SUPERWORD LEVEL PARALLELISM IN THE GLASGOW HASKELL COMPILER

By

Abhiroop Sarkar

A Dissertation Submitted to the Graduate

Faculty of University of Nottingham

in Partial Fulfillment of the

Requirements for the Degree of

MASTER OF SCIENCE

Major Subject: COMPUTER SCIENCE

Examining Committee:
Prof. Graham Hutton
Dr. Henrik Nilsson
Dept. of Computer Science

____
Prof. Graham Hutton, Dissertation Adviser

University of Nottingham
Wollaton Road, Nottingham

September 2018
(For Graduation December 2018)

CONTENTS

LIST OF FIGURES

ACKNOWLEDGEMENT

I would first like to thank my thesis adviser Professor Graham Hutton, the head of the Functional Programming laboratory at University of Nottingham. He consistently allowed this thesis to be my own work, but steered me in the right direction whenever he thought I needed it.

I would also like to thank Carter Schonwald, who unofficially mentored me through a lot of pitfalls of vectorisation and introduced me to the amazing world of numerical computing. Special thanks also goes out to Andreas Klebinger from the ghc IRC channel for always helping me out to understand the compiler better and better everyday.

Finally, I must express my very profound gratitude to my parents and friends for providing me with unfailing support and continuous encouragement throughout my years of study and through the process of researching and writing this thesis. This accomplishment would not have been possible without them. Thank you.

ABSTRACT

Superword parallelism or vectorisation has been a popular high performance computing technique in supercomputers for nearly 50 years now. With the rise of various memory intensive numerical computing as well as deep learning algorithms like neural networks, vectorisation is finding itself increasingly being used in commodity hardware as well. For a long time imperative languages have ruled the rooster for high performance computing, but over the last decade the purely functional language Haskell has steadily caught up with the speed and performance of languages like C in various benchmarks. However vectorisation remains an unsolved problem for various Haskell compilers. This thesis attempts to solve that issue by adding support for vectorisation to the Glasgow Haskell Compiler. It also presents a new library for vector programming called *lift-vector* which provides a declarative API for vector programming. We show improvements in performance of certain numerical computing algorithms using the library and propose a way forward to automatically vectorise programs inside the Glasgow Haskell Compiler.

1. INTRODUCTION

The ever-increasing complexity and performance requirements of modern software over the last couple of decades, has prompted hardware designers to introduce a wide range of parallel architectures. Unfortunately, the age old classical design of imperative/procedural programming languages and their runtimes are not always amenable to extract the maximum performance from the underlying parallel hardware.

In his highly cited paper [Fly66], MJ Flynn introduced a taxonomy of parallel machines, classifying them as Single Instruction Single Data (SISD), *Single Instruction Multiple Data (SIMD)* or Multiple Instruction Multiple Data (MIMD) machines. In this thesis we are concerned with extracting the maximum effectiveness of Single Instruction Multiple Data support, provided by various major hardware vendors like Intel, AMD, ARM Neon etc. We look at the origins of vectorisation in more depth in the following section.

Historically the support for SIMD instructions to the x86 instruction set was first added in the year 1999. Consequently, the Intel Pentium III line of processors started supporting vector instructions. Competing with Intel, ARM released the NEON vector instruction as part of its Cortex A5 line of processors in 2003.

SIMD parallelism, also known as super word level parallelism or *vectorisation*, effectively exposes 128 bits, 256 bits or 512 bits registers and allows data parallel operations using these super-word sized registers. SIMD technology uses a single instruction to perform the same operation in parallel on multiple data elements of the same type and size. This way, the hardware that generally operates on two 32-bit floats can perform four or eight or sixteen parallel operations of 32-bit floats in exactly the same amount of time.

The prominent compilers whose code generators support vectorisation (i.e they emit vector instructions) are the GNU Compiler Collection(GCC) and LLVM compiler tool chain. Certain APL implementations (like Dyalog APL) also provides support for vector instructions. Among the newer programming languages Go-lang has very experimental support for SIMD operations. The Rust language, by the virtue of being hosted using LLVM, also supports basic SIMD operations.

However if we notice, the focus of vectorisation support primarily exists for compilers of strictly imperative and high performance system languages. On the other side of the programming language landscape, functional languages have traditionally been at the forefront of correctness and verification related research, but have somehow lagged behind in the field of high performance computing compared to their imperative counterparts. With the inception of the Glasgow Haskell Compiler [Jon+93] in 1990, Jones et al showed that it is possible to efficiently compile a pure lazy functional language to stock hardware [Jon92] while keeping a transparent performance using the concepts of graph reduction machines [Cla+80].

Parallel programming in general introduces a number of complexities like locks, synchronisation, asynchronous exceptions, etc., compared to their sequential counterparts. Static typed functional programming, provides a user with appropriate abstractions in the form of higher order functions, typeclasses etc to hide these complexities and make the syntax much more declarative. Consequently, there has been a number of publications on declarative parallel functional programming primarily using algorithmic skeletons [Col89] or redesigning distributed and parallel runtimes for Haskell [Tri+96; LOP05] to hide the intricacies of synchronisation. Unfortunately, vectorisation (which is not dependent on number of cores of the machines) has not received a lot of attention, except the now defunct, Data Parallel Haskell project [Cha+07], which served as a general umbrella for automatic parallelization using nested data parallelism [Pey+08].

Broadly, the dissertation attempts to answer the question: **Can a purely functional language leverage the power of vectorisation, preserving its original syntax and semantics while being as efficient as low level systems programming languages?** We discuss our attempts to extend the GHC code generator with the support to emit vector instructions, primarily the engineering challenges that we tackled. We also present a library [Sar18] which wraps over the basic vector types and provide polymorphic SIMD functions which operates using the vector registers. Followed by that we present a simple cost model to assess the effectiveness of vectorisation. Finally we talk about the Throttled SLP algorithm [PJ15], a prospective automatic vectorisation algorithm for GHC and possible changes in intermediate representations to accommodate the algorithm.

2. VECTORISATION

2.1 Introduction to Vectorisation

This section talks briefly about the history of vectorisation and explains it conceptually. The original inception of the idea of vectorisation began with the development of supercomputers whose processing units were called *vector processors.* Vector processor is a central processing unit that contains instructions that operate on one dimensional arrays of data called vectors, compared to scalar processors, whose instructions operate on single data items. Cray 1 [Rus78] was the first supercomputer to implement the vector processor design. Cray 1 supported registers of size 64 bits at a time (1975) when the conventional commodity machines supported 8 bit or 16 bit registers. Modern day *Single Instruction Multiple Data* parallelism as well as GPU technologies are incarnations of the original vector processor design.

With the inception of a number of powerful supercomputers and vector machines, MJ Flynn wrote a famous paper in 1966 classifying various types of parallel machines [Fly66]. Figure 2.1 demonstrates the various class of parallel machines.

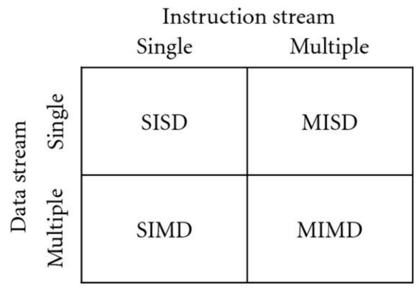


Figure 2.1: Flynn's Taxonomy

Let us elaborate on the nomenclature:

- `SISD` : Single Instruction Single Data
- `MISD` : Multiple Instruction Single Data
- `SIMD` : Single Instruction Multiple Data
- `MIMD` : Multiple Instruction Multiple Data

`SISD` implies a general sequential computer which executes a single instruction to operate on a single piece of data. `MISD` doesn't make a lot of sense and is not generally used. `SIMD` falls under the category of data parallelism, when a single instruction is able to operate on multiple slices of the same data which is commonly known as the vector machine approach or *vectorisation*. Finally `MIMD` generally implies any other parallelism approach like pipelining, multi core processing etc.

Keeping in mind the context of this report, vectorisation broadly involves loop unrolling followed by the generation of packed SIMD instructions supported by the hardware. These instructions operate on more than one data element at a time which results in better performance. From the Intel vectorisation manual [Dei+12],

> In Computer science the process of converting an algorithm from a scalar implementation, which does an operation one pair of operands at a time, to a vector process where a single instruction can refer to a vector (series of adjacent values) is called vectorisation.

Depending on the hardware, there are various available widths of registers as well as a variety of vector operations supported. If we assume a 128-bit wide vector register, it allows us to operate on four 32-bit wide floats. We can visualize it in Figure 2.2

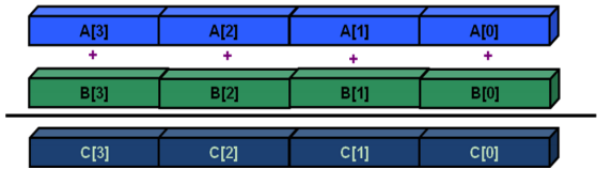


Figure 2.2: Vector addition

Although the figure demonstrates four addition operation, using an example vector instruction like *VADDPS* results in the addition of four 32-bit floats with one operation.

2.2 Vectorisation in the x86 instruction set

SIMD instructions are available in almost all major hardware vendors like Intel, ARM, AMD etc. For the purpose of this report, we essentially focus on the x86 instruction set. x86 is the de facto instruction set supported on all Intel and AMD processors. The very first SIMD support in x86 was brought in the form of *Matrix Math Extension (MMX)* [PW96] in 1996, which provided instructions operating on 64-bit registers. A number of graphic applications during that time utilised 8-bit or 16-bit registers which allowed MMX operation to operate on eight or four elements at a time. However, due to certain technical difficulties regarding the x87 FPU, the floating point support for MMX was disabled and this did not allow a number of complex math operations like logarithm, exponential, etc.

The newer reincarnation of SIMD support in Intel came in the form of the *Streaming SIMD Extensions* or the *SSE* family of instructions in 1999, which supported about seventy new vector instructions. SSE was primarily focused on enabling floating point support as most complex numerical and graphical applications utilise floating point operations. It introduced a new class of registers called the XMM registers which were each 128-bit wide.

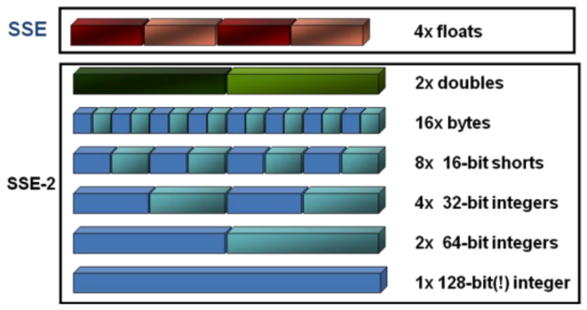


Figure 2.3: An XMM register

The term ”streaming” in SSE refers, to the representation of the data as a stream of infinite input, which is fed to each individual instruction and operated parallelly. Figure 2.4 demonstrates the concept visually. Followed by SSE, a number of extensions were made to the SIMD family in the form SSE 2, SSE 3, SSSE3 which included integer support for vector instructions, new operations like broadcast, shuffle, etc. This is relevant to us because in some of the following chapters we will witness an interplay between these various families of SIMD instructions.

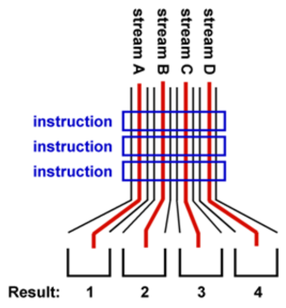


Figure 2.4: Streaming SIMD Extensions

Followed by SSE, Intel introduced the *Advanced Vector Extensions (AVX)* [Fir+08] family of instructions in 2008. AVX introduced a number of new instructions and provided even better single instruction support for vectorisation. For instance, broadcasting of floats to an XMM register can be done by a single AVX operation (*VBROADCASTPS*), instead of the complex bit twiddling techniques in the SSE family. However the primary difference was the operator-operand notation. Originally SSE family of instructions followed the notation:

$$Operator \quad R_1 \quad R_2$$

which roughly translates to $R_1 = R_1$ '*operator*' R_2. The AVX family notations are of the form:

$$Operator \quad R_1 \quad R_2 \quad R_3$$

which is read as $R_1 = R_2$ '*operator*' R_3. This notation makes the instructions much more readable. Note, that for this as well as the entirety of the report we shall choose the Intel syntax for assembly instructions rather than the AT&T instruction format. This is for the convenience of translating assembly instructions from the Intel instruction manual. However, GHC as well as GCC, clang and all major compilers use the AT&T notation while emitting the assembly and the GCC assembler understands the AT&T notation. This is a source of major confusion while reading assembly, so we discuss the notations in detail in Appendix A.

2.3 Loop Vectorisation

One of the primary targets of vectorisation is loops. Loop vectorisation transforms multiple scalar operations into single vector operations. It achieves this by unrolling the loops followed by packing the elements in vectors and operating on them. Let us take an example, assuming that we are operating in an array of size sixteen elements and performing simple addition of two arrays and storing the result in a third array. In pseudocode notation we have:

```
for (int i=0; i<16; ++i)
    C[i] = A[i] + B[i];
```

Loop unrolling transforms this program to:

```
for (int i=0; i<16; i+=4) {
    C[i]   = A[i]   + B[i];
    C[i+1] = A[i+1] + B[i+1];
    C[i+2] = A[i+2] + B[i+2];
    C[i+3] = A[i+3] + B[i+3];
}
```

Notice the change in the stride of the loop. We traverse four elements at a time. Vectorisation now packs four of these elements into a vector type and operates on them using a single operation.

```
for (int i=0; i<16; i+=4) {
    vec_A = pack(A[i],A[i+1],A[i+2],A[i+3]);
    vec_B = pack(B[i],B[i+1],B[i+2],B[i+3]);
    vec_C = vectorAdd(vec_A,vec_B);

    C[i]   = unpackToIndex(vec_C, i)
    C[i+1] = unpackToIndex(vec_C, i)
    C[i+2] = unpackToIndex(vec_C, i)
    C[i+3] = unpackToIndex(vec_C, i)
}
```

The original scalar version of the program used a total of sixteen addition operations. The vectorised version of the program uses four vector addition operations so it results in a save of twelve instructions on a data size of sixteen elements. This should imply a 3X speed straight away, but the cost model is not so simple. Although the number of operations are reduced we have added *pack* and *unpack* operations and they have their own overhead. In fact in such a small data size of sixteen elements, the cost of *packing-unpacking* will dominate over the gains that we have for vectorisation. In a larger data size of 1024 elements or even bigger data sets the gains of vectorisation will outweigh the loss of packing-unpacking. Also nested loops provide further challenges while vectorising.

We discuss about the compiler cost models as well as loop optimisation strategies in depth in the upcoming chapters.

2.4 Vector Machines vs SIMD vectorisation

Although we discussed the concepts of vectorisation in the context of vector machines as well as SIMD units in commodity computers in the same breath, a lot of the vectorisation theory from vector machines does not directly translate well to SIMD based vectorisation. Firstly, there are a number of architectural differences between the traditional vector machines and the SIMD units of computation. As stated in the vectorisation paper of GCC by Naishlos et al [Nai04],

> SIMD memory architectures are typically weaker than those of traditional vector machines.The former generally only support accesses to contiguous memory items, and only on vector length aligned boundaries. Computations, however, may access data elements in an order which is neither contiguous nor adequately aligned. SIMD architectures usually provide mechanisms to reorganise data elements in vector registers in order to deal with such situations. These mechanisms (packing and unpacking data elements in and out of vector registers and special permute instructions) are not easy to use, and incur considerable penalties.

The boundary alignment issues are among some of the important engineering problems that we have to tackle in the code generator. The memory access as well as update becomes much more involved and the onus is not only on the compiler writers to write the most efficient forms of accesses and updates, but also equally on the library authors to reduce packing-unpacking as much as possible. This is quite a challenging task, which we shall witness in the chapter discussing our *lift-vector* library.

An additional constraint, is presence of a number of domain specific instructions as well as the variety of micro-architectures supporting different forms of these instructions. We tackle most of these problems in the code generator and describe most of the high level choices that we make in the following chapters.

After giving a general overview of vectorisation as well as discussing the challenges very briefly, we now switch over to talking about Haskell and its compiler in the next couple of chapters.

3. A BRIEF TOUR OF HASKELL

This chapter comprises of a whirlwind tour of Haskell. This is principally targeted towards audience who are absolutely unfamiliar with the language. Interested readers are requested to read much more detailed treatises of the subject like *Programming in Haskell*[Hut16] by Graham Hutton. Experienced Haskell programmers can safely skip this section, although certain newer extensions of the original Haskell 98 specification like *type families* are also described in this part.

Haskell handles all values and data as immutable by default, which makes it a great language to express concurrency and parallelism. One of the most distinctive features of Haskell is it's order of evaluation. Unlike almost every major programming language in existence, all Haskell expressions are lazily evaluated. In more formal denotational semantics terminology, Haskell supports *non-strict* semantics, which allows it to bypass undefined values and in a way represent infinite data in the language. A number of advantages of lazy evaluation as well as immutability like modular code, is discussed in detail in a seminal paper by John Hughes [Hug89].

In addition to the evaluation order a unique feature of Haskell is that, it is a *pure* functional programming language. The purity or impurity of a language is a topic of much debate [Sab98], however to simplify the discussion we say that Haskell is a languages where side effects are not allowed. Any *effectful* function is expressed in the type system using the concept of a monad [Wad90].

One of the crown jewels of Haskell is its powerful type system. Haskell makes a very clear distinction between the *value level* and the *type level*. At the same time it allows advanced users, a lot of flexibility to blur the lines between values and types. The fundamental building block of the Haskell type system is algebraic data types (ADTs). They are basically composite data types which are composed of other data types. Algebraic Data Types are described as *sum types* or *product types*.

Figure 3.1 demonstrates a basic ADT. The keywords *data* as well as *newtype* are used to construct a new data type. The first string after an equal sign is the data constructor which is equivalent to object constructors in object oriented languages.

Now we take a look at some different ways of constructing types below:

```
-- An example of enumeration:
```

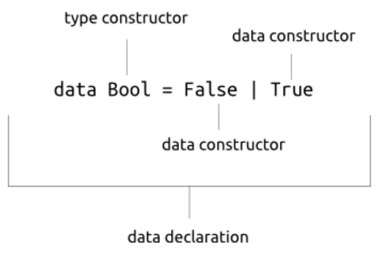


Figure 3.1: An algebraic data type

```haskell
data Season = Spring | Summer | Autumn | Winter

-- An example of a product type
data Pair = Pair Int Int

-- An example of a sum type
data Shape = Circle Float | Rec Float Float

-- Polymorphic and recursive
data Tree a = Leaf a
            | Node a (Tree a) (Tree a)

-- Type Synonyms
type Weather = String
> "warm" :: Weather

-- Newtypes
newtype Weather = Weather String
> Weather "warm" :: Weather
```

A *newtype* doesn't have any runtime overhead, however it is constrained to have only a single value constructor and a single value.

Typeclasses : Now we look at one of the most unique and novel features of Haskell called *typeclasses* [WB89] which allows us to utilise a very ad hoc form of polymorphism. We use the term *ad hoc* to imply that this type of polymorphism is not shipped as a fundamental feature of the type system and it is glued on top of the language.

```
 class (Eq a) => Ord  a  where
   compare :: a -> a -> Ordering

data Ordering = EQ | GT | LT
```

The above snippet creates a typeclass *Ord* that is parameterized on some polymorphic type *a* which itself must have an *Eq* typeclass instance. The function that a data type needs to implement this typeclass is the *compare* function which takes in two polymorphic types *a* and returns some value of type *Ordering*. In OOP terminology, a typeclass is crudely an analogue of an interface with extended functionalities. This is an important concept because we heavily use typeclasses in our *lift-vector* library.

We finally introduce the concept of *associated types* or *type families* [Cha+05] which is another important concept for understanding the *lift-vector* library as well as certain portions of the literature review.

Associated Types : The type system of Haskell allows us to perform a powerful form of *ad-hoc* overloading using typeclasses. We might further want to modify the data representation and algorithms at the type level while depending on the structure of the data, and associated types provide exactly the support for that. A very motivating use case of associated types was first given by Ralf Hinze [Hin00] in the representation of the *generalised trie* or *generic finite maps*. These maps change their representations depending on the key type k used to index the map.

```
class MapKey k where
  data Map k v  -- the Associated type
  empty :: Map k v
  lookup :: k -> Map k v -> v
```

The above snippet looks like an ordinary type class except the additional *associated type* declaration which forms a codomain of the Map. Effectively the representation of the map depends on the type *k* of keys, while it is also parametrically polymorphic in the value type v. The same constraint could also be expressed using *Multi Parameter* typeclasses and *functional dependencies* in Haskell 2010, but associated types have a cleaner syntax and is more expressive than functional dependencies which is discussed in detail in the paper by Chakravarty et al [Cha+05]. So depending on the instances the representation of the `Map` type would vary like below:

```haskell
instance MapKey Int where
  data Map Int v = MapInt (Patricia.Dict v)
  empty = MapInt Patricia.emptyDict
  lookup k (MapInt d) = Patricia.lookupDict k d

instance MapKey () where
  data Map () v = MapUnit (Maybe v)
  empty = MapUnit Nothing
  lookup Unit Nothing = error "unknown key"
  lookup Unit (Just v) = v

instance (MapKey a, MapKey b) ) =>
    MapKey (Either a b) where
  data Map (Either a b) v =
    MapEither (Map a v) (Map b v)
  empty = (Nothing, Nothing)
  lookup (Left a) (fm1, fm2)  = lookup a fm1
  lookup (Right b) (fm1, fm2) = lookup b fm2
```

Another relevant point is the difference between *data families* and *type families* which syntactically differ only in the use of the keyword *data* and *type* respectively. However the former constructs a new type like the instance definitions given above while the latter acts exactly like a type synonym without creating a new type. Type families are hence non-injective in nature. Injective type families were introduced in GHC 8 and are beyond the scope of this report.

We use associated types to create a polymorphic API for working with the multitude of vector data types and tuples in our *lift-vector* library. This concludes our tour of Haskell and in the next section we shall look at the internals of the Glasgow Haskell Compiler.

4. THE GLASGOW HASKELL COMPILER

The Glasgow Haskell Compiler [Jon+93] which shall henceforth be addressed as GHC, is a state of the art optimising functional language compiler. GHC is an implementation of the Haskell 2010 language spec along with the additional languages extensions. GHC is a *bootstrapping* compiler which means the compiler is itself written using the host language. The compiler contains close to 715,000 lines of Haskell and the runtime is written in about 85,000 lines of C. GHC is an old and formidable piece of codebase which has evolved to its current form over the last thirty years. The entire GHC compilation pipeline is summarised in the figure below. Image taken from [BW12].

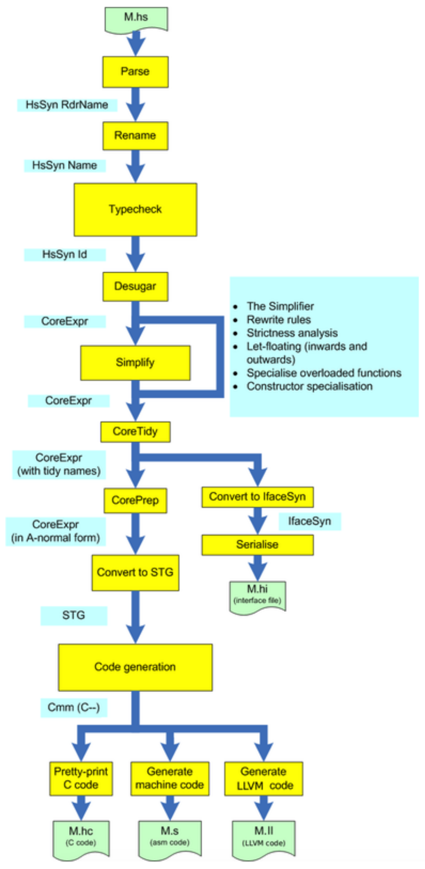


Figure 4.1: GHC compilation pipeline

Figure 4.1 gives a detailed overview of all the phases that a source file written in Haskell goes through. Each of the phases are discussed below:

- .hs files : The original source code is written in a *.hs* file which contains the syntax of the entire Haskell language specification. The parser parses this file.

- Parsing: GHC uses its own lexer and parser known as Alex and Happy respectively, both authored by Simon Marlow, which provides a functional API to the generally imperative alternatives Flex, Yacc, Bison etc.
- Renaming: This phase renames each of the identifiers to their fully qualified names, while identifying any out-of-scope identifier and raising the necessary errors.
- Type Checking : In this phase the types of each of the identifiers are checked to see if the program is type correct. The input to the type checker is `HsSyn Name` and the output is `HsSyn Id`, where `Id` is a Name with extra information: notable a type.
- Desugaring : In this phase all the syntactic sugar is shed and the full blown Haskell language is compiled down to a small intermediate language called Core. The Core language is a variant of lambda calculus known as System F [Rey74]
- Optimization passes : Followed by desugaring a number of optimization passes takes place in the form of the *Simplifier* which performs a series of correctness-preserving transformations.
- STG : The spineless tagless graph machine is an *abstract machine* which represents the execution model of a lazy language. We shall talk about STG and its following phases in a little more detail, as the majority of our code changes reside there.

Spineless Tagless G Machine : STG as a language, is even smaller than the Core language. It is entirely composed of *let* blocks and *case* clauses. *let* blocks are used for laziness and *case* clauses for eager evaluation. The STG program exists in an intermediate representation form know as *Administrative Normal Form* or *ANF* [SF93]. In ANF, all arguments to a function must be trivial. That is, evaluation of each argument must halt immediately.

The STG machine comprises of three major parts :

- STG registers : STG supports a set of virtual registers for manipulating the heap and stack, as well as generic registers for argument passing between function calls. GHC can handle these virtual registers by either storing them all on the Heap or pinning some of them to a certain hardware registers, depending on the portability.

Most of the STG registers actually reside on the stack in a block of memory pointed to by a special STG register called the *BaseReg*.

- STG stack: A region which grows downward in the memory. The STG stack is responsible for storing function arguments and continuations.
- STG heap : The major region which grows upwards in memory, towards the stack. It is responsible for storing functions, thunks and data constructors.

Haskell Execution Context : We talk about the actual machine code generator, after we discuss how the GHC runtime models STG. It does so in the form of what is called a *Haskell execution context* or *capability* of the runtime system. Each capability holds all of the information like the region of the heap in the nursery region or additionally what objects needs to be garbage collected. At the operating system level a capability maps to an individual OS thread and each thread maintains all the STG information mentioned above. We can visualise this translation between STG , the Haskell Execution Context and the actual machine registers in Figure 4.2. (Figure from [T12])

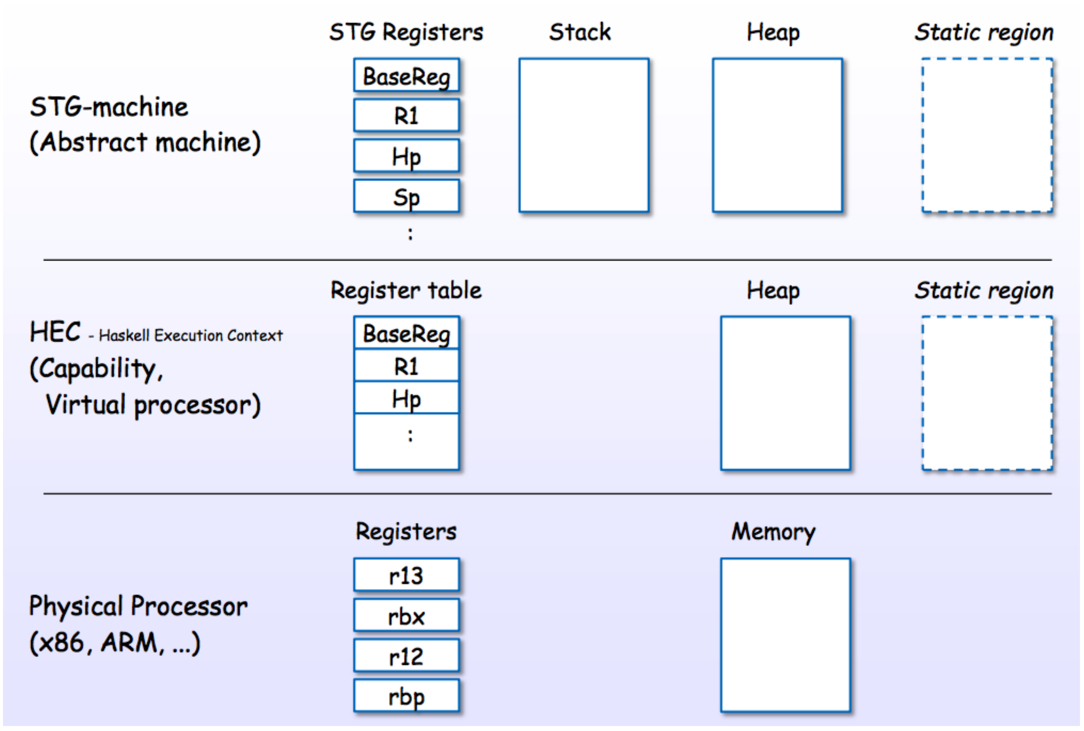


Figure 4.2: A Capability or a Haskell Execution Context

Code Generation: The STG code is finally translated to Cmm (or C minus minus) which is the final low level intermediate representation before the actual machine code generation. GHC's implementation of Cmm is based on the original C−− language spec [JRR99], with numerous changes. Cmm is a low level language which almost resembles C but has support for low level features like labels. Cmm also supports has an explicit stack to enable tail recursion. Cmm is a typed language. It primarily supports two types of values: bit vectors or floats. Bit vector is itself a polymorphic type and may come in several widths, e.g., *bits8, bits32* , or *bits64*. Additionally there is also the *bool* type.

We demonstrate certain sample Cmm programs:

```
/* Ordinary recursion */
export sp1;
sp1( bits32 n ) {
 bits32 s, p;
 if n == 1 {
   return( 1, 1 );
 } else {
  s, p = sp1( n-1 );
  return( s+n, p*n );
 }
}

/* Tail recursion */
export sp2;
sp2( bits32 n ) {
 jump sp2_help( n, 1, 1 );
}

sp2_help( bits32 n, bits32 s, bits32 p ) {
 if n==1 {
  return( s, p );
  } else {
  jump sp2_help( n-1, s+n, p*n )
 }
}
```

Cmm, while being relatively low level, still manages to abstract the gory details of assembly programs and even at this layer the actual machine level registers remain hidden. Almost half of our work lies in adding vector machine operators at the Cmm level and then parsing it to emit the correct assembly instructions, depending on the backend. GHC supports three backends:

- C backend : The C backend was the first backend written for GHC and it exists for the purpose of portability. Its performance as well as compilation times are much slower than the other backends. There are number of other issues related to poor performing optimisation passes in the assembly generated, which doesn't make it the best backend to implement vectorisation.
- Native code generator (NCG) : The most highly used and performant backend is the so called *native code generator*. This code generator supports a number of architectures like x86, x86_64, SPARC and Power PC. We primarily target the x86_64 and x86 vector instructions. However our general assumption is always a word size of 64 bits, and in a later section we shall see how we introduce integers of various other widths in GHC, to avoid any word size dependence confusion, which allows us to safely emit vector instructions for x86 as well as x86_64.
- LLVM backend : The LLVM backend is a relatively newer entry which emits LLVM bit code, which further undergoes all the LLVM related optimizations including vectorisation. In fact the LLVM backend is the only existing backend which can emit vector instructions. The Cmm code in its final phase is transformed into a CPS(continuation passing style) form and the backend emits the SSA(static single assignment) form LLVM IR. This overhead of the translation from CPS to SSA form, while rectifying the explicit stack happens through the *mem2reg* pass of LLVM. However the IR emitted is not optimal as detailed in [TC10]. We discuss about the translation overhead between these intermediate representations and provide some possible changes in the IR languages of GHC in the final chapter of this thesis. LLVM itself does automatic vectorisation and there need not be any changes in the code generator of GHC for vectorisation. However this leads to very poor quality vector code because a lot of relevant information is lost while translating between the IRs. In our case, we fundamentally change the code generator and the Cmm layer to understand vector operations. Additionally the existing LLVM backend is being

entire re-implemented[dev13] owing to a number of bugs related to code generation. Hence we choose to work with the x86 native code generator.

This brings us to the end of the discussion on the various phases of the Glasgow Haskell Compiler. Before we begin to demonstrate our work on the GHC source code, in the next section, we talk about a couple of important papers that we have reviewed in the context of vectorisation and its benefits in Haskell.

5. LITERATURE REVIEW

In this chapter we look at two papers published in 2013, which attempt to address the advantages of vectorisation in Haskell. The first paper studies the Intel Haskell Research Compiler [Liu+13] which intercepts the output of the *Core* language from GHC and applies some of its own optimisations which benefit array computations. The compiler then proceeds to emit a C like language called Pillar [And+07] which can be compiled with both GCC as well as the Intel C compiler.

The second paper on the other hand restricts itself to GHC and tries to leverage the possible vectorisation from the LLVM backend. The work in this paper is more high level, it talks about a new representation of streams [Hin08] which can utilise the vectorisation offered by the compiler. The work in this paper is relevant to the *lift-vector* library that we have designed.

5.1 The Intel Haskell Research Compiler

We discuss the paper *Automatic SIMD vectorisation for Haskell*[POG13] by Peterson et al, which works on the Intel Haskell Research Compiler (IHRC). The IHRC compiler utilizes a *static single assignment form or SSA* based representation as its intermediate language. An SSA form representation [RWZ88] is more common in compilers for imperative languages like Fortran or C. Functional languages like Haskell use different forms of intermediate representations(IR) like *continuation passing style or CPS* [Rey72] or *administrative normal form or ANF* [SF93] style of representation for their intermediate languages. This is a unique choice of IR for a functional language compiler.

An SSA form enforces that every variable is assigned exactly once and it is initialised before its use. For example:

```
y := 1
y := 2
x := y
```

In SSA the above becomes:

```
y1 := 1
```

```
y2 := 2
x1 := y2
```

As a result, the control flow graph [All70] for the program becomes relatively simpler to analyse. In the control flow graph of a program the loop becomes the strongest connected component of the graph. Most of the compiler is designed to work with mutable and immutable arrays, with the optimisations focused on the latter. All general compiler optimisations like inlining, contification[FW01], loop-invariant code motion and a general simplifier [AJ97] are present in the compiler. The compiler targets a variant of C known as Pillar [And+07]. The pillar code is finally translated using the Intel C compiler or GCC to emit the machine code. All of this is linked with a small run-time supporting garbage collection.

5.1.1 Working with Arrays

Apart from the different intermediate representation, a major point of difference is the handling of immutable arrays. Haskell provides certain high level libraries like Data.Vector and REPA [Kel+10] to work with general immutable arrays. However the handling of these arrays are very different compared to imperative languages. The arrays are represented as streams and the operations undergo a powerful optimisation called *Stream Fusion* [CLS07].

5.1.2 Stream Fusion

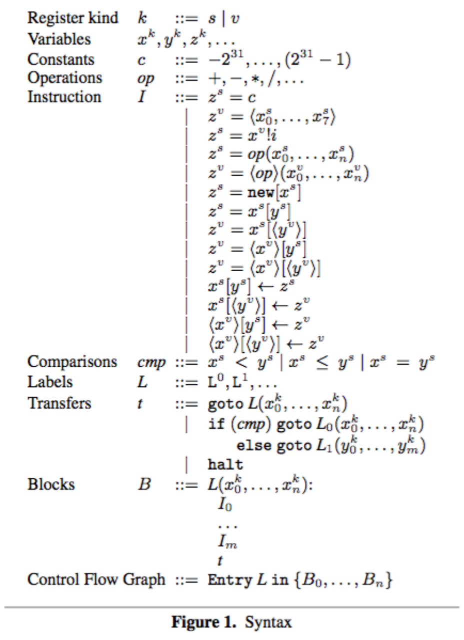
Higher order functions initially turns the array into a stream based representation (which we discuss in depth in the next literature review). The function is applied to individual element of the stream, followed by which a mutable array of the original size is created. Each element of the array is successively initialised using array updates and finally this mutable array is *frozen* to an immutable array type. This action of *thawing* and *freezing* a data structure in Haskell happens inside the ST Monad [LP94]. GHC utilises aggressive inlining and multiple simplification functions to eliminate all the intermediate structures created.

Unfortunately this form of mutable array creation and consumption is hard to optimise for the IHRC compiler. To handle this the IHRC compiler extends GHC with a primitive immutable array type and a special kind of operation termed as *initialising write*. According to the paper, The two invariants of initialising writes are that reading an array element must always follow the initialising write of that element, and that any element can only be initialised once. This style of write preserves the benefits of immutability from the compiler standpoint, since any initialising write to an element is the unique definition of that element.

As a result of the above two invariants, IHRC is able to work with immutable arrays and can vectorise loops which operates on these arrays. Loops generally comprises of a major part of a program. The purpose of the loops are to initialise these immutable arrays or perform reductions over them. IHRC specifically targets these loops and tries to generate `SIMD` code for these loops.

5.1.3 The IHRC Core Language

One fundamental novelty of IHRC is, it defines its own vector core language which is based on array computations. We will spend some time discussing the core language of IHRC as it can serve as a great template for adding auto-vectorisation support to GHC (which we discuss in the final two chapters). The entire syntax of the core language and operations of the compiler, as defined in the paper, is given in Figure 5.1



$$
\begin{array}{llll}
\text{Register kind} & k & ::= & s \mid v \\
\text{Variables} & x^k, y^k, z^k, \ldots & & \\
\text{Constants} & c & ::= & -2^{31}, \ldots, (2^{31}-1) \\
\text{Operations} & op & ::= & +, -, *, /, \ldots \\
\text{Instruction} & I & ::= & z^s = c \\
 & & \mid & z^v = \langle x^s_0, \ldots, x^s_7 \rangle \\
 & & \mid & z^s = x^v ! i \\
 & & \mid & z^s = op(x^s_0, \ldots, x^s_n) \\
 & & \mid & z^v = \langle op \rangle (x^v_0, \ldots, x^v_n) \\
 & & \mid & z^s = \mathtt{new}[x^s] \\
 & & \mid & z^s = x^s[y^s] \\
 & & \mid & z^v = x^s[\langle y^v \rangle] \\
 & & \mid & z^v = \langle x^v \rangle [y^s] \\
 & & \mid & z^v = \langle x^v \rangle [\langle y^v \rangle] \\
 & & \mid & x^s[y^s] \leftarrow z^s \\
 & & \mid & x^s[\langle y^v \rangle] \leftarrow z^v \\
 & & \mid & \langle x^v \rangle [y^s] \leftarrow z^v \\
 & & \mid & \langle x^v \rangle [\langle y^v \rangle] \leftarrow z^v \\
\text{Comparisons} & cmp & ::= & x^s < y^s \mid x^s \le y^s \mid x^s = y^s \\
\text{Labels} & L & ::= & \mathrm{L}^0, \mathrm{L}^1, \ldots \\
\text{Transfers} & t & ::= & \mathtt{goto}\ L(x^k_0, \ldots, x^k_n) \\
 & & \mid & \mathtt{if}\ (cmp)\ \mathtt{goto}\ L_0(x^k_0, \ldots, x^k_n) \\
 & & & \quad \mathtt{else\ goto}\ L_1(y^k_0, \ldots, y^k_m) \\
 & & \mid & \mathtt{halt} \\
\text{Blocks} & B & ::= & L(x^k_0, \ldots, x^k_n): \\
 & & & \quad I_0 \\
 & & & \quad \ldots \\
 & & & \quad I_m \\
 & & & \quad t \\
\text{Control Flow Graph} & & ::= & \mathtt{Entry}\ L\ \mathtt{in}\ \{B_0, \ldots, B_n\}
\end{array}
$$

Figure 1. Syntax

Figure 5.1: The IHRC core language

As we can see in the figure, there are two kinds of register given by $k ::= s|v$. `s` stands for scalar and `v` stands for vector. For the sake of simplicity the paper assumes a 32 bit machine with 32 bit width scalar registers and 256 bits wide SIMD registers. Handling other register widths and different micro architectures is a separate problem statement. We are more interested in the high level core language presented in the paper.

Most of the operations are very common and most of the variety of instructions lie in the loading and storing of various kinds of elements to the various types of registers.

The unique notion of immutability in the array types (denoted by $x^s[y^s]$ or $x^s[\langle y^v \rangle]$ etc) is itself an unchecked invariant of the language such that "every element of the array is initialised before it is read, and that every element of the array is initialised at most once." Hence immutability of the array type is obtained by constraining the write operation on arrays to be simply initialising write and not mutation.

Some notable operations are the *scatter* and *gather* operations which are very common in parallel computing. They are defined above as:

$z^v = x^s[\langle y^v \rangle]$

It takes a single array x^s and a vector of offsets y^v and binds them all together into the vector z^v. This operation can be seen as a "gathering" of multiple vectors and is called the *gather* operation. The dual of this operation is called *scatter* which is:

$x^s[\langle y^v \rangle] \leftarrow z^v$

and it writes or *scatters* the elements of the array z^v to the array of offsets. Generally the elements won't be laid out uniformly in the memory, however in the cases that this happens, the core language adds a special instruction support for vectors where stride in the layout of the memory is known to be one. Many architectures support these idioms directly, as a result of which the support of these instructions provide an improved performance.

A full IHRC program contains a single control flow graph with a designated entry label L and bunch of other labels leading ahead from it. Programs keep on executing, traversing the control flow graph until it encounters a `halt` instruction. The style of single static assignment form used in IHRC is comparable to the MLton compiler [Wee06].

Reflection : The paper shows benchmarks where there are major gains compared to the scalar code from native code generator as well as the vectorised code from the LLVM code generator. One of the principal reasons for this performance gain was the ability to tap into the stream fusion optimisation. GHC additionally allows us to specify rewrite rules which can aid the fusion and remove intermediate data structures. Another principal takeaway was the separate core language design which natively supports parallel computing operations to enable vectorisation.

5.2 Exploiting vectorisation in streams

The theme of this review is much more high level compared to the previous literature on the vector core language. We look at the paper *Exploiting Vector Instructions with Generalised Stream Fusion* [MLP13] by Mainland et al, which crafts a stream representation which can leverage the vectorisation support from the hardware.

5.2.1 Representing Streams

The principal necessity of streams (aside from expressing infinite structures) arises from the fact that compilers are not good at optimising recursive functions. As a result, a stream based representation is used, so that the functions applied to the data structures are not recursive in nature. We present the stream type below:

```
data Stream a where
  Stream :: (s -> Step s a) -> s -> Int -> Stream a
--           --------------     ^     ^
--                 ^            |     |
--                 |            |    size
--                 |     existentially quantified state
--          a step function which when given the state produces a step

data Step s a = Yield a s
              | Skip s
              | Done
```

A step either yields a new element and a new state, or demarcates the end of the stream, or skips producing a new element but returns the state. We can encode infinite structures like this using the `Yield` constructor as a *generator*.

The advantage of this representation is that, the higher order functions are no longer recursive. It can be demonstrated using the basic example of a `map` function over vectors from the Data.Vector library, which looks like this:

```
map ::(a -> b) -> Vector a -> Vector b
map f = unstream . maps f . stream
```

The auxiliary map_s function expresses the original recursive function in a non recursive fashion.

```
maps :: (a -> b) -> Stream a -> Stream b
maps f (Stream step s) = Stream step' s
  where
   step' s = case step s of
      Yield x s' -> Yield (f x) s'
      Skip s'    -> Skip s'
      Done       -> Done
```

The *stream* and *unstream* converts the vector to a stream representation and vice versa, respectively. Now this allows us to derive the compositional laws for map, using equational reasoning;

$map\ f \circ map\ g$

$\equiv unstream \circ map_s\ f \circ stream \circ unstream \circ map_s\ g \circ stream$

$(given\ stream \circ unstream = id)$

$\equiv unstream \circ map_s\ f \circ map_s\ g \circ stream$

As previously mentioned GHC supports something called *rewrite rules* [JTH01] which precisely allow us to express algebraic identities like $stream \circ unstream = id$. Stream fusion utilises these kind of rewrite rules to eliminate intermediate structures. Digressing slightly, this is one of the major advantages of having a datatype be an instance of a law abiding type class. We can capture these laws as *rewrite rules* which the GHC simplifier can heavily use to optimise and eliminate intermediate structures.

5.2.2 A vector stream representation

The stream representation clearly yield one element at a time and any data structure built around that notion would not be amenable to utilise the underlying vectorisation of the hardware. We need a presentation which yields n elements at a time, where n is the vector width. For vectorising 32-bit floats on an XMM register we would want the stream to yield four elements at a time. But what happens when the size of the stream element is not a multiple of four? And how does the number of elements yielded adjusts with the width? To answer the above questions Mainland et al, presents a vectorised stream representation which represents a stream as a *bundle* of streams.

```haskell
data Bundle a = Bundle
  { sSize :: Size
  , sElems :: Stream a
  , sChunks :: Stream (Chunk a)
  , sMultis :: Multis a}
```

The first field `sSize` denotes the size of the stream and the field `sElems` represents the original stream as denoted by the type of the field. The field `sChunks` enables the usage of bulk memory operations.

```haskell
data Chunk a = Chunk Int (forall s. MutableVector s a -> ST s ())
```

The definition of `Chunk` utilises the `MutableVector` type which provides a function called `copyMVector` which internally uses the `memcpy` instruction to copy the vector. This vector notably runs inside the ST Monad [LP94] which uses the concept of *thawing and freezing* as mentioned in the Section 5.1.2 to optimise the process.

While the `Chunk` representation can utilise the `memcpy` system call to optimise *vector append*, it is not the best possible representation for operations like *zipWith, fold* etc which are the best targets of vectorisation. The `sElems` field allow sequential stream operation but the vector variants are accessed in the `sMultis` field.

Broadly, the *Multis* type uses a type family (which was discussed in chapter 3) to depend on the vector width size and also on the producer and consumer demand rate to create a vector representation. The derivation of the entire *Multis* type is quite involved and the interested reader is encouraged to read the original paper [MLP13] for more details.

Reflection : In this section we saw that, the derivation of a data representation which is amenable to leverage the vectorisation is quite involved. Recursion is the bread and butter of functional programmers, however due to the compiler not being able to optimise recursive function, the authors had to jump through quite a few hoops to create a stream representation. We shall take inspiration from the above design to create the data representations of our *lift-vector* library discussed in chapter 7.

6. VECTORISATION IN GHC

The major part of this thesis has been tackling the engineering challenges in retrofitting the vector instruction support to the native code generator. We use the term *retrofitting* because the original design of Cmm did not account for the existence of vector operations. As a result, to leverage the multiple optimisation passes happening in Cmm itself, we have to fit the vector instructions in a relatively awkward form as we shall see in the following sections. We shall start by looking at the code generation pipeline in more detail.

6.1 Code generation pipeline

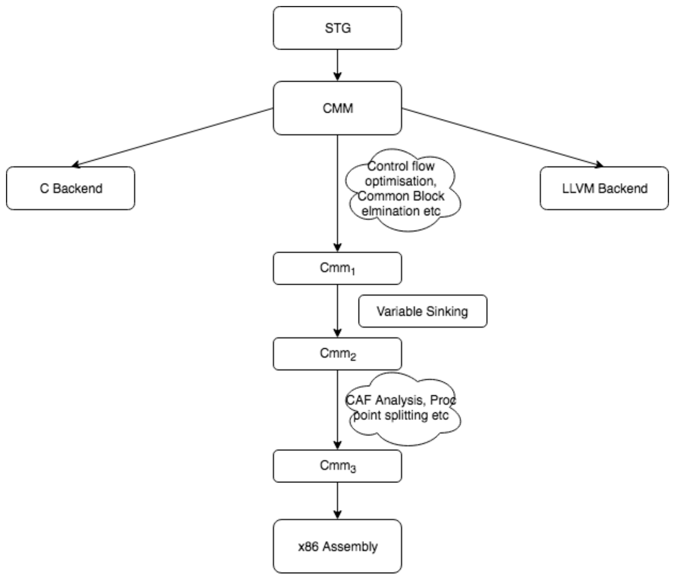

STG
CMM
C Backend
Control flow optimisation, Common Block elmination etc
LLVM Backend
Cmm$_1$
Variable Sinking
Cmm$_2$
CAF Analysis, Proc point splitting etc
Cmm$_3$
x86 Assembly

Figure 6.1: The Code generation pipeline

In Figure 6.1 we zoom in on the general code generation pipeline where most of our work resides. Majority of our work on `Floats` and `Doubles` is limited to this region, however when adding support for more precise integers we end up touching the type checker as well as certain portions of the Core expressions.

In the above figure each of the blocks with a subscript Cmm$_n$ takes a Cmm program as input and return a correctness preserving Cmm program with each of the optimisation pass applied. There are a number of optimisation passes in Cmm like:

- Control Flow Optimisation

- Common Block Elimination
- Proc-point splitting [Cli12]
- Layout stack
- Variable Sinking
- CAF analysis
- Convert Cmm to CPS style

We demarcate *variable sinking* specifically in the figure because the final form of the STG expression, of the vector operations, were specifically designed so that they could leverage the variable sinking pass. Other small changes were made in each of these individual passes to allow them to optimise the vector operations as well.

6.2 GHC PrimOps and PrimTypes

The name *PrimOps* is short for *Primitive Operations* in GHC. PrimOps and PrimTypes are functions and data types respectively, that cannot be implemented in Haskell and are provided natively by GHC. The distinction between *Haskell* and *GHC* should be noted in the last statement. Haskell, in itself, is simply a language specification [Mar+10; Jon03] which details the existence and specifications of fundamental data types like `Int, Float, Double`. On the other hand a compiler writer is free to implement these data types using as many optimisations possible, while sticking to the core specifications. For example the Haskell 98 report states,

`Int` is a fixed-precision integer type with at least the range $[-2^{29}..2^{29}-1]$.

The specification states the minimum range. A fixed precision integer, as provided by GHC, is machine dependent and has size 64-bits on a 64 bit machine or 32-bits on a 32 bit machine. It makes sure that it obeys the original specification but optimises the representation accordingly. Likewise other Haskell compilers like Yale Haskell uses 31-bits to represent `Int` (the 32nd bit is used for tagging.)

The primops and primtypes are made available as a virtual module called `GHC.Prim`. This module does not exists on the disk or the source tree. It is created during bootstrapping [1] the compiler. It parses the file `compiler/prelude/primops.txt.pp`. For each

[1]bootstrapping a compiler is compiling the compiler itself, using the same compiler!

primop in the file the following is defined:

- The actual operation name (eg. broadcastFloatX4#)
- The type signature of the operation
- The constructor name in GHC's PrimOp data type.
- An additional field to encode properties like `commutative`, `llvm_only` etc

An example of the integer multiplication primop is given below:

```
primop   IntegerMulOp   "timesInteger#" GenPrimOp
   Int# -> ByteArr# -> Int# -> ByteArr# -> (# Int#, ByteArr# #)
   with commutable = True
        out_of_line = True
```

The file `compiler/prelude/primops.txt.pp` is parsed by the source code in the module `utils/genprimopcode`. It contains a lexer, parser and code generator for the file, which generates the entire `GHC.Prim` module where the central data types reside. For adding vector types we shall modify the primops file. But however before proceeding to add the vector types we need to understand the notion of boxed, unboxed, lifted and unlifted types in GHC.

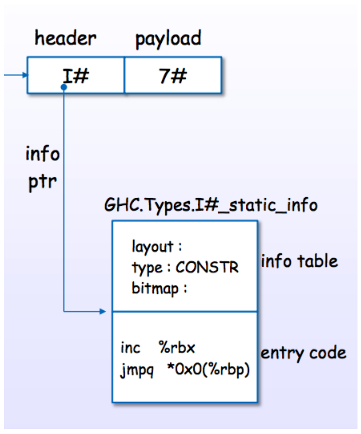
6.2.1 Boxed, Unboxed, Lifted, Unlifted types

The general data types in Haskell like `Int, Float` etc are what we refer to as `boxed` types. They are represented by a pointer to the heap, whereas unboxed types are just the values themselves. We take the example of an `Int` type, say the number "7". Its representation in memory is given in Figure 6.2. Figure from [T12].

The "7#" given in the figure represents the unboxed type. If we see how an `Int` is represented in GHC, it is like this:

```
data Int = I# Int#
```

The type `Int#` is what we call an unboxed type. It represents a raw machine integer. On a 64-bit machine it is 64-bit wide. In addition to the concept of boxing Haskell has a notion of *liftedness* or what is known in Haskell terminology as *levity*.



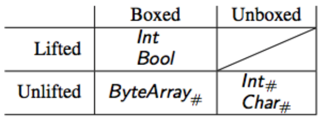
```
inc   %rbx
jmpq  *0x0(%rbp)
```

Figure 6.2: A lifted and boxed type represented in memory

A lifted type implies something that is lazy. Richard Eisenberg, a prominent GHC researcher states "It is considered lifted because it has one extra element beyond the usual ones, representing a non-terminating computation." The kind[2] of a lifted type is always '*'. On the other hand, an unlifted type is always strict. The kind of an unlifted type is '#'. So in Figure 6.2 "7#" is an unlifted and unboxed type, whereas the `Int` type "7" is lifted and boxed.

This begs to ask the question can a lifted type be unboxed? Indeed, it can be! A memory array is always represented with a pointer to a heap which makes it a lifted type. However, it representation in memory like the types ByteArray# is always strict. This makes it a lifted but unboxed type.

The relationship between these four types can be summarised using this table by Eisenberg from his paper on Levity Polymorphism [EP17].



	Boxed	Unboxed
Lifted	*Int* *Bool*	
Unlifted	*ByteArray*$_\#$	*Int*$_\#$ *Char*$_\#$

Figure 6.3: Lifted vs unlifted and boxed vs unboxed type

6.2.2 PrimTypes in Haskell

The previous section on lifted, unlifted, boxed and unboxed types provides us the vocabulary to talk about all the existing types and primops in GHC. From this point

[2]In type theory, a kind is the type of a type constructor or, less commonly, the type of a higher-order type operator.

onward, unless specified, we shall not make any distinction between lifted and boxed types or unlifted and unboxed type. This is because for all practical purposes of this report we won't be working with the lifted and unboxed types like ByteArray#.

In the section that follows this one, we shall talk about the necessary changes in the GHC source tree to add support for the vector types. Before that we want to compile a list of the fundamental primtypes that we have, which will help us create the vector types by composing these basic types.

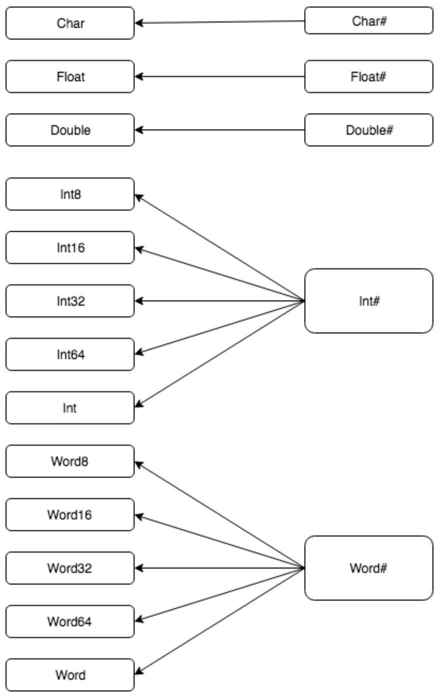


Figure 6.4: The core primtypes

Figure 6.4 represents the major numeric and character data types and the fundamental primtypes used to define them. We can already spot a fundamental issue that `Int8, Int16, Int32, Int64` and `Int` are all represented using the same `Int#` type which is always equal to the word width of the machine. We address this problem in the Section 6.4. Currently we shall take the example of `Float` which is defined as

```haskell
data Float = F# Float#
```

and discuss the support of vector instructions for `Float#` and the various other primops around it. Apart from the primtypes listed in Figure 6.4 there are also the following :

Table 6.1: Prim Types

GHC PrimTypes
Array#
SmallArray#
ByteArray#
MutableByteArray#
ArrayArray#
MutableArrayArray#
Addr#
MutVar#
TVar#
MVar#
State#
Proxy#

There are a number of other primtypes like Compact#(to represent compact normal forms) and a huge collection pointer primtypes, but for most of the discussion in this report we deal with the primtypes in Figure 6.4. The library *lift-vector* does deal with the primtypes like `Array#` or `ArrayArray#`[3] , however we choose to interact with the array types via a higher level library like *vector*[4].

6.3 Adding a sample vector instruction to NCG

In this entire section we shall demonstrate the changes necessary in GHC to accommodate a sample vector instruction. But initially, we change our context from the internals of GHC, to talk a little bit about our target architecture and an example instruction supported on that architecture.

6.3.1 The broadcast operation

Before we take a walk-through of the source code changes in the code generation pipeline to add a vector primop, in this section we will take a sample *broadcast* operation, and talk a little about the semantics of the operation. We discussed a superclass of *broadcast* in Section 5.1.3 of the IHRC compiler, called *scatter*, which is a common pattern in parallel computing.

There are a multitude of SIMD operations, spread across a number of micro-architectures and specialised domain specific instructions. We begin by picking the AVX family of SIMD

[3]array of arrays

[4]`http://hackage.haskell.org/package/vector`

instructions. AVX stands for *Advanced Vector Extensions* which was introduced with the Intel Sandy Bridge[5] micro-architecture introduced in 2011. The AVX family introduced sixteen 256-bit wide YMM registers. Most modern generation of Intel processors like Core - i3, i5, i7 are descendents of the Sandy Bridge micro-architectures. We run our experiments on a Macbook Pro 13" with a Core-i5 second generation processor. As discussed in the previous section we hope to apply the operation on an XMM register. In the Sandy Bridge micro-architecture, an XMM register is represented as the lower 128-bits of a YMM register.

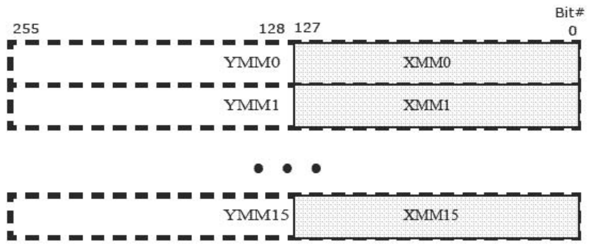


Figure 6.5: An XMM and YMM register

As we can see in Figure 6.5 the lower 128-bit of the YMM register is called an XMM register. Any operation on the XMM registers would involve zeroing out the upper 128-bit of the YMM register. Now we want to apply the broadcast operation, to broadcast four 32 bit floats to one XMM register. A broadcast operation can be demonstrated using Figure 6.6.

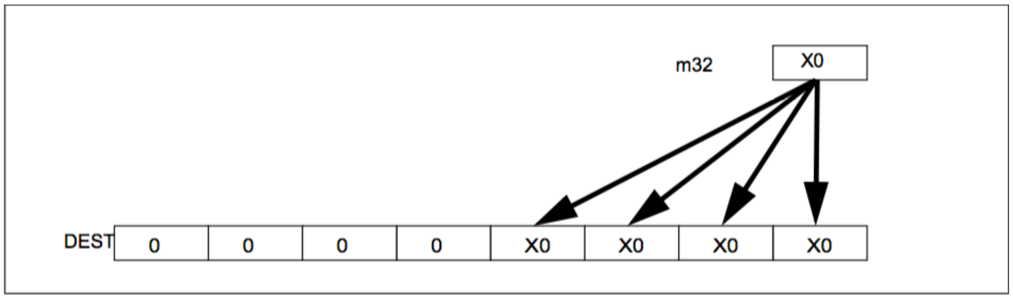


Figure 6.6: Broadcasting a float to an XMM register

Note the upper 32-bits of the YMM register has been zeroed out. The x86 instruction which provides us this operation is `VBROADCASTSS`. The syntax for the operation is,

```
VEX.128.66.0F38.W0 18 /r VBROADCASTSS xmm1, m32
```

[5]codename for the second generation of Intel Core (i3, i5, i7) processors

The string before the VBROADCASTSS refers to the encoding scheme of the instruction. Each of these string packs details such as alignment information, scale factor and other information relevant for assembly programmers. For the brevity of the report, we encourage the interested reader to find more details about the encoding scheme from the Intel Instruction manuals [Set12]. The relevant part of the notation for us is the following,

```
VBROADCASTSS xmm1, m32
     ^        ^     ^
     |        |     |
   operator   |  source
         destination
```

6.3.2 Adding vector primops and primtypes

We switch back our context now, to talk about the changes necessary in the GHC source tree. The vector attributes in the file `compiler/prelude/primops.txt.pp` is defined as a list of 3-tuples, which are of the form,

$< ELEM\_TYPE, SCALAR\_TYPE, LENGTH >$

ELEM_TYPE : the type of the elements in the vector eg : Int8

SCALAR_TYPE : the scalar type used to inject to/project from vector element eg: INT8

LENGTH : width of the vector

ELEM_TYPE and *SCALAR_TYPE* were not kept the same because to operate using a scalar value on a vector whose elements are of type `Int8`, we use `Int#`. This is part of the initial design of vectors as proposed by Mainland et al, however after the changes made in Section 6.4 they effectively become the same.

For our discussion we take the example of the `Float` type which is backed by the `Float#` primtype. We have added the support for working with XMM registers which are 128-bits wide. So each XMM register can hold four 32-bit floats. So, the 3-tuple we are concerned about is $< Float, Float\#, 4 >$.

The `utils/genprimcode` parser, parses the vector and emits the following three types

- FloatX4#
- Float#
- (# Float#, Float#, Float#, Float# #)

Of the three types generated the first type is the underlying vector type which is a 128-bit wide XMM register capable of holding four *raw floats*. As these are strict unlifted types, they refer to the actual vector value or register rather than a pointer, pointing to a region of memory. A 128-bit XMM vector register holds strict unlifted scalars and not their lifted version.

The second element plainly indicates the kind of scalar element that this vector shall hold. A point to be noted is that all of the vector types are strongly typed and the type annotations remain throughout the compilation pipeline until the STG representation. Though STG onward, the virtual STG registers take the place of the data types but the type annotations remain throughout the end of the compilation pipeline. This strong typing of intermediate representation throughout a complex pipeline of changes, makes debugging and correctness substantially easier.

The third element is a tuple type for the projection and injection of the scalars to the vectors. Specifically it is an *unboxed tuple*. An unboxed tuple, represented by $(\#e_1,\ e_2,\ ..e_n\#)$, is generally used for functions which needs to return multiple values. Unboxed tuples are different from ordinary tuples in Haskell. An ordinary tuple(from `Data.Tuple`) is syntactic sugar for,

```haskell
data Tuple2 a b   = Tuple2 a b
data Tuple3 a b c = Tuple3 a b c
```

On the other hand an unboxed tuple does not have any specific runtime representation in the heap. The values inside an unboxed tuple are either stored on registers or they spill to the stack. They are never allocated on the heap. This was a surprising discovery that we stumbled upon, which is not documented anywhere in GHC.

GHC Remark

Unboxed tuples are a compile-time construct! They do not have any runtime representation in the heap.

However, to work with broadcast operation of AVX family we don't need to deal with unboxed tuples currently. So the front end of the broadcast primop that we add to `compiler/prelude/primops.txt.pp` looks like this:

```
primop VecBroadcastOp "broadcast#" GenPrimOp
   SCALAR -> VECTOR
   { Broadcast a scalar to all elements of a vector. }
   with vector = ALL_VECTOR_TYPES
```

The line `with vector = ALL_VECTOR_TYPES` refers to the list of 3-tuples that we explained before. We have a separate parser to generate these function because there are a large number of data types and corresponding vector data types that we have to map between. Upon parsing the `utils/genprimopcode` code generator, generates the following function among others,

```
broadcastFloatX4# :: Float# -> FloatX4#
```

This function gets generated as part of the `GHC.Prim` virtual module, which is re exported via the `GHC.Exts` module which we shall use in all of our wrapper libraries as well as tests. Additionally the parser also generates an additional datatype declaration file[6] while bootstrapping the compiler which declares the type:

```
data PrimOp
  = .....
  | VecBroadcastOp
  | .... -- other vector primops
```

[6]the datatype declaration file is created by the stage 0 compiler. Refer the GHC remark for more details on the stages of bootstrapping

GHC Remark

The bootstrapping process of the GHC compiler happens in three stages.

- Stage 0 is the GHC you have installed. The ”GHC you have installed” is also called ”the bootstrap compiler”.
- Stage 1 is the first GHC we build, using stage 0. Stage 1 is then used to build the packages.
- Stage 2 is the second GHC we build, using stage 1. This is the one we normally install when you say make install.
- Stage 3 is optional, but is sometimes built to test stage 2.

6.3.3 Changes in Cmm and the STG→CMM bridge

The changes in the above section, would expose the function `broadcastFloatX4#` in the `GHC.Prim` module. However let us assume in a sample program we call this function we encounter the following:

```
-- assume this is defined inside a well typed Haskell 2010 function
brodcastFloatX4# 1.5#

-- compiling the above we get
"The native code generator does not support vector
 instructions. Please use -fllvm."
```

The above error message is emitted from the depths of the native code generator. We initially need to make changes to some very fundamental Cmm data types. Starting with the `compiler/cmm/CmmMachOp.hs` file which defines all the essential machine operations, which we can map a GHC primop to. There are primarily two crucial machine operation types. `MachOp` represents machines operations like addition, substraction, multiplication which can be reasonably delegated to the native code generator, whereas `CallishMachOp` type represents certain specialised operations like barriers, `memcpy`, `memset` etc, which needs to be implemented as a foreign call.

We saw in Section 6.3.1 that there is a direct broadcast operation supplied by x86 instruction set. So we make our changes in the `MachOp` type.

```
data MachOp =
  ........ -- a number of machops constructors possible in x86
  | MO_VF_Broadcast Length Width   -- Broadcast a scalar into a vector
  .....
  deriving(Eq, Show)

type Length = Int -- no of elements of the vector

-- width of each individual scalar
data Width =
          W8
        | W16 | W32 | W64
        | W80      -- Extended double-precision float,
                   -- used in x86 native codegen only.
        | W128
        | W256
        | W512
        deriving (Eq, Ord, Show)
```

The format of the data constructor is $MO\_VF\_Broadcast\ 4\ W32$, where the `Length` is the total number of elements in the vector and the `Width` type is the width of each scalar element that we are broadcasting. With the addition of the broadcast `MachOp` type we shift our attention to the STG → Cmm bridge, which converts an STG expression to Cmm. The most important function is the `emitPrimOp` function in the `compiler/cmm/StgCmmPrim.hs` file.

```
emitPrimOp :: DynFlags
           -> [LocalReg]        -- where to put the results
           -> PrimOp            -- the op
           -> [CmmExpr]         -- arguments
           -> FCode ()
```

The `FCode` is a reader, writer and state monad that is plumbed through the STG→Cmm code generator. This monad takes care of the local bindings in scope, the continuation to return to and other similar information before emitting a structure like a control flow graph. We don't need to modify anything in this plumbing, however we had to extend the `CmmExpr` type to add support for *vector literals*.

```
data CmmExpr
  = CmmLit CmmLit                 -- Literal
  | CmmLoad !CmmExpr !CmmType     -- Read memory location
  | CmmReg !CmmReg                -- Contents of register
  | CmmMachOp MachOp [CmmExpr]    -- Machine operation (+, -, *, etc.)
  | CmmStackSlot Area {-# UNPACK #-} !Int
                                  -- addressing expression of a stack slot
                                  -- See Note [CmmStackSlot aliasing]
  | CmmRegOff !CmmReg Int


data CmmLit =
  ....
  | CmmVec [CmmLit]                               -- Vector literal
```

We specifically need vector literals for the broadcast operation. There is no machine level concept for vector literals. Vectors are plainly a combination of multiple scalars residing in a register. However the Cmm expression that we want to emit for our broadcast operation looks like below:

```
_c1PG::Fx4V128 = <0.0 :: W32, 0.0 :: W32, 0.0 :: W32,
                  0.0 :: W32>;   // CmmAssign
_c1PH::Fx4V128 = %MO_VF_Broadcast_4_W32(_c1PG::Fx4V128,
                                        1.5 :: W32);   // CmmAssign
```

Listing 1: The Cmm for broadcast

The first statement represents a vector literal. However a natural question arises, what is the necessity of the first expression. Why can't we have a simpler Cmm expression like:

```
_c1PH::Fx4V128 = %MO_VF_Broadcast_4_W32(1.5 :: W32);   // CmmAssign
```

The reason we cannot do the above, is because of the semantics of the Cmm language. Although Cmm resembles a high level language in its features and syntax, we should not forget that it is primarily a low level assembly language equivalent which operates with registers. Each of the operands that we see are registers (Cmm has an infinite supply of *typed* registers), and it is not possible to allocate a broadcast operation to a register without initialising it. Although we would be able to generate a sane assembly instruction from the second expression as well, the register allocator of Cmm has a *liveness analysis* pass which would not allow the expression to pass. We shall talk about liveness analysis of registers briefly in the next section.

The changes described in this section are sufficient to emit the Cmm expression in Listing 1 from the `emitPrimOp` function. Next we shall see how we parse the Cmm expression in the native code generator to actually generate x86 assembly instructions, after we finish discussing about *liveness analysis*.

6.3.4 Liveness analysis

When working with intermediate representations in compilers, the assumption is always to take a large number of temporary variables and store all of them in registers. In reality, most RISC machines have a limited number of registers. We discussed in Section 6.3.1 that the Intel Sandy Bridge architecture has only sixteen vector registers. Similarly the number of general purpose registers are also limited and the compiler tries to optimise its usage.

The `cmmLocalLiveness` function in the source file `compiler/cmm/CmmLive.hs`, runs a liveness analysis to check which registers are live and does not allow us to enter a control flow block without initialising a register(initialisation is the start of the register lifetime). The formal definition of a live variable is,

We say that variable v is live at a statement n if there is a path in the CFG from this statement to a statement m such that $v \in use[m]$ and for each $n \leq k < m$: $v \notin def[k]$. That is, there is no assignment to v in the path from n to m.

The understanding of liveness is essential because liveness constrains the definition of a number of Cmm expressions (not very much unlike broadcast) as well as the optimisation passes running in Cmm depends on it.

6.3.5 Changes in the NCG

The GHC native code generator supports SPARC and PPC instruction sets apart from x86. We choose to concentrate our efforts on the x86 backend. Almost the majority of the code that we have written, is distributed in the bowels of the code generator. There are a number of helper functions as well as new data types that we have added for the vector support. Without going into the nitty gritties we mention the major types where we need to make the changes.

The `Instr` type in the file `compiler/nativeGen/X86/Instr.hs` defines the data constructors for all the x86 instructions. So our changes there looks like this,

```haskell
data Instr =
  .....
  | VBROADCAST  Format AddrMode Reg
```

Listing 2: Changes in the `Instr` data type

where the `Format` type decides the suffix to be attached with the instruction. In x86 `ss` suffix is used for single precision float, while `sd` is used for double precision floats. Similarly there are suffixes for each format like 8-bit integers and other types. Next, the `Reg` type is defined like this,

```haskell
data Reg
        = RegVirtual !VirtualReg
        | RegReal    !RealReg
        deriving (Eq, Ord)

data RealReg
        = RealRegSingle {-# UNPACK #-} !RegNo
        | RealRegPair   {-# UNPACK #-} !RegNo {-# UNPACK #-} !RegNo
        deriving (Eq, Show, Ord)

```

```haskell
data VirtualReg
        = VirtualRegI  {-# UNPACK #-} !Unique
        | VirtualRegHi {-# UNPACK #-} !Unique  -- High part of 2-word register
        | VirtualRegF  {-# UNPACK #-} !Unique
        | VirtualRegD  {-# UNPACK #-} !Unique
        | VirtualRegSSE {-# UNPACK #-} !Unique
        deriving (Eq, Show)
```

The `RealReg` type refers to the real machine registers like `%rax`, whereas the `VirtualReg` type allocates virtual registers which are eventually mapped to the `RealReg` or spilled to the stack. The final data constructor `VirtualRegSSE`[7] was added to indicated the vector register and some changes were made in the linear register allocator algorithm to allocate XMM registers. Finally the `AddrMode` type in Listing 2 refers to a memory address or immediate value.

Apart from the `compiler/nativeGen/X86/Instr.hs` file, the pretty printer is modified, which actually prints (or emits) the assembly instruction. The pretty printer uses the `Format` type to understand the type of the register and other information from the register allocator to print the appropriate register. The bulk of the code actually parsing the Cmm expressions and translating them into assembly resides in the file `compiler/nativeGen/X86/CodeGen.hs` of the source tree. There are also a number of other non-trivial code changes which involve the `data` and `newtype` declaration of vector types. Interested reader are encouraged to study the source code[8] to understand the finer engineering details.

After finally modifying the pretty printer we can tackle the logic of the assembly instructions. Corresponding to the Cmm expression in Listing 1, the first statement of initialising a vector with all zeroes can be achieved by *xor*ing the register with itself. And after we have initialised the vector register we can simply emit the broadcast operation by loading the literal as a memory address and broadcasting it in the zeroed out register.

So, in this way after finally emitting the assembly, let us look at the metamorphosis that the original Haskell operation undergoes in each step.

[7]the name includes SSE, the same data constructor is used for AVX registers as well
[8]source code available at `https://github.com/Abhiroop/ghc-1/tree/wip/simd-ncg-support`

```
broadcastFloatX4# 1.5#
```

```
[LclId] =
    [] \u []
        case broadcastFloatX4# [1.5#] of sat_s1M1 {
          __DEFAULT -> ...}
```

```
_c1PG::Fx4V128 = <0.0 :: W32, 0.0 :: W32, 0.0 :: W32,
                  0.0 :: W32>;   // CmmAssign
_c1PH::Fx4V128 = %MO_VF_Broadcast_4_W32(_c1PG::Fx4V128,
                                        1.5 :: W32);   // CmmAssign
```

```
xorps %xmm0,%xmm0
movss _n1PZ(%rip),%xmm1
vmovups %xmm0,%xmm0
movups %xmm1,(%rsp)
vbroadcastss (%rsp),%xmm0
.
.; other instructions elided
.
_n1PZ:
        .byte       0
        .byte       0
        .byte       192
        .byte       63
```

Listing 3: Broadcast in Haskell 2010, STG, Cmm and x86 assembly respectively

6.3.6 Sinking pass

We mentioned while discussing Figure 6.1 that the sinking pass, is an important compiler optimisation which affects the form of multiple Cmm expressions. This is particularly observable in the packing operation of vectors. If we take the example of packing four floats into a vector type,

```
packFloatX4# :: (# Float#, Float#, Float#, Float# #) -> FloatX4#

-- eg:
packFloatX4# (# 4.5#,7.8#,2.3#,6.5# #)
```

The corresponding Cmm expression that we emit for the packing operation is given below,

```
1  _c10m::Fx4V128 = <0.0 :: W32, 0.0 :: W32, 0.0 :: W32,
2                               0.0 :: W32>;   // CmmAssign
3  _c10n::Fx4V128 = %MO_VF_Insert_4_W32(_c10m::Fx4V128, 4.5 :: W32,
4                                                 0 :: W32);   // CmmAssign
5  _c10o::Fx4V128 = %MO_VF_Insert_4_W32(_c10n::Fx4V128, 7.8 :: W32,
6                                                 16 :: W32);   // CmmAssign
7  _c10p::Fx4V128 = %MO_VF_Insert_4_W32(_c10o::Fx4V128, 2.3 :: W32,
8                                                 32 :: W32);   // CmmAssign
9  _c10q::Fx4V128 = %MO_VF_Insert_4_W32(_c10p::Fx4V128, 6.5 :: W32,
10                                                48 :: W32);   // CmmAssign
```

This appears like a needlessly awkward representation. On the left hand side of line 1 we have a zeroed out vector register, each of the four insert operations take the index of the vector register where the element needs to be packed. A much more natural representation would be to take the original register _c10m and pack these elements in the respective indices of the vector. However we load the first element to the first 32-bits of _c10m and move the value to _c10n and continue to thread down each register with a pack followed by a move. This representation is owing to the variable sinking pass which is turned off by default in GHC. When the optimisation is switched on the Cmm becomes,

```
_s3yj::Fx4V128 = %MO_VF_Insert_4_W32
                    (%MO_VF_Insert_4_W32
                        (%MO_VF_Insert_4_W32
                            (%MO_VF_Insert_4_W32(<0.0 :: W32, 0.0 :: W32,
                                                  0.0 :: W32, 0.0 :: W32>,
                                                 4.5 :: W32,
                                                 0 :: W32),
                             7.8 :: W32, 16 :: W32),
                        2.3 :: W32, 32 :: W32),
                    6.5 :: W32, 48 :: W32);   // CmmAssign
```

In the above representation all the intermediate variables have been *sunk* and all the function calls inlined to avoid wasteful register spilling. This was an example, there are a number of other instructions including all the arithmetic operations, whose Cmm representations have been dictated by the variable sinking optimisation pass.

6.3.7 Reflection

In this entire section we have taken a sample vector instruction of `broadcast` and shown how to implement it across the full code generation pipeline. We have intentionally taken a simpler instruction as an example. For a number of operations like shuffling there are no direct corresponding x86 instructions and we have to design an optimal set of assembly instructions by combining other operations. The various optimisation passes like variable sinking, liveness analysis etc also contribute to the complexity of the implementation. Interested readers are encouraged to study the modified GHC source tree[9] for further details on the implementation.

6.4 Supporting smaller integer widths in GHC

A number of modern day software, tend to operate on large datasets containing data of smaller width. For instance, audio applications extensively use 16-bit data. Similarly the graphics industry operates on 8-bit data frequently. With the rise of machine learning algorithms, 8-bit and 16-bit floating point numbers have been introduced specifically for deep neural network algorithms [Gup+15]. When operating on microprocessors with 32-bits or 64-bits word size, the upper bits of the registers remains unused and they continue to consume unnecessary power.

SIMD registers would be particularly beneficial for these applications, as most of these applications tend to be parallel in nature. A 128-bit Xmm register can fit sixteen 8-bit integers and theoretically provide 16x speedups. However, as discussed in Figure 6.4 GHC defines all signed and unsigned variants of integers as wrappers over the same machine dependent primop.

```
data {-# CTYPE "HsInt8" #-} Int8   = I8# Int#

data {-# CTYPE "HsInt16" #-} Int16 = I16# Int#
```

[9]source code available at `https://github.com/Abhiroop/ghc-1/tree/wip/simd-ncg-support`

```

data {-# CTYPE "HsInt32" #-} Int32 = I32# Int#

data {-# CTYPE "HsInt64" #-} Int64 = I64# Int#
```

Similarly for the unsigned `Word` types, they are all represented underneath by the same machine dependent `Word#` primop. Both `Int#` and `Word#` have width equal to the word size which is 64-bits for a 64-bit machine. A major part of this dissertation was spent in enabling the support for `Int8#, Word8#, Int16#, Word16#, Int32# and Word32#`. This change makes each of the wrapper data type definitions much more precise, and allows us to work with smaller width data types in SIMD.

The process of adding vector instruction support for integers is not very different from what we discussed in the previous section. The crucial part is the new integer primtypes and their respective primops which we shall discuss briefly in the following sections.

6.4.1 Adding the new runtime representations

We want to add a new runtime representation for each of the small width primtypes. However, to add a new runtime representation we need to understand the kinding mechanism of GHC.

With the introduction of levity polymorphism [EP17] in GHC 8.0, the fundamental kinding mechanism of GHC was modified. Prior to version 8.0, GHC supported a subkinding[10] mechanism with the kind `OpenKind` being the parent of the kinds "*" and "#"

```
    OpenKind
    /       \
   /         \
  *           #

-- "This is a gross hack" - Simon Peyton Jones
```

[10]subkinding is a play on the word subtyping which implies a relation between types via a notion of substitutability

This subkinding mechanism was the cause of a number of spurious and hidden bugs. There was no notion of separate runtime representation for the lifted and unlifted types. GHC 8.0 repaired that by defining a type `Levity` which was defined as $data\ Levity = Lifted \mid Unlifted$ and defined the "*" and "#" kinds using the type `Levity`. Although a sound design was proposed in the original paper [EP17], the final implementation in GHC differed substantially from the paper. The final version of runtime representation which currently exists in the GHC mainline is not documented in any publication, but the source code looks like this,

```
type * = TYPE 'LiftedRep

type # = TYPE 'UnliftedRep

-- TYPE :: RuntimeRep -> *

data RuntimeRep = LiftedRep
                | UnliftedRep
                | IntRep
                | WordRep
                | ...
```

While most of the above makes sense, the tricky portion is understanding the magical `TYPE` which itself maps a `RuntimeRep` to the kind "*". For more details on this, the interested reader is directed to the paper by Eisenberg et al [EP17].

We are ourselves primarily concerned with the `RuntimeRep` type and extending it. Currently something like `Int#` is expressed as

```
 Int# :: TYPE 'IntRep
```

We hope to introduce runtime representations for all the sub-word[11] size types as well as a runtime representation for vectors. Hence we modify the `RuntimeRep` with the new data constructors for vectors as well as the sub-word types. (Note: the `RuntimeRep`

[11] we use sub-word to denote data types whose size is less than 64 bits

for just the vector type i.e `VecRep` was added by Richard Eisenberg and not the author of this report)

```
data RuntimeRep
  =  LiftedRep
  | UnliftedRep
  | VecRep VecCount VecElem   -- ^ a SIMD vector type
  | Int8Rep         -- ^ signed,  8-bit value
  | Int16Rep        -- ^ signed, 16-bit value
  | Int32Rep        -- ^ signed, 32-bit value
  | Int64Rep        -- ^ signed, 64-bit value (on 32-bit only)
  | Word8Rep        -- ^ unsigned,  8-bit value
  | Word16Rep       -- ^ unsigned, 16-bit value
  | Word32Rep       -- ^ unsigned, 32-bit value
  | Word64Rep       -- ^ unsigned, 64-bit value (on 32-bit only)
```

6.4.2 Changes in the typechecker

In the previous section we added the `RuntimeRep` data constructors for all the sub-word sized data types. Now we show how to allocate a type to each of these data types. A type in the typechecker is defined as an algebraic data type `Type`

```
data Type

  = .....
  | TyConApp
        TyCon
        [KindOrType]
  ....

  deriving Data.Data
```

This type is one of the central tenets of the GHC typechecker. The most important constructor for us is the `TyConApp` constructor which allows us to define both `data` and `newtype` data constructors. We have to represent our new `RuntimeRep`s as type constructors.

GHC has multiple categories of entities:

- Wired-in things - GHC knows everything about these
- Known-key things - GHC knows the name, including the Unique, but not the definition
- Orig RdrName things - GHC knows which module it's defined in

The `Int8#` and friends are "wired in things" and they are wired by allocating unique numbers to them and creating a data constructor using those unique numbers. Once we have the data constructors we can create a `Type` by composing these two helper functions which are already available in GHC.

```
promoteDataCon :: DataCon -> TyCon

mkTyConTy :: TyCon -> Type
```

There are some other minor engineering details around wiring in our subword type constructors in the type checker which are available in our second modified branch of GHC[12].

6.4.3 Extending and narrowing registers

We discussed about adding a new runtime representation and the changes in the typechecker. The other changes that we do are centred around the code generator. Most of the ground work on adding the primops like addition, subtraction, etc, around each of the sub-word size primtypes is quite similar to what we have discussed in Section 6.3.

The novel changes in the code generator involves the *extending* and *narrowing* of the Cmm registers for supporting the new primtypes . Consider passing a small (< word width) primitive like `Int8#` to a function through a register. It is actually non-trivial to do this without extending/narrowing.

[12]`https://github.com/Abhiroop/ghc-1/tree/int8`

- Global registers are considered to have native word width (i.e., 64-bits on x86-64) at the Cmm level, so the Cmm linter wouldn't allow us to allocate an 8-bit or 16-bit register to it
- The above holds true for the LLVM IR as well
- Lowering gets harder since on x86-32 not every register exposes its lower 8 bits (e.g., for %eax general purpose register we can use %al, but there isn't a corresponding 8-bit register for %edi). So we would either need to extend/narrow anyway, or complicate the calling convention.

So the general solution chosen was to always extend every parameter smaller than native word width in the in the Cmm and then truncate it back to the expected width before code generation.

Note that we do this entire `extending-narrowing` of registers in Cmm by adding a new `MO_XX_Conv` data constructor to the Cmm machine operation type (`MachOp`). This avoids requiring zero-/sign-extending - it's up to a backend to handle this in a most efficient way (e.g., a simple register move in the native code generator). We have additionally added primops for manually extending and narrowing the registers in our source code,

```
extendInt8# :: Int8# -> Int#
narrowInt8# :: Int# -> Int8#
```

The similar was added for the other sub-word size data types as well.

6.5 Example: Polynomial evaluation using GHC's vector types

In this section we take a look at an example which uses the low level vector types that we have added, to parallelise a sequential program. The specific example we choose is polynomial evaluation. Polynomials exhibits parallelism in evaluating each of its individual terms. Consider the following polynomial:

$$7x^4 + 8x^3 + 3x^2 + 2x + 1$$

This polynomial can be visualised as:

```
  7 * x * x * x * x
+ 8 * x * x * x
+ 3 * x * x
+ 2 * x
+ 1
```

Given the above structure of polynomial evaluation we can pack the first two columns into vectors and save the cost of three multiplication operations. So vectorisation of the above would look like the following:

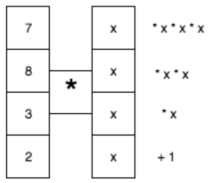


Figure 6.7: Vectorised polynomial evluation

While this saves us the cost of three multiplication operations, the cost of packing and unpacking are far higher then the x86 multiplication instruction which happens in the order of nanoseconds. To compensate the cost and see an actual benefit we need to evaluate polynomial of the order of 512 and higher. The actual code for evaluation using the vector types is given in Listing 4:

The program looks very different from a general declarative Haskell program. It accepts the value for which the polynomial has to be evaluated as a `Float` type. The coefficients of the polynomial are represented as a list . So a polynomial like $7x^4 + 8x^3 + 3x^2 + 2x + 1$ would be represented as `[7,8,3,2,1]`. In the cases of a missing term in the polynomial, we specify `0` in the list. So, $x^5 + 1$ is represented as `[1,0,0,0,0,1]`. Our contributions to the API should be visible in the form of the following functions:

- `packFloatX4#`
- `broadcastFloatX4#`
- `unpackFloatX4#`
- `timesFloatX4#`

In the program we have defined a separate helper function `splitEvery` for splitting the list into sublists of given size. The sublist size would be equal to the width of the

```haskell
evalPolyVec :: Float -> [Float] -> Float
evalPolyVec value coeffs = go (splitEvery 4 coeffs) (length coeffs)
  where
    go [[x]] _    = x
    go (x:xs) len =
      let [(F# a), (F# b), (F# c), (F# d)] = x
          (F# val)                         = value
          packed_coeff                     = packFloatX4# (# a, b, c, d #)
          vec_val                          = broadcastFloatX4# val
          step_length                      = len - sizeOfVec
      in (go' packed_coeff vec_val step_length) + (go xs step_length)
      where
        go' pc _ 0 =
          let (# a, b, c, d #) = unpackFloatX4# pc
          in ((F# a) * value ^ 3) +
             ((F# b) * value ^ 2) +
             ((F# c) * value) +
             (F# d)
        go' pc v l =
          let t = (timesFloatX4# pc v)
          in go' t v (l - 1)

splitEvery _ [] = []
splitEvery n list = first : (splitEvery n rest)
  where
    (first, rest) = splitAt n list
```

Listing 4: Vectorised polynomial evaluation.

vector register that we want to pack the scalars into. The rest of the logic, though slightly verbose, is quite simple. We are effectively iterating over the list of polynomial coefficients and packing and multiplying each component. In the final iteration we complete the evaluation by using the scalar operators.

While Listing 4 parallelises our program using SIMD evaluation, the parallelism comes at the cost of compromising the declarative nature of Haskell. And our primary goal is to aim for declarative parallel programming. The *lift-vector* library introduced in the following chapter will attempt to mitigate this problem to an acceptable extent. In Chapter 10 on evaluations we will present the same program written using the *lift-vector* library to make it more high level and declarative.

6.6 Contributions and Reflection

Throughout this chapter we discussed the low level details of how we made changes in the code generation pipeline. The chapter began with Figure 6.1 demonstrating the parts of the pipeline where our changes reside. Over the course of the past few sections we have tried to give a brief overview about the new datatypes that we introduced in the pipeline and the overall engineering efforts involved in adding vector instruction support to GHC. In the last section we have also presented a brief example of polynomial evaluation using the new vector types of GHC.

For the code generator, we took a sample vector operation and showed how we are currently generating its corresponding x86 assembly. Most of the other operations would involve similar changes in the code generator. All of our changes have been heavily code reviewed and all the changes are in the pipeline for the GHC version 8.9 release. Here is a list of what we can expect in GHC 8.9 and future releases,

- Support for `Int8#, Word8#, Int16#, Word16#, Int32# and Word32#` in GHC 8.9. The proposal[13] for small primitive types had been accepted by the GHC steering committee a year ago. Simon Marlow, the chief architect of nearly the entire backend of GHC has stated the following about this change,

  > This needs to happen, the only reason we didn't do this earlier was that we sneakily avoided needing real sized integral types by adding the narrow primops so that we could implement all of Data.Int and Data.Word using Int# and Word#. Once we have real sized types, we could re-implement Data.Int and Data.Word using sized primops and kill the narrow primops. (I realise this isn't part of this proposal, but it's a nice cleanup).

  This project has provided a firsthand implementation of the entire GHC proposal which was initially divided into three parts over a period of a year. This will be one of the first contributions to be merged into GHC 8.9

- Support for `FloatX4#` and `DoubleX2#`. These two primtypes have been implemented in their entirety and the groundwork for supporting all the other vector types has been laid in the parser. These two primops are in line to be merged with GHC 8.9 or latest by GHC 9.0

[13]`https://github.com/ghc-proposals/ghc-proposals/pull/74`

- NCG backend for vectorisation. The entire native code generator backend changes supporting all arithmetic operations as well as the packing and unpacking operations have been heavily reviewed and scrutinised and shall hopefully be merged to the GHC mainline version 9.0

This brings us to the end of most of our GHC related efforts. Primarily, we have designed the substrate on top of which high level libraries for vectorisation can be built. Also the control flow graph of GHC is now able to understand Cmm vector machine operations which opens the door for us to implement *automatic* vectorisation. Over the next few sections we address, how we can use the vectorised NCG backend of GHC to build high level libraries and frameworks for parallel programming.

7. LIFT-VECTOR: A LIBRARY FOR VECTORISATION IN HASKELL

In this chapter we talk about the *lift-vector*[14] library which provides higher order, *polymorphic* SIMD functions for vectorisation. The focus of the library was to devise a *polymorphic* front end API for vector programming.

7.1 An abstraction for vectors

The vector APIs that we have exposed till now, are entirely composed of unlifted types. As of the last chapter, to access the SIMD capabilities of GHC one needs to have a detailed understanding of the compiler internals as well as the cost of each vector operation. Most general Haskell programmers are not aware or educated about unlifted types and they generally exist in the bowels of the compiler. Also exposing someone to the details of the low-level instructions from the backend would effectively render a highly declarative language like Haskell to decay into the *leaky* abstraction bound imperative language territory.

One of the best ways to tackle this problem is to make the compiler intelligent enough and allow it to *auto-vectorise*. However automatic vectorisation is a full blown research question of its own and it would take a much more thorough and time consuming approach to integrate it . Instead what we attempt is to build a library, which though not as intelligent as auto-vectorisation would at-least provide library authors a starting point to work with GHC without delving into the details of the compiler.

7.1.1 Typeclasses and type-families to the rescue

A strong focus is laid on *polymorphism*, because the multitude of possible vector operations and the various ways of packing and unpacking them is hard to remember and is an unnecessary, low-level detail. The primary source of ad hoc polymorphism in Haskell is through type classes. We have briefly discussed about type classes in Section 3.

We take the example of the `packing` operation on vectors to abstract over. Currently we have the following,

```
packFloatX4#  :: (# Float#,  Float#, Float#, Float# #) -> FloatX4#
```

[14]`https://github.com/Abhiroop/lift-vector`

```
packDoubleX2# :: (# Double#, Double# #)                -> DoubleX2#
```

To start with, we can easily wrap over the `Float#` types as well as create a general wrapper around `FloatX4#`. However, do note it requires a non-trivial amount of changes in the NCG to support datatype wrappers over the vector primtypes. All of that work has been included in our modified branch of GHC. So, we can now declare,

```
data FloatX4  = FX4# FloatX4#

packFloatX4 :: (Float, Float, Float, Float) -> FloatX4
packFloatX4 (F# x1, F# x2, F# x3, F# x4) = FX4# (packFloatX4# (# x1, x2, x3, x4 #))
```

Now that, we have been able to hide the unlifted types, we can imagine for the various widths of the register as well as the multiple datatype like `Int, Float, Double, etc`, we will have a large number of packing functions like these:

```
packFloatX4 ::  (Float, Float, Float, Float) -> FloatX4
packDoubleX2 :: (Double, Double)             -> DoubleX2
packInt8X16  :: (Int8, Int8, Int8, Int8, Int8, Int8, Int8, Int8,
                 Int8, Int8, Int8, Int8, Int8, Int8, Int8, Int8) -> Int8X16
packInt16X8  :: (Int16, Int16, Int16, Int16,
                 Int16, Int16, Int16, Int16) -> Int16X8
packInt32X4  :: (Int32, Int32, Int32, Int32) -> Int32X4
packInt64X2  :: (Int64, Int64) -> Int64X2
```

Listing 5: Changes in the `Instr` data type

And the above are just for XMM registers of 128-bit width. There are additional 256-bit wide YMM registers as well as 512-bit wide ZMM registers, which implies six times three which is eighteen functions for just packing. Now imagine there are additionally unpacking, broadcasting, arithmetic, logic and other operations which could potentially result in over two hundred functions. So if we observe Listing 5, it has a repetitive pattern where it takes different types of tuples(either 2-tuple, 4-tupls, 8-tuples or 16-tuples) and maps to the respective vector types. We can attempt to be polymorphic on the vector type and define something like this:

```
class SIMDVector v where
```

```
  packVector :: (???) -> v
```

The issue is the first argument of `packVector` is itself *dependent* on the final type that it is mapping to. For a `FloatX4` it is a 4-tuple of `Float`s, whereas for a `Int8X16` it is a 16-tuple of `Int8`s. The type is *dependent* on a type level constraint and to emulate this class of dependent types, GHC supports associated types or type families [Cha+05]. Again we provided a brief introduction to associated types in Section 3, though an interested reader is encouraged to read the original paper by Chakravarty et al for more details.

Associated types hence allow us to modify the previous snippet of the `SIMDVector` typeclass to:

```
class SIMDVector v where
    type Elem v
    type ElemTuple v

  packVector :: ElemTuple v -> v
```

This allows us to define an instance for this typeclass while *depending* on the shape of the tuples as well as their content. So we can write,

```
instance SIMDVector FloatX4 where
  type Elem      FloatX4 = Float
  type ElemTuple FloatX4 = (Float, Float, Float, Float)
  packVector = packFloatX4
```

We can use this technique to abstract over all the basic operations like packing, unpacking, broadcasting, mapping, zipping, folding etc. This typeclass gives us the core polymorphic operations for construction and deconstruction of vectors. At the same time we choose to export only the polymorphic variants of our API while hiding the actual implementations (like `packFloatX4` etc).

Note when we mention *mapping, zipping or folding*, we are talking about simply folding that vector type. So something like `FloatX4` can be folded to a single `Float` using a suitable folding function. However what about vectorisation over more powerful data structures like lists and arrays? We address this in the next section.

7.2 Vectorised data structures

There are primarily two classes of data structures that we hope to divide this library into,

- Immutable and purely functional data structures [Oka99]
- Mutable array based structures

Among the purely functional data structures, within the current time frame we could only implement vectorised version of lists[15]. However we do hope to produce vectorised version of trees and other purely functional structures which we detail in the section on future work. We also discuss the about our implementation for vectorised arrays in this chapter. However we first look at a general typeclass that we have designed to operate on *all* vectorised data structures.

7.2.1 A typeclass for generic operations

We have attempted to devise a common set of operations which are applicable on any vectorised data structure or container. It is accomplished by parameterizing our typeclass over the container of type *t*. We expose general higher order functions which are commonly applied on structures likes lists and arrays. The novelty lies in *how* each of these functions operate on the containers. They process the elements inside the data structure in chunks of size four or eight or sixteen and applies a function on each of these chunks using vector operations. Let us have a look at this typeclass.

```haskell
class (Num a, Num b) =>
      ArithVector t a b
  where
  -- | The folding function should be commutative
  fold   :: (a -> a -> a) -> (b -> b -> b) -> b -> t b -> b
  zipVec :: (a -> a -> a) -> (b -> b -> b) -> t b -> t b -> t b
  fmap   :: (a -> a)      -> (b -> b)      -> t b -> t b
```

Listing 6: The `ArithVector` typeclass

One of the first noticeable features of this type class is the fact that it is parameterized over three parameters rather than one. GHC supports the concept of multi-parameter typeclasses which allows us to define a relationship between types. Multi-parameter type

[15]which is quite ubiquitous in functional programming

classes were not part of Haskell 98 and currently they are available in GHC as a language extension. Multi param typeclasses along with functional dependencies [Jon00] are almost as expressive as type families. However, here we do not need to express any complex type level constraint. We are operating on a uniform *shape t* of our data structure and multi parameter type classes simply provides a way to be polymorphic over three parameters.

The parameter b is clearly the type of the elements that the container holds. So the question arises then, what is the purpose of the type parameter a?

The scalar dribble problem : One of the issues when vectorising operations on any container is the *scalar dribble* problem. The issue arises because of the width of each of the vector registers are either 128-bits or 256-bits or 512-bits. So imagine if we are packing 32-bit floats on a 128-bit wide Xmm register, we can manage to fit in four floats in one of the registers. Now imagine we are operating on a list which contains eighteen elements. We pack it in batches of four, so what happens to the last two elements? These last two elements cannot be packed into a vector and even if we pack them with certain garbage values in the upper 64-bits, the operation will be unnecessarily costlier owing to the overhead of packing and unpacking. The remaining two elements contribute to what is known as the *scalar dribble.* This name is because the final two elements will be operated using scalar operators. The same problem arises on 256-bit wide or 512-bit wide registers as well. It is demonstrated in Figure 7.1.

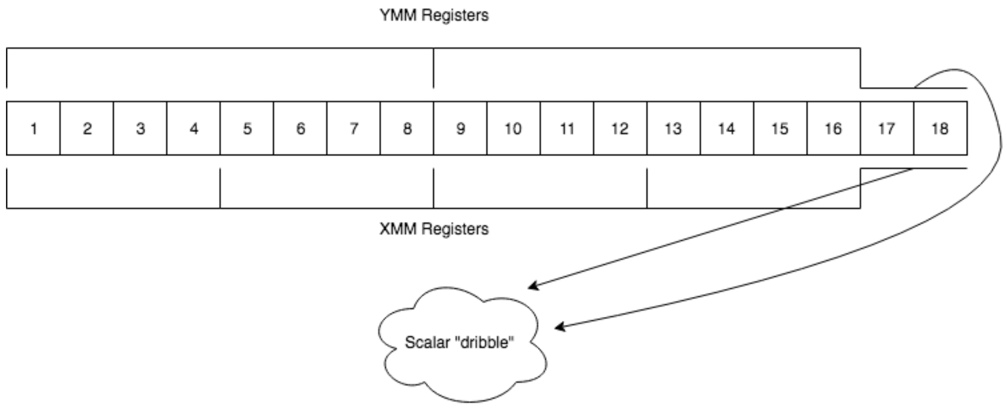


Figure 7.1: Demonstrating scalar ”dribble”

Owing to the *scalar dribble* problem, a consumer of the *lift-vector* library needs to provide two functions to the library. First the vector function and secondly its scalar analogue. So for instance, if we were adding all the elements in a vector list the consumer would provide the addition function for vectors as well as the addition function for scalars.

This is precisely the reason for us parameterizing over the additional type parameter a. In Listing 6 t is the container type, the second parameter a refer to the vector types like `FloatX4` while the third parameter b is for the corresponding scalar type `Float`.

Critique: While not the cleanest of designs, one of the pros of this typeclass is the amount of polymorphism this brings to the table. Not only does the consumer, need not know about the container type, they are also *considerably* oblivious to the vector type being used underneath. We say *considerably* because to satisfy the typechecker the consumer still needs to enter the corresponding vector type, however the operation itself like `+` or `*` is still overloaded. In general it is difficult for the compiler to always assess the domain specific advantage for a particular vector width type and instead of making a half-hearted guess, we give the power to choose the vector width to the library author.

The visible cons are a case of a *leaky* abstraction, where a general programmer is assumed to have knowledge of vector register width which is an extremely low level detail. However, in the current iteration of the project this is the best trade-off that we could choose. We hope to simplify the API in the future.

7.2.2 Vector Lists

The first class of containers that the `ArithVector` typeclass abstracts over is the vector list which is a plain wrapper around lists. It uses `newtype` to wrap around lists, which is simply a compile-time construct and doesn't add to any runtime overhead.

```haskell
newtype VecList a = VecList [a]
```

Currently the operations on vector lists are processed in chunks of size four (for `Float`s) and two (for `Double`s). None of the internal operations are exposed to the public API except the *to* and *from VecList* functions. All general operations on `VecLists` are carried out through the general combinators of *folding, mapping* and *zipping*. One of the most useful abstractions over plain Haskell lists are Monoids. So can we vectorise *monoidal* operations?

7.2.2.1 Why not a Semigroup or a Monoid?

Semigroups and monoids are an important class of abstraction which allows us to apply associative operations on various structures combining them to a single value. They are represented as:

```haskell
class Semigroup a where
  (<>) :: a -> a -> a

class (Semigroup a) => Monoid a where
  mempty  :: a
  mconcat :: [a] -> a
```

If we look at each of the functions, we can see that the constraint is on the *elements* of the list to be a monoid. Unfortunately most of the vector operations are on `Ints`, `Floats` and `Doubles` and none of them form a Monoid. To form a monoid we need to define an associative function combining the datatype. A general technique is to use a wrapper structure like `Sum a` to denote an associative operation like addition and use that as the *combining* operation for the monoid.

We should remember we are talking about the properties of the *container*. The container can be monoid irrespective of the properties of the elements. Like a list is a monoid. Similarly we can make a vector list a monoid as well. But almost all of our vector operations are arithmetic operations. So even if vector list were to be a monoid, there is nothing it would gain from being a monoid. It would simply be a wrapper over the list monoid.

On the other hand a list can be container for `String` or `IO` or a multitude of other types, each of which support the associative combining operation. Hence lists are monoids but we choose not to make vector lists to be a monoid.

7.2.3 Vector Arrays

The second class of data structures that we work with are vectorised arrays. Arrays are much more popular data structures for high performance computing. However, the in-place mutation API of arrays are quite opposed to the fundamental principles of immutable functional programming. Hence we provide a relatively functional API to work with arrays.

Not very much unlike vector lists, vector arrays are also wrappers around an existing Haskell data structure. Here we use unboxed *vector*s. The *vector*[16] package in Haskell is unrelated to SIMD vectors. It is simply an alternate name for list like structures which are backed by a mutable array underneath. The primary advantage of using a *vector* type is

[16]http://hackage.haskell.org/package/vector

the fact that it feeds of a powerful form of compiler optimisation known as Stream Fusion. This was discussed in some length in Section 5.1.2.

The *vector* package provides a number of data types to work with but we have chosen the variant of *unboxed vectors*. We have discussed about boxed and unboxed types of Haskell in considerable detail in Section 6.2.1. In the case of boxed and unboxed vectors the representation on heap would look like below.

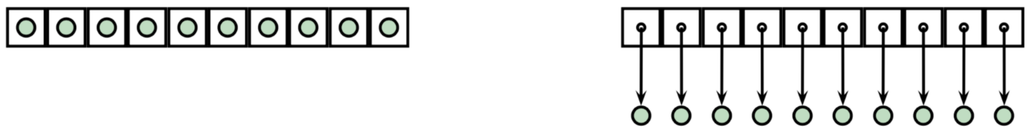


Figure 7.2: An unboxed and boxed vector respectively

The additional pointers in a boxed vector reduces the efficiency. Owing to this reason we choose to wrap over *unboxed vectors*. The representation of vector array that we use is given below:

```haskell
import qualified Data.Vector.Unboxed as U

data VecArray sh a = VecArray !sh (U.Vector a)
```

The powerful part of this representation is that we can represent two, three or any possible higher dimensional arrays using this. This work is not novel in any way and it is inspired from the research on the Repa library in Haskell [Kel+10], which introduced the concept of *shape polymorphism*. However we have chosen to simplify the API of Repa considerably. Repa uses a number of type family representations to choose between various possible vector types and has a huge number of dependencies which we have removed.

Repa internally uses unboxed vector for reads while it uses mutable vector, which is in turn backed by `MutableArray`, for writes. Unboxed vectors are great for `O(1)` reads but even a single write creates a new vector which takes linear time. To compensate for that mutable vectors provide `O(1)` writes by providing mutation inside the ST or IO Monad. Internally Repa does an `unsafeIO` operation to facilitate every single write.

To keep our API relatively simple we did not introduce any monadic structures there. We simply defer our writes by creating a linked list underneath for the various operations like *folding* and finally convert the list to an array. This happens at `O(n)` complexity and the entire write happens just once. Hence our simplification comes at the cost of efficiency. We shall see in chapter 10 on evaluations how the absence of in-place writes slows down the performance. For this chapter, our main aim is to demonstrate the declarative and polymorphic nature of our API.

The shape polymorphism part nonetheless comes as a nice add-on on top of our already existing polymorphic interface. Let us briefly discuss about how this relatively simple looking representation can capture all the possible dimensions of an array.

7.2.3.1 Shape Polymorphism

The basis of shape polymorphism, as defined by Keller et al [Kel+10], is the fact that the internal computer memory is essentially one dimensional and hence a similar one dimensional structure can be used to encode any higher dimensional containers. For instance, imagine we have a 3x5 two dimensional array. We lay out all the 3x5 = 15 elements linearly in the memory. When the consumer asks for the element at the position (1,2) we find that it exists in 5 * 1 + 2 = 7th position from the beginning.

To capture the dimensions at the type level, they are represented using a type constructor for zero and a *snoc*[17] list. Hence the dimensions are represented like the following:

```haskell
data Z = Z
data tail :. head = !tail :. !head

type DIM0 = Z
type DIM1 = DIM0 :. Int
type DIM2 = DIM1 :. Int
type DIM3 = DIM2 :. Int
```

The polymorphism on the above is obtained by defining a typeclass which defines all the standard operations that can be queried on the dimensions like rank, size etc. The most important polymorphic function is the `toIndex` function which computes the actual position of the element.

[17] *snoc* lists are opposite of cons lists. Refer to [Oka99] for more detail.

```
class Eq sh => Shape sh where
    toIndex :: sh -> sh -> Int
    .....
instance Shape Z where
    toIndex _ _ = 0
instance Shape sh => Shape (sh :. Int) where
    toIndex (sh1 :. sh2) (sh1' :. sh2')
        = toIndex sh1 sh1' * sh2 + sh2'
```

From the context of vectorisation, the beauty of the shape polymorphic API is that it allows us to define a uniform abstract typeclass for every possible dimension of arrays. Because the internal representation is a plain contiguous sequence of elements, we can once again vectorise in chunks of 16, 4, 8 and 2 elements and the operations become almost similar to working with lists. Also writing the internal vector functions for every data structure involves a lot of tedious traversal patterns for each separate representation. With the shape polymorphic API we have a uniform traversal pattern for all types of arrays. This is a testament to the power of parametric polymorphism in Haskell.

7.3 Performance penalties

Throughout the entirety of this chapter we have spoken about the high level design choices using *typeclasses* and *type-families*. However, in this part we would like to point to some performance penalties which a high level abstraction brings with it.

Let us take an example from vector lists to demonstrate our point. We take the example of the vectorised folding function over `VecLists`. Although the `VecList` type is defined as a `newtype` wrapper around plain lists which doesn't have any overhead, there are number of other types which are adding to the overhead. For instance,

```
data Float = F# Float#

data FloatX4 = FX4# FloatX4#

packFloatX4 :: (Float, Float, Float, Float) -> FloatX4
```

```

unpackFloatX4 :: FloatX4 -> (Float, Float, Float, Float)
```

Each of the types and functions listed above are wrappers over lifted types. And any computation involving lifted types involve pointers in the runtime, which adds on to the performance overhead. The packing and unpacking involves additional costs. We discuss about individual performances and timing in detail in Chapter 10.

Additionally there are certain hidden costs of working with the lifted vector types. For instance take a look at these two snippets:

```
foldFloatX4 ::
     (FloatX4 -> FloatX4 -> FloatX4)
  -> (Float -> Float -> Float)
  -> Float
  -> VecList Float
  -> Float

foldFloatX4 f g seed (VecList xs') = go seed (broadcastVector seed) xs'
  where
    go acc vec_acc [] = g (foldVector g vec_acc) acc
    go acc vec_acc (x1:x2:x3:x4:xs) =
      let op = f (packVector (x1,x2,x3,x4)) vec_acc
       in go acc op xs
    go acc vec_acc (x:xs) = go (g x acc) vec_acc xs
```

```
foldFloatX4 f g seed (VecList xs') = go seed xs'
  where
    go acc [] = acc
    go acc (x1:x2:x3:x4:y1:y2:y3:y4:xs) =
      let op = f (packVector (x1,x2,x3,x4)) (packVector (y1,y2,y3,y4))
       in go (g (foldVector g op) acc) xs
    go acc (x:xs) = go (g x acc) xs
```

Both of the above functions implement the same task of folding a vector list using some commutative fold function. We don't need to understand the entire functions, however in line no 6 of the second snippet the `foldVector` function actually unpacks a vector and applies a folding function on it. And as we can see clearly the line no 6 is part of the recursive call and it is called every time the loop runs.

On the other hand in line 10 of the first snippet the `foldVector` operation occurs simply at the end of the list. Throughout the entire loop, unpacking happens just once. So the intuitive guess would be that the first snippet should be much faster than the second snippet. The number of packing operations are same for both of the them, so given the same algorithm, the obvious winner should be the function with lesser unpack operations.

However surprisingly the second snippet is 2 to 3 times faster than the first snippet depending on the length of the list! The surprising cause of the performance penalty in the first case is not very clear, but we can speculate that the first snippet has to loop while accessing the `FloatX4` type accumulator in each loop. This access might be causing the performance loss. Also the second snippet packs eight elements in each loop while the first works with simple four elements at a time and uses the accumulator which could be another possible cause.

As the cost model is such a mysterious and hidden aspect of vectorisation, we provide an entire chapter to talk about a possible cost model for vectorisation in GHC.

7.4 Example: Demonstrating the *lift-vector* API

We provide an example of using the *lift-vector* library to operate on vector lists and arrays and compare the API with plain Haskell lists. A number of other examples[18] are available in the source tree of the *lift-vector* library.

Let us consider the dot product of vectors , which initially does a one to one multiplication of every element in two vectors and then sums the results. Using a plain Haskell list of floats, a dot product would be expressed as,

The above is a plain list based solution which consumes the list, one element at a time. Now our solution using the *lift-vector* library would look like the following,

The solutions looks fairly similar to the one on plain lists. The only difference is that the `fold` operation has to be fed two functions: the vector and the scalar version. And also the lists are converted to vector lists and vector arrays respectively. Let us take

[18]`https://github.com/Abhiroop/lift-vector/tree/master/src/example`

```haskell
dotp :: [Float] -> [Float] -> Float
dotp xs ys = sum (zipWith (*) xs ys)

-- Expressed as a fold
dotp' :: [Float] -> [Float] -> Float
dotp' xs ys = foldr (+) 0 (zipWith (*) xs ys)
```

Listing 7: Dot product using plain lists

```haskell
-- using vectorised lists
dotVec :: [Float] -> [Float] -> Float
dotVec xs ys =
  fold (\x y -> x + y :: FloatX4) (\x y -> x + y :: Float) 0 (
  zipVec
    (\x y -> x * y :: FloatX4)
    (\x y -> x * y :: Float)
    (toVecList xs)
    (toVecList ys))

-- using vectorised arrays
dotVec' :: [Float] -> [Float] -> Float
dotVec' xs ys
 = let l1 = length xs
       l2 = length ys
    in fold (\x y -> x + y :: FloatX4) (\x y -> x + y :: Float) 0 (
       zipVec
        (\x y -> x * y :: FloatX4)
        (\x y -> x * y :: Float)
        (toVecArray (Z :. l1) xs)
        (toVecArray (Z :. l2) ys))
```

Listing 8: Dot product using vector lists and arrays

a look at the amount of parallelism that this operation can utilise.

7.4.1 Parallel Dot Product Illustrated

In Figure 7.3 we illustrate the parallelism in the dot product function. It starts by parallelising the zipping of the two vector lists, using the vector multiplication operation and then proceeds to use the vector addition operation to fold the resultant list.

Generally any algorithm which can be represented using `folds, zips` and `maps` can be expressed in *lift-vector*. For algorithms which cannot be expressed using these higher order functions and combinators, we always have access to the more basic APIs in the

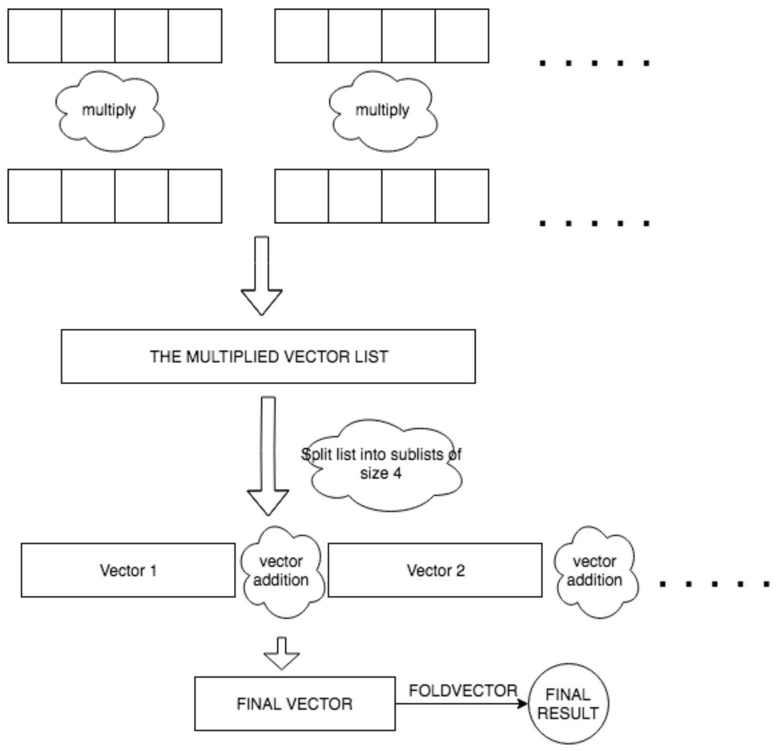


Figure 7.3: Parallelism in Dot Product Illustrated

form of `pack, unpack, broadcast` etc.

We discuss about the performance and trade-offs for this and various other algorithms in Chapter 10.

8. A COST MODELS FOR VECTORISATION

This chapter is going to introduce a very simple cost model of vectorisation. The real significance of cost model arises from automatic vectorisers. Generally the compiler would attempt to apply loop vectorisation or the superword level parallelism (SLP) algorithm [LA00]. However as we discussed in the previous chapter, it is quite hard to have an assessment of the gain or loss from vectorisation. To solve that, most major vectorising compilers like GCC or LLVM have the notion of a cost model which allows them to decide whether to vectorise or not.

While we are not adding automatic vectorisation support to GHC, we present this general cost model as it is a useful tool even for *manual* vectorisation. We initially present the cost model of LLVM, which is available as part of the LLVM codebase[19]. It is not a very fancy model but provides a good starting point.

8.1 LLVM's Cost Model

We are primarily dealing with loop vectorisation, so we are interested in the speedup obtained from vectorising loops. We define the vectorisation factor VF, as the number of scalar elements that fit inside a vector. In LLVM, every operation inside a loop which is affected by the vectorisation, is given a cost of 1 and all those costs are summed up. The sum is divided by the vectorisation factor to provide the possible gain.

$$cost_{loop} = \frac{1}{VF}\Sigma cost_{instr,VF}$$

Each of the individual costs are taken from hardware instruction set lookup tables. Followed by that the compiler calculates,

$$cost_{diff} = cost_{vec} - cost_{scalar}$$

A negative value results in the application of the vectorisation transformation. Also the percentage of instruction savings from vectorisation can be calculated using the below:

$$saving = \frac{|cost_{diff}|}{cost_{scalar}} = \frac{|cost_{vec} - cost_{scalar}|}{cost_{scalar}}$$

[19]`https://github.com/llvm-mirror/llvm/blob/master/lib/Analysis/CostModel.cpp`

8.2 An alternate cost model

While LLVM's cost model works well enough for simple enough cases of vectorisation, there has been studies [PCJ] stating the ineffectiveness of the cost model. This specific study [PCJ] found there is actually not a visible correlation in the instruction saving percentages predicted by LLVM and the actual cost savings. So we try to propose a simpler cost model where the correlation would be more obvious.

$$cost_{loop} = \frac{1}{VF}\Sigma cost_{instr,VF} - \Sigma cost_{pack,unpack,broadcast}$$

The only difference in our cost model is subtracting the cost of any form of packing, unpacking or broadcasting operation which is a major contributor in reducing the gains from vectorisation. A negative cost implies a reduction in the total cost of the computation which can be inferred as a positive outcome from vectorisation. A positive cost implies that the cost of the computation increase upon vectorisation which would discourage a user from vectorising. As for the actual instruction costs, LLVM has an extensive instruction set cost table available with it.

We take this opportunity to mention an extensive piece of work by Agner Fog known as *Agner Fog's optimisation manual*[Fog+11] where the author has provided a detailed set of cost metric for each and every possible instruction on the x86 instruction sets on almost every possible processor set that has been released by Intel and AMD. However, it is too low level for general use, and is primarily a tool for compiler writers to create a cost model.

We have currently not encoded any of this information in GHC. For the sake of an example, lets consider each packing/unpacking operation taking a cost of 1 and an arithmetic operator a cost of 4(equal to the vectorisation factor). Considering the following operation,

```
B[i] = (C[2*i]*(D[2*i]+(E[2*i]*F[2*i])))
```

So if we calculate the final cost for each iteration of the above loop,

$(4/4(multiplication) - (1(packing\ C) + (4/4(addition) - (1(packing\ D) + (4/4(multiplication) - (1(for\ packing\ E) + 1(for\ packing\ F))))))))$

$= (4/4 - (1 + (4/4 - (1 + (4/4 - (1 + 1))))))$

$= (1 - (1 + (1 - (1 + (1 - 2)))))$

$= (1 - (1 + (1 - (1 - 1))))$

$= (1 - (1 + (1 - 0)))$

$= (1 - (1 + 1))$

$= (1 - 2)$

$= -1$

Hence the final cost of vectorising this loop would be negative which implies a gain from vectorisation. And note, this implies the cost per iteration. For an array comprising of a million elements, we process four elements at a time (for Xmm registers) we run 250,000 iteration and we would save cost by -1 for every iteration, which is a generous profit.

However the balance between profitability and loss is quite fragile using this cost model. If we extend the same operation with another packing and an additional arithmetic operation i.e,

```
A[i] = B[i] * (C[2*i]*(D[2*i]+(E[2*i]*F[2*i])))
```

assuming the previous cost per instruction, the final cost of the loop comes out to be +1 which implies that vectorisation results in a loss. We show more example metrics involving this cost model in Chapter 10.

9. AUTOMATIC VECTORISATION

Most of the work that we have done till now involve *explicit* vectorisation. The programmer needs to have full awareness of the cost model as well as the architecture specific support for vector instructions. While it is possible for library authors to possess this knowledge, the major goal is to bring vectorisation to the masses while retaining the original declarative and high level features of Haskell. As we saw in chapter 7, the library *lift-vector* provides a few higher order functions and data types for vector programming but it again delegates the task of choosing between the `FloatX4` or `FloatX8` or `FloatX16` type to the programmer. Also a number of complex algorithms are not expressible using the limited number of combinators that *lift-vector* provides.

As an attempt to mitigate all of the above problems, we propose to make the compiler intelligent enough so that it can *implicitly* vectorise a given Haskell program. We sketch out a possible blueprint for automatic vectorisation in GHC, without delving into the details of various vectorisation algorithms in Figure 9.1.

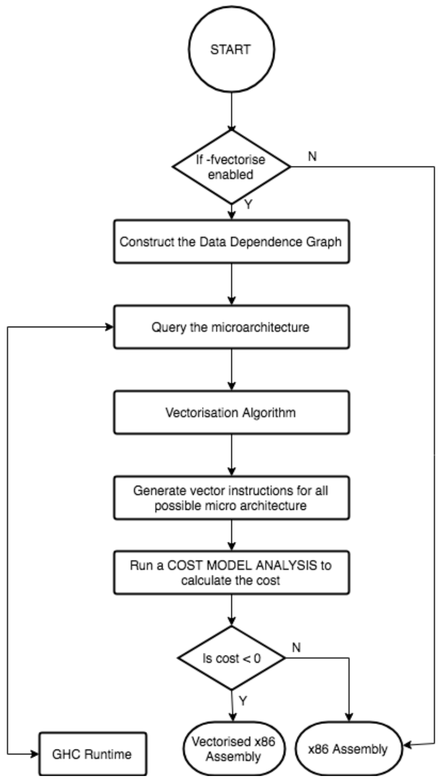

START
If -fvectorise enabled
N
Y
Construct the Data Dependence Graph
Query the microarchitecture
Vectorisation Algorithm
Generate vector instructions for all possible micro architecture
Run a COST MODEL ANALYSIS to calculate the cost
Is cost < 0
N
Y
GHC Runtime
Vectorised x86 Assembly
x86 Assembly

Figure 9.1: A blueprint of automatic vectorisation in GHC

The steps demonstrated in the above figure attempts to produce a possibly optimal assembly code. The *most optimal* code always need not be a vectorised code because the cost of the various packing and unpacking operations might dominate the actual gains from the vector operations, as demonstrated in the last chapter. We initially check if an imaginary `-fvectorise` flag is enabled. Followed by that it generates the data dependence graph of the entire program.

A data dependence graph [FOW87] is a directed graph where the nodes constitute an atomic block of instructions and the edges denotes the corresponding other instructions on whom a particular nodes *depends* on. We provide the formal definition of *dependence* in the next section. The data dependence graph allows the compiler to detect the loops and other control flow structures where vectorisation is possible. The same information could also be obtained from a control flow graph (which GHC already builds[20]) but that would eventually involve running a separate *dependence analysis* pass.

After crafting the dependence graph and detecting the loops, the compiler needs to query the processor micro-architecture from the runtime. As different micro-architecture support a wide permutation of vector instruction sets (AVX, SSE etc) the compiler should try to generate all the possible permutations. So the intermediate representation that the vectorisation algorithm emits, has to experiment with multiple permutations of vectorisation instructions and run a *cost model analysis* to detect the form with the least cost.

Critique: A number of operations in the above blueprint are computationally intensive. The vectorisation algorithm tends to be an NP-Complete problem. Similarly finding the ideal combination of instructions, mixing the various micro-architectures, can result in generating too many permutations. As a result, any of our experiments in automatic vectorisation are not ready to be merged to the GHC mainline currently. They result in drastic increase in compilation times which is a severe red flag for most industrial compilers. However, from a research perspective we highlight the possible vectorisation algorithms that we have briefly experimented with.

9.1 Loop Vectorisation and Dependence Analysis

There are primarily two classes of automatic vectorisation algorithms: loop based vectorisation [NZ08] and superword level parallelism (SLP) based algorithm [LA00]. In

[20]GHC's control flow graphs: `https://github.com/ghc/ghc/blob/master/compiler/cmm/MkGraph.hs`

this section we discuss about the first class of algorithms.

Loop vectorisation, as the name implies are focused on vectorizing loops. Generally loops are unrolled or to be more specific, they undergo a transformation known as strip-mining [Wei91]. Strip mining unrolls a loop, taking into account the vectorisation factor.

```
i = 1

do while (i<=n)

  a(i) = b(i) + c(i) ! Original loop code

  i = i + 1

end do
```

```
!The vectorizer generates the following two loops

i = 1

do while (i < (n - mod(n,4)))

! Vector strip-mined loop.

  a(i:i+3) = b(i:i+3) + c(i:i+3)

  i = i + 4

end do

do while (i <= n)

  a(i) = b(i) + c(i)     !Scalar clean-up loop

  i = i + 1

end do
```

Listing 9: Before and after strip-mining

As we can see in Listing 9 the final part of the loop undergoes undergoes a `scalar clean up` which is akin to the *scalar dribble* problem which we discussed in Section 7.2.2. Strip-mining is applicable on nested loops as well, as demonstrated by Weiss et al [Wei91]. However, we have chosen a contrived example in Listing 9 and generally the structure of loops are much more complex where they possess, what are called *inter and intra loop dependences*. Most loop vectorisation algorithms initially involve a phase of *dependence analyses*.

Dependence Analysis : An individual statement S_2 is said to be dependent on S_1 if S_2 can be executed only after S_1. For example:

```
S1: x = 5;
S2: y = x + 3;
```

In the above case S_2 depends on S_1 to calculate the value of x before it can calculate y. Similarly it is possible to have dependences in loops,

```
for(i = 2; i < n; i++)
   a[i] = a[i - 1] + b [i - 2];
```

There are principally four type of dependences:

- True or flow dependences : When a read happens after write. Example:

```
    for(i = 2; i < n; i++)
       a[i] = a[i - 1];
```

- Anti dependence: When a write happens after read. Example:

```
    for(i = 2; i < n; i++)
       a[i] = a[i + 1];
```

- Output dependence: When a write happens after a write

```
    for(i = 2; i < n; i++)
       a[i] = i;
       a[i] = 8;
```

- Input dependence: When a read happens after a read.

```
for(i = 2; i < n; i++)
   b = a[i] + 10;
   c = a[i] - 10;
```

Of all the classes of dependences mentioned above, flow dependences and anti dependences majorly affect vectorisation and are called *loop carried dependences*. The other dependences are majorly harmless or are eliminated by other optimisation passes. There are a number of famous dependence analysis algorithms like the *Omega test* [Pug91] which can be integrated into GHC. The Omega Test uses an integer programming algorithm and the authors state that, contrary to the common wisdom of NP complete problems being computationally intensive, it runs *relatively* fast.

We are yet to implement the Omega Test in GHC, however after integrating the solution into any compiler, strip mining becomes a trivial task which eventually would lead to an efficient automatic loop vectorisation algorithm in GHC.

9.2 Throttled superword level parallelism algorithm

Throttled superword level parallelism or the TSLP algorithm belong to a family of auto-vectorisation algorithms, which was originally born from the SLP algorithm introduced by Larsen et al. [LA00]. The Throttled SLP algorithm [PJ15], which is the current state of the art vectorisation algorithm inside LLVM[21] applies a modification on the original SLP algorithm by throttling it using a vectorisation cost model analysis. The SLP family of algorithms are not limited to loops, although in general the most profitable outcomes arises from vectorising loops.

The SLP algorithm: This algorithm is a subset of instruction level parallelism. SLP looks for a number of properties inside the intermediate representation of the program. Primarily it looks for instructions which are *isomorphic* to each other and packs them together. Isomorphic statements are those which undergo the same vector operation. It uses a chain like data structure called *use-def* chains to populate what it refers as the *pack set* which is the list of instruction it should pack together.

Additionally SLP also runs an *alignment analysis* pass which looks for memory accesses which are aligned with each other and tries to pack them together. The general

[21]as of version 3.6

idea is to embrace the cost model of the compiler wholeheartedly and use that to vectorise the operations. We show a sample pass of SLP algorithm below.

```
A = X[i + 0]
C = E * 3
B = X[i + 1]
H = C - A
D = F * 5
J = D - B
```

General loop vectorisation algorithms would have a hard time vectorising the above snippet because of the complex dependences between the various statements. However, SLP uses a simple logic of grouping the *isomorphic* operations (using *use-def* chains) and also consecutive memory accesses, across a number of passes to finally produce:

```
(A,B) = X[i : i + 1]
(C,D) = (E,F) * (3,5)
(H,J) = (C,D) - (A,B)
```

We use tuples to denote vector registers above and each of the arithmetic operators are now their vector equivalent.

Throttled SLP: As the name implies, the TSLP algorithm simply uses the appropriate cost model to throttle the vectorisation when it's cost exceeds the scalar version. TSLP constructs the entire SLP graph and then cuts the graph at all the positions and inspects the final cost of vectorisation. Quite unsurprisingly this is also a computationally intensive technique, as it effectively boils down to finding *all* the connected subgraphs including the root instruction, inside the SLP graph. The contribution of the paper [PJ15] is to find a linear time algorithm for the graph cutting problem. Interested readers are encouraged to study it to find more details about the linear time algorithm.

9.3 Reflection

Throughout this chapter we discuss about two forms of vectorisation algorithms: loop based and straight line vectorisation (SLP). It is quite visible that the general procedure of auto-vectorisation is relatively computation heavy, involving solving multiple

NP-Complete problems. Additionally the presence of a number of micro-architectures and their corresponding vector instruction sets further complicates the issue.

Our preliminary experiments with loop based vectorisation were disappointing, showing a drastic increase in compile time and no clear visible performance gains. Within the time frame we could only support a very incomplete prototype of the TSLP algorithm, inside the GHC compiler. We discuss in Section 11.2, our possible avenues of research to improve the automatic vectorisation support of GHC in the future.

10. EVALUATION

In this chapter we demonstrate a series of memory intensive algorithms and vectorise them using the *lift-vector* library. We compare the performance of the library with solutions using plain Haskell lists. Within the time frame of the project there was little to almost no performance optimisation done inside *lift-vector*. As a result the performance is not up to the mark that a properly optimised vectorised function can attain. We demonstrate the declarative API as well as the current numbers nonetheless. All the benchmarks were run on a MacBook Pro with 2.7 GHz Intel Core i5 processor(sandy bridge micro-architecture) and 8 GB DDR3 RAM. The timings were measured using `getCPUTime` function from the `System.CPUTime` module provided by base. All the test cases were randomly generated using the `random`[22] package. The entire source code of the benchmarks are available online[23]. For vectorisation the AVX family of vector instructions have been used.

10.1 Dot Product of Vectors

We have demonstrated the dot product of vectors using plain Haskell lists as well as the two central data structures of *lift-vector* : vector lists and vector arrays, in Section 7.4. The dot product functions presented in Listings 7 and 8 accepts two lists of floats. Each element represents a dimension of the vector. We ensure that the number of elements are same in both the lists. We provide input data size in the following order:

- 10 elements
- 100 elements
- 10,000 elements
- 100,000 elements
- 1,000,000 elements
- 2,000,000 elements
- 3,000,000 elements

[22] `http://hackage.haskell.org/package/random`
[23] `https://github.com/Abhiroop/lift-vector/blob/master/src/example/Main.hs`

- 4,000,000 elements
- 5,000,000 elements

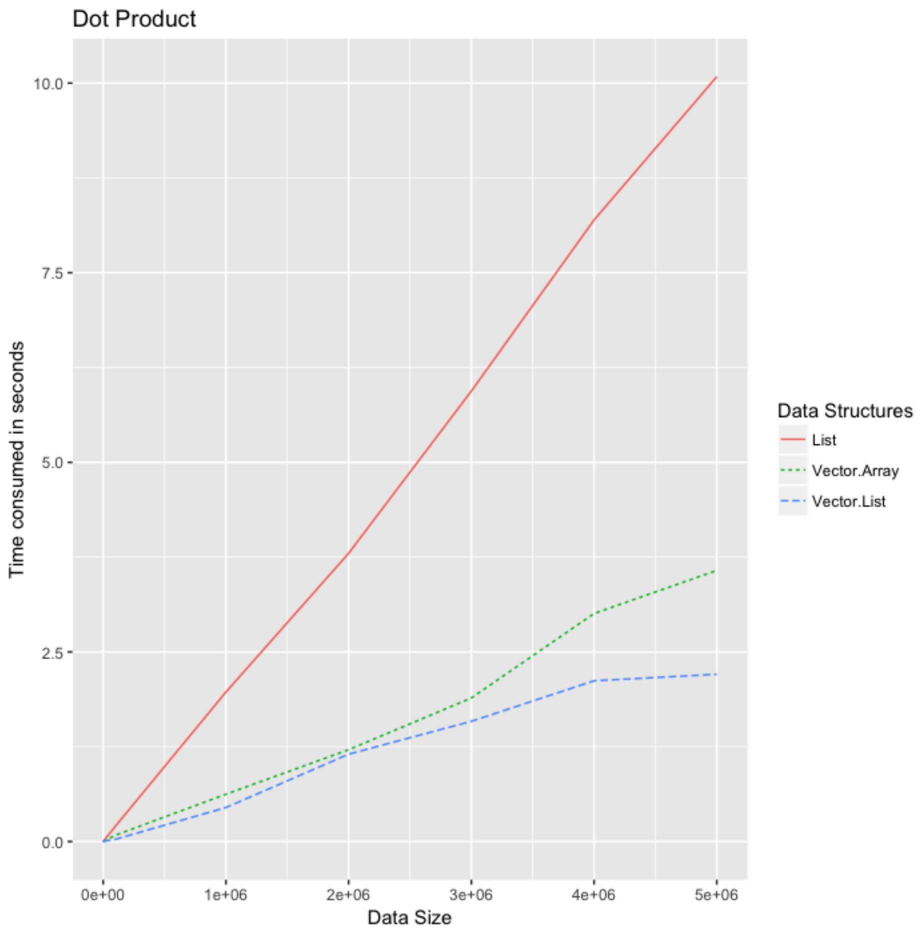


Figure 10.1: Benchmark of dot product of two vectors

In Figure 10.1 the X-axis represent the entire size of the lists. The Y-Axis represents the time taken for the function to run in seconds.

Discussion: We can see in the above figure, that vector lists performs the best out of the three data structures. An optimised vector array could effectively perform much better than a plain Haskell list or a vector list. However here we see its performance in the range between lists and vector lists. The reason, as discussed in Section 7.2.3, is the lack of in place writes. So the amount of performance gains that it has from vectorisation as well as stream fusion (from the *vector* library) is not fully visible due to the final `O(n)` write iteration. Additionally all of our efforts on performance optimisation like loop unrolling and benchmarking the correct data size for packing, have been focused on vector lists. Relatively the vector arrays are purely un-optimised code with just plain vectorisation.

Another important observation is as the size of the data increases, the impact of vectorisation becomes more profound. We can see in the final data set of 5 million elements, vector lists perform almost 2.5X faster than general lists.

10.2 Matrix Multiplication

Matrix multiplication is the traditional example used as a benchmark in most high performance computing literature. It is one of the most applied algorithms in computer science with applications in recommendation systems, machine learning, graphics, games and a number of other applications. As a result matrix multiplication is commonly one of the biggest targets of parallelism. When multiplying matrices each of the individual components being multiplied are independent of the other multiplications. As such the algorithm can leverage both vectorisation as well as multi-core parallelism. However, we choose to solely demonstrate the speedups obtained from vectorisation.

Using plain Haskell lists matrix multiplication can be written as:

```haskell
matmult :: [[Float]] -> [[Float]] -> [[Float]]
matmult a b = [[sum (zipWith (*) ar bc) | bc <- (transpose b)] | ar <- a]
```

This solution is definitely not the most optimal one. For one, the transpose operation is quite costly. But it is able to express matrix multiplication using the *zipping* and *folding* combinators. As a result this can be very easily translated to vector lists and vector arrays from *lift-vector*.

```haskell
-- using vector lists
matmultVec :: [[Float]] -> [[Float]] -> [[Float]]
matmultVec a b =
  [ [ fold (\x y -> x + y :: FloatX4) (\x y -> x + y :: Float) 0 (
  zipVec
    (\x y -> x * y :: FloatX4)
    (\x y -> x * y :: Float)
    (toVecList ar)
    (toVecList bc))
  | bc <- (transpose b)
  ]
  | ar <- a
  ]

matmultVec' :: [[Float]] -> [[Float]] -> [[Float]]
matmultVec' a@(f:_) b =
  let l = length f
  in [ [ fold (\x y -> x + y :: FloatX4) (\x y -> x + y :: Float) 0 (
         zipVec
         (\x y -> x * y :: FloatX4)
         (\x y -> x * y :: Float)
         (toVecArray (Z :. l) ar)
         (toVecArray (Z :. l) bc))
       | bc <- (transpose b)
       ]
     | ar <- a
     ]
```

The algorithm translates almost line by line from the list based solution. The benchmarks that we shall run on all three solutions are run for *dense* matrices. There are a number of techniques for vectorizing operations on sparse matrices [Bol+03]. However for the purpose of this benchmark we limit ourselves to randomly generated dense matrices. We use square matrices for our benchmarks, but the algorithms can accept any dimension. However we do not impose any runtime or compile time error for mismatched dimensions.

Lazy Evaluation: Every step of the matrix multiplication in the programs represented above is lazy. For the other examples we are using the *deepseq* function from the `Control.DeepSeq`[24] module. The general *seq* function evaluates a data structure to it's Weak Head Normal Form. However we require more deeper evaluation in case of this function. Also additionally we use the following function:

```
forceElements :: [a] -> ()
```

[24]http://hackage.haskell.org/package/deepseq

```
forceElements = foldr seq ()
```

to walk across each and every element of the two dimensional matrices and ensure that evaluation occurs. Hence `deepSeq` and `forceElements` together to evaluate the entire resultant matrix. Let us look at the performance benchmarks now.

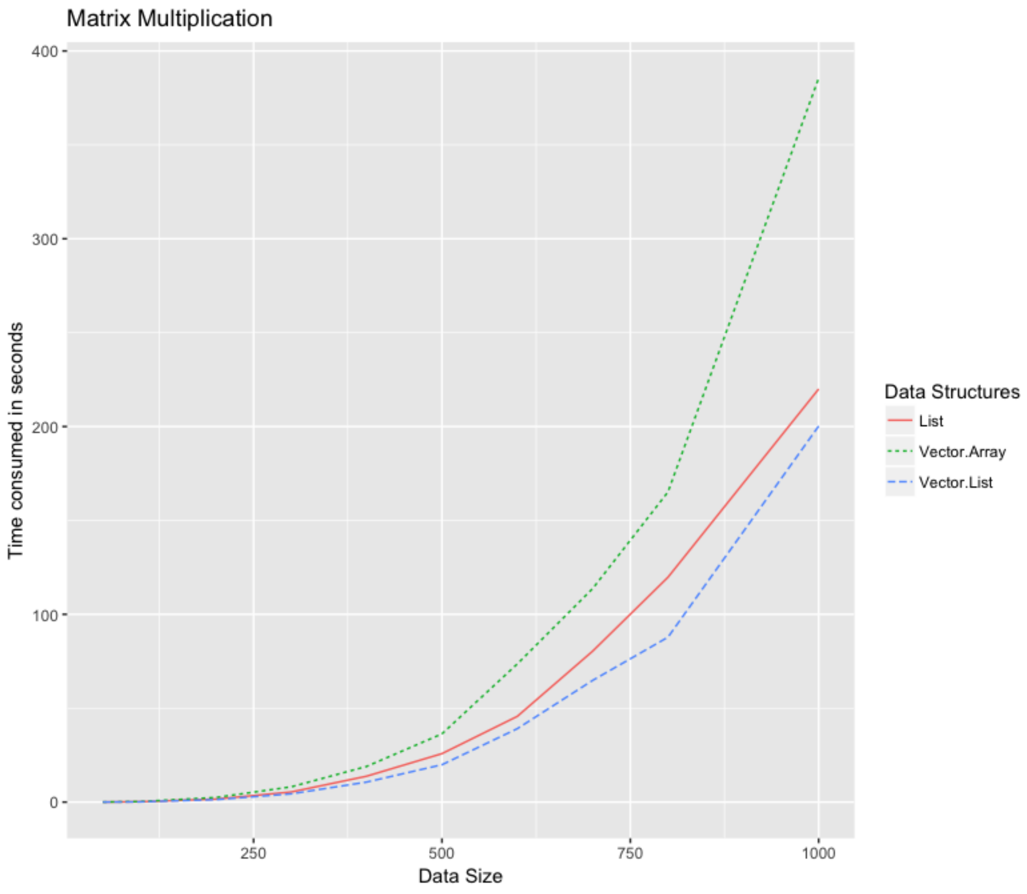


Figure 10.2: Benchmark of matrix multiplication of two dense matrices

The X-axis in the above figure refers to the number of rows or columns of a matrix. As they are square matrices the number of rows and columns are same. The Y axis represents the amount of time taken in seconds. It should be noted that the timings indicated in this benchmark also additionally includes the call to the `forceElements` function which walks across the entire matrix evaluating it.

Discussion : In the result we can see that vector lists once again perform better than vector arrays as well as plain Haskell lists. The poor performance of vector arrays is primarily due to the `O(n)` writes which is happening for each row n times. In the previous example the function folds the value so instead of writing to a new array it accumulated the value which allowed it to perform better than plain lists. But in this case the result is a full fledged matrix which is re-created and has a poor write performance. Hence, in any algorithm which would involve folding, vector arrays and vector lists would perform much better than plain lists due to vectorisation. But the linear time write spoils the overall gains in this case.

Additionally, the performance of vector lists weren't much better than plain lists. We use matrices of dimension 50x50, 100x100, 200x200, 300x300, 400x400, 500x500, 600x600, 700x700, 800x800 and 1000x1000. By the final matrix, vector lists take 200 seconds[25] while plain lists take approximately 220 seconds, which is only 1.1x the original performance. It is possible to do much better with more effective optimisation.

Also a basic performance analysis shows us that the `transpose` function is the real bottleneck, and matrix multiplication can be done without the costly `transpose` operation. In fact Repa defines matrix multiplication using a separate `traverse`[26] function where it is able to beat the performance of a fully vectorised hand written code. In Repa integer based matrix multiplication of 1000x1000 dimension matrices can happen in tens of seconds. Hence, if we integrate mutable vectors as well as define matrix multiplication using the traversal method, vector arrays can easily beat the performance of vector lists as well as plain lists by a huge margin.

10.3 Polynomial Evaluation

For our next section we have chosen the example of polynomial evaluation. We have already discussed about this problem Section 6.5. The implementation that we provided in Listing 4 was entirely written using the native unlifted types of GHC. Let us try to implement the same algorithm using *lift-vector*.

Unfortunately the polynomial evaluation problem cannot be written in a declarative way using folds and zips. The plain Haskell list based solution looks like this:

```haskell
evalPoly :: Float -> [Float] -> Float
evalPoly value coeffs = go coeffs (length coeffs - 1)
  where
    go [] _ = 0
```

[25]this timing includes the time taken to traverse the entire resultant matrix

[26]`https://hackage.haskell.org/package/repa-3.4.1.3/docs/src/Data-Array-Repa-Operators-Traversal.html#traverse`

```
    go (x:xs) len = (x * (value ^ len)) + go xs (len - 1)
```

The issue with this representation is that it is recursive and each iteration is dependent on the value of the previous. If we try to abstract over this and present the polynomial evaluation as a fold,

```
evalPolyFold :: Float -> [Float] -> Float
evalPolyFold value coeffs
  = let index_coeffs = (zip coeffs
                                (reverse
                                    ( take
                                        (length coeffs)
                                        (iterate (+ 1) 0))))
     in foldr (\\(c,p) v -> v * c^p) 0 index_coeffs
```

Even in the above representation we cannot get rid of the fact that each iteration calculates the value using the previous iteration. We showed in Figure 6.7 of Section 6.5 of how polynomial evaluation can be vectorised. And if we hope to leverage that vectorisation we have to use the more lower level primitives from the `Data.Primitive` module in *lift-vector*. The final vectorised programs looks like the following:

```
evalPolyVec :: Float -> [Float] -> Float
evalPolyVec value coeffs = go (broadcastVector value) coeffs (length coeffs)
  where
    go _ [] _      = 0.0
    go vec_val (x:y:z:w:xs) len =
      let packed_coeff                    = packVector (x, y, z, w) :: FloatX4
          step_length                     = len - sizeOfVec
      in (go' packed_coeff vec_val step_length) + (go vec_val xs step_length)
      where
        go' pc _ 0 =
          let (x1, x2, x3, x4) = unpackVector pc
          in (x1 * value ^ 3) +
       (x2 * value ^ 2) +
       (x3 * value) + x4
    go' pc v l =
     let t = pc * v
     in go' t v (l - 1)
  go vec_val (x:xs) len = x + (go vec_val xs len)
```

The unlifted types are noticeably missing and the program is starting to resemble a standard worker wrapper representation[GH09] in Haskell. However there is hefty amount of scalar cleanup that needs to be done owing to the structure of the program. Let us take a look at the benchmarks now.

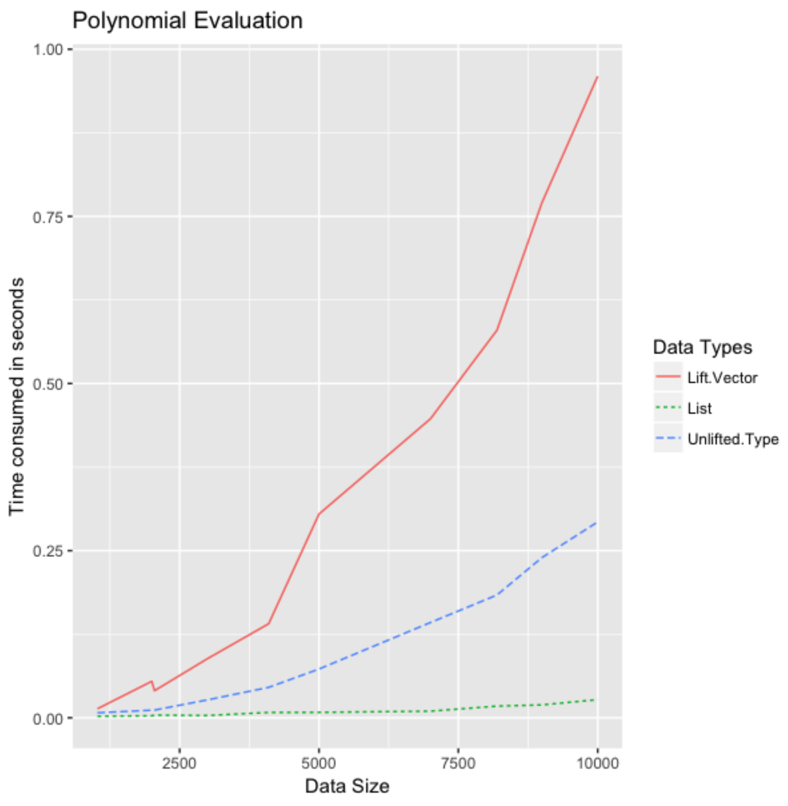


Figure 10.3: Benchmark of polynomial evaluation

The X axis indicates the order of the polynomial. We limited the possible orders to avoid blowing up the main memory. The Y axis indicates the time consumed in seconds for computing.

Discussion: The result is rather disappointing in the fact that plain Haskell lists performs way better than both the unlifted typed solution as well as the solution with *lift-vector*. The cost of the higher level abstractions of *lift-vector* starts to outweigh the benefits of vectorisation. At polynomials of order 10000, unlifted vector types perform close to 3.5 times better than the *lift-vector* solutions. That is not to say that abstractions are bad. It points to a scope of improvement especially for alignment related issues in vectorisation. In fact the entire cost model of Chapter 8 was conjured by analysing this particular algorithm a number of times.

We could see in the past two benchmarks, vector lists dominate all other alternatives, but this particular benchmark sheds light into the performance issues plaguing the *lift-vector* library. We discuss some possible steps to alleviate these issues in the section on future work.

10.4 Pearson Correlation coefficient

In this section we look at the calculation of the Pearson Correlation Coefficient. The Pearson Correlation coefficient provides a measure of correlation between two sets of data. It has applications in data mining and analytics. It is mathematically expressed as:

$$r = \frac{n\Sigma x_i y_i - \Sigma x_i \Sigma y_i}{\sqrt{n\Sigma x_i^2 - (\Sigma x_i)^2}\sqrt{n\Sigma y_i^2 - (\Sigma y_i)^2}}$$

In the above formula n is the sample size, x_i and y_i are the individual sample point indexed with i. The value of the correlation coefficient falls between -1 and +1. -1 implies entirely negative correlation whereas +1 means a positive correlation. A value close to 0 implies no correlation.

It is fairly straightforward to represent the above formula using plain Haskell lists.

```haskell
pearson :: [Float] -> [Float] -> Float
pearson xs ys
  | (length xs) /= (length ys) = error "Incorrect dataset"
  | otherwise =
    let n = fromIntegral (length xs) :: Float
        num = (n * (sum (zipWith (*) xs ys))) - ((sum xs) * (sum ys))
        denom1 = sqrt $ (n * (sum (map (^ 2) xs))) - ((sum xs) ^ 2)
        denom2 = sqrt $ (n * (sum (map (^ 2) ys))) - ((sum ys) ^ 2)
```

```
      in num / (denom1 * denom2)
```

The above program has a number of points of parallelism. Firstly there exists the `sum` operations over large data sets which can be represented as vectorised folds. Also the exponentiation operator can be reproduced using the `zipVec` combinator to exploit the parallelism. So in case of both vector lists as well as vector arrays we define a set of helper functions to make the code more declarative. This is what the final vectorised code in *lift-vector* looks like:

```
-- using vector lists
pearsonVec :: [Float] -> [Float] -> Float
pearsonVec xs ys
  | (length xs) /= (length ys) = error "Incorrect dataset"
  | otherwise =
    let n = fromIntegral (length xs) :: Float
        num = (n * (sumVecF (zipMultF xs ys))) - ((sumVecF xs) * (sumVecF ys))
        denom1 = sqrt ((n * (sumVecF (zipMultF xs xs))) - ((sumVecF xs) ^ 2))
        denom2 = sqrt ((n * (sumVecF (zipMultF ys ys))) - ((sumVecF ys) ^ 2))
     in num / (denom1 * denom2)

sumVecF :: [Float] -> Float
sumVecF xs =
  fold (\x y -> x + y :: FloatX4) (\x y -> x + y :: Float) 0 (toVecList xs)

zipMultF :: [Float] -> [Float] -> [Float]
zipMultF xs ys =
  fromVecList (
  (zipVec
     (\x y -> x * y :: FloatX4)
     (\x y -> x * y :: Float)
     (toVecList xs)
     (toVecList ys)))

-- using vector arrays
```

```haskell
pearsonVec' :: [Float] -> [Float] -> Float
pearsonVec' xs ys
  | (length xs) /= (length ys) = error "Incorrect dataset"
  | otherwise =
    let n = fromIntegral (length xs) :: Float
        num =
          (n * (sumVecF' (zipMultF' xs ys))) - ((sumVecF' xs) * (sumVecF' ys))
        denom1 = sqrt ((n * (sumVecF' (zipMultF' xs xs))) - ((sumVecF' xs) ^ 2))
        denom2 = sqrt ((n * (sumVecF' (zipMultF' ys ys))) - ((sumVecF' ys) ^ 2))
     in num / (denom1 * denom2)

sumVecF' :: [Float] -> Float
sumVecF' xs =
  fold
    (\x y -> x + y :: FloatX4)
    (\x y -> x + y :: Float)
    0
    (toVecArray (Z :. (length xs)) xs)

zipMultF' :: [Float] -> [Float] -> [Float]
zipMultF' xs ys =
  let l = length xs
   in fromVecArray (
      (zipVec
         (\x y -> x * y :: FloatX4)
         (\x y -> x * y :: Float)
         (toVecArray (Z :. l) xs)
         (toVecArray (Z :. l) ys)))
```

Both the programs heavily resemble the original list based solution. Again we should remember that this is not the most efficient form of the program. For instance, `fromVecArray` call is a very costly `O(n)` call, which is added in the all important `zipMultF'` function. Although this makes our API quite declarative, this will prove to be a costly design choice in our benchmarks. Lets us take a look at the benchmarks now.

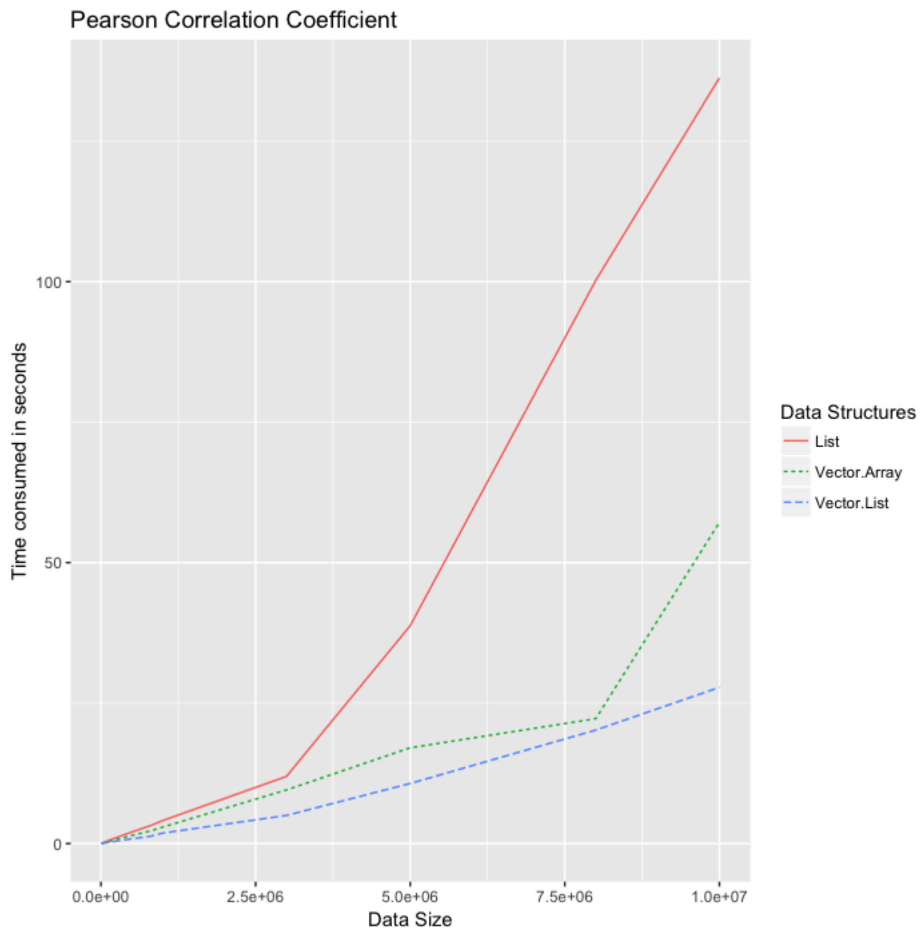


Figure 10.4: Benchmark of calculating the Pearson Correlation Coefficient

In Figure 10.4 the X axis represents the size of each dataset whereas the Y axis represents the time consumed in calculating the Pearson correlation coefficient, in seconds.

Discussion : Once again in this algorithm vector lists outperform both vector arrays as well as plain lists. The reasoning is quite similar to the discussion in the dot product algorithm. Consequently vector arrays show a decent performance despite the costly `fromVecArray` method call. An optimised vector array can easily outperform vector lists owing to stream fusion. In this case we have have used the final data set of 10 million elements where the performance of vector lists are 5 times better than plain lists, which is quite an encouraging gain.

10.5 Benchmarking infrastructure

In this section we briefly describe the benchmarking infrastructure that we use. Figure 10.5 gives a broad overview of the entire infrastructure.

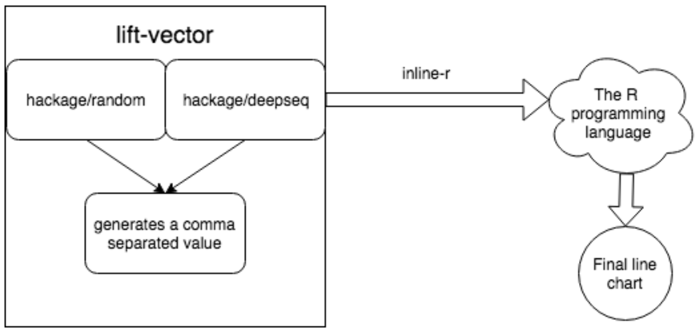


Figure 10.5: Benchmarking infrastructure in *lift-vector*

We have entirely used handcrafted Haskell 98 functions to design the benchmarking code. Two of the external package that we have used are *random* for generating various kinds of random inputs and *deepseq* for deep evaluation of data structure.

Haskell already provides a *seq* function for evaluating an argument, however *seq* evaluates an argument up to its weak head normal form. The *deepseq* function provides full evaluation. It can be understood through this example.

```
> [1,2,undefined] `seq` 3
 3

> [1,2,undefined] `deepseq` 3
*** Exception: Prelude.undefined
```

The above shows that *deepseq* evaluates to the inner depths of the structure. We run each benchmark and generate a comma separated value file, which is fed to the R programming language via the *inline-r*[27] package. R is a powerful language for data munging and graph generation and we exclusively generate all our charts using R.

10.6 Reflection

Throughout this chapter we have witnessed various benchmarks for measuring the performances of various components of the *lift-vector* library. Broadly *lift-vector* provides two major modules:

- `Data.Operations` - this defines the `ArithVector` typeclass which provides all the

[27]https://hackage.haskell.org/package/inline-r

higher order functions for mapping, folding and zipping. This is generally the first choice of API.

- `Data.Primitive` - when an algorithm cannot be expressed using `Data.Operations`, a library author has the option of manually packing and unpacking the vectors and handcrafting the entire algorithm. This should be used as a last resort when the algorithm has no chance of being expressed through the higher order functions

In most of the benchmarks vector lists triumph over the other data structures. This is primarily owing to some internal optimisations that we have applied to vector lists and obviously the vectorisation, parallelising the throughput. Vector arrays perform quite well, given the fact that there was no performance optimisations applied on it. However the polynomial evaluation function is a proof of the difficulties of vector computations. When alignment of the vectors are incorrect they tend to perform slower than the plain list based solution, as demonstrated in the polynomial example.

There are a number of other possible numeric computational examples like Fast Fourier Transform and stencil computation problems like Successive Over Relaxation, which could be possibly expressed using *lift-vector*. The API of the `ArithVector` typeclass needs to be extended to include vectorised generic traversals [LV02]. Additionally there is a huge scope of improving the performance of vector arrays by introducing mutable vectors and allowing in-place writes for such write-heavy and memory intensive algorithms.

11. CONCLUSION

This thesis outlines the design and implementation of vectorisation in the Glasgow Haskell Compiler. Vectorisation as a domain of high performance computing has primarily resided among the reigns of imperative programming languages like Fortran, C and C++. Through this project we show it is quite possible to exploit superword level parallelism in a purely functional language. Over the course of this report, we have demonstrated all the moving parts of the Glasgow Haskell Compiler as well as the components which we have modified across a period of three months. We have also introduced a new library for vector programming. We outline our contributions in the following sections.

11.1 Contributions

- We have extended the Cmm *MachOps* or *machine operations* to understand vector instructions. This lays the groundwork for vectorisation in GHC. Now the native code generator can be extended as much as necessary, to include newer and more advanced vector instruction sets like AVX-512.
- We have also provided a proof of concept support for actually emitting the vector assembly instructions from Cmm. We have chosen to implement arithmetic operations from both the AVX as well as the SSE family, to operate on the 128-bit wide Xmm registers and shown significant performance improvement.
- Another contribution was the support for 8, 16 and 32 bits integer types in GHC, which provides a substrate for implementing SIMD operations on integers in the future.
- The *lift-vector* library is the first of its kind, in the Haskell ecosystem which provides users a reasonably declarative API for directly programming with vectors. Its API is a significant improvement compared to imperative counterparts. However it has room for substantial progress both in terms of the API as well as performance.
- We have also proposed a new cost model based on our experiments, to predict the costs and benefits of vectorisation.
- Finally as part of this thesis, we have attempted to reason about possible benefits in automatic vectorisation by indtroducing other intermediate representations inside

GHC. Although the compilation times have suffered, but introducing a new IR in GHC could open up opportunities for other data flow analysis related improvements.

11.2 Future Work

Keeping in mind our contributions to the Glasgow Haskell Compiler infrastructure, in this section we talk about a set of research and engineering problems that we can tackle in the future, based on the current work.

11.2.1 Supporting wider vectors

We have modified the native code generator backend of GHC and extended the Cmm grammar to allow it to express vector operations. To demonstrate the effectiveness of the effort, we have added support for AVX and the SSE family of instructions. All of the vector instructions operate on the 128-bit XMM registers.

The compiler is currently incapable of utilising the 256-bits wide YMM and 512-bits wide ZMM registers. To support these superword registers, we need to make certain small changes in the the code generator. Adding support for the broader registers will also require us to provide wrapper types like `FloatX8`, `FloatX16` etc, which are relatively trivial to implement. At the same time we need to benchmark the performance gains from mixing various families and width of vector operations.

11.2.2 Improving the *lift-vector* API

The *lift-vector* API for vector lists, currently requires the consumer to provide both scalar and the vector arithmetic operations.

```haskell
-- Multiply all the elements of a list
multiply :: [Float] -> Float
multiply xs =
  fold (\x y -> x * y :: FloatX4) (\x y -> x * y :: Float) 0 (toVecList xs)
```

While the overloading of the vector operation simplifies the API considerably, the consumption of *both* the scalar and vector functions makes it slightly awkward. A possible solution might be to introduce a *monadic* API to carry around a dictionary which store the mapping from each vector to scalar function. However, that would force all the operations to be carried out inside the monad. The current API replicates the stateless API with lists quite faithfully, and the trade-off would be between expressivity and familiarity.

Additionally, there are a number of *compositional* typeclasses instantiated by lists like `Applicative, Monad, Alternative`. A lot of them do not have any currently visible advantage in vector lists, but we can research more to find any possible way to leverage these and some more useful typeclasses.

11.2.3 Improving the *lift-vector* performance

We have observed in Chapter 10, the performance regressions of *lift-vector* while evaluating polynomials. This was primarily caused by misaligned vectors and the heavy cost of packing-unpacking. The cost model described in Chapter 8 doesn't account for alignment issues. Alignment is an age old problem plaguing vectorisation and the only cure is a programmer meticulously checking the assembly and ensuring proper alignment.

Additionally the performance of vector arrays have always lagged behind vector lists despite leveraging the stream fusion optimisation. An initial line of work should be to integrate mutable vectors in vector arrays which would allow it to have `O(1)` in place writes. Also a vector array code is unable to tap in to the join point optimisation [Mau+17] happening in GHC, which should be investigated and integrated with the vector array module.

11.2.4 More vectorised data structures

We currently provide support for vectorised lists and arrays and they are sufficient enough to demonstrate the declarative API that we are aiming for. However lists or more formally linked lists are traditionally not very frequently used in performance critical applications. Most high performance programming revolves around programming with arrays.

This report serves as a proof of concept that vector instructions can be supported by the compiler, however there exists a number of interesting operations like traversals, finding the lowest predecessor etc on various data structures which could be implemented in the future inside the *lift-vector* library.

Some low hanging fruits like priority queue vectorisation [CRM92] or tree traversal vectorisation [JGK13] should show immediate gains compared to their scalar counterparts.

11.2.5 Automatic Vectorisation

We spoke about various automatic vectorisation approaches in Chapter 9. Initial results for automatic vectorisation were disappointing, however a major cause for the poor results were the intermediate representation languages used inside GHC. Almost every major vectorising compiler uses the *Static Single Assignment (SSA)* form representation [Cyt+89].

We provide a brief introduction to SSA in Section 5.1. The SSA form is an improvement over *use-def* chains for data flow and control analysis. Introducing an SSA based IR in GHC could result in improved code generation in the LLVM backend as well. It is a broad topic of its own and there are number of open research challenges in experimenting with not just SSA but other intermediate representations like program dependence graphs [FOW87], and finding out the best possible representation to aid auto-vectorisation.

11.2.6 Programming Models for Vectorisation

With the introduction of the Intel SPMD Compiler [PM12] in 2012, a new model of vector programming called *SPMD on SIMD* was introduced. *SPMD on SIMD* ports the programming model of Open MPI to vector machines, by providing abstractions which allow each SIMD unit to operate on different pieces of data in a parallel manner. There are no locks involved and it presents a very declarative language level API which can be succinctly expressed in Haskell. However there needs to be foundational changes in the GHC runtime to support this model. This presents another interesting research avenue of supporting alternate programming models of vectorisation in Haskell.

In closing we believe that high performance computing, by exploiting superword parallelism is very much possible in Haskell. The syntax and semantics of Haskell maps very naturally to the notions of data parallel programming. And if we can make the compiler and the code generator even more intelligent, we can rival the performance and speed of imperative languages while maintaining a declarative and elegant model of programming.

Bibliography

[Fly66] Michael J Flynn. “Very high-speed computing systems”. In: *Proceedings of the IEEE* 54.12 (1966), pp. 1901–1909.

[All70] Frances E Allen. “Control flow analysis”. In: *ACM Sigplan Notices*. Vol. 5. 7. ACM. 1970, pp. 1–19.

[Rey72] John C Reynolds. “Definitional interpreters for higher-order programming languages”. In: *Proceedings of the ACM annual conference-Volume 2*. ACM. 1972, pp. 717–740.

[Rey74] John C Reynolds. “Towards a theory of type structure”. In: *Programming Symposium*. Springer. 1974, pp. 408–425.

[Rus78] Richard M Russell. “The CRAY-1 computer system”. In: *Communications of the ACM* 21.1 (1978), pp. 63–72.

[Cla+80] TJW Clarke et al. “Skim-the s, k, i reduction machine”. In: *Proceedings of the 1980 ACM conference on LISP and functional programming*. ACM. 1980, pp. 128–135.

[FOW87] Jeanne Ferrante, Karl J Ottenstein, and Joe D Warren. “The program dependence graph and its use in optimization”. In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 9.3 (1987), pp. 319–349.

[RWZ88] Barry K Rosen, Mark N Wegman, and F Kenneth Zadeck. “Global value numbers and redundant computations”. In: *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM. 1988, pp. 12–27.

[Col89] Murray I Cole. *Algorithmic skeletons: structured management of parallel computation*. Pitman London, 1989.

[Cyt+89] Ron Cytron et al. “An efficient method of computing static single assignment form”. In: *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM. 1989, pp. 25–35.

[Hug89] John Hughes. “Why functional programming matters”. In: *The computer journal* 32.2 (1989), pp. 98–107.

[WB89] Philip Wadler and Stephen Blott. "How to make ad-hoc polymorphism less ad hoc". In: *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages.* ACM. 1989, pp. 60–76.

[Wad90] Philip Wadler. "Comprehending monads". In: *Proceedings of the 1990 ACM conference on LISP and functional programming.* ACM. 1990, pp. 61–78.

[Pug91] William Pugh. "The Omega test: a fast and practical integer programming algorithm for dependence analysis". In: *Supercomputing, 1991. Supercomputing'91. Proceedings of the 1991 ACM/IEEE Conference on.* IEEE. 1991, pp. 4–13.

[Wei91] Michael Weiss. "Strip mining on SIMD architectures". In: *Proceedings of the 5th international conference on Supercomputing.* ACM. 1991, pp. 234–243.

[CRM92] Ling-Yu Chuang, Vernon Rego, and Aditya Mathur. "An application of program unification to priority queue vectorization". In: *International Journal of Parallel Programming* 21.3 (1992), pp. 193–224.

[Jon92] Simon L Peyton Jones. "Implementing lazy functional languages on stock hardware: the Spineless Tagless G-machine". In: *Journal of functional programming* 2.2 (1992), pp. 127–202.

[Jon+93] SL Peyton Jones et al. "The Glasgow Haskell compiler: a technical overview". In: *Proc. UK Joint Framework for Information Technology (JFIT) Technical Conference.* Vol. 93. 1993.

[SF93] Amr Sabry and Matthias Felleisen. "Reasoning about programs in continuation-passing style". In: *Lisp and symbolic computation* 6.3-4 (1993), pp. 289–360.

[LP94] John Launchbury and Simon L Peyton Jones. "Lazy functional state threads". In: *ACM SIGPLAN Notices.* Vol. 29. 6. ACM. 1994, pp. 24–35.

[PW96] Alex Peleg and Uri Weiser. "MMX technology extension to the Intel architecture". In: *IEEE micro* 16.4 (1996), pp. 42–50.

[Tri+96] Philip W Trinder et al. "GUM: a portable parallel implementation of Haskell". In: *ACM SIGPLAN Notices.* Vol. 31. 5. ACM. 1996, pp. 79–88.

[AJ97] Andrew W Appel and Trevor Jim. "Shrinking lambda expressions in linear time". In: *Journal of Functional Programming* 7.5 (1997), pp. 515–540.

[Sab98] Amr Sabry. "What is a purely functional language?" In: *Journal of Functional Programming* 8.1 (1998), pp. 1–22.

[JRR99] Simon Peyton Jones, Norman Ramsey, and Fermin Reig. "C–: A portable assembly language that supports garbage collection". In: *International Conference on Principles and Practice of Declarative Programming*. Springer. 1999, pp. 1–28.

[Oka99] Chris Okasaki. *Purely functional data structures*. Cambridge University Press, 1999.

[Hin00] Ralf Hinze. "Generalizing generalized tries". In: *Journal of Functional Programming* 10.4 (2000), pp. 327–351.

[Jon00] Mark P Jones. "Type classes with functional dependencies". In: *European Symposium on Programming*. Springer. 2000, pp. 230–244.

[LA00] Samuel Larsen and Saman Amarasinghe. *Exploiting superword level parallelism with multimedia instruction sets*. Vol. 35. 5. ACM, 2000.

[FW01] Matthew Fluet and Stephen Weeks. "Contification using dominators". In: *ACM SIGPLAN Notices*. Vol. 36. 10. ACM. 2001, pp. 2–13.

[JTH01] Simon Peyton Jones, Andrew Tolmach, and Tony Hoare. "Playing by the rules: rewriting as a practical optimisation technique in GHC". In: *Haskell workshop*. Vol. 1. 2001, pp. 203–233.

[LV02] Ralf Lämmel and Joost Visser. "Typed combinators for generic traversal". In: *International Symposium on Practical Aspects of Declarative Languages*. Springer. 2002, pp. 137–154.

[Bol+03] Jeff Bolz et al. "Sparse matrix solvers on the GPU: conjugate gradients and multigrid". In: *ACM transactions on graphics (TOG)*. Vol. 22. 3. ACM. 2003, pp. 917–924.

[Jon03] Simon Peyton Jones. *Haskell 98 language and libraries: the revised report*. Cambridge University Press, 2003.

[Nai04] Dorit Naishlos. "Autovectorization in GCC". In: *Proceedings of the 2004 GCC Developers Summit*. 2004, pp. 105–118.

[Cha+05] Manuel MT Chakravarty et al. "Associated types with class". In: *ACM SIGPLAN Notices*. Vol. 40. 1. ACM. 2005, pp. 1–13.

[LOP05] Rita Loogen, Yolanda Ortega-Mallen, and Ricardo Pena-Mari. “Parallel functional programming in Eden”. In: *Journal of Functional Programming* 15.3 (2005), pp. 431–475.

[Wee06] Stephen Weeks. “Whole-program compilation in MLton”. In: *ML* 6 (2006), pp. 1–1.

[And+07] Todd Anderson et al. “Pillar: A parallel implementation language”. In: *International Workshop on Languages and Compilers for Parallel Computing.* Springer. 2007, pp. 141–155.

[Cha+07] Manuel MT Chakravarty et al. “Data Parallel Haskell: a status report”. In: *Proceedings of the 2007 workshop on Declarative aspects of multicore programming.* ACM. 2007, pp. 10–18.

[CLS07] Duncan Coutts, Roman Leshchinskiy, and Don Stewart. “Stream fusion: From lists to streams to nothing at all”. In: *ACM SIGPLAN Notices.* Vol. 42. 9. ACM. 2007, pp. 315–326.

[Fir+08] Nadeem Firasta et al. “Intel AVX: New frontiers in performance improvements and energy efficiency”. In: *Intel white paper* 19 (2008), p. 20.

[Hin08] Ralf Hinze. “Functional pearl: streams and unique fixed points”. In: *ACM Sigplan Notices.* Vol. 43. 9. ACM. 2008, pp. 189–200.

[NZ08] Dorit Nuzman and Ayal Zaks. “Outer-loop vectorization: revisited for short simd architectures”. In: *Proceedings of the 17th international conference on Parallel architectures and compilation techniques.* ACM. 2008, pp. 2–11.

[Pey+08] Simon Peyton Jones et al. “Harnessing the multicores: Nested data parallelism in Haskell”. In: *LIPIcs-Leibniz International Proceedings in Informatics.* Vol. 2. Schloss Dagstuhl-Leibniz-Zentrum fur Informatik. 2008.

[GH09] Andy Gill and Graham Hutton. “The worker/wrapper transformation”. In: *Journal of Functional Programming* 19.2 (2009), pp. 227–251.

[Kel+10] Gabriele Keller et al. “Regular, shape-polymorphic, parallel arrays in Haskell”. In: *ACM Sigplan Notices.* Vol. 45. 9. ACM. 2010, pp. 261–272.

[Mar+10] Simon Marlow et al. “Haskell 2010 language report”. In: *Available online http://www. haskell. org/(May 2011)* (2010).

[TC10] David A Terei and Manuel MT Chakravarty. "An LLVM backend for GHC". In: *ACM Sigplan Notices*. Vol. 45. 11. ACM. 2010, pp. 109–120.

[Fog+11] Agner Fog et al. "Instruction tables: Lists of instruction latencies, throughputs and micro-operation breakdowns for Intel, AMD and VIA CPUs". In: *Copenhagen University College of Engineering* 97 (2011), p. 114.

[BW12] Amy Brown and Greg Wilson. "The architecture of open source applications, volume ii". In: *Ebook, May* (2012).

[Cli12] Robert Clifton-Everest. "Optimisations for the LLVM back-end of the Glasgow Haskell Compiler". PhD thesis. Bachelors Thesis, Computer Science and Engineering Dept., The University of New South Wales, Sydney, Australia, 2012.

[Dei+12] M Deilmann et al. "A guide to vectorization with intel C++ compilers". In: *Intel Corporation, April* (2012).

[PM12] Matt Pharr and William R Mark. "ispc: A SPMD compiler for high-performance CPU programming". In: *Innovative Parallel Computing (InPar), 2012*. IEEE. 2012, pp. 1–13.

[Set12] Intel Xeon Phi Coprocessor Instruction Set. "Architecture Reference Manual". In: *Intel Corp., September* (2012).

[T12] Takenobu T. *GHC illustrated*. 2012. URL: `https://takenobu-hs.github.io/downloads/haskell_ghc_illustrated.pdf` (visited on 08/25/2018).

[dev13] GHC devs. *LLVM improved*. 2013. URL: `https://ghc.haskell.org/trac/ghc/wiki/ImprovedLLVMBackend` (visited on 08/25/2018).

[JGK13] Youngjoon Jo, Michael Goldfarb, and Milind Kulkarni. "Automatic vectorization of tree traversals". In: *Proceedings of the 22nd international conference on Parallel architectures and compilation techniques*. IEEE Press. 2013, pp. 363–374.

[Liu+13] Hai Liu et al. "The Intel labs Haskell research compiler". In: *ACM SIGPLAN Notices*. Vol. 48. 12. ACM. 2013, pp. 105–116.

[MLP13] Geoffrey Mainland, Roman Leshchinskiy, and Simon Peyton Jones. "Exploiting vector instructions with generalized stream fusio". In: *ACM SIGPLAN Notices*. Vol. 48. 9. ACM. 2013, pp. 37–48.

[POG13] Leaf Petersen, Dominic Orchard, and Neal Glew. "Automatic SIMD vectorization for Haskell". In: *ACM SIGPLAN Notices* 48.9 (2013), pp. 25–36.

[Gup+15] Suyog Gupta et al. "Deep learning with limited numerical precision". In: *International Conference on Machine Learning*. 2015, pp. 1737–1746.

[PJ15] Vasileios Porpodas and Timothy M Jones. "Throttling automatic vectorization: When less is more". In: *Parallel Architecture and Compilation (PACT), 2015 International Conference on*. IEEE. 2015, pp. 432–444.

[Hut16] Graham Hutton. *Programming in Haskell*. Cambridge University Press, 2016.

[EP17] Richard A Eisenberg and Simon Peyton Jones. "Levity polymorphism". In: *ACM SIGPLAN Notices*. Vol. 52. 6. ACM. 2017, pp. 525–539.

[Mau+17] Luke Maurer et al. "Compiling without continuations". In: *ACM SIGPLAN Notices*. Vol. 52. 6. ACM. 2017, pp. 482–494.

[Sar18] Abhiroop Sarkar. *lift-vector: Vector data types and polymorphic SIMD functions for Haskell*. `https://github.com/Abhiroop/lift-vector`. [Online; accessed 23-August-2018]. 2018.

[PCJ] Angela Pohl, Biagio Cosenza, and Ben Juurlink. "Correlating Cost with Performance in LLVM". In: ().

APPENDIX A
Intel vs AT&T syntax

The Intel vs AT&T syntax has been a common source of confusion in assembly programming and there has been a number of proposals to unify this two opposing syntaxes, but currently we deal with this non-uniformity across a number of documentation sources. The primary difference lies in the order of the operators.

We take a sample syntax:

```
VADDPS xmm1 xmm2 xmm3
```

The Intel syntax: The above instruction in Intel syntax implies the value at register `xmm2` and `xmm3` are added and the result is stored in `xmm1`.

AT&T syntax: The meaning of the snippet in AT&T syntax is that the content of the register `xmm1` and `xmm2` are added and the result is put in `xmm3`

For operators with more than two operands the simple rule of thumb is that the order of the operand in AT&T syntax is exactly the reverse of the order mentioned in Intel documentation. For this project, GHC currently uses GCC as an assembler and GCC understands the AT&T syntax. So all of the instructions emitted follow the AT&T notation.

APPENDIX B
Code contributions

This appendix serves to point an interested reader to all the sources of code that we have written.

B.1 Contributions to GHC

GHC uses the tool Phabricator for reviewing patches. So all of the code changes are immutable and they are fully available on Phabricator. The experimental changes which were not pushed on phabricator are made fully open sourced on github. All of the changes on phabricator has been code reviewed, with the code review history visible.

- SIMD support: `https://phabricator.haskell.org/D4813`
- Int16#, Word16# support: `https://phabricator.haskell.org/D5006`
- Int32#, Word32# support: `https://phabricator.haskell.org/D5032`
- Int64#, Word64# support: `https://github.com/Abhiroop/ghc-1/tree/wip-int64`
- base changes for lifted integer types: `https://github.com/Abhiroop/ghc-1/tree/wip-int8-redefine`
- SIMD integer support (experimental): `https://github.com/Abhiroop/ghc-1/tree/wip-int8-redefine`

B.2 The *Lift-vector* library

The *lift-vector* library provides polymorphic SIMD functions for vector programming. The entire library with examples and documentations is available here: `https://github.com/Abhiroop/lift-vector`

Although not a very reliable metric but the total lines of code written including rigorous tests were:

Approximate 1050 lines for each subword register support. For four patches: 1050 * 4 = 4200 loc. For SIMD support = 2100 loc. *lift-vector* = 829 loc. Total = 7129 loc.