The Looming Software Crisis due to the Multicore Menace

Saman Amarasinghe

Computer Science and Artificial Intelligence Laboratory

Massachusetts Institute of Technology

Today: The Happily Oblivious Average Joe Programmer

- Joe is oblivious about the processor
  - Moore's law bring Joe performance
  - Sufficient for Joe's requirements
- Joe has built a solid boundary between Hardware and Software
  - High level languages abstract away the processors
    - Ex: Java bytecode is machine independent
- This abstraction has provided a lot of freedom for Joe

- Parallel Programming is only practiced by a few experts

Joe the Parallel Programmer

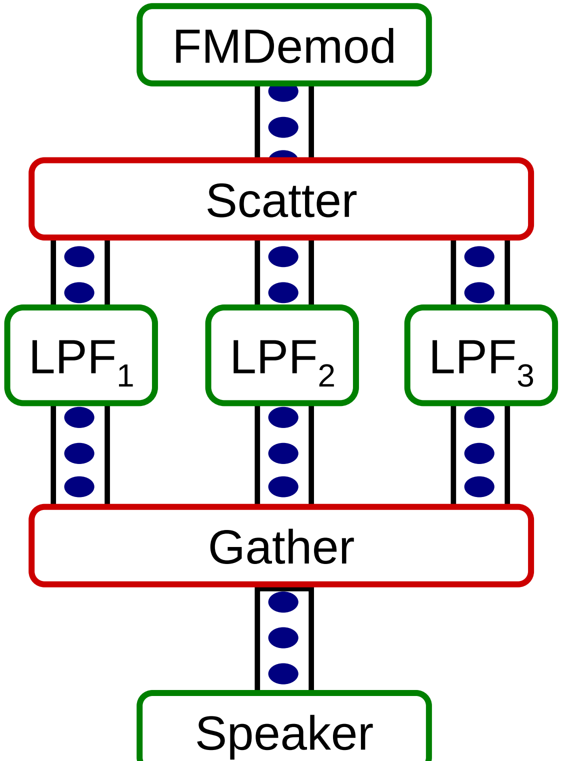
- Moore’s law is not bringing anymore performance gains
- If Joe needs performance he has to deal with multicores
  - Joe has to deal with performance
  - Joe has to deal with parallelism
- Is there a better way?

Why Parallelism is Hard

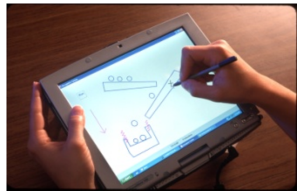
- A huge increase in complexity and work for the programmer
  - Programmer has to think about performance!
  - Parallelism has to be designed in at every level
- Humans are sequential beings
  - Deconstructing problems into parallel tasks is hard for many of us
- Parallelism is not easy to implement
  - Parallelism cannot be abstracted or layered away
  - Code and data has to be restructured in very different (non-intuitive) ways
- Parallel programs are very hard to debug
  - Combinatorial explosion of possible execution orderings
  - Race condition and deadlock bugs are non-deterministic and illusive
  - Non-deterministic bugs go away in lab environment and with instrumentation

Compiler-Aware Language Design

The StreamIt Experience



Stream Application Domain







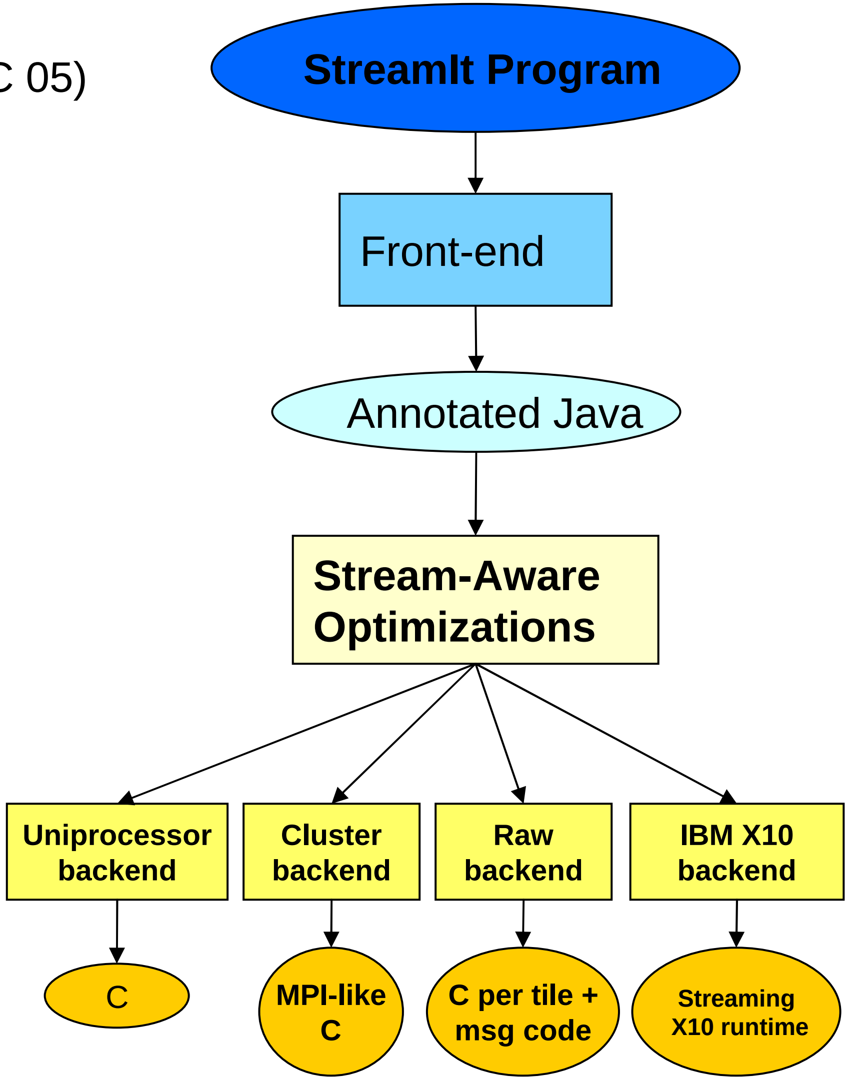





- Graphics
- Cryptography
- Databases
- Object recognition
- Network processing and security
- Scientific codes
- …

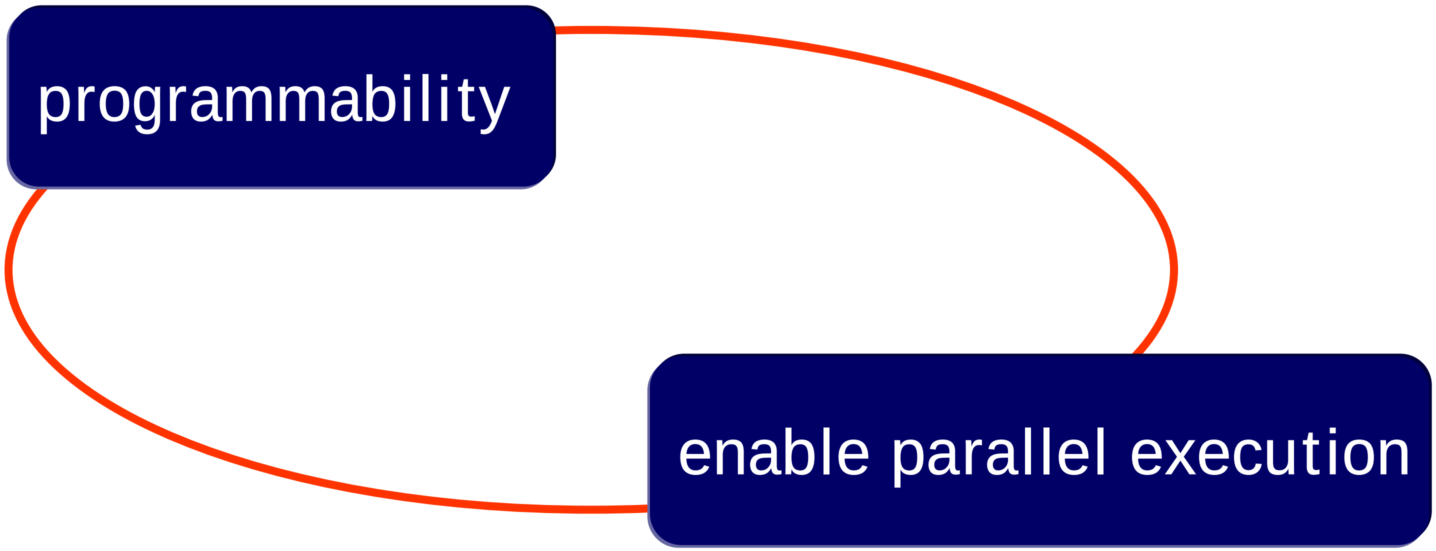
StreamIt Project

- **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], …



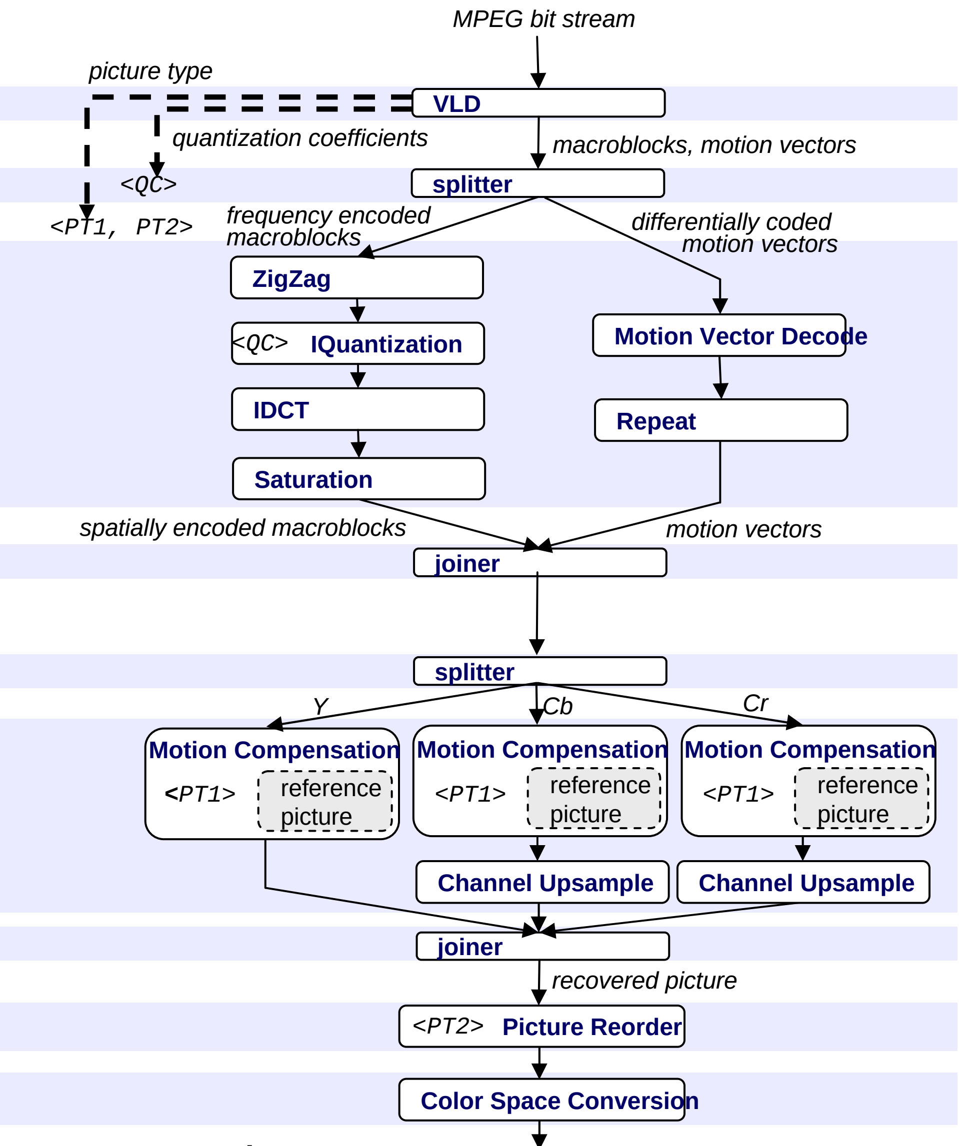
Compiler-Aware Language Design

boost productivity, enable faster development and rapid prototyping



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

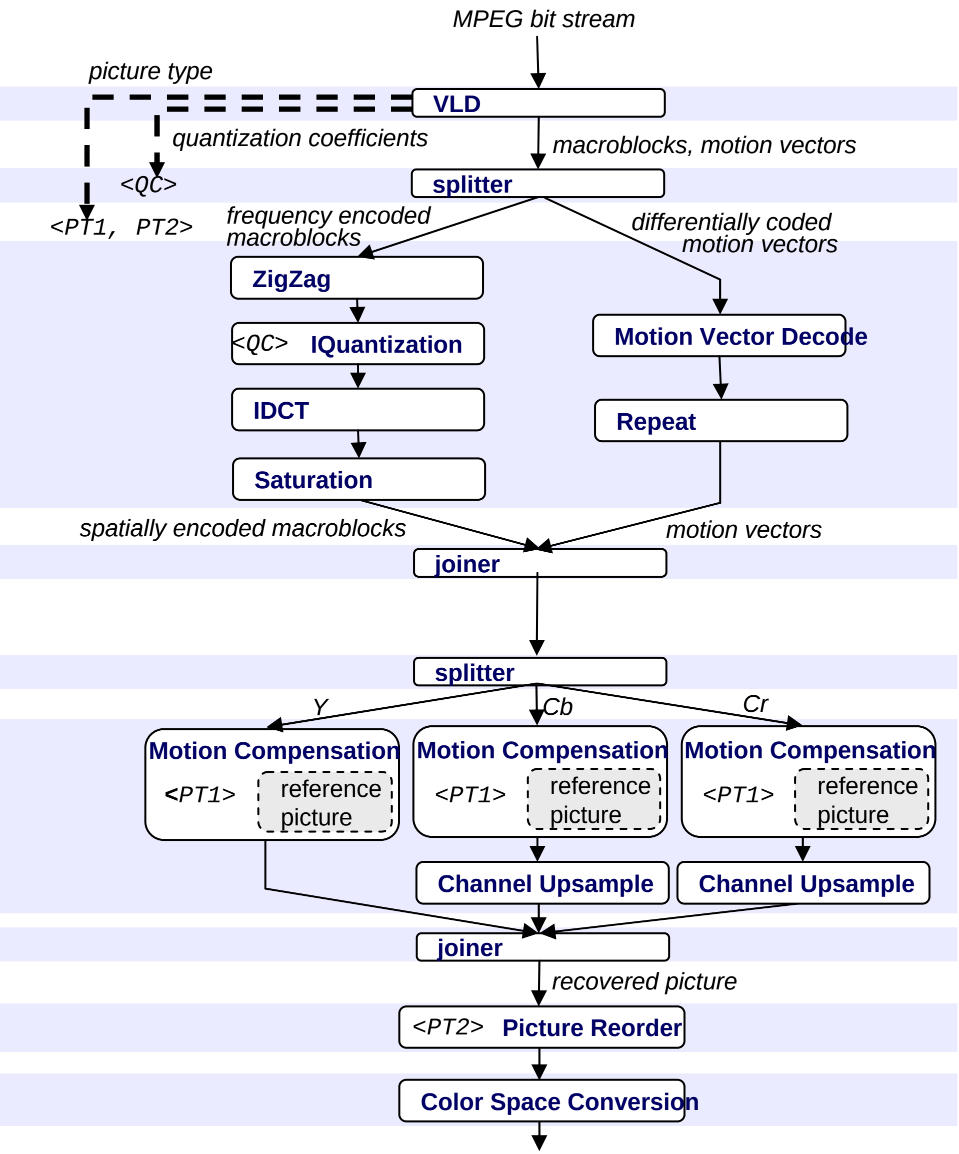
Streaming Application Design



MPEG-2 Decoder

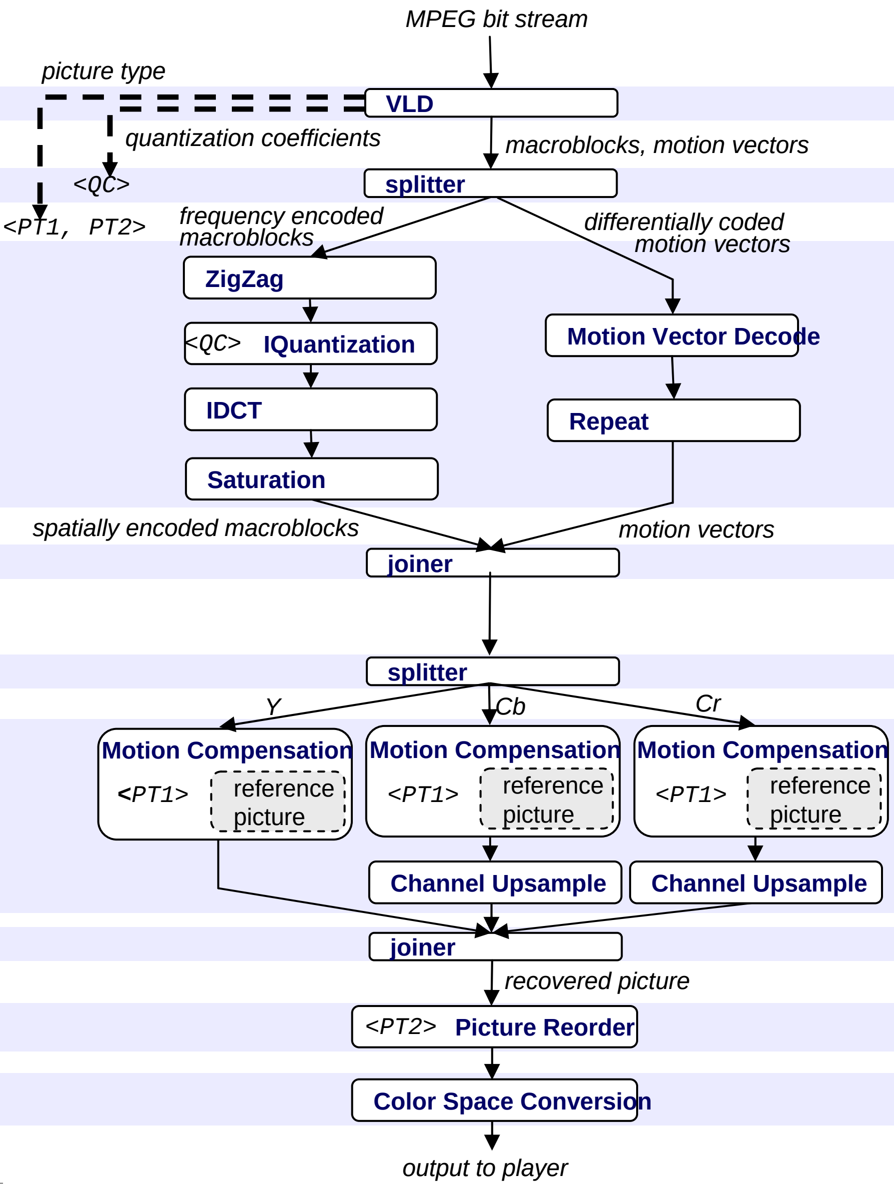
- Structured block level diagram describes computation and flow of data
- Conceptually easy to understand
  - Clean abstraction of functionality

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

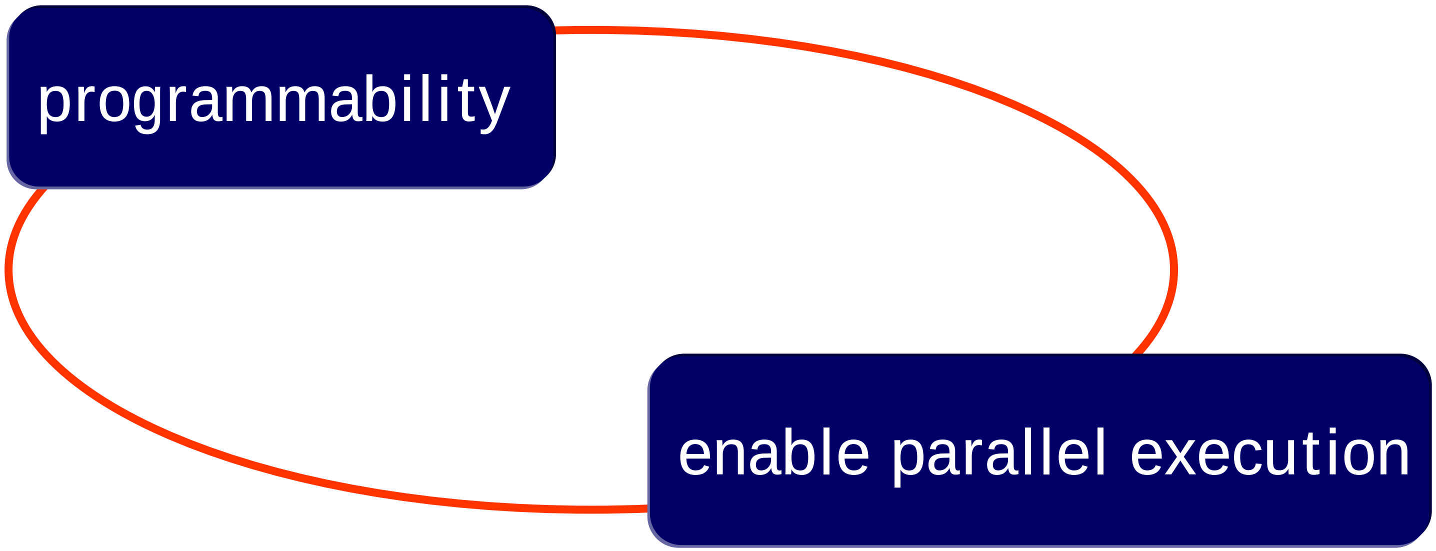
StreamIt Philosophy




```
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);
```

Compiler-Aware Language Design

boost productivity, enable faster development and rapid prototyping



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

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:

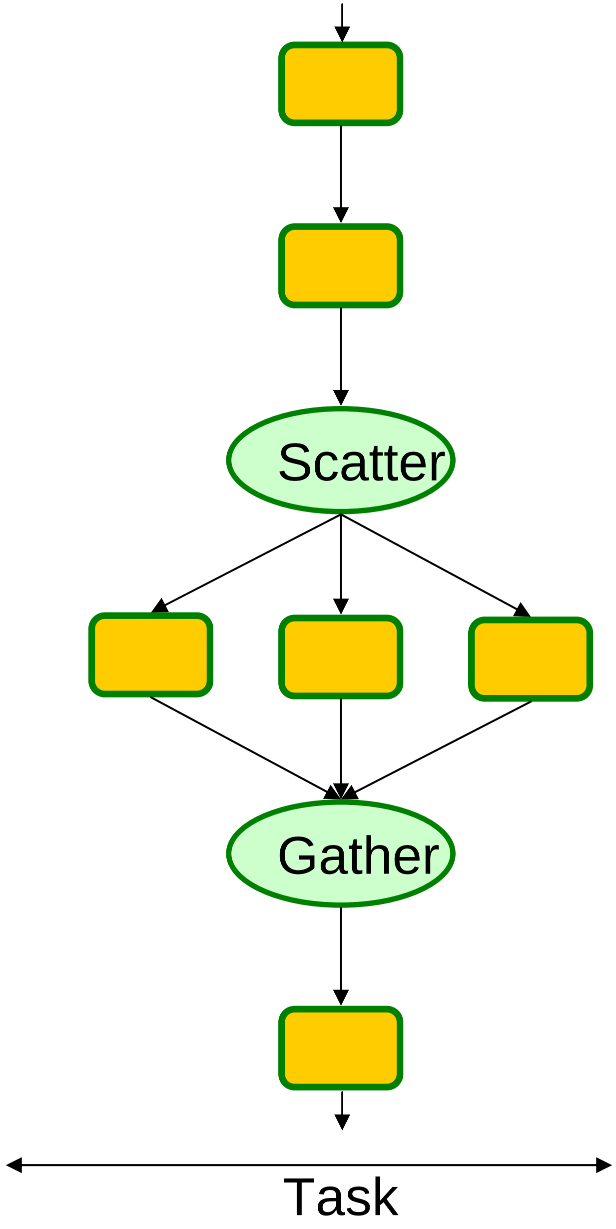
Common Properties
Multiple flows of control
Multiple local memories

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

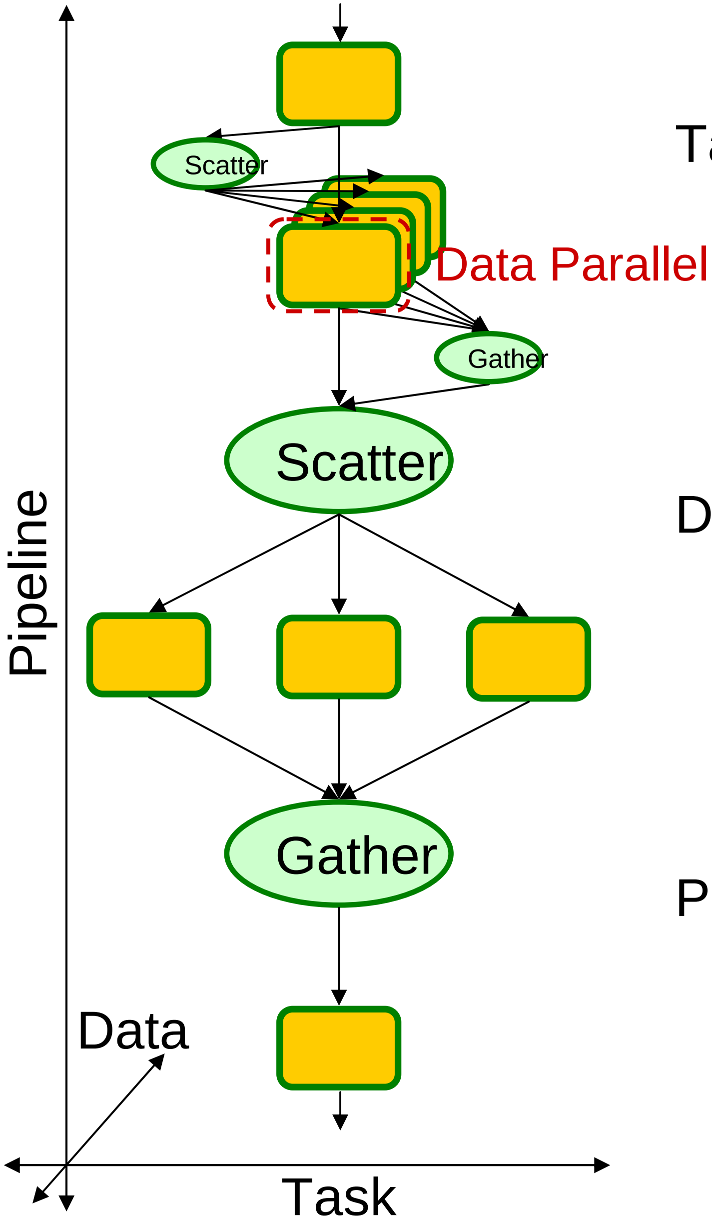
Types of Parallelism



Task Parallelism (traditionally thread fork/join)

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

Types of Parallelism



Task Parallelism (traditionally thread fork/join)

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

Data Parallelism (traditionally data parallel loops)

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

Pipeline Parallelism (traditionally in hardware)

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

Problem Statement

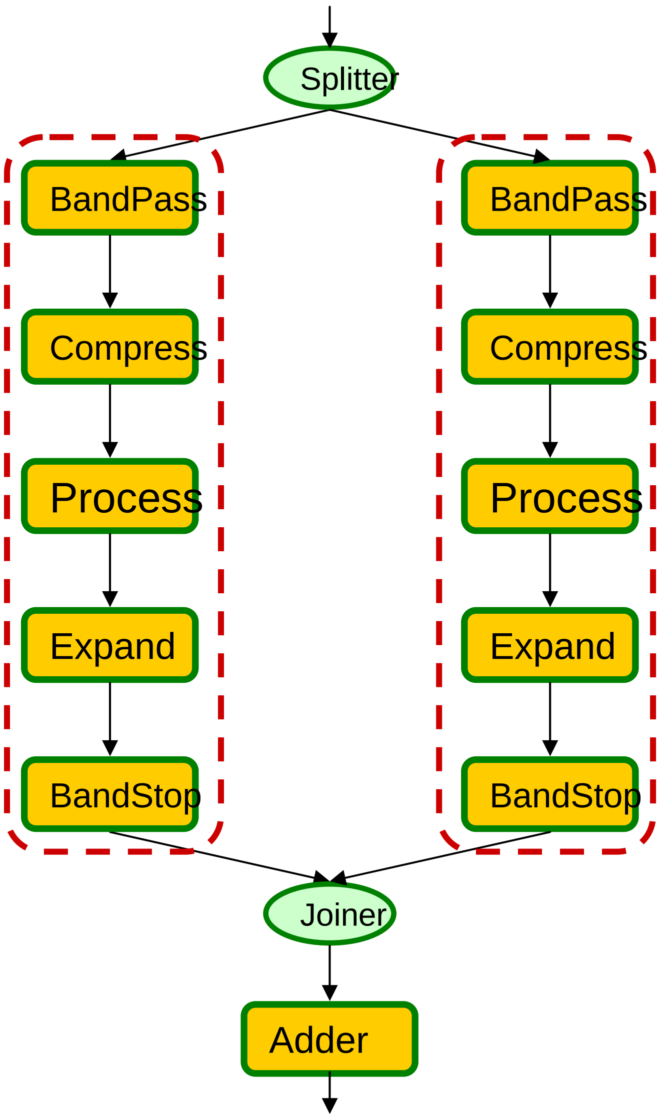
Given:

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

Find:

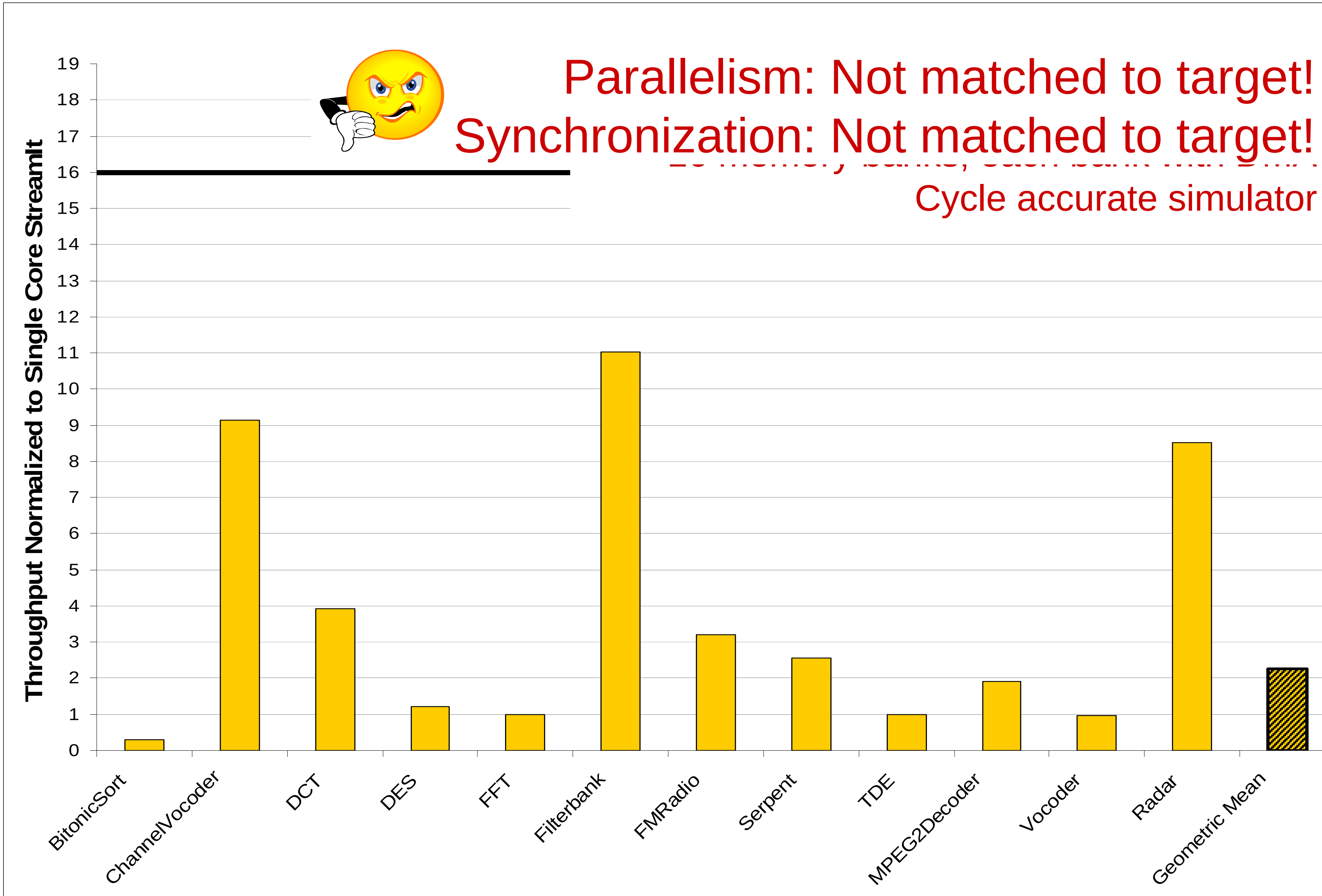
- Schedule of execution for the filters that best utilizes the available parallelism to fit the machine resources

Baseline 1: Task Parallelism

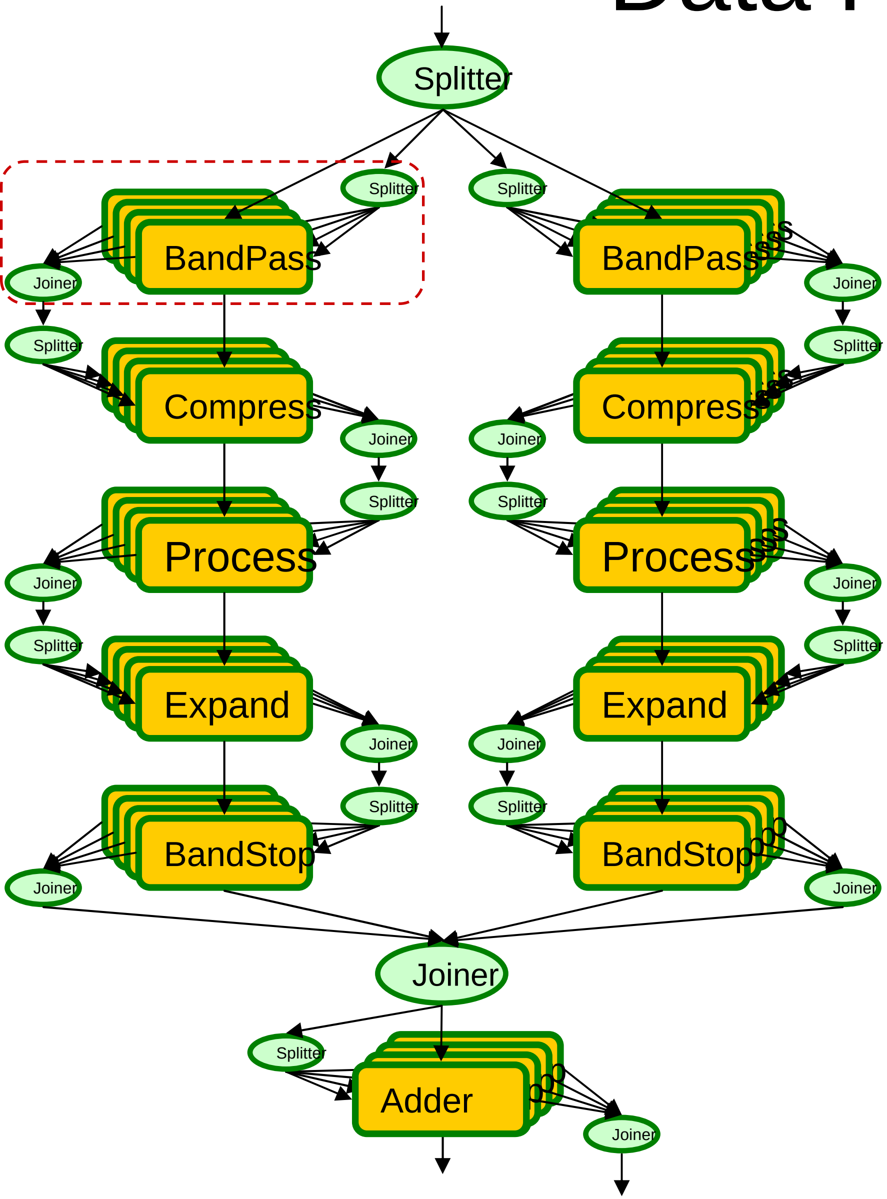


- Inherent task parallelism between two processing pipelines
- Task Parallel Model:
  - Only parallelize explicit task parallelism
  - Fork/join parallelism
- Execute this on a 2 core machine ~2x speedup over single core
- What about 4, 16, 1024, … cores?

Evaluation: Task Parallelism

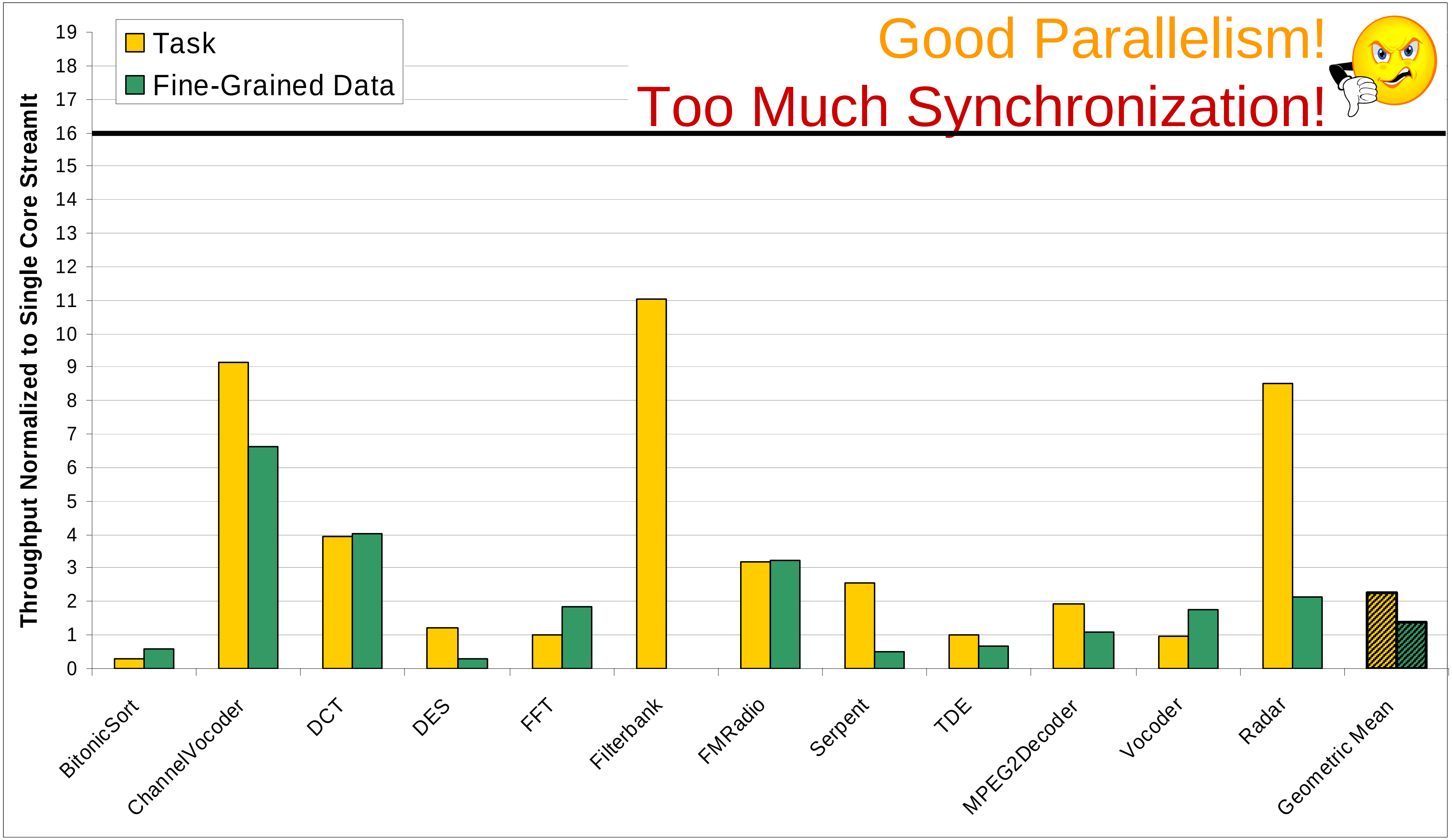


Baseline 2: Fine-Grained Data Parallelism

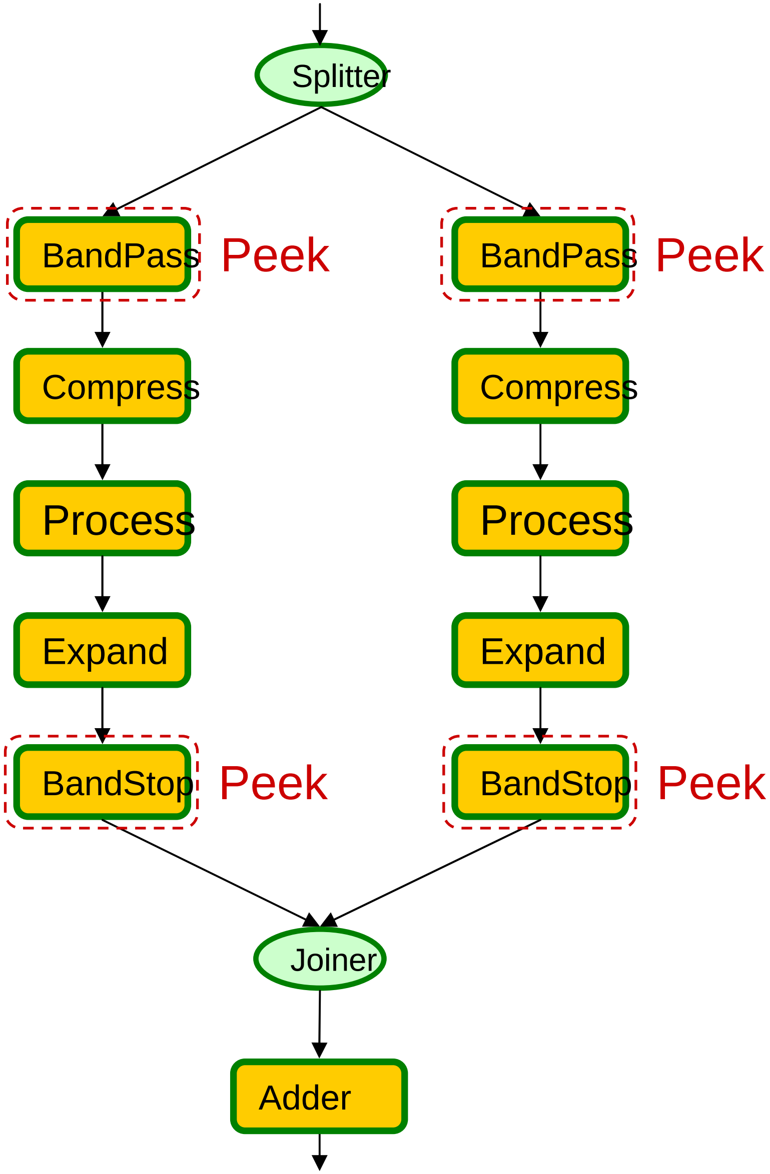


- Each of the filters in the example are stateless
- Fine-grained Data Parallel Model:
  - *Fiss* each stateless filter *N* ways (*N* is number of cores)
  - Remove scatter/gather if possible
- We can introduce data parallelism
  - Example: 4 cores
- Each fission group occupies entire machine

Evaluation: Fine-Grained Data Parallelism

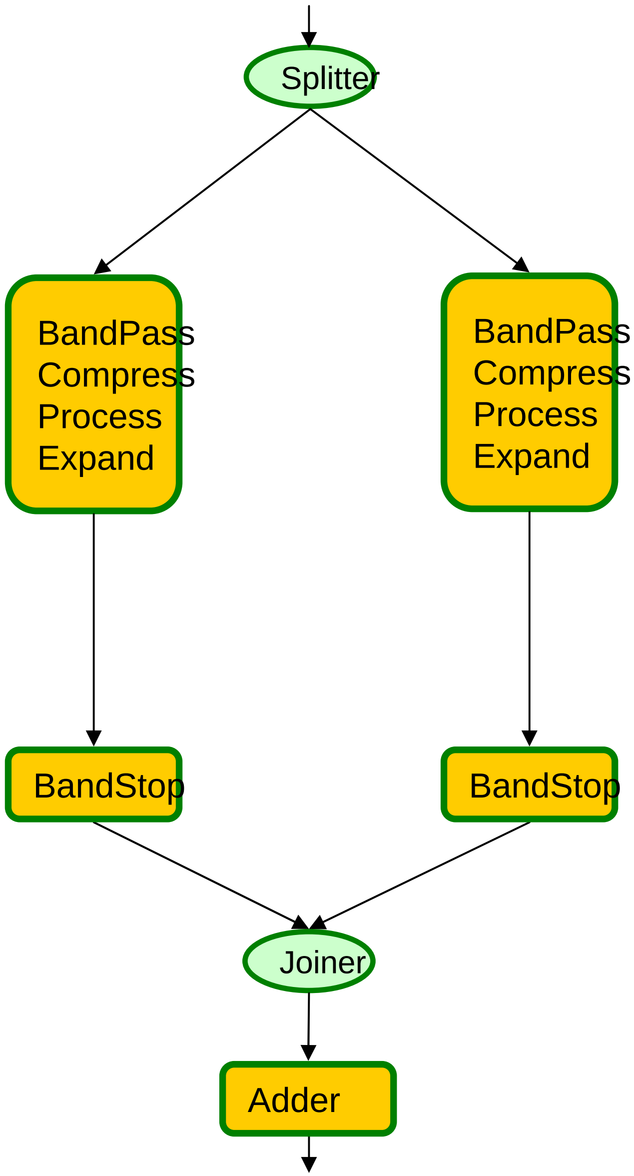


Phase 1: Coarsen the Stream Graph



- 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

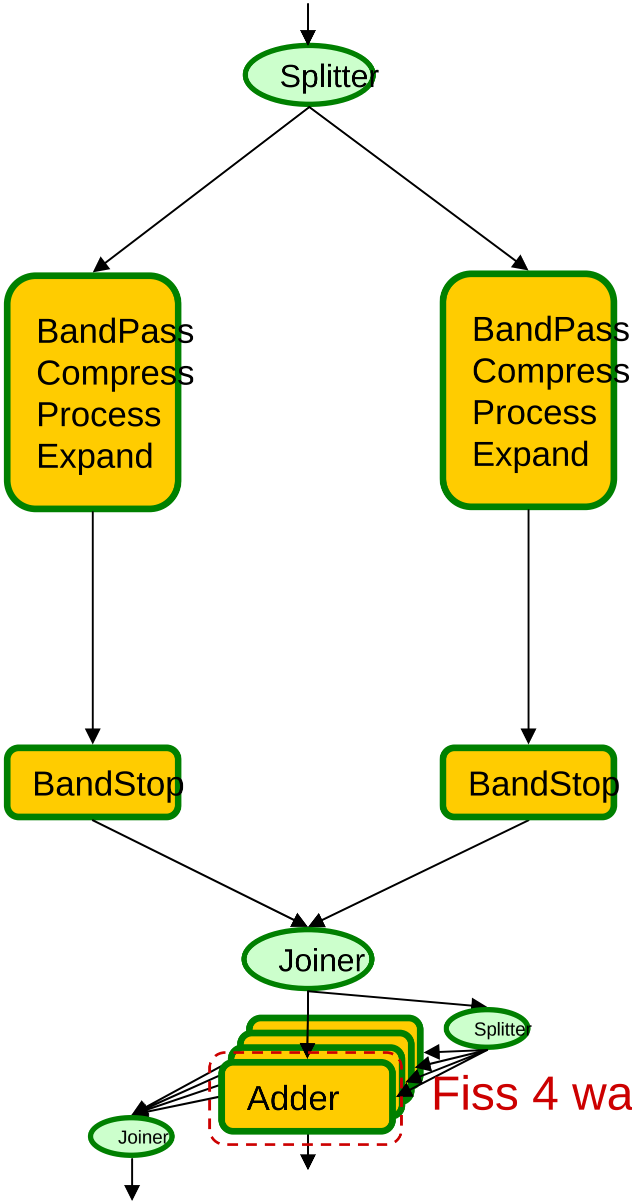
Phase 1: Coarsen the Stream Graph



- 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

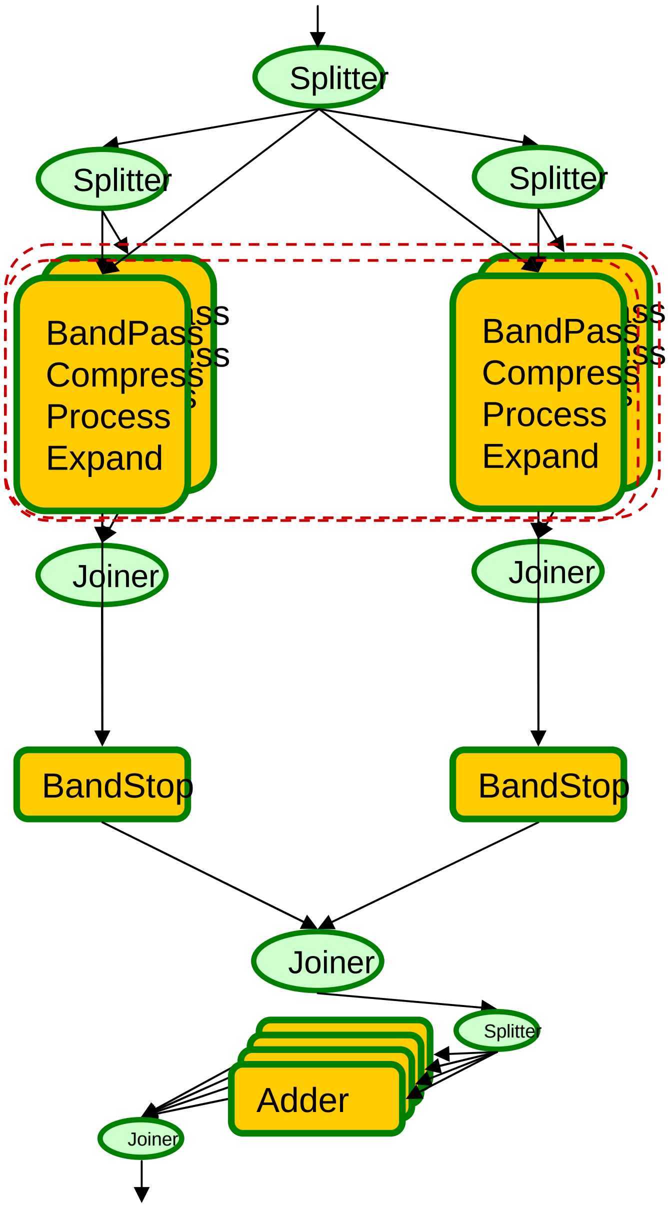
Phase 2: Data Parallelize

Data Parallelize for 4 cores



Fiss 4 ways, to occupy entire chip

Phase 2: Data Parallelize



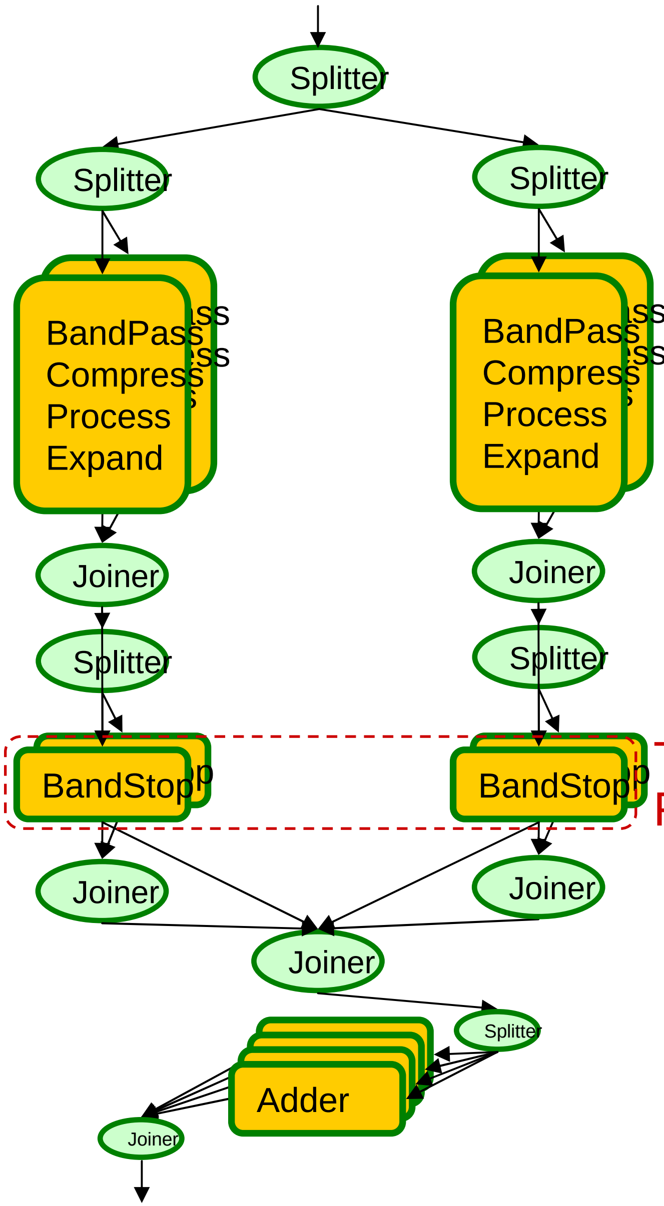
Data Parallelize for 4 cores

Task parallelism!

Each fused filter does equal work

Fiss each filter 2 times to occupy entire chip

Phase 2: Data Parallelize

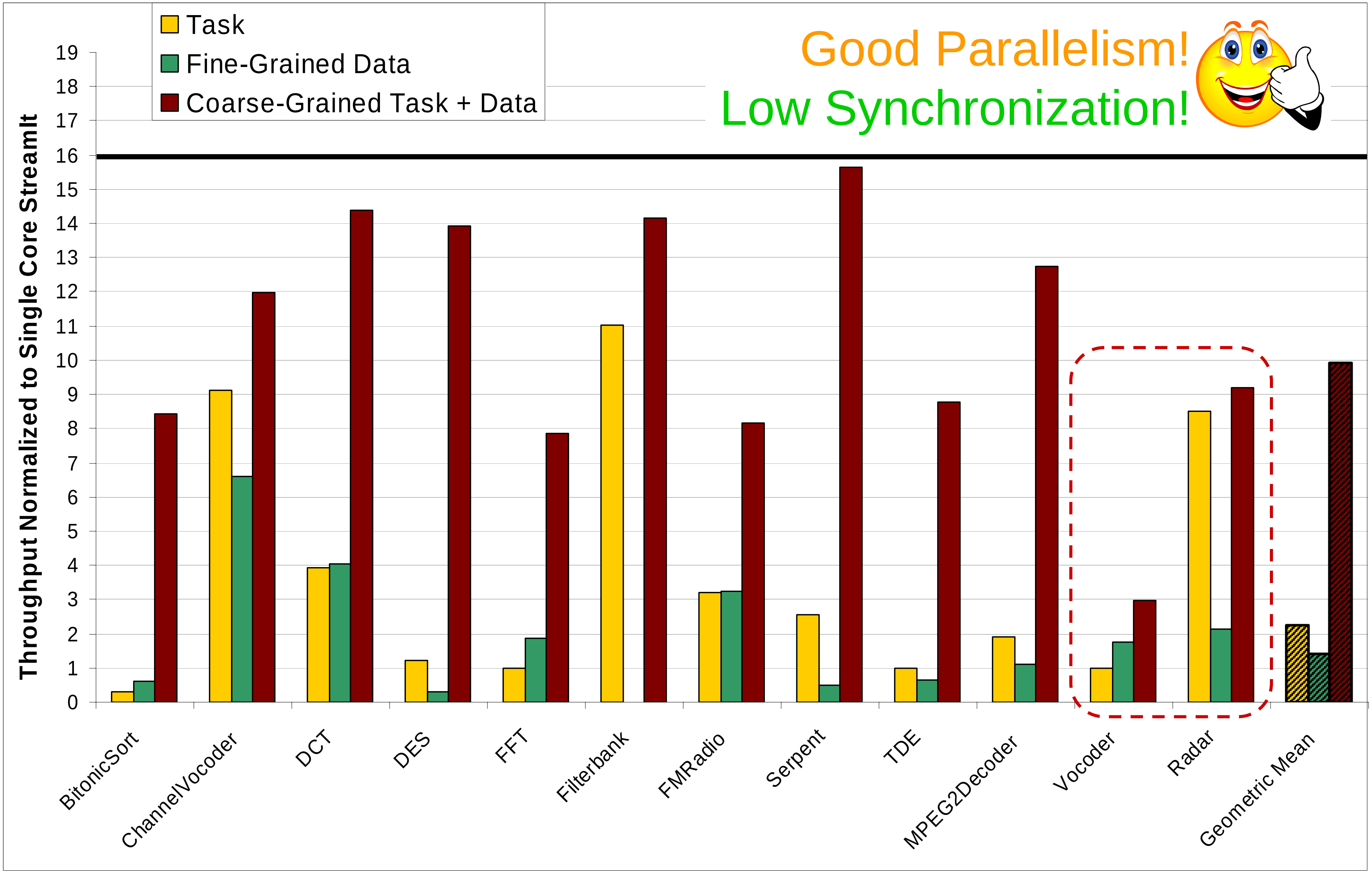


Task parallelism, each filter does equal work
Fiss each filter 2 times to occupy entire chip

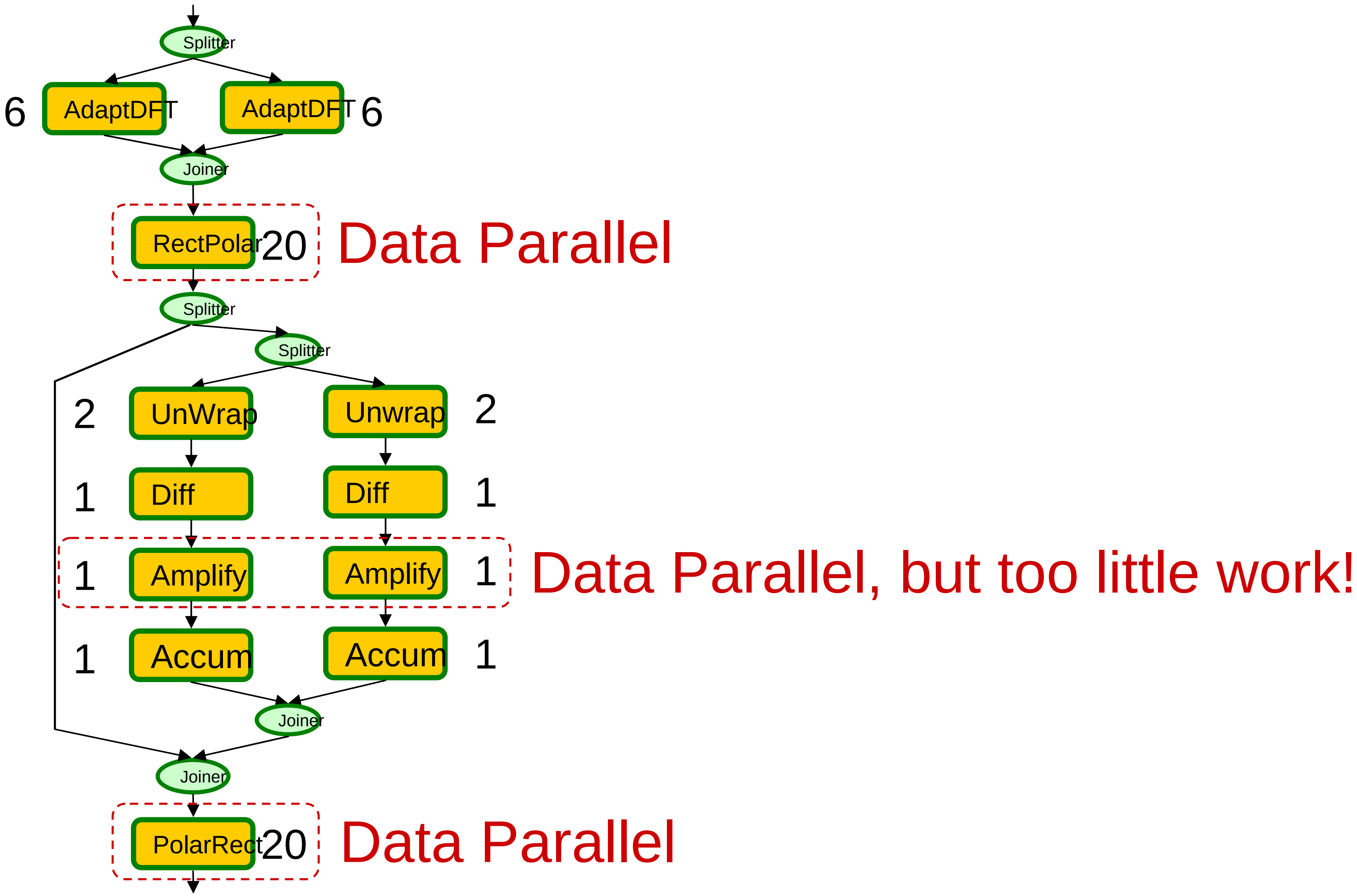
Data Parallelize for 4 cores

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

Evaluation: Coarse-Grained Data Parallelism

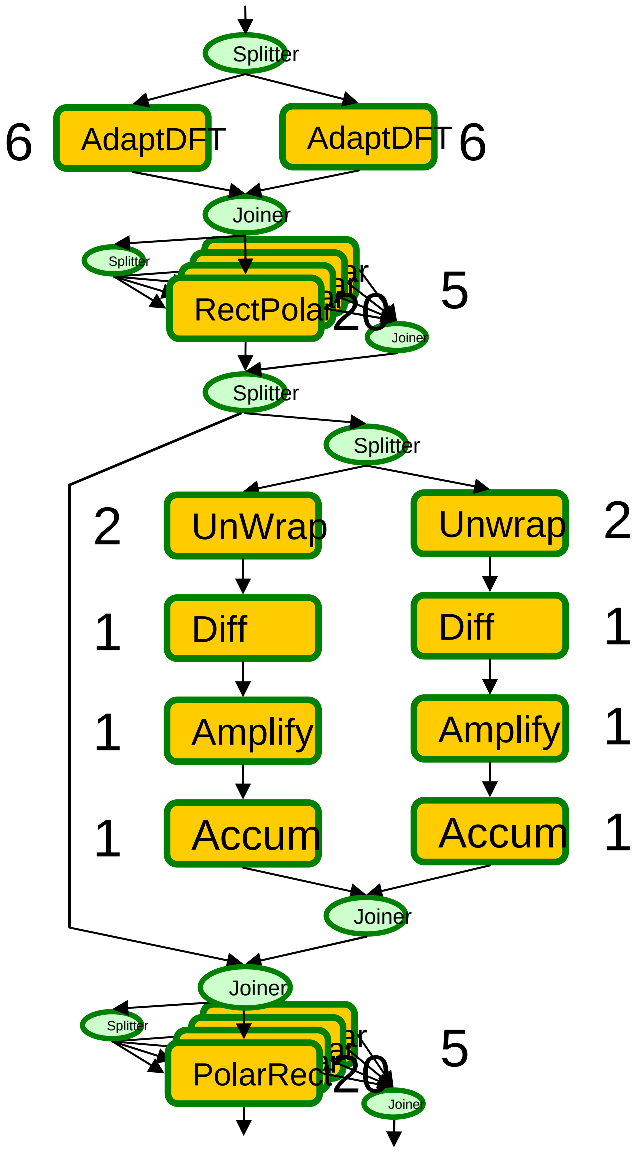


Simplified Vocoder



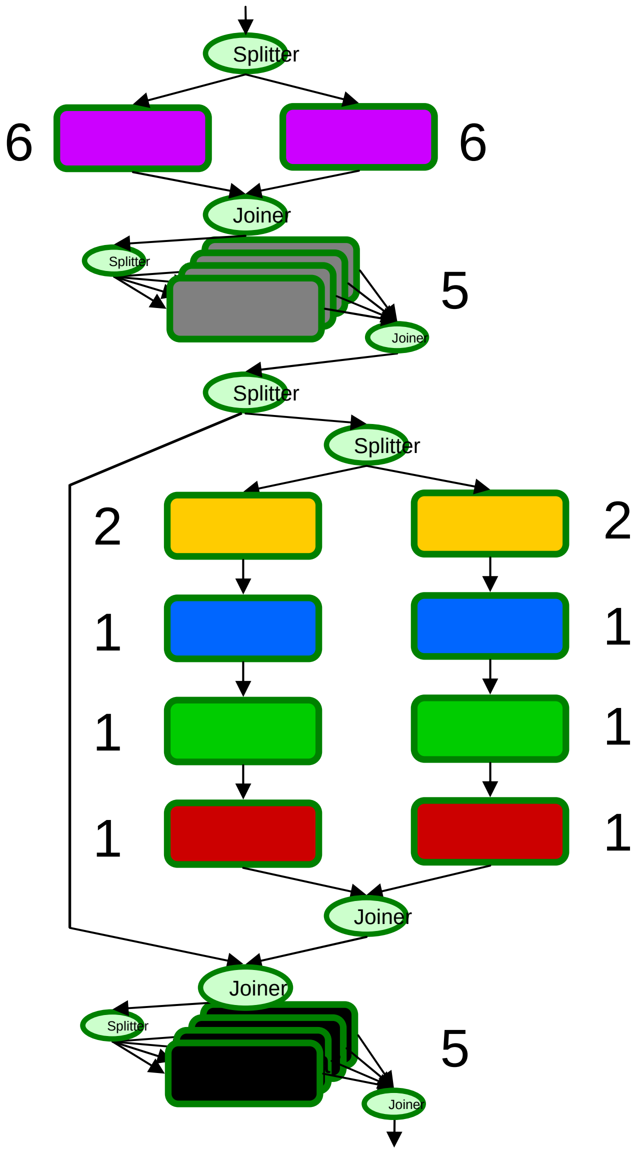
Target a 4 core machine

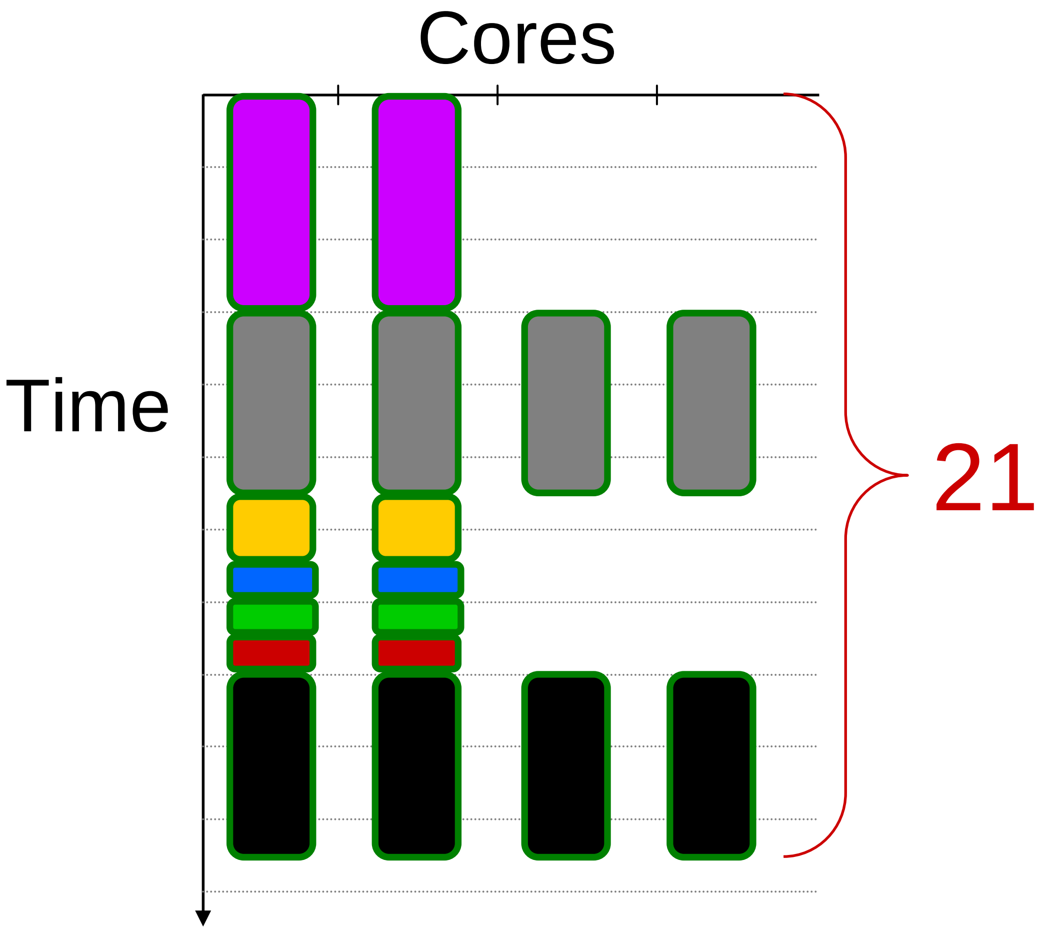
Data Parallelize



Target a 4 core machine

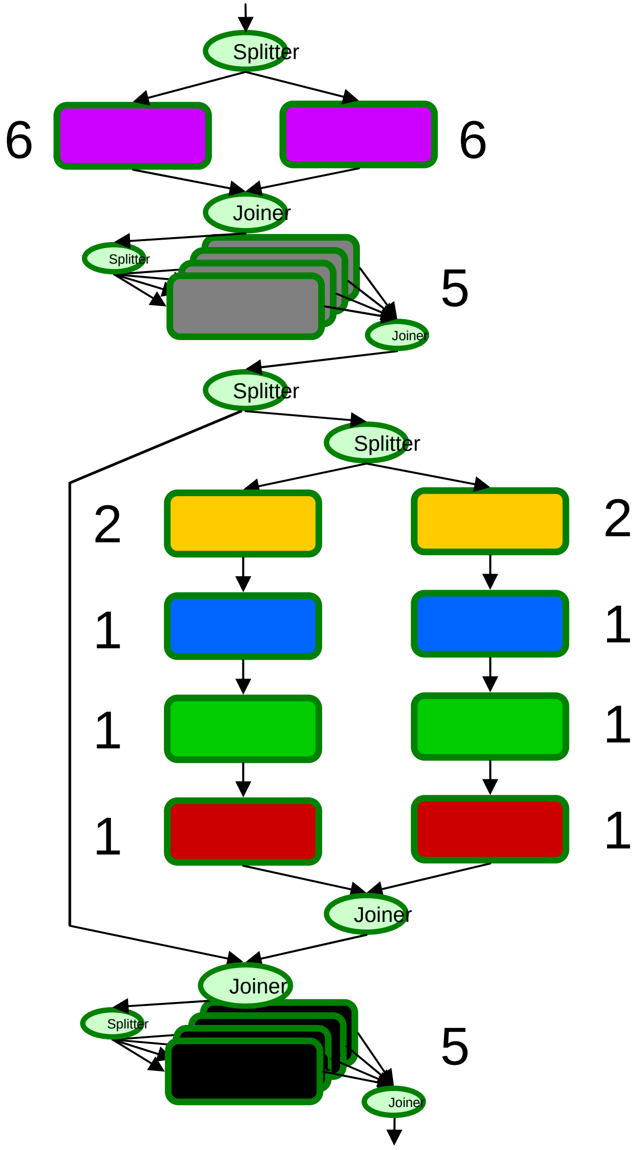
Data + Task Parallel Execution

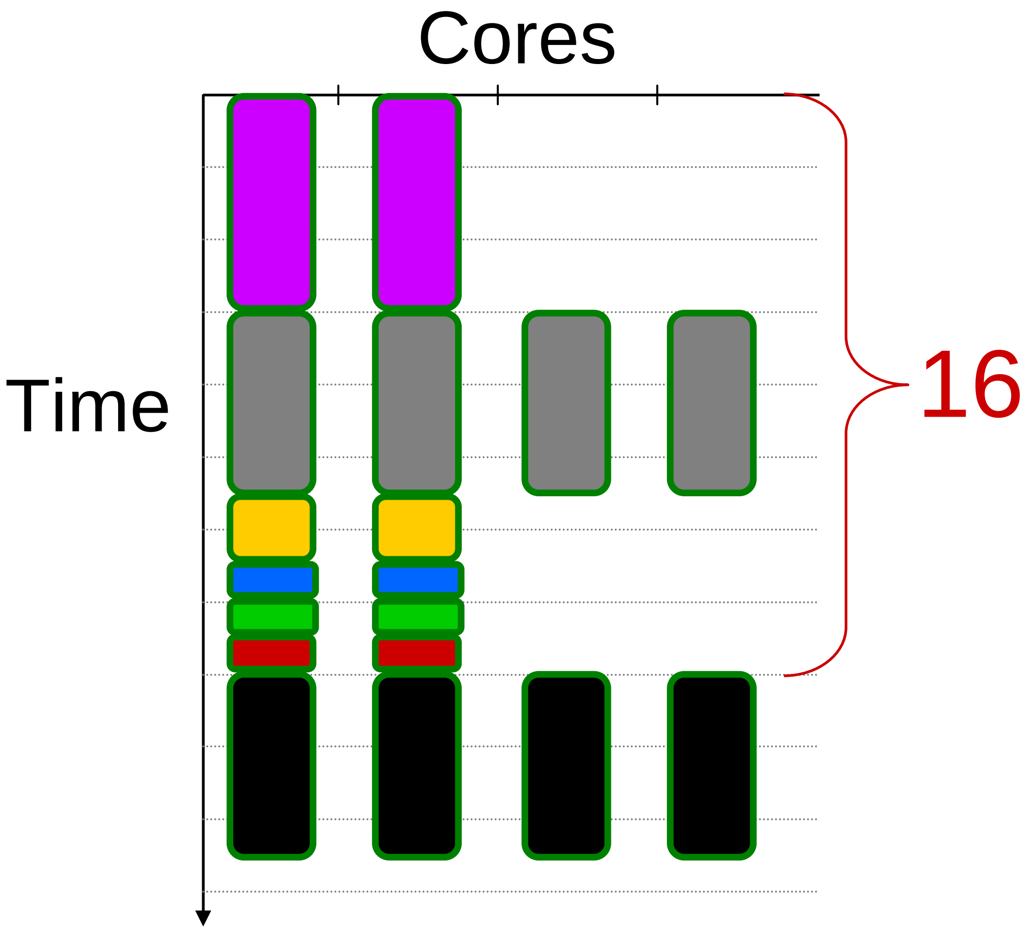




Target 4 core machine

We Can Do Better!



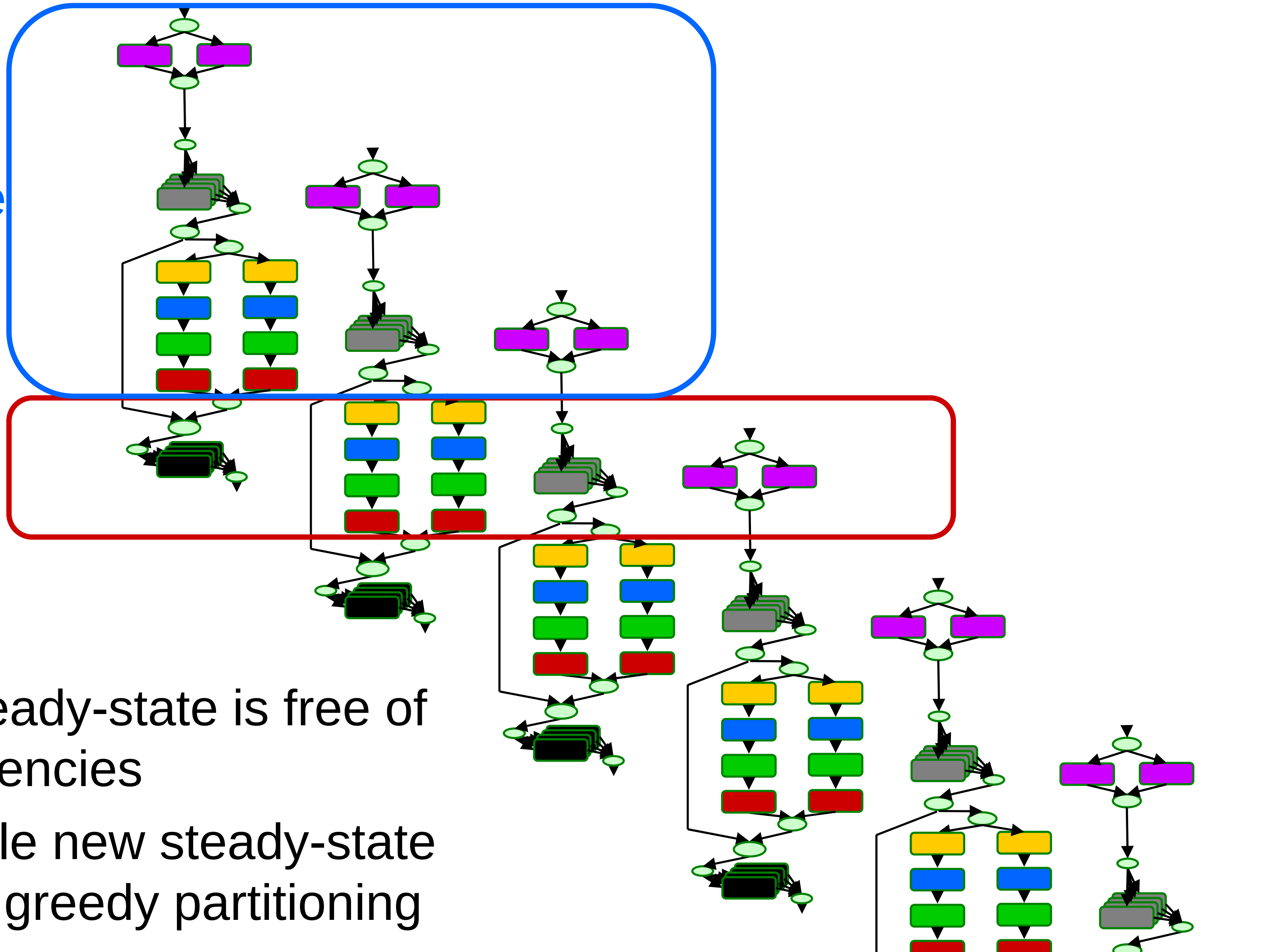


Target 4 core machine

Phase 3: Coarse-Grained Software Pipelining

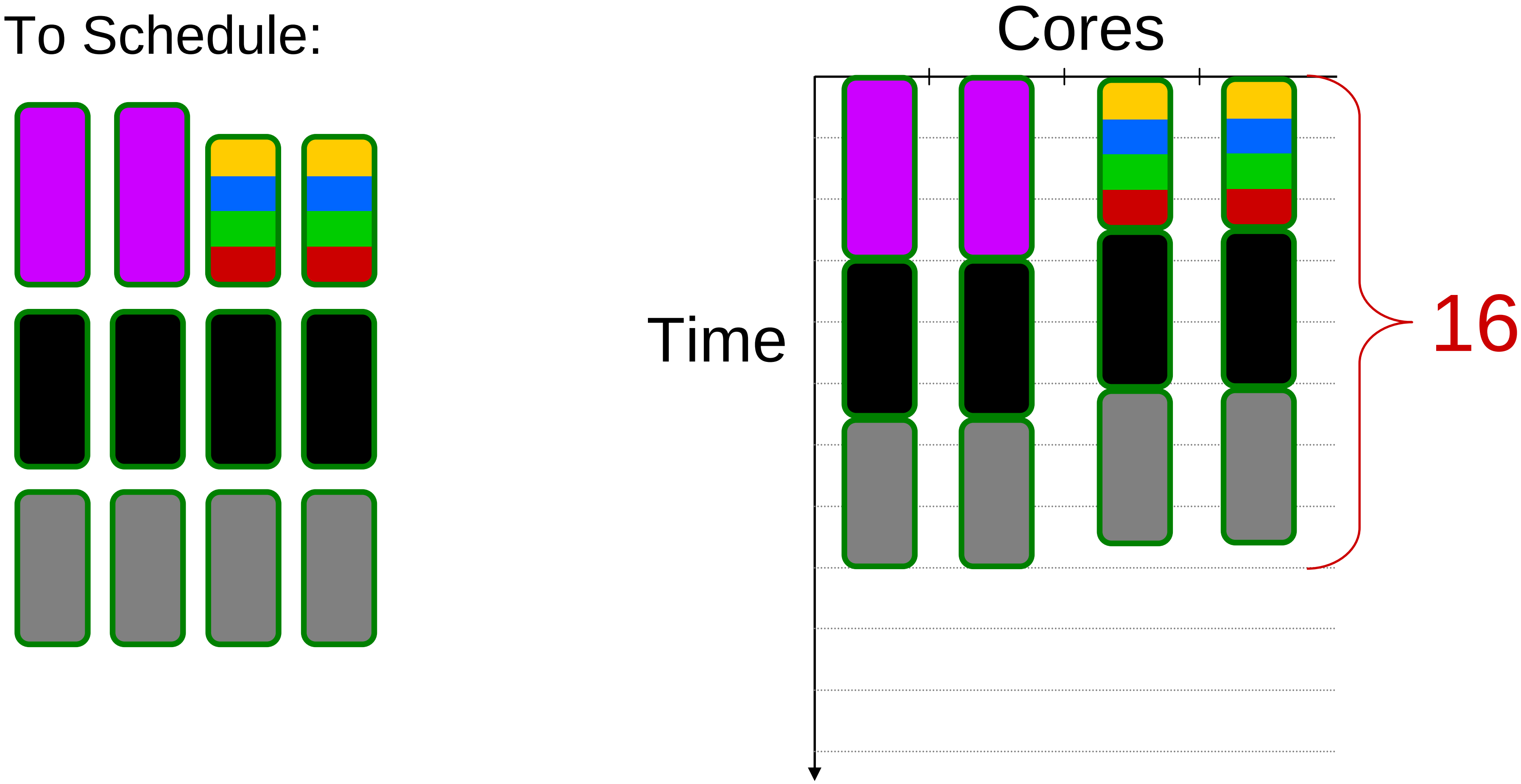
Prologue

New Steady State



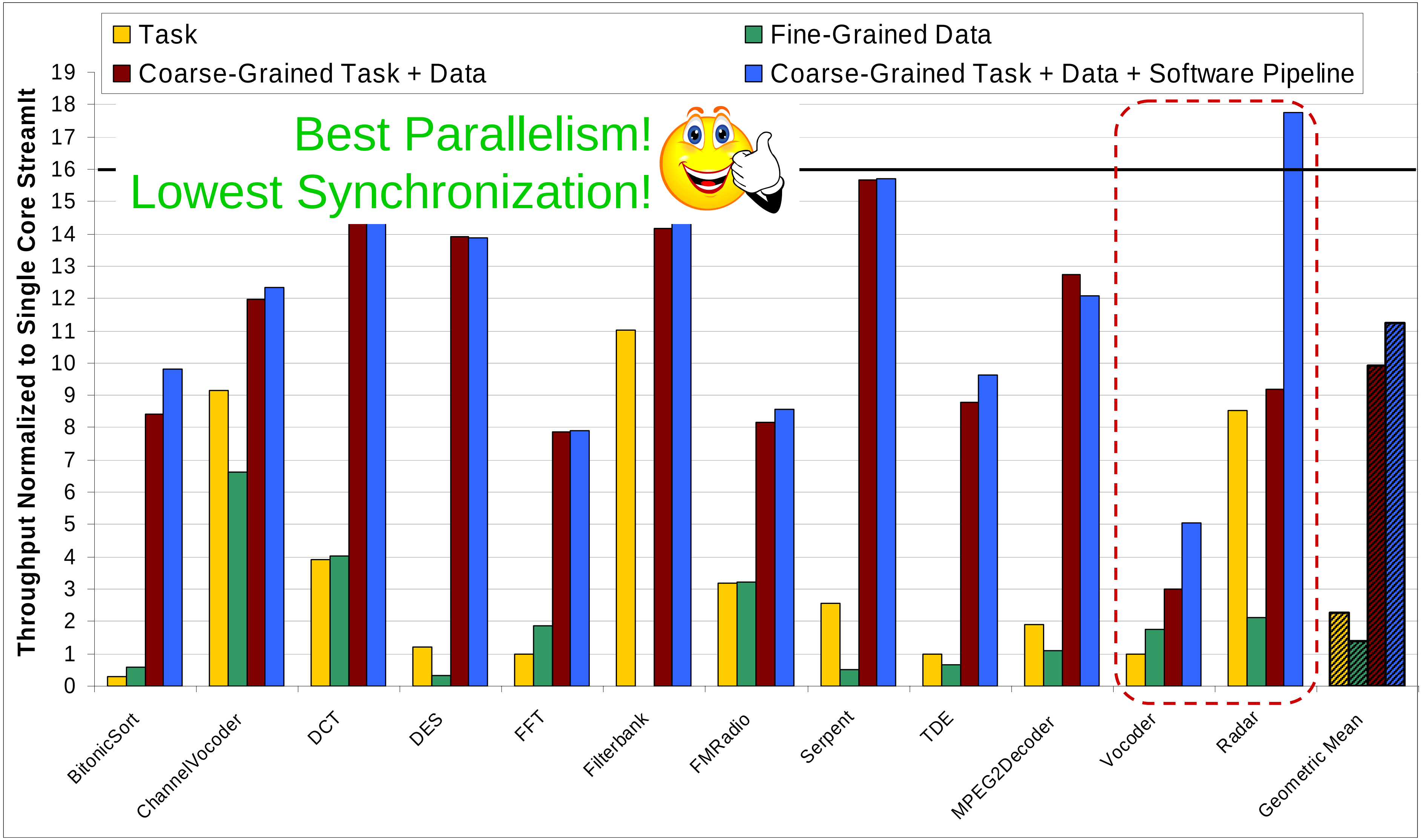
- New steady-state is free of dependencies
- Schedule new steady-state using a greedy partitioning

Greedy Partitioning

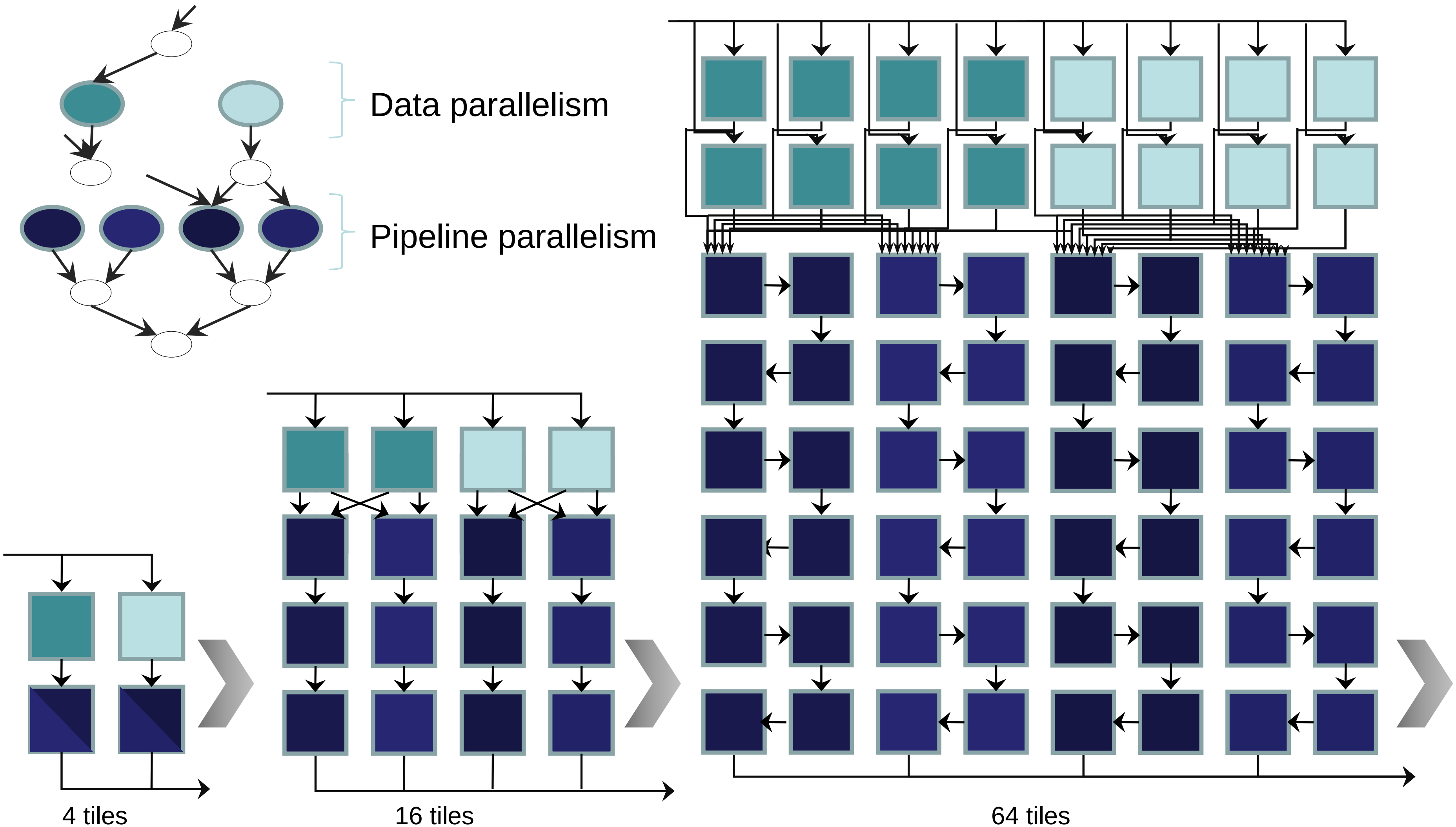


Target 4 core machine

Evaluation: Coarse-Grained Task + Data + Software Pipelining



Next: Scalable Stream Representation



Conclusions

- Computer Architecture is at a cross roads
  - Once in a lifetime opportunity to redesign from scratch
  - How to use the Moore's law gains to improve the programmability?
- Switching to multicores without losing the gains in programmer productivity may be the Grandest of the Grand Challenges
  - Half a century of work $\Rightarrow$ still no winning solution
  - Will affect everyone!
- Streaming programming model
  - Can break the von Neumann bottleneck
  - A natural fit for a large class of applications
  - An ideal machine language for multicores.
- Compiler can extract explicit and inherent parallelism
  - Parallelism is abstracted away from architectural details of multicores
  - Sustainable Speedups (5x to 19x on the 16 core Raw)
  - Increased abstraction does not have to sacrifice performance

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