Software Vector Chaining

M. Anton Ertl
TU Wien
anton@mips.complang.tuwien.ac.at

ABSTRACT

Providing vectors of run-time determined length as opaque value types is a good interface between the machine-level SIMD instructions and portable application-oriented programming languages. Implementing vector operations requires a loop that breaks the vector into SIMD-register-sized chunks. A compiler can fuse the loops of several vector operations together. However, during normal compilation this is only easy if no other control structures are involved. This paper explores an alternative: collect a trace of vector operations at run-time (following the program control flow during this collecting step), and then perform the combined vector loop. This arrangement has a certain run-time overhead, but its implementation is simpler and can happen independently, in a library. Preliminary performance results indicate that the overhead makes this approach beneficial only for long vectors (> 1KB). For shorter vectors, unfused loops should be used in a library setting. Fortunately, this choice can be made at run time, individually for each vector operation.

CCS CONCEPTS

• **Software and its engineering** → **Just-in-time compilers**; • **Computer systems organization** → *Single instruction, multiple data*;

ACM Reference Format:
M. Anton Ertl. 2018. Software Vector Chaining. In *15th International Conference on Managed Languages & Runtimes (ManLang'18), September 12–14, 2018, Linz, Austria.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3237009.3237021

1 INTRODUCTION

Hardware provides SIMD instruction set extensions such as AVX (Intel, AMD) to support data-parallel processing within a single thread.

Many programming languages do not have language-level support for SIMD instructions, and instead rely on auto-vectorization by the implementation. While auto-vectorization promises to work for legacy code, its success is unreliable, and it generally requires quite a bit of compile time, and is therefore not a good choice for JIT compilers. Auto-vectorization is also relatively complex, and is therefore rarely implemented in non-HPC compilers.

Another approach is to provide language features that allow expressing operations at the SIMD instruction level, such as the recent Java Vector API draft [ILSV18]. This is simple to implement, but puts the onus of not just vectorization, but also of optimizing the vector operations completely on the shoulders of the programmers.

The approach we look at in this paper uses a slightly higher-level interface: application-sized vectors in opaque value types (Section 2). This interface is abstract enough to allow a variety of implementation approaches and optimizations.

A simple implementation of this interface performs a loop per vector operation. In otherwise straight-line code several vector operations can be combined into a single loop, reducing the necessary loads and stores, and the loop overhead. However, extending this optimization across program-level control structures is complicated.

Instead, in this paper we propose to collect a trace (or chain) of vector operations *at run-time*, even across control-structures. Once such a trace is complete, code for it is generated, and executed. Actually, in the usual case, the same trace will have occured before, and instead of generating the code again, the previously generated code is used.

There are certain similarities of this optimization to the hardware chaining of vector operations on the Cray-1, hence the title of this paper.

For long vectors, this optimization technique gives a speedup over having a separate loop for each vector operation. It can be implemented as a library, and therefore without change to the existing JIT compiler.

This technique is the main contribution of this paper and we present it in more detail in Section 3, discuss its implementation in Section 4 and evaluate it in Section 5. Section 2 discusses the background of SIMD and vectors.

1.1 Potential misunderstandings

Manual vectorization. This paper is not about automatic vectorization, but about an optimization for manually vectorized code.

Vector traces every time. Our optimization builds a trace of vector operations every time it executes the vector operations, not just when the rest of the code is JIT-compiled.

Code generation only for unique traces. Our optimization uses a hash table to avoid having to re-generate the native code every time.

2 BACKGROUND

This section gives an overview of SIMD instructions and programming language approaches for making use of them.

2.1 SIMD instructions

In many applications one has to perform the same operations on a lot of data, mostly independently, sometimes combining the results. This is known as *data parallelism.*

ManLang'18, September 12–14, 2018, Linz, Austria

ACM ISBN 978-1-4503-6424-9/18/09…$15.00
https://doi.org/10.1145/3237009.3237021

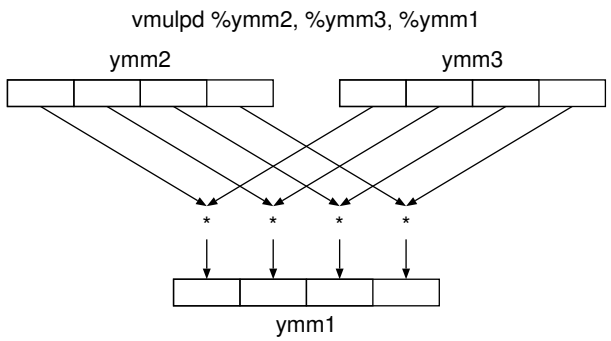

Figure 1: A SIMD instruction: `vmulpd` (AVX)

Data parallelism is obvious for many scientific applications, but can also be found in other applications, e.g., in the Traveling Salesman Problem[1]. So introducing a wordset for expressing data parallelism may be useful in more applications than one might think at first.

Computer architects provide SIMD (single instruction multiple data) instructions that allow to express some of this data parallelism to the hardware. The Cray-1 [Cra77] was an early machine with SIMD instructions, but starting in the 1990s, microprocessor manufacturers for general-purpose CPUs incorporated SIMD instructions in their architectures. E.g., Intel/AMD incorportated MMX, 3DNow, SSE, AVX etc. and ARM incorporated Neon, and has presented SVE.

These instruction set extensions typically provide registers with a given number of bits (e.g., 256 bits for the YMM registers of AVX), and pack as many items of a basic data type in there as fit; e.g., you can pack 16 16-bit integers or 4 64-bit FP values in a YMM register. A SIMD instruction typically performs the same operation on all the items in a SIMD register. E.g., the AVX instruction `vmulpd %ymm2, %ymm3, %ymm1`[2] multiplies each of the elements of ymm2 with the corresponding element in ymm3, and puts the result in the corresponding place in ymm1 (Fig. 1).

2.2 Automatic or manual vectorization?

Ever since the Cray-1 there has been the hope of auto-vectorization: Programmers would write scalar code oblivious of SIMD instructions, and the compiler would find out by itself how to make use of these instructions for that code [AK87].

While auto-vectorization occasionally succeeds in vectorizing a piece of code (especially benchmarks), it is an unreliable method; there are often obstacles that make it hard or impossible for the compiler to vectorize the code, e.g., the possibility of overlap between memory accesses in the loop; and if you ask the programmer to change his program to remove these obstacles, why stick with scalar code? If the programmer thinks in terms of vectorizing the program, the way to go is to directly express vector operations rather than expressing them through scalar operations and then hoping that the compiler will auto-vectorize them. Compiling language-level vector operations[3] to SIMD code also requires much less compiler complexity and compile time than auto-vectorization.

2.3 Hardware or application vectors?

For vectors as language features, one design decision is which vector length to support. GCC and the Java Vector API draft [ILSV18] decided to expose the SIMD granularity at the programming language level, while APL, Fortran, and a Forth extension [Ert17] support vector or array operations of arbitrary, dynamically determined size. Programming at SIMD granularity means that the programmer has to perform significantly more work when vectorizing the application, and it also puts all the responsibility (but also the opportunity) for optimizing vector operations on the shoulders of the programmer.

In particular, the program has to process the application data in SIMD-sized parts in a loop, with an extra loop for the extra elements, a transformation called strip-mining. Figure 2 shows a vectorizable scalar loop, Fig. 3 shows the same computation using the Vector API, with manually applied strip-mining; and Fig. 4 shows the same computation using dynamically-sized vectors.

2.4 Arrays or opaque vectors?

Another design decision is whether vector operations access the normal arrays of the programming language (as the Fortran 90 array operations do), or work on special vector types; and if the latter, what the properties of these vector types should be. Figure 5 shows the example in Fortran 90, with direct accesses to the arrays, while the variant in Fig. 4 first copies the data from arrays into a `FloatVect`, and copies the result back into an array. Directly accessing the existing arrays appears more natural, but it has a number of disadvantages:

- The operands of array operations often do not have a size that is a multiple of the SIMD granularity. This makes it necessary to have special treatment for the last elements, e.g., as in Fig. 3.
- The operands of array operations are often not aligned to SIMD granularity. Different operands may have different offsets from SIMD alignment. The program therefore has to perform unaligned SIMD accesses, which are less efficient than aligned accesses.
- Fortran 90 array slices (which can be operands of array operations) may contain elements that are not consecutive in memory, and accessing such array slices is several times more expensive than for arrays of consecutive elements.
- Operands of vector operations may overlap the result, so the parts of the vector operation may need to be performed in a special order, or one may need an intermediate copy.
- The operands of vector operations may may-alias with array accesses for control-flow, indexing or addressing purposes. As a consequence, the compiler may be unable to combine vector operations.

[1] <news:b2aed821-2b7e-456d-9a6d-c2ea1fdedd55@googlegroups.com>

[2] In this paper, we use the AT&T syntax for the AMD64 architecture; in contrast to Intel syntax, the destination of an instruction is the rightmost operand in AT&T syntax.

[3] You may wonder about what vector operations to support; we have discussed this in our earlier work [Ert17], but do not have a final answer yet; there may also be additional vector operations as the result of new hardware features, such as AVX512's masked instructions.

```
void scalarComputation(float[] a, float[] b, float[] c) {
   for (int i = 0; i < a.length; i++) {
        c[i] = (a[i] * a[i] + b[i] * b[i]) * -1.0f;
   }
}
```

Figure 2: A vector computation expressed as ordinary scalar Java code [ILSV18]

```
static final FloatVector.FloatSpecies<Shapes.S256Bit> SPECIES =
        FloatVector.speciesInstance(Shapes.S_256_BIT);

void vectorComputation(float[] a, float[] b, float[] c) {
    int i = 0;
    for (; i < (a.length & ~(SPECIES.length() - 1));
           i += SPECIES.length()) {
        //FloatVector<Shapes.S256Bit> va, vb, vc;
        var va = SPECIES.fromArray(a, i);
        var vb = SPECIES.fromArray(b, i);
        var vc = va.mul(va).
                    add(vb.mul(vb)).
                    neg();
        vc.intoArray(c, i);
    }

    for (; i < a.length; i++) {
        c[i] = (a[i] * a[i] + b[i] * b[i]) * -1.0f;
    }
}
```

Figure 3: The same vector computation expressed using the Vector API [ILSV18]

```
void VectComputation(float[] a, float[] b, float[] c) {
    FloatVect va = new FloatVect(a);
    FloatVect vb = new FloatVect(b);
    va.mul(va).add(vb.mul(vb)).neg().intoArray(c);
}
```

Figure 4: The same vector computation expressed with dynamically-sized vectors

```
subroutine ArrayComputation(a,b,c,n)
  integer n
  real a(n), b(n), c(n)
  c(1:n) = -(a(1:n)*a(1:n) + b(1:n)*b(1:n))
end subroutine
```

Figure 5: The same vector computation expressed using Fortran 90 array operations. This example uses the array slice notation instead of the shorter whole-array notation to demonstrate that these operations can be performed on parts of arrays.

Dealing with dependencies for SIMD and other parallelization purposes fills many papers and books [Wol96].

Separate, opaque types with value semantics for vectors [Ert17] solve several of these problems:

- Vector data can be padded to SIMD granularity, eliminating the special treatment for the last elements in many cases.
- Vector data can be aligned to SIMD granularity, so all operands of SIMD instructions are aligned.
- Results do not overlap input operands, thanks to value semantics.
- Operands of vector operations do not alias with array accesses or anything else from the scalar and array world. Indeed, vectors live in their own separate world, and data flow between these two worlds is always explicit, and often in one direction (towards vectors), see Fig. 6. This makes vector operations relatively loosely connected to the scalar/array world, and that is the basis of software vector chaining.

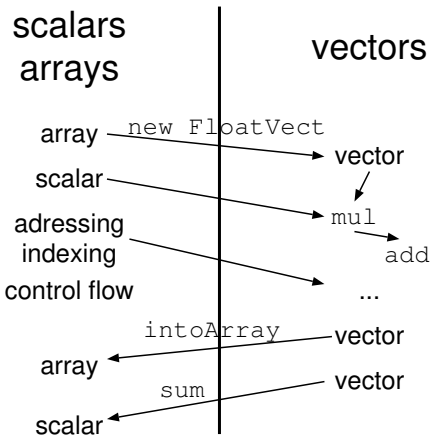

Figure 6: Separate vectors divide the system into a scalar/array world (with aliasing) and a vector world (with only explicit dependencies)

```
simple:
 vmovaps (%rdi,%r10,1),%ymm0
 vaddps  (%rsi,%r10,1),%ymm0,%ymm0
 vmovaps %ymm0,(%rdx,%r10,1)
 add     $0x20,%r10
 cmp     %r10,%rcx
 ja      simple
```

Figure 7: A loop of SIMD instructions that implements a `FloatVect add`

- Overall, with these vector types the compiler has a lot of freedom for (re)arranging the computations. And making good use of this freedom does not take heroic efforts.

You pay for these advantages by making explicit conversions between arrays and vectors (`new FloatVect` and `intoArray` in Fig. 4). But note that these explicit conversions can be optimized away in cases where they cost more than they buy. Conversely, an array-based approach could copy the slices under consideration to an aligned and padded intermediate area.

So is there a difference between using separate vectors and just using arrays? And in particular, is there an advantage to separate vectors? Yes: Separate vectors do not alias with array accesses, making various transformations much easier.

Also, the programmer can keep the data much longer in the vector world, than a compiler can keep array slices in an intermediate area, considerably reducing the amount of copying necessary. E.g., in $n \times n$ matrix multiplication, the programmer can convert an input matrix from $n \times n$ array to n vectors at the start, and reuse these vectors n times, and in the end convert the result back. This amortizes the conversion over many uses.

```
fused:
 vmovaps (%rdi,%r10,1),%ymm0
 vmulps  %ymm0,%ymm0,%ymm1
 vmovaps (%rsi,%r10,1),%ymm2
 vmulps  %ymm2,%ymm2,%ymm3
 vaddps  %ymm1,%ymm3,%ymm1
 vxorps  %ymm1,%ymm4,%ymm1
 vmovaps %ymm0,(%rdx,%r10,1)
 add     $0x20,%r10
 cmp     %r10,%r9
 ja      fused
```

Figure 8: A fused loop for the vector-world parts of Fig. 4 (no `new` or `intoArray`)

3 SOFTWARE VECTOR CHAINING

3.1 Vector loop fusion

A simple way to implement the vector operations is to translate each one into a loop of SIMD instructions. E.g., Fig. 7 shows such a *vector loop* for `FloatVect add`.

If you have several consecutive vector operations with same-length vectors, the compiler can fuse these vector loops. The criteria for "consecutive" are relatively easy to meet, thanks to the nice properties of our separate vectors.

If a vector operation reads a vector produced by a previous vector operation in the fused loop, the compiler can avoid the loads of the intermediate vector and instead use the results directly from the SIMD register. If the result of a vector operation has no other users except those in the fused loop, the store of the result of this vector operation can be eliminated. E.g., the loop in Fig. 8 performs the four vector operations of Fig. 4 (32 FLOPs per vector-loop iteration) in 10 instructions per loop iteration, compared to 24 instructions for the same work using 4 simple loops.

On a modern CPU with out-of-order execution, these loops tend to run as fast as hardware resource constraints permit: they exhibit data parallelism, so each iteration is almost independent from earlier iterations (apart from the loop-counting `add`, which causes only one cycle of delay between iterations). For CPUs with in-order execution, the compiler can employ software pipelining [Cha81, Rau94] to reorder the instructions.

For a reduction operation like `sum` that adds up all the elements in the vector, the intermediate result of each iteration depends on the result of the previous iteration, resulting in 4 cycles/iteration on recent Intel CPUs for a `FloatVect sum`. Loop fusion is even more beneficial in the presence of reductions, because you can add a lot of instructions to an iteration before it needs more cycles due to resource constraints.

A problem for compile-time loop fusion is that even with all the benefits of separate vectors, things become difficult once the basic block ends. Then you have to combine the control structure of the surrounding program with the vector loop, or limit yourself to combining vector operations within basic blocks.

```
static FloatVect[] matmul(float a[][], FloatVect[] vb)
{
  FloatVect[] vc=new FloatVect[a.length];
  for (int i=0; i<a.length; i++) {
    FloatVect vci = new FloatVect(vb[0].size());
    for (int k=0; k<vb.length; k++)
      vci = vb[k].mul(a[i][k]).add(vci);
    vc[i] = vci;
  }
  return vc;
}
```

Figure 9: Matrix multiplication using dynamically-sized vectors

As an example, consider matrix multiplication (Fig. 9). There are only two vector operations in the inner loop[4]. By unrolling the inner loop, we can get around this problem in this case. But what if, on average, half of the elements of `a` are 0? A programmer can modify the inner loop as follows to exploit this knowledge for improving performance:

```
for (int k=0; k<vb.length; k++)
  if (a[i][k]!=0.0)
    vci = vb[k].mul(a[i][k]).add(vci);
```

In such a loop, unrolling does not increase the number of vector operations in a basic block. More complex loop transformations [Wol96] can help, but they require substantial compiler complexity that most implementors of general-purpose compilers will not invest in vector implementation.

A trace-based compiler [BDB00, GPF06] extends the boundaries for moving vector operations as far as the scalar+vector traces go. A variant of the technique presented in this paper could be used in a trace-based compiler to increase vector processing performance with relatively low implementation effort, see Section 6.

3.2 Software vector chaining: Run-time fusion

However, in this paper we focus on a variant that can be implemented as a library. In this way we do not limit ourselves to trace-based compilation. Instead, we can implement the technique in all kinds of implementations, even in an interpreter.

Software vector chaining collects only the vector operations into a trace *at run-time*, instead of trying to combine them during compilation. Once the trace ends, a fused vector loop encompassing all the vector operations in the trace is invoked; if necessary, this loop is generated first, but in the usual case, this loop will already have been generated earlier in the execution, and is just looked up.

Collecting vector operations into a trace is possible, because the vectors live in a separate world from the rest of the computation, and so all the vector-only operations can be delayed until we need to deliver a result to the rest of the computation, as shown in Fig. 6.

This approach costs some overhead for collecting the trace and finally looking its code up, but the hope is that this will be made up by the possibility to combine more vector operations into one loop, reducing the cost of the vector computations by more than the additional overhead. For long vectors, vector operations are relatively heavy-weight, and loop fusion saves quite a bit of execution time, so this approach of generating a trace every time can pay off here.

For shorter vectors, the overhead of collecting a trace will not pay off. Instead of always collecting a trace, the behaviour can be made to depend on the vector length: For short vectors, perform the operation right away (i.e., use a simple vector loop per vector operation); for long vectors, add the operation to the current trace. This is possible because software vector chaining works at run time, and the operands and their vector lengths are known.

3.3 Trace end

There are several reasons for ending a trace:

- When a vector operation with a different length has to be performed. It needs a different controlling loop.
- When a vector operation produces a result for the scalar/array world, e.g., `intoArray` or `sum`.
- When the trace becomes too long. The benefits of combining more operations diminish, while the costs, in particular, the number of different loops that have to be generated and stored, increases. Also, the resulting loops might run into hardware limits, and that could reduce performance. These issues are best determined by measuring performance with different trace length limits.
- When the number of SIMD or general-purpose registers that need to be alive in the fused loop exceeds the available registers. This is another form of a too-long trace, but it is easier for the compiler to determine whether this limit has been reached, and it depends more on the data flow in the trace than on the raw length.

3.4 Avoiding dead stores

One issue is whether we can avoid storing all the computed vectors, including all intermediate results. In the matrix multiplication example (Fig. 9), in principle only the value of `vci` at the end of the inner loop is really needed. In software vector chaining, we need to store the result of last `add`; if the trace ends with a `mul`, we also need the result of that.

One way to achieve this would be if the vector representation has reference counters that represent the number of references to the vector; after ending a trace, if a vector only has references from the trace (such as all the vectors produced by `add` except the last one in the trace), we do not need to store the result to memory, but can just pass them along through SIMD registers.

While vector operations as presented in this paper, including software vector chaining, can be implemented as a library, this store-eliminating optimization would require VM cooperation in a garbage-collected VM, and it is probably hard to adapt such a VM to keep reference counters up-to-date.

An alternative is to enlist programmer help. E.g., the programmer could mark the last, consuming use of a vector with `.c`, as in the following variant of the loop above:

```
for (int k=0; k<vb.length; k++)
  if (a[i][k]!=0.0)
    vci = vb[k].mul(a[i][k]).c.add(vci.c);
```

[4] *Inner loop* refers to the programmer-written inner loop; we refer to the loop that implements a vector operation as *vector loop*.

The example from Fig. 4 would look as follows:

```
va.mul(va).c.add(vb.mul(vb).c).c.neg().c.intoArray(c);
```

`.c` does not conform to the value semantics that we otherwise maintain for vectors: If the program uses the resulting vector more than once, the result will be a null pointer exception on second use. We think that this price is acceptable given the benefit.

What is not covered by `.c` is if there are multiple uses of a vector in an expression, but none afterwards. One can introduce special methods for covering that case, but it is unclear if this is useful enough to be worth the implementation effort.

3.5 Hardware vector chaining

The parallels with hardware vector chaining are: In hardware chaining, each vector element[5] flows directly from one functional unit to the next, without having to wait for the first operation to complete on the whole vector, like in our fused loops. And like in our software chaining, hardware chaining performs this combining on its own, without intervention from the compiler, including across control flow [Cra77]. A difference is that software chaining also combines independent vector operations, while hardware chaining combines vector operations where the data flows from one operation to the next. Vector hardware also performs independent vector operations in parallel, if the hardware resources are available, but in hardware this is not called chaining.

4 IMPLEMENTATION

This section describes the implementation of software vector chaining. These code generation techniques are not original, and can be found in ancient compiler textbooks [ASU86], but they are described here to demonstrate that implementing software vector chaining is simple, especially when compared with auto-vectorization, or dependence analysis and loop transformations used in classical (function-level) compilation [Wol96].

4.1 Vector representation

A simple implementation of vectors (without chaining) stores the vector data, and a little bit of metadata, in particular the length.

With vector chaining, the implementation of vectors becomes slightly more sophisticated. Some vectors may never have any data stored in memory; their data only exists temporarily, as SIMD-sized pieces in SIMD registers, so only the metadata exists, while the data part is represented by a null pointer. The meta-data now also contains a flag that indicates whether the next use will be the last use, and information used in the compilation of the current trace: a SIMD register number, and an address register number (with a special value indicating that the vector is not to be stored).

The trace generation and vector loop generation works mostly like a simple basic-block-level compiler:

4.2 Trace construction

The trace is represented by an array of quadruples: the operation, the two source vectors, and the result vector.

[5]In contrast to current SIMD implementations, classical vector processors process vectors not completely in parallel, but in a pipelined fashion, one element per cycle.

Calling a vector operation method creates a result vector, and assigns a free SIMD and address register to it. The method also adds a quadruple for this operation to the current trace. If an operand of the vector operation has no earlier occurence in the trace, a load operation for it is added to the trace first. If an operand has its last-use flag set, its SIMD register is added to the pool of free registers; and if it is the result of a non-load operation, its address register is set to the *don't-store* value, and the address register is freed.

A more sophisticated register allocator could use fewer SIMD registers in the presence of non-`.c` vectors; while it would be relatively cheap (in both implementation effort and run-time), it is unclear whether the benefits are worth even these modest costs. The address registers live in the whole vector loop, so for them no better register allocation is possible.

4.3 Code generation

Once a trace ends, we may have to generate code for it. That is relatively easy: The loop control flow and loop counting are boilerplate code. For a load operation, a load from the address given by the address register plus the counter is generated. For other operations the appropriate operation is generated; if the address register indicates a store, the compiler then generates a store to the address given by the address register plus the counter. Figure 10 shows an example.

Before executing the code, memory for vector data is allocated for those vectors that have their data stored. The addresses of both the loaded and the stored vector data are then loaded into the appropriate address registers, and the code is invoked.

This works nicely for vector-parallel operations. Other operations have some deviations from this scheme:

The code generator treats a scalar operand of a vector operation (e.g., `mul(a[i][k])` in the matrix multiplication example) as follows: Like the input vector addresses, the code passes the scalar as parameter to the resulting code (allowing to use the same code with different scalars). Just before the loop, the generated code copies the scalar to all lanes of a SIMD register (e.g., the first two instructions in Fig. 11), and it then uses that SIMD register as operand inside the loop. This SIMD register is alive throughout the loop.

When the program copies data from an array to a vector (e.g., `new FloatVect(b)`), the vector library has to perform that copy immediately (the array could be changed right afterwards), so it is not incorporated into the trace. It does not end the current trace, though.

The vector library also has to copy data from a vector to an array (e.g., `intoArray`) immediately. The vector will usually depend on vector operations the trace, so this copy ends the trace and executes the operations in the trace. The vector loop could also include the copying to the array, but that would require special-casing the last iteration (if fewer array elements are to be stored than fit in a SIMD register), so we may prefer to do the copying separately.

For implementing reducing operations (e.g., `sum`), the compiler needs to fill the padded elements with the neutral element of the operation (e.g, 0 for `sum`) in front of the loop. In the vector loop, every iteration combines (e.g., adds) the operand SIMD register with the result SIMD register. Finally, after the vector loop, the

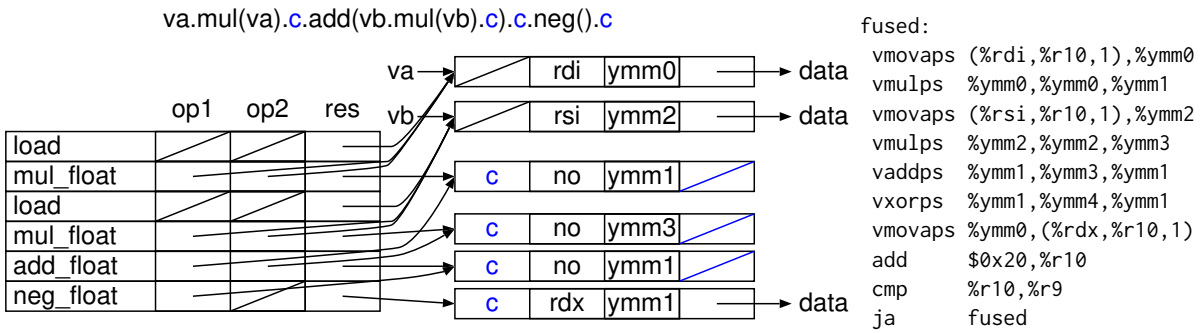

Figure 10: Source code and the corresponding quadruple representation and machine code. The `neg` gets an address register and data despite being marked with `c`, because the consumption of the resulting vector does not happen inside the trace.

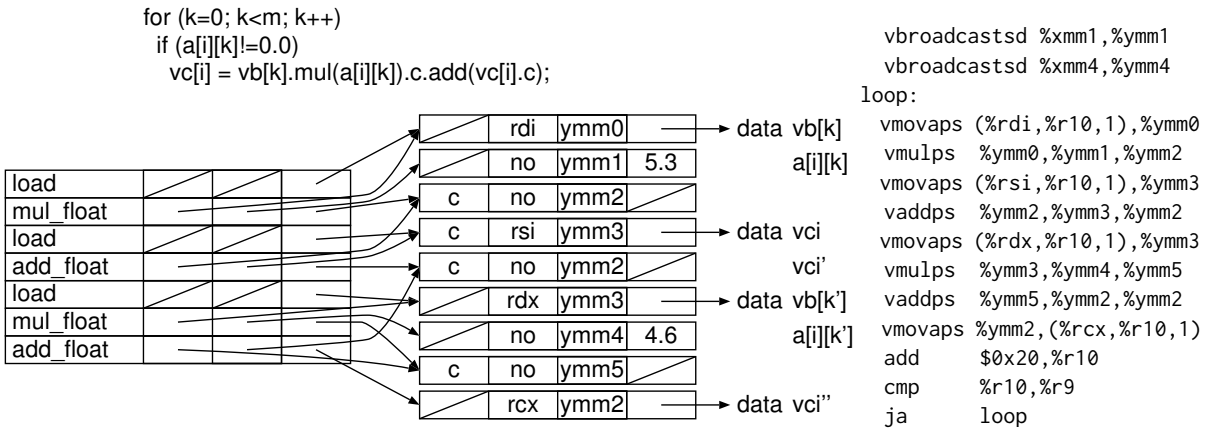

Figure 11: A loop, the trace from several iterations of the loop with two taken `if`s, and the resulting code

elements of the result SIMD register have to be combined to form a single scalar. The result of the reduction has to become available to the scalar world immediately, so a reduction ends the trace.

4.4 Caching

We do not want to perform complete compilation (with I-cache synchronization etc.) every time we perform a number of vector operations, so we cache the code produced for a trace in a hash table.

Building the trace and then looking it up in the hash table are time-critical, because they are performed on every execution, so can we reduce the effort necessary? We can delay register allocation (cheap as it is in our settig) until compilation, and record only on which erlier operations in the trace an operation depends, or if it depends on an input vector or a scalar, and whether the result is stored. A dependency is best represented by an index into the trace for ease of hashing and comparison.

5 EVALUATION

We added software vector chaining to the Forth vector library[6]. The implementation currently takes 386 lines of code. It does not generate machine code directly, but instead uses C (with GNU C's SIMD vector extension) as intermediate language. Still, the compilation to C is performed along the lines presented in Section 4. The resulting machine code is also pretty similar to what has been presented here, except that gcc often combines loads and operations into load-and-op instructions. The C code generation time is included in the results, the time for running gcc is not: code compiled earlier is reused through a file-caching mechanism.

We evaluate the performance using matrix multiplication: We multiply a 50×50 matrix with a $50 \times n$ matrix for n varying from 1–12000. This matrix multiplication is performed 500 times in order to reduce the influence of startup overheads. As a result, this benchmark performs 1,250,000 iterations of the inner loop (a vector-scalar multiplication, and a vector-vector addition, both with vector length n). By varying n, we can see what the startup overhead is, where one approach surpasses the other, and what the performance

[6] https://github.com/AntonErtl/vectors,
commit 4d060c16531ebd94e7556c49473f868134691acb used for the results

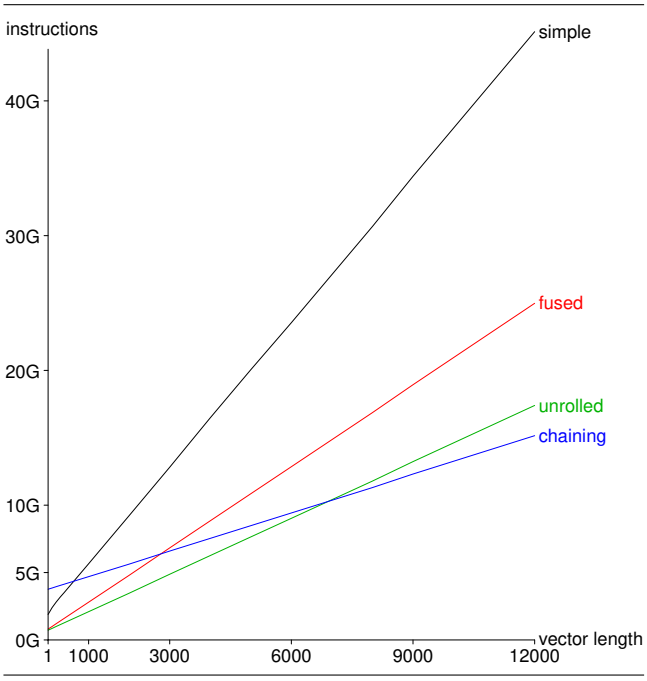

Figure 12: Instructions for 2.5M vector operations using AVX instructions

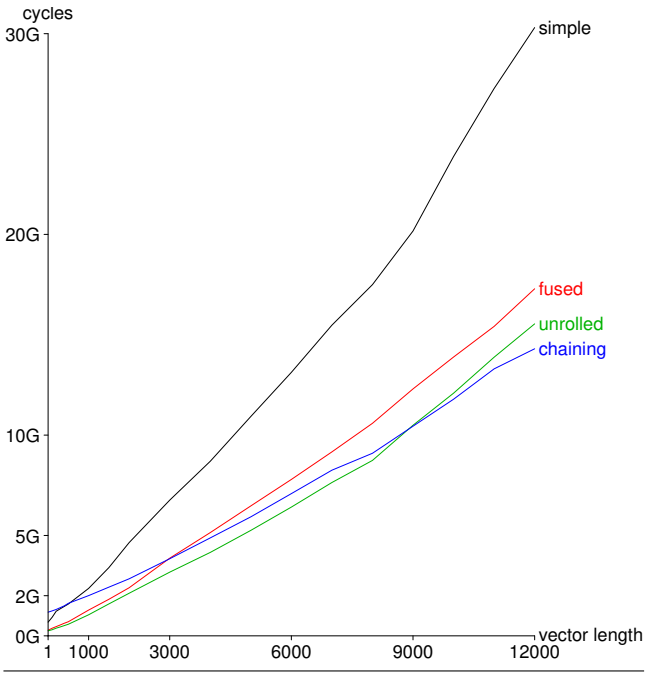

Figure 13: Cycles for 2.5M vector operations using AVX instructions

for long vectors is. In contrast to the examples in this paper, we use 64-bit FP numbers, though.

We compared the following variants:

simple Each vector operation gets its own vector loop, as in Fig. 7.

fused We manually fused the vector operations of one iteration of the inner loop in the source code of the matrix multiplication code, and use a special multiply-add vector operation. This simulates vector-loop fusion by the compiler within a basic block.

unrolled We manually unrolled the inner loop by a factor of 2 in the source code, and used a special multiply-add-multiply-add vector operation (with code similar to Fig. 11) in it to simulate the combination of inner-loop unrolling and vector-loop fusion.

chaining This employs our chaining implementation, including its overheads. For this benchmark, 16 vector operations (e.g., 8 inner loop iterations, but possibly also the last iterations of one instance of the inner loop, and the next iterations of the next one) are chained together.

We don't expect our implementation language to make a significant difference for *fused* and *unrolled*, but for the other two, there can be one:

In *simple*, every iteration allocates a vector and later frees one (and this takes a considerable proportion of the time); our implementation uses the `malloc`/`free` of glibc-2.19, which may perform differently from memory allocation and reclamation in a managed language, and different from other `malloc`/`free` implementations. *Fused* and *unrolled* avoid this overhead by storing the result of the operation into the memory no longer needed by one of the operands.

For *chaining*, the whole trace-collection and hash table mechanism is programmed in Forth, and run on Gforth[7], which is not the fastest Forth system around. We expect a slowdown by maybe a factor of five compared to implementing this functionality in C. We expect that the *chaining* overhead would be lower and the crossing points would be reached earlier with a faster implementation, and we discuss below how that would affect the conclusions.

The benchmarks were run on a Core i5-6600K (Skylake) running Debian 8.

Figure 12 shows the instructions needed with AVX, Figure 13 the cycles. In our implementation, *chaining* has 748 instructions (199 cycles) more overhead per vector operation than *simple*, and it takes until vector length > 500 (4KB) before the crossing point is reached. This is also reflected in the cycles result (Fig. 13), where the crossing is also at vector length > 500. For very long vectors, *chaining* saves two thirds of the instructions and half of the cycles compared to *simple*.

When comparing with *fused* and *unrolled*, we see that the crossing points are reached even later, and the benefit of *chaining* after the crossing point is much smaller, especially in terms of cycles.

The benefits of *chaining* are less pronounced when looking at the cycles than when looking at the instructions. Our explanation is that, for this benchmark, machine resource limits make themselves felt.[8]

[7] Commit c9da1b544b25491e8e29ca94dfcd5830a7abf1ef, engine `gforth-fast`.

[8] One seemingly obvious resource is L3 cache bandwidth; we tried to address that by implementing some kind of cache-blocking; but while we achieved a reduction in

6 PRELIMINARY RECOMMENDATIONS

Recommendations based on a single benchmark are not very trustworthy, so take the following with a large grain of salt. But based on the data above, our recommendations are:

The overhead of trace collection and hash table lookup for *chaining* is big. Even if we manage to reduce the overhead by a factor of 4, *chaining* is attractive only for long vectors (> 1KB). However, if you have long vectors, the speedups over *simple* can be substantial (factor 2).

If you are implementing vectors as a library, you should first implement *simple*. If you then want better performance for long vectors, implement *chaining*, but use it only for vectors beyond a certain length; you will have to determine the switching point empirically for your implementation.

If you can afford to integrate vector-loop fusion within basic blocks into your JIT compiler, this option is preferable over *chaining*. If you can afford to unroll loops or perform other control-flow transformations to extend the scope of loop fusion, even more so.

If you have a tracing JIT compiler, you use a *simple* implementation in the interpretive part. When recording the trace, ideally you record the vector operations instead of the loops that implement them, and you record the vector lengths for these vector operations. Once you have a complete trace, you can compile vector operations with the same length (relative to each other) into a a loop as outlined in Section 4. The length of the input vectors is then compared on every execution as one of the guards of the trace. Such a compiler could provide the benefits of *chaining* (combining many vector operations in one loop) without the overhead of having to build a trace on every execution.

7 CONCLUSION

Software vector chaining collects a trace of vector operations at run-time and then performs a fused loop for these vector operations. This loop is generated on first execution, using relatively simple compiler techniques; for later executions of traces with the same structure, the code is looked up in a hash table.

Software vector chaining does not attempt to compete with HPC compilers, but is instead a relatively simple technique that can be implemented in a vector library for any kind of language implementation. It offers substantial performance benefits over simple vector loops when dealing with long vectors.

However, software vector chaining incurs quite a bit of overhead per vector operation, and therefore is not beneficial when dealing with short vectors. Fortunately, the decision whether to use simple vector loops or chaining can be made based on the actual vector length encountered at run-time, separatly for each vector operation.

The basis for being able to use software vector chaining is opaque vector types. They provide the abstraction and isolation (in particular from non-vector memory accesses) that is necessary to rearrange the computations as software vectory chaining does.

ACKNOWLEDGMENTS

The reviewers provided valuable feedback on the paper.

L3 cache accesses, our attempts produced slowdowns. Apparently the bottleneck is elsewhere.

REFERENCES

[AK87] Randy J. Allen and Ken Kennedy. Automatic translation of Fortran programs to vector form. *ACM Transactions on Programming Languages and Systems*, 9(4):491–542, 1987.

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers. Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[BDB00] Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. Dynamo: A transparent dynamic optimization system. In *SIGPLAN '00 Conference on Programming Language Design and Implementation*, pages 1–12, 2000.

[Cha81] Alan E. Charlesworth. An approach to scientific array processing: The architectural design of the AP-120B/FPS-164 family. *Computer*, pages 18–27, September 1981.

[Cra77] Cray Research, Inc. *Cray-1 Computer System — Hardware Reference Manual*, 1977.

[Ert17] M. Anton Ertl. SIMD and vectors. In *33rd EuroForth Conference*, pages 25–36, 2017.

[GPF06] Andreas Gal, Christian W. Probst, and Michael Franz. HotpathVM: An effective JIT compiler for resource-constrained devices. In *Virtual Execution Environments (VEE'06)*, pages 144–153, 2006.

[ILSV18] Vladimir Ivanov, Razvan Lupusoru, Paul Sandoz, and Sandhya Viswanathan. JEP draft: Vector API. Technical report, OpenJDK, 2018.

[Rau94] B. Ramakrishna Rau. Iterative modulo scheduling: An algorithm for software pipelining. In *International Symposium on Microarchitecture (MICRO-27)*, pages 63–74, 1994.

[Wol96] Michael Wolfe. *High-Performance Compilers for Parallel Computing*. Addison-Wesley, 1996.