Lawrence Berkeley National Laboratory
LBL Publications

Title

Extensions to the SENSEI In situ Framework for Heterogeneous Architectures

Permalink

https://escholarship.org/uc/item/02h3w8d7

Authors

Loring, Burlen

Bethel, E Wes

Weber, Gunther H

et al.

Publication Date

2023-10-04

eScholarship.org Powered by the California Digital Library
University of California

Extensions to the SENSEI *In situ* Framework for Heterogeneous Architectures

Burlen Loring
bloring@lbl.gov
Lawrence Berkeley Lab
Berkeley, CA, USA

Gunther H. Weber
ghweber@lbl.gov
Lawrence Berkeley Lab
Berkeley, CA, USA

E. Wes Bethel
ewbethel@sfsu.edu
San Francisco State University
San Francisco, CA, USA

Michael W. Mahoney
mmahoney@stat.berkeley.edu
Lawrence Berkeley Lab, ICSI, and UC Berkeley
Berkeley, CA, USA

arXiv:2310.02926v1 [cs.DC] 4 Oct 2023

ABSTRACT

The proliferation of GPUs and accelerators in recent supercomputing systems, so called heterogeneous architectures, has led to increased complexity in execution environments and programming models as well as to deeper memory hierarchies on these systems. In this work, we discuss challenges that arise in *in situ* code coupling on these heterogeneous architectures. In particular, we present data and execution model extensions to the SENSEI *in situ* framework that are targeted at the effective use of systems with heterogeneous architectures. We then use these new data and execution model extensions to investigate several *in situ* placement and execution configurations and to analyze the impact these choices have on overall performance.

CCS CONCEPTS

• **Theory of computation → Models of computation**; **Distributed computing models**;

KEYWORDS

parallel computing, scientific computing, in situ analytics

ACM Reference Format:
Burlen Loring, Gunther H. Weber, E. Wes Bethel, and Michael W. Mahoney. 2023. Extensions to the SENSEI *In situ* Framework for Heterogeneous Architectures. In *Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis (SC-W 2023), November 12–17, 2023, Denver, CO, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3624062.3624161

1 INTRODUCTION

The current generation of heterogeneous high performance supercomputing systems provides massive computing power through a mix of CPUs, GPUs, and other accelerators such as FPGAs and AI/ML specific devices. The use of such specialized accelerators is only expected to increase in the future [19]. Heterogeneity creates several challenges for programming and interoperability on these systems. This includes dealing with multiple execution environments running asynchronously with respect to each other, as well as managing memory and data movement on and between accelerators and the host. Much recent work has been devoted to platform portability, the means by which one may compile and run a single body of source code on multiple systems. This has led to a proliferation of programming models (PMs) from which to choose, including those provided by vendors, through compilers, language standards, as well as third party solutions. Examples of such options include OpenMP, Kokkos, Raja, HIP, SYCL, OpenCL, DPC++, std::par, and CUDA, among others. These efforts have made it possible for code teams to cope with the rapidly evolving hardware landscape. However, the proliferation of PMs has created additional interoperability challenges when coupling codes written by different teams. In particular, different code teams may, for any of a

```
// allocate device memory
omp_set_default_device(devId);
auto devPtr = (double*)omp_target_alloc(nElem*sizeof(double), devId);

// wrap it in a shared pointer so it is eventually deallocated
std::shared_ptr<double> spDev(devPtr,
  [devId](double *ptr){ omp_target_free(ptr, devId); });

// initialize the array on the device
#pragma omp target teams distribute parallel for is_device_ptr(devPtr)
for (size_t i = 0; i < nElem; ++i)
  devPtr[i] = -3.14;

// zero-copy construct with coordinated life cycle management
auto simData = svtkHAMRDoubleArray::New("simData", spDev, nElem, 1,
                              svtkAllocator::openmp, svtkStream(),
                              svtkStreamMode::async, devId);

// pass the array to SENSEI for processing

// free up the container
simData->Delete();
```

Listing 1: Packaging device data for zero-copy data transfer.

```
// this data is located in host memory, initialized to 1
auto a1 = svtkHAMRDoubleArray::New("a1", nElem, 1,
                          svtkAllocator::malloc, svtkStream(),
                          svtkStreamMode::async, 1.0);

// this data is located in device 1 memory, unitialized
omp_set_default_device(dev1);
auto a2 = svtkHAMRDoubleArray::New("a2", nElem, 1,
                          svtkAllocator::openmp, svtkStream(),
                          svtkStreamMode::async, 2.0);

// pass data to libA for the calculations on device 2
auto a3 = libA::Add(dev2, a1, a2);

// pass data to libB for I/O
auto ofile = std::ofstream("data.txt");
libB::Write(ofile, a1);
ofile.close();
```

Listing 2: Illustration of PM interoperability.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).

SC-W 2023, November 12–17, 2023, Denver, CO, USA

© 2023 Copyright held by the owner/author(s).

ACM ISBN 979-8-4007-0785-8/23/11.
https://doi.org/10.1145/3624062.3624161

```
svtkHAMRDoubleArray*
Add(int dev, svtkHAMRDoubleArray *a1, svtkHAMRDoubleArray *a2)
{
  // use this stream for the calculation
  cudaStream_t strm = svtkStream();

  // get a view of the incoming data on the device we will use
  cudaSetDevice(dev);

  auto spA1 = a1->GetCUDAAccessible();
  auto pA1 = spA1.get();

  auto spA2 = a2->GetCUDAAccessible();
  auto pA2 = spA2.get();

  // allocate space for the result
  size_t nElem = a1->GetNumberOfTuples();

  auto a3 = svtkHAMRDoubleArray::New("sum", nElem, 1,
                             svtkAllocator::cuda_async, strm,
                             svtkStreamMode::async);

  // direct access to the result since we know it is in place
  auto pA3 = a3->GetData();

  // make sure the data in flight, if it was moved, has arrived
  a1->Synchronize();
  a2->Synchronize();

  // do the calculation
  int threads = 128;
  int blocks = nElem / threads + ( nElem % threads ? 1 : 0 );
  add<<<blocks,threads,0,strm>>>(pA3, pA1, pA2, nElem);

  return a3;
}
```

Listing 3: A library function in *libA* that adds two arrays using the CUDA PM.

```
void Write(std::ofstream &ofs, svtkHAMRDoubleArray *a)
{
  // get a view of the data on the host
  auto spA = a->GetHostAccessible();
  auto pA = spA.get();

  // make sure the data if moved has arrived
  a->Synchronize();

  // send the data to the file
  size_t nElem = a->GetNumberOfTuples();
  for (size_t i = 0; i < nElem; ++i)
    ofs << pA[i] << " ";
}
```

Listing 4: A library function in *libB* that writes an array to disk.

number of valid reasons, choose different PMs, and they may target execution on different devices.

In terms of the data management challenges that arise when coupling independently developed codes, the challenges we increasingly face include efficiently and correctly sending data between codes that are potentially written in different PMs and/or potentially processing the data on a different device or the host. Solutions that make it possible for the codes being coupled to automatically interoperate without the need to know each others' internal capabilities and implementation details, e.g., without the need to know which PM is used and which hardware is targeted, are desirable.

Nowhere are these challenges more prevalent than in the SENSEI generic *in situ* data analysis and visualization framework [1, 2, 5, 12, 13]. SENSEI is a system that couples simulation codes to multiple back-end data processing, data transport, I/O libraries, and visualization tools through a single instrumentation; and it allows for run time switching between these back-ends. Due to the broad diversity of simulation and back-end codes supported by SENSEI, there is a need to mediate data exchanges between simulations, potentially written in one PM and executing on one accelerator, and back-end processing codes, potentially written in a different PM and potentially executing on a different accelerator, or the host. Because simulations are resource hungry codes, often making full use of the available memory and compute capacity, *in situ* solutions such as SENSEI must be as efficient as possible. This means data transfers between the simulation and back end data consumer are ideally made in place, or zero-copy, whenever they can be, in order to avoid the increased memory footprint and data movement overheads associated with making a deep copy.

In this work, we present extensions to SENSEI's data and execution model to solve the challenges of *in situ* code coupling on heterogeneous systems. Section 2 describes the data model extensions that address these issues and make possible correct, efficient, data transfers through a single API. Our work is based on new high-level data structures that provide PM interoperability as well as multi-device memory management. Our solution makes use of modern C++ features such as templates for efficiency and smart pointers for automated resource management.

Section 3 describes the execution model extensions that enable run time scheduling of *in situ* processing on different devices or the host as well as control over synchronous or asynchronous execution. Section 4 investigates different scheduling, execution, and placement options for the *in situ* code, made possible by our extensions, in the context of a simulation programmed with OpenMP target offload and an *in situ* analysis programmed in CUDA. The investigation is designed to answer the question: given a fixed number of compute nodes, each with multiple accelerators and CPU cores, what is the most effective way to utilize the available resources for *in situ* processing? For instance, if a simulation makes heavy use of the GPUs, while CPU cores are under utilized, could overall run time be reduced by moving *in situ* processing to the host? Similarly, when both simulation and *in situ* make heavy use of the GPU could overall run time be reduced by moving *in situ* processing to a dedicated GPU? In Section 4.4 we analyze the impacts on the overall time to solution as well as simulation solver update times and *in situ* processing times. In Section 5 we wrap up and discuss potential future directions.

Background and Related Work. While many *in situ* frameworks already support the use of accelerators and GPUs (including SENSEI, ParaView Catalyst [3], VisIt Libsim [20], and Ascent [9]), limitations in the data models of these tools preclude or limit the possibility for zero-copy data transfers of data located on the accelerator between the simulation and *in situ* library to very specific scenarios. Recent work on the Conduit *in situ* data model, which is used by Catalyst and Ascent, reported zero-copy data transfer capabilities for complex mesh based data structures, with no mention or explanation of whether or how multi-accelerator heterogeneous systems are supported [6]. While existing data models could potentially directly access accelerator memory through technologies like CUDA's unified memory, not all simulation codes use these mechanisms, and not all accelerator hardware supports them. Furthermore, explicit synchronization would be required to ensure

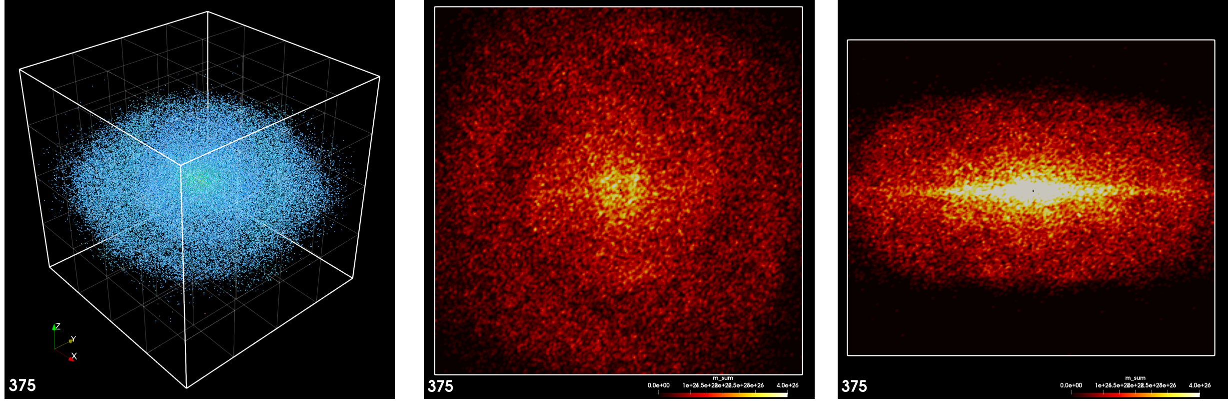

Figure 1: Left: N-body simulation of 100k bodies initialized from uniform random distributions in position, mass, and velocity with a massive body at the origin, run on 64 GPUs at step 187500. The runs reported in Section 4.3 used the same initialization method with 24M bodies and ran on 512 GPUs. Middle: *In situ* data binning in the x-y plane of the sum of mass on a 256×256 grid of the same time step. Right: *In situ* data binning in the x-z plane of the sum of mass on a 256×256 grid of the same time step.

correct behavior. None of the *in situ* data models we examined handle PM interoperability or synchronization issues directly; and some rely on PM specific functionality that is not uniformly available across all PMs.

In addition to the data and execution model extensions we made to SENSEI to support heterogeneous systems, our work investigates a number of execution configurations, some of which require on node inter-accelerator or accelerator-host data movement. A number of previous works have compared different strategies that moved data off node through inter-process communication for processing [4, 8, 14]. The data binning analysis technique we used to explore our data and execution model extensions has previously been successfully used for *in situ* data reduction [7, 17].

VTKm is a platform portable visualization library that can run on accelerators [16]. While VTKm is widely used internally for its accelerated visualization capabilities, none of the *in situ* libraries we examined use the VTKm data model in their simulation facing instrumentation APIs.

2 DATA MODEL EXTENSIONS

In the SENSEI data model, which is based on VTK [18], the , an abstract base class, defines the interfaces for managing and accessing array based data. The data sets representing mesh geometry, associated node and cell centered data, as well as the associated uncentered data are built on top of and make use of `svtkDataArray`. However, the subclasses implementing the `svtkDataArray` APIs available in VTK are designed and implemented for host only memory management. In order to support heterogeneous architectures, we add a new `svtkDataArray` subclass called the `svtkHAMRDataArray` (`HDA`) to the SENSEI data model. The `HDA` provides both host and device memory management as well as PM interoperability. Internally, `HDA` makes use of the HAMR memory management library [10]. Below, we present some of the relevant APIs along with illustrative examples.

Initialization. Before a `HDA` instance can be used it must be initialized for a particular PM and allocation strategy. This typically is part of the construction process, but APIs exist to initialize a default constructed instance as well. During initialization a passed `svtkAllocator` enumeration value specifies which PM, and which specific method within the PM, is used to allocate and subsequently manage the memory. SENSEI currently supports OpenMP offload, CUDA, and HIP allocators as well as host only allocators using `malloc`, and `new`. The CUDA and HIP allocators come in synchronous and asynchronous variants, variants that allocate universally addressable memory, as well as variants for allocating page locked memory. When using asynchronous allocators, a `svtkStream` and `svtkStreamMode` must also be specified. `svtkStream` is a class that abstracts the differences between PM streams. It has automatic conversions to and from PM native streams such that these can be used interchangeably. The `svtkStream` is used for ordering operations and explicit synchronization. A `svtkStreamMode` enumeration value specifies a synchronization mode. In asynchronous mode, calls to the `HDA` API return immediately while the operation is in progress making it possible to overlap allocation, data movement, and computation. The user must add synchronization points as needed. In synchronous mode, all operations complete before the `HDA` API call returns. Memory is allocated on the currently active device. This provides a way to control on which device the data is located. Line 15 of Listing 1 shows an example of the initialization of a newly created `HDA` instance.

Platform Portability and Code Execution. SENSEI currently supports the CUDA, OpenMP offload, and HIP PMs. Our data model extensions are platform portable when a platform portable programming mode is used. The selection of a PM is left entirely to users. Some users will opt for a platform portable option and others will opt for a vendor specific option. Our strategy is to manage data using the selected PM and provide interoperability with all of the supported PMs so that data can be passed between any two codes, including those written in different PMs, and those targeting execution on different accelerators or the host. Listing 1 illustrates platform portability achieved through the use of OpenMP. An example illustrating PM interoperability is given below.

Zero-copy Data Transfer. In SENSEI the simulation should always prefer a zero-copy transfer. With zero-copy transfer, the simulation shares a pointer that gives the *in situ* code direct access

Num. Nodes	*In-Situ* Method	Ranks per node	Ranks Total	*In-Situ* Location
128	lock step	4	512	all on host
				on same device
		3	384	1 dedicated device
		2	256	2 dedicated devices
	asynchr.	4	512	all on host
				on same device
		3	384	1 dedicated device
		2	256	2 dedicated devices

Table 1: A summary of the runs made to investigate *in situ* placement. See Section 4.4 for details.

to the data. The back-end data consumer can then decide if an explicit deep copy is needed or not. When the back-end can access the data in place, no additional work is done, reducing memory footprint and computational overhead.

Zero-copy transfer was easy to achieve on CPU-only architectures, as only a pointer and length were needed. On heterogeneous architectures, additional information is needed, e.g., the location of the data, PM used to manage it, and possibly associated PM specific context and synchronization data structures. `HDA`'s zero-copy data transfer APIs take pointers to externally allocated host or device memory and capture the necessary additional information. This includes: a host or device pointer to the memory; the length of the array; an `svtkAllocator` enumeration value identifying the PM specific allocator; an integer that identifies on which device the memory currently resides; as well as a `svtkStream` and `svtkStreamMode` that are used to control ordering and synchronization of subsequent operations. Listing 1 shows how data allocated and initialized using OpenMP is zero-copy transferred into a `HDA` instance. In this example, a smart pointer is used to coordinate memory life cycle between the simulation and `HDA` instance. We also implemented APIs that can take raw pointers. In that case, it's up to the user to manage memory life cycle.

PM Interoperability and Location Agnostic Access. We implemented an API that provides location and PM agnostic read only data access. The purpose of the API is to make it possible to efficiently and safely pass data in between independently developed codes which potentially make use of different PMs and potentially process data on different devices. The code accessing the data specifies the location, on host or device, and if on device on which device, and in which PM the data will be accessed. If the data to be accessed is already accessible on the requested device in the requested PM, no additional work is done, and direct access is granted. However, if the managed data is not accessible on the requested device a temporary is allocated and the data is moved. Any additional work required for interoperability between PMs is handled here as well. A `std::shared_ptr` is returned from the access API so that if a temporary were used it will automatically be cleaned up when the `std::shared_ptr` goes out of scope.

Listing 2 illustrates PM and location agnostic data access. Two `HDA` instances are allocated and initialized, one on the host in line 2, and the other on device 1 using OpenMP offload in line 8. On line 13, arrays are then passed into library *libA*, which will perform an element wise addition on device 2 and return the result. *libA* is implemented with the CUDA PM and is shown in Listing 3. In *libA*, device 2 is made active on line 8 then lines 10 and 13 use the `HDA` access API to obtain a view of the data to operate in the CUDA PM. If any host-device or inter-device data movement, or PM interoperability transformations, are needed, these are handled automatically in the `HDA` access API invisibly to *libA*. In this way *libA* can operate on data in any location managed by any PM with the same code. On line 24 *libA* uses the more efficient direct access API to get raw pointer to memory for the result because the location and PM are known.

Back in Listing 2, the result of the calculation made in *libA* is returned on line 13. A file is created and *libA*'s result is passed to library *libB* for output to disk. The implementation of *libB* in host only C++ is shown in Listing 4. *libB* calls the `HDA` access API on line 4 to get a view of the data to operate on the host. Any host-device data movement is handled automatically and invisibly to *libB* if it is needed. This example demonstrates how our data model handles PM interoperability and data movement automatically.

3 EXECUTION MODEL EXTENSIONS

In this section we describe two additions to SENSEI's execution model we made in order to support execution on heterogeneous architectures. The first is for the specification of an execution method. The second implements a number of *placement* options that give run time control over on which accelerator or the host the *in situ* code runs. These new features are exposed to users through an API and SENSEI's run time XML configuration.

The new execution methods are: *lockstep* where the simulation and *in situ* code take turns; and *asynchronous* where the *in situ* code uses threading to execute concurrently with the simulation. In terms of control over *in situ* placement, we implemented means for both manual explicit device selection and automatic device selection. Automatic device selection uses a number of run time provided control parameters along with the process's MPI rank and the number of on node devices to select a device to execute on according to the following rule:

$$d = (r \bmod n_u * s + d_0) \bmod n_a, \tag{1}$$

where: d is the assigned device; r is the MPI rank of the process making the query; n_u is the number of devices to use per node; s is the stride, d_0 is the offset, and n_a is the total number of device available on the node. r and n_a are initialized from system queries, while n_u, s, and d_0 can optionally be specified by the user. By default, $n_u = n_a$, $s = 1$, and $d_0 = 0$. The new control parameters and API are defined in the base class for SENSEI analysis back-ends and therefor available to all back-ends.

4 EVALUATION OF THE EXTENSIONS

In this section, we report a set of empirical results in which we seek to answer the question: "How can we best make use of multiple on node accelerators and CPUs for *in situ* processing?" We also demonstrate PM interoperability between a simulation written for OpenMP offload and an *in situ* analysis code written in CUDA, coupled through the SENSEI generic *in situ* framework.

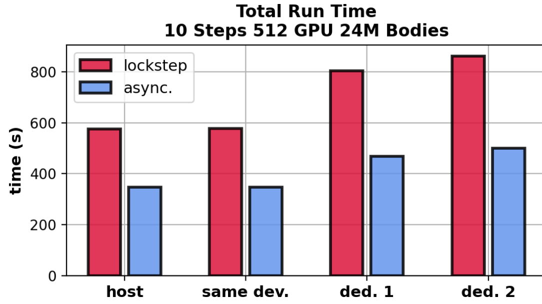

Figure 2: Total run time for lockstep(red) and asynchronous(blue) *in situ* for each of four *in situ* placements.

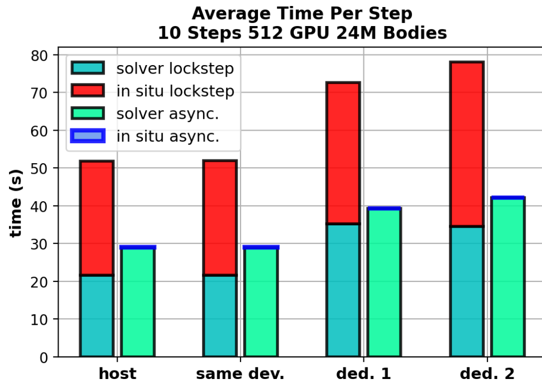

Figure 3: The average time per iteration of the solver and *in situ* processing, for each of the four *in situ* placements. Asynchronous *in situ* execution times are blue, while lockstep *in situ* times are red. Solver times are cyan.

4.1 Simulation

In our experiments we used the Newton++ simulation code [11]. Newton++ is an open source direct n-body simulation with a second order, time reversible, symplectic integration scheme. Newton++ is written in C++ and parallelized with MPI and OpenMP device offload. Each MPI rank owns a unique spatial subdomain of the simulated volume and is responsible for integrating bodies within its subdomain. As bodies evolve in time, a repartitioning phase migrates bodies that have moved out side of a given subdomain to the correct MPI rank. Newton++ is instrumented with SENSEI, and it has a VTK compatible output format for post processing and visualization. Newton++ also supports initial conditions generated by MAGI, the Many-component Galaxy Initializer [15], as well as initialization from uniform random distributions in position, mass, and velocity. The leftmost panel in Figure 1 shows example of the data generated by Newton++.

4.2 *In Situ* Data Binning

Given tabular data where columns represent different variables and rows represent co-occurring measurements or realizations of these variables, data binning specifies a subset of the variables to use as the coordinate axes of a uniform Cartesian mesh and transforms the data into the new coordinate system. For each realization, the values of the coordinate variables locate the mesh cell, or bin, to which the realization belongs. The low and high bounds of the mesh axes can be manually specified or obtained on the fly by calculating the minimum and maximum of the respective coordinate variables. Incrementing a per-mesh-cell counter creates a histogram of the data in terms of the chosen coordinates. Additional reduction operations are used to incorporate, or bin, non-coordinate variables into the result. The reduction operations we support are summation, minimum, maximum, and average.

For example, the first panel in Figure 1 shows data produced in a 100k body n-body simulation; the middle panels shows the result of data binning body mass with summation onto a 256×256 mesh with x and y body positions used as coordinate axes; and the right panel shows the same with body x and z positions used as coordinate axes. These examples show the use of the bodies' spatial coordinates as binning axes. However, it is common to use other per-body attributes, such as momentum or velocity, as the coordinate axes.

Our implementation is parallelized with MPI and CUDA. We provide a CPU implementation that runs on the host as well as a CUDA implementation that runs on an assigned device. Both implementations can run asynchronously in a C++ thread. We make use of the SENSEI execution model extensions described in Section 3 for placement and execution method control. The extensions to the data model described in Section 2 are used for data access and management.

4.3 Empirical Evaluation

In our empirical evaluation, we run the n-body simulation on a fixed number of GPUs on NERSC's supercomputer Perlmutter while changing how resources are allocated between simulation and *in situ* processing. We investigate four *in situ* placements in conjunction with two *in situ* execution methods for a total of eight cases.

The four *in situ* placements are: 1. all on host; 2. on the same device; 3. on a dedicated device; 4. on two dedicated devices. For all four *in situ* placements each simulation rank is assigned a specific GPU, there is always only 1 simulation rank per GPU. For the *host* placement, *in situ* calculations are scheduled on the host. Data is moved from the GPU on which it is generated to the system's main memory banks for *in situ* processing. With the *same device* placement, *in situ* calculations are scheduled on the device where it is generated. Four MPI ranks per-node, one per GPU, are used with the *host* and *same device* placements. With the *dedicated device* placement, one GPU per node is reserved for *in situ* processing. The three remaining GPUs per node are used exclusively by the simulation. In this placement three MPI ranks are used per node. Data is moved from the three simulation GPUs per node to the one *in situ* GPU per node for processing. With the *2 dedicated devices* placement each MPI rank has one GPU dedicated to simulation and one GPU dedicated to *in situ* processing. Data is moved from the simulation to its paired GPU for processing. In this placement two MPI ranks per node are used.

We investigated two execution methods: *lockstep*, and *asynchronous*. With *lockstep* execution the simulation and *in situ* processing take turns with the simulation waiting for the *in situ* to fully complete before proceeding. The *lockstep* method makes zero-copy copy data access possible in some of the cases. With *asynchronous* execution, the *in situ* analysis code runs in a separate thread, asynchronously with respect to the simulation. The *in situ* code deep

copies the relevant data, launches a thread for *in situ* processing, and returns immediately to the simulation. The simulation and *in situ* processing then proceed concurrently.

All runs were made on 128 Perlmutter nodes, using 512 GPUs. The various *in situ* placements allocated GPUs differently between *in situ* and simulation. The runs exercising the 8 cases described above are summarized in table 1. In all runs, Newton++ was configured with 24M bodies generated from the uniform random initial condition. *In situ* processing via SENSEI was performed at every iteration. I/O for *post hoc* visualization and body repartitioning were disabled during the runs. During *in situ* processing the data binning operator was applied to 10 variables over 9 coordinate systems for a total of 90 binning operations. Binning of each coordinate systems was done sequentially in a separate data binning operator instance and orchestrated by SENSEI using its XML configuration feature. The SLURM batch scripts and XML configs used in the experiments are provided in Appendix A.

4.4 Results and Discussion

Figures 2 and 3 summarize the data gathered in the eight experimental runs. Figure 2 shows the total run time including initialization, solver steps, *in situ* processing, and finalization. Figure 3 shows the average time per iteration spent in the solver and *in situ* processing in a stack plot. The total height is the average time spent per iteration. Both plots organize the data into four groups, one for each *in situ* placement. Within each of the four groups, one bar represents the *lockstep* execution method and the other *asynchronous* execution method.

These results show that, across all placements, executing *in situ* asynchronously is beneficial and reduced the total run time. The apparent time spent in *in situ* processing when *asynchronous* execution was used was very small, less than 10ms across all time steps and all placements. This makes it look like *in situ* is effectively free when executing asynchronously. However, comparing the solver time between the *lockstep* and *asynchronous* cases in Figure 3 it is clear that the solver was slowed down across all placements when the *in situ* was executed asynchronously. None the less running the *in situ* asynchronously significantly reduced the total run time.

The placements assigning one or two dedicated devices for *in situ* processing made use of a reduced total number of MPI ranks, as well as a reduced total number of GPUs used by the simulation and *in situ* respectively. The reduced levels of concurrency led to longer run times. For instance the *two dedicated devices* placement used only half of the MPI ranks that were used in the *same device* placement, and individually both simulation and *in situ* had access to half the number of GPUs. There was a negligible difference between the *host only* and *same device* placements. We observe that data binning is not an ideal algorithm for GPUs since it requires the use of atomic memory updates to deal with races between GPU threads accessing the same bin.

5 CONCLUSIONS AND FUTURE WORK

We have presented data and execution model extensions to the SENSEI *in situ* framework to automate data management and PM interoperability on heterogeneous architectures. We demonstrated the PM interoperability, inter-device memory management, and on node *in situ* placement capabilities provided by our extensions in a set of empirical evaluations that investigated different *in situ* placement and execution configurations on NERSC's Cray Perlmutter.

The asynchronous execution method we implemented resulted in overall reduced run time in spite of slowing the solver down relative to the lockstep method. In future work we plan to do deeper profiling to understand this better as well as more profiling to better understand the opportunities for improving performance when assigning one or two dedicated devices for *in situ* processing. We will profile and optimize the data binning implementation to achieve a speed up on the GPU relative to the CPU. We will also add support for SYCL as well as third party PMs such as Kokkos.

ACKNOWLEDGMENTS

This material is based upon work supported by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research under Award Number DE-AC02-05CH11231. This research used resources of the National Energy Research Scientific Computing Center (NERSC), a U.S. Department of Energy Office of Science User Facility.

REFERENCES

[1] Utkarsh Ayachit et al. 2016. Performance Analysis, Design Considerations, and Applications of Extreme-Scale in Situ Infrastructures. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Salt Lake City, Utah) *(SC '16)*. IEEE Press, Article 79, 12 pages.

[2] Utkarsh Ayachit et al. 2016. The SENSEI Generic In Situ Interface. In *Proceedings of In Situ Infrastructures for Enabling Extreme-scale Analysis and Visualization (ISAV 2016)*. Salt Lake City, UT, USA.

[3] Andrew C. Bauer et al. 2015. *The ParaView Catalyst User's Guide v2.0.* Kitware, Inc.

[4] Valentin Bruder et al. 2023. A Hybrid in Situ Approach for Cost Efficient Image Database Generation. *IEEE Transactions on Visualization and Computer Graphics* 29, 9 (2023), 3788–3798. https://doi.org/10.1109/TVCG.2022.3169590

[5] Junmin Gu et al. 2019. HDF5 as a Vehicle for in Transit Data Movement. In *Proceedings of the Workshop on In Situ Infrastructures for Enabling Extreme-Scale Analysis and Visualization* (Denver, Colorado, USA) *(ISAV '19)*. Association for Computing Machinery, New York, NY, USA, 39–43. https://doi.org/10.1145/3364228.3364237

[6] Cyrus Harrison et al. 2022. Conduit: A Successful Strategy for Describing and Sharing Data In Situ. In *2022 IEEE/ACM International Workshop on In Situ Infrastructures for Enabling Extreme-Scale Analysis and Visualization (ISAV)*. 1–6. https://doi.org/10.1109/ISAV56555.2022.00006

[7] James Kress et al. 2018. Binning Based Data Reduction for Vector Field Data of a Particle-In-Cell Fusion Simulation. In *High Performance Computing*. Springer International Publishing, Cham, 215–229.

[8] James Kress et al. 2019. Comparing the Efficiency of In Situ Visualization Paradigms at Scale. In *High Performance Computing*. Springer International Publishing, Cham, 99–117.

[9] Matthew Larsen et al. 2022. Ascent: A Flyweight In Situ Library for Exascale Simulations. In *In Situ Visualization For Computational Science*. Mathematics and Visualization book series from Springer Publishing, Cham, Switzerland, 255 – 279.

[10] Burlen Loring. 2022. *HAMR the Heterogeneous Accelerator Memory Resource.* https://doi.org/10.5281/zenodo.8394163

[11] Burlen Loring. 2023. *Newton++ An MPI+OpenMP offlaod parallel n-body code written in C++.* https://doi.org/10.5281/zenodo.8394150

[12] Burlen Loring et al. 2018. Python-Based in Situ Analysis and Visualization. In *Proceedings of the Workshop on In Situ Infrastructures for Enabling Extreme-Scale Analysis and Visualization* (Dallas, Texas, USA) *(ISAV '18)*. Association for Computing Machinery, New York, NY, USA, 19–24. https://doi.org/10.1145/3281464.3281465

[13] Burlen Loring et al. 2020. Improving Performance of M-to-N Processing and Data Redistribution in In Transit Analysis and Visualization. In *20th Eurographics Symposium on Parallel Graphics and Visualization*, Steffen Frey, Jian Huang, and Filip Sadlo (Eds.). Eurographics Association, 35–45. https://doi.org/10.2312/pgv.20201073

[14] Preeti Malakar et al. 2015. Optimal scheduling of in-situ analysis for large-scale scientific simulations. In *SC '15: Proceedings of the International Conference for*

High Performance Computing, Networking, Storage and Analysis. 1–11. https://doi.org/10.1145/2807591.2807656
[15] Yohei Miki et al. 2018. MAGI: many-component galaxy initializer. *Monthly Notices of the Royal Astronomical Society* 475, 2 (01 2018), 2269–2281. https://doi.org/10.1093/mnras/stx3327
[16] K. Moreland, C. Sewell, W. Usher, L. t. Lo, J. Meredith, D. Pugmire, J. Kress, H. Schroots, K. L. Ma, H. Childs, M. Larsen, C. M. Chen, R. Maynard, and B. Geveci. 2016. VTK-m: Accelerating the Visualization Toolkit for Massively Threaded Architectures. *IEEE Computer Graphics and Applications* (2016).
[17] Oliver Rübel et al. 2016. WarpIV: In Situ Visualization and Analysis of Ion Accelerator Simulations. *IEEE Computer Graphics and Applications* 36, 3 (2016), 22–35. https://doi.org/10.1109/MCG.2016.62
[18] Will Schroeder et al. 2006. *The Visualization Toolkit (4th ed.).* Kitware.
[19] Jeffrey S. Vetter et al. 2018. Extreme Heterogeneity 2018 - Productive Computational Science in the Era of Extreme Heterogeneity: Report for DOE ASCR Workshop on Extreme Heterogeneity. (12 2018). https://doi.org/10.2172/1473756
[20] Brad Whitlock et al. 2011. Parallel in Situ Coupling of Simulation with a Fully Featured Visualization System. In *Proceedings of the 11th Eurographics Conference on Parallel Graphics and Visualization* (Llandudno, UK) *(EGPGV '11)*. 101–109.

A REPRODUCIBILITY

We have organized all of the information needed to reproduce our experiments in a repository on github. https://github.com/SENSEI-insitu/ISAV_2023 The information is organized as follows:

Directory	**Description**
software	documents the versions of software used
environment	documents the requisite Perlmutter modules
build	Documents the configure and build steps
sensei_xml	SENSEI XML configurations used in the runs
run	SLURM batch scripts used in the runs
data	data gathered in the runs
analysis	scripts used to analyze the runs

Received 04 August 2023; revised XX; accepted XX