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 2, 2014

Selbstständigkeitserklärung

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

July 2, 2014

Timo von Holtz

Todo list

Contents

List of Figures

List of Tables

1. Introduction

Write Introduction

2. Technologies

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 instructions with a terminator at the end. Instructions are add, mult, call, . . . , but also

```c
int sum(int* a, int length) {
  int x = 0;
  for (int i=0;i<length;i++) {
    x += a[i];
  }
  return x;
}
```

Figure 2.1.: sum as a C function

```
define i32 @sum(i32* nocapture readonly %a, i32 %length) {
entry:
  %cmp4 = icmp sgt i32 %length, 0
  br i1 %cmp4, label %for.body, label %for.end

for.body:                                         ; preds = %entry, %for.body
  %indvars.iv = phi i64 [ %indvars.iv.next, %for.body ], [ 0, %entry ]
  %x.05 = phi i32 [ %add, %for.body ], [ 0, %entry ]
  %arrayidx = getelementptr inbounds i32* %a, i64 %indvars.iv
  %0 = load i32* %arrayidx, align 4
  %add = add nsw i32 %0, %x.05
  %indvars.iv.next = add nuw nsw i64 %indvars.iv, 1
  %lftr.wideiv = trunc i64 %indvars.iv.next to i32
  %exitcond = icmp eq i32 %lftr.wideiv, %length
  br i1 %exitcond, label %for.end, label %for.body

for.end:                                          ; preds = %for.body, %entry
  %x.0.lcssa = phi i32 [ 0, %entry ], [ %add, %for.body ]
  ret i32 %x.0.lcssa
}
```

Figure 2.2.: sum as a LLVM

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`

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).

```
define i32 @sum(i32* nocapture readonly %a, i32 %length) #0 {
entry:
  %cmp4 = icmp sgt i32 %length, 0
  br i1 %cmp4, label %for.body.preheader, label %for.end

for.body.preheader:                               ; preds = %entry
  %0 = add i32 %length, -1
  %1 = zext i32 %0 to i64
  %2 = add i64 %1, 1
  %end.idx = add i64 %1, 1
  %n.vec = and i64 %2, 8589934590
  %cmp.zero = icmp eq i64 %n.vec, 0
  br i1 %cmp.zero, label %middle.block, label %vector.body

vector.body:                                      ; preds = %for.body.preheader, %vector.body
  %index = phi i64 [ %index.next, %vector.body ], [ 0, %for.body.preheader ]
  %vec.phi = phi <2 x i32> [ %5, %vector.body ], [ zeroinitializer, %for.body.preheader ]
  %3 = getelementptr inbounds i32* %a, i64 %index
  %4 = bitcast i32* %3 to <2 x i32>*
  %wide.load = load <2 x i32>* %4, align 4
  %5 = add nsw <2 x i32> %wide.load, %vec.phi
  %index.next = add i64 %index, 2
  %6 = icmp eq i64 %index.next, %n.vec
  br i1 %6, label %middle.block, label %vector.body

middle.block:                                     ; preds = %vector.body, %for.body.preheader
  %resume.val = phi i64 [ 0, %for.body.preheader ], [ %n.vec, %vector.body ]
  %rdx.vec.exit.phi = phi <2 x i32> [ zeroinitializer, %for.body.preheader ], [ %5, %vector.body ]
  %rdx.shuf = shufflevector <2 x i32> %rdx.vec.exit.phi, <2 x i32> undef, <2 x i32> <i32 1, i32 undef>
  %bin.rdx = add <2 x i32> %rdx.vec.exit.phi, %rdx.shuf
  %7 = extractelement <2 x i32> %bin.rdx, i32 0
  %cmp.n = icmp eq i64 %end.idx, %resume.val
  br i1 %cmp.n, label %for.end, label %for.body

for.body:                                         ; preds = %middle.block, %for.body
  %indvars.iv = phi i64 [ %indvars.iv.next, %for.body ], [ %resume.val, %middle.block ]
  %x.05 = phi i32 [ %add, %for.body ], [ %7, %middle.block ]
  %arrayidx = getelementptr inbounds i32* %a, i64 %indvars.iv
  %8 = load i32* %arrayidx, align 4
  %add = add nsw i32 %8, %x.05
  %indvars.iv.next = add nuw nsw i64 %indvars.iv, 1
  %lftr.wideiv = trunc i64 %indvars.iv.next to i32
  %exitcond = icmp eq i32 %lftr.wideiv, %length
  br i1 %exitcond, label %for.end, label %for.body

for.end:                                          ; preds = %for.body, %middle.block, %entry
  %x.0.lcssa = phi i32 [ 0, %entry ], [ %add, %for.body ], [ %7, %middle.block ]
  ret i32 %x.0.lcssa
}
```

Figure 2.3.: Vectorized Sum

```
define void @foo(double %a1, double %a2, double %b1, double %b2, double* %A)
    {
entry:
  %add = fadd double %a1, %b1
  %mul = fmul double %add, %a1
  %div = fdiv double %mul, %b1
  %mul1 = fmul double %b1, 5.000000e+01
  %div2 = fdiv double %mul1, %a1
  %add3 = fadd double %div, %div2
  store double %add3, double* %A, align 8
  %add4 = fadd double %a2, %b2
  %mul5 = fmul double %add4, %a2
  %div6 = fdiv double %mul5, %b2
  %mul7 = fmul double %b2, 5.000000e+01
  %div8 = fdiv double %mul7, %a2
  %add9 = fadd double %div6, %div8
  %arrayidx10 = getelementptr inbounds double* %A, i64 1
  store double %add9, double* %arrayidx10, align 8
  ret void
}
```

Figure 2.4.: Vectorized Sum

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

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]:

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.

```
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.: Vectorized Sum

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.

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.

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)
```

2.2.1. LLVM Backend

[McD] uses gang workers.[Cha+07]

3. Contributions

3.1. accelerate

- StorableArray (errors in caching)

3.2. llvm-general

- Targetmachine (Optimization)
- fast-math

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.

longer introduction to monadic code generation

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.

Another approach would be to use an EDSL.

write about llvm-general-typed

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.

```
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

```
define i32 @foo(i32 %x) #0 {
entry:
  br label %for.cond

for.cond:                                         ; preds = %for.inc, %entry
  %res.0 = phi i32 [ 0, %entry ], [ %add, %for.inc ]
  %i.0 = phi i32 [ 0, %entry ], [ %inc, %for.inc ]
  %cmp = icmp slt i32 %i.0, %x
  br i1 %cmp, label %for.body, label %for.end

for.body:                                         ; preds = %for.cond
  %add = add nsw i32 %res.0, %x
  %inc = add nsw i32 %i.0, 1
  br label %for.cond

for.end:                                          ; preds = %for.cond
  ret i32 %res.0
}
```

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 |]`

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.

```
[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

```
[llg|
define i64 @foo(i64 %start, i64 %end) {
  entry:
    %x = i64 0

  for:
    for i64 %i in %start to %end {
        %x = add i64 %i, %x
    }

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

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

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.
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.

3.3.3. syntax extensions

llvm-general-quote supports all llvm instructions. I added the following 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> }`

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

for.head:                                     ; preds = %n0, %entry
  %x.12 = phi i64 [ 0, %entry ], [ %x.6, %n0 ]
  %i.4 = phi i64 [ %start, %entry ], [ %i.9, %n0 ]
  %for.cond.3 = icmp slt i64 %i.4, %end
  br i1 %for.cond.3, label %n0, label %for.end

n0:                                           ; preds = %for.head
  %x.6 = add i64 %i.4, %x.12
  %i.9 = add nuw nsw i64 %i.4, 1
  br label %for.head

for.end:                                      ; preds = %for.head
  ret i64 %x.12
}
```

Figure 3.5.: Expanded For Loop (SSA)

4. Implementation

4.1. Quasiquoter

The design of *llvm-general-quote* is inspired by *language-c-quote*, which is also used in the cuda implementation of Accelerate. I use “Happy” and *Alex*.

4.2. Extension for-loop

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. There are a multitude of algorithms to construct LLVM. The one most widely used (including in LLVM) is Cytron et al.'s arlgorithm. It is however is rather involved as it relies on dominance frontiers. An easier approach was presented by Braun et al.

write about implementation of SSA transformation

[Bra+13]

5. Skeletons

5.1. map

5.2. fold

5.3. scan

[LF80] My plan is the following:

```
void scan(double* in, double* out, double* tmp, unsigned length, unsigned
    start, unsigned end, unsigned tid) {
  double acc = 0;
  if (end==length) {
    tmp[0] = 0;
  } else {
    for (unsigned i=start;i<end;i++) {
      acc += in[i];
    }
    tmp[tid+1] = acc;
  }
  printf("block");
  acc = tmp[tid];
  for (unsigned j=start;j<end;j++) {
    acc += in[j];
    out[j] = acc;
  }
}

void scanAlt(double* in, double* out, unsigned length, unsigned start,
    unsigned end) {
  double acc = 0;
  for (unsigned j=start;j<end;j++) {
    acc += in[j];
    out[j] = acc;
  }
  printf("block");
  double add=0;
  if (start>0) {
    add = out[start-1];
  }
  for (unsigned i=start;i<end;i++) {
    out[i] += add;
    }
}
```

5.4. stencil

6. Conclusion

6.1. Related Work

A. Listings

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: *Proceedings of the International Conference on Compiler Construction.* Lecture Notes in Computer Science. 2013. URL: `http://pp.info.uni-karlsruhe.de/uploads/publikationen/braun13cc.pdf`

[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`

[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`

[McD] McDonell, Trevor L: *accelerate-llvm*. 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`