Utrecht University

Graduate School of Natural Sciences

Computing Science

Investigating the Performance of the Implementations of Embedded Languages in Haskell

---

Author
Bart Wijgers

Supervisors
Dr. Trevor L. McDonell
Prof. Dr. Gabriele K. Keller

July 5, 2022

Contents

Abstract

Accelerate is a language for high-performance computing embedded in Haskell. Embedding languages in other languages is a popular approach for language prototyping, as it reduces the language front-end work from the language designer. To execute the programs written in these languages, the designers need to implement at least an interpreter or a compiler. Tree-walking interpreters are known to be slow, and so different approaches might be implemented if performance is important. This thesis implements a tree-walking interpreter, a bytecode interpreter and a JIT compiler using LLVM in Haskell for a standalone language and compares the performance of each to an optimized bytecode interpreter written in C. However, the results indicate that tree-walking interpreters are often less than an order of magnitude slower and occasionally even faster than bytecode interpreters when both are written in Haskell.

Based on this result, it seemed unlikely that Accelerate's performance issues, especially in a multi-threaded environment, were caused by its tree-walking interpreter. In light of this, I investigated this performance deterioration using a nanosecond-precision profiler and varying heap sizes for the Haskell garbage collector. The results from this show that garbage collection has a large impact on the behaviour of Accelerate programs, and that the performance of these programs likely depends on overhead relating to this. However, the amount of garbage collection occurences is identified to not be the direct cause of the performance degradation. I suggest future work to see what can be done to investigate this problem further and how to fix these performance issues.

Acknowledgements

A huge “thank you” to:

My supervisors, dr. Trevor McDonell and prof. dr. Gabriele Keller, for their many rounds of feedback on this thesis, and for guiding me through the various stages of this research, the focus of which has shifted a few times over the past few months.

All of the Accelerate team, for the mutual support and expertise on Accelerate overall.

Tom Smeding, for making the `ghc-gc-hook` package, and helping me write a lot of the benchmarking and related statistics in this thesis, among which the “other” bytecode interpreter. And an additional thank you to him for the minor problems he helped me out with during this project.

My girlfriend, Kaia, for her support during the setbacks of this project.

1 Introduction

Accelerate [5] is a high-performance library for the Haskell programming language that executes array computations. It defines an *Embedded Domain-Specific Language* [8], or EDSL, with different functional-style combinators like map, fold, and stencil. Using these combinators, Accelerate's compiler tries to optimize the computations as much as possible before running them in the runtime system. The library supports different architectures, among which (multi-core) CPUs and GPUs.

In some cases, Accelerate's performance is comparable to hand-optimized code, but at other times it performs clearly less optimally. For instance, when running a dot product between two vectors on the CPU, the library performs similarly to hand-optimized C code on a low number of threads. Similarly, running a larger program with multiple iterations, such as LULESH, is often faster than running hand-optimized C code.[1] Conversely, running the highly parallizable dot product program on multiple threads is slower than running it on a single thread.

Both maintainers and users benefit from Accelerate being an EDSL. Benefits include:

- The language can easily interface with other code written in Haskell.
- The language does not have to define its own type system, and can instead rely on Haskell's type system.
- Accelerate has the same syntax as Haskell, which makes it easy to learn for Haskell programmers.

The Accelerate language is designed to compose together different functional-style combinators, facilitating the efficient computation and easy design of data-parallel algorithms. The library compiles the code by generating *LLVM*,[2] which can compile into native code; this can be done both using regular compilation and Just-In-Time compilation [2], or JIT compilation for short.

Running the Accelerate EDSL is managed by the runtime system. This system implements certain behaviours that the compiler expects to exist when the program is run. It manages, among other things, memory (de)allocation and variable declarations and references. The runtime also manages threads for parallel execution of Accelerate programs. Through these threads, it also takes care of running kernels, like the *dot* program illustrated in Listing 1a.

The code in Listing 1b implements the same functionality as the code in Listing 1a, but in the C programming language, rather than Accelerate. The C function takes two arrays of floats and the amount of elements to process. C arrays do not keep track of their sizes, so this parameter must be explicit in C, whereas it is implicit in the Accelerate version which does keep track of this information. The `for`-loop in the C code implements both the `zipWith` and the `fold` operations. The complete types of the array parameters are somewhat esoteric, but the `const` keyword makes sure the compiler knows that the arrays' values are constant, and the `restrict` keyword indicates that these arrays do not overlap. The addition of these keywords allows the compiler to properly optimize and vectorize the loop.

[1] https://github.com/tmcdonell/lulesh-accelerate
[2] https://llvm.org/

```
dot
  :: Acc (Array (Z :. Int) Float)
  -> Acc (Array (Z :. Int) Float)
  -> Acc (Array Z Float)
dot xs ys = A.fold (+) 0 (A.zipWith (*) xs ys)
```

(a) The dot product function written in Accelerate.

```
float dot(const float* restrict lhs,
      const float* restrict rhs,
      size_t length) {
  float result = 0.0F;
  for (size_t x = 0; x < length; ++x) {
    result += lhs[x] * rhs[x];
  }

  return result;
}
```

(b) The dot product function written in C.

Listing 1: Functions that run a dot product between two equal-length vectors of arbitrary size.

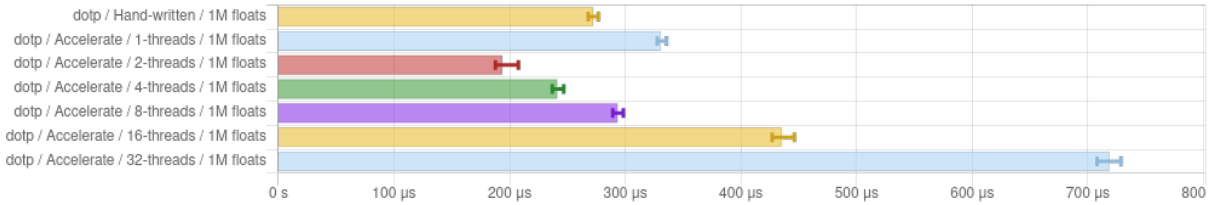


Figure 1: Benchmark results for Listings 1b and 1a. Only the Accelerate listing has results for multiple threads.

The Accelerate library generates native code for programs written in its EDSL, much like C does. When compiling the `dot` functions shown here, the result is very similar native code. As a result, one would expect that both the Accelerate and C programs run with the same performance on a single thread. Additionally, since the function is very parallelizable (as none of the loop iterations affect other iterations), one would expect that adding more threads makes the function faster, with a speedup expected to be around linear in the total amount of threads. Only the code in Listing 1a has results for the multithreaded version; the Accelerate version can be run in parallel by the runtime without changing the code, as the library has parallelized implementations for these combinators, whereas the C-version needs to be changed. EDSLs have a great advantage in this regard over a language like C, as embedded languages can more easily change how they execute their programs without changing the user's code.

Looking at the results of the benchmarks in Figure 1, two observations about the current state of Accelerate stand out:

- While it is still slightly slower, the generated (single-threaded) code for the dot product is very competetive with the hand-written C version, but the source code is much shorter.
- The runtime system has some issues running the computation in parallel with more than 2 threads.

The last observation is concerning, especially since the program is relatively simple. However, the simplicity of this program also means that certain latency and other overheads cannot be amoritized. These problems might be less concerning on larger programs.

The problem with parallel computation does not just arise in simple programs like the dot product function, but also appears in much larger programs such as LULESH. Figure 2 shows an excerpt from the output of a profiler on a run of LULESH. The image shows

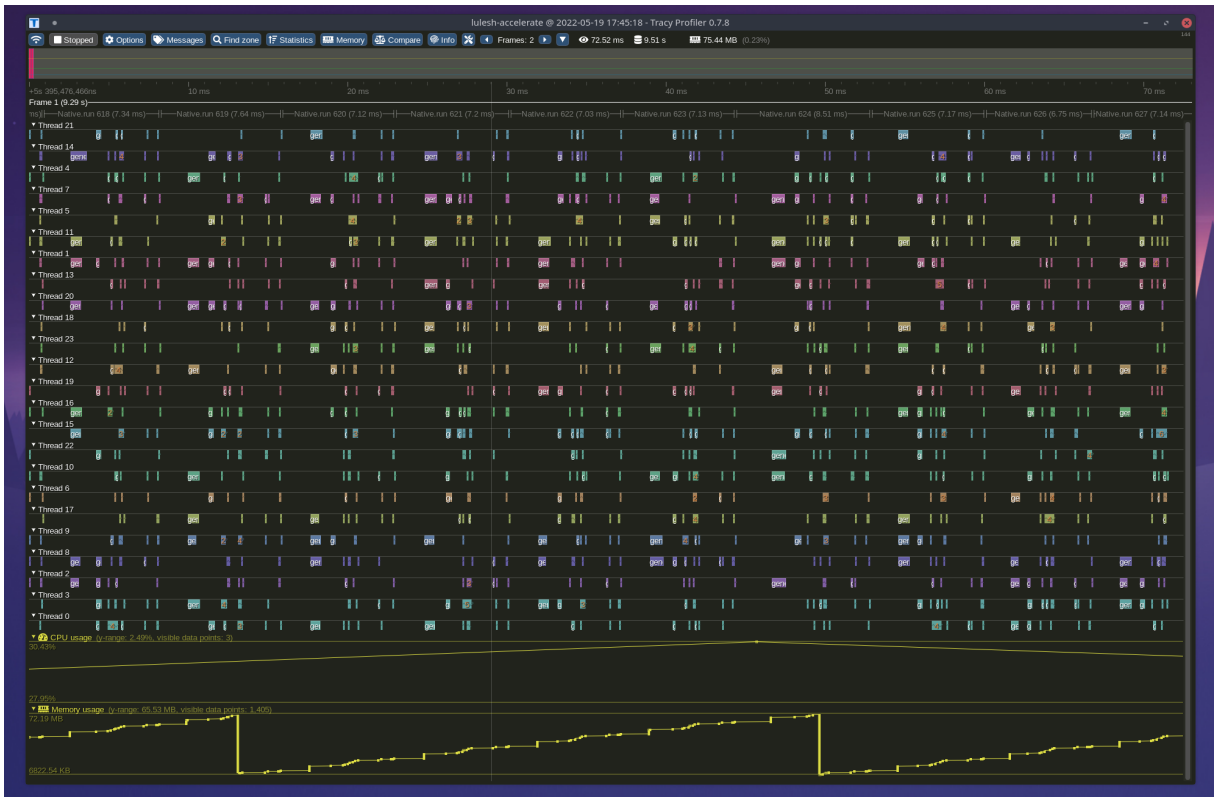


Figure 2: An excerpt from the execution of LULESH in Accelerate. The colored blocks indicate when the worker threads are performing kernel executions.

that there are large gaps in the work that the worker threads perform. At first glance, it looks like the time spent working is somewhere between 1% and 10%, and the rest of the time is unaccounted for. Even accounting for additional bookkeeping that Accelerate does between kernel calls, it looks like a lot of this time is wasted doing unknown work. If it is possible to eliminate the cost of these "gaps" in the work, Accelerate might see a 10-100x speedup!

It should be clear by now that the current implementation of Accelerate has some clear issues with regards to its goal, which is to be a high-performance library. It seems like single-threaded Accelerate will still see a clear speedup if the time between kernel calls is improved; looking at the previous benchmark results in Figure 1, this might still account for about 14% of the time spent. Multi-threaded Accelerate also has some clear issues with performance. Why Accelerate slows down when adding more threads is currently unknown, but parallelism seems to be a likely culprit. This thesis goes into detail on how to investigate and solve these issues.

Moreover, EDSLs like Accelerate are often implemented in languages like Haskell, that have support for higher-order functions and a rich type system [16]. These features reduce the heavy lifting required by language designers to implement a language, allowing them to make use of the host language's type system, parser, and other compiler front-end. However, Haskell is not known to be particularly efficient, especially concerning memory[3]. The data in this reference suggests that equivalent programs written in C and Haskell often see C being faster, and using less memory. We will investigate, among other aspects

[3] https://benchmarksgame-team.pages.debian.net/benchmarksgame/fastest/ghc-gcc.html

of Accelerate, how this embedded, high-performance language's performance is influenced by its Haskell implementation.

This thesis' findings indicate that interpretation and parallelism are likely not the main culprits that reduce performance in Accelerate. The most likely cause of the performance issues in Accelerate seems to be Haskell's garbage collector, and as such, indicate that interpretation is only a problem in the sense that it causes garbage collection in Haskell. The garbage collector also causes interference with multithreading in a way that makes running on more threads result in poorer performance. I propose future work on how to fix these issues, both in ways within Haskell and using an external language.

This thesis investigates how to improve the performance of EDSLs written in Haskell, with a focus on Accelerate as a case-study. More specifically, the following research questions will be investigated:

1. How does performance differ between full compilation and interpretation in Accelerate's runtime system?
2. How does Accelerate's parallellism affect the runtime performance of the library as a whole?

To this end, the rest of the thesis continues as follows. Section 2 gives more background information on Accelerate and interpreters, as well as challenges involving changing Accelerate's runtime system. Section 3 discusses the main aims of the research in more details. Section 4 investigates the performance of interpreters and JIT compilers in Haskell. Section 5 discusses the effect of Haskell's garbage collector on the performance of Accelerate, and the interplay between this and interpretation. Section 6 contains an outline of possible ways to improve the current situation. Finally, Section 7 gives some concluding remarks.

2 Background

As shown in the introduction, Accelerate struggles with certain workloads and with running programs on multiple threads. The goal of this thesis is to mitigate this problem. First, however, more background information will be given. Section 2.1 discusses what exactly an EDSL is, and 2.2 goes into more detail about what the Accelerate EDSL is. Then Section 2.3 discusses current issues with profiling Accelerate and how to improve this situation, before Section 2.4 gives a short introduction to Garbage Collection. Section 2.5 gives an example of different kinds of interpreters and a compiler; this section illustrates the differences between these approaches using a simple example EDSL. Finally, Sections 2.6 and 2.7 go into more detail of the benefits of a bytecode interpreter and a compiler for the Accelerate runtime system specifically.

2.1 EDSL

An EDSL is an *Embedded Domain-Specific Language*. This definition consists of two distinct parts: a domain-specific language and the fact that it is embedded. Each of these pieces is explained in this section.

A DSL is a language that lives at a higher level of abstraction than most general-purpose programming languages [8]. These languages allow users to express algorithms and/or data in a more concise, yet more specific manner. However, these languages also cannot express anything outside of the domain as easily. For instance, the `dot` function we saw earlier is much easier to express in Accelerate than it is in C, as we have seen. However, writing a parser in Accelerate would be much harder than it is in C, if it is even at all possible. This shows how a DSL is designed for a specific goal, whereas general-purpose languages can perform any task.

An embedded DSL is a language that is defined within another language. For example, Accelerate is an embedded language, and the `dot` function in Listing 1a from the introduction is an expression in this language. These languages come in two flavours: a language is either shallowly embedded, immediately constructing the result of the embedded expression, or it is deeply embedded, creating some form of an AST, or Abstract Syntax Tree [7], using data types in the host language [10]. The result of the former would be a textual value like the equivalent C code, whereas the latter would likely generate a tree such as the one in Figure 3. A language with a shallow embedding is harder to optimize [10] or transform in other ways before outputting the result. Using a deeply embedded language, however, one has all the information on the language that is normally acquired during language parsing. We explore ASTs in more detail in Section 2.5.

2.2 Accelerate

Accelerate is a library that allows users to express data-parallel programs in a DSL which is deeply embedded in Haskell, designed for (multi-dimensional) arrays, and the library takes care of compiling efficient code, with the possibility of doing this at run-time. The user writes code in a high-level style in a language with a strong type system, the goal of which is to allow them to quickly write code that is correct and maintainable, without sacrificing performance. It can also handle programs that are partially written in standard Haskell and partially in Accelerate, allowing a user to only write Accelerate code for the parts of their code that require the added heavy lifting that the library offers. The main

aim of the library is to allow users to write code in a functional, high-level style without having to worry about complex data-races or other low-level problems.

2.3 Profiling Accelerate

To be able to perform the experiment, Accelerate needs to have some way of profiling the time it runs non-kernel code. For this project, the kernel code will remain (largely) untouched. Instead, the goal is to improve the performance of executing the code *between* kernel calls.

Accelerate currently does have performance profiling for kernel code. It uses *Tracy*[4] for this. To profile the code and correctly find results, it is important to add profiling information to other categories of code as well, most notably:

- Scheduling new (sub-)tasks into the concurrent queue, which also includes marshalling arguments and allocating new buffers.
- Waiting on the lock of the concurrent queue.
- The work performed running code between taking a work-item off the queue and running the kernel, as well as identifying what code is *actually* run in this time.

Without this, measuring what exactly limits the performance of the runtime will be all the more difficult.

2.4 Garbage Collection

Writing a program in a language like C, where memory management must be done explicitly by the programmer, programmers need to implement algorithms such that this happens correctly. If they do not do this, memory leaks [21, 20] may occur. Garbage Collection, or GC, is a collection of techniques that relieve programmers of this burden, instead using an automatic reclamation scheme [20]. Using GC does come at a performance cost [4], which has been attributed in the past to be around 10% [20]. However, it allows programmers to write their code more quickly, assisting them in focusing on (other) performance constraints of programs [20]. Additionally, some garbage collectors can even *improve* the performance of certain programs [20], by improving data locality

[4] https://github.com/wolfpld/tracy

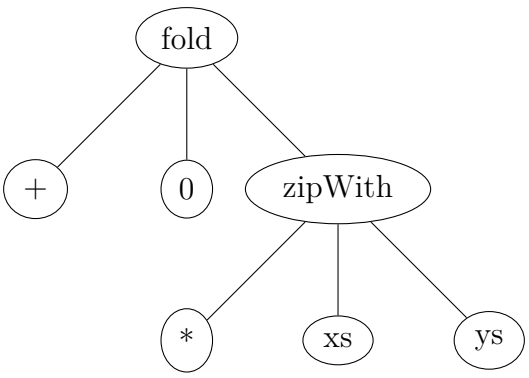


Figure 3: The AST for the dot product function from the introduction, Listing 1a.

```haskell
data ArrProg res where
  Fold    :: ArrProg (res -> a -> res) -> res
          -> ArrProg [a] -> ArrProg res
  ZipWith :: ArrProg (a -> b -> res) -> ArrProg [a]
          -> ArrProg [b] -> ArrProg [res]
  List    :: [res]       -> ArrProg [res]

  Add :: Num res => ArrProg (res -> res -> res)
  Mul :: Num res => ArrProg (res -> res -> res)

dotp :: ArrProg Int
dotp = Fold Add 0
    (ZipWith Mul (List [1..10]) (List [2, 4..20]))
```

Listing 2: A minimal array-program DSL called ArrProg. This DSL implements precisely enough operations to compile the dot-product program of Listings 1a and 1b.

and cache efficiency [9]. However, as we will see later, GC techniques can be detrimental to high-performance computing, as is the case in Accelerate, because of the reduced control a programmer has over the environment.

2.5 Interpreters and Full Compilation

Suppose we want to write a program that computes the dot product between two vectors. Writing a program that does this is relatively simple, as stated in the introduction. However, it is interesting to see how a language can be designed to implement this, and how the language's runtime can be designed to execute such a program, as well as the trade-offs in these cases based on complexity and performance. As such, we will see the design of a language that we will execute in different ways.

The code in Listing 2 defines a small DSL in Haskell that allows users to define a subset of array programs, and an example of a program in this language. This language is used to illustrate the differences between the different interpreters and compilers in the rest of this section.

The main goal of this section is to show the main trade-offs between compilers and different kinds of interpreters, and not to show how to properly implement these for a fully-fledged programming language. As such, this section does not go into detail about how the language should behave, and whether the implementation of the language is correct. Instead, it focuses on the performance and code size of each approach.

The rest of this section introduces compilers and different kinds of interpreters. Two well-known types of interpreters are tree-walking interpreters and bytecode interpreters [12]. This section explains different trade-offs between these programs, starting with the easiest to implement and moving to the hardest one, mentioning other benefits and problems that each approach has over the other ones.

2.5.1 Tree-Walking Interpreters

A tree-walking interpreter is a high-level program that traverses some tree-like representation of the source code of a program, often an AST [12, 7], to execute the code. Due to the nature of these programs, they are often slow compared to the alternatives [2], whilst relatively easy to implement [15]. This section dives deeper into why this is the case.

A tree-walking interpreter for the *ArrProg* language recursively iterates over the nodes

```haskell
runArrProg :: ArrProg res -> res
runArrProg (List xs) = xs
runArrProg Add = (+)
runArrProg Mul = (*)
runArrProg (Fold f a t) =
  foldl (runArrProg f) a (runArrProg t)
runArrProg (ZipWith f xs ys) =
  zipWith (runArrProg f) (runArrProg xs) (runArrProg ys)
```

Listing 3: A tree-walking interpreter for the ArrProg language.

in an AST, and execute any subexpressions in this way, before bubbling up back to the original expression [12]. Take the expression for `dotp` in Listing 2 for example: in order to evaluate the `Fold` statement, the `ZipWith` statement needs to be evaluated, and in order to do that, the `List` statements need to be evaluated. When the inner expressions are evaluated, the recursive algorithm evaluates the outer expression next. The way in which this program is evaluated is by using a postorder tree traversal [1] over the AST, which is why this kind of interpreter is known as a *tree-walking* interpreter.

Listing 3 illustrates how to implement a tree-walking interpreter for the *ArrProg* language. A downside of this approach is that each of these AST nodes stores each of its subtrees as separate pointers in this implementation, which means that the interpreter needs to do many scattered memory accesses to evaluate this program. As this scattered memory access pattern causes cache misses, this algorithm is inherently slower than implementations with more predictable access patterns [19].

2.5.2 Bytecode Interpreters

Bytecode interpreters are a middle ground in executing a language in many ways. They are faster than tree-walking interpreters, but slower than fully compiled programs [2]. However, even though they are slower than than fully compiled programs, they have some benefits over them. We discuss these benefits and the performance differences, as well as an example implementation in Haskell, in this section.

Bytecode interpreters convert the user's code into another format before executing it. This format consists of a set of instructions, each of which often a byte in size, which represent instructions for a virtual machine; hence the name bytecode. The benefit of this translation is that the resulting code can be run more efficiently than using a tree-walking interpreter [2]. One reason for this is that the access pattern during execution of a bytecode interpreter is more predictable than with tree-walking interpreters, as the only time that the code is not traversed linearly in a single direction is when a jump-instruction is executed.

As a first attempt to write bytecode for this language in Haskell, one might come up with code similar to that in Listing 4. This code defines the bytecode instructions as Haskell datatypes, and generates a list of these instructions by iterating over the AST. However, this is not actually beneficial to running the interpreter; each value of type `ArrProgBCInsn` requires at least one pointer in its representation, and the list representation, used in the return type of `toBytecode`, requires two pointers per list item: one for the head, and another for the tail. This means that the main performance detriment described in the section on tree-walking interpreters is not actually addressed with this implementation. Writing idiomatic Haskell is not always the best way to solve a problem, especially when it comes to data locality.

```haskell
data ArrProgBCInsn a
  = BCPushList [a]
  | BCZipWith
  | BCFold a
  | BCAdd
  | BCMul

toBytecode :: ArrProg a -> [ArrProgBCInsn a]
toBytecode (List xs) = [BCPushList xs]
toBytecode Add = [BCAdd]
toBytecode Mul = [BCMul]
toBytecode (ZipWith  f xs ys) =
       toBytecode xs ++ toBytecode ys
    ++ [BCZipWith]   ++ toBytecode f
toBytecode (Fold     f b  xs) =
    toBytecode xs ++ [BCFold b] ++ toBytecode f
```

Listing 4: A possible definition of a bytecode for the ArrProg language.

To fix these issues, one can use a contiguous array, like an `IOArray` in Haskell. Reading and writing in these arrays is similar to reading and writing arrays in C, which fixes the pointer chasing issues that the previous listing and the tree-walking interpreter have. As such, the code in Listing 5 defines a bytecode-compiler that results in a much better memory access pattern. Since the main idea of generating this bytecode relates to using a contiguous array, an approach that automatically transforms Haskell's algebraïc data types into a packed representation, such as the work by Vollmer et al. [19], can also be used. So far, we have only looked at the code that generates the bytecode. The next paragraph discusses the bytecode interpreter, which executes the code.

The bytecode interpreter is defined in Listing 6. In addition to improving the access pattern, this code also defines its own explicit variable stack and program counter, rather than purely relying on the host language's. This interpreter also uses its own explicit stack and program counter, rather than relying on the host language's. If the user was allowed to declare variables in this language, their values would also be pushed on the stack. This method of variable lookup is also faster than the one described for tree-walking interpreters, as this only pushes and pops from a contigious array, if implemented properly.

As we have seen, bytecode interpreters are often faster than tree-walking interpreters. The algorithm used is very different, as illustrated by the code in the listings. Note that, although bytecode interpreters are supposed to be faster, the algorithm requires a lot more code, and more complex code as well. In short, tree-walking interpreters are simpler, but slower, than bytecode interpreters [2, 12].

2.5.3 Compilers

In general, compilers are programs that translate code from one language into another. In this thesis, however, "compiler" will generally refer to programs that translate into *machine code*, i.e. code that can run directly on hardware. When a compiler is done processing the source program, the result is a program that runs without the need for an interpreter or the original source code. We discuss compilers, and the differences between them and interpreters, in this section.

One of the inherent differences between compilers and interpreters is portability [13]. As mentioned, the result of a compiler is code that can run directly on the underlying hardware, without the need for another program in between. This comes at a cost, however;

```
pushList, zipWith, fold, add, mul :: Word8
pushList = 0
zipWith = 1
fold = 2
add = 3
mul = 4

data Bytecode a = BC
  { insns     :: (IOArray Word8 Word8, Word8)
  , lists     :: (IOArray Word8 [a], Word8)
  , bases     :: (IOArray Word8 a, Word8)
  }

-- addBase and addList similar to addInsn
addInsn :: Word8 -> Bytecode a -> IO (Bytecode a)
addInsn i b = let (is, l) = insns b in do
  writeArray is l i
  return (b { insns = (is, l+1) })

toBytecode
  :: ArrProg a -> Bytecode a -> IO (Bytecode a)
toBytecode (List xs) b = let (_, l) = lists b in do
  b1 <- addInsn pushList b
  b2 <- addInsn l b1 -- Bytecode argument
  addList xs b2
toBytecode Add b = addInsn add b
toBytecode Mul b = addInsn mul b
toBytecode (ZipWith f xs ys) b = do
  b1 <- toBytecode xs b
  b2 <- toBytecode ys b1
  b3 <- addInsn zipWith b2
  toBytecode f
toBytecode (Fold f b  xs) b0 = let (_, l) = bases b in do
  b1 <- toBytecode xs b0
  b2 <- addBase b b2
  b3 <- addInsn fold b1
  b4 <- addInsn l b3 -- Bytecode argument
  toBytecode f b4
```

Listing 5: A possible definition of a bytecode for the ArrProg language. This is missing a lot of safety, like checking array sizes before writing in the arrays, which have been omitted for brevity.

the resulting binary can *only* run on the platform that it was designed to run on. On the other hand, an interpreter can run the same code on any platform that it is designed to run on; this makes the original source code more portable, if the interpreter exists for many platforms.

However, programs run slower in an interpreter than when they are fully compiled [2]. The main reason that interpreters are slower than compiled programs, is that an interpreter is a program that runs another program; as such, it needs to keep track of two program counters, two stacks, and two sets of code, one for each of the interpreter and the interpreted program. This obviously requires more work than running a single program, which requires only one of each, as keeping track of this information takes processing power and memory.

Compilers are often hard to write and maintain, if not purely due to the size of the involved code. As an illustration, Listing 7 shows how to compile ArrProg into C code, which can then be compiled by a regular C compiler into machine code. Not all of the code is shown due to its size; the case for `Fold` is very similar to the case for `ZipWith`, except that the left-hand side of the assignment is a single element, rather than an array of elements. This code is much larger than the previous interpreters, and it also requires an external program, namely the C compiler, to complete the translation to runnable

```haskell
runBytecode :: Num a => Bytecode a -> IO [a]
runBytecode b = go [] 0
  where
    (a, end) = insns b
    op 4 = (+) -- 4 is bytecode for add
    op 5 = (*) -- 5 is bytecode for mul
    -- Parameters are stack, program counter
    go s n =
      let
        m = n + 1
        l = m + 1
      in
        if n >= end
          then return (head s) -- No crash assuming valid program
          else readArray a n >>= \i -> case (s, i) of
            (_, 0) -> do -- pushList
              li <- readArray a m
              xs <- readArray (fst (lists b)) li
              go (xs : s) l
            (xs:ys:s', 1) -> do -- zipWith
              opC <- readArray a m
              go (zipWith (op opC) ys xs : s') l
            (xs : s', 2) -> do -- fold
              bi <- readArray a m
              opC <- readArray a l
              base <- readArray (fst (bases b)) bi
              go ([foldl (op opC) base xs] : s') (l + 1)
```

Listing 6: A bytecode interpreter for the ArrProg language. The stack that this program uses is still a list representation, although an `IOArray` would be better suited for this in general. In this specific case, the values of the language are also Haskell lists, so the value representation is quite slow in general.

code. With this illustration, it should be clear that compilers are the most difficult to write out of these three programs. A benchmark of different interpreters and compilers follows later on in this thesis, to supplement the claims about the performance of each.

JIT compilers [2], also known as Just In Time compilers, are an important variation of compiler. This type of compiler aims to achieve the best of both worlds of interpretation and compilation by compiling a program at runtime. It allows a user to run their code similarly to an interpreter, but the resulting code runs much faster. However, generating machine code takes a relatively long time: see Section 4.1 for more details. As such, one of the challenges of JIT compilers is balancing the time spent generating code to the time spent running that code.

2.6 Bytecode Interpreter in Accelerate

In its current state, Accelerate uses a hybrid compilation scheme to execute programs: the most performance-critical code, the kernel code, is compiled using LLVM, whilst the rest of the language is executed using a tree-walking interpreter. However, as mentioned before, the downside of such an interpreter is that it is slow [2]. Therefore, there is a performance benefit to be gained in changing the execution scheme of these programs; we will discuss this in the rest of this section.

Although a bytecode interpreter is often faster than a tree-walking interpreter, writing an efficient bytecode interpreter is not a trivial task, and a lot of research has been done on the topic. For instance, opcode dispatch, which is the process in which the interpreter decides which code to execute based on the bytecode, can be done in different ways, where each has a different benefit depending on the situation. Listing 8 gives an example

```haskell
compileArrProg :: Show res => ArrProg res -> String
compileArrProg p = (\ (_, _, x) -> x) (printCode p names)
  where
    names = fmap (("list_"++) . show) [0..]

printCode (List xs) (name:names) =
  let
    l = length xs
    newList = printf
      "int* %s = (int*)malloc(sizeof(int) * %d);" name l
    listInitF (n, i) = printf "%s[%i] = %s;" name i n
    listInit = listInitF <$> zip xs [0..]
  in (names, name, intercalate "\n" $ newList : listInit)
printCode (ZipWith f xs ys) names =
  let
    (n1:names2) = names
    l = progListLen xs
    (names3, n2, xsLines) = printCode xs names2
    (names4, n3, ysLines) = printCode ys names3
    resList = printf
      "int* %s = (int*)malloc(sizeof(int) * %d);" n1 l
    zipProgram = intercalate "\n"
    allLines =
      [ xsLines
      , ysLines
      , resList
      , printf "for (int i = 0; i < %d; ++i) {" l
      , printf "%s[i] = %s[i] %s %s[i];" n1 n2 (opFor f) n3
      , "}"
      ]
  in (names2, n1, intercalate "\n" allLines)
```

Listing 7: A compiler that compiles a subset of ArrProg expressions into C code.

of a switch-based opcode dispatch, implemented in C. Some papers describing different techniques for opcode dispatch include: [3, 13]

However, as [15] mentions, these methods are far less important now than they used to be. This paper says that the main issue with the switch-based opcode dispatch lies in branch misprediction, but also that the branch predictions of hardware have improved over the last iterations of CPUs. Furthermore, their results indicate that the performance cost of these mispredictions are lower with newer hardware. As such, for a first version, it seems simplest to stick with a simple bytecode interpreter, and evaluate the performance bottlenecks afterwards.

2.7 Full Compilation in Accelerate

As section 2.6 states, Accelerate currently uses a tree-walking interpreter to execute certain elements of its programs. In that section, the proposed solution was to use a bytecode interpreter to increase the performance. However, bytecode interpreters are slower than native code [15]. There are multiple methods of fully compiling Accelerate programs into native code. Firstly, Accelerate has its own attempt at this using Template Haskell [17] through the `runQ` function. This function generates all the code that would normally be generated at run-time, the goal of which is to allow GHC to more agressively inline code and eliminate intermediary structures. This will not be discussed in more detail. Secondly, two more advanced frameworks for full compilation that have been considered are MLIR and LLVM. LLVM is currently already in use by Accelerate, but only for compiling kernel code. Both frameworks will be discussed in greater detail in this section.

```
uint8_t* insns = ...; // instructions
uint8_t* pc = insns;
bool running = true;
while (running) {
  switch (*pc++) {
    case OPCODE_ADD:
      // do
      break;
    case OPCODE_MUL:
      // the thing
      break;
    // ...
  }
}
```

Listing 8: An example of switch-based opcode dispatch. This particular implementation does minimal checking, especially on when to end the program and whether the end of the instruction stream has been reached.

MLIR MLIR[5] is a framework for compiling code into some other form; the framework itself is very general on what the resulting form should be. The resulting form is often a different dialect of MLIR. Defining a compilation from a source language to some target usually involves making one or multiple dialects for the source language to compile to, and some translation from this new dialect into an already-existing dialect that can be translated into the target form, either directly or indirectly. An example of this is RISE [11], which translates functional programs into an MLIR dialect, before using that to translate into a dialect which can be translated into native code.

Because of the dialect-based nature of MLIR, many optimizations that are common in compiler design can be reused. For instance, constant propagation is integrated in MLIR. As such, using MLIR can reduce the maintanance strain on a compiler project like Accelerate.

Although the benefits of MLIR sound very promising, the framework is relatively new. As a result, there are not many resources on how to use the framework, and the API is still undergoing changes; this is especially true for the Haskell API, at the time of writing. Accelerate is written in Haskell, and as such, it would be extremely time consuming to create bindings to the framework, learn how the framework works, *and* integrate it into Accelerate. It also does not have any benefit for this particular research project. As a result, using MLIR is out of scope for this project, as the benefits do not outweigh the costs at this time.

LLVM Accelerate already uses LLVM to compile kernel code, as mentioned previously. Therefore, integrating LLVM into the project is relatively simple. However, writing a full compiler from Accelerate to LLVM is still a more complicated task than writing a bytecode interpreter [15]. Full compilation does bring performance benefits, as native code is faster to execute than interpreting bytecode for the same program.

Back to full compilation in general, this is also interesing in Accelerate because it might enable cross-kernel optimizations, which are currently impossible. Currently, there is no way for the LLVM compiler to do any optimization between kernels, because the code between the kernels is completely separate from the code in the kernels. However, when using full compilation in the right way, all of the code that is executed between the kernels

[5] https://mlir.llvm.org/

is also compiled by the LLVM compiler. In this case, using Link-Time Optimization,[6] or possibly even using the correct compiler optimizations, the compilation might result in something that adds optimization across the boundaries of the kernels. As such, the performance improvement might be bigger than only the difference of time spent in non-kernel code.

Accelerate also executes many different calls to (kernel) code using the Haskell FFI. However, making calls using the FFI often results in data marshalling and other bookkeeping being executed by the Haskell runtime, which costs time.[7] If a program contains many small kernel invocations, this might have a noticable effect on performance. However, when fully compiling an Accelerate program, the entire program will be executed in a *single* FFI call. As such, this can result in an additional performance benefit for Accelerate programs.

Furthermore, Accelerate contains a code caching mechanism for compiled code. If the internal code does not change, Accelerate can re-use kernel code without recompiling it. Whilst this does not improve the performance of a program itself, full compilation can still benefit from this. Running a fully compiled program will require compiling it first, which costs time. However, if this code is loaded from the cache, this does not have to be recompiled. Therefore, performing full compilation *might* result in a reduction of the kernel code that is run overall, over the full lifetime of the program execution.

However, as we will see, Accelerate does spend a lot of time between kernel calls, but this is not caused by interpretation inherently. Although fully compiling Accelerate will, of course, result in a direct performance boost, it is suspected that this difference is not integral to Accelerate's performance struggles. As mentioned in the introduction, single-threaded Accelerate might see up to a 14% performance improvement on simple code, but the more fundamental issue is that Accelerate slows down when running on multiple threads. This issue is not caused by interpretation by itself, but we will discuss this in further detail in the next sections.

[6] https://gcc.gnu.org/wiki/LinkTimeOptimization
[7] https://wiki.haskell.org/Performance/FFI

3 Research Questions

The main goal of this research project is to improve the performance of DSLs embedded in Haskell, with a focus on programs written in Accelerate. To reiterate, these are the research questions that will be investigated:

1. How does performance differ between full compilation and interpretation in Accelerate's runtime system?
2. How does Accelerate's parallellism affect the runtime performance of the library as a whole?

The first research question encompasses both tree-walking interpreters and bytecode interpreters. As argued before, Accelerate's current interpreter, a tree-walking interpreter, is likely slower than an equivalent bytecode interpreter. Additionally, all interpreters are likely to be slower than fully compiling an equivalent program. However, remember that Accelerate's runtime system includes full compilation for kernel functions already; only the "glue" between the kernel calls contains interpretation. As such, it is currently unknown how much of the "hidden work" that Accelerate performs is done in the interpreter, and thus the possible performance gain of fully compiling Accelerate code is unknown as well.

Writing automated benchmarks using different Accelerate programs is the easiest way of measuring the difference in performance between the different methods of execution. Hopefully, running these benchmarks on all versions of the execution will determine one that is overall better than the others; the hypothesis is that the compiled versions will be faster than the bytecode interpreter, and that the bytecode interpreter will be faster than the tree-walking interpreter. However, the code generation itself will also take time; if the code generation takes too long, it will diminish the benefits, or it might even reduce the performance.

However, completely rewriting Accelerate is a large time investment. Additionally, as mentioned, it is unknown how much of a benefit implementing the interpreters and compilers will give; as such, instead of directly implementing this into Accelerate, we will study a prototype of different interpreters and compilers, and evaluate the current performance of Accelerate to see what it truly spends its time on.

4 Prototype

To determine the performance benefit that full compilation has over the different interpreters, and the difference between these interpreters, it is illustrative to have a prototype. In order to provide a realistic benchmark on the efficiency of a bytecode interpreter, it would be best to compare a prototype like this against a real language that uses a bytecode interpreter. Popular languages that fall in this category are Python [14] and Lua [18]. However, these languages are relatively complex, as they are so general and popular. It is more realistic to write a full tree-walking and bytecode interpreter in Haskell for a smaller language, which is not as feature-rich. As such, the prototype will use a language that is smaller, but also already has an efficient bytecode interpreter: Lox [12].

We will be comparing different execution implementations of this langauge against the bytecode interpreter that Nystrom's book implements in part 3, called *clox*, which is implemented in C. The implementations to compare against are:

- A tree-walking interpreter.
- Two bytecode interpreters, written in Haskell, that attempt to be as efficient as possible; another person on the Accelerate team has provided one of these implementations. The interpreters were developed completely separately from each other.
- A JIT compiler that compiles the program based on bytecode equivalent to that emitted by *clox*, an approach similar to Java [6].
- A JIT compiler that compiles the program from the AST that the tree-walking interpreter uses.

All interpreters, other than *clox*, are implemented in Haskell, and the JIT compilers use LLVM as an intermediate language. The choice of these languages is no coincidence: Accelerate is implemented in Haskell and it uses LLVM internally as well. Using the same technology as Accelerate will make the results of this prototype as comparable as possible to results of changing the Accelerate runtime system.

The Lox programs that have been measured are the ones in Listings 9a and 9b. One other program has been tested as well: this program consists of a single multiplication operator call (`10 * 10;`). The single multiplication is supposed to be a very small program, to illustrate the costs of generating code vs. not doing so. Each of these code fragments does not contain any variables in the global scope; instead, all variables have been pushed into a local scope by enclosing them in braces. This removes the need for a by-name lookup of variables, which Accelerate does not have. Therefore, this brings the benchmark closer to what it would be for Accelerate.

4.1 Prototype Results

The results of this experiment were slightly unexpected; see Figure 4. When looking at the chart, we can define two clear different groups when looking at the performance of these programs: the Haskell interpreters and the JIT compilers. When running the longer programs, the Haskell interpreters are clearly much slower than the JITs. On the short multiplication program, on the other hand, the roles are reversed: the interpreters are much faster than the JIT compilers. *Clox* is somewhere in between these two groups; it is clearly faster than all the Haskell interpreters on all programs, but slower than the

```
{
var x = 0;
var y = 0;
while (x < 1000000) {
  x = x + 1;
  y = x + y;
}

print y;
}
```

(a) The `while-long` benchmark. It is comprised of a single while loop. The total amount of iterations is equal to that of the `while-nested` benchmark.

```
{
var x = 0;
var z = 0;
var y = 0;
while (x < 1000) {
  x = x + 1;
  y = 0;
  while (y < 1000) {
    y = y + 1;
    z = z + y + x;
  }
}

print z;
}
```

(b) The `while-nested` benchmark. It is comprised of a while loop nested in another while loop.

Listing 9: Two of the three benchmark programs used in the prototype analysis.

JITs on the larger programs. The execution of the JIT code is shown separately, as an illustration for how fast a fully-compiled program would be.

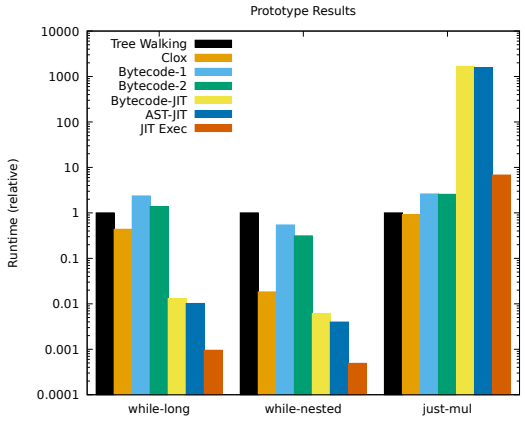
The bytecode interpreters perform worse than the tree-walking interpreter on the `while-long` benchmark, but they are faster on the `while-nested` benchmark. The bytecode-2 interpreter is always more efficient than the other bytecode interpreter, but more testing is required to see how it compares to the tree-walking interpreter overall.

The JIT execution is equally fast for both JIT engines; the code they generate is very similar, after LLVM's optimizations. The code generation speed of each is different, which is the only difference in their performance. The AST-JIT only generates an AST and from that generates the LLVM-code. On the other hand, the bytecode-JIT generates the AST, then the bytecode, and finally the LLVM-code. Generating the bytecode is probably the main reason the bytecode-JIT is slower. The smallest difference between the AST-JIT and *clox* is about 40x.

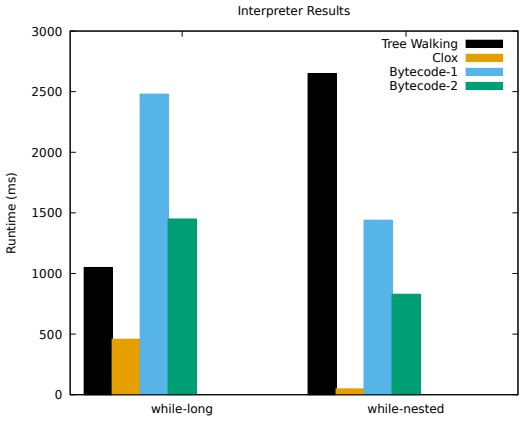
4.2 Prototype Discussion

These results are, of course, not *completely* generalizable to Accelerate. However, it can still be an indicator of how useful different methods are, and it shows some interesting results for the different interpreters. Earlier in the thesis, bytecode interpreters were said to be much more efficient than tree-walking interpreters. However, this is clearly not true for all implementations and for all executed programs.

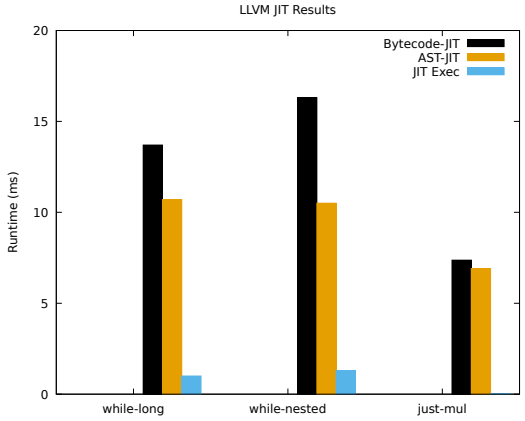
Based on these results, it looks like it is very difficult to write a bytecode interpreter in Haskell that is more efficient than an equivalent tree-walking interpreter. Not much care was taken to make the tree-walking interpreter efficient, but it is still faster on a subset of programs than bytecode interpreters that were written with efficiency in mind. This might be due to GHC being better at optimizing code that is more idiomatically Haskell, like the tree-walking interpreter, than it is at optimizing IO-heavy code (using IOArrays and IORefs). Moreover, even on the tested programs for which the Haskell bytecode interpreters were faster, they were clearly slower than *clox*, the reference implementation for a bytecode interpreter written in C. Based on these results, it looks like writing a bytecode interpreter for Accelerate in Haskell would bring hardly any benefit overall.



(a) A graph containing the relative runtimes of all benchmarks, and on all types of execution, illustrating the relation between all of these. A bar of height 1.0 means that it takes just as long as the tree-walking interpreter. A height of 10.0 means it takes 10x as long. Take note of the logarithmic scale.



(b) A graph containing only the runtimes of the longer benchmarks run by the interpreters. The `just-mul` benchmark is so fast, that none of the interpreters show up on this scale.



(c) A graph containing only the runtimes of the JIT executions on all benchmark programs.

Figure 4: Times it takes the different Lox implementations to run each benchmark program. The Bytecode interpreter types include genetation of bytecode, and the JIT implementation times include geneneration of machine code. None of the times include reading the source code or parsing the AST.

The simplicity of the example programs does complicate this matter somewhat, however. Looking at the difference in performance for both the `while-long` and `while-nested`, there is an argument to be made that keeping track of information in more complicated programs is harder for the tree-walking interpreter than it is for the faster bytecode interpreters. The code with the nested loop runs relatively faster using the bytecode interpreters compared to the tree-walking interpreter than the code that uses a single longer loop. The way the tree-walking interpreter handles scoping and variable lookups is a likely explanation for this; having more blocks with deeper code would likely result in less efficient code for the tree-walking interpreter, whilst this would not affect the bytecode interpreters as much as they keep track of this information in less computationally intensive ways. However, the Accelerate language's variable lookup already uses De Bruijn indices, which is a relatively efficient environment lookup compared to this tree-walking interpreter's scope-based lookup based on variable names. The lookup is implemented with a linked list, rather than a directly indexable data structure; it is currently unknown how much better the performance could be with a better data structure, but it would drop certain type-safety guarantees, which is why the Accelerate team currently opts against using one. The most important difference between these, however, is that Accelerate uses an $O(n)$ data structure, but a bytecode interpreter would use an $O(1)$ data structure *after a program transformation*; this transformation would still require using the current Accelerate datastructure. However, this program transformation does not *require* the usage of a bytecode interpreter, and this could also be done using full compilation or a slightly different tree-walking interpreter implementation.

The remaining options are a JIT compiler and a C-based bytecode interpreter. Earlier on in this thesis, a clear case was made that developing a bytecode interpreter is much simpler than writing a (JIT-)compiler. However, the use-case for Accelerate also complicates this matter. Accelerate already compiles kernel code, and it has to keep track of these functions so it can call them from within the user programs. Writing a bytecode interpreter in C and using the Foreign Function Interface, or FFI, would complicate calling these functions. These functions would have to be passed to C through the FFI, which would require adding naming conventions etc. as well, and passing these around as parameters. It would, however, remove the need to move back and forth between "native" code and "Haskell" code; a single call across the FFI is sufficient. Even so, having the code be split into multiple languages adds a maintenance burden to the code, as it requires maintainers to understand two different languages well enough to be able to write efficient code in them.

As such, a JIT compiler seems like the simplest option to improve the runtime performance of Accelerate. Accelerate already uses `llvm-hs` to JIT-compile kernel code, and it even has a caching mechanism in place so kernel code does not have to be compiled unless it changed. Changing the current tree-walking interpreter would then allow the JIT-compiler to make use of the same technology, which would mean that an unchanged program run on a different input would run without having to recompile anything, and it would run with higher performance, reducing the time between kernel calls. Additionally, this would reduce the complexity that the FFI currently adds in the same way that a C-based bytecode interpreter would, whilst maintaining a bytecode interpreter written in C would likely add more complexity than increasing the amount of code in the JIT compiler.

4.3 Prototype Conclusions

This prototype has given a clear indication to what the answer to one of the research questions would likely be. Although the answer is partial, it can advise on which implementation would be most useful. This partial answer is discussed in this section.

Using a different execution method will change the impact of the problem that Accelerate's runtime system struggles with; however, using a bytecode interpreter written in Haskell will likely not improve the time between kernel calls. The performance difference for full compilation and bytecode interpretation is hard to generalize out of these results, but it is safe to say that a bytecode interpreter would have to be written in another language to have a clear impact on the overall runtime of an Accelerate program. Writing a (JIT-)compiler is likely better keeping in mind the previous comments on predicted complexity; the JIT compiler would use technology already existing in the project, whereas writing a bytecode interpreter in a different language will add code complexity in the form of a different language, and an interface between these languages. The C-based bytecode interpreter would still have the benefit of only requiring a single call across the FFI boundary; however, the JIT compiler has the exact same benefit. These would both reduce some of the complexity that the current code has in the same way.

Based on this, the most promising option looks to be a JIT compiler. The compiler would have the best performance, it would eliminate a specific source of complexity by condensing all FFI calls into a single one, and it would not introduce new complexity by adding a new language to the project.

Referring back to the research questions in the last section (Section 3), there is now a partial answer to one of these questions. The runtime performance between full compilation and interpretation in Haskell is striking; interpretation is, as expected, very slow compared to compilation. However, even in light of this evidence suggesting that fully compiling the program will reduce time between kernel calls (and thus improve overall performance), the actual impact is unknown as of yet; although it is clear that Accelerate's runtime system does more work than strictly required between kernel calls, it is unknown what this work is. As such, the next section discusses what Accelerate is currently doing, and this will help in finding out what the possible benefit of fully compiling Accelerate would be.

5 Evaluation of Current Accelerate

This section goes into a detailed evaluation of Accelerate as it is right now. First, the usefulness of changing the execution of Accelerate from a tree-walking interpreter to a JIT compiler is evaluated. After this, the problems that multi-threaded Accelerate struggles with are evaluated.

5.1 Profiling Accelerate

In order to know what exactly happens during a program execution, Accelerate programs send data to a *Tracy* server. For more information on what the output of this program looks like, refer to Figure 5. This server shows different statistics based on the program. Currently, the application tells *Tracy* when it is running a kernel, when it is running any (other) task or waiting for a task to be queued. When a thread is waiting for a task to be queued, or for the queue to be available for this thread, the output in *Tracy* gives a red bar.

"Running a task" is intentionally vague in this context. It refers to processing any task in the concurrent queue that Accelerate uses to schedule its jobs. These tasks can either be (re-)scheduling tasks that are queued by splitting them up into smaller chunks so that they can be executed in parallel, or running actual kernels. (Re-)scheduling a task also involves marshalling the arguments for the kernel calls, as well as allocating the target buffers. Running a kernel then exclusively consists of running the foreign kernel function through the FFI; the time between starting the FFI call and the start of the kernel function is also measured, to determine how long calling a foreign function takes.

Running Accelerate programs is less efficient than slightly more optimized approaches. Two ways in which the performance can likely be improved are, as mentioned before, fully compiling the program and improving how Accelerate utilizes the threads. This is explored in more detail in the following sections.

All of the following experiments have been performed on an Intel Core i5-9300H CPU. The experiments do not include the time spent by Accelerate in compiling the program (or loading from cache), and for neither sets of programs does it include the allocation or initialization of the initial data buffers. Instead, the time taken is measured from the moment the program schedules its first job into its job queue.

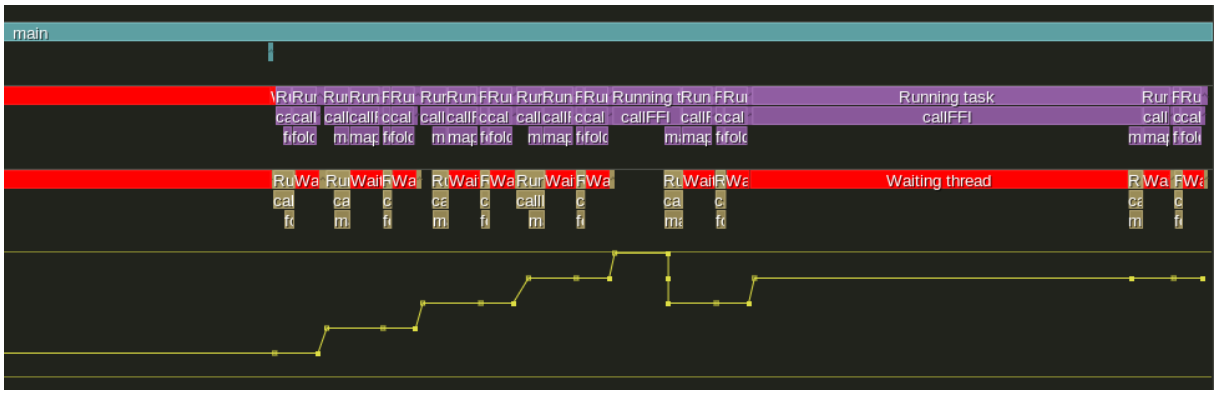


Figure 5: An example of what profiling output in *Tracy* looks like.

```
arrInc
  :: Acc (Array DIM1 Float)
  -> Acc (Array DIM1 Float)
arrInc = awhile (any (<5)) (map (+1))
```

(a) A function using an awhile loop in Accelerate.

```
dotp
  :: Acc (Array DIM1 Float)
  -> Acc (Array DIM1 Float)
  -> Acc (Scalar Float)
dotp xs ys = fold (+) 0 (zipWith (*) xs ys)
```

(b) A function computing the dot product of two vectors in Accelerate.

Listing 10: Two functions defined in Accelerate to illustrate the performance that Accelerate can still gain.

	arrInc			dotp		
Threads	**Acc**	**C**	**Speedup**	**Acc**	**C**	**Speedup**
1	8.1ms	7.69ms	5.1%	0.199ms	0.122ms	38.69%
7	27.96ms	3.41ms	87.8%	0.263ms	0.111ms	57.79%

Table 1: The results of running the benchmark programs, as well as their C-equivalents. The speedup is the amount of time saved by running the C-version compared to the Accelerate version. All programs are run on buffers of half a million 32-bit floating point numbers.

5.2 JIT Compilation

Accelerate currently does more work than strictly necessary when evaluating a program. To prove this, one can run a program through Accelerate while profiling this execution. This execution writes the object code for the kernels to disk for caching purposes; this makes it possible to link to the generated kernel code from a different program. Writing the same program in C, linking to the kernel code, should yield a similar result to using a JIT compiler for all of Accelerate's execution. Profiling this new program and comparing this to the Accelerate profile should show what the possible performance improvement is.

The Accelerate code in Listing 10 has been tested in exactly this way. The C programs run the work on worker threads, much like Accelerate does. For this evaluation, the programs have been compiled using similar approaches to the equivalent Accelerate programs; for instance, each iteration of the `awhile` loop allocates a new buffer and frees the old one. Note that there is no garbage collection in the C versions. Instead, all allocated memory is carefully freed when no longer needed.

Looking at the results for single-threaded performance in Table 1, it is clear that Accelerate still has some performance wins to make. Accelerate takes around 5-39% longer to run programs than a compiled C-program. This can have two causes: either Accelerate is slower because the code it runs is JIT-compiled code, or because of the interpreter code that runs between kernel code. JIT-compiled code can be slower than regularly compiled code, because optimizing code takes time, and a JIT compiler has to balance compilation time with performance [2]. However, as the kernel code used by the C-program is the same code that Accelerate uses, it is clear that this code should run with the same performance. As such, the interpreter code is likely the cause of the missing performance, and fixing this is as "simple" as making a fully compiled program.

Clearly, Accelerate still has some performance to win, even on a single-threaded program. However, the biggest issue Accelerate has does not lie in single-threaded code, but specifically in multi-threaded performance. Therefore, we will investigate the performance on multiple threads next.

5.3 Multithreading

Accelerate can also evaluate programs in a multi-threaded fashion; as a matter of fact, this is the default option. However, running the program using multiple threads can be very slow, especially as the amount of threads increases. We investigate this in the rest of this section.

The following experiments have been conducted using the same settings for each program as described in the previous section, apart from the amount of threads. On default settings, Accelerate uses 8 threads on the CPU on which these programs have been tested; Accelerate automatically generates one worker thread for each thread supported in hardware. However, the compiled C-versions of the benchmarks have some issues running on 8 threads; this probably happens because the main thread and 8 worker threads together are not supported in hardware, which causes the threads to deschedule more often. Running on 7 threads, on the other hand, works very well. As such, all of these benchmarks compare running on 7 threads instead.

Looking at the results in Table 1 again, we can conclude a few things.

- Accelerate's multithreading is slower than running a C program with the same type of queue.
- Accelerate's performance diminishes when running on multiple threads.

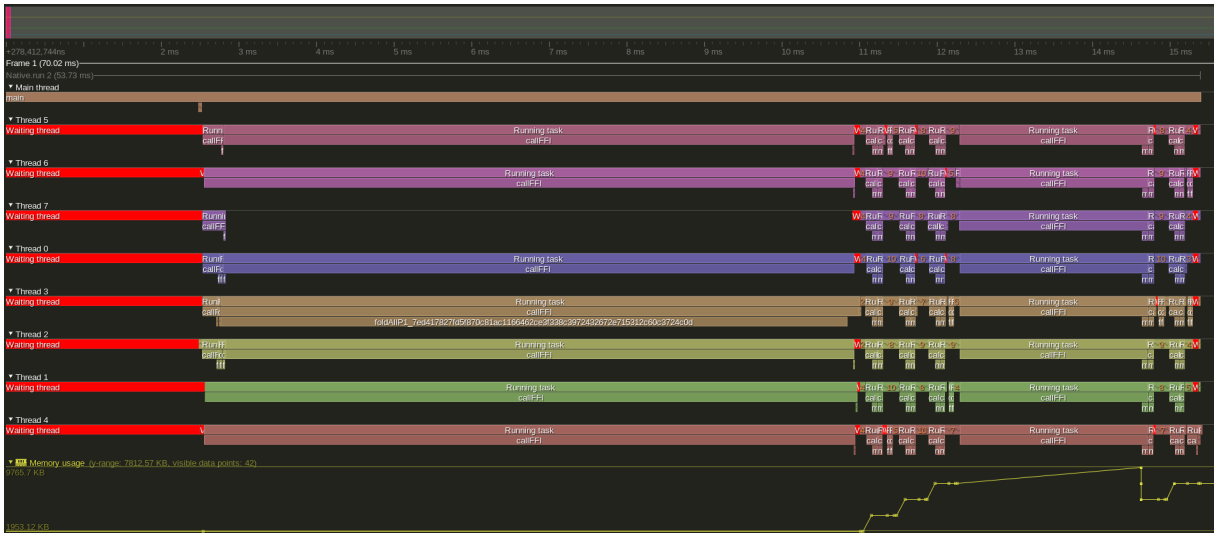
For instance, the `arrInc` Accelerate program runs about 3.5x slower on 7 threads than on 1, while the C version speeds up, being slightly more than 2x faster. This makes it clear that the way Accelerate works when running a program on multiple threads is less than optimal, and also that the type of queue itself is not the issue, since the C version uses the same type of queue.

5.4 Possible Causes

Given the information in the previous sections, it should be clear that running a program through Accelerate introduces some overhead, even on a single thread. However, it is unclear what causes this overhead exactly; the result for running the `arrInc` program in a single-threaded environment seems to show that the tree-walking interpreter does not add much overhead, at a mere 5.1% of overall runtime. For the `dotp` program, the runtime difference seems much larger, but most of this time is also not spent in the tree-walking interpreter. Therefore, the difference between the current tree-walking interpreter and compilation hybrid cannot be improved upon too much by applying full compilation.

The problem with the multithreading is more problematic. These results seem to indicate that there is an issue with the multithreading. However, looking at the results in *Tracy*, this does not look like the only problem. Looking at Figure 6a, it seems extremely unlikely that a call to `callFFI` *occasionally* takes about 10ms to reach the foreign code, whereas it often takes *less than 10µs*. This suggests that there is something else going on that degrades the performance, as the time this takes should not be quite this variable. Additionally, the threads wait longer than in the C version; this suggests that there is something else impacting performance than the concurrent queue.

The culprit appears to be the Garbage Collector, or GC. This can be seen by using the `ghc-gc-hook` package and running the `dotp` code without any command-line arguments



(a) Run with default settings.

(b) Run with overrided heap-size settings to suppress the garbage collector. The changed setting is `+RTS -A8192M`.

Figure 6: Two different runs of the Accelerate version of the `dotp` program in Listing 10b. The code is the exact same, but different command-line arguments have been passed to the Haskell runtime system. Note the difference in the timescale in both images.

and running with `+RTS -A8192M`; the former's *Tracy* log is shown in Figure 6a, and the latter's is shown in Figure 6b. The output of the GC hook is empty when passing the RTS flags; this shows that the garbage collector did not interfere with the program's run. On the other hand, without these flags, the GC hook reports that the garbage collector did run.

The difference in these program runtimes is striking. The runtime without garbage collection interference is about 2.9ms, whereas with the garbage collector this runtime grows to almost 13ms, using the timing described earlier in this section. The impact on the runtime of the complete program is relatively less pronounced, but still obvious, an almost 7x difference (70ms versus 487ms). The aforementioned problem with the `callFFI` performance is also reduced significantly; the longest-taking ones are around 8ms with the GC interference, but less than 900μs when the GC is never interfering. There is still some clear variance in the time one of these calls takes; this is likely due to thread scheduling. This shows that one of the causes of Accelerate's performance struggles is Haskell's GC.

Measuring the real impact on program performance from GC is a known hard problem [4]. Many assumptions in GC optimization have led to misguided approaches. Actually measuring how long the GC takes on a program compared to manual memory management is difficult, but it is safe to assume that this takes between 6% and 92% more CPU cycles [4]. However, how much time exactly this costs on Accelerate programs is unknown.

It is also unknown why this disproportionally affects multi-threaded execution of Accelerate programs. It is possible that the GC synchronization is the root cause of this. The GC cannot run while threads are using their data, and therefore it needs to cause all threads to wait [9]. When there is only one (worker) thread, the instant the thread starts waiting, the GC can run. However, when there are multiple worker threads, most of them have to wait longer than this, as they all have to wait until the last one is waiting as well. This increases the overall waiting time done for each thread, which increases the waiting time for the overall program in $\Omega(n)$ in the amount of threads, whereas the performance gain of adding more threads is $O(n)$, meaning that the performance of using more threads could deteriorate performance.

It is currently unknown whether this affects all Haskell programs or whether the specific use-case of Accelerate is a worst-case for this behavior to occur. What is known, however, is that there is still a performance improvement to be gained even if the GC does not interfere with the program's runtime: the compiled C-version of the `dotp` code runs about 10x faster than the Accelerate version. However, it is important to note that Accelerate has more *constant* overhead than the compiled C-version does; see Figure 7.

However, there is also evidence that increasing the amount of calls to the GC does not have a strong impact on the performance of certain Accelerate programs, even though those programs do have seemingly similar problems. For instance, Accelerate's LULESH implementation seems to have performance drops when the garbage collector runs as well, but changing the heap size in the same way does not change this situation. The images in Figure 8 show results from multiple executions of this program. It is clear that increasing the heap size reduces the amount of times the garbage collector runs. However, decreasing the amount of GC calls does not improve the overall performance. Clearly, something other than the sheer amount of GC calls is causing performance issues.

The garbage collector causes some interesting behaviours in the context of an Accelerate

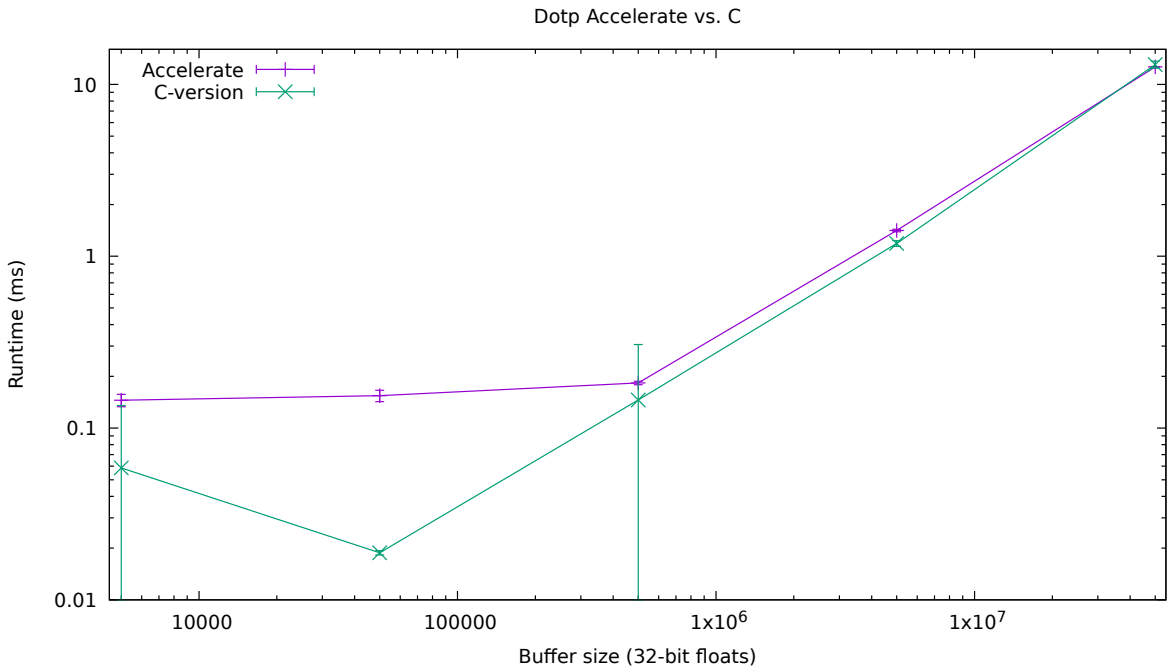


Figure 7: Running the Accelerate version of `dotp` compared to running the C version. The C version is slightly faster overall, but both run in $O(n)$ of the size of the buffer.

program, which also affect its performance. This paragraph discusses two recurring examples that stand out, but there are possibly other patterns that have yet to be found. The first example is shown in Figures 9a and 9b. What these images show is what happens *before* a garbage collection. The worker threads get paused when the garbage is being collected. However, the images show two very different cases: one shows a case in which the GC causes a relatively long stagnation, and the other a very minimal one. It is unknown what causes these differences in stagnation time. The other pattern is shown in Figure 9c. This image shows what happens *after* a GC: all but one of the worker threads resumes working. The last thread seems to be actually performing all the "memory freeing", as can be seen with the yellow line at the bottom, which shows the graph for memory usage; it is clear that a lot of data is freed during this time. This behaviour causes issues with how Accelerate handles parallelization of work. When one kernel depends on the result of another kernel, Accelerate queues the dependency first, which needs to be fully executed before the dependent kernel can run. Of course, this is the order in which this work *must* happen. However, the thread that performs the freeing often does this *after* taking a work item out of the queue. If the freeing takes a long time, this causes the other threads to wait for this work to be done as well; thankfully, freeing this memory is often very fast, but with large heap sizes, this can take a long time. This (partially) explains why the program takes longer to run once the heap size becomes too large.

This section has shown that there is both evidence supporting that multiple GC calls increases the run time of an Accelerate program, but also that the amount of GC calls is not a sole indicator of how long such a program takes. Therefore, more research is needed to find out how much of an influence the GC has on the performance of Accelerate programs. It is clear that something in the Haskell runtime system interferes with running Accelerate programs, and that at least part of these issues are caused by the garbage collector, but with the current data it is hard to say how much of the performance drop

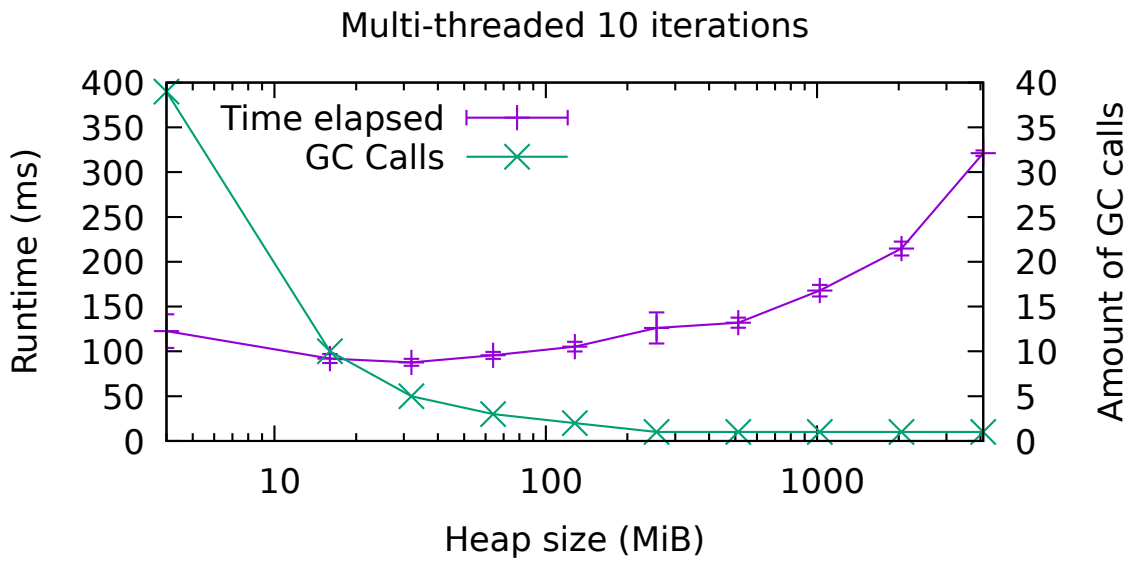
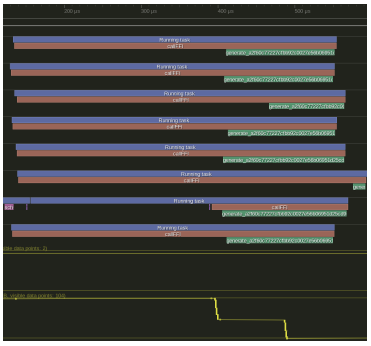


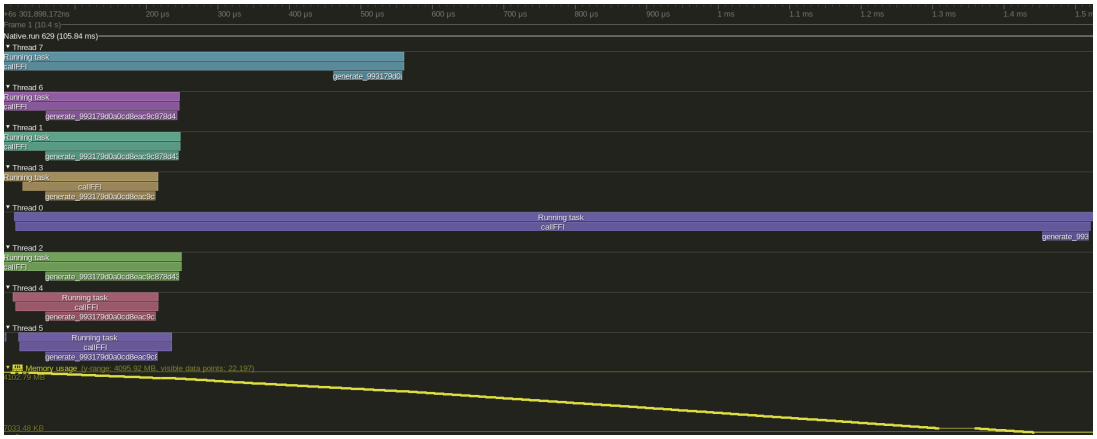
Figure 8: Different runs of LULESH, at 10 iterations, with the total time taken on the left y-axis and the amount of times the Haskell garbage collector was called on the right side. The amount of memory that needs to be allocated before a GC happens is shown on the X-axis.

is truly caused by it. The next section goes into future work that can show that this is the problem, as well as possible ways to fix this.

(a) A GC call that caused an extreme 37ms stagnation of the worker threads. This entire iteration took 44.7ms. For reference, most iterations take around 7-13ms, and the GC in the image to the right takes around 1/200th of the time.



(b) A GC call that caused almost no stagnation of the worker threads; the waiting time is around 200μs.



(c) The "freeing behaviour" of the Haskell GC. This is an extreme example that happens with very large heap sizes; this particular case happened with `+RTS -A4096M`. The freeing time is around 1.4ms.

Figure 9: Different interesting "places of interest" in an execution of LULESH through Accelerate. All executions use `+RTS -A16M` unless otherwise specified.

6 Future Work

The previous two sections have investigated what causes Accelerate to be slower than hand-compiled code in certain cases. We have seen that fully compiling Accelerate programs can result in a speedup, but that this might not eliminate the main issues with performance entirely. Although it is obvious that there are still improvements to be made, it is unclear what the best course of action is. This section gives an in-depth analysis of different possible ways to continue the investigation, or possible ways to reduce the problems Accelerate has.

The first option involves simply gathering more data on the problem and performing a more rigorous analysis on this data; Section 6.1 discusses exactly what this entails. Section 6.2 discusses a way of reducing the overhead the Haskell GC brings, by putting certain data into compact regions. Although compact regions might reduce the overall overhead that the GC brings, it might be more beneficial to eliminate the problem in its entirety; Section 6.3 lists some possibilities of entirely bypassing Haskell's GC, which should increase the performance with even more certainty. Finally, Section 6.4 discusses ways of finding out whether Accelerate's multithreading issues affect more Haskell programs than only Accelerate programs.

6.1 More In-Depth Investigation

Accelerate clearly has an issue with running programs in parallel on multiple threads. Additionally, it seems that Haskell's Garbage Collector does affect the performance of Accelerate programs. However, it is unclear to what degree this degrades the performance of Accelerate programs. This section describes further experiments that can determine the degree to which the GC interferes with performance more unambiguously.

One experiment involves correlating kernel executions that take relatively long with GC calls that happen during them. Using Tracy, it is possible to save an execution trace of an Accelerate program. The saved file can also be exported to a CSV file; one option for this type of file is to give the start of each occurrence of each trace, and the length of execution. If all "longer" executions of kernels fall during GC calls, that is definitive proof that the GC interferes with the kernel execution.

However, there are some intricacies to keep in mind. For instance, the definition of when a kernel takes "longer than normal", as well as how much of this time is actually caused by the GC. The details of this experiment are better left for the person performing this to decide.

Other experiments might want to include more information, such as different Accelerate programs. These experiments might also want to identify other possible interference, such as including context switch information, or other Haskell runtime information. This will show more clearly *how much* of the interference is caused by the GC, and how much it affects different programs, rather than only *that* the GC affects these programs.

6.2 Compact Regions

Haskell's GC clearly interferes with the execution of Accelerate programs. The effect of this can likely be reduced by reducing the time spent in each GC call. One way to

reduce the amount of garbage that the collector needs to sift through is by using compact regions.[8] We discuss what these do and how this can be useful in the rest of this section.

When performing a GC, the Haskell runtime system copies all data that can be referenced over from one data buffer to another. The details of how this work are not important, and can be found in [9], but the main idea is that all data that cannot be referenced is not copied to the new buffer, and as such, can be deleted when the collection is complete. There is also a mark-and-sweep alternative, which does not move the data; the details for this implementation can also be found in [9]. However, going through all data and references can take a long time if there are many of these references. Additionally, this work needs to be done multiple times for data that exists during all of the program's execution. If most of the data exists for all of the program's execution, the GC takes $O(n^2)$ time for longer program executions; linear in the amount of data in scope,[9] and linear in the amount of GC calls.

For Accelerate, some things need to stay in scope continuously, which can explain this problem. For instance, the AST of the program that is being run needs to remain in scope, as it is used in interpretation. This would be a good candidate for something to place inside a compact region, as it contains multiple references, and it also exists for most of the program's runtime. It is unknown how much this would help with the performance of Accelerate, or EDSLs in general for that matter, however. It would make for an interesting experiment to do this in Accelerate and/or other EDSLs to see how much this improves performance, or if it is even detrimental.

6.3 Compiled Runtime System

As mentioned in the last section, Haskell's GC interferes with Accelerate programs. Whereas the previous section suggested ways of fixing this by reducing the work the GC performs, this section explores ways of completely bypassing the GC, as well as ways of completely bypassing the Haskell runtime system.

One way to bypass the Haskell GC is by using manual memory management from within Haskell. Accelerate currently allocates managed memory through the Haskell runtime, which means that it will automatically free the buffers when they are no longer in use. However, the cost of this is that the GC will have more work to do in the collection and freeing phases. To mitigate this, large buffers can be allocated using `mallocBytes`.[10] These buffers also need to be freed manually, which is the downside of not having garbage collection. This will reduce the strain on the GC by reducing the amount of bytes allocated in it. However, when running most code in Haskell, thunks are also allocated for many statements; to mitigate this, it is not sufficient to allocate the buffers in this manner.

Fully bypassing the GC would involve replacing the current runtime written in Haskell with one written in another language. However, as the GC seems to be causing disturbances in the performance, the new language needs to not have a garbage collector either. Two good candidates in this case are C and LLVM. C can interface with Haskell through Haskell's Foreign Function Interface. LLVM is already generated for kernel code; rewrit-

[8] https://hackage.haskell.org/package/compact-0.2.0.0/docs/Data-Compact.html

[9] Even one of the main authors of the GHC runtime system, Simon Marlow, has mentioned this: https://stackoverflow.com/a/36779227

[10] https://hackage.haskell.org/package/base-4.16.1.0/docs/Foreign-Marshal-Alloc.html

ing more code in LLVM might be possible as well using the JIT compiler. However, it is likely easiest to replace single functions or modules at a time using C, with the FFI as mentioned. Being able to modularly rewrite the library is beneficial, because this means that only the largest cost centers can be rewritten and the rest of the code can stay in place, preventing the need for a complete rewrite.

Rewriting (part of) Accelerate in C additionally means that not only the GC, but also the rest of the Haskell runtime cannot interfere with its execution. C does not have a fancy runtime system that performs background bookkeeping and hidden computations, unlike Haskell. This makes the language more suited for a high-performance computing runtime compared to Haskell.

Although it is unknown how much faster such a (full) rewrite of the Accelerate runtime system will be, an upper bound has been found in Section 5. Having such an upper bound on the possible performance that can be gained is useful when deciding whether it is worth rewriting the library. It is also possible that such a new runtime system is slower. This can indicate two things: either the new runtime is relatively bad, or the Haskell runtime was not the issue. It is always important in high-performance computing like this to keep profiling to see where the issues lie.

Hopefully, bypassing the Haskell runtime system completely will accelerate Accelerate. As mentioned, however, it is interesting to know if Accelerate hits a bad case for the Haskell runtime, or if it is possible to recreate this behaviour with more standard Haskell programs. As such, the next section discusses a possible experiment to determine this.

6.4 Multithreading Slowdown

Accelerate programs slow down when run on multiple threads. Haskell's runtime system seems to be the problem, and not the embedded Accelerate language. However, it is unknown what causes Accelerate programs to interfere with the Haskell runtime, and it is therefore unknown whether this only affects Accelerate or a larger class of Haskell programs. This section lists certain constructs that Accelerate's internals use that can be investigated to see whether Haskell's runtime loses performance when using them in a multi-threaded environment in isolation, or if there is a certain interplay that causes these issues.

Accelerate uses, among other things, pinned memory allocations (using `newPinnedByteArray#`). This is one of the few things that would logically influence the garbage collector; allocating memory like this tells the GC that it is not allowed to move the value around. Of course, it does not seem like this should affect the GC very much at face value; however, it is interesting to see if this is what causes the issue.

Additionally, Accelerate uses foreign function calls to dynamically linked functions that are compiled at runtime. It uses LLVM to compile the functions at runtime, and it uses the `libffi` Haskell library to call them. Since the runtime interference often happens between calling `callFFI` from this library and actually calling the native compiled function, this is another point of interest.

One such experiment to see whether these constructs cause the runtime to *reduce* performance can consist of writing a parallelizable algorithm, such as a map over an array, or mergesort/quicksort of an array, and running this on multiple threads, similarly to how

Accelerate does this, with a concurrent task queue. If the program slows down, but it is not spending a lot of time waiting for the queue, as appears to be the case for Accelerate, this means that something else is interfering with the runtime system; if this is not the case, change the program so that it uses any of these constructs, and retry, and see if one of these constructs in isolation or a mix of these causes some issues.

7 Conclusion

As shown in this thesis, Accelerate suffers from performance issues. However, these issues likely lie in the Haskell runtime system. It is hard to say what exactly causes this, but the garbage collector likely causes at least some of the performance concerns, especially in a multi-threaded environment. It is important to keep in mind that Accelerate is not a standalone language, but that it is embedded in Haskell; as such, its performance is dependent not just on its own runtime system, but also on the runtime system of its host language. Other EDSLs are likely to suffer from this too; however, it is not important for all languages to be as fast as possible, so other languages might not have to consider the performance of the host language as strongly. While prototyping a language in Haskell is fast, and it is relatively simple to get a working version of the language, it is important to remember that interpreters written in Haskell are hard to make fast. To make sure that the language has good performance, it is better to perform a full compilation of the program, or at least the most performance sensitive parts. Examples of languages that do this at runtime already exist, such as Java [6].

Accelerate would see at least a marginal speed increase if the programs were fully compiled before execution, as is expected; this does not include the time spent compiling. However, the data found in this thesis suggests that the multi-threaded runtime would benefit more than a single-threaded execution, if Accelerate implements a complete runtime system, rather than relying on its host langugage's. Getting the most performance possible out of a program often requires a high amount of control, and writing a fully embedded language, both in source and in execution, simply is not cut out for this, as developers have to rely on the host language's runtime system to be perfectly suited to their needs.

References

[1] Niloofar Aghaieabiane, Henk Koppelaar, and Peyman Nasehpour. "An improved algorithm to reconstruct a binary tree from its inorder and postorder traversals". In: *Journal of Algorithms and Computation* 49.1 (2017), pp. 93–113.

[2] John Aycock. "A brief history of just-in-time". In: *ACM Computing Surveys* 35 (2003), pp. 97–113.

[3] Marc Berndl et al. "Context threading: A flexible and efficient dispatch technique for virtual machine interpreters". In: *International Symposium on Code Generation and Optimization*. IEEE. 2005, pp. 15–26.

[4] Zixian Cai et al. "Distilling the Real Cost of Production Garbage Collectors". In: *CoRR* (2021).

[5] Manuel MT Chakravarty et al. "Accelerating Haskell array codes with multicore GPUs". In: *Proceedings of the sixth workshop on Declarative aspects of multicore programming*. 2011, pp. 3–14.

[6] IBM Corporation. *The JIT compiler - IBM Documentation*. 2005-2022. URL: https://www.ibm.com/docs/en/sdk-java-technology/8?topic=reference-jit-compiler.

[7] Jean-Rémy Falleri et al. "Fine-grained and accurate source code differencing". In: *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*. 2014, pp. 313–324.

[8] Steve Freeman and Nat Pryce. "Evolving an embedded domain-specific language in Java". In: *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*. 2006, pp. 855–865.

[9] Ben Gamari. "A Concurrent Garbage Collector for the Glasgow Haskell Compiler". In: ().

[10] HaskellWiki. *Embedded domain specific language — HaskellWiki,* [Online; accessed 29-June-2022]. 2021. URL: %5Curl%7Bhttps://wiki.haskell.org/index.php?title=Embedded_domain_specific_language&oldid=64429%7D.

[11] Martin Lücke, Michel Steuwer, and Aaron Smith. "Integrating a functional pattern-based IR into MLIR". In: *Proceedings of the 30th ACM SIGPLAN International Conference on Compiler Construction*. 2021, pp. 12–22.

[12] Robert Nystrom. *Crafting Interpreters*. Genever Benning, 2021.

[13] Ian Piumarta and Fabio Riccardi. "Optimizing direct threaded code by selective inlining". In: *Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*. 1998, pp. 291–300.

[14] Python Software Foundation. *Welcome to python.org*. 2022. URL: https://www.python.org/.

[15] Erven Rohou, Bharath Narasimha Swamy, and André Seznec. "Branch prediction and the performance of interpretersDon't trust folklore". In: *2015 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE. 2015, pp. 103–114.

[16] Sean Seefried, Manuel Chakravarty, and Gabriele Keller. “Optimising embedded dsls using template haskell”. In: *International Conference on Generative Programming and Component Engineering*. Springer. 2004, pp. 186–205.

[17] Tim Sheard and Simon Peyton Jones. “Template meta-programming for Haskell”. In: *Proceedings of the 2002 Haskell Workshop, Pittsburgh*. Oct. 2002, pp. 1–16. URL: https://www.microsoft.com/en-us/research/publication/template-meta-programming-for-haskell/.

[18] The Lua Team. *Lua: about*. 2022. URL: https://www.lua.org/about.html.

[19] Michael Vollmer et al. “Compiling Tree Transforms to Operate on Packed Representations”. In: *31st European Conference on Object-Oriented Programming (ECOOP 2017)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik. 2017.

[20] Paul R Wilson. “Uniprocessor garbage collection techniques”. In: *International Workshop on Memory Management*. Springer. 1992, pp. 1–42.

[21] Yichen Xie and Alex Aiken. “Context-and path-sensitive memory leak detection”. In: *Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*. 2005, pp. 115–125.