SIMD and Vectors

M. Anton Ertl[*]
TU Wien

Abstract

Many programs have parts with significant data parallelism, and many CPUs provide SIMD instructions for processing data-parallel parts faster. The weak link in this chain is the programming language. We propose a vector wordset so that Forth programmers can make use of SIMD instructions to speed up the data-parallel parts of their applications. The vector wordset uses a separate vector stack containing opaque vectors with run-time determined length. Preliminary results using one benchmark show a factor 8 speedup of a simple vector implementation over scalar Gforth code, a smaller (factor 1.8) speedup over scalar VFX code; another factor of 3 is possible on this benchmark with a more sophisticated implementation. However, vectors have an overhead; this overhead is amortized in this benchmark at vector lengths between 3 and 250 (depending on which variants we compare).

1 Introduction

Current computer hardware offers several ways to perform operations in parallel:

Superscalar execution Independent instructions are executed in parallel, if enough functional units and other resources are available. This requires little programmer intervention: out-of-order processors find independent instructions by themselves.

SIMD instructions perform the same operation on multiple data in parallel. The programmer or compiler has to use these instructions explicitly.

Multi-core CPUs Programs have to be split into multiple threads or processes to make use of this feature.

As a close-to-the-metal language, Forth should provide ways to make use of these hardware features. At least SwiftForth and Gforth already contain features to make use of shared-memory multicores, by extending the classical Forth multi-tasking wordset. Superscalar execution is exploited by the hardware and/or the compiler [SKAH91] without programmer intervention.

In this paper I present the basic concepts and an initial version of a vector wordset (Section 3) that can make good use of SIMD instructions. The main concept and contribution is a vector stack that contains opaque vectors of dynamically determined length. The programmer can use this vector stack to express vector operations in a way that does not introduce additional dependencies, and is therefore the key to allowing very efficient implementations with relatively little compiler complexity, as well as enabling simpler implementations (with less performance). We present different ways to implement this wordset in Section 4, and Section 5 presents preliminary performance results of using this wordset. We discuss related work in Sections 2 and 6.

Terminology: In this paper, *vector* refers to application-level one-dimensional arrays with an arbitrary number of elements, while SIMD refers to what machine instructions offer: arrays with limited (and often fixed) number of elements.

2 Background

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.*

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 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,

[*]Correspondence Address: Institut für Computersprachen, Technische Universität Wien, Argentinierstraße 8, A-1040 Wien, Austria; `anton@mips.complang.tuwien.ac.at`

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

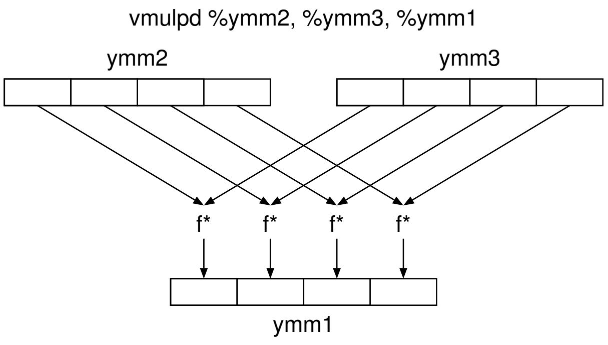


Figure 1: A SIMD instruction: `vmulpd` (AVX)

3DNow, SSE, AVX etc. and ARM incorporated Neon.

These instruction set extensions typically provide registers with a given number of bits (e.g., 128 bits for the XMM registers of SSE and AVX128, and 256 bits for the YMM registers of AVX256), 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).

On the application side, these instructions are usually used for implementing vector operations, such as the inner product.

Making use of these instructions in programs has been a major challenge. The following methods have been used, and Fig. 2 shows examples.

Assembly language allows specifying a specific SIMD instruction directly.

Intrinsics tell the compiler to use specific SIMD instructions; e.g., the intrinsic `_mm256_mul_pd` tells the Intel C compiler to use the `vmulpd` instruction. These intrinsics are just as architecture-dependent as assembly language, but at least they play nicely with the rest of the C code. Other compilers (e.g., gcc) typically support the same intrinsics as the Intel compiler.

Vectors as language feature APL and its modern descendent J have arrays as first-class data type, and many operations that work on arrays or generate arrays. The array sizes are determined at run-time.

```
;Assembly language
vmulpd ymm1, ymm2, ymm3

/* C with Intel Intrinsics */
__m256d a,b,c;
c = _mm256_mm_mul_pd(a, b);

NB. J
a =: 3 5 7 9
b =: 2 4 6 8
c =: a*b

!Fortran Array language
REAL, DIMENSION(4) :: a,b,c
c = a*b;

/* GNU C Vector Extensions */
typedef double v4d
   __attribute__ ((vector_size (32)));
v4d a,b,c;
c = a*b

/* C with auto-vectorization */
double a[4], b[4], c[4];
for (i=0; i<4; i++)
  c[i] = a[i] * b[i];
```

Figure 2: Using SIMD instructions in programs

Modern Fortran contains an array sublanguage that allows the programmer to express various operations on whole arrays and sub-arrays directly instead of through scalar[3] operations in loops; the example in Fig. 2 shows an example that can be directly translated to `vmulpd`, but vectors of any length (including dynamically determined lengths) are supported, as well as higher-dimensional arrays, and parts of arrays.

GNU C contains a simple vector extension[4], usable only with fixed-size vectors with 2^n elements, ideally the size of the SIMD registers (gcc generates relatively bad code for larger vectors). So it mostly is useful as an architecture-independent way to specify SIMD operations, and the programmer should compose the code for longer vectors himself; for run-time determined vector sizes, this is the only option.

Auto-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

[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]In the context of programming with vectors, *scalar* refers to a single value, i.e., a non-vector.

[4]`https://gcc.gnu.org/onlinedocs/gcc/Vector-Extensions.html`

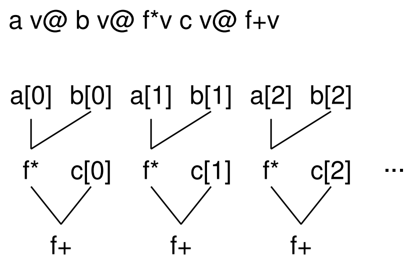


Figure 3: Dependences in a sequence of vector operations. Note that there are no dependences between computations of different elements.

find out by itself how to make use of these instructions for that code.

While auto-vectorization occasionally succeeds in vectorizing a piece of code (especially benchmarks), this 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 to SIMD code also requires much less complexity than auto-vectorization.

Therefore, in this paper I propose an approach to express vector operations in Forth.

3 Forth Vector Wordset

3.1 Vectors

A vector contains a dynamically determined number of bytes; different vectors can contain different numbers of bytes, but many operations require that all operands have the same length (as in APL, J, and Fortran).

Vectors are opaque: They do not reside in memory that an application program is allowed to access. Programmers can easily comply with this restriction: Only access vectors with vector words; moreover, when using this wordset, if you use the wrong words, the error will typically reveal itself quickly.

Vectors have value semantics, like cells or floats, and unlike strings in memory: When you copy a vector, the copy has an identity of its own, and it is unaffected by operations on the original (and vice versa).

The main benefit of these properties is that it gives a lot of freedom to the implementation of vector operations: Every part of every vector is independent of every other part of the same vector, other vectors, and main memory (see Fig. 3), so vectors can be processed in any order: front-to-back, back-to-front, in parallel, in some combination of these methods, or in some other order.

This restriction also gives the implementation a lot of freedom when storing the vector data: It can be stored in main memory with as much alignment and padding as is useful for an efficient implementation; or it can be (partially or fully) stored in SIMD registers, or in, e.g., graphics card memory; the implementation may also store the vector data in a way convenient for calling high-performance computing libraries in other languages (as long as the data used by these libraries are vectors or subvectors).

An alternative approach would represent vectors by an address/length pair, as is done in the standard for strings [For14, Section 3.1.4.2]; this is somewhat similar to Fortran's array language, that also works on the existing arrays in memory. The disadvantage of this approach is that efficient implementation is hard, and in some cases impossible: vectors would not necessarily be aligned for efficient memory access, the size may not be a multiple of the vector register size, and the memory areas of vectors specified in this way may overlap, so the order of element operations of a vector operation would be restricted, it would require significant compiler and/or run-time sophistication to achieve correct operation while using SIMD instructions, and efficiency would suffer as well.

Some have suggested combining the addr/len approach with restrictions on the program to avoid the compiler/run-time complexity and performance disadvantages, but this would add conceptual complexity to the usage of the wordset and very likely lead to non-portable programs, because such restrictions are hard to comply with in every instance: It is hard to find out by testing that you have not complied, unless every restriction is always exploited by the implementation you use. Also, when choosing between portability and performance, programmers will often choose performance (especially when dealing with a performance-enhancing feature).

3.2 Vector Stack

There is a separate vector stack that is used by vector words.

Why have a separate stack instead of just storing single-cell vector tokens on the data stack? Vector tokens on the data stack would be error-prone: it would be natural to use, e.g., `dup` to copy a vector token, or `drop` to get rid of it; this would be incompatible with otherwise attractive implementation options for the value semantics of vectors (see Section 4.1), and, worse, there is no implementation that would be compatible with it that does not use garbage collection.

By contrast, with a separate vector stack, the user has to use `vdup` and `vdrop` to deal with vectors, and these words (and all others dealing with vectore) can perform the bookkeeping necessary for preserving value semantics in the vector implementation.

The usual stack manipulation words get vector equivalents: `vdup vover vswap vrot vdrop vpick vroll`.

Once we have such a vector stack, we can also use it for other data, such as strings, bignums and matrixes, but that is outside the scope of the present work.

3.3 Data types

Forth uses only a few on-stack data type sizes: single-cell, double-cell and float. It uses additional in-memory sizes for communications with other software or hardware, and for making efficient use of memory.

For vectors, we usually also want to use the smallest element data types that are big enough for the application. This allows us to have shorter vectors and faster vector operations. So while on a 64-bit system we have only, e.g., `+` for adding 8-bit, 16-bit, 32-bit and 64-bit integers, for vectors we want `b+v w+v l+v x+v`, because, for long vectors, `b+v` will be 8 times faster than `x+v`.

The vector wordset uses the following prefixes for types: `b ub w uw l ul x ux sf df`.

In this paper, vector stack elements are denoted with `v` for general vectors, or *`type`*`v` for specific types, e.g. `uwv` for a vector of unsigned 16-bit (`uw`) values, and `sfv` for a vector of 32-bit floats.[5] Vector items are always on the vector stack, so this paper does not use a `V:` notation or somesuch for indicating that an item is on the vector stack.

Should we have vector types with cell-sized elements (`v uv`), and vectors with float-sized elements (`fv`)? Vectors with cell-sized elements would be fine, but are not implemented in the current wordset; the vector words dealing with them would be aliases of words dealing with `xv uxv` or `lv ulv` vectors.

Vectors with float-sized elements have the problem, that there are relevant Forth systems where the default float type uses the 80-bit 387 format, and is stored in memory in 10-byte (VFX) or 16-byte (iForth) units. There are no SIMD instructions for dealing with this format, so vector words for them would be slow. So, programmers should use `dfv` or `sfv` words for portability (including performance portability), and implementing `fv` words does not make much sense.

3.4 Vector operation patterns

There are a number of operation patterns when working with vectors:

Parallel vector/vector E.g., adding each element of the first vector to the corresponding element of the second vector, resulting in a third vector. This pattern works only with vectors of the same length. Words implementing this pattern have a `v` suffix.

Parallel vector/scalar E.g., adding a scalar to each element of a vector. Words implementing this pattern have a `vs` or `sv` suffix (`sv` only for non-commutative operations).

Reduce E.g., for the sum or the maximum of all elements of a vector, producing a scalar. Words implementing this pattern have suffix `r`. Currently the vector wordset does not support this pattern.

Generate a vector of a certain length, with all elements containing the same scalar (possibly unnecessary if we have `vs` instructions). Other generating operations are NGSPICE's vector(n) that produces a vector containing 0,1,2..9. Matlab's `linspace(x1,x2,n)` generates n points; the spacing between the points is $(x_2 - x_1)/(n-1)$ (always includes the endpoints). Currently the vector wordset does not support this pattern.

Reorder/shuffle elements of the vector, e.g., for use in a FFT. Certain shuffle operations are supported by SIMD instructions, but they are rather limited. For now, the vector wordset will not support this pattern.

Compress Given a vector of data and a vector of flags, pick only those data corresponding to true flags, and put them in a new (possibly shorter) vector. While this pattern is commonly used in APL, it is not well-supported in SIMD instructions, and the vector wordset will not support it for now.

Scan The APL operator \ produces the intermediate results of reducing a vector. E.g. `+\1 2 3` produces `1 3 6`. The vector wordset will not support this for now.

[5]The standard has `r` for on-stack FP numbers and `f` for flags, but uses the prefixes `f`, `sf`, `df` for dealing with FP numbers of various lengths in memory; should we use `sr` for 32-bit floats?

Index Sometimes the index of the first element satisfying a condition, or the index of the maximum or minimum element is needed. The vector wordset will not support this pattern for now.

3.5 Words

The vector wordset provides `v` (vector/vector parallel) versions of the arithmetic, logic and comparison operations `+ - * / mod negate and or xor invert lshift rshift`[6] `mux`[7] `abs max min < = > <= >= <>`, `vs` (parallel vector/scalar, with the first argument being the vector) versions of `+ - * / mod and or xor lshift rshift arshift max min < = > <= >= <>` (not `negate invert abs`, because they are unary operations, and not `mux`, because it is ternary and scalar operands do not make sense for it), and `sv` (parallel scalar/vector, with the first argument being scalar) versions of the non-commutative words `- / mod lshift rshift arshift < > <= >=`[8]. Signed integer `/ mod` may be symmetric or floored (and not necessarily the same as the scalar / and mod). The result of comparison operations is a vector with elements of the same size as the operand elements (e.g, 8 bits per element for `b<v`); all bits of the result element are 1 if the comparison result is true or 0 otherwise. For bitwise operations (`and or xor invert mux`), no type prefix is used.

For reducing words, one would use the associative binary operations: `+ * and or xor max min`.[9]

The combination of types, operations, and patterns produces a large number of words: In the current implementation, 137 `v` words, 123 `vs` words, and 76 `sv` words. We see the same thing in the SIMD extensions of computer architectures: They introduce a large number of instructions thanks to the combinations of types and operations (and possibly register widths).

3.6 Memory

Having vectors only on the vector stack with no way to move data to/from ordinary memory would be too restrictive for general usage. So the vector wordset provides two ways to deal with memory:

- Have concrete data on the memory side, i.e., a memory range where you can access individual elements with address arithmetics; the alignment of the memory range is not necessarily SIMD-friendly, nor is it padded to a multiple of the SIMD width; the memory after the last element may be inaccessible, so the last element requires special (expensive) treatment even on loading.
- Have opaque vectors in memory, i.e., the same representation as on the vector stack. Opaque vectors make the use of SIMD instructions easy by storing the vectors appropriately, but the implementation of these words has to manage the memory for the vectors with `allocate` or some other dynamic memory management method in order to allow proper padding and alignment, plus management information such as the size of the vector. Our wordset uses single-cell vector tokens.

The words for these memory accesses are:

`b!v ( v c-addr u -- )` store a vector into concrete memory; if the vector length is different from u, an error is thrown.

`b@v ( c-addr u -- v )` load a vector from concrete memory.

`v! ( v v-addr -- )` Store a vector into memory as opaque vector, storing the single-cell vector token into v-addr.

`v@ ( v-addr -- v )` Load a vector from an opaque vector in memory, accessed through the single-cell vector token v-addr.

`v@' ( v-addr -- v )` Fetch v, then clear v-addr. The advantage of this operation over `v@` is that the implementation can just use the existing reference to the opaque vector without incurring the implementation cost of copying the rest of the vector (Section 4.1). Moreover, a later `v!` to `v-addr` does not incur the implementation cost of deleting the vector that `v@` leaves at `v-addr`.

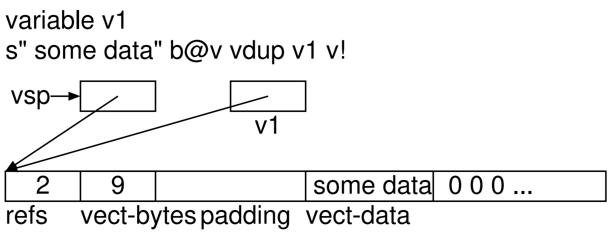
One danger of storing single-cell vector tokens in memory is that this provides a hole for subverting the implementation of value semantics: these tokens can be copied with `@ ! move` etc. If this proves to be a problem, a debugging mode can make these tokens location-dependent by xoring them with v-addr on `v!` and `v@`. This should quickly unveil accidential copying of this kind; there is, of course, no protection against intentional subversion of the implementation in Forth.

[6] Both signed and unsigned shifts right are supported.

[7] `Mux ( x1 x2 x3 -- x )` is a bitwise operation, that selects a bit from x1 if the corresponding bit from x3 is 1, otherwise it selects the corresponding bit from x2.

[8] E.g., `0 1-vs` would have no effect, while `0 1-sv` would be equivalent to `lnegatev`.

[9] `F+` and `f*` do not satisfy the associative law, but `df+r` is useful anyway, because it delivers an approximation to the rational/real value of the computation, just like `f+` and `f*` itself.



```
variable v1
s" some data" b@v vdup v1 v!
```

Figure 4: A vector in our implementation with refcounts (`refs`) after performing the shown code

4 Implementation

This section describes various implementation approaches, as well as the current implementation.

4.1 Vectors

A vector is stored in `allocate`d memory and is aligned and padded to the SIMD granularity (e.g., 32 bytes for AVX256). In front of the actual vector data, there is bookkeeping information: The number of bytes in the vector, and possibly a reference count (see below); in addition, there may be some padding to align the actual vector data, too (see Fig. 4). The address of the start of the `allocate`d memory is the vector token.

There are two ways to implement the value semantics:

linear Every vector has exactly one reference. Copying (`v@ vdup vover`) creates (`allocate`s) a new copy of the complete vector, and consuming or overwriting a token (`vdrop v!` and many other operations) `free`s the vector.[10]

refcount Every vector can have several tokens referring to it; the number of tokens is stored with the vector in a reference count. When copying the token, the reference count is increased, when consuming or overwriting a token, it is decreased; if the reference count reaches 0, the vector itself is `free`d. This reduces the number of `allocate`s and `free`s, at the cost of some additional complexity.

I normally avoid reference counting, because it does not handle cyclic data structures well, but vectors don't contain any pointers to other data at all, and therefore cannot form cycles, so reference counting is ok for this purpose.

In many operations (e.g., `df+v`), one vector (or more) is consumed, and another of the same length is created. Then one can use the memory of a consumed vector for the created one, avoiding the overhead of `free` and `allocate`, unless (in the refcount variant) the consumed vector still has references left.

[10]The name `linear` for this approach is inspired by Henry Baker [Bak94].

```
vec vect-bytes @ vect-data 0 vect-data ?do
  vec1 i + df@ vec2 i + df@ f+ vec i + df!
[ 1 dfloats ] literal +loop
```

Figure 5: The vector-processing loop of the *trivial* implementation of `f+v`. `vec`, `vec1` and `vec2` are locals containing vector addresses, `vect-bytes` and `vect-data` are the fields shown in Fig. 4.

The current implementation supports both approaches (based on a compile-time flag), so the user can determine the efficiency difference himself.

Another alternative that comes to mind is to copy just the references, but use garbage collection for managing the memory. A disadvantage is that words that consume and produce a vector (e.g., `f+v`) cannot reuse the vector memory directly, but would always have to allocate new vector memory. Allocation is about as expensive as with explicit `allocate/free` unless you use a copying garbage collector, and a copying garbage collector has relatively high collection overhead for big data structures (such as potentially our vectors, when they are used for sounds or pictures). Overall, this alternative does not look attractive.

Region-based memory allocation [Ert14] may be useful when dealing with longer-term storage of vectors, but is probably too cumbersome to be used for the memory management of intermediate vector results; also, it is not clear how to make region-based memory allocation work properly with reference counting.

Finally, a compiler that is analytical about the vector operations can avoid the overhead of vector allocation and freeing in many cases.

4.2 Vector stack

Once vectors have been implemented, the vector stack is trivial: It is just a stack of vector tokens. However, unlike in a normal stack of cells, the copying or consuming words have to perform the appropriate copying or deleting of the referenced vectors and/or reference-count bookkeeping.

4.3 Computations

Trivial implementation

The vector words can trivially be implemented in standard Forth with loops containing scalar computation words (see Fig. 5).

While this implementation realizes none of the SIMD speedup that the vector wordset is designed for, it provides a fallback option for users who want

```
simple:
 vmovapd (%rdi,%rax,1),%ymm0
 vaddpd  (%rsi,%rax,1),%ymm0,%ymm0
 vmovapd %ymm0,(%rdx,%rax,1)
 add     $0x20,%rax
 cmp     %rax,%rcx
 ja      simple
```

Figure 6: The vector-processing loop of the *simple* implementation of `df+v`.

```
sophisticated:
 vmulpd  (%rdi,%rax,1),%ymm0,%ymm1
 vaddpd  (%rsi,%rax,1),%ymm1,%ymm1
 vmovapd %ymm1,(%rdx,%rax,1)
 add     $0x20,%rax
 cmp     %rax,%rcx
 ja      sophisticated
```

Figure 7: The vector-processing loop of the *sophisticated* implementation of the sequence `df*vs df+v`.

to write portable code: They can write code using the vector wordset, and still run it (albeit slowly) on Forth systems that do not have a SIMD implementation of the vector wordset. It also provides a gradual approach for SIMD implementations: the implementor can implement the most important operations using SIMD instructions first, still falling back to the trivial implementation for the words he has not implemented yet.

Simple implementation

A simple implementation implements every vector word separately, but uses SIMD instructions for that.

For the vector-parallel `v`, `vs`, and `sv` words, the meat of the word is the *vector loop*: in each iteration, it loads the operand(s) from the `vect-data` memory into SIMD registers, uses a SIMD instruction to perform the operation n times in parallel, and then stores the result back into the memory for a vector (see Fig. 6). For running that on a CPU with in-order execution, you want to software-pipeline [Cha81] this loop for good performance; on out-of-order execution hardware, the hardware reorders the instruction execution by itself, achieving the same result.

For the `r` (reducing) words, the implementation is more involved: Thanks to associativity, there are many different ways to evaluate the result: A very parallel implementation of `+r` divides the vector into pairs of numbers, computes the sum of the pairs, resulting in an $n/2$-sized vector; repeat that until you have only one number left, the result. A very sequential implementation of `+r` would add up all the vector elements, one after the other, incurring $n-1$ times the latency of $+$.

One way to use SIMD instructions for reduction would be to add up SIMD-register-wide parts of the vector in a SIMD register; then each element of the vector occurs exactly once in the computation of exactly one of the components of the SIMD register; finally, the components of the SIMD register are reduced to form the final result. This may not fully utilize the resources of the CPU, especially for FP operations, which have a latency of more than one cycle. More parallelism can be exploited by adding up the elements of the vector in 4 or 8 SIMD registers in parallel (in an unrolled loop), then adding these registers together with SIMD instructions, and finally the components of the resulting SIMD register.

Sophisticated implementation

If we have several vector-parallel words in a Forth-level basic block[11], the simple implementation would produce several loops, with the data stored in memory between the loops, incurring loop overhead, and load and store overhead, and possibly overhead for allocating and freeing vectors for the intermediate results. Instead, several vector-parallel words can be combined into a single loop[12], with the intermediate results only in SIMD registers (i.e., not as full vectors, see Fig. 7).[13]

We can also let reducing vector words participate in this scheme, with some caveats: The result of the reduction must not be used in the same sequence of vector words, so the combining ends with the first word that uses the result of the reduction. And the unrolling that you may want for the reduction would complicate the rest of the code generation; on the other hand, a smaller unrolling factor (even 1) may be sufficient to achieve good performance given that the loop performs not just one reduction, but more.

Letting concrete-memory stores (e.g., `b!v`) participate in the combining also has caveats: If the memory of such a store overlaps the memory of concrete loads or other concrete stores, the result of a naïve combination of vector operations can produce an incorrect result. As a simple example,

```
a 1024 b@v a 64 + 1024 b!v
```

logically has to copy the whole 1024 bytes to the vector stack before it starts storing, but a naïve combining implementation might overwrite `a 64 +`

[11]A basic block is a straight-line code segment.

[12]This is a special case of the general optimization *loop fusion.*

[13]This is similar to *vector chaining* used in hardware-pipelined vector processors such as the Cray-1.

```
simple2:
 vmovapd (%rdi,%rax,1),%ymm0
 vaddpd  (%rsi,%rax,1),%ymm0,%ymm0
 vmovapd %ymm0,(%rdx,%rax,1)
 vmovapd 0x20(%rdi,%rax,1),%ymm0
 vaddpd  0x20(%rsi,%rax,1),%ymm0,%ymm0
 vmovapd %ymm0,0x20(%rdx,%rax,1)
 add     $0x40,%rax
 cmp     %rax,%rcx
 ja      simple2
```

Figure 8: The loop of Fig. 6 unrolled by a factor of 2

before loading this memory location, producing a different result (like the difference between `move` and `cmove`).

One solution to this problem is to check the memory ranges for overlaps before the loop; if there is an overlap, let the loop write to temporary, non-overlapping memory regions, and copy these to the target addresses in the right order after the loop.

Unrolling

By unrolling the vector loop (see Fig. 8), the loop overhead can be reduced for long vectors. It turns out that this does not improve performance significantly on the Core i5-6600K, but it may help on other CPUs.

Unrolling normally has to deal with left-over iterations. In the case of vectors we can avoid that by making the vector data long enough for our preferred unrolling factor (e.g., with 32-byte SIMD instructions and unrolling factor 2, always have multiples of 64 bytes as vector data).

Beyond basic blocks

Extending the combining of vector words beyond basic blocks is possible, but significantly more complex: the vector stack has to be analysed beyond basic blocks, and there are some issues to consider.

For `if`s, one implementation is to pull the `if` inside the loop implementing the combined vector operation; loop unrolling can reduce the number of dynamically executed `if`s (see Fig. 9).

Another way to deal with `if` is if-conversion [MLC+92]: Both branches are computed, and the result is selected with a `muxv` operation. However, in cases where if-conversion is beneficial, I expect programmers to perform it at the source level, so I would not perform this at the compiler level. Also, this is not possible for every operation, in particular not for stores.

If the vector operations are contained in a loop, we can extend the combining by unrolling this loop

```
df+v x 0< if
  a v@ df+v then
0.5e df*vs

# x in %r10, a in %r11, 0.5 in %ymm2
vector_loop:
  vmovapd (%rsi,%rax,1), %ymm0
  vmovapd 0x20(%rsi,%rax,1), %ymm1
  vaddpd  (%rdi,%rax,1),%ymm0,%ymm0
  vaddpd  0x20(%rdi,%rax,1),%ymm1,%ymm1
  test    %r10, %r10
  jns then
  vaddpd  (%r11,%rax,1),%ymm0,%ymm0
  vaddpd  0x20(%r11,%rax,1),%ymm1,%ymm1
then:
  vmulpd  %ymm2,%ymm0,%ymm0
  vmulpd  %ymm2,%ymm1,%ymm1
  vmovapd %ymm0, (%r12,%rax,1)
  vmovapd %ymm0, 0x20(%r12,%rax,1)
  add     $0x40,%rax
  cmp     %rax,%rcx
  ja      vector_loop
```

Figure 9: A vector code fragment containing an if, and a possible way to compile it. The `if` moves inside the vector loop, and loop unrolling (factor 2) is used to reduce its overhead.

```
n2 0 ?do
  b1 i th v@
  a j n2 * i + dfloats + df@
  f*vs f+v
loop

# %ymm3=a[j,i]
# %ymm2=a[j,i+1]
# %rbx=%r9=vtos vect-data
# %r11=b[i]   vect-data
# %r10=b[i+1] vect-data
sophisticated_unrolled:
  vmulpd (%rcx,%r11,1),%ymm3,%ymm0
  vaddpd (%rcx,%rbx,1),%ymm0,%ymm0
  vmulpd (%rcx,%r10,1),%ymm2,%ymm1
  vaddpd %ymm1,%ymm0,%ymm0
  vmovapd %ymm0,(%rcx,%r9,1)
  add    $0x20,%rcx
  cmp    %rcx,%r8
  ja     sophisticated_unrolled
```

Figure 10: A Forth loop containing vector words, and the assembly language for the vector loop (the (outer) do loop is not shown) for two iterations of the do loop (not the vector loop); i.e., the result of unrolling the do loop by a factor of 2.

```
typedef double
  vdf __attribute__ ((vector_size (32)));

static void dfplusv_(vdf*v1,
  vdf*v2, vdf*v, size_t bytes)
{
  size_t i;
  ...
  for (i=0; i<bytes; ) {
    *v = *v1++*v2;
    i+=SIMD_SIZE, v1++, v2++, v++;
  }
}
```

Figure 11: The C-level implementation of the vector loop of `df+v` in Gforth. The resulting assembly code is shown in Fig. 6.

```
genv-binary-c dfplusv_ vdf *v1+*v2
genv-binary df+v dfplusv_ df-type f+
```

Figure 12: Generating words: The first line generates the C function `dfplusv_` shown in Fig. 11 (note how the C expression from this line appears in the function), the second line generates the Forth vector word (including memory management) `df+v`, calling `dfplusv_` if available, otherwise generating a *trivial* implementation that uses `f+`.

(instead of or in addition to unrolling the vector loop). Figure 10 shows the inner do loop of the vector version of matrix multiplication, as well as the vector loop generated from an unrolled (factor 2) do loop body. In addition to reducing the vector loop overhead, the unrolling reduces the number of vtos accesses (only one load and one store vs. two each for the two iterations without unrolling). However, this kind of unrolling requires dealing with left-over iterations.

4.4 Current implementation

The current implementation of the vector wordset supports different implementation options: It supports choosing between *linear* and *refcount* options (independent of the other options), it includes a *trivial* implementation (for all systems that don't have anything better yet), and it has a *simple* implementation for Gforth that is based on GNU C's vector extensions (see Fig. 11). There are further configuration options for this variant: You can define the SIMD size (default 16 bytes), and the vector-loop unroll factor (default 1).

Given the large number of vector words, all following a few patterns, plus these configuration options, the words are not hand-coded, but are instead generated. Figure 12 shows the lines generating `df+v`. The `dfplusv_` function and Forth word is generated only on Gforth. Another system could provide its own (e.g. `code`) version of `dfplusv_` (the core vector loop), and this would then be called by `df+v`, upgrading this word from a trivial implementation to a simple implementation.

```
C_scalar:
 movsd (%rdi),%xmm1
 add   %rsi,%rdi
 mulsd %xmm0,%xmm1
 addsd (%rdx),%xmm1
 movsd %xmm1,(%rdx)
 add   %rcx,%rdx
 sub   $0x1,%r8
 jne   C_scalar
```

Figure 13: The inner loop of the *C scalar* implementation; it allows different strides for the involved vectors and therefore has separate increments for the addresses.

5 Results

This section gives some idea of the speedups achievable by using various vector word implementation approaches. We enhance the existing matrix multiplication code[14] with variants that use the vector wordset.

Note that while the vector wordset shows nice speedups over scalar code on large-matrix multiplication, calling a specialized matrix multiplication library probably shows even better performance, so this is not the ideal application area for vectors; but there are areas where no specialized libraries are available, and the vector wordset can be useful there. Here I use matrix multiplication for benchmarking, because it is vectorizable, because we already have a matrix multiplication benchmark, and because it allows scaling for arbitrary vector lengths.

We compared the following vector implementations and matrix multiplication variants:

trivial Vector words are implemented using scalar Forth words without loop unrolling (Fig. 5).

simple Matrix multiplication uses `f*vs` and `f+v`, each of which is implemented as a separate loop, with the intermediate vector stored in memory. The vector loops are written C with the GNU C vector extension and compiled to use AVX instructions (Fig. 6); the rest of the vector words is written in Forth.

[14] `http://theforth.net/package/matmul`

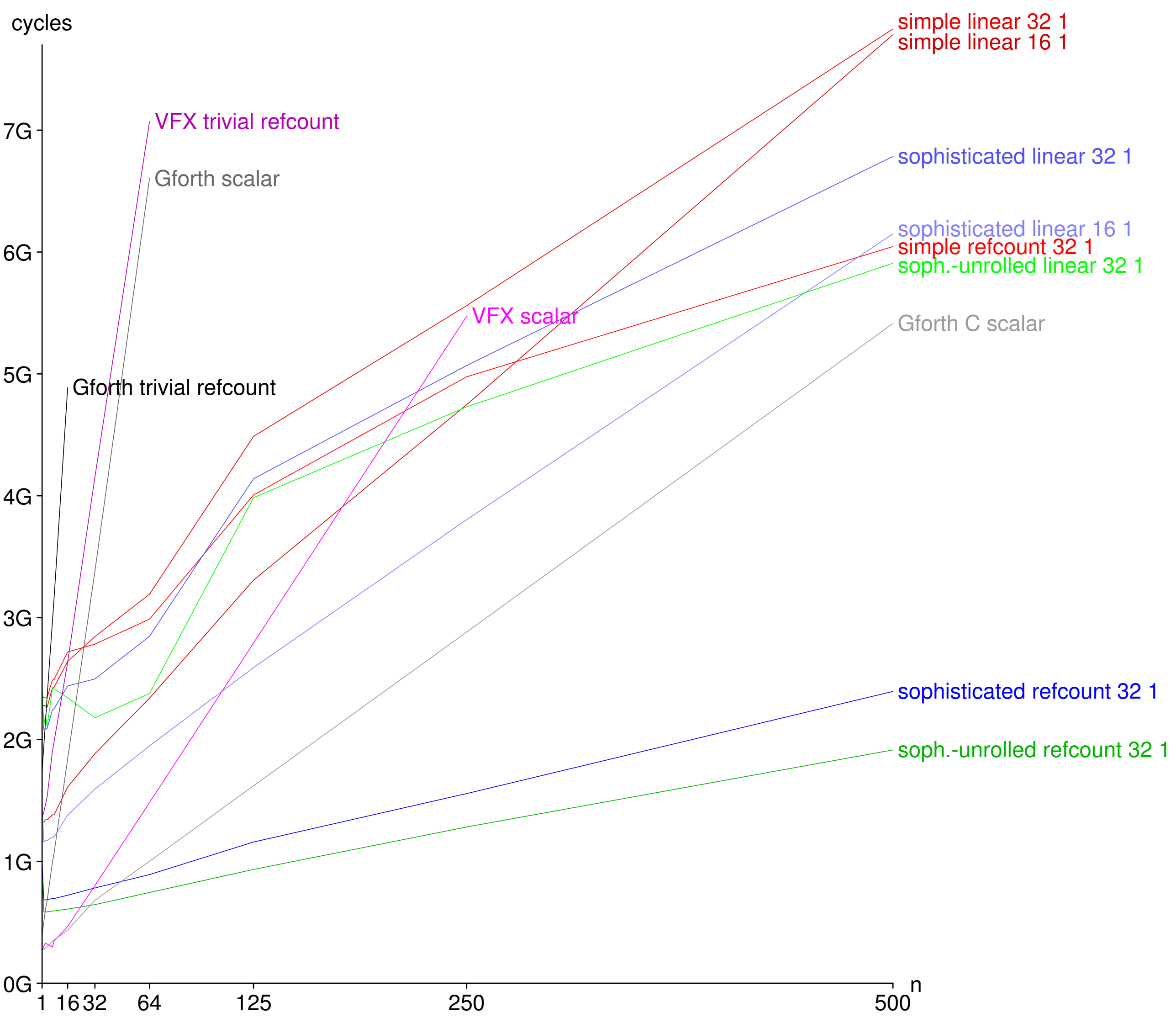


Figure 14: Timings for 20 matrix multiplications, each performing 250,000 times `f*vs f+v` (or equivalent) for n-wide vectors

sophisticated We have no sophisticated compiler, so we fake the effect by implementing a word `f*+vvs` and use that instead of the sequence `f*vs f+v`; `f*+vvs` combines Forth and GNU C in the same way as *simple* (Fig. 7).

soph.-unrolled We fake the effect of a sophisticated compiler with unrolling (factor 2) by writing a word that combines the effect of `v@ f*vs f+v v@ f*vs f+v`, resulting in the code shown in Fig. 10.

scalar The matrix multiplication code written in scalar Forth code, otherwise using the same algorithm. Unlike *trivial*, this version unrolls the loop by a factor of 4, providing a significant speedup on VFX (where the loop counter update otherwise limits performance).

C scalar The inner loop of the matrix multiplication uses scalar C code (Fig. 13). This is mostly useful for determining how good the (Forth) scalar implementation perfroms.

For all the vector (i.e., not scalar) variants, both *linear* and *refcount* was measured. For *simple*, *sophicticated*, and *soph.-unrolled*, SIMD sizes of 16 (AVX128) and 32 (AVX256) were measured, and vector-loop unrolling factors of 1, 2, and 4.

The benchmarks were run on a 4GHz Core i5-6600K (Skylake) running Debian 8 (glibc 2.19). Two Forth systems were used: `gforth-fast` (development version from August 2017) was used for all variants, VFX Forth 4.72 was used for *trivial* and *scalar*.

The benchmark multiplies a 500×500 matrix with a $500 \times n$ matrix for varying n; given the algorithm of the benchmark, this always produces 250,000 instances of `f*vs f+v` (or the scalar equivalent), with vector length n. I.e., for all n the overheads were the same. For a single matrix multiplication, compiling the vector wordset takes longer than some of the benchmark instances, so the benchmark performs 20 such matrix multiplications to mitigate this effect. Overall, the benchmark loads 10M$\times n$ FP values, stores 5M$\times n$ FP values, performs 5M$\times n$ FP additions and 5M$\times n$ FP multiplications.

Figure 14 shows a selection of the results. Showing all results would have overloaded the graph, so we only show the most relevant ones, but also discuss the other results here.

Unrolling the vector loop had little effect on performance, so here we show only unrolling factor 1.

SIMD size did not have the big effect I expected. SIMD size 16 (AVX128) was often slightly slower than SIMD size 32 (AVX256), but occasionally faster (some cases where it is faster are shown in Fig. 14). SIMD size 32 apparently produces some non-linear effects, especially for the variants that `allocate` and `free` memory, so I suspect some interference between AVX256 and the memory allocator. The AVX128 variants of the same benchmarks are closer to linear.

Refcount clearly beats *linear* for this benchmark, across all vector sizes. Apparently the overhead of `allocate` and `free` is much larger than that of reference counting, and for the larger vector sizes, you also have to pay for copying quite a bit of vector data on each `v@`. Still, the slowdown of *soph.-unrolled linear 32 1* over *soph.-unrolled refcount 32 1* is surprisingly large; one contributing factor is probably that *sophisticated* and *soph.-unrolled* with *refcount* doe not perform a single `allocate` or `free` in the core of the matrix multiplication. For the trivial implementations, we show only the *refcount* variants; the *linear* variants are slightly slower.

Given that much of the time is apparently spent in `allocate` and `free` for many of the results, repeating the benchmark on a platform with a different implementation of these words might give quite different results; in particular, a per-thread cache has been added to `malloc()` in glibc 2.26.[15] In the present case all the `allocate`d and `free`d vectors have the same size, so such a cache should work very well. One could also add such a cache to the vector wordset implementation, to reduce the performance dependency on the underlying `allocate` and `free` implementation.

Overall, as expected, for the larger vector sizes we have a big performance increase from *trivial* through *simple*, *sophisticated* up to *soph.-unrolled*, with the *scalar* results being between *trivial* and *simple*. The performance of *C scalar* is interesting, because it is faster than everything except *sophisticated* and *soph.-unrolled* (at small vector sizes, *C scalar* beats even them); however, when programming in Forth, we usually don't have a scalar C version at hand, so the (Forth) *scalar* results are more relevant.

By looking where the *scalar* lines cross those of various vector implementations, we can determine at what vector length using vector words starts paying off. For *Gforth scalar*, the crossover point with *soph.-unrolled refcount 32 1* is at vector length 3, with *sophisticated refcount 32 1* at vector length 4, *simple refcount 16 1* is faster at vector length 16 and *simple refcount 32 1* is faster at length 32.

VFX scalar is quite a bit faster, crossing over *soph.-unrolled refcount 32 1* only between vector lengths 16 and 32, and crossing over *simple refcount 32 1* between 125 and 250. However, these vector implementations all run on Gforth, and I expect that a SIMD-based vector implementation in VFX will run faster and reach crossover sooner.

Overall, we see that the vector words can provide a speedup, especially if the scalar code is not compiled optimally (whereas the inner loops of the vector words can be written in assembly language). However, vector words have an additional overhead, and that means that, for short vector lengths, using the vector words will produce a slowdown.

6 Related work

Related work in other languages has been introduced in Section 2.

The most significant difference between the vector wordset and the Fortran array sublanguage is that our vectors are stored separately from ordinary memory, avoiding alias problems, whereas the Fortran array sublanguage operates on arrays and subarrays that can be accessed in other ways, too, and therefore has to worry about alias problems.

The most significant difference between GNU C's vector extensions and the vector wordset is that GNU C's vector types have a fixed size that is restricted to be a power of 2, and in practice should be the same as the SIMD size; so it is essentially a way to express SIMD operations without resorting to architecture-specific intrinsics or assembler. In contrast, the vector words process vectors of arbitrary length, which does not even have to be a multiple of the SIMD length, thus providing a higher-level programming interface.

APL is much more sophisticated than the vector words, and includes operations that do not benefit from current SIMD instructions, and are hard to implement efficiently. If the vector words become popular and such features are asked for, the vector wordset may grow in the direction of APL in the future. Of course, you can instead use APL or J

[15] `https://lwn.net/Articles/729761/`

today (but then have to live without the features of Forth).

Closer to Forth, there is a Forth dialect designed for genetic programming that includes vector and matrix operations [HRvR07] in order to let the genetic programming system discover programs that benefit from such operations, such as signal processing. The Forth dialect uses a combined stack for all the types (including vectors and matrices), static type checking, and overloading resolution. Apart from that, the paper is very superficial in its description of the vector words. Our vector words are oriented towards the traditional Forth model of not performing type checking. The separate vector stack is a direct consequence of this model, especially because we want to treat vectors as opaque data type (unlike, traditionally, strings [Ert13]) to avoid aliasing.

7 Conclusion

By having opaque vectors and a wordset for them, we can make use of SIMD instructions without unreliable compiler complications such as alias analysis or auto-vectorization. The wordset can be implemented in different ways: The *simple* implementation is easy to implement; its performance for large vector sizes is better in our benchmark than using scalar Forth code on VFX. The *sophisticated* implementation provides a better speedup, but requires more implementation effort. The source code can be found on `https://github.com/AntonErtl/vectors`.

Acknowledgments

Herbert Pohlai provided valuable knowledge about APL and J. Marcel Hendrix and the anonymous reviewers provided valuable feedback on the paper. Marcel Hendrix provided information on further *generate* patterns.

References

[Bak94] Henry Baker. Linear logic and permutation stacks — the Forth shall be first. *ACM Computer Architecture News*, 22(1):34–43, March 1994.

[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.

[Ert13] M. Anton Ertl. Standardize strings now! In *29th EuroForth Conference*, pages 39–43, 2013.

[Ert14] M. Anton Ertl. Region-based memory allocation in Forth. In *30th EuroForth Conference*, pages 45–49, 2014.

[For14] Forth 200x Standardization Committee. *Forth Standard 2012*, 2014.

[HRvR07] Kenneth Holladay, Kay Robbins, and Jeffery von Ronne. FIFTH$^{\mathrm{TM}}$: A stack based GP language for vector processing. In Marc Ebner et al., editor, *Genetic Progamming (EuroGP)*, pages 102–113. Springer LNCS 4445, 2007.

[MLC^{+}92] Scott A. Mahlke, David C. Lin, William Y. Chen, Richard E. Hank, and Roger A. Bringmann. Effective compiler support for predicated execution using the hyperblock. In *25th Annual International Symposium on Microarchitecture (MICRO-25)*, pages 45–54, 1992.

[SKAH91] Mark Smotherman, Sanjay Krishnamurthy, P. S. Aravind, and David Hunnicutt. Efficient DAG construction and heuristic calculation for instruction scheduling. In *MICRO-24, 24th Annual Intl. Symp. on Microarchitecture*, pages 93–102, 1991.