Fast Coherent Particle Advection through Time-Varying Unstructured Flow Datasets

Mingcheng Chen and John C. Hart, Member, IEEE-CS

Fig. 1. A top down view of 40K particles (orange, lower right) traced over 40K fourth-order Runge-Kutta integrations through an unstructured unsteady (pulsatile) flow dataset of the abdominal aorta consisting of 1M cells and nearly 194K nodes each containing 25 velocity samples, colored by velocity (blue low to red high), into the femoral arteries (left). A recently published synchronous GPU algorithm [1] traces these particles in 59 s whereas our new GPU asynchronous blocked advection traces these particles in 34 s.

Abstract—Tracing the paths of collections of particles through a flow field is a key step for many flow visualization and analysis methods. When a flow field is interpolated from the nodes of an unstructured mesh, the process of advecting a particle must first find which cell in the unstructured mesh contains the particle. Since the paths of nearby particles often diverge, the parallelization of particle advection quickly leads to incoherent memory accesses of the unstructured mesh. We have developed a new GPU approach that reorganizes particles into spatially coherent bundles as they follow their advection paths, which greatly improves memory coherence and thus shared-memory GPU performance.

Index Terms—Flow topology, GPU, Lagrangian coherent structures, parallel algorithm.

1 INTRODUCTION

The analysis and visualization of flow datasets often rely on tracing the paths of collections of particles through the flow. This process is key to the production of standard visualization tools for steady and unsteady flow datasets including streamlines, streaklines, pathlines and rakes.

This particle path tracing process is also key to a recent approach for visualizing the topological structure of an unsteady flow called Lagrangian coherent structure (LCS). LCS visualizes the dynamic boundaries separating individual flow structures (e.g., vortices) in a time-varying flow. These boundaries are computed as ridges of the finite-time Lyapunov exponent (FTLE), which is a field of values indicating how long it takes for pathlines of nearby particles to diverge. While tracing small collections of particles suffices for visual effects [16], scientific studies of LCS requires significant computing.

A recent LCS analysis of revolving door heat loss [14] discussed the computation of their flow map, a map of start and end points of particle traces. In their case, they had a 1.65M cell unstructured grid whose nodes each held 200 time-varying velocities. The tracing of 1.34M particles for their flow map took 37 hours to compute on a pair of quad-core CPU’s. They stated that “the computation of the flow map is by far the most expensive part for any FTLE-based visualization,” and later “As future work, looking for ways to speed up the flow map computation is an obvious goal as it is computationally by far the most expensive part.”

In this paper, we investigate fast parallel many-core (GPU) algorithms for tracing particles through an unstructured mesh of time-varying flow velocities. We show how to reduce register utilization through a multi-stage Runge-Kutta integration, that a particle asynchronous approach leads to better throughput than does a recently published particle synchronous approach, and develop a block advection approach that localizes memory access to overcome the cache misses that degrade the performance of other methods as particles disperse across the flow domain.

Our result is a fast memory coherent particle advection engine for tracing particles through time-varying unstructured mesh flow datasets. The revolving door LCS flow map computation required four
minutes to trace its 1.34M particles for one of its 280 time frames. In less than four minutes, our GPU block advection algorithm can trace 8M particles over 10K integration steps.

Section 2 reviews recent work on high performance algorithms for integration and particle advection through flow fields, reviewing work on steady and unsteady flow, and for regular grids and unstructured meshes. Section 3 summarizes the basic kernel for unstructured mesh particle advection, including a multistage Runge-Kutta algorithm that reduces register utilization for faster GPU computing. Section 4 describes synchronous and asynchronous algorithms for advecting particles, and introduces the block advection method for asynchronous particle tracing. Section 5 analyzes the performance of block advection and provides specific heuristics for setting its constants. For example it finds a performance sweet spot when the block width is about five times the average cell width. Section 6 measures the performance of block advection and shows it outperforms recently published synchronous and asynchronous GPU particle advection methods. Section 7 concludes with a discussion of limitations and directions for further research.

2 Previous Work

Kruger et al. [10] showed the GPU could be used as an effective processing platform for particle advection in a full visualization system. This and related work [9, 10] focused on advecting particles through regular rectilinear grids.

We assume massless non-interacting particles. Our approach would not be suitable for other GPU particle simulations, e.g. SPH [7].

Burger et al., [2] used a vertex shader to trace dense particles through a uniform grid to create streak surfaces, controlling the particle density as the streak surface particles converged and diverged. A modern implementation might also use the tessellation shader for further particle population control. Fersl et al. [6] combined these GPU streak surfaces with GPU ridge extractions of the FTLE to compute LCS-like structures over a regular grid.

Data management and scheduling are key to scalable parallel implementation, especially when dealing with the large datasets used for unsteady flow fields. Burger et al. [3] used the CPU to page in new time intervals of the entire spatial extent of regular-grid unsteady flow data to support GPU’s particle integration. We similarly work with one time interval of unsteady data, linearly interpolating velocities from two velocity samples per node.

Irregular mesh, e.g. tetrahedral, data poses an additional challenge, as each particle must locate the mesh cell containing it to properly interpolate its advecting velocity. Often a k-D tree is used to initially assign cells to particles [8], and a neighborhood search finds the cell into which a particle has advected. Schirski et al. [15] combined GPU advection with CPU cell location, but requires costly CPU-GPU communication. Butler et al. [4] implemented a GPU cell location algorithms for neighborhood search [15] as well as GPU algorithms for exact and fast-approximate traversal of a k-D tree precomputed on the CPU. They showed that an approximate but divergence-free single pass through the k-D tree reduced the neighborhood search enough to justify its use. We rely on the stability measure that the time step is set low enough that velocities do not cause particles to move farther that one or two cells away.

Martin et al. [12] examined optimizations to cell walking when integration often moves particles farther than one cell away. Macpherson et al. [11] describes an algorithm for robustly walking through neighboring cells in locally non-manifold unstructured meshes with small overlaps and holes.

The GPU can also be applied to the general integration of ordinary differential equations in addition to particle tracing. Murray [13] implements reduced-memory versions of DOPRIS and RR(3,5)(2R+1)C, using an asynchronous approach for better load balancing. Our multi-stage Runge-Kutta integration is similar to this work, though Murray sought to minimize the storage of all constants whereas we focus specifically on reducing register usage.

3 Steaming Advection Kernels

Unstructured unsteady flow data is stored as synchronized time-sequences of velocity vectors stored at the nodes of a mesh. The mesh data structure is cell-based with connectivity information that permits easy access to neighboring cells. We work in 3-D, and the 3-cells (which we simply refer to as ‘cells’) are typically tetrahedra, though our approach can handle any convex polyhedral cell shapes.

We process as input the initial positions (typically organized in a regular rectilinear lattice) of particles to advect through the time-varying velocity field interpolated in space and time from the unstructured mesh nodes.

The advection of particles through an unsteady (time-varying) flow measured at the nodes of an unstructured mesh follows in general a repeating pattern of computation consisting of particle location, velocity interpolation and particle integration shown in Alg. 1.

Algorithm 1: Unstructured Particle Advection Kernel

```
repeat
  Location ;
  Interpolation ;
  Integration ;
until done;
```

3.1 Location

The task of locating a particle in an unstructured mesh determines for a given 3-D particle position \( x_i = (x_i,y_i,z_i) \), which cell in the unstructured mesh contains it. Each (convex) cell \( j \) of an unstructured mesh is the intersection of \( K \) half spaces bounded by the oriented planes \((A_{jk},B_{jk},C_{jk},D_{jk})\), using the function

\[
f_{jk}(x_i) = A_{jk}x_i + B_{jk}y_i + C_{jk}z_i + D_{jk}
\]

for \( k = 1 \ldots K \), such that \( x_i \) is in a cell if \( f_{j,1..K}(x_i) \geq 0 \). We assume \( A_{jk}^2 + B_{jk}^2 + C_{jk}^2 = 1 \) for all \( j,k \), such that (1) returns a comparable signed Euclidean distance from the particle position to the plane.

Neighboring cells can share a face, and by this definition the particle would be in both of these cells. In such cases, either cell can be used since we will be linearly interpolating velocity from cell vertices and the interpolated velocity for a point on a shared face only depends on the (shared) vertices of the shared face. The function \( \text{neighbor}(k) \) accesses the mesh adjacency data structure to return the index of the cell sharing face \( k \) with cell \( i \).

For each particle \( i \) we maintain a value \( j \) indicating the index of the cell currently containing the particle. When a particle position is moved, we check the signs of \( f_{j,1..K}(x_i) \). If any are negative, we need to update the cell for particle \( i \). The “walking” method [8] performs a local search for the cell containing position \( x_i \). If the current cell \( j \) does not contain particle \( i \) we find the plane \( k \) whose distance \( f_{jk}(x_i) \) is the most negative, and set the cell sharing plane \( k \) as the new current candidate to contain particle \( i \).

Algorithm 2: Cell Walking Particle Location [8]

```
Result: Updates \( j \) to a cell containing particle \( i \).

\[ k^* = \arg \min_k f_{jk}(x_i) \]

if \( f_{jk^*} < 0 \) then
  \( j = \text{neighbor}(k^*) \);
repeat;
```

Global location methods find a cell that either contains the particle or is nearby the particle, often accelerated by a spatial query data structure. Such global approaches might be appropriate for high velocity domains where particle integration tends to step across many cells. However, the stability of a flow simulation usually relies on setting the time resolution for integration consistent with the spatial resolution of
Abdominal Aorta
1,004,926 tetrahedra
193,840 nodes
25 velocities/node

UV (Upper Vasculature)
798,251 tetrahedra
165,547 nodes
25 velocities/node

TCPC (Total Cavo-Pulmonary Connection)
1,482,023 tetrahedra
272,574 nodes
100 velocities/node

Fig. 2. Three datasets used in our algorithm development

the mesh, so we expect particles infrequently move more than one cell away.

A global location method is also needed to determine the initial cell location of seed particles. In practice, however, seed particles often lie in a regular grid arrangement. Hence we propose the following parallel algorithms for finding the cell location for the seed particles.

Algorithm 3: Initial Cell Location

\[\text{input : } P \text{ seed particles, } M \text{ cells and } N \text{ threads, where } P \gg N.\]

Divide the \(P\) seed particles into \(N\) buckets:

\[
\text{foreach bucket } i \text{ in parallel do}
\]

\[
\text{let } x_i \text{ be the first particle in bucket } i;
\]

\[
\text{foreach cell } C_j \text{ do}
\]

\[
\text{test } x_i \in C_j;
\]

\[
\text{foreach bucket } i \text{ in parallel do}
\]

\[
\text{let } j \text{ be the cell containing the first particle } x_i;
\]

\[
\text{foreach other particle } x_k \text{ in bucket } i \text{ do}
\]

\[
\text{perform cell-walk (Alg. 2) on } j \text{ to locate } x_k;
\]

Alg. 3 distributes the seed particles into buckets, with one bucket for each thread. It then performs a synchronous serial sweep through all of the cells, with each thread determining if the current cell contains the first particle in that cell’s bucket. At the end of the sweep, the first particle in each bucket has located its containing cell. The algorithm then runs the cell walking scheme in parallel on the remaining particles in each thread’s bucket, updating the cell index \(j\) from the last particle to the cell containing the current particle. Since the initial particle positions are usually ordered and tightly clustered, this spatial coherence means the particles is usually in the previous particle’s cell, or lies in a close neighboring cell.

The most expensive part of Alg. 3 is the serial cell sweep in the first loop, with a total running time of \(O(P + M)\) assuming the number of cell walk steps in the worst case is bounded by a small constant. Parallel methods for building k-D trees \([17, 5]\) require \(O(M \log \frac{M}{N})\) construction time. Since we only need one global search to locate the seed particles, we do not use the improved \(O(\log M)\) search, and its benefits do not justify the construction cost.

Algorithm 4: Initial Cell Location for AA Grid Particles

\[\text{input : } P \text{ seed particles, } M \text{ cells and } N \text{ threads, where } P \gg N.\]

\[
\text{foreach cell } C_j \text{ in parallel do}
\]

\[
\text{let } B \text{ be the axis-aligned bounding box of } C_j;
\]

\[
\text{foreach grid particle position } x \in B \text{ do}
\]

\[
\text{test } x_i \in C_j;
\]

One limitation of Alg. 3 is that the flow domain must be convex, whereas a full global search would robustly place particles in cells in a non-convex domain. When the initial particles lie on a regular axis-aligned grid, Alg. 4 is faster, simpler to implement and will work in concave domains. This alternative algorithm streams through the cells in parallel and because of the regular grid particle structure can enumerate the particles in the cell bounding box, outputting only those particle positions in the current cell. This results in an improved \(O(M + P)\) time complexity if we assume the number of overlapping cell bounding boxes is constant. (In the worst case the overlap can be proportional to \(M\) since smaller long-skinny cells can incur more overlap, but this is not a realistic property for typical unstructured mesh.)

3.2 Interpolation

We assume the unstructured mesh consists of tetrahedra. For cell (tetrahedron) \(j\), we denote the vertex with index \(k\) as the vertex not in the plane with index \(k\). We estimate the velocity \(v\) at position \(x\) by the barycentric (linear) interpolation of the velocities \(v_k\) at the cell’s
four node positions \( x_k \) as

\[
v(x) = \sum_k \alpha_{jk} v_{jk}, \quad \alpha_{jk} = \frac{f_{jk}(x)}{f_{jk}(x_k)}.
\]

### 3.3 Integration

Numerical integration methods, such as fourth-order Runge-Kutta (RK4), consist of repeated scaled Euler integration steps,

\[
v_1 = v(x, t),
\]

\[
v_2 = v(x + \Delta t \frac{v_1}{2}, t + \Delta t \frac{v_2}{2}),
\]

\[
v_3 = v(x + \Delta t \frac{v_2}{2}, t + \Delta t \frac{v_3}{2}),
\]

\[
v_4 = v(x + \Delta t v_3, t + \Delta t),
\]

\[
x(t + h) = x(t) + \frac{\Delta t}{6} v_1 + \frac{\Delta t}{3} v_2 + \frac{\Delta t}{3} v_3 + \frac{\Delta t}{6} v_4.
\]

Such multi-stage integration techniques require multiple location and interpolation calls to find the velocities at the intermediate points, in this case \( x(t), x + \frac{\Delta t}{2} v_1, x + \frac{\Delta t}{2} v_2 \) and \( x + \Delta t v_1 \).

For RK4, the integration step might be as shown in Algorithm 5. Hence, the RK4 integration of a particle required four iterations through the location-interpolation-integration kernel, such that the intermediate Runge-Kutta points can be located and their velocities interpolated.

**Algorithm 5: Streaming Reentrant RK4**

**Data:** \((x_0, x, t, v, \text{stage})\) where

- \( x_0, x_1 \) maintain the original and accumulated particle positions,
- \( x, t \) are the current integration sample position and time,
- \( v \) is the velocity interpolated at position \( x \), and
- “stage” is the number of the current integration stage.

**Result:** Updated stage and new integration position \( x \) and time \( t \).

**switch(stage)**

- **case 1:** \( x_0 = x, \quad x_1 = x + \frac{\Delta t}{6} v, \quad x = x_0 + \frac{\Delta t}{2} v, \quad t = t + \frac{\Delta t}{2} \).
- **case 2:** \( x_1 = x_0 + \frac{\Delta t}{3} v, \quad x = x_0 + \frac{\Delta t}{2} v; \)
- **case 3:** \( x_1 = x_0 + \frac{\Delta t}{3} v, \quad x = x_0 + \Delta t v, \quad t = t + \frac{\Delta t}{2} \);
- **case 4:** \( x = x_1 + \frac{\Delta t}{6} v; \)

\( \text{stage} = (\text{stage mod 4}) + 1; \)

### 4 Streaming Advection Algorithms

We target the modern GPU as the computational platform for high-performance unstructured mesh particle advection. In general, the GPU is a many-core processor that distributes parallel threads onto multiple MIMD warps of typically 32 SIMD parallel processors, and multiple warps are collected into SM processing units that share local memory and draw from the same pool of registers. Register contention commonly hinders GPU performance by limiting the number of threads in flight, so we strive to reduce register utilization.

#### 4.1 Synchronous Advection

Ameli et al. [1] implemented synchronous advection, which keeps all particles synchronized as they advect through a flow field, as shown in Alg. 6. Since there are typically more particles than processors, particles are loaded from global memory, advected one time step and written back to global memory so other particles can similarly be advanced one time step to the same time.

#### 4.2 Asynchronous Advection

Particle synchronous advection has several advantages. The location-interpolation-integration kernel are separated into three individual subkernels, each run on all particles before moving to the next (e.g. all particles find their containing cell before any interpolate their velocity). This improves performance by reducing register usage since some subkernels use less registers that others. Synchronization also reduces code divergence since particles all run the same subkernel, and all particles will share the same integration state in the integration subkernel.

The main disadvantages of synchronous particle tracing are load balancing and global memory access. Each local locate subkernel (Alg. 2) runs for a small but unknown number of iterations, so some instantly located particles may need to wait for others to find the right neighboring cell that contains them. As stated earlier, there are more particles than processors, so particle state needs to be fetched from global memory and results written back to global memory intermittently in the subkernel parallel loops.

An additional global memory disadvantage is that as particles advect away from their initial tight configuration and explore different regions of the flow domain, the synchronous algorithm’s locate and interpolate subkernels require incoherent global memory fetches of the flow data that quickly overflow the limited GPU memory cache.

#### 4.3 Block Advection (BA)

The asynchronous approach assigns a particle to a thread, and lets that thread trace the particle from its initial position until it reaches the space (outflow) or time (interval) boundary of the flow data. Such approaches can benefit from spatial coherence if neighboring particles are assigned to processors that share the same cached portion of the unstructured grid. However, particles that are initially proximate can
easy diverge, and any benefits from this initial cache coherence soon degrade.

We propose to preserve this cache coherence by dynamically re-clustering particles as they advect through the unstructured mesh, such that these clusters can share the same cached copies of the unstructured mesh flow data. Rather than clustering particles, e.g. by k-means, we organize the flow domain into overlapping rectilinear block, such that particles within a block utilize the same cached unstructured mesh flow data. These blocks are organized on a uniform structured spatial grid. Blocks that do not intersect the unstructured mesh are ignored.

![Figure 3](image)

Fig. 3. A block caches any unstructured flow cells (gray) that have a portion within the outer (solid) block boundary. Particles in the solid boundary are traced until they exit. The region between the solid boundary and the dashed inner boundary overlaps that of neighboring blocks. Hence particles are not added to this block’s processing queue until they enter the dashed inner region.

Particles within a block are traced until they leave the block as shown in Figure 3, and their advection performance benefits from the cached flow mesh data. When a particle exits a block’s boundary, it must be removed from the current block and added to a neighboring block for processing. In order to avoid alternating block processing for particles that straddle their boundary, we allow the blocks to overlap, such that either block can process the particle in this shared overlapping region.

![Figure 4](image)

Fig. 4. Blocks overlap spatially but not temporally. A single particle trace is computed in seven segments as it crosses various block domains.

The blocks also extend in time, from a flow data sample time \( t_0 \) to sample time \( t_1 \). Since the particles flow forward only, evenly in units of the time step, and the time step evenly subdivides the time interval between flow data samples, the blocks need not overlap in time.

Each block contains the unstructured mesh cells that intersect the block. Hence, the cells around the boundary of a block will also be cached by neighboring blocks. As block size decreases, and as the overlap margin increases, the number of cells duplicated in block caches increases. For example, the upper-left, lower-left and lower-right cells in Figure 3 would not be cached if the blocks did not overlap.

### Algorithm 8: Block Advection (BA)

```plaintext
foreach space-time block in shared-memory parallel do
  load block’s cells into shared memory
  foreach particle i in a parallel thread do
    repeat
      locate-interpolate-integrate particle i
    until particle i exits current space-time block
  transfer exiting particle i to neighboring blocks
```

This block advection algorithm is described in Alg. 8. It maps a given number (one or more) of particles to each thread in a thread block. The threads in the thread block trace the particles until all have escaped the block, or a preset maximum thread time has been reached. Particles that escape the block are labeled “inactive” and even though they may continue to be processed by their threads, they remain frozen at the place and time of their escape.

Each block maintains a local cache of the unstructured mesh cells that overlap its region in space, and this cache uses its own address space. When a particle crosses the (outer) boundary of the block, its next location procedure will need to take place in the neighboring block’s local address space. We store with each locally cached cell a link to that cell’s global address. We also maintain a global hash table that returns the blocks that overlap that cell, and the local address of the cell in each of those blocks. We set block size larger than the largest cell size to ensure the maximum number of blocks that overlap a cell is no greater than eight. (If an unstructured mesh contained cells larger then a block, one could subdivide the unstructured mesh appropriately as a preprocess, interpolating the flow data at the new nodes.)

This blocked approach introduces additional code divergence beyond that of the asynchronous algorithm it is based on. If a particle leaves its flow block in a middle stage in a multi-stage integration, it will start in that stage in its neighboring flow block even though other particles may be at different stages of integration. To reduce this divergence, we sort particles by integration stage before we begin processing each flow block. Experiments show that RK stage sorting improves tracing performance by about 13% even after adding the extra expense of the sort.

### 5 Analysis and Tuning

We implemented our algorithms on an NVIDIA Tesla K20 GPU, with 13 SMX multiprocessors each with 192 cores for a total of 2,496 cores. The SMX multiprocessors can each execute up to 64 warps sharing 64K registers, reaching a peak 1.17/3.52 T(dp/sp)flops. The K20 has 5GB of GDDR5 memory accessed at 208 GB/s, with 64KB available per SMX multiprocessor, organized as 48KB shared memory and 16KB L1 cache, or vice versa.

#### 5.1 Particle Block Density

Particles are processed in bundles that lie in the same flow block. The characteristics of the distribution of particles in blocks evolves through the tracing procedure. At the beginning of the advection, the particles will lie in a few flow blocks, but continued advection will distribute the particles across many more flow blocks. Hence the particles in a flow block are processed by one or more thread blocks to better balance computational load as the number of particles in each blocks begins to vary significantly.

Fig. 5 plots the number of active particles measured at the beginning of each of the 344 executions of Block Advection (BA, Alg. 8). Recall that this block advection algorithm is repeatedly run until there are no more blocks with particles that have not reached the end of the current flow time interval. Then the blocked flow data from the next time interval is used to block-adveクト the particles. The aorta flow dataset contains 25 such time intervals.

The number of particles decreases with each new time interval due to outflow. Each BA execution adveクトs particles until they either exit.
their current block or reach the end of the current time interval. Each of these iterations takes an amount of time proportional to the number of active particles times the number of time steps, and this is not indicated along the horizontal axes. In the first few intervals, most of the particles remain in a few initial blocks. The iterations at later intervals reveal that most particles reach the end of the time interval, leaving a short tail of iterations needed for particles that straddle the boundaries between neighboring blocks.

Figure 6 plots the number of blocks that contain at least one active particle, again over the multiple BA executions within each of the 25 flowdata time intervals. The number of active blocks grows with each new interval as the particles disperse across the flow domain. Within each interval, the number of active blocks quickly drops to a short tail of a few blocks involved in straddling particle trails.

Figure 7 plots the average number of active particles per active block. At early intervals, fewer blocks contain many particles, whereas at later intervals many blocks contain few particles. Within an interval, blocks also become less densely populated as block advection drives particles from their current block. The BA execution counts are the same as in Fig. 5.

We can measure the distribution as the average number of particles per block as

\[ A = \frac{\text{total active particles}}{\text{total active blocks}}. \]  

Fig. 7 plots A over the 344 BA executions within the 25 intervals of the aorta dataset. The particle block density drops as particles disperse across the flow domain. Initially particle block density is high since fewer blocks contain more particles, but later particle block densities fall to below 1,000 particles per block by the 15th interval.

### 5.2 Thread Assignment

An ideal implementation of the block advection algorithm would store the flow mesh data of a block in the local memory (shared memory or cache) of one streaming multiprocessor (SM) unit, and assign to it all of the particles in that block. However, current GPU architectures (including the K20) do not allow the user to assign a thread block to a specific SM. We compromise by mapping a thread block to a block, and letting the GPU scheduler assign the SM.

We set a threshold \( A^* \) (found experimentally to be 256) that we use to distinguish two kinds of blocked advection workloads. When \( A \geq A^* \), we assume we have few flow block with many particles. We set the maximum thread block size to \( A^* \) threads, and allow each thread to trace a maximum of \( PT = \lfloor A/A^* \rfloor \) particles to a maximum of \( PT = 16 \). Hence

\[ PT = \min\{A/A^*, 16\}. \]  

When \( A < A^* \), we assume many flow blocks contain a few particles. We set the maximum thread block size to \( \lceil A \rceil \) threads, and allow each thread to trace only a single particle, \( PT = 1 \). Hence, if \( A < A^* \) then a flow block of \( N \) particles is processed by \( \lceil N/A \rceil \) thread blocks of single-particle threads. Otherwise \( A \geq A^* \) and the flow block’s \( N \) particles are processed by \( P \) particles per thread organized into \( \lceil N/(PA^*) \rceil \) thread blocks.

Figure 8 shows that low particle density occurs often at the end of each interval to flush out straggling particles that likely straddle the overlaps between flow blocks. For the aorta dataset, 71 of the 344 AB executions (about 20%) run with an average particles per block less than the \( A^* = 256 \) threshold. As this plot reveals, the particle block density threshold \( A^* = 256 \) is set based on the characteristics of GPU computation and not the specifics of the dataset. Similarly, the particles per thread \( PT \) is set to evenly the number of particles up to a thread maximum and is also set independent of the characteristics of the flow data.

### 5.3 Block Size

Figure 9 measures running times for various block widths, plotted over their overlap settings as a percentage of block width. The block widths are measured in centimeters for a \( 11.4 \times 8.24 \times 26.6 \) cm flow dataset. The minimum occurs for an overlap of 30% (on each side of the block) of an inner block width of 0.15 cm (outer width of the block is thus 0.24 cm).

Figure 10 shows that running times are almost completely dependent on the outer block width. Table 1 shows that nearly any combination of inner width and overlap percentage seem to optimize performance on the aorta dataset so long as the outer width is about 0.25 cm on the aorta dataset, and our experiments verify similar dependence on outer width for other datasets. This dependence on outer width suggests performance is dependent on the memory footprint of the block, since the amount of local flow data cells depends on the outer width. Table 2 compares the average cell volume with the optimal outer block volume for three datasets (aorta, TCPC, UV) and finds that the optimal
outer block size hold a fairly consistent number of cells, roughly 125. Hence we conclude that the outer block width should be set to roughly the width of five average cells.

![Graph showing running time on the aorta dataset for various (inner) block widths plotted over various block overlap portions, as a percentage of block width.](image)

**Fig. 9.** Running time (s) on the aorta dataset for various (inner) block widths (cm) plotted over various block overlap portions, as a percentage of block width.

![Graph showing running time on the aorta dataset for various (inner) block widths plotted over outer block width. Percentage overlaps from 0% to 60% in 5% increments plotted from left to right on each curve.](image)

**Fig. 10.** Running time (s) on the aorta dataset for various (inner) block widths (cm) plotted over outer block width. Percentage overlaps from 0% to 60% in 5% increments plotted from left to right on each curve.

<table>
<thead>
<tr>
<th>inner width</th>
<th>overlap %</th>
<th>outer width</th>
<th>running time</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.10 cm</td>
<td>60%</td>
<td>0.22 cm</td>
<td>249 s</td>
</tr>
<tr>
<td>0.15 cm</td>
<td>30%</td>
<td>0.24 cm</td>
<td>233 s</td>
</tr>
<tr>
<td>0.20 cm</td>
<td>15%</td>
<td>0.26 cm</td>
<td>244 s</td>
</tr>
<tr>
<td>0.25 cm</td>
<td>0%</td>
<td>0.25 cm</td>
<td>264 s</td>
</tr>
</tbody>
</table>

**Table 1.** Optimal overlap percentages (run on aorta dataset) for various block inner widths yield the same outer width

### 5.4 Memory Allocation

The global flow data consists of geometric node data that record a spatial position and a time-varying sequence of flow velocities, and the topological data that connect nodes into a tetrahedral mesh. We order this data in memory clustered by blocks to facilitate cache coherence and coalesced memory access. A redundant copy of the node and cell data shared in the overlap region of neighboring blocks is stored in each of the overlapping blocks, which inflates the memory footprint of this representation.

![Graph showing table of optimal block width settings on various datasets leads to a fairly consistent cells/block value.](image)

**Table 2.** Comparison of optimal block width settings on various datasets leads to a fairly consistent cells/block value.

We use the shared memory to locally store the cells rather than the cache which might be overwritten if the GPU assigned two different flow block’s threads to the same SM. We assign each SM 48KB of shared memory and 16KB of L1 cache. We explicitly assign the flow data used by any flow block processed by the SM that fits in its shared memory limit. We have experimentally validated that the overhead of cache flow data degrades performance when we assign e.g. 48KB of L1 cache.

![Graph showing tracing time (s) for 8M particles over 10K RK4 timesteps divided among the flow dataset's 25 time intervals, using 48KB of SM processing memory as explicitly-loaded shared memory vs. as an L1 cache for block advection.](image)

Figure 11 shows that our blocked tracing approach using the shared memory for memory coherence outperforms the GPU’s default use of the L1 cache to accelerate access to local flow blocks. Our explicit shared memory use is about 2 1/4 times faster than letting the GPU use it as L1 cache.

We estimate the amount of shared memory needed for each thread block that must be shared among other thread blocks in the SM as

\[
M = \text{total shared memory} \times \min(A \times A^*) / 512 \quad (10)
\]

where 512 was found by experimentation. The memory size \(M\) can be used to determine if there is enough space to store a local version of the unstructured mesh, or if the global mesh should be used. For reasonable particle block densities \((A \geq A^*)\) we divide the shared memory between two flow blocks. When \(A \ll A^*\), we have low particle densities and the benefits of a local copy of the unstructured mesh cells is not worth the memory copying expense. Equation 10 avoids this memory copy expense by setting its available memory for each of many flow blocks sharing an SM to a value likely lower than needed to contain the block’s flow data.

### 5.5 Exiting Particle Transfer

The last step of Block Advection (Alg. 8) transfers any particles that have exited the current block to a neighboring block. This step requires finding the appropriate neighboring block. While the inner blocks are mutually exclusive, we have found particles can lie very near to block boundaries, and when cell faces also lie near inner block boundaries, numerical issues can interfere with finding the correct cell in the correct block. Hence we check to ensure the particle’s enclosing cell is
properly included in the block’s list of cells. If not, we examine the neighboring blocks (up to seven in corner cases) and use the first block that includes the block’s containing cell.

When a particle is transferred from one block to a neighboring block, its enclosing cell will need a new local cell index relevant to the new flow block. We implement a GPU hash table algorithm to quickly return the local block addresses of a global cell.

At this stage (end of BA execution), to improve performance we sort the particles in order of active/nonactive status, then by new block number, then by integration stage. The active status sort key results in a compaction so we can limit computation to active particles. The block assignment key coalesces particle data by block for better cached memory coherence. The integration state avoids thread divergence in the integration kernel for any particles that switched blocks in the intermediate stages of Runge-Kutta integration. We implement this sort using a prefix-sum scan to collect active particles, followed by a radix sort on block ID and integration stage.

## 6 Results and Comparisons

We compare GPU implementations of synchronous advection, asynchronous advection and blocked advection. The synchronous advection implementation is CudaVC [1], a well optimized particle tracer implemented to accelerate the investigation of Lagrangian coherent structures. Our asynchronous algorithm is a simple research prototype, though with the improved Runge-Kutta integration. The block advection implementation is analyzed and tuned as described in the previous section. Our asynchronous and blocked advection implementations are publicly available via [github.com/linyufly/CUDATracer](https://github.com/linyufly/CUDATracer).

### 6.1 Register Allocation

Table 3 compares the register utilization between algorithm implementations.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Registers</th>
<th>Stack Frame</th>
</tr>
</thead>
<tbody>
<tr>
<td>Synchronous</td>
<td>73</td>
<td>0</td>
</tr>
<tr>
<td>Asynchronous</td>
<td>83</td>
<td>32 bytes</td>
</tr>
<tr>
<td>Block</td>
<td>113</td>
<td>32 bytes</td>
</tr>
</tbody>
</table>

Table 3. Comparison of register utilization between streaming advection algorithm implementations.

### 6.2 Overall Performance

Figure 12 shows the execution time over each time interval of our three datasets of the synchronous, asynchronous and block advection implementations, and the total runtimes are compared in Table 4.

Figure 12 shows the execution time for the three advection approaches on the aorta, TCPC and UV datasets. As expected, the synchronous and asynchronous approaches drop in performance from their initial stages as particles disperse through the flow dataset, due to the associated loss in memory coherence. The block advection performance does not show such a decrease because the tuned blocks achieve the goal of maintaining local coherent memory access. All of these algorithms perform better toward the end due to outflow.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>steps</th>
<th>Synch.</th>
<th>Asynch.</th>
<th>BA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Aorta</td>
<td>10K</td>
<td>2,146</td>
<td>1,130</td>
<td>233</td>
</tr>
<tr>
<td>TCPC</td>
<td>50K</td>
<td>7,558</td>
<td>3,516</td>
<td>1,098</td>
</tr>
<tr>
<td>UV</td>
<td>9K</td>
<td>1,910</td>
<td>627</td>
<td>225</td>
</tr>
</tbody>
</table>

Table 4. Execution times (s) of synchronous, asynchronous and block advection for 8M particles over datasets.

We can also compare to the performance of Bußler et al. [4], which traced particles interactively through time-varying flows (with time-varying meshes). They traced particles interactively and their Fig. 3 reported a runtime v. integration step, so their runtimes measure the time to move the entire particle population one integration timestep ($\Delta t$). Their “Gyro” dataset (unavailable to us) consists of 1.1M cells and so is comparable to the size of our aorta dataset. They report (virtually in Fig. 5) a running time of about 23 ms for 1M particles for one integration step. Block advection traces 8M particles over 10K integration time steps in 233 s which averages to just less than 4 ms per 1M particles per timestep. Their GPU (NVidia GTX 480) with 480 total cores running at 1.5GHz whereas the K20’s 2,496 cores run at 706 MHz, suggesting that the K20’s fivefold processors running at half speed should represent at $\frac{1}{2}$ $\times$ performance increase. We conclude that block advection represents over a $\times$2 performance improvement over Bußler et al. [4].

### 6.3 Timestep Robustness

Table 5 measures the performance of block tracing verses ordinary asynchronous over changes in the integration timestep $\Delta t$. While the running times of both BA and Asynchronous appear to scale linearly with the number of steps ($= \frac{1}{\Delta t}$), the speedup of BA over Asynchronous grows from just under 4x to over 5x as the timesteps decrease. The number of BA executions grows slightly (10% as the timesteps decrease to a tenth), the number of integration steps per BA execution grows linearly (at a 90% rate) with the number of integration steps. Hence the added costs of the blocked advection approach, including the shared memory loading of cells and the transfer of existing particles, do not increase drastically as the integration timestep decreases, suggesting that block advection can transfer GPU scalability directly to improved simulation precision.

<table>
<thead>
<tr>
<th>$\Delta t$ (ms)</th>
<th>Steps</th>
<th>BA</th>
<th>Asynch.</th>
<th>Speedup</th>
<th>BA Ex.</th>
<th>Steps/BA Ex.</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0</td>
<td>1,000</td>
<td>36.7</td>
<td>142</td>
<td>3.86</td>
<td>319</td>
<td>3.13</td>
</tr>
<tr>
<td>0.9</td>
<td>1,112</td>
<td>39.6</td>
<td>156</td>
<td>3.95</td>
<td>320</td>
<td>3.47</td>
</tr>
<tr>
<td>0.8</td>
<td>1,250</td>
<td>42.9</td>
<td>174</td>
<td>4.06</td>
<td>323</td>
<td>3.87</td>
</tr>
<tr>
<td>0.7</td>
<td>1,429</td>
<td>47.4</td>
<td>197</td>
<td>4.16</td>
<td>328</td>
<td>4.36</td>
</tr>
<tr>
<td>0.6</td>
<td>1,667</td>
<td>53.2</td>
<td>228</td>
<td>4.28</td>
<td>331</td>
<td>5.04</td>
</tr>
<tr>
<td>0.5</td>
<td>2,000</td>
<td>61.3</td>
<td>271</td>
<td>4.42</td>
<td>331</td>
<td>6.04</td>
</tr>
<tr>
<td>0.4</td>
<td>2,500</td>
<td>73.4</td>
<td>335</td>
<td>4.57</td>
<td>335</td>
<td>7.46</td>
</tr>
<tr>
<td>0.3</td>
<td>3,334</td>
<td>93.2</td>
<td>442</td>
<td>4.74</td>
<td>338</td>
<td>9.86</td>
</tr>
<tr>
<td>0.2</td>
<td>5,000</td>
<td>133</td>
<td>655</td>
<td>4.94</td>
<td>344</td>
<td>14.5</td>
</tr>
<tr>
<td>0.1</td>
<td>10,000</td>
<td>248</td>
<td>1,289</td>
<td>5.19</td>
<td>349</td>
<td>28.7</td>
</tr>
</tbody>
</table>

Table 5. Block advection (BA) v. Asynchronous execution times (s) for different integration time steps for one second of simulation, including number of block advection calls and the average number of integration steps per BA call.

## 7 Conclusion

We have established a new performance mark for tracing particles through unsteady unstructured flow datasets. We have shown that the performance of GPU particle tracing algorithms suffer as particles disperse across the flow domain, but these effects can be countered by block advection. We have also analyzed block advection and provide advice on its tuning, such as setting a particles per block threshold at 256, and setting the outer block width to about five cell widths.

Our algorithms and analysis do have some limitations worth discussing. We rely entirely on cell walking for particle location, which is not feasible for large velocities or time steps. Our multi-stage low register integration code does not support the adaptive methods commonly used for particle tracing. All of our algorithms require the entire dataset to be available in the GPU global memory. Our datasets are all vascular in nature and should be expanded to include unsteady unstructured flow datasets in other disciplines.

The initial success of the blocked advection approach considered with these limitations illuminate several opportunities for further research. The expectation that time-varying flow datasets fit in GPU
memory limits the scope of block advection. One could extend the block idea to include larger GPU blocks paged in from the CPU, using a similar inner and outer boundary idea to keep particles from switching GPU blocks too frequently.

ACKNOWLEDGMENTS

This work was supported in part by NSF Grant OCI-1047764 and NVIDIA. Thanks to Shawn Shadden, Siavash Ameli, Yogin Desai and Anil Hirani for their help with this project.

REFERENCES


