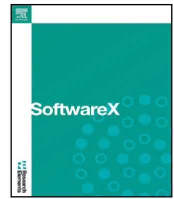
Contents lists available at ScienceDirect

SoftwareX

journal homepage: www.elsevier.com/locate/softx



Original software publication

Morpheus: A library for efficient runtime switching of sparse matrix storage formats

Christodoulos Stylianou *, Michèle Weiland

EPCC, The University of Edinburgh, Edinburgh, United Kingdom

ARTICLE INFO

Keywords:
Sparse matrix storage formats
Generic programming
Dynamic matrices
Performance portability
Productivity

ABSTRACT

Sparse matrix storage formats have evolved over the years to better exploit the particular strengths of different hardware architectures or to better match the sparsity patterns of matrices, with the aim to optimize operations on the matrices. However, the integration of new formats in existing source code is an invasive procedure that often requires a complete re-writing of the code. *Morpheus* introduces a framework that abstracts the notion of the different formats in order to optimize the performance of the sparse operations and increase the user's productivity by seamlessly matching the underlying data-structure to the computation at runtime, with minimal overheads.

Code metadata

Current code version	v1.0.0
Permanent link to code/repository used for this code version	https://github.com/ElsevierSoftwareX/SOFTX-D-23-00078
Code Ocean compute capsule	N/A
Legal Code License	Apache License, Version 2.0
Code versioning system used	git
Software code languages, tools, and services used	C++, Kokkos, OpenMP, Cuda, HIP
Compilation requirements, operating environments & dependencies	CMake 3.16+, C++17 compiler, Linux, GoogleTest
If available Link to developer documentation/manual	https://morpheus-org.github.io/morpheus
Support email for questions	c.stylianou@ed.ac.uk

1. Motivation and significance

Sparse matrix computations are a key component of many performance critical numerical simulations. A desire to represent sparse matrices efficiently in memory has led to the development of a plethora of sparse matrix storage formats, in particular given the evolution of hardware architectures. Each format is designed to exploit the particular strengths of an architecture or the specific sparsity pattern of a matrix. Filippone et al. [1] show that using an appropriate storage format for an operation can have significant performance gain over a general-purpose format.

Morpheus [2] is designed to enable efficient and transparent runtime-switching of sparse matrix storage formats across multiple backends. The library provides an abstraction for sparse matrices that can dynamically adapt the underlying sparse matrix data-structure to better suit an operation, target architecture and sparsity pattern of a matrix. The adoption of such an abstraction by developers and users allows them to focus on their scientific endeavour without the need to understand the specifics of each supported sparse matrix storage format. *Morpheus* enables them to develop efficient and performance portable code that is not tied to any particular storage format, using a single source code and thus eliminating the need for testing and maintaining multiple code bases. New optimization opportunities emerge through the option of switching to different storage formats at runtime. The simple design of *Morpheus* allows for the straightforward addition of new formats over time. Users can take advantage of such new formats without requiring any changes to their own code bases, allowing them to efficiently represent a wider range of sparsity patterns and thus increasing the lifetime and performance of their software.

Several other approaches specific to the optimization of sparse linear algebra through switching and selection of the most suitable

* Corresponding author.
E-mail addresses: c.stylianou@ed.ac.uk (Christodoulos Stylianou), m.weiland@epcc.ed.ac.uk (Michèle Weiland).

https://doi.org/10.1016/j.softx.2024.101775
Received 1 February 2023; Received in revised form 1 May 2024; Accepted 21 May 2024
Available online 29 May 2024
2352-7110/© 2024 The Authors. Published by Elsevier B.V. This is an open access article under the CC BY license (http://creativecommons.org/licenses/by/4.0/).

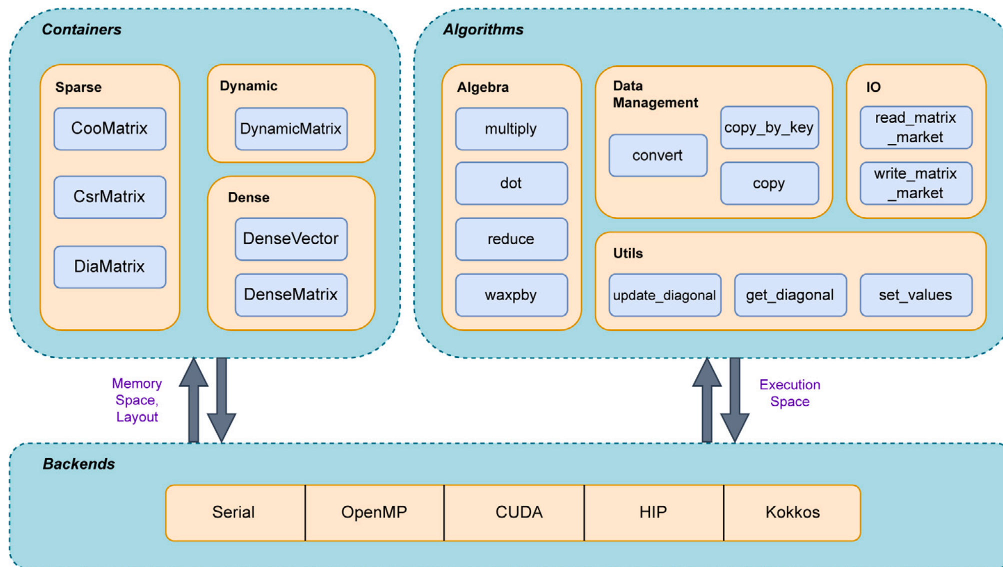


Fig. 1. High-level overview of *Morpheus* showing the major components of the library.

storage format also exist, each offering something different to the users. Filippone et al. [3] provide an object oriented design model for a sparse linear algebra package which relies on Design Patterns [4] and *Fortran* specific features in order to achieve efficient dynamic switching, but lacking support for multiple backends. *clSpMV* [5] provides a rich set of formats through a product type that can be used in a heterogeneous environment. However, the design of the framework prevents a straightforward and type-safe addition of new formats. *SMATER* [6] provides users with a unified programming interface that uses a single sparse format (Compressed Sparse Row (CSR)) at the interface level and automatically determines the optimal format and Sparse Matrix-Vector Multiplication (SpMV) implementation for any input sparse matrix at runtime. *GINKGO* [7] is a high-performance sparse linear algebra library for many-core systems that abstracts all functionality as linear operators in an object-oriented design. *Morpheus* also offers a single abstraction for sparse matrices, ensuring a simple and intuitive interface through a *functional* design. Compared to GINKGO, *Morpheus* focuses on operations that can be used as building blocks to sparse linear algebra libraries, such as SpMV, rather than offering complete solvers and provides a simple mechanism for switching across the available formats without the need of code modifications. In addition, the adoption of abstractions regarding hardware platforms and memory hierarchies ensures an extensible code base that can adapt to any new hardware introduced in the future. *Morpheus* preserves value semantics across the different data structures and algorithms, ensuring type-safety and reducing runtime errors.

2. Software description

Morpheus is a *C++17* header-only template library. It follows a functional design that separates the data structures (containers) from the functions (algorithms). Algorithms act on containers, and for each backend that is available a version of the algorithm exists. Conceptually, there are two layers of functionality that balance performance and flexibility:

1. Compile-time Layer: Deals explicitly with the different sparse matrix storage formats and their corresponding algorithms at compile-time resulting in zero runtime overheads.
2. Runtime Layer: Built on top of the compile-time layer. It is responsible for enabling the dynamic functionality of *Morpheus*. This layer needs to be lightweight and defer only computationally inexpensive operations to the runtime in order to not introduce prohibitive overheads.

Table 1
Compile-time parameters of each container. Only *ValueType* is required by the user and sensible defaults are selected by *Morpheus* if any of the rest is ommited.

Parameter	Description	Valid Type
ValueType	Type of values held by the container	Arithmetic type (Required)
IndexType	Type of indices held by the container	Integral type
Layout	The ordering of data in 1D memory storage	*Column-Major* or *Row-Major*
Space	The memory space (as defined by *Kokkos*)	HostSpace, Cuda/HIPSpace

To reduce the learning curve for dealing with sparse formats and operations, *Morpheus* is heavily based on templates and meta-programming techniques (discussed in more detail below) such that both layers expose the same unified API to the user.

2.1. Software architecture

A high-level overview of *Morpheus*'s design is shown in Fig. 1, which illustrates how the different components of the library are organized. In the following sections, each component will be introduced in turn. Note that in order to provide support for the various hardware platforms and memory hierarchies, *Morpheus* adopts two notions of abstraction introduced by *Kokkos* [8,9]:

1. the *Execution Space*, which specifies where code will be *executed*;
2. the *Memory Space*, which specifies where the data will *reside in memory*.

2.1.1. Containers

All the data structures supported by *Morpheus* are implemented individually as containers. Containers are responsible for acquiring and releasing their own resources following the principle of Resource Acquisition Is Initialization (RAII) [10]. Each container is uniformly parameterized by a set of compile-time template parameters described in Table 1. Currently, *Morpheus* supports three sparse, two dense and one dynamic container as shown in Table 2. Note that the *DynamicMatrix* container supported can represent any of the available sparse matrix containers and its functionality is discussed further in Section 2.2.4.

Table 2
Containers currently supported in *Morpheus* divided into their logical categories and their corresponding format.

Category	Container	Format
Sparse	*CooMatrix* *CsrMatrix* *DiaMatrix*	Coordinate (COO) CSR Diagonal (DIA)
Dense	*DenseVector* *DenseMatrix*	1-D array 2-D array
Dynamic	*DynamicMatrix*	Dynamic sparse matrix

Table 3
Different types of data management mechanisms with their associated requirements and overheads. Note that two containers have compatible type if they both have the same parameters as shown in Table 1.

Requirements	Shallow Copy	Deep Copy	Convert
Compatible Types	✓	✓	×
Same Space	✓	×	✓
Same Format	✓	✓	×
Overhead	Low	Medium	High

2.1.2. Algorithms

The different algorithms available in *Morpheus* are conceptually divided into four categories:

1. *Algebra:* Common linear algebra operations for dense, sparse and dynamic containers, with focus on the SpMV multiplication.
2. *Data Management:* Data management routines for copying containers between memory spaces and converting from one container to another.
3. *Input-Output: IO* operations such as reading/writing a file from/to disk. Currently, only operations that read/write sparse matrices in Matrix Market (MM) [11] file-format are supported.
4. *Utilities:* Routines for modifying the elements of the containers, such as updating the diagonal or values of a matrix.

2.1.3. Backends

Each execution space supported by *Morpheus* constitutes a separate backend. Currently, all algorithms in *Morpheus* can have generic and/or custom versions, depending on the backend. Custom algorithms exist for four backends: (1) *Serial* (Sequential), (2) *OpenMP* (Multi-threaded), (3) *CUDA* (NVIDIA GPUs) and (4) *HIP* (AMD GPUs). In addition, the generic backend uses *Kokkos* for generic performance portable kernels capable of targeting all major HPC platforms.

Algorithms that use the sparse containers are required to provide at least one implementation per backend and storage format. As a result, the development and maintenance effort grows drastically as more formats and custom backends are introduced. The adoption of a generic backend mitigates this issue by creating a single performance portable source code for each format capable of running across platforms. In the case where optimizations specific to a particular architecture and storage format are desired, a custom backend can be used instead, balancing development productivity with code performance.

2.2. Software functionalities

2.2.1. Data management

Morpheus has multiple mechanisms for dealing with data management between containers, each with a different set of requirements and associated costs. To ensure memory leaks are prevented, each container is responsible for acquiring and realizing its own resources during construction/destruction as well as once it goes out of scope.

Morpheus offers three types of data management routines, as shown in Table 3: (1) **Shallow** copy that results in the destination container sharing resources with the source container with minimal overheads as no data is being copied, (2) **Deep** copy that performs a bit-wise copy of data from source to destination using a *memcopy* operation and (3) **Convert** that allows conversions between two different storage formats through element-wise copies. It is worth pointing out that *Morpheus* does not implicitly manage data across memory spaces. Instead, the responsibility of synchronizing data across spaces lies to the user and facilitated via the *deep copy*3 semantics. To manage development overhead from direct format conversions, *Morpheus* uses a proxy format policy, with the COO format acting as the intermediate format. However, direct format conversions can be provided by users, if desired.

2.2.2. Mirroring

Morpheus supports a mirroring interface that, given a container, can create a compatible type in a user-specified memory space and allocate its size to match the original container. The `create_mirror()` routine always results in a new container with a new allocation and hence subsequent copies between the two containers will result in *deep* copies. On the other hand, the `create_mirror_container()` routine will result in a new container that is an alias to the mirroring container when the user-specified memory space is the same as the memory space of the mirroring container, otherwise it will result in the same behavior as the aforementioned routine. As a result, subsequent *deep* copies between the two containers will be transformed into *shallow* copies avoiding expensive data transfers between containers in the same space. This functionality is particularly useful in creating a single source performance portable code as it is demonstrated in Section 3.

2.2.3. Host-device model

The **Host-Device Model** is used to manage data transfers between different memory spaces using deep copies, such as between a CPU (host) and a GPU (device). By default, all containers are assumed to be on the device, and each container has an equivalent *HostMirror* type, that is always accessible by the host. Users can therefore utilize this functionality in conjunction with the mirroring routines to allocate containers and manage data transfers in both homogeneous and heterogeneous environments.

2.2.4. Abstract matrix representation

The main data structure of *Morpheus* is the *DynamicMatrix* container. The *DynamicMatrix* is a composition of all the available sparse matrix storage formats supported by *Morpheus* in a form of a type-safe union. Therefore, at any given time it can hold one of the available types and since the set of formats that it holds is known at compile time, the compiler can generate all versions of the algorithm *a priori* and only dispatch the right one at runtime by examining the active state of the container. This means low latency and therefore low runtime overheads, but also ease of use as the *DynamicMatrix* follows the same semantics as the other containers and it can therefore be declared, instantiated and invoked in the same way.

The *DynamicMatrix* acts as an abstract matrix representation that encapsulates the internal implementation details of each format, effectively resulting in a single interface that users can adopt to seamlessly use the available formats. The interface of the *DynamicMatrix* allows for format switching at runtime through the *activate* member function. This function evaluates an *enum* value that refers to the format to switch to. Note that switching will result in an empty matrix. In case it is required to also carry over the existing data, the *convert* routine can be used to perform an in-place conversion instead. Additionally, unsorted or sorted data can be passed to *DynamicMatrix* and will be sorted internally if needed.

2.2.5. Unified interface for algorithms across containers and spaces

Each algorithm is a *free* function. By exploiting the function overloading capabilities of *C*++ we can use the same function name to

represent functions that use different containers but perform the same conceptual algorithm. In addition, the adoption of a functional design naturally decouples the containers from the algorithms. Since containers are responsible for where the data is *located*, the algorithm only requires to know where it will be *executed*. As a result, the high-level interface of *Morpheus* for algorithms across the different spaces can remain identical simply by adding an extra parameter to the *free* function specifying the execution space the algorithm will be running in.

2.2.6. Adding new formats

The flat hierarchy design of the sparse containers in *Morpheus* allows developers to add new containers independently from the existing ones and thus each can be tested in isolation. For a new format to be included in the *DynamicMatrix*, it first has to be registered in the *FormatsRegistry* data structure by adding it to the union container. Then, for algorithms that use *DynamicMatrix*, each needs to be updated with an overload to the new format's implementation of the algorithm. In the case where the overload is accidentally omitted, this will result in a compile-time error requesting the developer to provide the missing implementation.

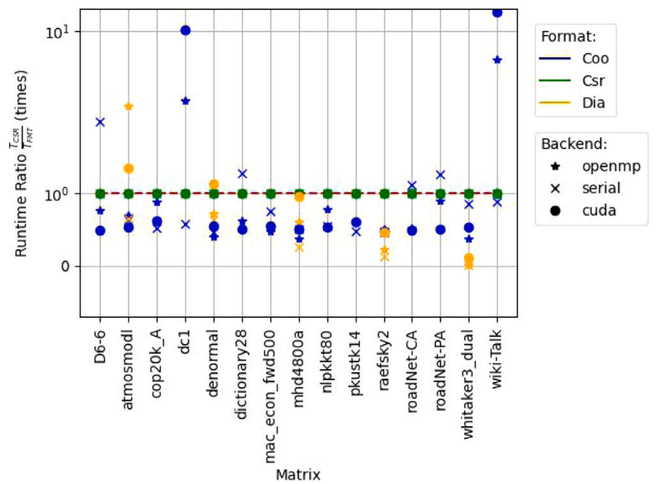


Fig. 2. SpMV runtime performance of the different formats and backends available in *Morpheus*, normalized by the runtime performance of the *CSR* format on the same backend. Various matrices with different sparsity patterns and from different domains, available in SuiteSparse collection, have been used.

3. Illustrative examples

In the following example, we demonstrate how *Morpheus* can be used in order to perform a SpMV multiplication using the *DynamicMatrix* container and on any of the supported backends, without any source code modifications.

```
#include <Morpheus_Core.hpp>
/* Default execution space: Either Serial or OpenMP backend.
 * If compiled with GPU support this is either HIP or CUDA. */
using Space = Morpheus::DefaultExecutionSpace;
// A random number generator running in default execution space
using Generator =
      Kokkos::Random_XorShift64_Pool<typename Space::execution_space>;
/* A Dynamic Matrix holding values of type double
 * and lives in the memory space of default execution space */
using Matrix    = Morpheus::DynamicMatrix<double, Space>;
/* A Dense Vector holding values of type double
 * and lives in the memory space of default execution space */
using Vector    = Morpheus::DenseVector<double, Space>;

int main(int argc, char* argv[]) {
  Morpheus::initialize(argc, argv);
  {
    std::string filename = argv[1];
    // Read format ID from command-line
    int fmt_id = atoi(argv[2]);
    // Load matrix on host
    typename Matrix::HostMirror Ah;
    Morpheus::IO::read_matrix_market_file(Ah, filename);
    /* In-place convert matrix to a dynamic matrix
     * with its active state set as per fmt_id */
    Morpheus::convert<Morpheus::Serial>(Ah, fmt_id);
    // Create a dynamic matrix that resides in Space
    Matrix A = Morpheus::create_mirror_container<Space>(Ah);
    // Copy data from host to container in Space
    Morpheus::copy(Ah, A);
    // Randomly initialize x and set y to zero
    Vector x(Ah.ncols(), Generator(0), 0, 1);
    Vector y(Ah.nrows(), 0);
    // SpMV multiplication in Space
    Morpheus::multiply<Space>(A, x, y);
  }
  Morpheus::finalize();
}
```

Listing 1: Performance portable SpMV example using the *DynamicMatrix* container. Changing the *fmt_id* at runtime will cause the active state of the matrix to change, and thus different *multiply* algorithms can be executed without code modifications.

Listing 1 illustrates how to read a sparse matrix from file into a *DynamicMatrix*, convert it to the format selected through the command line at runtime, and how to execute the multiply algorithm that performs SpMV in an arbitrary execution space defined at compile time. Lines 4-13 are shorthand definitions for the containers and spaces used in the example.

In the `main` program, the name of the matrix file to be loaded and the format ID to switch to for the *DynamicMatrix* are read from the command line (Lines 20-22). On Line 23 we load the matrix from file using the *read_matrix_market_file* routine. Note that the matrix is not loaded directly onto the device, but instead into a *DynamicMatrix* that resides on the Host defined on Line 22. The user is responsible for copying the matrix to the device.

On Line 26 we perform an in-place conversion that changes the active state of the *DynamicMatrix* to the one defined on the command line. The final step before performing the SpMV multiplication is to send all the containers that participate in the algorithm to the device. On Line 28 we create a new container that is a mirror of the host *DynamicMatrix* which is a new *DynamicMatrix* that lives in *Space* and on Line 30 we copy the data from the host to the device. Vectors x and y initialized (directly on the device) on Lines 32-33.

Finally, on Line 35 the SpMV multiplication is invoked with the algorithm executed in *Space*. Note that *Space* is defined on Line 4 to be the `Morpheus::DefaultExecutionSpace` that resolves to one of the supported execution spaces depending on how *Morpheus* was compiled. In other words, when it is compiled with GPU support it can be either *CUDA* or *HIP*, otherwise it will be *OpenMP* or *Serial*. This approach allows us to target different spaces without changing the source code, just by recompiling *Morpheus*, although it is also possible to explicitly define the execution space if so desired (Line 26).

Fig. 2 shows the runtime performance of the SpMV multiplication for the various formats and backends available in *Morpheus*, normalized by the runtime performance of the CSR format in the same backend. The example was executed on the CPU (*Serial* and *OpenMP*) and GPU (*CUDA*) nodes of the Cirrus supercomputer [12]. Each CPU node has two 2.1 GHz, 18-core Intel Xeon E5-2695 (Broadwell) processors. Each GPU node two 2.4 GHz, 20-core Intel Xeon Gold 6148 (Skylake) processors and four NVIDIA Tesla V100-SXM2-16 GB (Volta) GPU accelerators. The set of matrices used, available from the SuiteSparse [13] collection, includes matrices of different sparsity patterns and from different domains. The key takeaways from the experiment are:

1. For the same matrix, the format giving the best performance can vary across backends;
2. The choice of the optimum format can drastically improve runtime performance, sometimes even by an order of magnitude.
3. A poor choice of format can have a significant detrimental effect on performance.

The adoption of *Morpheus* allows users to evaluate the performance of different formats by simply changing the format ID on the command-line. In addition, by compiling *Morpheus* for different backends, users

can run on different types of hardware using the same code base. In both cases, **code modifications are not required** and performance optimizations due to format switching can be exploited.

4. Impact

Developers often choose a single format that performs generally well across operations and target hardware, even though a plethora of sparse matrix storage formats exist. This can be attributed to the fact that algorithms are coupled to a specific format and its interface. Any efforts to change the format require a complete re-write of the code, resulting in multiple versions of the same code with significant overheads introduced in development and maintenance.

For software that uses sparse matrices to be able to retain optimum performance throughout its lifetime it must be able to adapt to and support emerging formats and hardware architectures that can better utilize the different sparsity patterns of the matrices of interest. *Morpheus* provides a single dynamic sparse matrix representation with a unified interface and is capable of targeting multiple backends seamlessly. This allows users to decouple their algorithms from any specific format, while at the same time use an optimum data structure for a given operation, sparsity pattern and target architecture. As more formats and backends are added to *Morpheus*, users exploit these new developments without any further code modifications, thus improving the performance and extending the lifetime of their software. In addition, the overheads of supporting multiple formats and backends is shifted to the *Morpheus* developers, allowing users to remain agnostic about the low-level details of each format.

The adoption of *Morpheus* can assist in the optimization of iterative solvers through efficient format switching and selection. As a result, large scale applications that use iterative solvers, such as solving partial differential equations in the domains of Computational Fluid Dynamics (CFD)[14], urban earthquake response analyses [15], or weather/climate prediction [16] to name a few, can also be optimized. In addition, *Morpheus* can be used to transform CPU-only codes to be able to target heterogeneous hardware, e.g. with GPU accelerators.

Recently, *Morpheus* was successfully used to optimize the High Performance Conjugate Gradients (HPCG) benchmark [17] where it (1) enabled HPCG to target heterogeneous hardware and (2) provided runtime speed-up of up to 2.5× and 7× on CPUs and GPUs respectively (compared to the original MPI only version), via dynamically selecting the optimum format on each MPI process [2]. This result demonstrates the potential and usefulness of a library like *Morpheus* and can motivate the development of next-generation sparse linear algebra libraries and iterative solvers that are based on the principle of dynamic and adaptive matrices.

5. Conclusions

Morpheus is a library of sparse matrix storage formats that can be used as a building block into the development of sparse linear algebra applications and iterative solvers. By providing a *DynamicMatrix* container and a unified interface across multiple formats and backends, users can focus on the development of their applications without having to handle the low-level details of the different storage formats. *Morpheus* aims to provide performance portable sparse kernels not bounded to any particular storage formats, through an efficient and transparent runtime-switching of sparse matrix storage formats across multiple backends increasing the lifetime of their applications. Future developments of *Morpheus* focus on the expansion of the range of the storage formats supported, to better represent the different sparsity patterns available in science.

CRediT authorship contribution statement

Christodoulos Stylianou: Conceptualization, Data curation, Formal analysis, Investigation, Methodology, Project administration, Software, Validation, Visualization, Writing – original draft, Writing – review & editing. **Michèle Weiland:** Funding acquisition, Supervision, Writing – original draft, Writing – review & editing.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

No data was used for the research described in the article.

Acknowledgments

This research was funded in whole by EPSRC, United Kingdom under project ASiMoV (EP/S005072/1). For the purpose of open access, the author has applied a creative commons attribution (CC BY) licence to any author accepted manuscript version arising. We used the ARCHER2 UK National Supercomputing Service and the Cirrus UK National Tier-2 HPC Service at EPCC, funded by the University of Edinburgh, United Kingdom and EPSRC, United Kingdom (EP/P020267/1).

References

[1] Filippone Salvatore, Cardellini Valeria, Barbieri Davide, Fanfarillo Alessandro. Sparse matrix-vector multiplication on GPGPUs. ACM Trans Math Software 2017;43(4). http://dx.doi.org/10.1145/3017994.

[2] Stylianou C, Weiland M. Exploiting dynamic sparse matrices for performance portable linear algebra operations. In: 2022 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC). Los Alamitos, CA, USA: IEEE Computer Society; 2022, p. 47–57. http://dx.doi.org/10.1109/P3HPC56579.2022.00010.

[3] Filippone Salvatore, Buttari Alfredo. Object-oriented techniques for sparse matrix computations in fortran 2003. ACM Trans Math Software 2012;38(4). http://dx.doi.org/10.1145/2331130.2331131.

[4] Gamma Erich, Helm Richard, Johnson Ralph, Vlissides John. Design patterns: Elements of reusable object-oriented software. USA: Addison-Wesley Longman Publishing Co., Inc.; 1995.

[5] Su Bor-Yiing, Keutzer Kurt. ClSpMV: A cross-platform OpenCL SpMV framework on GPUs. In: Proceedings of the 26th ACM international conference on supercomputing. New York, NY, USA: Association for Computing Machinery; 2012, p. 353–64. http://dx.doi.org/10.1145/2304576.2304624.

[6] Tan Guangming, Liu Junhong, Li Jiajia. Design and implementation of adaptive SpMV library for multicore and many-core architecture. ACM Trans Math Software 2018;44(4). http://dx.doi.org/10.1145/3218823.

[7] Anzt Hartwig, Cojean Terry, Flegar Goran, Göbel Fritz, Grützmacher Thomas, Nayak Pratik, Ribizel Tobias, Tsai Yuhsiang Mike, Quintana-Ortí Enrique S. Ginkgo: A modern linear operator algebra framework for high performance computing. ACM Trans Math Software 2022;48(1):2:1–33. http://dx.doi.org/10.1145/3480935.

[8] Edwards H Carter, Trott Christian R, Sunderland Daniel. Kokkos: Enabling manycore performance portability through polymorphic memory access patterns. J Parallel Distrib Comput 2014;74(12):3202–16. http://dx.doi.org/10.1016/j.jpdc.2014.07.003, Domain-Specific Languages and High-Level Frameworks for High-Performance Computing.

[9] Trott Christian R, Lebrun-Grandié Damien, Arndt Daniel, Ciesko Jan, Dang Vinh, Ellingwood Nathan, Gayatri Rahulkumar, Harvey Evan, Hollman Daisy S, Ibanez Dan, Liber Nevin, Madsen Jonathan, Miles Jeff, Poliakoff David, Powell Amy, Rajamanickam Sivasankaran, Simberg Mikael, Sunderland Dan, Turcksin Bruno, Wilke Jeremiah. Kokkos 3: Programming model extensions for the exascale era. IEEE Trans Parallel Distrib Syst 2022;33(4):805–17. http://dx.doi.org/10.1109/TPDS.2021.3097283.

[10] Stroustrup Bjarne. The C++ programming language. 4th ed. Boston ; Munich [u.a.]: Addison-Wesley Professional; 2013.

[11] Boisvert Ronald F, Boisvert Ronald F, Remington Karin A. The matrix market exchange formats: Initial design, vol. 5935, US Department of Commerce, National Institute of Standards and Technology; 1996.

[12] EPCC. Cirrus UK national tier-2 HPC service. 2020, http://www.cirrus.ac.uk/.

[13] Davis Timothy A, Hu Yifan. The university of florida sparse matrix collection. ACM Trans Math Software 2011;38(1). http://dx.doi.org/10.1145/2049662.2049663.

[14] Owenson AMB, Wright SA, Bunt RA, Ho YK, Street MJ, Jarvis SA. An unstructured CFD mini-application for the performance prediction of a production CFD code. Concurr Comput: Pract Exper 2020;32(10):e5443. http://dx.doi.org/10.1002/cpe.5443, arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/cpe.5443, e5443 cpe.5443.

[15] Ichimura T, Fujita K, Yamaguchi T, Naruse A, Wells JC, Schulthess TC, Straatsma TP, Zimmer CJ, Martinasso M, Nakajima K, Hori M, Maddegedara L. A fast scalable implicit solver for nonlinear time-evolution earthquake city problem on low-ordered unstructured finite elements with artificial intelligence and transprecision computing. In: SC18: international conference for high performance computing, networking, storage and analysis. 2018, p. 627–37. http://dx.doi.org/10.1109/SC.2018.00052.

[16] Yang Chao, Xue Wei, Fu Haohuan, You Hongtao, Wang Xinliang, Ao Yulong, Liu Fangfang, Gan Lin, Xu Ping, Wang Lanning, Yang Guangwen, Zheng Weimin. 10M-core scalable fully-implicit solver for nonhydrostatic atmospheric dynamics. In: SC '16: Proceedings of the international conference for high performance computing, networking, storage and analysis. 2016, p. 57–68. http://dx.doi.org/10.1109/SC.2016.5.

[17] Dongarra Jack, Heroux Michael A, Luszczek Piotr. High-performance conjugate-gradient benchmark: A new metric for ranking high-performance computing systems. Int J High Perform Comput Appl 2016;30(1):3–10. http://dx.doi.org/10.1177/1094342015593158.