Lecture 23:
Domain-Specific Parallel Programming

CMU 15-418: Parallel Computer Architecture and Programming (Spring 2012)

Acknowledgments: Pat Hanrahan, Hassan Chafi

Announcements

- **List of class final projects**

  `http://www.cs.cmu.edu/~15418/projectlist.html`

- **You are encouraged to keep a log of activities, rants, thinking, findings, on your project web page**
  - It will be interesting for us to read
  - It will come in handy when it comes time to do your writeup
  - Writing clarifies thinking

Course themes:

Designing computer systems that <u>scale</u>

(running faster given more resources)

Designing computer systems that are <u>efficient</u>

(running faster under constraints on resources)

exploiting parallelism in applications
exploiting locality in applications
leveraging HW specialization

Hardware trend: specialization of execution

- **Multiple forms of parallelism**
  - SIMD/vector processing → fine-granularity parallelism: similar execution on different data
  - Multi-threading → mitigate inefficiencies of unpredictable data access
  - Multi-core
  - Multiple node
  - Multiple-server

  Multi-core, Multiple node, Multiple-server → varying scales of coarse-granularity parallelism

- **Heterogeneous execution capability**
  - Programmable, latency-centric (e.g., "CPU-like" cores)
  - Programmable, throughput-optimized (e.g., "GPU-like" cores)
  - Fixed-function, application-specific (e.g., image/video/audio processing)

Motivation: maximize compute capability given constraints on chip area, power

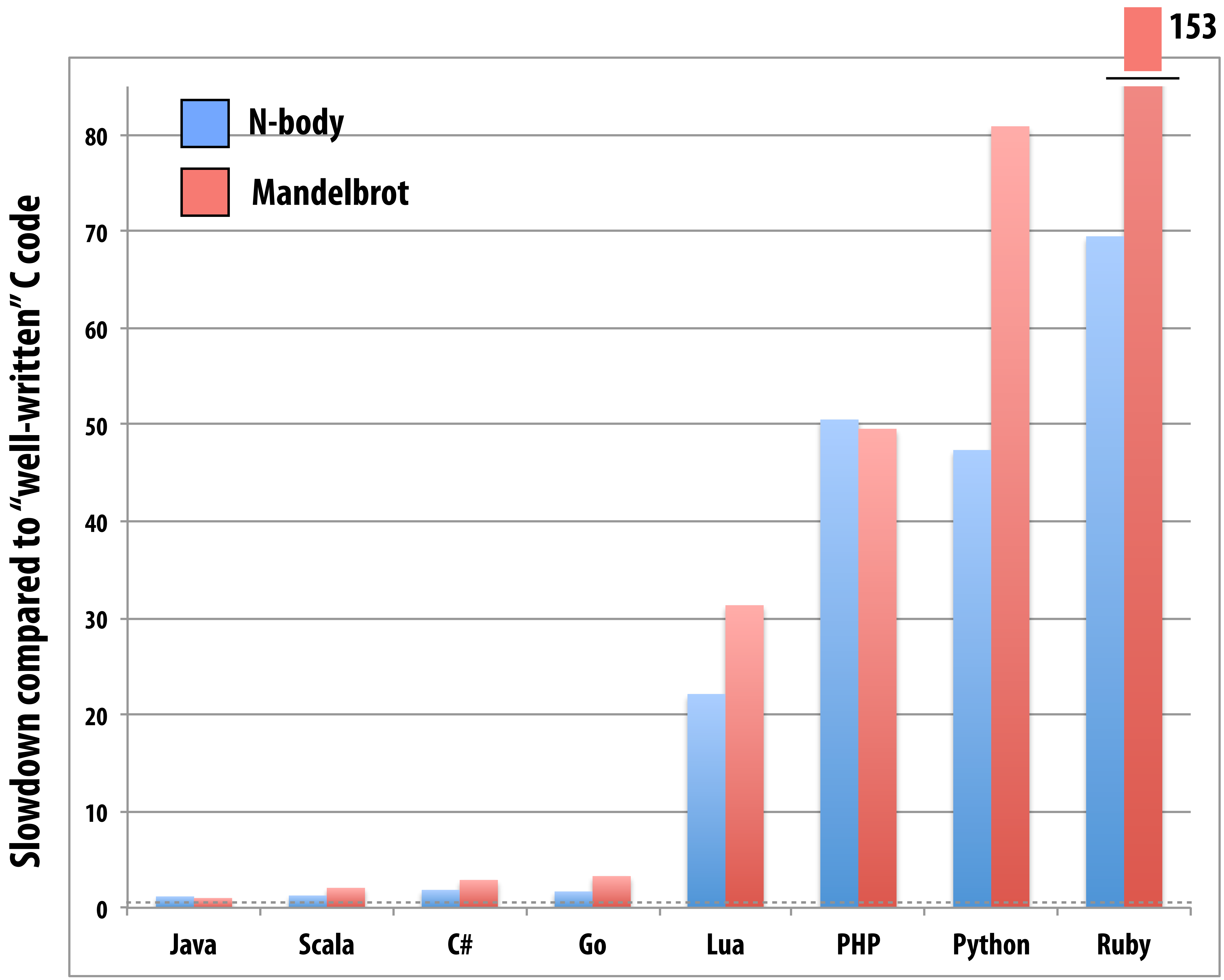
Most software is inefficient

- Consider basic sequential C code (baseline performance)

- Well-written sequential C code: ~ 5-10x faster
- Assembly language program: another small constant factor faster
- Java, Python, PHP, etc. ??

Credit: Pat Hanrahan

Code performance relative to C (single core)



Source: The Computer Language Benchmarks Game: http://shootout.alioth.debian.org

(CMU 15-418, Spring 2012)

Even good C code is inefficient

Recall Assignment 1's Mandelbrot program

For execution on this laptop: quad-core, Intel Core i7, AVX instructions...

Single core, with AVX vector instructions: 5.8x speedup over C implementation

Multi-core + hyper-threading + vector instructions: 21.7x speedup

Conclusion: basic C implementation leaves <u>a lot</u> of performance on the table

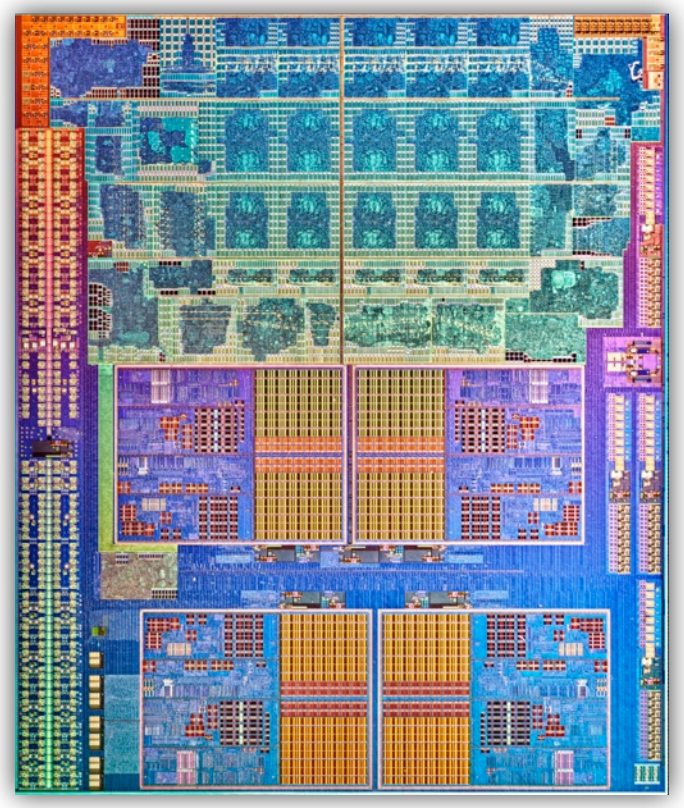
Making efficient use of modern machines is challenging (proof by assignments 2, 3, and 4)

In assignments you only programmed homogeneous parallel environments.
And parallelism in that context was not easy.

GPU only (assignment 2)
Blacklight: CPUs with relatively fast interconnect (assignment 3, 4)
(interesting: no one attempted to utilize SIMD on assignments 3 or 4)

Power-efficient heterogeneous platforms

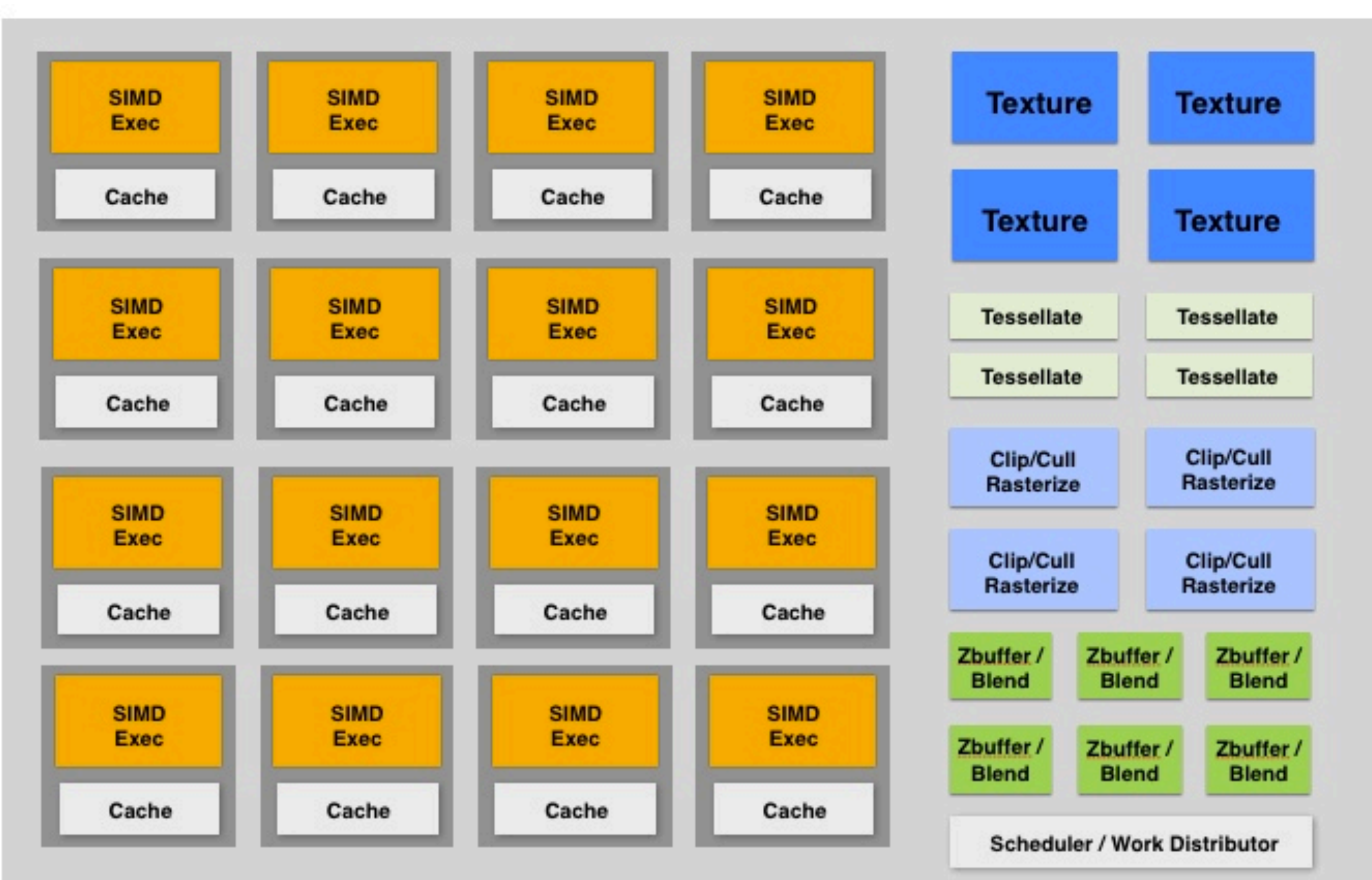
Integrated CPU + GPU

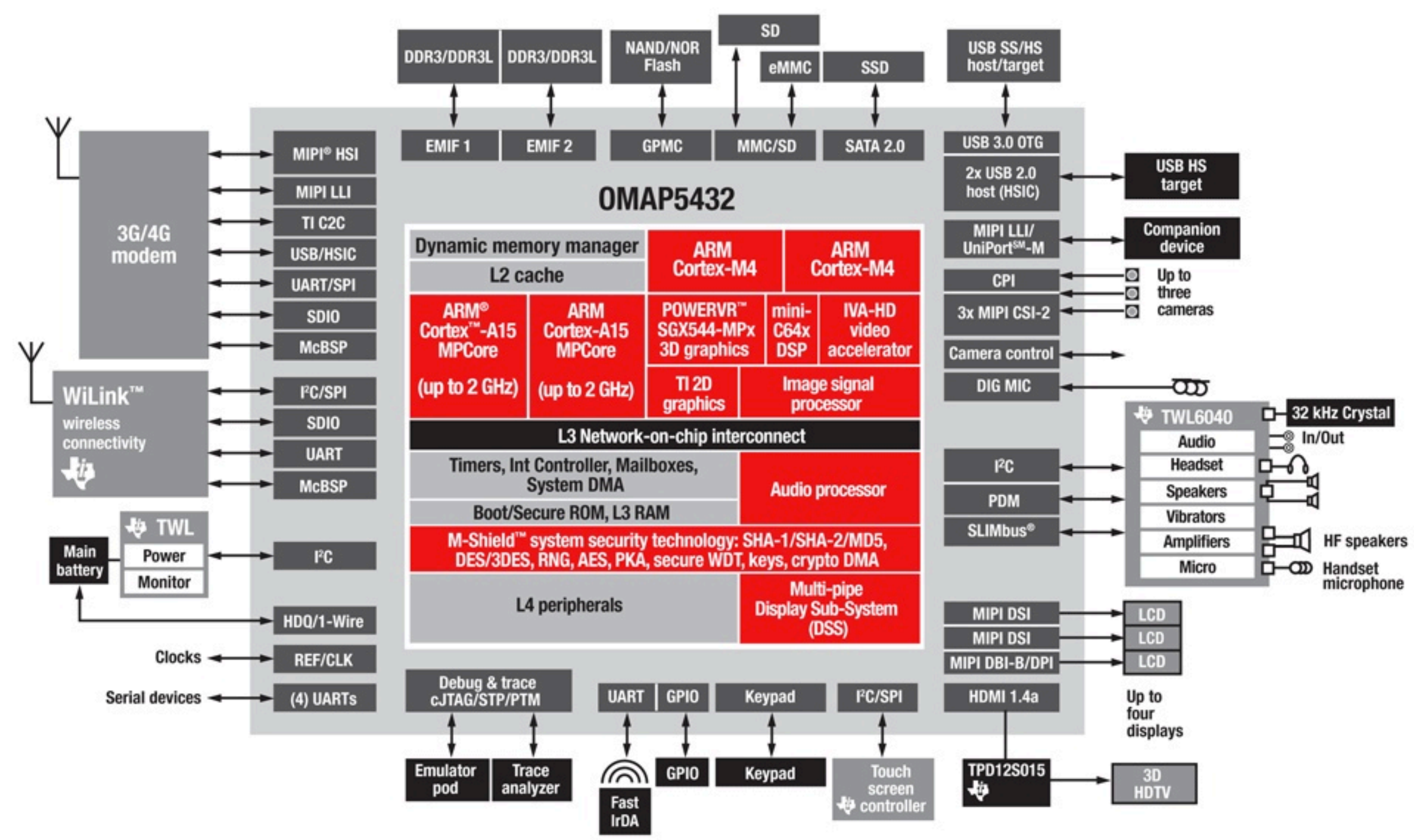


CPU+data-parallel accelerator

GPU:

throughput cores + fixed-function





Mobile system-on-a-chip:
CPU+GPU+media processing

Huge challenge

- Machines with very different performance characteristics
- Worse: different technologies and performance characteristics within the same machine at different scales
    - Within a core: SIMD, multi-threading: fine-granularity sync and comm.
    - Across cores: coherent shared memory via fast on-chip network
    - Hybrid CPU+GPU multi-core: incoherent (potentially) shared memory
    - Across racks: distributed memory, multi-stage network

Variety of programming models to abstract HW

- Machines with very different performance characteristics
- Worse: different technologies and performance characteristics within <u>the same machine</u> at different scales
  - Within a core: SIMD, multi-threading: fine grained sync and comm.
    - Abstractions: SPMD programming (ISPC, Cuda, OpenCL)
  - Across cores: coherent shared memory via fast on-chip network
    - Abstractions: OpenMP shared address space
  - Hybrid CPU+GPU multi-core: incoherent (potentially) shared memory
    - Abstractions: OpenCL, GRAMPS ??
  - Across racks: distributed memory, multi-stage network
    - Abstractions: message passing (MPI, Go channels)

Credit: Pat Hanrahan

Huge challenge

- Machines with very different performance characteristics
- Worse: different performance characteristics within the same machine at different scales
- To be efficient, software must be optimized for HW characteristics
  - Difficult even in the case of one level of one machine **
  - Combinatorial complexity of optimizations when considering a complex machine, or different machines
  - Loss of software portability

** Little success developing automatic tools to identify efficient HW mapping for arbitrary, complex applications

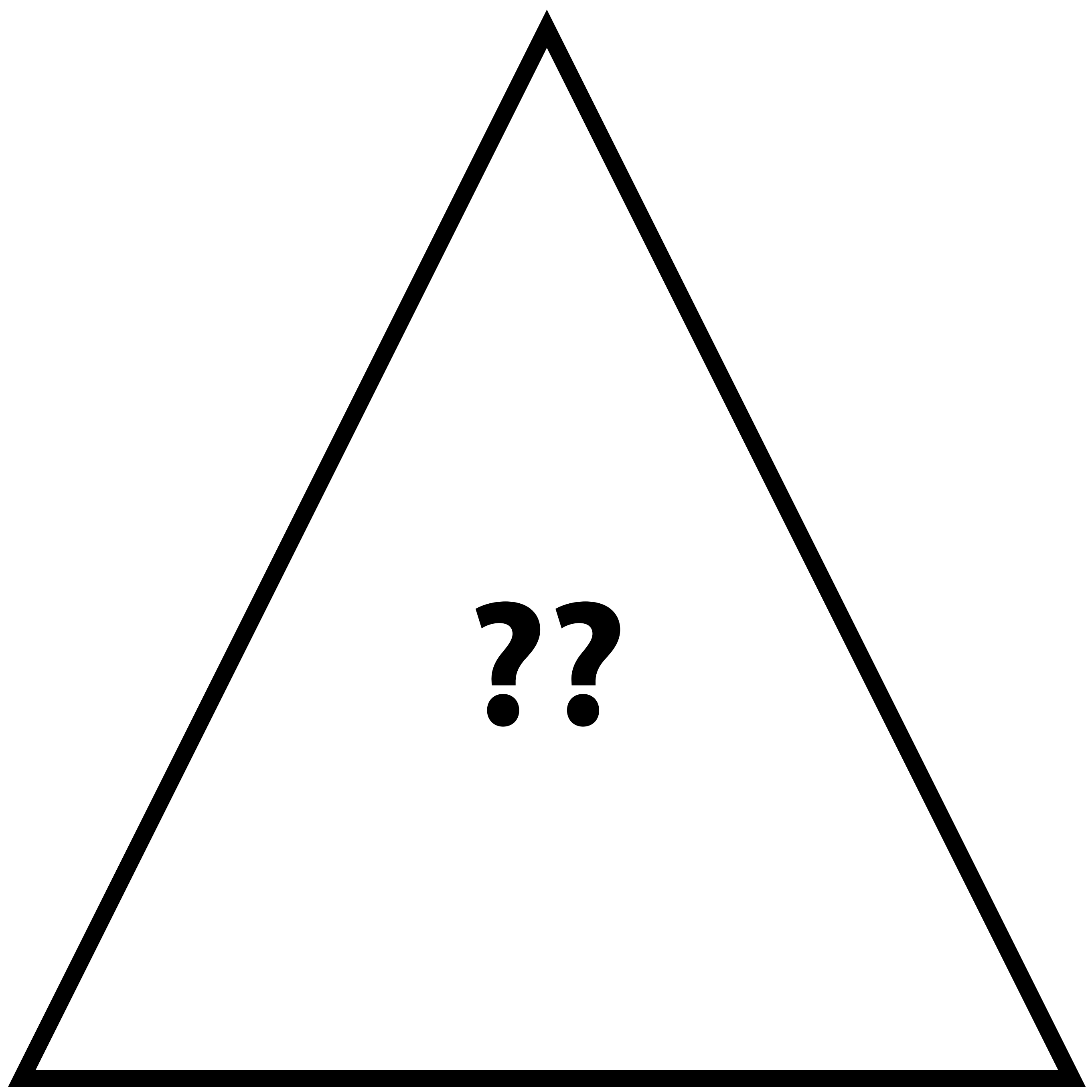
Credit: Pat Hanrahan

Open CS question:

How do we enable programmers to write software that efficiently uses these parallel machines?

The [magical] ideal parallel programming language

High Performance
(software is scalable and efficient)

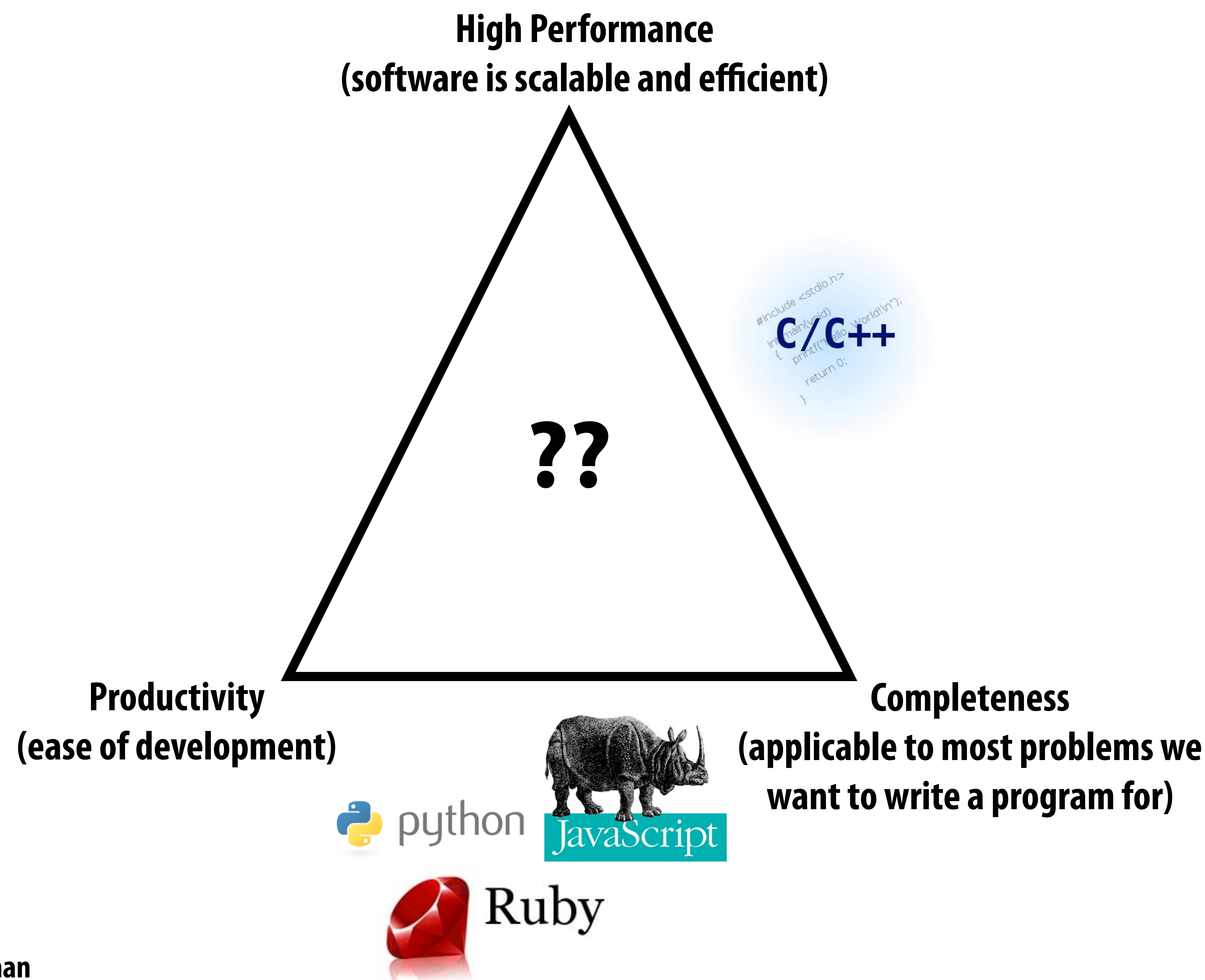


Productivity
(ease of development)

Completeness
(applicable to most problems we want to write a program for)

Credit: Pat Hanrahan

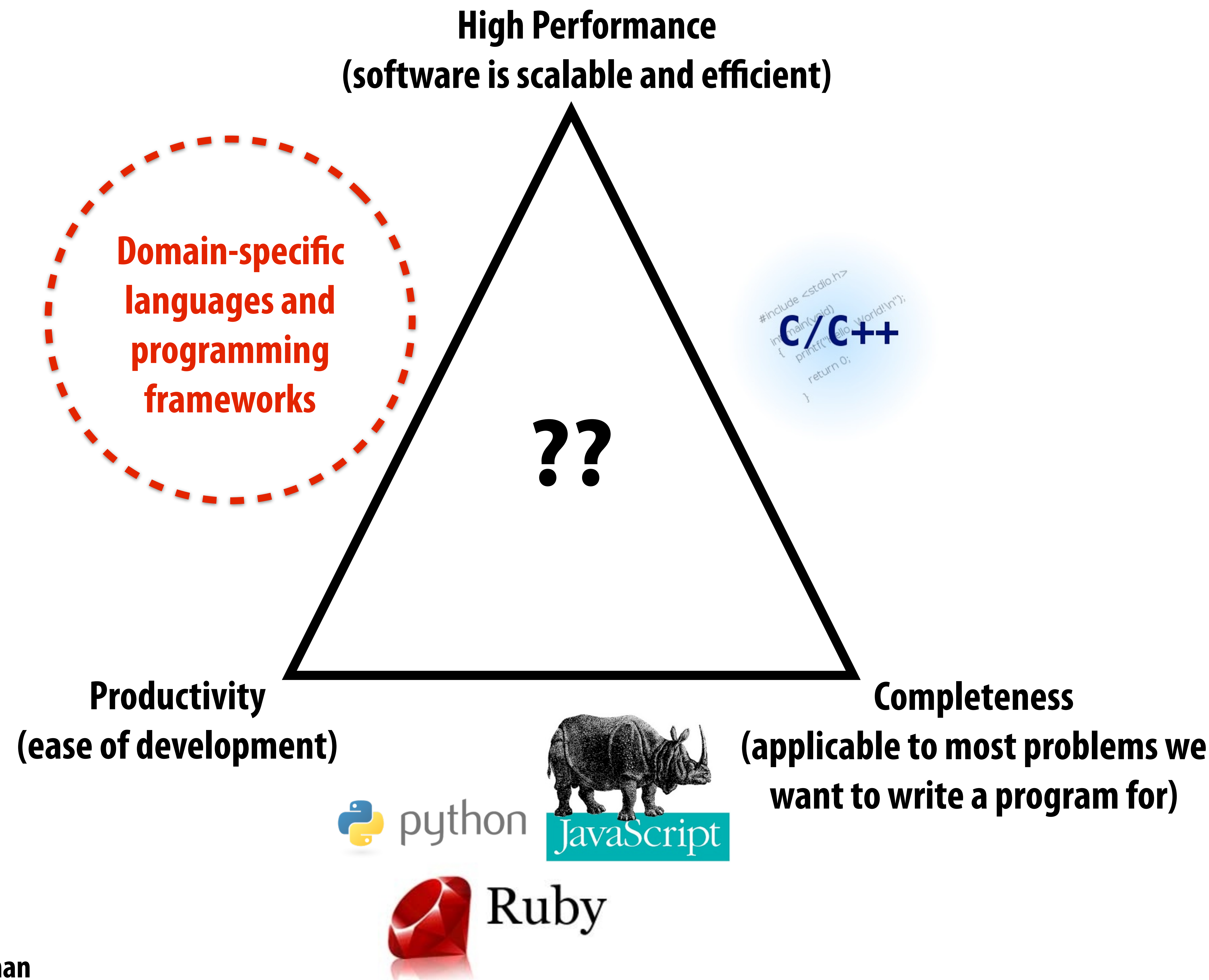
Successful programming languages



Credit: Pat Hanrahan

Growing interest in domain-specific programming systems

To realize high performance and productivity: willing to sacrifice completeness



Credit: Pat Hanrahan

Domain-specific programming systems

- Main idea: raise level of abstraction
- Introduce high-level programming primitives specific to domain
  - Productive: intuitive to use, portable across machines, primitives correspond to behaviors frequently used to solve problems in domain
  - Performant: system uses domain knowledge to provide efficient, optimized implementation(s)
    - Given a machine: what algorithms to use, parallelization strategies to employ
    - Optimization goes beyond efficient software mapping: HW platform can be optimized to the abstractions as well
- Cost: loss of generality/completeness

Two domain-specific programming examples

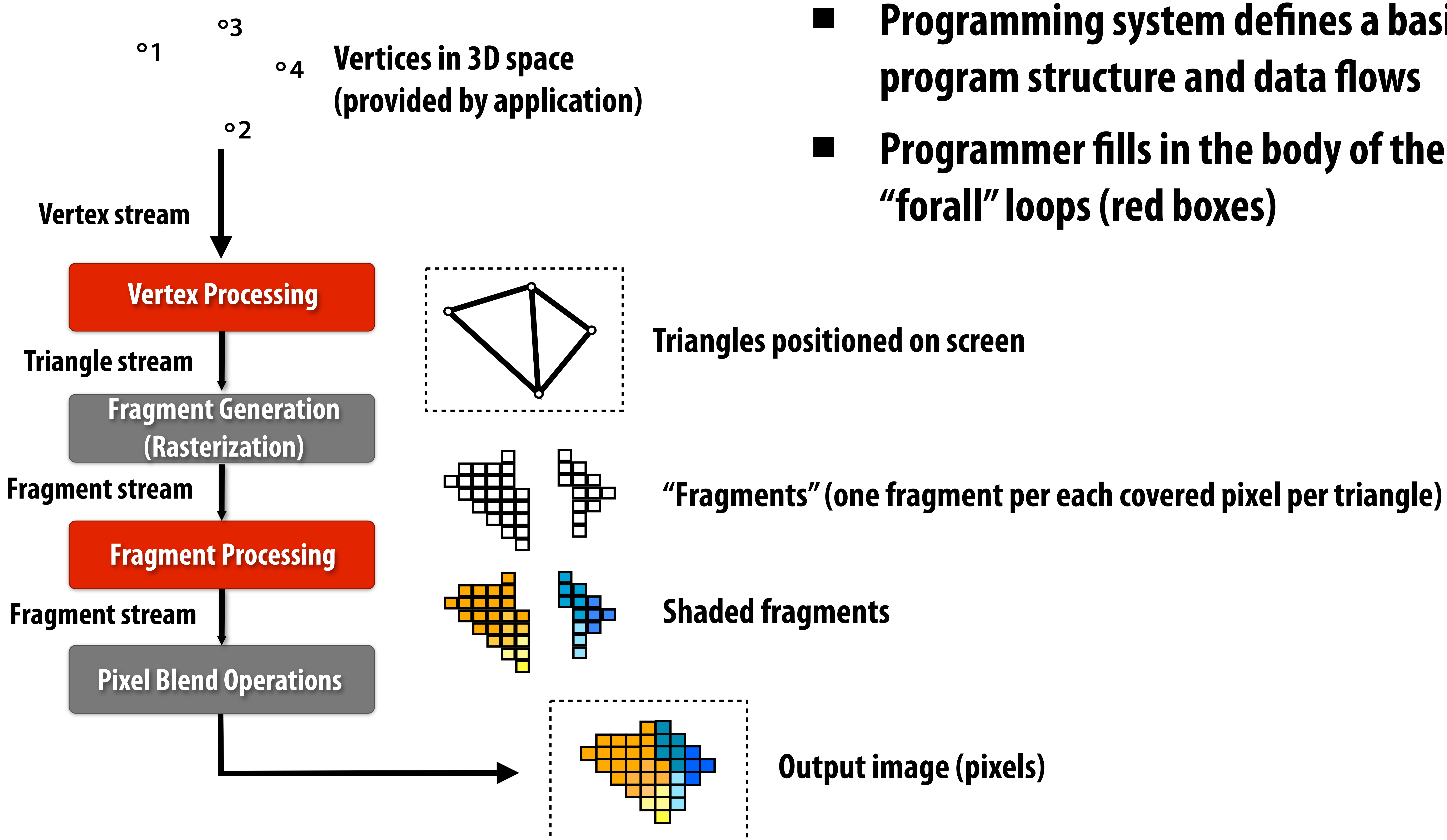
1. Graphics: OpenGL

2. Scientific computing: Liszt

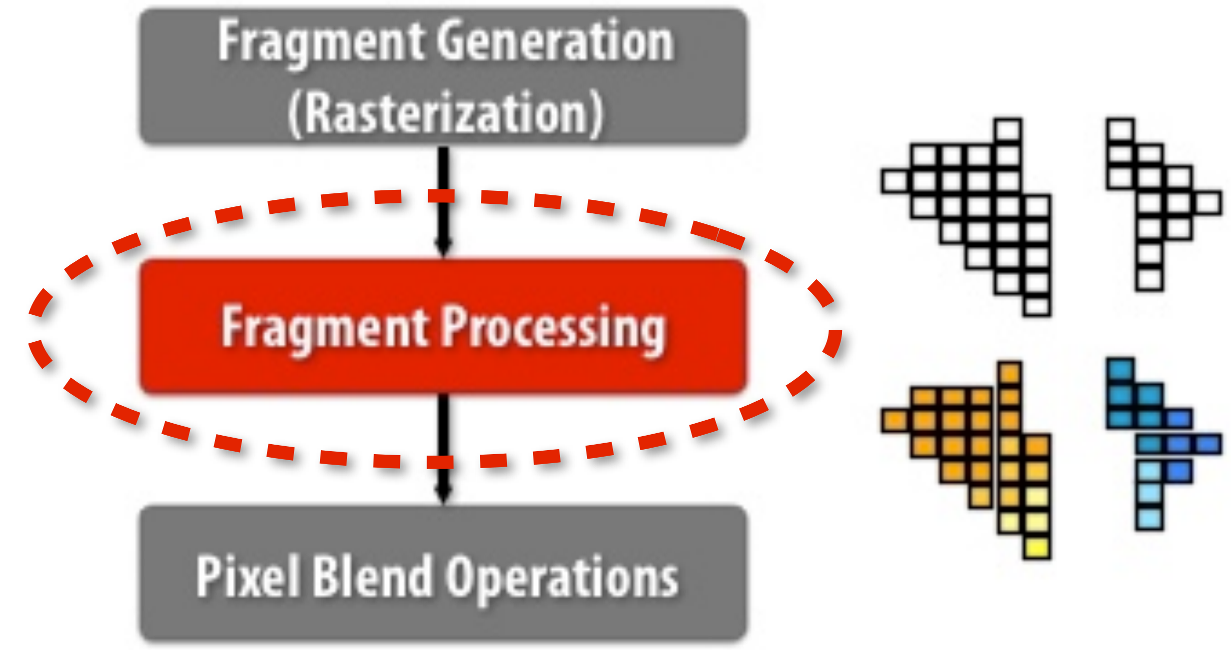
Example 1:
OpenGL: a programming system for real-time rendering

OpenGL graphics pipeline

- Key abstraction: graphics pipeline
- Programming system defines a basic program structure and data flows
- Programmer fills in the body of the "forall" loops (red boxes)



Fragment "shader" program



HLSL shader program: defines behavior of fragment processing stage
Executes once per pixel covered by each triangle

Input: a "fragment": information about the triangle at the pixel
Output: RGBA color (float4 datatype)

```
sampler mySamp;
Texture2D<float3> myTex;
float3 lightDir;

float4 diffuseShader(float3 norm, float2 uv)
{
  float3 kd;
  kd = myTex.sample(mySamp, uv);
  kd *= clamp(dot(lightDir, norm), 0.0, 1.0);
  return float4(kd, 1.0);
}
```

Productivity:
- SPMD program: no explicit parallelism
- Programmer writes no loops. Code is implicitly a loop body
- Code runs independently for each input fragment (no loops = impossible to express a loop dependency)

Performance:
- SPMD program compiles to wide SIMD processing on GPU
- Work for many fragments dynamically balanced onto GPU cores

- Performance Portability:
  - Scales to GPUs with different # of cores
  - SPMD abstraction compiles to different SIMD widths (NVIDIA=32, AMD=64, Intel=?)

Special language primitive for texture mapping

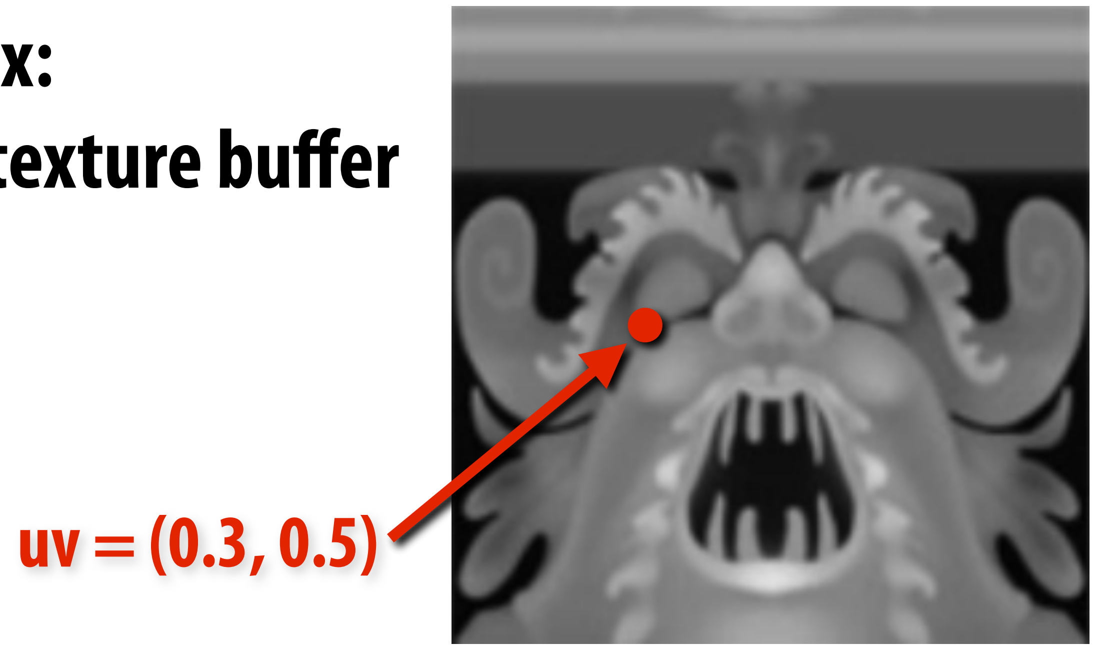
```
sampler mySamp;
Texture2D<float3> myTex;
float3 lightDir;

float4 diffuseShader(float3 norm, float2 uv)
{
  float3 kd;
  kd = myTex.sample(mySamp, uv);
  kd *= clamp(dot(lightDir, norm), 0.0, 1.0);
  return float4(kd, 1.0);
}
```

Productivity:

- **Intuitive: abstraction presents a texture lookup like an array access with a 2D floating point index.**

myTex:
NxN texture buffer



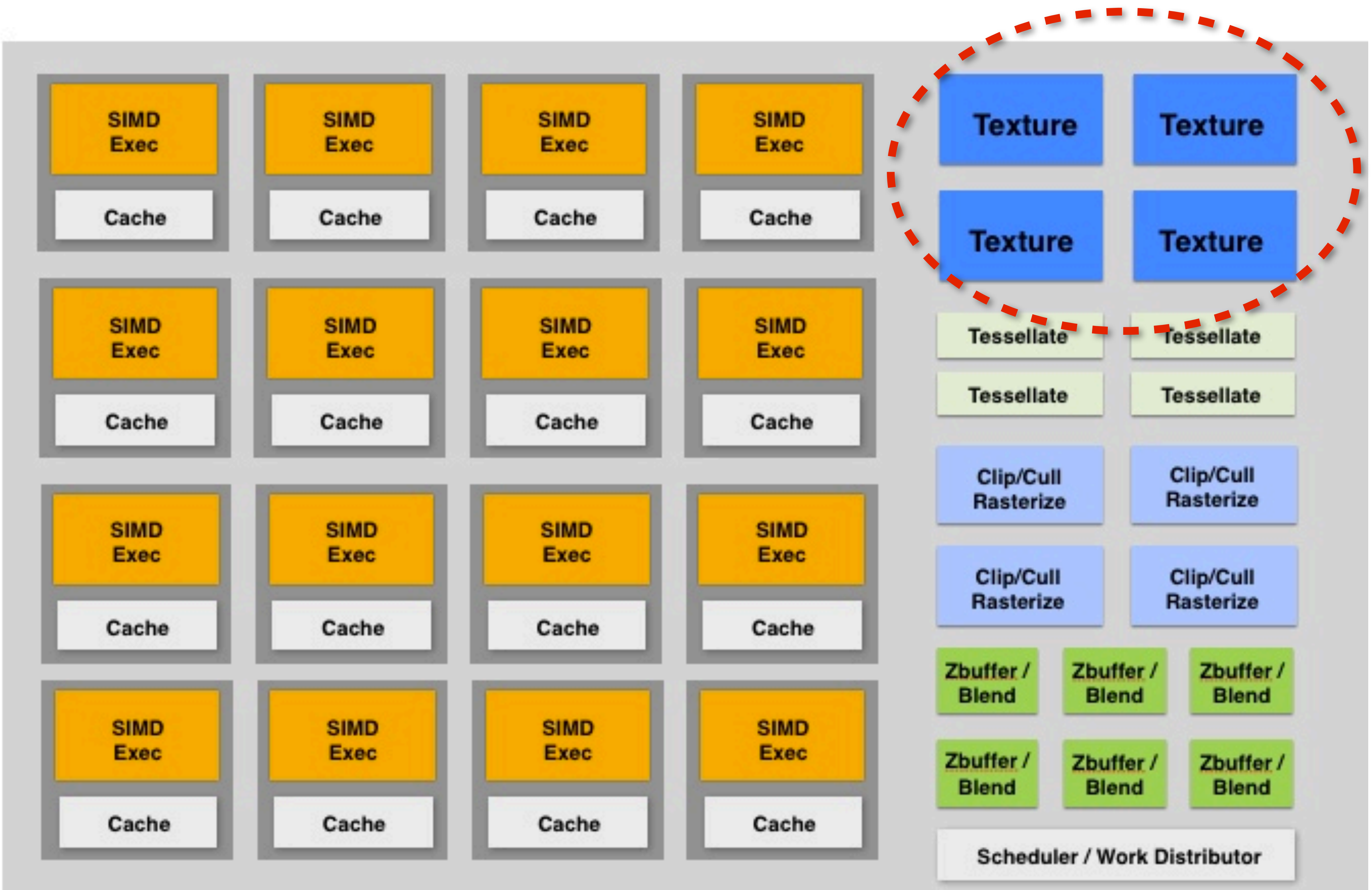
Result of mapping texture onto plane, viewed with perspective

Texture mapping is expensive (performance critical)

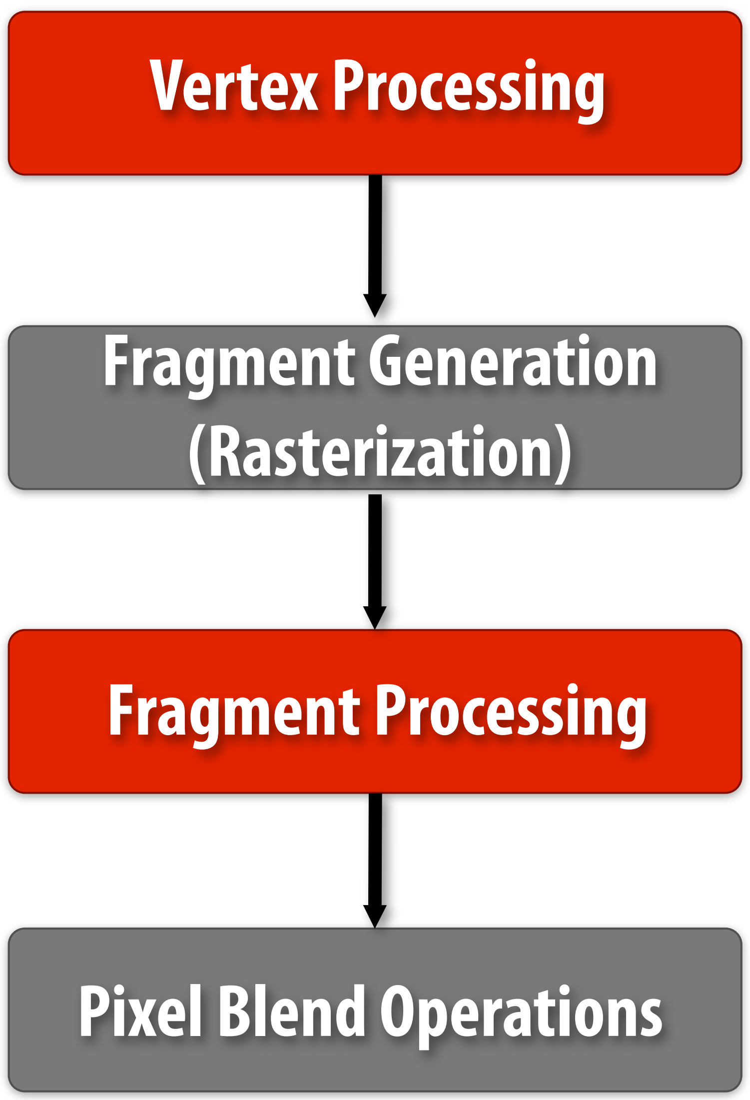
- Texture mapping is more than an array lookup (see 15-462)
  - ~50 instructions, multiple conditionals
  - Read at least 8 texture values
    - Unpredictable data access, little temporal locality
- Typical shader performs multiple texture lookups
- Texture mapping is one of the most computationally demanding AND bandwidth intensive aspects of the graphics pipeline
  - Resources for texturing must run near 100% efficiency
  - Not surprising it is encapsulated in its own primitive

Performance: texture mapping

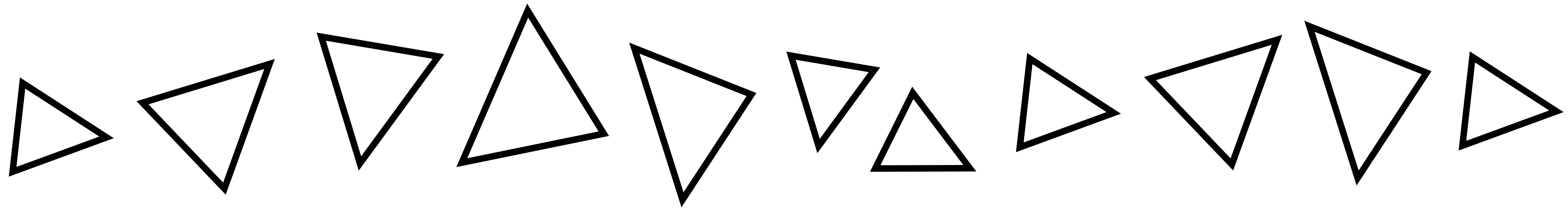
- Highly multi-threaded cores hide latency of memory access
(texture primitive = location of long mem. stalls explicit in programming model)
- Fixed-function HW to perform texture mapping math
- Special-cache designs to capture reuse, exploit read-only access to texture data



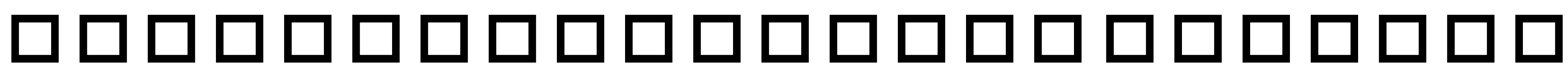
Performance: global application orchestration



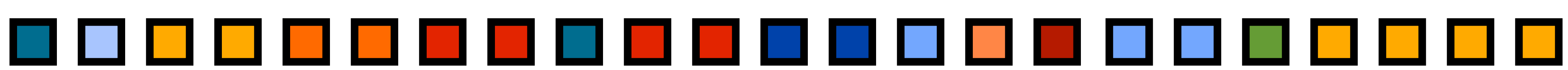
Parallel work:



Hundreds of thousands of triangles



Millions of fragments to shade



Millions of shaded fragments to blend into output image

Efficiently scheduling all this parallel work onto the GPU's pool of resources, while respecting the ordering requirements of the programming model, is challenging.

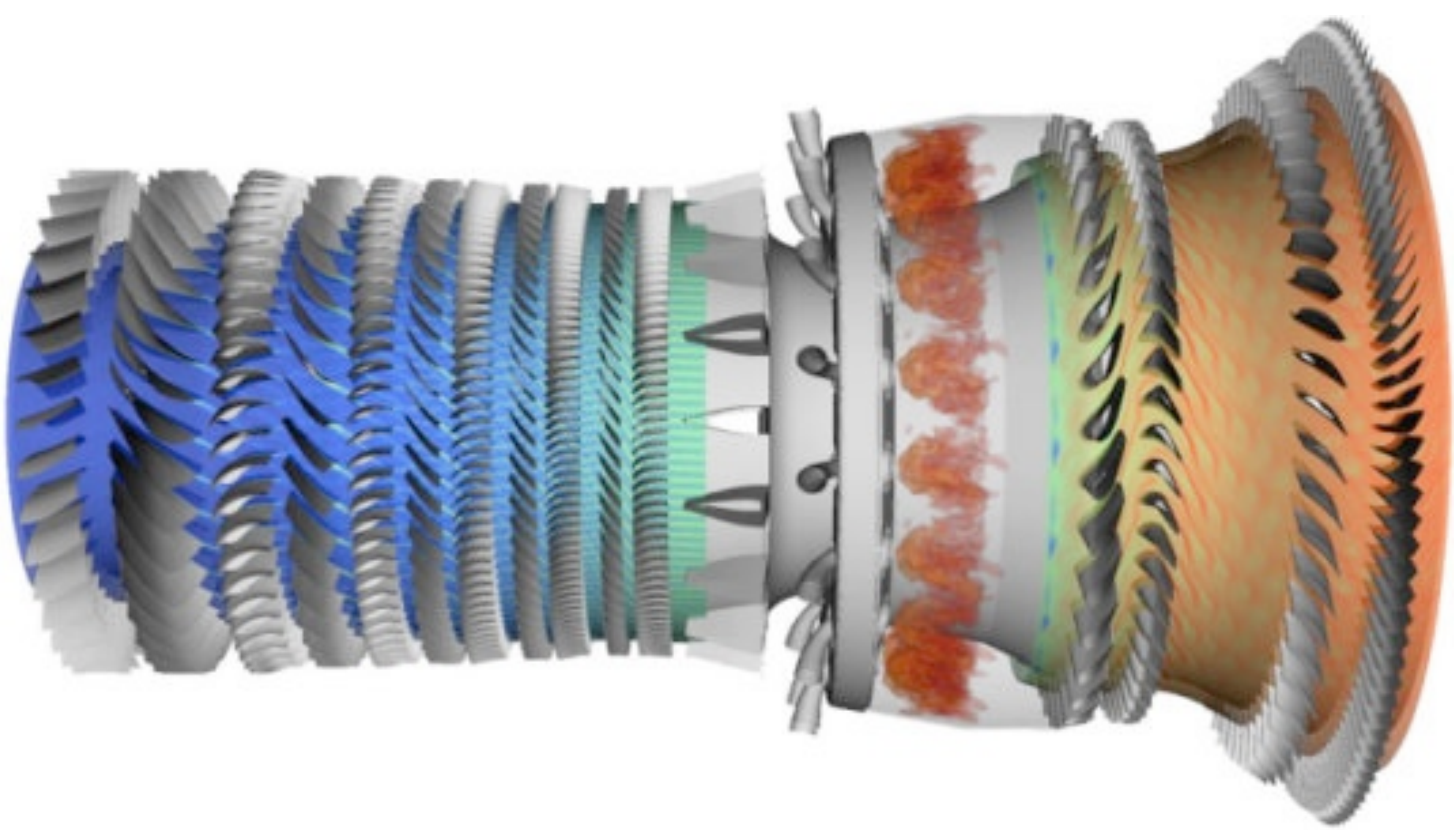
Each GPU vendor uses it's own custom strategy.

OpenGL summary

- **Productivity:**
  - High-level, intuitive abstractions (taught to undergrads in intro graphics class)
  - Application implements kernels for triangles, vertices, and fragments
  - Specific primitives for key functions like texture mapping
- **Portability**
  - Runs across wide range of GPUs: low-end integrated, high-end discrete, mobile
  - Has allowed significant hardware innovation without impacting programmer
- **High-Performance**
  - Abstractions designed to map efficiently to hardware
    (proposed new features disallowed if they do not!)
  - Encapsulating expensive operations as unique pipeline stages or built-in functions facilitates fixed-function implementations (texture, rasterization, frame-buffer blend)
  - Utilize domain-knowledge in optimizing performance / mapping to hardware
    - Skip unnecessary work, e.g., if a triangle it is determined to be behind another, don't generate and shade its fragments
    - Non-overlapping fragments are independent despite ordering constraint
    - Interstage queues/buffers are sized based on expected triangle sizes
    - Use pipeline structure to make good scheduling decisions, set work priorities

Example 2:
Lizst: a language for solving PDE's on meshes

See [DeVito et al. SC11, SciDac '11]



Slide credit for this section of lecture:
Pat Hanrahan, Stanford University

http://liszt.stanford.edu/

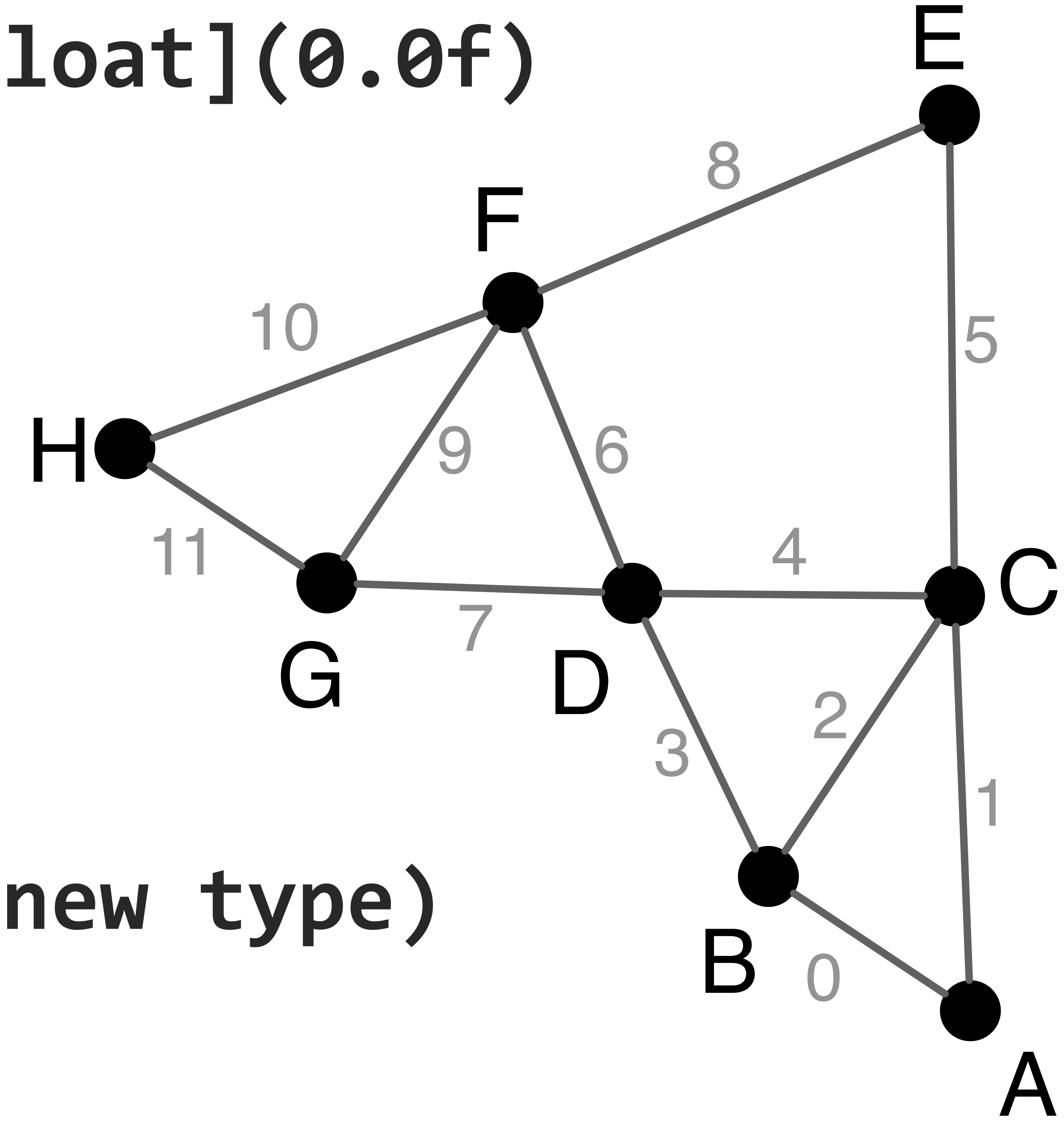
Fields on unstructured meshes

Fields

Mesh Entity

```
val Position = FieldWithLabel[Vertex,Float3](“position”)

val Temperature = FieldWithConst[Vertex,Float](0.0f)
val Flux = FieldWithConst[Vertex,Float](0.0f)
val JacobiStep = FieldWithConst[Vertex,Float](0.0f)
```




```
Notes:
Fields are a higher-kinded type
(special function that maps a type to a new type)
```

Explicit algorithm: heat conduction on grid

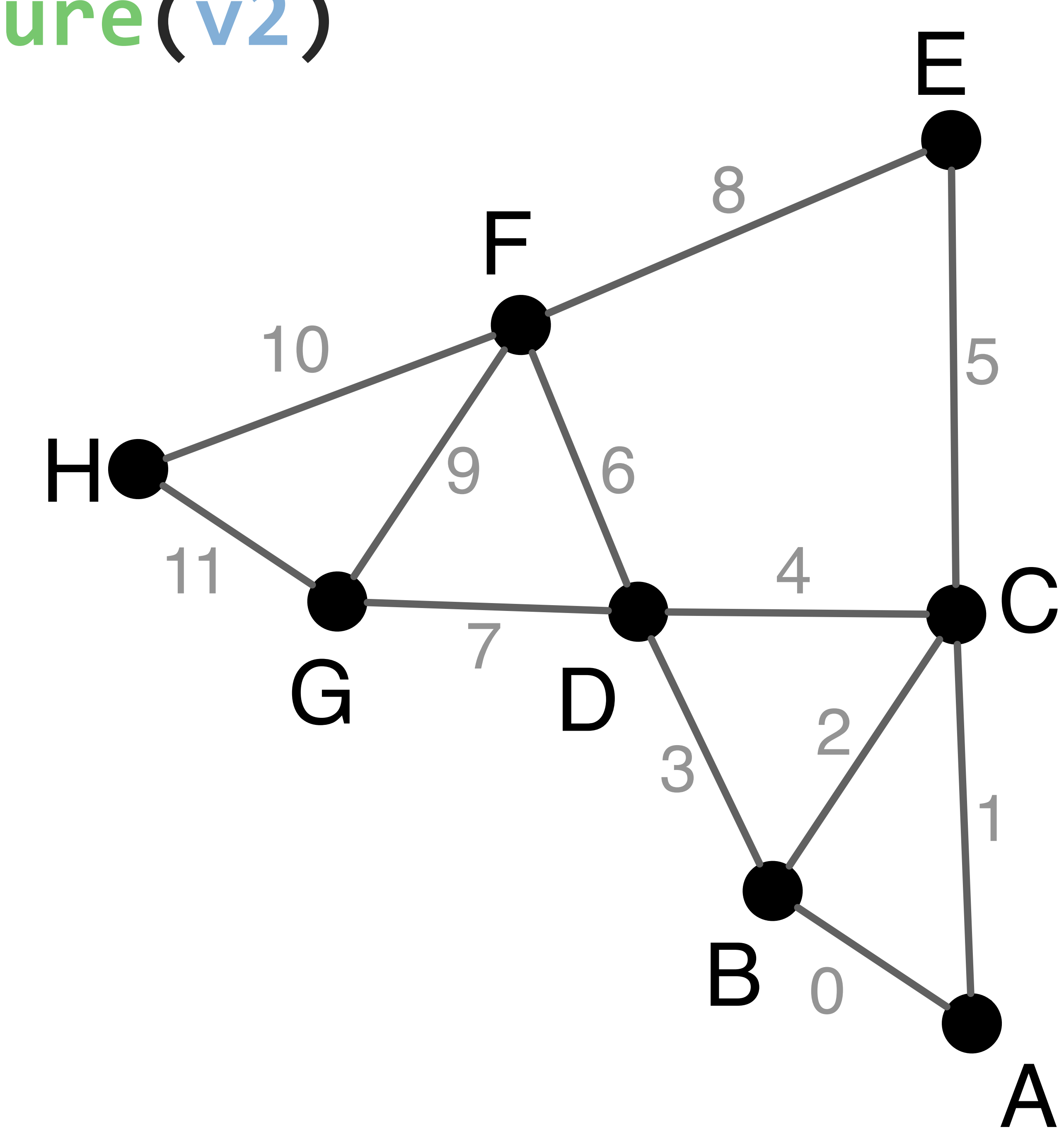
```
var i = 0;
while ( i < 1000 ) {
  Flux(vertices(mesh)) = 0.f;
  JacobiStep(vertices(mesh)) = 0.f;
  for (e <- edges(mesh)) {
    val v1 = head(e)
    val v2 = tail(e)
    val dP = Position(v1) - Position(v2)
    val dT = Temperature(v1) - Temperature(v2)
    val step = 1.0f/(length(dP))
    Flux(v1) += dT*step
    Flux(v2) -= dT*step
    JacobiStep(v1) += step
    JacobiStep(v2) += step
  }
  i += 1
}
```

Fields

Mesh

Topology Functions

Iteration over Set

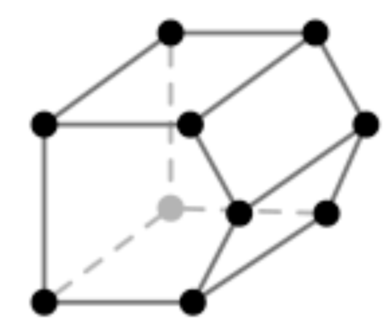


Liszt topological operators

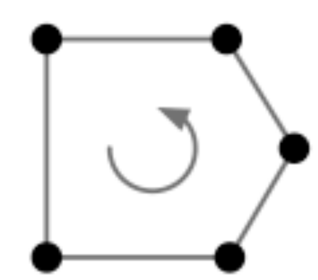
```
BoundarySet[1][ME <: MeshElement](name : String) : Set[ME]
vertices(e : Mesh) : Set[Vertex]
cells(e : Mesh) : Set[Cell]
edges(e : Mesh) : Set[Edge]
faces(e : Mesh) : Set[Face]
```

```
vertices(e : Vertex) : Set[Vertex]
cells(e : Vertex) : Set[Cell]
edges(e : Vertex) : Set[Edge]
faces(e : Vertex) : Set[Face]
```

```
vertices(e : Edge) : Set[Vertex]
facesCCW[2](e : Edge) : Set[Face]
cells(e : Edge) : Set[Cell]
head(e : Edge) : Vertex
tail(e : Edge) : Vertex
flip[4](e : Edge) : Edge
towards[5](e : Edge, t : Vertex) : Edge
```



```
cells(e : Cell) : Set[Cell]
vertices(e : Cell) : Set[Vertex]
faces(e : Cell) : Set[Face]
edges(e : Cell) : Set[Edge]
```



```
cells(e : Face) : Set[Cell]
edgesCCW[2](e : Face) : Set[Edge]
vertices(e : Face) : Set[Vertex]
inside[3](e : Face) : Cell
outside[3](e : Face) : Cell
flip[4](e : Face) : Face
towards[5](e : Face,t : Cell) : Face
```

Liszt programming

- Liszt program describes operations on fields of abstract mesh representation
- Application specifies type of mesh (regular, irregular) and its topology
- Mesh representation is chosen by Liszt
  - Based on mesh type, program behavior, and machine

Compiling to parallel computers

Recall challenges you have faced in your assignments

1. Identify parallelism
2. Identify data locality
3. Reason about required synchronization

Key: determining program dependencies

1. **Identify parallelism**
   - Absence of dependencies implies can be executed in parallel

2. **Identify data locality**
   - Partition data based on dependencies (localize dependent computations for faster synchronization)

3. **Reason about required synchronization**
   - Sync. needed to respect existing dependencies (must wait until values a computation depends on are known)

But in general programs, compilers are unable to infer dependencies at global scale: a[i] = b[f(i)]   (must execute f(i) to know dependency)

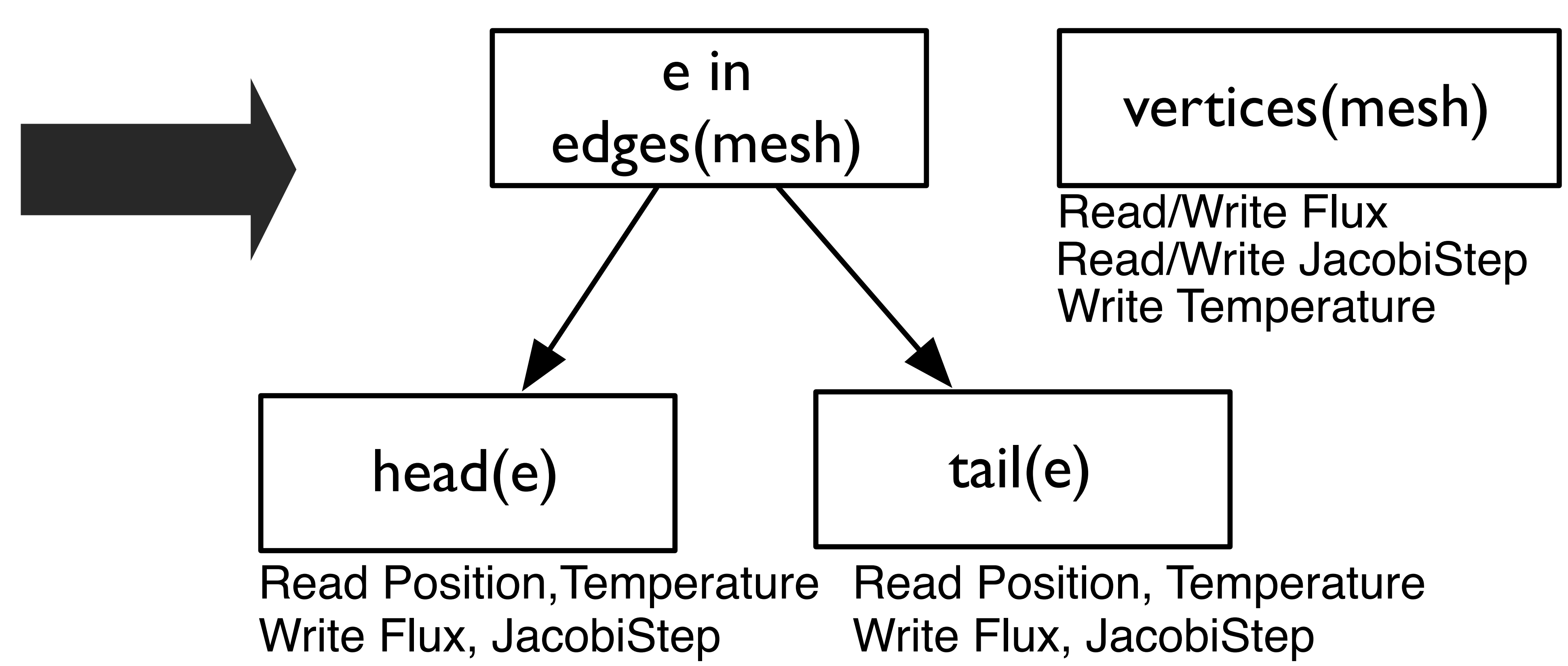
Liszt is constrained to allow dependency analysis

Inferring stencils: ("stencil" = mesh elements accessed in iteration of loop = dependencies for the iteration)

Statically analyze code to find stencil of each top-level for loop

- Extract nested mesh element reads
- Extract field operations

```
for (e <- edges(mesh)) {
  val v1 = head(e)
  val v2 = tail(e)
  val dP = Position(v1) - Position(v2)
  val dT = Temperature(v1) - Temperature(v2)
  val step = 1.0f/(length(dP))
  Flux(v1) += dT*step
  Flux(v2) -= dT*step
  JacobiStep(v1) += step
  JacobiStep(v2) += step
}
...
```



Restrict language for dependency analysis

"Language Restrictions"

- Mesh elements only accessed through built-in topological functions:

```
cells(mesh), …
```

- Single static assignment:

```
val v1 = head(e)
```

- Data in Fields can only be accessed using mesh elements:

```
Pressure(v)
```

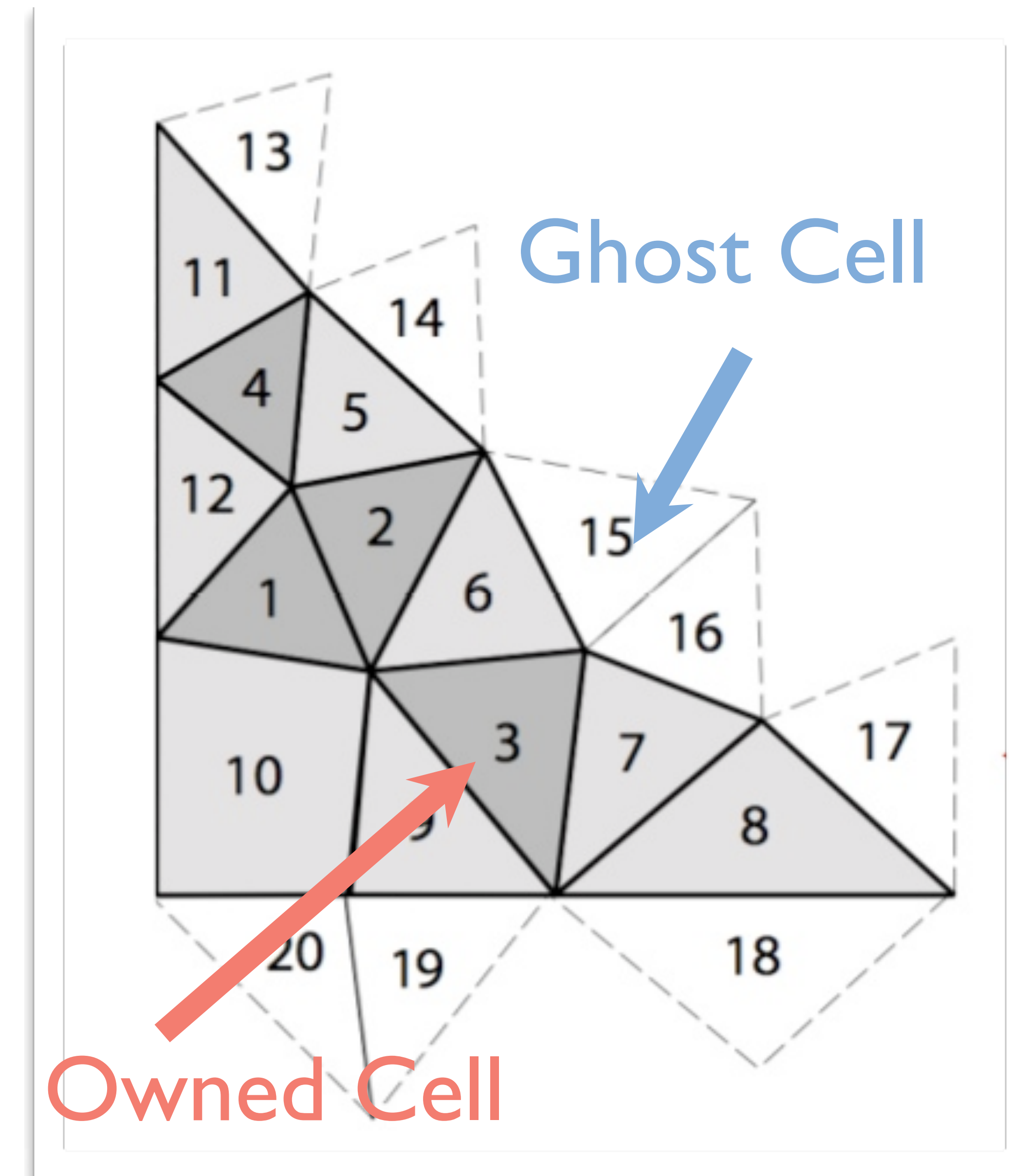
- No recursive functions

Allows compiler to automatically infer stencil

Portable parallelism: use dependencies to implement different parallel execution strategies

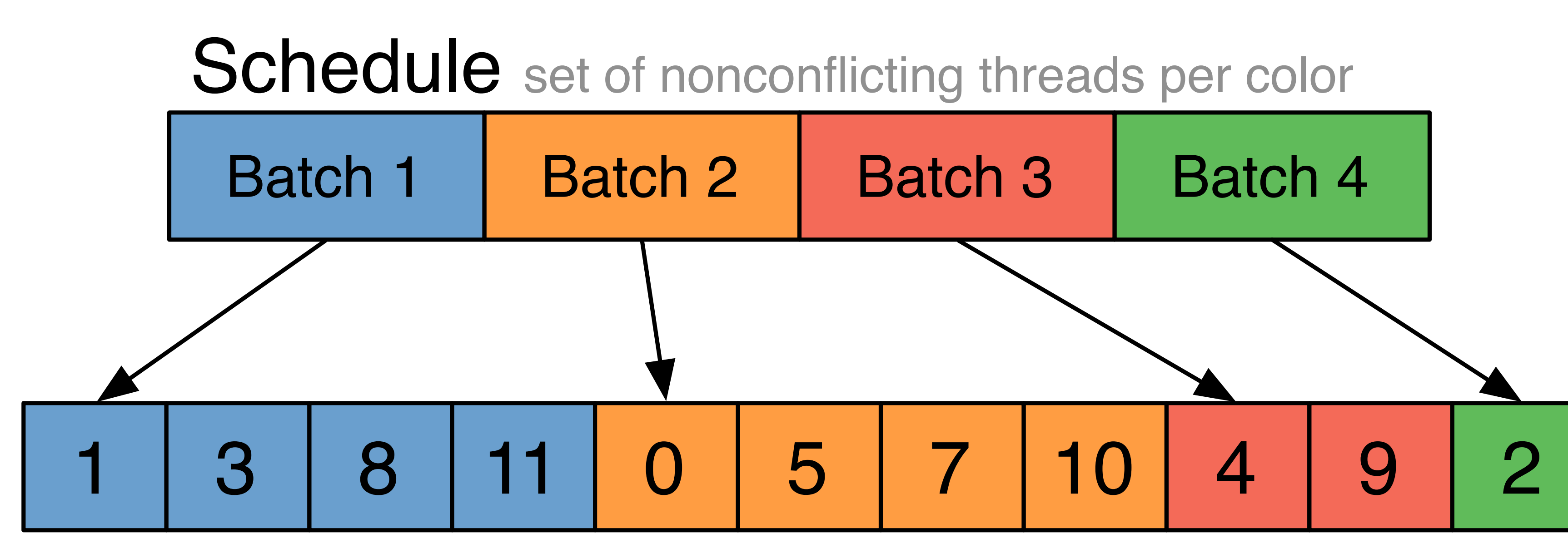
Partitioning

- Assign partition to each computational unit
- Use ghost elements to coordinate boundary communication.



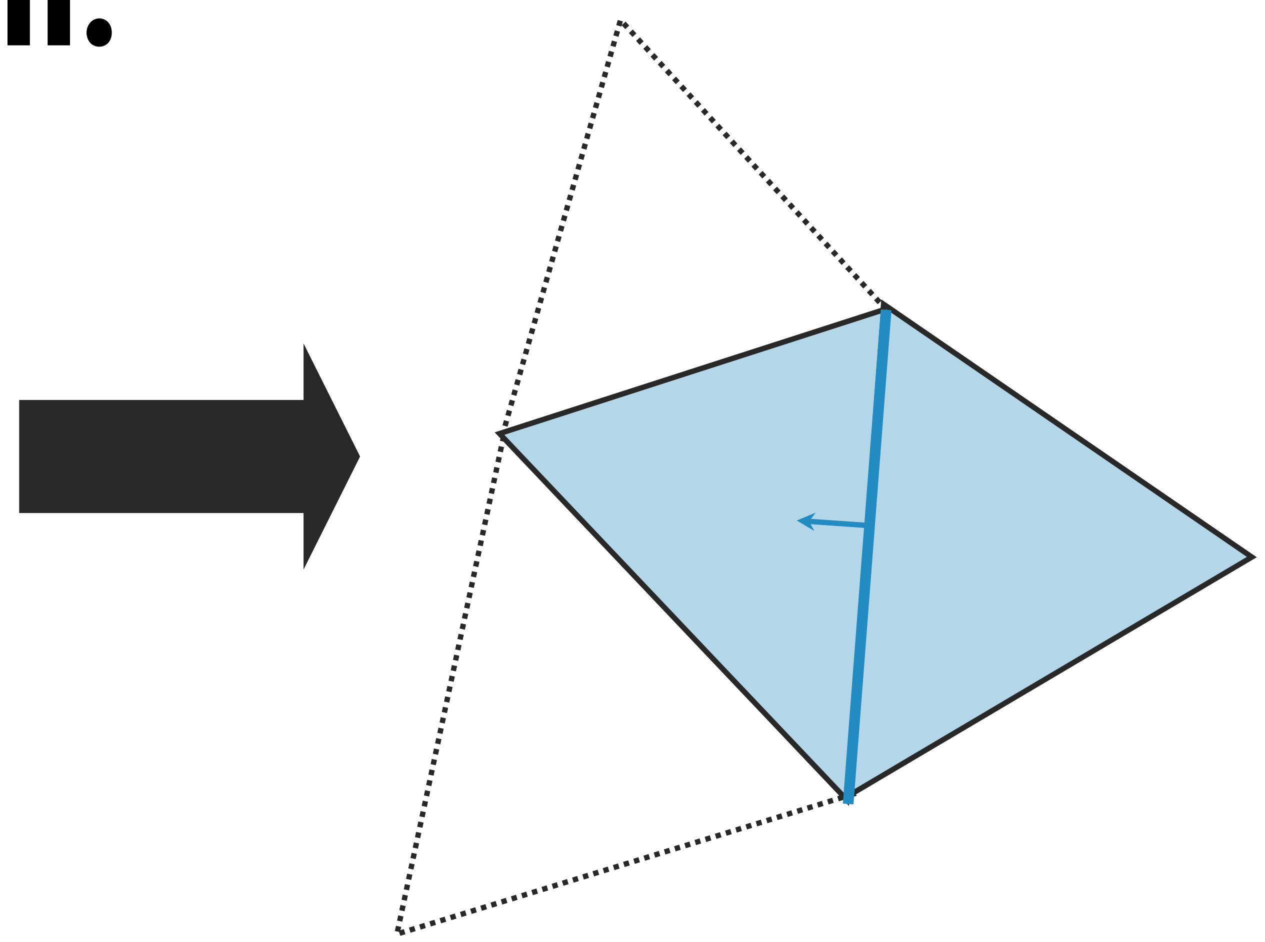
Coloring

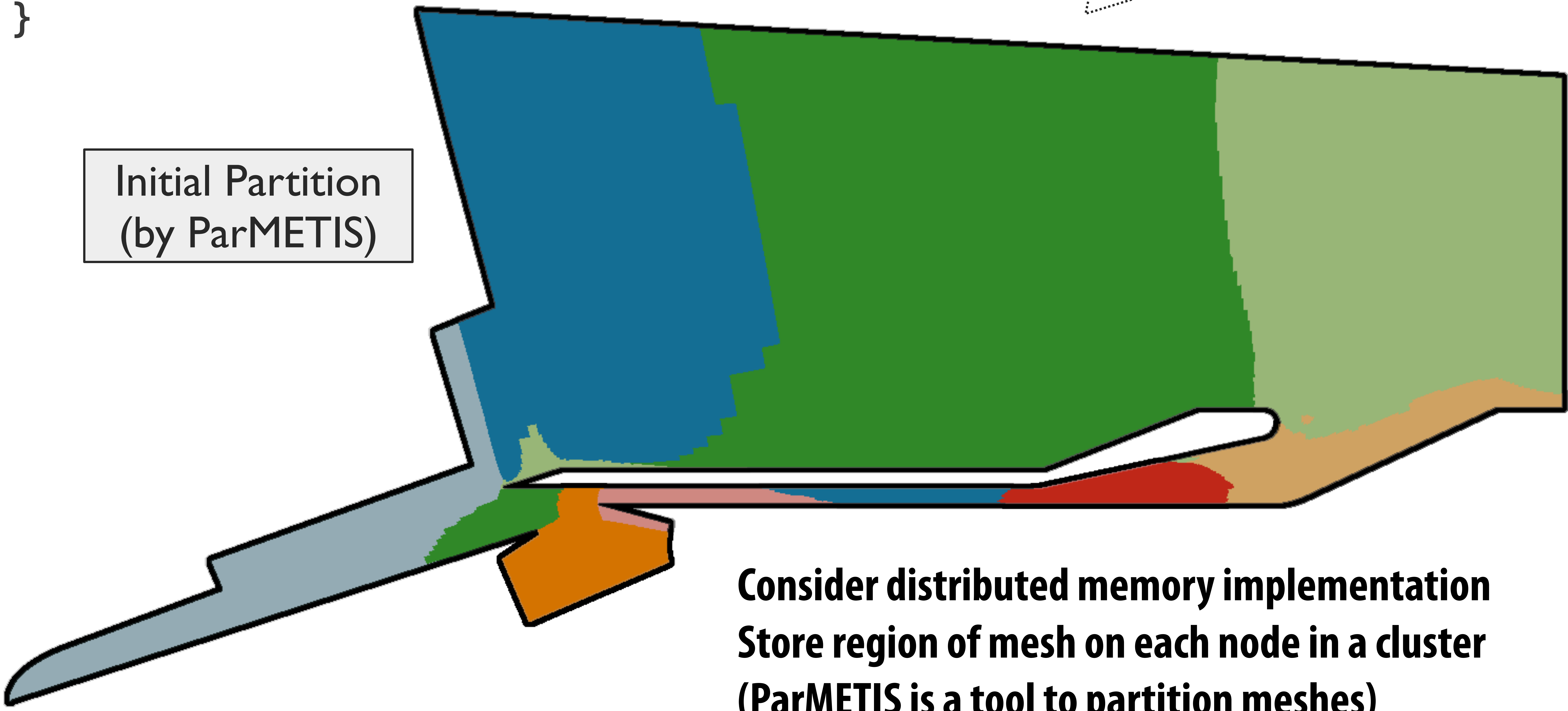
- Calculate interference between work items on domain
- Schedule work-items into non-interfering batches



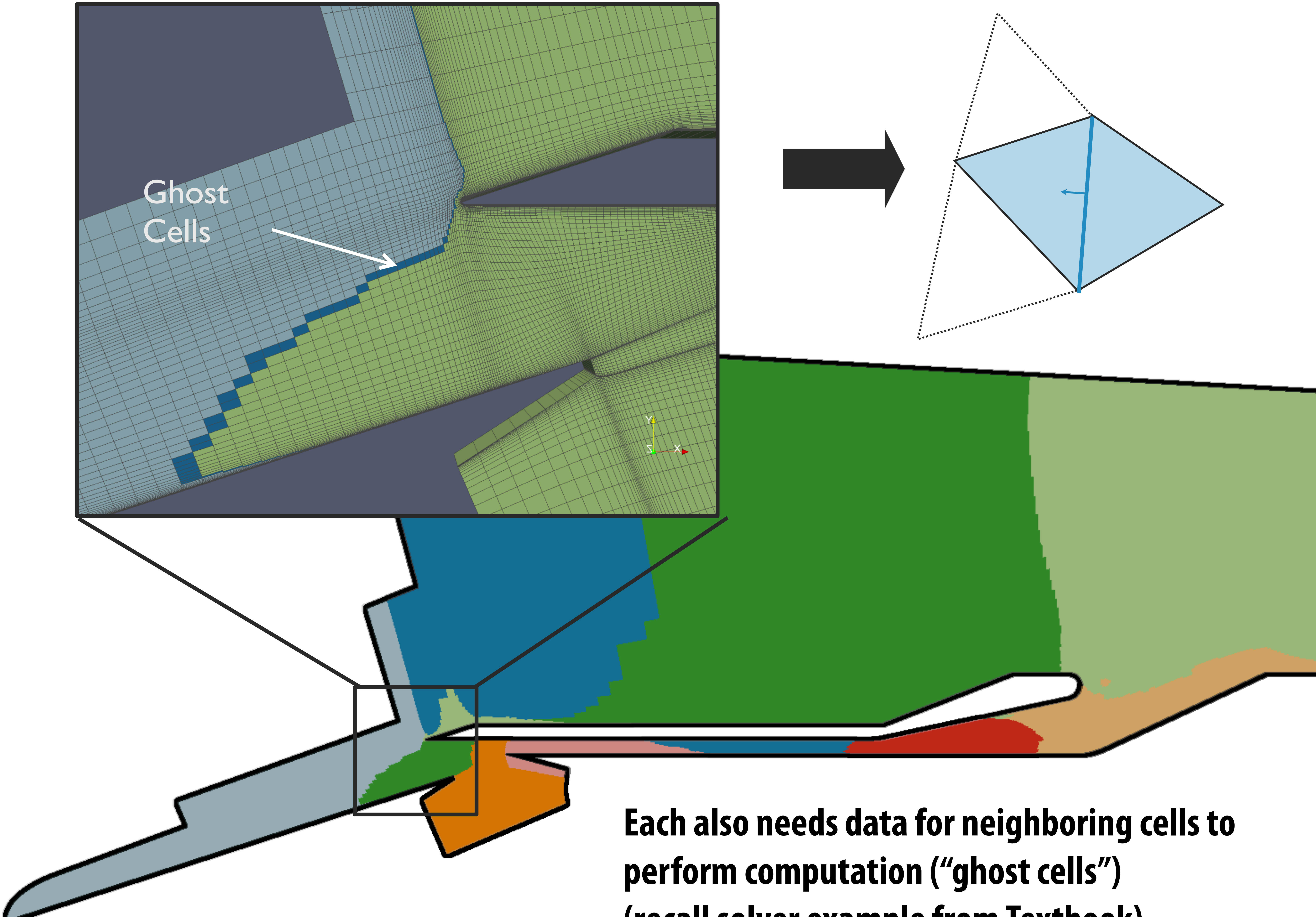
Distribution memory implementation:
Mesh + Stencil -> Graph -> Partition

```
for(f <- faces(mesh)) {
  rhoOutside(f) :=
    calc_flux( f,rho(outside(f) ))
    + calc_flux( f,rho(inside(f) ))
}
```





Consider distributed memory implementation
Store region of mesh on each node in a cluster
(ParMETIS is a tool to partition meshes)

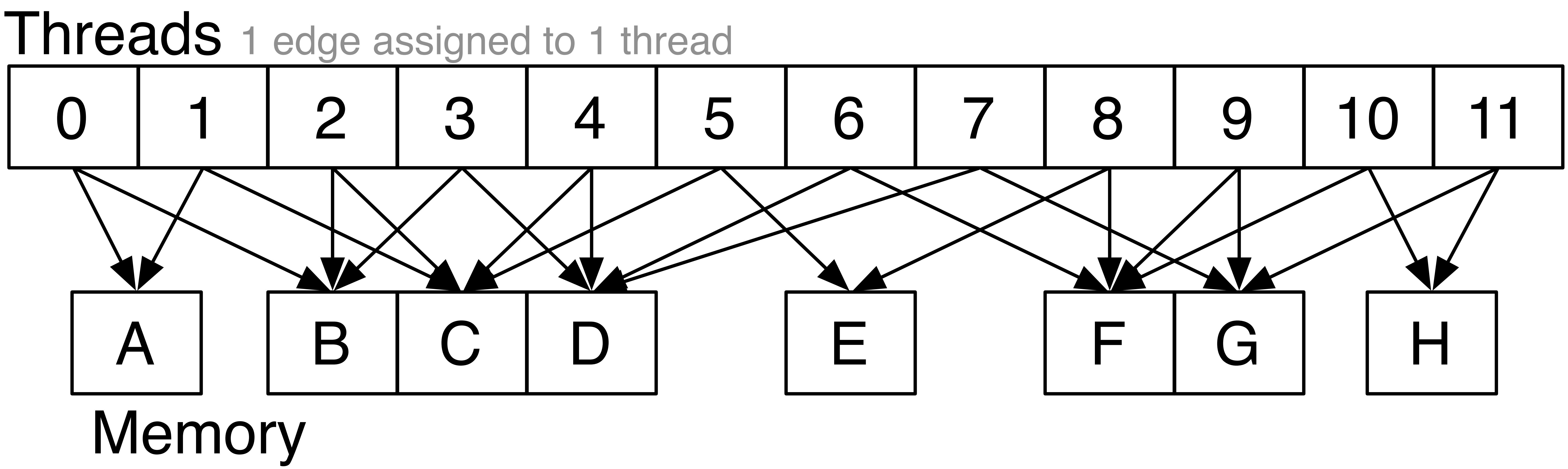


Each also needs data for neighboring cells to perform computation ("ghost cells")
(recall solver example from Textbook)

GPU implementation: parallel reductions

Previous example, one region of mesh per processor (or node in MPI cluster)

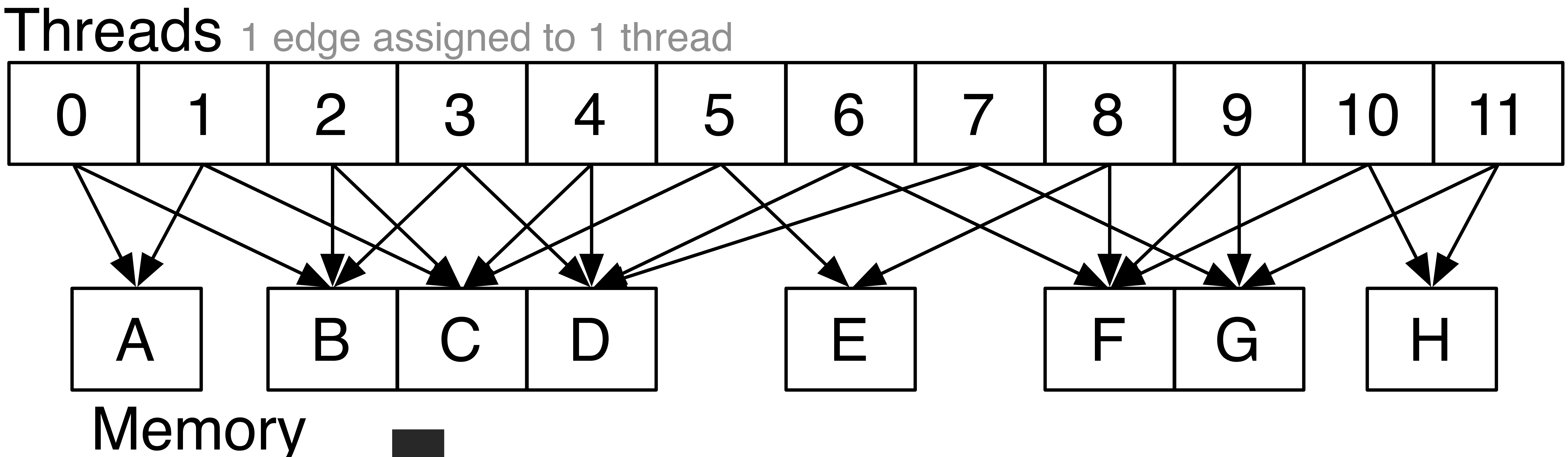
On GPU, natural parallelization is one edge per CUDA thread

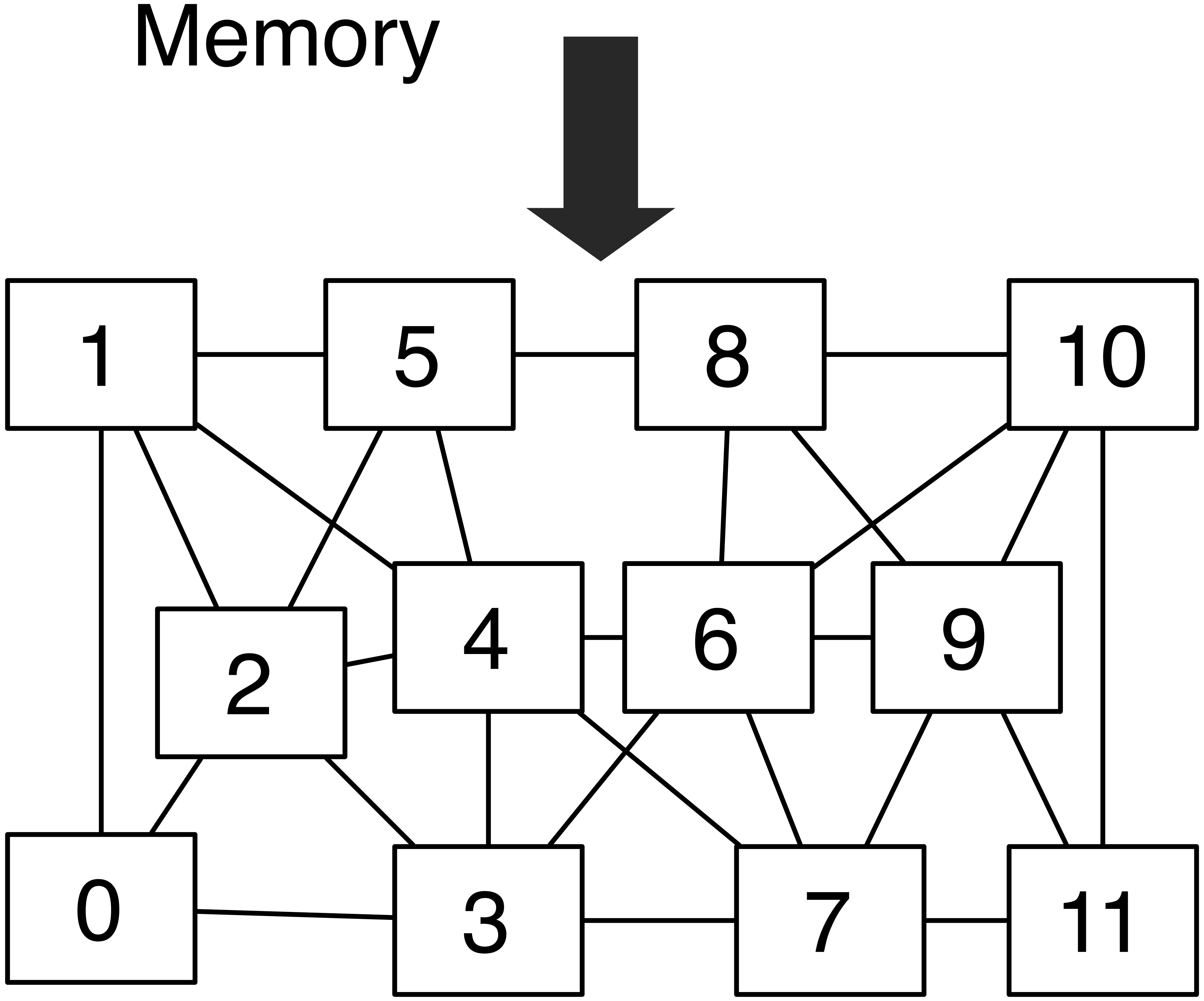



```
for (e <- edges(mesh)) {
  ...
  Flux(v1) += dT*step
  Flux(v2) -= dT*step
  ...
}
```

Different edges share a vertex: requires atomic update of per-vertex field data (Expensive: recall assignment 2)

GPU implementation: conflict graph

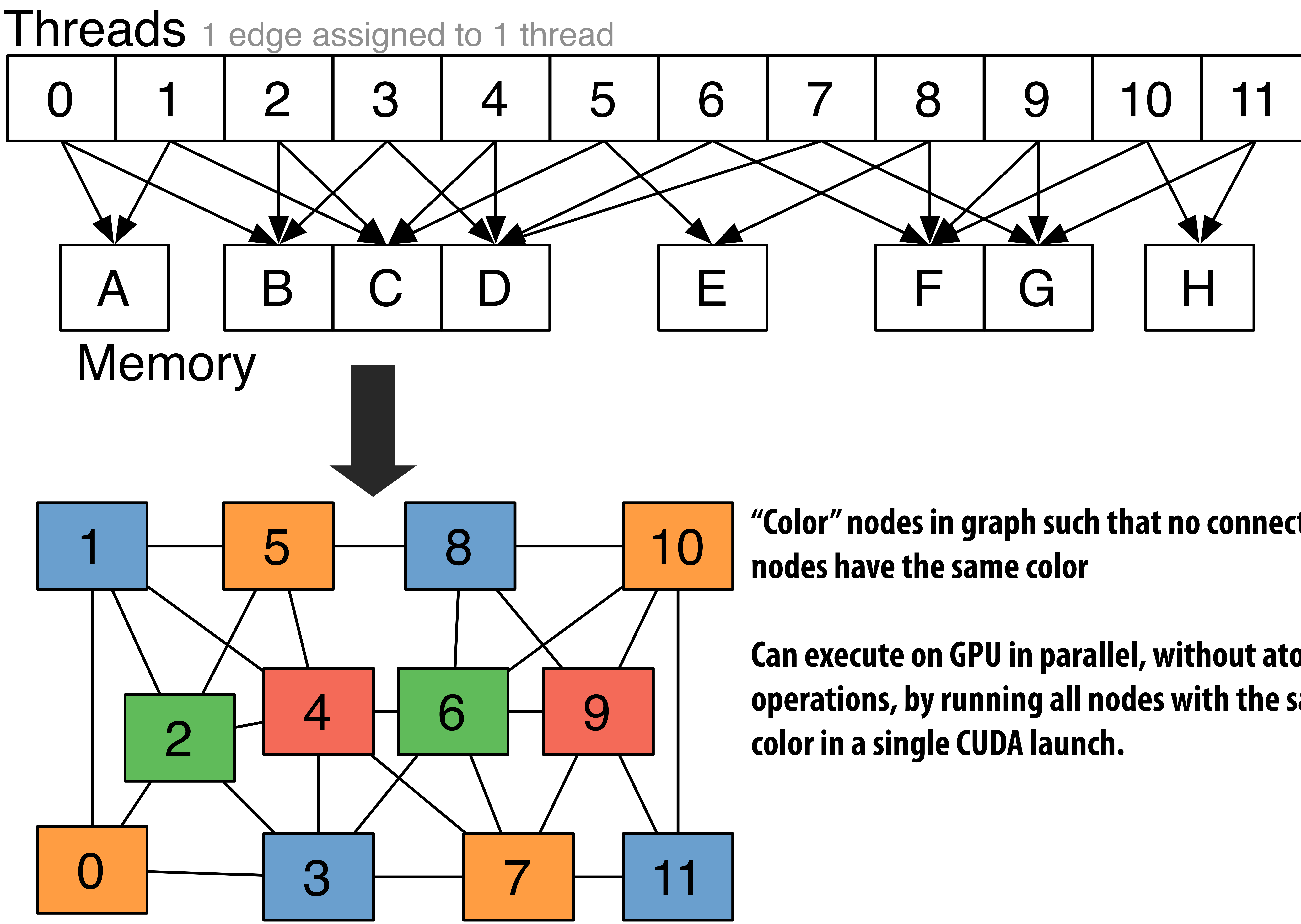




Identify mesh edges with colliding writes (lines in graph indicate presence of collision)

Can run program to get this information. (results valid provided mesh does not change)

GPU implementation: conflict graph

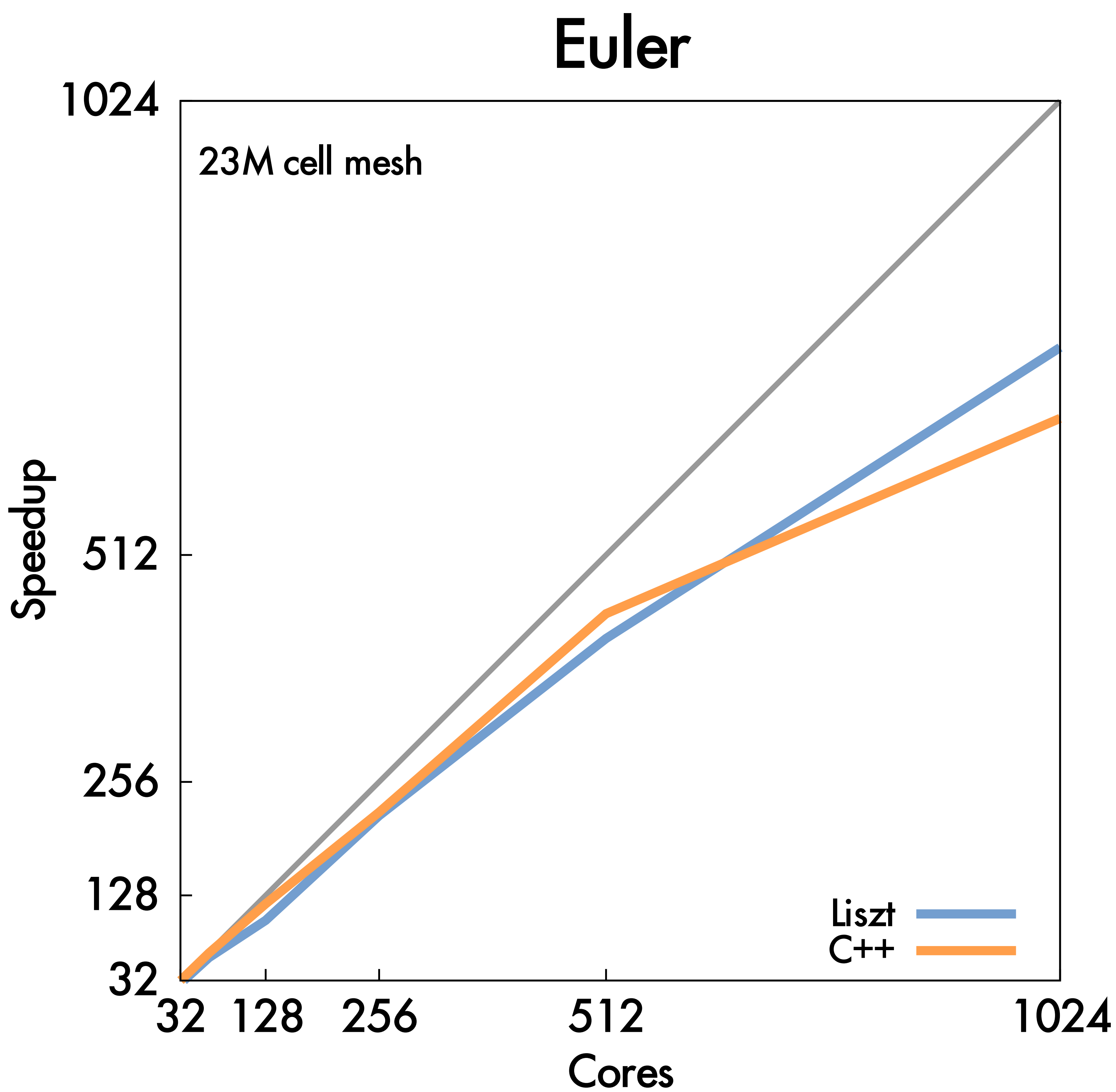


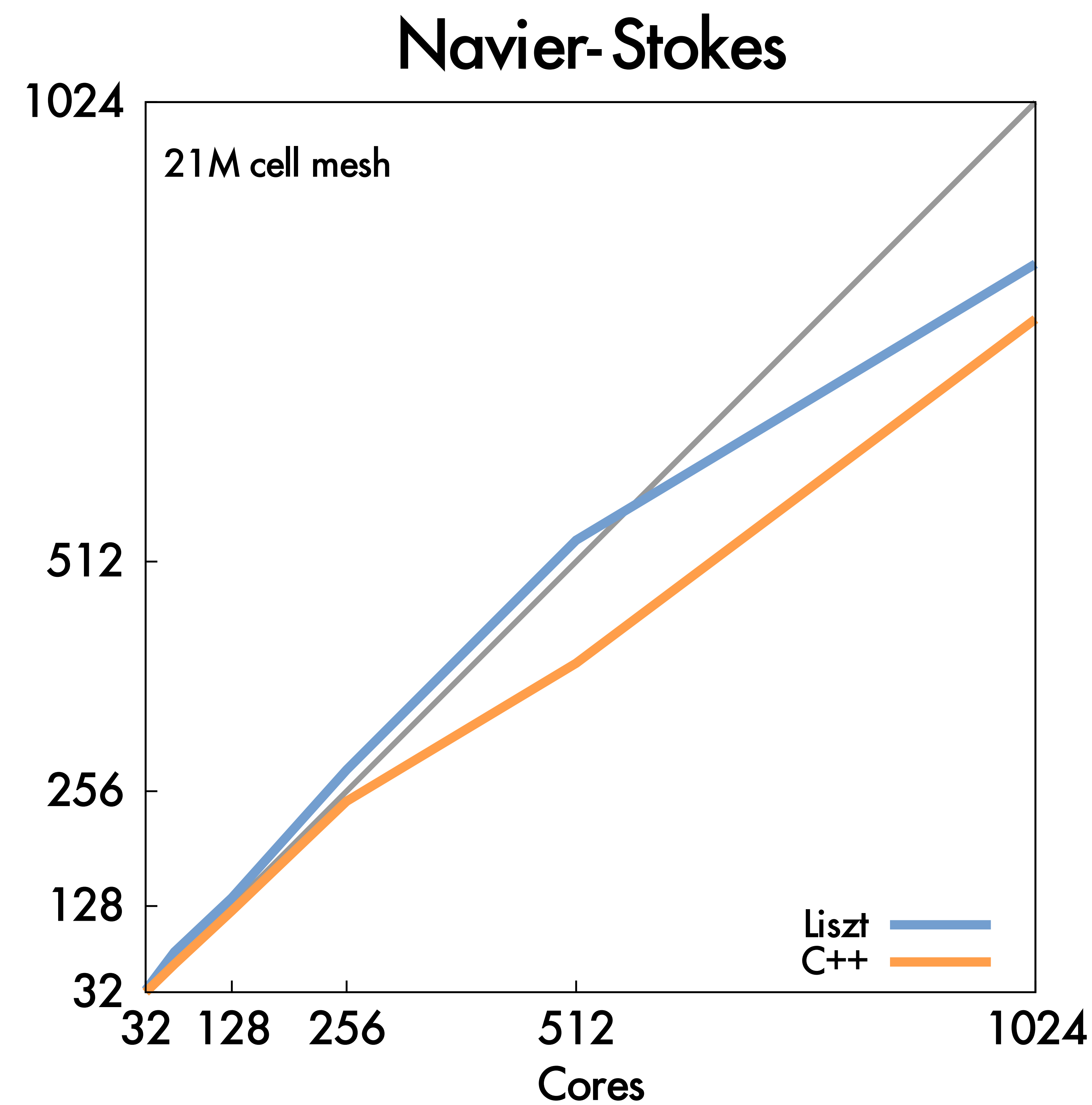
"Color" nodes in graph such that no connected nodes have the same color

Can execute on GPU in parallel, without atomic operations, by running all nodes with the same color in a single CUDA launch.

MPI Performance

256 nodes, 8 cores per node





Important:
Performance portability: same Liszt program also runs with high efficiency on GPU

Liszt summary

- **Productivity:**
  - Abstract representation of mesh: vertices, edges, faces, fields
  - Intuitive topological operators
- **Portability**
  - Same code runs on cluster of CPUs (MPI runtime) and GPUs
- **High-Performance**
  - Language constrained to allow compiler to track dependencies
  - Used for locality-aware partitioning in distributed memory implementation
  - Used for graph coloring in GPU implementation
  - Completely different parallelization strategies for difference platforms
  - Underlying mesh representation customized based on usage and platform (e.g, struct of arrays vs. array of structs)

Many other recent domain-specific programming systems

Less domain specific than examples given today, but still designed specifically for: data-parallel computations on big data for distributed systems (“Map-Reduce”)

Operations on graphs for machine learning

Model-view-controller paradigm for web-applications

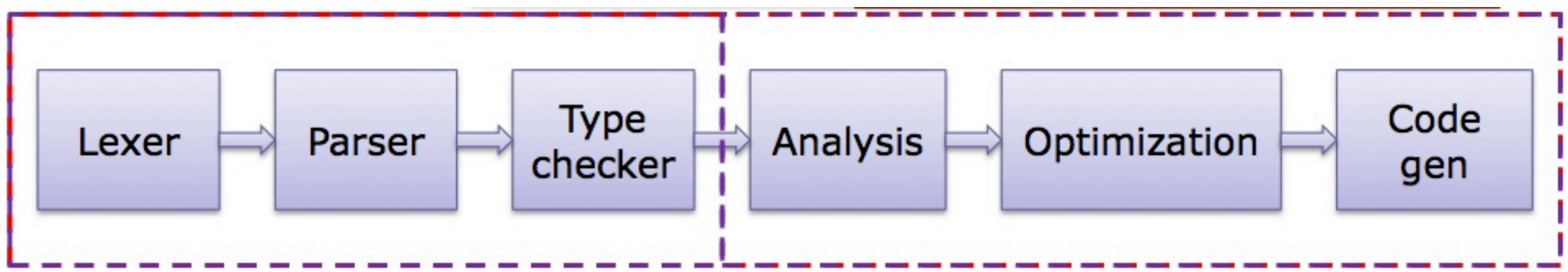
Emerging examples in:
Computer vision
Image processing
Statistics/machine learning

Domain-specific language development

- **Stand-alone language**
  - Graphics shading languages
  - MATLAB, SQL
- **Fully "embedded" in an existing generic language**
  - e.g., C++ library
  - GraphLab, OpenGL host-side API, Map-Reduce
- **Recent research idea:**
  - Design generic languages that have facilities that assist embedding of domain-specific languages

Facilitating development of new domain-specific languages

“Embed” domain-specific language in generic, flexible embedding language

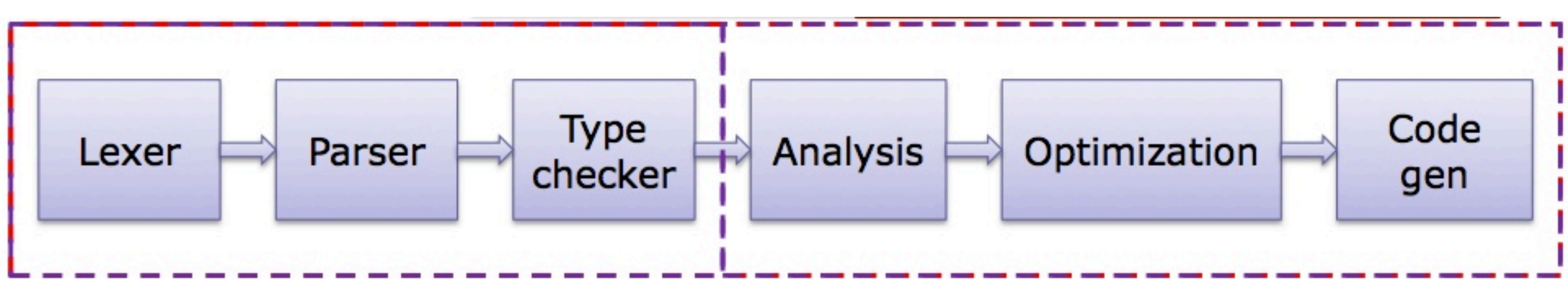


Typical Compiler

Stand-alone domain-special language must implement everything

---

“Modular staging” approach:



Domain language adopts front-end from highly expressive embedding language

Leverage techniques like operator overloading, modern OOP (traits), type inference, closures, to make embedding language syntax appear native:

Liszt code shown before was actually valid Scala!

But customizes intermediate representation (IR) and participates in backend optimization and code-generation phases (exploiting domain knowledge while doing so)

Credit: Hassan Chafi

Summary

- **Modern machines: parallel, heterogeneous**
  - Only way to increase compute capability in power-constrained world
- **Most software uses very little of peak capability of machine**
  - Very challenging to tune programs to these machines
  - Tuning efforts are not portable across machines
- **Domain-specific programming environments trade-off generality to achieve productivity, performance, and portability**
  - Examples today: OpenGL, Liszt