Máté Karácsony

Zeldspar

The road to Epiphany

(a.k.a. “Using Fusion to Enable Late Design Decisions for Pipelined Computations” @ FHPC'16)

With many thanks to: Emil Axelsson, Koen Claessen,
Anton Ekblad and Mary Sheeran

Budapest Haskell Hackathon 2016

Zeldspar?

Feldspar

EDSL in Haskell for digital signal processing

+

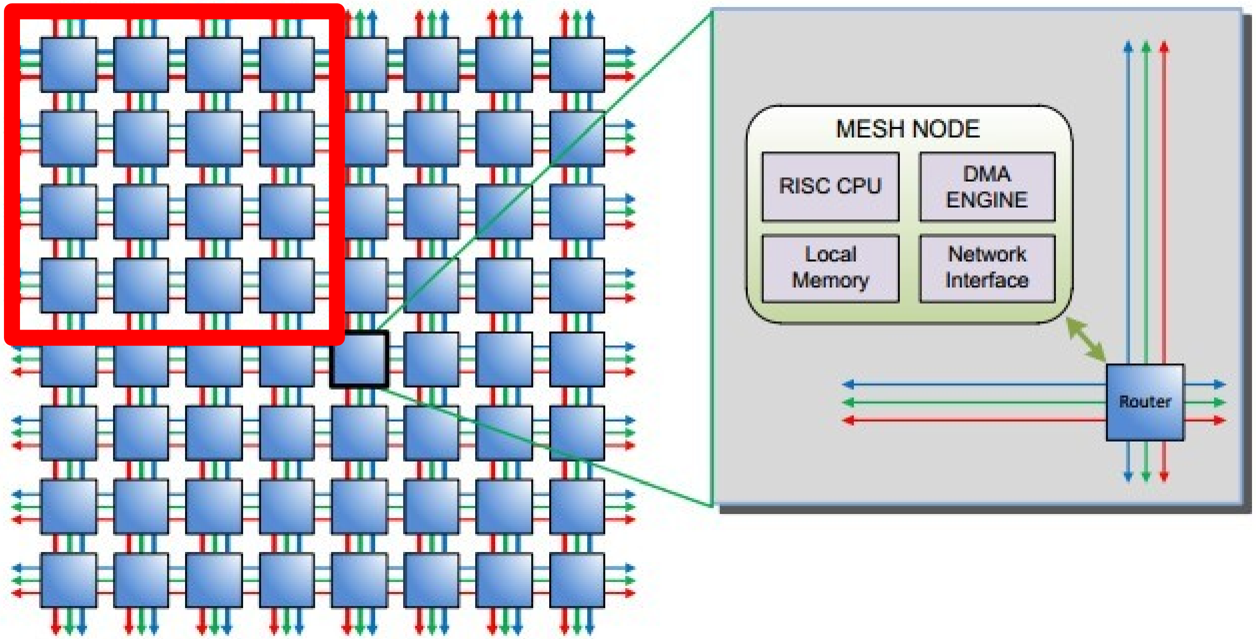
Ziria

DSL for low-level, pipelined bitstream processing

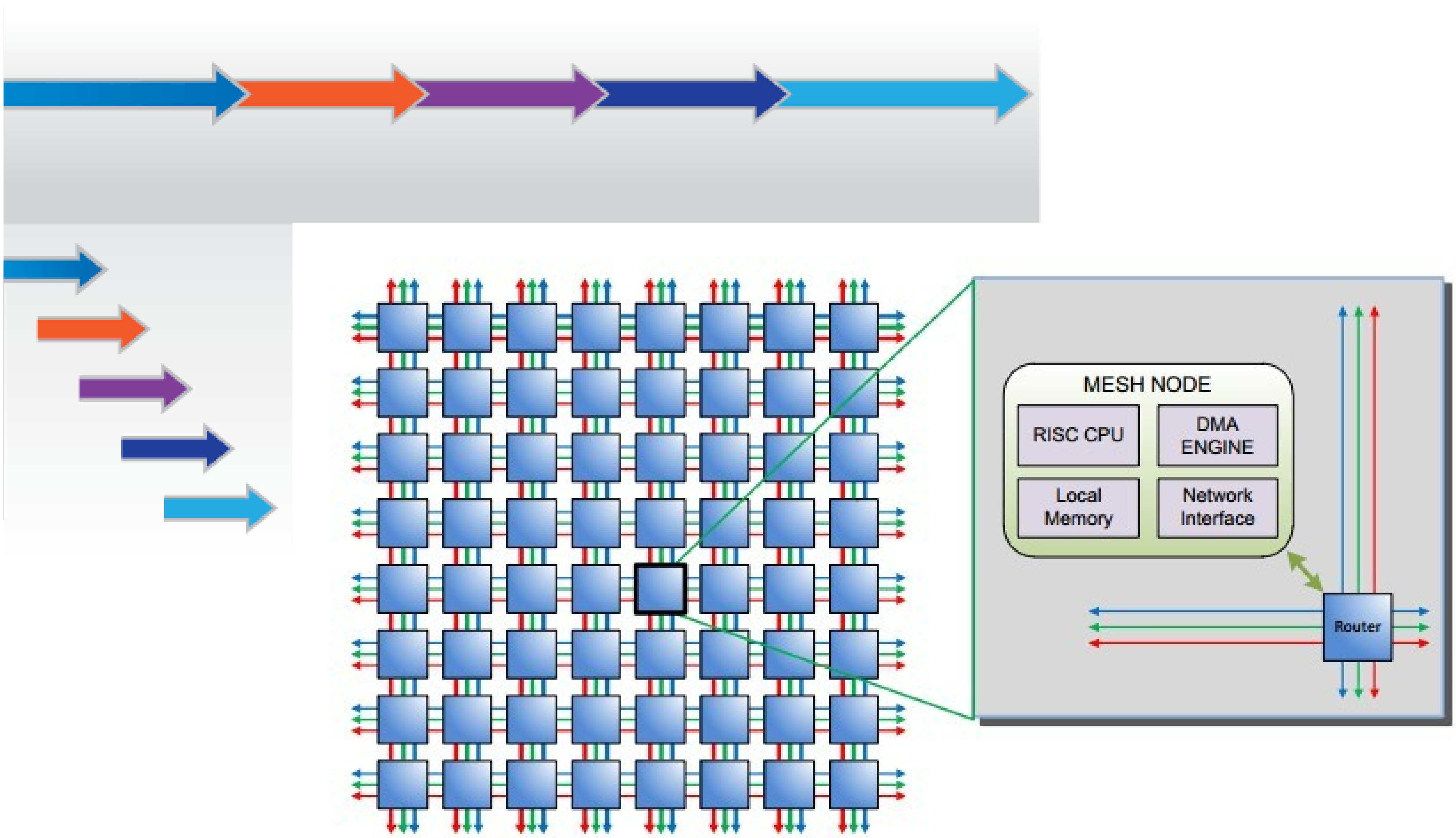
Zeldspar

EDSL in Haskell for constructing
digital signal processing pipelines

… *Epiphany?*


MESH NODE
RISC CPU
DMA ENGINE
Local Memory
Network Interface
Router

Zeldspar + Epiphany!



Language stack

```
Z
ParZ
>>>
|>>>|
```

Zeldspar

Transformer blocks, pipelining operators, program fusion

```
Data a, Pull a, …
Comp
Run
```

RAW-Feldspar

Primitives, expressions, vectors with fusion

```
ControlCMD, FileCMD
RefCMD, ArrCMD
C_CMD, PtrCMD
ThreadCMD, ChanCMD
```

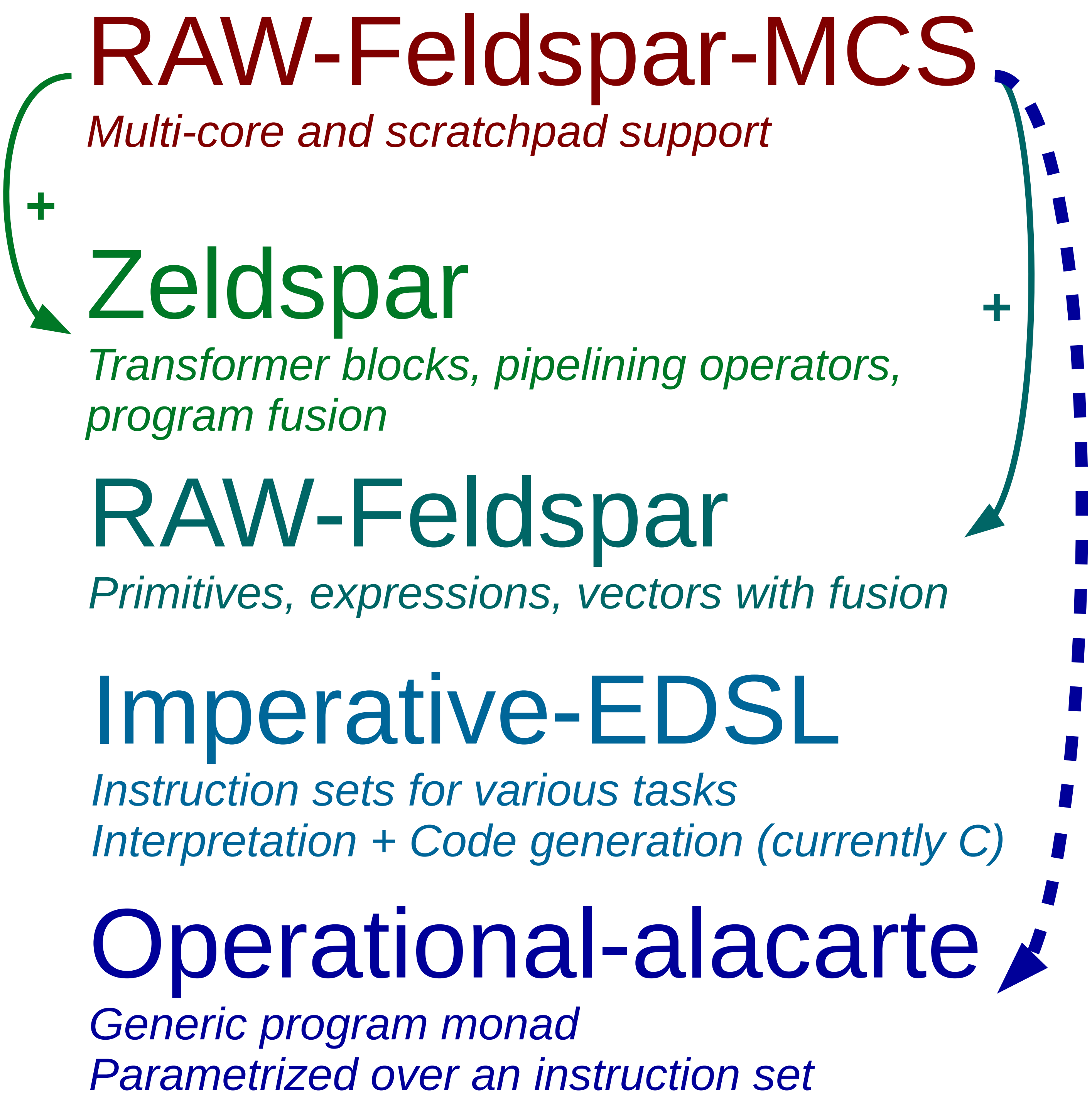
Imperative-EDSL

Instruction sets for various tasks
Interpretation + Code generation (currently C)

```
ProgramT
Inst1 :+: Inst2
```

Operational-alacarte

Generic program monad
Parametrized over an instruction set



```
SharedArr, LocalArr
CoreComp, Host
onHost, onCore
CoreZ, MulticoreZ
```

RAW-Feldspar-MCS

Multi-core and scratchpad support

```
Z
ParZ
>>>
|>>>|
```

Zeldspar

Transformer blocks, pipelining operators, program fusion

```
Data a, Pull a, …
Comp
Run
```

RAW-Feldspar

Primitives, expressions, vectors with fusion

```
ControlCMD, FileCMD
RefCMD, ArrCMD
C_CMD, PtrCMD
ThreadCMD, ChanCMD
```

Imperative-EDSL

Instruction sets for various tasks
Interpretation + Code generation (currently C)

```
ProgramT
Inst1 :+: Inst2
```

Operational-alacarte

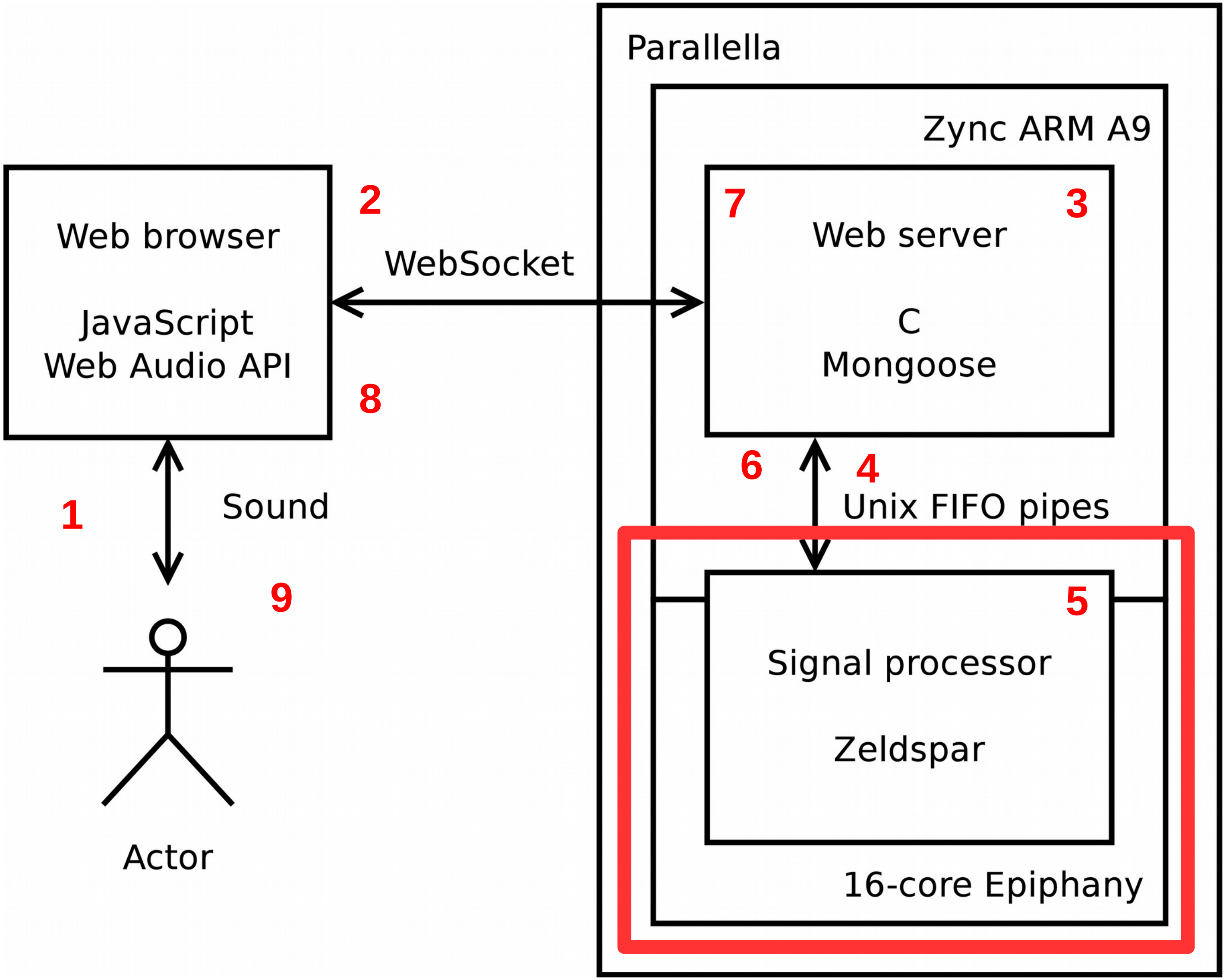
Generic program monad
Parametrized over an instruction set

Action!

Real-time signal processing

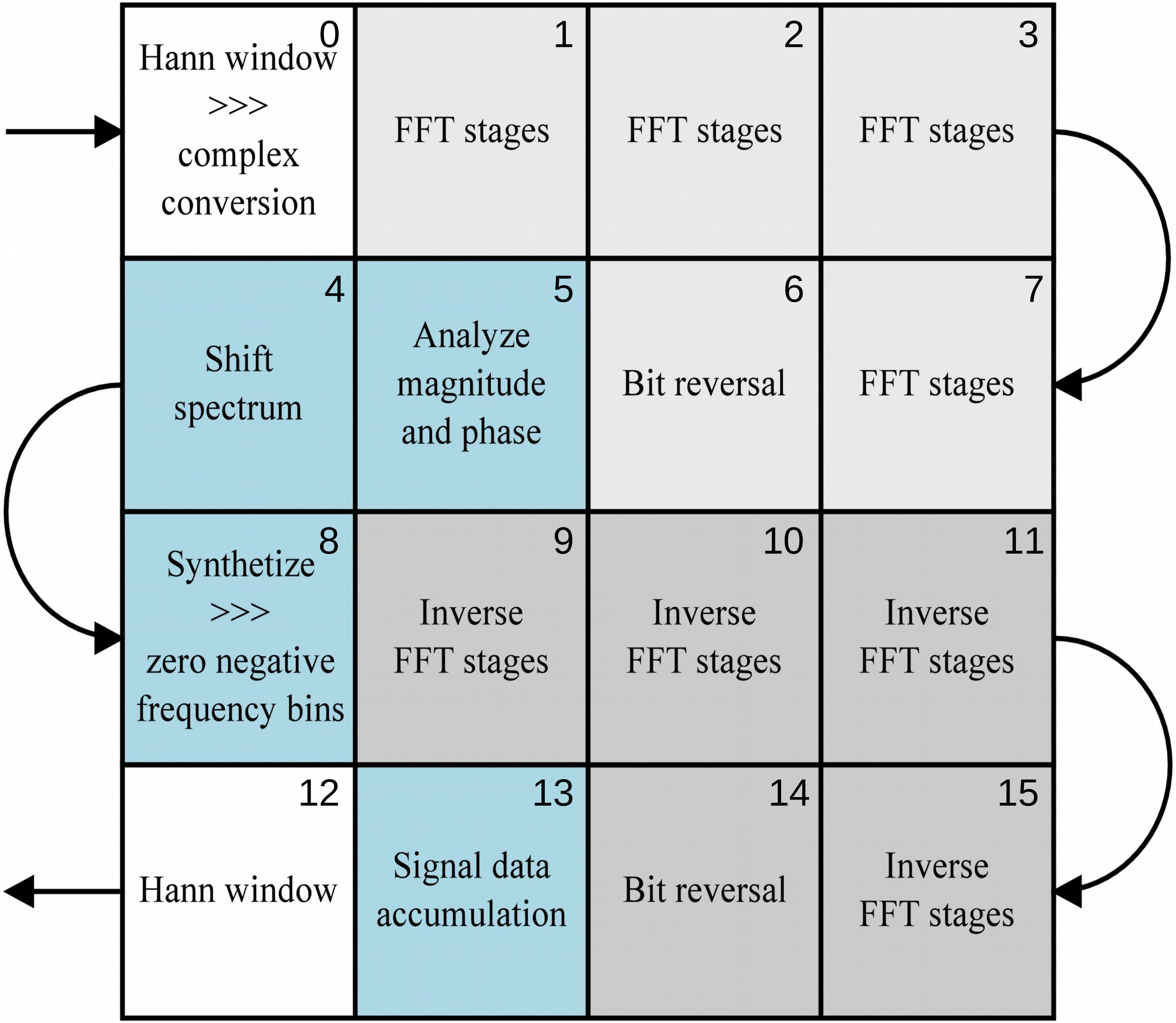
(on a real Parallella)

How?



Parallella
Zync ARM A9
Web browser
JavaScript
Web Audio API
2
WebSocket
8
7
3
Web server
C
Mongoose
6
4
Unix FIFO pipes
1
Sound
9
5
Signal processor
Zeldspar
16-core Epiphany
Actor

16-core Epiphany chip

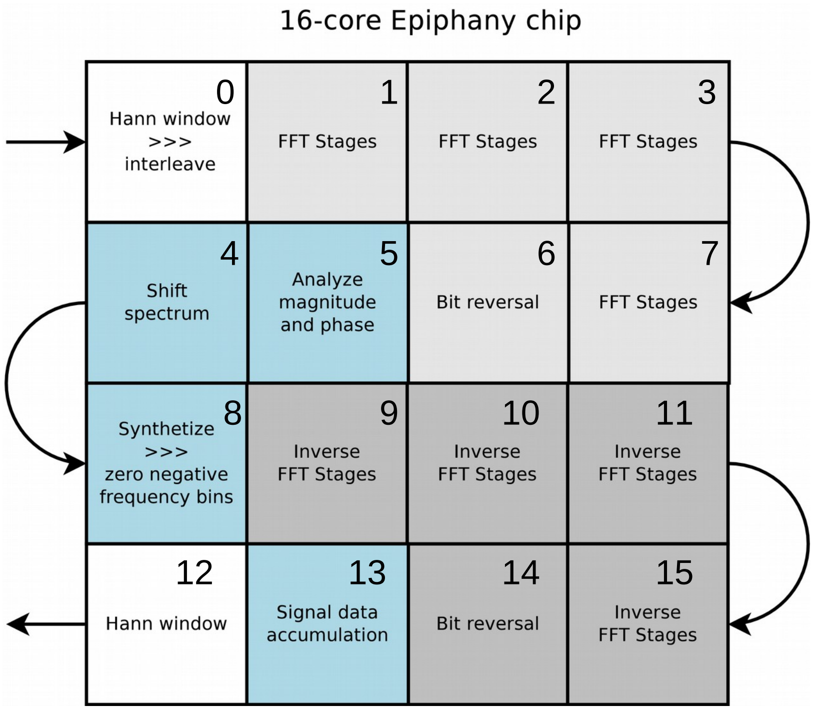


0 Hann window >>> complex conversion	1 FFT stages	2 FFT stages	3 FFT stages
4 Shift spectrum	5 Analyze magnitude and phase	6 Bit reversal	7 FFT stages
8 Synthetize >>> zero negative frequency bins	9 Inverse FFT stages	10 Inverse FFT stages	11 Inverse FFT stages
12 Hann window	13 Signal data accumulation	14 Bit reversal	15 Inverse FFT stages

The original sequential implementation in C: http://blogs.zynaptiq.com/bernsee

```
                                                     |>>>|
((window >>> asComplex)       `on` 0  |>>chanSize>>|
 fft  fftSize'            [1,2,3,7,6] |>>chanSize>>|
 analyze                      `on` 5  |>>halfChanSize>>|
 shiftPitch                   `on` 4  |>>halfChanSize>>|
 (synthetize >>> zeroNegBins) `on` 8  |>>chanSize>>|
 ifft fftSize'        [9,10,11,15,14] |>>chanSize>>|
 accumulate                   `on` 13 |>>chanSize>>|
 window                       `on` 12 )
```

Same mapping in Zeldspar!

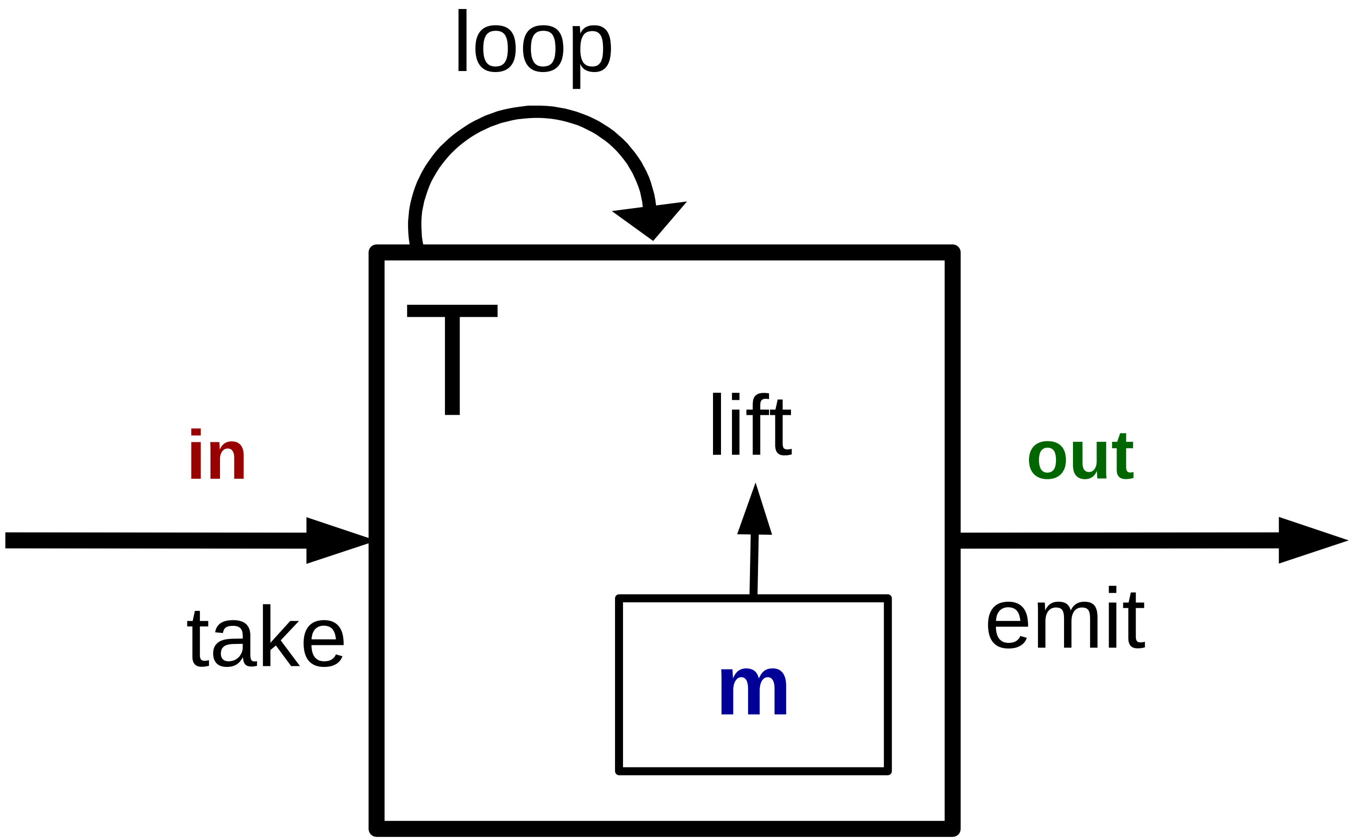


16-core Epiphany chip

0 Hann window >>> interleave	1 FFT Stages	2 FFT Stages	3 FFT Stages
4 Shift spectrum	5 Analyze magnitude and phase	6 Bit reversal	7 FFT Stages
8 Synthetize >>> zero negative frequency bins	9 Inverse FFT Stages	10 Inverse FFT Stages	11 Inverse FFT Stages
12 Hann window	13 Signal data accumulation	14 Bit reversal	15 Inverse FFT Stages

Zeldspar transformer block

T :: Z in out m a



```
type CoreZ inp out a = Z inp out CoreComp a
type RealSamples     = DPull Float
type ComplexSamples  = DPull (Complex Float)


window :: CoreZ RealSamples RealSamples ()
window = do
  hann <- lift $ ... -- calculate coefficients
  loop $ do
    input <- take
    emit $ zipWith (*) input hann

asComplex :: CoreZ RealSamples ComplexSamples ()
asComplex = loop $ do
  input <- take
  emit $ fmap (flip complex 0) input
```

Pipeline Fusion

```
(>>>) :: (Monad m, Storable mid)
      => Z inp mid m ()
      -> Z mid out m ()
      -> Z inp out m ()
```

```
window    :: CoreZ RealSamples RealSamples ()
asComplex :: CoreZ RealSamples ComplexSamples ()

(window >>> asComplex)
 :: CoreZ RealSamples ComplexSamples ()
```

Pipeline Fusion → Vector Fusion

```
(window >>> asComplex)
 :: CoreZ RealSamples ComplexSamples ()

float _a2[…] …;
float *a2 = _a2; // input

float _Complex _a5[…] …;
float _Complex *a5 = _a5; // output

for (v6 = 0; v6 < …; v6++) {
  a5[v6] = a2[v6] * a0[v6]; // fused programs
  // a0 contains Hann coefficients
}
```

Core mapping

```
type CoreId = Word32

on :: ( CoreTransferable minp
      , CoreTransferable mout
      , CoreTransferType CoreComp cinp minp
      , CoreTransferType CoreComp cout mout )
   => CoreZ cinp cout a
   -> CoreId
   -> MulticoreZ minp mout a

(window >>> asComplex) `on` 0
 :: MulticoreZ (Store RealSamples)
               (Store ComplexSamples) ()
```

Parallel composition

```
-- (p1 |>>>| p2)

(|>>>|) :: PrimType mid
        => MulticoreZ inp (Data mid) a
        -> MulticoreZ (Data mid) out b
        -> MulticoreZ inp out ()

-- (p1 |>>vectorLength>>| p2)

-- ((p1 |>> vectorLength)     p2)

(|>>) :: CoreTransferable mid
      => MulticoreZ inp mid a
      -> SizeSpec mid
      -> (MulticoreZ mid out b -> MulticoreZ inp out ())


(>>|) :: (MulticoreZ mid out b -> MulticoreZ inp out ())
      -> MulticoreZ mid out b
      -> MulticoreZ inp out ()
```

Transfer types and fusion

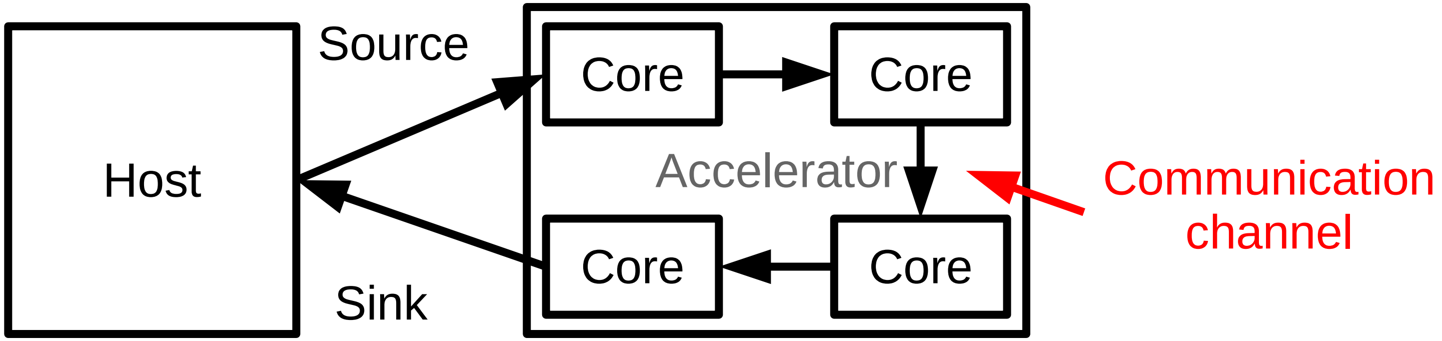
```
analyze    :: CoreZ ComplexSamples ComplexSamples ()
shiftPitch :: CoreZ ComplexSamples ComplexSampleStore ()

type ComplexSamples     = DPull (Complex Float)
type ComplexSampleStore = Store ComplexSamples

(analyze `on` 5 |>>halfChanSize>>| shiftPitch `on` 4)
  :: MulticoreZ ComplexSampleStore
                ComplexSampleStore ()
```

What about: `(analyze >>> shiftPitch)` ?

Or `(shiftPitch >>> analyze)` ?

Running multi-core Zeldspar

```
runZ :: ...
     => MulticoreZ inp out a          -- Multicore pipeline
     -> (Host (inp, Data Bool))       -- Source
     -> SizeSpec inp                  -- Source channel size
     -> (out -> Host (Data Bool))     -- Sink
     -> SizeSpec out                  -- Sink channel size
     -> Multicore ()
```



Channels

```
type SizeSpec a :: *
newChan :: CoreId -> CoreId
        -> SizeSpec a
        -> Multicore (CoreChan a)

hostId :: CoreId

type Slot a :: *
readChan  :: CoreChan a
          -> Slot a -> m (Data Bool)

writeChan :: CoreChan a
          -> a -> m (Data Bool)

closeChan :: CoreChan a -> m ()
```

CoreComp, Host, Multicore

```
CoreComp: ProgramT CoreCMD … Comp a
CoreCMD = CoreChanCMD
      :+: CoreHaltCMD
      :+: BulkArrCMD LocalArr
      :+: BulkArrCMD SharedArr
```

```
Comp → ProgramT CompCMD … Id a
CompCMD = ControlCMD
      :+: RefCMD
      :+: ArrCMD
```

CoreComp, Host, Multicore

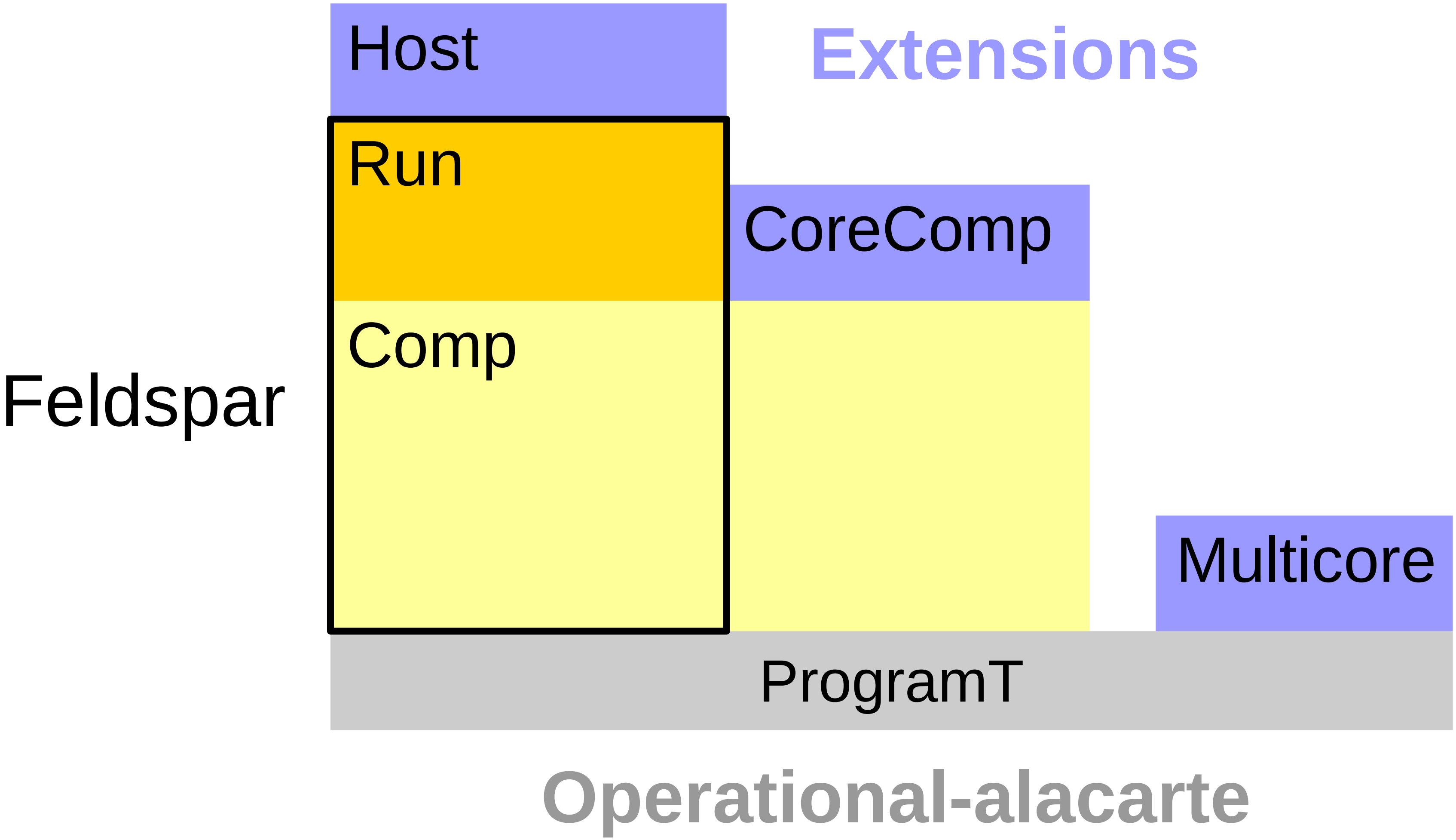
```
Host: ProgramT HostCMD … Run a
HostCMD = CoreChanCMD
      :+: CoreHaltCMD
      :+: BulkArrCMD LocalArr
      :+: BulkArrCMD SharedArr
      :+: MulticoreCMD
```

```
Run → ProgramT RunCMD … Comp a
RunCMD = FileCMD
     :+: ThreadCMD :+: ChanCMD
     :+: PtrCMD    :+: C_CMD
```

CoreComp, Host, Multicore

```
Multicore: ProgramT
    (AllocCMD :+: CoreChanAllocCMD) …

prog :: Multicore a
prog = do
  a <- alloc…
  onHost $ do
    (… :: Host a)
    onCore N $ (do … :: CoreComp a)
```

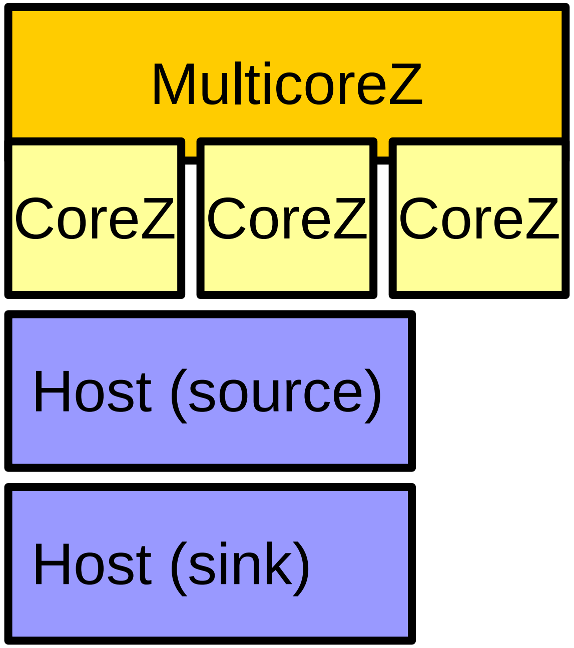
Monad stack


Host
Extensions
Run
CoreComp
Comp
Feldspar
Multicore
ProgramT
Operational-alacarte

Interpretation & compilation

Multicore

```
… <- alloc…
onHost $ …
runZ
```



Multicore

```
… <- alloc…
… <- newChan…
…
onHost $
   …
   onCore $ …
   onCore $ …
   onCore $ …
```

Interpretation & compilation

```
instance MonadRun Multicore
  where
    liftRun :: Multicore a -> Run a
```

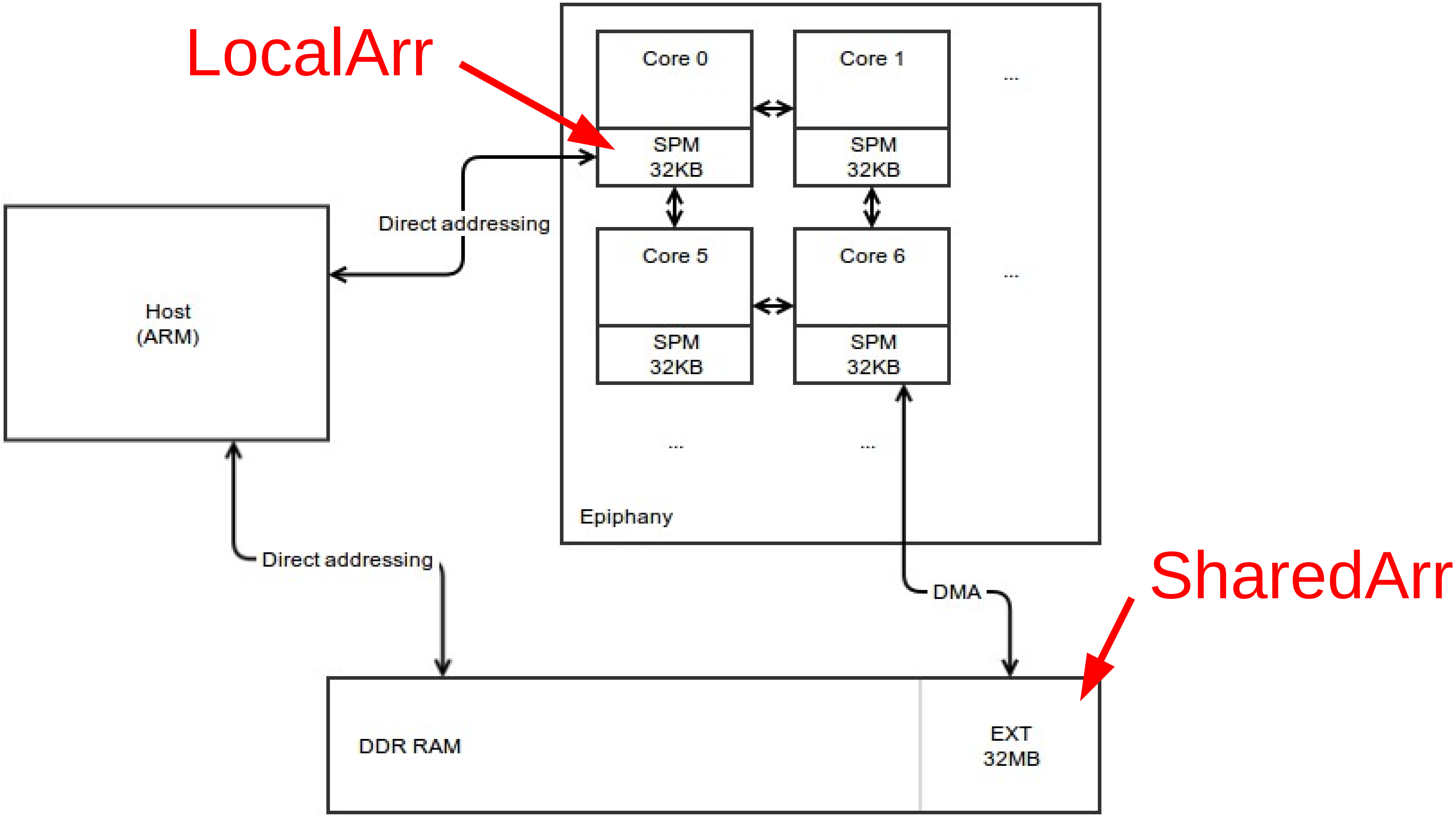
Default interpretation over Feldspar with threads:

```
compileAll prog  (→ runIO prog)
```

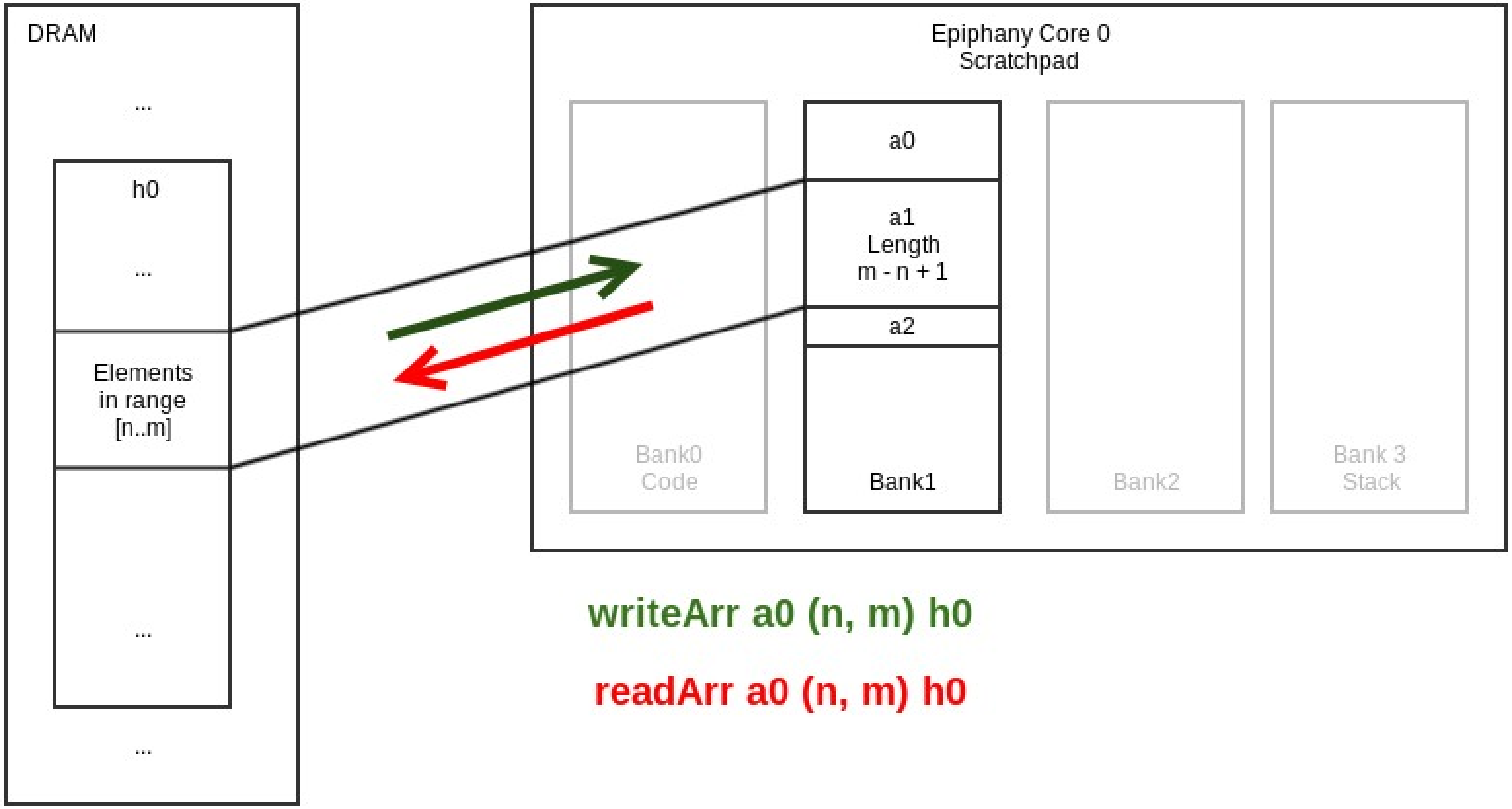
Compilation to Parallella with Epiphany-specific SDK calls:

```
compileAll `onParallella` prog
```

Memory architecture



Array operations



writeArr a0 (n, m) h0

readArr a0 (n, m) h0

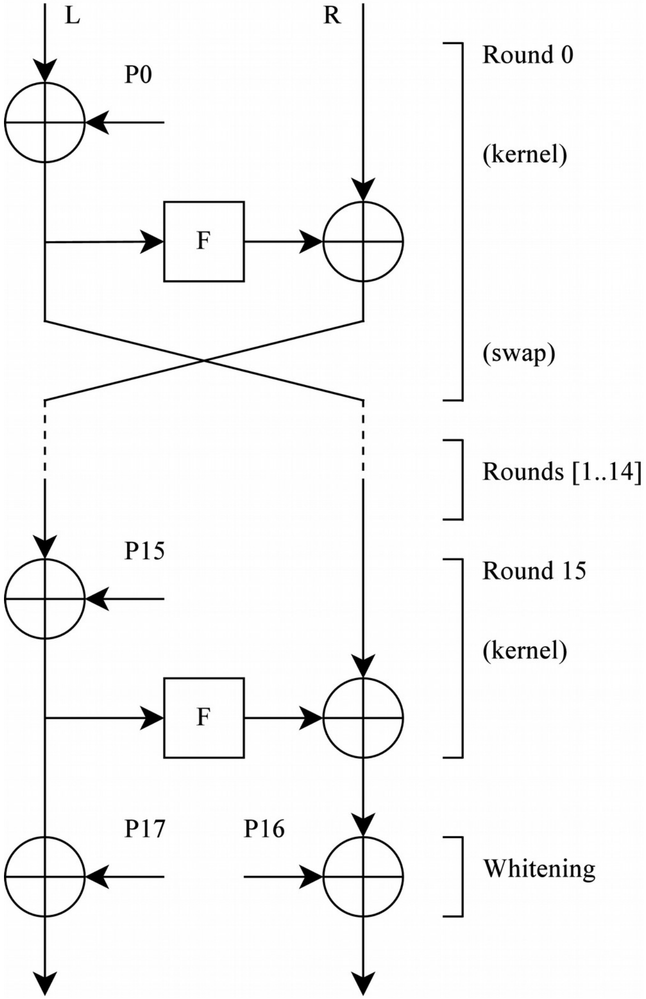
Shared and local arrays

```
prog :: Multicore ()
prog = do
  la :: LocalArr Bool <- allocLArr coreId len
  sa :: SharedArr Int32 <- allocSArr len
  onHost $ do
    writeArr la …
    readArr sa …
    onCore coreId $ do
      writeArr sa …
      a <- local la
      first <- getArr 0 a
      …
```

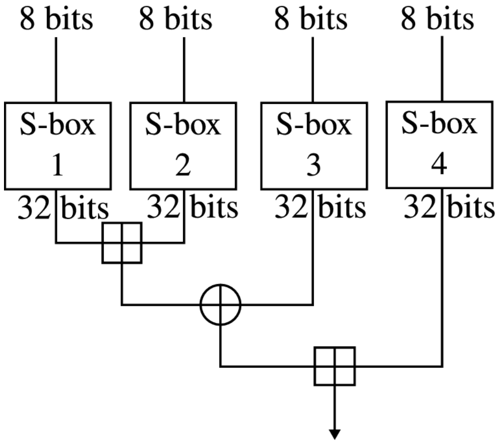
Shared and local arrays + Zeldspar

```
prog :: Multicore ()
prog = do
  la :: LocalArr Bool <- allocLArr coreId len
  sa :: SharedArr Int32 <- allocSArr len
  onHost $ do
    writeArr la …
    readArr sa …
  runZ (p1 la `on` 0 |>>>| p2 sa on `1`) …

p1 :: LocalArr Bool -> CoreZ …
p2 :: SharedArr Int32 -> CoreZ …
```

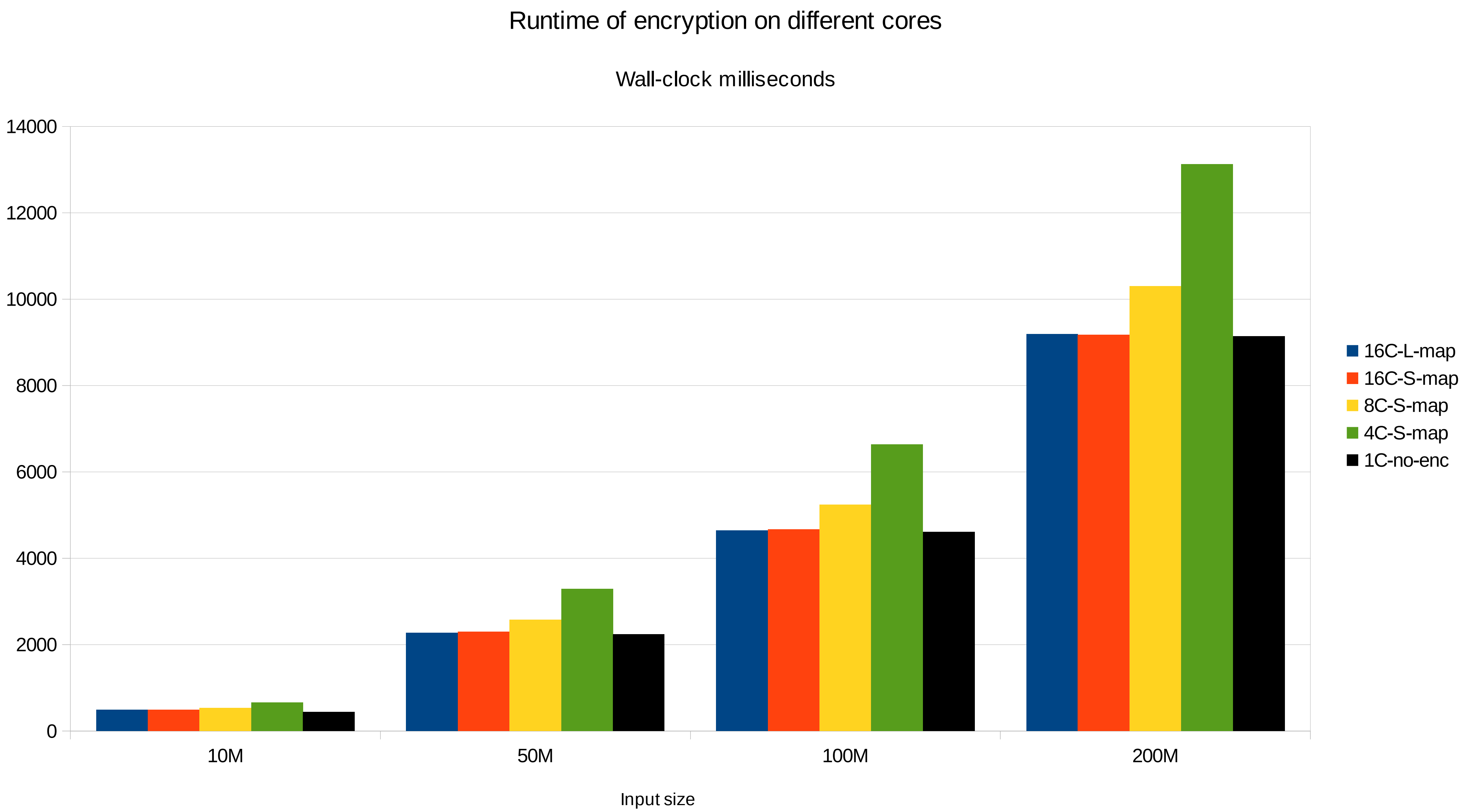
Action!



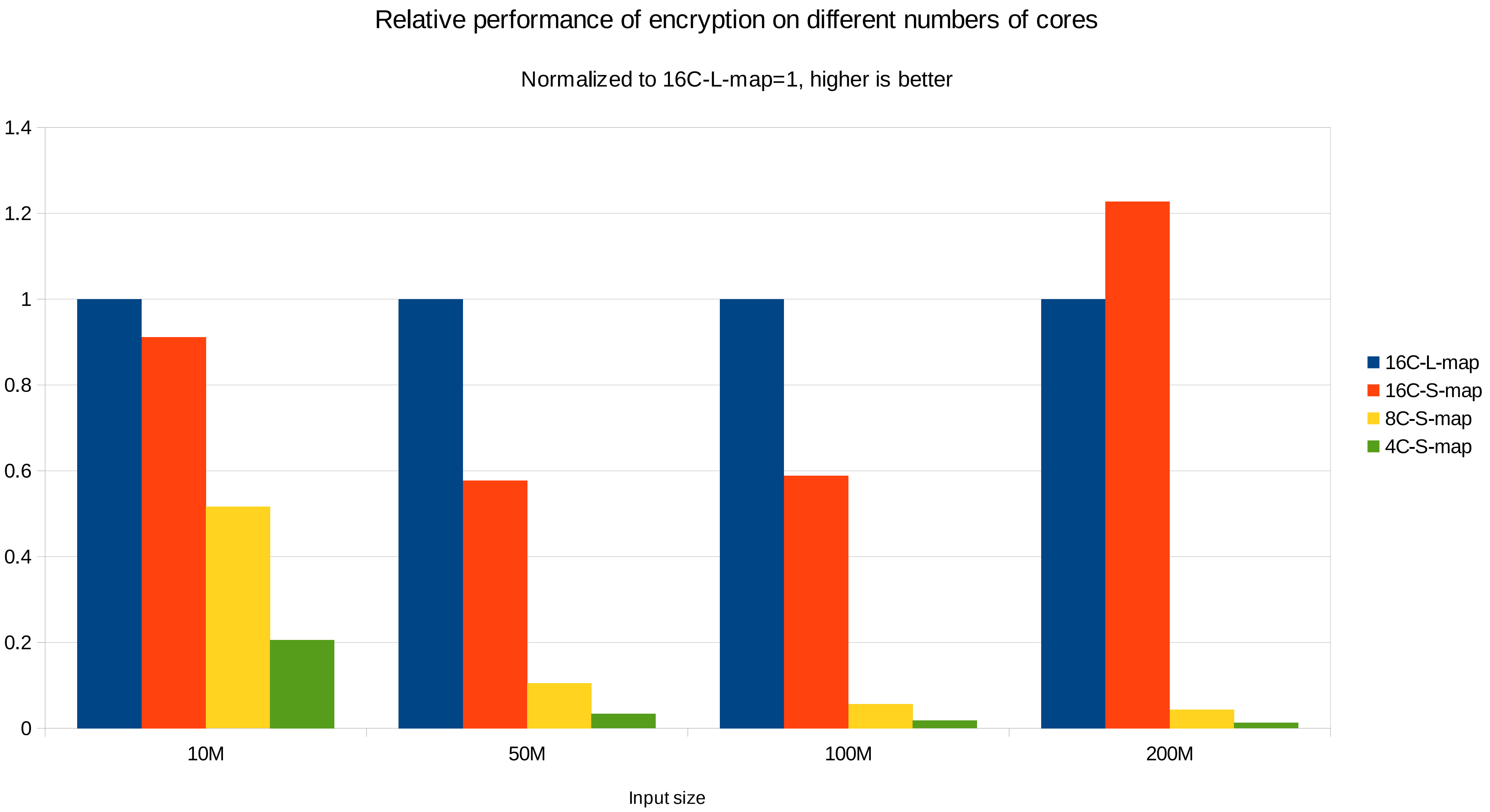
Encryption with Blowfish stream cipher



Encryption runtime measurements



Encryption runtime measurements



github.com/kmate/

raw-feldspar-mcs

parallella-dsp-demo

parallella-cipher-demo