Obsidian

A Domain Specific Embedded Language for Parallel Programming of Graphics Processors

Joel Svensson, Mary Sheeran, Koen Claessen
Chalmers University of Technology

Aim of the Obsidian project

- A platform for experimenting with data-parallel algorithms
  - Generate efficient code for GPUs from short and clean high level descriptions
  - Make design decisions easy
    - Where to place data in memory hierarchy
    - Control what is computed where and when

Graphics Processing Units (GPUs)

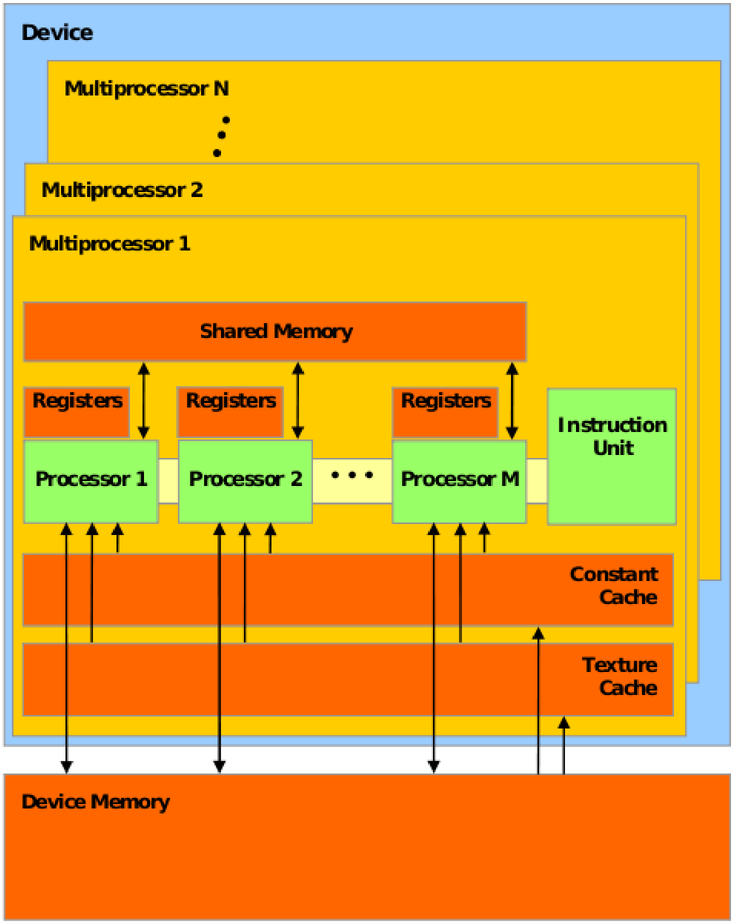
- NVIDIA 8800 GTX (G80)
  - 681 Million transistors
  - 128 Processing cores
    - In groups of 8 (16 ”multiprocessors”)

- Intel Core 2 Quad
  - 582 Million transistors
  - 4 cores

Transistor numbers from Wikipedia.

NVIDIA GPU

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



GPGPU

- General Purpose computations using a GPU
  - Many success stories
    - Sorting
    - Bioinformatics
    - Physics Modelling
  - For more information
    - www.nvidia.com/cuda
    - www.gpgpu.org

NVIDIA CUDA

- Compute Unified Device Architecture
  - NVIDIAs hardware architecture + programming model
  - Provides a Compiler and Libraries
    - An extension of C
  - Programs written for execution on the GPU are called Kernels.

- Related
  - Brook, Brook+
  - AMD ”Close to Metal”

CUDA Programming Model

- Execute a high number of threads in parallel
  - Block of threads
    - Up to 512 threads (1024 on the latest GPUs from NVIDIA)
    - Executed by a multiprocessor
    - Blocks are organized into grids
      - Maximum grid dimensions: 65536*65536
  - Thread Warp
    - 32 threads
    - Scheduled unit
    - SIMD execution (SIMT)

CUDA Example

```
__global__ static void sum(int * values,int n)
{
  extern __shared__ int shared[];

  const int tid = threadIdx.x;

  shared[tid] = values[tid];

  for (int j = 1; j < n; j *= 2) {
  __syncthreads();
    if ((tid + 1) % (2*j) == 0)
      shared[tid] += shared[tid - j];
  }

  values[tid] = shared[tid];

}
```

Obsidian

- Embedded in Haskell
- Tries to stay in the spirit of Lava
  - Combinator library
- Higher level of Abstraction compared to CUDA
  - While still assuming knowledge of architecture characteristics in the programmer.

First example

```
revIncr :: GArr IntE -> W (GArr IntE)
revIncr = rev ->- fun (+1) ->- sync
```

```
*Main> execute EMU revIncr [1..5]
[E (LitInt 6),E (LitInt 5),E (LitInt 4),E (LitInt 3),E
  (LitInt 2)]
```

```
execute :: ExecMode ->
              (GArr (Exp a) -> W (GArr (Exp b)) ->
                     [Exp a] -> IO [Exp b]
```

```
Type GArr a = Arr Global a
```

Matrix Scan

- Recently published GPGPU paper by Yuri Dotsenko et al. Presents a fast implementation of parallel prefix (Scan) on a GPU [1].
  - They call the algorithm matrix Scan.
  - Combines running several sequential reductions/scans on uniformly sized subarrays with parallel computations.

- The following slides will show a Scan in Obsidian taking much influence from the above described algorithm.

Scan

```
seqReduce op id = fun (foldl op id) ->>- sync
```

```
seqScan op id arr column = do
  arr' <- fun (tail . scanl op id) arr
  c    <- prefix (singleton id) column
  (zipp ->>- fun (\(xs,x) -> map (op x) xs) ->>- sync(arr',c)
```

```
chopN :: Monad m => Int -> Arr s a -> m (Arr s [a])
flatten :: (Choice a , Monad m ) => Arr s [a] -> m (Arr s a)
```

Scan

```
matrixScan op id w arr = do
  arr' <- chopN w arr
  sc   <- (seqReduce op id ->- sklansky op n) arr'
  (seqScan op id arr' ->- flatten) sc
    where
       n =  floor (logBase 2 (fromInt (len arr `div` w)))
```

Scan

```
matrixScan :: (Syncable (Arr t t1), Choice t1) =>
               (t1 -> t1 -> t1) -> t1 -> Int ->
                    Arr t t1 - > W (Arr t t1)
```

Scan Kernel

- Turn matrixScan into a kernel

```
scan_add_kernel :: GArr IntE -> W (GArr IntE)
scan_add_kernel = cache ->>- matrixScan (+) 0 32 ->>- wb ->>- sync
```

```
*Main> execute EMU (scan_add_kernel) [1..256]
[E (LitInt 1),E (LitInt 3),E (LitInt 6),...,E (LitInt 32896)]
```

```
cache :: Arr Global a -> W (Arr Shared a)
wb :: Arr Shared a -> W (Arr Global a)
```

Implementation Of Obsidian

- Array representation

```
data Arr s a = Arr (\IxExp -> a, Int)
type GArr a = Arr Global a
type SArr a = Arr Shared a
```

- Global Arrays
  - Live in device memory
  - Roughly 1GB
- Shared Arrays
  - Live in on-chip shared memory
  - 16KB

Implementation

1. Runing an Obsidian program produces two things
   1. Intermediate Code
   2. A symbol table, (name -> (type, size)) mapping
2. Intermediate Code goes through liveness analysis
   1. Outputs IC annotated with liveness information
3. Symbol table + annotated IC is used to build a memory map
   1. Outputs a memory map, (name -> adress) mapping
   2. Outputs IC annotated with ”number of threads needed info”
4. Memory mapped Code is generated from the output of stage 3

Implementation

- Now CUDA C code is generated from the memory mapped code.
  - Passed to CUDA C compiler
    - Taking advantage of whatever optimisations it performs.

```
__global__ static void generated(int *source0,char *gbase){
extern __shared__ char sbase[] __attribute__ ((aligned(4)));
const int tid = threadIdx.x;
const int n0 __attribute__ ((unused)) = 10;
((int *)(gbase+0))[tid] = (source0[((10 - 1) - tid)] + 1);
__syncthreads();

}
```

Conclusion

- Previous version of Obsidian showed that it is possible to get good performance out of the generated code
- The version described here is more general
  - But performance needs to be improved

- A nice platform for experimenting with algorithms on the GPU.
  - Compared to CUDA
    - Easier to experiment with different choices in
      - ☐ Where to place things in memory.
      - ☐ How much to compute per thread.

Reflections

- Working on this project has been a great learning experiance.
  - However, we do not yet have a clear picture exactly of how to write parallel programs for these kinds of processors.
    - Keep all the little processors busy
    - Use shared memory extensively
    - Large ”fan outs” is not a problem (will use efficient broadcasting capabilities)

Questions ?

References:

[1] : Fast Scan Algorithms on Graphics Processors Yuri Dotsenko Naga K. Govindaraju Peter-Pike Sloan Charles Boyd John Manferdelli Microsoft Corporation One Microsoft Way Redmond, WA 98052, USA {yurido, nagag, ppsloan, chasb, jmanfer}@microsoft.com

Performance

Experiments performed on previous version of Obsidian