https://doi.org/10.2352/EI.2025.37.12.HPCI-179

Steady-State Particle Advection Speed-Ups from GPU and CPU Parallelism

Abhishek Yenpure; Kitware Inc., USA
David Pugmire; Oak Ridge National Laboratory, USA
Hank Childs; Hank Childs, University of Oregon, USA

Abstract

This study evaluates the benefit of using parallelism from GPUs or multi-core CPUs for particle advection workloads. We perform 1000+ experiments, involving four generations of Nvidia GPUs, four CPUs with varying numbers of cores, two particle advection algorithms, many different workloads (i.e., number of particles and number of steps), and, for GPU tests, performance with and without data transfer. The results inform whether or not a visualization developer should incorporate parallelism in their code, what type (CPU or GPU), and the key factors influencing performance. Finally, we find that CPU parallelism is the better choice for most common workloads, even when ignoring costs for data transfer.

Introduction

Particle advection, i.e., displacing a massless particle according to a vector field, is a foundational operation for flow visualization. This operation is carried out by calculating particle trajectories by a series of "advection steps." Each advection step involves evaluating a vector field at one or more locations and then solving an ordinary differential equation. Flow visualization techniques may require many particles, many advection steps per particle, or even both. As a result, particle advection-based flow visualization can be very computationally expensive, hindering interactivity.

Parallel processing is a key approach for reducing long execution times. That said, this very simple question — "how much speed-up should I expect to get if I enhance my visualization software to use parallelism?" — has a surprisingly complex answer. The followup question "which type of parallelism should I use?: CPU or GPU" also is non-obvious. On the one hand, GPUs provide significant computational resources, making them potentially very useful for particle advection problems. That said, it can be difficult to achieve peak performance on a GPU, especially for data-intensive operations. Further, particle advection-based flow visualization includes diverse use cases which can lead to varying performance characteristics and varying speed-ups across GPU architectures. On the other hand, while multi-core CPUs often do not have the raw FLOPS of a GPU, they can compare favorably to a GPU either because of faster individual cores or because of direct access to data (i.e., no data transfers).

In response, we consider four related research questions on particle advection performance:

- **RQ1**: How much speed-up will a GPU provide over a serial CPU implementation? How much does individual GPU architecture matter?
- **RQ2**: How much speed-up will a multi-core CPU provide over a serial CPU implementation? How much does CPU concurrency matter?
- **RQ3**: When given the opportunity to use either a GPU or a multi-core CPU, which should a visualization developer choose?
- **RQ4**: What are the trends in performance by using newer hardware?

The main outcome of this study is informing visualization developers whether parallelism will speed up their particle advection workloads, and, if so, to what extent. That said, the space of possible experiments is quite large and, to maintain an achievable scope, we introduce two boundaries. For the first boundary, our study is targeted at visualization software running on desktop machines, and does not consider supercomputers. Desktop machines are more widely accessible, and they increasingly have general-purpose computing environments on their GPUs (CUDA, OpenCL, etc.) and also significant parallelism via CPU cores. Further, while we do not run distributed-memory experiments on supercomputers, our findings also have implications for this environment. For the second boundary, our study focuses exclusively on steady-state flow. Steady-state flow advection is a common use case, and particularly common for the workloads that consider many advection steps (i.e., the workloads that benefit most from parallel processing). That said, our findings again have implications beyond our scope, and we consider how our findings apply to the unsteady-state case in our conclusions.

Related Work

Shared-memory parallelism refers to using the parallelism over many cores that can all access the same main memory. Shared-memory parallelism is possible using both multi-core CPUs and GPUs. There has been a significant body of particle advection works that focuses on using shared-memory parallelism to improve the performance of related flow visualization algorithms.

Some past works have investigated the use of shared-memory parallelism for particle advection in the context of distributed-memory parallelism, i.e., MPI-hybrid parallelism. Camp et al. proposed two different algorithms for particle advection on large data that used multi-core CPUs for shared-memory parallelization [10]. The multiple cores of the CPU were used to perform particle advection, I/O operations, and communication between nodes. Camp et al. [11] later extended their work to use GPUs and compared the performance of GPUs to multi-core CPUs. Their findings reveal that the CPUs performed better for certain workloads where there are fewer particles or where the

duration of advection is long. However, the GPU was able to perform better for the other workloads. Childs et al. [13] compared various GPUs and CPUs to understand the relationship between the execution devices and the execution time. They made two key observations, 1) CPUs were better for medium to long duration of advection, and 2) for many cases with overall short execution times CPUs matched or outperformed the GPUs. Jiang et al. used the multiple threads of CPUs to perform I/O operation to hide the I/O latency and improve particle advection performance [20]. Hentschel et al. [19] studied the benefits of using SIMD extensions to achieve better performance for particle advection. They packed spatially close particles together to use SIMD extensions efficiently by improving the spatial locality of memory accesses. They reported a performance improvement of $5.6\times$ over a baseline implementation. While these works provide some insights into our study, their findings were rooted in distributed-memory parallelism, and so performance results were affected by load imbalances and other factors, and thus did not inform the benefit of using a CPU or a GPU on a desktop.

GPUs have gained a lot of popularity as general-purpose computing devices over the last decade because of specialized tools like Nvidia's CUDA library [27]. And as of lately, libraries like Kokkos [15], RAJA [1], and VTK-m [25] allow users to write C++ code that runs on GPUs without the user requiring much expertise, making it easy to write parallel applications. GPUs offer excellent parallelism, given that there is enough work that can be efficiently parallelized. Particle advection lends itself to parallelization easily as each particle can be advanced independently, making it embarrassingly parallel. That said, parallelism cannot be achieve over the advection steps for a particle, since each step depends on the result of the previous step.

Past studies have used GPUs for particle advection to perform interactive flow visualization. Krüger et al. investigated using GPUs to produce particle visualizations for a million particles at once [22]. Their strategy was to exploit the GPU's ability to perform particle advection and produce visualizations without having to move data between the GPU and the host. The authors demonstrated the efficacy of their system for unsteady-state particle visualizations and also for 3D steady-state visualizations like streamlines and stream ribbons. Bürger et al. extended the system of Krüger et al. to produce unsteady flow visualizations [8]. Their approach was to stream data to the GPU while the GPU was busy performing particle advection, such that the next data slice would be available when needed. Bürger et al. later demonstrated their system could efficiently recognize flow features using some measure of importance such as Finite Time Lyapunov Exponent (FTLE), helicity, vorticity, etc. [7]. Bürger et al. then demonstrated their system could render streak surfaces [6] by adaptively refining/coarsening the surface at interactive rates with the GPUs.

Pugmire et al. [32] implemented a platform portable particle advection solution using VTK-m [25]. They evaluated their implementation on multi-core CPUs and GPUs. They demonstrated their implementation can perform well against platform-optimized particle advection solutions. Their work has also been of crucial importance for many studies that investigate the efficiency of large scale MPI-hybrid parallel particle advection [4, 2, 5, 3]. Our work is different than that of Pugmire et al. as we focus on expected speed-ups when moving to parallelism, where their focus was on demonstrating portable performance.

There have been many studies about the performance of particle advection at large scale by optimizing certain aspects for distributed particle advection. Operating in a distrubuted setting often demands data to be decomposed into small blocks which are distributed among processes. Since particle advection is highly data dependant, domain decomposition is important to ensure load-balanced computation. Efficient domain decomposition can also help in avoiding unnecessary I/O and communication. To that end, many studies have proposed schemes for data decomposition for particle advection based algorithms [12, 30, 29, 34] or for efficient I/O performance [21]. Another important aspect of distributed particle advection is work distribution and scheduling that leads to efficient use of parallelization and underlying hardware. To that end, studies have proposed new parallelization strategies [31, 28, 9, 23, 26] and demonstrate ways to use the whole spectrum of execution devices available using co-processing [18].

Experimental Overview

This section describes the setup for our experiments, which varied over the following parameters:

- Advection workloads
  - Number of seeds: 5 options
  - Duration: 3 options
  - Seeding volume: 3 options
  - Algorithms: 2 options
- Hardware usage
  - GPU transfer modes: 2 options
  - Devices: 4 CPUs and 4 GPUs

These options create for a potential 1080 experiments total: 90 advection workloads $\times$ 12 hardware options (4 for the CPU and 8 for the GPU with the transfer modes). That said, for each research question, we considered only a subset of the experiments, tailored to answer the question. For example, **RQ1** considered 240 experiments while **RQ3** considered 180.

All experiments were run using particle advection modules implemented in VTK-m [25]. VTK-m takes a portably performant approach, i.e., a single code implementation can run efficiently in serial, in parallel on a CPU, or in parallel on a GPU. This approach has been demonstrated to produce code that runs as efficiently as CPU-specific code or GPU-specific code, with findings specifically considering particle advection [32] and also a meta-study considering nine different visualization algorithms [24].

The remainder of this section describes the options for our experiments in more depth.

Data Sets

To answer posed research questions we used the 'Noise' data set, a vector field on a 512^3 uniform grid. Noise is a reference data set provided with the VisIt project [14]. The data set begins as scattered data, consisting of one hundred points in a volume. VisIt then constructs a vector field on a rectilinear grid by smoothly interpolating between the scattered data values. Noise was chosen because it has the least variability in number of steps, i.e., particles hit zero-velocity spots or exit the volume less often, making for more consistent results.

We note that the data set used for particle advection based algorithms can significantly impact performance. The relation-

ship between the dataset and the performance of the algorithm is explored in the Appendix.

Workloads

For this study, a particle advection workload has four factors:

- **Number of seeds**: the number of particles placed in the volume. For this study, we considered five amounts: 100, 1000, 10,000, 100,000, and 1,000,000.
- **Duration**: the number of advection steps performed for each particle. For this study, we considered three amounts: 100, 1000, and 10000.
- **Seeding volume**: the size of the sub-volume where seeds are placed. For this study, we considered three seeding volumes: Small (i.e., all seeds are placed in a small region near each other), Medium, and Large (i.e., seeds are placed randomly throughout the entirety of the data set). This factor is considered since small seeding volumes can have better cache coherency, especially in combination with short durations.
- **Algorithm**: how particle trajectories will be used. For this study, we considered two algorithms: particle advection and streamlines. Particle advection refers to simply advecting the particles to find their final position, which is useful for Finite Time Lyaponov Exponents (FTLE) and some other advanced analyses. The streamline algorithm stores the resulting position of each advection step. These two algorithms were chosen because they demonstrate differences in the load they place on the memory system.

Hardware Usage

GPU Transfer Mode: This factor considers the difference in performance when data needs to be transferred to/from the GPU ("with transfer"), as opposed to when data is already in the GPU's memory ("without transfer"). We ran experiments of both types in our study, in the following way:

- **With Transfer:** time to transfer the vector field data to the GPU, the time to perform the algorithm (streamlines or particle advection), and the time to transfer the data back. Note that the amount of data transferred back is different based on algorithm: proportional to the number of seeds for particle advection and proportional to the number of steps (seeds times duration) for streamlines.
- **Without Transfer:** time to carry out the algorithm (streamlines or particle advection). In this scenario, the vector field data is already on the GPU, and the results are not transferred off the GPU.

Devices: Our experiments were run on four different machines which provided access to four different CPUs and four generations of Nvidia GPUs. Table 1 describes these configurations for these machines. Alaska, Voltar, and Saturn (CPUs and GPUs 1, 2, and 4 respectively) are hosted at the University of Oregon, and Summit (CPU and GPU 3) is hosted at the Oak Ridge National Laboratory.

Results

This section is organized around our four research questions.

Table 1: The list of CPUs an GPUs that were used for the experiments in the paper.

CPUs	GPUs
CPU1: 2 x Intel Xeon E5-1650 w/ 12 cores, 3.8 GHz, and 32 GB memory.	***GPU1:*** Nvidia Tesla K40C w/ 12 GB memory and double precision performance of 1.68 TFLOPS.
CPU2: 2 x Intel Xeon 6226R w/ 32 cores, 3.9 GHz, and 256 GB memory.	***GPU2:*** Nvidia Tesla P100 w/ 16 GB memory and a double precision performance of 4.7 TFLOPS.
CPU3: 2 x IBM Power9 w/ 32 cores, 3.8 GHz, and 512 GB memory.	***GPU3:*** Nvidia Tesla V100 w/ 16 GB memory and a double precision performance of 7 TFLOPS.
CPU4: 4 x Intel Xeon 8367HC w/ 104 cores, 4.2 GHz, and 376 GB memory.	***GPU4:*** Nvidia Tesla A100 w/ 80 GB memory and a double precision performance of 9.7 TFLOPS.

RQ1: How Much Speed-up Will a GPU Provide Over a Serial CPU Implementation?

Table 2: The 270 configurations used to explore **RQ1**. 240 of these configurations were on a GPU and 30 served as baseline experiments on a serial CPU. However, 24 GPU experiments were unable to finish as the required memory exceeded the device memory; these experiments all involved streamlines with many advection steps.

Parameter	Value	Total
Data Sets	Noise	1
Seed Volume	Large	1
Seeds	All	5
Duration	All	3
Algorithm	Particle Advection, Streamlines	2
Hardware	CPUs (Serial), GPUs (w/o Xfer, w/ Xfer)	9

Table 2 shows the parameters for this phase's experiments and Figure 1 shows the results for these experiments. This plot shows a wide range of outcomes — for some experiments a GPU can be over 100X faster than a serial CPU while other experiments show a serial CPU to be over 50X faster than a GPU. Across all experiments, however, the average GPU speed-up is 6.14X compared to a serial CPU, with the following breakdown into ranges:

GPU Speed-up	<1X	1-4X	4-16X	16-64X	>64X
% of tests	20%	18%	28%	20%	14%

The following subsections analyze these results with respect to number of steps, GPU architecture, algorithm, and memory transfer mode.

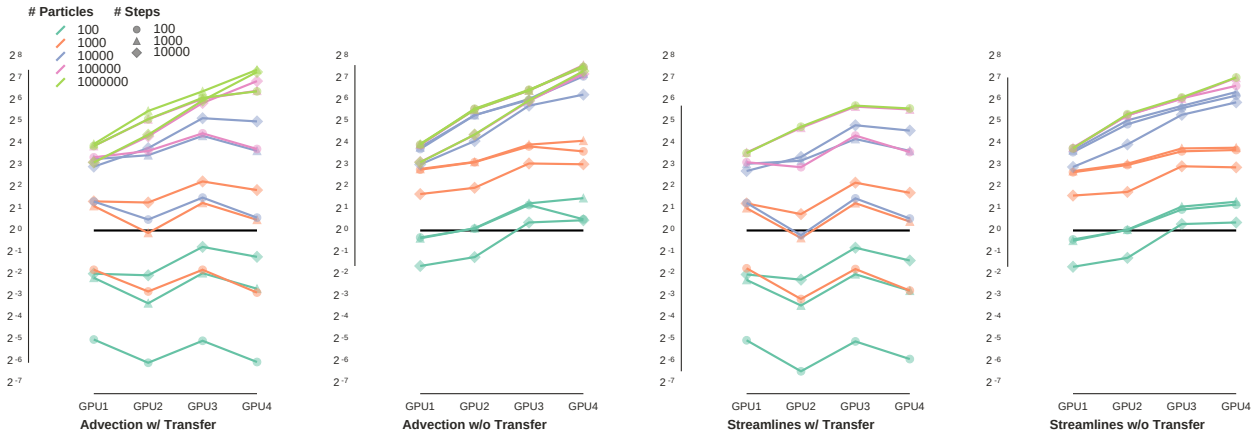

Figure 1: A chart showing GPU speed-up compared to a serial CPU. For each of the four figures, the X-axis represents the GPU generation, older to newer from left to right. The colors represent the number of particles in the workload and the glyphs represent the duration of the workload.

Effect from Number of Advection Steps

This section considers the effects from the number of advection steps, i.e., the combination of the number of particles and the duration of each.

Number of particles is a dominant factor in speed-up. This is an expected finding — parallelization occurs over particles, and having ten thousand particles or more allows all threads to be engaged. Focusing on the portion of Figure 1 devoted to advection without transfer, the speed-ups for ten thousand particles are nearly identical to those with one million particles. For the workloads with one million particles, the speed-ups are as low as 8X (on GPU1) and as high as 160X (on GPU4). Further, the workloads with ten thousand and one hundred thousand particles also show strong speed-ups. Looking at the other configurations in Figure 1, some ten thousand particle workloads are just as fast as million particle comparators. For others, the speed-up is less, but still significant. For example, for "streamlines without transfer" on GPU4, the speed-up with ten thousand particles is about 70X, while for one million particles it is over 120X. Across all configurations, the workloads with one hundred particles fare much worse, with speed-ups topping off at 2X, and many actually running slower on the GPUs. The workloads with one thousand particles perform better, with some seeing speed-ups of 8X, although some of these workloads are still slower on GPUs compared to a serial CPU.

Duration affects speed-up less. For the workloads with ten thousand particles or more, the expected speed-up does not change much as duration varies. For workloads with fewer particles, however, duration is a more significant factor. For example, for the "advection with transfer" case with one hundred particles, durations of 100 steps are 30X faster on a CPU while durations of 10000 steps are merely 4X faster on the CPU. In all, the effect of duration is only significant for workloads where GPUs provide little-to-no value.

Effect of GPU Architecture

Figure 1 shows the expected result that newer GPUs are able to offer better performance. For the workloads with the most advection work, each newer generation of GPU provided an improvement in performance, especially in cases where memory transfers were not considered. The newest GPU (GPU4) provided a maximum speed-up of 160X in case of particle advection and 130X in case of streamlines. The oldest GPU (GPU1) provided a maximum speed-up of 16X in all cases.

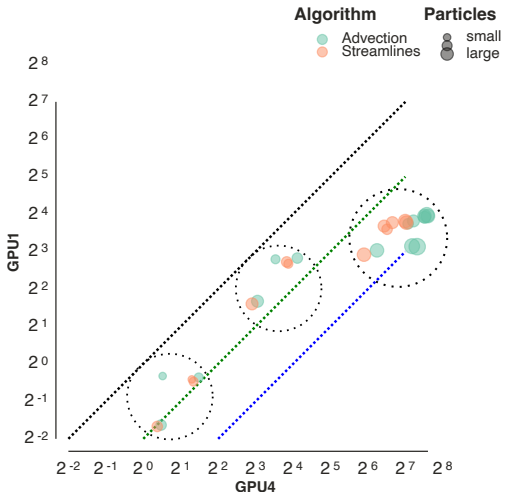

Figure 2: Scatter plot of speed-ups achieved for GPU4 (Ampere) versus GPU1 (Kepler) for workloads with no data transfer. If a given workload had a 30X speed-up on GPU4 and a 6X speed-up on GPU1, then that workload would be plotted at (30, 6) in this figure. The dotted lines show relationships between GPU1 and GPU4: black shows where performance between the two GPUs is equal, green shows where GPU4 is 4× faster than GPU1, and blue shows where GPU4 is 16× faster than GPU1. Finally, the three dotted circles indicate three clusters of similarly performing experiments.

Figure 2 plots the speed-ups for these extreme GPUs in our study: Ampere (GPU4) versus Kepler (GPU1). This plot shows three distinct clusters:

1. The first cluster contains workloads where neither GPU1 nor GPU4 offered any improvements over serial CPU. These

workloads generally have a small amount of work to do. That said, GPU4 still performs 4× better compared to GPU1 for these workloads.

2. The second cluster contains workloads where both GPU1 and GPU4 offer significant speed-ups over serial CPU. Once again, GPU4 is only $2-4\times$ faster than GPU1.

3. The third cluster contains workloads with a bigger disparity between GPU4 and GPU1 (much larger than 4X): ˜128X speed-ups for GPU4 versus only ˜8X for GPU1. This is where the majority of our workloads fall, and thus reflects the most common outcome within our corpus of tests. The best speed-ups were achieved by the workloads that only advected particles and did not generate streamlines, which leads into the next section on algorithm effects.

These clusters are revisited in the section discussing RQ4 which looks at hardware trends.

Effect of Algorithm

In the context of our performance study, there are two main effects due to algorithm. First, the streamline algorithm stores each particle position (12 bytes of position data for every advection step), which can stress the memory system. Second, the output of the streamline algorithm is much larger than particle advection, and so the configurations where data is transferred back to the CPU can potentially face bottlenecks. Figure 1 illustrates the impact of each effect. First, the second and fourth sub-figures of Figure 1 show the experiment results without transfer, i.e., they show the differences solely due to storing more particle positions. The average speed-up for streamlines (fourth sub-figure) is 10.45X, while the average speed-up for advection (second sub-figure) is 12.28X, i.e., streamlines' extra memory stressors cause a 17% slowdown. (Note that some streamline experiments could not complete due to exceeding memory, and the corresponding advection experiments were removed for this analysis.) Next, the first and third sub-figures of Figure 1 inform the effects of transfer. For the experiments involving transfer, the average speed-up for streamlines (third sub-figure) is 1.92X, while the average speed-up for advection (first sub-figure) is 2.23X. The gap between the two algorithms has narrowed from 17% to 16%, i.e., the fixed cost of transferring data set causes them both equal slowdown and the extra memory stressors for streamlines becomes slightly less pronounced. Finally, the averages presented in this analysis are geometric means, which help with interpreting behaviors that range between large speed-ups and slowdowns. Repeating the analysis with arithmetic means gives 13.25X, 37.78X, 9.88X and 27.32X for the four sub-figures. While these numbers are skewed higher by the experiments with many advection steps, the same trends hold: "without transfer" has a 38% slowdown for streamline memory stressors, while adding transfer times narrows the slowdown to 34%.

Finally, Figure 3 shows a scatter plot considering speed-ups for the two algorithms on GPUs compared to a serial CPU. This figure has several findings. First, streamlines consistently achieve speed-ups within a factor of two of advection. Second, the ratio between streamline speed-up and advection speed-up appears to get larger as the overall speed-up improves. In other words, in the cases where the speed-up is great (i.e., many advection steps), streamlines fall off the pace somewhat, due to stresses on the memory system. Third, the GPUs perform differently. In particular, GPU2 has worse streamline performance than the other hardware architectures.

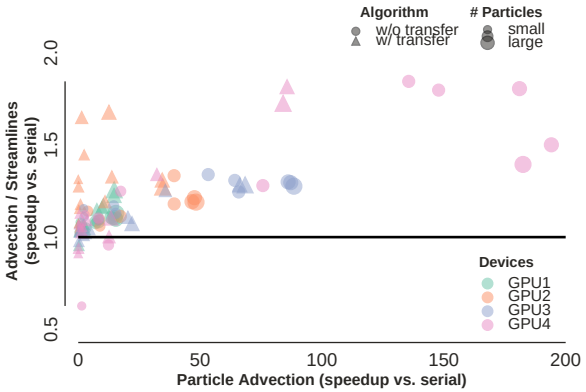

Figure 3: A scatter plot showing the differences between the particle advection and streamline algorithms. The X-axis represents the speed-up achieved by a certain workload for the particle advection workload, and the Y-axis represents how much faster the particle advection algorithm executed compared to the streamline algorithm. A glyph at (X, Y) indicates the particle advection algorithm running on a GPU achieved a speed-up of X times, and that its speed-up was Y times better than the streamline algorithm with the similar workload. The glyph color indicates the GPU device type, the glyph shape indicates whether or not data transfer was involved, and the glyph size indicates the number of particles (which informs how many GPU cores could be engaged).

Effect of GPU Transfer Mode

Figure 4 shows the effect of GPU transfer mode, i.e., if the data starts on the CPU, then how much effect is there to transfer the data to the GPU and back? In the "with transfer" experiments, the vector field was always transferred to the GPU, as were the starting seed positions. The data retrieved differed based on algorithm: either every position of every step (streamlines) or just final particle position (advection). This overhead was significant, as no experiment that involved transfers went faster than 0.18s. As a result, the overhead dominates the left portion of both figures: when the runtime "without transfer" is fast, then the slowdown is proportional to the data transfer time. For example, for streamlines on GPU4 with 10000 particles going 100 steps, the time without transfer is 0.007s and with transfer is 0.372s for a transfer slowdown of 50X. Both plots show an inflection point around execution times of 0.5s, when the data transfer overheads are more amortized. That said, relatively few streamline experiments are able to benefit from this amortization, since the streamline experiments that ran large numbers of advection steps to exceed 0.5s often ran out of memory. In other words, streamlines with memory transfer was almost always a poor idea in our set of experiments — with little work, the transfers dominated while with significant work, the experiment could not complete. Advection, on the other hand, showed significant benefit for the highest workloads. Summarizing, the main findings from this analysis are: (1) small workloads perform poorly due to data transfer overhead, (2) few workloads perform well with the streamline algorithm due to data

transfer overhead, and (3) large workloads can perform well with the advection algorithm.

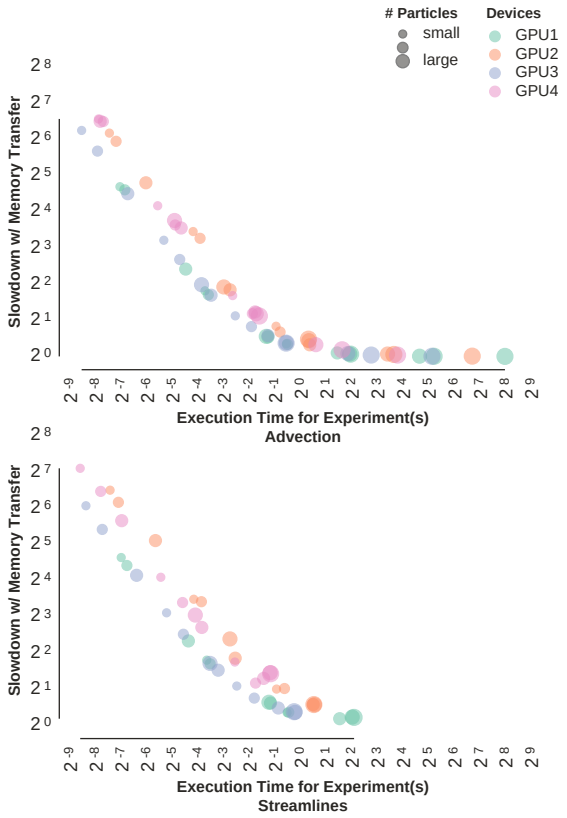

Figure 4: Plotting the slowdown for using memory transfer as a function of execution time. The figure is split into two, with advection experiments in the top figure and streamline experiments in the bottom figure. A glyph at (X, Y) means that a given workload took X seconds to execute without transfer and that workload was Y times slower when running with transfer. There are no glyphs for streamlines with execution times greater than four seconds, as those experiments exceeded GPU memory.

Takeaways for GPUs

Table 3 synthesizes the findings from this section. Each of the four factors (# of advection steps, architecture, transfer mode, algorithm) significantly affects performance:

- GPUs were not useful for many workloads with small numbers of steps, although the threshold for when they became useful varied based on GPU transfer mode.
- For workloads doing the particle advection algorithm and many advection steps, the GPU transfer mode becomes less relevant.
- The streamline algorithm is a little slower than the particle advection algorithm in all cases, but the magnitude of effect is not as big as the other factors. That said, the streamline algorithm is not viable for very large numbers of steps.
- The improvements in hardware architecture (GPU1 to GPU4) make an impact (4X-20X) in almost all cases where a GPU can outperform a serial CPU. The most notable cases where the change in hardware architecture does not make an impact is with the 10^7 workloads with data transfer. In these cases, the data transfer time is large enough that the increased computational power does not significantly affect speed-up. Larger workloads do benefit from the increased computational power from GPU4, while smaller workloads are affected by transfer times to the point that a serial CPU is preferable.

Table 3: The average speed-up, arranged by workload size, for GPU1 and GPU4 over serial execution for both algorithms and transfer modes. Each value represents the average of all workloads that used the specified number of steps. Note the 10^9 workload consists of 1M particles for 1K steps and 100K particles for 10K steps, while the 10^{10} workload consists of only 1M particles with 10K steps. The shorter duration (1K) workload ran faster, resulting some apparent slowdowns between the 10^9 and 10^{10} cases. In actuality, the 10K experiments did increase speed-up when going from 100K particles to 1M particles. Any time the CPU is faster is denoted < 1 and any time an experiment could not complete is denoted with an X.

# of Steps	Advection				Streamlines			
	w/ X		w/o X		w/ X		w/o X	
	G1	G4	G1	G4	G1	G4	G1	G4
$< 10^5$	< 1	< 1	< 1	< 1	< 1	< 1	< 1	< 1
10^5	< 1	< 1	2	5	< 1	< 1	2	5
10^6	< 1	< 1	3	14	< 1	< 1	2	10
10^7	6	8	8	60	5	7	8	39
10^8	11	61	12	138	9	38	11	100
10^9	11	139	11	166	X	X	X	X
10^{10}	8	155	8	160	X	X	X	X

RQ2: How Much Speed-up Will a Multi-core CPU Provide Over a Serial CPU Implementation?

Table 4: The 150 configurations used to explore **RQ2**. 120 of these configurations were on multi-core CPUs and 30 served as baseline experiments on a serial CPU. The largest serial CPU streamline experiment was unable to complete, so results from that experiment and its four multi-core counterparts are not presented in the analysis.

Parameter	Value	Total
Data Sets	Noise	1
Seed Volume	Large	1
Seeds	All	5
Duration	All	3
Algorithm	Particle Advection, Streamlines	2
Hardware	CPUs (Serial, Multi)	5

Table 4 shows the parameters for the experiments for this phase and Figure 5 plots the speed-up for multi-core CPUs using both the advection and streamlines algorithms. Figure 5 shows the expected result that newer CPUs with more cores perform better than older CPUs with fewer cores. Compared to GPUs experiments, this plot shows a relatively narrower range of outcomes — multi-core CPUs perform better than serial in all but two cases. These outcomes spanned a spectrum from 6X to 64X. Across all experiments, the average CPU speed-up is 13.91X compared to a serial CPU, with a breakdown into ranges as follows:

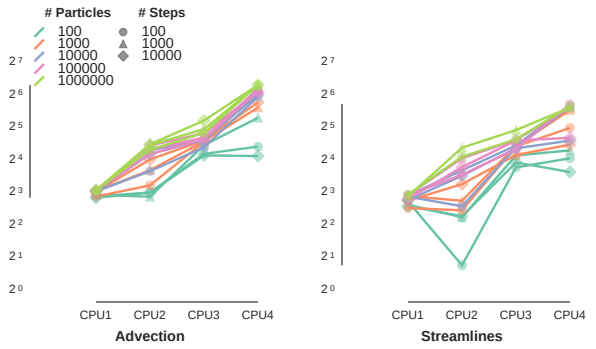

Figure 5: A chart showing CPU speed-up compared to a serial CPU. For both subfigures, the X-axis represents the CPU generation, older to newer from left to right. The colors represent the number of particles in the workload and the glyphs represent the duration of the workload.

CPU Speed-up	<1X	1-4X	4-16X	16-64X	>64X
% of tests	0%	1%	41%	50%	8%

In all cases, the multi-core CPUs fall short of their maximum possible speed-up, e.g., CPU4 (104-core Xeon) achieves ˜70X speed-ups instead of 104X speed-ups. That said, this level of speed-up is consistent with previous multi-core scaling studies.

Focusing on the results from the advection algorithm, all CPUs demonstrated the same behavior. CPU1 is able to consistently provide similar speed-up of ˜8X for all workloads, while the other CPUs demonstrate a spread in terms of their speed-ups based on the number of particles in the experiments. However, the spread between the best and worse speed-up using any CPU is smaller that the corresponding GPU experiments. The highest spread for CPUs is for CPU4 with a spread of ˜4.5X between the best and the worse experiment, while the spread for the corresponding GPU, GPU4, is ˜140X.

In terms of the parallel performance efficiencies, CPU2 and CPU3 ($> 75\%$) performed better than CPU4 and CPU1 (66%). For all CPU2 and CPU3 experiments, speed-up increased for both, i.e., speed-up increased when increasing the number of particles or increasing their duration. While CPU1 is able to offer its best performance even for the workloads of lower magnitude due to minimal overhead of using threads, the limited number of threads and slower memory prevents it from performing better for workloads with a greater magnitude. On the other hand, CPU4, which offers a lot of parallelism, suffers from an initialization cost for smaller workloads (thread launch) and poor memory accesses (across NUMA regions) for the larger ones.

The streamline results from Figure 5 shows the effects of memory accesses from storing each advection step. The 32-core CPU2 and CPU3 are both slowed by ˜2X. The 104-core CPU4 is affected more dramatically, as performance starts dropping as the number of advection steps increases, sometimes falling below the 32-core CPUs. In all, memory effects clearly denigrate the streamline algorithm's performance. Finally, Figure 6 shows a comparison between the 12-core CPU1 and the 104-core CPU4 which emphasizes the varying behavior across these architectures. It shows that the CPU1 has very consistent performance, while CPU4 spans the spectrum from doing ˜12X faster to performing

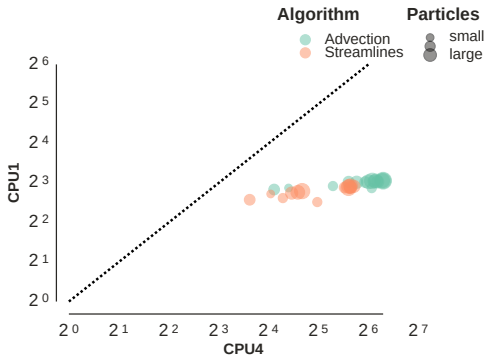

Figure 6: This scatter plot compares the performance of a 104-core Xeon with a 12-core Xeon. If a workload was 64X faster than a reference serial implementation on the 104-core Xeon and 6X faster on the 12-core Xeon, then a glyph would be placed at (64, 6).

˜70X faster. CPU4 is able to offer better scalability as the workload increases as each core gets substantial work relative to overhead of assigning work. Eventually, for streamlines, the performance is capped at at 40X, and for particle advection, the speed-ups can exceed 64X.

Takeaways for CPUs

Table 5: The average speed-up by CPU1 and CPU4 over serial execution for the two algorithms for different workload sizes. This table can be used to estimate the speed-up that can be expected for executing a certain workload. Any time an experiment could not complete is denoted with an X.

	Advection		Streamlines	
Workload	CPU1	CPU4	CPU1	CPU4
10^4	7	21	6	30
10^5	7	51	6	16
10^6	7	37	5	24
10^7	8	64	7	38
10^8	8	70	7	38
10^9	8	72	7	35
10^{10}	8	78	X	X

Table 5 shows the average speed-ups with respect to workload and architecture. For both algorithms, multi-core CPUs are able to achieve the best performance once the workload has a total number steps $\geq 10^7$. CPUs with a lower number of total cores (e.g., CPU1) are able to achieve their best performance even for workloads of lower magnitudes. CPUs with more cores (e.g., CPU4) show a steady increase in speed-ups with increases in the workloads.

RQ3: How Does GPU Performance Compare to Multi-Core CPU Performance?

Table 6 shows the parameters for the experiments for this phase and Figure 7 compares multi-core CPU performance with GPU performance for each of the four machines. Of note, the computational power of the CPU and GPU appears to be somewhat balanced across the machines, with the 12-core Xeon paired

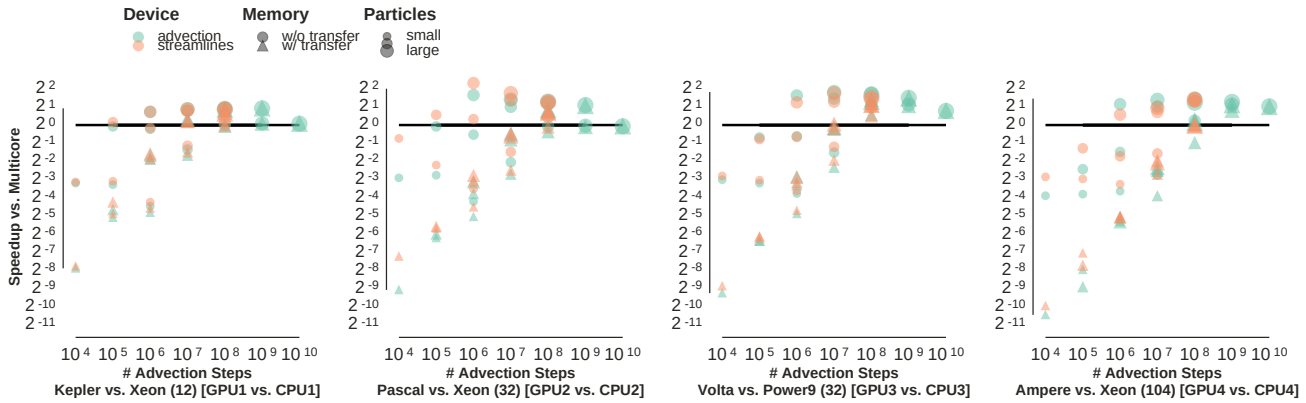

Figure 7: Comparison of GPU speed-ups for particle advection over their multi-core CPU counterparts with log scale in the Y-axis. This enables the identification of cases where the GPUs perform worse or better than a multi-core CPU. The horizontal line at $Y = 1$ is where the GPU performance equals that of a multi-core CPU. All data points below the line indicate slower GPU performance for an experiment.

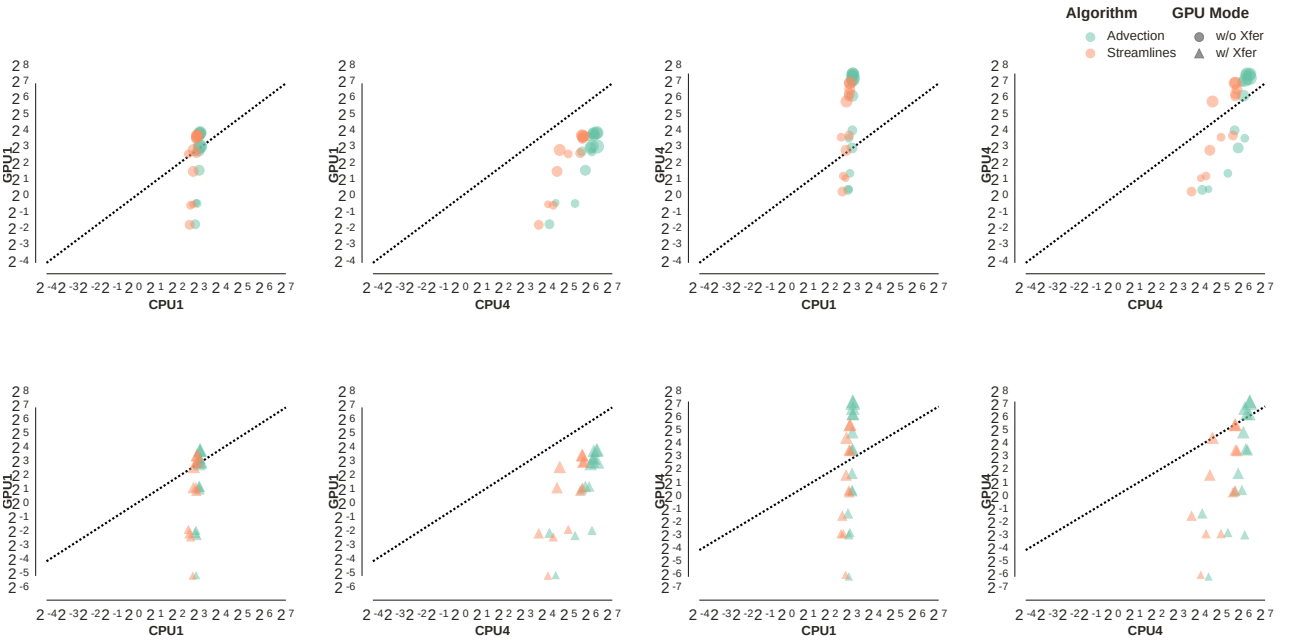

Figure 8: A small multiples chart comparing speed-ups for GPUs and multi-core CPUs compared to a serial CPU. Each figure contains a scatter plot with CPU results along the X-axis and GPU results along the Y-axis. Each point in the scatter plot corresponds to a workload when comparing to serial CPU times; if a workload was 30X faster on a multi-core CPU and 15X faster on a GPU, then a glyph would be placed at (30, 15). The overall layout of the small multiples chart is 4x2. The two rows correspond to data transfer, with top row plotting without data transfer and the bottom row plotting with data transfer. The four columns go through the combinations of best and worst CPU and GPU. The worst GPU in our study, GPU1 (Kepler), is in the left two columns, while the best GPU in our study, GPU4 (Ampere), is in the right two columns. The worst CPU in our study, CPU1 (Xeon 12 cores), is in the first and third columns, while the best CPU in our study, CPU4 (Xeon 104 cores), is in the second and fourth columns.

Table 6: The 360 configurations used to explore **RQ3**. 120 experiments were run on multi-core CPUs, 120 experiments were run on GPUs and did not involve data transfers, and the final 120 experiments were run on GPUs and involved data transfers.

Parameter	Value	Total
Data Sets	Noise	1
Seed Volume	Large	1
Seeds	All	5
Duration	All	3
Algorithm	Particle Advection, Streamlines	2
Hardware	CPUs (Multi), GPUs (w/o Xfer, w/ Xfer)	12

with the Kepler GPU (CPU1 and GPU1), the 32-core Xeon paired with the Pascal GPU (CPU2 and GPU2), the 32-core Power9 paired with the Volta GPU (CPU3 and GPU3), and the 104-core Xeon paired with the Ampere GPU (CPU4 and GPU4). In terms of results, an overwhelming trend is the imbalance in the range of outcomes — some multi-core CPU experiments are as much as 1000X faster than their GPU counterparts, although GPU experiments are never more than 4X faster than their multi-core CPU counterparts. For experiments without data transfer, the key factor in whether the GPU or CPU will be faster is the number of particles advected. For the most part, if 10,000 or more particles are advected, then the GPU is faster (since all GPU cores can be engaged), and if 1,000 or fewer particles are advected, then the CPU is faster (since the GPU cores cannot all be engaged).

Experiments involving data transfer and streamlines are almost always faster on the CPU. The only exceptions involve cases with 100M advection steps, and even then speed-ups were only modest ($< 2\times$). One reason is that GPUs can only outperform multi-core CPUs when there is significant work, but, for the streamline algorithm, the amount of work needed to offset transfer costs is so great that it exceeds GPU memory.

Finally, Figure 8 shows results when comparing the extreme of each architecture: best CPU (CPU4) vs worst GPU (GPU1), best CPU (CPU4) vs best GPU (GPU4), worst CPU (CPU1) vs worst GPU (GPU1), and worst CPU (CPU1) vs best GPU (GPU4). CPU4 beat the worst GPU1, and often by significant amounts, in all but a few configurations. CPU4 was also able to beat the GPU4 in most configurations. When considering experiments with data transfer, CPU4 is able to either beat or keep up with GPU4 in almost all cases, with the only exception being those with large workloads. CPU1 beat GPU1 only for smaller workloads. When comparing the CPU1 and the GPU4, GPU4 was substantially faster for almost all of the cases, losing only in the cases that involve small workloads.

Takeaways for GPUs and CPUs

Table 7 presents the average speed-up by using GPUs over multi-core CPUs. A value greater than 1 means that the GPU performed as many times better than their comparator CPU. The table informs when GPUs will be better than CPUs for a particular workload. In general, for workloads where the total number of steps is less than 10^8, GPUs should not be used as they perform worse than the comparator CPUs. The range where GPUs are helpful for generating streamlines is very narrow, as they need

Table 7: A table informing whether to use a GPU or a multi-core CPU for a certain workload. Each table entry corresponds to the average speed-up with running on a GPU as opposed to a multi-core CPU. This table displays two machines: GPU1 compared to CPU1 (denoted M1, for "machine 1") and GPU4 compared to CPU4 (M4). Any time the CPU is faster is denoted < 1 and any time an experiment could not complete is denoted with an X.

Workload	Advection—				Streamlines			
	w/ X		w/o X		w/ X		w/o X	
	M1	M4	M1	M4	M1	M4	M1	M4
$< 10^7$	< 1	< 1	< 1	< 1	< 1	< 1	< 1	< 1
10^7	< 1	1.0	< 1	< 1	< 1	1.1	< 1	1.02
10^8	1.4	1.5	< 1	1.9	1.3	1.5	< 1	2.58
10^9	1.4	1.4	1.9	2.2	X	X	X	X
10^{10}	1.0	1.0	1.9	2.0	X	X	X	X

significant amount of work to offset the costs of memory operations but are not able to support workloads which have memory requirements higher than the available GPU memory.

RQ4: What Are the Trends in Performance by Using Newer Hardware?

The three clusters identified in Figure 2 of the GPU analysis for RQ1 highlight the hardware trends over the past decade and offer insights for the future.

Cluster 1 includes workloads with minimal computational demands, where GPUs offer limited benefit. Although performance improvements between GPU1 and GPU4 cause these workloads to cross the threshold of outperforming a serial CPU when data transfer is not involved, it is unlikely that future GPUs will become significantly more effective for such small-scale tasks.

Cluster 2 includes workloads with modest speed-ups compared to serial CPUs. GPU1 offered 3X to 8X speed-ups for these workloads and GPU4 improved these to 8X to 20X However, given the advancements in multi-core CPUs, GPUs may not be the optimal choice for these workloads. Visualization programmers deciding between multi-core CPU and GPU implementations for these tasks would likely be better served by a multi-core CPU approach.

Cluster 3 includes workloads where recent GPU improvements have had the greatest impact. While GPU1 offered speed-ups of 8X to 16X, GPU4 pushed these gains to 64X to 128X. These workloads generally outperform their multi-core CPU counterparts, and with continued advancements in GPU technology (more cores, better memory infrastructure), further improvements are anticipated.

Finally, Figure 9 presents the scaling behaviors of all the GPUs and CPUs considered in our study. It combines the clusters identified in Figure 2 and uses it to enumerate the experiments in Figures 1 and 5. Additionally, Figure 9 introduces a fourth cluster (Cluster 4), representing streamline experiments that could not be completed on the GPUs due to limited memory capacity. From a workload perspective, each new generation of GPUs and CPUs showed significant speed-up improvements for the largest workloads (Cluster 3). This is primarily because newer CPUs and GPUs feature more cores, and each core performs substantially better than its predecessors. Workloads that fully utilize the cores

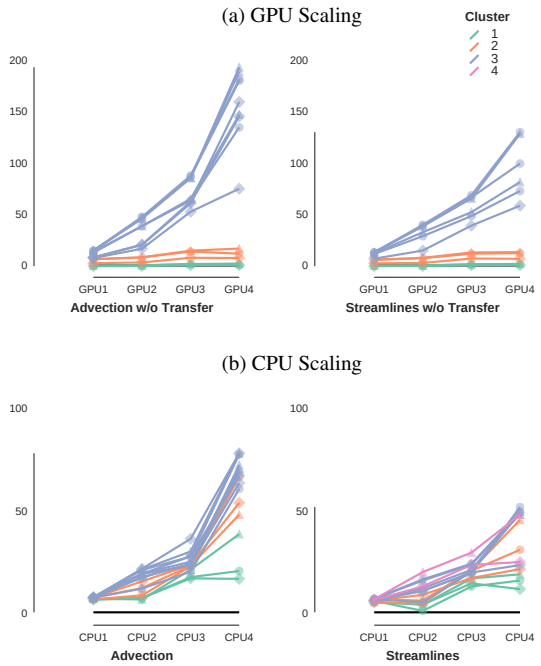

Figure 9: Figure plotting the speed-ups achieved for a workload by different generation of GPUs and CPUs for the two algorithms. (a) shows GPU scaling similar to Figure 1, and (b) shows CPU scaling data similar to Figure 5. Both of these figures use a linear Y axis unlike the reference figures which use a log-scaled Y axis. The lines are colored by the cluster they are categorized in Figure 2. Cluster 1 represents the poorest performing cluster and Cluster 3 represents the best performing cluster. Note that the additional Cluster 4 represents experiments that successfully completed on CPUs but failed on GPUs due to limited memory. This cluster is not depicted in Figure 2.

of the latest hardware benefit the most from these advancements. In contrast, for smaller workloads (Clusters 1 and 2), newer GPUs did not provide significant performance gains over previous generations. However, newer CPUs managed to offer modest speed-ups for these smaller workloads.

From an algorithmic perspective, newer GPUs exhibit consistent scaling behavior for both particle advection and streamline workloads. Specifically, workloads in Cluster 3 scale very well, while those in Clusters 1 and 2 do not. This trend does not hold for CPUs. While newer CPUs scale similarly for particle advection, they show much smaller gains for streamlines, sometimes performing worse than previous generations.

Nevertheless, there remains value in using CPUs for very large streamline workloads (total steps $\geq 10^8$). Due to high memory requirements, GPUs are unsuitable for these tasks, while CPUs can still offer decent speed-ups. These large workload cases are captured in Cluster 4 of Figure 9.

Hence, the findings can be summarized as:

- Newer generations of GPUs offer better scaling for both particle advection and streamlines, but only for workloads that can fully utilize all GPU cores.

- Newer generations of CPUs demonstrate superior scaling for particle advection but not for streamlines. However, they are still capable of handling very large workloads effectively.

Conclusion

The four research goals of this paper were to understand the benefits of using the parallelism of GPUs and multi-core CPUs for particle advection. In RQ1, our goals were to determine how much speed-up a GPU provided over using a serial CPU, and the differences between different GPU architectures. Our study shows that a GPU can provide speed-ups, but only for certain types of workloads. Over all our tests, GPU speed-ups were rather modest (2X-5X, depending on architecture). The maximal speed-ups (6X-25X, depending on architecture) were achieved for the larger workflows consisting of more than 1M steps. Because of memory costs on the GPU, the type of algorithm used has a large impact. The expected speed-ups across different architectures for the streamline algorithm ranges from 3-9X, while the particle advection algorithm ranges from 4-27X. The takeaway message is that a GPU implementation only makes sense when used on large enough workloads to overcome the costs associated with memory usage and data transfers. Finally, RQ1 also explored the difference between GPU architectures, and, on the whole, the best GPU in our study was ˜4X better than the worst. RQ2 investigated how much speed-up could be achieved using multi-core CPUs over using a serial CPU. We found that multi-core parallelism was always useful, and significantly useful if the number of particles was 1000 or more. We also found that the speed-up can vary based on the algorithm (streamlines strain memory more) and CPU architecture. As far as comparing CPU architectures, the 104-core Xeon ran fastest for all experiments with more than 100 particles, but the amount of improvement varied. RQ3 investigated if there were clear choices to be made for the community in deciding what type of parallelism to deploy for particle advection tools. For this question, the overall takeaway is that multi-core CPUs tend to be more efficient than GPUs. The costs for data transfer and memory usage are so high that it takes significant amounts of work to overcome. Further, because of the way the algorithms parallelize the work, only large number of particles can engage all of the cores on the GPU. On the other hand, CPUs with a large number of cores are very efficient at advecting particles. Finally, RQ4 presented the trends of performance improvement afforded by the advancements in both, GPUs and multi-core CPUs. The findings suggest that the both the architectures are able to benefit form the increasing concurrency of the execution hardware. The difference between the performance of the worse and the best GPUs and CPUs for the largest workloads was ˜10X.

While these experiments were run on a specific set of CPUs and GPUs, creating a potential concern about applicability as CPU and GPU technology evolve, we think these particular choices are helpful in informing our research questions. Most of the GPUs from this study are similarly-powered to desktop "gaming" GPUs that we feel will resonate with much of the audience for this paper deciding whether to migrate their advection code to the GPU. GPU4 is different, as it is similar to the "beefy" GPUs being used on modern supercomputers. Further, the nature of the findings serve to "future-proof" our study. Small workloads, which are more common in the visualization space, do not pro-

vide significant benefits on either modest GPUs or beefy GPUs. For large workloads, GPUs often win, and those winnings appear that they will grow bigger and bigger as GPUs evolve (RQ4).

Finally, our findings are somewhat different than the those by Pugmire et al. [32] who did a smaller study in 2018 comparing GPU and multi-core CPU performance again using VTK-m. We are using different architectures so direct comparison is difficult. That said, their comparisons on "Summit Dev" between Pascal and 20 cores of IBM Power-8 showed more significant speed-ups on the GPU than our comparisons on Summit between a Volta and 32 cores of IBM Power-9 performance. While none of our experiments match exactly, the most comparable involve our runs of one million particles with one thousand steps versus their runs of ten million particles with 100 steps. They achieved 1.4s with a P100, while we achieved 2.39s. On the CPU side, our run was 4.8s, while their run was 19.9s. The GPU slowndown may be attributable to many factors, but we point to changes in the last four years of the VTK-m source code. During that time, the code has been extended from "lean and mean" to be functional for a variety of use cases (FTLE, different grid types, electomagnetic fields, etc.) and this has led to more branching, etc., that can slow down GPUs. As a result, our findings should be interpreted as expected performance for a practical, richly-featured implementation.

In summary, we feel this paper provides information for the visualization community to guide decisions for tool development and deployment for particle advection. It will also help set expectations for algorithm performance on different hardware. This will help focus the efforts of developers to provide efficient solutions for the types of problems they plan to support and hardware that is available. While programming models for GPUs are improving, they are still challenging devices for development and debugging. Realistic expections for the performance on GPUs can be balanced against the development and maintenance cost for the anticipated uses cases of the software. For visualizations tools with large user bases, and use cases that are varied, this work provides practical information for the development of heuristics that can be used at run-time to make decisions on which hardware to target. Finally, we feel this work has significant implications for distributed-memory parallelism and in situ use cases. A distributed-memory implementation might need to support very large workloads, but this workload might be spread across a number of nodes. In such a case, there is globally a lot of work to do, but locally only modest amounts of work to do. For in situ processing, there are different options for how algorithms can be run. Is the simulation data already on the GPU? Are there CPU cores idle while the simulation runs? Is the GPU idle while simulation does communication, or switches to using the CPU cores? Is there enough work to warrant a transfer from the CPU to GPU, or vice versa? The best choice will vary based on the answers to these questions, and the findings in this paper can help inform these decisions.

Acknowledgments

This research was supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration.

References

[1] D. A. Beckingsale, J. Burmark, R. Hornung, H. Jones, W. Killian, A. J. Kunen, O. Pearce, P. Robinson, B. S. Ryujin, and T. R. Scogland. Raja: Portable performance for large-scale scientific applications. In *2019 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*, pp. 71–81. IEEE, 2019.

[2] R. Binyahib, D. Pugmire, and H. Childs. In Situ Particle Advection Via Parallelizing Over Particles. In *Proceedings of the Workshop on In Situ Infrastructures for Enabling Extreme-Scale Analysis and Visualization (ISAV)*, pp. 29–33. Denver, CO, Nov. 2019.

[3] R. Binyahib, D. Pugmire, and H. Childs. HyLiPoD: Parallel Particle Advection Via a Hybrid of Lifeline Scheduling and Parallelization-Over-Data. In *Eurographics Symposium on Parallel Graphics and Visualization (EGPGV)*, pp. 1–5. Zurich, Switzerland, June 2021.

[4] R. Binyahib, D. Pugmire, B. Norris, and H. Childs. A Lifeline-Based Approach for Work Requesting and Parallel Particle Advection. In *IEEE Symposium on Large Data Analysis and Visualization (LDAV)*. Vancouver, Canada, Oct. 2019.

[5] R. Binyahib, D. Pugmire, A. Yenpure, and H. Childs. Parallel Particle Advection Bake-Off for Scientific Visualization Workloads. In *IEEE International Conference on Cluster Computing (CLUSTER)*, pp. 381–391. Kobe, Japan, Sept. 2020.

[6] K. Bürger, F. Ferstl, H. Theisel, and R. Westermann. Interactive Streak Surface Visualization on the GPU. *IEEE Transactions on Visualization and Computer Graphics*, (6):1259–1266, 2009.

[7] K. Bürger, P. Kondratieva, J. Kruger, and R. Westermann. Importance-Driven Particle Techniques for Flow Visualization. In *2008 IEEE Pacific Visualization Symposium*, pp. 71–78. IEEE, 2008.

[8] K. Bürger, J. Schneider, P. Kondratieva, J. H. Krüger, and R. Westermann. Interactive Visual Exploration of Unsteady 3D Flows. In *EuroVis*, pp. 251–258, 2007.

[9] D. Camp, H. Childs, C. Garth, D. Pugmire, and K. I. Joy. Parallel stream surface computation for large data sets. In *Ieee symposium on large data analysis and visualization (ldav)*, pp. 39–47. IEEE, 2012.

[10] D. Camp, C. Garth, H. Childs, D. Pugmire, and K. Joy. Streamline Integration Using MPI-Hybrid Parallelism on a Large Multicore Architecture. *IEEE Transactions on Visualization and Computer Graphics*, 17(11):1702–1713, 2010.

[11] D. Camp, H. Krishnan, D. Pugmire, C. Garth, I. Johnson, E. W. Bethel, K. I. Joy, and H. Childs. GPU Acceleration of Particle Advection Workloads in a Parallel, Distributed Memory Setting. In F. Marton and K. Moreland, eds., *Eurographics Symposium on Parallel Graphics and Visualization*. The Eurographics Association, 2013.

[12] L. Chen and I. Fujishiro. Optimizing parallel performance of streamline visualization for large distributed flow datasets. In *2008 IEEE Pacific Visualization Symposium*, pp. 87–94. IEEE, 2008.

[13] H. Childs, S. Biersdorff, D. Poliakoff, D. Camp, and A. D. Malony. Particle Advection Performance Over Varied Architectures and Workloads. In *2014 21st International Con-*

ference on High Performance Computing (HiPC), pp. 1–10. IEEE, 2014.

[14] H. Childs, E. Brugger, B. Whitlock, J. Meredith, S. Ahern, D. Pugmire, K. Biagas, M. Miller, C. Harrison, G. H. Weber, et al. VisIt: An End-User Tool for Visualizing and Analyzing Very Large Data. 2012.

[15] H. C. Edwards and C. R. Trott. Kokkos: Enabling Performance Portability Across Manycore Architectures. In *2013 Extreme Scaling Workshop (xsw 2013)*, pp. 18–24. IEEE, 2013.

[16] E. Endeve, C. Y. Cardall, R. D. Budiardja, and A. Mezzacappa. Generation of Magnetic Fields by the Stationary Accretion Shock Instability. *The Astrophysical Journal*, 713(2):1219–1243, Apr 2010.

[17] P. Fischer, J. Lottes, D. Pointer, and A. Siegel. Petascale Algorithms for Reactor Hydrodynamics. In *Journal of Physics: Conference Series*, vol. 125, p. 012076. IOP Publishing, 2008.

[18] H. Guo, W. He, S. Seo, H.-W. Shen, E. M. Constantinescu, C. Liu, and T. Peterka. Extreme-scale stochastic particle tracing for uncertain unsteady flow visualization and analysis. *IEEE transactions on visualization and computer graphics*, 25(9):2710–2724, 2018.

[19] B. Hentschel, J. H. Göbbert, M. Klemm, P. Springer, A. Schnorr, and T. W. Kuhlen. Packet-Oriented Streamline Tracing on Modern SIMD Architectures. In C. Dachsbacher and P. Navrátil, eds., *Eurographics Symposium on Parallel Graphics and Visualization*. The Eurographics Association, 2015.

[20] M. Jiang, B. V. Essen, C. Harrison, and M. B. Gokhale. Multi-threaded streamline tracing for data-intensive architectures. In *4th IEEE Symposium on Large Data Analysis and Visualization, LDAV 2014, Paris, France, November 9-10, 2014*, pp. 11–18. IEEE Computer Society, 2014.

[21] W. Kendall, J. Huang, T. Peterka, R. Latham, and R. Ross. Toward a general i/o layer for parallel-visualization applications. *IEEE Computer Graphics and Applications*, 31(6):6–10, 2011.

[22] J. Krüger, P. Kipfer, P. Konclratieva, and R. Westermann. A Particle System for Interactive Visualization of 3D Flows. *IEEE Transactions on Visualization and Computer Graphics*, 11(6):744–756, 2005.

[23] K. Lu, H.-W. Shen, and T. Peterka. Scalable computation of stream surfaces on large scale vector fields. In *SC'14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1008–1019. IEEE, 2014.

[24] K. Moreland, R. Maynard, D. Pugmire, A. Yenpure, A. Vacanti, M. Larsen, and H. Childs. Minimizing Development Costs for Efficient Many-Core Visualization Using MCD^3. *Parallel Computing*, 108:102834, Dec. 2021.

[25] K. Moreland, C. Sewell, W. Usher, L.-t. Lo, J. Meredith, D. Pugmire, J. Kress, H. Schroots, K.-L. Ma, H. Childs, et al. VTK-m: Accelerating the Visualization Toolkit for Massively Threaded Architectures. *IEEE Computer Graphics and Applications*, 36(3):48–58, 2016.

[26] C. Müller, D. Camp, B. Hentschel, and C. Garth. Distributed parallel particle advection using work requesting. In *2013 IEEE Symposium on Large-Scale Data Analysis and Visualization (LDAV)*, pp. 1–6. IEEE, 2013.

[27] J. Nickolls. GPU Parallel Computing Architecture and CUDA Programming Model. In *2007 IEEE Hot Chips 19 Symposium (HCS)*, pp. 1–12. IEEE, 2007.

[28] B. Nouanesengsy, T.-Y. Lee, K. Lu, H.-W. Shen, and T. Peterka. Parallel particle advection and ftle computation for time-varying flow fields. In *SC'12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pp. 1–11. IEEE, 2012.

[29] B. Nouanesengsy, T.-Y. Lee, and H.-W. Shen. Load-balanced parallel streamline generation on large scale vector fields. *IEEE Transactions on Visualization and Computer Graphics*, 17(12):1785–1794, 2011.

[30] T. Peterka, R. Ross, B. Nouanesengsy, T.-Y. Lee, H.-W. Shen, W. Kendall, and J. Huang. A study of parallel particle tracing for steady-state and time-varying flow fields. In *2011 IEEE International Parallel & Distributed Processing Symposium*, pp. 580–591. IEEE, 2011.

[31] D. Pugmire, H. Childs, C. Garth, S. Ahern, and G. H. Weber. Scalable computation of streamlines on very large datasets. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, pp. 1–12, 2009.

[32] D. Pugmire, A. Yenpure, M. Kim, J. Kress, R. Maynard, H. Childs, and B. Hentschel. Performance-Portable Particle Advection with VTK-m. In *Proceedings of the Symposium on Parallel Graphics and Visualization*, EGPGV '18, pp. 45–55. Eurographics Association, Goslar, DEU, 2018.

[33] C. Sovinec, A. Glasser, T. Gianakon, D. Barnes, R. Nebel, S. Kruger, D. Schnack, S. Plimpton, A. Tarditi, M. Chu, et al. Nonlinear Magnetohydrodynamics Simulation Using High-Order Finite Elements. *Journal of Computational Physics*, 195(1):355–386, 2004.

[34] J. Zhang, H. Guo, F. Hong, X. Yuan, and T. Peterka. Dynamic load balancing based on constrained kd tree decomposition for parallel particle tracing. *IEEE transactions on visualization and computer graphics*, 24(1):954–963, 2017.

Appendix

This supplementary section explores the question "how much does the data set impact the performance of particle advection based algorithms?," posed as ***RQ5*** in addition to the questions in the main paper. For this exploration, four additional datasets were considered. The first three of these are vector fields over a 512^3 uniform grid.

- ***Astro*** comes from a GenASiS simulation code [16] of a magnetic field simulation surrounding a solar core collapse that results in a supernova.
- ***Fusion*** comes from the NIMROD simulation code [33] of a magnetic field in a fusion tokamak device used to model the behavior of burning plasma.
- ***Fishtank*** comes from the NEK5000 simulation code [17] of a fluid flow inside a chamber when water of different temperatures is injected through a small inlet.
- ***Zero***, consists of a single hexahedron with all vectors having zero magnitude. This data set served as a reference for cache performance.

RQ5: How Does Data Set Impact Performance?

Table 8: The configuration for experiments that were executed to answer **RQ5**. Based on the parameters used, a total of 936 experiments (of the possible 1080) were considered to answer this question of which 72 were run with Zero data set and served as a baseline ("Zero" data set used only one seeding volume).

Parameter	Value	Total
Data Sets	All	5
Seed Volume	All	3
Seeds	{100, 10000, 1000000}	3
Duration	All	3
Algorithm	Particle Advection	1
Hardware	CPUs (Multi), GPUs (w/o Xfer)	8

This section considers the effect from data set on performance on GPUs and multi-core CPUs. Table 8 shows the parameters for the experiments we conducted for this phase. Data set can affect performance in two fundamental ways: caching and divergence. With respect to caching, some data sets may attract particles to key regions, potentially increasing cache performance, while other data sets may move particles throughout the region uniformly, potentially decreasing cache performance. With respect to divergence, the fundamental issue is early termination, i.e., if a particle is supposed to advect for a duration of N steps, but it stops after K steps, where $K < N$. This early termination can happen because a particle entered a zero-velocity region (making further steps unnecessary) or because it exited the data set's spatial domain (making further steps impossible). In the context of a parallel architecture, the effect of early termination is that some threads will be asked to do asymmetric work — threads assigned particles that terminate early will perform fewer advection steps — which can create performance degradation due to divergence.

Measuring the effect of data set on performance is non-trivial. The execution time for a given workload on a given architecture reflects a combination of factors: clock speed, number of steps taken, and hardware efficiency (i.e., caching and divergence). Since our interest is on the hardware efficiency changes due to data set, we normalize our analysis with respect to clock speed and number of steps taken. We did this normalization by using a reference data set, which refer to as "Zero." This data set consists of a single cell with velocity value $(0,0,0)$ everywhere in the cell. For a workload W, a hardware architecture H, and a data set D, our analysis used the following terms:

- $N_{W,H,D}$: the number of advection steps performed for workload W, hardware architecture H, and data set D.
- $T_{W,H,D}$: the execution time for workload W, hardware architecture H, and data set D.
- $A_{W,H,D}$: the average time per step for workload W, hardware architecture H, and data set D. This is calculated as $A_{W,H,D} = \frac{T_{W,H,D}}{N_{W,H,D}}$.
- $Norm_{W,H,D}$: the normalized time per advection step for workload W, hardware architecture H, and data set D. This is calculated relative to the Zero data set as $Norm_{W,H,D} = \frac{A_{W,H,D}}{A_{W,H,Zero}}$.
- $AggrNorm_{H,D}$: the aggregated time per advection step for hardware architecture H and data set D over all six workloads. This aggregation is performed with a geometric mean, i.e., $(\prod_{w=1}^{w=6} Norm_{w,H,D})^{\frac{1}{6}}$.

To understand the meaning of these terms, consider an example. If Workload W4, hardware architecture GPU2, and data set Astro, have $Norm_{W4,GPU2,Astro} = 1.2$, then the average step was 20% slower than for the Zero data set. Further, if $AggrNorm_{GPU2,Astro} = 1.4$, then then the average step was 40% slower than for the Zero data set over all workloads. The remainder of this section focuses on *AggrNorm* values. That said, the individual *Norm* values for each comparison (over the four data sets, eight hardware architectures, and six workloads) can be found in the supplemental material,

Table 9: *AggrNorm* values for all combinations of hardware architectures and data sets. Each entry cell shows the aggregate slowdown (over six advection workloads) for a given data set compared to the *Zero* data set on a given architecture. For example, the number ***1.08*** in the top left of the table means that the Fusion data set took an average of 8% longer than the Zero data set on Xeon 8 architectures.

	Fusion	Astro	Fishtank	Noise
CPU1	1.08	1.02	1.35	0.98
CPU2	1.35	1.13	1.85	1.13
CPU3	1.23	1.11	1.76	1.05
CPU4	1.12	0.99	1.61	0.93
GPU1	2.77	2.49	7.04	1.97
GPU2	3.13	2.73	8.15	1.97
GPU3	4.99	4.17	9.55	3.90
GPU4	3.73	3.03	9.58	2.35

Table 9 contains the *AggrNorm* values for each combination of hardware architecture and data set. There are two primary findings from this table: effects from hardware architecture and effects from data set. With respect to hardware architecture, the impact for CPUs is significantly less than that for GPUs. The largest *AggrNorm* value for CPUs is 1.85, while the largest value for GPUs is 9.58. This means that our experiments showed at worst an 85% slowdown on CPU architectures, compared to a 858% slowdown for GPUs. Further, GPUs were almost always

twice as slow compared to the Zero data set, while CPUs were able to run nearly as quickly in some cases. In short, caching and divergence affected the CPUs much less than the GPUs. While this is an expected outcome, the magnitude of the effect (1.85 vs 9.58) was surprising. Another finding for hardware was that the *AggrNorm* values climbed on each subsequent generation of GPU, from an average of 3.12 on GPU1 to an average of 3.99 on GPU4. So while later GPUs offer higher performance, degradations due to data set effects become more prominent. With respect to data set, Table 9 informs the extent of slowdown due to data set. The Noise data set (which has few zero-velocity spots and does not regularly push particles outside its spatial boundary) is able to perform similarly to the single-cell Zero data set on CPUs, and performs better than the other data sets on GPUs. The Fishtank data set is the clear worst performer, with GPU performance being approximately three times slower than the other data sets. Once again, while this type of effect is to be expected, we found the magnitude of this effect to be surprising. That said, more analysis is needed to figure out the cause of the slowdown: divergence, caching, or both.

Figure 10: Figure demonstrating the proportion of fake steps contributed by different termination criteria when particles are not terminated at all.

Figure 10 and Table 10 isolate the effects from divergence. As stated earlier, divergence occurs because of early termination. Our solution is to modify our algorithm to not terminate these particles — if a particle hits a zero-velocity region, then the algorithm continues performance advection steps (and remaining

Table 10: Table demonstrating the relative slowdown in terms of time per step for different architectures when particles are not terminated at all. For example, for Xeon 8 CPU and Fusion data set the number ***1.06*** represents that the experiment is 1.06x slower than the ideal case.

	Fusion	Astro	Fishtank	Noise
CPU1	1.06	1.02	1.22	0.99
CPU2	1.14	1.05	1.42	1.11
CPU3	1.11	1.07	1.28	1.02
CPU4	1.02	0.97	1.26	0.94
GPU1	1.84	1.62	1.75	1.61
GPU2	1.54	1.46	1.78	1.48
GPU3	3.42	3.02	3.70	3.06
GPU4	2.28	2.07	2.62	1.90

in the same position) and if a particle exits the spatial domain, then we have it advect backwards in time (i.e., retrace the path from where it came). For ease of reference, we refer to these additional steps as "fake steps." Figure 10 plots the proportion of "real steps" versus "fake steps" (differentiating between those from zero velocity and those from exiting the boundary) for all data sets, workloads, and seeding volumes. It shows that:

- the Fusion data set is made up of 50% fake steps due to zero velocity (or, rather, that the number of steps taken under normal conditions is 2X less than the workload specifies),
- the Astro data set has a large range of outcomes (from 5% fake steps to 70% fake steps, mostly from exiting the boundary),
- the Fishtank data set always has more than 50% fake steps (from a combination of zero velocity and exiting the boundary), and
- the Noise data set has the least number of fake steps, although workloads with longer durations due ultimately find zero-velocity locations.

These results inform Table 10, which repeats the analysis from Table 9 but incorporates fake steps in the timings. As a result, these experiments have no effects from divergence and slowdowns are solely from caching effects. On the CPU side, Table 10 shows that the Fishtank data set is still slower than the other data sets. In this data set, a given particle will hit the ceiling of an assembly and can move in any direction, and then will recirculate and hit the ceiling again. As a result, this data set is the one that most stresses cache, as each particle can travel through the entire volume. The slowdowns for Fishtank and Fusion are roughly half of those in Table 9, while for Astro and Noise the slowdowns are very similar. In all, these experiments show that data set does affect CPU performance — two data sets appear to be affected by caching and divergence in approximately equal measure, while two other data sets appear to be affected only by caching. On the GPU side, eliminating divergence improved slowdown factors for all four data sets. The biggest effects were for Fishtank, with the GPU4 experiments dropping from a slowdown of 9.58 in Table 9 to 2.62 in Table 10. For this data set, divergence was a much larger cause of slowdown than caching. The effects are smaller for other data sets. For example, the Noise data set on GPU4 improved from 2.35 to 1.90. For this data set, cache performance appears to be a bigger issue than divergence. We are reluctant perform further analysis to quantify the relative effects of each.

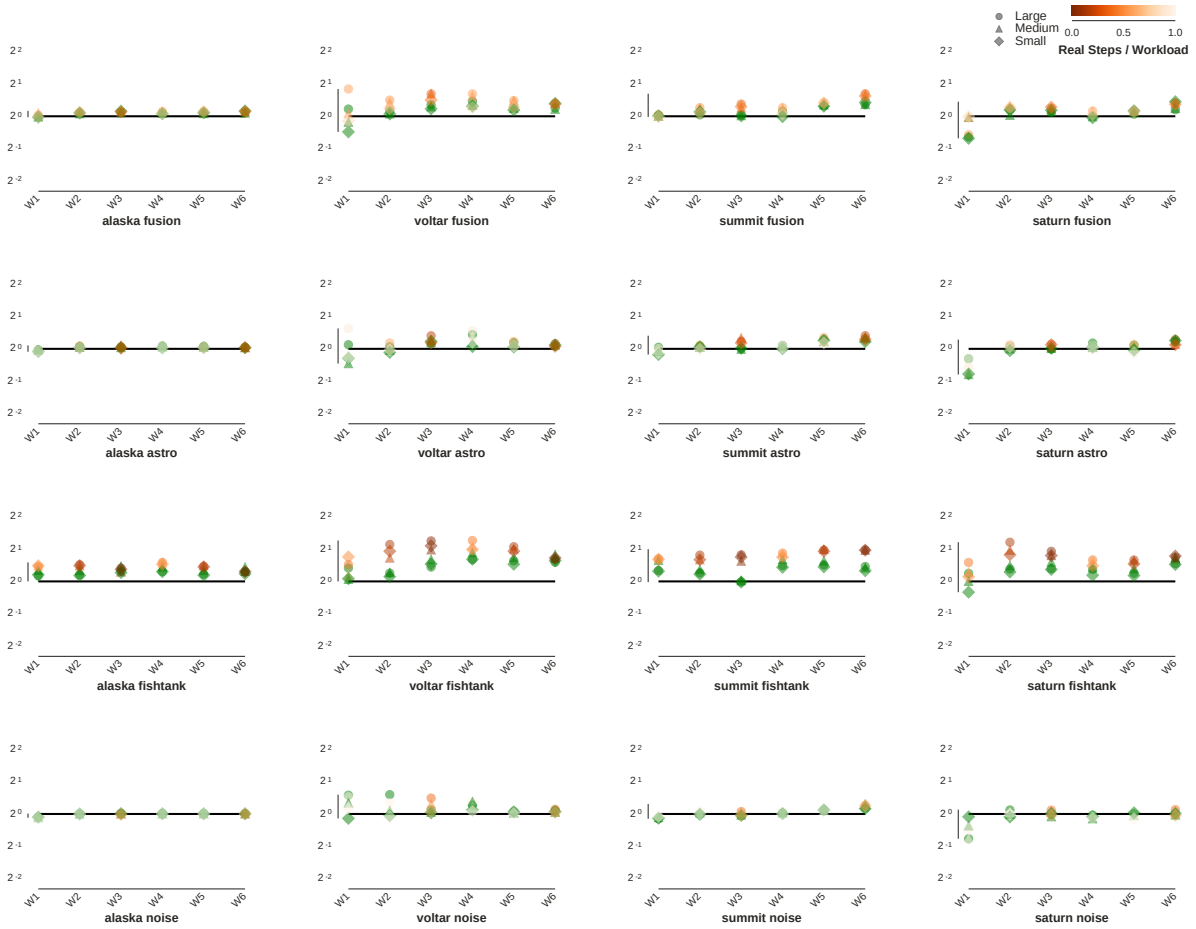

Figure 11: Comparison of CPU performance for different data sets. The Y-axis represents the ratio of time taken per step against the ideal case (equivalent experiment executed on the "Zero" data set), while the X-axis represents the workload. The different glyphs represent the seeding volume of the experiment. The green dots represent experiments where particles are not terminated at all. In contrast, the orange dots represent experiments where particles are terminated whenever they encounter zero-velocity regions or encounter spatial boundaries. Here a higher Y axis represents the experiment was y times slower than the ideal case.

In particular, the experiments in Table 10 have extra cache benefits from zero-velocity steps, which contributes to the reduction in slowdown.

Impact of Data Set on CPU and GPU Performance

Section evaluated the impact of data sets on the performance of particle advection. This appendix provides additional analysis to understand this impact better by expanding the data summarized in Table 9 and 10. Section presents the impact of data sets for the CPUs, and Section presents the impact of data sets for the GPUs.

CPU Performance on Different Data Sets

Figure 11 helps understand the outcomes for the different data sets against the "Zero" dataset in two ways. First, it considers the performance without terminating the particles (glyphs represented in green) to uncover the impact of caching and memory accesses involved in particle advection. For almost all CPUs and data sets, these experiments performed very close to the ideal case, implying that the impact of caching and memory accesses on CPUs for particle advection is minimal. Second, it considers the performance of terminating the particles normally (glyphs represented in oranges) to uncover the impact of divergence in work for different particles. Again, the experiments performed very close to the ideal case for almost all CPUs data sets, implying that the impact of divergence on CPUs for particle advection is minimal. The anomaly to both these observations was the Fishtank data set, which shows the effects of both poor caching and more divergence. Although Figure 10 demonstrates that while not terminating particles for the Fishtank data set leads to more cache-friendly steps (zero velocity), the particles subjected to this data set also suffer a high variance in the amount of duration contributing to real steps. These variances are the primary reason for this anomaly.

GPU Performance on Different Data Sets

Figure 12 presents a similar analysis as Section , but for GPUs. However, in contrast to CPUs, GPUs performance is impacted significantly due to caching and divergence. Across all

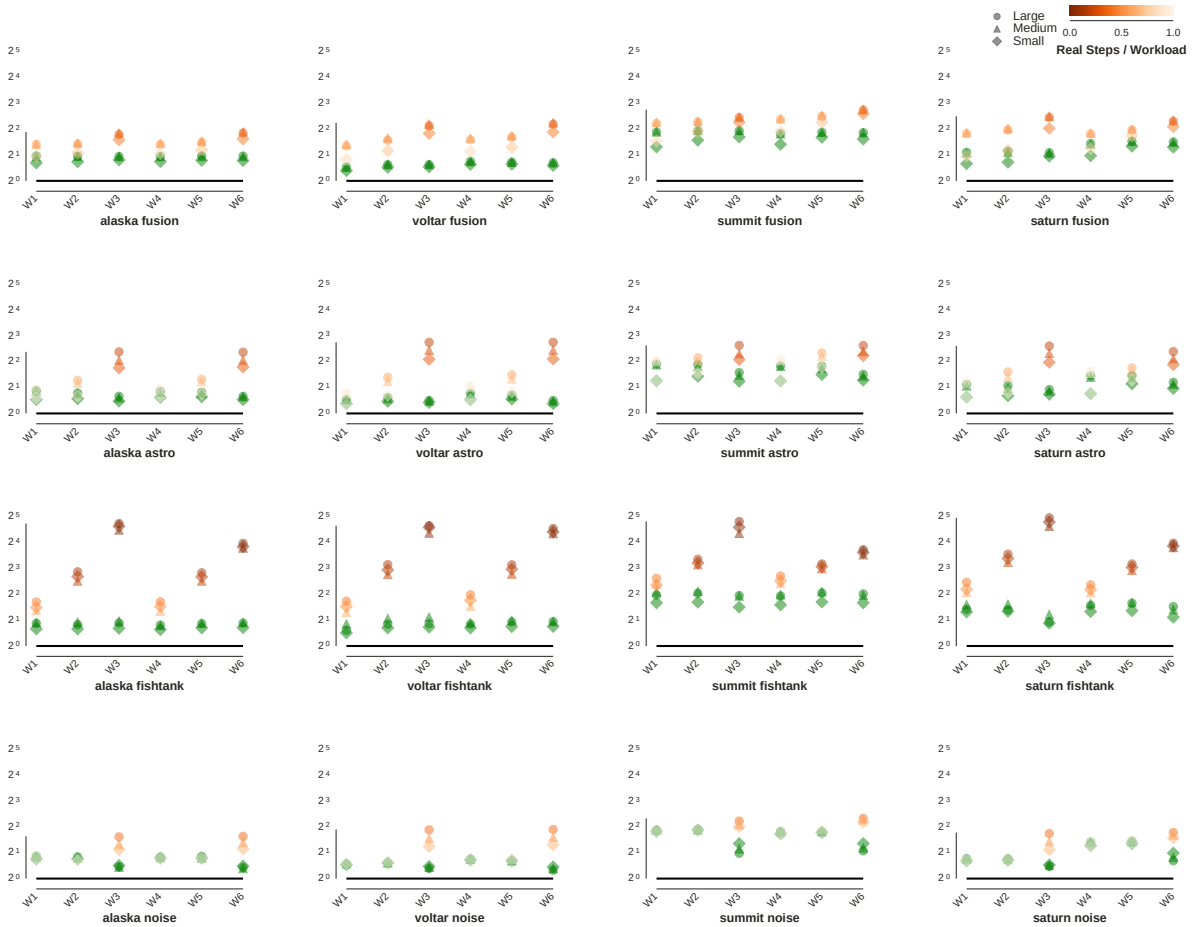

Figure 12: Comparison of GPU performance for different data sets. The Y-axis represents the ratio of time taken per step against the ideal case (equivalent experiment executed on the "Zero" data set), while the X-axis represents the workload. The different glyphs represent the seeding volume of the experiment. The green dots represent experiments where particles are not terminated at all. In contrast, the orange dots represent experiments where particles are terminated whenever they encounter zero-velocity regions or encounter spatial boundaries. Here a higher Y axis represents the experiment was ***y*** times slower than the ideal case.

the data sets, the two newer GPUs (GPU3 and GPU4) performed poorly in experiments studying caching (green glyphs). In other words, the newer GPUs can offer much better performance when memory accesses for an application are cache friendly. Particle advection involves random memory accesses, which hurts GPU performance. Further, the performance of a particle advection for a data set is tied to the amount of real work that the algorithm performs proportional to the workload (orange glyphs). For the case of divergence, the takeaway is that a low amount of real work relative to the workload results in poorer performance against the ideal case.

Author Biography

Abhishek Yenpure received his BE in Information Technology from the University of Pune (2013) and his Ph.D. in Computer Science from the University of Oregon (2022). Since then he has worked in the Scientific Computing team at Kitware Inc. His work is focused on scientific visualization and the usage of accelerated computing (GPUs, multi-core CPUs) to improving visualization workflows.

David Pugmire is a Distinguished Scientist at Oak Ridge National Laboratory and a Joint Faculty Professor in the Electrical Engineering and Computer Science Department at the University of Tennessee. He received his Ph.D. in Computer Science from the University of Utah in 2000. His research interests are in large-scale parallelism for analysis and visualization of scientific data.

Hank Childs is a professor of Computer Science at the University of Oregon. He received a Ph.D. in Computer Science from the University of California at Davis in 2006. His research focuses on Scientific Visualization, High Performance Computing, and the intersection of the two.