Massachusetts Institute of Technology

CSAIL
MIT COMPUTER SCIENCE AND ARTIFICIAL INTELLIGENCE LABORATORY

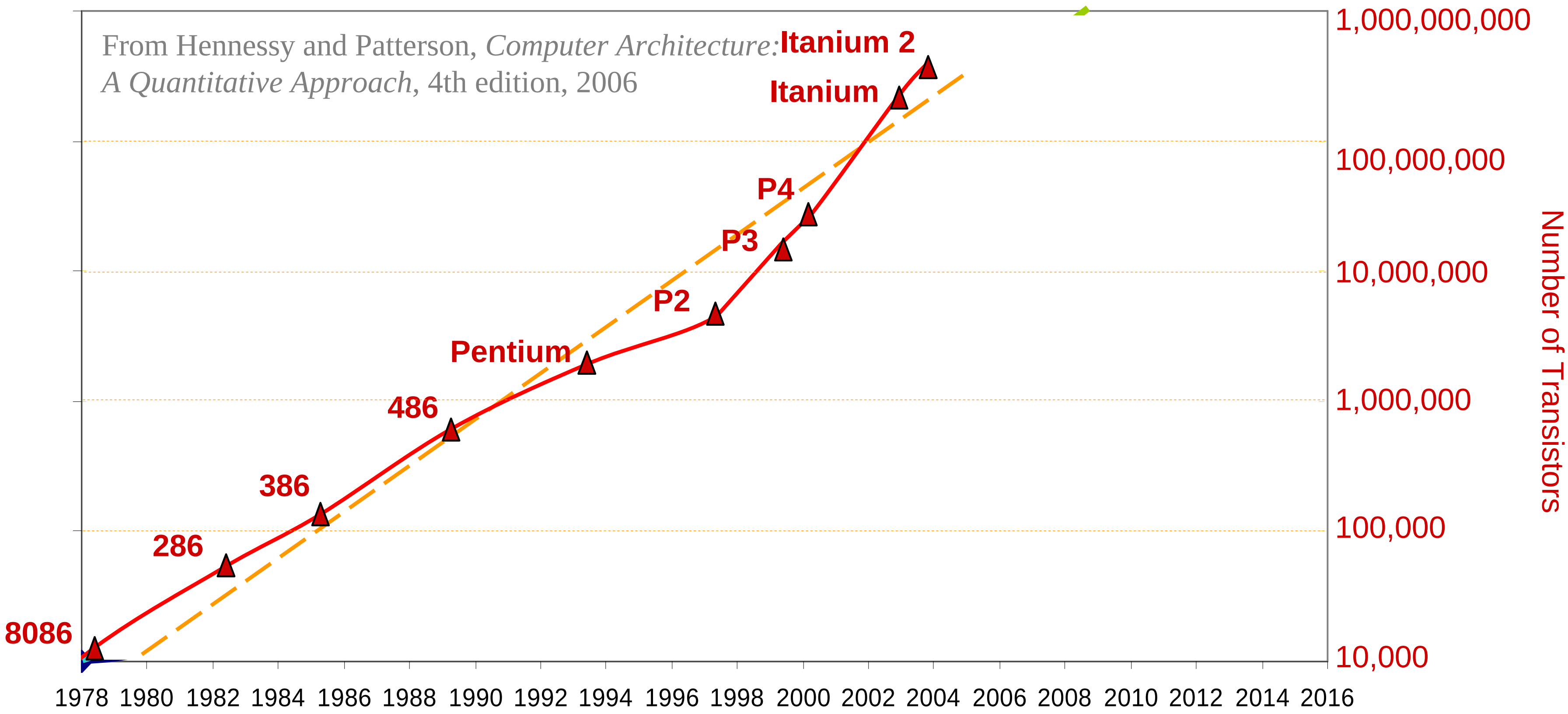
StreamIt – A Programming Language for the Era of Multicores

Saman Amarasinghe

StreamIt

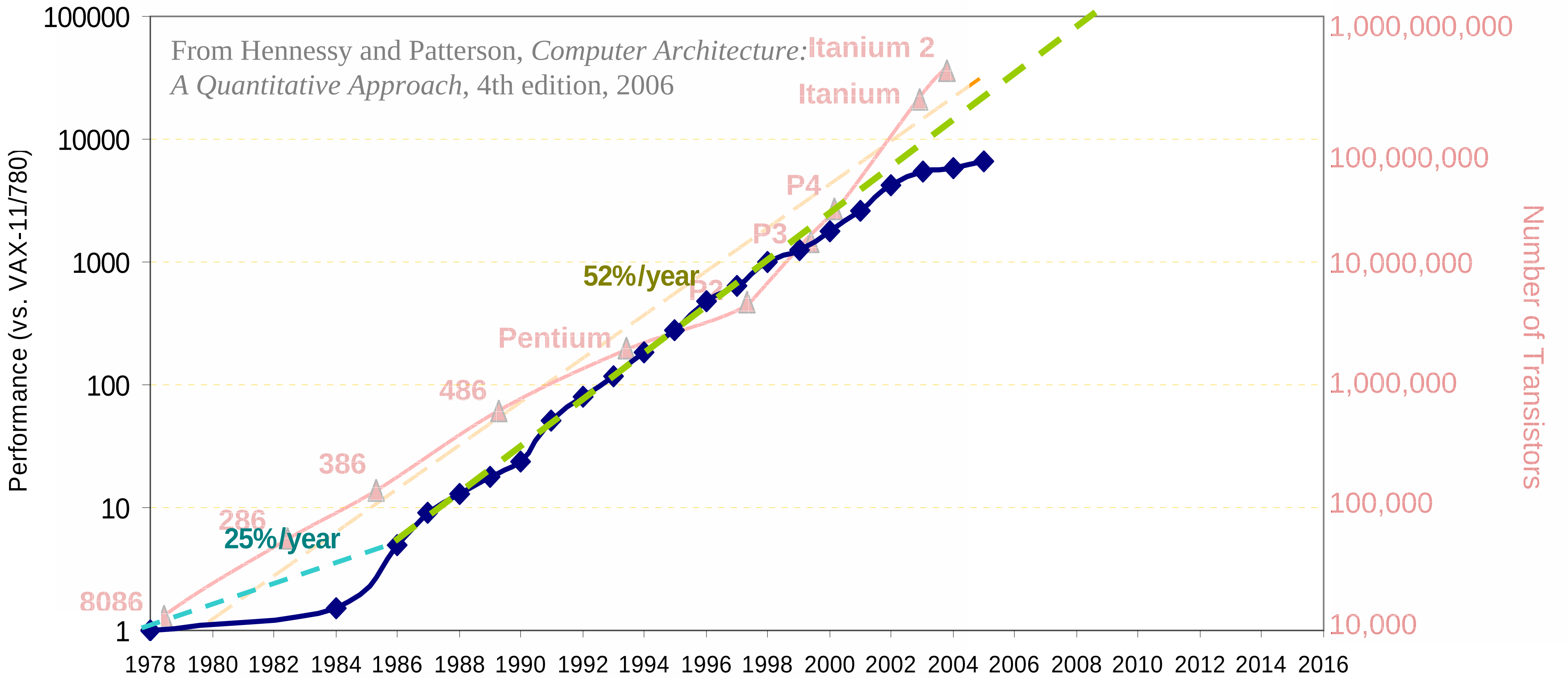
http://cag.csail.mit.edu/streamit

Moore's Law



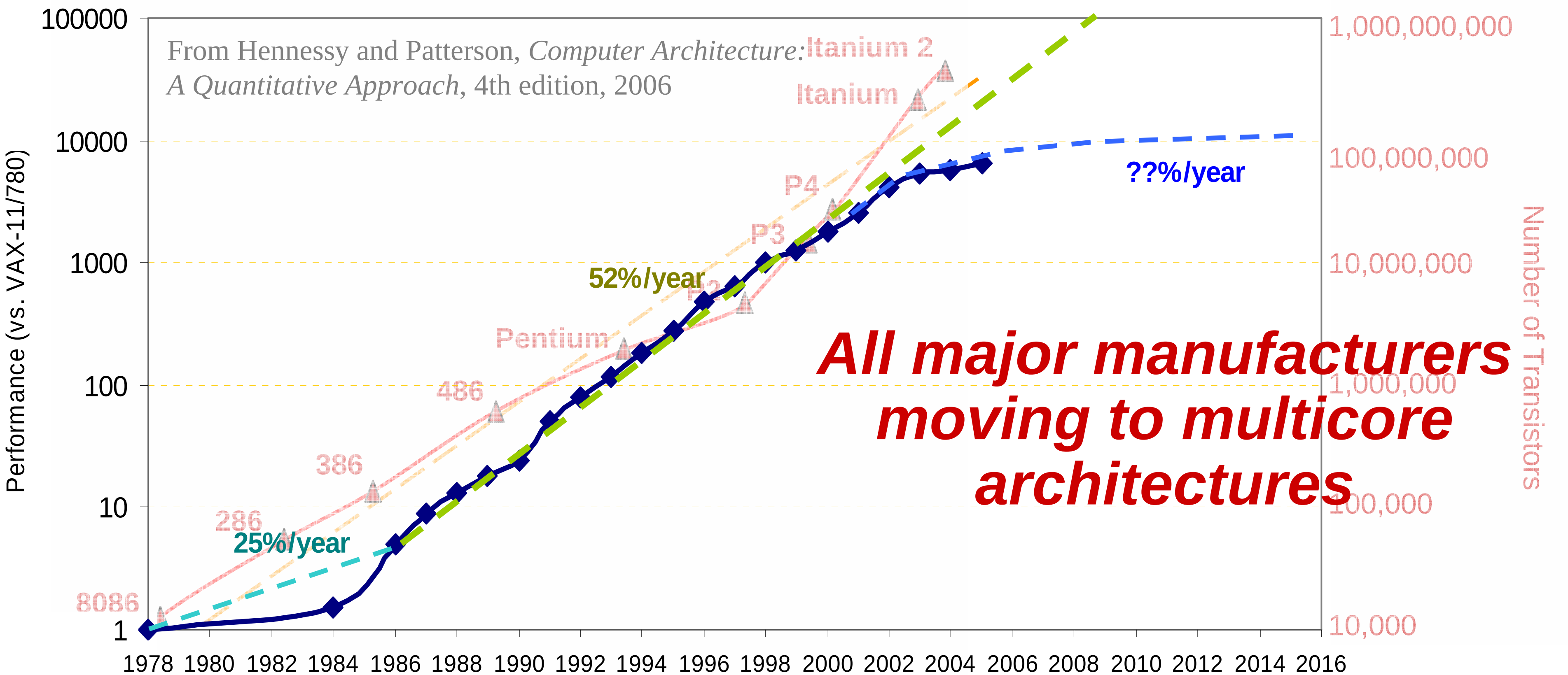
From David Patterson

Uniprocessor Performance (SPECint)



From David Patterson

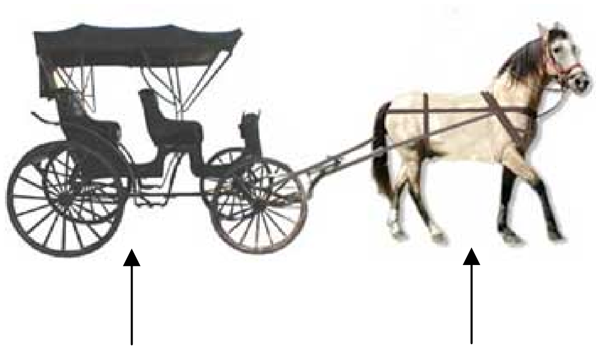
Uniprocessor Performance (SPECint)



- General-purpose unicores have stopped historic performance scaling
  - Power consumption
  - Wire delays
  - DRAM access latency
  - Diminishing returns of more instruction-level parallelism

From David Patterson

Programming Languages for Modern Architectures



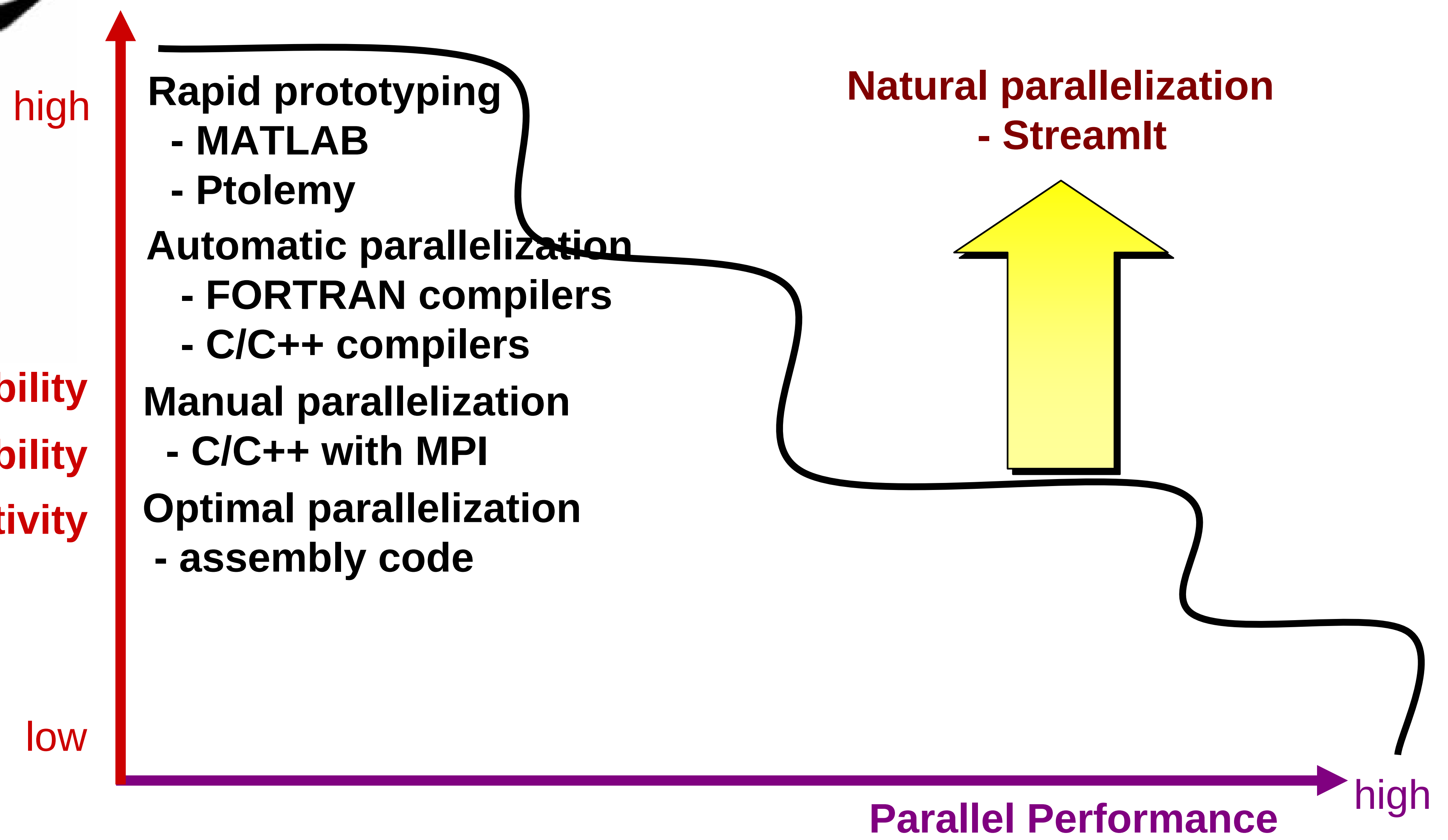
C ⇔ von-Neumann machine

Modern architecture

- Two choices:
  - Bend over backwards to support old languages like C/C++
  - Develop high-performance architectures that are hard to program

Parallel Programmer's Dilemma

$$F(u,v) = \frac{2}{N} C(u)C(v) \sum_{x=0}^{N-1} \sum_{y=0}^{N-1} f(x,y) \cos\frac{(2x+1)u\pi}{2N} \cos\frac{(2y+1)v\pi}{2N}$$

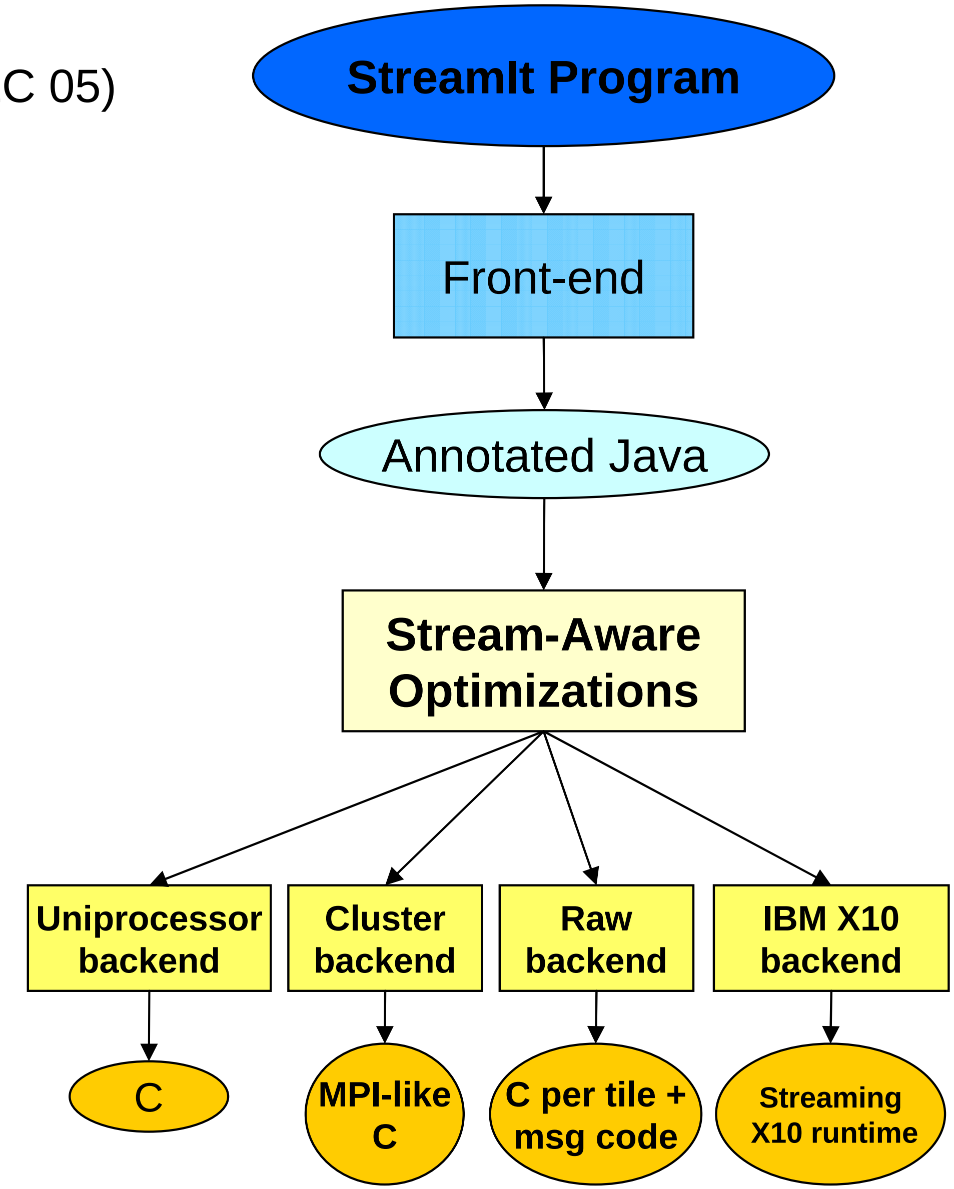


Stream Application Domain

- Graphics
- Cryptography
- Databases
- Object recognition
- Network processing and security
- Scientific codes
- …

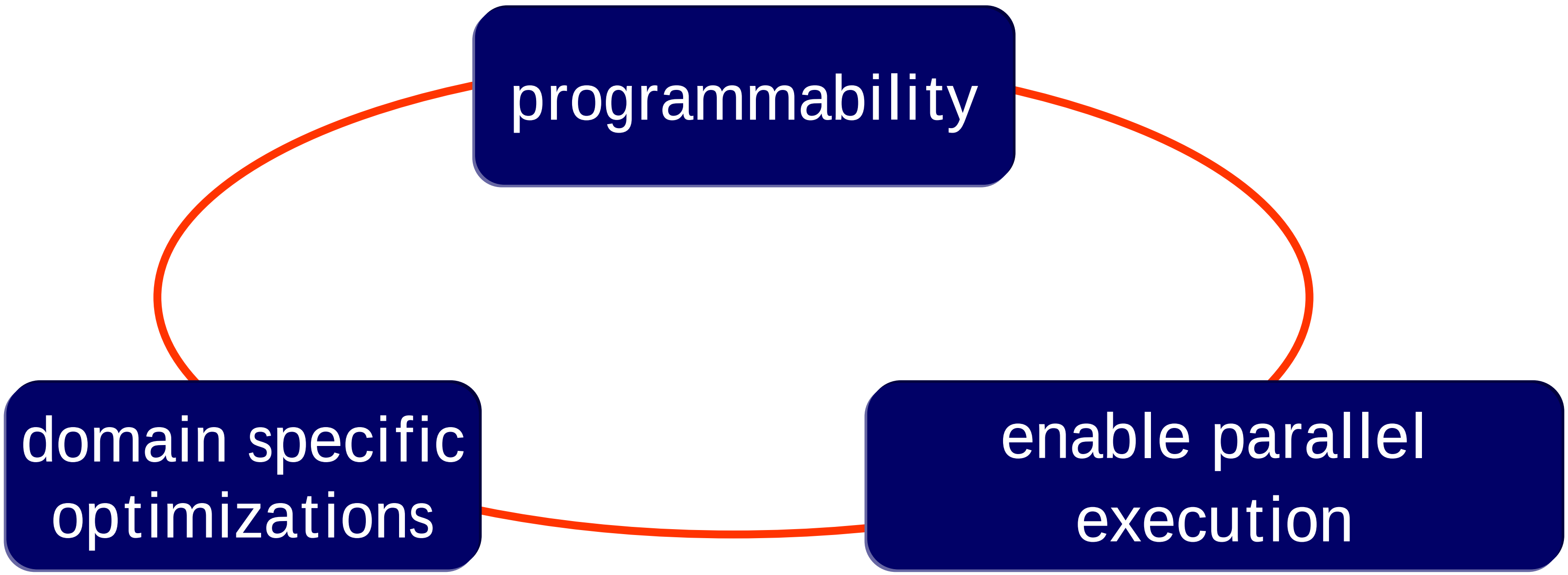
StreamIt Project

- **Language Semantics / Programmability**
  - StreamIt Language (CC 02)
  - Programming Environment in Eclipse (P-PHEC 05)
- **Optimizations / Code Generation**
  - Phased Scheduling (LCTES 03)
  - Cache Aware Optimization (LCTES 05)
- **Domain Specific Optimizations**
  - Linear Analysis and Optimization (PLDI 03)
  - Optimizations for bit streaming (PLDI 05)
  - Linear State Space Analysis (CASES 05)
- **Parallelism**
  - Teleport Messaging (PPOPP 05)
  - Compiling for Communication-Exposed Architectures (ASPLOS 02)
  - Load-Balanced Rendering (Graphics Hardware 05)
- **Applications**
  - SAR, DSP benchmarks, JPEG,
  - MPEG [IPDPS 06], DES and Serpent [PLDI 05], …



Compiler-Aware Language Design

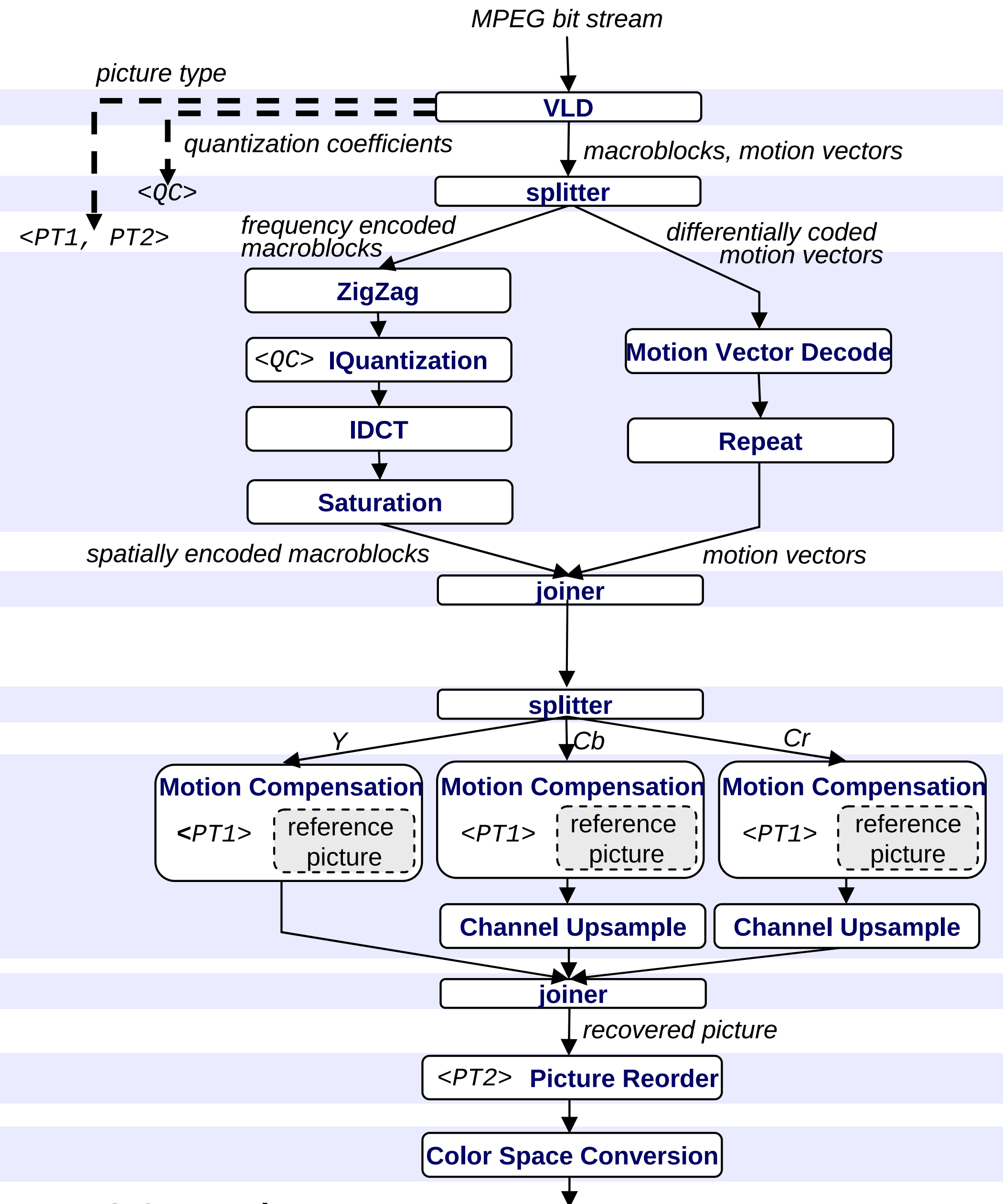
boost productivity, enable faster development and rapid prototyping



simple and effective optimizations for domain specific abstractions

target multicores, clusters, tiled architectures, DSPs, graphics processors, …

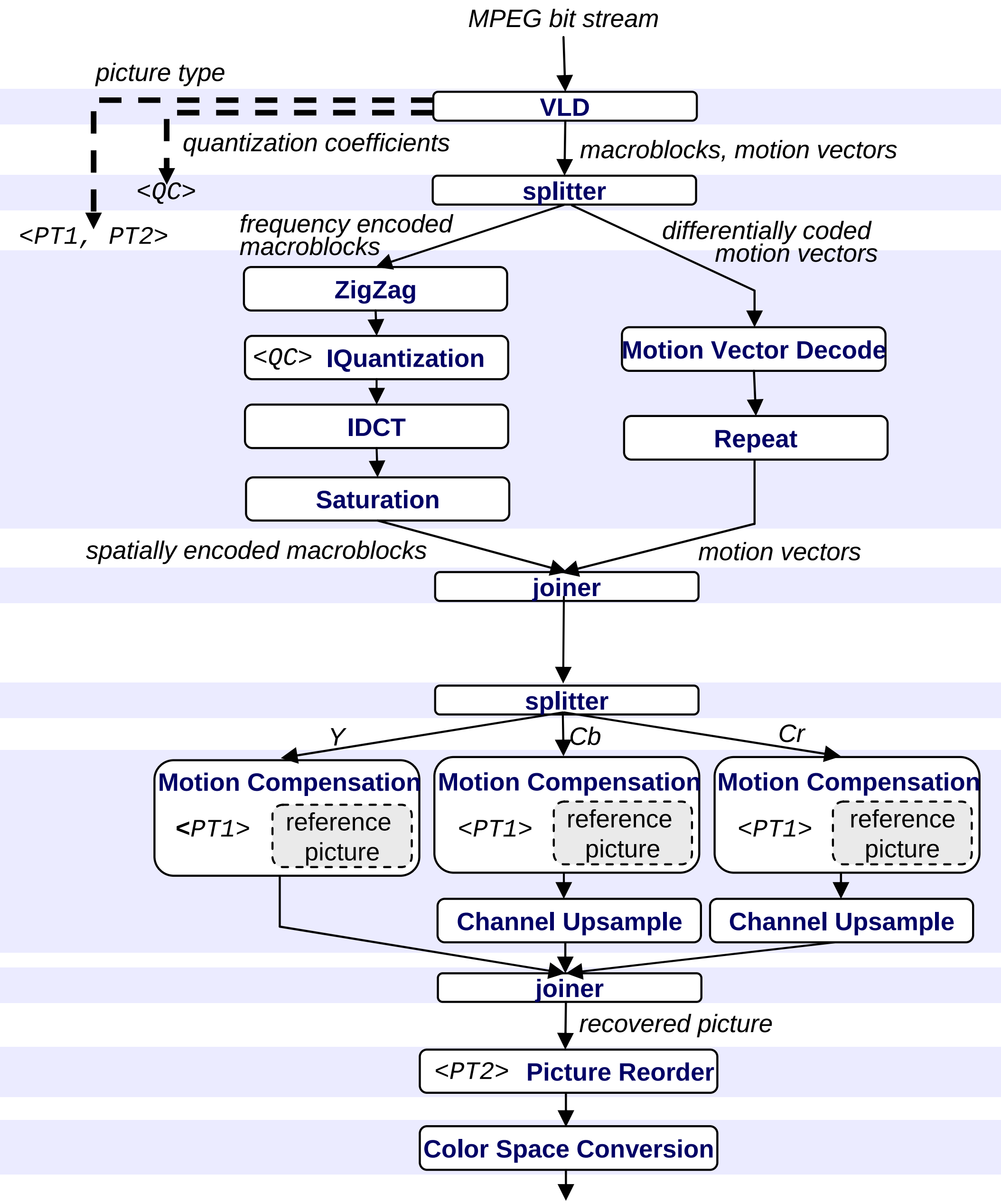
Streaming Application Design



MPEG-2 Decoder

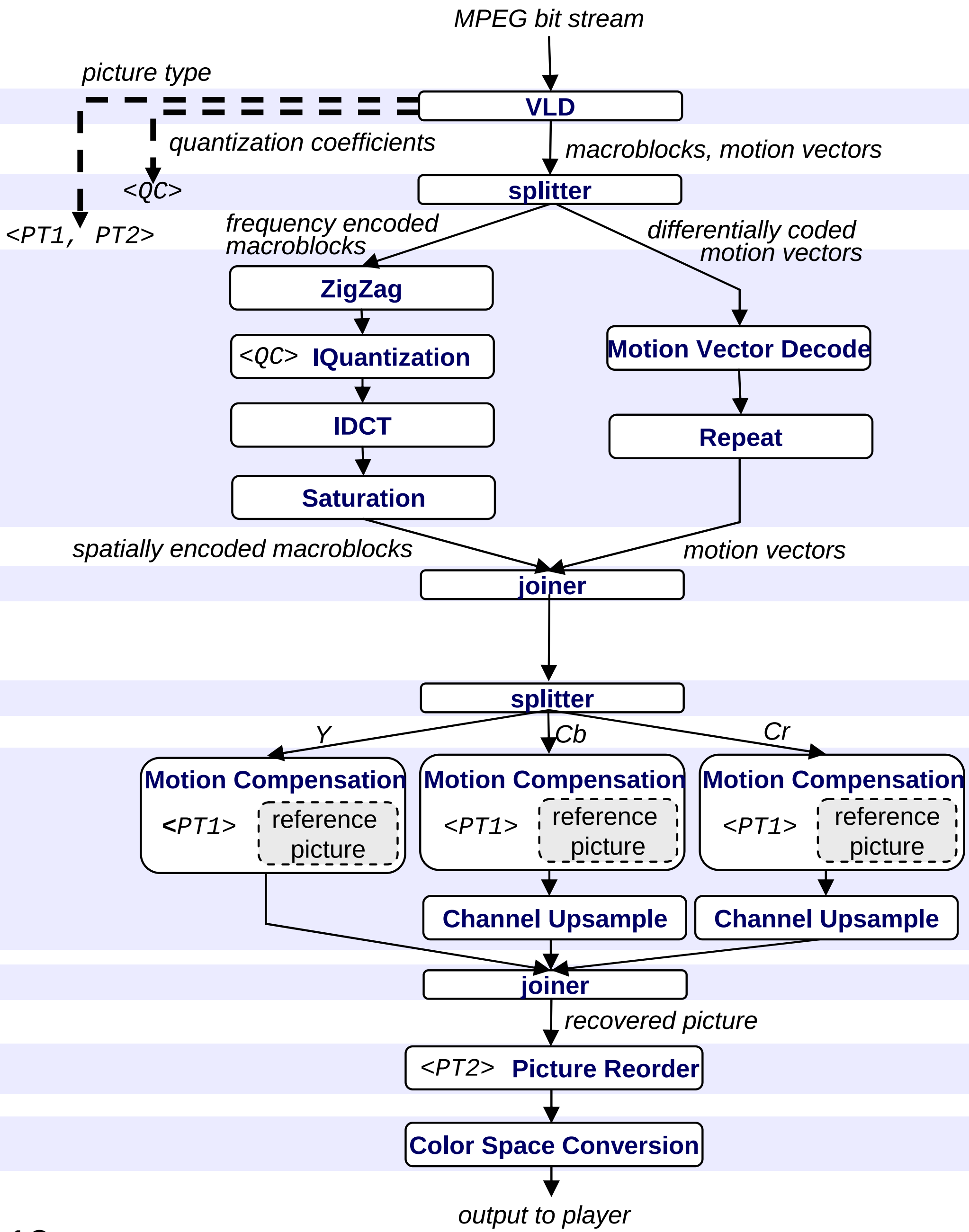
- Structured block level diagram describes computation and flow of data
- Conceptually easy to understand
  - Clean abstraction of functionality

StreamIt Philosophy



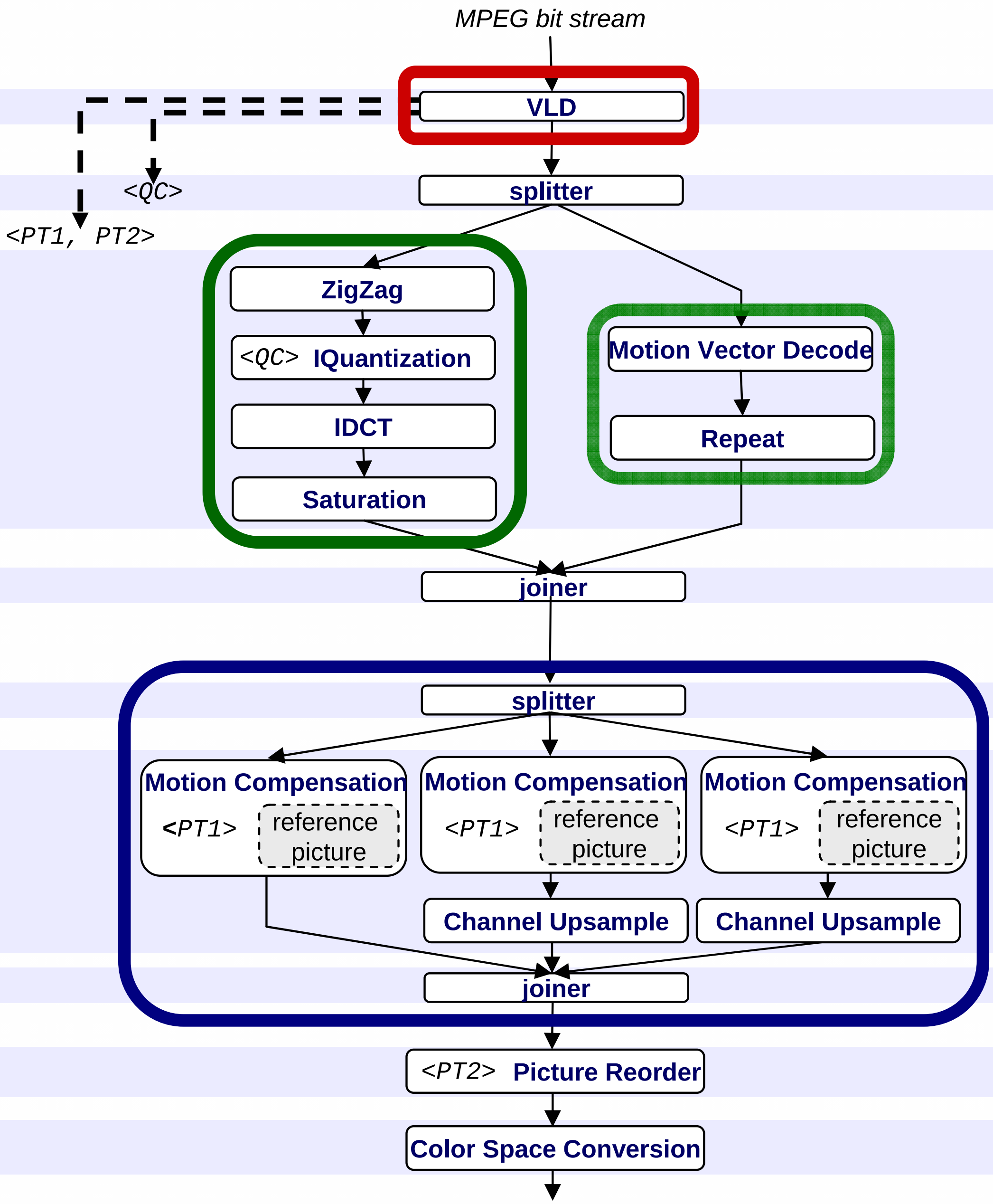
- Preserve program structure
  - Natural for application developers to express
- Leverage program structure to discover parallelism and deliver high performance
- Programs remain clean
  - Portable and malleable

StreamIt Philosophy




```
add VLD(QC, PT1, PT2);
add splitjoin {
    split roundrobin(N∗B, V);

    add pipeline {
        add ZigZag(B);
        add IQuantization(B) to QC;
        add IDCT(B);
        add Saturation(B);
    }
    add pipeline {
        add MotionVectorDecode();
        add Repeat(V, N);
    }

    join roundrobin(B, V);
}
add splitjoin {
    split roundrobin(4∗(B+V), B+V, B+V);

    add MotionCompensation(4∗(B+V)) to PT1;
    for (int i = 0; i < 2; i++) {
        add pipeline {
            add MotionCompensation(B+V) to PT1;
            add ChannelUpsample(B);
        }
    }

    join roundrobin(1, 1, 1);
}
add PictureReorder(3∗W∗H) to PT2;

add ColorSpaceConversion(3∗W∗H);
```

Stream Abstractions in StreamIt




```
add VLD(QC, PT1, PT2);
add splitjoin {
    split roundrobin(N∗B, V);

    add pipeline {
        add ZigZag(B);
        add IQuantization(B) to QC;
        add IDCT(B);
        add Saturation(B);
    }
    add pipeline {
        add MotionVectorDecode();
        add Repeat(V, N);
    }

    join roundrobin(B, V);
}
add splitjoin {
    split roundrobin(4∗(B+V), B+V, B+V);

    add MotionCompensation(4∗(B+V)) to PT1;
    for (int i = 0; i < 2; i++) {
        add pipeline {
            add MotionCompensation(B+V) to PT1;
            add ChannelUpsample(B);
        }
    }

    join roundrobin(1, 1, 1);
}
add PictureReorder(3∗W∗H) to PT2;

add ColorSpaceConversion(3∗W∗H);
```

filters

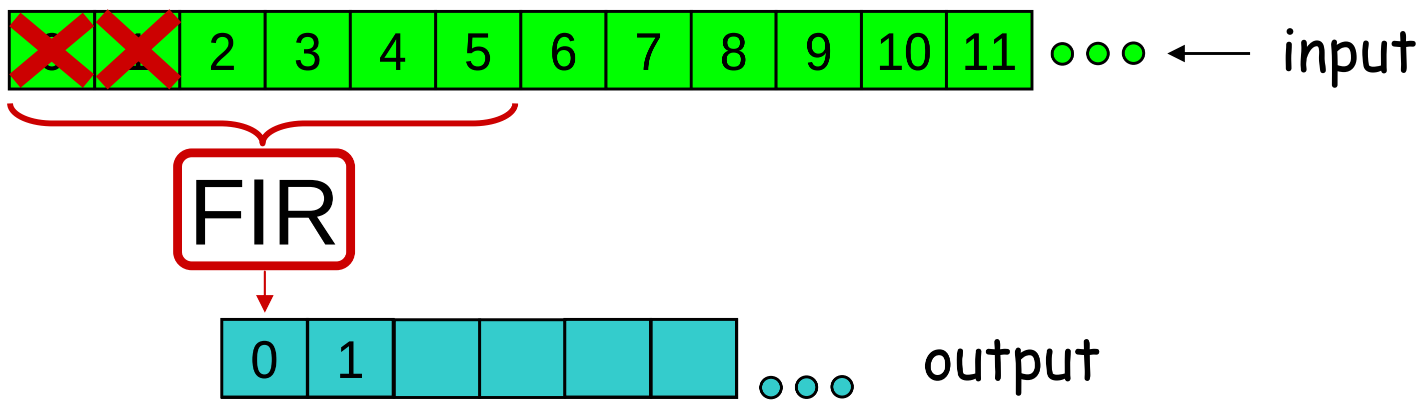
pipelines

splitjoins

StreamIt Language Highlights

- Filters
- Pipelines
- Splitjoins
- Teleport messaging

Example StreamIt Filter




```
float→float filter FIR (int N) {

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

FIR Filter in C

- FIR functionality obscured by buffer management details

- Programmer must commit to a particular buffer implementation strategy

```
void FIR(
  int* src,
  int* dest,
  int* srcIndex,
  int* destIndex,
  int srcBufferSize,
  int destBufferSize,
  int N) {

  float result = 0.0;
  for (int i = 0; i < N; i++) {
    result += weights[i] * src[(*srcIndex + i) % srcBufferSize];
  }
  dest[*destIndex] = result;
  *srcIndex = (*srcIndex + 1) % srcBufferSize;
  *destIndex = (*destIndex + 1) % destBufferSize;
}
```

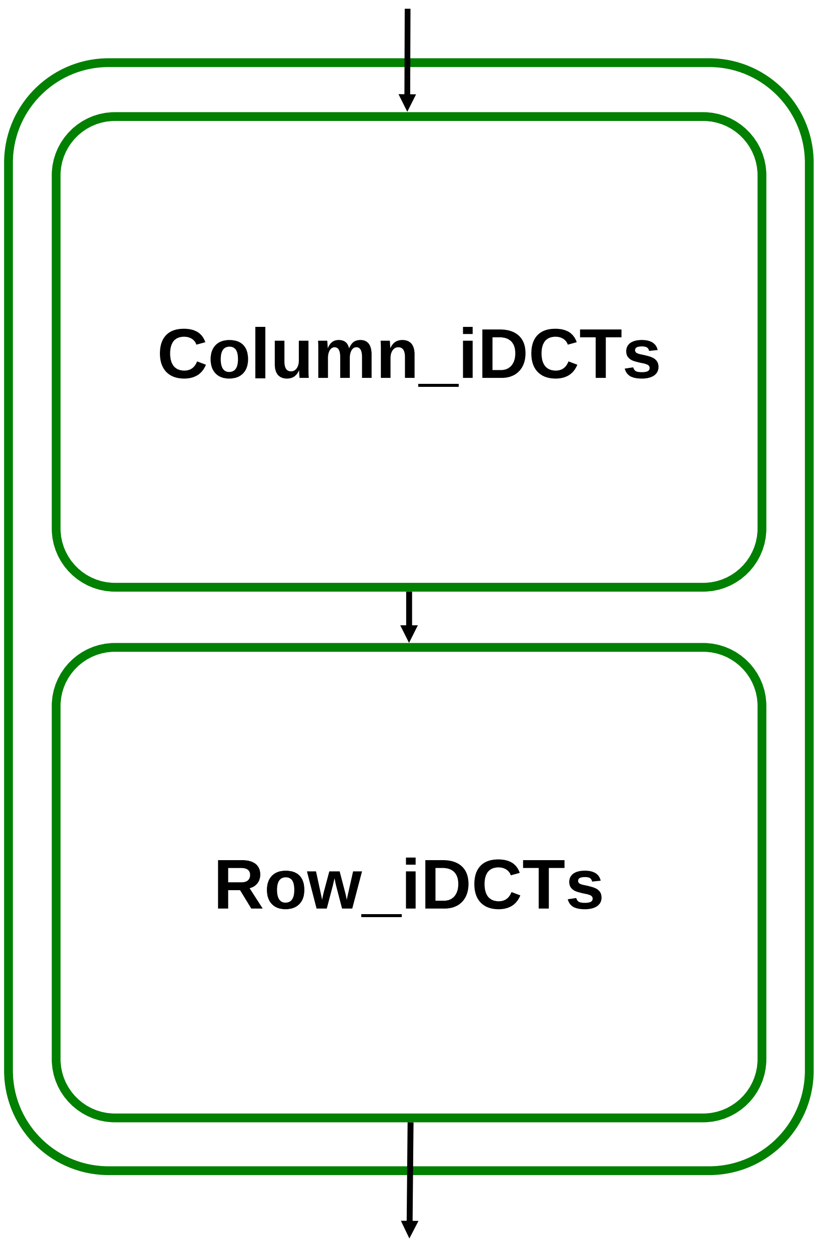
StreamIt Language Highlights

- Filters
- Pipelines
- Splitjoins
- Teleport messaging

Example StreamIt Pipeline

- Pipeline
  - Connect components in sequence
  - Expose pipeline parallelism

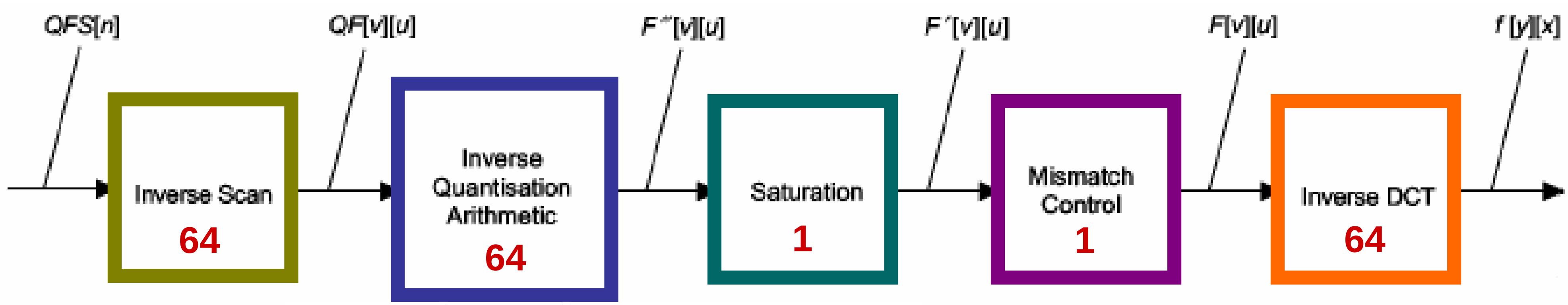
```
float→float pipeline 2D_iDCT (int N)
{

        add Column_iDCTs(N);
        add Row_iDCTs(N);

}
```



Preserving Program Structure

```
int->int pipeline BlockDecode(
         portal<InverseQuantisation> quantiserData,
         portal<MacroblockType> macroblockType) {
    add ZigZagUnordering();
    add InverseQuantization() to quantiserData, macroblockType;
    add Saturation(-2048, 2047);
    add MismatchControl();
    add 2D_iDCT(8);
    add Saturation(-256, 255);
}
```

Can be reused for JPEG decoding



From Figures 7-1 and 7-4 of the MPEG-2 Specification (ISO 13818-2, P. 61, 66)

In Contrast: C Code Excerpt

```
EXTERN unsigned char *backward_reference_frame[3];
EXTERN unsigned char *forward_reference_frame[3];
EXTERN unsigned char *current_frame[3];
...etc...
```

```
Decode_Picture {
  for (;;) {
    parser()
    for (;;) {
      decode_macroblock();
      motion_compensation();
      if (condition)
        then break;
    }
  }
  frame_reorder();
}
```

```
decode_macroblock() {
  parser();
  motion_vectors();
  for (comp=0;comp<block_count;comp++) {
    parser();
    Decode_MPEG2_Block();
  }
}
```

```
motion_vectors() {
  parser();
  decode_motion_vector
  parser();
}
```

```
motion_compensation() {
  for (channel=0;channel<3;channel++)
    form_component_prediction();
  for (comp=0;comp<block_count;comp++) {
    Saturate();
    IDCT();
    Add_Block();
  }
}
```

```
Decode_MPEG2_Block() {
  for (int i = 0;; i++) {
    parsing();
    ZigZagUnordering();
    inverseQuantization();
    if (condition) then
      break;
  }
}
```

- Explicit for-loops iterate through picture frames
- Frames passed through global arrays, handled with pointers
- Mixing of parser, motion compensation, and spatial decoding

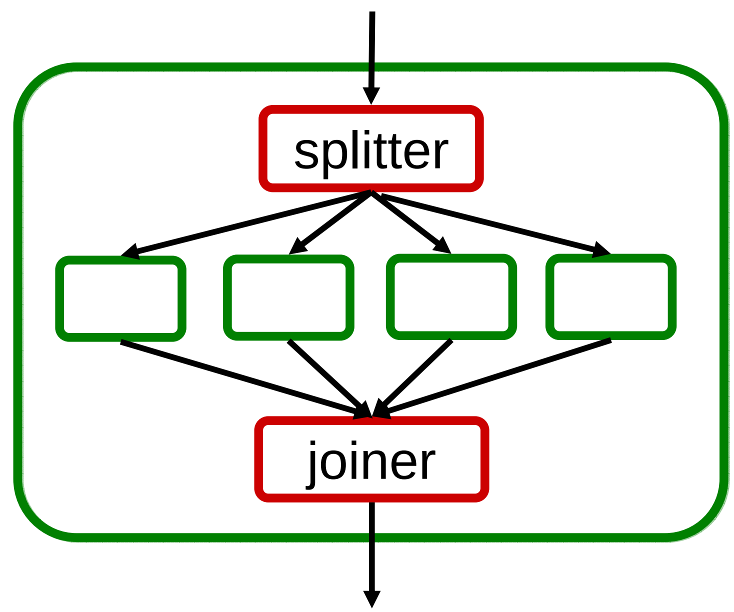
StreamIt Language Highlights

- Filters
- Pipelines
- Splitjoins
- Teleport messaging

Example StreamIt Splitjoin

- Splitjoin
  - Connect components in parallel
  - Expose task parallelism and data distribution

```
float→float splitjoin Row_iDCT (int N)
{
    split roundrobin(N);
    for (int i = 0; i < N; i++) {
        add 1D_iDCT(N);
    }
    join roundrobin(N);
}
```

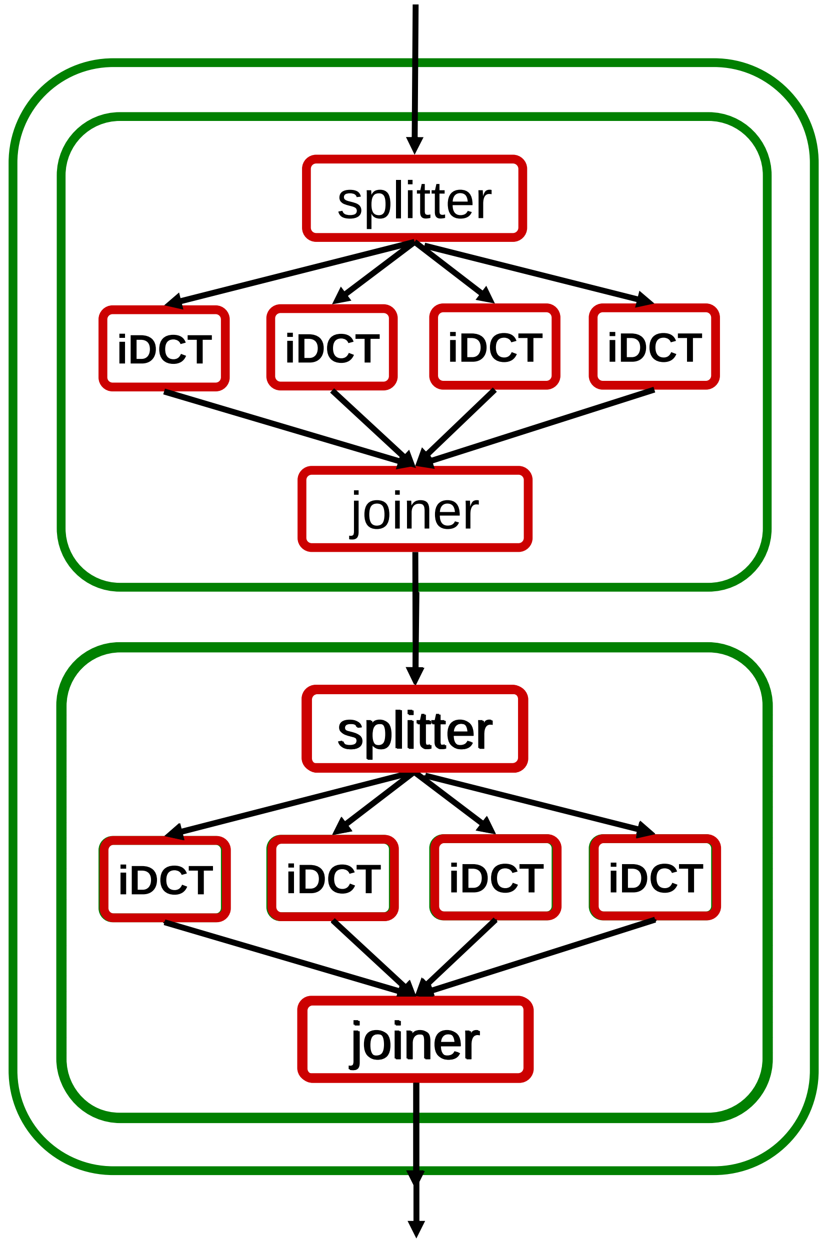


Example StreamIt Splitjoin

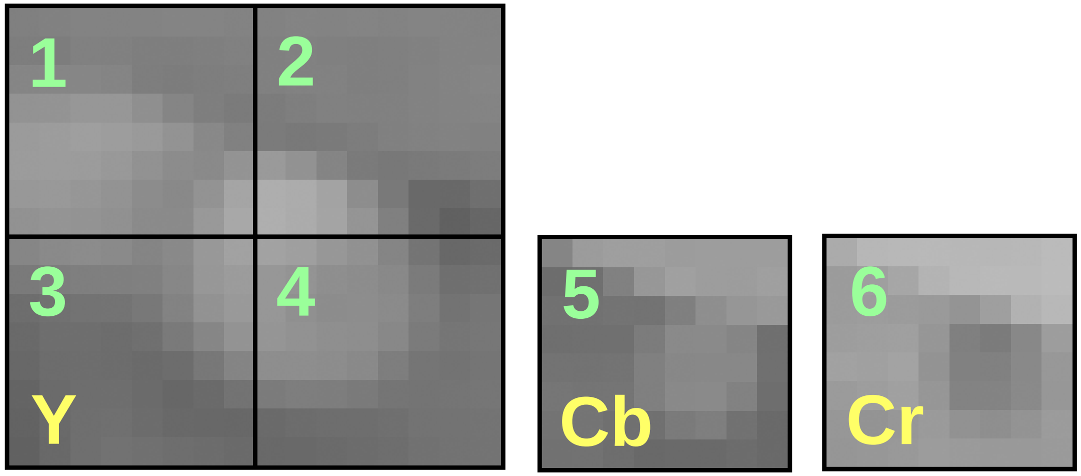
```
float→float pipeline 2D_iDCT (int N)
{
        add Column_iDCTs(N);
        add Row_iDCTs(N);
}
```

```
float→float splitjoin Column_iDCT (int N)
{
    split roundrobin(1);
    for (int i = 0; i < N; i++) {
        add 1D_iDCT(N);
    }
    join roundrobin(1);
}
```

```
float→float splitjoin Row_iDCT (int N)
{
    split roundrobin(N);
    for (int i = 0; i < N; i++) {
        add 1D_iDCT(N);
    }
    join roundrobin(N);
}
```



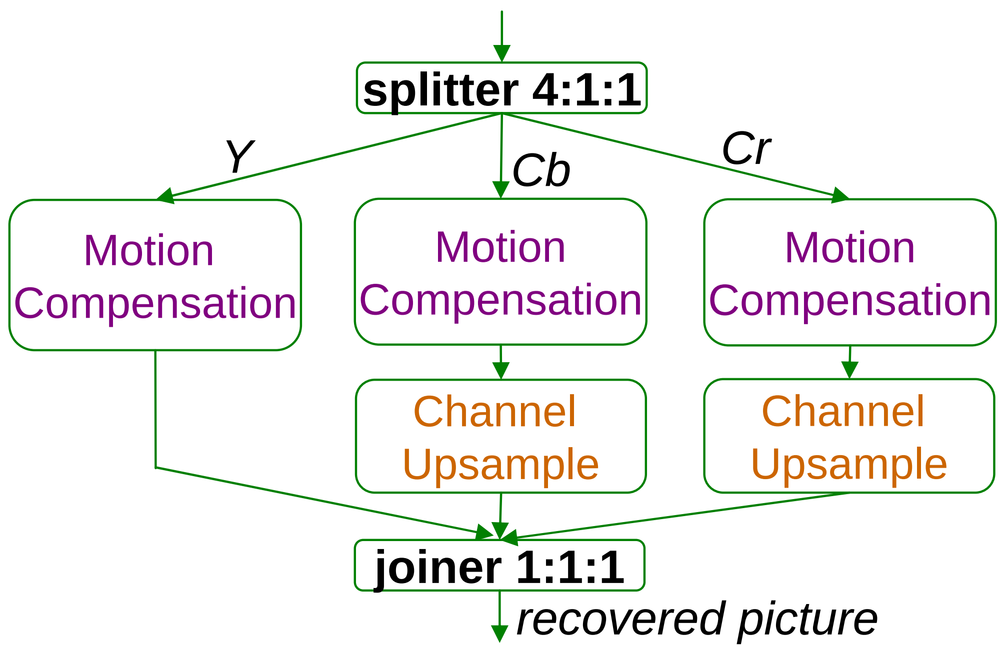
Naturally Expose Data Distribution



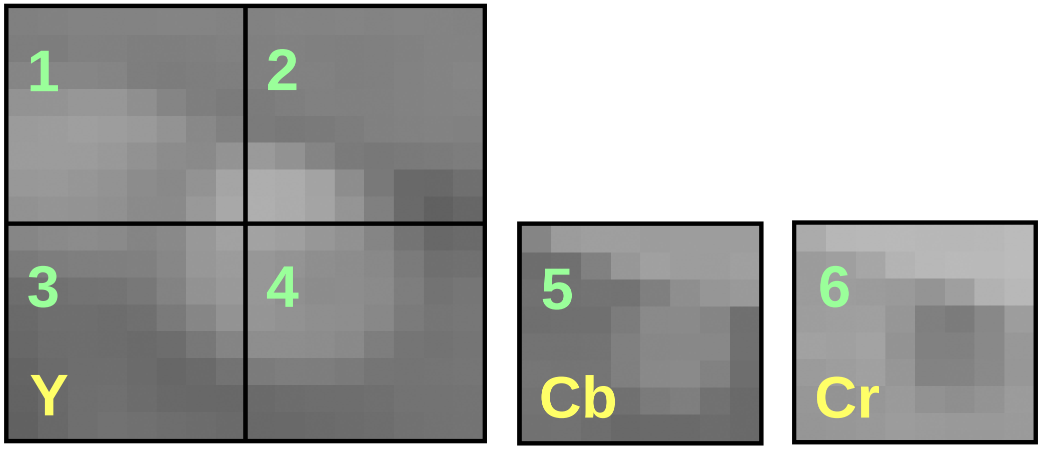
scatter macroblocks according to chroma format

```
add splitjoin {
   split roundrobin(4*(B+V), B+V, B+V);

   add MotionCompensation();
   for (int i = 0; i < 2; i++) {
       add pipeline {
           add MotionCompensation();
           add ChannelUpsample(B);
       }
   }

   join roundrobin(1, 1, 1);
}
```

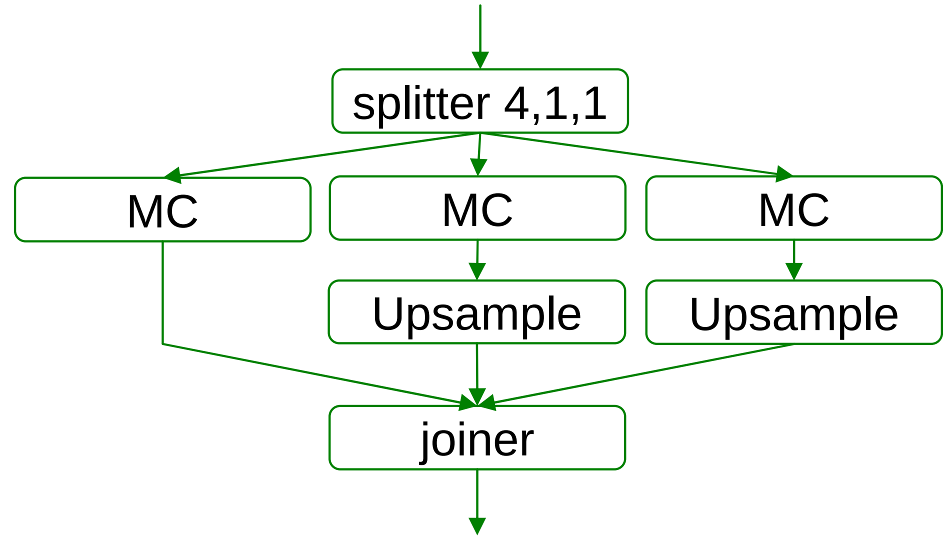
gather one pixel at a time

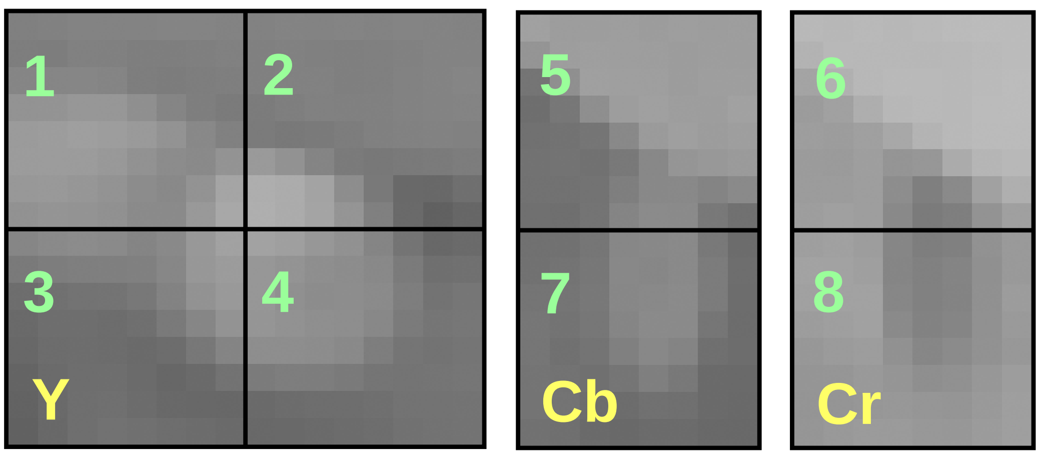


Stream Graph Malleability

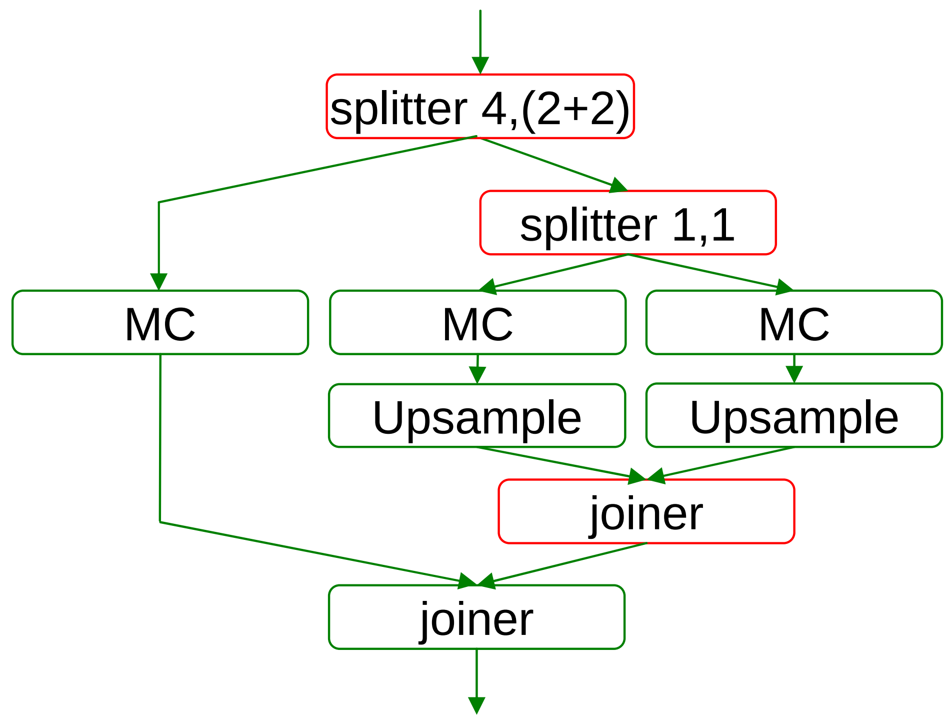


4:2:0 chroma format





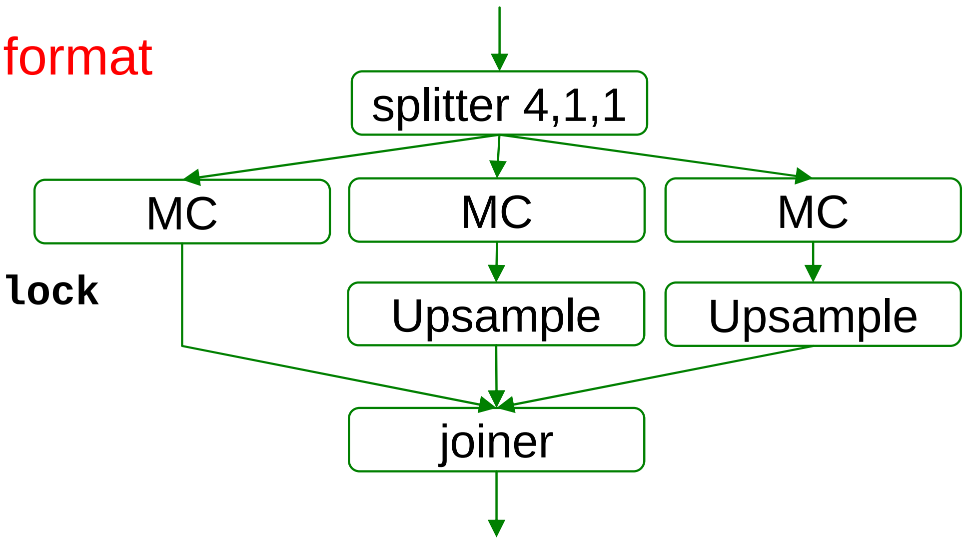
4:2:2 chroma format

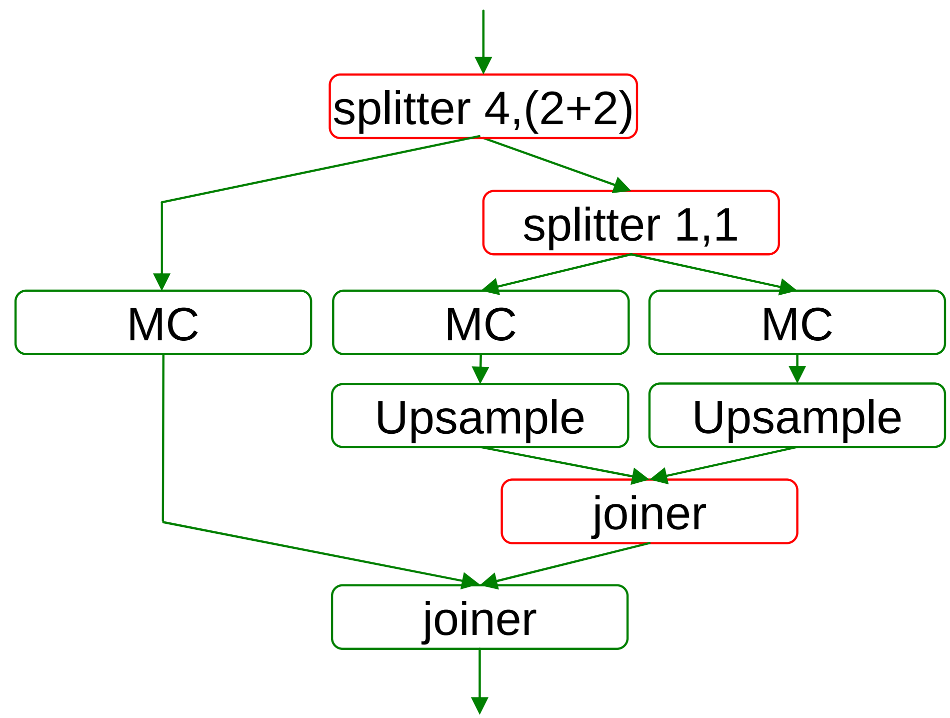


StreamIt Code Sample

red = code added or modified to support 4:2:2 format

```
// C = blocks per chroma channel per macroblock
// C = 1 for 4:2:0, C = 2 for 4:2:2
add splitjoin {
   split roundrobin(4*(B+V), 2*C*(B+V));

   add MotionCompensation();
   add splitjoin {
      split roundrobin(B+V, B+V);

      for (int i = 0; i < 2; i++) {
         add pipeline {
            add MotionCompensation()
            add ChannelUpsample(C,B);
         }
      }

      join roundrobin(1, 1);
   }

   join roundrobin(1, 1, 1);
}
```





In Contrast: C Code Excerpt

red = pointers used for address calculations

```
/* Y */
form_component_prediction(src[0]+(sfield?lx2>>1:0),dst[0]+(dfield?lx2>>1:0),
                          lx,lx2,w,h,x,y,dx,dy,average_flag);

if (chroma_format!=CHROMA444)  {
   lx>>=1; lx2>>=1; w>>=1; x>>=1; dx/=2;
}
if (chroma_format==CHROMA420)  {
   h>>=1; y>>=1; dy/=2;
}

/* Cb */
form_component_prediction(src[1]+(sfield?lx2>>1:0),dst[1]+(dfield?lx2>>1:0),
                          lx,lx2,w,h,x,y,dx,dy,average_flag);

/* Cr */
form_component_prediction(src[2]+(sfield?lx2>>1:0),dst[2]+(dfield?lx2>>1:0),
                          lx,lx2,w,h,x,y,dx,dy,average_flag);
```

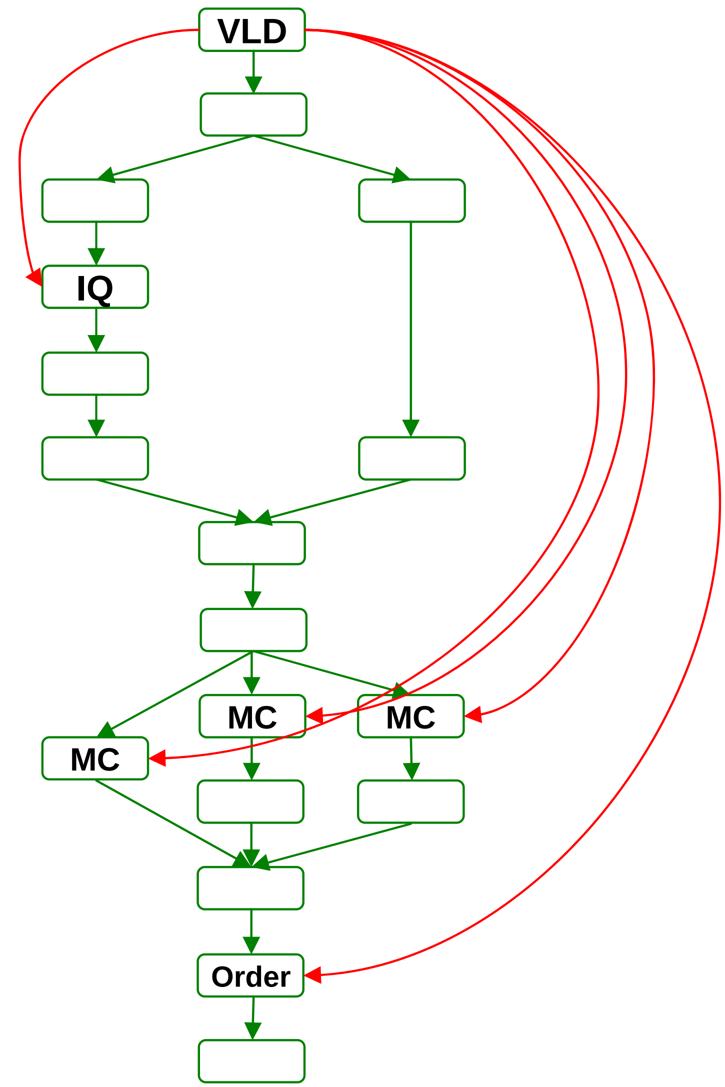
Adjust values used for address calculations depending on the chroma format used.

StreamIt Language Highlights

- Filters
- Pipelines
- Splitjoins
- Teleport messaging

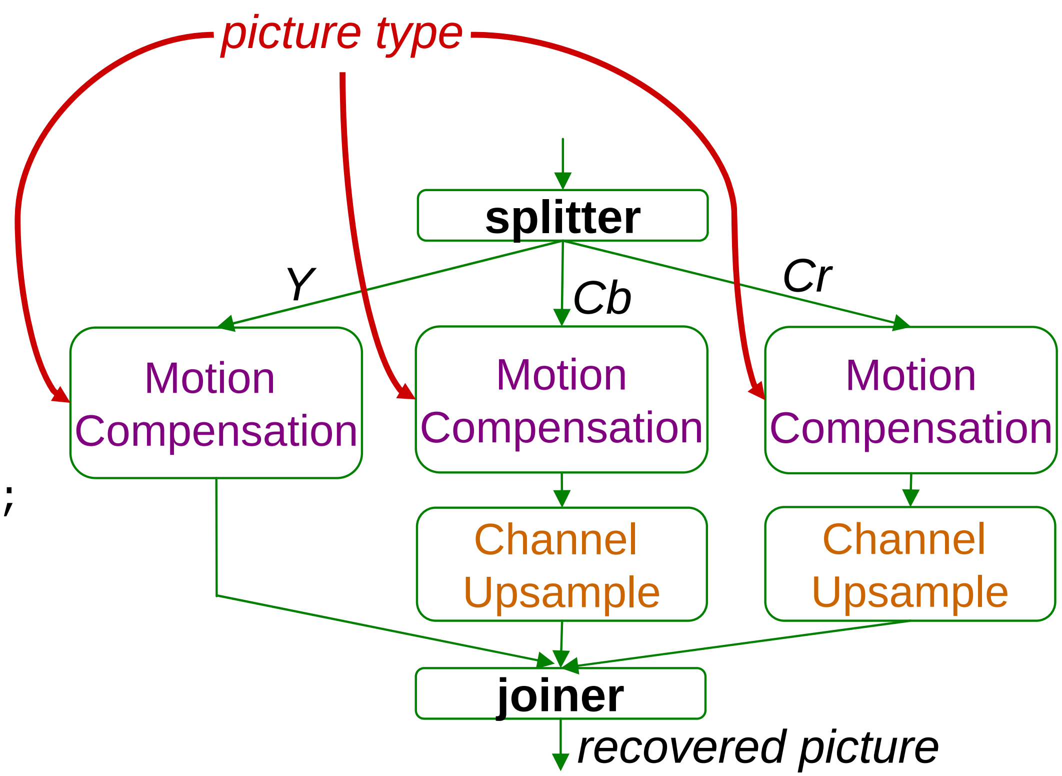
Teleport Messaging

- Avoids muddling data streams with control relevant information
- Localized interactions in large applications
  – A scalable alternative to global variables or excessive parameter passing



Motion Prediction and Messaging

```
portal<MotionCompensation> PT;

add splitjoin {
   split roundrobin(4*(B+V), B+V, B+V);

   add MotionCompensation() to PT;
   for (int i = 0; i < 2; i++) {
       add pipeline {
           add MotionCompensation() to PT;
           add ChannelUpsample(B);
       }
   }

   join roundrobin(1, 1, 1);
}
```

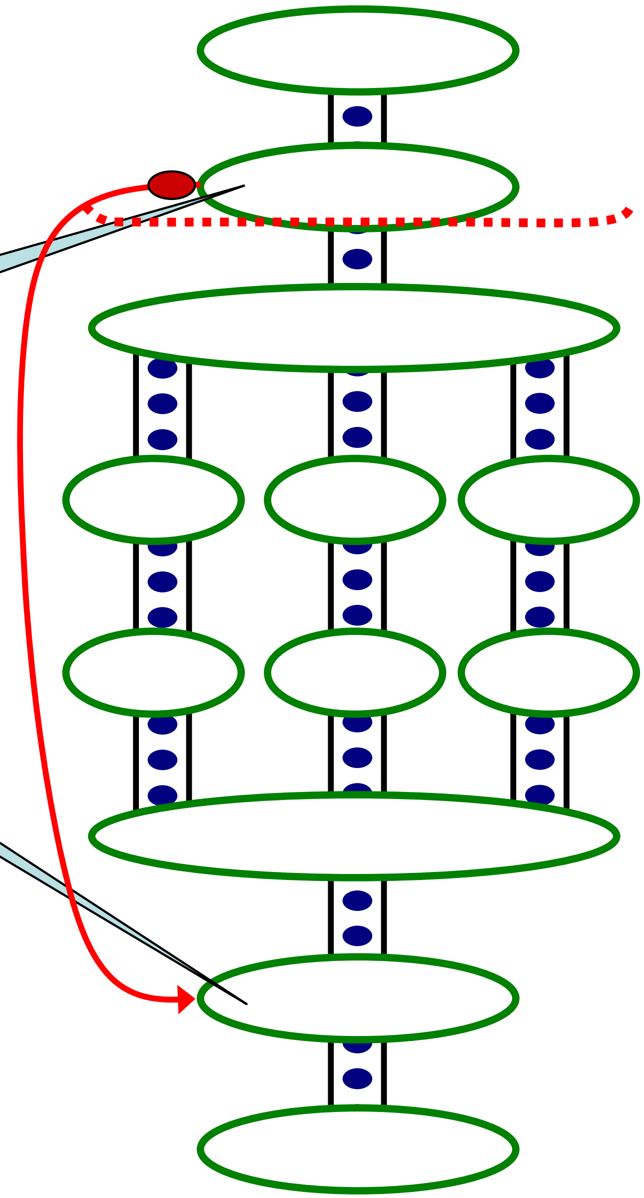


Teleport Messaging Overview

- Looks like method call, but timed relative to data in the stream

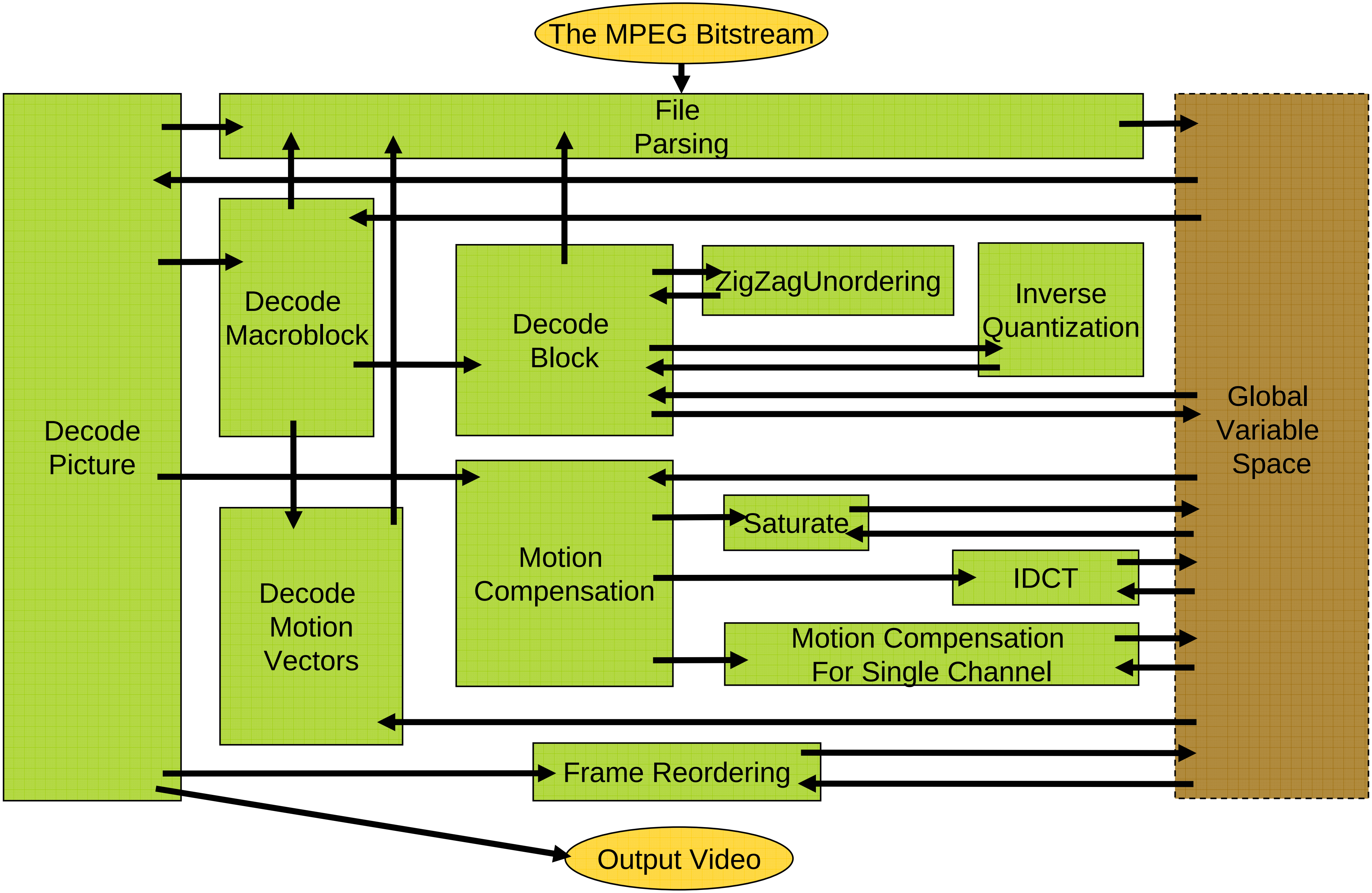
```
TargetFilter x;
if newPictureType(p) {
    x.setPictureType(p) @ 0;
}
```

```
void setPicturetype(int p) {
    reconfigure(p);
}
```



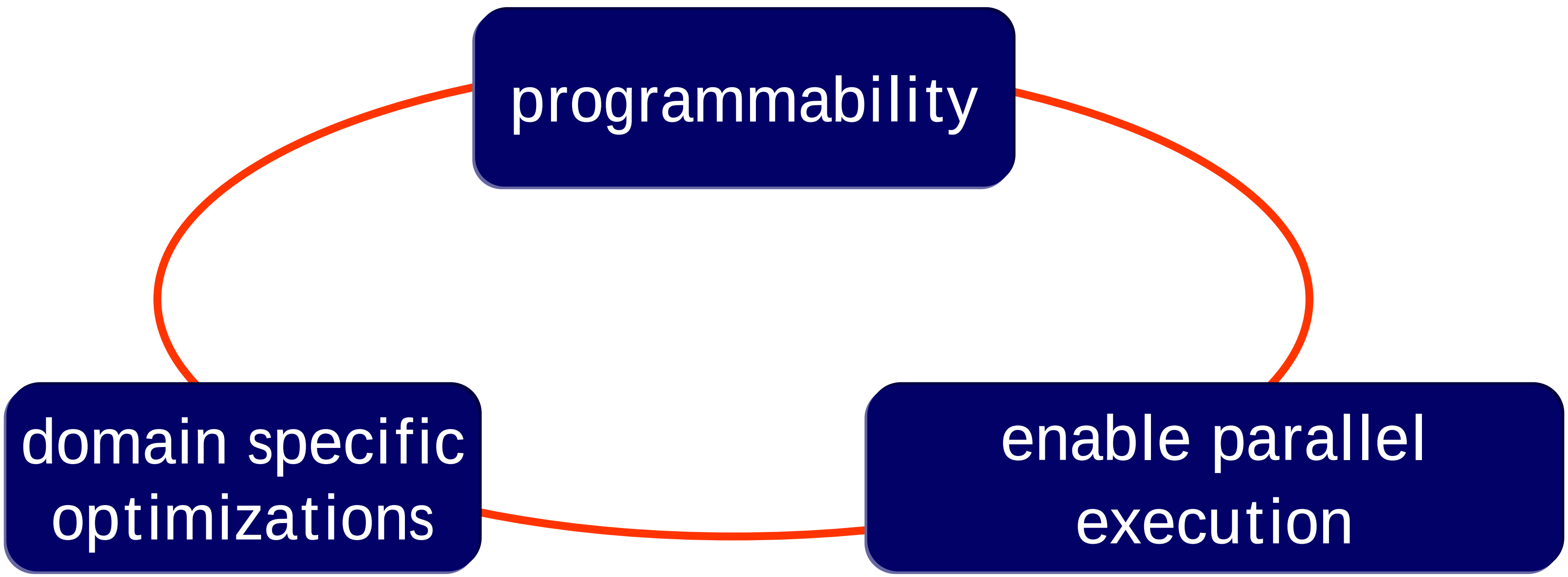
- Simple and precise for user
  - Exposes dependences to compiler
  - Adjustable latency
  - Can send upstream or downstream

Messaging Equivalent in C



Compiler-Aware Language Design

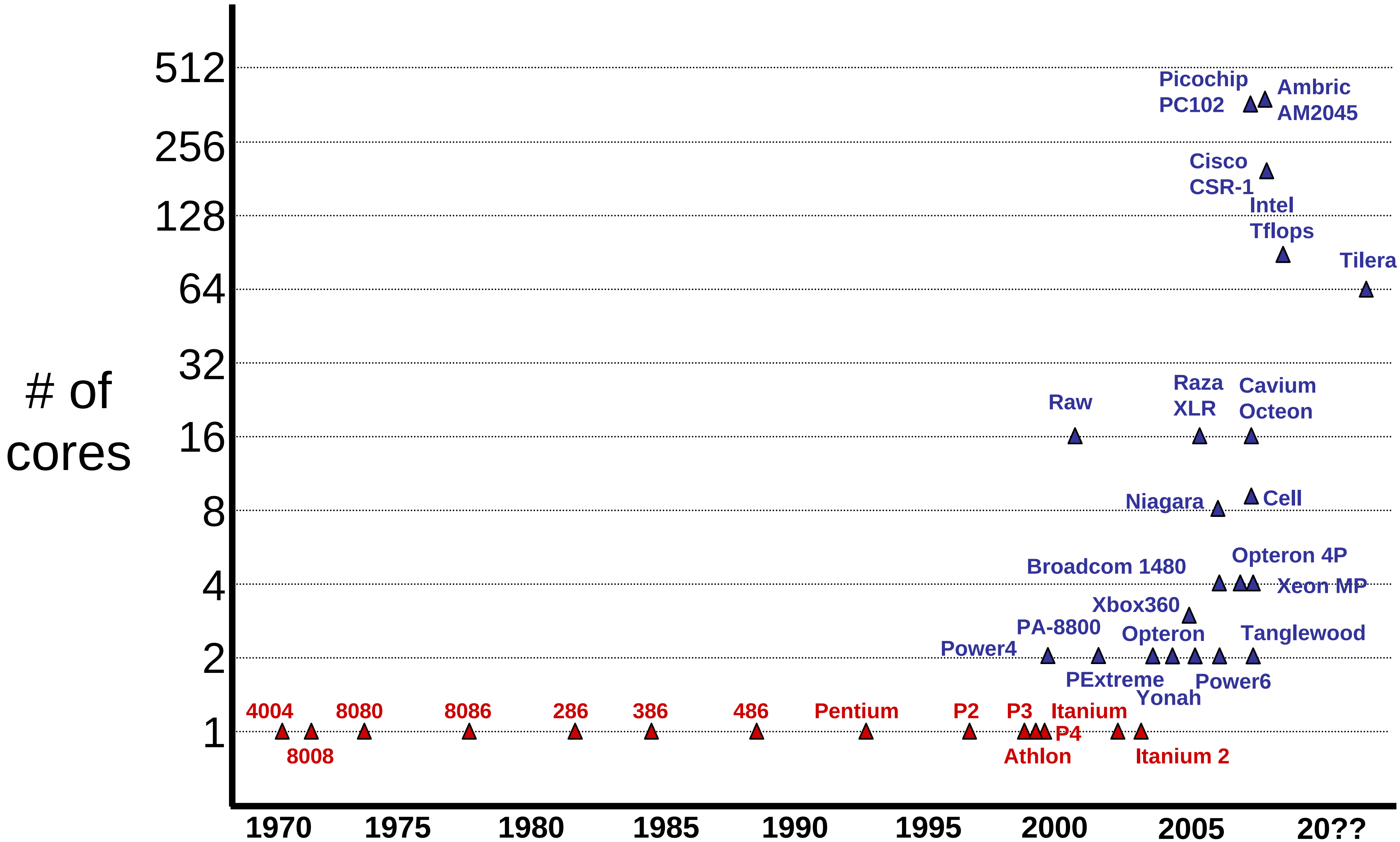
boost productivity, enable
faster development and
rapid prototyping



simple and effective
optimizations for domain
specific abstractions

target multicores, clusters,
tiled architectures, DSPs,
graphics processors, …

Multicores Are Here!



Von Neumann Languages

- Why C (FORTRAN, C++ etc.) became very successful?
  - Abstracted out the differences of von Neumann machines
  - Directly expose the common properties
  - Can have a very efficient mapping to a von Neumann machine
  - "C is the portable machine language for von Numann machines"

- von Neumann languages are a curse for Multicores
  - We have squeezed out all the performance out of C
  - But, cannot easily map C into multicores

Common Machine Languages

Unicores:

Common Properties
Single flow of control
Single memory image

Differences:	
Register File	Register Allocation
ISA	Instruction Selection
Functional Units	Instruction Scheduling

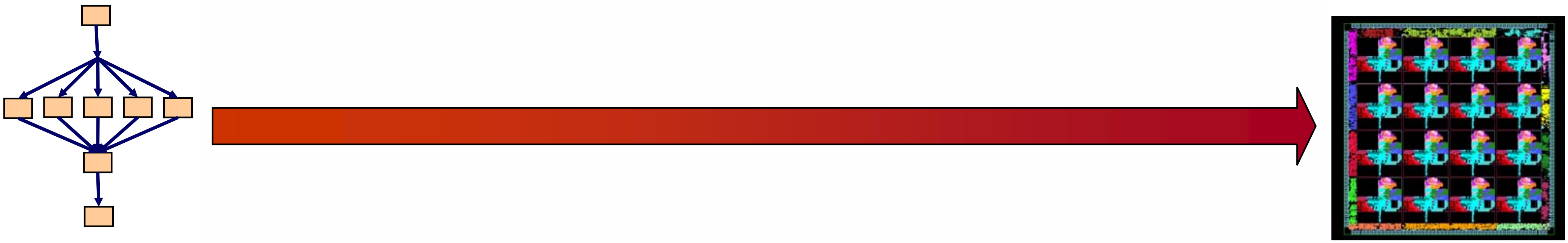
von-Neumann languages represent the common properties and abstract away the differences

Multicores:

Common Properties
Multiple flows of control
Multiple local memories

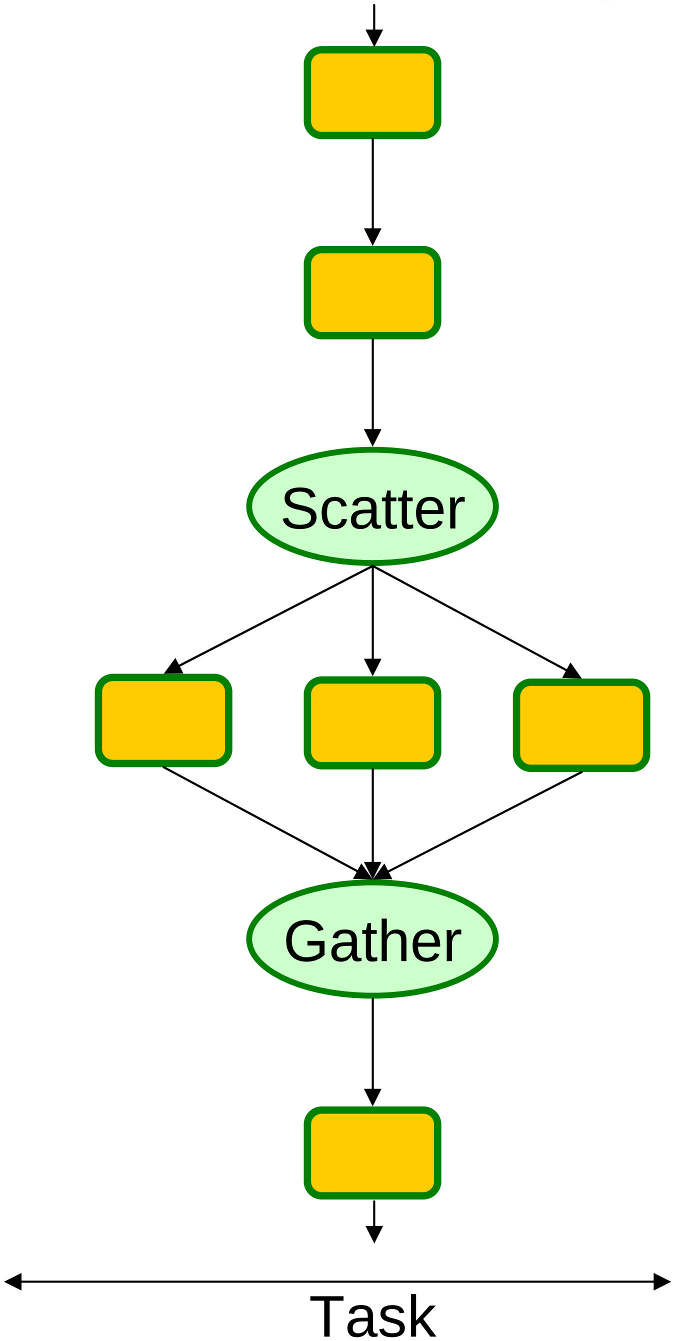
Differences:
Number and capabilities of cores
Communication Model
Synchronization Model

Bridging the Abstraction layers



- StreamIt exposes the data movement
  - Graph structure is architecture independent
- StreamIt exposes the parallelism
  - Explicit task parallelism
  - Implicit but inherent data and pipeline parallelism
- Each multicore is different in granularity and topology
  - Communication is exposed to the compiler
- The compiler needs to efficiently bridge the abstraction
  - Map the computation and communication pattern of the program to the cores, memory and the communication substrate

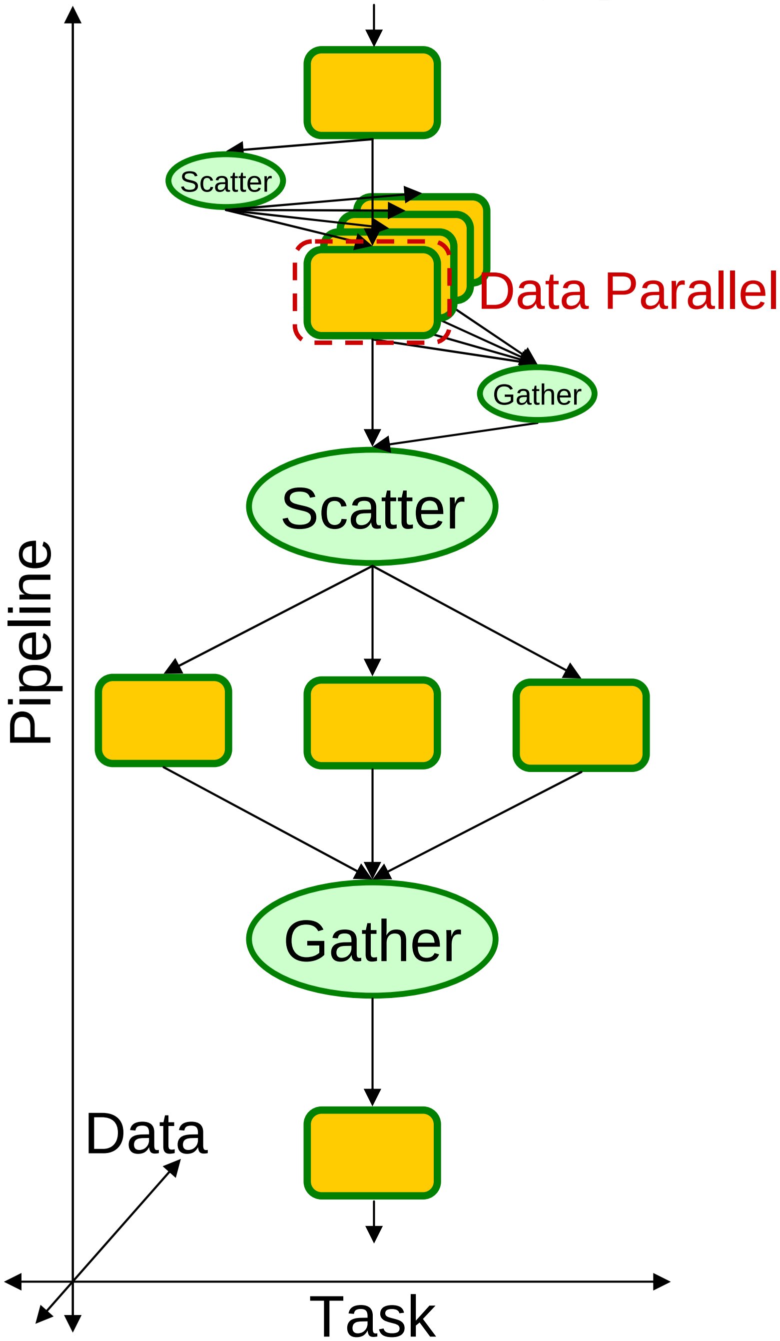
Types of Parallelism



Task Parallelism

- Parallelism explicit in algorithm
- Between filters *without* producer/consumer relationship

Types of Parallelism



Task Parallelism

- Parallelism explicit in algorithm
- Between filters *without* producer/consumer relationship

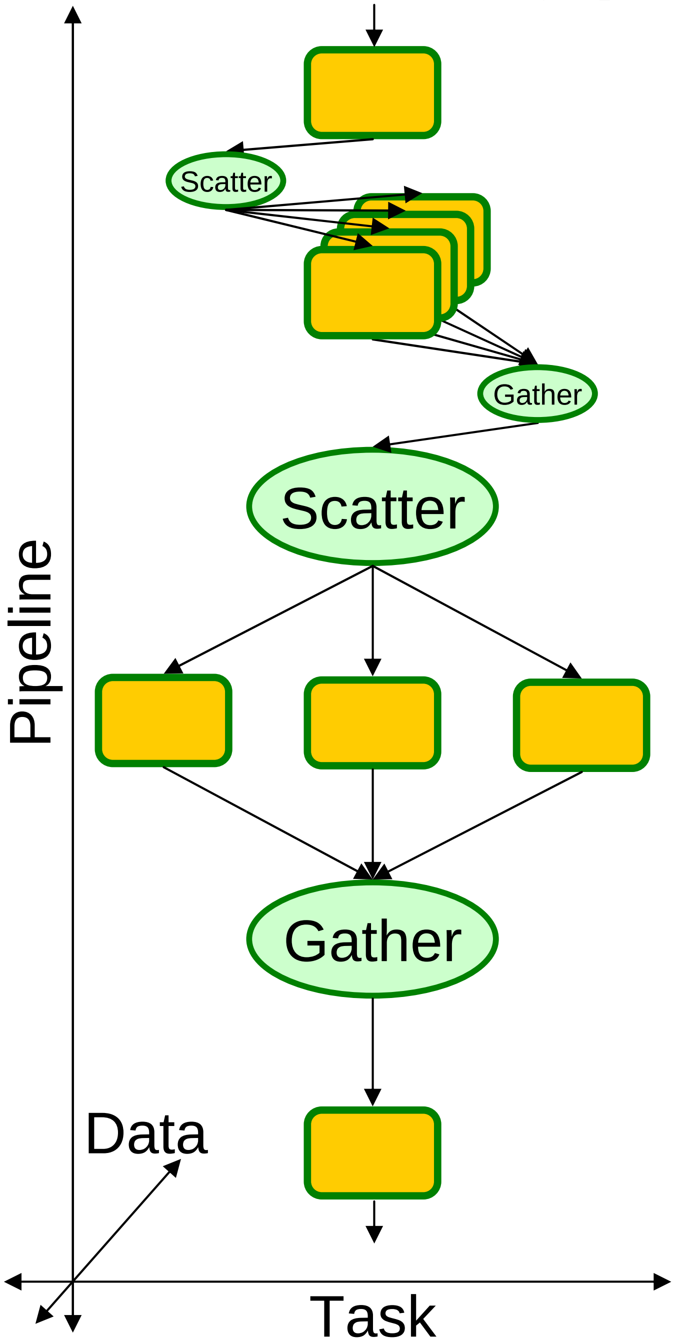
Data Parallelism

- Between iterations of a *stateless* filter
- Place within scatter/gather pair (*fission*)
- Can't parallelize filters with state

Pipeline Parallelism

- Between producers and consumers
- *Stateful* filters can be parallelized

Types of Parallelism



Traditionally:

Task Parallelism

– Thread (fork/join) parallelism

Data Parallelism

– Data parallel loop (**forall**)

Pipeline Parallelism

– Usually exploited in hardware

Problem Statement

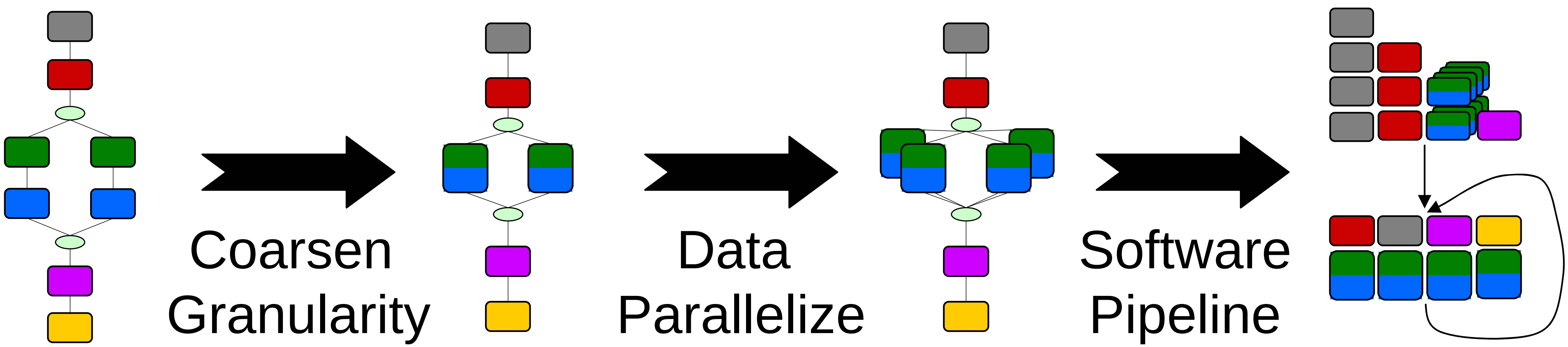
Given:

- Stream graph with compute and communication estimate for each filter
- Computation and communication resources of the target machine

Find:

- Schedule of execution for the filters that best utilizes the available parallelism to fit the machine resources

Our 3-Phase Solution

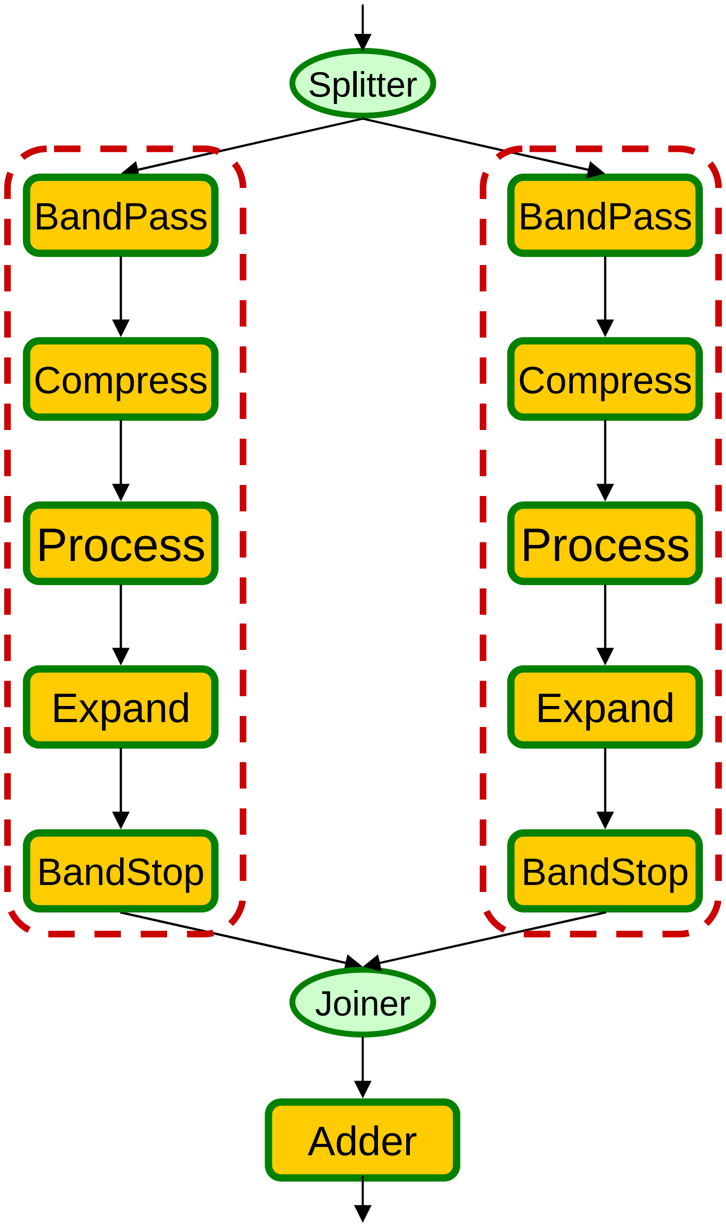


1. Coarsen: Fuse stateless sections of the graph
2. Data Parallelize: parallelize stateless filters
3. Software Pipeline: parallelize stateful filters

Compile to a 16 core architecture

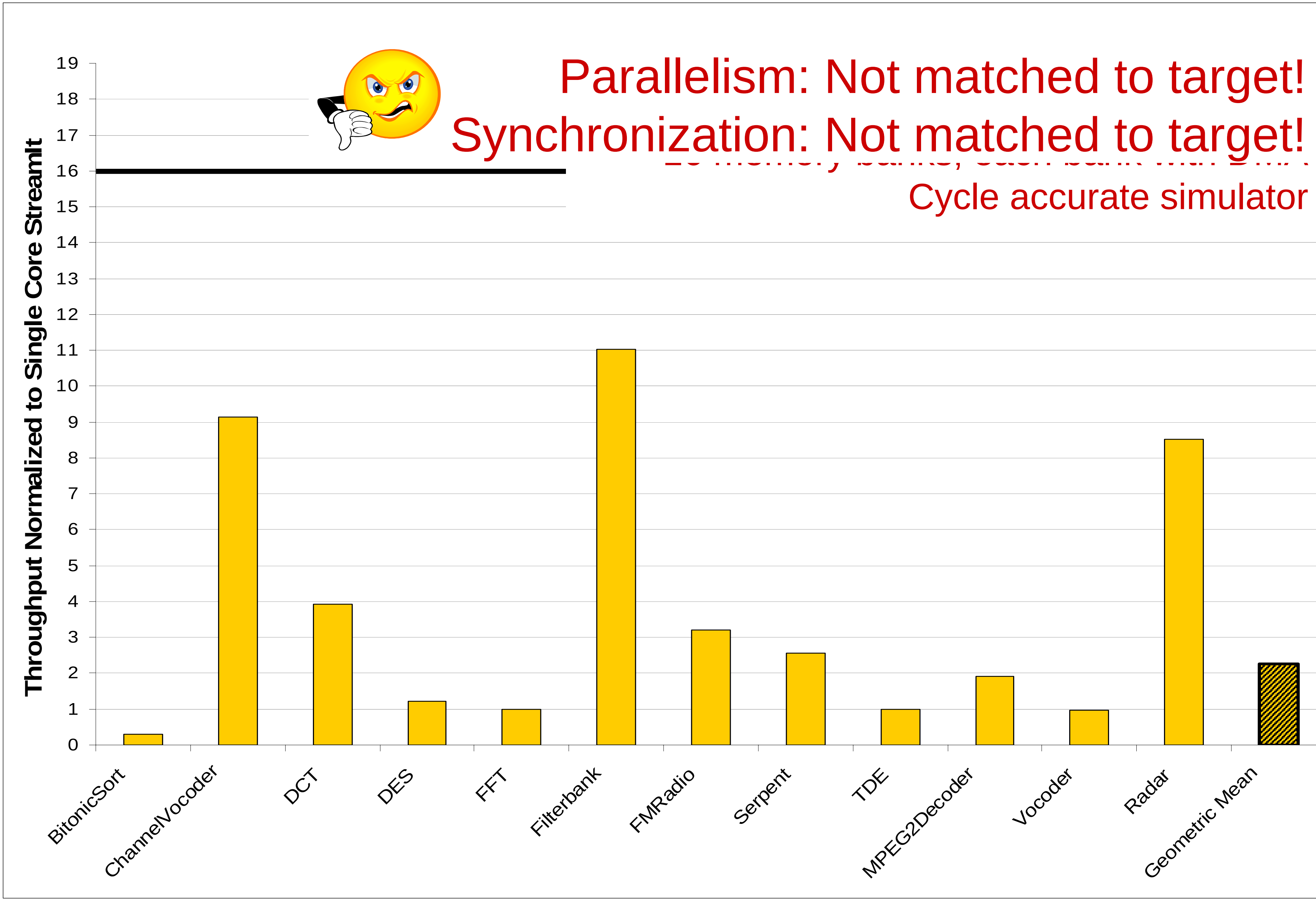
– 11.2x mean throughput speedup over single core

Baseline 1: Task Parallelism

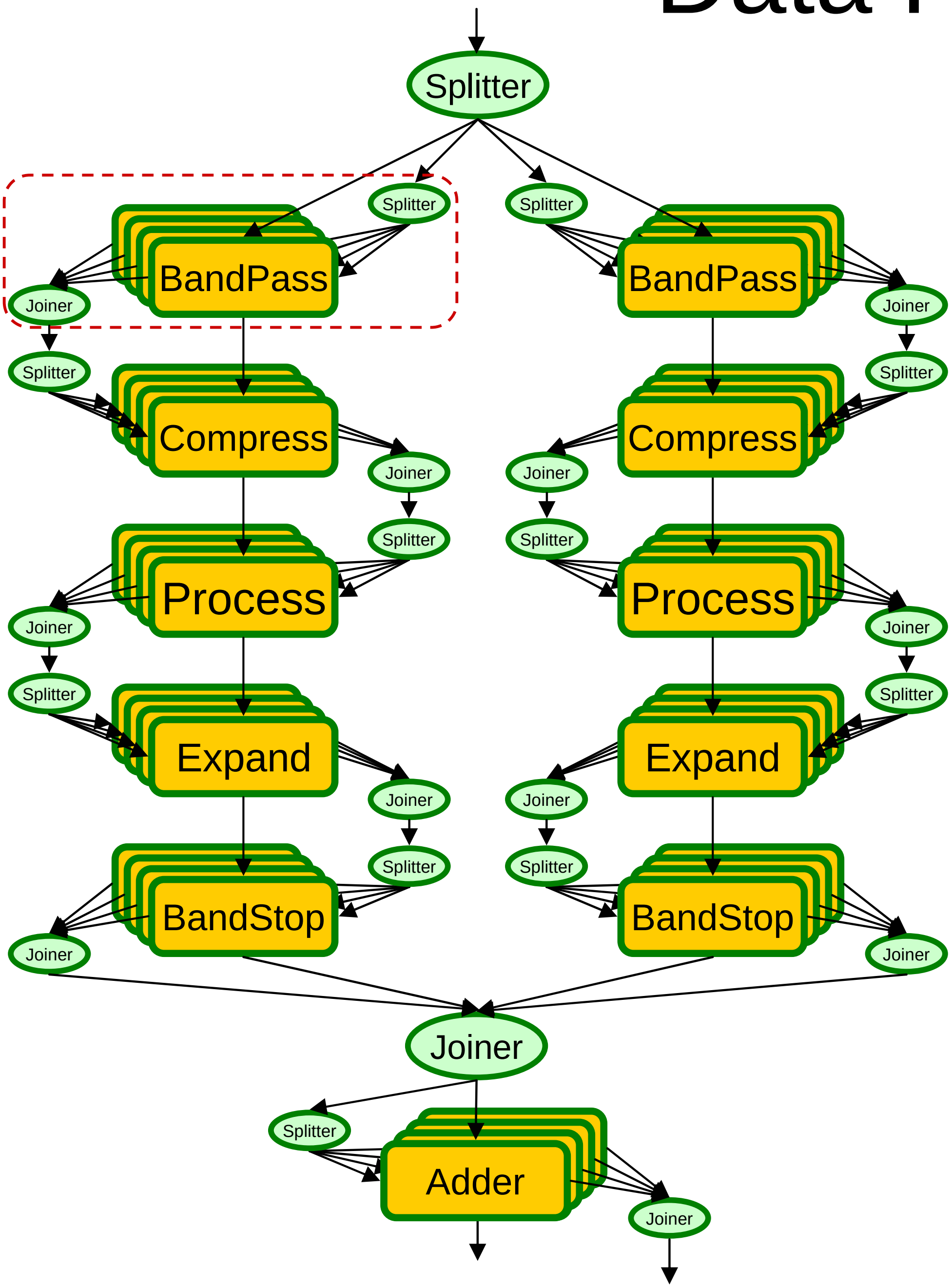


- Inherent task parallelism between two processing pipelines
- Task Parallel Model:
  - Only parallelize explicit task parallelism
  - Fork/join parallelism
- Execute this on a 2 core machine ~2x speedup over single core
- What about 4, 16, 1024, … cores?

Evaluation: Task Parallelism

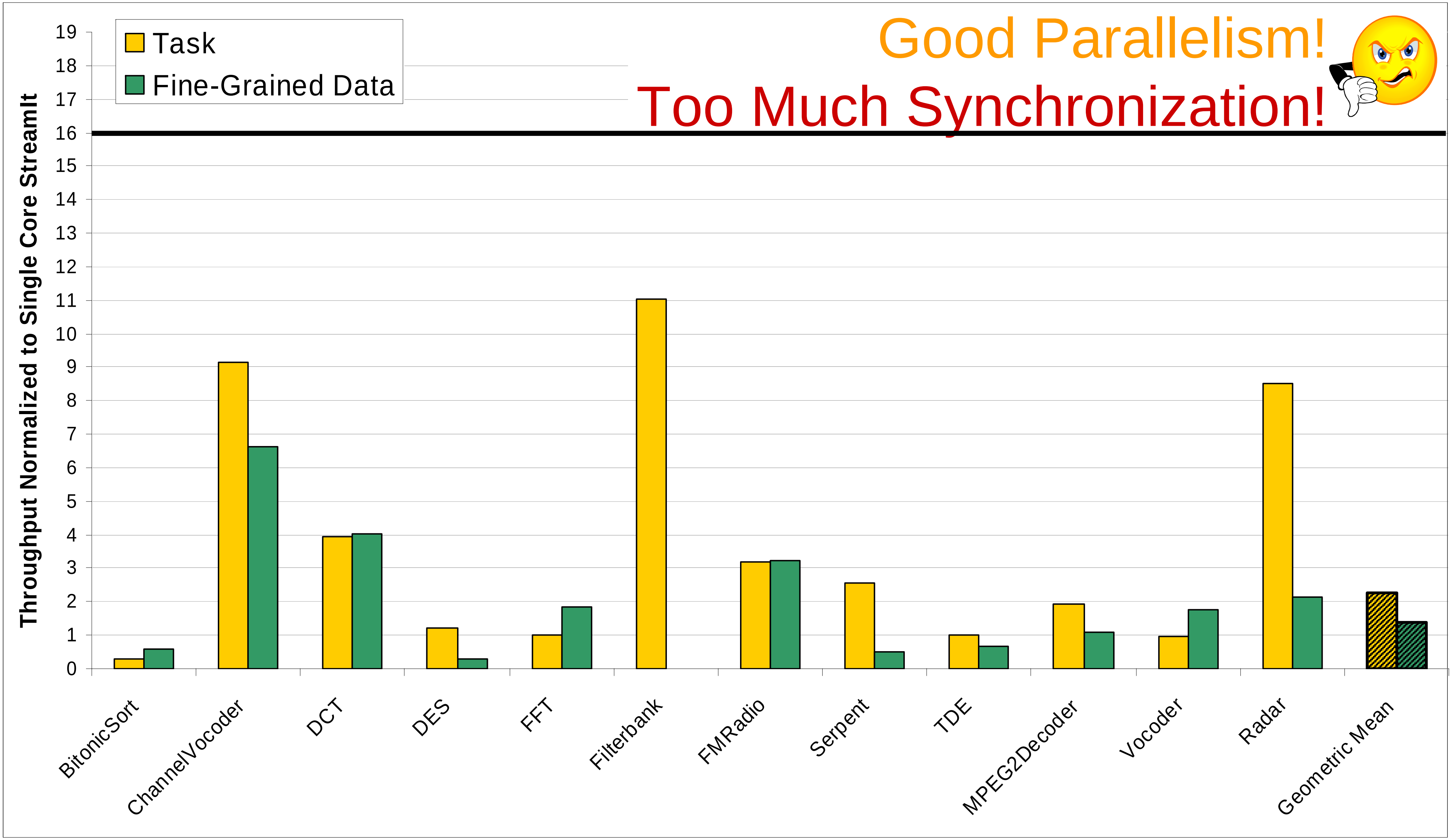


Baseline 2: Fine-Grained Data Parallelism

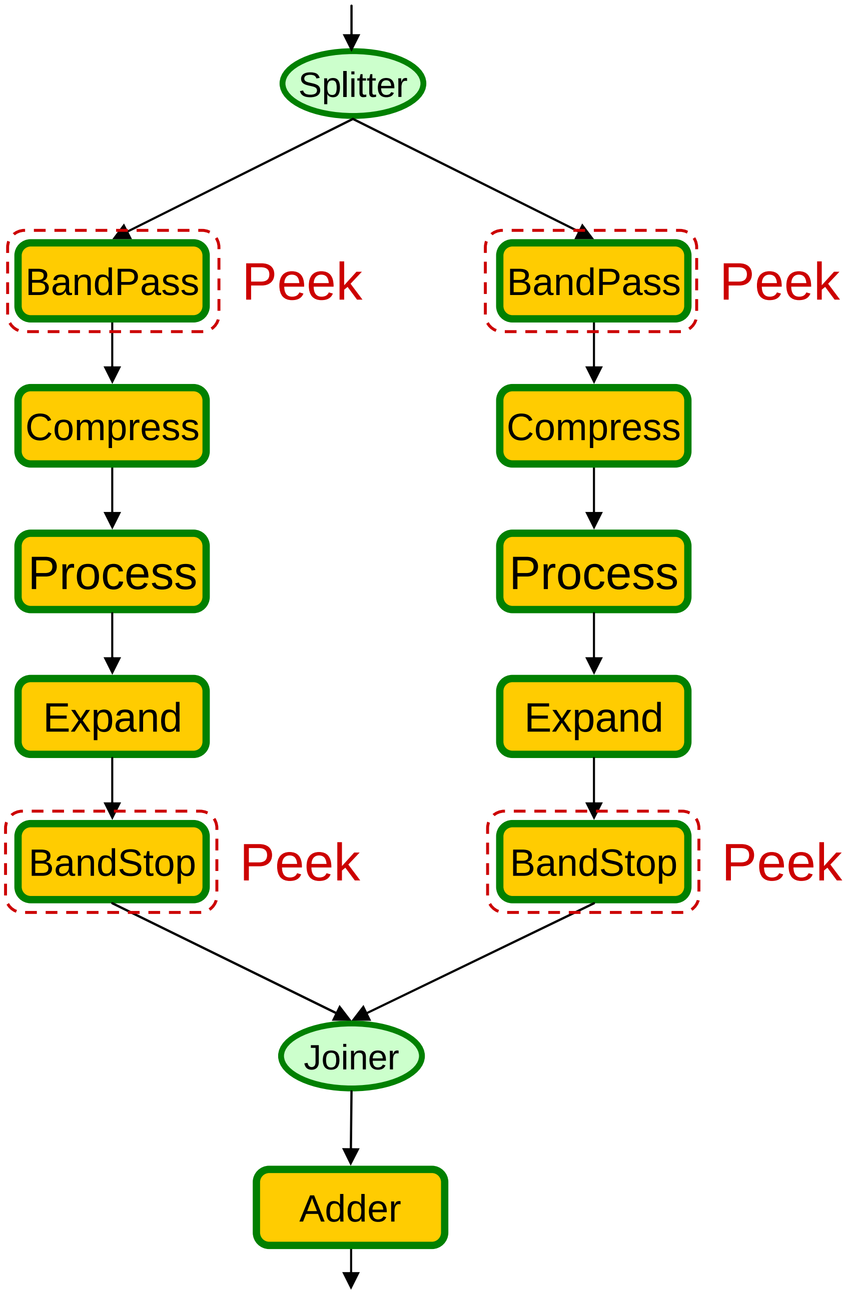


- Each of the filters in the example are stateless
- Fine-grained Data Parallel Model:
  - *Fiss* each stateless filter *N* ways (*N* is number of cores)
  - Remove scatter/gather if possible
- We can introduce data parallelism
  - Example: 4 cores
- Each fission group occupies entire machine

Evaluation: Fine-Grained Data Parallelism

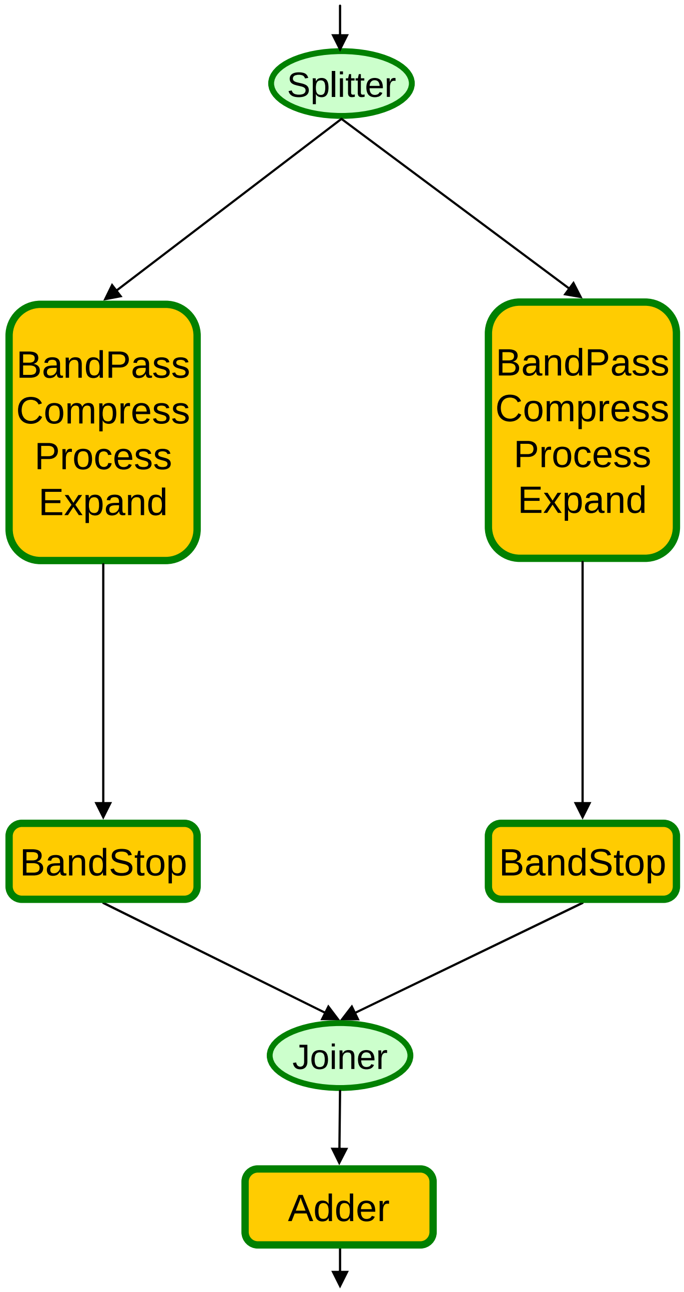


Phase 1: Coarsen the Stream Graph



- Before data-parallelism is exploited
- *Fuse* stateless pipelines as much as possible without introducing state
  - Don’t fuse stateless with stateful
  - Don’t fuse a peeking filter with anything upstream

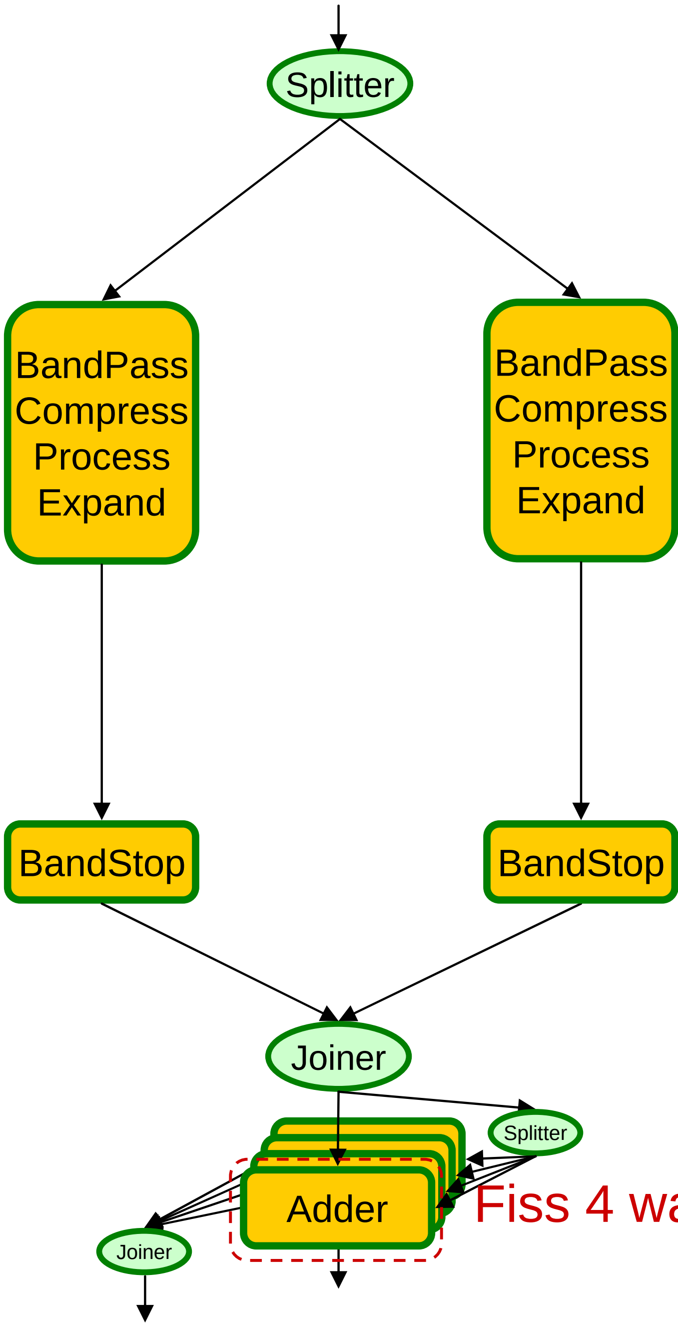
Phase 1: Coarsen the Stream Graph



- Before data-parallelism is exploited
- *Fuse* stateless pipelines as much as possible without introducing state
  - Don't fuse stateless with stateful
  - Don't fuse a peeking filter with anything upstream
- Benefits:
  - Reduces global communication and synchronization
  - Exposes inter-node optimization opportunities

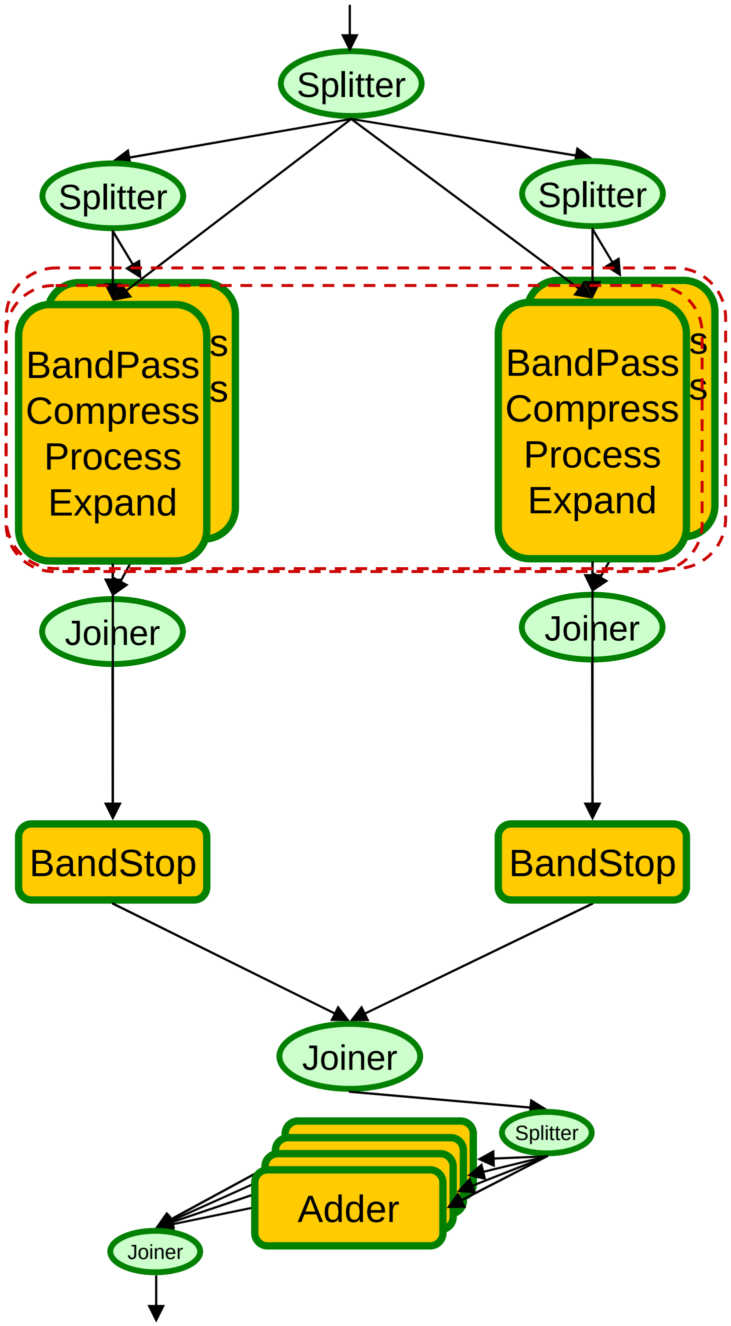
Phase 2: Data Parallelize

Data Parallelize for 4 cores



Fiss 4 ways, to occupy entire chip

Phase 2: Data Parallelize



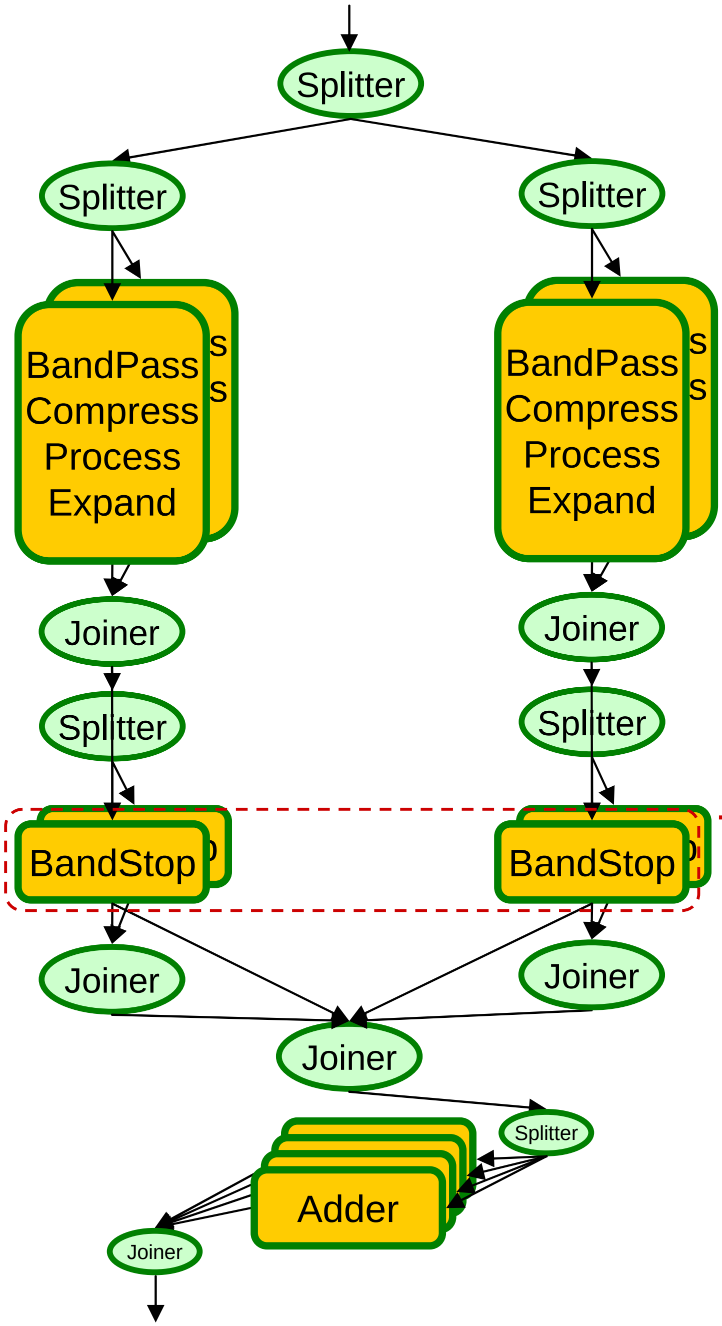
Data Parallelize for 4 cores

Task parallelism!

Each fused filter does equal work

Fiss each filter 2 times to occupy entire chip

Phase 2: Data Parallelize

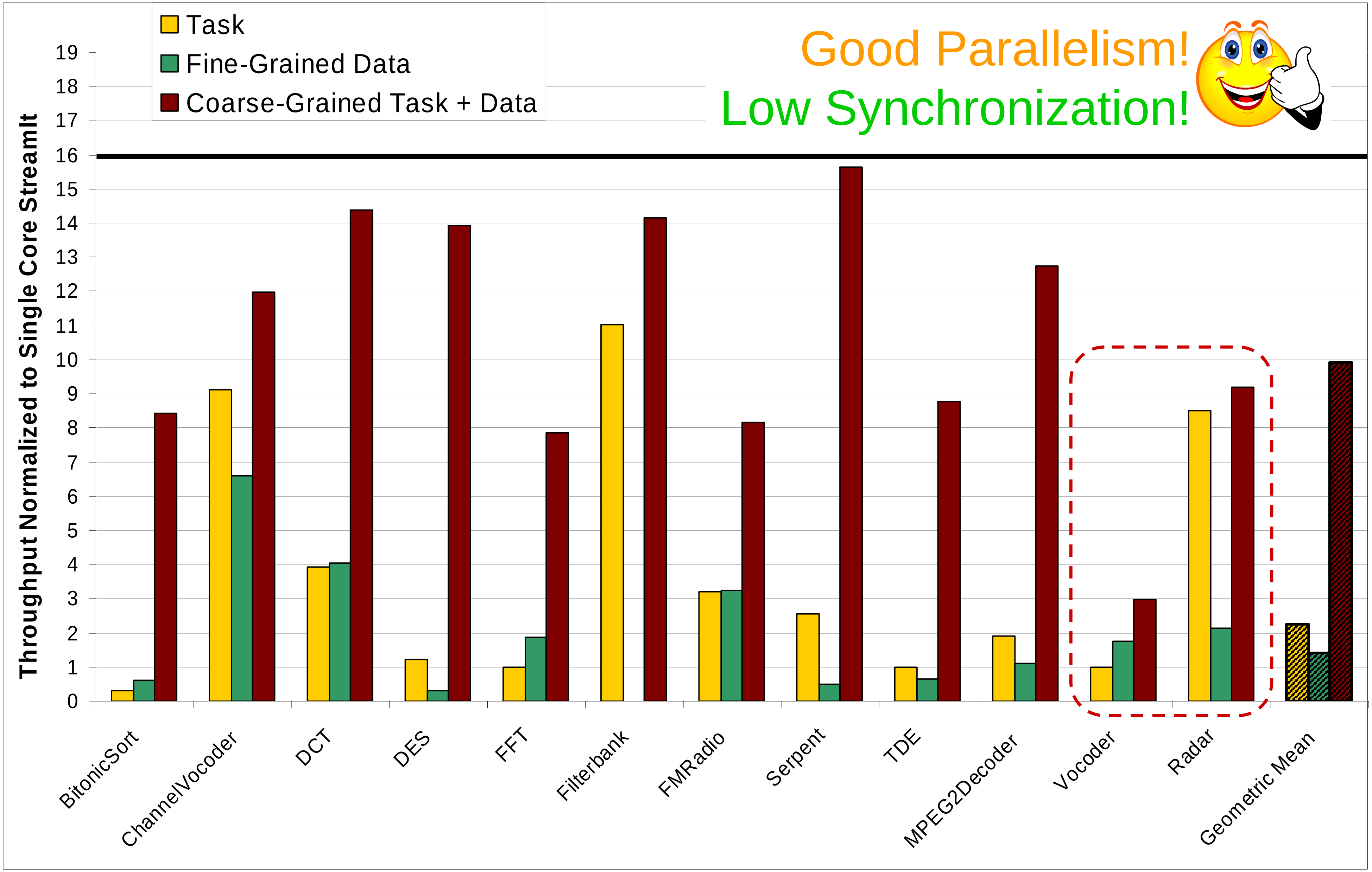


Task parallelism, each filter does equal work
Fiss each filter 2 times to occupy entire chip

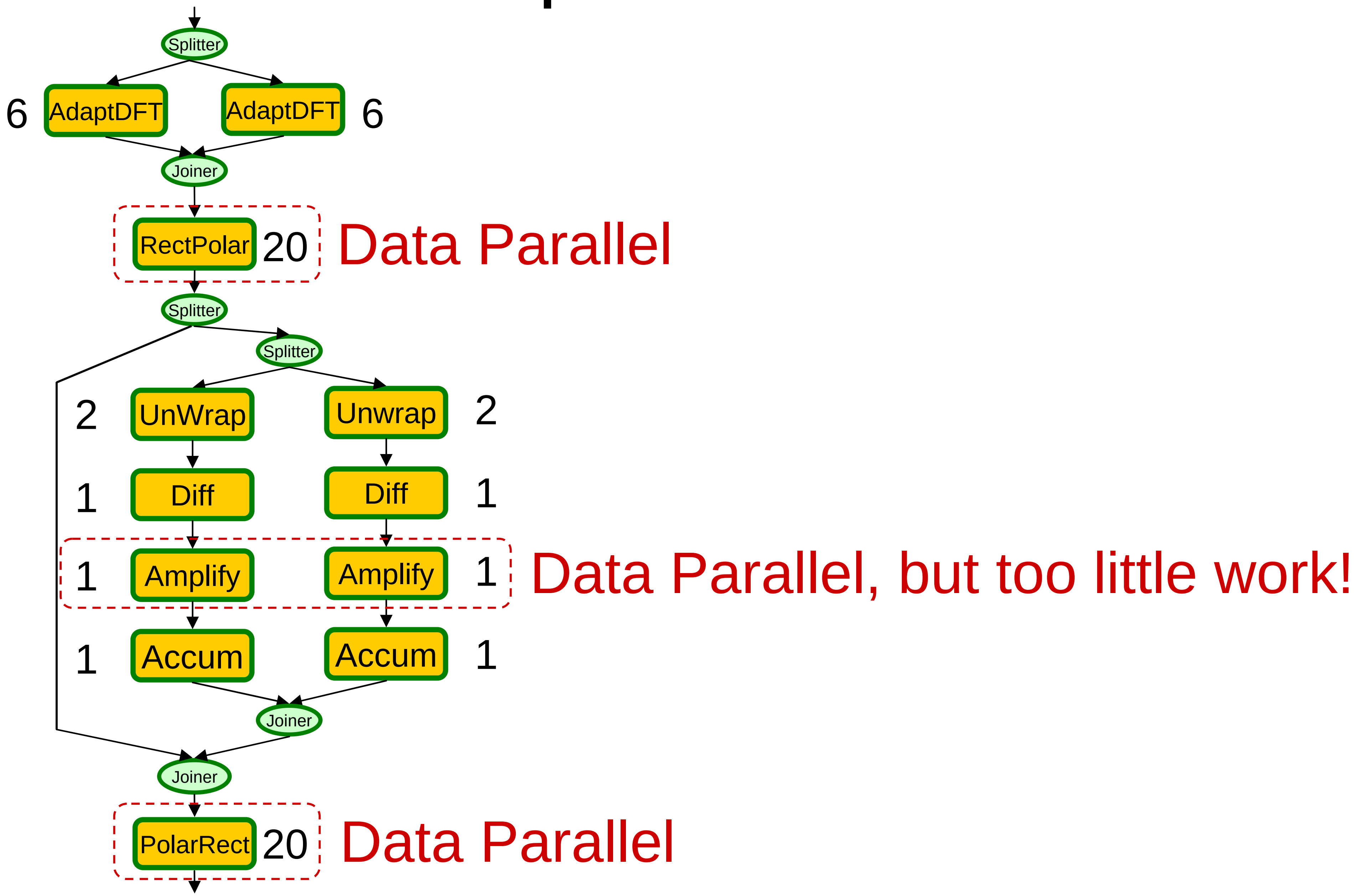
Data Parallelize for 4 cores

- Task-conscious data parallelization
  - Preserve task parallelism
- Benefits:
  - Reduces global communication and synchronization

Evaluation: Coarse-Grained Data Parallelism

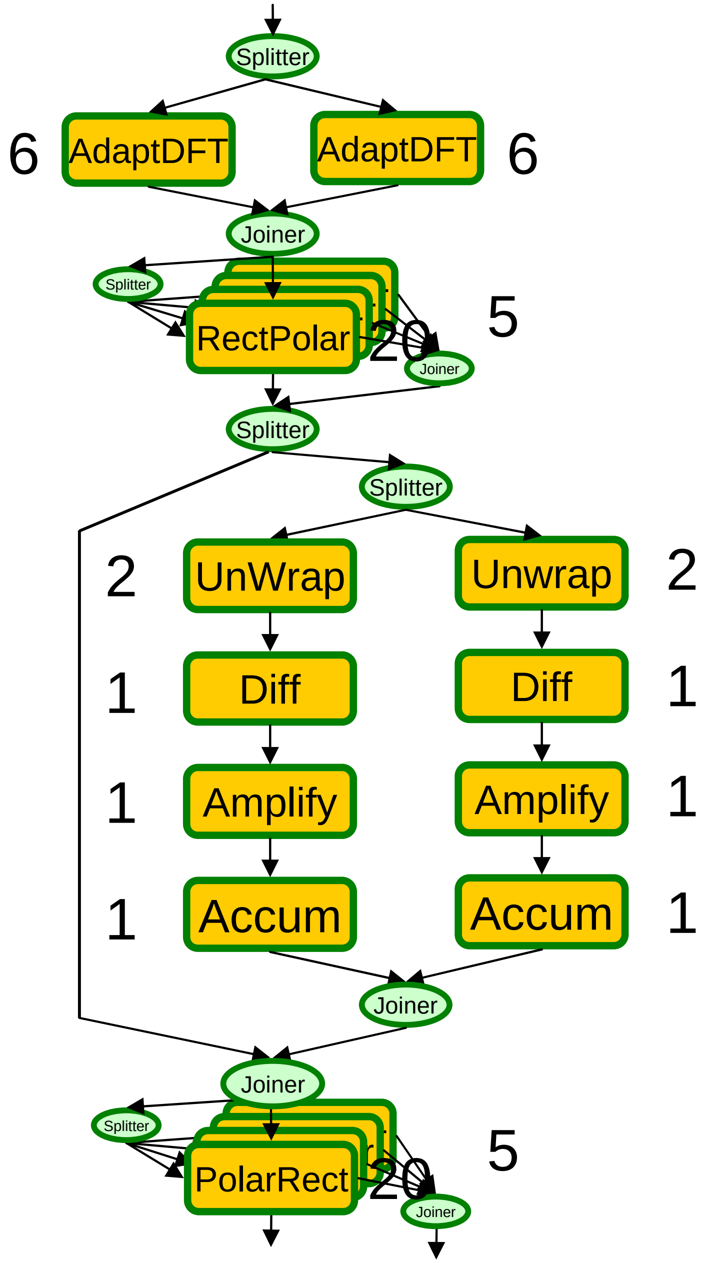


Simplified Vocoder



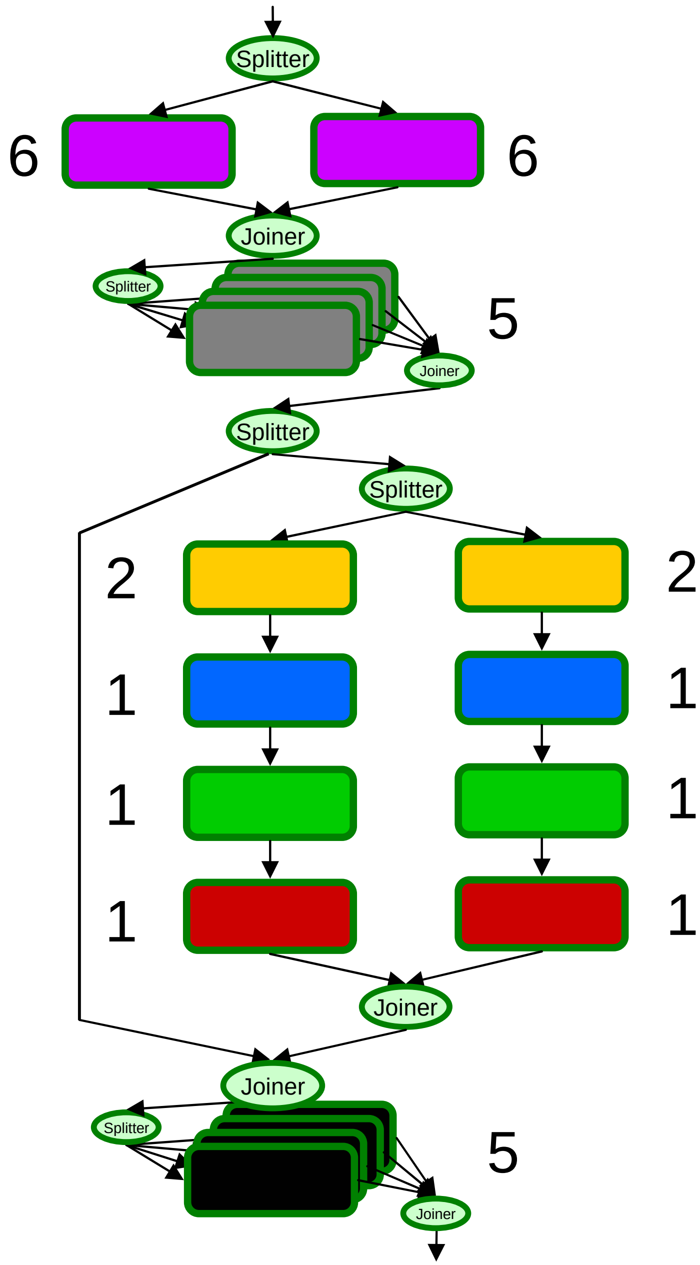
Target a 4 core machine

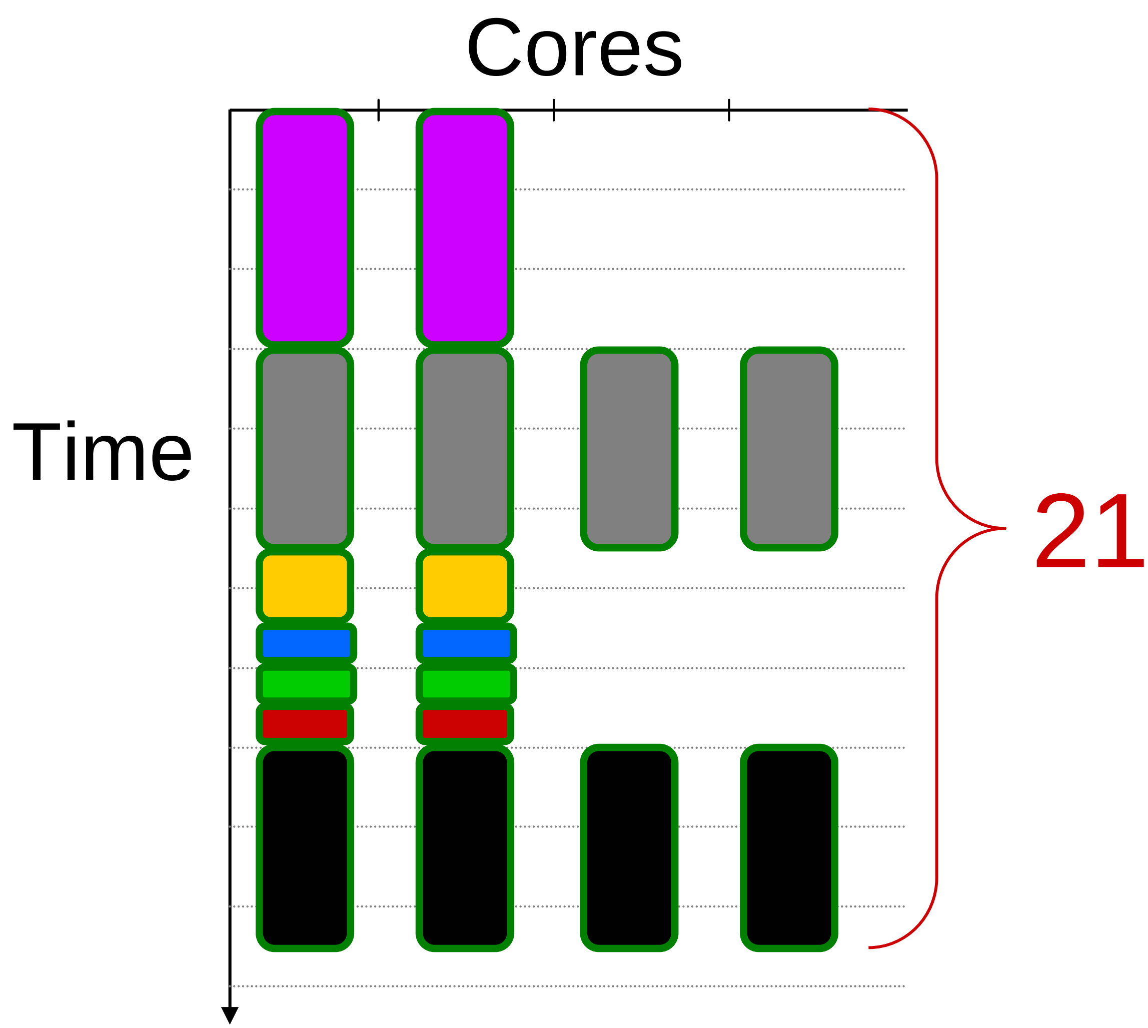
Data Parallelize



Target a 4 core machine

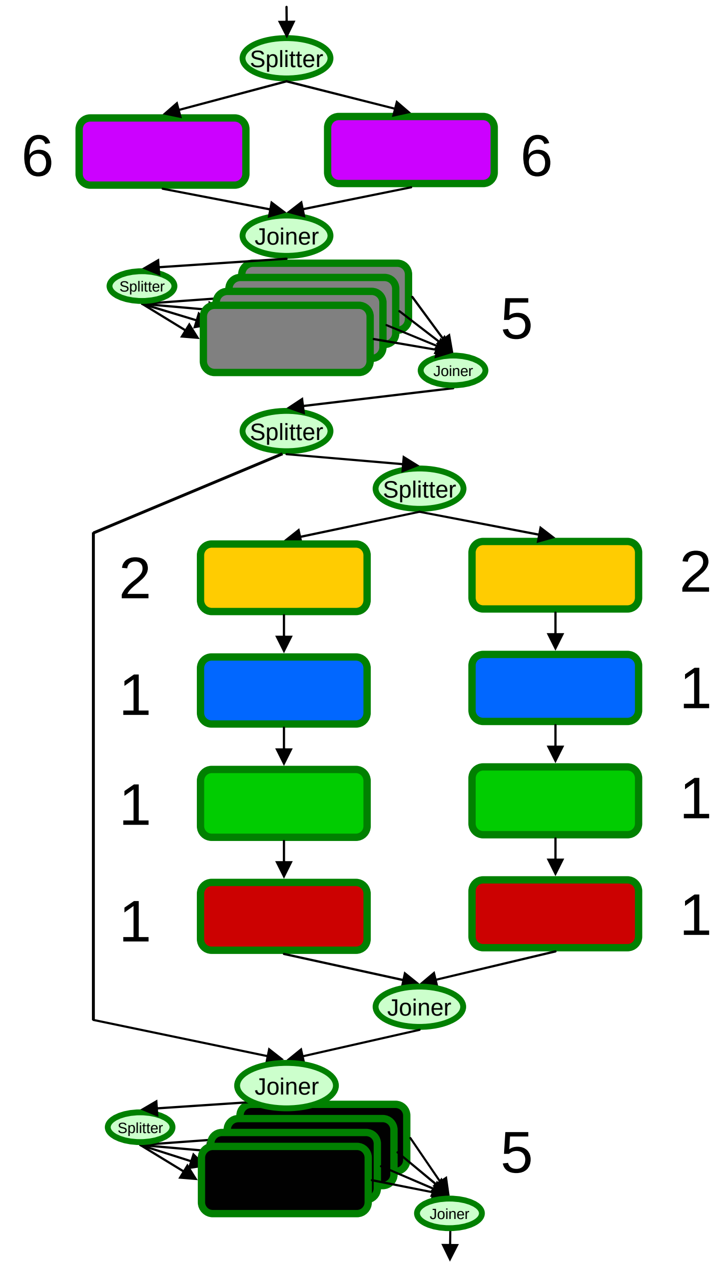
Data + Task Parallel Execution

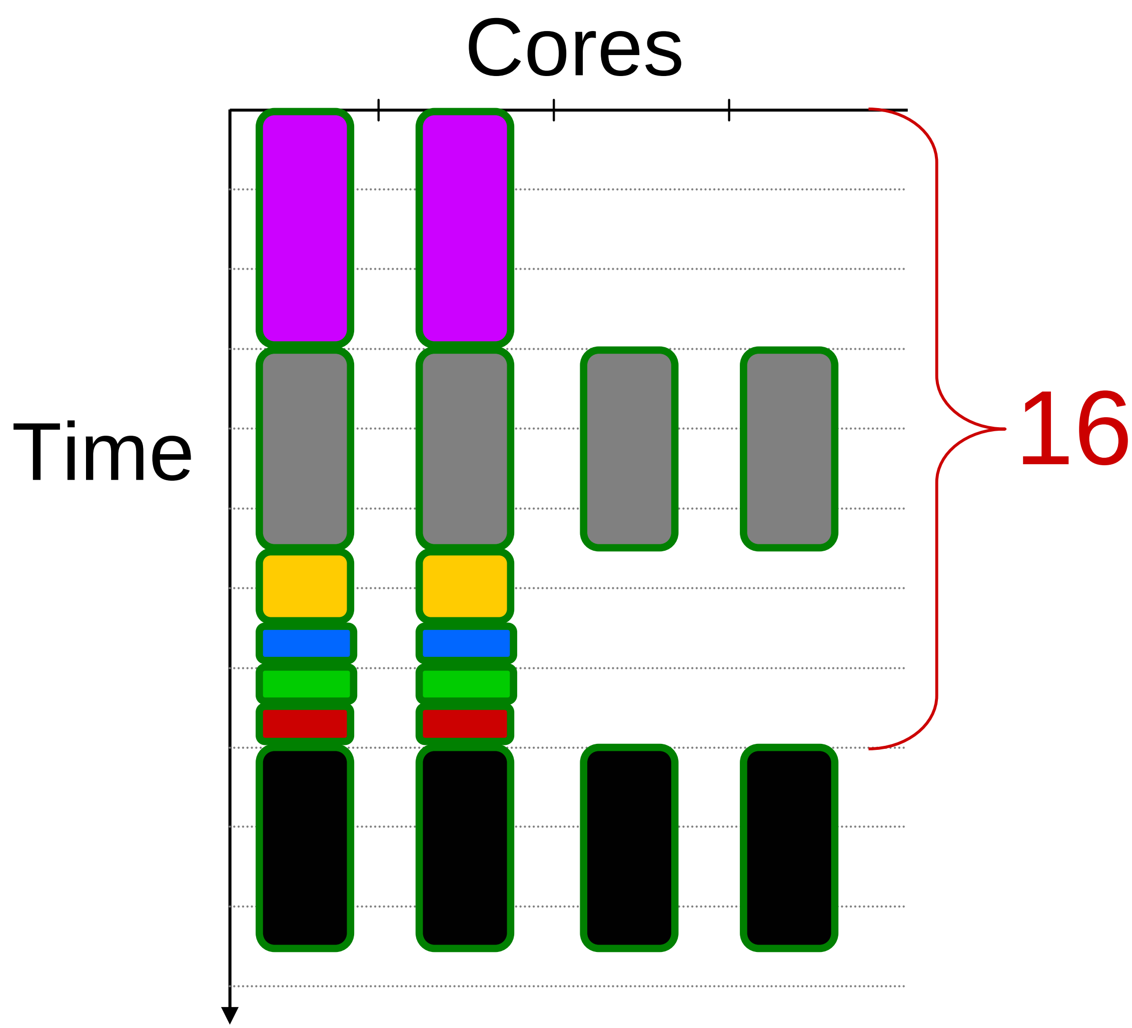




Target 4 core machine

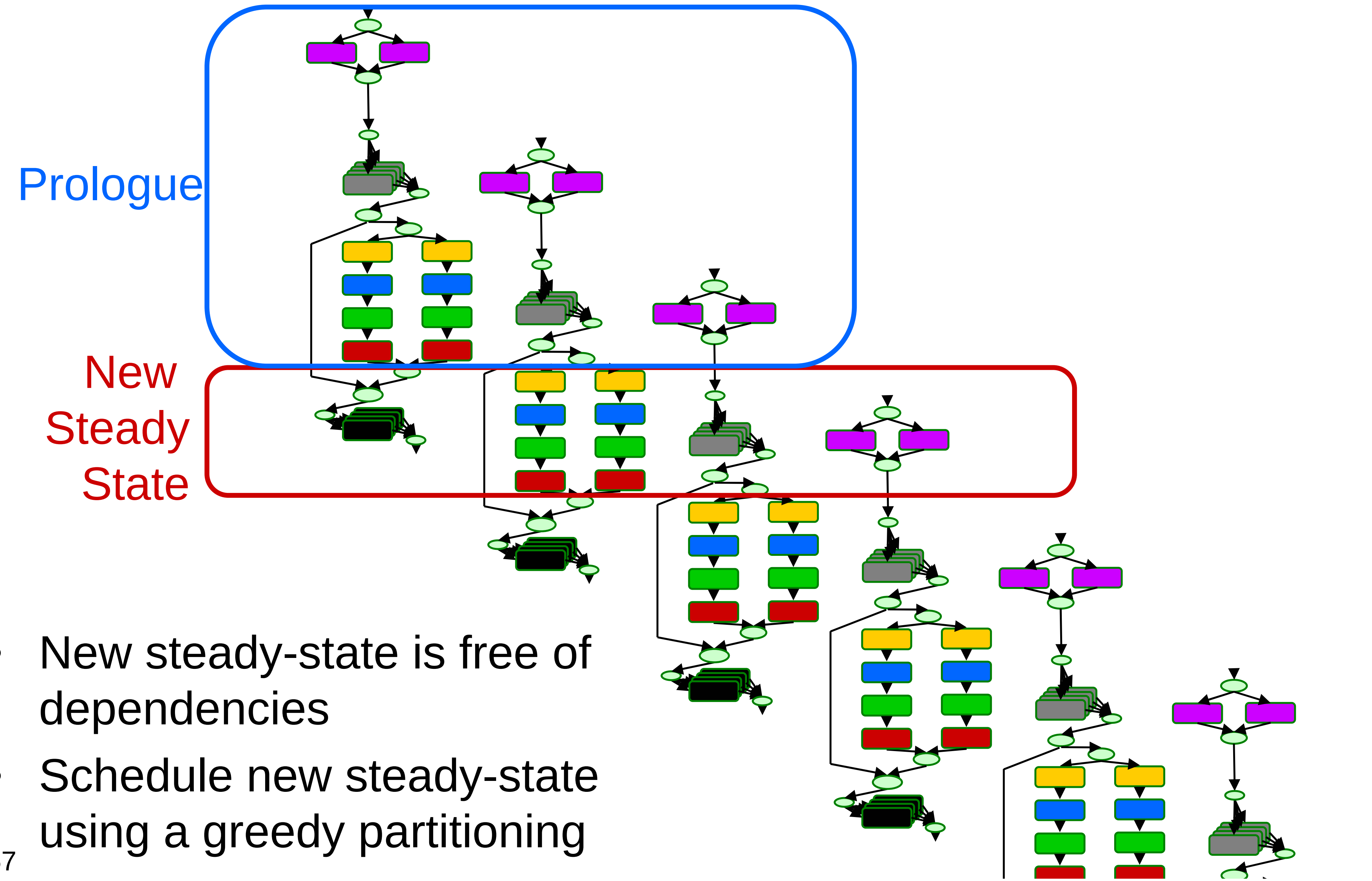
We Can Do Better!





Target 4 core machine

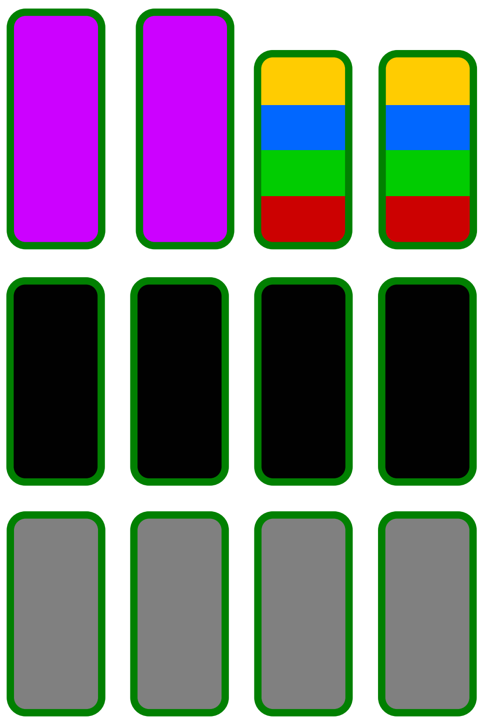
Phase 3: Coarse-Grained Software Pipelining

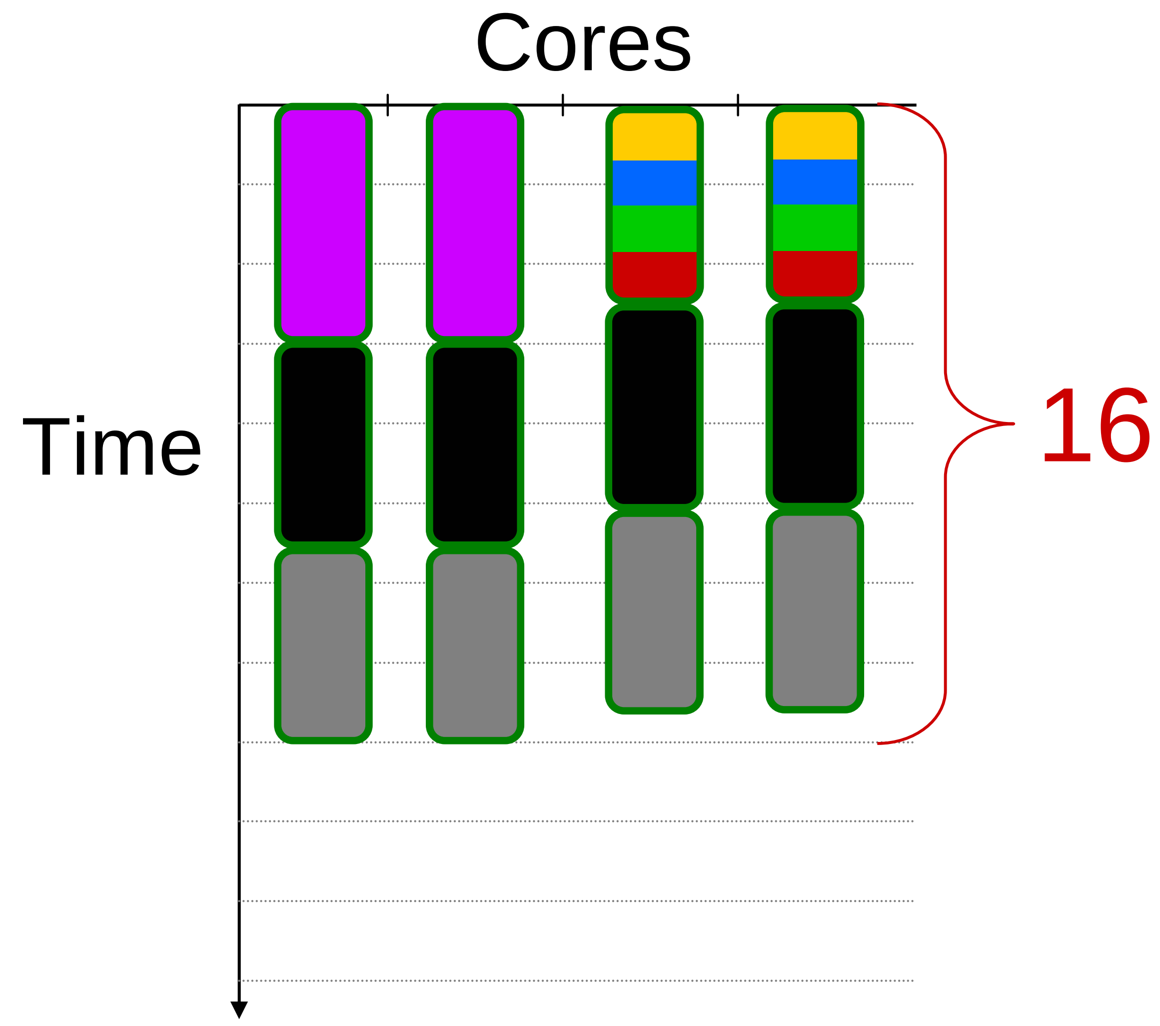


- New steady-state is free of dependencies
- Schedule new steady-state using a greedy partitioning

Greedy Partitioning

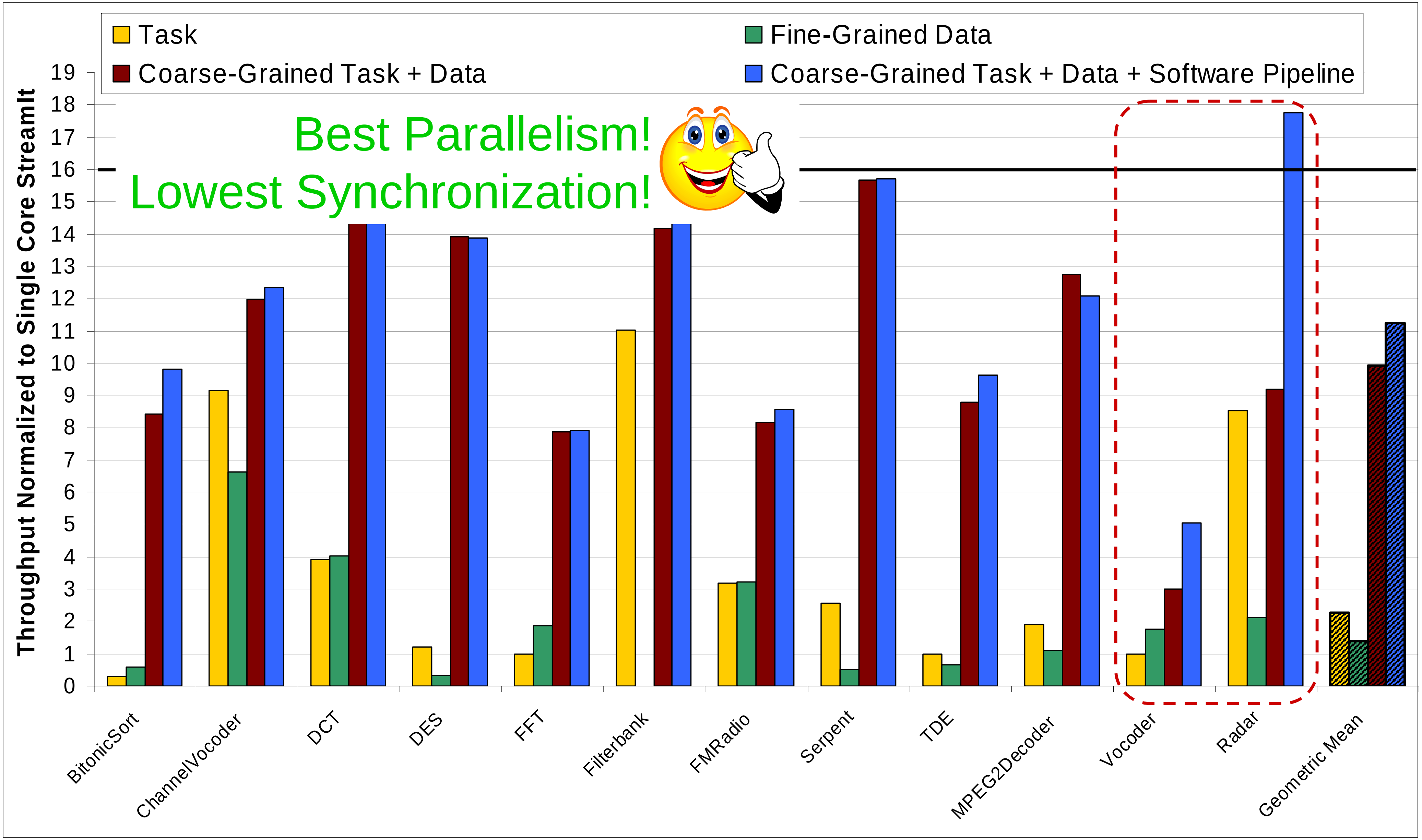
To Schedule:





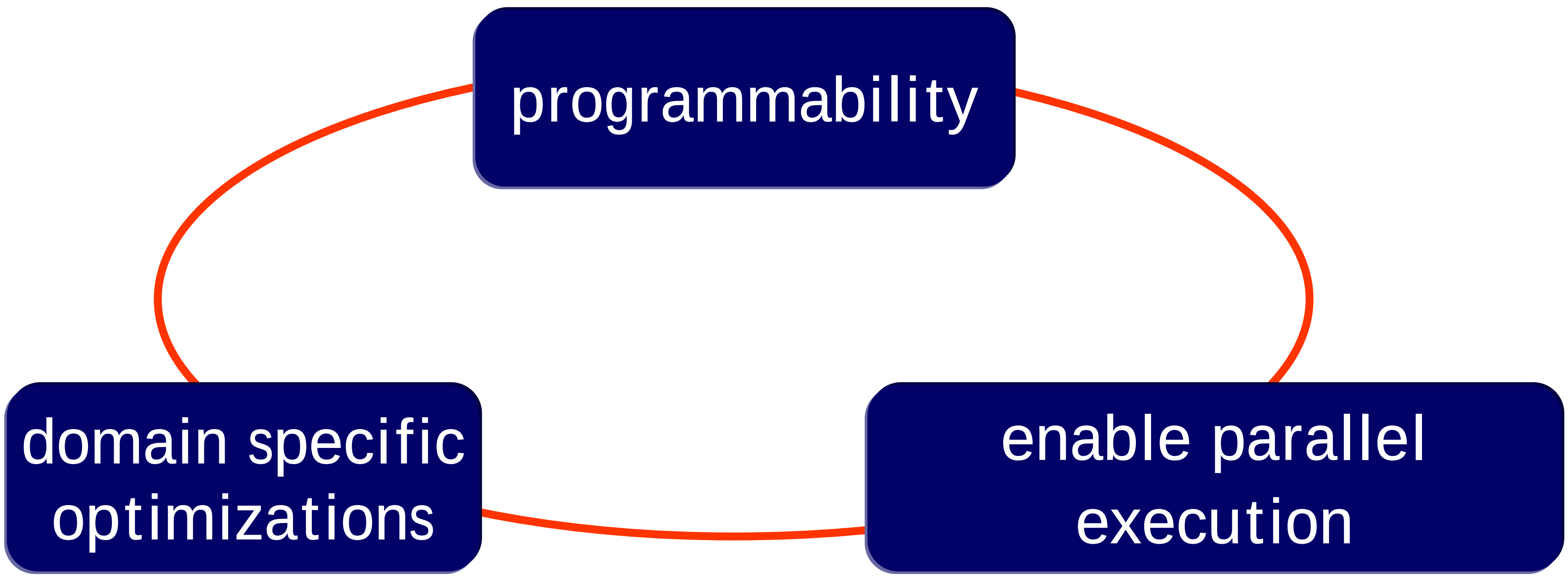
Target 4 core machine

Evaluation: Coarse-Grained Task + Data + Software Pipelining



Compiler-Aware Language Design

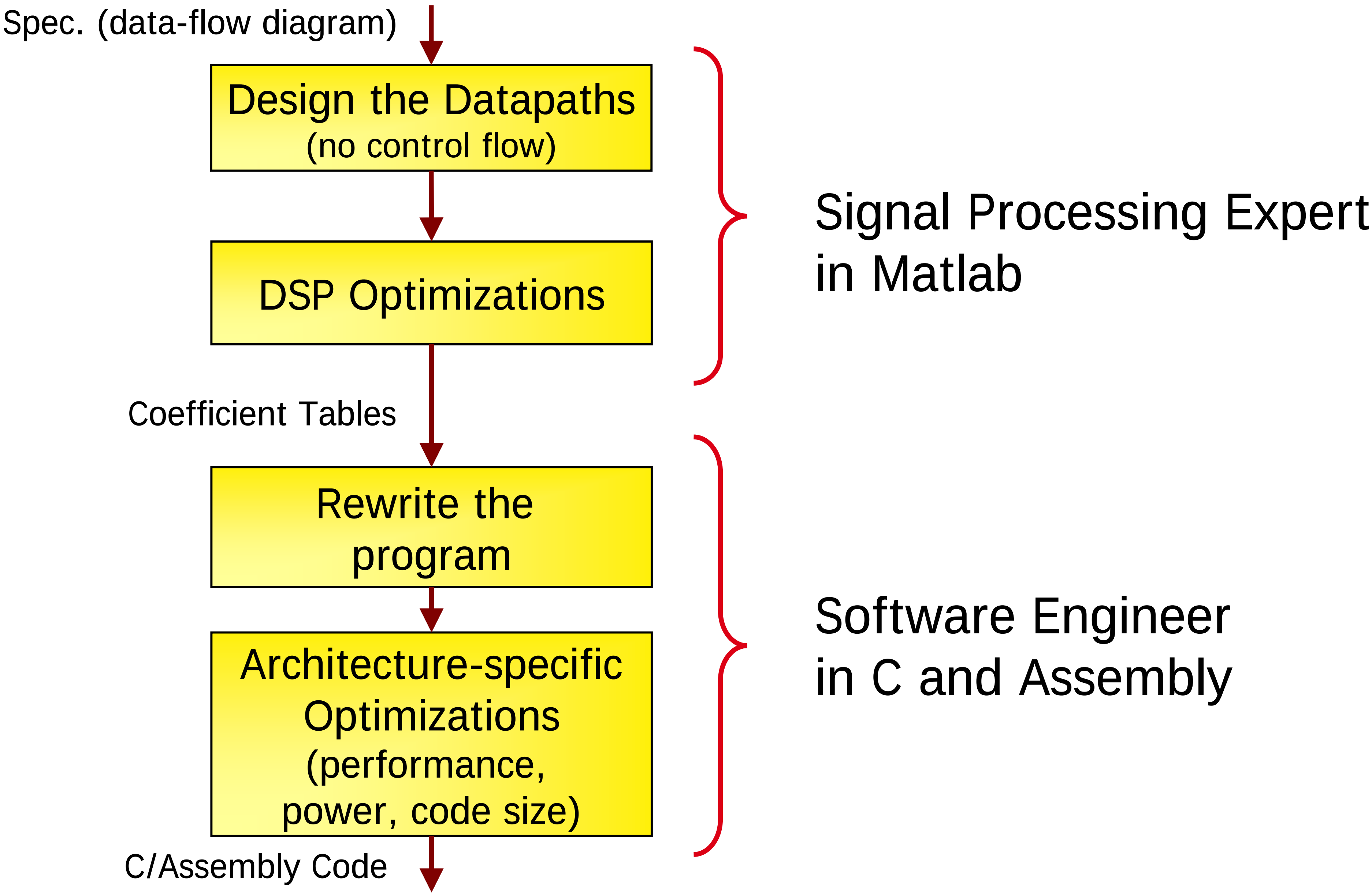
boost productivity, enable faster development and rapid prototyping



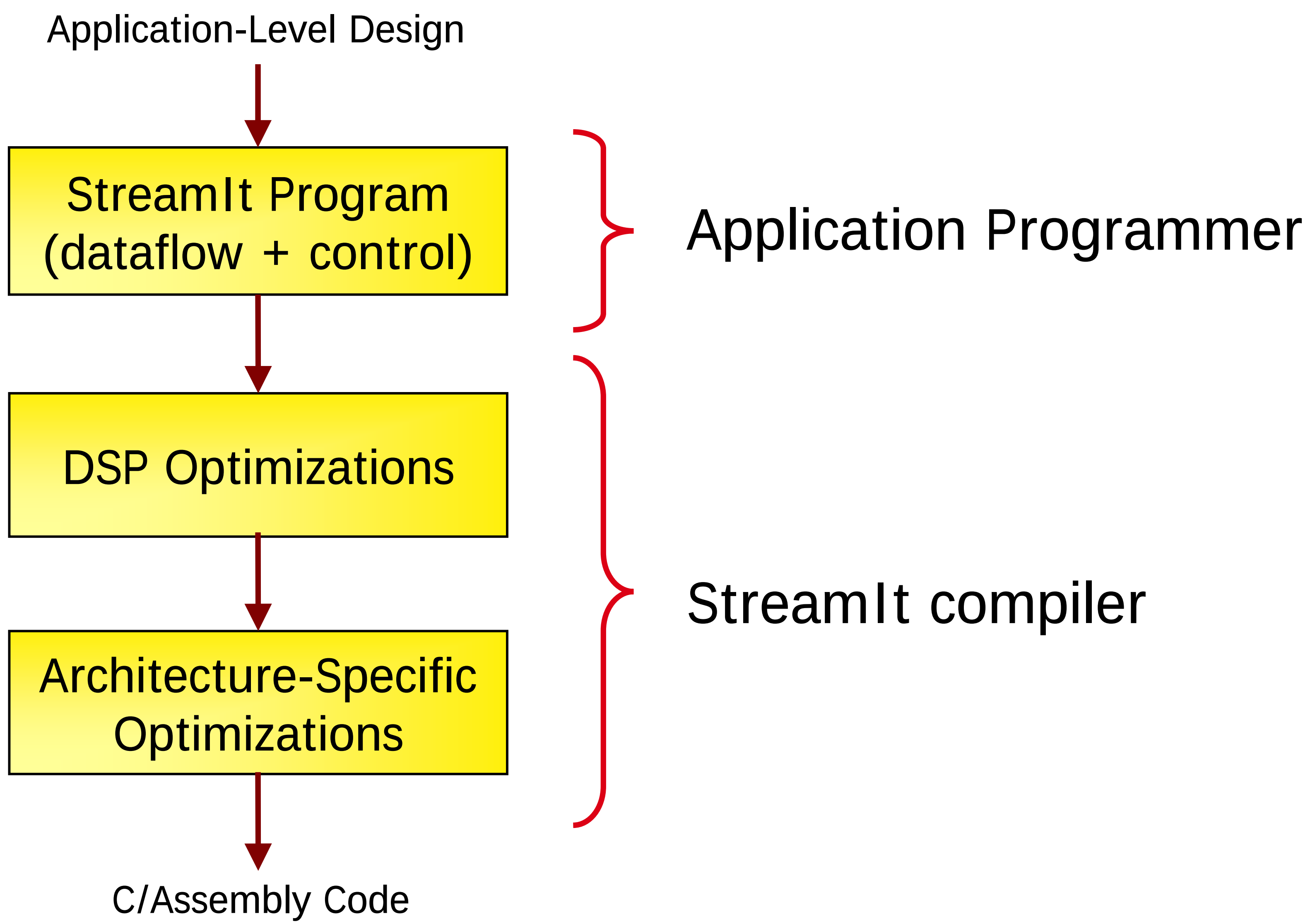
simple and effective optimizations for domain specific abstractions

target multicores, clusters, tiled architectures, DSPs, graphics processors, …

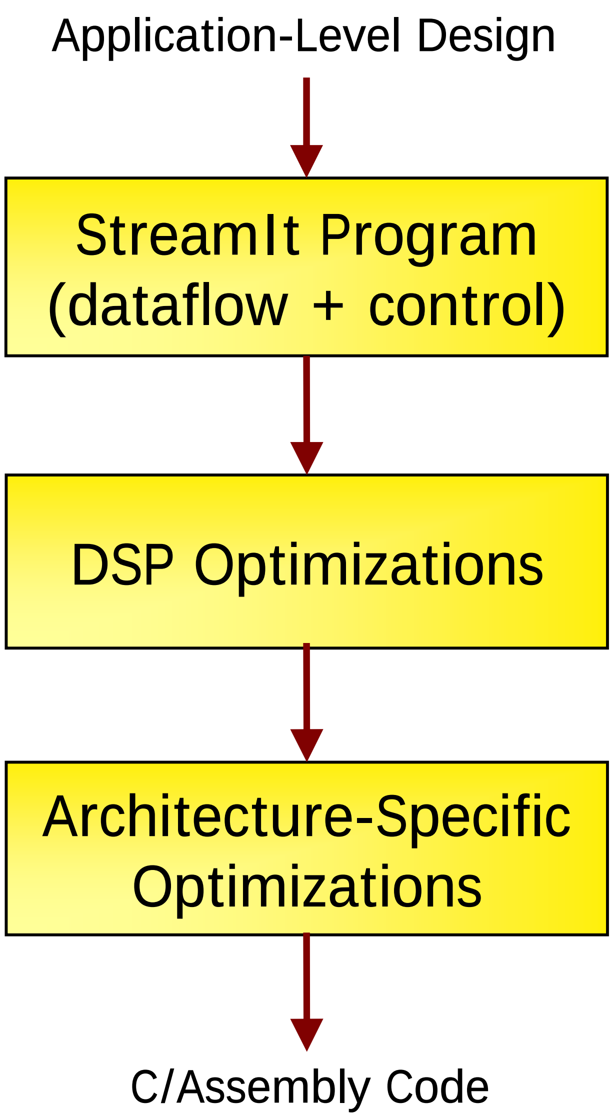
Conventional DSP Design Flow



Design Flow with StreamIt



Design Flow with StreamIt



- Benefits of programming in a single, high-level abstraction
  - Modular
  - Composable
  - Portable
  - Malleable
- The Challenge: Maintaining Performance

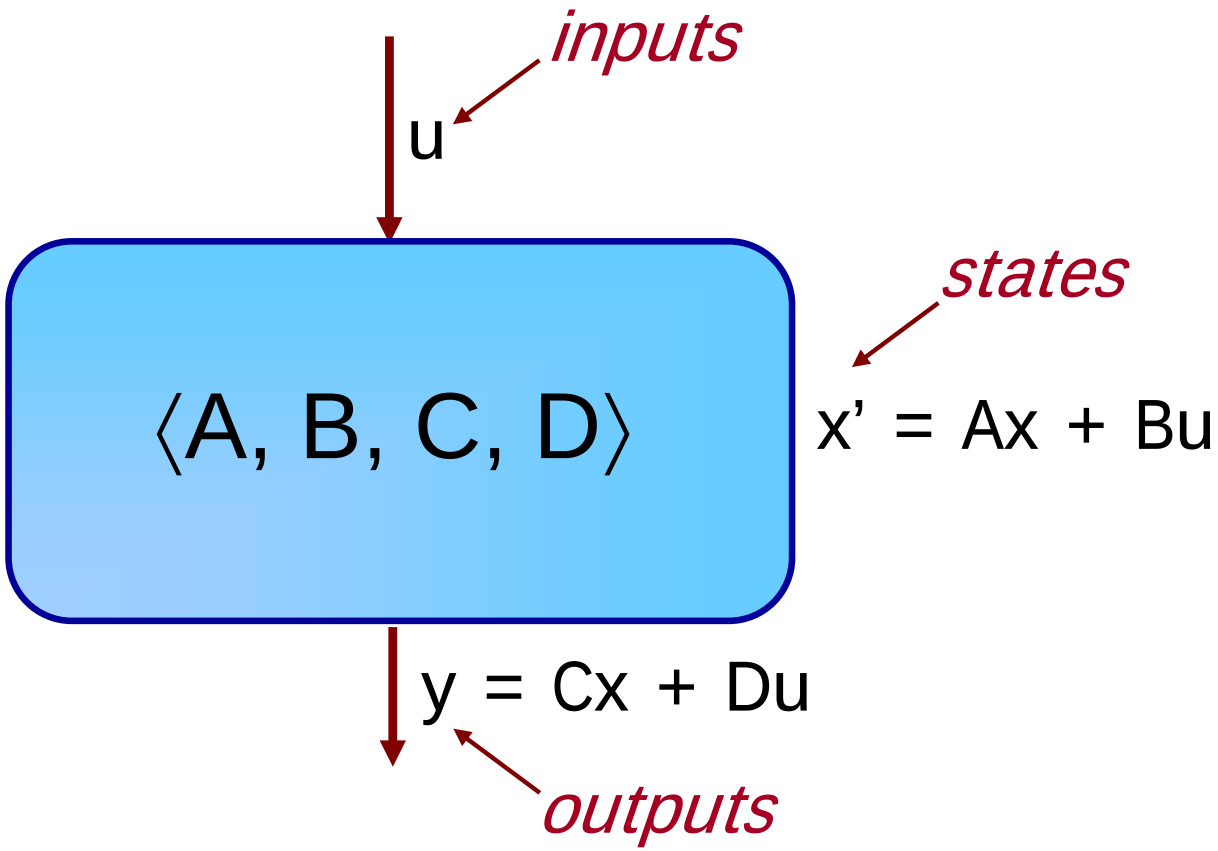
Focus: Linear State Space Filters

- Properties:
  1. Outputs are linear function of inputs and states
  2. New states are linear function of inputs and states

- Most common target of DSP optimizations
  - FIR / IIR filters
  - Linear difference equations
  - Upsamplers / downsamplers
  - DCTs

Representing State Space Filters

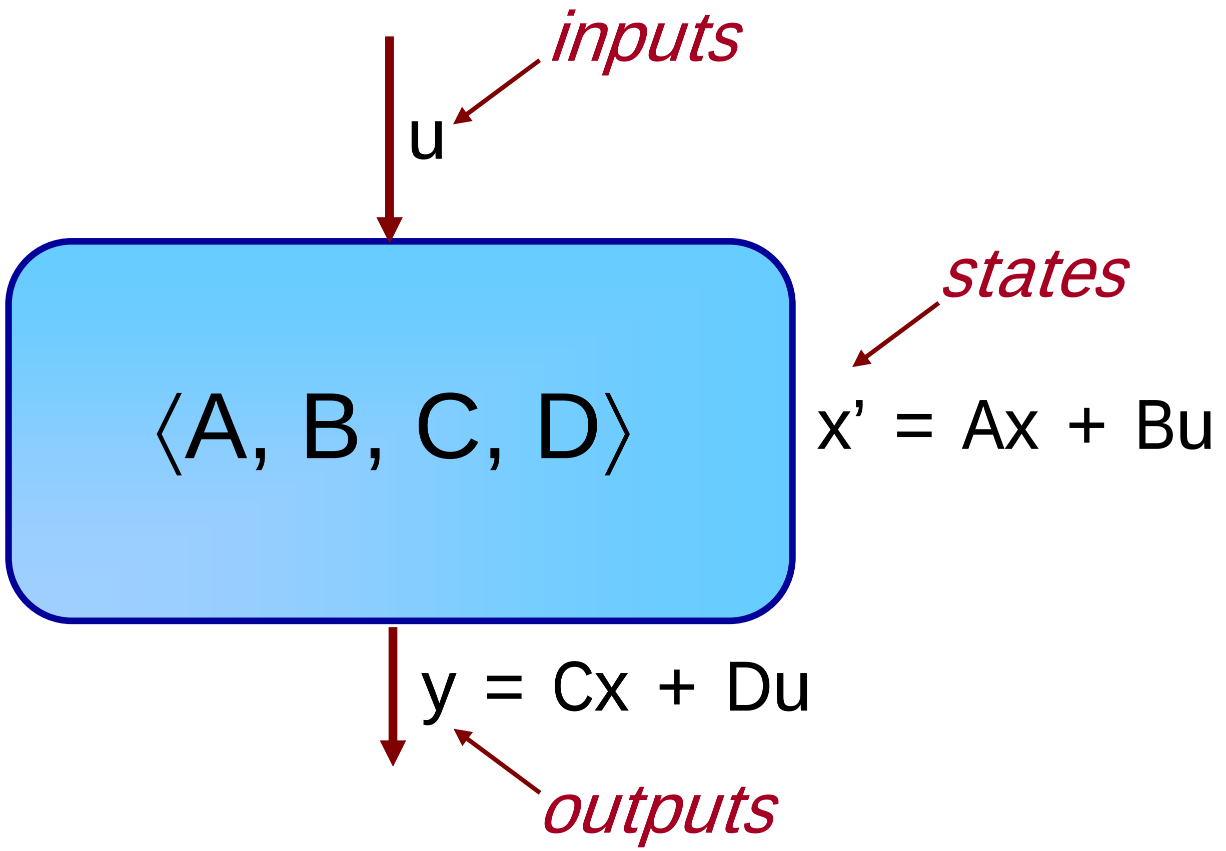
- A state space filter is a tuple ⟨A, B, C, D⟩



Representing State Space Filters

- A state space filter is a tuple ⟨A, B, C, D⟩

```
float->float filter IIR {
  float x1, x2;
  work push 1 pop 1 {
    float u = pop();
    push(2*(x1+x2+u));
    x1 = 0.9*x1 + 0.3*u;
    x2 = 0.9*x2 + 0.2*u;
} }
```



Representing State Space Filters

- A state space filter is a tuple ↑A, B, C, D〉

```
float->float filter IIR {
  float x1, x2;
  work push 1 pop 1 {
    float u = pop();
    push(2*(x1+x2+u));
    x1 = 0.9*x1 + 0.3*u;
    x2 = 0.9*x2 + 0.2*u;
} }
```

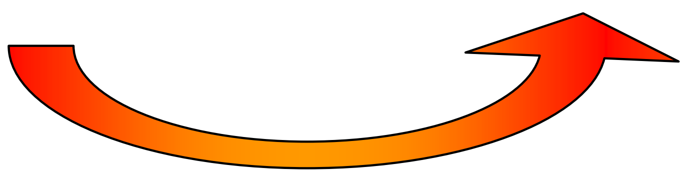
inputs u

$$A = \begin{bmatrix} 0.9 & 0 \\ 0 & 0.9 \end{bmatrix} \quad B = \begin{bmatrix} 0.3 \\ 0.2 \end{bmatrix}$$

$$C = \begin{bmatrix} 2 & 2 \end{bmatrix} \quad D = \begin{bmatrix} 2 \end{bmatrix}$$

states $x' = Ax + Bu$

$y = Cx + Du$ *outputs*

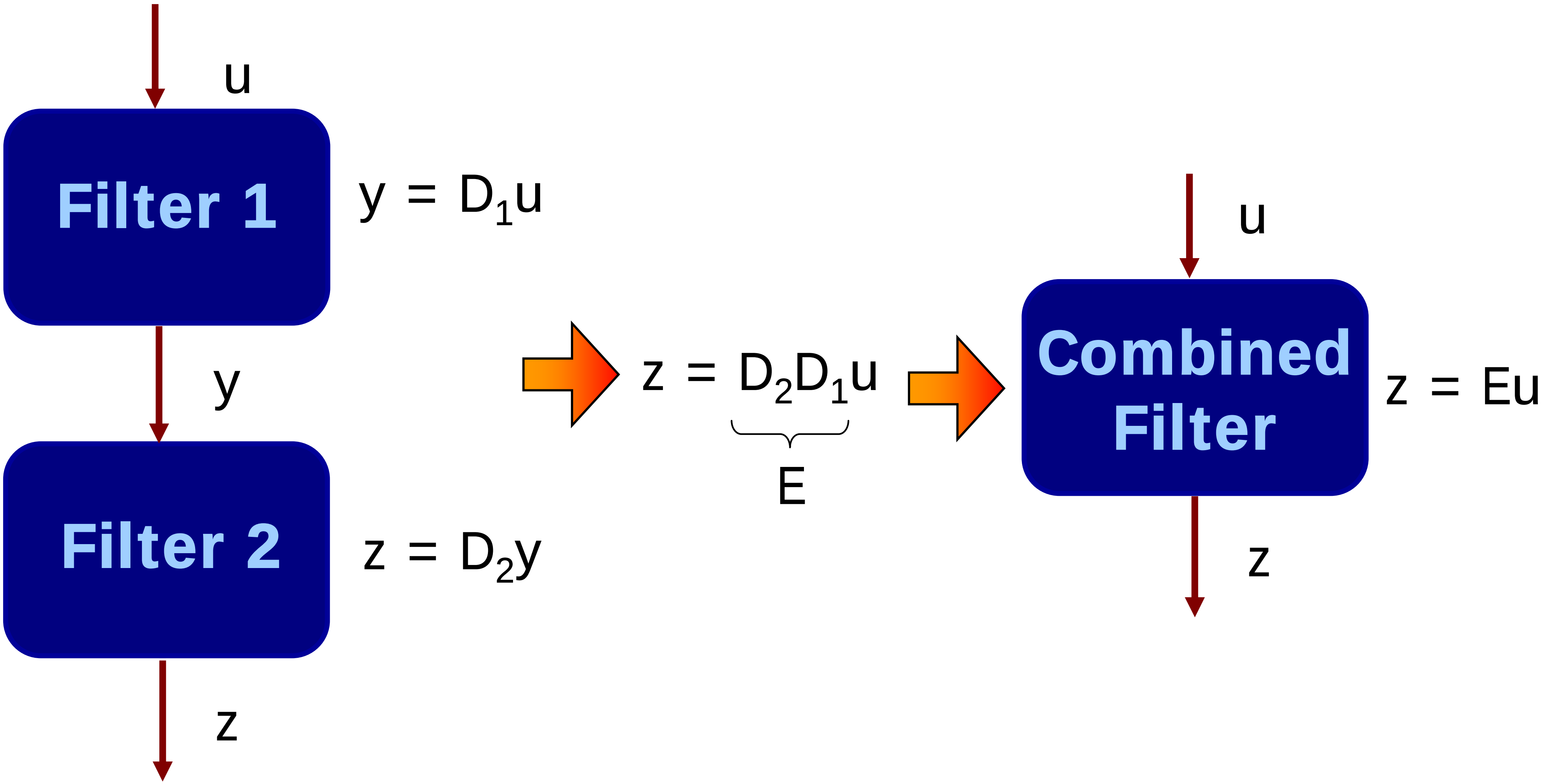


Linear dataflow analysis

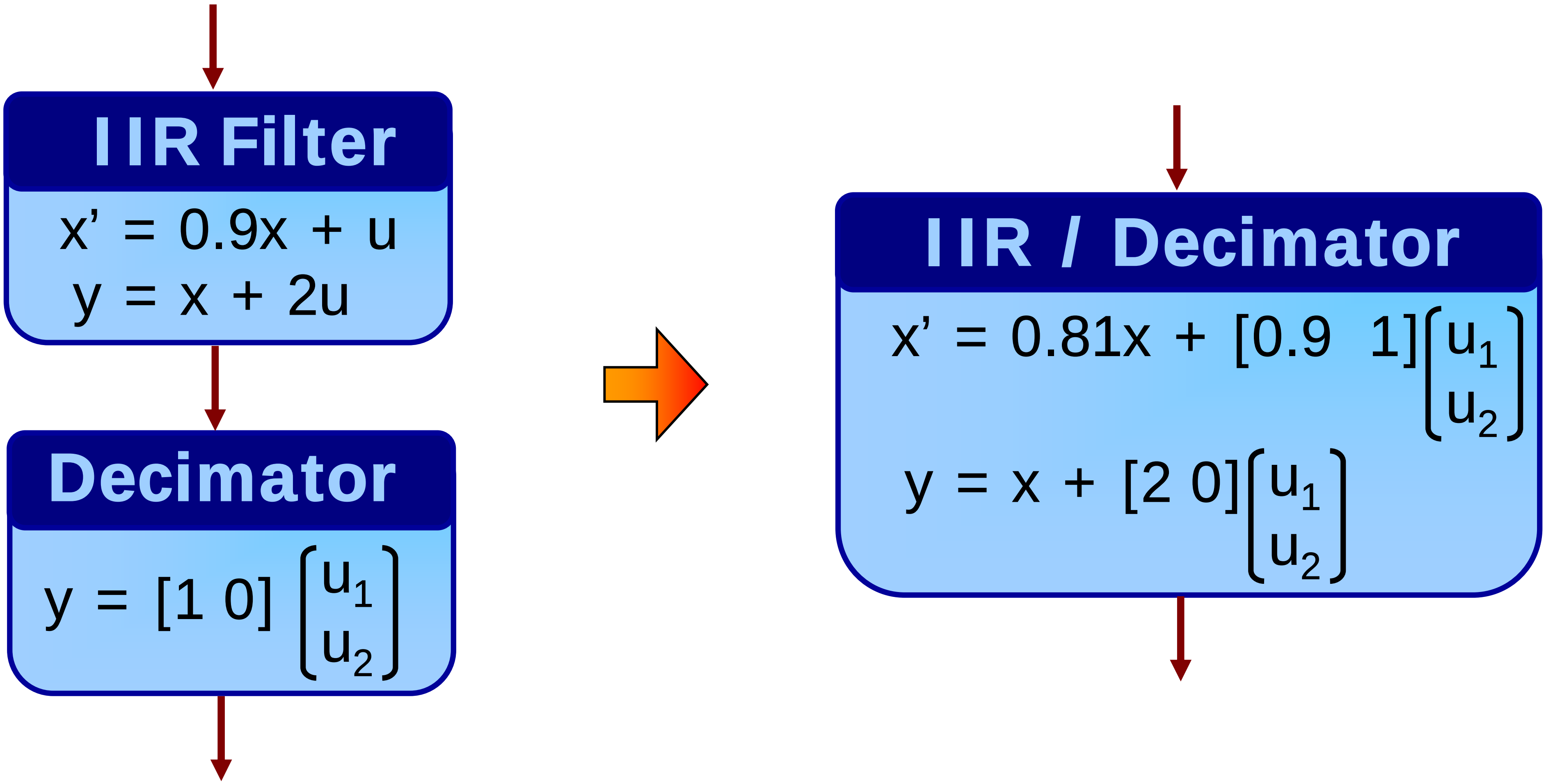
Linear Optimizations

1. Combining adjacent filters
2. Transformation to frequency domain
3. Change of basis transformations
4. Transformation Selection

1) Combining Adjacent Filters



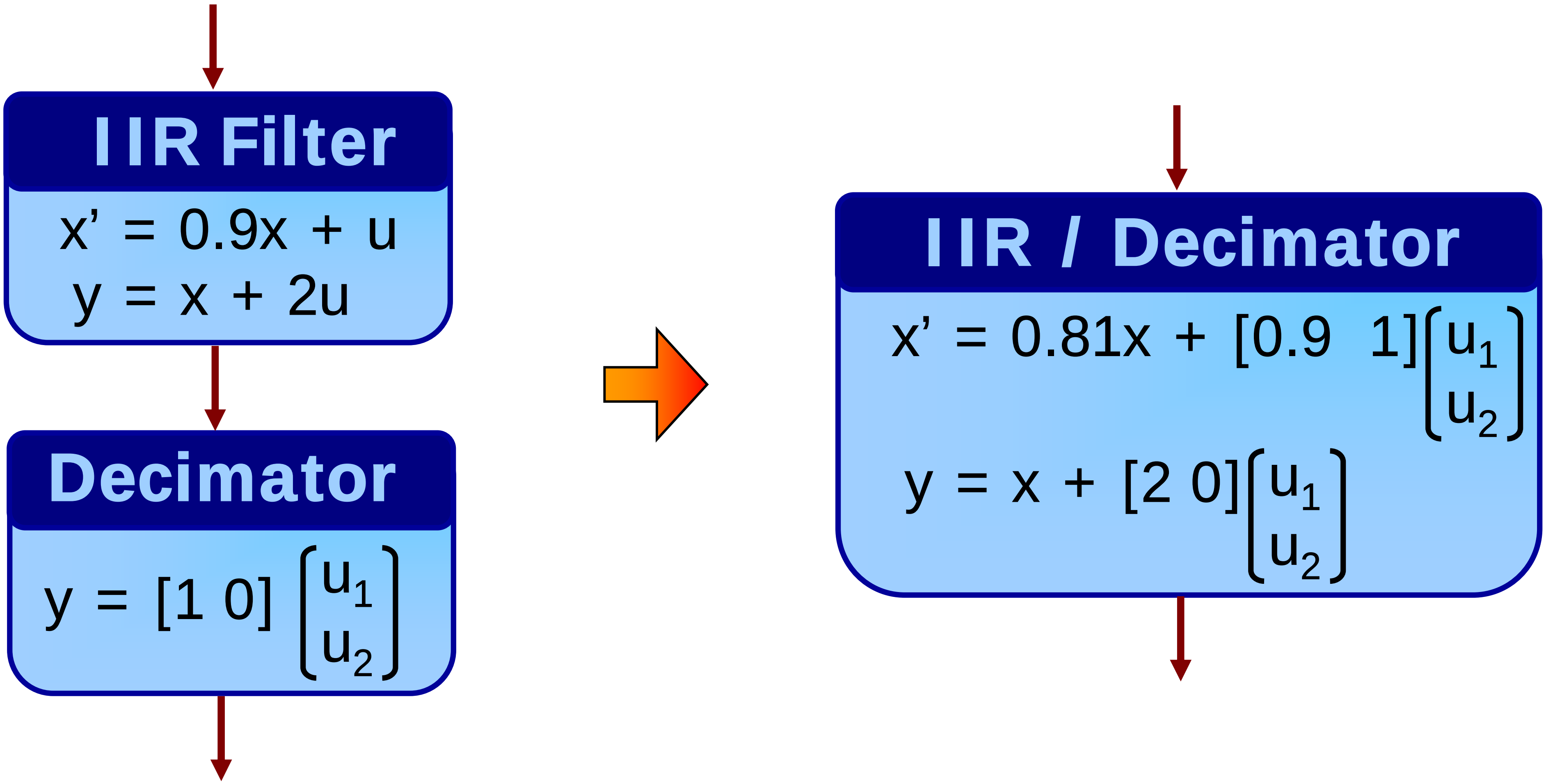
Combination Example



$$\frac{\text{8 FLOPs}}{\text{output}}$$ ☹

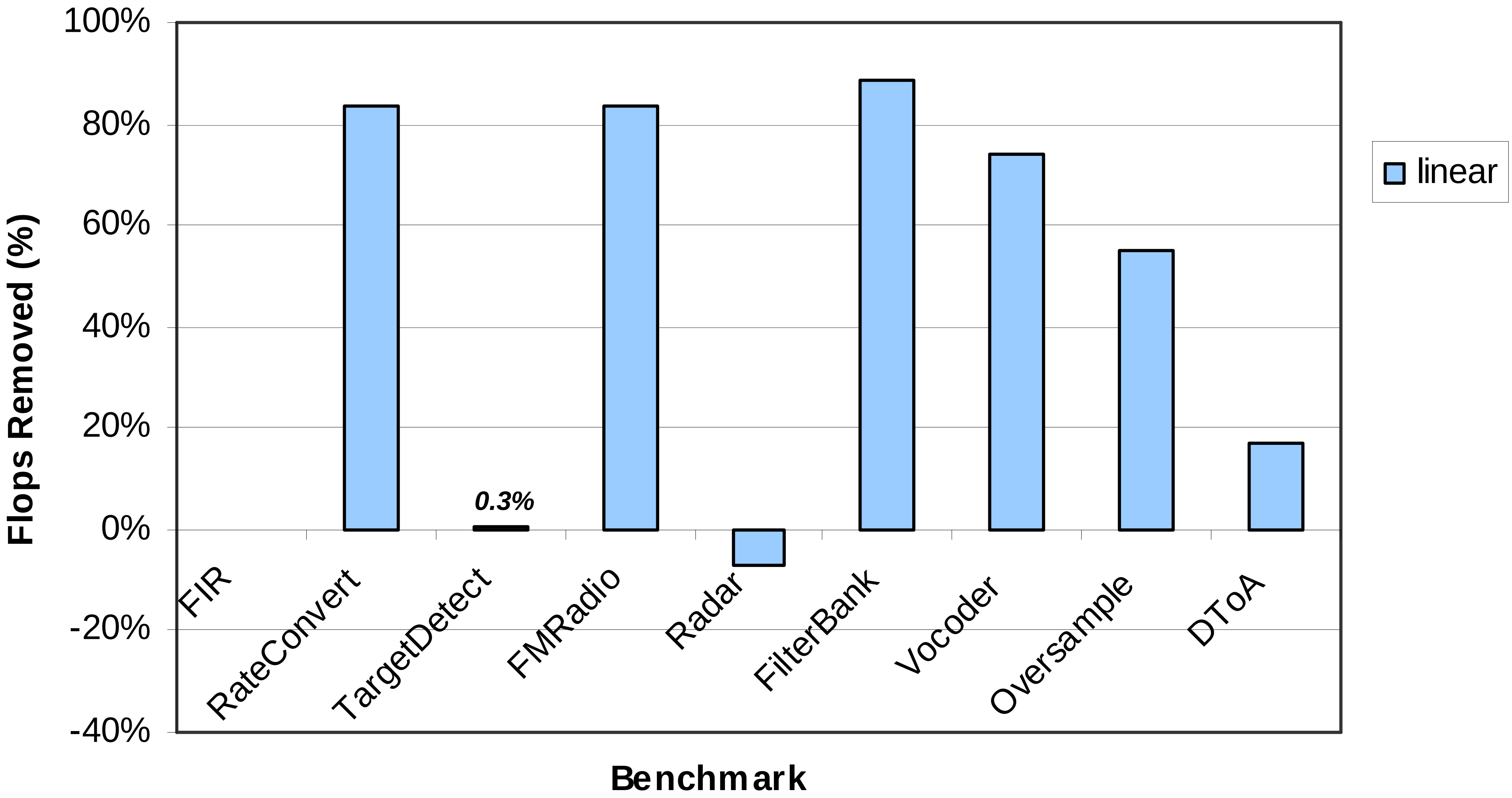
$$\frac{\text{6 FLOPs}}{\text{output}}$$ ☺

Combination Example



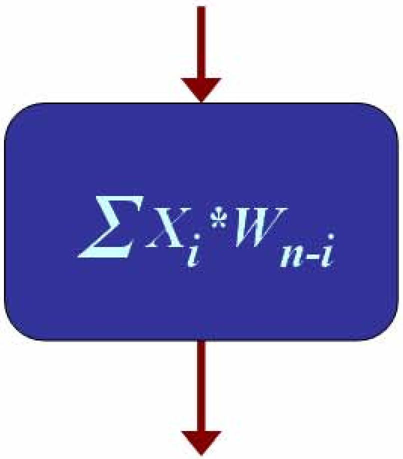
As decimation factor goes to ∞,
eliminate up to 75% of FLOPs.

Floating-Point Operations Reduction



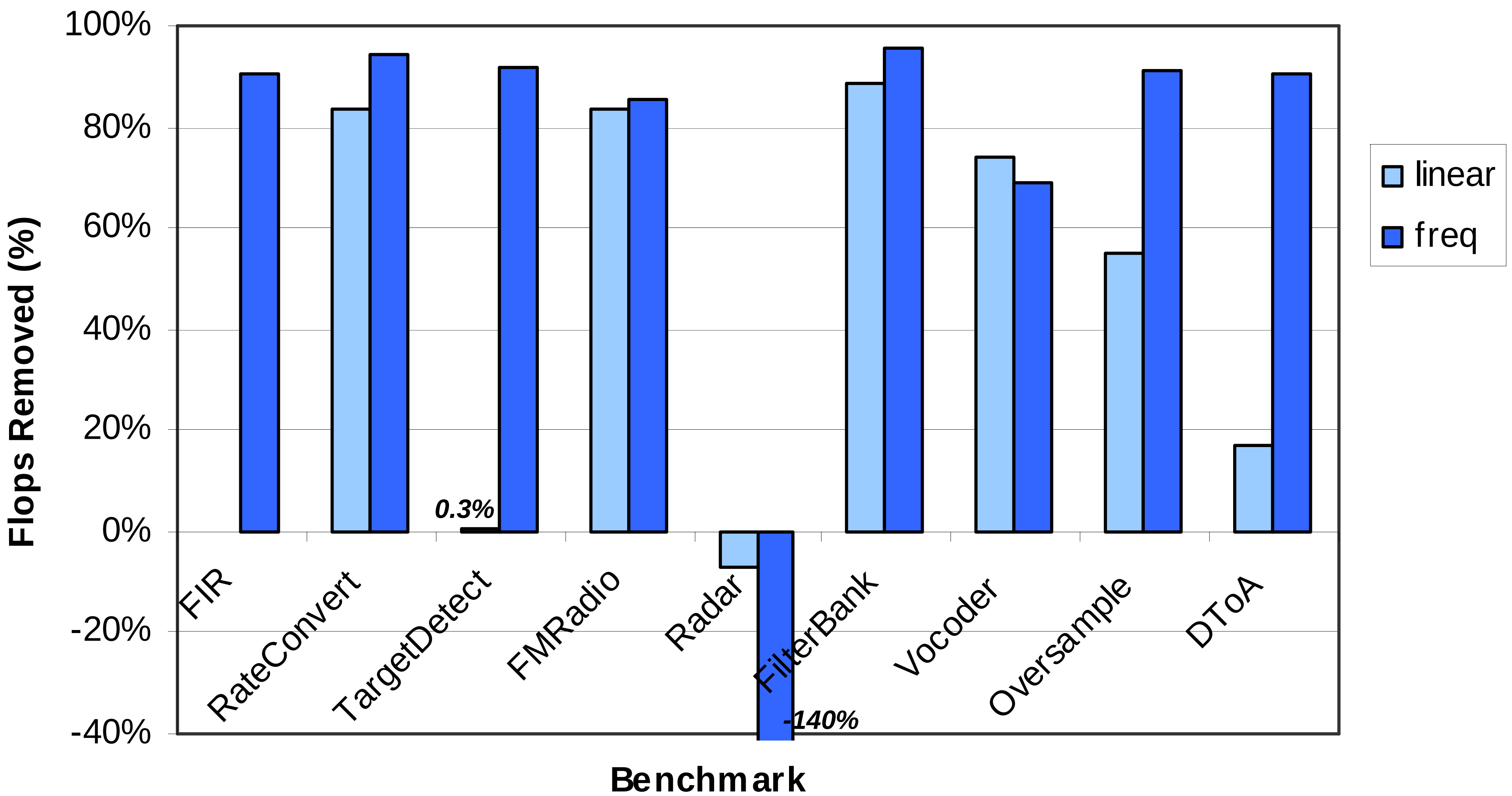
2) From Time to Frequency Domain

- Convolutions can be done cheaply in the Frequency Domain



- Painful to do by hand
  - Blocking
  - Coefficient calculations
  - Startup etc.

FLOPs Reduction



3) Change-of-Basis Transformation

$$x' = Ax + Bu$$
$$y = Cx + Du$$

T = invertible matrix, $z = Tx$

$$z' = A'z + B'u$$
$$y = C'z + D'u$$

$A' = TAT^{-1}$ $B' = TB$
$C' = CT^{-1}$ $D' = D$

Can map original states x to transformed states $z = Tx$ without changing I/O behavior

Change-of-Basis Optimizations

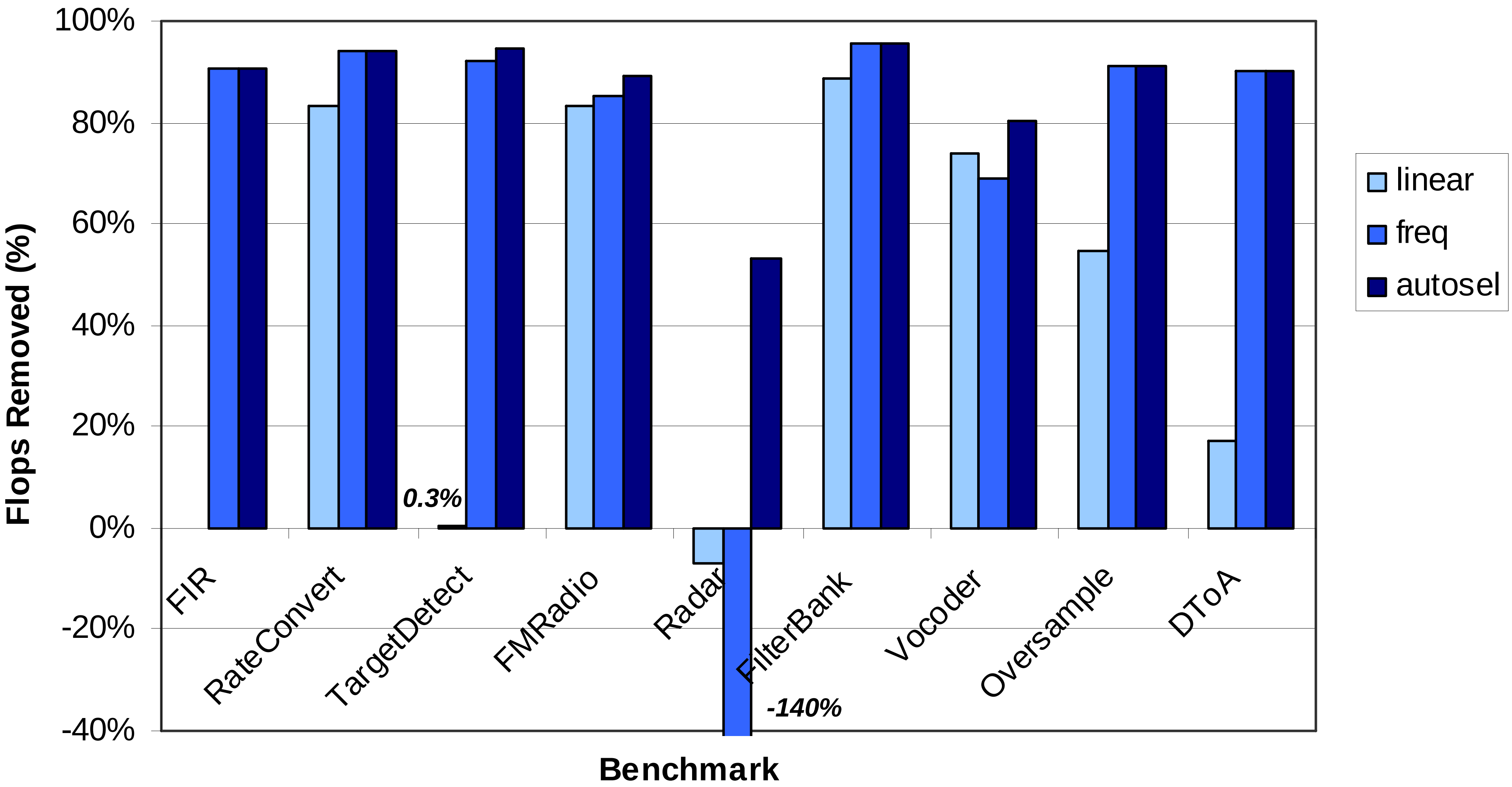
1. State removal
   - Minimize number of states in system
2. Parameter reduction
   - Increase number of 0’s and 1’s in multiplication

→ Formulated as general matrix operations

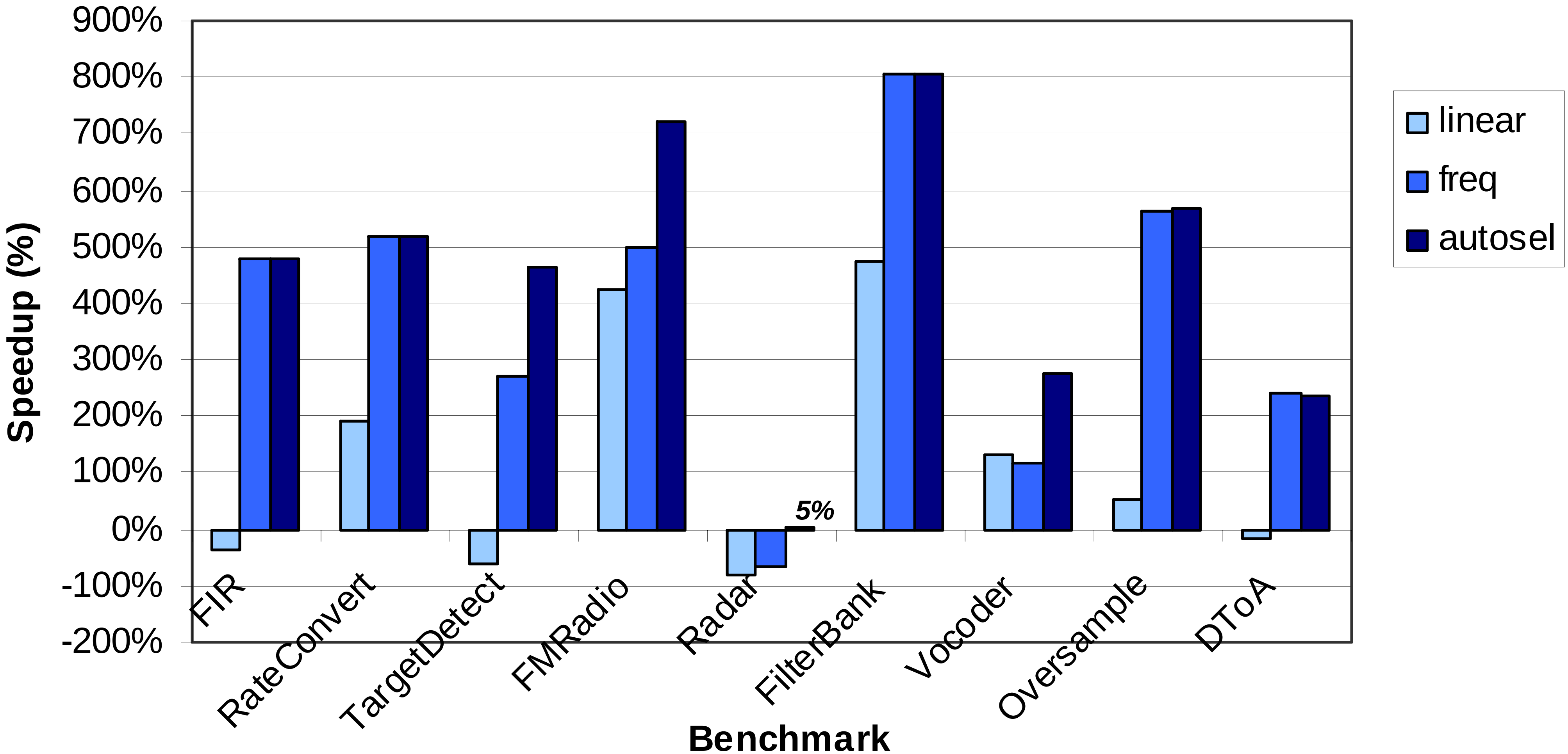
4) Transformation Selection

- When to apply what transformations?
  - Linear filter combination can increase the computation cost
  - Shifting to the Frequency domain is expensive for filters with pop > 1
    - Compute all outputs, then decimate by pop rate
  - Some expensive transformations may later enable other transformations, reducing the overall cost

FLOPs Reduction with Optimization Selection



Execution Speedup



On a Pentium IV

Conclusion

- Streaming programming model
  - Can break the von Neumann bottleneck
  - A natural fit for a large class of applications
  - An ideal machine language for multicores.
- Natural programming language for many streaming applications
  - Better modularity, composability, malleability and portability than C
- Compiler can easily extract explicit and inherent parallelism
  - Parallelism is abstracted away from architectural details of multicores
  - Sustainable Speedups (5x to 19x on the 16 core Raw)
- Can we replace the DSP engineer from the design flow?
  - On the average 90% of the FLOPs eliminated, average speedup of 450% attained
- Increased abstraction does not have to sacrifice performance
- The compiler and benchmarks are available on the web

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

Thanks for Listening!

Any questions?

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