# Multi-Architectural Support : A Generic and Generative Approach

Pierre Estérie

UNIVERSITÉ PARIS SUD
Comprendre le monde, construire l'avenir®

LABORATOIRE DE RECHERCHE EN INFORMATIQUE

**UNIVERSITY OF PARIS SUD XI**

**DOCTORAL SCHOOL**
**ECOLE DOCTORALE D'INFORMATIQUE**
**DE PARIS SUD XI (EDIPS)**

# P H D  T H E S I S

to obtain the title of

**PhD of Science**

of the University of Paris Sud XI
**Specialty : Computer Science**

Defended by
Pierre Esterie

# Multi-Architectural Support: A Generic and Generative Approach

Thesis Advisor: Brigitte Rozoy
Thesis Co-Advisor: Joel Falcou

prepared at Laboratoire de Recherche en Informatique
Parsys Team

defended on June 20, 2014

**Jury :**

| | | | |
|---|---|---|---|
| Philippe Clauss, | *Reviewer* | - | University of Strasbourg (ICube) |
| Lawrence Rauchwerger, | *Reviewer* | - | Texas A&M University (Parasol) |
| Sylvain Conchon, | *Examinator* | - | University of Paris Sud XI (LRI) |
| Joel Falcou, | *Examinator* | - | University of Paris Sud XI (LRI) |
| Sylvain Jubertie, | *Examinator* | - | University of Orléans (LIFO) |
| Brigitte Rozoy, | *Advisor* | - | University of Paris Sud XI (LRI) |

## Acknowledgments

At the end of this thesis, I want to thank my advisor, Professor Brigitte Rozoy, for her trust and kindness during this work. It helped me to achieve this thesis in a good working atmosphere.

My most heartfelt thanks go to my co-advisor, Assistant Professor Joel Falcou. Through those years, I learnt a lot by working with him and he also supported me at every moment during this work.

I also thank Philippe Clauss and Lawrence Rauchwerger for giving me the honor of reviewing this typescript and participating to the jury.

Sylvain Conchon presided the jury and Sylvain Jubertie accepted to join the jury, I sincerely thank them.

Now, I will end these acknowledgments in french.

Je tiens aussi à remercier sincèrement Daniel Etiemble, Mathias Gaunard, Lionel Lacassagne ainsi que les membres de l'équipe Parsys (anciennement Parall/Archi) et du LRI que j'ai cotoyés durant ces années pour leur soutien, leur parole avisée ainsi que tous les moments partagés avec eux : Adrien, Riadh, Mikolaj, Amina, Allah, Amal, Laurent, Sebastian, Yushan, Ian, Florence, Lenaïc, Antoine, Khaled, Sylvain, Sarah, Stéphanie, Cécile.

Pour finir, j'adresse mes plus forts remerciements à mes amis, mes proches ainsi qu'à mes parents et ma famille pour m'avoir soutenu et d'avoir été présents quelles que soient les circonstances.

## Abstract

The constant increasing need for computing power has pushed the development of parallel architectures. Scientific computing relies on the performance of such architectures to produce scientific results. Programming efficient applications that takes advantage of these computing systems remains a non trivial task.

In this thesis, we present a new methodology to design architecture aware software: the *AA-DEMRAL* methodology. This methodology aims at simplifying the development of parallel programming tools with multi-architectural support through a generic and generative approach.

We then present three high level programming tools that rely on this approach. First, we introduce the Boost.Dispatch library that provides a way to develop software based on the *AA-DEMRAL* methodology. The Boost.Dispatch library is a C++ generic framework for architecture aware function dispatching. Then, we present two C++ template libraries implemented as Architecture Aware *DSEL*s which assess the *AA-DEMRAL* methodology through the use of Boost.Dispatch: Boost.SIMD, that provides a high level API for SIMD extensions and $NT^2$ , which propose a Matlab like interface with support for multi-core and SIMD based systems. We assess the performance of these libraries and the validity of our new methodology through benchmarks.


**Keywords:** Parallel architectures, *DSEL*s , Active Library, Generative Programming, Generic Programming, C++ .

# Contents


**1 Introduction** **1**

1.1 Why high performance computing matters? . . . 1
1.2 Fast programming of fast applications? . . . 2
1.3 Objectives . . . 3
1.3.1 Architecture aware software designing . . . 3
1.3.2 High level tools for Scientific Computing . . . 4
1.3.3 Our contribution . . . 4

**2 From Architectures to Applications** **7**

2.1 The Wide Architecture Landscape . . . 7
2.1.1 SIMD extensions . . . 8
2.1.2 From single core to multi-core . . . 9
2.1.3 Accelerators . . . 10
2.1.4 Distributed memory systems . . . 12
2.1.5 Conclusion . . . 12
2.2 Programming the Landscape . . . 13
2.2.1 Low Level Parallel Tools . . . 13
2.2.2 Domain Specific Libraries for Scientific Computing . . . 20
2.2.3 Domain Specific Languages . . . 23
2.2.4 Domain Specific Embedded Language approach . . . 24
2.2.5 Conclusion . . . 25
2.3 Proposed Approach . . . 25

**3 A Generic and Generative Programming Approach for DSL** **27**

3.1 Software context . . . 27
3.1.1 The configuration space approach . . . 28
3.1.2 Library level exploration . . . 29
3.2 Programming techniques . . . 30
3.2.1 Template Meta-Programming . . . 30
3.2.2 Expression Templates . . . 30
3.2.3 The Boost.Proto library . . . 32
3.2.4 Conclusion . . . 36
3.3 The *DEMRAL* Methodology . . . 36
3.3.1 *DSEL*s design considerations . . . 36
3.3.2 From *DSEL*s to Architecture Aware *DSEL* . . . 37
3.4 Conclusion . . . 39

**4 The Boost Dispatch Library** **41**
4.1 Challenges . . . 41
4.1.1 Regular C++ function overloading . . . 42
4.1.2 Free function dispatching using SFINAE . . . 43
4.1.3 The Tag Dispatching technique . . . 44
4.1.4 Concept based overloading . . . 45
4.1.5 Conclusion . . . 45
4.2 The Boost.Dispatch Library . . . 46
4.2.1 The *AA-DEMRAL* methodology in Boost.Dispatch . . . 46
4.2.2 The Hierarchy Concept . . . 47
4.2.3 Compile Time Hierarchy Deduction . . . 48
4.2.4 Built-in Hierarchies . . . 49
4.2.5 Common API . . . 56
4.3 Conclusion . . . 59

**5 The Boost SIMD Library** **61**
5.1 Hardware context and software challenges . . . 62
5.2 The Boost.SIMD Library . . . 65
5.2.1 SIMD register abstraction . . . 65
5.2.2 Predicates abstraction . . . 69
5.2.3 Range and Tuple interface . . . 69
5.2.4 Supported functions . . . 71
5.2.5 Shuffling operations . . . 71
5.3 C++ Standard integration . . . 74
5.3.1 Aligned allocator . . . 74
5.3.2 SIMD Iterator . . . 75
5.3.3 SIMD Algorithms . . . 77
5.4 Case Analysis: Generic SIMD code generation . . . 77
5.4.1 Scalar version of the `dot` function . . . 77
5.4.2 Transition from scalar to SIMD code . . . 78
5.4.3 Building a SIMD loop nest . . . 78
5.4.4 Preparing the data . . . 79
5.4.5 Resulting code generation . . . 81
5.4.6 Choosing SIMD extensions at runtime . . . 82
5.5 Implementation . . . 85
5.5.1 Function Dispatching . . . 85
5.5.2 AST Manipulation with Boost.Proto . . . 86
5.6 Implementation Discussion . . . 90
5.6.1 Function inlining and ABI issue . . . 90
5.6.2 Unrolling . . . 90
5.7 Benchmarks . . . 91
5.7.1 Basic Kernels . . . 91
5.7.2 Black and Scholes . . . 94
5.7.3 Sigma-Delta Motion Detection . . . 95

5.7.4 The Julia Mandelbrot Computation . . . 97
5.8 Conclusion . . . 99

**6 NT2: an Architecture Aware DSEL Framework 101**
6.1 The NT2 Programming Interface . . . 102
6.1.1 Basic API . . . 103
6.1.2 Indexing and data reshaping . . . 103
6.1.3 Linear Algebra support . . . 104
6.1.4 `table` settings . . . 104
6.1.5 Compile-time Expression Optimization . . . 105
6.1.6 Parallelism Handling . . . 106
6.2 Implementation . . . 106
6.2.1 Function compile-time descriptors . . . 107
6.2.2 Compile-time architecture description . . . 107
6.3 Expression Evaluation with NT2 . . . 111
6.4 Benchmarks . . . 112
6.4.1 Basic Kernels . . . 112
6.4.2 Black and Scholes . . . 114
6.4.3 Sigma Delta Motion Detection . . . 115
6.4.4 The Julia Mandelbrot Computation . . . 116
6.5 Conclusion . . . 118

**7 Conclusions and Perspectives 119**
7.1 Conclusions . . . 119
7.2 Perspectives . . . 120

**A Algorithms Description 123**
A.1 AXPY Kernel . . . 123
A.2 Black and Scholes . . . 123
A.3 Sigma-Delta Motion Detection . . . 123
A.4 The Julia Mandelbrot Computation . . . 125

**B Architectures Description 127**
B.1 Boost.SIMD Benchmark Architectures . . . 127
B.2 $NT^2$ Benchmark Architectures . . . 127

**C ISO C++ Standard Proposal 129**
C.0.1 Our proposal . . . 129
C.1 Impact On the Standard . . . 130
C.1.1 Standard Components . . . 130
C.2 Technical Specifications . . . 133
C.2.1 `pack<T,N>` class . . . 133
C.2.2 `logical<T>` class . . . 134
C.2.3 Operators overload for `pack<T,N>` . . . 134
C.2.4 Functions . . . 146

C.2.5 Traits and metafunctions . . . . . 154

**D Meta-Unroller** **155**

**Bibliography** **157**

# List of Figures


1.1 Flynn's taxonomy . . . 3

2.1 Principle of a SIMD computation unit . . . 8
2.2 Picture of a quad-core die . . . 9
2.3 CUDA principle . . . 11
2.4 A Xeon Phi PCI card . . . 11
2.5 A picture of Titan . . . 12

3.1 General principles of *Expression Templates* . . . 31
3.2 The *DEMRAL* methodology . . . 37
3.3 The *AA-DEMRAL* methodology . . . 38

4.1 Architecture information for dispatch of generic components . . . 47
4.2 The properties of the built-in scalar hierarchy . . . 51
4.3 `plus_` node example . . . 56

5.1 Load strategy for SOA . . . 70
5.2 Load strategy for AOS . . . 71
5.3 Permutation example . . . 72
5.4 $4 \times 4$ matrix transpose in SIMD . . . 73
5.5 Address alignment for SIMD iterators . . . 75
5.6 Shifted iterator . . . 76
5.7 Boost.Proto AST of `a + b*c` . . . 87
5.8 `plus` function with Boost.Proto based arguments . . . 88
5.9 `assign` function call on the LHS and RHS . . . 89
5.10 Results for Black and Scholes algorithm on Excalibur . . . 94
5.11 Results for Sigma-Delta algorithm on Excalibur . . . 96
5.12 Results for Julia Mandelbrot algorithm on Excalibur . . . 98

6.1 `table` with `interleaved_` and `deinterleaved_` data . . . 105
6.2 Feature set comparison between $NT^2$ and similar libraries . . . 106
6.3 Parallel Skeletons extraction process . . . 110
6.4 Elementwise benchmark using SSE 4.2 on **Mini-Titan** . . . 113
6.5 GEMM kernel benchmarks using MKL on **Mini-Titan** . . . 113
6.6 GESV kernel benchmarks on **Mini-Titan** . . . 113
6.7 Black&Scholes Results in single precision . . . 114
6.8 Sigma Delta Results . . . 115
6.9 Mandelbrot Results in single precision on **Mini-Titan** . . . 117
6.10 Mandelbrot Results in single precision on **Sandy** . . . 117

A.1 Motion detection with Sigma Delta . . . 124

A.2 Definition of the Julia-Mandelbrot set . . . . . 125
A.3 Illustration of the Julia-Mandelbrot set . . . . . 125

# List of Tables


5.1 SIMD extensions in modern processors . . . . . 63
5.2 SIMD types available . . . . . 64
5.3 Boost.SIMD vs handwritten SIMD code for the AXPY kernel in *GFlop/s* . 92
5.4 Boost.SIMD vs Autovectorizers for the DAXPY kernel in *GFlop/s* . . . . 92
5.5 Boost.SIMD vs Autovectorizers for the SAXPY kernel in *GFlop/s* . . . . . 93
5.6 Results for Sigma-Delta algorithm in *cpp* . . . . . 95

B.1 Processor details . . . . . 127

C.1 Functions on `pack` . . . . . 153
C.1 Functions on `pack` . . . . . 154

Chapter 1

# Introduction

Contents


**1.1 Why high performance computing matters? . . . . . 1**
**1.2 Fast programming of fast applications? . . . . . 2**
**1.3 Objectives . . . . . 3**
1.3.1 Architecture aware software designing . . . . . 3
1.3.2 High level tools for Scientific Computing . . . . . 4
1.3.3 Our contribution . . . . . 4


Since the first electronic programmable computer architecture was launched in 1943 (Colossus), computing power has been the main driving force in computer science as expressed by Moore's law. After decades of architectural improvements like superscalar architectures, caches, out of order execution and more, the manufacturers were facing the limits connected with power dissipation. It became a major issue and high frequencies were no longer a solution to the race for computational power. The alternative to higher frequency appeared at the start of the $21^{st}$ century with the first multi-core processors. These new architectural designs allowed to safely increase the computational power of a chip. Multi-core processors are now the standard architecture. Parallel computing is not a new topic from the $21^{st}$ century as parallel computers exist since the early 60's. However the birth of multi- core architectures has changed the programming issues. After the change to multi-core based solutions, every machine can now be considered as a parallel architecture with different level of parallelism available through cores and *Single Instruction Multiple Data* (SIMD) units.

## 1.1 Why high performance computing matters?

The race for computing power is a response to the need of new high-performance applications. As an example, sensors can generate a large amount of data with a constant increase in precision and quality over the past decade. Image processing, photography, video and audio are good examples of such an increase. The size of data sets involved in these applications is now substantial. As a consequence, softwares need to deal with high numbers of memory allocations. As an other example, scientific problems often perform an important amount of computations due to the complexity of algorithms. Furthermore, some of these applications are real

time based and must ensure the supply data at a fix rate. The temporal constraint is therefore very important for the developer and every architectural features must be taken into consideration.

Scientific simulations are also good "consumers" of computational power with complex algorithms and big data sets. The main goal here is to analyze a phenomenon with the outputs of the simulation. The simulation is mostly used on different data sets with variable parameters for the algorithm embedded in the simulation application. In this context, simulations can reach several hours of execution. For scientists, the time constraint is a limitation to their analysis of a problem. Scientific researchers then become dependent on the execution time of the simulation.

Developing a fast and reactive application in a reasonable amount of time now requires expertise in various fields of computer science. The scientific community has a huge range of specialized domains and presents an even wider range of dedicated scientific applications to solve new challenges.

The common point in the previous examples is the constant need for speedup in applications. To satisfy this requirement, some solutions are available. First, working on the algorithm to limit the number of operations is a start but this approach is not always relevant as some accuracy or strength problems can occur. An old solution was to wait for the next generation of processor but the correlation in the Moore's law between the number of transistors on a chip and the computing power has changed. The transistor density is still increasing due to the rise of new parallel architectures. Applications can now take advantage of new architectural features and improvements. Using a parallel architecture at the best of its computing capabilities requires parallel programming.

## 1.2 Fast programming of fast applications?

Developing large applications in a simple, fast and efficient way has always been an issue for software developers. This limitation comes from severals factors. First, the diversity of architectures slows down the optimization process of an application. Mickael J. Flynn introduced a synthetic classification [47] reflecting the different types of architecture. Figure 1.1 illustrates this classification. *Multiple Instruction Single Data* architectures are rare and exist only for very specific tasks. From *Single Instruction Single Data* machine to *Multiple Instruction Multiple Data* machines, this taxonomy is not reflecting completely the wide landscape of parallel architecture. Eric E. Johnson completed the Flynn's taxonomy [63] by classifying MIMD architectures according to their memory and communication mechanisms. Regarding these two taxonomies, developing an optimized and portable application is a tenacious and time consuming task. In addition, each computing system is not always available for the software development due to the cost of such a test farm.

Another factor is the increasing number of parallel programming tools available. From pure system oriented libraries like thread managers to domain oriented libraries or low level intrinsics calls, developers are confronted to a very difficult

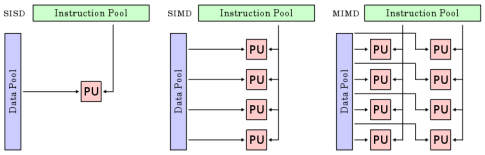



Figure 1.1: Flynn's taxonomy

choice when it comes to selecting the proper tools for their application.

The previous factors impact the source code of a application by increasing the verbosity of the programming style. Algorithms are then buried in several implementation details and the readability of the source code is decreased. To face the limitations of low level programming, domain oriented languages offer a high level interface tie to the corresponding domain. They are known as Domain Specific Languages (*DSL*s ) and their high expressiveness allows scientists to focus on their application challenges.

Computer science is a specific domain sitting near mathematic, physics, biology and countless others. The diversity of the scientific community comes with a lot of different programmer backgrounds. Some of them are programming experts while others have face the challenges of writing their own programs. Most of these programs are often suitable candidates for parallel speedups on modern computer architectures. Parallel programming requires a good knowledge of the different programming models and their dedicated frameworks. On top of that, architecture specific optimizations can be added which is a non trivial task for non expert programmers.

In this context, scientists are not on par with computer experts to write high performance applications. Therefore writing a fully parallel version of an application should not be part of their work. Their main focus should be designing new mathematical models or algorithms.

## 1.3 Objectives

Parallel architectures can provide computing power for scientific applications but the use of such architectures is a difficult task due to several aspects. The diversity of the hardware and the multiplicity of software solutions does not facilitate the development of applications. It is in this context that we present our work. It focuses on a new software design methodology for developing architecture aware tools that are able to provide expressiveness and performance.

### 1.3.1 Architecture aware software designing

Designing parallel software requires the use of multiple architectural features. The accessibility of such features is tied to different programming techniques. When

using these techniques, it then becomes difficult to design portable softwares that will be able to select architecture specific implementations. This multi-architectural approach also requires an extensible design of the software that will ease its maintenance. The design of such software requires a new methodology and developers need to have programming facilities to integrate this new type of approach.

### 1.3.2 High level tools for Scientific Computing

In addition to this new design methodology, scientific computing tools built on top of it must provide expressiveness to their users. This expressiveness can separate architectural implementation details from the original algorithm to give a high level programming style. This type of model is required to alleviate the user from non trivial architecture oriented programming. Thus, the performance of such tools must stay comparable to an original optimized version of the software. The code of such applications should be written once and just recompiled on different architecture.

### 1.3.3 Our contribution

We propose a methodology for designing software with multi-architectural support without loss of performance and expressiveness. In this thesis, we present this new approach and three programming tools that implement this new methodology. The typescript is organized as follow:

- **Chapter 2, From Architectures to Applications.** In this chapter, we present a quick overview of today's architectures with their multiple levels of parallelism. Then, we introduce the programming tools available for these architectures. From there, we detail domain specific libraries and Domain Specific Languages (DSL). For each of them, the state of the art focuses on their parallel features and expressiveness. We show that different approaches have been chosen to solve the performance/expressiveness issue. But a remaining challenge is to combine both of them inside a programming tool. At the end of this chapter, we conclude on the software design directions taken for the development of a multi-architecture parallel programming tool and we propose to focus our work on designing *DSEL*s in C++ .

- **Chapter 3, A Generic and Generative Programming Approach for DSL.** We first show the state of the art of the current practices for designing a *DSEL* in C++ , focusing especially on generic and generative programming techniques and the expression template technique. We then discuss the challenges behind the design of a Domain Specific Embedded Language in C++ and the DEMRAL methodology. For this purpose we introduce the need for genericity in such a context. We finally present our approach : the Architecture Aware DEMRAL methodology (*AA-DEMRAL* ) which aims at making the design of *DSEL*s aware of architectural information.

- **Chapter 4, The Boost.Dispatch Library.** In this chapter, we present a library that helps the development of software based on the *AA-DEMRAL* methodology: the Boost.Dispatch library. Function dispatch is a C++ technique that aims at selecting the best implementation of a function according to its argument types. In the context of parallel programming, specific architecture details need to be injected during the selection process of a function implementation. The Boost.Dispatch library is presented in this chapter as an architecture aware function dispatching library like the introduced concept in chapter 3. First, we present the challenges behind such a library. Then, go through an example to illustrate a typical use case of Boost.Dispatch.

- **Chapter 5, The Boost.SIMD Library.** This chapter presents the usefulness of the *AA-DEMRAL* methodology and Boost.Dispatch is used in this context to build a high level programming tool. We present Boost.SIMD, a C++ template library that aims at simplifying the programming of SIMD extensions. After describing the hardware context in which this library takes place, we discuss the challenges of such a tool. The API of the library is detailed and the library is illustrated with a case analysis. Then, implementation details are presented . We finally assess its performances.

- **Chapter 6, NT2: an Architecture Aware DSEL Framework.** After presenting Boost.SIMD, we introduce $NT^2$, a C++ template library with a Matlab like syntax. On top of various parallel programming idioms, $NT^2$ is built as a *DSEL* for high performance numerical computing on modern architectures. $NT^2$ adds a level of abstraction on top of Boost.SIMD and is able to handle multiple levels of parallelism. In this chapter, we present the challenges to design such a high level library and then we detail its API. We then discuss implementation details of the library and show how the core of $NT^2$ behave. Finally, we present benchmarks to validate the effectiveness of the library.

- **Conclusion and perspectives.** In this final chapter, we summarize all the results obtained in this typescript. To conclude, we discuss new directions for further research work.

Chapter 2

# From Architectures to Applications

Contents


**2.1 The Wide Architecture Landscape . . . . . 7**
2.1.1 SIMD extensions . . . . . 8
2.1.2 From single core to multi-core . . . . . 9
2.1.3 Accelerators . . . . . 10
2.1.4 Distributed memory systems . . . . . 12
2.1.5 Conclusion . . . . . 12
**2.2 Programming the Landscape . . . . . 13**
2.2.1 Low Level Parallel Tools . . . . . 13
2.2.2 Domain Specific Libraries for Scientific Computing . . . . . 20
2.2.3 Domain Specific Languages . . . . . 23
2.2.4 Domain Specific Embedded Language approach . . . . . 24
2.2.5 Conclusion . . . . . 25
**2.3 Proposed Approach . . . . . 25**


Architecture manufacturers have always pushed the design of machines whenever the hardware technology permitted it. With an ever growing architecture landscape, High Performance Computing (HPC) keeps taking advantage of these new designs. Software developers now need to stay up to date with the hardware when it comes to developing fast and reactive applications. Realizing such a task requires a rigorous methodology that combines hardware and software features. This chapter gives an overview of the various hardwares and softwares. We first present the contemporary hardware context of parallel programming with a description of the most relevant architectures. We then detail the software environment related to the introduced architectures.

## 2.1 The Wide Architecture Landscape

This section describes the various hardwares available in todays computing systems that range from SIMD extensions to distributed memory systems. The purpose of this section is not to fully detail every architecture but to give an outline that

illustrates what kind of architectures a modern scientific computing system can integrate.

### 2.1.1 SIMD extensions

Since the late 90's, processor manufacturers have been providing specialized processing units called multimedia extensions or Single Instruction Multiple Data (SIMD) extensions. The introduction of this feature has allowed processors to exploit the latent data parallelism available in applications by executing a given instruction simultaneously on multiple data stored in a single special register. Figure 2.1 illustrates the principle of an SIMD extension. With a constantly increasing need for performance in applications, today's processor architectures offer rich SIMD instruction sets working with increasingly larger SIMD registers.

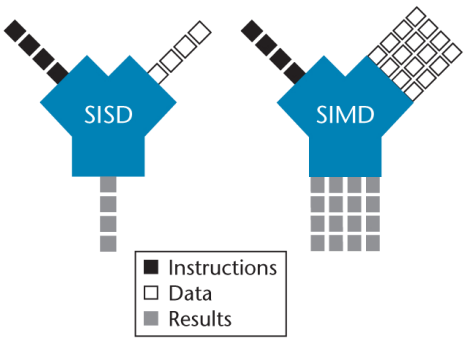



Figure 2.1: Principle of a SIMD computation unit

In the mid 90's, processor manufacturers focused their interests on developing parallel computing units that would permit to alleviate the CPU workload by computing a large amount of data at the same time. Indeed, business growth in multimedia applications brought out needs in terms of computing power. The preliminary tests of HP and Sun MicroSystem [91] permitted to fix the basics of SIMD extensions and opened the field for Intel and Motorola.

Intel enhanced the x86 instruction set with the SSE family. The MMX [80] instruction set is historically the first x86 SIMD instruction set introduced by Intel in 1997 with their P5-based Pentium series but it re-used floating point registers from the CPU, disabling scalar and SIMD computation at the same time. It also only works on integers types. The SSE family started officially in 1999 with the Pentium III for Intel and later with the AthlonXP for AMD.

In 1996, Motorola worked with Apple to design the new PowerPC G4 architecture. Motorola benefited from the experiences of its concurrents and decided to start from scratch a new extension. Finally, in 1999, the Apple PowerPC G4 went out with an Altivec unit from Motorola [35].

Since these early designs, SIMD extensions manufacturers have continued to increase the size of their dedicated registers and kept adding specific instructions. As an example, the forthcoming extension from Intel is AVX-512. It complements the SSE family and will be introduced in the next generation of the Xeon Phi series, Knights Landing coming in 2014. SIMD extensions introduced a method to handle data parallelism in mono-threaded applications.

### 2.1.2 From single core to multi-core

Processor manufacturers managed to optimize their CPU core by increasing the frequency and working on architectural optimizations like caches, instruction sets, pipelines, and superscalar architectures. The manufacturers then faced a technology limitation while increasing the frequency of their CPUs: the CPU power dissipation is directly related with the frequency. The dynamic power consumption of logic-gate activities in a CPU is approximated by the following formula: $P = CV^2f$ where $C$ is the capacitance, $f$ the frequency and $V$ the voltage. As a consequence, the power dissipation became a problem that is correlated with the decrease of the size of transistors. Since 2004, the processor frequency tends to stagnate. To alleviate this problem, manufacturers started to look for alternatives. Parallel computing was favored by the industry as a solution.

A lot of different configurations exists. Here, we show the main trend of manufacturers that is to constantly increase the core parallelism level in their new designs.

In 2001, IBM released the first multi-core based processor: the POWER4. It consists in two PowerPC AS with a unified L2 cache and works at 1 GHz. It was followed by Sun with the UltraSPARC IV composed of two UltraSPARC III cores (up tp 480 MHz). Intel and AMD launched their first multi-core in 2006. AMD released the Opteron server series (up to 2.6 GHz) and Intel, the Core Duo (up to 2.33 GHz), both with two cores.

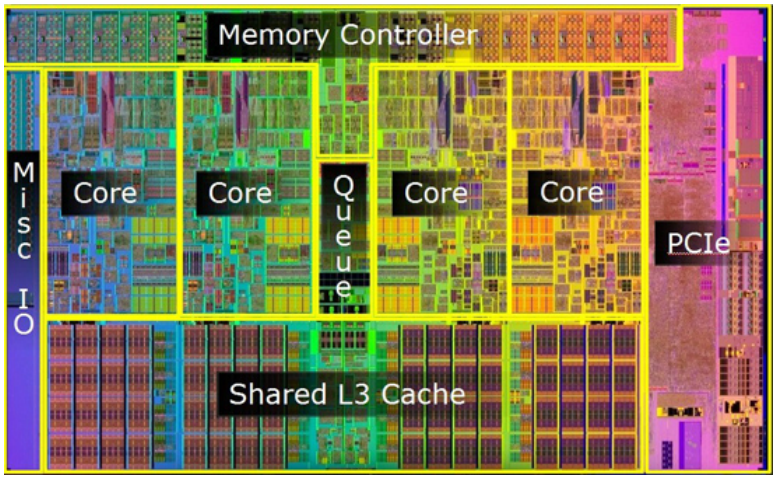



Figure 2.2: Picture of a quad-core die

Since then, multi-core designs has been the solution for manufacturers. Desktop and server processors are now proposing increasingly more cores. As an example the last series of Intel desktop processors proposes a Core i7-4770K running at 3.9 GHz with four physical cores (see figure 2.2). This processor embed the Hyper-Threated (HT) technology that was first released in the Pentium 4 Northwood (2000). It consists in sharing resources from a superscalar architecture by seeing two logical core inside a single physical core. In the case of the Core i7-4770K, the Operating System will see 8 logical cores and will be able to schedule 8 processes on these logical cores.

Server based solution increase significantly their number of cores. AMD Opteron solutions can go up to 16 cores and Intel Xeon solutions are up to 12 physical cores (24 logical cores with HT). Other manufacturers, like ARM, follows this approach and provides multi-core based architectures.

### 2.1.3 Accelerators

Multi-core architectures now propose a level of parallelism but it may not be sufficient with heavy computational applications. Accelerator based solutions appeared as a response to this demand. Accelerators are able to offload the CPU workload. In this section we present the main accelerators of our contemporary era.

- **General Purpose Graphic Processing Unit**

A Graphic Processing Unit or GPU is a dedicated integrated circuit designed to manipulate and accelerate the creation of images intended for output to a display system. These designs are a large consumer product thanks to the popularity of video games. They are designed as massively parallel to perform complex image calculation and their cost is reduced due to their availability in desktop computers. As parallel architectures for heavy computation are expensive and can not be acquired easily, GPUs are now a relevant solution for parallel computing [73]. At the end of 2006, NVIDIA released its first series of General Purpose GPU (GPGPU): the GeForce 8 series with 112 processing cores. It provides a theoretical peak single precision performance of 518 GFlops. This technology of GPGPU is called CUDA (Compute Unified Device Architecture). Figure 2.3 illustrates the principle of this technology.

Other manufacturers like AMD proposed a similar technology called ATI Stream but NVIDIA is still the leader in GPGPU. In addition of classic GPU enabled as CUDA devices, NVIDIA has a dedicated product line for CUDA based card. The NVIDIA Tesla K40 provides 2880 CUDA cores with a peak single precision floating point performance of 4.29 Tflops.

- **Intel Xeon Phi**

The Xeon Phi is a Many Integrated Core Architecture, as stated by its earlier name: the Intel MIC. This architecture include early research interests from the

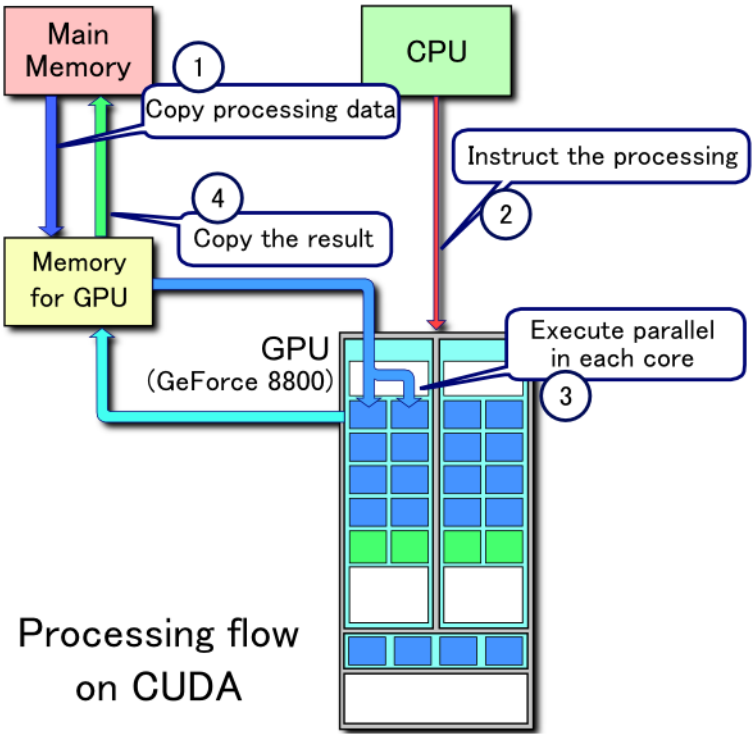



Figure 2.3: CUDA principle

Larrabee many core architecture, a research project called the Teraflops Research Chip and the Intel Single-chip Cloud Computer. Intel announced the launch of the Intel Xeon Phi family at the International Supercomputing Conference in 2012. Figure 2.4 shows the PCI card of the Xeon Phi. Two main architectures have been launched by Intel, the *Knights Corner* with more than 50 cores per chip (first trimester of 2013) and the *Knights Landing* with 72 Atom based cores with 4 threads per core (launch in 2014).

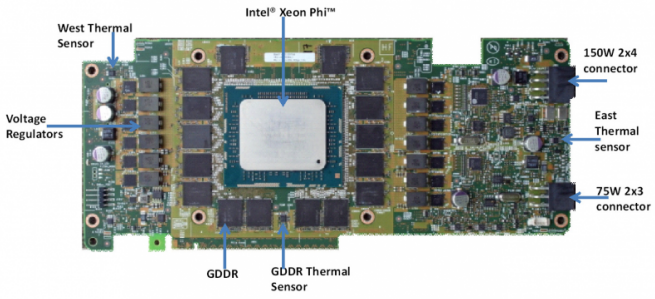



Figure 2.4: A Xeon Phi PCI card

The Xeon Phi proposes a x86 compatible multiprocessor architecture with the x86 ISA, 512 bits SIMD units, a 512 KB coherent cache per core and a wide ring bus connecting processors and memory. This solution proposed by Intel directly

competes with NVIDIA on the co-processor HPC market.

### 2.1.4 Distributed memory systems

A distributed memory system is a parallel machine with multiple computer architectures called *nodes* working together. Each node has its own memory and is connected with all the other nodes within a network topology creating a *cluster*.

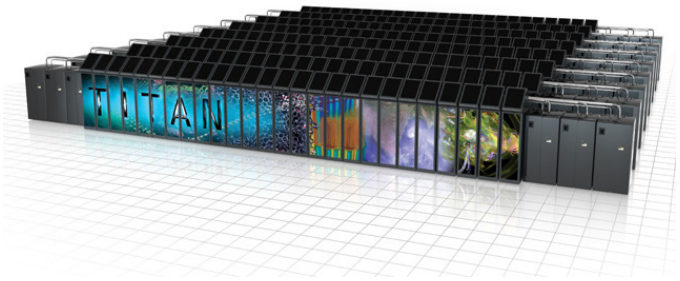



Figure 2.5: A picture of Titan

As each node has its own memory, these architectures requires a distributed computing approach. A node can present different configurations. It can hold several multi-core processors and a multi-GPU system or a Xeon Phi for example. A distributed system like Titan is composed of 299,008 AMD Opteron cores and 18,688 GPUs grouped in 18,688 nodes. Each node has 38 GB of memory. Titan is ranked at the second place of the top 500 super-computer list. Figure 2.5 shows the scale of a system like Titan. These type of clusters are designed for intensive applications with a heavy workload and large data sets to manipulate.

### 2.1.5 Conclusion

We saw in this section the diversity of architectures that developers encounters. Giving an exhaustive overview of such a large field is a hard task due to each manufacturer having different hardware architectures. Current trends are:

- Small-scale systems: cores coupled with SIMD units
- Big-scale systems : distributed memory systems + accelerators

A scientific computing system can be seen as a hierarchical system with different levels of parallelism. It can also present heterogeneous architectures. Architectures improvements are happening quickly and developers need to handle this race when it comes to programing parallel applications in a portable way.

## 2.2 Programming the Landscape

In this section, we present the tools available to develop on parallel architectures introduced in section 2.1. We present each tool by focusing on their expressiveness and their target support.

### 2.2.1 Low Level Parallel Tools

#### 2.2.1.1 SIMD tools

We introduced SIMD extensions in section 2.1.1 as a specific hardware computation unit for data parallelism. The specific instruction set of an SIMD extension can be used in different ways.

- The most common way to take advantage of a SIMD extension is to write calls to intrinsics. These low level C functions represent each SIMD instruction supported by the hardware, and while being similar to programming with assembly language it is definitely more accessible and optimization-friendly. With a lot of variants to handle all SIMD register types, the set of intrinsics usually only covers functionality for which there is a dedicated instruction, often lacking orthogonality or missing more complex operations like trigonometric or exponential functions. Due to its C interface, using intrinsics forces the programmer to deal with a verbose style of programming. Furthermore, from one extension to another, the Application Programming Interface (API) differs and the code needs to be written again due to hardware specific functionalities and optimizations. For example, some instruction sets provide fused operations that are optimized. Listings 2.1 presents a multiply and add operation implemented with SSE4.2.

Listing 2.1: SSE4 multiply and add implementation

```
1 __m128i a, b, c, result;
2 result = _mm_mullo_epi32(a, _mm_add_epi32(b, c));
```

On Altivec, the same multiply and add operation can be performed by a fused operation called FMA (Fused Multiply Add). Listing 2.2 shows the call to the intrinsic `vec_madd` that does the fused operation.

Listing 2.2: AltiVec FMA implementation

```
1 __vector int a, b, c, result;
2 result = vec_cts(vec_madd( vec_ctf(a,0)
3                          , vec_ctf(b,0)
4                          , vec_ctf(c,0)
5                          )
6                  ,0);
```

These two intrinsics examples demonstrate the complexity involved by the current programming model and its limitations to write portable applications.

• Compilers are now able to generate SIMD code through their autovectorizers. This allows the programmer to keep a standard code that will be analyzed and transformed into a vectorized code during the code generation process. Autovectorizers have the ability to detect code fragments that can be vectorized. For example, GCC autovectorizer [78] is currently available in GCC releases. This automatic process finds its limits when the user code is not presenting a clear vectorizable pattern (i.e. complex data dependencies, non-contiguous memory accesses, aliasing or control flows). The main approach is to transform the innermost loop-nest to enable its computation with SIMD extensions. The SIMD code generation stays fragile and the resulting instruction flow may be suboptimal compared to an explicit vectorization. Still on the compiler side, code directives can be used to enforce loop vectorisation (`#pragma simd` for ICC and GCC) but the code quality relies on the compiler and this feature is not available in every one of them. Dedicated compilers like ISPC [81], Sierra [72] or Cilk [85] choose to add a set of keywords to the language to explicitly mark the code fragments that are candidates to the automatic vectorization process. VaporSIMD [77] proposes another approach which consists in autovectorizing the C based code to get the intermediate representation of the compiler and then use a Just In Time based framework to generate portable SIMD code. With most of these approaches, the user code becomes non-standard and/or strongly dependent on specific compiler techniques. These techniques also rely on generating SIMD code from scalar code, disregarding the specificities of each of these computing units, including shuffle operations and intra- registers operations.

• Libraries like Intel MKL [60] or its AMD equivalent (ACML) [7]. Those libraries offer a set of domain-specific routines (usually linear algebra and/or signal processing) that are optimized for a given architecture. This solution suffers from a lack of flexibility as the proposed routines are optimized for specific use-cases that may not match arbitrary code constraints.

In opposition to this "black-box" approach, fine grain libraries like Vc [67] and macstl [2] propose to apply low level transformations to a specific vector type. For macstl, its support stops at SSE3 and its interface is limited to a few STL-compliant functions and iterators. Vc has a C++ class based approach with support for x86 processors only (SSE to AVX) and provide a list of SIMD enabled mathematical functions.

#### 2.2.1.2 Multi-core tools

Modern architectures are composed of multiple cores per chip and sometimes of multiple multi-cores like stated in section 2.1.2. Several tasks can be executed in parallel on these cores to speed up the computation. Multithreading allows to work with execution *threads* that are carried out on several computing units or cores. The main tools for such a programming model are pThreads, OpenMP and the Intel Thread Building Blocks Framework (TBB).

- **pThread**

The pThread library is an implementation of the POSIX 1003.1c Standard. The library provides a set of low level functions for creating, joining, synchronizing and destructing threads. These functions let the developer decides the life cycle of a thread. Listing 2.3 presents a simple sum of two arrays using pThread.

Listing 2.3: A pThread example - Sum of arrays

```
#include < pthread.h >

struct arg { float *a ,*b ,* r; };

void* func(void* in)
{
  arg * p = (arg*)(in);
  for(int i=0;i<250; i++)
    p->r[i] = p->a[i]+p->b[i];
  return NULL ;
}

int main ()
{
  float a[1000] , b[1000] , r[1000];
  pthread_t t[4];
  th_arg arg[4];

  for(int i =0;i <4; i ++)
  {
    arg [i ]. pa = &a[i *250];
    arg [i ]. pb = &b[i *250];
    arg [i ]. pr = &r[i *250];
  }

  for(int i=0;i <4; i++)
    pthread_create (&t[i], NULL, func, &arg[i]);

  for(int i=0;i<4;i++)
    pthread_join(t[i],NULL);
}
```

In this example, we instantiate four `pThread` structures to create four threads with the `pthread_create` function. The thread is created when this function terminates. We finally join and synchronize the threads with the `pthread_join` function.

pThread does not constraint the developer with a programming model. With this approach, the amount of applications that can take advantage of multithreading is more important. Task parallelism and data parallelism can be achieved with this library. However, pThread provides a really low level API resulting in a verbose programming style. Boost.Thread is an object oriented implementation of this Standard. Its interface is more high level and a set of classes permits to avoid the classical errors of concurrent programming.

- **OpenMP**

OpenMP is a Standard that specify directives and functions for using multithreaded programming through a high level interface [30]. These directives and functions are inserted in the source code and they enable to share the computation between cores. The directives tell to the compiler how to parallelize the corresponding code section. Listing 2.4 presents the same task achieved in listing 2.3 but written with OpenMP.

Listing 2.4: OpenMP directive example - Sum of arrays

```
#include <omp.h >

int main ()
{
  int i;
  float a [1000] , b [1000] , r [1000];

  # pragma omp parallel shared (a ,b ,r) private(i)
  {
    # pragma omp for schedule ( dynamic )
    for (i =0; i < 1000; i ++) r[i] = b[i ]+ a[i ];
  }
}
```

In this example, we first open a parallel section by using the `omp parallel` directive. In this section we define the scope of the variables. The arrays `a`, `b` and `c` are then shared between the cores via the `shared` directive. The `i` variable stays locals to each processors. The next section performs the effective parallelization with the `omp for` directive. It flags the following loop as a candidate for parallelization. The `schedule(dynamic)` option specifies how the loop will be distributed on each core. Here, each thread will get an iteration to perform and if a thread finishes its iterations it returns to get another one.

In opposition to pThread, OpenMP presents a simple model for programming Symmetric MultiProcessing machines. The data distribution and their decomposition is automated by the compiler directives. On the other hand, the portability of such a model relies on the support of the Standard inside compilers. The OpenMP 3.0 [14] Standard added the concept of *tasks* to the unified C/C++/Fortran specification of OpenMP 2.5. The new 4.0 version [15] released in July 2013 adds new features like : support for accelerators, SIMD, atomics, user defined reduction etc.

- **Intel TBB**

Intel TBB is a C++ template library that abstract multithreaded programming through high level primitives [83]. The library mostly provides parallel implementations of algorithm like `parallel_for`, `parallel_reduce` or containers like vectors

and queues. These implementation presents a Standard Template Library style. Listing 2.5 presents a simple average filter written with TBB.

Listing 2.5: TBB directive example - Sum of arrays

```
#include "tbb/parallel_for.h"
#include "tbb/blocked_range.h"

using namespace tbb;

struct sum
{
  const float* input1;
  const float* input2;
  float* output;
  void operator()( const blocked_range<int>& range ) const
  {
    for( int i=range.begin(); i!=range.end(); ++i )
      output[i] = input1[i]+input2[i];
  }
};

void parallelsum( float* output, const float* input
                , const float* input, size_t n )
{
  sum sum_;
  sum_.input1 = input1;
  sum_.input2 = input2;
  sum_.output = output;
  parallel_for( blocked_range<int>( 1, n ), sum_ );
}
```

In this example, we first declare a C++ function object that will take a `blocked_range<T>` as argument. TBB uses the range Concept to handle iterations. We then write a high level function which calls the `parallel_for` primitive. This primitive is close to OpenMP in terms of behavior. `parallel_for(range,body,partitioner)` provides a high level abstraction for parallel iterations. It represents parallel execution of `body` (`sum` in our example) over each value in `range` (a `blocked_range<T>` in our example). The optional `partitioner` specifies a partitioning strategy to distribute the iterations over the threads. Here, no partitioner appears in the call, TBB will use the default partitioner `auto_partitioner` which automatically split the range.

TBB takes advantage of C++ template programming to provide high level primitives. In addition, templates also add genericity when the code needs to work with different types.

- Further works on parallelizing loops has been done at the compiler level. By extending the polyhedral model to multi- core systems, new control structures provides a way to parallelize and nest parallelize loops. The *multifor* control structure introduced in [46] is an example of such an approach.

#### 2.2.1.3 Distributed memory system tools

- **MPI**

The main tools for such systems are implementations of the Message Passing Interface Standard [37]. Well known implementation are OpenMPI [51] and MPICH-2 [65]. This Standard presents functionalities for programming distributed memory systems:

- management of point to point and global communications;
- support for multiple languages (C, C++, Fortran, Java);
- possibility of developing high level libraries;
- heterogeneous support;
- support for multiple process topologies.

Listing 2.6 shows a "ping-pong" between two MPI processes that simply exchange their numerical identifier (rank). The `MPI_Init`, `MPI_Comm_rank` and `MPI_Finalize` functions starts the MPI environment and we use `MPI_Send` and `MPI_Recv` to communicate the rank.

Listing 2.6: A MPI example

```
int main ( int argc , char * argv [] )
{
  int rank, size;
  MPI_Status st;
  MPI_Init(&argc , &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  if(rank == 0)
  {
    MPI_Send(&rank, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
    MPI_Recv(&rank, 1, MPI_INT, 1, 1, MPI_COMM_WORLD, &st);
  }
  else if(rank == 1)
  {
    MPI_Recv (&rank, 1, MPI_INT,0, 0, MPI_COMM_WORLD, &st);
    MPI_Send (&rank, 1, MPI_INT,0, 1, MPI_COMM_WORLD);
  }
  MPI_Finalize();
  return 0;
}
```

MPI is a portable tool and can be used in the context of domain specific library design. MPI can also interact with debuggers and performance analysis tools. However, its API stays at a low level of abstraction. This lack of expressiveness makes it difficult to translate a sequential code to its parallel version. In addition, MPI has a verbose style of programming that impacts code readability and maintenance.

- **HPX**

HPX [56] for High Performance ParalleX is a C++ runtime system. It focuses on a unified programming model that is able to transparently use the available resources with a maximum of scalability. It relies on the C++11 Standard and provides a high level API. To maximize scalability, HPX combines different approaches: latency hiding, fine-grained parallelism and constraint based synchronizations are the main ones.

Listing 2.7: A HPX example - Fibonacci sequence

```
1  // Forward declaration of the Fibonacci function
2  boost::uint64_t fibonacci(boost::uint64_t n);
3
4  // Register the HPX Fibonacci action
5  HPX_PLAIN_ACTION(fibonacci, fibonacci_action);
6
7  boost::uint64_t fibonacci(boost::uint64_t n)
8  {
9    if (n < 2)
10     return n;
11
12   // We restrict ourselves to execute the Fibonacci function locally.
13   hpx::naming::id_type const locality_id = hpx::find_here();
14
15   fibonacci_action fib;
16   hpx::future<boost::uint64_t> n1 = hpx::async(fib, locality_id, n - 1);
17   hpx::future<boost::uint64_t> n2 = hpx::async(fib, locality_id, n - 2);
18
19   // Wait for the Futures to return their values
20   return n1.get() + n2.get();
21 }
22
23 int hpx_main(boost::program_options::variables_map& vm)
24 {
25   // extract command line argument, i.e. fib(N)
26   boost::uint64_t n = vm["n-value"].as<boost::uint64_t>();
27
28   {
29     // Wait for fib() to return the value
30     fibonacci_action fib;
31     boost::uint64_t r = fib(hpx::find_here(), n);
32   }
33
34   return hpx::finalize(); // Handles HPX shutdown
35 }
```

Listing 2.7 presents the Fibonacci sequence written with HPX. This example nicely illustrates the principles of *Future* on which the library is based. Futures are part of the C++11 Standard and HPX extends this feature to build an efficient runtime system. A HPX Future object encapsulates a delayed computation that can be performed on a *locality*. This object behaves like a proxy for a result that is not computed yet. It synchronizes the access of its result by suspending the thread requesting the value until the value is available. At line 16 and 17, we spawn HPX Futures to compute the $N-1$ and $N-2$ elements of the Fibonacci sequence. This is performed by asynchronously synchronizing the HPX `fibonacci_action`

recursively. Then, each elements of the sequence is computed asynchronously in a Future object spawns on the current machine (see the locality at line 13 that is set to `find_here()`). Futures will then returns their results as soon as they are available.

Such an approach enables HPX to get rid of global barriers for synchronizing threads of execution, then making the library able to improve the scalability of programs.

- **Stapl**

STAPL [8] (Standard Template Adaptive Parallel Library) is based on ISO Standard C++ components similar to the "sequential" ISO C++ Standard library. The library works with parallel equivalents of C++ containers (*pContainers*) and algorithms (*pAlgorithms*) that interacts through ranges (*pRange*). It provides support for shared and distributed memory and includes a complete runtime system, rules to easily extend the library and optimization tools.

Listing 2.8: A Stapl example - Parallel Sort

```
1 // Parallel container
2 stapl::pVector<int> pV(i,j);
3 // Call to parallel sort on a Stapl range
4 stapl::pSort(pV.get_pRange());
```

Listing 2.8 shows the use of Stapl parallel components.

#### 2.2.1.4 Conclusion

Programming modern parallel architectures requires a non negligible level of expertise due to the different abstraction levels introduced by the tools. From SIMD extensions to distributed memory systems, programming models differs and each application needs to be rethought in parallel. The source code also needs to be rewritten according to the chosen tools. Furthermore, the developer may want to combine different programming models and for example, take advantage of multicores and SIMD extensions within the same code. This task can be error prone and takes a significant amount of time. This limitation becomes particularly important when non parallel programming experts need to face this challenge.

### 2.2.2 Domain Specific Libraries for Scientific Computing

In this section, we present the most popular domain specific libraries related to scientific computing. These libraries provides "ready to use" sets of functions for a specific domain. We mainly focus on dense linear algebra, image and signal processing libraries to illustrates the solution proposed by domain specific libraries for Scientific Computing.

#### 2.2.2.1 Linear algebra libraries

The linear algebra domain uses intensively common operations like copying, vector scaling, vector dot products, linear combinations, and matrix multiplication. The Basic Linear Algebra Subprograms (BLAS) provides a set of low-level kernels that gathers these common operations. The first version was written in Fortran [71]. The BLAS API became a standard and several implementations have been realized. The standard implementation coming from Netlib is not optimized for parallel architectures. The Intel MKL library [60] and its concurrent, the AMD Core Math Library [7] (ACML), provides highly optimized BLAS routines for x86 processors with support for SIMD extensions and multithreading. GotoBLAS [53], OpenBLAS [108] and ATLAS [21] (Automatically Tuned Linear Algebra Software) are open source implementations of the BLAS and provide different levels of architecture optimizations. A GPGPU version of the BLAS is also available called cuBLAS [79] from NVIDIA and uses CUDA to accelerate the BLAS operations. The Accelerate framework [10] from Apple also provides an implementation of the BLAS.

The intent of BLAS subroutines is to be reused in linear algebra libraries and routines. The LAPACK [9] and LINPACK [36] libraries from Netlib [18] are making intensive use of the BLAS subroutines to build solver routines (simultaneous linear equations, least-squares solutions of linear systems of equations, eigenvalue problems, etc) or matrix factorization routines (LU, Cholesky, QR, etc). The performance of LAPACK is then dependent on the BLAS performance. The LAPACK API has also several optimized implementation. PLASMA [4] is a multithreaded implementation of LAPACK and can be linked with a SIMD optimized BLAS. MAGMA [4] proposes a hybrid support by taking advantage of GPUs and multi-core CPUs. ScaLAPACK [19] is optimized for distributed memory systems and can be linked with an optimized BLAS. The Intel MKL library proposes highly optimized implementations of LAPACK and ScaLAPACK.

The BLAS and LAPACK related libraries suffers from the lack of abstraction of their APIs. Mostly available in Fortran and C, their low level programming style hide the linear algebra domain specific informations. Even if the function semantic helps the developer, he still needs to handle memory allocation and not so easy function calls. A interesting project written in C proposes a new solution : the FLAME project [97].

The FLAME project proposes a new methodology for the development of dense linear algebra libraries. The project focuses on giving a full software suite that rethink the design of dense linear algebra algorithms. BLIS [103] is an optimized BLAS library written in C taking advantages of several SIMD extensions. The upper level tool, libFLAME [102], is a complete framework like LAPACK but its design is radically different. The internal design of libFLAME provides modern software engineering principles such as object abstraction and high level API

without sacrificing performances. The library handles abstractions that facilitates programming without array or loop indices, which allows the user to avoid painful index-related programming errors altogether.

#### 2.2.2.2 Image and signal processing

The domains of image and signal processing are often constrained by an output rate that needs to be fixed. This temporal constraint is now increasingly more important as the sizes and the numbers of inputs increases. This increase is related to technological improvements of sensors for example. This section presents the main solutions of these domains.

- **Intel Integrated Performance Primitives** [96] (Intel IPP) is a C/C++ multithreaded and SIMD optimized library for multimedia and data processing software. Intel provides support for MMX, SSE to SSE4, AES-NI and multi-core processors. The library provides functions for image and signal processing.

- **OpenCV** [16] (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 functionalities. The library is optimized with SSE2 and Intel TBB. It can also take advantage of Intel IPP if it is available.

- **FFTW** [50] (Fastest Fourier Transform in the West) is a computing library for discrete Fourier tranforms developed by Matteo Frigo and Steven G. Johnson at the Massachusetts Institute of Technology. The library provides support for SSE, SSE2, AVX, Altivec and ARM Neon SIMD extensions. Multithreaded support is also available. FFTW is a free software and its transforms are known as the fastest free implementations.

#### 2.2.2.3 Conclusion

The domain specific library approach introduces a level of abstraction with its function based programming style. Architectural optimizations are available and buried inside the library. Function semantic allow the developer to focus on domain specific calls but such an approach is still limited in terms of expressiveness. For example, memory allocations and specific library initialization techniques decrease the readability of the source code. The next level of abstraction can be reached when using a language designed with a semantic tied to a specific domain: a Domain Specific language or DSL.

### 2.2.3 Domain Specific Languages

A Domain Specific Language (DSL) is a programming language designed for a specific application domain. Its entire specifications are dedicated to the corresponding domain. It is then in opposition with classical programming languages that are designed for a general purpose. Martin Fowler proposes a definition of this approach in [48]. These languages improve the productivity of developers by alleviating the complexity of classic code and proposing a domain oriented abstraction for the language. Working with such languages also improves the interaction between domain experts and developers. A DSL provides a common syntax that has the advantage of being used by domain expert to express their ideas. Then, the DSL description of an algorithm is also an executable software. 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. In critical cases, like for example in languages like Erlang/OTP [33], domain specific validations or testing are made easier, since statements written with a given *DSL*s can be considered safe by design.

Several DSLs are available and their application domains can be very large. As a first example, LaTeX is a DSL for editing and preparing documents. LaTeX targets the communication and publication of scientific documents. The philosophy of LaTeX is that authors should focus on the content of the document instead of the visual presentation of the document. Authors then specify the logical structure of the document by using a set of tags. Another example is the Structured Query Language (SQL) that provides a normalized language for exploiting databases. A annotated bibliography of DSLs is available in [101]. Another notable initiative is the R language [98]. R proposes a complete environment for statistical computing and is widely used by data miners and statisticians. It is mostly used to design data analysis and statistical software.

In Scientific Computing, the related domains like mathematics or physics are targeted by the Matlab language [74]. The Matlab language is a DSL that comes with numerical computing environment. Matlab was initially designed by Cleve Moler from the University of New Mexico. His purpose was to give his students access to LINPACK without learning Fortran. Matlab then spread out to other universities and the applied mathematics community found a strong interest in this tool. Matlab was originally written in C and in 2000, it was rewritten to use LAPACK. It is developed by MathWorks and its main feature is its ability to manipulate matrices. It also provides implementation of algorithms, plotting facilities and interfaces with other language like C/C++ or Java. Additional toolboxes are available and they cover various domains. In 2004, Matlab community was estimated around one million users across industry and academia. Matlab 's' user base profiles range from science engineering to economics.

The DSL approach is tie to a computing environment. For a Scientific Computing DSL like Matlab , its efficiency relies on the performance of its computing environment. The Matlab language can be written directly in the Matlab command prompt or by using Matlab scripts. It is dynamically compiled which

introduced a significant overhead for the calculation time. Several *DSL*s are dynamically compiled or relie on interpreters or virtual machines to be executed. This implementation characteristic of *DSL*s impacts directly the performance of such languages. The Domain Specific Embedded Language (DSEL) approach tries to alleviate this limitation.

### 2.2.4 Domain Specific Embedded Language approach

A sub-class of *DSL*s actually provides a better way to mitigate the abstraction vs efficiency trade-off: **Domain Specific Embedded Languages** (or *DSEL*s ) [99, 59]. *DSEL*s are languages implemented inside another, usually general-purpose, host language [29]. They share the advantages of *DSL*s as they provide an API based on domain specific entities and relations. However, *DSEL*s usually 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** [28, 107] – in languages providing some level of introspection or providing constructs to manipulate statements from within the language. If such features are common in functional languages (like OCaml or Haskell) or scripting languages (like Ruby), they are less so in imperative languages. C++ is providing a set of similar features thanks to template based meta-programming [3].

- **Blitz++** [106] is a C++ template library for high performance mathematics. The library relies on advanced C++ techniques to provide optimized mathematical operations. Historically, Blitz++ is the first library to use C++ template meta-programming (see chapter 3) and has been recognized as a pioneer in this area.

- **Armadillo** [87] is a C++ open source library for scientific computing developed at NICTA in Brisbane. The API of Armadillo is similar to MATLAB (see section 2.2.3). It provides a various set of linear algebra and matrix based functions, trigonometric and statistics functions. Its implementation relies on lazy evaluation. This technique (see chapter 3 for details) permits to combine operations and reduce temporaries at compile time. 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** [13] is a C++ template library for high-performance dense and sparse arithmetic. Blaze implementation is based on the Expression Templates technique (see chapter 3 for details). It allows to manipulate and optimize an expression evaluation at compile-time. The performance of the libraries relies on BLAS implementations.

• **Eigen** [57] 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 better cache and register reuse in Linear Algebra kernels.

• **MTL**[1] [54] 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 handle combination of kernels. In addition to the performance demands, MTL4 hides all this code complexity from the user who writes applications in natural mathematical notation.

### 2.2.5 Conclusion

We saw in this section the diversity of available solutions for scientific computing. Low level libraries/tools permit fine-grained programming for parallel architectures but end up hiding domain specific informations therefore preventing the user from focusing on the domain related algorithm. Domain specific libraries are able to embed this low level approach inside functions. They provide functions with semantic information of the domain but are still dependent on languages like C or Fortran and the user needs to be familiar with them. *DSL*s alleviate these limitations by their high expressiveness tie to a specific domain but lack in performance due to their implementations prevents a *DSL* like MATLAB to be used in the context of Scientific Computing that is performance driven. The *DSEL* approach proposes an implementation design that enables the use of powerful features from the host language. In the case of C++ , *DSEL*s can reach architecture optimizations through its common support for C based libraries.

## 2.3 Proposed Approach

Our approach aims at developing high level tools for scientific computing. We want to take advantage of expressiveness for designing a simple and intuitive API for the user. As stated earlier in this section, *DSL*s are the most abstract languages with a domain oriented semantic but their performances are not relevant for scientific computing. *DSEL*s are then good candidates. Embedded in a host language, they allow to use the language features and then take advantage of the reachable performance of the language.

[1] Matrix Template Library

C++ through its template mechanism can be used for such an approach and several tools illustrate it. We therefore chose the approach of a C++ *DSEL* based solution for its expressiveness. Working with C++ also allows us to reach architecture level optimizations through its native aspect. On the architectural side, the diversity of available systems is significant. We then choose to target small-scale systems (multi-cores with SIMD extensions). The interest for such systems is rising due to their massive availability nowadays. To develop such a solution, we introduce two paradigms that will ease the development of a *DSEL* : Generic and Generative Programming.

Chapter 3

# A Generic and Generative Programming Approach for DSL

## Contents


**3.1 Software context . . . . . 27**
3.1.1 The configuration space approach . . . . . 28
3.1.2 Library level exploration . . . . . 29
**3.2 Programming techniques . . . . . 30**
3.2.1 Template Meta-Programming . . . . . 30
3.2.2 Expression Templates . . . . . 30
3.2.3 The Boost.Proto library . . . . . 32
3.2.4 Conclusion . . . . . 36
**3.3 The *DEMRAL* Methodology . . . . . 36**
3.3.1 *DSEL*s design considerations . . . . . 36
3.3.2 From *DSEL*s to Architecture Aware *DSEL* . . . . . 37
**3.4 Conclusion . . . . . 39**


In the previous chapter, we presented the approach chosen for our research works. *DSEL*s has been preferred for their ability to be embedded in a host language without losing their expressiveness aspect. We also introduced the need for software design to be tied to the parallel architecture features. In this chapter, we present the *DSEL* approach and its related techniques. We first detail the software context of such languages. Then, we introduce the existing design methods. We finally present a new methodology for designing Architecture Aware *DSEL*s (AA-DSELs).

## 3.1 Software context

In our case, the design of *DSEL*s takes place in the context of scientific computing. As stated in chapter 2, developing complex and fast scientific applications is not a trivial task. This difficulty is the result of a large **configuration space** composed of algorithmic and architectural requirements.

### 3.1.1 The configuration space approach

The configuration space approach gathers different optimization techniques. These techniques rely on a common optimization process which purpose is to explore a space of possible configurations. The optimization process then combines algorithmic and architectural factors for a specific system. This process leads to the selection of a factor combination. The resulted combination then ensures an optimal performance of the software. We can find such an approach in different software solutions.

*Iterative compilation* [100, 82] is part of the configuration space techniques. Classic compile-time optimization techniques relies on multiple code transformations that are applied through predictive heuristics. These heuristics become more and more inefficient when the underlying architecture is complex. Moreover these transformations can degrade the performance in certain cases. *Iterative compilation* alleviates these limitations by exploring an optimization space and then choosing the best one. This selection process is also used at the library level.

Library based solutions can take advantage of such an approach by using pre-built optimized binaries. ATLAS [21] is one of them. It relies on the *Iterative compilation* technique that is used during the installation of the library. The binaries are accelerated by a hierarchical tuning system that takes care of low level functions. This system applies a large selection process for these functions and then ensures an optimal performance. Optimized binaries are then generated and BLAS 3 operations can exploit these versions.

In the previous solutions, the configuration space exploration takes place at compile-time. Another method consists in exploring this space at runtime. StarPU [11] is a task programming library that uses a special runtime system. This runtime can monitor in real-time the performance of each function on a given hardware configuration. It then allows the runtime of StarPU to select the most optimized version of a function. This monitoring method is able to tune the underlying algorithm of a function by changing its parameters (tiling size, number of iterations, etc) or the targeted architecture (CPU, GPU, hybrid version). Such a method allows a fine tuning of the application. The task scheduling strategy is chosen after a runtime exploration that permits to reduce load balancing and data locality issues.

The compile-time approach depends on the compiler implementation and limits the portability of an application. The runtime approach can have some overhead while scheduling tasks in the case of StarPU. These methods are still valid approaches and gives good results. To alleviate these factors, we aim at providing a library level system for such exploration that will complement the compiler work. In the next section we detail techniques that provides a way to perform such an approach.

### 3.1.2 Library level exploration

#### 3.1.2.1 Current approach

Design techniques for library level software in C++ relies for the most part on the object-oriented approach. Design patterns [52] are a relevant work based on this approach and provides 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. This method has some limitations when it comes to dealing with many software components and especially with a large configuration space. In addition, the generalization of design patterns builds a semantic gap between domain abstractions on which rely expressiveness and programming language features on which rely performance. Then, performance penalties can then occur. To overcome these limitations, Czarnecki introduced **Generative Programming** [25, 26] as a new paradigm for software development.

#### 3.1.2.2 Generative programming

**Generative Programming** has been defined by Czarnecki in [24] as "*a comprehensive software development paradigm to achieving high intentionality, re-usability, and adaptability without the need to compromise the runtime performance and computing resources of the produced software*". This approach consists in defining a model to implement several components of a system. Current practices assemble manually these components. For example, the Standard Template Library provides components that the user needs to aggregate according to his *configuration knowledge*. Generative Programming pushes further this approach by bringing automation in such practices. The model that the developer uses to assemble components is moved to a generative domain model. This results in a generator embedding a *configuration knowledge* that takes care of combining the components. This process then relies on the three following steps as stated in [26]:

- design the implementation components to fit a common product-line architecture;
- model the *configuration knowledge* stating how to translate abstract requirements into specific constellations of components;
- implement the configuration knowledge using generators.

Respecting these design considerations will ensure the transition from a *configuration space* with domain-specific components and features to a solution space that encapsulates expertise at the source-code level. This method can be embedded within a library. These specific libraries are called *active libraries*.

#### 3.1.2.3 Active libraries

In opposition to classic libraries, *Active libraries* [107] takes an active role during the compilation phase to generate code. They aim at solving the abstraction/efficiency trade-off problem we introduced in chapter 2. They base their approach on defining a set of generative programming methods. These libraries provide domain-specific abstractions through generic components and also define the domain-driven generator to control how these components are optimized. By carrying domain-specific semantic at a high level, this technique enables a semantic analysis of the code before any real code generation process kicks in. Such informations and transformations are then carried on by a meta-language that allows the developer to embed meta-informations. Once the generator finds a solution space in the configuration space, the code generation phase starts resulting on an optimized version of the code. The main approach to design such libraries is to implement them as Domain Specific Embedded Languages (*DSEL*s ). As stated in section 2.2.4, *DSEL*s are easier to design as they reuse general purpose language features and existing compilers. They also rely on a domain dependent analysis to generate efficient code.

## 3.2 Programming techniques

The implementation of active libraries is possible through a technique called *Template Meta-programming*.

### 3.2.1 Template Meta-Programming

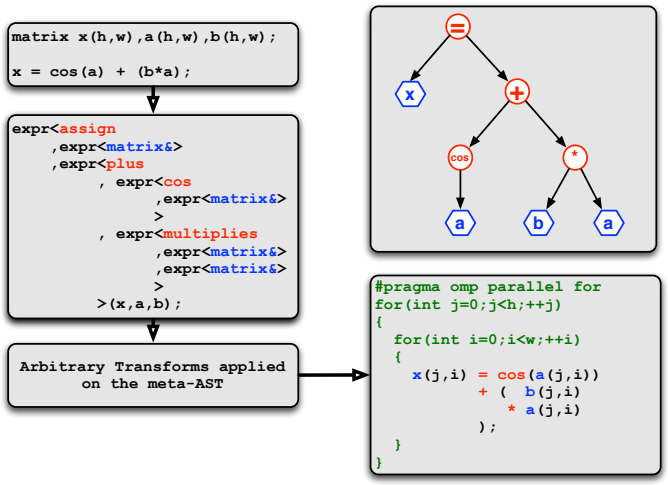
Template meta-programming is a technique used to design active libraries. Templates provides a generative programming technique in which they are used by a compiler to generate temporary source code. The code generation process is done by meta-programs that are executed by the compiler itself. The resulting temporary source code is then merged with the rest of the source code. The compiler can then finally finish the compilation process. Through this technique, compile-time constants, data structures and complete function can be manipulated. The execution of meta-programs by the compiler enables the library to implement domain-specific optimizations that lead to a complete domain oriented code generation. Such a technique can be hosted by several languages like C++ [3], D [17], Haskell [90] and OCaml [89]. Template meta-programming is then pushed further to design *DSEL*s . This technique is called *Expression Template*.

### 3.2.2 Expression Templates

**Expression Templates** [105, 104] is a technique implementing a form of textbfdelayed evaluation in C++ [92]. **Delayed evaluation** in C++ is the entry point of the Expression Template technique. The delayed evaluation technique is also called *lazy evaluation*. C++ does not support delayed evaluation of expression natively. Expression Templates are built around the *recursive type composition* idiom [62] that

allows the construction, at compile-time, of a type representing the abstract syntax tree of an arbitrary statement. This is done by overloading functions and operators on those types so they return a lightweight object which type represents the current operation in the Abstract Syntax Tree (AST) being built instead of performing any kind of computation. Once reconstructed, functions can be used to transform this AST into arbitrary code fragments using Template meta-programming on the AST type (see figure 3.1).

Figure 3.1: 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 complexity of the boilerplate code is usually as big as the actual library code, making such tools hard to maintain and extend. To avoid such a scenario, tools encapsulate the Expression Template technique as reusable frameworks with extended features.

The Portable Expression Template Engine or PETE [58] extends the expression template technique and provides an engine to handle user defined types in expression statements. It is used in the POOMA framework [84] that provides a set of C++ classes for writing parallel PDE solvers. With PETE, the user can use the engine and apply transformations at the AST level. PETE presents some limitations and its engine does not allow the user to perform common transformations on the AST as it only evaluates expressions with a bottom-up approach. This engine also lacks of domain specific consideration while manipulating expressions.

Besides C++ based programming techniques, Scala [88] provides a native support for AST constructions and transformations.

### 3.2.3 The Boost.Proto library

To alleviate these shortcomings, Niebler has proposed a C++ compiler construction toolkit for embedded languages called Boost.Proto [76]. 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 capture. Compared to hand-written Expressions Templates-based *DSEL*s , designing a new embedded language with Boost.Proto is done at a higher level of abstraction by designing and applying **Transforms** that are functions operating via pattern matching on *DSEL* statements. In a way, Proto supersedes the normal compiler workflow so that domain-specific code transformations can take place as soon as possible.

The main idea behind Boost.Proto is the construction of an AST structure through the use of terminals. A Boost.Proto `terminal` represents a leaf of an AST. The use of a `terminal` in a expression "infects" the expression and builds a larger Boost.Proto expression. These expressions are tied to specific domains as Boost.Proto aims at defining *DSEL*s . To illustrate the possibilities of the library, we present a simple analytical function *DSEL* written with Boost.Proto . This *DSEL* will allow us to evaluate analytical expressions of the following form:

```
(x*5 + 2.0*x - 5)
```

We will specify the value of `x` by using the parenthesis operator and it will also triggered the evaluation of the expression like in the following example:

```
(x*5 + 2.0*x - 5)(3.0)
```

Boost.Proto can be seen as a compiler in the sense that it provides a similar way to specify your own language. In comparison to classic compilers, the first entry point of the library is the specification of `grammar` rules. Boost.Proto automatically overloads all the operators for the user but some of them may not be relevant for a DSL. This means that it may be possible to create invalid domain expressions. Boost.Proto can detect invalid expressions through the use of a Boost.Proto `grammar`. A `grammar` is defined as a series of valid grammar elements. In our example, we want to allow the use of:

- classical arithmetic operators;
- analytical variables;
- numeric literals.

We then define a `grammar` that matches these requirements: it is presented in listing 3.1.

Listing 3.1: Analytical grammar with Boost.Proto

```
1  // Terminal type discriminator
2  struct variable_tag {};
3
4  struct analytical_function
5    : boost::proto::or_
6    <
7      boost::proto::terminal< variable_tag >
8
9    , boost::proto::or_
10       < boost::proto::terminal< int >
11       , boost::proto::terminal< float >
12       , boost::proto::terminal< double >
13       >
14
15   , boost::proto::plus<analytical_function,analytical_function>
16     // both unary and binary negate
17   , boost::proto::negate<analytical_function>
18   , boost::proto::minus<analytical_function,analytical_function>
19   , boost::proto::multiplies<analytical_function,analytical_function>
20   , boost::proto::divides<analytical_function,analytical_function>
21 >
22 {};
```

At line 7 of listing 3.1, we allow all terminals that hold a `variable_tag`. This type enables the discrimination between analytical variables and other terminals. At line 9, we allow numeric literals in our expressions. For this specific case, Boost.Proto wraps the literals in terminals. We finally allow the arithmetic operators.

Boost.Proto can now construct valid ASTs. These expression trees does not encapsulate any domain semantic for now. The AST type is a raw tree as if it was extracted from the work-flow of a compiler. The library allow us to add domain semantic to an AST through the declaration of a user-defined domain and a user-defined expression class. This process allow the user to merge the domain-semantic information with the raw structure of an expression.

The next step consists in specifying the domain for our analytical DSL. This is done by inheriting from `proto::domain` and linking this domain to an expression generator of a user defined expression type. Listing 3.2 shows the domain declaration.

Listing 3.2: Domain definition with Boost.Proto

```
1  // The user defined expression type.
2  template<typename AST>
3  struct analytical_expression;
4
5  // The analytical_domain inherits from proto::domain.
6  struct  analytical_domain
7        : boost::proto::domain< boost::proto::generator<analytical_expression>
8                              , analytical_function
9                              >
10 {};
```

Once the domain declaration is done, we can now build our `analytical_expression` class. We add a specific interface to this class as we want to be able to call the `operator()` on an expression to evaluate it with a given set of variables. It does not provide the definition of the `operator()`, we will see how we evaluate our expression later. At this point, we do not provide any particular behavior to this operator. We will see later how we evaluate an expression. Listing 3.3 presents the `analytical_expression` class that inherits from `proto::extends`. `proto::extends` is an expression wrapper that imbues an expression with analytical domain properties.

Listing 3.3: User-defined expression type with Boost.Proto

```
template<typename AST>
struct  analytical_expression
      : boost::proto::extends< AST
                             , analytical_expression<AST>
                             , analytical_domain
                             >
{
  typedef boost::proto::
          extends< AST
                 , analytical_expression<AST>
                 , analytical_domain
                 > extendee;

  // Expression must be constructible from an AST
  analytical_expression(AST const& ast = AST()) : extendee(ast) {}

  BOOST_PROTO_EXTENDS_USING_ASSIGN(analytical_expression)

  //Provides the operator() overloads and makes it a Callable Object.

  typedef double result_type;

  result_type operator()(double v0) const;
};
```

Now, we need to implement `operator()` so that Boost.Proto can evaluate the value of our analytical expressions. Boost.Proto handles that by providing **Transforms** which specifies rules that need to be performed when the AST is evaluated. A Transform is a *Callable Object* [93] defined in the same way that a Proto grammar. Transform rules can be extended with a semantic action that will describe what happens when a given rule is matched. The library provides a lot of default transforms that we will use in our example. Our transform that evaluates our expression needs to behave differently while walking the AST and encountering its nodes:

- If it is a terminal, we want to extract the corresponding value;
- If it's an operator, we want it to do what the C++ operators does.

To achieve this, we write the `evaluate_` transform that relies on the use of default transforms. `proto::when` is used here to associate a rule to a specific action. The `evaluate_` transform is presented in listing 3.4

Listing 3.4: The `evaluate_` transform

```
1  struct  evaluate_
2        : boost::proto::or_
3          <
4            boost::proto::when
5            < boost::proto::terminal< variable_tag >
6            , boost::proto::_state
7            >
8          , boost::proto::when
9            < boost::proto::terminal< boost::proto::_ >
10           , boost::proto::_value
11           >
12         , boost::proto::otherwise< boost::proto::_default<evaluate_> >
13         >
14 {};
```

If we want to evaluate an expression like `(x+2.0*x)(3.0)`, we need to evaluate each node and accumulate the result while we walk the AST. Transforms related to accumulation are common when processing ASTs. Boost.Proto provides a clear way to achieve these transforms: the `_state` of an AST. In our case, the `_state` is used at line 6 to pass the value of the analytical variable through each node and ask each node to evaluate themselves with it (see listing 3.5).

Listing 3.5: `operator()` implementation using `evaluate_`

```
1 result_type operator()(double v0) const
2 {
3   evaluate_ callee;
4   return callee(*this,v0);
5 }
```

The evaluation of the analytical expression is performed in the following way:

```
(x + 2.0*x)(3.0)
```

First, the '+' node:

```
( x(3.0) + (2.0*x)(3.0) )()
```

Then, the '*' node:

```
( x(3.0) + (2.0*x(3.0)) )()
```

And finally, the terminals evaluation:

```
3.0 + (2.0*3.0) = 9.0
```

We notice the use of `proto::_` (line 9) that permits to match any other terminals that are not analytical variables. In this particular case, we directly extract the value of the terminal. Literals will match such a case. At line 12, we simply tell the library to use the default behavior of operators.

At the end, we can write analytical expressions that match the correct grammar and evaluate it. This is done by defining terminals and building an expression using them. Listing 3.6 shows how our small analytical *DSL* in action.

Listing 3.6: Analytical expression in action

```
// Last step, we have to redefine _x to be an analytical_expression.
analytical_expression< boost::proto::terminal< variable_tag >::type > const
    _x;

int main()
{
  std::cout << (_x*3 + 9.0*_x)(2) << "\n"; // Output : 24
}
```

This small example showcases the BOOST.PROTO library. This library has a lot of powerful features that we can not present in a simple and concise way. The BOOST.PROTO library is one of the most complete solution for designing *DSEL*s in C++ . BOOST.PROTO can be seen as a *DSEL* to design *DSEL*s .

### 3.2.4 Conclusion

In this section, we presented the techniques related to active library design. From these techniques we can define the *configuration knowledge* (*i.e.* the generator) that will deduce from the *configuration space* a solution space. The maintenance of active libraries that embed such techniques is hard to achieve. BOOST.PROTO automates this approach and gives a powerful way to manipulate AST structures and apply domain-oriented transformations on them. The high interoperability between ASTs and transformations in BOOST.PROTO provides a clear approach for specifying *DSEL*s .

Meta-programming techniques then allow us to hide from the end-user the code generation process and also abstract the user interface with strong domain semantic but these approaches lack of methodology to specify a complete *DSEL* . Czarnecki has been studied a methodology to design new *DSEL*s : the *DEMRAL* methodology.

## 3.3 The *DEMRAL* Methodology

### 3.3.1 *DSEL*s design considerations

Czarnecki explored how *Generative Programming* would help the design of active libraries. Complex software systems can be broken down to:

- a list of interchangeable components which tasks are clearly identified;
- a series of generators that combines components by following rules defined by an *a priori* domain specific analysis.

In [27], Czarnecki proposed a methodology called **Domain Engineering Method for Reusable Algorithmic Libraries** (*DEMRAL* ) showing a possi-

ble formalization of Generative Programming techniques. It 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.

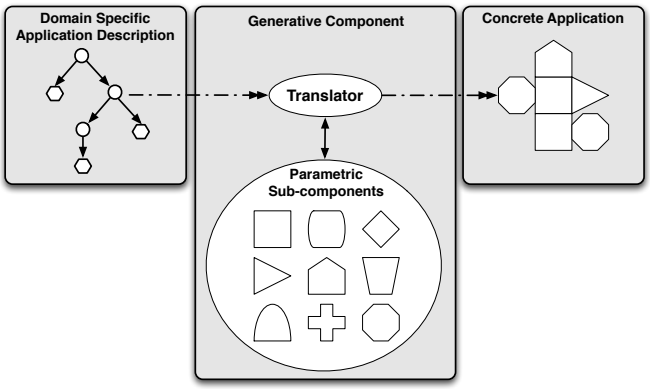



Figure 3.2: The *DEMRAL* methodology

Figure 3.2 illustrates the *DEMRAL* methodology. *DEMRAL* reduces the effort needed to develop software libraries by limiting the amount of code to write. As an example, a library providing $N$ algorithms operating on $P$ different related data-structures may need the design and implementation of $N * P$ 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 specificities. This approach allows high re-usability of generic components while their behaviors can be customized.

Taking into consideration the *DEMRAL* methodology helps designing *DSEL*s as this formalization relies on the software aspect of generic components. As we stated in chapter 2, in the context of scientific computing we need to inject architecture specific implementations in todays software. We then present the Architecture Aware *DEMRAL* methodology as the extension of the *DEMRAL* methodology.

### 3.3.2 From *DSEL*s to Architecture Aware *DSEL*

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.

In this Architecture Aware DEMRAL (*AA-DEMRAL* ) methodology, the implementation components are themselves generated from a generative component which translates an abstract architecture description into a set of concrete implementation components to be used 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. 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. Figure 3.3 illustrates the new *AA-DEMRAL* methodology.

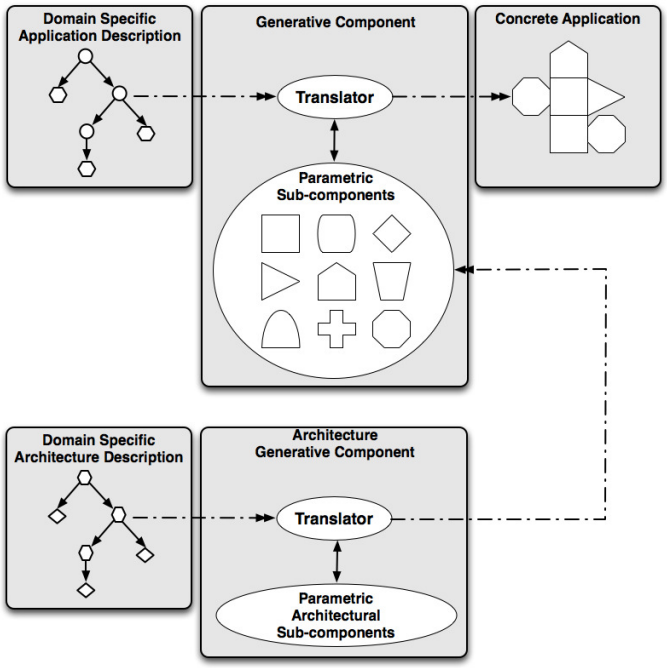



Figure 3.3: The *AA-DEMRAL* methodology

By analogy with 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 keeps the advantage of the *DEMRAL* methodology and permits to keep focusing on domain related optimizations while developing

the library. In addition, the genericity of the components at both levels (hardware and software) allows the generative components to explore a complete configuration space with a sub-part corresponding to specific architectural optimizations thus making the code generation process strongly aware of architectural aspects. The best solution space can then be selected by the generative components.

## 3.4 Conclusion

Our contribution pushes further the *DEMRAL* methodology and provides a new way for designing architecture aware *DSEL*s . With such an approach, active libraries can take in consideration hardware capabilities and then increase the quality of their code generation process. *DSEL*s keeps their expressiveness and, at the same time, are able to improve their evaluation strategy. This new methodology now requires to be easily implemented in the context of active libraries.

The challenges behind such an implementation are:

- Abstract architectural components to specify a *hardware configuration space*;
- Build a generative component that can embed knowledge to choose between hardware components;
- Build a generative component that can embed knowledge to aggregate software and hardware components.

Our work will focus on providing architectural components for SIMD and multi-core based systems (*i.e.* small-scale systems) like said at the end of chapter 2. We then propose a library that embed an architecture aware function dispatching system: the Boost.Dispatch library.

Chapter 4

# The Boost Dispatch Library

Contents


**4.1 Challenges . . . . . 41**
4.1.1 Regular C++ function overloading . . . . . 42
4.1.2 Free function dispatching using SFINAE . . . . . 43
4.1.3 The Tag Dispatching technique . . . . . 44
4.1.4 Concept based overloading . . . . . 45
4.1.5 Conclusion . . . . . 45
**4.2 The Boost.Dispatch Library . . . . . 46**
4.2.1 The *AA-DEMRAL* methodology in Boost.Dispatch . . . . 46
4.2.2 The Hierarchy Concept . . . . . 47
4.2.3 Compile Time Hierarchy Deduction . . . . . 48
4.2.4 Built-in Hierarchies . . . . . 49
4.2.5 Common API . . . . . 56
**4.3 Conclusion . . . . . 59**


In chapter 3, we introduced the notion of Architecture Aware *DSEL* (AADSEL) as an entry point for designing new parallel software tools. This approach allows the tool designer to inject architecture details inside the evaluation process of the *DSEL* . With an overgrowing architecture landscape and the necessity to fully exploit their potential, library designers are confronted to new software designs.

In this chapter we present Boost.Dispatch, a C++ template library for writing functions and functors dispatching using a generic Tag Dispatching system to simplify the application of the *AA-DEMRAL* methodology to software design.

## 4.1 Challenges

As introduced in chapter 3, Generic Programming is a powerful approach to design reusable software components. From the tool developer perspective, these programming techniques can improve the internal design of a library. From the user perspective, providing a generic interface (either for tool designers or application developers) impacts directly their code by giving an access to easily reusable software components in different contexts. C++ introduces a efficient way to write

and use Generic Programming through its template system. Generic Programming in C++ relies on several techniques that we introduced in chapter 3. One of them is the ability of C++ to provide different approaches to dispatch a function over an arbitrary list of types.

### 4.1.1 Regular C++ function overloading

Function overloading is a C++ feature that permits to have different function definitions with the same function name. When this function is called, the C++ compiler must choose which function definition to call. The decision between the different candidates is made by respecting a set of rules.

Listing 4.1: Regular overloading in C++

```
1  float f(float)
2  { return 0; }
3
4  int f(int)
5  { return 1; }
6
7  unsigned int f(unsigned int)
8  { return 2; }
9
10 int main()
11 {
12   cout << f(2.0) << endl;
13 }
```

We consider the example in listing 4.1. The overloaded function `f` is called and the compiler must distinguish between the candidates. The types of the call arguments are used here to perform the selection. In the example, the compiler will dispatch the call to the `float` version of the function. The process that models this decision is the **overload resolution process**. The complete processing of a function call is complex and the overload resolution is just a part of it. Here we just present the basic set of rules concerning the overload resolution process.

- The first step consists in looking up the function name and building the *overload set*. The template variants of the function are added after the template deduction occurred.
- In the overload set, any candidate that doesn't match the call is then eliminated (wrong number of arguments, implicit conversion mismatch, default argument mismatch). The resulting candidates are a set of *viable function candidates*.
- Then, the overload resolution is performed to select the *best* candidate. If the process fails, the call is ambiguous.

- The last step check the selected candidate. If it is a member function, it checks its accessibility. If it is an inaccessible private member, an error is issued.

According to this set of rules, function overloading provides a way to distinguish function variants. However, this approach is limited by its selection process that only relies on the types of the call arguments. A more fine-grained approach can not be achieved as the properties of the types and even further information can not be injected inside the overload resolution process. For example, if we want a definition of the function `f` that works for every floating types, the regular overload approach forces us to duplicate the function definition for the `float` and `double` types.

### 4.1.2 Free function dispatching using SFINAE

During the construction of the overload set, some overload candidates come from template parameter substitutions. If the template substitution fails, the corresponding candidate is then evicted from the overload set and the compiler does not issue an error. This process is called *Substitution Failure Is Not An Error* (SFINAE). SFINAE takes place during the template deduction phase and consists in dispatching a function by using an **arbitrary overloading** technique. In this case, SFINAE is used to dispatch a function call through through an arbitrary compile-time condition.

For example, we want to define a unary function `f` for all built-in arithmetic types that implement different algorithms depending on the actual type of the argument. For example, consider that f returns 0 if its argument is a floating point value, returns 1 if it is a signed integer or return 2 otherwise. You could do this using SFINAE and `std::enable_if`. Listing 4.2 illustrates this technique.

Listing 4.2: Free function dispatching using SFINAE

```
#include <type_traits>

template<class T>
typename std::enable_if<is_floating_point<T>::value,int>::type
f(T)
{ return 0; }

template<class T>
typename std::enable_if< is_signed<T>::value && is_integral<T>::value
                       , int >::type
f(T)
{ return 1; }

template<class T>
typename std::enable_if< !is_signed<T>::value && is_integral<T>::value >
                       , int >::type
f(T)
{ return 2; }
```

The SFINAE approach presents some limitations. First, when the function requires a non trivial set of overloads, the programming process doesn't scale. All overloads must be mutually exclusive. The programmer needs to ensure that every metaprogram leads to a correct instantiation of the function according to the type list. This is an error prone task to perform. In addition, the compilation time of a SFINAE-based overload resolution is linear. The compilation time will not be reasonable for a significant number of functions using this technique over a large list of types.

### 4.1.3 The Tag Dispatching technique

Another solution heralded by the Standard Template Library is to use a technique known as **Tag Dispatching**. This approach relies on a concept-based overloading that will select the most specific function from a set of specializations of a given function. This technique is often used hand- in-hand with traits classes able to associate information with a compile-time entity (a type, integral constant, or address). Here, a **Tag** is a class describing properties that are the expression of a concept. The dispatch decision process will take into account those properties to select the best overload. Inheritance of tags is used to encode the refinement hierarchy of concepts. The overloading mechanism will then pick the most specific overload based on the tag hierarchy.Listing 4.3 shows the previous example implemented with the Tag Dispatching approach.

Listing 4.3: Free function dispatching using Tag Dispatching

```
1  struct unknown_tag                                {};
2  struct fundamental_tag                            {};
3  struct floating_point_tag     : fundamental_tag {};
4  struct integral_tag           : fundamental_tag {};
5  struct signed_integral_tag    : integral_tag    {};
6  struct unsigned_integral_tag  : integral_tag    {};
7
8  template<class T> struct category_of{ typedef unknown_tag type; };
9  template<> struct category_of<float>{ typedef floating_point_tag type; };
10 template<> struct category_of<double>{ typedef floating_point_tag type; };
11 template<> struct category_of<int> { typedef signed_integral_tag type; };
12 template<> struct category_of<unsigned int>
13 { typedef unsigned_integral_tag type; };
14
15 template<class T> int f(T t)
16 { return f(t, typename category_of<T>::type() ); }
17
18 template<class T> int f(T, floating_point_tag const&)
19 { return 0; }
20
21 template<class T> int f(T, integral_tag const&)
22 { return 1; }
23
24 template<class T> int f(T, unsigned_integral_tag const&)
25 { return 2; }
```

By using a hierarchy of tags bound by inheritance, it's possible to make use of the best-match feature of C++ overloading to introduce specializations without requiring them to be mutually exclusive. The `iterator_category` system of standard iterators is a good example of that. But, doing this in a clean, concise, reusable and idiomatic manner presents some difficulties.

### 4.1.4 Concept based overloading

As specified in [55, 95] and introduced in chapter 3, Concepts come as a completely new approach for designing software in C++ . The approach consists in specifying and checking constraints on template arguments within the compiler. These constraints are checked at their point of use. It means that templates errors are caught early in the compilation process. This extension of the C++ language allows the user to specify formal properties on templates that enable a clear and reliable way for expressing function dispatching.

Listing 4.4: Concept overloading

```
1  template< typename T >
2    requires std::is_floating_point<T>
3  T f(T t)
4  { return 0; }
5
6  template< typename T >
7    requires std::is_integral<T>
8  T f(T t)
9  { return 1; }
10
11 template< typename T >
12   requires std::is_integral<T> && std::is_signed<T>
13 T f(T t)
14 { return 2; }
```

Listing 4.4 illustrates a Concept based overload of our example. The `requires` clause here permits to specify a constraint on the template parameter thus resulting to the best selection of the function definition. The `&&` operator is able to introduce a relation between constraints (`||` is also available). In our example, the overload resolution can distinguish that `std::is_integral<T> && std::is_signed<T>` is more specialized than `std::is_integral<T>` resulting to the best overload selection. We can notice that C++ can be easily extended with this language extension but the C++11 committee decided to remove Concepts from the draft standard in July 2009. No official decisions has yet been made for the future of Concepts inside the Standard.

### 4.1.5 Conclusion

The previous methodologies give a way to statically select the best implementation of a function within a set of constraints. But in certain cases this approach is not sufficient or available in the C++ Standard. A precise selection of a function call

relies on more than the information of the types. The properties of the function may impact the selection process and need to be injected during the dispatch phase. In chapter 2 and 3, we also saw the importance of architecture specifications while evaluating a function or a statement in a *DSEL* . Therefore, dispatching a function call requires to take into consideration these factors in order to use the best-match feature of C++ in a generic and powerful way. In section 4.2, we introduce Boost.Dispatch, a C++ template library for function dispatching with a generic tag dispatching system that proposes a solution to the previous limitations.

## 4.2 The Boost.Dispatch Library

In this section, we present a general overview of the abilities of Boost.Dispatch by introducing the main features of the library and showing the most relevant details of its implementation.

### 4.2.1 The *AA-DEMRAL* methodology in Boost.Dispatch

Boost.Dispatch provides a way to implement the *AA-DEMRAL* approach. The main contribution of the library is that function dispatching is aware of an architecture description. Boost.Dispatch achieves that by providing an extended and generic manner of dispatching functions in template contexts through **an extensible hierarchy system**. This component articulates the library through three specialized hierarchies:

- a **hierarchy for types** that expresses the properties of the argument types;
- a **hierarchy for functions** that expresses the properties of the functions;
- a **hierarchy for architectures** that adds architectural information for the dispatch process.

The architecture aware dispatch of a function occurs with the use of an architecture hierarchy. This hierarchy then encodes the architectural information that we want to inject in the dispatch system of the library. With this methodology, Boost.Dispatch is able to aggregate generic components according to type semantic, function semantic and architecture information. Figure 4.1 illustrates the injection of an architecture information inside the *AA-DEMRAL* approach.

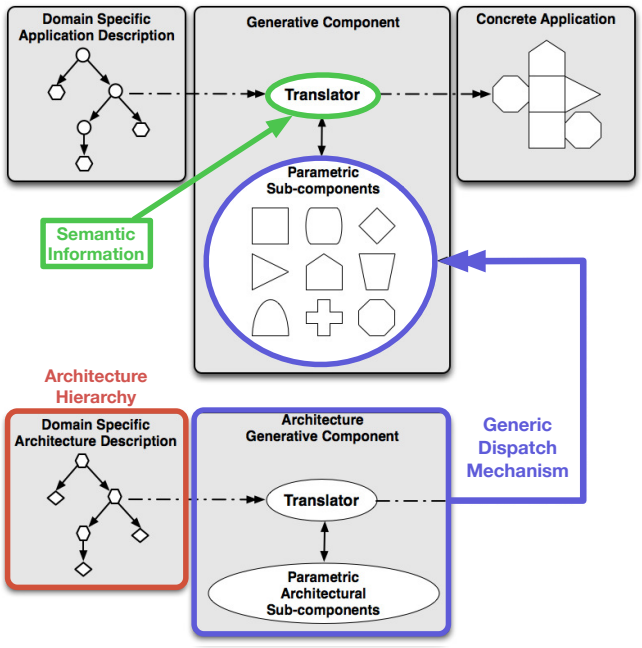



Figure 4.1: Architecture information for dispatch of generic components

The type hierarchy and the function hierarchy provide an entry to add semantic information during the dispatch process. The architecture hierarchy is able to inject an architecture description inside the dispatch process. The design of the library is then semantic and architectural aware. The Hierarchy Concept is the main component that articulates the library and it will be detailed in the next section.

### 4.2.2 The Hierarchy Concept

The Concept of `Hierarchy` is the key of the dispatching system. It provides an idiomatic way to define inheriting category tags. These tags will then embed arbitrary levels of intentionality for the dispatch mechanism.

Listing 4.5 presents the definition of the Hierarchy Concept.

Listing 4.5: The Hierarchy Concept

```
1 struct H : P
2 {
3     typedef P parent;
4 };
```

A model `H` of `Hierarchy` is nothing more than an empty type used to identify a category of types. It must inherit from another model of `Hierarchy` `P`. Multiple inheritances are discouraged as it easily leads to ambiguities while computing the

hierarchy of the type being hierarchized. It must also provide a `parent` typedef in order to allow composite hierarchies to be built around it. All hierarchies must inherit directly or indirectly from `boost::dispatch::meta::unspecified_<T>`, with `T` a concrete type (preferably the one being hierarchized).

For its built-in hierarchies, Boost.Dispatch chooses to make them templates (see listing 4.6), with the actual type being hierarchized as the template parameter. This allows to select the parent hierarchy according to the type, removing some of the limitations of single inheritance. Embedding the type inside the hierarchy also enables to use it directly for declaring template overloads in place of the real arguments.

Listing 4.6: The Hierarchy Concept for template types

```
1 template<typename T>
2 struct H<T> : P<T>
3 {
4     typedef P<T> parent;
5 };
```

### 4.2.3 Compile Time Hierarchy Deduction

This hierarchy system contains ready to use hierarchies that can be extended by the user. Listing 4.7 shows the free function dispatching example from section 4.1 rewritten with Boost.Dispatch.

Listing 4.7: Boost.Dispatch in action

```
1  #include <boost/dispatch/meta/hierarchy_of.hpp>
2  using namespace boost::dispatch;
3
4  template<class T> int f(T t)
5  { return f(t, typename meta::hierarchy_of<T>::type() ); }
6
7  template<class T> int f(T, scalar_< floating_<T> > const&)
8  { return 0; }
9
10 template<class T> int f(T, scalar_< integer_<T> > const&)
11 { return 1; }
12
13 template<class T> int f(T, scalar_< unsigned_<T> > const&)
14 { return 2; }
```

The metafunction `hierarchy_of` at line 5 will compute the hierarchy of the type `T` in the built-in hierarchy of Boost.Dispatch. It will permit the selection of one of the three specializations in the example. The first one at line 7 will be selected when `T` is a single or double precision floating point number. The second specialization at line 10 applies to any integral type, including unsigned ones. However, since there is also a specialization for unsigned types at line 13, the latter gets preferred, since `unsigned_` is a refinement of `integer_` in the built-in hierarchy.

We saw that hierarchies are built through the use of inheritance. This allows to build a partially ordered lattice. This lattice expresses the category of a given type into all its potential enclosing type sets. To be able to find the root of a hierarchy in the lattice, we need to pass the parent type to the next node of the lattice. This is done by making the hierarchies template as presented in listing 4.6.

Listing 4.8: `hierarchy_of` implementation

```
template<class T, class Origin = T>
struct  hierarchy_of
  : details::hierarchy_of< T
                         , typename remove_reference<Origin>::type
                         >
{};

//=====================================
// specialization for fundamental types
//=====================================
namspace details{
  template<class T,class Origin>
  struct  hierarchy_of< T
                      , Origin
                      , typename boost::
                        enable_if< boost::is_fundamental<T> >::type
                      >
  {
    typedef typename remove_const<Origin>::type stripped;

    typedef typename mpl::if_< is_same< T, stripped >
                             , stripped
                             , Origin>::type origin_;

    typedef scalar_<typename property_of<T, origin_>::type>  type;
  };
}
```

Listing 4.8 shows the implementation of the `hierarchy_of` metafunction. The specialization corresponds to the built-in hierarchy of types. This specialization is used when the type to compute is a fundamental type, which corresponds to every C++ standard based type. The metafunction will then return the `scalar_` hierarchy with the corresponding property of the type. For example, `hierarchy_of<float>::type` will return `scalar_< single_<int> >`.

### 4.2.4 Built-in Hierarchies

In this section we present the built-in hierarchies available in Boost.Dispatch. To fully illustrate the capabilities of the library, we will focus on constructing the call of a function performing the sum of its arguments in a generic context. This function is simply calculating the addition of two arguments and is named `plus`. Listing 4.9 shows the top-level `plus` function that performs a hierarchized call through the use of the `hierarchy_of` metafunction.

Listing 4.9: A hierarchized call of `plus`

```
template <class A0>
A0 plus(A0 const& a0, A0 const& a1)
{
  return impl::plus(a0, a1, typename meta::hierarchy_of<A0>::type() );
}
```

#### 4.2.4.1 Hierarchy for functions

With Boost.Dispatch the first step consists in declaring a generic tag for the function itself. This tag identifies the function in generic contexts and it needs to model the Hierarchy Concept. The function tag is tied to the function properties. The `plus` function is an elementwise function so we can introduce a specific elementwise evaluation context in the function hierarchy.

Listing 4.10: The `plus` function identifier

```
// elementwise_ hierarchy
template<class T>
struct elementwise_ : unspecified_<T>
{
  typedef unspecified_<T> parent;
};

// plus_ models :
namespace tag
{
  struct plus_ : elementwise_ < plus_ >
  {
    typedef elementwise_< plus_ > parent;
  };
}
```

Listing 4.10 shows how to introduce a new elementwise hierarchy and how to express a function tag modeling this hierarchy. Further function properties can then be added through this hierarchy like reduction operations for example.

#### 4.2.4.2 Hierarchy for scalar types

The library gives a built-in hierarchy for dispatching on scalar types. According to the types passed as arguments to the function, Boost.Dispatch will compute the hierarchy of these types through the metafunction `hierarchy_of` and select the best call available. The hierarchy of a built-in fundamental type T is the composite `scalar_< typename property_of<T>::type >`. `property_of` computes the property of the type being hierarchized. Built-in properties in Boost.Dispatch are tied to the intrinsic semantic of C++ fundamental types (*i.e.* `int`, `float`, `double`, `bool`, etc). Properties are decoupled from `scalar_` so that it is easy to create new hierarchies such as `foo_< integer_<T> >` etc.

A list of all properties available and how they relate to each other is listed in figure 4.2.

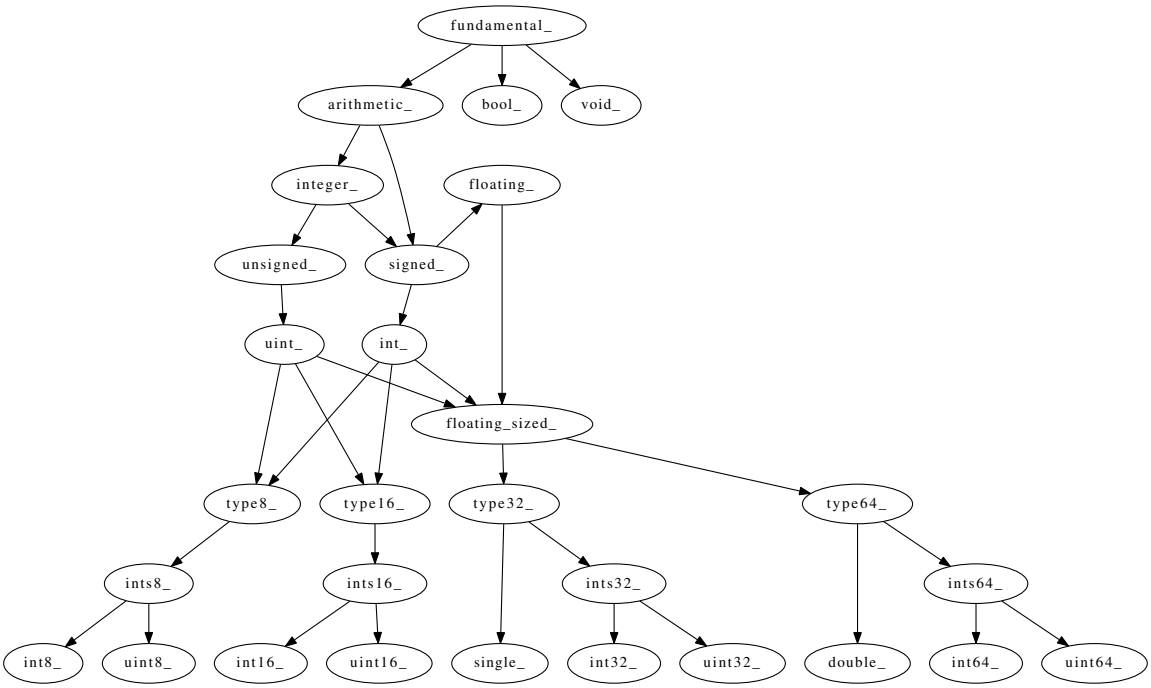



Figure 4.2: The properties of the built-in scalar hierarchy

Our `plus` function doesn't require any particular behavior as the '+' operator in C++ can handle every arithmetic scalar types. So we can write a function that takes care of every possible call of the `plus` function with scalar types by specifying the following composite: `scalar_< arithmetic_<T> >`. Listing 4.11 presents the resulting function.

Listing 4.11: Dispatching `plus` for scalar types

```
1 namespace impl
2 {
3   template <class A0>
4   A0 plus(A0 const& a0, A0 const& a1, scalar_< arithmetic_<A0> > const&)
5   {
6     return a0+a1;
7   };
8 }
```

With all the type properties available in Boost.Dispatch, we can easily imagine more complex functions requiring a very fine dispatch strategy depending on their arguments properties.

#### 4.2.4.3 Hierarchy for architectures

Another ability of the library is to be able to select a function call with a specific architecture information. In section 3.3.2 we discussed the necessity of injecting architectural specification during the evaluation of a *DSEL* . Boost.Dispatch proposes to specify a `Site` information. Boost.Dispatch `Sites` are computed as hierarchies and provide a way to explicitly declare function overloading for dedicated architecture contexts.

By default, the library gives two architecture aware evaluation contexts. The first one is the `formal_` site that defines the most abstracted evaluation context. This context is used in the case of high-level code transformation functions. Inheriting from the `formal_` site, the `cpu_` site defines the CPU based evaluation context for functions. This context is used when no specific architecture informations are available or required by a function. Function specializations under the `cpu_` context are usually used as common architecture independent implementation. Boost.Dispatch has the ability to automatically compute the site for a function tag. If different implementations are available, the library will compute the best site for the evaluation context. This is done by using the `default_site` metafunction that takes as parameter the tag of a function.

Boost.Dispatch is able to aggregate information about the underlying architecture. The library gathers information from compiler macros that are activated by compile flags or from user defined macros. The library then organizes hierarchically these informations and provides an automated way of computing the default evaluation context of a function. The `default_site` metafunction performs such a computation.

If we go back to our `plus` example, the scalar version of the `plus` functor is dispatched through the `cpu_` site like presented in listing 4.12.

Listing 4.12: Dispatching `plus` on default `cpu_` site

```
template <class A0> A0 plus( A0 const& a0, A0 const& a1)
{
  typedef default_site<tag::plus_>::type site;
  return impl::plus( a0, a1
                   , typename meta::hierarchy_of<A0>::type()
                   , typename meta::hierarchy_of<site>::type()
                   );
}

namespace impl
{
  template <class A0>
  A0 plus( A0 const& a0, A0 const& a1
         , scalar_< arithmetic_<A0> > const&, cpu_ const&
         )
  {
    return a0+a1;
  };
}
```

To push further the architecture aware ability of Boost.Dispatch, we will add an implementation of our `plus` function for a specific SIMD extension. SIMD extensions as introduced in chapter 2 work with wide registers able to store multiple data and their computation unit has the ability of applying the same operation on the data stored in these registers. For example, SSE can perform an addition on four single precision floating point numbers at the same time. Our `plus` function could take advantage of such an extension when it is available. To achieve this, we first need to provide a new hierarchy for SIMD types. Listing 4.13 shows the inheritance from the `unspecified_` tag to add an `simd_` hierarchy tag in the Boost.Dispatch tag dispatching mechanism.

Listing 4.13: SIMD type hierarchy

```
namespace tag{
  template<class T, class X>
  struct simd_ : simd_< typename T::parent, X >
  {
    typedef simd_< typename T::parent, X >  parent;
  };

  template<class T, class X>
  struct  simd_<T, X>
        : unspecified_< typename property_of<T>::type, X >
  {
    typedef unspecified_< typename property_of<T>::type, X > parent;
  };
}
```

A specific type needs to be used when working with SIMD registers. The SSE extension requires the use of the `__m128` type to store four `float` values in a SIMD register. We need to make Boost.Dispatch aware of the hierarchy of this type to activate the dispatch on SIMD floating point registers. Listing 4.14 shows the specialization of the `hierarchy_of` metafunction to register the `__m128` type in the type hierarchy.

Listing 4.14: Making `hierarchy_of` aware of SIMD register

```
template<class T, class Origin>
struct hierarchy_of< __m128, Origin>
{
  typedef simd_< single_<Origin>, sse_> type;
};
```

After adding this new entry in the type hierarchy, we need to register a new site for the SSE SIMD extension in the architecture hierarchy. First, we add an entry for a `simd_` site. This can be done by inheriting from the `cpu_` site tag. After this step we can ramify the new `simd_` site to handle multiple SIMD extensions. For the SSE family, a `sse_` tag that inherits from the `simd_` tag will be the entry point of the SSE extension family. Then, we would be able to continue the inheritance scheme with a `sse2_` hierarchy tag. Listing 4.15 summarizes the hierarchy tree of the new sites.

Listing 4.15: Hierarchy class for SSE

```
namespace tag{
  struct simd_ : cpu_  { typedef cpu_  parent; };
  struct sse_  : simd_ { typedef simd_ parent; };
}
```

Now, we can add a new version of our `plus` function for the `sse_` site. This version adds the `sse_` tag to our implementation. The '+' operation is then performed by the `_mm_add_ps` intrinsic. Listing 4.16 presents this version.

Listing 4.16: Dispatching `plus` for SSE

```
namespace impl
{
  template <class A0>
  A0 plus( A0 a0
         , A0 a1
         , simd_< single_<A0> > const&
         , tag::sse_ const&
         )
  {
    return _mm_add_ps( a0 , a1 ) ;
  }
};
```

Another feature of the library is that the type hierarchy enables to factorize the code of dispatched functions. For example, if a `multiplies` function is implemented like our `plus` function, we may want to reuse all the specific SIMD and scalar implementations of `multiplies` to implement a `square` function. `square` just needs to reuse `multiplies` in its implementation and `multiplies` will then be dispatched to the best implementation available according to the Boost.Dispatch hierarchies. This can be achieved by using the `generic_` entry in the type hierarchy of Boost.Dispatch.

Listing 4.17: `generic_` hierarchy for code reuse

```
namespace tag{
  template<class T,class X>
  struct  simd_< T, X >
        : generic_< typename property_of<T>::type >
  {
    typedef generic_< typename property_of<T>::type > parent;
  };
}
```

Listing 4.17 presents the `simd_` hierarchy that now inherits from the `generic_` entry point. In the case of a Boost.Dispatch `square` function, only the implementation presented in listing 4.18 is required. The Boost.Dispatch `multiplies` function will then perform its own dispatch through the library. Here, the `A0` type models an `arithmetic_` that will be passed to `multiplies`. If `A0` is a SIMD type, the SIMD implementation available will be called.

Listing 4.18: A `generic_` function call example

```
namespace impl
{
  template <class A0>
  A0 square( A0 const& a0
           , generic_< arithmetic_<A0> > const&
           , cpu_ const&
         )
  {
    return multiplies(a0,a0);
  };
}
```

#### 4.2.4.4 Hierarchy for Boost.Proto ASTs

The library can also dispatch functions that manipulates Boost.Proto ASTs. The top-level of the Boost.Proto expression hierarchy is the `ast_` hierarchy tag (see listing 4.19). It directly inherits from the `unspecified_` tag which means that all the Boost.Proto expressions passed as arguments match this hierarchy. A Boost.Proto domain can still be used by the dispatch system. The `ast_` hierarchy is then refined by the `node_` hierarchy.

Listing 4.19: The `ast_` hierarchy

```
// T is the AST being hierarchized
// D is the AST domain
template<class T, class D>
struct ast_ : unspecified_<T>
{
  typedef unspecified_<T> parent;
};
```

The `node_` hierarchy represents a tagged Boost.Protoexpression node. Each Boost.Proto node encodes its tag and arity. A node tag type in Boost.Proto describes the operation that created the node in the AST and the arity corresponds to the number of children of the node. Boost.Dispatch provides entries so that the dispatch system can be aware of these informations. Thus, the `node_` hierarchy takes into consideration the Boost.Proto arity, a Boost.Proto tag hierarchy and an expression semantic hierarchy. The expression semantic hierarchy encodes the information of the concrete type held by the Boost.Proto expression which means the actual hierarchy of this concrete type. `node_` is presented in listing 4.20.

Listing 4.20: The `node_` hierarchy

```
// T is the expression semantic hierarchy
// Tag is the expression node tag hierarchy
// N is the expression arity
// D is the expression domain hierarchy
template<class T, class Tag, class N, class D>
struct node_ : node_<T, typename Tag::parent, N, D>
{
  typedef node_<T, typename Tag::parent, N, D> parent;
};
```

The parent hierarchy of `node_` is then computed according to the semantic hierarchy of the Boost.Proto tag. Figure 4.3 illustrates the parent hierarchy computation of '+' Boost.Proto node. Once the `unspecified_` hierarchy is reached, the parent hierarchy of `node_` is computed as a `ast_` hierarchy.

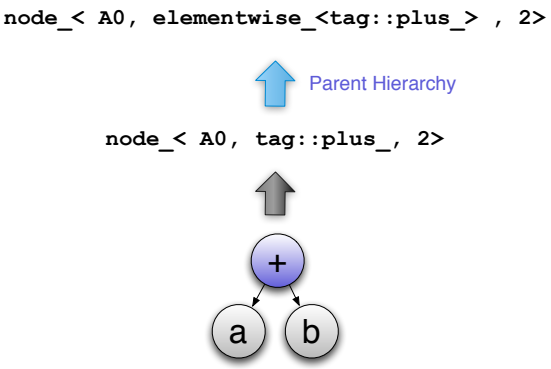



Figure 4.3: `plus_` node example

The last level of refinement in the AST hierarchy is the `expr_` hierarchy. This hierarchy represents a tagged Boost.Proto expression with a specific arity and a expression semantic hierarchy. Listing 4.21 presents this hierarchy. The parent hierarchies of `expr_` are computed by walking the parent hierarchy of the expression semantic hierarchy. Once hitting the `unspecified_` hierarchy, `expr_` parent is computed as a `node_` hierarchy.

Listing 4.21: The `expr_` hierarchy

```
// T is the expression semantic hierarchy
// Tag is the expression node tag hierarchy
// N is the expression arity
template<class T, class Tag, class N>
struct expr_ : expr_<typename T::parent, Tag, N>
{
  typedef expr_<typename T::parent, Tag, N>  parent;
};
```

### 4.2.5 Common API

When working with more than one function, this technique introduces verbosity in the code. Boost.Dispatch is able to automate this hierarchical approach by providing a common API. It is composed of macros that simplify the implementation of dispatched functions.

The first helper macro is `BOOST_DISPATCH_HIERARCHY_CLASS`. This macro facilitates the registration of Boost.Dispatch hierarchy classes. Listing 4.22 shows the registration of our previous SIMD sites.

Listing 4.22: Hierarchy class macro

```
namespace tag{
  BOOST_DISPATCH_HIERARCHY_CLASS(simd_, boost::dispatch::tag::cpu_);
  BOOST_DISPATCH_HIERARCHY_CLASS(sse_, simd_);
}
```

The next macro is `BOOST_DISPATCH_FUNCTION_IMPLEMENTATION`. It generates a dispatch-based function implementation. Listing 4.23 details this macro.

Listing 4.23: Implementation Macros of Boost.Dispatch

```
// Tag is the function tag
// Name is the name of the function
// N is the number of arguments for the function.
BOOST_DISPATCH_FUNCTION_IMPLEMENTATION(Tag, Name, N)
```

Our function example of `plus` can now be written as in listing 4.24

Listing 4.24: `plus` registration

```
template<class T>
namespace tag
{
  struct plus_ : elementwise_ < plus_ > {};
}

BOOST_DISPATCH_FUNCTION_IMPLEMENTATION(tag::plus_, plus, 2)
```

The `BOOST_DISPATCH_IMPLEMENT` macro (see listing 4.25) takes care of generating the Boost.Dispatch `functor` that handles the tag dispatching mechanism. It is also called an implement in the context of the library. The user can easily specify a special namespace where the function will be declared. Then the function tag and the site tag are passed to the macro. The user can finally name the template arguments of the function and specify in a preprocessor sequence the hierarchy tags associated with these arguments.

Listing 4.25: Extension Macros of Boost.Dispatch

```
// NS is the namespce for the implement.
// Tag is the function tag
// Site is the site tag
// Types are the template parameters for the arguments
// Seq is waiting for a sequence of types hierarchy tags
// Cond is a static condition that can include or not an implement

BOOST_DISPATCH_IMPLEMENT(NS, Tag, Site, Types, Seq)
```

Now, we can update our `plus` example and specify two Boost.Dispatch implementations. Listings 4.26 and 4.27 present these implementations.

Listing 4.26: Scalar overload with macros

```
BOOST_DISPATCH_IMPLEMENT( my_plus_namespace
                        , tag::plus_
                        , tag::cpu_
                        , (A0)
                        , (scalar_< arithmetic_<A0> >)
                          (scalar_< arithmetic_<A0> >)
                        )
{
  typedef A0 result_type;

  A0 operator(A0 const& a0, A0 const& a1) const
  { return a0 + a1; }
};
```

Listing 4.27: SIMD overload with macros

```
BOOST_DISPATCH_IMPLEMENT( my_plus_namespace
                        , tag::plus_
                        , tag::sse_
                        , (A0)
                        , ((simd_<single_<A0>,tag::sse_>))
                          ((simd_<single_<A0>,tag::sse_>))
                        )
{
  typedef A0 result_type;

  A0 operator(A0 a0, A0 a1) const
  { return _mm_add_ps(a0,a1); }
};
```

We can finally call our `plus` function like in listing 4.28. In the next section, we will present implementation details of the library to clarify the behavior of Boost.Dispatch.

Listing 4.28: SIMD overload with macros

```
int main()
{
  float a = 3.0 , float b = 5.0;
  float c = plus(a,b); // The scalar version is called here.

  __m128 vec0  = {1.0 , 2.0, 3.0, 4.0};
  __m128 vec1  = {11.0,12.0,13.0,14.0};
  __m128 r = plus(vec0, vec1); // The SSE version is called here.

  return 0;
}
```

## 4.3 Conclusion

Tag dispatching is a powerful technique used in the C++ standard library. Boost.Dispatch pushes further this technique by adding two entries besides the classical approach: tag dispatching on function properties and tag dispatching on architecture specifications. The library gathers these three entries through a generic tag dispatching system that allows the user to have a fine grain for selecting function overloads. Boost.Dispatch main feature is the integration of an architecture aware dispatching mechanism that is the first step to an *AA-DEMRAL* approach. Boost.Dispatch can be used in the context of tools development and provides a solution for injecting architectural specification inside an evaluation process.

Now, we will see how Boost.Dispatch helps the development of architecture aware tools by presenting two parallel programming tools:

- Boost.SIMD for programming SIMD extensions;
- $NT^2$ for programming small-scale systems (multicores coupled with SIMD extensions).

Chapter 5

# The Boost SIMD Library

## Contents


**5.1 Hardware context and software challenges . . . . . 62**
**5.2 The Boost.SIMD Library . . . . . 65**
5.2.1 SIMD register abstraction . . . . . 65
5.2.2 Predicates abstraction . . . . . 69
5.2.3 Range and Tuple interface . . . . . 69
5.2.4 Supported functions . . . . . 71
5.2.5 Shuffling operations . . . . . 71
**5.3 C++ Standard integration . . . . . 74**
5.3.1 Aligned allocator . . . . . 74
5.3.2 SIMD Iterator . . . . . 75
5.3.3 SIMD Algorithms . . . . . 77
**5.4 Case Analysis: Generic SIMD code generation . . . . . 77**
5.4.1 Scalar version of the `dot` function . . . . . 77
5.4.2 Transition from scalar to SIMD code . . . . . 78
5.4.3 Building a SIMD loop nest . . . . . 78
5.4.4 Preparing the data . . . . . 79
5.4.5 Resulting code generation . . . . . 81
5.4.6 Choosing SIMD extensions at runtime . . . . . 82
**5.5 Implementation . . . . . 85**
5.5.1 Function Dispatching . . . . . 85
5.5.2 AST Manipulation with Boost.Proto . . . . . 86
**5.6 Implementation Discussion . . . . . 90**
5.6.1 Function inlining and ABI issue . . . . . 90
5.6.2 Unrolling . . . . . 90
**5.7 Benchmarks . . . . . 91**
5.7.1 Basic Kernels . . . . . 91
5.7.2 Black and Scholes . . . . . 94
5.7.3 Sigma-Delta Motion Detection . . . . . 95
5.7.4 The Julia Mandelbrot Computation . . . . . 97
**5.8 Conclusion . . . . . 99**

In chapter 4 we introduced Boost.Dispatch, a function dispatching library with architecture aware dispatching capabilities. We want to demonstrate the ability of this library to be used in the development of a high level programming tool for fine grain parallelism. SIMD programming relies on special SIMD instruction sets that are working with wide data registers. In this context, Boost.Dispatch needs to provide a very low overhead to guarantee the performance of an instruction level programming model like SIMD. This chapter[1] presents the development of a high level programming tool for SIMD extensions: The Boost.SIMD Library [45, 44].

In this chapter, we introduce the challenges of such a library followed by its API. Afterwards we present the concepts on which such a library relies and its implementation. Then, we detail the technical choices we made for the development of Boost.SIMD by describing the full SIMD code generation process. Finally, we conclude this chapter with a discussion on the performance of the library.

## 5.1 Hardware context and software challenges

We introduced in section 2.1.1 the principle of SIMD extensions. We saw that these extensions allow to accelerate applications with a data parallelism scheme. Table 5.1 gives a full overview of these hardware extensions with the size of their dedicated SIMD registers.

For example, the AVX extension introduced in 2011 enhances the x86 instruction set for the Intel Sandy Bridge and AMD Bulldozer micro-architectures by providing a distinct set of 16 256-bit registers. Similarly, the Intel MIC [39] (Many Integrated Core, now known as Xeon Phi) architecture embeds 512-bit SIMD registers. Intel improved AVX with some new instructions and launched AVX 2.0 late 2013. The forthcoming extension from Intel is AVX-512 that will be introduced in the next generation of Xeon Phi, Knights Landing coming in 2014. Using SIMD processing units can also be mandatory for performance on such systems as demonstrated by the NEON and NEON2 ARM extensions [61] or the CELL-BE processor by IBM [69] which SPUs were designed as a SIMD-only system. IBM also introduced in 2012 the QPX [49] extension available on the third supercomputer design of the Blue Gene series. QPX works with 32 256-bit registers.

However, programming applications that take advantage of available SIMD extension on a given hardware remains a complex task. In addition, working with multimedia extensions implies a significant amount of code writing to handle most of the increasingly number of SIMD extensions available today. We previously detailed in chapter 2 section 2.2.1 the existing solutions for programming these extensions. Programmers that use low-level intrinsics have to deal with a verbose programming style due to the fact that SIMD instructions sets cover a few common functionali-

[1] This chapter is extended from the work published in [45, 44].

Table 5.1: SIMD extensions in modern processors

| Manufacturer | Extension | Registers size & number | Instructions |
|---|---|---|---|
| Intel | SSE | 128 bits - 8 | 70 |
| | SSE2 | 128 bits - 8/16 | 214 |
| | SSE3 | 128 bits - 8/16 | 227 |
| | SSSE3 | 128 bits - 8/16 | 227 |
| | SSE4.1 | 128 bits - 8/16 | 274 |
| | SSE4.2 | 128 bits - 8/16 | 281 |
| | AVX | 256 bits (float only)- 8/16 | 292 |
| | AVX2 + FMA3 | 256 bits - 8/16 | 297 |
| AMD | SSE4a | 128 bits - 8/16 | 231 |
| | XOP | 128 bits - 8/16 | 289 |
| IBM Motorola | VMX | 128 - 32 | 114 |
| | VMX128 | 128 bits - 128 | |
| | VSX | 128 bits - 64 | |
| | QPX | 256 bits - 32 | |
| | SPU | 128 bits - 128 | |
| ARM ARM | NEON | 128 bits - 16 | 100+ |
| | NEON2 | | |

ties, requiring to bury the initial algorithms in architecture specific implementation details. Furthermore, these efforts have to be repeated for every different extension that one may want to support, making design and maintenance of such applications very time consuming. These restrictions account for the small amount of solutions facing this challenge.

Due to the factors previously mentioned, providing high level tools able to mix a sufficient abstraction with performance is a nontrivial task that needs to solve important challenges. Developing a library like Boost.SIMD implies several goals to face properly the design of a high level programming tool for SIMD extensions.

- **A generic user interface**

  The first limitation faced by application developers is the multiplicity of SIMD register types. Table 5.2 shows a glimpse of the amount of data types available. Furthermore, all the intrinsics are qualified by each data type due to the low level C programming model of such extensions. This restriction forces the programmer to write different versions of the algorithm according to the targets he wants to support.

  Thus, the algorithm is duplicated for each data type with the correct intrinsic calls. We can see the limitation of this approach in terms of development time and maintenance of an application. By contrast, a generic approach expresses the algorithms and the data structures as abstract entities. The first challenge of such an approach is to design a high level user interface to

Table 5.2: SIMD types available

| Manufacturer | Extension | Floating registers | Integer registers |
|---|---|---|---|
| Intel/AMD | SSE | __m64, __m128 | Not Supported |
| | SSE2,3,4x | __m64, __m128, __m128d | __m128i |
| | AVX | __m256, __m256d | Not Supported |
| | AVX2 + FMA3 | __m256, __m256d | __m256i |
| IBM/Mot. | AltiVec flavored | __vector float | __vector int, __vector char, etc |
| ARM | NEON | float32x4_t, float32x2_t | int32x4_t, int32x2_t, etc |

keep a strong readability of the code and bury the verbosity of the classic SIMD programming style. By design, a generic library in C++ offers a high level of expressiveness and relies on Concepts that help the library in its code generation phase. For Boost.SIMD, we need to extract axioms [38] from the current SIMD programming model to correctly express and define a set of C++ Concepts. Afterward we will be able to use those Concepts to provide correct data structure and algorithm abstractions that fit with the expressibility of a fine grain data parallel problem.

- **C++ standard integration**

  A lot of existing code relies on the C++ Standard Template Library (STL). Most of them should be able to take advantage of the speedup provided by SIMD extensions. The STL is constructed as a generic library over the following trio of Concepts : Algorithms - Container - Iterators. Switching from a STL code to a fully vectorized version of it must stay straightforward for the user. To accomplish this integration properly, STL Concepts needs to be refined to satisfy SIMD based axioms. On top of that, Boost.SIMD needs to propose a standard like interface with wrappers able to adapt standard components and also a nice Boost integration for the use of Fusion [31] and MPL libraries.

- **Effective code generation**

  The architectural improvements provided by SIMD extensions leads to a significant speedup that we want to reach with Boost.SIMD. Despite the introduction of a generic interface, Boost.SIMD needs to keep the performance of the generated code close to the performance of a "hand written" code. The impact of the generic interface and the code generation engine must be low for the reliability of the library.

- **SIMD idioms**

  From the beginning, multimedia extensions were designed to accelerate massively data parallel problems like image processing or vector based code but, working with wide data registers does not solve everything. Storing data in SIMD registers can impact the algorithm in a way that predicates handling

or iteration patterns need to be rethought in a vectorized way. For example, some algorithms are not accessing contiguous memory blocks and this leads to a non trivial access pattern when working in SIMD. Moreover, when working with SIMD registers the user may want to rearrange the data in the register. These SIMD idioms require to be expressed, apart from standard based components.

- **Extensibility and maintainability**

  Developing a library remains a complex task in terms of easy extensibility and maintainability. Boost.SIMD is a tool giving access to multi-architectural support over SIMD extensions. The Boost.SIMD framework must be designed in a proper way to handle a straightforward addition of a new SIMD extension, a new data type or a new algorithm.

## 5.2 The Boost.SIMD Library

Boost.SIMD aims at bridging the lack of proper abstractions over the usage of SIMD registers. This abstraction should not only provide a portable way to use hardware-specific registers but also enable the use of common programming idioms when designing SIMD-aware algorithms. To achieve this, Boost.SIMD implements **an abstraction of SIMD registers** that allow the design of portable algorithms. In addition, a large set of functions are covering the classical set of mathematical functions and utility functions. This section details the components of the library and shows step by step the interface of these components along with their behavior.

### 5.2.1 SIMD register abstraction

The first level of abstraction introduced by Boost.SIMD is the `pack` class. For a given type `T` and a given static integral value `N` (`N` being a power of 2), a `pack` encapsulates the best type available to store a sequence of `N` elements of type `T`. For arbitrary `T` and `N`, this type is simply `std::array<T,N>` but when `T` and `N` matches the type and width of a SIMD register, the architecture-specific type used to represent this register is used instead. This semantic provides a way to use arbitrarily large SIMD registers on any system and let the library selects the best vectorizable type to handle them. By default, if `N` is not provided, `pack` will automatically select a value that will trigger the selection of the native SIMD register type. Moreover, by carrying informations about its underlying scalar type, `pack` enables proper instruction selection even when used on extensions (like SSE2 and above) that map all integral type to a single SIMD type (`__m128i` for SSE2).

`pack` handles these low-level SIMD register types as regular objects with value semantics, which includes the ability to be constructed or copied from a single scalar value, list of scalar values, iterator or range. In each case, the proper register loading strategy (splat, set, load or gather) will be issued.

Listing 5.6 illustrates how the `pack` register abstraction works.

Listing 5.1: Working with `pack`, computing a SIMD register full of 42

```
#include <iostream>
#include <boost/simd/sdk/simd/io.hpp>
#include <boost/simd/sdk/simd/pack.hpp>
#include <boost/simd/include/functions/splat.hpp>
#include <boost/simd/include/functions/plus.hpp>
#include <boost/simd/include/functions/multiplies.hpp>

int main(int argc, const char *argv[])
{
  typedef pack<float> p_t;

  p_t res;
  p_t u(10);
  p_t r = boost::simd::splat<p_t>(11);

  res = (u + r) * 2.f;

  std::cout << res << std::endl;

  return 0;
}
```

`pack` supports multiple constructors. It is copy and default constructible and also supports different methods to initialize a `pack`'s content (loading strategies).

A typedef statement is used before the declaration of the packs for brevity. These declarations include a so-called splatting constructor that takes one scalar value and replicates it in all elements of the pack.

```
p_t u(10);
```

This is equivalent to the constructor on the following line:

```
p_t r = boost::simd::splat<p_t>(11);
```

The user can also initializes every element of the `pack` itself by enumerating them.

```
pack<float> r(11,11,11,11);
```

This constructor makes the strong assumption that the size of the `pack` is correct. Unless required, it is always better to try not to depend on a fixed size for `pack`.

Once initialized, operations on `pack` instances are similar to operations on scalar as all operators and standard library math functions are provided. A simple pattern make those functions and operators available: if function foo is used, you need to include `boost/simd/include/functions/foo.hpp`. Here, we include `plus.hpp` and `multiplies.hpp` to be able to use `operator+` and `operator*`.

```
res = (u + r) * 2.f;
```

Note that type checking is stricter than one may expect when scalar and SIMD values are mixed. Boost.SIMD only allows mixing types of the same scalar kind, i.e reals with reals or integers with integers. Here, we have to multiply by `2.f` and not simply `2`. We need to keep in mind that fused operations are available for SIMD extensions and in the case of such a statement, we have to generate a call to a fused multiply and add instruction if the targeted extension supports it.

Finally, we display the `pack` content by using `operator«` provided by the `boost/simd/sdk/simd/io.hpp` header file.

```
18 std::cout << res << std::endl;
```

- **Compiling the code**

  The compilation of the code is rather straightforward: just pass the path to Boost.SIMD and use your compiler options to activate the desired SIMD extension support.

  For example, on gcc:

```
g++ my_code.cpp -O3 -o my_code -I/path/to/boost/ -msse2
g++ my_code.cpp -O3 -o my_code -I/path/to/boost/ -mavx
g++ my_code.cpp -O3 -o my_code -I/path/to/boost/ -maltivec
```

  Some compilers, like Microsoft Visual Studio, don't propagate the fact that a given architecture specific option is triggered. In this case, you need to also defines an architecture specific preprocessor symbol, for example:

```
cl /Oxt /DNDEBUG /arch:SSE2 /I$BOOST_ROOT my_code.cpp
cl /Oxt /DNDEBUG /DBOOST_SIMD_HAS_SSE4_2_SUPPORT
/I$BOOST_ROOT my_code.cpp
```

- **The result**

  We can then have a look at the program's output that should look like:

```
{42,42,42,42}
```

  Now, let's have a look at the generated assembly code for SSE2:

```
movaps 0x300(%rip),%xmm0
addps 0x2e6(%rip),%xmm0
mulps 0x2ff(%rip),%xmm0
movaps %xmm0,(%rsp)
```

  We correctly emitted `*ps` instructions. Note that the abstraction introduced by `pack` does not incur any penalty. Now we can look at the AVX generated assembly:

```
vmovaps 0x407(%rip),%ymm0
vaddps 0x3dc(%rip),%ymm0,%ymm0
vmulps 0x414(%rip),%ymm0,%ymm0
vmovaps %ymm0,(%rsp)
```

  We can see that Boost.SIMD generates again the proper AVX code with the call to AVX instructions with `ymm` registers. In the case of Altivec, we want to generate a call to a fused multiply and add operation as it provides such an instruction. The generated assembly code is the following:

```
vspltw v12,v12,0
vspltw v13,v13,0
vspltw v0,v1,0
vmaddfp v1,v12,v13,v0
stvx v1,r10,r9
```

  We can see that we correctly splat the data into SIMD registers and then call the FMA (Fused Multiply Add) instruction: `vmaddfp`.

### 5.2.2 Predicates abstraction

Comparisons between SIMD vectors yield a vector of boolean results. While most SIMD extensions store a $0\backslash\sim0$ bitmask in the same register type as the one used in the comparison, some like Intel Phi or QPX have a special register bank for those types. The Intel MIC has a dedicated 16-bit register to handle the result of the comparison. QPX comparisons put $-1.lf$ or $+1.lf$ inside a QPX register. To handle architecture-specific predicates, an abstraction over boolean values and a set of associated operations must be given to the user. The `logical` class encapsulates the notion of a boolean value and can be combined with `pack`. Thus, for any type `T`, an instance of `pack< logical<T> >` encapsulates the proper SIMD register type able to store boolean values resulting from the application of a SIMD predicate over a `pack<T>`. Thus, the comparison operators will return a `pack<logical<T> >`. The branching is performed by a dedicated function `if_else` that is able to vectorize the branching process according to the target architecture.

Unlike scalar branching, SIMD branching does not perform branching prediction. All branches of an algorithm are evaluated before the result is selected. Listing 5.7 shows a simple example of branching condition with `pack`.

Listing 5.7: Branching example

```
1 pack<int> a(3), b(1), r;
2 pack<int> inc(0,1,2,3), dec(3,2,1,0);
3 r = if_else(inc > dec, a, b); // r = [1,1,3,3]
```

In addition to the classic `if_else` structure, Boost.SIMD provides specific predicate functions that can be optimized. These functions are optimized depending on the types they work with. For example, the `seldec` and `selinc` functions respectively decrement or increment a `pack` according to the result of a comparison and their implementations for integer types rely on a masking technique.

### 5.2.3 Range and Tuple interface

By providing STL-compliant `begin` and `end` member functions, `pack` can be iterated at runtime as a simple container of `N` elements. In addition, the square brackets operator is available on `pack` as `pack` respects the Random Access Container Concept. Similarly, since the size of `pack` is known at compile-time for any given type and architecture, `pack` can also be seen as a tuple and used as a compile-time sequence. Thus, `pack` is fully compatible with Boost.Fusion [31] and respects the Fusion Random Access Sequence Concept. Listing 5.8 presents the range and Fusion like interface.

Listing 5.8: `pack` range interface

```
typedef typename pack<float,8> p_t;
float t[] = {0.0,1.1,2.2,3.3,4.4,5.5,6.6,7.7};
p_t data(&t[0]); // data = [0.0,1.1,2.2,3.3,4.4,5.5,6.6,7.7]
//Random Access Sequence
for(std::size_t i = 0; i<p_t::static_size; i++) data[i] += i;
// Boost Fusion Random Access Sequence
typename boost::fusion::result_of::value_at_c<p_t,0>::type sum;
sum = fusion::accumulate(data, 0.f, add()); // sum = 58.8
```

Another ability of `pack` is to act as an Array of Structures/Structure of Arrays adaptor. For any given type `T` adapted as a compile-time sequence, accessing the $i^{th}$ element of a `pack` will give access to a complete instance of T (acting as an Array of Structures) while iterating over the `pack` content as a compile-time sequence will yield a tuple of `pack` thus making `pack` acts as a Structure of Arrays.

Listing 5.9: `pack` SOA to AOS

```
using boost::fusion::vector;
using boost::simd::load;
using boost::simd::pack;
using boost::simd::uint8_t;

typedef vector<uint8_t, uint8_t, uint8_t> pixel;
typedef vector<pack<float>, pack<float>, pack<float> > simd_pixel_SOA;
typedef pack< vector<float,float,float> > simd_pixel_AOS;

pixel data[simd_pixel_AOS::static_size]; // [...]

simd_pixel_SOA soa = load<simd_pixel_SOA>(&data[0]);
simd_pixel_AOS aos = load<simd_pixel_AOS>(&data[0]);
```

Line 12, `soa` is loaded as a Structure Of Array. Each `pack` of the `vector` contains a unique color of pixel as illustrated in figure 5.1.

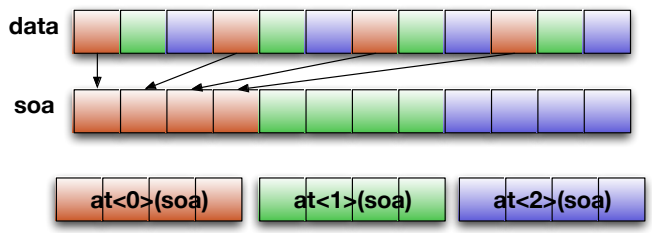



Figure 5.1: Load strategy for SOA

Line 13, `aos` is loaded with as a Array Of Structure. Each `vector` of the `pack` contains a pixel. While accessing `aos` as a compile-time sequence, the $i^{th}$ element of the sequence will yield a `pack` containing a unique color of pixel. Figure 5.2 illustrates this example.

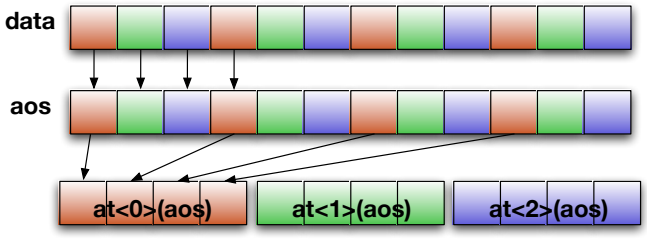



Figure 5.2: Load strategy for AOS

### 5.2.4 Supported functions

Listing 5.10 shows calls of SIMD functions on `pack` and a full list of available functions is reported in appendix C.2.4.4. The `pack` class is completed by a hundred high-level functions :

- **C++ operators:** including support for fused operations whenever possible,
- **Constant generators:** dealing with efficient constant SIMD value generation,
- **Arithmetic functions:** including `abs`, `sqrt`, `average` and various others,
- **IEEE 754 functions:** enabling bit-level manipulation of floating point values, including exponent and mantissa extraction,
- **Reduction functions:** for intra-register operations like sum or product of a register elements.

Listing 5.10: Function calls on `pack`

```
typedef typename pack<float,4> p_t;
float t[] = {0.0,1.1,2.2,3.3};
p_t data0(&t[0]); // data = [0.0,1.1,2.2,3.3]
p_t data1(-0.3,-0.2,-0.1,-0.0);

p_t r = simd::min(simd::abs(data1), data0);

std::cout << "r = " << r << std::endl;
  //Output: r = [0.0,0.2,0.1,0.0]
```

### 5.2.5 Shuffling operations

A typical SIMD use case is when the user wants to rearrange the data stored in `pack`. This operation is called *shuffling* the register. According to the cardinal of a `pack`, several permutations can be achieved between the data. To handle this, we introduce the `shuffle` function. This function accepts a metafunction class that will take as a parameter the destination index in the result register and return the correct index corresponding to the value from the source register. Listing 5.12 shows such a call.

Listing 5.11: `shuffle` example

```
// A metafunction that reverses the register
struct reverse_
{
  template<class Index, class Cardinal>
  struct apply
       : std::integral_constant<int, Cardinal::value - Index::value - 1> {};
};
[...]
pack<int,4> r{11,22,3,4};
r1 = boost::simd::shuffle<reverse_>(r); // r1 = {4,3,22,24}
```

A second version of the function is also available and allows the user to directly specify the indexes as template parameters. It is presented bellow:

```
r2 = boost::simd::shuffle<3,2,1,0>(r); // r2 = {4,3,22,24}
```

When called with a metafunction, `shuffle` has the ability of selecting the best permutation strategy available on the target. `shuffle` is implemented to recognize specific patterns that can be mapped to specific intrinsic calls. A generic matcher is able to match a specific permutation that leads to an optimized version of shuffling operation. For example, the permutation illustrates in figure 5.3 can be performed by the dedicated intrinsic `_mm_movehl_ps`. The compile-time generic matcher detects such a permutation pattern and `shuffle` dispatches automatically the call to this specific intrinsic.

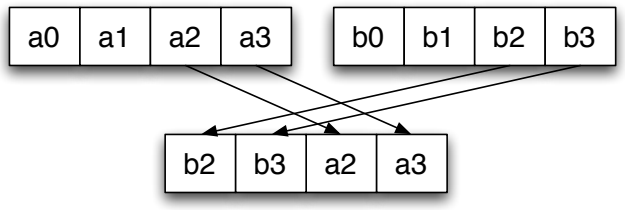



Figure 5.3: Permutation example

If no specific intrinsics can be called on the targeted architecture, the next choice is to use a general SIMD permutation unit. Such units can perform every permutation. SSSE3 has a special permute unit that permits to arbitrarily permute the values of a register. When SSSE3 is available on the architecture, this unit is used by `shuffle` for performing non optimized permutations through the `_mm_shuffle_epi8` intrinsic. ARM and Altivec also present such permute units.

The `shuffle` function uses its generic matcher to detect which call is the best. When the matcher fails to select a specific implementation of `shuffle`, a common version will be called and the permutation will be emulated.

A good example of shuffling operations is the transpose of a $4 \times 4$ matrix stored in 4 SIMD registers. Figure 5.4 illustrates the use of SIMD register to perform such

a transpose. The optimal sequence of intrinsic calls for performing the transpose is shown in listing 5.13.

Listing 5.13: Optimal transpose with SSE

```
__m128 row0 = {10,11,12,13}, row1 = {20,21,22,23},
       row2 = {30,31,32,33}, row3 = {40,41,42,43};
__m128 __r0 = row0, __r1 = row1, __r2 = row2, __r3 = row3;
__m128 __t0 = _mm_unpacklo_ps (__r0, __r1);
__m128 __t1 = _mm_unpacklo_ps (__r2, __r3);
__m128 __t2 = _mm_unpacklo_ps (__r0, __r1);
__m128 __t3 = _mm_unpacklo_ps (__r2, __r3);
row0 = _mm_movelh_ps (__t0, __t1);
row1 = _mm_movehl_ps (__t1, __t0);
row2 = _mm_movelh_ps (__t2, __t3);
row3 = _mm_movehl_ps (__t3, __t2);
```

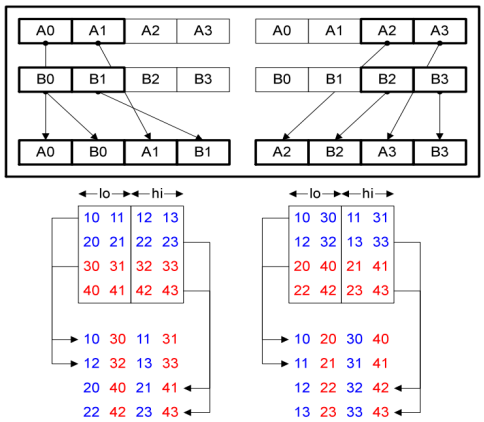



Figure 5.4: $4 \times 4$ matrix transpose in SIMD

The Boost.SIMD version in listing 5.14 is the equivalent of the intrinsic version shown in listing 5.13. The ability of `shuffle` to match the best intrinsic call through its generic matcher is used here and the exact same intrinsics calls are generated.

Listing 5.14: Boost.SIMD transpose with `shuffle`

```
pack<float,4> row0 = {10,11,12,13}, row1 = {20,21,22,23},
              row2 = {30,31,32,33}, row3 = {40,41,42,43};
pack<float,4> __r0 = row0, __r1 = row1, __r2 = row2, __r3 = row3;
pack<float,4> __t0 = shuffle<0,0,1,1>(__r0, __r1);
pack<float,4> __t1 = shuffle<0,0,1,1>(__r2, __r3);
pack<float,4> __t2 = shuffle<0,0,1,1>(__r0, __r1);
pack<float,4> __t3 = shuffle<0,0,1,1>(__r2, __r3);
row0 = shuffle<0,1,0,1>(__t0, __t1);
row1 = shuffle<2,3,2,3>(__t1, __t0);
row2 = shuffle<0,1,0,1>(__t2, __t3);
row3 = shuffle<2,3,2,3>(__t3, __t2);
```

In addition, this version is fully portable and works on every SIMD extensions.

## 5.3 C++ Standard integration

Writing small functions acting over a few `packs` has been covered in the previous section and we saw how the API of Boost.SIMD makes such functions easy to write by abstracting away the architecture-specific code fragments. Realistic applications usually require such functions to be applied over a large set of data. To support such a use case in a simple way, Boost.SIMD provides a set of classes to integrate SIMD computation inside C++ relying on the Standard Template Library (STL) components, thus totally reusing its generic aspect.

Based on Generic Programming as defined by [94], the STL is based on the separation between data, stored in various `Containers`, and the way one can iterate these data sets with `Iterators` and algorithms. Instead of providing SIMD aware containers, Boost.SIMD reuses existing STL Concepts to adapt STL-based code to SIMD computations. The goal of this integration is to find standard ways to express classical SIMD programming idioms, thus raising expressiveness and still benefiting from the expertise put into these idioms. More specifically, Boost.SIMD provides SIMD-aware allocators, iterators for regular SIMD computations – including interleaved data or sliding window iterators – and hardware-optimized algorithms.

### 5.3.1 Aligned allocator

The hardware implementation of SIMD processing units introduces constraints related to memory handling. Performance is guaranteed by accessing to the memory through dedicated `aligned_load` and `aligned_store` intrinsics that perform register-length aligned memory accesses. This constraint requires a special memory allocation strategy via OS and compiler-specific function calls.

Boost.SIMD provides two STL compliant allocators dealing with this kind of alignment. The first one called `simd::allocator` wraps these OS and compiler functions in a simple STL-compliant allocator. When an existing allocator defines a specific memory allocation strategy, the user can adapt it to handle alignment by wrapping it in `simd::allocator_adaptor`.

Listing 5.15: Aligned allocator

```
1 //Align the memory of a vector
2 std::vector<T, boost::simd::allocator<T> > p(5);
3
4 //Adapt an allocator
5 typedef std::allocator<float> base;
6 typedef boost::simd::allocator_adaptor<base> alloc;
7 std::vector<T, alloc > p(5);
```

### 5.3.2 SIMD Iterator

Modern C++ programming styles based on Generic Programming usually lead to an intensive use of various STL components like Iterators. Boost.SIMD provides iterator adaptors that turn regular random access iterators into iterators suitable for SIMD processing. It means that SIMD iterators must work on aligned memory to be efficient. These adapters act as free functions taking regular iterators as parameters and return iterators that output `pack` whenever dereferenced. These iterators are then usable directly in usual STL algorithms such as `transform` or `fold` (Listing 5.16).

Listing 5.16: SIMD Iterator with STL algorithm

```
1 vector<int,simd::allocator<int> > v(128), r(128);
2
3 transform ( simd::begin(v.begin())
4           , simd::end(v.end())
5           , simd::begin(r.begin())
6           , [](pack<int>& p){ return -p; }
7           );
```

If the memory of the container is not well prepared (i.e. aligned memory), the regular iterators are shifted to a correct aligned address. Some data are then omitted and need to be computed before and after the SIMD range of aligned iterators. Figure 5.5 illustrates such a scenario.

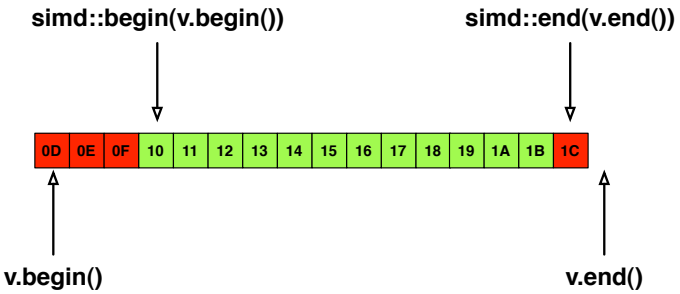



Figure 5.5: Address alignment for SIMD iterators

Some application domains like image or signal processing require specific memory access patterns in which the **neighborhood** of a given value is used in the computation. Digital filtering and convolutions are examples of such algorithms. The efficient techniques for vectorizing such operations consists on performing shifted loads *i.e.* loads from unaligned memory addresses, so that each neighbor can be available in a SIMD vector. To limit the number of such loads, a technique called register rotation technique [86] is often used. This technique allows filter-like algorithms to perform only one load per iteration, swapping neighbor values as the algorithm goes forward. This idiomatic way of implementing such algorithms usually increases performance by a significant factor and is a good candidate for encapsulation.

In Boost.SIMD, `shifted_iterator` is an iterator adaptor encapsulating such an abstraction. This iterator is constructed from an iterator and a compile-time width `N`. When dereferenced, it returns a static array of `N packs` containing the initial data and its shifted neighbors (Fig. 5.6). When incremented, the tuple value are internally swapped and the new vector of value is loaded, thus implementing register rotation. With such an iterator, one can simply write an average filter using `std::transform`.

Listing 5.17: Average filtering

```
struct average
{
  template<class T> typename T::value_type
  operator()(T const& t) const
  {
    typename T::value_type d(1./3);
    return (t[0]+t[1]+t[2])*d;
  }
};

vector<float> in, out;

transform( shifted_iterator<3>(in.begin())
         , shifted_iterator<3>(in.end())
         , begin(out.begin())
         , average()
         );
```

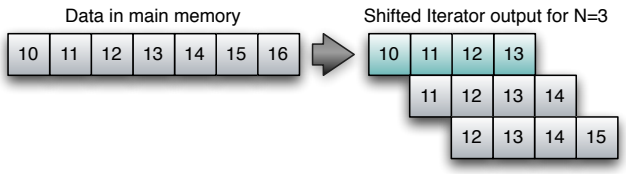



Figure 5.6: Shifted iterator

Code written this way keeps a conventional structure and facilitate the usage of template functors for both scalar and SIMD while also helps maximizing code reuse.

Listing 5.18: `pack` range interface

```
std::vector<int> data(pack<int>::static_size*3);
std::vector< pack<int> > dest(3);
[...]
std::cout << std::distance ( boost::begin(simd::input_range(data))
                           , boost::end(simd::input_range(data))
                           ) << std:endl;
// Output is 3 here.

boost::copy(simd::input_range(data), dest.begin()); // Copy data into dest
[...]
boost::fill(simd::output_range(data), simd::Zero<int>()); // Fill data with
    Zeros
```

### 5.3.3 SIMD Algorithms

Previous examples of integration within standard algorithms are still limited. Applying `transform` or `accumulate` algorithms on SIMD aware data requires the size of the data to be an exact multiple of the SIMD register width. Also, in the case of fold, it can potentially require to perform additional operations at the end of its call. To alleviate this limitation, Boost.SIMD provides its own overload for both `transform` and `fold` that takes care of potential trailing data and performs proper completion of `fold`. Listing 5.19 illustrate a call to the Boost.SIMD's version of `transform`.

Listing 5.19: SIMD `transform` algorithm

```
struct plus
{
  template<class T>
  T operator()(T const& t0, T const& t1) const
  {
    return t0 + t1;
  }
};

std::vector<float> data_in1(113);
std::vector<float> data_in2(113);
[...]
boost::simd::
transform( data_in1.begin(), data_in1.end()
         , data_in2.begin()
         , data_out1.begin()
         , [](pack<float> t0, pack<float> t1)->pack<float>{ return t0+t1; }
         );
```

## 5.4 Case Analysis: Generic SIMD code generation

In this section we detail how to write a generic SIMD code with Boost.SIMD. The objective is to move from a simple scalar version of the `dot` function to a full generic version of it. Thus, we will take a look at the generated assembly code.

### 5.4.1 Scalar version of the `dot` function

A generic scalar version of the `dot` function can be simply defined as shown in Listing 5.20

Listing 5.20: Scalar `dot` function

```
template<typename Value>
Value dot(Value* first1, Value* last1, Value* first2)
{
  Value v(0);

  while(first1 != last1) v += *first1++ *  *first2++;

  return v;
}
```

This template function iterates over data pointed by `first1` and `first2`, computes the product of said data and sums them.

### 5.4.2 Transition from scalar to SIMD code

If the algorithm is clearly vectorizable, it has to be modified in such a way that its SIMD nature becomes apparent.

First, we arbitrarily unroll the code by an arbitrary factor to make data parallelism obvious in Listing 5.21

Listing 5.21: Unrolling by 2 the `dot` function

```
template<typename Value>
Value dot(Value* first1, Value* last1, Value* first2)
{
  Value v,v1(0),v2(0);

  // Let's consider that (last1-first1) is divisible by 2
  while(first1 != last1)
  {
    v1 += *first1 *  *first2;
    first1++;
    first2++;

    v2 += *first1 *  *first2;
    first1++;
    first2++;
  }

  v = v1 + v2;

  return v;
}
```

The algorithm is split in two parts:

- We first run over every elements inside both datasets and multiply them.
- We then sum the intermediate values into the final result.

By unrolling this pattern to arbitrary size, we expose the fact that the multiplication between the two dataset is purely "vertical" and so, is vectorizable. The sum of the partial result itself is a "horizontal" operation, i.e a vectorizable computation operating across the element of a single vector.

### 5.4.3 Building a SIMD loop nest

We are now going to use `pack` to actually vectorize this algorithm. The main idea is to compute a partial sum inside an instance of `pack` and perform a final summation at the end. For this purpose, we use the `load` function to load data from `first1` and `first2`, process those `pack` instances using the proper operators and advance the pointers by the size of the `pack`. Let's consider that `(last1-first1)` is divisible by the size of the pack.Listing 5.22 shows the SIMD version of `dot`.

Listing 5.22: SIMDization the `dot` function

```
#include <boost/simd/sdk/simd/pack.hpp>
#include <boost/simd/include/functions/sum.hpp>
#include <boost/simd/include/functions/load.hpp>
#include <boost/simd/include/functions/plus.hpp>
#include <boost/simd/include/functions/multiplies.hpp>

template<typename Value>
Value dot(Value* first1, Value* last1, Value* first2)
{
  using boost::simd::sum;
  using boost::simd::pack;
  using boost::simd::load;

  typedef pack<Value> type;
  type tmp=0;

  while(first1 != last1)
  {
    // Load current values from the datasets
    pack<Value> x1 = load< type >(first1);
    pack<Value> x2 = load< type >(first2);

    // Computation
    tmp = tmp + x1 * x2;

    // Advance to the next SIMD vector
    first1 += type::static_size;
    first2 += type::static_size;
  }

  return sum(tmp);
}
```

The computation code looks a lot like the scalar version. We simply jump over data using the `pack` size.

### 5.4.4 Preparing the data

Now that our SIMD dot product function is ready, we can apply it on some data. As currently written, one can simply call dot on any piece of memory of the proper size.

Listing 5.23: Simple `main`

```
#include <vector>

int main()
{
  std::vector<float> a(1024), b(1024);

  // ... fill up a and b

  float r = dot(&a[0], &a[0]+1024, &b[0]);
}
```

Even if this version works, the issue is that we don't use aligned load to fill SIMD register from the memory. On some systems, typically pre-Nehalem for x86

or PowerPC, unaligned loads and stores cost far more than aligned ones. Therefore it is important to use aligned data in memory. To do so, we need to modify the code in two places.

- First the `dot` function should use the `aligned_load` function that behaves exactly as `load` but uses aligned memory accesses as an input. The code then becomes:

```
pack<Value> x1 = aligned\_load< type >(first1); \\
pack<Value> x2 = aligned\_load< type >(first2); }
```

  An alternative is to use the constructor from aligned pointer of `pack`, giving us the following code:

```
pack<Value> x1(first1); \\
pack<Value> x2(first2); }
```

- Then, we need to provide to `dot` a pointer to aligned memory. This can be done by using the Boost.SIMD `allocator` class as the `std::vector` allocator 5.26.

Listing 5.26: Aligned memory for data

```
#include <vector>
#include <boost/simd/sdk/simd/pack.hpp>
#include <boost/simd/include/functions/sum.hpp>
#include <boost/simd/include/functions/load.hpp>
#include <boost/simd/include/functions/plus.hpp>
#include <boost/simd/include/functions/multiplies.hpp>

template<typename Value>
Value dot(Value* first1, Value* last1, Value* first2) {
  using boost::simd::sum;
  using boost::simd::pack;
  using boost::simd::load;
  typedef pack<Value> type;
  type tmp;

  // Let's consider that (last1-first1) is divisible by the size of the pack.
  while(first1 != last1) {
    pack<Value> x1 = aligned_load< type >(first1);
    pack<Value> x2 = aligned_load< type >(first2);
    tmp = tmp + x1 * x2;
    first1 += type::static_size;
    first2 += type::static_size;
  }
  return sum(tmp);
}

int main()
{
  std::vector<float, boost::simd::allocator<float> > a(1024), b(1024);
  // ... fill up a and b
  float r = dot(&a[0], &a[0]+1024, &b[0]);
}
```

### 5.4.5 Resulting code generation

Listing 5.27 shows the resulting code generation for a SSE2 system. The assembly code is still showing no abstraction penalty. The entire computation is performed in SIMD here, `*ps` instructions are generated in place of the scalar ones.

Listing 5.27: Assembly code generated with Boost.SIMD on SSE2

```
        cmp     %rdi,%rsi
        xorps   %xmm1,%xmm1
        je      end
begin:  movaps  (%rdi),%xmm0
        add     $0x10,%rdi
        mulps   (%rdx),%xmm0
        add     $0x10,%rdx
        cmp     %rdi,%rsi
        addps   %xmm0,%xmm1
        jne     begin
end:    movaps  %xmm1,%xmm2
        shufps  $0x4e,%xmm1,%xmm2
        addps   %xmm1,%xmm2
        movaps  %xmm2,%xmm0
        shufps  $0x91,%xmm2,%xmm0
        addps   %xmm2,%xmm0
        retq
```

Now if we look to the AVX version in listing 5.28, the assembly code is correctly generated and presents calls to `*pd` instruction as this version was compiled for double precision floating point values. The genericity of this code makes this change really simple and the generated code stay correct.

Listing 5.28: Assembly code generated with Boost.SIMD on AVX

```
        vxorpd          %xmm0,%xmm0,%xmm0
        nopw            0x0(%rax,%rax,1)
begin:  vmovapd         (%rdi,%rax,8),%ymm1
        vmulpd          (%rsi,%rax,8),%ymm1,%ymm1
        add             $0x4,%rax
        cmp             %rcx,%rax
        vaddpd          %ymm0,%ymm1,%ymm0
        jb              begin
        vhaddpd         %ymm0,%ymm0,%ymm0
        vextractf128    $0x1,%ymm0,%xmm1
        vaddpd          %xmm1,%xmm0,%xmm0
```

The Altivec assembly in listing 5.30 also confirm that Boost.SIMD generates the correct code. The FMA instruction is successfully generated.

Listing 5.29: Assembly code generated with Boost.SIMD on Altivec

```
begin:  rlwinm  r10,r9,2,0,29
        addi    r9,r9,4
        lvx     v13,r6,r10
        lvx     v1,r7,r10
        cmplw   cr7,r9,r8
        vmaddfp v0,v13,v1,v0
        blt     begin
        lvx     v13,0,r19
        lvx     v1,0,r20
```

```
10          addi    r9,r1,1088
11          vperm   v13,v0,v0,v13
12          vaddfp  v0,v0,v13
13          vperm   v1,v0,v0,v1
14          vaddfp  v0,v1,v0
15          stvx    v0,r9,r18
```

This first hand-written version of `dot` still has some shortcomings as it requires the size of the data to be a multiple of the `pack` cardinal. It also does not perform loop unrolling but this can be handle with ease.

### 5.4.6 Choosing SIMD extensions at runtime

Using Boost.SIMD requires compiling for a particular target machine which has particular SIMD instructions available. For many architectures (x86 in particular), SIMD instructions may be conditionally supported depending on the exact hardware being used, with more recent hardware typically supporting more SIMD instructions than older ones.

We will demonstrate how to switch between SSE and AVX for the same code depending on the capabilities of the x86 hardware that the program is running on.

Choosing between what is supported by the hardware can be done using the `boost::simd::is_supported` function.

#### The Boost.SIMD model, Translation Units and Shared Objects

Boost.SIMD assumes that you are building for a specific architecture for the whole duration of a translation unit (the compilation of a single `.cpp` file). It is not possible to switch between targeting an architecture with AVX and without AVX in the same translation unit. The only option supported is to recompile with different compilation flags. This model is the safest one and allows to make the best of all compilers. It also implies to work with non template functions.

Linking objects compiled with different settings can also lead to subtle issues, such as breaking the One Definition Rule when collapsing inline functions. For this reason it is recommended to isolate the translation units in DLLs or shared object with hidden visibility.

#### The code

Here, we keep the code of the `dot` function presented in listing 5.31. We ensure that the code is extension-agnostic so that we can compile the same code for different targets and we move it to a simple `dot.cpp` file.

Listing 5.31: dot.cpp file

```
#include <vector>
#include <boost/simd/sdk/simd/pack.hpp>
#include <boost/simd/include/functions/sum.hpp>
#include <boost/simd/include/functions/load.hpp>
#include <boost/simd/include/functions/plus.hpp>
#include <boost/simd/include/functions/multiplies.hpp>
#include <boost/config.hpp>

BOOST_SYMBOL_EXPORT
float dot(float* first1, float* last1, float* first2, BOOST_SIMD_DEFAULT_SITE
    )
{
  using boost::simd::sum;
  using boost::simd::pack;
  using boost::simd::load;

  typedef pack<float> type;
  type tmp;

  while(first1 != last1)
  {
    pack<float> x1 = aligned_load< type >(first1);
    pack<float> x2 = aligned_load< type >(first2);

    tmp = tmp + x1 * x2;

    first1 += type::static_size;
    first2 += type::static_size;
  }
  return sum(tmp);
}
```

We use the preprocessor symbol `BOOST_SIMD_DEFAULT_SITE`, which expands to the current SIMD extension being target, to decorate the symbol. We can now compile different variants of `dot.cpp`.

With GCC on Linux:

```
g++ -O3 -DNDEBUG -shared -fvisibility=hidden -msse2 -I$BOOST_ROOT
dot.cpp -o libmy_dot_sse2.so
g++ -O3 -DNDEBUG -shared -fvisibility=hidden -mavx -I$BOOST_ROOT
dot.cpp -o libmy_dot_avx.so
```

With MSVC on Windows:

```
cl /Oxt /EHsc /MD /DNDEBUG /DWIN32 /D_WINDOWS /fp:precise /LD
/arch:SSE2 /I$BOOST_ROOT dot.cpp /Femy_dot_sse2.dll
cl /Oxt /EHsc /MD /DNDEBUG /DWIN32 /D_WINDOWS /fp:precise /LD
/arch:AVX /I$BOOST_ROOT dot.cpp /Femy_dot_avx.dll
```

### The Dispatcher

We need now to create a dispatcher that will call the specific version of the `dot` function according to the SIMD extension detected at runtime. Listing 5.32 shows the correct way to write this dispatcher.

Listing 5.32: `my_dot_dispatcher.cpp` file

```
#include <boost/simd/sdk/simd/extensions/meta/tags.hpp>
#include <boost/simd/sdk/config/is_supported.hpp>
#include <boost/config.hpp>

BOOST_SYMBOL_IMPORT
float dot(float* first1, float* last1, float* first2, BOOST_SIMD_DEFAULT_SITE
    );

float my_dot(float* first1, float* last1, float* first2)
{
  if(boost::simd::is_supported<boost::simd::tag::avx_>())
    return dot(first1, last1, first2, boost::simd::tag::avx_());
  else
    return dot(first1, last1, first2, boost::simd::tag::sse2_());
}

int main()
{
  // ...
}
```

We then can compile the entire code to generate a portable binary. With GCC on Linux:

```
g++ -O3 -DNDEBUG -I$BOOST_ROOT my_dot_dispatcher.cpp -o my_dot
-lmy_dot_sse2 -lmy_dot_avx
```

With MSVC on Windows:

```
cl /Oxt /EHsc /MD /DNDEBUG /DWIN32 /D_WINDOWS /fp:precise
/I$BOOST_ROOT my_dot_dispatcher.cpp /Femy_add /link
my_add_sse2.lib my_add_avx.lib
```

## 5.5 Implementation

Boost.SIMD's implementation relies on two elements: the use of Boost.Proto [76] to capture and transform expressions at compile time and Boost.Dispatch (see chapter 4) that allows for fine to coarse grain function specialization handling both types and architectures.

### 5.5.1 Function Dispatching

To be able to extend Boost.SIMD we need a way to add an arbitrary function overload on any function depending on the argument types, the related SIMD extension and the properties of the function itself. Boost.Dispatch is used here to build a hierarchy of tags which is computed as follow:

- For each SIMD family, a hierarchy of classes is defined to represent the relationship between each extension variant. For example a SSE3 tag inherits from the SSE2 tag as SSE3 is more refined than SSE2. Listing 5.33 shows the corresponding hierarchy for the x86 family.

Listing 5.33: Hierarchy of classes for x86 SIMD family

```
namespace boost { namespace simd { namespace tag
{
  // Tag hierarchy for SSE extensions
  BOOST_DISPATCH_HIERARCHY_CLASS(sse_, simd_);
  BOOST_DISPATCH_HIERARCHY_CLASS(sse2_, sse_);
  BOOST_DISPATCH_HIERARCHY_CLASS(sse3_, sse2_);
  BOOST_DISPATCH_HIERARCHY_CLASS(sse4a_, sse3_);
#ifdef BOOST_SIMD_ARCH_AMD
  BOOST_DISPATCH_HIERARCHY_CLASS(ssse3_, sse4a_);
#else
  BOOST_DISPATCH_HIERARCHY_CLASS(ssse3_, sse3_);
#endif
  BOOST_DISPATCH_HIERARCHY_CLASS(sse4_1_, ssse3_);
  BOOST_DISPATCH_HIERARCHY_CLASS(sse4_2_, sse4_1_);
  BOOST_DISPATCH_HIERARCHY_CLASS(avx_, sse4_2_);
  BOOST_DISPATCH_HIERARCHY_CLASS(fma4_, avx_);
  BOOST_DISPATCH_HIERARCHY_CLASS(xop_, fma4_);
#ifdef BOOST_SIMD_ARCH_AMD
  BOOST_DISPATCH_HIERARCHY_CLASS(fma3_, xop_);
#else
  BOOST_DISPATCH_HIERARCHY_CLASS(fma3_, avx_);
#endif
  BOOST_DISPATCH_HIERARCHY_CLASS(avx2_, fma3_);
} } }
```

- For each argument type, a Boost.Dispatch **hierarchy** is automatically computed. This hierarchy contains information about: the type of register used to store the value (SIMD or scalar), the intrinsic properties of the type (floating point, integer, size in bytes) and the actual type itself. These hierarchies are also ordered from the most fine grained description (for example, `scalar_< int8_<char> >`) to the largest one (for example, `scalar_< arithmetic_<char> >`).

Each function overload is then discriminated by the type list built from the hierarchy of the current architecture and the hierarchies of every argument of the function. This unique set of hierarchies is then used to select a function object to perform the function call. A specific intrinsic call then occurs when the hierarchies select an architecture specific function implementation. We introduced such a mechanism in section 4.2.4 of chapter 4.

In section 4.2.4.3 of chapter 4 we also introduced the `generic_` hierarchy entry that enables architecture independent code reuse. Boost.SIMD uses the Boost.Dispatch `generic_` hierarchy to implement functions that do not rely on a specific architecture implementation. Boost.SIMD functions are then reused inside the library to build higher order functions. The generic version of a function then relies on the architecture aware dispatch of the used functions.

### 5.5.2 AST Manipulation with Boost.Proto

A fundamental aspect of SIMD programming relies on the effective use of fused operations like multiply-add on VMX extensions or sum of absolute differences on SSE extensions. Unlike simple wrappers around SIMD operations [66], `pack` relies on *Expression Templates* [28] to capture the Abstract Syntax Tree (AST) of large `pack`-based expressions and performs compile-time optimizations on this AST. These optimizations include the detection of fused operation and replacement or reordering of reductions versus elementwise operations. This compile-time optimization pass ensures that every architecture-specific optimization opportunity is captured and replaced by the superior version of the code. Moreover, the AST-based evaluation process is able to merge multiple function calls into a single inlined one, contrary to solutions like MKL where each function can only be applied on the whole data range at a time. This increases data locality and ensure high performance for any combination of functions.

As stated earlier, SIMD instruction sets usually provide DSP-like fused operations that are able to implement complex computation in a single cycle. Operations like fused multiply- add and sum of absolute differences are available on an increasing sub-range of SIMD extensions sets. The main issue is that writing portable and efficient code that will use these fused operations whenever available is difficult. It implies handling a large number of variation points in the code and people unaware of their existence will obtain poor performance. To limit the amount of per-architecture expertise required by the developer of SIMD applications, Boost.SIMD is designed as an Embedded Domain Specific Language [92]. Expressions Templates [28] have been a tool of choice for such designs but writing complex EDSLs by hand leads to a hard to maintain code base. As introduced in chapter 3, thanks to Boost.Proto , and contrary to other EDSL- based solutions[57], Boost.SIMD does not directly evaluate its compile-time AST after its capture. Instead, it relies on a multi-pass system: a first one optimizes the AST and a second one takes care of the proper code generation.

#### 5.5.2.1 AST building

Boost.Dispatch has the ability to statically dispatch inline function calls according to a hierarchy of types (see chapter 4). The library can also dispatch functions that manipulates Boost.Proto ASTs as introduced in section 4.2.4.4. Boost.SIMD takes advantage of this by detecting an expression matching a candidate for optimization and then dispatching the evaluation of its expression to the right function calls.

We take the following `pack` expression as an example:

```
pack<T> d = a + b*c;
```

This expression can be optimized by a Fused Multiply and Add (FMA) operation when it is available on the targeted SIMD extension. We describes how Boost.SIMD handles such optimizations.

First, Boost.Proto builds an AST for this expression as `pack` is a Boost.Proto terminal. The use of `pack` in this statement results in a contaminated construction of a Boost.Proto based expression. Figure 5.7 illustrates the AST of our example.

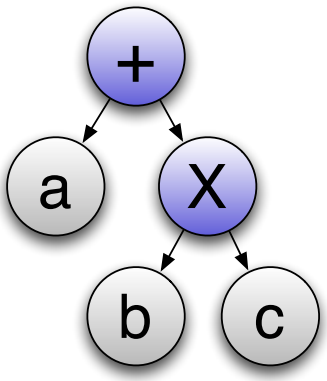



Figure 5.7: Boost.Proto AST of `a + b*c`

When Boost.Proto builds such an expression, every C++ operators can have meanings and behaviors independent of any context. Thus, we can control the behavior of these operators by overloading them. Boost.SIMD then overloads each Boost.Proto operators to call a Boost.Dispatch based function. This function is hierarchized with the tag corresponding to the operator. The RHS of our expression is the following:

```
plus( pack<T> , multiplies(pack<T>, pack<T>) )
```

As `pack` is a Boost.Proto terminal, the call to the Boost.Dispatch function is also hierarchized according to the AST hierarchy. The concrete arguments of

the `plus` function are Boost.Proto based expressions. Its first argument is a terminal (*i.e.* a `pack`) and the second is the multiplies node of the AST. Such a call is illustrated in figure 5.8.

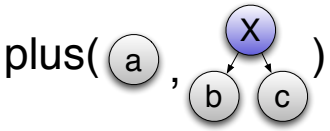



Figure 5.8: `plus` function with Boost.Proto based arguments

The function call then matches the following Boost.Dispatch AST hierachy:

```
plus( expr_< simd_<T,X>, tag::terminal_  , 0 >
    , node_< simd_<T,X>, tag::multiplies_, 2 >
    )
```

We can see that every argument is tagged with its semantic information then making the function call aware of the next node properties. Our `plus` function is aware of the upcoming node. The only thing we need to do is to provide a `plus` implementation matching this specific AST hierarchy to dispatch this call to a FMA function. The corresponding `plus` function of Boost.SIMD matching such an expression pattern is presented in listing 5.34.

Listing 5.34: The `plus` function matching a FMA

```
1  BOOST_SIMD_FUNCTOR_IMPLEMENTATION( boost::simd::tag::plus_
2                                   , tag::formal_
3                                   , (D)(A0)(A1)
4                                   , (unspecified_<A0>)
5                                     ((node_< A1, boost::simd::tag::multiplies_
6                                             , mpl::long_<2>
7                                             , D>))
8                                   )
9  {
10   BOOST_DISPATCH_RETURNS(2, (A0 const& a0, A1 const& a1),
11     simd::fma(boost::proto::child_c<0>(a1), boost::proto::child_c<1>(a1), a0)
12   )
13 };
```

In listing 5.34 we see that all of these optimizations are performed at the top-level of the architecture hierarchy through the tag `formal_`. The AST optimization is then independent of the architecture at this point. The first argument of `plus` matches the `unspecified_` hierarchy which means that every sub-AST or node can be passed to the function. The returned `fma` function is still taking AST nodes as arguments and returns an FMA node with an arity of 3. The optimization scheme is propagated until the end of the AST as we still work at the AST level. Here, we introduced an example relying on `pack` operators. The same optimization scheme is performed with function calls.

The library has now the ability to know what will come in the remaining part of an AST. while dispatching function calls through the AST hierarchy, we can inject optimizations by providing function overloads for specific expression patterns on specific architectures. While we walk down the AST, we are able to match and select such optimizations. The **look-ahead** optimization of Boost.SIMD allows to directly optimized the AST during its construction. Another approach could have been to construct the AST and then optimize it. This requires to modify the AST after its construction which introduces a significant overhead.

#### 5.5.2.2 AST Evaluation

Now that the AST is constructed, we need to evaluate it.

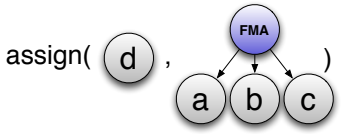



Figure 5.9: `assign` function call on the LHS and RHS

The entry point of the evaluation is the '=' operator of `pack`. The RHS of the '=' operator is now a fully constructed and optimized AST and the LHS is a terminal. At this point, we reconstruct the entire AST with the '=' node by calling the `assign` function with the LHS and the RHS as arguments (see figure 5.9). We can then pass the complete AST to the `evaluate` function.

```
evaluate( assign( expr_< simd_<T,X>, tag::terminal_, 0 >
                , node_< simd_<T,X>, tag::fma_     , 3 >
                )
        )
```

The only purpose of the `evaluate` function here is to call the `run` function on the entire AST. `run` walks down the AST and replaces all the top-level function calls with their Boost.SIMD architecture aware versions. The code generation occurs at this point. The call to Boost.SIMDfunctions then results to the specific intrinsic calls on the specific register held by each terminal of the AST. This results in the generation of a full SIMD version of the expression.

We demonstrated how Boost.SIMD is able to detect optimization patterns at the expression level and generate the corresponding SIMD code. The efficiency of such an implementation relies on some issues that we address in the next section.

## 5.6 Implementation Discussion

In this section we discuss implementation issues that impacts the development of Boost.SIMD and its efficiency.

### 5.6.1 Function inlining and ABI issue

The main issue when implementing this library efficiently is tied to how the compiler will handle the `pack` type. In particular how the Application Binary Interface (ABI) defines that objects of these types are passed to functions. Indeed, since `pack` is defined as a `struct`, many ABIs (with the notable exception of Intel x86-64 on Linux) will be unable to pass that structure directly in registers. Certain ABIs will also reject passing these types by value due to the alignment requirement being often higher than that of the stack.

As a result "stack dance" – the unnecessary writing and reading of SIMD register contents to stack memory – might occur whenever a non-inlined function is called. A possible way to solve this problem is to force a wrapper function to be inlined and make its call use the native type of the platform to be more friendly with the ABI. Boost.SIMD forces every functions to be inlined for the previous reasons.

### 5.6.2 Unrolling

When working at the instruction level, specific low level optimizations tied to the hardware are relevant. Loop unrolling is a technique that allows to overcome limitations related to the execution of an instruction. These shortcomings are : instruction latencies (memory operations), branching penalties or pipeline effects. A significant gain can be obtain with such a technique but this implies to manually unroll the loop by replicating the statements of the loop-nest.

This is done at the expense of the binary size. In addition, this optimization is empirical due to its correlation with the algorithm and a given architecture. In consequence, we decided to not include an abstract and automatic mechanism for loop unrolling inside Boost.SIMD. One solution could have been to write a meta-unroller able to unroll a function (unary or binary) via a Duff's devices optimization. This approach is presented in appendix D.

## 5.7 Benchmarks

This section presents the Boost.SIMD performance on several benchmarks. Every benchmark has been tested using the SSE4.2, AVX and Altivec instruction sets to demonstrate the portability of the code. The benchmarks include: an implementation of the `AXPY` kernel and three image processing algorithms featuring the various types of SIMD iterator abstractions. Unless stated otherwise, the tests have been run using g++ 4.6. The SSE2, AVX and Altivec benchmarks have been executed on the Nehalem, Sandy Bridge and PowerPC G5 microarchitectures respectively. Appendix B summarizes the frequencies and extensions of the processors used for the following benchmarks. In appendix A are also described all the algorithms presented in the benchmarks. The benchmarks results are reported in GFlop/s and cycles per element (*cpe*) or cycles per point (*cpp*) depending on the algorithm.

### 5.7.1 Basic Kernels

#### The AXPY Kernel

The AXPY kernel is one the most basic and used BLAS routine. We want to assess two things. First, how does Boost.SIMD implementation performs against a naive hand written AXPY SIMD version. And finally, how Boost.SIMD performs against autovectorizers. Listing 5.35 shows the Boost.SIMD implementation of the AXPY kernel.

Listing 5.35: Boost.SIMD version of the AXPY kernel

```
1  using boost::simd::pack;
2  using boost::simd::aligned_store;
3
4  typedef pack<T> type;
5  std::size_t step_size_ = boost::simd::meta::cardinal_of<type>::value;
6  for (std::size_t i = 0; i<size_; i+=step_size_)
7  {
8    type X_pack(&X[i]);
9    type Y_pack(&Y[i]);
10   aligned_store( alpha * X_pack + Y_pack, &Y[i] );
11 }
```

Tables 5.3 shows how Boost.SIMD performs against handwritten SIMD code without loop unrolling. The results of the generated code are equivalent to the SSE4.2 code and assess that Boost.SIMD delivers the expected speedup.

Table 5.3: Boost.SIMD vs handwritten SIMD code for the AXPY kernel in *GFlop/s*

| Type | Size | Version | SSE4.2 |
|---|---|---|---|
| float | $2^9$ | Ref. SIMD | 4.03 |
| | | Boost.SIMD | 4.70 |
| | $2^{14}$ | Ref. SIMD | 3.40 |
| | | Boost.SIMD | 3.49 |
| | $2^{19}$ | Ref. SIMD | 3.41 |
| | | Boost.SIMD | 3.98 |

The MKL Library proposes an optimized routine of this algorithm for the x86 processor family. Autovectorizers in compilers are also able to capture this type of kernel and generate optimized code for the targeted architecture. Tables 5.4 and 5.5 shows how Boost.SIMD performs against the two of them.

Table 5.4: Boost.SIMD vs Autovectorizers for the DAXPY kernel in *GFlop/s*

| Type | Size | Version | SSE2 | AVX |
|---|---|---|---|---|
| double | 16 | gcc | 1.10 | 1.10 |
| | | mkl | 0.76 | 0.76 |
| | | B.SIMD | 1.28 | 4.00 |
| | 64 | gcc | 0.55 | 0.55 |
| | | mkl | 2.61 | 0.76 |
| | | B.SIMD | 2.17 | 4.00 |
| | 256 | gcc | 1.49 | 1.49 |
| | | mkl | 5.82 | 0.76 |
| | | B.SIMD | 1.71 | 2.93 |
| | 1024 | gcc | 1.35 | 1.35 |
| | | mkl | 7.91 | 0.76 |
| | | B.SIMD | 2.00 | 3.03 |
| | 4096 | gcc | 1.17 | 1.17 |
| | | mkl | 4.91 | 0.76 |
| | | B.SIMD | 1.91 | 2.76 |

The sizes of the used vectors are chosen according to the cache sizes of their respective targets so that they all fit in the L2 cache. The `gcc` version shows the autovectorizer work on the AXPY kernel written in C++ code. The `mkl` version shows the performance of the Intel MKL AXPY BLAS function.

First, the GNU compiler is unable to vectorize and unroll the loop properly due to its inability to go through the various layer of the C++ code. The MKL version featuring both SIMD and loop optimizations is clearly superior to all the other versions except for very small sizes. Measurements show that the performance of

Table 5.5: Boost.SIMD vs Autovectorizers for the SAXPY kernel in *GFlop/s*

| Type | Size | Version | SSE2 | AVX |
|---|---|---|---|---|
| float | 16 | gcc | 1.16 | 1.16 |
| | | mkl | 1.07 | 1.07 |
| | | B.SIMD | 3.20 | 4.00 |
| | 64 | gcc | 1.31 | 1.31 |
| | | mkl | 3.66 | 3.66 |
| | | B.SIMD | 3.77 | 7.55 |
| | 256 | gcc | 1.52 | 1.52 |
| | | mkl | 7.65 | 7.65 |
| | | B.SIMD | 4.36 | 9.67 |
| | 1024 | gcc | 1.42 | 1.42 |
| | | mkl | 11.96 | 11.96 |
| | | B.SIMD | 3.96 | 5.65 |
| | 4096 | gcc | 1.23 | 1.23 |
| | | mkl | 12.35 | 12.35 |
| | | B.SIMD | 3.99 | 5.48 |

Boost.SIMD is better than the GCC version while performing worse than MKL on larger sizes. This is because of the lack of unrolling and fine low-level code tuning, necessary to reach the peak performance of the target. The MKL library goes up to 12 GFlop/s in single precision (7.9 in double precision) and outperforms the previous results. The AXPY kernel of MKL is provided as a user function with high architecture optimizations for the Intel processors and introduces an architecture dependency in user code.

Loop optimizations and fine load/store scheduling strategies can be added on top of Boost.SIMD to increase performance. The previous results show that Boost.SIMD provides a portable way to access the latent speed-up of the architectures. However, it is not a special-purpose library like MKL, its performance on this very demanding test is satisfactory yet still far from the peak performance. Other optimizations like loop unrolling and jamming are necessary to compete with the library solutions.

### 5.7.2 Black and Scholes

Listing 5.36 shows the Boost.SIMD code for the Black and Scholes algorithm. The code was tested with single precision floating point numbers.

Listing 5.36: Boost.SIMD version of Black and Scholes

```
template <class A0>
A0 blackandscholes( A0 const &a0, A0 const &a1, A0 const &a2
                  , A0 const &a3, A0 const &a4)
{
  A0 da = simd::sqrt(a2);
  A0 d1 = simd::log(a0/a1)
        + (simd::fma(simd::sqr(a4),simd::Half<A0>(),a3)*a2)/(a4*da);
  A0 d2 = simd::fnms(a4,da,d1);
  return simd::fnms( a1*simd::exp(-a3*a2)
                   , simd::normcdf(d2)
                   , a0*simd::normcdf(d1));
}
```

Figure 5.10 shows the results of this implementation on Excalibur. We can see that SSE2 performs better than the expected ×4 speedup. This is due to the SIMD implementations of `log` and `exp` that are optimized. These implementation perform better than the scalar implementation of the Standard library which leads to higher speedups. The small difference between AVX and AVX 2.0 is due to the use of integers in the implementation of `log` and `exp` while working with IEEE representation. The speedups obtained are then better than the theoretical expected ones.

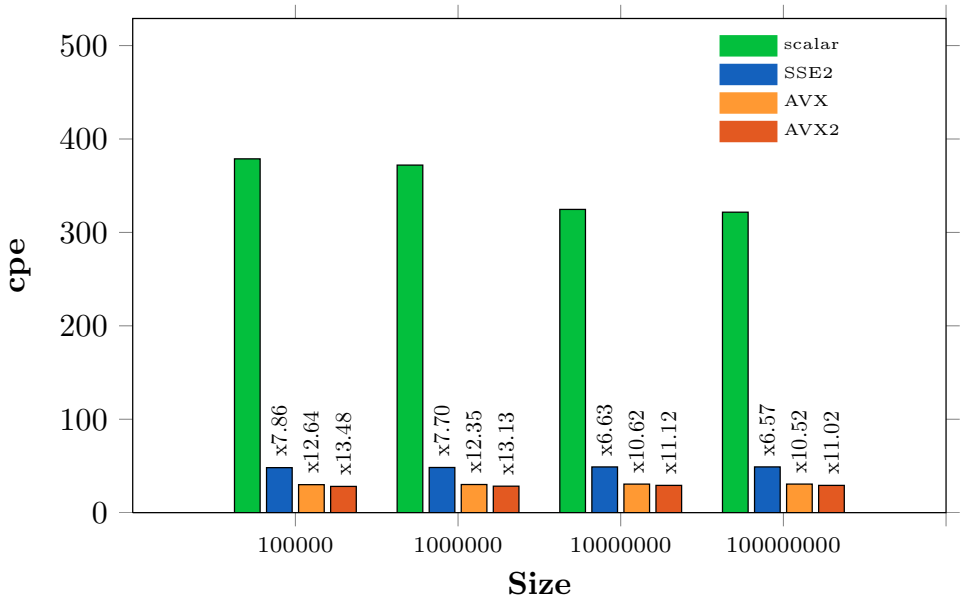



Figure 5.10: Results for Black and Scholes algorithm on Excalibur

The code of this application keeps its genericity and the speedups remain stable while increasing the size of the data set.

### 5.7.3 Sigma-Delta Motion Detection

The Sigma-Delta algorithm [70] can be expressed by a series of additions, subtractions and various boolean selections. As pointed by Lacassagne in [70], the Sigma-Delta algorithm is mainly limited by memory bandwidth and no optimizations beside SIMDization is efficient as only point-to-point operations are issued. Listing 5.37 shows the Boost.SIMD implementation of the Sigma-Delta algorithm.

Listing 5.37: Boost.SIMD version of Sigma Delta

```
template<class Pixel>
Pixel sigmadelta(Pixel &bkg, const Pixel &fr, Pixel &var)
{
  Pixel diff_img, mul_img, zero=0;
  bkg = selinc( bkg < fr, seldec( bkg > fr, bkg ) );
  diff_img = max(bkg, fr) - min(bkg, fr);

  mul_img = adds(adds(diff_img,diff_img),diff_img);

  var = if_else( diff_img != zero, selinc( var < mul_img
                                         , seldec( var > mul_img
                                                 , var
                                                 )
                                         )
                                 , var
                                 );
  return if_zero_else_one( diff_img < var );
}
```

Table 5.6 details how Boost.SIMD performs against scalar versions of the algorithm. The benchmarks use greyscale images. To handle this format, the type `unsigned char` is used and each vector of the SSE4.2, Altivec or AVX extensions can carry 16 elements. On the AVX side, the instruction set is not providing a support for this type so Boost.SIMD emulates such a vector but AVX 2.0 supports integer types and can hold 32 elements.

Table 5.6: Results for Sigma-Delta algorithm in *cpp*

| Extension | SSE4.2 | | Altivec | |
|---|---|---|---|---|
| Size | $256^2$ | $512^2$ | $256^2$ | $512^2$ |
| Scalar C++(1) | 9.237 | 9.296 | 14.312 | 27.074 |
| Scalar C icc | 2.619 | 2.842 | - | - |
| Scalar C gcc | 8.073 | 7.966 | - | - |
| Ref. SIMD(2) JRTIP[70] | 1.394 | 1.281 | 1.380 | 4.141 |
| Boost.SIMD(3) | 1.106 | 1.125 | 1.511 | 5.488 |
| Speedup(1/3) | 8.363 | 8.263 | 9.469 | 4.933 |
| Overhead(2/3) | -26% | -13.9% | 8.7% | 24.5% |

The execution time overhead introduced by the use of Boost.SIMD stays below 8.7%. On SSE4.2, it performs better than the SSE4.2 handwritten version

while on Altivec, a slow-down appears with images of $512 \times 512$ elements. Such a scenario can be explained by the number of images used by the algorithm and their sizes. Three vectors of type `unsigned char` need to be accessed during the computation which is the critical section of the Sigma-Delta algorithm. The 512 KBytes L2 cache of the PowerPC 970FX can not contain the three images in cache. Cache misses becomes preponderant and the Load/Store unit of the Altivec extension keeps waiting for data from the main memory. The L3 cache level of the Nehalem microarchitecture overcomes this problem. The `icc` autovectorizer generates SSE4.2 code with the C version of Sigma-Delta while `gcc` fails. The C++ version keeps its fully scalar properties even with the autovectorizers enabled due to the lack of static information introduced by the Generic Programming Style of the C++ language.

Figure 5.11 shows the frames per second that Boost.SIMD can obtain against the scalar version of the code on Excalibur. We can see that SSE2 provides an average speedup of $\times 4$ and AVX emulation mode performs significantly better. On the other hand, AVX 2.0 provides good speedups that outperforms other extensions due to its wide registers supporting for 8-bit integers. We can easily see the cache memory effects that impacts the speedups for all extensions while increasing the size of images.

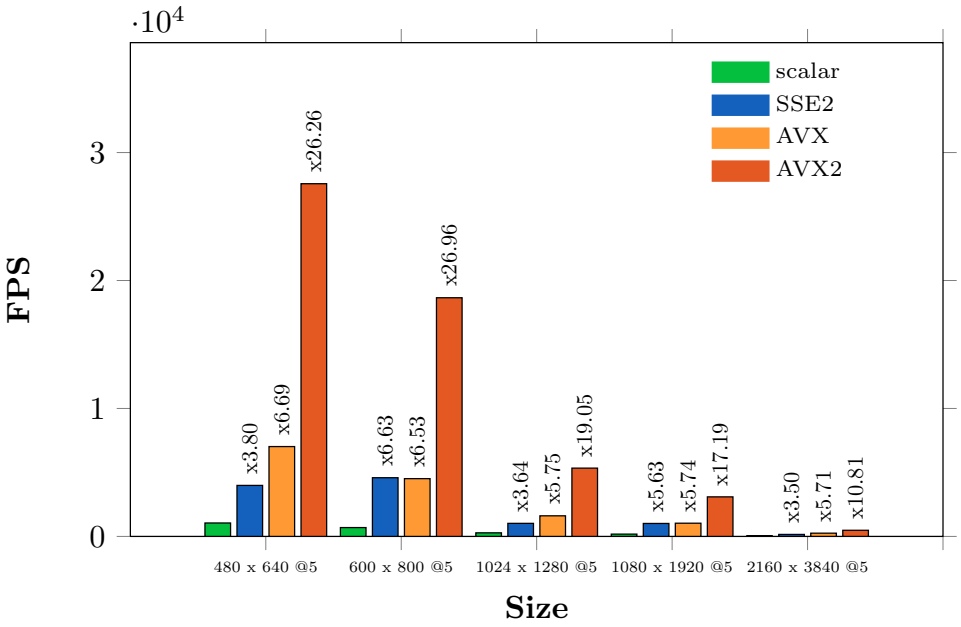



Figure 5.11: Results for Sigma-Delta algorithm on Excalibur

Boost.SIMD keeps the high level abstraction provided by the use of STL code and is able to reach the performance of the vectorized reference code. In addition, the portability of the Boost.SIMD code gives access to the original speedups without rewriting the code.

### 5.7.4 The Julia Mandelbrot Computation

Listing 5.38 and 5.38 show the Boost.SIMD implementation of the Julia Mandelbrot algorithm. In this algorithm we apply the same transformation on points coming from the complex plane. We then define a `step` that will perform this transformation for each of them. It is presented in listing 5.38.

Listing 5.38: Boost.SIMD version of Julia Mandelbrot

```
namespace mandelbrot
{
  struct step
  {
    template<class Sig> struct result;
    template<class This, class A0, class A1>
    struct result<This(A0,A1)>
    { [...] };

    step(std::size_t const& n) : max_iter_(n) {}

    template<class T>
    typename result<step(T,T)>::type operator()(T const& a, T const& b) const
    {
      typedef typename result<step(T const&, T const&)>::type iter_type;
      typedef typename boost::simd::meta::scalar_of<T>::type     s_type;
      iter_type iter = boost::simd::Zero<iter_type>();
      iter_type const o = boost::simd::One<iter_type>();
      T x = boost::simd::Zero<T>();
      T y = boost::simd::Zero<T>();
      T x2, y2, xy, m2;
      typename boost::simd::meta::as_logical<T>::type mask;
      std::size_t i = 0;
      do
      {
        x2 = x * x;
        y2 = y * y;
        xy = s_type(2) * x * y;
        x = x2 - y2 + a;
        y = xy + b;
        m2 = x2 + y2;
        mask = m2 < s_type(4);
        iter = boost::simd::seladd(mask, iter, o);

        i++;
      }
      while(boost::simd::any(mask) && i < 256);
      return iter;
    }
    std::size_t max_iter_;
  };
}
```

We can now apply this `step` to our complex plane. First, the `step` is applied from the first aligned address and then performs aligned memory accesses. We finally finish the processing with a scalar computation. The code of the `step` can be used either for SIMD or scalar computations as Boost.SIMD enables such a genericity. This makes the algorithm independent of any architecture details. Listing 5.39 presents the call to the `julia` step.

Listing 5.39: Call to the Julia Mandelbrot step

```
// [...]
mandelbrot::step julia(256);
std::size_t step_size_ = boost::simd::meta::cardinal_of<type>::value;
std::size_t aligned_sz = size_ & ~(step_size_-1);
std::size_t it = 0;

for(std::size_t m=aligned_sz; it != m; it+=step_size_)
{
  type A_pack = (&A[it]);
  type B_pack = (&B[it]);
  aligned_store(julia(A_pack, B_pack), &C[it]);
}

for(std::size_t m=size_; it != m; it++)
  C[it]=julia(A[it],B[it]);
```

Figure 5.12 presents the results obtain on Excalibur. The Julia Mandelbrot algorithm does not present any particular shortcomings for SIMD computations so we expect that the speedups will be close to the theoretical ones. SSE2 reaches 75% of the theoretical speedup. AVX and AVX 2.0 are able to store twice more elements and they double the SSE2 speedup as expected.

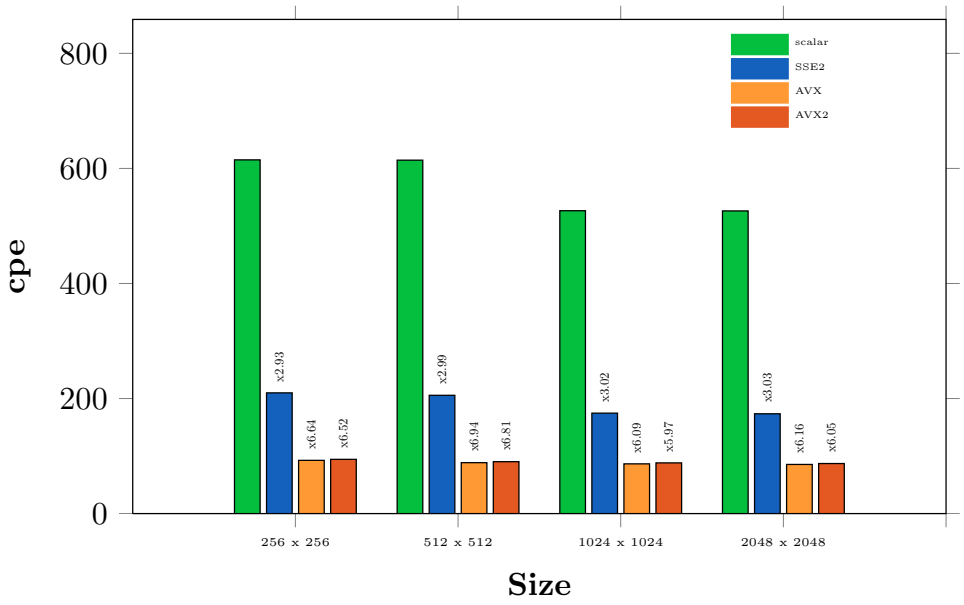



Figure 5.12: Results for Julia Mandelbrot algorithm on Excalibur

## 5.8 Conclusion

Building a library for SIMD extensions with a high level API without loss of performances is not a simple task. Especially when the library needs to be designed in an extensible way for further architecture support. The conception of multi-architectural tools faces the challenge of the integration of architecture specific optimizations within a generic approach. Such a library design is limited by the possibilities offered by the host language. With C++ and its generic capabilities, this approach can be explored and Boost.SIMD is an example of it.

Boost.SIMD relies on template metaprogramming techniques and more generally on Boost.Dispatch. The library provides a new abstraction for SIMD based code, its main contributions are the following:

- The SIMD register abstraction combined with high level functions makes SIMD computation easy to write and portable over architectures. Its API fits the Standard requirements and is fully compatible with C++ Standard based code. So already existing code can take advantages of SIMD speedups without a lot of effort.
- Boost.SIMD is designed for an easy architecture support. With a generic framework for adding new extensions and injecting architecture specific optimizations in the evaluation process, the library provides extensibility and maintainability.
- The benchmarks show a efficient implementation with similar performances compared to handwritten SIMD code.

Boost.SIMD demonstrates that genericity and performance can be reach for SIMD code generation without sacrificing a standard integration.

Chapter 6

# NT2: an Architecture Aware DSEL Framework

## Contents


**6.1 The NT2 Programming Interface . . . . . 102**
6.1.1 Basic API . . . . . 103
6.1.2 Indexing and data reshaping . . . . . 103
6.1.3 Linear Algebra support . . . . . 104
6.1.4 `table` settings . . . . . 104
6.1.5 Compile-time Expression Optimization . . . . . 105
6.1.6 Parallelism Handling . . . . . 106
**6.2 Implementation . . . . . 106**
6.2.1 Function compile-time descriptors . . . . . 107
6.2.2 Compile-time architecture description . . . . . 107
**6.3 Expression Evaluation with NT2 . . . . . 111**
**6.4 Benchmarks . . . . . 112**
6.4.1 Basic Kernels . . . . . 112
6.4.2 Black and Scholes . . . . . 114
6.4.3 Sigma Delta Motion Detection . . . . . 115
6.4.4 The Julia Mandelbrot Computation . . . . . 116
**6.5 Conclusion . . . . . 118**


In chapter 5 we presented Boost.SIMD, a library that aims at facilitating access to SIMD extensions with a simple interface without losing the benefits of such a powerful hardware feature. On top of SIMD computation, other types of parallelism are available and need to be exploited. Modern architectures present multi-cores and accelerator based systems. Embedded systems also start to use more powerful parallel components. These levels of parallelism nowadays can't be ignored in the development of applications.

Regarding this context, this chapter[1] will present the Numerical Template Toolbox ($NT^2$), a C++ library which aims at simplifying the development of high

[1] This chapter is extended from the upcoming work accepted for publication in the Journal of Parallel and Distributed Computing.

performance numerical computing applications with a multi- architectural support. First, we will introduce the challenges of such a library and present its programming interface. Then, the core of the library will be detailed and a case analysis will be shown to illustrate the implementation of $NT^2$ . Finally, benchmarks will assess the performances of the library.

As discuss in chapter 2, architecture aware programming requires expertise. From the diversity of parallel programming tools to low level architecture oriented optimizations, non expert programmers face a lack of expressiveness in most of today's solutions. In chapter 3, we introduced the Domain Specific Language approach and decided to design a *DSEL* for High Performance Computing (HPC).

- **High expressiveness**

  The first challenge for such an approach is to provide a high level of expressiveness that provides an easy programming process for the user. It will also keep the expressiveness of the algorithm as the *DSEL* needs to be designed for the said domain. With expressiveness comes the challenge of designing an intuitive interface. This API must express the domain in the best way. The semantic of the *DSEL* is the key to well designed *DSEL* .

- **Performances**

  After expressiveness comes the performance of the tool. Performances need to be on par with an optimized code on a specific architecture. The *DSEL* should not introduce an overhead and if it does, this overhead should stay reasonable. An automatic code generation process can introduce some loss of performance but this quantity should stay as low as possible to reduce the development time. To keep the performance close to an optimized code, the best evaluation strategy needs to be selected by the code generation system. Such an ability requires the knowledge of the targeted architecture during the evaluation process of the *DSEL* .

## 6.1 The NT2 Programming Interface

$NT^2$ has been designed to be as close as possible to the MATLAB language. Ideally, a MATLAB to $NT^2$ conversion process should be limited to copying the original MATLAB 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 versions 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.

### 6.1.1 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 functions ranging from standard arithmetic, exponential, hyperbolic and trigonometric functions, bitwise and boolean operations, IEEE related functions, various pattern generators and some statistic and polynomial functions. All those functions support vectorization thanks to BOOST.SIMD[44, 45] (see chapter 5). Moreover, and contrary to most similar library, NT$^2$ provides support for all real and integral types, both real or complex. Combined with the large set of functions available, this allows NT$^2$ to be used in a wider variety of domains.

Listing 6.1: Sample NT$^2$ code

```
1 table<double> A1 = _(1.0,1000.0);
2 A2 = A1 + randn(size(A1));
3 double rms = sqrt( sum(sqr(A1(_) - A2(_))) / numel(A1) );
```

Listing 6.1 shows 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`. Listing 6.2 shows the corresponding MATLAB code.

Listing 6.2: Corresponding MATLAB code

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

### 6.1.2 Indexing and data reshaping

Indexing and reshaping of data is one of the main assets of the MATLAB language as it maximizes the expressiveness of array-based expressions. In NT$^2$, accessing parts of a table is done with `operator()` which handles various indexing values: integer and table of integers, range created by the `colon` function (`_` for short) or contextual keywords like `begin_` and `end_`. Arbitrary extraction, dimension reinterpretation, shifting, and stencil computations can be expressed with that syntax. Listing 6.3 shows how a Jacobi update step can be written using such indexing.

Listing 6.3: Cross stencil for the update step of the Jacobi method with NT$^2$

```
1 new_(_(begin_+1, end_-1), _(begin_+1, end_-1))
2   = (   old_(_(begin_  , end_-2), _(begin_+1, end_-1))
3       + old_(_(begin_+2, end_)  , _(begin_+1, end_-1))
4       + old_(_(begin_+1, end_-1), _(begin_  , end_-2))
5       + old_(_(begin_+1, end_-1), _(begin_+2, end_  ))
6     )/4.f;
```

### 6.1.3 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 take care of maximizing the memory reuse of output parameters so the minimal amount of copies and allocations are performed.

### 6.1.4 `table` settings

`table` can be parametrized by special settings. It allows the user to specify statically some properties of the `table` class. As these settings are statically known, NT$^2$ can select the right evaluation strategy according to the settings.

`table` comes with the following list of settings:

- **Allocators:** `my_allocator<float>`. The user can provide a specific standard based allocator like shown in listing 6.4.

Listing 6.4: NT$^2$ `allocator` setting example

```
table< float, my_allocator<float> > t(104);
```

- **Static size:** `of_size<...>` allows a `table` to have a static size known at compile-time. Listing 6.5 illustrates this setting.

Listing 6.5: NT$^2$ `of_size<...>` setting example

```
table< float, of_size<4,4> > t;
```

- **Indexing:** `C_index_`, `matlab_index_`. The indexing policy of `table` can be changed to a C like policy. `matlab_index_` is the default one. Listing 6.6 shows how to declare a `table` with a C like index.

Listing 6.6: NT$^2$ `C_index_` setting example

```
table<float, C_index_> t(104);
[...] // Fill the table
for(int i = 0; i < 104; ++i) // C indexing
  { cout << t(i) << endl; }
```

- **Interleaved data:** `interleaved_, deinterleaved_`. If the `table` contains interleaved data, it can be statically specified like in listing 6.1 and the correct SOA or AOS loading strategy will be performed.

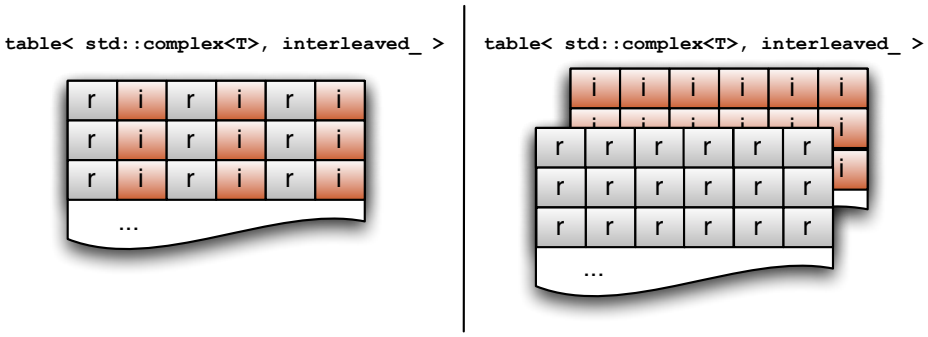



Figure 6.1: `table` with `interleaved_` and `deinterleaved_` data

- **Shape:** `rectangular_, diagonal_`, etc. If the `table` has a particular shape like the diagonal one, it can be specified (see listing 6.7) and the memory allocation will be optimized for this shape.

Listing 6.7: NT$^2$ `diagonal_` setting example

```
1 table< float, diagonal_ > t(4,4);
```

- **Sharing memory:** `shared_, owned_`. A `table` can own its memory but it can also share memory with an external component like in listing 6.8.

Listing 6.8: NT$^2$ `shared_` setting example

```
1 float data[] =  { 1,2,3,4,5,6 };
2 table<float, settings(shared_)> x(of_size(3,2), share(data));
```

- **Setting composition:** NT$^2$ has the ability of composing several settings. Listing 6.9 shows the instantiation of a diagonal and C index based table.

Listing 6.9: NT$^2$ setting composition example

```
1 table<float, settings(diagonal_, C_index_)> x(of_size(3,3));
```

### 6.1.5 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. This compile-time expression optimization is similar to the one introduced in Boost.SIMD. Considered optimizations include:

- Fixed-point transformations like `trans(trans(x))` or other functions combinations that can be precomputed as being equivalent to a simpler function;
- Fusion of operations like `mtimes(a, trans(b))` which can directly notify the GEMM BLAS primitive that `b` is tranposed;
- Architecture-driven optimizations like transforming `a*b+c` into `fma(a,b,c)`.
- Sub-matrix access like `a(_,i)` into an optimized representation enabling vectorization.

### 6.1.6 Parallelism Handling

Table 6.2 sums up the difference and similarities between $NT^2$ and the libraries introduced in 2.2.4. For the shared memory parallelism, $NT^2$ supports two backends: OpenMP and Intel Threading Building Blocks.

| **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 | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |

Figure 6.2: Feature set comparison between $NT^2$ and similar libraries

## 6.2 Implementation

$NT^2$ is a Expression Template based *DSEL* that uses BOOST.PROTO (see section 3.2.3), BOOST.DISPATCH (see chapter 4) and BOOST.SIMD (see chapter 5). BOOST.PROTO is used as its expression template engine and 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. The expression evaluation strategy of $NT^2$ is driven by the a *AA-DEMRAL* methodology introduced in chapter 3). It is based on:

- a strategy to select the proper implementation of a given function according to a given architecture and the function properties, a compile-time description of function properties, an architecture description *DSEL* describing architectures and their relationship. All of this is based on BOOST.DISPATCH;
- a compile-time process for rescheduling $NT^2$ statements in a way that optimal loop nests can be generated;
- a parallel code generator using parallel skeletons that takes care of different types and levels of parallelism.

### 6.2.1 Function compile-time descriptors

NT$^2$ uses Boost.Dispatch to handle every function call. Each function call is resolved by Boost.Dispatch as a call to a function object handling the tags computation and dispatching, leaving the user API clear of any implementation leaks. The first level of gathered information is function properties. Each NT$^2$ function (as a symbol) is tied to a Boost.Dispatch **tag**. Whenever a function `foo` is called, NT$^2$ tries to find a valid implementation of `foo` by calling a Boost.Dispatch function overloaded for a descriptor class `foo_`. Those tags include:

- elementwise functions that operate on their arguments at a certain position, without dependencies between operations on different positions. They are the core of NT$^2$ expressions, and combining them results into a single kernel or loop nest. They include: regular function like `plus` or `sin`, data generators like `colon` or `zeros`, functions modifying a table logical size like `reshape` or `diag`;
- non-elementwise functions, which output can not be combined with an elementwise function but which input is still combinable. Their properties and parallel potential depends on the considered functions. They include reduction and partial reduction functions, scan functions like `cumsum` and external kernels.

Listing 6.10: Function descriptors for some NT$^2$ functions

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

As an example, listing 6.10 presents the descriptors for various functions. `plus` is registered as a classical elementwise operation. `sum` is a reduction and its descriptor defines it as a reduction based on `plus` and `zero`. Then, the matrix-matrix product function is registered as an external kernel.

### 6.2.2 Compile-time architecture description

Once a proper function implementation has been selected for either a concrete function tag or for a more general function family, we need to select the best implementation for the current architecture. NT$^2$ uses Boost.Dispatchability to describe an architecture as a compile-time tag, similar to the function descriptors (figure 6.11).

Listing 6.11: Some NT$^2$ architecture descriptors

```
1 struct cpu_   : unspecified_<cpu_> {}; // cpu_: no special info
2 struct simd_ : cpu_ {};               // simd_: any SIMD architecture
3
4 // shared memory architecture using OpenMP as runtime
5 template<typename Core> struct openmp_  : Core {};
```

For every supported architecture, a descriptor is defined using inheritance to organize related architecture components. In addition to this inheritance scheme, architectures descriptors can 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 nested code at 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 will be generated through combination of the OpenMP and AVX backends (section 6.2.2.1). This compile-time architecture description extends the SIMD architecture hierarchy available in Boost.SIMD.

#### 6.2.2.1 Parallel code generation

The code generation presented here works in a similar way that the one introduced in Boost.SIMD (see chapter 5). The **look- ahead** optimization scheme introduced by Boost.SIMD is also reused in NT$^2$ leading to the detection of expression patterns that are candidates for an architecture specific optimization.

The functions and architecture descriptors introduced in section 6.2.1 schedule the evaluation of each type of nodes (*i.e.* functions) involved in a single statement. The NT$^2$ code generator will generate successions of loop nests based on the top level AST node descriptor. The NT$^2$ expression evaluation is based on the possibility to compute the size and value type. This size is used to construct a loop, which can be parallelized using arbitrary techniques, which then evaluates 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 function. `run` takes care of selecting the best way to evaluate 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 and dispatch to a specific loop nest generator for each family of functions (elementwise, reduction, etc).

To take the architectural information into account at this point, NT$^2$ relies on **Parallel Skeletons** [22]. Parallel skeletons are recurrent parallel patterns designed as higher-order functions that describe an efficient solution to a specific problem. Cole details in [23] the need for specific skeletons providing enough abstraction to be used in the context of parallel frameworks. This abstraction can introduce semantic information that will help the composition of a skeleton abstraction with an efficient implementation of the corresponding skeletons. Aldinucci addresses these approach with an expandable skeleton environment called Muskel [5]. The abstraction/efficiency trade-off of skeleton based programming has been explored by Kuchen in [68] and it shows that such an approach can lead to efficient library based implementations without losing levels of abstraction.

Skeletons usually behave as higher order functions, *i.e.* functions parametrized by other functions, including other skeletons. This composability reduces the difficulty of designing complex parallel programs as any combination of skeletons is viable 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: first, skeletons can be specified in an abstract manner and encapsulate architecture specific implementation; second, the communications/computations patterns are known in advance and can be optimized [6, 41].

Even if a large number of skeletons have been proposed in the litterature [68, 20], NT$^2$ focuses on three data-oriented skeletons:

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

Those skeletons are tied to families of loop-nest that can or can not be nested. Those families are :

- **elementwise loop nests** that represent loop nests implementable via a call to **transform** and which can only be nested with other elementwise operations.
- **reduction loop nests** that 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** that represent loop nests implementable via a call to **scan**. Successive prefix scans, like reductions, are not nestable but can contain nested elementwise loop nests.

Those families of loop nests are used to tag functions provided by NT$^2$ so that the type of the operation itself can 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 has to take care of separating expressions requiring non-nestable loop nests 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 a compatible descriptor. Two nodes have compatible descriptors 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 6.3 shows how the expression `A = B / sum(C+D)` is built and split into sub-ASTs handled by a single type of skeleton.

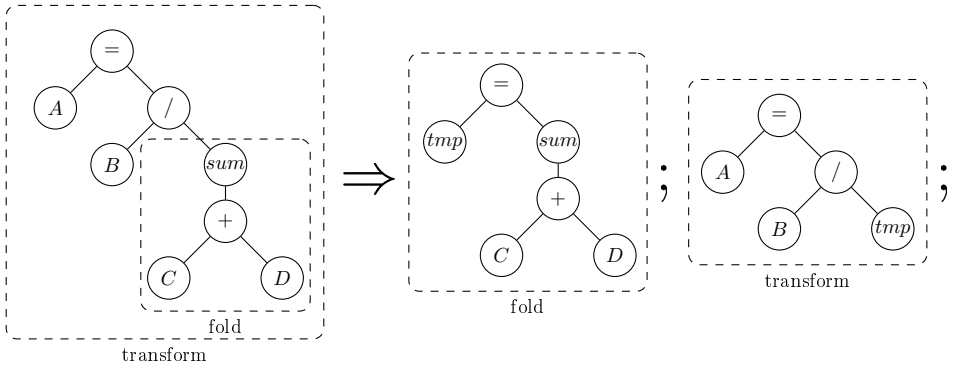



Figure 6.3: Parallel Skeletons extraction process
*Nesting of different kinds of skeletons into a single statement is automatically unwrapped at compile time as a sequence of single skeleton statements.*

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 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 nestability of parallel skeletons to call the SIMD and/or scalar version of each skeleton involved in a serie 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 skeleton is a $NT^2$ function object. They are handled by Boost.Dispatch 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 to this evaluation by describing step by step the evaluation of a $NT^2$ expression.

## 6.3 Expression Evaluation with NT2

In this section we detail the exact scenario for the evaluation of a NT$^2$ DSL expression. As an 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 general code generation process. As every node in this expression are elementwise operations, `run` will select `transform` as the skeleton to use. The current architecture descriptor being `openmp_<avx_>`, run forward to the OpenMP version of `transform` as shown in listing 6.12.

Listing 6.12: 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)
  {
    ntd::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 descriptor of its inner core, the OpenMP `transform` only deals with laying out the needed OpenMP structure around a call to its inner architecture `transform` version. This recursive definition limits the amount of code to write to handle architecture combinations as each skeleton implementation is only responsible to generate current architecture code. In this case, the OpenMP layer will take care of computing the optimal block size for 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 allow us to use a single API for all our SIMD related code (see figure 6.13).

Listing 6.13: 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+=N)
    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 can be actually vectorized and call the scalar version on the left-over data by using the scalar version of `transform`. Once done, the code generated will automatically perform the required parallel operations. The final 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 guarantee that the abstraction cost of the system is negligible.

## 6.4 Benchmarks

This section presents the execution time of various benchmarks to give an idea of the performance attainable with NT$^2$ with different scenarios. The first benchmark, inspired from 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 LAPACK bindings. Then, three more complex application kernels evaluate NT$^2$ performance under realistic conditions, their descriptions are detailed in appendix A. All benchmarks were run over thousands of executions from which the median execution time has been kept as the end result. When possible, results are compared with an equivalent kernel implemented using a selection of similar library or with the 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 those performance benchmarks. Their descriptions can be found in appendix B.

### 6.4.1 Basic Kernels

Basic kernel benchmarks aim at validating that NT$^2$ basic features perform correctly against state of the art libraries.

#### 6.4.1.1 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 cycles per computed element, are given in table 6.4. We see that $NT^2$ performance are comparable to those of SIMD enabled library like 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 6.4: Elementwise benchmark using SSE 4.2 on **Mini-Titan**

#### 6.4.1.2 BLAS operations

This benchmark evaluates the efficiency of the BLAS binding of $NT^2$ by performing a chain of three matrix-matrix product of decreasing size:

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

Results, in cycles per computed element, are given in table 6.5. We see that $NT^2$ performance are comparable to the performance of other libraries. Armadillo exhibits the best performance due to its matrix-matrix product reordering phase.

| **Scale** | Armadillo | Blaze | Eigen | MTL | uBlas | $NT^2$ |
|---|---|---|---|---|---|---|
| $100 \times 20$ | 59.0 | 154.2 | 88.40 | 126.7 | 77.82 | 79.11 |
| $2000 \times 400$ | 91.9 | 204.1 | 216.6 | 211.2 | 172.8 | 177.4 |

Figure 6.5: GEMM kernel benchmarks using MKL on **Mini-Titan**

#### 6.4.1.3 LAPACK operations

This benchmark assesses 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:

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

| **Scale** | C LAPACK | $NT^2$ LAPACK |
|---|---|---|
| $1024 \times 1024$ | 75 | 75 |
| $2048 \times 2048$ | 149 | 148 |

Figure 6.6: GESV kernel benchmarks on **Mini-Titan**

Table 6.6 show the GFLOPS rate attained by using either direct C++ calls to LAPACK and to the corresponding $NT^2$ code. Results shows that the overhead against the direct call to the LAPACK version of the kernel is negligible.

### 6.4.2 Black and Scholes

The code of the Black and Scholes algorithm is defined in figure 6.14. The Black & Scholes algorithm involves multiples high latency and high register count operations. The SIMD version of `log`, `exp` and `normcdf` use polynoms and precision refinement step that consume a large amount of registers.

Listing 6.14: Black & Scholes NT$^2$ implementation

```
1  table<float> blackscholes ( table<float> const& S,table<float> const& X
2                            , table<float> const& T,table<float> const& r
3                            , table<float> const& v
4                            )
5  {
6    table<float> d  = sqrt(T);
7    table<float> d1 = log(S/X)+(fma(sqr(v),0.5f,r)*T)/(v*d);
8    table<float> d2 = fma(-v,d,d1));
9
10   return S*normcdf(d1)-X*exp(-r*T)*normcdf(d2);
11 }
```

The Black & Scholes algorithm involves multiples high latency and high register count operations. The SIMD version of `log`, `exp` and `normcdf` use polynoms and precision refinement step that consume a large amount of registers. Results shown on figure 6.7 demonstrates that our SIMD implementation hits roughly 65% of the peak speed- up in SIMD due to the important number of spilled variables. The speed-ups of the multi-threaded versions go up to 90% of the peak speed-ups. When combining SIMD and OpenMP, the gain raises but the workload of the SIMD computation units is still to heavy.

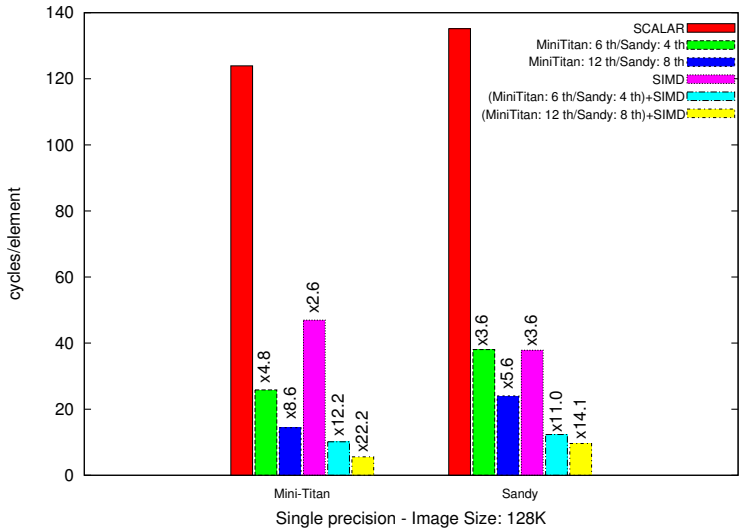



Figure 6.7: Black&Scholes Results in single precision

### 6.4.3 Sigma Delta Motion Detection

This algorithm is composed of point-wise operations with two double if-then-else patterns. Its SIMD implementation is not straightforward, as the multiplication and the absolute difference require to promote 8-bit data to temporary 16-bit data or use saturated arithmetic. Its low arithmetic intensity always leads to Memory Bound implementations.

Listing 6.15: Sigma-Delta $NT^2$ implementation

```
1  background = selinc( background < frame
2                     , seldec( background > frame, background )
3                     );
4
5  diff      = max(background, frame) - min(background, frame);
6  sigma3    = muls(diff,uint8_t(3));
7
8  variance = if_else( diff != uint8_t(0)
9                    , selinc( variance < sigma3
10                             , seldec(variance > sigma3, variance)
11                             )
12                   , variance
13                   );
14
15 detected = if_zero_else_one( diff < variance );
```

The main challenge for $NT^2$ is also to preserve performance despite a multi-statement implementation that, when compiled for OpenMP, leads to spurious barriers. 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 C.2.1.

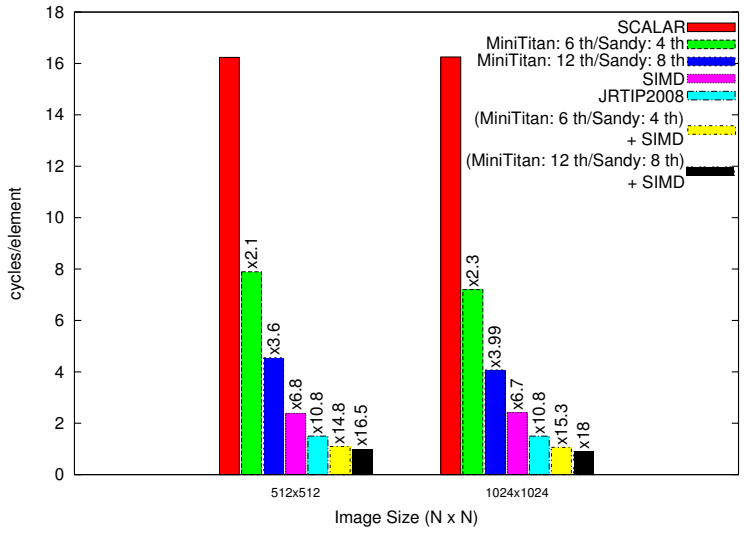



Figure 6.8: Sigma Delta Results

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 6.8). With many load and store operations, the strong scalability of the algorithm can not be preserved. When SSE is enabled, both versions (single-threaded-and multi-threaded) of the code increase their efficiency until hitting the maximum of the memory bandwidth. The SIMD only version is one cycle slower than the handwritten optimized one. This loss comes from very fine grain optimizations introduced in the code. Typically, the difference image 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.

### 6.4.4 The Julia Mandelbrot Computation

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

- It uses complex arithmetic, but to ensure performance the code should 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, $NT^2$ 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 6.9 and 6.10 illustrates 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 the theoretical ones and the efficiency is greater than 80%. For **Sandy**, the efficiency decreases due to generation of the Julia Space. In fact, this generation relies on SIMD integer support which is not available on AVX. If we remove this phase from the benchmark, the speedups are raising to the expected ones. The integer support for AVX will be addressed in the future to avoid this loss of performance.

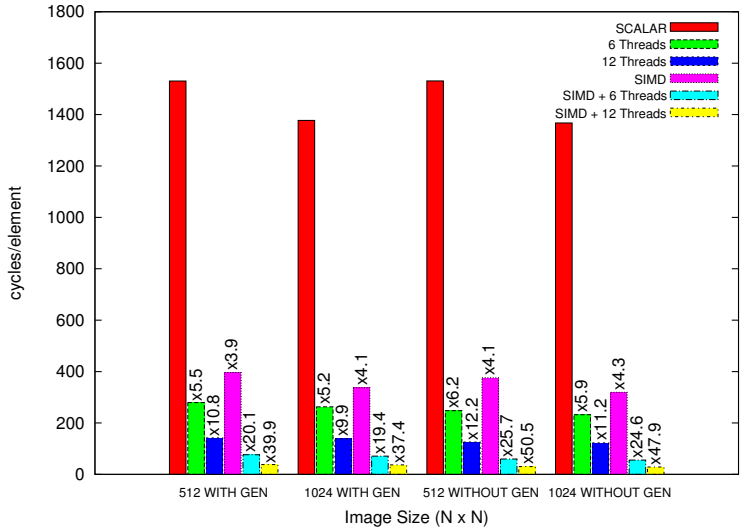



Figure 6.9: Mandelbrot Results in single precision on **Mini-Titan**

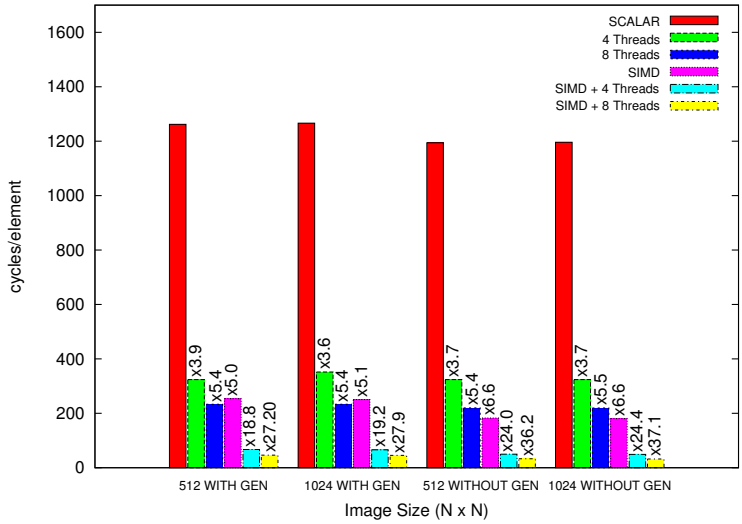



Figure 6.10: Mandelbrot Results in single precision on **Sandy**

## 6.5 Conclusion

Designing a high level programming tool for High Performance Computing is not an easy challenge. We show that NT$^2$ has a nice and simple API that guarantee a high level of abstraction. While keeping expressiveness at its maximum, NT$^2$ takes advantage of the architecture informations to deliver a high level of performance. It allows portability over various architectures and provides a systematic way of implementing new architectural supports. Its generic internal framework permits an easy extensibility of the *DSEL* . Our benchmarks that show both on simple and complex task that NT$^2$ is able to deliver performance within the range of state of the art implementation.

NT$^2$ uses expression template techniques and generative programming. It also relies on Boost.Dispatch and Boost.SIMD. Its main contributions are the following:

- Generative Programming helps implementing more flexible scientific computing software with a very high level of abstractions and high efficiency.
- Generic programming inside NT$^2$ permit an easy multi-architectural support for today's architectures.
- The benchmarks show a efficient code generation system.

Designed as an active library, NT$^2$ proposes a solution for the design of high level programming tool with multi-architectural support.

Chapter 7

# Conclusions and Perspectives

Contents

7.1 Conclusions . . . . . 119
7.2 Perspectives . . . . . 120

## 7.1 Conclusions

In this thesis we have presented two active libraries that aims at simplifying the development of high performance applications. The main objectives that were considered for the design of these two tools are:

- A high level abstraction coupled with a high expressiveness for designing applications that take advantage of parallel architectures;
- An efficient code generation process leading to performance close to a hand written parallel code;
- An easy extensibility of the libraries via a generic approach for their design.

Our contribution is based on several approaches. First, we have looked at the solutions available in our era of interests and we have studied their advantages and drawbacks. Then, we have proposed a new methodology with its implementation. Finally, we have validated the efficiency of our contributions by measuring the execution time of well known applications in various domains. This final chapter summarizes synthetically our contributions and gives some perspectives for future research work.

Developing parallel applications that take advantages of architectural features is not a trivial task. Many solutions exist with different approaches to face this problem. The existing tools propose a balance between expressiveness and performance that leads to prevail one of the two. To alleviate this compromise, we studied new techniques that permit an easy design for an architecture aware tool with high expressiveness and performance.

The first step of our contribution is the new methodology called *AA-DEMRAL* . We inject the architecture specifications inside the evaluation process of a *DSEL* . To implement this new methodology, we use C++ as a host language for our *DSEL* because it presents techniques that combine expressiveness with performance. Generic and generative programming are emerging techniques that allow to couple a high level of abstraction with a driven code generation process within the compiler. Such techniques are possible through the template mechanism of C++ . Thus we use Boost.Dispatch, Boost.SIMD[44, 45] and $NT^2$ [42]. Boost.Dispatch is a function dispatching library with a generic tag dispatching technique that has the ability to dispatch a function call according to its arguments, its properties as a function and an architecture specific information. Boost.SIMDis a high level programming tool for SIMD extensions and its implementation uses Boost.Dispatch. $NT^2$ is a C++ library providing a *DSEL* which aims at simplifying the development of high performance numerical computing applications with a multi- architectural support. $NT^2$ 's implementation relies on Boost.Dispatchand Boost.SIMD.

Boost.SIMD and $NT^2$ are two high level tools that provide a high abstraction for the development of high performance applications. Boost.Dispatch has proved its ability to simplify the multi-architectural support for parallel programming tools by being successfully used inside Boost.SIMD and $NT^2$ . Boost.SIMD demonstrates its capability for an easy instruction level parallelism code writing. $NT^2$ shows that expressiveness allows to tie domain specific informations with optimization strategies. These three libraries illustrates the power of generic programming for building abstractions that simplify architecture aware programming.

On the effectiveness side, Boost.SIMD and $NT^2$ show the performance of their code generation systems with a relevant number of tests. From basic kernels to real applications coming from different domains, the benchmarks demonstrate the capabilities of our three libraries to combine expressiveness and performance. The results obtained are comparable to hand written and optimized code from the state of the art. Both tools have proved the effectiveness of the code generation process introduced by our new methodology. Boost.SIMD has also been successfully used inside industrial code. The efficiency of our approach is validate by our implementations and the experiments illustrate a relevant tool designing approach.

## 7.2 Perspectives

From these results, several perspectives and research interests are possible for the future.

Boost.SIMD currently include support for x86 processors and Altivec based architectures. The ARM version is currently under development. Targeting other architectures like the Xeon Phi or DSP based architecture is the next step. Another

feature we want to add is the ability of `pack` to work with SIMD registers when the user asks for a `pack` with a wide cardinal. For example, a `pack` with 8 elements should trigger the use of two SSE2 registers if this extension is available. Another step in the AST exploration system is to estimate the proper unrolling and blocking factor for any given expression. This requires a really fine architecture specification to be able to correlate in a efficient way the functions properties with a certain level of unrolling.

For $NT^2$, the architecture support can be extended. A generalized support for distributed and shared memory system will allow $NT^2$ to target a wider range of applications. This can be done by using different backends. MPI may be a first step but some implementation considerations are not fitting properly with $NT^2$ like global barriers and the MPI environment handling. Using an asynchronous runtime back-end like Charm++ [64] or HPX [32] would allow $NT^2$ to take advantage of non blocking runtimes. With an asynchronous approach, the limitation of barriers while evaluating multiple statements can be solved. Asynchronous evaluation is a good candidate in this aspect. With OpenMP, this limitation still exists as we evaluate multiple statements in different loop-nests. We consider the introduction of a syntax for explicit loop fusion in order to avoid multiple barriers and maximize locality in multi-statements code. Still on the loop- nest limitation we are thinking of exploring the benefits of embedding a meta- programmed subset of the polyhedral model [12, 46] inside the $NT^2$ skeleton system to refine the combination of loop nests that can be generated.

$NT^2$ relies on the availability of a native C++ compiler for the targeted architecture. Support for multi-stage programming [40] will allow $NT^2$ to target system like DSPs on which no C++ compiler is natively available. Porting $NT^2$ to new architecture like the Xeon Phi and OpenCL based Altera reconfigurable systems is interesting to us as multiple accelerator based architecture are rising nowadays. GPGPU are also good candidates for the architectural support of $NT^2$ but their integration to our framework need a non trivial lifting phase. Such an addition to the framework is taken into consideration. Working with different backends like Cuda or OpenCl implies the use of different programming techniques and models. We need to synthesize how accelerators can be added to the framework in a generic manner and multi-stage programming can be a first step to achieve this.

The results presented in this typescript proved the efficiency of our libraries. Parallel programming is a fast growing world and tools need to be adaptable. The main perspectives presented in this last section have one common point which is: the simplicity and the genericity of our tools must keep expressiveness and performances close.

Appendix A

# Algorithms Description

## A.1 AXPY Kernel

It is a basic linear algebra subprogram. The AXPY kernel computes a vector- scalar product and adds the result to another vector. The algorithm is the following :

$$Y \leftarrow \alpha X + Y.$$

This algorithm is part of the BLAS libraries and is really often used in linear algebra algorithms.

## A.2 Black and Scholes

The Black and Scholes algorithm [75] represents a mathematical model able to give a theoretical estimate of the price of European- style options. In this mathematical model, the price of the option is a stochastic process in real time. The full algorithm is describe in algorithm 1.

**Input**: $S$, Spot price
**Input**: $X$, Strike (exercise) price
**Input**: $r$, Interest rate
**Input**: $\sigma$, Standard deviation of the underlying asset, eg stock
**Input**: $time$, Current date
$time\_sqrt \leftarrow \sqrt{time}$
$d1 \leftarrow \frac{\log \frac{S}{X} + r \times time}{0.5 \times \sigma \times time\_sqrt}$
$d2 \leftarrow d1 - (\sigma \times time\_sqrt)$
$c \leftarrow S \times \text{Normal\_distribution}(d1) - X \times \mathrm{e}^{-r \times time} \times \text{Normal\_distribution}(d2)$

**Algorithm 1:** Black and Scholes algorithm

## A.3 Sigma-Delta Motion Detection

The Sigma-Delta algorithm [70] is a motion detection algorithm used in image processing to discriminate moving objects from the background. Contrary to simple thresholded background subtraction algorithms, Sigma-Delta uses a per-pixel variance estimation to filter out outliers due to lighting conditions or contrast variation inside the image. The full algorithm is describe in 2. Figure A.1 shows images taken from a sequence. Pictures in row (1) present the original sequence. Row (2)

illustrates the output of the Sigma-Delta algorithm. We can see on row (2) that the algorithm outputs noise which can be disrupting when tracking objects for example. The typical approach consists in adding a morphological post processing which is illustrated in row (3).

**Input**: $I_t$ current image, $M_t$ previous background
**Result**: $M_t$ the current background, $E_t$ motion mask
**foreach** *pixel* $x$ **do** [step #1: $M_t$ estimation]
  **if** $M_{t-1}(x) < I_t(x)$ **then** $M_t(x) \leftarrow M_{t-1}(x) + 1$
  **if** $M_{t-1}(x) > I_t(x)$ **then** $M_t(x) \leftarrow M_{t-1}(x) - 1$
  **otherwise** $M_t(x) \leftarrow M_{t-1}(x)$
**foreach** *pixel* $x$ **do** [step #2: $O_t$ computation]
  $O_t(x) = |M_t(x) - I_t(x)|$
**foreach** *pixel* $x$ **do** [step #3: $V_t$ update]
  **if** $V_{t-1}(x) < N \times O_t(x)$ **then** $V_t(x) \leftarrow V_{t-1}(x) + 1$
  **if** $V_{t-1}(x) > N \times O_t(x)$ **then** $V_t(x) \leftarrow V_{t-1}(x) - 1$
  **otherwise** $V_t(x) \leftarrow V_{t-1}(x)$
  $V_t(x) \leftarrow max(min(V_t(x), V_{max}), V_{min})$
**foreach** *pixel* $x$ **do** [step #4: $\hat{E}_t$ estimation]
  **if** $O_t(x) < V_t(x)$ **then** $\hat{E}_t(x) \leftarrow 0$
  **else** $\hat{E}_t(x) \leftarrow 1$

**Algorithm 2:** algorithme $\Sigma\Delta$ initial

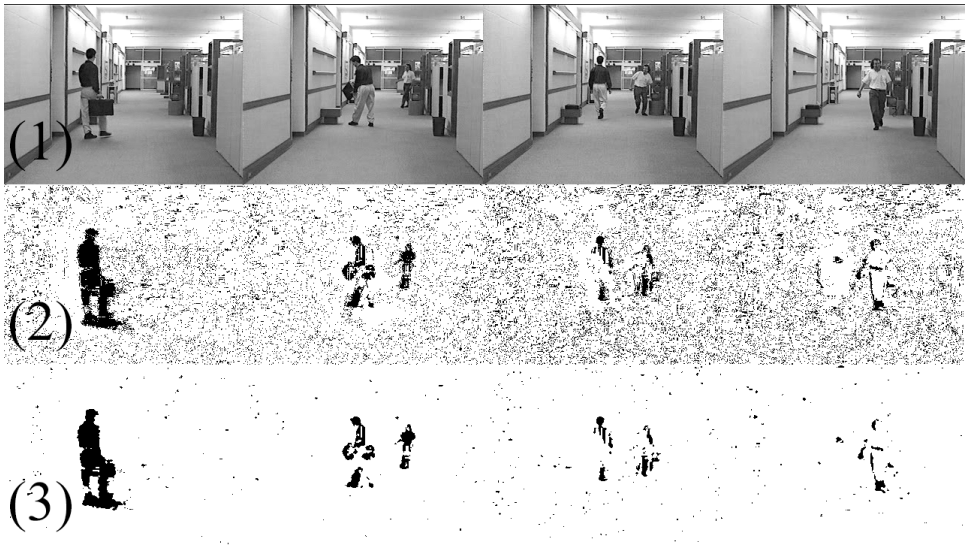



Figure A.1: Motion detection with Sigma Delta

## A.4 The Julia Mandelbrot Computation

The Mandelbrot set [34] defines a mathematical set of points that is very closed to the Julia sets. Its boundary results in a two-dimensional fractal shape. The fractal is obtained by sampling complex numbers and determining for each of them if the iterative application of a mathematical operation tends towards infinity.

$$\begin{cases} x_0 = y_0 = 0 \\ x_{n+1} = x_n^2 - y_n^2 + a \\ y_{n+1} = 2x_n y_n + b \end{cases}$$

Figure A.2: Definition of the Julia-Mandelbrot set

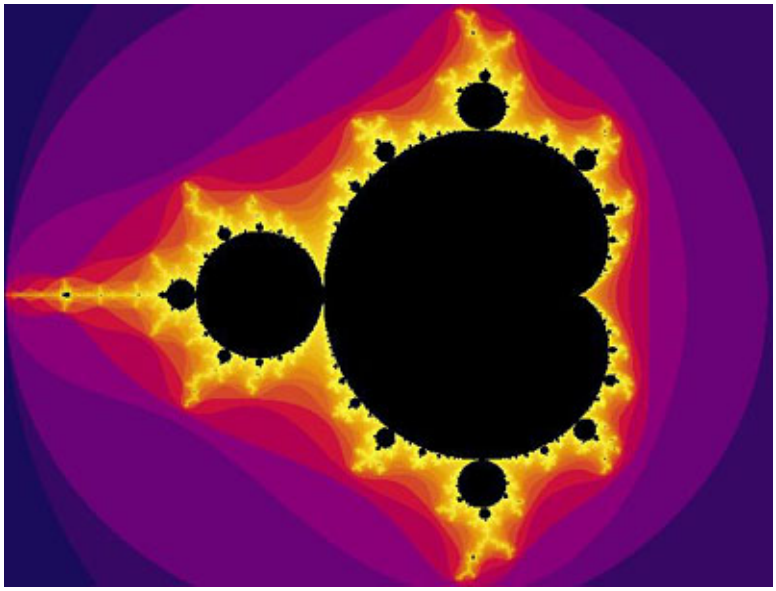


Figure A.3: Illustration of the Julia-Mandelbrot set

Appendix B

# Architectures Description

## B.1 Boost.SIMD Benchmark Architectures

Table B.1: Processor details

| Architecture | Nehalem | Sandy Bridge | PowerPC G5 | Haswell |
|---|---|---|---|---|
| Max. Frequency | 3.6 GHz | 3.8 GHz | 1.6 GHz | 3.4 GHz |
| SIMD Extension | SSE4.2 | AVX | Altivec | SSEX to AVX 2.0 |

## B.2 NT$^2$ Benchmark Architectures

- **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.

Appendix C

# ISO C++ Standard Proposal

This appendix shows the retained specification for SIMD support in the C++ Standard. The full proposal N3571 is available in [43]

### C.0.1 Our proposal

For maximum accessibility, programmers should be able to vectorize their code without needing a high level of expertise for every single SIMD extension. This proposal introduces a high-level abstraction to the user that gives access to SIMD computation in an instinctive way. It comes as a C++ template library, headers only that relies on a possibly full library implementation. With a high level template type for abstracting a SIMD register, the user can easily introduce SIMD in his application by instantiating this type and applying high level functions on it. Working at the register level rather than the loop nest or big array level keeps the abstraction thin, efficient and flexible. By keeping the control of the vectorization process, the programmer is explicitly expressing the SIMD version of his algorithm, not only guaranteeing that vectorization does indeed take place, but also empowering the user to define his algorithm in a way that is vectorizable. A single generic code can be written for both the scalar and SIMD types or different code paths may be selected. The library is also modular and easily extensible by the user.

In addition to types and functions operating on them, higher-order functions to manipulate and transform data with respect to every hardware constraints are provided.

Furthermore, processing multiple data in SIMD registers breaks typical scalar dataflows when dealing with branching conditions or when shifting or shuffling values. As a result, special functions to deal with SIMD-specific idioms are also introduced.

The idea of this proposal is inspired from the Boost.SIMD open-source library (not part of the Boost C++ libraries as of this writing) developed by the authors of this paper. This library has been deployed in several academic and industrial projects where it has shown significant advantages over other approaches to optimize code for SIMD-enabled processors. Boost.SIMD is available as part of the $NT^2$ software project hosted on GitHub [1]. Publications with experimental results are available in [45] and [44].

## C.1 Impact On the Standard

This proposal comes as a library extension that does not impact existing standard classes, functions or headers. This addition is non-intrusive; its implementation is fully standards-based and does not require any changes to the core language.

### C.1.1 Standard Components

#### C.1.1.1 SIMD Allocator

```
namespace std { namespace simd
{
  template<class T>
  struct allocator
  {
    typedef T               value_type;
    typedef T*              pointer;
    typedef T const*        const_pointer;
    typedef T&              reference;
    typedef T const&        const_reference;
    typedef size_t          size_type;
    typedef ptrdiff_t       difference_type;

    template<class U>
    struct rebind
    {
      typedef allocator<U> other;
    };

    allocator();

    template<class U>
    allocator(allocator<U> const&);

    pointer       address(reference r);
    const_pointer address(const_reference r);

    size_type max_size() const;

    void construct(pointer p, const T& t);
    void destroy(pointer p);

    pointer allocate( size_type c, const void* = 0 );
    void deallocate (pointer p, size_type s);
  };

  template<class T>
  bool operator==(allocator<T> const&, allocator<T> const&);

  template<class T>
  bool operator!=(allocator<T> const&, allocator<T> const&);

  template<class Allocator>
  struct allocator_adaptor
  {
    typedef Allocator base_type;

    typedef typename base_type::value_type       value_type;
    typedef typename base_type::pointer          pointer;
```

```
50      typedef typename base_type::const_pointer    const_pointer;
51      typedef typename base_type::reference        reference;
52      typedef typename base_type::const_reference  const_reference;
53      typedef typename base_type::size_type        size_type;
54      typedef typename base_type::difference_type  difference_type;
55
56      template<class U>
57      struct rebind
58      {
59        typedef allocator_adaptor<typename Allocator::rebind<U>::other> other;
60      };
61
62      allocator_adaptor();
63      allocator_adaptor(Allocator const& alloc);
64      ~allocator_adaptor();
65
66      template<class U>
67      allocator_adaptor(allocator_adaptor<U> const& src);
68
69      base_type&        base();
70      base_type const&  base() const;
71
72      pointer allocate( size_type c, const void* = 0 );
73      void deallocate (pointer p, size_type s);
74    };
75
76    template<class T>
77    bool operator==(allocator_adaptor<T> const& a, allocator_adaptor<T> const&
          b);
78
79    template<class T>
80    bool operator!=(allocator_adaptor<T> const& a, allocator_adaptor<T> const&
          b);
81  } }
```

### C.1.1.2 SIMD Algorithms

```
1   namespace std { namespace simd
2   {
3     template<class T, class U, class UnOp>
4     U* transform(T const* begin, T const* end, U* out, UnOp f);
5
6     template<class T1, class T2, class U, class BinOp>
7     U* transform(T1 const* begin1, T1 const* end, T2 const* begin2, U* out,
          BinOp f);
8
9     template<class T, class U, class F>
10    U accumulate(T const* begin, T const* end, U init, F f);
11  } }
```

```
1   template<class T, class U, class UnOp>
2   U* transform(T const* begin, T const* end, U* out, UnOp f);
```

*Requires:* `UnOp` is `Callable<U(T)>` and `Callable<pack<U>(pack<T>)>`

*Effects:* Writes to out the result of the application of `f` for each element in the range [`begin`, `end`[, by either loading scalar or SIMD values from the range if sufficient

aligned data is available.

*Returns:* The iterator past the last written-to position

```
template<class T1, class T2, class U, class BinOp>
U* transform(T1 const* begin1, T1 const* end, T2 const* begin2, U* out, BinOp
     f);
```

*Requires:* `BinOp` is `Callable<U(T1, T2)>` and `Callable<pack<U>(pack<T1>, pack<T2>)>`

*Effects:* Writes to out the result of the application of `f` for each pair of elements in the ranges [`begin1`, `end`[ and [`begin2`, `begin2+(end-begin1)`[, by either loading scalar or SIMD values from the ranges if sufficient aligned data is available.

*Returns:* The iterator past the last written-to position

```
template<class T, class U, class F>
U accumulate(T const* begin, T const* end, U init, F f);
```

*Requires:* `F` is `Callable<U(U, T)>` and `Callable<pack<U>(pack<U>, pack<T>)>`

*Effects:* Accumulate the result of the application of `f` with the accumulation state and each element of the range [`begin`, `end`[, potentially scalar by scalar of SIMD vector by vector if sufficient aligned data is available.

*Returns:* The final accumulation state

**Non-normative Note**: possible implementation of binary transform

```
template<class T1, class T2, class U, class BinOp>
U* transform(T1 const* begin1, T1 const* end, T2 const* begin2, U* out, BinOp
     f)
{
  // vectorization step based on ideal for output type
  typedef pack<U> vU;
  static const size_t N = vU::static_size;

  typedef pack<T1, N> vT1;
  typedef pack<T2, N> vT2;

  std::size_t align = N*sizeof(U);
  std::size_t shift = reinterpret_cast<U*>((reinterpret_cast<uinptr_t>(out)+
      align-1) & ~(align-1)) - out;
  T1 const* end2 = begin1 + shift;
  T1 const* end3 = end2 + (end - end2)/N*N;

  // prologue until 'out' aligned
  for(; begin1!=end2; ++begin1, ++begin2, ++out)
    *out = f(*begin1, *begin2);

  // vectorized body while more than N elements
  for(; begin1!=end3; begin1 += N, begin2 += N, out += N)
    simd::store(f(simd::unaligned_load<vT1>(begin1), simd::unaligned_load<vT2
        >(begin2)), out);
```

```
23
24   // epilogue for remaining elements
25   for(; begin1!=end; ++begin1, ++begin2, ++out)
26     *out = f(*begin1, *begin2);
27
28   return out;
29 }
```

## C.2 Technical Specifications

Here, we present the public interface required for the proposal.

### C.2.1 pack<T,N> class

```
1  namespace std { namespace simd
2  {
3    template<class T, std::size_t N = $\textit{unspecified}$ >
4    struct alignas(sizeof(T)*N) pack
5    {
6      typedef T                    value_type;
7      typedef value_type&          reference;
8      typedef value_type const&    const_reference;
9      typedef T*                   iterator;
10     typedef T const*             const_iterator;
11
12     static const size_t static_size = N;
13
14     // does not initialize values of pack
15     pack();
16
17     // copy constructor
18     pack(pack const& p);
19
20     // splat t N times into pack
21     pack(T t);
22
23     // fill pack with values from init
24     template<class T>
25     pack(initializer_list<T> init);
26
27     reference         operator[](std::size_t i);
28     const_reference   operator[](std::size_t i) const;
29     iterator          begin();
30     const_iterator    begin() const;
31     iterator          end();
32     const_iterator    end() const;
33     std::size_t       size() const;
34     bool              empty() const;
35   };
36 } }
```

### C.2.2 `logical<T>` class

Listing C.1: The `logical` structure

```
template<typename T>
struct logical;
```

`logical` is a marker for packs of boolean results and cannot be used in scalar mode.

### C.2.3 Operators overload for `pack<T,N>`

#### C.2.3.1 Unary Operators

```
namespace std { namespace simd
{
  template<class T, std::size_t N>
  pack<T,N> operator+(pack<T,N> p);

  template<class T, std::size_t N>
  pack<T,N> operator-(pack<T,N> p);

  template<class T, std::size_t N>
  typename as\_logical< pack<T,N> >::type operator!(pack<T,N> p);

  template<class T, std::size_t N>
  pack<T,N> operator~(pack<T,N> p);
} }
```

```
template<class T, std::size_t N>
pack<T,N> operator+(pack<T,N> p);
```

*Requires:* `T` is not a `logical` type.

*Effects:* Apply unary `operator+` on every element of `p`

*Returns:* A `pack<T,N>` value `r` so that $\forall i \in [0, N[$, `r[i] = + p[i]`

```
template<class T, std::size_t N>
pack<T,N> operator-(pack<T,N> p);
```

*Requires:* `T` is not a `logical` type.

*Effects:* Apply unary `operator-` on every element of `p`

*Returns:* A `pack<T,N>` value `r` so that $\forall i \in [0, N[$, `r[i] = - p[i]`

```
template<class T, std::size_t N>
typename as_logical< pack<T,N> >::type operator!(pack<T,N> p);
```

*Effects:* Apply unary `operator!` on every element of `p`

*Returns:* A `as_logical< pack<T,N> >::type` value `r` so that $\forall i \in [0, N[$, `r[i] = ! p[i]`

```
template<class T, std::size_t N>
pack<T,N> operator~(pack<T,N> p);
```

*Effects:* Apply unary `operator~` to every element of `p`

*Returns:* A `pack<T,N>` value `r` so that :

- If `T` is an integral type: $\forall i \in [0, N[$, `r[i] = ~ p[i]`;
- If `T` is a floating point type, the operation is performed on $r[i]$ bit pattern;
- If `T` is a `logical` type: $\forall i \in [0, N[$, `r[i] = !p[i]`.

### C.2.3.2 Binary Operators

```
namespace std { namespace simd
{
  template<class T, std::size_t N>
  pack<T,N> operator+(pack<T,N> p,pack<T,N> q);
  template<class T, class U, std::size_t N>
  pack<T,N> operator+(pack<T,N> p, U q);
  template<class T, class U, std::size_t N>
  pack<T,N> operator+(U p, pack<T,N> q);

  template<class T, std::size_t N>
  pack<T,N> operator-(pack<T,N> p,pack<T,N> q);
  template<class T, class U, std::size_t N>
  pack<T,N> operator-(pack<T,N> p, U q);
  template<class T, class U, std::size_t N>
  pack<T,N> operator-(U p, pack<T,N> q);

  template<class T, std::size_t N>
  pack<T,N> operator*(pack<T,N> p,pack<T,N> q);
  template<class T, class U, std::size_t N>
  pack<T,N> operator*(pack<T,N> p, U q);
  template<class T, class U, std::size_t N>
  pack<T,N> operator*(U p, pack<T,N> q);

  template<class T, std::size_t N>
  pack<T,N> operator/(pack<T,N> p,pack<T,N> q);
  template<class T, class U, std::size_t N>
  pack<T,N> operator/(pack<T,N> p, U q);
  template<class T, class U, std::size_t N>
  pack<T,N> operator/(U p, pack<T,N> q);

  template<class T, std::size_t N>
  pack<T,N> operator%(pack<T,N> p,pack<T,N> q);
  template<class T, class U, std::size_t N>
  pack<T,N> operator%(pack<T,N> p, U q);
  template<class T, class U, std::size_t N>
  pack<T,N> operator%(U p, pack<T,N> q);


  template<class T, std::size_t N>
  pack<T,N> operator&(pack<T,N> p,pack<T,N> q);
  template<class T, class U, std::size_t N>
  pack<T,N> operator&(pack<T,N> p, U q);
  template<class T, class U, std::size_t N>
```

```
44   pack<T,N> operator&(U p, pack<T,N> q);
45
46   template<class T, std::size_t N>
47   pack<T,N> operator|(pack<T,N> p,pack<T,N> q);
48   template<class T, class U, std::size_t N>
49   pack<T,N> operator|(pack<T,N> p, U q);
50   template<class T, class U, std::size_t N>
51   pack<T,N> operator|(U p, pack<T,N> q);
52
53   template<class T, std::size_t N>
54   pack<T,N> operator^(pack<T,N> p,pack<T,N> q);
55   template<class T, class U, std::size_t N>
56   pack<T,N> operator^(pack<T,N> p, U q);
57   template<class T, class U, std::size_t N>
58   pack<T,N> operator^(U p, pack<T,N> q);
59 } }
```

```
template<class T, std::size_t N>
pack<T,N> operator+(pack<T,N> p, pack<T,N> q);
```

*Requires:* `T` is not a `logical` type.

*Effects:* Apply binary `operator+` between every element of `p` and `q`

*Returns:* A `pack<T,N>` value `r` so that $\forall i \in [0, N[$, `r[i] = p[i] + q[i]`

```
template<class T, class U, std::size_t N>
pack<T,N> operator+(pack<T,N> p, U q);
```

*Requires:* `T` is not a `logical` type.

*Effects:* Apply binary `operator+` between every element of `p` and `q`

*Returns:* A `pack<T,N>` value `r` so that $\forall i \in [0, N[$, `r[i] = p[i] + static_cast<T>(q)`

```
template<class T, class U, std::size_t N>
pack<T,N> operator+(U p, pack<T,N> q);
```

*Requires:* `T` is not a `logical` type.

*Effects:* Apply binary `operator+` between `p` and every element of `q`

*Returns:* A `pack<T,N>` value `r` so that $\forall i \in [0, N[$, `r[i] = static_cast<T>(p) + q[i]`

```
template<class T, std::size_t N>
pack<T,N> operator-(pack<T,N> p, pack<T,N> q);
```

*Requires:* `T` is not a `logical` type.

*Effects:* Apply binary `operator-` between every element of `p` and `q`

*Returns:* A `pack<T,N>` value `r` so that $\forall i \in [0, N[$, `r[i] = p[i] - q[i]`

```
template<class T, class U, std::size_t N>
```

```
pack<T,N> operator-(pack<T,N> p, U q);
```

*Requires:* `T` is not a `logical` type.

*Effects:* Apply binary `operator-` between every element of `p` and `q`

*Returns:* A `pack<T,N>` value `r` so that $\forall i \in [0, N[$, `r[i] = p[i] - static_cast<T>(q)`

```
template<class T, class U, std::size_t N>
pack<T,N> operator-(U p, pack<T,N> q);
```

*Requires:* `T` is not a `logical` type.

*Effects:* Apply binary `operator-` between `p` and every element of `q`

*Returns:* A `pack<T,N>` value `r` so that $\forall i \in [0, N[$, `r[i] = static_cast<T>(p) - q[i]`

```
template<class T, std::size_t N>
pack<T,N> operator*(pack<T,N> p, pack<T,N> q);
```

*Requires:* `T` is not a `logical` type.

*Effects:* Apply binary `operator*` between every element of `p` and `q`

*Returns:* A `pack<T,N>` value `r` so that $\forall i \in [0, N[$, `r[i] = p[i] * q[i]`

```
template<class T, class U, std::size_t N>
pack<T,N> operator*(pack<T,N> p, U q);
```

*Requires:* `T` is not a `logical` type.

*Effects:* Apply binary `operator*` between every element of `p` and `q`

*Returns:* A `pack<T,N>` value `r` so that $\forall i \in [0, N[$, `r[i] = p[i] * static_cast<T>(q)`

```
template<class T, class U, std::size_t N>
pack<T,N> operator*(U p, pack<T,N> q);
```

*Requires:* `T` is not a `logical` type.

*Effects:* Apply binary `operator*` between `p` and every element of `q`

*Returns:* A `pack<T,N>` value `r` so that $\forall i \in [0, N[$, `r[i] = static_cast<T>(p) * q[i]`

```
template<class T, std::size_t N>
pack<T,N> operator/(pack<T,N> p, pack<T,N> q);
```

*Requires:* `T` is not a `logical` type.

*Effects:* Apply binary `operator/` between every element of `p` and `q`

*Returns:* A `pack<T,N>` value `r` so that $\forall i \in [0, N[$, `r[i] = p[i] / q[i]`

```
template<class T, class U, std::size_t N>
pack<T,N> operator/(pack<T,N> p, U q);
```

*Requires:* `T` is not a `logical` type.

*Effects:* Apply binary `operator/` between every element of `p` and `q`

*Returns:* A `pack<T,N>` value `r` so that $\forall i \in [0, N[$, `r[i] = p[i] / static_cast<T>(q)`

```
template<class T, class U, std::size_t N>
pack<T,N> operator/(U p, pack<T,N> q);
```

*Requires:* `T` is not a `logical` type.

*Effects:* Apply binary `operator/` between `p` and every element of `q`

*Returns:* A `pack<T,N>` value `r` so that $\forall i \in [0, N[$, `r[i] = static_cast<T>(p) / q[i]`

```
template<class T, std::size_t N>
pack<T,N> operator%(pack<T,N> p, pack<T,N> q);
```

*Requires:* `T` is not a `logical` type.

*Effects:* Apply binary `operator%` between every element of `p` and `q`

*Returns:* A `pack<T,N>` value `r` so that $\forall i \in [0, N[$, `r[i] = p[i] % q[i]`

```
template<class T, class U, std::size_t N>
pack<T,N> operator%(pack<T,N> p, U q);
```

*Requires:* `T` is not a `logical` type.

*Effects:* Apply binary `operator%` between every element of `p` and `q`

*Returns:* A `pack<T,N>` value `r` so that $\forall i \in [0, N[$, `r[i] = p[i] % static_cast<T>(q)`

```
template<class T, class U, std::size_t N>
pack<T,N> operator%(U p, pack<T,N> q);
```

*Requires:* `T` is not a `logical` type.

*Effects:* Apply binary `operator%` between `p` and every element of `q`

*Returns:* A `pack<T,N>` value `r` so that $\forall i \in [0, N[$, `r[i] = static_cast<T>(p) % q[i]`

```
template<class T, std::size_t N>
```

```
pack<T,N> operator&(pack<T,N> p, pack<T,N> q);
```

*Requires:* `T` is not a `logical` type.

*Effects:* Apply binary `operator&` between every element of `p` and `q`

*Returns:* A `pack<T,N>` value `r` so that $\forall i \in [0, N[$, `r[i] = p[i] & q[i]`

```
template<class T, class U, std::size_t N>
pack<T,N> operator&(pack<T,N> p, U q);
```

*Requires:* `T` is not a `logical` type.

*Effects:* Apply binary `operator&` between every element of `p` and `q`

*Returns:* A `pack<T,N>` value `r` so that $\forall i \in [0, N[$, `r[i] = p[i] & static_cast<T>(q)`

```
template<class T, class U, std::size_t N>
pack<T,N> operator&(U p, pack<T,N> q);
```

*Requires:* `T` is not a `logical` type.

*Effects:* Apply binary `operator&` between `p` and every element of `q`

*Returns:* A `pack<T,N>` value `r` so that $\forall i \in [0, N[$, `r[i] = static_cast<T>(p) & q[i]`

```
template<class T, std::size_t N>
pack<T,N> operator|(pack<T,N> p, pack<T,N> q);
```

*Requires:* `T` is not a `logical` type.

*Effects:* Apply binary `operator|` between every element of `p` and `q`

*Returns:* A `pack<T,N>` value `r` so that $\forall i \in [0, N[$, `r[i] = p[i] | q[i]`

```
template<class T, class U, std::size_t N>
pack<T,N> operator|(pack<T,N> p, U q);
```

*Requires:* `T` is not a `logical` type.

*Effects:* Apply binary `operator|` between every element of `p` and `q`

*Returns:* A `pack<T,N>` value `r` so that $\forall i \in [0, N[$, `r[i] = p[i] | static_cast<T>(q)`

```
template<class T, class U, std::size_t N>
pack<T,N> operator|(U p, pack<T,N> q);
```

*Requires:* `T` is not a `logical` type.

*Effects:* Apply binary `operator|` between `p` and every element of `q`

*Returns:* A `pack<T,N>` value `r` so that $\forall i \in [0, N[$, `r[i] = static_cast<T>(p) | q[i]`

```
template<class T, std::size_t N>
pack<T,N> operator^(pack<T,N> p, pack<T,N> q);
```

*Requires:* `T` is not a `logical` type.

*Effects:* Apply binary `operator^` between every element of `p` and `q`

*Returns:* A `pack<T,N>` value `r` so that $\forall i \in [0, N[$, `r[i] = p[i] ^ q[i]`

```
template<class T, class U, std::size_t N>
pack<T,N> operator^(pack<T,N> p, U q);
```

*Requires:* `T` is not a `logical` type.

*Effects:* Apply binary `operator^` between every element of `p` and `q`

*Returns:* A `pack<T,N>` value `r` so that $\forall i \in [0, N[$, `r[i] = p[i] ^ static_cast<T>(q)`

```
template<class T, class U, std::size_t N>
pack<T,N> operator^(U p, pack<T,N> q);
```

*Requires:* `T` is not a `logical` type.

*Effects:* Apply binary `operator^` between `p` and every element of `q`

*Returns:* A `pack<T,N>` value `r` so that $\forall i \in [0, N[$, `r[i] = static_cast<T>(p) ^ q[i]`

### C.2.3.3 Logical Operators

```
namespace std { namespace simd
{
  template<class T, std::size_t N>
  pack<T,N> operator<(pack<T,N> p,pack<T,N> q);
  template<class T, class U, std::size_t N>
  pack<T,N> operator<(pack<T,N> p, U q);
  template<class T, class U, std::size_t N>
  pack<T,N> operator<(U p, pack<T,N> q);

  template<class T, std::size_t N>
  pack<T,N> operator>(pack<T,N> p,pack<T,N> q);
  template<class T, class U, std::size_t N>
  pack<T,N> operator>(pack<T,N> p, U q);
  template<class T, class U, std::size_t N>
  pack<T,N> operator>(U p, pack<T,N> q);

  template<class T, std::size_t N>
  pack<T,N> operator<=(pack<T,N> p,pack<T,N> q);
  template<class T, class U, std::size_t N>
  pack<T,N> operator<=(pack<T,N> p, U q);
```

```
21    template<class T, class U, std::size_t N>
22    pack<T,N> operator<=(U p, pack<T,N> q);
23
24    template<class T, std::size_t N>
25    pack<T,N> operator>=(pack<T,N> p,pack<T,N> q);
26    template<class T, class U, std::size_t N>
27    pack<T,N> operator>=(pack<T,N> p, U q);
28    template<class T, class U, std::size_t N>
29    pack<T,N> operator>=(U p, pack<T,N> q);
30
31    template<class T, std::size_t N>
32    pack<T,N> operator==(pack<T,N> p,pack<T,N> q);
33    template<class T, class U, std::size_t N>
34    pack<T,N> operator==(pack<T,N> p, U q);
35    template<class T, class U, std::size_t N>
36    pack<T,N> operator==(U p, pack<T,N> q);
37
38    template<class T, std::size_t N>
39    pack<T,N> operator!=(pack<T,N> p,pack<T,N> q);
40    template<class T, class U, std::size_t N>
41    pack<T,N> operator!=(pack<T,N> p, U q);
42    template<class T, class U, std::size_t N>
43    pack<T,N> operator!=(U p, pack<T,N> q);
44
45    template<class T, std::size_t N>
46    pack<T,N> operator&&(pack<T,N> p,pack<T,N> q);
47    template<class T, class U, std::size_t N>
48    pack<T,N> operator&&(pack<T,N> p, U q);
49    template<class T, class U, std::size_t N>
50    pack<T,N> operator&&(U p, pack<T,N> q);
51
52    template<class T, std::size_t N>
53    pack<T,N> operator||(pack<T,N> p,pack<T,N> q);
54    template<class T, class U, std::size_t N>
55    pack<T,N> operator||(pack<T,N> p, U q);
56    template<class T, class U, std::size_t N>
57    pack<T,N> operator||(U p, pack<T,N> q);
58
59
60  } }
```

```
template<class T, std::size_t N>
as_logical<pack<T,N» operator<(pack<T,N> p, pack<T,N> q);
```

*Effects:* Apply binary `operator<` between every element of `p` and `q`

*Returns:* A logical value `r` so that $\forall i \in [0, N[$, `r[i] = p[i] < q[i]`

```
template<class T, class U, std::size_t N>
as_logical<pack<T,N» operator<(pack<T,N> p, U q);
```

*Effects:* Apply binary `operator<` between every element of `p` and `q`

*Returns:* A logical value `r` so that $\forall i \in [0, N[$, `r[i] = p[i] < static_cast<T>(q)`

```
template<class T, class U, std::size_t N>
as_logical<pack<T,N» operator<(U p, pack<T,N> q);
```

*Effects:* Apply binary `operator<` between `p` and every element of `q`

*Returns:* A logical value `r` so that $\forall i \in [0, N[$, `r[i] = static_cast<T>(p) < q[i]`

```
template<class T, std::size_t N>
as_logical<pack<T,N»> operator>(pack<T,N> p, pack<T,N> q);
```

*Effects:* Apply binary `operator>` between every element of `p` and `q`

*Returns:* A logical value `r` so that $\forall i \in [0, N[$, `r[i] = p[i] > q[i]`

```
template<class T, class U, std::size_t N>
as_logical<pack<T,N»> operator>(pack<T,N> p, U q);
```

*Effects:* Apply binary `operator>` between every element of `p` and `q`

*Returns:* A logical value `r` so that $\forall i \in [0, N[$, `r[i] = p[i] > static_cast<T>(q)`

```
template<class T, class U, std::size_t N>
as_logical<pack<T,N»> operator>(U p, pack<T,N> q);
```

*Effects:* Apply binary `operator>` between `p` and every element of `q`

*Returns:* A logical value `r` so that $\forall i \in [0, N[$, `r[i] = static_cast<T>(p) > q[i]`

```
template<class T, std::size_t N>
as_logical<pack<T,N»> operator<=(pack<T,N> p, pack<T,N> q);
```

*Effects:* Apply binary `operator<=` between every element of `p` and `q`

*Returns:* A logical value `r` so that $\forall i \in [0, N[$, `r[i] = p[i] <= q[i]`

```
template<class T, class U, std::size_t N>
as_logical<pack<T,N»> operator<=(pack<T,N> p, U q);
```

*Effects:* Apply binary `operator<=` between every element of `p` and `q`

*Returns:* A logical value `r` so that $\forall i \in [0, N[$, `r[i] = p[i] <= static_cast<T>(q)`

```
template<class T, class U, std::size_t N>
as_logical<pack<T,N»> operator<=(U p, pack<T,N> q);
```

*Effects:* Apply binary `operator<=` between `p` and every element of `q`

*Returns:* A logical value `r` so that $\forall i \in [0, N[$, `r[i] = static_cast<T>(p) <= q[i]`

```
template<class T, std::size_t N>
as_logical<pack<T,N»> operator>=(pack<T,N> p, pack<T,N> q);
```

*Effects:* Apply binary `operator>=` between every element of `p` and `q`

*Returns:* A logical value `r` so that $\forall i \in [0, N[$, `r[i] = p[i] >= q[i]`

```
template<class T, class U, std::size_t N>
as_logical<pack<T,N» operator>=(pack<T,N> p, U q);
```

*Effects:* Apply binary `operator>=` between every element of `p` and `q`

*Returns:* A logical value `r` so that $\forall i \in [0, N[$, `r[i] = p[i] >= static_cast<T>(q)`

```
template<class T, class U, std::size_t N>
as_logical<pack<T,N» operator>=(U p, pack<T,N> q);
```

*Effects:* Apply binary `operator>=` between `p` and every element of `q`

*Returns:* A logical value `r` so that $\forall i \in [0, N[$, `r[i] = static_cast<T>(p) >= q[i]`

```
template<class T, std::size_t N>
as_logical<pack<T,N» operator==(pack<T,N> p, pack<T,N> q);
```

*Effects:* Apply binary `operator==` between every element of `p` and `q`

*Returns:* A logical value `r` so that $\forall i \in [0, N[$, `r[i] = p[i] == q[i]`

```
template<class T, class U, std::size_t N>
as_logical<pack<T,N» operator==(pack<T,N> p, U q);
```

*Effects:* Apply binary `operator==` between every element of `p` and `q`

*Returns:* A logical value `r` so that $\forall i \in [0, N[$, `r[i] = p[i] == static_cast<T>(q)`

```
template<class T, class U, std::size_t N>
as_logical<pack<T,N» operator==(U p, pack<T,N> q);
```

*Effects:* Apply binary `operator==` between `p` and every element of `q`

*Returns:* A logical value `r` so that $\forall i \in [0, N[$, `r[i] = static_cast<T>(p) == q[i]`

```
template<class T, std::size_t N>
as_logical<pack<T,N» operator!=(pack<T,N> p, pack<T,N> q);
```

*Effects:* Apply binary `operator!=` between every element of `p` and `q`

*Returns:* A logical value `r` so that $\forall i \in [0, N[$, `r[i] = p[i] != q[i]`

```
template<class T, class U, std::size_t N>
as_logical<pack<T,N» operator!=(pack<T,N> p, U q);
```

*Effects:* Apply binary `operator!=` between every element of `p` and `q`

*Returns:* A logical value `r` so that $\forall i \in [0, N[$, `r[i] = p[i] != static_cast<T>(q)`

```
template<class T, class U, std::size_t N>
as_logical<pack<T,N» operator!=(U p, pack<T,N> q);
```

*Effects:* Apply binary `operator!=` between `p` and every element of `q`

*Returns:* A logical value `r` so that $\forall i \in [0, N[$, `r[i] = static_cast<T>(p) != q[i]`

```
template<class T, std::size_t N>
as_logical<pack<T,N» operator&&(pack<T,N> p, pack<T,N> q);
```

*Effects:* Apply binary `operator&&` between every element of `p` and `q`

*Returns:* A logical value `r` so that $\forall i \in [0, N[$, `r[i] = p[i] && q[i]`

```
template<class T, class U, std::size_t N>
as_logical<pack<T,N» operator&&(pack<T,N> p, U q);
```

*Effects:* Apply binary `operator&&` between every element of `p` and `q`

*Returns:* A logical value `r` so that $\forall i \in [0, N[$, `r[i] = p[i] && static_cast<T>(q)`

```
template<class T, class U, std::size_t N>
as_logical<pack<T,N» operator&&(U p, pack<T,N> q);
```

*Effects:* Apply binary `operator&&` between `p` and every element of `q`

*Returns:* A logical value `r` so that $\forall i \in [0, N[$, `r[i] = static_cast<T>(p) && q[i]`

```
template<class T, std::size_t N>
as_logical<pack<T,N» operator||(pack<T,N> p, pack<T,N> q);
```

*Effects:* Apply binary `operator||` between every element of `p` and `q`

*Returns:* A logical value `r` so that $\forall i \in [0, N[$, `r[i] = p[i] || q[i]`

```
template<class T, class U, std::size_t N>
as_logical<pack<T,N» operator||(pack<T,N> p, U q);
```

*Effects:* Apply binary `operator||` between every element of `p` and `q`

*Returns:* A logical value `r` so that $\forall i \in [0, N[$, `r[i] = p[i] || static_cast<T>(q)`

```
template<class T, class U, std::size_t N>
as_logical<pack<T,N» operator||(U p, pack<T,N> q);
```

*Effects:* Apply binary `operator||` between `p` and every element of `q`

*Returns:* A logical value `r` so that $\forall i \in [0, N[$, `r[i] = static_cast<T>(p) || q[i]`

### C.2.3.4 Ternary Operators

```
namespace std { namespace simd
{
  template<class T, class U, std::size_t N>
  pack<T,N> if_else(pack<U,N> c,pack<T,N> t,pack<T,N> f);

  template<class T, class U, std::size_t N>
  pack<T,N> if_else(U c,pack<T,N> t,pack<T,N> f);

  template<class T, class U, std::size_t N>
  pack<T,N> if_else(pack<U,N> c, T t,pack<T,N> f);

  template<class T, class U, std::size_t N>
  pack<T,N> if_else(pack<U,N> c, pack<T,N> t,T f);
} }
```

```
template<class T, class U, std::size_t N>
pack<T,N> if_else(pack<U,N> c,pack<T,N> t,pack<T,N> f);
```

*Effects:* Apply ternary `operator?:` between every element of `c`, `t` and `f`

*Returns:* A `pack<T,N>` value `r` so that $\forall i \in [0, N[$, `r[i] = c[i] ? t[i] : f[i]`

```
template<class T, class U, std::size_t N>
pack<T,N> if_else(U c,pack<T,N> t,pack<T,N> f);
```

*Effects:* Apply ternary `operator?:` between `c` and every element of `t` and `f`

*Returns:* A `pack<T,N>` value `r` so that $\forall i \in [0, N[$, `r[i] = c ? t[i] : f[i]`

```
template<class T, class U, std::size_t N>
pack<T,N> if_else(pack<U,N> c,T t,pack<T,N> f);
```

*Effects:* Apply ternary `operator?:` between `t` and every element of `c` and `f`

*Returns:* A `pack<T,N>` value `r` so that $\forall i \in [0, N[$, `r[i] = c[i] ? t : f[i]`

```
template<class T, class U, std::size_t N>
pack<T,N> if_else(pack<U,N> c,pack<T,N>  t,T f);
```

*Effects:* Apply ternary `operator?:` between `f` and every element of `c` and `t`

*Returns:* A `pack<T,N>` value `r` so that $\forall i \in [0, N[$, `r[i] = c[i] ? t[i] : f`

**Non-Normative Note** If `c` is a `logical` type or a `pack` of `logical` type, implementation of `if_else` can be optimized by not requiring the conversion of `c` to an actual SIMD bitmask.

### C.2.4 Functions

#### C.2.4.1 Memory related Functions

```
namespace std { namespace simd
{
  // replicate a scalar value in a pack
  template<class T, class U>
  T splat(U v);

  // convert a pack
  template<class T, std::size_t N, class U>
  T splat(pack<U, N> v);

  // aligned load
  template<class T, class U>
  T load(U* p);

  template<class T, class U>
  T load(U* p, std::ptrdiff_t o);

  // aligned gather
  template<class T, class U, class V, std::size_t N>
  T load(U* p, pack<V, N> o);

  // load with static misalignment
  template<class T, std::ptrdiff_t A, class U>
  T load(U* p);

  template<class T, std::ptrdiff_t A, class U>
  T load(U* p, std::ptrdiff_t o);

  // gather with static misalignment
  template<class T, std::ptrdiff_t A, class U, class V, std::size_t N>
  T load(U* p, pack<V, N> o);

  // unaligned load
  template<class T, class U>
  T unaligned_load(U* p);

  template<class T, class U>
  T unaligned_load(U* p, std::ptrdiff_t o);

  // gather
  template<class T, class U, class V, std::size_t N>
  T unaligned_load(U* p, pack<V, N> o);

  // aligned store
  template<class T, class U>
  void store(T v, U* p);

  template<class T, class U>
  void store(T v, U* p, std::ptrdiff_t o);

  // aligned scatter
  template<class T, class U, class V, std::size_t N>
```

```
53   void store(pack<T, N> v, U* p, pack<V, N> o);
54
55   // unaligned store
56   template<class T, class U>
57   void unaligned_store(T v, U* p);
58
59   template<class T, class U>
60   void unaligned_store(T v, U* p, std::ptrdiff_t o);
61
62   // scatter
63   template<class T, class U, class V, std::size_t N>
64   void unaligned_store(pack<T, N> v, U* p, pack<V, N> o);
65 } }
```

```
1 template<class T, class U>
2 T splat(U v);
```

*Effects:* Convert the value `v` to the type `T`, replicate the value if `T` is a pack.

*Returns:* If `T` is `pack<T2,N>`, return a value `r` so that $\forall i \in [0, N[$, `r[i] = static_cast<T2>(v)`.
else `r = static_cast<T>(v)`.

```
1 template<class T, class U, std::size_t N>
2 T splat(pack<U, N> v);
```

*Requires:* `T` is `pack<T2,N>`.

*Effects:* Convert each element of `v` from `U` to `T2`.

*Returns:* Return a value `r` so that $\forall i \in [0, N[$, `r[i] = static_cast<T2>(v[i])`.

*Note:* While the cardinal of the two packs is the same, the size of the element and therefore the register type being used may change arbitrarily between the input and output of this function.

```
1 template<class T, class U>
2 T load(U* p);
```

*Requires:* `U` is not a pack type, `p` is aligned on a boundary suitable for loading objects of type `T`.

*Effects:* Load an object of type `T` from aligned memory, possibly after doing a type conversion.

*Returns:* If `T` is `pack<T2,N>`, return a value `r` so that $\forall i \in [0, N[$, `r[i] = static_cast<T2>(p[i])`.
else `r = static_cast<T>(*p)`.

```
template<class T, class U>
T load(U* p, std::ptrdiff_t o);
```

*Requires:* `U` is not a pack type, `p+o` is aligned on a boundary suitable for loading objects of type `T`.

*Effects:* Load an object of type `T` from aligned memory, possibly after doing a type conversion.

*Returns:* If `T` is `pack<T2,N>`, return a value `r` so that $\forall i \in [0, N[$, `r[i] = static_cast<T2>(p[o+i])`.
else `r = static_cast<T>(p[o])`.

```
template<class T, class U, class V, std::size_t N>
T load(U* p, pack<V, N> o);
```

*Requires:* `U` is not a pack type, `T` is `pack<T2,N>` and all of of `p+o[i]` are aligned on a boundary suitable for loading objects of type `T`.

*Effects:* Load an object of type `T` from aligned indexed memory, possibly after doing a type conversion.

*Returns:* Return a value `r` so that $\forall i \in [0, N[$, `r[i] = static_cast<T2>(p[o[i]])`.

*Note:* This is usually known as a *gather* operation.

```
template<class T, std::ptrdiff_t A, class U>
T load(U* p);
```

*Requires:* `U` is not a pack type, `p-A` is aligned on a boundary suitable for loading objects of type `T`.

*Effects:* Load an object of type `T` from memory whose misalignment is $A$, possibly after doing a type conversion.

*Returns:* If `T` is `pack<T2,N>`, return a value `r` so that $\forall i \in [0, N[$, `r[i] = static_cast<T2>(p[i])`.
else `r = static_cast<T>(*p)`.

```
template<class T, std::ptrdiff_t A, class U>
T load(U* p, std::ptrdiff_t o);
```

*Requires:* `U` is not a pack type, `p+o-A` is aligned on a boundary suitable for loading objects of type `T`.

*Effects:* Load an object of type `T` from memory whose misalignment is $A$, possibly after doing a type conversion.

*Returns:* If `T` is `pack<T2,N>`, return a value `r` so that $\forall i \in [0, N[$, `r[i] = static_cast<T2>(p[o+i])`.
else `r = static_cast<T>(p[o])`.

```
template<class T, std::ptrdiff_t A, class U, class V, std::size_t N>
T load(U* p, pack<V, N> o);
```

*Requires:* `U` is not a pack type, `T` is `pack<T2,N>` and all of of `p+o[i]-A` are aligned on a boundary suitable for loading objects of type `T`.

*Effects:* Load an object of type `T` from indexed memory whose misalignment is $A$, possibly after doing a type conversion.

*Returns:* Return a value `r` so that $\forall i \in [0, N[$, `r[i] = static_cast<T2>(p[o[i]])`.

*Note:* This is usually known as a *gather* operation.

```
template<class T, class U>
T unaligned_load(U* p);
```

*Requires:* `U` is not a pack type.

*Effects:* Load an object of type `T` from memory, possibly after doing a type conversion.

*Returns:* If `T` is `pack<T2,N>`, return a value `r` so that $\forall i \in [0, N[$, `r[i] = static_cast<T2>(p[i])`.
else `r = static_cast<T>(*p)`.

```
template<class T, class U>
T unaligned_load(U* p, std::ptrdiff_t o);
```

*Requires:* `U` is not a pack type.

*Effects:* Load an object of type `T` from memory, possibly after doing a type conversion.

*Returns:* If `T` is `pack<T2,N>`, return a value `r` so that $\forall i \in [0, N[$, `r[i] = static_cast<T2>(p[o+i])`.
else `r = static_cast<T>(p[o])`.

```
template<class T, class U, class V, std::size_t N>
T unaligned_load(U* p, pack<V, N> o);
```

*Requires:* `U` is not a pack type, `T` is `pack<T2,N>`.

*Effects:* Load an object of type `T` from indexed memory, possibly after doing a type conversion.

*Returns:* Return a value `r` so that $\forall i \in [0, N[$, `r[i] = static_cast<T2>(p[o[i]])`.

*Note:* This is usually known as a *gather* operation.

```
template<class T, class U>
void store(T v, U* p);
```

*Requires:* `U` is not a pack type, `p` is aligned on a boundary suitable for storing objects of type `T`.

*Effects:* Store the object `v` to memory to aligned memory, possibly after doing a type conversion.
If `T` is `pack<T2,N>`, $\forall i \in [0, N[$, `p[i] = static_cast<T2>(v[i])`.
else `*p = static_cast<T>(v)`.

```
template<class T, class U>
void store(T v, U* p, std::ptrdiff_t o);
```

*Requires:* `U` is not a pack type, `p+o` is aligned on a boundary suitable for storing objects of type `T`.

*Effects:* Store the object `v` to memory to aligned memory, possibly after doing a type conversion.
If `T` is `pack<T2,N>`, $\forall i \in [0, N[$, `p[o+i] = static_cast<T2>(v[i])`.
else `p[o] = static_cast<T>(v)`.

```
template<class T, class U, class V, std::size_t N>
void store(pack<T, N> v, U* p, pack<V, N> o);
```

*Requires:* `U` is not a pack type and all of `p+o[i]` are aligned on a boundary suitable for storing objects of type `T`.

*Effects:* Store the object `v` to aligned indexed memory, possibly after doing a type conversion.
Return a value `r` so that $\forall i \in [0, N[$, `p[o[i]] = static_cast<T2>(v[i])`.

*Note:* This is usually known as a *scatter* operation.

```
template<class T, class U>
void unaligned_store(T v, U* p);
```

*Requires:* `U` is not a pack type.

*Effects:* Store the object `v` to memory, possibly after doing a type conversion.
If `T` is `pack<T2,N>`, $\forall i \in [0, N[$, `p[i] = static_cast<T2>(v[i])`.
else `*p = static_cast<T>(v)`.

```
template<class T, class U>
void unaligned_store(T v, U* p, std::ptrdiff_t o);
```

*Requires:* `U` is not a pack type.
*Effects:* Store the object `v` to memory, possibly after doing a type conversion.
If `T` is `pack<T2,N>`, $\forall i \in [0, N[$, `p[o+i] = static_cast<T2>(v[i])`.
else `p[o] = static_cast<T>(v)`.

```
template<class T, class U, class V, std::size_t N>
void unaligned_store(pack<T, N> v, U* p, pack<V, N> o);
```

*Requires:* `U` is not a pack type and all of `p+o[i]` are aligned on a boundary suitable for storing objects of type `T`.

*Effects:* Store the object `v` to indexed memory, possibly after doing a type conversion.
Return a value `r` so that $\forall i \in [0, N[$, `p[o[i]] = static_cast<T2>(v[i])`.

*Note:* This is usually known as a *scatter* operation.

### C.2.4.2 Shuffling Functions

```
namespace std { namespace simd
{
  template<class F, class T, std::size_t N>
  pack<T,N> shuffle(pack<T,N> p);
} }
```

*Requires:* `F` is a metafunction class.

*Effects:* fills the elements of the destination `pack<T,N> r` with the elements of `p` respecting the following expression : `r[i] = p[F::template apply<i,N>::value]` $\forall i \in [0, N[$.

*Returns:* The resulting `pack<T,N>`.

```
namespace std { namespace simd
{
  template<std::ptrdiff_t... I, class T, std::size_t N>
  pack<T,N> shuffle(pack<T,N> p);
} }
```

*Requires:* `sizeof...(I)` is equal to `N` and `I` belongs to $[0, N[$.

*Effects:* fills the elements of the destination `pack<T,N> r` with the elements of `p` respecting the following expression : `r[i] = p[F::template apply<i,N>::value]` $\forall i \in [0, N[$.

*Returns:* The resulting `pack<T,N>`.

```
namespace std { namespace simd
{
  template<class F, class T, std::size_t N>
  pack<T,N> shuffle(pack<T,N> p1, pack<T,N> p2);
} }
```

*Requires:* `F` is a metafunction class.

*Effects:* fills the elements of the destination `pack<T,N> r` with the elements of `p` respecting the following expression : `r[i] = (F::template apply<i,N>::value<N) ? p[F::template apply<i,N>::value] : p[F::template apply<i,N>::value - N]` $\forall i \in [0, N[$.

*Returns:* The resulting `pack<T,N>`.

```
namespace std { namespace simd
{
  template<std::ptrdiff_t... I, class T, std::size_t N>
  pack<T,N> shuffle(pack<T,N> p1, pack<T,N> p2);
} }
```

*Requires:* `sizeof...(I)` is equal to `N` and `I` belongs to $[0, N[$.

*Effects:* fills the elements of the destination `pack<T,N> r` with the elements of `p` respecting the following expression : `r[i]=(F::template apply<i,N>::value<N)?p[F::template apply<i,N>::value]:p[F::template apply<i,N>::value-N]` $\forall i \in [0, N[$.

*Returns:* The resulting `pack<T,N>`.

### C.2.4.3 Reduction Functions

```
namespace std { namespace simd
{
  template<class T               > T sum(T p);
  template<class T, std::size_t N> T sum(pack<T,N> p);
  template<class T               > T prod(T p);
  template<class T, std::size_t N> T prod(pack<T,N> p);
  template<class T               > T min(T p);
  template<class T, std::size_t N> T min(pack<T,N> p);
  template<class T               > T max(T p);
  template<class T, std::size_t N> T max(pack<T,N> p);

```

```
12   template<class T                > bool all(T p);
13   template<class T, std::size_t N> bool all(pack<T,N> p);
14   template<class T                > bool any(T p);
15   template<class T, std::size_t N> bool any(pack<T,N> p);
16   template<class T                > bool none(T p);
17   template<class T, std::size_t N> bool none(pack<T,N> p);
18 } }
```

#### C.2.4.4 `cmath` Functions

The function supported includes all of the mathematical functions available in the `cmath` header C.1.

Table C.1: Functions on `pack`

| Generic Name | Description |
|---|---|
| abs | computes the absolute value |
| div | the quotient and remainder of integer division |
| fmod | remainder of the floating point division operation |
| remainder | signed remainder of the division operation |
| fma | fused multiply-add operation |
| max | larger of two values |
| min | smaller of two values |
| dim | positive difference of two floating point values |
| nan | not-a-number |
| exp | returns e raised to the given power |
| exp2 | returns 2 raised to the given power |
| expm1 | returns e raised to the given power, minus one |
| log | computes natural (base e) logarithm (to base e) |
| log10 | computes common (base 10) logarithm |
| log1p | natural logarithm (to base e) of 1 plus the given number |
| log2p | base 2 logarithm of the given number |
| sqrt | computes square root |
| cbrt | computes cubic root |
| hypot | computes square root of the sum of the squares of two given numbers |
| pow | raises a number to the given power |
| sin and variants | computes sine (arc sine, hyperbolic sine) |
| cos and variants | computes cosine (arc cosine, hyperbolic cosine) |
| tan and variants | computes tangent (arc tangent, hyperbolic tangent) |
| erf | error function |
| erfc | complementary error function |
| lgamma | natural logarithm of the gamma function |
| tgamma | gamma function |
| ceil | nearest integer not less than the given value |
| floor | nearest integer not greater than the given value |
| trunc | nearest integer not greater in magnitude than the given value |
| round | nearest integer, rounding away from zero in halfway cases |
| nearbyint | nearest integer using current rounding mode |
| rint | nearest integer using current rounding mode with exception if the result differs |
| frexp | decomposes a number into significand and a power of 2 |
| ldexp | multiplies a number by 2 raised to a power |
| modf | decomposes a number into integer and fractional parts |
| logb | extracts exponent of the number |
| nextafter/nexttoward | next representable floating point value towards the given value |

Table C.1: Functions on `pack`

| Generic Name | Description |
|---|---|
| copysign | copies the sign of a value |
| isfinite | checks if the given number has finite value |
| isinf | checks if the given number is infinite |
| isnan | checks if the given number is NaN |
| isnormal | checks if the given number is normal |
| signbit | checks if the given number is negative |
| isgreater | checks if the first floating-point argument is greater than the second |
| isgreaterequal | checks if the first floating-point argument is greater or equal than the second |
| isless | checks if the first floating-point argument is less than the second |
| islessequal | checks if the first floating-point argument is less or equal than the second |
| islessgreater | checks if the first floating-point argument is less or greater than the second |
| isunordered | checks if two floating-point values are unordered |

### C.2.5 Traits and metafunctions

```
namespace std { namespace simd
{
  template<class T> struct scalar_of;

  template<class T> struct cardinal_of;

  template<class T> struct as_logical;
} }
```

```
template<class T> struct scalar_of;
```

*Returns:* If `T` is a cv or reference qualified `pack<T2, N>` type, return `T2` with the same cv and reference qualifiers. Otherwise return `T`.

```
template<class T> struct cardinal_of;
```

*Returns:* If `T` is a cv or reference qualified `pack<T2, N>` type, return `integral_constant<size_t, N>`. Otherwise return `integral_constant<size_t, 1>`.

```
template<class T> struct as_logical;
```

*Returns:* If `T` is a cv or reference qualified `pack<T2, N>` type with `T2` a non-logical type, return `pack<logical<T2>, N>` with the same cv and reference qualifiers. Else if `T` is a cv or reference qualified non-logical type `T2`, return `logical<T2>` with the same cv and reference qualifiers. Otherwise return `T`.

Appendix D

# Meta-Unroller

Listing D.1 presents an implementation of a meta-unroller based on a Duff's devices optimization.

Listing D.1: A Meta-Unroller based on a Duff's devices optimization

```
template<int N>
struct unroll
{};

template<>
struct unroll<0>
{
  template<class SimdInputIterator, class SimdOutputIterator, class UnOp>
  inline static void
  apply( SimdInputIterator& in, SimdInputIterator const& end
       , SimdOutputIterator& out, UnOp f)
  {
    while(in != end)
    *out++ = f(*in++);
  }

  template<class SimdInputIterator, class SimdOutputIterator, class BinOp>
  inline static void
  apply( SimdInputIterator& in1, SimdInputIterator& in2
       , SimdInputIterator const& end, SimdOutputIterator& out, BinOp f)
  {
    while(in1 != end)
    *out++ = f(*in1++,*in2++);
  }

};

template<>
struct unroll<2>
{
  template<class SimdInputIterator, class SimdOutputIterator, class UnOp>
  inline static void
  apply( SimdInputIterator& in, SimdInputIterator const& end
       , SimdOutputIterator& out, UnOp f)
  {
    typename SimdInputIterator::difference_type distance = end - in;
    typename SimdInputIterator::difference_type n = (distance + 1) / 2;

    switch(distance % 2)
    {
      case 0 : do{
        *out++ = f(*in++);
      case 1 : *out++ = f(*in++);
      } while(--n > 0);
    }
  }
```

```
47
48    template<class SimdInputIterator, class SimdOutputIterator, class BinOp>
49    inline static void
50    apply( SimdInputIterator& in1, SimdInputIterator& in2
51         , SimdInputIterator const& end, SimdOutputIterator& out, BinOp f)
52    {
53      typename SimdInputIterator::difference_type distance = end - in1;
54      typename SimdInputIterator::difference_type n = (distance + 1) / 2;
55
56      switch(distance % 2)
57      {
58        case 0 : do{
59          *out++ = f(*in1++,*in2++);
60        case 1 : *out++ = f(*in1++,*in2++);
61        } while(--n > 0);
62      }
63    }
64  };
65
66  template<>
67  struct unroll<4>
68  {
69    template<class SimdInputIterator, class SimdOutputIterator, class UnOp>
70    inline static void
71    apply( SimdInputIterator& in, SimdInputIterator const& end
72         , SimdOutputIterator& out, UnOp f)
73    {
74      typename SimdInputIterator::difference_type distance = end - in;
75      typename SimdInputIterator::difference_type n = (distance + 3) / 4;
76
77      switch(distance % 4)
78      {
79        case 0 : do{
80          *out++ = f(*in++);
81        case 3 : *out++ = f(*in++);
82        case 2 : *out++ = f(*in++);
83        case 1 : *out++ = f(*in++);
84        } while(--n > 0);
85      }
86    }
87
88    template<class SimdInputIterator, class SimdOutputIterator, class BinOp>
89    inline static void
90    apply( SimdInputIterator& in1, SimdInputIterator& in2
91         , SimdInputIterator const& end, SimdOutputIterator& out, BinOp f)
92    {
93      typename SimdInputIterator::difference_type distance = end - in1;
94      typename SimdInputIterator::difference_type n = (distance + 3) / 4;
95
96      switch(distance % 4)
97      {
98        case 0 : do{
99          *out++ = f(*in1++,*in2++);
100       case 3 : *out++ = f(*in1++,*in2++);
101       case 2 : *out++ = f(*in1++,*in2++);
102       case 1 : *out++ = f(*in1++,*in2++);
103       } while(--n > 0);
104     }
105   }
106
107 };
```

# Bibliography


[1] The numerical template toolbox. http://github.com/MetaScale. (Cited on page 129.)

[2] ustl. (Cited on page 14.)

[3] David Abrahams and Aleksey Gurtovoy. *C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond (C++ in Depth Series)*. Addison-Wesley Professional, 2004. (Cited on pages 24 and 30.)

[4] Emmanuel Agullo, Jim Demmel, Jack Dongarra, Bilel Hadri, Jakub Kurzak, Julien Langou, Hatem Ltaief, Piotr Luszczek, and Stanimire Tomov. Numerical linear algebra on emerging architectures: The plasma and magma projects. In *Journal of Physics: Conference Series*, volume 180, page 012037. IOP Publishing, 2009. (Cited on page 21.)

[5] Marco Aldinucci, Marco Danelutto, and Patrizio Dazzi. Muskel: an expandable skeleton environment. *Scalable Computing: Practice and Experience*, 8(4), 2001. (Cited on page 108.)

[6] Marco Aldinucci, Marco Danelutto, and Jan Dünnweber. Optimization techniques for implementing parallel skeletons in grid environments. In Sergei Gorlatch, editor, *Proc. of CMPP: Intl. Workshop on Constructive Methods for Parallel Programming*, pages 35–47, Stirling, Scotland, UK, July 2004. Universitat Munster, Germany. (Cited on page 109.)

[7] AMD. Amd core math library. http://developer.amd.com/libraries/acml/pages/default.aspx. (Cited on pages 14 and 21.)

[8] Ping An, Alin Jula, Silvius Rus, Steven Saunders, Tim Smith, Gabriel Tanase, Nathan Thomas, Nancy Amato, and Lawrence Rauchwerger. Stapl: An adaptive, generic parallel c++ library. In *Languages and Compilers for Parallel Computing*, pages 193–208. Springer, 2003. (Cited on page 20.)

[9] Edward Anderson, Zhaojun Bai, Christian Bischof, Susan Blackford, James Demmel, Jack Dongarra, Jeremy Du Croz, Anne Greenbaum, S Hammerling, Alan McKenney, et al. *LAPACK Users' guide*, volume 9. Siam, 1999. (Cited on page 21.)

[10] APPLE. Accelerate framework reference. https://developer.apple.com/library/mac/documentation/Accelerate/Reference/AccelerateFWRef/_index.html, 2014. (Cited on page 21.)

[11] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André 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 28.)

[12] Cédric Bastoul and Paul Feautrier. Adjusting a program transformation for legality. *Parallel processing letters*, 15(01n02):3–17, 2005. (Cited on page 121.)

[13] Blaze. The blaze library. https://code.google.com/p/blaze-lib/, 2014. (Cited on page 24.)

[14] OpenMP Architecture Review Board. Openmp 3.0 specifications. http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf, 2008. (Cited on page 16.)

[15] OpenMP Architecture Review Board. Openmp 4.0 specifications. http://www.openmp.org/mp-documents/spec30.pdf, 2013. (Cited on page 16.)

[16] Gary Bradski and Adrian Kaehler. *Learning OpenCV: Computer vision with the OpenCV library*. O'Reilly Media, Inc., 2008. (Cited on page 22.)

[17] Walter Bright. Templates revisited. http://dlang.org/templates-revisited.html, 2014. (Cited on page 30.)

[18] Shirley Browne, Jack Dongarra, Eric Grosse, and Tom Rowan. The netlib mathematical software repository. *D-Lib Magazine, Sep*, 1995. (Cited on page 21.)

[19] Jaeyoung Choi, Jack J Dongarra, Roldan Pozo, and David W Walker. Scalapack: A scalable linear algebra library for distributed memory concurrent computers. In *Frontiers of Massively Parallel Computation, 1992., Fourth Symposium on the*, pages 120–127. IEEE, 1992. (Cited on page 21.)

[20] Philipp Ciechanowicz and Herbert 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*, pages 108–113. IEEE, 2010. (Cited on page 109.)

[21] R Clint Whaley, Antoine Petitet, and Jack J Dongarra. Automated empirical optimizations of software and the atlas project. *Parallel Computing*, 27(1):3–35, 2001. (Cited on pages 21 and 28.)

[22] Murray Cole. Bringing skeletons out of the closet: a pragmatic manifesto for skeletal parallel programming. *Parallel computing*, 30(3):389–406, 2004. (Cited on page 108.)

[23] Murray I Cole. *Algorithmic skeletons: structured management of parallel computation*. Pitman London, 1989. (Cited on page 108.)

[24] Krzysztof Czarnecki. Generative programming: Principles and techniques of software engineering based on automated configuration and fragment-based component models. 1998. (Cited on page 29.)

[25] Krzysztof Czarnecki and Ulrich W. Eisenecker. Components and generative programming. In *ESEC / SIGSOFT FSE*, pages 2–19, 1999. (Cited on page 29.)

[26] Krzysztof Czarnecki and Ulrich W Eisenecker. Generative programming. 2000. (Cited on page 29.)

[27] Krzysztof Czarnecki and Ulrich W. Eisenecker. *Generative programming - methods, tools and applications*. Addison-Wesley, 2000. (Cited on page 36.)

[28] Krzysztof Czarnecki, Ulrich W. Eisenecker, Robert Glück, David Vandevoorde, and Todd L. Veldhuizen. Generative programming and active libraries. In *Generic Programming*, pages 25–39, 1998. (Cited on pages 24 and 86.)

[29] Krzysztof Czarnecki, John T. O'Donnell, Jörg Striegnitz, and Walid Taha. Dsl implementation in metaocaml, template haskell, and c++. In *Domain-Specific Program Generation*, pages 51–72, 2003. (Cited on page 24.)

[30] Leonardo Dagum and Ramesh Menon. Openmp: an industry standard api for shared-memory programming. *Computational Science & Engineering, IEEE*, 5(1):46–55, 1998. (Cited on page 16.)

[31] Joel de Guzman, Dan Marsden, and Tobias Schwinger. Boost.fusion library. `http://www.boost.org/doc/libs/release/libs/fusion/doc/html`. (Cited on pages 64 and 69.)

[32] Chirag Dekate, Matthew Anderson, Maciej Brodowicz, Hartmut Kaiser, Bryce Adelstein-Lelbach, and Thomas Sterling. Improving the scalability of parallel n-body applications with an event-driven constraint-based execution model. *Int. J. High Perform. Comput. Appl.*, 26(3):319–332, 2012. (Cited on page 121.)

[33] John Derrick, Neil Walkinshaw, Thomas Arts, Clara Benac Earle, Francesco Cesarini, Lars-Åke Fredlund, Víctor M. Gulías, John Hughes, and Simon J. Thompson. Property-based testing - the protest project. In *FMCO*, pages 250–271, 2009. (Cited on page 23.)

[34] Robert L Devaney. *A first course in chaotic dynamical systems*. Westview Press, 1992. (Cited on page 125.)

[35] Keith Diefendorff, Pradeep K Dubey, Ron Hochsprung, and HASH Scale. Altivec extension to powerpc accelerates media processing. *Micro, IEEE*, 20(2):85–95, 2000. (Cited on page 8.)

[36] Jack J Dongarra, James R Bunch, Cleve B Moler, and Gilbert W Stewart. *LINPACK users' guide*, volume 8. Siam, 1979. (Cited on page 21.)

[37] Jack J Dongarra, Steve W Otto, Marc Snir, and David Walker. An introduction to the mpi standard. *Communications of the ACM*, 1995. (Cited on page 18.)

[38] Gabriel Dos Reis, Bjarne Stroustrup, and A Merideth. Axioms: Semantics aspects of c++ concepts. Technical report, Tech. Rep, 2009. (Cited on page 64.)

[39] Alejandro Duran and Michael Klemm. The intel® many integrated core architecture. In *High Performance Computing and Simulation (HPCS), 2012 International Conference on*, pages 365–366. IEEE, 2012. (Cited on page 62.)

[40] Jason Eckhardt, Roumen Kaiabachev, Emir Pasalic, Kedar Swadi, and Walid Taha. Implicitly heterogeneous multi-stage programming. *New Gen. Comput.*, 25(3):305–336, January 2007. (Cited on page 121.)

[41] Kento Emoto, Kiminori Matsuzaki, Zhenjiang Hu, and Masato Takeichi. Domain-specific optimization strategy for skeleton programs. In Anne-Marie Kermarrec, Luc Bougé, and Thierry Priol, editors, *Euro-Par 2007 Parallel Processing*, volume 4641 of *Lecture Notes in Computer Science*, pages 705–714. Springer Berlin Heidelberg, 2007. (Cited on page 109.)

[42] Pierre Esterie, Joel Falcou, Mathias Gaunard, Jean-Thierry Lapreste, and Lionel Lacassagne. The numerical template toolbox: A modern c++ design for scientific computing. *Journal of Parallel and Distributed Computing*, 2013. En cours. (Cited on page 120.)

[43] Pierre Esterie, Mathias Gaunard, and Joel Falcou. A proposal to add single instruction multiple data computation to the standard library. *ISO C++ Proposal Paper, N3571*, 2013. En cours. (Cited on page 129.)

[44] Pierre Esterie, Mathias Gaunard, Joel Falcou, et al. Exploiting multimedia extensions in c++: A portable approach. *Computing in Science & Engineering*, 14(5):72–77, 2012. (Cited on pages 62, 103, 120 and 129.)

[45] Pierre Estérie, Mathias Gaunard, Joel Falcou, Jean-Thierry Lapresté, and Brigitte 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 pages 62, 103, 120 and 129.)

[46] Imèn Fassi, Philippe Clauss, Matthieu Kuhn, Yosr Slama, et al. Multifor for multicore. In *IMPACT 2013, Third International Workshop on Polyhedral Compilation Techniques*, pages 37–44, 2013. (Cited on pages 17 and 121.)

[47] Michael Flynn. Some computer organizations and their effectiveness. *Computers, IEEE Transactions on*, 100(9):948–960, 1972. (Cited on page 2.)

[48] Martin Fowler. *Domain-specific languages*. Pearson Education, 2010. (Cited on page 23.)

[49] Thomas Fox, Michael Gschwind, and Jaime Moreno. Qpx architecture: Quad processing extension to the power isa. *Software: Practice and Experience*, 2011. (Cited on page 62.)

[50] Matteo Frigo and Steven G Johnson. Fftw: An adaptive software architecture for the fft. In *Acoustics, Speech and Signal Processing, 1998. Proceedings of the 1998 IEEE International Conference on*, volume 3, pages 1381–1384. IEEE, 1998. (Cited on page 22.)

[51] Edgar Gabriel, Graham E Fagg, George Bosilca, Thara Angskun, Jack J Dongarra, Jeffrey M Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, et al. Open mpi: Goals, concept, and design of a next generation mpi implementation. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 97–104. Springer, 2004. (Cited on page 18.)

[52] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Pearson Education, 1994. (Cited on page 29.)

[53] Kazushige Goto. Gotoblas. *Texas Advanced Computing Center, University of Texas at Austin, USA. URL< http://www. otc. utexas. edu/ATdisplay. jsp*, 2007. (Cited on page 21.)

[54] Peter Gottschling, David S. Wise, and Michael 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 25.)

[55] Douglas Gregor, Jaakko Järvi, Jeremy Siek, Bjarne Stroustrup, Gabriel Dos Reis, and Andrew Lumsdaine. Concepts: linguistic support for generic programming in c++. In *ACM SIGPLAN Notices*, volume 41, pages 291–310. ACM, 2006. (Cited on page 45.)

[56] Stellar Group. Hpx. `http://stellar.cct.lsu.edu/projects/hpx/`, 2014. (Cited on page 19.)

[57] Gaël Guennebaud, Benoît Jacob, et al. Eigen v3. http://eigen.tuxfamily.org, 2010. (Cited on pages 25 and 86.)

[58] Scott Haney and James Crotinger. Pete: The portable expression template engine. *Dr. Dobb's journal*, 24(10), 1999. (Cited on page 31.)

[59] Paul Hudak. Building domain-specific embedded languages. *ACM Comput. Surv.*, 28(4es), December 1996. (Cited on page 24.)

[60] Intel. Math kernel library. `http://developer.intel.com/software/products/mkl/`. (Cited on pages 14 and 21.)

[61] Minwoo Jang, Kukhyun Kim, and Kanghee Kim. The performance analysis of arm neon technology for mobile platforms. In *Proceedings of the 2011 ACM Symposium on Research in Applied Computation*, RACS '11, pages 104–106, New York, NY, USA, 2011. ACM. (Cited on page 62.)

[62] Jaakko Jarvi. Compile time recursive objects in c++. In *Technology of Object-Oriented Languages, 1998. TOOLS 27. Proceedings*, pages 66–77. IEEE, 1998. (Cited on page 30.)

[63] Eric E Johnson. Completing an mimd multiprocessor taxonomy. *ACM SIGARCH Computer Architecture News*, 16(3):44–47, 1988. (Cited on page 2.)

[64] Laxmikant V Kale and Sanjeev Krishnan. *CHARM++: a portable concurrent object oriented system based on C++*, volume 28-10. ACM, 1993. (Cited on page 121.)

[65] Nicholas T Karonis, Brian Toonen, and Ian Foster. Mpich-g2: A grid-enabled implementation of the message passing interface. *Journal of Parallel and Distributed Computing*, 63(5):551–563, 2003. (Cited on page 18.)

[66] Matthias Kretz and Volker Lindenstruth. Vc: A c++ library for explicit vectorization. *Software: Practice and Experience*, 2011. (Cited on page 86.)

[67] Matthias Kretz and Volker Lindenstruth. Vc: A c++ library for explicit vectorization. *Software: Practice and Experience*, 42(11):1409–1430, 2012. (Cited on page 14.)

[68] Herbert Kuchen. *A skeleton library*. Springer, 2002. (Cited on pages 108 and 109.)

[69] Jakub Kurzak, Wesley Alvaro, and Jack Dongarra. Optimizing matrix multiplication for a short-vector simd architecture : Cell processor. *Parallel Computing*, 35(3):138 – 150, 2009. (Cited on page 62.)

[70] Lionel Lacassagne, Antoine Manzanera, Julien Denoulet, and Alain Mérigot. High performance motion detection: some trends toward new embedded architectures for vision systems. *Journal of Real-Time Image Processing*, 4:127–146, 2009. (Cited on pages 95 and 123.)

[71] Chuck L Lawson, Richard J. Hanson, David R Kincaid, and Fred T. Krogh. Basic linear algebra subprograms for fortran usage. *ACM Transactions on Mathematical Software (TOMS)*, 5(3):308–323, 1979. (Cited on page 21.)

[72] Roland Leißa, Immanuel Haffner, and Sebastian Hack. Sierra: A simd extension for c+. (Cited on page 14.)

[73] David Luebke, Mark Harris, Naga Govindaraju, Aaron Lefohn, Mike Houston, John Owens, Mark Segal, Matthew Papakipos, and Ian Buck. Gpgpu:

general-purpose computation on graphics hardware. In *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 208. ACM, 2006. (Cited on page 10.)

[74] MathWorks. The matlab language. `http://www.mathworks.fr/products/matlab/`, 2014. (Cited on page 23.)

[75] Robert C Merton. Option pricing when underlying stock returns are discontinuous. *Journal of financial economics*, 3(1):125–144, 1976. (Cited on page 123.)

[76] Eric Niebler. Proto : A compiler construction toolkit for DSELs. In *Proceedings of ACM SIGPLAN Symposium on Library-Centric Software Design*, 2007. (Cited on pages 32 and 85.)

[77] Dorit Nuzman, Sergei Dyshel, Erven Rohou, Ira Rosen, Kevin Williams, David Yuste, Albert Cohen, and Ayal 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 14.)

[78] Dorit Nuzman and Richard 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 14.)

[79] NVIDIA. cublas. `https://developer.nvidia.com/cublas`, 2014. (Cited on page 21.)

[80] Alex Peleg and Uri Weiser. Mmx technology extension to the intel architecture. *Micro, IEEE*, 16(4):42–50, 1996. (Cited on page 8.)

[81] Matt Pharr and William R Mark. ispc: A spmd compiler for high-performance cpu programming. In *Innovative Parallel Computing (InPar), 2012*, pages 1–13. IEEE, 2012. (Cited on page 14.)

[82] Louis-Noël Pouchet, Uday Bondhugula, Cédric Bastoul, Albert Cohen, J Ramanujam, and P Sadayappan. Combined iterative and model-driven optimization in an automatic parallelization framework. In *High Performance Computing, Networking, Storage and Analysis (SC), 2010 International Conference for*, pages 1–11. IEEE, 2010. (Cited on page 28.)

[83] James Reinders. *Intel threading building blocks - outfitting C++ for multi-core processor parallelism*. O'Reilly, 2007. (Cited on page 16.)

[84] John VW Reynders, Paul J Hinker, Julian C Cummings, Susan R Atlas, Subhankar Banerjee, William F Humphrey, Steve R Karmesin, Katarzyna Keahey, Marikani Srikant, and Mary Dell Tholburn. Pooma: A framework for scientific simulations on parallel architectures. *Parallel Programming in C+*, pages 547–588, 1996. (Cited on page 31.)

[85] Arch D Robison. Composable parallel patterns with intel cilk plus. *Computing in Science & Engineering*, 15(2):0066–71, 2013. (Cited on page 14.)

[86] Tarik Saidani, Lionel Lacassagne, Joel Falcou, Claude Tadonki, and Samir Bouaziz. Parallelization schemes for memory optimization on the cell processor: a case study on the harris corner detector. In Per Stenström, editor, *Transactions on high-performance embedded architectures and compilers III*, pages 177–200. Springer-Verlag, Berlin, Heidelberg, 2011. (Cited on page 75.)

[87] Conrad Sanderson et al. Armadillo: An open source c++ linear algebra library for fast prototyping and computationally intensive experiments. *Report Version*, 2, 2010. (Cited on page 24.)

[88] Scala. Scala documentation. `http://docs.scala-lang.org/overviews/reflection/symbols-trees-types.html`, 2014. (Cited on page 31.)

[89] Jocelyn Serot and Joel Falcou. Functional meta-programming for parallel skeletons. In *Computational Science–ICCS 2008*, pages 154–163. Springer, 2008. (Cited on page 30.)

[90] Tim Sheard and Simon Peyton Jones. Template meta-programming for haskell. In *Proceedings of the 2002 ACM SIGPLAN workshop on Haskell*, pages 1–16. ACM, 2002. (Cited on page 30.)

[91] Nathan Slingerland and Alan Jay Smith. Performance analysis of instruction set architecture extensions for multimedia. In *the 3rd Workshop on Media and Stream Processors*, pages 204–217, 2001. (Cited on page 8.)

[92] Diomidis Spinellis. Notable design patterns for domain-specific languages. *Journal of Systems and Software*, 56(1):91 – 99, 2001. (Cited on pages 30 and 86.)

[93] C++ Standard. The callable concept. `http://en.cppreference.com/w/cpp/concept/Callable`, 2014. (Cited on page 34.)

[94] Alexander Stepanov and Meng Lee. The standard template library. Technical Report 95-11(R.1), HP Laboratories, 1995. (Cited on page 74.)

[95] Andrew Sutton and Bjarne Stroustrup. Design of concept libraries for c++. In *Software Language Engineering*, pages 97–118. Springer, 2012. (Cited on page 45.)

[96] Stewart Taylor. *Optimizing Applications for Multi-Core Processors, Using the Intel Integrated Performance Primitives*. Intel Press, 2007. (Cited on page 22.)

[97] FLAME Team. Flame project. `http://www.cs.utexas.edu/~flame/web/index.html`, 2014. (Cited on page 21.)

[98] RDevelopment Core Team et al. R: A language and environment for statistical computing. *R foundation for Statistical Computing*, 2005. (Cited on page 23.)

[99] Laurence Tratt. Model transformations and tool integration. *Journal of Software and Systems Modelling*, 4(2):112–122, May 2005. (Cited on page 24.)

[100] Spyridon Triantafyllis, Manish Vachharajani, Neil Vachharajani, and David 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 28.)

[101] Arie Van Deursen, Paul Klint, and Joost Visser. Domain-specific languages: An annotated bibliography. *Sigplan Notices*, 35(6):26–36, 2000. (Cited on page 23.)

[102] Field G Van Zee, Ernie Chan, Robert A Van de Geijn, Enrique S Quintana-Ort, and Gregorio Quintana-Ort. The libflame library for dense matrix computations. *Computing in science & engineering*, 11(6):56–63, 2009. (Cited on page 21.)

[103] Field G. Van Zee and Robert A. van de Geijn. FLAME Working Note #66. BLIS: A framework for generating blas-like libraries. Technical Report TR-12-30, The University of Texas at Austin, Department of Computer Sciences, November 2012. (Cited on page 21.)

[104] David Vandevoorde and Nicolai M. Josuttis. *C++ Templates*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002. (Cited on page 30.)

[105] Todd L. Veldhuizen. Expression templates. *C++ Report*, 7(5):26–31, June 1995. Reprinted in C++ Gems, ed. Stanley Lippman. (Cited on page 30.)

[106] Todd L Veldhuizen. Blitz++: The library that thinks it is a compiler. In *Advances in Software tools for scientific computing*, pages 57–87. Springer, 2000. (Cited on page 24.)

[107] Todd L. Veldhuizen and Dennis 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. (Cited on pages 24 and 30.)

[108] Z Xianyi, W Qian, and Z Chothia. Openblas, version 0.2. 8. *URL http://www. openblas. net/. Fe tched*, pages 09–13, 2013. (Cited on page 21.)

**Multi-Architectural Support: A Generic and Generative Approach**

**Abstract:** The constant increasing need for computing power has pushed the development of parallel architectures. Scientific computing relies on the performance of such architectures to produce scientific results. Programming efficient applications that takes advantage of these computing systems remains a non trivial task.

In this thesis, we present a new methodology to design architecture aware software: the *AA-DEMRAL* methodology. This methodology aims at simplifying the development of parallel programming tools with multi-architectural support through a generic and generative approach.

We then present three high level programming tools that rely on this approach. First, we introduce the Boost.Dispatch library that provides a way to develop software based on the *AA-DEMRAL* methodology. The Boost.Dispatch library is a C++ generic framework for architecture aware function dispatching. Then, we present two C++ template libraries implemented as Architecture Aware *DSEL*s which assess the *AA-DEMRAL* methodology through the use of Boost.Dispatch: Boost.SIMD, that provides a high level API for SIMD extensions and $NT^2$, which propose a Matlab like interface with support for multi-core and SIMD based systems. We assess the performance of these libraries and the validity of our new methodology through benchmarks.

**Keywords:** Parallel architectures, *DSEL*s , Active Library, Generative Programming, Generic Programming, C++ .