NNT : 2016SACLS285

Thèse de doctorat de l'Université Paris-Saclay préparée Université Paris-Sud

Ecole doctorale n°580
Sciences et technologies de l'information et de la communication
Spécialité de doctorat : Informatique

par

M. Ian Masliah

Méthodes de génération automatique de code appliquées à l'algèbre linéaire numérique dans le calcul haute performance

Thèse présentée et soutenue à Gif-Sur-Yvette, le 26 septembre 2016.

Composition du Jury :

M.	Emmanuel Chailloux	Professeur Université Pierre et Marie Curie	(Président du jury)
M.	Paolo Bientinesi	Professeur Aachen University	(Rapporteur)
M.	David Hill	Professeur Université Blaise Pascal	(Rapporteur)
M.	Frédéric Magoulès	Professeur Ecole Centrale Paris	(Examinateur)
M.	Marc Baboulin	Professeur Université Paris-Sud	(Directeur de thèse)
M.	Joël Falcou	Maître de Conférences Université Paris-Sud	(Co-Directeur de thèse)

université PARIS-SACLAY
ÉCOLE DOCTORALE
Sciences et technologies de l'information et de la communication (STIC)

Titre : Méthodes de génération automatique de code appliquées à l'algèbre linéaire numérique dans le calcul haute performance

Keywords : programmation générique, DSELs, programmation générative, C++, algèbre linéaire, GPU

Résumé : Les architectures parallèles sont aujourd'hui présentes dans tous les systèmes informatiques, allant des smartphones aux supercalculateurs en passant par les ordinateurs de bureau. Programmer efficacement ces architectures en fonction des applications requiert un effort pluridisciplinaire portant sur les langages dédies (Domain Specific Languages - DSL), les techniques de génération de code et d'optimisation, et les algorithmes numériques propres aux applications.

Dans cette thèse, nous présentons une méthode de programmation haut niveau prenant en compte les caractéristiques des architectures hétérogènes et les propriétés existantes des matrices pour produire un solveur générique d'algèbre linéaire dense. Notre modèle de programmation supporte les transferts explicites et implicites entre un processeur (CPU) et un processeur graphique qui peut être généraliste (GPU) ou intégré (IGP). Dans la mesure où les GPU sont devenus un outil important pour le calcul haute performance, il est essentiel d'intégrer leur usage dans les plateformes de calcul. Une architecture récente telle que l'IGP requiert des connaissances supplémentaires pour pouvoir être programmée efficacement. Notre méthodologie a pour but de simplifier le développement sur ces architectures parallèles en utilisant des outils de programmation haut niveau. À titre d'exemple, nous avons développé un solveur de moindres carrés en précision mixte basé sur les équations semi-normales qui n'existait pas dans les bibliothèques actuelles.

Nous avons par la suite étendu nos travaux à un modèle de programmation multi-étape ("multi-stage") pour résoudre les problèmes d'interopérabilité entre les modèles de programmation CPU et GPU. Nous utilisons cette technique pour générer automatiquement du code pour accélérateur à partir d'un code effectuant des opérations point par point ou utilisant des squelettes algorithmiques. L'approche multi-étape nous assure que le typage du code généré est valide. Nous avons ensuite montré que notre méthode est applicable à d'autres architectures et algorithmes. Les routines développées ont été intégrées dans une bibliothèque de calcul appelée NT2.

Enfin, nous montrons comment la programmation haut niveau peut être appliquée à des calculs groupés et des contractions de tenseurs. Tout d'abord, nous expliquons comment concevoir un modèle de container en utilisant des techniques de programmation basées sur le C++ moderne (C++-14). Ensuite, nous avons implémenté un produit de matrices optimisé pour des matrices de petites tailles en utilisant des instructions SIMD.

Université Paris-Saclay
Espace Technologique / Immeuble Discovery
Route de l'Orme aux Merisiers RD 128 / 91190 Saint-Aubin, France

université PARIS-SACLAY

ÉCOLE DOCTORALE
Sciences et technologies de l'information et de la communication (STIC)

Title : Automatic code generation methods applied to numerical linear algebra in high performance computing

Keywords : Generic programming, DSELs, Generative programming, C++, linear algebra, GPU

Abstract : Parallelism in today's computer architectures is ubiquitous whether it be in supercomputers, workstations or on portable devices such as smartphones. Exploiting efficiently these systems for a specific application requires a multidisciplinary effort that concerns Domain Specific Languages (DSL), code generation and optimization techniques and application-specific numerical algorithms.

In this PhD thesis, we present a method of high level programming that takes into account the features of heterogeneous architectures and the properties of matrices to build a generic dense linear algebra solver. Our programming model supports both implicit or explicit data transfers to and from General-Purpose Graphics Processing Units (GPGPU) and Integrated Graphic Processors (IGPs). As GPUs have become an asset in high performance computing, incorporating their use in general solvers is an important issue. Recent architectures such as IGPs also require further knowledge to program them efficiently. Our method aims at simplifying the development on parallel architectures through the use of high level programming techniques. As an example, we developed a least-squares solver based on semi-normal equations in mixed precision that cannot be found in current libraries. This solver achieves similar performance as other mixed-precision algorithms.

We extend our approach to a new multistage programming model that alleviates the interoperability problems between the CPU and GPU programming models. Our multistage approach is used to automatically generate GPU code for CPU-based element-wise expressions and parallel skeletons while allowing for type-safe program generation. We illustrate that this work can be applied to recent architectures and algorithms. The resulting code has been incorporated into a C++ library called NT2.

Finally, we investigate how to apply high level programming techniques to batched computations and tensor contractions. We start by explaining how to design a simple data container using modern C++-14 programming techniques. Then, we study the issues around batched computations, memory locality and code vectorization to implement a highly optimized matrix-matrix product for small sizes using SIMD instructions. By combining a high level programming approach and advanced parallel programming techniques, we show that we can outperform state of the art numerical libraries.

Université Paris-Saclay
Espace Technologique / Immeuble Discovery
Route de l'Orme aux Merisiers RD 128 / 91190 Saint-Aubin, France

Table of contents

List of Figures

Introduction

In recent years, architectures and programming languages have gone through major changes and diversification. The constant need for more computational power and more generic software has been the driving force behind these changes. Solving linear systems of equations $Ax = b$ is part of many computational applications leading to a vast amount of research. The development of dense linear algebra has gone in hand with the evolution of computer languages and architectures. The High Performance Computing landscape has gone through many changes with the development of multicore processors and accelerators. One can safely predict that this trend will continue as current architectures are at a crossroad with the limits being reached for the lithography process. At the same time, computer languages have started to diversify with a focus on high level language concepts and functional programming paradigms. Recent languages such as Scala, Dotty, Rust or modern C++ are prime examples of this new direction. Such languages are also starting to become standard in High Performance Computing (HPC) to develop scalable and robust softwares. In this PhD thesis, we present a method of high level programming that takes into account the features of heterogeneous architectures and algorithmic properties to build numerical libraries.

The issues we consider throughout our work are related to software design and optimization for heterogeneous architectures. Software has historically being designed around solving problems for specific architectures, which limits portability and reuse. We propose an approach to develop generic interfaces that support architectural extensions. As the programming model can vary tremendously depending on the hardware, being able to give specifications when necessary and properly categorize hardware for code reuse is essential. This requires high level programming techniques that cannot be found in every language. The concept called Domain Specific Embedded Languages (DSEL) relies on a generic programming language that is flexible and expressive enough to enable the development of a sub-language with a definite syntax. This technique is known to be used in the languages mentioned above. In our work, we employ such programming techniques to design an architecture aware solver for dense linear systems. The resulting codes have been incorporated into a C++ library called NT2 [1]. We also consider the issues around batched computations and tensor contractions. This work is done outside of the NT2 library. A detailed explanation on how to develop a simple high level memory container without relying on current software is given. We then develop an optimized matrix-matrix product for small sizes using a tuning approach. In our benchmarks, computations on the GPU are run using the IEEE 754 compliant mode [138] for arithmetic operations and with CUDA ECC activated.

[1]NT2 is available at http://github.com/jfalcou/nt2

This manuscript is organized as follows:

In Chapter 1, we present the background for our work. First, we consider the evolution of some numerical libraries and describe how architectures and programming languages have influenced them. Secondly, we discuss the main algorithms to solve dense linear systems and how to use them in the most popular HPC libraries. Then, we give a general description of current architectures and the main programming tools and techniques available. This includes low-level architectural tools that require expertise to properly program the hardware and high-level programming techniques to make abstraction of such problems. Then, we present the programming language and library we use throughout most our work.

In Chapter 2, we present our method to design a high level programming framework to generate dense linear algebra software . We start by giving a software context as there are several approaches to such a problem. Then, we present the code generation techniques that we for our CPU/GPU programming model. This model will then enable the development of a dense linear solver that can support hybrid architectures. We illustrate our approach with a least squares solver based on semi-normal equations for CPU, GPU and Integrated Graphic Processors (IGP) that cannot be found in state of the art libraries.

In Chapter 3, we extend the work described in Chapter 2 by proposing a multi-stage programming approach. We give a detailed background on related work and libraries available in C++ to solve the code portability and support problems. Then, we explain the process behind our multi-stage approach in C++ and how we have implemented it into the NT2 library. We test our approach with experiments on the Black & Scholes algorithm for GPU and IGPs and by applying our multi-stage approach to the work described in Chapter 2.

In Chapter 4, we investigate how to apply high level programming techniques to batched computations and tensor contractions. We start by explaining how to design a simple data container using modern C++14 programming techniques. Then, we study the issues around batched computations, memory locality and code vectorization to implement a highly optimized matrix-matrix product for small sizes using SIMD instructions. By combining a high level programming approach and advanced parallel programming techniques, we show that we can outperform state of the art numerical libraries.

The main contributions of this PhD thesis can be summarized as follows:

– 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.

- We present a multi-stage approach to alleviate the host/device programming model with a cost model for offloading code. This provides adaptable strategy to generate CUDA kernel directly from a single C++ source file containing NT^2 statements.

- Our method takes advantage of modern C++ design for hybrid computations and interface development without relying on current software.

- We describe a strategy using performance analysis based on hardware features and counters to implement efficient algorithms on specific architecture. This work is based on the implementation of a small batched matrix-matrix product that can perform better than state of the art libraries.

Chapter 1

The issue of adapting numerical libraries to parallel architectures

Contents

Parallelism in todays computer architectures is ubiquitous whether it be in supercomputers or small portable devices such as smartphones or watches. While computers hardware is evolving, computing power as expressed by Moore's law is still the incentive for all manufacturers. If frequency scaling has been a temporary solution, power dissipation and the increasing complexity in CPU design has lead manufacturers to develop more sophisticated architectures. These new architectures that allow to increase the computational power vary from CPU multicores to GPUs, or even integrated system-on-chip (SOC). Exploiting the full performance of such systems for numerical problems has become very challenging.

In order to take fully advantage of these new architectures, software libraries are developed to give user access to efficient linear algebra computations. The first major numerical libraries to appear include the Scientific Subroutine Package by IBM [72] in 1968, the Basic Linear Algebra Subprograms (`BLAS` [41]) or the Linear Algebra Package (`LINPACK` [42]) in 1979. Each of these packages were already implemented using some form of modular programming for better flexibility. Since then, architecture specific libraries for dense linear algebra have been developed such as LAPACK [6] for cache-based CPUs, ScaLAPACK [21] for distributed architectures, PLASMA [126] for multicores introducing tiled algorithms, data layouts and more recently MAGMA [12, 127] for accelerator-based architectures with GPUs or Xeon Phi.

These evolutions have gone in hand with programming techniques and languages. As we go back, LINPACK was written in Fortran 66 and LAPACK in Fortran 77. The more recent libraries were developed in C for PLASMA and C++ for MAGMA. In each new library the level of abstraction has been raised by using generic programming techniques. However, due to retro-compatibility concerns and the need to maintain a similar interface for the routines, the use of high level programming techniques is still very limited. Moreover, with the increasing parallelism and heterogeneity as well as the ever increasing data-communication cost, the algorithms were completely modified and redesigned to take advantage of each new architecture. The disparity between these libraries that target different architectures illustrates one of the major issues in designing optimized linear algebra software.

Parallel architectures are essential for scientific applications due to the high computing power they provide. Using efficiently these architectures is a difficult task due to the diversity of the hardware and the large amount of software solutions. Algorithms for CPU are still evolving to better adapt to architectures. For examples, the ReLAPACK [105] library has been developed using recursive programming techniques and can outperform LAPACK. It is in this context that we place our work. We focus on architecture aware software design for scientific computations.

In this chapter, we give an overview of the different architectures and programming techniques that are the premises for our work. After explaining why high level programming matters in Section 1.1, we present the algorithms and solvers for dense numerical libraries in Section 1.2. We then detail in Section 1.3 the different architectures and the impact they can have on the different algorithms. Following this, we present the programming tools and techniques in Section 1.1 that are used to develop high level libraries. Finally, we discuss in Section 1.5 about the language and library we used for our work.

1.1 High level Programming for Scientific Computing

The evolution of semiconductor technology is dramatically transforming the balance of future computer systems, producing changes at every level. From the point of view of numerical libraries, and the myriad of applications that depend on them, having more computational power allows to work on bigger simulations. These simulations can take up to months of continual computation depending on size of the problem. For this reason, having robust frameworks has become essential.

Developing fast applications in a reasonable amount of time requires a multi-disciplinary effort incorporating Domain Specific Languages, code generation and optimization techniques, domain science and application-specific numerical algorithms. As computational power and architectures are growing each year, it has now become a major concern to properly design libraries. In 2008 the first Petaflop machine entered the Top 500 while a research article was already explaining the challenges for ExaScale computing [18] and the emerging architectures. Modularity is now one of the main concern when writing software frameworks. Modern software such as GEANT4, a simulation toolkit for simulating the passage of particles through matter [4], or ROOT [23], for High Energy and Nuclear Physics, are developed using high level languages like C++ and can integrate other languages such as Python or R.

Exploring the use of accelerators as a general purpose co-processor to increase the computational power has been an active research field since the release of the CUDA language in 2007. Dedicated libraries for such architectures have emerged such as MAGMA, CUBLAS by NVIDIA$^{\text{TM}}$ or ACML by AMD$^{\text{TM}}$ but these are architecture specific. The MAGMA library exists in different versions depending on the hardware vendor. For major frameworks such as GEANT4, we can find research studies explaining how to implement algorithms for new architectures such as Xeon Phi [115]. These changes require advanced methodologies to compile and execute any scientific program. The article also shows that applications need to be heavily tuned to increase performance. The scientific community has a wide range of domains that require dedicated scientific applications to solve the programming challenges. Having an adaptable library does not need to be reimplemented for each new architecture is one of the main challenge of Scientific Computing.

1.2 Solving dense linear systems

Many scientific applications whether dense or sparse require at some point to solve a linear system of equations $Ax = b$. Examples range from fields such as Fluid Dynamics or Electromagnetism using Boundary Integral equations to tensor-based simulations [80]. Due to this, linear algebra algorithms must be robust, accurate and scalable to fulfill the needs of High Performance Computing.

This notion of numerical stability [69] plays a crucial role in designing algorithms as the result obtained by a numerical algorithm does not correspond to the solution of the problem in exact arithmetic. The different sources of errors may come from the input data of the algorithm, which may be caused by prior computations or measurement errors, which are themselves limited to the machine precision currently under the IEEE 754 norm [77]. Also, errors may be caused by approximations made by the algorithm. Depending on the desired accuracy, several algorithms can be chosen to solve a linear system.

We can find two major approaches to solve linear systems : direct or iterative methods. A direct method uses a finite sequence of operations to provide an exact solution x if there are no rounding errors. The common method used to solve such systems is called Gaussian elimination. It consists in adding the coefficients of one equation to the others in order to eliminate a variable and continue this process until only one variable is left. Iterative methods start with an approximation of the solution x^0 and successively compute a sequence of approximations x^i to improve the solution. In our work we focus on direct methods which are generally used to solve dense linear systems. Direct methods provide higher numerical precision and a good granularity of computations making it easier to exploit the latent parallelism. We note the main decompositions that exist [57] : LU, Cholesky, QR, SVD, LDL$^{\mathrm{T}}$.

LU: Decomposes a general matrix A as $L \times U$ where L is unit lower triangular and an U is upper triangular (about $2 \times n^3/3$ flops).

Cholesky: A symmetric positive definite (SPD) matrix A is factored as $A = LL^T$ where L is lower triangular (about $n^3/3$ flops).

QR: An m-by-n matrix A is factored as $A = QR$ with Q an m-by-m orthogonal matrix and R an m-by-n upper triangular matrix ($2n^2(m - n/3)$ flops).

SVD: Decomposition of an m-by-n matrix A as $A = U\Sigma V^T$, U an m-by-m orthogonal matrix, Σ an m-by-n diagonal matrix (where the diagonal contains the singular values of A) and V an n-by-n orthogonal matrix V (the cost is about mn^2 flops to compute Σ).

LDL^T: factorization (using symmetric pivoting) is $PAP^T = LDL^T$ where P is a permutation matrix, A is a symmetric indefinite square matrix, L is unit lower triangular and D is block-diagonal, with blocks of size 1×1 or 2×2 (about $n^3/3$ flops).

1.2.1 LU factorization

The LU factorization [57, p.111] is a modified Gaussian elimination without pivoting that decomposes a nonsingular matrix A into a lower triangular matrix, and an upper triangular matrix U. The number of floating point operations of the LU decomposition for a square matrix n is about $2n^3/3$. For dense matrices, the LU factorization is usually performed in place. This means that the output factors L and U overwrite the matrix A during the factorization.

Algorithm 1 LU factorization in place, no pivoting

```
Input: $A$ is $n \times n$ matrix
 1: for $k = 1 : n-1$ do
 2:   for $i = k+1 : n$ do
 3:     $A(i,k) = A(i,k)/A(k,k)$
 4:   end for
 5:   for $i = k+1 : n$ do
 6:     for $j = k+1 : n$ do
 7:       $A(i,j) = A(i,j) - A(i,k) * A(k,j)$
 8:     end for
 9:   end for
10: end for
```

The LU factorization is generally implemented using pivoting strategies because there are several important issues that can appear without pivoting. For example, if a zero is found on the diagonal of the matrix, a division by zero will occur leading to a numerical error. Also, if there are elements of small magnitude on the diagonal, the factors will end up growing too much also leading to erroneous results. We note $PA = LU$, with P a permutation matrix, when using partial pivoting in LU factorization. The stability of Gaussian elimination can be measured with the growth factor [69]. The growth factor measures the ratio between the entries of the matrix after the elimination steps and the original entries. The growth factor of a square matrix A under Gaussian elimination is defined as:

$$g_n(A) = \frac{\max_{i,j,k} |a_{ij}^{(k)}|}{\max_{i,j} |a_{i,j}|},$$

where $a_{ij}^{(k)}$ is the element of index (i,j) after the step number k of the elimination [78]. The most common pivoting strategy used is called partial pivoting. It consists in permuting the rows but not the columns to ensure that the pivot is the largest entry in its column. For a step j of the Gaussian elimination, partial pivoting permutes the rows such that $|a_{j,j}| \geqslant |a_{i,j}|$ for all $i > j$. This guarantees that $||L||_\infty \leqslant 1$. Its growth factor upper bound is 2^{n-1} which can be reached for certain problems [53].

Algorithm 2 LU factorization in place, partial pivoting

```
Input: $A$ is a $n \times n$ matrix
1:  for $k = 1 : n$ do
2:    $index \leftarrow k$
3:    for $i = k + 1 : n$ do
4:      if $|A(i, k)| > |A(index, k)|$ then
5:        $index = i$
6:      end if
7:    end for
8:    swap rows $k$ and $index$
9:    for $i = k + 1 : n$ do
10:     $A(i, k) \leftarrow A(i, k)/A(k, k)$
11:   end for
12:   for $i = k + 1 : n$ do
13:     for $j = k + 1 : n$ do
14:       $A(i, j) = A(i, j) - A(i, k) * A(k, j)$
15:     end for
16:   end for
17: end for
```

The drawback of pivoting comes from the communication cost due to the comparisons performed to find the pivot and to the resulting swap of rows. For a square matrix, the partial pivoting requires $\mathcal{O}(n^2)$ comparisons. Other well know pivoting strategies are total pivoting ($\mathcal{O}(n^3)$ comparisons) and rook pivoting ($\mathcal{O}(1.5n^{3/4 \log n})$ comparisons).

The LU factorization is a classical technique that is used with different implementations in high performance linear algebra libraries. We can find applications with different pivoting strategies, precision, with iterative refinement or even in mixed precision. For this reason, we have used the LU factorization and its various forms as an important part of our generic solver for dense linear systems that we are going to detail in Chapter 2.

1.2.2 QR factorization

The QR factorization [57, p.246] decomposes a matrix A into the product of an upper triangular matrix R and an orthogonal matrix Q. This is the main method used to solve the least square problem which consists in finding a minimal solution to a system $Ax = b$. This means solving $||Ax - b||_p$, A being an m by n matrix, for a defined p. There are multiple methods to compute the QR factorization of a matrix. Here we will focus on factorization with the Householder transforms which can be found in numerical libraries. We will described in Chapter 2 how we implemented a versions of mixed-precision semi-normal equations based on this QR factorization.

During the QR factorization, applying the Householder transformations on the matrix A will remove every entry below the diagonal. By repeating this process n

times, we obtain the upper triangular matrix R such that :

$$H_n H_{n-1} ... H_1 A = R.$$

We note the form of an orthogonal matrix computed durign the factorization as :

$$H = I - \frac{2}{||u||^2} uu^T.$$

If we set $Q = H_1 ... H_n$, we can then obtain A = QR which means we have $Q^T A = R$. Algorithm 3 illustrates the QR factorization as described by Golub using the house function [57, p.237].

Algorithm 3 HouseHolder QR in place, no pivoting

```
Input: $A$ is a $m \times n$ matrix
1: for $k = 1 : n$ do
2:     $[v, \beta] = house(A(j : m, j))$
3:     $v_j = [0...0, 0, 1, v^j_{j+1} .... v^j_m]^T$
4:     $\beta_j = \frac{2}{1+||A(j+1:m,j||^2)}$
5:     $A(j : m, j : n) = (I - \beta vv^T)A(j : m, j : n)$
6:     if $j < m$ then
7:         $A(j + 1 : m, j) = v(2 : m - j + 1)$
8:     end if
9: end for
```

We note that this method is known to be numerically stable [139].

1.2.3 Solvers in LAPACK and MAGMA

Solvers for linear systems can be found in LAPACK and MAGMA constitute the building blocks to write a generic library for dense linear algebra. The routines given in these libraries can be distinguished by the static properties of the matrix. These properties correspond to the structure of a dense matrix which are for instance general, band, symmetric, positive definite or tridiagonal. Other structures can be considered but they may require new implementations to be written. The more common class of linear systems correspond to general dense matrices. This study focuses on dense matrices which already cover a wide range of applications.

What we describe below is the guideline of the main algorithms that are implemented. This however does not represent the implementation since it will depend of the structure of the matrix and also the target architecture. LAPACK being for CPU, the implementation is more straightforward and matches the algorithms while MAGMA will tend to be more specific. Indeed, since MAGMA is designed for GPU, it is important that the transitions between CPU and GPU is smooth to ensure minimal communication cost.

1.2.3.1 General solvers

The solver used in LAPACK/MAGMA for general linear systems (`xgesv`, the letter `x` indicating the precision, either real (`s`), double-real (`d`), complex (`c`) or double-complex (`z`) ; `ge` standing for general) is based on the LU decomposition with partial row pivoting. This technique, called `Gaussian elimination with partial pivoting` is implemented in most numerical libraries including LAPACK and MAGMA and is used in the LINPACK benchmark for the TOP500 list.

To solve the linear system $Ax = b$, we compute a factorization $PA = LU$ with P permutation matrix, L lower triangular and U upper triangular. Then the solution x is computed by solving successively the triangular systems $Ly = Pb$ followed by $Ux = y$.

Algorithm 4 Solving $Ax = b$ with LU factorization

```
1: Compute the factorization $PA = LU$ with $P$ permutation matrix, $L$ lower
   triangular and $U$ upper triangular.
2: Solve $Ly = Pb$.
3: Solve $Ux = y$.
4: Solution is $x$.
```

In the case of rectangular matrices, the routine `xgelsy` uses the QR decomposition of the matrix (possibly with column pivoting) to solve the linear least square problem [20] :

$$\min_{x\in\mathbb{R}^n} \|Ax - b\|_2$$

Algorithm 5 Solving $Ax = b$ with QR factorization

```
1: Compute the factorization $A = QR$ with $Q$ orthogonal matrix and $R$ upper
   triangular matrix.
2: Form $d = Q^T b$.
3: Solve $Rx = d$ by backward substitution.
4: Solution is $x$.
```

The interest of these factorizations is that we can reuse the obtained factors for other computations which can also be very expensive such as computing the condition number [69], or performing iterative refinement [38]. In the case of a square matrix, we will identify three types of solvers that are available in LAPACK and will mention others that can added. In the case of a rectangular matrix we can identify a unique solver, which is the routine `xgelsy`.

The first one is the routine `xgesv` to compute a solution using the LU factorization. This can be seen as the standard case when there is no particular information about the system. Other cases include iterative refinement and mixed precision.

1.2.3.2 Iterative refinement

The second one is the routine `xgesvx` using iterative refinement which enables it to improve the accuracy of the computed solution. Iterative refinement is a method that produces a correction to the computed solution by iterating on it. Each m^{th} iteration in this process consists of computing the residual $r_m = b - Ax_m$, solving the new system $Ad_m = r_m$, and adding the correction $x_{m+1} = x_m + d_m$.

Algorithm 6 One iteration of iterative refinement

```
1: Compute the residual $r = b - Ax$.
2: Solve the new system $Ad = r$.
3: Add the correction $x = x + d$.
```

This method is computationally more expensive but can provide better solution for ill-conditioned systems.

1.2.3.3 Mixed precision algorithm

The third method is the solver `dsgesv` that uses mixed precision iterative refinement [10]. This method takes advantage of the fact that computing in single precision is in general much faster than computing in double precision. The factorization (most expensive part of the computation) is computed in single precision (ε_s) (if the matrix is not ill-conditioned) and the solution is refined in double precision (ε_d) using the classical iterative refinement method.

Algorithm 7 Mixed precision, iterative refinement

```
1: Compute $PA = LU$              $(\varepsilon_s)$
2: Solve $Ly = Pb$                $(\varepsilon_s)$
3: Solve $Ux_0 = y$               $(\varepsilon_s)$
   do k = 1,2,...
4: $r_k = b - Ax_{k-1}$           $(\varepsilon_d)$
5: Solve $Ly = Pr_k$              $(\varepsilon_s)$
6: Solve $Uz_k = y$               $(\varepsilon_s)$
7: $x_k = x_{k-1} + z_k$          $(\varepsilon_d)$
   check convergence
```

These methods can be found in current open source libraries. There are also other interesting methods that can provide satisfactory performance. An example is a method that accelerates linear system solutions using randomization techniques [14], in which the system is modified by a multiplicative preconditioning based on Random Butterfly Transformations [104] that removes the need for pivoting (which is an expensive process). Another method is based on communication-avoiding algorithms that minimize the cost of pivoting [11, 63].

1.3 Architectures : Design and evolution

1.3.1 SIMD extensions

Single instruction multiple data (SIMD) architectures correspond to processing units that can perform an operation on multiple data simultaneously (see Figure 1.1). SIMD extensions are designed to help processors exploit the latent data parallelism available in many applications at a smaller cost than duplicating Arithmetic Logic Units (ALU).

Multiple constructors have implemented vector extensions. SIMD extensions started mostly with Sun Microsystems Visual Instruction Set (VIS) for the SPARC V9 [79] and Hewlett-Packard Multimedias Acceleration eXtensions (MAX) for the PARISC instruction set [84]. This lead other constructors to develop their own SIMD extension set.

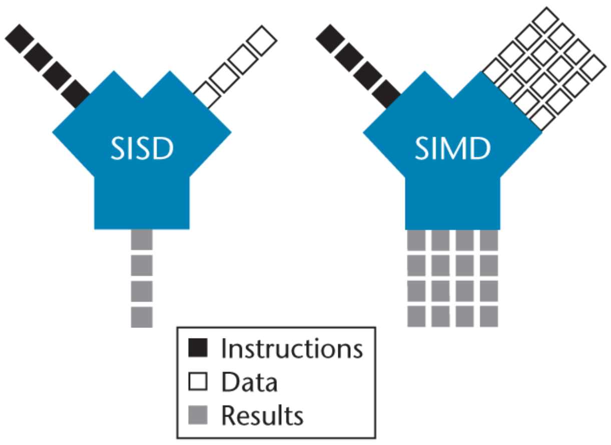


Figure 1.1: SIMD concept

Motorola™ worked with Apple™ and IBM™ on the PowerPC architecture to develop the Altivec extensions [40] set. ARM™ introduced its first SIMD instruction set on ARMv6 which operated on 32-bit general purpose registers. Later on, ARM developed the Neon extension set for ARMV7-A with 64 bytes registers. Then, ARM extended Neon to 128 bits and added double precision computation. Intel™ introduced its SIMD extensions called streaming multimedia extensions (SSE) with the P5-based Pentium x86 processors. This first extension called MMX [106] used floating point registers instead of having specific vector registers which disabled scalar computations. Later on, they developed their first processors with dedicated vector registers (Pentium III) with 64 bytes registers. Since then, manufacturers have continued to increase the size of dedicated SIMD registers which now go up to 512 bytes vectors for Intel Xeon Phi coprocessors [26] (which corresponds to a line of L1 cache).

Exploiting the parallelism offered by these extensions is critical for high performance computing as it can theoretically increased performance up to 8 times with 512 bytes registers for double precision operations. Using SIMD extensions can also increase the bandwidth. Linear algebra can benefit greatly from SIMD extensions as BLAS operations which are fundamental for most algorithm are vector based.

1.3.2 Multicore systems

The processor manufacturers faced a technology limitation to their traditional approach of boosting clock speeds and increasing throughput due to power dissipation problems. This lead to the development of multicore architectures and hyper-threading as a solution which is a turning point in terms of software development. While constructors could increase the frequency in processors freely, the performance of a program would naturally increase which each new generation of processor. This went on until the end of the Intel Pentium 4 processors in 2006 with frequencies around 3.6 GHz. By then multicores were already available on the market but were not yet generalized. The common concept behind a multicore processor is to let each core have an L1 independent cache and put a shared L2 cache on the die which is the interface to the main memory. Figure 1.2 show a multicore systems with two levels of cache. Modern CPUs generally have two levels of cache in each core and a shared L3 cache. Having multiple CPU cores on the same die allows for a highly efficient cache coherent system at a much higher clock rate than if the signal had to travel off chip.

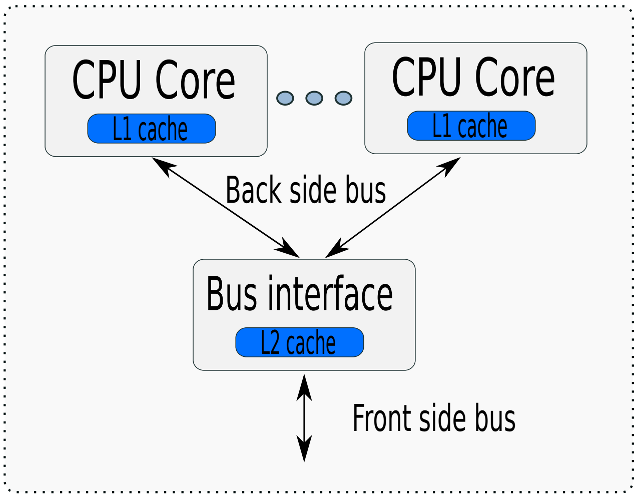


Figure 1.2: Multicore concept

Most devices ranging from desktop computers and supercomputers to smartphones are based on a multicore design (see Figure 1.2). The number of cores can vary greatly depending on the solution going up to 22 cores on recent server based Intel Xeon processors family. Other parallelization techniques based on simultaneous multithreading (SMT) to improve the efficiency of superscalar CPUs can be found. As an example, Hyper threading (HT) consists in sharing resources by having two logical core for one single physical core. Another solution is the Non Uniform Memory Access (NUMA) architecture. Such systems are generally built around having multiple processors with each processor having its own memory bank (see Figure 1.3). Each processor is able to access the memory bank of the others but latencies will vary depending on the locality of each processor. To support this system, the operating system provides a virtual memory address space for all the processors in the NUMA node.

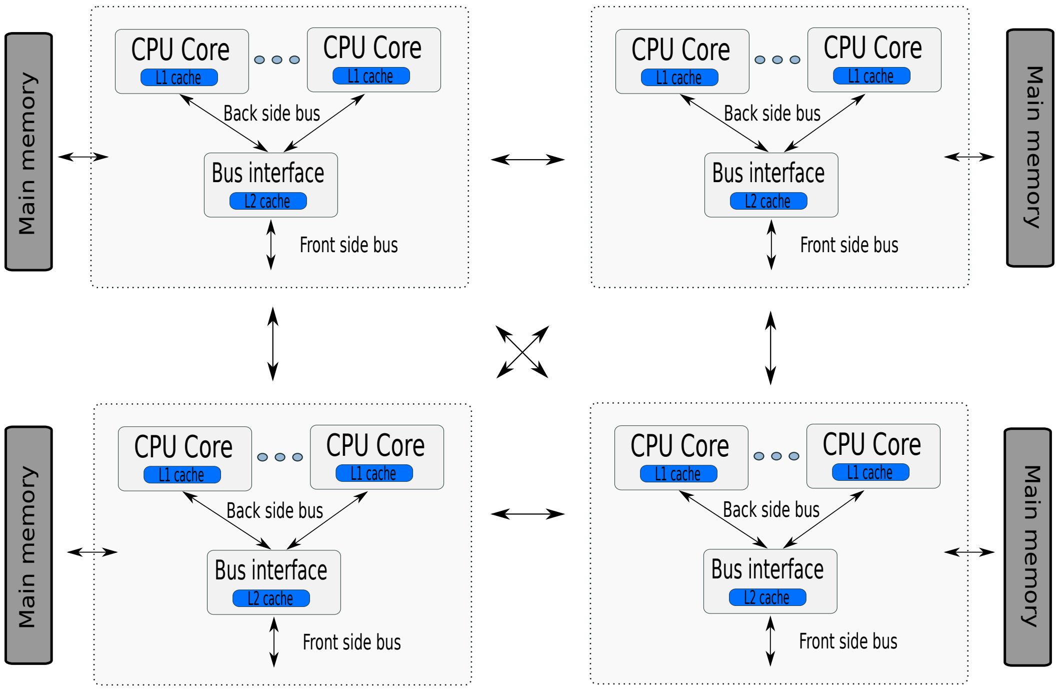


Figure 1.3: NUMA concept

1.3.3 Graphics Processing Units (GPU)

Applications are becoming more demanding in workload and complexity as architectures become more powerful. CPUs have shown their limits in terms of computational power as they tend to be more general. This technology gap has led to the development of new architectures that are more specialized and can therefore provide more computational power for specific applications. GPUs are an example of a new type of architecture that has been emerging during the last decade.

A GPU is a specialized hardware to accelerate image display and processing. The first programming language for such architectures was released in 2007 by NVIDIA. The Compute Unified Device Architecture (CUDA) is a platform designed to work with scientific programming languages such as C, C++ and Fortran. CUDA has a

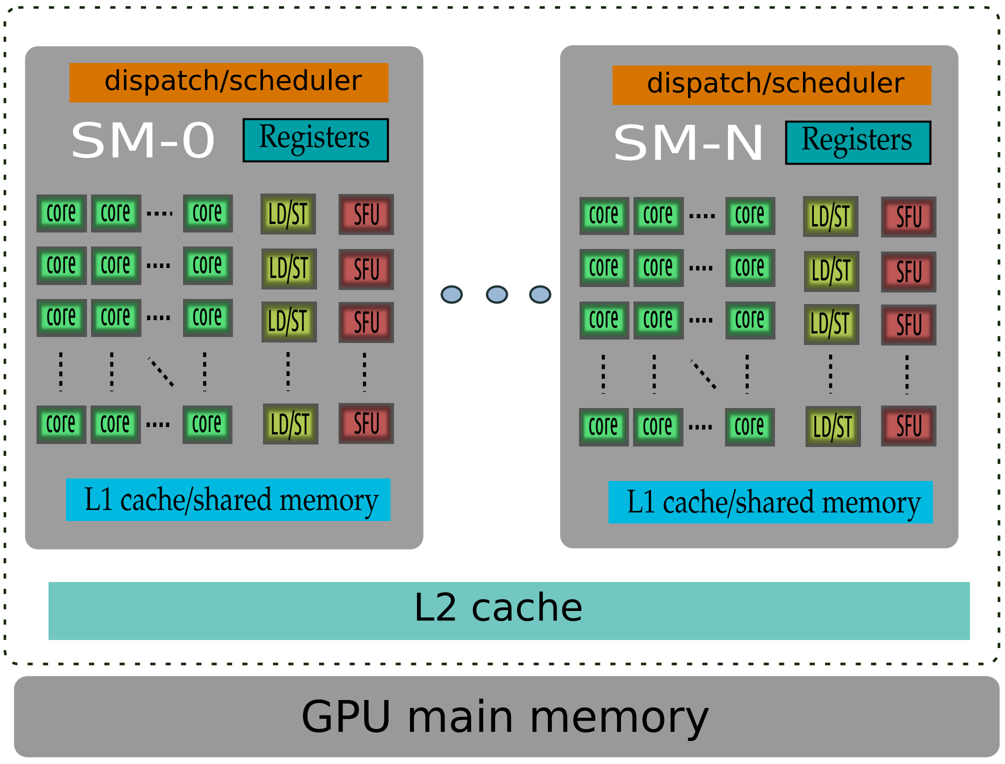


Figure 1.4: GPU Kepler based architecture

dedicated compiler called NVIDIA CUDA Compiler (NVCC) that supports C and a subset of C++. As of the November 2015 TOP 500 ranking, 75 machines are leveraging NVIDIA GPUs while only 55 were doing so in the June ranking. GPUs are now on the front scene of HPC as they are becoming increasingly important to improve the peak performance of systems. The other main vendor of GPU is AMD with its Accelerated Processing Platform (AMD APP).

GPUs architectures are very different from multicore CPUs and provide more computational power and better energy efficiency. The number of processing units available in a GPU is much higher than in a CPU (see Figure 1.4). A Tesla K40 has 2880 processor cores available but each runs at a lower frequency (745 GHz). Each of the cores can be seen as simple ALU that can only be used for computations. Cores are grouped together in what is called streaming multiprocessor (SM) with 192 CUDA cores for Kepler cards. Each SM has registers, an shared L1 cache and has a shared L2 cache between each SM with a maximum of 15 SM for Kepler cards. Similarly to CPUs, the number of cache levels on the GPU has grown to mitigate the high latency from the GPUs main memory.

Contrary to the CPU where data goes through each cache level for Intel processors, the data on the GPU does not necessarily stay at each cache level. Some data management therefore needs to be done to optimize data movement. The GPU

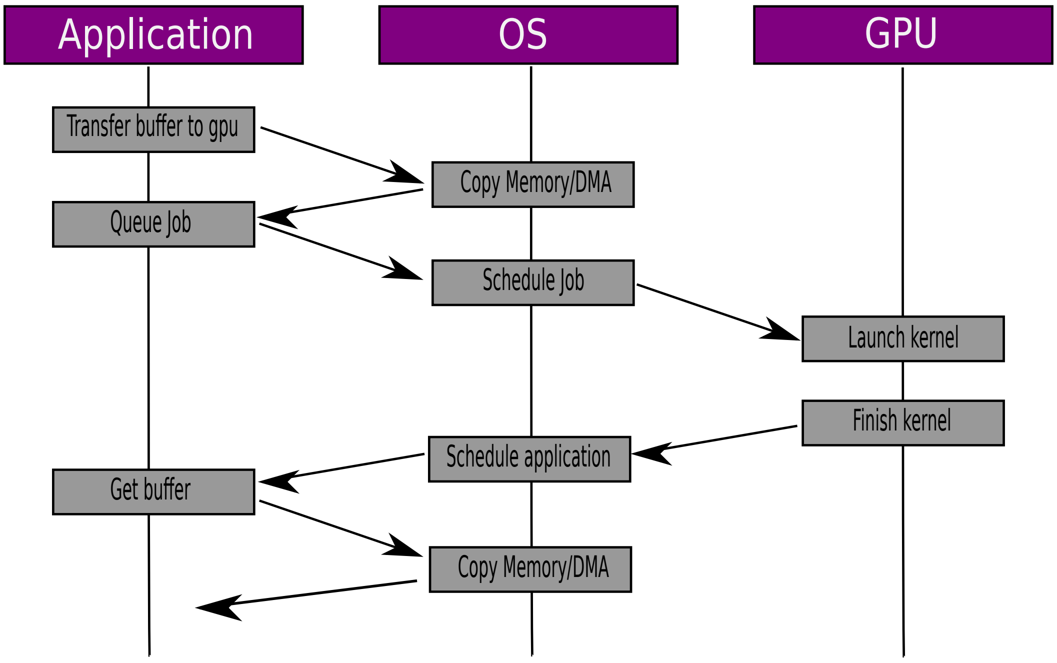


Figure 1.5: Work flow for a GPU

has a thread scheduler and control units in each SM(see Figure 1.4). The execution model of the GPU is based on deploying a large number of threads (in the order of millions) as they have no creation cost compared to a CPU. The scheduler will then distribute the threads minimizing the number of stall cycles. This creates a high parallel programming model where a high number of compute units will work in parallel. Communications with the CPU use the PCI express bus. The speed of the PCI bus can vary depending on the technology, with a maximum of 6 GB/s for PCI express gen 2 (16x) and 12 GB/s for PCI express gen 3. Depending on the application, the PCI express bus can be a bottleneck for memory-bound problems due to data transfers (see Figure 1.6). With each new architecture generation, GPU architectures often go through many changes but manage to maintain the same programming model.

1.3.4 Accelerated Processing Units

GPUs and accelerator based hardware are a major component of todays systems. However, using accelerators to their full potential can be impossible due to the limitations of the PCI express bus. Furthermore, GPUs are not stand alone processors and requires a CPU to function. This has lead to the development of System On Chip (SOC) hardware where a CPU and a GPU are combined on a single die. NVIDIA and AMD were the first manufacturers to develop this type of hardware with the NVIDIA Tegra brand in 2008 and AMDs Accelerated Processing Unit (APU) in 2011. Later on, Intel released its Integrated Graphic Processors for the

Intel core processors (Figure 1.8). As there is no official term for such hardware, we will refer to them as Integrated Graphic Processors (IGP). IGPs do not require a PCI Express bus as the CPU and GPU are on the same die (see Figure 1.6).

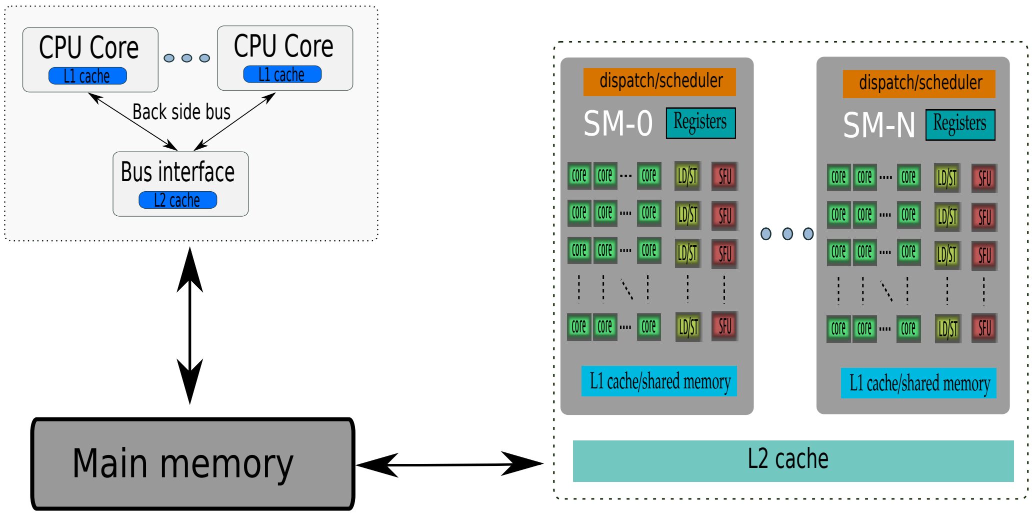


Figure 1.6: IGP architecture

As both hardware have a direct access to the main memory, the GPU has a higher memory bandwidth. The first implementations of IGPs by AMD did not have a unified memory system between the GPU and CPU. This meant that they could not access the same address space limiting the interactions between each processor. With it's second generation of IGP called Kaveri which was released in 2014, AMD implemented a zero-copy model where CPU and GPU can access the memory with the same address space. This means pointers can be freely passed between CPU and GPU, thus preventing actual deep copies. NVIDIA also released an IGP with zero-copy through the Tegra K1 and later on the X1. AMD IGPs are coupled with AMD CPUs while NVIDIA uses ARMs CPUs. It is important to note that current IGP have very poor support for double precision computations.

1.4 Programming tools and techniques

Section 1.3 described the current architectures without giving too much details about their programming models or the tools available for such platforms. In the next section, we are going to describe some of these tools and techniques often used to develop code for parallel architectures. Each tool will be described by its support for different architecture and expressiveness while techniques will explain how to build a tool for more expressiveness and architectural support.

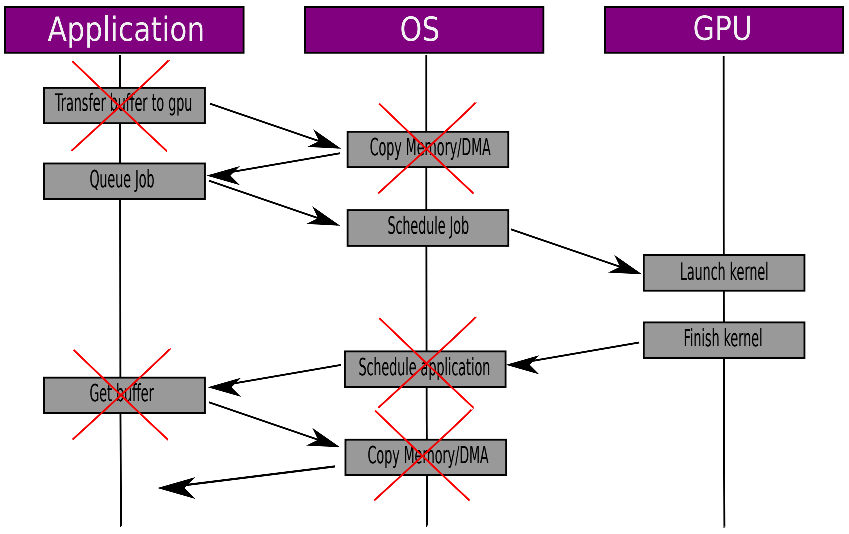


Figure 1.7: Work flow for a zero copy IGP

1.4.1 Low-level programming tools

1.4.1.1 SIMD support

In section 1.3.1, we introduced the concept of SIMD extensions. Exploiting SIMD units in todays code is essential to reach the peak performance of a modern CPU. The most common method to exploit these extensions is by using the function intrinsics available. These functions are available in C but have been ported to other languages even web oriented ones such as JavaScript. The low level intrinsics in C are still used in every day programming and tend to be quite verbose. Depending on the size of the SIMD registers or the architecture, the intrinsics are also not the same. This causes some API problems and render complex the generalization of SIMD extensions. In Listing 1.1 we can see an example of the intrinsic for the multiplication of two 256 bits vectors.

```
__m256 a, b, c ;
c = _mm_mul_ps(b,c);
```

Listing 1.1: Intel AVX2 single precision multiplication

The verbosity of SIMD instructions has been diminished in recent years and computational intrinsics such as addition, multiplication, etc.. can be replaced by its operator as seen in Listing 1.2.

```
__m256 a, b, c ;
c = b * c;
```

Listing 1.2: Intel AVX2 single precision multiplication no intrinsic

On ARM NEON, the SIMD maximum size is 128 bit. It is therefore not possible to do the same operation as above but we can do a similar one with 128 registers. It is also possible to simply call the operator instead of the full intrinsic (Listing 1.3).

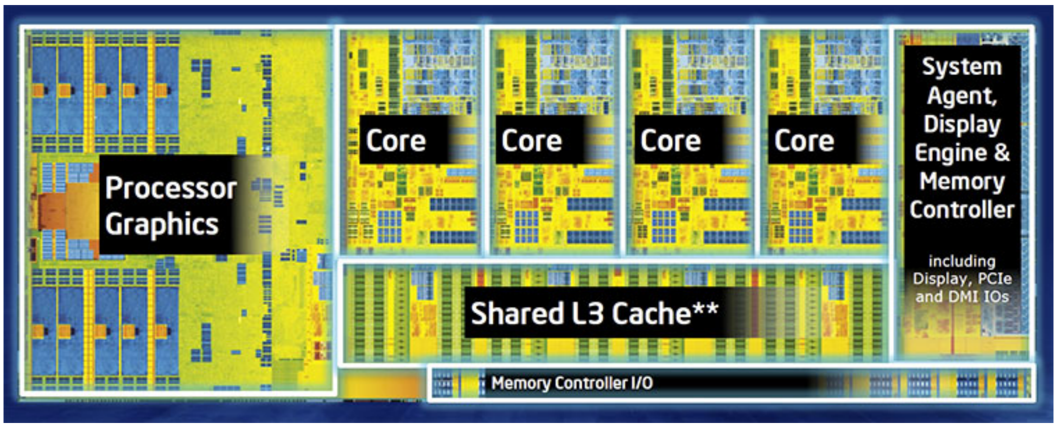

Processor Graphics
Core
Core
Core
Core
System Agent, Display Engine & Memory Controller
including Display, PCIe and DMI IOs
Shared L3 Cache**
Memory Controller I/O

Figure 1.8: Intel's 3rd generation IGP architecture

```
float32x4_t a, b, c ;
c = b * c;
```

Listing 1.3: ARM NEON single precision multiplication

In these examples we do not show the load and store operations needed to fill SIMD registers. For Intel processors, data also needed to be aligned on older processors to properly use SIMD extensions. On modern processors, unaligned load and store are quite sufficient to reach good performance most of the time as long as data doesn't cross cache lines. With the introduction of ARM supercomputer based system, recent studies [111] have shown how to improve energy efficient and design of HPC systems on low-power cores. Studying how these ARM based system can perform is therefore of interest.

Compilers for low level languages such as C, C++ can generate SIMD code through their auto-vectorization techniques. The programmer can write a simple code that will be translated to SIMD code during the compilation phase. GCC [100] is an example of a compiler using a vectorizer. It is also possible to use Just In Time (JIT) compilation to vectorize code with tools such as VaporSIMD [99] or use library based solutions such as Boost.SIMD [49] or VC [83]. The common problem with generating SIMD comes from how patterns are recognized. It is not always obvious how to vectorize a problem especially if data is not contiguous. Compiler can provide directives such as *#pragma simd* for ICC or *#pragma GCC ivdep* for GCC. These directives usually tells the compiler that their is no loop-carried dependencies which prevents consecutive execution of a loop with SIMD. Complex operations such as shuffle or intra-registers are hard for compiler to generate as they tend to be very specific. Some compilers also use SIMD extensions in their implementation of the standard library such as GCC.

1.4.1.2 Multicore support

As we have seen in section 1.3.2 modern CPUs contain multiple cores on the same die. Most programming languages do no support multicore programming by default meaning the programmer must do the effort. We can still cite a language such as Chapel by CRAY which is available on Github. Chapel [25] is designed around multi-threaded execution via high-level abstractions for data parallelism, task parallelism, concurrency and nested parallelism (see example in Listing 1.4).

```
use CyclicDist;            // use the Cyclic distribution library
config const n = 100000;   // override default using ./a.out --n=<val>

forall i in {1..n} dmapped Cyclic(startIdx=1) do
 writeln("Hello from iteration ", i," of ", n, " running on node ", here.id);
```

Listing 1.4: Chapel Hello World

The main tools for parallel programming on multicore systems are Pthreads, OpenMP, TBB and MPI.

Shared memory multi-threading (pthreads) is an implementation of the POSIX 1003.1c Standard. It is based on a model where threads are spawned on the fly while each thread is given a task at its creation. The threads can then communicate through shared memory variables. The Pthread standard provides functions for thread synchronization - barriers, semaphores, and critical regions in its API. Pthread does not have a specific programming model and allows for task or data parallelism but requires a complex semantic and is error prone. It is a low-level explicit model where the programmer has to allocate threads, synchronize them, and then recompose them. Higher level implementation of Pthread can be found in programming language such as the C++11 $std :: thread$.

OpenMP is a high level parallel programming model. It uses pragmas to add parallelism on a sequential implementation of an application. This requires the code section to be run in parallel to be dependency free between iterations. OpenMP also allows for task based parallelism to express dependencies. It uses a runtime library for thread management and scheduling techniques. The standard is constantly evolving and since OpenMP 4.0 supports extensions for accelerators based architectures.

Intel Thread Building Blocks (TBB) [108] is a C++ template library for data parallelism. It is based on the concept of tasks instead of threads which are higher level entities. TBB uses parallel skeletons constructs as building blocks. We note for example $parallel_for$, $parallel_reduce$, $parallel_scan$ and $parallel_sort$. The task based abstraction is an efficient high level model and provide simple abstractions with the help of lambda expressions since C++11.

Message passing (MPI) is a popular programming model originally designed for distributed computing. It provides a low-level explicit model where each core of a processor can be addressed independently having its own memory pool. MPI is more of a task-based parallelism model but also supports data parallelism by deploying similar MPI processes on multiple data. Recent MPI standard have added special communicators for shared memory programming.

1.4.1.3 Accelerator support

In section 1.3.3 we have explained how the GPU programming model is based on offloading data and using data parallelism. The standard programming tools used to program accelerators are CUDA and OpenCL.

CUDA is the programming language for NVIDIA graphic cards. It works for GPU as well as NVIDIA IGPs without having to change the device code. As explained before, the NVCC compiler supports a subset of the C++ language. To execute code on a GPU, the programming model includes a host code for the CPU and a device code which will be executed on the GPU. CUDA is an expressive language and simplifies the use of accelerators following the C++ standard evolutions. Unfortunately, it is not cross platform which can be a heavy restriction for portable code.

OpenCL is the portable solution for programming accelerators. It can work with CPUS and GPUs indistinctly but a single code cannot be highly efficient on both a CPU and a GPU. Similarly to CUDA, an OpenCL program has two parts. A kernel which will be executed on one or more OpenCL devices (similar syntax to CUDA) , and a host program which is executed on the host. The Host code for OpenCL is very verbose as it is designed to work for every platform and several libraries try to alleviate this complexity. One of the main problem with OpenCL is that it is mostly C compliant and is therefore hard to mesh with high level programming without having to redesign around OpenCL.

1.4.2 Domain Specific Languages

By definition, a Domain-Specific Language (DSL) is a computer language designed to fit a specific application domain. This is contrary to a general-purpose language, which is widely applicable across multiple domains and thereof lacks features for a specialized domain. These languages focus on simplifying specific tasks for developers by removing the complexity of a programming language. Adding this layer of abstraction helps developers to focus on the specificities of the domain described by the language, reducing the gap between domain experts and simple developers. DSL also have the effect of increasing the maintainability and ease the validation of codes.

There are a multitude of DSLs available ranging from domain such as editing documents to expressing physical problems. Tex, a markup language, is an example of a widely used DSL for document editing created by Donald Knuth in 1977. Tex offers control over document formating with commands to help properly structure a text without having to focus too much on it. Other examples of well know DSLs are the Structured Query Language (SQL) to exploit databases or Hyper Text Markup Language (HTML) to create web pages.

In numerical computing, MATLAB^{TM} [65], a multi-paradigm numerical environment, is the most popular DSL. The original purpose of MATLAB what to ease the use of the advanced LINPACK library without having to know any FORTRAN. MATLAB is mainly focused on matrix manipulations by exploiting optimized libraries such as LAPACK for CPU while providing utility tools for function plotting and data management. It is also interfacing with other languages such as C, C++, Java, FORTRAN or Python.

DSLs are usually part of a developing environment as they cannot be used as stand alone. To implement a DSL, one needs to write a compiler/interpreter for its language as it is not part of a general purpose computing language. This is a heavy cost as developing a DSL require more than just writing a language and can often suffer from performance issues due to the lack of optimizations. Another approach that we use in our work is based on Domain Specific Embedded Languages (DSEL) that we describe next.

1.4.3 Domain Specific Embedded Languages

Domain Specific Embedded Languages (DSELs) are a subclass of DSL that rely on an existing general-purpose language to host it. DSELs then reuse the host language syntax and tool ecosystem to be compiled or interpreted. This means inheriting from the languages constructs (structure, conditions, functions, etc) while adding primitives that allow programmer to have access to a higher level of abstraction. Differences and advantages of DSELs over DSL are discussed by Abrahams [3, p.226]

There are several forms of DSEL available in different languages. Javascript is in a sense an embedded language in an HTML page, while HTML is a DSL. In scientific computing, DSEL rely on a generic language that is flexible and expressive enough to enable the development of a subset language with a definite syntax. This technique is know to be used in languages such as Lisp, Haskell, SmallTalk, C++, Scala and more. It is not always possible to reproduce a specific syntax in a general purpose language and sometimes a symbol needs to be replaced. For example, in C++ it is not possible to reuse the colon operator (:) as defined in MATLAB. This is one of the main shortcomings of a DSEL with the fact that it must be implemented inside a library and cannot be added to a language. Designing a DSEL is however often easier as they reuse existing compiler and rely on domain dependent analysis to generate efficient code.

We will describe how we can design a DSEL in C++ using meta-programming techniques in Chapter 2. Our work will focus on building a generic solver for dense linear algebra with multi-architectural support.

1.4.4 Multi-stage programming

Multi-stage programming (MSP) is another approach than DSL and DSEL to increase the expressiveness of a programming language. MSP is defined as a meta-programming technique which consists in writing a program that will be able to generate new code. This means the state of the program will be modified during the compilation phase by the code meta-programmed.

MSP is usually applied by adding specific language extensions to trigger a new compilation phase. As an example, MetaOcaml [125] is a multi-stage language based on Ocaml [86] with three basic constructs for runtime code optimization. It adds type code values and constructs to build them called quoting and splicing. The generated code with MSP can then be stored in a file or compiled and linked directly to a Ocaml program enabling run-time code optimization.

More recent adoptions of *MSP* for parallel computing include techniques like Lightweight modular staging (LMS) [112] in SCALA. Instead of using quasi-quotation LMS is based on types and uses an environment for MSP. Similarly to DSEL, using MSP with a host language requires this language to be flexible enough to adapt to MSP. Other approaches include Language Virtualization [24] (base on LMS) or Terra [39] using language extensions for HPC in LUA.

We will describe our work on MSP in Chapter 3 based on the C++ language and the CMake tool for the multi-level code generation.

1.5 Considered language and library

1.5.1 C++ language and programming techniques

Generic programming is a paradigm that aims at achieving reusable, adaptable libraries. Pioneered by Alexander Stepanov and David Musser, Generic Programming obtained its first major success when the Standard Template Library became part of the C++ Standard. The process followed in generic programming is to first find the similarities among different implementations of a same algorithm in the form of **concepts** [61] , and then provide an abstraction that can match all of these implementations. **Concepts** are a subtle mix of constraints and axioms used to represent the problem. The constraints define the statically evaluable predicates on the properties and syntax of the system, while axioms state the semantic informations about the types requirements, providing a better understanding of the problem. This process, called **lifting**, is then repeated until the generic algorithm has reached a suitable level of abstraction, where it provides maximal re-usability without sacrificing performance:

- Step 1 : Find similarities among different implementations
- Step 2 : Provide an abstraction that can match all of those implementations
- Step 3 : Iterate until a satisfactory level is reached

Generic Programming and Concepts provide support for Concept Checking, which allows for a limited support of parametrized types constraints [118], simplifying error checking at compile-time. In C++, concepts are being studied to be integrated in the standard as extension to the template system. They have been published as an ISO Technical Specification ISO/IEC TS 19217:2015.

Generative Programming is about bringing the benefits of automation to software development. Following this paradigm, any complex software system can be broken down to a list of interchangeable components which tasks are clearly identified and a series of generator that combine components by following rules set by an a priori domain specific analysis. The application of generative programming to parallel programming and hardware synthesis can be seen as a way to leverage the classical limitations of classic, library based tools by encapsulating expertise at the source code level and not only at the binary level. One of the most classical techniques for generative programming is Template Meta-programming.

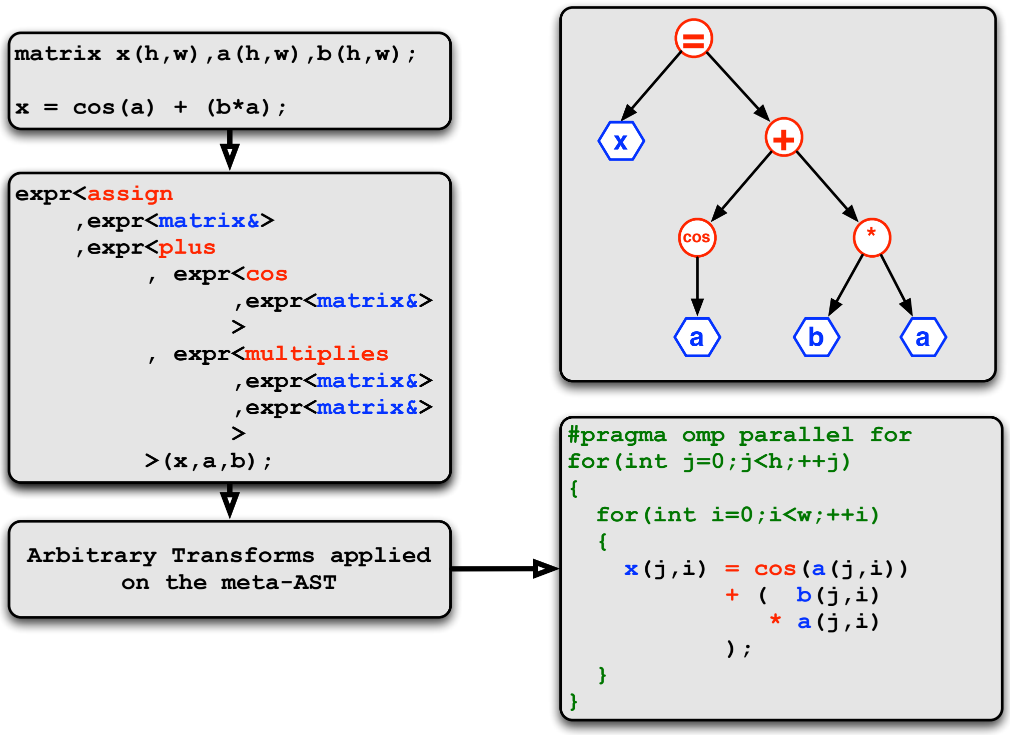
Template Meta-programming is a meta-programming technique in which templates are used by a compiler to generate temporary source code, which is merged with the rest of the source code and then finally compiled. The output of these templates includes compile-time constants, data structures, and complete functions. The use of templates can be thought of as compile-time execution. This technique is used by a number of languages, the most well-known being C++ [3], D [22], Haskell [116] and OCaml [124].

1.5.2 The NT^2 library

NT^2 [48] is a numerical computing C++ library implementing a subset of the MATLAB language as a *DSEL* . The work that we described in Chapter 2 and 3 are extensions for this library. NT^2 simplifies the development of data-parallel applications on a large selection of architectures currently including multi-core systems[128] with SIMD extensions. Simply put, a MATLAB program can be converted to NT^2 by copying the original code into a C++ file and performing minor cosmetic changes (defining variables, calling functions in place of certain operators). NT^2 also takes great care to provide numerical precision as close to MATLAB as possible, ensuring that results between a MATLAB and an NT^2 code are sensibly equal.

Internally, NT^2 is designed to leverage the well known *Expression Templates* [132] C++ idiom to build at compile time a flexible representation of the abstract syntax tree of any C++ expression containing at least one NT^2 component. This compile-time tree is then transformed in actual code to be executed on a parallel system. Contrary to other libraries based on the same technique, NT^2 relies on BOOST.PROTO , an external *Expression Templates* system to handle the creation and transformation of Abstract Syntax Trees (AST) [98]. BOOST.PROTO allows us to replace the direct walk-through of the compile-time AST done in most

C++ *DSELs* by the execution of a mixed compile-time/runtime algorithm over the predefined AST structure generated by Boost.Proto (fig. 1.9).



```
matrix x(h,w),a(h,w),b(h,w);

x = cos(a) + (b*a);
```

```
expr<assign
    ,expr<matrix&>
    ,expr<plus
         , expr<cos
               ,expr<matrix&>
               >
         , expr<multiplies
               ,expr<matrix&>
               ,expr<matrix&>
               >
         >(x,a,b);
```

```
#pragma omp parallel for
for(int j=0;j<h;++j)
{
  for(int i=0;i<w;++i)
  {
    x(j,i) = cos(a(j,i))
           + ( b(j,i)
             * a(j,i)
           );
  }
}
```

Figure 1.9: *Expression Templates* in NT^2

Finally, the other main difference between NT^2 and similar tools is the fact that architectural specificities of the generated code are handled by an execution model based on **Algorithmic Skeletons**[27]. Those skeletons simplify the extension of NT^2 for various hardware by separating the concerns of optimizing the AST code for different type of architecture. The main element of NT^2 is the `table` class. `table` is a template class that can be parametrized by its element type and an optional list of settings. Instances of `table` behave like Matlab array and supports the same operators and functions. NT^2 covers a very large subset of Matlab functionality, from standard arithmetic, exponential, hyperbolic and trigonometric functions, bitwise and boolean operations, IEEE related functions and of course linear algebra.

```
// Matlab: A = 1:1000;
table<double> A = _(1.,1000.);

// Matlab: B = A + randn( size(A1) );
table<double> B = A + randn(size(A1));

// Matlab: r = sqrt(sum((A(:)-B(:)).^2)/numel(A));
double r = sqrt(sum(sqr(A(_)-B(_))) / numel(A));
```

Listing 1.5: NT^2 RMSD Computation

Listing 1.5 showcases some NT^2 basic features including the mapping of the colon function `(:)` to the `_` object, various functions, a random number generator and some utility functions like `numel` or `size`.

In Chapter 2, we will explain how to build a hybrid programming model and how we have integrated ours inside NT^2 . In Chapter 3, we will tackle the problem of MSP and explain how we can integrate it inside a C++ library using expression templates and the CMake tool.

1.6 Conclusion

In this chapter, we have presented the linear algebra algorithms we will work with and their various versions in state of the art high performance dense linear algebra libraries. Then we have described the architectures and their different evolutions that have gone in hand with the development of new libraries to support these architectures.

We have also given a general description of the various programming tools available for these architectures. This is done as a reminder of how complex the architectural landscape and the various tools available is. We have then mentioned high level programming techniques to develop more generic and flexible software that can be adaptable to new architectures. Finally, we have described some of the C++ techniques we rely on and the library in which some of our code is implemented.

From what we have seen, it is clear that the diversity of architectures, which we have not all described, combined with the amount of libraries and programming languages available make for a very large landscape. In this manuscript, we will explain in detail techniques that can alleviate such problems. These techniques are implemented using the C++ language but are not exclusive to this language. As of now, C++ is a widespread high-level language in HPC. It is unclear which language will take this role in the future, but we can say that the programming techniques employed will have a place in it. It is in this context that we present our work.

In the next chapter, we will present one of our contribution in the form of a multi-architectural generic solver for dense linear algebra that is expendable to new architectures. We will show how to design simple GPU containers and a consistent memory model that will also be used in Chapter 3.

Chapter 2

High-level programming for dense linear algebra

Contents

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.

The work presented in this chapter was accepted and presented at Ispa 2015 [93]. The contributions of this chapter are :

- The definition of a generic hybrid programming model that can be used as a stand-alone
- An architecture aware binding between NT^2 and LAPACK/MAGMA based on type tags to dispatch between the different architectures and runtime backends in an extensible way.

- 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 chapter is organized as follows: in Section 2.1 we give the software context and related work on high level libraries. Then, in Section 2.2, 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. After that, we explain in Section 2.3 the concept behind our GPU container and how it should interact with the CPU. In Section 2.4.3, we detail our methods to develop efficient dense linear algebra software. We present in Section **??**, as an example of an application, the code generation of a mixed-precision linear least squares solver. We give performance comparisons on CPU and GPU using respectively to the QR routines from LAPACK and MAGMA. To our knowledge such a solver does not exist in public domain libraries LAPACK, PLASMA [126] and MAGMA. Concluding remarks are given in Section 2.5.

2.1 Related Work

A major concern when developing dense linear algebra software is to propose a user-friendly Application Programming Interface (API) that provides similar performance as BLAS-like 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. In our study we consider the dense linear algebra libraries LAPACK (serial library for CPU processors) and MAGMA (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 mostly 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 (see Figure 2.1). However, performance inhibits the flexibility and adaptability of libraries.

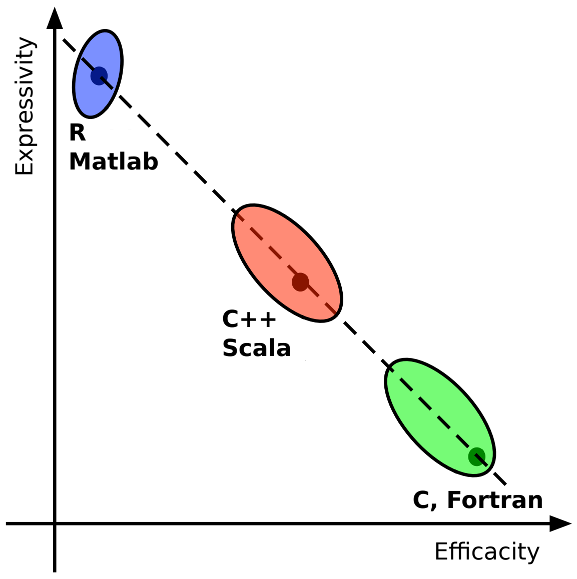


Figure 2.1: Abstraction/Efficiency trade-off for languages

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 [131] is a good example. It offers a framework to develop dense linear solvers using algorithmic skeletons 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 [110] for signal processing or other linear algebra libraries such as Design by transformation [90], Build to Order [119], or Hydra [87] that develop code generation through the use of a *Domain Specific Language* (or DSL) to express data dependencies. A more closely related work is a linear algebra compiler [51]. It is based on the Mathematica language and optimizes the algorithms by reusing variables and mapping to BLAS function calls using problem specific knowledge [50]. By definition, a DSL is a computer language specialized for a particular application domain. This implies that the use of a DSL requires a preprocessor 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 Integrated development environment (IDE) [54]. Since writing a compiler is very complex and time-consuming while often not being 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* or other *generative programming* [35] principles to design a *Domain Specific Embedded Language*. 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 (see Section 1.4.3). 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 penalties and slows down algorithms. Examples of such libraries are Armadillo [28] and MTL [58]. Armadillo provides good performance with BLAS and LAPACK bindings and an API close to MATLAB 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 [64], Flens [85], Ublas [134] and Blaze [73].

Our objective in this chapter 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, 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” [132].

2.2 Generative programming for designing numerical libraries

2.2.1 Optimization approaches based on a configuration space

As stated in Section 2.1, 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* [130], 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 [137] 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 [7] 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 [133] that will complement the compilers work. One way to do this is to use *generative programming*.

2.2.2 Generative programming in software development

Generative programming (see Section 1.5.1) consists of bringing the benefits of automation to software development. Following this paradigm, a model can be drawn to implement the different components [31] 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 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.

2.2.3 Domain engineering methods for active libraries

We call *active libraries* [30] a technique which combines a set of *generative programming* and *meta-programming* methods to solve the *abstraction/efficiency trade-off problem* mentioned in Section 2.1. 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 methodol called *Domain Engineering Method for Reusable Algorithmic Libraries* [33] (or DEMRAL depicted in the blocks 1-3 of Figure 2.2) 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 2.2). 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 [52]. Once these configurations have been implemented, it is possible to program the parametric components that they represent (see Block 2 in Figure 2.2). 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 2.2). In NT^2, this corresponds to the solver we have implemented which will be described later on.

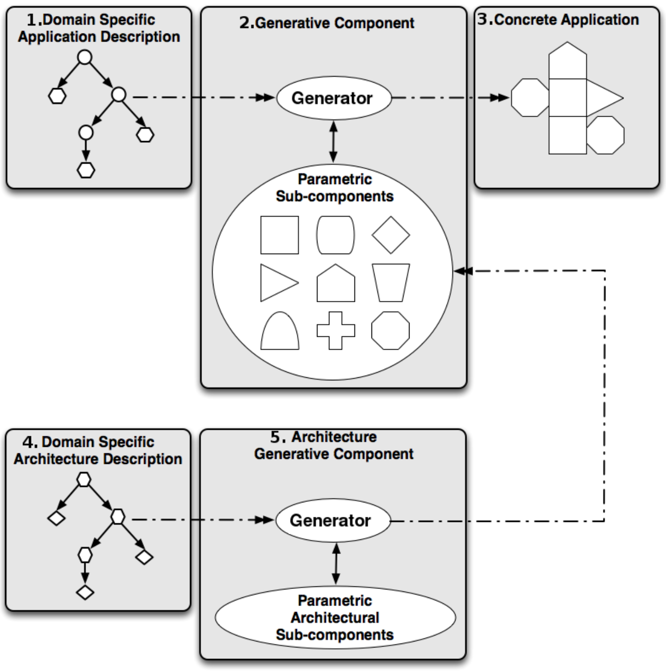

1.Domain Specific Application Description
2.Generative Component
3.Concrete Application
Generator
Parametric Sub-components
4.Domain Specific Architecture Description
5. Architecture Generative Component
Generator
Parametric Architectural Sub-components

Figure 2.2: Overview of the NT^2 skeleton based generation process

DEMRAL can be seen as a specialization of a paradigm like object-oriented programming, aspect programming or model driven engineering [114]. 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 method provides a high re-usability, allowing components to be customized while retaining the efficiency of statically configured code [96]. We extend it by adding an architectural layer in the design with a "Domain specific architecture description" (block 4 of Figure 2.2) and a specialized generator for GPUs (block 5 of Figure 2.2) based on this description [47]. In NT^2 , 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.

2.3 Building a CPU/GPU programming model

2.3.1 GPU container concept

Integrating GPU support in a library that only supports CPUs requires to decide at which level and how this integration should take place. In this work we will focus on CUDA integration to exploit the MAGMA library. This can simplify our task if we focus on calling existing functions that have already been compiled by NVCC. To details this, a major problem with the CUDA environment is the need of a special compiler for GPU code. This means a library compiled with a standard C++ compiler will not be able to seamlessly support CUDA code as it will require a new compilation step. It is also not possible to compile modern C++ code with NVCC as the compiler does not support properly the standard. In the situation where we have access to a binary, it is possible to directly call the function from a code compiled in C++.

Our approach will then focus on creating a minimalist GPU vector that will encapsulate the pointer and basic functionalities. This will enable us to have a recognizable structure for GPU data while having no need to pass through NVCC. The Thrust library [17] uses an opposite model where it allows random data access on a GPU vector but requires the code the be in .cu file and compiled with NVCC. In our implementation, we declare an element access operator [ ] for conformity reasons but it is not usable (will make a static assert).

2.3.2 Integration in NT^2

If the single system computation model of NT^2 is rather classic, we needed a way to handle accelerators in a generic and extensible way. The first step is to provide a simple way to **locate** tables (see Section 1.5.2) on either the host system or on a device. This is simply done by using a couple of settings in the definition of NT^2 table instances. In the Section 2.3.1 we described a low level vector. Tables are high level entities that include a container with settings on the locality (which we added), allocator type and many more. Listing 2.1 shows the `nt2::host_` and `device_` settings that are used to specify if a `table` contains data stored on the host memory or device memory. The device flag will make it so the table uses our GPU vector as a back-end with its restrictions.

```
// Generates a host table by default
table<double> A( of_size(1e3,1e3) );

// Generates a host table explicitly
table<double,host_> A2( of_size(1e3,1e3) );

// Generates a device table
table<double,device_> D( of_size(1e3,1e3) );

// Generates a host pineed table with CUDA
table<double,pinned_> D2( of_size(1e3,1e3) );
```

Listing 2.1: NT^2 host and device specifications

Note that a special function `on_device` can be used to specify on which device the memory must be allocated in the case where multiple devices are available.

Semantic of operations between host and device tables is quite straightforward as they will be carried on the proper memory segment of each table. When mixing tables of different location, memory transfers are implicitly performed. This means that assigning a host table to a device table is equivalent to performing a CUDA memory transfer. This can be used for example to simplify interaction with existing GPU kernels as shown in listing 2.2. As streams are not assigned to tables, this transfer will be synchronous. A copy function is also available to perform asynchronous memory transfers when a non-default stream is given.

```
// X is a 1e3 x 1e3 matrix full of 1.
table<double> X = ones(1e3,1e3);

// Transfer to device
table<double,device_> Y = X;

// cuBLAS direct call
cublasDscal( Y.size(), 5. , Y.data(), 1.);

// Transfer back to host
X = Y;
```

Listing 2.2: NT^2 interaction with cuBLAS

This semantic of transfer by assignment is a classical way of performing such operation transparently. It as been used by tools like Thrust or VexCL [37] and have been proved to be easy enough for the user while allowing for fine grain performance tuning.

The current GPU programming model often requires to use pinned memory on the CPU for optimal transfer time and computation overlap. For this reason, we have added a CUDA pinned allocator that will use the CUDA CPU allocation function instead of a standard CPU alloc (Listing 2.3).

```
// X is a 1e3 x 1e3 matrix full of 1 allocated in CUDA pinned memory.
table<double,pinned_> X = ones(1e3,1e3);

// Faster transfer to the device
table<double,device_> Y = X;

// cuBLAS direct call
cublasDscal( Y.size(), 5. , Y.data(), 1.);

// Faster Transfer back to host
X = Y;
```

Listing 2.3: NT^2 cuda CPU allocator

To enable support for IGP, we have also extended this model with new functionalities. As there is no simple way as of now to detect if the system is an IGP or not, we have added a compile flag (NT2_CUDA_INTEGRATED=ON/OFF, OFF is by default). This will activate specific parts to the CUDA integrated part such as flags definitions (Listing 2.4) using the const object setup idiom [1]. In case the pinned allocator is used for the CPU, it will automatically call the cudaHostAlloc function with the correct flag cudaHostAllocMapped.

```
#if defined(NT2_CUDA_INTEGRATED)
struct integrated_cuda_settings
{
  integrated_cuda_settings() { cudaSetDeviceFlags(cudaDeviceMapHost); }
};

static const integrated_cuda_settings proper_settings= {};
#endif
```

Listing 2.4: Initialize CUDA for IGP

As on IGP we do not want to do transfers between CPU and GPU using the cudaMemCpy function, we have to provide functions that can satisfy these conditions. We have designed two functions called to_host and to_device that provide these functionalities.

```
// X is a 1e3 x 1e3 matrix full of 1 allocated in CUDA pinned memory.
table<double,pinned_> X = ones(1e3,1e3);

// Will create a view of X on IGPs
auto Y = to_device(X);

// cuBLAS direct call
cublasDscal( Y.size(), 5. , Y.data(), 1.);

// Will not do the transfer on IGPs
to_host(Y,X);
```

Listing 2.5: NT^2 IGP functions

[1]This technique will be superseded in 2017 with the adoption of inline variables

The function to_device will verify if the data of X is pinned and if the flag NT2_CUDA_INTEGRATED is activated (Listing 2.5). If that is the case, the function will return a view of the entry table using its semantic information. If one of these conditions are not met, the type returned will be a table on the GPU and data from X will automatically be transfered. The to_host function will also check if the architecture is an IGP and if the output table is allocated as pinned. If these conditions are met, it will apply no modifications to Y. Otherwise, data will be transfered from Y on the GPU to X on the CPU. It is important to note that for IGP the function to_host holds a double purpose. Due to the IGP model, it is possible to run computations from the same memory space on GPU and CPU simultaneously. This means that if the user is not careful enough he can compute on the same data concurrently removing consistency between results. The function to_host will therefore apply a synchronization to make sure the computation is finished. Another solution is to use the CUDA_LAUNCH_BLOCKING=ON flag.

2.4 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.

2.4.1 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 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 2.1, 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 2.2.2) 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 2.2.1). These parameters are not mutually exclusive and can be extended/combined.

Table 2.1: 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:

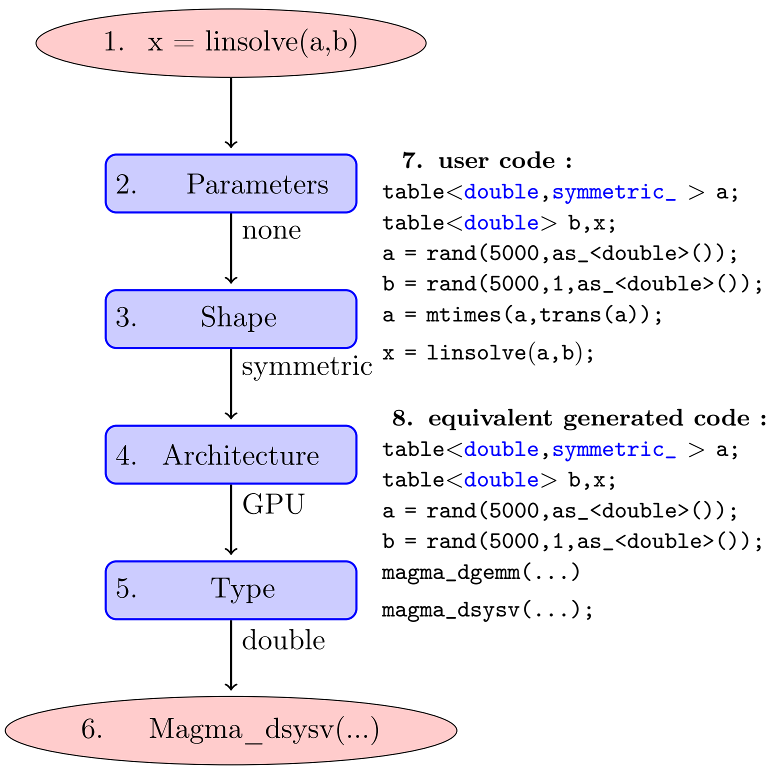
```
x = nt2::linsolve(nt2::ill_conditioned_(a),b);
```

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 3.2 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 3.2. When the code is analyzed by the C++ compiler, the call corresponding to Part 1 of Figure 3.2 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 3.2) 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.



7. user code :

```
table<double,symmetric_ > a;
table<double> b,x;
a = rand(5000,as_<double>());
b = rand(5000,1,as_<double>());
a = mtimes(a,trans(a));
x = linsolve(a,b);
```

8. equivalent generated code :

```
table<double,symmetric_ > a;
table<double> b,x;
a = rand(5000,as_<double>());
b = rand(5000,1,as_<double>());
magma_dgemm(...)
magma_dsysv(...);
```

Figure 2.3: Example of a generation process for a symmetric system

2.4.2 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, while the second uses a dynamic approach like in SkePU [46].

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.

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 2.4 correspond to the following call made in NT^2 through *linsolve* that can run either on CPU or GPU.

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

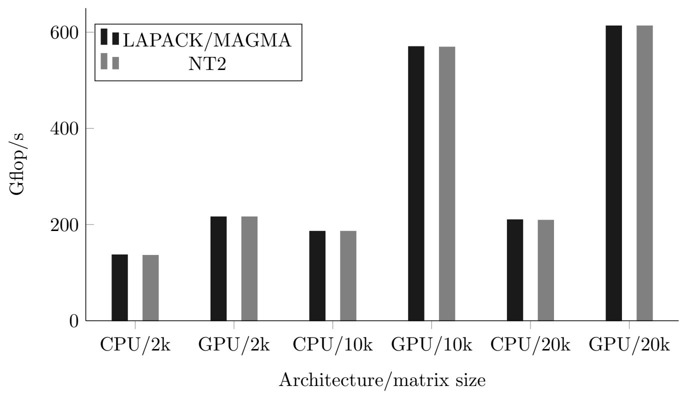


Figure 2.4: Performance comparison between LAPACK/MAGMA routines and generated codes via NT^2 for general dense linear system solution

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.

2.4.3 Application: example of a least squares solver

In this section we illustrate how the *generative programming* method described in Section 2.2 can be used to generate automatically new implementations of algorithms and achieve satisfactory performance.

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 $\text{cond}(A)^2$ [20, 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 [19] 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 [20, 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 $Aw = r$
4. Corrected solution $y = \tilde{x} + w$

It is shown in [69, p. 392] that, if $\text{cond}(A)^2 u \leq 1$ (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.

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% [10] 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 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 [38] 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 [13] is smaller than the inverse of the lower precision used (*i.e.* here 10^8).

Algorithm 8 Mixed-Precision CSNE

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

The first step of Algorithm 8 in NT^2 is implemented in Listing 2.6 with *triu(qr)* , while the second and third steps are performed by the following calls to *mtimes* and *linsolve*. Note that the code is similar in terms of syntax and number of instructions to what would be written in MATLAB.

```
table<double> mcsne(table<double> const& A, table<double> const& B)
{
  double anrm = lange(A,'I');
  double cte  = anrm*Eps<double>()*nt2::sqrt(width(a));

  table<float> SA = cast<float>(A);

  table<float,upper_triangular_> SR = triu(qr(SA));
  table<float> SX = mtimes(trans(SA),cast<float>(B)));

  SX = linsolve(trans(SR),SX);
  SX = linsolve(SR       ,SX);

  table<double> X = cast<double>(SX);
  table<double> E = B - mtimes(A,X);

  std::size_t i = 0;

  do
  {
    SX = cast<float>(mtimes(trans(A),cast<float>(E)));
    SX = linsolve(trans(SR),SX);
    SX = linsolve(SR       ,SX);

    E = cast<double>(SX)

    double RNRM = maximum(abs(E(_)));

    X += E;
    double XNRM = maximum(abs(X(_)));

    E = B - mtimes(A,X);
    ++i;
  } while( not(RNRM < XNRM*cte) && (i<max_iter));

  return X;
}
```

Listing 2.6: NT^2 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_);
```

Here in Listing 2.7 we show the corresponding code without using `linsolve` and implementing directly with MAGMA. It produces the same result as our code described in Listing 2.6 but is much more verbose and dependent on MAGMA functions. We still use our simple cuda_buffer to allocate simply data on the GPU and for automatic deallocation.

```
int lda,m,na,nhrs,ldb // standard lapack definitions
int nb = magma_get_dgeqrf_nb(m);
int lwkopt = (2*std::min(m,na)+ ((na+31)/32)*32)*nb;

nt2::container::table<double> x(nt2::of_size(ldb,nhrs));
nt2::container::table<float> tau(nt2::of_size(std::min(m,na),1));
nt2::memory::cuda_buffer<double>     dA(m*na,a);
nt2::memory::cuda_buffer<double>     dB(ldb*nhrs,b);
nt2::memory::cuda_buffer<double>     dX(ldb*nhrs);
nt2::memory::cuda_buffer<double>     dE(ldb*nhrs);
nt2::memory::cuda_buffer<double>     temp(ldb*nhrs);
nt2::memory::cuda_buffer<float>      dSR(m*na);
nt2::memory::cuda_buffer<float>      dSX(ldb*nhrs);
nt2::memory::cuda_buffer<float>      dT(lwkopt*nhrs);

double eps = boost::simd::Eps<double>();
double anrm = magmablas_dlange(MagmaInfNorm, m, na, dA.data(), m, dE.data()
    );
double cte = anrm*eps*nt2::sqrt( type_t(na));
double xnrm, rnrm;

magmablas_dlag2s(m,na,dA.data(),m,dSR.data(),m, &info);
magmablas_dlag2s(ldb,nhrs,dB.data(),ldb,dSX.data(),ldb,&info);
magmablas_sgemv( MagmaTrans, m, na, one, dSR.data(), m, dSX.data(), one, 0,
    dSX.data(), one );
magma_sgeqrf2_gpu( m, na, dSR.data(), m, tau.data(), &info );
magmablas_strsm(MagmaLeft,MagmaUpper,MagmaTrans,MagmaNonUnit,na,nhrs,(float)
    one,dSR.data(),m,dSX.data(), na);
magmablas_strsm(MagmaLeft,MagmaUpper,MagmaNoTrans,MagmaNonUnit,na,nhrs,(
    float)one,dSR.data(),m,dSX.data(), na);
magmablas_slag2d(ldb,nhrs,dSX.data(),ldb,dX.data(),ldb,&info);
magmablas_dlacpy(MagmaFull, ldb, nhrs, dB.data(), ldb, dE.data(), ldb);
magmablas_dgemv( MagmaNoTrans, lda, na, negone, dA.data(), lda, dX.data(),
    one, one, dE.data(), one );

for(size_t i = 1; i<=10;++i){
  magmablas_dgemv( MagmaTrans, lda, na, one, dA.data(), lda, dE.data(), one,
      0, temp.data(), one );
  magmablas_dlag2s(ldb,nhrs,temp.data(),ldb,dSX.data(),ldb,&info);
  magmablas_strsm(MagmaLeft,MagmaUpper,MagmaTrans,MagmaNonUnit,na,nhrs,(
      float)one,dSR.data(),m,dSX.data(), na);
  magmablas_strsm(MagmaLeft,MagmaUpper,MagmaNoTrans,MagmaNonUnit,na,nhrs,(
      float)one,dSR.data(),m,dSX.data(), na);
  magmablas_slag2d(na,nhrs,dSX.data(),ldb,dE.data(),ldb,&info);
  nt2::memory::copy(dE,x);
  rnrm = nt2::maximum(nt2::abs(x(_)));
  magmablas_dgeadd(na,nhrs,one,dE.data(),ldb,dX.data(),ldb);
  nt2::memory::copy(dX,x);
  xnrm = nt2::maximum(nt2::abs(x(_)));
  magmablas_dlacpy(MagmaFull, ldb, nhrs, dB.data(), ldb, dE.data(), ldb);
  magmablas_dgemv( MagmaNoTrans, lda, na, negone, dA.data(), lda, dX.data(),
      one, one, dE.data(), one );
  if(rnrm < xnrm*cte) { break; }
}
 nt2::memory::copy(dX,x);
 return x(_(1,na));
```

Listing 2.7: Implementation of MCSNE with MAGMA

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 [74] version 10.2.3, MAGMA 1.9 with CUDA 7.0 [101] and gcc 4.8 [59]. We compare ourselves with Eigen 3.2.5 and Armadillo 5.200.2. As a reminder, computations on the GPU are run using the IEEE 754 compliant mode for arithmetic operations and with CUDA ECC activated. The random test problems were generated using the method described in [103]. Performance results include data transfers between CPU and GPU and data.

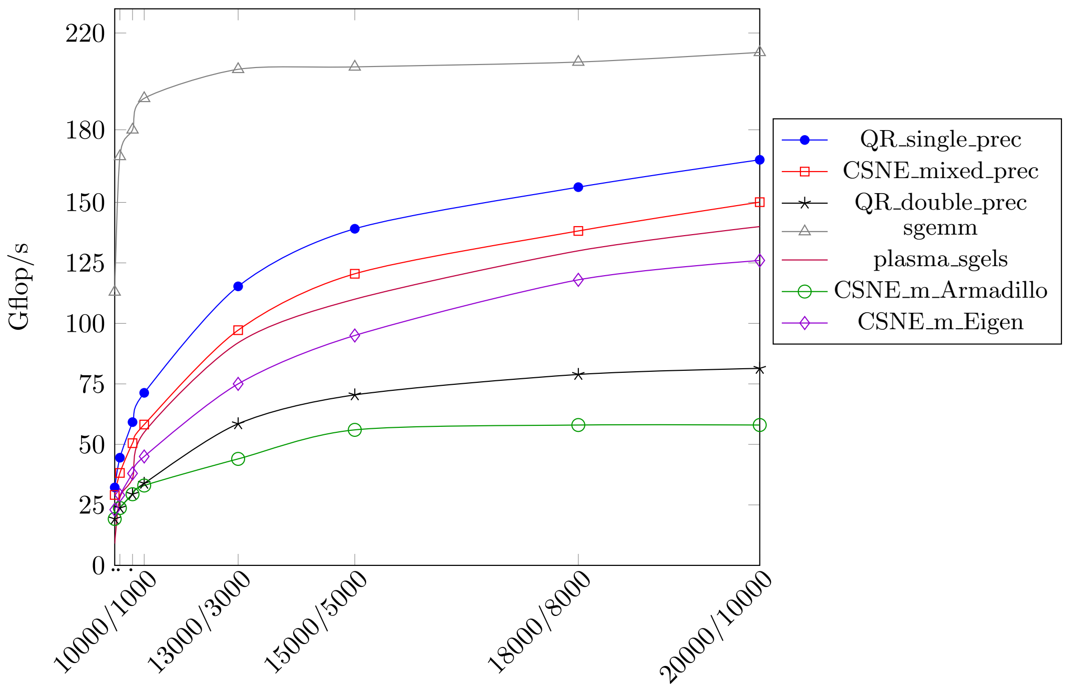


Figure 2.5: Performance results of generated code on CPU

In Figure 2.5 we compare the performance of CSNE_mixed_prec 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 than that 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.

If we compare our results with Armadillo, we can see that the implementation of MCSNE with Armadillo (Listing 2.8) is slower than `dgels`. While the performance of `dgels` in Armadillo is the same as in NT^2, it is much harder to obtain high performance when not relying entirely on the BLAS. For examples, the extraction of the Q,R matrices are very expensive and the cast is not vectorized.However, as Armadillo aims toward a good balance between speed and ease of use, this is justified. The interface is intuitive and the documentation is well structured which makes writing code quite appreciable.

```
mat A = randu<mat>(h,w);
vec B = randu<vec>(h);
double n = 0 , n1 = 0;

fmat Q,R; Q.set_size(h,w); R.set_size(h,w); Q.zeros() ; R.zeros();

fmat a = conv_to<fmat>::from(A);

qr_econ(Q,R,a);

fvec b = conv_to<fvec>::from(B);

fvec sx = trans(a)*b;

sx = solve(trans(trimatu(R)), sx);

sx = solve(trimatu(R), sx);

vec X = conv_to<vec>::from(sx);

vec e = B - A*X ;

for (std::size_t i =1 ; i <5 ; ++i)
{
        sx = conv_to<fvec>::from(trans(A) *e);

        sx = solve(trans(trimatu(R)),sx);

        sx = solve(trimatu(R),sx);

        e = conv_to<vec>::from(sx);

        X = X + e ;

        e = B - A*X;
}
```

Listing 2.8: CPU Implementation of MCSNE with Armadillo

Eigen (Listing 2.9) on the other side manages to reach better performance but is still slower than NT^2 . This is due to better memory management in NT^2 and some optimization functionalities that are not present in Eigen. The Eigen is however harder to use to implement a version of MCSNE and also less readable. Some of the code for MCSNE had to be hand-written as Eigen could not properly extract the upper triangular matrix R. Having an implementation where the solver is a member function of the matrix/solver makes the code look less readable and lose some of its generality and flexibility.

On the GPU, the performance of MCSNE in NT^2 is depicted in Figure 2.6. It also approaches 90% of the performance of `magma_sgels` on GPU (QR_single_prec) 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 [8, p. 15] for the LU factorization. Neither Armadillo nor Eigen offer proper support for GPU to allow us to write an MCSNE version.

```
MatrixXd A = MatrixXd::Random(h,w);
  VectorXd B = VectorXd::Random(h);
  double n = 0 , n1 = 0;

  MatrixXf a = A.cast<float >();

  HouseholderQR<MatrixXf> qr(a);

  MatrixXf rf = qr.matrixQR();

  VectorXf b = B.cast<float >();

  VectorXf sx = a.transpose()*b;

  size_t cols = a.cols();
  size_t rows = a.rows();
  MatrixXf sr(cols,cols);
  size_t size_n = sizeof(float)*(cols);

for(size_t i=0; i<cols; i++)
{
  std::memcpy(sr.data()+i*cols, rf.data()+i*rows, size_n);
}

  sx = sr.triangularView<Eigen::Upper>().transpose().solve(sx);

  sx = sr.triangularView<Eigen::Upper>().solve(sx);

  VectorXd X = sx.cast<double>();

  VectorXd e = B - A*X ;

  for (std::size_t i =1 ; i <5 ; ++i)
  {
    sx = (A.transpose()*e).cast<float >();

    sx = sr.triangularView<Eigen::Upper>().transpose().solve(sx);

    sx = sr.triangularView<Eigen::Upper>().solve(sx);

    e = sx.cast<double>();
    X = X + e ;
    e = B - A*X;
  }
```

Listing 2.9: CPU Implementation of MCSNE with Eigen

In Section 1.3.4, we described the concept behind the IGP architecture and current available architectures. We also note that current IGPs are not adapted for double precision. In Listing 2.7, we can see that the DGEMM routine on the NVIDIA Tegra X1 has very poor performance compared to the SGEMM. This is due double precision computations being emulated on the GPU.

An algorithm using mixed-precision will therefore be performing much better than in double precision since most computations are done in single precision. This is confirmed in Figure 2.7. The implementation of MCSNE that achieves good performance does not use directly MAGMA routines. The reason is that MAGMA does not have proper support for IGP and most computational routines are hybrid.

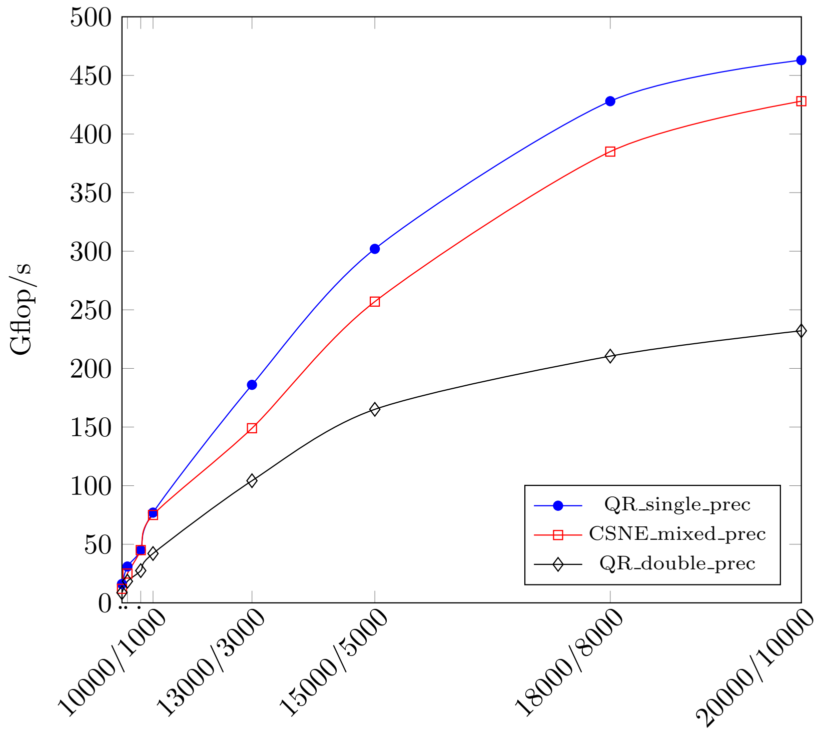


Figure 2.6: Performance results of generated code on GPU

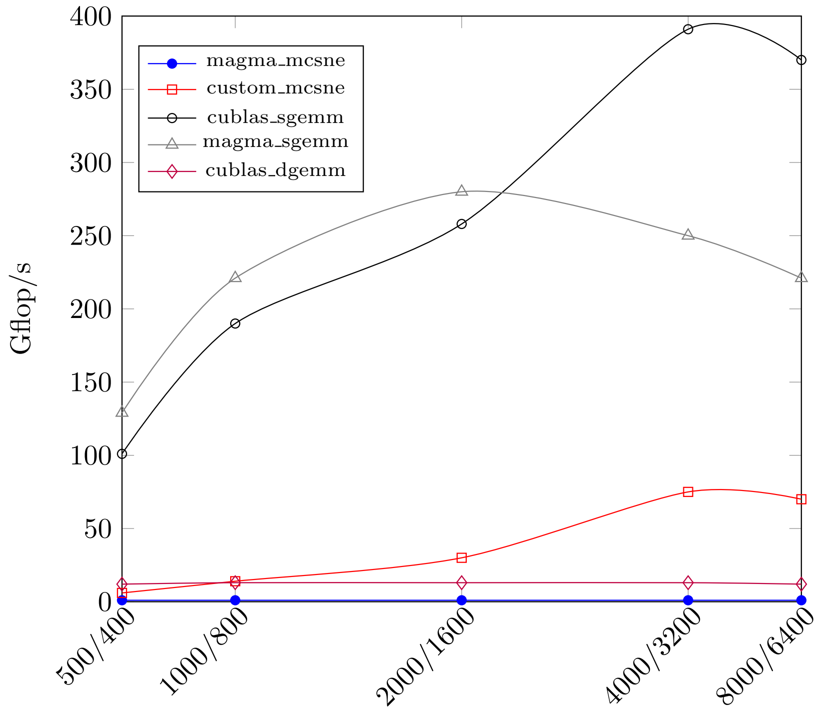


Figure 2.7: Performance results on the Tegra X1 GPU

MAGMA routines end up doing memory transfers between the CPU and GPU instead of using a zero-copy model. On IGPs, this is very costly, thus limiting the performance of the mixed-precision MAGMA routine for QR to 1 Gflop/s. It is also interesting to note that the MAGMA and CUBLAS implementations of SGEMM do not perform the same, with one being more efficient than the other depending on the size of the problem. This clearly shows that the size of the problem has an impact on how the code should be written for current IGPs.

2.5 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 [9, 14] or communication-avoiding algorithms [11] for dense linear systems. Moving to sparse problems is also a possibility where libraries like Cups [16] or VexCL provide an interesting approach. Raising the level of expressiveness stays a major concern while trying to add content in NT^2 .

In the next chapter, we will explain how we have build a multi-stage programming tool which is included in NT^2 for GPU operations.

Chapter 3

Multi-stage programming in C++ for GPGPUs

Contents

Developing large applications in a simple, fast and efficient way has always been an issue for software developers. As computing hardware complexity rose with the advent of SIMD, multi-processor, multi-core systems and more recently accelerators like GPUs [102] or Intel Xeon Phi [29], software design methods did not undergo the same amount of changes. This complicates the exploitation of such hardware in mainstream applications.

Designing *DSL* has been presented as a solution to these issues. As *DSL*s allow solutions to be expressed in their programming idiom with a level of abstraction equivalent to the problem domain, the maintainability and quality of code is increased. One of the most popular examples is MATLAB which provides a large selection of toolboxes that allow a direct expression of high-level algebraic and numerical constructs in a easy-to-use imperative language. In this scope, **Domain Specific Embedded Languages** (or *DSEL*s) [71, 129] are languages implemented inside a general-purpose, host language [34]. without the requirement of a dedicated compiler or interpreter as they are designed as a library-like component.

In Chapter 2, we presented a method to take advantage of heterogeneous architectures using *DSEL* and providing support current HPC libraries. The problem of not being able to write simple CUDA kernels was however not addressed. Here, we wish to present an approach based on multi-stage programming that can circumvent these problems.

NT^2 – The Numerical Template Toolbox – is such a *DSEL* using C++ template meta-programming to provide a MATLAB -inspired API while supporting a large selection of parallel architectures and keeping a high level of expressiveness. NT^2 was designed to support architectural features like SIMD extensions and multicore programming. However, the support for accelerators like GPGPUs was limited as GPU kernel compilers where unable to process NT^2 C++ 11 based implementation of C++ based *DSEL* . In chapter 2, we addressed the first part of the problem by providing a method for GPU based computing for dense linear algebra kernels.

In this chapter, we present a new method and extension for NT^2 that takes care of such accelerators, especially CUDA based GPUs through multi-stage programming [44] (or *MSP*). *MSP* consists in doing multiple compilation phases allowing for type-safe program generation. The work presented in this chapter was submitted and accepted at ATMG 2016 [94]. Our contributions include:

- A multi-stage method to alleviate the host/device programming model with a cost model for offloading code
- An adaptable strategy to generate CUDA kernel directly from a single C++ source file containing NT^2 statements
- The integration of this kernel generator with existing CUDA libraries like cuBLAS or MAGMA.

The purpose of this system is to provide the user some leeway on how to distribute the data between the host and device through a simple mechanism. As having a perfect cost model for load balancing is very complex to put in place and costly, letting the user provide some insight on data locality is beneficial.

After reviewing the concurrent state of the art software libraries (Section 3.1), we then describe the kernel generator process and how it can be integrated with an existing library (Section 3.2). Finally, we present benchmarks assessing the generated code quality (Section 3.4) and conclude on the future work regarding NT^2 and accelerators.

3.1 Related Work

Software libraries for GPU computing try to simplify the new programming paradigm brought by many-core based systems. The general trend followed in recent years by C++ libraries is to provide a high-level interface through template meta-programming techniques. This goal is reached by providing device containers with architecture-aware generic parallel algorithms and/or a code generation based process. We will give a detailed explanation of both type of libraries that support OpenCL/CUDA or both.

Thrust [70] is a header only library providing a similar interface to the C++ Standard Template Library. Its high-level interface is based on meta-programming and traits to provide efficient parallel skeletons that can run on either CPU or GPU. Container locality is expressed through an explicit container declaration limiting the abstraction but allowing for easier transfers between host and device. Locality for functions is defined by default for device vectors and can be extended with tag dispatching. Overall, it is a well rounded utility library which can be combined with CUDA Toolkit libraries such as CUBLAS, CUFFT and NPP. However it lacks of code generation features and does not support OpenCL.

C++AMP [62] is a framework developed by Microsoft that includes a C++ library and an associated compiler. The C++ AMP programming model includes multidimensional arrays, indexing, memory transfer, tiling, and a mathematical function library. You can use C++ AMP language extensions to control how data is moved from the CPU to the GPU and back. It relies on parallel skeletons such as parallel_for to iterate on data. It does not provide any AST optimization or MSP and is more of an API based approach like Thrust with OpenCL.

SyCLs [123] aim is to simplify programming manycore processors by offering a modern C++ interface on top of OpenCL. SyCL is similar to C++AMP in the sense that it provides single-source development and inlined kernels. That means kernel code can be embedded within C++ code. They have added support for templates in kernel code which is not possible with OpenCL. It does not provide a model using the CUDA language.

VexCL [37] is an expression template library for OpenCL/CUDA. It provides a high level generic interface that is suitable for both back-ends with static parameters defined within a *DSEL* for linear algebra. The expression template mechanism allows for code generation by lazy evaluation of vectors and elementary operations within the AST. Similarly to Thrust, it provides STL-like functions on containers that have a defined locality. It is also possible for the user to define custom functions on device that will be dynamically generated for CUDA. However, the transform used for the generation process requires a unique data locality limiting hybrid algorithms.

ViennaCL [113] is also an expression template library for OpenCL/CUDA. This library strictly follows the uBLAS programming interface and STL like algorithms for easier integration with other softwares. It has implementations for BLAS kernels and high level solvers for sparse and dense computation that provide good performance. Through the mechanism of expression templates it can evaluate basic linear algebra operations with operator overloading. ViennaCL focuses more on OpenCL due to the necessity for separate compilation with CUDA limiting its support through the OpenCL language. It is however possible to generate CUDA code with ViennaCL at runtime.

Boost.Compute [89] is a header only C++ library based on the OpenCL standard. Similar to other libraries, it manages device memory through a designated container. It provides an interesting concept of future for asynchronous copy on the device allowing for more versatility. Boost.Compute also supports closures and adaptable structures for device. Similarly to thrust, it is a well rounded library based on OpenCL to simplify the coding process on accelerators. It however lacks support for numerical analysis and cannot generate CUDA code.

Eigen [64] is a popular library to solve linear systems using expression templates. It should be able to support accelerator code with CUDA by writing Eigen code in a .cu file. It however does not provide any high-level interface or special containers for GPU computing. Since Eigen 3.3 it has a partial experimental CUDA support for Eigen's objects and algorithms within CUDA kernels. Eigen does not support OpenCL.

Kokkos [45] is a C++ developed by Sandia that provides an API for hybrid computations. It does not follow the guidelines of standard C++ libraries and uses view based on a shared pointer. It uses parallel skeletons and lambda as their parallel model. The advantage of Kokkos is that their API is rather simple and can be directly compiled with nvcc simplifying integration. It however does not provide any AST and high level operations.

SciPAL [82] is an expression template library with a *DSEL* for dense and sparse linear algebra. It focuses mainly on recognizing GEMM calls by parsing the AST and regular operations on containers. Its code generation process for CUDA kernels is done mostly by explicitly writing the code in callable objects. The code generation process is then done at runtime for non-BLAS kernels. SciPAL does not support OpenCL.

Feature	Thrust	VexCL	ViennaCL	Boost.C	NT^2	C++AMP	Kokkos
MATLAB API	–	–	–	–	✓	–	–
AST optimization	–	✓	✓	✓	✓	–	–
Device Arrays	✓	✓	✓	✓	✓	✓	✓
CUDA code gen	✓	✓	–	–	✓	–	✓
OpenCL code gen	–	✓	✓	✓	–	✓	–
parallel skeletons	✓	✓	–	✓	✓	✓	✓
CUBLAS support	✓	–	✓	–	✓	–	✓
Static code gen	–	–	–	–	✓	–	–
dense LA solvers	–	–	✓	–	✓	–	–
sparse LA solvers	–	–	✓	–	–	–	–

Figure 3.1: Feature set comparison between NT^2 and similar libraries

In Figure 3.5, we compare features between NT^2 and the previously described libraries. We did not include SciPAL since the code is not available, and Eigen as it does not have any real support for code generation or a device API yet.

The purpose of the previously described libraries is usually to provide a wrapper over the C++ language for GPGPU computing. For this reason, the *DSEL*s based on expression templates or BOOST.PROTO are usually lightweight and consist mostly of overloading elementary operations for containers. The code generation phase then ends up doing a dynamic compilation of the CUDA code/OpenCL kernel which adds a significant overhead on small sized or low arithmetic intensity problems.

Furthermore, as mentioned previously, it is not possible to compile a NT^2 source code with NVCC even with the latest version (7.5). To address these issues, we have added a two step compilation in NT^2 using a serialization mechanism based on Boost.Serialization. Before describing this process, we will first detail our concept behind MSP.

3.2 Multi-stage programming in C++

As described in Section 3.1, code generation for accelerators in C++ is based on Meta-programming techniques. This process is however limited by the fact that C++ does not natively support language extensions for runtime code generation and program execution [124]. Creating a multi-stage paradigm is therefore necessary to remove the runtime cost of compiling CUDA code.

3.2.1 Designing a multi-stage software

We have described in Section 1.4.4 some of the core concepts behind MSP. MSP has been used extensively in fields based on stochastic problems such as energy planning [107] as it relies on two-stage problems. LMS described in Section 1.4.4 is conceptually similar to Meta-programming techniques in C++ applied to the *Domain Engineering Method for Reusable Algorithmic Libraries* [32] which we described in Section 2.2.3. Language visualization described also in Section 1.4.4 goes further by combining *DSEL* techniques with multi-stage programming to cover multiple DSEL (Probabilities, machine learning, Scripting ...) and combine them with heterogeneous hardware. We wish to stay within the same *DSEL* of linear algebra but use multi-stage to remove compatibility problems with accelerator based systems due to compiler incompatibilities. In Chapter 2, we have shown that by relying on compiled libraries such as MAGMA, we can ease the programming model by not needing NVCC. This however limits code development inside a library as it is not possible to directly write CUDA code in NT^2 . What we want to provide is model that supports using existing optimized libraries while being still able to extend them for more flexibility.

MSP is an ideal solution to such as problem as it allows type-safe program generation and is expendable. We differentiate ourselves from other standard MSP approaches by wanting a second step compilation. Most MSP approaches add language extensions to a program which is compiled before the actual program itself. What we wish is a MSP layer that will safely generate accelerator code based on our algorithm skeletons for CPU. We want to perform code generation by providing CMake macro to generate all the required calls to our kernel generation system. CMake is a cross-platform tool for managing the build process of software using a compiler independent method. It has been widely adopted in recently year instead of Makefiles as it provide more abstraction and interoperability. The CMake layer is very lightweight and can be replaced by any generation tool or even simple scripts.

3.2.2 Multi-stage programming in C++

The way we use *MSP* is not based on language extensions but on a method we developed as explained in Section 3.2.1. It is possible to create a multi-stage compilation of a program using this representation. In C++ , we base this process on doing a compilation phase with an incomplete link to generate only an object file. This is depicted in Phases 1-2 of Figure 3.2. Generating an executable program using a compiler is a multi-stage process divided into two main components: **compilation** and **linking**.

By definition, **compilation** refers to the processing of source code files to create object files. This process does not create an executable file for the user. This object file contains the compiled code in binary form of the symbols defined in the source code.

Linking refers to the creation of an executable or a library from an object file set. During the **compilation**, if the compiler cannot find the definition of a specific function in a file, it will assume that this function is defined in another file. As long as the source code is well-formed, the compiler will not complain. It is then to the linker to replace the references to undefined symbols with the correct addresses. Remaining undefined symbol are left untouched if not found and the **linking** process becomes incomplete. This will generate a new object file where remaining undefined symbol are left untouched.

It is then possible to use a demangling tool like `cppfilt` or `nm` to decode the C++ ABI names (steps 2-3 of Figure 3.2). Each demangled symbol will correspond to the internal C++ representation with a complete prototype of the incomplete function. By parsing this representation we can generate the CUDA/OpenCL kernel code with a corresponding host code to complete the link phase. Figure 3.2 describes this multi-stage process. To benefit from this paradigm, it is necessary to include a two stage compilation that cannot be deployed in a header only library. To implemented this process we use the CMake tool. It is also important to develop a readable intermediate representation that can be easily parsed when demangled.

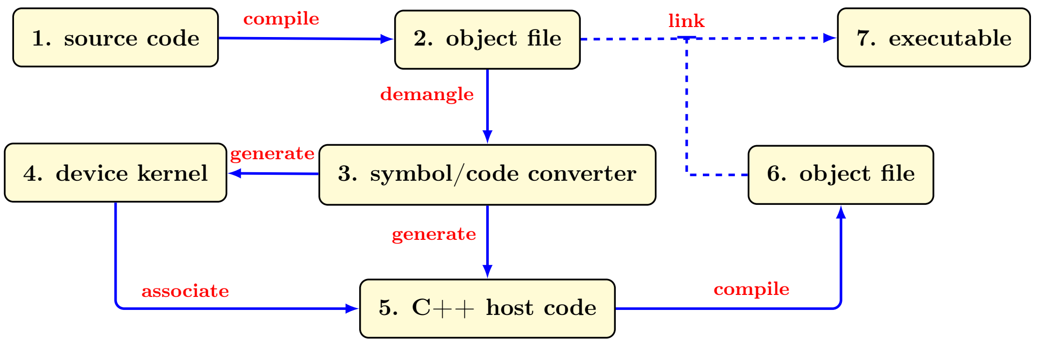


Figure 3.2: Two phase compilation for device code generation

3.2.3 Multi-stage programming tool for NT^2

A specific tool called *symbol/code converter* (Figure 3.2, Part 3) coupled with a serialization process was developed as an add-on to NT^2 to solve the issues described previously. The serialization process to solve the readability of the abstract representation is based on `Boost.Serialization` . This representation is similar to the *expression-template* AST based on BOOST.PROTO that is parsed by the C++ compiler. This enables us to get the semantic information of the container described in chapter 2 like its data locality, data type or matrix shape. It is however not possible to have run-time informations like the container size.

The role of *symbol/code converter* is two-fold. First, it must parse the demangled symbols obtained. This is done with `Boost.Spirit` [36], a C++ library to parse expressions and generate outputs based on them. It is implemented as a *DSEL* using *Expression templates* and *Meta-programming* techniques. With this library, we can generate outputs corresponding to the semantic information of the containers and operators. Secondly, it must generate the device and host code (Figure 3.2, Part 4-5) from the output generated by `Boost.Spirit`. In order to achieve this, we must deserialize the abstract representation in which the semantic informations are represented by a tag.

We will first describe the generation of the CUDA kernel. Each representation obtained from the AST corresponds either to an elementary expression of containers such as $a = b + c$ (or $a = b + c + d \ldots$), a sequence of operations if a fused operator (tie) is called or to an algorithmic skeleton (reduce, scan...). The parsing is separated between the left and right hand-side of the computation. The left hand-side will check if the expression is a terminal or an AST and generate the left part of the CUDA kernel expression. If it is a fused operator, it will generate a sequence of left operators in the kernel. The right hand-side will parse the operator and generate from the NT^2 CUDA back-end the corresponding operation with its parameters. Similarly, a fused operator will generate a sequence of right-hand side.

The generation of the host code consists in creating the source file corresponding to the functions with missing symbols. This amounts to adding the includes for the currents back-end, the CUDA kernel call and streaming data if necessary. As the device locality is available in the AST under the tag $nt2 :: device\_$ (see Section 2.3), we can stream the data to the GPU only if needed. Chaining non-fused operations that are on the host will obviously trigger back and forth data transfers. The overhead generated by such a process is diminished by the streaming and multi-stream process enabled for each operation.

We just described the key concepts of the *symbol/code converter* tool implemented in NT^2 for multi-stage programming. In the next section we will give concrete examples of generated kernels from *symbol/code converter* in NT^2 and how it supports hybrid computation.

3.3 Software implementation for NT^2

To describe the generation process, we first use the Triad kernel for element-wise operations which consists in doing a fused multiply-add (or fma : $a = b + c * d$).

The resulting code in NT^2 is :

```
// Define host table
table<float> A,B,C,D;

// Triad kernel
 A = B + C*D;
```

Listing 3.1: NT^2 Triad kernel

The code in Listing 3.1 corresponds to Part 1 of Figure 3.2. During the compilation phase, the operation in Listing 3.1 is replaced with a call to the CUDA transform skeleton. This skeleton will then call the device kernel that is not implemented yet resulting in the incomplete link sequence(Part 1-2, Listing 3.1). The equivalent code for transform is detailed in Listing 3.2. This transform will analyze the informations on the input matrices and the architecture to decide if it should proceed with the generation process.

```
table<float> A,B,C,D;

// Triad kernel replace with transform call
 transform(A, B + C*D);
```

Listing 3.2: NT^2 Triad transform

From there, once we compile our code and go through phase 2 of Figure 3.2 we obtain the following mangled code for our transform as described in Listing 3.3.

```
U _ZN3nt215external_kernelINS_3tag10transform_ENS1_5cuda_
IN5boost4simd3tag4avx_EEEE4callIKNS_9container4viewINS1_6
table_EfFNS_8settingsEvEEEKNSB_10expressionINS4_5proto7exp
rns_10basic_exprINS6_4fma_ENSJ_7argsns_5list3INSC_ISD_KfSF
EESQ_SQ_EELl3EEENS_6memory9containerISD_fFSE_NS_8of_si
ze_ILln1ELln1ELln1ELln1EEEEEEEEEEvRT_RT0_
```

Listing 3.3: NT^2 CUDA Triad mangled

We can then demangle the symbols resulting in the code described in Listing 3.4. This code corresponds to the Boost.Proto AST in NT^2 (see Figure 3.3)that we parse to generate the host and device code. The architectural tag of the machine is depicted in purple in Listing 3.4 and the fma computation node in red.

```
void nt2::external_kernel<nt2::tag::transform_, nt2::tag::cuda_<boost::simd::
    tag::avx_>>::call<nt2::container::table<float, nt2::settings ()>, nt2::
    container::expression<boost::proto::exprns_::basic_expr<boost::simd::tag
    ::fma_, boost::proto::argsns_::list3<nt2::container::view<nt2::tag::
    table_, float const, nt2::settings ()>, nt2::container::view<nt2::tag::
    table_, float const, nt2::settings ()>, nt2::container::view<nt2::tag::
    table_, float const, nt2::settings ()> >, 3l>, nt2::memory::container<nt2
    ::tag::table_, float, nt2::settings (nt2::of_size_<-1l, -1l, -1l, -1l>)>
    > const>(nt2::container::table<float, nt2::settings ()>&, nt2::container
    ::expression<boost::proto::exprns_::basic_expr<boost::simd::tag::fma_,
    boost::proto::argsns_::list3<nt2::container::view<nt2::tag::table_, float
     const, nt2::settings ()>, nt2::container::view<nt2::tag::table_, float
    const, nt2::settings ()>, nt2::container::view<nt2::tag::table_, float
    const, nt2::settings ()> >, 3l>, nt2::memory::container<nt2::tag::table_,
     float, nt2::settings (nt2::of_size_<-1l, -1l, -1l, -1l>)> > const&)
```

Listing 3.4: NT^2 CUDA Triad demangled

The function tag (nt2::tag::transform_) corresponds to each element-wise operations in NT^2 . Any function can be constructed using external kernel if necessary. As an example, algorithmic skeletons (scan, reduce, zip ...) will call external kernel when the CUDA back-end is activated. Then, we can generate a **thrust** code in a .cu file that will then be compiled with NVCC. Using this technique, we can overcome a weakness of the **thrust** library while benefiting from its optimized routines.

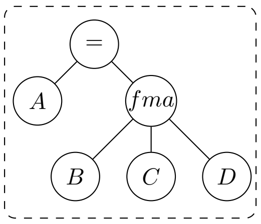


Figure 3.3: Triad kernel transform AST

The generated code for the .cu file is described in Listing 3.5 for Kepler cards. It corresponds to the AST representation with additional semantic information specific to the CUDA language. A function wrapper that calls the CUDA kernel (*fma4_wrapper*) is used to separate the compilation of the .cu file with NVCC from the corresponding C++ host code. The triad kernel directly calls the fma function from CUDA as the NT^2 AST recognizes every occurrence of an fma and replaces it by its function (Listing 3.4, see boost::simd::tag::fma_). This optimization in itself is not essential since it can be done by NVCC but is still interesting as it demonstrate the potential of code generation. As NT^2 already optimizes the AST by replacing patterns with corresponding functions, we can benefit from this analysis. We can then call the corresponding CUDA function if available or our own implementation for each pattern.

```
__global__ void fma4(float* t0, const float* t1, const float* t2, const float
    * t3)
{
  int idx = blockIdx.x * blockDim.x + threadIdx.x;
  t0[idx] = fmaf(t1[idx],t2[idx],t3[idx]);
}

void fma4_wrapper(float* t0, const float* t1 , const float* t2, const float*
    t3, dim3 Grid , dim3 Block, cudaStream_t & Str, int Shr)
{
  triad<<<Grid,Block,Shr,Str>>>(t0,t1,t2,t3);
}
```

Listing 3.5: NT^2 CUDA Triad kernel

The host code is partially described in Listing 3.6 and 3.7. Listing 3.6 corresponds to the initialization of parameters and data before doing the actual computation on the device. The blockSize and stream number is determined during the generation process depending on the number of parameters and architecture. The blockSize is usually generated by measuring the bandwidth of transfers from host to device in a range and choosing the most optimal one.

```
using boost::proto::child_c;
using boost::proto::value;

std::size_t size = numel(boost::proto::child_c<0>(a1));
std::size_t blockSize = std::min(std::size_t(40000),size);
std::size_t nStreams = std::min(std::size_t(2),size/blockSize);
std::size_t n = size / blockSize;
std::size_t leftover = size % blockSize ;
dim3 blockDim = std::min(std::size_t(1024),size);
dim3 dimGrid  = blockSize/size_t(1024);
cudaStream_t stream[nStreams];

// Allocating memory on the device
value(a0).specifics().allocate(blockSize,nStreams,size,true);
value(child_c<0>(a1)).specifics().allocate(blockSize,nStreams,size);
value(child_c<1>(a1)).specifics().allocate(blockSize,nStreams,size);
value(child_c<2>(a1)).specifics().allocate(blockSize,nStreams,size);

// checks redundancy between inputs and outputs
std::unordered_set<const float*> addr;
addr.insert(child_c<0>(a0).data());

for(std::size_t i=0; i < nStreams ; ++i)
{
  cudaStreamCreate(&stream[i]);
}
```

Listing 3.6: NT^2 CUDA Triad Host Code initialization

Using our tuning framework, we observed that to benefit best from the Kepler GPU bandwidth a block size of 40000 is necessary for single precision values. Double precision computations would lead to a lower block size generated. Similarly, the blockDim used is the highest value available for the architecture as we generate element-wise operations. Depending on the number of parameters the size may be lowered to limit the allocations. The allocation process includes pinned memory and device memory allocation. If containers were defined on the GPU with $nt2 :: device\_$, they would not appear in the allocation phase. As we have no information on the pointer for each container, we use an unordered set to limit redundancy in memory transfers.

Listing 3.7 describes the computation phase. It relies on block streaming with transfers to the GPU only if NT^2 tables are on the host. This streaming process is based on the overlap data transfers concept described by NVIDIA. It consists in creating multiple streams (the number depends on the architecture and problem intensity/size) and launching for each stream a transfer host to device, the CUDA kernel and the transfers device to host for a block. As the host memory has already been allocated, we must first transfer the data to pinned memory with cudaHostAlloc to benefit from GPU optimizations. Since the difference in bandwidth between pinned and page-able memory only increases with new architectures, this optimization can give a speedup even with a mono-stream program.

```
for(std::size_t i = 0; i < n ; ++i)
{
  std::size_t j = i % nStreams;
  value(a0).specifics().transfer_htd(a0, i, stream[j], j );
  value(child_c<0>(a1)).specifics().transfer_htd(child_c<0>(a1), i, stream[
      j], j ,addr);
  value(child_c<1>(a1)).specifics().transfer_htd(child_c<1>(a1), i, stream[
      j], j ,addr);
  value(child_c<2>(a1)).specifics().transfer_htd(child_c<2>(a1), i, stream[
      j], j ,addr);

  fma4_wrapper( value(a0).specifics().data(j), value(child_c<0>(a1)).
      specifics().data(j), value(
  child_c<1>(a1)).specifics().data(j), value(child_c<2>(a1)).specifics().
      data(j),dimGrid,blockDim,stream[j]);

  boost::proto::value(a0).specifics().transfer_dth(a0, i, stream[j], j );
}

if(leftover =0)
{
  ...
}
```

Listing 3.7: NT^2 CUDA Triad Host Code streaming

As stated in Section 2.3, computations on the device only occur if some conditions are met. As of now, these conditions are limited to the problem size and data locality but can be extended as the call to transform is automatic when NT^2 has defined that CUDA is available. Due to the hierarchical tag dispatching in NT^2 , a system with an Intel processor coupled with an NVIDIA card will have a tag similar to the following : $cuda\_ < openmp\_ < simd\_extension >>$. Therefore, if conditions for dispatch on the GPU are not met we will call the next level of transform (*i.e.*openmp). This enables us to use both the CPU and GPU in parallel depending on the problem which is a functionality rarely implemented in libraries.

To describe the generation process for algorithmic skeletons based on Thrust we will use a reduction. The resulting code in NT^2 is :

```
// Define host table
table<float ,device_> A;

// Define the type of reduction
functor<sum_> sum;

// Reduction using the sum of elements
 auto result = global(sum,A);
```

Listing 3.8: NT^2 Reduction kernel

Reductions are simpler to generate as they only require to call the correct underlying parallel skeleton and take care of memory transfers. The evolution of GPU architectures has an impact on the underlying implementation of the algorithm and its performance. Therefore, we prefer to rely on Thrust which is a standard NVIDIA library instead of implementing each parallel skeleton. It is simpler to use Thrust as a back-end in our MSP process rather than implementing complex algorithms such as fold,scan... and optimize them for each architecture. The demangled symbols corresponding to Listing 3.8 can be seen in Listing 3.9.

```
U void nt2::external_kernel<nt2::tag::global_ , nt2::tag::cuda_<boost::
    dispatch::tag::cpu_> >::call<nt2::container::table<short , nt2::device_>,
    boost::simd::tag::plus_ , short>(nt2::container::table<short , nt2::device_
    >&, boost::simd::tag::plus_&, short&)
```

Listing 3.9: NT^2 CUDA Reduction demangled

The demangled code is similar to the one described previously in Listing 3.4. The function tag is changed to $nt2 :: tag :: global\_$ which corresponds to the reduction skeleton. From there, it is easy to generate the corresponding host and kernel code. The kernel code described in Listing 3.10 is a wrapper for the thrust reduction function. There is no overhead compared to calling the function directly with Thrust.

```
#include <thrust/reduce.h>
#include <thrust/functional.h>
#include <thrust/device_ptr.h>

void thrust_reduce1de0a5eb(short* begin , short* end , short & result )
{
  thrust::device_ptr<short> b_ptr = thrust::device_pointer_cast(begin);
  thrust::device_ptr<short> e_ptr = thrust::device_pointer_cast(end);
  result = thrust::reduce(b_ptr,e_ptr, result ,thrust::plus<short>());
}
```

Listing 3.10: NT^2 Reduction kernel

The host code described in Listing 3.11. It defines the thrust function wrapper and calls it in the external kernel function.

```
#include <nt2/sdk/external_kernel/external_kernel.hpp>
#include <nt2/sdk/cuda/cuda.hpp>
#include <nt2/core/container/table/table.hpp>
#include <nt2/core/functions/global.hpp>
#include <cuda.h>

void thrust_reduce1de0a5eb(short* begin , short* end , short & result );
namespace nt2 {

template<> template <>
void nt2::external_kernel<nt2::tag::global_ , nt2::tag::cuda_<boost::dispatch
    ::tag::cpu_> >::call<nt2::container::table<short , nt2::device_>, boost::
    simd::tag::plus_ , short> (nt2::container::table<short , nt2::device_>& a0,
     boost::simd::tag::plus_&, short& a2)
{
  thrust_reduce1de0a5eb(a0.begin(), a0.end(), a2);
}
}
```

Listing 3.11: NT^2 Reduction kernel

This method can be extended simply as we have shown. Any functions implementing *external_kernel* will end up triggering our MSP process. Each function being differentiated by the function tag used. It is then possible to extend the *symbol/code converter* to support this new function. Each function can be mixed with one another and also used the MAGMA back-end we described in Section 2.4. Listing 3.12 is an example of a code doing a reduction and an elementwise operations that will go through our MSP process.

```
// Define host table
table<float ,device_> A,B;

// Define the type of reduction
functor<sum_> sum;

// Reduction using the sum of elements
 auto result = global(sum,A+B);
```

Listing 3.12: NT^2 Reduction and Elementwise kernel

The demangled object will contain both missing symbol for global and transform. The *symbol/code converter* will parse each symbol and generate the corresponding code.

3.4 Experiments

In this section, we will show that our generation process produces satisfactory performances in most situations. The benchmarks are realized with the following components :

- CPU : 2 x 6 cores Intel Xeon E5-2620 15MB L3, AVX

- GPU : Tesla K40m
  - Pageable host to device (HTD) : 3 GB/s
  - Pinned host to device : 9.8 GB/s
- Memory : 65 GB with a memcpy bandwidth of 5GB/s
- GCC 4.9, CUDA 7.5

3.4.1 Black & Scholes kernel

The Black & Scholes algorithm represents a mathematical model that gives a theoretical estimate of the price of European call and put options on a non-dividend-paying stock. It is a bandwidth bound algorithm for GPU if we take into account the memory transfers. Black & Scholes requires an important number of registers and high latency operations such as log or exp. The Black & scholes algorithm is a fusion of multiple transform skeletons requiring large entry sizes to obtain suitable performance.

The code is given in Listing 3.13 using the loop-fused technique described previously with the operator tie. The $nt2 :: device\_$ tag is specific for accelerator enabled architectures. However, if we use the tag while no accelerator is available we will fall back to the default architecture which is $nt2 :: host\_$.

```
table<T> blackscholes(table<T> const& S ,table<T> const& X
                      ,table<T> const& Ta , T const r
                      , T const v
                          )
{
  auto s = extent(Ta);
  table<T , device_ > d(s) , d1(s), d2(s);
  table<T> r;

  tie(d,d1,d2,r) = tie( sqrt(Ta)
                      , log(S/X)+(fma(sqr(v),0.5f,r)*Ta)/(v*d)
                      , fma(-v,d,d1)
                      , S*normcdf(d1)-X*exp(-r*Ta)*normcdf(d2)
                      );
  return r;
}
```

Listing 3.13: NT^2 black and scholes

This additional semantic information on memory locality can help to avoid useless memory transfers while staying simple enough for the user.

This will result in the .cu file in Listing 3.14 generated for floating point values. Since the AST does not contain the name of the parameters, the kernel generator has to give a different name to each one. Even with duplicated parameters, the memory transfers are only done once and a single pointer can be passed multiple times. Memory allocation on device and data transfers between host and device do pointer checking in the host code (see triad example) to insure no redundant work is done incurring also negligible overhead. The fnms function is due to an NT^2 AST transformation and corresponds to the fused negated multiply-subtract of three values.

```
 __global__ void bs ( float* t0 , float* t1 , float* t2 , float* t3, const
    float* t4, const float* t5, const float* t6, const float t7, const float
    * t8, const float t9, const float* t10, const float t11, const float*
    t12, const float* t13, const float* t14, const float t15, const float*
    t16, const float* t17, const float* t18, const float* t19)
{
 int i = blockIdx.x*blockDim.x+threadIdx.x;
 t0[i] = sqrtf(t4[i]);
 t1[i] = plus(logf(divides(t5[i],t6[i])),divides(multiplies(t7,t8[i]),
    multiplies(t9,t10[i])));
 t2[i] = fnms(t11,t12[i],t13[i]);
 t3[i] = fnms(multiplies(t14[i],expf(multiplies(t15,t16[i]))),fastnormcdf(t17
    [i]),multiplies(t18[i],fastnormcdf(t19[i])));
}
```

Listing 3.14: NT^2 black and scholes fused CUDA kernel

The Black & Scholes algorithm involves high latency and high register count operations. This will result in sub-optimal performance on SIMD for the CPU due to spilled registers while a Kepler GPU will not have any such problem. As seen in Figure 3.4, the execution time of the kernel on the GPU is negligible (2-3 ms) compared to the overall time of the optimized version with SIMD and OPENMP in NT^2 .

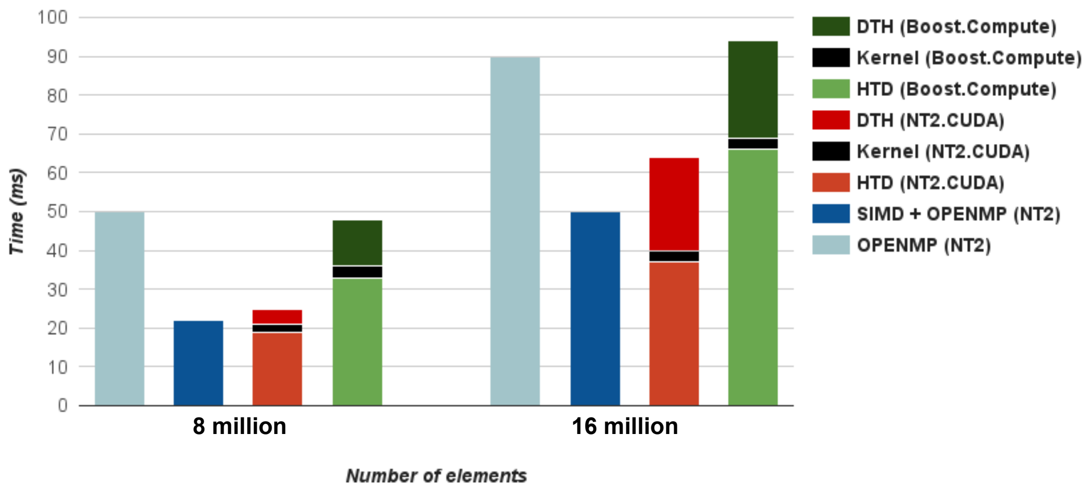


Figure 3.4: Black and Scholes Performance Comparison (in ms)

This is due to the highly parallel nature of the GPU. As its computational power is much higher, simple elementwise operations will naturally fit the programming model while CPUs must be optimized. Most of the time is spent transferring the data between host and device memory which can be avoided with the right semantic information available (*nt*2 :: *device*_). In the scenario presented in Listing 3.13, which represents the worse case, we still have better performance than the `Boost.Compute` version on which most C++ libraries are based as we use block streaming with pinned memory reaching a near-optimal throughput (average of 9.7 GB/s) for device transfers. As the bandwidth of a memcpy on the CPU (5 GB/s) is faster than page-able transfers (3GB/s) even without any overlap between transfers and computation we still increase the performance. This optimization can be disabled depending on the CUDA architecture. We have also compared with an optimized CUDA implementation and the performance was identical. This shows that our approach to generate element-wise operations does not incur any overhead on the kernel computation time.

As computations on the GPU are often performed in great number, if the user allocates the data on the GPU he will pay no extra transfer cost for the rest of the computations and in this case the CUDA kernel is up to twelve times faster than the hand-optimized version for CPU. Also, using the flag *nt*2 :: *pinned*_ if tables must stay on CPU allows us to gain up to a $x3$ factor in transfers time. The optimized Black & Scholes version we implemented in CUDA reached the same performance in kernel and transfers time when using the same type of memory allocation for data.

Size	CPU	GPU	speedup
8000000	120	8.2	14.6
16000000	245	16.1	15.2

Figure 3.5: Performance comparison on Tegra X1 with NT^2 (time in ms)

On the Tegra X1, the GPU is much more efficient than the CPU. This is caused by the large difference in performance between the Maxwell GPU and the 4-core ARM CPU. The ARM cortex A57 has around 64 Gflops while the GPU achieves 500 Gflops. The code in Listing 3.15 corresponds to the one executed on the Tegra for the GPU. As the no-copy model from NVIDIA for IGPs require data to be pinned, we have to allocate CPU tables with the *pinned*_ flag.

```
table<T> blackscholes( table<T,pinned_> const& S ,table<T,pinned_> const& X
                      ,table<T,pinned_> const& Ta , T const r
                      , T const v
                          )
{
  auto s = extent(Ta);
  table<T , device_ > d(s) , d1(s), d2(s);
  table<T,pinned_> r;

  tie(d,d1,d2,r) = tie( sqrt(Ta)
                      , log(S/X)+(fma(sqr(v),0.5f,r)*Ta)/(v*d)
                      , fma(-v,d,d1)
                      , S*normcdf(d1)-X*exp(-r*Ta)*normcdf(d2)
                      );
  return r;
}
```

Listing 3.15: NT^2 Black and Scholes for Tegra

The important performance difference between the K40m and the Tegra is due to the on device bandwidth difference between the two cards. On our system, the K40m has an on device bandwidth of 200 GB/s while the Tegra has around 30GB/s.

3.4.2 Linsolve kernel

Linsolve is the MATLAB solver for linear systems. As NT^2 has its own implementation of linsolve with a LAPACK/MAGMA back-end, we can combine it with the code generation process. As an example, we consider the solution of a dense linear system using the LU factorization and apply one step of iterative refinement:

1. Compute $r = b - A\hat{x}$.
2. Solve $Ad = r$.
3. Update $y = \hat{x} + d$.

The equivalent NT^2 code is the following :

```
table<T, device_> A,b,x,e;
table<T, settings(device_, upper_triangular_)> u ;
table<T, settings(device_, lower_triangular_)> l ;

tie(l,u) = lu(A)
x = linsolve(l,b);        // lower triangular solve
x = linsolve(u,x);        // upper triangular solve

// One-step refinement
e = b - nt2::mtimes(A,x);
e = nt2::linsolve(l,e);
e = nt2::linsolve(u,e);

x = x + e;
```

Listing 3.16: NT^2 LU linear solve with iterative refinement

If executed on the CPU, the code in Listing 3.16 will call the LAPACK routines. The semantic information upper_triangular_ allows `linsolve` to call the triangular solver instead of doing the classic linear solve. If executed on the GPU, the same optimizations will be applied and the iterative refinement process will trigger calls to transform for both element-wise operations.

The performance results in Figure 3.6 attest that the performance obtained with our model is relevant. The GPU version with *MSP* calls magma kernels using the CUBLAS DGEMM routine without doing any transfer and reaches near peak performance of a K40m GPU which corresponds to 1.40 Tflop/s. The version that does not use *MSP* is slower as transfers are done during the iterative refinement step. The CPU version quickly reaches the peak performance of both CPU which is 210 Gflop/s. As we can see, there is no performance loss while call the LAPACK/-MAGMA back-ends and if device pointers are passed to our code generator, there will be no memory transfers. Similar performance would also be reached using the other factorizations available in NT^2 .

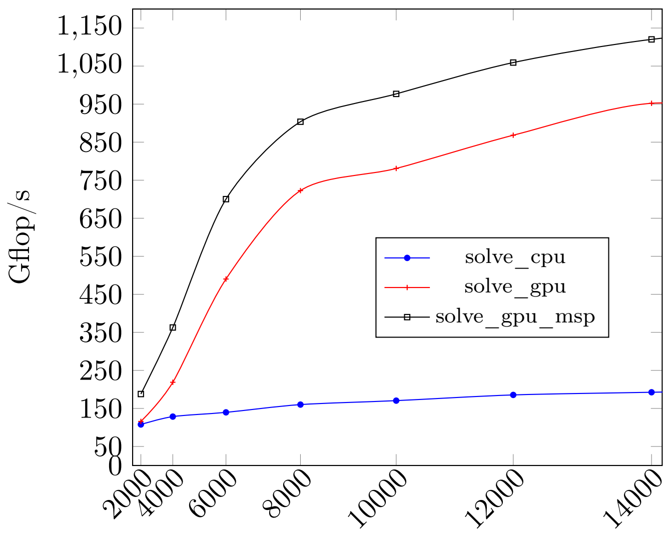


Figure 3.6: Performance comparison of NT^2 linear solve (in Gflop/s)

3.5 Conclusion

The development of tools for simplifying accelerator programming has been an active topic since accelerators have become a mainstream element of high-performance computing systems. In this chapter, we proposed an extension of a high-level, data-parallel scientific computing library to specifically handle GPU accelerators. Our main objectives were to keep a similar level of expressiveness in the client code while supporting different use cases of accelerator programming.

To reach this goal, we have implemented a **multi-stage** system in which the initial C++ code is used to automatically generate the equivalent CUDA kernel by reusing our internal representation of this code. This representation is based on C++ *Expression Templates* and the **Algorithmic Skeleton** to identify and classify expressions based on the kind of loop nest that is required. Finally, we have illustrated on a selection of examples that the performance obtained is close to the hardware capability and exhibits benefits compared to other solutions.

Work is still on-going on this system, including support for more specific functions on the latest GPUs. Implementing a more thorough cost model to ensure better scheduling of computation between CPU and GPU is also being studied. The natural evolution of this work is to extend our approach to the Intel Xeon Phi coprocessor, the runtime-based CUDA compiler recently released, or OpenCL devices. An interesting track of research can be derived from the support of OpenCL by targeting OpenCL enabled FPGAs, as we could bridge the gap between high-level C++ and hardware design.

Chapter 4

Application to batched computations and tensor contractions

Contents

In recent years, tensors have started to take an important place in HPC applications. This is due to tensors' frequent use in physics and engineering, where tensors provide a mathematical tool for brief and comprehensive formulations and solutions of problems in areas such as elasticity, fluid mechanics, multi-physics, quantum chemistry, general relativity, and many others [81]. Tensors are multi-dimensional arrays that can be used to describe physical properties featuring multilinear relations. Well known mathematical objects like scalars, vectors, and matrices can be generalized to tensors that are of order zero, one, and two, respectively. Also, tensor transformations like flattening of a tensor to matrices or reshaping of matrices into tensors, can be used to link tensor computations to the developments in high-performance numerical linear algebra. Therefore, similar to many applications, tensor computations can also significantly benefit from representing their computations in terms of BLAS, as well as from other dense LA algorithms and techniques for multicore and GPU architectures.

This work is the result of a collaboration with Azzam Haidar, Stanimire Tomov, Ahmad Abdelfattah and Jack Dongarra from the Innovative Computing Laboratory in Tennessee. Parts of the work presented in this chapter have been accepted at ICCS 2016 [2] and Europar 2016 [92]. As this project is meant to be part of the MAGMA library, we decided not to use NT^2 to limit external dependencies. However, the techniques presented here stems from the same principles and have been integrated in NT^2 . The contributions of this chapter are the following :

- A method to take advantage of modern C++ design for hybrid computations and interface development
- A strategy using performance analysis based on hardware features and counters to implement efficient algorithms no specific architecture
- An implementation of a small batched matrix-matrix product that can perform better than state of the art libraries

We start by giving in Section 4.1 a small overview of some Tensor libraries for numerical algebra. Then, in Section 4.2 we explain the challenges we face while developing a tensor interface and how to exploit modern C++ (post C++11). The next Section 4.4 explains how we can optimize batched computations for small matrix-matrix products which are often found in tensor computations. We finish this chapter by giving concluding remarks in Section 4.5.

4.1 Tensors in numerical libraries

The use of tensors in physics and engineering applications has motivated an extensive research in both algorithms for tensor algebra and computational aspects for tensor-based simulations. See the survey [81], and the references cited there, for an overview of the linear algebra aspects of tensors research, including higher-order tensor decompositions, their applications, and available software. A general overview with current and future direction in tensor research is presented in the *Future directions in tensor-based computation and modeling* workshop report [1].

There has been an important number of software packages developed for tensors. Some are designed to be used through numerical computing mathematical environments like the Tensor Toolbox[1] for Matlab, GRTensor II[2] for Maple, or Ricci[3] for Mathematica. These packages tend to focus on providing proper tensor computations and do not provide highly optimized operations or target accelerators. Others

[1]http://www.sandia.gov/∼tgkolda/TensorToolbox/
[2]http://grtensor.phy.queensu.ca/
[3]http://www.math.washington.edu/∼lee/Ricci/

such as Tensor Transpositions Compiler [121] (TTC), an open-source parallel compiler for multidimensional tensor transpositions, focus on specific optimizations to generate efficient code. Tensor packages for specific applications such as quantum chemical computations [136] present ongoing work on using accelerators to accelerate tensor contractions on GPUs. The approach uses code generation techniques and is incorporated in the NW Chem computational chemistry suite. More recent work [97] also uses code generation techniques and autotuning (with a system called Baracuda, based on CUDA-CHiLL and a high-level module Optimizing Compiler with Tensor OPeration Intelligence (OCTOPI)) to report significant acceleration compared to NW Chem on particular tensor contractions.

While we use code generation and auto-tuning, our approach also relies on context knowledge, and in particular that tensor reshaping techniques can often transform tensor contractions from many applications into many GEMMs, right from machine learning. This transformation can not easily be detect automatically, and moreover, even if the transformation is somehow indicated, it is well known that general GEMM optimizations using only compiler techniques can not match in performance best-tailored solutions.

For distributed memory system that are CPU only, [120] executes contractions via GEMM on a properly ordered and structured tensor. As in the other approaches in quantum chemistry, large tensor contractions are targeted. In contrast, we target many but small contractions, that are often very small size (see next section), resulting in large-scale computations. Tensor reshuffle operations are done to cast the contractions to GEMMs, when possible, and a batched approach with custom-built BLAS kernels is used to efficiently execute them. Additional closely related efforts include [91, 117] and [122].

4.2 Applications and challenges

The evolution and diversification of computer hardwares and programming languages is transforming the balance of computer systems. This evolution is producing changes at every level from programming efficiently the hardware to developing high level programming tools. From the point of view of numerical libraries, and the important number of applications that depend on them, three challenges stand out:

- the need to exploit unprecedented amounts of parallelism;
- the need to maximize the use of data locality and vectorized operations; and
- the need to cope with component heterogeneity.

Below, we highlight our main contributions related to the algorithm's design and optimization strategies aimed at addressing these challenges on multicore CPU and

GPUs.

Exploit Parallelism and Vector Instructions:

Clock frequencies are expected to stay constant, or even decrease to conserve power; consequently, as we already see, the primary method of increasing computational capability of a chip will be to dramatically increase the number of processing units (cores), which in turn will require an increase of orders of magnitude in the amount of concurrency that routines must be able to utilize as well as increasing the computational capabilities of the floating point units by extending it to the classical Streaming SIMD Extensions set (SSE-1, to SSE-4) in the earlier 2000, and recently to Advanced Vector Extensions (AVX, AVX-2, AVX-512). We developed specific optimization techniques that demonstrate how to use the many cores (currently multi-socket $10 - 22$ cores for the Haswell CPU or a K40 GPU using MAGMA to get optimal performance. The techniques and kernels developed in this work are generic and can be used elsewhere.

Hierarchical Communication Techniques that Maximizes the use of Data Locality:

Recent reports (e.g., [56]) have made it clear that time per flop, memory bandwidth, and communication latency are all improving, but at exponentially different rates. So computation on very small matrices, that can be considered as computation-bound on old processors, is, –today and in the future– communication-bound and depends from the communication between levels of the memory hierarchy. We demonstrate that, performance is indeed harder to get on new manycore architectures unless hierarchical communications and optimized memory management are considered in the design. We show that, only after we developed multilevel memory design, our implementations reach optimal performance.

Performance Analysis and Auto-tuning:

We demonstrate the theoretical maximal performance bounds that could be reached for computation on very small matrices. We studied various instructions and performance counters, as well as proposed a template design with different tunable parameters in order to evaluate the effectiveness of our implementation and optimize it to reach the theoretical limit.

4.3 Container concept

The design of our code is done using new features of C++ for better re-usability and adaptability of the code. By using advanced template techniques we can create high-level interfaces without adding any cost even for small matrix-matrix products. To do so, we have designed a batched structure which will contain a C++ vector for the data and static dimensions. By using the C++ constexpr keyword and integral constants we can make a generic batched code that will dispatch at compile time the correct version depending on the size of matrices. We use this environment for each code sequence we generate.

To develop high-quality HPC software for tensor contractions, we impose the following three main requirements on our interface design:

Convenience of use: Interfaces of HPC libraries are extremely important for the libraries' adoption by the community. Interfaces must provide convenience and practicality of use, including ease of interoperability between libraries. Ideally, a tensor data type standard must be defined. The standard for a dense matrix is a pointer, sizes, leading dimension, and assumption for column-major data layout, e.g., as in BLAS and LAPACK. For tensors, motivated by reviewing interfaces in available libraries and the success of BLAS, we propose to represent a tensor by its dimensions and a data layout abstraction. The abstraction is to provide a choice of predefined layouts, and ways to switch it, or to easily add user-specified layouts, without changing the underlying algorithms. In general, a specific layout provides the formula of mapping the multi-dimensional tensor to the memory, e.g., a second order tensor can be stored as a column-major matrix A with leading dimension lda, in which case the abstraction maps $A_{i,j}$ to $A[i + j * lda]$.

Readability: Numerical software must be understandable, which is needed for ease of maintenance, as well as code optimizations, and testing. While we can easily implement any interface, e.g., even expressing the interface and tensor APIs in a DSEL if needed (plus code generation techniques to translate the DSEL to a standard language; such as the Einstein notation used in tensor computations), we determined that it is better to provide implementations relying on a standard language and the code generation features provided within the language. The C++14 standard for example is expressive enough to allow us to implement readable and easy to use interfaces.

Performance: While we expect to obtain high performance mostly through algorithmic design and auto-tuning (see Section 4.4), we do not want to compromise on optimization opportunities like removing parameters checking and loop unrolling to eliminate jumps and loop count decrements. These optimizations are essential especially for the small computations that we target. Therefore, in our design we consider the use of compiler features related to code generation (e.g., templates, etc.), as further discussed below.

4.3.1 C++11/14 features

The development of programming languages and their use has dramatically changed in recent years leading to continuous evolution. C++ is an example of such a programming language. The standardization committee has decided to make a new standard every 3 years, with the next release being the C++17 standard. Compatibility with the C language has also starting to break with the introduction of the auto keyword. The cause of these changes is the need for higher level language that provide better idioms for generic and generative programming. Here we will discuss new features of the C++11/14 standard that we use to develop a high level tensor interface.

The first feature of the C+11 language that we will discuss is auto [76]. Consider the following declaration in Listing 4.1 :

```
// x is the type of 7 : int
auto x = 7;
```

Listing 4.1: C++ auto

Here x will have the type int because it is the type of its initializer. In general, we can write

```
// x is the type of expression
auto x = expression;
```

Listing 4.2: C++ generic auto

and x will be of the type from the value expression in Listing 4.2. For any variable, auto specifies that the type of the variable that is being declared will be automatically deduced from its initializer. This allows to write high level complex code without having the burden of complex types that can appear. We can apply the auto keyword on several features of the C++ language. As an example, in Listing 4.3 we can iterate with a range for statement on a vector of a generic type without defining the type of the value x.

```
template<typename T>
void f(vector<T>& v)
{
  for (auto& x : v) ++x;
}
```

Listing 4.3: C++ generic for auto

Another important feature of the C++11 standard is the constexpr [43] keyword. constexpr provides a mechanism that can guarantee that an initialization is done at compile time. It also allows constant expressions involving user-defined types.

In Listing 4.4, the fibonnaci function is guaranteed to be executed at compile time if the value passed x is available at compile time.

```
constexpr long long fibonacci(const int x)
{
    return x <= 1 ? 1 : fibonacci(x - 1) + fibonacci(x - 2);
}
```

Listing 4.4: C++ constexpr

An important addition to the C++ standard for extensible design is variadic templates [60]. In computer programming, variadic templates are defined as templates that take a variable number of arguments. Prior to C++11, templates could only take a fixed number of arguments which were specified at the declaration of the template.

```
// The [...] defines a variadic template
template<typename... Values> class tuple;
```

Listing 4.5: C++ Variadic template

Listing 4.5 is an example of a variadic template class that takes a variable number of arguments. Listing 4.6 allows for a more in-depths description of variadic templates.

```
template<typename T>
T add(T v) {
  return v;
}

template<typename T, typename... Args>
T add(T first , Args... args) {
  return first + add(args...);
}
```

Listing 4.6: C++ Variadic template add

The function add will accept any number of arguments and will compile as long as the `operator+` can be applied between the arguments. This is type-safe as the verification is done at compile time. The resolution will be done using C++'s template and overloading resolution rules. Putting the [...] before the parameter name, `typename... Args` is called a template parameter pack. Using the [...] after a definition `Args... args` is called function parameter pack. This is how the function add is defined : the first argument is peeled off from the template parameter pack into type T. With each subsequent call, the parameter pack gets shorter by one parameter until the last occurrence with only one parameter in the pack is attained.

```
// sum is of type int
auto sum = add(1,2,3,6);
// sum1 is of type string
std::string x = "string", y = "add", z = "function";
auto sum1 = add(x,y,z);
```

Listing 4.7: C++ Variadic template add call

Examples of how the function add can be called are shown in Listing 4.7.

```
// Apply a fonction on a list of arguments
template<class F, class...Ts> F for_each_args(F f, Ts&&...a)
{
  return (void)std::initializer_list<int>{((void)std::ref(f)(std::forward<Ts
      >(a)),0)...}, f;
}
```

Listing 4.8: C++ Variadic template function apply

Varadic templates are a very powerful tool that has some very useful tricks. For example, in Listing 4.8, we have defined a function that take another function and a list of arguments and applies the function on each argument.

Using `constexpr` and the features described previously also allow for integral constants. Integral constants are part of the C++ standard and wrap a static constant of a specific type in a class. We show in Listing 4.9 a possible definition for them[4].

```
template<class T, T v>
struct integral_constant {
    static constexpr T value = v;
    typedef T value_type;
    typedef integral_constant type;
    constexpr operator value_type() const noexcept { return value; }
    constexpr value_type operator()() const noexcept { return value; }
};
```

Listing 4.9: Possible integral constant definition

These types of expression can be used as non-type template arguments, array sizes, and in other contexts that require constant expressions. They are often used in conjunction with meta-programming techniques. The most common library pre-C++11 that was based around integral constant is Boost.MPL[5]. Integral constants can be declared in the following way : $integral\_constant < int, 25 > ic\_25$, with ic_25 being the type for the value 25. We can the use integral constants in conjunction with user literals. User literals allow integer, floating-point, character, and string literals to produce objects of a wanted type by defining a suffix. Listing 4.10 is the code to declare a user defined integral constant literal that we can then pass as function parameter. The keyword `using` is also a new feature of the C++11 standard that replaces `typedef` and allow for template parameters.

```
template<unsigned long N>
using int_ = std::integral_constant<unsigned long,N>;

constexpr int to_int(char c) { return static_cast<int>(c) - 48; }

template <std::size_t N, typename I>
constexpr long long parse(const char (&arr)[N], I const&, int )
{
    long long number = 0, base = 1;
    for (std::size_t i = 0; i < N; ++i) {
        number += to_int(arr[N - 1 - i]) * base;
        base *= 10;
        }
    return number;
}

template <char ...c>
constexpr auto operator"" _c()
{ return int_<parse<sizeof...(c)>({c...},int_<sizeof...(c)>{},1)>{}; }
```

Listing 4.10: C++ User defined literals

[4]http://en.cppreference.com/w/cpp/types/integral_constant
[5]http://www.boost.org/doc/libs/1_61_0/libs/mpl/doc/index.html

User defined literals give us the means to simply dispatch on these values or use them as static function parameters instead of using templates. This allows for a more natural interface for the user and a simpler code generation process. Using Listing 4.10, we can now just use 25_c to represent the value 25. It will be automatically converted into an integral constant. Since C++11, operator"" is used for user-defined literals allowing for specific implementations. The easiest way to parse such a literal taking a `variadic template` char argument is by unpacking it, hence the const char array as a parameter to the parse function.

The last important feature that we will talk about is `lambda` functions [75]. A `lambda` expression is a mechanism for specifying a function object. The primary use is to specify a simple action to be performed by some function. For example:

```
std::vector<int> in,in1,out;
std::transform( in.begin() ,in.end(), in1.begin(), out.begin()
  , [](int a, int b)->int
{
    return a + b;
});
```

Listing 4.11: C++ lambda using std transform

$std :: transform$ will iterate on each value of in and in1 and apply the lambda function passed on each element. The resulting operation will be written in the out vector. The lambda starts with the [ ] identifier, called the capture specification. It serves to specify how local variable are passed (by value : = , by reference : &). Lambdas are similar to functors: the variables captured by reference are like private pointers, and the ones captured by value are like private variables.

Lambdas are lightweight, nameless functions that can be defined where they are used. Lambdas are now a part of the C++ standard template library and current and future algorithms such as <thread> or <future>, allow you to specify function arguments as lambdas beside C-like functions and functors. Parallel skeletons implemented in the C++ standard library and Thrust have added support lambda functions. Modern C++ libraries also support lambdas as seen in Section 3.1. An example for thrust is shown in Listing 4.12.

```
thrust::host_vector<int> in,in1,out;
thrust::transform( in.begin(), in.end(),in1.begin(),out.end
  , [](int a, int b)->int
{
    return a + b;
});
```

Listing 4.12: C++ lambda using thrust transform

Since CUDA 7.5, NVIDIA has also added experimental support for device `lambda` using the _device_ flag. A proposal to use `ranges`[6] for the C++ standard library instead of iterators is also in study.

[6]https://ericniebler.github.io/std/wg21/D4128.html

4.3.2 Developing a tensor interface

To develop a high level tensor interface, we use the C++ features detailed previously while keeping in mind the challenges described in Section 4.2. Related to performance, a lost optimization opportunity is if static checking and compile time information is not provided as part of the scientific libraries used. As an example, LAPACK routines start by checking entry parameters dynamically, which results in extra time for small size computations. The type of algorithms that we intend to use, also require specific tuning whether it be on CPU or GPU with MAGMA batched [67] as performance will greatly vary depending on numerous parameters.

To avoid these performance drawbacks, and also benefit from optimization techniques like compiler loop unrolling for static dimension problems, we use features of the new C++14 standard. In particular, the `constexpr` specifier enables to evaluate the value of a function or variable at compile time. We use this feature with integral constants and template specialization to design our tensor dimensions layout:

```
// Template specialization
constexpr auto layout = of_size<5,3>();
// Using Integral constant
constexpr auto layout1 = of_size(5_c,3_c);
// Using dynamic dimensions
constexpr auto layout2 = of_size(5,3);
// Using static and dynamic dimensions
constexpr auto layout3 = of_size(5_c,3);
// Access Dimensions at compile time
constexpr auto dim1 = layout(1);
// Will not cause a compile time error
static_assert( x4(1) == 5 , "first dimension is static");
```

Listing 4.13: Dimensions for Tensors

As seen in Listing 4.13, we propose several ways to specify dimensions using the function *of_size* which returns a layout containing the static or dynamic dimension. Each operator inside the layout uses the `constexpr` keyword which enables us to return sizes at compile time if possible. Using `constexpr` will give the compiler more optimization opportunities such as loop unrolling.

The layout is class based on `variadic templates` to have a parameterizable number of dimensions. At the initialization of the layout, parameters are automatically unpacked, giving the value for the nth dimension if it is available at compile time. Dynamically sized dimensions will get an initial value of -1 to say they have not been initialized. To enable this optimization, the number of dimensions in a layout is decided at compile time leaving only the size of each dimension to be changeable. In Listing 4.14, we give an example of the definition for a simple layout to better view the key concepts behind it.

```
template<typename N0, typename... N> struct layout
{
  template<typename M0, typename... M> using rebind = layout<M0,M...>;

    static constexpr size_t nbDims = sizeof...(N)+1;
    int dim[nbDims];

    constexpr layout() : dim{ is_static_value<N0>::value ? static_value<N0>::
        value : 0
              , (is_static_value<N>::value ? static_value<N>::value : -1)...
              }
    {}

    constexpr layout(N0 n0, N... n) : dim{ n0, n... } {}

    constexpr std::size_t operator()(int n) const noexcept
    {
      return dim[n-1];
    }
};
```

Listing 4.14: Simple layout without strides

We can then freely extract the dimensions (Listing 4.13) and use them to specify our CPU and GPU kernels at compile time. Our tensor model is based on our layout, the data type of the tensor and its locality. The memory buffer is based on vector from the Standard Template Library for CPU and Thrust for GPU. To generate a tensor, we need to pass a data type and locality as template parameter and the size to the constructor (Listing 4.15).

```
// Create a rank 2 tensor of size 5,3 on GPU
constexpr tensor<float,gpu_> d_ts(of_size<5,3>());
// Create a rank 2 tensor of size 5,3 on CPU
constexpr tensor<float> ts(of_size<5,3>());
// Use thrust to fill d_ts with 9
thrust::fill(d_ts.begin() , d_ts.end() , 9);
// Copy d_ts from GPU to ts on CPU
copy( d_ts , ts ) ;
```

Listing 4.15: Create Tensor and copy

Transfers between CPU and GPU tensors can be expressed through the function copy (Listing 4.15). This function will use the stream 0 by default but a stream can be passed for asynchronous copy. Locality definitions are simple classes that contain the semantic information necessary. For example, *gpu_* will be a class definition that will inherit from the locality class itself. It is then possible to iterate on each parameter of the layout looking for a locality class. Meta-functions are available in the C++ standard library (utility package) to do type checking. For example, *std* :: *is_base_of* verifies if a class inherits from another one. If we cannot find a corresponding locality in the class definition, we can then fall-back to the default locality we have decided.

We have designed two models for batched computing (Listing 4.16). The first one is based on allocating a single memory block for all tensors to improve data locality while the other is a group of same size tensors.

```
// Create a batch that will contain 15 tensors of size 5,3,6
constexpr auto batch<float , gpu_> b = make_batch(of_size(5_c,3_c,6_c) , 15);
// Accessing a tensor from the batch returns a view on it
constexpr auto view_b = b(0);
// Create a grouping of tensors of same size tensors
constexpr auto group<float ,gpu_> g(of_size(5_c,3_c));
// Add a tensor to the group
constexpr auto tensor<float ,gpu_> d_ts( of_size(5_c,3_c) );
g.push_back(d_ts);
```

Listing 4.16: Batched tensors

Once we have defined these functions we can call the kernel to compute a batched DGEMM on tensors of dimension 2.

```
constexpr auto batch<float , gpu_> b =  make_batch(of_size(5_c,3_c) , 15);
constexpr auto batch<float , gpu_> b1 = make_batch(of_size(5_c,3_c) , 15);
// Product of two tensor batched of dimension 2 for matrix product using C++
    operator
constexpr auto c = b * b1;
// Product using the batch dgemm function that can be specialized depending
    on parameters
constexpr auto c = batch_gemm(b , b1 );
```

Listing 4.17: Tensor Operations

The DGEMM batched function can be dispatched depending on the locality, calling either our implementation for small matrix product with static sizes on CPU or the MAGMA/CUBLAS implementation if the tensor is defined on the GPU.

4.4 Optimizing small matrix-matrix products

4.4.1 Performance measures

To evaluate the efficiency of our algorithms we derive theoretical bounds for the maximum achievable performance $P_{max} = F/T_{min}$, where F is the number of operations needed by the computation and T_{min} is the fastest time to solution. For simplicity, consider $C = \alpha AB + \beta C$ on square matrices of size n. We have $F \approx 2n^3$ and $T_{min} = min_T(T_{Read(A,B,C)} + T_{Compute(C)} + T_{Write(C)})$. Note that we have to read/write $4n^2$ elements, or $32n^2$ Bytes for double precision (DP) calculations. Thus, if the maximum achievable bandwidth is B (in Bytes/second), and we assume $T_{Compute(C)} \to 0$ for very small computation, then $T_{min} = T_{Read(A,B,C)} + T_{Write(C)} = 4n^2/B$ in DP. Note that this time is theoretically achievable if the computation totally overlaps the data transfer and does not disrupt the maximum rate B of read/write to the GPU memory. Thus,

$$P_{max} = \frac{2n^3 B}{32n^2} = \frac{nB}{16} \text{ in DP.}$$

The achievable bandwidth can be obtained by benchmarks. For our measures, we used the STREAM benchmark [95] and the Intel memory latency checker 3.0 tool for CPU with gcc 5.2, and the NVIDIA's `bandwidthTest` for GPU. Our tests show that the practical CPU bandwidth we are able to achieve using different benchmarks is about 44 GB/s per socket. On the K40 GPU with ECC on the peak is 180 GB/s, so in that case P_{max} is 2.75 n GFlop/s per socket for the CPU and 11.25 n GFlop/s for the K40 GPU. Thus, when $n = 16$ for example, we expect a theoretical maximum performance of 180 GFlop/s in DP on the K40 GPU.

The implementation of a matrix-matrix products kernel for very small matrices for CPUs requires specific design and optimizations. As we can store three double precision matrices of size up to 32×32 in the L1 cache of an Intel Xeon E5-2650 v3 processor, one can expect that any implementation will not suffer from data cache misses. This can be seen on Figure 4.6 where the performance of an ijk implementation, which is not cache-aware and cannot be vectorized, is pretty close to the ikj one. For smaller sizes, the ijk implementation is even more efficient than the ikj one, as it optimizes the number of stores (Figure 4.2). To obtain a near optimal performance, we conduct an extensive study over the performance counters using the PAPI [135] tools. Our analysis concludes that in order to achieve an efficient execution for such computation, we need to maximize the occupancy and minimize the data traffic while respecting the underlying hierarchical memory design while maximizing the use of SIMD registers. Unfortunately, today's compilers cannot introduce highly sophisticated cache/register/SIMD based loop transformations and, consequently, this kind of optimization effort should be studied and implemented by the developer [88]. This includes techniques like reordering the data so that it can be easily vectorized, reducing the number of instructions so that the processor spends less time in decoding them, prefetching the data that will be reused in registers, and using an optimal blocking strategy.

4.4.2 Data Access Optimizations and Loop Transformation Techniques

In our design, we propose to order the iterations of the nested loops in such a way that we increase locality and expose more parallelism for vectorization. We can then use AVX2 SIMD instructions to optimize to the fullest our code. The matrix-matrix product is an example of perfectly nested loops which means that all the assignment statements are in the innermost loop. Hence, loop unrolling, loop peeling, and loop interchange can be useful techniques for such algorithm [5, 15]. These transformations improve the locality and help to reduce the stride of an array based computation. In our approach, we propose to unroll the two innermost loops so that the accesses to matrix B are independent from the loop order, which also allows us to reorder the computations for continuous access and improved vectorization. This technique enables us to prefetch and hold some of the data of B into the SIMD registers. Here, we manage to take advantage from the knowledge of the algorithm, and based on the principle of locality of references [68], to optimize both the temporal and spatial data locality.

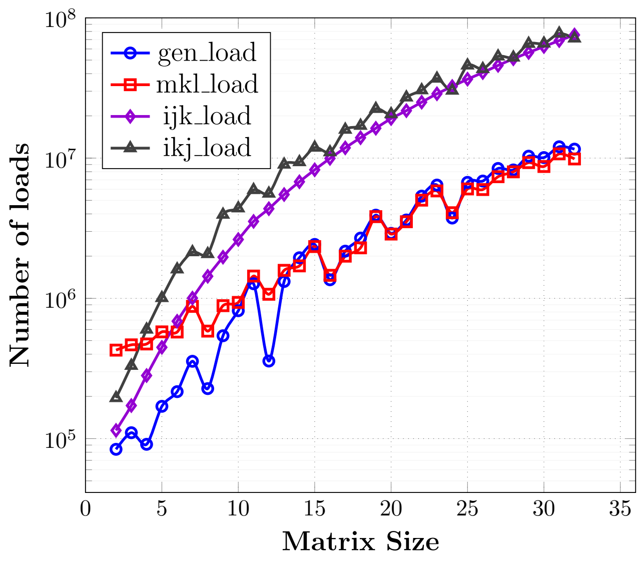


Figure 4.1: # of load instructions

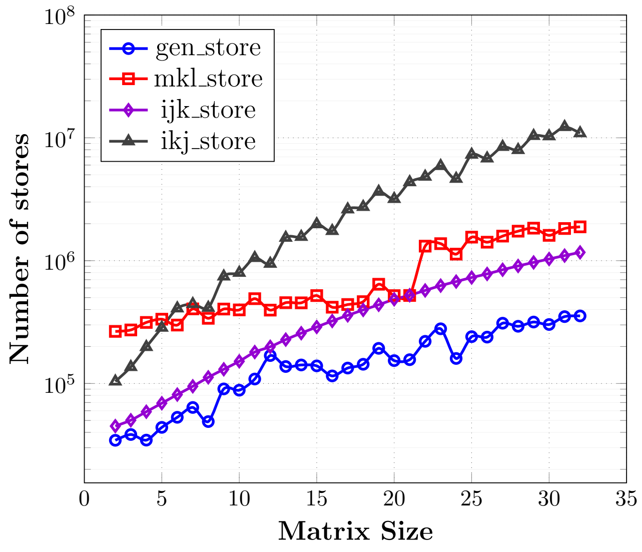


Figure 4.2: # of store instructions

4.4.3 Register Data Reuse and Locality

Similarly to the blocking strategies for better cache reuse in numerically intensive operations (e.g., large matrix-matrix products), we focus on register blocking to increase the performance. Our study concludes that the register reuse ends up being the key factor for performance. The idea is that when data is loaded into SIMD register, it will be reused as much as possible before its replacement by new data. The amount of data that can be kept into registers becomes an important tuning parameter. For example, an 8×8 matrix requires 16 256-bit AVX-2 registers to be completely loaded. As the targeted hardware consists of only 16 256-bit AVX-2 registers, one can expect that loading the whole B will not be optimal as we will have to reload the vectors for A and C. However, if we load only 8 registers for B, which is equal to 4 rows, we can compute a row of C at each iteration and reuse these 8 registers for each iteration. We propose an auto-tuning process to check all the possible scenarios and provide the best option. This reduces the number of load, store, and total instructions from $O(n^2)$ to $O(n)$, compared to a classical *ijk* or *ikj* implementation as depicted in Figures 4.1, 4.2, and 4.3, respectively.

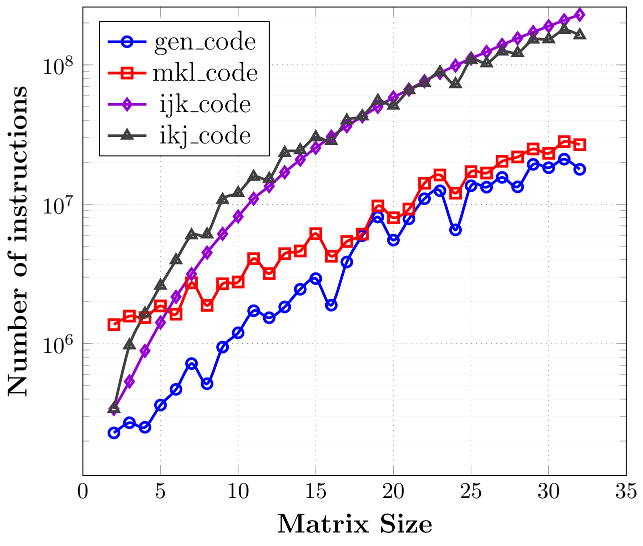


Figure 4.3: Total CPU instruction count

4.4.4 Algorithmic Advancements

Algorithm 9 is an example of our method for a matrix-matrix product of 16×16 matrices. In this pseudo-code, we start by loading four 256-bit AVX-2 registers with values of B which correspond to the first row. These registers are reused throughout the algorithm. In the main loop (Lines 4-14), we start by computing the first values of every multiplication (stored into a register named M=A×B) based on the prefetched register in line 1.

Then, we iterate on the remaining rows (Lines 7-11) loading B, multiplying each B by a value of A, and adding the result into M. Once the iteration over a row is accomplished, the value of M is the final result of A×B and thus, we can load the initial values of C, multiply by α and β, and store it back before moving toward the next iteration such a way to minimize the load/store as shown in Figure 4.14.2.

Each C ends up being loaded/stored once. We apply this strategy to matrix sizes ranging from 8 to 32 as for smaller sizes the whole matrix can fit in registers. Different blocking strategies (square versus rectangular) have been studied through

Algorithm 9 Generic matrix-matrix product applied to matrices of size 16×16

```
1:  Load B0, B1, B2, B3
2:  Load $\alpha$, $\beta$
3:  S = 16
4:  for i = 0, 1, ... , S-1 do
5:      Load A[i*S]
6:      Mi0 = A[i*S] * B0; ... Mi3 = A[i*S] *B3
7:      for u = 1, 2, ... , S-1 do
8:          Load A[i*S + u]
9:          Load Bu0, Bu1, Bu2, Bu3
10:         Mi0 += A[i*S+u] * Bu0; ... Mi3 += A[i*S+u] *Bui3
11:     end for
12:     Mi0 = $\alpha$ Mi0 + $\beta$ (Load Ci0); ... Mi3 = $\alpha$ Mi3 + $\beta$ (Load Ci3)
13:     Store Mi0, Mi1, Mi2, Mi3
14: end for
```

our auto-tuning process in order to achieve the best performance. We generate each matrix-matrix product function at compile time with C++ templates. The matrix size is passed as a function parameter using C++ integral constants.

As explained in Section 1.4, using SIMD units is quite simple if the algorithm has been well designed. As we have taken into consideration from the start of the algorithmic design SIMD problems, we can see that Algorithm 9 is very similar to the code in Listing 4.18. The main differences come from the syntactic sugar of the Intel SIMD instruction set.

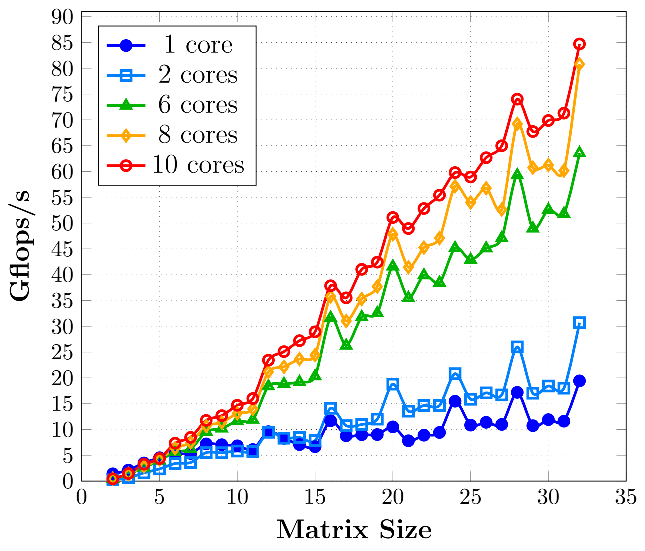


Figure 4.4: Effect of the number of CPU cores

The code written here can only be efficient using the `constexpr` keyword. Indeed, the compiler will only be able to unroll the loops properly at compile time if the number of iterations in the loops are fixed.The inner loop can only be factorized in such a way because we can trust the compiler. Most C++ compilers use the polyhedral model to optimize loops [109] and compile time information is needed for proper optimizations.

```
inline void batch_Mult( const double * A , const double * B , double * C  ,
    double alpha , double beta , std::integral_constant<unsigned long,16>) {

constexpr int ind = 16 * block_num;
auto v_b      = _mm256_loadu_pd( &B[0  ]);
auto v_b_     = _mm256_loadu_pd( &B[4  ]);
auto v_b__    = _mm256_loadu_pd( &B[8  ]);
auto v_b___   = _mm256_loadu_pd( &B[12]);

auto alpha_  = _mm256_set1_pd(alpha);
auto beta_   = _mm256_set1_pd(beta);

for  (int iA = 0; iA < 16; iA++){

  auto tmp = _mm256_set1_pd( A[iA*ind]  );
  auto v_c  = tmp * v_b ;
  auto v_c1 = tmp * v_b_;
  auto v_c2 = tmp * v_b__;
  auto v_c3 = tmp * v_b___;

 for(int d = 1  ; d < 16  ; ++d ){
   tmp = _mm256_set1_pd( A[iA*ind +d  ]);
   auto v_bt       = _mm256_loadu_pd( &B[d*ind]);
   auto v_b_t      = _mm256_loadu_pd( &B[d*ind+ 4]);
   auto v_b__t     = _mm256_loadu_pd( &B[d*ind+ 8]);
   auto v_b___t    = _mm256_loadu_pd( &B[d*ind+ 12]);
   v_c  += tmp * v_bt     ;
   v_c1 += tmp * v_b_t    ;
   v_c2 += tmp * v_b__t  ;
   v_c3 += tmp * v_b___t;
  }

  v_c  = _mm256_loadu_pd(&C[iA*ind])     * beta_ + v_c  * alpha_  ;
  v_c1 = _mm256_loadu_pd(&C[iA*ind+4])   * beta_ + v_c1 * alpha_  ;
  v_c2 = _mm256_loadu_pd(&C[iA*ind+8])   * beta_ + v_c2 * alpha_  ;
  v_c3 = _mm256_loadu_pd(&C[iA*ind+12]) * beta_ + v_c3 * alpha_  ;


  _mm256_storeu_pd( &C[iA*ind]     , v_c );
  _mm256_storeu_pd( &C[iA*ind+4]  , v_c1  );
  _mm256_storeu_pd( &C[iA*ind+8]  , v_c2  );
  _mm256_storeu_pd( &C[iA*ind+12], v_c3  );
 }
}
```

Listing 4.18: Generated matrix-matrix product applied to matrices of size 16 × 16

4.4.5 Effect of the Multi-threading

As described above, operating on matrices of very small sizes is memory-bound computation and thus, increasing the number of CPU cores may not always increase the performance since the performance will be limited by the bandwidth which can be saturated by a few cores. We performed a set of experiments towards clarifying this behavior and illustrate our findings in Figure 4.4. As shown, the notion of perfect speed-up does not exist for a memory-bound algorithm, and adding more cores increases the performance slightly. We performed a bandwidth evaluation when varying the number of cores to find that a single core can achieve about 18 GB/s while 6 and 8 cores (over the available 10 cores) can reach about 88% and 93% of the practical peak bandwidth, which is about 44 GB/s.

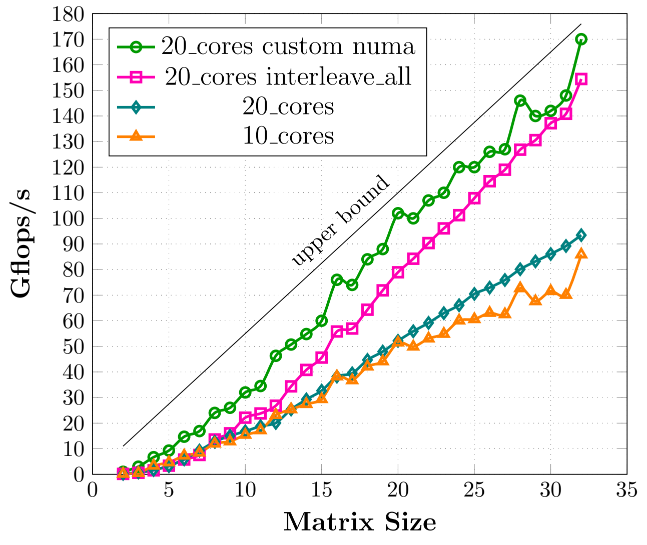


Figure 4.5: Effect of the NUMA memory management

4.4.6 Effect of the NUMA-socket and Memory Location

We also studied NUMA-socket (non-uniform memory access) [66] when using two Xeon sockets as seen in Figure 4.5. A standard memory allocation puts all of the data in the memory slot associated to the first socket until it gets filled, then starts filling the second socket. Since the problem size we are targeting is very small, most of the data is allocated on one socket, and thus using extra 10 cores of the second socket will not increase the performance. This is due to the fact that the data required by the cores of the second socket goes through the memory bus of the first socket, and thus is limited by the bandwidth of one socket (44 GB/s). There are ways to overcome this issue.

By using NUMA with the interleave=all option, which spreads the allocation over the two sockets by memory pages, we can improve the overall performance. However, for very small sizes, we observe that such solution remains far from the optimal bound since data is spread out over the memory of the two sockets without any rules that cores from socket 0 should only access data on socket 0, and vice versa. To further improve performance, we use a specific NUMA memory allocation, which allows us to allocate half of the matrices on each socket. As shown in Figure 4.5,

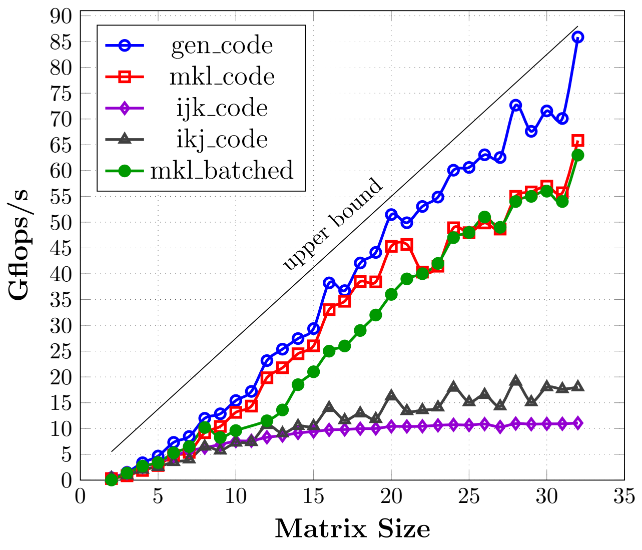


Figure 4.6: Experimental results of the matrix-matrix multiplication on CPU

this allows our implementation to scale over the two sockets and to reach close to the peak bound.

4.4.7 Application to ARM processor

The ARM processor that we use for this benchmark is the CPU of the Tegra X1, a 4-core Cortex A57. The problematics details earlier still apply to the Tegra but on a different scale. Indeed, the ARM intrinsics only support 128-bit vectors which severely limit the SIMD use for double precision computations. In Figure 4.7, we compare the performance between our generated code, an ijk code, an ikj code and OpenBLAS [140] using the latest version available from the develop branch on Github (28/6/2016).

Results follow the same trend we saw on the Intel processors. On very small sizes, ijk and ikj versions are quite efficient as the arithmetic intensity is very low, limiting the usefulness of parallelism. With increased sizes, we start to see these version stale and reach a limit set around 3.5 Gflops. The OpenBLAS version provides good performance but is limited by its blocking model which is not adapted for very small sizes.

4.4.8 Efficient wrapper for GPU batched GEMM

As we have seen in Section 4.3.2, our tensor class supports GPU data. In Figure 4.8, we show the performance comparison between the MAGMA DGEMM batched routine, the read-only cache version (rocache) in black and the fully optimized version in red, and CUBLAS using CUDA 7.5.

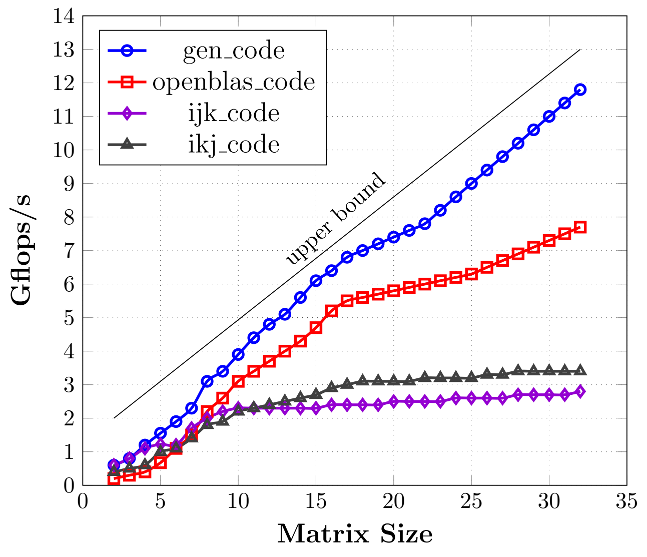


Figure 4.7: Experimental results of the matrix-matrix multiplication on the Tegra X1

The most important factors to optimize GPU performance is to maximize the occupancy and the efficiency of global memory reads. The first one is the ratio between the number of active warps per active cycles and the maximum number of warps that can run on an SM. The second is defined as the ratio between the load throughput requested by the kernel, and the actual required throughput needed to fulfill the kernel load requests.

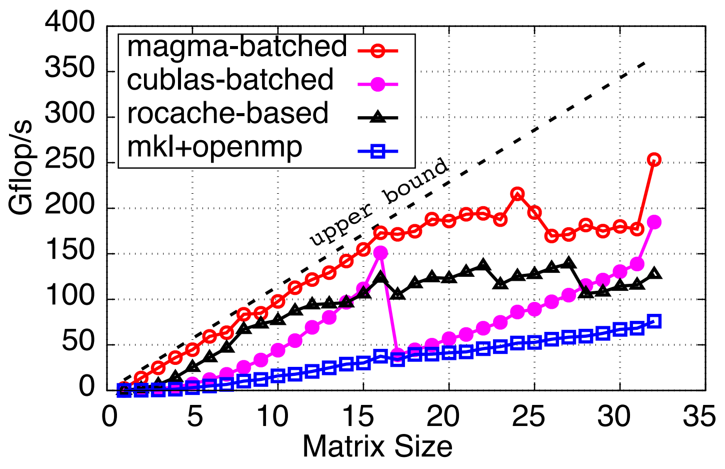


Figure 4.8: GPU performance on K40

As MAGMA manages to reach better performance than CUBLAS, we use the MAGMA routine for GPU batched.

4.5 Conclusion

In this chapter, we provide a method to develop high level interface using modern programming techniques in C++. We also give a technique using architectural tools and performance analysis to implement efficient algorithms for a specific architecture. Furthermore, we show how to develop a flexible interface that can take advantage of optimized libraries such as MAGMA or CUBLAS with minimal effort. This work can easily be extended because the interface is generic. The use of `variadic templates` allow for simple extensions of the container as additional parameters can be added and parsed without modifying current code.

Our work is motivated by a large number of applications, ranging from machine learning to big data analytics, that require fast linear algebra on many independent problems that are of size 32 and smaller. For these applications, the use of batched GEMM for small matrices is essential to achieve performance.

Conclusion and Perspectives

Conclusion

Throughout this manuscript, we studied the challenges behind designing high-level software and optimized codes for modern architectures. It is currently one of the main challenge faced to develop high performance software. The new Top 500[7] ranking of June 2016 has shown the appearance of the Chinese machine from the National Supercomputing Center in Wuxi. This machine is now ranked first in the Top 500 and has a new architecture and applications must be designed specifically for it [55][8]). The software stack for this machine includes compiler support for C, C++ and Fortran. Currently, C++ is the most common high level language used for high performance computing. While algorithms must be reimplemented for this machine, a library can just be extended.

In the first part of this thesis, we presented a method of high level programming that takes into account the features of heterogeneous architectures and the properties of matrices to build a generic dense linear algebra solver. Our programming model supports both implicit or explicit data transfers for GPUs and IGPs. 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 using generative programming is a valid software development approach for addressing these issues while maintaining a high level of performance. We focused on supporting programming models for heterogeneous architectures and we addressed this issue. The resulting software has been integrated into a C++ library called NT^2 which is available on Github.

The second contribution of our thesis is to provide a multi-stage system that can alleviate interoperability problems between the CPU and GPU programming models. Our multistage approach is used to automatically generate GPU code for CPU-based element-wise expressions and parallel skeletons while allowing for type-safe program generation. This work has also been integrated into the NT^2 library and used conjointly with previous work. In this context, we provide an adaptable strategy to generate CUDA kernels directly from a single C++ source file containing NT^2 statements. We illustrated our approach on a selection of examples and the performance obtained is close to the hardware capability and exhibits benefits compared to other solutions. Finally, we showed that this approach can be extended to new architectures or implemented using other programming languages.

[7]`http://www.top500.org/`
[8]http://www.netlib.org/utk/people/JackDongarra/PAPERS/sunway-report-2016.pdf

Our last contribution is to apply high level programming techniques to batched computations and tensor contractions using modern C++. Using the experience we gained from our previous work, we were able to show that by combining a high level programming approach and advanced parallel programming techniques, we can outperform state of the art numerical libraries without relying on current libraries. To do so, we studied the issues around batched computations, memory locality and code vectorization to implement a highly optimized matrix-matrix product for small sizes using SIMD instructions. To obtain a near optimal performance, we conducted an extensive study over the performance counters using the PAPI tool and the Intel memory checker V3.0 for memory bandwidth.

Perspectives

As future work, we are interested in developing support for the OpenCL 2.0 standard in both our solver and multi-stage programming model. As CUDA is not generic enough and works only for NVIDIA cards, it is a limiting factor to develop a multi-platform library. An implementation of MSP in NT^2 has been proposed for OpenCL but it is still not complete. The recent Tegra X1 card also has limited support for OpenCL which is a problem. As recent library have added a layer on top of OpenCL for C++, it can become interesting to study and use such an approach.

We have also started looking at applying similar high level programming techniques to sparse problems. In Chapter 4, we presented a first approach to tensors and an application to batched GEMMs. It would be interesting to keep digging in this direction and try to support more features and benchmark applications.

With the incoming new architectures such as the Xeon Phi Knight's Landing or the future NVIDIA GPUs with NVLINK and IBM processors, current software will need to be extended. Indeed, these new architectures will bring changes in the programming models and load-balancing strategies. Proving that a high level generic approach can be a safe and efficient way to program these architectures will be a challenge to face.

Bibliography

[1] Future directions in tensor-based computation and modeling, May 2009. Workshop. Arlington, Virginia. Technical report published at http://www.cs.cornell.edu/CV/TenWork/FinalReport.pdf. (Cited on page 71.)

[2] A. Abdelfattah, M. Baboulin, V. Dobrev, J. Dongarra, C. Earl, J. Falcou, A. Haidar, I. Karlin, T. Kolev, I. Masliah, et al. High-performance tensor contractions for GPUs. *Procedia Computer Science*, 80:108–118, 2016. (Cited on page 71.)

[3] D. Abrahams and A. Gurtovoy. *C++ template metaprogramming: concepts, tools, and techniques from Boost and beyond.* Addison-Wesley Professional, 2004. (Cited on pages 25 and 27.)

[4] S. Agostinelli, J. Allison, K al Amako, J. Apostolakis, H. Araujo, P. Arce, M. Asai, D. Axen, S. Banerjee, G. Barrand, et al. Geant4—a simulation toolkit. *Nuclear instruments and methods in physics research section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, 506(3):250–303, 2003. (Cited on page 8.)

[5] N. Ahmed, N. Mateev, and K. Pingali. Tiling imperfectly-nested loop nests. In *Supercomputing, ACM/IEEE 2000 Conference*, pages 31–31, Nov 2000. (Cited on page 82.)

[6] 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.* SIAM, Philadelphia, 3 edition, 1999. (Cited on page 7.)

[7] 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*, 23(2):187–198, 2011. (Cited on page 34.)

[8] 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. (Cited on page 48.)

[9] 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)*, pages 14–24, 2012. (Cited on page 51.)

[10] 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*, 180(12):2526–2533, 2009. (Cited on pages 14 and 44.)

[11] 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)*, volume 9 of *Procedia Computer Science*, pages 17–26. Elsevier, 2012. (Cited on pages 14 and 51.)

[12] M. Baboulin, J. Dongarra, J. Demmel, S. Tomov, and V. Volkov. Enhancing the performance of dense linear algebra solvers on gpus in the magma project. Poster at Supercomputing (SC'08), Austin USA, November 15, 2008. (Cited on page 7.)

[13] 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*, 16(7):517–533, 2009. (Cited on page 44.)

[14] M. Baboulin, J. Dongarra, J. Herrmann, and S. Tomov. Accelerating linear system solutions using randomization techniques. *ACM Trans. Math. Softw.*, 39(2), 2013. (Cited on pages 14 and 51.)

[15] D. F. Bacon, S. L. Graham, and O. J. Sharp. Compiler transformations for high-performance computing. *ACM Comput. Surv.*, 26(4):345–420, December 1994. (Cited on page 82.)

[16] 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. (Cited on page 51.)

[17] N. Bell and J. Hoberock. Thrust: A productivity-oriented library for CUDA. *GPU Computing Gems*, 7, 2011. (Cited on page 36.)

[18] K. Bergman, S. Borkar, D. Campbell, W. Carlson, W. Dally, M. Denneau, P. Franzon, W. Harrod, K. Hill, J. Hiller, et al. Exascale computing study: Technology challenges in achieving exascale systems. *Defense Advanced Research Projects Agency Information Processing Techniques Office (DARPA IPTO), Tech. Rep*, 15, 2008. (Cited on page 8.)

[19] A. Björck. Stability analysis of the method of semi-normal equations for least squares problems. *Linear Algebra and its Applications*, 88/89:31–48, 1987. (Cited on page 43.)

[20] A. Björck. *Numerical methods for least squares problems.* Siam, 1996. (Cited on pages 13 and 43.)

[21] L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. *ScaLAPACK Users' Guide*. SIAM, Philadelphia, 1997. (Cited on page 7.)

[22] W. Bright. D language Templates revisited. `http://dlang.org/templates-revisited.html`. (Cited on page 27.)

[23] R. Brun and F. Rademakers. Root—an object oriented data analysis framework. *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, 389(1):81–86, 1997. (Cited on page 8.)

[24] H. Chafi, Z. DeVito, A. Moors, T. Rompf, A. K. Sujeeth, P. Hanrahan, M. Odersky, and K. Olukotun. Language virtualization for heterogeneous parallel computing. In *ACM Sigplan Notices*, volume 45, pages 835–847. ACM, 2010. (Cited on page 26.)

[25] B. L. Chamberlain, D. Callahan, and H. P. Zima. Parallel programmability and the chapel language. *International Journal of High Performance Computing Applications*, 21(3):291–312, 2007. (Cited on page 23.)

[26] G. Chrysos. Intel® xeon phi^{TM} coprocessor-the architecture. *Intel Whitepaper*, 2014. (Cited on page 15.)

[27] M. Cole. Bringing skeletons out of the closet: a pragmatic manifesto for skeletal parallel programming. *Parallel computing*, 30(3):389–406, 2004. (Cited on page 28.)

[28] S. Conrad. Armadillo: An open source C++ linear algebra library for fast prototyping and computationally intensive experiments. Technical report, NICTA, Australia, October 2010. (Cited on page 33.)

[29] T. Cramer, D. Schmidl, M. Klemm, and D. an Mey. OpenMP Programming on Intel Xeon Phi Coprocessors: An Early Performance Comparison. In *Proceedings of the Many-core Applications Research Community (MARC) Symposium at RWTH Aachen University*, pages 38–44, November 2012. (Cited on page 52.)

[30] K. Czarnecki, U. Eisenecker, R. Glück, D. Vandevoorde, and T. Veldhuizen. Generative programming and active libraries. In *Generic Programming*, pages 25–39. Springer, 2000. (Cited on page 34.)

[31] K. Czarnecki and U. W. Eisenecker. Components and generative programming. In *Software Engineering—ESEC/FSE'99*, pages 2–19. Springer, 1999. (Cited on page 34.)

[32] K. Czarnecki and U. W. Eisenecker. *Generative programming - methods, tools and applications*. Addison-Wesley, 2000. (Cited on page 56.)

[33] K. Czarnecki and U. W. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 2000. (Cited on page 35.)

[34] K. Czarnecki, J. T. O'Donnell, J. Striegnitz, and W. Taha. DSL Implementation in MetaOCaml, Template Haskell, and C++. In *Domain-Specific Program Generation*, pages 51–72, 2003. (Cited on page 52.)

[35] K. Czarnecki, K. Østerbye, and M. Völter. Generative programming. In *Object-Oriented Technology ECOOP 2002 Workshop Reader*, pages 15–29. Springer, 2002. (Cited on page 32.)

[36] J. de Guzman. The boost spirit parser generator framework. *URL http://spirit. sourceforge. net*, 2008. (Cited on page 58.)

[37] D. Demidov, K; Ahnert, K. Rupp, and P. Gottschling. Programming CUDA and OpenCL: a case study using modern C++ libraries. *SIAM Journal on Scientific Computing*, 35(5):C453–C472, 2013. (Cited on pages 37 and 54.)

[38] 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.*, 32(2):325–351, 2006. (Cited on pages 13 and 44.)

[39] Z. DeVito, J. Hegarty, A. Aiken, P. Hanrahan, and J. Vitek. Terra: a multi-stage language for high-performance computing. In *ACM SIGPLAN Notices*, volume 48, pages 105–116. ACM, 2013. (Cited on page 26.)

[40] K. Diefendorff, P. K. Dubey, R. Hochsprung, and H. Scale. Altivec extension to PowerPC accelerates media processing. *Micro, IEEE*, 20(2):85–95, 2000. (Cited on page 15.)

[41] J. Dongarra. Basic Linear Algebra Subprograms Technical Forum Standard. *Int. J. of High Performance Computing Applications*, 16(1), 2002. (Cited on page 7.)

[42] J. Dongarra, J. R. Bunch, C. B. Moler, and G. W. Stewart. *LINPACK users' guide*. Siam, 1979. (Cited on page 7.)

[43] G. Dos Reis and B. Stroustrup. General constant expressions for system programming languages. In *Proceedings of the 2010 ACM Symposium on Applied Computing*, pages 2131–2136. ACM, 2010. (Cited on page 75.)

[44] J. Eckhardt, R. Kaiabachev, E. Pasalic, K. Swadi, and W. Taha. Implicitly heterogeneous multi-stage programming. *New Gen. Comput.*, 25(3):305–336, January 2007. (Cited on page 53.)

[45] H. C. Edwards, C. R. Trott, and D. Sunderland. Kokkos: Enabling manycore performance portability through polymorphic memory access patterns. *Journal of Parallel and Distributed Computing*, 74(12):3202–3216, 2014. (Cited on page 55.)

[46] 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*, pages 5–14. ACM, 2010. (Cited on page 42.)

[47] P. Estérie. *Multi-Architectural Support: A Generic and Generative Approach*. PhD thesis, Paris 11, 2014. (Cited on page 36.)

[48] P. Estérie, 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*, 74(12):3240–3253, 2014. (Cited on page 27.)

[49] P. Estérie, M. Gaunard, J. Falcou, J. T. Lapresté, and B. Rozoy. Boost.SIMD: generic programming for portable SIMDization. In *Proceedings of the 21st international conference on Parallel architectures and compilation techniques*, pages 431–432. ACM, 2012. (Cited on page 22.)

[50] D. Fabregat-Traver and P. Bientinesi. Application-tailored linear algebra algorithms: A search-based approach. *International Journal of High Performance Computing Applications*, page 1094342013494428, 2013. (Cited on page 32.)

[51] D. Fabregat-Traver and P. Bientinesi. A domain-specific compiler for linear algebra operations. In *High Performance Computing for Computational Science-VECPAR 2012*, pages 346–361. Springer, 2013. (Cited on page 32.)

[52] J. Falcou. *Software Abstractions for Parallel Architectures*. Accreditation to supervise research, Universite de Paris 11, December 2014. (Cited on page 35.)

[53] L. V. Foster. Gaussian elimination with partial pivoting can fail in practice. *SIAM Journal on Matrix Analysis and Applications*, 15(4):1354–1362, 1994. (Cited on page 10.)

[54] M. Fowler. Language workbenches: The killer-app for domain specific languages. `http://www.issi.uned.es/doctorado/generative/Bibliografia/Fowler.pdf`, 2005. (Cited on page 32.)

[55] H. Fu, J. Liao, J. Yang, L. Wang, Z; Song, X. Huang, C. Yang, W. Xue, F. Liu, F. Qiao, et al. The sunway taihulight supercomputer: system and applications. *Science China Information Sciences*, pages 1–16, 2016. (Cited on page 92.)

[56] S. H. Fuller and Editors; Committee on Sustaining Growth in Computing Performance; National Research Council L .I. Millett. *The Future of Computing Performance: Game Over or Next Level?* The National Academies Press, 2011. (Cited on page 73.)

[57] G. H. Golub and C. F. Van Loan. *Matrix Computations*, volume 3. JHU Press, 2012. (Cited on pages 9, 11 and 12.)

[58] 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*, pages 116–125, New York, NY, USA, 2007. ACM Press. (Cited on page 33.)

[59] B. J. Gough and R. M. Stallman. *An Introduction to GCC*. Network Theory Ltd., 2004. (Cited on page 47.)

[60] D. Gregor and J. Järvi. Variadic templates for c++. In *Proceedings of the 2007 ACM symposium on Applied computing*, pages 1101–1108. ACM, 2007. (Cited on page 75.)

[61] 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*, volume 41, pages 291–310. ACM, 2006. (Cited on page 26.)

[62] K. Gregory and A. Miller. C++ amp: accelerated massive parallelism with microsoft visual c++. 2014. (Cited on page 54.)

[63] L. Grigori, J. Demmel, and H. Xiang. CALU: a communication optimal LU factorization algorithm. *SIAM J. Matrix Anal. and Appl.*, 32:1317–1350, 2011. (Cited on page 14.)

[64] G. Guennebaud, B. Jacob, et al. Eigen: A C++ Linear Algebra Library. `http://eigen.tuxfamily.org/`, 2014. (Cited on pages 33 and 55.)

[65] MATLAB User's Guide. The mathworks. *Inc., Natick, MA*, 5:333, 1998. (Cited on page 25.)

[66] G. Hager and G. Wellein. *Introduction to High Performance Computing for Scientists and Engineers*. CRC Press, 2011. (Cited on page 87.)

[67] A. Haidar, T. Dong, P. Luszczek, S. Tomov, and J. Dongarra. Batched matrix computations on hardware accelerators based on GPUs. *IJHPCA*, doi:10.1177/1094342014567546, 02 2015. (Cited on page 79.)

[68] J. L. Hennessy and D. A. Patterson. *Computer Architecture, Fifth Edition: A Quantitative Approach*. Morgan Kaufmann Publ. Inc., San Francisco, CA, USA, 2011. (Cited on page 82.)

[69] N. J. Higham. *Accuracy and Stability of Numerical Algorithms*. SIAM, Philadelphia, 2 edition, 2002. (Cited on pages 8, 10, 13 and 44.)

[70] J. Hoberock and N. Bell. Thrust: A parallel template library. *Online at http://thrust. googlecode. com*, 42:43, 2010. (Cited on page 54.)

[71] P. Hudak. Building domain-specific embedded languages. *ACM Comput. Surv.*, 28(4es), December 1996. (Cited on page 52.)

[72] IBM. System/360 Scientific Subroutine Package. http://media.ibm1130.org/1130-006-ocr.pdf, 1968. (Cited on page 7.)

[73] K. Iglberger, G. Hager, J. Treibig, and U. Rüde. Expression templates revisited: a performance analysis of current methodologies. *SIAM Journal on Scientific Computing*, 34(2):C42–C69, 2012. (Cited on page 33.)

[74] Intel. Math Kernel Library (MKL). http://www.intel.com/software/products/mkl/. (Cited on page 47.)

[75] J. Järvi and J. Freeman. C++ lambda expressions and closures. *Science of Computer Programming*, 75(9):762–772, 2010. (Cited on page 78.)

[76] J. Järvi and B. Stroustrup. Decltype and auto (revision 3). *ANSI/ISO C++ Standard Committee Pre-Sydney mailing*, (1607):04–0047, 2004. (Cited on page 75.)

[77] W. Kahan. Ieee standard 754 for binary floating-point arithmetic. *Lecture Notes on the Status of IEEE*, 754(94720-1776):11, 1996. (Cited on page 8.)

[78] D. R. Kincaid and E. W. Cheney. *Numerical analysis: mathematics of scientific computing*, volume 2. American Mathematical Soc., 2002. (Cited on page 10.)

[79] L. Kohn, G. Maturana, M. Tremblay, A. Prabhu, and G. Zyner. The visual instruction set (VIS) in UltraSPARC. In *Computer Conference, IEEE International*, pages 462–462. IEEE Computer Society, 1995. (Cited on page 15.)

[80] T. G. Kolda and B. W. Bader. Tensor decompositions and applications. *SIAM review*, 51(3):455–500, 2009. (Cited on page 8.)

[81] T. G. Kolda and B. W. Bader. Tensor decompositions and applications. *SIAM Rev.*, 51(3):455–500, August 2009. (Cited on pages 70 and 71.)

[82] S. C. Kramer and J. Hagemann. SciPAL: Expression templates and composition closure objects for high performance computational physics with cuda and openmp. *ACM Transactions on Parallel Computing*, 1(2):15, 2015. (Cited on page 55.)

[83] M. Kretz. *Extending C++ for explicit data-parallel programming via SIMD vector types*. PhD thesis, 2015. (Cited on page 22.)

[84] R. B. Lee. Accelerating multimedia with enhanced microprocessors. *IEEE Micro*, 15(2):22–32, 1995. (Cited on page 15.)

[85] M. Lehn. FLENS. http://http://www.mathematik.uni-ulm.de/~lehn/FLENS/, 2013. (Cited on page 33.)

[86] X. Leroy, D. Doligez, A. Frisch, J. Garrigue, D. Rémy, and J. Vouillon. The objective caml system release 3.12. *Documentation and user's manual. INRIA*, 2010. (Cited on page 26.)

[87] 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*, 13(1):324, 2012. (Cited on page 32.)

[88] D. Loshin. *Efficient Memory Programming*. McGraw-Hill Profess., 1st edition, 1998. (Cited on page 82.)

[89] K. Lutz. Boost.Compute. `http://github.com/kylelutz/compute`, 2015. (Cited on page 55.)

[90] 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*, pages 362–378. Springer, 2013. (Cited on page 32.)

[91] S. Markidis, J. Gong, M. Schliephake, E. Laure, A. Hart, D. Henty, K. Heisey, and P. F. Fischer. Openacc acceleration of the nek5000 spectral element code. *IJHPCA*, 29(3):311–319, 2015. (Cited on page 72.)

[92] I. Masliah, A. Abdelfattah, A. Haidar, S. Tomov, M. Baboulin, J. Falcou, and J. Dongarra. High-performance matrix-matrix multiplications of very small matrices. `http://icl.cs.utk.edu/news_pub/submissions/main.pdf`, 2016. (Cited on page 71.)

[93] I. Masliah, M. Baboulin, and J. Falcou. Metaprogramming dense linear algebra solvers applications to multi and many-core architectures. In *2015 iIEEE TrustCom/BigDataSE/ISPA, Helsinki, Finland, August,*, volume 3, pages 69–76, 2015. (Cited on page 30.)

[94] I. Masliah, M. Baboulin, and J. Falcou. Meta-programming and multi-stage programming for gpgpus. `https://hal.inria.fr/hal-01204661/document`, 2016. (Cited on page 53.)

[95] J. D. McCalpin. Memory bandwidth and machine balance in current high performance computers. *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, pages 19–25, December 1995. (Cited on page 81.)

[96] 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. (Cited on page 36.)

[97] T. Nelson, A. Rivera, P. Balaprakash, M. W. Hall, P. D. Hovland, E. R. Jessup, and B. Norris. Generating efficient tensor contractions for gpus. Technical report, 2015. (Cited on page 72.)

[98] E. Niebler. Proto : A compiler construction toolkit for DSELs. In *Proceedings of ACM SIGPLAN Symposium on Library-Centric Software Design*, 2007. (Cited on page 27.)

[99] D. Nuzman, S. Dyshel, E. Rohou, I. Rosen, K. Williams, D. Yuste, A. Cohen, and A. Zaks. Vapor simd: Auto-vectorize once, run everywhere. In *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, pages 151–160. IEEE Computer Society, 2011. (Cited on page 22.)

[100] D. Nuzman and R. Henderson. Multi-platform auto-vectorization. In *Proceedings of the International Symposium on Code Generation and Optimization*, pages 281–294. IEEE Computer Society, 2006. (Cited on page 22.)

[101] NVIDIA. *NVIDIA CUDA C Programming Guide*, 04/16/2012. Version 4.2. (Cited on page 47.)

[102] J. D.. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, Aa. Lefohn, and T. J. Purcell. A Survey of General-Purpose Computation on Graphics Hardware. *Computer Graphics Forum*, 26(1):80–113, 2007. (Cited on page 52.)

[103] C. C. Paige and M. A. Saunders. LSQR: An algorithm for sparse linear equations and sparse least squares. *ACM Transactions on Mathematical Software*, 8(1):43–71, 1982. (Cited on page 47.)

[104] D. S. Parker. Random butterfly transformations with applications in computational linear algebra. Technical Report CSD-950023, Computer Science Department, UCLA, 1995. (Cited on page 14.)

[105] E. Peise and P. Bientinesi. Recursive algorithms for dense linear algebra: The relapack collection. *arXiv preprint arXiv:1602.06763*, 2016. (Cited on page 7.)

[106] A. Peleg and U. Weiser. MMX technology extension to the Intel architecture. *Micro, IEEE*, 16(4):42–50, 1996. (Cited on page 15.)

[107] M. VF Pereira and L. MVG. Pinto. Multi-stage stochastic optimization applied to energy planning. *Mathematical Programming*, 52(1-3):359–375, 1991. (Cited on page 56.)

[108] C. Pheatt. Intel® threading building blocks. *Journal of Computing Sciences in Colleges*, 23(4):298–298, 2008. (Cited on page 23.)

[109] L. N. Pouchet, C. Bastoul, A. Cohen, and J. Cavazos. Iterative optimization in the polyhedral model: Part ii, multidimensional time. In *ACM SIGPLAN Notices*, volume 43, pages 90–100. ACM, 2008. (Cited on page 86.)

[110] M. Püschel, J. MF 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*, 18(1):21–45, 2004. (Cited on page 32.)

[111] N. Rajovic, A. Rico, N. Puzovic, C. Adeniyi-Jones, and A. Ramirez. Tibidabo: Making the case for an arm-based hpc system. *Future Generation Computer Systems*, 36:322–334, 2014. (Cited on page 22.)

[112] T. Rompf and M. Odersky. Lightweight modular staging: a pragmatic approach to runtime code generation and compiled dsls. In *Acm Sigplan Notices*, volume 46, pages 127–136. ACM, 2010. (Cited on page 26.)

[113] K. Rupp, F. Rudolf, and J. Weinbub. ViennaCL-a high level linear algebra library for GPUs and multi-core CPUs. *Proc. GPUScA*, pages 51–56, 2010. (Cited on page 54.)

[114] D. C Schmidt. Guest editor's introduction: Model-driven engineering. *Computer*, 39(2):0025–31, 2006. (Cited on page 35.)

[115] P. Schweitzer, S. Cipière, A. Dufaure, H. Payno, Y. Perrot, DRC Hill, and L. Maigne. Performance evaluation of multithreaded geant4 simulations using an intel xeon phi cluster. *Scientific Programming*, 2015, 2015. (Cited on page 8.)

[116] T. Sheard and S. P. Jones. Template Meta-programming for Haskell. *SIGPLAN Not.*, 37(12):60–75, December 2002. (Cited on page 27.)

[117] J. Shin, M. W. Hall, J. Chame, C. Chen, P. F. Fischer, and P. D. Hovland. Speeding up nek5000 with autotuning and specialization. In *Proceedings of the 24th ACM International Conference on Supercomputing*, ICS '10, pages 253–262, New York, NY, USA, 2010. ACM. (Cited on page 72.)

[118] J. Siek and A. Lumsdaine. Essential language support for generic programming. *ACM SIGPLAN Notices*, 40(6):73–84, 2005. (Cited on page 27.)

[119] 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*, pages 1–8. IEEE, 2008. (Cited on page 32.)

[120] E. Solomonik, D. Matthews, J. Hammond, J. Stanton, and J. Demmel. A massively parallel tensor contraction framework for coupled-cluster computations. Technical Report UCB/EECS-2014-143, EECS Department, University of California, Berkeley, Aug 2014. (Cited on page 72.)

[121] P. Springer, J. R Hammond, and P. Bientinesi. Ttc: A high-performance compiler for tensor transpositions. *arXiv preprint arXiv:1603.02297*, 2016. (Cited on page 72.)

[122] K. Stock, T. Henretty, I. Murugandi, P. Sadayappan, and Robert J. Harrison. Model-driven simd code generation for a multi-resolution tensor kernel. In *IPDPS*, pages 1058–1067. IEEE, 2011. (Cited on page 72.)

[123] Khronos OpenCL Working Group — SYCL subgroup. Opencl sycl provisional specification. `https://www.khronos.org/registry/sycl/specs/sycl-1.2.pdf`, 2015. (Cited on page 54.)

[124] W. Taha. A gentle introduction to multi-stage programming. In *Domain-Specific Program Generation*, pages 30–50. Springer, 2004. (Cited on pages 27 and 56.)

[125] W. Taha, C. Calcagno, X. Leroy, E. Pizzi, E. Pasalic, J. L. Eckhardt, R. Kaiabachev, O. Kiselyov, et al. Metaocaml-a compiled, type-safe, multi-stage programming language, 2006. *See: http://www. metaocaml. org.* (Cited on page 26.)

[126] University Tennessee. PLASMA users' guide, parallel linear algebra software for multicore architectures, version 2.3. `http://icl.cs.utk.edu/plasma/software/`, 2010. (Cited on pages 7 and 31.)

[127] S. Tomov, J. Dongarra, and M. Baboulin. Towards dense linear algebra for hybrid GPU accelerated manycore systems. *Parallel Computing*, 36(5&6):232–240, 2010. (Cited on page 7.)

[128] A. Tran Tan, J. Falcou, D. Etiemble, and H. Kaiser. Automatic task-based code generation for high performance domain specific embedded language. *7th International Symposium on High-Level Parallel Programming and Applications (HLPP 2014)*, 2014. (Cited on page 27.)

[129] L. Tratt. Model transformations and tool integration. *Journal of Software and Systems Modelling*, 4(2):112–122, May 2005. (Cited on page 52.)

[130] 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*, pages 204–215. IEEE, 2003. (Cited on page 33.)

[131] 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*, 11(6):56–63, 2009. (Cited on page 32.)

[132] T. Veldhuizen. Expression templates. *C++ Report*, 7:26–31, 1995. (Cited on pages 27 and 33.)

[133] 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 (OO'98)*. SIAM Press, 1998. (Cited on page 34.)

[134] J. Walter and M. Koch. The boost uBLAS library. `http://www.boost.org/libs/numeric/ublas`, 2002. (Cited on page 33.)

[135] V.M. Weaver, M.Johnson, K.Kasichayanula, J.Ralph, P.Luszczek, and S.Moore D.Terpstra. Measuring energy and power with PAPI. In *41st International Conference on Parallel Processing Workshops*, September 2012. (Cited on page 82.)

[136] M. Wenjing, S. Krishnamoorthy, O. Villay, and K. Kowalski. Acceleration of Streamed Tensor Contraction Expressions on GPGPU-Based Clusters. In *Cluster Computing (CLUSTER), 2010 IEEE International Conference on*, pages 207–216, Sept 2010. (Cited on page 72.)

[137] R. C. Whaley and J. Dongarra. Automatically tuned linear algebra software. In *Proceedings of the 1998 ACM/IEEE Conference on Supercomputing*, pages 1–27. IEEE Computer Society, 1998. (Cited on page 33.)

[138] N. Whitehead and A. Fit-Florea. Precision & Performance: Floating Point and IEEE 754 Compliance for NVIDIA GPUs. *nVidia technical white paper*, 2011. (Cited on page 2.)

[139] J. H. Wilkinson. *Rounding erros in algebraic processes*. Courier Corporation, 1994. (Cited on page 12.)

[140] Z. Xianyi, W. Qian, and Z. Chothia. Openblas. *URL: http://xianyi. github. io/OpenBLAS*, 2012. (Cited on page 88.)

universite PARIS-SACLAY

ÉCOLE DOCTORALE
Sciences et technologies de l'information et de la communication (STIC)

Titre : Méthodes de génération automatique de code appliquées à l'algèbre linéaire numérique dans le calcul haute performance

Mots clefs : programmation générique, programmation générative, C++, algèbre linéaire, GPU

Résumé : Les architectures parallèles sont aujourd'hui présentes dans tous les systèmes informatiques, allant des smartphones aux supercalculateurs en passant par les ordinateurs de bureau. Programmer efficacement ces architectures requiert un effort pluridisciplinaire portant sur les langages dédiés (Domain Specific Languages - DSL), les techniques de génération de code et d'optimisation, et les algorithmes numériques propres aux applications. Dans cette thèse, nous présentons une méthode de programmation haut niveau prenant en compte les caractéristiques des architectures hétérogènes ainsi que les propriétés des matrices pour produire un solveur générique d'algèbre linéaire dense. Dans la mesure où les GPUs sont devenus un outil important pour le calcul haute performance, il est essentiel de les utiliser dans les plateformes de calcul. Nous avons par la suite étendu nos travaux à un modèle de programmation multi-étape ("multi-stage") pour résoudre les problèmes d'interopérabilité entre les modèles de programmation CPU et GPU. Nous utilisons cette technique pour générer automatiquement du code pour accélérateur à partir d'un code effectuant des opérations point par point ou utilisant des squelettes algorithmiques. Enfin, nous montrons comment la programmation haut niveau peut être appliquée à des calculs groupés et des contractions de tenseurs. Tout d'abord, nous expliquons comment concevoir un modèle de conteneur en utilisant des techniques de programmation basées sur le C++ moderne (C++-14). Ensuite, nous avons implémenté un produit de matrices optimisé pour des matrices de petites tailles en utilisant des instructions SIMD.

Title : Automatic code generation methods applied to numerical linear algebra in high performance computing

Keywords : Generic programming, Generative programming, C++, linear algebra, GPU

Abstract : Parallelism in today's computer architectures is ubiquitous whether it be in supercomputers, workstations or on portable devices such as smartphones. Exploiting efficiently these systems for a specific application requires a multidisciplinary effort that concerns Domain Specific Languages (DSL), code generation and optimization techniques as well as application-specific numerical algorithms. In this PhD thesis, we present a method of high level programming that takes into account the features of heterogeneous architectures and the properties of matrices to build a generic dense linear algebra solver. As GPUs have become an asset in high performance computing, incorporating their use in general solvers is an important issue. We extend our approach to a new multistage programming model that alleviates the interoperability problems between the CPU and GPU programming models. Our multistage approach is used to automatically generate GPU code for CPU-based element-wise expressions and parallel skeletons while allowing for type-safe program generation. Finally, we investigate how to apply high level programming techniques to batched computations and tensor contractions. We first explain how to design a simple data container using modern C++-14 programming techniques. Then, we study the issues around batched computations, memory locality and code vectorization to implement a highly optimized matrix-matrix product for small sizes using SIMD instructions.

Université Paris-Saclay
Espace Technologique / Immeuble Discovery
Route de l'Orme aux Merisiers RD 128 / 91190 Saint-Aubin, France