The Numerical Template toolbox: A Modern C++ Design for Scientific Computing

Article *in* Journal of Parallel and Distributed Computing · July 2014

DOI: 10.1016/j.jpdc.2014.07.002

CITATIONS
13

READS
771

5 authors, including:

Mathias Gaunard
NumScale
5 PUBLICATIONS **92** CITATIONS
SEE PROFILE

Jean-Thierry Lapresté
Université Clermont Auvergne
57 PUBLICATIONS **1,301** CITATIONS
SEE PROFILE

Lionel Lacassagne
Sorbonne Université
125 PUBLICATIONS **688** CITATIONS
SEE PROFILE

The Numerical Template Toolbox: A Modern C++ Design for Scientific Computing

Pierre Estérie[a], Joel Falcou[a,b,*], Mathias Gaunard[b], Jean-Thierry Lapresté[c], Lionel Lacassagne[a]

[a] *Laboratoire de Recherche en Informatique, Université Paris Sud, 91405 Orsay France*
[b] *MetaScale SAS, 86 rue de Paris, 91400 Orsay France*
[c] *Institut Pascal, UMR 6602 CNRS/UBP/IFMA, 24 Avenue des Landais, 63171 Aubière*

Abstract

The design and implementation of high level tools for parallel programming is a major challenge as the complexity of modern architectures increases. *Domain Specific Languages* (or DSL) have been proposed as a solution to facilitate this design but few of those *DSL*s actually take full advantage of said parallel architectures. In this paper, we propose a library-based solution by designing a C++ *DSL*s using generative programming: NT^2 . By adapting generative programming idioms so that architecture specificities become mere parameters of the code generation process, we demonstrate that our library can deliver high performance while featuring a high level API and being easy to extend over new architectures.

Keywords: C++ , Embedded Domain Specific Languages, Parallel Skeletons, Generic Programming, Generative Programming

*Principal corresponding author. Phone: +33615077137 - Fax: +33169156579
Email addresses: pierre.esterie@lri.fr (Pierre Estérie), joel.falcou@lri.fr (Joel Falcou), mathias.gaunard@metascale.fr (Mathias Gaunard), lapreste@univ-bpclermont.fr (Jean-Thierry Lapresté), lionel.lacassagne@lri.fr (Lionel Lacassagne)

Introduction

Developing large applications in a simple, fast and efficient way has always been an issue for software developers. For a long time, developers have been limited by cost and availability of computing systems. But in the late 90's, as the yearly increase in CPU frequency started to stall, new ways of using the ever growing number of transistors on chips appeared. SIMD extensions like SSE, Altivec or AVX, multi-processor and multi-core systems, accelerators like GPUs [1] or the Xeon Phi [2] have all reshaped the way computing systems are built. During the same time period, programming methodologies have not changed a lot. Scientific computing applications are still implemented using low-level languages, for instance C or FORTRAN, losing the high-level structure given by the application domain.

Designing **Domain Specific Languages** (or *DSL*) has been presented as a solution to this problem. As *DSL*s allow solutions to be expressed in the idiom and at the level of abstraction of 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. Another example, in thecontext of concurrent programming, is Erlang/OTP [3], a domain specific language that guarantees the properties of concurrent programs. Validation and testing are made easier, since statements written within this *DSL* exhibit the correct concurrent behavior by design.

Domain Specific Embedded Languages (or *DSEL*s) [4, 5] are a subclass of *DSL*s that provide a good way to mitigate the abstraction vs efficiency trade-off. *DSEL*s are languages implemented insidea general-purpose, host language [6]. They share the advantages of *DSL*s as they provide an API based on domain specific entities and relations. However, *DSEL*s do not require a dedicated compiler or interpreter to be used as they exist inside another general purpose language. They are usually implemented as a library-like component – often called **Active Libraries** [7, 8] – in languages providing introspection or constructs to manipulate statements within the language. While such features are common in some functional languages (like OCaml or Haskell) and their template-enabled variants (like MetaOCaml or `template` Haskell), they are less so in imperative languages. Similar features are available in C++ via template based meta-programming [9]. In this context, we propose a new C++ active library called NT^2 – The Numerical Template Toolbox – that provides a MATLAB -inspired *DSEL* in C++ while delivering high performance, supporting a large selection of parallel architectures and keeping a high level of expressiveness thanks to the exploitation of architecture-specific information as early as possible in the code generation process.

This paper is organized as follows: section 1 describes how Generative Programming techniques have been modified to support the features of different architectures. Section 2 presents the NT^2 API. Section 3 describes the application of such idioms in NT^2 . Section 4 presents various NT^2 benchmarks. Finally, we discuss related works in section 5.

1. Tool building with Generative Programming

The wide landscape of parallel architectures available today requires the use of multiple programming models in a single application. As stated earlier, current solutions for writing efficient and portable applications rely on low level tools thus making the development of such applications rather difficult. In this work, we explore how design techniques such as Expression Templates and Generative Programming can be extended and applied to the design of parallel programming libraries.

1.1. Current Methodologies

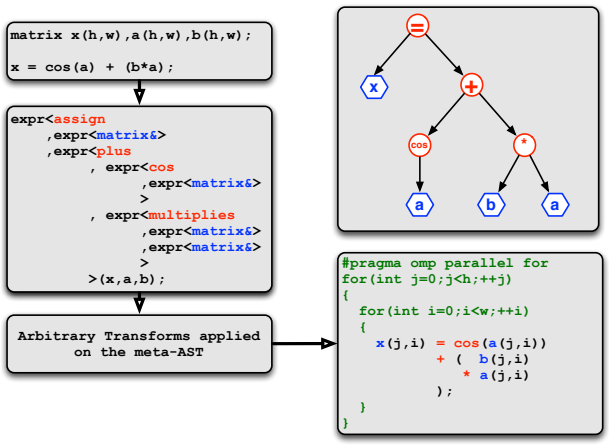
Design techniques for library level software in C++ rely for the most part on the object-oriented approach. Design patterns [10] are a relevant work based on this approach and provide a set of standard reusable designs for software development. Introduced almost 20 years ago, this approach relies on recurrent patterns found in software development that can be generalized and reused in other contexts. However, this method has some limitations when it comes to dealing with many software components. In addition, the generalization of design patterns builds a semantic gap between the domain abstractions and features of programming languages, creating a conflict between expressiveness and performance. Expression Templates are often proposed as a solution to this problem.

1.2. Expression Templates

Expression Templates [11, 12] is a well known technique implementing a form of delayed evaluation in C++ [13]. This idiom allows the compile-time construction of a type representing the Abstract Syntax Tree (AST) of

an arbitrary statement. This is done by overloading functions and operators so they return a lightweight object which represents the current operation in the AST being built rather than performing any computation. Once reconstructed, the AST can be transformed into arbitrary code fragments (see figure 1).

Figure 1: General principles of *Expression Templates*



While Expression Templates should not be limited to the sole purpose of removing temporaries and memory allocations from C++ code, few projects actually go further. The boilerplate code is often as complex as the actual library code, making such tools hard to maintain and extend. To reduce those difficulties, Niebler proposed a C++ compiler construction toolkit for embedded languages called `Boost.Proto` [14]. It allows developers to specify grammars and semantic actions for *DSEL*s and provides a semi-automatic generation of all the template structures needed to perform the AST construction. Compared to *DSEL*s based on hand-written Expressions Templates, designing a new embedded language with `Boost.Proto` is done at a

higher level of abstraction by using pattern matching on *DSEL* statements.

1.3. Generative Programming

Traditionally, tools based on expression templates use only the compile-time AST in a top-down approach, walking through the AST and generating code fragments on the fly. Several tools, such as Blaze [15] or Armadillo [16], perform a small degree of AST optimizations at either compile-time or run-time. In our approach, we maximize the use of the explicit and implicit information carried by the compile-time AST. To achieve this, we explore how **Generative Programming** [17] would help us reach this goal. Following this paradigm, any complex software system can be broken down to a list of interchangeable components in which tasks are clearly identified and a series of generators that combine components by following a set of domain specific rules. Czarnecki [18] proposed a methodology called *DEMRAL* [1] showing a set of axioms to define Generative Programming. This relies upon the fact that algorithmic libraries are based on a set of well defined concepts:

- Algorithms, that are tied to a mathematical or physical theory;
- Domain entities and concepts, which can be represented as Abstract Data Types with container-like properties;
- Specialized functions encapsulating algorithm variants depending on Abstract Data Types properties.

In a simple way, *DEMRAL* reduces the effort needed to develop software libraries by limiting the amount of code to be written. For example, a library

[1] Domain Engineering Method for Reusable Algorithmic Libraries

providing N algorithms operating on P different related data-structures may need the design and implementation of $N \times P$ functions. Using *DEMRAL* , only N generic algorithms and P data structure descriptions are needed as the code generator will specialize the algorithm with respect to the data structure.

1.4. From DEMRAL to AA-DEMRAL

The common factor of all existing *DSEL*s for scientific computing is that the architecture level is mostly seen as a problem that requires specific solutions to be dealt with. The complexity of hand-maintained Expression Templates engines is the main reason why few abstractions are usually added at this level. We propose to integrate the architectural support as another generative component. To do so, we introduce a new methodology which is an hardware-aware extension of the *DEMRAL* methodology: Architecture Aware DEMRAL (*AA-DEMRAL*).

With *AA-DEMRAL* (fig. 2), the implementation components are themselves generated from a generative component which translates an abstract architecture description into a set of concrete implementation components for use by the software generator. In the same way that *DEMRAL* initially removed the complexity of handling a large amount of variations of a given set of algorithms, the Architecture-Aware approach that we propose leverages the work needed for supporting different architectures.

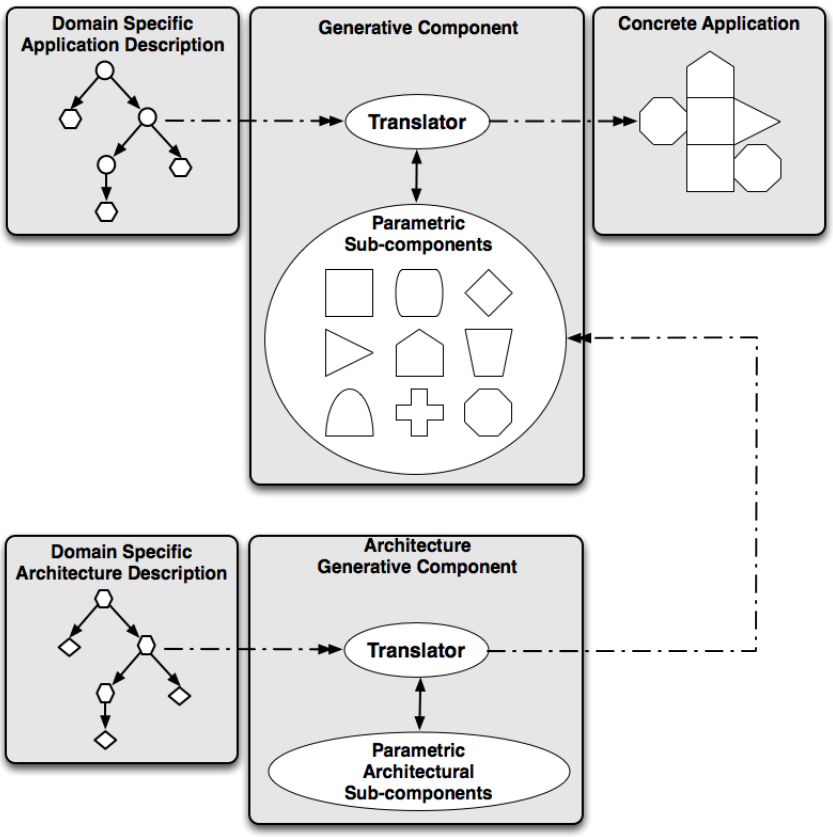


Figure 2: The *AA-DEMRAL* methodology

By designing a small-scale *DSEL* for describing architectures, the software components used by the top-level code generator are themselves the product of a generative component able to analyze an abstract architecture description to specify the structure of these components. If we extend the *DEMRAL* methodology, a library providing N algorithms operating on P different related data structures while supporting Q architectures will only need the design and development of $N+P+Q$ software components and the two different generative components (the hardware one and the software one).

The library is still designed with high re-usability and its development and maintenance are simplified. Such an approach maintains the advantage of the *DEMRAL* methodology thus allowing the focus to be placed on domain related optimizations. In addition, the generic nature of the components allows the code generator to explore a complete configuration space at both levels (hardware and software).

2. The NT^2 Library

As an application of the *AA-DEMRAL* methodology, we implemented a numerical computing C++ library called NT^2 . Based on a language similar to MATLAB , NT^2 simplifies the development of data-parallel applications on a large selection of architectures, with a focus on multi-core systems with SIMD extensions[2].

By design, 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. This section will go through the main elements of the NT^2 API and how they interact with the set of supported architectures.

Internally, NT^2 uses `Boost.Proto` as its expression template engine and

[2]Future works include distributed systems and accelerators based systems.

replaces the classical direct walk-through of the compile-time AST done in most C++ *DSEL*s by the execution of a mixed compile-time/runtime algorithm over a `Boost.Proto` standardized AST structure.

2.1. Basic API

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. A instance of `table` behaves like a MATLAB multi-dimensional array – including 1-based indexing and column major storage order – and supports the same set of operators and functions. Those operators and functions are, unless specified otherwise, applied to every element of the table, following the standard MATLAB semantic. 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, various pattern generators and some statistical and polynomial functions. Moreover, and contrary to similar libraries (see section 2.5), NT^2 provides support for all real and integral types, both real or complex. This, combined with the large set of functions available, allows NT^2 to be used in a wider variety of domains.

Figure 3: Sample NT^2 code

```
table<double> A1 = _(1.,1000.), A2 = A1 + randn(size(A1));
double rms = sqrt( sum(sqr(A1(_) - A2(_))) / numel(A1) );
```

Figure 3 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`. The equivalent MATLAB code is shown in figure 4.

Figure 4: Corresponding MATLAB code

```
A1 = (1.0:1000.0);
A2 = A1 + randn(size(A1));
rms = sqrt( sum(sqr(A1(:) - A2(:))) / numel(A1) );
```

2.2. Linear Algebra support

NT^2 supports the most common matrix decompositions, system solvers and related linear algebra operations via a transparent binding to BLAS and LAPACK. MATLAB syntax is preserved for most of these functions, including the multi-return for decompositions and solvers or the various options for customizing algorithms. The QR decomposition of a given matrix `A` while retrieving the decomposition permutation vector is done this way:

```
tie(Q,R,P) = qr(A,vector_);
```

which can be compared to the equivalent MATLAB code:

```
[Q,R,P] = qr(A,'vector');
```

The `tie` function is optimized to maximize the reuse of memory for the output parameters.

2.3. Compile-time Expression Optimization

Whenever a NT^2 statement is constructed, potential automatic rewriting may occur at compile-time on expressions for which a high-level algorithmic or an architecture-driven optimization is possible. The optimizations considered include:

- Fixed-point transformations such as `trans(trans(x))` can be simplified;
- The fusion of operations such as `mtimes(a, trans(b))` which can directly notify the GEMM BLAS primitive that `b` is tranposed;
- Architecture-driven optimizations such as transforming `a*b+c` into the corresponding Fused Multiply Add operation available on the target: `fma(a,b,c)`;
- Sub-matrix access such as `a(_,i)` into an optimized representation enabling vectorization.

Another important optimization is **loop fusion**. Loop fusion increases cache locality and thus is critical to attain peak performance in large applications. However, the major disadvantage of expression templates based libraries is the fact that the capture and optimization is limited to statements. Various strategies can be put into place to remove this limitation:

- Overload the comma operator to chain statements. Once chained, this large temporary expression object evaluates itself when its destructor is called. This is the most aesthetic way to perform such fusion. However, as it relies on arbitrary code to be executed in destructors, it prevent exception propagation.
- Provide an explicit fusion operator for statements. Although it adds clutter to the user's code, it does not rely on subtle side effects.

NT^2 uses the second method through the multi-purpose `tie` function which group the set of statements to be assigned to a set of containers.

For example, the multi-statement listing 5 can be turned into the single assignment of listing 6.

Figure 5: Multi-statement code fragment

```
table<float> x,y,z;
x = 3.f * y + z;
z = 1.f / x;
```

Figure 6: Loop fusion code fragment

```
table<float> x,y,z;
tie(x,z) = tie( 3.f * y + z, 1.f / x);
```

In listing 6, a single loop nest is generated and parallelized, thus limiting locality issues and reduce the number of barriers between statements if compiled for a shared memory system.

2.4. Supported architectures and runtimes

The main asset of NT^2 is its ability to support architecture and runtime back-ends in an extensible way. The current set of supported architectures and runtimes includes:

- **Support for SIMD extensions** for Altivec on PowerPC and all SSE variations (from SSE2 to SSE4.2) and AVX on x86;
- **Support for shared memory systems** by either using OpenMP [19] or Intel Thread Building Blocks [20];
- **Support for LAPACK-like runtimes** for NETLIB LAPACK, MKL and MAGMA [21] on CUDA capable GPUs.

NT^2 distinguishes between architecture and runtime support so users can, from a single source code, experiment with different runtime combinations for a given architecture until they are satisfied with performance. This system is backed up by a compile-time process that guarantees the performance of any combination of runtime support. NT^2 selects which architectural features are available from either compiler-based options or user-defined preprocessor definitions. In all cases, no additional code changes are required on the NT^2 code to use any of these back-ends.

2.5. Comparison to similar libraries

Table 7 sums up the difference and similarities between NT^2 and a selection of state of the art numerical computation libraries in C++.

Feature	Armadillo	Blaze	Eigen	MTL	uBlas	NT^2
MATLAB API conformance	✓	–	–	–	–	✓
AST optimization	✓	✓	–	–	–	✓
SSEx support	✓	✓	✓	–	–	✓
AVX support	✓	✓	–	–	–	✓
Altivec support	–	–	✓	–	–	✓
Shared memory parallelism	–	✓	–	–	–	✓
BLAS/LAPACK binding	✓	✓	✓	✓	✓	✓
MAGMA binding	–	–	–	–	–	✓

Figure 7: Feature set comparison between NT^2 and similar libraries

The main assets of NT^2 are its MATLAB API conformance and the fact that it covers a larger subset of parallel architectures than most other libraries due to its implementation based on *AA-DEMRAL* .

3. *AA-DEMRAL* based implementation

NT^2 implementation is based on multiple subsystem that enables the *AA-DEMRAL* methodology as shown in figure 8.

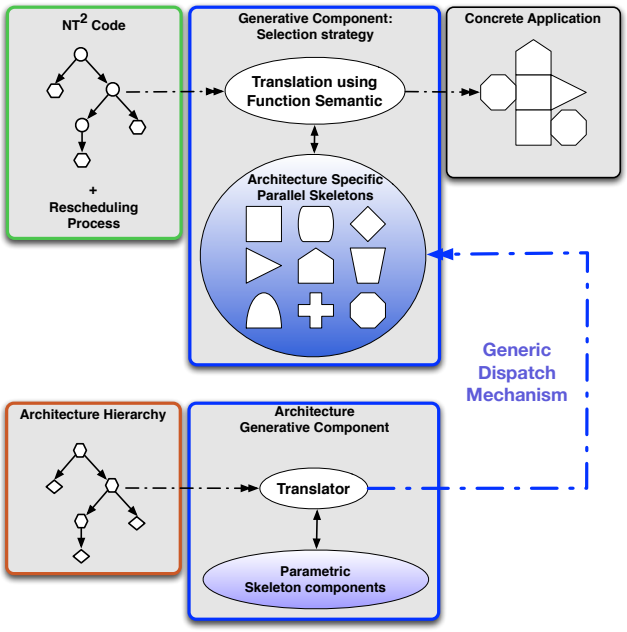


Figure 8: The NT^2 *AA-DEMRAL* implementation principles

These components are:

- A strategy to select the proper implementation of a given function for a given architecture and functions properties (see 3.1). This strategy is embedded in an architecture aware generative component.
- A compile-time description of function's properties (see 3.2) that allows us to pass and use the functions semantic within the selection strategy.

- An architecture description *DSEL* describing architectures and their relationship (see 3.3) that makes the selection strategy aware of architectural features.

- A compile-time process for rescheduling NT^2 statements in a way that optimal loop nests can be generated and a parallel code generator using parallel skeletons that take care of different types and levels of parallelism (see 3.4).

3.1. Function overloads handling

NT^2 has over 300 MATLAB functions, the choice of the most optimal function implementation is based on the architecture and the data types. Therefore, a means of determining the most suitable overload for any given function is required. Classical overloading techniques in C++ include:

- **SFINAE**[3] combined with Traits [22] uses the ability of the C++ compiler to prune incorrect overloads due to errors in dependent type resolution. Extending such a system is often difficult when the conditions for pruning are interrelated.

- **Concepts** [23] rely on a high level semantic description of type properties. Language support is required to resolve overloads based on the conformance of types to a given Concept. Concept-based overloading is still an experimental language feature under consideration for inclusion in a future C++ standard.

[3]Substitution Failure Is Not An Error

- **Tag Dispatching** uses class hierarchies, called **tags**, to represent type properties. To resolve an overload, the tag of a specific argument is extracted and used as an argument in a trampoline function. Unlike SFINAE-based solutions, it uses C++ overloading directly and has the advantage of providing best match selection in cases of multiple matches.

NT^2 implements the *AA-DEMRAL* methodology via an architecture aware Tag Dispatching technique. NT^2 achieves this by providing an extended and generic manner of dispatching functions in template contexts through **an extensible hierarchy system**. This component defines three specialized hierarchies:

- a **type hierarchy:** expresses the properties of the argument types in a similar manner to Concept based overloading;
- a **function hierarchy:** expresses the properties of a function;
- a **architectural hierarchy:** adds architecture specific information to the dispatch process.

Using these hierarchies, NT^2 can aggregate generic components according to the type semantic, the function semantic and the architecture information. The type hierarchy and the function hierarchy provide a mean of exploiting semantic information during the dispatch process while the architecture hierarchy can inject the architecture description inside the dispatch process. The design of the library is then semantic and architectural aware. The Concept of `Hierarchy` is the basis of the dispatching system. It provides an idiomatic

way of defining a family of tags via inheritance. Therefore, an NT^2 function is implemented as a function object which considers data types, the function and the architecture and perform the actual dispatch.

3.2. Functions Hierarchy

The first level of information gathered by the dispatch system is the function properties. Each NT^2 function (as a symbol) is tied to a compile-time structure called a **tag**. Whenever a function `foo` is called, NT^2 tries to find a valid implementation of `foo` by calling a generic function overloaded for a tag class `foo_` which carries the function semantic. Through inheritance, each function tag is registered within the function hierarchy. This hierarchy includes:

- elementwise functions that operate on their arguments at a position p and produce an output at the same position. Elementwise functions do not have any dependencies between operations on different positions. They are the core of NT^2 expressions, and combining them through a specific `elementwise_` hierarchy tag allows us to evaluate them within a single kernel or loop nest. Such functions include: regular functions such as `plus`, data generators such as `zeros` or functions modifying a `table` logical size such as `reshape`.

- non-elementwise functions are not combinable with elementwise functions. Their properties and intrinsic parallelism depends on the function semantic. These non-elementwise functions include reduction and partial reduction functions, scan functions like `cumsum` and external kernels such as BLAS level functions.

Figure 9: Function hierarchy for some NT^2 functions

```
struct plus_ : elementwise_<plus_> {};
struct sum_ : reduction_<sum_,plus_,zero_> {};
struct mtimes_ : unspecified_<mtimes_> {};
```

For example, figure 9 presents the hierarchy tags for various functions available in NT^2. `plus` is registered as an elementwise operation through the `elementwise_` hierarchy tag. `sum` is a reduction and its hierarchy tag defines it as a reduction based on `plus` and `zero`. A matrix-matrix product function is registered as an external kernel.

3.3. Compile-time architecture description

Using its Tag Dispatching technique, NT^2 can now select the correct implementation of a function while still being aware of its properties. NT^2 then selects the best implementation for the current architecture. NT^2 provides a way to register an architecture as a compile-time tag in a hierarchy, similar to the function hierarchy (figure 10).

For every architecture supported, a tag is defined using inheritance to organize related architectural components. In addition to this inheritance scheme, architecture tags may be nested (such as `openmp_`). This nesting is computed at compile-time by exploiting information given by the compiler or by user-defined preprocessor symbols. This nesting is used to automatically generate code for different architecture levels. For example, the default architecture computed for a code compiled using AVX and OpenMP is `openmp_< avx_ >`. This nesting will then be exploited when parallel loop nests are gen-

Figure 10: Some NT^2 architecture hierarchy tag

```
struct cpu_  : unspecified_<cpu_> {}; // cpu_: no special info
struct simd_ : cpu_ {};               // simd_: any SIMD architecture

// shared memory architecture using OpenMP as runtime
template<typename Core> struct openmp_  : Core {};

// SSE enabled architectures
struct sse_  : simd_ {};
struct sse2_ : sse_  {};
...
struct avx_  : sse4_ {};
```

erated through a combination of the OpenMP and AVX backends. Such a scenario is detailed in section 3.4.

3.4. Parallel code generation

The function and architecture tags introduced in the previous section schedule the evaluation of each type of nodes (*i.e.* functions) involved in a single statement. The NT^2 code generator generate a succession of loop nests based on the tag of the node (*i.e.* the function tag) at the top of the current AST. The NT^2 expression evaluation is based on the possibility of computing the size and value type. This size is used to construct a loop, which can be parallelized using arbitrary techniques, which then evaluate the operation for any position p, either in scalar or in SIMD mode. The main entry point of this system is the `run` function that is defined for every function or family of functions (*i.e.* elementwise, reduction, etc.). This selects the best way of evaluating a given function in the context of its local AST and

the current output element position to compute. At this point, NT^2 exploits the information about the function properties to use a specific loop nest generator.

To account for the architecture, NT^2 relies on **Parallel Skeletons** [24]. Parallel skeletons are recurrent parallel patterns designed as higher-order functions that describe an efficient solution to a specific problem. Cole details in [25] the need for specific skeletons providing enough abstraction to be used in the context of parallel frameworks. This abstraction introduces semantic information which is used in the efficient implementation of the corresponding skeletons. Aldinucci addresses this approach with an extensible skeleton environment, Muskel [26]. Kuchen also shows that such an approach can lead to efficient library based implementations without losing levels of abstraction [27].

Skeletons usually behave as higher order functions, *i.e.* functions taking functions as parameters, including other skeletons. This composability reduces the difficulty of designing complex parallel programs as any combination of skeletons is correct by design. The other main advantage of skeletons is the fact that the actual synchronization and scheduling of a skeleton's parallel task is encapsulated within the skeleton. Once a skeleton semantic is defined, programmers do not have to specify how synchronizations and scheduling happen. This has two implications: firstly, skeletons can be specified in an abstract manner encapsulating architecture specific implementations; secondly, the communication/computation patterns are known in advance and can be optimized [28, 29].

Although a large number of skeletons have been proposed in the literature [27, 30], NT^2 focuses on three data-oriented skeletons:

- **transform** applies an arbitrary operation to each (or certain) element(s) of input `tables` and stores the result in some output `tables`.
- **fold** applies a partial reduction of the elements of input `tables` to a given table dimension and stores the result in the output `tables`.
- **scan** applies a prefix scan of the elements of input `tables` to a given table dimension and stores the result in output `tables`.

Those skeletons are tied to families of loop-nests that may or may not be nested. These families are:

- **elementwise loop nests** represent loop nests implementable via a call to **transform** and which can only be nested with other elementwise operations.
- **reduction loop nests** represent loop nests implementable via a call to **fold**. Successive reductions are not generally nestable as they can operate on different dimensions but can contain a nested elementwise loop nest.
- **prefix loop nests** represent loop nests implementable via a call to **scan**. Successive prefix scans, such as reductions, are not nestable but can contain nested elementwise loop nests.

These families of loop nests are used to tag functions provided by NT^2 so that the type of the operation may be introspected to determine its loop

nest family. As the AST of an arbitrary expression containing at least one NT^2 custom terminal (mainly `table` or `_`) is being built at compile-time, the AST construction function must ensure that expressions requiring non-nestable loop nests are separated by fetching the loop nest family associated with the current top-most AST node.

This is done during the template AST construction by splitting the AST into smaller ASTs of nodes with compatible tags. Two nodes have compatible tags if their code can be generated in a single, properly sized loop nest. If two nodes are incompatible, the most complex one is replaced by a temporary terminal reference pointing to the future result of the node evaluation. The actual sub-tree is then scheduled to be evaluated in advance, providing data to fill up the proxy reference in the original tree. As an example, figure 11 shows how the expression `A = B / sum(C+D)` is built and split into sub-ASTs handled by a single type of skeleton. `sum(C+D)` is a reduction loop nest (*i.e.* tagged as a `reduction_` function) and the division is performed with an elementwise loop nest (*i.e.* tied to the `divide` function and tagged as an `elementwise_` function). These loop nests may not be fused so the expression is rescheduled for the use of the `fold` skeleton followed by a `transform` skeleton.

The split ASTs are logically chained by the extra temporary variable inserted in the right-hand side of the first AST and as the left-hand size of the second. The life-cycle management of this temporary is handled by a C++ shared pointer and ensures that the data computed when crossing the AST barrier lives long enough. Notice that, as the `C+D` AST is an elementwise operation, it stays nested inside the `sum` node. NT^2 then uses the

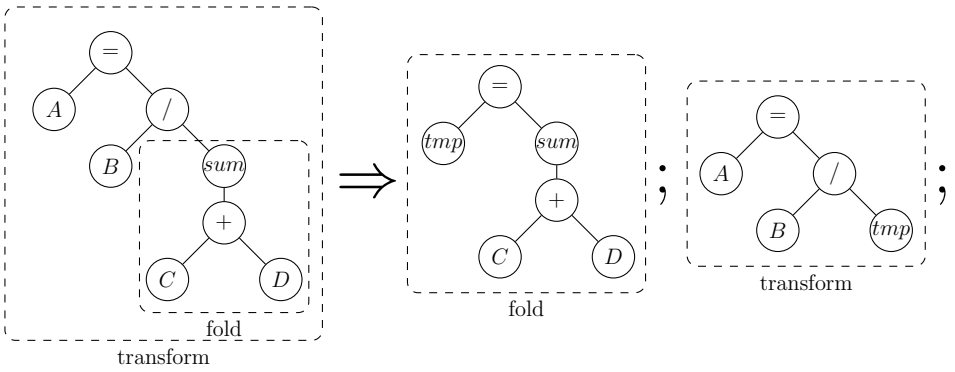


Figure 11: Parallel Skeletons scheduling process

nestability of parallel skeletons to call the SIMD and/or scalar version of each skeleton involved in a series of statements to recursively and hierarchically exploit the target hardware. At the end of the compilation, each NT^2 expression has been turned into the proper series of nested loop nests using combinations of OpenMP, SIMD and scalar code. Each of these skeletons is a NT^2 function object and thus can be specialized on a per-architecture basis.

In this section we introduced the core of NT^2, its expression framework. By providing a multi-pass evaluation process, NT^2 is able to transform and evaluate different scenarios with the proper parallel strategy for the code generation. Now, we will take a closer look at this process by describing each step in the evaluation of an NT^2 expression.

For example, consider the code generation of the `a = b+c` expression on an OpenMP+AVX system. `a = b+c` is first evaluated as a compile-time AST structured as:

```
expr< assign_, args< expr<terminal_, args<table<T,S> > >
                  , expr<plus_, args< expr<terminal_, args<table<T,S> >
                                    , expr<terminal_, args<table<T,S> >
    > > > >
```

Note the capture of the = node which allow NT^2 to optimize sub-matrix indexing using the code generation process. As every node in this expression is elementwise operations, `run` will select `transform` as the skeleton to use. The current architecture tag is `openmp_<avx_>` and the correct version of `transform` as shown in figure 12.

Figure 12: OpenMP `transform`

```
template<class LHS, class RHS, class Core>
void transform(LHS& a0, RHS& a1, int p, int s, openmp_<Core> const&)
{
  int bs = block_size();
  #pragma omp parallel firstprivate(bs)
  {
    nt2::functor<tag::transform_,Core> f;

    #pragma omp for schedule(dynamic) nowait
    for(int n=0;n<(s/bs);++n) f(a0,a1,p+n*bs,bs);

    #pragma omp single nowait
     if(s%bs) f(a0,a1,p+(s/bs)*bs,s%bs);
  }
}
```

As the OpenMP architecture is parametrized by the architecture tag of its inner core, the OpenMP `transform` only deals with laying out the required OpenMP structure around a call to the inner architecture `transform`

version. This recursive definition limits the amount of code written to handle architecture combinations as each skeleton implementation is only responsible for generating current architecture code. In this case, the OpenMP layer will take care of computing the optimal block size for the current architecture, perform a parallel loop nest over the nested transform call and handle the left-over data.

In a similar way, the AVX version of `transform` is in fact the common SIMD `transform` version, as Boost.SIMD allows us to use a single API for all our SIMD related code (see figure 13).

Figure 13: SIMD `transform`

```
template<class LHS, class RHS, class Core>
void transform(LHS& a0, RHS& a1, int p, int s, simd_ const&)
{
  typedef boost::simd::native<typename LHS::value_type> vector_type;
  int aligned_sz  = s & ~(vector_type::static_size-1);

  for(int it=p;it<p+aligned_sz; it+=vector_type::static_size)
    run( a0, it, run(a1, it, as_<target_type>()) );

  functor<transform_,cpu_> f;
  f(a0,a1,p+aligned_sz,sz-aligned_sz);
}
```

This version computes the slice of data which may be vectorized and calls the scalar version on the remaining data by using the scalar version of `transform`. Once completed, the code generated will automatically perform the required parallel operations. The call of `run` over either scalar or SIMD values is then deferred to Boost.SIMD for proper vectorization. The

compile-time aspect of this descent guarantees that the abstraction cost of the system is negligible.

3.5. Code generation considerations

3.5.1. Boost.SIMD *dependency*

NT^2 vectorization support is provided by the Boost.SIMD [31, 32] library. Boost.SIMD provides a high level abstraction to handle specific SIMD register types. Its wrapper class, `pack`, automatically selects the best SIMD register type according to its scalar template type. In addition, the library provides a set of functions which operate on the `pack` class including classic C++ operators, constant generators and arithmetic/IEEE 754/reduction functions. NT^2 embeds Boost.SIMD thus making the implementation of its code generator clear of any SIMD implementation details. The implementation of Boost.SIMD relies on the *AA-DEMRAL* approach applied to SIMD extensions, making the interoperability of NT^2 and Boost.SIMD straightforward. The Standard Boost.SIMD interface eases the integration of portable software components inside NT^2 . For example, NT^2 can then automatically vectorize TR1 polymorphic callable objects [33] that are passed to functions like `bsxfun`.

3.5.2. Importance of inlining

Function inlining is a important aspect of the implementation of such a library. The use of generative programming in such a context results in a significant amount of nested function calls. Non-inlined functions may generate countless function calls rendering the use of recursive definitions useless in a high-performance setting. NT^2 solves this problem by forcing all

skeleton related functions and all low level SIMD and scalar functions to be inlined.

3.5.3. Compilation time

Compilation time of heavily templated C++ code is often cited as a limit to the extensive use of such techniques. It is no different with NT^2 . However, two points must be considered. Firstly, the additional architecture-based dispatching is done early in the function resolution process, thus only a small subset of possible valid targets must be considered. The main cause of high compilation time is often due to compilers having to keep track of all already instantiated template types as long as a translation unit is not done compiling. Slicing NT^2 into separate translation units often mitigates this. How the compiler deduces types is another cause of the high compilation time. For example, NT^2 compiles up to 20% faster on compilers with compliant implementations of `decltype` and `auto`. The fact that NT^2 mimics the MATLAB API allows developers to design and debug using MATLAB , thus the bulk of the development process is performed using MATLAB . Only the final stable and correct version of the algorithm is ported to NT^2 , meaning the the effect of the long compilation time is minimized.

4. Experimental Results

In this section the execution time of various benchmarks are presented to give an idea of the performance attainable with NT^2 in different scenarios. The first benchmarks, inspired by the Armadillo benchmarks suite, assess the efficiency of the basic components of the library: the expression template engine using `Boost.Proto` and the efficiency of the BLAS and LA-

PACK bindings. Three more complex application kernels are also evaluated to demonstrate how NT^2 performs in realistic conditions. All benchmarks were run over thousands of executions from which the median execution time is taken. Where possible, results are compared with an equivalent kernel implemented using a selection of similar libraries or with direct calls to the underlying runtime when other libraries where unable to provide the required support (special mathematical functions, handling of small integers, or advanced control structures). Two different machines have been used for these performance benchmarks:

- **Sandy**, a Intel Core SandyBridge processor with 4 hyper-threaded cores, 8GB of RAM and a 8MB L3 cache. Code is compiled using g++-4.7 using AVX and/or OpenMP version 3.1;
- **Mini-Titan** composed of 2 sockets of Intel Core Westmere processors with 6 cores coupled with a NVIDIA Tesla C2075, 2x24GB of RAM and a 12MB L3 Cache. Code is compiled using g++-4.7 using SSE4.2 and/or OpenMP version 3.1.

In the following section, **FULL** benchmarks correspond to 12 threads running on **Mini-Titan** (12 physical cores) and 8 threads running on **Sandy** (4 physical cores with hyperthreading). **HALF** benchmarks are run with 6 threads on **Mini-Titan** (6 physical cores) and 4 threads on **Sandy** (4 physical cores).

4.1. Basic Elementwise operations

This benchmark evaluates the quality of code generation of NT^2 Expression Template engine by computing a series of elementwise operations on a

container of $n \times n$ elements:

```
R = 0.1f*A + 0.2f*B + 0.3f*C
```

Results, in CPU clock cycles per computed element, are given in table 14. We see that NT^2 performs comparable with other of SIMD enabled libraries such as Armadillo, Blaze and Eigen.

Size	Armadillo	Blaze	Eigen	MTL	uBlas	NT^2
50^2	5.2	3.1	2.1	25.2	14.4	1.5
1000^2	4.9	3.4	2.7	21.9	12.9	3.3

Figure 14: Elementwise benchmark using SSE 4.2 on **Mini-Titan**

4.2. BLAS operations

This benchmark evaluates the efficiency of the BLAS binding of NT^2 by performing a chain of three matrix-matrix products of decreasing size where the outer product is performed with the results of two smaller inner products:

```
Q = mtimes( mtimes(A,B), mtimes(C,D) );
```

Results, in cycles per computed element, are given in table 15. We see that NT^2 performance are comparable to other libraries. Armadillo exhibits the best performance due to its matrix-matrix product reordering phase.

4.3. LAPACK operations

These benchmarks assess the quality of the LAPACK binding by benchmarking a call to a linear system solver based on the GESV kernel and bound to the MATLAB like function `linsolve`. The code executed is:

Scale	Armadillo	Blaze	Eigen	MTL	uBlas	NT^2
100×20	59.0	154.2	88.40	126.7	77.82	79.11
2000×400	91.9	204.1	216.6	211.2	172.8	177.4

Figure 15: GEMM kernel benchmarks using MKL on **Mini-Titan**

```
X = linsolve(A,B);
```

Table 16 show the GFLOPS rate attained by using either direct C++ calls to LAPACK and MAGMA and to the corresponding NT^2 code. Results shows that the overhead versus the direct call to the LAPACK or MAGMA version of the kernel is not measured within a statistical certainty.

Scale	C LAPACK	C MAGMA	NT^2 LAPACK	NT^2 MAGMA
1024×1024	85.2	85.2	83.1	85.1
2048×2048	350.7	235.8	348.2	236.0
12000×1200	735.7	1299.0	734.1	1300.1

Figure 16: GESV kernel benchmarks on **Mini-Titan**

4.4. Black & Scholes options pricing

The Black & Scholes algorithm represents a mathematical model which gives a theoretical estimate of the price of European-style options. The code is defined in figure 17:

The Black & Scholes algorithm involves multiple high latency and high register count operations. The SIMD versions of `log`, `exp` and `normcdf` requires the evaluation of polynomials followed by a refinement step that consumes many registers.

Figure 17: Black & Scholes NT^2 implementation

```
table<float> blackscholes ( table<float> const& S,table<float> const& X
                          , table<float> const& T,table<float> const& r
                          , table<float> const& v
                          )
{
  table<float> d  = sqrt(T);
  table<float> d1 = log(S/X)+(fma(sqr(v),0.5f,r)*T)/(v*d);
  table<float> d2 = fma(-v,d,d1);

  return S*normcdf(d1)-X*exp(-r*T)*normcdf(d2);
}
```

This algorithm is also a good candidate for loop fusion. Using tie, the code can be rewritten as in figure 18.

Results shown in figure 19 demonstrates that our SIMD implementation hits roughly 65% of the peak speed-up in SIMD due to the number of spilled registers. The multi-threaded versions display almost linear speed-up. The difference between the regular and fusion code is roughly a factor of 2 for the multi-threaded version. This is due to the reduced number of barriers between statements and the increased locality.

4.5. Julia Mandelbrot set

The Julia Mandelbrot computation is a well known algorithm featuring load balancing issues: each iteration has a constant duration, but the number of iterations varies for each point. Like Black and Scholes, Mandelbrot is compute bound, but differs in two regards:

- Complex arithmetic is required. To ensure performance the code should

Figure 18: Black & Scholes NT^2 implementation with loop fusion

```
table<float> blackscholes ( table<float> const& S,table<float> const& X
                          , table<float> const& T,table<float> const& r
                          , table<float> const& v
                          )
{
  table<float> d, d1, d2, r;

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

  return r;
}
```

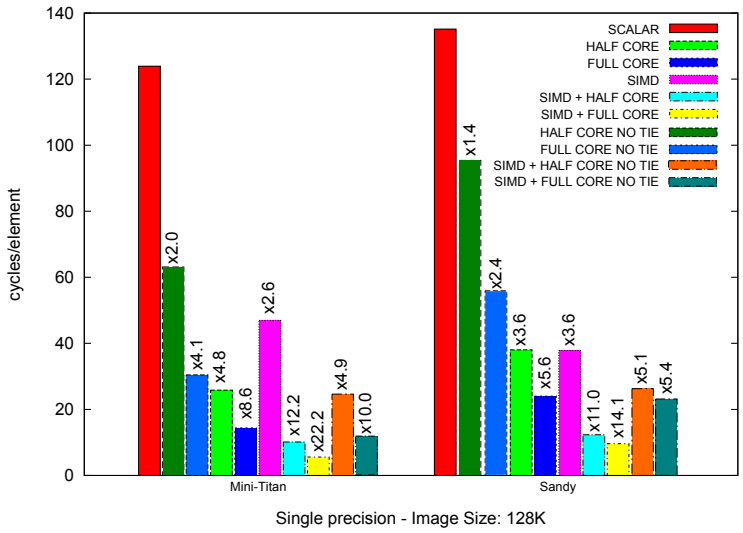


Figure 19: Black&Scholes Results in single precision

avoid temporary results for complex multiplication and division;

- It is composed of low-latency instructions (at most 5 cycles for multiplications) while Black and Scholes is dominated by longer latency instructions like square root, log and exponential.

To ensure proper parallelization, the actual NT^2 code relies on its implementation of the MATLAB function `bsxfun` that applies a given elementwise function object to every elements of a set of input tables. Contrary to other libraries, the NT^2 version of `bsxfun` relies on the fact that NT^2 can vectorize the code of any polymorphic callable object, *i.e.* a function object with a template function call operator. The NT^2 implementation is given as:

```
res = bsxfun( julia(), linspace(-1.,1,100), trans(linspace(-1.,1,100)) );
```

Figures 20 and 21 illustrate how the NT^2 single precision implementation of Julia Mandelbrot computation performs on both test machines. The speedups obtained on **Mini-Titan** are very close to linear with almost 80% efficiency. For **Sandy**, the efficiency decreases due to the generation of the input data that relies on SIMD integer support which is not available on AVX, thus it is emulated using a pair of SSE registers. If we remove this phase from the benchmark, the speed-ups are close to the expected ones. The integer support for AVX will be addressed in the future to avoid this loss of performance.

4.6. Sigma-Delta Motion detection

Sigma Delta is one of the most efficient mono-modal algorithms for motion detection [34]. It is composed of point-wise operations with two double if-then-else patterns. Its SIMD implementation is not straightforward, as the

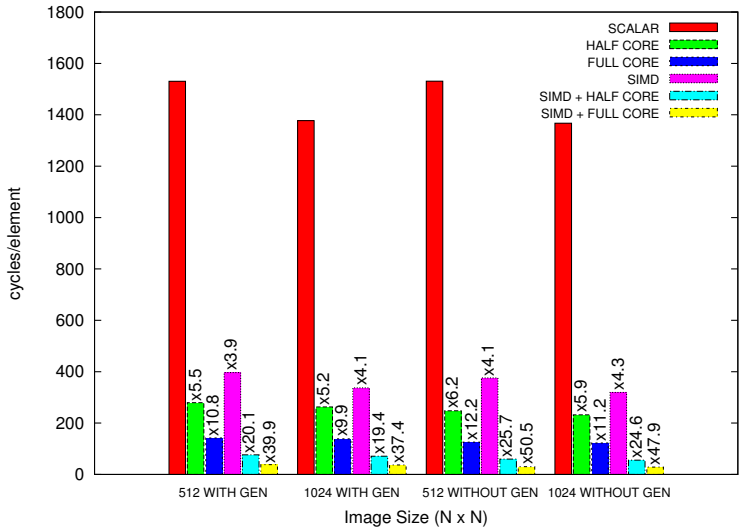


Figure 20: Mandelbrot Results in single precision on **Mini-Titan**

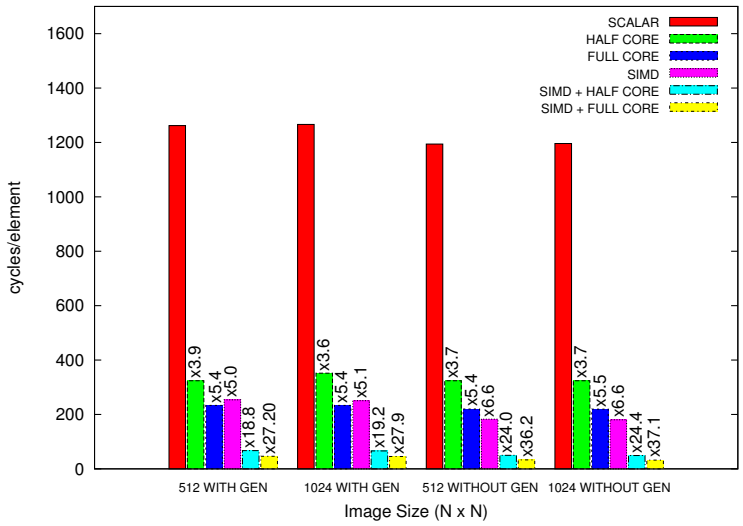


Figure 21: Mandelbrot Results in single precision on **Sandy**

multiplication and the absolute difference must be done using 16-bit data to get proper results. The solution is either converting 8-bit data to temporary 16-bit data or use saturated arithmetic. In the first case, we reduce the expected parallelism by a factor of two. In the second case, we keep the same expected parallelism but we use more complex operations. Its low arithmetic intensity means that this algorithm is memory bound. On the other hand, NT^2 provides support for integer types that is not a common feature available in other libraries. The NT^2 implementation of Sigma Delta in 8-bit unsigned integers using saturated arithmetic is given in figure 22:

Figure 22: Sigma-Delta NT^2 implementation

```
background = selinc( background < frame
                   , seldec( background > frame, background )
                   );

diff     = max(background, frame) - min(background, frame);
sigma3   = muls(diff,uint8_t(3));

variance = if_else( diff != uint8_t(0)
                  , selinc( variance < sigma3
                          , seldec(variance > sigma3, variance)
                          )
                  , variance
                  );

detected = if_zero_else_one( diff < variance );
```

As the algorithm is designed to work with unsigned 8-bit integers, the code cannot take advantage of AVX and thus has only been tested on **Mini- Titan** (see figure 23). With many load and store operations, strong scaling of the

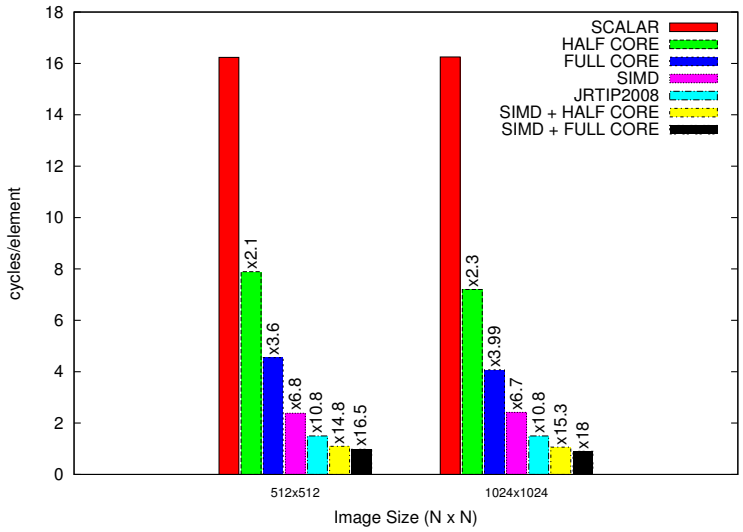


Figure 23: Sigma Delta Results

algorithm may not be preserved. When SSE is enabled, both versions (single-threaded-and multi-threaded) of the code increase their efficiency until the limits of memory bandwidth are reached. The SIMD only version is one cycle per element slower than the handwritten optimized one. This loss comes from very fine grain optimizations introduced in the code. Typically, the difference image (*i.e.* the different between two consecutive frames) does not need to be stored when working with an outer loop on the current frame being processed (C version). The Sigma-Delta implementation shows that the code generation engine of NT^2 leads to a proper optimized version of the application.

5. Related Works

We consider two kinds of related work: actual C++ scientific computing built around expression templates or other generative programming principles and external code generators designed for high performance computing.

5.1. Parallel C++ scientific computing libraries

Armadillo [16] is a C++ open source library for scientific computing developed at NICTA in Brisbane. It shares a common goal with NT^2 : providing an API as close as possible to MATLAB . Armadillo uses a classical Expression Template (ET) implementation designed from scratch and tailored to the actual need of the library. Optimization for elementary expressions is done using SSEx and AVX instructions sets. Armadillo uses BLAS for matrix multiplication, meaning the speed is dependent on the implementation of BLAS which is possibly multi-threaded.

Blaze [15] is a high-performance C++ math library for dense and sparse arithmetic. Blaze implements the so-called "Smart" Expression Templates which uses standard ET approaches for optimization of expression evaluation, and integration of performance optimized BLAS. This system is similar in principle to the optimization pass of NT^2 expression construction and handles similar use cases.

Eigen [35] is a header-only C++ library developed by Guennebaud et al. Started as a sub-project to KDE, Eigen3, the current major version, provides classes for many forms of matrices, vectors, arrays and decompositions. It integrates SIMD vectorization while exploiting knowledge about fixed-size matrices. It implements standard unrolling and blocking techniques for bet-

ter cache and register reuse in Linear Algebra kernels.

MTL[4] [36] is a generic library developed by Peter Gottschling and Andrew Lumsdaine for linear algebra operations on matrices and vectors. It supports BLAS-like operations with an intuitive interface but its main focus is on sparse matrices and vector arithmetic for simulation software. The library use Expression Templates at a lower scale than most tools, as it is restricted to handling combinations of kernels. In addition to the performance demands, MTL4 hides all this code complexity from the user who writes applications in natural mathematical notation.

Open CV [37] (Open Source Computer Vision) is a real-time computer vision library originally developed by Intel and now supported by Willow Garage enterprise since 2008. Written in C++, the library proposes a large set of functions able to process raw images and video streams. From basic filtering to facial recognition, OpenCV covers a wide range of functionality. The library is optimized with SSE2 and Intel TBB. It can also exploit of Intel IPP, if available.

5.2. Other code generation systems

Delite [38] is a compiler framework and runtime for parallel embedded domain-specific languages from Stanford University PPL. Delite's goal is to enable the rapid construction of high performance, highly productive DSLs. Delite has the ability to compile a user-defined language and provides several facilities like built-in parallel execution patterns, optimizers for parallel code, code generators for Scala, C++ and CUDA and a heterogeneous run-

[4]Matrix Template Library

time for executing DSLs. NT^2 and its use of *AA-DEMRAL* could be seen as C++ equivalent to Delite in the sense that it is built on similar sub-systems. The main difference between Delite and NT^2 is the fact that Delite has access to information like variable names that NT^2 can not currently access.

DESOLA [5] is a linear algebra library developed to explore multi-stage programming as a way to build active libraries [39]. The idea is to delay library call executions and generate optimized code at runtime, contrary to NT^2 which maximizes compile-time code generation and optimization. Calls made to the library are used to build a *recipe* for the delayed computation. When the execution is finally forced by the need for a result, the recipe will often represent a complex composition of primitive calls. In order to improve performance over a conventional library, it is important that the generated code should be executed faster than a statically generated counterpart in a conventional library.

TOM is a language extension designed to manipulate tree structures and XML documents [40]. It provides pattern matching facilities to inspect objects and retrieve values in C, Java, Python, C++ or C#. Tom is a language extension which adds new matching primitives to C and Java: `%match`. This construct is similar to the match primitive found in functional languages. The patterns are used to discriminate and retrieve information from an algebraic data structure. Therefore, Tom is a good language for programming by pattern matching. This techniques is similar in practice to what `Boost.Proto` transforms can achieve within C++ itself, thus enabling sim-

[5] Delayed Evaluation Self Optimizing Linear Algebra

ilar kind of efficient rewrites.

Conclusion

A new methodology for implementing more flexible scientific computing software: the *AA-DEMRAL* methodology is presented in this paper. We demonstrate that systematized applications of design strategies like Generative Programming helps building architecture aware software with a very high level of abstraction and high efficiency. It was shown that, like any general-purpose language, *DSEL*s require information about the hardware system to be efficient and that such information may be carried over by the very same C++ idioms that are classically used to design those *DSEL*s .

NT^2's design takes advantage of this information to deliver a high level of performance while keeping a high level of abstraction. It allows portability over various architectures and provides a systematic way of implementing new architectural supports. Our benchmarks show that with both simple and complex tasks, NT^2 is able to deliver performance within the range of state of the art implementation. As an open-source project, NT^2 is publicly available on `http://www.github.com/MetaScale/nt2`.

Current work includes:

- Generalized support for distributed and shared memory systems by using an asynchronous runtime back-end like Charm++ [41] or HPX [42];
- Support for multi-stage programming [43]. This will allow NT^2 to target system like DSPs or OpenCL based Altera reconfigurable systems on which no C++ compiler is natively available;

- Explore benefits of embedding a meta-programmed subset of the polyhedral model [44] inside the NT^2 skeleton system to refine the combination of loop nests that can be generated.

On a larger scope, we want to explore a deeper exploitation of information about latent parallelism in more complex kernels. Such information, going beyond the simple data-parallel skeletons shown previously, may help to redesign, in a generative way, frameworks like LAPACK without having to rethink the algorithm implementation for every new architecture and by maximizing inter-procedural optimizations across kernels and architectures.

Acknowledgments

the authors wish to thank Ian Masliah and Antoine Tran Tan for their technical assistance in setting up the benchmarks, Daniel Etiemble for his comments and Alan Kelly for his extensive review of the paper.

References

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

[2] T. Cramer, D. Schmidl, M. Klemm, 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, 2012, pp. 38–44.

[3] J. Derrick, N. Walkinshaw, T. Arts, C. B. Earle, F. Cesarini, L.-Å. Fredlund, V. M. Gulías, J. Hughes, S. J. Thompson, Property-Based Testing - The ProTest Project, in: FMCO, 2009, pp. 250–271.

[4] L. Tratt, Model transformations and tool integration, Journal of Software and Systems Modelling 4 (2005) 112–122.

[5] P. Hudak, Building domain-specific embedded languages, ACM Comput. Surv. 28 (1996).

[6] K. Czarnecki, J. T. O'Donnell, J. Striegnitz, W. Taha, DSL Implementation in MetaOCaml, Template Haskell, and C++, in: Domain-Specific Program Generation, 2003, pp. 51–72.

[7] K. Czarnecki, U. W. Eisenecker, R. Glück, D. Vandevoorde, T. L. Veldhuizen, Generative Programming and Active Libraries, in: International Seminar on Generic Programming, Dagstuhl Castle, Germany, April 27 - May 1, 1998, Selected Papers, 1998, pp. 25–39.

[8] T. L. Veldhuizen, D. Gannon, Active Libraries: Rethinking the roles of compilers and libraries, in: In Proceedings of the SIAM Workshop on Object Oriented Methods for Inter-operable Scientific and Engineering Computing (OO'98, SIAM Press, 1998.

[9] D. Abrahams, A. Gurtovoy, C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond (C++ in Depth Series), Addison-Wesley Professional, 2004.

[10] E. Gamma, R. Helm, R. Johnson, J. Vlissides, Design patterns: elements of reusable object-oriented software, Pearson Education, 1994.

[11] T. L. Veldhuizen, Expression templates, C++ Report 7 (1995) 26–31. Reprinted in C++ Gems, ed. Stanley Lippman.

[12] D. Vandevoorde, N. M. Josuttis, C++ Templates, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[13] D. Spinellis, Notable design patterns for domain-specific languages, Journal of Systems and Software 56 (2001) 91 – 99.

[14] E. Niebler, Proto: A compiler construction toolkit for DSELs, in: Proceedings of the 2007 Symposium on Library-Centric Software Design, ACM, 2007, pp. 42–51.

[15] K. Iglberger, G. Hager, J. Treibig, U. Rüde, Expression Templates Revisited: A Performance Analysis of Current Methodologies, SIAM J. Scientific Computing 34 (2012).

[16] C. Sanderson, Armadillo: An Open Source C++ Linear Algebra Library for Fast Prototyping and Computationally Intensive Experiments, Technical Report, NICTA, Australia, 2010.

[17] K. Czarnecki, U. W. Eisenecker, Components and Generative Programming, in: ESEC / SIGSOFT FSE, 1999, pp. 2–19.

[18] K. Czarnecki, U. W. Eisenecker, Generative programming - methods, tools and applications, Addison-Wesley, 2000.

[19] OpenMP Architecture Review Board, OpenMP Application Program Interface Version 3.0, 2008. URL: `http://www.openmp.org/mp-documents/spec30.pdf`.

[20] J. Reinders, Intel threading building blocks - outfitting C++ for multi-core processor parallelism., O'Reilly, 2007.

[21] S. Tomov, J. Dongarra, M. Baboulin, Towards dense linear algebra for hybrid GPU accelerated manycore systems, Parallel Computing 36 (2010) 232–240.

[22] N. C. Myers, Traits: a new and useful template technique, C++ Report (1995). URL: http://www.cantrip.org/traits.html.

[23] D. Gregor, J. Järvi, J. Siek, B. Stroustrup, G. Dos Reis, A. Lumsdaine, Concepts: linguistic support for generic programming in C++, in: Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications, OOPSLA '06, ACM, New York, NY, USA, 2006, pp. 291–310. URL: http://doi.acm.org/10.1145/1167473.1167499. doi:10.1145/1167473.1167499.

[24] M. Cole, Bringing skeletons out of the closet: a pragmatic manifesto for skeletal parallel programming, Parallel computing 30 (2004) 389–406.

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

[26] M. Aldinucci, M. Danelutto, P. Dazzi, Muskel: an expandable skeleton environment, Scalable Computing: Practice and Experience 8 (2001).

[27] H. Kuchen, A skeleton library, Springer, 2002.

[28] M. Aldinucci, M. Danelutto, J. Dünnweber, Optimization Techniques for Implementing Parallel Skeletons in Grid Environments, in: S. Gorlatch (Ed.), Proc. of CMPP: Intl. Workshop on Constructive Methods for Parallel Programming, Universitat Munster, Germany, Stirling, Scotland, UK, 2004, pp. 35–47.

[29] K. Emoto, K. Matsuzaki, Z. Hu, M. Takeichi, Domain-Specific Optimization Strategy for Skeleton Programs, in: A.-M. Kermarrec, L. Bougé, T. Priol (Eds.), Euro-Par 2007 Parallel Processing, volume 4641 of *Lecture Notes in Computer Science*, Springer Berlin Heidelberg, 2007, pp. 705–714. URL: http://dx.doi.org/10.1007/978-3-540-74466-5_74. doi:10.1007/978-3-540-74466-5_74.

[30] P. Ciechanowicz, H. Kuchen, Enhancing Muesli's data parallel skeletons for multi-core computer architectures, in: High Performance Computing and Communications (HPCC), 2010 12th IEEE International Conference on, IEEE, 2010, pp. 108–113.

[31] P. Estérie, M. Gaunard, J. Falcou, J.-T. Lapresté, B. Rozoy, Boost.SIMD: generic programming for portable SIMDization, in: Proceedings of the 21st international conference on Parallel architectures and compilation techniques, ACM, 2012, pp. 431–432.

[32] P. Esterie, M. Gaunard, J. Falcou, et al., Exploiting Multimedia Extensions in C++: A Portable Approach, Computing in Science & Engineering 14 (2012) 72–77.

[33] C++ Standard, The Callable Concept, `http://en.cppreference.com/w/cpp/concept/Callable`, 2014.

[34] L. Lacassagne, A. Manzanera, J. Denoulet, A. Mérigot, High Performance Motion Detection: Some trends toward new embedded architectures for vision systems, Journal of Real Time Image Processing (2008) 127–148. doi:10.1007/s11554-008-0096-7.

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

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

[37] G. Bradski, A. Kaehler, Learning OpenCV: Computer vision with the OpenCV library, O'Reilly Media, Inc., 2008.

[38] K. J. Brown, A. K. Sujeeth, H. J. Lee, T. Rompf, H. Chafi, M. Odersky, K. Olukotun, A heterogeneous parallel framework for domain-specific languages, in: Parallel Architectures and Compilation Techniques (PACT), 2011 International Conference on, IEEE, 2011, pp. 89–100.

[39] F. P. Russell, M. R. Mellor, P. H. Kelly, O. Beckmann, DESOLA: An active linear algebra library using delayed evaluation and runtime code generation, Science of Computer Programming 76 (2011) 227 – 242.

[40] P.-E. Moreau, C. Ringeissen, M. Vittek, A pattern matching compiler for multiple target languages, in: Proceedings of the 12th international conference on Compiler construction, CC'03, Springer-Verlag, Berlin, Heidelberg, 2003, pp. 61–76.

[41] L. V. Kale, S. Krishnan, CHARM++: a portable concurrent object oriented system based on C++, volume 28-10, ACM, 1993.

[42] C. Dekate, M. Anderson, M. Brodowicz, H. Kaiser, B. Adelstein-Lelbach, T. Sterling, Improving the scalability of parallel N-body applications with an event-driven constraint-based execution model, Int. J. High Perform. Comput. Appl. 26 (2012) 319–332.

[43] J. Eckhardt, R. Kaiabachev, E. Pasalic, K. Swadi, W. Taha, Implicitly heterogeneous multi-stage programming, New Gen. Comput. 25 (2007) 305–336.

[44] C. Bastoul, P. Feautrier, Adjusting a program transformation for legality, Parallel processing letters 15 (2005) 3–17.