

A photograph of an NVIDIA Tesla GPU card, viewed from an isometric perspective. The card is dark green and black, with a large black fan in the center. The text "NVIDIA TESLA" is visible on the top surface. The card is set against a dark grey background.

General Purpose Computations on GPUs

Parallel Functional Programming 2016

Bo Joel Svensson
Division of Computer Engineering
Chalmers

Background

Challenges in Computer Architecture

- The Power wall.
- The Instruction level parallelism (ILP) wall.
- The Memory wall.

Challenges in Computer Architecture

- The Power wall.
- The Instruction level parallelism (ILP) wall.
- The Memory wall.

All of these lead to more complicated (from the programmer's point of view) computer architectures.

The Power Wall

- More capable processors use more power.
 - It may not be possible to make use of all the resources in a chip at once. Parts of it needs to be turned off to not become too hot (draw too much power).
 - (solution) Chips consisting of different kinds of special purpose (or general purpose). compute capabilities, not all used at once (for example the big.LITTLE).

The ILP Wall

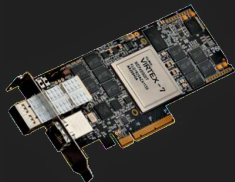
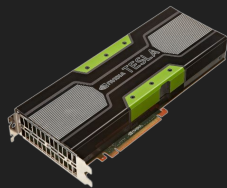
- It is now hard to push computer performance further by speeding up single threaded execution by automatic ILP or by increasing the frequency.
- (solution) More, but simpler, cores. Accelerators.

The Memory Wall

- Processor performance and memory performance show diverging trends.
- (solution) Larger caches, more complicated memory hierarchies, programmer managed scratch-pad memories.

Heterogeneous Systems

Node



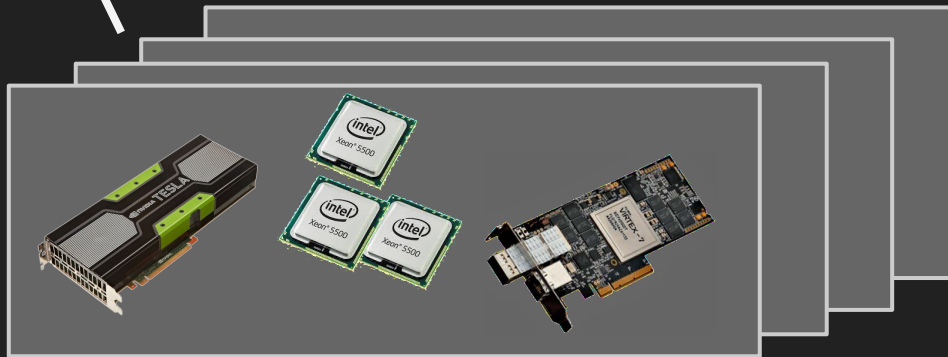
An HPC node today:

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

Upcoming:

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

Simple Programming Model

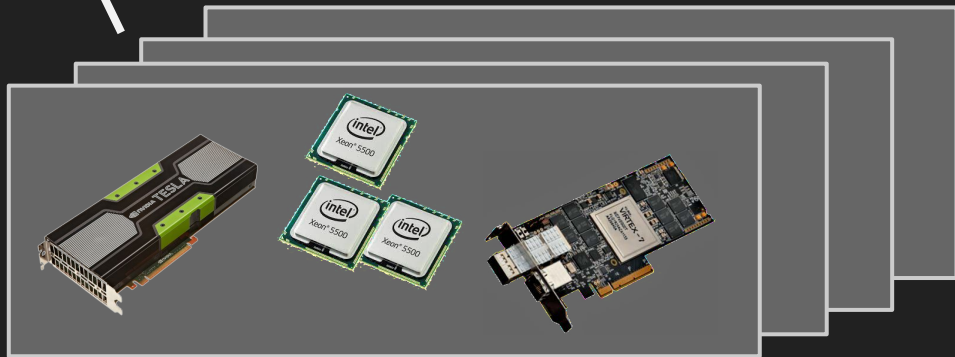


Efficient code

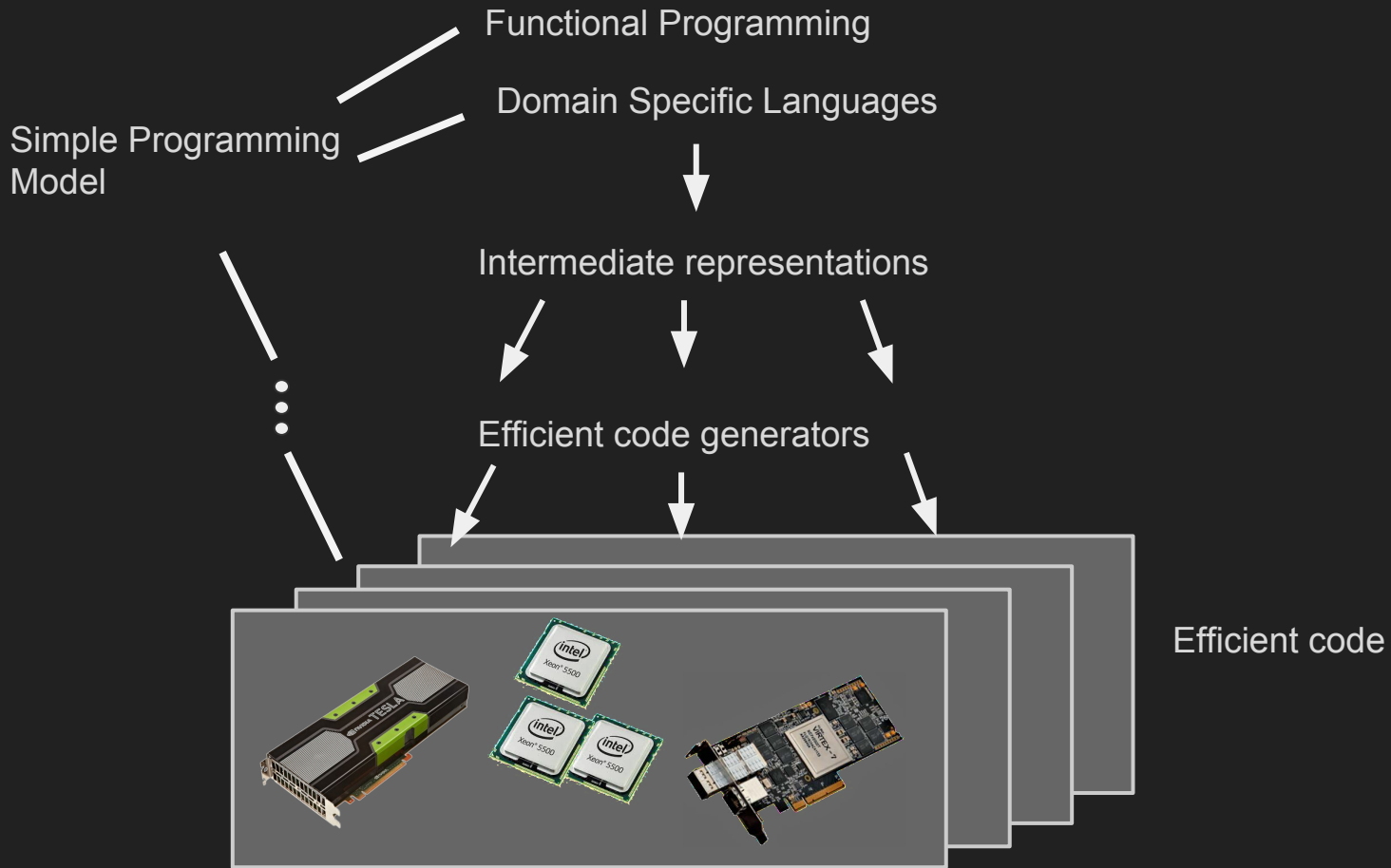
Simple Programming Model

Functional Programming

Domain Specific Languages



Efficient code

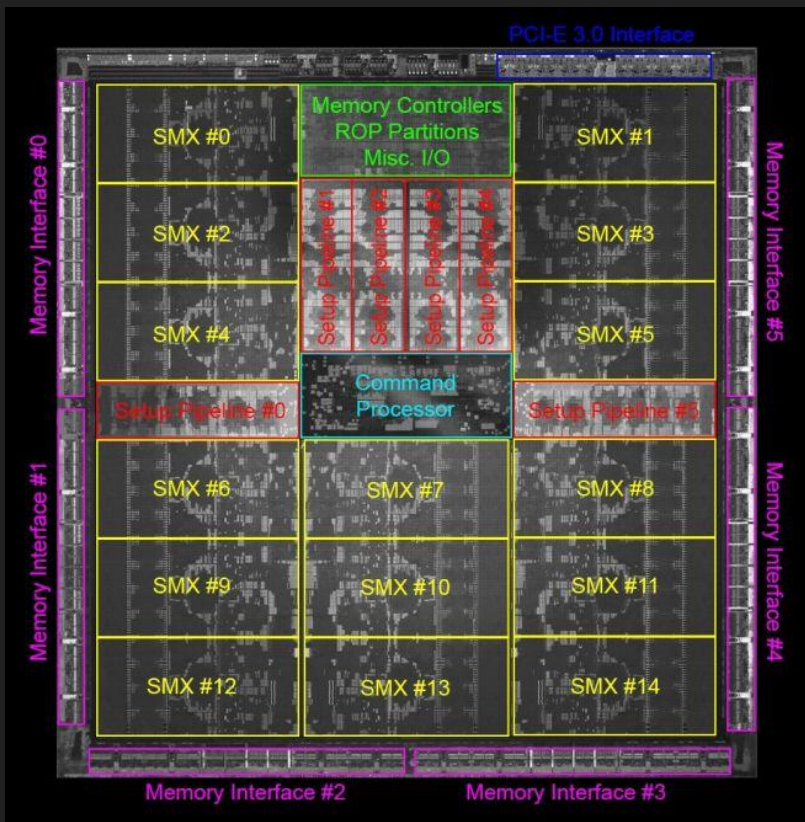


GPUs

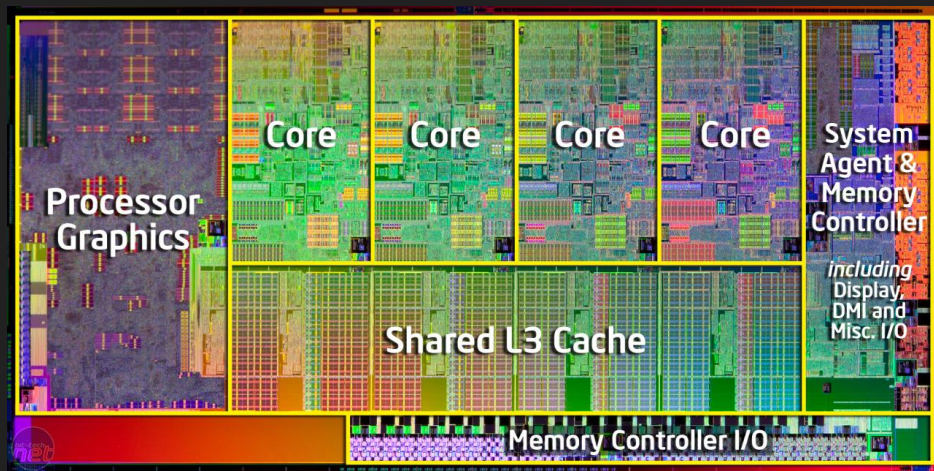
“X on the GPU”

“X on the GPU”

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



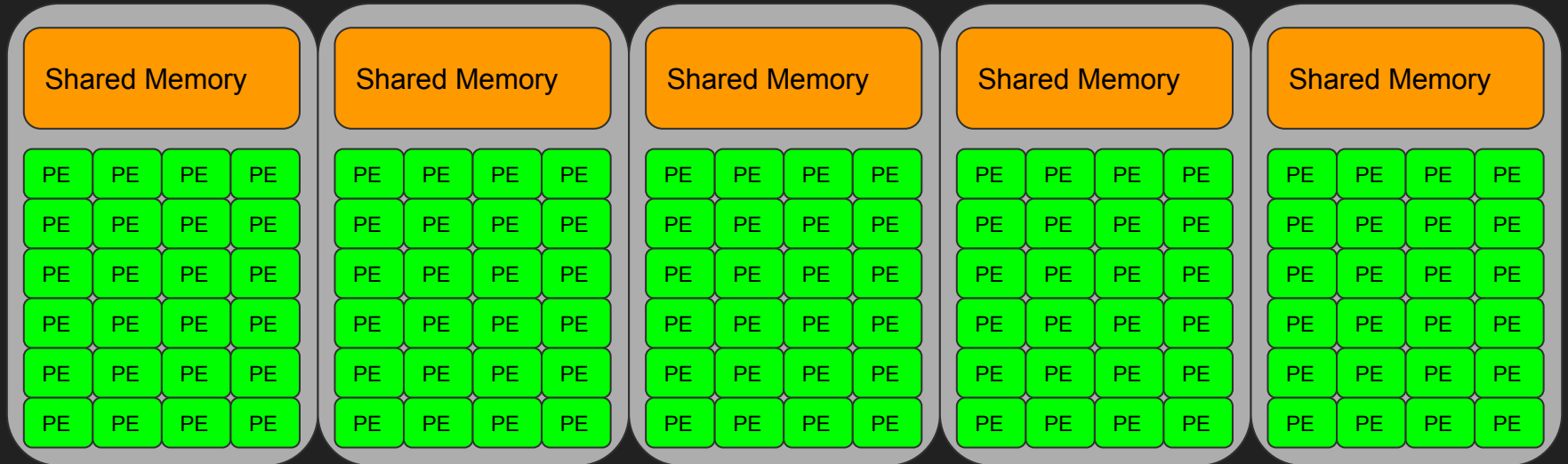
NVIDIA Kepler (GK110)



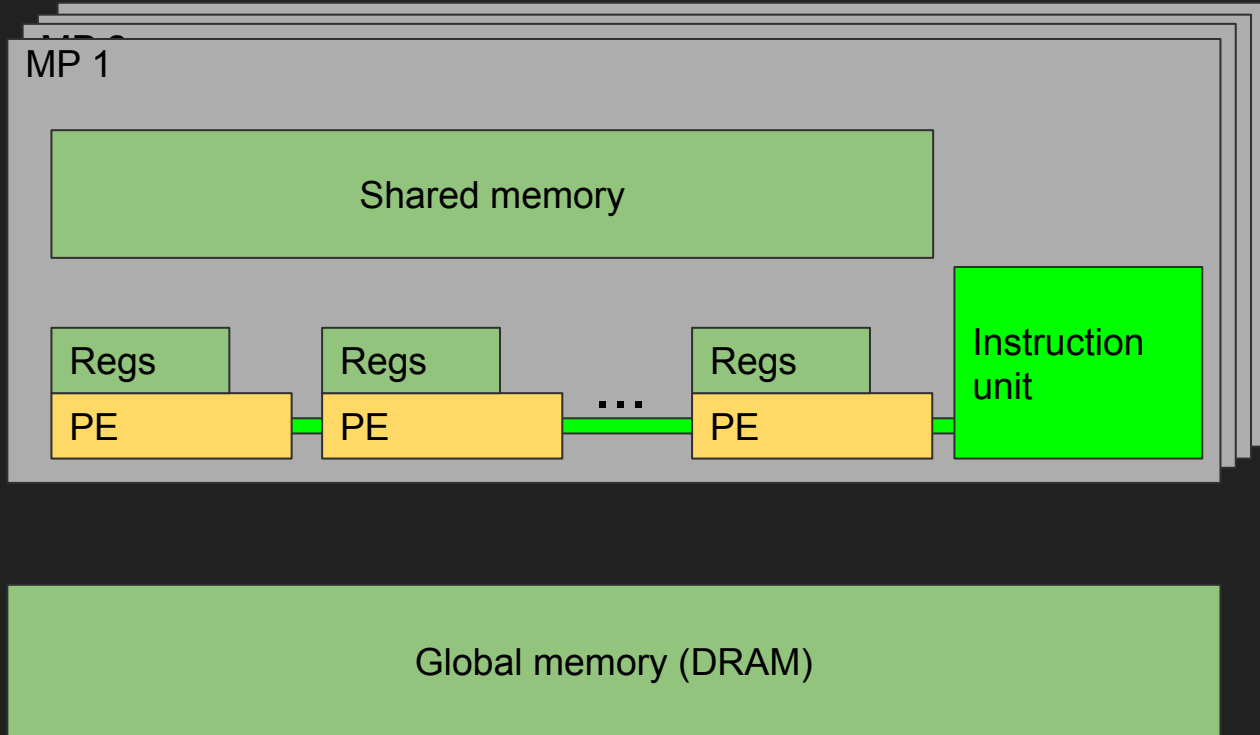
Sandy Bridge

GPU: The Architecture

DRAM



GPU: Zoom in on an MP



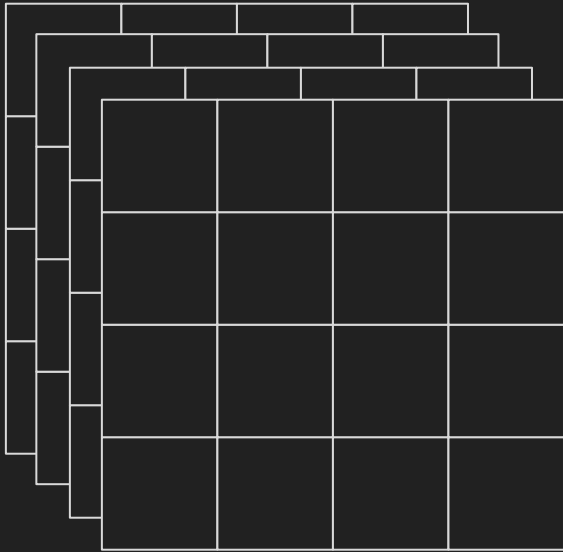
CUDA: Concepts

- 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

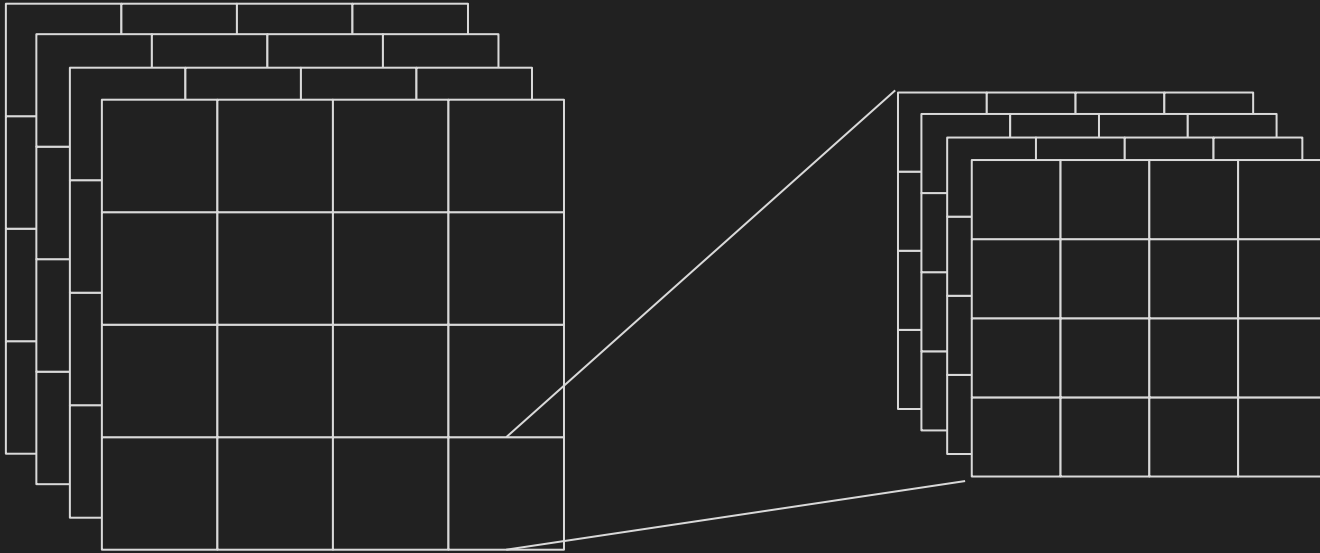
- Threads
 - All threads are described by a single program (SIMT).
- Blocks
 - Up to 1024 cooperating threads per block.
 - Many blocks share an MP.
 - More Threads per block than processors per MP. (syncthreads)
 - 1,2 or 3d shaped iteration space.
- The Grid
 - Work is launched onto the GPU in a unit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

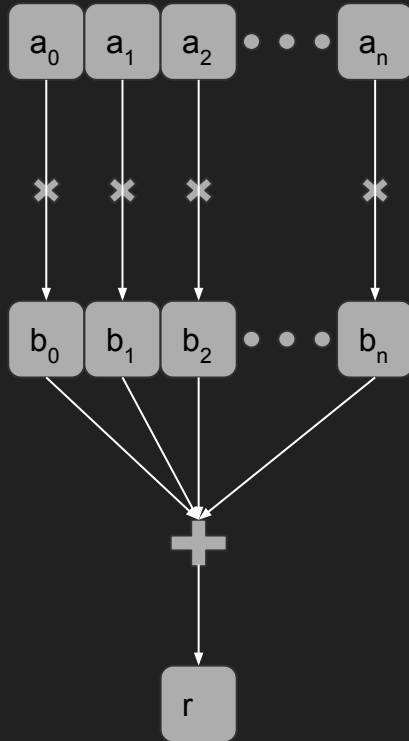
Launching a Grid

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

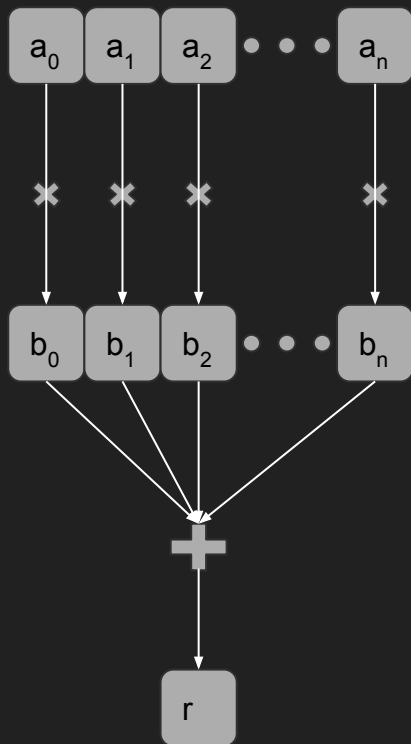
The Kernel Code

- 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);
```

Functional GPU Programming

Haskell Based Embedded Languages

Accelerate

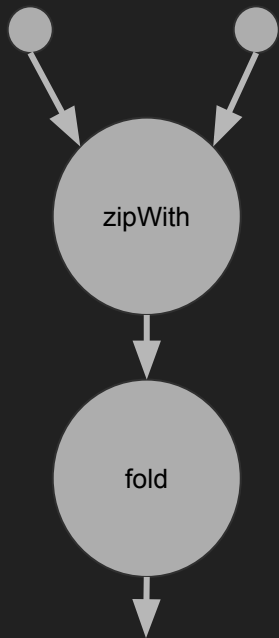
Obsidian

Accelerate

Accelerate: An Example

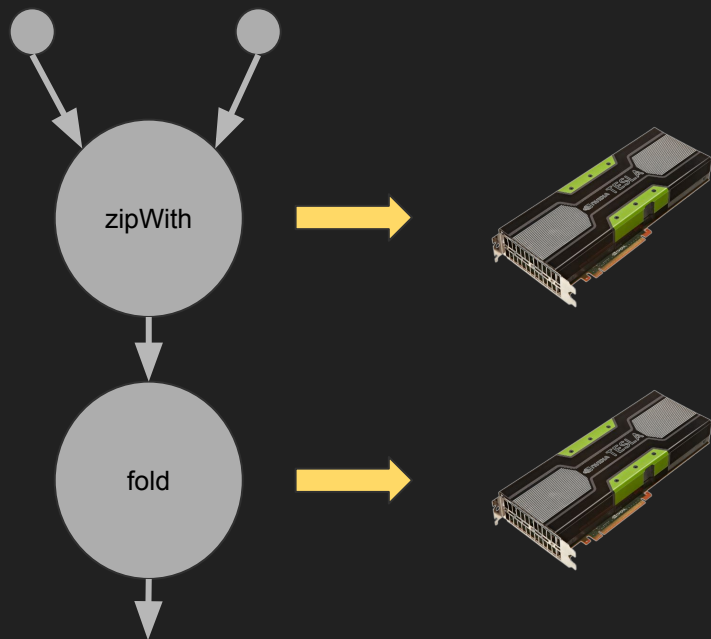
```
dotp :: Num n => Vector n -> Vector n -> Acc (Scalar n)
dotp xs ys = let xs' = use xs
               ys' = use ys
               in fold (+) 0 (zipWith (*) xs' ys')
```

Accelerate: An Example

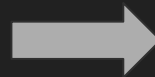
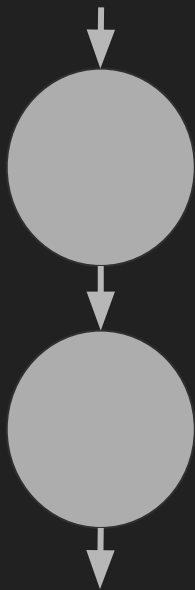


```
dotp :: Num n => Vector n -> Vector n -> Acc (Scalar n)
dotp xs ys = let xs' = use xs
               ys' = use ys
               in fold (+) 0 (zipWith (*) xs' ys')
```

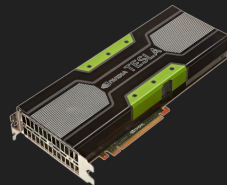
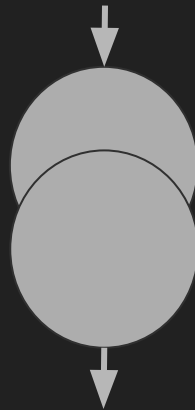

Accelerate: An Example



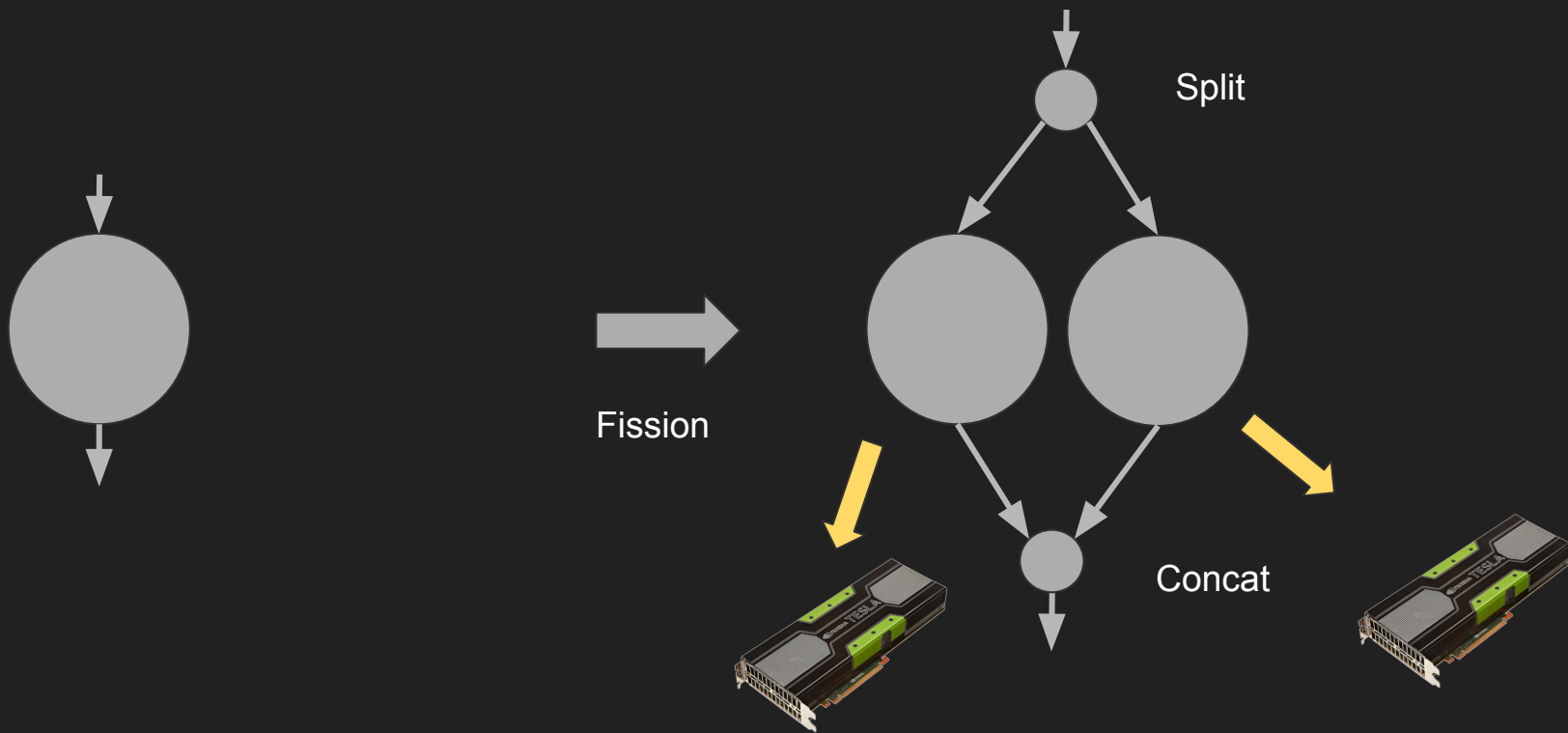
Accelerate: High Level Optimisations



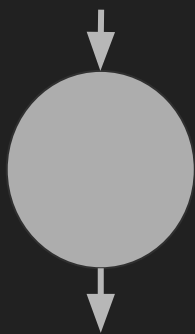
Fusion



Accelerate: High Level Optimisations



Accelerate: Operations



Generate

Permute

Map

ZipWith

Fold

Scan

...

Accelerate: Array shapes

Shapes:

```
data Z = Z
```

```
data tail :: head = tail :: head
```

One dimensional shape:

```
Z :: Int
```

Two dimensional shape:

```
Z :: Int :: Int
```

```
myShape :: Z :: Int
```

```
myShape = Z :: 10
```

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)
```

Obsidian

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.

Obsidian: A small example

```
increment :: Num a => SPull a -> SPull a  
increment arr = fmap (+1) arr
```

Obsidian: A small example

```
increment :: Num a => SPull a -> SPull a  
increment arr = fmap (+1) arr
```

```
incrementKernel :: Num a => DPull a -> DPush Grid a  
incrementKernel arr = asGrid $ fmap (push . increment) arr'  
  where  
    -- make a selection of how many elements to process per CUDA block  
    arr' = splitUp 2048 arr
```

Obsidian: A small example

```
increment :: Num a => SPull a -> SPull a  
increment arr = fmap (+1) arr
```

```
incrementKernel :: Num a => DPull a -> DPush Grid a  
incrementKernel arr = asGrid $ fmap (push . increment) arr'  
  where  
    -- make a selection of how many elements to process per CUDA block  
    arr' = splitUp 2048 arr
```

```
incrementKernel' :: Num a => Word32 -> DPull a -> DPush Grid a  
incrementKernel' n arr = asGrid $ fmap (push . increment) arr'  
  where  
    arr' = splitUp n arr
```

Obsidian: Running the small example on the GPU

```
performInc :: IO ()
performInc =
  withCUDA $
  do
    kern <- capture 512 incrementKernel

    useVector (V.fromList [0..4096 :: Word32]) $ \i ->
      withVector 4096 $ \o ->
        do
          fill o 0

          o <== (2,kern) <> i

          r <- copyOut o
          lift $ putStrLn $ show r
```

Obsidian: Pull and Push arrays

- “Delayed” arrays.
 - A description of how to compute values.
 - A “compute” functions makes the arrays “real” in memory.
- Operations on pull/push arrays automatically fuse.
 - Unless “compute” is used between operations.

Pull Arrays

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

Pull Arrays

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

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

Pull Arrays

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

Fusion:

```
arr = Pull n ixf
```

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


Push Arrays

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

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

```
type Writer a = a -> EWord32 -> Program Thread ()
```

Push Arrays

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

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

```
type Writer a = a -> EWord32 -> Program Thread ()
```

```
map f (Push s p) = Push s $ \wf -> p (\e ix -> wf (f e) ix)
```

SPull, SPush, DPull, DPush

The size of an array can be either Word32 or Exp Word32

- Statically known size a requirement in kernels that use shared memory.
- Dynamic size allowed at the top level of the hierarchy (in multiples of the block size).

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.

Obsidian: Programming the Hierarchy

Push Arrays have hierarchy level type parameter: Push t s a

- Thread, Warp, Block, Grid
- Influences how iteration pattern is realised in the generated CUDA code.
 - Sequential / parallel

Obsidian: Programming the Hierarchy

The type parameter seen earlier on
Push arrays and on Programs.

```
data Thread  
data Step t
```

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

Obsidian: Programming the Hierarchy

The type parameter seen earlier on
Push arrays and on Programs.

```
data Thread  
data Step t
```

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

```
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)
```

Obsidian: Programming the Hierarchy

```
class (t *<=* Block) => AsBlock t where
  asBlock :: SPull (SPush t a) ->
           SPush Block a
  asBlockMap :: (a -> SPush t b)
              -> SPull a
              -> SPush Block b
```

```
instance AsBlock Thread where
  asBlock      = tConcat
  asBlockMap f = tConcat . fmap f
```

```
instance AsBlock Warp where
  asBlock      = pConcat
  asBlockMap f = pConcat . fmap f
```

```
instance AsBlock Block where
  asBlock      = sConcat
  asBlockMap f = sConcat . fmap f
```


Obsidian: Programming the Hierarchy

```
class (t *<=* Block) => AsBlock t where
  asBlock :: SPull (SPush t a) ->
            SPush Block a
  asBlockMap :: (a -> SPush t b)
              -> SPull a
              -> SPush Block b
```

```
class (t *<=* Warp) => AsWarp t
```

```
instance AsBlock Thread where
  asBlock      = tConcat
  asBlockMap f = tConcat . fmap f
```

```
instance AsBlock Warp where
  asBlock      = pConcat
  asBlockMap f = pConcat . fmap f
```

```
instance AsBlock Block where
  asBlock      = sConcat
  asBlockMap f = sConcat . fmap f
```

Obsidian: Programming the Hierarchy

```
class (t *<=* Block) => AsBlock t where
  asBlock :: SPull (SPush t a) ->
    SPush Block a
  asBlockMap :: (a -> SPush t b)
    -> SPull a
    -> SPush Block b
```

```
instance AsBlock Thread where
  asBlock      = tConcat
  asBlockMap f = tConcat . fmap f
```

```
instance AsBlock Warp where
  asBlock      = pConcat
  asBlockMap f = pConcat . fmap f
```

```
instance AsBlock Block where
  asBlock      = sConcat
  asBlockMap f = sConcat . fmap f
```

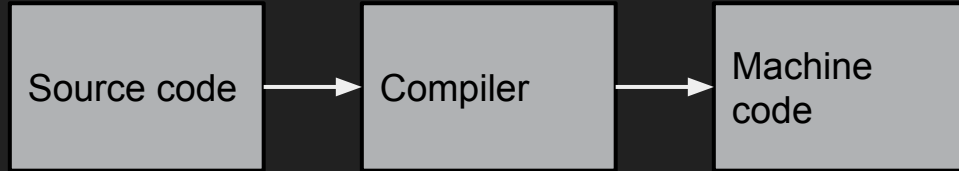
```
class (t *<=* Warp) => AsWarp t
```

```
asThread :: ASize l
  => Pull l (SPush Thread b)
  -> Push Thread l b
```

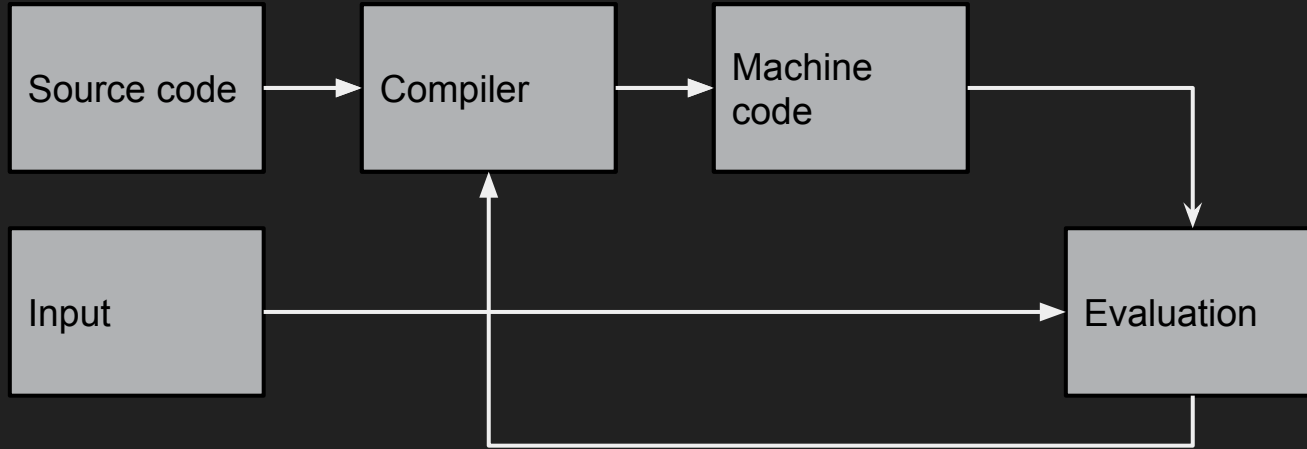
```
asGrid :: ASize l
  => Pull l (SPush Block a)
  -> Push Grid l a
```

Obsidian and Auto-Tuning

Compilation



Compilation: Tuning framework



Tuning Framework

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

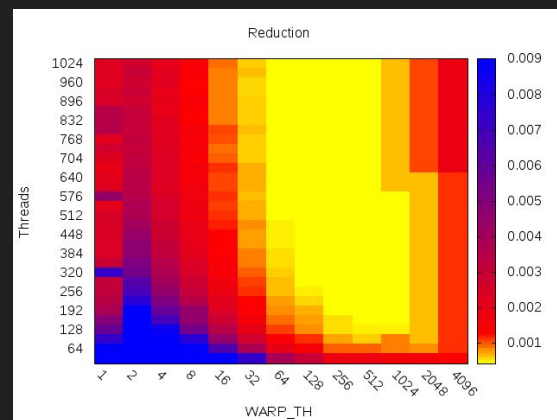
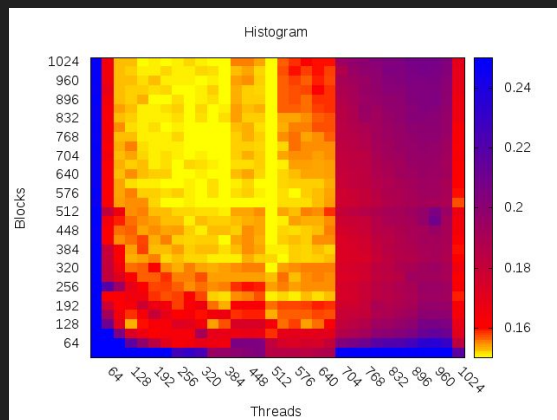
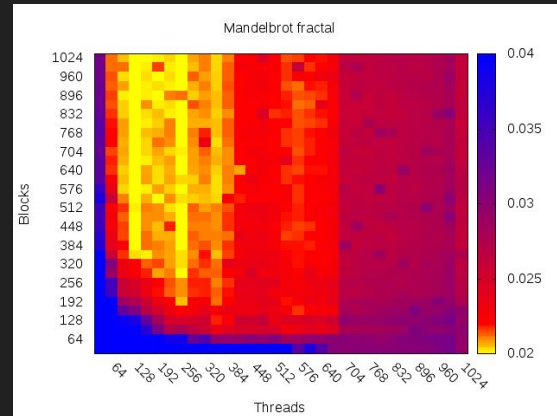
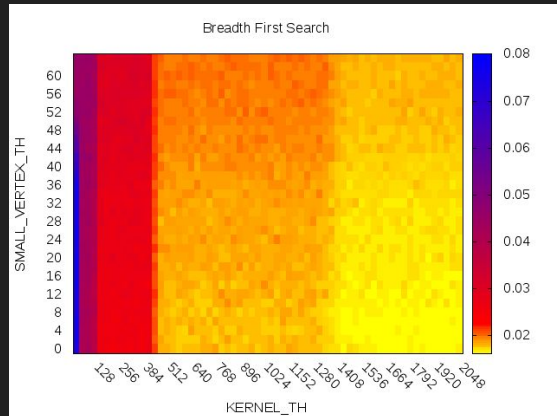
```
type ParamIdx = Int
```

```
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
  )
```

Obsidian: Tuning

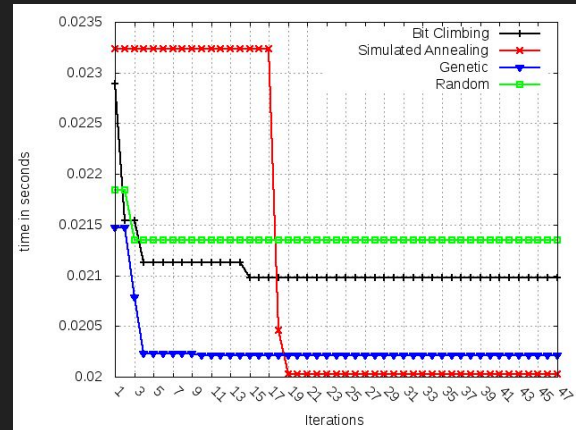
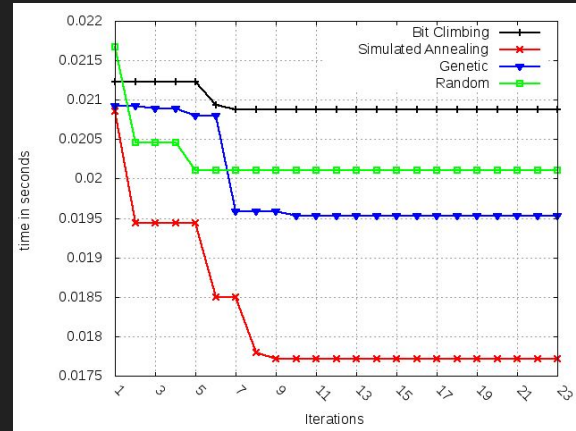
- Auto-tuning
- Specialised code variants



Obsidian: Tuning

- Exhaustive
- Random
- Simulated annealing
- Hill climbing

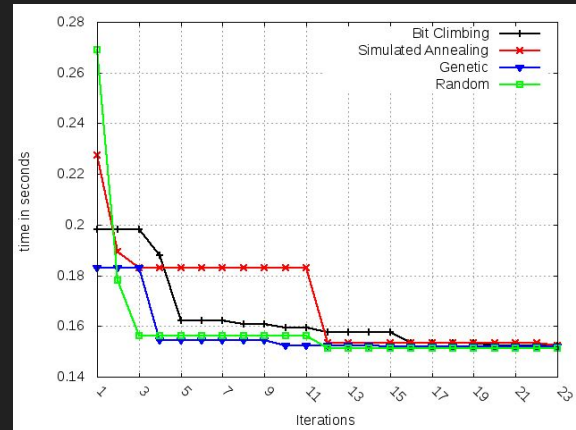
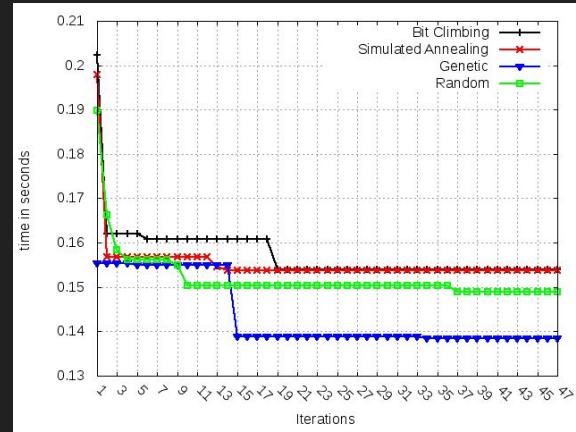
Mandelbrot



Obsidian: Tuning

- Exhaustive
- Random
- Simulated annealing
- Hill climbing

Histogram



Obsidian And Accelerate Conclusions

Obsidian

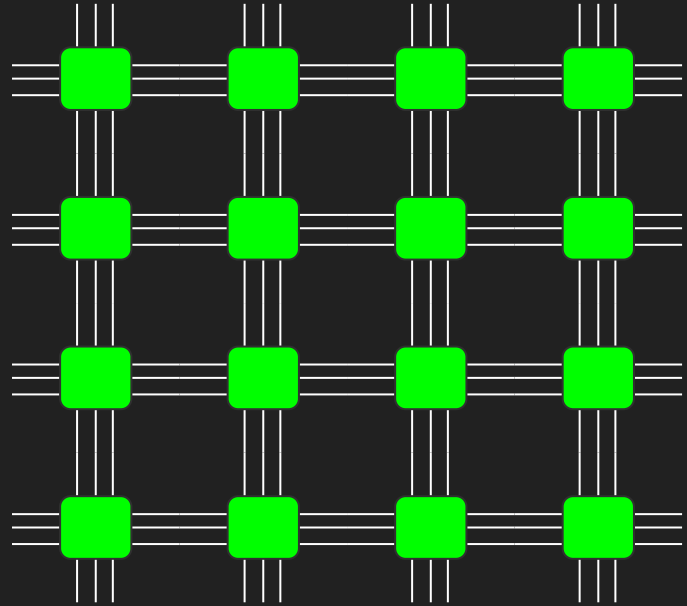
- Control
 - of what the GPU actually does.
 - of Shared Memory .
- Kernels.

Accelerate

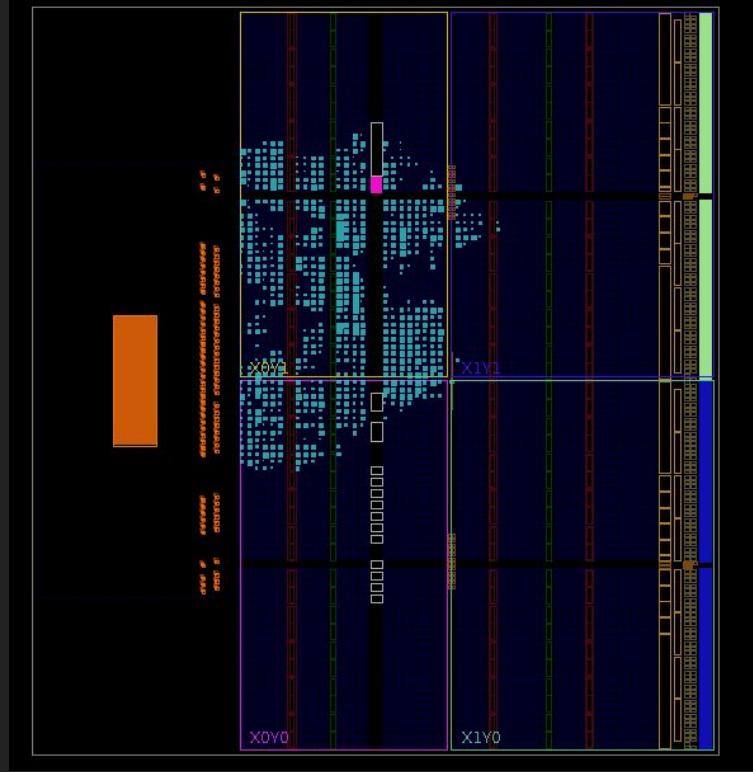
- High level programs.
 - High level optimisations.
- Entire applications.
- Multi-device RTS.

High Performance Computing and FPGAs

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



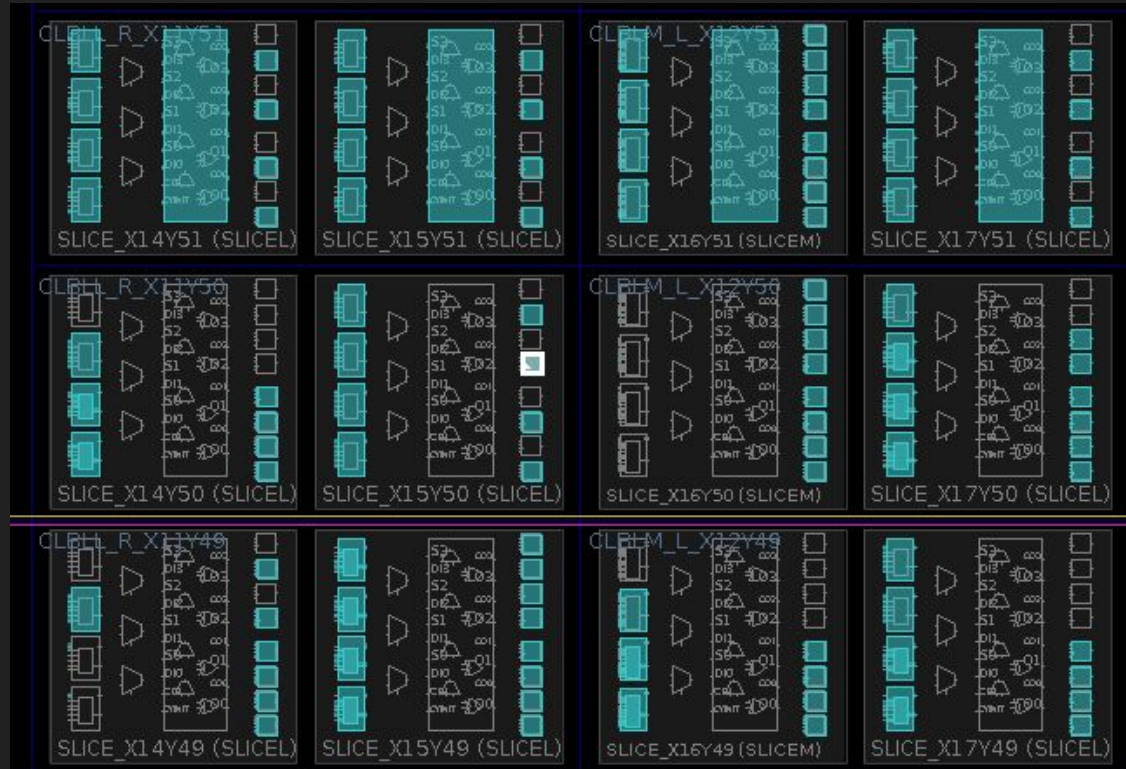
FPGA: What ?



FPGA: What ?



FPGA: What ?



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

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

Motivation

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

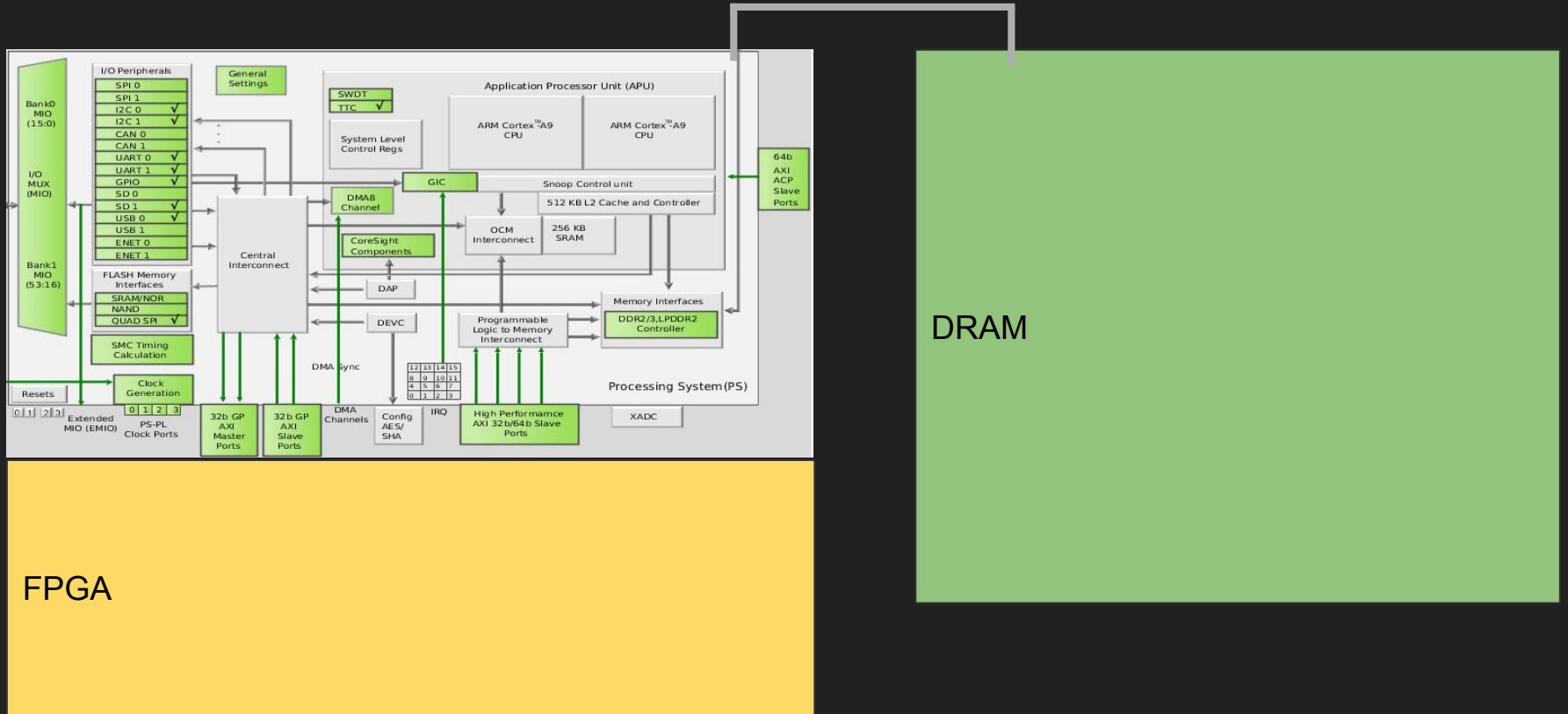
FPGA: Programming

- Bluespec/Bluecheck
 - Generate Verilog from high level models
 - Testing
- The traditional way: Verilog, VHDL
- Well supported HLS: OpenCL, C, SystemC
- Maxeler MaxJ
 - Embedded Language (In Java!)
- Lava, Wired, Kansas Lava. York Lava
 - Embedded Languages (Haskell), generates VHDL
- Feldspar to FPGA
 - Is work in progress.

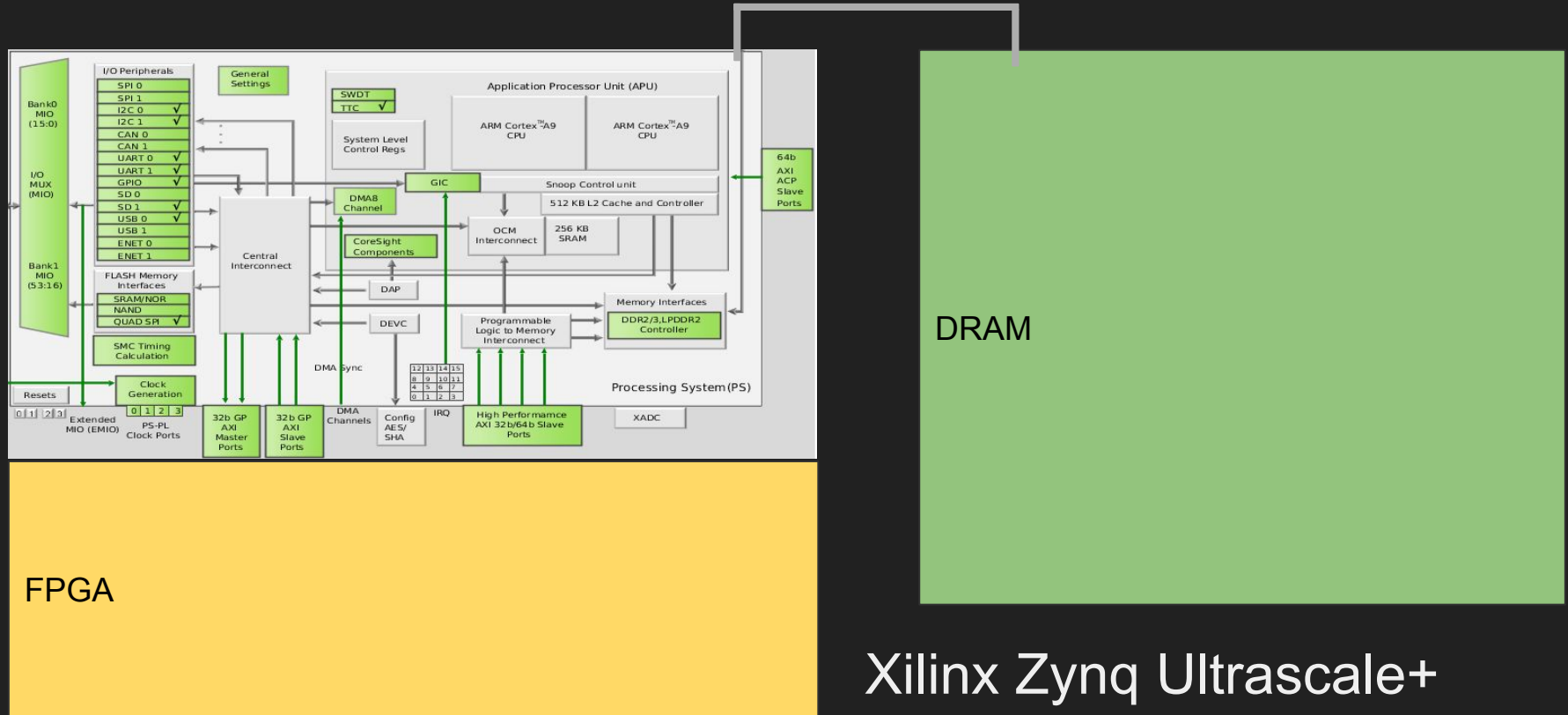
FPGA: How should it be used

- “Soft-core” (MicroBlaze).
- Application specialised instruction sets.
- Implement ϵ in hardware.

The Zynq ARM + FPGA System



The Zynq ARM + FPGA System



Xilinx Zynq Ultrascale+

The Zynq ARM + FPGA System

Retarget Obsidian for the Zynq:

- OpenCL generation, with FPGAs in mind, is work in progress.
- But is this even a good idea ?

The Zynq ARM + FPGA System

Retarget Obsidian for the Zynq:

- Obsidian mimics the GPU hierarchy.
- OpenCL mimics the GPU hierarchy.
- On an FPGA we are not bound by a specific hierarchy (well..)!

The Zynq ARM + FPGA System

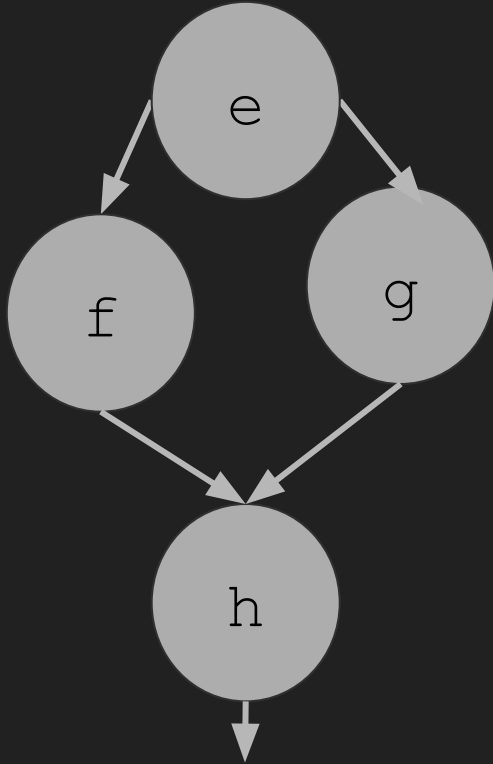
~~Retarget Obsidian~~ Implement a new high/low level language for the Zynq:

- And for the GPU, Xeon Phi, CPU ...
- Allowing the programmer to specify computational hierarchies that are perfect for the application.
- One high level program, many target platforms.
- Describe how the programmer specified computation hierarchy maps to a fixed processing hierarchy. (Or accept best effort from an automatic transformation)
- If interested in contributing in this exciting area, talk to us about a Master's thesis project.

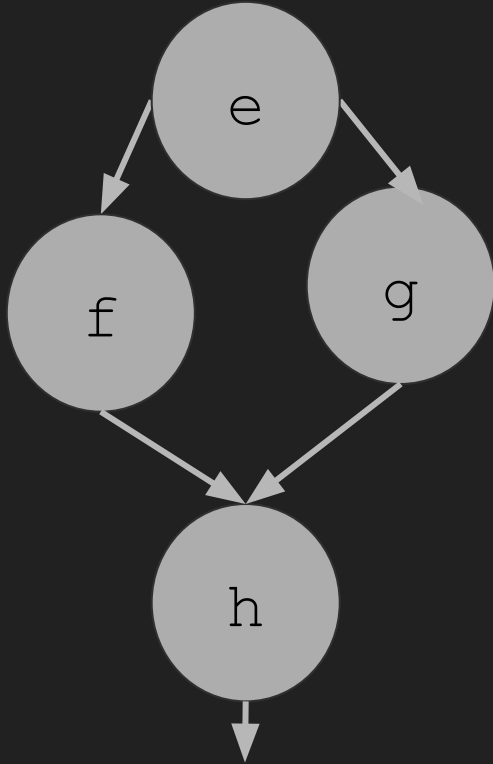
“X on an FPGA” ?

The Future is Heterogeneous

CPU, GPU and FPGA

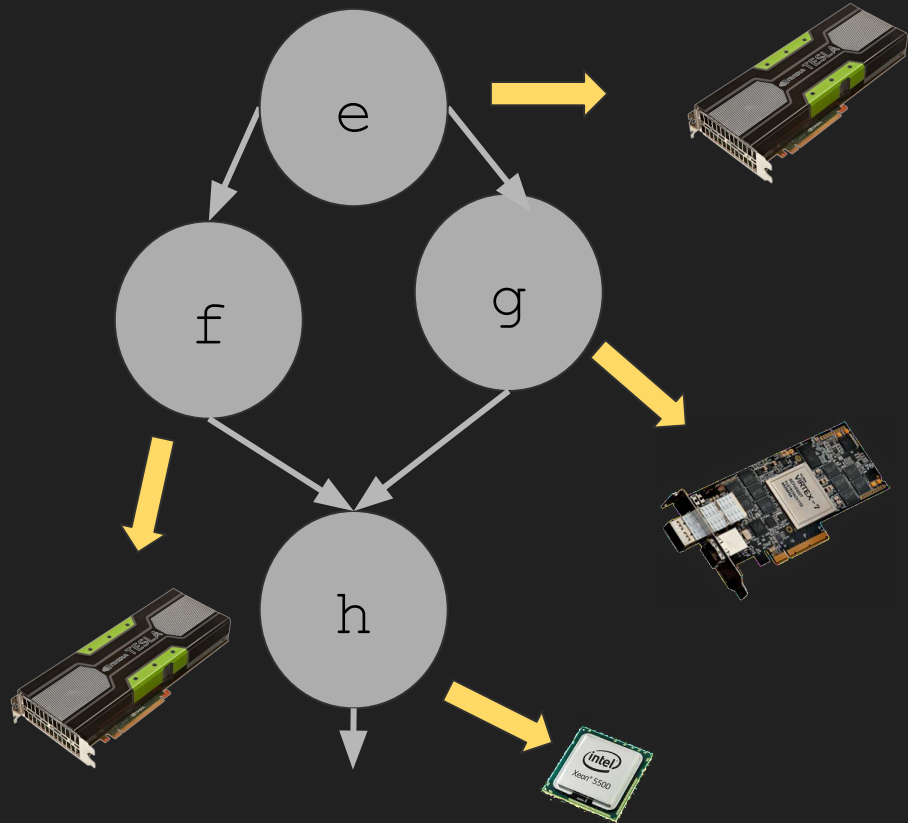


CPU, GPU and FPGA



- Scheduling issues
 - Reconfiguration
 - Suitability
 - Data locality
 - Power
 - Availability

CPU, GPU and FPGA



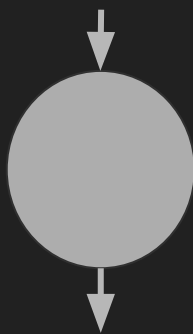
- Scheduling issues
 - Reconfiguration
 - Suitability
 - Data locality
 - Power
 - Availability

Heterogeneous Computing Challenges

- Runtime systems.
 - What to execute where (CPU,XeonPhi, GPU, FPGA...).
 - When to reconfigure FPGA.
 - Scheduling
 - for speed.
 - for low power consumption.
 - for total system utilisation.
- Programming (accessibility).
 - Languages, Libraries and Tools.
- This is another exciting area where you can contribute (Master's thesis ?).

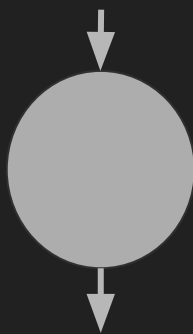
The End

Obsidian: An Example



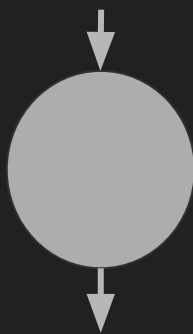
```
red f arr
| len arr == 1 = return (push arr)
| otherwise    =
do let (a1,a2) = halve arr
    imm <- compute (zipWith f a1 a2)
    red f imm
```


Obsidian: An Example



```
reduce n f arr =
  do
    execBlock $ do
      arr' <- compute (asBlock (fmap (seqRed f) (splitUp n arr)))
      red f arr'
  where
    red f arr
      | len arr == 1 = return (push arr)
      | otherwise    =
        do let (a1,a2) = halve arr
            imm <- compute (zipWith f a1 a2)
            red f imm
```

Obsidian: An Example



```
reduce :: Data a
       => Word32
       -> (a -> a -> a)
       -> Pull Word32 a
       -> Push Block Word32 a

reduce n f arr =
  do
    execBlock $ do
      arr' <- compute (asBlock (fmap (seqRed f) (splitUp n arr)))
      red f arr'

where
  red f arr
  | len arr == 1 = return (push arr)
  | otherwise    =
    do let (a1,a2) = halve arr
         imm <- compute (zipWith f a1 a2)
         red f imm
```