# EFFICIENT PARALLEL PARTICLE ADVECTION VIA TARGETING DEVICES

by

KRISTI BELCHER

A THESIS

Presented to the Department of Computer and Information Science
and the Graduate School of the University of Oregon
in partial fulfillment of the requirements
for the degree of
Master of Science

June 2020

THESIS APPROVAL PAGE

Student: Kristi Belcher

Title: Efficient Parallel Particle Advection Via Targeting Devices

This thesis has been accepted and approved in partial fulfillment of the requirements for the Master of Science degree in the Department of Computer and Information Science by:

| | |
|---|---|
| Hank Childs | Chair |
| Michal Young | Committee Member |
| Allen Malony | Committee Member |

and

| | |
|---|---|
| Kate Mondloch | Interim Vice Provost and Dean of the Graduate School |

Original approval signatures are on file with the University of Oregon Graduate School.

Degree awarded June 2020

## THESIS ABSTRACT

Kristi Belcher

Master of Science

Department of Computer and Information Science

June 2020

Title: Efficient Parallel Particle Advection Via Targeting Devices

Particle advection is a fundamental operation for a wide range of flow visualization algorithms. Particle advection execution times can vary based on many factors, including the number of particles, duration of advection, and the underlying architecture. In this study, we introduce a new algorithm for parallel particle advection which improves execution time by targeting devices, i.e., adapting to use the CPU or GPU based on the current work. This algorithm is motivated by the observation that CPUs are sometimes able to better perform part of the overall computation since CPUs operate at a faster rate when the threads of a GPU can not be fully utilized. To evaluate our algorithm, we ran 162 experiments and compared our algorithm to traditional GPU-only and CPU-only approaches. Our results show that our algorithm adapts to match the performance of the faster of CPU-only and GPU-only approaches.

## CURRICULUM VITAE

NAME OF AUTHOR: Kristi Belcher

GRADUATE AND UNDERGRADUATE SCHOOLS ATTENDED:

University of Oregon, Eugene, OR, USA
Texas State University, San Marcos, TX, USA

DEGREES AWARDED:

Master of Science, Computer Science, 2020, University of Oregon
Bachelor of Science, Computer Science, 2016, Texas State University

AREAS OF SPECIAL INTEREST:

High Performance Computing
GPU Programming

PROFESSIONAL EXPERIENCE:

Graduate Research Assistant, University of Oregon, 2017-2020
Intern, Oak Ridge National Laboratory, 2019
Undergraduate Research Assistant, Texas State University, 2015-2016

GRANTS, AWARDS AND HONORS:

Phillip Seeley Graduate Research Award, University of Oregon, 2018
Graduate Research Fellowship, National Science Foundation, 2017

## ACKNOWLEDGEMENTS

I would like to thank my advisor, Hank Childs, without whom this work would not be possible. I would also like to thank Abhishek Yenpure and David Pugmire for their contributions in this work. Finally, I would like to thank my family for their never ending support and inspiration. This research used resources of the Oak Ridge Leadership Computing Facility, which is a DOE Office of Science User Facility supported under Contract DE-AC05-00OR22725. Additionally, this material is based upon work supported by the National Science Foundation Graduate Research Fellowship under Grant No. 1842486.

TABLE OF CONTENTS

## LIST OF FIGURES

# CHAPTER I

# INTRODUCTION

Particle advection is a fundamental operation for flow analysis and visualization, used for techniques such as streamlines, pathlines, stream surfaces, and Finite-Time Lyapunov Exponents (FTLE). Particle advection is used to determine the trajectory a particle follows after being placed at some starting point (i.e., seed location). The particle locations are calculated using an underlying vector field. Additionally, since particle behavior is based on some vector field (i.e. velocity), a particle moves depending on the data set and its trajectory is not known a priori. The particle advection calculation is an iterative operation that results in a particle location moving along the domain (this advancement is referred to as an *advection step*). The total number of steps that will be calculated depends on the number of particles and the duration of particle advancement.

Implementing an efficient large-scale parallel particle advection operation remains a challenge. Particle advection is a computationally heavy, data-dependent operation where the amount of work involved varies by workload. Depending on the number of total particles in the workload, there could be millions if not billions of total advection steps. Thus, efficiently using resources and coordinating those resources is key for good performance.

Additionally, as data sets become larger and larger, they can no longer fit into the memory of just a single computer. To address this limitation, large-scale particle advection simulations are run on modern supercomputers so that efficient results, even on the most computationally heavy workloads, can be obtained in a reasonable amount of time.

Modern supercomputers have varied hardware architectures, ranging from modest computational power (CPUs) to very high computational power (GPUs) [11]. For particle advection, the device architecture plays a significant role in overall performance. While previous approaches have focused on a single architecture (i.e., supporting only CPU or only GPU), this research considers utilizing both architectures. For many workloads, there may not be enough work to fully load, and therefore take full advantage of, the GPU. For other kinds of workloads, there may be an overwhelming amount of work for the CPU. Therefore, it may be better to target one device during certain stages of the particle advection calculation, then switch to the other device for the remaining stages. Our new algorithm uses this idea and continually targets a device (CPU or GPU) as the parallel particle advection work evolves.

This study contributes a new hybrid algorithm that efficiently performs a particle advection calculation via targeting the appropriate device. Our algorithm is motivated by the observation that sometimes the workload cannot take full advantage of GPU threads, making it slower than just using the CPU. In such cases, leaving the data on the CPU to be computed would perform better. At other times, the workload may be much more able to fully load the GPU and thus take advantage of its hardware capability. These observations are reinforced by our results, which show that when running workloads that typically perform better using CPU-only or GPU-only, our algorithm targets the appropriate device and performs similarly. In some cases, our algorithm performs better than either device because it can target and switch to the better device as the algorithm runs. Overall, our work explores two related research questions on the role of device targeting for parallel particle advection: (1) **Can device targeting be used to**

**match the best performing single-architecture algorithm?** (2) **Can it be used to exceed both single-architecture implementations? If so, by how much?**

Important contributions of this paper include:

- A new parallel particle advection algorithm that determines whether the CPU or the GPU can best handle the current work, targeting that particular device in order to achieve a more performant result.
- An analysis of how certain conditions of the workload can be used to determine the appropriate device to target.
- A comparison of our results and findings with other implementations which use CPU-only or GPU-only and an analysis of our results to provide insight for visualization scientists who are studying particle advection and other flow visualization algorithms.

The remainder of the thesis is organized as follows. Chapter II discusses the background of our design and related previous works. Next, chapters III, IV, and V describe our algorithm, experiments, and results, respectively. Finally, chapter VI summarizes our findings and discusses future work.

# CHAPTER II

# BACKGROUND AND RELATED WORK

This chapter discusses the relevant background for this study. Section 2.1 describes the particle advection calculation and other pertinent background of the particle advection calculation, followed by a discussion of relevant parallelization paradigms. Next, section 2.2 discusses related works that focus on using accelerators or distributed memory environments for parallelization. Lastly, section 2.3 concludes with an explanation of hardware agnostic environments and their role in this study.

The closest work to my own is the study done by Camp et al. [8] (discussed in section 2.2). Camp et al. determined what workloads should be run on the CPU and what workloads should be run on the GPU. Our work expands on this idea to perform device targeting throughout execution. Explicitly, the key difference is that while the Camp et al. study determines one device to run the entirety of the algorithm on, our study can use either device or both depending on the work remaining.

## 2.1 Parallel Particle Advection

Because particle advection is a data-dependent and computationally heavy operation, it is important to understand how it works. Section 2.1.1 explains the particle advection operation in more detail. Next, section 2.1.2 gives an overview of several different parallelization paradigms used to implement parallel particle advection.

**2.1.1 Particle Advection Algorithm.** Particle advection starts by displacing a particle for a short distance in a vector field (e.g., velocity). This displacement is also known as an *advection step.* That particle will be advected from one location to another nearby. Particles typically advect and evolve through the problem domain, giving information about what is happening in the data. The sequence of advection steps from the initial location to the terminal location is referred to as the particle's integral curve. The integral curve shows the trajectory along which the particle travels. Because the advection steps must be calculated sequentially, advecting particles is a data dependent problem. In other words, calculating a particle's $N^{th}$ location requires knowing its $(N - 1)^{th}$ location. Additionally, the number of advection steps for any given particle varies, adding more complexity to the problem. For example, the particle could advect into a sink, go outside the problem domain, or meet some other kind of termination criteria. Thus, particle advection requires variable computational resources to successfully trace the particle trajectory.

**2.1.2 Parallelization Paradigms.** Because processing integral curves has remained a key computation for vector field visualization [15], much work has explored how to make this calculation fast and efficient. As data sets become very large, fitting all the relevant data into memory becomes challenging, and so the data has to be divided into blocks. Previous work has introduced two fundamental approaches to calculating the integral curves for vector field visualization [19, 13, 6]. First, *Parallelize-over-Particles* (POP) involves distributing the particle seeds across nodes to be processed independent of each other. Figure 1 shows a visual representation of the two parallelization paradigms used to implement particle advection, inspired by Camp et al. [6]. The right side of figure 1 demonstrates how three processors would parallelize three particles with the POP approach. POP often does not scale very well because of redundant I/O operations. For example, following the blue particle on the right side of figure 1 involves crossing over four

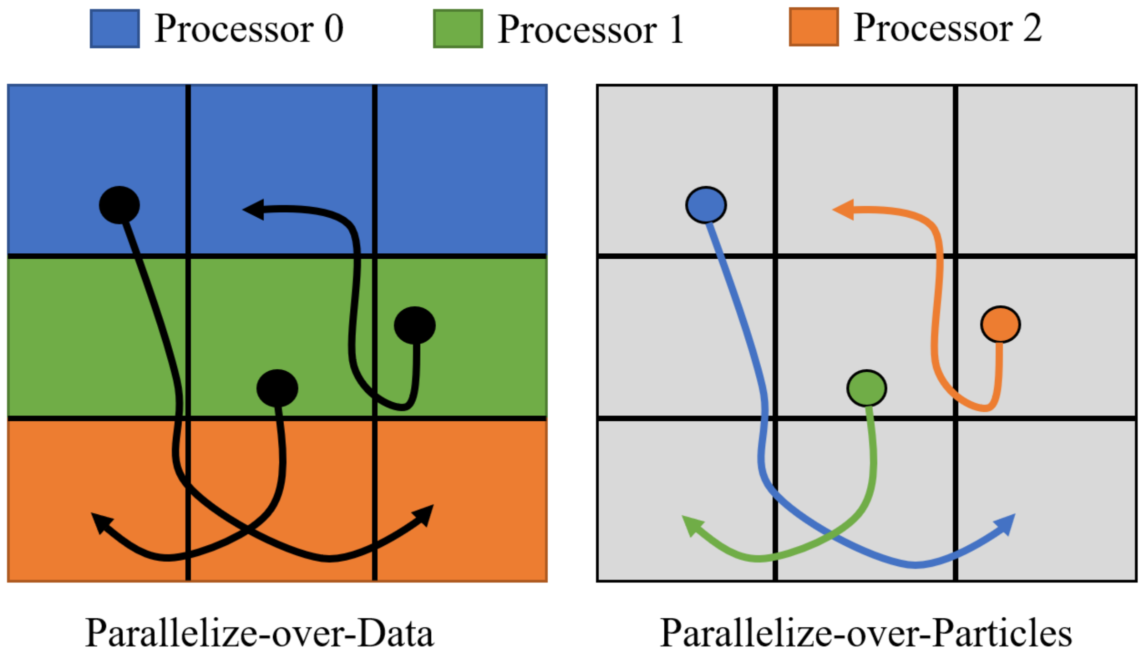


Figure 1. A diagram of the two major parallelization approaches for particle advection. The colors represent different processors, while the dots with arrows represent particles and where they advect.

boundary lines. This means that there will be four blocks that need to be loaded from disk. On a large scale, this kind of I/O-heavy approach could be detrimental to overall performance.

This leads to the second approach, *Parallelize-over-Data* (POD), where each node is assigned a data block of the whole domain to process. The node assigned to a particular data block is in charge of calculating traces for any particles that enter its block. The left side of figure 1 demonstrates how three processors would parallelize the three particles with the POD approach instead. Still, it is easy to see how load balancing will become a problem for this approach as not all pieces of the domain have a proportional amount of work. For example, from the depiction of POD in figure 1, Processor 0 (blue) will have less work than the other Processors since one of its blocks is untouched by particles. Therefore, it is common to use some kind of combination of both of these approaches to increase performance and alleviate the potential downfalls of each approach by itself. The next section explores the previous works which have studied solutions to that exact issue.

## 2.2 Distributed Memory and Accelerator Environments

There are many different ways to implement a parallel particle advection operation, each with performance benefits and drawbacks. This section analyzes previous works that have focused on distributed memory and/or accelerator parallelization for parallel particle advection.

Pugmire et al. [9] explored particle advection in a distributed memory parallel environment. Camp et al. [7] proposed a MPI and OpenMP based solution where MPI is used to parallelize across multiple nodes, and OpenMP is used to parallelize local advection calculations. They proposed a scheduling approach to manage how these two APIs interact. Then, in a separate work by Camp et al. [8], they proposed a similar schedule with CUDA implementations substituted for OpenMP. Here they analyzed the performance impact between the two.

Other previous work focuses on optimizing the POD approach. Yu et al. [25] proposed a data-aware POD approach. Similarly, Nouanesengsy et al. [22] used a statistical strategy to calculate the propagation probability of particles across blocks. They predicted the movement of particles to more efficiently partition blocks across nodes. In their work, they explored the POD approach and found a unique way to optimize it. Additionally, Nouanesengsy et al. [18] also used time intervals as a way of optimizing how blocks get assigned to the different nodes. With this strategy, they can more efficiently compute the Finite Time Lyapunov Exponent (FTLE) by distributing different time intervals to nodes and generating pathlines accordingly.

Many previous works focused on optimizing POP and POD approaches by incorporating load balancing or *work-requesting* between nodes. Müller et al. [17] proposed a *work-requesting* scheduling scheme which alleviated much of the load imbalance problem of POP for streamlines. In their approach, data blocks are assigned to nodes and when one node runs out of work to do on its assigned block, it sends a message to a neighboring block requesting half of its remaining work. In this way, most of the load imbalance problem is solved unless there are other imbalance issues in other areas of the problem. Similarly, Lu et al. [14] proposed a scheduling scheme, Random Scheduling Method (RSM), for stream surfaces that was designed to minimize idle time by optimizing how nodes communicate with each other when requesting work. Binyahib et al. [3] proposed a Lifeline Scheduling Method (LSM) that improves upon RSM [5] by optimizing how nodes request and receive work from other nodes.

## 2.3 Hardware Agnostic Environments

This section discusses the use of software frameworks which can abstract away the specifics of using any one particular hardware device for parallelization. This is important background because this study builds off of a hardware agnostic framework. Although there are many such frameworks (i.e., Kokkos [10] and Raja [1]), VTK-m is the framework used in this study.

VTK-m [16] is a parallel visualization framework used heavily within the scientific visualization community. Pugmire et al. [20] used VTK-m to parallelize particle advection with MPI communication. Similarly, this study uses VTK-m as the backbone for which our new algorithm was built upon. Thus, this section briefly explains VTK-m and some of its key features that made our new algorithm possible.

In 1990, Blelloch proposed a model focusing around Data Parallel Primitives (DPPs), where specific operations could be carried out on vectors of size $N$ in $O(\log(N))$ time in the worst case [4]. This worked inspired the use of DPPs such as Scan, Scatter, Map, Reduce, and Gather used in libraries like NVIDIA's Thrust [2] library. Libraries that use DPPs can be compiled to work across a variety of parallel architectures.

Performance-portable libraries such as VTK-m use DPPs as the basis for its hardware agnostic framework. VTK-m is a hardware agnostic solution for implementing key visualization algorithms across a wide range of platforms. The VTK-m framework was created to address the growing concern that as modern supercomputers are equipped with a wider range of hardware varieties, from programmable graphics processors (GPUs) and many-core co-processors (Intel Xeon Phi) to large multi-core CPUs, visualization software developers must still provide optimized software that can be deployed on those hardware platforms. VTK-m provides software developers with the ability to write a single implementation of their code and have it run on a variety of architectures obtaining excellent performance on each distinct platform. In short, VTK-m was created to provide the same functionalities as VTK [21] (the Visualization ToolKit), while also achieving portable performance across a variety of many-core and multi-core architectures.

# CHAPTER III

# ALGORITHM

This section begins with a discussion of our parallel particle advection algorithm. Namely, section 3.1 gives a line-by-line explanation of our algorithm. The key component, the Oracle, is discussed in more detail in section 3.2.

## 3.1 Algorithm Overview

In this study, our particle advection algorithm uses a *POD* approach. Our algorithm is designed for data that is so large that it must be broken into sub-domains. We handle this by partitioning the sub-domains across MPI ranks. Conceptually, our algorithm can easily support the case where there are $M$ sub-domains and $N$ ranks, with $M > N$. That said, our current implementation assumes that the number of MPI ranks and sub-domains are the same, i.e., $M = N$. Thus, each MPI rank will operate over those particles that are located within the bounds of its sub-domain.

Algorithm 1 shows our implementation which is performed by each MPI rank on its corresponding sub-domain. Line 1 calls the $Oracle()$ which is explained further by algorithm 2. This call will determine what device to target given the current work remaining. At this point in the code, the work remaining is the entire workload.

Line 2 starts by initiating a while loop which iterates throughout the progression of this algorithm. By the time this while loop is finished, all particles will have been processed and the algorithm will be complete. Line 3 calls a function which advects any currently active particles (i.e., those particles which can be processed immediately) until they either terminate or go out of bounds. The $AdvectParticles()$ function produces updated particle trajectories. The particles

**Algorithm 1** Advect a List of Particles

```
 1: Oracle()
 2: while $true$ do
 3:   AdvectParticles()
 4:   if !InParticles.Empty() then
 5:     Oracle()
 6:   end if
 7:   if !OutParticles.Empty() then
 8:     CommunicateParticles()
 9:     Oracle()
10:   end if
11:   if !TermParticles.Empty() then
12:     $totalNumTerm$ += $numTerm$
13:     if $totalNumTerm == totalNumParticles$ then
14:       $break$
15:     end if
16:   end if
17: end while
```

could be leaving the current MPI rank's sub-domain and thus be *outgoing*, some particles may be done advecting and therefore be *terminated*, or still other particles from another MPI rank could be entering the current rank's sub-domain and be *incoming*. Lines 4-14 check for and handle these three possible states.

Specifically, Line 4 checks if there are any incoming particles. If there are, the *Oracle*() function is called again because those incoming particles must be assigned a device to run on. If not, it continues on to Line 7 where it checks if there are any outgoing particles. If there are outgoing particles, they must be communicated (Line 8) to the corresponding MPI rank which is in charge of the sub-domain those particles have entered in to. Otherwise, it moves on to Line 11 where it checks for any terminating particles. If particles have terminated, then a counter variable, $totalNumTerm$, is incremented (Line 12) by the amount of particles that have terminated.

Once all particles across all MPI ranks have terminated, the algorithm will exit the while loop (Lines 13 and 14). If instead there are still more particles that need to be advected, then it loops back to Line 2. From there it will again process those newly active particles with *AdvectParticles*() and continue checking the resulting particle statuses. As the program progresses, the particles travel through the sub-domains, creating their trajectories.

## 3.2 Oracle

A key contribution of this algorithm is the Oracle design, along with the insight and benefits it provides to overall performance. The Oracle uses the amount of *work* left to do in order to determine whether the newly active particles are best suited for the GPU or not. The *work* is a function of the current particles already assigned to the GPU and any newly active particles.

In other words, if the Oracle determines that the GPU can still be fully loaded (i.e. there is still sufficient work to do), then the Oracle returns *true* to target the GPU to process those particles. Otherwise, the Oracle returns *false* and the CPU is targeted for execution instead. The goal here is to take advantage of the GPU's compute capability when there is sufficient work, but the same time, also taking advantage of the CPU's better latency handling to advect the last few, straggling particles (if any) that still need to be advected before becoming inactive or terminating.

**Algorithm 2** Oracle

```
1: if $work > threshold$ then
2:     UseGPU()
3: else
4:     UseCPU()
5: end if
```

Algorithm 2 is a simplified representation of the Oracle logic. Line 1 consists of an *if* statement that determines if the current *work* is sufficient for the GPU or not by using a threshold. The threshold value in this algorithm is based off preliminary experiments that determined where a cross-over point exists between those workloads that performed better on the CPU to those that performed better on the GPU. If the current *work* is sufficient for the GPU, then it will process the newly active particles (Line 2). Otherwise, on Line 4, the CPU is used to process them instead. Although this piece of logic is relatively simple, it allows the algorithm to target the appropriate device to run the current work left to do.

# CHAPTER IV

# STUDY OVERVIEW

The purpose of this chapter is to describe the details of the experiments conducted to evaluate our new algorithm described in the previous chapter. Section 4.1 describes the different configuration parameters of our experiments. Next, sections 4.2, 4.3, and 4.4 discuss the other important details of our experiments including hardware/software environments, the threshold value, and performance measurements, respectively. Lastly, limitations of the study are addressed in section 4.5 and additional miscellaneous details are covered in section 4.6.

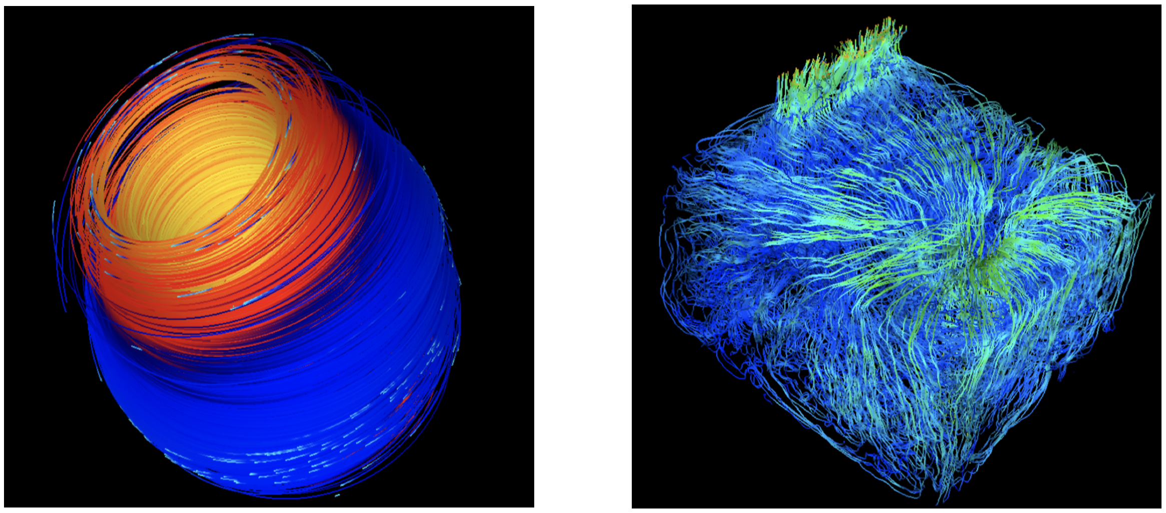
## 4.1 Algorithm Comparison Factors

To study the effect of adaptively selecting the target device during a particle advection computation, we conducted various experiments over a range of carefully tuned parameters. Our experiments varied over 3 main factors:

- **Data Set** (2 options)
- **Number of Particles** (9 options)
- **Number of Steps** (3 options)

Our experiments consisted of each of the different particle sizes, $p$, with each of the step counts, $s$, and for both data sets, $d$. These experiments also varied over implementation: CPU-only, GPU-only, and Oracle. Thus, there were a total of $3(p * s * d) = 162$ experiments[1] which were run on the Summit supercomputer at Oak Ridge National Laboratory [11]. The remainder of this section describes each of these three components in more detail.

[1]Each of the experiments was also ran three times for completeness. See section 4.6



(a) NIMROD Fusion simulation. (b) NEK5000 Fish Tank simulation.

Figure 2. Example visualizations of both data sets used in this study. These images are courtesy of Binyahib et al. [3].

#### 4.1.1 Data Set.

This study used different simulation codes that were used to thoroughly test the new algorithm. The goal was to test how the Oracle worked on different data sets that behave differently from each other.

Our simulations included:

– *Fusion*: The Fusion data set comes from the NIMROD [23] simulation code. In this particular simulation, a magnetically confined fusion reaction in a tokamak device causes the magnetic field to travel in a helical fashion around the torus to achieve stable equilibrium. The particles behave in a highly circular way as they traverse the torus-shaped vector field repeatedly.

– *Fish Tank*: This data set is from a thermal hydraulics simulation with the NEK5000 code [12]. Twin inlets pump water of differing temperatures into a box, creating mixing behavior and fluctuating water temperatures. In other words, the vector field of this simulation reflects fluid flow within a box, or tank.

Figures 2a and 2b show an example visualization of the simulations [3]. Using different data sets to test our new algorithm gives insight in to how it will perform across varying data. In section V, results for both data sets are shown.

**4.1.2 Number of Particles.** Testing a wide range of particles is crucial for finding important and interesting results. The number of particles varied from 8, 216, 1000, 4096, 8000, 27000, 512000, 1000000, to 8000000 in our experiments.

This range of numbers includes some that should better run with a CPU (smaller numbers), some that are better run with a GPU (very large numbers), and some in between that the Oracle should excel at. These numbers are given by 1, 3, 5, 8, 10, 15, 40, 50, and 100 (respectively), cubing that number, and multiplying by 8. The cube of the numbers is used because this works well with how the data is arranged in memory. Additionally, the numbers are multiplied by 8 since there are 8 MPI ranks.

**4.1.3 Number of Steps.** Our experiments used 100, 1000, and 10000 steps. The step size remained constant at 0.001. This way we could also see how varying our steps would effects the decisions of the Oracle.

## 4.2 Hardware and Software Used

As mentioned above, these experiments were run on the Summit supercomputer at Oak Ridge National Laboratory. Summit has 4608 nodes where each node has two IBM Power9 CPUs and six NVIDIA Volta GPUs [24]. Each experiment was run over two such nodes.

As described in section 2.3, all experiments used the VTK-m framework. VTK-m provides a particle advection operation which, for this study, has been partitioned over eight MPI ranks for all experiments. Each MPI rank made use of one GPU and one CPU core.[2]

## 4.3 Threshold Values

Preliminary experiments were performed to inform a cross-over point. The cross-over point describes the workload at the point where the CPU stops being the better performing single-device implementation and the GPU takes over. In this paper, the amount of work at the cross-over point is referred to as the *threshold* value. The *threshold* is discussed further in chapter III. For the experiments done in this study, the value 100 was used as the threshold. Again, note that the threshold is calculated in terms of units of work.

Previous work suggested the cross-over point should be around 1000 particles (discussed in chapter II). Since this value is data-dependent, the threshold was evaluated for each data set.

## 4.4 Performance Measurement

The performance of the new algorithm was given by the total time the particle advection work took to complete. This runtime includes the advection time, sleep time, communication time, etc for all MPI ranks. However, it does not include I/O times to load domains from disk. Excluding this time is consistent with previous studies, especially because supercomputer I/O performance can be inconsistent.

## 4.5 Limitations

An important limitation of our study to take in to consideration is that while the GPU is parallelized with CUDA, the CPU in our study is running serially. Although MPI provides some kind of distributed memory parallelism

[2]See section 4.5 for more on this.

across nodes, more parallelism, and potentially better performance, could be had if the CPU was also parallelized with OpenMP. A solution to this limitation is currently under development.

## 4.6 Additional Information

The goal of this section is to provide other important miscellaneous information about our study. First, in order to verify that the correctness of the program was maintained throughout all our various code modifications, the resulting particle trajectories were saved out and reviewed. Secondly, for the purpose of minimizing "OS-Jitter", each of our experiments, across both data sets, was repeated three times. In total, we ran $(3 * 162) = 486$ experiments. The lowest runtime of the three was included in the reported results of chapter V. Those runtimes have been normalized using a $log_{10}$ scale to make the charts and figures of results more readable.

# CHAPTER V

# RESULTS

Our results show that when there is a workload that is best suited for a CPU, our Oracle is similar to that runtime. When there is a workload that is better suited for a GPU, our Oracle instead is similar to that runtime. In effect, in most cases our Oracle runtime, $O$, can be reflected by a formula ($O \approx \min(G_t, C_t)$), where $G_t$ represents the runtime for the GPU-only implementation and $C_t$ represents the runtime for the CPU-only approach. There is also a subset of workloads which exist in a *sweet-spot* range where $O$ is better than both $G_t$ and $C_t$. For both data set results, the Oracle implementation had the fastest runtime a majority of the time. This behavior was caused by the Oracle's ability to target the appropriate device according to the current work remaining. Usually, in the cases where the Oracle did not have the fastest time, it was slower by just a fraction of a second. The remainder of this section describes these Oracle results in more detail.

## 5.1 Big Picture Performance

For the Fusion data set, the Oracle performed very well across all of our experiments. Although the Oracle did well at 100 steps, it did even better at 1000 or 10000 steps. Generally, for experiments with less than or equal to 1000 particles, the CPU had the winning single-device runtime. At 4096 and greater particles, the GPU started to have the winning single-device runtime. In most cases, the Oracle exceeds the faster single-device performance since it is able to target the appropriate device for the work remaining. Additionally, in most of the cases where the Oracle does not have the fastest performance, it is just slightly slower than the fastest single-device implementation. In the best case, the Oracle had a

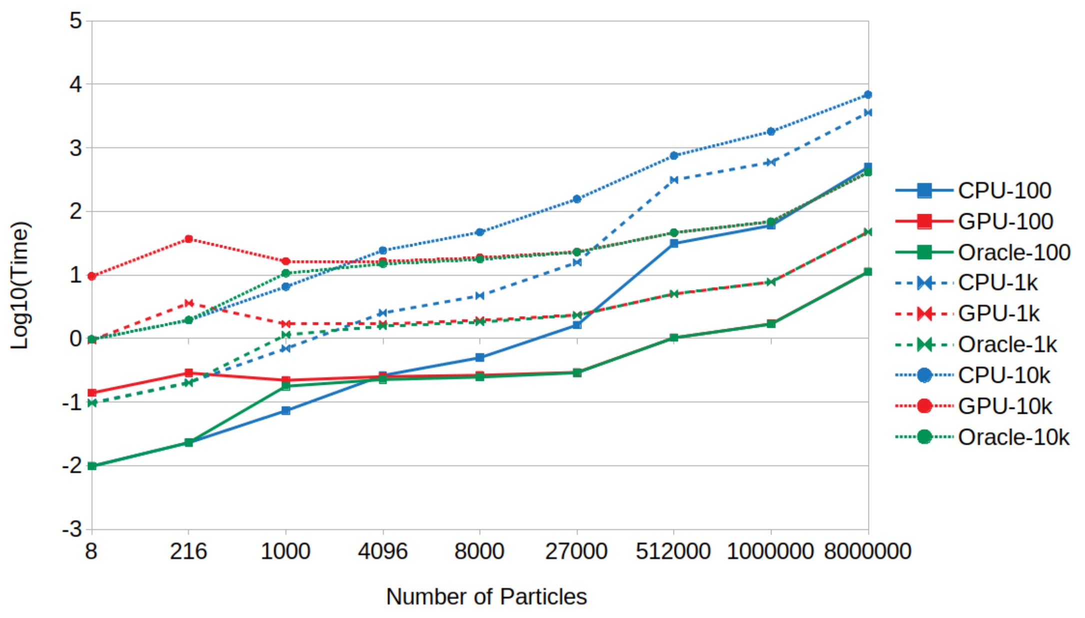


Figure 3. Execution times for all experiments with the Fish Tank data set. Dotted lines and circular glyphs correspond to 10000 steps, dashed lines and diamond glyphs correspond to 1000 steps, and solid lines with square glyphs correspond to 100 steps. Blue denotes the CPU-only tests, red denotes the GPU-only tests, and green denotes the Oracle tests.

10% speedup over the GPU (the winning single-device implementation) and 63% speedup over the CPU.

Figures 3 and 4 show the performance of the CPU-only (blue), GPU-only (red), and Oracle (green) implementations with the Fusion data set for 10000 (dotted lines with circular glyphs) steps, 1000 (dashed lines with diamond glyphs) steps, and 100 (solid lines with square glyphs) steps experiments. In regard to all figures presented in this chapter, the x-axis is the number of particles, while the y-axis is the runtime. Again, note that the y-axis is on a $\log_{10}$ scale to make the figures more readable.

Both figures show that the GPU-only version has much slower performance than the Oracle or CPU-only implementations for the small numbers of particles.

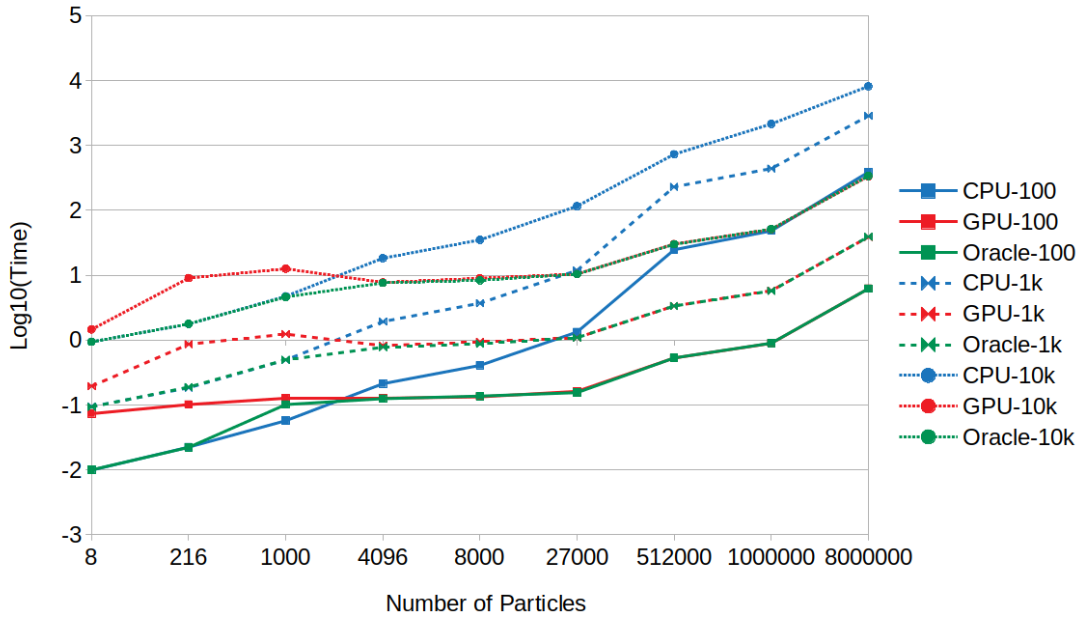


Figure 4. Execution times for all experiments with the Fusion data set. Dotted lines and circular glyphs correspond to 10000 steps, dashed lines and diamond glyphs correspond to 1000 steps, and solid lines with square glyphs correspond to 100 steps. Blue denotes the CPU-only tests, red denotes the GPU-only tests, and green denotes the Oracle tests.

As the number of particles increases, the GPU-only performance increases until there is a cross-over point where the CPU-only version starts to perform worse. Meanwhile, the Oracle implementation performs similarly to the CPU-only version when the number of particles is small, but starts performing more similarly to the GPU-only version once that cross-over point happens. Although the trend in both figure 3 and figure 4 is similar, for the Fish Tank data set, the Oracle does not give as much of a performance boost as it did with the Fusion data set. In other words, the Oracle has a greater speedup over both the single-device implementations for the Fusion data set experiments than with the Fish Tank data set. Thus, the amount of added performance is dependent on the data set.

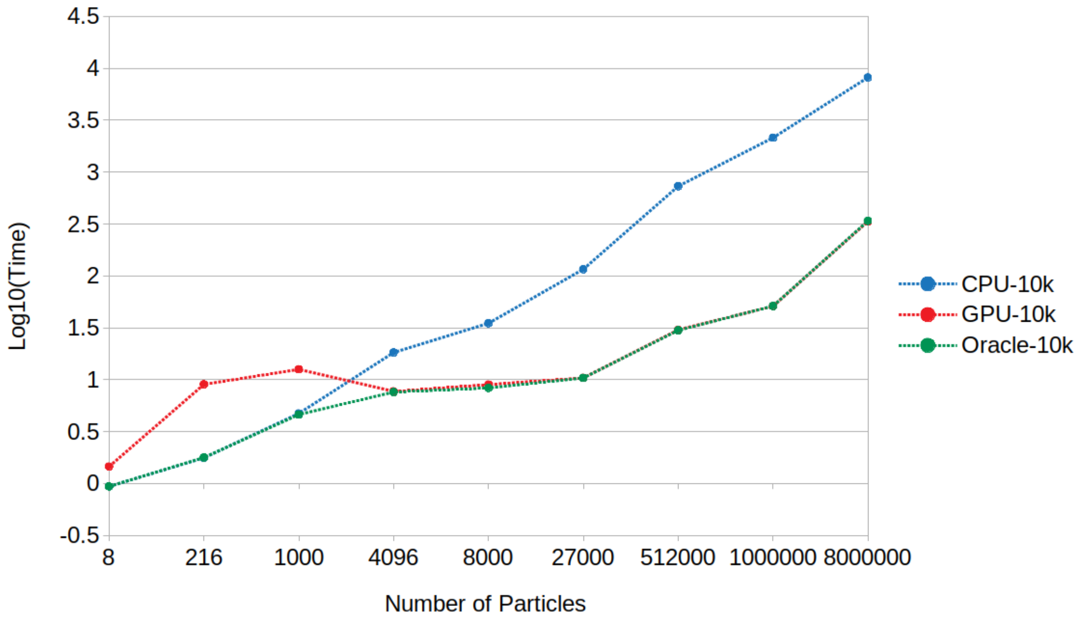


Figure 5. Execution times for the Fusion data set with 10000 steps.

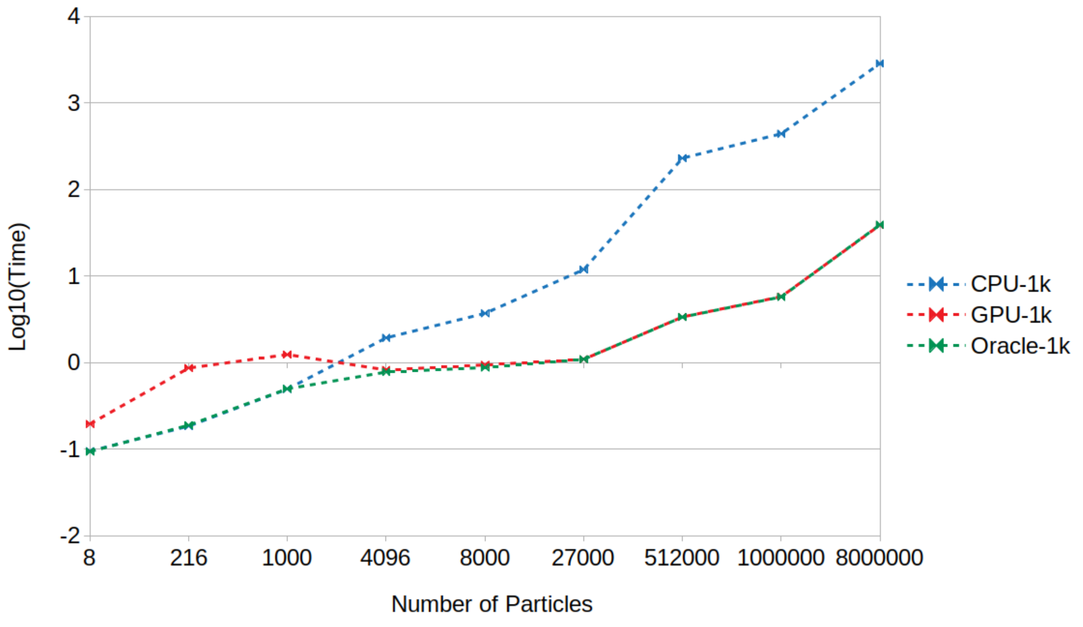


Figure 6. Execution times for the Fusion data set with 1000 steps.

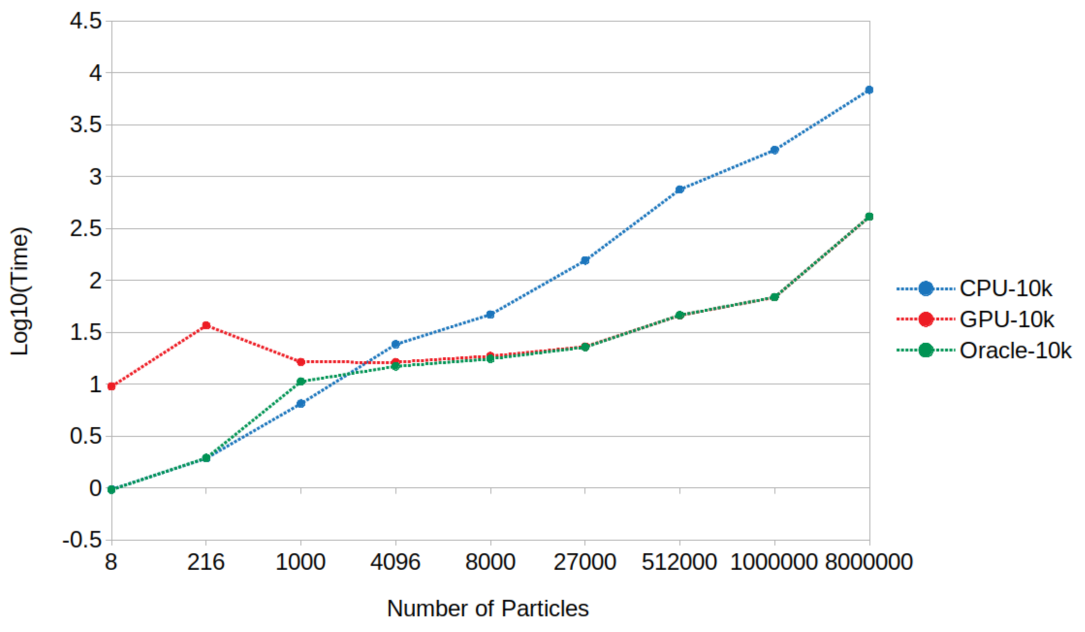


Figure 7. Execution times for the Fish Tank data set with 10000 steps.

To study this trend more closely, figures 5 and 6 show the performance of the Fusion data set with 10000 and 1000 steps, respectively. These experiments were shown separately because they more clearly show the performance trends of the Oracle.

In all experiments with the Fusion data set, there is a similar trend of the CPU starting with better performance than the GPU, and eventually the GPU having better performance once the problem size gets large enough. The Oracle tries to mimic the performance of the faster single-device implementation. In all three figures, the Oracle starts by targeting the CPU for the smaller problem sizes. As the workload increases, the Oracle targets the GPU more and more. All the while, the performance of the Oracle is closest to the best single-device performance.

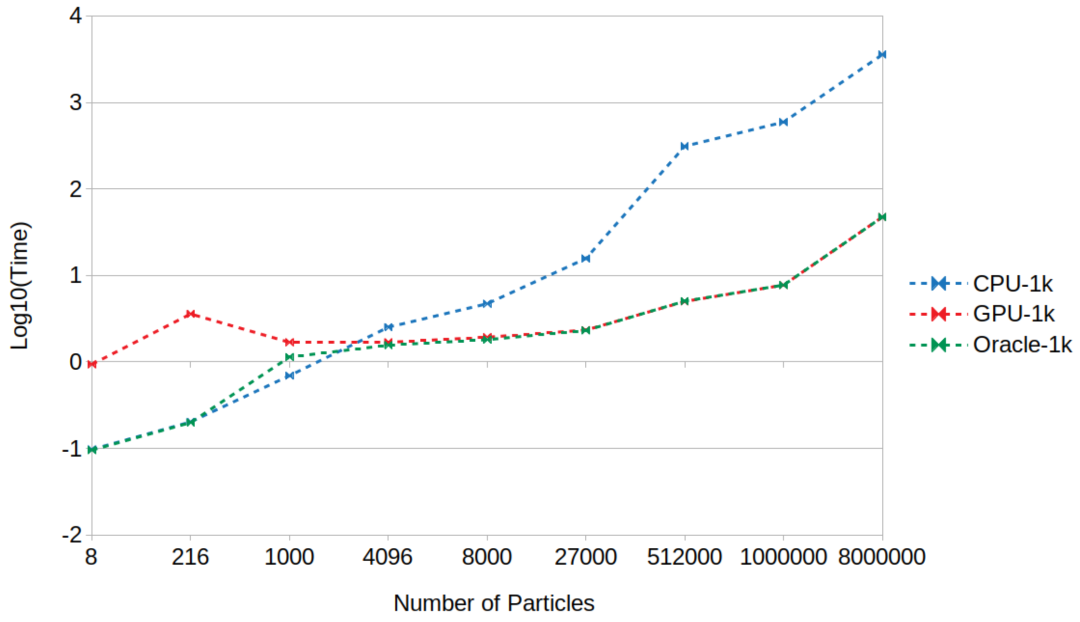


Figure 8. Execution times for the Fish Tank data set with 1000 steps.

Additionally, figures 7 and 8 show the performance of the Fish Tank data set with 10000 and 1000 steps, respectively. Just like the results for the Fusion data set, these results also show a trend where the CPU starts with better performance, but as the problem size increases, the GPU performs better and better. Again, the Oracle tries to mimic the performance of the faster single-device implementation, starting with targeting the CPU and eventually targeting the GPU more and more as the problem size increases.

In both sets of experiments, there is a *sweet-spot* range where the Oracle clearly does better than both. This range will be discussed in more detail below.

## 5.2 Sweet-Spot Performance

There were a couple workloads that were particularly bad for the GPU hardware (namely, those with a small amount of particles, yet had to advect over many steps). With our POD approach, this means that the performance of the

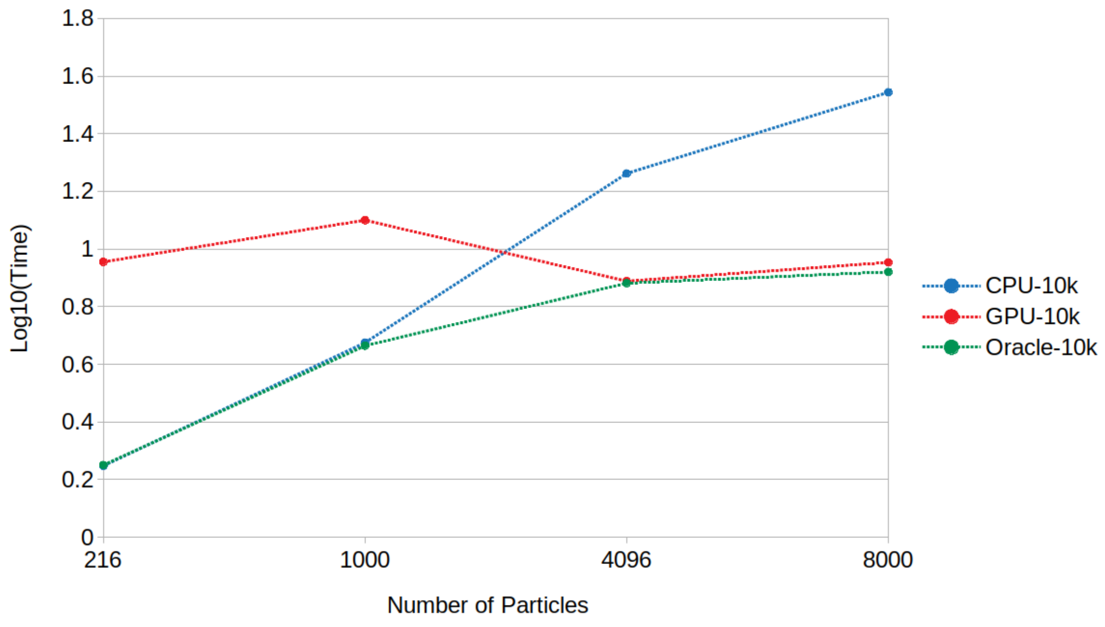


Figure 9. A zoomed-in chart showing the Fusion "sweet-spot" range with 10000 steps.

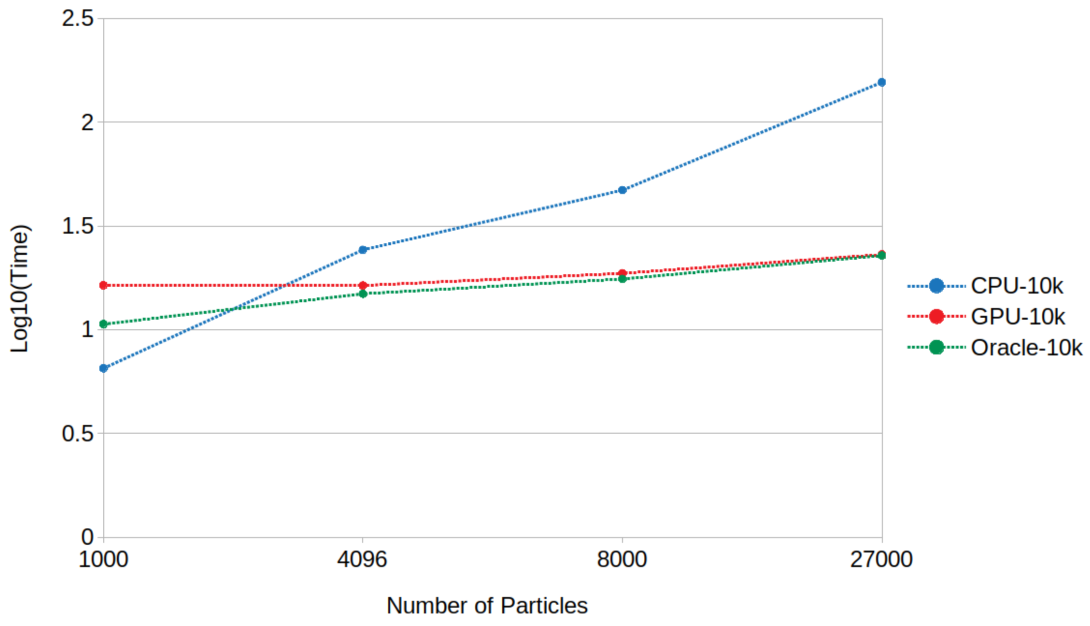


Figure 10. A zoomed-in chart showing the Fish Tank "sweet-spot" range with 10000 steps.

GPU threads will suffer due to latency. For these cases, the Oracle was able to determine that the CPU was better equipped to handle this kind of work, and targeted that device. The "*sweet-spot*" range refers to the subset of workloads where the Oracle consistently has better performance than either of the other implementations.

Since the Oracle can target the CPU to work on those particles that would have taken longer to be done on the GPU and can also target the GPU to work on those particles that would take longer to be done on the CPU, the Oracle's overall performance can beat both the GPU-only and the CPU-only performance. Figures 9 and 10 show these specific workloads at a close-up view. In figure 9, the "sweet-spot" range is seen mostly after 216 particles, but before 8,000 particles for the Fusion data set. However, for the Fisk Tank data set, the "sweet-spot" range is seen mostly after 1000 particles, but before 27,000 particles (shown in Figure 10). The difference in the "sweet-spot" ranges is a symptom of the data-dependent nature of particle advection.

Our results demonstrate that our Oracle was overall successful in deciding which device to run the current workload on. The most crucial criteria for our Oracle is a tuned parameter for a *threshold* that describes whether or not the workload could fully load the GPU or not. In most experiments, our Oracle was able to determine when the workload was better suited for the GPU or if it could instead benefit more from a CPU. This result leads to the conclusion that although the Oracle algorithm had some added overhead, it was not significant enough to affect overall performance.

# CHAPTER VI

# CONCLUSION

Our study shows the results of a new algorithm that targets the appropriate device to perform particle advection operations. By analyzing the current work remaining and targeting the appropriate device, our Oracle can achieve similar or better performance over the traditional single-device alternatives. The results show that there is a “sweet-spot” for the Oracle where it can best outperform the other devices. For workloads that are not in this sweet spot range, the Oracle approximately matches the performance of the device that is best suited for the workload. Thus, our study shows the results of a new algorithm that targets the appropriate device to perform particle advection operations, which in turn answers our research questions.

In most of these cases, the Oracle gets a small speedup, but in a few cases the Oracle is outperformed by the other implementation. The results suggest that the Oracle gets better as the numbers of steps increases. When there are small numbers of steps, the Oracle was outperformed the most, although overall, it performed best. Most importantly, the overall trend seen across the step counts is that *first, the Oracle is matching the CPU performance for the small workload sizes. Next, there is a “sweet spot” range where the Oracle is actually getting better performance than both the GPU alone and the CPU alone. Then, the Oracle starts matching the GPU performance for the larger workload sizes.* In other words, our Oracle targets the device which will perform better given the current work to do and the work remaining. Our other findings include:

- With relatively little added effort, a hybrid VTK-m implementation can be created that will determine, during the execution of the program, which device to use for the current work remaining.

- VTK-m has the functionality to target multiple devices within the same program, a capability that is very new and still needs more testing.

The results of this study will help inform the scientific visualization community about those architectures that give optimal performance at certain stages of the simulation run. The findings described in this paper highlight the performance benefits of using an Oracle approach to target the appropriate device. Although, these insights are specifically for the particle advection operation, they could be extrapolated to other algorithms as well.

For future work of this study, solving the limitation mentioned earlier is top priority. Having an implementation that uses as much parallelization within VTK-m as possible (i.e., using OpenMP for the CPU parallelization, CUDA for GPU parallelization) is optimal to get a more in-depth analysis of the Oracle. Additionally, it would be beneficial to see how this idea could be extended to other scientific visualization algorithms. For example, how will the performance change for other algorithms with an Oracle implementation? What about algorithms which have more irregular control flows or memory access patterns? There are several interesting research questions that can be explored by extending this idea to other kinds of problems. All in all, we feel this study is a good starting point for more evaluation and performance analysis of device targeting and the Oracle approach. Further, the impact of this topic can be high, since modern supercomputers often have both architectures. Finally, another topic of future work is to use both CPU and GPU simultaneously. This direction may be more effort-intensive, i.e., require significantly more complex algorithmic approaches.

## REFERENCES CITED

[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. W. Scogland. RAJA: Portable performance for large-scale scientific applications. *IEEE/ACM International Workshop on Performance, Portability, and Productivity in HPC*, 2019.

[2] Nathan Bell and Jared Hoberock. Thrust: Productivity-oriented library for cuda. *Astrophysics Source Code Library*, 7:12014–, 12 2012.

[3] Roba Binyahib, David Pugmire, Boyana Norris, and Hank 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.

[4] Guy Blelloch. In *Vector Models for Data-Parallel Computing*, volume 356. MIT Press, 1990.

[5] R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. volume 46, pages 720–748. ACM, Sept. 1999.

[6] David Camp, Hank Childs, Christoph Garth, David Pugmire, and Kenneth I. Joy. Parallel Stream Surface Computation for Large Data Sets. In *Proceedings of IEEE Symposium on Large Data Analysis and Visualization (LDAV)*, pages 39–47, Seattle, WA, October 2012.

[7] David Camp, Christoph Garth, Hank Childs, David Pugmire, and Kenneth I. Joy. Streamline Integration Using MPI-Hybrid Parallelism on a Large Multicore Architecture. *IEEE Transactions on Visualization and Computer Graphics (TVCG)*, 17(11):1702–1713, November 2011.

[8] David Camp, Hari Krishnan, David Pugmire, Christoph Garth, Ian Johnson, E. Wes Bethel, Kenneth I. Joy, and Hank Childs. GPU Acceleration of Particle Advection Workloads in a Parallel, Distributed Memory Setting. In *Proceedings of EuroGraphics Symposium on Parallel Graphics and Visualization (EGPGV)*, pages 1–8, Girona, Spain, May 2013.

[9] C. Garth D. Pugmire, T. Peterka. Parallel integral curves. In *In High Performance Visualization: Enabling Extreme-Scale Scientific Insight.* Chapman & Hall, CRC Press, 2012.

[10] H. Carter Edwards, Christian R. Trott, and Daniel Sunderland. Kokkos: Enabling manycore performance portability through polymorphic memory access patterns. *Journal of Parallel and Distributed Computing*, 74(12):3202 – 3216, 2014. Domain-Specific Languages and High-Level Frameworks for High-Performance Computing.

[11] Oak Ridge National Lab Leadership Computing Facility. Summit user guide. 2019.

[12] P. Fischer, J. Lottes, D. Pointer, and A. Siegel. Petascale Algorithms for Reactor Hydrodynamics. *Journal of Physics: Conference Series*, 125:15, 2008.

[13] Wesley Kendall, Jingyuan Wang, Melissa Allen, Tom Peterka, Jian Huang, and David Erickson. Simplified parallel domain traversal. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC 11, New York, NY, USA, 2011. Association for Computing Machinery.

[14] Kewei Lu, Han-Wei Shen, and Tom Peterka. Scalable computation of stream surfaces on large scale vector fields. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC 14, page 10081019. IEEE Press, 2014.

[15] Tony McLoughlin, Robert S. Laramee, Ronald Peikert, Frits H. Post, and Min Chen. Over two decades of integration-based, geometric flow visualization. *Computer Graphics Forum*, 29(6):1807–1829, 2010.

[16] Kenneth Moreland, Christopher Sewell, William Usher, Lita Lo, Jeremy Meredith, David Pugmire, James Kress, Hendrik Schroots, Kwan-Liu Ma, Hank Childs, Matthew Larsen, Chun-Ming Chen, Robert Maynard, and Berk Geveci. VTK-m: Accelerating the Visualization Toolkit for Massively Threaded Architectures. *IEEE Computer Graphics and Applications (CG&A)*, 36(3):48–58, May/June 2016.

[17] 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)*, pages 1–6, Oct 2013.

[18] Boonthanome Nouanesengsy, Teng-Yok Lee, Kewei Lu, Han-Wei Shen, and Tom Peterka. Parallel particle advection and ftle computation for time-varying flow fields. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 61:1–61:11, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press.

[19] David Pugmire, Hank Childs, Christoph Garth, Sean Ahern, and Gunther H. Weber. Scalable Computation of Streamlines on Very Large Datasets. In *Proceedings of the ACM/IEEE Conference on High Performance Computing (SC09)*, Portland, OR, November 2009.

[20] David Pugmire, Abhishek Yenpure, Mark Kim, James Kress, Robert Maynard, Hank Childs, and Bernd Hentschel. Performance-Portable Particle Advection with VTK-m. In *Eurographics Symposium on Parallel Graphics and Visualization (EGPGV)*, pages 45–55, Brno, Czech Republic, June 2018.

[21] W. J. Schroeder, K. M. Martin, and W. E. Lorensen. The design and implementation of an object-oriented toolkit for 3d graphics and visualization. In *Proceedings of Seventh Annual IEEE Visualization '96*, pages 93–100, Oct 1996.

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

[23] C.R. Sovinec, A. Glasser, T. Gianakon, Dan Barnes, R.A. Nebel, Scott Kruger, D. Schnack, S.J. Plimpton, A. Tarditi, and M.S. Chu. Nonlinear magnetohydrodynamics with high-order finite elements. *Journal of Computational Physics*, 195:355–386, 06 2005.

[24] S S Vazhkudai, B R de Supinski, A S Bland, A Geist, J Sexton, J Kahle, C J Zimmer, S Atchley, S H Oral, D E Maxwell, V G Vergara Larrea, A Bertsch, R Goldstone, W Joubert, C Chambreau, D Appelhans, R Blackmore, B Casses, G Chochia, G Davison, M A Ezell, E Gonsiorowski, L Grinberg, B Hanson, B Hartner, I Karlin, M L Leininger, D Leverman, C Marroquin, A Moody, M Ohmacht, R Pankajakshan, F Pizzano, J H Rogers, B Rosenburg, D Schmidt, M Shankar, F Wang, P Watson, B Walkup, L D Weems, and J Yin. The design, deployment, and evaluation of the coral pre-exascale systems. 7 2018.

[25] Hongfeng Yu, Chaoli Wang, and Kwan-Liu Ma. Parallel hierarchical visualization of large time-varying 3d vector fields. In *Proceedings of the 2007 ACM/IEEE Conference on Supercomputing*, SC '07, pages 24:1–24:12, New York, NY, USA, 2007. ACM.