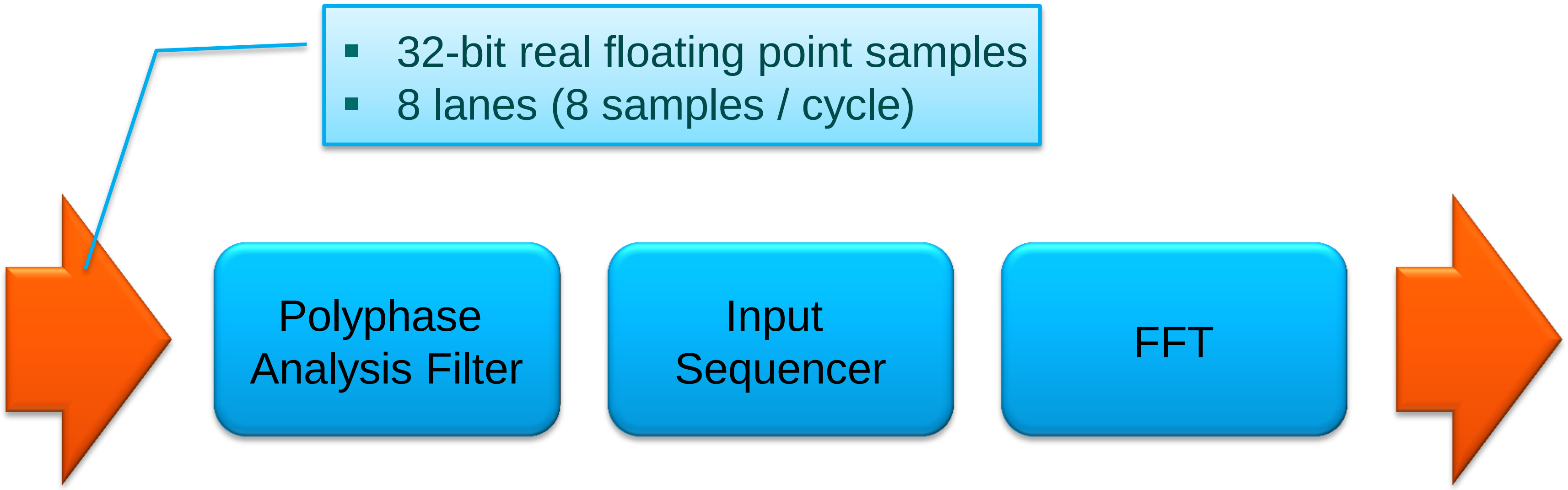
FPGA Channelizer Design in OpenCL

A Walkthrough of Tools, Concepts, and Results of an FPGA Channelizer Design Written in OpenCL

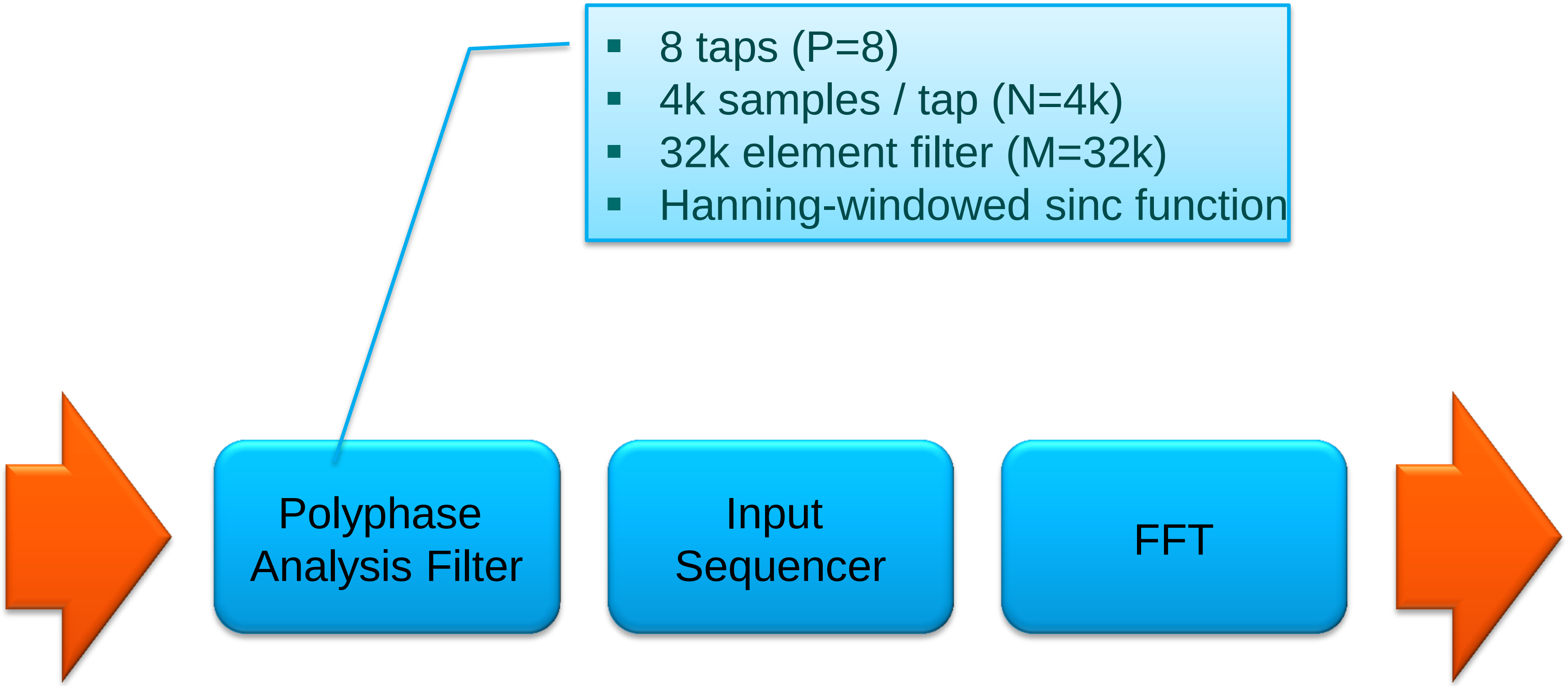
John Freeman
Supervisor, HLD Platforms Team
Altera Toronto Technology Center

High Level System Design

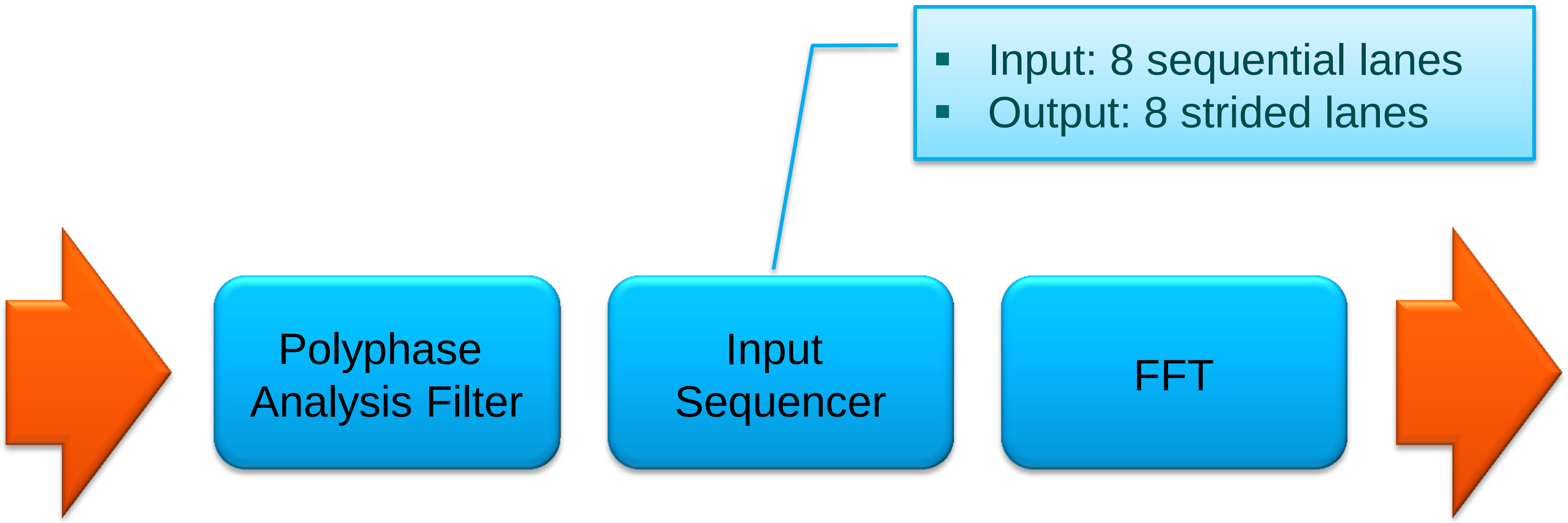


32-bit real floating point samples
8 lanes (8 samples / cycle)
Polyphase Analysis Filter
Input Sequencer
FFT

High Level System Design


- 8 taps (P=8)
- 4k samples / tap (N=4k)
- 32k element filter (M=32k)
- Hanning-windowed sinc function

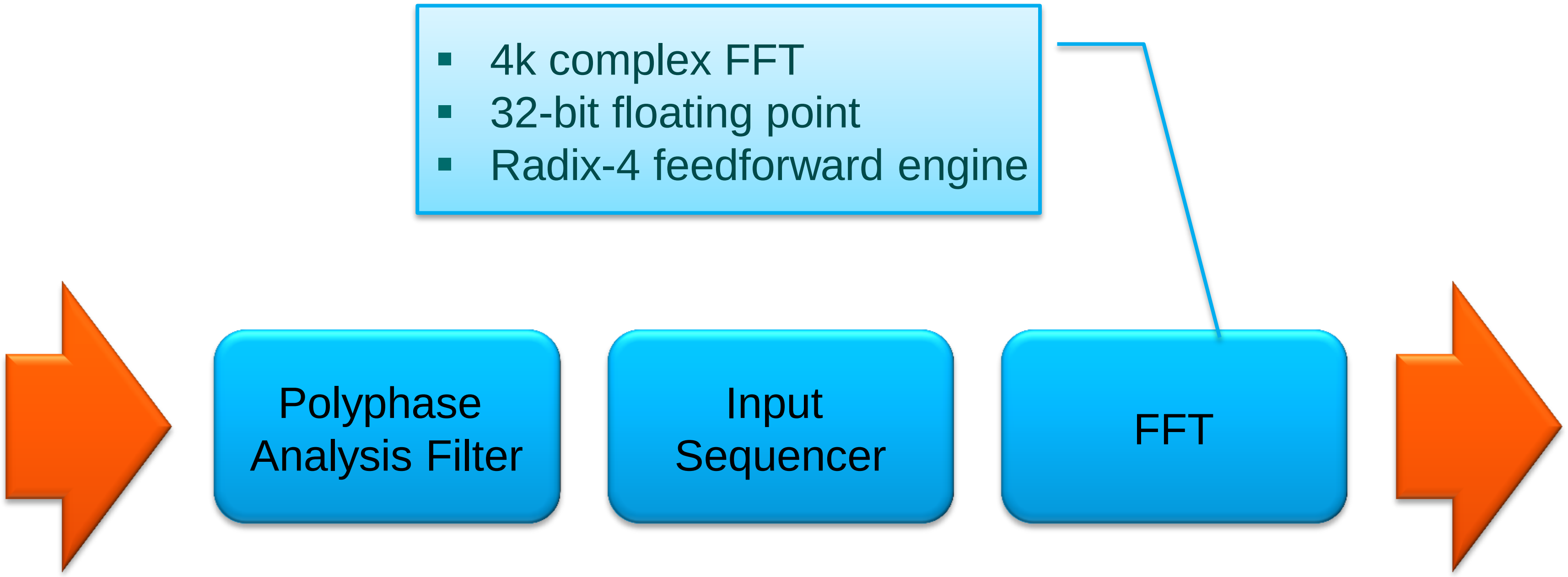
High Level System Design



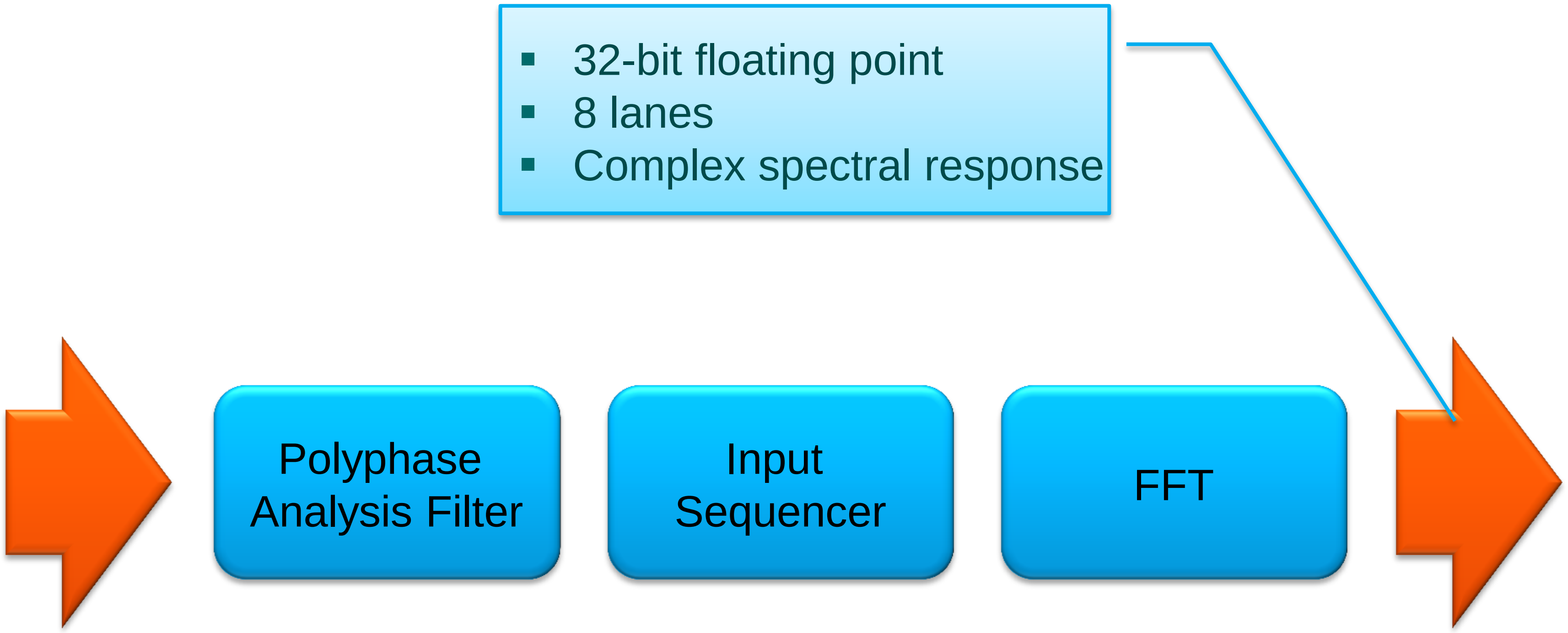
- Input: 8 sequential lanes
- Output: 8 strided lanes

ALTERA® MEASURABLE ADVANTAGE™

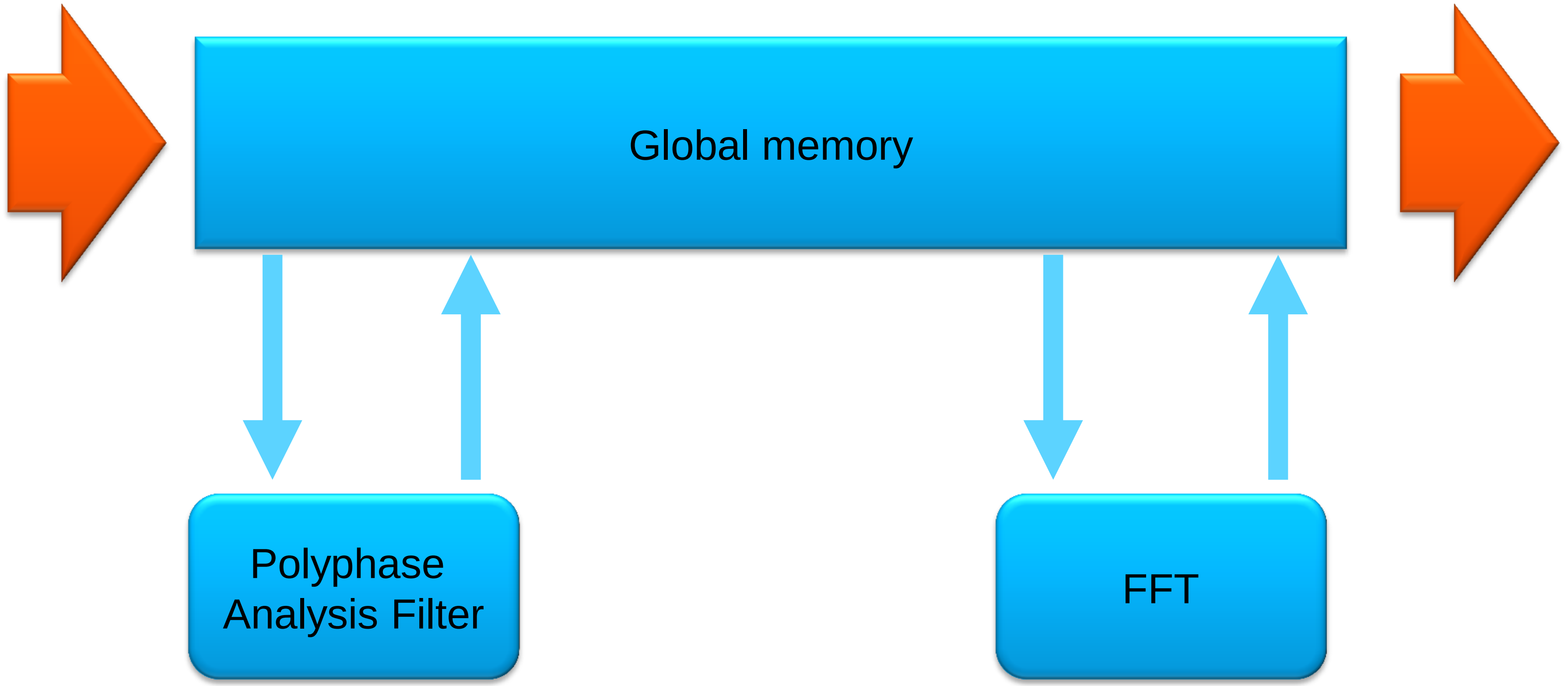
High Level System Design



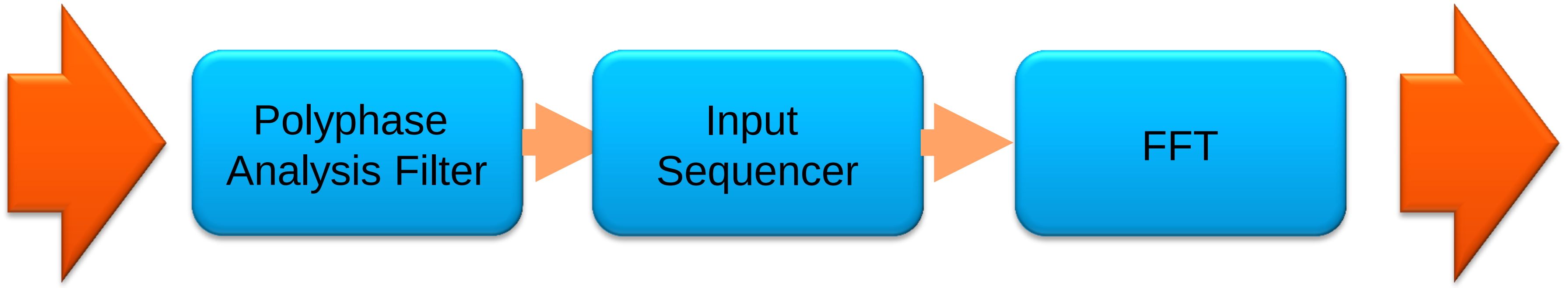
High Level System Design



Data Streaming with Channels



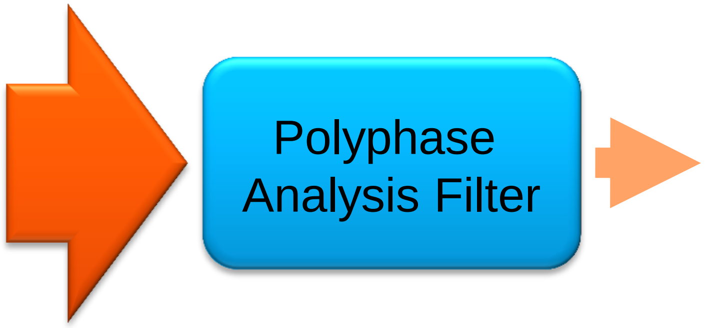
Data Streaming with Channels



Data Streaming with Channels

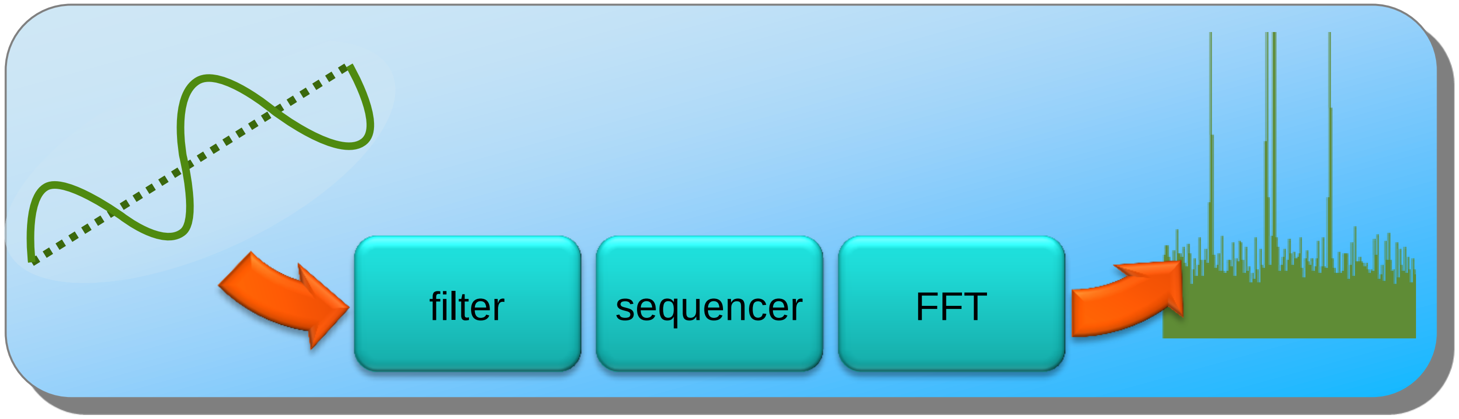
```
channel float8 DATA_IN __attribute(( io( “data_in” ) ));
channel float8 ANALYZER_OUT __attribute(( depth( 8 ) ));

kernel void filter( ) {
  read_channel_altera( DATA_IN );
  // ...
  write_channel_altera( ANALYZER_OUT );
}
```

- **I/O channels stream to/from Avalon-ST (data/valid/stall)**
- **Depth attribute creates rate balancing FIFOs**



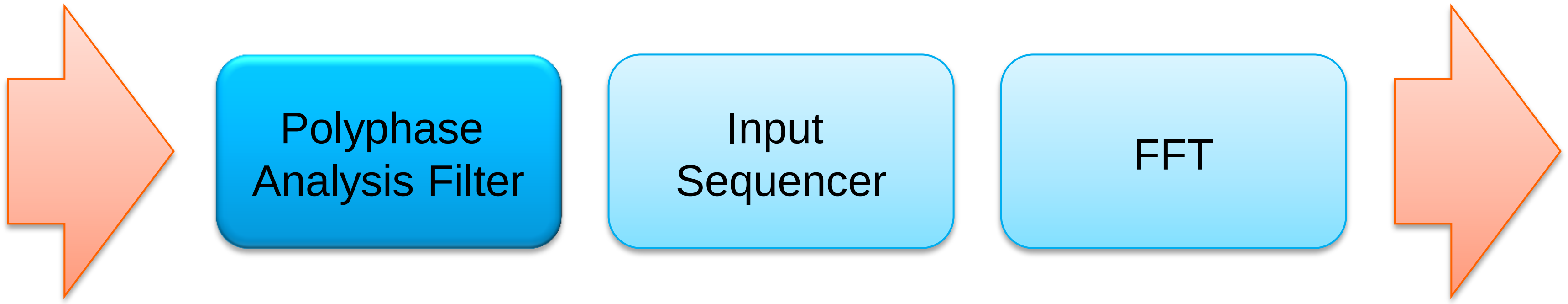
Results Summary

	Used	Available (SV-A7)	Utilization
RAM (M20k)	870	2,560	34.0%
Logic (ALMs)	85,876	234,720	36.6%
DSPs (27x27)	184	256	71.9%
FMAX (MHz)	279.4	--	--
Throughput (samples/s)	2.23 billion	--	--



Polyphase Filter Design

	Used	Available (SV-A7)	Utilization
RAM (M20k)	542	2,560	21.0%
Logic (ALMs)	27,432	234,720	12%
DSPs (27x27)	64	256	25%
FMAX (MHz)	321	--	--



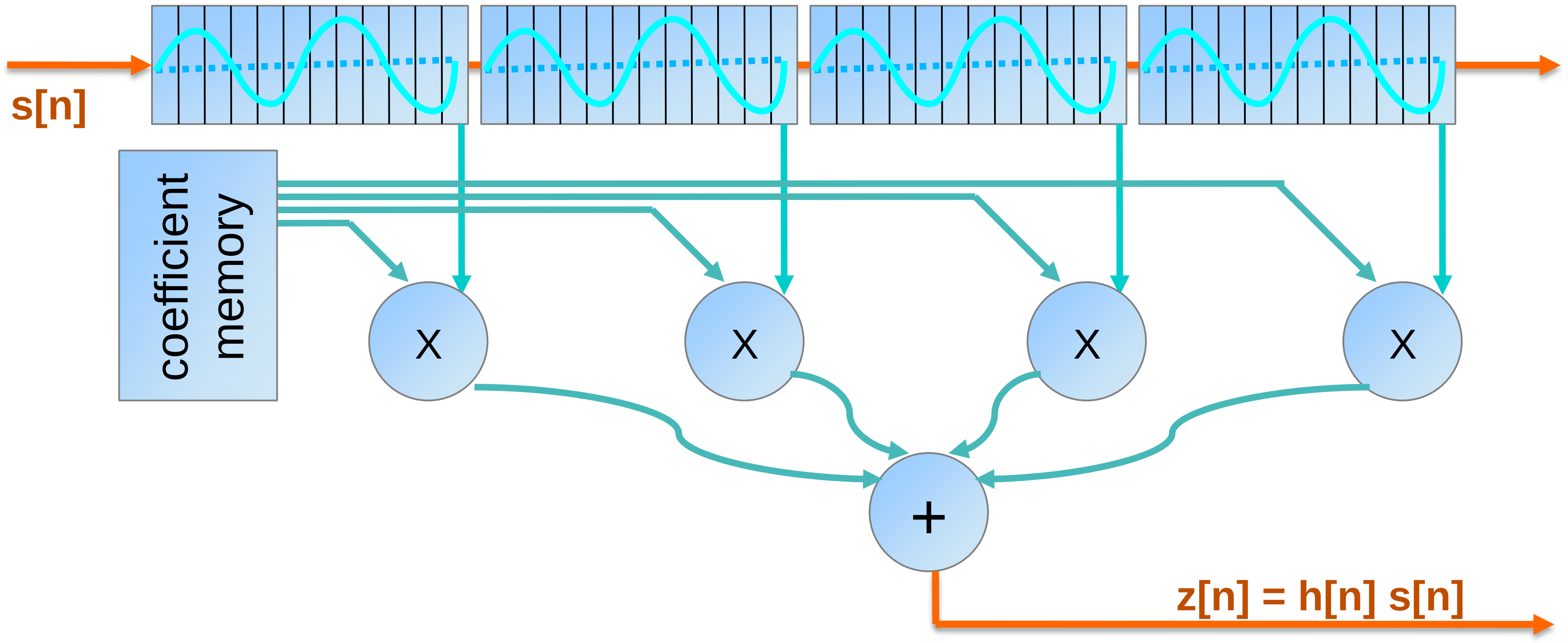
Polyphase Analysis Filter

- **Based on an implementation in**
  - *"A Mathematical Review of Polyphase Filterbank Implementations for Radio Astronomy",* C. Harris and K. Haines
- **Filter: Hanning-windowed sinc function**
- **Filter Parameters:**

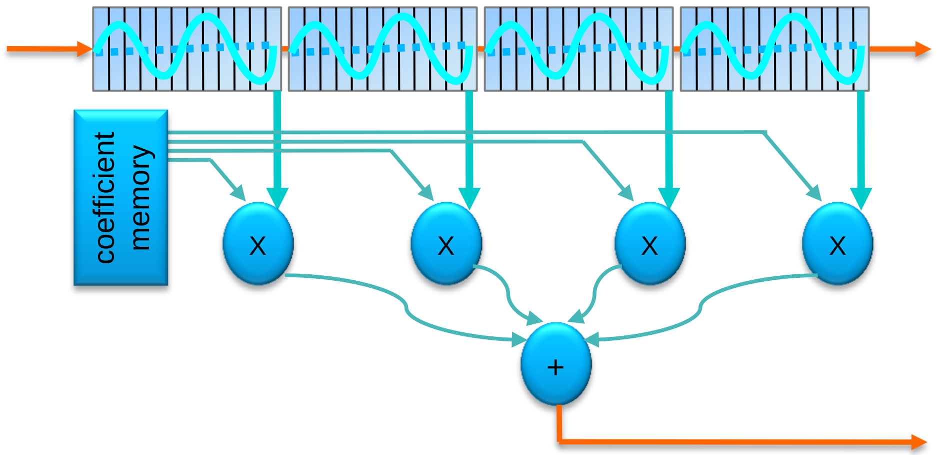
Number of taps (P)	8
Length (N)	4,096
Total samples (M=N*P)	32,768
Sample rate	8 samples / cycle

Filter – Desired Architecture

- ***Showing 4 taps (P=4) for brevity***



Filter – Deep Dive



```
// Filter coefficients
#include "pfb_coefs.h"

kernel
void fir(void) {
  for (int i=0; i<M/LANES; i++) {
    // Fetch the incoming samples
    // Fetch the coefficients
    // Push the data into a shift register
    // Tap data, multiply, and sum
    // Send the result to the next stage
  }
}
```

MEASURABLE ADVANTAGE™

Filter – Single-Threaded Kernel

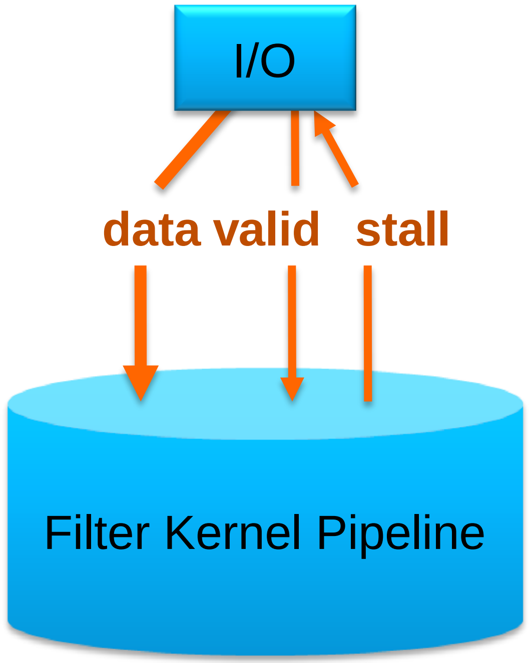
```
kernel
void fir(void) {
  for (int i=0; i<M/LANES; i++) {
```

- Single work-item
- Auto-parallelism
- Easily handle data dependencies
- Shift-register inference
- Ideal for channels

```
// Filter coefficients
#include "pfb_coefs.h"

kernel
void fir(void) {
  for (int i=0; i<M/LANES; i++) {
    // Fetch the incoming samples
    // Fetch the coefficients
    // Push the data into a shift register
    // Tap data, multiply, and sum
    // Send the result to the next stage
  }
}
```

Filter – I/O Channel

```
// Fetch the incoming samples
float8 input =
  read_channel_altera( DATA_IN );
```



```
// Filter coefficients
#include "pfb_coefs.h"

kernel
void fir(void) {
  for (int i=0; i<M/LANES; i++) {
    // Fetch the incoming samples
    // Fetch the coefficients
    // Push the data into a shift register
    // Tap data, multiply, and sum
    // Send the result to the next stage
  }
}
```

Filter – Constant Tables

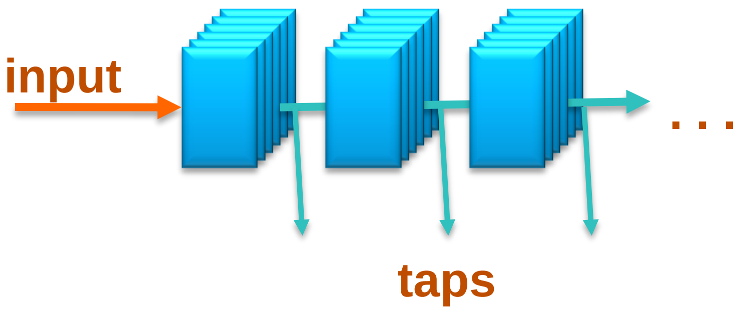
```
// Fetch the coefficients
#pragma unroll P
for (int j=0; j<P; j++) {
  coefs[ 0 ][ j ] = coefs_pt0[ idx*P+j ];
  coefs[ 7 ][ j ] = coefs_pt7[ idx*P+j ];
}
```

```
// Filter coefficients
#include "pfb_coefs.h"

kernel
void fir(void) {
  for (int i=0; i<M/LANES; i++) {
    // Fetch the incoming samples
    // Fetch the coefficients
    // Push the data into a shift register
    // Tap data, multiply, and sum
    // Send the result to the next stage
  }
}
```

Filter – Shift Register Inference

```
// Push the data into a shift register
#pragma unroll
for (int j=0; j<(P-1)*N; j++) {
  taps[ j ] = taps[ j+8 ];
}
#pragma unroll
for (int j=0; j<LANES; j++) {
  taps[ (P-1)*N+j ] = input[ j ];
}
```

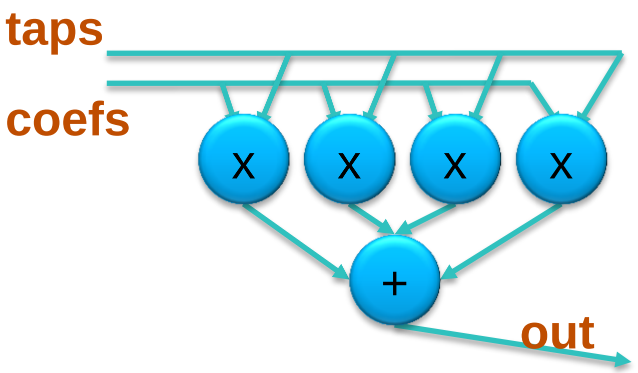
```
// Filter coefficients
#include "pfb_coefs.h"

kernel
void fir(void) {
  for (int i=0; i<M/LANES; i++) {
    // Fetch the incoming samples
    // Fetch the coefficients
    // Push the data into a shift register
    // Tap data, multiply, and sum
    // Send the result to the next stage
  }
}
```



Filter – Multiply/Add

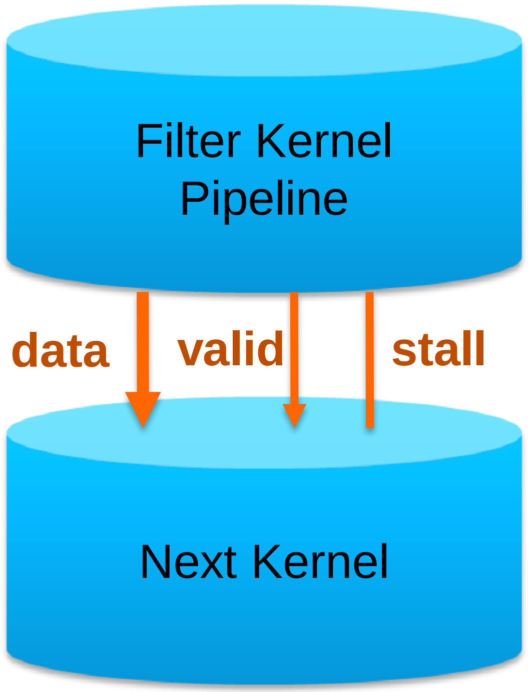
```
// Tap, multiply, add
#pragma unroll
for (int j=0; j<LANES; j++) {
  out[ j ]=0.0f;
  #pragma unroll
  for (int k=0; k<P; k++) {
    out[ j ] += coefs[ j ][ k ] * taps[ j+k*N ];
  }
}
```

```
// Filter coefficients
#include "pfb_coefs.h"

kernel
void fir(void) {
  for (int i=0; i<M/LANES; i++) {
    // Fetch the incoming samples
    // Fetch the coefficients
    // Push the data into a shift register
    // Tap data, multiply, and sum
    // Send the result to the next stage
  }
}
```



Filter – Output

```
// Send the result to the next stage
write_channel_altera(FIR_OUT, out);
```



```
// Filter coefficients
#include "pfb_coefs.h"

kernel
void fir(void) {
  for (int i=0; i<M/LANES; i++) {
    // Fetch the incoming samples
    // Fetch the coefficients
    // Push the data into a shift register
    // Tap data, multiply, and sum
    // Send the result to the next stage
  }
}
```

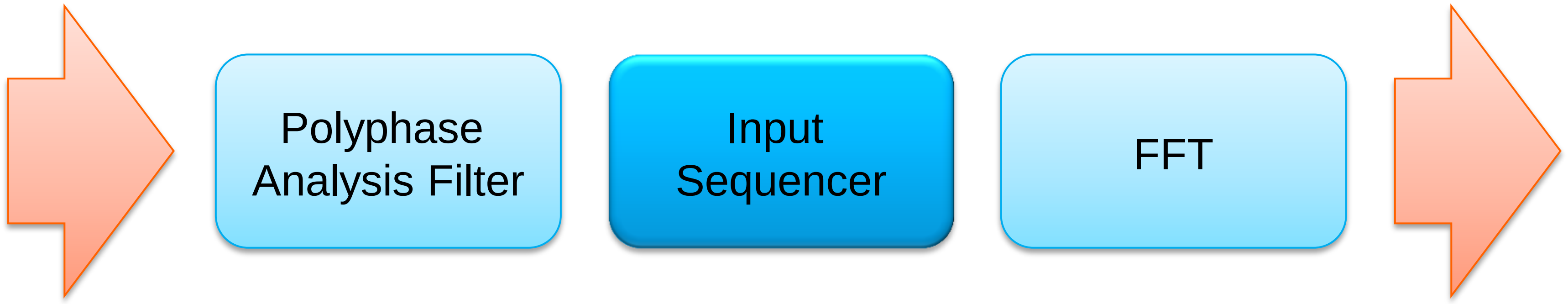
Filter – Full Code Listing

```
// Filter coefficients
#include "pfb_coefs.h"

kernel
void fir(void) {
  float taps[(P-1)*N+8];
  uint idx = 0;
  for (int i=0; i<M/LANES; i++) {
    // Fetch the input
    float8 in = read_channel_altera(DATA_IN);
    float coefs[LANES][P];
    float8 out;
    // Fetch the coefficients
    #pragma unroll
    for (int j=0; j<P; j++) {
      coefs[0][j] = coefs0[idx*P+j];
      // ...
      coefs[7][j] = coefs7[idx*P+j];
    }
    idx = (coef_idx+1)&(N/LANES-1);
```

```
    // Push the data into a shift register
    #pragma unroll
    for (int j=0; j<(P-1)*N; j++)
      taps[j] = taps[j+8];
    #pragma unroll
    for (int j=0; j<LANES; j++)
      taps[(P-1)*N+j] = in[j];
    // Multiply and add
    #pragma unroll
    for (int j=0; j<LANES; j++) {
      out[j]=0.0f;
      #pragma unroll
      for (int k=0; k<P; k++)
        out[j] += coefs[j][k] * taps[j+k*N];
    }
    // Send results to the next stage
    write_channel_altera(FIR_OUT, out);
  }
}
```

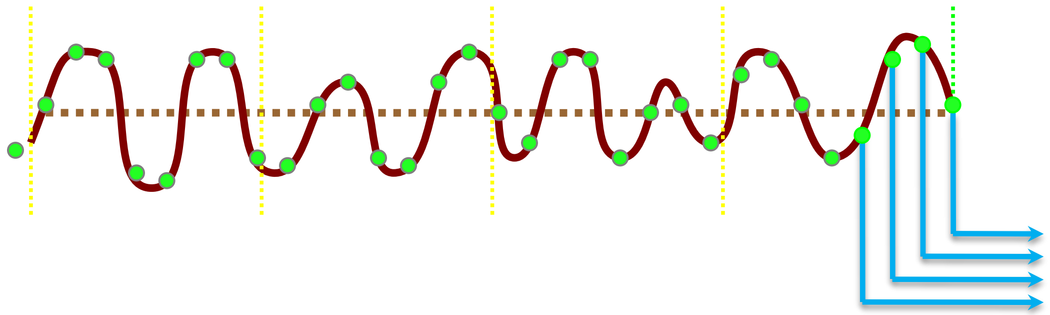
Channel Sequencer Design

	Used	Available (SV-A7)	Utilization
RAM (M20k)	87	2,560	3.3%
Logic (ALMs)	4,700	234,720	2.0%
DSPs (27x27)	0	256	0.0%
FMAX (MHz)	374	--	--

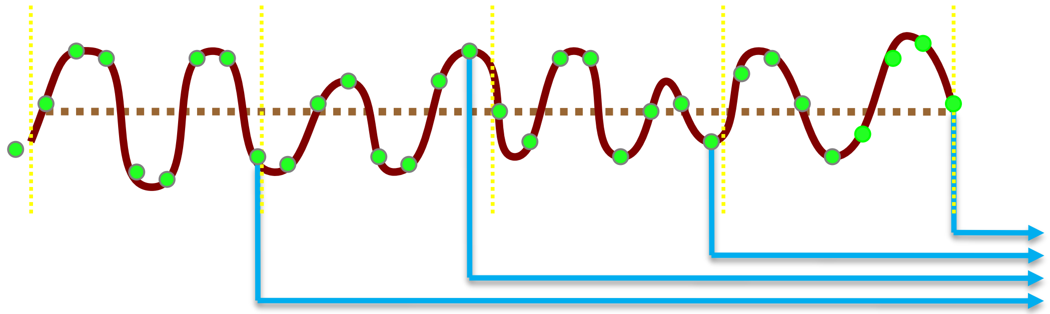


Channel Sequencer

- Sampled data arrives sequentially



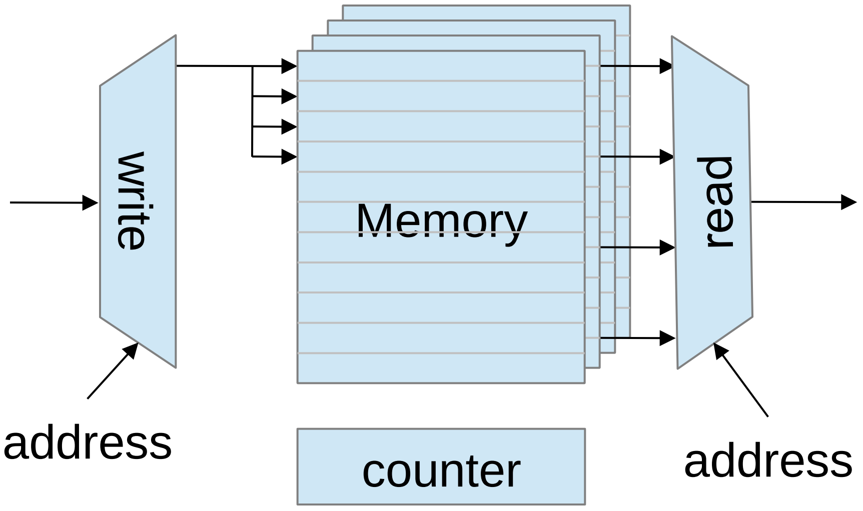
- FFT expects strided data



Channel Sequencer Deep Dive

```
__attribute((reqd_work_group_size((1<< (LOGN-3)),1,1)))
kernel void sequence() {
  // local memory for storing one window
  local float8 buf8[ N ];
  local float *buf = (local float *)buf8;
  float8 data;
  int lid = get_local_id(0);

  // Read 8 points into the local buffer
  buf8[ lid ] = read_channel_altera( FIR_OUT );
  barrier( CLK_LOCAL_MEM_FENCE );

  // Stream fetched data over 8 channels to the FFT engine
  data.s0 = buf[ 0 * N/8 + lid ];
  data.s1 = buf[ 4 * N/8 + lid ];
  data.s2 = buf[ 2 * N/8 + lid ];
  data.s3 = buf[ 6 * N/8 + lid ];
  data.s4 = buf[ 1 * N/8 + lid ];
  data.s5 = buf[ 5 * N/8 + lid ];
  data.s6 = buf[ 3 * N/8 + lid ];
  data.s7 = buf[ 7 * N/8 + lid ];

  // Send strided data off to the FFT
  write_channel_altera(SEQ_OUT, data);
}
```

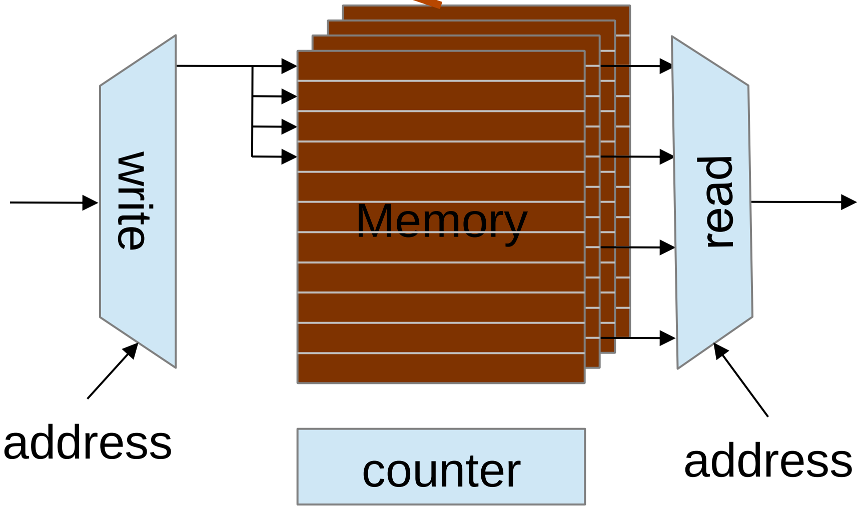
- **Local memory is automatically optimized by the compiler**
  - Required workgroup size implies sizing
  - Auto-duplicate buffers based on pipeline depth
  - Fine grained banking to eliminate contention



Channel Sequencer – Local Memory

```
__attribute((reqd_work_group_size((1<< (LOGN-3)),1,1)))
kernel void sequence() {
  // local memory for storing one window
  local float8 buf8[ N ];
  local float *buf = (local float *)buf8;
  float8 data;
  int lid = get_local_id(0);

  // Read 8 points into the local buffer
  buf8[ lid ] = read_channel_altera( FIR_OUT );
  barrier( CLK_LOCAL_MEM_FENCE );

  // Stream fetched data over 8 channels to the FFT engine
  data.s0 = buf[ 0 * N/8 + lid ];
  data.s1 = buf[ 4 * N/8 + lid ];
  data.s2 = buf[ 2 * N/8 + lid ];
  data.s3 = buf[ 6 * N/8 + lid ];
  data.s4 = buf[ 1 * N/8 + lid ];
  data.s5 = buf[ 5 * N/8 + lid ];
  data.s6 = buf[ 3 * N/8 + lid ];
  data.s7 = buf[ 7 * N/8 + lid ];

  // Send strided data off to the FFT
  write_channel_altera(SEQ_OUT, data);
}
```

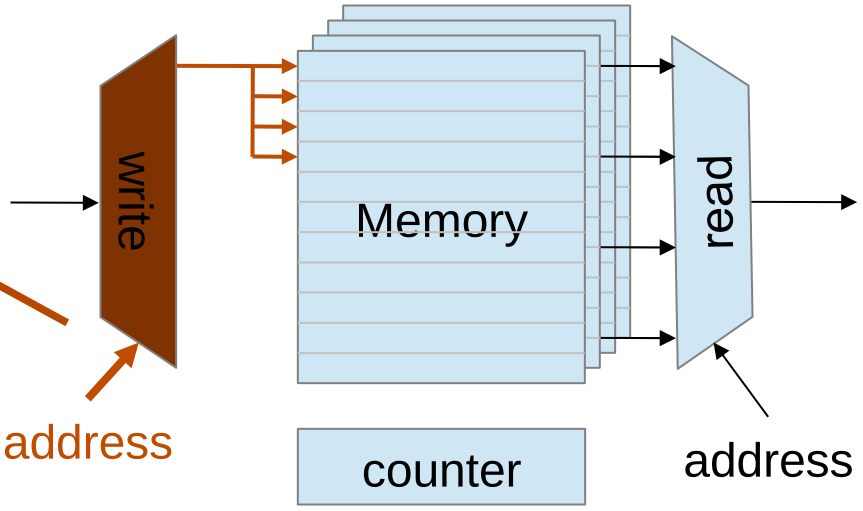
- **Local memory is automatically optimized by the compiler**
  - Required workgroup size implies sizing
  - Auto-duplicate buffers based on pipeline depth
  - Fine grained banking to eliminate contention



Channel Sequencer – Local Writes

```
__attribute((reqd_work_group_size((1<< (LOGN-3)),1,1)))
kernel void sequence() {
  // local memory for storing one window
  local float8 buf8[ N ];
  local float *buf = (local float *)buf8;
  float8 data;
  int lid = get_local_id(0);

  // Read 8 points into the local buffer
  buf8[ lid ] = read_channel_altera( FIR_OUT );
  barrier( CLK_LOCAL_MEM_FENCE );

  // Stream fetched data over 8 channels to the FFT engine
  data.s0 = buf[ 0 * N/8 + lid ];
  data.s1 = buf[ 4 * N/8 + lid ];
  data.s2 = buf[ 2 * N/8 + lid ];
  data.s3 = buf[ 6 * N/8 + lid ];
  data.s4 = buf[ 1 * N/8 + lid ];
  data.s5 = buf[ 5 * N/8 + lid ];
  data.s6 = buf[ 3 * N/8 + lid ];
  data.s7 = buf[ 7 * N/8 + lid ];

  // Send strided data off to the FFT
  write_channel_altera(SEQ_OUT, data);
}
```

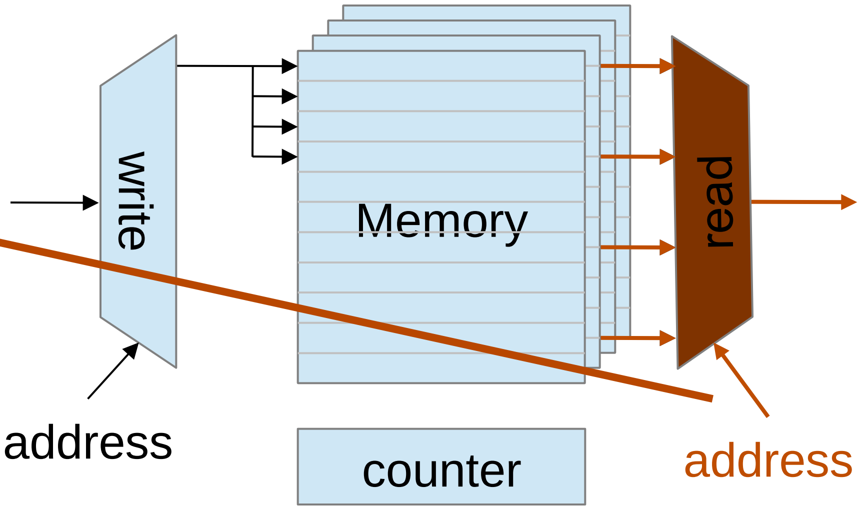
- **Local memory is automatically optimized by the compiler**
  - Required workgroup size implies sizing
  - Auto-duplicate buffers based on pipeline depth
  - Fine grained banking to eliminate contention



Channel Sequencer – Local Reads

```
__attribute((reqd_work_group_size((1<< (LOGN-3)),1,1)))
kernel void sequence() {
  // local memory for storing one window
  local float8 buf8[ N ];
  local float *buf = (local float *)buf8;
  float8 data;
  int lid = get_local_id(0);

  // Read 8 points into the local buffer
  buf8[ lid ] = read_channel_altera( FIR_OUT );
  barrier( CLK_LOCAL_MEM_FENCE );

  // Stream fetched data over 8 channels to the FFT engine
  data.s0 = buf[ 0 * N/8 + lid ];
  data.s1 = buf[ 4 * N/8 + lid ];
  data.s2 = buf[ 2 * N/8 + lid ];
  data.s3 = buf[ 6 * N/8 + lid ];
  data.s4 = buf[ 1 * N/8 + lid ];
  data.s5 = buf[ 5 * N/8 + lid ];
  data.s6 = buf[ 3 * N/8 + lid ];
  data.s7 = buf[ 7 * N/8 + lid ];

  // Send strided data off to the FFT
  write_channel_altera(SEQ_OUT, data);
}
```

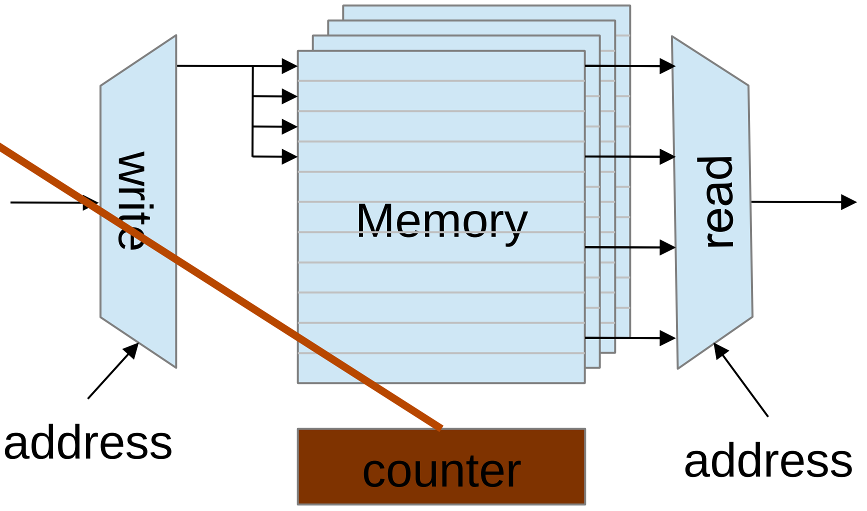
- **Local memory is automatically optimized by the compiler**
  - Required workgroup size implies sizing
  - Auto-duplicate buffers based on pipeline depth
  - Fine grained banking to eliminate contention



Channel Sequencer – Barriers

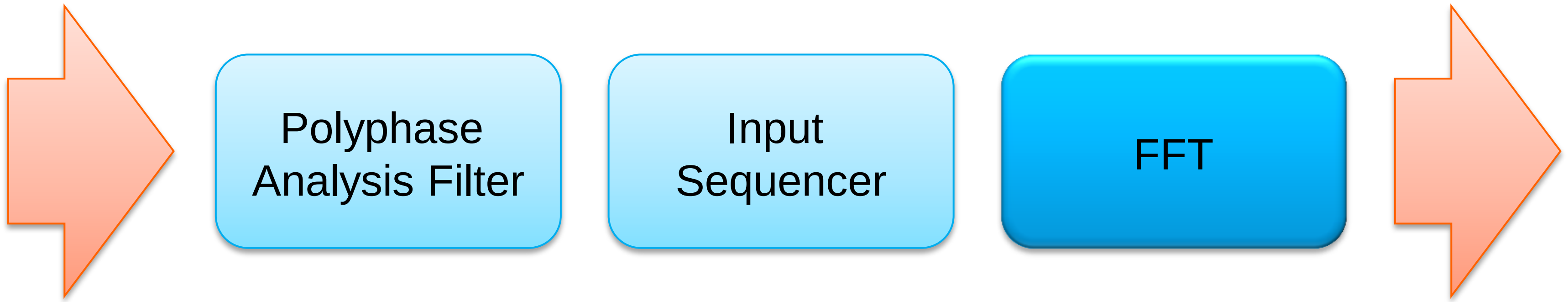
```
__attribute((reqd_work_group_size((1<< (LOGN-3)),1,1)))
kernel void sequence() {
  // local memory for storing one window
  local float8 buf8[ N ];
  local float *buf = (local float *)buf8;
  float8 data;
  int lid = get_local_id(0);

  // Read 8 points into the local buffer
  buf8[ lid ] = read_channel_altera( FIR_OUT );
  barrier( CLK_LOCAL_MEM_FENCE );

  // Stream fetched data over 8 channels to the FFT engine
  data.s0 = buf[ 0 * N/8 + lid ];
  data.s1 = buf[ 4 * N/8 + lid ];
  data.s2 = buf[ 2 * N/8 + lid ];
  data.s3 = buf[ 6 * N/8 + lid ];
  data.s4 = buf[ 1 * N/8 + lid ];
  data.s5 = buf[ 5 * N/8 + lid ];
  data.s6 = buf[ 3 * N/8 + lid ];
  data.s7 = buf[ 7 * N/8 + lid ];

  // Send strided data off to the FFT
  write_channel_altera(SEQ_OUT, data);
}
```

- **Local memory is automatically optimized by the compiler**
  - Required workgroup size implies sizing
  - Auto-duplicate buffers based on pipeline depth
  - Fine grained banking to eliminate contention



FFT Design

	Used	Available (SV-A7)	Utilization
RAM (M20k)	128	2,560	5.0%
Logic (ALMs)	66,701	234,720	28%
DSPs (27x27)	120	256	47%
FMAX (MHz)	299	--	--



OpenCL FFT

- **Used the 1D FFT design example 'as-is'**
  - http://www.altera.com/support/examples/opencl/opencl.html
  - See M. Garrido, J. Grajal, M. A. Sanchez, O. Gustafsson, *Pipelined Radix-2k Feedforward FFT Architectures.* IEEE Trans. VLSI Syst. 21(1): 23-32 (2013)
- **4k single-precision floating point 1D FFT**
  - Radix-4 feedforward pipeline design
- **Additional enhancements**
  - Use I/O channels instead of global memory
    - Please contact Altera if you are interested in using I/O channels
  - Application data arrives as 4k real samples
    - Tied all complex components off to 0 for this design
    - A more efficient implementation would take advantage of the symmetry from real inputs

OpenCL FFT

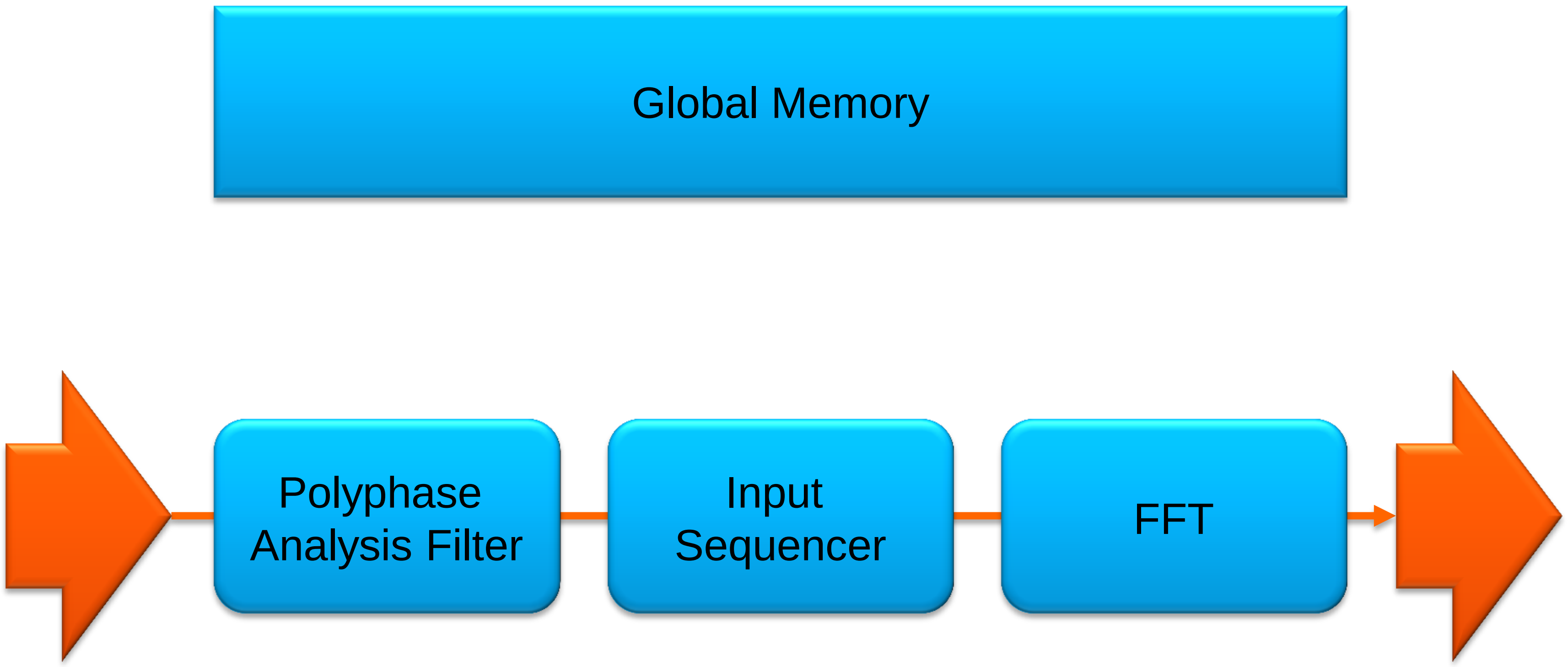
- **Builds on the core concepts already presented**
  - Shift register inference, constant tables, channels loop unrolling
- **C-based description enables easy, modular design**

```
float2x8 fft_step(float2x8 data, int step, float2 *delay, bool inverse, const int logN) {
    // Swap real and imaginary components if doing an inverse transform
    if (inverse)
       data = swap_complex(data);
    // Stage 0 of feed-forward FFT
    data = butterfly(data);
    data = trivial_rotate(data);
    data = trivial_swap(data);
    // Stage 1
    data = butterfly(data);
    data = complex_rotate(data, step & (N / 8 - 1), 1, N);
    data = swap(data);
```

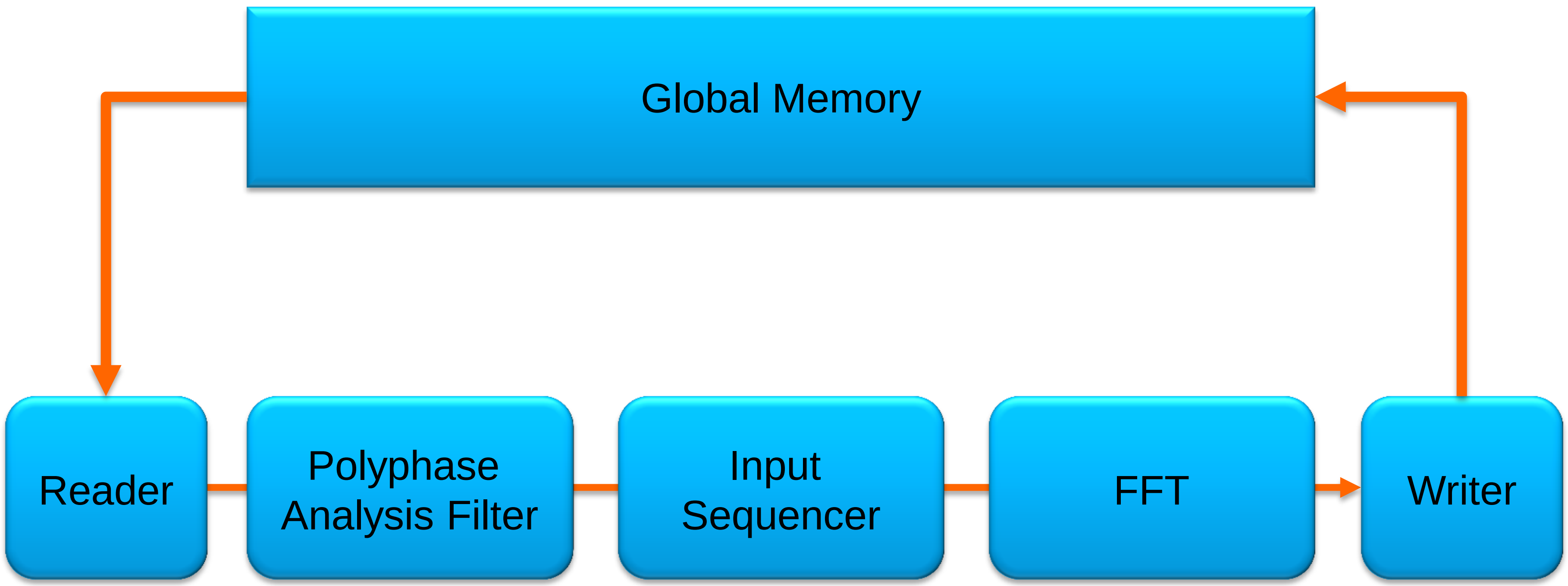
Larger FFTs

- **Eventually the FPGA runs out of room to store intermediate FFT results**
  - Necessitates storing data off-chip (e.g. to DDR)
  - Careful use of the available DDR bandwidth is likely necessary
    - Off-chip storage is easily described as a global memory pointer in Altera’s OpenCL SDK

Testing


Global Memory
Polyphase Analysis Filter
Input Sequencer
FFT

Testing

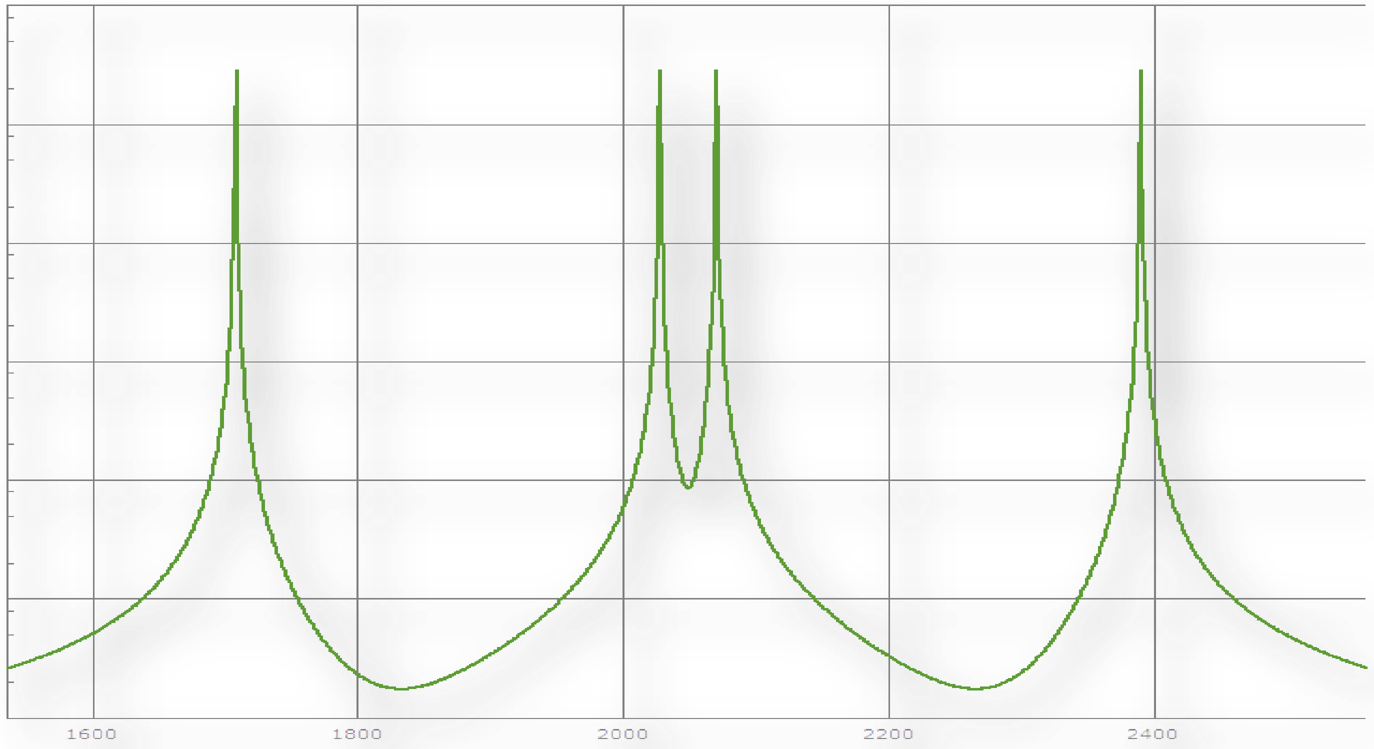

Global Memory
Reader
Polyphase Analysis Filter
Input Sequencer
FFT
Writer

Testing

```
kernel void
reader(global float16* data_in) {
  uint gid = get_global_id(0);
  float8 data = data_in[gid].s02468ace;
  write_channel_altera(DATA_IN, data);
}
```

```
kernel void
writer(global float16* data_out) {
  uint gid = get_global_id(0);
  data_out[gid] = read_channel_altera(DATA_OUT);
}
```

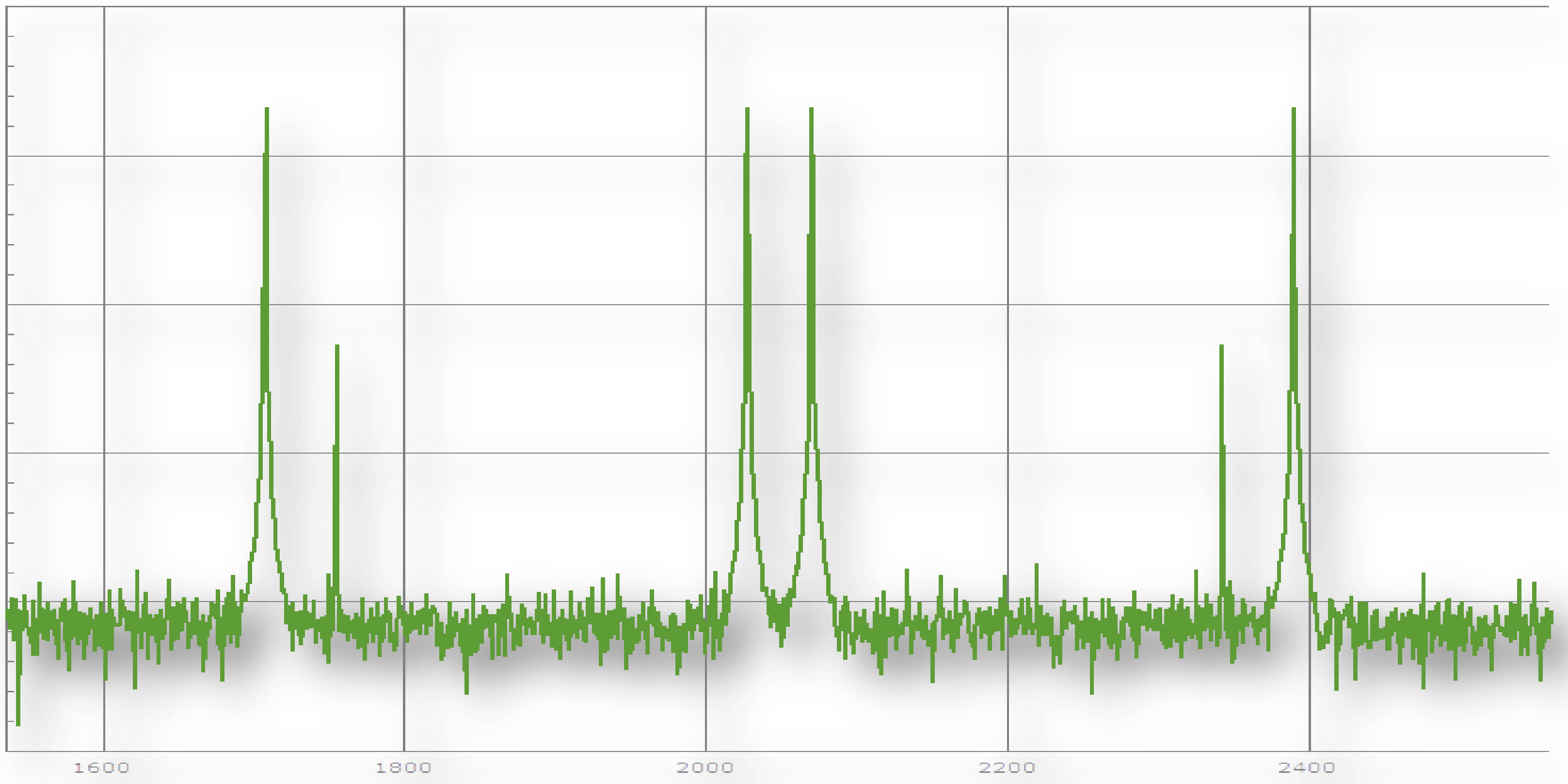
Results: FFT Only



$$\cos\left(\frac{128\times\pi}{3}\times\frac{n}{N}\right)+\cos\left(\frac{2048\times\pi}{3}\times\frac{n}{N}\right)+0.0001\times\cos\left(\frac{1765\times\pi}{3}\times\frac{n}{N}\right)$$

ALTERA® MEASURABLE ADVANTAGE™

Results: Analysis Filter



$$\cos\left(\frac{128\times\pi}{3}\times\frac{n}{N}\right)+\cos\left(\frac{2048\times\pi}{3}\times\frac{n}{N}\right)+0.0001\times\cos\left(\frac{1765\times\pi}{3}\times\frac{n}{N}\right)$$

Thank You