

[Yet Another Lambda Blog](#)

because inside every programmer there is a haskeller trying to get out

- [Home](#)
- [About](#)

[Haskell as fast as C: A case study](#)

By [Jan Stolarek](#), 02/04/2013

Once in a while someone, most likely new to Haskell, asks how does Haskell performance compare with C. In fact, when I was beginning with Haskell, I asked exactly the same question. During last couple of days I've been playing a bit with squeezing out some performance from a very simple piece of Haskell code. Turned out that the results I got are comparable with C so I thought I might share this. This will be a short case study, so I don't intend to cover the whole subject of Haskell vs. C performance. There was a lot written on this already so I encourage to search through the Haskell-cafe archives as well as some other blogs. Most of all I suggest reading [this](#) and [this](#) post on Don Stewart's blog.

Here is my simple piece of code:

```
sumSqrL :: [Int] -> Int
sumSqrL = sum . map (^2) . filter odd
```

It takes a list of `Int`s, removes all even numbers from it, squares the remaining odd numbers and computes the sum. This is idiomatic Haskell code: it uses built-in list processing functions from the standard Prelude and relies on function composition to get code that is both readable and modular. So how can we make that faster? The simplest thing to do is to switch to a more efficient data structure, namely an unboxed `Vector`:

```
import Data.Vector.Unboxed as U

sumSqrV :: U.Vector Int -> Int
sumSqrV = U.sum . U.map (^2) . U.filter odd
```

The code practically does not change, except for the type signature and namespace prefix to avoid clashing with the names from Prelude. As you will

see in a moment this code is approximately three times faster than the one working on lists.

Can we do better than that? Yes, we can. The code below is three times faster than the one using `Vector`, but there is a price to pay. We need to sacrifice modularity and elegance of the code:

```
sumSqrPOp :: U.Vector Int -> Int
sumSqrPOp vec = runST $ do
  let add a x = do
    let !(I# v#) = x
        odd#     = v# `andI#` 1#
    return $ a + I# (odd# *# v# *# v#)
  foldM` add 0 vec -- replace `with ' here
```

This code works on an unboxed vector. The `add` function, used to fold the vector, takes an accumulator `a` (initiated to `0` in the call to `foldM``) and an element of the vector. To check parity of the element the function unboxes it and zeros all its bits except the least significant one. If the vector element is even then `odd#` will contain `0`, if the element is odd then `odd#` will contain `1`. By multiplying square of the vector element by `odd#` we avoid a conditional branch instruction at the expense of possibly performing unnecessary multiplication and addition for even elements.

Let's see how these functions compile into Core intermediate language. The `sumSqrV` looks like this:

```
$wa =
  \vec >
    case vec of _ { Vector vecAddressBase vecLength vecData ->
      letrec {
        workerLoop =
          \index acc ->
            case >=# index vecLength of _ {
              False ->
                case indexIntArray# vecData (+# vecAddressBase index)
                of element { __DEFAULT ->
                  case remInt# element 2 of _ {
                    __DEFAULT ->
                      workerLoop (+# index 1) (+# acc (*# element element));
                    0 -> workerLoop (+# index 1) acc
                  }
                }
              };
            True -> acc
          }; } in
      workerLoop 0 0
    }
```

while `sumSqrPOp` compiles to:

```
$wsumSqrPrimOp =
```

```

\ vec ->
runSTRep
( (\ @ s_X1rU ->
  case vec of _ { Vector vecAddressBase vecLength vecData ->
    (\ w1_s37C ->
      letrec {
        workerLoop =
          \ state index acc ->
            case >=# index vecLength of _ {
              False ->
                case indexIntArray# vecData (+# vecAddressBase index)
                  of element { __DEFAULT ->
                    workerLoop
                      state
                        (+# index 1)
                        (+# acc (*# (*# (andI# element 1) element) element))
                      };
              True -> (# state, I# acc #)
            }; } in
          workerLoop w1_s37C 0 0)
      })
  )

```

I cleaned up the code a bit to make it easier to read. In the second version there is some noise from the ST monad, but aside from that both pieces of code are very similar. They differ in how the worker loop is called inside the most nested case expression. First version does a conditional call of one of the two possible calls to `workerLoop`, whereas the second version does an unconditional call. This may seem not much, but it turns out that this makes the difference between the code that is comparable in performance with C and code that is three times slower.

Let's take a look at the assembly generated by the LLVM backend. The main loop of `sumSqrV` compiles to:

```

LBB1_4:
    imulq    %rdx, %rdx
    addq     %rdx, %rbx
.LBB1_1:
    leaq     (%r8,%rsi), %rdx
    leaq     (%rcx,%rdx,8), %rdi
    .align   16, 0x90
.LBB1_2:
    cmpq     %rax, %rsi
    jge      .LBB1_5
    incq     %rsi
    movq     (%rdi), %rdx
    addq     $8, %rdi
    testb    $1, %dl
    je       .LBB1_2
    jmp      .LBB1_4

```

While the main loop of `sumSqrP0p` compiles to:

```
.LBB0_4:
    movq    (%rsi), %rbx
    movq    %rbx, %rax
    imulq   %rax, %rax
    andq    $1, %rbx
    negq    %rbx
    andq    %rax, %rbx
    addq    %rbx, %rcx
    addq    $8, %rsi
    decq    %rdi
    jne     .LBB0_4
```

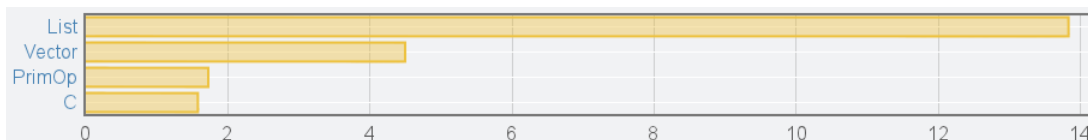
No need to be an assembly expert to see that the second version is much more dense.

I promised you comparison with C. Here's the code:

```
long int c_sumSqrC( long int* xs, long int xn ) {
    long int index = 0;
    long int result = 0;
    long int element = 0;
Loop:
    if (index == xn) goto Return;
    element = xs[index];
    index++;
    if ((0x1L & element) == 0) goto Loop;
    result += element * element;
    goto Loop;
Return:
    return result;
}
```

You're probably wondering why the hell did I use `gotos`. The reason is that the whole idea of this sum-square-of-odds function was taken from the paper [“Automatic transformation of series expressions into loops”](#) by Richard Waters and I intended to closely mimic the solution produced by his fusion framework.

I used criterion to compare the performance of four presented implementations: based on list, based on vector, based on vector using `foldM+primops` and C. I used FFI to call C implementation from Haskell so that I can benchmark it with criterion as well. Here are the results for a list/vector containing one million elements:



C version is still faster than the one based on primops by about 8%. I think this

is a very good achievement given that the version based on Vector library is three times slower.

A few words of summary

The [vector](#) library uses stream fusion under the hood to optimize the code working on vectors. In the blog posts I mentioned in the beginning Don Stewart talks a bit about stream fusion, but if you want to learn more you'll probably be interested in two papers: [Stream Fusion. From Lists to Streams to Nothing at All](#) and [Haskell Beats C Using Generalized Stream Fusion](#). My `sumSqrPop` function, although as fast as C, is in fact pretty ugly and I wouldn't recommend anyone to write Haskell code in such a way. You might have realized that while efficiency of `sumSqrPop` comes from avoiding the conditional instruction within the loop, the C version does in fact use the conditional instruction within the loop to determine the parity of the vector element. The interesting thing is that this conditional is eliminated by `gcc` during the compilation.

As you can see it might be possible to write Haskell code that is as fast as C. The bad thing is that to get efficient code you might be forced to sacrifice the elegance and abstraction of functional programming. I hope that one day Haskell will have a fusion framework capable of doing more optimisations than the frameworks existing today and that we will be able to have both the elegance of code and high performance. After all, if `gcc` is able to get rid of unnecessary conditional instructions then it should be possible to make GHC do the same.

A short appendix

To dump Core produced by GHC use `-ddump-simpl` flag during compilation. I also recommend using `-dsuppress-all` flag, which suppresses all information about types - this makes the Core much easier to read.

To dump the assembly produced by GHC use `-ddump-asm` flag. When compiling with LLVM backend you need to use `-keep-s-files` flag instead.

To disassemble compiled object files (e.g. compiled C files) use the `objdump -d` command.

Update - discussion on Reddit

There was some discussion about this post on [reddit](#) and I'd like to address some of the objections that were raised there and in the comments below.

Mikhail Glushenkov pointed out that the following Haskell code produces the same result as my `sumSqrPOp` function:

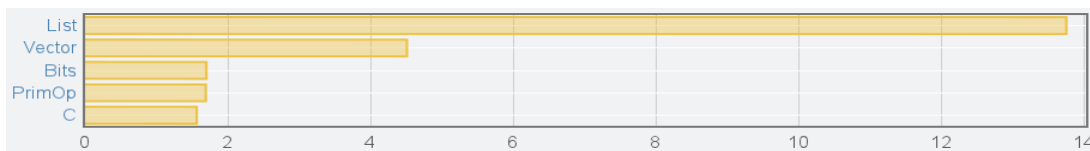
```
sumSqrB :: U.Vector Int -> Int
sumSqrB = U.sum . U.map (\x -> (x .& 1) * x * x)
```

I admit I didn't notice this simple solution and could have come with a better example were such a solution would not be possible.

There was a request to compare performance with idiomatic C code, because the C code I have shown clearly is not idiomatic. So here's the most idiomatic C code I can come up with (not necessarily the fastest one):

```
long int c_sumSqrC( long int* xs, long int xn ) {
    long int result = 0;
    long int i = 0;
    long int e;
    for ( ; i < xn; i++ ) {
        e = xs[ i ];
        if ( e % 2 != 0 ) {
            result += e * e;
        }
    }
    return result;
}
```

The performance turns out to be the same as before ("Bits" represents Mikhail Glushenkov's solution, "C" now represents the new C code):



There was a suggestion to use the following C code:

```
for(int i = 0; i < xn; i++) {
    result += xs[i] * xs[i] * (xs[i] & 1);
}
```


Author claims that this code is faster than the version I proposed, but I cannot confirm that on my machine - I get results that are noticeably slower (2.7ms vs 1.7ms for vectors of 1 million elements). Perhaps this comes from me using GCC 4.5, while the latest available version is 4.8.

Finally, there were questions about overhead added by calling C code via FFI. I was concerned with this also when I first wanted to benchmark my C code via FFI. After making some experiments it turned out that this overhead is so small that it can be ignored. For more information see [this post](#).


Tags: [benchmarking](#), [c](#), [ghc](#), [llvm](#)

 [haskell](#)


12 Responses to “Haskell as fast as C: A case study”

1.  *Mikhail Glushenkov* says:
[02/04/2013 at 17:22](#)

Can't you just use something like ``sum . map (\x -> (x .&. 1) ^2)`` in the first example?

2.  *Mikhail Glushenkov* says:
[02/04/2013 at 17:38](#)

Sorry, I meant ``sum . map (\x -> (x .&. 1) * x * x)``. But you probably got the idea. This produces the following inner loop: <https://gist.github.com/23Skidoo/5293171>, which looks a lot like the Core produced from your `runST` code. Haven't measured, though.


3.  *Jonathan* says:
[02/04/2013 at 18:25](#)

If you didn't notice, your blog post was posted at reddit.com/r/haskell.

4.  *Jan Stolarek* says:
[02/04/2013 at 18:36](#)


Mikhail > you're right. I tested the performance of your code and it is exactly the same as mine solution. Looks like I could have chosen a better example. I've been working on a couple of such algorithms and among them are such that cannot be written with your approach (in fact they required adding new primops to the compiler).

Jonathan > yeah, I noticed :) But thanks anyway.

5.  *Robert Harper* says:
[02/04/2013 at 18:56](#)

The original code using “lists” is a red herring that makes Haskell look worse than it really is. The fact is that Haskell doesn't have a type of lists, only of streams, so the passage to finite vectors is more than just an “optimization”, it's a fundamental change of semantics. It's important to

compare comparable things to be credible. The rest of the results are entirely consistent with the situation with ML for the last 25+ years. The idea that ML or Haskell is somehow automagically “slower than C” is largely bullshit. However, there are situations in which the run-time system of Haskell or some versions of ML can really get in the way; these don’t come into play here. BTW, your example of vectors is pretty much a variant of what Mark Hayden and co showed many years ago in Ensemble (packet processing in network protocols) and what we demonstrated with the Fox Project even longer ago than that. But of course few people were listening, because it was axiomatic that functional languages are irrelevant. Instead we wasted decades on object nonsense while we the enlightened honed our tools. We win in the end.

6.  *Mikhail Glushenkov* says:
[02/04/2013 at 19:12](#)


Looking forward to your next post.

7.  *Jan Stolarek* says:
[02/04/2013 at 21:30](#)


Robert > list were actually last-minute addition to my post, just for the sake of showing “idiomatic” Haskell code. Thanks for pointing out the older stuff. I wasn’t aware of that.

8.  *Greg Fitzgerald* says:
[11/04/2013 at 20:25](#)


Have you tried comparing the performance of the C code using Clang built against the same version of LLVM that GHC’s LLVM backend uses? And then compare to Clang 3.2 and determine of GHC needs to follow suit!

9.  *Jan Stolarek* says:
[11/04/2013 at 22:19](#)


Nope, I haven’t tried that.

10.  *John* says:
[08/09/2013 at 18:45](#)

I don’t think any C compiler knows how to optimize gotos due to the fact that gotos are rarely used.

11.  [Franklin Chen](#) says:
[26/09/2015 at 15:47](#)

Can you provide a link to a code repo that is set up to run these benchmarks so that I can reproduce and inspect the results today, 2 years later?

12.  [Jan Stolarek](#) says:
[26/09/2015 at 18:11](#)

Sadly, there is no repo. Even more surprisingly, I can't actually find the code on my hard drive. But if you really need it I suppose I could just reconstruct it based on the information in the post. Then again, you can probably do the same...

Leave a Reply

Name (required)

Mail (will not be published) (required)

Website



CAPTCHA Code(required)

Add your Reply

• Archives

- ► [2015 \(6\)](#)
- ► [2014 \(9\)](#)
- ► [2013 \(20\)](#)
- ► [2012 \(40\)](#)

• Search

Search for:

• Links

- [Planet Haskell](#)
- [Learn You A Haskell](#)
- [Real World Haskell](#)
- [Haskell Wiki](#)
- [Lambda The Ultimate](#)

• Categories

- [coursera](#) (4)
- [general](#) (4)
- [linux](#) (4)
- [programming](#) (55)
 - [dependent types](#) (10)
 - [haskell](#) (46)
 - [matlab](#) (1)
 - [scheme](#) (2)
- [resources](#) (16)
 - [books](#) (7)
 - [online resources](#) (9)
- [theory](#) (8)

• Tags

[agda](#) [automata](#) [benchmarking](#) [c](#) [cabal](#) [coinduction](#) [combinatory logic](#)
[compilers](#) [coq](#) [cuda](#) [eclipse](#) [eclipsefp](#) [emacs](#) [erlang](#) [ffi](#) [folds](#) [games](#) [ghc](#)
[haskell](#) [hcar](#) [hunit](#) [IDE](#) [idris](#) [java](#) [lambda calculus](#) [leksah](#) [llvm](#) [matlab](#) [prolog](#)

[quickcheck](#) [guine](#) [racket](#) [recursion](#) [repa](#) [ruby](#) [scala](#) [scheme](#) [sml](#) [stg](#) [template-haskell](#)
[testing](#) [types](#)

- **Other**



visits so far.

Yet Another Lambda Blog © Jan Stolarek 2012-2014. Powered by [Wordpress](#)

Staypressed theme by [Themocracy](#)