Metaprogramming Dense Linear Algebra Solvers Applications to Multi and Many-Core Architectures

Conference Paper · August 2015

DOI: 10.1109/Trustcom.2015.614

Metaprogramming dense linear algebra solvers Applications to multi and many-core architectures

Ian Masliah
University of Paris-Sud and Inria,
F-91405 Orsay, France
Email: ian.masliah@lri.fr

Marc Baboulin
University of Paris-Sud and Inria,
F-91405 Orsay, France
Email: marc.baboulin@lri.fr

Joel Falcou
University of Paris-Sud and Inria,
F-91405 Orsay, France
Email: joel.falcou@lri.fr

***Abstract*—The increasing complexity of new parallel architectures has widened the gap between adaptability and efficiency of the codes. As high performance numerical libraries tend to focus more on performance, we wish to address this issue using a C++ library called NT^2. By analyzing the properties of the linear algebra domain that can be extracted from numerical libraries and combining them with architectural features, we developed a generic approach to solve dense linear systems on various architectures including CPU and GPU. We have then extended our work with an example of a least squares solver based on semi-normal equations in mixed precision that cannot be found in current libraries. For the automatically generated solvers, we report performance comparisons with state-of-the-art codes, and show that it is possible to obtain a generic code with a high-level interface (similar to MATLAB) which runs either on CPU or GPU without generating a significant overhead.**

I. Introduction

A major concern when developing dense linear algebra software is to propose a user-friendly Application Programming Interface (API) that is as performant as BLAS-like [1] optimized routines. Moreover, with the increasing parallelism and heterogeneity as well as the ever increasing data-communication costs, numerical libraries often require to be modified or redesigned in order to take advantage of new features in parallel architectures [2]. In our study we consider the dense linear algebra libraries LAPACK [3] (serial library for CPU processors) and MAGMA [4] (for Graphics Processing Units). The disparity between these libraries that target different architectures illustrate one of the issues in designing optimized linear algebra software. While being able to maintain a similar interface for the routines, the code and structure of all algorithms ported from LAPACK to MAGMA has to be rewritten to match the architectural features and the programming language of the accelerators. Furthermore, these libraries are implemented using low-level languages like C or FORTRAN and thus cannot provide a high-level interface that would be closer to the specification language of the numerical linear algebra practitioner without losing performance. This issue is represented by the *abstraction/efficiency trade-off* problem where raising the abstraction level with object-oriented and generic programming techniques is obtained at the cost of performance. However, performance inhibits the flexibility and adaptability of libraries.

Some solutions have been proposed in recent years but they tend to solve partially the *abstraction/efficiency trade-off* problem. The method followed by the Formal Linear Algebra Methods Environment (FLAME) with the Libflame library [5] is a good example. It offers a framework to develop dense linear solvers using algorithmic skeletons [6] and an API which is more user-friendly than LAPACK, giving satisfactory performance results.

Another method is the one used by code generation projects like Spiral [7] for signal processing or other linear algebra libraries such as Design by transformation [8], Build to Order [9], or Hydra [10] that develop code generation through the use of a *Domain Specific Language* (or DSL) to express data dependencies. A more closely related work is the linear algebra compiler by Fabregat-Traver and Bientinesi [11]. It is based on the Mathematica language and optimizes the algorithms by reusing variables and mapping to BLAS function calls. By definition, a DSL is a computer language specialized for a particular application domain. This implies that the use of a DSL requires a pre-processor and a custom compiler or interpreter. Furthermore, only the DSL compiler is aware of the underlying compiler existence limiting the integration of the DSL with other components such as an Interagred development environment (IDE) [12].Since writing a compiler is very complex to optimize and time-consuming while often not re-usable, we do not wish to do such a task.

A more generic approach is the one followed in recent years by C++ libraries built around *expression templates* [13] or other *generative programming* [14] principles to design a *Domain Specific Embedded Language* (or DSEL). DSEL are languages implemented inside another host language. Designing a DSEL is easier than a DSL as it reuses existing compilers and relies on domain dependent analysis to generate efficient code. Problems such as copy elision and return value optimization that are implemented by compilers are often not exploited by complex DSLs like MATLAB which incurs copy penalities and slow down algorithms. Examples of such libraries are Armadillo [15] and MTL [16]. Armadillo provides good performance with BLAS and LAPACK bindings and an API close to MATLAB [17] for simplicity. However it does not provide a generic solver like the MATLAB routine *linsolve* that can analyze the matrix type and choose the correct routine to call from the LAPACK library. It also does not support GPU computations which are becoming mandatory for medium to large dense linear algebra problems. In a similar way, while MTL can topple the performance of vendor-tuned codes, it does neither *linsolves*-like implementation nor GPU support. Other examples of such libraries include Eigen [18],

Flens [19], Ublas [20] and Blaze [21].

Our objective in this paper is to provide a solution to the problems of *portability* and *adaptability* on new computer architectures. To this end, we propose a hybrid solver for CPU and GPU architectures with a single interface to solve dense linear systems. Our solution is designed on top of NT^2 [22], [23], an open-source scientific library written in C++ available at www.github.com/NumScale/nt2. NT^2 provides a MATLAB-inspired API and its implementation is based on a *meta-programming* technique known as "expression templates" [13]. The contributions of this paper are the following.

- We propose an architecture aware binding between NT^2 and LAPACK/MAGMA based on type tags to dispatch between the different architectures and runtime back-ends in an extensible way.
- We provide an implementation of *linsolve* (in reference to the MATLAB routine) that takes into account both hardware and algorithmic features to select and generate at compile time the proper LAPACK-/MAGMA routine from the high-level C++ code, mapping over 160 kernels. Note that the support of different factorizations (*QR*, *Cholesky*, *LU*, *SVD*) is also provided in NT^2 to facilitate the development of new solver.
- An application based on a linear least squares solver (using mixed precision) that uses of the already available routines for several architectures in NT^2. This application can therefore be written with a single interface and automatically generated for CPU or GPU while showing a level of expressiveness similar to MATLAB.

This paper is organized as follows: in Section II, we describe various programming techniques that combine algorithmic and architectural features in libraries. The methods that we used in NT^2 are then introduced. They enable us to achieve re-use and adaptability of library codes while preserving performance. In Section III-A, we show how to combine these techniques in order to develop efficient dense linear algebra software. Then, as an example of application, we present in Section III-C the code generation of a mixed-precision linear least squares solver for which we give performance comparisons on CPU and GPU using respectively the QR routines from LAPACK and MAGMA. To our knowledge such a solver does not exist in public domain libraries LAPACK, PLASMA [24] and MAGMA. Concluding remarks are given in Section IV.

II. GENERATIVE PROGRAMMING FOR DESIGNING NUMERICAL LIBRARIES

A. Optimization approaches based on a configuration space

As stated in Section I, developing complex linear algebra software is a non trivial task due to the large amount of both algorithmic and architectural requirements. These combined factors create a *configuration space* containing the various configurations available for a given system. Choosing the correct combination of factors from a *configuration space* will then ensure optimal performance.

Compiler techniques based on *iterative compilation* [25], where several optimizations from a *configuration space* are tested and the best one is selected, is a classical technique to improve performance.

An example of these methods can be found in the ATLAS [26] library which is based on using optimized binaries. Each function's binary is generated during the installation phase with the *iterative compilation* technique. The generation process is accelerated by a hierarchical tuning system. In this system, the lower level functions are subject to a large selection process ensuring their optimal performance. High-level functions like BLAS 3 routines can then exploit feedback from the previous steps of the configuration process.

A second method is based on a performance analysis at runtime. For instance, a system like StarPU [27] uses a monitored runtime system in which the performance of each function on a given hardware configuration is monitored in real-time. This monitoring allows StarPU to select the most optimized version of the algorithm by changing its parameters (tiling size, number of iterations,...) or the targeted architecture (CPU, GPU, hybrid).

Both methods described above are valid approaches in the field of high performance computing. However, in our case, we aim at providing a library level system for such exploration [28] that will complement the compilers work. One way to do this is to use *generative programming*.

B. Generative programming in software development

Generative programming consists of bringing the benefits of automation to software development. Following this paradigm, a model can be drawn to implement the different components [29] of a system. It is then possible to build a generator that will combine these components based on a generative domain model. This generator (or *configuration knowledge*) will ensure the transition from a *configuration space* with domain-specific concepts [30] and features to a *solution space* that encapsulates expertise at the source-code level. The code generation process will be hidden from the end-user by various *meta-programming* techniques which turn the user interface into a simple and clean API, where few to none details about the algorithms and structures are visible.

Template *meta-programming* is a classical *generative programming* technique in which templates are used by a compiler to generate temporary source code. It is then merged with the rest of the source code and finally compiled. The output of these templates includes compile-time constants, data structures, and functions. The use of templates can be thought of as a compile-time execution that enables us to implement domain-specific optimizations. The technique is used by a number of languages, the most well-known being C++ [31], D [32], Haskell [33] and OCaml [34].

C. Domain engineering methods for active libraries

We call *active libraries* [35] a technique which combines a set of *generative programming* and *meta-programming* methods to solve the *abstraction/efficiency trade-off problem* mentioned in Section I. The main idea is to perform high-level optimization based on a semantic analysis of the code before any real compilation process. Such informations and transformations are then carried on by a meta-language that allows the developer to embed meta-informations in the source code itself, helping compilers to generate a better code by using these semantic informations. Active libraries are often implemented as DSEL.

Czarnecki proposed a methodology called *Domain Engineering Method for Reusable Algorithmic Libraries* [36] (or DEMRAL depicted in the blocks 1-3 of Figure 1) based on the techniques described previously. DEMRAL is a DSEL-based method where domain specific descriptors are used to represent the various states of the system (represented as Block 1 in Figure 1). The various combinations of descriptors represent all the possible configurations in the system. In NT^2, these parameters are available at the API level for the user. Once these configurations have been implemented, it is possible to program the parametric components that they represent (see Block 2 in Figure 1). In NT^2, these would correspond to the various skeletons available and various kernels for the CPU. The final step is to build a generator that will take the various descriptors as parameters, choose the corresponding component and generate the concrete application at compile-time based on this (see Block 3 in Figure 1). In NT^2, this corresponds to the solver we have implemented which will be described later on.

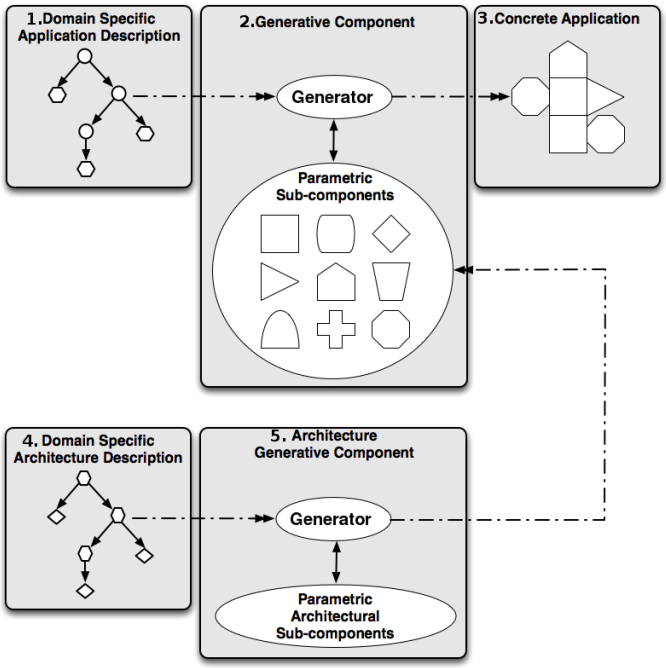

Fig. 1: Overview of the AA-*DEMRAL* process

DEMRAL can be seen as a specialization of a paradigm like object-oriented programming, aspect programming or model driven engineering [37]. While we can find a large number of algorithms (N) and implementations for distinct data structures (P), the problem is that combining them can result in a large number of code to write ($N * P$). Using DEMRAL, only N generic algorithms and P data structure descriptions are needed since the generator will choose the correct domain-specific implementation from the *configuration space* with the help of the *configuration knowledge*.

The DEMRAL methodology provides a high re-usability, allowing components to be customized while retaining the efficiency of statically configured code [38]. We extend it by adding an architectural layer in the design with a "Domain specific architecture description" (block 4 of Figure 1) and a specialized generator for GPUs (block 5 of Figure 1) based on this description [39]. In $NT2$, this would represent an extension to the API with a GPU tag and the addition of GPU based skeletons and kernels based on the MAGMA library. This enables us to have a separate specific generator for accelerators that will create a generic component with the appropriate marker that can then be combined with the already existing ones.

III. Application to linear algebra solvers

In this section, we describe our approach to automatically generate linear algebra solvers on parallel architectures. Our solution stems from the programming techniques combined with a proper *configuration space* and *smart containers* for data management on GPU.

A. Linear system solvers

The first step to build *linsolve* is to identify the key properties of the *configuration space* and the proper way to represent them. Once this analysis is done, we can refine these properties into high-level abstractions that will parameterize *linsolve*. These abstractions will then be used to define the *configuration knowledge* necessary to ensure the transition to the *solution space*. These properties represent the informations necessary to dispatch on the various solvers that we can find in the numerical libraries LAPACK and MAGMA. Concerning dense linear systems, we can identify three main properties that need to be taken into account: matrix structure, condition number, and targeted architecture.

A matrix structure can be divided into subcategories that can be identified statically (data type, storage scheme, matrix type and storage format). The data type and storage scheme parameters are already identified through the problem domain, respectively being scalar entries (real, double or single/double complex) and a dense matrix. The storage format is defined by NT^2 and shares a common interface with FORTRAN77 (column-major arrays). The matrix types correspond to the different ones available in the numerical libraries LAPACK and MAGMA (*e.g., general, symmetric, hermitian...*).

The second domain corresponds to the conditioning of the system. In current numerical libraries, the linear solvers are usually based on LU or QR factorizations in fixed precision, or mixed-precision algorithms [40] with iterative refinement. It is not possible to identify statically if a system

is ill-conditioned since it requires expensive computations which are not manageable at compile-time. Furthermore, it would be too costly to estimate the condition number at runtime for mixed-precision routines since it requires the factored form of the matrix (the LAPACK function `gecon` estimates the reciprocal condition number but requires the LU form bringing the cost to $\theta(n^3)$ for an $n*n$ matrix). However, since current dense linear algebra libraries propose mixed precision routines, it needs to be part of the *configuration space*.

The last key domain of our solver is the dispatch between different architectures. As explained in Section I, the architectural features of a GPU result in a very different language compared to a CPU. The solution we used to solve this abstraction problem is to provide through the use of a DSEL (Section II-B) a common syntax between CPU and GPU routines. Using architecture aware binding, we can then freely decide whether to call LAPACK or MAGMA routines by dispatching on the different back-ends in an extensible way. It is now possible to define a grammar that encapsulates these ideas into a *configuration space* (see Section II-A). These parameters are not mutually exclusive and can be extended/combined.

TABLE I: Configuration space parameter levels

0-Matrix type	general | band | diagonal | symmetric | positive definite
1-Data type	float | double | single/double complex
2-Precision	fixed | mixed-precision
3-Conditioning	no information | ill-conditioned
4-Storage scheme	general | packed
5-Architecture	CPU | GPU

Most of the parameters we can access are defined by the user and therefore configurable at the API level. In MATLAB, the *linsolve* routine does not take into account the data type, and the matrix type needs to be defined in a parameter structure containing the different matrix properties recognized (lower/upper triangular, upper Hessenberg, symmetric, positive definite, rectangular). While creating a matrix in NT^2, the user has the possibility to define the matrix and data type which are optimized as meta-data properties of the matrix, using the following instruction:

```
nt2::table<double,nt2::symmetric_> a;
```

When calling *linsolve*, he will then have the possibility to give additional information on the conditioning of the matrix either as a parameter of the system:

```
x = nt2::linsolve(a,b,nt2::ill_conditioned_);
```

or of the matrix:

```
x = nt2::linsolve(nt2::ill_conditioned_(a),b);
```

It is also possible to ask for complementary information as output like the reciprocal condition number returned by LAPACK :

```
nt2::tie(x,r) = nt2::linsolve(a,b);
```

Once this is done, *linsolve* will be able to parse the *configuration space* by reading out the nested domain-specific features while assigning default values to the unspecified ones.

Figure 2 shows the various steps to perform a call to linsolve with a symmetric matrix (here of size 5000) in NT^2 linked with the MAGMA library. The user starts by defining the entry and output matrices with the correct description as seen in Part 7 of Figure 2. When the code is analyzed by the C++ compiler, the call corresponding to Part 1 of Figure 2 will end up triggering the generation phase. The routine in Part 6 is generated using a combination of the parameters mentioned in Parts 2 to 5. This routine can be a LAPACK/MAGMA kernel (if available, which is the case of Figure 2) or a kernel directly implemented in NT^2. After the C++ compilation phase, we obtain a code similar to the one given in Part 8.

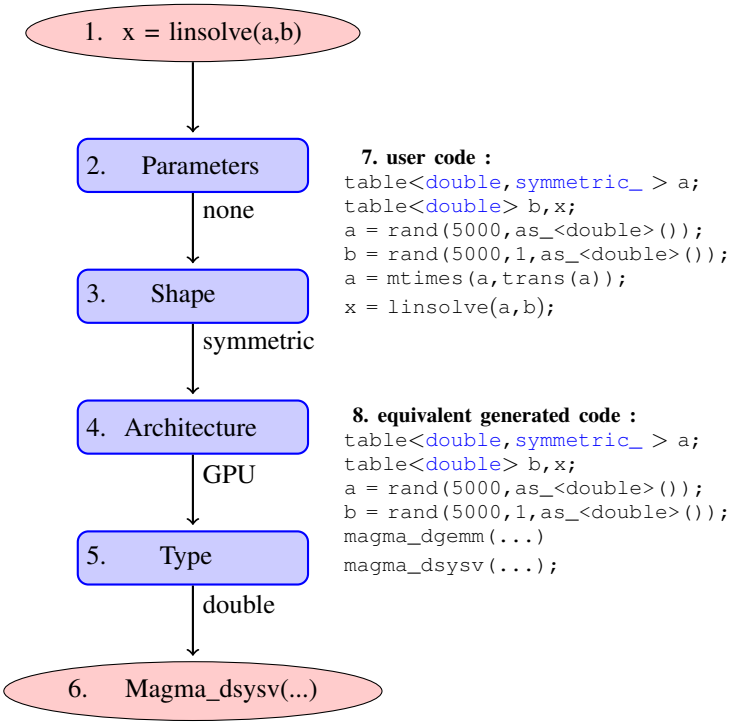

Fig. 2: Example of a generation process for a symmetric system

B. Memory management for hybrid computation

When using GPU-based systems, we need to ensure data consistency between the different physical memories. In this section, we discuss how these techniques are used jointly with *linsolve*. We can discern the two most common approaches to CPU and GPU containers. The first one is to statically define the locality of the container which is done in the Thrust library [41], while the second uses a dynamic approach like in SkePU [42].

The memory management mechanism in a dynamic approach allows to change the locality of a container and reallocate the data. The container then needs to manage the

memory and ensure consistency between data and locality. In this situation, it is not possible to statically define the locality of a container. Therefore, our approach consists of adding an architectural tag similarly to the matrix type tag on our container (default locality is CPU). The purpose of this method is to enable the user to write programs using GPU functions in a transparent way.

```
nt2::table<double,nt2::gpu_> a;
```

It is then possible to ensure the transitions from CPU to GPU memory by using explicitly the tag. This does not prevent the decision-making process of the solver when no locality tag is given by the user. The solver can generate a GPU code performing data transfers from CPU to GPU as well as the reverse. The generation process will choose the architecture based on a combination of factors, mainly the matrix size and the algorithm. The GPU tag can also hold complementary informations passed as template parameters of the tag.

The definition of container locality being static, it is easier to define a data efficient memory management unit. Let's use the following scenario as an example :

```
x = nt2::linsolve(a,b);
```

From here, we can apply different strategies depending on the locality of x, a and b. In a situation where all three containers are on GPU (respectively CPU) memory there will be no locality problem as various data are located on the same device. However, the scenario where x is on the GPU (respectively CPU) while a and b are on the CPU (resp. GPU) will generate a conflict. The rules to solve locality conflicts are static and do not depend on the runtime. Therefore, the priority will be given to the locality of the result x to ensure consistency between the data and container locality.

Experiments were carried out on a system using 2 sockets of Intel Xeon E5645 2.40GHz and a Tesla C2075. We consider single precision random square matrices of size 2000, 10000 and 20000 and we solve a system of linear equations $Ax = b$ using the LU factorization. The light grey bars in Figure 3 correspond to the following call made in NT^2 through *linsolve* that can run either on CPU or GPU.

```
x = nt2::linsolve(a,b);
```

The dark grey bars correspond to C++ calls to either the LAPACK function `sgesv` or the MAGMA function `magma_sgesv`. These results show that automatically generated routines do not exhibit any overhead compared to direct calls to LAPACK or MAGMA. Note that the performance of all others generated routines does not incur any overhead as well.

C. Application to linear least squares

In this section we illustrate how the *generative programming* method described in Section II can be used to generate automatically new implementations of algorithms and achieve satisfactory performance.

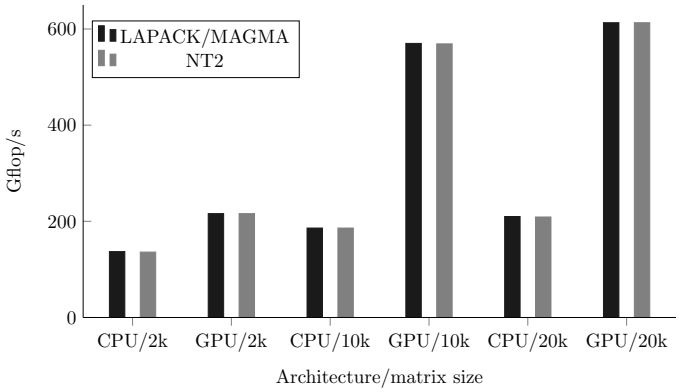

Fig. 3: Performance comparaison between LAPACK/MAGMA routines and generated codes via NT2 for general dense linear system solution

1) Solving least squares by semi-normal equations: We consider the overdetermined full rank linear least squares (LLS) problem $\min_{x\in\mathbb{R}^n} \|Ax - b\|_2$, with $A \in \mathbb{R}^{m\times n}, m \geq n$ and $b \in \mathbb{R}^m$.

The most classical methods for solving linear least squares problems are based on the QR factorization or the normal equations. The latter method is twice cheaper (mn^2 vs $2mn^2$ operations) but the error is then proportional to $\mathrm{cond}(A)^2$ [43, p. 49]). However if A can be saved, we can also use the semi-normal equations (SNE) method where we solve the system

$$R^T Rx = A^T b,$$

where R is the triangular factor from the QR factorization of A (this is a straightforward reformulation of the normal equations). It is shown in [44] that, similarly to the normal equations method, the forward error bound involves a factor $cond(A)^2$, even if we use a R-factor that is of better quality than the Cholesky factor because it has been computed via a backward stable algorithm. However, as explained in [43, p. 126 and p. 250], the accuracy of the SNE method can be improved by using the corrected semi-normal equations method (CSNE) that consists in adding one step of fixed precision iterative refinement to the SNE as follows:

1) Let $\tilde{x}$ solve $R^T Rx = A^T b$
2) Compute $\tilde{r} = b - A\tilde{x}$
3) Solve $R^T Rw = A^T \tilde{r}$
4) Corrected solution $y = \tilde{x} + w$

It is shown in [45, p. 392] that, if $\mathrm{cond}(A)^2 u \leq 1$ (u being the unit roundoff), then the forward error bound for the CSNE method is similar to that of a backward stable method (and even smaller when $r = Ax - b$ is small). In that case, the CSNE method is more satisfactory than the SNE method but this is not true for all A. In the following we propose to use the CSNE method to solve LLS in mixed precision.

2) Mixed-Precision Corrected Semi-Normal Equation: The efficiency of mixed precision algorithms has been proved on linear systems based on the LU factorization with results that can reach up to 90% [40] of floating point computational rate in the lowest precision on current architectures. The method

to solve mixed precision CSNE (or MCSNE) consists of first performing the factorization in single precision (ε_s) (if the matrix is not too ill-conditioned) with the computational cost of $\theta(mn^2)$ and then refine the solution in double precision (ε_d) where operations cost $\theta(n^2)$. Iterative refinement [46] is a method that produces a correction to the computed solution by iterating on it. Each k^{th} iteration in this process consists of computing the residual $r_k = b - Ax_{k-1}$, solving the new system $Ad_k = r_k$, and adding the correction $x_{k+1} = x_k + d_k$. Mixed precision iterative refinement will work as long as the condition number of the least squares problem [47] is smaller than the inverse of the lower precision used (*i.e.* here 10^8).

Algorithm 1 Mixed-Precision CSNE

Compute $A = QR$	(ε_s)
Solve $R^T x = A^T b$	(ε_s)
Solve $Rx_0 = x$	(ε_s)
`do k = 1,2,...`	
$r_k = b - Ax_{k-1}$	(ε_d)
Solve $R^T x = r_k$	(ε_s)
Solve $Rd_k = x$	(ε_s)
$x_k = x_{k-1} + d_k$	(ε_d)
`check convergence`	

The first step of Algorithm 1 in NT^2 is implemented in line 8 of Figure 4, while the second and third steps are performed by two calls to *linsolve* in lines 11 and 12. Note that the code is similar in terms of syntax and number of instructions to what would be written in MATLAB.

```
1  table<double> mcsne(table<double> const& A, table<double>
       const& B)
2  {
3    double anrm = lange(A,'I');
4    double cte  = anrm*Eps<double>()*nt2::sqrt(width(a));
5
6    table<float> SA = cast<float>(A);
7
8    table<float,upper_triangular_> SR = triu(qr(SA));
9    table<float> SX = mtimes(trans(SA),cast<float>(B)));
10
11   SX = linsolve(trans(SR),SX);
12   SX = linsolve(SR       ,SX);
13
14   table<double> X = cast<double>(SX);
15   table<double> E = B - mtimes(A,X);
16
17   std::size_t i = 0;
18
19   do
20   {
21     SX = cast<float>(mtimes(trans(A),cast<float>(E)));
22     SX = linsolve(trans(SR),SX);
23     SX = linsolve(SR       ,SX);
24
25     E = cast<double>(SX)
26
27     double RNRM = maximum(abs(E(_)));
28
29     X += E;
30     double XNRM = maximum(abs(X(_)));
31
32     E = B - mtimes(A,X);
33     i++;
34   } while( !(RNRM < XNRM*cte) && (i<max_iter));
35
36   return X;
37 }
```

Fig. 4: NT2 implementation for MCSNE

Once the solver for MCSNE is implemented using NT^2, it becomes possible to add it to *linsolve* as a dispatch case of mixed precision solver for overdetermined linear systems. This would result in the following call :

```
x = nt2::linsolve(a,b,nt2::mixed_precision_);
```

3) Performance results for MCSNE: Benchmarks were carried out using 2 sockets of Intel Xeon E5645 2.40GHz (peak Gflop/s is 230) and a Tesla C2075 (peak Gflop/s is 1030.4). We used Intel MKL [48] version 10.2.3, MAGMA 1.3 with CUDA 5.0 [49] and gcc 4.8 [50]. The random test problems were generated using the method described in [51]. Performance results include data transfers between CPU and GPU and data.

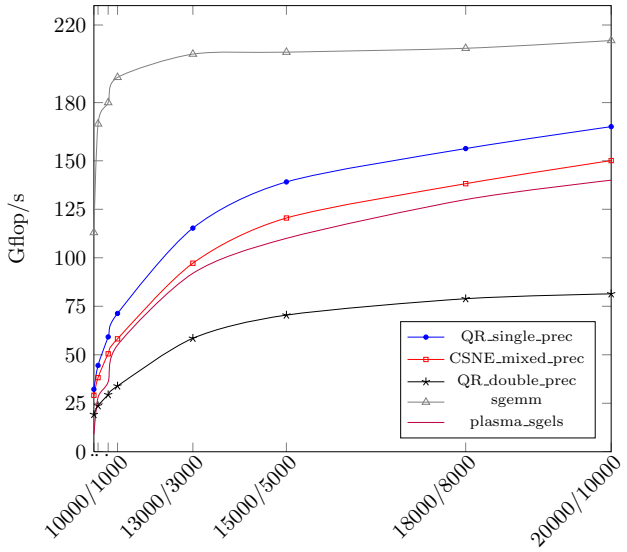

Fig. 5: Performance results of Generated code on CPU of gels (QR solver) and mcsne

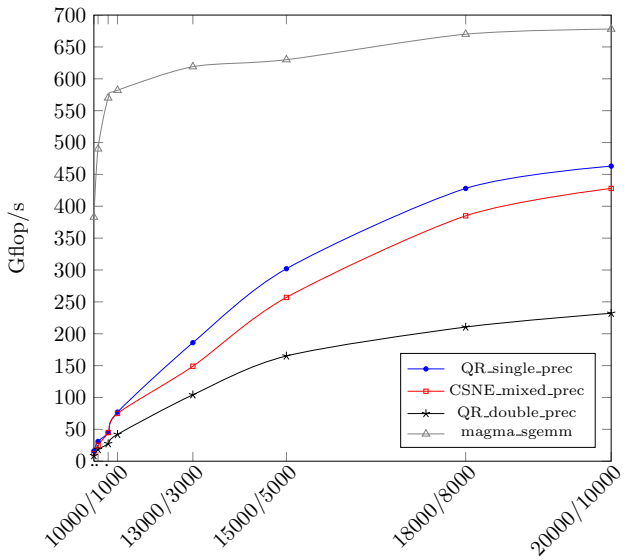

Fig. 6: Performance results of Generated code on GPU of gels (QR solver) and mcsne

In Figure 5 we compare the performance of MCSNE on the CPU with the LAPACK routine `xgels` that solves the LLS problem with a QR factorization without column pivoting. The results show that performance of MCSNE is only 10% less that the rate of `sgels`. Note that this represents around 75% of the peak performance of a matrix-matrix multiply in single precision (routine `sgemm`). We use random matrices and the iterative refinement converged in less than 4 iterations.

On the GPU, the performance of MCSNE in NT^2 is depicted in Figure 6. It also approaches 90% of the performance of **magma_sgels** on GPU while being near twice faster than the routine in double precision. The behavior of MCSNE when compared with QR solvers in double and single precision is similar to what was observed in [52, p. 15] for the LU factorization.

IV. Conclusion

Combining the large number of algorithms available in numerical libraries and architectural requirements in a generic solver for dense linear systems is a complex task. We showed that *generative programming* is a valid software development approach for addressing these issues while maintaining a high level of performance. Our contribution furthers the work in *active libraries* by providing a viable way to make our software architecture-aware. Performance results illustrate that for both existing routines like those in *linsolve* and new ones such as MCSNE, the delivered performance is close to what state of the art libraries achieve.

The other interesting result is that software like NT^2 can quickly prototype new algorithms while providing support for various architectures. With NT^2, we reach a good combination of high-level codes for linear algebra problems that gives good speedups and offers the users enough expressiveness to describe the problem in the most efficient way.

Future work includes support for more architectures like Intel Xeon Phi, with work on new algorithms that provide good performances while not being available in numerical libraries like randomized algorithms [53], [54] or communication-avoiding algorithms [55] for dense linear systems. Moving to sparse problems is also a possibility where libraries like Cups [56] or VexCL [57] provide an interesting approach. Raising the level of expressiveness stays a major concern while trying to add content in NT^2.

The code generator and examples described in this paper with NT^2 are available at www.github.com/NumScale/nt2.

References

[1] J. Dongarra, "Basic Linear Algebra Subprograms Technical Forum Standard," *Int. J. of High Performance Computing Applications*, vol. 16, no. 1, 2002.

[2] P. Luszczek, J. Kurzak, and J. Dongarra, "Looking back at dense linear algebra software," *Journal of Parallel and Distributed Computing*, 2013.

[3] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen, *LAPACK Users' Guide*, 3rd ed. Philadelphia: SIAM, 1999.

[4] S. Tomov, J. Dongarra, and M. Baboulin, "Towards dense linear algebra for hybrid GPU accelerated manycore systems," *Parallel Computing*, vol. 36, no. 5&6, pp. 232–240, 2010.

[5] F. G. Van Zee, E. Chan, R. A. Van de Geijn, E. S. Quintana-Orti, and G. Quintana-Orti, "The libflame library for dense matrix computations," *Computing in science & engineering*, vol. 11, no. 6, pp. 56–63, 2009.

[6] M. I. Cole, *Algorithmic skeletons: structured management of parallel computation*. Pitman London, 1989.

[7] M. Püschel, J. M. Moura, B. Singer, J. Xiong, J. Johnson, D. Padua, M. Veloso, and R. W. Johnson, "Spiral: A generator for platform-adapted libraries of signal processing alogorithms," *International Journal of High Performance Computing Applications*, vol. 18, no. 1, pp. 21–45, 2004.

[8] B. Marker, J. Poulson, D. Batory, and R. van de Geijn, "Designing linear algebra algorithms by transformation: Mechanizing the expert developer," in *High Performance Computing for Computational Science-VECPAR 2012*. Springer, 2013, pp. 362–378.

[9] J. G. Siek, I. Karlin, and E. R. Jessup, "Build to order linear algebra kernels," in *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*. IEEE, 2008, pp. 1–8.

[10] S. Lewis, A. Csordas, S. Killcoyne, H. Hermjakob, M. R. Hoopmann, R. L. Moritz, E. W. Deutsch, and J. Boyle, "Hydra: a scalable proteomic search engine which utilizes the hadoop distributed computing framework," *BMC bioinformatics*, vol. 13, no. 1, p. 324, 2012.

[11] D. Fabregat-Traver and P. Bientinesi, "A domain-specific compiler for linear algebra operations," in *High Performance Computing for Computational Science-VECPAR 2012*. Springer, 2013, pp. 346–361.

[12] M. Fowler, "Language workbenches: The killer-app for domain specific languages," http://www.issi.uned.es/doctorado/generative/Bibliografia/Fowler.pdf, 2005.

[13] T. Veldhuizen, "Expression templates," *C++ Report*, vol. 7, pp. 26–31, 1995.

[14] K. Czarnecki, K. Østerbye, and M. Völter, "Generative programming," in *Object-Oriented Technology ECOOP 2002 Workshop Reader*. Springer, 2002, pp. 15–29.

[15] S. Conrad, "Armadillo: An open source C++ linear algebra library for fast prototyping and computationally intensive experiments," NICTA, Australia, Tech. Rep., October 2010.

[16] P. Gottschling, D. S. Wise, and M. D. Adams, "Representation-transparent matrix algorithms with scalable performance," in *ICS '07: Proceedings of the 21st annual international conference on Supercomputing*. New York, NY, USA: ACM Press, 2007, pp. 116–125.

[17] MATLAB, *version 8.10.0 (R2010a)*. Natick, Massachusetts: The MathWorks Inc., 2013.

[18] G. Guennebaud, B. Jacob *et al.*, "Eigen v3," http://eigen.tuxfamily.org, 2010.

[19] M. Lehn, "FLENS," http://http://www.mathematik.uni-ulm.de/~lehn/FLENS/, 2013.

[20] J. Walter and M. Koch, "The boost uBLAS library," http://www.boost.org/libs/numeric/ublas, 2002.

[21] K. Iglberger, G. Hager, J. Treibig, and U. Rüde, "Expression templates revisited: a performance analysis of current methodologies," *SIAM Journal on Scientific Computing*, vol. 34, no. 2, pp. C42–C69, 2012.

[22] P. Esterie, J. Falcou, M. Gaunard, J. T. Lapresté, and L. Lacassagne, "The numerical template toolbox: A modern C++ design for scientific computing," *Journal of Parallel and Distributed Computing*, 2014.

[23] J. Falcou, J. Sérot, L. Pech, and J. T. Lapresté, "Meta-programming applied to automatic SMP parallelization of linear algebra code," in *Euro-Par 2008–Parallel Processing*. Springer, 2008, pp. 729–738.

[24] U. Tennessee, "PLASMA users' guide, parallel linear algebra software for multicore architectures, version 2.3," http://icl.cs.utk.edu/plasma/software/, 2010.

[25] S. Triantafyllis, M. Vachharajani, N. Vachharajani, and D. I. August, "Compiler optimization-space exploration," in *Code Generation and Optimization, 2003. CGO 2003. International Symposium on*. IEEE, 2003, pp. 204–215.

[26] R. C. Whaley and J. Dongarra, "Automatically tuned linear algebra software," in *Proceedings of the 1998 ACM/IEEE Conference on Supercomputing*. IEEE Computer Society, 1998, pp. 1–27.

[27] C. Augonnet, S. Thibault, R. Namyst, and P. Wacrenier, "StarPU: a unified platform for task scheduling on heterogeneous multicore architectures," *Concurrency and Computation: Practice and Experience*, vol. 23, no. 2, pp. 187–198, 2011.

[28] T. L. Veldhuizen and E. Gannon, "Active libraries: Rethinking the roles of compilers and libraries," in *Proceedings of the SIAM Workshop on Object Oriented Methods for Inter-operable Scientific and Engineering Computing (OO98)*. SIAM Press, 1998.

[29] K. Czarnecki and U. W. Eisenecker, "Components and generative programming," in *Software EngineeringESEC/FSE99*. Springer, 1999, pp. 2–19.

[30] D. Gregor, J. Järvi, J. Siek, B. Stroustrup, G. Dos Reis, and A. Lumsdaine, "Concepts: linguistic support for generic programming in C++," in *ACM SIGPLAN Notices*, vol. 41. ACM, 2006, pp. 291–310.

[31] D. Abrahams and A. Gurtovoy, *C++ template metaprogramming: concepts, tools, and techniques from Boost and beyond*. Addison-Wesley Professional, 2004.

[32] W. Bright, "D language Templates revisited," http://dlang.org/templates-revisited.html.

[33] T. Sheard and S. P. Jones, "Template Meta-programming for Haskell," *SIGPLAN Not.*, vol. 37, no. 12, pp. 60–75, Dec. 2002. [Online]. Available: http://doi.acm.org/10.1145/636517.636528

[34] W. Taha, "A gentle introduction to multi-stage programming," in *Domain-Specific Program Generation*. Springer, 2004, pp. 30–50.

[35] K. Czarnecki, U. Eisenecker, R. Glück, D. Vandevoorde, and T. Veldhuizen, "Generative programming and active libraries," in *Generic Programming*. Springer, 2000, pp. 25–39.

[36] K. Czarnecki and U. W. Eisenecker, *Generative Programming: Methods, Tools, and Applications*. New York, NY, USA: ACM Press/Addison-Wesley Publishing Co., 2000.

[37] D. C. Schmidt, "Guest editor's introduction: Model-driven engineering," *Computer*, vol. 39, no. 2, pp. 0025–31, 2006.

[38] D. R. Musser, G. J. Derge, and A. Saini, *STL tutorial and reference guide: C++ programming with the standard template library*. Addison-Wesley Professional, 2009.

[39] P. Estérie, "Multi-architectural support: A generic and generative approach," Ph.D. dissertation, Paris 11, 2014.

[40] M. Baboulin, A. Buttari, J. Dongarra, J. Kurzak, J. Langou, P. Luszczek, and S. Tomov, "Accelerating scientific computations with mixed precision algorithms," *Computer Physics Communications*, vol. 180, no. 12, pp. 2526–2533, 2009.

[41] N. Bell and J. Hoberock, "Thrust: A productivity-oriented library for CUDA," *GPU Computing Gems*, vol. 7, 2011.

[42] J. Enmyren and C. W. Kessler, "SkePU: a multi-backend skeleton programming library for multi-GPU systems," in *Proceedings of the fourth international workshop on High-level parallel programming and applications*. ACM, 2010, pp. 5–14.

[43] A. Björck, *Numerical methods for least squares problems*. Siam, 1996.

[44] ——, "Stability analysis of the method of semi-normal equations for least squares problems," *Linear Algebra and its Applications*, vol. 88/89, pp. 31–48, 1987.

[45] N. J. Higham, *Accuracy and Stability of Numerical Algorithms*, 2nd ed. Philadelphia: SIAM, 2002.

[46] J. Demmel, Y. Hida, W. Kahan, X. S. Li, S. Mukherjee, and E. J. Riedy, "Error bounds from extra-precise iterative refinement," *ACM Trans. Math. Softw.*, vol. 32, no. 2, pp. 325–351, 2006.

[47] M. Baboulin, J. Dongarra, S. Gratton, and J. Langou, "Computing the conditioning of the components of a linear least-squares solution," *Numerical Linear Algebra with Applications*, vol. 16, no. 7, pp. 517–533, 2009.

[48] Intel, "Math Kernel Library (MKL)," http://www.intel.com/software/products/mkl/.

[49] *NVIDIA CUDA C Programming Guide*, NVIDIA, 04/16/2012, version 4.2.

[50] B. J. Gough and R. M. Stallman, *An Introduction to GCC*. Network Theory Ltd., 2004.

[51] C. C. Paige and M. A. Saunders, "LSQR: An algorithm for sparse linear equations and sparse least squares," vol. 8, no. 1, pp. 43–71, 1982.

[52] M. Baboulin, "Fast and reliable solutions for numerical linear algebra solvers in high-performance computing," http://tel.archives-ouvertes.fr/tel-00967523, 2012, habilitation thesis - University of Paris-Sud.

[53] M. Baboulin, D. Becker, and J. Dongarra, "A Parallel Tiled Solver for Dense Symmetric Indefinite Systems on Multicore Architectures," in *Proceedings of IEEE International Parallel & Distributed Processing Symposium (IPDPS 2012)*, 2012, pp. 14–24.

[54] M. Baboulin, J. Dongarra, J. Herrmann, and S. Tomov, "Accelerating linear system solutions using randomization techniques," *ACM Trans. Math. Softw.*, vol. 39, no. 2, 2013.

[55] M. Baboulin, S. Donfack, J. Dongarra, L. Grigori, A. Rémy, and S. Tomov, "A class of communication-avoiding algorithms for solving general dense linear systems on CPU/GPU parallel machines," in *International Conference on Computational Science (ICCS 2012)*, ser. Procedia Computer Science, vol. 9. Elsevier, 2012, pp. 17–26.

[56] N. Bell and M. Garland, "Cusp: Generic parallel algorithms for sparse matrix and graph computations," http://cusp-library.googlecode.com, 2012, version 0.3.0.

[57] D. Demidov, "VexCL: Vector expression template library for OpenCL," http://github.com/ddemidov/vexcl, 2012.