2011 International Conference on Parallel Architectures and Compilation Techniques

StVEC: A Vector Instruction Extension for High Performance Stencil Computation

Naser Sedaghati, Renji Thomas, Louis-Noël Pouchet, Radu Teodorescu, P. Sadayappan
Department of Computer Science and Engineering
The Ohio State University
{sedaghat,thomasr,teodores,pouchet,saday}@cse.ohio-state.edu

***Abstract*—Stencil computations comprise the compute-intensive core of many scientific applications. The data access pattern of stencil computations often requires several adjacent data elements of arrays to be accessed in innermost parallel loops. Although such loops are vectorized by current compilers like GCC and ICC that target short-vector SIMD instruction sets, a number of redundant loads or additional intra-register data shuffle operations are required, reducing the achievable performance. Thus, even when all arrays are cache resident, the peak performance achieved with stencil computations is considerably lower than machine peak.**

In this paper, we present a hardware-based solution for this problem. We propose an extension to the standard addressing mode of vector floating-point instructions in ISAs such as SSE, AVX, VMX etc. We propose an extended mode of paired-register addressing and its hardware implementation, to overcome the performance limitation of current short-vector SIMD ISA's for stencil computations. Further, we present a code generation approach that can be used by a vectorizing compiler for processors with such an instructions set. Using an optimistic as well as a pessimistic emulation of the proposed instruction extension, we demonstrate the effectiveness of the proposed approach on top of SSE and AVX capable processors. We also synthesize parts of the proposed design using a 45nm CMOS library and show minimal impact on processor cycle time.

I. Introduction

Stencil computations arise in the core kernels of many scientific applications and their optimization has been the focus of several recent publications [1], [2], [3], [4], [5], [6]. Stencil codes are generally easily vectorized by compilers such as GCC and Intel's ICC because they typically feature parallel innermost loops where array elements are accessed at unit stride. However, as we illustrate using a simple example below, the realized performance often falls far short of machine peak, even when all accessed data is resident in the L1 cache. The main reason for the loss of performance is that when compiling stencil codes for current vector instruction architectures, it is necessity to use either redundant and unaligned load operations or intra-register shuffle or other intra-register data reorganizing operations. In this paper we propose a hardware based solution along with a compiler code generation approach to address the problem.

Consider the following loops S1, S2, and S3 (shown in Figure 1). The first loop S1 multiplies a scalar element K with each element of vector B, and add the result to an element of vector A with the same index, in order to produce vector result A. Assuming that the hardware vector size is 4 (as in SSE for float data), and that $A[0]$ and $B[0]$ are aligned to a boundary that is a multiple of the hardware vector size, the vector code generated by a compiler for S1, in every iteration of the outer loop t, will use $\frac{N}{4} - 1$ aligned vector load and store operations to read/write the elements of B and A — compute A[4*i:4] by loading B[4*i:4] (the notation A[i:V] denotes a vector of V consecutive elements of A starting at index i) and multiplying with a vector register containing four identical copies of the scalar K.

```
  for (t = 0; t < T; t++)
      for (i = 4; i < N; i++)
S1:     A[i] += B[i] * K;

  for (t = 0; t < T; t++)
      for (i = 4; i < N; i++)
S2:     A[i] += B[i-1] * K;

  for (t = 0; t < T; t++)
      for (i = 4; i < N; i++)
S3:     A[i] += B[i-1] * B[i];
```

Figure 1. Vector multiply-add loop with different multiply operands: aligned-constant (S1), unaligned-constant (S2), unaligned-aligned (S3).

Code	Nehalem (i7-920)	Sandy Bridge (i7-2600K)	Core2 Quad (Q6600)	Phenom (9850BE)
S1	4.10	11.36	3.70	3.84
S2	3.75	7.80	0.87	2.71
S3	2.83	6.51	0.83	2.27

Table I
PERFORMANCE (GFLOPS) FOR S1, S2 AND S3 ON DIFFERENT MACHINES FOR N=1024 AND T=500000.

With loop S2, the number of vector multiplication operations and the number of vector load/store operations is the same as for S1, but either A or B will require unaligned load/store operations. With loop S3, in addition to unaligned loads, redundant loads or intra-register data movement operations will be required since an overlapping and unaligned vector is involved in the multiplication of B[4*i-1:4] and B[4*i:4]. Table I shows the performance of S1, S2, and S3 on four different processors (the details of the hardware platforms are provided later in Section.IV). It may be seen that on all platforms, the performance of S2 is worse than S1 and S3 is worse than S2. All three statements execute the same number of vector arithmetic operations and process

1089-795X/11 $26.00 © 2011 IEEE
DOI 10.1109/PACT.2011.59

the same number of distinct data elements. The difference in performance is due to the overheads incurred by one or more of the following: i) redundant load of data elements into different "slots" in different vector registers, ii) unaligned loads instead of aligned loads, and iii) shuffle or alignment operations to move data elements into a different position in a vector register. With all current and proposed short-vector SIMD instruction set architectures, stencil computations will incur overheads similar to that of S3, limiting achievable performance.

In this paper, we propose an architectural solution with compiler support to address the problem. The key idea is to enhance the addressing modes for vector operands in vector arithmetic instructions, by allowing elements from a pair of registers to form one of the operands. We present an architectural design and its simulation to assess the overheads. We also present a compiler algorithm for generating intrinsics-based code for the extended architecture. Using a number of stencil benchmarks, we experimentally evaluate the effectiveness of the approach by using an optimistic and pessimistic emulation of the approach on four platforms. To the best of our knowledge this is the first hardware-based solution along with compiler support to address the performance limitations of current short-vector SIMD architectures for stencil computations. We note that in the following, we assume complementary loop transformations such as loop tiling [7] have been performed to control data cache misses, ensuring the loop nest to be considered accesses data that fits into the L1 cache. The selection and application of such transformations is orthogonal to the work presented here.

The paper is organized as follows. Section. II presents the instruction set enhancement and the hardware implementation of a register file to enable the new addressing modes. Section. III develops the code generation algorithm through a sequence of examples of increasing complexity and performance. The approach to experimental evaluation is discussed in Section. IV and Section. V reports on the experimental evaluation of the proposed approach. Related work is discussed in Section. VI and conclusions are provided in Section.VII.

II. Vector ISA Enhancement via StVEC

Stencil computation typically involves access to adjacent array elements. As a result, vectorized stencil code often uses operands that span multiple vector registers. Aligning vector instruction operands requires additional loads and/or shuffle operations even though the needed operands are already in the register file (albeit unaligned). StVEC proposes changes to how operands are addressed and read from the physical vector register file to allow automatic alignment of operands when needed. This eliminates the need for additional alignment instructions, significantly improving performance of stencil code.

A. StVEC Functionality: An Example

Considering the stencil example in Figure 1(S3), we demonstrate how such a vectorizable loop executes on a 4-wide vector unit (total 128-bit wide vector operands). For this purpose, we show two different versions of the S3 code generated and vectorized by Intel ICC compiler for two different x86-based machines. By having a closer look at the inefficiencies in execution for the generated codes, we then demonstrate how StVEC allows efficient vector code generation and execution. Note that, only for demonstration purposes, we use Intel's SSE instruction notations (i.e. xmm register names, SSE ISA, etc).

In order to vectorize the loop nest in Figure 1(S3), at every iteration of the innermost loop *i*, the vector multiplication ($B[i-1] * B[i]$) requires two operands whose memory alignments are different. Using stride 4 for vectorizing the loop, and since the index starts at 4, first vector operand ($B[i-1]$) is an unaligned access (i.e. vector load) to memory while the second operand ($B[i]$) is aligned. Based on the cost estimation for underlying architectures, vectorizing compilers (e.g. Intel ICC) tend to generate different vector instructions in order to deal with such overlapping memory accesses. The following subsections describe in details the two possible existing compiler solutions along with snapshot of the vector register file (VRF) after executing each code. We also show how StVEC can help overcoming the bottleneck in stencil computation's performance. We assess code efficiency of the different approaches in terms of number of overhead instructions (i.e. unnecessary/redundant vector load, register alignment, register copy, etc) generated per stencil computation (multiplication, in this case).

Extra vector load with alignment: One way to generate code for building an unaligned operand is to load two consecutive vector slots from memory and combine them using data manipulation instructions (i.e. $shuffle$ or $palignr$). This case (shown in Figure 2(a) as generated code for Intel Core2 Quad and the VRF snapshot) is cost-efficient for architectures where unaligned loads are either expensive or not supported. As generated assembly code shows, $palignr$ uses $xmm1$ (that holds a copy of $B[i]$) and another vector slot $xmm14$ (which arbitrarily selected by compiler to hold $B[i-4]$) in order to generate the unaligned multiplication operand, $B[i-1]$. Note that this approach requires registers to hold copies due to limitation in a destructive instruction format where first source operand and destination must be the same. In terms of code efficiency, this solution requires four overhead operations (two loads, one copy and one shuffle) for every single vector multiplication.

Unaligned vector load: For architectures where unaligned loads are successfully implemented with lower costs (i.e. Intel Nehalem), vectorizing compilers generate the code as shown in Figure 2(b). Neither multiplication operands requires extra permutations in this case. We only need two

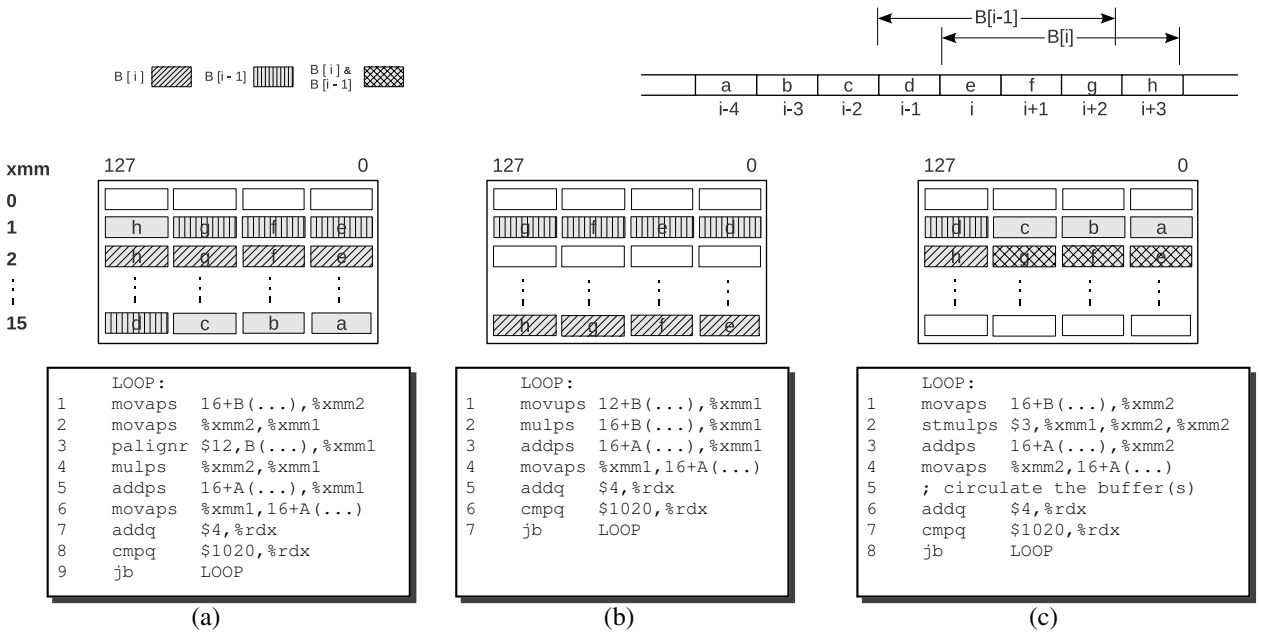


Figure 2. Illustrtion of VRF use and assembly code for code from Figure. 1 (S3): (a) Using extra vector load and alignment instructions, (b) Using unaligned vector loads and (c) using StVEC.

registers ($xmm1$ and $xmm15$) to hold unaligned $B[i-1]$ and aligned $B[i]$ operands, respectively. However, for vector ISAs where unaligned vector load is not supported (i.e. in IBM Power), this approach is not practical. In terms of code efficiency, this solution executes two unaligned loads for every vector multiplication.

We use the VRF snapshot for the two solutions to describe their execution inefficiency. We show execution of the only vector multiplication ($B[i-1] * B[i]$) using labeled elements for simplicity. For the solution in Figure 2(a) with extra vector load, in order to build the aligned operand (45 degrees hatching), all the elements (e, f, g, and h) need to be stored in one register ($xmm2$). However, the unaligned operand (90 degrees hatching) requires four elements that are spread across two different vector registers (d in $xmm15$ and e, f, and g in $xmm1$). Moreover, this operand leaves four already-loaded elements untouched (a, b, c, and h). Thus, the first solution suffers from unnecessary memory accesses, register copy and also the necessity for alignment instructions. Second approach (*Unaligned vector load*), however, suffers from redundant memory accesses due to overlap between the two vector operands (elements e, f, and g that are loaded into both vector registers $xmm1$ and $xmm15$). This cost is in addition to the necessity for architectures to support unaligned memory access.

StVEC (no shuffle and no unaligned load): As shown in Figure 2(c), using StVEC, one can load vectors one by one using *aligned* vector loads. Then, by a simple hardware support in the VRF, vector elements can be read from two different vector registers (i.e. to build the unaligned operand, $B[i-1]$). This solves the overlapping vectors because elements are loaded once but can be (re)used many times. As a result, there are elements in the VRF (crossed 45 degrees hatching representing e, f, and g) that need to be read for more than one vector operands, but don't need to be loaded more than once, due to the StVEC ISA support. Therefore, StVEC eliminates the need for register alignment (using the VRF augmentation to implement in-place implicit alignment) and also the need for unaligned memory access. In fact, as shown in Section III, using some code optimization techniques (i.e. using software pipelining to eliminate register copy in the *buffer circulation* process), StVEC can improve the stencil computation performance by only executing one *aligned* vector load per stencil computation. Providing more details, the following is how StVEC's execution model works in practice.

B. StVEC Execution Model

As discussed briefly, StVEC introduces a set of new arithmetic instructions, that can handle unaligned operands without introducing execution of some overhead instructions. For hardware simplicity, and also destructive instruction format (first source operand is the destination as well), StVEC only supports unalignment for the second source operand. To build such an operand, in cases where it does not fit into one vector register, the instruction requires three pieces of information: a *base* register, an *extension* register, and an offset value. The *base* register is used to locate the vector register in which the first group of elements of the source operand is stored. The *extension*, however, determines which

vector register contains the second group of elements. The *offset* value is used to identify where the first element of the first group located in the *base* register. Thus, considering the example in Figure 2(c), in order to build $B[i-1]$, an adequate indexing information would contain register specifiers $0x01$ and $0x02$ (for $xmm1$ and $xmm2$ as the base and extension registers, respectively) and the offset value of $0x03$. As discussed in Section. III, since vector elements in the *base* register could be loaded in previous iteration(s) of the vectorized loop, it is required to generate register-copy instruction(s) to circulate these temporary buffers. However, using other compiler optimizations (i.e. software pipelining), we will remove the extra register copies. Note that, due to the inherent stride-1 access pattern for stencils (that are considered in this work), knowing one offset value is sufficient to identify all the vector elements. Meaning, for offset value of i, vector width of W, and the two given source operands, we can find the elements in two groups: first from [i:$W-i$] in *base* register and second from [0:i] in the *extension* register. Also, in general, one can assume that the two *base* and *extension* registers are distinct (and not necessarily always consecutive registers such as $xmm1$ and $xmm2$ in Figure 2(c)).

From the functional perspective, for a second vector operand named $VOPR_2$, *base* and *extension* registers, VR_x and VR_y, and different values of offset *ofs*, the second vector operand will be found as shown in Figure 3.

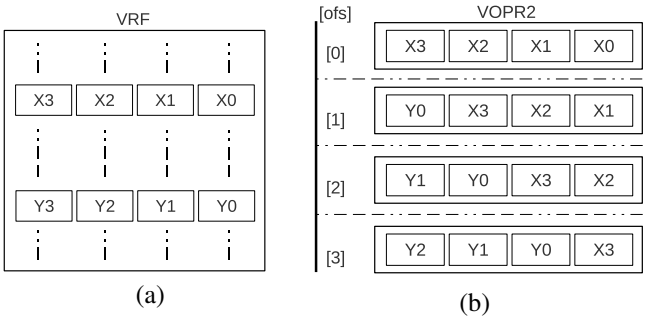


Figure 3. Operand encoding: (a) Two source vector registers, VR_x (X_x elements) and VR_y (Y_y elements), (b) Different permutations for second vector operand, $VOPR_2$, based on values of offset, ofs.

According to the Figure 3, if *offset* (i.e. ofs) is zero, then second operand is the same as the *base* register, VR_x. If *offset* is one, the decoder will select the row associated to VR_x in banks 1, 2 and 3 and to VR_y in bank 0, and so on. Therefore, other than offset value of zero, the first group of elements is in VR_x (starting at the position equals to the offset value) and the second group is stored in VRy. Also, number of elements in the first and second groups are *(W - ofs)* and *(ofs)*, respectively. Note that *W* refers to the vector width.

As suggested earlier in this section, StVEC only requires some changes to the "read-ports" of the vector register file.

StVEC Instruction	Operation
$stvadd\ k,\ VR_x, VR_y, VR_z$	VR_z += VR_x{k:W-k}, VR_y{0:k}
$stvsub\ k,\ VR_x, VR_y, VR_z$	VR_z -= VR_x{k:W-k}, VR_y{0:k}
$stvmul\ k,\ VR_x, VR_y, VR_z$	VR_z *= VR_x{k:W-k}, VR_y{0:k}
$stvdiv\ k,\ VR_x, VR_y, VR_z$	VR_z /= VR_x{k:W-k}, VR_y{0:k}

Table II
STVEC INSTRUCTION FORMAT FOR BASIC SINGLE-PRECISION FLOATING POINT VECTOR OPERATIONS (VECTOR OF SIZE W).

We stipulate a vector register file design that includes four banks, with each bank containing a single word (32-bits) of the multi-word register. Normally all four banks would be accessed with the same address to access a single register. StVEC changes this design to allow each bank to be accessed with a different address. In addition, each bank can provide an element associated to any position in the final output. To support this, we add logic at the output of the register file that shift the output of each bank to the required position in the vector operation's input operand. No changes to the write port to the register file are needed since all the realignment operations are done when reading from the register file.

For the rest of this paper, a generic notation (i.e. *stvadd* for vector-add in StVEC format) and 128-bit wide vector elements are considered as the reference cases to which the StVEC extension will be introduced in details. However, StVEC can be extended to support all the primitive vector arithmetic instructions (i.e. addition, subtraction, multiplication and division) in the existing vector ISAs.

C. StVEC Instruction Format

There are three source operands as vector registers (VR) and an additional 8-bit immediate value encoded in an StVEC instruction. First operand is the *imm8* value for offset, k. Second and third places are taken by *base* and *extension* registers, respectively. The last operand is used for destination vector register. Four primitive StVEC single-precision floating point vector operations are shown in Table II. The same pattern is used for double-precision instructions as well.

Note that StVEC instructions are designed as register-register type such that all source operands are vector registers. Register-memory or register-immediate formats have to be converted by the compiler to sequence of move-compute operations in order to be mapped to register-register style. To illustrate how to use the StVEC format, suppose $stvadd$ instruction is generated with the following operands:

$$stvadd\ \$2,\ VR_2,\ VR_0,\ VR_7$$

According to the Table II, the execution results in adding VR_7 with an operand whose first two elements are taken from VR_2 and the second two elements are taken from VR_0. So, for a vector of size 4 and offset of 2, we have:

$$VR_7 = VR_7 + VR_2\{2:2\},\ VR_0\{0:2\}$$

D. Decoding StVEC Instructions

Decoding StVEC instructions requires special handling of the *base* and *extension* registers and also the offset value. The Decoder generates distinct addresses for each of the vector register banks using the Bank Address Generator (BAG) logic. The BAG logic uses *base* and *extension* register specifiers (7-bit each, in this case) plus the offset value to compute the address for each register bank.

The four bank addresses will be carried along the other information with the vector operand to the operand-read stage where they will be fed to the register file. Note that the BAG logic can be implemented anywhere in the pipeline, after the renaming logic and before the register-read stages.

StVEC instructions read their second operand in two different registers. As a result, they are dependent on three registers rather than two in the case of regular vector operations. These dependencies have to be enforced by the out-of-order scheduling logic. For instance, if the processor uses reservation stations to store pending instructions, reservation station entries need to have one additional pointer to the instruction generating the third register value. The same is true for a reorder buffer-based implementation. The scheduling logic has to enforce these dependencies and not allow the dispatch of an instruction until all dependent registers are available. In theory, adding an extra dependency could slow down execution. In practice this is not an issue because these are true dependencies for stencil codes and the input operands have to be in the register file anyway before execution can proceed.

E. Modified Vector Register File

The second change in the pipeline is re-structuring the vector register file. In general, a vector register file (VRF) is constructed of multiple register banks, each containing one single element. Number of banks is equal to the vector width and number of such physical registers is the same as number of rows in the VRF. In order to demonstrate the architectural changes, we use a sample VRF model containing 128 registers of 128-bit wide each (4 32-bit banks). Note that there is no hardware change introduced to the vector load instructions by StVEC. Therefore, unlike read ports, write ports of the vector register file are not subject to any hardware changes.

To read a 4-wide vector (128-bit), a 7-bit register specifier is fed into the VRF. The corresponding read-port decoder activates word-select lines of the banks accordingly which causes the same row to be selected in all the four banks. When words are read from the banks, i.e. at the end of the register-read stage, slot 0 of the output vector operand contains a word from bank 0, slot 1 from bank 1 and so on. In this normal vector read operation, words are placed in the appropriate output slots such that no adjustment operation (i.e. shift or rotate) will be required.

Offset	W_0	W_1	W_2	W_3
0x00	B_0	B_1	B_2	B_3
0x01	B_1	B_2	B_3	B_0
0x02	B_2	B_3	B_0	B_1
0x03	B_3	B_0	B_1	B_2

Table III
VECTOR REGISTER ADJUSTMENT (VRA) MAPPING BETWEEN BANKS' OUTPUTS (B) AND FINAL ADJUSTED ELEMENTS (W).

However, in order to support StVEC execution model, VRF has to be modified in the following way. Each bank is provided with its own 7-bit address. Instead of having a single decoder feeding the signals (i.e. bit/line select) into all four banks, each bank is outfitted with its own decoder. This is designed to facilitate read accesses to arbitrary registers of each bank. In addition, each bank can provide elements associated to any position in the final output. To support this, we add Vector Register Adjustment (VRA) logic to the output of the VRF. The VRA shifts the register elements to the appropriate positions in the operand (e.g. a block from bank 2 is moved to position 0 in the output when the offset is 0x02). The new VRF design is shown in Figure 4.

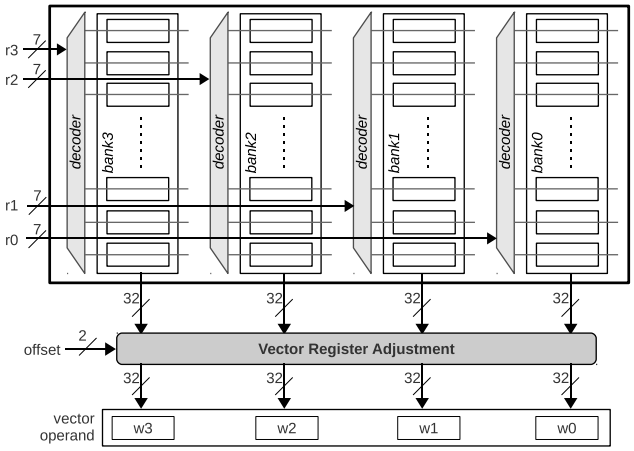


Figure 4. Read-port of a modified VRF including the VRA logic.

VRA logic is similar to a shifter except for the *offset* value which does not directly imply the "shift amount". Table III presents the mapping between the offset value, output elements of the banks (B_0, B_1, B_2, and B_3) and also the final adjusted elements (W_0, W_1, W_2 and W_3).

For aligned operands (offset zero), B elements will be assigned to Ws, with no shift involved. But, in cases where offset is not zero (i.e. an unaligned operand), the VRA connects B elements to appropriate output W slots in order to build the final "aligned" output.

F. Generalizing StVEC

The requirement for an architecture to support StVEC execution model is to be able to decode the proposed instruction format and feature the vector register file such that arbitrary elements spread across different rows can be obtained by

one register read operation. Such a general extension can be implemented on top of the existing SIMD ISAs, such as Intel's SSE or AVX families and IBM-Freescale-Motorola's AltiVec (known as VMX) family. Note that performance improvement achieved by StVEC extension (as will be shown in Section V) substantially depends on the underlying architecture and on the penalty paid by executing unaligned memory accesses and data manipulation instructions.

III. Code Generation

In this section, we describe the compiler algorithm for code generation. We first discuss how to create vectorized code for stencil loops using standard vector intrinsics. Then we show how to generate code to use StVEC instructions.

A. Program Representation

The code generation algorithm operates on an abstract syntax tree (AST) representation of the input program, suitable for detection of innermost loops as well as complex loop transformations such as peeling, unrolling and software pipelining. We assume the code is in three-address form, in order to simplify the process of copying and/or moving specific operations in the loop. The focus of our algorithm is innermost loops that are vectorizable; we assume the required transformations have been done beforehand to expose such loops [8].

We assume candidate innermost vectorizable loops have the following properties:

- Loop bounds are expressions that do not change during an execution of the loop.
- The loop induction variable increments by steps of 1.
- Dependence analysis ensures the absence of loop-carried dependences in the loop.
- The loop has a single entry and single exit point.

We require all memory references in the innermost loop to be of stride-0 or stride-1. That is, for all memory references, two consecutive iterations of the loop must either access two consecutive data elements in memory (stride 1) or the same element in memory (stride 0). Note that stride-1 implies that the innermost loop iterator appears only in the right-most dimension of an array reference for row-major implementation of arrays in languages like C/C++.

Without loss of generality, for the context where the StVEC-based code generation is performed, the expression $expr$ used to dereference a memory address (e.g. in `A[..][`$expr$`]`) is of the form

$$expr = liexpr + iterator + c$$

where $liexpr$ is an arbitrary expression of program symbols whose value is loop invariant during loop execution, $iterator$ is the loop iterator and c is an arbitrary scalar constant. For instance, in the reference `A[i][42*N + j + 3]` with j as the innermost loop iterator, $42 * N$ is a loop invariant expression if N is never assigned in the loop j, and $c = 3$ is the scalar constant for the reference.

B. Auto-vectorization Using Intrinsics

In the following, we present a target-independent algorithm to generate vector intrinsics using vectors of size W, with abstract intrinsics such as `vadd`, etc. to represent vector operations. Our experiments (discussed in Section. V) are based on the SSE and AVX vector instruction sets, but the code generation approach can be used with other vector instruction sets such as Altivec, LRBni, etc.

1) Basic Vector Intrinsics Generation: The input to this algorithm is a representation of an innermost vectorizable loop that conforms to the conditions stated above. We now describe a systematic vector code synthesis scheme to translate this loop into a SIMDized loop, using standard vector intrinsics (such as SSE intrinsics; but we use an abstract notation instead of a target specific notation).

The first stage of the algorithm is to create a basic SIMDized version of the loop, where each stride-1 read memory reference in the scalar code is translated to a vector load in the generated code. Note that to preserve clarity, we outline a general algorithm operating on abstract vector operations, which does not distinguish between aligned and unaligned data. Later in this section, we explain how code can be generated using only aligned loads exclusively, thereby avoiding any overheads of unaligned loads.

The algorithm proceeds as follows, on each candidate innermost loop L:

1. Peel a suitable number of iterations at the end of the loop, so that the number of vectorized iterations is a perfect multiple of the vector length.
2. Change the loop to increment by the vector length.
3. For all variables V with stride-0 access in L, splat V into a vector temporary V_{tmp} and substitute the references to V in L with V_{tmp}.
4. For all read references r_r with stride-1 access in L, create a vector load VL_{tmp} and insert it before the reference, and substitute the reference r_r with VL_{tmp}.
5. For all write references r_w with stride-1 access in L, create a vector store VS_{tmp} and insert it after the reference, and substitute the reference r_w with VS_{tmp}.
6. Remove unnecessary loads and store, typically coming from multiple reads to the same address. Compute use-def chains to remove loads and stores to temporaries.
7. Substitute all arithmetic operations with their vector equivalent.

We illustrate the application of the algorithm using the simple example shown in Figure 5(a). The translation code with vector intrinsics is shown in Figure 5(b). Note that for simplicity the lower boundary of the loop (lba) in Figure 5 is assumed to be an aligned version of the original one (lb).

2) Software Pipelining: The above algorithm presents a simple translation of a loop into its vector equivalent, by using multiple vector loads for adjacent memory accesses. In order to improve the efficiency of the generated code

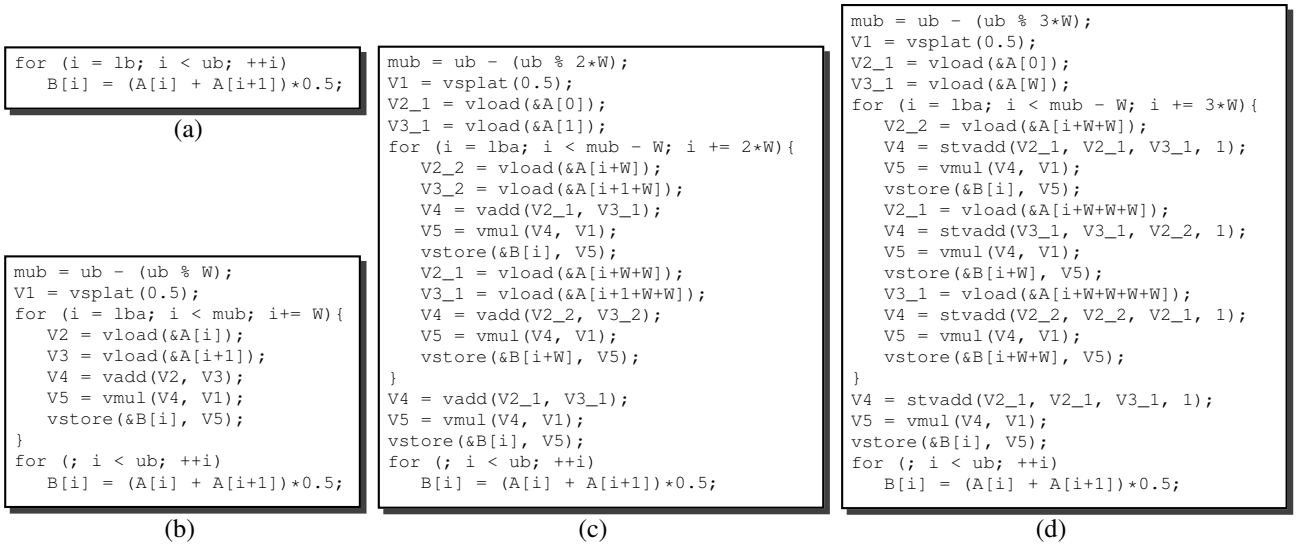


Figure 5. StVEC code generation example: original loop (a), intrinsics translation (b), intrinsics plus software-pipeline (c) final StVEC code (d).

we perform software pipelining [9]. The objective is to overlap computation and data movement, benefiting from instruction-level parallelism. We illustrate this with 2-stage pipelining, where data is fetched one iteration ahead of its use. The algorithm for software pipelining is sketched as follows:

1. Make a copy of the relevant `vload` operations before the loop. Rename the associated vector variables from VX to VX_1 in the copy created.

2. Re-time the `vload` by one iteration in the loop body, and rename the associated vector variables from VX to VX_2.

3. Change references to VX into VX_1 in the arithmetic vector operations in the loop body.

4. Make a copy of the relevant vector operations (`vstore` and arithmetic vector operations) after the loop. Rename the associated vector variables from VX to VX_2 in the copy created.

5. Peel the last iteration of the loop.

6. Unroll the loop by two, as we use a two-stage software pipelining, to avoid the need for variable swap.

7. In the part of the loop body corresponding to the second loop iteration unrolled, substitute all references to VX_1 by VX_2, and conversely.

Returning to the above example, the software-pipelined version is shown in Figure 5(c).

C. Integration of StVEC extension

We now present the code generation technique to use the StVEC ISA extension we have proposed. Our approach is based on the introduction of four new vector intrinsics, and the required modification to our vector code synthesis algorithm to use them.

1) New Intrinsics Proposed: Standard vector intrinsics such as `vadd`, `vmul`, `vsub` and `vdiv` operate on two vector variables and perform the arithmetic operation on these two vectors. Our extension consists in four new intrinsics that each use three vectors and an offset. They are shown in Figure 6.

Standard	Extended
`V1 = vadd (V2, V3)`	`V1 = stvadd (V2, V3, V4, offset)`
`V1 = vsub (V2, V3)`	`V1 = stvsub (V2, V3, V4, offset)`
`V1 = vmul (V2, V3)`	`V1 = stvmul (V2, V3, V4, offset)`
`V1 = vdiv (V2, V3)`	`V1 = stvdiv (V2, V3, V4, offset)`

Figure 6. New intrinsics

In the above, the arithmetic vector operation will use parts of `V3` and `V4` to form the second operand of the vector operation. The `offset` argument is used to specify how many elements come from `V3` and how many from `V4`, as discussed in Section II.

2) Modified Code Generation Algorithm: In the basic code generation algorithm, there is one vector load per memory reference. When accessing `A[i:W]` (`i:W` represents the W consecutive elements of `A` starting from the address `i`) and `A[i+1:W]` in the same loop iteration, two distinct loads are performed at each iteration. Neither of the vectors `A[i:W]` or `A[i+1:W]` is reused at the next iteration. Our ISA extension allows the formation of `A[i+1:W]` from `A[i:W]` and `A[i+W:W]`. An immediate benefit is the ability to reuse `A[i+W:W]` at the next iteration.

The modified code generation algorithm works by detecting the set(s) of `vload` operations such that there are common scalar elements loaded from memory. Given two vector loads `vload(addr1)` and `vload(addr2)`, they load common scalar elements iff:

$$|addr1 - addr2| < W \quad (1)$$

If Eq (1) is true, then the vector arithmetic operations using these two loads can be converted into their `stvxx` equivalent. More precisely, the code generation algorithm constrains one of the two operands to be memory aligned. For the promotion to actually occur, we either have $addr1\%W = 0$ or $addr2\%W = 0$.

The algorithm is outlined as follows, and proceeds by analyzing the various vector loads generated by the previous basic vector intrinsics generation scheme:

1. Peel the x first loop iteration(s) if $(lb + liexpr_{arrays})\%W \neq 0$, $x < W$. The actual value of x is determined at run-time, so that the loop lower bound lba maximizes the number of aligned memory references in the loop. Perform additional statement retiming to minimize the number of unaligned loads (details are provided in Section III-D).

2. Generate a basic intrinsics version, according to the previous algorithm (without software pipelining).

3. Given a set of vector loads to the same array, of the form `A[cst+i+`c`:W]`, for all $k \geq 0$ and until unprocessed loads remain, do

3.1. take the set of the n vector loads VL_p, $0 < p < n$ such that $k.W \leq |c_p| < (k+1).W$,

3.2. if there is no aligned reference in the set (e.g. $\forall p,\ c_p\%W = 0$) or only one reference in the set, proceed with the next set,

3.3. otherwise insert a vector load VL_{new} which loads from the address $cst + i + (k + sign(c_p)).W$, convert all vector operations that consume VL_p such that one of the two operands is either a vector loaded aligned or a local variable, into the corresponding `stvxxx` equivalent. If $c_p > 0$, the operand VL_p is replaced by $VL_{align}, VL_{new}, offset$, where VL_{align} corresponds to the vector load of address $i + k.W$, and $offset = c_p\%W$. If $c_p < 0$, the operand VL_p is replaced by $VL_{new}, VL_{align}, offset$ where $offset = W - c_p\%W$.

4. Perform dead-code elimination, to remove vector loads made useless through the `stvxxx` promotion.

5. Perform a 3-stage software pipelining, to maximize the reuse of the vector loads for the `stvxxx` operations. Three stages are required to avoid register copy since the extended intrinsics address 3 vectors operands.

To illustrate the algorithm, we show its application on the running example in Figure 5(d).

D. Avoiding All Unaligned Loads

The `stvxxx` operations allow the formation of one of the operands from two registers that contain consecutive data elements. We have discussed how loading only aligned data in these two registers is enough for the `stvxxx` second operand. Further, our design requires the first operand to also be aligned. In other words, for the promotion of a vector arithmetic operation into its `stvxxx` equivalent, three registers formed with aligned data are required. This implies the equivalence of the problem of maximizing the number of promotions to `stvxxx` operations with the problem of minimizing the number of unaligned loads.

Consider $cst + i + c$, the index expression of an array used as an operand in an arithmetic operation in the original program, with lb the value of the first iteration of the loop i. It requires only aligned vector loads if $(cst+lb+c)\%W = 0$. That is, if this property is verified for one of the two operands of each of the operations that are the first to consume data elements from the main memory, the promotion to the `stvxxx` equivalent operation is possible and no unaligned load is needed for this operation

Each operation can be retimed freely (that is, iteration shifting is applied to this specific operation to modify which specific *instance* of the statements is executed in the same iteration of the loop i) provided all dependent operations are retimed by the same factor. Retiming changes which data element is accessed at a given iteration, i.e., affects whether or not the data elements accessed at the first iteration are aligned in memory. Consider the example below:

```
for (i = lb; i < ub; ++i) {
   B[i] = A[i] + A[i+1];   // R
   D[i] = A[i+2] + A[i+3]; // S
}
```

Retiming S by $+2$ leads to the following code:

```
for (i = lb; i < lb + 2; ++i)
   B[i] = A[i] + A[i+1];   // R
for (i = lb + 2; i < ub; ++i) {
   B[i] = A[i] + A[i+1];   // R
   D[i-2] = A[i] + A[i+1]; // S
}
for (i = ub; i < ub + 2; ++i)
   D[i-2] = A[i] + A[i+1]; // S
```

After retiming, `A[i]` is an operand of both operations. As soon as $(lb + 2)\%W = 0$, the vector loads required for these operations only loads aligned data. Since we can always dynamically peel iterations of the loop such that $(lba)\%W = 0$, with x being the number of peeled iterations and $lba = lb + 2 + x$, the loop lower bound in the above example, only aligned loads are required. In general, the retiming factor σ_R is chosen such that, for the first consumed operand, we have $c_R = \sigma_R$. This allows the use of only aligned loads for this operand. We generalize this reasoning by considering the only existing retiming constraint *between* operations: all dependent operations are to be retimed with the same σ factor to ensure that semantics is preserved. Since we focus on synchronization-free inner loops, we note that there is no loop-carried dependence. This implies that for the operations $S1, S2, ..., Sn$ which are in dependence, the array index function that causes the dependence is identical in the chain of dependent operations, i.e., $c_{S1} = c_{S2} = ... = c_{Sn}$. This implies that the retiming factors required to align the operand's data access are $\sigma_{S1} = \sigma_{S2} = ... = \sigma_{Sn}$, which will preserve the semantics. By computing individual σ

factors for each set of dependent operations in our 3-address representation, it is thus possible to eliminate all unaligned loads on arithmetic operations.

IV. Evaluation Methodology

The effectiveness of our design was assessed by using a number of stencil benchmarks, using a combination of optimistic and pessimistic emulation on four different processors, as explained below.

A. Baseline Implementation:

The baseline for comparison (named *sp-intrin*) was an implementation of the kernels using standard SSE intrinsics, as described in the first part of Section III. The generated codes use two-way unrolling and software pipelining to perform register loads in the loop iteration prior to use.

B. StVEC Implementations:

Code using StVEC intrinsics was generated, as explained in Section III. This code was then transformed to create three variants.

For the *st-func* variant, each StVEC intrinsic was replaced by a sequence of standard SSE intrinsics that implement the new intrinsic's functionality. This version was used to verify functional correctness of the generated StVEC code.

The *st-pes* variant is a *pessimistic* emulation of the extended instructions in that each $stvxxx$ instruction in the generated code (*st-func*) was replaced by a sequence of two vector arithmetic operations. For instance, the instruction "$stvmul\ 1,\ VR_0,\ VR_4,\ VR_7$" would be replaced by the following two vector multiplications:

$vmul\ VR_4,\ VR_0$
$vmul\ VR_7,\ VR_4$

This version is intended to mimic all data dependences of the StVEC instruction and an execution upper bound on the time required for the StVEC instruction by executing a sequence of two vector arithmetic operations available on existing processors.

The *st-opt* variant is an *optimistic* version that was generated by replacing the StVEC intrinsics simply with a standard SSE intrinsic for that arithmetic operation, using only one of the two paired registers in the StVEC intrinsic. This version serves as a basis for measuring a lower bound for the execution time of the StVEC based program. For the previously considered example, the *st-opt* version of the $stvmul$ intrinsic would execute only one vector multiplication, "$vmul\ VR_7,\ VR_4$". Note that for any $stvxxx$, among all the three source registers (first operand, second-base and second-extension), the extension register (i.e. VR_4) is the latest one which is defined in the program sequence, according to our code generation algorithm.

In our evaluation, we considered the auto-vectorization performance by compiling a C version of the kernel, using the highest levels of compiler optimization. We used both ICC and GCC for our evaluation.

C. Experimental Setup

The hardware platforms used for our experiments are four x86-64 based machines: Intel Sandy Bridge, Intel Core i7-920 (*Nehalem* microarchitecture), Intel Core2 Quad Q6600, and AMD Phenom 9850BE (*K10h* microarchitecture). We use the following labels to refer to the four machines: *i7-sb*, *i7-n*, *core2* and *phenom*. Machine characteristics are provided in Table IV.

Machine	GHz	Cores	SIMD ISA	Peak (GFlop/s)
i7-sb	3.4	4	SSE4.2 + AVX	$\sim$ 56
i7-n	2.66	8	SSE4.2	$\sim$ 21
core2	2.4	4	SSSE3	$\sim$ 19
phenom	2.5	4	SSE4	$\sim$ 20

Table IV
Hardware platforms used for emulating StVEC instructions.

The peak throughput for the machines is shown for single-precision. The double-precision peak performance is around half that for single-precision. Vector data movement and manipulation instructions perform differently on the four platforms, even though all are x86-64 architectures.

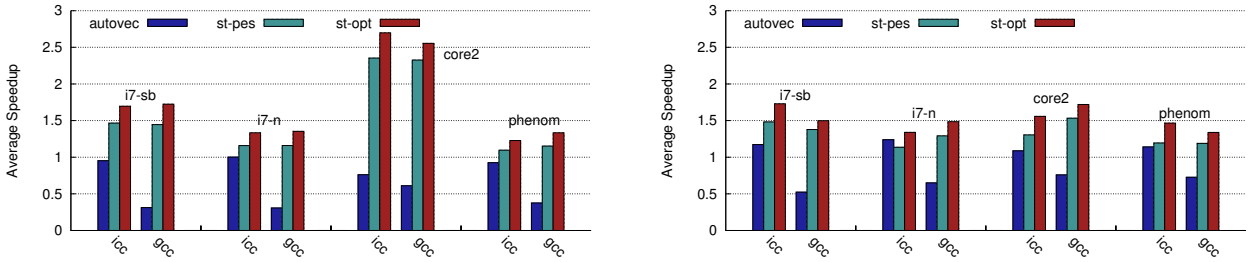
Two compilers, GCC (version 4.4.4) and ICC (version 12.0) were used for the experimental study. Table V lists different compiler optimization options used for enabling auto-vectorization on different machines.

	Options				
Compiler	Common	i7-sb	i7-n	core2	phenom
ICC	-fast	-xavx	-msse4.2	-msse3	-msse4
GCC	-O3	-mavx	-msse4.2	-msse3	-msse4

Table V
Optimization options for ICC/GCC on different machines.

D. Stencil Benchmarks

A set of twelve stencil kernels was used to evaluate this work. The Jacobi kernels form a symmetric stencil pattern been used in many scientific computations, including image processing as well as explicit PDE solvers. We experimented with Jacobi stencils in one-dimension (2, 3, 5 and 7 points), 2D (5 and 9 points) and also 3D (27 points) with different weights for different points. We label the Jacobi kernels in the results section using dimensionality and number of points (i.e. *j2d5p* represents 2-D 5-Point Jacobi). The Parallel Ocean Program (POP) is an ocean circulation model that solves the three-dimensional primitive equations and computes finite-difference discretizations. The two most compute-intensive loop nests of the POP code (as labeled *pop1* and *pop2*) differ from the Jacobi stencils in that the weights (coefficients) are different at each grid point. The *fdtd* 2D kernel represents the core computation in the Finite Difference Time Domain method, widely used in computational electromagnetics. The *rician* 2D denoising kernel is used to remove noise from MRI images by repeatedly executing a stencil computation. Finally, the *heattut* 3D is



(a) Single Precision

(b) Double Precision

Figure 7. Average (geometric means) of relative speedup with StVEC for single and double precision across machines and compilers.

a kernel from the Berkeley stencil probe [10] based on a discretization of the heat equation PDE.

V. Experimental Results

A. Performance Evaluation

We evaluate StVEC's performance on multiple benchmarks, across different machines and with two different compilers. The goal is to observe StVEC's performance on a wide range of platforms. The results are summarized in Figure 7. We show the geometric mean of runtime relative to the baseline (*sp-intrin*). For StVEC, we show the two versions *st-pes* and *st-opt*. For reference we also include performance obtained by automatic vectorization for each compiler (*autovec*). We show data for both single precision Figure 7(a) and double precision Figure 7(b) operations.

StVEC demonstrates consistent performance improvement across the two different compilers (*ICC* and *GCC*), on all the machines. With *GCC*, the StVEC performance improvement ranges from 20% on the *phenom* to $2.47\times$ on the *core2* for *st-opt* and 7% to $2.26\times$ for *st-pes*. The *ICC* improvements are very similar. Also note that both *st-opt* and *st-pes* cases are consistently higher than *autovec*.

StVEC performance improvement is, on average, much higher for *core2* that for the other machines. This is because unaligned memory instructions are very expensive on the Core 2 system. By eliminating these accesses, StVEC achieves a dramatic reduction in execution time.

StVEC performance improvements also scale well to double precision operations. Figure 7 shows average performance improvements ranging from 30 to 65% with *GCC* and 32 to 53% with *ICC* for *st-opt*. For double precision, StVEC does not achieve as large a speedup as for single precision on the Core 2 system. This is because double precision code uses fewer unaligned memory operations that can be eliminated by StVEC.

For reference absolute performance numbers (in GFlop/s) of the baseline kernel, *sp-intrin*, on different machines and compilers and for all the benchmarks are shown in Table VI.

We also take a closer look at StVEC's performance across the benchmarks we test. Figure 8 and Figure 9 show relative speedup for StVEC for single and double

Abs. GFlops	i7-sb		i7-n		core2		phenom	
	SP	DP	SP	DP	SP	DP	SP	DP
j1d2p	25.3	12.4	10.4	5.2	3.3	3.3	7.9	3.7
	20.9	10.6	12.3	4.5	2.3	3.6	8.9	4.5
j1d3p	22.9	11.5	12.6	5.5	3.3	4.3	9.9	4.5
	18.5	9.2	14.2	6.0	3.3	3.4	9.3	4.9
j1d5p	21.2	16.6	13.2	7.8	5.0	4.2	11.7	4.5
	10.5	8.7	13.9	7.9	4.6	3.7	11.9	6.3
j1d7p	18.9	11.8	11.1	6.0	4.3	4.5	11.0	4.8
	8.6	3.7	9.3	5.1	3.9	3.6	9.9	5.4
j2d5p	31.4	16.6	13.0	5.8	5.4	4.9	10.3	5.2
	27.2	14.7	10.6	4.6	5.0	3.1	11.6	6.0
j2d9p	24.2	9.1	13.2	5.0	3.3	3.3	8.3	3.4
	19.4	7.6	10.7	3.2	3.1	2.4	8.0	2.8
j3d27p	13.0	8.0	8.4	4.8	3.5	3.2	5.8	3.0
	8.1	3.6	3.5	1.9	2.1	1.4	2.0	1.1
pop1	12.3	6.5	6.5	3.4	2.6	2.3	4.3	2.3
	7.6	4.2	3.6	1.9	1.8	1.4	2.6	1.2
pop2	12.3	6.7	8.0	4.0	2.8	2.5	5.6	2.5
	7.5	4.2	3.8	1.9	2.0	1.4	2.6	1.5
fdtd	18.0	9.5	9.1	4.6	4.0	3.6	6.0	3.1
	11.4	7.4	6.0	3.0	3.4	2.2	5.4	2.8
heattut	17.0	9.3	8.5	3.3	4.2	3.4	6.7	3.0
	14.4	7.8	6.1	3.2	3.7	2.4	5.7	3.3
rician	14.7	3.3	11.2	2.1	5.5	1.2	5.4	2.0
	11.4	3.0	8.6	2.1	4.8	1.1	4.4	1.9

Table VI
Absolute performance numbers for baseline code (*sp-intrin*): ICC (top) and GCC (bottom).

precision benchmarks, respectively. Note that both *st-opt* and *st-pes* kernels show significant improvement over most the stencil kernels. The only exceptions are *fdtd* which sees a performance degradation and *rician* which sees virtually no performance gain. The *fdtd* benchmark uses 2-point stencil pattern in the outer loop which is not beneficial with the StVEC execution model, where stencils in the innermost loop can only benefit the ISA enhancement.

rician is another case where there is little benefit from StVEC. Although *rician* uses stencil code, its execution time is mostly dominated by vector division operations which are very expensive across all the hardware platforms. Consequently, eliminating unaligned memory operations and shuffle operations has only marginal benefits.

Overall, StVEC achieves very significant performance improvements which are consistent across most benchmarks, different compilers, architectures and computation preci-

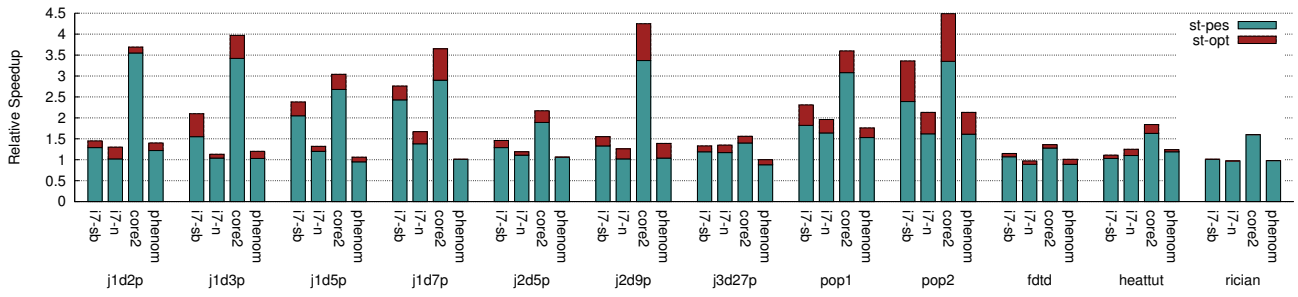


Figure 8. Summary of single-precision improvement across machines.

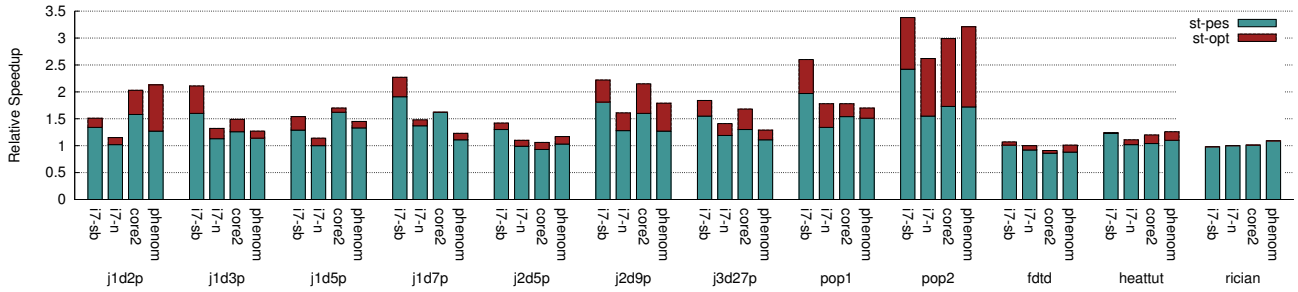


Figure 9. Summary of double-precision improvement across machines.

sions. This shows that eliminating unaligned loads and efficiently re-using already loaded elements is beneficial for cases where unaligned memory access and data manipulation instructions are very expensive (or for architectures that do not support unaligned operations such as such as the IBM Power).

B. Hardware Overhead

To estimate the overhead of the additional hardware required by StVEC, we build a model of the StVEC Register File using CACTI [11]. We augment this model with delay information for the Vector Register Adjustment logic (Figure 4) required by StVEC. The Vector Register Adjustment logic design was synthesized using the Synopsys Design Compiler [12] for 45nm technology using Nangate's Open Cell Library [13]. The synthesized logic is used to determine the additional delay introduced by VRA.

# Regs.	# Banks	Reg. size (bits)	BVRF (ns)	StVRF (ns)
128	4	128	0.24	0.30
256	4	128	0.26	0.32
128	8	256	0.34	0.47
256	8	256	0.37	0.50

Table VII
ACCESS TIME FOR BASELINE AND STVEC VECTOR REGISTER FILES (BVRF AND STVRF) IN 45NM CMOS TECHNOLOGY.

Table VII shows the access time for the StVEC Vector Register File (StVRF) compared to a baseline Vector Register File (BVRF). We show access time for 128 and 256-entry register files with 128 bit (4 word) and 256 bit (8 word) registers. The VRA overhead ranges from 25% to 37% of the total VRF access time. Note that the standard cell library used in the synthesis is not a production library. As a result, the delay measurements are conservative. This overhead can be reduced by using high-speed custom logic. Even with the additional overhead, the StVRF can still be accessed in a single cycle at 3GHz (128 bit configuration) or at 2GHz (256 bit configuration).

VI. RELATED WORK

Several recent studies [1], [2], [3], [4], [5], [6]. have reported on different aspects of optimizing stencil computations, including tiling, effective vectorization, and parallelization on shared-memory and distributed-memory systems, as well as GPUs. However, we are unaware of any work that has proposed a architecture/compiler approach to optimizing stencil computations.

Vectorization for short-vector SIMD architectures has also been a subject of much research [14], [15], [16], [17]. The majority of work on this topic addresses compiler algorithms for generating efficient code for existing SIMD architectures. In contrast, in this paper, we propose a hardware/compiler approach to enhancing the performance of stencil computations on short-vector SIMD architectures.

Previous work has examined the benefits of flexible access and addressing of the register file. For instance, row-wise and column-wise access has been proposed for speeding up matrix operations [18], [19], [20]. These designs are generally more complex than StVEC because they require

concurrent addressing and access to arbitrary words in the register file, including accessing the same word in different registers, needed for column-wise access. This requires a complete redesign of the register file, with word level address decoding. In [21], a flexible permutation of arbitrary-sized data blocks within SIMD registers is proposed. But unlike StVEC, it does not allow operands to span multiple registers.

Henretty et al. have proposed a software-based method to address the stream-alignment conflict of stencils [22]. Their technique requires either a program-wide data-layout transformation or a data layout conversion before and after stencil computations. In contrast, our approach imposes no global layout constraints or data layout conversion overhead.

VII. Conclusion

This paper has addressed a fundamental performance limiting factor with implementation of stencil computations using current short-vector SIMD instruction sets such as SSE — due to the unavoidable overhead of performing multiple loads of data elements from memory or inter-register shuffle operations. An enhanced addressing mode was introduced that allows data elements from two different vector registers to be combined to form operands for vector instructions. A hardware implementation of register files was developed to implement the enhanced addressing mode and a compiler code generation scheme was described for the enhanced vector instruction set architecture. The effectiveness of the new architecture and code generation strategy were demonstrated by using a combination of optimistic and pessimistic emulation on four different x86 CPUs.

Acknowledgments

We are very thankful to the PACT'11 reviewers and program committee for the valuable comments and feedback. We also thank J. Holewinski, T. Henretty, A. Ashari, M. Ravishankar and J. Eisenlohr for their support, comments and suggestions. This research was supported in part by funding for the Center for Domain-Specific Computing (CDSC) through the NSF Expedition in Computing Award CCF-0926127.

References

[1] W. Augustin, V. Heuveline, and J. Weiss, "Optimized stencil computation using in-place calculation on modern multicore systems," in *Euro-Par*, 2009.

[2] H. Dursun, K.-I. Nomura, L. Peng, R. Seymour, W. Wang, R. K. Kalia, A. Nakano, and P. Vashishta, "A multilevel parallelization framework for high-order stencil computations," in *Euro-Par*, 2009.

[3] K. Datta, S. Kamil, S. Williams, L. Oliker, J. Shalf, and K. Yelick, "Optimization and performance modeling of stencil computations on modern microprocessors," *SIAM Rev.*, vol. 51, no. 1, 2009.

[4] S. Krishnamoorthy, M. Baskaran, U. Bondhugula, J. Ramanujam, A. Rountev, and P. Sadayappan, "Effective automatic parallelization of stencil computations," in *PLDI*, 2007.

[5] A. Solar-Lezama, G. Arnold, L. Tancau, R. Bodik, V. Saraswat, and S. Seshia, "Sketching stencils," in *PLDI*, 2007.

[6] G. Wellein, G. Hager, T. Zeiser, M. Wittmann, and H. Fehske, "Efficient temporal blocking for stencil computations by multicore-aware wavefront parallelization," in *COMPSAC*, 2009.

[7] M. Wolfe, "More iteration space tiling," in *Supercomputing*, 1989.

[8] K. Kennedy and J. Allen, *Optimizing compilers for modern architectures: A dependence-based approach*. Morgan Kaufmann, 2002.

[9] M. S. Lam, "Software pipelining: An effective scheduling technique for vliw machines," in *PLDI*, 1988.

[10] S. Kamil, P. Husbands, L. Oliker, J. Shalf, and K. Yelick, "Impact of modern memory subsystems on cache optimizations for stencil computations," in *MSP*, 2005.

[11] N. Muralimanohar, R. Balasubramonian, and N. P. Jouppi, "CACTI 6.0: A tool to model large caches," HP Labs, Tech. Rep. HPL-2009-85, 2009.

[12] "Synopsys Design Compiler," http://synopsys.com.

[13] "Nangate Open Cell Library," http://www.nangate.com/.

[14] A. E. Eichenberger, P. Wu, and K. O'Brien, "Vectorization for SIMD architectures with alignment constraints," *SIGPLAN Not.*, vol. 39, no. 6, 2004.

[15] L. Fireman, E. Petrank, and A. Zaks, "New algorithms for SIMD alignment," in *CC*, 2007.

[16] M. Hohenauer, F. Engel, R. Leupers, G. Ascheid, and H. Meyr, "A SIMD optimization framework for retargetable compilers," *ACM Trans. Archit. Code Optim.*, vol. 6, no. 1, 2009.

[17] D. Nuzman and A. Zaks, "Outer-loop vectorization: revisited for short SIMD architectures," in *PACT*, 2008.

[18] Y. Jung, S. Berg, D. Kim, and Y. Kim, "A register file with transposed access mode," in *ICCD*, 2000.

[19] A. Shahbahrami, B. Juurlink, and S. Vassiliadis, "Matrix register file and extended subwords: two techniques for embedded media processors," in *CF*, 2005.

[20] B. Hanounik and X. Hu, "Liner-time matrix transpose algorithms using vector register file with diagonal registers," in *IPDPS*, 2001.

[21] L. Huang, L. Shen, Z. Wang, W. Shi, N. Xiao, and S. Ma, "SIF: overcoming the limitations of SIMD devices via implicit permutation," in *HPCA*, 2010.

[22] T. Henretty, K. Stock, L.-N. Pouchet, F. Franchetti, J. Ramanujam, and P. Sadayappan, "Data layout transformation for stencil computations on short-vector simd architectures," in *CC*, 2011.