Lift Documentation

The Lift Team

Sep 11, 2019

Table of Contents:

1 Getting Started with *Lift* **3**
- 1.1 Download . . . 3
- 1.2 Installation . . . 4
- 1.3 Building Lift . . . 6

2 Overview of *Lift* **15**
- 2.1 Patterns . . . 15
- 2.2 Rewrite Rules . . . 17
- 2.3 Compilation Flow . . . 17
- 2.4 Inferring OpenCL Thread Counts . . . 18

3 Developing *Lift* **19**
- 3.1 Testing . . . 19
- 3.2 Guidelines for Contributing . . . 20
- 3.3 How to 22

4 Generating and Running Kernels **27**
- 4.1 Generating Kernels . . . 27
- 4.2 Running Kernels . . . 27
- 4.3 High-Level Rewrite . . . 28

5 Indices and tables **31**

This is the documentation of the *Lift* programming language and compiler. The lift project is a research project by a team at the University of Edinburgh together with further collaborators.

CHAPTER 1

Getting Started with *Lift*

1.1 Download

To download *Lift* to your local machine perform the following steps:

1. Ensure that `git` and `git-lfs` are installed on your machine

   Lift is distributed via https://github.com using the `git` version control system[1].

   To download *Lift* to your computer you, therefore, have to first ensure that you have `git` and `git-lfs` installed on your system:

   ```
   > git --version
   > git lfs version
   ```

   If this prints out an error message instead of an version number, you should install `git` and `git-lfs` via you Linux distribution's package manager or from the official webpages[2][3].

2. Clone the *Lift* git repository

   Run the following command to clone the repository:

   ```
   > git clone https://github.com/lift-project/lift.git
   ```

   This will download the repository in a directory called `lift` in the current working directory.

[1] If you want to learn more about github and git this site is a good starting point: https://help.github.com/articles/good-resources-for-learning-git-and-github/

[2] Git Version Control System https://git-scm.com/

[3] Git Large File Storage https://git-lfs.github.com/

Links

1.2 Installation

There are two main options how to install *Lift*:

- Maybe the easiest options is to install it via `docker` but unfortunately, accessing GPUs from `docker` might be an issue.
- For proper development we recommend to install *Lift* natively which provides the most flexibility but might require a bit more time to setup.

Note: *Lift* currently is only supported on Linux machines

1.2.1 Installation via Docker

Docker is a technology which allows to package software with all its dependencies into a *container*.

We have bundled *Lift* an all of its dependencies inside such a container. The *Lift* container can be executed either directly via `docker`, which does not allow the execution of *Lift* programs on GPUs, or via `nvidia-docker`, which allows the execution of *Lift* programs on Nvidia GPUs from inside the docker container.

Option A: Using plain `docker`

To create and use the *Lift* docker container follow these steps:

1. Install `docker`

   Docker requires a 64-bit Linux installation with a kernel version 3.10 or higher. Instructions to install docker can be found here: https://docs.docker.com/engine/installation/.

2. Build the *Lift* docker container image

   To build the container image and giving it the name `lift` execute the following command from the *Lift* root directory.

   ```
   docker build -t lift docker
   ```

3. Run the *Lift* docker container

   To run the container in with an interactive shell execute the following command from the *Lift* root directory.

   ```
   docker run -it --rm -v `pwd`:/lift lift
   ```

   This command makes the current directory accessible inside the container under the path `/lift`. The container will be deleted after you exit the interactive shell. Changes to the files in the `/lift` directory of the container are persistent while all other changes inside the container are not.

4. Compile *Lift* and run the test suite

   You are now ready to compile *Lift* and run the test suite to see if everything works as expected.

Option B: Using `nvidia-docker`

To create and use the *Lift* docker container with `nvidia-docker` follow these steps:

1. Install `docker` **and** `nvidia-docker`

   Docker requires a 64-bit Linux installation with a kernel version 3.10 or higher. Instructions to install docker can be found here: https://docs.docker.com/engine/installation/.

   `nvidia-docker` is a plugin for docker which makes Nvidia GPUs accessible inside the container. Installation instructions for `nvidia-docker` can be found here: https://github.com/NVIDIA/nvidia-docker/wiki/Installation.

2. Build the *Lift* docker container image

   To build the container image and giving it the name `lift` execute the following command from the *Lift* root directory.

   ```
   docker build -t lift nvidia-docker
   ```

3. Run the *Lift* docker container

   To run the container in with an interactive shell execute the following command from the *Lift* root directory.

   ```
   nvidia-docker run -it --rm -v `pwd`:/lift lift
   ```

   This command makes the current directory accessible inside the container under the path `/lift`. The container will be deleted after you exit the interactive shell. Changes to the files in the `/lift` directory of the container are persistent while all other changes inside the container are not.

4. Compile *Lift* and run the test suite

   You are now ready to compile *Lift* and run the test suite to see if everything works as expected.

1.2.2 Native Installation

Lift has a number of dependencies which have to be installed first before compiling and using *Lift*.

Dependencies

Lift has the following dependencies:

- OpenCL
- CMake and a C++ compiler (i.e. `gcc` or `clang`)
- Java 8 SDK
- `SBT` (the *Scala Build Tool*)

In the following we individually discuss how to install the dependencies.

OpenCL

OpenCL is a framework for programming heterogeneous devices such as GPUs. *Lift* currently generates OpenCL kernels, functions implemented in the OpenCL C programming language and executed in parallel. Therefore, you have to have a working OpenCL implementation for executing *Lift* programs.

Andreas Klöckner maintains a very good guide on how to install OpenCL for Linux: https://wiki.tiker.net/OpenCLHowTo.

CMake and a C++ Compiler

For *Lift* to communicate with OpenCL it relies on a small library which connects the *Lift* compiler implemented in Scala and running in the Java virtual machine to the C OpenCL implementation. To compile this library we need a C++ compiler and CMake.

A C++ Compiler should be easy to install via your Linux distribution's package manager.

Similarly, CMake should be easy to install via your package manage. If not, CMake is easy to build from source as explained here: https://cmake.org/install/.

Java

Lift is implemented in Scala which is a programming language running on top of the Java virtual machine. Therefore, a Java installation is required for running *Lift*. The package manager of your Linux distribution will most likely provide a Java implementation. Oracle Java implementation is accessible here: http://www.oracle.com/technetwork/java/javase/downloads/index.html.

Note: Java 8 is required for *Lift*. This is a strict requirement!

To check the installed Java version run:

```
> java -version
```

and

```
> javac - version
```

Both commands should print `"1.8.0"` or newer. If this is not the case you have to install a newer version of Java.

`SBT`

`SBT`, the *Scala Build Tool*, is a versatile tool for building Scala source code. It downloads the exact version of the Scala compiler required by *Lift*. It also handles all dependencies to Scala libraries.

To install `SBT` follow the instructions provided here: http://www.scala-sbt.org/0.13/docs/Installing-sbt-on-Linux.html.

1.3 Building Lift

After downloading the *Lift* repositroy and installing all dependencies (either directly or via docker) we will build the *Lift* compiler via `sbt` and to run all unit test to check if *Lift* is operating as expected.

1.3.1 Building the *Lift* Compiler

To build the *Lift* compiler simply run the following command from the repositroy root directory:

```
> sbt compile
```

When executed the first time, this will:

1. Bootstrap Scala by downloading the required Scala version;
2. Download the `arithexpr`[4] library used by the main *Lift* compiler;
3. Compile the native library used by *Lift* to communicate with OpenCL;
4. Compile the Scala and Java files implementing the *Lift* compiler.

1.3.2 Running all unit tests

To run all unit tests simply run:

```
> sbt test
```

This might take a few minutes and should run without any tests failing.

You can read more about the unit tests in the section about testing.

1.3.3 Configuring the IntelliJ IDEA IDE (optional)

If you want to use an integrated development environment (which might be a good choice if you are new to *Lift* or maybe Scala altogether) we recommend IntelliJ IDEA from Jetbrains.

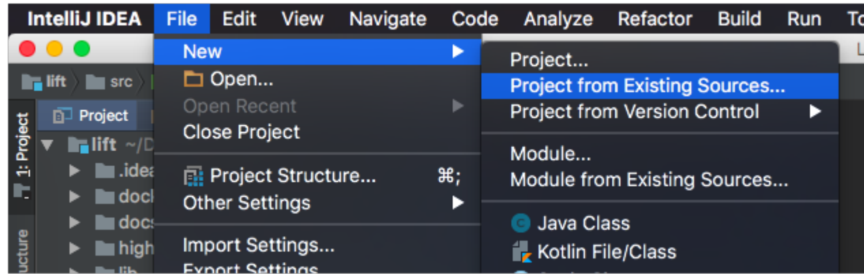
There is a free *Community* edition of IDEA available to download from Jetbrains website. It is important to install the Scala plugin, which is possible as part of the first launch setup of IDEA.

To configure the *Lift* project in the IDE perform the following steps:

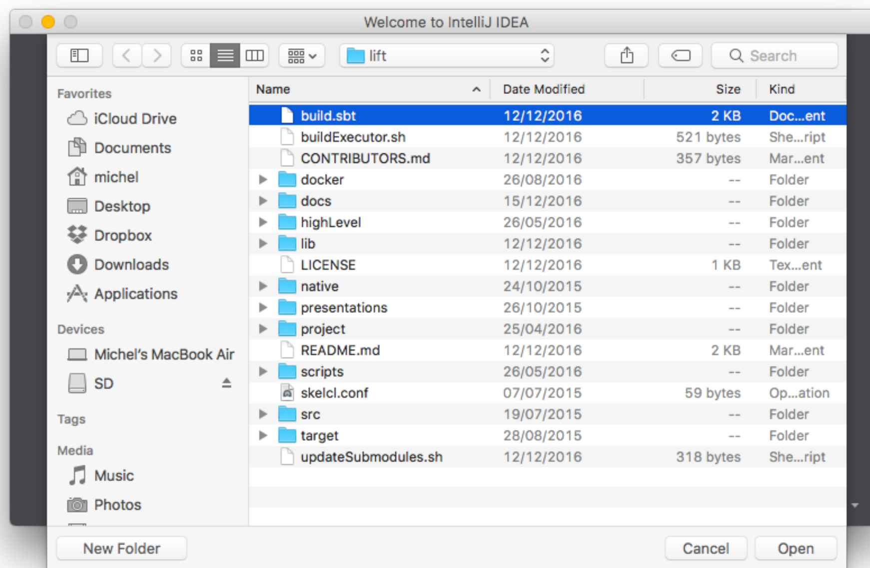
1. Launch IDEA and select `"Import Project"` from the launch screen:

[4] The `arithexpr` library is a Scala library developed by the *Lift* team and also hosted on github at https://github.com/lift-project/arithexpr.

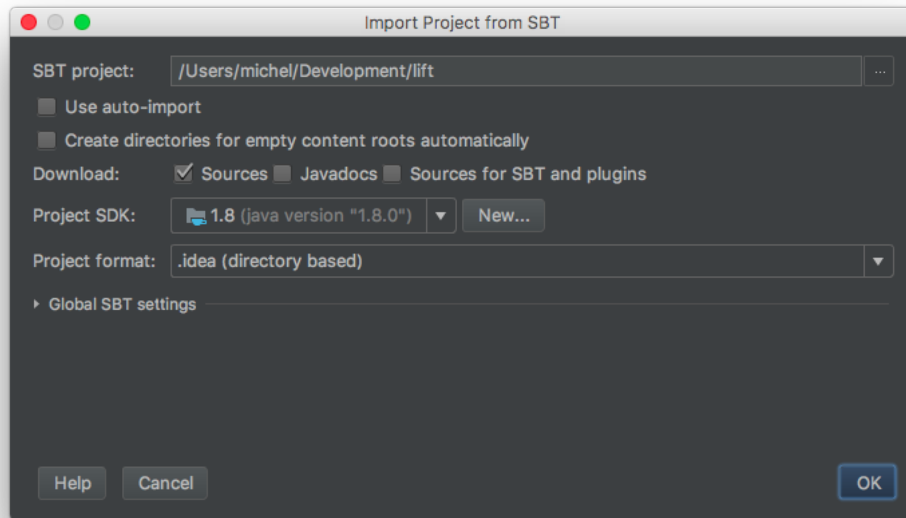
Or select "`File -> New -> Project from Existing Source ...`":



2. Select the `build.sbt` file at the root of the *Lift* repositroy:



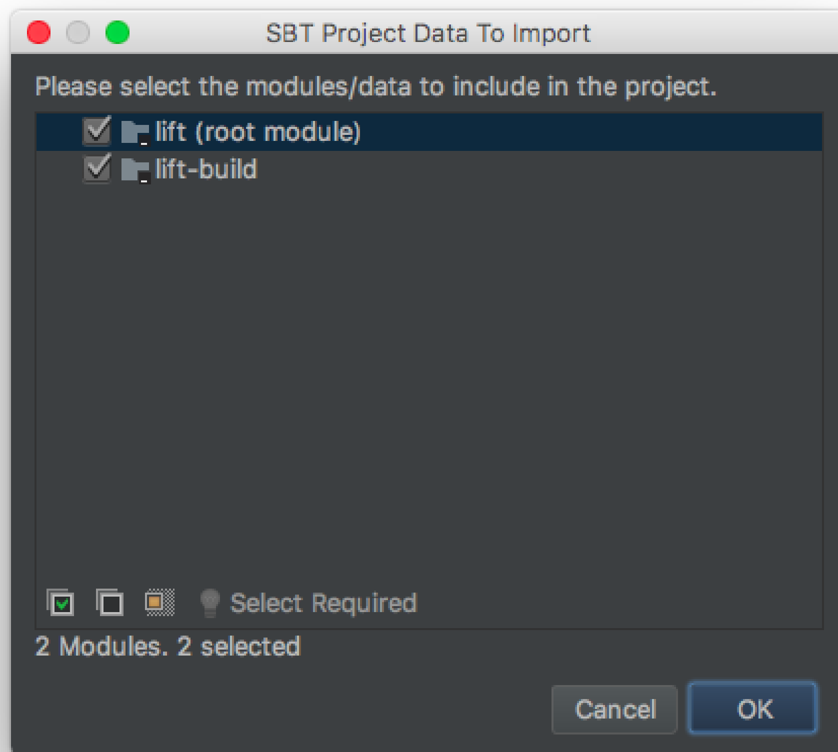
3. In the appearing "`Import Project from SBT`" dialog ensure that there is a *Project SDK* selected with at least *Java version* 1.8:



If this is not the case click `"New ..."` next to *Project SDK* and select a Java SDK of version 1.8 or newer installed on your machine[5].

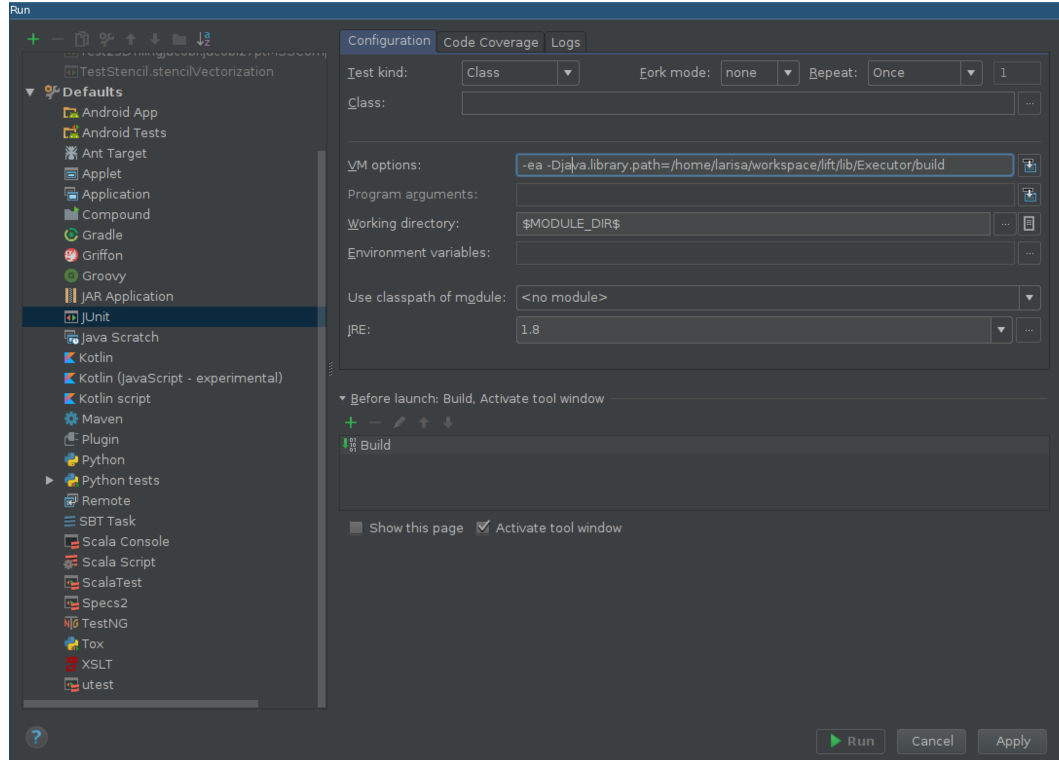
4. Click `OK` and also `OK` in the following dialog `"SBT Project Data To Import"`:

[5] You can read more about configuring SDKs in IDEA at https://www.jetbrains.com/help/idea/2016.3/sdk.html.



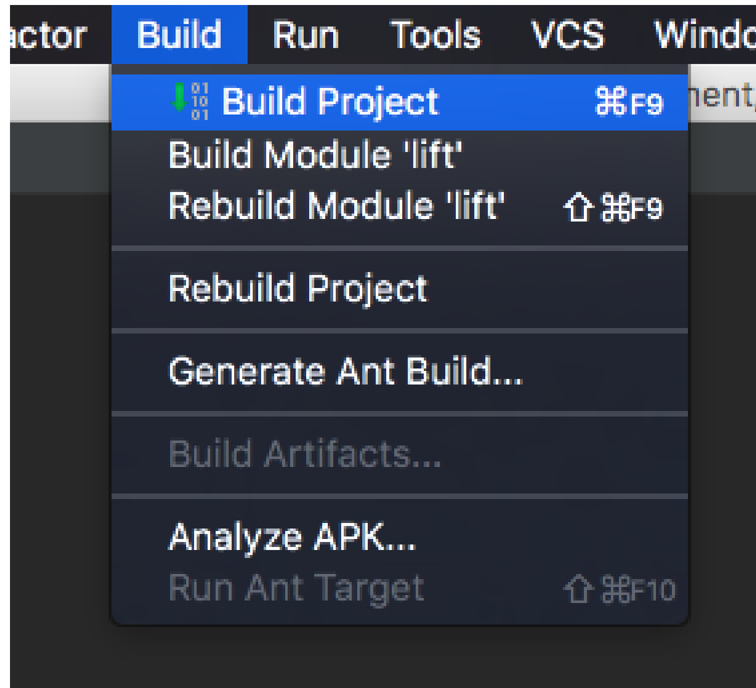
6. **Click "`Run -> Edit Configurations`" and then "`Default -> JUnit`" on the left pane of the dialog and ad**

   ```
   `-Djava.library.path=$LIFT_ROOT/lib/Executor/build/" (with the
   proper path from your system)`
   ```



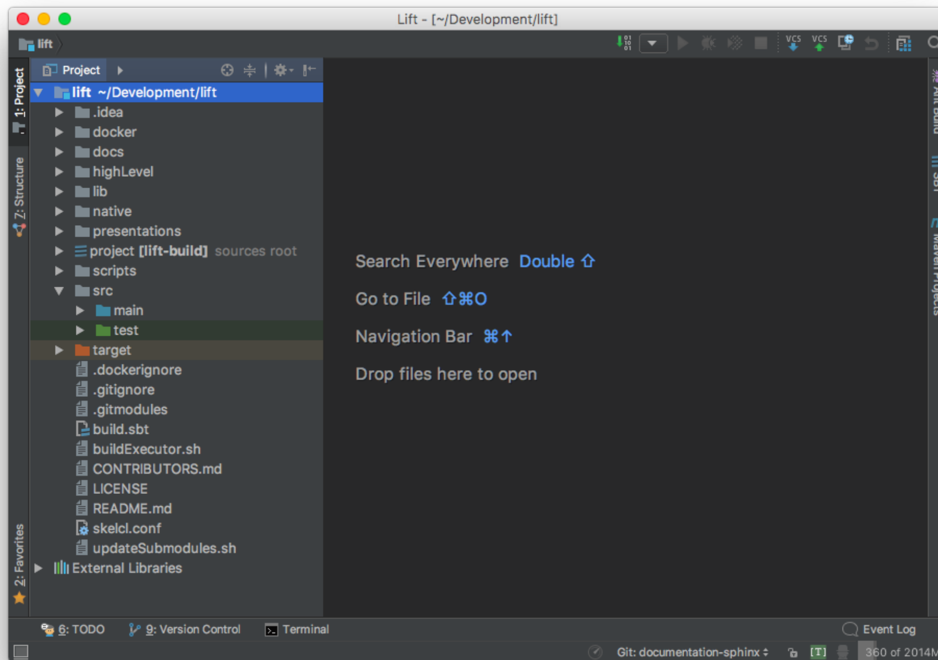
Alternatively, the `Executor` build directory can be added directly to your *LD_LIBRARY_PATH*.

6. Click "`Build -> Build Project`" to build the *Lift* compiler:

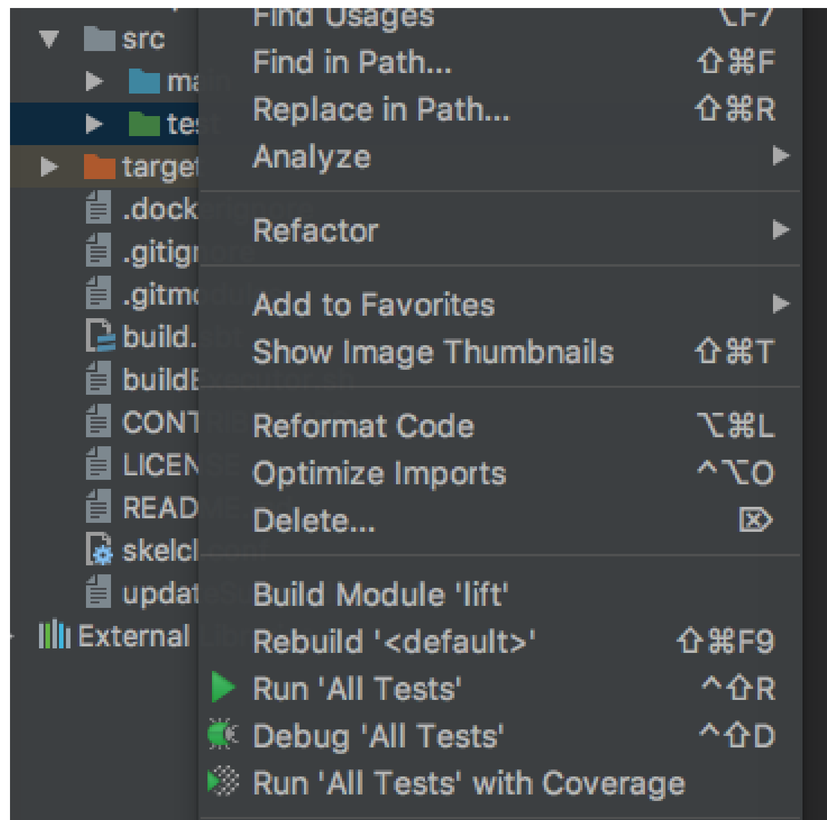


This is the equivalent action to `sbt build` on the command line.

7. To run all unit tests unfold the project structure as shown in the picture:



Then right click on the `test` folder and select `"Run 'All Tests'"`:



This is the equivalent action to `sbt test` on the command line.

There is exhaustive documentation material for IDEA available at https://www.jetbrains.com/help/idea/2016.3/meet-intellij-idea.html, if you are unfamiliar with the IDE.

Links

CHAPTER 2

Overview of *Lift*

2.1 Patterns

2.1.1 Algorithmic Patterns

`Map` $: (a \rightarrow b) \rightarrow [a]_n \rightarrow [b]_n \equiv$

$$map\ f\ [x_1, \ldots, x_n] = [f\ x_1, \ldots, f\ x_n]$$

The `map` pattern

`Zip` $: [a]_n \rightarrow [b]_n \rightarrow [(a, b)]_n \equiv$

$$zip\ [x_1, \ldots, x_n]\ [y_1, \ldots, y_n] = [(x_1, y_1), \ldots, (x_n, y_n)]$$

The `zip` pattern

`Reduce` $: ((a, a) \rightarrow a) \rightarrow a \rightarrow [a]_n \rightarrow [a]_1 \equiv$

$$reduce\ (+)\ 0\ [x_1, \ldots, x_n] = [0 + x_1 + \ldots + x_n]$$

The `reduce` pattern

`ReduceSeq` $: ((a, b) \rightarrow a) \rightarrow a \rightarrow [b]_n \rightarrow [a]_1 \equiv$

$$reduceSeq\ (+)\ 0\ [x_1, x_2, \ldots, x_n] = [(\ldots((0 + x_1) + x_2) + \ldots + x_n)]$$

The `reduceSeq` pattern

`Split` $: n \rightarrow [a]_m \rightarrow [[a]_n]_{m/n} \equiv$

$$split\ n\ [x_1, \ldots, x_m] = [[x_1, \ldots, x_n], \ldots, [x_{m-n}, x_m]]$$

The `split` pattern

`Join` : $[[a]_n]_m \rightarrow [a]_{n\times m} \equiv$

$$join\ [[x_1,\ldots,x_n],\ldots,[x_{(m\times n)-n},x_{n\times m}]] = [x_1,\ldots,x_{n\times m}]$$

The `join` pattern

`Iterate` : $n \rightarrow (m \rightarrow [a]_{k\times m} \rightarrow [a]_m) \rightarrow [a]_{k^n\times l} \rightarrow [a]_l \equiv$

$$iterate\ n\ f\ xs = iterate\ (n-1)\ f\ (f\ xs)$$
$$iterate\ 0\ f\ xs = xs$$

The `iterate` pattern

`Reorder` : $((Int, Type) \rightarrow Int) \rightarrow [a]_n \rightarrow [a]_n \equiv$

$$reorder\ idx\ xs = ys$$

The `reorder` pattern

`Transpose` : $[[a]_m]_n \rightarrow [[a]_n]_m \equiv$

$$transpose\ [[x_{1,1},\ldots,x_{1,m}],\ldots,[x_{n,1},\ldots,x_{n,m}]] = [[x_{1,1},\ldots,x_{n,1}],\ldots,[x_{1,m},\ldots,x_{n,m}]]$$

The `Transpose` pattern. Added for convenience, can be implemented using `Join`, `Reorder` and `Split`.

2.1.2 OpenCL Patterns

`MapGlb` : $(a \rightarrow b) \rightarrow [a]_n \rightarrow [b]_n \equiv$

$$mapGlb = map$$

The `map` pattern for mapping work onto global threads.

`MapWrg` : $(a \rightarrow b) \rightarrow [a]_n \rightarrow [b]_n \equiv$

$$mapWrg = map$$

The `map` pattern for mapping work onto work-groups (groups of threads).

`MapLcl` : $(a \rightarrow b) \rightarrow [a]_n \rightarrow [b]_n \equiv$

$$mapLcl = map$$

The `map` pattern for mapping work onto local threads.

`MapSeq` : $(a \rightarrow b) \rightarrow [a]_n \rightarrow [b]_n \equiv$

$$mapSeq = map$$

The `map` pattern for mapping work sequentially.

`PartRed` : $((a, a) \rightarrow a) \rightarrow a \rightarrow m \rightarrow [a]_{m\times n} \rightarrow [a]_m$

The `PartRed` pattern. Performs a partial reduction to size `m`.

`toGlobal` : $(a \rightarrow b) \rightarrow (a \rightarrow b)$

The `toGlobal` pattern

`toLocal` : $(a \to b) \to (a \to b)$

The `toLocal` pattern

`toPrivate` : $(a \to b) \to (a \to b)$

The `toPrivate` pattern

`asVector` : $m \to [a]_{m \times n} \to [\langle a \rangle_m]_n$

The `asVector` pattern.

`asScalar` : $[\langle a \rangle_m]_n \to [a]_{m \times n}$

The `asScalar` pattern.

`Vectorize` : $m \to (a \to b) \to (\langle a \rangle_m \to \langle b \rangle_m)$

The `Vectorize` pattern.

2.2 Rewrite Rules

2.2.1 Algorithmic Rewrite Rules

TODO: distinguish axioms from theorems

map(f) o map(g) == map(f o g)

reduceSeq(z, f) o mapSeq(g) == reduceSeq(z, (acc,x) -> f(acc, g(x)))

reduce(z, f) == reduce(z, f) o partialReduce(z, f)

partialReduce(z, f) == join o map(partialReduce(z, f)) o split | if f is associative and z is the neutral element (i.e. f(z,x) = x)

New rules (possibly unimplemented/unverified)

reduce(z, map(f) o zip) == map((e) -> (reduce(e._0, f) (e._1))) o zip(z) o T

map(map(f) o T) o slide(step,size) == map(map(f) o slide(step,size)) o T

2.2.2 OpenCL Rewrite Rules

New rules (possibly unimplemented/unverified)

mapSeq(f) o slide(step,size) == slideSeqPlus(step,size,f)

2.3 Compilation Flow

Compiling a program from the *Lift* language to OpenCL has the following steps[1]:

1. Type Analysis
2. Memory Allocation

[1] Lift: A Functional Data-Parallel IR for High-Performance GPU Code Generation: http://www.lift-project.org/papers/steuwer17LiftIR.pdf

3. Multi-Dimensional Array Accesses
4. Barrier Elimination
5. OpenCL Code Generation

2.4 Inferring OpenCL Thread Counts

To automatically determine how many threads to launch for a rewritten program, or as a convenience, the following simple algorithm is used.

The algorithm tries to eliminate as many loops resulting from the different `Map` patterns as possible. It'll choose thread counts to match the data sizes being mapped over. So, e.g. a `MapGlb` over an array of length `N` would get N threads in that dimension (and the local size would be undefined). similarly the number of work-group for `MapWrg` and the number local threads for `MapLcl` are determined. If there are several `MapLcl` in the same dimension, the most common thread count will be chosen, e.g. if there are 3 `MapLcl` in dimension 0, mapping over arrays of lengths 32, 16 and 32, 32 will be chosen. If there is no `MapGlb` or `MapWrg` and `MapLcl` pair in some dimension, a thread count of 1 will be chosen no parallelism is present.

CHAPTER 3

Developing *Lift*

3.1 Testing

There exists a large set of tests. These are regularly run by a continous integration (CI) server as part of the software development process.

A few tests with long execution time are not run by default, but these tests are still run by the CI server for pull requests.

Every single commit into the `master` branch **must** pass **all** tests.

3.1.1 Running tests

The test suite can either be run from the command line or via the IDE.

- To run the default set of tests from the command line execute the following command from the Lift root directory:

```
> sbt test
```

- To run the tests in the IntelliJ IDEA IDE, right click on the `src/test` folder and select `Run 'All Tests'`.

More verbose output can be enabled by setting the `LIFT_VERBOSE` environment variable.

NB: if you encounter a `lift.arithmetic.NotEvaluableException` the compiler will not print the stack trace which can be annoying for debugging. This is because this exception implements `ControlThrowable` (which itself implements `NoStackTrace`) for performance reasons. If you want to temporarily enable the stack trace, change `ControlThrowable` to `Throwable` in the definition of the exception and also change `val` to `def` in the companion object otherwise you will always get the same stack trace. Now you should have a more helpful exception.

3.1.2 Tests with long execution time

Some tests for the rewriting system can take several minutes to run and are therefore disabled by default. They can be included by setting the `LIFT_LONG_TESTS` environment variable and rerunning the tests using `sbt test` or the

IDE. Tests are marked as a long running one, by calling the `opencl.executor.LongTestEnabled()` function in the test method or the entire class.

3.1.3 Ignoring tests

Tests related to issues which are not yet resolved are marked with the `@Ignore` annotation. This annotation should be used only as an exception.

3.1.4 Tests for particular architectures

Some tests use hardware features only available on specific architectures or trigger bugs on some architectures. The methods in `org.junit.Assume` are used to ignore these based on some condition.

For example:

- The device might not be able to use double precision;
  See for example test `opencl.generator.TestMisc.testDouble`.
- The test uses warp sizes specific to NVIDIA GPUs;
  See for example test `opencl.generator.TestReduce.NVIDIA_C`.
- The compiler for some architectures generates wrong code;
  See for example test `opencl.generator.TestReduce.NVIDIA_B`.

3.2 Guidelines for Contributing

Although Lift is a primarily research focused project, treating it as the complex software project it is, and therefore maintaining a number of software development practices is in all our interests. With these practices, we aim to interrupt fast iteration of design and code as little as possible, while at the same time maintain sufficient stability (of APIs etc) to allow other users to develop features independently.

This section describes our core guidelines for working with this project, how new features and changes should be developed, and added to the core codebase.

3.2.1 Development workflow

This project operates along using fairly standard branch, pull request, and merge git methodology. In essence, all changes to the project should be made in branches of the project, which will be merged back into the master branch after acceptance via pull request. This process should be lightweight enough that creating new features requires as little effort as possible, but restrictive enough that large breaking changes are rare, and occur predictably enough that other users can comment on, and adapt to them.

As a general guideline we prefer short living branches which have a clear purpose and are merged back quickly into master. This style should be – whenever possible – favored over long living branches *as explained below*.

Basic workflow overview

The basic workflow for adding a single small feature (e.g. a standalone benchmark, without any compiler modifications) to the lift project is as follows:

1. Create a feature branch from the master branch, i.e.:

```
git checkout master && git checkout -b feature_branch
```

2. Make changes within the feature branch.
3. Make sure all tests pass locally, and on the continuous integration server.
4. Open a pull request on bitbucket[1] , from the feature branch (`feature_branch`) to master.
5. Wait for comments, and if positive, merge it in.

Bitbucket has a very nice tutorial[2] explaining the workflow in more detail.

Links

Motivation for avoiding long living development branches

Unfortunately, it is rare that we have the luxury of producing one small standalone feature/piece of work, which can be easily described in a single branch, and developed separately. More often than not, we have a separate larger project (e.g. a paper that we're writing) that has multiple separate interacting components that we wish to develop. In this case, although we might want to develop on a single branch, and push all the changes once we're done, that can be frustrating for other users, as the branch may contain a large number of breaking changes that are difficult or time consuming to merge and fix.

We therefore advise that *where possible*, all changes should be broken out into separate feature branches as soon as possible, and pull requests to master submitted. For example, say I have a branch `big_paper`, which is up to date with master (i.e. is no commits behind master, and several ahead), and a feature that could be separately added to the master branch. In this instance, we would prefer that the changes that make up such a feature are added to a separate branch from `big_paper`, and a pull request submitted to master.

The alternative, submitting a large pull request at the end of a project, is unfavorable, as it produces a lot of work for other developers, and often involves lots of complex changes. In these instances, we would request that at the end of a large project, a developer splits any new additions into separate feature branches, which then all have independent pull requests opened. This creates more work on the behalf of the feature developer, yes, but allows maintainers to more selectively and carefully accept or delay features that might cause issues to the project.

Pull requests

All changes and additions to the Lift project must be done in the form of a pull request from a branch or fork. In addition, we are fairly opinionated with regards to what makes a "good" pull request, but also flexible enough to recognise that as this is a research project, a "good" pull request won't always be possible to produce.

As a general guideline, we don't want pull requests that are "too big". What this means, exactly, is fairly hazy, but in general a pull request should:

1) Add a single feature, or change, to the project (features include the associated tests and minor changes that they might require). This could be a change in how we generate code for a parallel pattern, the addition of a new pattern, a benchmark (without any new changes to the compiler - they should have been separate pull requests previously), or a new pattern.
2) Be small enough, or contain little enough code, that a reviewer can read and understand it within an hour. This is an arbitrary deadline, that shouldn't be taken seriously, but which sets the tone for the amount of code that is "too much".

Of course, in cases where a number of features, or changes, are highly coupled, then it is inevitable that they will all be submitted together. In this case, simply be prepared for the pull request to take longer to be approved.

[1] https://www.atlassian.com/git/tutorials/making-a-pull-request
[2] https://www.atlassian.com/git/tutorials/comparing-workflows/feature-branch-workflow

Breaking changes

In some cases, a pull request will contain code that fundamentally changes API, or some other structure of the code that may potentially impact the work of other developers. In this case, our aim is to minimise this impact, by notifying other developers, and delaying the merge until we are sure that it is necessary. As a developer you can also help by reducing the size of the pull request as much as possible to mitigate any potential breakages. Of course, this is not always possible and we do recognise the importance of breaking code to move forward.

3.3 How to . . .

3.3.1 Read me first

These guides try to clarify how and where things are done in Lift, what files you have to modify if you want to extend the language, what class needs to be extended, etc.

At each step during this process, you **should** look at the existing stuff, find some code which looks close to what you want to implement and take this as a starting point.

Good luck ;)

3.3.2 . . . add a new OpenCL pattern

Define an AST Node

Each pattern is a case class extending `ir.ast.Pattern`. It has to be defined in the `opencl.ir.pattern` package. Take a look at the `FPattern` and `isGenerable` traits because you may want to implement them.

When extending `Pattern` you should also provide the number of arguments your pattern expects.

The `FPattern` trait is used when pattern matching to catch all patterns that contain a `Lambda` inside them.

After defining you pattern, the first thing you have to do is override the `checkType` method which:

1. Checks that type of your pattern's arguments match its expectations.
2. Checks the types of any nested functions based on the pattern's input type.
3. Returns the type of your pattern's output, determined by the types its inputs or the nested functions.

The `MapSeq` pattern can be implemented like this:

```
/**
 * Sequential map
 *
 * @param f a function to be applied to every element of the input array
 */
case class MapSeq(f: Lambda1) extends Pattern(arity=1)
                              with FPattern
                              with isGenerable {

  override def checkType(argType: Type, ...): Type =
    argType match {
      case ArrayType(ty, len) => {
        // f has type: `argType -> a`
        f.params.head.t = argType
        // map(f) has type `[argType],,n,, -> [a],,n,,`
```

(continues on next page)

(continued from previous page)

```
        ArrayType(TypeChecker.check(f), len)
      }
      case _ => // throw an exception
    }
}
```

Visiting the Pattern

Some operations need to "visit" all nodes of an expression (which may contain your pattern). You should take a look at the `visit*` and `replace` methods in `ir/ast/Expr.scala` to ensure you pattern is visited correctly.

Note: if you implement `FPattern`, it may work out of the box. See how `ReduceWhileSeq` is handled as a typical example if you have to do extra work.

Code Generation

Then you have to implement the code generation for your pattern. Look at the `OpenCLGenerator.generate` method in the `opencl.generator` package. It takes a type-checked lambda (e.g. an expression that can contain an instance of your pattern) and returns a string (the OpenCL code).

Here are the different steps of this process.

OpenCL Address Spaces

OpenCL has a hierarchy of different address spaces: global memory, local memory and private memory. You need to specify where to store the output of your pattern. If you extend `FPattern`, it might work out of the box without changing anything. But you should take a look at `InferOpenCLAddressSpace.setAddressSpaceFunCall`, see how the other patterns are handled and probably add a case for your pattern.

Ranges and Counts

If your pattern generates a for loop in the OpenCL code, it should contain a "loop variable" in its definition. The loop variable determines where the pattern accesses its input array in every iteration of the generated loop.

Should your pattern contain a "loop variable", you should add a case in the `RangesAndCounts.apply` method. See how `MapSeq` is handled for example.

Memory Allocation

Only user functions allocate memory. You have to tell the nested user functions how much memory they will need if they are nested in your pattern.

It is done in `OpenCLMemoryAllocator.alloc`. More specifically, you have to add a case in `allocFunCall`.

All these functions take an argument `inMem` which is the memory, OpenCL buffer(s), that the first user function in the pattern will read from.

Also, some other classes are called during this process. You may need to edit `OpenCLMemory` as well.

Unrolling loops

`ShouldUnroll` isn't really an optimisation at this point. We have to unroll loops when we use private memory as we flatten all private memory arrays into variables and can't therefore index into them using variables. We represent the private arrays as variables to try and force the compiler to store the arrays in registers instead of spilling into global memory. That means instead of a definition `int a[2];` we have `int a_0; int a_1; int a_2;` and for reading/writing `a[i]` when `i` is `1` we need to emit `a_1` instead.

Barrier elimination

`BarrierElimination` is an optimisation consisting in removing unnecessary barriers in the OpenCL generated code. Don't look at that in the first place.

Debugging

The `if (Verbose()) { ... }` part of the `generate` function prints some helpful information, like types, that is displayed if you set the `LIFT_VERBOSE` environment variable. It will work out of the box if you have correctly extended `Expr`'s methods (see "Visiting your pattern above)

The Views

The views are used to determine the locations in arrays where user functions in any expression read from and write to.

Add support for your pattern in `InputView.buildViewFunCall` and `OutputView.buildViewFunCall`. This means "explaining" how your pattern modifies the reading/writing locations of nested functions.

Some Plumbing

The definition of tuple types and user functions should work out of the box.

Generating the Kernel

Once all of the above passes have been implemented, you are able to generate `OpenCLAST` nodes. This is done in `generateKernel` but you probably do not have to edit this function and should directly look at the private `generate` method of `OpenCLGenerator`. Add a case for your pattern.

It is probably a good idea to take a look at the classes defined in `OpenCLAST.scala` and at the utility functions like `generateForLoop` defined at the end of `OpenCLGenerator.scala`.

Testing

You have to check that your pattern works as expected. For that add a test class in the test folder in the `opencl.generator` package with some tests.

For example, for `MapSeq`, you could have:

```
object TestMapSeq {
  @BeforeClass def before(): Unit = {
    Executor.loadLibrary()
    println("Initialize the executor")
    Executor.init()
  }

  @AfterClass def after(): Unit = {
    println("Shutdown the executor")
    Executor.shutdown()
  }
}

class TestMapSeq {
  @Test def simpleMap(): Unit = {
    val size = 1024
    val input = Array.fill(size)(util.Random.nextInt)
    val N = SizeVar("N")

    val add2 = UserFun("add2", "x", "return x+2;", Int, Int)

    val kernel = fun(
      ArrayType(Int, N),
      array => MapSeq(add2) $ array
    )

    val (output: Array[Int], _) = Execute(size)(kernel, input)

    assertArrayEquals(input.map(_ + 2), output)
  }
}
```

Useful tips

- Use the debugger to compare what you have in your pattern and in an already existing one at different points in the compilation process.
- Look at the generated OpenCL code. To see it, enable the verbose output by setting the `LIFT_VERBOSE` environment variable to `1`.
- Try to have something that compiles as soon as possible even if works only in some specific situations. It is easier to start from a simpler version of your pattern and then extend it.

3.3.3 . . . switch branch

To switch/create/delete branches, you can either use IntelliJ or the command line using the following command:

- Create a new branch at the current commit (and switch to it):

```
> git checkout -b BRANCHNAME
```

- Create a new branch configured to follow a remote branch:

```
> git checkout -b BRANCHNAME origin/REMOTE_BRANCHNAME
```

- Switch to an existing branch:

```
> git checkout BRANCHNAME
```

NB: If BRANCHNAME does not exist locally, git will try the following:

```
git checkout -b BRANCHNAME origin/BRANCHNAME
```

- Delete a branch:

```
git branch -d BRANCHNAME
```

NB: you can only create and delete local branches this way.

Try to always stay up-to-date with bitbucket using the `fetch` command before any switching to get the remote information about the repository. This can be done directly in IntelliJ or via the command line:

```
> git fetch
```

When switching from one branch to another, it is important to ensure that the `ArithExpr` submodule is being updated. Since IntelliJ does not support well submodules, you should type on the command line:

```
> git submodule update
```

After having switched branches. Note that:

```
> ./updateSubmodules.sh
```

Will do the same but will show more detailed information.

CHAPTER 4

Generating and Running Kernels

4.1 Generating Kernels

After having set up and built *Lift* navigate to the root folder of the repository and create the wrapper scripts for running the compiled programs:

```
scripts/buildRunScripts.py
```

You can now run all stages of the rewrite system for a program (in this example for matrix multiplication):

```
scripts/compiled_scripts/HighLevelRewrite highLevel/mmTransposedA
scripts/compiled_scripts/MemoryMappingRewrite --gr10 mmTransposedA
scripts/compiled_scripts/ParameterRewrite -f highLevel/mm.json mmTransposedA
```

The generated OpenCL code for the application is now in the `mmTransposedACl` folder along with `exec_*.csv` files saying what thread-counts and memory sizes to use for executing them.

The results from the intermediate stages can be seen in the `mmTransposedA` and `mmTransposedALower` folders.

4.2 Running Kernels

To run them clone the harness from https://github.com/lift-project/harness, build it using `cmake` and add the harness programs to your `PATH`:

```
git clone https://github.com/lift-project/harness.git
cd harness
mkdir build && cd build
cmake ..
make
export PATH=`pwd`:$PATH
```

To run all program variations change to the mmTransposedACl folder and use the following command, substituting the desired platform and device numbers.:

```
for i in `seq 1 250`; do find . -mindepth 1 -type d -exec sh -c '(cd {} && timeout 5m
↪harness_mm -k 1024 -n 1024 -m 1024 --transpose-A -d $DEVICE -p $PLATFORM)' ';'; done
```

`-k 1024 -n 1024 -m 1024` indicate the sizes to use for the inputs and can also be adjusted.

The runtimes for the kernels will be stored in `time_*.csv` files.

4.3 High-Level Rewrite

This stage performs algorithmic rewriting of the input program.

4.3.1 Filtering Heuristics

Expression Nesting Depth

Counts how deeply *Map*/*Reduce* patterns are nested inside the program. The deepest nesting is reported.

Some examples:

```
\(input => plusOne $ input) // Depth 0

\(input => Map(plusOne) $ input) // Depth 1
\(input => Map(plusOne) o Map(plusOne) $ input) // Depth 1
\(input => Reduce(add, 0.0f) $ input) // Depth 1

\(input => Map(Map(plusOne)) $ input) // Depth 2

\(input => Map(Map(Map(plusOne))) $ input) // Depth 3
\(input => Map(Map(plusOne) o Join() o Map(Map(plusOne))) $ input) // Depth 3
\(input => Map(Map(Reduce(add, 0.0f))) $ input) // Depth 3
```

Adjusted using the `--depth` command line option.

User-Function Distance

Tries to evaluate how well the rewritten program has simplified by counting the number of data-layout patterns on the data-flow path between user-functions. Simplification here refers to removing superfluous data-layout patterns (*Split*, *Join*, *Scatter*, *Gather*, *Transpose*, *TransposeW*, *asVector* and *asScalar*). The largest count is reported.

Some examples:

```
\(input => Map(plusOne) $ input) // Distance 0, only one user-function

\(input => Map(add) $ Zip(Map(plusOne) $ input, Map(plusOne) $ input)) // Distance 0

\(input => Map(plusOne) o Join() o Map(Map(plusOne)) $ input) // Distance 1, Join

\((input2D, input1D) =>
  Map(add) $ Zip(Join() o Map(Map(plusOne)) $ input2D, Map(plusOne) $ input1D)) //
↪Distance 1

\(input => Map(Map(plusOne)) o Split(x) o Join() o Map(Map(plusOne)) $ input) //
↪Distance 2, Split and Join
```

Adjusted using the `--distance` command line option.

CHAPTER 5

Indices and tables

- genindex
- modindex
- search