Automatic SIMD Vectorization for Haskell

Leaf Petersen
Intel Labs
leaf.petersen@intel.com

Dominic Orchard
Computer Laboratory
University of Cambridge
dominic.orchard@cl.cam.ac.uk

Neal Glew
Intel Labs
aglew@acm.org

Abstract

Expressing algorithms using immutable arrays greatly simplifies the challenges of automatic SIMD vectorization, since several important classes of dependency violations cannot occur. The Haskell programming language provides libraries for programming with immutable arrays, and compiler support for optimizing them to eliminate the overhead of intermediate temporary arrays. We describe an implementation of automatic SIMD vectorization in a Haskell compiler which gives significant vector speedups for a range of programs written in a natural programming style. We compare performance with that of programs compiled by the Glasgow Haskell Compiler.

1. Introduction

In the past decade, power has increasingly been recognized as the key limiting resource in micro-processors, in part due to the limited availability of battery on mobile devices, but more fundamentally due to the non-linear scaling of power usage with frequency. This has forced micro-processor designs to explore (or re-explore) different avenues at both the micro-architectural and the architectural levels. One successful architectural trend has been the increasing emphasis on data parallelism in the form of single-instruction multiple-data (SIMD) instruction sets.

The key idea of data parallelism is that significant portions of many sequential computations consist of uniform (or almost uniform) operations over large collections of data. SIMD instruction sets exploit this at the machine instruction level by providing data paths for and operations on fixed-width vectors of data. Programmers (or preferably compilers) are then responsible for finding data parallel computations and expressing them in terms of fixed-width vectors. In the context of languages with loops (or compiled to loops), this corresponds to choosing loops for which multiple iterations can be computed simultaneously.

The task of automatically finding SIMD vectorizable code in the context of a compiler has been the subject of extensive study over the course of decades. For common imperative programming languages, such as C and Fortran, in which programs are structured as loops over mutable arrays of data, the task of rewriting loops in SIMD vector form is straightforward. However, side-effects and dependencies within a loop body may result in a rewriting changing the program semantics. Furthermore, the task of discovering which loops permit a valid rewriting (via dependency analysis) is in general undecidable, and has been the subject of extensive research (see e.g. [12] and overview in [14]). Even quite sophisticated compilers often are forced to rely on programmer annotations to *assert* that a given loop is a valid target of SIMD vectorization (for example, the `INDEPENDENT` keyword in HPF [7]).

A great deal of the difficulty in finding vectorizable loops simply goes away in the context of a functional language. In functional languages, the absence of mutation eliminates large classes of the more difficult kinds of dependence violations that block vectorization in compilers for imperative languages. In this sense, SIMD vectorization would seem to be quite low-hanging fruit for functional language compiler writers. In this paper, we describe the implementation of an automatic SIMD vectorization optimization pass in the Intel Functional Language Compiler (our compiler). We argue that in a functional language context, this optimization is straightforward to implement and frequently gives excellent results.

Our compiler was originally developed to compile programs written in an experimental strict functional language, and the vectorization optimization that we present here was developed in that context. However, the main bulk of our compiler infrastructure was designed to be fairly language agnostic, and we have more recently used it to build an experimental Haskell compiler using the Glasgow Haskell Compiler (GHC) as a front end. While Haskell is quite a complex language on the surface, the GHC front end reduces it to a relatively simple intermediate representation based on System F, which is fairly straightforward to compile using our compiler. By intercepting the GHC intermediate representation (IR) after the main high-level optimization passes have run, we gain the benefit of the substantial and sophisticated optimization infrastructure built up in GHC, and in particular we benefit from its ability to optimize away temporary intermediate arrays.

We make the following three contributions:

- We design a core language with arrays and SIMD vector primitives (Section 3).
- We outline a novel vectorization process over this language (Section 4).
- We evaluate our approach compared to GHC with the native backend and LLVM backend, showing speedups of between 2x and 10x using our compiler, for a selection of array processing benchmarks (Section 5).

We begin with a brief overview of our compiler and its relationship to GHC.

2. The Intel Functional Language Compiler

2.1 Structure of our compiler

The Intel Functional Language Compiler provides an optimization platform aimed at producing high-performance executables from functional-style code. The core intermediate language of the

[Copyright notice will appear here once 'preprint' option is removed.]

compiler is a static single-assignment style, control-flow graph based intermediate representation (IR). It is strict and has explicit thunks to represent laziness. While the intermediate representation is largely language agnostic, it is not paradigm agnostic. The core technologies of the compiler are designed specifically around compiling functional-language programs. Most importantly for the purposes of this paper, the compiler relies on maintaining a distinction between mutable and immutable data, and focuses almost all of its optimization effort on the latter. The compiler is structured as a whole-program compiler—however, none of the optimizations in the compiler rely on this property in any essential way. In addition to the usual inlining-style optimizations, contification [5] is performed to turn many uses of (mutual) recursion into loops. This optimization is crucial for our purposes since we structure our SIMD-vectorization optimization as an optimization on loops. Many standard compiler optimizations have been implemented in the compiler, including loop-invariant code motion and a very general simplifier in the style of Appel and Jim [2]. The compiler also implements a number of inter-procedural representation optimizations using a field-sensitive flow analysis [10].

The compiler generates output in a modified extension of the C language called *Pillar* [1]. Among other things, the Pillar language supports tail calls and second-class continuations. Most importantly, it also provides the necessary mechanisms to allow for accurate garbage collection of managed memory. While an early version of Pillar was implemented as a modification of a C compiler, Pillar is currently implemented as a source to source translation targetting standard C, which is then compiled using either the Intel C Compiler or GCC. Our garbage collector and a small runtime are linked in to produce the final executable.

2.2 Haskell and GHC

The Haskell language and the GHC extensions to it provide a large base of useful high-level libraries written in a very powerful programming language. These libraries are often written using a high-level functional-language style of programming, relying on excellent compiler technology to achieve good performance. While we wished to be able to compile Haskell programs and take advantage of the many libraries, we had no interest in attempting the monumental task of duplicating all of the high-level compiler technology implemented in GHC. The approach we took therefore was to essentially use GHC as a Haskell frontend, thereby taking advantage of the existing type-checker and high-level optimizer.

Fortunately for our purposes, GHC provides some support for writing its main System-F-style internal representation (called Core) out to files. While this code is not entirely maintained by the GHC implementers, we were able to easily modify it to support all the current Core features we require. We use this facility to build a Haskell compiler pipeline by first running GHC to perform type checking, desugaring, and all of its Core-level optimizations, and then writing out the Core IR immediately before it would otherwise be translated to the next-lower representation. Our compiler then reads in the serialized Core, and after several initial transformations and optimizations (including making thunks explicit), performs a closure-conversion pass and transforms it down to our main CFG based intermediate representation for subsequent optimization.

2.3 GHC Modifications

Using this approach, we are able to provide a fairly close to functionally complete implementation of the Haskell language as implemented by GHC. However, some of the choices of primitives used in GHC do not match up well with the needs of our compiler, and so some modification of GHC itself was required. A notable example of this for the purposes of this paper is the treatment of arrays in the Data.Vector libraries.

The GHC libraries Data.Vector and REPA [6] provide clean high-level interfaces for generating immutable arrays. Programmers can use typical functional-language operations like maps, folds, and zips to compute arrays from other arrays. These operations are naturally data parallel and amenable to SIMD vectorization by our compiler. For example, consider a map operation over an immutable array represented with the Data.Vector library. Such arrays are represented internally to the libraries using streams, and so map first turns the source array into a stream, applies the map operation to the stream, and then unstreams that back into the final array. Unstreaming is implemented by running a monadic computation in the state monad that first creates a mutable array of the appropriate size, then successively initializes the elements of the array using array update, and finally does an unsafe freeze of the mutable array to an immutable array type. GHC has aggressive inlining and many simplification operations, which together eliminate all the intermediate streaming, mapping, and unstreaming operations. The result of all of this optimization is a piece of code that sequences the array creation, the calling of a tail-recursive function that initializes the array elements, and the unsafe freeze. After translation into our internal representation, our compiler can then contify the tail-recursive function into a loop, resulting in a natural piece of code that can be very effectively optimized using relatively standard loop based optimizations adapted to leverage the benefits of immutability.

Unfortunately, a careful reader will observe that as described in the previous paragraph we do not actually get the benefits of immutability with an unmodified GHC. Instead, we end up with code that creates a mutable array and then initializes it with general array update operations—a style of code that we argue is drastically harder to optimize well in general, and to perform SIMD vectorization on in particular. As we discuss in Section 4.7, immutable arrays make determining when a loop can be SIMD vectorized much easier. For our purposes then, what we would like GHC to generate instead of the code described above is code that creates a new uninitialized immutable array and then uses *initializing writes* to initialize the array. The two invariants of initializing writes are that reading an array element must always follow the initializing write of that element, and that any element can only be initialized once. This style of write preserves the benefits of immutability from the compiler standpoint, since any initializing write to an element is the unique definition of that element. Initializing writes are key to being able to represent the initialization of an immutable array using the usual loop code, while preserving the ability to leverage immutability.

In order to get immutable code written in this style from GHC and the libraries discussed above, we modified GHC with new primitive immutable array types[1] and primitive operations on them. The additional operations include creation of a new uninitialized array, initializing writes, and a subscript operation. We also modified the Data.Vector library and a small part of the REPA library to use these immutable arrays instead of mutable arrays and unsafe freeze (of course, as with the standard GHC primitives, correctness requires that the library writer use the primitives correctly). With these modifications, the code produced by GHC is suitable for optimization by our compiler, including in many cases SIMD vectorization.

2.4 Vectorization in our compiler

SIMD vectorization in our compiler targets loops for which the compiler can show that executing several iterations in parallel is semantics preserving, rewriting them to use SIMD vector instruc-

[1] GHC does already have some such types—however, we added new ones to avoid compatibility issues

$$
\begin{array}{llll}
\text{Register kind} & k & ::= & s \mid v \\
\text{Variables} & x^k, y^k, z^k, \ldots & & \\
\text{Constants} & c & ::= & -2^{31}, \ldots, (2^{31}-1) \\
\text{Operations} & op & ::= & +, -, *, /, \ldots \\
\text{Instruction} & I & ::= & z^s = c \\
 & & \mid & z^v = \langle x^s_0, \ldots, x^s_7 \rangle \\
 & & \mid & z^s = x^v!i \\
 & & \mid & z^s = op(x^s_0, \ldots, x^s_n) \\
 & & \mid & z^v = \langle op \rangle (x^v_0, \ldots, x^v_n) \\
 & & \mid & z^s = \mathtt{new}[x^s] \\
 & & \mid & z^s = x^s[y^s] \\
 & & \mid & z^v = x^s[\langle y^v \rangle] \\
 & & \mid & z^v = \langle x^v \rangle [y^s] \\
 & & \mid & z^v = \langle x^v \rangle [\langle y^v \rangle] \\
 & & \mid & x^s[y^s] \leftarrow z^s \\
 & & \mid & x^s[\langle y^v \rangle] \leftarrow z^v \\
 & & \mid & \langle x^v \rangle [y^s] \leftarrow z^v \\
 & & \mid & \langle x^v \rangle [\langle y^v \rangle] \leftarrow z^v \\
\text{Comparisons} & cmp & ::= & x^s < y^s \mid x^s \leq y^s \mid x^s = y^s \\
\text{Labels} & L & ::= & \mathtt{L}^0, \mathtt{L}^1, \ldots \\
\text{Transfers} & t & ::= & \mathtt{goto}\, L(x^k_0, \ldots, x^k_n) \\
 & & \mid & \mathtt{if}\ (cmp)\ \mathtt{goto}\, L_0(x^k_0, \ldots, x^k_n) \\
 & & & \quad \mathtt{else}\ \mathtt{goto}\, L_1(y^k_0, \ldots, y^k_m) \\
 & & \mid & \mathtt{halt} \\
\text{Blocks} & B & ::= & L(x^k_0, \ldots, x^k_n){:} \\
 & & & \quad I_0 \\
 & & & \quad \ldots \\
 & & & \quad I_m \\
 & & & \quad t \\
\text{Control Flow Graph} & & ::= & \mathtt{Entry}\ L\ \mathtt{in}\ \{B_0, \ldots, B_n\}
\end{array}
$$

Figure 1. Syntax

tions. Usually these loops serve to initialize immutable arrays, or to perform reductions over immutable arrays. The compiler generates the SIMD loops and all of the related control logic simply as code in our intermediate language. Rather than requiring the compiler to generate only the SIMD instructions provided by the target machine, we provide a more uniform SIMD instruction set that may be a superset of the machine-supported instructions. Our small runtime provides a library of impedance-matching macros that implements the uniform SIMD instructions in terms of SIMD intrisics understood by the underlying C compiler. The C compiler is then responsible for low-level optimizations including register allocation and instruction selection.

In the next section we define a vector core language with which to discuss the vectorization process. This core language is clean, uniform, and is in fact almost exactly faithful to (a small subset of) our actual intermediate representation.

3. Vector Core language

We define here a small core language in which to present the vectorization optimization. We restrict ourselves to an intra-procedural fragment only, since for our purposes here we are not interested in inter-procedural control flow. Programs in the language are given as control-flow graphs written in a variant of static single-assignment (SSA) form. To keep things concrete, our language is based on a 32-bit machine with vector registers that are 256-bits wide; handling other register and vector widths is a straightforward extension.

3.1 Objects

The primitive objects of the vector core language consist of heap objects and register values. Heap objects are variable-length immutable arrays subscripted by 32-bit integers. The elements contained in the fields of heap objects are constrained to be 32-bit register values. Register values are small values that may be bound to variables and are implicitly represented at the machine level via registers or spill slots. The register values consist of 32-bit integers, 32-bit pointers to heap values, and 256-bit SIMD vectors each consisting of eight 32-bit register values. Adding additional primitive register values such as floating-pointer numbers, booleans, etc. is straightforward and adds nothing essential to the presentation.

3.2 Variables

Variables are divided into two classes according to size: scalar variables x^s that may only be bound to 32-bit register values and vector variables x^v that may only be bound to 256-bit vector values. In practice of course, a real intermediate language will also have additional variable sizes: 64-bit variables for doubles and 64-bit integers, and a range of vector widths possibly including 128- and 512-bit vectors. Note that the variables x^s and x^v are considered to be distinct variables. Informally, we will sometimes use x^v to denote the variable containing a vectorized version of an original program variable x^s—however, this is intended to be suggestive only and has no semantic significance.

3.3 Instructions

The instructions of the vector language serve to introduce and operate on the primitive objects of the language. Most of the instructions bind a new variable of the appropriate register kind, with the exception of the array-initialization instructions which write values into uninitialized array elements. The move instruction $z^s = c$ binds a variable to the 32-bit integer constant c. The vector introduction instruction $z^v = \langle x^s_0, \ldots, x^s_7 \rangle$ binds the vector variable z^v to a vector composed of the contents of the eight listed 32-bit variables. We use the derived syntax $z^v = \langle c^s_0, \ldots, c^s_7 \rangle$ to introduce a vector of constants—this may be derived in the obvious way by first binding each of the constants to a fresh 32-bit variable. A component of a vector may be selected out using the select instruction $z^s = x^v!i$, which binds z^s to the ith component of the vector in x^v (where i must be in the range $0 \ldots 7$).

The instruction $z^s = op(x^s_0, \ldots, x^s_n)$ corresponds to a family of instructions operating on primitive 32-bit integers, indexed by the operation *op* (e.g. addition, subtraction, negation etc.). Primitive operations must be fully saturated—that is, the number of operands provided must equal the arity of the given operation. We lift the primitive operations on integers to pointwise primitive operations on vectors with the $z^v = \langle op \rangle (x^v_0, \ldots, x^v_n)$ instruction, which applies *op* pointwise across the vectors. That is, semantically $z^v = \langle op \rangle (x^v_0, \ldots, x^v_n)$ behaves identically to a sequence of instructions performing the scalar version of *op* to each of the corresponding elements of the argument vectors, and then packaging up the result as the final result vector.

Array objects in the vector language are immutable sequences of 32-bit data allocated in the heap. New uninitialized array objects are allocated via the $z^s = \mathtt{new}[x^s]$ instruction which allocates a new array of length x^s (where x^s must contain a 32-bit integer) and binds z^s to a pointer to the array. The initial contents of the array are undefined, and every element must be initialized before being read. It is an unchecked invariant of the language that every element of the array is initialized before it is read, and that every element of the array is initialized at most once. It is in this sense that the arrays are immutable—the write operation on arrays serves only as an initializing write but does not provide for mutation. Ordinary scalar element initialization is performed via the $x^s[y^s] \leftarrow z^s$

operation which initializes the element of array x^s at offset y^s to z^s. Scalar reads from the array are performed using the $z^s = x^s[y^s]$ instruction, which reads a single element from the array x^s at offset y^s and binds z^s to it.

In order to support SIMD vectorization, the vector language also provides a general set of vector subscript and initialization operations permitting vectors of elements to be read from and written to arrays (or vectors of arrays) in a single instruction. The set of instructions we provide are more general than what is supported by most existing machine architectures. This is done intentionally since it is often beneficial to vectorize a loop even if some instructions must be emulated via reduction to scalar operations. Morever, we believe it is useful to present the language in its most general form—it is always possible to restrict the set of generated instructions as desired. We defer discussion of the situations in which these various styles of loads and stores arise to subsequent sections on the actual vectorization operation.

The simplest vector array loads and stores are the operations that read multiple elements directly from a single array into a vector variable, or initialize multiple elements of a single array with elements contained in a vector variable. The instruction $z^v = x^s[\langle y^v \rangle]$ takes a single array x^s and a vector of offsets y^v and binds z^v to a vector containing the elements of x^s from the given offsets. That is, semantically we may treat the instruction $z^v = x^s[\langle y^v \rangle]$ as equivalent to the following list of instructions:

$$\begin{array}{ll} y_0^s & = y^s!0 \\ z_0^s & = x^s[y_0^s] \\ \ldots & \\ y_7^s & = y^s!7 \\ z_7^s & = x^s[y_7^s] \\ z^v & = \langle z_0^s, \ldots, z_7^s \rangle \end{array}$$

This instruction is commonly known as a "gather" instruction. The intializing store $x^s[\langle y^v \rangle] \leftarrow z^v$, commonly known as the "scatter" instruction, writes the elements of the vector z^v to the fields of x^s at offsets given by the vector y^v.

There are several interesting special cases of scatters and gathers that are worth further discussion. A common idiom that arises in SIMD vectorization consists of array loads and stores such that the index vector is constructed by adding multiples of a constant stride to a scalar index. We choose to introduce this idiom as derived syntax as follows:

$$\begin{array}{l} z^v = x^s[\langle y^s{:}i \rangle] \stackrel{\text{def}}{=} z^v = x^s[\langle y^v \rangle] \\ x^s[\langle y^s{:}i \rangle] \leftarrow z^v \stackrel{\text{def}}{=} x^s[\langle y^v \rangle] \leftarrow z^v \\ \text{where} \quad \left\{ \begin{array}{l} y_l^v = \langle y^s, \ldots, y^s \rangle \\ y_b^v = \langle 0, \ldots, 7 * i \rangle \\ y^v = \langle + \rangle (y_l^v, y_b^v) \end{array} \right. \end{array}$$

We also provide derived syntax for the further special case in which the stride is known to be one:

$$\begin{array}{l} z^v = x^s[\langle y^s{:}\rangle] \stackrel{\text{def}}{=} z^v = x^s[\langle y^s{:}1 \rangle] \\ x^s[\langle y^s{:}\rangle] \leftarrow z^v \stackrel{\text{def}}{=} x^s[\langle y^s{:}1 \rangle] \leftarrow z^v \end{array}$$

For the purposes of simplicity of the vector language, we have left these as derived forms. However, it is worth noting that many architectures provide support for these idioms directly (and in fact may provide support *only* for these idioms) and hence from a pragmatic standpoint it can be important to provide primitive support for these addressing modes.

A second mode of addressing for array loads and stores covers the case where a vector of arrays is indexed pointwise by a single fixed scalar offset. The $z^v = \langle x^v \rangle [y^s]$ instruction produces a vector z^v such that the ith element of z^v is produced by indexing the ith array from the vector of arrays x^v, using offset y^s. Similarly, the $\langle x^v \rangle [y^s] \leftarrow z^v$ instruction sets the element of the ith array in the vector x^v at offset y^s to the value in the ith position of z^v.

The final mode of addressing for array loads and stores covers the case in which a vector of arrays is indexed pointwise using a vector of offsets. The $z^v = \langle x^v \rangle [\langle y^v \rangle]$ instruction produces a vector z^v such that the ith element is produced by indexing the ith array from the vector of arrays x^v using the ith offset from the vector of offsets y^v. Similarly, the $\langle x^v \rangle [\langle y^v \rangle] \leftarrow z^v$ instruction sets the element of the ith array in the vector x^v at the offset given by the ith element of the vector of offsets y^v to the value in the ith position of z^v.

3.4 Basic Blocks

Instructions are grouped into basic blocks consisting of a labeled entry, a sequence of instructions, and a transfer which terminates the block and either transfers control to another block or terminates the program. We assume an infinite supply of distinct program labels ranged over by metavariable *L* used to label basic blocks that we distinguish by numbering distinct labels with distinct integers (as $\mathtt{L}^0$, $\mathtt{L}^1$, etc.).

Basic block headers consist of the label of the block and a list of entry variables which are defined on entry to the block. The entry variables are given their defining values by the transfer which transfers control to the block. Block entry variables may have scalar or vector kind.

Transfers terminate basic blocks, and serve to transfer control within a program. The goto $L(x_0^k, \ldots, x_n^k)$ transfer shifts control the block labeled with *L*. Well-formed programs must have matching arities on transfers and the block headers which they target, that is, the block labeled with *L* must have entry variables $z_0^k, \ldots, z_n^k$. The transfer of control effected by goto $L(x_0^k, \ldots, x_n^k)$ also defines the entry variables $z_0^k, \ldots, z_n^k$ to be the values of $x_0^k, \ldots, x_n^k$.

The conditional control transfer

$$\texttt{if } (\mathit{cmp}) \texttt{ goto } L_0(x_0^k, \ldots, x_n^k) \texttt{ else goto } L_1(y_0^k, \ldots, y_m^k)$$

behaves as goto $L_0(x_0^k, \ldots, x_n^k)$ if the comparison holds, and as goto $L_1(y_0^k, \ldots, y_m^k)$ if the comparison does not hold. Note that the targets of the two different arms of the conditional are not required to be distinct.

The halt transfer terminates the program. We do not consider inter-procedural control-flow in this core langauge since it is not relevant to the vectorization optimization—however it is straightforward to add additional transfers to account for this if desired.

3.5 Programs

Programs in the core language consist of a single control flow graph. A control flow graph consists of a designated entry label *L* and set of associated basic blocks, each with a distinct label, one of which (the entry block) is labeled with the entry label *L*. The entry block must expect no entry variables. Program execution proceeds by starting at the designated entry block and executing block instructions and transfers until a halt instruction is reached (if ever). Well-formed programs must not contain transfers whose target label is not in the control-flow graph, and every block in the control-flow graph must be labeled with a distinct label. As usual with static single-assignment form, variables may only be defined once, must be defined before their first use, and have scope given by the dominator tree of the control-flow graph rooted at the entry label. The style of static single-assignment form used in this core language is similar to that used in compilers such as the MLton compiler [15].

4. Vectorization

The SIMD-vectorization optimization attempts to rewrite a loop to a new loop that executes multiple iterations of the original loop on each iteration using SIMD instructions and registers. While the core of the idea is quite simple, there are a number of issues that must be accounted for in order to preserve the semantics of the original program. In the first part of this section we introduce the key issues using a series of examples. Before doing so however, we first quickly review some preliminary concepts.

4.1 Loops

We have informally described our vectorization efforts as focusing on loops. Loops tend to account for a high proportion of executed instructions in programs, and as such are natural targets for introducing SIMD-vector code. We define loops using the standard notion of a natural loop, merging loops with shared headers in the usual way to ensure that the nesting structure of loops form a tree. We do not assume a reducible control-flow graph (since in general the translation of mutually-recursive functions into loops may introduce irreducible control flow). We focus our vectorization efforts more specifically on the innermost loops that make up the leaves of the loop forest of the control-flow graph. For simplicity, we only target bottom-test loops. A separate optimization pass inverts top-test loops to form bottom-test loops, both to enable vectorization and loop-invariant code motion.

4.2 Induction variables

Our notion of induction variable is standard, but we review it in some detail here since induction variables play a critical role in our algorithm. We define the *base* induction variables of a loop to be the entry variables of the entry block of the loop such that the definition of that variable provided on the loop back edge is produced by adding a constant to the variable itself. The *step* of a base induction variable is the constant added each time around the loop. The full set of induction variables is the least set of variables satisfying:

- A base induction variable is an induction variable.
- A variable defined by $x^s = +(y^s, z^s)$, where y^s is an induction variable and z^s is loop invariant, is an induction variable.
- A variable defined by $x^s = *(y^s, z^s)$, where y^s is an induction variable and z^s is a constant (defined by $z^s = c$), is an induction variable.

A full implementation must also deal with the symmetric cases of the last two cases, or canonicalize programs into the above forms, and may also consider operations such as subtraction and negation.

Induction variables can be characterized as affine functions of the iteration count of the loop. We define the characteristic function of an induction variable i^s as $i^s = s * \# + d$ where s is a constant (the step of the induction variable), and d is also a constant (the initial value of the induction variable). The symbol $\#$ stands for the iteration number: the value of the induction variable i^s for iteration j can be computed directly from its characteristic function simply by replacing $\#$ with j. If the characteristic function of an induction variable is $s * \# + d$ then the step of the induction variable is s.

The characteristic function of an induction variable is defined as follows:

- The characteristic function of a base induction variable is $s * \# + d$ where s is the step of the base induction variable as defined above, and d is the constant initial value of the induction variable.
- The characteristic function of an induction variable defined by $x^s = +(y^s, z^s)$ is $s*\#+d+c$ where z^s is a constant (defined by $z^s = c$), and $s * \# + d$ is the characteristic function of y^s.
- The characteristic function of an induction variable defined by $x^s = *(y^s, z^s)$ is $c*s*\#+c*d$ where z^s is a constant (defined by $z^s = c$), and $s * \# + d$ is the characteristic function of y^s.

A subtle but important point is that in these computations the compiler must ensure that the overflow semantics of the underlying numeric type are respected to avoid introducing or eliminating numeric overflow. It is possible to extend the definition of characteristic functions to allow loop invariants to take the place of the constants in the above definition, at the expense of representing the characteristic function symbolically.

4.3 Vectorization by example

Consider the following simple program which computes the pointwise sum of two arrays b^s and c^s, each of length l^s.

$$
\begin{array}{ll}
\mathtt{L}^0(): & \mathtt{L}^1(i^s): \\
\quad a^s = \mathtt{new}[l^s] & \quad x^s = b^s[i^s] \\
\quad i^s_0 = 0 & \quad y^s = c^s[i^s] \\
\quad \mathtt{goto}\ \mathtt{L}^1(i^s_0) & \quad z^s = +(x^s, y^s) \\
 & \quad a^s[i^s] \leftarrow z^s \\
 & \quad i^s_1 = +(i^s, 1) \\
 & \quad \mathtt{if}\ (i^s_1 < l^s)\ \mathtt{goto}\ \mathtt{L}^1(i^s_1) \\
 & \qquad\qquad \mathtt{else}\ \mathtt{goto}\ \mathtt{L}^{\mathrm{exit}}(i^s_1)
\end{array} \quad (1)
$$

Ignoring the crucial issue of whether or not it is in fact *valid* to vectorize this loop, there are a number of issues that need to be addressed in order to produce a vector version of this program.

4.3.1 The vector loop

The core of the vectorization optimization is to produce an inner loop each iteration of which executes multiple iterations of the original loop. For Example 1, this corresponds to generating code to vector-load eight elements of the arrays b and c at appropriate offsets, perform a pointwise vector addition of the loaded elements, and to perform a vector write of the eight result elements into the new array a. The comparison which controls the exit test at the bottom of the loop must also be adjusted appropriately to account for the fact that multiple iterations are being performed. Finally, on the exit edge of the loop, the use of the last value of the induction variable i^s_1 must also be accounted for.

To motivate the general treatment of the vectorization optimization, we first consider the individual instructions of Example 1, beginning with the load instruction $x^s = b^s[i^s]$. Executing this instruction during iteration j of the loop loads a single element from the array b^s at an offset given by the value of i^s. In the vector loop, we wish perform iterations $j, j+1, \ldots, j+7$ simultaneously. The vector version of this instruction then must load eight elements from b^s at offsets given by the values of i^s at iteration $j, j+1, \ldots, j+7$. A natural first attempt at vectorization might be to simply assume that vector versions of all variables are available, and to then generate the corresponding vector instruction using the vector variables. In this case, if we assume the existence of variables b^v and i^v containing the values of b^s and i^s for iterations $j, \ldots, j+7$ then the vector instruction $x^v = \langle b^v \rangle [\langle i^v \rangle]$ puts into x^v the appropriate values of x^s for iterations $j, \ldots, j+7$.

While this approach is correct (and in the most general case is sometimes necessary) a key insight in SIMD vectorization is that it is often possible to do much better by using knowledge of loop invariants and induction variables. In this case, the variable b^s is loop invariant, and consequently b^v will consist simply of eight copies of the same value. Semantically then, we can replace our use of the general vector load instruction with the simpler gather instruction $x^v = b^s[\langle i^v \rangle]$, which does not require us to produce the vector version of b^s and which may also have a more efficient implementation. More importantly, the variable i^s here is a base induction variable with step 1. It is consequently possible to predict

the value of i^s at each of the next eight iterations: that is, the value of i^v will always be $\langle i^s, i^s{+}1, \ldots, i^s{+}7\rangle$. The vector load can then be seen to be a load of eight contiguous elements from b^s, for which we have provided the derived instruction $x^v = b^s[\langle i^s{:}\rangle]$, which performs a contiguous load of eight elements, as desired. Since this form of load is generally well supported in vector hardware, generating this specialized form in preference to the general form is highly desireable.

The second load instruction can then be vectorized in an exactly analagous fashion, resulting in two vector variables x^v and y^v. The addition instruction $z^s = +(x^s, y^s)$ can then be computed using vector addition on the vector variables, as $z^v = \langle+\rangle(x^v, y^v)$. To produce a vector version of the instruction $a^s[i^s] \leftarrow z^s$ which writes the computed value to the new array we follow a similar argument to that above for the loads to produce the $a^s[\langle i^s{:}\rangle] \leftarrow z^v$ instruction which performs a contiguous write of the values in z^v into the array a^s starting at offset i^s.

The last instruction of the loop, $i^s_1 = +(i^s, 1)$, is the induction variable computation, and as such requires special treatment. The result of this instruction is used for three purposes: in the computation of the test to decide whether to proceed with another iteration, to provide a new value for the induction variable in the next iteration, and to provide the value passed out on the exit edge.

For the latter two uses, the required value is easy to see. Each iteration of the vector loop corresponds to eight iterations of the scalar loop, and we require on entry to the loop that the induction variable i^s contain the value appropriate for the first of the eight iterations. Given the value of i^s for iteration j then, the back edge of the loop requires the value of i^s_1 for iteration $j + 7$, which is one plus the value of i^s on iteration $j+7$. Similarly, the exit edge of the loop requires the value of i^s_1 at iteration $j{+}7$. Since the computation of the value of i^s for each iteration depends on the value of itself in the previous iteration, we might imagine ourselves to be stuck. However, the nature of induction variables as affine transformations means that we can compute the value of i^s_1 at iteration $j{+}7$ directly: each iteration adds 1 to the original value, hence the value of i^s at iteration $j + 7$ is given by $i^s + 7$, and the value of i^s_1 at iteration $j + 7$ is $i^s + 7 + 1$. Hence we can compute the appropriate value of i^s_1 for both the back and the exit edge as $i^s + 8$.

The remaining use of the induction variable i^s_1 is to compute the loop exit test. The key insight here is to observe that it is no longer sufficient to check that there is at least one iteration remaining: in order to execute the vector loop again, we must have at least eight remaining iterations. Upon completion of a vector iteration computing scalar iterations $j, \ldots, j + 7$, the question that must be answered then is whether the scalar loop would always execute iterations $j + 8, \ldots, j + 15$. Because of the monotonic nature of the induction variable computation this in turn can be reduced to the question of wether or not the value of i^s_1 at iteration $j + 15$ is less than l^s. Since i^s_1 is an induction variable which is incremented by 1 on each iteration, this corresponds to changing the exit test to ask whether $i^s + 15 < l^s$.

4.3.2 Entry and exit code

Using the ideas from previous section, it is straightforward to write a vector version of the core loop. In addition to the issues touched on above, vectorizing this example also requires accounting for the possibility that there may be insufficient iterations to allow use of the vector loop, and also for the possibility that there may be extra iterations left over if the exit test succeeds with fewer than eight but more than zero iterations left. To account for these we keep around the original scalar loop, and use it to perform any iterations that cannot be computed with the vector loop. A preliminary test is inserted before the loop choosing either to target the scalar loop (if less than eight total iterations will be performed) or the vector loop (otherwise). Similarly, after the vector loop exits, a check is done for the presence of extra iterations, which if required are again performed using the scalar loop.

Using these transformations, we arrive at the final desired vectorized program.

$$
\begin{array}{ll}
\mathtt{L}^0(): & \mathtt{L}^{\text{check}}(): \\
\quad a^s = \mathtt{new}[l^s] & \\
\quad i^s_0 = 0 & \quad \mathtt{if}\ (i^s_3 < l^s)\ \mathtt{goto}\ \mathtt{L}^1(i^s_3) \\
\quad \mathtt{if}\ (7 < l^s)\ \mathtt{goto}\ \mathtt{L}^2(i^s_0) & \qquad\quad \mathtt{else\ goto}\ \mathtt{L}^{\text{exit}}(i^s_3) \\
\qquad\quad \mathtt{else\ goto}\ \mathtt{L}^1(i^s_0) & \\
 & \\
\mathtt{L}^2(i^s_2): & \mathtt{L}^1(i^s): \\
\quad x^v = b^s[\langle i^s_2{:}\rangle] & \quad x^s = b^s[i^s] \\
\quad y^v = c^s[\langle i^s_2{:}\rangle] & \quad y^s = c^s[i^s] \\
\quad z^v = \langle+\rangle(x^v, y^v) & \quad z^s = +(x^s, y^s) \\
\quad a^s[\langle i^s_2{:}\rangle] \leftarrow z^v & \quad a^s[i^s] \leftarrow z^s \\
\quad i^s_3 = +(i^s_2, 8) & \quad i^s_1 = +(i^s, 1) \\
\quad i^s_4 = +(i^s_2, 15) & \quad \mathtt{if}\ (i^s_1 < l^s)\ \mathtt{goto}\ \mathtt{L}^1(i^s_1) \\
\quad \mathtt{if}\ (i^s_4 < l^s)\ \mathtt{goto}\ \mathtt{L}^1(i^s_3) & \qquad\quad \mathtt{else\ goto}\ \mathtt{L}^{\text{exit}}(i^s_1) \\
\qquad\quad \mathtt{else\ goto}\ \mathtt{L}^{\text{check}}() &
\end{array}
\tag{2}
$$

4.4 Automatic vectorization

The reasoning of the previous section leads us to the desired result, but seems completely ad hoc and unsuitable to implementation in a compiler. Fortunately, it is possible to derive a simple compositional transformation which accomplishes the same thing in a general manner. In this section we describe the key ideas behind this transformation, and show that it generates good vector code.

The essential design principles that guide the design of the optimization are those of orthogonality and compositionality. As we will see, while the local transformation of instructions produces specialized code depending on the particulars of the instruction, each instruction is nonetheless translated in a compositional fashion in the sense that the translation does not depend on how the result is used. A consequence of this choice is that the compositional translation may produce numerous extraneous, redundant, or loop-invariant instructions, which could be avoided in a less compositional approach. The principle that we follow is that the elimination of extraneous, redundant, and loop-invariant instructions is an orthogonal issue, which is already addressed by dead-code elimination, common sub-expression elimination, and loop-invariant code motion respectively. So long as the vectorization transformation is defined appropriately, we can safely rely on these optimizations to sweep away the chaff, leaving behind only the essential bits.

The core of the vector transformation works by translating each scalar instruction into a sequence of instructions which compute three separate values: the scalar value, the vector value, and the last value. For a vector loop computing iterations $j, \ldots, j + 7$ of a scalar loop, the scalar value of a variable x is the value computed at iteration j, the last value is the value computed at iteration $j{+}7$, and the vector value is the vector containing the computed values for all of the iterations. While it is clear that there is always a degenerate implementation that simply computes the vector variable and then computes the scalar and last values by projecting out the first and last iteration value, we often compute the scalar and last values separately. This is crucial for a number of reasons: in the first place, we may be able to compute the vector variable more efficiently as a direct function of the scalar value; but more importantly, it will frequently be the case that the vector variable (or similarly the last value) will be unused. It is therefore important not to introduce a data-dependence of the scalar (or last) value on the vector value lest we keep an expensive-to-compute vector value live unnecessarily.

For clarity in the translation, we define meta-operators for producing fresh variables from existing scalar variables. For a variable x^s, the variable ${}^{\mathsf{fv}}x^s$ is taken to be a fresh scalar variable, which by convention will contain the scalar value of x^s in the vector loop. Similarly, we take ${}^{\mathsf{lv}}x^s$ to be a fresh scalar variable, which by convention will contain the last value of x^s; and we take ${}^{\mathsf{vv}}x^v$ to be a fresh vector variable, which by convention will contain the vector value of x^s.

Induction variables require a small amount of additional mechanism. For every base induction variable i^s with step s, we say that the *affine basis* of i^s is the vector $\langle 0*s, 1*s, \ldots, 7*s\rangle$ (the multiplications here are at the meta-level). Note that for a base induction variable i^s, adding i^s to each of the elements of its affine basis gives the values of i^s for each of the next eight iterations. Other (non-base) induction variables also have affine bases, which are computed by code emitted as part of the transformation described below, starting from the bases of the base induction variables. We use the notation ${}^{\mathsf{bv}}i^v$ to denote a fresh variable which by convention holds the affine basis for the scalar induction variable i^s.

It is occasionally necessary to introduce a *promoted* (or *lifted*) version of a variable: that is, for a variable x^s, to produce a vector variable containing the vector $\langle x^s, \ldots, x^s\rangle$. By convention, we use ${}^{\mathsf{pv}}x^v$ to denote a fresh variable containing the promoted version of x^s.

In order to simplify the algorithm, it is convenient to apply it only to loops for which all variables defined in the loop and used outside of the loop (that is, live out from the loop) are explicitly passed as parameters on the exit edge. This is in no way necessary, but avoids the need to rename variables throughout the rest of the program to preserve the SSA property. This does not restrict the generality of the algorithm since it is always possible to transform loops into this form.

The vectorization algorithm applies to single block loops of the form

$$
\begin{array}{l}
L(i^s_0, \ldots, i^s_n):\\
\quad I_0\\
\quad \ldots\\
\quad I_m\\
\quad \texttt{if}\ (i^s\ <\ l^s)\ \texttt{goto}\ L(i^s_{01}, \ldots, i^s_{n1})\\
\qquad\qquad \texttt{else goto}\ L^{\text{exit}}(x^s_0, \ldots, x^s_p)
\end{array}
$$

where the variables $i^s_0, \ldots, i^s_n$ are base induction variables, i^s is an induction variable with a positive step (the case where the step is zero or negative make no sense), and the variables $x^s_0, \ldots, x^s_p$ are the live out variables of the loop. All the instructions I_0, …, I_m must be scalar instructions (and thus define scalar variables), and cannot create new arrays. It is straightforward to support exit tests of the form $i^s\ \leq\ l^s$ as well. In all cases, the variable l^s must be defined outside of the loop (and hence be loop invariant).

We assume[2] the loop has a preheader, that is a block that ends with `goto` $L(ii^s_0, \ldots, ii^s_n)$ and that only this block and the loop transfer to *L*.

As suggested by the ad hoc development from the previous section, the vectorization transformation is required to produce three new pieces of code: an entry test that decides whether there are sufficient iterations to enter the vector loop; the core vector loop itself that performs the SIMD iterations; and a cleanup test that checks to see if there are any iterations remaining after the SIMD loop has finished, which must be processed by the original scalar loop which remains in the program unchanged. By convention we will take L^{vec} and L^{check} to be fresh labels for the new vector loop and cleanup test respectively.

[2] Assuming preheaders does not restrict our algorithm - transforming a program into an equivalent one where all loops have preheaders is a standard technique.

4.4.1 Entry tests

The job of the entry test is to determine if there are sufficient iterations to execute the vector loop, and if not to go straight to the original scalar loop. It also needs to set up some initial values for use in the vector loop. In particular, any variables used in the loop but not defined in the loop must have scalar, last value, and vector versions for use by the instructions in the vector loop..

First, for each variable y^s used in the loop but not defined in it (and not an entry variable), we add the following instructions to the end of the instructions of the preheader:

$$
\begin{array}{l}
{}^{\mathsf{fv}}y^s = y^s\\
{}^{\mathsf{lv}}y^s = y^s\\
{}^{\mathsf{vv}}y^v = \langle y^s, \ldots, y^s\rangle
\end{array}
$$

Next, we must determine whether the vector loop should be entered at all. The desired property of the entry test is that the vector loop should only be entered if there are at least 8 iterations to be performed. The monotonic nature of induction variables means that this question is equivalent to asking whether at least one more iteration remains after performing the first 7 iterations: that is, we wish to know the result of the exit test of the original loop on the 7th iteration. If the characteristic function for i^s is $s * \# + d$, then the value of i^s on the 7th iteration can be obtained by replacing $\#$ with 6 and calculating the result statically. The appropriate entry test is then of the form $ii^s\ <\ l^s$ where ii^s is defined as $ii^s = s*6+c$

The value $s*6+c$ is a compiler computed constant, and the compiler can statically determine that this does not overflow. If we generalize the notion of characteristic function to include loop-invariants in addition to constants, this test may require generating multiplication and addition instructions, and code must also be emitted to ensure that overflow has not been introduced.

4.4.2 Vector loop

To generate code for the main vector loop, we perform three operations. We first generate the last value and vector versions of the base induction variables, using their steps to produce these directly from the scalar values. For each of the original instructions of the loop we then generate instructions to compute the scalar, the vector, and last value versions of the instruction. Finally, we adjust the exit test.

For each entry variable i^s_j for $0\ \leq\ j\ \leq\ n$, which is a base induction variable of step s_j, we produce the vector, basis, lift, and last value versions of the induction variable as follows. The basis variable and lifted variables ${}^{\mathsf{bv}}i^v_j$ and ${}^{\mathsf{pv}}i^v_j$ are defined as

$$
\begin{array}{l}
{}^{\mathsf{bv}}i^v_j = \langle 0*s_j, \ldots, 7*s_j\rangle\\
{}^{\mathsf{pv}}i^v_j = \langle {}^{\mathsf{fv}}i^s_j, \ldots, {}^{\mathsf{fv}}i^s_j\rangle
\end{array}
$$

The vector version of i^s_j can then be defined directly as

$$
{}^{\mathsf{vv}}i^v_j = \langle + \rangle({}^{\mathsf{bv}}i^v_j, {}^{\mathsf{pv}}i^v_j)
$$

and the last value ${}^{\mathsf{lv}}i^s_j$ can be defined directly using the step.

$$
{}^{\mathsf{lv}}i^s_j = +({}^{\mathsf{fv}}i^s_j, 7*s_j)
$$

This initial step defines vector and last value variables for each base induction variable of the loop, as well as basis and lift variables. The translation of each instruction then proceeds compositionally, with the translated version of each instruction making use of the vector and last variable variables produced by the translation of all of the previous instructions. In addition, each induction variable makes use of the previously defined lift and basis variables to produce its own vector, last value, basis, and lift variables directly. The complete definition of this translation is given in Figure 2.

Adjusting the exit test of the loop is again straightforward using the step information. The desired new exit test must check whether

Instruction	Vector translation	Side conditions
$z^s = c$	${}^{\mathsf{fv}}z^s = c$ ${}^{\mathsf{vv}}z^v = \langle z^s, \ldots, z^s \rangle$ ${}^{\mathsf{lv}}z^s = {}^{\mathsf{fv}}z^s$	
$z^s = +(x^s, y^s)$	${}^{\mathsf{fv}}z^s = +({}^{\mathsf{fv}}x^s, {}^{\mathsf{fv}}y^s)$ ${}^{\mathsf{bv}}z^v = {}^{\mathsf{bv}}x^v$ ${}^{\mathsf{pv}}z^v = \langle {}^{\mathsf{fv}}z^s, \ldots, {}^{\mathsf{fv}}z^s \rangle$ ${}^{\mathsf{vv}}z^v = \langle + \rangle({}^{\mathsf{bv}}z^v, {}^{\mathsf{pv}}z^v)$ ${}^{\mathsf{lv}}z^s = +({}^{\mathsf{lv}}x^s, {}^{\mathsf{fv}}y^s)$	If z^s and x^s are induction variables.
$z^s = *(x^s, y^s)$	${}^{\mathsf{fv}}z^s = *({}^{\mathsf{fv}}x^s, {}^{\mathsf{fv}}y^s)$ ${}^{\mathsf{bv}}z^v = \langle * \rangle({}^{\mathsf{bv}}x^v, {}^{\mathsf{vv}}y^v)$ ${}^{\mathsf{pv}}z^v = \langle {}^{\mathsf{fv}}z^s, \ldots, {}^{\mathsf{fv}}z^s \rangle$ ${}^{\mathsf{vv}}z^v = \langle * \rangle({}^{\mathsf{bv}}z^v, {}^{\mathsf{pv}}z^v)$ ${}^{\mathsf{lv}}z^s = *({}^{\mathsf{lv}}x^s, {}^{\mathsf{fv}}y^s)$	If z^s and x^s are induction variables.
$z^s = op(x^s_0, \ldots, x^s_n)$	${}^{\mathsf{fv}}z^s = op({}^{\mathsf{fv}}x^s_0, \ldots, {}^{\mathsf{fv}}x^s_n)$ ${}^{\mathsf{vv}}z^v = \langle op \rangle({}^{\mathsf{vv}}x^v_0, \ldots, {}^{\mathsf{vv}}x^v_n)$ ${}^{\mathsf{lv}}z^s = {}^{\mathsf{vv}}z^v!7$	If z^s is not an induction variable.
$z^s = x^s[y^s]$	${}^{\mathsf{fv}}z^s = {}^{\mathsf{fv}}x^s[{}^{\mathsf{fv}}y^s]$ ${}^{\mathsf{vv}}z^v = {}^{\mathsf{fv}}x^s[\langle {}^{\mathsf{fv}}y^s{:} \rangle]$ ${}^{\mathsf{lv}}z^s = {}^{\mathsf{fv}}x^s[{}^{\mathsf{lv}}y^s]$	If x^s is loop invariant, and y^s is an induction variable with step 1.
$z^s = x^s[y^s]$	${}^{\mathsf{fv}}z^s = {}^{\mathsf{fv}}x^s[{}^{\mathsf{fv}}y^s]$ ${}^{\mathsf{vv}}z^v = {}^{\mathsf{fv}}x^s[\langle {}^{\mathsf{fv}}y^s{:}i \rangle]$ ${}^{\mathsf{lv}}z^s = {}^{\mathsf{fv}}x^s[{}^{\mathsf{lv}}y^s]$	If x^s is loop invariant, and y^s is affine with step i
$z^s = x^s[y^s]$	${}^{\mathsf{fv}}z^s = {}^{\mathsf{fv}}x^s[{}^{\mathsf{fv}}y^s]$ ${}^{\mathsf{vv}}z^v = {}^{\mathsf{fv}}x^s[\langle {}^{\mathsf{vv}}y^v \rangle]$ ${}^{\mathsf{lv}}z^s = {}^{\mathsf{fv}}x^s[{}^{\mathsf{lv}}y^s]$	If x^s is loop invariant, and y^s is not an induction variable.
$z^s = x^s[y^s]$	${}^{\mathsf{fv}}z^s = {}^{\mathsf{fv}}x^s[{}^{\mathsf{fv}}y^s]$ ${}^{\mathsf{vv}}z^v = \langle {}^{\mathsf{vv}}x^v \rangle[{}^{\mathsf{fv}}y^s]$ ${}^{\mathsf{lv}}z^s = {}^{\mathsf{lv}}x^s[{}^{\mathsf{fv}}y^s]$	If x^s is not loop invariant, and y^s is loop invariant.
$z^s = x^s[y^s]$	${}^{\mathsf{fv}}z^s = {}^{\mathsf{fv}}x^s[{}^{\mathsf{fv}}y^s]$ ${}^{\mathsf{vv}}z^v = \langle {}^{\mathsf{vv}}x^v \rangle[\langle {}^{\mathsf{vv}}y^v \rangle]$ ${}^{\mathsf{lv}}z^s = {}^{\mathsf{lv}}x^s[{}^{\mathsf{lv}}y^s]$	If both x^s and y^s are not loop invariant.
$x^s[y^s] \leftarrow z^s$	${}^{\mathsf{fv}}x^s[\langle {}^{\mathsf{fv}}y^s{:} \rangle] \leftarrow {}^{\mathsf{vv}}z^v$	If x^s is loop invariant, and y^s is an induction variable with step 1.
$x^s[y^s] \leftarrow z^s$	${}^{\mathsf{fv}}x^s[\langle {}^{\mathsf{fv}}y^s{:}i \rangle] \leftarrow {}^{\mathsf{vv}}z^v$	If x^s is loop invariant, and y^s is an induction variable with step i.
$x^s[y^s] \leftarrow z^s$	${}^{\mathsf{fv}}x^s[\langle {}^{\mathsf{vv}}y^v \rangle] \leftarrow {}^{\mathsf{vv}}z^v$	If x^s is loop invariant, and y^s is not an induction variable.
$x^s[y^s] \leftarrow z^s$	$\langle {}^{\mathsf{vv}}x^v \rangle[{}^{\mathsf{fv}}y^s] \leftarrow {}^{\mathsf{vv}}z^v$	If x^s is not loop invariant, and y^s is loop invariant.
$x^s[y^s] \leftarrow z^s$	$\langle {}^{\mathsf{vv}}x^v \rangle[\langle {}^{\mathsf{vv}}y^v \rangle] \leftarrow {}^{\mathsf{vv}}z^v$	If both x^s and y^s are not loop invariant.

Figure 2. Vector Block Instruction Translation

$i^s < l^s$ would succeed for at least eight more iterations. Since i^s is an induction variable, letting s be its step, it increases by s on each iteration. Thus the value of i^s on the next eight iterations will be ${}^{\mathsf{lv}}i^s + 0 * s, \ldots, {}^{\mathsf{lv}}i^s + 7 * s$. These will all be less than l^s if the last of them is (recall that s is positive and). Thus the desired new exit condition is ${}^{\mathsf{lv}}i^s + 7 * s < l^s$.

Putting this all together the vector loop will be, where i'^s is a fresh variable:

$L^{\text{vec}}({}^{\mathsf{fv}}i^s_0, \ldots, {}^{\mathsf{fv}}i^s_n)$:
  Instructions for base induction variables
  Transformed $I_0, \ldots, I_m$
  $i'^s = +({}^{\mathsf{lv}}i^s, 7 * s)$
  `if` $(i'^s < l^s)$ `goto` $L^{\text{vec}}({}^{\mathsf{lv}}i^s_{11}, \ldots, {}^{\mathsf{lv}}i^s_{n1})$
    `else goto` $L^{\text{check}}()$

Note that the back edge targets the vector loop, the exit edge targets the cleanup check, and the last values of the base induction variables are passed on the back edge.

4.4.3 On the subject of overflow

For the most part, the vectorization transformation here is careful to only compute the same values computed in the original program. Generating entry and exit tests violate this property. As mentioned in Section 4.4.1 the compiler can check statically that overflow does not occur on the computation of the constant for the initial entry test. For the exit test, it is sufficient to check before entry to the vector loop that $l^s < \text{MI} - 7 * s$ where MI is the largest representable integer and s is the step of i^s. For signed types, this check must be adjusted in the obvious way when loop bounds or steps are negative. In all cases, if the checks fail then the original scalar loop is used to perform the calculation using the original code.

4.4.4 Cleanup check

The cleanup check is responsible for determining if scalar iterations remain to be performed after the vector loop has exited. This in turn can be done simply by performing the original loop exit test. Scalar iterations remain to be performed exactly when ${}^{\mathsf{lv}}i^s < l^s$, where

Entry test	Vector loop	Cleanup check	Optimized result
$\mathtt{L}^0()$: $a^s = \mathtt{new}[l^s]$ $i_0^s = 0$ ${}^{\mathsf{fv}}a^s = a^s$ ${}^{\mathsf{vv}}a^v = \langle a^s, \ldots, a^s \rangle$ ${}^{\mathsf{lv}}a^s = a^s$ ${}^{\mathsf{fv}}b^s = b^s$ ${}^{\mathsf{vv}}b^v = \langle b^s, \ldots, b^s \rangle$ ${}^{\mathsf{lv}}b^s = b^s$ ${}^{\mathsf{fv}}c^s = c^s$ ${}^{\mathsf{vv}}c^v = \langle c^s, \ldots, c^s \rangle$ ${}^{\mathsf{lv}}c^s = c^s$ ${}^{\mathsf{fv}}l^s = l^s$ ${}^{\mathsf{vv}}l^v = \langle l^s, \ldots, l^s \rangle$ ${}^{\mathsf{lv}}l^s = l^s$ $i_0'^s = +(i_0^s, 6 * 1)$ $ii^s = +(i_0'^s, 1)$ if $(ii^s < l^s)$ goto $\mathtt{L}^2(i_0^s)$ else goto $\mathtt{L}^{\text{check}}(i_0^s)$	$\mathtt{L}^2({}^{\mathsf{fv}}i^s)$: ${}^{\mathsf{bv}}i^s = \langle 0 * 1, \ldots, 7 * 1 \rangle$ ${}^{\mathsf{pv}}i^v = \langle {}^{\mathsf{fv}}i^s, \ldots, {}^{\mathsf{fv}}i^s \rangle$ ${}^{\mathsf{vv}}i^v = \langle + \rangle({}^{\mathsf{bv}}i^v, {}^{\mathsf{pv}}i^v)$ ${}^{\mathsf{lv}}i^s = +({}^{\mathsf{fv}}i^s, 7 * 1)$ ${}^{\mathsf{fv}}x^s = {}^{\mathsf{fv}}b^s[{}^{\mathsf{fv}}i^s]$ ${}^{\mathsf{vv}}x^v = {}^{\mathsf{fv}}b^s[\langle {}^{\mathsf{fv}}i^s : \rangle]$ ${}^{\mathsf{lv}}x^s = {}^{\mathsf{fv}}b^s[{}^{\mathsf{fvlv}}i^s]$ ${}^{\mathsf{fv}}y^s = {}^{\mathsf{fv}}c^s[{}^{\mathsf{fv}}i^s]$ ${}^{\mathsf{vv}}y^v = {}^{\mathsf{fv}}c^s[\langle {}^{\mathsf{fv}}i^s : \rangle]$ ${}^{\mathsf{lv}}y^s = {}^{\mathsf{fv}}c^s[{}^{\mathsf{fvlv}}i^s]$ ${}^{\mathsf{fv}}z^s = +({}^{\mathsf{fv}}x^s, {}^{\mathsf{fv}}y^s)$ ${}^{\mathsf{vv}}z^v = \langle + \rangle({}^{\mathsf{vv}}x^v, {}^{\mathsf{vv}}y^v)$ ${}^{\mathsf{lv}}z^s = z^v!7$ ${}^{\mathsf{fv}}a^s[\langle {}^{\mathsf{fv}}i^s : \rangle] \leftarrow {}^{\mathsf{vv}}z^v$ ${}^{\mathsf{fv}}i_1^s = +({}^{\mathsf{fv}}i^s, 1)$ ${}^{\mathsf{bv}}i_1^v = {}^{\mathsf{bv}}i^v$ ${}^{\mathsf{pv}}i_1^v = \langle {}^{\mathsf{fv}}i_1^s, \ldots, {}^{\mathsf{fv}}i_1^s \rangle$ ${}^{\mathsf{vv}}i_1^v = \langle + \rangle({}^{\mathsf{bv}}i_1^v, {}^{\mathsf{pv}}i_1^v)$ ${}^{\mathsf{lv}}i_1^s = +({}^{\mathsf{lv}}i^s, 1)$ $i_1'^s = +({}^{\mathsf{lv}}i_1^s, 7 * 1)$ if $(i_1'^s < l^s)$ goto $\mathtt{L}^2({}^{\mathsf{lv}}i_1^s)$ else goto $\mathtt{L}^{\text{check}}()$	$\mathtt{L}^{\text{check}}()$: $i_2^s = +({}^{\mathsf{fv}}i^s, 7 * 1)$ $i_3^s = +(i_2^s, 1)$ if $(i_3^s < l^s)$ goto $\mathtt{L}^1(i_3^s)$ else goto $\mathtt{L}^{\text{exit}}(i_3^s)$	$\mathtt{L}^0()$: $a^s = \mathtt{new}[l^s]$ $i_0^s = 0$ $i_0'^s = +(i_0^s, 6)$ $ii^s = +(i_0'^s, 1)$ if $(ii^s < l^s)$ goto $\mathtt{L}^2(i_0^s)$ else goto $\mathtt{L}^{\text{check}}(i_0^s)$ $\mathtt{L}^2({}^{\mathsf{fv}}i^s)$: ${}^{\mathsf{lv}}i^s = +({}^{\mathsf{fv}}i^s, 7)$ ${}^{\mathsf{vv}}x^v = b^s[\langle {}^{\mathsf{fv}}i^s : \rangle]$ ${}^{\mathsf{vv}}y^v = c^s[\langle {}^{\mathsf{fv}}i^s : \rangle]$ ${}^{\mathsf{vv}}z^v = \langle + \rangle({}^{\mathsf{vv}}x^v, {}^{\mathsf{vv}}y^v)$ $a^s[\langle {}^{\mathsf{fv}}i^s : \rangle] \leftarrow {}^{\mathsf{vv}}z^v$ ${}^{\mathsf{lv}}i_1^s = +({}^{\mathsf{lv}}i^s, 1)$ $i_1'^s = +({}^{\mathsf{lv}}i_1^s, 7)$ if $(i_1'^s < l^s)$ goto $\mathtt{L}^2({}^{\mathsf{lv}}i_1^s)$ else goto $\mathtt{L}^{\text{check}}()$ $\mathtt{L}^{\text{check}}()$: $i_2^s = +({}^{\mathsf{fv}}i^s, 7)$ $i_3^s = +(i_2^s, 1)$ if $(i_3^s < l^s)$ goto $\mathtt{L}^1(i_3^s)$ else goto $\mathtt{L}^{\text{exit}}(i_3^s)$

Figure 3. Vectorized version of Example 1

${}^{\mathsf{lv}}i^s$ is the last value for i^s computed in the vector loop (since this represents the value of i^s from the last iteration which has been completed). Since the vector loop dominates the cleanup check, all its entry variables and variables it defines are in scope, so the cleanup check could be formed as follows:

$$L^{\text{check}}():$$

$$\mathtt{if}\ ({}^{\mathsf{lv}}i^s < l^s)\ \mathtt{goto}\ L({}^{\mathsf{lv}}i_{11}^s, \ldots, {}^{\mathsf{lv}}i_{n1}^s)$$
$$\mathtt{else\ goto}\ L^{\text{exit}}({}^{\mathsf{lv}}x_0^s, \ldots, {}^{\mathsf{lv}}x_p^s)$$

Note that the last values of the base induction variables are passed to the scalar loop and the last values of the live-out variables are passed to the exit label.

However, instead of using the last values computed in the vector loop, it is better to recompute them in order to avoid introducing data dependencies that may keep instructions live in the loop which could otherwise be eliminated. This turns out to be straightforward: the portion of the translation in Figure 2 that computes last values can simply be repeated. This results in a complete calculation of all of the last values, including ${}^{\mathsf{lv}}i^s$. While most of these instructions will turn out to be unnecessary, a single pass of dead code elimination is suffient to eliminate them.

4.5 Transformed example

The actual process of producing SIMD vector versions of loops seems at first quite ad hoc and difficult to do cleanly, but turns out to be amenable to a simple and straightforward translation which systematically produces vector versions of scalar instructions, relying on orthogonal optimizations to clean up unnecessary generated code. To illustrate the results of this process, Figure 3 shows the results of applying the algorithm to Example 1. First note that the base induction variables are just i^s of step 1, and the only other induction variabl is i_1^s also of step 1. The variables a^s, b^s, c^s, and l^s are used in the loop but not defined by it. Block $\mathtt{L}^0$ is the preheader for the loop. First we transform the preheader to define variables used by but not defined by the loop and to do the entry test. This results in Figure 3 under the header "Entry test". Next we generate the instructions for the base induction variable, transformed instructions of the loop, and adjusted exit condition to form the vector loop. The result of this appears under the header "Vector loop". Finally we form the cleanup check by regenerating the last-value computations and using the original loop-exit condition. For simplicity we just show enough instructions to compute the last value of i_1^s, the only needed last value. This block appears under the header "Cleanup check".

This code, as expected, is terribly verbose. However, by applying copy propagation followed by dead-code elimination, the extraneous parts disappear, yielding the code shown under the header "Optimized result". By applying common sub-expression elimination this code can be further improved by eliminating the calculation of i_2^s in favor of ${}^{\mathsf{lv}}i^s$, and then i_3^s in favor of ${}^{\mathsf{lv}}i_1^s$. Finally, simplifying the chained arithmetic expressions yields the same vectorized loop as derived by the initial ad hoc vectorization in the previous example.

4.6 Reductions

The SIMD vectorization process defined so far primarily applies to loops which serve to create new arrays. An additional idiom that is important to handle in vectorization is that of loops which serve (in total or in part) to compute a reduction using a binary operator. That is, instead of, or in addition to initializing a new array, each iteration of the loop accumulates a value into an accumulator parameter as a function of the value from the last iteration. For example, the innermost dot product of a matrix multiply routine generally takes the form of a reduction.

Adding support for reductions to the vectorization transformation is not difficult. In the same manner that we identify certain variables as induction variables and treat them specially, we iden-

tify variables that fit the pattern of reductions and add additional cases to the vectorization transformation to handle them differently. Specifically, we say that a variable x^s is a reduction variable if it is a parameter to the loop that is not an induction variable, and which is defined on the back edge of the loop by the application of an associative binary operator applied to x^s and some other variable. For example, the loop:

$$
\begin{array}{ll}
\mathtt{L}^0(): & \mathtt{L}^1(i^s, x^s): \\
\quad x_0^s = 1 & \quad y^s = c^s[i^s] \\
\quad i_0^s = 0 & \quad x_1^s = +(x^s, y^s) \\
\quad \mathtt{goto}\ \mathtt{L}^1(i_0^s, x_0^s) & \quad i_1^s = +(i^s, 1) \\
 & \quad \mathtt{if}\ (i_1^s < l^s)\ \mathtt{goto}\ \mathtt{L}^1(i_1^s, x_1^s) \\
 & \qquad\quad \mathtt{else}\ \mathtt{goto}\ \mathtt{L}^{\text{exit}}(x_1^s)
\end{array}
\tag{3}
$$

computes one more than the sum of the elements of the array c^s via a reduction (assuming that l^s is the length of c^s) and passes out the result on the exit edge as x_1^s. We say that x^s and x_1^s are reduction variables in this loop, with initial value 1.

The general strategy for SIMD vectorization of reductions is to pass a vector variable around the loop, each element of which contains a partial reduction. Implicitly, we re-associate the reduction so that element i of the vector variable contains the partial reduction of all of the values of the iterations which are congruent to i modulo the vector width (8 here). After exiting the vector loop, performing a horizontal reduction on the vector variable itself results in the full reduction of all the iterations (albeit performed in a different order, hence the requirement of associativity).

More concretely to vectorize a reduction variable x^s, we simply promote x^s to a vector variable x^v. In the preheader of the loop, we lift the definition of the initial value for the reduction variable to contain the original initial value in element 0, and the unit for the reduction operation in all other elements. In Example 3 this corresponds to defining the initial value as $x_0^v = \langle 1, 0, \ldots, 0 \rangle$ (since 0 is unit for addition). The reduction operation in the loop is lifted to perform the same vector operation pointwise, hence $x_1^v = \langle + \rangle(x^v, y^v)$ for the example. Finally, in the cleanup check block, the lifted reduction variable x_1^s is itself reduced to produce the final scalar result of the portion of the reduction performed by the vector loop (which is then passed in as the initial value to the scalar loop if subsequent iterations remain to be computed). This final reduction may be performed using scalar operations.

It is possible to maintain last value and scalar values for reduction variables within the vector loop as well as the vector value, in the same fashion as was done for ordinary operations. However, doing so requires performing a full reduction on the vector value on each iteration which is potentially expensive enough to make SIMD vectorization a poor idea. Consequently, we simply choose to reject SIMD vectorization of loops which would require the calculation of scalar or last values for reduction variables within the body of the vector loop.

Unfortunately, the associativity requirement for the binary operation used in reductions can be problematic since most floating point operations (e.g. addition and multiplication) are not associative. In practice, reductions over floating point operators are important idioms. By default, our compiler does not re-associate floating point operations, and hence will refuse to vectorize loops which perform reductions over floating point data. We provide a flag which permits the compiler to re-associate floating point operations only for SIMD vectorization (but not other optimizations) which allows such loops to be vectorized at the expense of occasionally producing slightly different results from the sequential version.

4.7 Dependence Analysis

Attentive readers may at this point be concerned that we have left aside a critical point in our development. While we have shown *how* to vectorize a loop, we have said nothing about when (or why) it might be valid to do so. The crux of any kind of automatic vectorization lies in determining when it is semantics preserving to execute multiple iterations of a scalar loop in parallel. In the most general case this can be a tremendously hard problem to solve, and there is a vast literature on techniques for showing that particular loops can validly be vectorized (see for example Bik[3] for a good introduction and bibliography). For the particular case of immutable arrays however, the problem becomes drastically easier. In this section we give a brief overview of why this is the case.

We say that an instruction I_2 is dependent on an instruction I_1 if I_2 must causally follow I_1. There are a number of different reasons that this ordering might be required, each inducing a different kind of dependence. Flow dependence (or data-dependence) is a standard dataflow dependence induced by a read of the value of a variable or location which was written at an earlier point in the program execution. This is sometimes known as a read-after-write dependence. An anti-dependence on the other hand is induced by a write after a read: that is, a location or variable is updated with a new value after an old value has been read, and therefore the write must happen after the read to avoid changing the value seen by the read. Finally, output dependence is dependence induced by a write after a write: that is, a location or variable is updated with a new value which overwrites an old value written by a previous statement (and hence the writes cannot be re-ordered without affecting the final value of the written to location or variable).

The key insight for automatic vectorization of immutable arrays is that neither anti-dependence nor output dependence can occur. This is obvious by construction, since the nature of immutability is such that the only writes are initializing writes. Therefore, there cannot be a write after a read (since this would imply a read of uninitialized data), nor can there be a write after a write (since this would imply mutation). The only remaining dependence then is flow-dependence. Flow-dependences which are carried by variable definitions and uses are straightforward to account for, and do not generally cause unusual trouble. In general, it is possible for loop-carried flow-dependences to block vectorization, however. For example, consider the following simple program:

$$
\begin{array}{ll}
\mathtt{L}^0(): & \mathtt{L}^1(i^s): \\
\quad a^s = \mathtt{new}[l^s] & \quad a^s[i^s] \leftarrow i^s \\
\quad a^s[0] \leftarrow 0 & \quad x^s = a^s[i^s - 1] \\
\quad \mathtt{goto}\ \mathtt{L}^1(1) & \quad \mathtt{if}\ (i^s < l^s)\ \mathtt{goto}\ \mathtt{L}^1(i^s + 1) \\
 & \qquad\quad \mathtt{else}\ \mathtt{goto}\ \mathtt{L}^{\text{exit}}(x^s)
\end{array}
$$

While this program uses only initializing writes, it nonetheless incurs a loop-carried dependence in which a read in one iteration depends on a write in a previous iteration. Clearly then, the problem of dependence analysis is not completely trivial even for functional programs even after eliminating the problems of anti-dependences and output-dependences. Note that while this example program reads and writes a^s through the same variable name, there is nothing preventing the binding of a^s to a different variable name, and hence allowing accesses (reads or initializing writes) to the array via an alias.

In practice however, all programs that we have considered (and indeed, based on the way that immutable arrays are produced, all programs that we are at all likely to consider) have the very useful property that they are allocated and initialized in the same lexical scope. The implication of this is that it is usually trivially easy to conservatively determine that an initializing write in a loop does not alias with any of the reads in the loop simply by checking that no aliases for the newly allocated array are introduced before the

loop. As a result, a very naive analysis can be quite successful at breaking read after write dependences in these style of loops.

5. Benchmarks

We show performance results for six benchmarks, measured on an Intel Xeon E5-4650 (Sandybridge) processor implementing the AVX SIMD instruction set. Two of the benchmarks are small synthetic benchmarks. The first computes the pointwise sum of two arrays of 32-bit floating point numbers using the following code:

```
addV v1 v2 = zipWith (+) v1 v2
```

The second computes the horizontal sum of an array of 32-bit floating point numbers using the following code:

```
sumV v = foldl (+) 0 v
```

Both of the kernels are implemented exactly as given, using the operations from our modified Data.Vector.Unboxed library. These code snippets are each run in a benchmark harness which generates test data and repeatedly runs the same kernel a number of times in order to increase the runtime to reasonable lengths for benchmarking.

The third benchmark is an n-body simulation kernel which iteratively calculates the forces between n gravitational bodies, represented as 32-bit floating point numbers. Each iteration uses the results from the previous iteration as its initial configuration. The code is written using the REPA array library.

The fourth benchmark is a matrix-matrix multiply routine written using the matrix multiply algorithm from the REPA array library. The matrix elements are double precision (64-bit) floating point numbers. The program generates random arrays, performs a single multiplication, computes a checksum, and writes the result out to a file. The portion of the program that is timed and reported here covers only the matrix multiplication and the checksum computation. The measurements presented here are for 700 by 700 element arrays.

The fifth benchmark is a two dimensional five-by-five stencil convolution written using the REPA stencil libraries [8]. The convolution is repeatedly performed 4000 consecutive times in a loop.

The final benchmark is the "blur" image processing example included with the REPA distribution. The benchmark was modified slightly to match our benchmark harness requirements, with timing code inserted around the kernel of the computation to avoid including file input/output times in the measurements. No changes to the code implementing the algoritm itself were made. The timings presented measure the result of performing twenty successive blurrings on a 600x512 pixel image.

The performance of these six benchmarks is given in figure 4. We show runtime normalized to our standard scalar compiler without vectorization, since the primary goal is to demonstrate the speedups obtained from automatic SIMD vectorization optimization. We also show the performance of unmodified GHC 7.6.1 using both the standard back end and the new LLVM backend [13]. This is intended primarily to demonstrate that the scalar code from which we begin is suitably optimized. Both the unmodified and our own modified version of GHC were run with the optimization flags "-O2", "-msse2". Various combinations of the flags "-fno-liberate-case", "-funfolding-use-threshold", "-funfolding-keeness-factor" were also used following the documentation of the REPA libraries, and based on some limited experimentation as to which arguments gave better performance. In all cases both our scalar and SIMD compiler preserve floating point value semantics and require the backend C compiler to do the same. The only exception is that the flag discussed in Section 4 to permit the vectorization of floating point reductions was used. For some benchmarks, this caused small perturbations in the results.

For the vector-vector addition benchmark, our baseline compiler gives large speedups over both the standard GHC backend and the LLVM backend. Unfortunately this benchmark is poorly structured for measuring floating point performance. Performance analysis shows that the run time is almost entirely dominated by memory costs associated with initializing the newly allocated arrays. Each iteration of the vector-vector product allocates a new result array which is not in cache, and the inner loop does not do enough work to hide the latency of the memory accesses. The vector sum benchmark on the other hand performs a reduction over the input vector producing only a scalar result. Consequently memory latency issues are not critical since the input vectors are in cache, and the SIMD vectorized version of the code runs in half the time of our baseline compiler.

The N-Body simulation benchmark gives the best results for SIMD vectorization, since the core of the inner loop does substantially more floating point work on each iteration. While our baseline compiler gives only slightly better performance than the LLVM backend, the SIMD vectorized version takes only 23% of the runtime of the scalar version.

The matrix multiplication routine also shows good benefits from SIMD vectorization, runing in 64% of the time of the scalar version (recall that this benchmark uses double precision floating point).

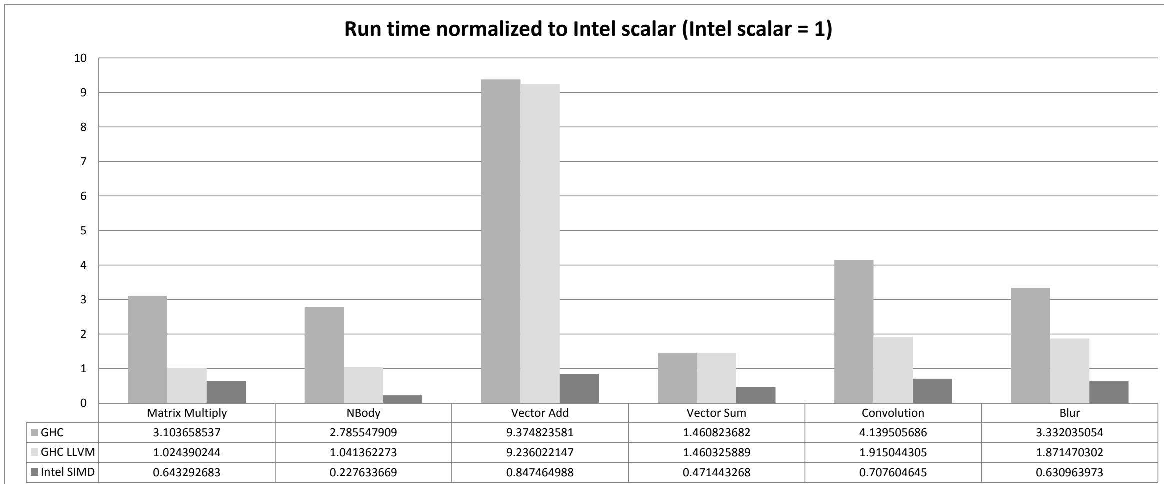
The convolution and blur benchmarks suffer also get decent speedups from vectorization. GHC (or the REPA libraries) seem to be arranging for the inner loop to be executed in an unrolled fashion. Unfortunately, this means that the SIMD vector version cannot generate unit stride loads supported by the native architecture and must emulate them. The resulting scalar loads and vector shuffles limit the benefit of vectorization somewhat. Nonetheless, we are able to still get good speedup over scalar code.

Of particular importance to note with respect to all of the benchmarks but most especially for the last two is that the timings presented here are not for the vectorized loop alone, but for the entire kernel of the program (essentially everything except the input (or input generation) and output).

6. Related and Future Work, Conclusions

There is a vast and rich literature on automatic SIMD vectorization in compilers. Bik [3] provides an excellent overview of the essential ideas and techniques in a modern setting. Many standard compiler textbooks also provide at least some rudimentary introduction to SIMD vectorization. The NESL programming language [4] pioneered the idea of writing nested data parallel programs in a functional language, an idea which has more recently been picked up by the Data Parallel Haskell project [11]. A related project focusing on regular arrays [6] (REPA) has had good success in getting array computations to scale on multi-processor machines. Programs written using the REPA libraries are promising targets for SIMD vectorization, either automatic as in this work, or via explicit language constructs in the library implementation. Mainland et al [9] have pursued this latter approach. They implement primitives in GHC that correspond to SIMD instructions and modify the stream fusion used in the Data.Vector library to generate these SIMD primitives. In this way, they achieve SIMD vectorization in the library, where we achieve SIMD vectorization in the compiler. Two different, possibly complimentary, ways to achieve the same ends.

Our initial implementation of SIMD vectorization targets only simple loops with no conditional tests. An obvious extension to this work would be to incorporate support for predicated instructions to allow more loops to be SIMD vectorized. However, hardware support for this is somewhat limited in current processors, and the performance effects of generating predicated code are harder to predict. A more limited effort to target the special case of conditionals performing array bounds checks seems like a promising



	Matrix Multiply	NBody	Vector Add	Vector Sum	Convolution	Blur
GHC	3.103658537	2.785547909	9.374823581	1.460823682	4.139505686	3.332035054
GHC LLVM	1.024390244	1.041362273	9.236022147	1.460325889	1.915044305	1.871470302
Intel SIMD	0.643292683	0.227633669	0.847464988	0.471443268	0.707604645	0.630963973

Figure 4. Benchmark Performance

area for investigation. As functional languages tend to allocate a lot, supporting allocation of heap objects inside vectorized loops could be important. A vector allocation instruction allocates 8 objects and places pointers to them in a vector variable. Implementing this operation for fixed-size objects is straightforward and should be quick—a single limit check, vector instructions to compute the pointers, and a scatter to initialize the object meta-data used by the garbage collector. Implementing vectorized allocation of variable-length arrays should also be straightforward. A special-case instruction for when the length is loop-invariant would likely have a faster implementation.

To summarize, we have utilized the naturally data-parallel typical functional-language operations on immutable arrays like maps, folds, zips, etc., utilized the large body of work on eliminating intermediate arrays, and utilized our own internal operations on immutable arrays to produce nice loops ammenable to SIMD vectorization. Implementing that SIMD vectorisation automatically is a surprisingly straightforward process. A key part of this simplicity is the use of immutable objects and operations on them, and in particular on our use of initializing writes. As performance from using SIMD instruction sets is important on modern architectures, our techniques are an important and low-hanging fruit for functional-language implementors.

References

[1] T. Anderson, N. Glew, P. Guo, B. T. Lewis, W. Liu, Z. Liu, L. Petersen, M. Rajagopalan, J. M. Stichnoth, G. Wu, and D. Zhang. Pillar: A parallel implementation language. In V. Adve, M. J. Garzarán, and P. Petersen, editors, *Languages and Compilers for Parallel Computing*, pages 141–155. Springer-Verlag, Berlin, Heidelberg, 2008. ISBN 978-3-540-85260-5.

[2] A. Appel and T. Jim. Shrinking lambda expressions in linear time. *Journal of Functional Programming*, 7(5), Sept. 1997.

[3] A. J. C. Bik. *The Software Vectorization Handbook*. Intel Press, 2004. ISBN 0974364924.

[4] G. E. Blelloch. Programming parallel algorithms. *Commun. ACM*, 39 (3):85–97, Mar. 1996. ISSN 0001-0782.

[5] M. Fluet and S. Weeks. Contification using dominators. In *Proceedings of the sixth ACM SIGPLAN international conference on Functional programming*, ICFP '01, pages 2–13, Florence, Italy, 2001. ACM. ISBN 1-58113-415-0.

[6] G. Keller, M. M. Chakravarty, R. Leshchinskiy, S. Peyton Jones, and B. Lippmeier. Regular, shape-polymorphic, parallel arrays in Haskell. In *Proceedings of the 15th ACM SIGPLAN international conference on Functional programming*, ICFP '10, pages 261–272, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-794-3.

[7] K. Kennedy, C. Koelbel, and H. Zima. The rise and fall of High Performance Fortran: an historical object lesson. In *Proceedings of the third ACM SIGPLAN conference on History of programming languages*, pages 1–22. ACM, 2007.

[8] B. Lippmeier and G. Keller. Efficient parallel stencil convolution in haskell. In *Proceedings of the 4th ACM symposium on Haskell*, Haskell '11, pages 59–70, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0860-1.

[9] G. Mainland, R. Leshchinskiy, S. P. Jones, and S. Marlow. Haskell beats C using generalized stream fusion. Available at `http://research.microsoft.com/en-us/um/people/-simonpj/papers/ndp/haskell-beats-C.pdf`; Unpublished, 2013.

[10] L. Petersen and N. Glew. GC-safe interprocedural unboxing. In *Proceedings of the 21st international conference on Compiler Construction*, CC'12, pages 165–184, Tallinn, Estonia, 2012. Springer-Verlag. ISBN 978-3-642-28651-3.

[11] S. Peyton Jones. Harnessing the multicores: Nested data parallelism in Haskell. In *Proceedings of the 6th Asian Symposium on Programming Languages and Systems*, APLAS '08, pages 138–138, Bangalore, India, 2008. Springer-Verlag. ISBN 978-3-540-89329-5.

[12] W. Pugh. A practical algorithm for exact array dependence analysis. *Communications of the ACM*, 35(8):102–114, 1992.

[13] D. A. Terei and M. M. Chakravarty. An llvm backend for ghc. In *ACM Sigplan Notices*, volume 45, pages 109–120. ACM, 2010.

[14] N. Vasilache, C. Bastoul, A. Cohen, and S. Girbal. Violated dependence analysis. In *Proceedings of the 20th annual international conference on Supercomputing*, pages 335–344. ACM, 2006.

[15] S. Weeks. Whole-program compilation in MLton. In *Proceedings of the 2006 workshop on ML*, ML '06, pages 1–1, Portland, Oregon, USA, 2006. ACM. ISBN 1-59593-483-9.