Obsidian: GPU Kernel Programming in Haskell

Licentiate Seminar

Discussion leader: Prof. Andy Gill
University of Kansas

Presenter: Joel Svensson
Chalmers University of Technology

Supervisors: Mary Sheeran and Koen Claessen
GPUs
GPUs
GPUs
General Purpose Computations on GPUs (the GPGPU field)
GPGPU

• Use graphics processors for things not directly graphics related.

• Evolving area.
  • Used to be hard: “exploit” the graphics API in clever ways to make the GPU compute something
  • Greatly improved by high-level languages such as NVIDIA CUDA

• GPUs are still hard to program
NVIDIA CUDA

- Compute Unified Device Architecture
  - Architecture and programming model of modern NVIDIA GPUs.
CUDA Architecture

• A CUDA GPU is based on a scalable architecture.
  • One or more Multiprocessors.
  • Device memory.
    – Memory accessible by all multiprocessors.
Close up on a Multiprocessor

• One MP corresponds to what people often call a “core”.
  • Has a number of processing units (SP).
    – SIMD style.
    – 8, 32 SPs (Now called “CUDA cores”)
  • Local memory accessible by the SPs.
    – Kilobytes
  • Multithreading, handled by scheduler.
    – Hundreds of threads per MP
CUDA Programming Model

- Fits the scalable nature of the architecture.
- Programmer writes a “Kernel”
  - SPMD (Single Program Multiple Data) style of programming
  - Is executed on the GPU by a hierarchy of threads
    - Threads
      - Warps
    - Blocks
    - Grid
CUDA Example
CUDA Example: Dot Product

Sequential C code

```c
float dotProduct(float *x, float *y, unsigned int n) {
    float r = 0.0f;
    for (int i = 0; i < n; ++i) {
        r += x[i] * y[i];
    }
    return r;
}
```

- Not what you want to do on a GPU
CUDA dotProduct

- Split computation into a multiplication phase and a summing phase.
  - Both end up being parallelizable.
CUDA dotProduct: elementwise multiplication

CUDA Code

```c
__global__ void multKernel(float *result, float *x, *float *y) {
    unsigned int gtid = blockIdx.x * blockDim.x + threadIdx.x;
    result[gtid] = x[gtid] * y[gtid];
}
```

- threadIdx: identifies a thread within a block (a vector of 3 integers)
- blockIdx: identifies a block within a grid (a vector of 2 integers)
- blockDim: specifies the extents of the blocks (a vector of 3 integers)
- __global__: specifies that this function is a CUDA kernel.
CUDA dotProduct: summation

- Sum up the products.

\[
\begin{array}{cccccccc}
  a_0 & a_1 & a_2 & a_3 & a_4 & a_5 & a_6 & a_7 \\
  \Sigma a_0 a_1 & a_1 & \Sigma a_2 a_3 & a_3 & \Sigma a_4 a_5 & a_5 & \Sigma a_6 a_7 & a_7 \\
  \Sigma a_0 a_3 & a_1 & \Sigma a_2 a_3 & a_3 & \Sigma a_4 a_7 & a_5 & \Sigma a_6 a_7 & a_7 \\
  \Sigma a_0 a_7 & a_1 & \Sigma a_2 a_3 & a_3 & \Sigma a_4 a_7 & a_5 & \Sigma a_6 a_7 & a_7
\end{array}
\]
CUDA Code

```c
__global__ void sumKernel(float *result, float *x){
    unsigned int tid = threadIdx.x;

    extern __shared__ float sm[];

    sm[tid] = x[blockIdx.x * blockDim + tid];

    for (int i = 1; i < blockDim.x; i *=2) {
        __syncthreads();
        if (tid % (2*i) == 0) {
            sm[tid] += sm[tid + i];
        }
    }
    if (tid == 0) {
        result[blockIdx.x] = sm[0];
    }
}
```

- Sums up an array “that fits in a block”
- Uses shared memory
- Uses `__syncthreads()`
CUDA dotProduct

• We have:
  • A CUDA kernel that can multiply two arrays (of “any” size) element-wise.
  • A CUDA kernel that can sum up arrays
    – 32, 64, 128, 256, … 1024 elements for example.
CUDA dotProduct

- Launching Kernels on the GPU

CUDA Host Code

```c
multKernel<<<128,512,0>>>(device_x, device_x, device_y);
sumKernel<<<128,512,512*sizeof(float)>>>(result, device_x);
sumKernel<<<1,128,128*sizeof(float)>>>(result, result);
```

- Computes the dotProduct of two (128 * 512)-element arrays.
- `device_x`, `device_y` and `result` arrays in “device memory”
CUDA dotProduct: potential optimizations

- Create a fused “mult-sum” kernel
  ```
  multSumKernel<<<128,512,0>>>(result,device_x,device_y);
  sumKernel<<<1,128,128*sizeof(float)>>>(result,result);
  ```
- Compute more “results” per thread
  - Use fewer threads in total.
  - Better utilize memory bandwidth.
- Unroll loops, specialize kernel to a particular block-size.
  - Many few-threaded blocks, few many-threaded blocks?
- Changes like these means:
  - Rethink indexing computations.
  - Much rewriting of your kernels.
CUDA

- CUDA is a huge improvement.
- CUDA C is not C.
- Kernels are not easily composed.
- “tweaking” kernels is cumbersome.
Obsidian
Obsidian

• A language for GPU kernel implementation.
  • Embedded in Haskell.

• Goals
  • Compose kernels easily
  • Raise the level of abstraction while maintaining control.
    – Use combinators (higher order functions).
    – Building blocks view on kernel implementation.
    – Think in “data-flow” and “connection-patterns” instead of indexing
  • Generate CUDA code
Obsidian: Small Example

• Increment every element in an array

```haskell
incrAll :: Num a => Arr a :-> Arr a
incrAll = pure$ fmap (+1)
```

• Can be executed on the GPU

```haskell
*Main> execute incrAll [0..9 :: IntE]
[1,2,3,4,5,6,7,8,9,10]
```
Or look at the generated CUDA.

```haskell
*Main> execute incrAll [0..9 :: IntE]
[1,2,3,4,5,6,7,8,9,10]
```

```haskell
genKernel "incrKern" incrAll (mkArr undefined 10 :: Arr IntE)
```

Generated CUDA

```c
__global__ void incrKern(word* input0,word* result0){
  unsigned int tid = (unsigned int)threadIdx.x;
  ix_int(result0,(tid + (10 * blockIdx.x))) =
    (ix_int(input0,(tid + (10 * blockIdx.x))) + 1);
}
```
Obsidian: Small Example
The details

\[ \text{incrAll :: Num } a \rightarrow \text{Arr } a :\rightarrow \text{Arr } a \]
\[ \text{incrAll} = \text{pure} \circ \text{fmap } (+1) \]
Obsidian: Small Example

The details

• **Arrays (Arr a)**

```haskell
data Arr a = Arr (IndexE -> a) Int
```

• Describes how to compute an element given an index.

```haskell
rev :: Arr a -> Arr a
rev arr = mkArr ixf n
  where
    ixf ix = arr ! (fromIntegral (n-1) - ix)
    n = len arr
```

```haskell
fmap f arr = mkArr (\ix -> f (arr ! ix)) (len arr)
```
*Main> execute incrAll [0..9 :: IntE]
[1,2,3,4,5,6,7,8,9,10]

genKernel "incrKern" incrAll (mkArr undefined 10 :: Arr IntE)

data DExp
  = LitInt Int
  | LitUInt Word32
  | LitBool Bool
  | LitFloat Float
  | Op2 Op2 DExp DExp
  | Op1 Op1 DExp
  | If DExp DExp Dexp
...

type Exp a = E DExp
type IntE  = Exp Int
Building Blocks

\[
\left( \begin{array}{cccc}
a_0 & a_1 & a_2 & a_3 \\
 & b_0 & b_1 & b_2 & b_3
\end{array} \right)
\]

zip

\[
\left( \begin{array}{cccc}
(a_0, b_0) & (a_1, b_1) & (a_2, b_2) & (a_3, b_3)
\end{array} \right)
\]

unzip

\[
\left( \begin{array}{cccc}
a_0 & a_2 & a_4 & a_6 \\
 & a_1 & a_3 & a_5 & a_7
\end{array} \right)
\]

riffle

\[
\left( \begin{array}{cccccccc}
a_0 & a_1 & a_2 & a_3 & a_4 & a_5 & a_6 & a_7
\end{array} \right)
\]

unriffle

\[
\left( \begin{array}{cccccccc}
a_0 & a_1 & a_2 & a_3 & a_4 & a_5 & a_6 & a_7
\end{array} \right)
\]
Obsidian:
Expanding the small example

incrRev0 :: Num a => Arr a ->> Arr a
incrRev0 = incrAll ->> pure rev

• Sequential composition of Kernels.

*Main> execute incrRev0 [0..9 :: IntE]
[10,9,8,7,6,5,4,3,2,1]
Obsidian:
Expanding the small example

```haskell
incrRev1 :: Num a => Arr a -> Arr a
incrRev1 = pure$ rev . fmap (+1)
```

- Could have used Haskell function composition for same result.

```haskell
*Main> execute incrRev1 [0..9 :: IntE]
[10,9,8,7,6,5,4,3,2,1]
```
Obsidian:
Expanding the small example

*Main> execute incrRev2 [0..9 :: IntE]
[10,9,8,7,6,5,4,3,2,1]

- Store intermediate result in shared memory.

incrRev2 :: (Flatten a, Num a) => Arr a :-> Arr a
incrRev2 = incrAll ->- sync ->- pure rev
The generated code

```c
__global__ void incrRev1(word* input0, word* result0) {
    unsigned int tid = (unsigned int) threadIdx.x;
    ix_int(result0, (tid + (10 * blockIdx.x))) =
        (ix_int(input0, ((9 - tid) + (10 * blockIdx.x))) + 1);
}

__global__ void incrRev2(word* input0, word* result0) {
    unsigned int tid = (unsigned int) threadIdx.x;
    extern __shared__ unsigned int s_data[];
    word __attribute__((unused)) *sm1 = &s_data[0];
    word __attribute__((unused)) *sm2 = &s_data[10];
    ix_int(sm1, tid) =
        (ix_int(input0, (tid + (10 * blockIdx.x))) + 1);
    __syncthreads();
    ix_int(result0, (tid + (10 * blockIdx.x))) =
        ix_int(sm1, (9 - tid));
}
```
Seen so far

- Create kernels.
  pure
- Sequential composition of kernels.
  \(-\rightarrow-\)
- Store intermediate results in shared memory or “fuse” computations.
  sync
- Combinators and permutations.
  fmap and rev
- Length of array specifies degree of parallelism
data a :-> b where
   Pure :: (a -> b) -> (a :-> b)
   Sync :: (a -> Arr FData) -> (Arr FData :-> b) -> (a :-> b)
Obsidian: dotProduct

- Same approach as in the CUDA example
  - A multKernel and one sumKernel

```haskell
multKern :: (Arr FloatE, Arr FloatE) -> Arr FloatE
multKern = pure$ zipWith (*)
```
Obsidian: dotProduct

- Implement a similar binary tree shaped summation as in the CUDA example.

```haskell
reduce :: Flatten a => Int -> (a -> a -> a) -> Arr a :-> Arr a
reduce 0 f = pure id
reduce n f = pure op ->- sync ->- reduce (n-1) f
  where
    op = fmap (uncurry f) . pair
```

- Sync is used to obtain parallelism.
Obsidian: dotProduct

dotProduct :: Int -> (Arr FloatE, Arr FloatE) :-> Arr FloatE
dotProduct n = multKern ->- sync ->- reduce n (+)
__global__ void dotProduct(word* input0, word* input1, word* result0) {
  unsigned int tid = (unsigned int)threadIdx.x;
  extern __shared__ unsigned int s_data[];
  word __attribute__((unused)) *sm1 = &s_data[0];
  word __attribute__((unused)) *sm2 = &s_data[8];
  ix_float(sm1, tid) = (ix_float(input0, (tid + (8 * blockIdx.x))) * 
                        ix_float(input1, (tid + (8 * blockIdx.x))));

  __syncthreads();
  if (tid < 4) {
    ix_float(sm2, tid) = (ix_float(sm1, (tid << 1)) +
                           ix_float(sm1, ((tid << 1) + 1)));
  }
  __syncthreads();
  if (tid < 2) {
    ix_float(sm1, tid) = (ix_float(sm2, (tid << 1)) +
                           ix_float(sm2, ((tid << 1) + 1)));
  }
  __syncthreads();
  if (tid < 1) {
    ix_float(sm2, tid) = (ix_float(sm1, (tid << 1)) +
                           ix_float(sm1, ((tid << 1) + 1)));
  }
  __syncthreads();
  if (tid < 1) {
    ix_float(result0, (tid + blockIdx.x)) = ix_float(sm2, tid);
  }
}
CUDA dotProduct: potential optimizations

- Create a fused "mult-sum" kernel
  ```c
  multSumKernel<<<128,512,0>>>(result,device_x,device_y);
  sumKernel<<<1,128,128*sizeof(float)>>>(result,result);
  ```

- Compute more "results" per thread
  - Use fewer threads in total.
  - Better utilize memory bandwidth.

- Unroll loops, specialize kernel to a particular block-size
  - Many few-threaded blocks, few many-threaded blocks?

- Changes like these means:
  - Rethink indexing computations.
  - Much rewriting of your kernels.

- The reversed is hard
Case Studies
Case Studies

- Reduction
- **Dot product**
- Mergers
- Sorters
- **Parallel prefix**
Case Studies: dotProduct

dotProduct0 :: Int -> (Arr FloatE, Arr FloatE) :-> Arr FloatE
dotProduct0 n = multKern ->- sync ->- reduce n (+)

dotProduct1 :: Int -> (Arr FloatE, Arr FloatE) :-> Arr FloatE
dotProduct1 n = multKern ->- reduce n (+)

<table>
<thead>
<tr>
<th>Kernel</th>
<th>Threads per block</th>
<th>Elems. per block</th>
<th>ms per block</th>
</tr>
</thead>
<tbody>
<tr>
<td>dotProduct0</td>
<td>256</td>
<td>2*256</td>
<td>0.0015</td>
</tr>
<tr>
<td>dotProduct1</td>
<td>128</td>
<td>2*256</td>
<td>0.0013</td>
</tr>
<tr>
<td>CUDA</td>
<td>256</td>
<td>2*256</td>
<td>0.0030</td>
</tr>
</tbody>
</table>
Case Studies: Parallel prefix

• Given an associative binary operator, $\oplus$, and an array $s$, compute:

\[
\begin{align*}
    r[0] &= s[0] \\
    r[1] &= s[0] \oplus s[1] \\
    \vdots \\
    r[n] &= s[0] \oplus \ldots \oplus s[n]
\end{align*}
\]

• A building block in many parallel algorithms
  • Stream compaction
  • Sorting
  • Many examples in: Prefix Sums and Their Applications - Guy E. Blelloch
Case Studies:
Parallel prefix
Case Studies: Parallel prefix

```
sklansky :: (Flatten a, Choice a) =>
    Int -> (a -> a -> a) -> (Arr a :-> Arr a)

sklansky 0 op = pure id
sklansky n op = two (sklansky (n-1) op) ->- pure (fan op)
    ->- sync

fan op arr = conc (a1, (fmap (op c) a2))
where (a1,a2) = halve arr
    c = a1 ! (fromIntegral (len a1 - 1))
```
Obsidian

- Language for implementing kernels
  - No “kernel coordination”

- Kernel composition is easy
  - sequentially
    - two special kind of parallel composition

- Kernels use shared memory for intermediate results.
Related Work

- Data.Array.Accelerate
  - Higher level language compared to Obsidian.
  - Not the same detailed control.
  - Most complete and well developed tool for Haskell based GPU programming today.
Related Work

• Nikola
  • Higher level language compared to Obsidian.
    – Similar to Accelerate in that sense.
  • Effort seems to be in developing methods for EDSL implementation.
Related Work

• Copperhead (Data-parallel python)
  • Compiles a subset of Python into CUDA for GPU execution.
  • Very fresh.
Conclusions and Future work

- Obsidian strong sides.
  - Elegant descriptions of kernels.
  - Compositional kernels.

- Kernel Coordination is needed.
  - Depending on path of GPU evolution.

- Kernel Performance can be improved.
  - Hard to compare performance to that of related work.
Extra Generated Code
```c
__global__ void sklansky128(word* input0, word* result0) {
  unsigned int tid = (unsigned int) threadIdx.x;
  extern __shared__ unsigned int s_data[];
  word __attribute__((unused))*sm1 = &s_data[0];
  word __attribute__((unused))*sm2 = &s_data[128];
  ix_int(sm1, tid) = (((tid & 0xffffff81) < 1) ?
    ix_int(input0, (tid + (128 * blockIdx.x))) :
    (ix_int(input0, ((tid & 0x7e) + (128 * blockIdx.x))) +
    ix_int(input0, (tid + (128 * blockIdx.x)))));

  __syncthreads();
  ix_int(sm2, tid) = (((tid & 0xffffff83) < 2) ?
    ix_int(sm1, tid) :
    (ix_int(sm1, ((tid & 0x7c) | 0x1)) + ix_int(sm1, tid)));
  __syncthreads();
  ix_int(sm1, tid) = (((tid & 0xffffff87) < 4) ?
    ix_int(sm2, tid) :
    (ix_int(sm2, ((tid & 0x78) | 0x3)) + ix_int(sm2, tid)));
  __syncthreads();
  ix_int(sm2, tid) = (((tid & 0xffffff8f) < 8) ?
    ix_int(sm1, tid) :
    (ix_int(sm1, ((tid & 0x70) | 0x7)) + ix_int(sm1, tid)));
  __syncthreads();
  ix_int(sm1, tid) = (((tid & 0xffffff9f) < 16) ?
    ix_int(sm2, tid) :
    (ix_int(sm2, ((tid & 0x60) | 0xf)) + ix_int(sm2, tid)));
  __syncthreads();
  ix_int(sm2, tid) = (((tid & 0xffffffbf) < 32) ?
    ix_int(sm1, tid) :
    (ix_int(sm1, ((tid & 0x40) | 0x1f)) + ix_int(sm1, tid)));
  __syncthreads();
  ix_int(sm1, tid) = ((tid < 64) ?
    ix_int(sm2, tid) :
    (ix_int(sm2, 63) + ix_int(sm2, tid)));
  __syncthreads();
  ix_int(result0, (tid + (128 * blockIdx.x))) = ix_int(sm1, tid);
}
```