Los Alamos
NATIONAL LABORATORY
EST. 1943

LA-UR-20-28242
Approved for public release; distribution is unlimited.

Title: Parallelization and Performance Portability in Hydrodynamics Codes

Author(s): Dunning, Daniel Jeffrey

Intended for: Dissertation Defense

Issued: 2020-10-14

Disclaimer:
Los Alamos National Laboratory, an affirmative action/equal opportunity employer, is operated by Triad National Security, LLC for the National Nuclear Security Administration of U.S. Department of Energy under contract 89233218CNA000001. By approving this article, the publisher recognizes that the U.S. Government retains nonexclusive, royalty-free license to publish or reproduce the published form of this contribution, or to allow others to do so, for U.S. Government purposes. Los Alamos National Laboratory requests that the publisher identify this article as work performed under the auspices of the U.S. Department of Energy. Los Alamos National Laboratory strongly supports academic freedom and a researcher's right to publish; as an institution, however, the Laboratory does not endorse the viewpoint of a publication or guarantee its technical correctness.

Parallelization and Performance Portability in Hydrodynamics Codes [1]

Daniel Dunning

Texas Tech College of Engineering, Computer Science Department

October 15, 2020

[1] LA-UR-

Table of Contents

Table of Contents

CLAMR

- Mini-app intended for research in cell-based mesh methods
- Served as a testbed for efficient memory structures, space filling curves, and hashing
- Built using the shallow water equations
- Written in C++ with OpenMP, MPI, and OpenCL (GPU) implementations

Table of Contents

Motivation

- AMR is challenging
- Physicist must be aware of the dynamically changing mesh when writing physics routines
- Code development is harder: architecture portage, debugging, optimization, parallelization, etc.

AMR Hybrid

- **Cell-based AMR**
- Pros
  - More efficient (fewer cells)
  - Easier load balancing
  - Less mesh imprinting
- Cons
  - Data is unstructured (Mesh is structured, but data can be in any order relative to the mesh)

- **Patch-based AMR**
- Pros
  - Patches are a regular grid
  - More common, so more research devoted[1]
- Cons
  - Refinement clustering is a non-local operation making it difficult to put on GPUs

[1] *Berger an LeVeque: Adaptive mesh refinement using wave-propagation algorithms for hyperbolic systems*

Phantom-Cell AMR: A Hybrid Approach

- Use the best of both current approaches
- Use a cell-based mesh with the appearance of a regular grid
- Fastlane to Exascale computing
  - Cell-based operations are local
  - Allows methods that can be written for GPU architectures
- Expands the possibilities for both old and new applications

Phantom-Cell AMR (cont.)

- **Goal: Separate physics and mesh codes**
  - Computational scientists can focus on the physics
  - Mesh handling maintained by different developers
- **Deployable, maintainable, optimizable AMR**
- Regular physics on an irregular grid
- Guarantee: Every cell has two neighbors in each direction at the same level of refinement

Cell creation

- Phantom cells are created at all refinement boundaries
- In the same way ghost cells communicate between processors, phantom cells communicate between refinement levels

Single Level Representation

Picture

Mass Conservation and Interpolation

- Uses technique borrowed from Berger and Leveque's work on patch-based AMR that takes the finer cell flux and applies it to the coarser cell
- Mass conservation is not achieved without this "correction"
- Currently looking for alternative schemes

Flux Correction

Finer cells 21 and 22 will pass their positive x Flux values (red) to their coarse neighbor, cell 20, who will use the aggregate of those fluxes as its negative x Flux value. PICTURE

Mesh Data Structures

- Different methods (and accompanying data structures) are provided as a result of phantom cells
- **In-place methods**
  - Phantom cells are added to existing cell list hash maps (single dimension)
  - Refinement is unaffected, state routines incorporate same-level neighbors
- **Regular grid methods**
  - Separate grid into separate sub-grids (more memory)
  - Refinement still unaffected, state routines may be changed to incorporate spatial indexing (i, j, k)

Access pattern

Picture

Mesh Overview

3 level picture

Table of Contents

Methodology

platform – a particular execution environment (i.e. hardware an operating system, some compilation and runtime tools)

application – any suite of software that can accept a given problem as input and produce an output that can be validated against some existing measure of correctness [2]

[2] Pennycook, A Metric for Performance Portability

Methodology (cont.)

Performance – Any measurable property of an application's correct execution of a problem on a platform.

Portability – The ability of an application to execute a problem correctly on a given set of platforms.

Performance Portability – A measurement of an application's performance efficiency for a given problem that can be executed correctly on all platforms in a given set. [3]

[3] Pennycook, A Metric for Performance Portability

Application Efficiency

application efficiency – achieved performance relative to the best implementation on that platform

- Measures how well the different performance-based implementations perform across the platforms
- Look at different (parallel) implementations
  - Normally only feasible for small applications

Architectural Efficiency

architectural efficiency – achieved performance relative to the peak theoretical performance of the hardware

- Measures how well the application performs on the hardware
- Use hardware counters to compare different characteristics
  - Bandwidth
  - FLOPs
  - Energy consumption

Method Bandwidth

Architecture

Method	Intel	AMD	ARM	IBM
Cell	6521	6923	1933	2760
Face	6521	6923	1933	2760
Cell in-place	4571	4900	1730	2891
Face in-place	7318	6733	2028	3300
Reggrid Cell	5848	4238	1667	3948
Reggrid Face	2879	1597	706	1668

Bandwidth in MBytes / sec for each method.

Method Efficiency

method efficiency – achieved performance of a method relative to the best implementation of that method

- Compares implementations of a function
- Like with architectural efficiency, we need a metric
  - Difficult because of different characteristics each method presents
- Helps look at which methods are good targets for optimization

Method runtimes

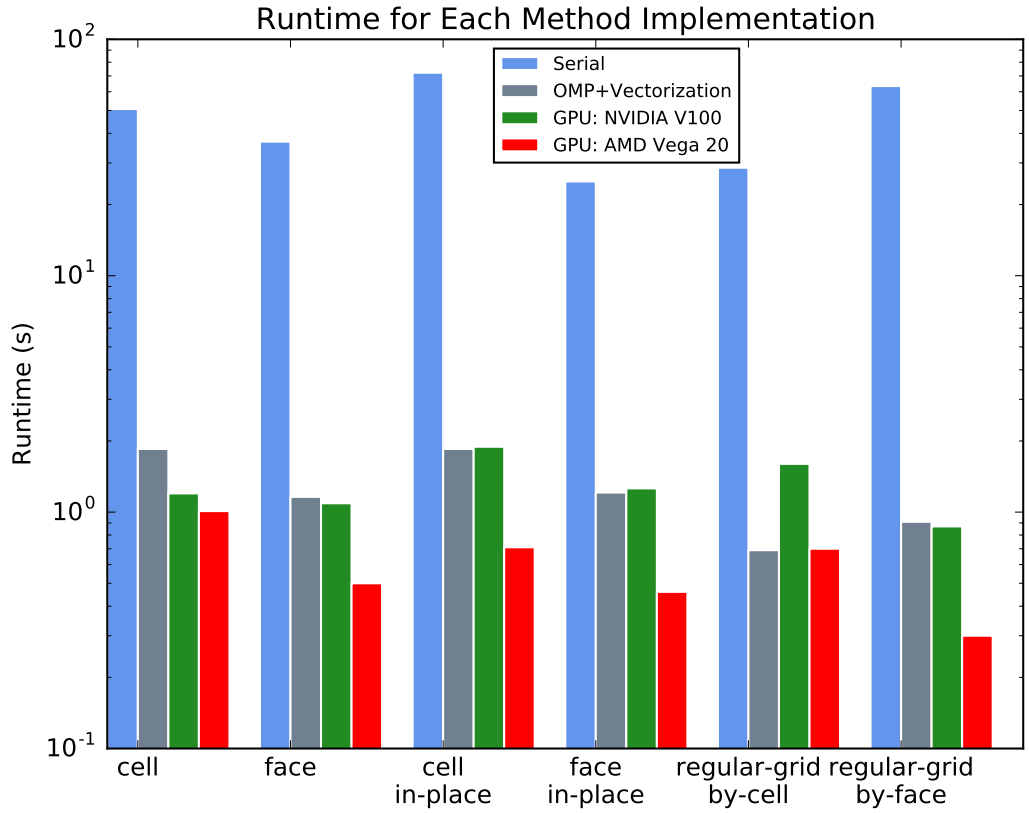


Figure: **Run-times for all method implementations.**

Method Similarity

	cell	cell in-place	face	face in-place	regular-grid	regular-grid-by-faces
cell	0.00	27.48	18.35	16.34	27.79	22.99
cell in-place	27.48	0.00	22.17	17.82	1.51	37.24
face	18.35	22.17	0.00	5.05	23.39	15.14
face in-place	16.34	17.82	5.05	0.00	18.92	19.51
regular-grid	27.79	1.51	23.39	18.92	0.00	38.40
regular-grid-by-faces	22.99	37.24	15.14	19.51	38.40	0.00

Figure: **Cosine similarity of the methods on the GPU, in terms of degrees.**

Metric Abundance

Run	Time	V/S ALU Inst	Scalar ALU/fetch Inst	Vector ALU/Mem Inst	Vector ALU Utilization	V/S ALU Busy	FetchSize	WriteSize	L1CacheHit	L2CacheHit	MemUnit Busy	MemUnit Stalled	WriteUnit Stalled	LDS Bank Conflict
cell	1.48	6.64	37.45	32.71	95.59	5.44	23417	12455	67.12	43.22	15.01	0.21	0.01	2.45
face	0.80	12.20	2.86	9.25	98.13	8.35	59837	28164	72.14	36.86	64.32	6.05	0.15	0.00
cell in-place	1.22	48.33	1.41	13.29	99.32	30.27	54097	59704	62.90	29.19	57.41	18.58	0.29	0.00
face in-place	0.74	15.01	3.20	10.36	99.37	10.25	49989	28892	70.60	32.63	60.59	12.83	0.14	0.00
regular-grid	1.16	27.35	2.80	8.88	98.54	19.18	49538	57355	71.09	39.88	75.43	24.46	0.29	0.00
regular-grid-by-faces	0.42	13.02	1.58	4.80	97.19	2.65	55312	9455	66.19	24.13	75.96	29.91	0.01	0.00

Optimization Efficiency

optimization efficiency – achieved performance of an implementation relative to the implementation effort

- Measure how "worth it" an optimization is
- Simple to conceptualize, difficult to quantify

Developer Effort

- Cannot improve all aspects of a code
- Developer must decide what improvements will be the most beneficial
- What can we measure
  - Coding time (including future efforts)
  - Developer cost
  - Readability

Examples

$$\text{effort [person} - \text{months]} = A \cdot M \cdot (\text{SIZE})^F$$

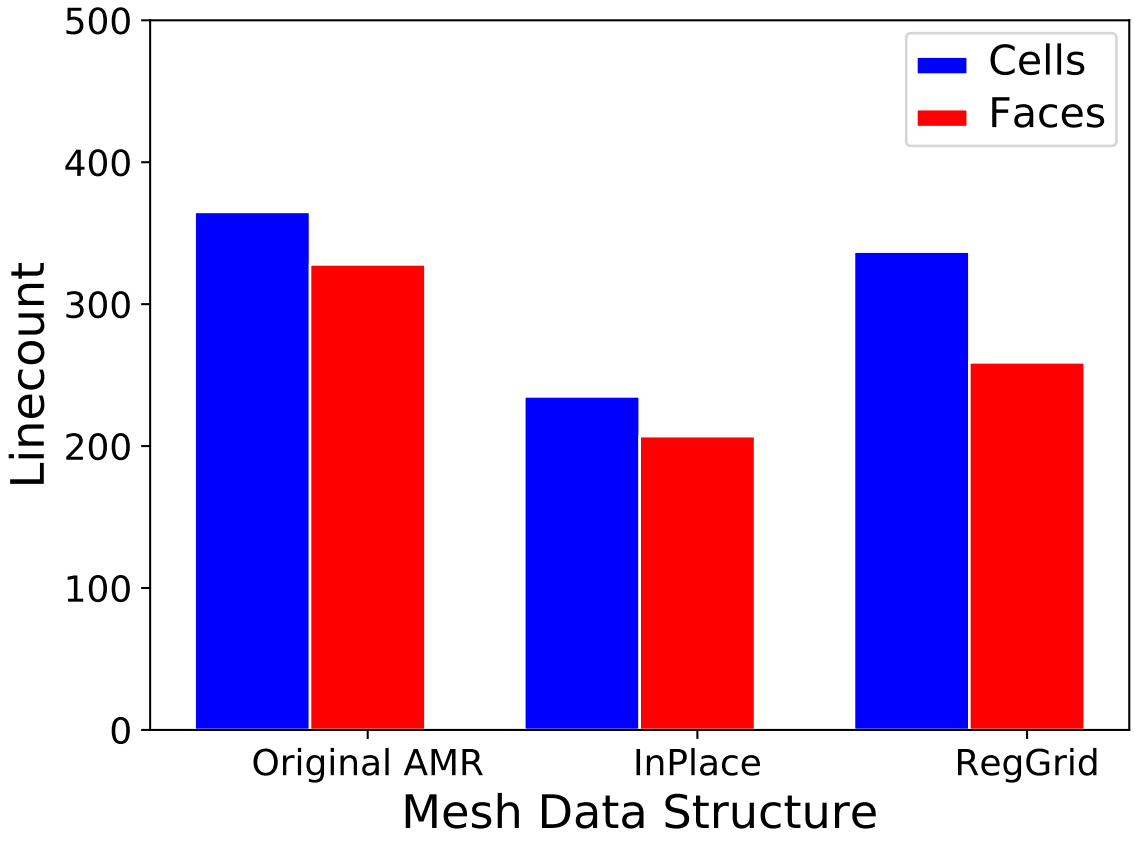


Figure: Line Counts for Phantom AMR versus Original AMR methods

Examples (cont.)

Vectorization

- Supposed to give 2-3 times speedup
- Can be difficult to achieve
- Hardware/Software dependent

Parallel Computing

- Efforts necessary to write parallel implementations
- Phantom-cell AMR: 2 months vs. 2 years
- Kokkos-like frameworks in the future

Table of Contents

FIESTA

- **F**ast **I**nterface **E**volution, **S**hocks and **T**ransition in the **A**tmosphere
- Written in C++ with Kokkos and MPI support
- Current C++ "transition" from previous fire-model codes
- Future work will include combustion model(s) and cloud physics

Table of Contents

Motivation

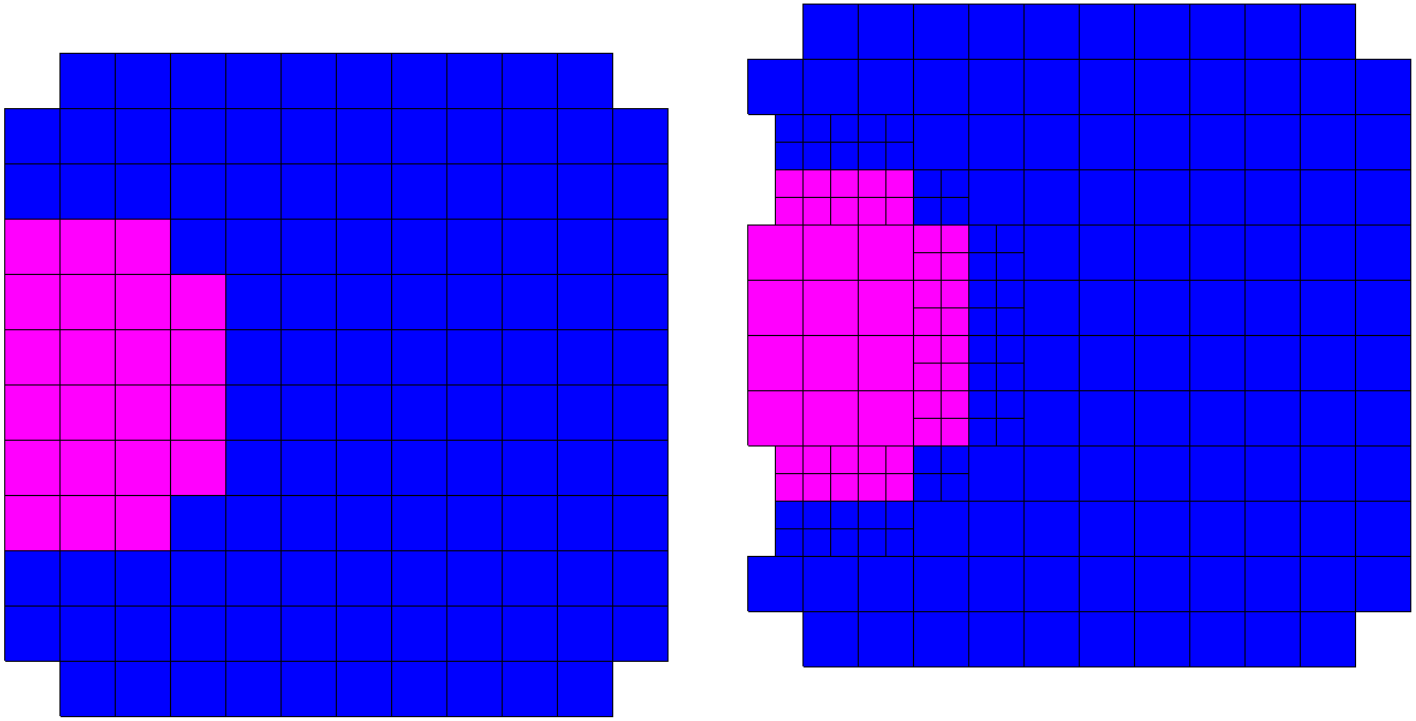
- Atmospheric calculations are large and require lots and data and computations
- We need AMR to help with the progress of the research, we can put the computation only where it is needed
- Example
  - 100 km x 100 km area with 10 m refinement
  - 100,000,000 cells, each with 8 variables of 8 bytes $\rightarrow$ 6.4 GBytes of data
  - Increase with more dimensions, higher refinement scale, more physics attributes
- Phantom-cell AMR is the key to an efficient transition to AMR

Task

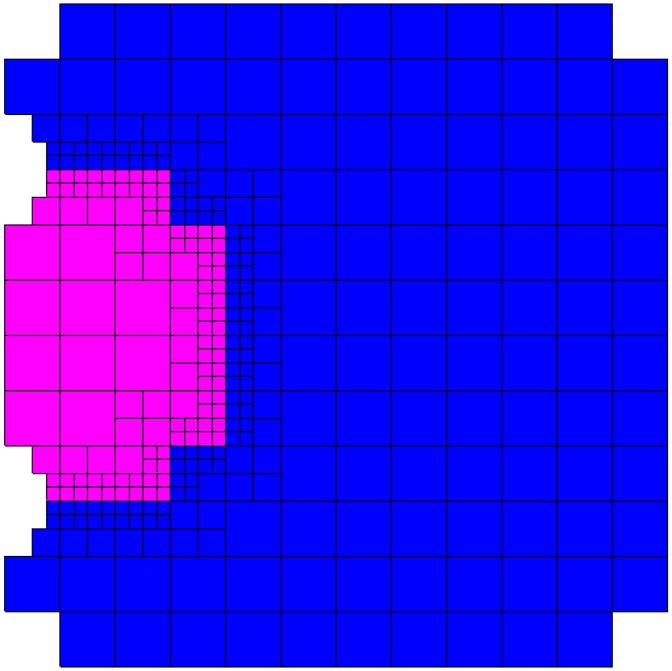
Need to transition the current regular grid code to an AMR based code

- Link CLAMR into the FIESTA runtime environment
- Change original physics routines to cooperate with CLAMR data structures
  - Change 3D data structures and loops to 1D
  - Change neighbor access pattern
- Add AMR mesh routines
- Verify the results (Real-time graphics with OpenGL)

Current State



Current State (cont.)



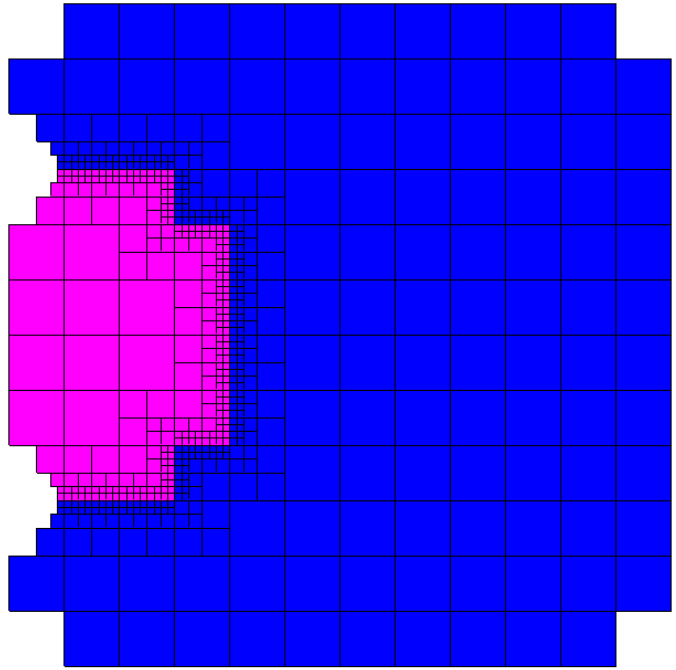


Table of Contents

Future Work

- Real-time graphics for original code
- More realistic model
  - Current test is a simple rising gas bubble
  - Transition to grass fire test and further
- Incorporate Kokkos into CLAMR for a smoother interface
- Add third dimension capability to CLAMR to accommodate necessary models
- Include corner cells for phantom cells

Table of Contents

Object-Oriented Programming (OOP)

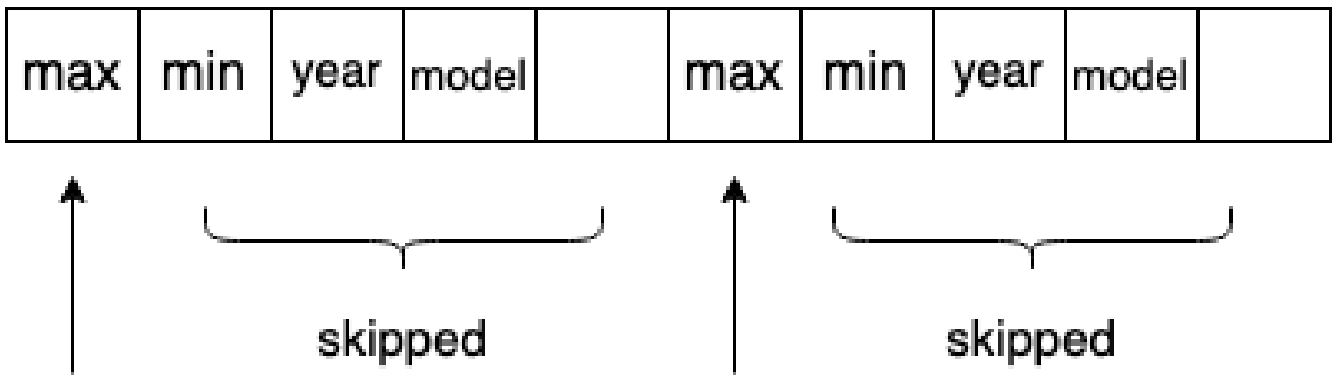
```
Class car
{
  double max_speed;
  double year;
  string model;
  string color;

  //constructors...
  car(double max_speed, double_year, string model, string color)
};
```

- Great for organizing code and finding commonalities between the objects.
- Great for working on a *single* object, does not scale when working with many objects.

OOP Limitations

If only one of the variables is being accessed at a time, the the data already loaded into cache is skipped.



Alternative: Data Oriented Design

Instead of focusing on objects, we need to shift focus to the data itself.

- *How* is the data laid out in memory?
- *How* will it be read and processed?
- *What* are the best data structures for my problem?

MATAR

MATAR (**MAT**rices and **AR**ray) is a `C++` library that provides **data oriented** data structures that aim to be Performance, Portable, and Productive.

- **Performant**: by allocating contiguous blocks of memory for data that will be accessed sequentially, leading to nearly perfect cache usage.
- **Portable**: by writing MATAR data structures to take advantage of Kokkos
- **Productive**: user doesn't have to think about memory, e.g., Did I `delete/free` my 6D array?

Table of Contents

Overview of MATAR's data structures

MATAR provides the following data types, and we will briefly cover some of them.

		Indexing pattern	
		0-indexed	1-indexed
Access pattern	Column major	FArray ViewFArray	FMatrix ViewFMatrix
	Row major	CArray ViewCArray	CMatrix ViewCMatrix

Figure: MATAR's **dense** data structures, grouped by memory access and indexing patterns

		Indexing pattern
		0-indexed
Access pattern	Column major	RaggedDownArray DynamicRaggedDownArray SparseColArray
	Row major	RaggedRightArray DynamicRaggedRightArray SparseRowArray

Figure: MATAR's **sparse** data structures, grouped by memory access and indexing patterns

Arrays

	j = 0	j = 1	j = 2	j = 3
i = 0	10.9	0.5	1.2	8.9
i = 1	0.9	110.1	7.3	45.1
i = 2	11.3	0.125	8.19	1
i = 3	90	81	14.12	11.9

Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Data	10.9	0.5	1.2	8.9	0.9	110.1	7.3	45.1	11.3	0.125	8.19	1	90	81	14.12	11.9

Figure: An example of how a 2D `CArray` object is accessed and laid out in memory

Secondary Motivation: Linked list

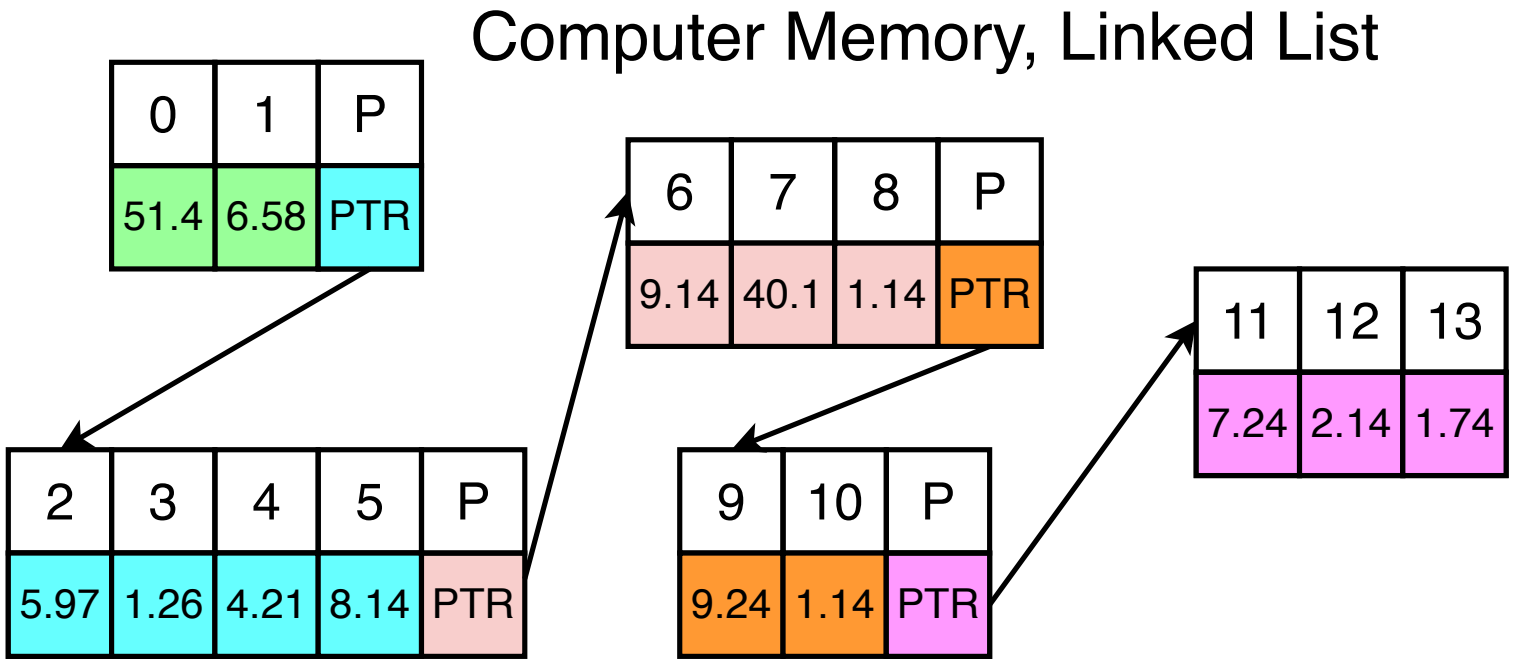


Figure: An example of how a linked list (where data are grouped by rows) might be laid out in memory

Difficult to harness parallel work efficiency

Ragged data structures

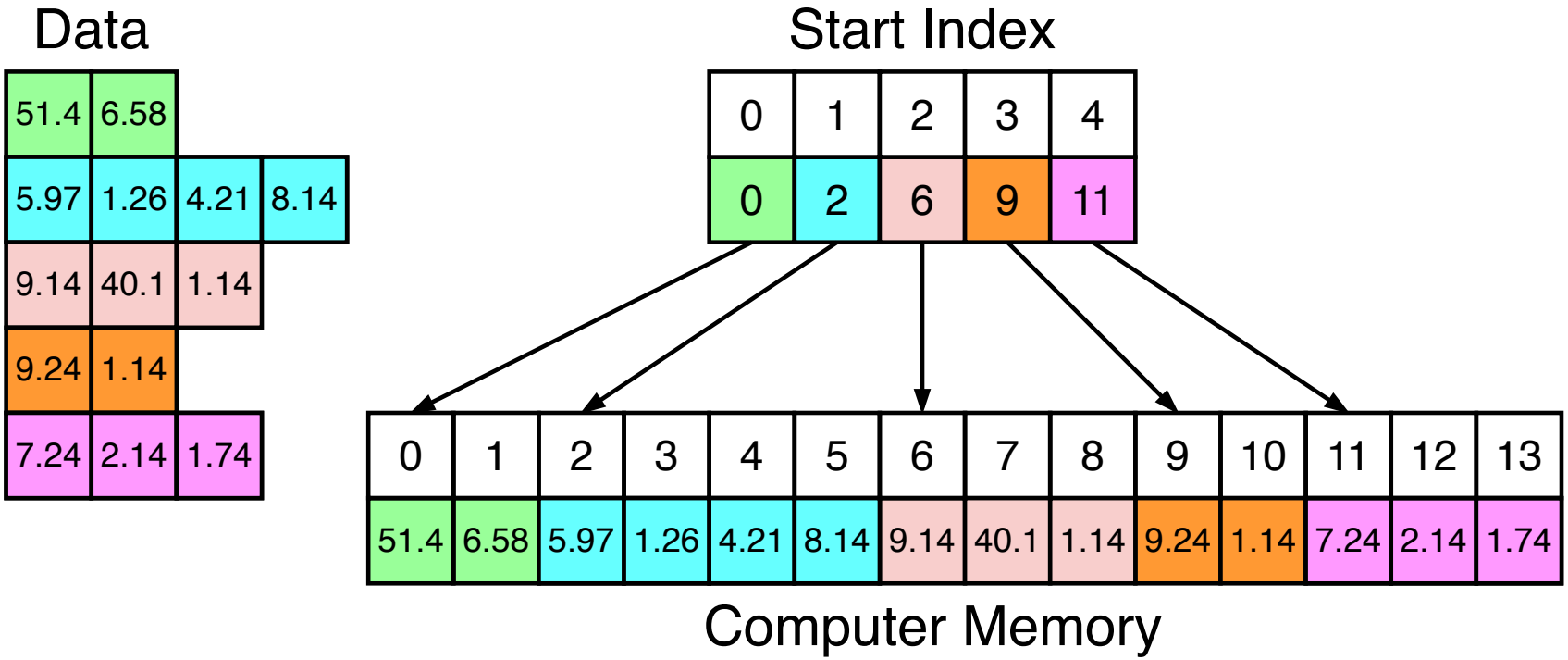


Figure: An example of how a `RaggedRightArray` object is accessed and laid out in memory

Ragged data structures (cont.)

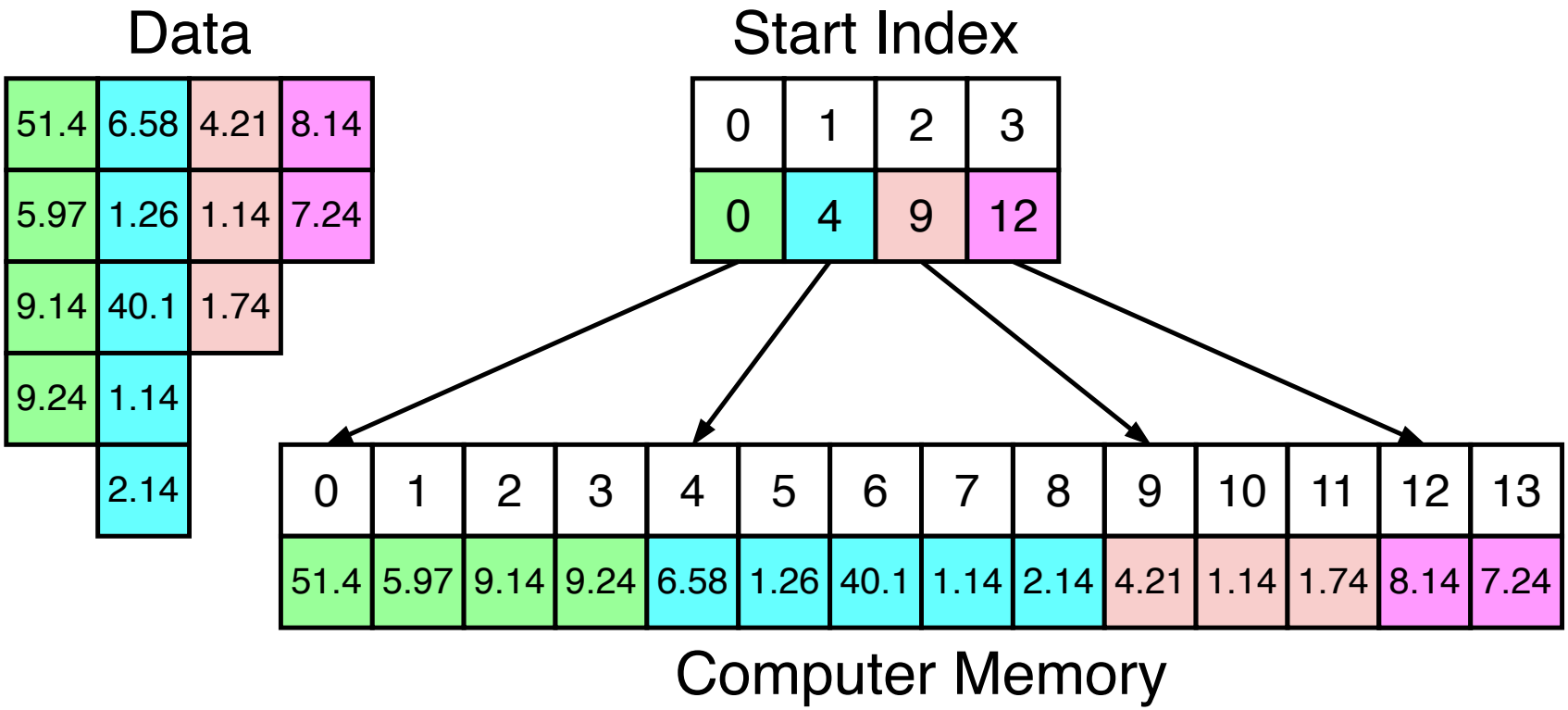


Figure: `RaggedDownArray` object

Table of Contents

MATAR vs. "regular" arrays

We conducted a series of performance benchmarks with the dense MATAR structures and "regular" `C++` arrays and collected the aforementioned information with LIKWID.
We will show some results from the following benchmarks:

- From the STREAM benchmark suite [4]:
  - The triad benchmark `z = ax + y`, where
    - `a` is a scalar and
    - `x, y, z` are vectors of the appropriate dimensions.

Note

"Regular" `C++` arrays are allocated dynamically (and not necessarily contiguously in memory).

[4] *Memory Bandwidth and Machine Balance in Current High Performance Computers.* McCalpin

MATAR vs. "regular" arrays (cont.)

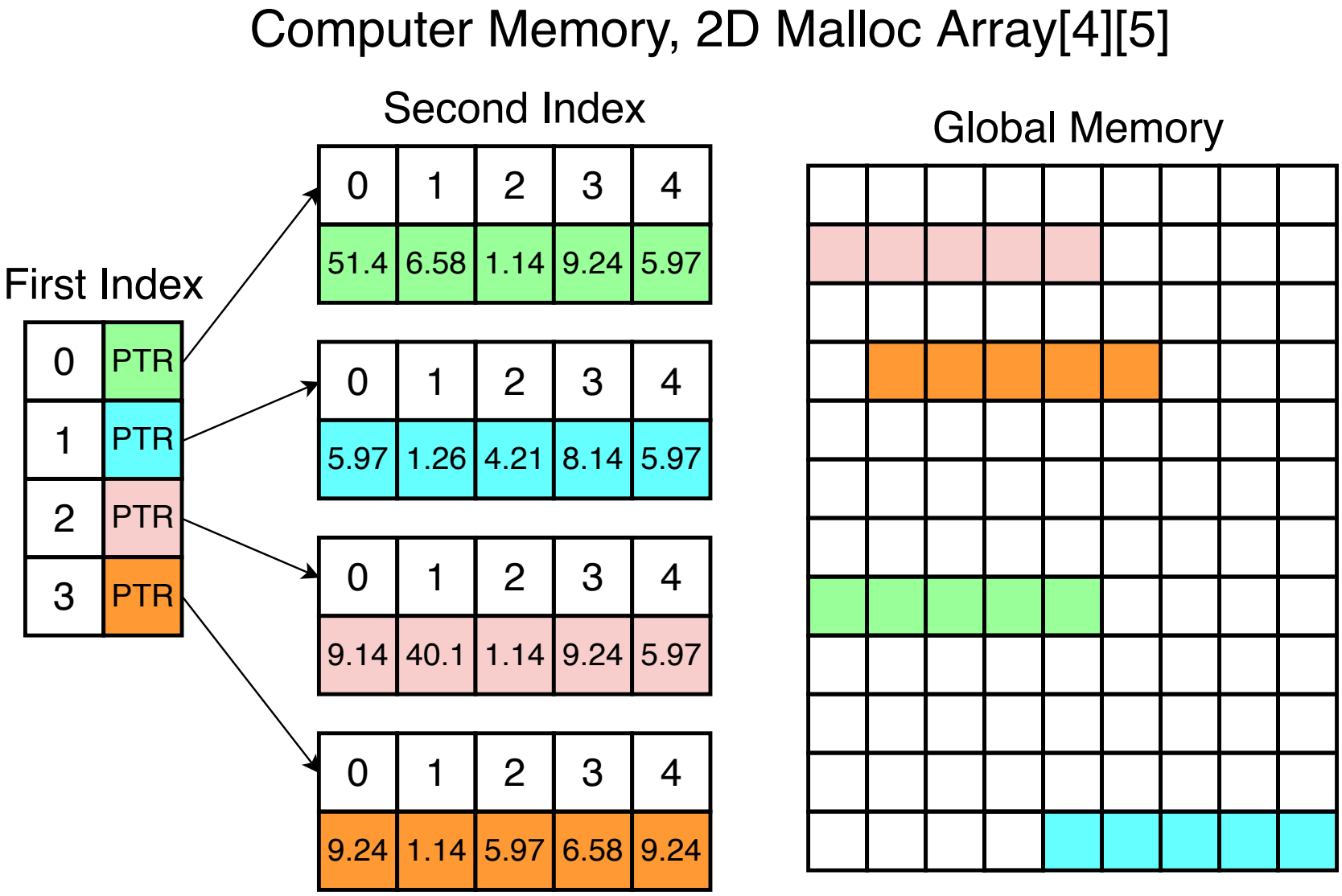


Figure: Example of "regular" C++ 2D array

Stream Benchmark Results (OpenMP)

Table: 1D (size 16,777,216) and 3D (size 256x256x256) STREAM Benchmark Test with Kokkos (CPU using OpenMP with max thread count). Units are milliseconds (averaged over 100 runs). *Benchmark was run on an IBM Power 9.*

	Data Structure	Copy	Scale	Sum	Triad	Dot Prod.
1D	CArrayKokkos	6.01	8.10	5.99	6.16	6.12
	Trad. Kokkos View	5.79	8.45	5.78	6.41	6.15
	ViewCArrayKokkos	5.84	7.92	5.97	6.21	6.04
3D	CArrayKokkos	6.56	6.26	8.16	8.26	6.37
	Trad. Kokkos View	6.57	6.49	8.51	9.00	7.08
	ViewCArrayKokkos	6.56	6.26	8.16	8.26	6.37

Stream Benchmarks Results (GPU)

Table: 1D (size 16,777,216) and 3D (size 256x256x256) STREAM Benchmark Test with Kokkos (GPU). Units are milliseconds (averaged over 100 runs). *Benchmark was run on a Nvidia V100.*

	Data Structure	Copy	Scale	Sum	Triad	Dot Prod.
1D	FArrayKokkos	0.44	0.44	0.49	0.49	0.45
	Trad. Kokkos View	0.45	0.44	0.49	0.49	0.45
	ViewFArrayKokkos	0.45	0.44	0.49	0.49	0.45
3D	FArrayKokkos	0.45	0.45	0.50	0.50	1.83
	Trad. Kokkos View	0.41	0.41	0.49	0.49	1.69
	ViewFArrayKokkos	0.45	0.45	0.50	0.50	1.83

Matrix Multiply

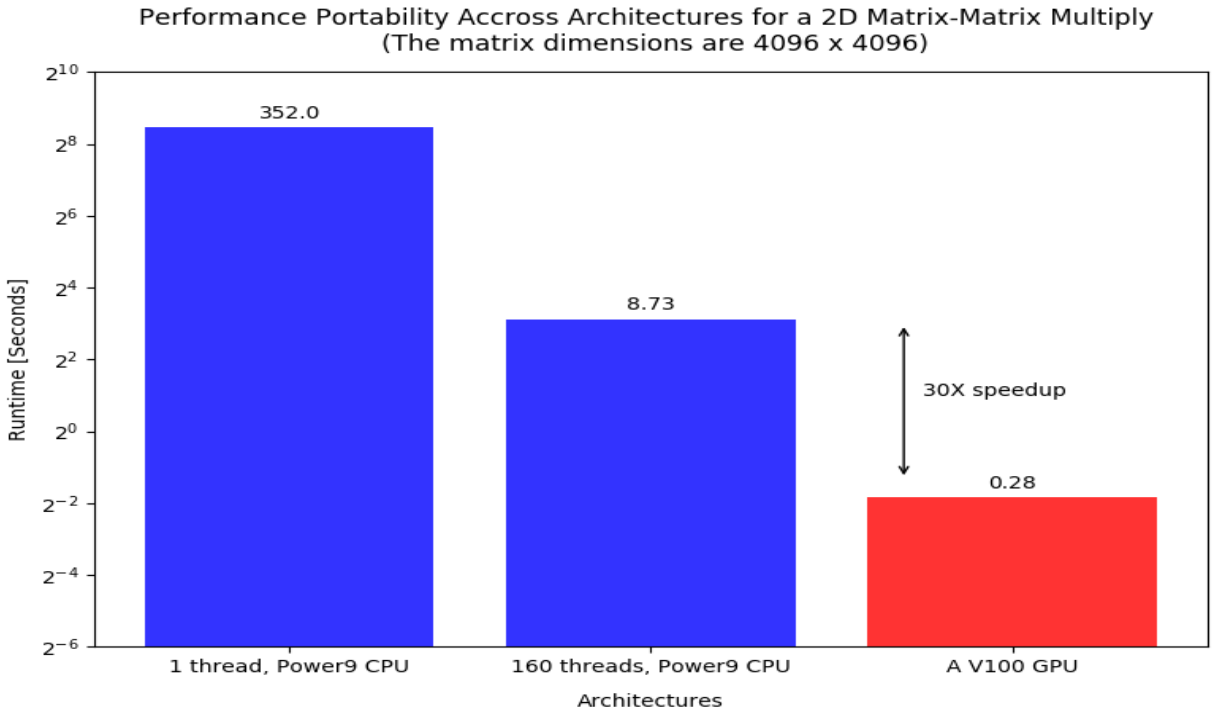


Figure: 2D matrix-matrix multiply run-time results on the IBM Power 9 architecture and Nvidia V100 GPU. The run-time is shown for CPU serial, CPU parallel (with max threads), and GPU parallel runs.

Future work

- Case study with a real application, both dense and sparse data structures
- More tests with sparse arrays on the GPU, especially dynamic arrays
- Work to get the library to be as plug-and-play as possible

Acknowledgements

Dr. Zhuang and Dr. Chen

Acknowledgements

Dr. Zhuang and Dr. Chen

Nathaniel Morgan and Bob Robey

Acknowledgements

Dr. Zhuang and Dr. Chen

Nathaniel Morgan and Bob Robey

Dr. Lim and Dr. Bornia

Acknowledgements

Dr. Zhuang and Dr. Chen

Nathaniel Morgan and Bob Robey

Dr. Lim and Dr. Bornia

Texas Tech Computer Science Department