Adding arrays to Accelerate's expression language

"Help, I'm stuck in the AST"

Hugo Peters

Introduction

- Accelerate is a deeply embedded language
- Facilitates composition of parallelizable operations
- Both CPU and GPU backends

Introduction

```haskell
sum :: Vector Float -> Vector Float -> Vector Float
sum = zipWith (+)

-- Then becomes

import Accelerate as A

sum :: Acc (Vector Float) ->
       Acc (Vector Float) ->
       Acc (Vector Float)
sum = A.zipWith (A.+)
```

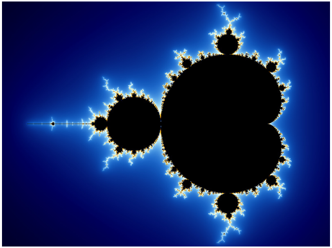
Expression Language

- Array computations in `Acc`
- Element expressions in `Exp`

```haskell
map :: (Exp a -> Exp b) -> Acc (Vector a) -> Acc (Vector b)
```

- Typeclasses like `Num` are already implemented for `Exp` expressions

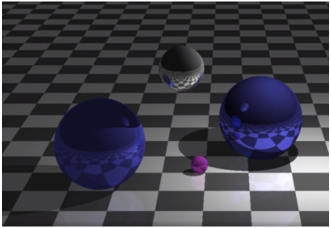
We can also use Accelerate to generate images in a concurrent manner:

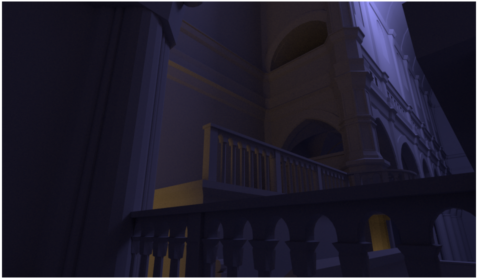


Of course this is easiest when the computations for each pixel are mutually independent (embarrassingly parallel)

Raytracing

Real eyecandy



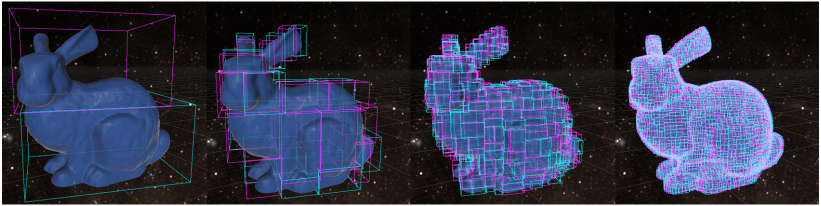


Naive raytracing: $O(n * m)$

```
for pixel in image:
    ray = mkRay(pixel)
    color = BLACK
    min_t = 10000000
    for obj in scene:
        t = isect(ray, obj):
        if t < min_t:
            color = obj.color
            min_t = t
    pixel.set(color)
```

Horribly infeasible for scenes with millions of triangles

- Solution: Bounding Volume Hierarchy
- Traversal cost $O(log\ m)$
- Total cost $O(n\ log\ m)$



Efficient Traversal Requires a stack [Hap]

```
for pixel in image:
    ray = mkRay(pixel)
    color = BLACK
    stack = [scene.root]
    while not stack.empty():
        node = stack.pop()
        if not slabTest(ray, node):
            continue
        if node.isLeaf():
            # Intersect with triangles
        else
            stack.push(node.lhs)
            stack.push(node.rhs)
```

Now in Accelerate

```
render :: Acc Scene -> Acc (Matrix Colour)
render s = map (sceneIntersect s) rays

sceneIntersect :: Acc Scene -> Exp Ray -> Exp Colour
sceneIntersect bvh ray = let s = push (bvh ! I1 0) emptyStack
                                 ...
                             in ...
```

But the concept of a stack is not native to Accelerate's expression language

Why not use Vector?

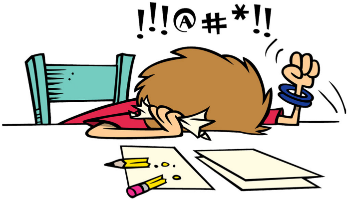
- We can index and update global arrays.
- What about a 2D array of size *rays* $\times$ *stackSize*?

Why not use Vector!

- Focusing on GPU architecture...
- Traversal context will be in global memory
- Access is considered random
- Writes trigger cache flushes
- Memory stalling will dominate execution time
- Conclusion: stack must be in `Exp`, allowing for register use

Changing Accelerate is scary

- Standing on the shoulders of PhD students...
- You only see mist



Attempt 1: Stack in vanilla Accelerate

- Idea: leverage native tuple support
- list `[0,1,2,3]` becomes `(3, (2, (1, (0, ()))))`
- stack is then just `(headIdx :: Int, list)`

Attempt 1: Stack in vanilla Accelerate

```haskell
type family Pile (n :: Nat) a = r | r -> n where
Pile 'NZ a = ()
Pile ('NS n) a = (Pile n a, a)

-- Stack bottom is at root of the tuple list;
-- stack tip is n down towards the ()
data Stack (n :: Nat) a = Stack_ Int (Pile n a)

-- Syntax sugar for creating an Exp (Stack n a) constructor
pattern Stack :: (KnownNat n, A.Elt a, A.Elt (Pile n a))
          => A.Exp Int -> A.Exp (Pile n a) -> A.Exp (Stack n a)
pattern Stack len pr = A.Pattern (len, pr)
```

We need *{-# LANGUAGE ConstraintKinds #-}* to derive the constraints needed to convince Accelerate.

```
emptyPile :: forall n a. A.Elt a => Proxy a -> NatTerm n -> A.Exp (Pile n a)
emptyPile _ ZZ = A.constant ()
emptyPile proxy (SS nn) =
    let p = emptyPile proxy nn
    in case pileHasElt (Proxy :: Proxy a) nn of
         Has -> A.T2 p A.undef

emptyStack' :: forall n a. A.Elt a => NatTerm n -> A.Exp (Stack n a)
emptyStack' nn
  | Has <- pileHasElt (Proxy :: Proxy a) nn
  , Has <- natTermIsKnown nn
  = Stack 0 (emptyPile (Proxy :: Proxy a) nn)

emptyStack :: forall n a. (A.Elt a, KnownNat n) => A.Exp (Stack n a)
emptyStack = emptyStack' inferNat
```

```haskell
-- This becomes a right-leaning tree of conditionals
pilePush :: forall n a. A.Elt a => NatTerm n -> Int -> A.Exp Int -> A.Exp (Pile
pilePush ZZ _ len _ _ = A.undef
pilePush (SS nn) depth len prep x
  | Has <- pileHasElt (Proxy @a) nn
  , A.T2 p y <- prep  -- needs evidence from 'pileHasElt'
  = A.cond (len A.== A.constant depth)
           (A.T2 len (A.T2 p x))
           (case pilePush nn (depth + 1) len p x of
              A.T2 len' p' -> A.T2 len' (A.T2 p' y))

stackPush' :: forall n a. A.Elt a => NatTerm n -> A.Exp (Stack n a) -> A.Exp a
stackPush' nn stackexpr x
  | Has <- natTermIsKnown nn
  , Has <- pileHasElt (Proxy @a) nn
  , Stack len p <- stackexpr
  , A.T2 len' p' <- pilePush nn 0 len p x
  = Stack (len'+1) p'
```

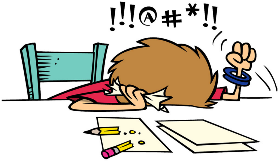
Approximate C equilavent

```
template<typename T>
void pilePush(Pile<T>* pile, int loc, T el) {
    switch loc {
        case 0: *pile->head() = el; break;
        case 1: *pile->head()->next() = el; break;
        case 2: *pile->head()->next()->next() = el; break;
        // ... etc.
    }
}
```

Control flow branches are a GPU's nightmare :(

In Summary

- Source is very complicated
- Lots of branches
- Elements not stored in contiguous memory
- Compilation time of kernels is exponential
- Runtime performance is a disaster



Attempt 2: Extend Accelerate

- Accelerate compiles to LLVM for all backends
- With a bit of luck no GPU specific changes are required
- LLVM has support for 'SIMD' vectors, including indexing and updating

Original Interface

```
ExtractElement  :: Int32
                -> Operand (Vec n a)
                -> Instruction a

InsertElement   :: Int32
                -> Operand (Vec n a)
                -> Operand a
                -> Instruction (Vec n a)
```

No dynamic indexing...

Updated Interface

```
ExtractElement  :: IntegralType i
                -> Operand (Vec n a)
                -> Operand i
                -> Instruction a

InsertElement   :: IntegralType i
                -> Operand (Vec n a)
                -> Operand i
                -> Operand a
                -> Instruction (Vec n a)
```

Intermezzo: Witnesses

- We want `i` to be in a specific set of supported types
- GADTs offer a closed alternative to typeclasses

```
data IntegralType a where
  TypeInt     :: IntegralType Int
  TypeInt8    :: IntegralType Int8
  TypeWord8   :: IntegralType Word8
  TypeInt16   :: IntegralType Int16
  ...
```

Updated Interface

Previous uses can easily be adapted by lifting constant values

```
constOp :: (IsScalar a) => a -> Operand a
constOp x = ConstantOperand
              (ScalarConstant scalarType x)
```

Implementation: Extend Exp AST

```
VecIndex      :: KnownNat n
              => VecR n s tup
              -> IntegralType i
              -> OpenExp env aenv (Vec n s)
              -> OpenExp env aenv i
              -> OpenExp env aenv s
```

Implementation

- For simplicity, `VecIndex` only
- `VecWrite` is very similar

Implementation: Code Generation for LLVM IR

```
cvtE :: forall t. OpenExp env aenv t -> IROpenExp arch env aenv t
cvtE exp =
  case exp of
     ...
     VecIndex vecr ti ve ie -> do i <- cvtE ie
                                  v <- cvtE ve
                                  vecIndexGen (vecRvector vecr) ti v i


vecIndexGen :: VectorType (Vec n a)
            -> IntegralType i
            -> Operands (Vec n a)
            -> Operands i
            -> CodeGen arch (Operands a)
vecIndexGen tv ti (op tv -> v) (op ti -> i)
            = instr $ ExtractElement ti v i
```

Implementation: Expose to front end

- `derive` is a shortcut here for simplicity

```
vecIndex :: (VecElt a, KnownNat n)
         => Exp (Vec n a)
         -> Exp Int
         -> Exp a
vecIndex (Exp v) (Exp i) = mkExp $
    VecIndex derive integralType v i
```

Result

```haskell
data Stack n a = Stack_ Int (A.Vec n a) deriving (Generic)
deriving instance (KnownNat n, A.VecElt a) => A.Elt (Stack n a)

type Stacking n a = (KnownNat n, A.VecElt a)

pattern Stack :: (KnownNat n, A.VecElt a)
              => A.Exp Int
              -> A.Exp (A.Vec n a)
              -> A.Exp (Stack n a)
pattern Stack headIdx dat = A.Pattern (headIdx, dat)
```

Result

```haskell
stackPush :: (Stacking n a)
          => A.Exp (Stack n a)
          -> A.Exp a
          -> A.Exp (Stack n a)
stackPush (Stack headIdx dat) v =
     Stack (headIdx+1) (A.vecWrite dat headIdx v)

stackPop :: (Stacking n a)
         => A.Exp (Stack n a)
         -> A.Exp (Stack n a, a)
stackPop (Stack headIdx dat) =
     A.T2 (Stack (headIdx-1) dat) (A.vecIndex dat (headIdx-1))
```

Result

- Let's take it for a ride!
- 278502 triangles
- 32767 bvh nodes

Result

Result

- Runs in realtime
- 90.9% occupancy (of theoretical maximum)
- Memory throughput is the bottleneck as expected

References (I)

[Hap] Michal Hapala, *Efficient stack-less bvh traversal for ray tracing - sci.utah.edu*.