Research Paper

International Journal of HIGH PERFORMANCE COMPUTING APPLICATIONS

The International Journal of High Performance Computing Applications
2020, Vol. 34(6) 645–658
© The Author(s) 2020
Article reuse guidelines:
sagepub.com/journals-permissions
DOI: 10.1177/1094342020938421
journals.sagepub.com/home/hpc
SAGE

MAGMA templates for scalable linear algebra on emerging architectures

Mohammed Al Farhan[1], Ahmad Abdelfattah[1], Stanimire Tomov[1], Mark Gates[1], Dalal Sukkari[1], Azzam Haidar[2], Robert Rosenberg[3] and Jack Dongarra[1,4,5]

Abstract

With the acquisition and widespread use of more resources that rely on accelerator/wide vector–based computing, there has been a strong demand for science and engineering applications to take advantage of these latest assets. This, however, has been extremely challenging due to the diversity of systems to support their extreme concurrency, complex memory hierarchies, costly data movement, and heterogeneous node architectures. To address these challenges, we design a programming model and describe its ease of use in the development of a new *MAGMA Templates* library that delivers high-performance scalable linear algebra portable on current and emerging architectures. MAGMA Templates derives its performance and portability by (1) building on existing state-of-the-art linear algebra libraries, like MAGMA, SLATE, Trilinos, and vendor-optimized math libraries, and (2) providing access (seamlessly to the users) to the latest algorithms and architecture-specific optimizations through a single, easy-to-use C++-based API.

Keywords

Accelerator-driven computing, HPC, linear algebra, performance portability, polymorphism, programming models

1 Introduction

The dramatic advancements in microprocessor design over the past couple of decades have significantly improved the performance of scientific simulations. Nevertheless, the ever-expanding gap between the developing demands for massive computations and the languishing transistor budgets triggered by the "retirement" of Moore's Law has inevitably deteriorated the possible performance gains out of the architectural advancements in the hardware design. Therefore, fine-grained parallelism (Abduljabbar et al., 2018) required at the node-level is becoming pervasive, especially since the performance of a compute node that powers the current and future supercomputers is highly dependent upon the performance provided by a tightly coupled specialized hardware for accelerator-driven computing (e.g., GPUs) connected directly to the compute node via a high-bandwidth, high-speed interconnect (e.g., NVIDIA NVLink) (Abduljabbar et al., 2017).

The hierarchical, synergistic level of parallelism induced by the complicated heterogeneity of the HPC compute node makes squeezing out the full performance potentials from supercomputers a daunting proposition. To this end, many development efforts have been directed toward either porting existing scientific kernels onto various emerging HPC architectures, or developing new kernels from ground up targeting new hardware generations. Thus, the current available scientific software stack, which is mostly diversified based on either a specific hardware implementation or various algorithm-centric formulations, is very extensive and divergent. This in turn makes application developers confused about which kernels to invoke or choose on a particular hardware, or how to switch from one kernel that already powers production-level applications to another, based on the underlying hardware or software.

To overcome the aforementioned issues and challenges, we develop *MAGMA Templates*, which is a set of APIs and computational kernels/patterns—in other words, templates—combined in a single new computational library. MAGMA Templates provides the performance-portable computational backend that many HPC scientific

[1] The University of Tennessee, Knoxville, TN, USA
[2] Nvidia Corporation, Santa Clara, CA, USA
[3] Naval Research Laboratory, Washington, DC, USA
[4] Oak Ridge National Laboratory, Oak Ridge, TN, USA
[5] University of Manchester, Manchester, England, UK

Corresponding author:
Mohammed Al Farhan, Innovative Computing Laboratory, University of Tennessee, Knoxville, TN 37996, USA.
Email: farhan@icl.utk.edu

simulation production codes need in order to be easily ported to new architectures.

The name of MAGMA Templates library is derived from Matrix Algebra on GPU and Multicore Architectures (MAGMA) library (Agullo et al., 2009; Tomov et al., 2010). The reason is that we initially aimed to develop a modern, high-level C++ backend API for MAGMA library. However, as the scope of coverage was extended beyond MAGMA, e.g. to include support for distributed-memory systems, we designed MAGMA Templates to be a high-level thin layer set on top of multiple numerical kernels/libraries. We therefore include support for Software for Linear Algebra Targeting Exascale (SLATE) (Gates et al., 2019), Trilinos/PETSc/HYPRE, and vendor-optimized math libraries.

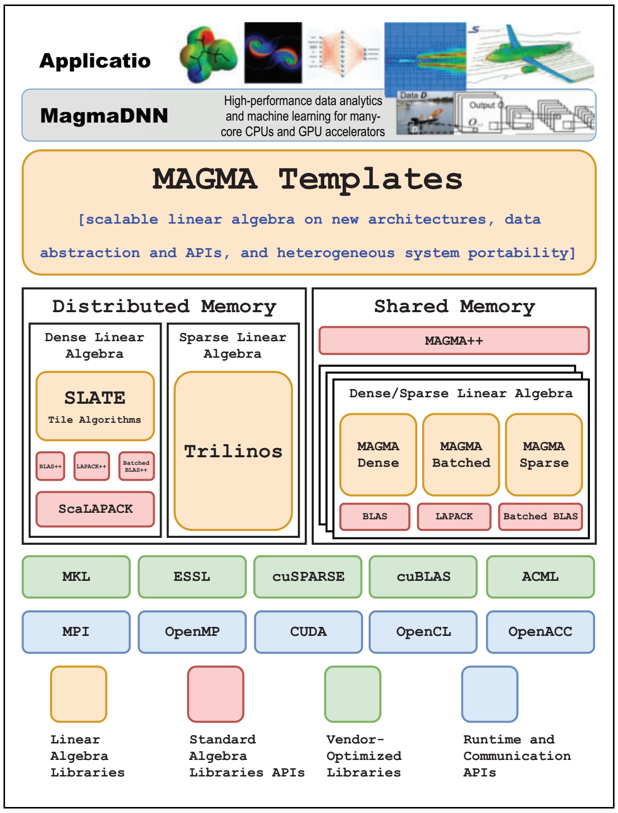
MAGMA Templates creates a layered package with APIs that make the routines easily pluggable, extendable, tunable, and interoperable with various linear algebra libraries and achieves the following objectives:

1. Make the most up-to-date algorithms and highly tuned numerical kernels available as building blocks for production codes on emerging architectures.
2. Develop a high-level C++ software toolkit to provide a single, easy-to-use interface to a wide variety of data structures and solvers for dense and sparse linear algebra on a broad spectrum of shared- and distributed-memory, large-scale systems, which is useful for application developers seeking scalable performance.
3. Design caliber data abstractions and APIs to ease interoperability and integration through the familiar Basic Linear Algebra Subprograms (BLAS), Linear Algebra PACKage (LAPACK), and Scalable Linear Algebra PACKage (ScaLAPACK) interfaces, wherever possible.
4. Implement self-contained, novel linear algebra algorithms that can replace the currently used libraries in production codes, including, but not limited to, low-rank compression, dense matrix factorizations and solvers, iterative solvers, and preconditioners.

Having introduced the goals and objectives of the MAGMA Templates library, the rest of the paper is organized as follows. In Section: "MAGMA Templates Software Design", we illustrate the high-level design philosophy of MAGMA Templates. Section: "MAGMA++: A high-level C++ API for MAGMA" details the components and functionalities of the underlying MAGMA++: MAGMA Templates high-level C++ backend API for MAGMA. Section: "Auto-tuning" describes some experiments regarding auto-tuning of MAGMA library done through the MAGMA Templates interface. Finally, in Section: "Conclusions and Future Directions" we discuss our conclusions and the highlights from our ongoing work.

2 MAGMA templates software design

The overall MAGMA Templates software design is illustrated in Figure 1. High performance is derived from using vendor-optimized BLAS/LAPACK and linear algebra libraries, when available, or via taking advantage of the basis libraries upon which MAGMA Templates depends.


Applicatio
MagmaDNN
High-performance data analytics and machine learning for many-core CPUs and GPU accelerators
MAGMA Templates
[scalable linear algebra on new architectures, data abstraction and APIs, and heterogeneous system portability]
Distributed Memory
Dense Linear Algebra
Sparse Linear Algebra
SLATE
Tile Algorithms
BLAS++
LAPACK++
Batched BLAS++
ScaLAPACK
Trilinos
Shared Memory
MAGMA++
Dense/Sparse Linear Algebra
MAGMA Dense
MAGMA Batched
MAGMA Sparse
BLAS
LAPACK
Batched BLAS
MKL
ESSL
cuSPARSE
cuBLAS
ACML
MPI
OpenMP
CUDA
OpenCL
OpenACC
Linear Algebra Libraries
Standard Algebra Libraries APIs
Vendor-Optimized Libraries
Runtime and Communication APIs
Figure 1. MAGMA Templates ecosystem.

Higher in the software stack, MAGMA Templates relies on the MAGMA sub-libraries to provide dense and sparse linear algebra kernels for shared-memory heterogeneous architectures, but provides a single library with a unified interface that is functionally and performance portable across various architectures through its MAGMA++ API.

MAGMA Templates uses the SLATE library (Gates et al., 2019) to provide dense linear algebra kernels for distributed-memory heterogeneous architectures (Abdelfattah et al., 2017; Kurzak et al., 2017, 2019a). Note that Figure 1 shows that Trilinos (Heroux et al., 2005) is the backend for the distributed-memory sparse linear algebra kernels on which MAGMA Templates depends. However, the current implementation is only a "proof of concept" for some sparse kernels targeting distributed-memory, and linking against Trilinos and other libraries (e.g., PETSc and HYPRE), is still a key consideration in our ongoing work. Having said that, integrating a specific backend library may require building a middle layer, such as MAGMA++, to implicitly handle the transition, and to maximize the ability to switch between different sparse matrix data structures and their representations based on performance and the application-specific compatibility.

Since SLATE is a well-designed C++ library for distributed-memory dense linear algebra, we base the

MAGMA Templates high-level design on the interface of SLATE, which is very modular and serves our objective. However, MAGMA—the shared-memory backend of MAGMA Templates—is written in C and has completely different interface than SLATE. Therefore, some of our efforts in this project was focused upon developing the C++ interface for MAGMA (MAGMA++), so that MAGMA Templates can switch, seamlessly, between MAGMA and SLATE with no overhead, while using a unified interface. To develop a unified matrix class that is perfectly suitable for shared-and distributed-memory settings and, meanwhile, it is very simple and abstract, we first implemented the MAGMA++ API to provide a C++ set of layers specific for MAGMA library. With that on mind, MAGMA Templates becomes a C++ "thin layer" that dwells on top of various linear algebra libraries to carry out the matrix computations without handling any sort of heavy lifting computations. In addition, MAGMA++ is an independent API, which can be configured, installed, and invoked without MAGMA Templates. Thus, MAGMA users, who use C++ to develop their application codes, would find using MAGMA++ more convenient to use rather than directly invoking MAGMA routines in a C++ code. However, if application developers would like to seamlessly switch from one library into another, then MAGMA Templates provides this smooth, straightforward, and easy transition.

MAGMA Templates is therefore a convenient one-stop shop for many combinations of HPC software and hardware for scientific and engineering simulations. For instance, with the extendable matrix class implemented in MAGMA Templates, the mapping illustrated in Figure 1 to SLATE, Trilinos, MAGMA, and other linear algebra libraries is fairly easy. The matrix class is just a data structure wrapper that exploits the already allocated user-space memory, and then MAGMA Templates implicitly does on-the-fly, "zero-overhead" translation into the suitable kernel implemented by the basis software stack. Using our novel mapping and transition algorithm, MAGMA Templates translates the function arguments, initializes the target library, architecture and required workspaces, builds prerequisite execution policies and queues, and accurately maps the user space without allocating extra memory. To this end, wrapping existing user memory on given devices, allows caller to retain ownership of data and the responsibility for maintaining it over the lifetime of the matrix object, including any shallow copies.

2.1 Functional portability

Functional portability is handled by design, through applying polymorphic approaches for a modular design. Details on the approach are given below. The modular design allows adding more functionality and support for different architectures, when developed. When a kernel or algorithm implementation for particular architecture is still missing, the code will still run, but the runtime system will schedule the execution on hardware for which there is implementation.

2.2 Performance portability

Performance portability is achieved through (auto-)tuning and specialization in the implementations. The current design allows us to add different versions of the same algorithm, but is designed and optimized for possibly different architectures. Every implementation is additionally parameterized to allow for subsequent tuning. Currently we have auto-tuning settings for particular kernels. The goal is to extend this for the entire library.

3 MAGMA++: A high-level C++ API for MAGMA

3.1 Data abstractions

We design two main data abstractions based on the object-oriented programming (OOP) paradigm, and we develop various APIs to support them. The implementation is through C++ classes, namely: `Matrix` class and `Tensor` class. The standard way to represent a matrix in BLAS and LAPACK is through a pointer to the memory location where a matrix starts, the matrix size (m and n for an $m \times n$ matrix), and leading dimension (ld). Typically, a column-major data layout is assumed. To abstract the particular data storage type (column-major, row-major, dense/ sparse,) and storage memory location (CPU, GPU,), we design an abstract storage class, namely `SharedStorage`, which the `Matrix` class and `Tensor` class use to hide their data storage implementation details, and make the data structures extendable to encapsulate several storage formats and layouts.

The data layout is abstracted in Listing 1, which manifests the `SharedStorage` class. [Note: For convenience we keep two pointers to the data: `data_` that is for CPU data and `ddata_` for device/GPU data, which allows us to offload to a run-time system the management of moving data between CPUs and GPU devices. The location of the most up-to-date data is stored in `device_`.] Every matrix has a `SharedStorage` object stored as a shared pointer by all matrix classes.

The matrix data structure is presented in Listing 2. The design decision of having all the information defining a matrix stored in the matrix class allows us to reduce the number of arguments in standard API routines, like in BLAS and LAPACK, where matrices and submatrices are explicitly described through input arguments for different matrix sizes, transpose operations, leading dimension specifications, etc. In addition, this makes the base code extendable, readable, and, most importantly, bug free, by having less error-prone code. Also, one can integrate internal exception handlers specific to the data object.

The `BaseMatrix` class provides a function to read the local matrix entries stored in the function caller's memory

Listing 1. MAGMA++: `SharedStorage` class.

```
namespace magma {
// Templated data type:
// float, double, std::complex< float >,
// and std::complex< double >.
template< typename scalar_t >
class SharedStorage {
public:
// 1) Class constructor:
// 1.1) Wrap existing data.
// 1.2) Caller retains ownership.
SharedStorage(int64_t size, scalar_t* data, int
     device);
// 2) Class constructor:
// 2.1) Allocate new data
SharedStorage(int64_t size, int device);
// 3) Class destructor.
~SharedStorage()

...

protected:
// Pointers to contiguous data in the
// memory of a particular size.
scalar_t* data_; // CPU data.
scalar_t* ddata_; // GPU/device data.
int64_t size_; // Size of the allocation.
// Location of the memory: CPU, GPU, ...
// [Note: If the location is main memory
// (CPU memory), then the device number is
// automatically set to: -1, otherwise it
// takes the GPU/device id (i.e., 0, 1, ...).]
int device_;
// Indication if a particular object owns the
// data or not.
bool own_;
// To enable OpenCL implementations,
// the storage is given through the below object,
// and the availability of which is controlled
// at installation time by a macro definition
// (#define directive).
#ifdef MAGMA_WITH_OPENCL
cl_mem mem_;
#endif
}; };
```

Listing 2. MAGMA++: Matrixparentclass(`BaseMatrix`).

```
namespace magma {
template< typename scalar_t >
class BaseMatrix {
public:
// Default constructor initializes empty matrix
// no memory is allocated.
BaseMatrix();
// Constructor wraps existing memory of m-by-n
// matrix, stored in an ld-by-n array, on the
// given device. Caller retains ownership of data
// and is responsible for maintaining it over the
// lifetime of this matrix object,
// including any shallow copies.
BaseMatrix( int64_t m, int64_t n, scalar_t* A,
     int64_t ld, int device);
...

protected:
// To access a submatrix, the following class
// members give the row and column offsets,
// respectively, in regards to the matrix data
// layout object.
int64_t ioffset_, joffset_;
int64_t m_, n_; // Matrix sizes (dimensions).
int64_t ld_; // Leading dimension.
// Indication if there are some operation
// associated with the matrix,
// like transpose, conjugate transpose,
// non-transpose, ...
Op op_;
// The data layout
std::shared_ptr< SharedStorage< scalar_t > > storage_
     ;
}; };
```

location, which facilitates access class members with no race condition, see Listing 3.

Listing 3. MAGMA++: Reference to a matrix local element.

```
// Reference to local element {i, j}
// operator overloading.
// Write access.
scalar_t& operator() (int64_t i, int64_t j)
{
  return const_cast< scalar_t& >(static_cast<
    BaseMatrix const& >(*this)(i, j));
}
// Read access.
scalar_t const& operator() (int64_t i, int64_t j)
    const
{
  if (op_ == Op::NoTrans) return storage_->data()[
    offset_ + i +  j * ld_];
  else return storage_->data()[offset_ + j +  i * ld_
    ];
}
```

Specialized classes for various types of matrices (general, trapezoid, triangular, symmetric, etc.) are derived from the `BaseMatrix` class. For example, Listing 4 is the triangular matrix (`TriangularMatrix`) class.

Listing 4. MAGMA++: Triangular Matrix class (`TriangularMatrix`).

```
namespace magma {
template< typename scalar_t >
class TriangularMatrix : public TrapezoidMatrix<
    scalar_t > {
public:
// Empty matrix
TriangularMatrix(): TrapezoidMatrix< scalar_t >() {}
// Wrap existing memory
TriangularMatrix(Uplo uplo, int64_t n, scalar_t* A,
    int64_t ld, int device=host_device):
    TrapezoidMatrix< scalar_t >(uplo, n, n, A, ld,
    device) {}
// Allocate new matrix
TriangularMatrix(Uplo uplo, int64_t n, int64_t ld=-1,
     int device = host_device): TrapezoidMatrix<
    scalar_t >(uplo, n, n, ld, device) {}

...

}; };
```

3.1.1 Tensor. We have generalized this data structure into a `Tensor` class; see Listing 5 for the `Tensor` class design. In contrast to the `Matrix` class, where the matrix dimensions are represented by m_ and n_, the tensor size is an `std:: vector`, namely `dim_`, and the dimensions of the tensor is the size of `dim_` container. The meanings of the `offset_`, `storage_`, and `dim_ld_` members are similar to the `offset_`, `storage_`, and ld_ from the `matrix` class, except that here the corresponding values are given in the corresponding vector elements.

3.2 Naming conventions and calling specifications

The naming convention follows BLAS, LAPACK, and ScaLAPACK, despite some negligible differences,

Listing 5. MAGMA++: `Tensor` class.

```
namespace magma {
template< typename scalar_t >
class Tensor {
public:
// Constructors and other member functions
Tensor( std::vector< int > dim_ );

...

private:
std::vector< int64_t > offset_, dim_, dim_ld_;
std::shared_ptr< SharedStorage< scalar_t > > storage_
    ;
}; };
```

characterized by the language features. For example, the first character in these standards specifies the precision: “d” for double, “s” for single, “z” for double complex, and “c” for single complex. MAGMA follows this convention: the implementations are written for the “z” precision, and the implementations for the other precisions are generated by scripts. However, in MAGMA++, no precision is specified, and we use C++ templates to represent generic function types. Thus, the compiler generates the code for different precisions. For example, the Cholesky factorization, namely ZPOTRF in LAPACK for double-complex precision, is shown in Listing 6. Note that the implementation is templated for the target architecture, which allows us to have different implementations that rely on already developed/available software components through vendor libraries, or the MAGMA libraries. In particular, details on this concept for the implementation of the Cholesky factorization from Listing 6 are given in Listing 7. This is the basis of our polymorphic approach that allows us to use already developed parts, tuned for particular architectures, and further update the routines when new versions become available.

The BLAS and LAPACK interface uses overloaded APIs to support both CPUs and GPUs. Each API is templated for precision, as described before. For instance, the API for the matrix multiplication (GEMM) looks like Listing 8. Note the extra argument `Queue`, which is a C++ class that provides an execution context for managing GPUs. It hides the complexity of dealing with the GPU runtime, which is provided by the vendor. The user must create a queue and pass it to the BLAS call in order to execute the routine on the GPU.

Listing 6. MAGMA++: Cholesky factorization `potrf` function.

```
namespace magma {
// Returns i if leading minor of order
// i is not positive definite.
// Throws other (non-numeric) errors.
// Right-looking version.
template< Target target = Target::Hybrid, typename
    scalar_t >
// The calling specifications are simplified due
// to a reduced number of parameters passed to the
// routines, as many of them are packed into
// the matrix. class.
// The matrix sizes and storage are passed through
// the matrix object, and the rest is for specifying
// implementation, tuning options, and results.
int64_t potrf(LookType< Look::Right >, magma::
    HermitianMatrix< scalar_t >& A, std::vector<
    magma::Queue >& queues, std::vector< std::pair<
    Option, Value > >& opts
) {

...

}; };
```

Listing 7. MAGMA++: Polymorphic approach for a modular design.

```
#if DISPATCH
// Dispatch based on location and
// target type to existing routines.
if (target == Target::Hybrid) {
  if (A.device() == host_device)
    info = magma::potrf(A.uplo(), n, A.data(), A.
     stride());
  else
    info = magma::potrf_gpu(A.uplo(), n, A.data(), A.
     stride());
}
else if (target == Target::CpuOnly)
  lapack::potrf(A.uplo(), n, A.data(), A.stride());
else if (target == Target::DeviceOnly)
  info = magma::potrf_native(A.uplo(), n, A.data(), A
    .stride());
// We have exception handling engine specific
// to MAGMA, implemented internally inside
// MAGMA++ library, to catch run-time errors
// and handle them through the utilization
// of C++ OOP functionalities.
if (info < 0) throw magma::exception();

...
```

Listing 8. MAGMA++: Matrix multiplication (gemm).

```
// CPU GEMM interface
namespace magma {
template< typename scalar_t >
void gemm(scalar_t alpha, Matrix< scalar_t > const& A
    , Matrix< scalar_t > const& B, scalar_t beta,
    Matrix< scalar_t >& C );
// GPU GEMM interface
template< typename scalar_t >
void gemm(scalar_t alpha, Matrix< scalar_t > const& A
    , Matrix< scalar_t > const& B, scalar_t beta,
    Matrix< scalar_t >& C, Queue& queue );
};
```

As illustrated before, matrices are no longer represented by raw pointers. C++ classes are implemented to represent matrices as objects with properties. Explicit instantiation is required to support a certain precision. The library supports the four standard precisions (single, double, complex, and double complex). For example, for double precision, see Listing 9.

Listing 9. MAGMA++: Double precision instantiation.

```
// CPU interface for double precision template
namespace magma {
void gemm(double alpha, magma::Matrix< double > const
    & A, magma::Matrix< double > const& B, double
    beta, magma::Matrix< double >& C);
// GPU interface for double precision template
void gemm(double alpha, magma::Matrix< double > const
    & A, magma::Matrix< double > const& B, double
    beta, magma::Matrix< double >& C, magma::Queue&
    queue );
};
```

LAPACK interfaces, on the other hand, are similar to SLATE's. They are templated for precision, but they accept a list of options that are currently used only to specify the targeted hardware for execution, see Listing 10.

Listing 10. MAGMA++: Triangular solve (`getrf`).

```
namespace magma {
template < typename scalar_t >
int64_t getrf(magma::Matrix< scalar_t >& A, std::
     vector< int64_t >& ipiv, std::map< Option, Value
     > const& opts);
};
```

The `opts` argument is an `std:: map` of pairs that lets the user specify certain options during the execution time. A LAPACK routine reads the list of options before running the correct routine. Right now, the routine only reads the `Option:: Target` option. Two modes are supported:

- If the `target` option is set to `Target:: Hybrid`, then the routine will call the hybrid MAGMA implementation. The input matrix can be stored in the CPU memory or in the GPU memory.
- If the `target` option is set to `Target:: Host`, then MAGMA Templates calls the equivalent routine from the LAPACK++ library (Gates et al., 2017). The data are assumed to be resident in the CPU memory only. Otherwise, the user will be notified with an error.

Similarly, an explicit instantiation is required to support the four standard precisions.

3.3 Domain-specific language (DSL)

MAGMA++ provides a Python-based code generation tool to easily develop portable implementations. It provides a python script that generates GPU CUDA code (or parallel OpenMP CPU code) from DSL constructs/templates. The templates cover certain computational patterns that are not covered by the standard linear algebra routines available in the MAGMA library. It simplifies development and porting of code to use the MAGMA++ library for different architectures.

3.4 Sparse data abstractions

The sparse data formats supported in MAGMA++ are inherited from MAGMA Sparse. A subdomain can be stored on a CPU memory or on a GPU/device memory. For CPUs, we support COOrdinate format (COO) and Compressed Sparse Row (CSR). COO is a sparse matrix format that stores only the nonzero coefficients by compressing the entire 2D coefficient table. The nonzero elements are stored in a 1D array row-wise. Additional row index and column index arrays are used to identify both the row and the column of each nonzero element. The type representing the precision is templated. The CSR, on the other hand, is a standard sparse matrix format that many packages support. It stores only the nonzero coefficients by compressing each row of the matrix (compressed sparse row format). The nonzero elements are stored in a 1D array row-wise, and are accompanied by a row pointer array that stores the starting index of each row. An additional column index array is used to identify the column of each nonzero element. The type representing the precision, as in all other supported formats, is templated.

For GPUs/devices, we support more formats since performance for the different formats is more sensitive to the application (the nonzeros structure of the matrix). In particular, in addition to COO and CSR, we support ELL, SELL-P, CSR5 (Liu and Vinter, 2015), and HYB (Guo et al., 2016) formats. Details on these formats are given in Anzt et al. (2014, 2017). Routines for conversion between the formats are also provided. Note that other sparse formats, e.g. JAD (Li and Saad, 2013), are considered to be supported in our ongoing work.

In order to automate the memory management, we use standard C++ smart pointer classes: `std:: unique_ptr` and `std:: shared_ptr`. This means that the users are never required to explicitly allocate or free the memory when working with MAGMA++. Instead, the library handles this automatically.

To demonstrate a simple possible use of the MAGMA++ sparse API, Listing 11 provides a code snippet to exemplify the memory management abstractions supported by MAGMA++.

Listing 11. MAGMA++: Read a sparse matrix from a file in CSR format.

```
auto A = magma::read_from_mtx< magma::MatrixCsr<
     scalar_t > >( "filename".mtx, {cpu|gpu} );
// Copy a sparse matrix from CPU/GPU to GPU/CPU
auto B = magma::copy_to< magma::MatrixCsr< scalar_t >
      >( A.get(), {cpu|gpu} );
```

3.5 Domain decomposition data abstraction

The distributed sparse matrices interface of MAGMA++ is derived from the basic single node matrix class (i.e., `MatrixCsrDist` inherits `MatrixCsr`). Thus, the local format and storage are the same as the parent nodal matrix class (`MatrixCsr`). The distributed matrices, however, are enhanced with the connectivity information for the neighboring subdomains. For example, if a global sparse matrix A is represented as a collection of sparse matrices $A_{i,j}$, where i, j vary from 1 to a number of subdomains S, the local `MatrixCsrDist` matrix for subdomain i stores $A_{i,i}$ in the standard `MatrixCsr` format, as well as the $A_{i,j}$ matrices ($j = 1, .., i-1, i+1, .., S$). The columns that hold nonzero entries are reordered for each of the $A_{i,j}$ matrices to have continuous span (starting from 0). This is done in order to optimize the performance of global operations like Sparse Matrix-Vector multiplication (SpMV). In particular, in order to apply $A_{i,j}$ to a global vector in subdomain i, subdomain j must send to i^{th} elements of the vector that correspond to columns with nonzeroes in $A_{i,j}$. By preprocessing the data that has to be sent, we group the data that j has to send to i in one package. This minimizes costly latencies that otherwise would have to be incurred if communication was done through a number of sends. Thus, a global matrix-vector product involves the following steps (steps 1.3 and 4 are overlapped; steps 1.4 must finish before step 5 is applied):

1. Pack the values, which have to be sent to other subdomains.
2. Send (i.e., point-to-point communication) all packages to the neighboring subdomains.
3. Receive the corresponding packages from the neighbors.
4. Apply local SpMV, using MAGMA Sparse.
5. Add local contributions to SpMV for the data coming from the neighbors.

Note that the current distributed CSR matrix implemented in MAGMA++ aims to utilize MAGMA Sparse library, and thereby we replicate the matrix locally across the compute nodes. Indeed, this is not the optimal approach in terms of memory efficiency, in contrast to the most efficient and common technique utilized by HYPRE or PETSc, where they store only two CSR matrices: 1) one for the local domain, and 2) one for all the off-domain submatrices. However, with data replication we can exploit the performance capability of MAGMA Sparse locally at the node-level. Nevertheless, once the middle layer interface for the distributed sparse matrix computations library is developed, we will be able to opt-out the current sparse matrix distribution and rely upon invoking the sparse kernels provided by the chosen distributed sparse library, e.g. Trilinos. This feature presented herein is to prototype the distributed-memory sparse matrix algebra capability of MAGMA Templates.

In MAGMA++, we develop three main ways for users to interact with the distributed-memory sparse solvers to support the following functionalities:

1. A matrix can be read from file (e.g., in the format used for the University of Florida sparse matrix collection, Davis and Hu, 2011), and distributed as specified by the user.
2. Users can generate the matrix on the fly and initialize the `MatrixCsrDist`. A C++ `MatrixLaplaceDist` class is developed as a test example, which inherits the `MatrixCsrDist` class and generates entries for 2D/3D Laplacian discretizations on a regular grid to be filled in a distributed CSR matrix format. [Note: This example allows us also to easily generate very large matrices and study the performance of the solvers provided.]
3. Users can use their own data formats but alternatively provide the basic building blocks for the MAGMA Templates solvers; the main building blocks needed are SpMV, dot product, and adding vectors (corresponding to the Level-1 BLAS AXPY routine), and all are developed in MAGMA++.

4 Auto-tuning

The MAGMA Templates library provides a unified interface through which different libraries with different backends can be called. In this context, and in order to maintain performance portability, auto-tuning for performance is often done on the backend level rather than on the high-level interface (i.e., MAGMA Templates). Eventually, any performance auto-tuning that is carried out on any backend gets automatically leveraged to the MAGMA Templates library.

We will present some auto-tuning experiments performed on MAGMA++ to support MAGMA library backend, which provides several algorithms for dense and sparse linear algebra on GPU-accelerated systems. Considering performance tuning parameters of each algorithm, we recognize two types of auto-tuning experiments, compile-time and run-time (Li et al., 2009). The following subsections detail these experiments.

4.1 Experimental platforms

In order to avoid having a very long discussion, we will limit our scope here to tune the batched GEMM kernel for two different GPUs, and for the single test case of square matrix multiplication using the MAGMA Templates interface of MAGMA. More specifically, the experimental setup can be summarized as follows:

- **Hardware**: Two systems with two different GPUs: 1) the first one has a Pascal P100 GPU, and 2) the second has a Volta V100 GPU. The host CPU in each system connects its device GPU through a PCI-e connection. Both systems have the same host CPU, which is a dual-socket Intel Haswell CPU. Each socket has 10 CPU cores, resulting in a 20-core CPU per system. The two systems has a 64 GB of DRAM.
- **Software**: The latest MAGMA release to date (v2.5.1), compiled with CUDA Toolkit 10.1. The MAGMA library requires a BLAS/LAPACK provider for the CPU side. We use Intel MKL 2018.
- **Algorithms**:
  - **Compile-Time Tuning Parameters:** Batched matrix multiplication for square sizes. We show sample results for double-precision arithmetic only. The experiments are replicated for the other three standard precisions (i.e., single precision, single-complex precision, and double-complex precision).

- **Run-Time Tuning Parameters:** One-sided matrix factorization (LU/QR/Cholesky). We show sample results for double-precision arithmetic only. The experiments are replicated for the other three standard precisions (i.e., single precision, single-complex precision, and double-complex precision).

4.2 Compile-time tuning parameters

As the name suggests, such parameters must have their values defined during the compilation time. The necessity of such a condition might be due to different reasons. For example, compilers can easily unroll loops whose intervals are constants, which helps eliminate the cost of branching and potentially some of the memory address calculations inside the loop (Al Farhan, 2019; Al Farhan and Keyes, 2018; Al Farhan et al., 2016). Another reason is static register allocation and register indexing, which are crucial for GPU kernels (otherwise, the compiler spills the associated variable to the global memory, resulting in a severe performance drop). In most cases, compile-time tuning parameters are popular in low-level kernels that implement important building blocks of an algorithm. The MAGMA library is one of the backends of the MAGMA Templates library. It implements several important GPU kernels for dense linear algebra algorithms. Ideally, such kernels require tuning for different GPU architectures and across different precisions. We will present an example for auto-tuning the batched GEMM kernel, which implements matrix multiplication on a batch of relatively small matrices. Such a kernel is a very important building block in many applications beyond linear algebra solvers, such as tensor contractions and machine learning algorithms.

The batched GEMM kernel in the MAGMA library has at least five compile-time tuning parameters. This means that the search space is very large for such a kernel. In fact, a full auto-tuning sweep for the batched GEMM routine can be quite exhaustive. This is due to the following reasons:

- The search space has five independent tuning parameters.
- The standard batched GEMM kernel must support the four standard precisions (single, double, single complex, and double complex).
- A single kernel instance with a specified set of parameter values must be tested against several use cases for the batched GEMM routine. Examples are square sizes, rank updates with tall-and-skinny matrices, rank updates with a mix of tall-and-skinny; small square matrices, multiplications with different matrix transpositions, and many others. In other words, the test space for a single kernel instance is not trivial even for a single architecture.

There are few countermeasures that can be taken in order to mitigate the time and effort of a brute-force sweep of the batched GEMM kernel. The following characterizes some aspects that can be utilized to prune the search space:

1. **Applying constraints.**
   (a) *Hard constraints*: They are imposed by the hardware. In a typical GPU-accelerated environment, hard constraints usually represent capacity limits for different aspects of the GPU execution model. For example, the maximum shared-memory that can be allocated by a single thread block, the maximum number of registers per thread, the maximum number of thread-blocks per multiprocessor, and many others. The search space can then be pruned for a specific GPU architecture based on its hardware limitations.
   (b) *Soft constraints*: They are regarded as heuristics defined by experienced developers who have enough knowledge to judge a specific kernel configuration without having to run it. For instance, having too many threads per thread block in a batched kernel usually limits the ability of the GPU runtime to schedule many thread blocks on the same multiprocessor, which results in a low occupancy and a bad performance. In this regard, we can set a soft constraint to limit the maximum number of threads to 256. Since the hardware limit for this aspect is 1024, the soft constraint cuts the search space for this aspect by 75%. Other soft constraints can be defined for other aspects as well.
2. **Considering the most important and relevant test cases based on domain-specific knowledge.**

   (a) *The test space can be pruned for the most widely used test cases for batched GEMM*: The most important use cases for batched GEMM are square multiplications on small sizes, and rank-k updates for batched matrix factorization.

   The experiment is conducted using three types of "automatic scripts." Each script has its distinct functionality in the tuning process. The reason behind the separation of scripts (i.e., rather than using one big tuning script), is that some steps in the tuning process might not be repeated as frequently as other steps. Therefore, we recognize the following three steps, each being assigned to a separate automatic script.

1. **Search space enumeration**: In this step, we define distinct kernel instances with unique combinations of tuning parameter values. Each combination represents a unique kernel version, and is usually assigned a unique ID number for future referencing in the following step. The enumeration process

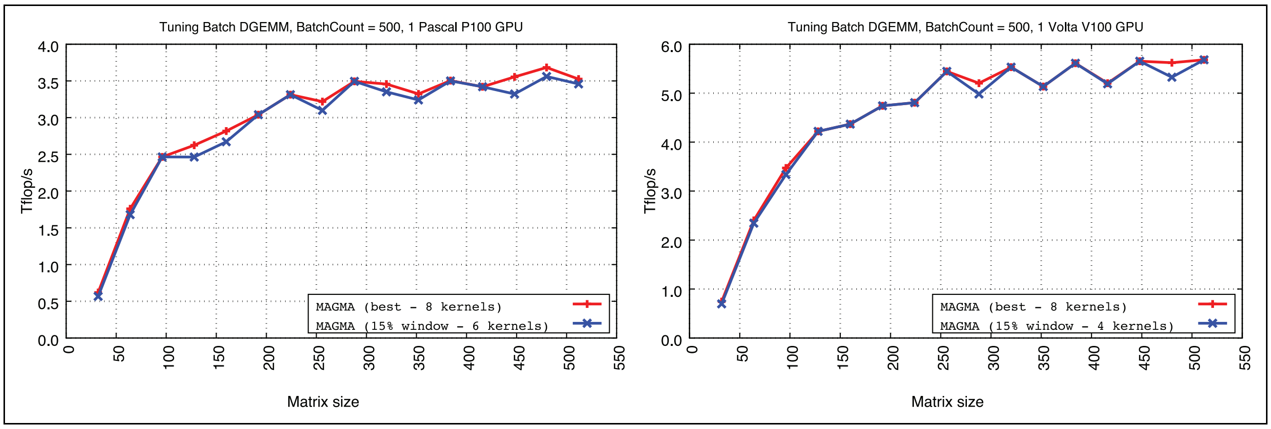


Figure 2. Tuning batched GEMM kernel of MAGMA through MAGMA Templates interface. The results are for the Pascal P100 GPU (left), as well as for the Volta V100 GPU (right).

prunes the search space on the fly, using both the hard constraints and the soft constraints discussed earlier. Obviously, this step should be repeated only if there is a change in the kernel design itself, or if there is a change in the target architecture that would restrict/relax one or more of the constraints.

2. **Performance Evaluation**: This step performs a brute-force sweep over the kernel instances defined in the previous step. For each kernel instance, this step performs both the compilation and the performance evaluation. While the compilation is required once, the performance evaluation can include multiple sizes and test cases. As mentioned earlier, we will limit our discussion to testing on non-transposed square matrices in double precision only. This step should be conducted if the search space is changed. It should also be conducted for every new released architecture.
3. **Result analysis**: This step is basically a post-processing phase of the results collected in step (2). This step usually analyzes the collected data to pick the best performing kernel instance for each test case (size, precision, matrix settings, etc.). In some cases, in particular for library developers, it is important to keep the number of compiled binaries relatively small. This is mainly to avoid large binaries when compiling a library with various routines (such as MAGMA).

Figure 2 shows the best performance observed across the 157 versions of the batched DGEMM kernel on square sizes. The red graph shows the best performance for any given point, while the blue graph shows the performance when we are willing to sacrifice up to 15% of the best observed performance. On both GPUs, we need 8 kernel instances to get the best possible performance at every point. Note that this number is for a particular test case (square sizes) and for a particular precision (double). If we choose to accept up to a 15% drop in performance, we can reduce the number of kernels by 25% for the P100 GPU, and by 50% for the V100 GPU.

4.3 Run-time tuning parameters

This type of parameter is not required to be defined during compilation time as a constant. It can be defined during the exact moment the application is to be launched. A famous example for such parameters is the blocking size (often referred to as *nb*) used in one-sided matrix factorization (LU factorization, QR factorization, and Cholesky factorization). These algorithms are the fundamental components of solving linear systems of equations or least squares problems. The blocking size often affects the performance of the trailing-matrix updates, which often involves a matrix multiplication operation. Generally speaking, the larger the value of nb, the better the performance of the matrix multiplication step, and in turn the entire factorization. However, this also means that the panel factorization step will deal with very wide panels, which is not always a good approach since the panel factorization usually includes memory-bound kernels.

For the MAGMA library backend, hybrid algorithms are usually used in one-sided matrix factorization. The CPU performs the panel factorization step, while the GPU is performing the compute-bound update of the previous iteration. A key performance metric here is that the total execution time to: (1) send the panel to the CPU, (2) perform the factorization, and (3) send the factorized panel back to the GPU, which should be less than or equal to the time of the trailing-matrix update on the GPU. It is clear that there is a performance trade-off with respect to blocking sizes selection. For instance, a very large blocking size may result in the CPU, and thereby the CPU-and-GPU bus interconnect, being the bottleneck, which causes idle times for the GPU, and that leads to a bad performance. The best blocking size may even change from a matrix size to another, and from a GPU architecture to another. This is why auto-tuning is very important.

Experiments that involve run-time tuning parameters are often simpler to perform, since they do not require a compilation of the backend for each set of tuning parameters. One way to perform the tuning sweep is to expose the

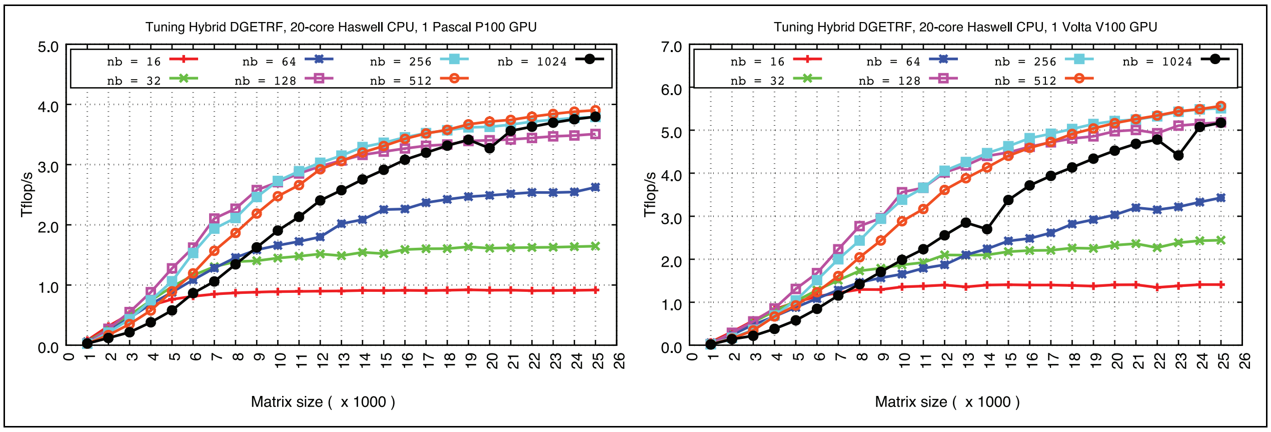


Figure 3. Tuning batched DGETRF kernel of MAGMA through MAGMA Templates interface.

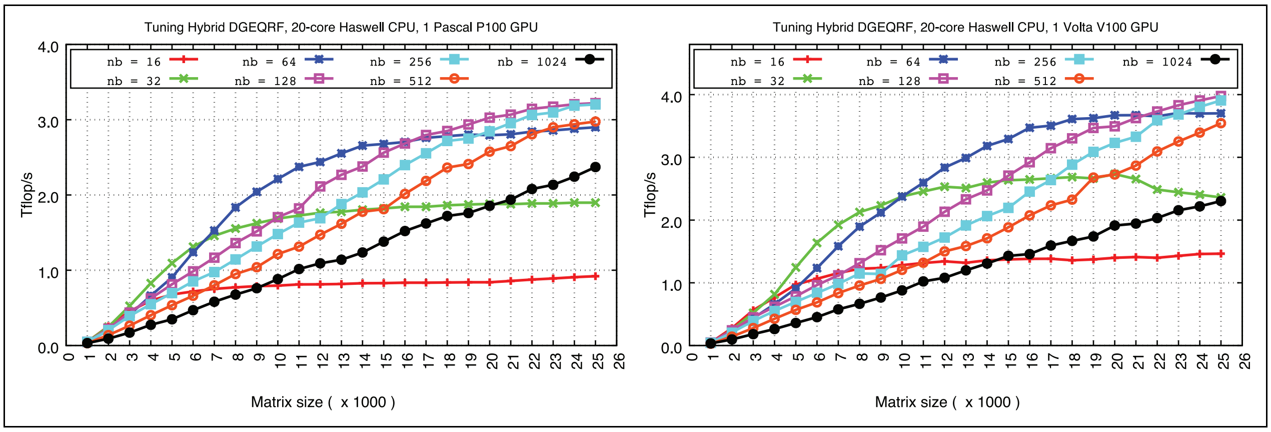


Figure 4. Tuning batched DGEQRF kernel of MAGMA through MAGMA Templates interface.

blocking size *nb* through the high-level interface of each factorization algorithm. Such exposure enables defining *nb* during run time. When the tuning experiment is complete, we roll back the interface to its standard (i.e., LAPACK-compliant) form, and the value of *nb* will be defined through a separate function, as further detailed below.

Figure 3 exhibits the tuning experiment for the LU factorization (DGETRF) on the two GPUs mentioned above. In general, we observe similar behavior for the blocking sizes across both GPUs. Moving from the P100 GPU to the V100 GPU, we observe around a 40% asymptotic performance gain. We also notice that choosing a very large blocking size (e.g., 1024) might have implications for performance. In terms of the best blocking sizes, they are 128, 256, and 512. The only difference between the two GPUs is where to switch from $nb = 256$ to $nb = 512$.

As for the QR factorization (DGEQRF), Figure 4, we can see more noticeable differences between the two systems, especially for the middle range of sizes. The blocking size $nb = 64$ is good for sizes between 7 k and 15 k on the P100 GPU, while it pays off between sizes 10 k and 20 k on the V100 GPU. This is why it is best to have a tunable value of *nb* that can change according to the GPU architecture. The best asymptotic performance is achieved with $nb = 128$. Larger values for *nb* lead to performance drops.

Cholesky factorization, Figure 5, presents a different behavior for the impact of *nb* on performance. In fact, the Cholesky factorization seems to benefit from very large values of *nb*. This is because the Cholesky panel factorization is much simpler than LU and QR panel factorization. This makes the CPU workload minimal with respect to the GPU workload, which always benefits from large matrix sizes.

5 Related work

There have been many efforts in the HPC research community that target building portable APIs across architectures for the software stack, upon which the scientific applications' performance depends. This is indeed due to the fact that the hardware ecosystem keeps changing dramatically in completely unpredictable directions and the programming models may be different across architectures. In addition, the application requirements vary differently based upon the problems that are being addressed, which are highly dependent upon the time and the availability of

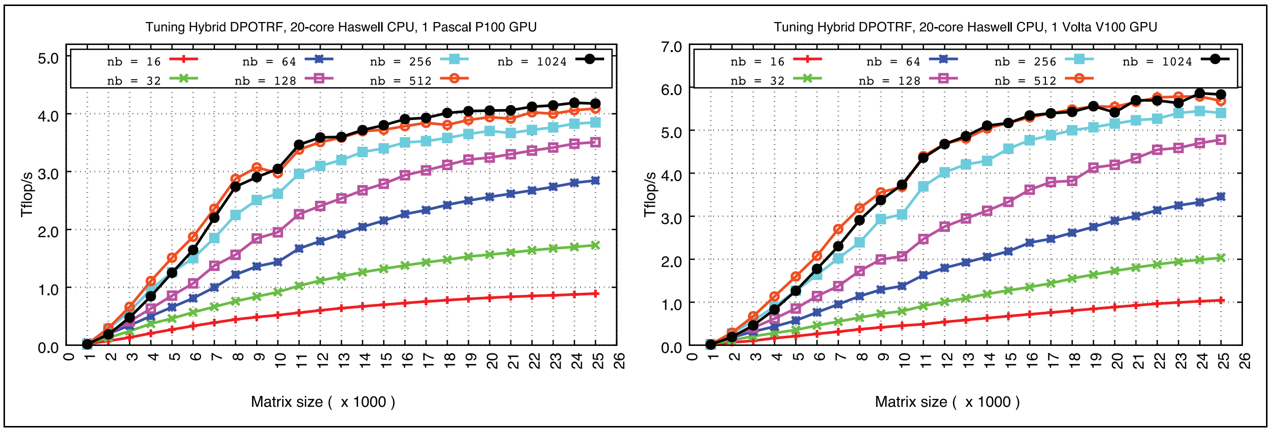


Figure 5. Tuning batched DPOTRF kernel of MAGMA through MAGMA Templates interface.

the computational resources—hardware and software. For example, the interest in machine learning has increased recently due to the significant performance gains brought by the advancements in GPU-driven computing. This is in spite of the fact that artificial intelligence, in particular—on which machine learning algorithms and methodologies are based one way or the other—has been around for decades. In this section, we concisely describe three (closely related to MAGMA Templates) performance-portable layers that are being actively developed.

Kokkos: Portability framework for node-level parallel programming that aims to make performance-centric application development portable across different architectures, irrespective of the underlying hardware (Edwards et al., 2014). Ultimately, Kokkos provides a unified programming model that works for different architectures, similar to OpenMP, OpenACC and OpenCL, with a unique characteristic of preserving the application performance across different hardware.

RAJA: A performance portability layer that leverages the fine-grained parallelism at the node-level with cross-platform support RAJA.

OCCA: An open-source library which aims to make it easy to program different types of devices, provides a unified API for interacting with backend device APIs, and uses just-in-time compilation to build backend kernels (Medina et al., 2014).

Eigen: A high-level C++ linear algebra library based on OOP EIGEN (EIGEN, 2018).

ATLAS: A research effort focusing on figuring out empirical approaches to provide portable performance (Whaley et al., 2001).

BONSAI: A state-of-the-art auto-tuning library for generation and pruning of the parameter search space to find the optimal parameter combinations on a specific architecture (Kurzak et al., 2019b).

All of the aforementioned libraries and frameworks aim to provide performance portable APIs that work on different hardware; however, their performance is driven from the kernels that are included in the libraries. Indeed, at least in the area of dense linear algebra, if a BLAS kernel is coded through a single source (plus code generation for various architectures, or even multiple sources through the use of some polymorphic approach), the resulting performance in general will be much lower than that of vendor-optimized code for each particular architecture (e.g., that most probably would be written in an assembly language, Abalenkovs et al., 2015). Thus, in these cases, while the systems mentioned will achieve functional portability, performance will not be portable unless vendor-optimized BLAS is called, as in the approach adopted in the MAGMA Templates library.

In addition, programming using these libraries requires application developers to change their code significantly to adapt to the requirements of different libraries used. On the contrary, MAGMA Templates provides a thin layer that enables the use of multiple existing libraries through unified function calls; no data structures or memory layouts are required to be changed. Furthermore, the performance is preserved because it is driven from the underlying numerical kernels, which are actively being developed. Finally, the MAGMA Templates interface allows us to easily encapsulate any new kernels or link against new libraries based on the user needs, without changing the application codes; it is a library that is easily pluggable to any application code that uses linear algebra and strives for performance on modern architectures.

A more complete review of different programming models, software technology trends and analysis for their applicability in the area of dense linear algebra can be found in Abdelfattah et al. (2017).

6 Conclusions and future directions

Numerical libraries that can harness the current petascale computing systems to solve today's real world problems may confront severe challenges and a need to embrace opportunities at the dawn of the exascale supercomputers era. Therefore, revising such traditional libraries to leverage their capabilities to extreme-scale settings becomes missioncritical for pushing the edge of what is possible for science and engineering to develop the technology of the future.

This paper presented a performance portable approach for high-performance scalable linear algebra, the backbone for many scientific and engineering applications on current and emerging architectures. The approach uses polymorphism to reuse existing developments and to provide access, seamlessly to the users, to latest algorithms and architecture-specific optimizations through a single, easy-to-use C++ based API. As a proof of concept, the ideas were implemented and illustrated in a new, MAGMA Templates library. The library provides an essential computational linear algebra backend, either for sparse or dense computations, that many production codes need in order to be ported to new architectures. MAGMA Templates created a layered package with APIs that make the routines easily pluggable, extendable, tunable, and interoperable with other numerical libraries and applications.

The main accomplishments leveraged MAGMA's prior developments through a polymorphic approach with a task-based OpenMP+MPI programming model. MAGMA Templates is based on MAGMA versions for various architectures, but provides a single library with a unified interface that is portable across architectures. MAGMA Templates also relies on the SLATE library, which is currently under development at ICL, to provide dense linear algebra routines for distributed-memory systems. The sparse computations component of MAGMA Templates, which is built on the single node functionalities in the MAGMA Sparse library, was extended to distributed-memory heterogeneous systems using domain decomposition–based parallelization. This is significant because it aligns with the wide acquisition of hardware resources that rely on accelerator/wide-vector–based computing, and provides a portable software solution for the strong need for science applications and many production codes to take advantage of the latest hardware assets. Magma Templates also enhances user-productivity since it permits a single application code to run on top of various backend high-performance libraries.

While MAGMA Templates is a proof of concept development, the design is flexible and easy to extend, so future work includes extending the package by adding more functionalities, access to newer algorithms, tuning for new architectures, as well as linking and applying it to applications of interest. Future work also includes adding support for mixed-precision factorizations and solvers, including support for FP16 and use of mixed-precision hardware accelerated FP16 computations, like the Tensor Cores (TC) in Nvidia's V100 GPUs. Some of these routines, including mixed-precision LU factorizations and mixed-precision iterative refinement, are already available through the MAGMA library (Haidar et al., 2017, 2018).

Acknowledgment

We would like to thank NVIDIA for their hardware donations used for development and testing.

Declaration of conflicting interests

The author(s) declared no potential conflicts of interest with respect to the research, authorship, and/or publication of this article.

Funding

The author(s) disclosed receipt of the following financial support for the research, authorship, and/or publication of this article: This project was funded by the DoD High Performance Computing Modernization Program through its User Pro- ductivity Enhancement, Technology Transfer, and Training (PETTT) program.

ORCID iD

Mohammed Al Farhan https://orcid.org/0000-0002-4988-4674

References

Abalenkovs M, Abdelfattah A, Dongarra J, et al. (2015) Parallel programming models for dense linear algebra on heterogeneous systems. *Supercomputing Frontiers and Innovations* 2(4): 67–86. DOI:10.14529/jsfi1504.

Abdelfattah A, Anzt H, Bouteiller A, et al. (2017) Roadmap for the development of a linear algebra library for exascale computing: SLATE: software for linear algebra targeting Exascale. SLATE Working Notes 1, ICL-UT-17-02.

Abduljabbar M, Al Farhan M, Al-Harthi N, et al. (2018) Extreme scale FMM-accelerated boundary integral equation solver for wave scattering. *SIAM Journal on Scientific Computing* 41(3): C245–C268.

Abduljabbar M, Al Farhan M, Yokota R, et al. (2017) Performance evaluation of computation and communication kernels of the Fast Multipole Method on Intel Manycore architecture. In: *23rd international conference on parallel and distributed computing Euro-Par 2017: Parallel Processing, LNCS*, Vol. 10417. Santiago de Compostela, Spain: Springer, pp. 553–564.

Agullo E, Demmel J, Dongarra J, et al. (2009) Numerical linear algebra on emerging architectures: the PLASMA and MAGMA projects. *Journal of Physics: Conference Series* 180: 012037.

Al Farhan MA (2019) Unstructured computations on emerging architectures. DOI:10.25781/KAUST. Available at: http://hdl.handle.net/10754/644902 (accessed 2 September 2019).

Al Farhan MA and Keyes DE (2018) Optimizations of unstructured aerodynamics computations for many-core architectures. *IEEE Transactions on Parallel and Distributed Systems* 29(10): 2317–2332.

Al Farhan MA, Kaushik DK and Keyes DE (2016) Unstructured computational aerodynamics on many integrated core architecture. *Parallel Computing* 59: 97–118. Theory and Practice of Irregular Applications.

Anzt H, Boman E, Dongarra J, et al. (2017) *Magma-Sparse Interface Design Whitepaper*. Technical Report ICL-UT-17-05.

Anzt H, Tomov S and Dongarra J (2014) *Implementing a Sparse Matrix Vector Product for the Sell-C/Sell-C-σ Formats on NVIDIA GPUs*. Technical Report UT-EECS-14-727.

Davis TA and Hu Y (2011) The university of florida sparse matrix collection. *ACM Transactions on Mathematical Software* 38(1): 1:1–1:25.

Edwards HC, Trott CR and Sunderland D (2014) Kokkos: enabling manycore performance portability through polymorphic memory access patterns. *Journal of Parallel and Distributed Computing* 74(12): 3202–3216. Available at: http://www.sciencedirect.com/science/article/pii/S0743731514001257 (accessed 2 September 2019). Domain-Specific Languages and High-Level Frameworks for High-Performance Computing.

EIGEN (2018) Available at: https://eigen.tuxfamily.org/dox-devel/GettingStarted.html (accessed 2 September 2019).

Gates M, Kurzak J, Charara A, et al. (2019) Slate: Design of a modern distributed and accelerated linear algebra library. In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '19. New York, NY, USA: Association for Computing Machinery. ISBN 9781450362290. DOI:10.1145/3295500.3356223.

Gates M, Luszczek P, Abdelfattah A, et al. (2017) *C++ API for BLAS and LAPACK*. Technical Report 2, ICL-UT-17-03. Revision 02-21-2018.

Guo D, Gropp W and Olson LN (2016) A hybrid format for better performance of sparse matrix-vector multiplication on a GPU. *The International Journal of High Performance Computing Applications* 30(1): 103–120.

Haidar A, Tomov S, Dongarra J, et al. (2018) Harnessing gpu tensor cores for fast FP16 arithmetic to speed up mixed-precision iterative refinement solvers. In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*, SC '18. Piscataway, NJ, USA: IEEE Press, pp. 47:1–47:11.

Haidar A, Wu P, Tomov S, et al. (2017) Investigating half precision arithmetic to accelerate dense linear system solvers. In: *SC16 ScalA17: 8th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems*. ACM, Denver, CO: ACM.

Heroux MA, Bartlett RA, Howle VE, et al. (2005) An overview of the trilinos project. *ACM Transactions on Mathematical Software* 31(3): 397–423.

Kurzak J, Gates M, Charara A, et al. (2019a) Least squares solvers for distributed-memory machines with gpu accelerators. In: *Proceedings of the ACM International Conference on Supercomputing*, ICS '19. New York, NY, USA: ACM. ISBN 978-1-4503-6079 -1, pp. 117–126. DOI:10.1145/3330345.3330356.

Kurzak J, Tsai YM, Gates M, et al. (2019b) Massively parallel automated software tuning. In: *Proceedings of the 48th International Conference on Parallel Processing*, ICPP 2019. New York, NY, USA: Association for Computing Machinery. ISBN 9781450362955. DOI:10.1145/3337821.3337908.

Kurzak J, Wu P, Gates M, et al. (2017) Designing SLATE: software for linear algebra targeting Exascale. SLATE Working Notes 3, ICL-UT-17-06.

Li R and Saad Y (2013) GPU-accelerated preconditioned iterative linear solvers. *The Journal of Supercomputing* 63(2): 443–466.

Li Y, Dongarra J and Tomov S (2009) A note on auto-tuning GEMM for GPUs. In: *Proceedings of the 2009 International Conference on Computational Science, ICCS'09*. Baton Roube, LA: Springer.

Liu W and Vinter B (2015) CSR5: an efficient storage format for cross-platform sparse matrix-vector multiplication. In: *Proceedings of the 29th ACM on International Conference on Supercomputing*, ICS '15. New York, NY, USA: ACM. ISBN 978-1-4503-3559 -1, pp. 339–350.

Medina DS, St-Cyr A and Warburton T (2014) OCCA: a unified approach to multi-threading languages. *arXiv preprint arXiv: 1403.0968*.

RAJA (2016) Available at: https://raja.readthedocs.io/en/master/getting_started.html (accessed 2 September 2019).

Tomov S, Dongarra J and Baboulin M (2010) Towards dense linear algebra for hybrid gpu accelerated manycore systems. *Parallel Computing System Application* 36(5-6): 232–240.

Whaley RC, Petitet A and Dongarra JJ (2001) Automated empirical optimizations of software and the ATLAS project. *Parallel Computing* 27(1): 3–35. New Trends in High Performance Computing.

Author biographies

Mohammed Al Farhan received PHD'19 and MS'13 in Computer Science from King Abdullah University of Science and Technology (KAUST), where he was a member of the Extreme Computing Research Center (ECRC). He is currently a Postdoctoral Researcher at the University of Tennessee's Innovative Computing Laboratory (ICL). His research interests include high-performance computing, distributed systems, and numerical linear algebra. Mohammed received BS'12 in Computer Science from King Faisal University.

Ahmad Abdelfattah received his PhD in computer science from King Abdullah University of Science and Technology (KAUST) in 2015, where he was a member of the Extreme Computing Research Center (ECRC). He is currently a research scientist at the Innovative Computing Laboratory, the University of Tennessee. His research interests include numerical linear algebra, HPC on many core architectures, and mixed-precision solvers. Ahmad has B.Sc. and M.Sc. degrees in computer engineering from Ain Shams University, Egypt.

Stanimire Tomov is Research Assistant Professor at the Innovative Computing Laboratory at the University of Tennessee, Knoxville. He specializes in parallel algorithms, data analytics, and high-performance scientific computing. Currently, his work is concentrated on the development of numerical linear algebra software, and in particular the MAGMA libraries, targeting to provide a modernized LAPACK/ScaLAPACK on new architectures. He has lead and contributed to numerous NSF, DOE, and DOD-funded HPC projects and industry collaborations.

Mark Gates is a research assistant professor in the Innovative Computing Laboratory at the University of Tennessee, where he develops algorithms for linear algebra on distributed, multi-core, GPU-accelerated computers. He received his Ph.D. in Computer Science from the University of Illinois at Urbana-Champaign in 2011. His research interests are in scientific computing, including linear algebra, optimization, interpolation, and approximation.

Dalal Sukkari received the MSc and PhD degrees in Applied Mathematics and Computational Science from the King Abdullah University of Science and Technology (KAUST), in 2013 and 2019, respectively, where she was a member of the Extreme Computing Research Center (ECRC) directed by Prof. David Keyes. She is currently a post doctoral research associate in the Innovative Computing Laboratory (ICL), University of Tennessee led by Prof. Jack Dongarra. Her work centers on providing more functionality and applying several optimization techniques to the SLATE (Software for Linear Algebra Targeting Exascale) library.

Azzam Haidar is a Senior Engineer at NVIDIA developing HPC and deep learning software. He received a Ph.D. in 2008 major Computer Science and Applied Mathematics from the National Polytechnic Institute of Toulouse and from the CERFACS Lab, France. Before joining NVIDIA, he was a Research Director at the Innovative Computing Laboratory at the University of Tennessee, Knoxville. His research interests focuses on the development, optimization and implementation of parallel scalable HPC algorithms for distributed multicore/GPU-based architectures, for extreme-scale scientific applications. He is also interested in developing optimized kernels for Deep learning algorithm and studying techniques to speedup the learning process. He also developed novel algorithms for singular value (SVD) and eigenvalue problems as well as approaches that uses data flow representations to express parallelism in scientific applications. He received and participated in several NSF, DOE, Intel and Nvidia grant awards.

Robert Rosenberg is a computational scientist working at the Naval Research Laboratory's Center for Computational Sciences (CCS) in Washington, DC. He has helped users at the Laboratory optimize their codes for supercomputers. He has led the CCS's scientific visualization laboratory and developed an MPI interface program to port serial Fortran codes to MPI on a loop by loop basis. His current interests lie with exploring new architectures such as the NEC's SX-Aurora TSUBASA vector engine as well assisting in refactoring the Navy's latest numerical weather prediction code, NEPTUNE for GPUs.

Jack Dongarra received a Bachelor of Science in Mathematics from Chicago State University in 1972 and a Master of Science in Computer Science from the Illinois Institute of Technology in 1973. He received his Ph.D. in Applied Mathematics from the University of New Mexico in 1980. He worked at the Argonne National Laboratory until 1989, becoming a senior scientist. He now holds an appointment as University Distinguished Professor of Computer Science in the Electrical Engineering and Computer Science Department at the University of Tennessee and holds the title of Distinguished Research Staff in the Computer Science and Mathematics Division at Oak Ridge National Laboratory (ORNL); Turing Fellow at Manchester University; an Adjunct Professor in the Computer Science Department at Rice University. He is the director of the Innovative Computing Laboratory at the University of Tennessee. He is also the director of the Center for Information Technology Research at the University of Tennessee which coordinates and facilitates IT research efforts at the University. He specializes in numerical algorithms in linear algebra, parallel computing, the use of advanced-computer architectures, programming methodology, and tools for parallel computers. His research includes the development, testing and documentation of high quality mathematical software. He has contributed to the design and implementation of the following open-source software packages and systems: EISPACK, LINPACK, the BLAS, LAPACK, ScaLAPACK, Netlib, PVM, MPI, NetSolve, Top500, ATLAS, and PAPI. He has published over 400 articles, papers, reports and technical memoranda and he is coauthor of several books. He was awarded the IEEE Sid Fernbach Award in 2004 for his contributions in the application of high-performance computers using innovative approaches; in 2008 he was the recipient of the first IEEE Medal of Excellence in Scalable Computing; in 2010 he was the first recipient of the SIAM Special Interest Group on Supercomputing's award for Career Achievement; in 2011 he was the recipient of the IEEE Charles Babbage Award; in 2013 he was the recipient of the ACM/IEEE Ken Kennedy Award for his leadership in designing and promoting standards for mathematical software used to solve numerical problems common to high-performance computing, in 2019 he was awarded the SIAM/ACM Prize in Computational Science and Engineering, and in 2020 he received the IEEE Computer Pioneer Award for leadership in the area of high-performance mathematical software. He is a Fellow of the AAAS, ACM, IEEE, and SIAM and a Foreign Member of the Russian Academy of Sciences, a Foreign Fellow of the British Royal Society, and a Member of the US National Academy of Engineering.