Contents

G54REM Coursework II

Abhiroop Sarkar

May 2018

0.1 Introduction

In this literature review, we survey the state of vectorization in Haskell, the programming language. For our review we initially consider the Intel Haskell Research compiler [Liu+13] and secondly the most popular Haskell compiler - *GHC or the Glorious Glasgow Haskell Compilation System.* We study a general automatic vectorization algorithm in our first paper and then in the second paper we actually study how to leverage the support of vectorization from the compiler to model a representation of data structures, ideal for vector instructions usage.

The second paper provides a comprehensive set of benchmarks on a particular algorithm, which demonstrates that GHC can emit faster code than *GCC (the GNU C compiler)* in certain cases. The first paper also provides benchmarks showing that in certain cases GHC can be made even faster by extending the "core" language and intermediate representation of the compiler. Most of the above benchmarks study algorithms which are parallel in nature. We utilize the vector support provided by the underlying hardware to speed up our code. So let us start by understanding what exactly is *vectorization.*

0.2 Parallelism and Vectorization

In 1965, Gordon Moore in his seminal paper titled the *Moore's Law* [Moo65] observed "the number of transistors per square inch of a processor had doubled every 18 months". This law held true for years to come as a result of which, every new generation of processor by Intel, AMD and the other manufacturers increased their speed and performance at a very steady rate. However over the last couple of decades the size of the each transistor has been approaching such a minuscule number (the world's smallest transistor is 1 nm wide, compare that to the atomic size of silicon which is about 0.2 nanometers) that it is very clear that the Moore's law is finally going to approach its saturation.

As a result of which hardware manufacturers are investing in research in various other possibilities to speed up the processors. A natural progress was to embed multiple computing units(more commonly termed multi core) in a single processors and achieve parallelism through that. Another approach was to introduce CPU pipelining to ensure partial overlap in the execution of multiple instructions. There are multiple such strategies and as a result of which the various kinds of parallelism can be grouped into two major parts:

- Data Parallelism
- Instruction -level parallelism

While there are many more possible sub groupings of parallelism like task level parallelism, thread level parallelism etc that appear in various parallel computing literature, these are majorly examples of software based parallelism. Here we are mostly interested in hardware support for parallelism. In his highly

cited 1966 paper [Fly66], MJ Flynn introduced what is now known as "Flynn's Taxonomy" to classify various types of hardware. Its easier to demonstrate the Flynn's Taxonomy with the help of a figure:

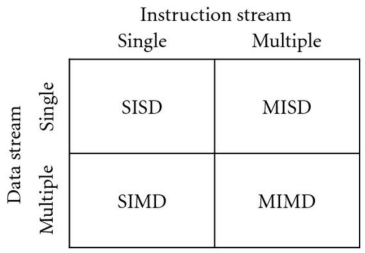


Figure 1: Flynn's Taxonomy

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 data which is commonly known as vector machine approach or *vectorization* and finally `MIMD` generally implies any other parallelism approach like pipelining, multi core processing etc.

For the purpose of this literature review we will be dealing with Single Instruction Multiple Data operations which henceforth shall be always referred by the acronym `SIMD` pronounced as "*seem-dee*".

0.3 Intel Vector Intrinsics

As mentioned in the previous section vectorization is an example of data level parallelism. Using SIMD operations we can execute the same instruction on multiple pieces of data at the same time. Let us take an example.

```
add.d r3, r1, r2

addvec.d v3, v1, v2
```

The first instruction is a plain `SISD` instruction. So imagine *r1* and *r2* are normal registers in a 64 bit machine holding an integer. In a 64 bit machine the

size of a pointer is 64 bits, however the size of an integer is generally 4 bytes or 32 bits. So the first instruction adds two 32 bit integers in *r1* and *r2* and stores the result in the register *r3*.

Now let us assume for the second instruction that it is from the Intel Nehalem set of microarchitectures which supports 16 byte width registers. So the registers *v1* and *v2* are `SIMD` registers of 16 byte or 128 bit width. This implies that each of these registers can hold four 32 bit integers. So the instruction *addvec.d* parallely operates on four pieces of data at the same time. The operation can be visualized(figure from Intel SIMD extensions technology page) something like this:

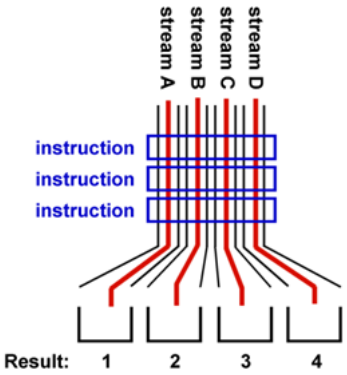


Figure 2: Intel vector intrinsics

As deomonstrated in the figure above the data is divided into streams of data and each instruction is fed from multiple streams. This streaming nature of the data gives the name *Streaming SIMD Extensions or SSE* to the first family of these instruction sets. Consequently Intel has provided support for multiple other operations like addition, subtraction, multiplication, shuffling, comparing, cryptographic etc. In the year 2008, Intel provided an additional set of even wider registers which supported 256 bit operations(i.e operating on eight 32 bit integers at the same time) with the Intel Sandy Bridge microarchitecture and provided a more modern and convenient set of vector APIs calling it *AVX or Advanced Vector Extensions*. In 2015 an even wider 512 bit register set was introduced in the Intel Xeon Phi range of micro-architectures which were targeted for usage in supercomputers and high performance servers. ARM also introduced a similar line of instruction set calling them the *ARM Neon* instruction set.

These vector instructions are used by a multitude of Fortran and C compilers to parallelize and speed up their programs. Major C compilers like GCC and clang as well as Fortran compilers like Intel Fortran compiler and GNU Fortran compiler provide full support for SIMD operations.

The term "compiler intrinsics" is used to define special built-in functions provided by the language compiler to undertake various specialized functions

like inlining, vectorization etc. As a part of my dissertation, I hope to add support for the same vector compiler intrinsics in the *Glasgow Haskell Compiler or GHC*. This would make Haskell the fifth *major* programming language (after C, C++, Fortran, D) to have full support for vectorization(Rust and Go have very experimental support).

In the following section of the paper we will be studying the Intel Haskell Research Compiler which experimented with and modified certain portions of the GHC compiler pipeline and intermediate language to attain automatic vectorization.

Chapter 1

Automatic SIMD Vectorization for Haskell by Petersen et al

1.1 Introduction to GHC

The paper that we are reviewing here [POG13], is innately connected to the *Glasgow Haskell Compiler*, as the frontend of the IHRC compiler(the Intel Haskell Research compiler referenced by its acronym from here onwards) is GHC itself. So let us briefly have a look at the compilation pipeline of GHC[Jon+93].

Figure 1.1: The GHC compilation pipeline

The above figure demonstrates the general compilation phases that a Haskell program goes through. We are not concerned with the entire pipeline. All Haskell programs are reduced to a variant of lambda calculus [Chu36] called System F [Rey74]. System F provides a small and compact core intermediate language for the vast Haskell language to compile down to.

The IHRC intercepts the System F core representation, as a result it gains the sophisticated optimization which are applied on GHC before compiling down to System F. Haskell being a purely functional language, emphasizes on *immutability*. As a result of which it generates a lot of intermediate structures and array. The first phase of compilation in GHC also attempts to eliminates as many intermediate arrays as possible which simplifies the job of the IHRC compiler.

1.2 The Intel Haskell Research Compiler

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 initialized before its use. For eg:

```
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 analyze. 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 optimizations focused on the latter. All general compiler optimizations 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.

1.3 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 optimization called *Stream Fusion* [CLS07].

1.3.1 Stream Fusion

Higher order functions initially turns the array into a stream based representation (which we define in depth in the second 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 initialized 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 utilizes 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 optimize 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 *initializing write*. According to the paper,

> The two invariants of initializing writes are that reading an array element must always follow the initializing write of that element, and that any element can only be initialized once. This style of write preserves the benefits of immutability from the compiler standpoint, since any initializing write to an element is the unique definition of that element.

As a result of the above two invariants, we can again work with immutable arrays in the IHRC compiler and try vectorizing loops which operates on these arrays. Loops generally comprises of a major part of the program. The purpose of the loops are to initialize these immutable arrays or perform reductions over them. We specifically target these loops and try to generate `SIMD` code for these loops.

1.4 The Vector Core Language

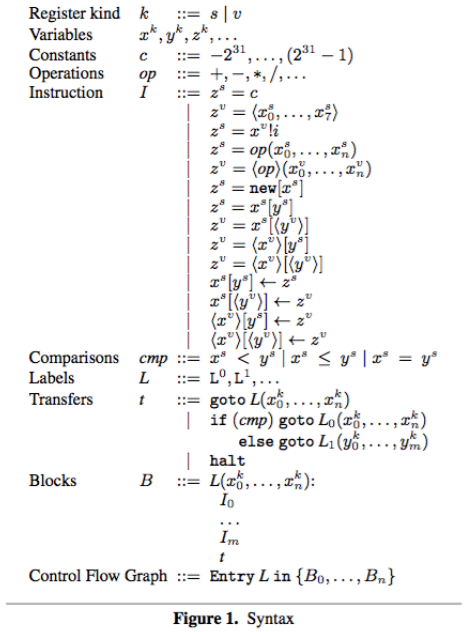
The entire syntax of the core language and operations of the compiler, as defined in the paper, is given in Figure 1.2

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 auto-vectorization algorithm 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 superscript `s` and `v` similarly stand for scalar and vector registers respectively.

The unique notion of immutability in the array types, denoted by $x^s[y^s]$ or $x^s[\langle y^v \rangle]$ etc, is emphasized by this note:

> It is an unchecked invariant of the language that every element of the array is initialized before it is read, and that every element of the array is initialized at most once. It is in this sense that the



Register kind	k	::=	$s \mid v$
Variables	$x^k, y^k, z^k, \ldots$		
Constants	c	::=	$-2^{31}, \ldots, (2^{31}-1)$
Operations	op	::=	$+, -, *, /, \ldots$
Instruction	I	::=	$z^s = c$
		|	$z^v = \langle x_0^s, \ldots, x_7^s \rangle$
		|	$z^s = x^v ! i$
		|	$z^s = op(x_0^s, \ldots, x_n^s)$
		|	$z^v = \langle op \rangle (x_0^v, \ldots, x_n^v)$
		|	$z^s = \texttt{new}[x^s]$
		|	$z^s = x^s[y^s]$
		|	$z^v = x^s[\langle y^v \rangle]$
		|	$z^v = \langle x^v \rangle [y^s]$
		|	$z^v = \langle x^v \rangle [\langle y^v \rangle]$
		|	$x^s[y^s] \leftarrow z^s$
		|	$x^s[\langle y^v \rangle] \leftarrow z^v$
		|	$\langle x^v \rangle [y^s] \leftarrow z^v$
		|	$\langle x^v \rangle [\langle y^v \rangle] \leftarrow z^v$
Comparisons	cmp	::=	$x^s < y^s \mid x^s \leq y^s \mid x^s = y^s$
Labels	L	::=	$\mathtt{L}^0, \mathtt{L}^1, \ldots$
Transfers	t	::=	$\texttt{goto}\ L(x_0^k, \ldots, x_n^k)$
		|	$\texttt{if}\ (cmp)\ \texttt{goto}\ L_0(x_0^k, \ldots, x_n^k)$ $\texttt{else goto}\ L_1(y_0^k, \ldots, y_m^k)$
		|	$\texttt{halt}$
Blocks	B	::=	$L(x_0^k, \ldots, x_n^k)$: I_0 $\ldots$ I_m t
Control Flow Graph		::=	$\texttt{Entry}\ L\ \texttt{in}\ \{B_0, \ldots, B_n\}$

Figure 1. Syntax

Figure 1.2: The Vector core language

> arrays are immutable—the write operation on arrays serves only as an initializing write but does not provide for 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 provide 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.

The description of an entire program is given at the bottom of the figure. It would contain 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 here is compared to the MLton

compiler [Wee06].

1.5 Automatic Vectorization in IHRC

To study vectorization in the IHRC compiler the paper takes an example program: computing the point-wise sum of two arrays b^s and c^s, each of the length l^s. A naive non-vectorized code for this program looks like this:

```
L⁰():
  a^s = new[l^s]
  i⁰_s = 0
  goto L¹(i⁰_s)

L¹(i^s):
  x^s = b^s[i^s]
  y^s = c^s[i^s]
  z^s = +(x^s, y^s)
  a^s[i^s] ← z^s
  i¹_s = +(i^s, 1)
  if (i¹_s < l^s) goto L¹(i¹_s)
          else goto L^exit(i¹_s)
```

Figure 1.3: The scalar code from the paper

As we can see above all of the register used are of type s which are scalar registers. The paper further introduces some terminologies as given below:

- Base induction variable: The entry variable in the entry block of the loop, such that the definition of the variable on the loop-back edge is defined from this variable itself.
- Step: The constant added each time around the loop
- Induction variable:
  - The base induction variable
  - A variable defined by $x^s = +(y^s, z^s)$, where y^s is an induction variable and z^s is loop invariant
  - A variable defined by $x^s = *(y^s, z^s)$, where y^s is an induction variable and z^s is a constant (defined by $z^s = c$)

As an example, in the following code:

```
int i, j;
for (i = 0; j = i * 2 ; i < 10; i ++){
...
}
```

$i = 0$ is the base induction variable, the step size is one and both j and i in every subsequent iteration is an induction variable.

A characteristic function of an induction variable i^s is defined as $i^s = s*\#+d$. Here d is the initial value of the induction variable, s the step function and # is

the iteration number. The compiler ensures that the numeric overflow semantics of the underlying types are respected.

A naive vectorization strategy would be to convert every scalar register to a vector format but consider this case: converting $x^s = b^s[i^s]$ to $x^v = \langle b^v \rangle [\langle i^v \rangle]$ would involve the redundancy that the variable $\langle b^v \rangle$ would just comprise eight copies of b^s. A better option would be $x^v = b^s[\langle i^v \rangle]$. A further optimization would be utilizing the knowledge that the step value in this loop is one and as mentioned in the previous section that certain hardware would have native support for vectors laid out in a uniform stride. So the instruction reduces to $x^v = b^s[\langle i^s :\rangle]$

Applying a similar vectorization strategy for the other array, the vectorization for the reduction operation is fairly simple as to just using the vector add instruction instead of the scalar add. Writing the final computation to the new array is again dual to the previous instructions that we read. It reduces to $a^s[\langle i^s :\rangle] \leftarrow z^v$ utilizing the contiguous write sequence.

The final instruction to vectorize is the $i_1^s = +(i^s, 1)$ instruction. While this looks deceptively simple, it is important to realize that there are three functionalities for this instruction: (1) to check whether to proceed with another iteration (2) to provide a new value for the induction variable (3) to provide the value passed on to the exit edge. For the last two cases it is sufficient enough to see if there are 8 such iterations (8 is the width of the SIMD register). For the loop exit test case, the paper defines the test as:

> in order to execute the vector loop again, we must have at least eight remaining iterations. Upon completion of a vector iteration computing scalar iterations j, . . . , j + 7, the question that must be answered then is whether the scalar loop would always execute iterations j + 8, . . . , j + 15. Because of the monotonic nature of the induction variable computation this in turn can be reduced to the question of whether or not the value of i_1^s at iteration j + 15 is less than l^s.

So in the case, as mentioned above, if there are not enough iterations remaining, we resort back to the scalar loop. And putting all of the ideas discussed above, the vectorized loop is given in Figure 1.4

The automatic vectorization algorithm translates each scalar instruction to compute a scalar value, a vector value and a last value(value generated at the end of the loop). As a result the code generated is highly verbose. However, the compiler depends on other orthogonal optimization passes like dead code elimination, common sub-expression elimination, loop invariant code motion [ASU86] to remove the verbosity.

The remaining vectorization section of the paper goes into intricate details of how each variable undergoes the vector transformation and all the checks associated with them, which is beyond the scope of this review.

Finally the paper talks about dependence analysis [Ban97] which is used to determine *when* it is possible to do a vectorization transformation. While this

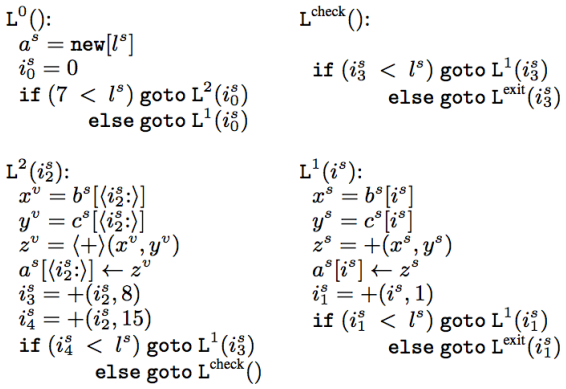


Figure 1.4: The vectorized code from the paper

a well known hard problem to solve, in the case of IHRC this becomes quite simple owing to dealing with immutable arrays. There are generally three types of dependences:

- Flow Dependence: When a read happens after a write
- Anti Dependence: When a write happens after a read
- Output Dependence: When a write happens after a write

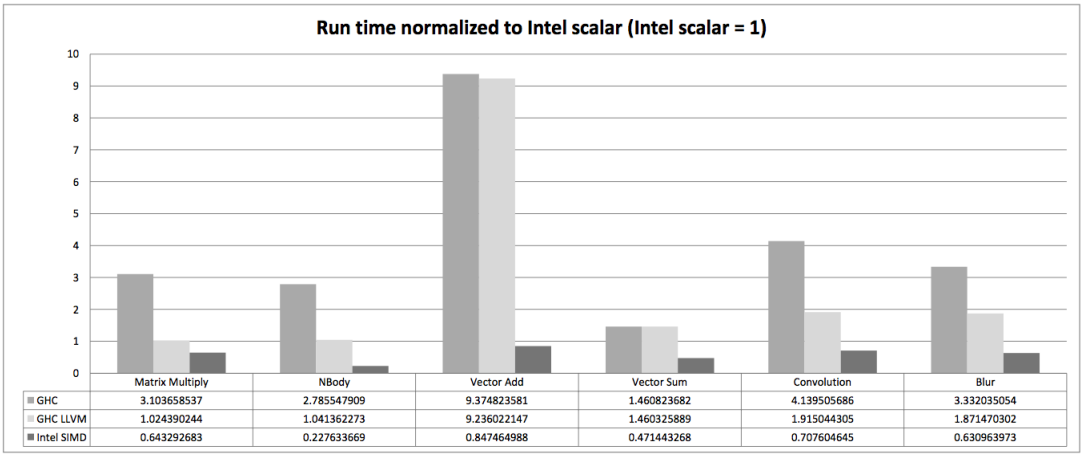
As we discussed in Section 1.3 of working with arrays, the concept of *initializing write* ensure that anti dependence can never occur. Output dependence implies mutation and as we are dealing with immutable arrays, that is also ruled out. So we only deal with flow(read after write) dependence in the IHRC compiler. And according to the paper:

> a very naive analysis can be quite successful at breaking read after write dependences in these style of loops.

1.6 Benchmarks

The benchmarks in the paper are carried out with an Intel Xeon E5-4650 (Sandybridge) processor supporting the AVX SIMD instuction set. The programs used are (1) pointwise sum of two arrays of 32 bit floating point numbers (2) horizontal sum of an array of 32 bit floating point number (3) an n-body simulation kernel (4) a regular matrix multiplication routine (5) a two dimensional five by five stencil convolution written using REPA (6) "blur" image processing example included in REPA

The comparison is done between the IHRC compiler, GHC with the native code generator as well as the GHC with the LLVM backend [TC10]. Benchmarks are demonstrated in the Figure 1.5



	Matrix Multiply	NBody	Vector Add	Vector Sum	Convolution	Blur
GHC	3.103658537	2.785547909	9.374823581	1.460823682	4.139505686	3.332035054
GHC LLVM	1.024390244	1.041362273	9.236022147	1.460325889	1.915044305	1.871470302
Intel SIMD	0.643292683	0.227633669	0.847464988	0.471443268	0.707604645	0.630963973

Figure 1.5: Benchmarks

As we can see from the figure, the IHRC compiler outperforms the sequential code by a vast margin as well as the LLVM code(which has some support for SIMD) by a decent margin. Future work would involve, more optimization passes to inline and simplify the code further as well as better array bound checks.

Chapter 2

Exploiting Vector Instructions with Generalized Stream Fusion by Mainland et al

2.1 Streams and Vector Instructions

In this paper, we study an application of vectorization. This builds on the previous paper of adding support for vectorization at the compiler level. This current paper doesn't speak at the level of the hardware or compiler, but rather the ideal representation of data structures in Haskell, to utilize the SIMD instructions.

The paper "Exploiting Vector Instructions with Generalized Stream Fusion" [MLP13] talks in detail about Stream Fusion [CLS07], something that we touched on very briefly in a Section 1.3.1.

As this literature review, is targeted to be read from a generalist's point of view, I introduce basic idioms in Haskell on the way. The paper in general doesn't use any fancy Haskell features except maybe associated type [Cha+05], which I try to explain in the given word limit. Interested readers are encouraged to read the cited paper for more details.

2.2 A very brief tour of Haskell

The most important aspect of Haskell we will be dealing with are ADTs or *algebraic data types*. The terminologies are best described by the Figure 2.1 from the book Learn you a Haskell for great good [Lip11].

While Figure 2.1 describes a simple ADT, there exists various types of ADTs as given in Figure 2.2

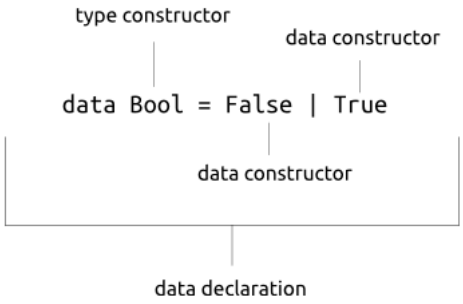


Figure 2.1: Algebraic Data Type

Enumeration:
data Season = Summer | Winter | Autumn | Spring

Product:
data Pair = Pair Int Int

Sum:
data Shape = Circle Float | Rect Float Float

Polymorfic & Recursive:
data Tree a = Leaf a | Node (Tree a) (Tree a)

Figure 2.2: Types of Algebraic Data Types

Apart from this the only other concept that we need are *typeclasses* [WB89] which is something akin to interfaces in object oriented languages but more powerful and *ad-hoc*. Eg:

```
class Eq a where
  (==) :: a -> a -> Bool
```

where the `a` implies polymorphic type, which means any type Int or Bool or anything else can be an instance of this typeclass.

2.3 Stream Fusion

In this section, we will look at the representation of streams and stream fusion in more detail. The reason for that is, most high performance array based libraries use a stream based representation to utilize this "stream fusion" optimization. And eventually the paper modifies the stream representation to utilize SIMD registers as well.

The main issue is that compilers are not good at optimizing 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.

```
data Stream a where
  Stream :: (s -> Step s a) -> s -> Int -> Stream a

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

The paper takes 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

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
```

As we can see above the `map` function now doesn't have a recursive structure, and what we have gained is, when we are performing composition of functions like $mapf \circ mapg$, we can substitute and see:

$mapf \circ mapg \equiv unstream \circ mapSf \circ stream \circ unstream \circ mapSg \circ stream.$

It is immediately obvious that $stream \circ unstream$ by the law of composition becomes identity. GHC supports something called *rewrite rules* [JTH01] to allow us to freely express these equations. Stream fusion utilizes this rule to eliminate intermediate structures.

2.4 Bulk memory operations and SIMD

As discussed over the course of this entire literature survey, SIMD register are 2 or 4 or 8 or even 16 element wide. They prefer to work with multiple elements at a time. If we look at the existing representation of the stream that is used, it *yields* one element at a time and operates on those. This representation is very bad for SIMD computations as well as bulk memory operations supported in the hardware like `memcpy` or `memset`. A classic example is the vector append operation which can easily utilize the `memcpy` function and also the replicate function which would simply involve calling `memset`.

We use the example of `dot product` to demonstrate the usability of bulk memory operations:

```
dotp :: Vector Double -> Vector Double -> Double
dotp v w = sum (zipWith (*) v w)
```

This is quite intuitive, we take two vectors multiply each of their components point-wise and then calculate the sum of those multiplications. Both the `zipWith` function as well as the `sum` function are targets for vectorization.

So the present representation of streams are not suitable for use in SIMD computations. We want a representation which yields multiple values at a time (preferably equal to the width of the SIMD register) and in the last bout when there are not enough elements it yields a scalar stream of values called the "dribble". There exists another representation and like most problems in computer science it is solved by adding another level of indirection.

2.5 Generalized Stream Fusion

The generalized stream fusion framework introduced in the paper, utilizes a representation which solves both the problem of bulk memory operations as well as utilizing SIMD operations. It represents a stream just as a *bundle* of stream.

```
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. So what does a `Chunk` represent?

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

The definition of `Chunk` while quite simple, contains some concepts which are unfortunately outside the scope of this review. Importantly, it utilizes 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 1.3.1 to optimize the process.

While the `Chunk` representation works great for *vector append*, it is not the best possible representation for operations like *zipWith, fold* etc for which we have the `sElems` field with the Stream representation. The next section is entirely devoted on deriving the appropriate type of the `sMultis` field, called `Multis`.

2.6 Working with SIMD computation

The paper uses the concept of *associated types* or *type families* [Cha+05] to represent a SIMD value. Associated types are a relatively advanced type level concept, and for some one very new to Haskell might be hard to grasp. But it is essential to understand the following sections. We try to explain it very briefly below.

> 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, and associated types provide exactly the support for that. A very motivating use case of associated types was first given by Ralf Hinze [Hin00] where the representation of the *generalized trie* depended on the type level.
>
> ```
> class MapKey k where
>   data Map k v  -- the Associated type
>   empty :: Map k v
>   lookup :: k -> Map k v -> v
> ```
>
> So depending on the instance the representation of the `Map` type would vary like this:
>
> ```
> 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
>
> ... etc
> ```

So given the description of the associated types above we can utilize them, to model the representation of SIMD types. Specifically we want an associated type `Multi` which would contain a short vector whose size is given by a *multiplicity* function which is the width of the SIMD register. We also define specific function definitions like `multimap, multifold` etc whose instances would operate on the required size of the SIMD vector.

```
class MultiType a where
  data Multi a -- Associated type
```

```
multiplicity :: Multi a -> Int
multienum :: Multi a
multireplicate :: a -> Multi a
multimap ::(a -> a) -> Multi a -> Multi a
multifold ::(b->a->b) -> b -> Multi a -> b
multizipWith :: (a -> a -> a) -> Multi a -> Multi a -> Multi a
```

The `Multi a` is the associated type which contains a vector (which would have the appropriate width) of elements. However, when we have an odd number of elements in the stream, there is an issue. We know the width is either 2, 4 or 8 (16 in case of AVX-512 supercomputers), so for an odd number of elements or a non multiple of any of these element size, would cause an issue. So the paper uses the `Either` type to encode either a `Multi a` or a simple scalar element `a`.

```
data Either a b = Left a | Right b
type MultisP a = Stream (Either a (Multi a))
```

It is named `MultisP`, the P because the *producer* chooses what will be yielded at each step.

This representation works very well for `sum, fold` or any kind of reducing operation however when we are dealing with zip operation a new issue arises. A zip operation can be visualized something like this:

```
zipWith (+) [1..10] [11..20]

1  2  3  4  5  6  7  8  9 10
+  +  +  +  +  +  +  +  +  +
11 12 13 14 15 16 17 18 19 20

[12,14,16,18,22,24,26,28,30]
```

It quite evident to notice that when we try to apply `zipWith` on a `MultisP` type if the producer 1 produces a `Multi a` and producer 2 produces a simple `a`, they cannot be zipped. As a result of which we have to invert the control and give the power to *consumer*, not the *producer*.

```
data MultisC a where
  MultisC :: (s -> Step s (Multi a))
          -> (s -> Step s a)
          -> s
          -> MultisC a
```

Again the full explanation of this type is beyond the scope of the review, but it provides the consumer control to either yield a value of type `Multi a` given by $(s \to Steps(Multia))$, or a value of type `a` given by $(s \to Stepsa)$.

Now all we have to do is combine `MutisP a` and `MultisC a` to produce a single type `Multis a` as defined in the original `Bundle` type.

```
type Multis a = Either (MultisC a) (MultisP a)
```

This representation gives us a chance to specialize our implementation function based on the type that is most suitable for the appropriate function. For something like `mZipWith` we use `MultisC` while for reduction operations we use `MultisP`.

2.7 Benchmark

The paper goes ahead to show the implementation of the dot product algorithm using the representation presented in the previous section. It provides benchmarks compared to hand written C code executed with plain GCC as well as GCC with support for SIMD instructions. Also competing is the Goto BLAS implementation [GV08] which is the fastest known implementation. The benchmark is ran on a 3.40GHz Intel i7-2600K processor, averaged over 100 runs.

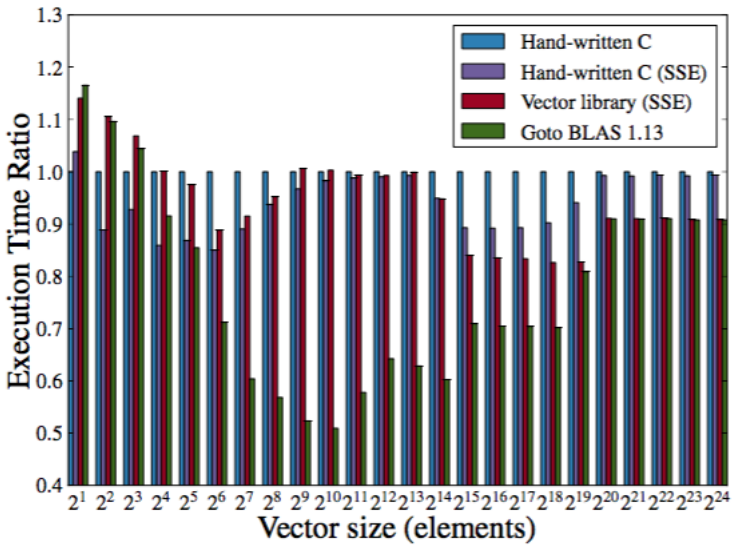


Figure 2.3: Benchmarks of GCC vs Goto BLAS vs GHC vs GCC with SIMD

2.8 Future Work and Conclusion

Over the span of this entire literature survey we saw how SIMD instructions allow us to leverage the parallelism support provided by the hardware. As a part of my summer thesis I will be extending the GHC native code generator

with support for SIMD instructions. There already exists the LLVM backend, which has some support for SIMD instruction, but there are certain other issues relating to register allocation in the LLVM backend, as a result of which it is being rewritten as well. The support for SIMD instructions would allow commercial and industrial users of Haskell to utilize these instructions to parallelize their code for free. We plan to provide basic support for the instructions first and later research on other avenues of auto-vectorization opportunities.

Bibliography

[Chu36] Alonzo Church. "A note on the Entscheidungsproblem". In: *The journal of symbolic logic* 1.1 (1936), pp. 40–41.

[Moo65] Gordon Moore. "Moore's law". In: *Electronics Magazine* 38.8 (1965), p. 114.

[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.

[ASU86] Alfred V Aho, Ravi Sethi, and Jeffrey D Ullman. "Compilers, Principles, Techniques". In: *Addison Wesley* 7.8 (1986), p. 9.

[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.

[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.

[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.

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

[Ban97] Utpal Banerjee. *Dependence analysis*. Vol. 3. Springer Science & Business Media, 1997.

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

[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.

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

[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.

[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.

[GV08] Kazushige Goto and Robert Van De Geijn. “High-performance implementation of the level-3 BLAS”. In: *ACM Transactions on Mathematical Software (TOMS)* 35.1 (2008), p. 4.

[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.

[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.

[Lip11] Miran Lipovaca. *Learn you a haskell for great good!: a beginner's guide*. no starch press, 2011.

[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.