Exploiting Coarse-Grained Task, Data, and Pipeline Parallelism in Stream Programs

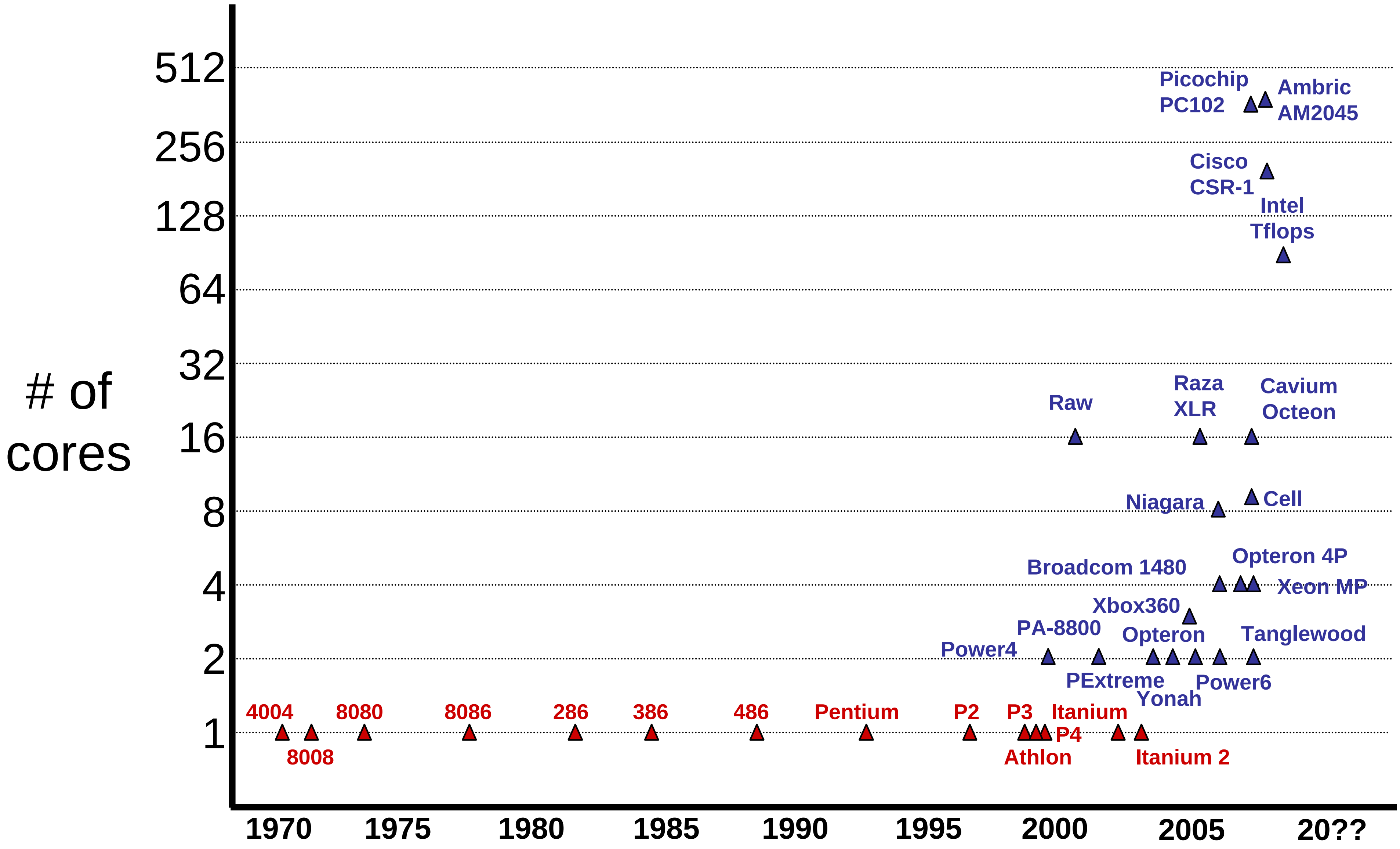
Michael Gordon, William Thies, and
Saman Amarasinghe

Massachusetts Institute of Technology

ASPLOS
October 2006
San Jose, CA

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

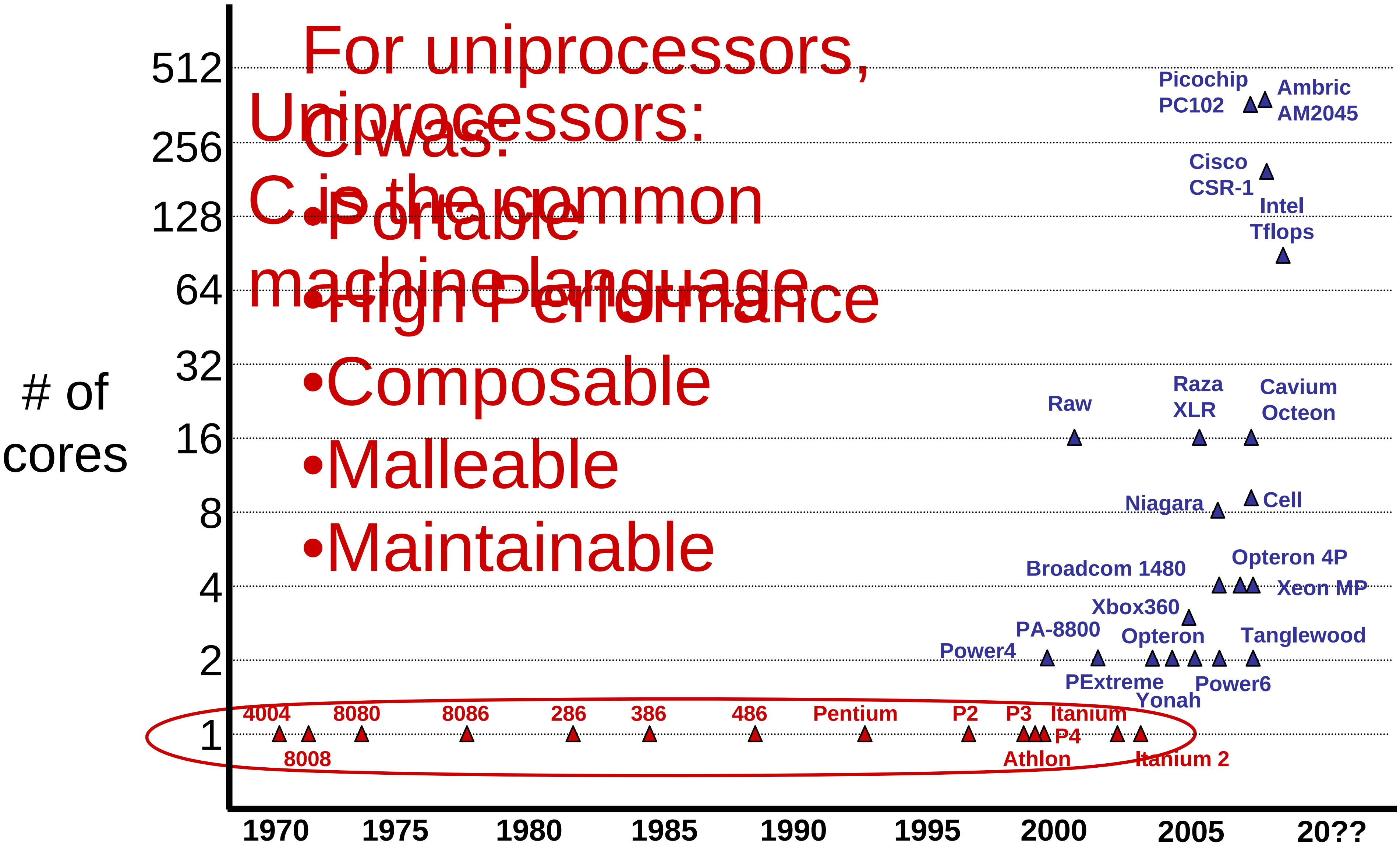
Multicores Are Here!



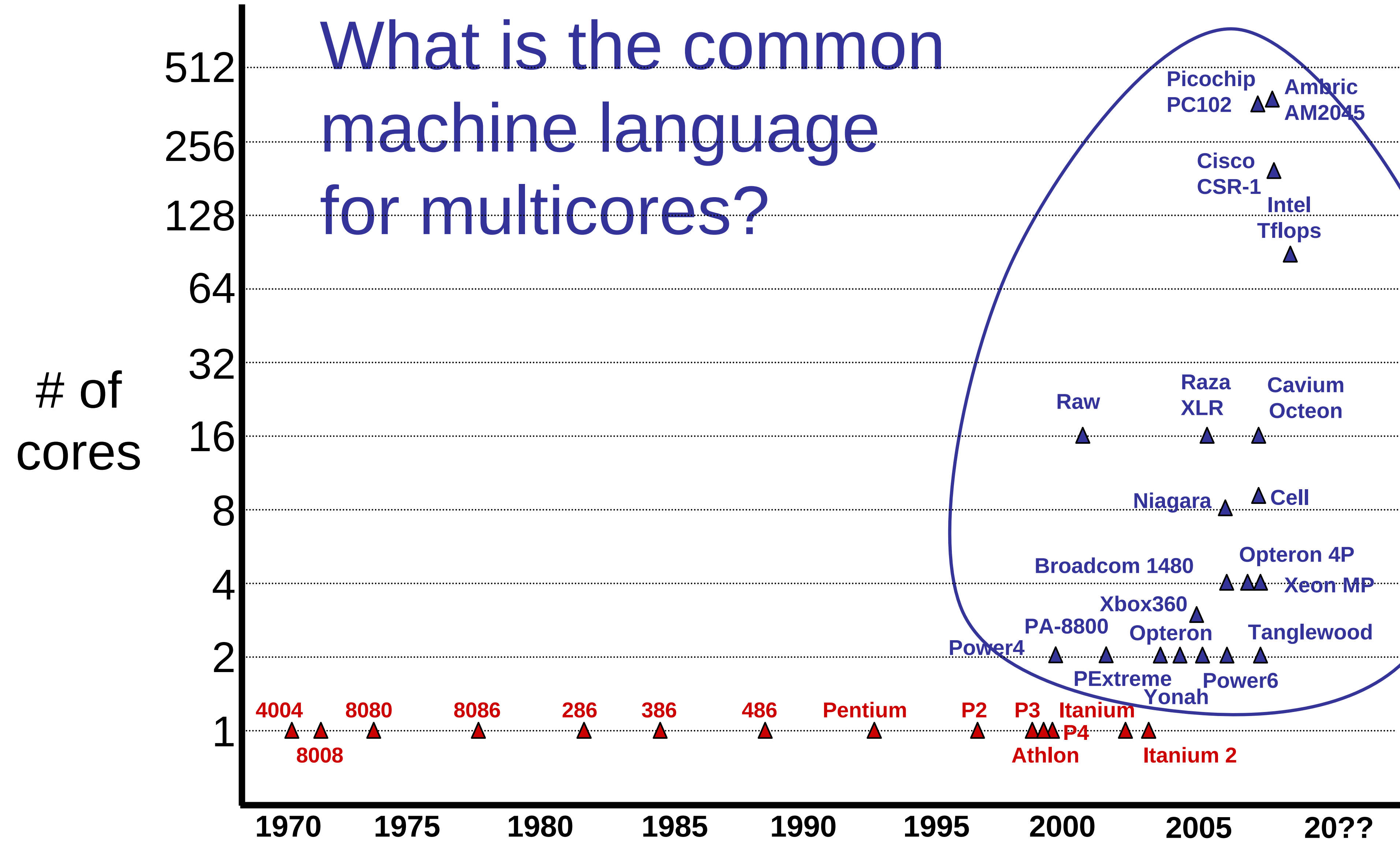
Multicores Are Here!

For uniprocessors, C was:
- •Portable
- •High Performance
- •Composable
- •Malleable
- •Maintainable

Uniprocessors: C is the common machine language



Multicores Are Here!



Common Machine Languages

Uniprocessors:

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:

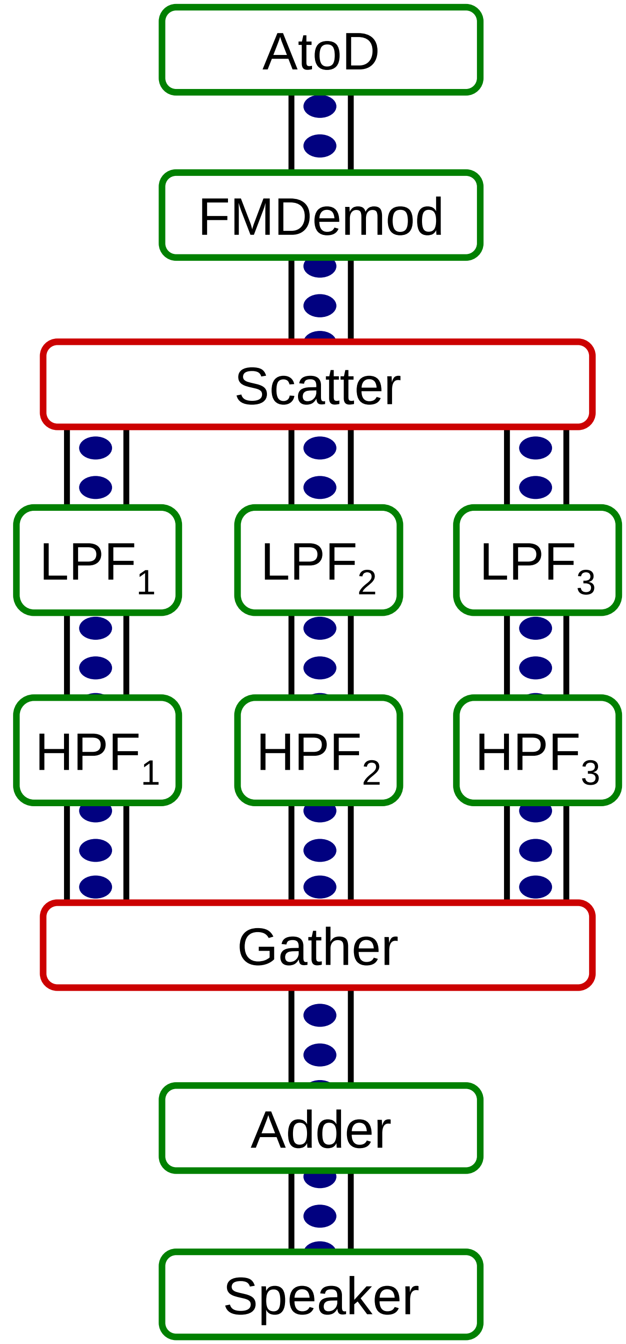
Common Properties
Multiple flows of control
Multiple local memories

Differences:
Number and capabilities of cores
Communication Model
Synchronization Model

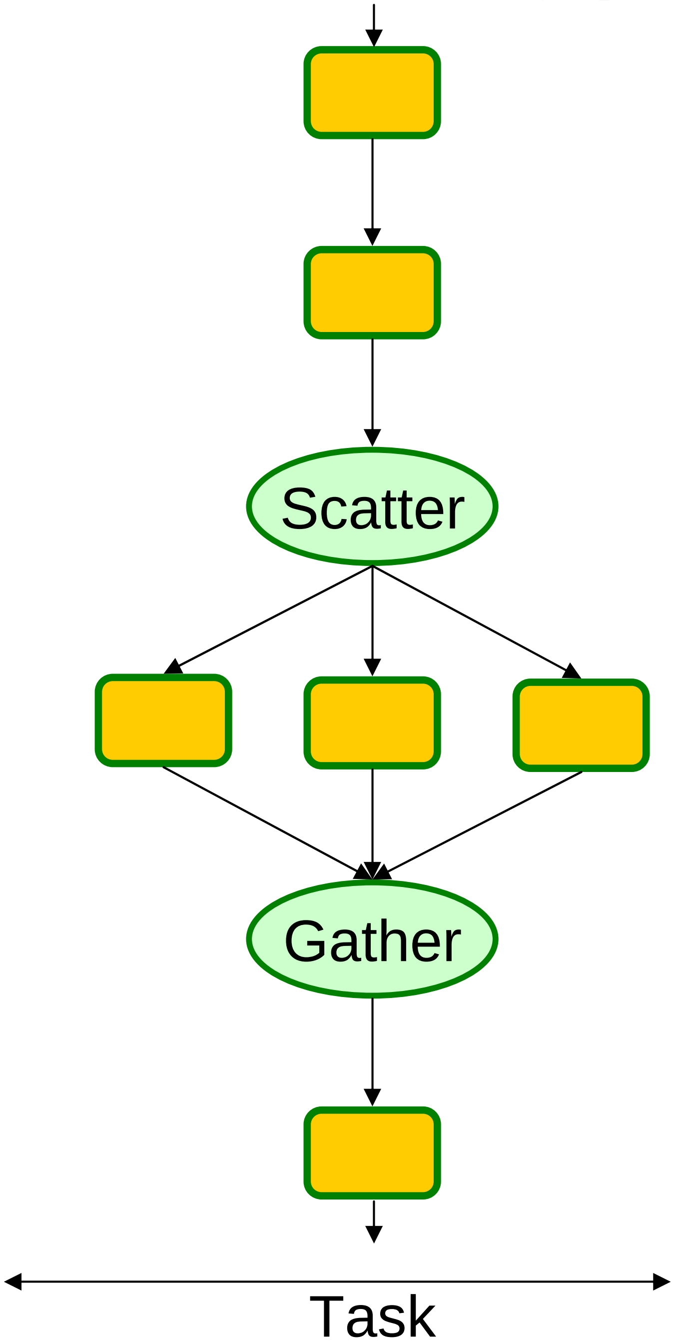
Need common machine language(s) for multicores

Streaming as a Common Machine Language

- Regular and repeating computation
- Independent filters
  with explicit communication
  – Segregated address spaces and multiple program counters
- Natural expression of Parallelism:
  – Producer / Consumer dependencies
  – Enables powerful, whole-program transformations



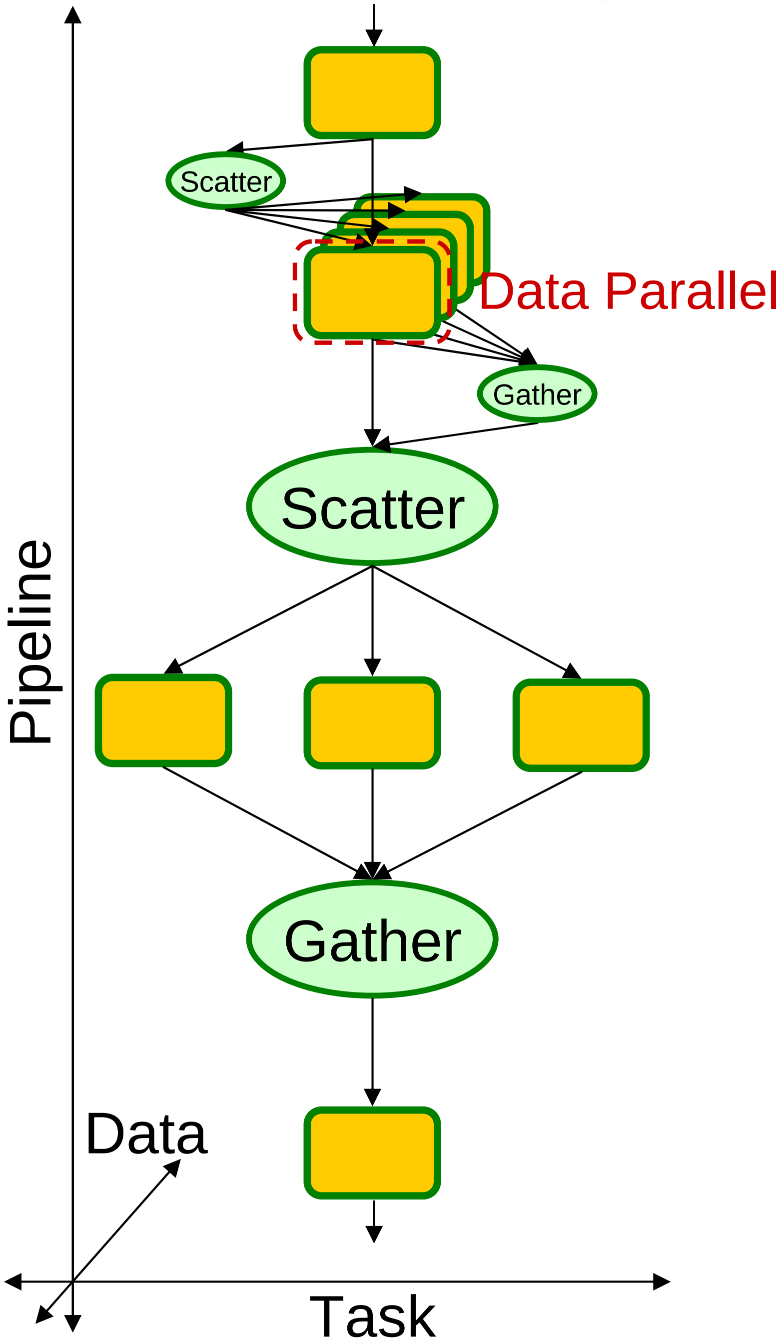
Types of Parallelism



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

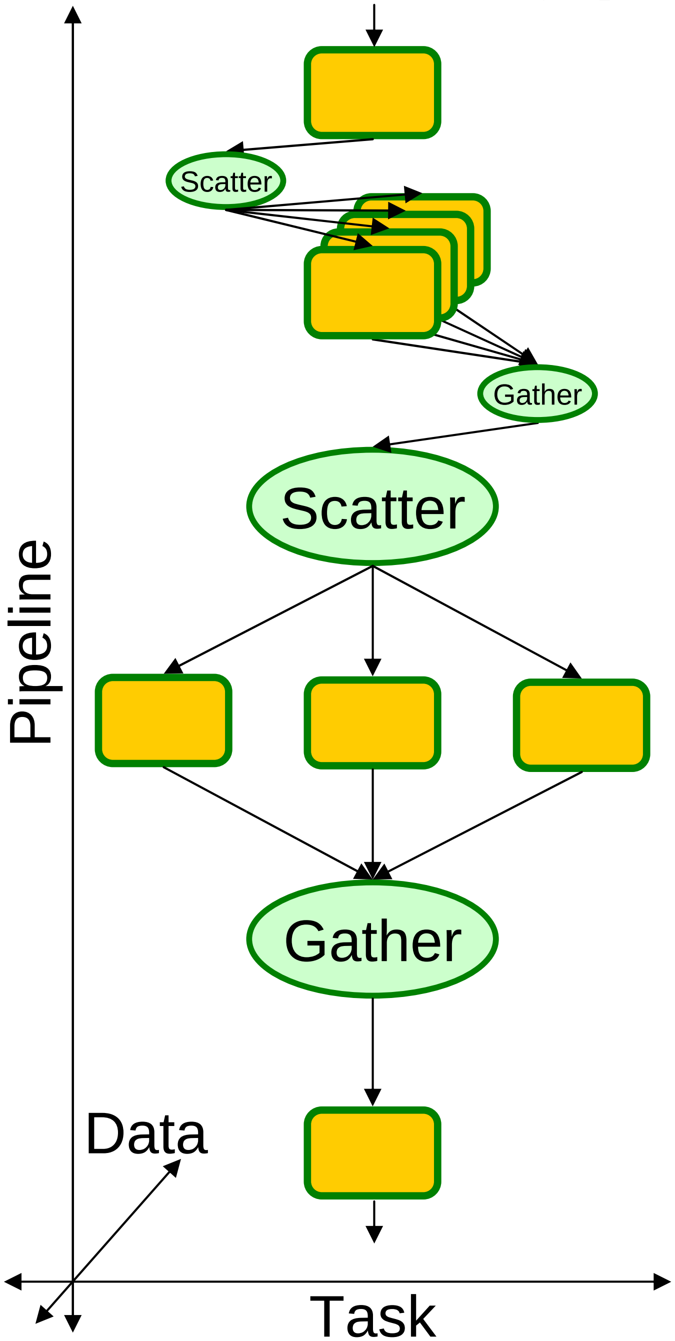
Data Parallelism

- 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

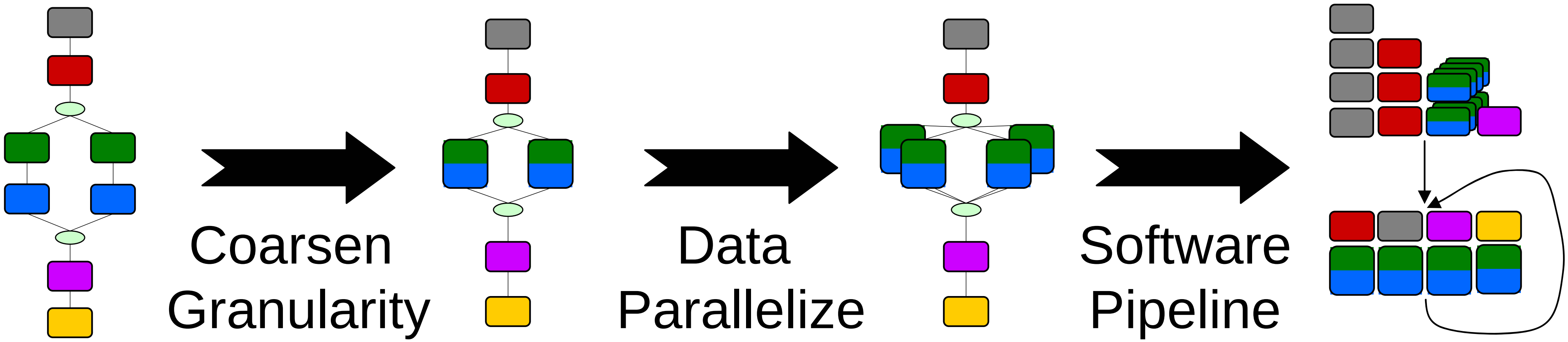
Given:

- 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

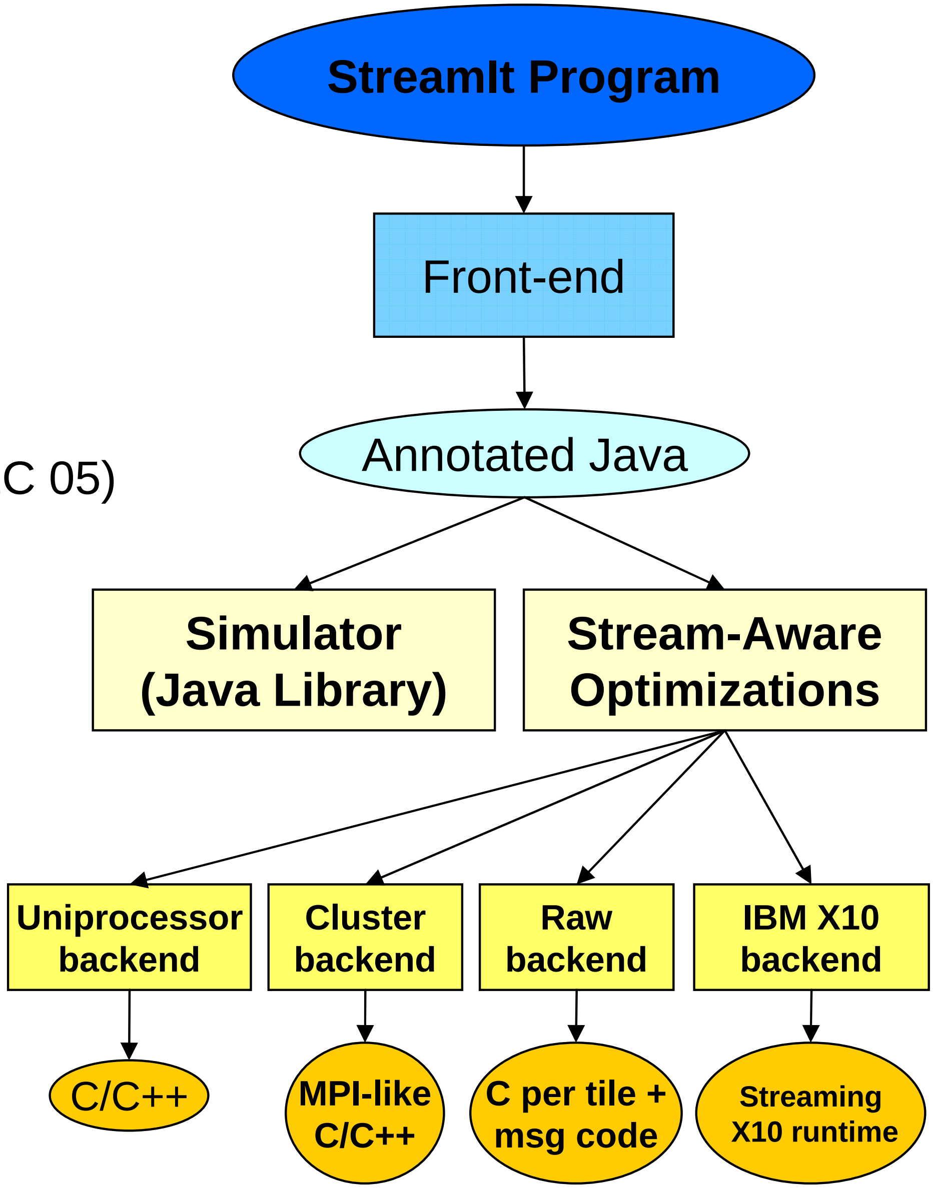
– 11.2x mean throughput speedup over single core

Outline

- StreamIt language overview
- Mapping to multicores
  - Baseline techniques
  - Our 3-phase solution

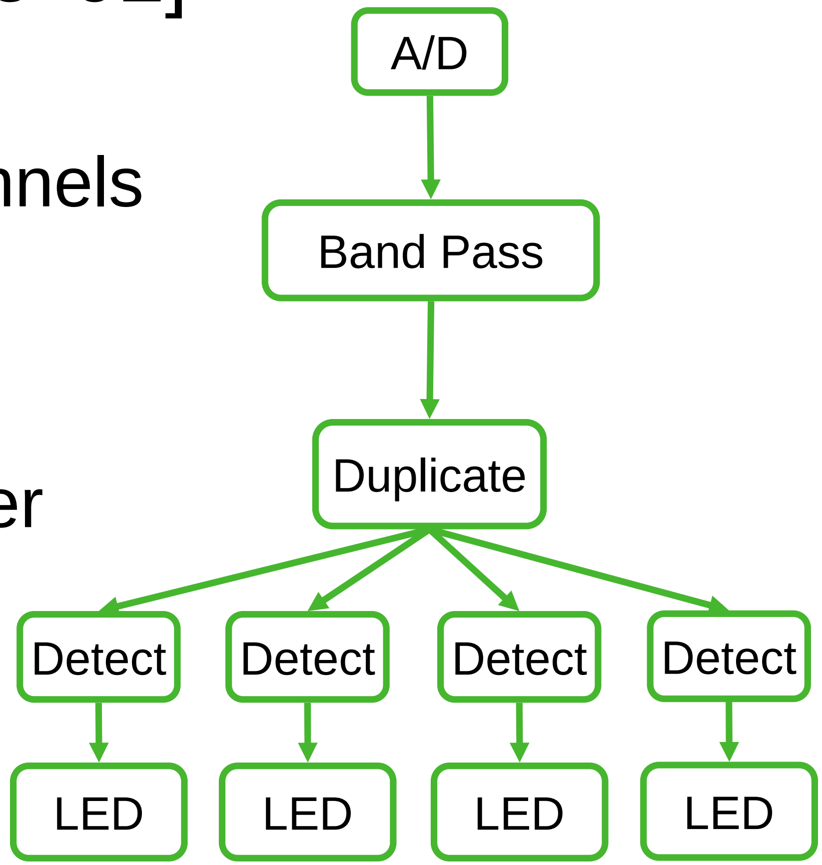
The StreamIt Project

- **Applications**
  - DES and Serpent [PLDI 05]
  - MPEG-2 [IPDPS 06]
  - SAR, DSP benchmarks, JPEG, …
- **Programmability**
  - StreamIt Language (CC 02)
  - Teleport Messaging (PPOPP 05)
  - Programming Environment in Eclipse (P-PHEC 05)
- **Domain Specific Optimizations**
  - Linear Analysis and Optimization (PLDI 03)
  - Optimizations for bit streaming (PLDI 05)
  - Linear State Space Analysis (CASES 05)
- **Architecture Specific Optimizations**
  - Compiling for Communication-Exposed Architectures (ASPLOS 02)
  - Phased Scheduling (LCTES 03)
  - Cache Aware Optimization (LCTES 05)
  - Load-Balanced Rendering (Graphics Hardware 05)

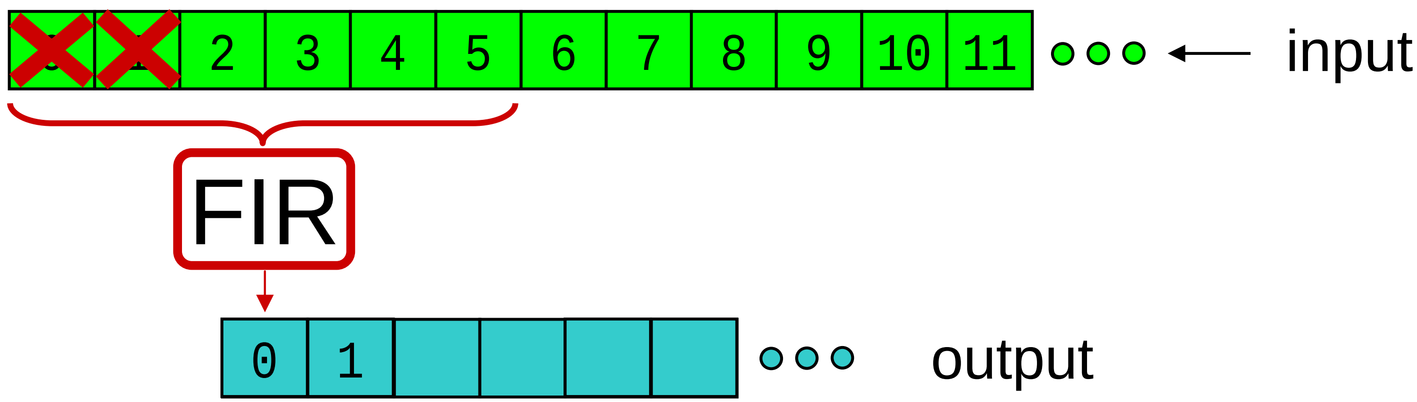


Model of Computation

- Synchronous Dataflow [Lee '92]
  - Graph of autonomous filters
  - Communicate via FIFO channels
- Static I/O rates
  - Compiler decides on an order of execution (schedule)
  - Static estimation of computation



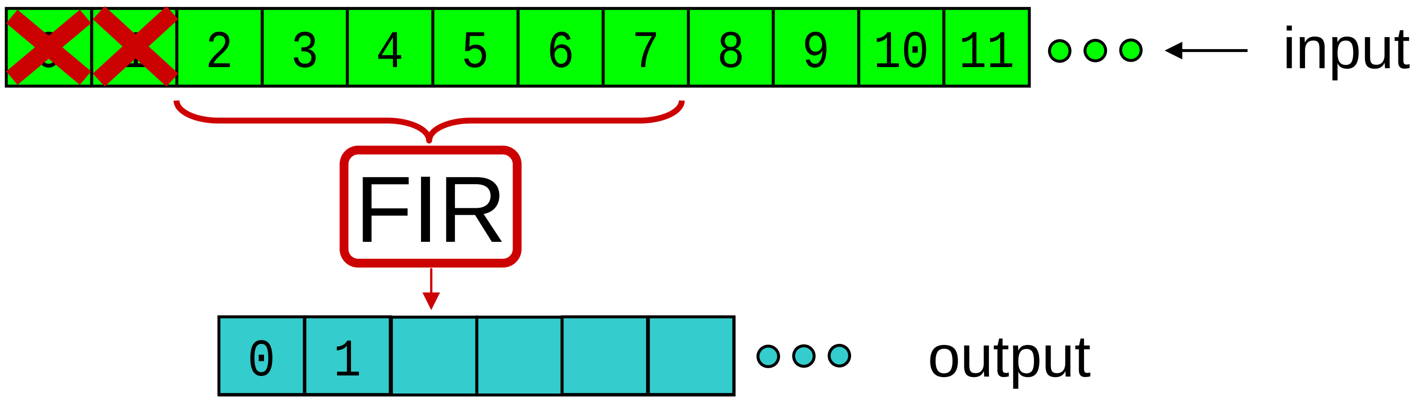
Example StreamIt Filter




```
float→float filter FIR  (int N, float[N] weights) {
                                                     Stateless
    work push 1 pop 1 peek N {
        float result = 0;

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

Example StreamIt Filter

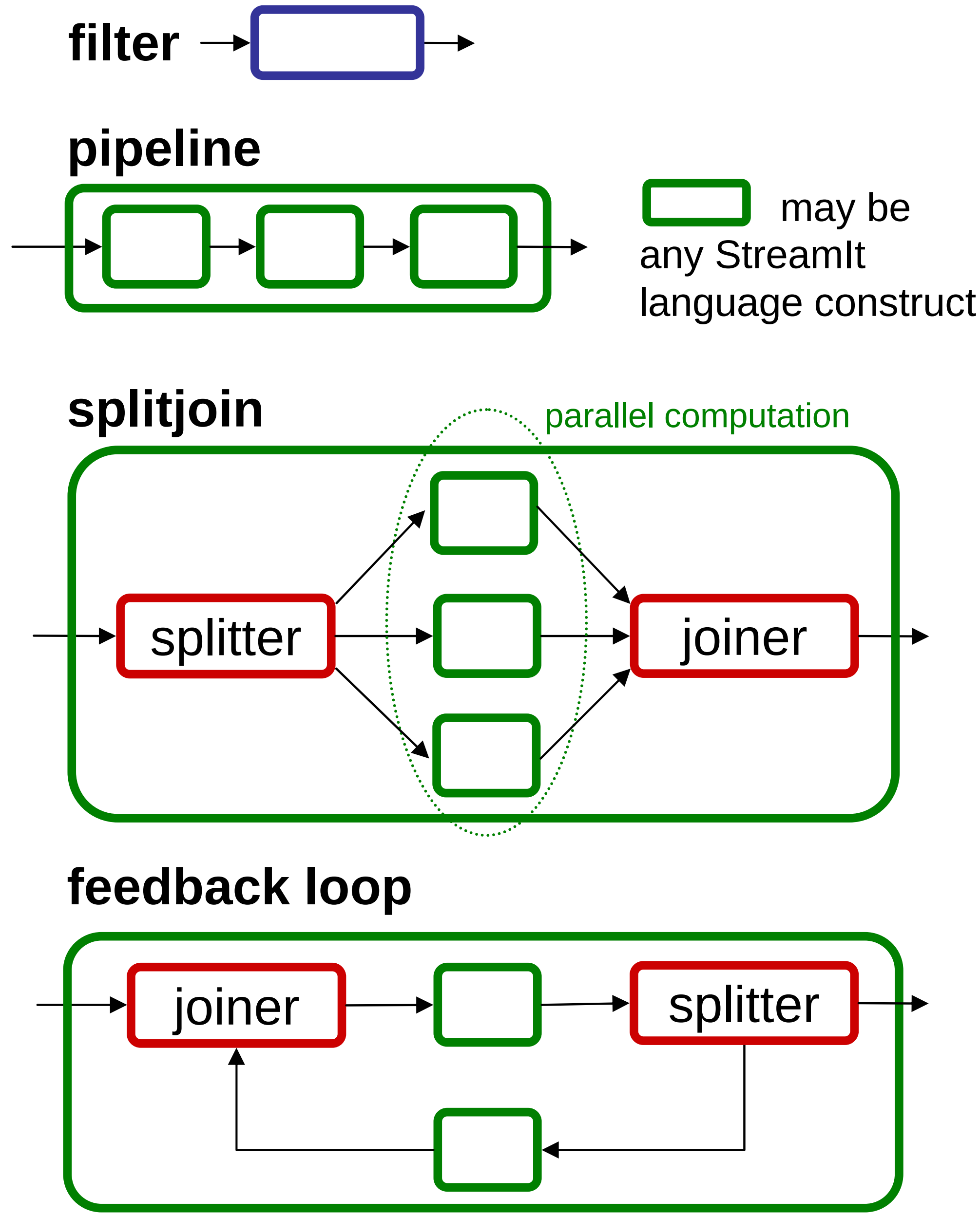



```
float→float filter FIR  (int N) {
                    ;
    work push 1 pop 1 peek N {
        float result = 0;
        weights = adaptChannel(weights);
        for (int i = 0; i < N; i++) {
            result += weights[i] ∗ peek(i);
        }
        pop();
        push(result);
    }
}
```

Stateful

StreamIt Language Overview

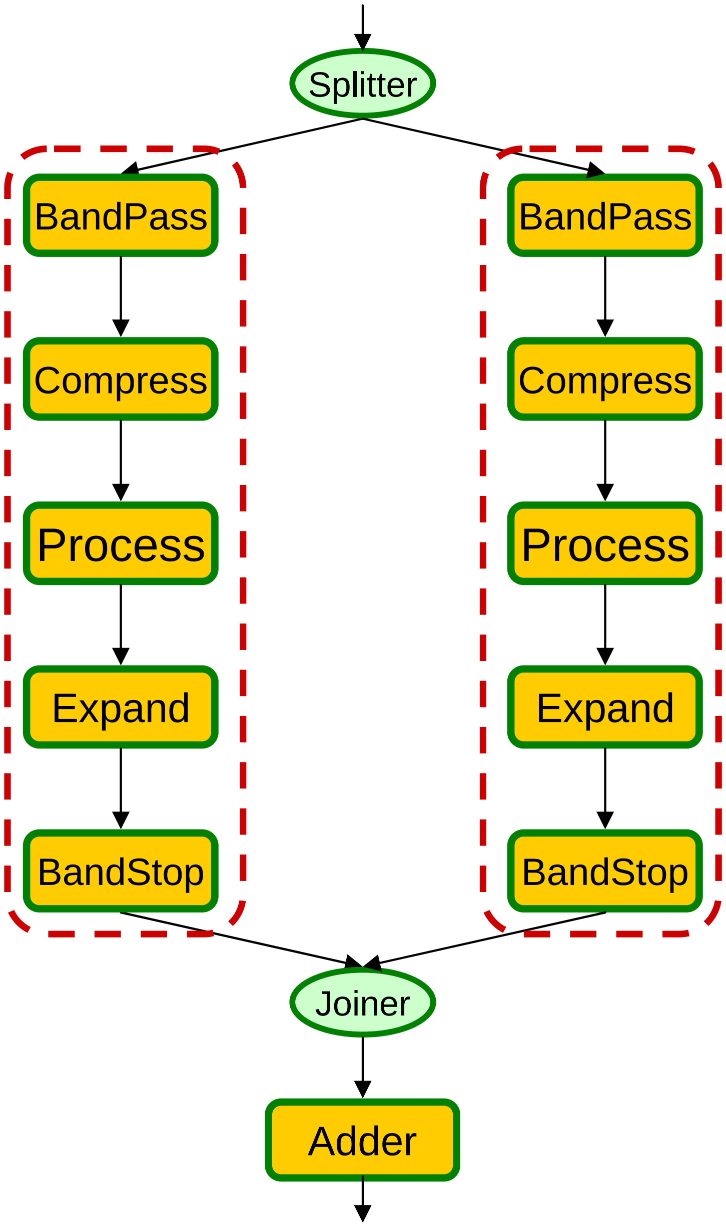
- StreamIt is a novel language for streaming
  - Exposes parallelism and communication
  - Architecture independent
  - Modular and composable
    - Simple structures composed to creates complex graphs
  - Malleable
    - Change program behavior with small modifications



Outline

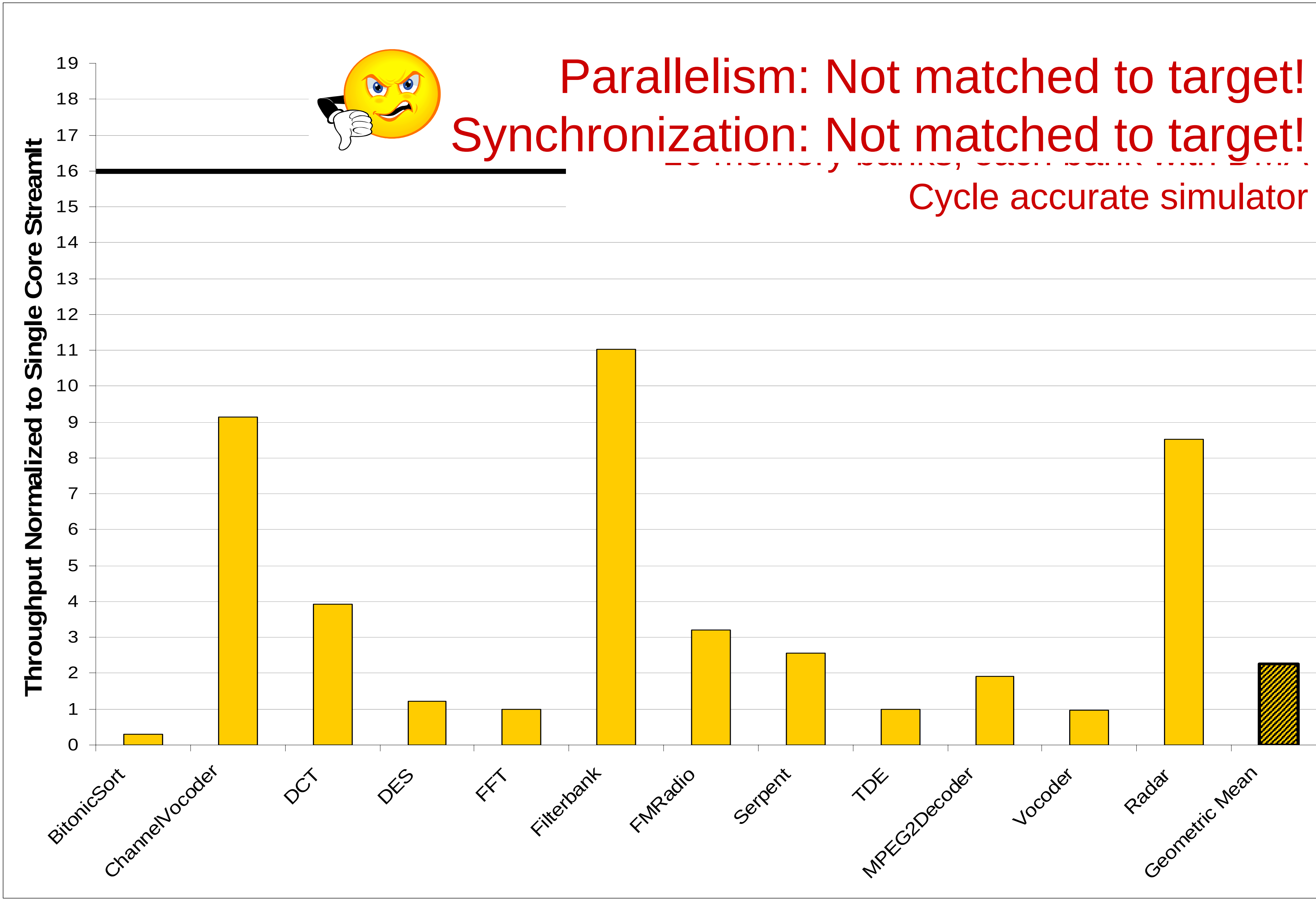
- StreamIt language overview
- Mapping to multicores
  - Baseline techniques
  - Our 3-phase solution

Baseline 1: Task Parallelism

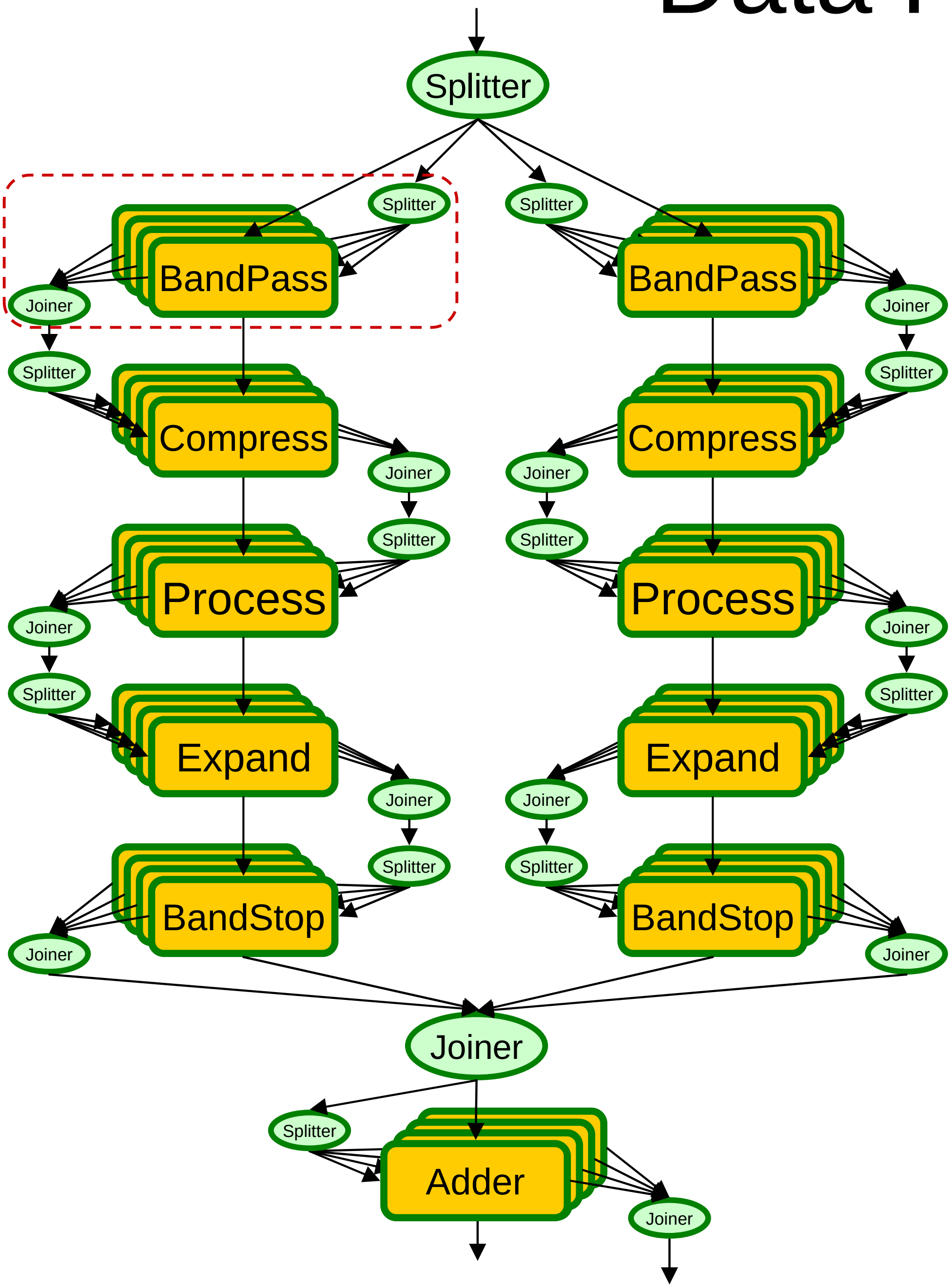


- Inherent task parallelism between two processing pipelines
- Task Parallel Model:
  - Only parallelize explicit task parallelism
  - Fork/join parallelism
- Execute this on a 2 core machine ~2x speedup over single core
- What about 4, 16, 1024, … cores?

Evaluation: Task Parallelism

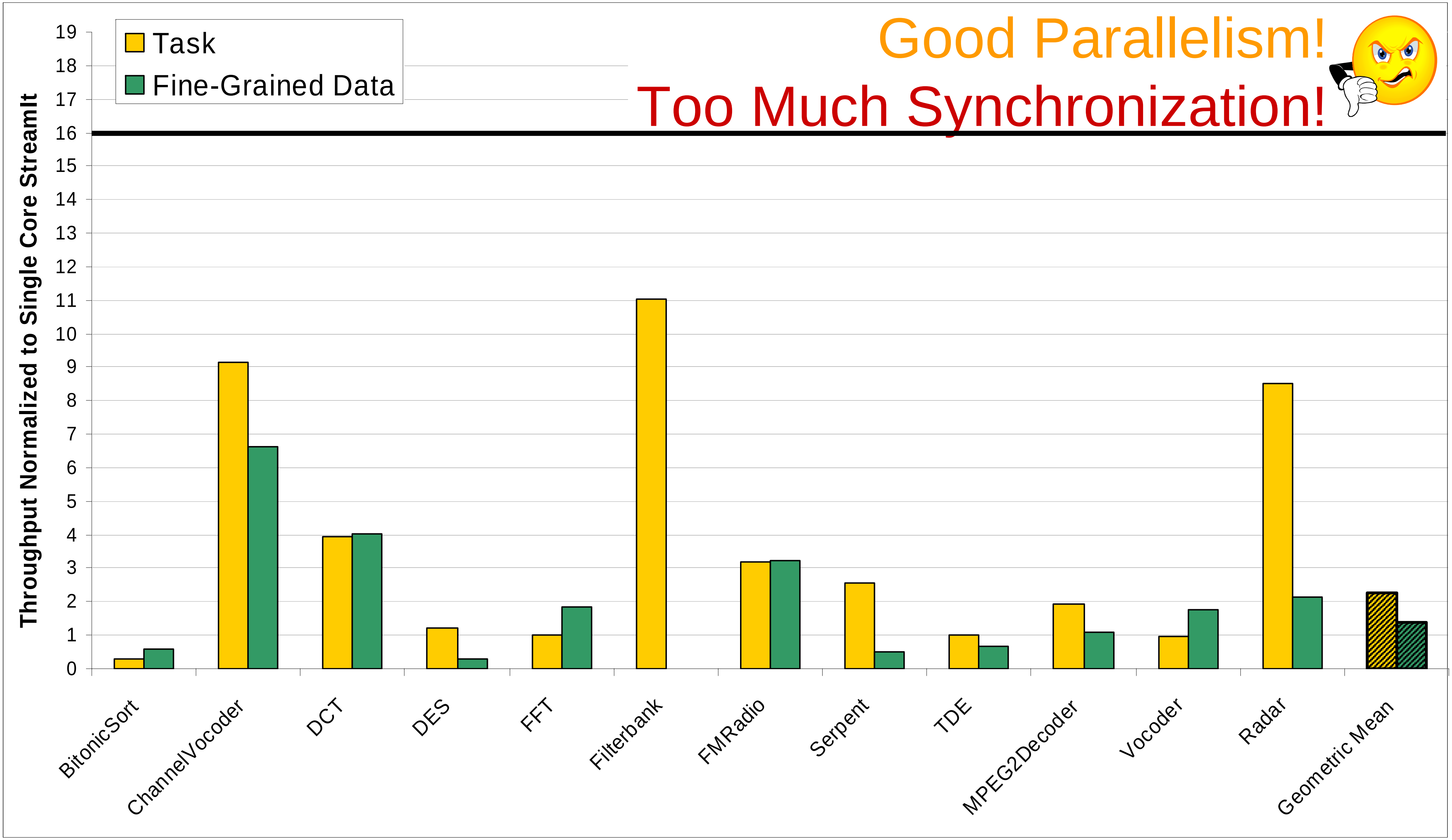


Baseline 2: Fine-Grained Data Parallelism



- 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

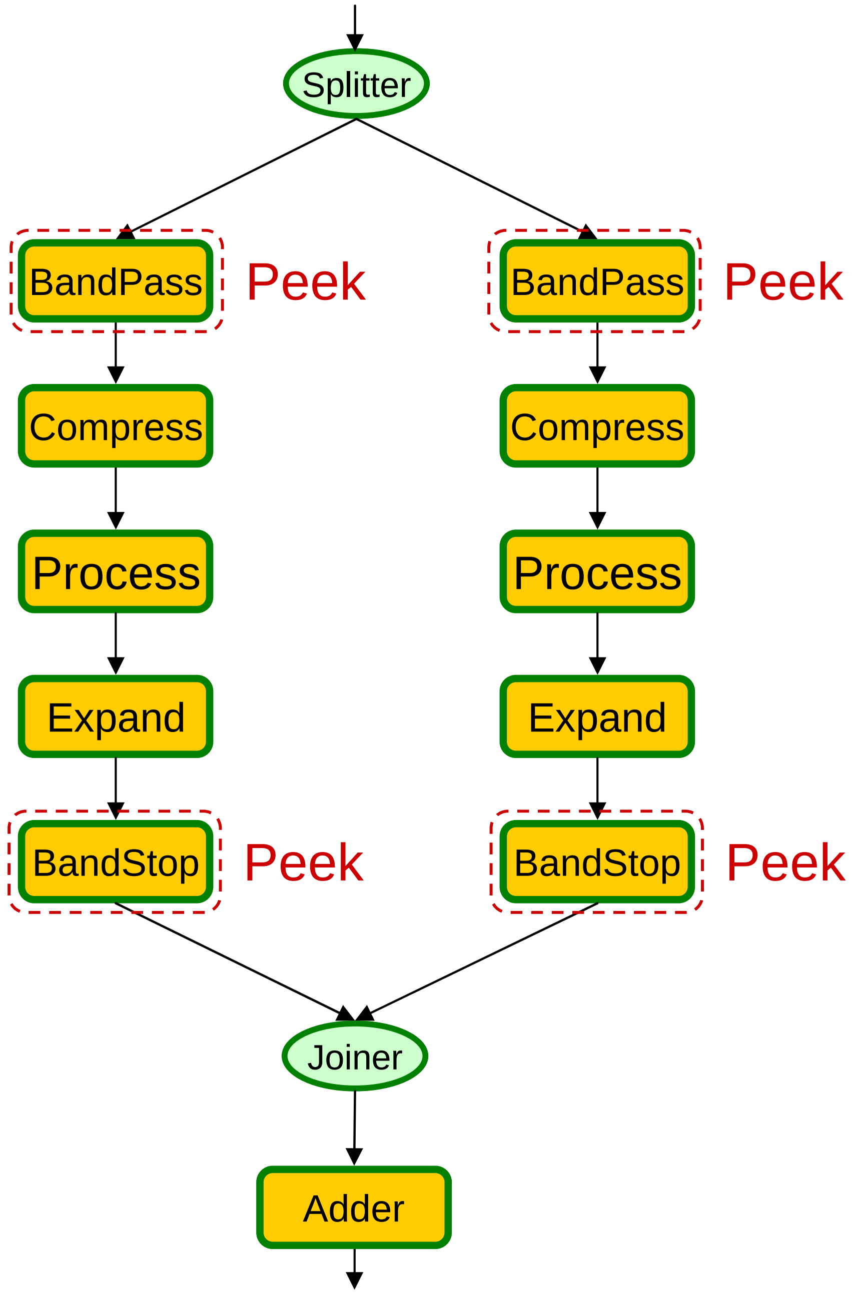
Evaluation: Fine-Grained Data Parallelism



Outline

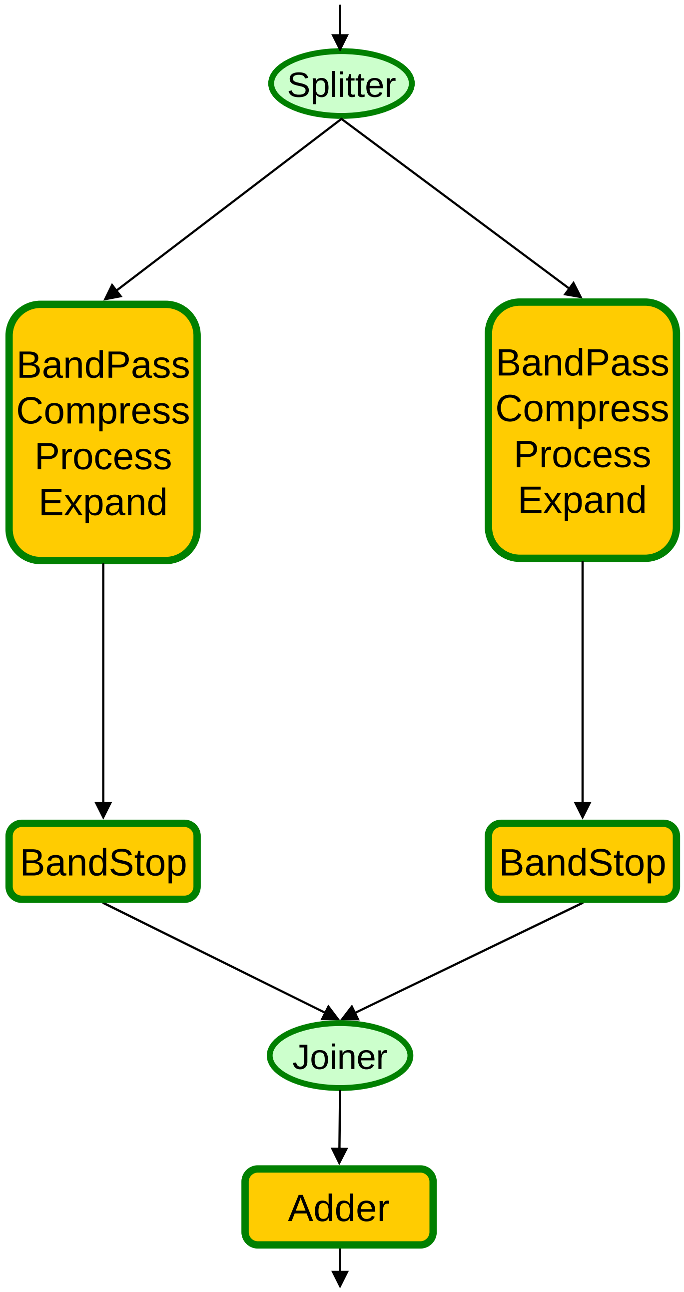
- StreamIt language overview
- Mapping to multicores
  - Baseline techniques
  - Our 3-phase solution

Phase 1: Coarsen the Stream Graph



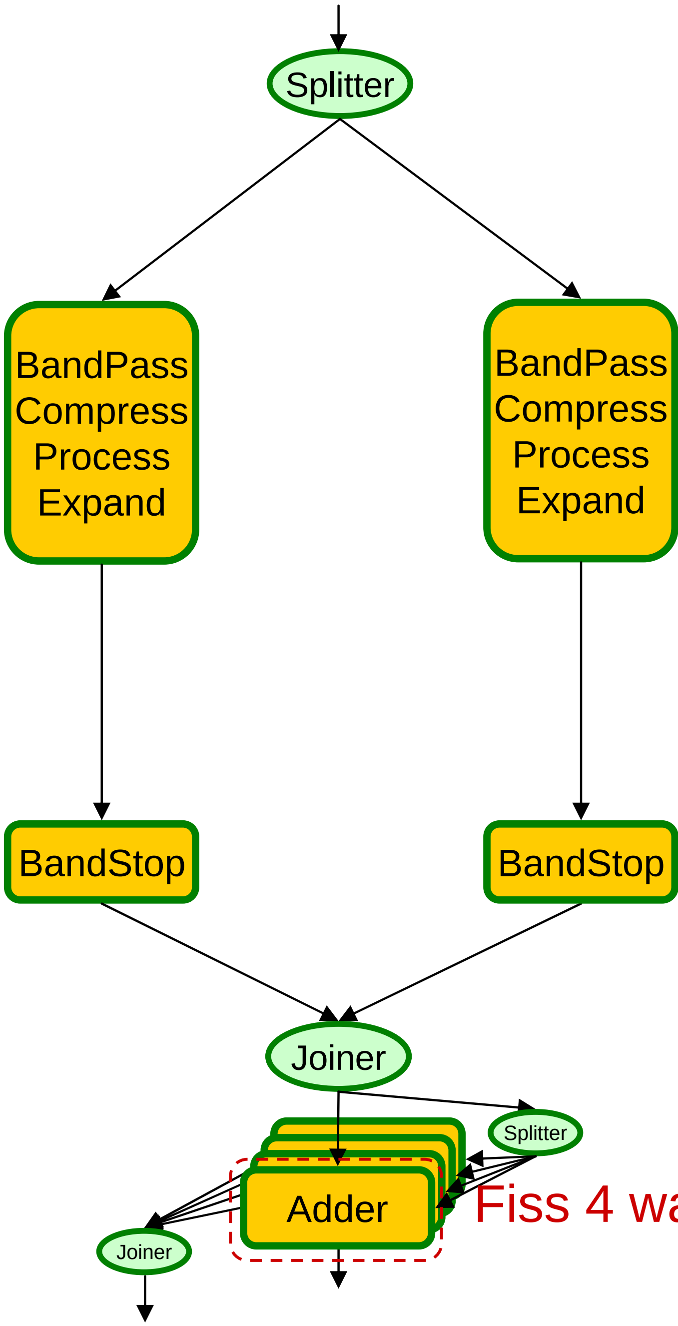
- Before data-parallelism is exploited
- *Fuse* stateless pipelines as much as possible without introducing state
  - Don’t fuse stateless with stateful
  - Don’t fuse a peeking filter with anything upstream

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

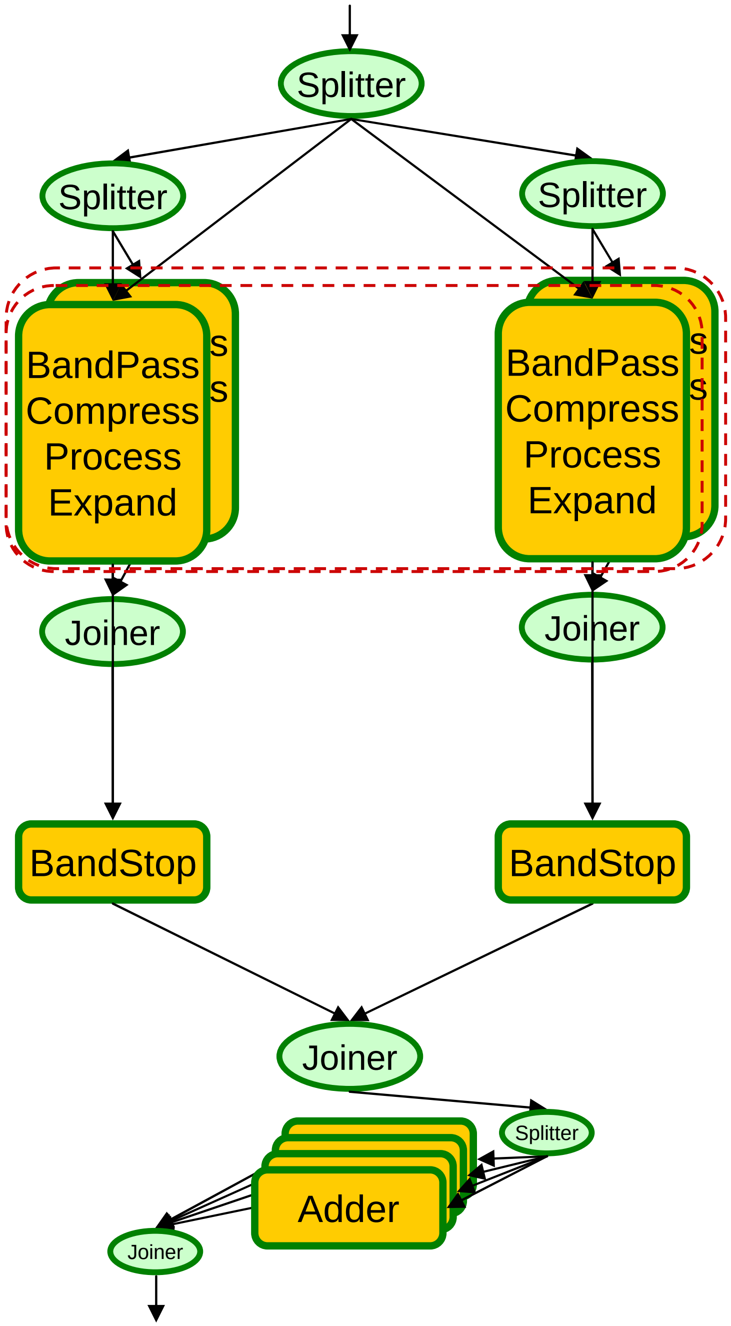
Phase 2: Data Parallelize



Data Parallelize for 4 cores

Fiss 4 ways, to occupy entire chip

Phase 2: Data Parallelize



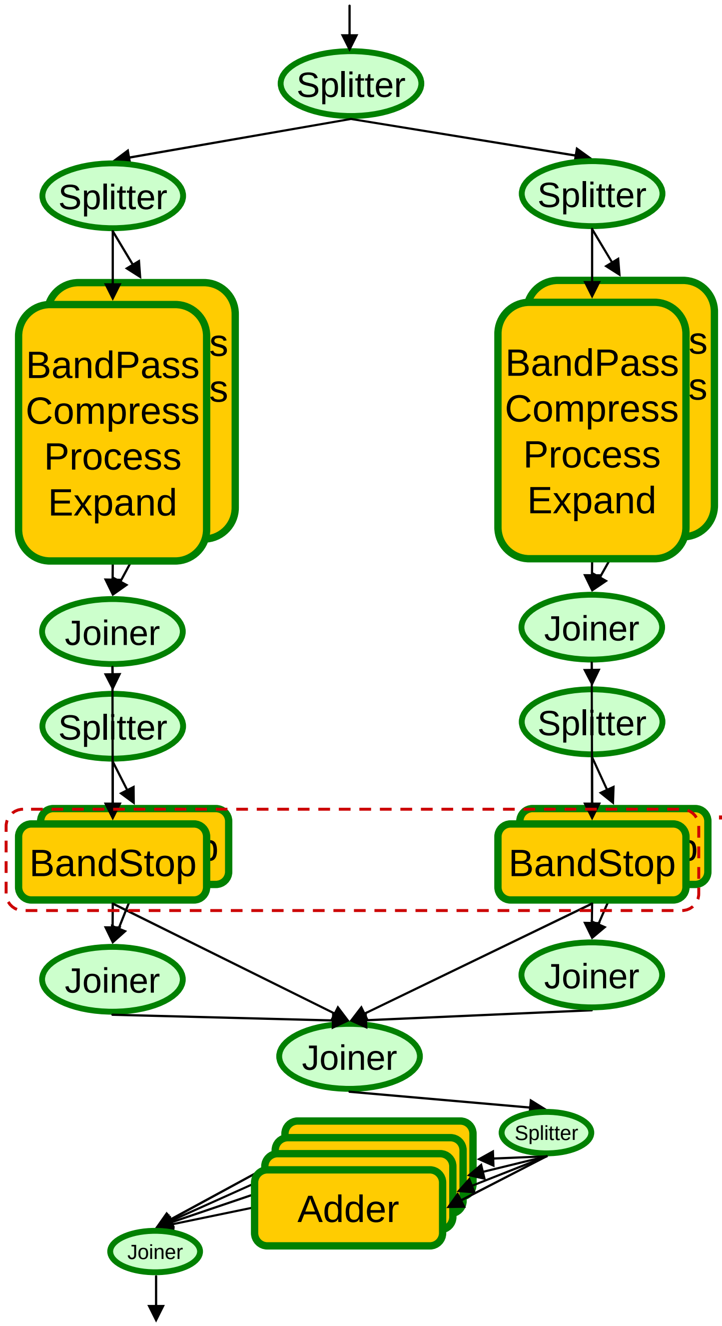
Data Parallelize for 4 cores

Task parallelism!

Each fused filter does equal work

Fiss each filter 2 times to occupy entire chip

Phase 2: Data Parallelize

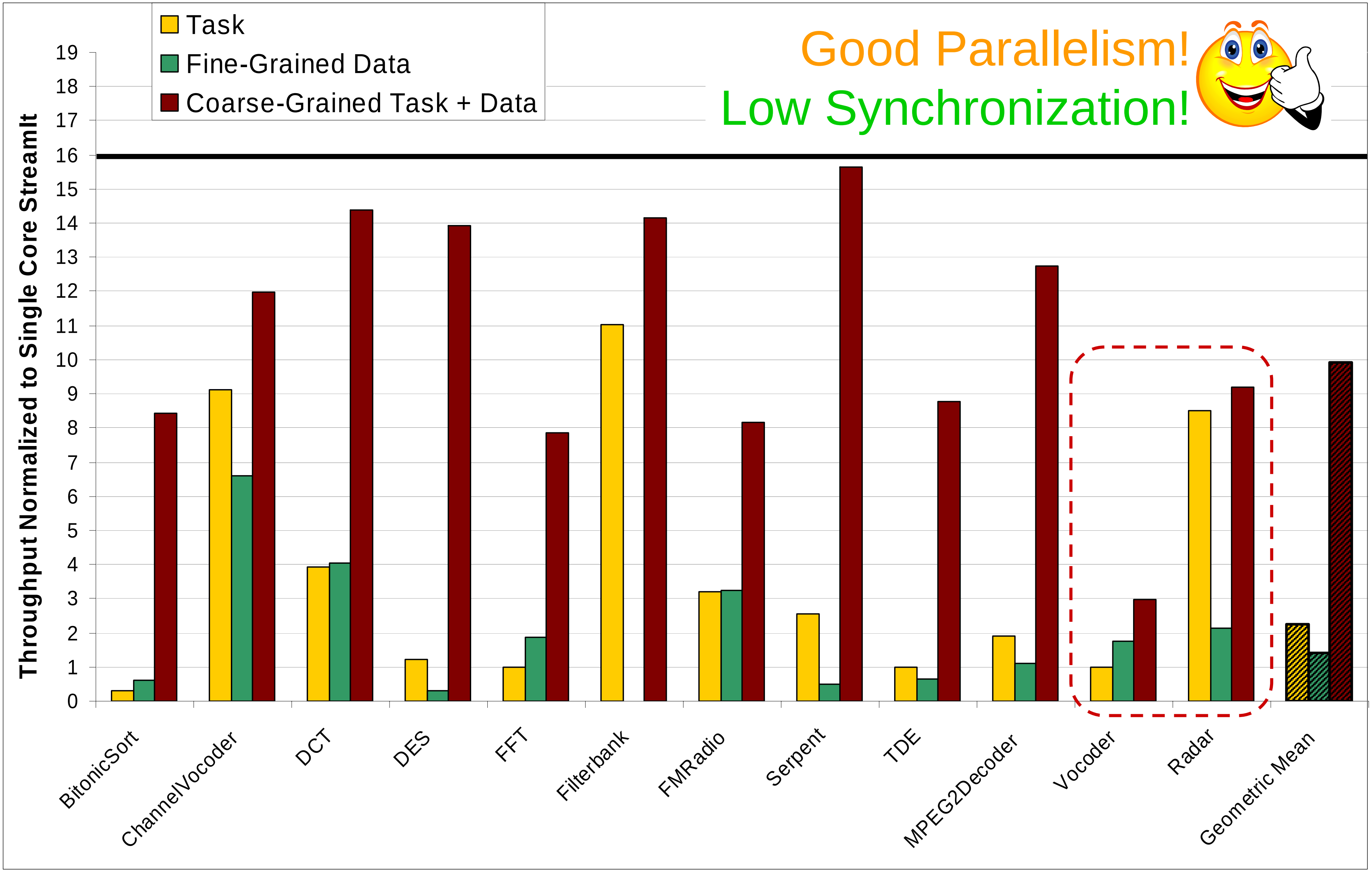


Data Parallelize for 4 cores

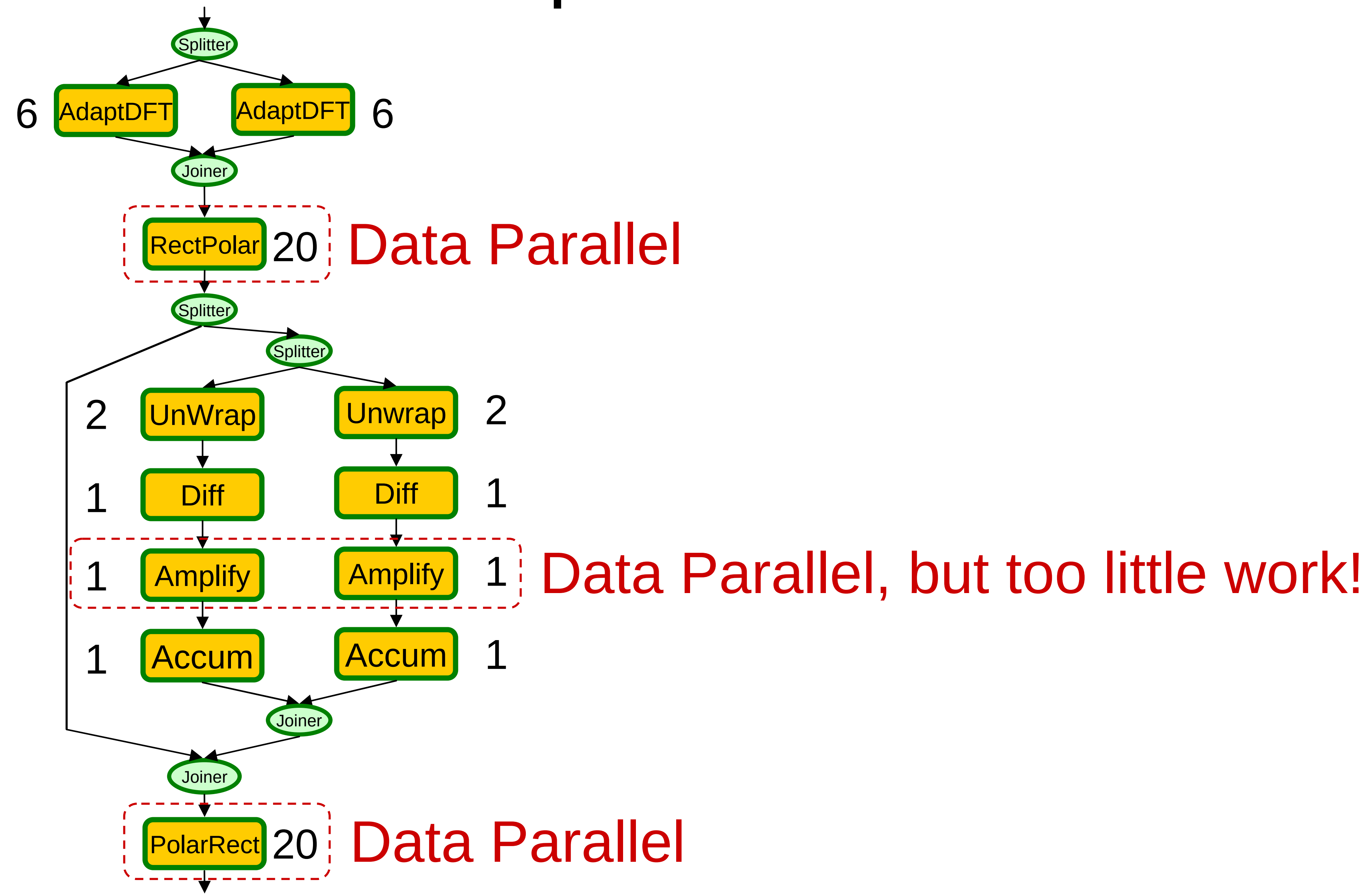
- Task-conscious data parallelization
  - Preserve task parallelism
- Benefits:
  - Reduces global communication and synchronization

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

Evaluation: Coarse-Grained Data Parallelism

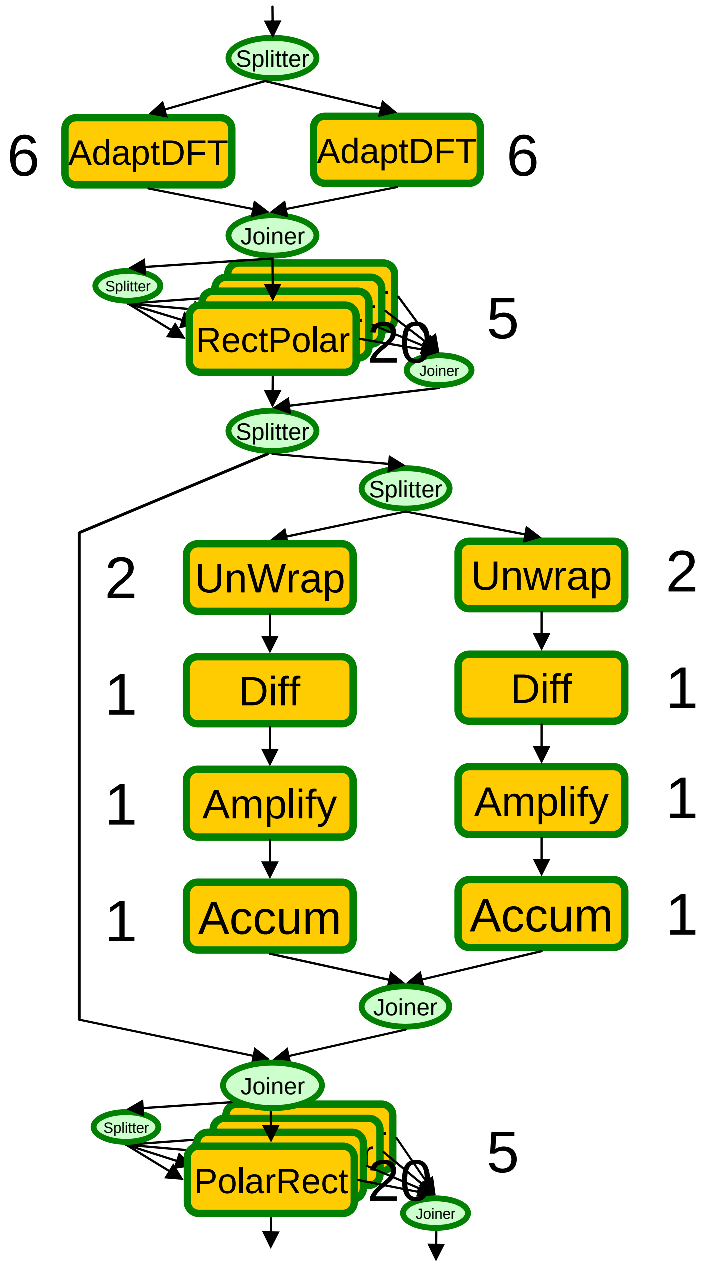


Simplified Vocoder



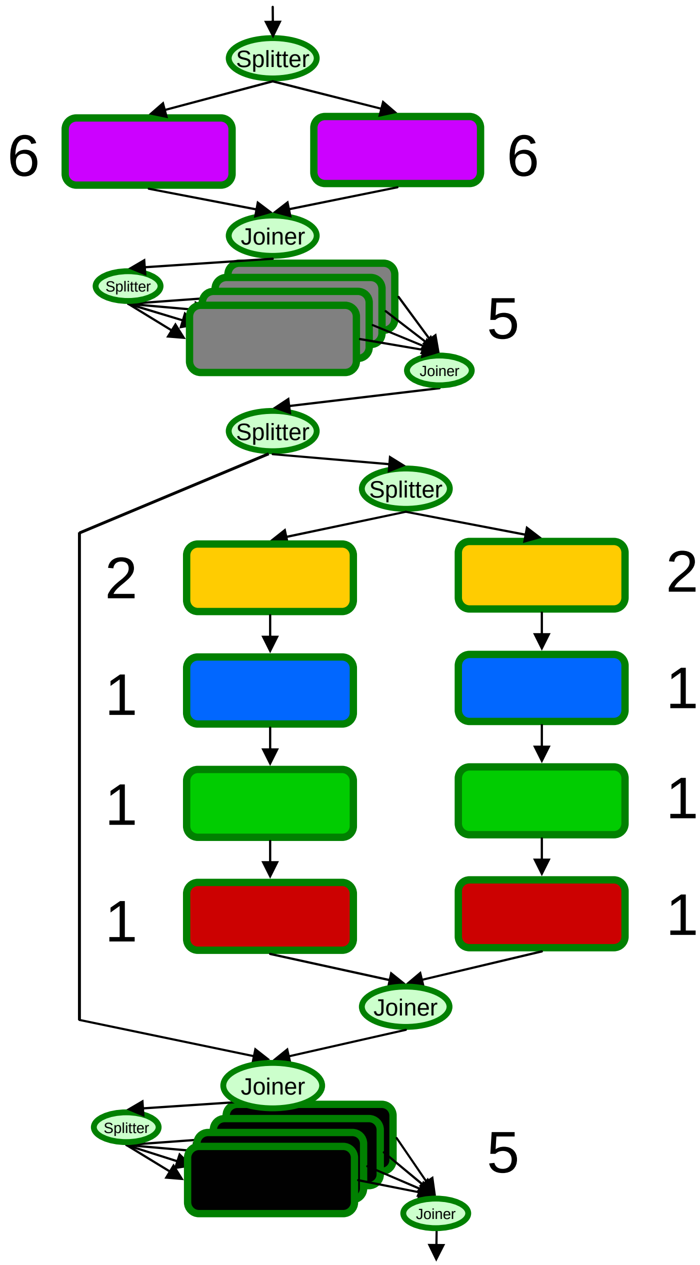
Target a 4 core machine

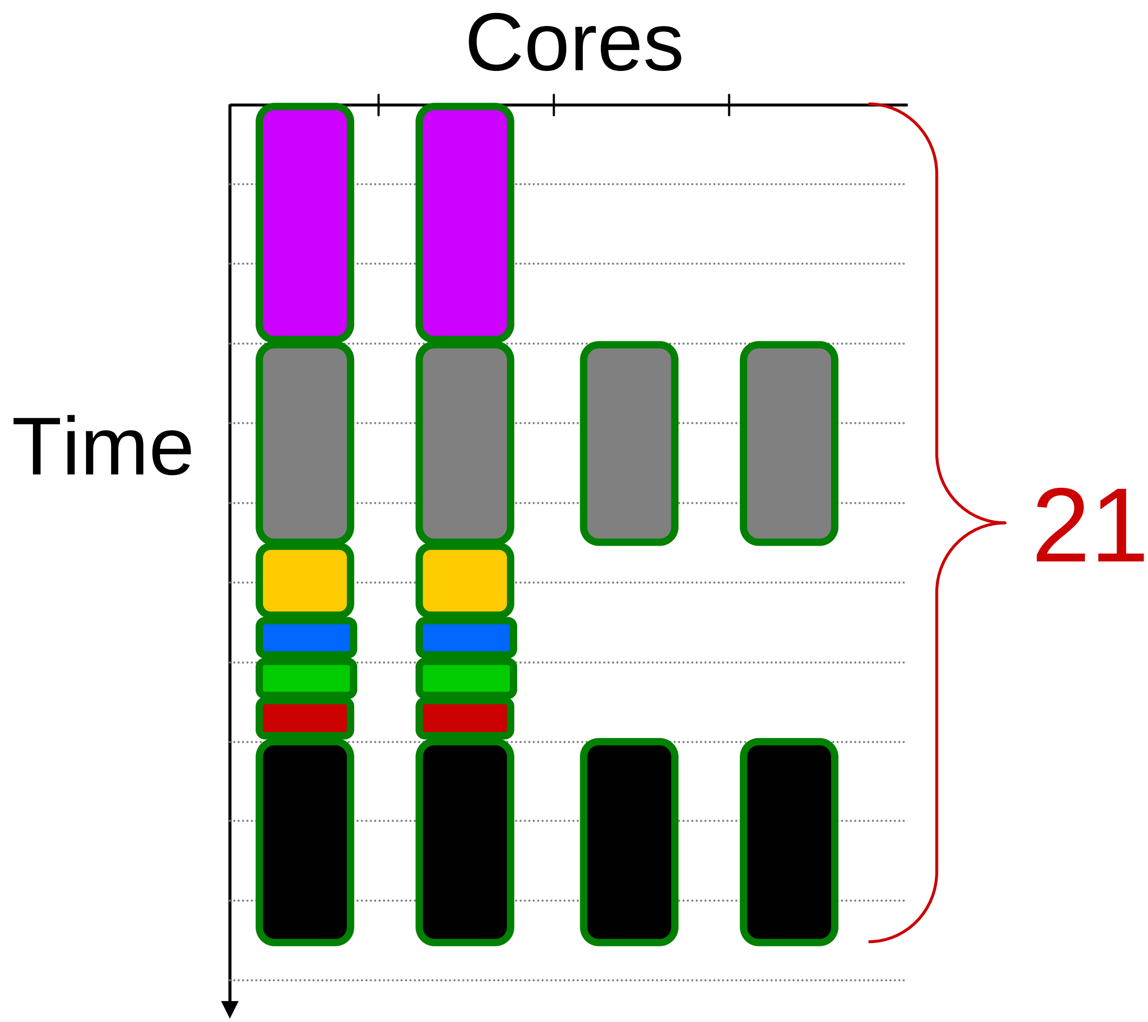
Data Parallelize



Target a 4 core machine

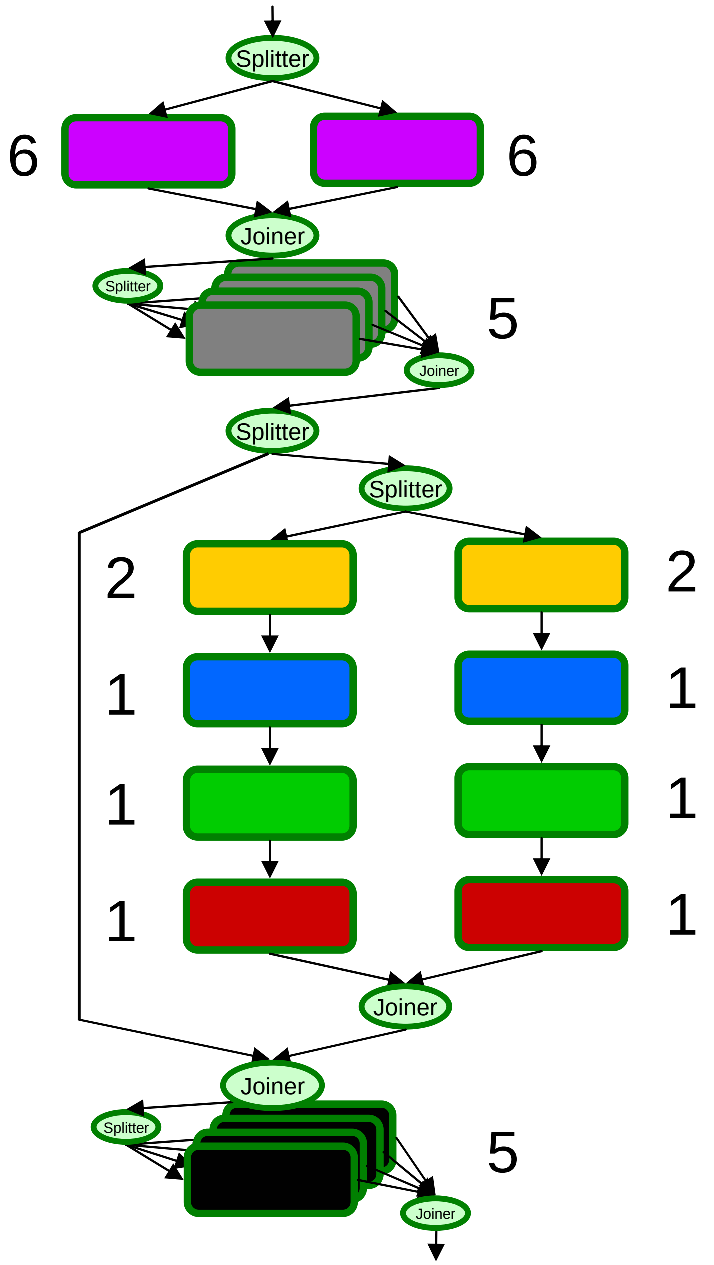
Data + Task Parallel Execution

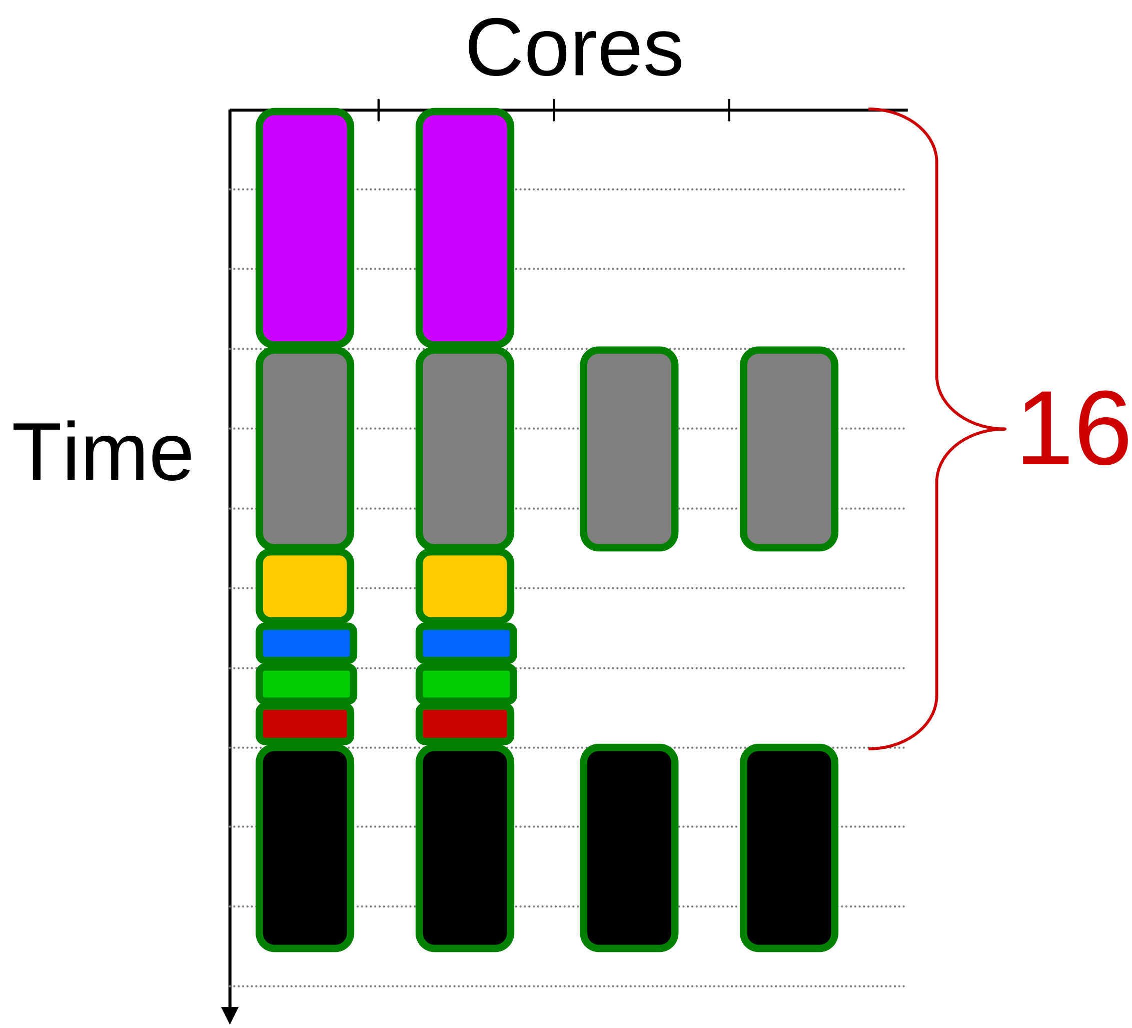




Target 4 core machine

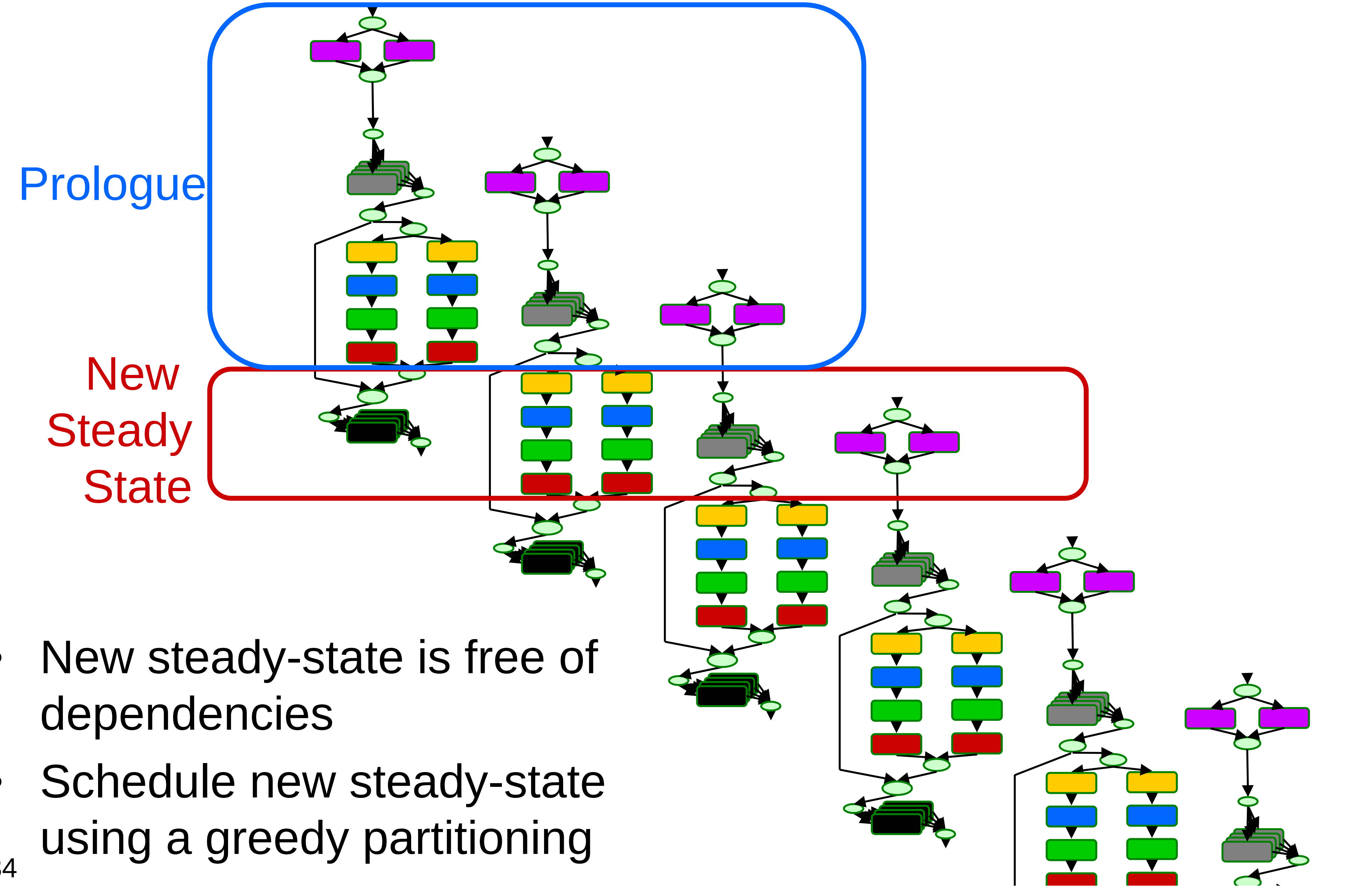
We Can Do Better!





Target 4 core machine

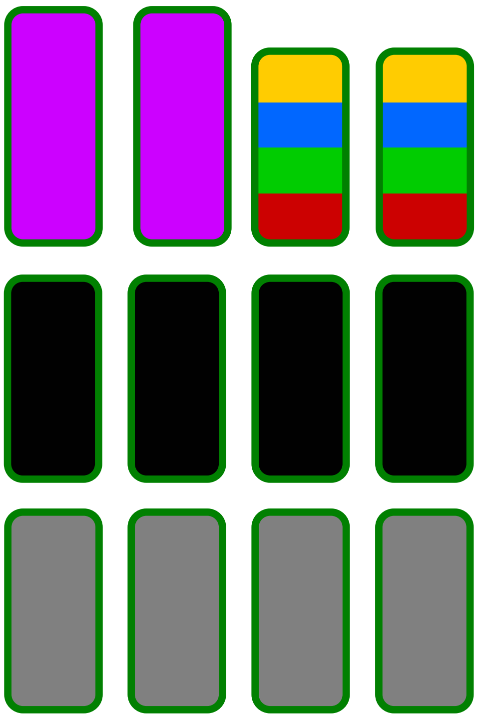
Phase 3: Coarse-Grained Software Pipelining



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

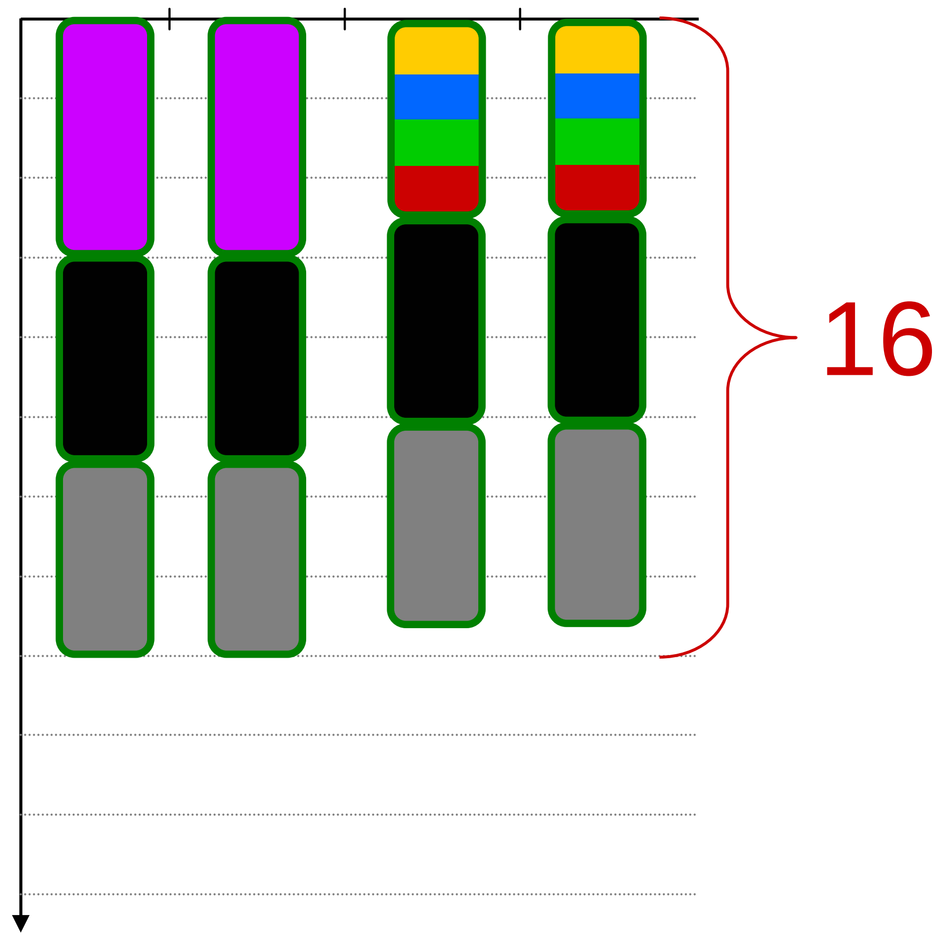
Greedy Partitioning

To Schedule:



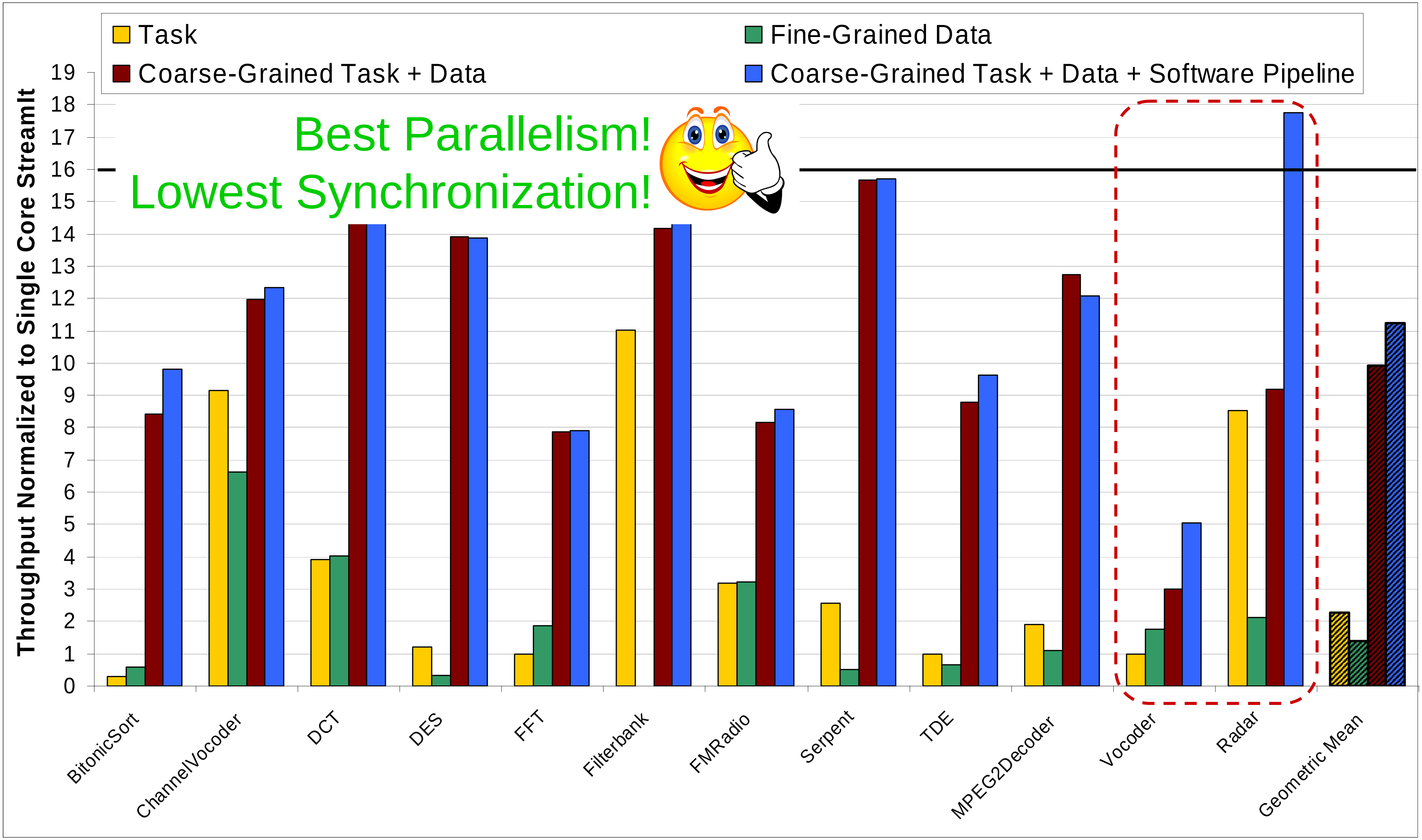
Cores

Time



Target 4 core machine

Evaluation: Coarse-Grained Task + Data + Software Pipelining



Generalizing to Other Multicores

- Architectural requirements:
  - Compiler controlled local memories with DMA
  - Efficient implementation of scatter/gather

- To port to other architectures, consider:
  - Local memory capacities
  - Communication to computation tradeoff

- Did not use processor-to-processor communication on Raw

Related Work

- Streaming languages:
  – Brook [Buck et al. ’04]
  – StreamC/KernelC [Kapasi ’03, Das et al. ’06]
  – Cg [Mark et al. ‘03]
  – SPUR [Zhang et al. ‘05]
- Streaming for Multicores:
  – Brook [Liao et al., ’06]
- Ptolemy [Lee ’95]
- Explicit parallelism:
  – OpenMP, MPI, & HPF

Conclusions

- Streaming model naturally exposes task, data, and pipeline parallelism
- This parallelism must be exploited at the correct granularity and combined correctly

	Task	Fine-Grained Data	Coarse-Grained Task + Data	Coarse-Grained Task + Data + Software Pipeline
Parallelism	**Not matched**	**Good**	**Good**	**Best**
Synchronization	**Not matched**	**High**	**Low**	**Lowest**

- Good speedups across varied benchmark suite
- Algorithms should be applicable across multicores