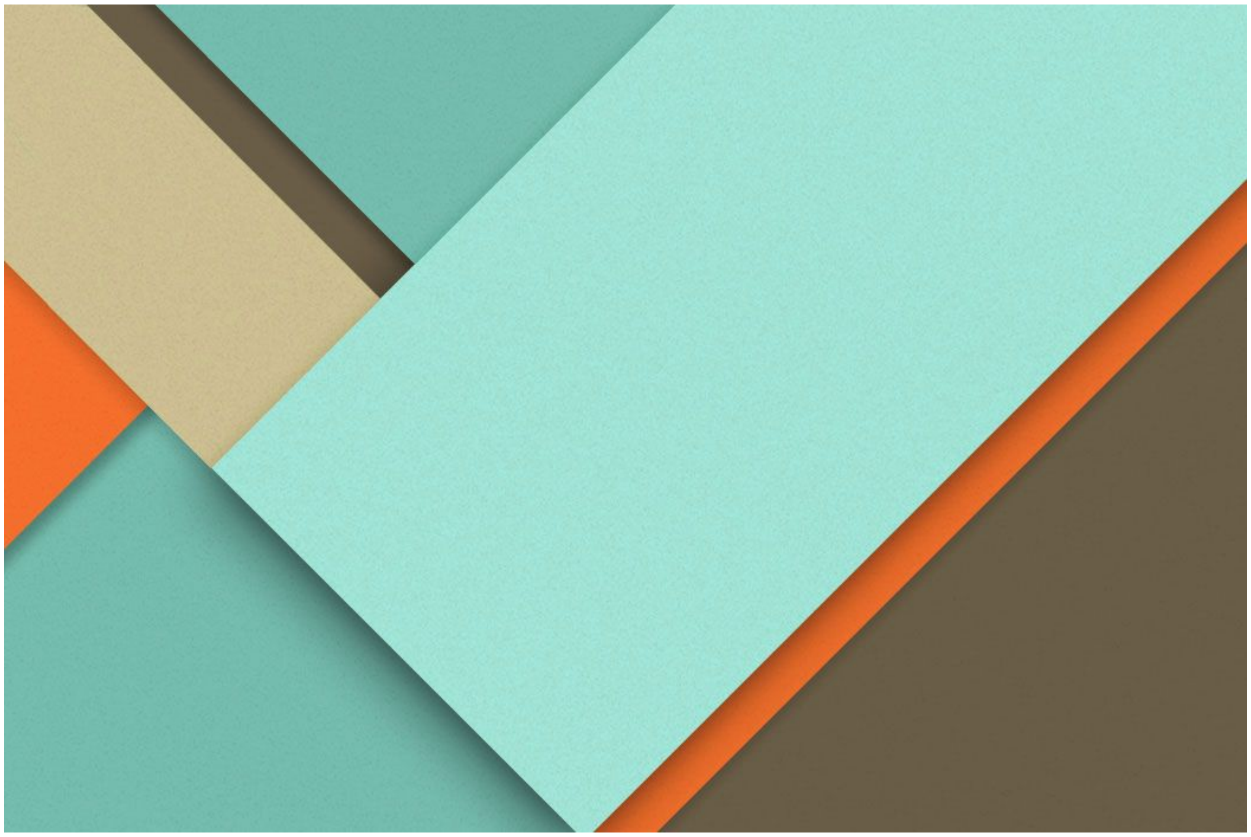

SoH Proposal: Completing the LLVM backend for Accelerate

01.05.2016

Ziheng Jiang
ziheng@apache.org
https://github.com/ZihengJiang

Overview

Preliminaries: Accelerate LLVM

Data.Array.Accelerate [1] defines an embedded language of array computations for high-performance computing in Haskell, which can accelerate the computation with multicore CPU/GPU.

As a simple example, consider the computation of a dot product of two vectors of single-precision floating-point numbers:

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

The computations are online-compiled and executed on a range of architectures for performance — for example, using Data.Array.Accelerate.CUDA, it may be on-the-fly off-loaded to a GPU.

However, only two complete Accelerate backends have materialised: the interpreter, and the CUDA backend targeting GPUs.

With the spirit of being reusable and portable across diverse architectures, the Accelerate-LLVM backend [2, 3] was introduced at Haskell Symposium last year: a framework for constructing backends targeting LLVM IR, which is also demonstrated to have a very good performance. The following benchmarks demonstrate that Accelerate-LLVM multicore backend has very good performance compared to Repa, the current defacto multicore array programming library in Haskell:

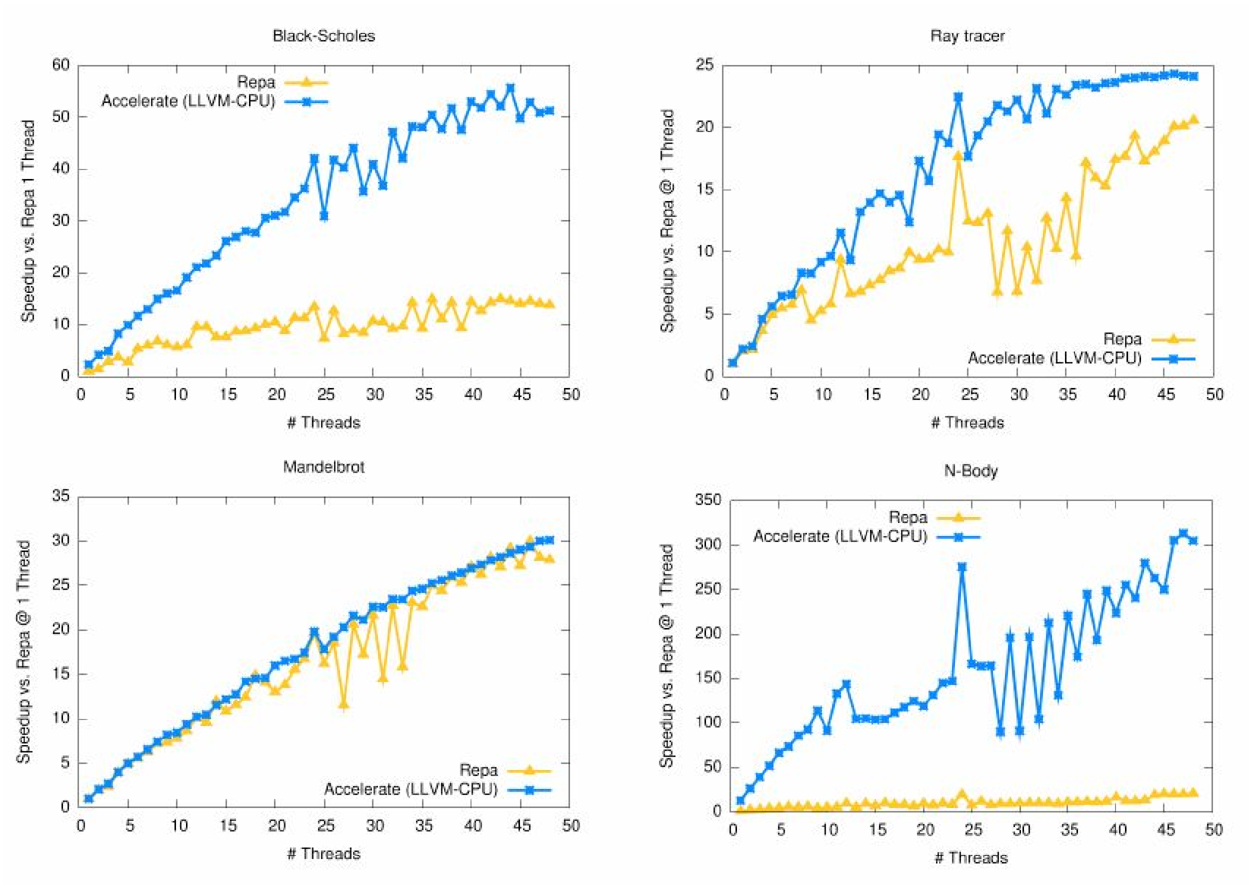

Figure 1. Accelerate-LLVM has a significant performance improvement over Repa

Unfortunately, the Accelerate LLVM backend is not yet complete, so users can not take advantage of these great performance gains. If we can complete the LLVM backends, especially the CPU backend, then Accelerate will be usable on both GPUs (using the current CUDA backend or new, improved LLVM GPU backend) as well as multicore CPUs (with the new LLVM multicore backend) so all Haskellers we be able to take advantage of the great performance offered by Accelerate.

For more details on Accelerate, see these papers:

Type-safe Runtime Code Generation: Accelerate to LLVM [3] (slides [4]) (video [5])

Accelerating Haskell Array Codes with Multicore GPUs [6]

Optimising Purely Functional GPU Programs [7] (slides [8])

Embedding Foreign Code [9]

Scope

I'm going to implement the remaining collective operations in order to complete the Accelerate-LLVM backend, which are listed on the Accelerate-LLVM repo [2]. However, I will not implement stencil operations (a basic, unoptimised implementation exists which will be sufficient for now) and the foreign function interface (as this aspect contains an unknown research component). More details about those operations can be found at the Specifications section.

Since the Accelerate CUDA backend exists as a reference implementation for the GPU component, there should not be any surprises.

Impact

Make High Performance Array Programming Available to More People in The Haskell Community

The overarching aim of Accelerate is to make high performance array programming available to more people in the Haskell community. We want to reach a point where Accelerate-LLVM is installed incidentally, as a dependency of some other package where the author used Accelerate to speed up some parts of the computation.

Easy Migration from Repa to get a Performance Boost

The Haskell Symposium paper [3] demonstrated that Accelerate is several times faster than Repa. Since Accelerate and Repa have very similar APIs, anybody using Repa for [parallel] array programming should be able to switch to Accelerate-LLVM with relative ease and get a performance boost. It is worth mentioning that Repa also recommends LLVM be installed for good performance (compile with the -fllvm flag to GHC), so the migration from Repa to Accelerate-LLVM imposes no additional external dependencies.

Have A Stronger Expression Capability

Repa actually has a very small API compared to Accelerate, so in some sense it is actually quite difficult to implement many interesting algorithms in Repa simply because it lacks many parallel operations (such as scan) which are found in Accelerate. So, it should be easier for more developers to add parallelism to their programs with Accelerate.

About the student: Ziheng Jiang

I(Ziheng Jiang [10]), am a second year undergraduate majoring in Computer Science. I have been learning Haskell for one year and have a good knowledge about parallel computing on GPU and basic knowledge about compilers.

Last month, I joined DMLC [11], a group to collaborate on open-source machine learning projects, and build a tool to do CPU/GPU profiling [12]. For the next next two months, I'll be working on improving performance of mxnet [13] under the guidance of DMLC members. I also built a tiger compiler front-end [14] in my sophomore year.

About the mentors

Trevor [17], who will be my primary mentor, is a postdoctoral fellow at The University of New South Wales and the lead developer of Accelerate, having implemented both the CUDA and LLVM backends.

Manuel M. T. Chakravarty [18] and Ryan Newton [19], two other developers of Accelerate, have mentored GSoC students in the past and have offered to advise in an unofficial capacity, to help me define milestones and project scopes.

My mentors are knowledgeable in the technical aspects of the work and have the project management experience necessary to ensure that the project is achievable within the Summer of Haskell timeline.

Goal

The goal of the project is to complete the implementation of the Accelerate-LLVM backend by adding support for the missing collective operations during this summer, as listed on the Accelerate-LLVM repo [2] (except for stencil and foreign functions).

For the multicore-CPU backend, these operations are:

- scanl
- scanr
- scanl1
- scanr1
- scanl'
- scanr'
- foldSeg
- permute

For the GPU backend:

- fold
- foldAll
- scanr
- scanl1
- scanr1
- scanl'

- scanr’
- foldSeg
- permute

Specifications

The Accelerate-LLVM Backend

This project focuses on work in the LLVM backend of Accelate. The engineering infrastructure of this backend is already complete, as demonstrated by the Haskell Symposium paper [3]. Thus, this is largely an engineering project to complete the missing collective operations, and does not require large design decisions for completing the compiler. The LLVM backend backend consists of two main components:

Native Backend

The LLVM-based backend for the Accelerate targeting multicore CPUs.

PTX Backend

The LLVM-based backend for the Accelerate code targeting CUDA capable GPUs. The existing (and complete) Accelerate-CUDA backend will form a reference implementation in order to complete this aspect of the project.

LLVM IR

LLVM IR is a Static Single Assignment (SSA) based representation that provides type safety, low-level operations, flexibility, and the capability of representing ‘all’ high-level languages cleanly. It is the common code representation used throughout all phases of the LLVM compilation strategy. The infrastructure for generating LLVM instructions for Accelerate is completed already. This project aims to use this infrastructure to build out the remaining collective operations.

The Accelerate-CUDA Backend

The accelerate-cuda backend [20] compiles Accelerate code down to NVIDIA's CUDA language for general-purpose GPU programming, which forms a reference implementation (for the GPU part).

The Accelerate-Examples Package

The accelerate-examples package [21] provides a fairly comprehensive quickcheck and unit-test based test suite to check whether my implementations are correct. There are also several benchmark programs I can use to measure performance. They are already set up to use the LLVM backend. I will use this as my main test and benchmark suite to ensure the project progresses correctly.

Modularity of This Project

This project consists of very modular tasks, so will still be a success even if not all of the operators are implemented, it we will still improve the completeness of the LLVM backends and hence its utility for other Haskellers.

Details on the operations to be implemented

Operation: fold (multidimensional)

```
fold :: (Shape ix, Elt a)
    => (Exp a -> Exp a -> Exp a)
    -> Exp a -> Acc (Array (ix :. Int) a) -> Acc (Array ix a)
```

Reduction of the innermost dimension of an array of arbitrary rank. The first argument needs to be an associative function to enable an efficient parallel implementation.

Operation: foldAll

```
foldAll :: (Shape sh, Elt a)
       => (Exp a -> Exp a -> Exp a)
       -> Exp a -> Acc (Array sh a) -> Acc (Scalar a)
```

Reduction of an array of arbitrary rank to a single scalar value.

Operation: scanl, scanr

```
scanl :: Elt a
     => (Exp a -> Exp a -> Exp a)
     -> Exp a -> Acc (Vector a) -> Acc (Vector a)
```

Data.List style left-to-right scan, but with the additional restriction that the first argument needs to be an associative function to enable an efficient parallel implementation. The initial value (second argument) may be arbitrary.

scanr: Right-to-left variant of scanl.

Operation: scanl1, scanr1

```
scanl1 :: Elt a
       => (Exp a -> Exp a -> Exp a)
       -> Acc (Vector a) -> Acc (Vector a)
```

Data.List style left-to-right scan without an initial value (aka inclusive scan). Again, the first argument needs to be an associative function.

scanr1: Right-to-left variant of scanl1.

Operation: scanl', scanr'

```
scanl' :: Elt a
      => (Exp a -> Exp a -> Exp a)
      -> Exp a -> Acc (Vector a)
      -> (Acc (Vector a), Acc (Scalar a))
```

Variant of scanl, where the final result of the reduction is returned separately.

scanr': Right-to-left variant of scanl'.

Operation: foldSeg

```
foldSeg :: (Shape ix, Elt a, Elt i, IsIntegral i)
        => (Exp a -> Exp a -> Exp a)
        -> Exp a -> Acc (Array (ix :. Int) a)
        -> Acc (Segments i)
        -> Acc (Array (ix :. Int) a)
```

Segmented reduction along the innermost dimension. Performs one individual reduction per segment of the source array. These reductions proceed in parallel.

Operation: permute

```
permute :: (Shape ix, Shape ix', Elt a)
        => (Exp a -> Exp a -> Exp a)
        -> Acc (Array ix' a)
        -> (Exp ix -> Exp ix')
        -> Acc (Array ix a)
        -> Acc (Array ix' a)
```

Forward permutation specified by an index mapping. The result array is initialised with the given defaults and any further values that are permuted into the result array are added to the current value using the given combination function.

Milestones

I. May 22nd - June 12th

Community bonding period on the timeline.

- Keep communication with community
- Read documents and relevant papers
- Go deep in the source code further
- Write documents about existed operations of Accelerate-LLVM and try to commit.

June 13th, students begin work on the timeline.

II. June 13th - June 26th || TWO WEEKS

- Implement operations “fold”, “foldAll” on PTX backend
- Run unit-test to check the correctness
- Write document

III. June 27th - July 10th || TWO WEEKS

- Implement operations “scanl, scanr’ on Native backend
- Implement operations “scanl, scanr’ on PTX backend
- Run unit-test to check the correctness
- Write document

IV. July 11th - July 17th || ONE WEEK

- Implement operations “scanl1”, “scanr1” on Native backend
- Run unit-test to check the correctness
- Write document

July 18st, midterm evaluations begin on the timeline

V. July 18th - July 24th || ONE WEEK

- Implement operations “scanl1”, “scanr1” on PTX backend
- Run unit-test to check the correctness
- Write document

VI. August 25th - August 7th || TWO WEEKS

- Implement operations “scanl’ ”, “scanr’ “ on Native backend
- Implement operations “scanl’ “, “scanr’ “ on PTX backend
- Run unit-test to check the correctness
- Write document

VII. August 8th - August 21th || TWO WEEKS

- Implement operations “foldSeg” on Native backend
- Implement operations “foldSeg“ on PTX backend
- Run unit-test to check the correctness
- Write document

VIII. August 22th - September 1st || TWO WEEKS

- Implement operations “permute” on Native backend
- Implement operations “permute“ on PTX backend
- Run unit-test to check the correctness
- Write document
- Improve the code quality

September 2nd, end of work period on the timeline.

References

[1] https://github.com/AccelerateHS/accelerate

[2] https://github.com/AccelerateHS/accelerate-llvm

[3] *Type-safe Runtime Code Generation: Accelerate to LLVM*. Trevor L. McDonell, Manuel M. T. Chakravarty, Vinod Grover, and Ryan R. Newton. Haskell Symposium (2015).

[4] https://speakerdeck.com/tmcdonell/type-safe-runtime-code-generation-accelerate-to-llvm

[5] https://www.youtube.com/watch?v=snXhXA5noVc

[6] *Accelerating Haskell Array Codes with Multicore GPUs*. Manuel M. T. Chakravarty, Gabriele Keller, Sean Lee, Trevor L. McDonell, and Vinod Grover. Declarative Aspects of Multicore Programming (2011).

[7] *Optimising Purely Functional GPU Programs.* Trevor L. McDonell, Manuel M. T. Chakravarty, Gabriele Keller, Ben Lippmeier. ACM SIGPLAN International Conference on Functional Programming (2013).

[8] https://speakerdeck.com/tmcdonell/optimising-purely-functional-gpu-programs

[9] *Embedding Foreign Code.* Robert Clifton-Everest, Trevor L. McDonell, Manuel M. T. Chakravarty, and Gabriele Keller. International Symposium on Practical Aspects of Declarative Languages (2014)

[10] https://github.com/ZihengJiang

[11] https://github.com/dmlc , http://dmlc.ml/

[12] https://github.com/dmlc/minpy/commit/a040e4542e03b58ff7def57e7eff1d8434ecef2e

[13] https://github.com/dmlc/mxnet

[14] https://github.com/ZihengJiang/TigerCompiler

[15] http://www.jzhthomas.com/wp-content/uploads/2016/05/Ziheng_Jiang_Resume.pdf

[16] http://www.jzhthomas.com/

[17] https://github.com/tmcdonell

[18] http://www.cse.unsw.edu.au/~chak/

[19] http://www.cs.indiana.edu/~rrnewton/homepage.html

[20] https://github.com/AccelerateHS/accelerate-cuda

[21] https://github.com/AccelerateHS/accelerate-examples