Obsidian

GPU Programming in Haskell

Joel Svensson
Joint work with Koen Claessen and Mary Sheeran
Chalmers University

GPUs

- Offer much performance per $
- Designed for the highly data-parallel computations of graphics

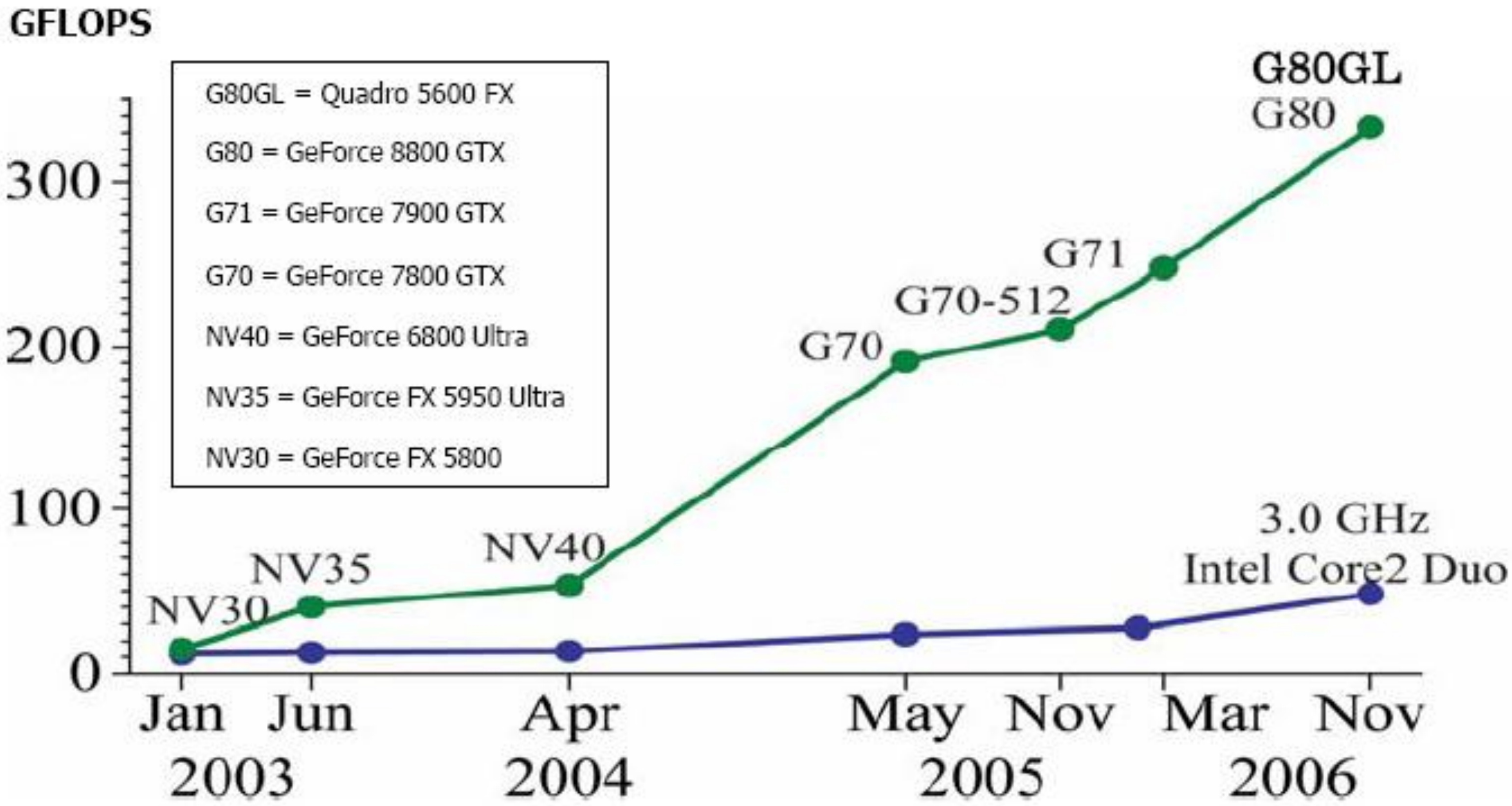
GPGPU: General-Purpose Computations on the GPU

- Exploit the GPU for general-purpose computations
  - Sorting
  - Bioinformatics
  - Physics Modelling

www.gpgpu.org

GPU vs CPU GFLOPS Chart

Source: NVIDIA CUDA Programming Manual



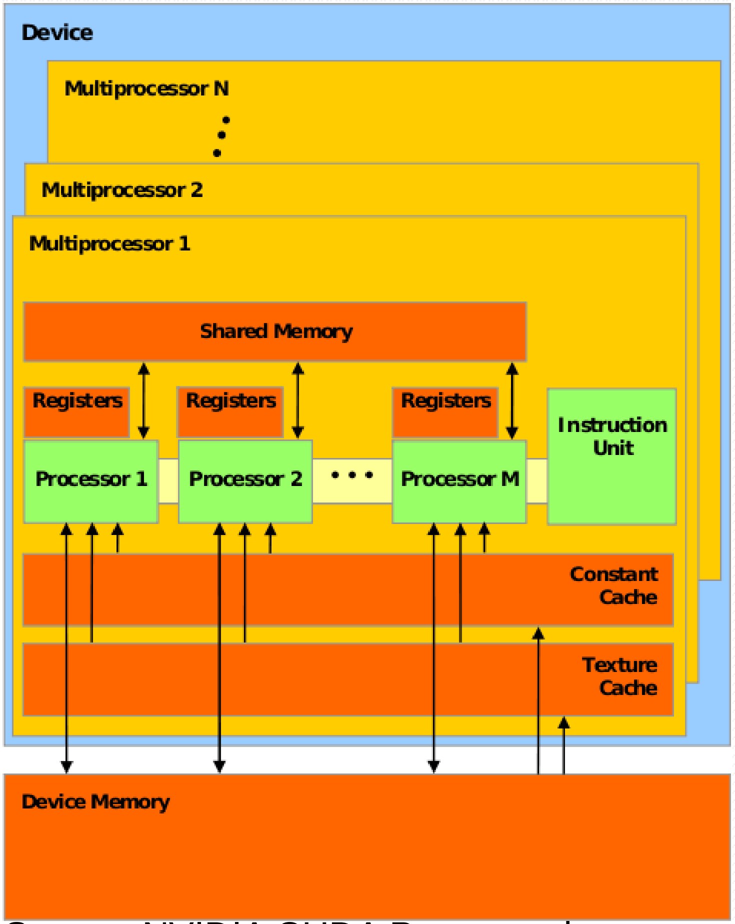
An example of GPU hardware

- NVIDIA GeForce 8800 GTX
  - 128 Processing elements
  - Divided into 16 Multiprocessors
  - Exists with up to 768MB of Device memory
  - 384-bit bus
  - 86.4GB/sec Bandwidth

www.nvidia.com/page/geforce_8800.html

A Set of SIMD Multiprocessors

- In each Multiprocessor
  - Shared Memory (currently 16Kb)
  - 32 bit registers (8192)
- Memory
  - Uncached Device Memory
  - Read-only constant memory
  - Read-only texture memory



Source: NVIDIA CUDA Programming manual

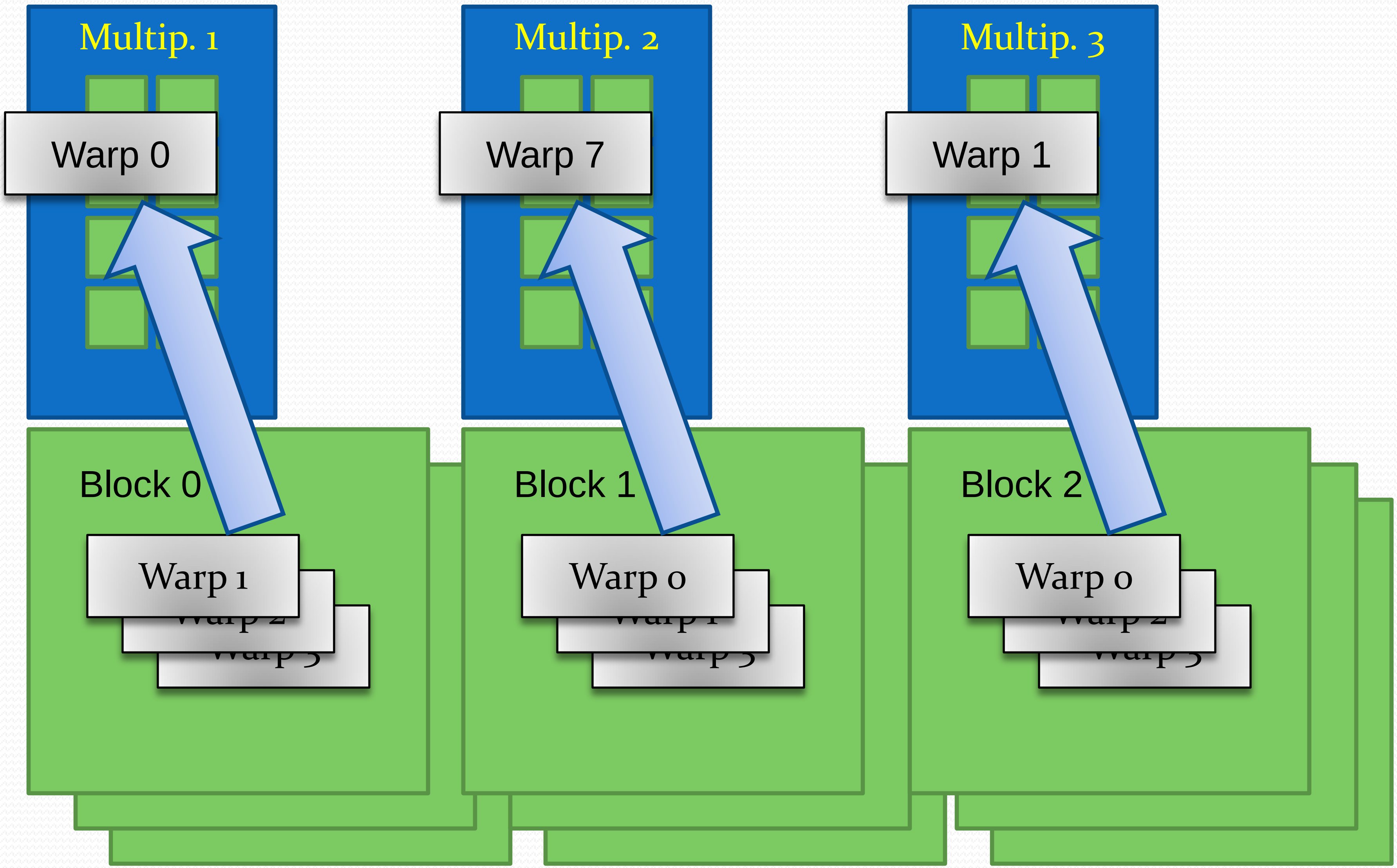
NVIDIA CUDA

- CUDA: Compute Unified Device Architecture
  - Simplifies GPGPU programming by:
    - Supplying a C compiler and libraries
    - Giving a general purpose interface to the GPU
  - Available for high end NVIDIA GPUs

www.nvidia.com/cuda

CUDA Programming Model

- Execute a high number of threads in parallel
  - Block of threads
    - Up to 512 threads
    - Executed by a multiprocessor
    - Blocks are organized into grids
      - Maximum grid dimensions: 65536*65536
  - Thread Warp
    - 32 threads
    - Scheduled unit
    - SIMD execution


Multip. 1
Multip. 2
Multip. 3
Warp 0
Warp 7
Warp 1
Block 0
Block 1
Block 2
Warp 1
Warp 0
Warp 0

CUDA Programming Model

- A program written to execute on the GPU is called a *Kernel*.
  - A kernel is executed by a block of threads
  - Can be replicated across a number of blocks.
- The Block and Grid dimensions are specified when the kernel is launched.

CUDA Programming Model

- A number of constants are available to the programmer.
  - threadIdx
    - A vector specifying thread ID in <x,y,z>
  - blockIdx
    - A vector specifying block ID in <x,y>
  - blockDim
    - The dimensions of the block of threads.
  - gridDim
    - The dimensions of the grid of blocks.

CUDA Syncronisation

- CUDA supplies a synchronisation primitive, __syncthreads()
  - Barrier synchronisation
  - Across all the threads of a block
  - Coordinate communication

Obsidian

- Embedded in Haskell
- High level programming interface
  - Using features such as higher order functions
- Targeting NVIDIA GPUs
  - Generating CUDA C code
- Exploring similarities between structural hardware design and data-parallel programming.
  - Borrowing ideas from Lava.

Obsidian and Lava: Parallel programming and Hardware design

- Lava
  - Language for structural hardware design.
  - Uses combinators that capture connection patterns.
- Obsidian
  - Explores if a similar programming style is applicable to data-parallel programming.

Obsidian and Lava

Obsidian

- Generates C code.
- Can output parameterized code.
- Iteration inside kernels

Lava

- Generates netlists.
- Recursion

Obsidian Programming

A small example, reverse and increment:

```
rev_incr :: Arr (Exp Int) -> W (Arr (Exp Int))
rev_incr = rev ->- fun (+1)
```

```
*Obsidian> execute rev_incr [1..10]
[11,10,9,8,7,6,5,4,3,2]
```

Code is Generated, Compiled and it is Executed on the GPU

Obsidian Programming

CUDA C code generated from `rev_incr`:

```
__global__ static void rev_incr(int *values, int n)
{
  extern __shared__ int shared[];
  int *source = shared;
  int *target = &shared[n];
  const int tid = threadIdx.x;
  int *tmp;
  source[tid] = values[tid];
  __syncthreads();
  target[tid] = (source[((n - 1) - tid)] + 1);
  __syncthreads();
  tmp = source;
  source = target;
  target = tmp;

  __syncthreads();
  values[tid] = source[tid];
}
```

Setup

1

2

About the generated Code

- Generated code is executed by a single block of threads.
- Every Thread is responsible for writing to a particular array index.
  - Limits us to 512 elements. (given 512 threads)

Obsidian Programming

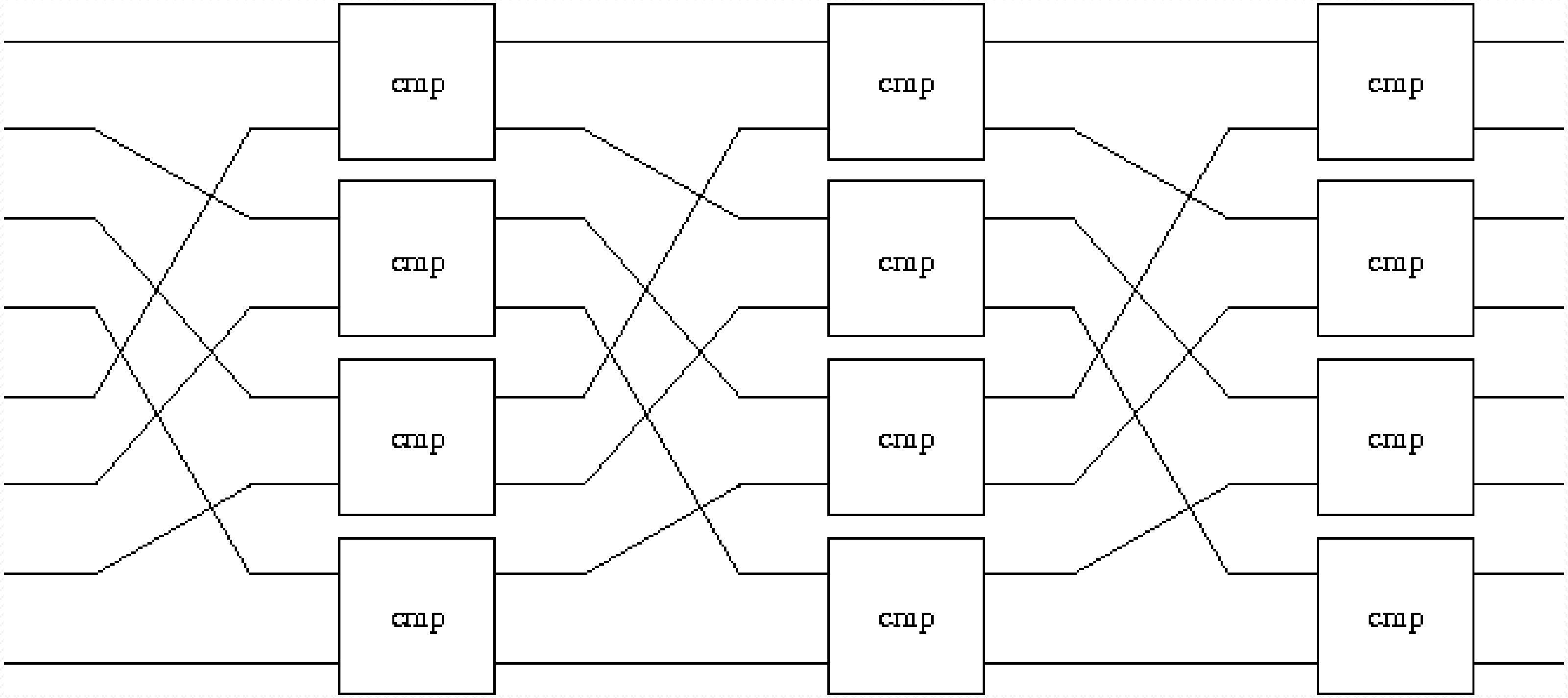
- A larger example and a comparison of Lava and Obsidan programming
  - A sorter called Vsort is implemented in both Lava and Obsidian
  - Vsort
    - Built around:
      - A two-sorter (`sort2`)
      - A shuffle exchange network (`shex`)
      - And a wiring pattern here called (`tau1`)

Lava Vsort

- Shuffle exchange network

```
rep 0 f = id
rep n f = f ->- rep (n-1) f

shex n f = rep n (riffle ->- evens f)
```

Shuffle Exchange Network



Lava Vsort

- Periodic merger using `tau1` and `shex`

```
one f = parl id f

tau1 = unriffle ->- one reverse

mergeIt n = tau1 ->- shex n sort2
```

Haskell list reverse

- Vsort in Lava

```
vsortIt n = rep n (mergeIt n)
```

Obsidian Vsort

```
one f = parl return f

tau1 = unriffle ->- one rev

shex n f = rep n (riffle ->- evens f)

mergeIt n = tau1 ->- shex n sort2

vsortIt n = rep n (mergeIt n)
```

Rep primitive

Vsort

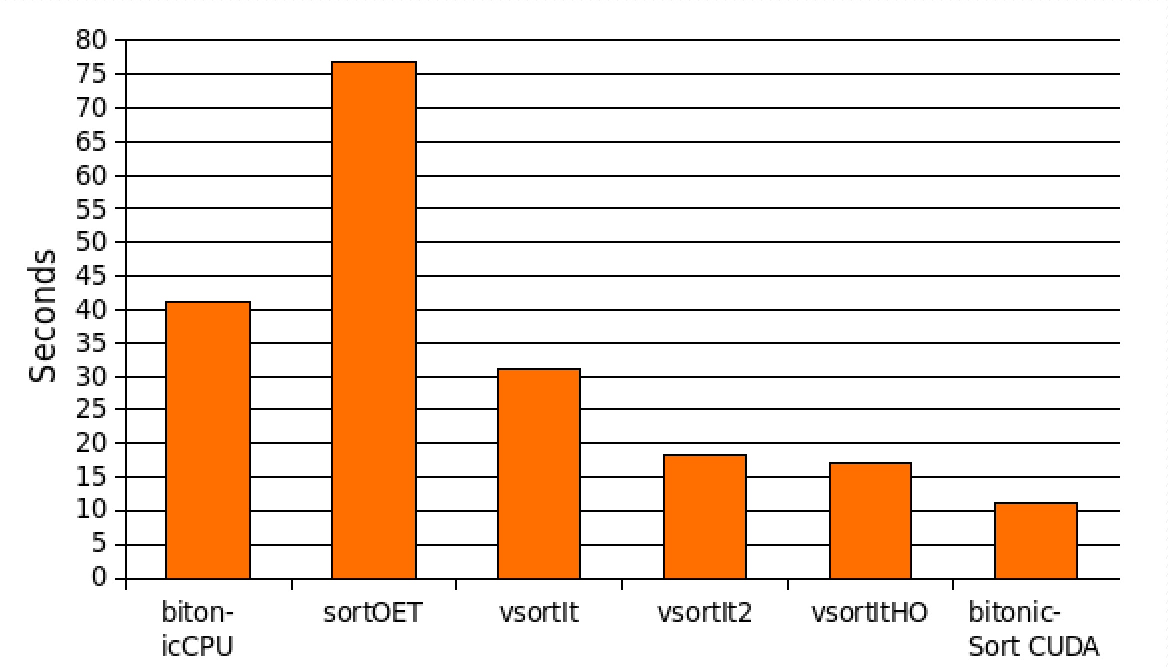
```
Vsort> simulate (vsortIt 3) [3,2,6,5,1,8,7,4]
[1,2,3,4,5,6,7,8]

Vsort> simulate (vsortIt 4) [14,16,3,2,6,5,15,1,8,7,4,13,9,10,12,11]
[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16]


Vsort> emulate (vsortIt 3) [3,2,6,5,1,8,7,4]
[1,2,3,4,5,6,7,8]
```

`emulate` is simialar to `execute` but the code is run on the CPU

Obsidian applications

- We have used Obsidian in implementing
  - Sorting algorithms
    - A comparison of sorters is coming up.
  - A parallel prefix (Scan) algorithm
  - Reduction of an array (fold of associative operator)

Comparison of Sorters



Implementation of Obsidian

- Obsidian describes operations on Arrays
  - Representation of an array in Obsidian
    - `data Arr a = Arr (IxExp -> a,IxExp)`
  - Helper functions
    - `mkArray`
    - `len`
    - `!`

Implementation of Obsidian

- `rev` primitive
  - reverses an array

```
rev :: Arr a -> W (Arr a)
rev arr =
   let n = len arr
   in  return $ mkArray (\ix -> arr ! ((n  - 1) - ix)) n
```

Implementation of Obsidian

- `halve`

```
halve :: Arr a -> W (Arr a, Arr a)
halve arr =
   let n = len arr
       nhalf = divi n 2
       h1 = mkArray (\ix -> arr ! ix) (n - nhalf)
       h2 = mkArray (\ix -> arr ! (ix + (n - nhalf))) nhalf
   in  return (h1,h2)
```

Implementation of Obsidian

- **Concatenate arrays:** `conc`

```
conc :: Choice a => (Arr a, Arr a) -> W (Arr a)
conc (arr1,arr2) =
   let (n,n') = (len arr1, len arr2)
   in  return $ mkArray (\ix -> ifThenElse (ix <* n)
                                 (arr1 ! ix)
                                 (arr2 ! (ix - n))) (n+n')
```

Implementation of Obsidian

- The `W` monad
  - Writer monad
  - Extended with functionality to generate Identifiers
    - Loop indices

Implementation of Obsidian

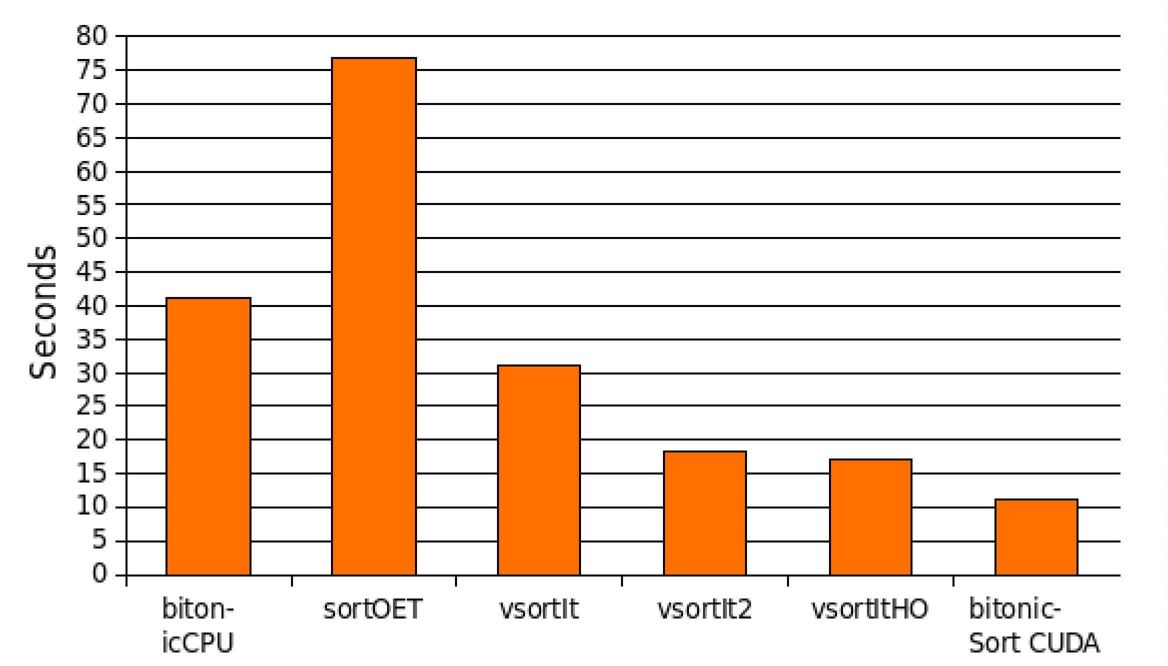
- The `sync` operation
  - `sync :: Arr a -> W (Arr a)`
  - Operationally the identity function
  - Representation of program written into `W` monad
  - Position of syncs may impact performance of generated code but not functionality.

Implementation of Obsidian

- The sync operation
  - An example

```
shex n f = rep n (riffle ->- evens f)

shex n f = rep n (riffle ->- sync ->- evens f)
```

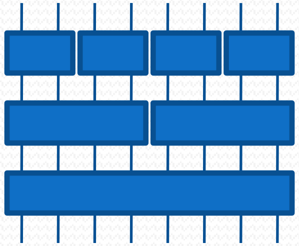
Comparison of Sorters



Latest developments

- At the Kernel level
  - Combinators that capture common recursive patterns
    - `mergePat`

`mergePat` can be used to implement a recursive sorter:



```
merger  = pshex sort2
recSort = mergePat (one rev ->- merger)
```

Latest developments

- At the Kernel level
  - Going beyond 1 element/thread
    - A merger that operates on two elements per thread
  - Important for efficiency
    - High level decision that effects performance
    - Hard in CUDA, easy in Obsidian
      - Has to be decided early in CUDA flow.
  - Needs to be generalised
    - Now allows 1 elem/thread and 2 elem/thread

Latest developments

- At the block level
  - Kernel Coordination Language
    - Enable working on large arrays
    - An FFI allowing coordnation of computations on the GPU from within Haskell.
    - Work in progress
    - Large sorter based on Bitonic sort
      - Merge kernels and sort kernels generated by Obsidian

http://www.cs.um.edu.mt/DCC08

Questions?

References

1. Guy E. Blelloch. NESL: A Nested Data-Parallel language. Technical report CMU-CS-93-129, CMU Dept. Of Cumputer Science April 1993.
2. Manuel M. T. Chakravarty, Roman Leshchinskiy, Simon P. Jones, Gabriele Keller, and Simon Marlow. Data parallel haskell: a status report. In DAMP ’07: Proceedings of the 2007 workshop on Declarative aspects of multicore programming, pages 10–18, New York, NY, USA, 2007. ACM Press.
3. Conal Elliot. Functional images. In The Fun of Programming, Cornerstones of Computing. Palgrave, March 2003
4. Conal Elliot. Programming graphics processors functionally. In Proceedings of the 2004 Haskell Workshop. ACM Press, 2004
5. Calle Lejdfors and Lennart Ohlsson. Implementing an embedded gpu language by combining translation and generation. In SAC’06: Proceedings of the 2006 ACM symposium on Applied computiong, pages 1610-1614. New York, NY, USA, 2006. ACM

Related Work

- NESL [1]
  - Functional language
  - Nested data-parallelism
  - Compiles into VCode
- Data Parallel Haskell [2]
  - Nested data-parallelism in Haskell

Related Work

- Pan [3]
  - Embedded in Haskell
  - Image synthesis
  - Generates C code
- Vertigo [4]
  - Also embedded in Haskell
  - Describes *Shaders*
  - Generates GPU Programs

Related Work

- PyGPU [5]
  - Embedded in Python
  - Uses Pythons introspective abilities
  - Graphics applications
  - Generates code for GPUs

Future Work

- Optimisation of generated code.
  - Currently no optimisations are performed .
- The coordination of Kernels
  - Enable computations on very large arrays by composing kernels.
  - Make use of entire GPU
    - Currently work in progress
- Capture more recursive patterns with combinators.

Reflections

- Currently Obsidian suffers from limitations
  - Some will be helped by the Kernel coordination layer.
    - Stuck in a block
    - 512 elements
  - More generality within a block is also needed
    - Not only arrays of integers
    - More expressive power
      - Combinators capturing recursive patterns

Reflections

- Obsidian supplies a high level programming interface
  - Quick prototyping of Algorithms.
  - Simplify data-parallel programming by its novel programming style.
- Usefulness of Obsidian will improve with:
  - Kernel coordination layer
  - More generality at the block level.

Obsidian Programming

An example using iteration:

```
revs arr = let n = len arr
           in  repE n rev arr
```

```
*Obsidian> execute revs [1..10]
[1,2,3,4,5,6,7,8,9,10]

*Obsidian> execute revs [1..11]
[11,10,9,8,7,6,5,4,3,2,1]
```

Obsidian Programming

CUDA C code generated from `revs`:

```
for (int i0 = 0;(i0 < n);i0 = (i0 + 1)){
  target[tid] = source[((n - 1) - tid)];
  __syncthreads();
  tmp = source;
  source = target;
  target = tmp;
}
```

Parametric Sorter

Lava and Obsidian

- Very similar implementations of Vsort in Lava and Obsidan.
- But the above example does not use the generality of Obsidian.
  - Obsidian can be used to generate parametric code.

Parametric Vsort in Obsidian

- Built around parametric versions of:
  - The Shuffle exchange network (`pshex`)
  - The periodic merger (`pmergeIt`)
  - Using a slightly different version of the repetition combinator called `repE`

Parametric Vsort in Obsidian

```
pshex f arr =
    let n = log2i (len arr)
    in  repE n (riffle ->- evens f) arr

pmergeIt = tau1 ->- pshex sort2

pvsortIt arr =
    let n = log2i (len arr)
    in (repE n pmergeIt) arr
```

VSort

```
Vsort> emulate pvsortIt [3,2,6,5,1,8,7,4]
[1,2,3,4,5,6,7,8]

Vsort> emulate pvsortIt [14,16,3,2,6,5,15,1,8,7,4,13,9,10,12,11]
[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16]
```