Diderot A Parallel Domain Specific Language for Image Analysis and Visualization

THE UNIVERSITY OF CHICAGO COMPUTER SCIENCE

Lamont Samuels, Nicholas Seltzer, Charisee Chiw, John Reppy, Gordon Kindlmann Department of Computer Science; University of Chicago http://diderot-language.cs.uchicago.edu

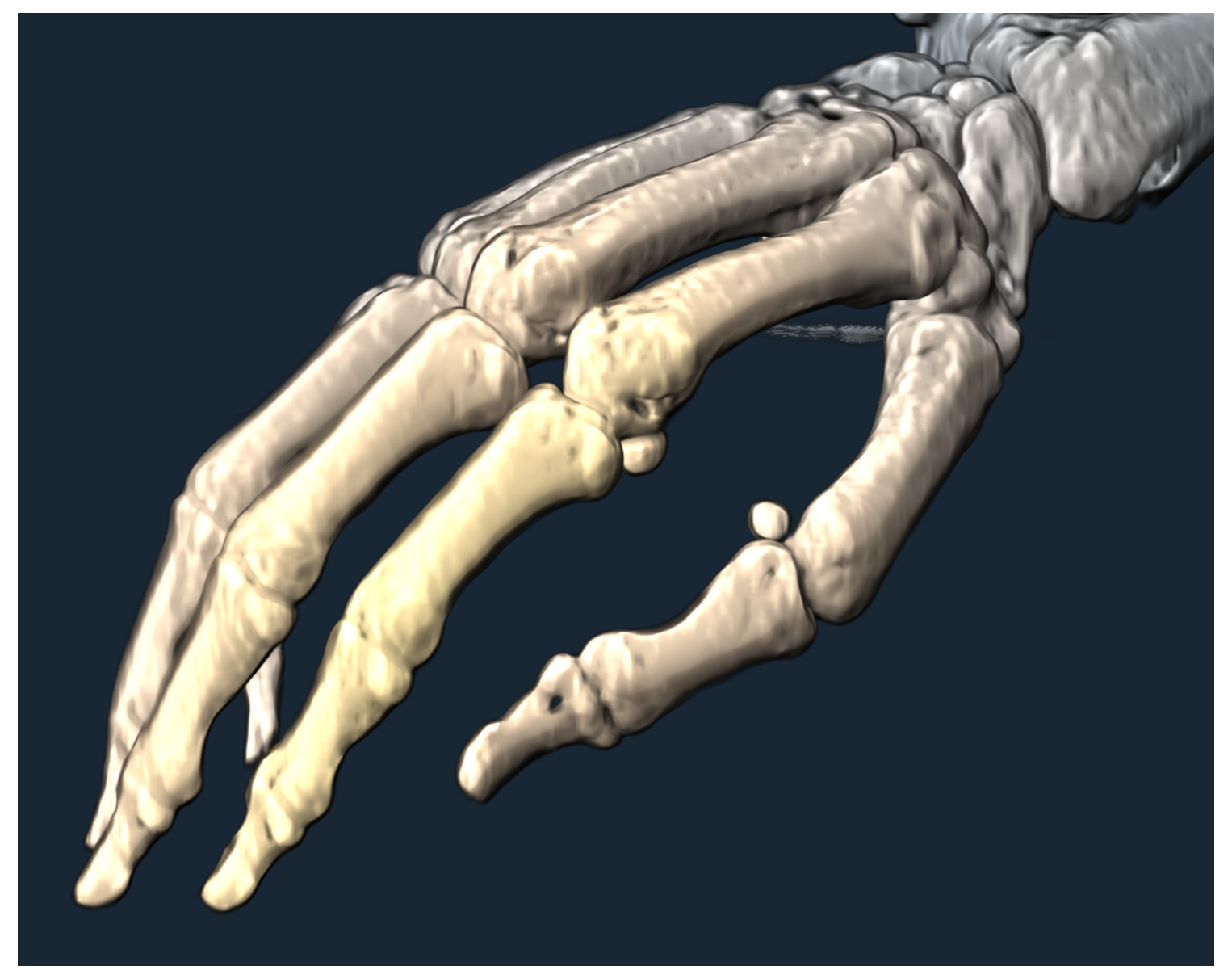
Introduction

Diderot is a parallel domain-specific language that is designed for biomedical image-analysis and visualization algorithms and provides a high-level mathematical programming model [2]. Diderot allows domain experts to implement familiar image analysis and visualization algorithms directly. The mathematical style of Diderot also makes it a great candidate for educational settings where students may not have time to learn more complex methods. Since Diderot is a domain-specific language, it achieves good performance on a range of parallel platforms, without requiring knowledge about parallel programming. This removes the burden of forcing programmer to learn low-level details of various target platforms but rather focus on implementing their algorithms.

Parallelism model

- A Diderot program is composed of a collection of autonomous strands (lightweight threads) that perform independent computations on a continuous tensor field
- Each strand has its own local state and an update method, which encapsulates the computational kernel of the algorithm
- All strands have access to global variables, including input images and tensor fields
- Computational kernels are expressed using common concepts and direct-style notation of tensor calculus. For example:
    - Tensor Operations (+, -, •, ×, ⊗)
    - Tensor Field Operations (+, -, ∇, ∇×, ∇⊗)
    - Convolving (⊛) images with various univariate kernels (e.g., tent and bspln3)
- Programs execute in a bulk synchronous fashion with each strand updated in each step
- Strands terminate by either
    - stabilizing, in which case their state contributes to the output
    - dying, in which case the strand's state is discarded

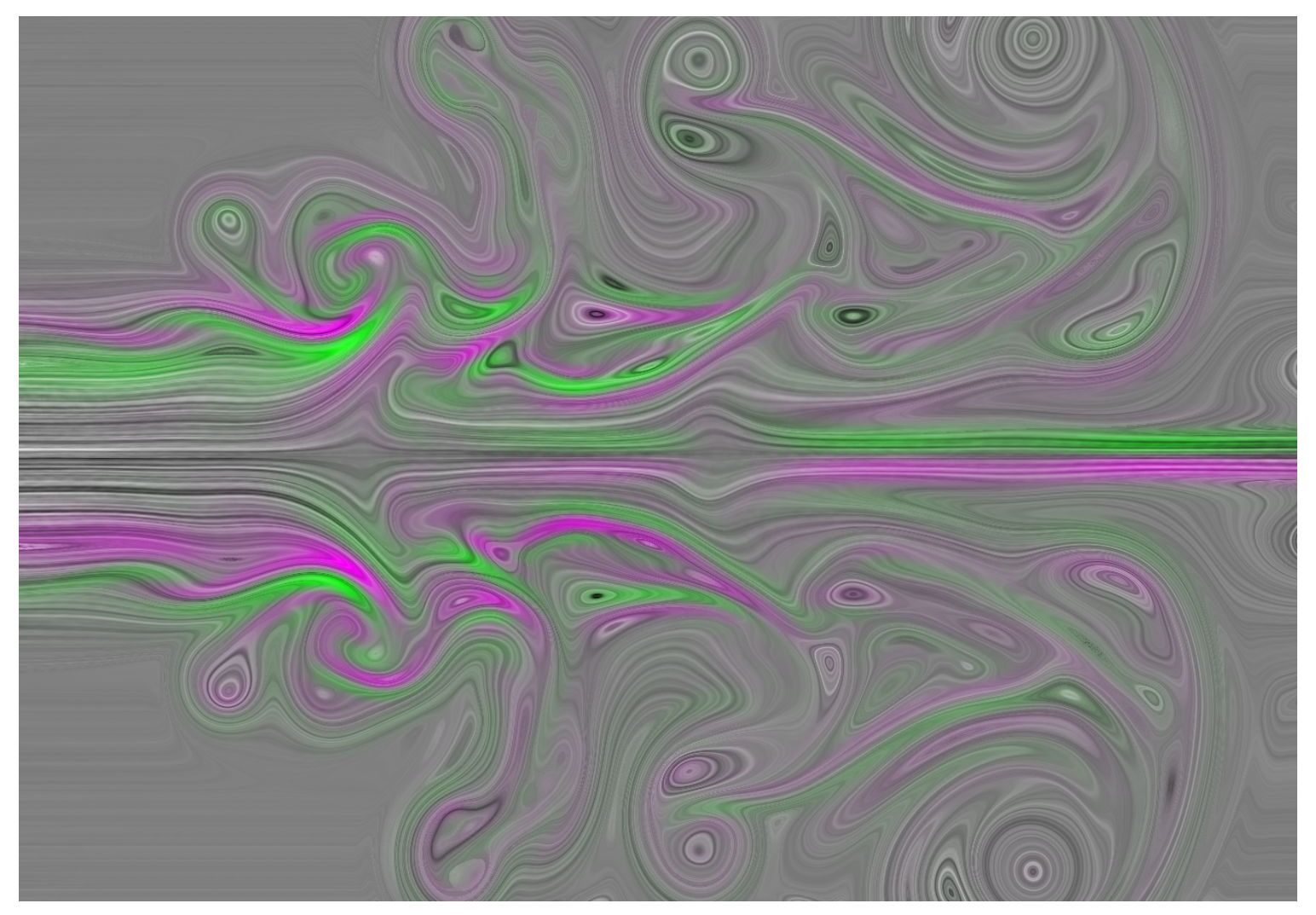
Example Applications



Illustrative volume rendering of a CT scan [3, 4]. This illustrative volume rendering of a CT scan of a human hand is a typical example of a possible application for Diderot. Every strand computes a separate ray integral in parallel, one per pixel of the output image. The final image was colored with a curvature-based transfer function involving second derivatives of the data.

Line Integral Convolution [1]. This LIC was created using the midpoint integration method, with coloration indicating curl. The pieces of Diderot code shown below were selected to illustrate the structure of the program used to create this LIC.

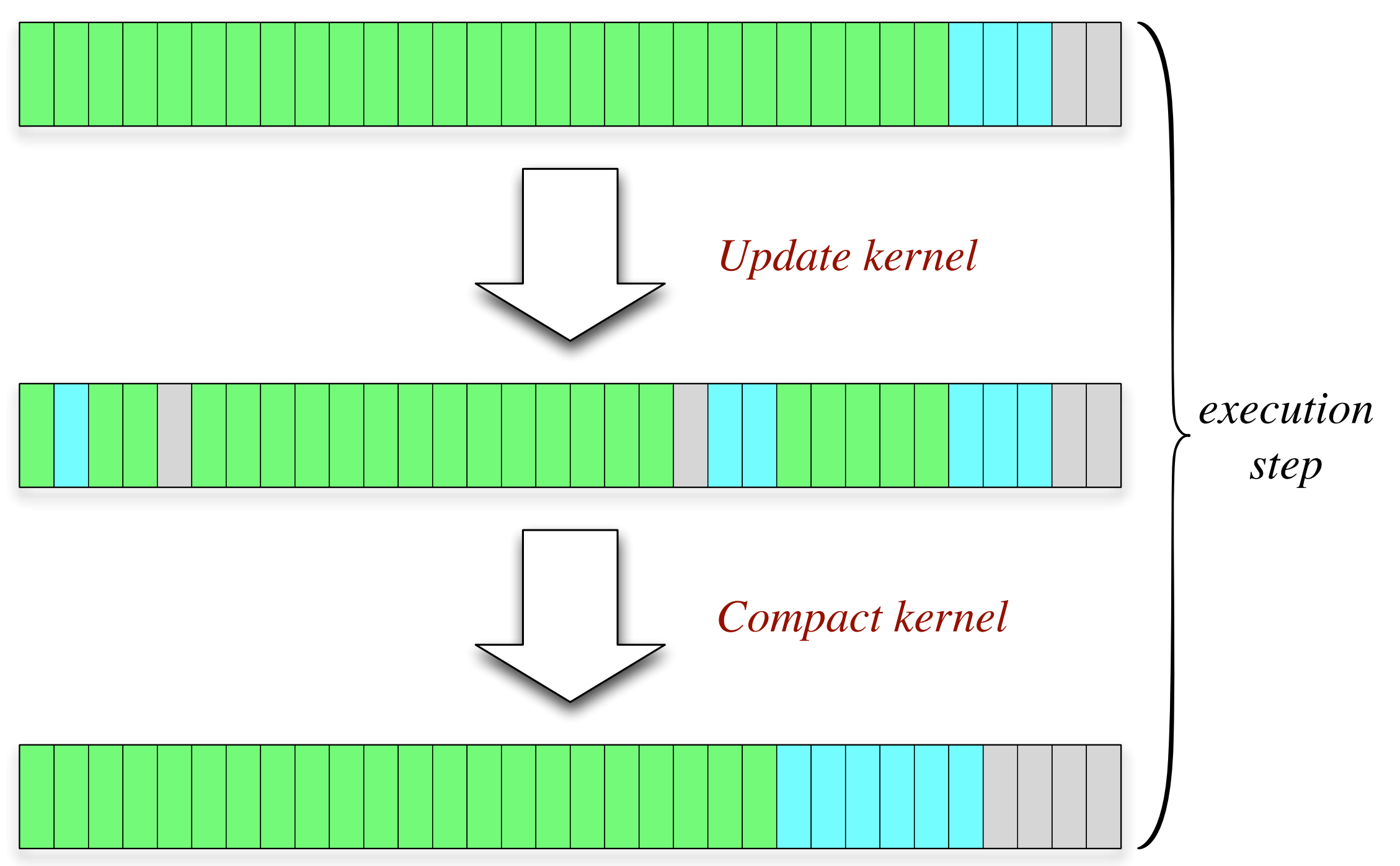
```
field#2(2)[2] V = load("vectors.nrrd") ⊛ bspln3;
field#0(2)[] R = load("noise.nrrd") ⊛ tent;
strand LIC (int xi, int yi) {
   vec2 p = [xi,yi];
   output real sum = 0;
   update {
      p += h*normalize(V(p +
            0.5*h*normalize(V(p))));
       sum += R(p);
      if (step == stepMax) {stabilize;}
   }
}


initially [ LIC(xi, yi) | yi in 0..(imgSizeY-1), xi in 0..(imgSizeX-1) ];
```



Program Execution

Diderot currently supports multiple backends (vectorized C and OpenCL) and runtimes (Sequential C, Parallel C, and OpenCL). The sequential C version executes on a single CPU core, while the parallel C runtime executes with threads on multiple CPU cores. The GPU runtime uses the OpenCL API and is discussed in further detail below.

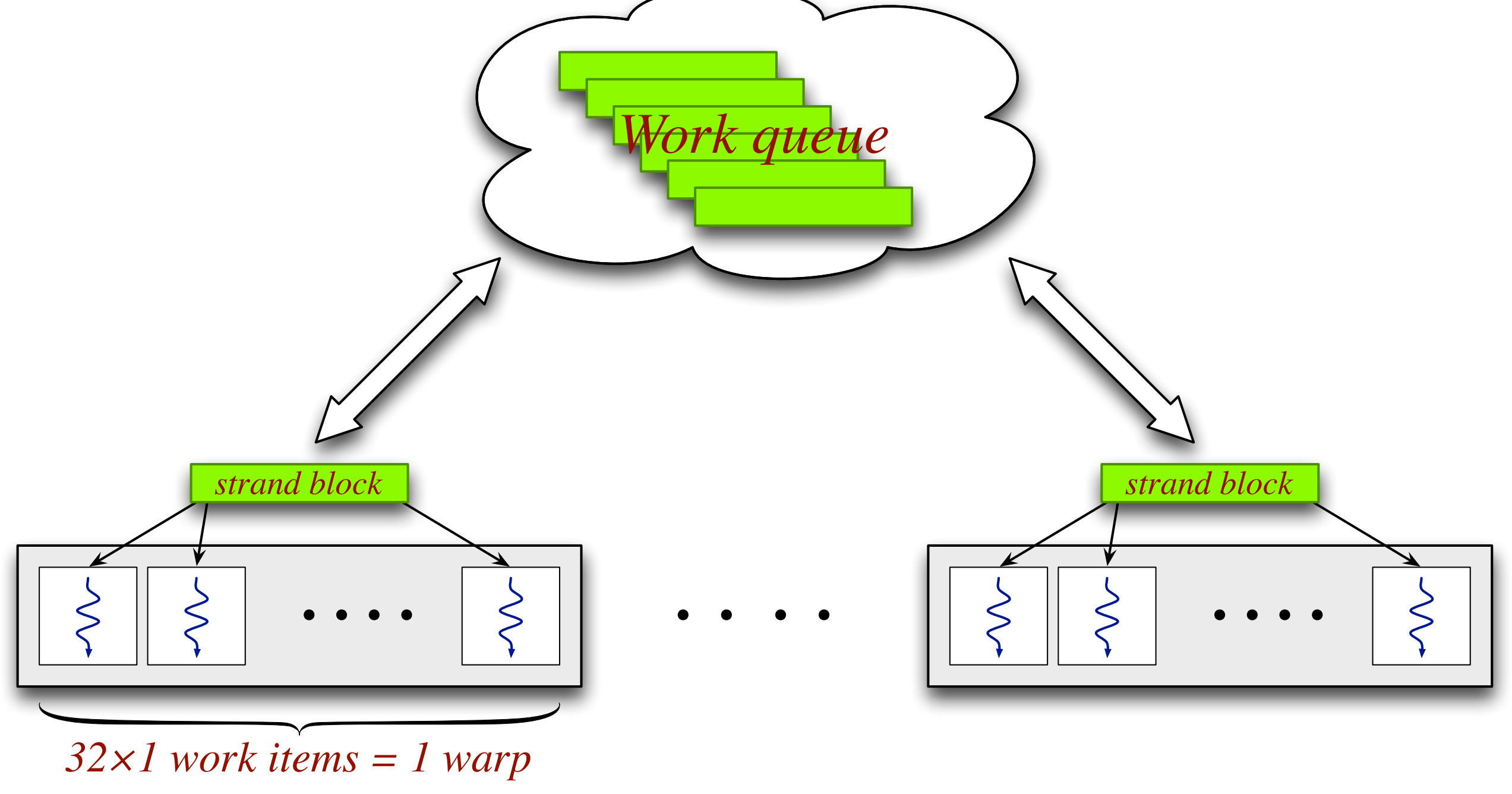
The host side converts all necessary image data and strand data into their OpenCL representations and loads them onto the GPU. The figure below illustrates one execution step for the GPU implementation. The update kernel calls the update function on each active strand (colored in green). After execution, strands may stabilize (colored in blue), die (colored in grey) or remain active. Since the update functions for stable and dead strands are not executed in future iterations, there is a potential for divergence. We solve this problem by running a compaction kernel after the update kernel. The compaction kernel handles reordering strands within their blocks so active strands remain in the front and stable and dead strands are placed at the end.



Persistent Thread Scheduler

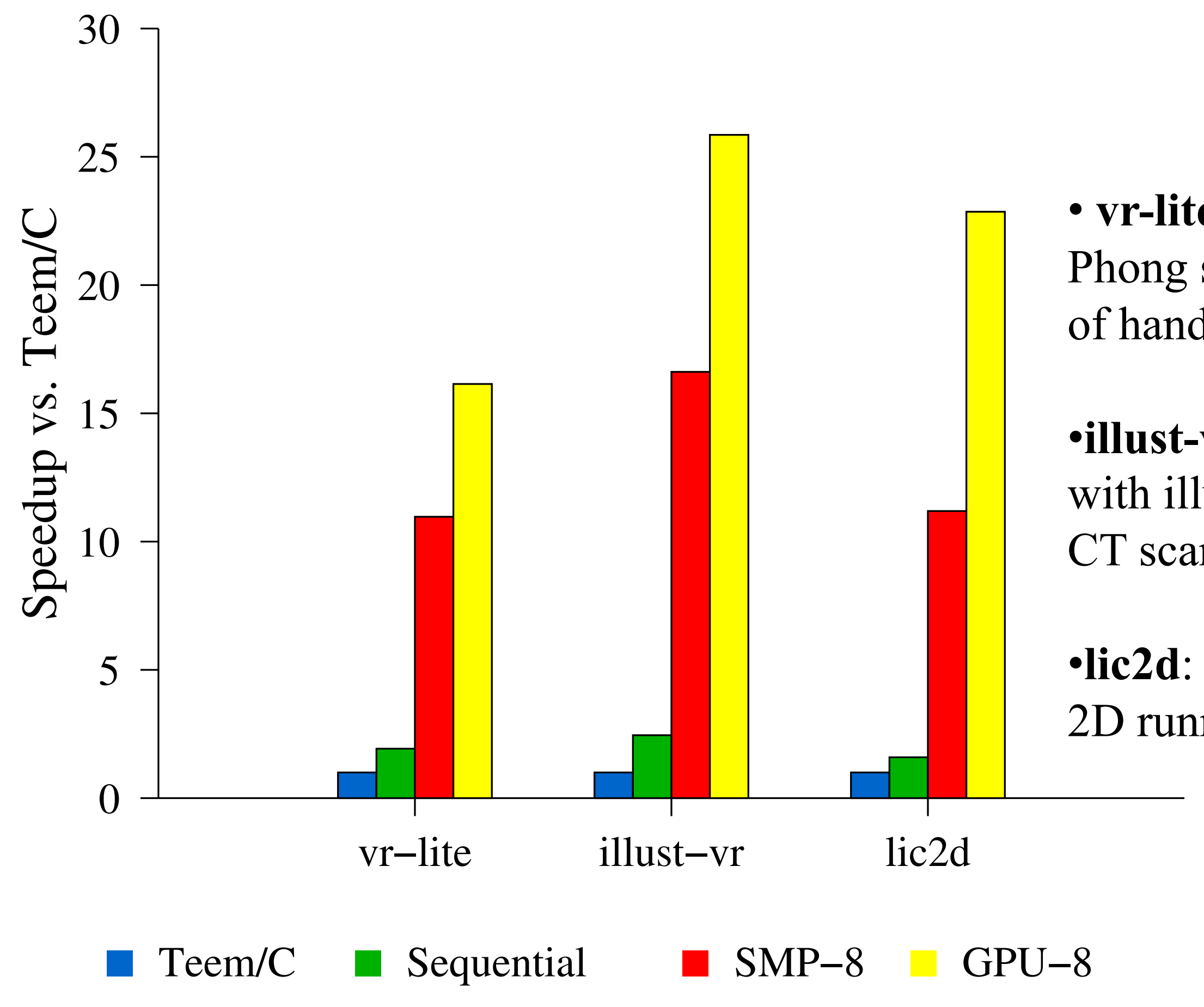
One of the challenges with developing Diderot for GPUs was determining how strands would execute on the GPU to incur good performance. Initially each strand was mapped to a single work-item with each workgroup having the size of a single warp. This direct mapping works well when strands stabilize synchronously because this limits the number of idle work-items. But this approach does not scale well for other algorithms Diderot will support, such as fiber tractography and particle systems. With these algorithms, irregular workloads (i.e. a mixture of active, stable, and dead strands) can occur. The current implementation now includes the idea of persistent threads [5]. The idea is to launch enough work-items to fill the machine once and keep them alive for the entire kernel execution. This allows workgroups to continuously grab work off a queue within the same kernel call and avoids the need for global synchronization.

As shown below, each workgroup runs a warp-wide strand block. Each strand block contains a warps worth of strands and meta-information about the number of active, stable, and dead strands. Workgroups will execute the update methods for each strand in the strand block. Once all work items are done executing, the workgroup grabs another warp-wide strand block until there are no remaining blocks.



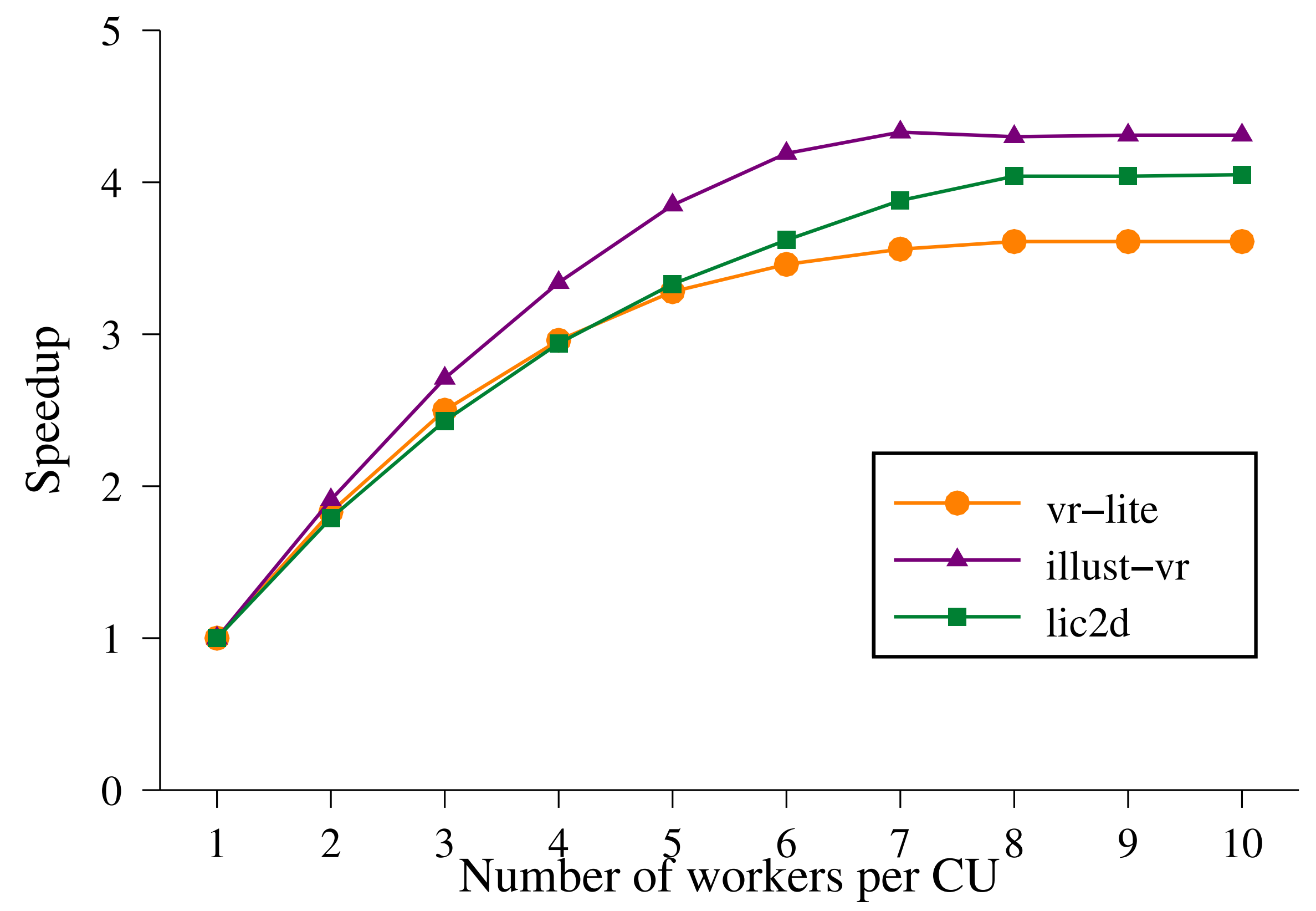
Performance

Our test platform are an 8-core MacPro with 2.93 GHz Xeon X5570 processors (SSE-4), and a Linux machine box with NVIDIA Tesla C2070. The graph below compares four versions of benchmarks: Teem/C, Sequential Diderot, Parallel Diderot, and GPU Diderot.



- **vr-lite**: simple volume-render with Phong shading running on CT scan of hand
- **illust-vr**: fancy volume-renderer with illustrative shading running on CT scan of hand
- **lic2d**: line integral convolution in 2D running on synthetic data

To hide memory latency, we run multiple workgroups per GPU compute unit and get even better performance. These can also be adjusted dynamically by the runtime.



Future Work

- Add a CUDA backend
- Expand the range of supported algorithms and targeted platforms
- Allow strands to interact and communicate with each other
- Allow dynamic creation of strands
- Add a global computation phase
- Type inference and dimension polymorphism

REFERENCES
[1] B. Cabral and L. C. Leedom. Imaging vector fields using line integral convolution. In Proc. SIGGRAPH, pages 263–270, 1993.
[2] C. Chiw, G. Kindlmann, J. Reppy, L. Samuels, and N. Seltzer. Diderot: A Parallel DSL for Image Analysis and Visualization. To appear in Proc. PLDI, 2012.
[3] R. A. Drebin, L. Carpenter, and P. Hanrahan. Volume rendering. In Proc. SIGGRAPH, pages 65–74, 1988.
[4] G. Kindlmann, R. Whitaker, T. Tasdizen, and T. M¨oller. Curvature based transfer functions for direct volume rendering: Methods and applications. In Proc. Visualization 2003, pages 513–520, 2003.
[5] Hoberock et al. 2009; Parker 2010; Wald 2011