NORTHWESTERN
UNIVERSITY

Computer Science Department

Technical Report
Number: NU-CS-2025-33

August, 2025

Village: From High Level Parallelism to High Performance

Kir Nagaitsev, Griffin Dube, Karl Hallsby, Peizhi Liu, Qinze Jiang, Lucas Myers, David Krasowska, Alexander Butler, Ruiqi Xu, Peter Dinda

Abstract

Today's commonly used low-level parallel languages (LLPLs) allow for extensive performance engineering, resulting in performant programs that are closely tied to specific architectures. This process is exploding in complexity due to heterogeneity. In contrast, high-level parallel languages (HLPLs) allow for good algorithmic expression and avoid architectural ties, resulting in portable expression of algorithms, but typically with lower performance. Can we have both the architecture-independent expressive power of an HLPL and the performance characteristics of an LLPL? Can an HLPL implementation leverage the semantic richness of HLPL representations to map algorithms to complex or even custom heterogeneous hardware? To address these questions, we present Village, a new implementation of the classic NESL HLPL, as well as VIR-V, an extension to the RISC-V architecture specifically for Village. Village and VIR-V are based on VIR, a vector dataflow representation of the program, that is, by construction instead of analysis, very amenable to optimization and lowering to hardware. We then compare the measured performance of lightly-tuned, high-level Village and well-tuned LLPL implementations of several NAS and PBBS benchmarks, both on x64 and RISC-V, finding that Village achieves 50-70% the performance of well-tuned LLPL code. We also evaluate the performance of VIR-V with Village-compiled microbenchmarks, showing it can outperform CPU implementations by up to 40%.

This effort was partially supported by the United States National Science Foundation (NSF) under awards CNS-2211315, CCF-2119069, and CNS-2211508. Nagaitsev and Krasowska are Department of Energy (DOE) Computational Science Graduate Fellows. Liu is an NSF Graduate Research Fellow.

Keywords

high-level parallel languages, heterogeneous parallelism, dataflow, NESL, RISC-V, custom hardware

Village: From High Level Parallelism to High Performance

Kir Nagaitsev Griffin Dube Karl Hallsby Peizhi Liu Qinze Jiang
Lucas Myers David Krasowska Alexander Butler Ruiqi Xu Peter Dinda

Northwestern University

Abstract

Today's commonly used low-level parallel languages (LLPLs) allow for extensive performance engineering, resulting in performant programs that are closely tied to specific architectures. This process is exploding in complexity due to heterogeneity. In contrast, high-level parallel languages (HLPLs) allow for good algorithmic expression and avoid architectural ties, resulting in portable expression of algorithms, but typically with lower performance. Can we have both the architecture-independent expressive power of an HLPL and the performance characteristics of an LLPL? Can an HLPL implementation leverage the semantic richness of HLPL representations to map algorithms to complex or even custom heterogeneous hardware? To address these questions, we present Village, a new implementation of the classic NESL HLPL, as well as VIR-V, an extension to the RISC-V architecture specifically for Village. Village and VIR-V are based on VIR, a vector dataflow representation of the program, that is, by construction instead of analysis, very amenable to optimization and lowering to hardware. We then compare the measured performance of lightly-tuned, high-level Village and well-tuned LLPL implementations of several NAS and PBBS benchmarks, both on x64 and RISC-V, finding that Village achieves 50-70% the performance of well-tuned LLPL code. We also evaluate the performance of VIR-V with Village-compiled microbenchmarks, showing it can outperform CPU implementations by up to 40%.

Keywords

high-level parallel languages, heterogeneous parallelism, dataflow, NESL, RISC-V, custom hardware

1 Introduction

High-performance computing (HPC) has become increasingly reliant on heterogeneous hardware to make up for the performance gap left by slowing of Moore's law and the end of Dennard scaling. All of the top five systems in the most recent Top500 rankings have heterogeneous node architectures [1] that conform to this pattern. Additional trends such as composable computing [42], processing near memory [26, 36], and the use of a broader range of architectures, like RISC-V and ARM, in HPC [33] aim to further diversify the set of accelerators available in modern and future systems. According to the ACM Turing Award winners for 2019, we are living in a new golden age for computer architecture [27].

Motivation. And yet... All this innovation has also been making parallel systems increasingly difficult to program, as well as making the code to program them become increasingly divorced from parallel algorithm design. Modern high-performance parallel systems are commonly programmed using what are fundamentally low-level parallel languages (LLPLs), for example C/C++ with CUDA, OpenMP, and MPI. This makes the algorithm design opaque, ties the code closely to specific architectures, and increasingly restricts performance portability. This "architecture up" approach to parallel software development does however have the clear benefit of resulting in very performant code, in part because these languages are well suited to performance engineering down to essentially the microarchitectural level.

In contrast, high-level parallel languages (HLPLs), which are common in the "theory down" approach to parallel software development, hew very closely to parallel algorithm design and have few if any ties to specific architectures. Indeed, in cases such as functional languages, not even the execution model itself is preordained, much less its mapping to hardware resources, memory models, etc. This mapping flexibility garnered from rich semantics and the avoidance of premature lowering can be extensively leveraged in an implementation [2, 50]. Interestingly, not only are HLPLs able to express complex parallel algorithms (including non-numeric algorithms) without reference to the target architecture, but they can also provide clean expression of more traditional scientific algorithms [48, 49]. However, performance and the ability to do performance engineering are often compromised.

Can we have both the architecture-independent expressive power of an HLPL and the performance characteristics of an LLPL? Can an HLPL implementation leverage the semantic richness of the high-level representation to map algorithms to complex or even custom heterogeneous hardware?

Key insights and contributions. To study these questions, we have developed Village, a new implementation of the classic NESL nested data parallel HLPL. Village compiles NESL through an intermediate vector dataflow form, VIR, in which nodes are high-level operators that themselves may be parallel. The VIR dataflow graph (DFG) is amenable to a range of optimizations which may transform the DFG, including operator fusion and in-place updates. This is made possible through known guarantees which would otherwise need to be proven in an LLPL. In an LLPL, making such guarantees would typically involve complex alias analysis, and is often impossible.

The singular VIR DFG is also amenable to analysis using general purpose graph tools, not just those targeting compilation. Subgraphs of a VIR DFG may be split across hardware domains with edges representing communication. Because VIR operators come with straightforward semantics, VIR also makes it possible to change the data representation, allocation strategy, and other aspects of execution without involving the application programmer. At the lowest level, Village uses LLVM [34] and thus can target any architecture LLVM supports. Here, we use x64 and RISC-V.

This effort was partially supported by the United States National Science Foundation (NSF) under awards CNS-2211315, CCF-2119069, and CNS-2211508. Nagaitsev and Krasowska are Department of Energy (DOE) Computational Science Graduate Fellows. Liu is an NSF Graduate Research Fellow.

We have also developed VIR-V, a set of extensions to the RISC-V architecture that incarnate VIR operators as hardware and expose them as new RISC-V instructions. Our VIR-V augmented RISC-V machine runs under FPGA-assisted simulation in the Berkeley Firesim environment, which provides sufficient speed to boot and run complete Linux environments and off-the-shelf applications. Village can target this platform, and selectively lower VIR DFG nodes to these instructions, allowing high-level operations whose design is driven by the needs and opportunities of the "theory down" NESL language to be bridged to the "architecture up" of hardware logic design.[1]

Using Village, VIR, and VIR-V, we explore the above questions through the lens of single core performance, with and without hardware augmentation, on x64 and RISC-V.[2] We place ourselves in competition with known strong LLPL (usually C/C++/Fortran) implementations of classic HPC benchmarks as well as tuned LLPL implementations of more general parallel benchmarks. Can we approach or even beat the performance of such well-tuned codes?

Our contributions are as follows:

- We describe the design and implementation of a new, modern version of the classic NESL data parallel HLPL that ultimately targets LLVM.
- We show how to convert the stack-based implementation of NESL's original vector intermediate representation (IR) into our VIR dataflow IR. This includes a way to represent the necessary control flow aspects of recursion and conditionals in the dataflow graph framework.
- We show that VIR, by leveraging semantic properties from NESL, is amenable to a range of optimizations including vectorization, fusion, data representation flexibility, and in-place optimizations.
- We describe a foreign function interface (FFI) for NESL, both at the NESL and VIR levels, to allow the programmer to interface with external libraries that use the system ABI.[3] At the same time, the VIR-level FFI allows for automatic optimizations that leverage such libraries.
- We describe the design and implementation of the VIR-V extensions to the RISC-V architecture, as well as the mechanisms by which Village can automatically use them.
- We evaluate the single core performance of Village on several NAS and PBBS benchmarks, comparing performance with known strong implementations. We achieve about 50-70% of their performance on x64 and unmodified RISC-V platforms, starting from much more abstract implementations than the LLPL code we compare against.
- We evaluate Village-compiled microbenchmarks on VIR-V accelerators in a RISC-V machine on Firesim, showing VIR-V outperforms CPU implementations by up to 40% and demonstrating the lowering of VIR to custom hardware.

Experimental methodology and artifact availability. Our work is based on the design and implementation of Village, VIR, and VIR-V, as well as their evaluation through measurement and analysis on NAS [6, 30] and PBBS [43] benchmarks. Measurement is done on an x64 platform, an unmodified RISC-V platform, and an augmented RISC-V platform under Firesim. Section 5 provides more details.

Limitations of the proposed approach. While our work is on high-level parallel languages, this paper focuses mostly on single core performance (including the extended VIR-V core with its special hardware support for Village). Our goals here are to introduce Village, VIR, and VIR-V and to bridge the gap in basic performance that is often encountered when starting with a high-level parallel language, particularly in the HPC environment.

2 Background

2.1 LLPLs

Low-level parallel languages (LLPLs) generally refer to parallel programming models that are built upon low-level languages like C/C++ and Fortran. Some of the most common examples include CUDA, OpenMP, and MPI. Other examples of LLPLs that have built upon C/C++ and Fortran include UPC/C++ [13, 51], Coarray Fortran [39] and High Performance Fortran (HPF) [37]. Making an LLPL perform well on a particular architecture requires tying the code closely to that specific architecture, increasingly restricting the performance portability and readability of the code.

2.2 Performance Portability

Enabling performant execution across heterogeneous targets without requiring architecture-specific details defines performance portability. Often, engineering for performance portability still involves programming in an LLPL, with some high-level semantics augmented on top via a library [7, 14, 31, 41]. In these cases, the programmer is typically not exposing all the available parallelism of their algorithm to the compiler, but rather they are using a performance portability library to parallelize critical sections of their code as they see fit. In this way, performance portability layers allow the programmer to maintain a degree of separation from architecture-specific optimizations, but the code must still denote exactly when and how to parallelize. The result blends LLPL code with high-level library semantics, straying from a high-level algorithm description while simultaneously restricting how parallelism can be exploited. Additionally, programs of this nature become tightly coupled to the choice of data structures and representations as more optimizations are introduced.

2.3 HLPLs

High-level parallel languages (HLPLs) come with built-in operators and constructs to enable parallelism by design. Beyond enabling the high-level implementation of parallel algorithms, different HLPLs have a range of designs and goals.

Chapel [17] is an imperative language with parallel for-loop semantics, along with built-in parallel operators such as scans and reductions. Work sharing/stealing techniques are utilized to handle nested parallelism. The language is still quite reliant on the programmer to manually identify and specify parallelism opportunities, contrary to the goals of most HLPLs.

[1] Lowering subgraphs in addition to individual nodes is a work in progress.

[2] Although we do not leverage it for performance evaluation in this paper, our implementation can also use thread parallelism to make operator execution internally parallel, taking advantage of all the cores of a machine.

[3] The ability to invoke such libraries is essential to allow HLPL programmers to leverage the same tools that LLPL programmers do without much thought.

Unlike Chapel, most HLPLs are functional languages. When functional HLPLs are coupled with array/vector constructs, along with parallel map/reduce semantics, compiler analyses to extract parallelism can become greatly simplified. There is no longer a need to do complex loop analysis for the sake of extracting parallelism retrospectively at the compiler level.

NESL [11, 12] is a classic functional HLPL which was the first to introduce nested data parallelism, exposing the available parallelism of an algorithm to the compiler at all depths. Despite the influx of information from NESL about available parallelism, performance of the original implementation of NESL is suboptimal. This is due in part to the *flattening transformation*, which reduces the complex nested data parallel execution model to an execution model that is data parallel over segmented vectors. While this transformation simplifies the mapping of the nested parallelism to existing hardware, it has negative performance implications.

Numerous works have sought to overcome these limitations, particularly in the context of GPUs. The Nessie compiler [9, 10] is an implementation of NESL that targets GPUs, introducing some fusion and flattening optimizations. Futhark [28] is a functional data-parallel array HLPL with in-place updates which generates efficient OpenCL code for GPUs. Incremental flattening [29] for Futhark demonstrates how a multitude of code versions can be generated with different flattenings of nested parallelism, tuning the choice of flattening version to particular GPUs and datasets. Futhark does not support recursion by design, limiting itself to a fixed depth of nested parallelism. Lift [44] is an HLPL that uses grammar rewrite rules to encode algorithmic and hardware specific optimizations to take advantage of GPUs, exploring a range of valid rewrites.

Nested data parallelism has been adopted in other high-level languages, such as Data Parallel Haskell [15, 16] (now shipped in the Glasgow Haskell Compiler's mainline) and Manticore [20–22]. Beyond data parallelism, the MPL variant of Parallel ML [3, 38, 47] is an HLPL which demonstrates efficient fork-join task parallelism with in-place updates. In spite of this broad range of HLPLs that employ a range of parallelization techniques, all of these HLPLs still struggle to achieve performance parity on CPUs when compared against hand-tuned LLPLs.

2.4 NESL and VCODE

In this work, we build on the classic high level parallel language NESL [11, 12], whose syntax is reminiscent of Standard ML. NESL's nested data parallelism affords elegant expression of parallel algorithms with their available parallelism laid bare. In effect, what's added to the basic functional language framework is just three things: (a) collection types, (b) an unordered parallel filter / map / list comprehension expression that operates over collections, and (c) recursive parallelism, where function calls happen in parallel.

Figure 1 shows a classic example, a parallel quicksort. 9 lines of code capture two forms of parallelism: the parallelism of the pivot comparison and element reordering, and the parallelism of applying quicksort simultaneously on the resulting "<pivot" and ">pivot" results. The source code readily captures the fact that this implementation combines $O(n \log(n))$ work complexity and $O(\log(n))$ depth complexity (workspan). This fact is also readily

```
function quicksort(a) =
  if (#a < 2) then a
  else let
    pivot   = a[#a/2];
    lesser  = {e in a | e < pivot};
    equal   = {e in a | e == pivot};
    greater = {e in a | e > pivot};
    result  = {quicksort(v) : v in
                              [lesser,greater]};
  in result[0] ++ equal ++ result[1];
```

Figure 1: Quicksort in NESL. All the available parallelism in the algorithm is laid bare.

visible to the compiler. In the original NESL work, an important goal was to combine this algorithmic model with an abstract hardware model, giving the ability to derive the work and depth complexity of the program on corresponding hardware.

We leverage and extend the front-end of NESL, transform and extend its IR to be more suitable for modern optimization and hardware targets, introduce a range of optimizations based on our IR, and leverage the LLVM toolchain for additional optimization and architecture-dependence, all while inlining a new runtime with specialized data representations and allocation strategies.

The classic front-end of NESL transforms to an intermediate representation called VCODE. A key part of this process is the *flattening transformation* where all nested data parallelism is flattened to simple parallelism over segmented vectors. In the quicksort example, the recursion flattens to a *single* call (not two), with segmentation used to partition the input.

As mentioned earlier, this simplifying transformation sometimes has negative performance implications. The transformation has the ability to obliterate the cache locality of deeply nested calls, opting to always keep computing the results of operations over massive segmented vectors, rather than cutting off parallelism at a point and executing sequentially beyond that point. Incremental flattening in Futhark [29] is one solution to this problem targeting GPUs, as mentioned before. While we did not implement incremental flattening for NESL, we do implement other aggressive optimization techniques, including inlining, fusion, and in-place operations, to alleviate these performance limitations.

In the classic NESL implementation, an interpreter executes the VCODE IR directly; targeting a new hardware environment entails building an interpreter for it. Village, in contrast, ultimately generates a static executable.

Figure 2 describes the VCODE instruction set, as well as the changes that we made to it for VIR. In VIR, we refer to these as the *functional units* to make clear their connection to DFG level optimization and hardware generation. The algorithmic complexities given are for a PRAM and n refers to the length of the largest input vector to the functional unit. The attributes we list provide insight into possible optimization opportunities, including lowering into hardware. For example, *info-at-start* means all information needed to estimate the cost of the instruction is available when it is started, *count-at-start* means some computation needs to be done at start to estimate the cost, and *count-compute* means the cost of

Count (94)	Instruction Type	Example	Key Attributes	Work	Depth
38	Element-wise	`* INT`	streaming, unordered, info-at-start, no-overlap	$O(n)$	$O(1)$
14	Scans/Reductions	`+_SCAN`	streaming, ordered, info-at-start, no-overlap	$O(n)$	$O(\lg n)$
6	Permutations	`DPERMUTE FLOAT`	unordered, count-at-start, scatter/gather	$O(n)$	$O(1)$ to $O(\lg n)$
2	Extract/Replace	`EXTRACT INT`	unordered, count-at-start, resizing, scatter/gather	$O(n)$	$O(1)$ to $O(\lg n)$
1	Pack	`PACK INT`	monotonic,count-compute, resizing, gather	$O(n)$	$O(1)$ to $O(\lg n)$
2	Ranking	`RANK_UP`	unordered, info-at-start, overlapping-read/write	$O(n \lg n)$	$O(\lg n)$
2	Distribute/Index	`INDEX`	monotonic, count-at-start, resizing	$O(n)$	$O(1)$ to $O(\lg n)$
3	Segment Descriptor	`LENGTHS`	info-at-start	$O(n)$	$O(1)$
7	Basic I/O	`READ`	none	$O(n)$	$O(n)$
			VIR Changes		
6	Stack (removed)	`CPOP`	none	$O(1)$	$O(1)$
3	Conditional (removed)	`IF/ELSE/ENDIF`	forward-branch-only	$O(1)$	$O(1)$
3	Functions (inlined)	`FUNC/CALL/RET`	none	$O(1)$	$O(1)$
1	Gate (Mux) (added)	`GATE`	hardware-analog	$O(1)$	$O(1)$
1	Gated Recursion (added)	`CALLGATE`	hardware-analog	$O(1)$	$O(1)$
1	FFI (added)	`FDECL`	system-abi	$O(1)$	$O(1)$
4	Binary I/O (added)	`FREAD_BINARY`	none	$O(n)$	$O(1)$

Figure 2: Classifications, attributes, and typical asymptotic complexities for classic VCODE instructions and VIR's deletions, additions, and changes. In VIR, we also refer to these as the *functional units*.

the computation is only fully known after completing it. *Streaming* suggests a deep pipeline is possible, which is made easier given *unordered* operation and *no-overlap*ing memory references. *Resizing* and *scatter* or *gather* operations make optimization harder.

VCODE instructions operate on an abstract stack machine in which the data stack elements are arbitrary length vectors of basic types. A VCODE instruction conceptually pops vector operands from the stack and pushes vector results onto the stack. A VCODE vector is an arbitrary length sequence of a basic type (essentially, booleans, the integral types, and the floating point types), where the representation is left to the implementation. A VCODE vector of unsigned integers (e.g. `uint64_t`) can also be used as an segment descriptor, where the elements of the segment descriptor vector describe the partition of the index space of a separate data vector into subvectors, called segments. The elements of the segment descriptor denote the sizes of the segments in the data vector. Only one level of segmentation can occur.

Importantly, VCODE instructions are aware of segments when necessary, and operate using them. Due to how the VCODE vector operations are chosen, the work and depth complexity of the VCODE instructions themselves, as shown in Figure 2, are dependent only on the size of data vectors, not on the number of segments they contain. Consequently, even after NESL is lowered to VCODE, a considerable semantic richness remains.

The VCODE execution model has a separate call stack, and functions operate essentially as macro instructions in that they too pop vector operands from the data stack and push vector results onto the data stack. Beyond this, the only additional control flow construct is a conditional forward branch ("if-then-else"). Instructions with side effects are limited to execution abort and I/O instructions.

As a consequence of the constrained control flow model (limited to function calls, if-then-else, and abort) it is possible to easily transform from the stack-based VCODE IR to an IR that can be alternatively viewed as being a simple dataflow graph with recursion, or an SSA form with similarly restricted control flow. In either case, the data stack is eliminated and replaced with explicit communication edges between instructions. Even if-then-else can be transformed into a multiplexor combined with additional logic. Section 3.3 describes the transformation from original VCODE IR to our VIR form.

As we describe later, our VIR form allows for aggressive inlining such that an entire program or high-level function may be turned into a dataflow graph. Of course, recursion is a limiter here, and a recursive function cannot be entirely inlined. However, we demonstrate that a recursive functions can be augmented to include hardware-like enablement signals. This allows us to avoid, for example, computing both inputs of a multiplexor, or to halt a recursion even if the recursive function is partially and repeatedly inlined.

The VIR dataflow graph, which is made possible by VCODE's model, also enables novel lowering to hardware. There are three reasons for this. First, a node on the graph (VIR functional unit) is itself a relatively complex operation that is amenable to being directly implemented in hardware, which we provide an example of in Section 4.1. Second, subgraphs, either hot subgraphs from a specific program or common hot subgraphs from a corpus of programs, have the potential to be lowered into very deep hardware pipelines. Finally, the dataflow graph can be readily partitioned among different hardware units (accelerators) and machines without having to be concerned about control flow.

2.5 Intermediate Representations

A core element of enabling optimizations at the compiler level is the choice of intermediate representation to enable these optimizations. Our VIR dataflow form closely resembles gated-single assignment (GSA) [40] where ϕ-nodes are transformed to gate nodes (effectively multiplexors) that depend on the condition of the if-else

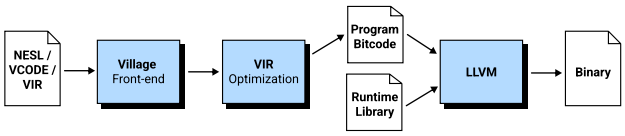

Figure 3: Our new compilation toolchain for NESL/VCODE.

branch in the original control flow. This VIR dataflow form is then amenable to a range of optimizations, including operator fusion and in-place updates. After performing these optimizations, VIR is lowered to LLVM IR [34] to exploit the additional optimizations and architecture targets that LLVM offers.

Optimizing within our own VIR dataflow representation is a matter of convenience which makes optimization and graph analysis quite straightforward. Many of the optimizations that we perform could also be achieved with different IR lowerings. For example, our operator fusion optimizations closely resemble what can be accomplished with affine loop fusion in MLIR [35]. SDFGs [8] are another IR which combines control-flow and dataflow to readily allow graph rewriting, pattern matching, and tiling, with the goal of enabling performance portability.

3 Compilation

We now describe the compilation process of the Village compiler. The compiler driver allows the user to start from NESL, VCODE, VIR, or any other language available in the LLVM installation (e.g, C, C++), and to generate either an executable, object file, or a library that allows invocation of a NESL function from another language. The compiler driver behaves in a uniform way with regard to options and inclusion of libraries or separately compiled object files. This allows us to assure that benchmarks written in different languages are compiled with the same LLVM system using the same options, and are linked with the identical support libraries. Figure 3 illustrates our toolchain.

3.1 Front-end

Compilation can start from NESL, VCODE, or VIR.

NESL. To compile NESL, we use a modified version of the original NESL system, which transforms from NESL to VCODE as described in more detail in Section 2.4. This system is written in Common Lisp and also includes a set of library macros that build additional primitives out of VCODE instructions. This transformation is very fast, typically taking less than a second for our benchmarks.

Because NESL is designed to generate code that is interpreted, several corner cases arise when the goal is pure compilation. For example, computed global constants need to be handled differently. Additionally, since NESL uses type inference, and we want to avoid generation of code that selects among type variants, we require an expression[4] to be supplied that allows us to determine the type signature of every function. We decorate each function with this information and leverage it throughout the compilation process.

[4]A typical expression might be the top-level function call (e.g., the equivalent of C's `main()`.) or an example call to a function we want to export. From the system ABI level (i.e., calling in from C) the NESL function looks like it takes a number of input arrays and produces a number of output arrays.

Another extension is the ability to specify and call foreign functions at the NESL level. This is critical to allow NESL programs to be able to leverage tuned node-specific libraries, such as FFTW or Intel's Signal Processing Toolbox, that are essential in HPC. This language-level support is backed with new VCODE instructions to declare external functions, including the characteristics of their arguments. The standard CALL function uses this information to allow calling them.[5] Independently of NESL-declared foreign functions, we can also use VCODE-level declarations and calls to allow us to optimize dataflow graphs by replacing subgraphs with calls to libraries that support the native ABI, for example BLAS.

We have also added several VCODE I/O instructions to facilitate much faster binary loads of data from files.

VCODE. We parse a VCODE input into an abstract syntax tree (AST) using a PEG, or parsing expression grammar, through the PEGTL library[6]. A PEG formally describes rules for recognizing strings in a language, producing a single valid parse tree by matching the first pattern it encounters, unlike context-free grammars which can be ambiguous when encountering multiple possible matches, leading to shift-reduce conflicts [23]. The AST hierarchically describes the VCODE program: each VCODE program is made up of a set of functions. Each function is comprised of VCODE instructions and each instruction contains information about the operation it is performing and the datatypes of vectors it is using in it's computation.

At this point, the execution model is still that of a stack machine, and data stack instructions such as POP and PUSH remain in place. In Section 3.3 we will describe the destackification of this form and the creation of the VIR dataflow graph form.

VIR and other representations. VIR itself has a simple serializer/unserializer. The file format is text-based. We can also output the representation using a graphviz format, to visualize dataflow graphs of programs. Finally, we can transform VIR into a simple vertex and edge model with minimal markup. The markup allows us to add data generated by our dynamic profiler (described later) as node and edge weights on the static graph. We can use then use form as input to general purpose graph mining tools.

3.2 Runtime system and library

The Village compiler leverages the Village runtime system and library, which is comprised of the implementations of VCODE's *functional units*, or support functions, used in code generation of VCODE instructions. The term functional unit here is used unironically—these library functions or compositions of them via the VIR DFG, could also be implemented as hardware.

The library's *functional units* are designed to facilitate invocations as calls to custom hardware accelerators, as well as to be extremely amenable to optimizations such as automatic vectorization and parallelization. Thanks to the semantics of the VCODE language, we are able to explicitly provide guarantees about the

[5]Calling out to a system ABI-supporting function (i.e., calling out to C) looks much the same as calling in from C: the function takes some number of input arrays and produces some number of output arrays. Allocation responsibility is determined via the declaration meta data. Neither calling into NESL or calling out to C reflects a complete native interface, but is rather geared to the VCODE execution model.
[6]https://github.com/taocpp/PEGTL

```
void vcode_mul_int(const int64_t *restrict lhs,
                   const int64_t *restrict rhs,
                   int64_t *restrict result) {
  uint64_t count = VCODE_VEC_LEN(result);
  for (uint64_t i = 0; i < count; i++) {
    result[i] = lhs[i] * rhs[i];
  }
}
```

Figure 4: Village library function for a vector multiply. This function performs the computation portion of a `* INT` VCODE instruction. 92 lines of code are omitted for simplicity. The directives, which follow from VCODE semantics, allow for aggressive, machine-specific optimization of the loop, inlining of it at call sites to the function, and joint optimization with calling code.

```
(%V1) = vcode_bpermute_int(%A1, %A2, %S1, %S2)
(%V2) = vcode_mul_int(%A3, %V1)
(%V3) = vcode_plus_reduce_int(%V2, %S3)
```

Figure 5: Example of SpMV dataflow in VIR text form

data used as inputs and outputs to a given library function. We annotate our library functions with useful guarantees that vectors do not alias to not burden the compiler with the task of proving this through the use of costly analyses such as alias analysis, which would be required when programming in a LLPL, and may potentially result in conservative or limited optimizations.

Figure 4 shows a version of an example function from the Village library that performs vector multiplication of two inputs and returns the result by reference. It is important to understand this is a Bowdlerized representation—the actual function currently comprises a macro that is 100 lines long. Given the directives shown (and others on the 92 lines not shown), Clang/LLVM can produce a software-pipelined, cache-cognizant version of the loop that uses AVX512 instructions on x64 machines, and this loop can be freely mixed with others at each call site of `vcode_mul_int`.

The Village library implements *functional units* for each of the VCODE instructions that performs a vector operation, while the Village compiler only generates calls into the library to perform operations. This means that in order to modify the target platform used by Village, all that is required is to modify the *functional units* corresponding to the instructions one wishes to change.

3.3 VIR

VIR is a vector dataflow IR, where recursive calls are augmented with enablement signals. More specifically, VIR describes the dataflow between *functional units*, which are described in Section 3.2. This dataflow representation is amenable to a wide range of optimizations. In this section we will describe how stack-based VCODE is transformed into VIR, as well as how control-flow/dataflow forms of VIR are practically interchangeable when lowering to VIR from NESL programs. A complete specification of the VIR is not provided, as the representation serves purely as an effective form for internal optimization within the Village compiler. Instead, pertinent, simplified examples are provided that demonstrate the utility of VIR.

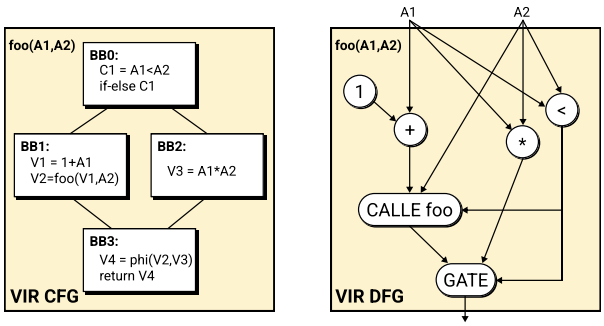

Figure 6: View of recursion from VIR CFG & DFG perspectives

Figure 5 shows one such example of simple VIR dataflow in text form. This example shows the neccesary *functional unit* interdependencies to perform a Sparse Matrix Vector (SpMV) multiplication, an example which will be discussed further later. The key is that the values being passed are either segment descriptors or segmented vectors, and a dependence between *functional units* represents an edge in the dataflow graph. Detaching the dataflow of these *functional units* from their internal implementations creates a simple, yet powerful, scheme for optimization, as we discuss further in Section 3.5.

3.3.1 Destackification. Traditional VCODE operates as an abstract stack machine. Each VCODE instruction pushes and pops values onto the stack, and there are also control operations such as POP, COPY, etc. for removing/copying/moving values directly. Since all VCODE functions and instructions have a fixed number of inputs and outputs, we can run the abstract stack machine to completion using dummy in/out values. When encountering if-else branches in the VCODE, we simply clone the current stack for either side of the branch and observe the in/out result at the end to produce ϕ-nodes accordingly. The result is a complete control-flow graph (CFG) of the computational VCODE instructions, with the control operations eliminated. Eliminating this stack was an essential step for performance, as it eliminates the need to run the abstract stack machine at runtime, wasting time on additional stack control operations.

3.3.2 Control-flow/dataflow relationship. Within the boundaries of control-flow basic blocks, dataflow relationships between all *functional units* of the basic block can be trivially identified. However, this is not the case across basic block boundaries due to ϕ-nodes and call instructions. We leverage GSA [40] to convert the control-flow form that we obtain from VCODE into a dataflow form which replaces ϕ-nodes with gate nodes and augments recursive calls with enablement signals. This is, in fact, the bare minimum that we need to make this transformation possible, given that VCODE programs do not have back edges, but instead must implement loops via recursion.

Figure 6 demonstrates an example of a recursive function in initial VIR CFG form, along with the corresponding DFG that we can transform to. Any recursive call gets transformed to a CALLE

(call with enable), where we pass the set of conditions that need to be true to reach the basic block in which this function call resides. In the case where the condition is false, the CALLE operation can return anything, because this result is discarded regardless. The GATE acts as a multiplexor, choosing the input that corresponds to the incoming condition. This comes directly from the translation of ϕ-nodes to GSA form.

Being able to transform into this DFG form is useful for a couple of reasons. First, it becomes possible to tile or perform fusion across what would typically be basic block boundaries. Second, it enables us to have a much more clear mapping to hardware without the involvement of control flow.

3.3.3 Emitting Program Bitcode. In practice, the dataflow form of VIR makes sense for direct hardware implementations, but it is impractical for CPU targets. When lowering to CPU targets, multiplexor gates of the IR are currently kept in control flow form for the sake of performance, to prevent execution of both sides of an if-else branch. LLVM bitcode is generated by making calls to the *functional unit* library implementations, with those calls ordered in a similar way to the original VCODE ordering, though mobility of placement is possible depending on dataflow dependencies. Depending on the hardware target, we make use of either software *functional unit* implementations for CPU, or hardware implementations for VIR-V.

3.4 Whole program linking and re-optimization

We generate the LLVM bitcode library alongside the bitcode program generated by the Village compiler. Once we have linked our program and library together as a single bitcode file, we compile this to a binary using LLVM's middle- and back-ends (as invoked via Clang). We are able to take advantage of LLVM's optimization passes to perform operations like inlining, loop unrolling, and autovectorization. These optimizations lead to nearly all *functional units* being inlined completely and further optimized.

3.5 Ease of VIR Optimization

Decoupling the dataflow of *functional units* in VIR from the implementations of those units makes for clear optimization opportunities, as we demonstrate in this section.

3.5.1 Vectorization. One clear benefit of implementing well-defined, constrained runtime *functional units* is that they can be extremely fine-tuned (or even implemented in hardware), as long as they accomplish their task. One example of this is the clean vectorization of these functional units, given that the runtime function loops can be written in a way that is highly amenable to vectorization by LLVM. Note that none of this burden is placed on the NESL programmer, it is entirely within the Village runtime library implementation. Figure 7 shows an example of vector assembly produced by `vcode_mul_int`. This is just one example of how the programmer can unlock architecture-specific optimizations with an HLPL, without any extra effort.

3.5.2 Fusion. Nested data-parallelism over vectors is contained entirely within *functional units*, thanks to the NESL flattening transformation, making each vector encountered in those units a vector of segments in reality. From the perspective of building *functional units* to execute efficiently on CPU targets, this has some downsides. Each *functional unit* can be implemented as a loop over the input vectors in C, as was shown in the `vcode_mul_int` example of Figure 4. However, if multiple operations are being applied to a vector in sequence, this means that each of these operations will have its own loop, and intermediate results will each require a new vector allocation without any in-place optimizations present.

Our primary solution to this dilemma is fusion. While prior work has also dealt with this problem via incremental flattening [29], we find that aggressive fusion, combined with other optimizations shown here, is often sufficient.

Our fusion pass on the VIR works by automatically identifying subgraphs that can be merged into single fused tiles, combining the *functional units* into one single unit which we generate automatically. Figure 8 demonstrates how this works with a cross product example taken from the Convex Hull PBBS benchmark. The cross product before fusion consumes ~ 40% of runtime in this benchmark.

We perform automatic fusion by identifying nodes in the DFG which are used exactly once by subsequent nodes. In Figure 8a, the first step of this process is identifying that the left and right

```
x86_64
vmovdqu (%rsi,%rax,8),%ymm0
vpmullq (%rdi,%rax,8),%ymm0,%ymm0
vmovdqu 0x20(%rsi,%rax,8),%ymm1
vpmullq 0x20(%rdi,%rax,8),%ymm1,%ymm1

ARMv8
fmul z0.d, z0.d, z2.d
ld1d { z3.d }, p0/z, [x13, x10, lsl #3]
fmul z1.d, z1.d, z3.d
st1d { z0.d }, p0, [x2, x10, lsl #3]

RISC-V
addi t1, t0, 16
vfmul.vv v8, v8, v10
vfmul.vv v9, v9, v11
vse64.v v8, (t0)
```

Figure 7: Vector instructions for x86_64, ARMv8, and RISC-V generated by the Village library via autovectorization in `clang`. Note that we are able to generate AVX512VL instructions for x86, SVE instructions for ARM, and V instructions for RISC-V with no additional effort.

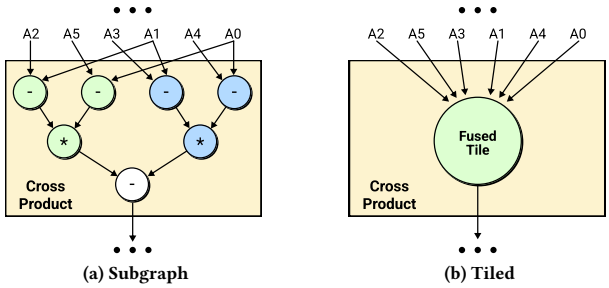

Figure 8: Fusion of cross product subgraph in VIR

```
for (uint64_t i = 0; i < count; i++) {
  out[i] = ((A2[k] - A1[k]) * (A5[k] - A0[k]));
}
```

Figure 9: Generated loop fusion for cross product subtrees

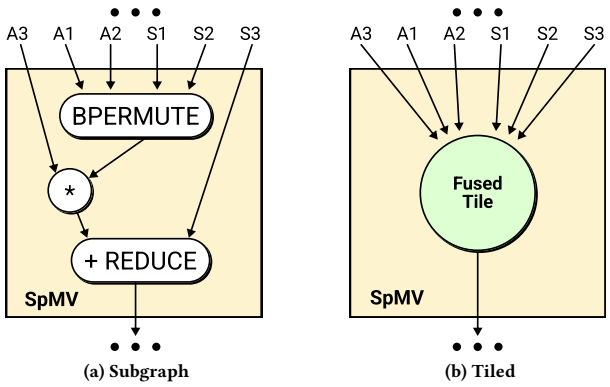

Figure 10: Fusion of SpMV subgraph in VIR

subtrees of the root node are fusion opportunities. Figure 9 shows C code that we automatically generate for either of these subtrees, executing the fused version of these operations. It is clear from this example that this avoids the cost of computing 3 separate loops for each subtree, and it avoids the need for storing intermediate results in new vectors.

However, rather than stopping there, our fusion pass actually saves these fusion opportunities, then continues progressing down the program DFG. We then encounter the root node in Figure 8a, and find that both of its children are valid fusion subtrees which are consumed by the root node. As a result, we can perform an even larger fusion, composing all 7 loops into a single, automatically generated loop for efficient CPU execution. We do not show this generated code, but note that it can be easily inferred based on Figure 9.

Beyond straightforward element-wise *functional units* such as addition, multiplication, and subtraction, our fusion pass can fuse more complex units together. Figure 10 shows a set of *functional units* used to perform SpMV multiplication in the NAS CG benchmark. Here, the permutation is selecting elements out of a dense vector corresponding with sparse row indices, then the multiplication followed by the plus reduce perform a dot product between that and the sparse row. Our fusion pass is capable of automatically fusing these operations by performing the indirection, multiplication, and reduction in a single set of nested loops over the segments and elements.

3.5.3 In-place Updates. Our VIR optimizer is capable of generating in-place versions of operations. A particular input of an operation can be marked as the in-place input if it is used once and consumed by the operation which we are making in-place. This input must also be the same length as the expected output vector size. The process of determining this is not described here, as it is extremely similar to the analysis needed for fusion. Prior works such as Futhark [28] and Parallel ML [25] have demonstrated already how in-place updates can benefit HLPLs.

3.5.4 Data Representation Flexibility. The fundamental data structure managed by the runtime is an arbitrary length unsegmented vector. The compiler generates calls to allocate vectors of specific types and sizes. For example, to invoke a VIR `+FLOAT` instruction, it will generate a call to allocate a FLOAT result of the correct size, and then invoke the `+FLOAT` implementation with the resulting pointer as the destination.

This pointer returned by the allocator is actually to metadata about the allocation. While the metadata tracks the type and size, a more interesting aspect is that it tracks the data representation and whether or not the data has been *allotted* and whether or not the representation has been *materialized*. The first concept is straightforward: we do eager allocation of the metadata (which includes the representation) and lazy allotment of the data. The data, which we call the raw representation, is the usual array of the type one might expect. In order for us to access the raw representation, it must be allotted and materialized.

An instruction implementation can choose not to allot and/or materialize its result and instead describe the result data using an alternative representation. We currently support three alternative representations:

- SCALAR: This representation encodes just one copy of a value when it is known that all result elements take on the same value, avoiding redundant memory and computing costs.
- RLE: This run-length encoding representation stores only one copy of each repeated value, and its repeat count. Note that RLE is analogous to SCALAR, but for a segmented vector.
- INDEX: This representation avoids the memory and computational costs involved with sparse arrays and non-array-based algorithms by storing `start:stride:end` sequences.

Instruction implementations that consume alternative representations must be built to understand them. While this could lead to a combinatorial explosion in complexity, in practice we can focus on specific instruction categories or instructions that matter. Any instruction's baseline implementation simply materializes all input vectors, at which point it knows it can deal with the raw representation. Our system tracks materializations of alternative encodings and their sizes, creating a profile of materialization costs for every run. This is a flat profile in which each instruction and encoding pair is included, ranked by the total size (in vector elements) of materializations it involved. Using this profile, the developers can incrementally add alternative representation support where it will have the most effect. There is a push/pull effect in that one developer can add a new representation for an instruction's output, while another can then find the downstream instructions most likely to benefit from being able to support the new representation.

Custom allocator. An allocated vector may be used in multiple places because the fanout for a given instruction in the data flow graph may be > 1. To enable this, we use reference counting, with

the compiler generating calls to the allocator's increment and decrement functions. Due to this reuse the allocation lifetime can be complex, thus requiring heap allocation. However, the structure of the dataflow graph also creates a "stack-like" flavor to some extent.

When using the native allocator (glibc malloc and jemalloc were tried) we found that it was common for the allocator to give pages back to the kernel only to reacquire them soon afterward. Even if the explicit support for converting large mallocs directly into mmaps and large frees directly into munmaps was disabled, these allocators would regularly do this by readjusting their brks. The problem is that when physical pages are reallocated to the process, the kernel must not only add them to the page table, but also clean them. Linux does this on demand. Consequently, a large fraction of execution time, sometimes more than half of execution time and over 90% of kernel time was spent in recycling, the kernel zeroing pages that we had previously returned.

To address this issue, we developed a custom implicit free list allocator for the Village runtime. This allocator is different in three ways. First, it avoids eagerly returning pages to the kernel, thus avoiding the recycling cost noted above. Second, it optimizes for large allocations, using multiple-cacheline alignments that play well with the lazy allocator, metadata system, and data representation system. Finally, the free-list search on allocation leverages the "stack-like" allocation/deallocation behavior.

An additional benefit of our custom allocator is that it provides a straightforward locus of control over memory. For example, it can use `MAP_LOCK` functionality to force immediate allocation of pages instead of leaving this up to a heuristic. A future extension of the allocator might use functionality like HPMMAP [32] to directly control physical memory allocation in the Linux context. Beyond the ability to further reduce allocation and paging overhead, this kind of mechanism would also allow us to provide allocations that are simultaneously accessible by host CPUs and DPUs on PIM memory, such as UPMEM [26].

3.6 FFI

We have extended the NESL language with a foreign function interface (FFI) to enable calling functions written in other languages. Creating bindings for a foreign library still requires manual effort and an understanding of Village's underlying execution model, but this is a one-time cost; once written, such bindings require no special knowledge or effort to be used by a NESL programmer, and operate harmoniously with NESL's high-level parallel constructs. To demonstrate the FFI, we have created a simple binding to the FFTW library [24] for performing one-dimensional FFTs, and used this binding in our NESL implementation of the NAS FT benchmark.

We define our own *wrapper* functions around FFTW's because our FFI does not currently allow calling *arbitrary* externally-defined functions. Instead we require external libraries to adopt conventions specific to our implementation and runtime. In exchange, the external function implementation is given *full* access to internal runtime details like reference-counts and specialized data representations (Section 3.5.4), and is able to directly access vector contents without unnecessary copying.

At runtime, each argument-to and result-from `dft_1d` is represented as three vectors: a vector of real components, a vector of imaginary components, and a segment descriptor. Each use-site in the FT benchmark calls `dft_1d` in parallel over a 3D matrix, but flattening transforms this to a *single* call, with the segment descriptor capturing all the FFTs to be performed—in effect, flattening *delegates* the source program's parallelism to the external function implementation. Furthermore, our optimized SCALAR representation means that the segment descriptor can be represented as a single value and a count (since the FFTs are of uniform size), directly exposing the *regularity* of parallelism without constraining language semantics.

Our FFI allows our implementation of `dft_1d` to efficiently bridge the semantic gap between NESL and FFTW's C interface. We can delegate the source program's parallelism to FFTW's batch-execution capabilities, FFTW can operate directly on the input data without copying. Further, when VIR can show that the input vectors will never be used again, FFTW may reuse them as outputs.

4 VIR-V Hardware

We have implemented a subset of the VCODE instruction / VIR functional units shown in Figure 2 as hardware in a RISC-V processor design. The current implementation has 33 of the instructions, including **all** Integer and Boolean element-wise and scan/reduction operations, and one of the permutations. These are visible as new instructions at the ISA level, plus there are several additional instructions for configuring VIR-V. To use an instruction, a variant of Figure 4 is built that simply is a wrapper for an assembly block that sets up and invokes the instruction. The Village compilation process can then select between the VIR-V wrapper or the baseline run-time function for each individual use of the instruction.

4.1 Understanding RISC-V & RoCC Accelerators

RISC-V is an open-source instruction set architecture (ISA) [45]. One of its key features is that it has customization *built into* the specification, allowing researchers to develop new ideas without needing to create a whole new specification and without breaking already-existing applications.

We leverage Rocket-Chip [4], a complete tile-based System-on-Chip with a Rocket-Core at its heart. A Rocket-Core is a single-issue five-stage in-order pipelined implementation[7] of RISC-V GCB ISA and RISC-V's privileged architecture, supports all privilege modes, and boots Linux.

One way that Rocket-Core supports the ISA's customization is by providing a Rocket Custom Coprocessor (RoCC) interface. RoCC instructions use the custom ISA encoding space to transparently provide instruction-level support for custom hardware. Rocket-Chip handles the work of dispatching a RoCC instruction, blocking the core, and ensuring the whole remains stable.

RISC-V recently ratified their vector extension (RISC-V V Extension) [46]. This implements a vector machine akin to the Crays of yesteryear. Many parts of the extension are amenable to VCODE and have a direct 1:1 mapping. An LLVM toolchain that has V support will readily allow us to target this extension. While the V-extension is a good target for Village, the flexibility offered by a

[7] Rocket-Core is a "classic 5-stage RISC CPU" like the ones taught and developed in computer architecture courses.

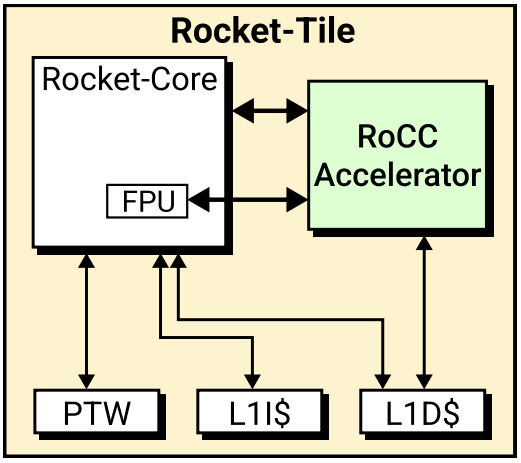

Figure 11: RoCC Accelerator Connection to RISC-V

blank-slate RoCC implementation *dedicated* to VCODE instructions offers more interesting optimization opportunities.

Rocket-Chip's tile-based implementation means that the main core is part of a larger tile structure. Depending on the configuration of the tile, a single tile may have tile-local L1 caches, a floating-point unit, and a page table walker; and each of these components may have different parameters between tiles. Figure 11 shows how RoCC accelerators are just another member of the tile; almost an equal to the main Rocket-Core in terms of capabilities. RoCC accelerators can connect to the tile-local L1 caches, the tile-shared L2 cache, the tile's floating-point unit, the tile's page table walker, and even the system bus (for RAM access). Eventually, all tiles get connected together using TileLink [18].

The RISC-V specification defines four custom opcode encoding spaces. Rocket-Chip "uses" these encoding spaces to support RoCC instructions, allowing up to four accelerators to be connected to a single Rocket-Core. This design means RoCC instructions flow through the core's pipeline like any other instruction until they reach the execute stage, where they then get dispatched to the accelerator. This allows the accelerator to potentially operate *in parallel* with the main core, and does not require custom support from compilers/assemblers. The RoCC interface allows the addresses of integer registers, their contents, and several control signals to be passed to the accelerator.

4.2 VIR-V/RoCC

Our VIR-V implementation is a RoCC accelerator. Our accelerator decodes and verifies the instruction, to ensure it is one that we support. If the instruction is not valid, then the accelerator delivers an interrupt to the main core which is handled like any other RISC-V interrupt. Overall, this accelerator totals just 1200 lines of Chisel [5] code.

Recall that the VCODE instructions / VIR functional units operate over in-memory segmented vectors of arbitrary length. VIR-V/RoCC instructions do the same, assuming the raw encoding of the data (that is, the source vector operands must be materialized, and destination vector operands must be allotted. When executing an instruction, VIR-V/RoCC accesses memory directly using the tile's L1 data cache.

VIR-V/RoCC is a lane-oriented SIMD processor, which is able to process multiple 64-bit data elements at a time. VCODE's higher-level notion of vectors is very amenable to hardware, and VCODE's restriction on data types means that most hardware can be reused across most instructions without modification. In particular, we rely on NESL's (and therefore VCODE's) assurance that no two vectors alias.

4.3 Continuing Enhancement

We are currently extending VIR-V/RoCC to support floating-point operations. The same frameworks that we have already developed will apply to these, and the primary concern is which FPU implementation to use. There are two potential options to implement this feature. One is integrating the floating-point units (FPUs) in Chipyard with our accelerator, which were developed in the fully Linux-capable Rocket cores supporting RV64GC and will be more compatible with our accelerator [4]. The other is generating a customized FPU, which will be helpful to meet specific requirements of VCODE floating-point arithmetic. To customize an FPU, FloPoCo [19] is an available FPU generator that provides Verilog implementations of floating-point operations following the IEEE 754 standard. We are evaluating the feasibility and engineering effort of the options mentioned above.

We next intend to make VIR-V/RoCC into a deeply pipelined streaming processor. This would allow us to leverage the inherent parallelism of the relevant VCODE / VIR instructions, as noted in the work and depth complexity breakdowns of Figure 2.

Also on the agenda is to make the components of VIR-V/RoCC composable so that an entire VIR DFG subgraph could be mapped into the accelerator and deeply pipelined. We hope to be able to offload large, commonly occurring subgraphs of VIR programs.

It will also be important to investigate *how* this accelerator is connected to memory. If the L1 data cache is used, we can take advantage of locality, as the main processor may use the vector immediately before or after the accelerator. However, this could over-utilize the L1 cache, causing slowdowns due to large reads and writes. If this is the case, connecting to the L2-crossbar, which is directly connected to the system bus and bypasses the L1 data cache, is the better implementation. If possible, connecting directly to main memory would be the most appropriate for operations over very large vectors.

5 Results

We evaluate Village by measuring its execution time on a number of benchmarks, then comparing these results with an equivalent version written in C. All runtimes are normalized to the C implementation's performance. We run all benchmarks on an x86_64 machine, and subsets of these benchmarks on both unmodified and augmented RISC-V machines. Our x86_64 machine is an AMD EPYC 7443P with 48 threads and 256 GB RAM. Our unmodified RISC-V machine is a SOPHON SG2042 with 64 threads and 128 GB RAM. We also evaluated Village in Firesim, an FPGA-accelerated cycle-accurate hardware simulator, on a 64-bit Rocket-Core with an 8-lane VIR-V accelerator with 4GiB of RAM.

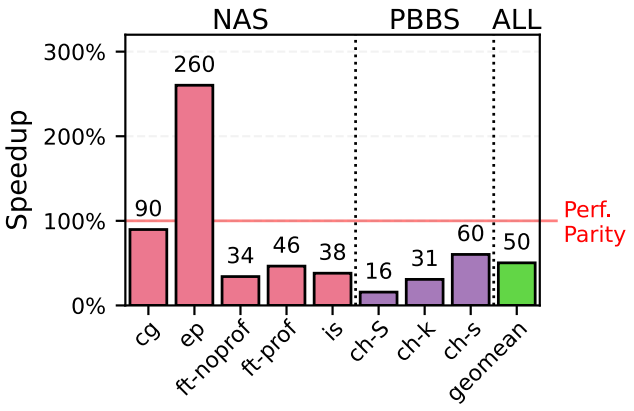

Figure 12: Speedup of Village compared to baseline C/C++ on x64 (the runtime of each C/C++ benchmark is divided by the runtime of the respective Village benchmark). Both are serial versions. Higher is better for Village. Village comes close to, and sometimes exceeds, performance parity of 100%, where the runtimes of Village and baseline are equal.

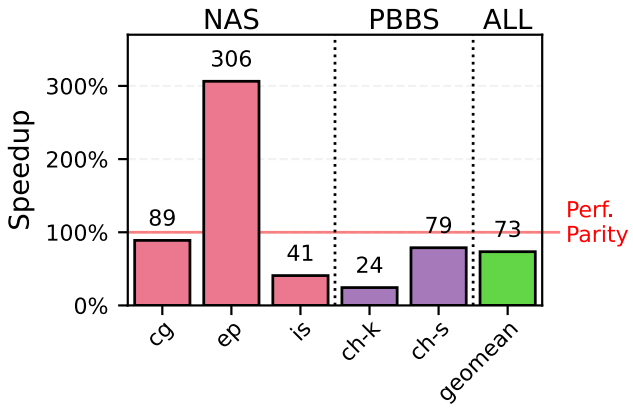

Figure 13: Speedup of Village compared to baseline C/C++ on unmodified RISC-V. These are serial benchmarks, with higher being better for Village.

We choose the NAS parallel benchmarks[8] with class `B` to represent common high-performance and parallel workloads, and the Problem Based Benchmark Suite (PBBS) with `large` class to demonstrate parallel algorithms that NESL can efficiently represent. All benchmarks for C and Village were compiled with equivalent levels of optimization. All benchmarks were executed serially, as Village currently supports vectorization but is not yet capable of fully exploiting multi-core parallelism. This benchmarking is meant to demonstrate the feasibility of Village even in a serial context, which shows the flexibility of the NESL language and ability of our implementation to come within a reasonable performance difference of popular LLPLs. While we understand that these implementations often outperform our toolchain, they are not designed with an inherent ability to express parallel algorithms like HLPLs are.

5.1 x64 Benchmarking

The results of Village compared to C/C++ baseline benchmarks on x64 are shown in Figure 12. Village comes close to achieving performance parity with these LLPLs, despite being written in high-level, minimally optimized NESL. Aside from FT, which makes use of our FFI to demonstrate calling a C library from NESL, all of the Village versions are written entirely in NESL. Benchmarks are selected which implement approximately the same algorithm between Village and baseline, to make a fair comparison. Of course, the NESL code implements these algorithms in a more high-level manner with many more degrees of compiler optimization freedom.

Almost all the benchmarks benefited from our automatic fusion optimization, sometimes reducing runtimes by up to 50% for benchmarks like CG and EP with core kernels that consume most of the runtime. The EP version of Village performs so much better both because of these strong fusion benefits, and strong vectorization of Village operators is likely a contributor as well, given that the hot loop of the EP baseline we are comparing against was not designed to be vectorizable. The variation of Convex Hull (CH) depending on the graph input can be attributed to the fact that the time consuming portions of a program compiled by Village do not neccesarily align with those of an LLPL, leading to variance based on workload characteristics. Convex Hull spends more time managing memory, but is highly optimized in its core dot product kernel due to fusion.

Fourier Transform (FT) is a special case in that we demonstrate the capabilities of our FFI by making calls from NESL to an external FFT library, FFTW. Benchmark `ft-noprof` runs without profiling, and `ft-prof` allows FFTW to profile for one run before collecting performance results, showing slight improvement. Our Village implementation computes a 3D FFT by transposing its input to make successive 1D FFT calls to FFTW. This demonstrates how one could algorithmically describe the approach of scaling a 1D FFT to being multi-dimensional, and we show that this can be done without substantially harming performance. This benchmark stands slightly apart from the rest because it does harness the performance benefits of using the external FFTW library, unlike the remaining Village benchmarks which are written entirely in NESL.

5.2 RISC-V Benchmarking

Results for unmodified RISC-V are shown in Figure 13. These results closely resemble the x64 results, with a geomean of 73%, even closer to performance parity than on x64. However, note that FT and `ch-S` benchmarks are not included. FT is not included because it was primarily a demonstration of NESL FFI functionality. `ch-S` is omitted because of segfaults by the baseline PBBS implementation with this input on the RISC-V machine, though the NESL version runs fine. The key takeaway of these results is that they are extremely similar to those on x64, as should be expected, which confirms that NESL can be compiled to run efficiently for this architecture.

5.3 VIR-V (RoCC) Benchmarking

The performance of VIR-V on microbenchmarks is shown in Figure 14. VIR-V accelerators show promise, consistently performing

[8] https://github.com/benchmark-subsetting/NPB3.0-omp-C

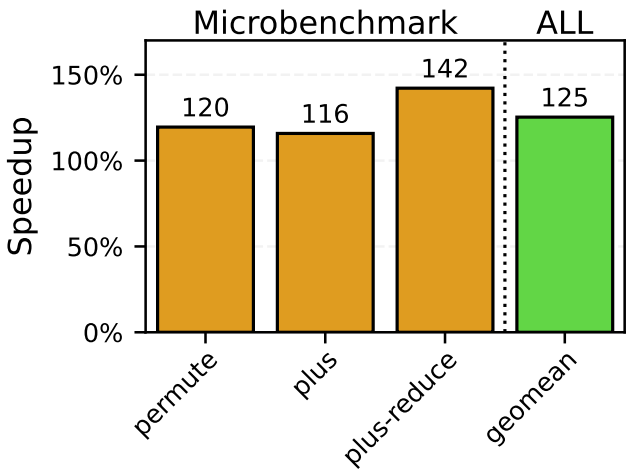

Figure 14: Speedup of microbenchmarks with VIR-V accelerators compared to CPU execution without VIR-V, both executed on Firesim. 100% indicates equal runtimes, with higher being better for VIR-V.

above CPU execution. These microbenchmarks were generated entirely using Village, demonstrating that our *functional units* can be implemented directly in hardware, and can be lowered to via our toolchain. The CPU versions which we compare against utilize our optimized C runtime functions without invoking the VIR-V accelerators.

6 Conclusion

We have presented Village, a new implementation of the classic NESL HLPL, as well as VIR-V, an extension to the RISC-V architecture specifically for Village. Village and VIR-V are enabled by VIR, a vector dataflow IR that is amenable to a range of optimizations by construction. We demonstrate that VIR is amenable to a range of optimizations, allowing the original HLPL code to remain in its readable, architecture-agnostic form while the compiler takes care of mapping the code effectively to a range of hardware targets. Village shows great promise, often achieving 50-70% the performance of well-tuned LLPLs on NAS and PBBS benchmarks. VIR-V outperforms CPU implementations of microbenchmarks by up to 40%, demonstrating a mapping of Village directly to custom hardware.

References

[1] Top500 2024. *TOP500 List - November 2024.* Top500. https://top500.org/lists/top500/2024/11/

[2] Umut A Acar, Arthur Charguéraud, , Adrien Guatto, Mike Rainey, and Filip Sieczkowski. 2018. Heartbeat Scheduling: Provable Efficiency for Nested Parallelism. In *39th ACM SIGPLAN Conference on Programming Language Design and Implementation.*

[3] Jatin Arora, Sam Westrick, and Umut A. Acar. 2021. Provably Space Efficient Parallel Functional Programming. In *Proceedings of the 48th Annual ACM Symposium on Principles of Programming Languages (POPL)".*

[4] Krste Asanović, Rimas Avizienis, Jonathan Bachrach, Scott Beamer, David Biancolin, Christopher Celio, Henry Cook, Daniel Dabbelt, John Hauser, Adam Izraelevitz, Sagar Karandikar, Ben Keller, Donggyu Kim, John Koenig, Yunsup Lee, Eric Love, Martin Maas, Albert Magyar, Howard Mao, Miquel Moreto, Albert Ou, David A. Patterson, Brian Richards, Colin Schmidt, Stephen Twigg, Huy Vo, and Andrew Waterman. 2016. *The Rocket Chip Generator.* Technical Report UCB/EECS-2016-17. University of California at Berkeley. http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-17.html

[5] Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avižienis, John Wawrzynek, and Krste Asanović. 2012. Chisel: Constructing hardware in a Scala embedded language. In *DAC Design Automation Conference 2012* (2012-06). 1212–1221. doi:10.1145/2228360.2228584 ISSN: 0738-100X.

[6] D. Bailey, E. Barszcz, J. Barton, D. Browning, R. Carter, L. Dagum, R. Fatoohi, S. Fineberg, P. Frederickson, T. Lasinksi, R. Schreiber, H. Simon, V. Venkatakrishnan, and S. Weeratunga. 1994. *The NAS Parallel Benchmarks (NAS 1).* Technical Report RNR-94-007. NASA.

[7] Michael Bauer, Sean Treichler, Elliott Slaughter, and Alex Aiken. 2012. Legion: Expressing locality and independence with logical regions. In *SC '12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis.* 1–11. doi:10.1109/SC.2012.71

[8] Tal Ben-Nun, Johannes de Fine Licht, Alexandros N. Ziogas, Timo Schneider, and Torsten Hoefler. 2019. Stateful dataflow multigraphs: a data-centric model for performance portability on heterogeneous architectures. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Denver, Colorado) *(SC '19).* Association for Computing Machinery, New York, NY, USA, Article 81, 14 pages. doi:10.1145/3295500.3356173

[9] Lars Bergstrom, Matthew Fluet, Mike Rainey, John Reppy, Stephen Rosen, and Adam Shaw. 2013. Data-only Flattening for Nested Data Parallelism. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP 2013).*

[10] Lars Bergstrom and John Reppy. 2012. Nested Data-parallelism on the Gpu. In *Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming (ICFP 2012).*

[11] Guy E. Blelloch, Siddhartha Chatterjee, Jonathan Hardwick, Jay Sipelstein, and Marco Zagha. 1994. Implementation of a Portable Nested Data-Parallel Language. *J. Parallel and Distrib. Comput.* 21, 1 (April 1994), 4–14.

[12] Guy E. Blelloch and John Greiner. 1996. A Provable Time and Space Efficient Implementation of NESL. In *Proceedings of the International Conference on Function Programming (ICFP).*

[13] William W Carlson, Jesse M Draper, David E Culler, Kathy Yelick, Eugene Brooks, and Karen Warren. 1999. *Introduction to UPC and language specification.* Technical Report. Technical Report CCS-TR-99-157, IDA Center for Computing Sciences.

[14] H. Carter Edwards, Christian R. Trott, and Daniel Sunderland. 2014. Kokkos: Enabling manycore performance portability through polymorphic memory access patterns. *J. Parallel and Distrib. Comput.* 74, 12 (2014), 3202–3216. doi:10.1016/j.jpdc.2014.07.003 Domain-Specific Languages and High-Level Frameworks for High-Performance Computing.

[15] Manuel Chakravarty, Gabriele Keller, Roman Leshchinskiy, and Wolf Pfannenstiel. 2001. Nepal—Nested Data-Parallelism in Haskell. In *Proceedings of the 7th International Euro-Par Conference (EUROPAR).*

[16] Manuel Chakravarty, Roman Leshchinskiy, Simon Peyon Jones, Gabriele Keller, and Simon Marlow. 2007. Data Parallel Haskell: A Status Report. In *Proceedings of the Workshop on Declarative Aspects of Multicore Programming.*

[17] Bradford Chamberlain, David Callahan, and Hans Zima. 2007. Parallel Programmability and the Chapel Langauge. *International Journal of High Performance Computing Applications* 21, 3 (August 2007), 291–312.

[18] Henry Cook, Wesley Terpstra, and Yunsup Lee. 2017. Diplomatic Design Patterns: A TileLink Case Study. (2017).

[19] Florent de Dinechin and Bogdan Pasca. 2011. Designing Custom Arithmetic Data Paths with FloPoCo. *IEEE Design & Test of Computers* 28, 4 (July 2011), 18–27.

[20] Matthew Fluet, Nic Ford, Mike Rainey, John Reppy, Adam Shaw, and Yingqi Xiao. 2007. Status Report: The Manticore Project. In *Proceedings of the 2007 ACM SIGPLAN Workshop on ML.*

[21] Matthew Fluet, Mike Rainey, John Reppy, and Adam Shaw. 2008. Implicitly Threaded Parallelism in Manticore. In *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming (ICFP).*

[22] Matthew Fluet, Mike Rainey, John Reppy, Adam Shaw, and Yingqi Xiao. 2007. Manticore: A Heterogeneous Parallel Language. In *Proceedings of the Workshop on Declarative Aspects of Multicore Programming.*

[23] Bryan Ford. [n. d.]. Parsing Expression Grammars: A Recognition-Based Syntactic Foundation. ([n. d.]).

[24] M. Frigo and S.G. Johnson. 2005. The Design and Implementation of FFTW3. 93, 2 (2005), 216–231. doi:10.1109/JPROC.2004.840301

[25] Adrien Guatto, Sam Westrick, Ram Raghunathan, Umut Acar, and Matthew Fluet. 2018. Hierarchical memory management for mutable state. *ACM SIGPLAN Notices* 53, 1 (2018), 81–93.

[26] Juan Gómez-Luna, Izzat El Hajj, Ivan Fernandez, Christina Giannoula, Geraldo F. Oliveira, and Onur Mutlu. 2022. Benchmarking a New Paradigm: Experimental Analysis and Characterization of a Real Processing-in-Memory System. *IEEE Access* 10 (2022).

[27] John L. Hennessy and David A. Patterson. 2019. A New Golden Age for Computer Architecture. *Commununcations of the ACM* 62, 2 (February 2019), 48–60. Turing Award Lecture.

[28] Troels Henriksen, Niels G. W. Serup, Martin Elsman, Fritz Henglein, and Cosmin E. Oancea. 2017. Futhark: Purely Functional GPU-programming with

Nested Parallelism and In-place Array Updates. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Barcelona, Spain) *(PLDI 2017)*. ACM, New York, NY, USA, 556–571. doi:10.1145/3062341.3062354

[29] Troels Henriksen, Frederik Thorøe, Martin Elsman, and Cosmin Oancea. 2019. Incremental flattening for nested data parallelism. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming* (Washington, District of Columbia) *(PPoPP '19)*. Association for Computing Machinery, New York, NY, USA, 53–67. doi:10.1145/3293883.3295707

[30] H. Jin, M. Frumkin, and J. Yan. 1999. *The Open MP Implementation of NAS Parallel Benchmarks and Its Performance (NAS 3)*. Technical Report NAS-99-011. NASA.

[31] Jungwon Kim, Seyong Lee, Beau Johnston, and Jeffrey S. Vetter. 2021. IRIS: A Portable Runtime System Exploiting Multiple Heterogeneous Programming Systems. In *2021 IEEE High Performance Extreme Computing Conference (HPEC)*. 1–8. doi:10.1109/HPEC49654.2021.9622873

[32] B. Kocoloski and J. Lange. 2014. HPMMAP: Lightweight Memory Management for Commodity Operating Systems. In *Proceedings of the 28th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*.

[33] Mario Kovač, Dominik Reinhardt, Oliver Jesorsky, Matthias Traub, Jean-Marc Denis, and Philippe Notton. 2019. European Processor Initiative (EPI)—An Approach for a Future Automotive eHPC Semiconductor Platform. In *Electronic Components and Systems for Automotive Applications* (Cham, 2019) *(Lecture Notes in Mobility)*, Jochen Langheim (Ed.). Springer International Publishing, 185–195. doi:10.1007/978-3-030-14156-1_15

[34] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the International Symposium on Code Generation and Optimization* (Palo Alto, California) *(CGO '04)*. IEEE Computer Society.

[35] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. 2021. MLIR: Scaling Compiler Infrastructure for Domain Specific Computation. In *Proceedings of the IEEE/ACM International Symposium on Code Generation and Optimization* (Virtual Event, Republic of Korea) *(CGO '21)*. IEEE Press, 2–14. doi:10.1109/CGO51591.2021.9370308

[36] Sukhan Lee, Shin-haeng Kang, Jaehoon Lee, Hyeonsu Kim, Eojin Lee, Seungwoo Seo, Hosang Yoon, Seungwon Lee, Kyounghwan Lim, Hyunsung Shin, Jinhyun Kim, O Seongil, Anand Iyer, David Wang, Kyomin Sohn, and Nam Sung Kim. 2021. Hardware Architecture and Software Stack for PIM Based on Commercial DRAM Technology : Industrial Product. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. 43–56. doi:10.1109/ISCA52012.2021.00013

[37] David B Loveman. 1993. High performance fortran. *IEEE Parallel & Distributed Technology: Systems & Applications* 1, 1 (1993), 25–42.

[38] MPL [n. d.]. MPL compiler. https://github.com/mpllang/mpl.

[39] Robert W Numrich and John Reid. 1998. Co-Array Fortran for parallel programming. In *ACM Sigplan Fortran Forum*, Vol. 17. ACM New York, NY, USA, 1–31.

[40] Karl J. Ottenstein, Robert A. Ballance, and Arthur B. MacCabe. 1990. The program dependence web: a representation supporting control-, data-, and demand-driven interpretation of imperative languages. In *Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation* (White Plains, New York, USA) *(PLDI '90)*. Association for Computing Machinery, New York, NY, USA, 257–271. doi:10.1145/93542.93578

[41] Ruyman Reyes, Gordon Brown, Rod Burns, and Michael Wong. 2020. SYCL 2020: More than meets the eye. In *Proceedings of the International Workshop on OpenCL* (Munich, Germany) *(IWOCL '20)*. Association for Computing Machinery, New York, NY, USA, Article 4, 1 pages. doi:10.1145/3388333.3388649

[42] Robert B. Ross, George Amvrosiadis, Philip Carns, Charles D. Cranor, Matthieu Dorier, Kevin Harms, Greg Ganger, Garth Gibson, Samuel K. Gutierrez, Robert Latham, Bob Robey, Dana Robinson, Bradley Settlemyer, Galen Shipman, Shane Snyder, Jerome Soumagne, and Qing Zheng. 2020. Mochi: Composing Data Services for High-Performance Computing Environments. 35, 1 (2020), 121–144. doi:10.1007/s11390-020-9802-0

[43] Julian Shun, Guy E. Blelloch, Jeremy T. Fineman, Phillip B. Gibbons, Aapo Kyrola, Harsha Vardhan Simhadri, and Kanat Tangwongsan. 2012. Brief announcement: the problem based benchmark suite. In *Proceedings of the twenty-fourth annual ACM symposium on Parallelism in algorithms and architectures* (Pittsburgh Pennsylvania USA, 2012-06-25). ACM, 68–70. doi:10.1145/2312005.2312018

[44] Michel Steuwer, Christian Fensch, Sam Lindley, and Christophe Dubach. 2015. Generating performance portable code using rewrite rules: from high-level functional expressions to high-performance OpenCL code. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming* (New York, NY, USA, 2015-08-29) *(ICFP 2015)*. Association for Computing Machinery, 205–217. doi:10.1145/2784731.2784754

[45] Andrew Waterman and Krste Asanović (Eds.). 2016. *The RISC-V Instruction Set Manual, Volume I: User-Level ISA*. RISC-V Foundation. http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-17.html

[46] Andrew Waterman and Krste Asanović (Eds.). 2024. *The RISC-V Instruction Set Manual, Volume I: User-Level ISA*. RISC-V Foundation. https://drive.google.com/file/d/1uviu1nH-tScFfgrovvFCrj7Omv8tFtkp/view?usp=drive_link

[47] Sam Westrick, Rohan Yadav, Matthew Fluet, and Umut A. Acar. 2020. Disentanglement in Nested-Parallel Programs. In *Proceedings of the 47th Annual ACM Symposium on Principles of Programming Languages (POPL)"*.

[48] Michael Wilkins. 2024. *On Transparent Optimizations for Communication in Highly Parallel Systems*. Ph. D. Dissertation. Department of Computer Science, Northwestern University. Available as Technical Report NU-CS-2024-01, Department of Computer Science, Northwestern University.

[49] Michael Wilkins, Garrett Weil, Luke Arnold, Nikos Hardavellas, and Peter Dinda. 2023. Evaluating Functional Memory-Managed Parallel Languages for HPC using the NAS Parallel Benchmarks. In *Proceedings of the 28th International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS 2023); 2023 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*.

[50] Mike Wilkins, Sam Westrick, Vijay Kandiah, Alex Bernat, Brian Suchy, Enrico Deiana, Simone Campanoni, Umut Acar, Peter Dinda, and Nikos Hardavellas. 2023. WARDen: Specializing Cache Coherence for High-Level Parallel Languages. In *Proceedings of the 21st IEEE/ACM International Symposium on Code Generation and Optimization (CGO 2023)*.

[51] Yili Zheng, Amir Kamil, Michael B Driscoll, Hongzhang Shan, and Katherine Yelick. 2014. UPC++: a PGAS extension for C++. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium*. IEEE, 1105–1114.