Memory Reduction for Scientific Computing on GPUs

Background

Currently, the LES uses a large number of intermediate arrays which are stored in the global memory.

I contend that we only need the actual input arrays and we can recompute all the rest from those. So rather than writing back to memory, we store what we need in a smart buffer.

For the LES we have u,v,w,p as inputs, but we actually use:

```
p (double)
u,v,w
usum,vsum,wsum
f,g,h
fold,gold,hold
sm
rhs
```

So my contention is that we could go from these 16 arrays to only 4 plus 12 smart buffers which only take 3/kp of the space, so in total:

4+12*3/kp = 4.45 for kp=80 so a reduction in memory of 3.6x, so we could go from the current domain size of 300x300 to 568x568

Current status

F77 -> F95 without globals F95 without globals -> OpenCL with maps and folds OpenCL with maps and folds -> TyTraCL & TyTraIR -> FPGA TyTraCL -> TyTraIR TyTraIR -> FPGA

Removing intermediate arrays

TyTraCL

The TyTraCL language is syntactically a subset of Haskell, with the addition of two types of vectors of fixed size.

Types

```
v_i :: Vec n a
s_i :: SVec k Int
f_i :: a -> b
```

Primitives

```
-- As in Haskell but on Vec rather than []
map : (a -> b) -> Vec n a -> Vec n b
-- Haskell's foldl but on Vec rather than []
fold :: (a -> b -> a) -> a -> Vec n b -> a

-- Ordinary map but only works on SVec
maps : (a -> b) -> SVec k a -> SVec k b

-- Every element of the vector is replaced by the stencil defined by the first argument
stencil : SVec k Int -> Vec n a -> Vec n (SVec k a)

-- Like Haskell's zip/unzip but takes a tuple as argument and works on Vec
zipt :: (...,Vec n a_i,...)  -> Vec n (...,a_i,...)]
unzipt :: Vec n (...,a_i,...) -> (...,Vec n a_i,...)

-- Apply a tuple of functions to a tuple of values
applyt :: (...,a_i -> b_i,...) (...,a_i,...) -> (...,b_i,...)
applyt (...,f_i,...) (...,e_i,...) =\(...,e_i,...) -> (...,f_i e_i,...)

-- Generalisation of Haskell's fst/snd
elt :: i:: Int -> (...,a_i,...) -> a_i
elt i (...,e_i,...) = e_i
```

Other TyTraCL primitives

```
split :: k -> Vec n a -> Vec k (Vec n/k a)
merge :: Vec k (Vec n/k a) -> Vec n a

select :: SVec k Int -> Vec n a -> Vec k a
select patt v = maps (\idx -> v !! idx) patt
-- Example:
v' = select [ i*jm*km+j*km+k_const | i <- [0 .. im-1],  j <- [0 .. jm-1], k <- [0..km-1] ] v

insert :: SVec k Int -> Vec k a -> Vec n a -> Vec n a
insert target_pattern small_list target_list

replicate :: Int k => k -> a -> Vec k a
```

Rewrite rules

```
map f_2 (map f_1 v) = map (f_2 . f_1) v

v = map id v

zipt (map f_1 v_1, map f_2 v_2) = map (applyt (f_1,f_2)) (zipt (v_1, v_2))

applyt (g_1,g_2) $ applyt (f_1,f_2)  = applyt (g_1 . f_1, g_2 . f_2)

(elt i) . unzipt . (map f (zipt (...,v_i,...))) = map ((elt i) . f) (zipt (...,v_i,...))

stencil s_1 (map f_1) = map (maps f_1) (stencil s_1)
```

Aim

The aim is to rewrite the program so that it is a single `map` call.

AST

Starting from the OpenCL Fortran code:

1. Scalarize all functions
2. Insert `stencil` calls as required
3. If a call requires a stencil of a previous call, insert a `maps` as required

Then we transform the resulting AST into a simpler AST for rewriting:

- The AST is a list of assignments:

  ```
  AST = [(LHS,RHS)]
  ```

- We define helper types:

  ```
  F-like = Function | ApplyT |  MapS
  V-like = Vector | ZipT | Stencil
  S-like = Scalar | Tuple
  ```

- Left-hand side:

  ```
  LHS =
      Scalar
    | Tuple
    | Vector
   | ZipT, a tuple of vectors
  ```

- Right-hand side:

```
RHS =
    Map F-like V-like
  | Fold F-like S-like V-like
  | Stencil SVec V-like
  -- In principle also allowed:
  | Vector Int S-like
  | ZipT
  | Scalar
  | Tuple
```

- Initially, all RHS nodes will be Map, Fold or Stencil.

Rewrite algorithm

Without `fold`

For every call, we look at the V-like argument first. If it is not a vector then it must be a ZipT or a Stencil. For a ZipT we do each arg in turn. In principle this maybe have to be done recursively until we hit an actual Vector. For that Vector we find the assignment where it is on the LHS. Then we check the type of the RHS and the type of the parent node. As our aim is to reduce the maps, we always apply rules that aim to transform a nest call into a single map.

Fortran-OpenCL implementation

```
! stencil
integer, dimension(size_s,2) :: s
real, dimension(size_s) :: v_s
do s_idx = 1,size_s
  v_s(s_idx) = v(j+s(s_idx,1),k+s(s_idx,2))
end do

! maps
real :: u_0
real, dimension(size_s) :: u_s
do s_idx = 1,size_s
  v_s(s_idx) = v(j+s(s_idx,1),k+s(s_idx,2))
  call f(u_0,v_s)
  u_s(s_idx)=u_0
end do
```

Reduction

- Pair up `applyt` with `zipt` and `maps` with `stencil`
- Reduce via SSA

Staging

However, this requires staging. For example, take the case of the rhs for the SOR.

This is calculated from p,u,v,w using a reduction, and then the reduced value is subtracted from the original.

So what we have to do is always recompute rather than store the intermediates in memory.

Proofs

```
s1 : SVec k Int
f1 : a -> b

stencil s1 (map f1) = map (maps f1) (stencil s1)

v0 : Vec n a
f1 : a -> b
v1 : Vec n b

v1s : Vec n (SVec k b)
v1 = map f1 v0
v1s = stencil s1 v1

v0s : Vec (SVec k a)
v0s = stencil s1 v0

maps f1 : SVec k a -> SVec k b
v1s : Vec n (SVec k b)
v1s = map (maps f1) v0s

or

stencil s1 (map f1 v1) = map (maps f1) (stencil s1 v1)
```

Dealing with halos

As we have

```
v_i :: Vec n a
```

we can define a to be a type that contains the halo information:

```
data Halo1D b = LIHalo b | RIHalo b |
                LOHalo b | ROHalo b |
                Core b
```

or for an original 2-D domain

```
data IHalo2D b = NIHalo b | EIHalo b | SIHalo b | WIHalo b |
                 NEIHalo b | SEIHalo b | SWIHalo b | NWIHalo b |
                 NOHalo b | EOHalo b | SOHalo b | WOHalo b |
                 NEOHalo b | SEOHalo b | SWOHalo b | NWOHalo b |
                 Core b
```

and for a 3-D domain, where we'd have 6 planes, 12 ribs and 8 corners

```
data IHalo3D b = NIHalo b | EIHalo b | SIHalo b | WIHalo b |  TIHalo b | BIHalo b | -- plane
                 TNIHalo b | TEIHalo b | TSIHalo b | TWIHalo b | -- top ribs
                 BNIHalo b | BEIHalo b | BSIHalo b | BWIHalo b | -- bottom ribs
                 NEIHalo b | SEIHalo b | SWIHalo b | NWIHalo b | -- vertical ribs
                 TNEIHalo b | TSEIHalo b | TSWIHalo b | TNWIHalo b | -- top corners
                 BNEIHalo b | BSEIHalo b | BSWIHalo b | BNWIHalo b | -- bottom corners
                 NOHalo b | EOHalo b | SOHalo b | WOHalo b |  TOHalo b | BOHalo b | -- plane
                 TNOHalo b | TEOHalo b | TSOHalo b | TWOHalo b | -- top ribs
                 BNOHalo b | BEOHalo b | BSOHalo b | BWOHalo b | -- bottom ribs
                 NEOHalo b | SEOHalo b | SWOHalo b | NWOHalo b | -- vertical ribs
                 TNEOHalo b | TSEOHalo b | TSWOHalo b | TNWOHalo b | -- top corners
                 BNEOHalo b | BSEOHalo b | BSWOHalo b | BNWOHalo b | -- bottom corners
                 Core b
```

and thus we can write e.g.

```
v_i :: Vec n (Halo3D a)
```

The opaque functions change also to

```
f_i :: Halo1D a -> Halo1D b
```

and inside the function we can pattern match on the type to decide what to do with the boundary points. Note that the type tells us which point is involved in a boundary calculation when updated, so these are the "inner halos" (`IHalo`),

i.e. for a 1-D case with n+2 points where the outer halos are 0 and n+1, the points with type LIHalo and RIHalo will be 1 and n, whereas LOHalo and ROHalo will be 0 and n+1.

The question is if instead I could do something Like

```
data Halo b = Halo (idx : Int) b
```

This would effectively mean that all points would have a unique type which carries the index. Within the context of TyTraCL this would solve all our problems. I'm not sure if this is really possible because the type information would not be known at compile time. Of course in the worst case I can always to something simpler and less elegant:

```
data IdxVal a = (a,Int) -- or is it newtype?

elt :: IdxVal Real
elt = (elt_val, elt_idx)

toIdxVal :: (a,Int) -> IdxVal a

hv = mapToIdxVal v

mapToIdxVal :: Vec n a -> Vec n IdxVal a
mapToIdxVal v = let
    idxs = 0 .. (length v - 1)
    tups = zip v idxs
  in
    map toIdxVal tups
```

What to do for a circular boundary? Say,

```
p(1,j,k) = p(ip+1,j,k)
```

The points `(1,j,k)` will be identified as the inner front plane, i.e. SIHalo. However, although in principle we can use stencils, in practice we can't as it would take too much memory. So the actual solution will have to be that we extract all planes with circular conditions from the stream. This requires a new TytraCL primitive which I will call `extract`

```
  extract :: SVec k Int -> Vec n a -> Vec k a
  -- possible Haskell implementation
  extract patt v = map (\idx -> v !! idx) patt

  south_plane = extract [ (im+1)*jm*km+j*km+k | i <- [0 .. im-1],  j <- [0 .. jm-1], k <- [0
```

So now the opaque function can take the extracted vector as a non-map argument.

```
v' = map (f_i south_plane) v
```

If the index information is accessible inside the opaque function, this can work. What this means in practice is that the TyTraIR will use the index information in the stream (which I guess is globally accessible) to decide when to index into the plane array.