Parsimony: Enabling SIMD/Vector Programming in Standard Compiler Flows

Vijay Kandiah
vijayk@u.northwestern.edu
Northwestern University, USA

Daniel Lustig
dlustig@nvidia.com
NVIDIA, USA

Oreste Villa
ovilla@nvidia.com
NVIDIA, USA

David Nellans
dnellans@nvidia.com
NVIDIA, USA

Nikos Hardavellas
nikos@northwestern.edu
Northwestern University, USA

Abstract

Achieving peak throughput on modern CPUs requires maximizing the use of single-instruction, multiple-data (SIMD) or vector compute units. Single-program, multiple-data (SPMD) programming models are an effective way to use high-level programming languages to target these ISAs. Unfortunately, many SPMD frameworks have evolved to have either overly-restrictive language specifications or under-specified programming models, and this has slowed the widescale adoption of SPMD-style programming. This paper introduces Parsimony (PARallel SIMd), a SPMD programming approach built with semantics designed to be compatible with multiple languages and to cleanly integrate into the standard optimizing compiler toolchains for those languages. We first explain the Parsimony programming model semantics and how they enable a standalone compiler IR-to-IR pass that can perform vectorization independently of other passes, improving the language and toolchain compatibility of SPMD programming. We then demonstrate a LLVM prototype of the Parsimony approach that matches the performance of `ispc`, a popular but more restrictive SPMD approach, and achieves 97% of the performance of hand-written AVX-512 SIMD intrinsics on over 70 benchmarks ported from the Simd Library. We finally discuss where Parsimony has exposed parts of existing language and compiler flows where slight improvements could further enable improved SPMD program vectorization.

CCS Concepts: • **Software and its engineering → Compilers**; **Semantics**; • **Computer systems organization → Single instruction, multiple data**.

CGO '23, February 25 – March 1, 2023, Montréal, QC, Canada

ACM ISBN 979-8-4007-0101-6/23/02…$15.00
https://doi.org/10.1145/3579990.3580019

Keywords: Parallel Computing, Vectorization, Code Translation, Single-instruction Multiple-data, Compiler Design

ACM Reference Format:
Vijay Kandiah, Daniel Lustig, Oreste Villa, David Nellans, and Nikos Hardavellas. 2023. Parsimony: Enabling SIMD/Vector Programming in Standard Compiler Flows. In *Proceedings of the 21st ACM/IEEE International Symposium on Code Generation and Optimization (CGO '23), February 25 – March 1, 2023, Montréal, QC, Canada.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3579990.3580019

1 Introduction

Achieving high computational performance on modern CPUs often requires making effective use of those CPUs' SIMD or vector units. Although single-thread performance scaling has slowed in recent years, single-instruction, multiple-data (SIMD) and vector ISAs continue to be an area of active innovation [28, 29, 44]. SIMD/vector registers are getting wider, with 512b registers already in widespread use. New ISA extensions such as x86 AVX-512 [9], ARM SVE [42], and the RISC-V "V" extension [35] continue to introduce instructions with richer computational power. For many workloads, these innovations can translate directly into improved throughput; however, targeting these new ISAs remains a major challenge for developers and toolchain providers alike.

Programming approaches targeting CPU SIMD/vector units fall broadly into three categories today. The simplest approach for programmers is auto-vectorization of serial code. This works well for some applications [33] but can partially or completely fail to vectorize in other cases [31]. Moreover, the serial semantics of loops do not allow users to express synchronization points across loop iterations. This restriction makes it impossible to express operations such as "shuffles", which are often performance-critical to parallel workloads. A second approach is explicit SIMD/vector programming. This approach takes many forms including inline assembly, low-level C intrinsics, or pre-packaged SIMD-optimized libraries such as Enoki [47] or SLEEF [40]. Forcing developers to write low level code that explicitly maps the SIMD-amenable portions of their problem onto differing hardware ISAs is tedious, error-prone, and burdensome. The third approach is using a single-program, multiple

data (SPMD) programming model that assumes a fixed number of threads or program instances executing in parallel. SPMD programming models such as `ispc` [30] have already proven effective at extracting good performance from CPU SIMD/vector units while retaining a user-friendly interface.

Unfortunately, current SPMD frameworks have made programming model decisions that make it difficult to express certain classes of algorithms and hard to integrate their compilation logic into existing compiler flows. For example, although `ispc` [30] delivers great performance, it requires writing programs in a custom "C-like" programming language as well as using a specialized standalone compiler infrastructure (derived from LLVM [16]); this increases the burden of adopting it into large projects. Another example, still from `ispc`, is the size of the thread "gang"[1] which is specified using a compiler flag. This approach is far from ideal. For instance, in a 512b SIMD architecture, a gang size of 16 would be ideal for 32b values, but inefficient for 8b values. A gang size of 64 would be ideal for 8b values but add tremendous register pressure with 32b values. Having to select a single gang size for the entire compilation unit makes performance tuning difficult. Similarly, while threads in an `ispc` gang execute in synchronous fashion, later innovations in GPU SPMD programming models such as CUDA [26] have deprecated such "warp-synchronous" programming approaches in order to improve the soundness of the threading model [25].

As such, the goal of Parsimony is to introduce a well-defined SPMD programming model and compiler flow that efficiently targets a CPU's SIMD/vector units while remaining compatible with standard programming models, languages, and compiler toolchains. As shown in Figure 1, Parsimony's design starts at the language semantics level and is designed to be compatible with any number of front-end language syntax choices. A Parsimony-compatible language must only introduce the ability to conceptually instantiate a programmer-specified set of threads—using the term "thread" in the semantic sense, not necessarily as a true operating system (OS) thread. Inter-thread communication is permitted, but only when obeying standard inter-thread communication rules. Thus, Parsimony must expose efficient "horizontal synchronization" operations to facilitate synchronization within gangs. Due to this primarily single-threaded model, the code can pass through any standard optimization flow in the compiler. Vectorization instead occurs through a standalone IR-to-IR transformation pass that translates the SPMD-annotated function(s) into architecture-independent vector IR. Each architecture's standard back-end can then optimize the translated IR for the target ISA as it sees fit.

Overall, the contributions of this work are as follows:

1. We present Parsimony, a well-specified programming model and compiler framework fully compatible with standard language semantics and compiler flows.

[1]A "gang" in `ispc` is a group of concurrent program instances.

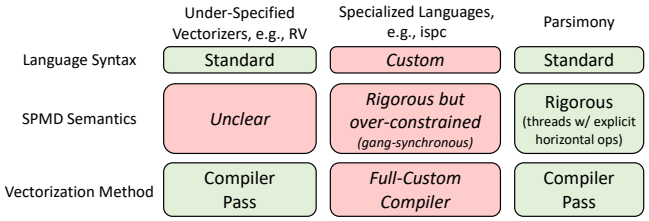


Figure 1. Existing SPMD vectorizers are effective but have shortcomings. Whole-Function Vectorization [13] and Region Vectorizer (RV) [21] do not clearly specify their intended semantics. Others (e.g. `ispc`) are overly-restrictive and hard to integrate into large projects. Parsimony targets well-defined SPMD semantics compatible with standard compilers, while achieving similar performance targets.

2. We demonstrate a prototype implementation of Parsimony in LLVM, with performance results showing that our SPMD variant performs as well as state-of-the-art SPMD frameworks (i.e., `ispc`) and custom AVX-512 code, without requiring the use of a specialized programming language or compiler.
3. We identify places where improvements/extensions to LLVM's IR would facilitate better integration of SPMD flows, and we provide takeaways for how languages and language extensions like C++ and OpenMP can integrate the Parsimony approach to SPMD for improved performance and programmer productivity.
4. We publicly release our Parsimony compiler framework and benchmarks to facilitate further research.

2 Background and Motivation

SIMD ISA extensions employ a fixed-width SIMD register file and an instruction set that operates on fixed-width operands, e.g., 128b, 256b, or 512b in the case of x86 AVX-512 [9]. Conversely, "vector" ISA extensions such as ARM SVE [42] and RISC-V "V" [35] employ a vector-length-agnostic (VLA) instruction set that allows for implementations with different vector widths to support the same ISA. For example, ARM SVE supports hardware vector width implementations that can vary between 128b and 2048b in 128b increments. This enables pre-compiled code to run seamlessly across the supported vector widths without requiring recompilation. In this paper, unless otherwise specified, we use the terms "SIMD" and "vector" interchangeably as the differences between these approaches are important only if programmers are using low-level intrinsics, and are generally not significant if being targeted by a SPMD-style program.

The currently mainstream techniques to leverage SIMD and vector instruction sets and extract SIMD-level parallelism on CPUs are briefly discussed below.

Auto-Vectorization: In classical loop auto-vectorization, the compiler attempts to transform a region of serial code (usually a loop) into a block of vector instructions [1, 23,

45]. To do this, it relies on algorithms such as alias analysis as well as target-dependent heuristics to determine whether vectorization would be both legal and profitable. While this approach requires little to no additional programmer effort, auto-vectorization is opportunistic and is generally limited by the level of sophistication of the compiler's analysis abilities. As such, it tends to produce highly variable performance characteristics across systems. Language extensions such as C++ `std::execution::unseq` [41] or OpenMP `#pragma omp simd` [27] aim to improve the efficiency of auto-vectorization by providing user annotations or hints to the compiler, e.g., to ignore cases where the compiler cannot prove there are no loop-carried dependencies; however, many of the same fundamental challenges remain.

Vectorization remains an active area of research. Outer-loop vectorization [24] focuses on loops that are not the innermost in a hierarchy. This introduces additional challenges, as it raises the probability that there will be divergent control flow among outer loop instances. SLP vectorization [15, 32] is another auto-vectorization technique that combines similar independent scalar instructions to form vector instructions. Hence it offers more flexibility than loop vectorization because it does not just target parallelism across loop iterations. Additionally, auto-vectorization is performed on serial loops, and serial loops do not allow programmers to express horizontal communication between iterations.

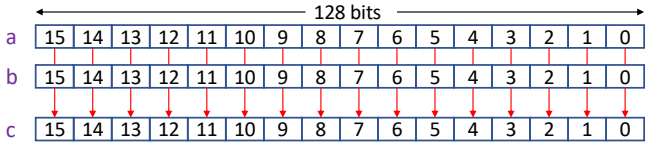
Low-Level Intrinsics: SIMD/vector intrinsics are small functions that map nearly 1:1 to assembly instructions for a particular ISA. Manually inserting SIMD/vector intrinsics requires significant low-level programmer effort and is inherently non-portable. Nevertheless, due to the limitations of the other programming approaches described in this section, libraries aiming for peak performance often contain extensive use of intrinsics in spite of the engineering costs [34].

SIMD Libraries: Libraries like enoki [47] and SLEEF [40] shift code portability into a library-supported layer for different architectures. While using SIMD libraries does make the source code more readable than the lower level intrinsics approach, similarly high performance can be achieved only if the source code can be expressed in terms of the limited set of exposed APIs that are specified by these libraries.

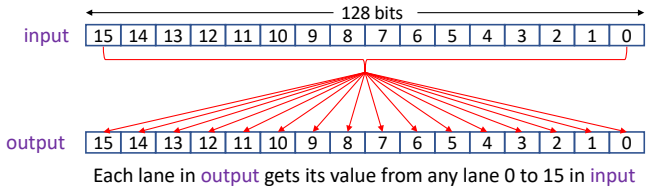
Dedicated SPMD Languages: SPMD-on-SIMD programming models such as `ispc` [30] generate multiple conceptual program instances that operate on different data from scalar C-like code. Each instance of the program is then mapped to a different SIMD lane to extract parallelism. While such programming models are very appealing to efficiently utilize the CPU SIMD/vector units, their use of non-trivial keywords like "varying" and their reliance on non-standard compilation toolchains has limited their widespread adoption.

2.1 Mapping SPMD Programs to SIMD/Vector Units

When viewed through the lens of a SPMD program, both SIMD and vector ISA extensions enable traditional data-level parallel operations through "vertical" operations in which multiple logical "lanes" all operate independently. The number of lanes that can concurrently execute in hardware is thus a function of the SIMD/vector width and the data width of the operand being operated upon. For example, Figure 2a shows a SIMD add instruction performing a vertical addition of two 8-bit values across 16 lanes of two input 128b registers. In contrast, "horizontal" instructions operate across the lanes. For instance, a shuffle instruction exchanges values across a single SIMD/vector input register as shown in Figure 2b. Modern SIMD/vector ISAs also include complex instructions that are neither purely vertical nor purely horizontal. For example, AVX-512 includes instructions that perform a vertical operation in combination with a horizontal operation (e.g., *vpsadbw* [9]). Such instructions are harder for compilers to target, but recent work [3] has improved the situation.



(a) A "Vertical" add instruction operating independently within each lane.



(b) A "Horizontal" shuffle operation permitting any-to-any lane output.

Figure 2. SIMD/Vector operations can occur both per-lane and across lanes in high performance ISAs.

An enabling feature of modern SIMD and vector ISAs that allows for efficient SPMD programming is the support for masked execution with per-lane predication of execution output. The predication mask registers have one bit per lane and masked-off lanes will not modify their sub-portion of the output register used by the SIMD instruction. This fine-grained predication is critical when mapping programs onto a single thread executing SIMD instructions even though the SPMD threads diverge along different control flow paths.

2.2 Motivating Improved SPMD Semantics

In the examples below, we analyze variants of a simple program that copies data from each position in an array into an adjacent array position. This program is not as innocent as it may seem; it highlights several interesting subtleties that can arise in SPMD programming model decisions and exposes compiler implementation issues that may appear.

```
// OpenMP version
template<typename T, unsigned N>
void foo(T* a) {
    #pragma omp simd
    for (unsigned i = 0; i < N; i++) {
        T tmp = a[i];
        // data race! cannot synchronize
        a[i+1] = tmp;
    }
}
```

Listing 1. OpenMP maintains serial execution semantics.

```
// ispc version (limited support for templates)
void foo(uniform int a[]) {
    foreach(uniform i : 0 ... N) {
        int tmp = a[i];
        // implicitly gang-synchronous!
        // correct only if N <= compile time gang size
        a[i+1] = tmp;
    }
}
```

Listing 2. `ispc` code is "gang-synchronous".

Listing 1 presents a version of the program written in C++ with OpenMP. As required by most `#pragma` implementations, the program semantics can be fully understood by ignoring the `#pragma`: e.g., each loop iteration reads the value of `a[i]` and writes it to `a[i+1]`, where the latter is then read during the next loop iteration. However, the OpenMP `#pragma` allows the compiler to legally ignore loop-carried dependencies that can be difficult to analyze (though in this case, the dependency is obvious). Note that it does *not* specify that a loop-carried dependency *must* be ignored. Ignoring it would allow all loop iterations to first perform the load before any iteration performs its store; i.e., it would allow vectorization of the loop. Specifying this type of synchronization requires explicitly breaking the single loop into two. While doing so would be straightforward in this program, it becomes complex or impossible in larger regions. Meanwhile, there is no way to clearly specify the intended interpretation of the program as written. Some compilers will take advantage of the pragma and vectorize the code in spite of the loop-carried dependency. Others will choose not to vectorize in order to maintain the original single-threaded semantics.

An `ispc` version of the same program is shown in Listing 2. Due to `ispc`'s gang-synchronous execution model, `ispc` *requires* all threads in the gang to execute the load before any thread executes its store. However, in `ispc`, the gang size is a compilation flag that is tightly coupled with the ISA SIMD width of the target machine. Programmers can access it through the `programCount` variable, but not set it. Therefore, the correctness of this code changes depending on the relationship between gang size and `N`. This is less than ideal from a programming model perspective.

Listing 3 now demonstrates how the running example would be written using Parsimony, using `#psim` syntax as one example of how to demarcate an explicit SPMD parallel region. Described in more detail in Section 3, Parsimony compatible code explicitly instantiates a programmer-specified number of independent threads that can be grouped into gangs. The gang size need not match the hardware's SIMD width; the compiler back-end can map any gang size onto any target ISA. Because the number of threads is specified at the program level, a developer can reason about program correctness strictly based on the programming model. There is no requirement to know the compiler options being specified nor the SIMD/vector width of future hardware the program will be executed on. As with modern GPUs [26], but differing from `ispc`, Parsimony eschews a gang-synchronous programming model and instead requires the programmer to explicitly synchronize across a gang when needed. This makes it easier to incorporate standard sequential semantic compiler passes and facilitates the possible adoption of more flexible forward progress guarantees in the future.

```
// Parsimony version:
template<typename T, unsigned N>
void foo(T* a) {
    #psim gang_size(N) {
        uint64_t i = psim_get_lane_num();
        T tmp = a[i];
        psim_gang_sync();  // explicit!
        a[i+1] = tmp;
    }
}
```

Listing 3. Parsimony makes gang size and horizontal synchronization explicit.

```
// Parsimony version:
template<typename T, unsigned N>
void foo(atomic<T>* a) {
    #psim gang_size(N) {
        uint64_t i = psim_get_lane_num();
        a[i].fetch_add(1, memory_order_relaxed);
        a[i+1].fetch_add(1, memory_order_relaxed);
    }
}
```

Listing 4. Example showing how "gang-synchronous" behavior can break compiler optimizations legal for single-threaded code.

Listing 4 presents a different example showing how a gang-synchronous programming model can introduce semantics incompatible with standard compiler optimizations in languages such as C++. Because the example operations are atomics, there are no concerns about data races regardless of the actual execution order in hardware. A typical single-thread compiler optimization pass can tell that the atomics are performed to two adjacent non-aliasing addresses. Therefore, it would be legal for the compiler to reorder the atomics arbitrarily. However, in a gang-synchronous model, all threads in the gang are required to perform the first atomic before any thread in the gang performs the second atomic. Therefore, the second atomic in each thread must read the result written by the first atomic from the adjacent thread (except at the boundary condition). To preserve this semantic,

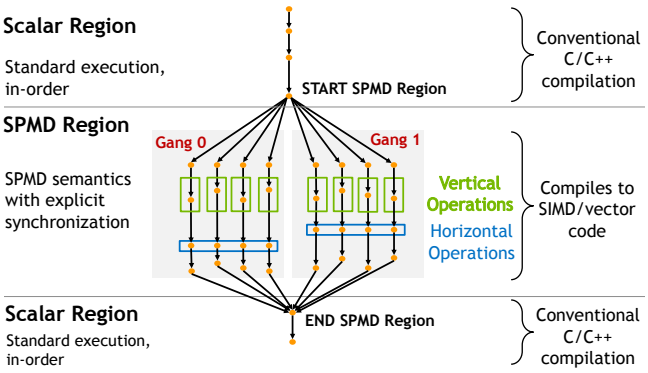


Figure 3. The Parsimony SPMD programming model.

the compiler cannot reorder the atomics. Hence, optimization passes capable of reordering memory operations in cases such as this have to be explicitly disabled, modified, or specialized, making it difficult to integrate "gang-synchronous" SPMD models with modern vectorizing compiler flows.

The examples above show three important takeaways that motivate Parsimony's design. First, the semantics of a SPMD programming model should be well-defined in a way that the programmer can reason about at the language level, i.e., the semantics should not depend on any compile-time flags. Second, designing the semantics independently from the syntax allows the SPMD semantics to be integrated into widely used languages, facilitating adoption. Finally, the SPMD semantics should strive to be compatible with standard single-thread semantics to facilitate integration into standard compiler flows. The next section explains Parsimony's programming model and how it meets all of these goals.

3 Parsimony Programming Model

Parsimony is a general-purpose SPMD programming model designed to integrate cleanly into any programming language that supports threading and shared memory semantics. For explanatory purposes and in our implementation, we use standard C++ as the target language; however, the same principles extend to other languages.

In Parsimony's SPMD programming model, which is depicted in Figure 3, a SPMD region is a region of code in which a fixed number of conceptually independent threads are created. The SPMD region executes within the parent thread. This means the threads are conceptually "forked" at the start of the region and "rejoined" at the end of the region, where the parent thread continues its execution. However, no threads are actually forked in an operating system sense; the "fork" and "join" describes the threads' behavior within the language semantics. Within each thread, standard intra-thread sequencing rules apply.

Threads are also grouped into *gangs* of a fixed size, determined by the programmer as part of the syntax declaring the parallel region. This allows different SPMD regions in a

```
1 void foo(uint32_t* a, uint32_t* b) {
2   #psim gang_size(16) num_spmd_threads(N) {
3     size_t i = psim_get_lane_num();
4     if (a[i] + i < b[i]) {
5       a[i] += 1;
6     }
7     b[i] = psim_shuffle_sync<uint32_t>(a[i], i + 4);
8   }
9 }
```

Listing 5. Parsimony syntax, as embedded in C++.

program to operate on different gang sizes, which is useful when differing functions operate on data structures having different element sizes. This differs from `ispc`'s approach, which specifies the size of the gang using a target-dependent compiler flag. Also unlike `ispc`, Parsimony threads are not "gang-synchronous"; there is no implicit synchronization between threads at every sequence point (i.e., before or after each statement). As mentioned earlier, this choice provides more optimization and scheduling flexibility to the compiler.

In Parsimony, synchronization between threads is instead performed using explicit horizontal operations. In contrast to auto-vectorization of loops or other language constructs such as `std::execution::unseq`, which is the current C++ standard recommendation for code targeting SIMD units [41], Parsimony threads may also communicate through memory using standard inter-thread memory ordering rules. As long as data races between threads are avoided, communication through memory is well-defined behavior.

Parsimony guarantees concurrency and weak forward progress among threads in each gang: if all threads individually make forward progress, then all threads in the gang will eventually make forward progress. This rule is necessary to ensure that horizontal operations behave correctly. However, Parsimony does not provide *global* forward progress guarantees, e.g., if one thread in a gang is waiting on a spinloop that will be signaled by another thread in the same gang, then the stalling thread may block the progress of the entire gang. Additionally, there is no guarantee of concurrency or forward progress among different gangs. These restrictions are tighter than those in place on modern GPUs [25] which support a single-instruction multiple-thread (SIMT) programming model. GPUs may have hardware-assisted independent thread scheduling [25], whereas Parsimony relies on a more restricted forward progress model to ensure that there is no need for a software implementation of concepts such as SIMT convergence stacks [6, 17] or launching of multiple OS threads to enable thread preemption.

Listing 5 shows the syntax we have used to prototype Parsimony and employed in the examples in this paper. These syntax choices are not fundamental and could be adapted as needed for different languages/frameworks. As shown, a SPMD region is identified with the `#psim` construct and prefixed with syntax indicating the gang size (`gang_size`) as well as the number of total threads (`num_spmd_threads`) or gangs (`num_spmd_gangs`). This gang size can take any

compile-time constant value; there is no dependency on the hardware vector width. The last gang may be partially full depending on whether the number of threads is a multiple of the gang size. The user can obtain the unique thread number within the SPMD region using `psim_get_thread_num()`, the gang number with `psim_get_gang_num()`, and the lane number within the gang with `psim_get_lane_num()`.

To allow further compiler optimization, the user can call `psim_is_tail_gang()` and `psim_is_head_gang()` to explicitly identify the first and the last gang in the region. This is unique and important because the first and last gang are typically used to perform operations on the boundary of data structures. Hence, more expensive boundary condition checks are often performed there and a programmer may not want to burden all threads with performing those. The compiler can use this information to automatically extract the first and last gang into a copy of the function that is separate from the rest, so that the boundary condition checks can be optimized away from the non-boundary gang execution. Parsimony provides no guarantee of ordering among gangs, so depending on the compiler implementation they can be executed sequentially, out of order, and/or in parallel.

The body of the SPMD region automatically captures variables from outside the region by reference, as needed; the SPMD region itself takes no explicit arguments. SPMD threads may contain any arbitrary language constructs, including arbitrary control flow or memory access patterns, subject to standard language semantics. Parsimony also provides a set of APIs for operations not typically exposed in standard language APIs, such as saturating math operations and horizontal shuffle and data exchange operations.

The choices described above were made to facilitate the use of a standalone IR-to-IR vectorization pass that can be integrated easily into standard language toolchains. The next section describes the implementation details of such a pass.

4 Parsimony Compiler Implementation

We now describe how Parsimony SPMD semantics integrate into a typical compiler flow and our prototype that manifests these concepts. We use LLVM for our implementation, though these concepts should generalize to other compilers.

The overall flow for Parsimony compilation works as follows, with each step described in further detail below. First, the program is compiled from source into the compiler's intermediate representation (IR) by the compiler front-end. The front-end is modified only in two ways: to support SPMD semantics within the source language as needed, and to disable any early-stage auto-vectorization that might occur by default. Second, the new Parsimony IR-to-IR vectorization pass is added to the middle-end optimization process. This new pass vectorizes the SPMD regions and is the core of the Parsimony design. Finally, the IR is translated to machine-specific assembly using the unmodified compiler back-end.

```
// Original source code, before extraction
void foo(int* a) {
    // code before...
    #psim gang_size(G) num_spmd_threads(N) {
        // SPMD region code
    }
    // code after...
}
////////////////////////////////////////////////////

// After extraction
void foo(int* a) {
    // code before...
    for (unsigned i = 0; i < N; i += G) {
        if (i + G <= N) {
            foo_extracted_full(/*captured vars*/);
        } else {
            foo_extracted_partial(/*captured vars*/);
        }
    }
    // code after...
}
// SIMD annotation: gang size G
inline void foo_extracted_full(/*captured vars*/) {
    // SPMD region code
}
// SIMD annotation: gang size G
inline void foo_extracted_partial(/*captured vars*/) {
    if (thread_id < N) {
        // SPMD region code
    }
}
```

Listing 6. An abstracted representation of the SPMD region extraction process performed by the front-end.

4.1 Front-End

The job of the compiler front-end within Parsimony is to produce a list of SPMD regions to be vectorized by the middle-end. We assume that the vectorizer operates at the level of whole functions, and as such, the front-end must extract SPMD regions from serial code into standalone SPMD-annotated functions. The vectorized function can later be re-inlined by the back-end in order to avoid the overhead of an extra function call. The SPMD annotation attached to the function must record relevant metadata such as the gang size and the total number of threads executing that region as specified by the Parsimony programming model.

Our prototype implements SPMD function extraction by piggybacking on Clang support for the extraction of `#pragma omp parallel` code regions. OpenMP parallel regions are implemented in Clang by outlining the parallel region into a standalone function, implicitly capturing any needed variables being referenced. After function extraction, the Parsimony front-end re-intercepts the OpenMP thread fork API and replaces it with a loop around a call to the Parsimony-vectorized function(s). This loop, which iterates over all of the gangs in the region, is specialized based on whether the total number of threads is known to be an exact multiple of the gang size and whether there are calls to APIs such as `psim_is_head_gang()` or `psim_is_tail_gang()`. Listing 6 shows a stylized example of the front-end flow.

4.2 Middle-End Vectorizer

The Parsimony vectorization phase is responsible for vectorizing SPMD-annotated functions generated by the front-end. This phase follows a flow similar to many existing vectorizers [13, 21, 30] but is tailored specifically to Parsimony's flavor of SPMD semantics. It is important to note that existing vectorizers often rely on being placed at a particular point within a bespoke sequence of optimization passes [10], whereas Parsimony's vectorization pass can be placed anywhere in the optimization pipeline. Parsimony's middle-end flow starts with the analysis of the scalar code, followed by transformation into vector code, as described below.

4.2.1 Control Flow and Mask Calculation. An SPMD annotation indicates that the function must be translated into a version in which G independent threads execute the function in SIMD fashion, where G is the gang size. The vectorized function's control flow must account for the possibility that the conceptually independent threads can diverge along different control flow paths. Capturing this divergent behavior requires the SIMD thread to reach all control flow branches executed by any of the conceptual SPMD threads. Threads that are not currently actively executing any particular control flow path need to be masked off so as to not disturb the values in those threads' lanes of the vector values.

Parsimony uses the following process to calculate its vectorization masks. First, it uses pre-existing LLVM support for "structurizing" the control flow graph into a state where all forward control flow consists only of "if-then" patterns[2] [20]. Then, similar to prior work [13], two masks are prepared for each basic block: an entry mask and an active mask. In loop headers, the entry mask represents the mask of threads that entered the loop, and hence those which must also collectively exit the loop once all threads have finished iterating. In other basic blocks, both masks are identical. The active mask for each basic block is calculated as the logical-AND of the predecessor's entry mask and (if applicable) the condition on the branch at the end of the predecessor block. Loops also receive a dedicated mask for each exit; threads incrementally update these masks as they exit the loop. Once all threads have reconverged at the loop exit, the exit masks are used to steer subsequent control flow.

4.2.2 Shape Analysis. Shape analysis is a blanket term for various techniques described in literature as stride, affine, uniform, convergence, or divergence analysis [4, 18, 36, 37]. Shape analysis attempts to track patterns in all SPMD threads' copies of a single variable. For example, if the compiler can prove that a variable will always have identical contents in all SPMD threads, then it is *uniform*. If the compiler can prove that a variable will always be equal to some base value common to all threads plus a per-thread offset that is some fixed multiple of the thread number, then it is *strided*.

Shape analysis is critical to the performance of vectorized code in several ways. First, uniform values can be stored in scalar registers and be operated on by scalar instructions which can improve latency, throughput, and/or register pressure in many CPU architectures. Second, uniform branches can also be translated into scalar branches, thus decreasing execution of fully masked dead code paths. Finally, shape analysis is crucial to the selection of efficient memory access instructions. The naive vectorization of a load and store instruction where each SPMD thread may be accessing unrelated memory addresses generate a SIMD gather or scatter operation. SIMD gathers and scatters are very slow on most modern CPUs—often no faster than performing each individual serialized scalar accesses. However, if the shape of the addresses accessed can be proven to be either uniform or strided, the compiler can generate highly efficient scalar or packed SIMD operations, respectively.

Parsimony classifies all value shapes into one of two categories: *indexed* or *varying*. Indexed values can be represented as a fixed common base value that may or may not be known at compile time, plus a per-thread offset that must be known at compile time. The common base values are maintained as scalar values in the IR, but the offsets are stored as metadata within the compiler. Varying values are those which are not indexed; these are stored as vector values in the IR. Note that both uniform and strided values are subsets of indexed values; the broader indexed category allows for more shape patterns to be captured, thus enabling more optimization.

Parsimony's shape analysis iterates on a per-instruction basis. Constants and function arguments are marked uniform. Calls to Parsimony APIs have operation-dependent shapes. For example `psim_get_lane_num()` is indexed with stride 1, while `psim_get_num_threads()` is uniform. The shape of each instruction is calculated by applying the semantics of the instruction to the shapes of its operands and then, if possible, interpreting the result as a new indexed value. If this is not possible, the output shape is marked as varying. If an instruction's input operand is not immediately available, e.g., due to a circular dependency within a loop, then the calculation proceeds speculatively but optimistically; the process then advances iteratively, recalculating any speculated shapes, until the result converges.

For example, consider an integer add instruction applied to two indexed operands with values $(a_{base}+a_i)$ and $(b_{base}+b_i)$, respectively, where a_{base} and b_{base} are the common base values and a_i and b_i are the offsets for lane i. Addition produces

$$(a_{base} + a_i) + (b_{base} + b_i) = (a_{base} + b_{base}) + (a_i + b_i),$$

which can easily be interpreted as a new indexed value. Integer multiplication produces

$$(a_{base} \times b_{base}) + (a_i \times b_{base}) + (b_i \times a_{base}) + (a_i \times b_i).$$

[2] This pass assumes the control flow is structured. For unstructured control flow, partial linearization [21] could be used.

This value can only be interpreted as indexed if a_{base} and b_{base} are known at compile time [36]. Otherwise, the two middle addends are neither common across all lanes nor per-lane values that are known at compile time.

For many instructions, the ability to classify a shape as indexed depends on certain facts about the input operands. For example, for a logical-AND operation, the outcome

$$(a_{base} + a_i)\&(b_{base} + b_i) = (a_{base}\&b_{base}) + (a_i\&b_{base})$$

holds if b is a uniform negative power of two and a is an even multiple of $-b$, but may not hold otherwise. To enable this, some vectorizers also track metadata about properties such as variable alignment manually [30].

Parsimony performs shape analysis with the help of the z3 SMT solver [22] in two phases. In an offline phase, a large set of conditional shape transformations (such as shown above for logical-AND) are verified for correctness. At compilation time, known facts about IR values are tracked as z3 model constraints and a particular shape transform is applied only after verifying that its preconditions are satisfied by the operands. Although verifying the transformations can be slow, checking the preconditions takes just fractions of a second, so this online checking imposes negligible compile-time overhead. This two-phase validation of transformations allows any new proposed transformation to be rigorously, yet easily, verified before being deployed in Parsimony.

4.2.3 Instruction Transformation. Transformation is the step where each instruction in the original scalar function is converted into the form it will take in the vectorized function. Most instructions will be vectorized, but some may remain scalar, e.g., if operating only on indexed values. We describe the handling of various instruction types below.

Arithmetic instructions are converted into vector form if their output shape is varying. For example, an instruction

```
%2 = mul nsw i32 %0, %1
```

operating on varying values `%0` and `%1` and producing varying value `%2` will be transformed into

```
%2 = mul nsw <G x i32> %0, %1
```

where `G` is the gang size. Arithmetic instructions operating on and producing only indexed values remain scalar, as only their common base value is stored at runtime.

In `alloca` instructions (stack allocation), the original size is multiplied by the gang size and pointer types are adjusted accordingly. A more optimized implementation could also (where possible) swizzle the data layout from array-of-structs into struct-of-arrays to avoid unnecessary gather/scatter operations on stack-allocated values [30].

Memory instructions are converted into a number of forms dependent on the shape of their address operands. Loads from a uniform address remain as regular scalar loads into uniform values. Stores to a uniform address are racy, unless only one thread is active; Parsimony chooses to emit a compile time warning then chooses one active thread to perform the scalar store. Loads from, or stores to, an address which is indexed with offset stride equal to the size S of the scalar type being accessed are converted into *packed* vector loads or stores of $G \times S$ consecutive bytes, respectively. These packed operations are typically an order of magnitude more efficient than gather/scatter on all CPUs we have tested with Parsimony. Loads and stores of indexed values with other forms of stride may be converted into a packed load/store plus shuffle operation(s) if the indices remain within a particular bound (in our implementation, 4× the gang size), as the accesses plus the extra shuffle(s) are still faster than performing gather/scatters. However, loads or stores of varying values must be converted into gather/scatter operations. All vector memory accesses are masked by the thread block's active mask to ensure that inactive lanes do not clobber data in memory or perform out-of-bounds accesses.

Branch instructions with varying values used as the condition are transformed into non-conditional branches to the originally-taken branch. This ensures that all paths through the CFG, potentially taken by any thread, will be properly evaluated. This can be further optimized by explicitly checking at runtime if any thread takes the branch and following the not-taken branch if none do. Prior projects have chosen to do this both implicitly [21] or explicitly via keywords such as `ispc`'s `cif` [30]. Branch instructions with uniform condition values remain as conditional branches.

The behavior of function call transformations depends on the callee. Calls to Parsimony intrinsic functions are implemented to match the semantics of that function. In many cases, e.g., for `psim_get_thread_num()`, the function can be replaced by a scalar or vector constant. Calls to functions with known vector interfaces can be made directly, adjusting for API peculiarities as needed (e.g., only some gang sizes may be available). Annotations analogous to `#pragma omp declare simd` [27] could be used to indicate that any standalone function should be vectorized and exported. Our prototype currently supports the SLEEF math library [40], but we envision generalizing this in the future. Calls to scalar functions that cannot be inlined are transformed into a serial loop of scalar calls by each active thread individually. This is another way in which the lack of gang-synchronous execution requirements makes Parsimony code easier to compose, as separately-compiled scalar functions cannot be transformed to execute in gang-synchronous fashion.

ϕ nodes that have varying output values and are the join point for two forward edges must be converted into `select` operations. This operation picks the contents of each lane in the output vector value individually, based on the active mask of whichever predecessor is the 'then' block in the 'if-then' pattern that the entire CFG was earlier adapted into. This step is the key to ensuring that live values are not clobbered by unmasked arithmetic instructions executed by active and inactive lanes in the CFG predecessors. Other ϕ nodes can be transformed just as regular arithmetic instructions are.

4.3 Back-End

Once the vectorization pass has completed, the result can be passed to any number of other optimization passes and then to the unmodified compiler back-end. As part of this process, the IR will in most cases be further simplified. The back-end is also responsible for unrolling each vector instruction if the IR instruction's vector width (i.e., usually the gang size) does not match the width of the instructions available on the target. For example, with a gang size of 32 and a target ISA with 512b vector registers, an integer add IR operation on 32b `int`s (32×32b=1024b) would reduce down to two 512b SIMD assembly add instructions. The back-end is free to schedule these instructions however it chooses, subject to not breaking the semantics of horizontal operations.

Our Parsimony prototype focuses on x86 and AVX-512. We explored support for ARM SVE, but LLVM support for vector length-agnostic ISAs in general is less mature than for AVX-512, so we leave an evaluation on SVE as future work.

5 Evaluation Methodology

We evaluate the Parsimony prototype on two benchmark suites. First, we ported the `ispc` benchmark suite to Parsimony enabled C++. Comparing to `ispc` allows us to quantify if `ispc`'s more restrictive SPMD model enables better, worse, or similar performance to the more general Parsimony SPMD model. We adapted the `ispc` versions into Parsimony maintaining the same algorithms. Second, we ported 72 benchmarks from the Simd Library [8], a popular high-performance image processing and machine learning library. This suite contains multiple versions of each benchmark, including serial and hand-coded versions specifically optimized for SIMD/vector ISA back-ends using manually-tuned low-level intrinsics. Due to pragmatic limitations such as the Simd Library making heavy use of templates and custom C++ datatypes, we were unable to port these benchmarks to `ispc`—demonstrating the need for maintaining language and compiler level compatibility in SPMD programming systems.

Our IR-to-IR Parsimony pass is based on LLVM 15.0.1 [20] and our auto-vectorization comparisons were performed with LLVM's default vectorization (loop + SLP) pipeline. We also compared against various research and production auto-vectorizers, but elide the results because the broad trends were similar to LLVM's auto-vectorization, despite some variations in individual benchmarks. We compiled the `ispc` code with the latest release version of `ispc` (v1.18.0) [11] with default compilation flags. For all results, we report averages collected over five workload executions on a Intel® Xeon® Gold 6258R CPU with AVX-512 support compiled with Clang options `-O3 -march=native -mprefer-vector-width=512`. All experiments are single-threaded from the OS's point of view because Parsimony's SPMD design focuses on efficient SIMD/vector execution within a core.

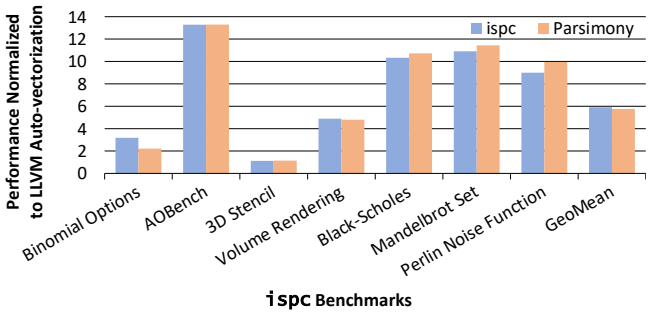


Figure 4. Parsimony and `ispc` performance compared to LLVM Auto-vectorization.

6 Experimental Results

Figure 4 shows the performance of Parsimony and `ispc` on 7 `ispc` benchmarks [30] normalized to the baseline LLVM 15.0.1 auto-vectorized serial implementation. Parsimony and `ispc` achieve a geomean speedup of 5.9× and 6× relative to auto-vectorization respectively. Parsimony closely matches `ispc`'s performance on all benchmarks except Binomial Options, for which Parsimony achieves 0.71× of `ispc`'s performance. We narrow this performance gap down to `ispc`'s use of its built-in SIMD math library function `pow`. Our Parsimony prototype uses the SLEEF [40] math library for math functions such as `pow`, and SLEEF's implementation of `pow` for x86 AVX-512 is 2.6× slower. This performance difference is not inherent to the `ispc` or Parsimony SPMD design choices. This demonstrates that gang-synchronous and non gang-synchronous SPMD designs can achieve nearly identical performance on modern architectures. *Therefore, we conclude that there is no performance penalty for choosing our easier-to-adopt non-synchronous SPMD semantics.*

To demonstrate the robustness of the Parsimony approach, Figure 5 shows the performance of Parsimony SPMD implementations, auto-vectorized serial C++ implementations, and hand-written AVX-512 implementations of 72 Simd Library benchmarks normalized to un-vectorized scalar implementations. LLVM's Auto-vectorization yields a geomean 3.46× speedup over LLVM's scalar baseline, while Parsimony yields a geomean 7.7× speedup; this results in a geomean 2.23× speedup for Parsimony over auto-vectorization. Furthermore, the handwritten AVX-512 intrinsics implementations perform negligibly better than Parsimony, as Parsimony achieves a geomean 0.97× performance relative to handwritten implementations. *From these results, we conclude that Parsimony's flavor of SPMD semantics is capable of delivering near-peak SIMD performance without requiring programmers to resort to architecture-specific low-level intrinsic programming.* Moreover, Parsimony manages to achieve high performance while having a 7× average code reduction relative to handwritten implementations while ensuring code portability with good compiler and language compatibility.

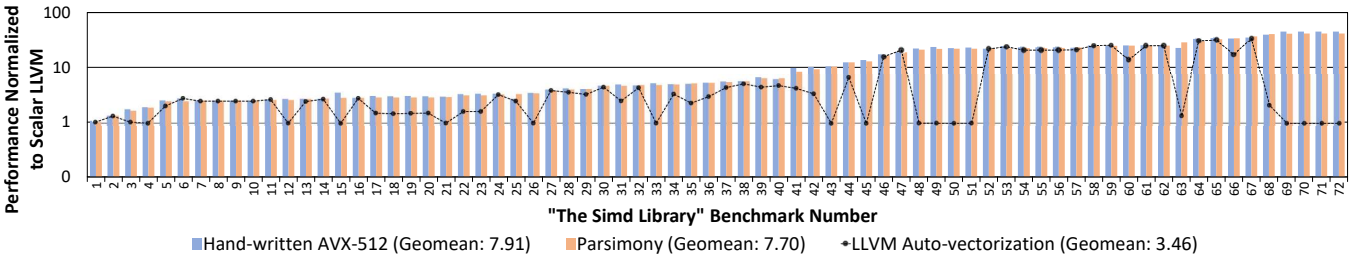


Figure 5. Speedup over LLVM scalar compilation, i.e., with vectorization disabled, on 72 Simd Library benchmarks.

7 Discussion

For pragmatic reasons Parsimony uses a small number of architecture-specific IR constructs during instruction transformation. We envision that important, common operations such as "multiply and return upper half" will be included as general-purpose compiler IR constructs in the future in order to further decouple vectorization from architectural constraints. For existing instructions that are neither purely horizontal or vertical, we explored exporting language level APIs with higher-level portable abstractions. For instance, we abstracted the AVX-512 `vpsadbw` instruction, which accumulates the sum of absolute differences of 8b values in sets of eight lanes from the input register into a single 16b value shared by 8 lanes, using an opaque data structure added to the Parsimony programming API that could have multiple back-end implementations. For other instructions which may truly be unique to a particular ISA, developing a clean general-purpose exposure for them up through the programming model would be an interesting area of future work.

Parsimony's SPMD programming model differs from other contemporary parallel language approaches in several important ways. C++ uses `std::execution::unseq` to describe loops in which iterations are not related by the "sequenced before" relationship that otherwise orders operations within the same thread. Unfortunately, concurrent accesses by unsequenced evaluations to the same address are racy and hence have undefined behavior. Similarly `std::execution::par` allows spawning of threads to execute instances, but describes instances as indeterminately-sequenced, implying that there is no concurrency between iterations assigned to the same thread, which precludes horizontal operations with high-performance ISA support from being used. This could be resolved by introducing a `std::execution::spmd` execution policy relying on Parsimony SPMD semantics and horizontal operations and other relevant SPMD APIs. Likewise, OpenMP `#pragma omp simd` and OpenACC pragmas generally maintain serial semantics and hence also do not permit horizontal operations. These languages could similarly introduce keywords or annotations for interpreting loops as to be executed using Parsimony SPMD semantics, but these would likely no longer use `#pragma` notation, as pragmas are generally meant to be safe to ignore.

8 Related Work

Compiler auto-vectorization has a long history [1, 14, 38, 39, 46]; loop vectorization and superword-level parallelism (SLP) vectorization are well researched classical optimizations that are enabled in many compilers today. Traditional loop vectorization has seen numerous advances such as outer-loop vectorization [24] and vectorization for interleaved [23] and misaligned [5] data access patterns. SLP vectorization [2, 15, 32] has been developed as a more flexible approach to loop vectorization. In contrast to these, Parsimony does not need to extract SIMD/vector parallelism from source code for vectorization thanks to its explicitly parallel SPMD semantics.

SPMD programming models with data parallel languages such as `ispc` [30] and ones with C++ SIMD extensions such as Sierra [19] have well-defined SPMD semantics but are more restrictive than Parsimony's proposed semantics. Prior work has also studied the use of GPU-focused SPMD programming models to target CPU SIMD units [7, 43]. Compiler passes such as the Whole Function Vectorizer (WFV) [13] support vectorization of arbitrary functions using SPMD-like semantics and Moll and Hack [21] extend this to support arbitrary unstructured control flows. However, unlike Parsimony, these passes do not provide precisely defined semantics.

9 Conclusion

SPMD programming within mainstream languages is an effective method of programming a CPU's SIMD/vector units rather than relying on custom languages or low-level intrinsic programming. In this work we demonstrate that introducing more rigorous semantics can facilitate the adoption of SPMD into widely-used general-purpose programming languages and toolchains. Our Parsimony compiler prototype shows that such rigor comes with little cost; code written using Parsimony semantics matches the performance of code written using `ispc` and AVX-512 assembly intrinsics.

Acknowledgments

The authors thank Matt Pharr and the anonymous reviewers for their helpful feedback. This work was partially funded by NSF awards CCF-2119069, CCF-2028851, and CNS-1763743.

A Artifact Appendix

A.1 Abstract

The artifact [12] comprises the source code of our prototype Parsimony compiler framework, our modified copy of the Simd Library and `ispc` benchmark suite, including the source code of the benchmarks we ported to Parsimony-enabled C++, the source code of several simple Parsimony correctness tests, and supporting scripts. The artifact is available publicly through an archived repository.

In the artifact, we include documentation that provides instructions for building our prototype compiler and using it to generate binaries for Parsimony-enabled C++ benchmarks. Additionally, the artifact includes documentation for the provided Parsimony API feature set, the Parsimony compilation flow, and instructions for extending the Parsimony API feature set. The artifact can be used to reproduce the key Parsimony results shown in Figures 4 and 5.

A.2 Artifact check-list

- **Algorithm:** A LLVM-based compiler framework for Parsimony, a SPMD programming model. Parsimony-enabled C++ implementations of benchmarks from the `ispc` benchmark suite and the Simd Library.
- **Program:** C++ benchmarks including `ispc` and Parsimony-enabled C++ implementations from the `ispc` benchmark suite and the Simd Library.
- **Compilation:** Clang 15.0.1, ispc 1.18.0, and GCC 8.5.0.
- **Data set:** All data sets used in the experiments are publicly available and are also included in the artifact repository.
- **Run-time environment:** Red Hat Enterprise Linux 8.7.
- **Hardware:** Intel®Xeon®Gold 6258R CPU server or other similar system with x86 AVX-512 (avx512bw) support.
- **Execution:** There should be no other application running on the system during performance profiling. Performance profiling for the LLVM scalar, LLVM auto-vectorized, Parsimony, and Hand-written AVX-512 implementations of the 72 Simd Library benchmarks takes 20 minutes to complete. Performance profiling for the `ispc`, Parsimony, and LLVM auto-vectorized implementations of the 7 `ispc` benchmarks takes 10 minutes to complete. All benchmarks are single threaded and are run serially one after the other.
- **Metrics:** The artifact reports execution time in milliseconds for the 72 Simd Library benchmarks and clock cycle counts for the 7 `ispc` benchmarks.
- **Output:** Figures 4 and 5. The artifact also provides an excel sheet that contains the raw data collected from the authors' machine which were used to generate these two graphs.
- **Experiments:** The artifact includes a set of scripts and instructions to start from cloning the Parsimony repository, building the prerequisites, building the Parsimony compiler, building and running the Simd Library and `ispc` benchmarks, and finally generating Figures 4 and 5.
- **How much disk space required (approximately)?:** 10 GB.
- **How much time is needed to prepare workflow (approximately)?:** Building the prerequisites, including LLVM 15.0.1, and building the Parsimony compiler and benchmarks can take 30 minutes in total.
- **How much time is needed to complete experiments (approximately)?:** Performance profiling for the 72 Simd Library benchmarks and the 7 `ispc` benchmarks can take 30 minutes in total.
- **Publicly available?:** Yes.
- **Code licenses (if publicly available)?:** Apache License v2.0 with LLVM Exceptions.
- **Workflow framework used?:** The LLVM pass framework is used in the Parsimony compilation flow. The build and run frameworks in the Simd Library and `ispc` benchmark suites are used for building Parsimony-enabled C++ benchmarks and for the performance profiling of these benchmarks.
- **Archived (provide DOI)?:** https://doi.org/10.5281/zenodo.7524279

A.3 Description

A.3.1 How to access. The artifact described in this paper is archived on Zenodo at https://doi.org/10.5281/zenodo.7524279 under an Apache License v2.0 with LLVM Exceptions. The tarball file `parsimony_artifact.tgz` downloaded from Zenodo can be decompressed with "`tar -xvf parsimony_artifact.tgz`" to create a directory `Parsimony/`. We refer to this `Parsimony/` directory below as the archived artifact repository. We also provide the Parsimony artifact as a public GitHub repository at https://github.com/NVlabs/Parsimony-CGO23. Interested users should refer to the GitHub repository for the latest version of Parsimony.

A.3.2 Hardware dependencies. The prototype Parsimony compiler needs to be built and run on a system with x86 AVX-512 support (specifically *avx512bw*) such as Intel®Xeon®Gold 6258R CPU.

A.3.3 Software dependencies. The Parsimony prototype compiler requires Clang/LLVM 15.0.1 to build. Parsimony also requires Z3Prover (v4.11.2) and Sleef(v3.5.1). `ispc` compiler v1.18.0 was used to compile the `ispc` implementations of benchmarks presented in Figure 4. The full list of software dependencies is provided in the `README.md` file in the archived artifact repository.

A.3.4 Data sets. All data sets used in the experiments are from the publicly available Simd Library repository and `ispc` benchmark suite, and are also included in the archived artifact repository.

A.4 Installation

A thorough setup guide for Parsimony is available at the `README.md` file in the archived repository.

A.5 Experiment workflow

The primary experiment consists of running and collecting performance measurements for Parsimony-enabled C++, LLVM auto-vectorized, `ispc`, LLVM scalar, and hand-written x86 AVX-512 implementations of benchmarks from the Simd Library and `ispc` benchmark suite. A thorough experiment workflow for the artifact is available at the `README.md` file in the archived repository.

A.6 Evaluation and expected result

The process to generate Figures 4 and 5 after cloning the archived artifact repository involves the following primary steps:

- Building all necessary prerequisites.
- Building the Parsimony compiler.
- Building the Simd Library and `ispc` benchmark suite.
- Collecting performance measurements for various implementations of the 72 Simd Library benchmarks and the 7 `ispc` benchmarks.
- Placing the collected results into the provided excel sheet to generate Figures 4 and 5.

These steps are explained in greater detail at the `README.md` file in the artifact repository.

In the generated Figure 4, the geomean performance of Parsimony implementations of `ispc` benchmarks relative to the geomean performance of `ispc` implementations of the same should be ≥ 90%. Similarly, in the generated Figure 5, the geomean performance of Parsimony implementations of Simd Library benchmarks relative to the geomean performance of hand-written x86 AVX-512 implementations of the same should be ≥ 90%. These results show that Parsimony implementations achieve performance parity with ispc and hand-written AVX-512 implementations of `ispc` benchmarks and Simd Library benchmarks respectively.

A.7 Experiment customization

The `compiler/README.md` file in the archived artifact repository provides documentation of the current Parsimony API feature set, the Parsimony compilation flow, and insight into extending the Parsimony API feature set. This document is provided as a starting point for porting more benchmarks to Parsimony-enabled C++, and for extending Parsimony.

References

[1] Randy Allen and Ken Kennedy. 1987. Automatic translation of Fortran programs to vector form. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 9, 4 (1987), 491–542.

[2] Yishen Chen, Charith Mendis, and Saman Amarasinghe. 2022. All You Need is Superword-Level Parallelism: Systematic Control-Flow Vectorization with SLP. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 301–315.

[3] Yishen Chen, Charith Mendis, Michael Carbin, and Saman Amarasinghe. 2021. 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*. 902–914.

[4] Bruno Coutinho, Diogo Sampaio, Fernando Magno Quintao Pereira, and Wagner Meira Jr. 2011. Divergence analysis and optimizations. In *2011 International Conference on Parallel Architectures and Compilation Techniques*. 320–329.

[5] Alexandre E Eichenberger, Peng Wu, and Kevin O'brien. 2004. Vectorization for SIMD architectures with alignment constraints. *ACM SIGPLAN Notices* 39, 6 (2004), 82–93.

[6] Wilson WL Fung and Tor M Aamodt. 2011. Thread block compaction for efficient SIMT control flow. In *2011 IEEE 17th International Symposium on High Performance Computer Architecture*. 25–36.

[7] Michael Haidl, Simon Moll, Lars Klein, Huihui Sun, Sebastian Hack, and Sergei Gorlatch. 2017. Pacxxv2+ RV: an LLVM-based portable high-performance programming model. In *Proceedings of the Fourth Workshop on the LLVM Compiler Infrastructure in HPC*. 1–12.

[8] Ihar Yermalayeu et al. 2019. The Simd Library. https://github.com/ermig1979/Simd. Accessed: 2022-1-9.

[9] Intel. 2022. Intel® 64 and ia-32 architectures software developer's manual - Volume 1-4. https://cdrdv2.intel.com/v1/dl/getContent/671200. Accessed: 2022-1-9.

[10] Intel. 2022. ispc: Intel SPMD Program Compiler. https://github.com/ispc/ispc/blob/v1.18.0/src/opt.cpp#L466. Accessed: 2022-1-9.

[11] Intel. 2022. ispc: Intel SPMD Program Compiler. https://github.com/ispc/ispc/releases/tag/v1.18.0. Accessed: 2022-1-9.

[12] Vijay Kandiah, Daniel Lustig, Oreste Villa, David Nellans, and Nikos Hardavellas. 2023. *Artifact for CGO'23 paper "Parsimony: Enabling SIMD/Vector Programming in Standard Compiler Flows"*. https://doi.org/10.5281/zenodo.7524279

[13] Ralf Karrenberg. 2015. Whole-function vectorization. In *Automatic SIMD vectorization of SSA-based control flow graphs*. 85–125.

[14] David J. Kuck, Robert H. Kuhn, Bruce Leasure, and Michael Wolfe. 1980. The Structure of an Advanced Vectorizer for Pipelined Processors. In *Proceedings of the 4th International Conference on Computer Software and Applications (COMPSAC)*. 709–715.

[15] Samuel Larsen and Saman Amarasinghe. 2000. Exploiting superword level parallelism with multimedia instruction sets. *ACM SIGPLAN Notices* 35, 5 (2000), 145–156.

[16] Chris Lattner. 2002. *LLVM: An Infrastructure for Multi-Stage Optimization*. Master's thesis. Computer Science Dept., University of Illinois at Urbana-Champaign, Urbana, IL.

[17] Yunsup Lee, Vinod Grover, Ronny Krashinsky, Mark Stephenson, Stephen W Keckler, and Krste Asanović. 2014. Exploring the Design Space of SPMD Divergence Management on Data-Parallel Architectures. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 101–113.

[18] Yunsup Lee, Ronny Krashinsky, Vinod Grover, Stephen W Keckler, and Krste Asanović. 2013. Convergence and Scalarization for Data-Parallel Architectures. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 1–11.

[19] Roland Leißa, Immanuel Haffner, and Sebastian Hack. 2014. Sierra: a SIMD extension for C++. In *Proceedings of the 2014 Workshop on Programming models for SIMD/Vector processing*. 17–24.

[20] LLVM Community. 2022. LLVM 15.0.1. https://github.com/llvm/llvm-project/releases/tag/llvmorg-15.0.1. Accessed: 2022-1-10.

[21] Simon Moll and Sebastian Hack. 2018. Partial control-flow linearization. *ACM SIGPLAN Notices* 53, 4 (2018), 543–556.

[22] Leonardo de Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. 337–340.

[23] Dorit Nuzman, Ira Rosen, and Ayal Zaks. 2006. Auto-vectorization of interleaved data for SIMD. *ACM SIGPLAN Notices* 41, 6 (2006), 132–143.

[24] Dorit Nuzman and Ayal Zaks. 2008. Outer-loop vectorization: revisited for short simd architectures. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*. 2–11.

[25] NVIDIA Corporation. 2017. NVIDIA Tesla V100 GPU architecture. https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf. Accessed: 2022-1-9.

[26] NVIDIA Corporation. 2022. NVIDIA CUDA C Programming Guide. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html. Accessed: 2022-1-9.

[27] OpenMP Architecture Review Board. 2013. OpenMP Application Program Interface Version 4.0. https://www.openmp.org/wp-content/uploads/OpenMP4.0.0.pdf. Accessed: 2022-1-9.

[28] Philippos Papaphilippou, Paul HJ Kelly, and Wayne Luk. 2021. Extending the RISC-V ISA for exploring advanced reconfigurable SIMD instructions. *arXiv preprint arXiv:2106.07456* (2021).

[29] Kariofyllis Patsidis, Chrysostomos Nicopoulos, Georgios Ch Sirakoulis, and Giorgos Dimitrakopoulos. 2020. RISC-V2: A Scalable RISC-V Vector Processor. In *2020 IEEE International Symposium on Circuits and Systems (ISCAS)*. 1–5.

[30] Matt Pharr and William R Mark. 2012. ispc: A SPMD compiler for high-performance CPU programming. In *2012 Innovative Parallel Computing (InPar)*. 1–13.

[31] Angela Pohl, Biagio Cosenza, Mauricio Alvarez Mesa, Chi Ching Chi, and Ben Juurlink. 2016. An evaluation of current SIMD programming models for C++. In *Proceedings of the 3rd Workshop on Programming Models for SIMD/Vector Processing*. 1–8.

[32] Vasileios Porpodas, Alberto Magni, and Timothy M Jones. 2015. PSLP: Padded SLP automatic vectorization. In *2015 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 190–201.

[33] Oliver Reiche, Christof Kobylko, Frank Hannig, and Jürgen Teich. 2017. Auto-vectorization for image processing DSLs. In *Proceedings of the 18th ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems*. 21–30.

[34] Jie Ren, Samyam Rajbhandari, Reza Yazdani Aminabadi, Olatunji Ruwase, Shuangyan Yang, Minjia Zhang, Dong Li, and Yuxiong He. 2021. ZeRO-Offload: Democratizing Billion-Scale Model Training. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. 551–564.

[35] RISC. 2021. RISC-V "V" Vector Extension. https://github.com/riscv/riscv-v-spec/releases/tag/v1.0. Accessed: 2022-1-9.

[36] Diogo Sampaio, Rafael Martins, Sylvain Collange, and Fernando Magno Quintao Pereira. 2012. Divergence analysis with affine constraints. In *2012 IEEE 24th International Symposium on Computer Architecture and High Performance Computing*. 67–74.

[37] Diogo Sampaio, Rafael Martins de Souza, Caroline Collange, and Fernando Magno Quintão Pereira. 2014. Divergence analysis. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 35, 4 (2014), 1–36.

[38] Randolph G. Scarborough and Harwood G. Kolsky. 1986. A vectorizing Fortran compiler. *IBM Journal of Research and Development* 30, 2 (1986), 163–171.

[39] Paul B. Schneck. 1972. Automatic Recognition of Vector and Parallel Operations in a Higher Level Language. *ACM SIGPLAN Notices* 7, 11 (1972), 45–52.

[40] Naoki Shibata and Francesco Petrogalli. 2020. SLEEF: A Portable Vectorized Library of C Standard Mathematical Functions. *IEEE Transactions on Parallel and Distributed Systems* 31, 6 (2020), 1316–1327.

[41] Standard C++ Foundation. 2020. ISO International Standard ISO/IEC 14882:2020(E) – Programming Language C++. https://isocpp.org/std/the-standard. Accessed: 2022-1-9.

[42] Nigel Stephens, Stuart Biles, Matthias Boettcher, Jacob Eapen, Mbou Eyole, Giacomo Gabrielli, Matt Horsnell, Grigorios Magklis, Alejandro Martinez, Nathanael Premillieu, et al. 2017. The ARM scalable vector extension. *IEEE micro* 37, 2 (2017), 26–39.

[43] John A Stratton, Vinod Grover, Jaydeep Marathe, Bastiaan Aarts, Mike Murphy, Ziang Hu, and Wen-mei W Hwu. 2010. Efficient Compilation of Fine-Grained SPMD-Threaded Programs for Multicore CPUs. In *Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 111–119.

[44] Giuseppe Tagliavini, Stefan Mach, Davide Rossi, Andrea Marongiu, and Luca Benini. 2019. Design and evaluation of SmallFloat SIMD extensions to the RISC-V ISA. In *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. 654–657.

[45] Konrad Trifunovic, Dorit Nuzman, Albert Cohen, Ayal Zaks, and Ira Rosen. 2009. Polyhedral-model guided loop-nest auto-vectorization. In *2009 18th International Conference on Parallel Architectures and Compilation Techniques*. 327–337.

[46] Dorothy Wedel. 1975. Fortran for the Texas Instruments ASC System. *ACM SIGPLAN Notices* 10, 3 (1975), 119–132.

[47] Wenzel Jakob. 2019. Enoki: structured vectorization and differentiation on modern processor architectures. https://github.com/mitsuba-renderer/enoki. Accessed: 2022-1-9.

Received 2022-09-02; accepted 2022-11-07