Compiler Enhancements to Reverse AD

The lecture material of DPP'21 has had a strong focus on the compiler transformations that are at the foundation of various parallel programming models, for example, flattening of all kinds (regular, "incremental" and irregular), polyhedral optimizations, automatic differentiation, etc. To keep consistent with the curriculum, it is only natural to try to suggest some projects that aim at the implementation of such code transformations. In previous editions of DPP, we have deemed such types of projects unfeasible or highly-risky, because compiler implementations typically take a much longer time than, say, prototyping a technique in a low/high level language, and the students have roughly one month (or less) to work on such a project---which is typically too little time for the task.

However, the work on reverse-mode AD has opened up some compiler extensions that we deem feasible to achieve even in such a short time frame:

(1) On the one hand, this is because reverse-mode AD is implemented towards the front-end of the compiler, which means that you will be playing with a representation that is morally close to the source language.

(2) On the other hand, this is because the said extensions refer to a large extent to treating more efficiently special cases of reduces and scans---i.e., special operators such as multiplication, vectorized addition---and the rationale is already presented in the anonymously-authored paper available on Absalon.

(3) On the third hand, this is because such cases are already implemented on a different branch that will be named if/when you get stuck (hence the risk is small or at least navigable).

(4) Finally, we promise to be much more lenient with these type of (compiler) projects, because we know that no matter how simple they look, they are still non-trivial because you need to first learn about the compiler representation and the surrounding code (and available functionality).

If you happen to chose one of these projects, we suggest you make your own branch of the Futhark branch named ***ad-genred-opt*** and then you quickly run

$ stack haddock

that will generate the documentation, without which you would be living on borrowed time.

The transformation is implemented in folder **src/Futhark/AD,** both forward and reverse mode. In some sense, the entry point for the reverse mode is the **Rev.hs** file, but the re-write rules for various second-order array combinators (SOACs) are implemented in folder **Rev/** in the file bearing the corresponding name, for example **Reduce.hs** and **Scan.hs**.

We suggest that you start by taking a look at some relevant examples in folder **tests/ad/** that you may compile and see the code that is being produced by:

futhark dev -s tests/ad/scan0.fut

Taking a closer look to **scan0.fut** also reveals an “easy” way to validate your code: you may compute the whole Jacobian in both forward and reverse mode, and validate your enhancements with respect to the forward mode (which is deemed simpler and safer). You are of course encouraged to add and test your own code examples there.

Please note that a file named **Reduce-by-Index.hs** does not exist yet. If you wish to work on some special cases of the **reduce-by-index**:

(1) you either have to start by adding some stub code in file **SOAC.hs** (and perhaps **Rev.hs** as well), and then providing your implementation in a new file, say **Rev/Reduce-by-Index.hs**,

(2) or you may just tell us and we will add a stub for you.

Finally, after such a long opening, here are some of the potential enhancements:

(1) Currently, the general case of **reduce** is implemented, and also the special case when the binary operator is min/max. One enhancement is to add a case when the operator is scalar multiplication. Another potential avenue of enhancements is to specially treat the cases when the binary operator is vectorized plus or vectorized multiplication or a vectorized min/max, i.e., map2 (*), map2 (+), map2 f32/64.min/max. However, first check whether the current “generic” treatment of such cases is not already simplified to exactly what you would like to generate.

(2) **Reduce-by-index** is not implemented yet at all in said branch (albeit the special cases are implemented on another branch---do not waste your time trying to find it, just ask and we will tell you). You may chose to implement any of the reduce-by-index special cases, namely, plus, multiplication, min/max, vectorized addition, etc.

(3) **Scan** is a tricky one. Currently it works correctly only when the binary operator receives two scalars as arguments. You first have to understand the code in relation to the high-level re-write rule presented in the paper. The enhancement can simply be to treat the case of vectorized operators, e.g., scan (map2 (+)) or scan (map2 (*)).
The high-level rationale is also presented in the paper, in that

scan (**map2** bop) (**replicate** n e_bop) arr-2d

is equivalent with

transpose <| **map** (**scan** bop e_bop) <| **transpose** arr-2d

hence, if we know how to treat the scalar-case of scan, then we should be able to derive the vectorized operator as well. The crudest and quickest implementation would be to simply re-write the **scan (map2 op)** as above and differentiate the resulting code.

We suggest that you start implementing the enhancement that you feel most attracted to, but perhaps you start with a simple one first for warm up, and then you go to the next one, if time permits. We do not have a number in mind, just go one by one and see how many you have time to do. We expect at least one enhancement, two would be excellent, everything else is simply amazing.

We do however expect you to perform a thorough testing in order to validate your implementation, and also a thorough job aimed at the most-efficient translation possible---not only in terms of work-depth asymptotic (it goes without saying that the asymptotic of the original program must be respected), but also to the constant factors, i.e., reasoning whether the resulted code is fusible and how many global-memory accesses is your code going to perform and such.

Remember that sometimes “less is more”, and in what compiler work is concerned “less and thorough” always trumps “more and negligent”.