GPU programming in Obsidian

Ack: Obsidian is developed by Joel Svensson.

Accelerate

Get acceleration from your GPU by writing familiar combinators

Hand tuned skeleton templates

Compiler cleverness to fuse and memoise the resulting kernels

Leaves a gap between the programmer and the GPU (which most people want)

Obsidian

Can we bring FP benefits to GPU programming, without giving up control of low level details?

This is an instance of the research questions in our big SSF project called Resource Aware Functional Programming

Assumptions

To get really good performance from a GPU, one must control

- use of memory
- memory access patterns
- synchronisation points
- where the boundaries of kernels are
- patterns of sequential code (control of task size)

Vital to be able to experiment with variants on a kernel easily

Assumptions

To get really good performance from a GPU, one must control

use

me

wh

pa

Vital t
kerne

We aim to give the programmer this control

We avoid compiler cleverness!

Cost model should be entirely transparent

Building blocks

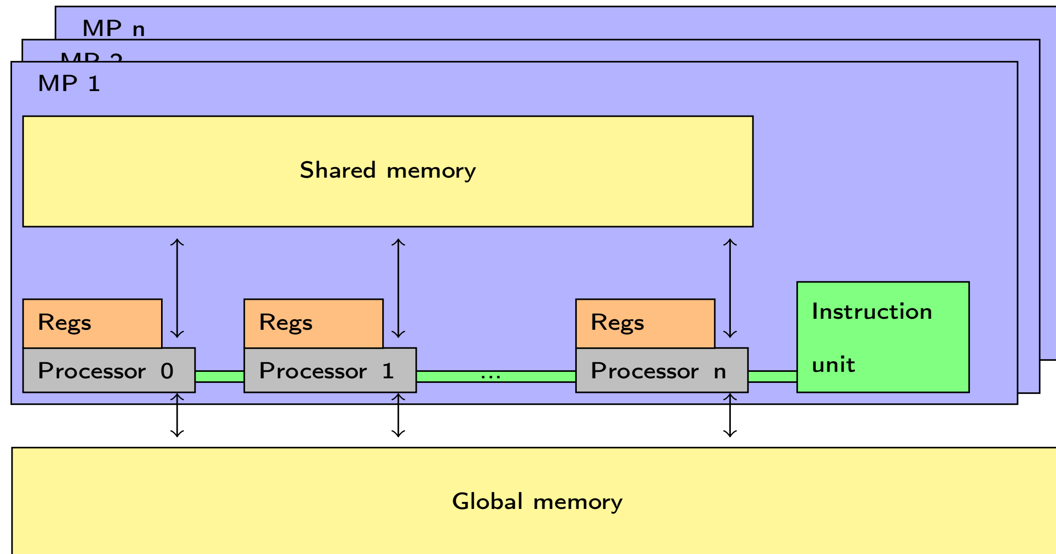
Embedded DSL in Haskell

Pull and push arrays (remember Jean-Philippe's lecture!)

Use of types to allow "hierarchy-polymorphic" functions (Thread, Warp, Block, Grid)

A form of virtualisation to remove arbitrary limits like max #threads per block

GPU



CUDA programming model

Single Program Multiple Threads

Kernel = Function run N times by N threads

Hierarchical thread groups

Associated memory hierarchy

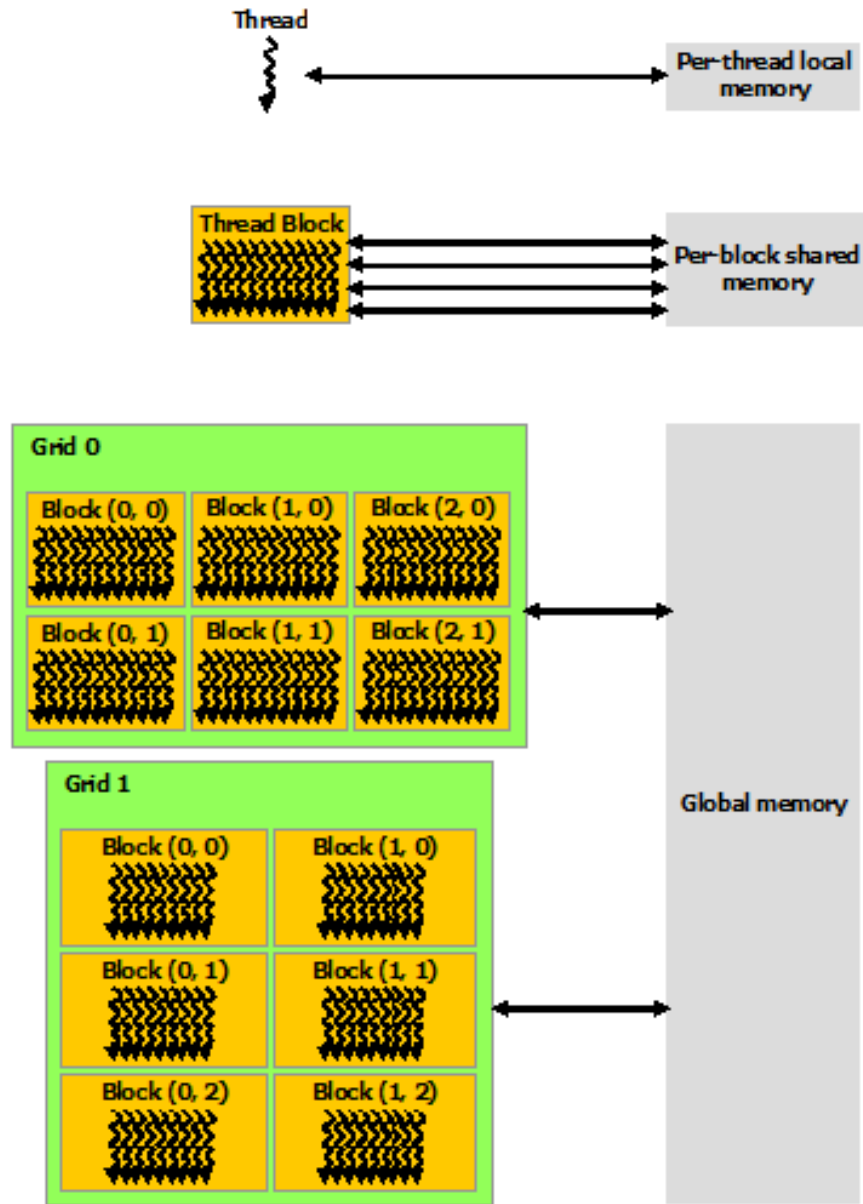


Image from http://docs.nvidia.com/cuda/cuda-c-programming-guide/#memory-hierarchy

The flow of kernel execution

Initialize/acquire the device (GPU)

Allocate memory on the device (GPU)

Copy data from host (CPU) to device (GPU)

Execute the kernel on the device (GPU)

Copy result from device (GPU) to host (CPU)

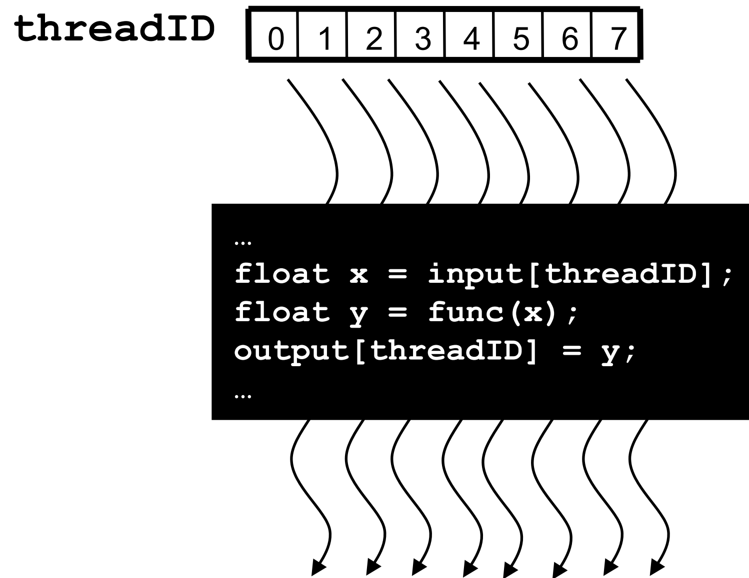
Deallocate memory on device (GPU)

Release device (GPU)

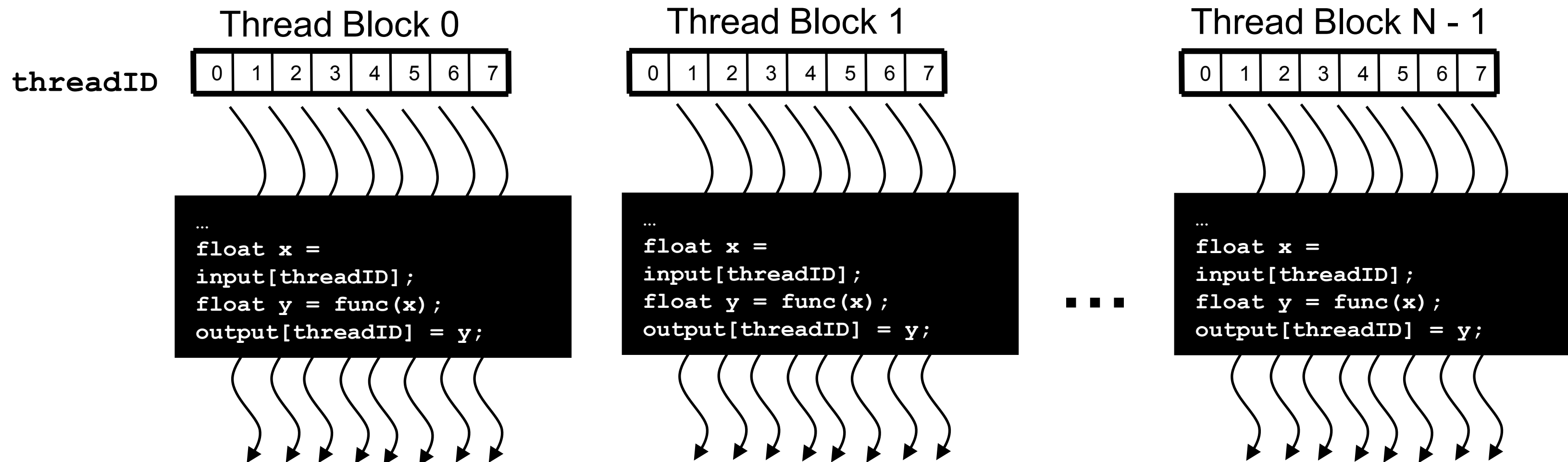
CUDA kernel

Executed by an array of Threads

Each thread has an ID that is used to compute memory addresses and make control decisions



Blocks



Threads within a block communicate via shared memory and barrier synchronisation (__ syncthreads();)

Threads in different blocks cannot cooperate

Memory access patterns

Some patterns of global memory access can be coalesced. Others cannot. Missing out on coalescing ruins performance!

Global memory works best when adjacent threads access a contiguous block

For shared memory, successive 32 bit words are in different banks. Multiple simultaneous access to a bank = bank conflict = another way to ruin performance. Conflicting accesses are serialised.

Thread ID is usually built from

blockIdx	Block index within a grid	uint3
blockDim	Dimension of the block	dim3
threadIdx	Thread index within a block	uint3

gridDim gives the dimensions of the grid (the number of blocks in each dimension)

We'll use linear blocks and grids (easier to think about)

For more info about CUDA see https://developer.nvidia.com/gpu-computing-webinars
esp. the 2010 intro webinars

First CUDA kernel

```
__global__ void inc(float *i, float *r){
  unsigned int ix = blockIdx.x * blockDim.x + threadIdx.x;
  r[ix] = i[ix]+1;
}
```

Host code

```
#include <stdio.h>
#include <cuda.h>
#define BLOCK_SIZE 256
#define BLOCKS 1024
#define N (BLOCKS * BLOCK_SIZE)

int main(){
  float *v, *r;
  float *dv, *dr;

  v = (float*)malloc(N*sizeof(float));
  r = (float*)malloc(N*sizeof(float));

  //generate input data
  for (int i = 0; i < N; ++i) {
     v[i] = (float)(rand () % 1000) / 1000.0; }

  /* Continues on next slide */
```

Host code

```
  cudaMalloc((void**)&dv, sizeof(float) * N );
  cudaMalloc((void**)&dr, sizeof(float) * N );

  cudaMemcpy(dv, v, sizeof(float) * N,cudaMemcpyHostToDevice);

  inc<<<BLOCKS, BLOCK_SIZE,0>>>(dv,dr);

  cudaMemcpy(r, dr, sizeof(float) * N, cudaMemcpyDeviceToHost);

  cudaFree(dv);
  cudaFree(dr);

  for (int i = 0; i < N; ++i) {
      printf("%f ", r[i]); }
  printf("\n");

  free(v);
  free(r);
}
```

Obsidian

```
incLocal arr = fmap (+1) arr
```

Building an AST just like in Accelerate

Obsidian Pull arrays

```
incLocal :: Pull Word32 EWord32 -> Pull Word32 EWord32
incLocal arr = fmap (+1) arr
```

Pull size element-type

Immutable

Obsidian Pull arrays

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

(length and function from index to value, the *read-function*, see Elliott's Pan,
also called delayed arrays)

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

A consumer of a pull array needs to iterate over those indices of the array it is interested in and apply the pull array function at each of them.

Fusion for free

```
fmap f (Pull n ixf) = Pull n (f . ixf)
```

Example

```
incLocal arr = fmap (+1) arr
```

This says what the computation should do

How do we lay it out on the GPU??

```
incPar :: Pull EWord32 EWord32 -> Push Block EWord32 EWord32
incPar = push . incLocal
```

`push` converts a pull array to a push array and pins it to a particular part of the GPU hierarchy

No cost associated with pull to push conv.

Key to getting fine control over generated code

GPU Hierarchy in types

```
-- A hierarchy!
data Step a -- A step in the hierarchy
data Zero

type Thread = Zero
type Warp   = Step Thread
type Block  = Step Warp
type Grid   = Step Block
```

Program data type

```
data Program t a where

  Identifier :: Program t Identifier

  Assign :: Scalar a
            => Name
            -> [Exp Word32]
            -> (Exp a)
            -> Program Thread ()

...

   -- use threads along one level
   -- Warp, Block, Grid.
   ForAll ::  EWord32
           -> (EWord32 -> Program Thread ())
           -> Program t ()     -- (really atleast Step t) !

. . .
```

Obsidian push arrays

```
data Push p l a =
  Push l filler-function
```

Length

a function that generates a loop at a particular level of the hierarchy

The general idea of push arrays is due to Koen Claessen

Obsidian push arrays

```
data Push p l a =
  Push l (receiver -> Program p ())
```

generates a Program at level p

Push array only allows bulk request to push ALL elements via a receiver function

The general idea of push arrays is due to Koen Claessen

Obsidian push arrays

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

Each push array is waiting to be passed a receiver function, which takes a value (a) and index (EWord32), and generates single-threaded code to store or use that value. Given a receiver, a push array is then responsible for generating a program that traverses the push array's iteration space, invoking the receiver as many times as necessary.

The general idea of push arrays is due to Koen Claessen

Obsidian push array

A push array is a length and a filler function

Filler function encodes a loop at level t in the hierarchy

Its argument is a receiver function

Push array allows only a bulk request to push all elements via a receiver function

When invoked, the filler function creates the loop structure, but it
inlines the code for the receiver inside the loop.

A push array with elements computed by f and receiver rcv corresponds to a loop
for (I in [1,N]) {rcv(i,f(i));}

When forced to memory, each invocation of rcv would write one memory location
A[i] = f(i)

Note

Neither pull nor push arrays are manifest

Both fuse by default.

Both immutable.

Argh. Why two types of array??

Concatenation of pull arrays is inefficient.
Introduces conditionals (which can ruin performance)

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.

Pull good for reading. Push good for writing. Pull -> Push functions common

Programming the hierarchy

```
-- Enter into hierarchy
tConcat :: Pull l (Push Thread Word32 a) -> Push t l a

-- Step upwards in hierarchy
pConcat :: Pull l (Push Word32 t a) -> Push (Step t) l a

-- Remain on a level of the hierarchy
sConcat :: Pull l (Push t Word32 a) -> Push t l a
```

From our recent paper (which I will post)

Combinators

```
pMap f n = pConcat . (fmap f). splitUp n

tMap f n = tConcat . (fmap f) . splitUp n

sMap f n = sConcat . (fmap f) . splitUp n
```

Last night's thoughts ☺

Need to think harder about API to user!

Combinators

```
pMap f n = pConcat . (fmap f). splitUp n

tMap f n = tConcat . (fmap f) . splitUp n

sMap f n = sConcat . (fmap f) . splitUp n



e.g.

pMap
  :: ASize l =>
     (SPull a1 -> SPush t a) -> Word32 ->
      Pull l a1 -> Push (Step t) l a
```

Back to example

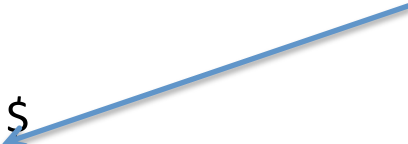
```
increment1 = pMap (push . incLocal)
```

Back to example

```
increment1 = pMap (push . incLocal)
```

Gives   loopnest like   parfor parfor

Threads per block



```
getIncPar = putStrLn $
       genKernel 256 "incPar"
         (increment1 256 :: DPull EWord32 -> DPush Grid EWord32)
```

Elements per block

```
extern "C" __global__ void incPar(uint32_t* input0, uint32_t n0,
                                  uint32_t* output1)
{
    uint32_t bid = blockIdx.x;
    uint32_t tid = threadIdx.x;

    for (int b = 0; b < n0 / 256U / gridDim.x; ++b) {
        bid = blockIdx.x * (n0 / 256U / gridDim.x) + b;
        output1[bid * 256U + tid] = input0[bid * 256U + tid] + 1U;
        bid = blockIdx.x;
        __syncthreads();
    }
    bid = gridDim.x * (n0 / 256U / gridDim.x) + blockIdx.x;
    if (blockIdx.x < n0 / 256U % gridDim.x) {
        output1[bid * 256U + tid] = input0[bid * 256U + tid] + 1U;
    }
    bid = blockIdx.x;
    __syncthreads();
}
```

```
getIncPar = putStrLn $
          genKernel 128 "incPar"
            (increment1 256 :: DPull EWord32 -> DPush Grid EWord32)
```

```
extern "C" __global__ void incPar(uint32_t* input0, uint32_t n0,
                                  uint32_t* output1)
{
    uint32_t bid = blockIdx.x;
    uint32_t tid = threadIdx.x;

    for (int b = 0; b < n0 / 256U / gridDim.x; ++b) {
        bid = blockIdx.x * (n0 / 256U / gridDim.x) + b;
        for (int i = 0; i < 2; ++i) {
            tid = i * 128 + threadIdx.x;
            output1[bid * 256U + tid] = input0[bid * 256U + tid] + 1U;
        }
        tid = threadIdx.x;
        bid = blockIdx.x;
        __syncthreads();
    }
    bid = gridDim.x * (n0 / 256U / gridDim.x) + blockIdx.x;
    if (blockIdx.x < n0 / 256U % gridDim.x) {
        for (int i = 0; i < 2; ++i) {
            tid = i * 128 + threadIdx.x;
            output1[bid * 256U + tid] = input0[bid * 256U + tid] + 1U;
        }
        tid = threadIdx.x;
    }
    bid = blockIdx.x;
    __syncthreads();
}
```

```
increment3 m  = pMap (tMap (push . incLocal) m)
```

parfor parfor for

```
extern "C" __global__ void incPar1(uint32_t* input0, uint32_t n0,
                                   uint32_t* output1)
{
    uint32_t bid = blockIdx.x;
    uint32_t tid = threadIdx.x;

    for (int b = 0; b < n0 / 256U / gridDim.x; ++b) {
        bid = blockIdx.x * (n0 / 256U / gridDim.x) + b;
        if (threadIdx.x < 8) {
            tid = 0 + threadIdx.x;
            for (int i0 = 0; i0 < 32U; ++i0) {
                output1[bid * 256U + (tid * 32U + i0)] = input0[bid *
                                                                256U +
                                                                (tid *
                                                                 32U +
                                                                 i0)] + 1U;
            }
        }
        tid = threadIdx.x;
        bid = blockIdx.x;
        __syncthreads();
    }
    bid = gridDim.x * (n0 / 256U / gridDim.x) + blockIdx.x;
    if (blockIdx.x < n0 / 256U % gridDim.x) {
        if (threadIdx.x < 8) {
            tid = 0 + threadIdx.x;
            for (int i0 = 0; i0 < 32U; ++i0) {
                output1[bid * 256U + (tid * 32U + i0)] = input0[bid *
                                                                256U +
                                                                (tid *
                                                                 32U +
                                                                 i0)] + 1U;
            }
        }
        tid = threadIdx.x;
    }
    bid = blockIdx.x;
    __syncthreads();
}
```

```
increment4 m = sMap (pMap (push. incLocal) m)
```

```
extern "C" __global__ void incPar1(uint32_t* input0, uint32_t n0,
                                   uint32_t* output1)
{
    for (int i0 = 0; i0 < n0 / 256U; ++i0) {
        for (int b = 0; b < 8U / gridDim.x; ++b) {
            bid = blockIdx.x * (8U / gridDim.x) + b;
            if (threadIdx.x < 32) {
                tid = 0 + threadIdx.x;
                output1[i0 * 256U + (bid * 32U + tid)] = input0[i0 * 256U +
                                                                 (bid *
                                                                  32U +
                                                                  tid)] +
                    1U;
            }
            tid = threadIdx.x;
            bid = blockIdx.x;
            __syncthreads();
        }
        bid = gridDim.x * (8U / gridDim.x) + blockIdx.x;
        if (blockIdx.x < 8U % gridDim.x) {
            if (threadIdx.x < 32) {
                tid = 0 + threadIdx.x;
                output1[i0 * 256U + (bid * 32U + tid)] = input0[i0 * 256U +
                                                                 (bid *
                                                                  32U +
                                                                  tid)] +
                    1U;
            }
            tid = threadIdx.x;
        }
        bid = blockIdx.x;
        __syncthreads();
    }
}
```

Gentle persuasion by type system

```
incrementwrong m = tMap (pMap (push . incLocal) m)
```

```
LecEx.hs:71:26:
    Couldn't match type `Step t0' with `Zero'
    Expected type: SPull EWord32 -> SPush Thread EWord32
      Actual type: Pull Word32 EWord32 -> Push (Step t0)
Word32 EWord32
    In the return type of a call of `pMap'
    In the first argument of `tMap', namely
      `(pMap (push . incLocal) m)'
    In the expression: tMap (pMap (push . incLocal) m)
Failed, modules loaded: none.
```

Autotuning springs to mind!

Recursion is unwound

```
sumUp :: Pull Word32 EWord32 -> EWord32
sumUp arr
  | lenarr==1=arr!0
  | otherwise =
     let (a1,a2) = halve arr
         arr2 = zipWith (+) a1 a2
     in sumUp arr2
```

force

For making arrays manifest (in memory) to share results between threads

Forcing a pull array results in a loop that computes the indexing function at each index

Forcing a push array instantiates the iteration schema encoded in the array and writes all elements to memory using that strategy

```
sumUp' :: Pull Word32 EWord32 → Program Block
EWord32
sumUp' arr
  | len arr==1=return(arr!0)
  | otherwise =
   do let (a1,a2) = halve arr
      arr2 ← forcePull (zipWith (+) a1 a2)
      sumUp' arr2
```

Gives a tree shaped parallel reduction

Hierarchy agnostic function

```
agnostic arr =
do imm1 ← forcePull (fmap (+1) arr)
   imm2 ← forcePull (fmap (*2) imm1)
   imm3 ← forcePull (fmap (+3) imm2)
return (push imm3)
```

Can be instantiated at Block level or below Why?

Behaves differently at different levels

Block level (code outline)

```
parfor (i in 0..255) {
imm1[i] = input[blockID * 256 + i] + 1;
__syncthreads();

imm2[i] = imm1[i] * 2;
__syncthreads();

imm3[i] = imm2[i] + 3;
__syncthreads();
}
```

Warp level

```
parfor (i in 0..255) {
warpID = i / 32;
warpIx = i % 32;
imm1[warpID * 32 + warpIx] = input[blockID * 256 + warpID * 32 + warpIx] + 1;
imm2[warpID * 32 + warpIx] = imm1[warpID * 32 + warpIx] * 2;
imm3[warpID * 32 + warpIx] = imm2[warpID * 32 + warpIx] + 3;
}
```

Warp level

```
parfor (i in 0..255) {
warpID = i / 32;
warpIx = i % 32;
imm1[warpID * 32 + warpIx] = input[blockID * 256 + warpID * 32 + warpIx] + 1;
imm2[warpID * 32 + warpIx] = imm1[warpID * 32 + warpIx] * 2;
imm3[warpID * 32 + warpIx] = imm2[warpID * 32 + warpIx] + 3;
}
```

No synchronsations!

Warp level

```
parfor (i in 0..255) {
warpID = i / 32;
warpIx = i % 32;
imm1[warpID * 32 + warpIx] = input[blockID * 256 + warpID * 32 + warpIx] + 1;
imm2[warpID * 32 + warpIx] = imm1[warpID * 32 + warpIx] * 2;
imm3[warpID * 32 + warpIx] = imm2[warpID * 32 + warpIx] + 3;
}
```

Beware, though that in CUDA 6 it is no longer the case that syncs in warps can be dropped …. (so Obsidian will have to adapt)

Case study, reductions, one example

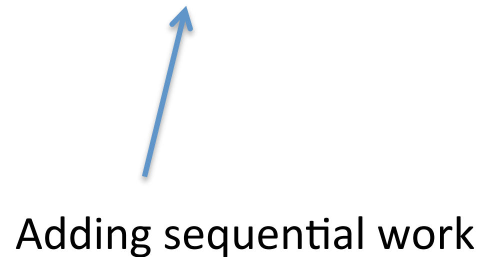
```
red5' :: MemoryOps a ⇒ Word32 -> (a -> a -> a) -> Pull Word32 a -> Program Block a
red5' n f arr =
do
   arr2 <-  force (tConcat (fmap (seqReduce f) (coalesce n arr)))
   red3 2 f arr2
```

See the draft paper on the lectures page

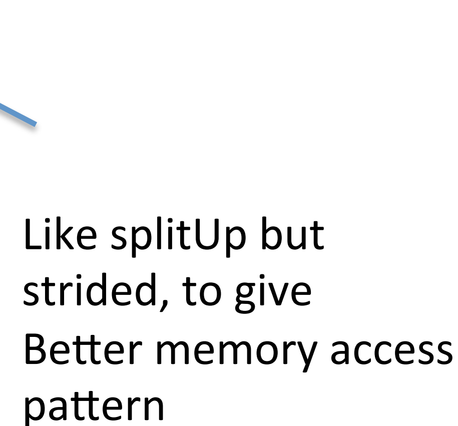
Case study, reductions, one example

```
red5' :: MemoryOps a ⇒ Word32 -> (a -> a -> a) -> Pull Word32 a -> Program Block a
red5' n f arr =
do
   arr2 <-  force (tConcat (fmap (seqReduce f) (coalesce n arr)))
   red3 2 f arr2
```

reuse

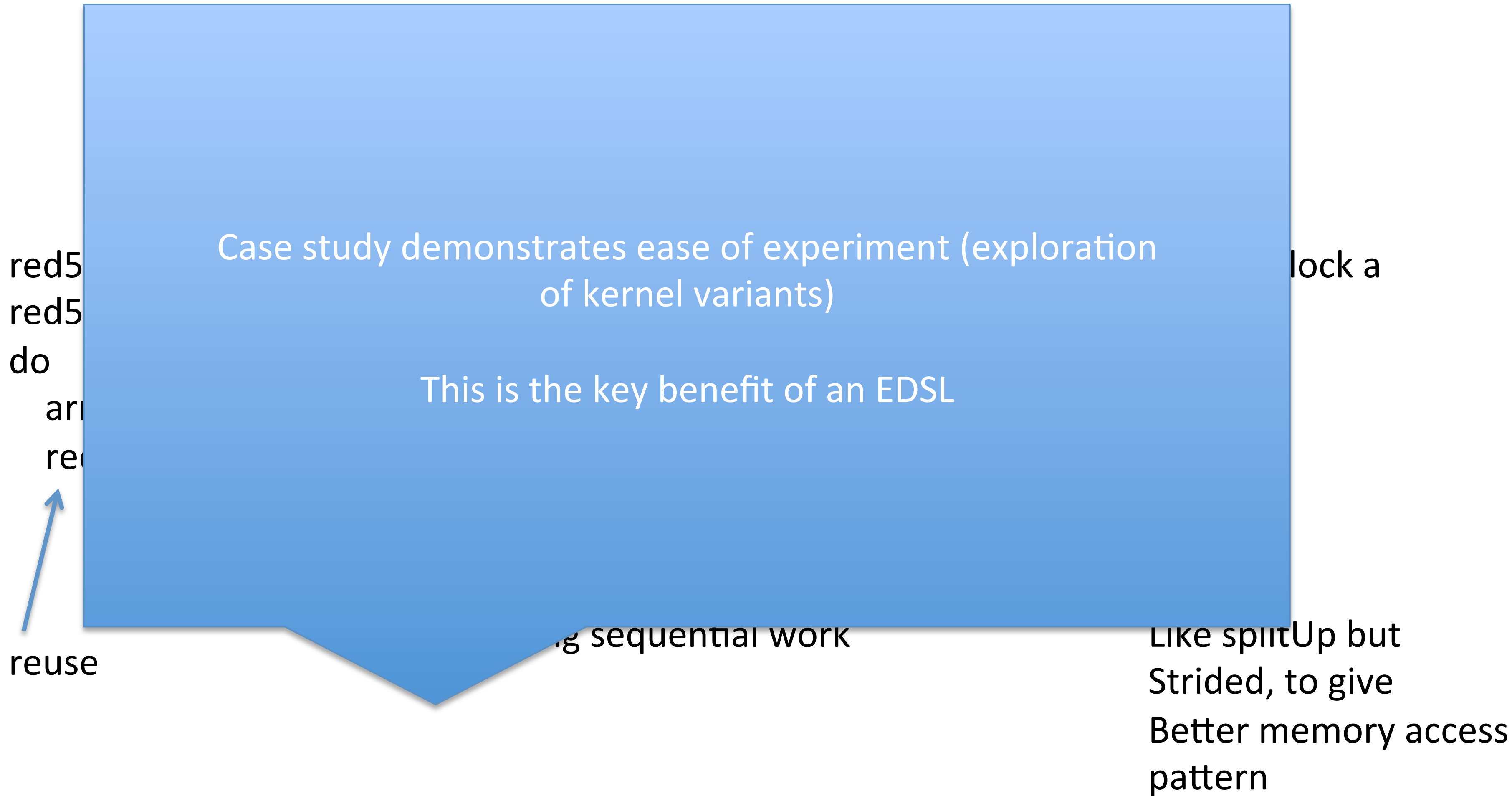


Adding sequential work



Like splitUp but
strided, to give
Better memory access
pattern

Case study, reductions, one example



Reduce 2^24 elements (1000 times)

Variant	Parameter	Seconds	Parameter*	Seconds*
ACC	Loop	2.767		
ACC	AWhile	2.48		
Red1	256 threads	0.751	32	2.113
Red2	256 threads	0.802	32	2.413
Red3	256 threads	0.799	32	2.410
Red4	512 threads	1.073	1024	2.083
Red5	256 threads	0.706	1024	1.881
Red7	128 threads	0.722	1024	1.968

Best parameter selection

worst parameter selection

Reduce 2^24 elements (1000 times)

Performance is most satisfactory

Need to do more benchmarks (including scan!)

The degree of control for the user finally feels at the proper control freak level

Still need to think more about the API

See Ulvinge's thesis for an interesting development

We need to do a lot of benchmarking to turn this into science

…rameter selection

worst parameter selection

We would be happy if any of you wanted to work on using or developing Obsidian ☺

CUDA programming is fun, but Obsidian programming is even more fun!