Portable parallelism in Diderot

John Reppy

University of Chicago

December 2, 2011

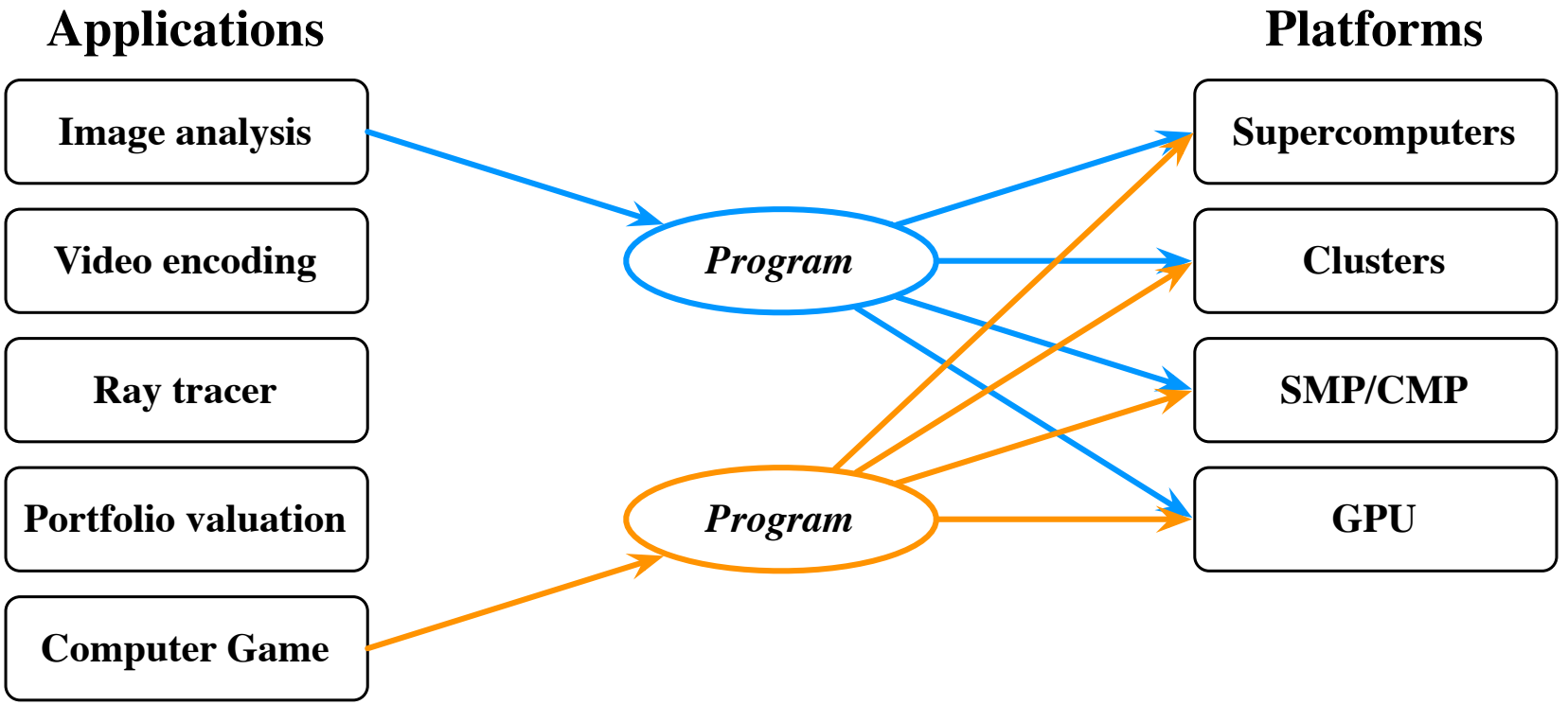
The challenge of portable parallelism

Wide range of parallel hardware with varying memory architectures:

- CMP (multicore): shared cache, uniform shared memory.
- SMP: separate caches, non-uniform shared memory (NUMA).
- GPUs: wide-vector instructions, explicit memory hierarchy, distributed memory.
- Cluster: separate caches and distributed memory.
- Supercomputer: specialized interconnects, heterogeneous architectures, *etc.*

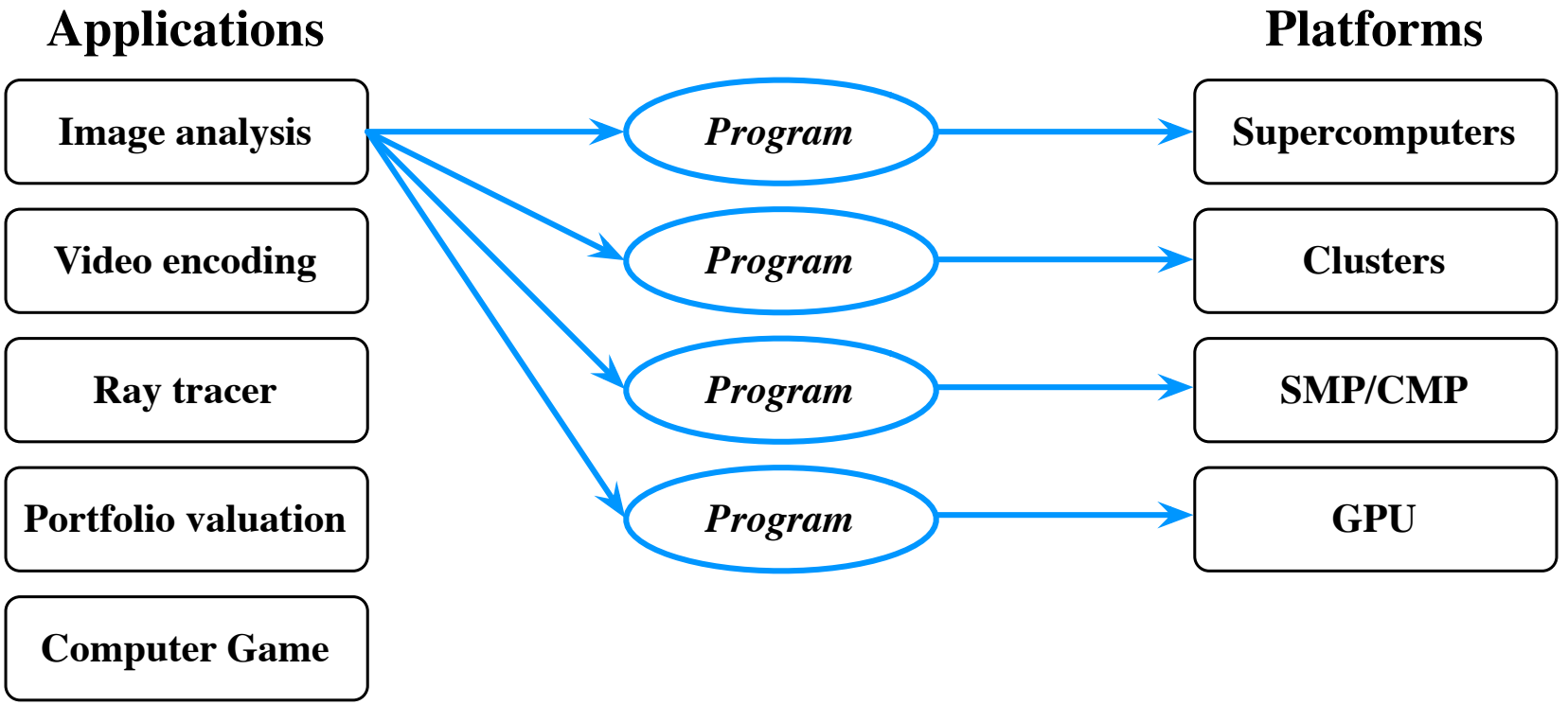
And we have a wide range of parallel applications.

Portable parallel programming



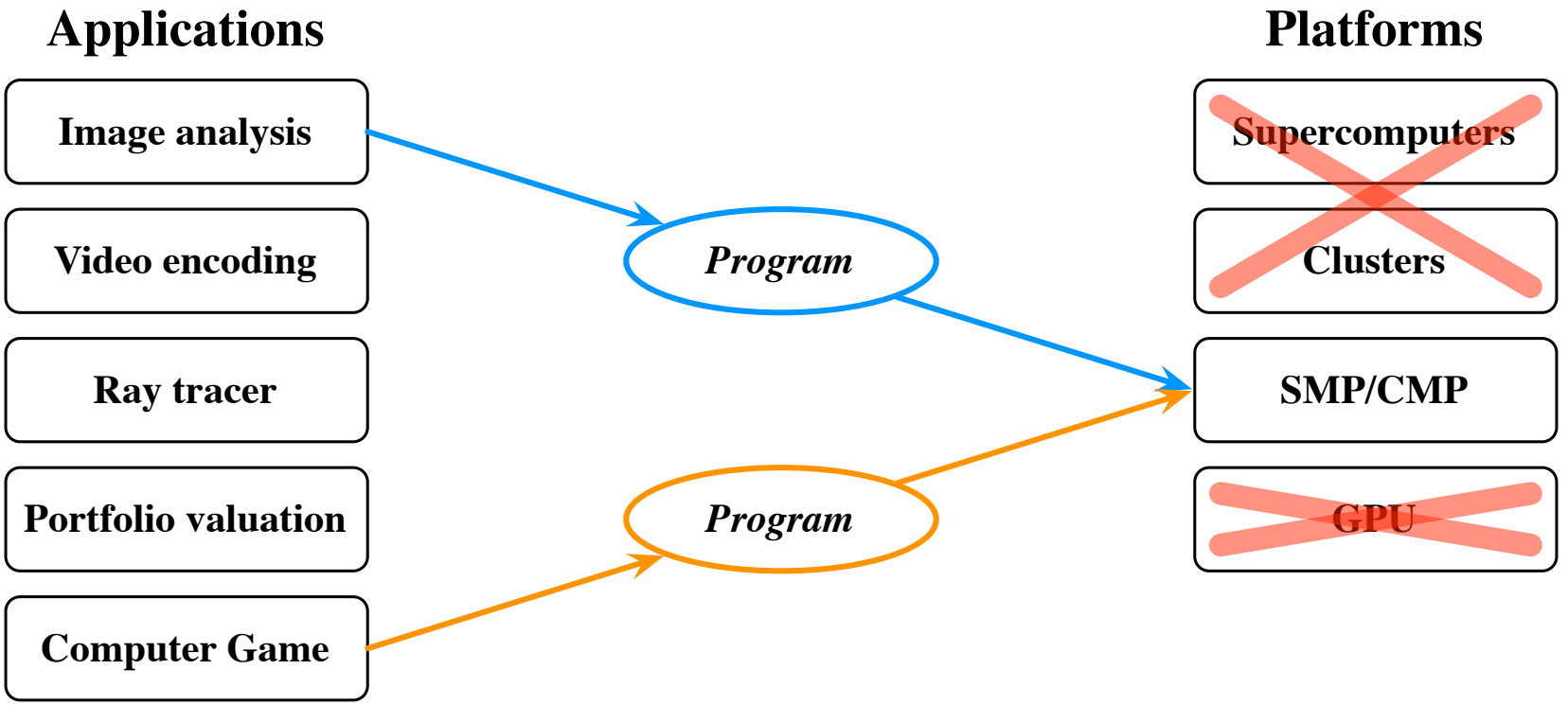
The Ideal: write once, run everywhere, for any application

Portable parallel programming



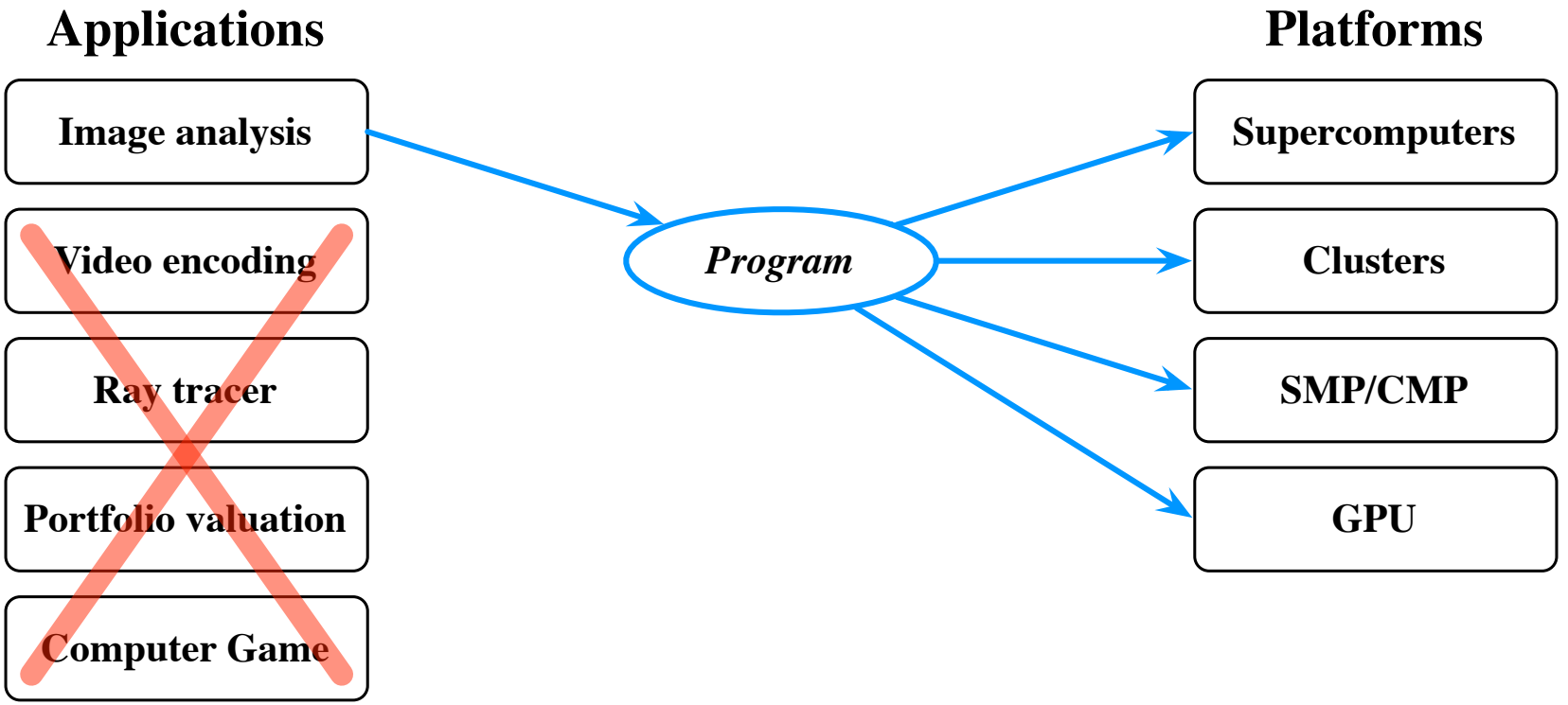
The Reality: write once per platform per application

Portable parallel programming



Manticore: restrict platforms

Portable parallel programming



Diderot: restrict applications (also Spiral and Delite)

Diderot

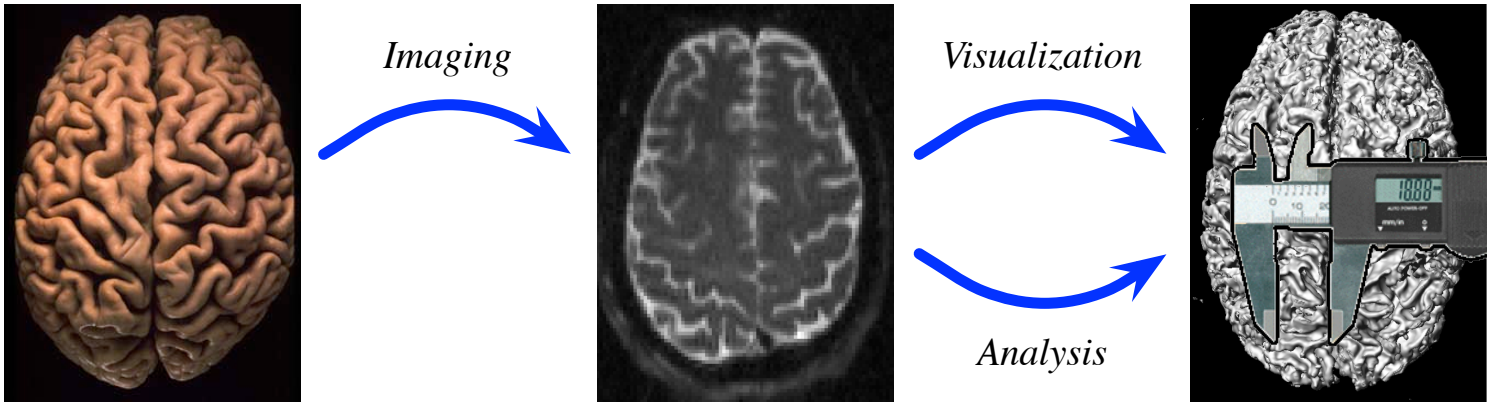
Diderot is a cross-discipline project involving

- Biomedical image analysis and visualization
- Programming language design and implementation

Plan: use ideas from programming languages to improve the state of the art in image-analysis and visualization.

Joint work with Gordon Kindlmann and students Charisee Chiw, Lamont Samuels, and Nick Seltzer.

Why image analysis is important

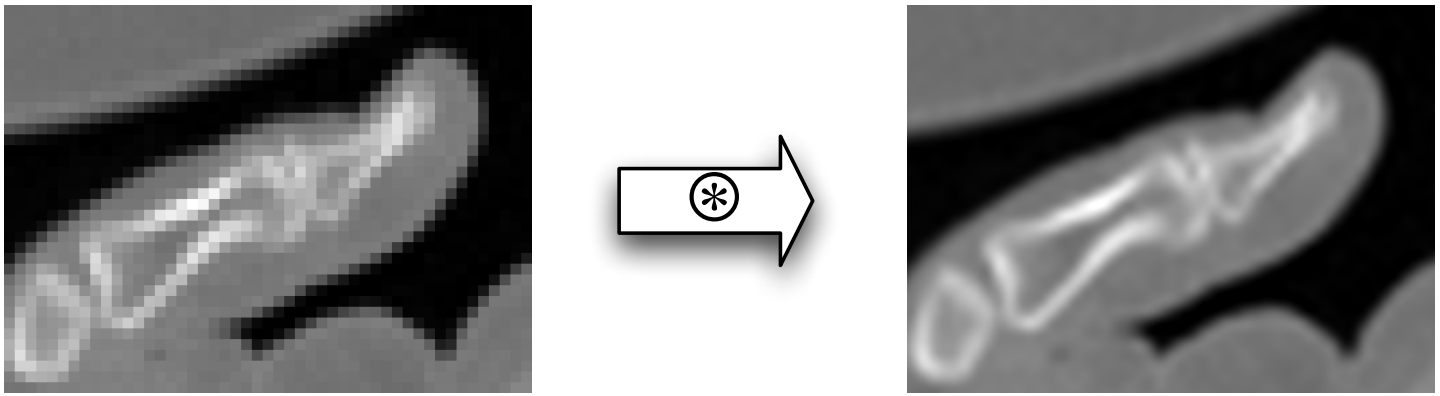


Physical object **Image data** **Computational representation**

Scientists need tools to extract structure from many kinds of image data.

Image analysis and visualization

- We are interested in a class of algorithms that compute geometric properties of objects from imaging data.
- These algorithms compute over a continuous tensor field that is reconstructed from discrete data using a separable convolution kernel.



Discrete image data *Continuous field*

Image analysis and visualization *(continued ...)*

Examples include

- Direct volume rendering (requires reconstruction, derivatives)
- Fiber tractography (requires tensor fields)
- Particle systems (requires dynamic numbers of computational elements)

Image analysis and visualization *(continued ...)*

Examples include

- Direct volume rendering (requires reconstruction, derivatives)
- Fiber tractography (requires tensor fields)
- Particle systems (requires dynamic numbers of computational elements)

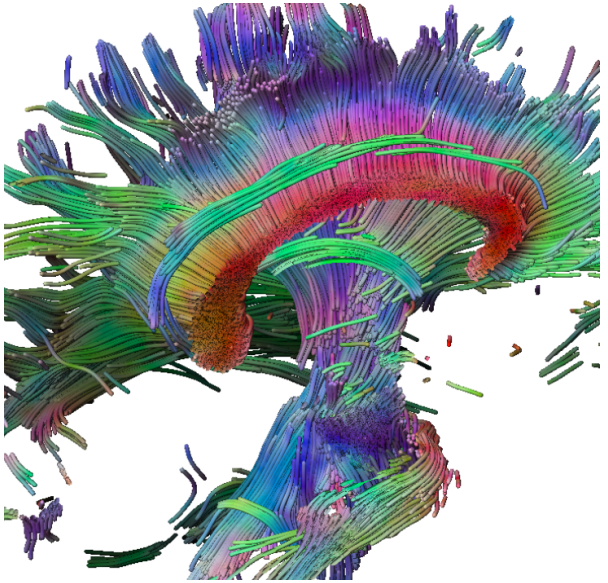
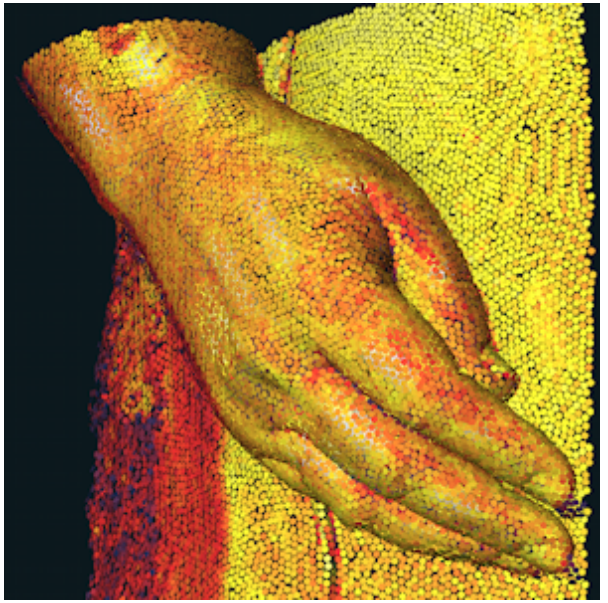


Image analysis and visualization *(continued ...)*

Examples include

- Direct volume rendering (requires reconstruction, derivatives)
- Fiber tractography (requires tensor fields)
- Particle systems (requires dynamic numbers of computational elements)



DSL for image analysis

Diderot is a parallel DSL for image analysis and visualization algorithms.

We have two main design goals for Diderot:

- Provide a high-level mathematical programming model that abstracts away from discrete image data and the target architecture.
- Use domain knowledge to get good performance on a range of parallel platforms, without requiring an understanding of parallel programming.

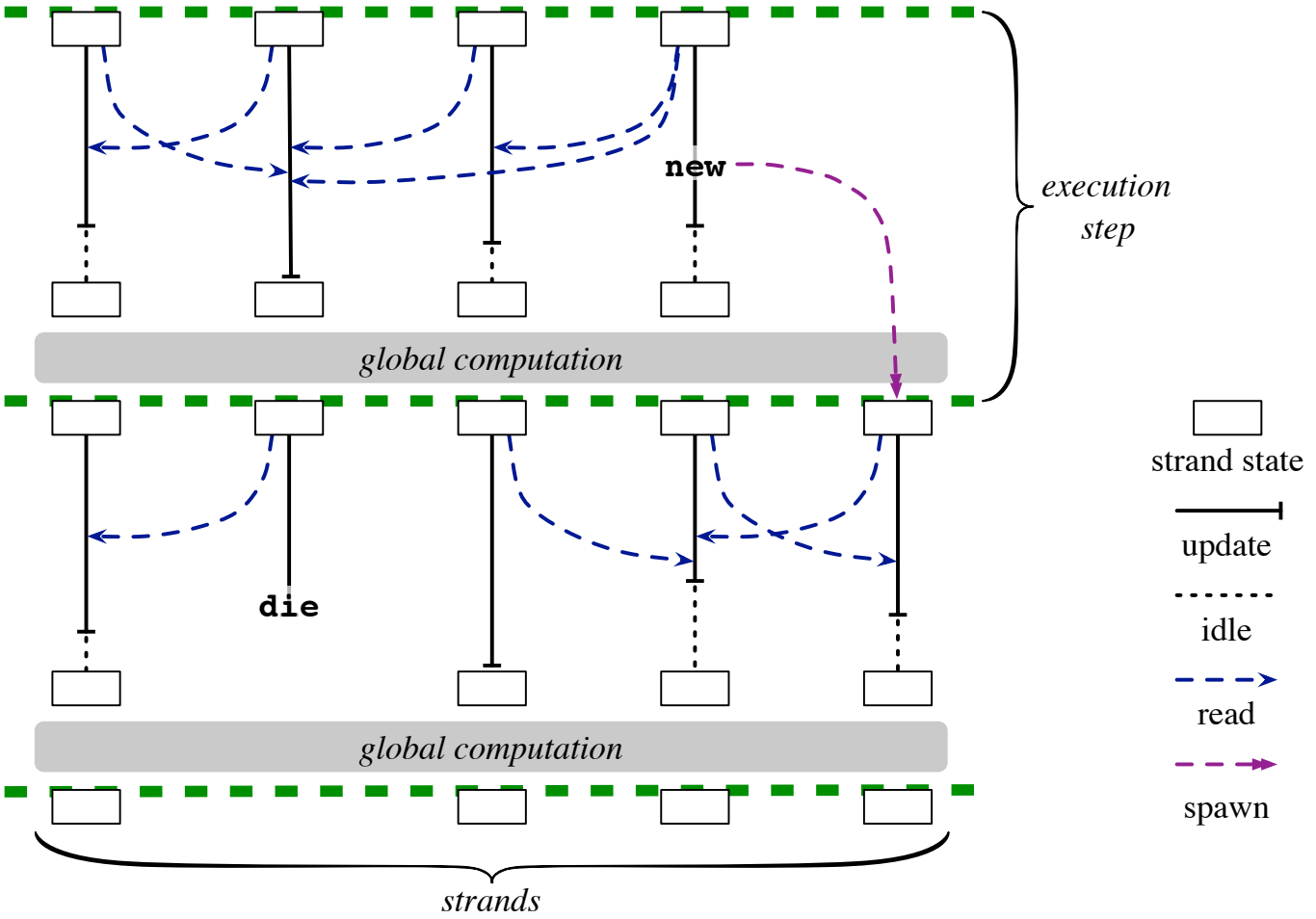
Note: Diderot is not an embedded DSL.

Diderot programming model

- The Diderot programming model is based on a collection of mostly autonomous strands that are embedded in a continuous tensor field.
- Each strand has a state and an **`update`** method, which encapsulates the computational kernel of the algorithm.
- Diderot abstracts away from details such as the discrete image-data, the representation of reals (float vs double), and the target machine (*e.g.*, CPU vs GPU).
- The computational kernel of a Diderot program is expressed using the concepts and direct-style notation of tensor calculus. These include tensor operations ($\bullet$, $\times$) and higher-order field operations (∇), *etc.*
- No shared mutable state.

Diderot parallelism model

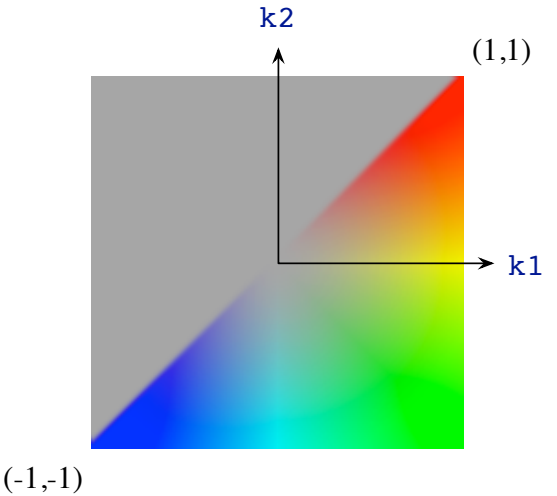
Bulk-synchronous parallel with "deterministic" semantics.

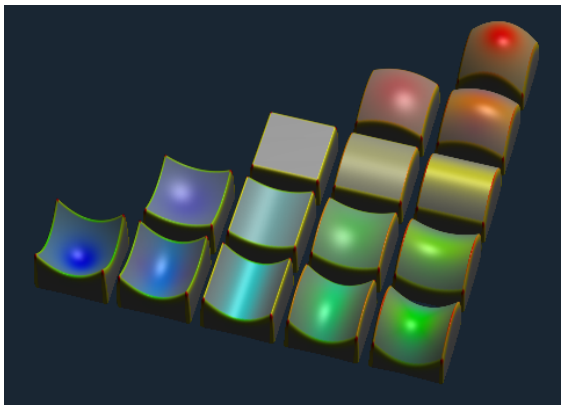


Note: the current language implements a subset of this model.

Example — Curvature

```
field#2(3)[] F = bspln3 ⊛ load("quad-patches.nrrd");
field#0(2)[3] RGB = tent ⊛ load("2d-bow.nrrd");
...
strand RayCast (int ui, int vi) {
  ...
  update {
    ...
    vec3 grad = -∇F(pos);
    vec3 norm = normalize(grad);
    tensor[3,3] H = ∇⊗∇F(pos);
    tensor[3,3] P = identity[3] - norm⊗norm;
    tensor[3,3] G = -(P•H•P)/|grad|;
    real disc = sqrt(2.0*|G|^2 - trace(G)^2);
    real k1 = (trace(G) + disc)/2.0;
    real k2 = (trace(G) - disc)/2.0;
    vec3 matRGB = // material RGBA
        RGB([max(-1.0, min(1.0, 6.0*k1)),
             max(-1.0, min(1.0, 6.0*k2))]);
    ...
  }
...
}
```





Example — 2D Isosurface

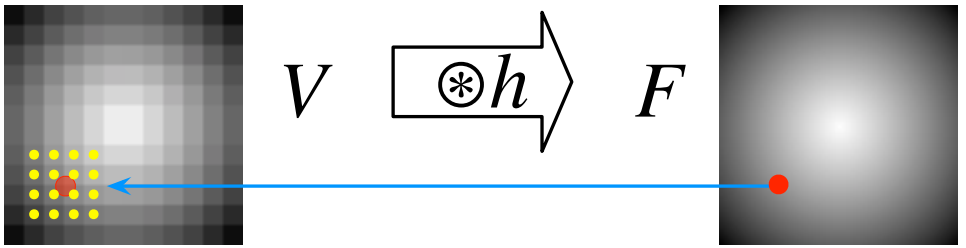
```
...
strand sample (int ui, int vi) {
  output vec2 pos = ···;
// set isovalue to closest of 50, 30, or 10
  real isoval = 50.0 if F(pos) >= 40.0
              else 30.0 if F(pos) >= 20.0
              else 10.0;
  int steps = 0;
  update {
    if (!inside(pos, F) || steps > stepsMax)
      die;
    vec2 grad = ∇F(pos);
  // delta = Newton-Raphson step
    vec2 delta = normalize(grad) * (F(pos) - isoval)/|grad|;
    if (|delta| < epsilon)
      stabilize;
    pos = pos - delta;
    steps = steps + 1;
  }
}
```

Diderot compiler and runtime

- Compiler is 21,000 lines of SML (2,500 in front-end).
- Multiple backends: vectorized C and OpenCL (CUDA under construction).
- Multiple runtimes: Sequential C, Parallel C, OpenCL.
- Designed to generate libraries, but also supports standalone executables.

Probing tensor fields

A probe gets compiled down into code that maps the world-space coordinates to image space and then convolves the image values in the neighborhood of the position.



Discrete image data *Continuous field*

In 2D, the reconstruction is (recall that h is separable)

$$F(\mathbf{x}) = \sum_{i=1-s}^{s} \sum_{j=1-s}^{s} V[\mathbf{n} + \langle i,j \rangle] h(\mathbf{f}_x - i) h(\mathbf{f}_y - j)$$

where s is the support of h, $\mathbf{n} = \lfloor \mathbf{M}^{-1}\mathbf{x} \rfloor$ and $\mathbf{f} = \mathbf{M}^{-1}\mathbf{x} - \mathbf{n}$.

Probing tensor fields *(continued ...)*

In general, generating the probe operations is more challenging.
The first step is to normalize field expressions. For example,

$$\begin{aligned} \nabla(s*(V \circledast h)) &\Rightarrow s*(\nabla(V \circledast h)) \\ &\Rightarrow s*(V \circledast (\nabla h)) \end{aligned}$$

In the implementation, we view ∇ as a "tensor" of partial-derivative operators

$$\nabla = \begin{bmatrix} \frac{\partial}{\partial x} \\ \frac{\partial}{\partial y} \end{bmatrix} \qquad \nabla \otimes \nabla = \begin{bmatrix} \frac{\partial^2}{\partial x^2} & \frac{\partial^2}{\partial xy} \\ \frac{\partial^2}{\partial xy} & \frac{\partial^2}{\partial y^2} \end{bmatrix}$$

Probing tensor fields *(continued ...)*

Each component in the partial-derivative tensor corresponds to a component in the result of the probe.

$$
\begin{aligned}
V \circledast (\nabla h) &= V \circledast \begin{bmatrix} \frac{\partial}{\partial x} h \\ \frac{\partial}{\partial y} h \end{bmatrix} \\
&= \begin{bmatrix} \sum_{i=1-s}^{s} \sum_{j=1-s}^{s} V[\mathbf{n} + \langle i, j \rangle] h'(\mathbf{f}_x - i) h(\mathbf{f}_y - j) \\ \sum_{i=1-s}^{s} \sum_{j=1-s}^{s} V[\mathbf{n} + \langle i, j \rangle] h(\mathbf{f}_x - i) h'(\mathbf{f}_y - j) \end{bmatrix}
\end{aligned}
$$

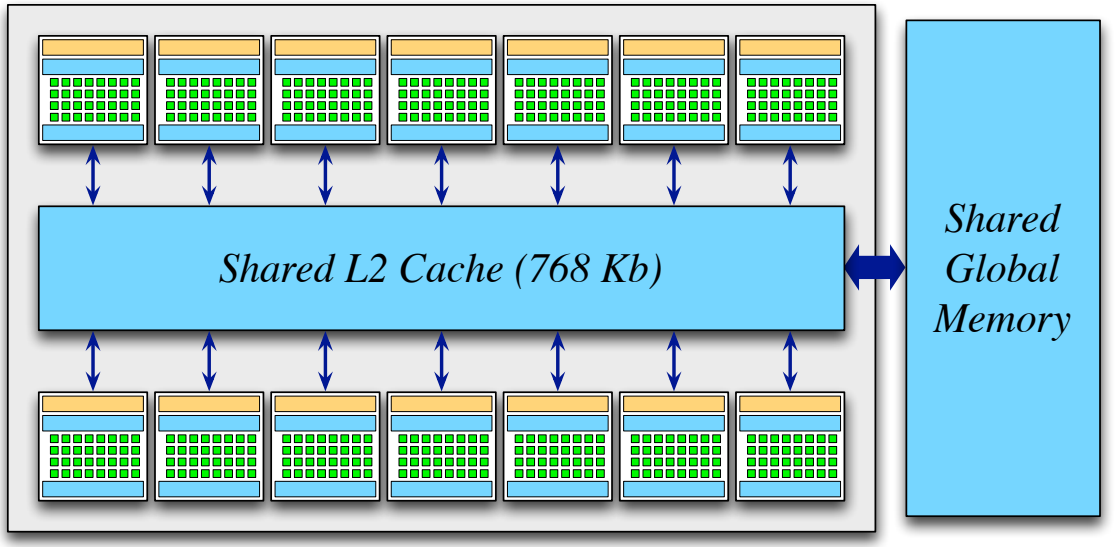
A later stage of the compiler expands out the evaluations of h and h'.

Probing code has high arithmetic intensity and is a good candidate for vectorization and GPUs.

Targeting GPUs

- Standard GPGPU programming models (CUDA and OpenCL) are low-level and expose hardware details.
- Diderot frees the programmer from those issues, but the compiler and runtime must still handle them.
- We need to be smart about memory access and divergence.

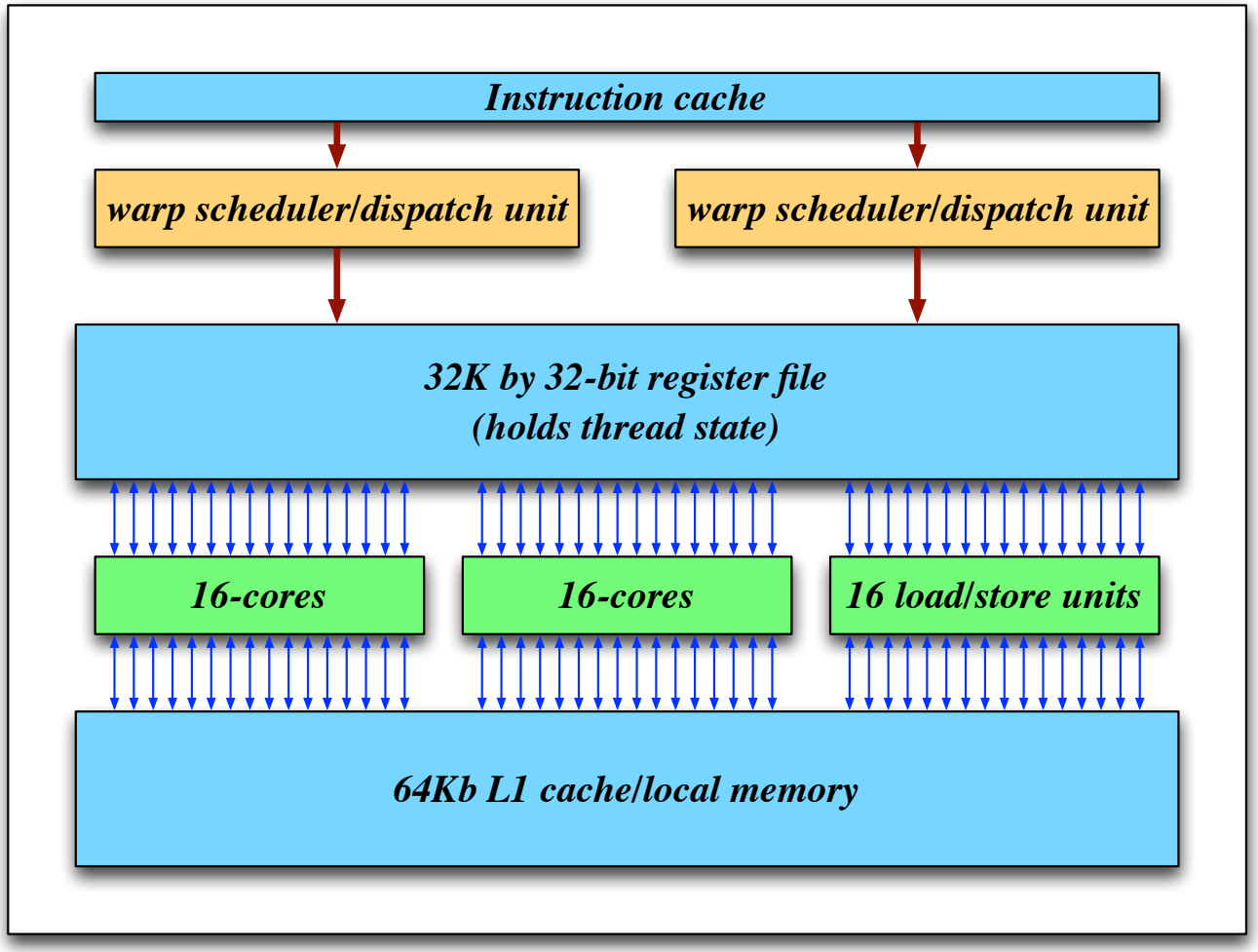
Nividia's Fermi architecture



- Multi-processor compute units share L2 cache and global memory.
- **S**ingle-**I**nstruction, **M**ultiple-**T**hread execution model.
- Each warp (32 threads) executes the same instruction.
- Predication used to handle divergent control flow.
- Each compute unit runs its own warps.

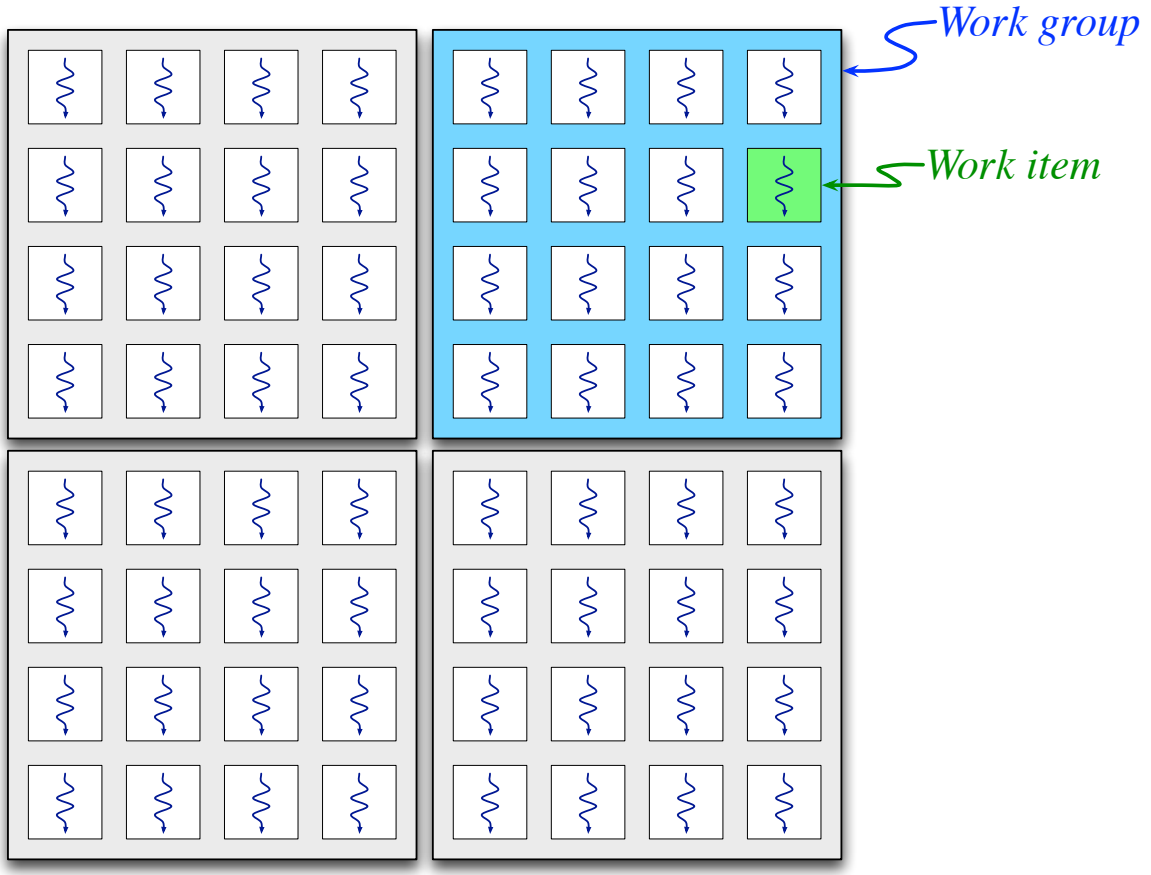
Fermi Compute Unit (CUDA 2.0)

Dispatch two half-warps per clock.



OpenCL Parallelism Model

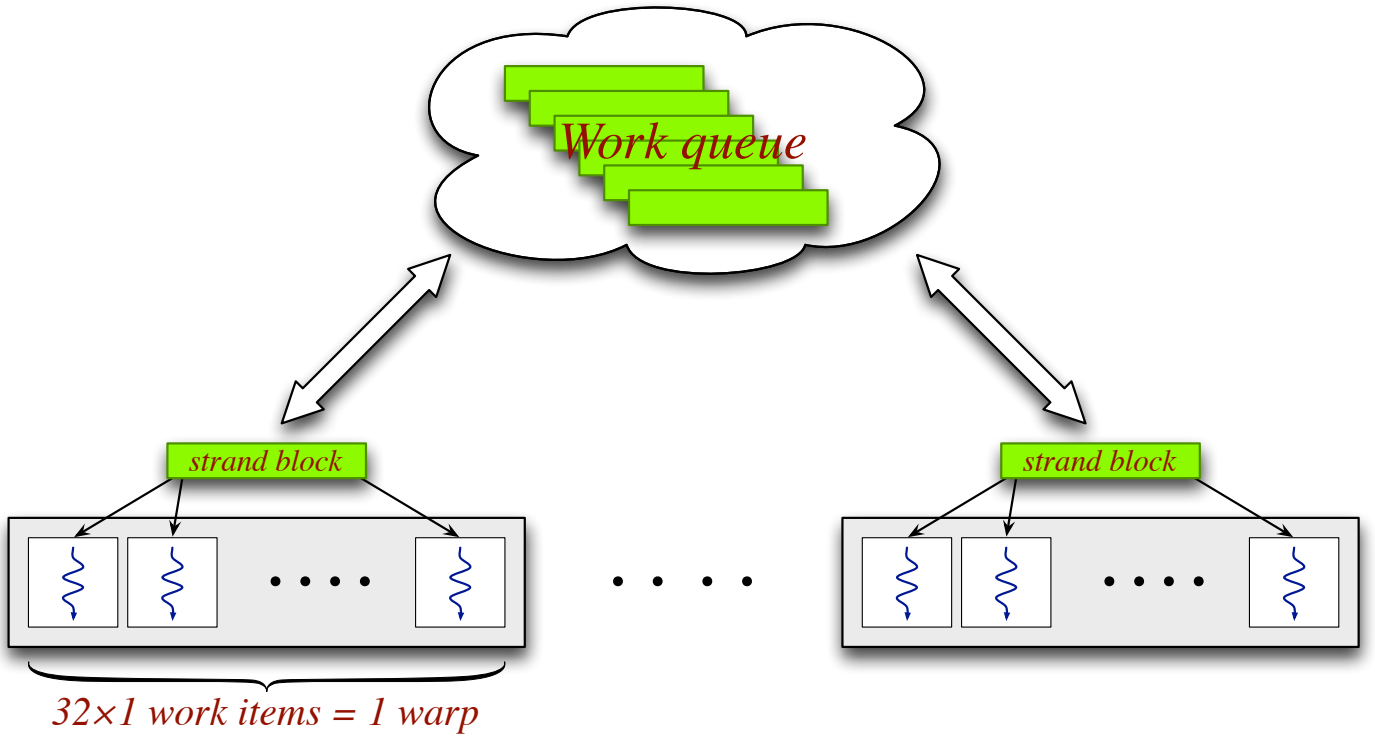
Grid of work items (threads) organized into work groups.



Standard approach: map data to the grid and run data-parallel computation.
This approach does not work well for irregular workloads.

Persistent threads

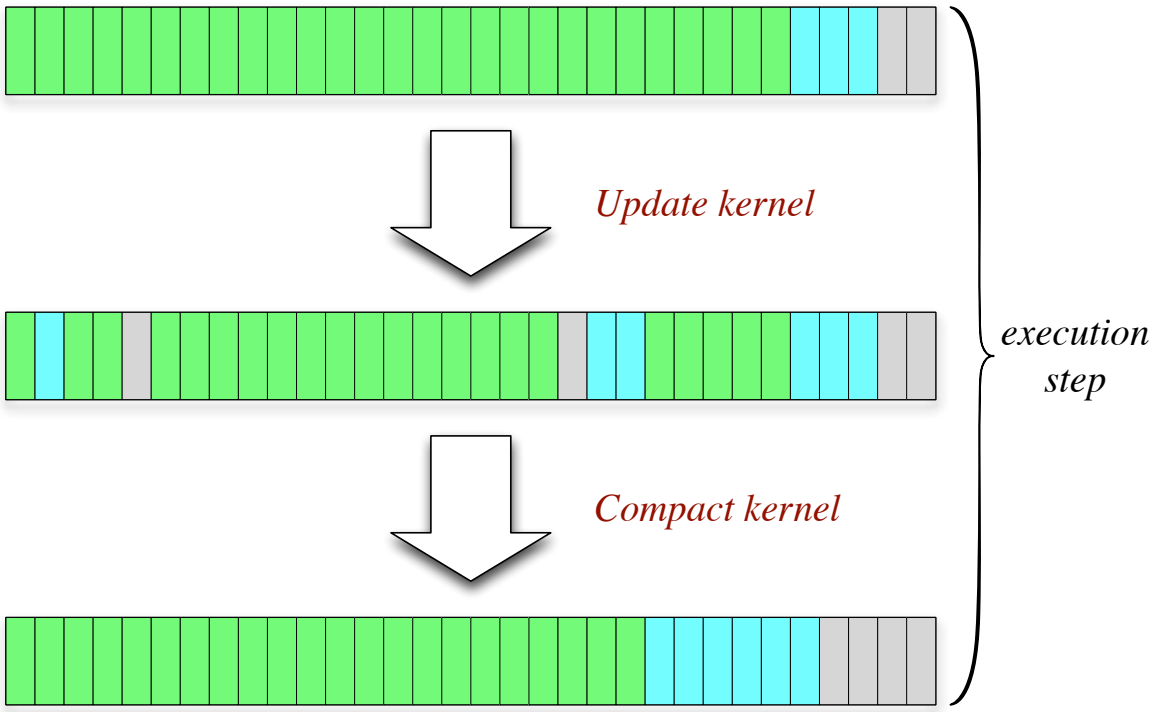
[Hoberock *et al.* 2009; Parker 2010; Wald 2011]
Instead of using the GPU scheduler, each workgroup runs a 32-wide parallel strand scheduler (64-wide on AMD hardware).



Each scheduler runs strand update methods until there are no more blocks.

Avoiding divergence

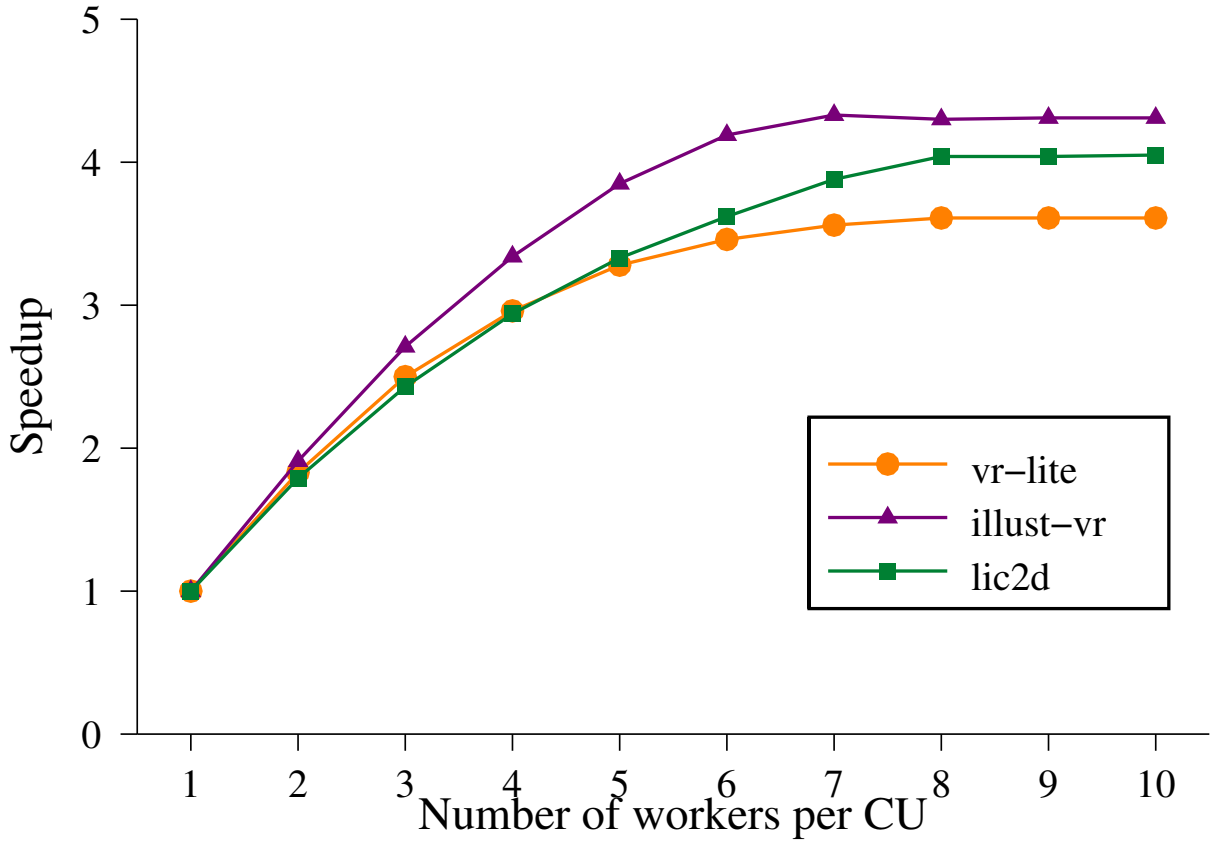
Each execution step is divided into two phases: update and compaction.



When occupancy gets too low, we compact across blocks.

Latency hiding

To hide memory latency, we run multiple workgroups per GPU compute unit.



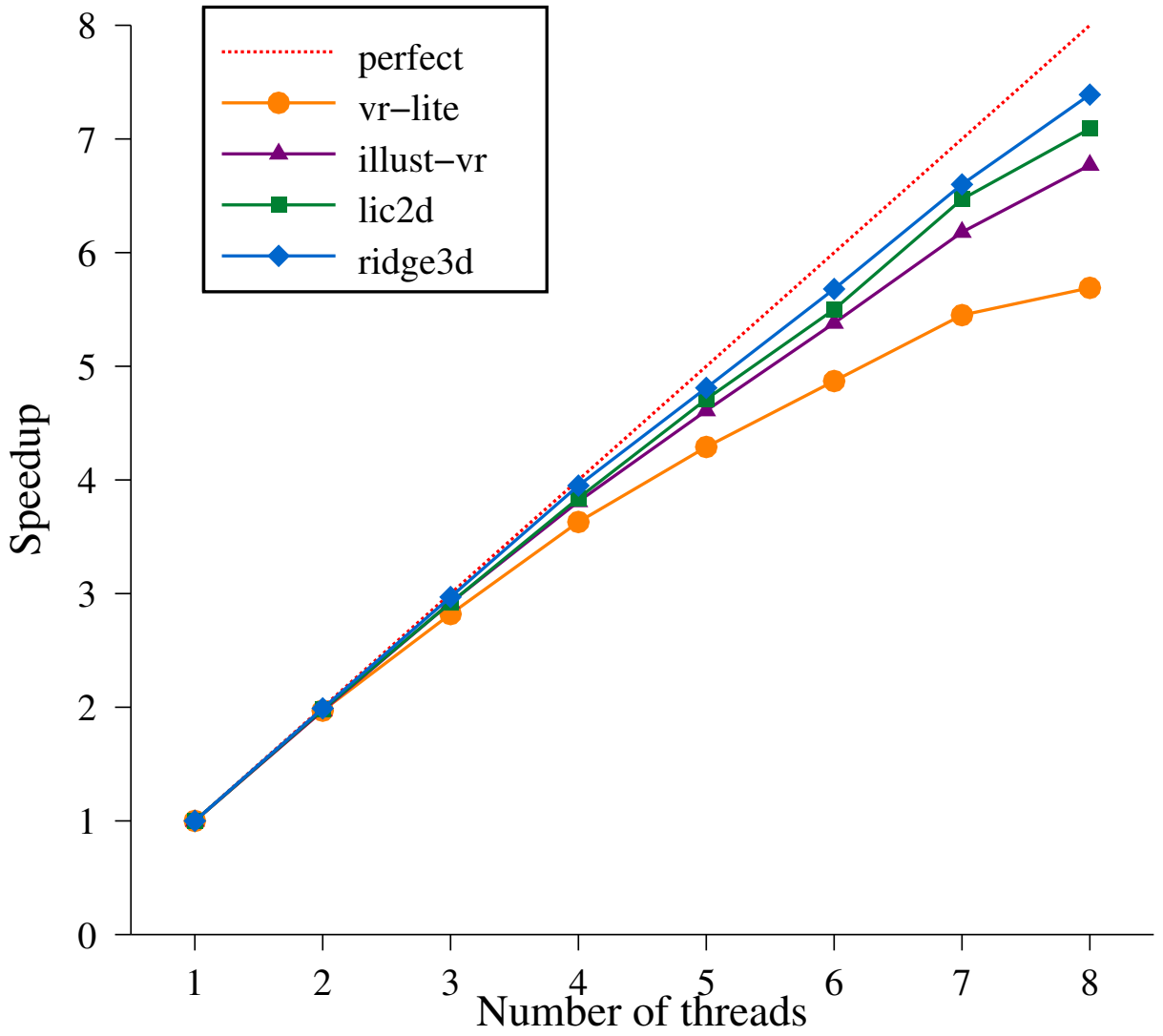
Runtime system could adjust the number of cores dynamically.

Experimental framework

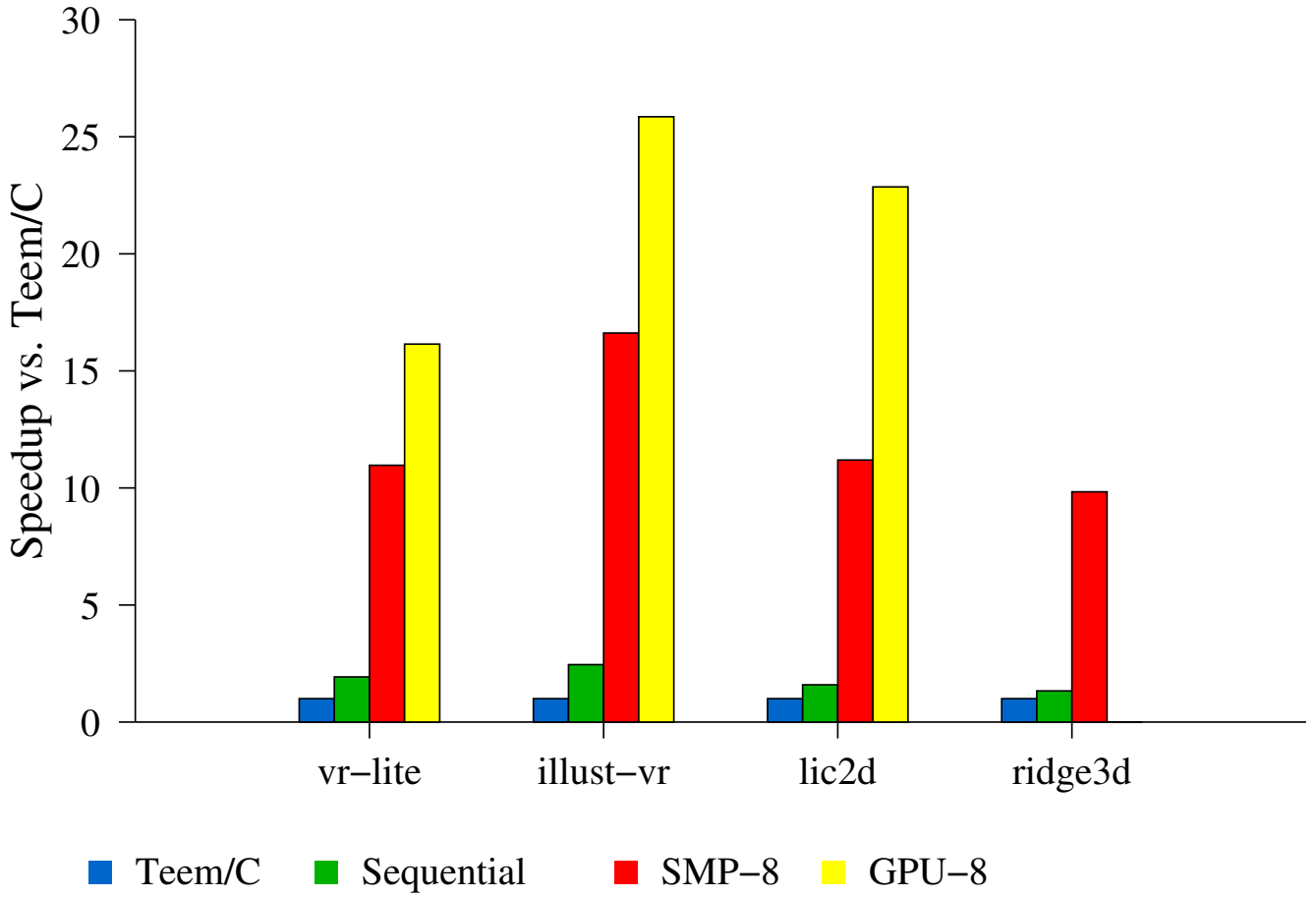
- Compare four versions of benchmarks: Teem/C, Sequential Diderot, Parallel Diderot, GPU Diderot.
- SMP machine: 8-core MacPro with 2.93 GHz Xeon X5570 processors (SSE-4)
- GPU machine: Linux box with NVIDIA Tesla C2070 (14×32 cores).
- Four typical benchmark programs
  - **vr-lite** — simple volume-renderer with Phong shading running on CT scan of hand
  - **illust-vr** — fancy volume-renderer with cartoon shading running on CT scan of hand
  - **lic2d** — line integral convolution in 2D running on synthetic data
  - **ridge3d** — particle-based ridge detection running on lung data

SMP scaling

Parallel performance scaling with respect to sequential Diderot.



Performance comparison



Note that **ridge3d** triggers a bug in NVIDIA's OpenCL compiler.

Language evolution

- Dynamic strand creation.
- Strand-strand interactions.
- Global computation mechanisms.
- Type inference and dimension polymorphism.

Long-term goals

In the future, we would like to generalize this work in two directions:

- Extend Diderot to other classes of algorithms (*e.g.*, object recognition).
- Generalize approach to other domains.

Long-term goals

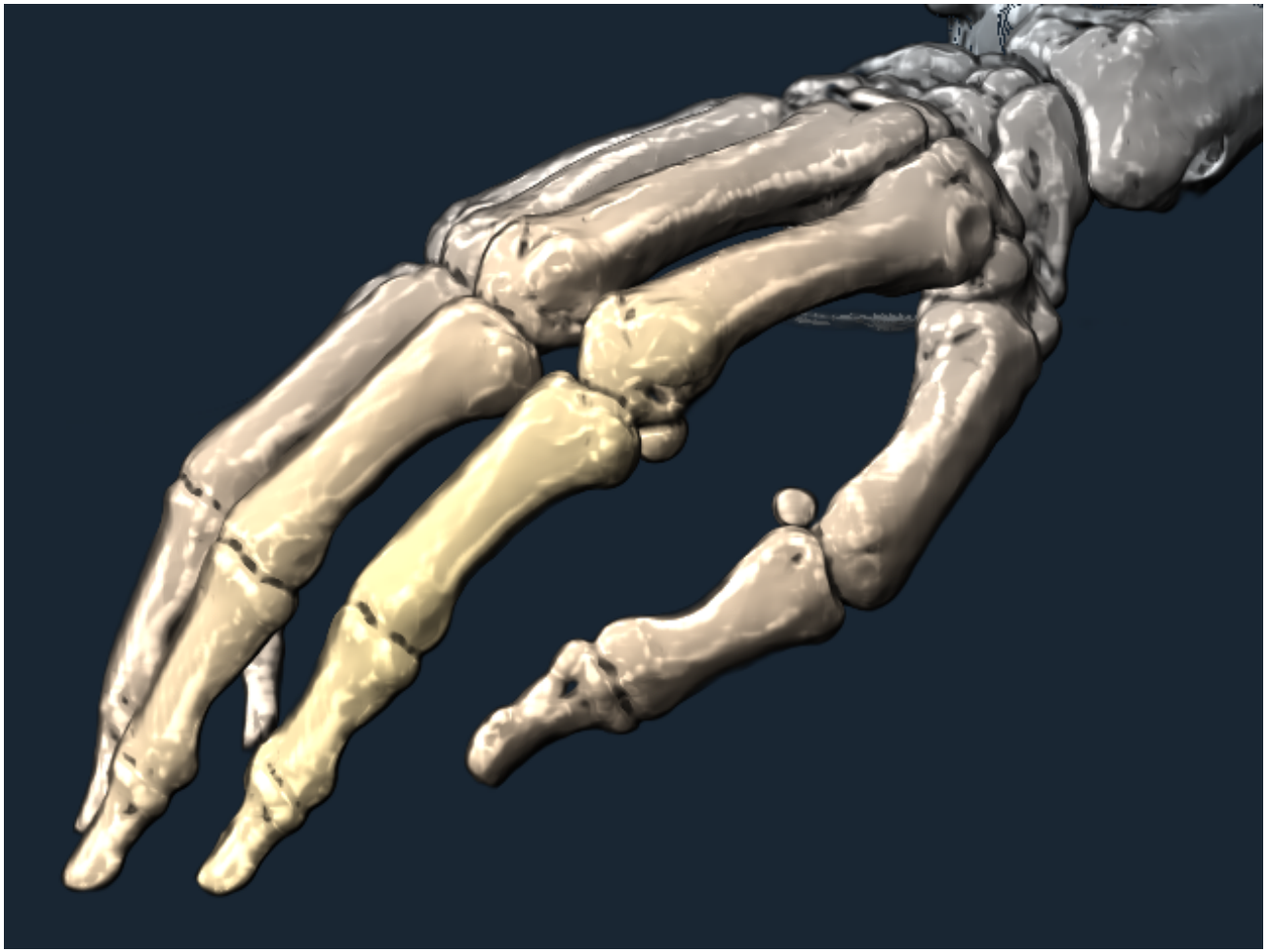
In the future, we would like to generalize this work in two directions:

- Extend Diderot to other classes of algorithms (*e.g.*, object recognition).
- Generalize approach to other domains.

Conclusion

Domain-specific languages can provide both high-level notation and portable parallel performance.

Questions?



`http://diderot-language.cs.uchicago.edu`