Compiling Array Languages for SIMD

V. E. McHale

February 8, 2025

Abstract

The C compilation model has dominated compilers literature but it opposes Single-Intruction, Multiple Data (SIMD). We articulate the assumptions behind textbook methods, where new CPUs diverge, and give a first cut of a better approach used in the Apple Array System. By reconsidering expectations about compiler optimization, we give a program for high-level languages to offer performant SIMD. Array languages turn out to be an especially good candidate, expressing efficient operations in a high-level language; we give examples in Apple.

Contents

1 Background

1.1 Textbook Compilation

C implementations compile the language's constructs (functions) to sequences of steps in assembly which read and write registers for arguments and return values defined by convention. The correspondence is elegant [1], but not suited to SIMD.

Consider a compiler written as mutually recursive functions:

[1] Kell [3] points out that C is uniquely suited to interfacing with other languages, in part because of this compilation practice.

```
writeExpr :: Expr -> Temp -> [Stmt]

writeFn :: Expr -> [Temp] -> Temp -> [Stmt]
```

The first function takes an expression and a register and creates a series of statements that will write that its value to the given register. The second takes an expression representing a function, registers to read arguments from, and a register to write the return value, and returns a series of statements. This is used in Appel [1][pp. 148-169], for instance.

Naïvely[2], we might try to compile `λas. map (λx. f x) as`, which maps over a vector, as follows:

- Generate temporaries `x`, `y` for the argument and return value of `f`, respectively.
- Compile `λx. f x` to `fsteps` using `writeFn`.
- Generate code to loop over the input vector's elements; within the loop:
  - Generate code to read from the input array at the current index, putting the value in `x`
  - Place the statements `fsteps`
  - Append code to store the result `y` in the output array.

This is appealing because of its simplicity, and also has the advantage that `f` can be defined in another language. However in the case of

```
λas. map (λx. x*2) as
```

it would step over the input vector's elements one at a time. This is not as efficient as it could be—SIMD instructions can multiply several elements at once. Thus compiling `map` should not resort to a single canonical compiled version of `f`.

1.2 Functions in Theory

Landin [5] points out that Algol corresponds to the lambda calculus. Functions in the lambda calculus have a formal definition and are applicable to programming in practice, but only some have efficient SIMD implementations: `λx. 1-x^2` and `if x≥0 then x else 0` are both functions but the latter involves a conditional which in general is compiled to a jump; they must be treated differently during compilation but are not distinguished in the source language.

Cataloguing which functions can be compiled with vector instructions would be a welcome contribution to the compilers literature.

[2]This was the approach taken in the Apple array system at first.

1.3 Limits of C

What makes C special in practice is that functions, which can be composed, are compiled to sequences of assembly instructions stored in object files; compositionality is borne through until linking. This compositionality is at odds with SIMD, where a high-level function `λx. x*2` should be compiled using different instructions depending on whether it is lifted to operate on an array. macOS provides

```
float expf(float x);
vFloat vexpf(vFloat x);
void vvexpf(float *y, const float *x, const int *n);
```

as three shared library functions to address this. A library providing a sigmoid function (for instance) would need to expose three versions in a shared library. The clumsiness proliferates and the mathematical entity no longer corresponds to compiled code but rather three versions of itself depending on the context in which it is used.

A compiler-as-a-library is appealing because it could be invoked by another compiler, dispatching different versions of a function without intervention from the programmer. Alternately new formats for object files could include several versions of a function, with information that other languages' compilers could use to pick the appropriate implementation.

2 Efficient Compilation

The full employment theorem for compiler writers states that it is impossible to write an algorithm that transforms an arbitrary program into an optimal version. Of course, such a "big hammer" would be handy, but the problem sitting before us is narrower. A compiler ferries a (possibly high-level) language to efficient code, and actually a source language with fewer low-level facilities offers more invariants. "Optimization" writ large gets much attention, but transformations that make high-level languages possible[3]have not entered the computer science canon, particularly for functional or exotic languages.

"Amenability to efficient compilation" also informs the design of the source language. One can also consider languages where the programmer can expect efficient compilation by reasoning about the source-language in its own terms. This is not the case for C++, where efficiency of generated code can vary dramatically by compiler or across versions of the same compiler [6, 4, 2]. There is no contract with the user; performance depends on internals that are not documented, justified, analyzed with the same scrutiny as programming languages.

Array languages have a chance to do better since they naturally express computations that can be executed via SIMD, in a domain where such CPU features offer significant gains.

[3]Without deforestation, `fold (+) 0 [1..1000]` would build up the whole list `[1..1000]` in memory (in a strict language) [8]—deforestation is necessary for this style with higher-order functions to stand toe-to-toe with a direct implementation.

2.1 Typing & Indexing

Suppose we wish to compute the 7-day moving average. We will do so using \`, inspired by J's infix [7], which lifts a vector function to operate on windows of 7-vectors, arranging the (scalar) results into a vector. Examining type annotations, we see that the fold of the inner loop (/) is inferred have type `(float → float → float) → Vec 7 float → float`:

```
 > :ann λxs. [(+)/x%ℝ(:x)]\`7 xs
λ(xs : Vec (i + 7) float).
    ((\`7 : (Vec 7 float → float) → Vec (i + 7) float → Vec (i + 1) float)
     λ(x : Vec 7 float).
         ((% : float → float → float)
          ((/ : (float → float → float) → Vec 7 float → float)
           (+ : float → float → float)
           (x : Vec 7 float))
          ((ℝ : int → float) ((: : Vec 7 float → int) (x : Vec 7 float))))
     (xs : Vec (i + 7) float))
```

and in fact the inner loop is compiled without parity checks:

```
 > :asm λxs. [(+)/x%ℝ(:x)]\`7 xs

    ⋮
    ldr d2, [sp, x2, LSL #3]
    fadd d5, d5, d2
    add x4, x4, #0x1
apple_1:
    mov x2, #0x10
    add x2, x2, x4, LSL #3
    ldr q2, [sp, x2]
    fadd v3.2d, v3.2d, v2.2d
    add x4, x4, #0x2
    cmp x4, x3
    b.LT apple_1
    faddp d2, v3.2d
    fadd d5, d5, d2
    ⋮
```

Compare the code generated to compute the average of a vector of unspecified size; note the `tbz x1, #0, apple_0` to handle the case when the elements cannot be divided exactly into vector registers.

```
 > :ty [(+)/x%ℝ(:x)]
Vec (i + 1) float → float
 > :asm [(+)/x%ℝ(:x)]
```

```
    ⋮
    ldr d3, [x2], #0x8
    eor v2.16b, v2.16b, v2.16b
    mov x4, #0x1
    cmp x4, x3
    b.GE apple_1
    sub x1, x3, #0x1
    tbz x1, #0, apple_0
    ldr d0, [x2], #0x8
    fadd d3, d3, d0
    add x4, x4, #0x1
apple_0:
    cmp x4, x3
    b.GE apple_1
    ldr q0, [x2], #0x10
    fadd v2.2d, v2.2d, v0.2d
    add x4, x4, #0x2
    cmp x4, x3
    b.LT apple_0
apple_1:
    faddp d0, v2.2d
    fadd d3, d3, d0
    ⋮
```

Similar analyses are already performed by industrial compilers, but types are part of the source language and thus available to any programmer who knows the language—contrast autovectorization in C++, which is a matter for compiler experts. Moreover, type systems, unlike compiler internals, are not subject to change between versions.

In a type system with shapes, dimension information proliferates (based on rules). As an example, in the Apple expression

```
λwh.λwo.λbh.λbo.
{ X ← ⟨⟨0,0⟩,⟨0,1⟩,⟨1,0⟩,⟨1,1⟩⟩;
  Y ← ⟨0,1,1,0⟩;
  Σ ⇐ [(+)/o x y];
  sigmoid ← [1%(1+e:(_x))];
  sDdx ← [x*(1-x)];
  ho ← sigmoid`{0} ([(+)`(bh::Vec 2 float) x]'X%.wh);
  prediction ← sigmoid ∴ (+bo)'ho%:wo;
  l1E ← (-)`Y prediction;
  l1Δ ← (*)`(sDdx'prediction) l1E;
  he ← l1Δ (*)⊗ wo;
  hΔ ← (*)`{0,0} (sDdx`{0} ho) he;
  wha ← (+)`{0,0} wh (|:X%.hΔ);
  woa ← (+)`wo (|:ho%:l1Δ);
```

```
  bha ← Σ`{0,1} bh hΔ;
  boa ← Σ bo l1Δ;
  (wha,woa,bha,boa)
}
```

all dimensions are known to the compiler, with only one type annotation and `X`, `Y` being supplied as constants. The programmer can verify this in the Apple array system with

```
 > :ann xor
```

and supply hints as type annotations if necessary or desired. This mode of interaction is entirely absent from industrial compilers—optimizations are confined to compiler internals, at best one can dump information and then adjust flags or add pragmas. These are peculiarities of the compiler rather than language, and the user's control of the compilation output is indirect.

3 A First Cut at a Solution for the Apple Array System

SIMD in the Apple array system is not God-anointed but provides a starting example for array compiler authors.

3.1 Map

Apple adds functions

```
write2 :: Expr -> Temp2 -> [Stmt]

writeFn2 :: Expr -> [Temp2] -> Temp2 -> [Stmt]
```

which compile expressions such as `λx. x*2` to statements operating on vector registers.

Then, when compiling, say, `map f`, we use

```
hasSIMD :: Expr -> Bool
```

to determine whether `f` can operate on vector registers. `hasSIMD` traverses the AST, returning `False` when, say, `log` is found. If the function has a SIMD implementation:

- Generate temporaries `x0`, `y0` and vector temporaries `x`, `y` for arguments and return values of `f`, respectively.
- Compile `λx. f x` twice, once with `writeFn` and once with `writeFn2`.
- Generate code to loop over the input array, increasing index by 2 each time; within the loop

  - Generate code to read two values from the input array at the current index, storing in `x`
  - Place the statements `fsteps2`
  - Append code to store the two result values in `y` in the output array.

- Generate code to conditionally execute if there is still one element remaining in the input array after iterating by 2. Inside the conditional:
  - Generate code to read `x` from the input array.
  - Place the statements `fsteps`
  - Append the code to store the result `y` in the output array.

If the function does not have a SIMD implementation, we fall back on compiling `map` as described in §1.1

3.2 Fold

Suppose we are compling `fold (λx.λy. op x y) seed`. At a high level, the Apple array system attempts a SIMD implementation as follows:

- Determine how to seed the vector register from a scalar value by inspecting `op`. For instance, in the case of `max`, we fill all elements of the vector accumulator `acc` with the scalar seed; in the case of `(+)`, we zero the vector accumulator and then write the seed value to one of its elements.
- Iterate over the input array by 2, reading to the vector argument `x` and executing `fsteps2 = writeF2 op [x, acc] acc` to update the vector accumulator.
- Combine the elements of the vector accumulator using `op`, writing the result to the return register `t`. Often there is a specialized instruction, e.g. `faddp`, `haddpd`, `fmaxp`.
- If there is still one remaining element after iterating by 2, read it a register `x0` and update the return register by executing `fsteps = writeF op [x0, t] t`

The Apple array system defines the helper functions:

```
seedVector :: Expr -> Maybe (FTemp -> F2Temp -> [Stmt])

combine :: Expr -> F2Temp -> FTemp -> [Stmt]
```

`seedVector` generates a sequence of `Stmt`s to seed a given vector register from a value in a register when possible. If the operation is not recognized, we give up and return `Nothing`.

`combine` generates code to apply `op` to the vector elements of a given register, writing the result to a specified register.

Acknowledgments

Thanks to Marshall Lochbaum for suggested improvements.

References

[1] Andrew Appel. *Modern Compiler Implementation in ML*. Cambridge University Press, 1998.

[2] hl3mukkel. Why does this code snippet produce radically different assembly code in c and c++? `https://stackoverflow.com/q/48837118/11296354`, 2018.

[3] Stephen Kell. Some were meant for c: the endurance of an unmanageable language. In *Proceedings of the 2017 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, Onward! 2017, page 229–245, New York, NY, USA, 2017. Association for Computing Machinery.

[4] Marek Kolodziej. Matrix multiplication on cpu. `https://marek.ai/matrix-multiplication-on-cpu.html`, 2019.

[5] P. J. Landin. Correspondence between algol 60 and church's lambda-notation: part i. *Commun. ACM*, 8(2):89–101, February 1965.

[6] Dan Luu. Assembly v. intrinsics. `https://danluu.com/assembly-intrinsics/`, October 2014.

[7] Bob Therriault, Henry Rich, and Ian Clark. Vocabulary/bslash. `https://code.jsoftware.com/wiki/Vocabulary/bslash`, 2023.

[8] Philip Wadler. Deforestation: transforming programs to eliminate trees. *Theoretical Computer Science*, 73(2):231–248, 1990.