Introduction to Parallel Programming Models

Tim Foley

Stanford University

Overview

- Introduce three kinds of parallelism
  - Used in visual computing
  - Targeting throughput architectures

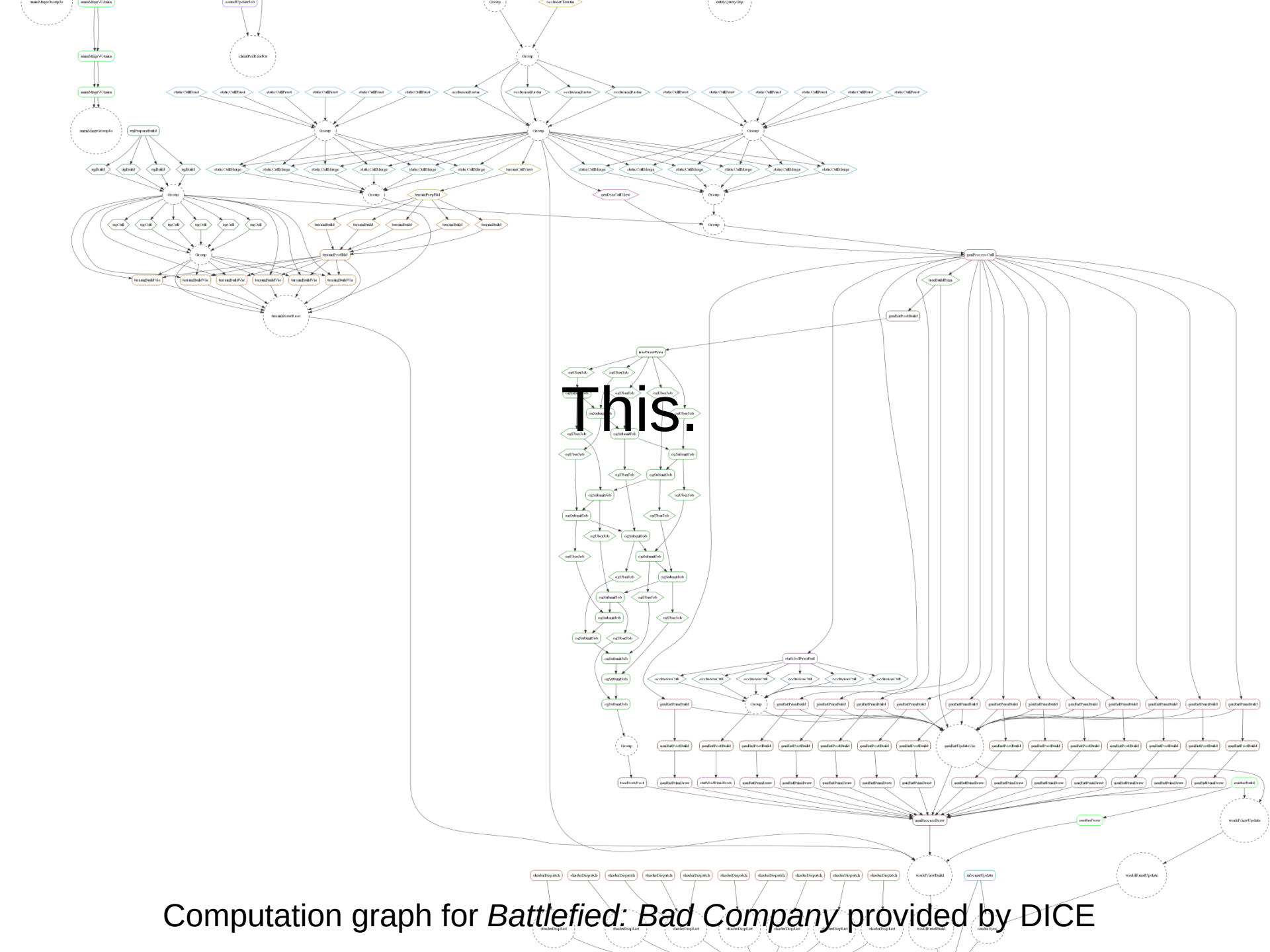
- Goals
  - Establish basic terminology for the course
  - Recognize idioms in your workloads
  - Evaluate and select tools

Scope

- Games as representative application
  - Demand high performance, visual quality
  - Already using MC, throughput and heterogeneous HW
  - Visibility, illumination, physics, simulation

- Not covering every possible approach
  - Explicit threads, locks
  - Message-passing/actors/CSP
  - Transactions/REST

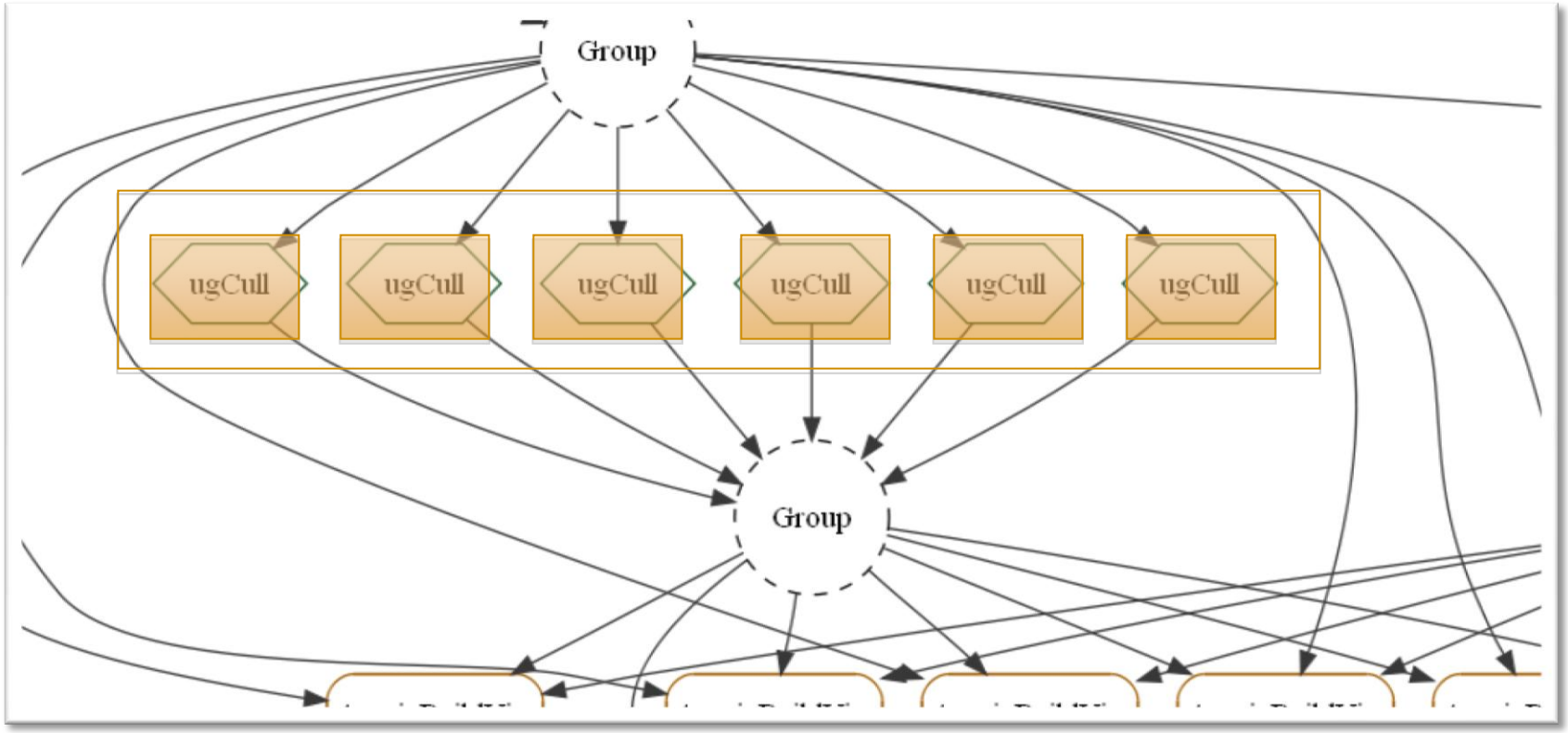
What goes into a game frame?



Computation graph for *Battlefied: Bad Company* provided by DICE

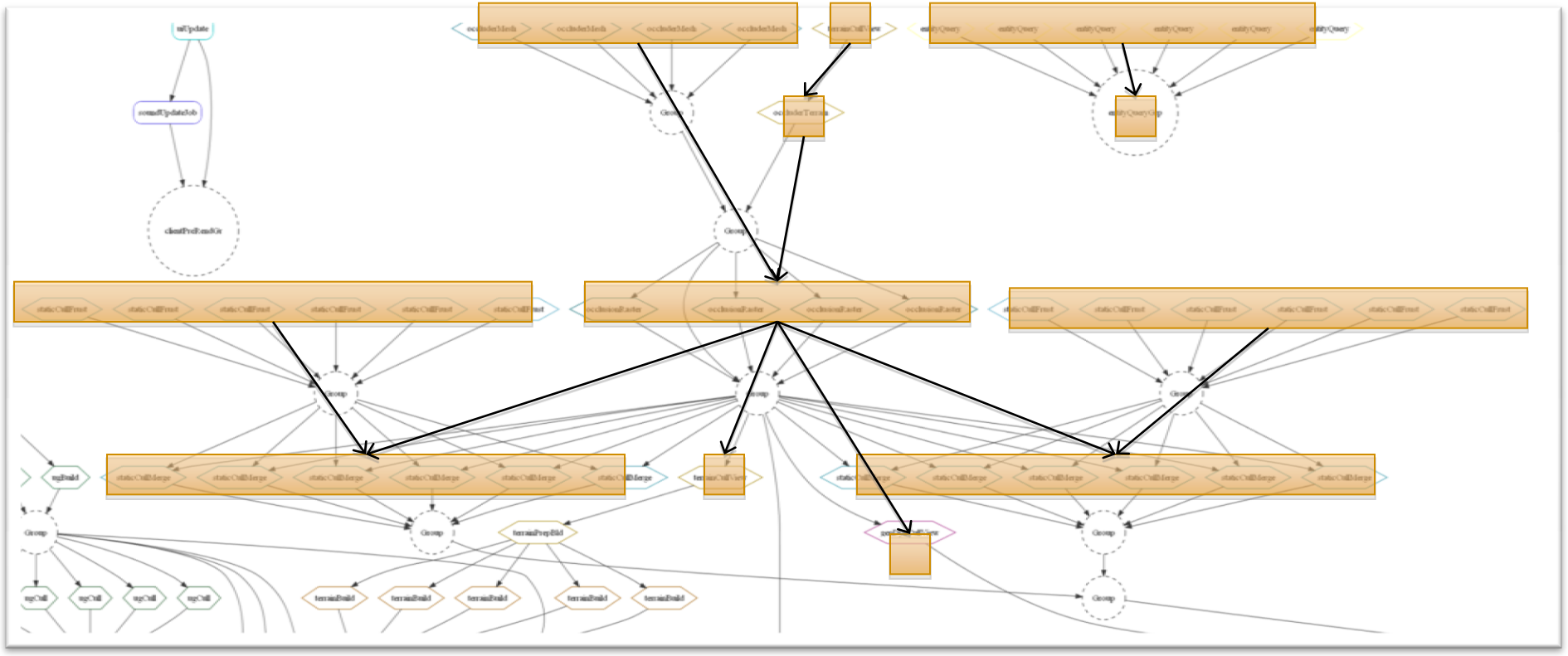
A modern game is a mix of…

Data-parallel algorithms



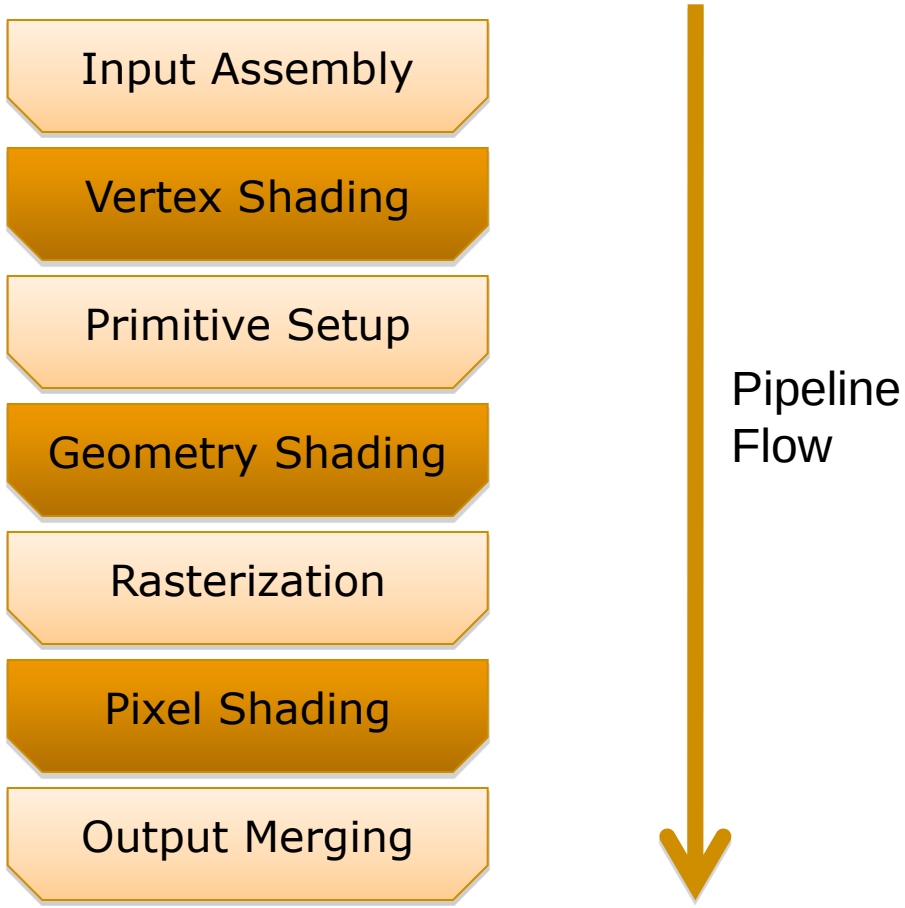
A modern game is a mix of…

Task-parallel algorithms and coordination

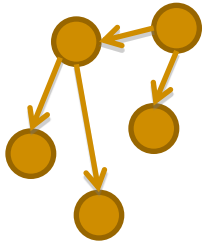


A modern game is a mix of…

Standard and extended graphics pipelines



Data-Parallel



Task-Parallel

Pipeline-Parallel

Structure of this talk

- For each of these approaches
  - Key idea
  - Mental model
  - Applicability

- Composition
  - How these models combine in the real world

Caveats

- Turing Tar Pit
  - Just being able to express it doesn't make it fast!
- Most general model is not always best
  - Constraints are what enable optimizations
- Not every model requires dedicated tools
  - These patterns can be expressed in many languages

Data parallelism

Key Idea

- Run a single kernel over many elements
- Per-element computations are independent
- Can exploit throughput architecture well
  - Amortize per-element cost with SIMD/SIMT
  - Hide memory latency with lightweight threads

Mental Model

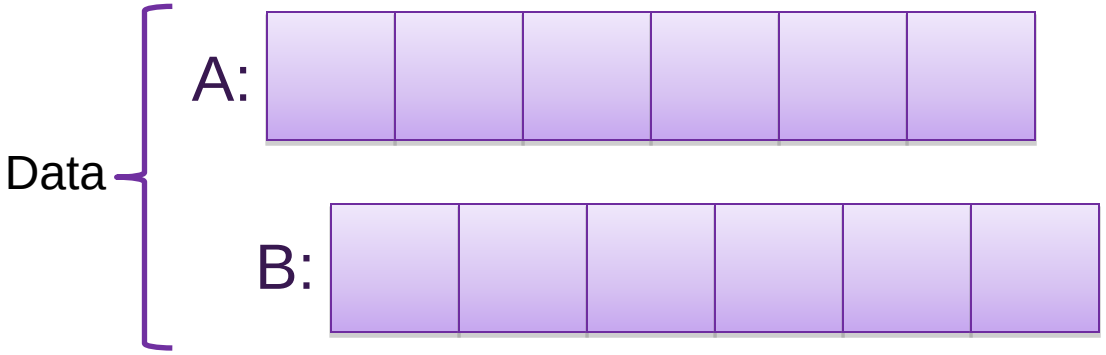
- Execute **N** independent **work items**
  - aka “elements”, “fragments”, “strands”, “threads”
- All work items run the same program: **kernel**
- Work item uses data determined by 0 <= i < **N**
  - [0, **N**) is the **domain of computation**

Domain of computation

- Determines number and “shape” of work items
- Often based on input/output data structure
  - Not required – domain and data may be decoupled
- Many domain “shapes” possible
  - Regular
  - Nested
  - Irregular

Simple Data-Parallelism

- Data structure
  - Regular array

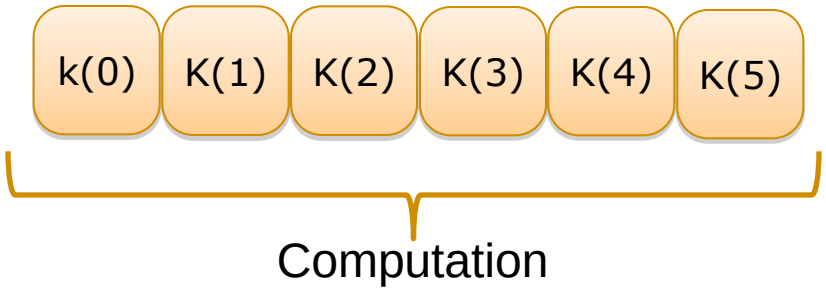


- Kernel

```
void k(int i) {
  B[i] += A[i];
}
```

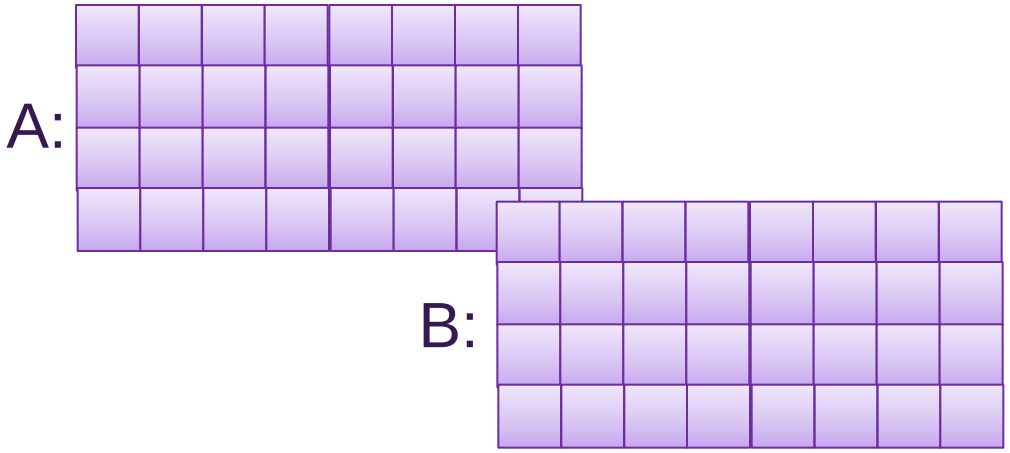
Program

- Domain of computation
  - 1D interval



Simple Data-Parallelism

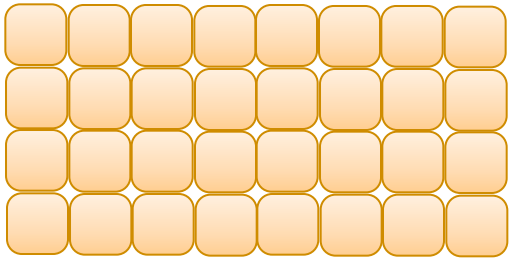
- Data structure
  - N-D array



- Kernel

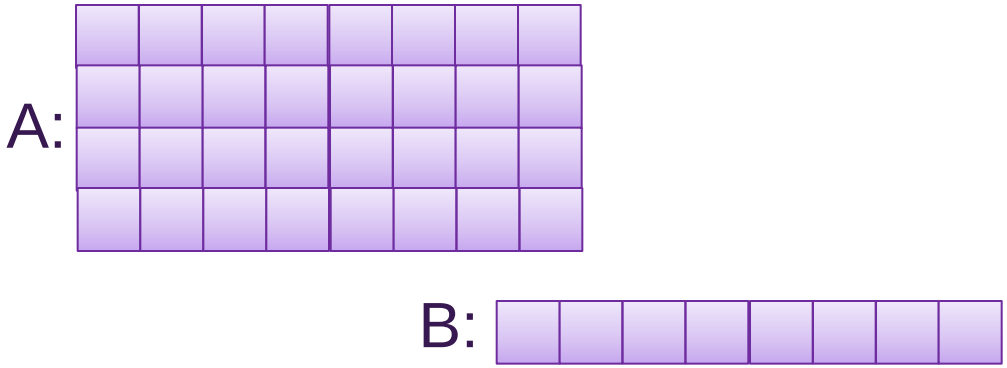
```
void k(int i, int j) {
  B[i][j] += A[i][j];
}
```

- Domain of computation
  - N-D interval



Shapes need not match

- Data structure
  - N-D array
  - 1D array



- Kernel

```
void k(int i) {
  for(int j = 0; j < M; j++)
    B[i] += A[i][j];
}
```

- Domain of computation
  - N-D interval

Advanced data-parallelism

- Hierarchical domains
  - Allow work items to communicate
  - Useful for sums, scans, sorts
- Irregular domains
  - Nested or “ragged” data structures

“Flat” domains

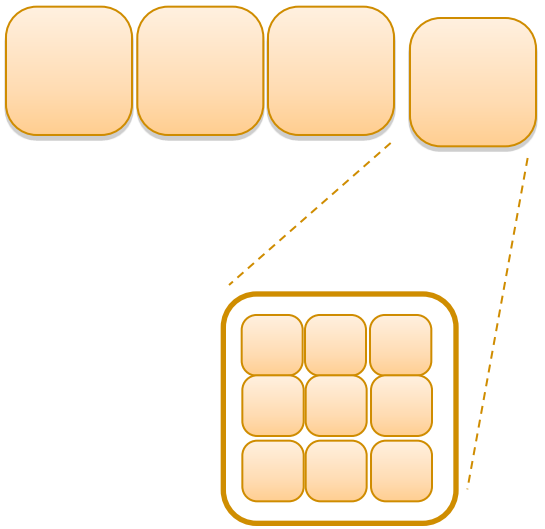
- Kernel temporaries / scratch data are
  - Private: inaccessible to other work items
  - Transient: inaccessible after work item completes
- Flat domain exposes work-item locality
- Optimization: put scratch in register file or caches

Communication

- Need to communicate intermediate results
  - Each work item computed value, now want sum
- Write to main memory, launch a new kernel?
  - Don’t exploit locality, rest of memory hierarchy
- Employ a hierarchy of domains

Hierarchical domains

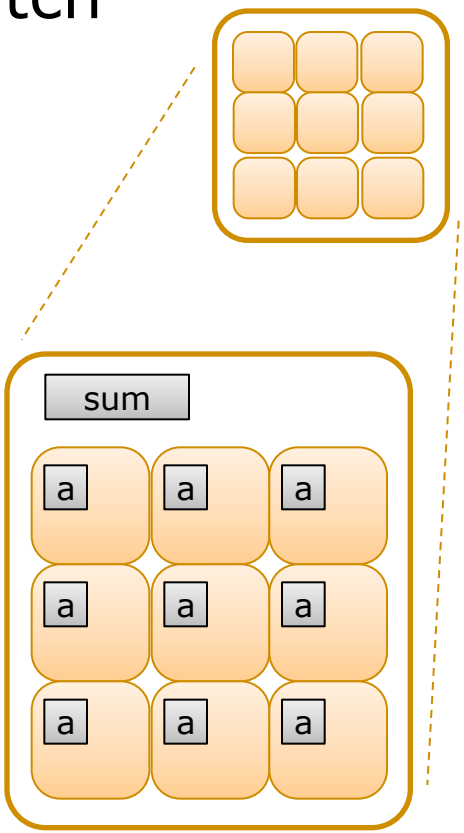
- A domain composed of smaller domains
  - Each level has its own scratch memory
  - Often tied to memory hierarchy
    - ex. Registers, L1$, L2$, DRAM



- Work item can access
  - Kernel parameters
  - Own scratch memory
  - Scratch memory of ancestors in hierarchy

Hierarchical domains

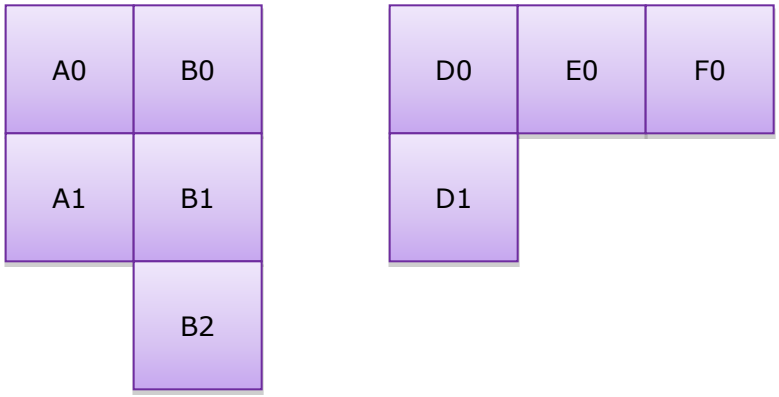
- Communicate through parent item scratch
  - ex. Each element computes value “a”
  - Add local value into shared “sum”
- Data races are now possible
  - Atomic operations
  - Synchronization barriers
- Also possible for global memory…



Irregular Domains

- "Ragged array" data structure
  - N-D array- / grid-of-lists

{{A0,A1}, {B0,B1,B2}, {}, {D0,D1}, {E0}, {F0}}

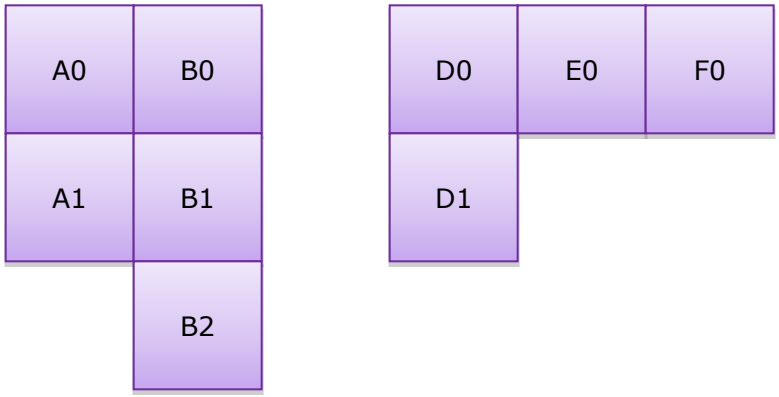


- Used for
  - Bucketing: particles in a cell
  - Collision: potential collidees
  - …

Irregular Domains

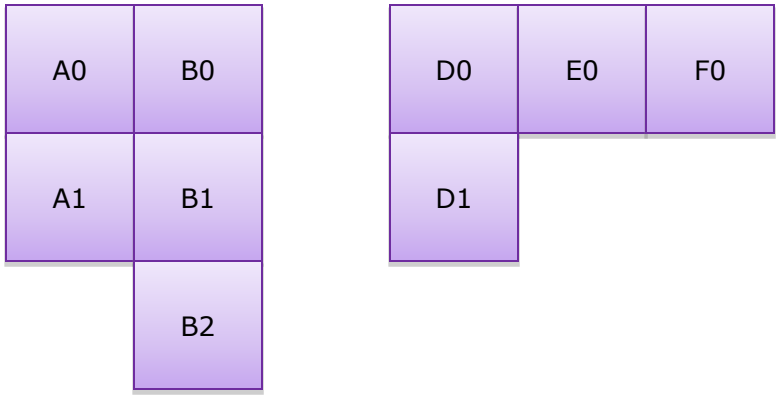
- Must choose in-memory representation
  - Pointer per bucket?

{{A0,A1}, {B0,B1,B2}, {}, {D0,D1}, {E0}, {F0}}



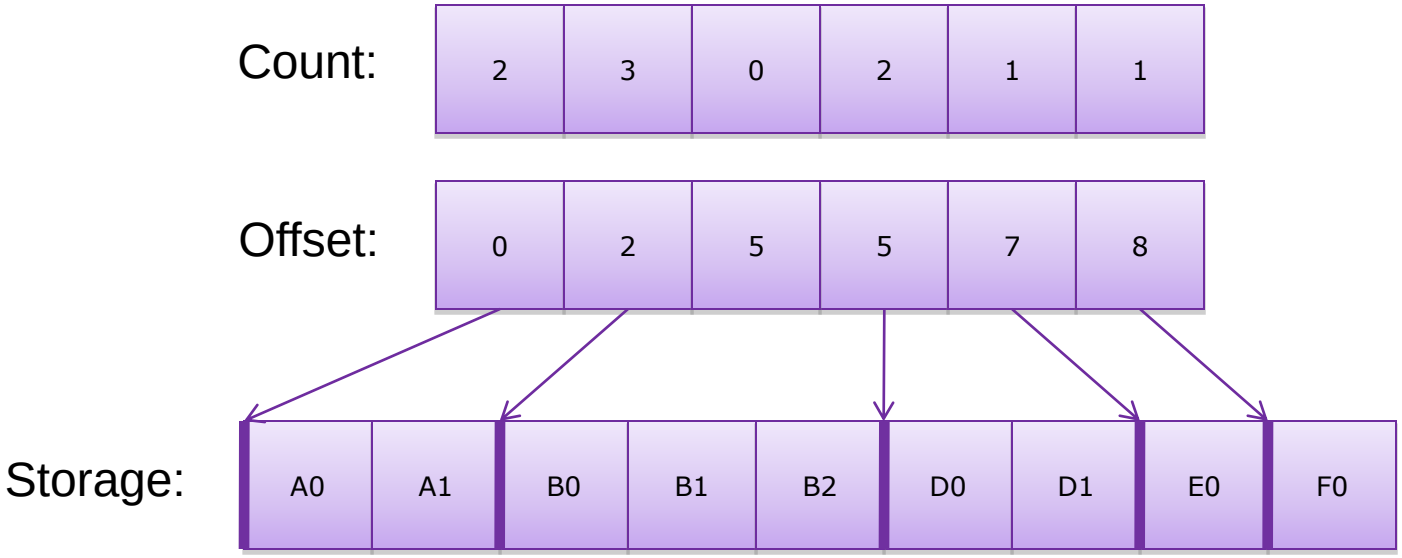
- Performance
- Required operations
  - Apply kernel to each bucket?
  - Apply kernel to each element?

A simple representation

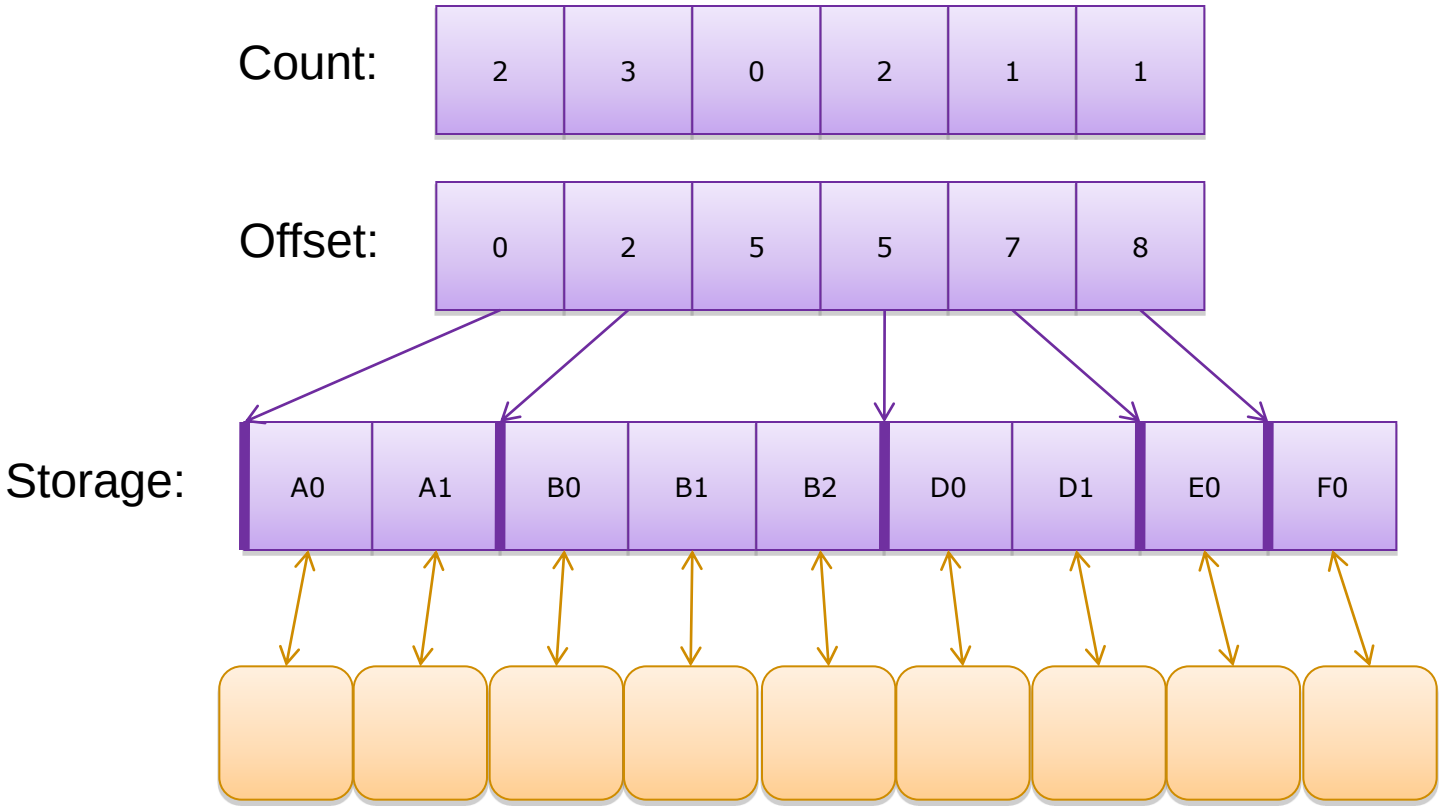


Logical

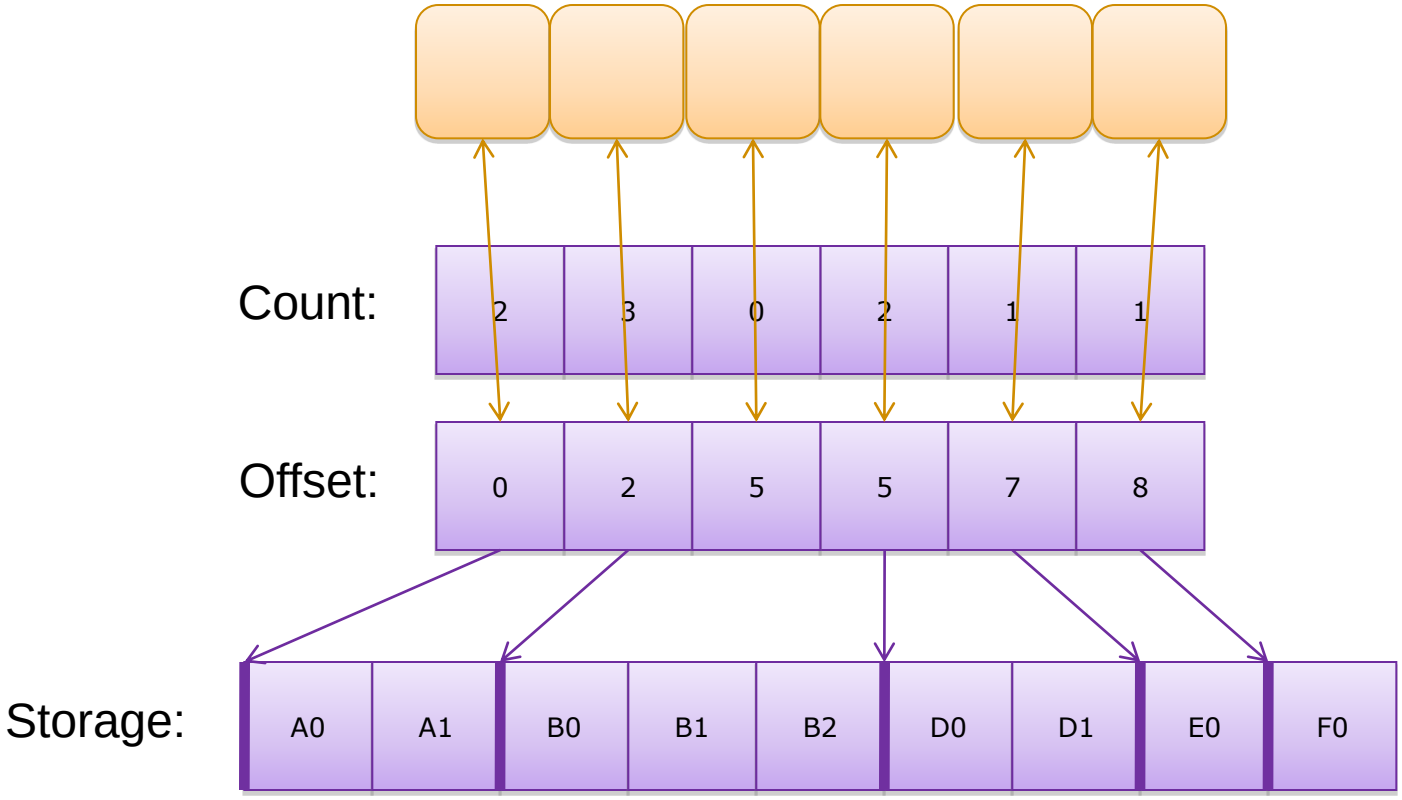
Physical



Apply to each element



Apply to each bin



Irregular data parallelism

- Key insight: represent irregular structure as flat index and storage arrays
  - Many other representations possible

- Allows efficient data-parallel implementation of some irregular algorithms
  - Many examples in the literature

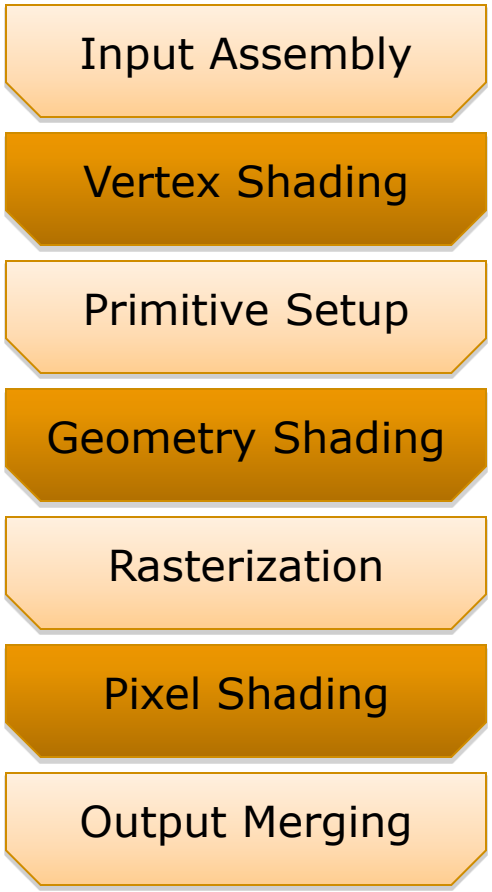
Pipeline parallelism

Key Idea

- Algorithm is an ordered sequence of stages
  - Each stage emits zero or more items
- Increase throughput by running stages in parallel
- Exploit producer-consumer locality
  - On-chip FIFOs
  - Efficient bus between cores

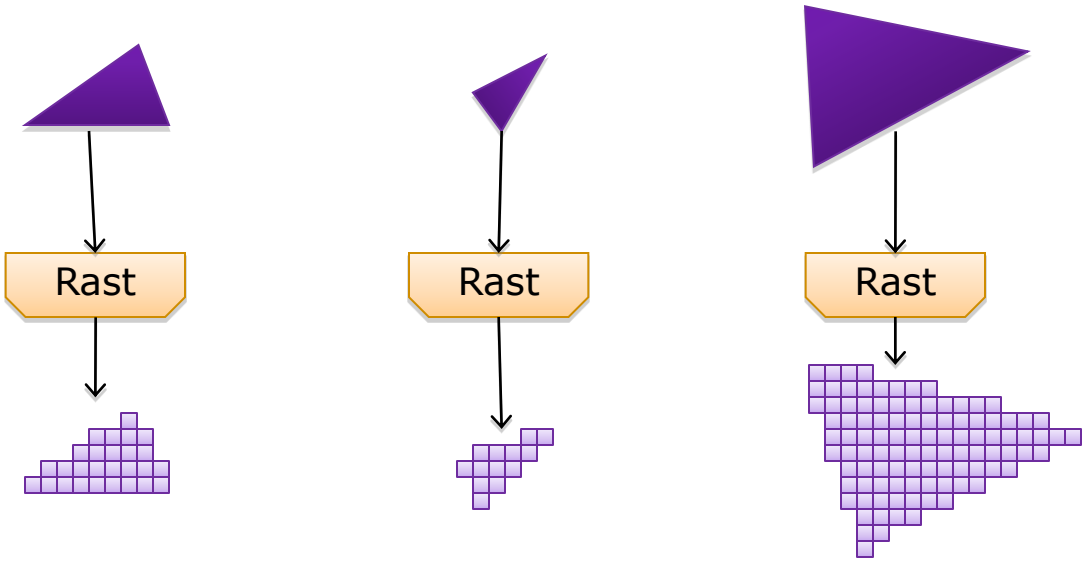
GPU Pipeline (DX10)

- Pipeline of
  - Fixed-function stages
  - Programmable stages
    - Data-parallel kernels
- Stages run in parallel
  - Even for unified cores
- Queues between stages
  - Often in HW



Why pipelines?

- Variable rate amplification
  - Rasterizer: 1 tri in, 0-N fragments out
  - Ray tracer: 1 hit in, 0-N secondary/shadow rays out
  - Load imbalance



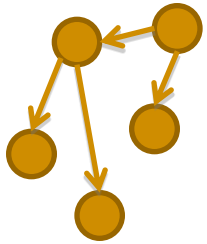
Pipelines can cope with imbalance

- Re-balance load between stages
  - Buffer up results for next stage
- Optimize for locality
  - Specialized inter-stage FIFOs
  - On-chip caches, busses or scratchpads

User-defined pipelines

- Standard practice for console developers
  - Custom Cell/RSX graphics pipelines on PS3
- Pipeline-definition tools still research area
  - GRAMPS [Sugerman et al. 2009]
- Challenges
  - Bounding intermediate storage
  - Scheduling algorithms

Task parallelism



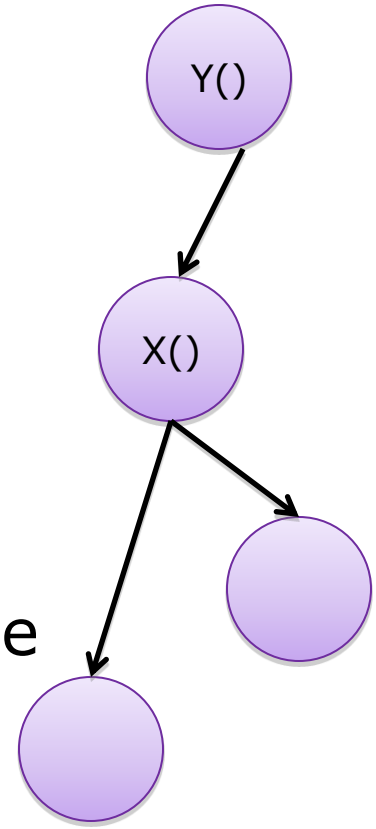
Key Idea

- Achieve scalability for heterogeneous and irregular work by expressing dependencies directly

- Lightweight cooperative scheduling

What is a Task?

- Think of it as an asynchronous function call
  - "Do X at some point in the future"
  - Optionally "… after Y is done"
- Might be implemented in HW or SW
- Almost always cooperative, not preemptive



Why tasks?

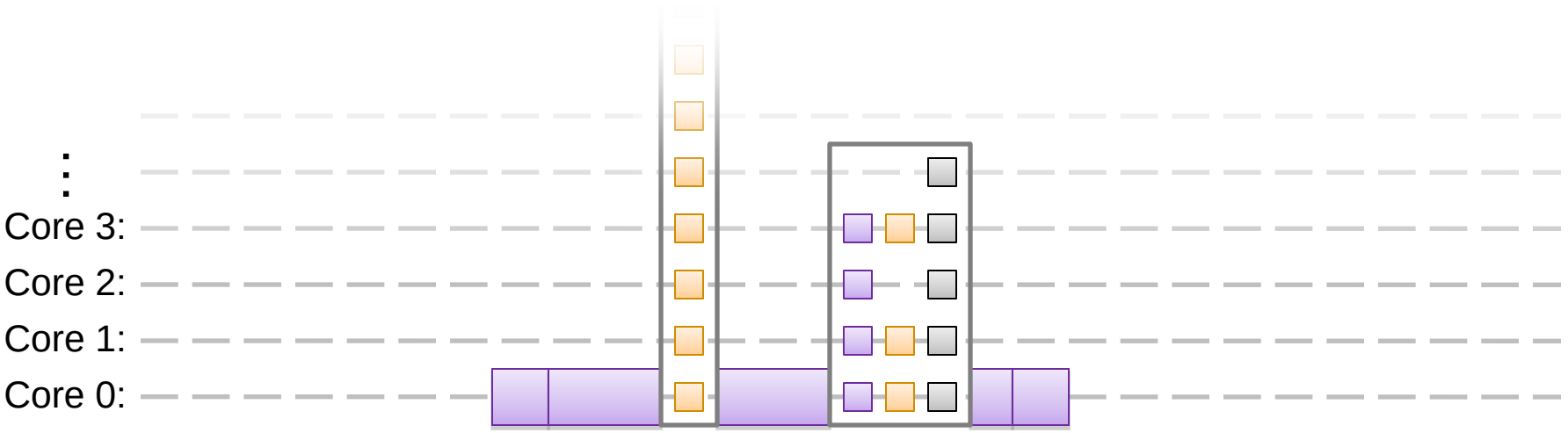
- Start with sequential workload

Why tasks?

- Identify data- and pipeline-parallel steps

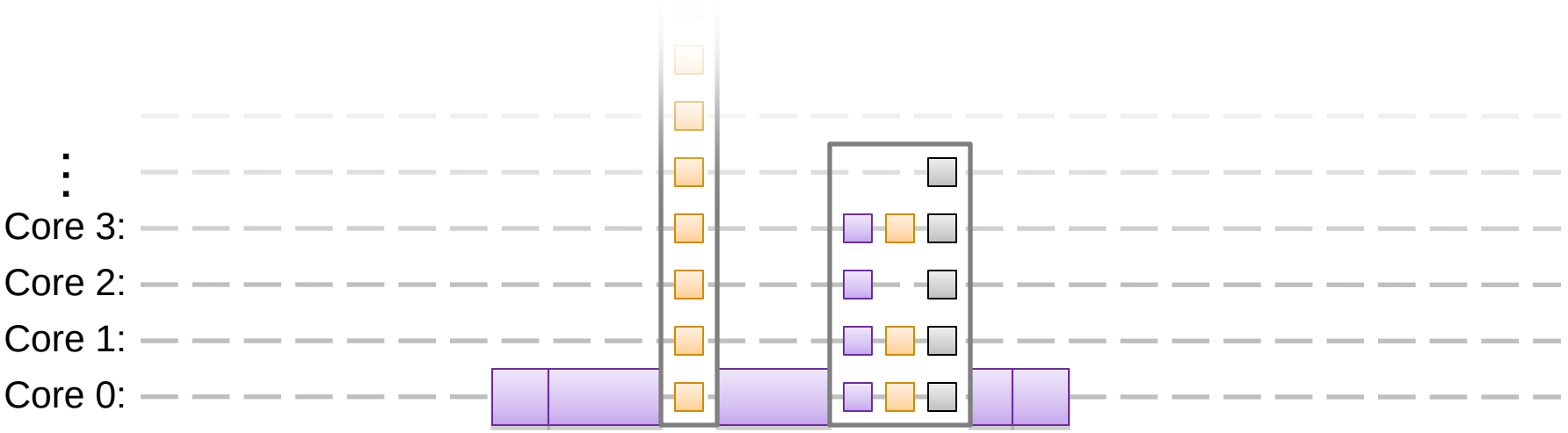
Why tasks?

- Identify data- and pipeline-parallel steps
- Assume perfect scaling



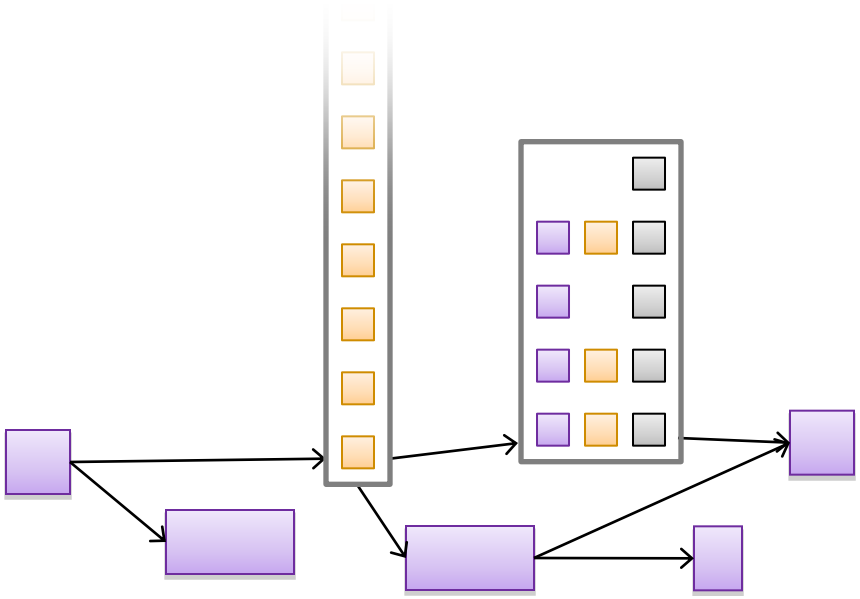
Why tasks?

- Cost now dominated by sequential part
  - The part not suited to data- or pipeline-parallelism
- Oh yeah… that's just Amdahl's Law



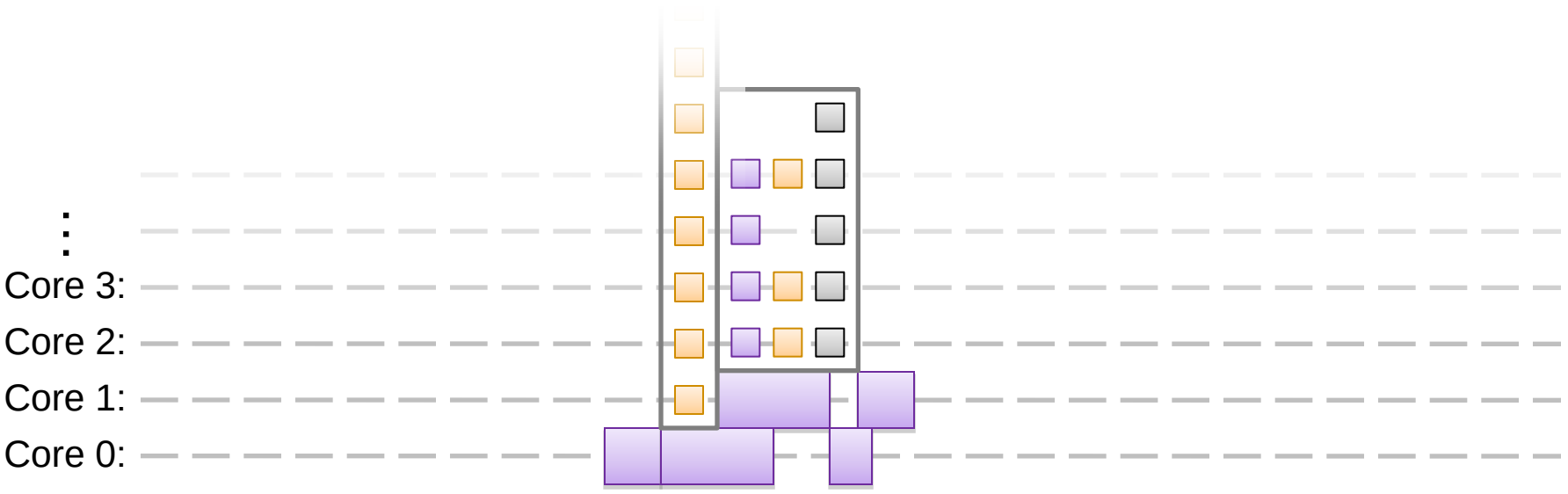
Using tasks

- If we know dependencies between the steps



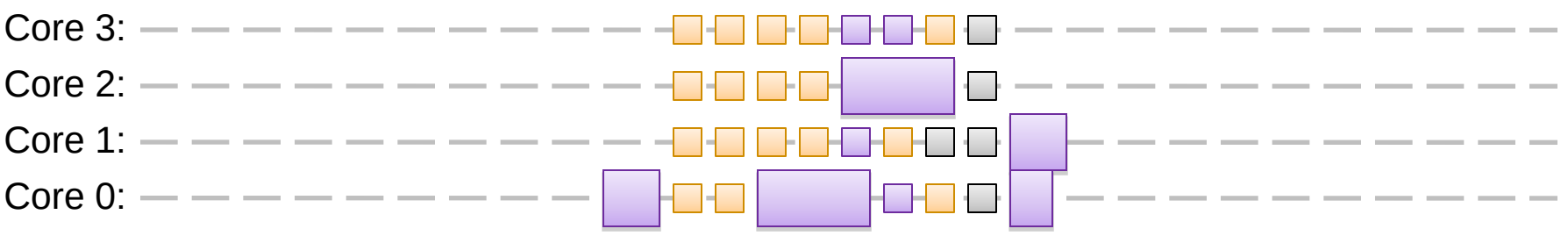
Using tasks

- If we know dependencies between the steps
- We can distribute the work across cores
  - Respecting the dependencies



Finite # of cores

- It looks more like this
  - Multiple kinds of work fill in the “cracks”

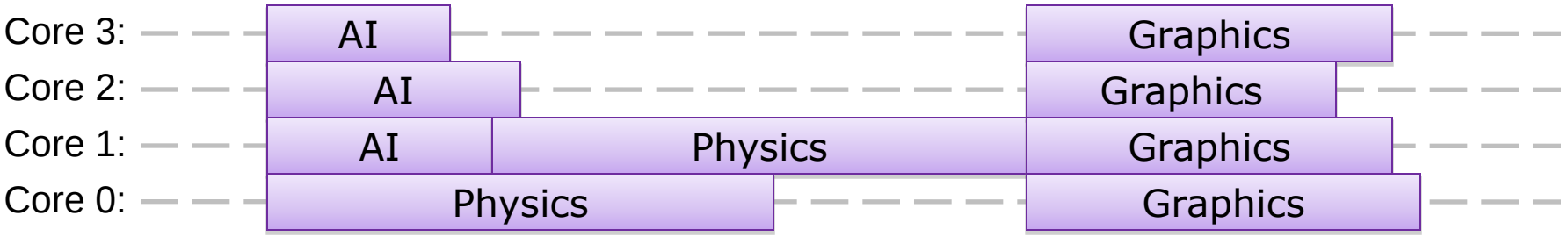


Task/job systems

- Standard practice for PS3 games
  - Gaining currency on other consoles, desktop
- One worker thread per HW context
  - Cooperative scheduling
  - Pull tasks from an incoming queue
  - Load balance using “work stealing” [Cilk]

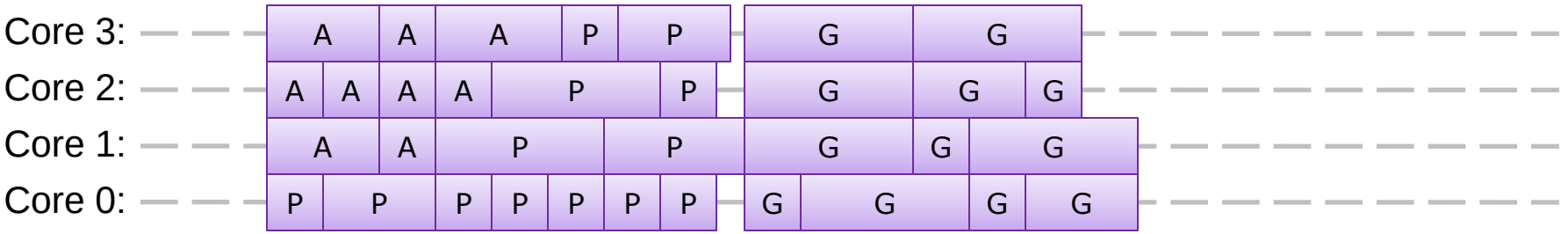
Task granularity

- Coarse-grained tasks easy to identify
- Can schedule poorly
  - Coarse-grained dependencies
  - "Bubble" waiting for predecessor to clear



Task granularity

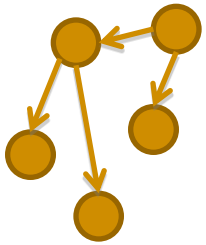
- Fine-grained tasks pack well
- More scheduling overhead
  - Tune task size to strike a balance



Tasks take-away

- Can’t write sequential app with parallel pieces
  - Amdahl’s Law will bite you every time
- Must involve parallelism from the top down
- Task systems
  - Handle the code that won’t fit other models
    - Heterogeneous, irregular
    - Dynamically generated work, dependencies
  - Provide scalability and load balancing

Composition



Picking the right tools

- No one model is best for all apps
  - Or even all parts of one app

- Real-world parallel apps use combinations
  - Case in point: the graphics "pipeline"
    - Pipeline-parallel buffering between stages
    - Programmable stages run data-parallel
    - Task-parallel sharing of unified shader cores

Data Parallelism

- Strengths
  - Easy to get high utilization of throughput architecture
  - Implicit use of SIMD/SIMT
  - Implicit memory latency hiding
- Weaknesses
  - Works best for large, homogeneous problems
  - Work efficiency drops with irregularity
  - Core resources divided amongst all elements

Pipeline Parallelism

- Strengths
  - Copes with variable data amplification
  - Can exploit producer-consumer locality

- Weaknesses
  - Best scheduling strategy workload-dependent
  - No general-purpose tools for current HW

Task Parallelism

- Strengths
  - Scales even with irregular/dynamic problems
  - Viable parallelism approach for global app structure

- Weaknesses
  - No automatic support for latency-hiding
  - Need to explicitly target SIMD width

Summary

- Data-, pipeline- and task-parallelism
  - Three proven approaches to scalability
  - Applicable to many problems in visual computing

- Look for these to surface as we discuss
  - Architectures
  - Tools
  - Algorithms

Questions?

Backup

Many possible syntaxes

Kernel Language

```
kernel void k(
  float* A, float* B,
  float* C) {
  C[id] = A[id] + B[id];
}
…
k<N>(A, B, C);
```

Parallel “Loop”

```
par_for(int i = 0; i < N;
i++)
    C[i] = A[i] + B[i];
```

Array Operations

```
Stream<float> A, B, C;

…

C = A + B;
```

Parallel Functional Map

```
fun k(a, b) = a + b

…

C = par_map(k, A, B)
```

Example syntax

Kernel Language

```
kernel void k(…)
{
 level_2 float sum = 0;
 level_1 float a;

  a = ...;

  atomic_add(&sum, a);
}

...

k<N, M>(A, B, C);
```

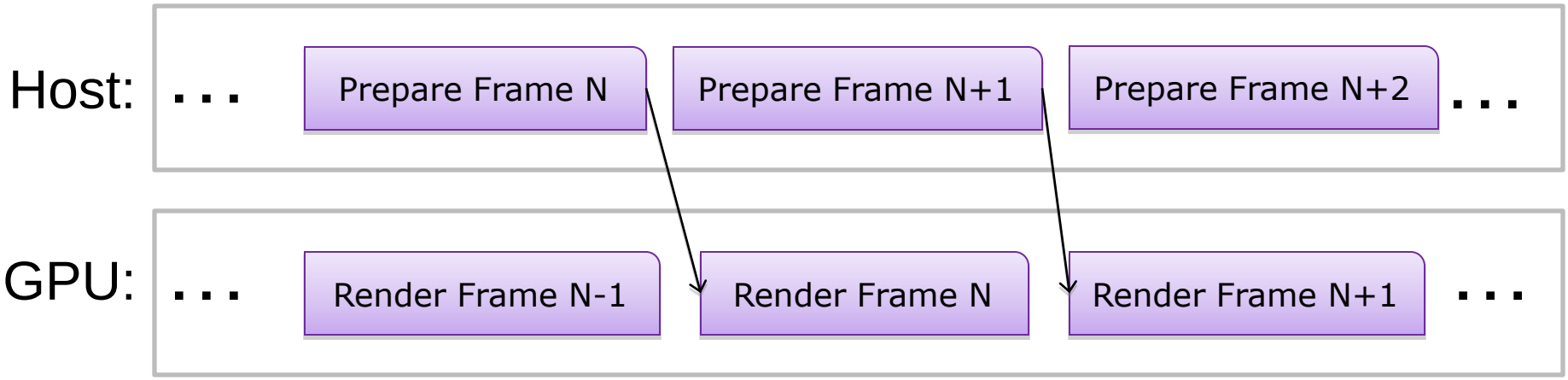
Parallel “Loop”

```
par_for(int i=0; i < N; i++)
{
  float sum = 0;

  par_for(int j=0; j < M;
j++)
  {
    float a;

    a = ...;
    atomic_add(&sum, a);
  }
}
```

Host/GPU pipeline

- Graphics command stream
  - Host packs, GPU consumes in parallel

- Distribute pack work across N host cores
  - Common technique in console graphics
  - Will eventually translate to desktop



Tasks and threads

- Task looks a lot like an OS thread
  - Created with function to execute
  - Waits on a queue to be scheduled to a core
  - May trigger event on completion

- Differences
  - Cooperative, not preemptive scheduling
  - Lightweight create/destroy
  - "Join" often restricted and lightweight