoadU#0	[XSIMD,EVE,..,VCL]
LoadU#1	[XSIMD,EVE,..,VCL]
Mul	[XSIMD,EVE,..,VCL]
StoreU	[XSIMD,EVE,..,VCL]

Constraint Information

iters	1024
ISABits	512
ElemBits	32

(c)

```
"N": [16, 32, ..., 1024]
"LoadU#0": [XSIMD, EVE, ..., VCL] / [TSIMD]
"LoadU#1": [XSIMD, EVE, ..., VCL] / [TSIMD]
"Mul": [XSIMD, EVE, ..., VCL]
"StoreU": [XSIMD, EVE, ..., VCL] / [TSIMD]
```

(d)

```
"N": [64]
"LoadU#0": [XSIMD]
"LoadU#1": [EVE]
"Mul": [TSIMD]
"StoreU": [VCL]
```

Fig. 5: Static analysis and search space generation for a double-loop multiplication program. (a) shows a double-loop multiplication example written using the HybridSIMD unified interface, where *AUTO* acts as a placeholder for any SIMD library. (b) shows the search space information and constraint information. The search space information is used to construct the full set of potential configurations for exploration, while (c) the constraint information helps prune the search space to eliminate suboptimal configurations. (d) represents a potential configuration within the generated search space.

HybridSIMD's front-end performs static analysis on the input SIMD program and constructs a pruned search space for optimization. This process efficiently narrows down potential configurations, focusing on promising strategies for performance enhancement.

A. Static Analysis

To optimize vectorized programs written using the HybridSIMD unified interface, static analysis is conducted to gather essential information for constructing the optimization search space. Key details extracted include the total iteration count of the innermost loop (*iters*), the range of available vector element counts (*N*), the bit width of a single element (*ElemBits*), the bit width of the highest supported SIMD instruction set (*ISABits*), and the types of operations present in the program (*optypes*, e.g., addition, multiplication, load, store, etc.).

Figure 5 illustrates this process for a double-loop multiplication program. Figure 5(a) shows the program using HybridSIMD's unified interface, with `AUTO` as a placeholder for any SIMD library to enable back-end auto-tuning. Figure 5(b) presents the derived search space information, used to enumerate potential configurations, and constraint information, which aids in pruning suboptimal options. Figure 5(d) depicts a potential configuration from this generated space.

To enhance the scalability of our static analysis and search space construction, we have developed specific optimizations for SGEMM (Single-Precision General Matrix Multiplication), a critical deep learning workload. SGEMM involves three nested loops with diverse unrolling strategies (e.g., outermost loop unrolling or combining inner loops). HybridSIMD incorporates SGEMM-specific unrolling templates, fully compatible with our unified interface. During static analysis for SGEMM, we augment the search space with a new component: the type of loop unrolling template, expanding the optimization opportunities tailored for matrix multiplication.

B. Search Space Pruning

The initial search space is generated by enumerating and combining all elements derived from the static analysis, as depicted in Figure 5(b). This exhaustive space can be represented by the product of options for each parameter:

$$\text{Search Space} = \prod_{i=1}^{n} Options_i \tag{3}$$

where $Options_i$ refers to possible choices for parameters such as the number of elements (*N*) and the specific SIMD library type assigned to each operation. It is important to note that this formula represents the theoretical upper bound of the search space. In practice, our strategic pruning process significantly reduces the number of configurations that need to be evaluated.

To ensure efficient exploration, HybridSIMD applies a strategic pruning process to significantly reduce the size of this search space. This involves two main steps:

Firstly, pruning is applied based on disabled operations. A global `disabledOps` dictionary is maintained within the HybridSIMD library. This dictionary is pre-configured by the library developers as a one-time setup task and is not intended for modification by the end-user. It maps specific operation types (*optypes*) to lists of SIMD library types known to be either unsupported or consistently inefficient for those operations(e.g., `disabledOps=LoadUnaligned:[TSIMD], Exp:[EVE,VCL]`). The entries are determined in two ways: (1) unsupported operations, identified from library documentation and confirmed through testing (e.g., TSIMD's unaligned load/store), and (2) low-performance operations, identified by benchmarking and flagging outlier operations with high execution time.

During search space construction, if an operation in the user's vectorized program is listed in `disabledOps`, any configurations involving the corresponding disabled SIMD library types for that operation are immediately excluded. For instance, as shown in Figure 5(c), configurations utilizing TSIMD for unaligned load/store operations are pruned due to TSIMD's lack of native support. Similarly, for complex mathematical functions like `Exp`, if EVE or VCL implementations are known to degrade performance (e.g., due to higher precision but slower execution), configurations using them for `Exp` are also discarded.

Secondly, pruning is performed based on vector element count conditions. For the set of possible vector element counts *N*, values are filtered against two criteria involving the inner loop iteration count (*iters*), the element bit width (*ElemBits*), and the highest supported SIMD instruction set bit width (*ISABits*). Values of *N* are eliminated if they either exceed *iters* or fall below $\frac{ISABits}{ElemBits}$. This is motivated by two key observations: 1) If *N* is greater than *iters*, the inner loop becomes more than fully unrolled, leading to redundant operations and degraded performance. 2) If *N* is smaller than $\frac{ISABits}{ElemBits}$, the SIMD registers are underutilized, preventing the processor from achieving its full potential. These combined pruning strategies effectively narrow down the search space, allowing HybridSIMD to efficiently identify optimal configurations while avoiding suboptimal or infeasible ones.

VI. Back-end Exploration

HybridSIMD's back-end efficiently explores the search space generated by the front-end to produce an optimized executable. Despite front-end pruning, the search space can remain vast (potentially up to 10^{10} for programs with many operations). To accelerate optimization, we employ a hierarchical search strategy, prioritizing parameters with the greatest performance impact. The search space S primarily consists of two components: the number of vector elements (*N*) and the operation types (*optypes*). Given *M* operations in *optypes*, *E* available SIMD library types, and *n* possible values for *N*, the total search space is $O = nE^M$.

A. Heuristic Tuning

Our hierarchical search approach, outlined in Algorithm 1, proceeds through three stages, prioritizing parameters with the most significant performance impact. Empirical results (as discussed in Section VII-B4) indicate that the number of vector elements (*N*) has the greatest impact on performance, with

SIMD library types playing a secondary role; thus, optimizing *N* is our primary focus.

In **Stage 1 (Tuning *S.N*)**, all operations in *optypes* are randomly assigned to a SIMD library, and *S.N* is tuned. The TopK method identifies the top *K* values of *S.N* that yield the best performance, forming $S.N_{\text{top}}$. For **Stage 2 (Coarse-Grained SIMD Library Tuning)**, with the top *N* values from Stage 1 fixed, all `AUTO` SIMD library types within the program's operations are uniformly set to a single candidate library to evaluate overall performance for each library. The TopK method then selects the top *K'* most suitable single SIMD libraries, forming E_{top}. While this first two stages may potentially exclude the globally optimal configuration, it serves as a crucial trade-off to enhance the search speed. Finally, in **Stage 3 (Fine-Grained Mixed SIMD Library Tuning)**, a genetic algorithm is employed for detailed optimization. The initial population comprises combinations derived from the best configurations of the first two stages. Each individual is a dictionary mapping gene types to potentially different SIMD library types for each operation. The algorithm refines mixed library assignments via crossover and mutation on a population of 64 individuals per generation. Fitness is measured by execution speedup over the scalar baseline. Evolution runs for 10 generations, evaluating 640 configurations in this fine-grained optimization stage.

Each stage operates within its own subspace, effectively reducing the overall search complexity. The total optimized search space is $O' = K + K' + O_{\text{stage3}}$, where O_{stage3} is significantly smaller than E^M. This hierarchical approach, by prioritizing parameters based on their performance impact, substantially reduces the search space and accelerates the optimization process.

Algorithm 1 Hierarchical Search Approach

```
1: Input: Search space $S$, Set of SIMD Libraries $L$
2: Output: Optimized configuration
3: Stage 1: Tuning $S.N$
4: for each candidate $S.N_i$ in $S.N$ do
5:     Evaluate performance for $S.N_i$
6: end for
7: Select top $K$ values of $S.N_i$ with best performance: $S.N_{\text{top}}$
8: Stage 2: Coarse-Grained SIMD Library Tuning
9: for each candidate library $L_j$ in $L$ do
10:     Evaluate performance by setting all operations to $L_j$
11: end for
12: Select top $K'$ libraries with best performance: $L_{\text{top}}$
13: Stage 3: Fine-Grained Mixed SIMD Library Tuning
14: Initialize population with combinations of $S.N_{\text{top}}$ and $L_{\text{top}}$
15: while termination condition not met do
16:     for each individual in population do
17:         Evaluate fitness (performance)
18:     end for
19:     Select parents for crossover
20:     Perform crossover and mutation to generate offspring
21:     Replace population with new generation
22: end while
23: Return: Best configuration found
```

B. Configuration Generation

Generated configurations are represented as a dictionary, mapping C++ source code operations to specific SIMD library types and the selected vector element count *N*. Applying these configurations directly involves iterating through the dictionary: for each key-value pair, the designated SIMD library type and vector element count *N* replace placeholder values in the C++ code. This transformation converts each optimized configuration into executable C++ code, which is then used for performance benchmarking.

VII. Experiments

A. Experiments Setup

a) Platform: HybridSIMD was evaluated on x86 AVX2, x86 AVX512, and ARM NEON architectures using LLVM 16.0.6 with C++20. Performance was measured using Google Benchmark to compute speedup relative to a scalar O3 baseline. Consistent compiler flags, `-march=native` and `-O3`, were applied across all platforms for optimal architecture-specific optimization.

b) Benchmarks: To demonstrate HybridSIMD's effectiveness, we selected six diverse real-world HPC benchmarks: BlackScholes, Mandelbrot, Julia, NewtonFractal, Quadratic, and SGEMM. These benchmarks are widely recognized as classic test cases for evaluating SIMD performance [17]–[21]. They collectively cover a rich variety of computational patterns and SIMD operations, including mathematical functions, arithmetic, memory access, reductions, and conditional logic. This diverse selection ensures a thorough assessment of HybridSIMD's capabilities across different application domains like finance, mathematics, and physics.

c) Baseline: HybridSIMD was compared against Clang O3 (as a scalar baseline) and several state-of-the-art SIMD libraries: XSIMD [8], TSIMD [9], EVE [11], VCL [6], HIGHWAY [7], MIPP [22], and STD_SIMD [10] (VCL and TSIMD lack ARM support). Our auto-tuning back-end integrates five of these for optimization—XSIMD, TSIMD, EVE, VCL, and HIGHWAY—while MIPP and STD_SIMD were included for a broader comparison. Each competing SIMD library was evaluated at its optimal performance for each benchmark (e.g., EVE selecting between `wide<ElemType>` and `wide<ElemType, fixed<N»`). Benchmarks written with our unified interface (Figure 4) maintained consistent underlying code logic for fair comparison.

d) Correctness Validation: Program correctness was verified by comparing SIMD output with scalar implementations. A program was deemed correct if the maximum relative error did not exceed 10^{-4}. Minor numerical precision differences in floating-point operations, common in SIMD due to parallel

TABLE II: Speedup comparisons of HybridSIMD and other SIMD libraries across six benchmarks on AVX2, AVX512, and NEON, relative to a scalar O3 baseline. For a fair comparison, each competing library (e.g., EVE) was also evaluated at its peak performance, using its optimal vector count `N` where applicable. The *Our (Untuned)* column shows the best performance achieved with an optimal vector count `N` and a single, uniformly applied SIMD library (the result of Stages 1-2). The *Our (Tuned)* column shows the final performance after our full three-stage search, which enables fine-grained, mixed-library optimization.

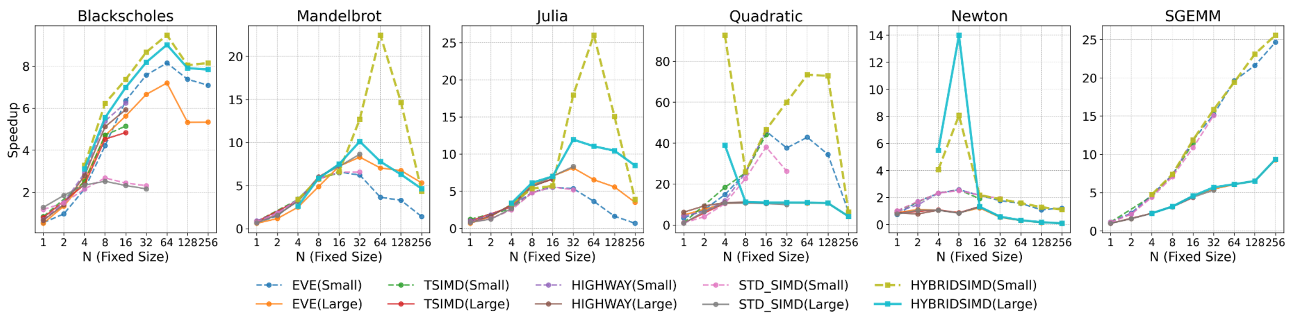
BM	ISA	Size	Our (Untuned)	Our (Tuned)	XSIMD	HWY	TSIMD	EVE	VCL	STD	MIPP
BlackScholes	NEON	large	2.66	**4.07**	1.38	1.60	1.00	2.08	1.00	1.44	1.36
		small	2.74	**4.02**	1.34	1.61	1.00	2.10	1.00	1.46	1.35
	AVX2	large	8.88	**15.05**	5.66	4.31	4.41	7.04	4.52	2.01	4.08
		small	6.61	**19.73**	4.92	3.86	3.91	5.58	3.81	1.55	3.37
	AVX512	large	9.02	**19.86**	6.71	5.93	4.89	7.20	6.74	2.52	4.33
		small	9.48	**18.87**	7.19	6.27	5.19	8.15	7.18	2.68	4.60
Mandelbrot	NEON	large	5.33	**5.35**	2.64	2.66	1.00	5.29	1.00	5.18	2.68
		small	7.97	**8.00**	2.47	2.49	1.00	4.19	1.00	4.04	2.51
	AVX2	large	12.93	**13.90**	6.17	6.30	6.14	10.82	6.15	8.65	6.11
		small	69.91	**122.54**	5.46	5.55	5.50	8.28	5.50	6.80	5.41
	AVX512	large	10.64	**16.48**	6.84	7.48	7.37	8.30	5.84	8.64	7.23
		small	22.47	**30.62**	5.95	6.66	6.48	6.57	5.63	6.71	6.66
Julia	NEON	large	5.43	**5.44**	1.52	1.50	1.00	3.08	1.00	3.95	1.43
		small	70.72	**71.32**	1.14	1.13	1.00	1.68	1.00	2.09	1.07
	AVX2	large	15.02	**16.75**	5.14	5.11	5.14	7.91	5.06	5.05	5.19
		small	126.13	**185.34**	3.30	3.32	3.31	3.81	3.27	3.27	3.32
	AVX512	large	11.95	**15.91**	7.05	7.04	7.14	8.13	6.84	8.33	6.67
		small	26.02	**29.57**	5.39	5.61	5.71	5.52	5.75	5.57	5.61
NewtonFractal	NEON	large	2.52	**2.56**	2.07	2.43	1.00	2.43	1.00	2.40	2.39
		small	3.40	3.42	2.80	3.26	1.00	**4.06**	1.00	3.77	3.22
	AVX2	large	3.16	**3.53**	2.61	2.46	2.45	3.20	2.61	2.72	2.60
		small	5.52	**6.11**	5.06	4.84	4.81	5.10	4.91	5.06	5.02
	AVX512	large	14.01	**15.89**	1.30	1.33	1.33	1.27	1.32	1.30	1.21
		small	8.08	**9.69**	1.98	2.58	2.57	2.58	2.01	2.51	2.08
Quadratic	NEON	large	2.67	**18.10**	2.34	2.50	1.00	3.11	1.00	2.01	2.10
		small	2.85	**6.58**	2.24	2.39	1.00	3.34	1.00	2.03	2.01
	AVX2	large	28.08	**29.40**	27.87	28.08	27.98	27.98	27.77	27.45	27.66
		small	15.92	**27.68**	4.23	13.96	14.02	20.37	4.20	10.19	4.22
	AVX512	large	39.04	**41.56**	10.91	11.23	11.01	10.91	10.94	10.94	10.91
		small	92.73	**97.80**	43.97	46.15	44.16	45.74	42.15	41.98	44.54
SGEMM	NEON	large	4.20	**10.94**	3.02	3.21	1.00	4.13	1.00	3.91	3.18
		small	7.73	**18.56**	3.30	3.48	1.00	7.31	1.00	7.47	3.76
	AVX2	large	22.11	**30.80**	4.27	4.27	4.22	21.45	4.27	11.65	4.24
		small	25.80	**48.06**	5.30	5.27	5.30	21.78	5.32	11.32	5.30
	AVX512	large	9.37	**19.97**	4.58	4.56	4.39	9.28	4.55	5.33	4.51
		small	25.58	**26.49**	11.61	11.85	11.45	24.65	11.75	15.08	11.68

execution and reordered computations, are considered acceptable given the substantial performance benefits.

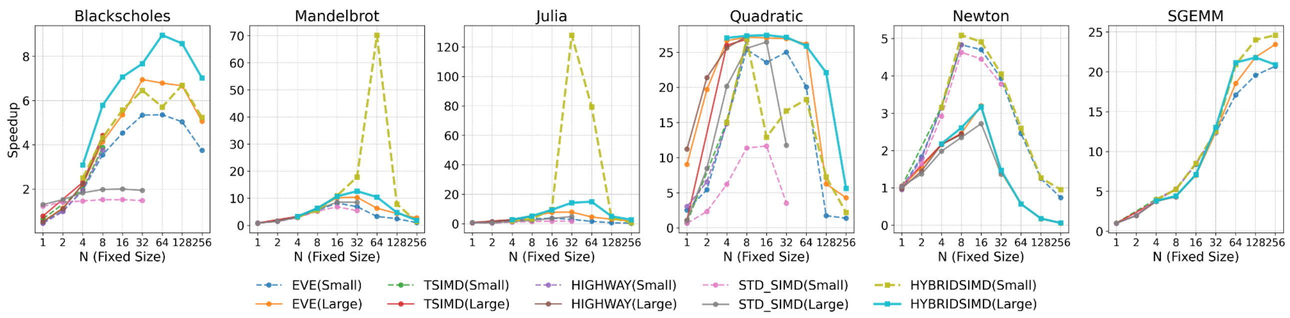
B. Experiment Results

1) Main Results: Our experiments demonstrate that HybridSIMD consistently outperforms state-of-the-art SIMD libraries across all six real-world HPC benchmarks on AVX2, AVX512, and NEON architectures, for both small and large data sizes, as detailed in Table II. Notably, in benchmarks like Julia with smaller inputs, our tuned method achieves a remarkable speedup of **185.34**$\times$ on AVX2, far surpassing other approaches. It is also worth noting that in the case of NewtonFractal with small input sizes on the ARM NEON architecture, the performance of HybridSIMD is lower than that of EVE. This can be attributed to the complex call structure of the NewtonFractal benchmark, which has a relatively small portion that can be vectorized. Additionally, NEON's vector element count is restricted to only 4 float types, further limiting the optimization potential. The design of HybridSIMD, where the adapter imposes conversions, may not be fully optimized under the O3 optimization level on ARM platforms, leading to some redundancy. Despite this, HybridSIMD still largely surpasses other SIMD libraries on NEON for this specific case and consistently outperforms all others across other platforms, demonstrating the overall effectiveness of our approach.

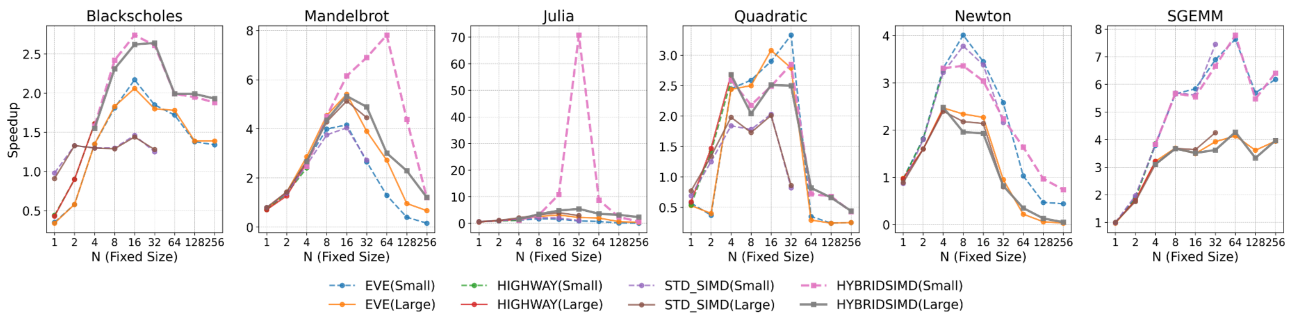
2) Auto-Tuning Execution Time: The time required for auto-tuning varies across different machines. Our evaluation was conducted on an AVX2-compatible AMD EPYC 7713 64-Core Processor running Ubuntu 22.04 LTS. On average,



(a) AVX512



(b) AVX2



(c) NEON

Fig. 6: Performance speedup comparison of HybridSIMD (non-tuned configuration) against other SIMD libraries across varying vector element counts (N), different SIMD instruction sets (AVX512, AVX2, NEON), and both large and small input sizes. The x-axis represents the vector element count N, while the y-axis shows the speedup relative to the scalar O3 baseline. Each subplot corresponds to a specific instruction set, illustrating how performance changes with N for each benchmark and library, revealing optimal N values and performance trends.

each benchmark was executed approximately **662 times** during the auto-tuning process, with an average execution time of around **187.5 seconds**. It is important to note that the tuning time can vary depending on the input size and hardware specifications. These variations are expected, as different architectures and workloads influence the search space and optimization efficiency. Our experimental findings consistently highlight the importance of auto-tuning for achieving optimal SIMD performance. The variations in speedups observed across different workloads, instruction set architectures, and input sizes underscore that performance is highly context-dependent. This variability demonstrates that effective auto-tuning, as implemented in our library, is beneficial to navigate this complex performance landscape.

3) Validation of Hierarchical Tuning Strategy: To validate our methodological choices, we compared our hierarchical search strategy against three standard tuning algorithms: a one-pass **Greedy** search, a **Pure Genetic** algorithm (without our staged initialization), and **Random Search**. The comparison, shown in Table III, evaluates the final speedup achieved on the AVX2 architecture and the average tuning time for each method.

```
// Core computation in a simple loop
do {
  Vec x = zr * zr - zi * zi + cr;
  Vec y = 2.0f * zr * zi + ci;
  zr = Select(m, x, zr);
  zi = Select(m, y, zi);
  // ...
} while (...);
```

(a) HybridSIMD: High-Level C++ Source

```
; Operations on a single, wide vector
%zr_sq = fmul <32 x float> %zr, %zr
%zi_sq = fmul <32 x float> %zi, %zi
%sub = fsub <32 x float> %zr_sq, %zi_sq
%add = fadd <32 x float> %sub, %cr
...
```

(b) HybridSIMD: Lowering to Unified LLVM IR

```
<...bab0>:
  ; Work for 4 native iterations fused
  ; into one large loop body.
  vmulps %ymm11, ... ; iter 1
  ...
  vmulps %ymm4, ...  ; iter 2
  ...
  vmulps %ymm5, ...  ; iter 3
  ...
  vmulps %ymm10, ... ; iter 4
  ...
  ; A single jump for all 4 sets of work
jne <...bab0>
```

(c) HybridSIMD: Optimized & Unrolled Assembly

```
<...78f0>:
  vmulps %ymm14, %ymm14, %ymm2
  vmulps %ymm12, %ymm12, %ymm6
  inc    %r12
  vsubps %ymm6, %ymm2, %ymm2
  vaddps %ymm14, %ymm14, %ymm6
  vmulps %ymm6, %ymm12, %ymm6
  vaddps %ymm2, %ymm4, %ymm2
  vaddps %ymm6, %ymm5, %ymm6
  vblendvps %ymm13, %ymm2, %ymm14, ...
  ...
  ; Frequent jump after each vector's work
  jne    <...78f0>
```

(d) Highway: Conventional Assembly Loop

Fig. 7: The compilation journey and competitive comparison for the Julia benchmark. HybridSIMD's high-level C++ abstraction (a) is cleanly lowered to a unified wide-vector IR (b), which enables the compiler to generate a massive, deeply unrolled assembly loop (c). This contrasts with the conventional high-frequency loop generated by libraries like Highway (d).

TABLE III: Speedup (×) and Avg. Tuning Time (s) Comparison on AVX2.

Algorithm	**Time(s)**	**BS**	**JU**	**MA**	**NF**	**QD**	**SG**
Our Hierarchical	187.5	**19.7**	**185.3**	122.5	**6.1**	27.7	**48.1**
Greedy	608.8	8.9	101.0	116.0	5.3	12.1	23.6
Pure Genetic	**147.5**	8.4	75.9	53.5	5.3	22.7	9.1
Random Search	415.8	15.5	136.4	**124.7**	5.5	**28.4**	17.4

The results confirm the effectiveness of our approach. Greedy search is both slow and prone to local optima. Pure Genetic search, while fast, fails to effectively explore the vast search space without our intelligent initialization. Although Random Search can occasionally find good solutions, its performance is inconsistent. In contrast, our hierarchical search consistently discovers top-tier configurations while maintaining a competitive average tuning time, thus striking the best balance between search efficiency and the quality of the final optimized code.

4) Cross-Architecture SIMD Performance with Variations in N: This experiment validates our solution (Section II-A), which addresses the limitations of existing SIMD libraries in handling varying vector element counts based on N. Our experimental results show differing levels of support for variations in vector type based on N across various SIMD libraries. Specifically, EVE, TSIMD, HIGHWAY, and STD_SIMD provide this flexibility. Among these, STD_SIMD has a limitation, supporting a maximum of only 32 elements. The adaptability of TSIMD and HIGHWAY's vector element count is constrained by the register width of the underlying instruction set; for example, under AVX2, they can only handle vectors of up to 8 32-bit elements. In contrast, both EVE and our HybridSIMD library can handle larger vector element counts that exceed the native register width.

Figure 6 illustrates the non-tuned performance across varying N values. A consistent trend emerges: speedup generally increases with N until a certain threshold, beyond which it plateaus or declines. The optimal N for peak speedup is benchmark-specific; for instance, on AVX2, Blackscholes and Mandelbrot peak at N=64, while Julia, Quadratic, and NewtonFractal achieve maximum speedups at N=32, N=16, and N=8, respectively. SGEMM shows continuous speedup up to N=256, indicating a wider optimal range. This variability highlights that the ideal vector element count depends on workload computational complexity, code characteristics, the specific SIMD library, and input size. The graphs confirm that different SIMD libraries yield distinct optimal N values, and even within the same SIMD library, optimal N can vary with input size. HybridSIMD's adaptive selection of N ensures robust high performance across varying input scales, exemplified by a 126× speedup at N=32 for small Julia inputs on AVX2. Comparing instruction sets, AVX2 and AVX512 generally offer better speedups than NEON, primarily due to NEON's limitation of four concurrent floating-point operations. However, for certain workloads like Julia (NEON/AVX2 exceeding AVX512) and Mandelbrot with small inputs (AVX2 exceeding AVX512), a lower vector element count can surprisingly yield higher performance. This demonstrates that optimal vector element count is indeed workload-dependent, and a larger *N* does not always guarantee better performance. This intricate interplay of *N*, workload, architecture, and input size underscores the critical necessity of auto-tuning methodologies like HybridSIMD to adapt SIMD parameters for maximal efficiency.

C. Case Study

To explain the 185.34× speedup on the Julia benchmark, we trace the compilation journey of its core computational loop. This multi-level analysis, visualized in Figure 7, demonstrates how HybridSIMD's clean abstraction enables synergistic optimizations across control flow, memory access, and computation.

The journey begins with a clear, high-level C++ abstraction (Figure 7a), where the entire logic operates on our `Vec` type. The compiler seamlessly lowers this into LLVM IR (Figure 7b), representing the computation as operations on a single, large logical `<32 x float>` vector. This clean, unified IR is the key that empowers the compiler's back-end to generate exceptionally efficient assembly (Figure 7c). The final machine code benefits in several ways. First, the compiler fuses the work of four native iterations into a massive, unrolled loop body, drastically reducing branch frequency and amortizing control flow overhead. Second, results are written back to memory using efficient 128-bit wide vector store instructions (`vmovdqu`), avoiding the scalarization bottleneck observed in libraries like EVE, which rely on 32 individual scalar stores (`vmovss`). Finally, the large, linear block of instructions exposes a wide window of independent operations, allowing the CPU's out-of-order execution engine to better hide latencies and maximize computational throughput.

In stark contrast, the assembly generated by conventional libraries like Highway (Figure 7d) exhibits a less optimized structure. Their underlying abstractions can lead to more fragmented IR, constraining the compiler. This results in two key inefficiencies: first, a compact but high-frequency loop that incurs significant branch overhead; and second, sub-optimal memory access patterns, such as Highway's 64-bit chunk stores (`vmovq`), which require twice the memory operations of HybridSIMD. These compounding inefficiencies explain the significant performance gap.

VIII. Related Work

SIMD is an essential technology for modern high-performance computing, accelerating computations by enabling parallel processing of multiple data points with a single instruction. Its applications include scientific simulations, machine learning, and multimedia processing. To facilitate SIMD utilization across diverse hardware, various libraries provide abstractions for portable, efficient, and maintainable code, simplifying development for heterogeneous and emerging architectures.

a) SIMD Libraries: Notable SIMD libraries include HIGHWAY [7], XSIMD [8], TSIMD [9], MIPP [22], VCL [6], STD_SIMD [10], VC [23], and EVE [12]. They vary in supported architectures and C++ standards (Table IV). HIGHWAY and XSIMD support x86, ARM, RISC-V [24], and WebAssembly; TSIMD and VCL focus on x86. EVE supports x86, ARM, PPC with C++20 and no fixed element limit. STD_SIMD supports up to 32 elements with C++17. VecCore [25] provides a unified abstraction over multiple libraries but lacks fine-grained operation-level switching, which can limit flexibility in certain performance-critical scenarios, especially for mixed workloads.

b) Auto-Tuning Techniques: SkePU [26] auto-tunes multi-GPU systems, OpenATLib [27] tunes sparse solvers, MPFFT [28] auto-tunes FFTs on GPUs, and AutoTuneTMP [29] combines template meta-programming with JIT compilation. AutoPas [30] selects optimal particle simulation configurations, while Kernel Launcher [31] enables highly optimized CUDA kernels through runtime auto-tuning. These methods demonstrate the benefits of adaptive, performance-aware code generation and runtime optimization across different hardware and problem domains.

c) Compiler-Based SIMD Vectorization: Compiler approaches exploit Superword-Level, Loop-Level, Mixed-Level, and Partial SIMD Parallelism [32]–[34]. Outer-loop vectorization [35] and polyhedral transformations [36] improve SIMD code generation for short vectors and complex loops. VeGen [1] automates vectorizer generation for SIMD and beyond. Our work complements these approaches by enabling fine-grained, runtime-selectable SIMD execution, maintaining consistently high performance across diverse architectures, workloads, and applications.

IX. Discussion

This section analyzes the mechanisms behind HybridSIMD's performance, explaining how its collaborative concept opens up valuable new optimization dimensions and clarifying phenomena observed in experiments.

A. Advantages of the Collaborative Concept

The main driver of HybridSIMD's performance is its collaborative design, which resolves the fragmentation issues in Section II. Through its unified interface, it enables seamless operator-level multi-library optimization, a capability fundamentally different from relying on a monolithic library. The auto-tuner can globally explore combinations of implementations, fully exploiting the unique strengths of each library. For example, one library may serve arithmetic-heavy parts of a kernel while another better handles memory access. This synergy, identified through a comprehensive rather than greedy search, also enables deeper compiler optimizations such as fusion, reordering, and scheduling, producing novel code patterns unattainable with a single library.

TABLE IV: Overview of SIMD Libraries.

Name	Supported ISA	Supported Element Type	C++ Standard
HIGHWAY	x86,ARM,PPC,RISCV,WebAssembly	f16,f32,f64,i8,i16,i32,i64	C++14
EVE	x86,ARM,PPC	f32,f64,i8,i16,i32,i64	C++20
XSIMD	x86,ARM,RISCV,WebAssembly	f32,f64,i8,i16,i32,i64	C++11
TSIMD	x86	f32,i8,i16,i32	C++11
MIPP	x86,ARM	f32,f64,i8,i16,i32,i64	C++11
VCL	x86	f32,f64,i8,i16,i32,i64	C++17
STD_SIMD	x86,ARM,PPC	f32,f64,i8,i16,i32,i64	C++17

B. Unlocking New Optimization Dimensions: The Role of N

A central benefit of the collaborative concept is unlocking optimization dimensions often constrained in individual libraries. The most critical of these is adaptive management of the vector element count, *N*.

Most libraries tie vector length to a SIMD register (e.g., 8 floats for AVX2), which severely limits software loop unrolling. By decoupling logical length from register size, HybridSIMD's `Vectors<ElemType, N>` allows *N* values much larger than a single register (e.g., 16 or 32 on AVX2). This makes aggressive, automatic unrolling possible, significantly reducing loop overhead, improving register utilization, and mitigating branch prediction penalties in practice.

Figure 6 highlights the substantial speedup from optimizing *N*, also showing that the best choice is highly dynamic, depending on input size, workload, and ISA. HybridSIMD's auto-tuner is specifically designed to detect and apply this optimal *N*, consistently delivering peak performance across diverse scenarios by exploiting a dimension often inaccessible in conventional libraries.

C. Analysis of Cross-Architecture Performance

Our experiments also revealed nuanced performance behaviors, particularly the surprising cases where AVX2 sometimes surpasses AVX512 on small Julia and Mandelbrot inputs. This stems from several interacting factors: higher AVX512 power use can cause thermal throttling under sustained workloads; memory-bound workloads may suffer from doubled bandwidth demand; and compiler optimizations for AVX512 are not always as mature or reliable as those for AVX2. Collectively these factors challenge the traditional "wider-is-always-better" assumption and further highlight the importance of a holistic, adaptive auto-tuning approach like HybridSIMD, which can carefully navigate the subtle tradeoffs between vector length, architecture, and workload characteristics.

X. Limitations and Future Work

While HybridSIMD shows notable performance gains, there are several limitations. Our evaluation focuses on x86 and ARM architectures using LLVM/Clang. Future work should extend this to other platforms, such as RISC-V, and compilers like GCC or ICC. Additionally, we only consider single-threaded performance; integrating HybridSIMD with multi-threading models (e.g., OpenMP) raises challenges, particularly for vectorized reductions. Exploring thread-safe SIMD optimizations is a direction for future research.

The adoption of HybridSIMD for existing codebases requires a manual porting effort. Developers need to identify and refactor performance-critical code sections to use the unified interface, which may include adapting loops, vector types, and arithmetic operations to the HybridSIMD API. This step is necessary to enable cross-library auto-tuning and take advantage of the collaborative back-end. A potential direction for future work is the development of source-to-source transformation tools or semi-automated migration aids that could simplify this process and facilitate adoption of HybridSIMD in HPC applications.

Finally, from a practical standpoint, integrating multiple SIMD libraries leads to a noticeable increase in binary executable size. In our evaluation, the binary size grew by approximately 4×, primarily due to the aggressive loop unrolling strategies enabled by HybridSIMD. While we consider this an acceptable trade-off in high-performance computing, where performance improvements typically outweigh binary size concerns, future work could explore multi-objective optimization strategies that balance execution speed with binary size overhead.

XI. Conclusion

In this paper, we addressed the challenges of performance and usability fragmentation in existing C++ SIMD libraries. We argued that these library-level issues benefit from a new design concept and introduced the concept of collaborative SIMD design. We presented HybridSIMD, a C++ library that implements this collaborative principle. By providing a unified interface that integrates multiple SIMD libraries, HybridSIMD enables an operator-level collaborative back-end. Combined with a hierarchical auto-tuning engine, this approach systematically explores combinations of library implementations to address fragmentation and improve hardware utilization. Our experimental evaluation across six HPC benchmarks on AVX2, AVX512, and ARM NEON architectures showed that HybridSIMD can achieve notable performance improvements, with speedups of up to 185.34×, 97.80×, and 71.32× compared to baseline implementations. The results indicate that the collaborative approach provides a practical mechanism to mitigate fragmentation among SIMD libraries and support efficient use of diverse SIMD implementations.

References

[1] Y. Chen, C. Mendis, M. Carbin, and S. Amarasinghe, "Vegen: a vectorizer generator for simd and beyond," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2021, pp. 902–914.

[2] V. Kandiah, D. Lustig, O. Villa, D. Nellans, and N. Hardavellas, "Parsimony: Enabling simd/vector programming in standard compiler flows," in *Proceedings of the 21st ACM/IEEE International Symposium on Code Generation and Optimization*, 2023, pp. 186–198.

[3] S. Thomas and J. Bornholt, "Automatic generation of vectorizing compilers for customizable digital signal processors," in *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, 2024, pp. 19–34.

[4] D. Habich and J. Pietrzyk, "Simdified data processing-foundations, abstraction, and advanced techniques," in *Companion of the 2024 International Conference on Management of Data*, 2024, pp. 613–621.

[5] J. Pietrzyk, A. Krause, D. Habich, and W. Lehner, "Designing and implementing a generator framework for a simd abstraction library," *arXiv preprint arXiv:2407.18728*, 2024.

[6] Agner Fog. (2019) Vcl. [Online]. Available: https://github.com/vectorclass/version2

[7] M. Blacher, J. Giesen, P. Sanders, and J. Wassenberg, "Vectorized and performance-portable quicksort," *arXiv preprint arXiv:2205.05982*, 2022.

[8] W. V. M. R. Johan Mabille, Sylvain Corlay and S. Guelton, "xsimd," 2016. [Online]. Available: https://github.com/xtensor-stack/xsimd

[9] Jeffamstutz. (2017) tsimd. [Online]. Available: https://github.com/jeffamstutz/tsimd

[10] C++ Standard Committee, "std::experimental::simd," https://en.cppreference.com/w/cpp/experimental/simd, 2023.

[11] J. Penuchot, J. Falcou, and A. Khabou, "Modern generative programming for optimizing small matrix-vector multiplication," in *2018 International Conference on High Performance Computing & Simulation (HPCS)*. IEEE, 2018, pp. 508–514.

[12] ——, "Modern generative programming for optimizing small matrix-vector multiplication," in *2018 International Conference on High Performance Computing Simulation (HPCS)*, 2018, pp. 508–514.

[13] F. Black and M. Scholes, "The pricing of options and corporate liabilities," *Journal of political economy*, vol. 81, no. 3, pp. 637–654, 1973.

[14] B. B. Mandelbrot and B. B. Mandelbrot, *The fractal geometry of nature*. WH freeman New York, 1982, vol. 1.

[15] M. F. Barnsley, R. L. Devaney, B. B. Mandelbrot, H.-O. Peitgen, D. Saupe, R. F. Voss, Y. Fisher, and M. McGuire, *The science of fractal images*. Springer, 1988, vol. 1.

[16] W. D. Withers, "Newton's method for fractal approximation," *Constructive Approximation: Special Issue: Fractal Approximation*, pp. 151–170, 1989.

[17] J. M. Cebrian, M. Jahre, and L. Natvig, "Optimized hardware for suboptimal software: The case for simd-aware benchmarks," in *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 2014, pp. 66–75.

[18] M. D'orto, S. Sjöblom, L. S. Chien, L. Axner, and J. Gong, "Comparing different approaches for solving large scale power-flow problems with the newton-raphson method," *IEEE access*, vol. 9, pp. 56 604–56 615, 2021.

[19] R. Ferrer, J. Planas, P. Bellens, A. Duran, M. Gonzalez, X. Martorell, R. M. Badia, E. Ayguade, and J. Labarta, "Optimizing the exploitation of multicore processors and gpus with openmp and opencl," in *Languages and Compilers for Parallel Computing: 23rd International Workshop, LCPC 2010, Houston, TX, USA, October 7-9, 2010. Revised Selected Papers 23*. Springer, 2011, pp. 215–229.

[20] A. Pohl, B. Cosenza, M. A. Mesa, C. C. Chi, and B. Juurlink, "An evaluation of current simd programming models for c++," in *Proceedings of the 3rd Workshop on Programming Models for SIMD/Vector Processing*, 2016, pp. 1–8.

[21] M. Bagherbeik, P. Ashtari, S. F. Mousavi, K. Kanda, H. Tamura, and A. Sheikholeslami, "A permutational boltzmann machine with parallel tempering for solving combinatorial optimization problems," in *International Conference on Parallel Problem Solving from Nature*. Springer, 2020, pp. 317–331.

[22] A. Cassagne, O. Aumage, D. Barthou, C. Leroux, and C. Jégo, "Mipp: a portable c++ simd wrapper and its use for error correction coding in 5g standard," in *Proceedings of the 2018 4th Workshop on Programming Models for SIMD/vector Processing*, 2018, pp. 1–8.

[23] M. Kretz and V. Lindenstruth, "Vc: A c++ library for explicit vectorization," *Software: Practice and Experience*, vol. 42, no. 11, pp. 1409–1430, 2012.

[24] RISC. (2021) Risc-v "v" vector extension. [Online]. Available: https://github.com/riscv/riscv-v-spec/releases/tag/v1.0

[25] G. Amadio, P. Canal, D. Piparo, and S. Wenzel, "Speeding up software with veccore," in *Journal of Physics: Conference Series*, vol. 1085, no. 3. IOP Publishing, 2018, p. 032034.

[26] U. Dastgeer, J. Enmyren, and C. W. Kessler, "Auto-tuning skepu: a multi-backend skeleton programming framework for multi-gpu systems," in *Proceedings of the 4th International Workshop on Multicore Software Engineering*, 2011, pp. 25–32.

[27] K. Naono, T. Sakurai, M. Igai, T. Katagiri, S. Ohshima, S. Itoh, K. Nakajima, and H. Kuroda, "A fully run-time auto-tuned sparse iterative solver with openatlib," in *2012 4th International Conference on Intelligent and Advanced Systems (ICIAS2012)*, vol. 1. IEEE, 2012, pp. 143–148.

[28] Y. Li, Y.-Q. Zhang, Y.-Q. Liu, G.-P. Long, and H.-P. Jia, "Mpfft: An auto-tuning fft library for opencl gpus," *Journal of Computer Science and Technology*, vol. 28, no. 1, pp. 90–105, 2013.

[29] D. Pfander, M. Brunn, and D. Pflüger, "Autotunetmp: auto-tuning in c++ with runtime template metaprogramming," in *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 2018, pp. 1123–1132.

[30] F. A. Gratl, S. Seckler, N. Tchipev, H.-J. Bungartz, and P. Neumann, "Autopas: Auto-tuning for particle simulations," in *2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 2019, pp. 748–757.

[31] S. Heldens and B. van Werkhoven, "Kernel launcher: C++ library for optimal-performance portable cuda applications," *arXiv preprint arXiv:2303.12374*, 2023.

[32] S. Larsen and S. Amarasinghe, "Exploiting superword level parallelism with multimedia instruction sets," *Acm Sigplan Notices*, vol. 35, no. 5, pp. 145–156, 2000.

[33] H. Zhou and J. Xue, "A compiler approach for exploiting partial simd parallelism," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 13, no. 1, pp. 1–26, 2016.

[34] ——, "Exploiting mixed simd parallelism by reducing data reorganization overhead," in *Proceedings of the 2016 International Symposium on Code Generation and Optimization*, 2016, pp. 59–69.

[35] D. Nuzman and A. Zaks, "Outer-loop vectorization: revisited for short simd architectures," in *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, 2008, pp. 2–11.

[36] M. Kong, R. Veras, K. Stock, F. Franchetti, L.-N. Pouchet, and P. Sadayappan, "When polyhedral transformations meet simd code generation," in *Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation*, 2013, pp. 127–138.