GPU Programming

Obsidian: Internals

Bo Joel Svensson

April 18, 2013

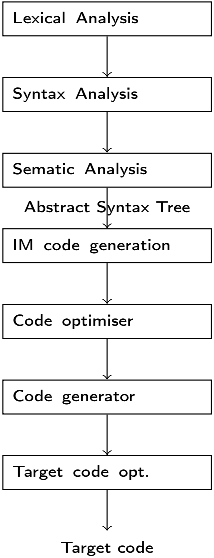
Today!

- More about arrays:
  - What is a pull array, really ?
  - What is a push array, really ?
- Programs:
  - `TProgram`
  - `BProgram`
  - `GProgram`
- Implementation:
  - Library functions.
  - Code generation.

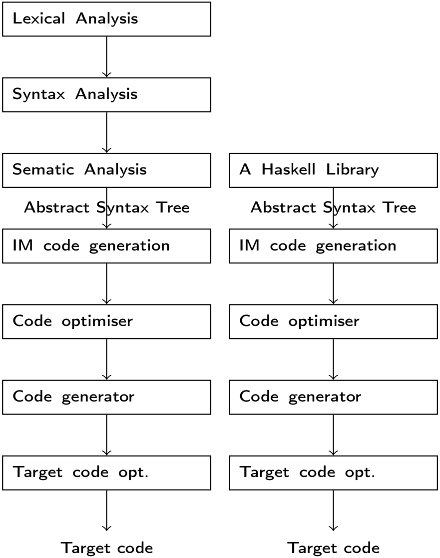
But first: A bug in previous lecture

```
splitUp :: Word32 -> DPull a -> DPull (SPull a)
splitUp n arr =
  mkPullArray (m `div` fromIntegral n) $ \i ->
    mkPullArray n $ \j -> arr ! (i * fromIntegral n + j)
  where
    m = len arr
```

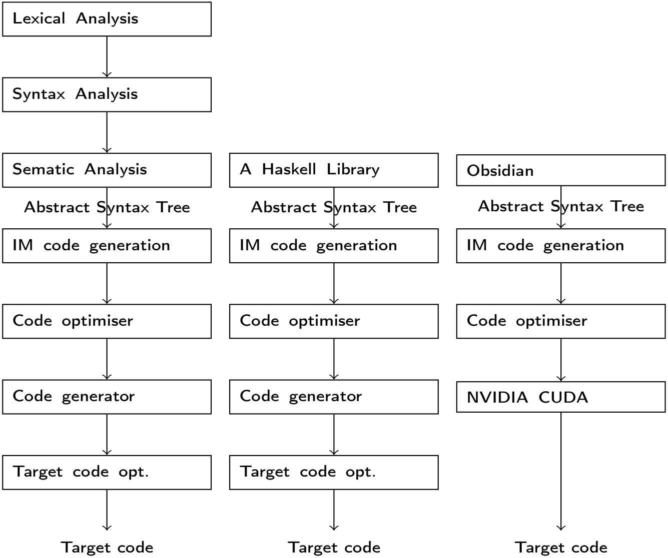
Introduction to compiled embedded languages



Introduction to compiled embedded languages



Introduction to compiled embedded languages



Scalar expressions I

```
data Exp a where
  Literal :: Scalar a
             => a
             -> Exp a

  WarpSize :: Exp Word32

  BlockDim :: DimSpec -> Exp Word32

  BlockIdx :: DimSpec
              -> Exp Word32
  ThreadIdx :: DimSpec
               -> Exp Word32

  Index    :: Scalar a =>
             (Name,[Exp Word32])
             -> Exp a
```

Scalar expressions II

```
If       :: Scalar a
            => Exp Bool
            -> Exp a
            -> Exp a
            -> Exp a

BinOp    :: (Scalar a,
             Scalar b,
             Scalar c)
            => Op ((a,b) -> c)
            -> Exp a
            -> Exp b
            -> Exp c

UnOp     :: (Scalar a,
             Scalar b)
            => Op (a -> b)
            -> Exp a
            -> Exp b
```

Scalar expressions: Example Operations I

```
data Op a where
  Add :: Num a => Op ((a,a) -> a)
  Sub :: Num a => Op ((a,a) -> a)
  Mul :: Num a => Op ((a,a) -> a)
  Div :: Num a => Op ((a,a) -> a)

  Mod :: Integral a => Op ((a,a) -> a)

  -- Trig
  Sin :: Floating a => Op (a -> a)
  Cos :: Floating a => Op (a -> a)

  -- Comparisons
  Eq  :: Ord a => Op ((a,a) -> Bool)
  NotEq :: Ord a => Op ((a,a) -> Bool)
  Lt  :: Ord a => Op ((a,a) -> Bool)
  LEq :: Ord a => Op ((a,a) -> Bool)
  Gt  :: Ord a => Op ((a,a) -> Bool)
```

Scalar expressions: Smart constructors

```
instance Num (Exp Int) where
  (+) a (Literal 0) = a
  (+) (Literal 0) a = a
  (+) (Literal a) (Literal b) = Literal (a+b)
  (+) a b = BinOp Add a b
  ...
```

Scalar expressions: Smart constructors

```
instance Num (Exp Int) where
  (+) a (Literal 0) = a
  (+) (Literal 0) a = a
  (+) (Literal a) (Literal b) = Literal (a+b)
  (+) a b = BinOp Add a b
  ...
```

Applies some optimisations!

Pull arrays

```
data Pull s a = Pull {pullLen :: s,
                      pullFun :: EWord32 -> a}

type SPull = Pull Word32
type DPull = Pull EWord32
```

Pull arrays

```
data Pull s a = Pull {pullLen :: s,
                      pullFun :: EWord32 -> a}

type SPull = Pull Word32
type DPull = Pull EWord32
```

- A dynamic or static length.
- A function from index to element.

Pull arrays: Implement fmap

```
fmap :: (a -> b) -> Pull l a -> Pull l b
fmap f (Pull n ixf) = Pull n (f . ixf)
```

Pull arrays: Map fusion for free

```
mapFusion :: SPull EFloat -> SPull EFloat
mapFusion = fmap (+1) . fmap (*2)
```

Pull arrays: Map fusion for free

```
mapFusion :: SPull EFloat -> SPull EFloat
mapFusion = fmap (+1) . fmap (*2)

mapFusionG :: DPull EFloat -> DPush Grid EFloat
mapFusionG arr = mapG (return . mapFusion) (splitUp 256 arr)
```

Pull arrays: Map fusion for free

```
mapFusion :: SPull EFloat -> SPull EFloat
mapFusion = fmap (+1) . fmap (*2)

mapFusionG :: DPull EFloat -> DPush Grid EFloat
mapFusionG arr = mapG (return . mapFusion) (splitUp 256 arr)

> getMapFusionG

__global__ void mapFusion(float* input0
                         ,uint32_t n0
                         ,float* output0){

    uint32_t t0 = ((blockIdx.x*256)+threadIdx.x);
    output0[t0] = ((input0[t0]*2.0)+1.0);


}
```

Pull arrays: Map fusion for free

```
mapFusion2 :: SPull EFloat -> SPull EFloat
mapFusion2 = fmap ((+1) .(*2))
```

Pull arrays: Map fusion for free

```
mapFusion2 :: SPull EFloat -> SPull EFloat
mapFusion2 = fmap ((+1) .(*2))

> getMapFusion2G

__global__ void mapFusion2(float* input0
                          ,uint32_t n0
                          ,float* output0){

    uint32_t t0 = ((blockIdx.x*256)+threadIdx.x);
    output0[t0] = ((input0[t0]*2.0)+1.0);


}
```

Pull arrays: Choose not to fuse!

```
mapNotFused :: SPull EFloat -> BProgram (SPull EFloat)
mapNotFused arr =
  do
    arr1 <- force $ fmap (*2) arr
    return $ fmap (+1) arr
```

Pull arrays: Choose not to fuse!

```
mapNotFused :: SPull EFloat -> BProgram (SPull EFloat)
mapNotFused arr =
  do
    arr1 <- force $ fmap (*2) arr
    return $ fmap (+1) arr
```

```
> getMapNotFusedG

__global__ void mapNotFused(float* input0
                           ,uint32_t n0
                           ,float* output0){

    uint32_t t1 = ((blockIdx.x*256)+threadIdx.x);
    extern __shared__ __attribute__ ((aligned(16))) uint8_t sbas
    ((float*)sbase)[threadIdx.x] = (input0[t1]*2.0);
    __syncthreads();
    output0[t1] = (input0[t1]+1.0);


}
```

The Program monad

```
data Program t a where
  Assign :: Scalar a
            => Name
            -> [EWord32]
            -> (Exp a)
            -> Program Thread ()

  ForAll :: EWord32
            -> (EWord32 -> Program Thread a)
            -> Program Block a

  ForAllBlocks :: EWord32 -> (EWord32 -> Program Block a)
                  -> Program Grid a

  ...
```

The Program monad

```
data Program t a where
  Assign :: Scalar a
            => Name
            -> [EWord32]
            -> (Exp a)
            -> Program Thread ()

  ForAll :: EWord32
            -> (EWord32 -> Program Thread a)
            -> Program Block a

  ForAllBlocks :: EWord32 -> (EWord32 -> Program Block a)
                  -> Program Grid a

  ...

instance Monad (Program t)
```

Push arrays

```
data Push t s a =
  Push s ((a -> EWord32 -> Program Thread ()) -> Program t ())
```

Push arrays: Implement fmap

```
fmap   f (Push s p) = Push s $ \wf -> p (\e ix -> wf (f e) ix)
```

Why Push and Pull arrays

- concatenation of pull arrays is inefficient.
  Introduces conditionals.
- concatenation of Push arrays is efficient.
  No conditionals.
- splitting arrays up and using parts of them is easy using pull arrays.
- Push and Pull arrays seem to have strengths and weaknesses that complement each other.

Convert a Pull array to a Push array

Converting from a Pull array to a Push array is cheap!

```
convertToPush :: SPull a -> SPush Block a
convertToPush (Pull n ixf) =
    Push n $
      \wf -> ForAll (fromIntegral n) $ \i -> wf (ixf i) i
```

Convert a Push array to a Pull array

Is much more costly!

- Compute all values of the Push array.
- Store values into a memory array.
- Return a pull array that represents reading from that memory.

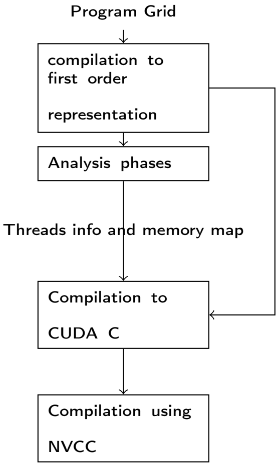
Convert a Push array to a Pull array

Is much more costly!

- Compute all values of the Push array.
- Store values into a memory array.
- Return a pull array that represents reading from that memory.

This is what the `force` function does.

Outline of code generation



Obsidian on GitHub

https://github.com/svenssonjoel/Obsidian

Obsidian on GitHub

https://github.com/svenssonjoel/Obsidian

- Many branches.
- Many failed experiments.
- Some successful ones.
- A “fork”.

Master thesis about Functional GPU Programming

Talk to me and Mary if you are interested in doing a master's thesis related to GPU programming using a declarative/functional approach.

- Help me improve some aspect of Obsidian.
- Add a feature to Obsidian.
- A language for kernel coordination and full GPU applications entirely from within Haskell.
- A new more targeted EDLS (possibly using Obsidian to generate code).
- A virtual machine for heterogeneous data-parallel computations. (Compiler course "star")

Next lecture

Friday 19th April (Tomorrow)
Dr. Jost Berthold
Will talk about Skeletons!
Please, bring your laptop to the lecture.

End