








Article

On a Simplified Approach to Achieve Parallel Performance and Portability Across CPU and GPU Architectures

Nathaniel Morgan ^{1,*}, Caleb Yenusah ², Adrian Diaz ³, Daniel Dunning ¹, Jacob Moore ^{3,†},
Erin Heilman ³, Calvin Roth ^{3,‡}, Evan Lieberman ³, Steven Walton ², Sarah Brown ¹, Daniel Holladay ⁴,
Marko Knezevic ⁵, Gavin Whetstone ^{3,§}, Zachary Baker ^{1,||} and Robert Robey ^{3,¶}

¹ Engineering Technology & Design Division, Los Alamos National Laboratory, Los Alamos, NM 87545, USA

² Theoretical Division, Los Alamos National Laboratory, Los Alamos, NM 87545, USA

³ Computational Physics Division, Los Alamos National Laboratory, Los Alamos, NM 87545, USA

⁴ Computer, Computational & Statistical Sciences Division, Los Alamos National Laboratory, Los Alamos, NM 87545, USA

⁵ Department of Mechanical Engineering, University of New Hampshire, Durham, NH 03824, USA

* Correspondence: nmorgan@lanl.gov

† Now a research professor at Mississippi State University.

‡ Graduate student at University of Minnesota.

§ Graduate student at Texas A&M University.

|| Undergraduate student at University of Colorado.

¶ Now working at AMD corporation.

Abstract: This paper presents software advances to easily exploit computer architectures consisting of a multi-core CPU and CPU+GPU to accelerate diverse types of high-performance computing (HPC) applications using a single code implementation. The paper describes and demonstrates the performance of the open-source C++ **matrix** and **array** (MATAR) library that uniquely offers: (1) a straightforward syntax for programming productivity, (2) usable data structures for data-oriented programming (DOP) for performance, and (3) a simple interface to the open-source C++ Kokkos library for portability and memory management across CPUs and GPUs. The portability across architectures with a single code implementation is achieved by automatically switching between diverse fine-grained parallelism backends (e.g., CUDA, HIP, OpenMP, pthreads, etc.) at compile time. The MATAR library solves many longstanding challenges associated with easily writing software that can run in parallel on any computer architecture. This work benefits projects seeking to write new C++ codes while also addressing the challenges of quickly making existing Fortran codes performant and portable over modern computer architectures with minimal syntactical changes from Fortran to C++. We demonstrate the feasibility of readily writing new C++ codes and modernizing existing codes with MATAR to be performant, parallel, and portable across diverse computer architectures.

Keywords: performance; portability; productivity; GPUs; dense and sparse data; fine-grained parallelism



Citation: Morgan, N.; Yenusah, C.; Diaz, A.; Dunning, D.; Moore, J.; Heilman, E.; Roth, C.; Lieberman, E.; Walton, S.; Brown, S.; et al. On a Simplified Approach to Achieve Parallel Performance and Portability Across CPU and GPU Architectures. *Information* **2024**, *15*, 673. <https://doi.org/10.3390/info15110673>

Academic Editors: Lenore Mullin and John L. Gustafson

Received: 16 September 2024

Revised: 5 October 2024

Accepted: 10 October 2024

Published: 28 October 2024



Copyright: © 2024 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (<https://creativecommons.org/licenses/by/4.0/>).

1. Introduction

For decades, a CPU's performance grew by adding more transistors per square centimeter. The performance often doubled every 18 to 24 months, which is Moore's law. Currently, 5 nm chips exist, but it is becoming more difficult and costly for the semiconductor industry to reach smaller sizes (e.g., TSMC is working towards 2 nm transistor sizes [1]), so Moore's law has been slowing down. This is especially true for server-sized chipsets. At some point, achieving smaller sizes will be prohibitive financially or physically.

To achieve continued performance gains in computing, hardware manufacturers have been adding more cores to a CPU (termed a homogeneous architecture) or adding a GPU to a multi-core CPU (termed a heterogeneous architecture). Fine-grained parallelism is

required to take advantage of these modern computer architectures, but there are notable challenges for software developers. Heterogeneous computer architectures are designed around vendor-specific programming languages (e.g., CUDA, HIP, SYCL), and those languages differ from those used for fine-grained parallelism on homogeneous computer architectures (e.g., OpenMP or pthreads). The variance in fine-grained parallelization languages/approaches creates challenges with writing and maintaining portable software. There are many approaches currently being researched to simplify the process of writing performance portable software. They fall into two categories, programming models or custom optimizing compilers. There are multiple compiler side options that attempt to achieve performance portability [2,3]. These tend to be coupled with automated optimization frameworks [4,5] to generate high performance executables for highly specific applications, generally machine learning inference or training. They can give excellent results for these applications with predictable runtime pathways and repeated matrix operations, but are not generally applicable to complex multi-physics simulations because of their inherent inability to predict process flow. Also, a review of the programming syntax shows that they do not meet our requirements of having straightforward syntax for performance portability. There are also multiple programming models being developed to allow a single code implementation to run across homogeneous and heterogeneous architectures [6–8]. Of these, Kokkos is the most robust and has the broadest range of backend support for HCP applications. Kokkos currently allows developers to target CUDA, HIP, SYCL, HPX, OpenMP and C++ threads as backend programming models with several other backends in development and is built using industry standard compilers. As such, it is the natural choice as a starting point for simplifying performance portability.

The approach taken by the Kokkos library provides advantages over similar portability approaches such as writing native OpenCL or using Lift+OpenCL [9]. The approach of OpenCL was that each vendor (Nvidia, AMD, Intel, etc.) would create an OpenCL software development kit (SDK), and any code written using OpenCL could be executed on that vendor's hardware. With this approach, OpenCL had an advantage over languages like CUDA because it was an open standard, and it allowed for incremental porting of functions since memory transfers were not required for code correctness on the CPU vendors' SDKs. The ability to incrementally rewrite code is especially important for developers who maintain large code bases. However, OpenCL had a notable disadvantage compared to CUDA because the Nvidia OpenCL SDK did not have the full functionality available in CUDA. The approach taken by Kokkos was to shift the responsibility away from vendors and onto themselves to implement their own backend for each vendor. In this approach, Kokkos created its own C++ application programming interface (API) and implemented a backend for vendor-specific programming languages. For example, rather than Nvidia creating an OpenCL SDK, Kokkos created a CUDA backend that could take full advantage of vendor-specific capabilities. This same process can be repeated for any vendor, and since Kokkos also supports CPU backends, incremental porting is possible with this approach. With Kokkos's approach, one maintains the portability and the ability to incrementally port capabilities as was available with OpenCL, but also gains the performance attained by using a vendor-specific backend such as CUDA. Kokkos, at this time, supports many backends, including OpenMP and pthreads for multi-core CPUs, CUDA for Nvidia GPUs, HIP for AMD GPUs, and SYCL for Intel GPUs.

The Kokkos portability library uses C++ template meta-programming that is exceptionally powerful and flexible, but the coding syntax can be daunting to non-expert developers. In addition, modernizing large codes to take advantage of the Kokkos library is a challenging task and the complexity increases with Fortran codes. The open-source C++ **matrix** and **array** (MATAR) library [10] (MATAR can be accessed at <https://github.com/lanl/MATAR>, accessed on 5 October 2024) was created to help efforts to modernize existing software and write new performant and portable C++ software as shown in Figure 1. Performance portability across architectures is essential to ensure codes can utilize advanced computer architectures comprised of GPUs, which are used in the three upcoming exascale ma-

chines—El Captain, Frontier, and Aurora. MATAR is a header-only file (for easy integration with software) and works with all commonly used C++ compilers, including GCC, Intel, Clang, PGI, and more. The complexity of Kokkos is hidden inside MATAR to create an easy-to-use syntax to aid adoption by both novice and expert Fortran, Python, and C/C++ programmers. In other words, MATAR is a higher-level API for performance portability that makes GPU programming accessible to all.

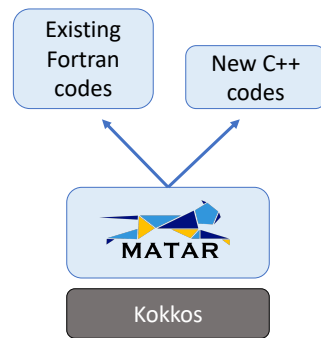


Figure 1. The C++ MATAR library builds on the Kokkos library to enable diverse software to run on multi-core CPUs and GPUs with a single implementation and enables a common approach to modernize existing Fortran software and to write new C++ software.

The Kokkos library is not tailored to a specific application, so by itself, it does not support dynamic and static sparse data structures such as ragged arrays and sparsely populated matrices. Within the Kokkos ecosystem, there is a separate C++ Kokkos Kernels library [11] that provides some sparse data structures to support, for instance, cuBLAS, BLAS, Sparse BLAS, and Graph Kernels. Support for diverse sparse data types is important to efficiently implement many numerical methods and the data structures that underpin engineering and physics codes. To address this need, the MATAR library extends Kokkos to support a wide range of dense and sparse data representations (both static and dynamic) used in diverse software and engineering codes that are implemented in C++ or Fortran and run on high-performance computing (HPC) machines. The data types in MATAR are written using data-oriented programming (DOP) for performance. The data types, functions, and macros for parallel execution offered by MATAR are designed for general software applications.

MATAR is currently in use for many software applications for engineering and scientific research. MATAR has been used in single physics mini-apps ranging from phase field modeling for simulating chemical kinetics during solidification [12] to evolving the fully resolved microstructure of metals using the elasto-viscoplastic fast Fourier transform (EVPFFT) model [13]. MATAR is also the backbone of the 2024 R&D 100 award winning open-source C++ Fierro mechanics code [14], which has both implicit and explicit Lagrangian methods [15–25] and micromechanical models [13,26,27]. Given MATAR's growing adoption in software applications, there is a need for documentation on the MATAR implementation syntax and how to use it for performance portability.

This paper is also concerned with addressing the challenges of making existing Fortran codes performant and portable over multi-core CPUs and multi-core CPUs plus GPUs. The MATAR library offers a straightforward Fortran-like syntax and contains unique data types to support Fortran codes. Examples are presented in this paper to illustrate the advantages and versatility of using the C++ MATAR library to readily modernize existing Fortran codes to be performant and portable across multi-core CPUs and GPUs. To demonstrate the utility of MATAR for diverse software applications, a broad range of test cases covering graph construction, artificial neural networks (ANNs), simulating heat transfer in geologic applications, and solving the isotropic 3-wave kinetic equation are presented with their performance across both CPUs and GPUs used in HPC systems.

The novelty of this paper is as follows: (1) It introduces new capabilities in the MATAR library that simplify software development for leveraging the power of GPUs and CPUs

with a single code implementation. (2) It documents the simple syntax to use the MATAR library for performance portability. (3) It presents a new approach, based on the MATAR library, to transition well-established Fortran codes to C++ while enabling parallel execution on both CPUs and GPUs. (4) It demonstrates that the MATAR library allows a diverse suite of C++ applications to run quickly in parallel across GPUs and CPUs.

The layout of the paper is as follows: The details of the MATAR library are described in Section 2. The performance results for various computer science and computational physics focused test cases are presented in Section 3, and concluding remarks and a summary of our findings are given in Section 4.

2. The MATAR Library

The design philosophy for the C++ MATAR library can be summarized with four key points. (1) Enable a coding syntax that is discernible to C, Python, and Fortran programmers while adhering to the C++ standard. (2) Eliminate all fine-grained parallel language syntax in a code (e.g., no pragmas, no CUDA, etc., should be seen in the coding). (3) Enable existing codes to run in parallel on multi-core CPUs and GPUs after minor changes to the original syntax. (4) Easily manage memory on CPUs and GPUs. The subsections that follow here will describe the MATAR library in greater detail.

2.1. Performance and Data Types

The MATAR library is based on the data-oriented programming (DOP) approach because DOP can deliver superior runtime performance compared to object-oriented programming (OOP) and other programming paradigms. This is because DOP matches the data layout and access pattern to best suit the underlying hardware. The concept with the OOP approach is to create many objects that have corresponding data, and when accessing the data, it can result in cache misses that result in poor runtime performance. The foundational concept of the DOP approach is to contiguously access data so that multiple values are loaded into the cache and used, which gives excellent performance and allows for vectorization. Further, data are co-located on the same or a smaller set of pages to reduce page misses, which are technically known as translation look-aside buffer (TLB) misses, and avoid page walks to locate the data. The OOP approach has many merits (for instance, with regards to cleanly organizing code), and DOP can be used in concert with it. The data types in MATAR conveniently hide the complexity of using the DOP approach to enhance the productivity of writing codes. While the original paper on MATAR [10] largely focused on the DOP implementation, this paper addresses more applications and how to use the library.

The MATAR library contains both multi-dimensional array (0 indexed) and matrix (1 indexed) data types implemented using the DOP approach. Figure 2 summarizes some of the architecture portable data types in MATAR. The data inside the multi-dimensional array data types are accessed using indices from 0 to less than N, which follows the C/C++ language. The data inside the multi-dimensional matrix data types are accessed with indices from 1 to N, like the Fortran language, which aids in the conversion of existing Fortran coding to C++. MATAR also contains multi-dimensional dense and sparse data types that access the data sequentially following either the row-major (C/C++) or column-major (Fortran) conventions, which is key to ensuring code loops deliver the optimal runtime performance, see Listing 1. The naming conventions for the multidimensional dense data types in MATAR include a C or F to denote the layout convention, e.g., CMatrixKokkos follows the C/C++ access convention and FMatrixKokkos follows the Fortran access convention. The Kokkos designation in the name denotes a data type that interfaces with the Kokkos library. The MATAR library also contains data types based on DOP that do not interface with the Kokkos library, and those data types only run on the CPU.

		Indexing pattern	
		0-indexed	1-indexed
Access pattern	Column major	FArrayKokkos ViewFArrayKokkos DFArrayKokkos DViewFArrayKokkos	FMatrixKokkos ViewFMatrixKokkos DFMatrixKokkos DViewFMatrixKokkos
	Row major	CArrayKokkos ViewCArrayKokkos DCArrayKokkos DViewCArrayKokkos	CMatrixKokkos ViewCMatrixKokkos DCMatrixKokkos DViewCMatrixKokkos

		Indexing pattern
		0-indexed
Access pattern	Column major	RaggedDownArrayKokkos DynamicRaggedDownArrayKokkos CSCArrayKokkos
	Row major	RaggedRightArrayKokkos RaggedRightArrayOfVectorsKokkos DynamicRaggedRightArrayKokkos CSRArrayKokkos

Figure 2. A range of dense (**left** chart) and sparse (**right** chart) data types in MATAR are shown above that are designed for performance portability across CPUs and GPUs. Additional data types are provided in MATAR than shown above here; for instance, there are data types that solely run on CPUs and dynamically resizable 1D array and 1D matrix types.

Listing 1. Following the DOP approach, data can be sequentially accessed from multidimensional MATAR array and matrix types when using the appropriate loop ordering based on the C/C++ and Fortran language conventions. The coding shown in (a) and (b) contain two separate code snippets, where the first snippet is the memory allocation that is performed on the host (always a CPU) and the second snippets are for the loops (arranged to give optimal performance with the C/C++ or Fortran matrix access types) that would be inside a routine that is run on the device, either a multi-core CPU or a GPU, respectively.

```
// ...
// inside a routine run on the host

// Allocate a 2D Matrix on the device
CMatrixKokkos <double> c_matrix(3,4);

// ...
// inside a routine run on the device

// DOP with the C/C++ access convention
for (i = 1; i <= 3; i++) {
    for (j = 1; j <= 4; j++) {
        c_matrix(i,j) = 3.14;
    }
}
```

(a) A 2D matrix with the C/C++ access convection

```
// ...
// inside a routine run on the host

// Allocate a 2D Matrix on the device
FMatrixKokkos <double> f_matrix(3,4);

// ...
// inside a routine run on the device

// DOP with the Fortran access convention
for (j = 1; j <= 4; j++) {
    for (i = 1; i <= 3; i++) {
        f_matrix(i,j) = 3.14;
    }
}
```

(b) A 2D matrix with the Fortran access convection

The memory on a GPU is typically separate from the memory of the CPU (true at the time of writing of this paper). All Kokkos-based data types in MATAR that allocate memory

do so on the device memory (e.g., the GPU), and then access that memory on the device by default. Special Kokkos-based data types are also offered that allocate memory on both the host and the device; these are termed dual data types. The dual data types start with the letter D as in `DCArrayKokkos`, `DCMatrixArray`, `DFArrayKokkos`, and `DFMatrixKokkos` (see Listing 2). The dual data types contain member functions to readily copy data between the host and device (and vice versa) and member functions to access the data on either the host or the device side. For the case of running a code in parallel on a CPU, the dual data types only allocate memory on the CPU since the device is the host (i.e., there is no duplicate memory when building with the OpenMP or pthreads backends).

Listing 2. MATAR offers dual data types that allocate memory on both the CPU and the GPU.

```
// allocate a 4x5 array on the CPU and GPU
DCArrayKokkos <double> array2D (4,5);

// use array2D on the CPU (always the host)
array2D.host(i,j) = 3.45;

// use array2D on the device (e.g., GPU)
array2D(i,j) = 3.45;

// methods to copy data
array2D.update_device(); // update e.g., GPU
array2D.update_host(); // update CPU
```

A salient capability of the MATAR library is offering a rich set of sparse data types, see Figure 2. Some of the sparse data types have compact storage, containing only enough room to store the non-zero values in the array. Examples of sparse data types with an optimal memory footprint, and adhering to DOP, are the compressed row and column storage types (`CSRArrayKokkos` and `CSCArrayKokkos`) and the ragged array types (`RaggedRightArrayKokkos` and `RaggedDownArrayKokkos`). MATAR also supports sparse types that contain a memory buffer that can be pushed back into dynamically to save more data. The memory buffer size is set when data types are allocated. An example of a dynamic sparse type is the `DynamicRaggedRightArrayKokkos`. The flexibility to dynamically resize the array comes at the expense of carrying additional memory (i.e., they have a larger memory footprint than a compressed storage type) and they break DOP. MATAR also offers portable dynamically resizable 1D array and 1D matrix types that run across CPUs and GPUs called `DynamicArrayKokkos` and `DynamicMatrixKokkos`. These dynamic data types are key to applications across engineering and physics applications requiring dynamically resizable arrays, examples include codes with adaptive mesh refinement and multi-material multiphysics codes.

The MATAR library also offers data types to view or slice existing arrays or matrices. The MATAR views work with existing allocated memory (in a 1D layout), and can be used to access the 1D data as a multi-dimensional array or matrix layout. As such, all MATAR views differ notably from the `Kokkos::View` data type. MATAR view data types are supported for just a CPU or for the device (which can be the CPU or the GPU depending on the Kokkos backend used). Listing 3 illustrates how to slice an existing array on the device. A special dual view in MATAR addresses the case of viewing 1D data (that was already allocated on the CPU) on both the CPU and GPU as a multi-dimensional array or matrix (see Listing 4). For that dual view data type, a memory allocation on the GPU is performed for the user and filled with data from the existing CPU memory. The MATAR views are key to incorporating DOP in a code and greatly simplify code implementation. Furthermore, the dual view data types are quite useful for converting a subset of subroutines within a large Fortran code (or an existing C/C++ code) to run in parallel on GPUs.

Listing 3. MATAR enables viewing data as multidimensional on the device, as well as the slicing of data. In this example, the stress in a single element is sliced out of an array and then used on the device (e.g., GPU). Modifying the values inside the stress array will also modify the corresponding values in the global `elem_stress` array.

```
// ...
// inside a routine run on the device

// slice out the stress in an element of the mesh
int elem_id=314; // the element index
ViewCArrayKokkos <double> stress(&elem_stress(elem_id,0,0),3,3);

// Modify the stress
for (int i = 0; i < 3; i++) {
    stress(i,i) = -pressure;
} // end for loop
```

Listing 4. MATAR offers many data types to view data as multidimensional on both the CPU side and GPU side.

```
// on the CPU
int A[27]; // an allocated array
DViewCArrayKokkos <int> array3D (&A[0],3,3,3);

// use array3D on the CPU (always the host)
array3D.host(i,j,k) = 3;

// use array3D on the device (e.g., GPU)
array3D(i,j,k) = 4;

// methods exist to update GPU or CPU array3D
array3D.update_host();
```

2.2. Simplifying Parallelism and Portability

The Kokkos data types in the MATAR library are designed for fine-grained parallelism on CPUs and GPUs, which differs from coarse-grained parallelism commonly used in massively parallel HPC codes. Both types of parallelism can be used in a code. One example of coarse-grained parallelism is to decompose the simulation mesh into blocks and use the MPI (message passing interface) to communicate between the smaller mesh blocks. With fine-grained parallelism, the concept is to spread the work of individual loops over the cores, or the equivalent on a GPU, via launching threads. There are several ways to implement fine-grained parallelism within a code. One approach is to thread every loop in a function and another approach is to thread over a loop that calls a function. The performance gains with these two approaches to fine-grained parallelism are dependent on the function. To achieve the best performance for fine-grained parallelism, the work performed in the parallel loop should be equal between each thread, i.e., thread divergence and load imbalance should be minimized. Thread divergence can be especially detrimental to GPU performance.

The MATAR library offers a simple syntax to execute a loop in parallel (see Listing 5) on the CPU in the case of using OpenMP or pthreads, or a GPU in case of using HIP, CUDA, or SYCL. A similar parallel loop syntax is offered for loops inside the member functions of a class (or struct) that need to access the private variables as shown in Listing 6. MATAR also offers a simple syntax for parallel sum, min, and max reduction operations on the device, see Listing 7.

Listing 5. The parallel loop syntax with the MATAR library is FOR_ALL or DO_ALL. Up to three levels of parallelism are supported. The coding shown here will run in parallel on a multi-core CPU using, e.g., OpenMP or pthreads, and will run in parallel on a GPU using, e.g., CUDA for Nvidia hardware or HIP for AMD hardware. A serial loop syntax is also offered in the MATAR library for consistent coding syntax; simply replace the ALL with LOOP in the above examples for a serial loop.

```

// a parallel loop on the device from // a parallel loop on the device from
// i=0 to i<N                          // i=1 to i<=N
FOR_ALL (i, 0, N, {                       DO_ALL (i, 1, N, {
    // coding is here                      // coding is here
});                                       });

// a parallel loop on the device from // a parallel loop on the device from
// i=0 to i<N and j=0 to j<M           // i=1 to i<=N and j=1 to j<=M
FOR_ALL (i, 0, N,                         DO_ALL (j, 1, M,
    j, 0, M, {                             i, 1, N, {
    // coding is here                        // coding is here
});                                       });

// a parallel loop on the device from // a parallel loop on the device from
// i=0 to i<N,                          // i=1 to i<=N,
// j=0 to j<M, and                       // j=1 to j<=M, and
// k=0 to k<P                             // k=1 to k<=P
FOR_ALL (i, 0, N,                         DO_ALL (k, 1, P,
    j, 0, M,                               j, 1, M,
    k, 0, P, {                             i, 1, N, {
    // coding is here                        // coding is here
});                                       });

```

Listing 6. A parallel loop syntax is offered with the MATAR library for loops inside a class or struct that must access private member variables. Up to three levels of parallelism are supported; a single level of parallelism is shown above here. Supporting parallelization inside member functions is key to supporting object-oriented programming.

```

// a parallel loop on the device from
// i=0 to i<N
FOR_ALL_CLASS (i, 0, N, {
    // coding is here
});

```

MATAR also offers a way to run serial coding on the device, see Listing 8. This is important in applications such as setting pointers, setting an enumeration value, or setting simulation parameters on a GPU. In addition, it is also potentially helpful to run a serial loop that is not thread-safe and accesses variables stored in the GPU memory. It is important to emphasize that accessing the device values in a Kokkos-based data type can only be performed on the device; as such, access must be performed inside a FOR_ALL, FOR_REDUCE_SUM, RUN, etc.

The MATAR library offers a simple syntax to exploit nested parallelism. Listing 9 shows examples of nested parallelism. This nested parallelism refers to loosely nested parallelism, differing from the tightly nested parallelism explained earlier in this section. For clarity, Kokkos and others refer to this as ‘hierarchical parallelism’. This type of parallelism arises when inner loop(s) need information from the outer loop(s) for loop definition. Shown in Listing 9a, the inner loop needs to use outer loop’s *i* index as the size of the next loop, such as in a ragged access pattern. With a standard tightly nested parallel loop, this index cannot be abstracted and used for inner loop sizing.

Listing 7. The MATAR library offers a simple syntax for parallel sum, max, and min reductions. The parallel sum reduction is shown above here. Up to three levels of parallelism are supported with the reduction operations. The coding shown here will run in parallel across CPU and GPU hardware.

```

// a parallel sum on the device from
// i=0 to i<N
FOR_REDUCE_SUM (i, 0, N,
                local_answer, {
                // coding is here
                }, answer);

// a parallel sum on the device from
// i=0 to i<N and j=0 to j<M
FOR_REDUCE_SUM (i, 0, N,
                j, 0, M,
                local_answer, {
                // coding is here
                }, answer);

// a parallel sum on the device from
// i=0 to i<N,
// j=0 to j<M, and
// k=0 to k<P
FOR_REDUCE_SUM (i, 0, N,
                j, 0, M,
                k, 0, P,
                local_answer, {
                // coding is here
                }, answer);

// a parallel sum on the device from
// i=1 to i<=N
DO_REDUCE_SUM (i, 1, N,
               local_answer, {
               // coding is here
               }, answer);

// a parallel sum on the device from
// i=1 to i<=N and j=1 to j<=M
DO_REDUCE_SUM (j, 1, M,
               i, 1, N,
               local_answer, {
               // coding is here
               }, answer);

// a parallel sum on the device from
// i=1 to i<=N,
// j=1 to j<=M, and
// k=1 to k<=P
DO_REDUCE_SUM (k, 1, P,
               j, 1, M,
               i, 1, N,
               local_answer, {
               // coding is here
               }, answer);

```

Listing 8. The syntax to run serial code on the device is shown.

```

// run code on the device
RUN ({
    // coding is here
});

// run code on the device inside a member
// function that accesses private variables
RUN_CLASS ({
    // coding is here
});

```

While this parallelism is powerful, it is certainly limiting. Specifying each layer targets GPU team, thread, and vector sizes, respectively. There is no other parallel layer device, so this parallelism is limited to three layers of parallelism, shown in Listing 9 by FIRST, SECOND, and THIRD names. The standard tightly nested parallelism described previously in this section is more performant for most applications and has more flexibility (e.g., more layers of parallelism), so this hierarchical parallelism should only be used when necessary.

Listing 9. The MATAR programming syntax is shown for nested parallelism. The ranges are from $i \in [0 : N - 1]$, $j \in [0 : M - 1]$, and $k \in [0 : P - 1]$. In examples (a) and (b), the array is set equal to a value. In example (c), we multiply a vector and an array in parallel on the device, $y = A \cdot x$. In example (d), two arrays are multiplied together in parallel on the device, $C = A \cdot B$. Nested parallelism is for special cases where parallel loops or reduction operations are performed inside other parallel loops.

<pre> // run code on the device FOR_FIRST (i, 0, N, { // coding can be here too FOR_SECOND{j, 0, i, { A(i,j) = 1.0; }}; }); </pre> <p>(a) A nested parallel for loop</p>	<pre> // run code on the device FOR_FIRST (i, 0, N, { // coding can be here too FOR_REDUCE_SUM_SECOND{j, 0, M, local_answer, { local_answer += A(i,j)*x(j); }, answer); y(i) = answer; }); </pre> <p>(c) A nested sum reduction</p>
<pre> // run code on the device FOR_FIRST (i, 0, N, { // coding can be here too FOR_SECOND{j, 0, M, { // coding can be here too FOR_THIRD{k, 0, P, { A(i,j,k) = 1.0; }}; }}; }); </pre> <p>(b) Two nested parallel for loops</p>	<pre> // run code on the device FOR_FIRST (i, 0, N, { // coding can be here FOR_SECOND{j, 0, M, { // coding can be here too FOR_REDUCE_SUM_THIRD{k, 0, P, local_answer, { local_answer += A(i,k)*B(k,j); }, answer); C(i,j) = answer; }}; }); </pre> <p>(d) A nested parallel for loop and a sum reduction</p>

2.3. Conversion of Fortran Coding to C++

A benefit of using MATAR is that it reduces the syntactical changes that must be made to the original Fortran code to run with C++ on GPUs. For instance, the multidimensional column major matrix data types in MATAR are accessed identically to those written in Fortran with the same syntax, which allows large chunks of Fortran coding to be converted at once. While MATAR is a useful tool to help modernize Fortran codes, the Fortran to C++ conversion process still requires manually changing syntax within the code. One of the notable changes to a code is the conversion of the existing Fortran 'do' loop syntax to the DO_LOOP parallel syntax; however, the same loop ordering can be preserved, see Listing 10. Other minor syntactical changes include the following: if statements, logical operators, memory allocations, and comments. These syntactical changes to a code can be identified and implemented pragmatically, thus increasing the programmer's productivity. A side-by-side comparison of the C++ coding syntax for MATAR and Fortran is shown in Listing 10.

Listing 10. The MATAR programming syntax is compared to the Fortran programming syntax. The C++ coding syntax with MATAR is designed to be straightforward and has similarities to the Fortran language. In (a), the matrix is allocated on the device, which can be either the CPU or the GPU, depending on the Kokkos backend used, and the values of the matrix are set equal to 1. The contents inside the parallel loop, shown in (b), are executed on the device.

<pre> // allocate 10X10X10 matrix FMatrixKokkos <int> matrix3D (10,10,10); // Initialize values to 1 matrix3D.set_values(1); </pre>	<pre> ! Allocate 10X10X10 matrix INTEGER :: matrix3D (10,10,10) ! Initialize values to 1 matrix3D(:, :, :) = 1 </pre>	<p>(c) A 3D Fortran matrix allocation and setting the values equal to 1</p>
<pre> // Use matrix3D in parallel on the device DO_ALL (k, 1, 10, j, 1, 10, i, 1, 10, { matrix3D(i,j,k) += 1; }); </pre>	<pre> ! Use matrix3D DO k = 1, 10 DO j = 1, 10 DO i = 1, 10 matrix3D(i,j,k) = & matrix3D(i,j,k) + 1 END DO END DO END DO </pre>	<p>(d) Example use of a Fortran 3D matrix</p>
<p>(a) A 3D MATAR matrix allocation on the device and setting the values equal to 1 in parallel on the device</p>	<p>(b) Example use of the MATAR 3D matrix</p>	

2.4. Comparisons Between MATAR and Native Kokkos Syntax

The Kokkos library is very general, suitable for many applications, but its syntax can be difficult to understand and use. As such, the MATAR library was developed to be a higher-level API to Kokkos, allowing greater adoption of Kokkos in diverse applications. To illustrate the merits of such an API, we compare native Kokkos syntax to the corresponding syntax with the MATAR library in Listing 11.

2.5. Member Functions and Other Utilities with MATAR

The data types in the MATAR library offer many member functions to perform useful, routine operations or to return meta data about the array/matrix. For the latter case, member functions exist to return (a) the dimensions of the array/matrix, e.g., `array2D.dims(0)` returns the size of the first dimension in the 2D array; (b) the order of the array/matrix (i.e., it returns 1, 2, 3, ..., or 7 dimensions); or (c) the length of the 1D array storing all the data in the array/matrix, e.g., `array2D.length()` returns $N \times M$ where N is the first dimension and M is the second dimension of the array. For the sparse data types, member functions exist that return information on the stride size (e.g., the number of values in a row of a `RaggedRightArrayKokkos` type) or other information on a sparse/ragged array. If compiling MATAR with the debug option, all data types during runtime perform internal checks to insure the access of the array/matrix is within the correct dimensions and with the correct array/matrix order. Such checks help code developers quickly identify and resolve implementation errors.

On offering utility functions, the Kokkos-based data types in MATAR have a `set_values` member function that initializes the values in parallel on the device. To help developers with debugging, MATAR allows developers to supply names to the data types, e.g., see Listing 12; if an error is encountered during runtime, the name of the matrix/array is printed to the screen, informing the developer of which array/matrix has a problem. There is a member function to return the name of the array/matrix. To help developers couple a MATAR-implemented code to other libraries, there is a pointer member function that returns the address of the raw underlying data. The examples listed here are only a subset

of the member functions offered in the MATAR library, and more are being added as the utility becomes clear.

Listing 11. A side-by-side comparison of the MATAR programming syntax to the Kokkos syntax for allocating a 2D array, setting the values to 1, incrementing the values by 1, and performing a sum reduction. The MATAR library hides the complexity to use Kokkos, allowing more programmers to leverage the library for performance portability.

```

// allocate 10x10 array
CArrayKokkos <int> array2D(10,10);

// set value in parallel
array2D.set_values(1);

// increment value in parallel
FOR_ALL (i, 0, 10,
         j, 0, 10, {
             array2D(i,j) += 1;
         });

// sum the values in parallel
int loc_sum;
int answer;
FOR_REDUCE_SUM(i, 0, 10,
               j, 0, 10,
               loc_sum, {
                   loc_sum += array2D(i,j);
               }, answer);

```

(a) C++ MATAR 2D array example

```

#define LoopOrder Kokkos::Iterate::Right
using Layout = Kokkos::LayoutRight;
using ExecSpace = Kokkos::Cuda;
using MemoryTraits = void;

// allocate 10x10 array
Kokkos::View<int**, Layout, ExecSpace,
            MemoryTraits> array2D("arr2d",10,10);

// set value in parallel
Kokkos::parallel_for(
    Kokkos::MDRangePolicy
    <Kokkos::Rank<2,LoopOrder,LoopOrder>>
    ( {0, 0}, {10, 10} ), KOKKOS_LAMBDA
    (const int i, const int j){
        array2D(i,j) = 1;
    });

// increment the value in parallel
Kokkos::parallel_for(
    Kokkos::MDRangePolicy
    <Kokkos::Rank<2,LoopOrder,LoopOrder>>
    ( {0, 0}, {10, 10} ), KOKKOS_LAMBDA
    (const int i, const int j){
        array2D(i,j) += 1;
    });

// sum the values in parallel
int loc_sum;
int answer;
Kokkos::parallel_reduce(
    Kokkos::MDRangePolicy
    <Kokkos::Rank<2,LoopOrder,LoopOrder>>
    ( {0, 0}, {10, 10} ), KOKKOS_LAMBDA
    (const int i, const int j,
     int &loc_sum){
        loc_sum += array2D(i,j);
    }, answer);

```

(b) Kokkos 2D array example

Listing 12. The syntax is shown to name an array or matrix data type and to access that name.

```

// include a name in
FMatrixKokkos <double> matrix3D (10,10,10,"matrix3d");

// access th name of the array
std::str name = matrix3D.get_name();

```

3. Test Cases

In this section we present and discuss the runtime results on diverse test problems using multi-core CPUs and GPUs. The first test focuses on a pure computer science application, and calculates the average shortest distance between nodes in a Watts–Strogatz

graph using the CArrayKokkos data type and several types of parallel loops. The second test case is an ANN. This test case involves a sequence of vector-array multiplications with vastly different sizes, stressing the sparse data access and nested parallelism capabilities in MATAR. The third test case is a mini app that simulates the cooling of the Earth’s tectonic plates and assesses the parallel performance of the DynamicRaggedRightArrayKokkos data type. The last test case is a mini app that solves the isotropic 3-wave kinetic wave equations in parallel. Simulating a diverse set of test problems is key to building confidence that MATAR enables a simplified approach to achieve parallel performance and portability across CPU and GPU architectures. Furthermore, we use the graph construction code and the 3-wave kinetic wave solver code to demonstrate the performance gains of using MATAR over a similar implementation in Python to show that MATAR was designed to be discernible to C, Fortran, and Python programmers.

3.1. Computer Science: Watts–Strogatz Graph

A Watts–Strogatz (WS) [28] graph (Figure 3) is a random graph model introduced in the paper “Collective dynamics of ‘small-world’ networks” by Watts and Strogatz [28]. This graph type has many interesting properties, such as the clustering coefficient (how many three vertex triples are connected triangles). In this subsection, a WS graph is created with different parameters, and a Floyd Warshall (FW) test is run over this graph to calculate the shortest distance between any two nodes to find an average.

A WS graph is created following the process described below using parameters n , k , and p .

- Make an empty n vertex matrix.
- For each node i connect i with a directed edge to its nearest $2k$ neighbors, k is a parameter. This forms a ring lattice.
- For each edge, rewire the node connectivity with probability p to a random vertex. For practical reasons, we disallow self loops and repeated edges.

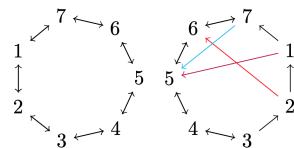


Figure 3. On the left, a ring lattice with seven nodes is shown where each node is connected to its nearest one neighbor. On the right we replace three edges with random edges. In particular, we replaced (1,2) with (1,5), (2,3) with (2,6), and (7,1) with (7,5). This is a simple demonstration of the Watts–Strogatz random graph process.

The elegance of the WS graph is that when most entities have a large number of local connections, a few “far” edges to anywhere in the network can dramatically decrease the average shortest path. This type of graph is common in a network. The relatively simple WS model can capture the behavior of many networks, which have a surprisingly small average shortest path.

3.1.1. Expected Distance Calculation

When there is no rewiring, meaning each node is connected to k nearest nodes on each side, calculating the average distance is relatively straightforward. First, the algorithm must only go, at most, half way around the ring lattice to reach any other node, so we will calculate the average distance of the $N/2$ nodes clockwise from the starting point. The first

k nodes are only one step away, the next k are two steps away, and so on. The last set of nodes will be $\text{ceiling}(\frac{N}{2k})$ steps away. The average distance is calculated using:

$$\frac{2}{N} \sum_{i=1}^{\text{ceiling}(\frac{N}{2k})} k \approx \frac{2}{N} k \frac{N(N+1)}{8k^2} \quad (1)$$

$$= \frac{N+1}{4k} \quad (2)$$

$$\approx \frac{N}{4k} \quad (3)$$

On the other extreme, if $p = 1$, then the graph looks like an Erdos–Renyi graph [29] where each node has a fixed number of edges going out from the node. The expected distance in an Erdos–Renyi graph model is logarithmic in terms of N , so we should expect something similar to logarithmic in N .

3.1.2. Testing the Watts–Strogatz Graph Code

The goal is to test the MATAR dense CArrayKokkos data type, parallel for loops (FOR_ALL), and a parallel summation (FOR_REDUCTION_SUM). The runtime will be measured with varying problem sizes. In this test case, we initialize a WS graph by assigning the local edges and then flipping a weighted coin for every edge corresponding to the rewiring probability. When the network is rewired, a local edge is replaced with a random edge. There were no safeguards put in place to make sure double edges were prevented, and in theory, this makes walks slightly longer. After initializing this graph, we run the FW algorithm to compute the shortest path length for each pair of nodes [30]. The FW algorithm is $O(n^3)$, and is shown in Listing 13. The fine-grained parallel implementation of the FW algorithm in C++ with MATAR is shown in Listing 13b.

Listing 13. The coding syntax with MATAR is intended to be readily understood by Python programmers while adhering to the C++ standard.

```
for k in range(n_nodes):
    for i in range(n_nodes):
        for j in range(n_nodes):
            if(i != j):
                dist1 = R[i,k] + R[k,j]
                dist2 = R[i,j]
                R[i,j] = min(dist1, dist2)
```

```
avg = 0
for i in range(n_nodes):
    for j in range(n_nodes):
        avg += R[i,j]
```

```
avg /= n*n
```

(a) Python coding for the FW algorithm demonstrates the similarities between the Python and MATAR coding syntax. The runtime comparisons with Python for this graph test case used the Networkx package and not the above coding.

Listing 13. *Cont.*

```

for(k = 0; k < n_nodes; k++){
    FOR_ALL(i, 0, n_nodes,
           j, 0, n_nodes, {
                if(i != j){
                    float dist1 = R(i,k) + R(k,j);
                    float dist2 = R(i,j);
                    R(i,j) = fminf(dist1, dist2);
                }
            });
}

double avg = 0;
double loc_sum; // local sum
FOR_REDUCE_SUM(i, 0, n_nodes,
              j, 0, n_nodes,
              loc_sum, {
                  loc_sum += (double) R(i,j);
              }, avg);

avg /= n*n;

```

(b) The parallel FW implementation with MATAR is shown.

3.1.3. Runtime Results for the Watts–Strogatz Graph Test

For the first timing studies, the rewiring probability is 0.0, and a node is connected to the 12 nearest nodes (i.e., the 6 nearest on each side of the node). Timing studies were performed with an AMD EPYC 7502 32-core CPU, a Nvidia Tesla V100 GPU with 32 GB of memory, a Nvidia A100 GPU, a Nvidia GTX 8000 GPU, and an AMD MI50 GPU. The A100 GPU delivers the best performance, the runtimes with the other GPUs are clustered together, followed by the parallel CPU calculations with OpenMP using 8 and 16 cores. The WS graph test case shows GPUs can deliver favorable accelerations of the runtimes compared to multi-core CPUs, and results are shown in Figure 4. The code implementation also delivers close to linear strong scaling on the multi-core CPU, see Figure 5.

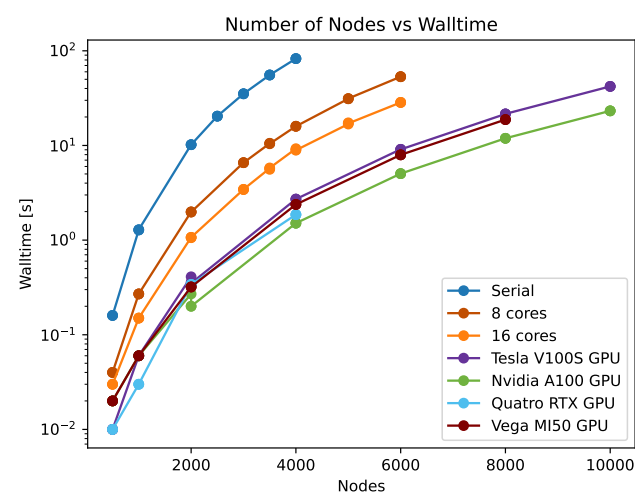


Figure 4. Comparison of runtimes, as a function of the nodes in the WS graph, on diverse GPUs and on an AMD EPYC 7502 multi-core CPU. A serial calculation (blue) is presented for comparison purposes. All GPUs deliver favorable acceleration of this test case over using 16 cores on the CPU.

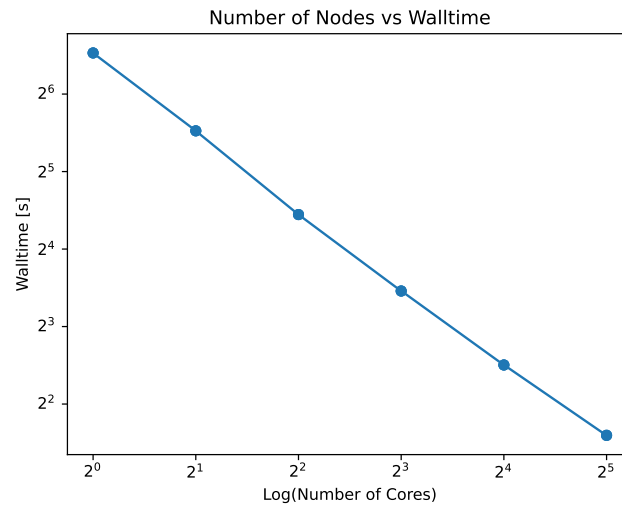


Figure 5. The strong scaling on the WS graph with 4000 nodes is shown. This scaling test was run at powers of 2^n for the number of cores from 1 to 32. Displayed is the log–log plot of number of cores versus runtime. Near perfect scaling is observed.

This WS test was run and compared against an implementation of the same problem in Python. The Python package Networkx [31] has a built-in implementation of the of the WS graph and the FW algorithm. For this comparison, the rewiring probability was changed while holding the number of nodes fixed at 4000. This accomplishes two things. First, there is a sharp decay in the average distance in the network even when there is a very low probability of edges being rewired, see Figure 6. Second, this helps verify the code written in C++ using MATAR yields the same qualitative behavior as the Python code. The accelerations of the MATAR implementation relative to the python implementation are provided in Figure 7. The MATAR-based C++ implementation is significantly faster than the Python code.

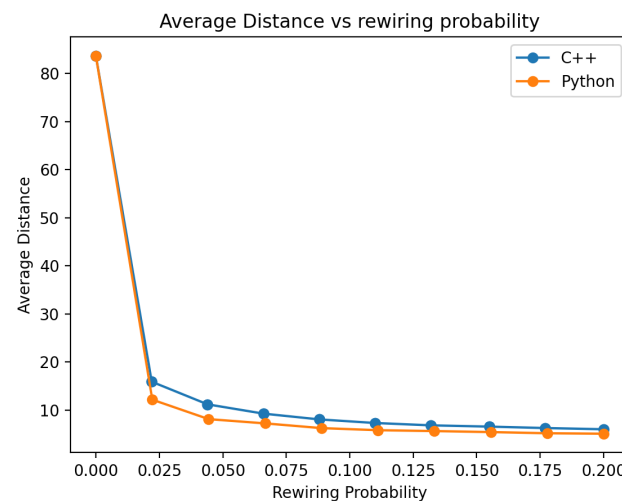


Figure 6. A comparison between the Python networkx package and the C++ implementation with MATAR is shown for the average distance between nodes as a function of the rewiring probability. The same trend is observed between the Python and C++ implementations, which is a desired outcome. Both codes yielded the expected, correct behavior for this network test case. An exact match between the two implementations is not expected since the results are probabilistic.

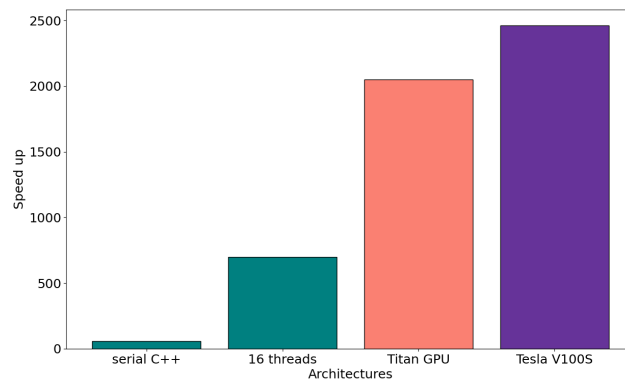


Figure 7. Compared to the serial Python code, the MATAR implementation is serially $60\times$ faster, is about $700\times$ faster using 16 threads on an AMD EPYC 7502 multi-core CPU, is about $2050\times$ faster on a Titan GPU, and is $2460\times$ faster on a V100 GPU. The WS graph speed-up results reported here correspond to a simulation using 4000 nodes.

3.2. ANN Test Case: Forward Propagation

ANNs [32] are made up of interconnected nodes, known as artificial neurons, organized into layers. The layers between the input and output layers are referred to as hidden layers. Each node's value is computed by summing its inputs, each multiplied by corresponding weights, and then passing this sum through an activation function. One common activation function is the sigmoid function, which produces an output ranging from zero to one. The propagation of an input to a node j in layer l of the ANN can be expressed as

$$y_j^l = F\left(\sum_i w_{ij}^l x_i^{l-1} + b_j^l\right), \quad (4)$$

where x_i^{l-1} is an input from the $(l-1)^{th}$ layer, w_{ij}^l is the corresponding weight for the connection between node i on the previous layer and node j , b_j^l is a bias, F is the activation function, y_j is the result at node j , and the summation is over the nodes in the previous layer $l-1$. Equation (4) is used to calculate all nodal values in the ANN. The process of passing inputs through a layer to generate outputs that serve as inputs for the subsequent layer is called forward propagation [33].

An artificial neural network was implemented with MATAR containing 5 hidden layers comprised of 30,000, 8000, 4000, 100, and 25 neurons, respectively. The sigmoidal activation function was used. This ANN received 64,000 inputs and had 6 outputs. The ANN was implemented using sparse storage, ensuring an efficient memory footprint. The vastly varying sizes of each layer allowed for the testing of vector-array sparse multiplication across large and small dimensions, testing of sparse storage, as well as quantifying the efficacy of nested parallelism. The performance portable coding for a vector-array multiplication with MATAR is shown in Listing 9.

For this runtime test case, the weights in the ANN were set equal to one and the biases were set equal to zero. To apply the ANN to a specific application, the weights and biases would be trained to fit supplied data as a function of specified inputs, and then read into the code. However, the goal of this work is to quantify the runtime performance of using the ANN code across CPU and GPU architectures using a single code implementation.

Timing studies of the forward propagation process were performed with a Haswell multi-core CPU, a Nvidia Tesla V100 GPU, a Nvidia A100 GPU, and an AMD MI50 GPU. The speed-up results using multi-core CPUs and GPUs to perform forward propagation through the ANN are shown in Figure 8. As shown, both Nvidia and AMD GPUs provide favorable accelerations compared to CPUs. The A100 GPU delivered the fastest results. Also, for performance comparison purposes, we implemented this same ANN in Python using the Keras library [34]. Using one Haswell CPU core, the serial runtime of the Python

ANN code was slightly slower, but close to the serial runtime of the MATAR written ANN code, thus allowing comparisons of the parallel performance results presented in this paper to a serial Python code implementation using the Keras library.

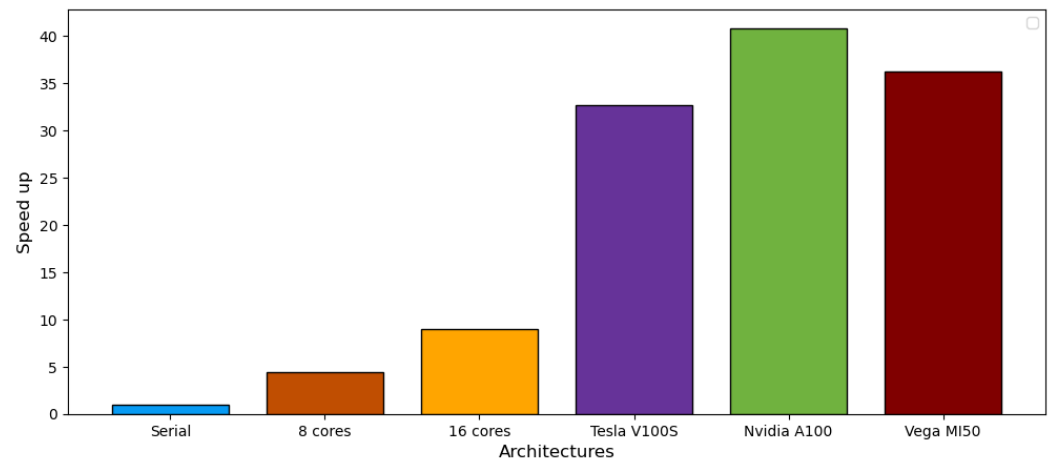


Figure 8. Speed-up compared to serial of the forward propagation through an artificial neural network using 8 cores and 16 cores on a Haswell CPU with openMP, a Nvidia Tesla V100 GPU, a Nvidia A100 GPU, and an AMD MI50 GPU. The GPU architectures deliver favorable accelerations with widely varying sizes of the vector-array multiplications.

3.3. Geophysics: Half-Space Cooling Test Case

The halfspace cooling model is used to simulate the generation and cooling of tectonic plates in the mid-ocean ridge. The model states that the oceanic lithosphere (plate) is created at a mid-ocean ridge and increases in thickness with distance from the ridge. In this example, the oceanic lithosphere is created at a fixed boundary—in this instance the start of the array—and has a temperature equal to that of the mantle (T_m). The upper boundary of the lithosphere has a depth and temperature of 0. This simplified model ignores horizontal conductive heat transport, latent heat, and compaction, and the lithosphere is allowed to cool to unlimited depth. The equation to solve for temperature as a function of depth and age is as follows from Stein and Stein [35].

$$T(t, z) = T_m \operatorname{erf}(z(4\kappa t)^{-\frac{1}{2}}) \quad (5)$$

A schematic of half-space cooling in the lithosphere is shown in Figure 9 and it is based on the diagram from Cardoso and Hamza [36]. The model follows the shape of an error function as the lithosphere increases in age and moves away from the ridge.

In this test, the temperature of the lithosphere as a function of depth and age is populated into a `DynamicRaggedRightArrayKokkos`, see Listing 14. The array is first given a buffer size that is dependent on total problem size, i.e., the maximum age supplied by the user. The array will be populated outward in each row, increasing in stride, until the temperature of the lithosphere has reached the mantle temperature and then moves on to the next age.

Runtime Results for the Half-Space Cooling Test

This test was conducted in serial and in parallel on a Haswell multi-core CPU and four kinds of GPUs (Nvidia Tesla V100, Nvidia A100, and Quadro RTX 8000, and MI50) with an increasing size of the problem. In this case, increasing the age of the plate that the cooling is computed for, and the y axis is the total walltime in seconds. As shown in Figure 10, we find that the serial and parallel cores scale almost linearly with increasing problem size while the GPUs have constant runtimes with increasing problem size. Between GPU architectures, the Quadro RTX 8000 performs the best across all problem sizes, while the

MI50 (a prior generation AMD GPU) is slightly slower than the other GPUs. Yet, the MI50 GPU still delivers notable accelerations over the multi-core CPU simulations.

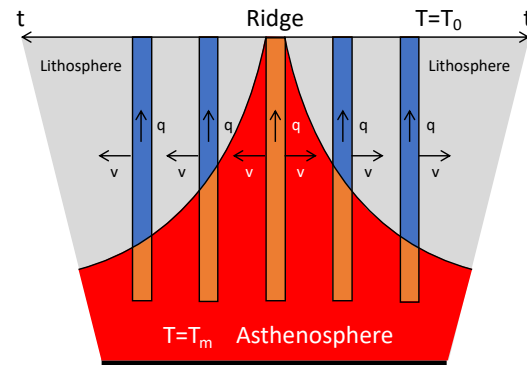


Figure 9. Diagram of the mid-ocean ridge showing oceanic lithosphere increasing in thickness as it increases in distance from the ridge. Gray is the oceanic lithosphere, and red is the asthenosphere (mantle).

Listing 14. Coding used in the half-space cooling test case is shown. The MATAR library offers dynamic ragged data types that are key to efficient code implementations.

```
DynamicRaggedRightArrayKokkos <double> dyn_ragged_right (max_age+1, depth+1);
DO_ALL(i, 0, max_age, {
    for (int j = 0; j <= depth; j++) {
        if (i == 0 && j == 0){
            dyn_ragged_right.stride(i)++;
            dyn_ragged_right(i, j) = mantle_temp;
        }
        double temp = mantle_temp * erf(j/(2.0*sqrt(thermal_diff*(i*1e6))));
        dyn_ragged_right.stride(i)++;
        dyn_ragged_right(i, j) = temp;

        // check if we have reached the mantle
        if (round(dyn_ragged_right(i, j)) == 1350) break;
    }
});
```

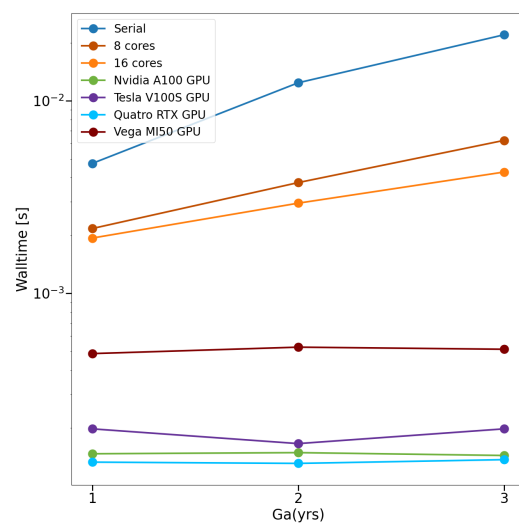


Figure 10. Scaling plot for the half-space cooling test problem showing walltime in seconds for serial, 8 cores, and 16 cores on a Haswell CPU; and for Nvidia Tesla V100 GPU, Nvidia A100 GPU, Quadro RTX GPU, and AMD MI50 GPU architectures with increasing problem size (e.g., billions of years). GPU architectures significantly accelerate the calculation compared to the Haswell CPU.

3.4. Mathematical Physics: Isotropic 3-Wave Kinetic Equation

Wave turbulence theory (WTT) has its origins in the early to mid part of the last century with the work of Peierls, Benney-Saffman, Hasselman, Benney-Newell, Zakharov and many others [37–41]. In contrast to hydrodynamic turbulence, wave turbulence describes interactions among waves that lack vorticity. An example we are all familiar with is that of a wind-driven sea. This scenario was considered by Klaus Hasselmann, the recent Nobel laureate, in 1962 [39] wherein one of the first formal derivations of a wave kinetic equation (WKE) was given [39,42]. The intent was to describe the transfer of energy among waves on the open ocean; that is, if we treat the waves like particles and analyse their collisions then what is the behavior of the energy? This is analogous to considering the distribution of particles with a certain velocity, described by the Boltzmann equation. Later, Zakharov and Filonenko 1967 [41] were able to derive analytically quasi-stationary solutions to the wave kinetic equation describing capillary wave systems. These solutions are now called the Kolmogorov–Zakharov solutions or K-Z spectra and have been found for many other systems [43]. These solutions are analogous to the Kolmogorov 5/3 law of strong turbulence, with the important difference that the exponent of these power law solutions depends on the system under consideration. The K-Z spectra describe the energy flux from low to high wavenumbers (direct cascade) and from high to low wavenumbers (inverse cascade). Wave turbulence remains an active area of research. Its predictions have been verified numerically and experimentally [44,45], and applications have been found in weather forecasting and climate modeling, to name a few examples.

Beyond stationary solutions, we can consider the evolution of the energy density in the time dynamic case. For 3-wave kinetic equations (3-WKE), a rigorous study of the longtime behavior of solutions was carried out by Soffer and Tran [46]. To carry out a numerical test of these longtime solutions, we consider the following conservative form of the forward cascade term in the isotropic 3-WKE originally derived in [47]

$$\begin{aligned} \partial_t f(t, p) &= \partial_p \mathcal{Q}_3[f](t, p) \quad (t, p) \in \mathbb{R}^+ \times \mathbb{R}^+ \\ f(0, p) &= f_0(p), \end{aligned} \tag{6}$$

where the collision term is given by

$$\begin{aligned} \mathcal{Q}_3[f](t, p) &= -2 \int_0^p \int_0^p V_{1,2} f_1 f_2 \chi\{p < p_1 + p_2\} dp_{21} \\ &+ \int_0^\infty \int_0^\infty V_{1,2} f_1 f_2 \chi\{p < p_1 + p_2\} dp_{21}, \end{aligned} \tag{7}$$

with $V_{1,2} = (p_1 p_2)^{\gamma/2}$ for $\gamma \in [1, 2]$ the degree of of the interaction coefficient, $V_{1,2}$, and $\chi\{A\}$ denotes the characteristic function of the set A . To simplify the collision term we use $f_i = f(t, p_i)$ for $i = 1, 2$ and $dp_{21} = dp_2 dp_1$. Equation (6) with the range of specified values for γ is appropriate for acoustic and capillary wave systems and provides a statistical description of triadic wave collisions and energy transfer. The quantity $f(t, p)$ is often referred to as the wave-action and can be thought of as the “occupation” number of waves with wavenumber p . The 3-WKE is a principle object in the theory of weak wave turbulence (WWT) and for a more detailed introduction, from the physicists perspective, the reader is referred to the works [43,48] and the many references therein.

In the works [47,49], a finite volume scheme and deep learning algorithm were derived to solve Equation (6), respectively. Rather than solving for the wave-action, which is not conserved, the authors reformulated Equation (6) to solve for the energy density $g(t, p) = pf(t, p)$, which is conserved. It was proved rigorously in [46] that the energy comprises two parts: the energy concentrated at the point $p = \infty$ and the energy everywhere else, which we denote here by $g_{\{\infty\}}$ and $g_{[0,\infty)}$, respectively. They showed that the total energy evacuates any finite interval with decay rate $\sim \mathcal{O}(t^{-1/2})$ and accumulates at the point $p = \infty$. This phenomenon is analogous to the well-known run-away particle growth or gelation behavior of the Smoluchowski coagulation equation, which may be considered

a special case of the 3-WKE. These theoretical results, in the absence of analytic solutions, were used to validate the numerics in [47,49].

The finite volume approximation of $g(t, p)$ reads (see [47] for more details)

$$g^{n+1}(p_i) = g^n(p_i) + \lambda_i \left(Q_{i+1/2}^n \left[\frac{g}{p} \right] - Q_{i-1/2}^n \left[\frac{g}{p} \right] \right), \tag{8}$$

where $\lambda_i = \frac{p_i \Delta t}{h_i}$ and $h_i = \Delta p_i$, so that

$$\partial_p Q_3 \left[\frac{g}{p} \right] (t, p) \approx \frac{1}{h_i} \left(Q_{i+1/2}^n \left[\frac{g}{p} \right] - Q_{i-1/2}^n \left[\frac{g}{p} \right] \right), \tag{9}$$

is a suitable approximation of the flux collision term (see [47] for more details). For simplicity, we show a forward-Euler time update but use an explicit second-order Runge–Kutta scheme in practice.

The 3-WKE is a highly non-linear, non-local integral differential equation and thus a straightforward implementation of the finite volume scheme (8) can be very computationally expensive. What is more, a sufficient condition on the time step Δt for stability and positivity of solutions, that is physically relevant solutions, was derived in [47]. This CFL condition was shown to depend on the initial energy and can be quite restrictive for initial conditions that contain large amounts of energy.

Here, we compare a serial implementation in Python and an implementation using MATAR with the CUDA backend in Kokkos. It is worth emphasizing that the most significant changes to the code when going from one implementation to the other is changing the standard Python for loop to the MATAR syntax detailed in Listing 5.

We give computation times for each implementation on a simple test. We start with the initial condition $g_0(p) = 1.26157e^{-50(p-1.5)^2}$, for $p \in [0, 100]$, with $h = 0.5, 0.2, 0.1$ keeping $\Delta t = 0.05$ as the wavenumber resolution is refined. The serial simulations in Python were performed on a Macbook Pro with a 2.4 GHz 8-Core Intel Core i9 processor and 32 GB 2667 MHz DDR4 memory. To assess the performance of MATAR with the CUDA backend in Kokkos, computations were performed on an NVIDIA Tesla V100 GPU. The results are shown in Figure 11 for different resolutions. We see that with minor syntactical changes to the code, we are able to attain great gains in terms of speed-up.

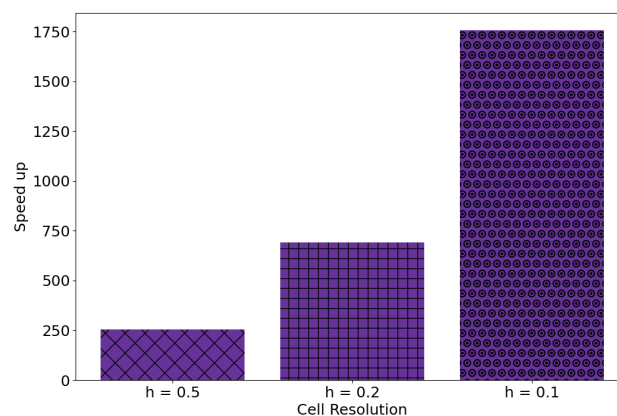


Figure 11. Speed-up with mesh refinement provided by MATAR with CUDA over the serial Python code using an V100 GPU is shown. 253X, 69X, and 1756X accelerations are observed.

4. Conclusions

This paper presented a novel approach with the C++ MATAR library to incorporate fine-grained parallelism within a code and readily write portable software to take advantage of diverse computer architectures (both multi-core CPUs and GPUs). The approach leverages the Kokkos library for performance portability, supplies simple-to-use interfaces to support fine-grained parallelism, and supports a wide range of dense and sparse data

representations (both static and dynamic) for writing new C++ codes or modernizing existing Fortran codes.

The paper described the programming syntax with MATAR to use the diverse data types and to write parallel loops. The programming syntax is designed to be discernible to C, Python, and Fortran programmers while adhering to the C++ standard. Furthermore, the programming syntax for fine-grained parallelism is akin to normal serial programming, which is key to ensuring the readability of a code, aiding the adoption of the Kokkos library by a broad audience, and accelerating code development. A suite of examples were presented in the paper on how to implement routines that run in parallel with MATAR. Likewise, side-by-side comparisons with Python, Fortran, and C++ with MATAR were presented.

A suite of tests were run to demonstrate the merits of using MATAR on diverse applications covering: graph construction, forward propagation of inputs through an artificial neural network, heat transfer, and solving the isotropic 3-wave kinetic equations, respectively. These applications greatly differed from each other, which is key to showing MATAR delivers performance portability using a straightforward coding syntax. For the graph test case, nearly perfect strong scaling was achieved on an AMD multi-core CPU; in addition, a V100 GPU could deliver $2460\times$ acceleration over a comparable Python implementation run serially. For the artificial neural network test case, the MI50, V100 GPU and A100 GPU all delivered favorable accelerations compared to a serial calculation on a Haswell CPU. The artificial neural network had vastly varying network sizes, descending from 64,000 inputs to 6 outputs across 5 hidden layers. This test case quantified the performance of nested parallelism, sparse data storage, and vector-array multiplication. For the heat transfer problem, the MI50, V100 GPU, A100 GPU, and RTX GPU all delivered orders of magnitude acceleration compared to a serial calculation on a Haswell CPU. This heat transfer test problem used a dynamic ragged right data type in MATAR. For the case of solving the 3-wave kinetic equations, the code implementation with MATAR was run on a V100 GPU with mesh resolutions of $h = 0.5, 0.2,$ and 0.1 . This test demonstrated $253\times,$ $692\times,$ and $1756\times$ accelerations (for the respective mesh resolutions) over a similar Python implementation run serially.

The primary conclusion of this work is that the MATAR library provides a straightforward solution to achieve parallel performance and portability across CPU and GPU architectures, making it beneficial for a wide range of applications.

Author Contributions: Conceptualization, N.M., C.Y., A.D., D.D., J.M., E.H., C.R., E.L., S.W., S.B., D.H., M.K., G.W., Z.B. and R.R.; methodology, N.M., C.Y., A.D., D.D., J.M., E.H., C.R., E.L., S.W., S.B., D.H., M.K., G.W., Z.B. and R.R.; software, N.M., C.Y., A.D., D.D., J.M., E.H., C.R., E.L., S.W., S.B., G.W., Z.B. and R.R.; validation, N.M., C.Y., A.D., D.D., J.M., E.H., C.R., E.L., S.W., S.B., G.W., Z.B. and R.R.; formal analysis, N.M., C.Y., A.D., D.D., J.M., E.H., C.R., E.L., S.W., S.B., G.W., Z.B. and R.R.; investigation, N.M., C.Y., A.D., D.D., J.M., E.H., C.R., E.L., S.W., S.B., G.W., Z.B. and R.R.; resources, N.M.; data curation, N.M., C.Y., A.D., D.D., J.M., E.H., C.R., E.L., S.W., S.B., G.W., Z.B. and R.R.; writing—original draft preparation, N.M., C.Y., A.D., D.D., J.M., E.H., C.R., E.L., S.W., S.B., M.K. and G.W.; writing—review and editing, N.M., C.Y., A.D., D.D., J.M., E.L., S.W., S.B. and G.W.; visualization, N.M., C.Y., A.D., D.D., J.M., E.H., C.R., E.L., S.W., S.B., M.K., G.W. and Z.B.; supervision, N.M., M.K. and R.R.; project administration, N.M., M.K. and R.R.; funding acquisition, N.M. All authors have read and agreed to the published version of the manuscript.

Funding: Los Alamos National Laboratory (LANL) is operated by Triad National Security, LLC for the U.S. Department of Energy's NNSA under contract number 89233218CNA000001.

Data Availability Statement: Dataset available on request from the authors.

Acknowledgments: We gratefully acknowledge the support of the Laboratory Directed Research and Development (LDRD) program at LANL. The Advanced Simulation and Computing (ASC) program also supported code work in the Fierro mechanics code and the MATAR library. This research used resources provided by the Darwin testbed at LANL, which is supported by the

Computational Systems and Software Environments subprogram of LANL's ASC program. The Los Alamos unlimited release number is LA-UR-22-20105.

Conflicts of Interest: The authors declare no conflicts of interest.

References

- Sicard, E.; Trojman, L. Introducing 2-nm/20 Å Nano-Sheet FET Technology with Buried Power Rails and Nano Through-Silicon-Vias in Microwind. Ph.D. Thesis, INSA Toulouse, Toulouse, France, 2022.
- Chen, T.; Moreau, T.; Jiang, Z.; Zheng, L.; Yan, E.; Shen, H.; Cowan, M.; Wang, L.; Hu, Y.; Ceze, L.; et al. TVM: An automated End-to-End optimizing compiler for deep learning. In Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18), Carlsbad, CA, USA, 8–10 October 2018; pp. 578–594.
- Haidl, M.; Gorchach, S. PACXX: Towards a unified programming model for programming accelerators using C++ 14. In Proceedings of the 2014 LLVM Compiler Infrastructure in HPC, New Orleans, LA, USA, 17 November 2014; pp. 1–11.
- Zheng, L.; Jia, C.; Sun, M.; Wu, Z.; Yu, C.H.; Haj-Ali, A.; Wang, Y.; Yang, J.; Zhuo, D.; Sen, K.; et al. Anso: Generating High-Performance tensor programs for deep learning. In Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20), Virtual Event, 4–6 November 2020; pp. 863–879.
- Rasch, A.; Schulze, R.; Steuwer, M.; Gorchach, S. Efficient auto-tuning of parallel programs with interdependent tuning parameters via auto-tuning framework (ATF). *ACM Trans. Archit. Code Optim. (TACO)* **2021**, *18*, 1–26. [[CrossRef](#)]
- Edwards, H.C.; Trott, C.; Sunderland, D. Kokkos. *J. Parallel Distrib. Comput.* **2014**, *74*, 3202–3216. [[CrossRef](#)]
- Beckingsale, D.A.; Burmark, J.; Hornung, R.; Jones, H.; Killian, W.; Kunen, A.J.; Pearce, O.; Robinson, P.; Ryujin, B.S.; Scogland, T.R. RAJA: Portable performance for large-scale scientific applications. In Proceedings of the 2019 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC), Denver, CO, USA, 22 November 2019; pp. 71–81.
- Arndt, D.; Lebrun-Grandie, D.; Trott, C. Experiences with implementing Kokkos' SYCL backend. In Proceedings of the 12th International Workshop on OpenCL and SYCL, Chicago, IL, USA, 8–11 April 2024; pp. 1–11.
- Steuwer, M.; Remmelg, T.; Dubach, C. Lift: A functional data-parallel IR for high-performance GPU code generation. In Proceedings of the 2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO), Austin, TX, USA, 4–8 February 2017; pp. 74–85.
- Dunning, D.J.; Morgan, N.R.; Moore, J.L.; Nelluvelil, E.; Tafolla, T.V.; Robey, R.W. MATAR: A Performance Portability and Productivity Implementation of Data-Oriented Design with Kokkos. *J. Parallel Distrib. Comput.* **2021**, *157*, 86–104. [[CrossRef](#)]
- Rajamanickam, S.; Acer, S.; Berger-Vergiat, L.; Dang, V.; Ellingwood, N.; Harvey, E.; Kelley, B.; Trott, C.R.; Wilke, J.; Yamazaki, I. Kokkos kernels: Performance portable sparse/dense linear algebra and graph kernels. *arXiv* **2021**, arXiv:2103.11991.
- Yenusah, C.; Morgan, N.; Robey, R.; Stone, T.; Liu, Y.; Chen, L. Incorporating performance portability and data-oriented design in phase-field modeling. In Proceedings of the ASME 2022 International Design Engineering Technical Conferences and Computers and Information in Engineering Conference IDETC/CIE2022, St. Louis, MO, USA, 14–17 August 2022.
- Yenusah, C.O.; Morgan, N.R.; Lebensohn, R.A.; Zecevic, M.; Knezevic, M. A parallel and performance portable implementation of a full-field crystal plasticity model. *Comput. Phys. Commun.* **2024**, *300*, 109190. [[CrossRef](#)]
- Morgan, N.; Moore, J.; Brown, S.; Chiravalle, V.; Diaz, A.; Dunning, D.; Lieberman, E.; Walton, S.; Welsh, K.; Yenusah, C.; et al. Fierro. 2021. Available online: <https://github.com/LANL/Fierro> (accessed on 5 October 2024).
- Diaz, A.; Morgan, N.; Bernardin, J. A parallel multi-constraint topology optimization solver. In Proceedings of the ASME 2022 International Design Engineering Technical Conferences and Computers and Information in Engineering Conference IDETC/CIE2022, St. Louis, MO, USA, 14–17 August 2022.
- Diaz, A.; Morgan, N.; Bernardin, J. Parallel 3D topology optimization with multiple constraints and objectives. *Optim. Eng.* **2023**, *25*, 1531–1557. [[CrossRef](#)]
- Chiravalle, V.; Morgan, N. A 3D finite element ALE method using an approximate Riemann solution. *Int. J. Numer. Methods Fluids* **2016**, *83*, 642–663. [[CrossRef](#)]
- Burton, D.; Carney, T.; Morgan, N.; Sambasivan, S.; Shashkov, M. A Cell Centered Lagrangian Godunov-like method of solid dynamics. *Comput. Fluids* **2013**, *83*, 33–47. [[CrossRef](#)]
- Liu, X.; Morgan, N.; Burton, D. A high-order Lagrangian discontinuous Galerkin hydrodynamic method for quadratic cells using a subcell mesh stabilization scheme. *J. Comput. Phys.* **2019**, *386*, 110–157. [[CrossRef](#)]
- Liu, X.; Morgan, N.R.; Lieberman, E.J.; Burton, D.E. A fourth-order Lagrangian discontinuous Galerkin method using a hierarchical orthogonal basis on curvilinear grids. *J. Comput. Appl. Math.* **2022**, *404*, 113890. [[CrossRef](#)]
- Lieberman, E.; Liu, X.; Morgan, N.; Luscher, D.J.; Burton, D. A higher-order Lagrangian discontinuous Galerkin hydrodynamic method for solid dynamics. *Comput. Methods Appl. Mech. Eng.* **2019**, *353*, 467–490. [[CrossRef](#)]
- Lieberman, E.J.; Liu, X.; Morgan, N.R.; Burton, D.E. A multiphase Lagrangian discontinuous Galerkin hydrodynamic method for high-explosive detonation physics. *Appl. Eng. Sci.* **2020**, *4*, 100022. [[CrossRef](#)]
- Abgrall, R.; Lipnikov, K.; Morgan, N.; Tokareva, S. Multidimensional staggered grid residual distribution scheme for Lagrangian hydrodynamics. *SIAM J. Sci. Comput.* **2020**, *42*, A343–A370. [[CrossRef](#)]
- Moore, J.; Morgan, N.; Horstemeyer, M. ELEMENTS: A high-order finite element library in C++. *SoftwareX* **2019**, *10*, 100257. [[CrossRef](#)]

25. Morgan, N.; Moore, J.; Kiviahio, J.; Diaz, A. A 3D arbitrary-order element mesh library to support diverse numerical methods. In Proceedings of the ASME 2022 International Design Engineering Technical Conferences and Computers and Information in Engineering Conference IDETC/CIE2022, St. Louis, MO, USA, 14–17 August 2022.
26. Zecevic, M.; Lebensohn, R.; Rogers, M.; Moore, J.; Chiravalle, V.; Lieberman, E.; Dunning, D.; Shipman, G.; Knezevic, M.; Morgan, N. Viscoplastic self-consistent formulation as generalized material model for solid mechanics applications. *Appl. Eng. Sci.* **2021**, *6*, 100040. [[CrossRef](#)]
27. Zecevic, M.; Lebensohn, R.A.; Capolungo, L. New large-strain FFT-based formulation and its application to model strain localization in nano-metallic laminates and other strongly anisotropic crystalline materials. *Mech. Mater.* **2022**, *166*, 104208. [[CrossRef](#)]
28. Watts, D.J.; Strogatz, S.H. Collective dynamics of ‘small-world’ networks. *Nature* **1998**, *393*, 440–442. [[CrossRef](#)]
29. Erdos, P.; Rényi, A. On the evolution of random graphs. *Publ. Math. Inst. Hung. Acad. Sci.* **1960**, *5*, 17–60.
30. Floyd, R.W. Algorithm 97: Shortest path. *Commun. ACM* **1962**, *5*, 345. [[CrossRef](#)]
31. Hagberg, A.A.; Schult, D.A.; Swart, P.J. Exploring Network Structure, Dynamics, and Function using NetworkX. In Proceedings of the 7th Python in Science Conference, Pasadena, CA, USA, 19–24 August 2008; Varoquaux, G., Vaught, T., Millman, J., Eds.; Los Alamos National Laboratory (LANL): Los Alamos, NM, USA, 2008; pp. 11–15.
32. McCulloch, W.S.; Pitts, W. A logical calculus of the ideas immanent in nervous activity. *Bull. Math. Biophys.* **1943**, *5*, 115–133. [[CrossRef](#)]
33. Drori, I. *The Science of Deep Learning*; Cambridge University Press: Cambridge, UK, 2022. Available online: <http://www.dlbook.org> (accessed on 5 October 2024).
34. Chollet, F. And Others Keras. 2015. Available online: <https://keras.io> (accessed on 5 October 2024)
35. Stein, C.A.; Stein, S. A model for the global variation in oceanic depth and heat flow with lithospheric age. *Nature* **1992**, *359*, 123–129. [[CrossRef](#)]
36. Cardoso, R.R.; Hamza, V.M. Finite half space model of oceanic lithosphere. In *Horizons in Earth Science Research*; Veress, B., Szigethy, J., Eds.; Nova Science Publishers, Inc.: Hauppauge, NY, USA, 2011; Volume 11, pp. 375–395.
37. Peierls, R. Zur kinetischen Theorie der varmeleitung in kristallen. *Ann. Phys.* **1929**, *395*, 1055–1101. [[CrossRef](#)]
38. Benney, D.J.; Saffman, P.G. Nonlinear interactions of random waves in a dispersive medium. *Proc. R. Soc. Lond. A* **1966**, *289*, 301–320.
39. Hasselmann, K. On the non-linear energy transfer in a gravity-wave spectrum Part 1. General theory. *J. Fluid Mech.* **1962**, *12*, 481–500. [[CrossRef](#)]
40. Benney, D.J.; Newell, A.C. Random wave closures. *Stud. Appl. Math.* **1969**, *48*, 29–53. [[CrossRef](#)]
41. Zakharov, V.E.; Filonenko, N.N. Weak turbulence of capillary waves. *J. Appl. Mech. Tech. Phys.* **1967**, *8*, 37–40. [[CrossRef](#)]
42. Hasselmann, K. On the spectral dissipation of ocean waves due to white capping. *Bound.-Layer Meteorol.* **1974**, *6*, 107–127. [[CrossRef](#)]
43. Nazarenko, S. Wave Turbulence. In *Lecture Notes in Physics*; Springer: Berlin/Heidelberg, Germany, 2011; Volume 825, p. xvi+279.
44. Falcon, E.; Mordant, N. Experiments in Surface Gravity–Capillary Wave Turbulence. *Annu. Rev. Fluid Mech.* **2022**, *54*, 1–25. [[CrossRef](#)]
45. Kochurin, E.; Ricard, G.; Zubarev, N.; Falcon, E. Three-dimensional direct numerical simulation of free-surface magnetohydrodynamic wave turbulence. *Phys. Rev. E* **2022**, *105*, L063101. [[CrossRef](#)]
46. Soffer, A.; Tran, M.B. On the energy cascade of 3-wave kinetic equations: Beyond Kolmogorov–Zakharov solutions. *Commun. Math. Phys.* **2020**, *376*, 2229–2276. [[CrossRef](#)]
47. Walton, S.; Tran, M.B. A numerical scheme for wave turbulence: 3-wave kinetic equations. *SIAM J. Sci. Comput.* **2023**, *45*, B467–B492. [[CrossRef](#)]
48. Galtier, S. *Physics of Wave Turbulence*; Cambridge University Press: Cambridge, UK, 2022.
49. Walton, S.; Tran, M.B.; Bensoussan, A. A deep learning approximation of non-stationary solutions to wave kinetic equations. *Appl. Numer. Math.* **2022**, *199*, 213–226. [[CrossRef](#)]

Disclaimer/Publisher’s Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.