StreamIt

A Compiler Infrastructure for Stream Programs

William Thies

Joint work with Michael Gordon, Michal Karczmarek, Jasper Lin, Andrew Lamb, David Maze, Rodric Rabbah and Saman Amarasinghe

Massachusetts Institute of Technology

IBM PL Day

May 21, 2004

Streaming Application Domain

- Based on audio, video, or data stream
- Increasingly prevalent and important
  - Embedded systems
    - Cell phones, handheld computers
  - Desktop applications
    - Streaming media
    - Software radio
    - Real-time encryption
    - Graphics packages
  - High-performance servers
    - Software routers (Example: Click)
    - Cell phone base stations
    - HDTV editing consoles

Properties of Stream Programs

- A large (possibly infinite) amount of data
  - Limited lifetime of each data item
  - Little processing of each data item
- Computation: apply multiple filters to data
  - Each filter takes an input stream, does some processing, and produces an output stream
  - Filters are independent and self-contained
- A regular, static communication pattern
  - Filter graph is relatively constant
  - A lot of opportunities for compiler optimizations

The StreamIt Project

- Goals:
  - Provide a high-level stream programming model
  - Invent new compiler technology for streams
- Contributions:
  - Language Design, Structured Streams, Buffer Management *(CC 2002)*
  - Exploiting Wire-Exposed Architectures *(ASPLOS 2002, ISCA 2004)*
  - Scheduling of Static Dataflow Graphs *(LCTES 2003)*
  - Domain Specific Optimizations *(PLDI 2003)*
  - Public release: Fall 2003

Outline

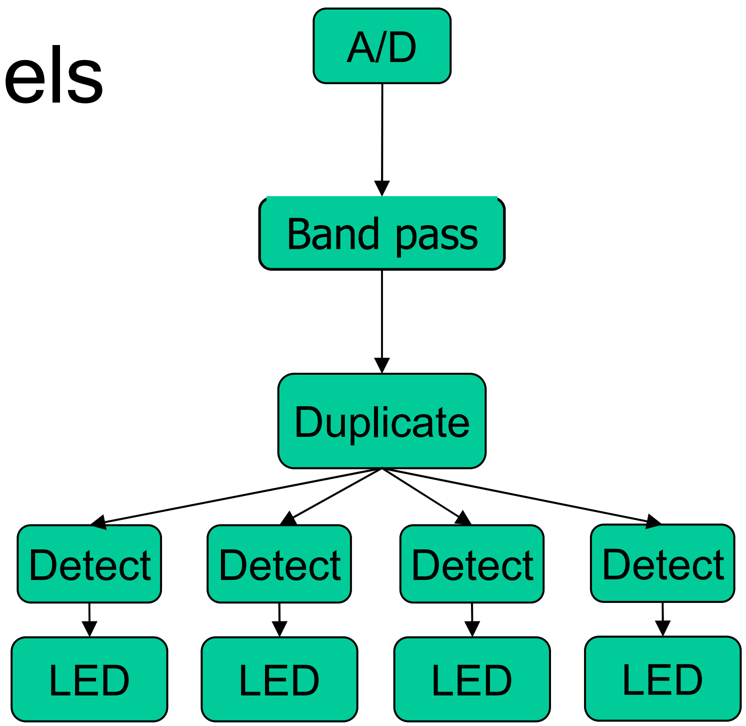
- Introduction
- StreamIt Language
- Domain-specific Optimizations
- Targeting Parallel Architectures
- Public Release
- Conclusions

Outline

- Introduction
- StreamIt Language
- Domain-specific Optimizations
- Targeting Parallel Architectures
- Public Release
- Conclusions

Model of Computation

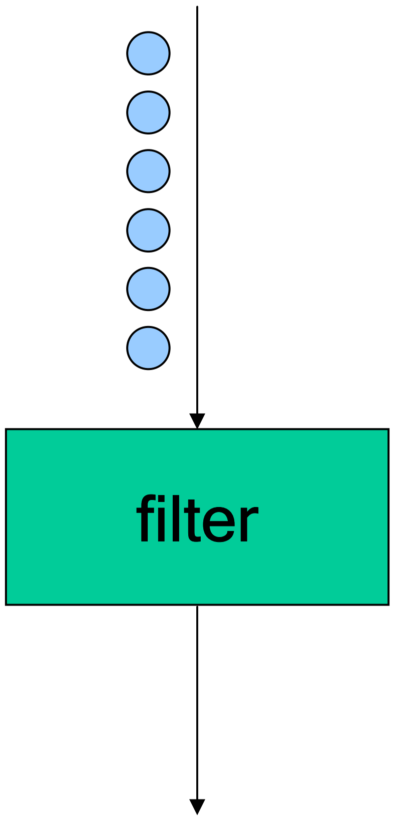
- Synchronous Dataflow [Lee 1992]
  - Graph of independent filters
  - Communicate via channels
  - Static I/O rates



Freq band detector

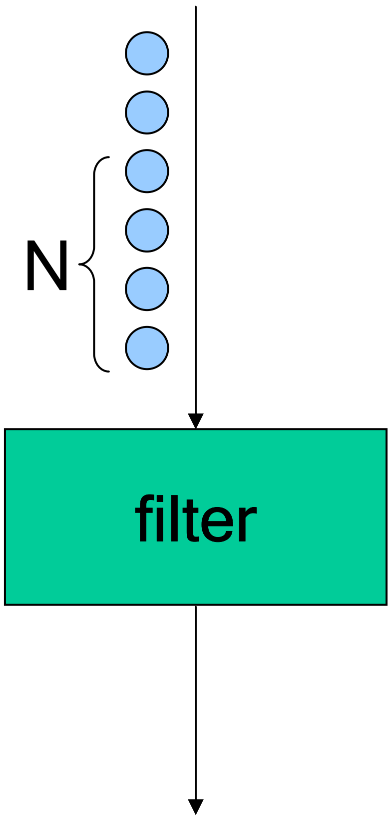
Filter Example: LowPassFilter

```
float->float filter LowPassFilter (int N, float freq) {
    float[N] weights;

    init {
        weights = calcWeights(N, freq);
    }

    work peek N pop 1 push 1 {
        float result = 0;
        for (int i=0; i<weights.length; i++) {
            result += weights[i] * peek(i);
        }
        push(result);
        pop();
    }
}
```



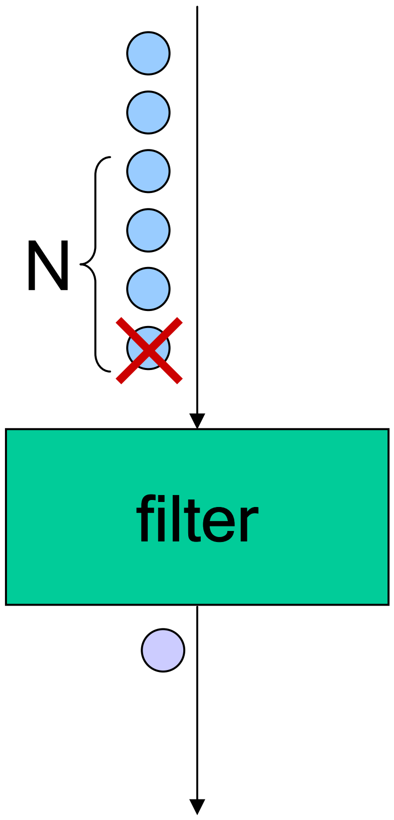
Filter Example: LowPassFilter

```
float->float filter LowPassFilter (int N, float freq) {
  float[N] weights;

  init {
    weights = calcWeights(N, freq);
  }

  work peek N pop 1 push 1 {
    float result = 0;
    for (int i=0; i<weights.length; i++) {
      result += weights[i] * peek(i);
    }
    push(result);
    pop();
  }
}
```



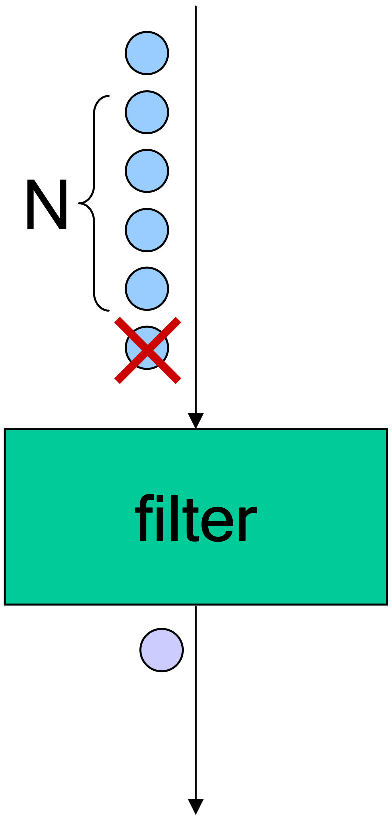
Filter Example: LowPassFilter

```
float->float filter LowPassFilter (int N, float freq) {
  float[N] weights;

  init {
    weights = calcWeights(N, freq);
  }

  work peek N pop 1 push 1 {
    float result = 0;
    for (int i=0; i<weights.length; i++) {
      result += weights[i] * peek(i);
    }
    push(result);
    pop();
  }
}
```



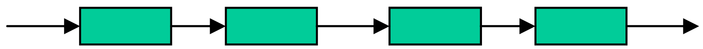
Filter Example: LowPassFilter

```
float->float filter LowPassFilter (int N, float freq) {
    float[N] weights;

    init {
        weights = calcWeights(N, freq);
    }

    work peek N pop 1 push 1 {
        float result = 0;
        for (int i=0; i<weights.length; i++) {
            result += weights[i] * peek(i);
        }
        push(result);
        pop();
    }
}
```

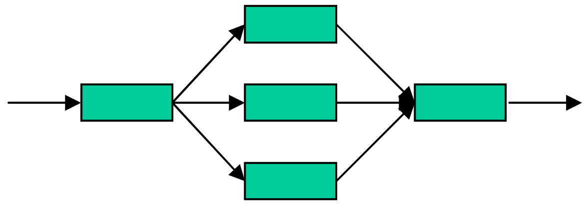


Composing Filters: Structured Streams

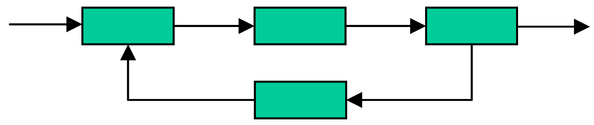
- Hierarchical structures:
  - Pipeline



  - SplitJoin



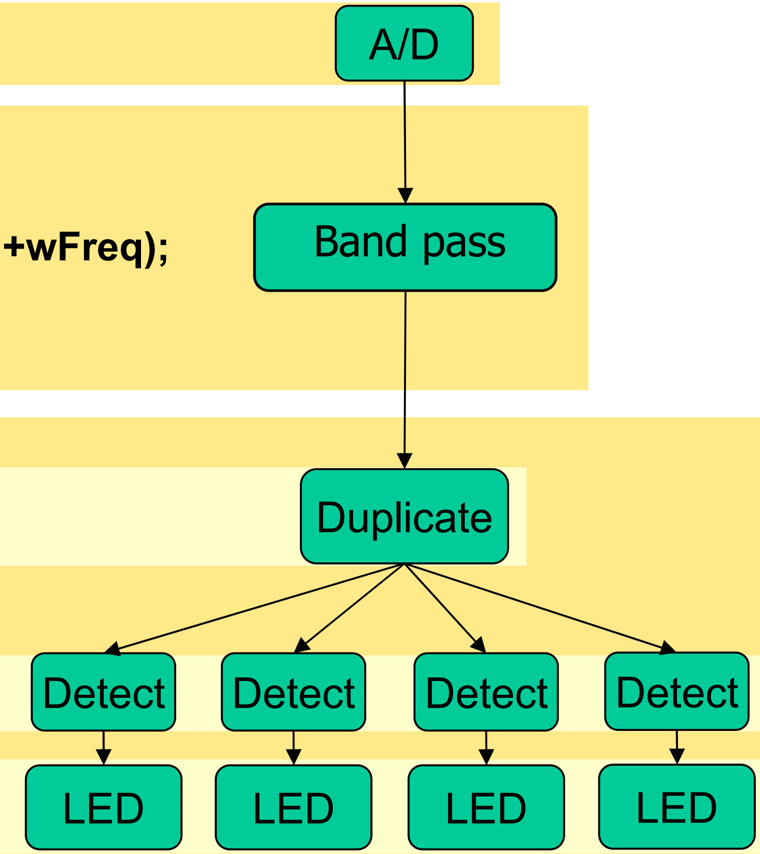
  - Feedback Loop



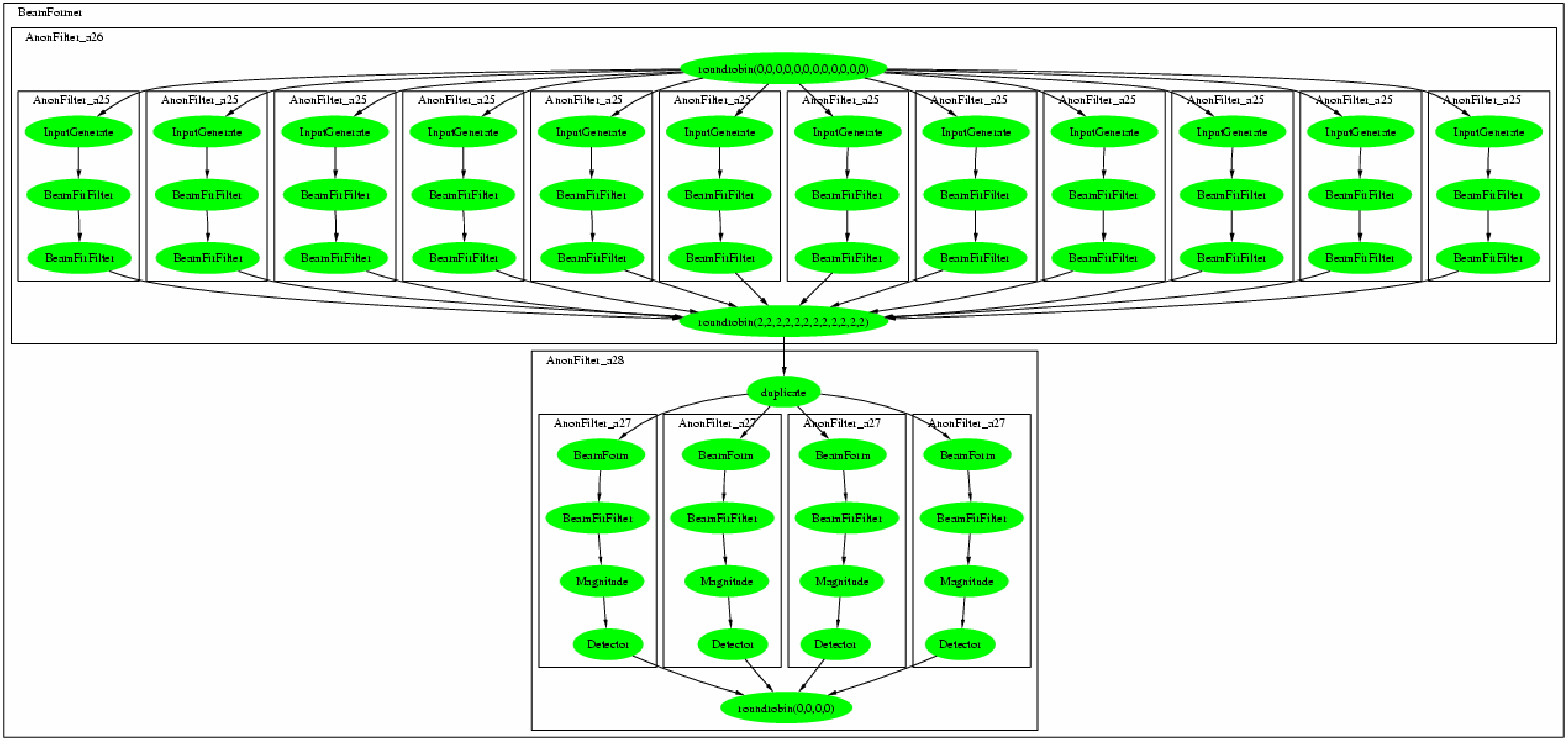
- Basic programmable unit: Filter

Freq Band Detector in StreamIt

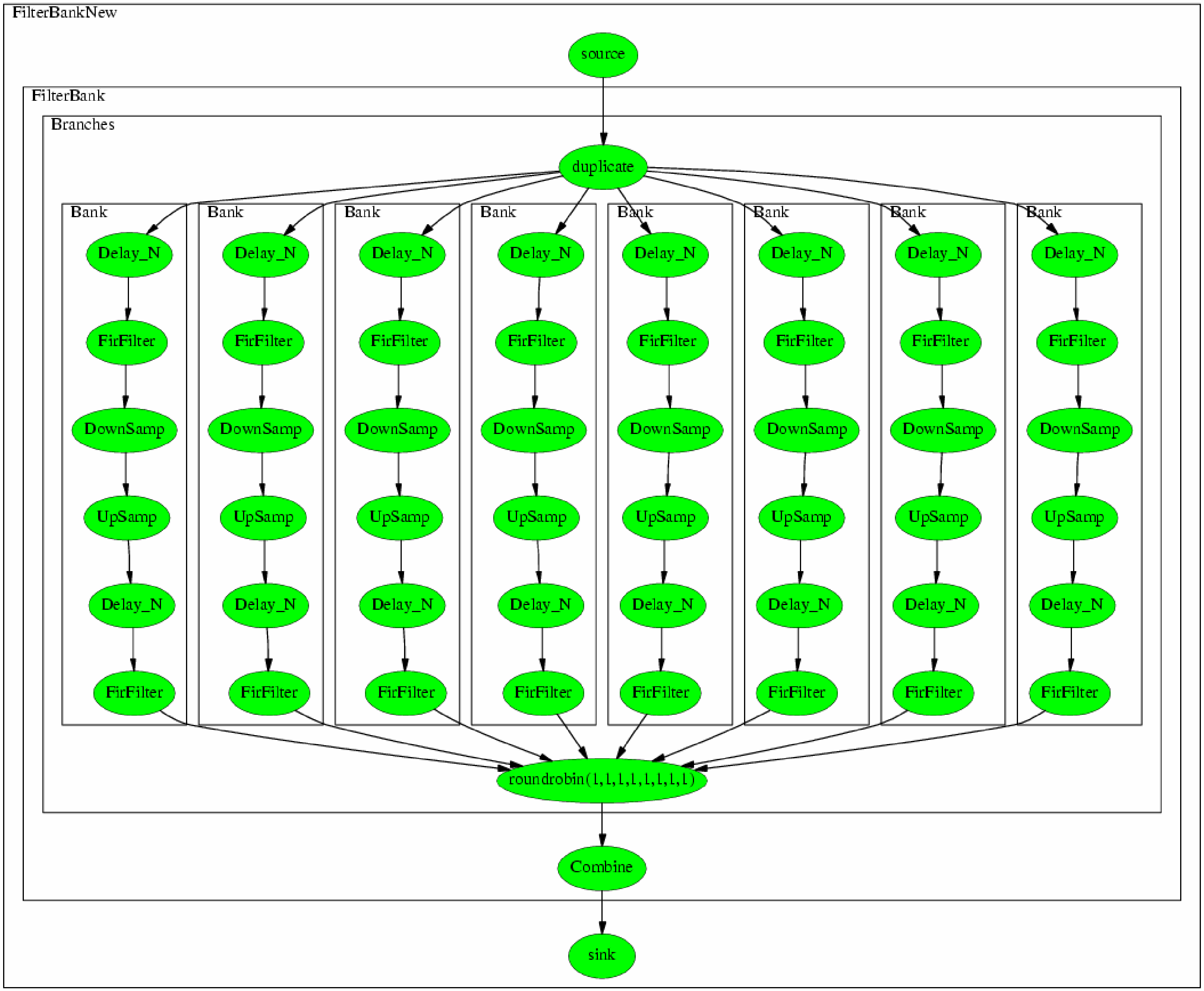
```
void->void pipeline FrequencyBand {
  float sFreq = 4000;
  float cFreq = 500/(sFreq*2*pi);
  float  wFreq = 100/(sFreq*2*pi);

  add D2ASource(sFreq);

  add BandPassFilter(100, cFreq-wFreq, cFreq+wFreq);

  add splitjoin {
    split duplicate;
    for (int i=0; i<4; i++) {
      add pipeline {
        add Detect (i/4);
        add LED (i);
      }
    }
    join roundrobin(0);
  }
 }
}
```



Radar-Array Front End

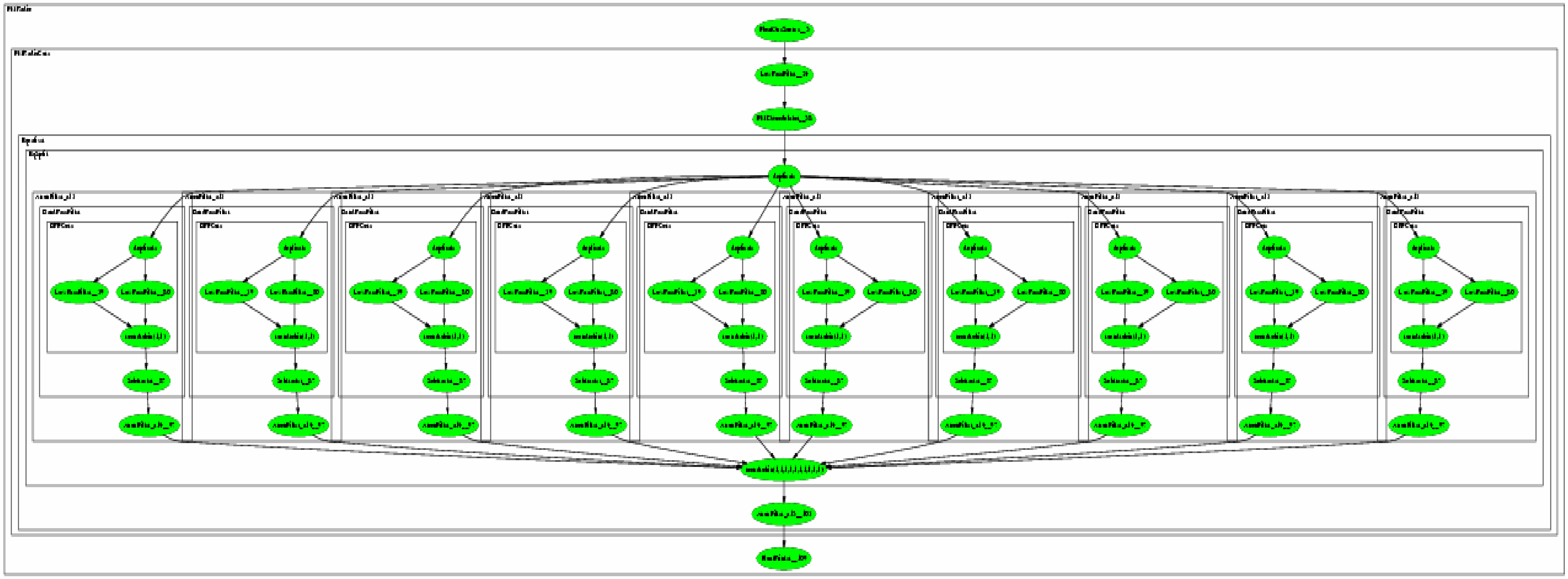


Filterbank

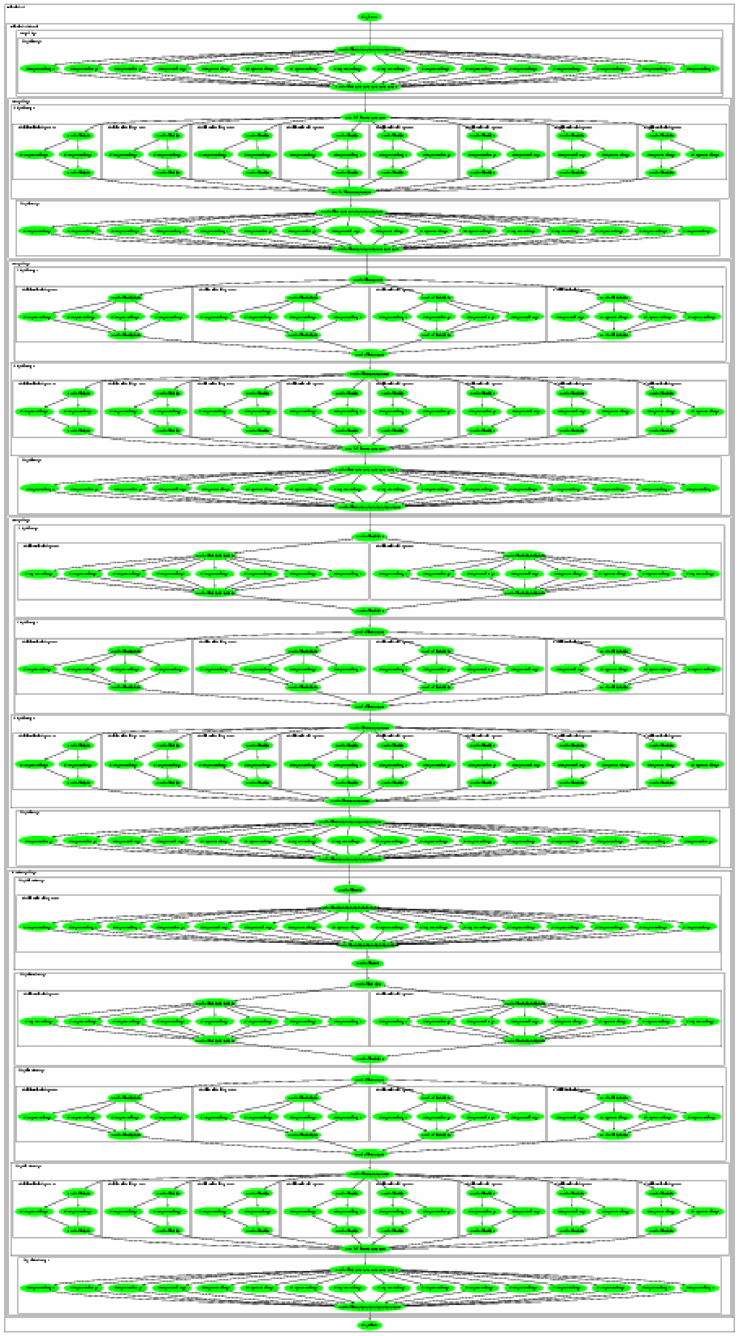


FilterBankNew
source
FilterBank
Branches
duplicate
Bank
Delay_N
FirFilter
DownSamp
UpSamp
Delay_N
FirFilter
roundrobin(1,1,1,1,1,1,1,1)
Combine
sink

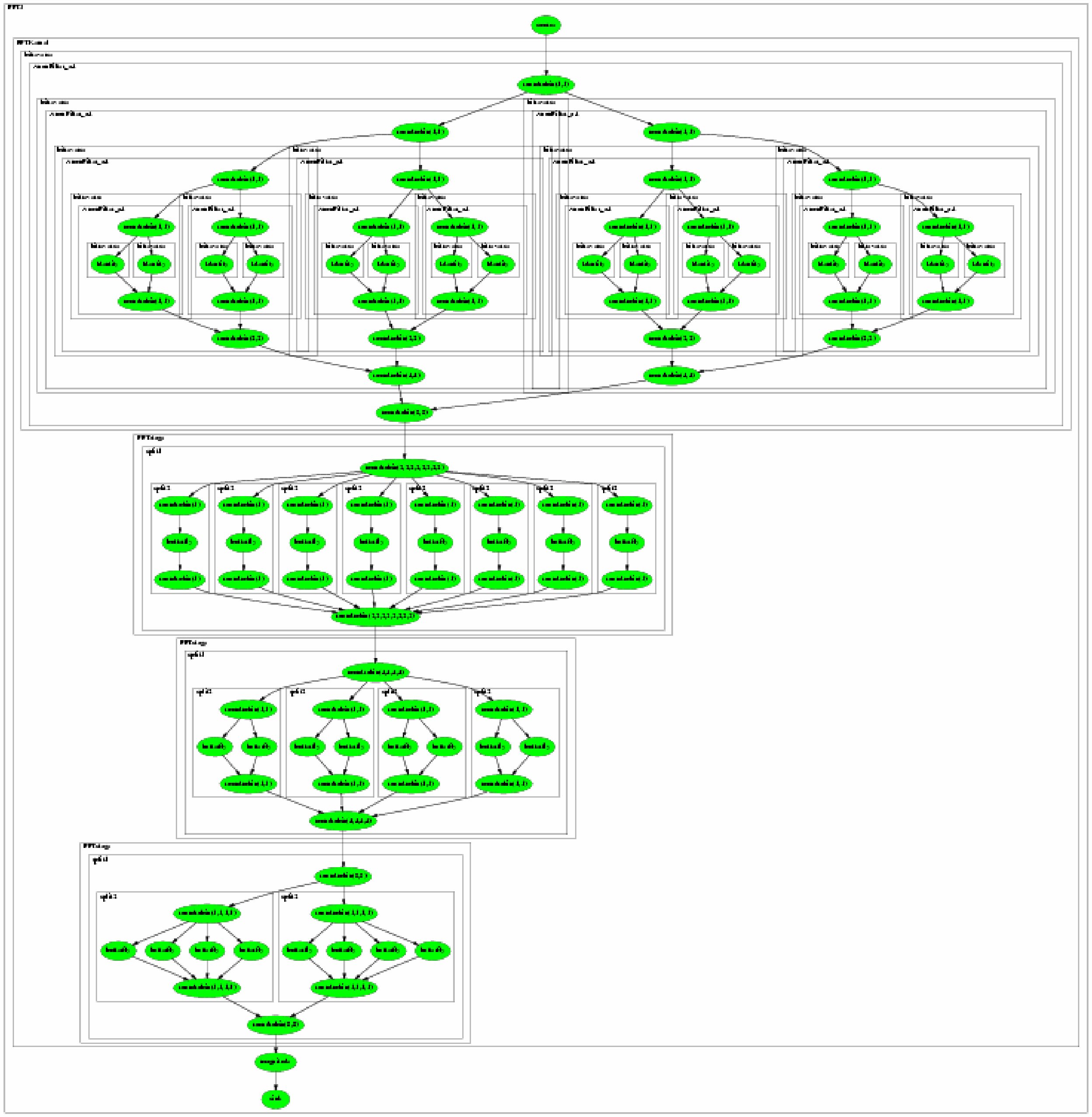
FM Radio with Equalizer



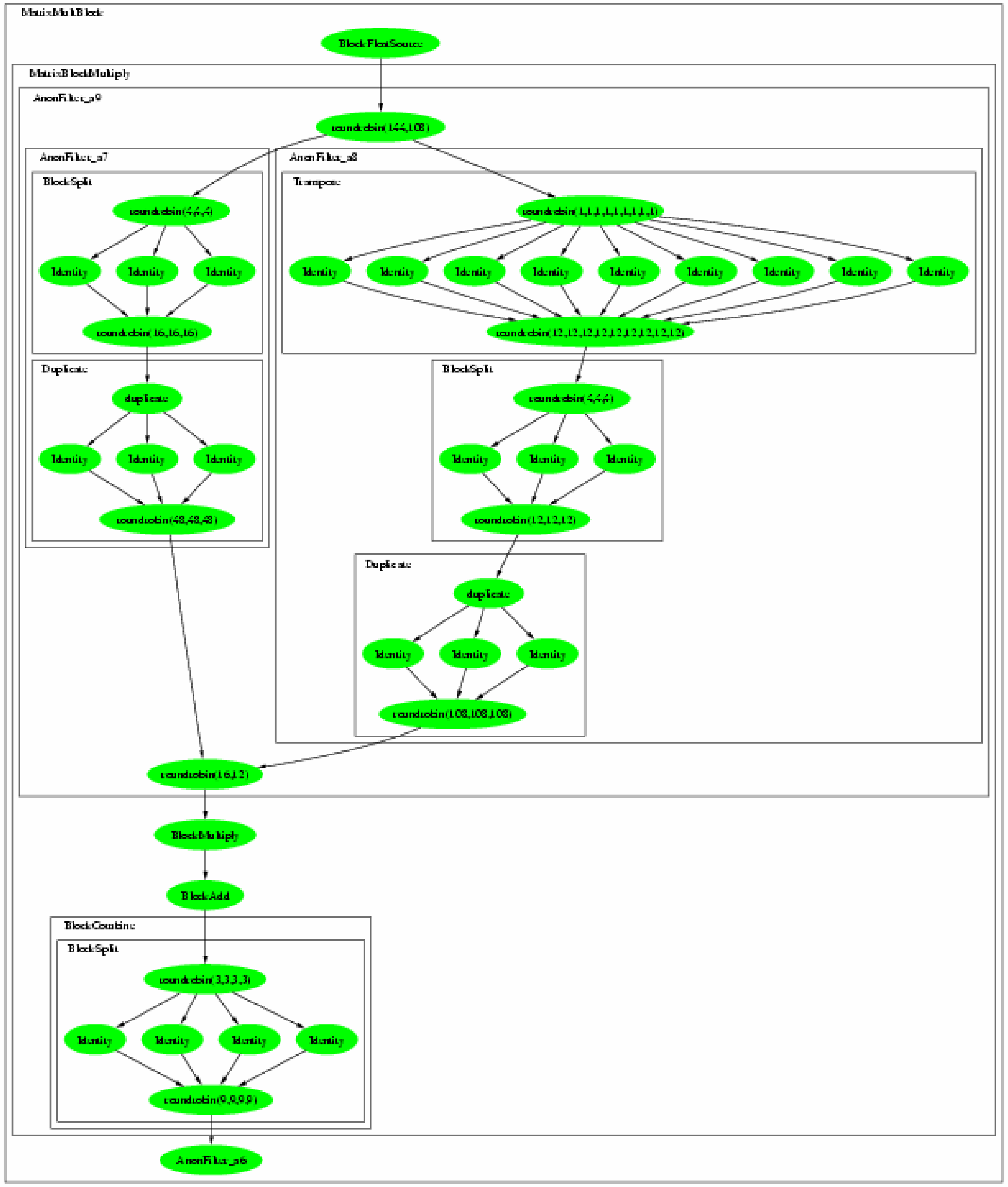
Bitonic Sort



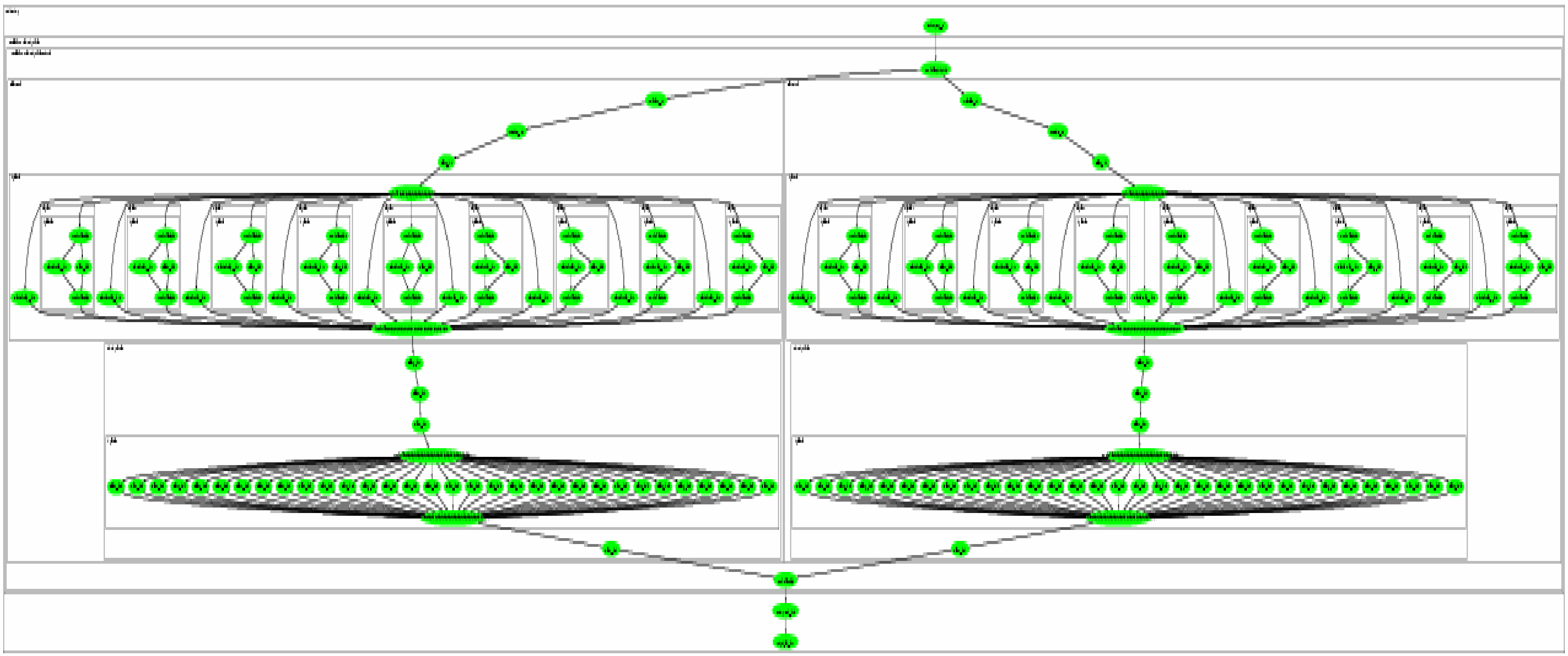
FFT



Block Matrix Multiply



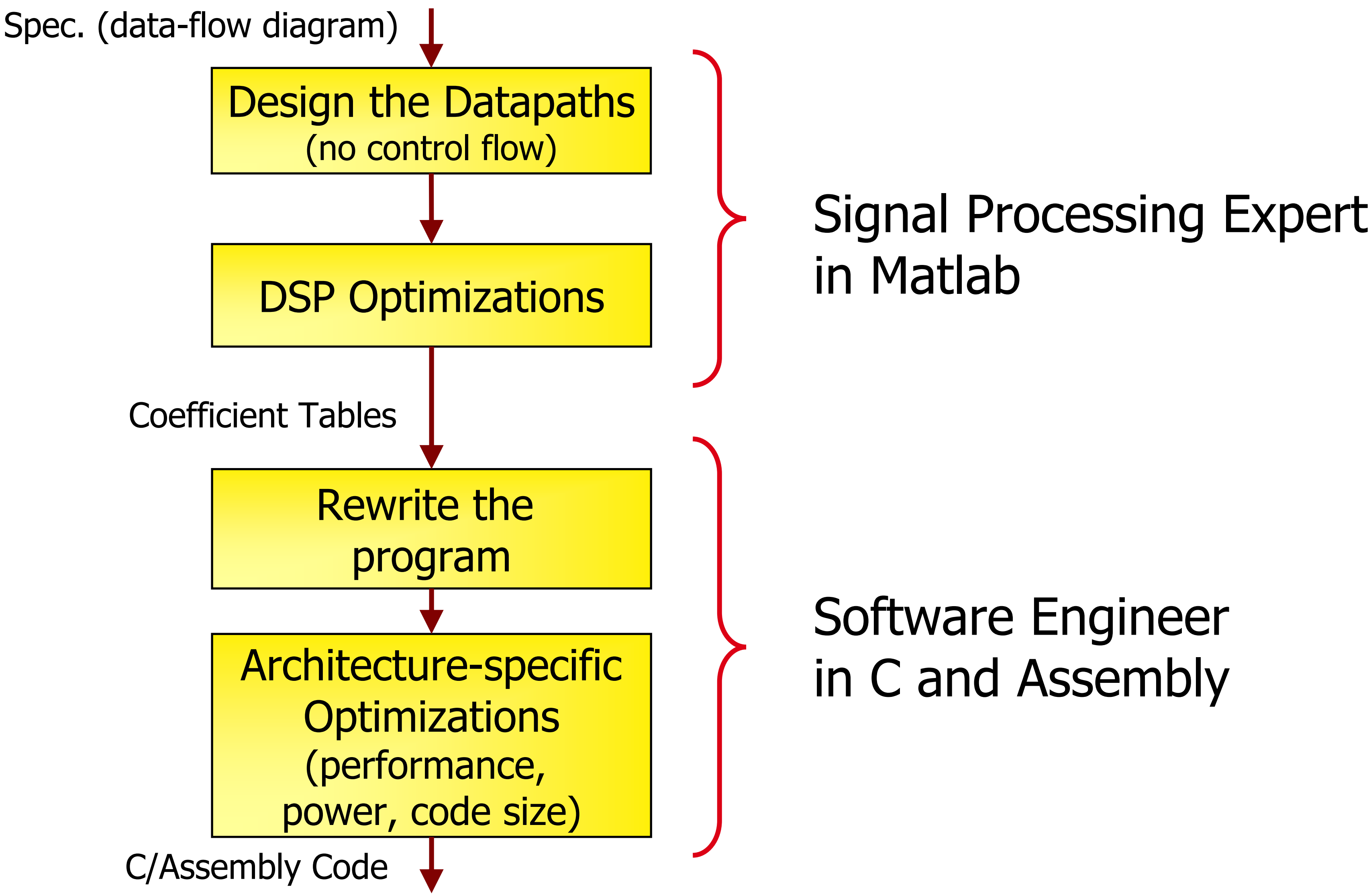
MP3 Decoder



Outline

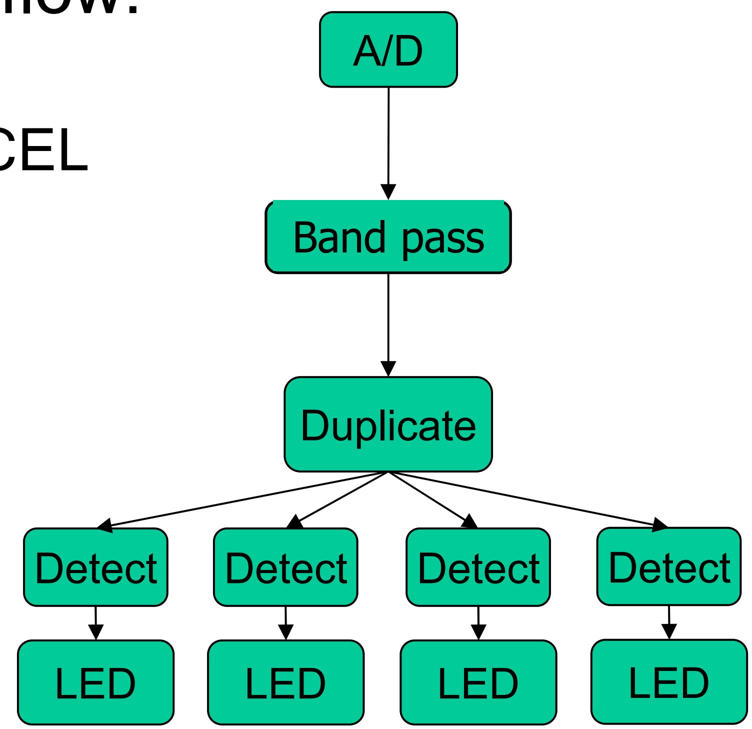
- Introduction
- StreamIt Language
- Domain-specific Optimizations
- Targeting Parallel Architectures
- Public Release
- Conclusions

Conventional DSP Design Flow



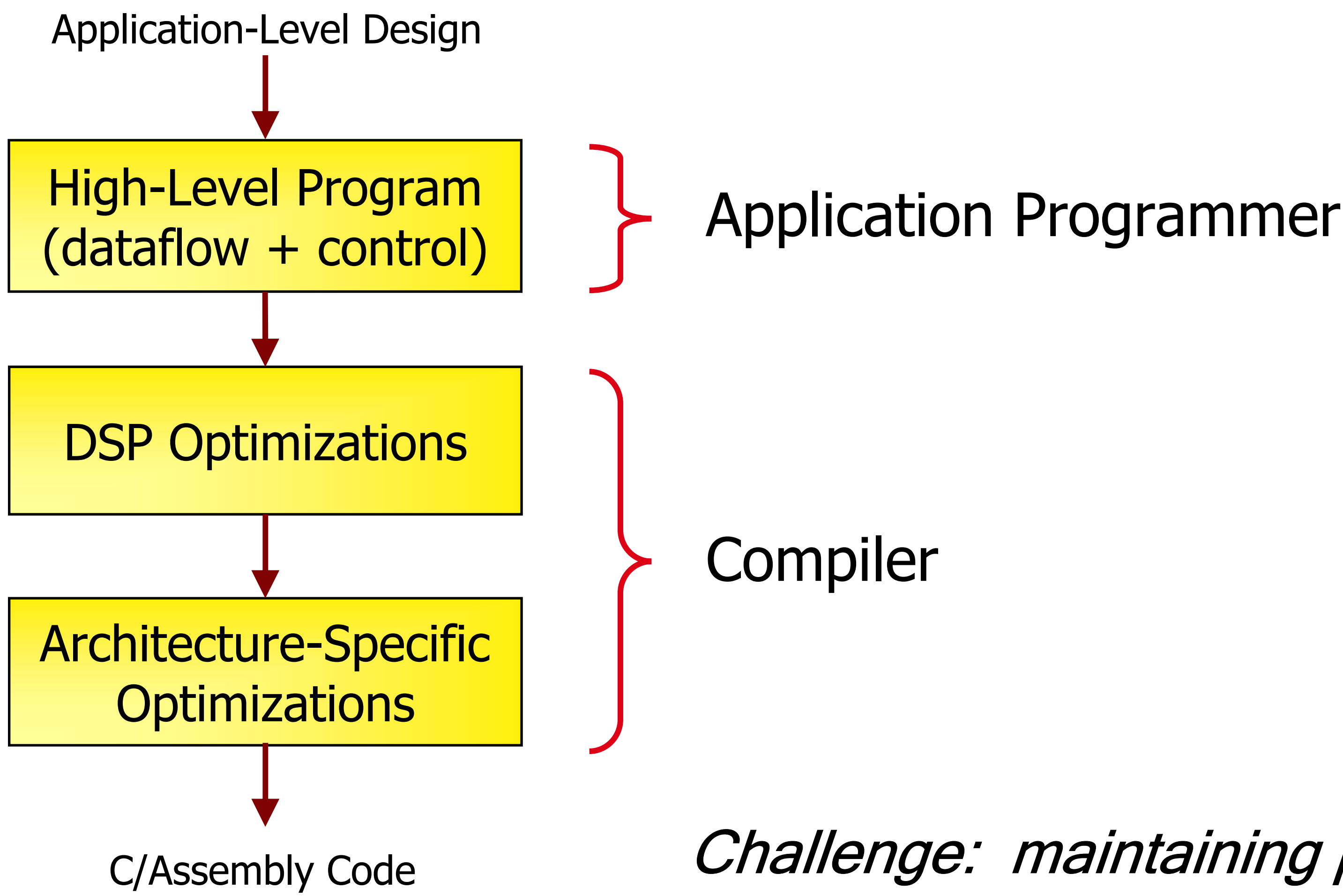
Any Design Modifications?

- Center frequency from 500 Hz to 1200 Hz?
  - According to TI,
    in the conventional design-flow:
    - Redesign filter in MATLAB
    - Cut-and-paste values to EXCEL
    - Recalculate the coefficients
    - Update assembly



Source:
Application Report SPRA414
Texas Instruments, 1999

Ideal DSP Design Flow



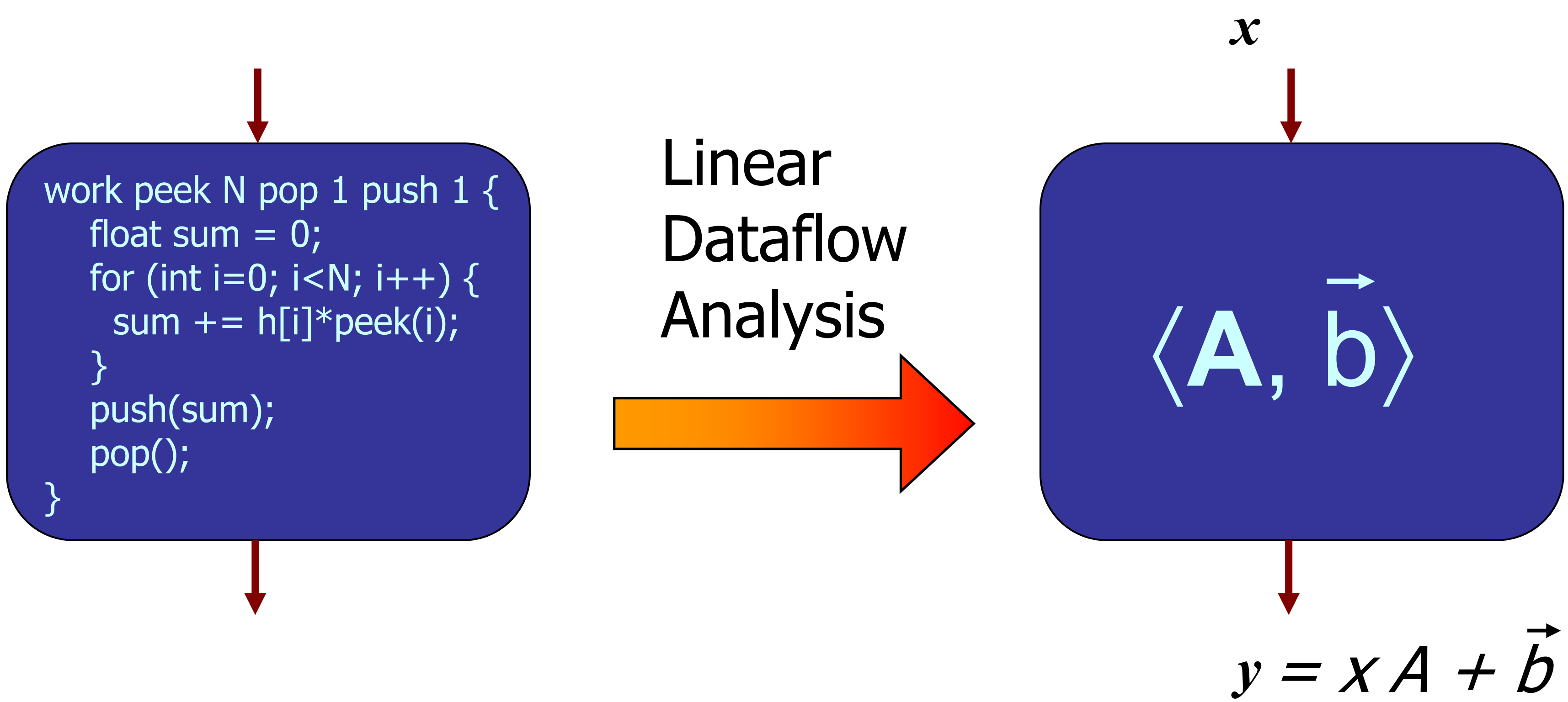
Challenge: maintaining performance

Our Focus: Linear Filters

- Most common target of DSP optimizations
  - FIR filters
  - Compressors
  - Expanders
  - DFT/DCT

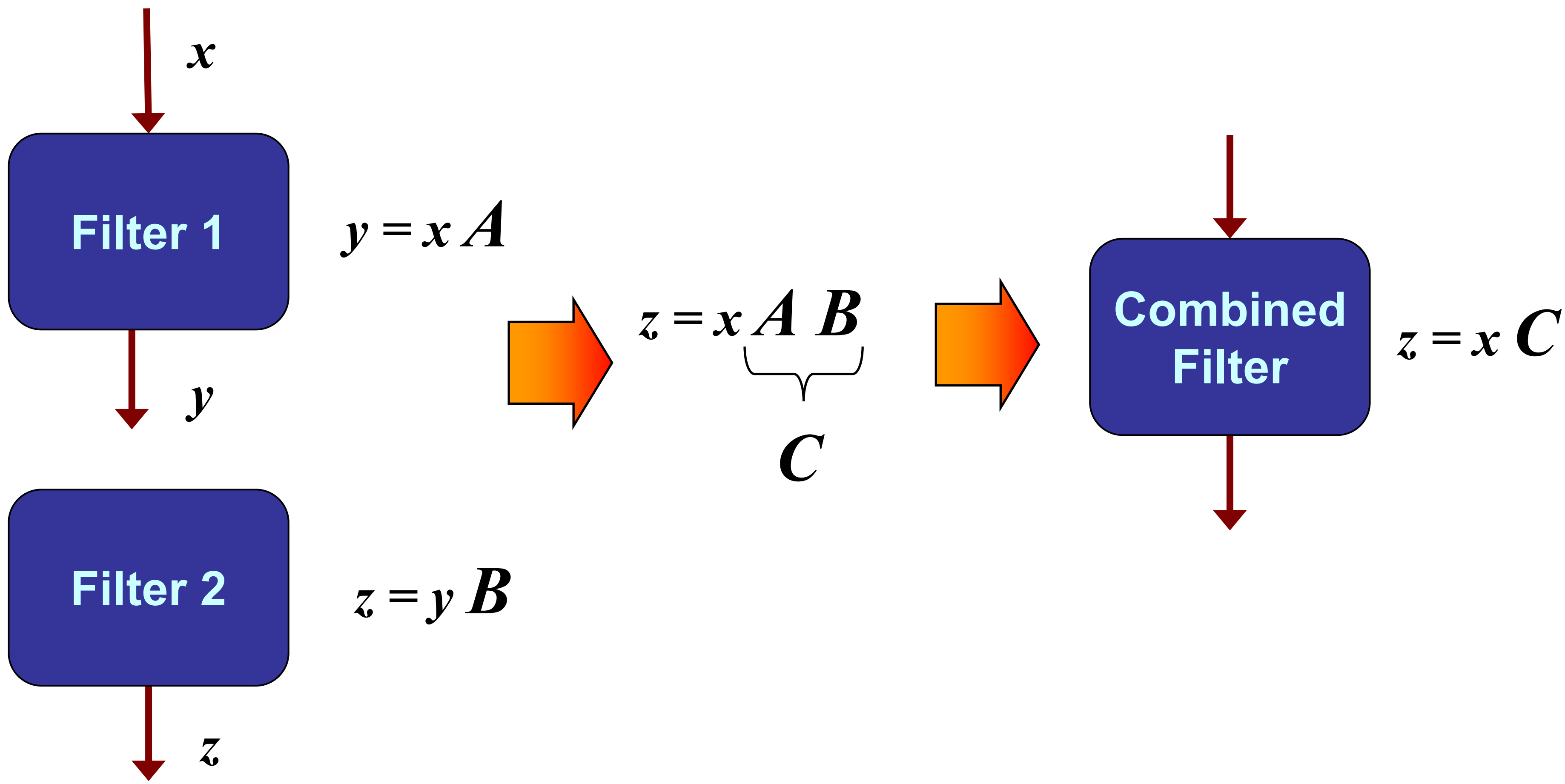
Output is weighted sum of inputs

Extracting Linear Representation

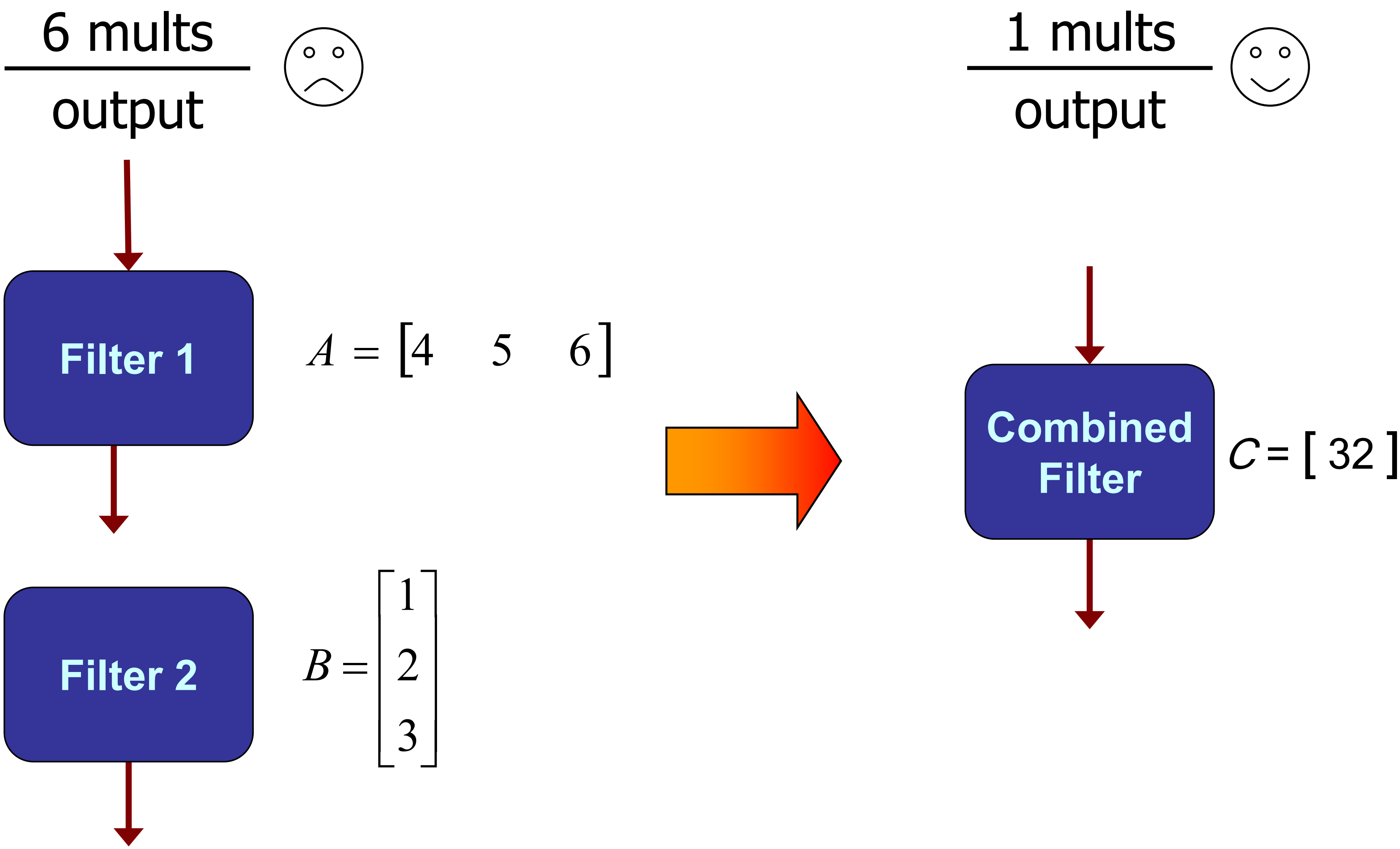


1) Combining Linear Filters

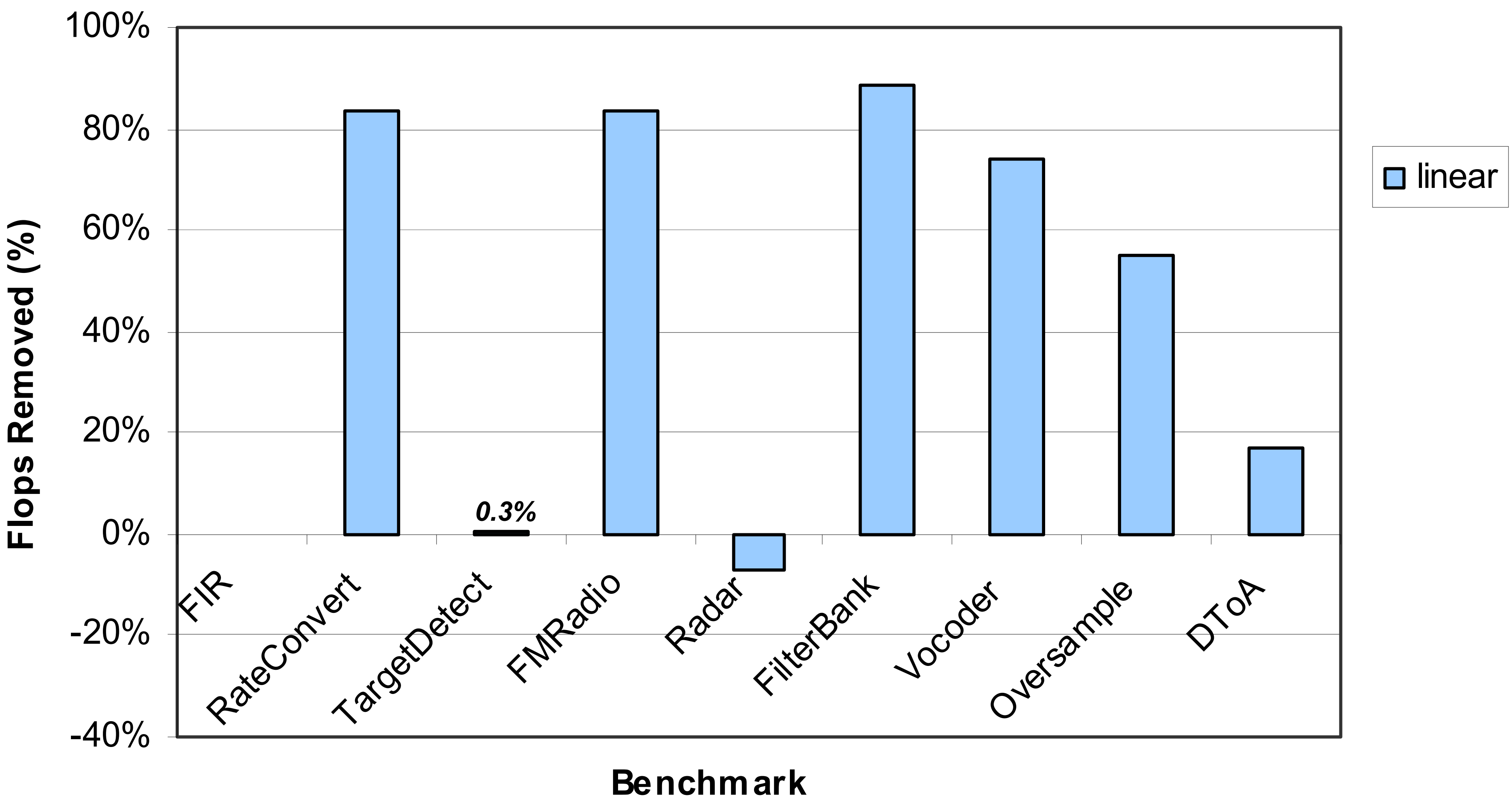
- Pipelines and splitjoins can be collapsed
- Example: pipeline



Combination Example

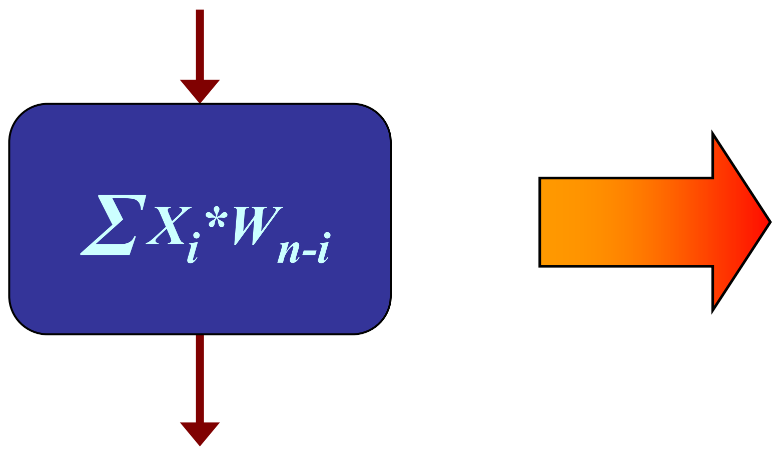


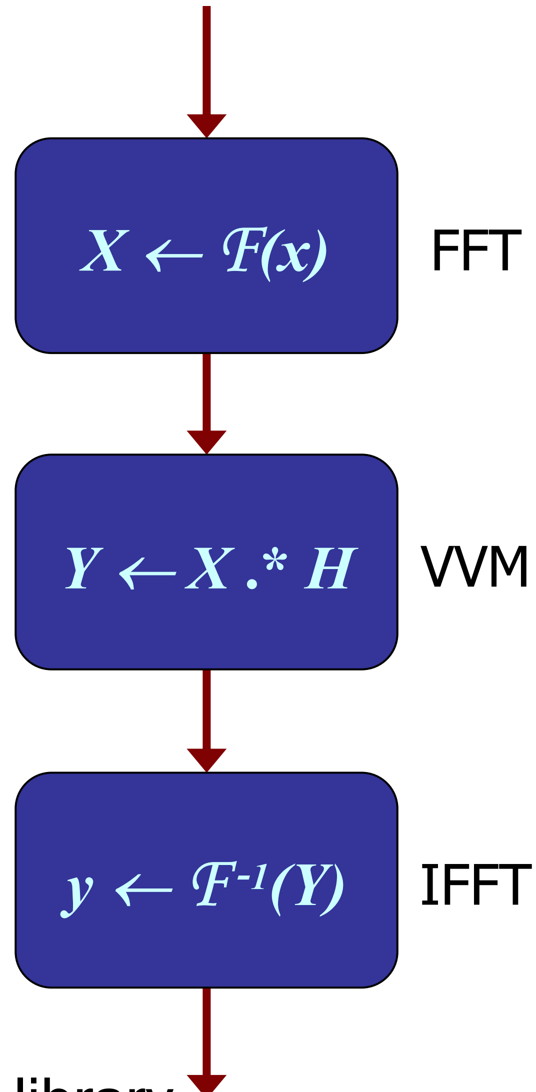
Floating-Point Operations Reduction



2) From Time to Frequency Domain

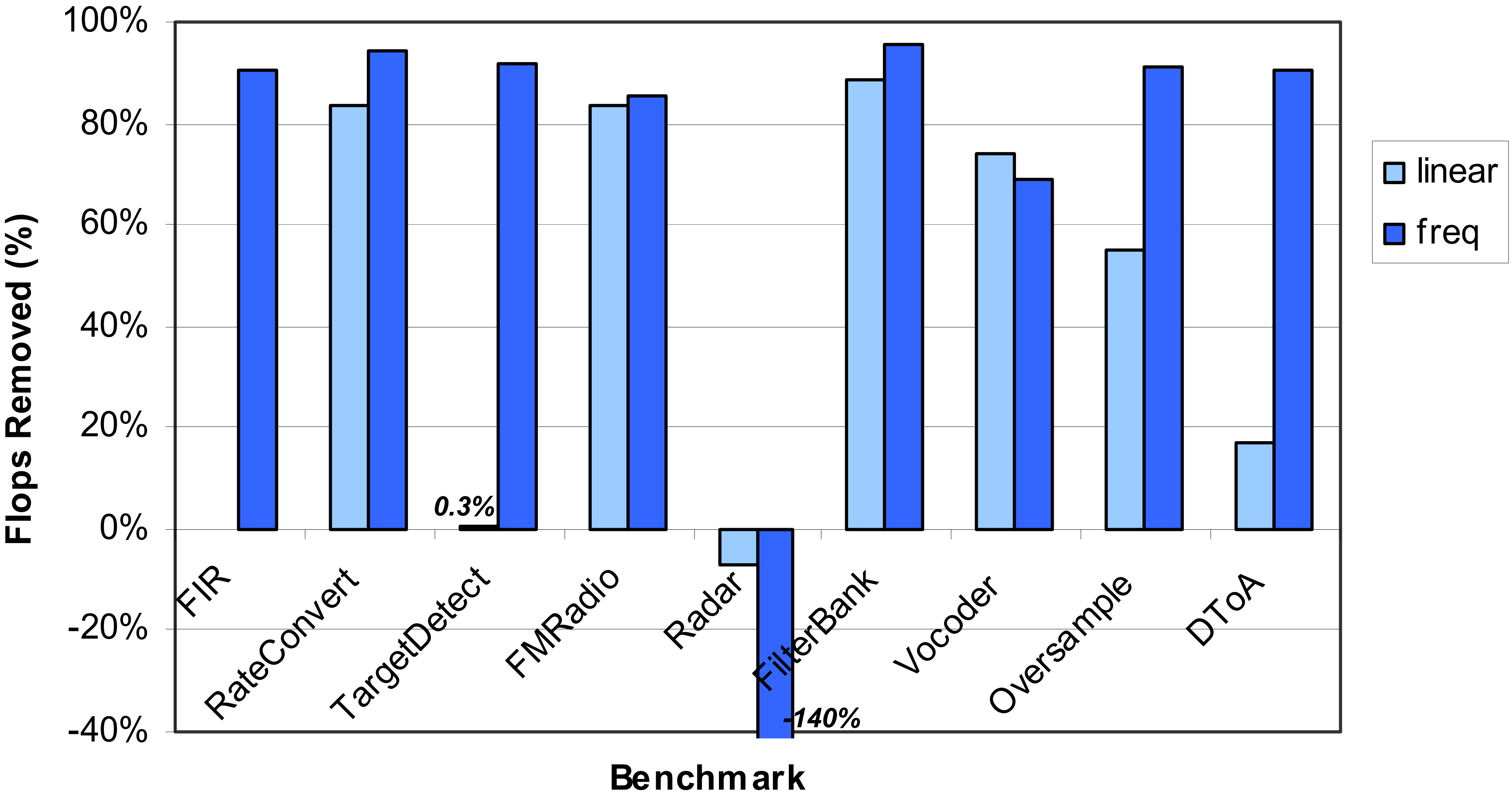
- Convolutions can be done cheaply in the Frequency Domain





- Painful to do by hand
  - Blocking
  - Coefficient calculations
  - Startup
  - Multiple outputs
  - Interfacing with FFT library
  - Verification

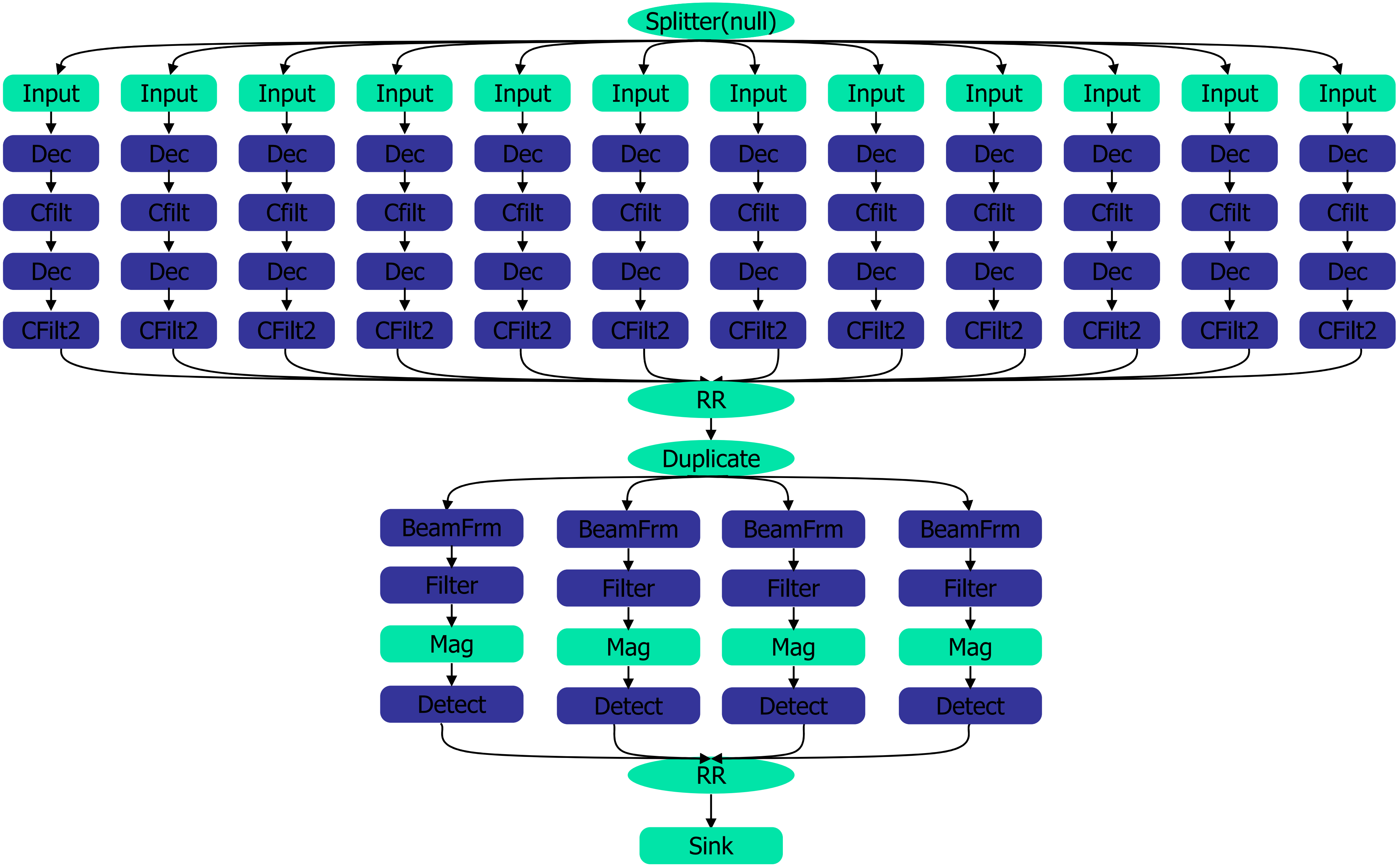
Floating-Point Operations Reduction



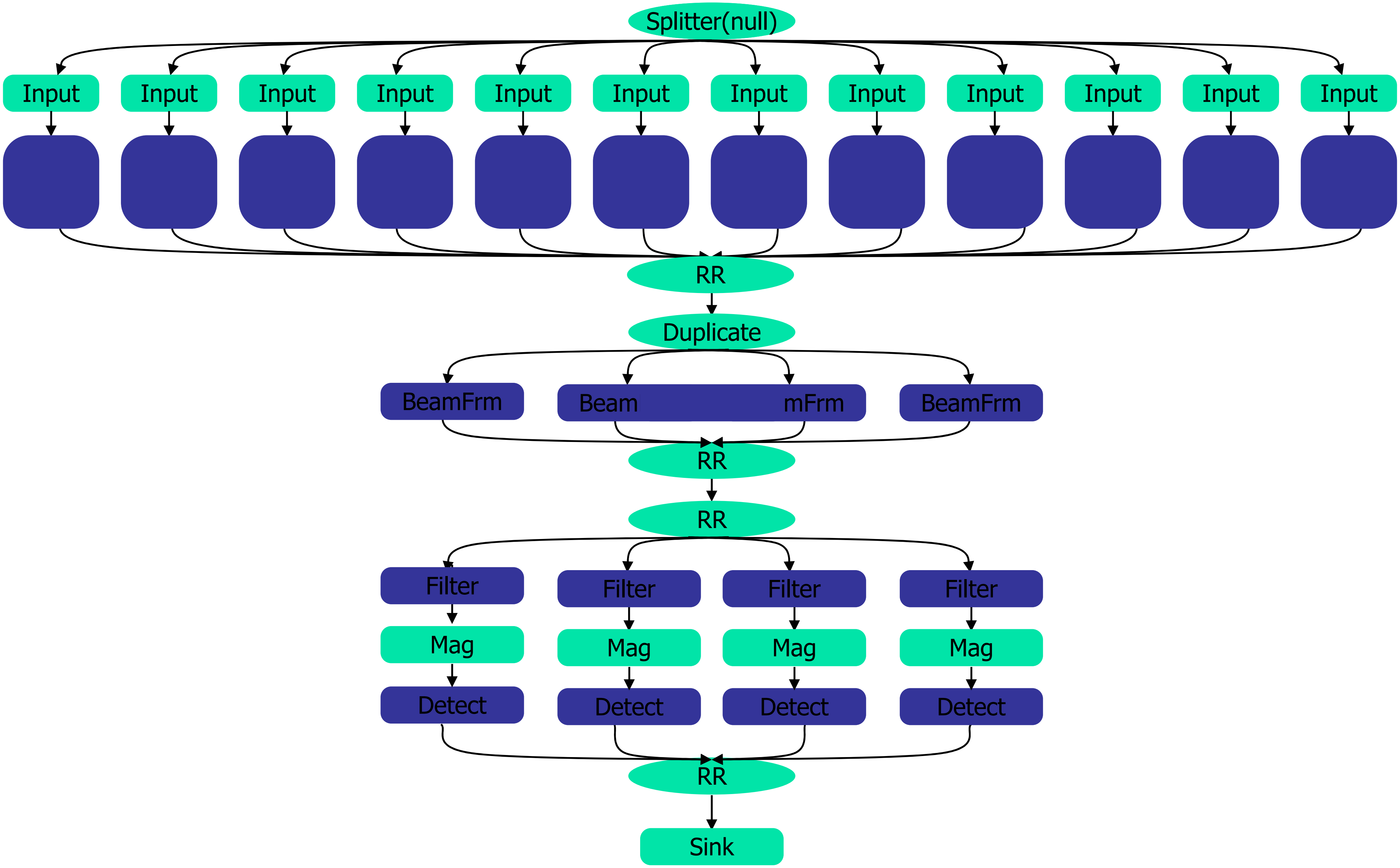
3) When to Apply Transformations?

- Estimate minimal cost for each structure:
  - Linear combination
  - Frequency translation
  - No transformation
    - If hierarchical, consider all rectangular groupings of children
- Overlapping sub-problems allows efficient dynamic programming search

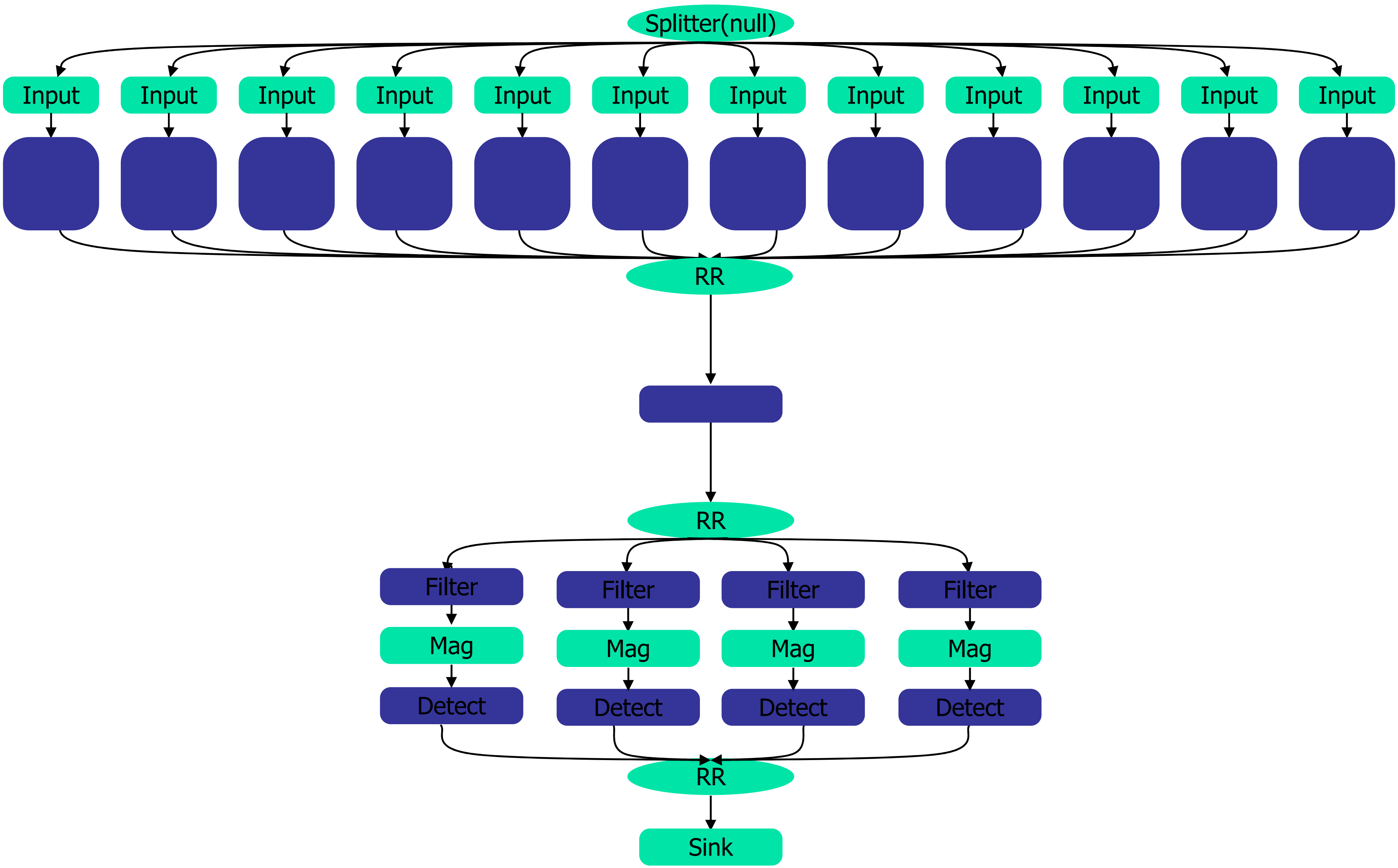
Radar (Transformation Selection)



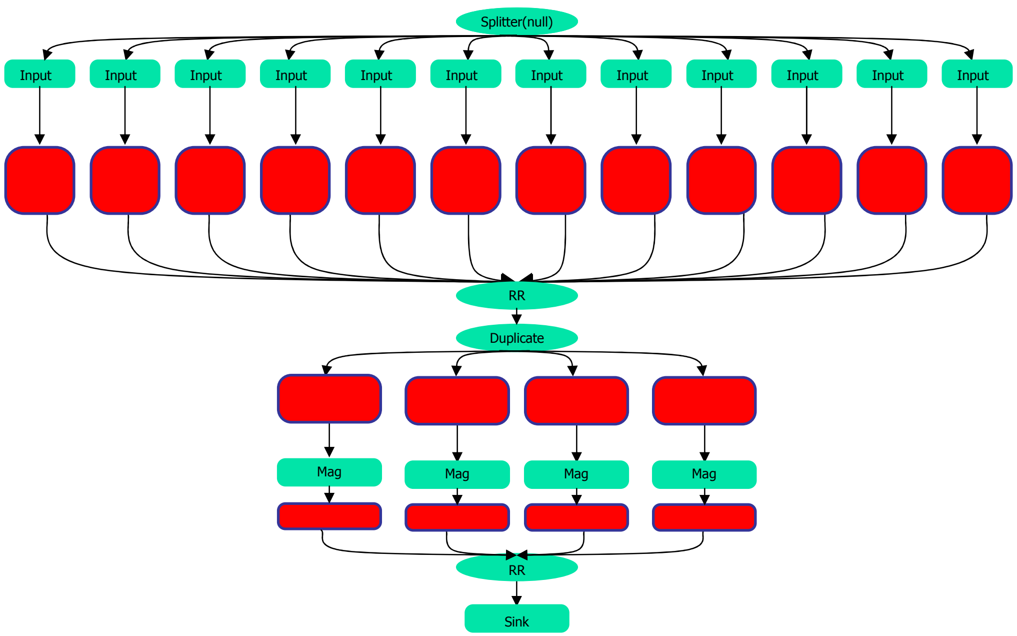
Radar (Transformation Selection)



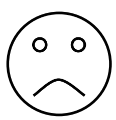
Radar (Transformation Selection)



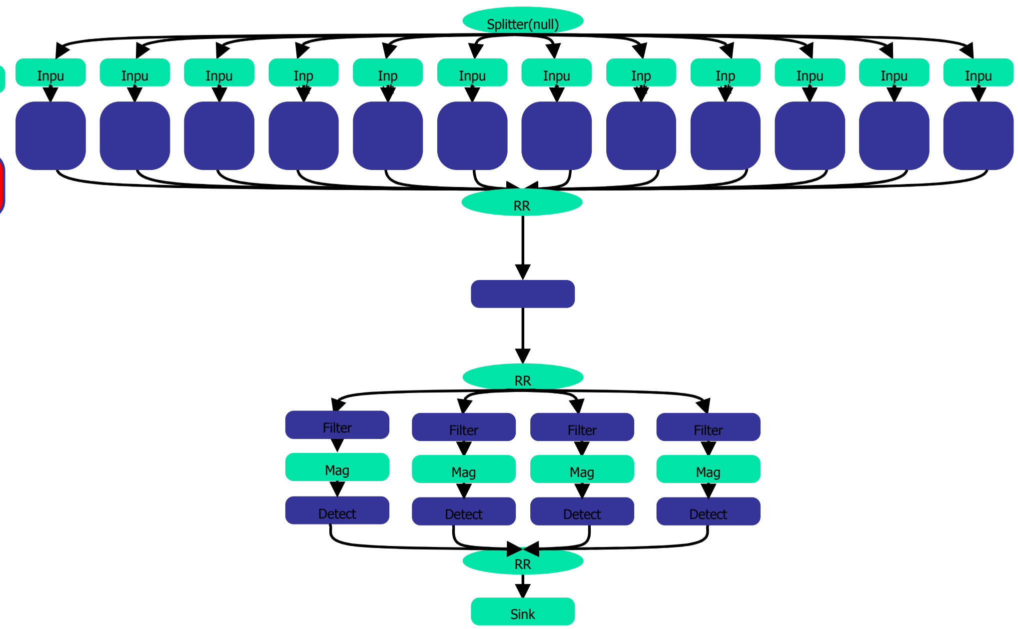
Radar



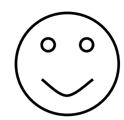
Maximal Combination and Shifting to Frequency Domain



2.4 times as many FLOPS

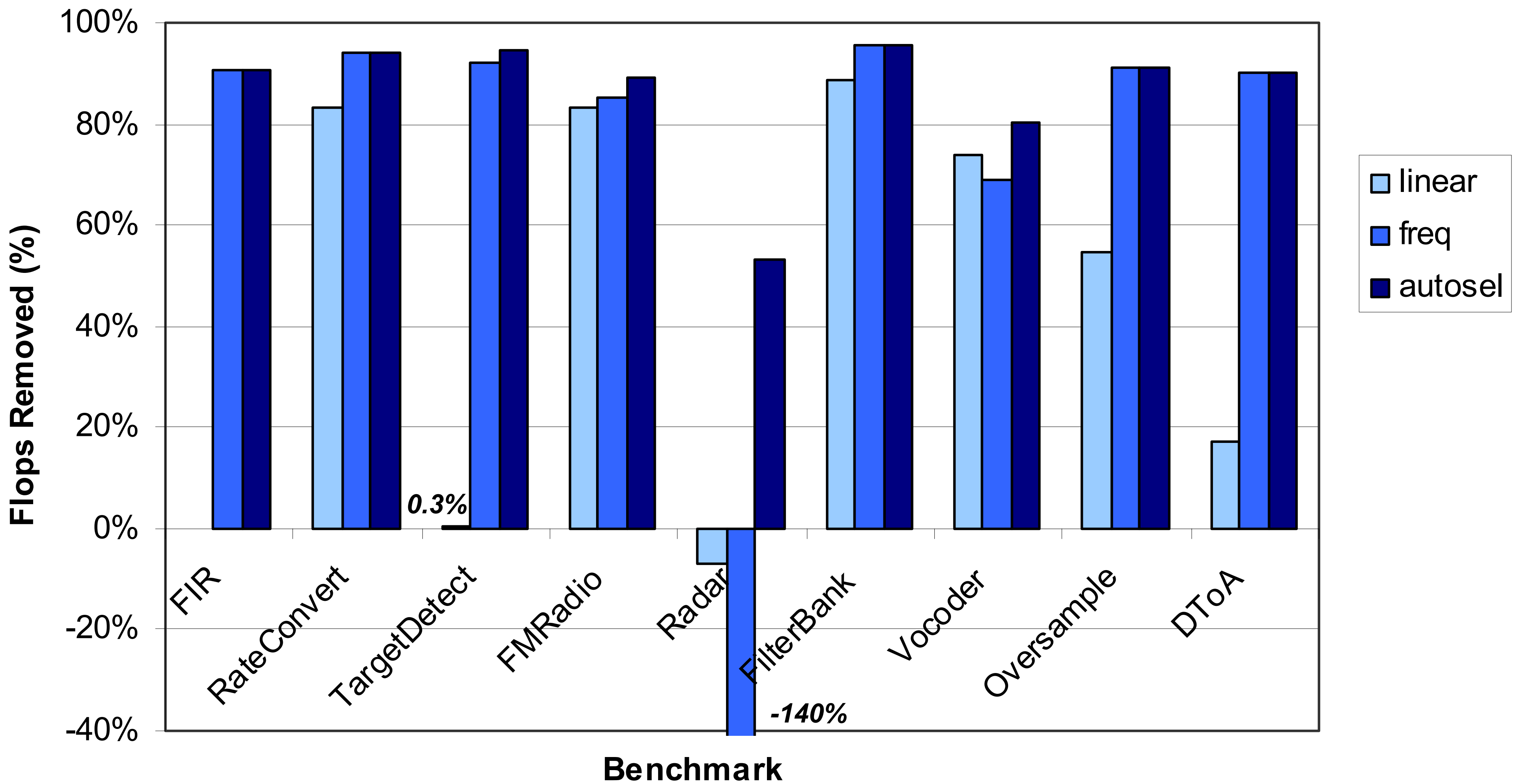


Using Transformation Selection



half as many FLOPS

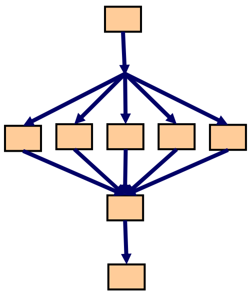
Floating-Point Operations Reduction

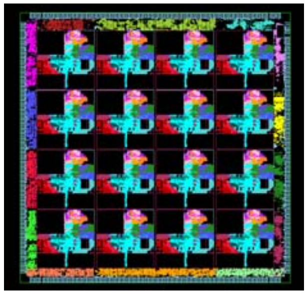


Outline

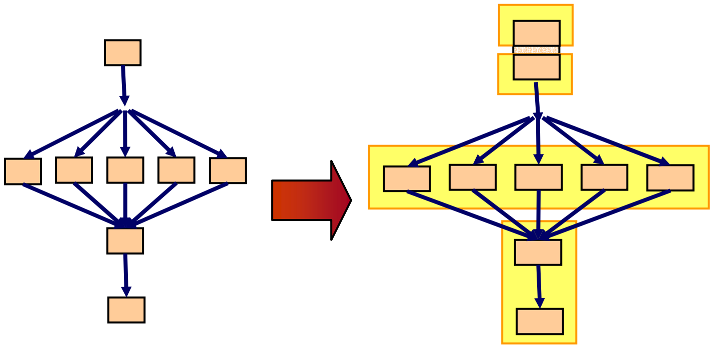
- Introduction
- StreamIt Language
- Domain-specific Optimizations
- Targeting Parallel Architectures
- Public Release
- Conclusions

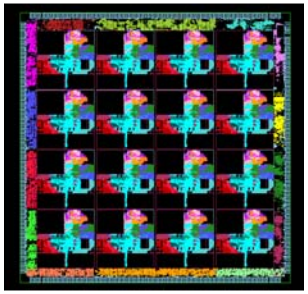
Compiling to the Raw Architecture





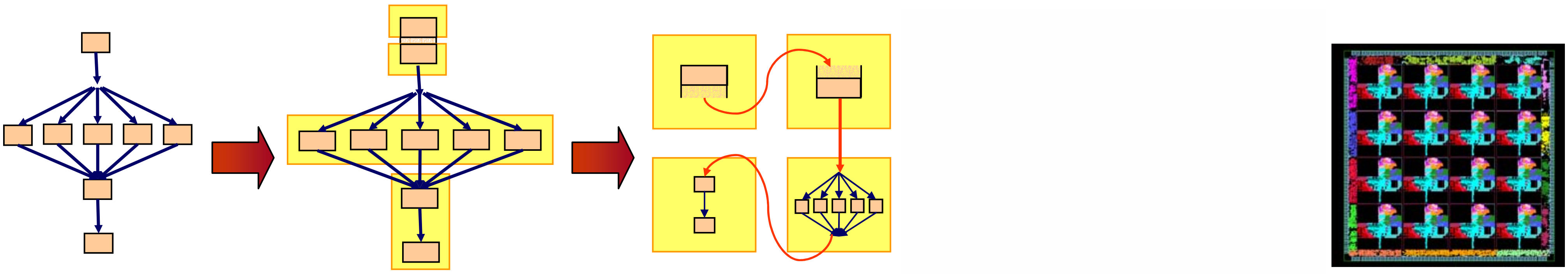
Compiling to the Raw Architecture





1. Partitioning: adjust granularity of graph

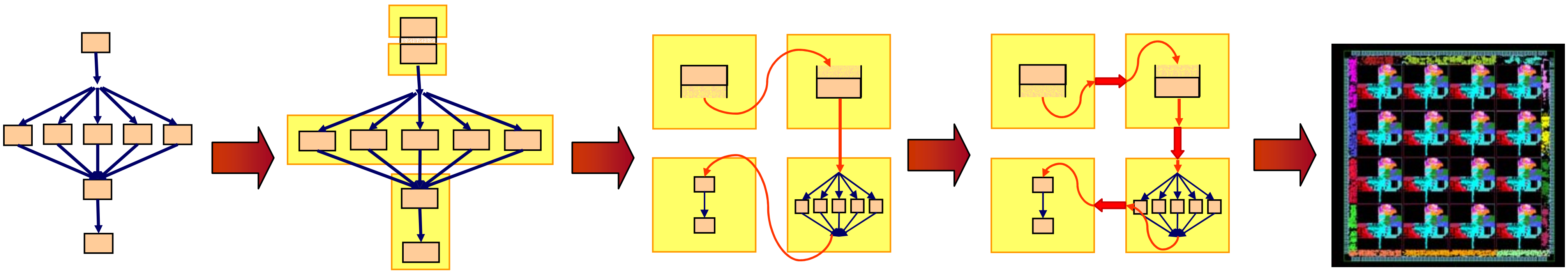
Compiling to the Raw Architecture



1. Partitioning: adjust granularity of graph

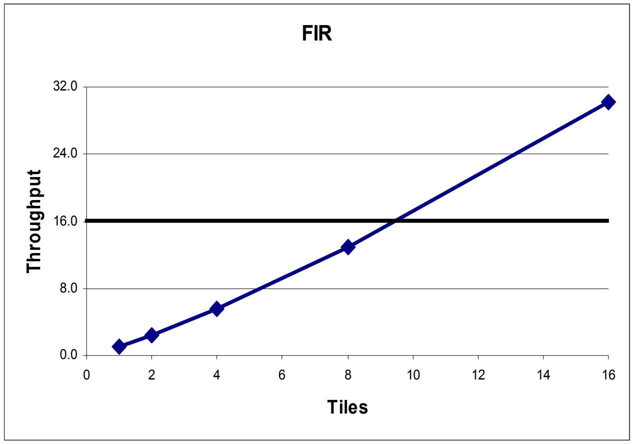
2. Layout: assign filters to tiles

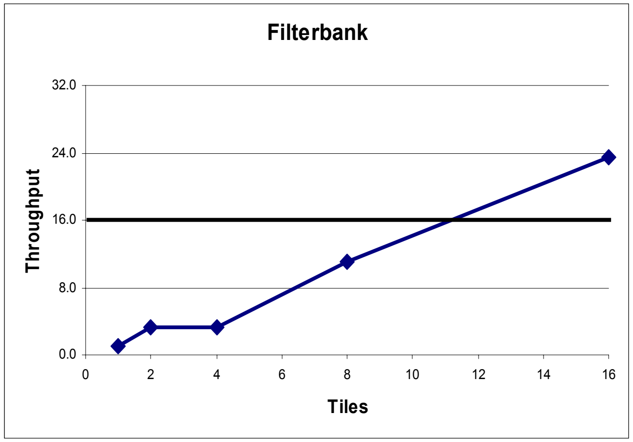
Compiling to the Raw Architecture

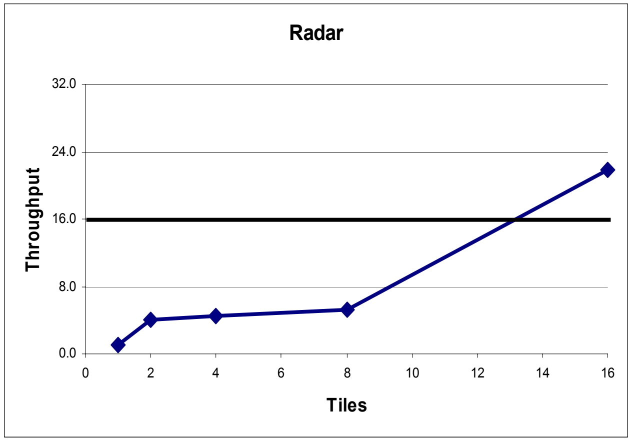


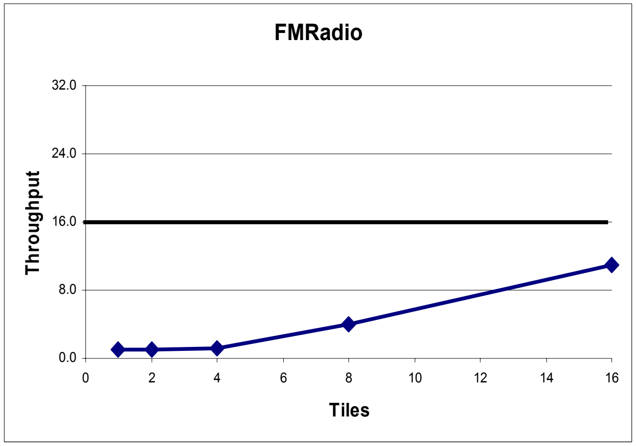
1. Partitioning: adjust granularity of graph
2. Layout: assign filters to tiles
3. Scheduling: route items across network

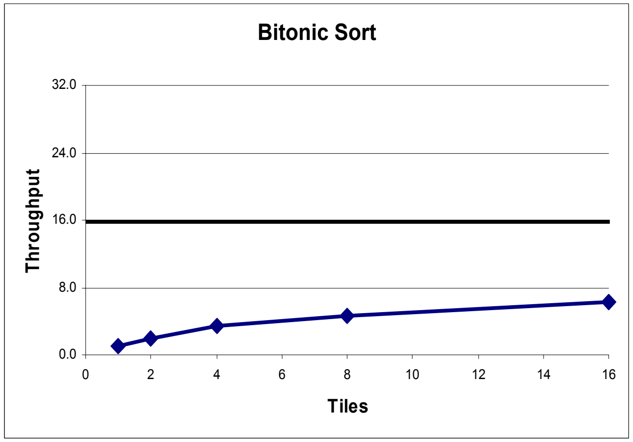
Scalability Results

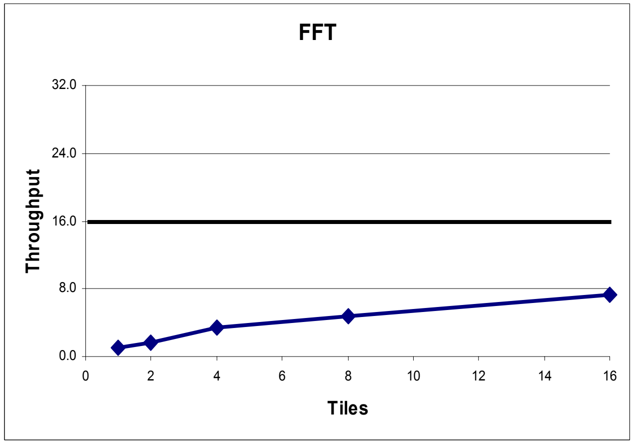




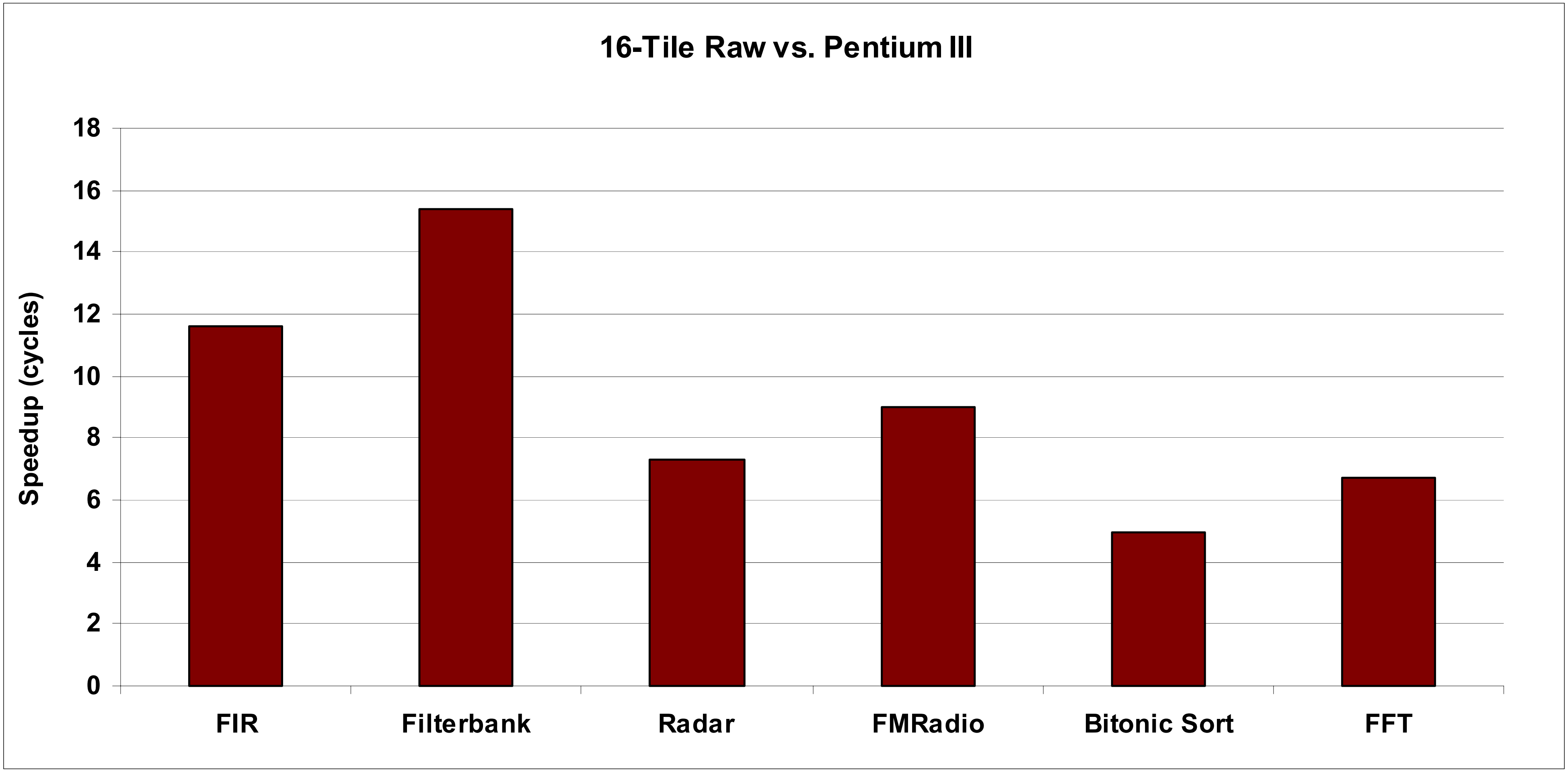








Raw vs. Pentium III



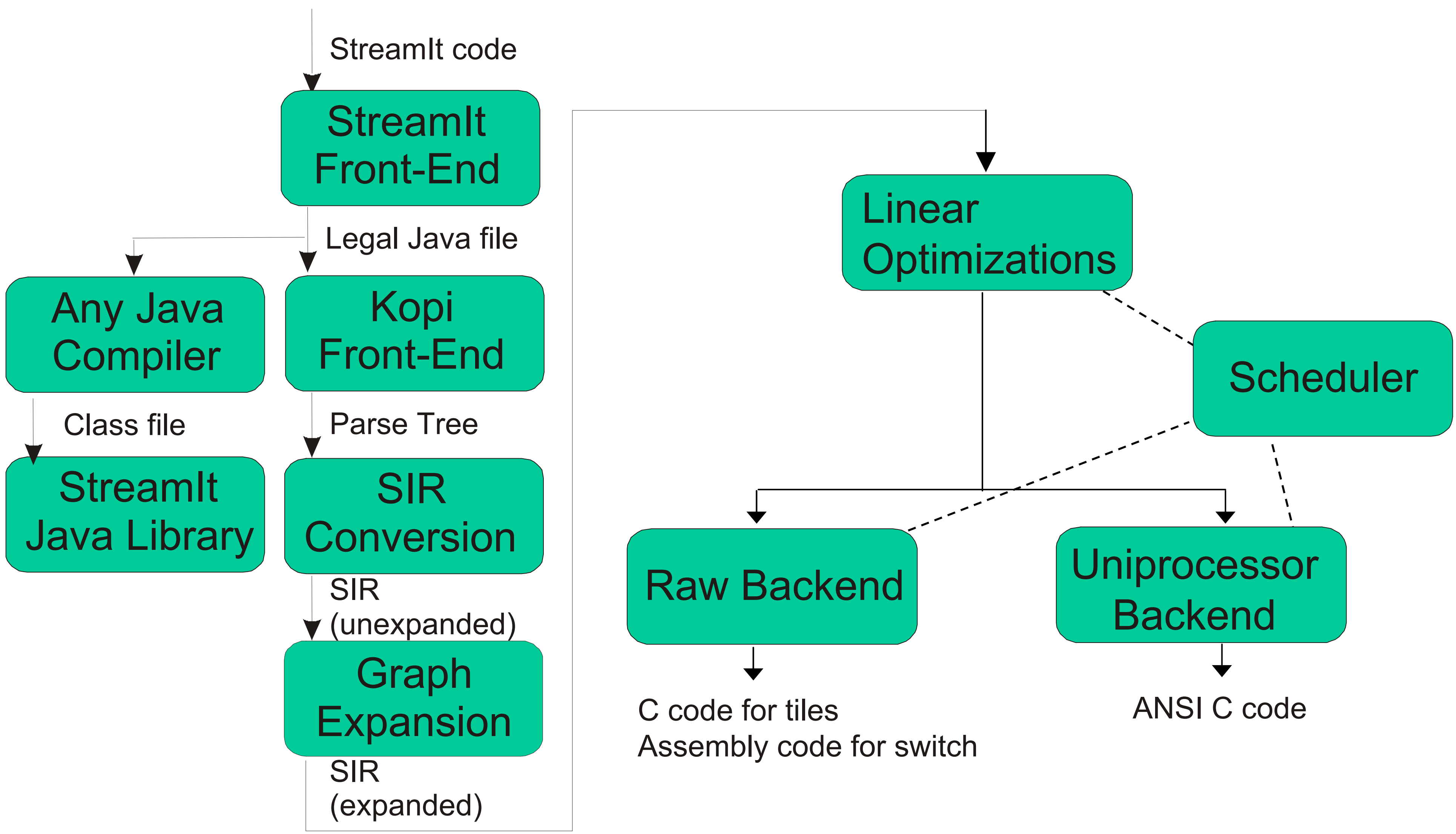
Outline

- Introduction
- StreamIt Language
- Domain-specific Optimizations
- Targeting Parallel Architectures
- Public Release
- Conclusions

StreamIt Compiler Infrastructure

- Built on Kopi Java compiler (GNU license)
  - StreamIt frontend is on MIT license
- High-level hierarchical IR for streams
  - Host of graph transformations
    - Filter fusion, filter fission
    - Synchronization removal
    - Splitjoin refactoring
    - Graph canonicalization
- Low-level “flat graph” for backends
  - Eliminates structure; point-to-point connections
- Streaming benchmark suite

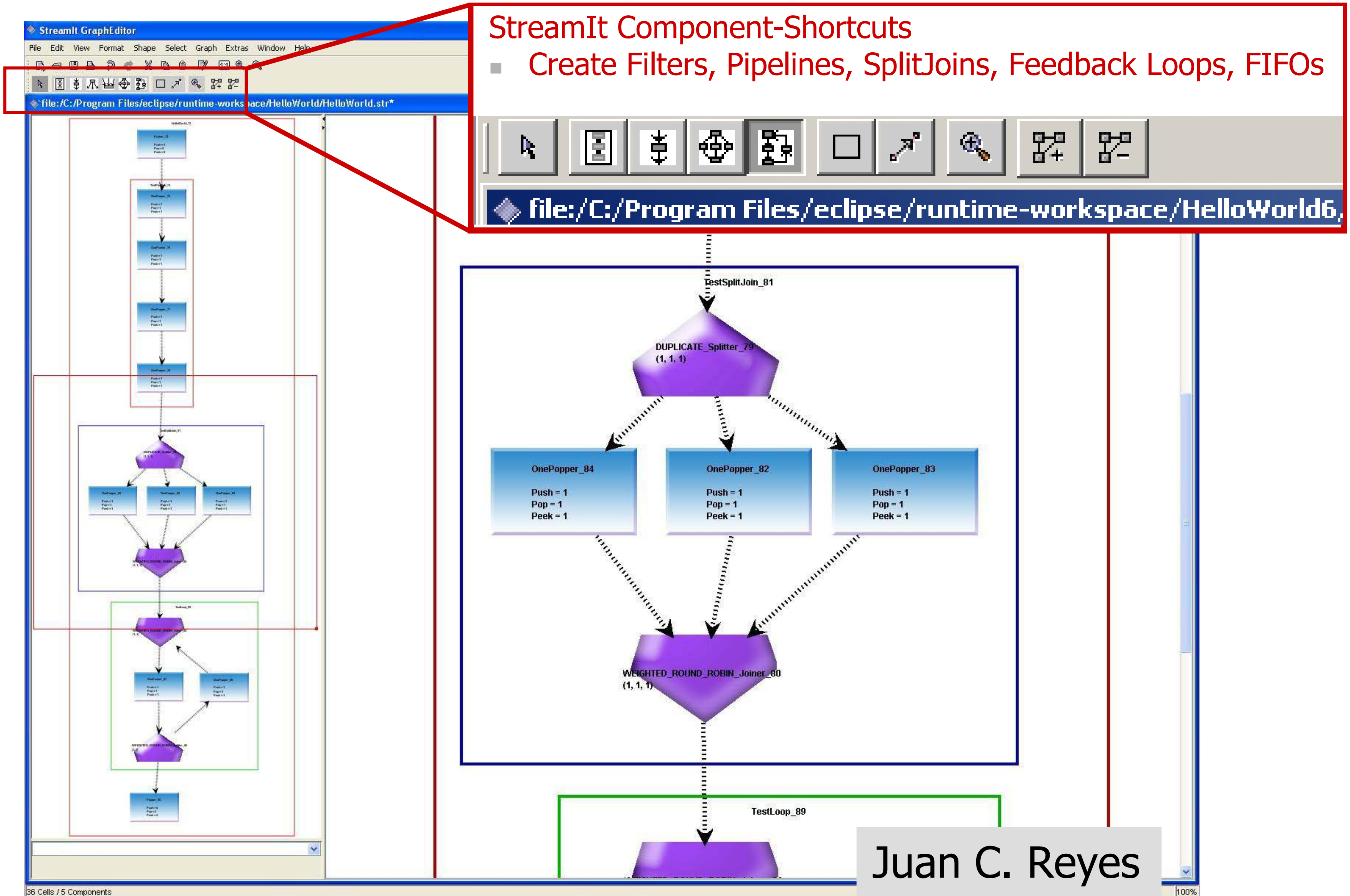
Compiler Flow



Building on StreamIt

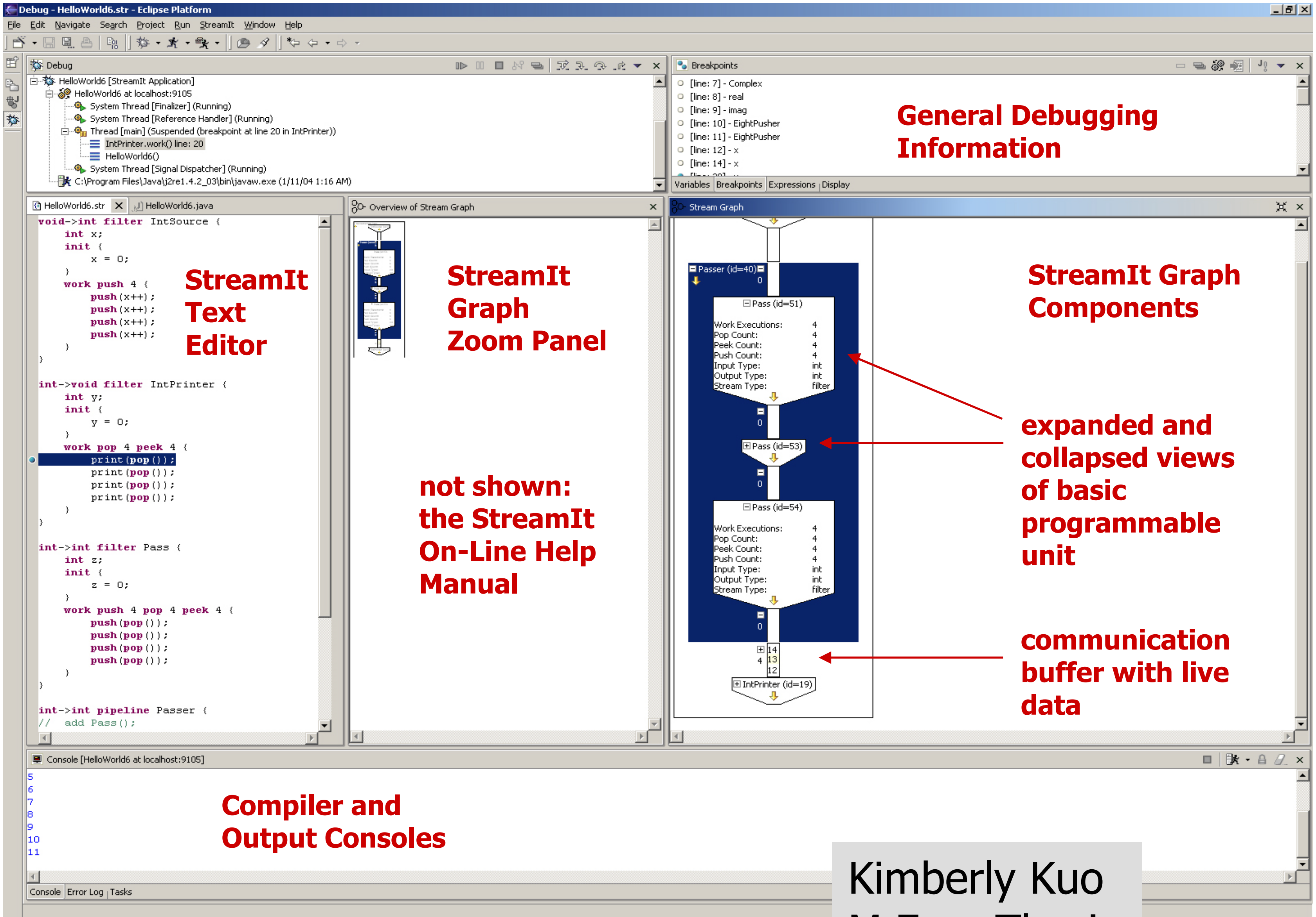
- StreamIt to VIRAM [Yelick et al.]
  - Automatically generate permutation instructions
- StreamBit: bit-level optimization [Bodik et al.]
- Integration with IBM Eclipse Platform

StreamIt Graphical Editor



Juan C. Reyes
M.Eng. Thesis

StreamIt Debugging Environment



Kimberly Kuo
M.Eng. Thesis

Outline

- Introduction
- StreamIt Language
- Domain-specific Optimizations
- Targeting Parallel Architectures
- Public Release
- Conclusions

Related Work

- Stream languages
  - KernelC/StreamC, Brook: augment C with data-parallel kernels
  - Cg: allow low-level programming of graphics processors
  - SISAL, functional languages: expose temporal parallelism
  - StreamIt exposes more task parallelism, easier to analyze
- Control languages for embedded systems
  - LUSTRE, Esterel, etc.: can verify of safety properties
  - Do not expose high-bandwidth data flow for optimization
- Prototyping environments
  - Ptolemy, Simulink, etc.: provide graphical abstractions
  - StreamIt has more of a compiler focus

Future Work

- Backend optimizations for linear filters
  - Template assembly code: asymptotically optimal
- Fault-tolerance on a cluster of workstations
  - Automatically recover if machine fail
- Supporting dynamic events
  - Point-to-point control messages
  - Re-initialization for parts of the stream

Conclusions

- StreamIt: compiler infrastructure for streams
  - Raising the level of abstraction in stream programming
  - Language design for both programmer and compiler

Compiler Analysis	Language features exploited
Linear Optimizations	- peek primitive (data reuse) - atomic work function - structured streams
Backend for Raw Architecture	- exposed communication patterns - exposed parallelism - static work estimate

- Public release: many opportunities for collaboration

http://cag.csail.mit.edu/streamit

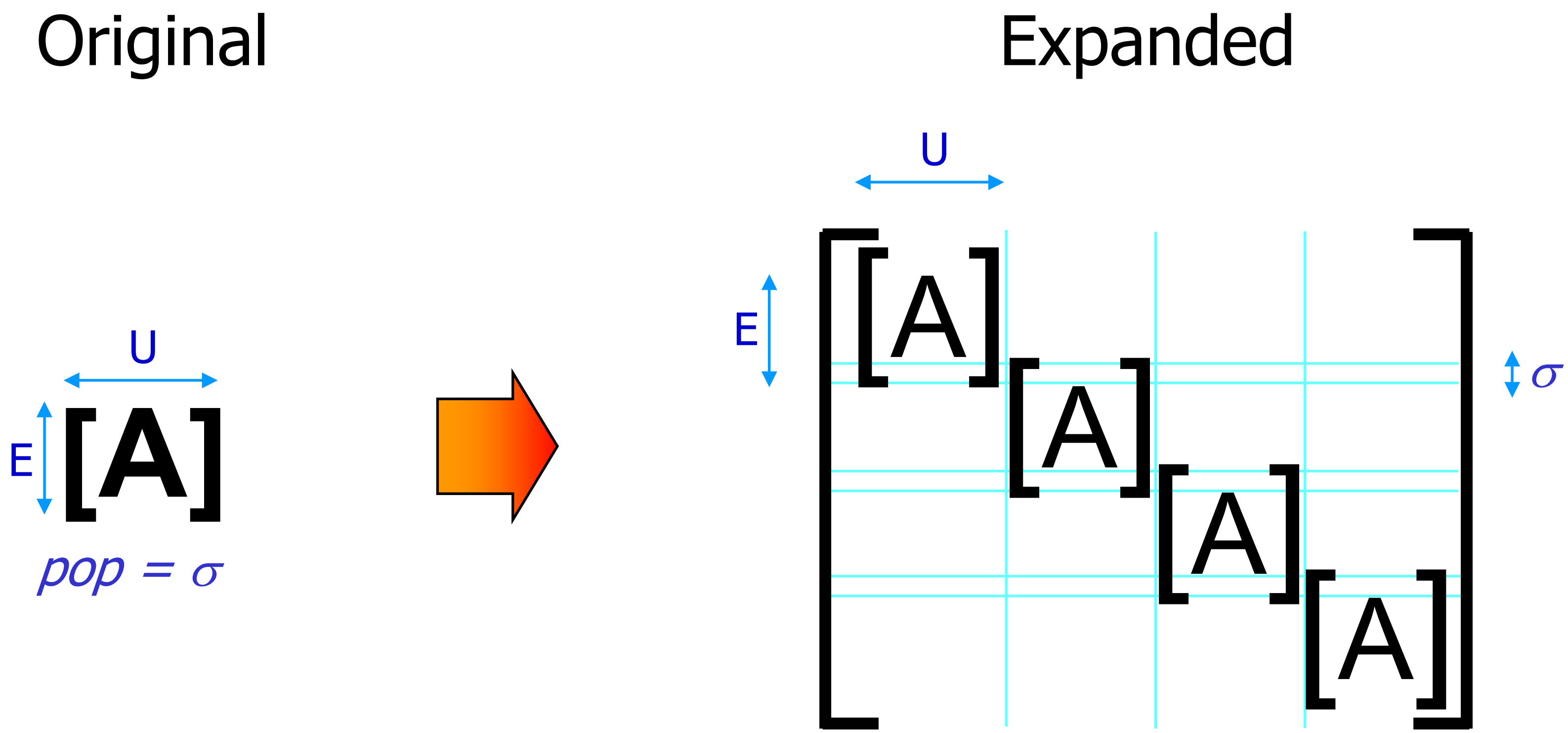
Extra Slides

StreamIt Language Summary

Feature	Programmer Benefit	Compiler Benefit
Autonomous Filters	• Encapsulation • Automatic scheduling	• Local memories • Exposes parallelism • Exposes communication
Peek primitive	• Automatic buffer management	• Exposes data reuse
Structured Streams	• Natural syntax	• Exposes symmetry • Eliminate corner cases
Scripts for Graph Construction	• Reusable components • Easy to maintain	• Two-stage compilation

AB for any A and B?

- Linear Expansion



Backend Support for Linear Filters

- Can generate custom code for linear filters
  - Many architectures have special support for matrix mult.
  - On Raw: assembly code templates for tiles and switch
    - Substitute coefficients, peek/pop/push rates
- Preliminary result: FIR on Raw
  - StreamIt code: 15 lines
  - Manually-tuned C code: 352 lines
  - Both achieve 99% utilization of Raw FPU's
    - Asymptotically optimal
- Current focus: integrating with general backend