Master Thesis

An LLVM Backend for Accelerate

Christian-Albrechts-Universität zu Kiel
Department of Computer Science
Programming Languages and Compiler Construction

student: **Timo von Holtz**

advised by: Priv.-Doz. Dr. Frank Huch
Assoc. Prof. Dr. Manuel M. T. Chakravarty

Kiel, July 21, 2014

Selbstständigkeitserklärung

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbstständig und nur unter Verwendung der angegebenen Literatur und Hilfsmittel angefertigt habe.

July 21, 2014

Timo von Holtz

Abstract

Implementing parallel computations is traditionally done in C or FORTRAN. This allows great control over the details of execution. Unfortunately, this also comes with the complexities of using a low-level language. To make this easier, Chakravarty et al. have developed Accelerate, an embedded array language for computations for high-performance computing in Haskell. The main execution target of this EDSL is GPUs via CUDA.

McDonell started working on an alternative implementation using LLVM. I contribute to this work by implementing a different approach to code-generation for LLVM using quasi-quotation. I use this quasi-quoter to implement the skeletons in the LLVM backend.

Todo list

Contents

List of Figures

List of Tables

1 Introduction

write introduction

2 Technologies

write intro to Technologies chapter

2.1 LLVM

LLVM[Lat02] is a compiler infrastructure written in C++. In contrast to GCC it is designed to be used as a library by compilers. Originally implemented for C and C++, the language-agnostic design (and the success) of LLVM has since spawned a wide variety of front ends: languages with compilers that use LLVM include ActionScript, Ada, D, Fortran, OpenGL Shading Language, Haskell, Java bytecode, Julia, Objective-C, Python, Ruby, Rust, Scala and C.

2.1.1 LLVM IR

LLVM defines it's own language to represent programs. It uses Static Single Assignment (SSA) form.[AWZ88; RWZ88] A program is said to be in SSA form if each of its variables is defined exactly once, and each use of a variable is dominated by that variable's definition. SSA form greatly simplifies many dataflow optimizations because only a single definition can reach a particular use of a value, and finding that definition is trivial.

To get idea of how this looks like in practice, let's look at an example. Figure 2.1 shows a simple C function to sum up the elements of an array. The corresponding LLVM code is shown in figure 2.2.

The first obvious difference is the lack of sophisticated control structures. In LLVM every function is divided into basic blocks. A basic block is a continuous stream of

```
1 int sum(int* a, int length) {
2   int x = 0;
3   for (int i=0;i<length;i++) {
4     x += a[i];
5   }
6   return x;
7 }
```

Figure 2.1: sum as a C function

```
1  define i32 @sum(i32* nocapture readonly %a, i32 %length) {
2  entry:
3    %cmp4 = icmp sgt i32 %length, 0
4    br i1 %cmp4, label %for.body, label %for.end
5
6  for.body:                                         ; preds = %entry, %
       for.body
7    %indvars.iv = phi i64 [ %indvars.iv.next, %for.body ], [ 0, %entry ]
8    %x.05 = phi i32 [ %add, %for.body ], [ 0, %entry ]
9    %arrayidx = getelementptr inbounds i32* %a, i64 %indvars.iv
10   %0 = load i32* %arrayidx, align 4
11   %add = add nsw i32 %0, %x.05
12   %indvars.iv.next = add nuw nsw i64 %indvars.iv, 1
13   %lftr.wideiv = trunc i64 %indvars.iv.next to i32
14   %exitcond = icmp eq i32 %lftr.wideiv, %length
15   br i1 %exitcond, label %for.end, label %for.body
16
17 for.end:                                          ; preds = %for.body, %
       entry
18   %x.0.lcssa = phi i32 [ 0, %entry ], [ %add, %for.body ]
19   ret i32 %x.0.lcssa
20 }
```

Figure 2.2: sum as a LLVM

instructions with a terminator at the end. Instructions are add, mult, call, ..., but also call. Terminators can either be used to jump to another block (branch) or return to the calling function.

To allow for dynamic control flow, there are Φ-nodes. These specify the the value of a new variable depending on what the last block was. The Φ-nodes in the SSA are represented with **phi**-instructions. In LLVM, these have to preceed every other instruction in a given basic block.

Types

The LLVM type system is one of the most important features of the intermediate representation. Being typed enables a number of optimizations to be performed directly, without having to do extra analyses on the side before the transformation.

Important Types are:

- **void**, which represents no value
- integers with specified length N: `iN`
- floating point numbers: **half, float, double, ...**
- pointers: `<type> *`
- function types: `<returntype> (<parameter list>)`

- vector types: `< <# elements> x <elementtype> >`
- array types: `[<# elements> x <elementtype>]`
- structure types: `{ <type list> }`

2.1.2 Vectorization

Modern CPUs all have SIMD units to execute an instruction on multiple datasets in parallel. Usind these units is easy with LLVM. All operations (**`add`**, **`fadd`**, **`sub`**, ...) can be used with vector arguments the same way as with scalar arguments.

To manually exploit this can be tricky however. LLVM has multiple strategies to fuse similar instructions or tight inner loops into vectorized code.[LLV14]

Loop Vectorizer

The Loop Vectorizer tries to vectorize tight inner loops. To get an idea of how these work, lets look at the example from figure 2.1. To build the sum, every element of the array is added to an accumulator. Since addition is associative and commutative, the additions can be reordered. Given a vector width of 2, the sum of $2 * n$ and $2 * n + 1$ are calculated in parallel and then added together. In addition to this, the vectorizer will also unroll the inner loop to make fewer jumps necessary. Figure 2.3 shows the corresponding llvm code without loop unroll, as this increases code size dramatically.

Apart from reductions, LLVM can also vectorize the following:

- Loops with unknown trip count
- Runtime Checks of Pointers
- Reductions
- Inductions
- If Conversion
- Pointer Induction Variables
- Reverse Iterators
- Scatter / Gather
- Vectorization of Mixed Types
- Global Structures Alias Analysis
- Vectorization of function calls
- Partial unrolling during vectorization

A detailed description can be found at `http://llvm.org/docs/Vectorizers.html`

```
1  define i32 @sum(i32* nocapture readonly %a, i32 %length) #0 {
2  entry:
3    %cmp4 = icmp sgt i32 %length, 0
4    br i1 %cmp4, label %for.body.preheader, label %for.end
5
6  for.body.preheader:                               ; preds = %entry
7    %0 = add i32 %length, -1
8    %1 = zext i32 %0 to i64
9    %2 = add i64 %1, 1
10   %end.idx = add i64 %1, 1
11   %n.vec = and i64 %2, 8589934590
12   %cmp.zero = icmp eq i64 %n.vec, 0
13   br i1 %cmp.zero, label %middle.block, label %vector.body
14
15 vector.body:                                       ; preds = %for.body.preheader, %
       vector.body
16   %index = phi i64 [ %index.next, %vector.body ], [ 0, %for.body.preheader ]
17   %vec.phi = phi <2 x i32> [ %5, %vector.body ], [ zeroinitializer, %for.body.preheader
         ]
18   %3 = getelementptr inbounds i32* %a, i64 %index
19   %4 = bitcast i32* %3 to <2 x i32>*
20   %wide.load = load <2 x i32>* %4, align 4
21   %5 = add nsw <2 x i32> %wide.load, %vec.phi
22   %index.next = add i64 %index, 2
23   %6 = icmp eq i64 %index.next, %n.vec
24   br i1 %6, label %middle.block, label %vector.body
25
26 middle.block:                                      ; preds = %vector.body, %
       for.body.preheader
27   %resume.val = phi i64 [ 0, %for.body.preheader ], [ %n.vec, %vector.body ]
28   %rdx.vec.exit.phi = phi <2 x i32> [ zeroinitializer, %for.body.preheader ], [ %5, %
         vector.body ]
29   %rdx.shuf = shufflevector <2 x i32> %rdx.vec.exit.phi, <2 x i32> undef, <2 x i32> <
         i32 1, i32 undef>
30   %bin.rdx = add <2 x i32> %rdx.vec.exit.phi, %rdx.shuf
31   %7 = extractelement <2 x i32> %bin.rdx, i32 0
32   %cmp.n = icmp eq i64 %end.idx, %resume.val
33   br i1 %cmp.n, label %for.end, label %for.body
34
35 for.body:                                          ; preds = %middle.block, %for.body
36   %indvars.iv = phi i64 [ %indvars.iv.next, %for.body ], [ %resume.val, %middle.block ]
37   %x.05 = phi i32 [ %add, %for.body ], [ %7, %middle.block ]
38   %arrayidx = getelementptr inbounds i32* %a, i64 %indvars.iv
39   %8 = load i32* %arrayidx, align 4
40   %add = add nsw i32 %8, %x.05
41   %indvars.iv.next = add nuw nsw i64 %indvars.iv, 1
42   %lftr.wideiv = trunc i64 %indvars.iv.next to i32
43   %exitcond = icmp eq i32 %lftr.wideiv, %length
44   br i1 %exitcond, label %for.end, label %for.body
45
46 for.end:                                           ; preds = %for.body, %middle.block, %
       entry
47   %x.0.lcssa = phi i32 [ 0, %entry ], [ %add, %for.body ], [ %7, %middle.block ]
48   ret i32 %x.0.lcssa
49 }
```

Figure 2.3: Vectorized Sum

```
1  define void @foo(double %a1, double %a2, double %b1, double %b2, double*
       %A) {
2  entry:
3    %add = fadd double %a1, %b1
4    %mul = fmul double %add, %a1
5    %div = fdiv double %mul, %b1
6    %mul1 = fmul double %b1, 5.000000e+01
7    %div2 = fdiv double %mul1, %a1
8    %add3 = fadd double %div, %div2
9    store double %add3, double* %A, align 8
10   %add4 = fadd double %a2, %b2
11   %mul5 = fmul double %add4, %a2
12   %div6 = fdiv double %mul5, %b2
13   %mul7 = fmul double %b2, 5.000000e+01
14   %div8 = fdiv double %mul7, %a2
15   %add9 = fadd double %div6, %div8
16   %arrayidx10 = getelementptr inbounds double* %A, i64 1
17   store double %add9, double* %arrayidx10, align 8
18   ret void
19 }
```

Figure 2.4: `foo` without SLP vectorization

SLP Vectorizer

The goal of SLP vectorization (a.k.a. superword-level parallelism) is to combine similar independent instructions into vector instructions. Memory accesses, arithmetic operations, comparison operations, PHI-nodes, can all be vectorized using this technique. For example, the following function performs very similar operations on its inputs (a1, b1) and (a2, b2).

```
1 void foo(double a1, double a2, double b1, double b2, double *A) {
2   A[0] = a1*(a1 + b1)/b1 + 50*b1/a1;
3   A[1] = a2*(a2 + b2)/b2 + 50*b2/a2;
4 }
```

The SLP vectorizer may combine these into vector operations. Figures 2.4 and 2.5 show the the corresponding LLVM before and after SLP vectorization.

Fast-Math Flags

In order for the Loop vectorizer to work, the operations involved need to be associative. When dealing with floating point numbers, the basic operations like **`fadd`** and **`fmul`** don't satisfy this condition. To vectorize the code regardless, LLVM has the notion of fast-math-flags. These tell the optimizer to assume certain properties that aren't true in general.

Available flags are[LLV]:

```
define void @foo(double %a1, double %a2, double %b1, double %b2, double*
    %A) {
entry:
  %0 = insertelement <2 x double> undef, double %a1, i32 0
  %1 = insertelement <2 x double> %0, double %a2, i32 1
  %2 = insertelement <2 x double> undef, double %b1, i32 0
  %3 = insertelement <2 x double> %2, double %b2, i32 1
  %4 = fadd <2 x double> %1, %3
  %5 = fmul <2 x double> %4, %1
  %6 = fdiv <2 x double> %5, %3
  %7 = fmul <2 x double> %3, <double 5.000000e+01, double 5.000000e+01>
  %8 = fdiv <2 x double> %7, %1
  %9 = fadd <2 x double> %6, %8
  %10 = bitcast double* %A to <2 x double>*
  store <2 x double> %9, <2 x double>* %10, align 8
  ret void
}
```

Figure 2.5: `foo` with SLP vectorization

nnan No NaNs - Allow optimizations to assume the arguments and result are not NaN. Such optimizations are required to retain defined behavior over NaNs, but the value of the result is undefined.

ninf No Infs - Allow optimizations to assume the arguments and result are not +/-Inf. Such optimizations are required to retain defined behavior over +/-Inf, but the value of the result is undefined.

nsz No Signed Zeros - Allow optimizations to treat the sign of a zero argument or result as insignificant.

arcp Allow Reciprocal - Allow optimizations to use the reciprocal of an argument rather than perform division.

fast Fast - Allow algebraically equivalent transformations that may dramatically change results in floating point (e.g. reassociate). This flag implies all the others.

2.1.3 llvm-general

There are multiple bindings to LLVM in Haskell. The most complete is llvm-general.[Sca13] Instead of exposing the LLVM API directly, it uses an ADT to represent LLVM IR. This can then be translated into the corresponding C++ object. It also supports the other way, transforming the object back into the ADT.

Using an ADT instead of writing directly to the C++ object has many advantages. This way, the code can be produced and manipulated outside the IO context. It is also much easier to reason about within Haskell.

llvm-general also supports optimization and jit-compilation. The optimization is controlled by a `PassSetSpec` This offers the user 2 different options: `CuratedPassSetSpec` and `PassSetSpec`. Using `CuratedPassSetSpec` is the easier option of the 2. It offers a similar level of control as specifying `-On` at the command line. Using `PassSetSpec` gives much more control over the exact passes run, but you have to specify them one by one. Both of these specs also come with fields to specify information about the target.

2.2 Accelerate

Accelerate[Cha+11; McD+13] is an an embedded array language for computations for high-performance computing in Haskell. Computations on multi-dimensional, regular arrays are expressed in the form of parameterised collective operations, such as maps, reductions, and permutations. It is very similar to Repa[Kel+10] in the way computations are specified. The difference is in the way the computation gets evaluated. While Repa produces its result immediately, Accelerate collects the computation which can then be executed. This approach is necessaray, as Accelerate isn't limited to be run on a CPU. The main back-end of accelerate is in fact based on CUDA.

2.2.1 Usage

To give an example of how to use Accelerate, lets look at the dot product of 2 vectors. This could easily implemented in Haskell like this.

```
dotp_list :: [Float] -> [Float] -> Float
dotp_list xs ys = foldl (+) 0 (zipWith (*) xs ys)
```

You can write nearly exactly the same function in Accelerate:

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

The first is the difference is the use of a different datastructure. A `Vector` is an `Array` with one dimension. Similarly, a `Scalar` is an `Array` with 0 dimensions. These are then wrapped in the `Acc` type. `Acc a` is a computation, that yields an `a`.

The second difference is the `fold`. Unlike the `foldl` in the Haskell example, this `fold` doesn't specify the order of operations. This is important as it allows for an efficient implementation. In order of this to work, the operation passed to `fold` has to be associative.

If we want to call this function, we have to somehow produce values of type `Acc (Vector Float)`.

First the list is converted with `fromList`. To see what it does, lets first look at its type.

```
fromList :: (Elt e, Shape sh) => sh -> [e] -> Array sh e
```

In addition to the list of elements, `fromList` takes the shape of the resulting `Array`. Shapes can have an arbitrary amount of dimensions, but in this case it is only one. To construct a shape, there are 2 datatypes:

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

The head in this case represents the innermost dimension. The tail has to also be a shape.

With this, we get

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

and

```
type Scalar a = Array DIM0 a
type Vector a = Array DIM1 a
```

To lift the Arrays we get via `fromList` we use the function

```
use :: Arrays arrays => arrays -> Acc arrays
```

This is all we need to call `dotp`.

```
dotp' :: [Float] -> [Float] -> Acc (Scalar Float)
dotp' xs ys =
  let  xs' = use (fromList (Z :.length xs) xs)
       ys' = use (fromList (Z :.length xs) ys)
  in dotp xs' ys'
```

But we still only have the computation. To get the result out of the computation, we have to `run` it. After that we can convert the `Array` back to a list and extract the element.

```
dotp_list1 :: [Float] -> [Float] -> Float
dotp_list1 xs ys = head . toList $ run $ (dotp' xs ys)
```

2.2.2 Representation

Accelerate represents its arrays internally as regular unboxed one-dimensional arrays. In addition to this, the lengths of each dimension is stored. In case of a matrix, this would be height and width.

Rather than just simple types like `Int` or `Float`, Accelerate also supports tuples as elements These cannot be easily stored in an unboxed array. Instead, arrays of tuples are stored as tuples of arrays.

```
Vector (Int, Float) ~ (Vector Int, Vector Float)
```

2.2.3 Skeletons

To execute any computation on a given architecture, the involved functions like `fold` or `map` have to be implemented. This is easy if the target is Haskell. The reason however that it is easy is the fact, that Haskell has support for higher-level functions like `fold` and `map`. In C, this can be emulated by using function pointers. This approach however is not very performant if the function is relatively simple as it involves multiple jumps and the manipulation of the call stack.

Accelerate uses skeletons instead. The idea is to write the function as you would normally, but leave blanks for the concrete types and passed function. When needed it can then be instantiated with the correct types and function. Figure 2.6 illustrates the idea using `map`.

Instead of the single array for input and output however, there will often be multiple. This is because a singe array in Accelerate be represented by multiple arrays internally.

2.2.4 Fusion

Consider the following example from earlier.

```
dotp xs ys = fold (+) 0 (zipWith (*) xs ys)
```

```
1  void map
2  (
3    TyOut *            d_out,
4    const TyIn *       d_in,
5    const int          length
6  ){
7    for (int i=0; i < length; ix += 1) {
8      d_out[i] = apply(d_in[i]);
9    }
10 }
```

Figure 2.6: `map` Skeleton in C

This can be rewritten as

```
dotp xs ys =
  let zs = zipWith (*) xs ys
  in fold (+) 0 zs
```

First, the elements of the 2 arrays are combined pair-wise using `zipWith` and stored in a temporary array. As a second step, these elements are then combined into a `Scalar` using `fold`. Although this works perfectly well, it produces an intermediate array. This means additional writes and reads from memory. Considering that memory access is very costly this should better be avoided.

The solution is to delay the computation of the intermediate array. This means, that instead of computing the array in memory, it is done on-the-fly. This changes the construction of the skeletons somewhat. Instead of declaring the inputs directly, the arrays needed are passed in as an array environment. This is then accessed through a function `delayedLinearIndex`.

2.2.5 LLVM Backend

Apart from the CUDA-backend and the Interpreter, There are a number of incomplete backends for Accelerate. One of those that looks promising is the LLVM backend.[McD14] It uses llvm-general to bind to the LLVM API. It supports both CPUs as well as NVIDIA GPUs through PTX. In contrast to the CUDA backend, this has the advantage that no external program has to be called for compilation of the kernels. This is also true for the CPU side. Unfortunately, it is not possible to "write once, run everywhere" with LLVM. Although the instructions are exactly the same between architectures, the supplied libraries are not. That is why the Skeletons cannot be shared between the 2.

To run a computation in the LLVM native backend, the skeletons are first optimized and compiled to a callable function. Similar to repa, it uses uses gang workers[Cha+07]

for execution. These are multiple threads, typically as many as there are capabilities available. They wait on an `MVar` for IO actions to perform. At first, the load is split evenly between the workers. If a thread finishes early however, it looks for more work. If it finds that another thread has excess work available, it will steal half of it. This process is called work stealing. On a shared memory architecture, this has very little overhead.

In this thesis, I will mainly work on the native backend, but the results should be transferable to the PTX backend as well.

2.3 Template Haskell

Template Haskell is an extension to Haskell that allows you to do type-safe compile-time meta-programming, with Haskell both as the manipulating language and the language being manipulated. Pre-7.8, one would get a Template Haskell expression using quasiquoters like this.

```
foo :: Q Exp
foo = [|Just 5|]
```

It is not clear which type of expression it is. Another nasty side-effect of this is that the expression is not checked to be consistent internally. That leads to something like this being legal.

```
bar :: Q Exp
bar = [|case 1 of
          Nothing -> 42
          True    -> ()|]
```

Apart from being syntactically correct, this makes no sense at all. With GHC 7.8, a new form of quoting Haskell code was introduced.

```
foo :: Q (TExp (Maybe Int))
foo = [||Just 5||]
```

The code is now typechecked along with the rest of the code in the module. This means it is no longer possible to easily produce nearly untraceable type errors with Template Haskell. The structure of the expressions remains unchanged however.

```
newtype TExp a = TExp { unType :: Exp }
```

This means it is possible to use typed expressions in places where untyped an untyped expression is exprected. The reverse is also true, but type safety is lost in the process.

```
unsafeTExpCoerce :: Q Exp -> Q (TExp a)
```

usage and stage restrictions

3 Contributions

write intro to contribution chapter

3.1 accelerate

While testing some of the skeletons, in particular the first stage of `scanl`, I had trouble with lost writes. These happened only if multiple threads wrote to adjacent memory locations. Normally this behaviour should be prevented by cache coherency protocols. These, however can be disabled if the memory involved is believed to be constant.

The Internal representation of `Arrays` in Accelerate uses a `UArray` to store it's data. This type of Array is uses a `ByteArray#`, which is assumed to be constant. Switching the representation from `UArray` to `StorableArray` fixed the problem.

3.2 llvm-general

llvm-general offers a nice set of bindings to the LLVM API. It is not feature-complete however. One area where I could make an improvement on this is the optimization. The sole difference between the `PassSetSpect` and `CuratedPassSetSpec` should be the way they define passes. The `CuratedPassSetSpec` however was lacking important fields for data layout and target machine.[1] Also, they were mostly ignored when supplied to the `PassSetSpec`.[2] In the process, I also added support for loop and slp vectorization to `CuratedPassSetSpec`.

Another issue was the lack of support for fast-math flags.[3] They are now supported with the following datatype:

```
data FastMathFlags
  = NoFastMathFlags
  | UnsafeAlgebra
  | FastMathFlags {
      noNaNs :: Bool,
```

[1] https://github.com/bscarlet/llvm-general/pull/101
[2] https://github.com/bscarlet/llvm-general/issues/91
[3] https://github.com/bscarlet/llvm-general/issues/90

```
    noInfs :: Bool,
    noSignedZeros :: Bool,
    allowReciprocal :: Bool
  }
```

3.3 llvm-general-quote

When writing a compiler using LLVM in Haskell there is a good tutorial on how to do it at `http://www.stephendiehl.com/llvm/`. It uses *llvm-general* to interface with LLVM. The Idea followed in this tutorial is to use a monadic generator to produce the AST. The goal of monadic code generation is to use the state of the monad to store the instructions.

Figure 3.1 shows how to implement a simple for loop using monadic generators.

Unfortunately, this produces a lot of boilerplate code. We have to define the basic blocks manually and add the instructions one by one. This has some obvious drawbacks, as the code can get unreadable pretty quickly.

A more clean approach is to use a complete EDSL. The idea here is to use block structure to specify the individual elements. An implementation of this idea is *llvm-general-typed*. It actually goes further, as it also typechecks the produced code.

I propose a third approach using quasiquotation[Mai07]. The idea behind quasiquotation is, that you can define a DSL with arbitrary syntax, which you can then directly transform into Haskell data structures. This is done at compile-time, so you get the same type safety as writing the AST by hand.

Figure 3.3 shows how to implement a simple function in LLVM using quasiquotation. Compared to the other 2 solutions, this has already the advantage of being very close to the produced LLVM IR.

Without the ability to reference Haskell variables, this would be fairly useless in most cases. But quasiquotation allows for antiquotation as well. This means you can still reference arbitrary Haskell variables from within the quotation. Using this the following are equivalent:

- `[llinstr| add i64 %x, 1 |]`
- `let y = 1 in [llinstr| add i64 %x, $opr:y |]`

```
for :: Type                                  -- type of the index
    -> Operand                               -- starting index
    -> (Operand -> CodeGen Operand)          -- loop test to keep going
    -> (Operand -> CodeGen Operand)          -- increment the index
    -> (Operand -> CodeGen ())               -- body of the loop
    -> CodeGen ()
for ti start test incr body = do
  loop  <- newBlock "for.top"
  exit  <- newBlock "for.exit"

  -- entry test
  c    <- test start
  top  <- cbr c loop exit

  -- Main loop
  setBlock loop
  c_i <- freshName
  let i = local c_i

  body i

  i'   <- incr i
  c'   <- test i'
  bot  <- cbr c' loop exit
  _    <- phi loop c_i ti [(i',bot), (start,top)]

  setBlock exit
```

Figure 3.1: Monadic generation of for loop

```
1  define i32 @foo(i32 %x) #0 {
2  entry:
3    br label %for.cond
4
5  for.cond:                                         ; preds = %for.inc, %
       entry
6    %res.0 = phi i32 [ 0, %entry ], [ %add, %for.inc ]
7    %i.0 = phi i32 [ 0, %entry ], [ %inc, %for.inc ]
8    %cmp = icmp slt i32 %i.0, %x
9    br i1 %cmp, label %for.body, label %for.end
10
11 for.body:                                         ; preds = %for.cond
12   %add = add nsw i32 %res.0, %x
13   %inc = add nsw i32 %i.0, 1
14   br label %for.cond
15
16 for.end:                                          ; preds = %for.cond
17   ret i32 %res.0
18 }
```

3.3.1 Control Structures

You still have to specify the Φ-nodes by hand. This is fairly straight forward in a simple example, but can get more complicated very quickly.

The real goal is a language that "feels" like a high-level language, but can be trivially translated into LLVM. This means

- using LLVM instructions unmodified.
- introduce higher-level control structures like for, while and if-then-else.

Figure 3.2 shows a for loop using this approach. The loop-variable (%x in this case) is specified explicitly and can be referenced inside and after the for-loop. To update the %x, I overload the return statement to specify which value should be propagated. At the end of the for-loop the code automatically jumps to the next block. Figure 3.3 shows the produced LLVM code. This is clearly more readable.

3.3.2 SSA

The above approach unfortunately only works for some well-defined cases. With multiple loop-variables for example, it quickly becomes cluttered. On top of this, it relies on some "magic" to resolve the translation and legibility suffers.

The main reason why it is not possible to define a better syntax is that LLVM code has to be in SSA form. This means, that

1. every variable name can only be written to once.

```
[llg|
define i64 @foo(i64 %start, i64 %end) {
  entry:
    br label %for

  for:
    for i64 %i in %start to %end with i64 [0,%entry] as %x {
        %y = add i64 %i, %x
        ret i64 %y
    }

  exit:
    ret i64 %y
}
|]
```

Figure 3.2: For Loop using *llvm-general-quote*

```
define i64 @foo(i64 %start, i64 %end) {
entry:
  br label %for

for:                                              ; preds = %for.body, %entry
  %i = phi i64 [ %i.new, %for.body ], [ %start, %entry ]
  %x = phi i64 [ %y, %for.body ], [ 0, %entry ]
  %for.cond = icmp ule i64 %i, %end
  %i.new = add nuw nsw i64 %i, 1
  br i1 %for.cond, label %for.body, label %for.end

for.body:                                         ; preds = %for
  %y = add i64 %i, %x
  br label %for

for.end:                                          ; preds = %for
  ret i64 %x
}
```

Figure 3.3: Expanded For Loop

```
1  [llg|
2  define i64 @foo(i64 %start, i64 %end) {
3    entry:
4      %x = i64 0
5
6    for:
7      for i64 %i in %start to %end {
8          %x = add i64 %i, %x
9      }
10
11   exit:
12     ret i64 %x
13 }
14 |]
```

Figure 3.4: For Loop using *llvm-general-quote* and SSA

2. Φ-nodes are necessary where control flow merges.

To loosen this restriction, I implemented SSA recovery pass as part of the quasiquoter. This means that I can now reassign variables inside the LLVM code. Using this, I was able to define a much more simple syntax for the `for`-loop. The placement of Φ-nodes is not optimal, but redundant ones can be easily remove by LLVM's InstCombine pass. Figure 3.4 and 3.5 show the quoted and the produced code after InstCombine respectively. The code produced by the for loop using SSA recovery is not identical to the code other code, but it is equivalent. In the new approach, the new loop counter is calculated at the end, whereas it was calculated in the loop head before.

3.3.3 syntax extensions

llvm-general-quote supports all llvm instructions. For better usage however, I added control structures like the for loop discussed earlier. These are the additions I made to the syntax:

- direct assignment: `<var> = <ty> <val>`
- if: `if <cond> { <instructions> } [ else { <instructions> } ]`
- while: `while <cond> { <instructions> }`
- for: `for <ty> <var> in <val1> ( to | downto )<val2> { <instructions> }`

```
1  define i64 @foo(i64 %start, i64 %end) {
2  entry:
3    br label %for.head
4
5  for.head:                                         ; preds = %n0, %entry
6    %x.12 = phi i64 [ 0, %entry ], [ %x.6, %n0 ]
7    %i.4 = phi i64 [ %start, %entry ], [ %i.9, %n0 ]
8    %for.cond.3 = icmp slt i64 %i.4, %end
9    br i1 %for.cond.3, label %n0, label %for.end
10
11 n0:                                               ; preds = %for.head
12   %x.6 = add i64 %i.4, %x.12
13   %i.9 = add nuw nsw i64 %i.4, 1
14   br label %for.head
15
16 for.end:                                          ; preds = %for.head
17   ret i64 %x.12
18 }
```

Figure 3.5: Expanded For Loop (SSA)

4 Implementation

write intro to Implementation chapter

4.1 Quasiquoter

A quasiquoter basically works like a mini compiler. You get a string as input and have to generate Haskell code out of that. In it's simplest form this would be just a parser. But it is not limited to that, since all you need to define for a quasiquoter is a function

```
quoteExp :: String -> Q Exp
```

The `Q Exp` in this case is defined by Template Haskell. Since it uses Template Haskell to construct expressions, it is possible to reference variables in scope using antiquotation. Figure 4.1 shows a the design of a typical quasiquoter.

The design of *llvm-general-quote*, the quasiquoter I defined for LLVM, is inspired by *language-c-quote*, which is also used in the cuda implementation of Accelerate. This means I use *Alex* to specify the lexer and "Happy" for the parser. The goal of this quasiquoter is to generate an AST as defined by llvm-general-pure. If I want to do more than just parse complete LLVM code, I can't use this as a target for the parser. This means it is necessary to copy almost the whole AST and add additional constructors for antiquotation etc. This step wasn't necessary in *language-c-quote* since it's target datastructure still has constructors for antiquotation.

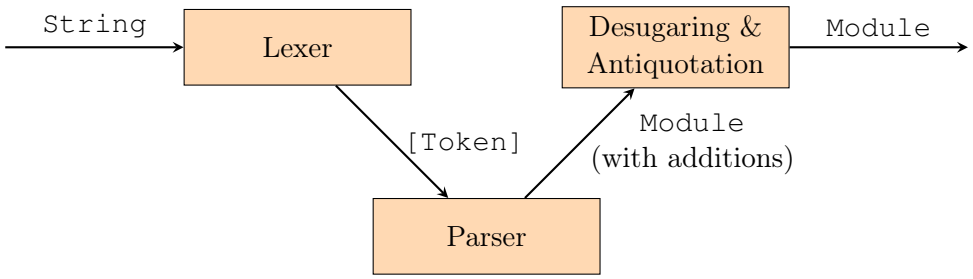

Figure 4.1: flowchart quasiquoter

At this point the construction of the 2 quasiquoters differs significantly. *language-c-quote* assumes that constructors stay the same, as long as there isn't a different case available. It does this by using the function

```
dataToExpQ :: Data a => (forall b. Data b => b -> Maybe (Q Exp)) -> a ->
```

This functions converts anything that is of typeclass `Data` into a Template Haskell expression. In addition to this however, it also takes an extra function to handle exceptions. This gets executed on every subterm recursively. If it returns `Nothing`, then the term gets translated into an expression directly. It it returns `Just exp`, then `exp` is used instead. The transformation function is then glued together as

```
qqExp :: Typeable a => a -> Maybe (Q Exp)
qqExp = const Nothing  `extQ` qqStringE
                       `extQ` qqIdE
...
```

Since I didn't use the same datatype for input and output, I took a different approach. I created the following typeclass.

```
type Conversion a b = forall m.(CodeGenMonad m) => a -> Q (TExp (m b))

class QQExp a b where
  qqExpM :: Conversion a b
  qqExp :: a -> TExpQ b
  qqExp x =
    [||let s = ((0,M.empty) :: (Int,M.Map L.Name [L.Operand]))
       in fst runState $$(qqExpM x) s||]
```

The `CodegenMonad` is important mostly to supply new names to unnamed basic blocks. It also provides an iterface to antiquote a list of basicblocks produced by monadic code generation. This is necessary to interface with existing code in accelerate-llvm.

```
class (Applicative m, Monad m) => CodeGenMonad m where
  newVariable    :: m L.Name
  exec           :: m () -> m [L.BasicBlock]
```

The `Applicative` context is not necessary since `Monad` is strictly more powerful than `Applicative`.[1] I decided to include it however since it allows for a more concise syntax. Using a typeclass instead of a single function has multiple advantages. When there is

[1] With GHC 7.10, `Applicative` will become a superclass of `Monad`. In the meantime, all instances of `Monad` should be adapted to also provide an instance of `Applicative`. Details can be found at `http://www.haskell.org/haskellwiki/Functor-Applicative-Monad_Proposal`.

a case missing, it will complain when the quoter is compiled instead of when it is used. This approach relies on typed expressions, which were introduced with GHC 7.8.

Here is a simple example of an instance declaration of `QQExp`.

```
instance QQExp A.Module L.Module where
  qqExpM (A.Module n dl tt ds) =
    [||L.Module <$> $$(qqExpM n) <*> $$(qqExpM dl)
                <*> $$(qqExpM tt) <*> $$(qqExpM ds)||]
```

In this case the structure of both the source and target constructors are the same.[2] Antiquotations have to be handled a little different. Since there can't be any information on the antiquoted expression, it has no specific type. To use it in a typed expression anyway, it has to be coerced.

```
instance QQExp A.Operand L.Operand where
  qqExpM (A.AntiOperand s) =
    [||$$(unsafeTExpCoerce $ antiVarE s)||]
```

Here I decided to first produce the typed expression and then splice it into a quoted expression. This is mostly a cosmetic choice, as the same code could be produced without doing this round trip.

toInteger etc.

4.2 Control Structures

When implementing the control structures, I had to decide on what level I wanted to introduce them. In a traditional procedural language like C, they would sit alongside the other expressions like assignments or function calls. This would correspond to the instructions in LLVM. For this to be possible however, I need to be able to extend them into just a sequence of instructions. In case of an if-then-else this is still kind of possible using select to get the result value. The loop structures however are not possible, as they require an arbitrary number of jumps.

[2] Instances like this could be generated using a Template Haskell function. I decided against using Template Haskell here, since this would make the code harder to debug.

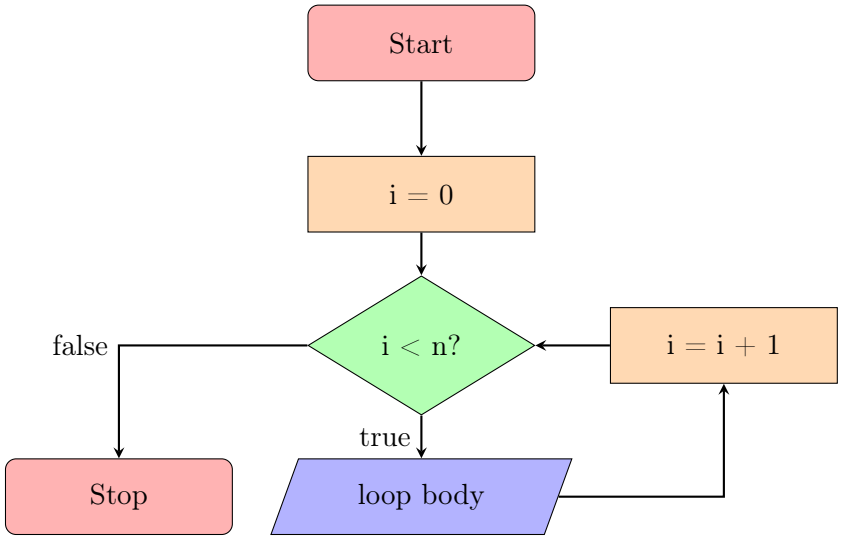

Figure 4.2: flowchart for-loop

4.2.1 direct approach

The solution is to have the control structures on the level of basic blocks. This way it is possible to just produce multiple basic blocks. For something to behave like a basic block, it must have all the elements of a basic block. In llvm-general, a basic block is represented as

```
data BasicBlock =
  BasicBlock Name [Named Instruction] (Named Terminator)
```

It consists of a name, a list of instructions and a terminator. The name is used as a target label for jumps. The terminator can either be a jump to a different block or a return statement. The instructions are just a stream without any jumps, but can also contain function calls.

Lets look at a classic for loop. It starts by initializing a counter. Then the counter is checked against a maximum. If this is not yet reached, the loop body gets executed and the counter incremented. If it is, then the loop exits and the next expression is evaluated. Figure 4.2 shows the process.

To translate this into LLVM, I need a label, the name of the counter, it's minimum and maximum value and the loop body. In a normal language these would be sufficient. LLVM doesn't have mutable variables, however. This means that any state, like an accumulator, needs to be explicitly defined. The syntax I chose for this is the following:

```
for <ty1> <var1> in <val1> to <val2> with <ty2> <values> as <var2>,
          label <jumptarget> { <loop body> }
```

The values here work the same way as with **phi**-instructions. It is often necessary to nest loops. To allow for this, the loop body can consist of multiple basic blocks. This way, it works much like function body, with the loop counter and the accumulator as arguments. With this analogy, I reuse the return statement to indicate the end of the loop. The value returned is used as the new value of the accumulator.

examples

When implementing this, there are a few things that make matters non-trivial. Since LLVM doesn't have mutable variables, it is necessary to introduce **phi** instructions manually. This node has to have a value specified for every incoming block. First these are the blocks and values specified in the loop header. On top this, these are all the blocks inside the loop returning a value. The values are then extracted and appended to the existing list.

examples

This is relatively straight forward if you were to implement it as a regular Haskell function. Working with quasiquoters though, It all has to be implemented in a Template Haskell. This means that it is sometimes necessary to move code around just so that it compiles, although the types are correct. The reason for this is the stage restriction of Template Haskell. A value cannot be spliced into an expression if it was defined locally. Another big difference is type safety. Before GHC 7.8, the expressions inside a quoted block would not be typechecked. You would still get type errors for the code, but rather than complaining at the definition site it would complain at the usage site. This makes defining complex functions nearly impossible. But even with GHC 7.8 you get some warnings when splicing code in rather than where you defined them. For example, it is not checked if a pattern match is exhaustive. Although this check could be done in the type checker, it is done while desugaring to core. I filed this as a bug in GHC (#9113) and it seems that there is work done which would solve this issue.

write about desugaring

4.2.2 nextblock

In principle basic blocks are not ordered. The only exception is the first block, which also cannot have any predecessors. In reality however, they are mostly ordered in the order of control flow. With this, it is not strictly necessary to specify which block will be next after a loop as it is mostly the textually next one. This simplifies the syntax somewhat.

```
for <ty1> <var1> in <val1> to <val2> with <ty2> <values> as <var2>
          { <loop body> }
```

examples

write about desugaring

- cumbersome syntax
- limited functionality
- jumpnext

4.2.3 mutable variables

Although the above methods work, there are some fundamental problems with them. The fact that you have to specify the variables in the loop header and then return the new value is awkward. It abuses the return for something that it is not, namely jumping to the loop header. I could have chosen to use a jump instead, but then there would be no way to update the accumulator. Another more important flaw is the limitation to one value. It is possible, of course, to bundle all needed values together in a struct, but this adds a lot of necessary boilerplate code.

In an ordinary language a for loop would consist of just the loop counter and would handle accumulators through mutable variables. LLVM doesn't support mutable variables however. Since I need however, I implemented support for them through my quasiquoter. The implementation is described in 4.3.

With this idea, the syntax for the for loop is now reduced to

```
for <ty1> <var1> in <val1> to <val2> { <loop body> }
```

This is a great improvement upon the first approach. The main benefit now is the increased flexibility. It is also more concise and doesn't abuse the return to pass through variables.

As a side effect, the implementation got a lot simpler. It is now no longer necessary to scan for return statements to fill **phi**-instructions for both counter and accumulator. Instead, the last block in the loop body just jumps to an end-block, where the counter gets increased and then jumps to the head. Now the for-loop can be implemented exactly like shown in figure 4.2.

To allow mutable variables, there has to be a way to initialize them. The syntax for this is

```
<var> = <ty> <val>
```

To implement this, I used the select statement. The above code gets translated into

```
<var> = select i1 true, <ty> <val>, <ty> <val>
```

It always selects the first value. Another way would be to leave the second value undefined.

```
<var> = select i1 true, <ty> <val>, <ty> undef
```

Obviously this adds another unnecessary instruction to the produced code. This is negligible however, as it is easily removed by LLVM's constant propagation.

4.3 SSA

The standard method of producing SSA form in LLVM is to use stack allocated variables instead of registers to store values. The LLVM optimizer then uses the mem2reg pass to transform these into registers, adding the necessary phi-nodes in the process.

Since one of the goals of the quasiquoter was to produce llvm-code that is as close to what the programmer wrote as possible, I decided against this approach. Instead I do the transformation myself. To this end, I provide a single function

```
toSSA :: [BasicBlock] -> [BasicBlock]
```

This is called with the list of all `BasicBlock`'s of a given function.

To produce SSA form, variables have to be versioned, so that it is clear which value it holds at each given time. There are a multitude of algorithms to do this. The one most widely used (including in LLVM) is Cytron et al.'s algorithm. It is however is rather involved as it relies on dominance frontiers. An easier approach was presented by Braun et al.

In this algorithm, the SSA is produced directly from the AST, without the need of extra analysis passes. To do this, the definitions of variables are tracked and updated as necessary. Without Φnodes, this is straight forward, as Figure 4.3 shows.

If the given variable has no definition in the current block, an empty Φ-node is inserted and the local definition is set to the Φ-node. Now the lookup is done recursively and the values are added to the Φ-node. This extra step is necessary to detect loops correctly. I

```
1 writeVariable(variable, block, value):
2     currentDef[variable][block] <- value
3
4 readVariable(variable, block):
5     if currentDef[variable] contains block:
6         # local value numbering
7         return currentDef[variable][block]
8     # global value numbering
9     return readVariableRecursive(variable, block)
```

Figure 4.3: Implementation of local value numbering

```
1  readVariableRecursive(variable, block):
2      if |block.preds| = 0:
3          # First block
4          val <- variable
5      else:
6          # Break potential cycles with operandless phi
7          val <- new Phi(block)
8          writeVariable(variable, block, val)
9      writeVariable(variable, block, val)
10     return val
```

Figure 4.4: Implementation of global value numbering

use a slightly different approach than presented in the paper. The original version tracks if the predecessors of a block are all processed. This is necessary since only then can the empty Φ-nodes be filled. This approach makes sense if this is the norm. In case the input is an unordered list of blocks however, I can't make predictions about this. Instead I assume that no block is processed until all blocks are. Figure 4.4 shows the modified `readVariableRecursive`.

Like most efficient graph algorithms, this algorithm relies heavily on mutable data structures. Although Haskell - being purely functional - doesn't support these normally, there are multiple methods to implement them inside a pure Monad. The obvious choices are `State` and `ST`. `IO` also provides mutable state, but is not an option here, since there is no safe way of running it in a pure context. I use `ST`, since it natively provides an arbitrary number of mutable variables with direct access.

To use this however, the `BasicBlocks` have to be in a mutable format.

```
type CFG s =[(Name, MutableBlock s)]

data MutableBlock s = MutableBlock {
  blockName :: Name,
  blockIncompletePhis :: STRef s (M.Map Name (MutableInstruction s)),
  blockPhis :: STRef s [MutableInstruction s],
  blockInstructions :: [MutableInstruction s],
  blockTerminator :: MutableTerminator s,
  blockPreds :: [Name],
  blockDefs :: STRef s (M.Map Name Name)
  }

type MutableInstruction s = STRef s (Named Instruction)
type MutableTerminator  s = STRef s (Named Terminator)
```

```
toSSA :: [BasicBlock] -> [BasicBlock]
toSSA bbs = runST $ do
  cfg <- toCFG $ bbs
  ctr <- newSTRef 1

  -- process all Instructions
  mapM_ (blockToSSAPre ctr) (map snd cfg)
  -- replace names in Phis with correct references
  mapM_ (blockToSSAPhi ctr cfg) (map snd cfg)
  -- replace names in newly added Phis with correct references
  handleIncompletePhis ctr cfg

  fromCFG cfg
```

Figure 4.5: `toSSA`

Although not everything is mutable in this representation, it is sufficient. After the conversion, the `BasicBlocks` can be processed one by one. First, the variables created by `Instructions` are adjusted. The same is done for usages. If there a variable is not defined in a given block and there are predecessors, an incomplete `Phi` instruction is added. Incomplete here means, that the values for every given predecessor are left undefined. When this is done, the variables referenced by existing `Phi` instructions are replaced by the new values.

Every time a recursive read is done, a new incomplete `Phi` instruction is added. These are handled last by reading the variable in all preceding blocks and replacing the undefined values. Since this involved reading however, new incomplete `Phi` instructions can be added in the process. To handle all of them, the function is executed until there are no incomplete `Phi` instructions.

Figure 4.5 shows the complete `toSSA` function.

5 Skeletons

5.1 map

5.2 fold

5.3 scan

[LF80] My plan is the following:

```
1  void scan(double* in, double* out, double* tmp, unsigned length,
       unsigned start, unsigned end, unsigned tid) {
2    double acc = 0;
3    if (end==length) {
4      tmp[0] = 0;
5    } else {
6      for (unsigned i=start;i<end;i++) {
7        acc += in[i];
8      }
9      tmp[tid+1] = acc;
10   }
11   printf("block");
12   acc = tmp[tid];
13   for (unsigned j=start;j<end;j++) {
14     acc += in[j];
15     out[j] = acc;
16   }
17 }
18
19 void scanAlt(double* in, double* out, unsigned length, unsigned start,
       unsigned end) {
20   double acc = 0;
21   for (unsigned j=start;j<end;j++) {
22     acc += in[j];
23     out[j] = acc;
24   }
25   printf("block");
26   double add=0;
27   if (start>0) {
28     add = out[start-1];
29   }
30   for (unsigned i=start;i<end;i++) {
31     out[i] += add;
```

```
32        }
33    }
```

5.4 stencil

6 Conclusion

6.1 Benchmarks

6.2 Related Work

References

[AWZ88] Alpern, Bowen ; Wegman, Mark N ; Zadeck, F Kenneth: Detecting equality of variables in programs. In: *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM. 1988, pp. 1–11. URL: https://courses.cs.washington.edu/courses/cse501/04wi/papers/alpern-popl88.pdf

[Bra+13] Braun, Matthias et al.: Simple and Efficient Construction of Static Single Assignment Form. In: Jhala, Ranjit ; Bosschere, Koen (eds.): *Compiler Construction*. Vol. 7791. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2013, pp. 102–122. DOI: 10.1007/978-3-642-37051-9_6. URL: http://dx.doi.org/10.1007/978-3-642-37051-9%5C_6

[Cha+11] Chakravarty, Manuel MT et al.: Accelerating Haskell array codes with multicore GPUs. In: *Proceedings of the sixth workshop on Declarative aspects of multicore programming*. ACM. 2011, pp. 3–14. URL: http://morri.web.cse.unsw.edu.au/~chak/papers/acc-cuda.pdf

[Cha+07] Chakravarty, Manuel MT 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. URL: http://www.cse.unsw.edu.au/~chak/papers/data-parallel-haskell.pdf

[Cyt+91] Cytron, Ron et al.: Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. In: *ACM Transactions on Programming Languages and Systems* 13 (Oct. 1991) Nr. 4, pp. 451–490. URL: http://doi.acm.org/10.1145/115372.115320

[Die14] Diehl, Stephen: *Implementing a JIT Compiled Language with Haskell and LLVM*. Jan. 2014. URL: http://www.stephendiehl.com/llvm/

[DM] Dornan, Chris ; Marlow, Simon: *Alex: A lexical analyser generator for Haskell*. URL: http://www.haskell.org/alex/

[GM95] Gill, Andy ; Marlow, Simon: Happy: the parser generator for Haskell. In: *University of Glasgow* (1995)

[Hol14] Holtz, Timo von: *llvm-general-quote*. 2014. URL: https://github.com/tvh/language-llvm-quote

[How14] Howell, Nathan: *llvm-general-typed*. 2014. URL: https://github.com/alphaHeavy/llvm-general-typed

[Kel+10] Keller, Gabriele et al.: Regular, shape-polymorphic, parallel arrays in Haskell. In: *ACM Sigplan Notices*. Vol. 45. 9. ACM. 2010, pp. 261–272. URL: http://www.cse.unsw.edu.au/~chak/papers/repa.pdf

[LF80] Ladner, Richard E ; Fischer, Michael J: Parallel prefix computation. In: *Journal of the ACM (JACM)* 27 (1980) Nr. 4, pp. 831–838. URL: https://engr.smu.edu/~seidel/courses/cse8351/papers/LadnerFischer80.pdf

[Lat02] Lattner, Chris Arthur: *LLVM: An infrastructure for multi-stage optimization*. MA thesis. University of Illinois, 2002. URL: https://llvm.org/svn/llvm-project/www-pubs/trunk/2002-12-LattnerMSThesis-book.pdf

[LLV14] LLVM Project: *Auto-Vectorization in LLVM*. [Online; accessed 1-July-2014]. 2014. URL: http://llvm.org/docs/Vectorizers.html

[LLV] LLVM Project: *LLVM Language Reference Manual*. [Online; accessed 1-July-2014]. URL: http://llvm.org/docs/LangRef.html

[Mai07] Mainland, Geoffrey: Why it's nice to be quoted: quasiquoting for haskell. In: *Proceedings of the ACM SIGPLAN workshop on Haskell workshop*. ACM. 2007, pp. 73–82. URL: https://www.cs.drexel.edu/~mainland/publications/mainland07quasiquoting.pdf

[MC] Mainland, Geoffrey ; Chakravarty, Manuel MT: *language-c-quote*. URL: http://hackage.haskell.org/package/language-c-quote

[McD14] McDonell, Trevor L: *accelerate-llvm*. 2014. URL: https://github.com/AccelerateHS/accelerate-llvm

[McD+13] McDonell, Trevor L et al.: Optimising Purely Functional GPU Programs. In: *Proceedings of the The 18th ACM SIGPLAN International Conference on Functional Programming*. 2013. URL: http://cs3141.web.cse.unsw.edu.au/~chak/papers/acc-optim.pdf

[RWZ88] Rosen, Barry K ; Wegman, Mark N ; Zadeck, F Kenneth: 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. URL: http://www.cs.wustl.edu/~cytron/cs531/Resources/Papers/valnum.pdf

[Sca13] Scarlet, Benjamin: *llvm-general*. 2013. URL: http://hackage.haskell.org/package/llvm-general