Selecting Vulkan Queue Families in Haskell

idgd

1 Problem Statement

When working with Vulkan, there's lots of small problems to solve in order to render a triangle. One such example of this is deciding which queue families to use. Queue families, for the purposes of this article, can be thought of as places to submit work to the GPU. Each one of these families has a set of capabilities: graphics (rendering images), presentation (showing images on screen), and/or compute (general processing). The Vulkan specification only guarantees that graphics and presentation will be present; the presence of a compute queue family is optional. Vulkan's API to distinguish those queue families presents us with an array of queue family property objects. You can query these objects for their capabilities, mostly using some bit manipulation. Exactly how isn't important, just the end result: the indices into the original array for graphics, present, and compute queue families in a tuple.

With these, we need to create a logical (aka virtual) device. The function to create that device takes a vector of records, each with an index for which queue family we want this logical device to use. Our goal is to pass the smallest vector of records into that function as possible, while still covering (at minimum) graphics and presentation capabilities.

Due to the nature of the problem, we don't have one single solution; some solutions are also suboptimal.

$([1,2],[2,3],[3,4]) \rightarrow$
$\quad [1,2,3]$ -- suboptimal
$\quad [1,3,4]$ -- suboptimal
$\quad [2,3]$ -- optimal
$([1],[2,3],[\,]) \rightarrow$
$\quad [1,2]$ -- optimal
$\quad [1,3]$ -- optimal

Because of this, we're going to pick an arbitrary answer from the best options: minimizing the sum of each vector. One way to think of the solutions in general is a set of lists; we'll call these solution space (set) and solution (lists), respectively. Each solution is a list where each element from the three source lists is paired with one element in each other list without duplicates. Over many possible pairs, we have solution space.

2 Solution

Let's start with exploring our module and imports. We export three functions, which we'll explore next in turn. We use a state monad later for in-place vector sorting, along with the vector and insertion sort libraries. We picked insertion sort because our inputs are usually very small; a length of only one or two is not unusual. We import word to match the types from Vulkan. Finally, we hide some of the prelude to avoid any shadowing to vector functions.

```
module QueueFamilySelect
  (selectQueueFamilies
  , pairgp
  , pairc
  ) where
import Control.Monad.ST
import Data.Vector
import Data.Vector.Algorithms.Insertion
import Data.Word
import Prelude hiding ((++), minimum, concatMap, null, filter, elem)
```

Given our input tuple, we produce the optimal solution with two function calls. We'll explore those two next. The way this function is used, we pair *graphics* and *present*. We use the result of that to pair with *compute*; at that point, we've found the optimal solution for all three items.

```
selectQueueFamilies :: (Vector Word32, Vector Word32, Vector Word32)
   → Vector Word32
selectQueueFamilies (graphics, present, compute) =
  pairc (pairgp graphics present) compute
```

We'll look at *pairgp* first. We take a two vectors, *a* and *b*. This pair is constrained in that it is guaranteed never to be empty, so we can safely define this function partially. These vectors are passed into *concatMap*, along with a lambda. *concatMap* will map the given function over the vector, which will generate another vector, and then concatenate the results. The outer lambda maps another lambda over *b*; the inner lambda takes each element of *a* and appends it to a vector with each element of *b*. We end up with every possible pairing of each element in both vectors.

We take the minimum from this list, and pass it into a custom function which will sort it and remove any duplicates. This prevents us from presenting any incorrect answers like *([0], [0]) → [0,0]*. Once we have this, we can move on to pairing this result and the (possibly empty) compute vector.

```
pairgp :: Vector Word32 → Vector Word32 → Vector Word32
pairgp a b = vUniqSort $ minimum $
  concatMap (λi → fmap (λj → fromList [i, j]) b) a
```

Here, only *b*, as the vector for compute queue family indices, could be null. We check for that, and shortcut to just *a* if that's the case. Otherwise, we check if any elements from *a* exist in *b*. If not, that means we need to gather something from *b*, so we pick the minimum. Finally, if we don't need to, we return *a* by itself, since it sufficiently covers our input.

```
pairc :: Vector Word32 → Vector Word32 → Vector Word32
pairc a b = if null b
  then a
  else if null $ filter (λi → elem i b) a
  then cons (minimum b) a
  else a
```

Let's finally take a look at our combination nub (remove duplicates) and sort. We use the *vector-algorithms* package to sort our list. Since this package does in-place sorting, we use the strict state thread monad to 'thaw' (make mutable), sort, and 'freeze' (make immutable) our newly mutable vector. Finally, we return the output of *uniq*, which removes consecutive duplicates. Since our vector is sorted, we can guarantee that *uniq* in this case is equivalent to a nub.

```
vUniqSort :: Vector Word32 → Vector Word32
vUniqSort a = runST $ do
  t ← thaw a
  sort t
  f ← freeze t
  return $ uniq f
```

3 Testing

To test this, we need to define our inputs and outputs. Our inputs are:

- Three word32 vectors
- The first two of the three lists are guaranteed to have one item in them
- The last one is not guaranteed to have any

Our outputs are:

- One word32 vector
- The first two input lists have at least one element represented
- At most one element of the third input list is represented
- No duplicates
- At most three items

We obviously have our library; otherwise we couldn't test it. Also here are one function from the list type, along with vector and word. We also hide some functions from prelude in order to simplify vector calls. Finally, we import QuickCheck in order to automate our testing.

```
import Data.List (nub)
import Data.Vector
import Data.Word
import Prelude hiding (elem, any, null, length, minimum)
import QueueFamilySelect
import Test.QuickCheck
import Test.QuickCheck.Instances.Vector ()
```

Our first test will be on the innermost function in our library, *pairgp*. We first constrain the input so we throw away any null inputs; we're guaranteed that graphics and presentation queue families will have an entry, so we reflect that here. Next, we use our *chk* function to validate that each input list is represented. We'll explore that next.

```
pairgpElements :: (Vector Word32, Vector Word32) → Property
pairgpElements (a, b) =
  if (null a ∨ null b)
  then property Discard
  else property (Prelude.all id [chka, chkb])
  where
    d :: Vector Word32
    d = pairgp a b
    chka :: Bool
    chka = chk d a
    chkb :: Bool
    chkb = chk d b
```

chk is a little shared utility function that checks to see if any element from x exists in y. Above, and further on, we use it to validate that the output of our functions still has elements from our input.

```
chk :: Vector Word32 → Vector Word32 → Bool
chk x y = any id $ fmap (λi → elem i y) x
```

We'll do the same again with *pairc*; the second input can be null here, so we change our constraints to match. We'll chain our if statements a bit further here as well, in order to check all possible null inputs. A null b should guarantee a as the output, so we check that; otherwise, this function should operate exactly the same as *pairgp*.

```
paircElements :: (Vector Word32, Vector Word32) → Property
paircElements (a, b) =
  if null a
  then property Discard
  else if null b
  then pairc a b === a
  else property (Prelude.all id [chka, chkb])
  where
    d :: Vector Word32
    d = pairc a b
    chka :: Bool
    chka = chk d a
    chkb :: Bool
    chkb = chk d b
```

This same check is done here, but we permit the third input to be null, as it could be in real usage.

```
selectQueueFamiliesElements ::
  (Vector Word32, Vector Word32, Vector Word32)
  → Property
selectQueueFamiliesElements (a, b, c) =
  if (null a ∨ null b)
  then property Discard
  else property $ (Prelude.all id [chka, chkb])
    ∧ if null c
      then True
      else chkc
  where
    d :: Vector Word32
    d = selectQueueFamilies (a, b, c)
    chka :: Bool
    chka = chk d a
    chkb :: Bool
    chkb = chk d b
    chkc :: Bool
    chkc = chk d c
```

We also want to know if the length of the output is less than three. Because our end goal is to select one from each, and deduplicate, we should always be less than three at the end.

```
selectQueueFamiliesLength ::
  (Vector Word32, Vector Word32, Vector Word32)
  → Property
selectQueueFamiliesLength (a, b, c) =
  if (null a ∨ null b)
  then property Discard
  else property $ length (selectQueueFamilies (a, b, c)) ⩽ 3
```

Similarly, there shouldn't be any duplicates in the output, and it should be sorted. A nub of our output should be equivalent to the output.

```
selectQueueFamiliesDupes ::
  (Vector Word32, Vector Word32, Vector Word32)
   → Property
selectQueueFamiliesDupes (a, b, c) =
  if (null a ∨ null b)
  then property Discard
  else (fromList ∘ nub ∘ toList $ selectQueueFamilies (a, b, c))
     === (selectQueueFamilies (a, b, c))
```

Finally, let's run the complete set of tests. We've tested both inner functions, and the three properties of our output that we know are invariant.

```
main :: IO ()
main = do
  quickCheck pairgpElements
  quickCheck paircElements
  quickCheck selectQueueFamiliesElements
  quickCheck selectQueueFamiliesLength
  quickCheck selectQueueFamiliesDupes
  return ()
```

Now that we've done all that, I should note that most real-world inputs to this whole function will be *([0,1],[0],[0])*. If we were going for practicality, and not overengineered correctness, we could've done something much simpler. But this was much more fun. ☺

4 How was this document generated?

We use a custom setup in our package.yaml file, which stack uses to execute the below Haskell (in Setup.hs). It acts equivalently to a shell script; we could use the pandoc package directly, but lhs2TeX provides no such in-language support; so we standardized on shell.

```
import Distribution.Simple
import System.Process

main = do
  createProcess (shell
    "lhs2TeX src/Lib.lhs > Lib.tex && texi2pdf -c -q Lib.tex")
  createProcess (shell
    "pandoc --from=LaTeX+lhs --to=plain src/Lib.lhs -o Lib.txt")
  createProcess (shell
    "pandoc --from=LaTeX+lhs --to=plain test/Spec.lhs -o Spec.txt")
  defaultMain
```

I will admit I also used some manual editing to the HTML files. This text in the top level of our package.yaml file is what guarantees Setup.hs is run.

```
build-type: Custom
custom-setup:
  dependencies: base >= 4.7 && < 5, Cabal, process
```