Vector LLVA: A Virtual Vector Instruction Set for Media Processing *

Robert L. Bocchino Jr. Vikram S. Adve

Department of Computer Science
University of Illinois at Urbana-Champaign
201 N. Goodwin Ave., Urbana, IL 61801
{bocchino,vadve}@cs.uiuc.edu

Abstract

We present Vector LLVA, a virtual instruction set architecture (V-ISA) that exposes extensive static information about vector parallelism while avoiding the use of hardware-specific parameters. We provide both arbitrary-length vectors (for targets that allow vectors of arbitrary length, or where the target length is not known) and fixed-length vectors (for targets that have a fixed vector length, such as subword SIMD extensions), together with a rich set of operations on both vector types. We have implemented translators that compile (1) Vector LLVA written with arbitrary-length vectors to the Motorola RSVP architecture and (2) Vector LLVA written with fixed-length vectors to both AltiVec and Intel SSE2. Our translator-generated code achieves speedups competitive with handwritten native code versions of several benchmarks on all three architectures. These experiments show that our V-ISA design captures vector parallelism for two quite different classes of architectures and provides virtual object code portability within the class of subword SIMD architectures.

Categories and Subject Descriptors D.3.4 [*Software*]: Programming Languages—Processors

General Terms Performance, Languages

Keywords Multimedia, Vector, SIMD, Virtual Instruction Sets

1. Introduction

High-performance media processing applications are becoming increasingly important on a wide range of systems. Like most applications, these programs are generally compiled and shipped in the form of native binary code for a particular processor architecture. This approach has two particular drawbacks for media processing applications where careful tuning for very high performance is important. First, even when written in a source-level language (e.g., C), application code must often be tuned for hardware details such as available operations and memory access restrictions, and such tuning is not likely to carry over from one architecture to another (e.g., Motorola's AltiVec vs. Intel's SSE2). This makes it difficult and expensive to write a single source program that achieves the highest performance on different architectures, even though these architectures are fairly similar in most important ways. Second, as processor generations evolve within a single architecture, it may even be difficult to tune code for multiple generations of a single architecture (e.g., if new instructions are added for a common vector idiom or explicit support for streaming vector data from memory is added in the memory system).

One approach to addressing these drawbacks is to *defer native code generation and processor-specific optimization* until as late as possible (e.g., until install time on an end-user's system). Bytecode-based virtual machines such as JVM [19] and CLI [21] have this ability (by using just-in-time compilation), but their primary goal is not maximum optimization. In previous work, we have described a low-level virtual instruction set and compilation strategy designed to enable such "lifelong" code generation and optimization, aimed primarily at high performance [18, 1]. This instruction set is called Low Level Virtual Architecture (LLVA) and is aimed at supporting general purpose applications on ordinary superscalar processors.

In this paper, we describe a set of extensions to LLVA to support vector (SIMD) parallelism for media processing, which we refer to as Vector LLVA. We believe that bringing the virtual instruction set approach to vector code would have the following benefits:

1. The virtual instruction set (V-ISA) can provide a single portable virtual object code across multiple classes of vector architectures, e.g., streaming processors like RSVP and subword-SIMD ISA extensions like AltiVec and SSE. As should be clear from the design in Section 2, this code can be compiled and run on a wide range of processor architectures from multiple processor classes. Note, however, that achieving near-hand-coded performance on all these classes from a single representation of a program may be difficult, and we do not address that goal in this work.
2. The V-ISA can provide near-hand-coded performance *within each architecture class*, including across multiple architectures in a single class, e.g., AltiVec and SSE2 in the class of subword SIMD extensions. This is particularly important for multimedia vector architectures, which tend to expose many low-level architectural details, and to be notoriously difficult targets for programming and compilation. While some of the difficulties are fundamental to the architecture (e.g., alignment), many can be abstracted away by the V-ISA.

* Robert Bocchino is the recipient of a Motorola Partnerships in Research Grant. This work is supported in part by an NSF CAREER award (EIA-0093426).

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

VEE'06 June 14–16, Ottawa, Ontario, Canada.

Copyright © 2006 ACM 1-59593-332-6/06/0006...$5.00.

3. The programmer or high-level compiler can generate a single bytecode representation that is later optimized by a *processor-specific* code generator (the “translator”), with full information about all the details of a particular processor chip and system configuration. Furthermore, the difficult issues of auto-vectorization and vector code optimization can be confined to the “front-end” compiler, leaving the translator simple and allowing it to focus on machine-level optimization and code generation issues.

Other qualitative benefits not evaluated in this work, but discussed briefly in Section 5, include portability across generations of a single architecture and significant productivity savings for designers through the use of common programming tools (e.g., for debugging, instrumentation, and tuning).

The following scenario can help illustrate these benefits. Consider a media processing application being developed for a mix of streaming and subword SIMD platforms. At design time, the application would be structured in terms of one or more available kernel libraries and initially implemented in ordinary C, with minimal use of explicit Vector LLVA code (expressed via C intrinsics). If a “front-end” auto-vectorizing compiler [16, 4, 24, 22, 27] is available, the need for these intrinsics is greatly reduced; otherwise, all vector computations must be expressed using the intrinsics. The library could be written in Vector LLVA form (either directly or in C with heavy use of the intrinsics) and tuned carefully and separately for the two classes, but not for each individual hardware ISA. The (hopefully small) fraction of remaining application code that consumes significant execution time could also be incrementally tuned for each class. Overall, debugging and tuning of the application would occur for one version of code for each architecture class, and *not* for each specific target architecture. The overall code would be linked and shipped in two versions, one for each class. The virtual object code would be optimized and compiled to native code separately on each target device, exploiting both processor-specific information and application-level information. The latter could be extracted from all parts of the code because Vector LLVA is carefully designed for effective optimization (unlike machine-level assembly language). Overall, the resulting program would reflect all these stages of tuning – library, application, link-time, and install-time, and potentially even between program executions based on end-user-specific profile information [1].

Designing a virtual vector instruction set for media processing that provides all these benefits presents new challenges (over and above the design of base LLVA) for at least two reasons. First, media-processing applications demand very high performance, and important kernels are often *hand-tuned at the assembly level*. A V-ISA that reduced performance significantly compared with hand-tuned assembly code would likely not be acceptable. In fact, *we consider this criterion to be the primary design goal in our work*. Second, as noted earlier, multimedia vector architectures tend to expose many low-level, idiosyncratic hardware details, and abstracting away the important common details in a compact design (i.e., not a union of features of different processors) requires substantial design effort.

The design of Vector LLVA includes several novel features and/or capabilities, in order to meet the challenges above and to address other design goals we considered important:

- *Machine-independent operations and semantics that capture important hardware features*: This is the basic goal of Vector LLVA and, to our knowledge, our V-ISA is the first *virtual* instruction set specifically designed for vector operations on modern vector architectures. The closest previous example we are aware of, the Stream Virtual Machine, has similar design goals but focuses on abstracting communication between architecture components (while using the ordinary native instruction set for each processor). We compare this work with Vector LLVA briefly in Section 4.
- *Explicit support for variable-length and fixed-length vector operations*: To enable very high performance programming and tuning, we distinguish two broad classes of vector architectures that are proposed or available today: architectures that support operations on arbitrary-length vectors or “streams” [8, 13] and subword SIMD extensions [9, 12, 5]. To support both classes, our virtual instruction set includes both variable-length vectors and fixed-length vectors, and includes several features to enable careful tuning for each processor class, listed below. The variable-length vectors can also be used to achieve portability across the classes, though perhaps not with optimal performance.
- *Asynchronous load and store semantics for long vectors*: We provide a way of expressing vector loads and stores that allows the translator to interleave the individual loads and stores with no further analysis. This allows vector computations to be “blocked” for different vector lengths much more easily than if vector loads and stores were “atomic” operations. It also allows streaming prefetches of long vectors from and to memory when hardware allows (this is possible on RSVP and we believe should be a key feature on future processors).
- *Alignment*: Memory loads and stores can specify an alignment attribute (treated as a guarantee by the programmer), allowing the translator to generate efficient code for properly aligned values.
- *SSA form and type information*: Vector LLVA inherits two important features from base LLVA: an infinite-register SSA form and type information. These features apply to all vector registers and operands, and allow more precise analysis and more flexible optimization and code generation in the translator, while reducing the need for complex program analysis on virtual object code.
- *Keeping the translator simple*: A Vector LLVA program tuned for a processor class can be translated to native code for a particular member of that class without requiring sophisticated program analysis or an auto-vectorizing compiler. This is necessary to keep the translator fast and with a small memory footprint, so that it can be invoked at install time or even runtime, and can be used in small-scale systems like a PDA or a cell phone.

We evaluate the suitability of Vector LLVA for high performance programming on representative architectures from the two classes identified above. For the first class (arbitrary-length vector architectures), we consider the Motorola Reconfigurable Streaming Vector Processor (RSVP) [8]. For the second class (subword SIMD architectures), we consider both PowerPC/AltiVec [9] and Intel SSE/SSE2 [5]. Our evaluation examines three issues: suitability for writing high performance code for a class, feasibility of writing a translator without overly complex compiler technology, and experimental results showing the performance achieved for a collection of important media processing kernels.

The experimental results in this paper show that our translator can compile Vector LLVA kernels into code competitive with hand-coded assembly for RSVP, and we discuss how programs can be written at the Vector LLVA level to tune for this processor. The results also show that kernels can be written and tuned in a *single* Vector LLVA representation that is compiled to *both* AltiVec and SSE2, achieving performance competitive with hand-coded assembly on both processors. (In contrast, the Intel autovectorizer for SSE2 [5] is unable to vectorize scalar versions of three out of five of our SSE2 benchmarks.) Together, these results provide

evidence that our design achieves the goals we set out earlier in this section: a single machine-independent virtual instruction set and programming model, but with the ability to tune code separately for distinct classes of processors.

Section 2 describes Vector LLVA and the rationale behind its key design features. Section 3 discusses the suitability of our design for the two classes, including the experimental results summarized above. Section 4 compares our approach with previous programming and compilation strategies for media processors. We conclude with a brief summary of the strengths and weaknesses of Vector LLVA today and a discussion of how it can be used within the broader context of programming media processing applications.

2. Vector Extensions to LLVA

We designed Vector LLVA to *express* vector parallelism and related information (such as possibilities for interleaving loads and stores) and *communicate* that information to the translator, while we shift the burden to the programmer or high-level compiler to *extract* the information and ensure that the resulting program is correct. This allows us to

- Provide a clean separation between the high-level (programming and high-level compiler) functions and the low-level translation. We want to keep as many of the machine-specific details as possible in the low-level translator, while still providing high performance.
- Factor as much of the analysis that is common across various architectures as possible into a single high-level compiler. For example, alias, dependence, and alignment analysis can be done once and used to generate Vector LLVA code that expresses possibilities for vectorization and instruction scheduling. Different translators can use this information for different targets without having to derive it repeatedly.
- Keep the translator simple. We envision that the translator would reside in memory on the target machine, and that translation would occur at install time or even at runtime. Therefore, the translator should be fast and should have a small memory footprint.

The base LLVA is a RISC-like three-address, load-store instruction set, but with several unusual features compared with traditional processor ISAs: (1) an infinite virtual register set; (2) type information suitable for sophisticated compiler optimization, including pointers, structures, arrays, and functions; (3) an explicit control flow graph (CFG); (4) the use of Static Single Assignment (SSA) form for all virtual registers, which provides an explicit register-level dataflow representation; (5) mechanisms to expose stack, global, and heap memory; and (6) an exception mechanism that allows greater flexibility for code reordering by the translator. Equally important, the V-ISA does *not* include machine-level details that are better suited to the native ISA, including a fixed register set, stack frame layout, low-level addressing modes, limits on immediate constants, delay slots, speculation, predication, or explicit interlocks.

Table 1 shows the new instructions defined in this work. The instruction set retains all the key properties of the base LLVA, including an infinite virtual register set, SSA form and explicit type information for all virtual registers, the RISC-style load-store nature of the ISA, and simple primitive operations that allow extensive optimization by external compilers.

2.1 The Vector Type

Vector LLVA extends the existing LLVA type system by adding a *vector type* that represents n values of any LLVA scalar type (integer, floating point, or Boolean), where $n > 0$ is an arbitrary value not necessarily known at compile time. We chose to limit the vector element types to scalars because we wanted to keep the representation close to the target architecture. Higher-level constructs such as arrays of arrays or arrays of structs map straightforwardly to Vector LLVA. If the access pattern is regular enough, strided loads and stores can be used to express these constructs. Otherwise, one can use multiple vectors contained within outer loops.

The vector type supports compilation to a target machine or family of machines (such as AltiVec and SSE) that provide vector registers of fixed length. In this case, n is a compile-time constant. The vector type also supports compilation to target architectures that provide variable-length vectors (such as the Motorola RSVP architecture), or to multiple architectures that provide different vector lengths. In this case, n is unknown at compile time.

The vector type is a first-class type, and it behaves like other first-class types in LLVA. It is held in SSA-form virtual registers. It can be used in computational instructions or passed to functions. It can be loaded from or stored to memory, as discussed below. We provide an instruction `vimm` that populates all elements of a vector with the same (immediate) value. We also provide instructions for moving vector data between registers and memory, as discussed below.

When the vector length is a compile-time constant, our implementation stores the vector length in the LLVA type. Otherwise, we do not explicitly represent the length in the type; instead, the back end infers the vector lengths from the operations that produce vectors (vector load and `vimm`) and propagates them to other parts of the program. We chose this implementation because our base (scalar) LLVA implementation does not support parametric types. In an implementation that does support parametric types, it should be straightforward to encode all vector lengths in the type information.

2.2 Vector-Memory Operations

We extend the LLVA `load` and `store` instructions to operate on vector types. The `load` instruction takes a pointer to a vector in memory (i.e., a contiguous set of scalar values of specified length) and loads that vector into a register. The `store` instruction takes a vector register and a pointer to a vector in memory, and it stores the register value into memory.

Load and store operations on vectors are restricted to stride-one memory access. We also define new instructions `vgather` and `vscatter` for accessing arrays in memory with arbitrary dimension and stride, or with indirection vectors. For targets such as RSVP that explicitly support these access patterns, the `vgather` and `vscatter` instructions translate directly to operations on the target machine. For other architectures (such as AltiVec and SSE, which provide only limited access to memory at strides other than one), the translator must introduce data movement operations or produce scalar code. As discussed below, we have designed the semantics of `vgather` and `vscatter` to make this easy for the translator.

The arguments to `vgather` are a memory address p and one or more pairs (I, m), where I is an index group and m is a multiplier. An index group specifies a list of indices, either as a triple (start, end, stride), or as an explicit vector of indices. The explicit vector of indices provides the full traditional gather (indexed load) capability found in, e.g., Fortran 90. Values are loaded from memory address p into register r according to the following formula:

$j = 0$
for each index i_1 in index group I_1
...
  for each index i_n in index group I_n
    $r[j++] = p[i_1 \cdot m_1 + \ldots + i_n \cdot m_n]$

Table 1. Summary of Vector Extensions

Instruction	Explanation of Syntax	Function
Vector-Memory Operations and Immediate Values		
vimm i, n	i is a scalar value; n is a length	Form a vector of length n in which each position has value i
load p	p is a pointer to a vector in memory	Load a vector into a vector register
store v, p	v is a vector register; p is a pointer to a vector in memory	Store register v into memory location p
vgather $p, I_1, m_1, I_2, m_2, \ldots, I_n, m_n$	p is a pointer; I_i is either (1) a triple l_i, u_i, s_i giving lower bound, upper bound, and stride, or (2) a vector of indices; m_i is a multiplier	Gather an array slice into a vector register
vscatter $v, p, I_1, m_1, I_2, m_2, \ldots, I_n, m_n$	v is a vector register; p, I_i, and m_i are as for vgather	Scatter a vector register into an array slice
aligned	Attribute that may be applied to load, store, vgather, and vscatter	Tells the translator that a load or store may be translated directly, without additional analysis or runtime alignment checks, on an architecture that has alignment requirements
Arithmetic, Logic, Compare, and Select		
add v_1, v_2	v_1 and v_2 are vectors	Add the elements of two vectors
sub, mul, div, rem, and, or, xor, vseteq, vsetne, vsetle, vsetge, vsetlt, vsetgt	Same as add	Perform an arithmetic or comparison operation on two vectors
adds, subs	Same as add	Saturated add or subtract
sat v, t	v is an integral vector; t is a scalar type of smaller precision than the element type of v	Saturate each element of v to type t
shl v, a shr v, a	v is a vector; a is an unsigned byte	Shift each element of v left or right by a bits
shrrnd v, a	v is a vector; a is an unsigned byte	Do a rounded shift right of each element of v by a bits
vselect b, v_1, v_2	b is a vector of booleans; v_1 and v_2 are vectors of the same type	Form a new vector by selecting an element from v_1 where b is true and from v_2 where b is false
sign v	v is a vector	Form the vector consisting of -1 in places where v is less than zero, 0 in positions where v equals zero, and 1 in positions where v is greater than zero
max v_1, v_2 min v_1, v_2	v_1 and v_2 are vectors	Form the vector consisting of the maximum or minimum element value of v_1 and v_2 at each position
Data Movement Operations		
extract v, l, s, n	v is a vector, l is a lower bound, s is a stride, and n is a length	Form a new vector of length n by extracting n elements of v starting at l with stride s
extractelement v, i	v is a vector; i is an index	Extract element i of vector v. Similar to extract $v, i, 1, 1$, except that the result is a scalar value rather than a vector
combine v_1, v_2, l, s	v_1 and v_2 are vectors, l is a lower bound, and s is a stride	Form a new vector by replacing elements $l, l+s, \ldots, l+s(n-1)$ of v_1 with the corresponding elements of v_2, where n is the length of v_2
combineelement v, e, i	v is a vector, e is a scalar element, and i is an index	Form a new vector by replacing element i of v with e. Similar to combine $v_1, v_2, i, 1$, except that e is a scalar element rather than a vector
permute v, i	v is a vector; i is an index vector	Produce a new vector with the elements of v permuted according to i

Note that this indexing scheme is very flexible. It allows row or column access on row-major or column-major arrays, by interchanging the multipliers. Because the multipliers need not be compile-time constants, the array extents need not be statically known. Also, by using a zero multiplier, one can generate repeated values. The syntax of `vscatter` is identical, except that `vscatter` requires an additional operand (the vector value to be stored) and produces no value.

The semantics of `vgather` and `vscatter` are different from `load` and `store`. `vgather` and `vscatter` guarantee only that the loads and stores defined above occur, and that the loaded and stored values are placed into the destination (register or memory) in the order specified above. The translator is free to schedule the load of any component value of a vector at any point in the program from a `vgather` to the first use of that value. Similarly, the translator may schedule a store at any point from the definition of the value to the `vscatter` of that value.

These rules are important because the translator often needs to interleave loads and stores. For example, if the program is written using long vectors, and the translator is compiling to a short-vector or scalar architecture, the translator must strip-mine the vector computations on the shorter length. Even where the hardware allows for long vectors (such as RSVP), long vector loads will usually be pipelined with other computations. With stricter semantics (e.g., that of Fortran 90, which provides that the program must behave as if all loads in a single statement occur before any stores in the same statement), the translator must do additional, often complex, analysis to validate the strip-mining, scalarization, or pipelining, in effect repeating the analysis required to vectorize the program in the first place [2]. Our semantics allows the translator to avoid this additional analysis.

Note that the burden is on the programmer or high-level compiler to write the Vector LLVA program so that these rules produce correct results. For example, in the well-known case `X[1:N] = X[0:N-1]`, it would *not* be correct simply to write a `vgather` followed by a `vscatter`. One way to write the program correctly is to put an explicit reversed loop (using `extractelement` and `combineelement`) between the `vgather` and `vscatter` (see Section 3.1.1 below for an example of such a loop). This loop would tell the translator that any blocking or strip-mining of the loads and stores must be done with a reversed loop. We envision that `vgather` and `vscatter` would be used for long vectors and vectors whose length is unknown at compile time (i.e., vector accesses that must be strip-mined or scalarized by the translator), while vector loads and stores would be used for accesses to short fixed-length vectors supported by a particular target.

2.3 Arithmetic, Logic, Compare, and Select

Vector LLVA provides the following operations on vectors:

- *Arithmetic, logical, and bit shift operations.* These LLVA operations may be applied directly to a single vector operand (bit shift) or a pair of vector operands of the same type.

- *Vector comparison operations.* Vector LLVA extends the existing LLVA comparison operations with new instructions that take two vector values and produce a vector of booleans.
- *Vector select.* Vector LLVA adds an operation `vselect` that takes a vector of booleans as the predicate and performs the corresponding selection at each position from the input vectors. We chose the select operation, rather than a mask operation as found on some vector architectures, because LLVA already supports this operation and because it seems to be more common on multimedia vector architectures. If true masking is needed (e.g., to hide a divide by zero that would otherwise cause an exception), it can be simulated by introducing "safe" values (e.g., divide by one instead of zero) and one or more additional select operations.

In all cases the operands must have the same length, or the results are unspecified. In keeping with our goal of designing a low-level virtual architecture for high performance, we chose not to include length checks (which would incur some performance overhead) as part of the semantics. It is possible to express such checks in Vector LLVA, and a programmer or compiler can always add them if necessary. This is similar to how array bounds checks are handled in machine code (and in LLVA).

The vector compare operations in Vector LLVA translate directly to the vector compare operations provided by RSVP, AltiVec, and SSE. `vselect` translates directly to the elementwise select operations available in AltiVec (vector select), SSE (synthesized with logical operations), and RSVP (`vif`).

We also extend the Vector LLVA instruction set to provide primitive multimedia operations found in media processing instruction sets such as AltiVec, SSE, and RSVP. While we have not attempted to produce a comprehensive list of multimedia operations, we have developed the following criteria for adding a new operation:

- *Is it straightforward to express the operation in terms of more primitive instructions?* If the answer was "Yes," we avoided a new instruction to keep the representation simple. An important question here is whether a pattern matching code generator can recognize the pattern reliably and generate the correct instruction or instruction sequence. Where this is not the case (e.g., with saturation, which can be expressed in multiple ways), we provided a new instruction.
- *Is the operation used across various instruction sets?* For some operations (max/min and saturation are important examples), the answer was yes. For more exotic cases (such as AltiVec `mradds`, which multiplies two vectors, does a rounded shift right *by 15*, and does a saturated add of the result with another vector), the answer was no. We decided to express these computations in terms of more primitive, widely used instructions, and rely on the code generator to exploit available features of the hardware.

If a programmer finds that Vector LLVA needs further extension for a particular architecture or application, such an extension is straightforward to do. LLVA already supports the concept of an *intrinsic*, which is an operation (expressed as a function call) that is lowered to a specified sequence of instructions for each supported architecture. Given a particular programming idiom for a particular architecture, a programmer could encapsulate that idiom as an intrinsic translated to a special implementation on that architecture while being lowered to an ordinary Vector LLVA code sequence (which is translated as usual) on other architectures.

2.4 Data Movement Operations

Data movement operations are an important feature of many vector architectures. For example, on AltiVec or SSE, data may be loaded only at stride one, so strided access must be done with permute or shuffle operations. Similarly, when the size of a computed value changes (as in promoting a short to an int), a pack or unpack instruction is needed to preserve the overall number of bits in the vector. Another reason for data movement is the alignment requirement. When a data access is unaligned, the resulting vector must be shifted to ensure that the computation is correct.

RSVP is much more flexible: it provides variable vector lengths, allows operations of different precisions on the same vector, provides direct strided access to memory, and has no alignment requirement. However, data movement instructions are still useful. For example, access to interleaved data (e.g., triples of RGB data) can be done via several strided data accesses, or it can be done with a single stride-one memory access followed by several strided accesses to the resulting vector. The second method saves input stream units (a key limited resource on RSVP) and may be more efficient in some cases.

To express vector data movement, Vector LLVA adds five new instructions: `extract`, `extractelement`, `combine`, `combineelement`, and `permute`. `extract` extracts a strided subvector of a vector register, the type of which is also a vector, while `extractelement` extracts a single indexed element (of scalar type) of a vector. `combine` overlays a smaller vector on top of a larger one, spaced at a specified stride. `combineelement` overlays a single element (of scalar type) on top of a vector at a specified index. In both cases, the result is assigned to a new vector register, to preserve the SSA property. `permute` takes a vector and an index vector; each position in the index vector specifies where in the input vector to get the value for that position in the output vector. In Section 2 below, we discuss in more detail how these instructions can represent the data movement operations found in RSVP, AltiVec, and SSE.

2.5 Alignment

Many vector architectures in use today have data alignment requirements. For example, on AltiVec, pointers used in vector load and store instructions must be aligned on 16 bytes (i.e., the last four bits of the address must be zero). If a pointer is unaligned, the last four bits of the address are dropped. In the absence of an alignment guarantee, the translator for such an architecture must insert runtime alignment checks and data shuffling instructions, which can be very inefficient. SSE does provide for unaligned loads and stores, but they are less efficient than aligned loads and stores.

At the application level, much more efficient options are possible. The programmer can control alignment in many ways, e.g., via carefully chosen aligned object allocation (usually supported by memory allocators or linkers), via loop restructuring, or via algorithmic changes. The compiler could automatically ensure aligned references in some cases, listed below, but only by using much more sophisticated analysis than we want the translator to do. For these reasons, the goals of our design are: (1) to enable the programmer or external compiler to express available knowledge of alignment; and (2) to enable the translator to generate aligned loads or stores without complex program analysis.

To meet these goals, we provide an `aligned` attribute that may be applied to vector loads and stores and to `vgather` and `vscatter` operations. `aligned` guarantees that the load or store always occurs with an aligned pointer. The alignment boundary is *not* encoded in this attribute: the value of this parameter is typically exploited at other places in the algorithm (e.g., allocation, loop bounds, or data shuffling).

To convert load and store operations to aligned load and store operations, the programmer or high-level compiler would use some combination of the following techniques:

- Use non-automatic techniques such as those listed above to *design* the kernel for aligned memory accesses.
- Use aligned object allocation in simple cases that can be handled automatically.
- Use interprocedural alignment analysis [10, 26] and/or program transformations [17] to ensure aligned accesses;
- Where an access would be unaligned, use an aligned access followed by data movement (`extract` and `combine`) operations, e.g., as described in [10].

Where all of these techniques fail, the operation can be left as an unmarked (i.e., possibly unaligned) load or store, forcing the translator to insert a runtime alignment check at some cost to efficiency.

3. Evaluation

In this section, we present the results of an evaluation of Vector LLVA that we carried out with three goals in mind:

1. To determine whether Vector LLVA can be used to write code for a range of vector architectures. Within a family of related architectures, we want to write code that is as portable as possible.
2. To determine whether Vector LLVA can be translated to native code for several architectures using a simple translator for each one.
3. To determine whether handwritten Vector LLVA can be translated with performance competitive with handwritten native code.

To perform this evaluation, we used Vector LLVA to write programs and generate code for three architectures: the Motorola Reconfigurable Streaming Vector Processor (RSVP) architecture [8], AltiVec [9], and SSE2 [5]. For the AltiVec and SSE2 programs, we compiled a *single* Vector LLVA program to *both* AltiVec and SSE2. We hand-tuned the Vector LLVA code for AltiVec/SSE2, but not specifically for either one. For the RSVP programs, we hand-tuned the Vector LLVA code for RSVP. In all cases, we compared the generated code with handcoded native versions of the same benchmarks.

It was *not* a goal of the present work to show hand-coded performance from a single program representation across both RSVP and AltiVec/SSE2. These architectures have significant differences, and we believe it is very difficult, if not impossible, to achieve near-handcoded performance with a single representation across these architectures. Several researchers are currently working on the problem of efficiently compiling long vectors to short-vector architectures with data access and alignment constraints [22, 27, 26, 14]. This research will likely make it possible for Vector LLVA written with long vectors to be portable to AltiVec, SSE2, and other architectures with moderately good performance, but achieving near-handcoded performance with this approach is an open question.

For the evaluation below, we wrote Vector LLVA code "by hand." To simplify this task, we have developed a C API providing intrinsics for the Vector LLVA instructions, similar to the APIs available for programming AltiVec and SSE2.

3.1 Suitability for RSVP

RSVP [8] is a vector coprocessor architecture that accelerates vectorizable computations on arbitrary length vectors (called *streams*); a host processor performs the other computations. The coprocessor and host share main memory (and the implementations we worked with share the entire cache hierarchy). RSVP defines a dataflow graph (DFG) language for writing coprocessor code. Each DFG describes the body of a single basic-block loop; typically, each iteration of the loop body reads elements of one or more input vectors, performs some computations described by the graph, and writes back elements of one or more output vectors. The number of iterations, the descriptions of the input and output streams, and any needed scalar values are specified as inputs to the DFG by the host. Each stream is described as a rectangular subsection of an array, specified as a *stride*, *span*, and *skip* triple for each dimension. The architecture provides unlimited vector registers (DFG nodes), but a limited number of scalar registers, "tunnels", and "accumulators" (used to express loop-carried dependences). All dependences, including loop-carried dependences (which can only cross one iteration), are explicitly stated. Any conditional code must be converted to use conditional select operations or must be executed on the host.

The hardware implementations of RSVP use VLIW cores with 20 units, and a rich network for interconnecting the units. A software scheduler maps the DFG operations onto the RSVP hardware using aggressive pipelining of data parallel loops. A programmer-visible local memory, called the tile buffer, is used to buffer and reuse stream data values. The latest RSVP implementation (version 2) provides three input streams and one output stream.

3.1.1 Writing Vector LLVA for RSVP

Because RSVP uses a loop-based idiom to express vector parallelism, the most natural way to write Vector LLVA code for RSVP is as a loop operating sequentially on vectors with the following properties (we refer to this as a "vector loop"):

1. The loop consists of a single basic block ending in a backward branch to the head.
2. No memory operations (vector or scalar) appear in the loop body. In particular, all `vgather` and load operations appear before the loop, and all `vscatter` and store operations appear after the loop.
3. All operations inside the loop are *scalar* operations; elements are extracted from vectors (using `extractelement`) before they are used in the loop and inserted into vectors (using `combineelement`) that are stored or used after the loop.

Figure 1 illustrates this "vector loop" form for the kernel SAD (sum of absolute differences), which is given by `z[0:n-1] = sumreduce(abs(x[0:n-1]-y[0:n-1]))`.

```
ENTRY:
        T0 = add n, -1
        T1 = vgather x, 0, T0, 1, 1
        T2 = vgather y, 0, T0, 1, 1
        T4 = seteq n, 0
        br T4, EXIT, LOOP

LOOP:
        T5 = phi (0, ENTRY), (T12, LOOP)
        T6 = phi (0, ENTRY), (T11, LOOP)
        T7 = extractelement T1, T5
        T8 = extractelement T2, T5
        T9 = sub T7, T8
        T10 = abs T9
        T11 = add T10, T6
        T12 = add T5, 1
        T13 = setlt T12, n
        br T13, LOOP, EXIT

EXIT:
        T14 = phi (0, ENTRY), (T11, LOOP)
        vscatter T14, z, 0, T0, 1, 1
```

Figure 1. SAD Benchmark

This "vector loop" idiom expresses the full range of RSVP constructs while exposing vector parallelism to the translator. For example, checking index expressions is much simpler than arbitrary dependence analysis, because any complicated memory access patterns have been factored out of the loop and put into `vgather` and `vscatter` operations. Inside the loop, the translator need identify only the relatively simple "streaming" (i.e., successive access) patterns supported by RSVP. Further, as in the RSVP DFG language itself, all dependences are explicitly stated, facilitating compiler analysis. (Loop-independent dependences become def-use chains, while loop-carried dependences are expressed as phi nodes.)

For kernels that do not use constructs that require such explicit loops (such as loop-carried dependences and conditional stores), we can also write Vector LLVA in a "pure" vector form that uses no explicit loops: vector values are loaded into vector registers, computed, and stored. This form of Vector LLVA is compact and easy to analyze. It is also a higher-level idiom than the vector loop form that could be easier to translate to other architectures as well. For kernels that can be written this way, it is straightforward to lower this representation to the vector loop idiom and then generate RSVP code. To avoid complex loop fusion requirements in the translator we impose the requirement that pure vector and vector loop forms cannot be mixed in the same code sequence (more precisely, they will be translated into separate vector kernels).

We considered including higher-level abstractions, such as map-reduce, in our Vector LLVA representation, but decided against it. LLVA is deliberately a *low level* representation. It aims to encode enough information to enable sophisticated compiler analysis and transformation, while remaining close enough to the target to allow the programmer or compiler careful control over performance characteristics of the code. We believe that higher-level constructs, such as the kernel functions and streaming loops of StreamC and KernelC [23] can be mapped straightforwardly to Vector LLVA.

3.1.2 Generating RSVP Code

Generating RSVP code from Vector LLVA involves the following steps:

1. Identify and group the sets of vector instructions that will become distinct RSVP DFGs.
2. Lower each distinct group of pure vector instructions to "vector loop" form, forming a single "vector loop."
3. For each "vector loop" generated in the previous step, map the computations to the software-exposed resources of RSVP (streams, scalars, accumulators and tunnels) and generate a DFG.
4. For the non-vectorizable code, generate code for the host processor, including the RSVP library calls to set up and execute the DFG.

In step one, for kernels already written in "vector loop" form, we use the grouping implied by the loop. For kernels written as a sequence of vector statements, we use a partitioning algorithm similar to the one used for scalar renaming [2, p. 197]. The algorithm forms the largest set of vector instructions that can legally be transformed to an RSVP DFG (making the DFG as large as possible is a good heuristic in most cases). It does this by following def-use chains upwards until a `vgather` or `vimm`, and downwards until a `vscatter`. This grouping process must satisfy the same requirements as loop fusion, i.e., the loop bounds of successive vector instructions (treating each as a distinct loop) must match. This property does not need to be checked, however, because the vector lengths in a def-use chain must be consistent to ensure well-defined semantics (see Section 2.3). Note that mixing vector loop code with pure vector code would break this property, which is why we do not mix the two, as noted earlier.

Our implementation also identifies uses of input and output streams and scalar registers and maps them to the corresponding hardware resources. Tunnel and accumulator operations are easily recognized by the presence of a phi node and the pattern of uses. Currently, we have implemented this algorithm only for the loop-free Vector LLVA idiom; for the "vector loop" idiom, we do the mapping by hand.

Once the Vector LLVA has been partitioned and mapped to the RSVP resources, translating the individual operations into DFG operations is straightforward. The `vgather` and `vscatter` instructions are mapped to the input and output vector stream units (VSUs). Tunnel and accumulator operations have already been identified in the partition/mapping step, described above. Other operations are directly mapped to distinct DFG nodes; in most cases this is a one-to-one mapping. In some cases we apply pattern matching (e.g., we code generate a division by a constant power of two as an RSVP `div2n`).

3.1.3 Results

We tested our representation and code generator using several C kernels from the audio and image processing application areas. We identified computation-intensive vectorizable loops and manually rewrote them using our C language extensions for writing Vector LLVA. We tested the following kernels:

- **Saxpy**: Multiplying a vector by a scalar and adding it to another vector.
- **Quant**: Vector quantization from the H.263 video coding standard.
- **MAD Filter**: A filter for enhancing stereo separation from the MAD benchmark in the MediaBench suite.
- **RGB2YUV**: Conversion from RGB color representation to YUV chrominance and luminance representation.
- **Transpose**: Eight by eight matrix transpose.
- **DCT**: Discrete cosine transform (Chen's algorithm) for video encoding.
- **SAD**: sum of absolute differences between two vectors, used in motion estimation for video encoding

In each case, we vectorized a single scalar loop from the original benchmark. We wrote the first five benchmarks in the "pure" vector idiom (with no LLVA loops). We wrote SAD in the "vector loop" idiom, as discussed above.

We are only able to experiment with small kernels such as the above because we have to manually write two or three native versions plus two Vector LLVA versions of each one, and tune each of them for the experiments. Nevertheless, we believe such kernels are acceptable here for at least two reasons. First, it is well known that such kernels dominate the computations of many media processing applications (e.g., SAD and DCT account for over 50% of the scalar execution cycles of Motorola's MPEG 4 encoder benchmark for RSVP). Second, because media processing is heavily driven by industry-wide standards like MPEG, JPEG, TIFF, etc., the same well-defined kernels tend to appear repeatedly in multimedia codes. For these reasons, we expect multimedia programmers to identify and isolate computations into well-tuned kernels, as we did in this work.

We ran the Vector LLVA code through our RSVP code generator, generating C code for the host (an ARM9 processor) and DFG code for RSVP. Of the four codes, MAD Filter required three DFGs because of the patterns of loads and stores (it is possible to implement MAD Filter with a single DFG, but doing so requires

interleaving the loads and stores in a way that actually reduces performance). The other benchmarks required one DFG each.

We ran the generated code (host code and DFGs) on a cycle-level simulator from Motorola that models both the ARM processor and the RSVP coprocessor [8]. The simulator reports total cycle counts for the ARM processor, including cycles spent in computation and cycles spent setting up and waiting for the RSVP coprocessor. We measured the speedup achieved when running the ARM/RSVP version of the loop compared with running the entire scalar loop on the ARM host. This is the same metric as used in [8], since RSVP is a coprocessor and has no "single-processor" version to use as a baseline.

Figure 2 shows the speedups that our code generator achieved and compares them to the speedups achieved by handwritten RSVP DFGs for a range of vector values. (We used the same host code for both versions). As expected, the vector performance consistently improves with increasing vector length. For the handcoded DFGs, we attempted to use the most efficient way of coding the DFGs. For Saxpy, Quant, RGB2YUV, DCT, and SAD, we used DFGs written and tuned by the authors of [8].

The Vector LLVA version was very close to, or exceeded, the performance of the handcoded version in all cases. For Saxpy and DCT, the Vector LLVA performance was slightly better because the compiler-generated DFG specified the precision for all operations, saving some execution cycles. For Quant, the Vector LLVA version was significantly better because our LLVA compiler transformations were able to propagate a constant from the surrounding C program into the Vector LLVA code, so that the value became a constant (`vconst`) instead of a variable (`vscalar`) in the resulting DFG. An optimization like this would be useful if, for example, a kernel is tuned once by hand and used in different caller contexts, where the compiler can apply different optimizations (inlining, constant propagation, etc.) appropriate to the different contexts. We could have performed this constant propagation by hand on the RSVP DFG, but because the C compiler is not aware of the RSVP semantics, automatic constant propagation from the C code to the RSVP code is not possible. This optimization illustrates a potential benefit of writing the scalar C code and the vector kernel in a single language (Vector LLVA).

3.2 Suitability for AltiVec and SSE2

We have also written two code generators that translate Vector LLVA to code for either AltiVec or SSE2. Our aim is to enable programmers to write one Vector LLVA program for either of these targets (and other similar ones). AltiVec and SSE2 are similar in that they both operate on 128-bit wide registers with the same data types, have many overlapping operations, and impose similar constraints on alignment and data movement. However, there are important differences between the two. Some operations are supported on one architecture but not the other. They use different idioms in some cases, e.g., AltiVec has a select operation, while SSE2 synthesizes this operation with bitwise logical operations.

3.2.1 Writing Vector LLVA for AltiVec and SSE2

In targeting AltiVec and SSE2, we hand wrote Vector LLVA in a form tailored to these targets. Our AltiVec/SSE2 form of Vector LLVA has the following salient features:

- All vectors are of fixed length (128 bits). Longer vector operations are handled with loops blocked on the vector length. We do this blocking by hand for now. In many cases, it would be straightforward to have the translator block loops automatically. However, because of the vector size and memory access constraints imposed by AltiVec and SSE, automatic generation of optimal code from the long vector form is an open research problem.
- We use `load` and `store` operations, instead of `vgather` and `vscatter`, to reflect that AltiVec and SSE2 loads and stores are at stride one and do not have asynchronous semantics.
- We ensure alignment of allocated vectors and use the `aligned` attribute to indicate aligned loads and stores.
- We write explicit data movement and shuffling operations using the Vector LLVA operations discussed in Section 2.4, both for promoting and demoting data types, and for performing strided memory access.

Figure 3 illustrates our AltiVec/SSE2 form of Vector LLVA in schematic form for the Quant benchmark (`getelementptr` is an LLVA operation for computing the address of an element in an array or struct [1]). Note that expressing data movement operations is straightforward. This example shows a pack of two vectors of short into a vector of char (the actual quantization computation is omitted). Unpacking operations are similar and use `extract`. We can also use `extract` to express permute/shuffle operations with a fixed stride and `permute` to express arbitrary permute/shuffle operations. Note also that the loop has been unrolled to two iterations to support the pack operation. This shows how Vector LLVA can be hand-tuned for fixed-length vector architectures, while retaining portability across examples of such architectures. A loop like this could be generated automatically from the long vector representation by a compiler, but to achieve optimal performance the compiler would need to be sophisticated enough to generate this unrolling and packing pattern.

```
T0  = getelementptr in, 2*i
T1  = aligned load T0 ; short
T2  = ... ; quantize T1 to char
T3  = getelementptr in, 2*i+1
T4  = aligned load T3
T5  = ... ; quantize T4 to char
T6  = vimm char 0, 16
T7  = combine T6, T2, 0, 1
T8  = combine T7, T5, 8, 1
T9 = getelementptr out, i
aligned store T8, T9
```

Figure 3. Quant Benchmark for AltiVec and SSE2

3.2.2 Generating AltiVec and SSE2 Code

We wrote two simple pattern-matching translators to take the same Vector LLVA representation, described above, to AltiVec and SSE2 Code. To keep the implementation simple, we generated C, augmented with the source-level compiler intrinsics for AltiVec (supported by gcc) and SSE2 (supported by the Intel C compiler as well as gcc).

In many cases, the Vector LLVA to AltiVec or SSE2 translation is one-to-one. In the following cases, we matched patterns of several Vector LLVA instructions and converted them to a single AltiVec or SSE2 instruction:

- Where an instruction is important for performance on the target architecture, but is too specialized to warrant a separate Vector LLVA instruction (e.g., the AltiVec `mradds`).
- For packing, unpacking, and shuffling operations. For these operations, we used sequences of the more general Vector LLVA data movement instructions, as discussed above.

In some cases, we generated a sequence of instructions from a single Vector LLVA instruction. This is particularly true on SSE2, where some basic operations (e.g., vector select and packing without saturation) are not directly supported and must be synthesized. We added new intrinsics for saturation and max/min operations.

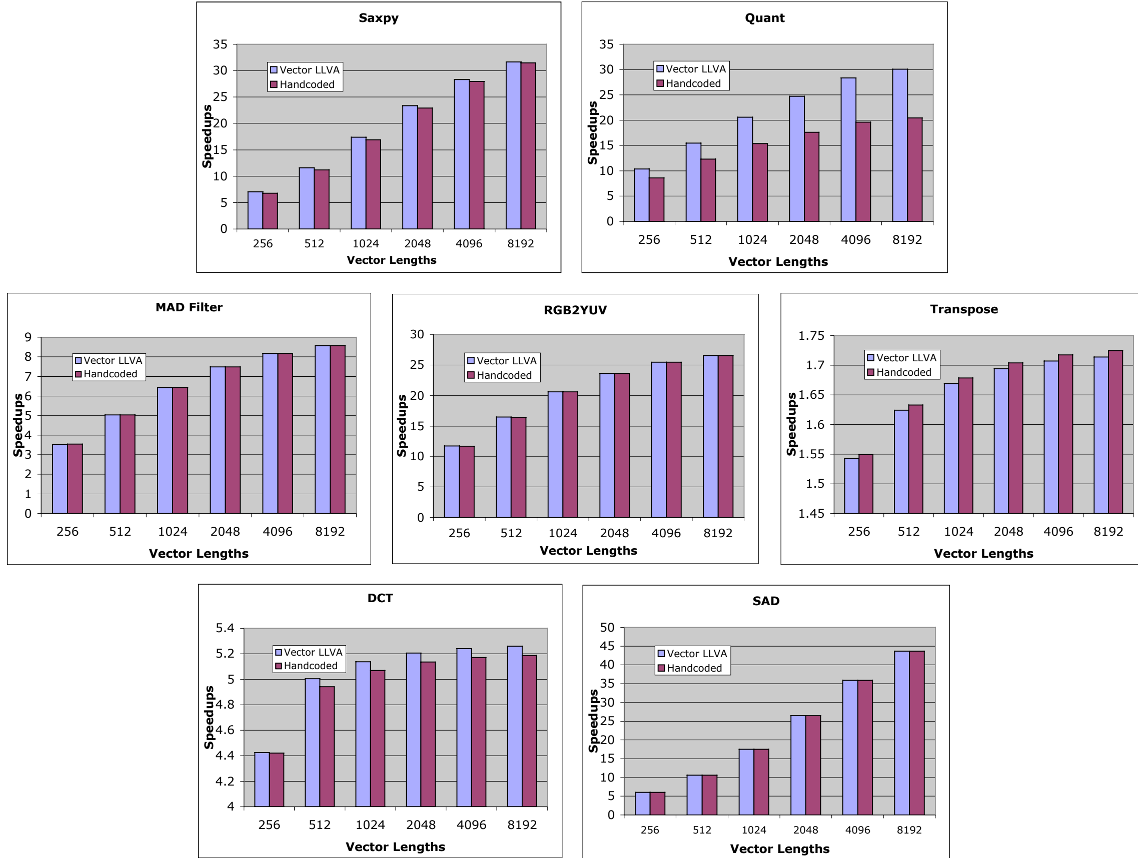

Figure 2. Speedups on RSVP

3.2.3 Results

We ran our translators on the same list of benchmarks given in Section 3.1.3 above, with two changes. First, we ran RGB2YUV only on AltiVec, because we did not see a way to hand code this benchmark on SSE2 with reasonable efficiency given the limitations on SSE2's shuffling operations. Second, we did not run SAD on AltiVec or SSE because SAD is a special instruction on SSE2 and it cannot be expressed in terms of more primitive operations (because SSE2 lacks instructions to sum-reduce a vector). Therefore, SAD is probably best dealt with as a special case and made into an intrinsic that is translated to the appropriate code on AltiVec (using more primitive instructions) or SSE2 (using the SAD instruction). For Quant, RGB2YUV, Transpose, and DCT, we used the hand-coded AltiVec versions available from Freescale. To obtain hand-coded SSE2 versions, we hand-translated them following Apple Computer's guidelines for AltiVec to SSE2 translation [3]. We wrote the handcoded versions of Saxpy and MAD Filter for both AltiVec and SSE2.

To generate the handcoded performance numbers for AltiVec, we put our handwritten C code with AltiVec intrinsics through gcc with AltiVec intrinsics enabled. For SSE2, we did the same thing using the Intel SSE2 intrinsics and the freely available, non-commercial version of the Intel C compiler (icc). To generate the Vector LLVA numbers, we first translated the Vector LLVA to C with intrinsics, as described in Section 3.2.2 above, and then put the result through gcc (for AltiVec) or icc (for SSE2).

Tables 2 and 3 give performance numbers and ratios comparing the handcoded kernels to Vector LLVA for both scalar and vector

Table 2. AltiVec Performance Results (in milliseconds except ratios)

	Scalar			Vector		
	Hand	LLVA	Ratio	Hand	LLVA	Ratio
Saxpy	3410	4350	0.78	300	300	1.0
Quant	1350	1430	0.94	100	80	1.25
MAD Filter	4450	4910	0.91	390	410	0.95
RGB2YUV	9700	15560	0.62	230	240	0.96
Transpose	800	1810	0.44	300	250	1.2
DCT	2920	5180	0.56	160	170	0.94

performance. The performance numbers are the user times reported by the POSIX utility `times`. These times include cycles consumed in scalar operations (such as surrounding loops). We show the scalar comparison to illustrate the differences in the base back end infrastructures. With the Intel compiler, the scalar C code compiled to X86 performed nearly identically to the code that we first put through our C back end for LLVA and then compiled to X86. For PowerPC using gcc, however, the scalar results were quite different in some cases. This makes it difficult to compare the vector numbers for PowerPC/AltiVec. Regardless of the difference in scalar numbers, however, the vector ratios are encouragingly close. On AltiVec, Vector LLVA performance ranges from 6% worse (DCT) to 25% better (Quant) than handcoded. On SSE, Vector LLVA performance ranges from 8% worse (DCT) to 21% better (Transpose) than handcoded.

A pertinent question is whether an auto-vectorizing compiler could achieve some of the portability benefits of Vector LLVA by

Table 3. SSE2 Performance Results (in milliseconds except ratios)

	Scalar			Vector		
	Hand	LLVA	Ratio	Hand	LLVA	Ratio
Saxpy	2020	2000	1.0	180	180	1.0
Quant	1100	1090	1.01	100	100	1.0
MAD Filter	2390	2390	1.0	340	350	0.97
Transpose	610	610	1.0	350	290	1.21
DCT	2220	2200	1.01	390	420	0.92

automatically compiling a common, higher-level program down to vector hardware (instead of our approach of using lower-level code tuned for a generic, fixed-length SIMD architecture). We put scalar versions of our benchmarks (with dependence hints) through the autovectorizer included with the Intel compiler, the best autovectorizer for SIMD available to us. For the simplest benchmarks, Saxpy and MAD Filter, the autovectorizer was able to generate code that performed just about as well as the handcoded and Vector LLVA versions: 220 for Saxpy and 340 for MAD Filter (these are not shown in the tables). However, for the other three benchmarks, the Intel autovectorizer was unable to produce *any* speedups, even with dependence hints. These three benchmarks require more complicated patterns of instructions, such as packing, unpacking, shuffling, and loop unrolling, to obtain good performance. While the Intel compiler was not sophisticated enough to recognize these patterns automatically from the scalar code, we could express these patterns directly in Vector LLVA, obtaining code with essentially handcoded performance that was also portable to AltiVec.

4. Related Work

There has been a recent surge of work on compiling to vector processors, particularly subword SIMD architectures such as AltiVec and SSE. Much of this work has focused on vectorizing or "SIMDizing" scalar source [16, 4, 24, 22, 27]. Others have focused on generating efficient code subject to the alignment, vector size, and data movement constraints of subword SIMD architectures [17, 26, 14]. There has been some work on portable vector programming using explicit vectors [7, 11]. We are unaware of any work that proposes a *virtual vector instruction set*, and we believe our work is unique in this regard.

Wu et al. [27] describe a framework for compiling scalar C source to AltiVec. Their compiler internally abstracts physical vectors into "virtual vectors," and they describe techniques for lowering the virtual vectors to actual vectors satisfying the requirements of AltiVec, including fixed length and alignment. In contrast to our work, Wu et al. do not address the problem of achieving near-handcoded performance on multiple different architectures. They do not discuss compilation to arbitrary-length vector or streaming architectures such as RSVP; in fact, the lengths of their virtual vectors must be compile-time constants. Finally, their virtual vectors are used only within a compiler intermediate representation, rather than a virtual instruction set. This is primarily a difference in goals: two of our goals have been to give programmers a more uniform architectural model for multiple families of vector architectures, and to provide virtual object code portability across these families.

The Stream Virtual Machine of Labonte et al. [15] has similar goals to ours, but the two designs emphasize distinct and complementary issues. We focus on expressing vector parallelism mapped to a single vector processor, whereas Labonte et al. use the native instruction set (directly or through a C API) for individual processing cores. Their design focuses on modeling the communication among multiple different components of a streaming architecture. They provide a parametric model for describing such an architecture and show that the model can accurately describe several actual architectures. We believe that in the long term, it would be valuable to combine Labonte et al.'s model for expressing communication among multiple SIMD cores and memories with our virtual instruction set for expressing vector operations in a single core.

VCODE [6] is an intermediate-level language designed for use in studying the compilation of data-parallel languages to large-scale parallel architectures. Like Vector LLVA, VCODE is intended to be a portable representation and supports arbitrary-length vectors. However, VCODE is not designed, as Vector LLVA is, to express hand-tuned high performance code on streaming and subword SIMD architectures. This leads to several significant design differences. For example, VCODE is more abstract than Vector LLVA: it is a stack machine, it has no scalar type and no explicit loops (only recursion is allowed), and it allows vectors to be partitioned into *segments* for applications such as parsing and DNA sequencing. VCODE also contains no explicit support for multimedia operations like saturated arithmetic, gather, and scatter.

A few experimental, domain-specific languages like StreamC and KernelC for the Imagine stream processor [20, 13], StreamIt [25], and SPIRAL [28] have been proposed for media processing and signal processing. These languages provide high-level features such as streams or tensors appropriate for the application domain, whereas we provide low-level primitives appropriate for capturing (in somewhat abstract terms) the important features of media processing hardware. Compilers for these languages could benefit by targeting Vector LLVA, allowing them to focus on high-level semantics and optimization, leaving machine-level optimization and code generation to the translator.

5. Summary and Future Work

We have proposed a processor-level virtual instruction set called Vector LLVA, designed to encode explicit vector computations for media processing hardware. The instruction set aims to abstract the important architectural features and operations of hardware processors, while allowing programmers to write carefully tuned code manually for a particular processor family. We have implemented translators from Vector LLVA for three processor architectures, in two processor families: the Motorola RSVP processor, and AltiVec and SSE in the subword SIMD family. Our experience with writing benchmarks in Vector LLVA by hand shows that it is possible to tune code carefully for each of the families and achieve code competitive with hand-coded native assembly on each hardware architecture, using a single instruction set.

In the near term, we aim to make two improvements in our work. First, our AltiVec and SSE translators must be enhanced to exploit some special features of the hardware such as specialized data movement operations and complex instructions like the SAD operation on SSE. Second, we would like to continue to port other benchmarks and applications to Vector LLVA in order to evaluate our translators more extensively, and enhance them where needed.

In the longer term, there are three directions that would help realize the potential benefits of this work. First, we believe we can design source-level language extensions (e.g., to C) that make it simpler to write application programs for the Vector LLVA model while achieving high performance *within* each processor family. Second, we would like to demonstrate that high-performance *libraries* can be written in a combination of source code and hand-coded Vector LLVA code, while achieving the exacting performance goals that library writers demand in this domain. A key question here will be how much separate tuning is needed for each target family, and to what extent that tuning can be made incremental because of the functional portability that Vector LLVA provides. Third, we would like to be able to extend our performance goals to achieve high performance from a *single* Vector LLVA program on multiple families of processors, including the long-vector and subword SIMD families discussed in this paper. This would allow Vector

LLVA to provide binary portability for a single object code program with variable vector lengths across all its target classes of processors. As discussed earlier, results from previous researchers on compiling long-vector computations to subword-SIMD hardware (e.g., Wu et al. [27]) show that such a representation would likely provide good performance, but achieving performance competitive with hand-coded programs is an open question.

Finally, Vector LLVA also provides additional long-term benefits (which are difficult to evaluate experimentally). First, Vector LLVA can abstract away evolutionary design differences between multiple generations of a single architecture, reducing the need to modify, debug and tune code as an architecture evolves. Second, it provides a single, rich vector programming model that can be used by programmers to design algorithms, and by system developers to target source-level compilers, debugging tools, and performance tools across a wide range of vector families. Although this benefit is difficult to evaluate experimentally, we believe this is a crucial goal for improving programmer productivity for media processing applications.

Acknowledgments

The authors wish to thank the engineers in the Systems Architecture Lab at Motorola, particularly Ray Essick, Phil May, Silviu Ciricescu, and Mike Schuette, for their assistance with RSVP compilation. We also wish to thank Chris Lattner for his insightful discussions regarding the design of Vector LLVA.

References

[1] V. Adve, C. Lattner, M. Brukman, A. Shukla, and B. Gaeke. LLVA: A Low-Level Virtual Instruction Set Architecture. In *Proc. ACM/IEEE Int'l Symp. on Microarchitecture (MICRO)*, pages 205–216, San Diego, CA, Dec. 2003.

[2] R. Allen and K. Kennedy. *Optimizing Compilers for Modern Architectures*. Morgan Kaufmann Publishers, Inc., San Francisco, CA, 2002.

[3] Apple Computer, Inc. AltiVec/SSE Migration Guide. http://developer.apple.com/documentation/Performance/VelocityEngine-date.html, 2005.

[4] L. Baumstark, Jr., and L. Wills. Exposing Data-Level Parallelism in Sequential Image Processing Algorithms. In *Proc. Working Conf. on Reverse Engineering (WCRE)*, 2002.

[5] A. J. Bik. *The Software Vectorization Handbook: Applying Multimedia Extensions for Maximum Performance*. Intel Press, 2004.

[6] G. E. Blelloch and S. Chatterjee. VCODE: A Data-Parallel Intermediate Language. In *Proc. Symp. on the Frontiers of Massively Parallel Computation*, pages 471–480, Oct. 1990.

[7] G. Cheong and M. Lam. An Optimizer for Multimedia Instruction Sets. In *Proc. Second SUIF Compiler Workshop*, 1997.

[8] S. Ciricescu, R. Essick, B. Lucas, P. May, K. Moat, J. Norris, M. Schuette, and A. Saidi. The Reconfigurable Streaming Vector Processor (RSVP). In *Proc. ACM/IEEE Int'l Symp. on Microarchitecture (MICRO)*. IEEE Computer Society, Dec. 2003.

[9] K. Diefendorff, P. K. Dubey, R. Hochsprung, and H. Scales. AltiVec Extension to PowerPC Accelerates Media Processing. In *Proc. ACM/IEEE Int'l Symp. on Microarchitecture (MICRO)*, 2000.

[10] A. Eichenberger, P. Wu, and K. O'Brien. Vectorization for SIMD Architectures with Alignment Constraints. In *Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, 2004.

[11] R. Fisher and H. Dietz. Compiling for SIMD Within a Register. In *Proc. Int'l Workshop on Languages and Compilers for Parallel Computing (LCPC)*, 1998.

[12] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy. Introduction to the Cell Multiprocessor. *IBM Journal of Research and Development*, 49(4/5):589–604, 2005.

[13] U. J. Kapasi, S. Rixner, W. J. Dally, B. Khailany, J. H. Ahn, P. Mattson, and J. D. Owens. Programmable Stream Processors. *IEEE Computer*, pages 54–62, Aug. 2003.

[14] A. Kudriavtsev and P. Kogge. Generation of Permutations for SIMD Processors. In *Conf. on Language, Compiler, and Tool Support for Embedded Systems (LCTES)*, 2005.

[15] F. Labonte, P. Mattson, I. Buck, C. Kozyrakis, and M. Horowitz. The Stream Virtual Machine. In *Proc. Int'l Conf. on Parallel Architectures and Compilation Techniques (PACT)*, 2004.

[16] S. Larsen and S. Amarasinghe. Exploiting Superword Level Parallelism with Multimedia Instruction Sets. In *Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, 2000.

[17] S. Larsen, E. Witchel, and S. Amarasinghe. Increasing and Detecting Memory Address Congruence. In *Proc. Int'l Conf. on Parallel Architectures and Compilation Techniques (PACT)*, 2002.

[18] C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation. In *Proc. Int'l Symp. on Code Generation and Optimization (CGO)*, San Jose, Mar 2004.

[19] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, Reading, MA, 1997.

[20] P. R. Mattson. *A Programming System for the Imagine Media Processor*. PhD thesis, Computer Science Dept., Stanford University, 2002.

[21] E. Meijer and J. Gough. A Technical Overview of the Common Language Infrastructure. http://research.microsoft.com/˜emeijer, 2002.

[22] G. Ren, P. Wu, and D. Padua. An Empirical Study on the Vectorization of Multimedia Applications for Multimedia Extensions. In *Proc. Int'l Parallel and Distributed Processing Symp.*, 2005.

[23] B. Serebrin, J. D. Owens, C. H. Chen, S. P. Crago, U. J. Kapasi, B. Khailany, P. Mattson, J. Namkoong, S. Rixner, and W. J. Dally. A Stream Processor Development Platform. In *Proc. Int'l Conf. on Computer Design (CDES)*, 2002.

[24] J. Shin, J. Chame, and M. Hall. Exploiting Superword-Level Locality in Multimedia Extension Architectures. *Journal of Instruction-Level Parallelism*, 31(5):1–28, 2003.

[25] W. Thies, M. Karczmarek, and S. Amarasinghe. StreamIt: A Language for Streaming Applications. In *Proc. Int'l Conf. on Compiler Construction (CC)*, 2002.

[26] P. Wu, A. Eichenberger, and A. Wang. Efficient SIMD Code Generation for Runtime Alignment and Length Conversion. In *Proc. Int'l Symp. on Code Generation and Optimization (CGO)*, 2005.

[27] P. Wu, A. Eichenberger, A. Wang, and P. Zhao. An Integrated Simdization Framework Using Virtual Vectors. In *Proc. Int'l Conf. on Supercomputing (ICS)*, 2005.

[28] J. Xiong, J. Johnson, R. Johnson, and D. Padua. SPL: A Language and Compiler for DSP Algorithms. In *Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, 2001.