Programming Accelerators

Ack: Obsidian is developed by Joel Svensson
thanks to him for the black slides and ideas

github.com/svenssonjoel/Obsidian
for latest version of Obsidian

Developments in computer architecture place demands on programmers!

3 main challenges

Power wall

ILP wall

Memory wall

Power wall

More capable processors use more power

Solution

Make processors as collections of different specialised (and gen. purpose) compute capabilities. Example ARM big.LITTLE

Turn some of them off

Instuction Level Parallelism Wall

Finding ILP and increasing frequency out of steam

Solution

More but simpler cores
Accelerators

Memory Wall

Processor performance and memory performance diverging

Solution

Larger caches
More complicated memory hierarchies
Programmer controlled scratchpad memories

Memory Wall

Processo
performa

Note Chalmers expertise here!
McKee Stenström

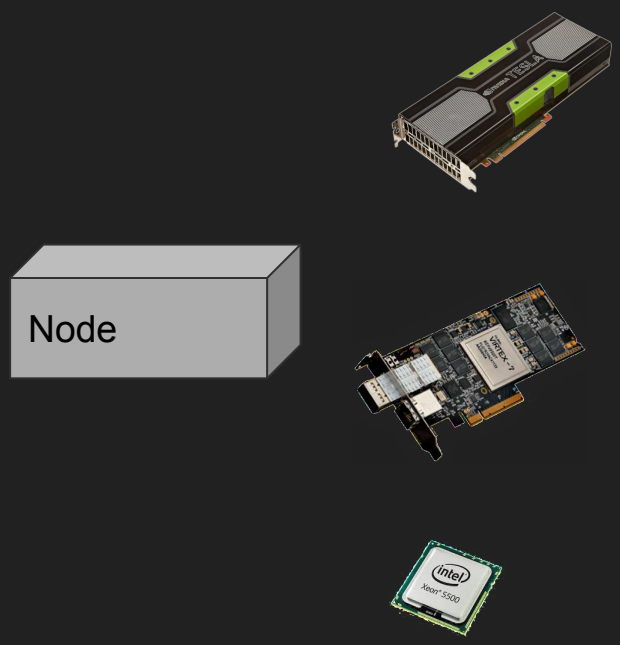
Solution

Larger caches
More complicated memory hierarchies
Programmer controlled scratchpad memories

Summary

Programming gets harder

Heterogeneity is everywhere!



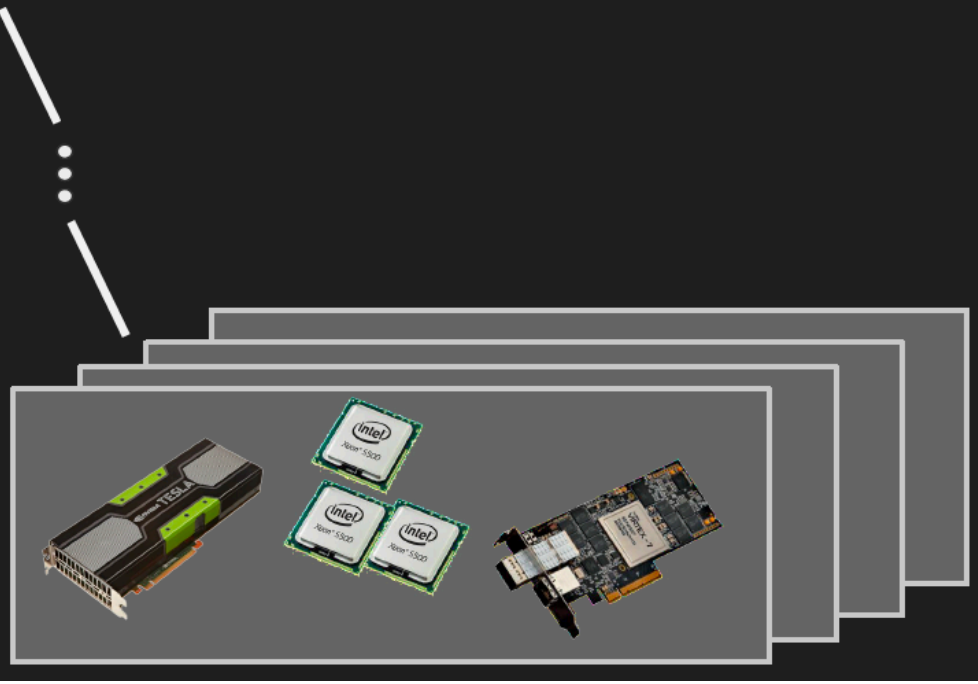
An HPC node today:

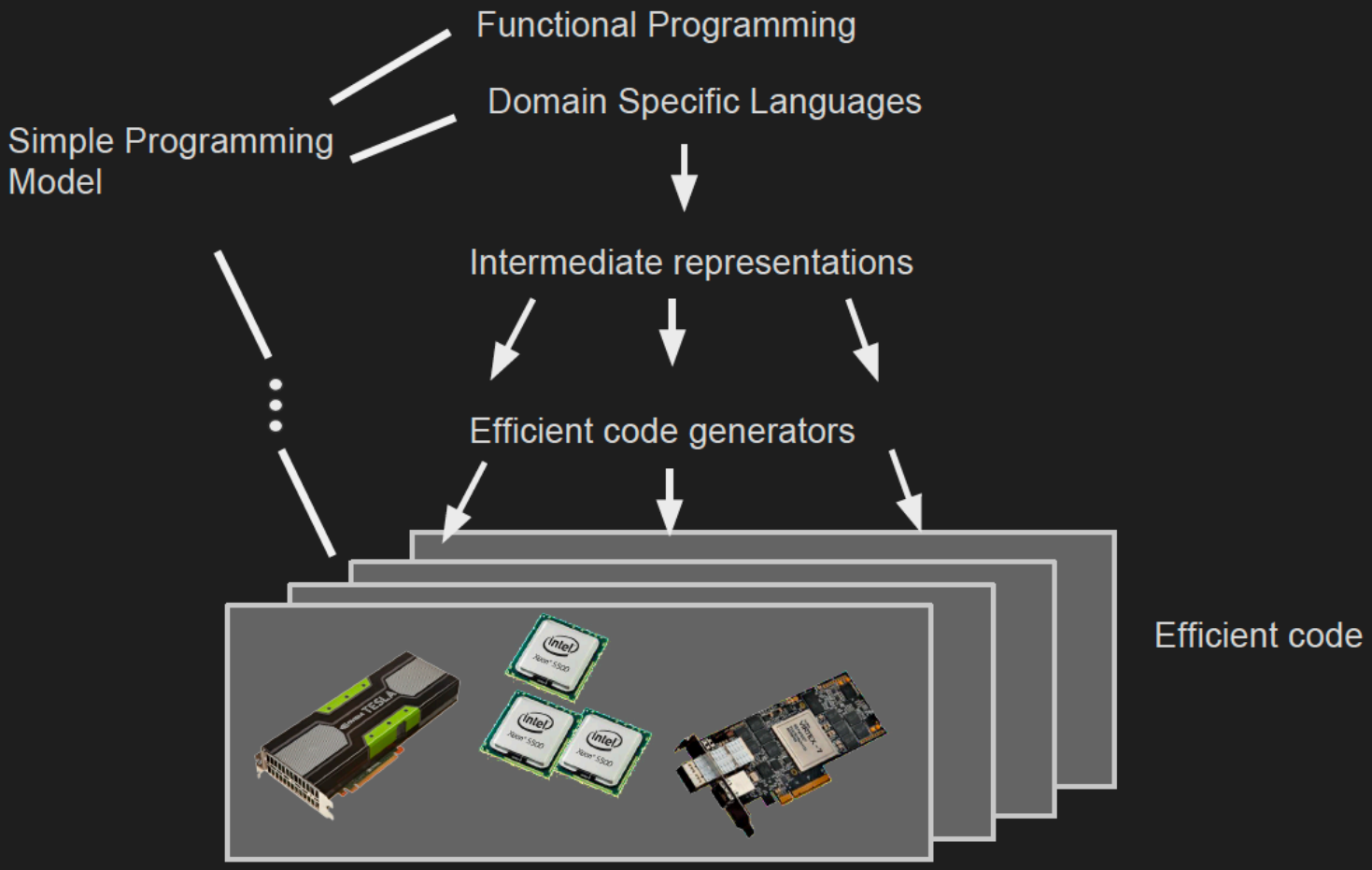
- Processors (traditional CPUs)
- GPUs
- And/Or Xeon PHI

Upcoming:

- Field Programmable Gate Arrays
  - Xilinx Zynq Ultrascale+
  - Xeon + FPGA

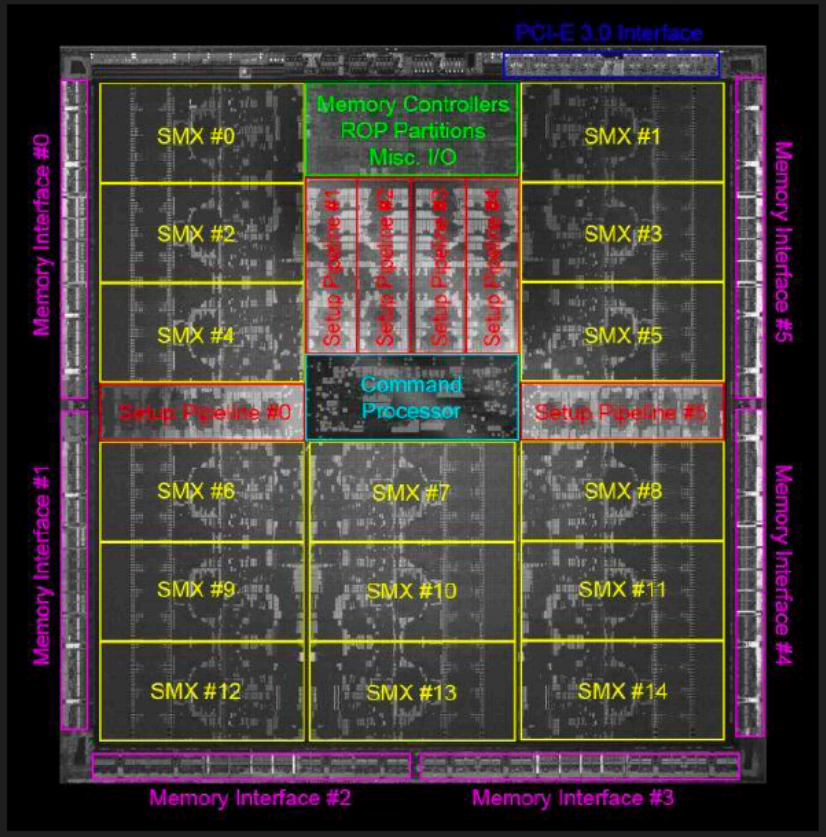
Simple Programming Model



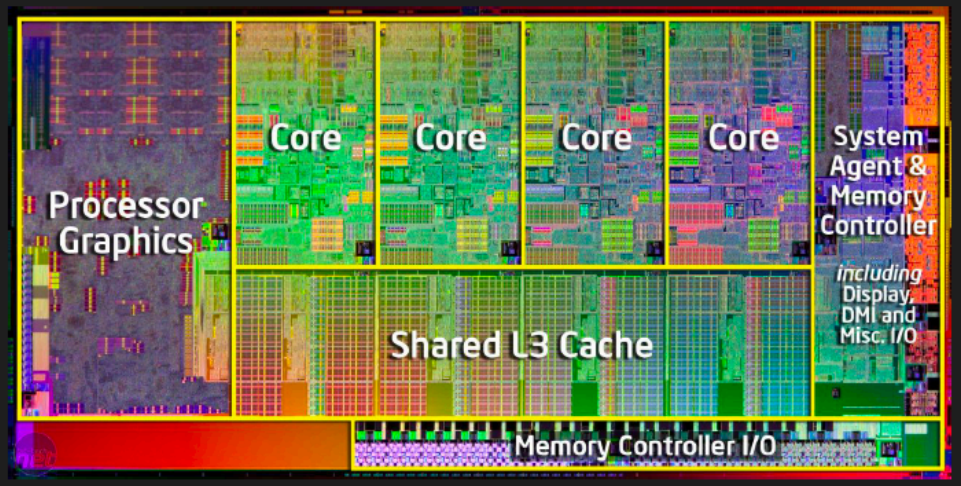

Functional Programming
Domain Specific Languages
Simple Programming Model
Intermediate representations
Efficient code generators
Efficient code

“X on the GPU”

- 10x - 100x speedup.
- Very complicated.

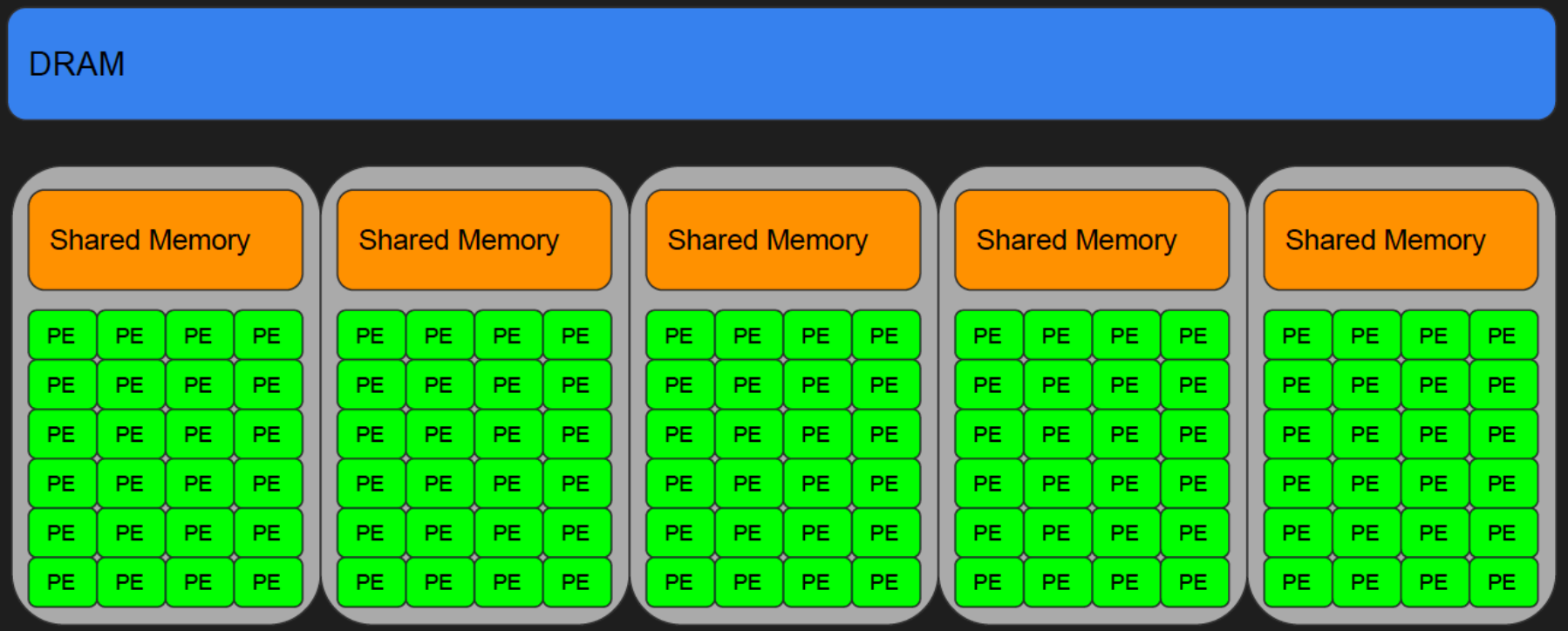


NVIDIA Kepler (GK110)

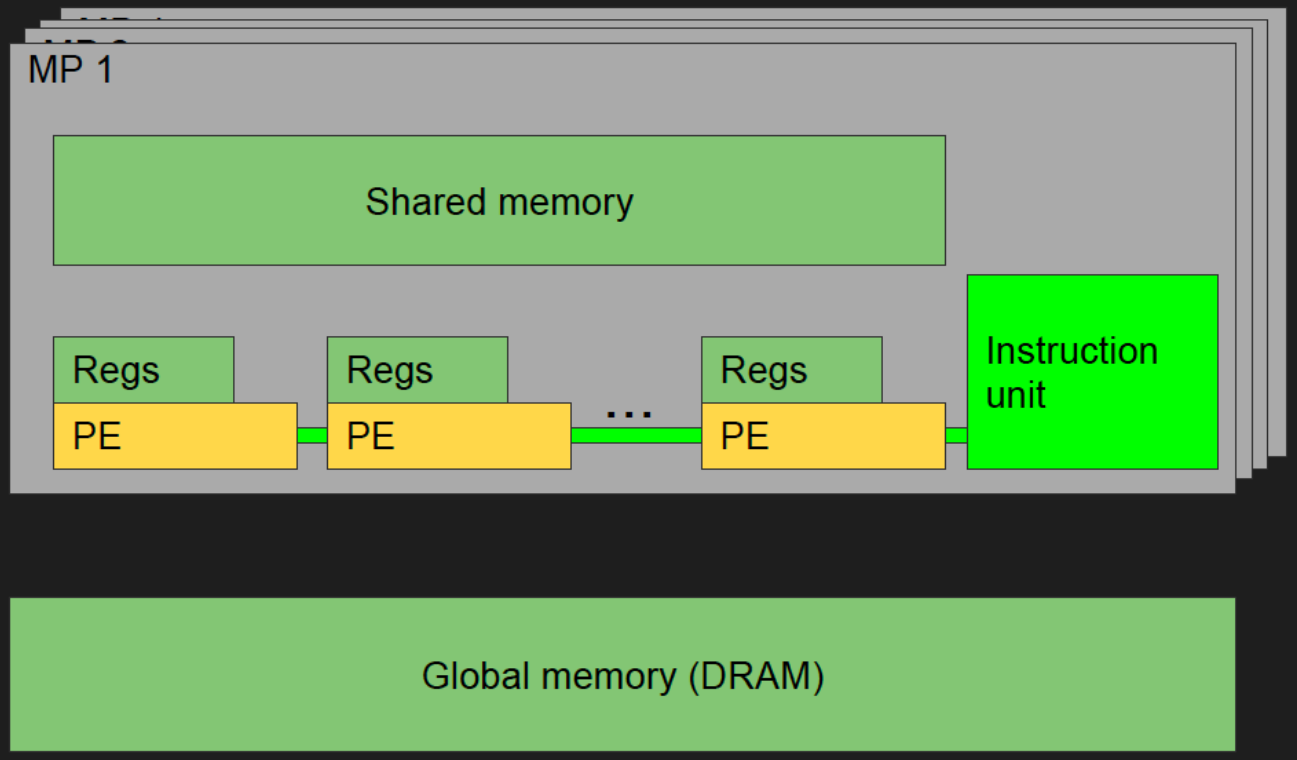


Sandy Bridge

GPU: The Architecture



GPU: Zoom in on an MP



CUDA programming model

Single Program Multiple Threads

Kernel = Function run N times by N threads

Hierarchical thread groups

Associated memory hierarchy

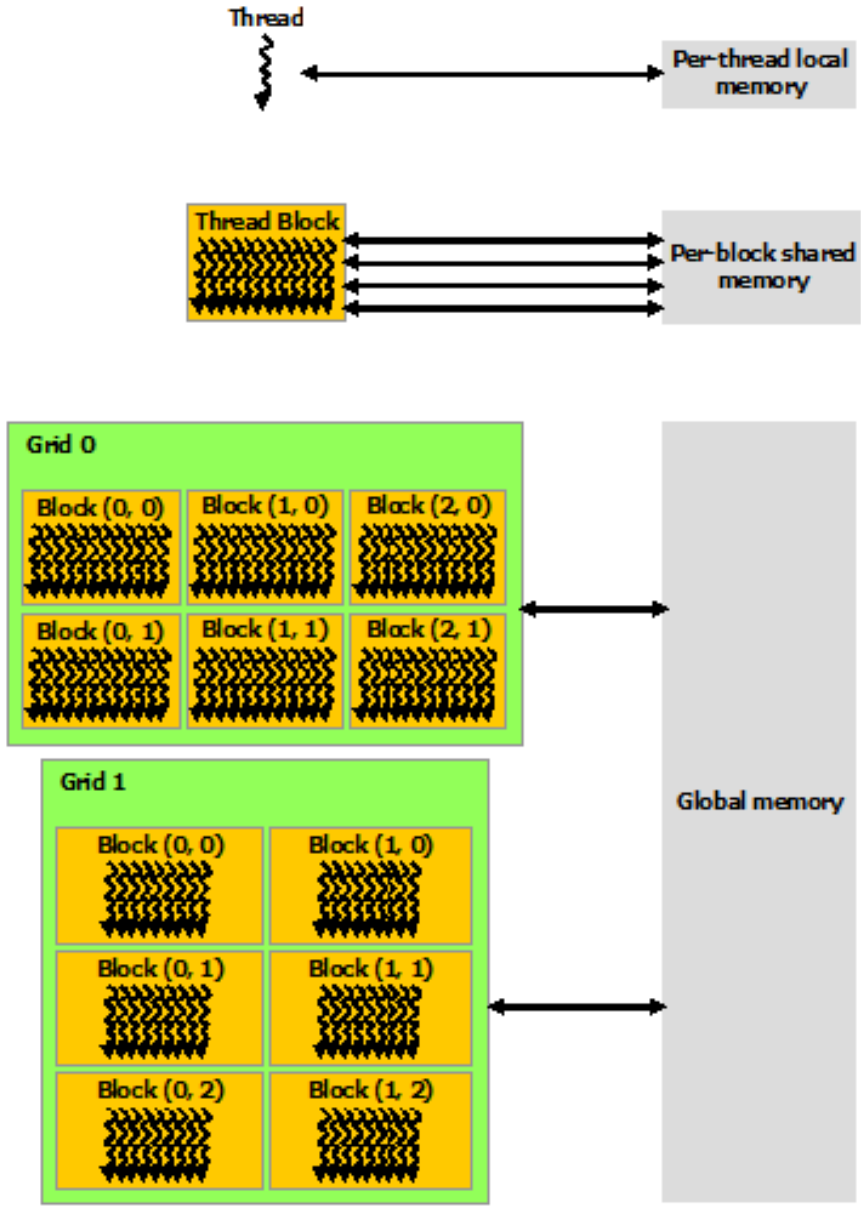


Image from http://docs.nvidia.com/cuda/cuda-c-programming-guide/#memory-hierarchy

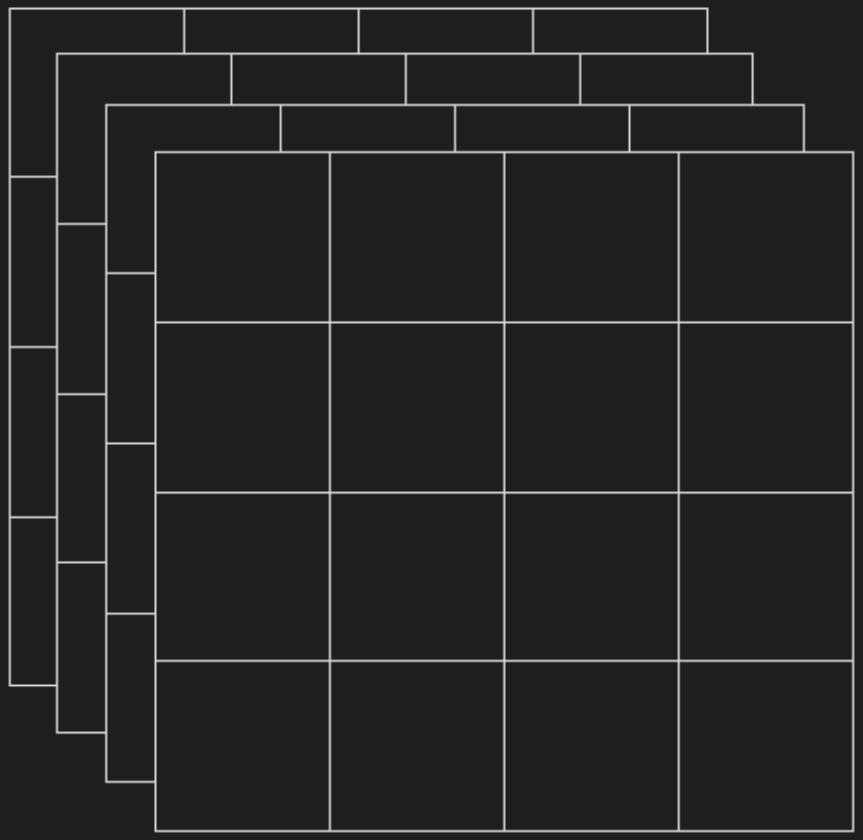
CUDA: Concepts

- Threads
  - Executes in the PEs.
- Warps
  - 32 threads, one PC.
- Blocks
  - Group of threads that cooperates.
  - Can synchronize.
  - Shared Memory.
- The Grid
  - The collection of blocks.

CUDA: More details

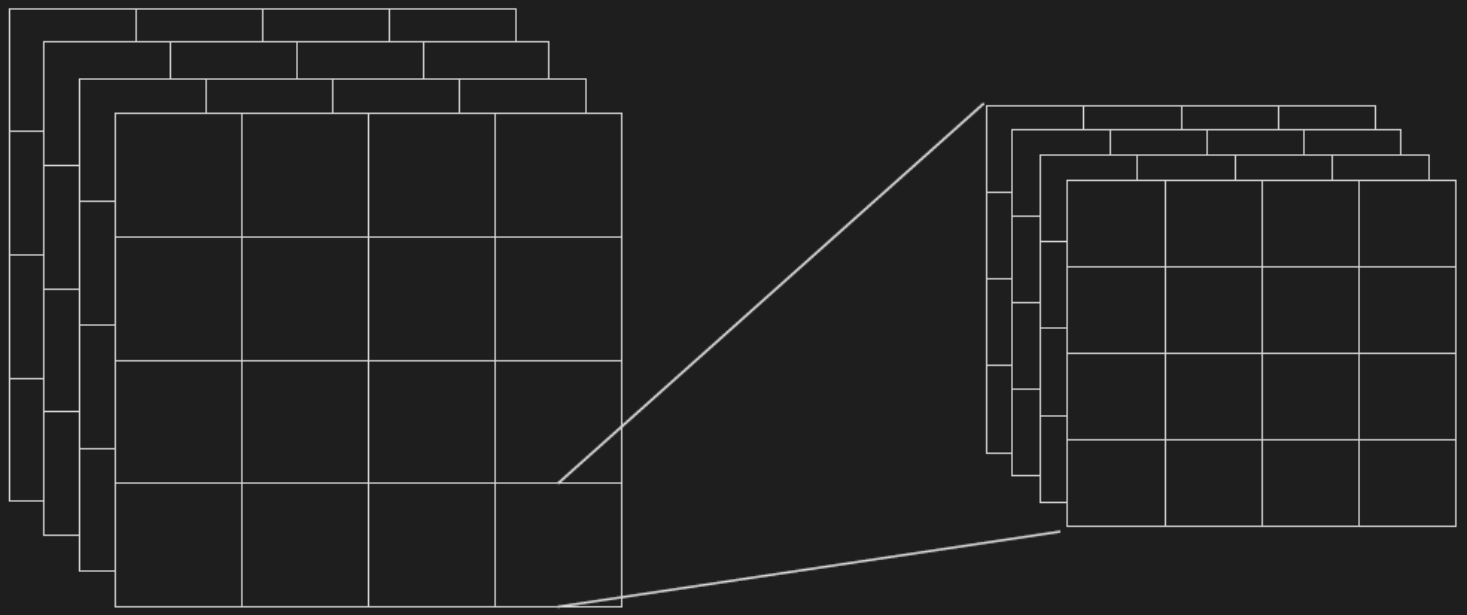
- Threads
  - All threads are described by a single program (SIMT).
- Blocks
  - Up to 1024 cooperating threads per block.
  - Many blocks share an MP.
  - More Threads per block than processors per MP. (syncthreads)
  - 1,2 or 3d shaped iteration space.
- The Grid
  - Work is launched onto the GPU in a unit called a grid.
  - 1,2 or 3d grid of blocks.

Grid of Blocks of Threads



dim3 grid_dim(4,4,4);

Grid of Blocks of Threads



dim3 grid_dim(4,4,4);

dim3 block_dim(4,4,4);

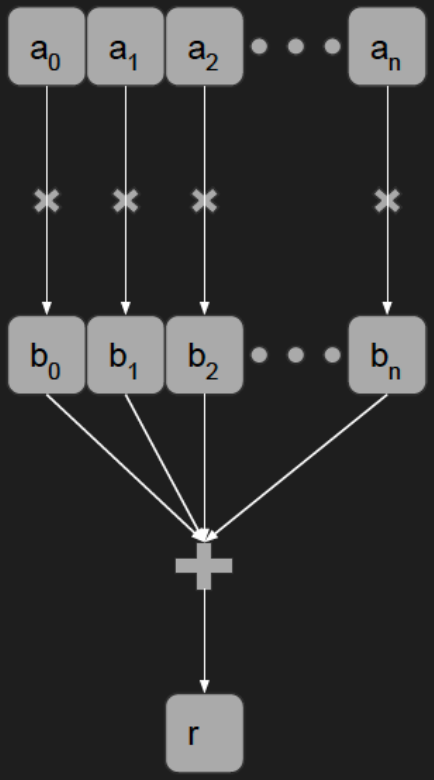
Launching a Grid

```
kernel<<<grid_dim,block_dim>>>(arg1,...,argn);
```

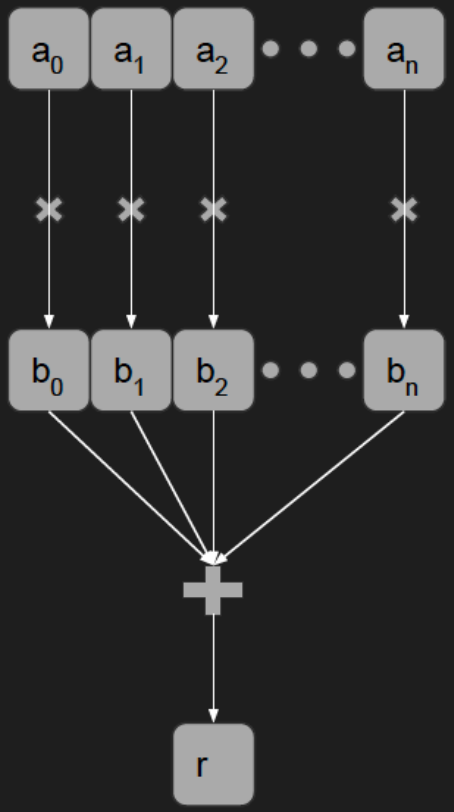
The Kernel Code

- One code executed by ALL threads of the grid.
  - Identifies its position in the grid/block using:
    - blockIdx.x, blockIdx.y, blockIdx.z.
    - threadIdx.x, threadIdx.y, threadIdx.z.
  - Can query the dimensions of the grid/block using:
    - gridDim.x, gridDim.y, gridDim.z.
    - blockDim.x, blockDim.y, blockDim.z.
  -

CUDA: An Example Kernel



CUDA: An Example Kernel




```
__global__ void dot( int* a, int* b, int* c ) {
 __shared__ int tmp[THREADS_PER_BLOCK];

 int gid = threadIdx.x +
           blockIdx.x *
           blockDim.x;

 tmp[threadIdx.x] = a[gid] * b[gid];

 __syncthreads();

 /* REDUCE */
 if (threadIdx.x == 0) {
      int sum = 0;
      for (int i = 0; i < THREADS_PER_BLOCK; ++i)
        sum += tmp[i];
      atomicAdd(c, sum);
 }
}
```

CUDA: A Launch Example

```
dot<<<1000,1000>>>(a,b,result);
```

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)

Hierarchy

Level	Parallelism	Shared Memory	Thread synchronisation
Thread	No	Yes	No
Warp	Yes	Yes	Lock-step execution
Block	Yes	Yes	Yes
Grid	Yes	No	No

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

another 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);
}
```

Haskell EDSLs for GPU programming

Accelerate

Obsidian

Accelerate

Get acceleration from your GPU by writing familiar combinators

Hand tuned skeleton templates

Compiler cleverness to fuse and memoise the resulting kernels

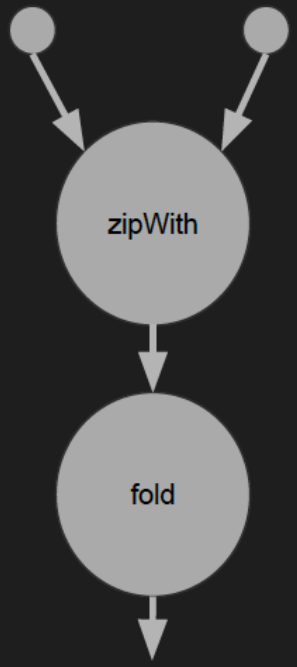
Leaves a gap between the programmer and the GPU (which most people want)

See Chap. 6 of PCPH for description of accelerate-cuda
However, it will be replaced by accelerate-llvm-ptx

Accelerate: An Example

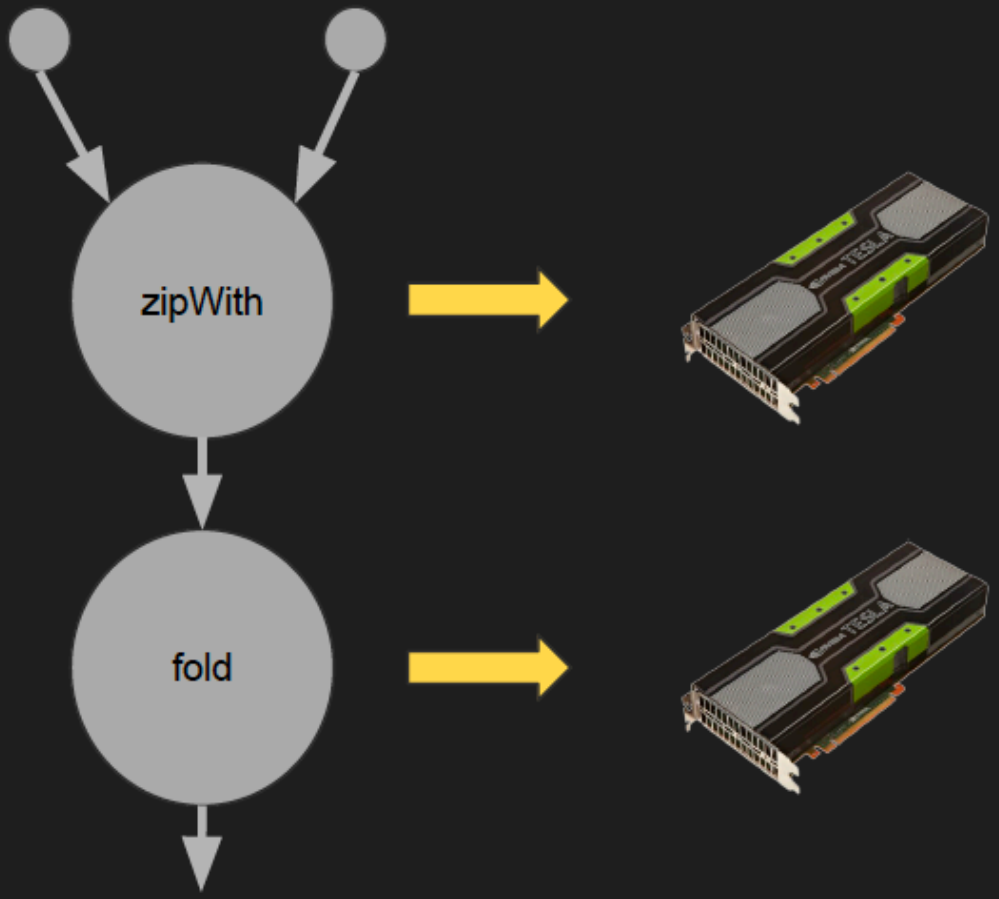
```
dotp :: Num n => Vector n -> Vector n -> Acc (Scalar n)
dotp xs ys = let xs' = use xs
                 ys' = use ys
             in  fold (+) 0 (zipWith (*) xs' ys')
```

Accelerate: An Example

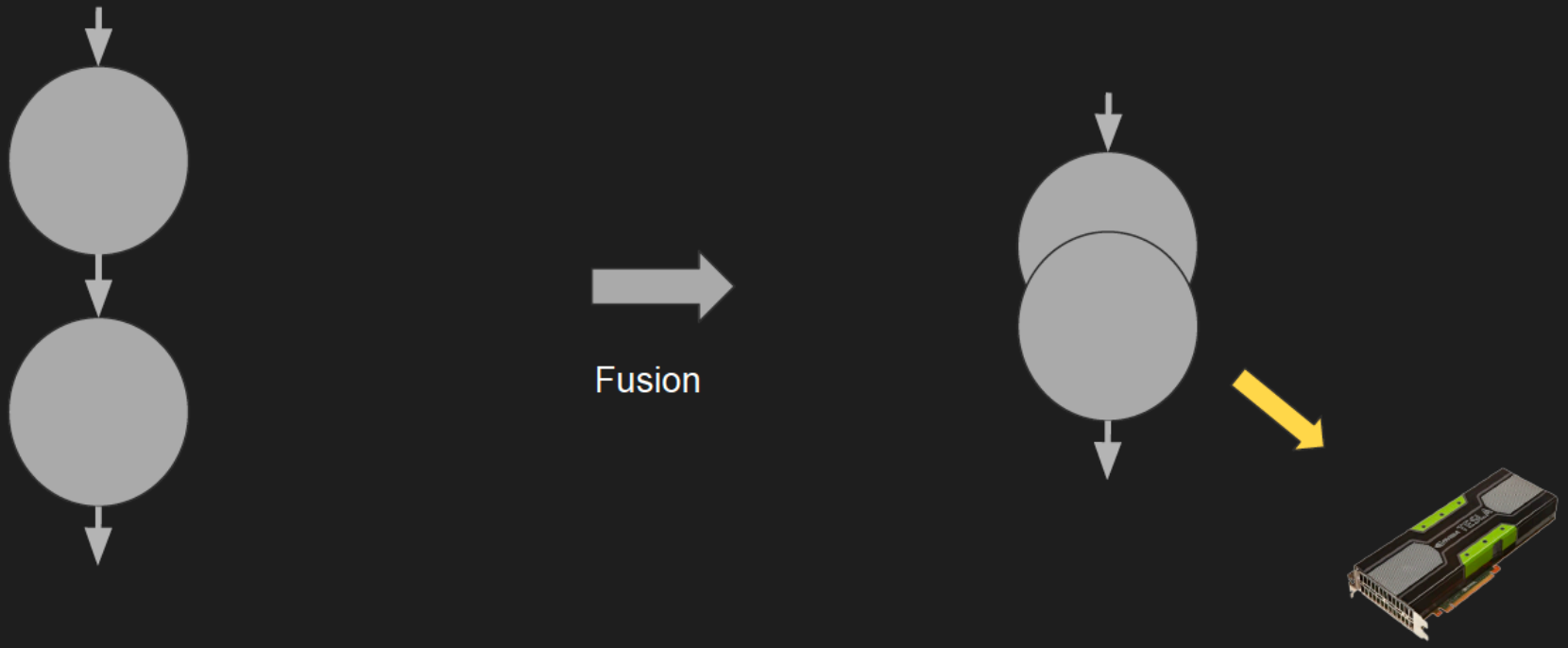



```
dotp :: Num n => Vector n -> Vector n -> Acc (Scalar n)
dotp xs ys = let xs' = use xs
                 ys' = use ys
             in  fold (+) 0 (zipWith (*) xs' ys')
```

Accelerate: An Example

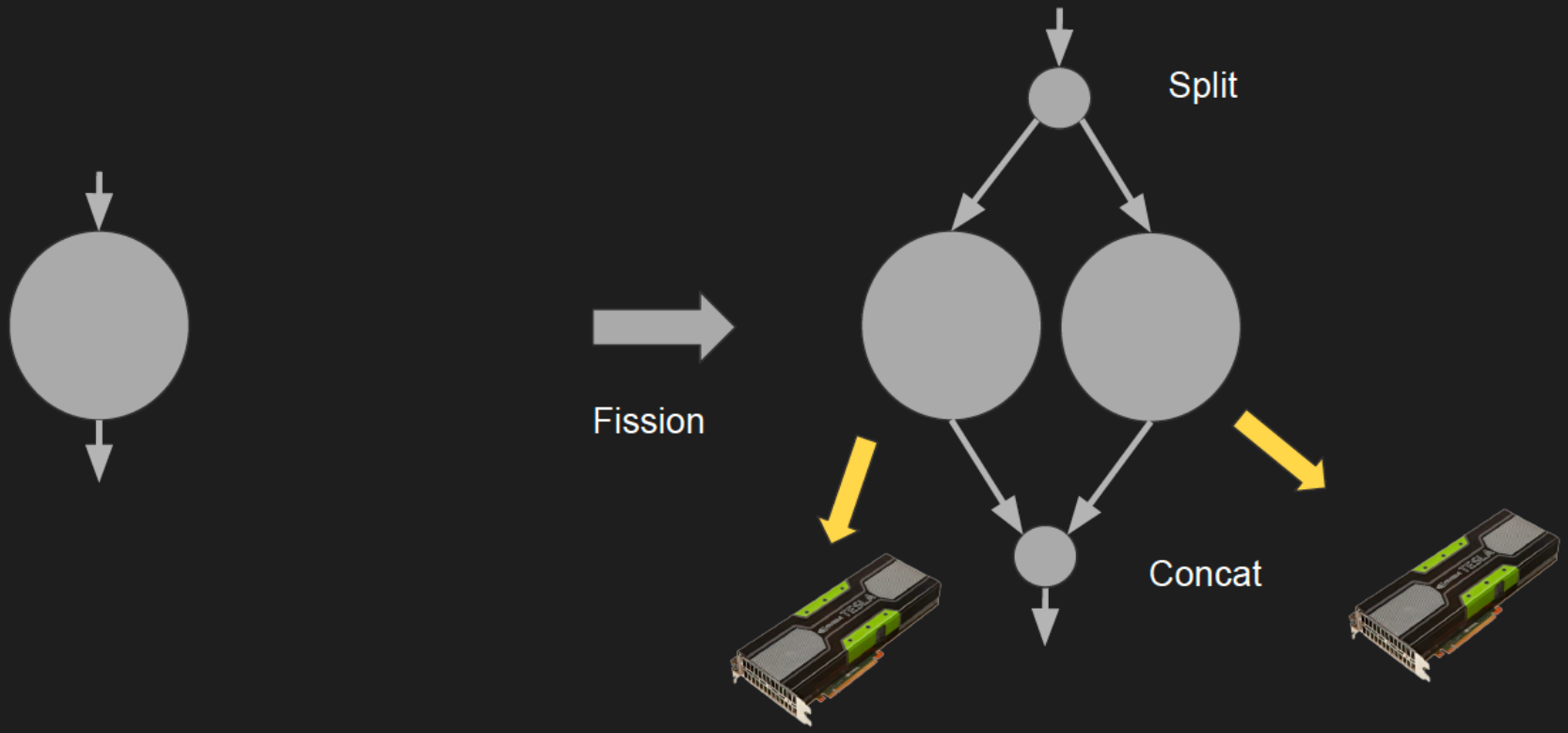


Accelerate: High Level Optimisations



Optimising Purely Functional GPU Programs
Trevor L. McDonell, Manuel M. T. Chakravarty, Gabriele Keller, and Ben Lippmeier.
ICFP'13 paper slides

Accelerate: High Level Optimisations



Converting Data-Parallelism to Task-Parallelism by Rewrites

Bo Joel Svensson, Michael Vollmer, Eric Holk, Trevor L. McDonell, and Ryan R. Newton

Workshop on Functional High Performance Computing (FHPC'15)

paper

Accelerate Operations With Types

```
use :: Array sh e -> Acc (Array sh e)

map :: (Exp a -> Exp b) -> Acc (Array sh a) -> Acc (Array sh b)

zipWith :: (Exp a -> Exp b -> Exp c) -> Acc (Array sh a) -> Acc (Array sh b) -> Acc (Array sh c)

fold :: (Exp a -> Exp a -> Exp a) -> Exp a -> Acc (Array sh :. Int a) -> Acc (Array sh a)
```

Note Array shapes just like in Repa

See many interesting papers, installation instructions etc on the Accelerate home page

Obsidian

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

Obsidian

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

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

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

Memory layout is taken care of (statically)

Building blocks

Embedded DSL in Haskell

Pull and push arrays

Use
functi

A form
like ma

Delayed arrays
See Pan by Elliot http://conal.net/pan/
Or even Compilation and Delayed Evaluation in APL, Guibas and Wyatt, POPL'78

Building blocks

Embedded DSL in Haskell

Pull and push arrays

Use of types
functions (Tl

A form of vir
like max #th

A new array representation due to Claessen
will come back to this

Obsidian

- High/Low-level programming.
- Mimics the GPU hierarchy.
- Generate GPU kernels.
- Easily generate code variants.
- Expose parameters for auto-tuning.
  - Always a parameter: Number of “real” threads, Number of “real” blocks.

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

```
increment :: DPull (SPull EWord32) -> DPush Grid EWord32
increment arr = asGridMap body arr
  where body a = push (incLocal a)
```

```
performInc :: IO ()
performInc
  =  withCUDA $
     do
       kern <- capture 64 (increment . splitUp 256)

       useVector (V.fromList [0..1023]) $ \i ->
         withVector 1024  $ \o ->
         do
           fill o 0
           o <== (1,kern) <> i

           r <- copyOut o
           lift $ putStrLn $ show r

main :: IO ()
main = performInc
```

```
performInc :: IO ()
performInc
  =  withCUDA $
     do
       kern <- capture 64 (increment . splitUp 256)

       useVector (V.fromList [0..1023]) $ \i ->
         withVector 1024  $ \o ->
         do
           fill o 0
           o <== (1,kern) <> i

           r <- copyOut o
           lift $ putStrLn $ show r

main :: IO ()
main = performInc
```

Threads per block

Array elements per block

```
#include <stdint.h>

extern "C"  __global__ void gen0(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 < 4; ++i) {
             tid = i * 64 + 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 < 4; ++i) {
            tid = i * 64 + threadIdx.x;
            output1[bid * 256U + tid] = input0[bid * 256U + tid] + 1U;
         }
         tid = threadIdx.x;
      }
      bid = blockIdx.x;
      __syncthreads();
}
```

```
*Main> performInc
[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,
27,28,29,30,31,32,33,34,35,36,37,38,39,40,41,42,43,44,45,46,47,48,49
,
....

,913,914,915,916,917,918,919,920,921,922,923,924,925,926,927,928,
929,930,931,932,933,934,935,936,937,938,939,940,941,942,943,944,9
45,946,947,948,949,950,951,952,953,954,955,956,957,958,959,960,96
1,962,963,964,965,966,967,968,969,970,971,972,973,974,975,976,977
,978,979,980,981,982,983,984,985,986,987,988,989,990,991,992,993,
994,995,996,997,998,999,1000,1001,1002,1003,1004,1005,1006,1007,
1008,1009,1010,1011,1012,1013,1014,1015,1016,1017,1018,1019,102
0,1021,1022,1023,1024]
```

Obsidian Pull arrays

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

```
type SPull = Pull Word32
```

Static size Word32 = Haskell value known at compile time

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

```
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

```
arr = Pull n ixf

map f (map g arr) = map f (Pull n (g . ixf)

                  = Pull n (f . g . 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

```
data Thread
data Step t

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

GPU Hierarchy in types

```
-- | Type level less-than-or-equal test.
type family LessThanOrEqual a b where
   LessThanOrEqual Thread   Thread   = True
   LessThanOrEqual Thread   (Step m) = True
   LessThanOrEqual (Step n) (Step m) = LessThanOrEqual n m
   LessThanOrEqual x y               = False


type a *<=* b = (LessThanOrEqual a b ~ True)
```

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
  -- Thread, Warp, Block.
ForAll :: (t *<=* Block) => EWord32
            -> (EWord32 -> Program Thread ())
            -> Program t ()

. . .
```

Program data type

```
seqFor :: EWord32 -> (EWord32 ! Program t ()) -> Program t ()

. . .

Sync     :: (t *<=* Block) => Program t ()

. . .
```

Program data type

```
. . .


Return :: a -> Program t a
Bind   :: Program t a -> (a -> Program t b) -> Program t b
```

```
instance Monad (Program t) where
  return = Return
  (>>=) = Bind
```

See

Svenningsson, Josef, & Svensson, Bo Joel. (2013). Simple and Compositional Reification of Monadic Embedded Languages. ICFP 2013.

Obsidian push arrays

```
data Push t s a = Push s (PushFun t a)
```

The general idea of push arrays is due to Koen Claessen

Obsidian push arrays

```
-- | Push array. Parameterised over Program type and size type.

data Push t s a = Push s (PushFun t a)

type PushFun t a = Writer a -> Program t ()
```

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

The general idea of push arrays is due to Koen Claessen

Obsidian push arrays

```
-- | Push array. Parameterised over Program type and size type.

data Push t s a = Push s (PushFun t a)

type PushFun t a = Writer a -> Program t ()
type Writer    a = a -> EWord32 -> TProgram ()
```

consumer of a push array needs to apply the push-function to a suitable writer

Often the push-function is applied to a writer that stores its input value at the provided input index into memory. This is what the compute function does when applied to a push array.

The general idea of push arrays is due to Koen Claessen

Obsidian push arrays

The function push converts a pull array to a push array:

```
push ::  (t *<=* Block) => ASize s => Pull s e -> Push t s e
push (Pull n ixf) =
  mkPush n $ \wf ->
    forAll (sizeConv n) $ \i -> wf (ixf i) i
```

Obsidian push arrays

The function push converts a pull array to a push array:

```
push ::  (t *<=* Block) => ASize s => Pull s e -> Push t s e
push (Pull n ixf) =
  mkPush n $ \wf ->
    forAll (sizeConv n) $ \i -> wf (ixf i) i
```

This function sets up an iteration schema over the elements as a forAll loop. It is not until the t parameter is fixed in the hierarchy that it is decided exactly how that loop is to be executed. All iterations of the forAll loop are independent, so it is open for computation in series or in parallel.

```
forAll :: (t *<=* Block) => EWord32
          -> (EWord32 -> Program Thread ())
          -> Program t ()
forAll n f = ForAll n f
```

ForAll iterates a body (described by higher order abstract syntax) a given number of times over the resources at level t

iterations independent of each other

t = Thread => sequential

t = Warp, Block => parallel

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 writer function

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

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

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

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

Push and pull arrays

Neither pull nor push arrays are manifest

Both fuse by default.

Both immutable.

Don’t appear in Expression or Program datatypes
Shallow Embedding

See Svenningsson and Axelsson on combining deep and shallow embeddings

Why two kinds of arrays ?

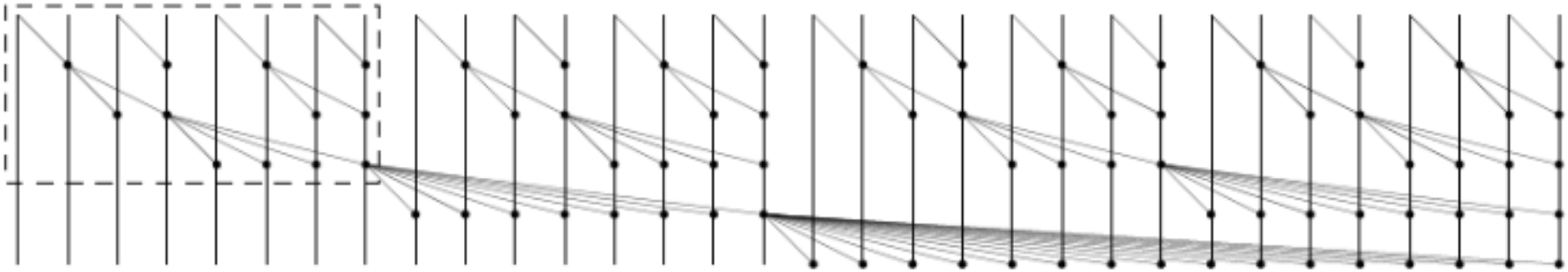
Pull Arrays:

- Efficient indexing.
- No efficient concatenation
- Consumer decides iteration pattern

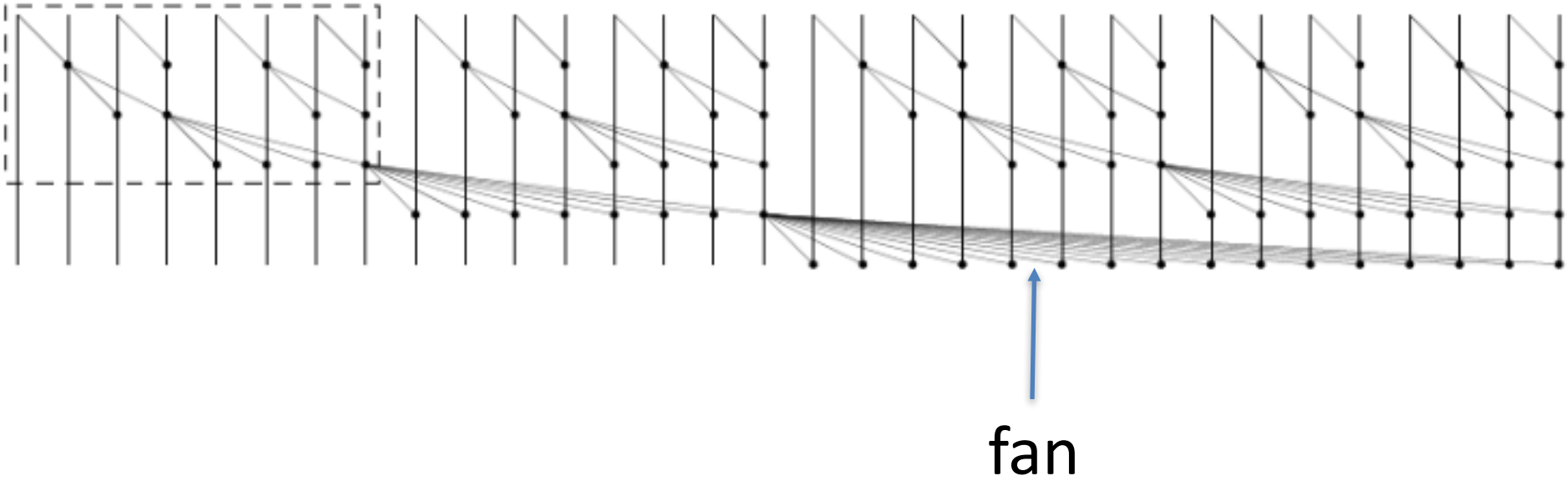
Push Arrays:

- Efficient concatenation.
- No efficient indexing.
- Producer decides iteration pattern.

Another scan (Sklansky 60)



Another scan (Sklansky 60)



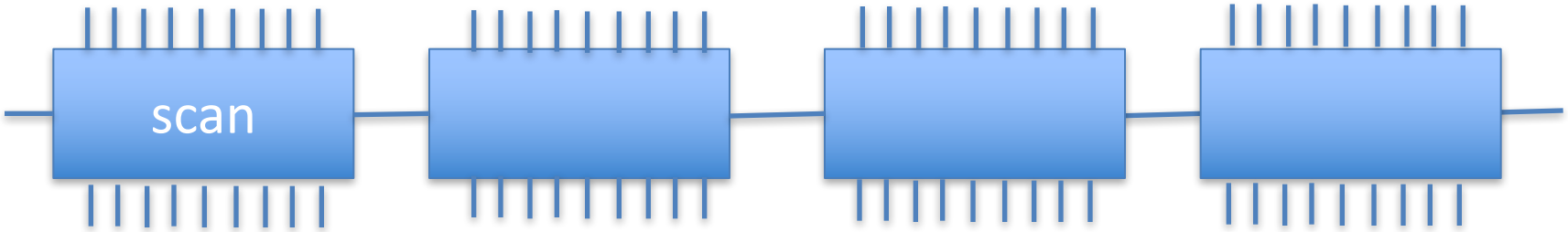
Block scan

```
fan :: (ASize s, Choice a)
    => (a -> a -> a)
    -> Pull s a
    -> Pull s a
fan op arr =  a1 `append`  fmap (op c) a2
  where (a1,a2) = halve arr
        c = a1 ! (fromIntegral (len a1 - 1))
```

Block scan

```
sklanskyLocalPull :: Data a
                  => Int
                  -> (a -> a -> a)
                  -> SPull a
                  -> BProgram (SPull a)
sklanskyLocalPull 0 _  arr = return arr
sklanskyLocalPull n op arr =
  do
    let arr1 = unsafeBinSplit (n-1) (fan op) arr
    arr2 <- compute $ push arr1
    sklanskyLocalPull (n-1) op arr2
```

hybrid



Block scan

```
sklanskyLocalCin :: Data a
                 => Int
                 -> (a -> a -> a)
                 -> a -- cin
                 -> SPull a
                 -> BProgram (a, SPush Block a)
sklanskyLocalCin n op cin arr = do
  arr' <- compute (applyToHead op cin arr)
  arr'' <- sklanskyLocalPull n op arr'
  return (arr'' ! (fromIntegral (len arr'' - 1)), push arr'')
    where
      applyToHead op cin arr =
        let h = fmap (op cin ) $ take 1 arr
            b = drop 1 arr
        in h `append` b
```

```
sklanskies n op acc arr
  = sMapAccum (sklanskyLocalCin n op) acc (splitUp 512 arr)



sklanskies' :: (Num a, Data a )
            => Int
            -> (a -> a -> a)
            -> a
            -> DPull (SPull a)
            -> DPush Grid a

sklanskies' n op acc = asGridMap (sklanskies n op acc)
```

```
perform
  =  withCUDA $
     do
       kern <- capture 512(sklanskies' 9 (+) 0 . splitUp 1024)
       useVector (V.fromList [0..1023 :: Word32]) $ \i ->
         withVector 1024 $ \ (o :: CUDAVector Word32) ->
       do
         fill o 0
         o <== (1,kern) <> i
         r <- peekCUDAVector o
         lift $ putStrLn $ show
```

```
*Main>
perform[0,1,3,6,10,15,21,28,36,45,55,66,78,91,105,120,136,153,171,190,210,231,25
3,276,300,325,351,378,406,435,465,496,528,561,595,630,666,703,741,780,820,861,90
3,946,990,1035,1081,1128,1176,1225,1275,1326,1378,1431,1485,1540,1596,1653,1711,
1770,1830,1891,1953,2016,2080,2145,2211,2278,2346,2415,2485,2556,2628,2701,2775,
2850,2926,3003,3081,3160,3240,3321,3403,3486,3570,3655,3741,3828,3916,4005,4095,
4186,4278,4371,4465,4560,4656,4753,4851,4950,5050,5151,5253,5356,5460,5565,5671,
5778,5886,5995,6105,6216,6328,6441,6555,6670,6786,6903,7021,7140,7260,7381,7503,
7626,7750,7875,8001,8128,8256,8385,8515,8646,8778,8911,9045,9180,9316,9453,9591,
9730,9870,10011,10153,10296,10440,10585,10731,10878,11026,11175,11325,11476,1162
8,11781,11935,12090,12246,12403,12561,12720,12880,13041,13203,13366,13530,13695,
...
432915,433846,434778,435711,436645,437580,438516,439453,440391,441330,442270,443
211,444153,445096,446040,446985,447931,448878,449826,450775,451725,452676,453628
,454581,455535,456490,457446,458403,459361,460320,461280,462241,463203,464166,46
5130,466095,467061,468028,468996,469965,470935,471906,472878,473851,474825,47580
0,476776,477753,478731,479710,480690,481671,482653,483636,484620,485605,486591,4
87578,488566,489555,490545,491536,492528,493521,494515,495510,496506,497503,4985
01,499500,500500,501501,502503,503506,504510,505515,506521,507528,508536,509545,
510555,511566,512578,513591,514605,515620,516636,517653,518671,519690,520710,521
731,522753,523776]
```

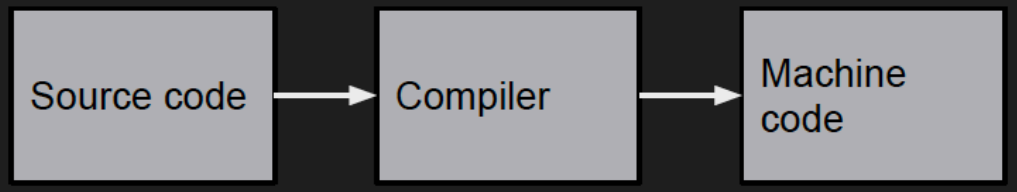
User experience

A lot of index manipulation tedium is relieved

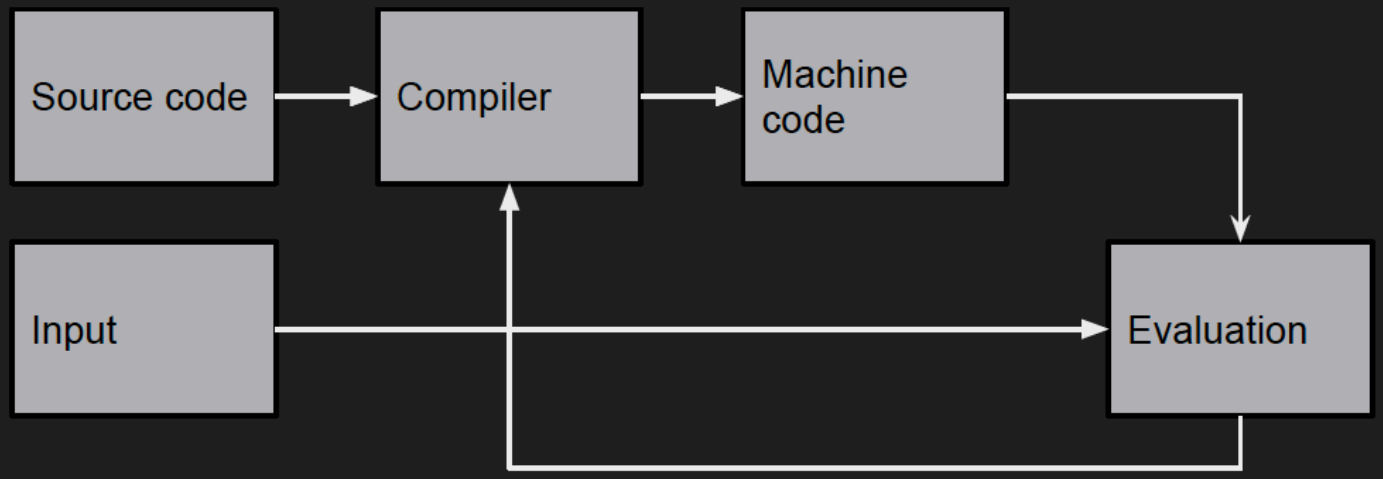
Program composition and reuse greatly eased

Autotuning springs to mind!!

Compilation



Compilation: Tuning framework



Tuning Framework

```
class TuneM m where
  -- | Get parameter by index.
  getParam :: ParamIdx -> m Int

type ParamIdx = Int

scoreIt :: (MonadIO m, TuneM m)
        => m (Maybe Result)
scoreIt = do
  threads <- getParam 0
  blocks  <- getParam 1
  liftIO $ catch (
    do time <- timeIt threads blocks
       return $ Just
              $ Result ([threads,blocks],time)
    )
    (\e -> do putStrLn (show (e :: SomeException))
              return Nothing
    )
```

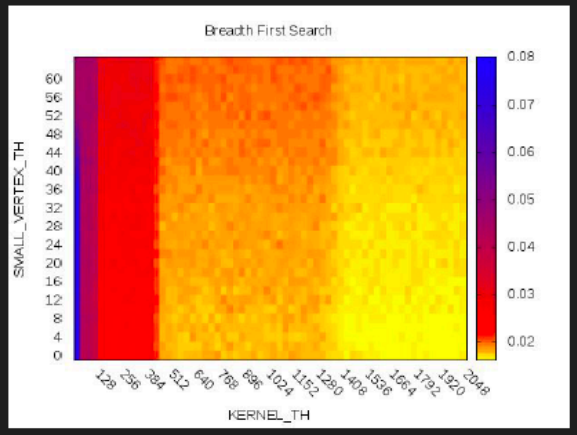
Meta-Programming and Auto-Tuning in the Search for High Performance GPU Code

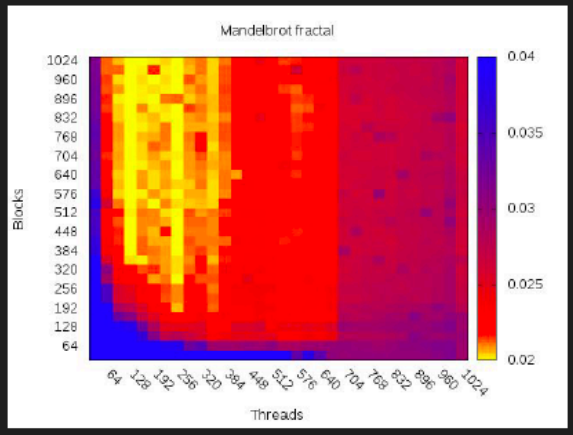
Michael Vollmer, Bo Joel Svensson, Eric Holk, Ryan Newton

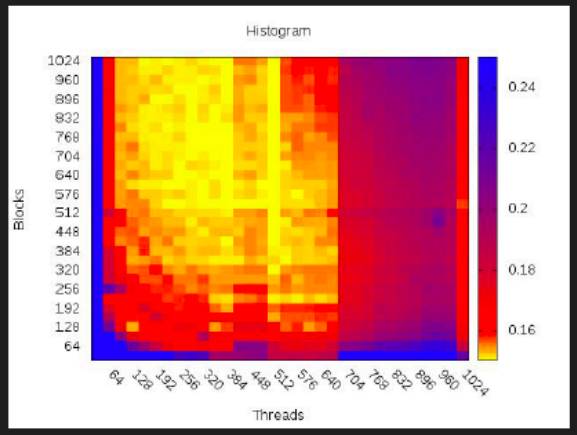
FHPC'15 video paper

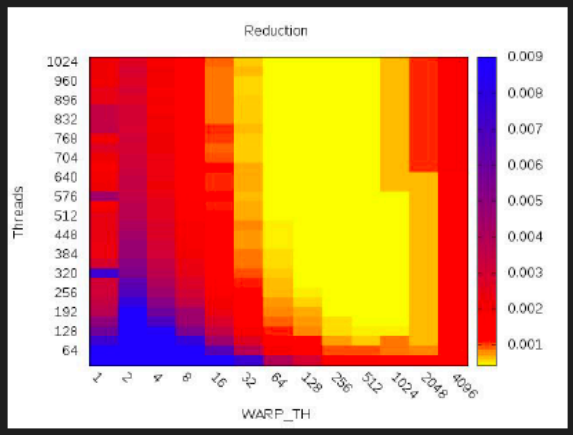
Obsidian: Tuning

- Auto-tuning
- Specialised code variants





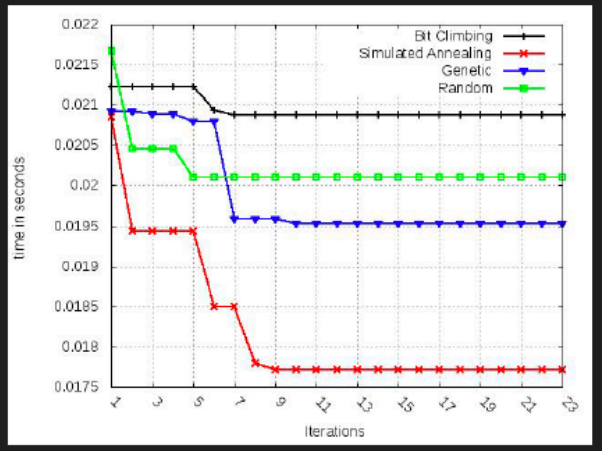


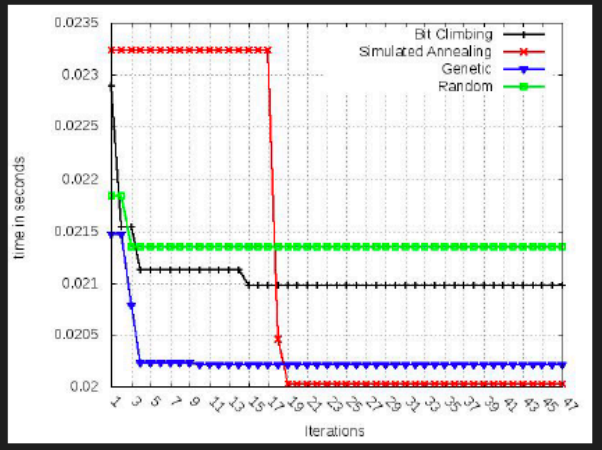


Obsidian: Tuning

- Exhaustive
- Random
- Simulated annealing
- Hill climbing

Mandelbrot

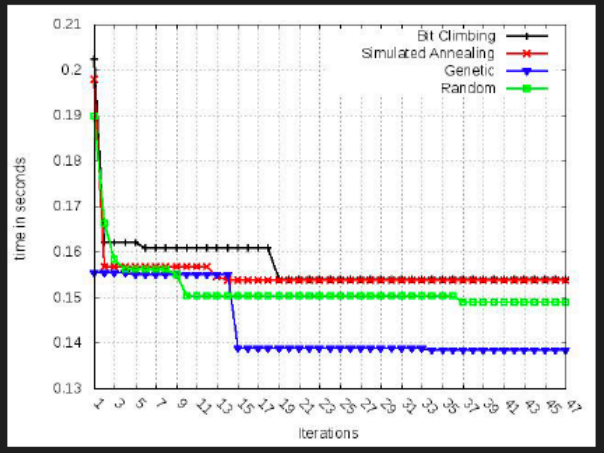


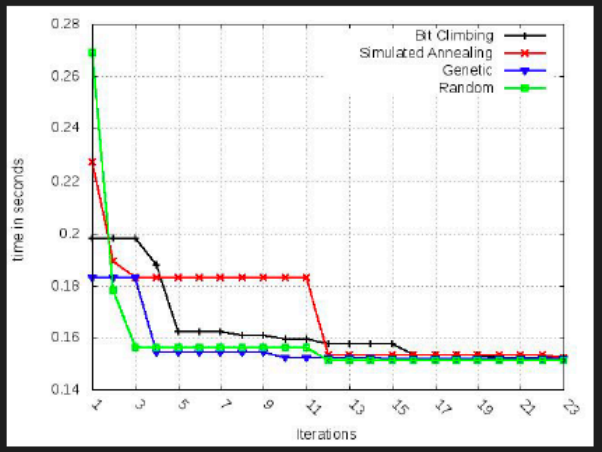


Obsidian: Tuning

- Exhaustive
- Random
- Simulated annealing
- Hill climbing

Histogram





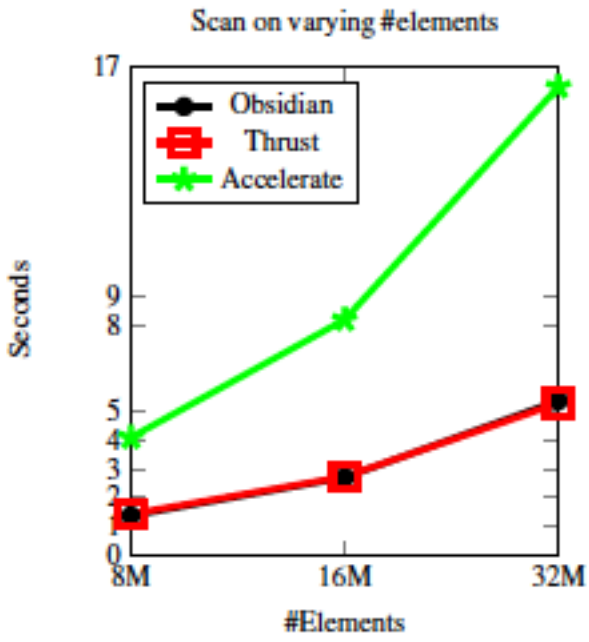


Fig. 18. The running time of scan algorithms for larger data sizes. The time reported is the sum of 1000 executions, excluding data transfer to and from the GPU memory. These number are collected on an NVIDIA GTX680. The presented Accelerate numbers are estimates based on a lower number of iterations as explained in Section 7.3.

Compilation to CUDA (overview)

1. Reification Produce a Program AST
2. Convert Program level datatype to list of statements
3. Liveness analysis for arrays in memory
4. Memory mapping
5. CUDA code generation (including virtualisation of threads, warps and blocks)

Compilation to CUDA (overview)

1 Reification [illegible]duce a Program AST

2 Convert Prog[illegible]datatype to list of statements

3 Liveness ana[illegible]

4 Memory ma[illegible]

5 CUDA code g[illegible] virtualisatio[illegible]ks)

Obsidian is quite small
Could be a good EDSL to study!!

A language for hierarchical data parallel design-space exploration on GPUs
BO JOEL SVENSSON, RYAN R. NEWTON and MARY SHEERAN [paper]
Journal of Functional Programming / Volume 26 / 2016 / e6

Summary I

Key benefit of EDSL is ease of design exploration

Performance is very satisfactory (after parameter exploration)
comparable to Thrust

“Ordinary” benefits of FP are worth a lot here
(parameterisation, reuse, higher order functions etc)

Pull and push arrays a powerful combination

In reality, probably also need mutable arrays (and vcopy from Feldspar)

Summary II

Flexibility to add sequential behaviour is vital to performance

Use of types to model the GPU hierarchy interesting!
    similar ideas could be used in other NUMA architectures

What we REALLY need is a layer above Obsidian (plus autotuning)
    see spiral.net for inspiring related work

I want a set of combinators with strong algebraic properties (e.g. for data-independent algorithms like sorting and scan).
Array combinators have not been sufficiently studied.

Need something simpler and more restrictive than push arrays

Obsidian And Accelerate Conclusions

Obsidian

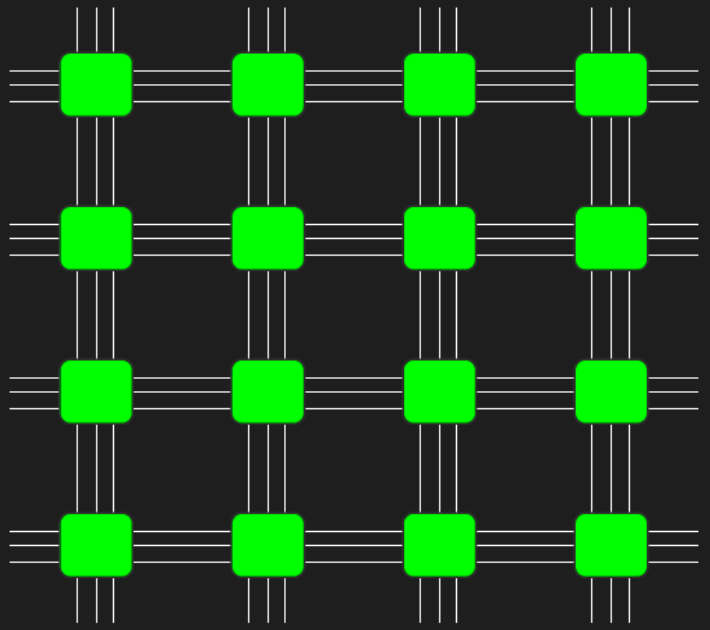
- Control
  - of what the GPU actually does.
  - of Shared Memory .
- Kernels.

Accelerate

- High level programs.
  - High level optimisations.
- Entire applications.
- Multi-device RTS.

High Performance Computing and FPGAs

- 1 or more CPUs.
- 0,1 or more GPUs.
- Xeon Phi.
- FPGA.



Motivation

	FPGA	GPU	CPU
Execution time	0.00787s	0.0858s	4.291s
Speed-up	545x	50x	1x
Dynamic power	20W	95W	40W
Total power	150W	225W	170W
Energy	1.1805J	19.305J	729.47J
Development time	60 days	3 days	1 day

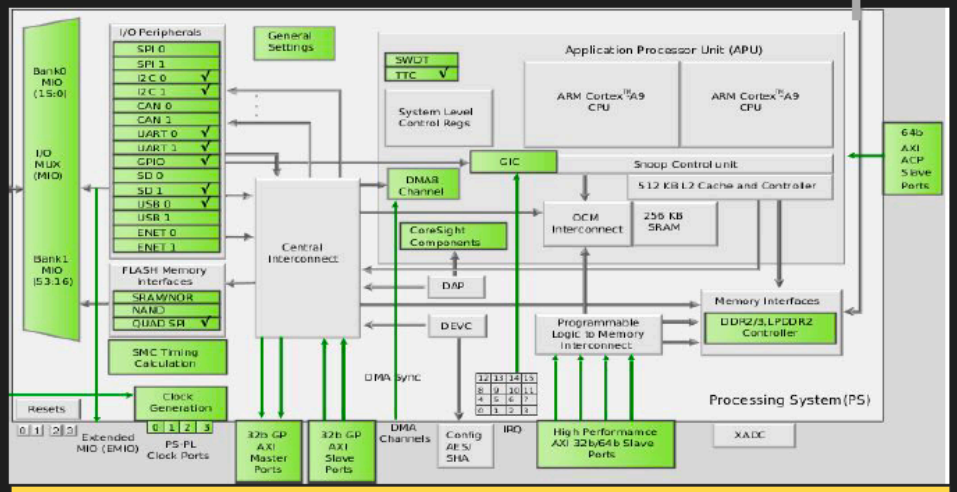
Xian Tian, Khaled Benkrid: Monte-Carlo Simulation-Based Financial Computing on the Maxwell FPGA Parallel Machine

Motivation

	FPGA	GPU	CPU
Speed-up	29x	33x	1x
Power	18W	160W	125W
Efficiency (Msamples/J)	12.8	7.5	0.6

Victor Medeiros et.al: High Performance Implementation of RTM Seismic Modeling on FPGAs: Architecture, Arithmetic and Power Issues

The Zynq ARM + FPGA System



FPGA

DRAM

Xilinx Zynq Ultrascale+

FPGAs -> HPC

FPGAs -> Data Centers

See Microsoft's use of FPGAs at amazing scale paper

Needed

A new high/low language for Zynq

And for Xeon Phi, GPU, CPU …

Allow the programmer to specify computational hierarchy suited to the application

Separately specify how that hierarchy maps to a real computational system (or accept choice of automated compiler / autotuner)

Need cost models that support that process in practice!

One program, many target platforms

(see for example Anton's work on Aplite)

Needed

A

A

A
h

Se
real computational system (or accept choice of automated compiler / autotuner)

Talk to us if you are interested in either MSc or PhD projects

There is one PhD position in Heterogeneous Systems currently advertised
And there will be more …

One program, many target platforms