MONT-BLANC

D5.5– BOAST: a Metaprogramming Framework to Produce Portable and Efficient Computing Kernels for HPC Applications
Version 1.0

Document Information

Contract Number	610402
Project Website	www.montblanc-project.eu
Contractual Deadline	M18
Dissemintation Level	PU
Nature	Report
Coordinator	CNRS
Contributors	Florent Bouchez Tichadou (UJF), Thierry Deutsch (CEA), Luigi Genovese (CEA), Jean-François Méhaut (UJF), Kevin Pouget (CNRS), Brice Videau (CNRS)
Reviewers	Olivier Aumage (INRIA), Frédéric Desprez (INRIA), Harald Servat (BSC)
Keywords	BOAST, Deliverable

Notices: *The research leading to these results has received funding from the European Community's Seventh Framework Programme [FP7/2007-2013] under grant agreement n° 288777 and 610402*

©Mont-Blanc 2 Consortium Partners. All rights reserved.

Change Log

Version	Description of Change
v0.1	Initial version of the deliverable
v0.2	First version sent to reviewers
v0.3	With corrections from reviewers
v0.4	Additional corrections from reviewers
v1.0	Version to be sent to the EU

Contents

Executive Summary **4**

1 Introduction **5**

2 Background and Motivation **5**
2.1 Scientific Computing Applications . . . 5
2.2 How Should Computing Kernels be Written? . . . 5
2.3 Evolution of HPC Architectures . . . 6

3 Motivating Example: OpenCL Laplace **6**
3.1 The Laplace Filter . . . 7
3.2 Possible Optimizations of the OpenCL Version . . . 7
3.3 Improving the Methodology . . . 9

4 BOAST: Using Code Generation in Application Autotuning **9**
4.1 Kernel Description Language . . . 11
4.1.1 BOAST Keywords . . . 11
4.1.2 BOAST Abstractions . . . 12
4.2 BOAST Run-time . . . 14
4.2.1 Multi-target Language Generation . . . 14
4.2.2 Compilation . . . 15
4.2.3 Execution . . . 15
4.2.4 Kernel Replay . . . 15
4.3 Non Regression Testing Using Trace Debugging . . . 15

5 BOAST Use Cases **17**
5.1 Laplace Filter Kernel . . . 17
5.1.1 Optimization Space . . . 17
5.1.2 Performance Results . . . 18
5.1.3 Laplace Conclusion . . . 18
5.2 Creating an Auto-Tuned Convolution Library for BigDFT using BOAST . . . 19
5.2.1 BigDFT . . . 19
5.2.2 A Generic Convolution Library . . . 20
5.2.3 Performance report . . . 21
5.3 Porting SPECFEM3D to OpenCL using BOAST . . . 21
5.3.1 SPECFEM3D . . . 22
5.3.2 Porting to OpenCL . . . 22

6 Related Work **24**
6.1 Application Auto-Tuning . . . 24
6.2 Kernel Description DSL . . . 24
6.3 Optimization Space Pruner . . . 25

7 Conclusion and Future Works **25**

Executive Summary

This document describes BOAST, a metaprogramming framework to produce portable and efficient computing kernels for HPC application. BOAST offers an embedded domain specific language to describe the kernels and their possible optimization. BOAST also supplies a complete run-time to compile, run, benchmark, and check the validity of the generated kernels. BOAST is being used in two flagship HPC applications BigDFT and SPECFEM3D, to improve performance portability of those codes.

1 Introduction

Porting and tuning HPC applications to new platforms is tedious and costly in terms of human resources. Nonetheless, it is a very important aspect of the Mont-Blanc project. Indeed, for the project, more than ten applications were selected to be ported and optimized for the prototype platform.

Unfortunately, portability efforts are often lost when migrating to a new architecture. Worse, code may lose maintainability because several versions of some functionalities coexist, usually with a lot of duplication.

Thus productivity of porting and tuning efforts is low as a huge fraction of those developments are never used after the platform they were intended for is decommissioned.

Genericity of HPC codes is often limited. One of the reason is that producing generic code in Fortran 90/95 is difficult as the language does not really fit for it. Sometimes, adding genericity degrades performance as optimization opportunities that come from over-specification are lost.

Functionality of HPC codes is tied to the previous point. Without genericity, adding new functionalities can be quite costly.

2 Background and Motivation

2.1 Scientific Computing Applications

Scientific Computing Applications are usually developed by physicist, chemist or meteorologist. Those codes are usually written in Fortran for historical and performance reasons. Codes can be quite huge (several thousands lines of code [LOC]) with lots of functionalities. Nonetheless, they are usually based on computing kernels. Computing kernels are resource intensive and well-defined parts of a program that usually work on precisely defined data. Those kernels represent the most time-consuming part of an HPC application, and consequently, they are the prime target for optimization.

Those applications are often developed by several individuals. Sometimes, some of those developers only work a few months on the application. Maintaining optimized code written by someone else is quite a challenge. Several languages and programming paradigms can also be used in a project and thus the maintainer must be knowledgeable in several areas of expertise. Portability problems can also be caused by the availability of an optimizing compiler for a specific language and architecture. C compilers are usually the first available while Fortran may sometimes arrive later.

In Section 5 we will present two HPC applications that we used as use cases: SPECFEM3D and BigDFT. They are both based on computing kernels and were selected as candidate applications in the Mont-Blanc project.

2.2 How Should Computing Kernels be Written?

The problematic here is to obtain computing kernels that present good performances on the architectures encountered by the application while still being portable after the optimization process took place. Indeed the application might encounter one of the many architectures that can be found in HPC. Investing manpower to optimize the application for a new architecture is reasonable, suffering hindrance from previous optimization work is not. Thus optimizations have to be as orthogonal as possible from one another so as to be easily activated and deactivated.

If this paradigm is followed by developers then they will rapidly be confronted with a huge optimization space to search. They will need to be able to test easily the performance impact of the chosen optimizations without running the full application. The same reasoning implies that kernels should be tested for non regression without running the full application.

What we want is computing kernels that are written:

- in a *portable* manner,
- in a way that raises developer *productivity*,
- and, present good *performance*.

2.3 Evolution of HPC Architectures

Evolution of HPC Architectures is rapid and also diverse: in the last 5 years no less than 6 architectures have been number one in the Top500:

- Intel Processor + Xeon Phi (Tianhe-2)
- AMD Processor + NVIDIA GPU (Titan)
- IBM BlueGene/Q (Sequoia)
- Fujitsu SPARC64 (K computer)
- Intel Processor + NVIDIA GPU (Tianhe-1)
- AMD Processor (Jaguar)

Being able to efficiently use those architectures on such a small time-frame is challenging.

The race to exascale is not going to simplify the environment. All of the above architectures can be considered. Network architectures also can be very diverse. For instance European FP7 project DEEP considers using Accelerators (XEON Phi) while the European FP7 project Mont-Blanc considers using low-power embedded processor with integrated GPU.

Running existing applications on those new architectures is an open research subject as well as an ongoing porting effort. Thus, those projects have work packages dedicated to applications. Those work packages are dedicated to porting and optimizing Scientific applications on those new architectures. In the DEEP project six applications were selected for porting and optimizing, eleven were selected in the Mont-Blanc project.

3 Motivating Example: OpenCL Laplace

During one of the Mont-Blanc face to face meeting a talk on OpenCL optimization on the Mali GPU was given. One of the case study was a Laplace filter on the GPU [2].

This is a good example of an algorithm that is simple in its formulation but can be complex to optimize because many different optimizations are available and can be combined together. The correct combination of optimizations will depend on the targeted architecture.

3.1 The Laplace Filter

Listing 1 shows a simple implementation of the Laplace filter in C99. For clarity, boundary conditions have been omitted.

```
void laplace(const int width,
             const int height,
             const unsigned char src[height][width][3],
                   unsigned char dst[height][width][3]){
  for (int j = 1; j < height-1; j++) {
    for (int i = 1; i < width-1; i++) {
      for (int c = 0; c < 3; c++) {
        int tmp = -src[j-1][i-1][c] -    src[j-1][i][c] - src[j-1][i+1][c]\
                - src[j  ][i-1][c] + 9*src[j  ][i][c] - src[j  ][i+1][c]\
                - src[j+1][i-1][c] -    src[j+1][i][c] - src[j+1][i+1][c];
        dst[j][i][c] = (tmp < 0 ? 0 : (tmp > 255 ? 255 : tmp));
      }
    }
  }
}
```

Listing 1: Laplace C99 Filter

A naive implementation in OpenCL is proposed in Listing 2. Each work item processes one pixel of the resulting image.

```
kernel laplace(const int width,
               const int height,
               global const uchar *src,
               global       uchar *dst){
  int i = get_global_id(0);
  int j = get_global_id(1);
  for (int c = 0; c < 3; c++) {
    int tmp = -src[3*width*(j-1) + 3*(i-1) + c]\
            - src[3*width*(j-1) + 3*(i  ) + c]\
            - src[3*width*(j-1) + 3*(i+1) + c]\
            - src[3*width*(j  ) + 3*(i-1) + c]\
          + 9*src[3*width*(j  ) + 3*(i  ) + c]\
            - src[3*width*(j  ) + 3*(i+1) + c]\
            - src[3*width*(j+1) + 3*(i-1) + c]\
            - src[3*width*(j+1) + 3*(i  ) + c]\
            - src[3*width*(j+1) + 3*(i+1) + c];
    dst[3*width*j + 3*i + c] = clamp(tmp, 0, 255);
  }
}
```

Listing 2: Laplace OpenCL Filter

3.2 Possible Optimizations of the OpenCL Version

Several optimizations were proposed and successively applied to the previous implementation.

Vectorization The first proposed optimization involves computing five pixels instead of one using the vectors available to the Mali architecture. The vectors have to reside in memory to ensure an efficient execution. Using vectors of 16 elements, 15 useful components are simultaneously loaded and can be used in computation. This optimization yields speedups between 1.5 and 6 depending on the image size.

```
kernel laplace(const int width,
               const int height,
               global const uchar *src,
               global       uchar *dst){
  int i = get_global_id(0);
  int j = get_global_id(1);
```

```
  uchar16 v11_ = vload16( 0, src + 3*width*(j-1) + 3*5*i - 3 );
  uchar16 v12_ = vload16( 0, src + 3*width*(j-1) + 3*5*i     );
  uchar16 v13_ = vload16( 0, src + 3*width*(j-1) + 3*5*i + 3 );
  uchar16 v21_ = vload16( 0, src + 3*width*(j  ) + 3*5*i - 3 );
  uchar16 v22_ = vload16( 0, src + 3*width*(j  ) + 3*5*i     );
  uchar16 v23_ = vload16( 0, src + 3*width*(j  ) + 3*5*i + 3 );
  uchar16 v31_ = vload16( 0, src + 3*width*(j+1) + 3*5*i - 3 );
  uchar16 v32_ = vload16( 0, src + 3*width*(j+1) + 3*5*i     );
  uchar16 v33_ = vload16( 0, src + 3*width*(j+1) + 3*5*i + 3 );

  int16 v11 = convert_int16(v11_);
  int16 v12 = convert_int16(v12_);
  int16 v13 = convert_int16(v13_);
  int16 v21 = convert_int16(v21_);
  int16 v22 = convert_int16(v22_);
  int16 v23 = convert_int16(v23_);
  int16 v31 = convert_int16(v31_);
  int16 v32 = convert_int16(v32_);
  int16 v33 = convert_int16(v33_);

  int16 res = v22 * (int)9 - v11 - v12 - v13 - v21 - v23 - v31 - v32 - v33;
        res = clamp(res, (int16)0, (int16)255);
  uchar res_ = convert_uchar16(res);

  vstore8(res_.s01234567, 0, dst + 3*width*j + 3*5*i);
  vstore8(res_.s89ab,     0, dst + 3*width*j + 3*5*i + 8);
  vstore8(res_.scd,       0, dst + 3*width*j + 3*5*i + 12);
  dst[3*width*j + 3*5*i + 14] = res_.se;
}
```

Listing 3: Laplace OpenCL Filter Vectorized

Synthesizing Loads It is possible to reduce the number of loads since the vectors are overlapping. Listing 4 shows how the vectors are loaded in this case. This optimization yields marginal improvements.

```
  uchar16 v11_ = vload16( 0, src + 3*width*(j-1) + 3*5*i - 3 );
  uchar16 v13_ = vload16( 0, src + 3*width*(j-1) + 3*5*i + 3 );
  uchar16 v12_ = uchar16( v11_.s3456789a, v13_.s56789abc );
  uchar16 v21_ = vload16( 0, src + 3*width*(j  ) + 3*5*i - 3 );
  uchar16 v23_ = vload16( 0, src + 3*width*(j  ) + 3*5*i + 3 );
  uchar16 v22_ = uchar16( v21_.s3456789a, v23_.s56789abc );
  uchar16 v31_ = vload16( 0, src + 3*width*(j+1) + 3*5*i - 3 );
  uchar16 v33_ = vload16( 0, src + 3*width*(j+1) + 3*5*i + 3 );
  uchar16 v32_ = uchar16( v31_.s3456789a, v33_.s56789abc );
```

Listing 4: Laplace OpenCL Filter Vectorized with Synthesized Loads

Temporary Variables Size Using *int* to store intermediary results is unnecessary. Listing 5 show how the code is modified to use smaller types. This yields a speedup of 1.3 for most image sizes.

```
  short16 v11 = convert_short16(v11_);
  short16 v12 = convert_short16(v12_);
  short16 v13 = convert_short16(v13_);
  short16 v21 = convert_short16(v21_);
  short16 v22 = convert_short16(v22_);
  short16 v23 = convert_short16(v23_);
  short16 v31 = convert_short16(v31_);
  short16 v32 = convert_short16(v32_);
  short16 v33 = convert_short16(v33_);

  short16 res = v22 * (short)9 - v11 - v12 - v13 - v21 - v23 - v31 - v32 - v33;
          res = clamp(res, (short16)0, (short16)255);
```

Listing 5: Laplace OpenCL Filter Vectorized Using *short*

More Optimizations Several other optimizations can be attempted at this point. For instance reducing or increasing the number of pixels each work item is processing. They can yield improved performance in some cases, especially when avoiding memory alignment problems. These optimizations will not be shown here but can be found in [2].

3.3 Improving the Methodology

The process described in the previous Subsection is quite tedious and requires some intimate knowledge of the target architecture. The results are impressive, as speedups of almost 10 are observed compared to the naive OpenCL implementation.

Nonetheless, the process is frustrating. The optimizations are never evaluated independently from one another. Some were arbitrarily configured (the number of pixels chosen for instance). Testing the different combinations of those optimizations and the different parameters would prove very costly in developer time and like all repetitive and tedious tasks, error prone. Thus every created kernel would have to be thoroughly tested to ensure no error was done.

In the next Section we will present our answer to this problematic: BOAST. BOAST is a framework dedicated to kernel description, optimization, regression testing and autotuning.

4 BOAST: Using Code Generation in Application Autotuning

BOAST provides scientific application developers with a framework to develop and test application computing kernels [5]. Figure 1 illustrates the envisioned work-flow and program structure. The user starts from an application kernel (either designed or implemented), and write it in a dedicated language (step 1). The language provides enough flexibility for the kernel to be meta-programmed with several orthogonal optimizations. Then the developer selects a kernel configuration and a target language. Those parameters define the output source code that will be generated by BOAST (step 2). The resulting code source is then built according to the user specified compiler and options (step 3). If input data are available then the kernel can be benchmarked and tested for non regression. Based on the results, other optimizations can be selected (step 4) or new optimizations can be added to the BOAST sources. The process can be repeated until a good candidate is found on the target platform. The resulting kernel is then added to the program (step 5).

Several steps in the work-flow can be automated: optimization and compiler flags exploration, non regression testing, benchmarking... This automation can be scripted by the user or by interfacing other dedicated tools. In order to achieve those goals three aspects should be considered:

- code description,
- code generation,
- and kernel execution run-time.

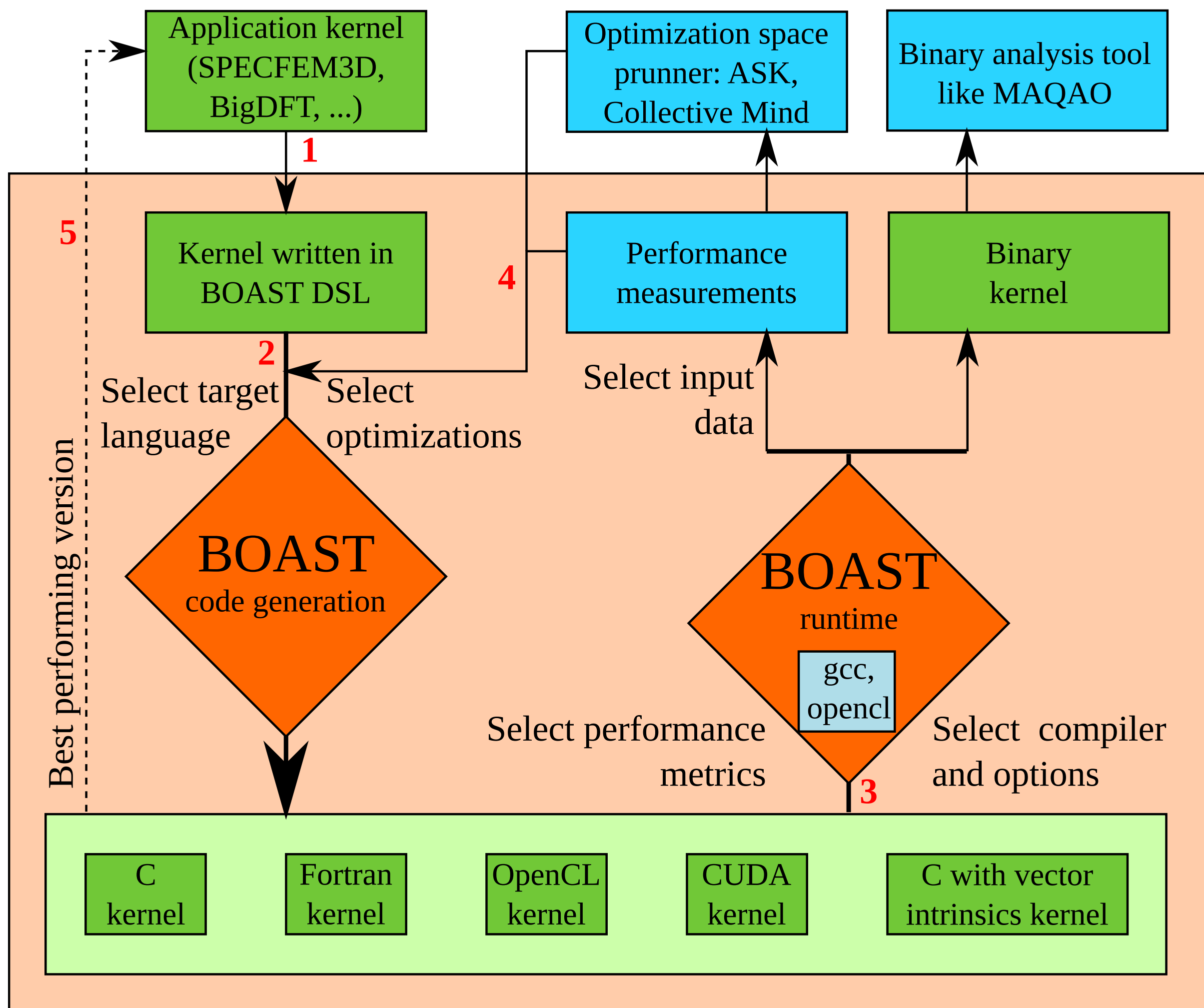

Figure 1: Structure and Work-flow of the BOAST framework.

4.1 Kernel Description Language

Usually computing kernels are hotspots of an HPC application, and they are most of the time based on a loop nest. A lot of efforts are dedicated to their tuning and the obtained result is often quite different from the original procedure. Several transformations can be applied to such kernels. And those optimizations are often applied manually as compiler may fail to recognise the opportunity.

There are many different loop optimization techniques [29]. We can cite loop skewing [30] (derives nested loops wavefronts) or loop interchange [3] (loop variables change places).

The importance of correct loop imbrication on BLAS [19] operations is studied in [25], and shows performance increase of a factor up to 5 when using correct loop imbrication. The importance of code transformation is stressed in [31], where a selection of GPU kernels are ported to CPU and optimized.

BOAST kernel description language should be able to express all these optimizations. This gives us a set of constraints to implement in the language:

- Arbitrary number of variables have to be created and manipulated (types, attributes...).
- Procedures have to be abstracted (reunite Fortran and C like languages, attributes...).
- Functions must be available.
- Variables, constants and functions must be composed in complex Expressions.
- Basic control structures (for, while, if/else...) have to be abstracted.
- Powerful array management (several dimensions, transformations, indexing...).

In order to manipulate those abstractions we want to have a syntax resembling what programmers use. For instance, commonly used operators have to be available and behave as expected. It must also be possible to differentiate an action on an abstraction in the context of an expression and in the context of the management of this expression. $c = a + b$, an expression that affects the results of $a + b$ to c is not equal to $c \leftarrow a + b$, which saves the expression $a + b$ to a variable c. This is why an embedded domain specific language approach was selected [16]. This allows for the coexistence of two languages: the host language and the Domain Specific Language (DSL). In our case, DSL allows the description of the kernel ($c = a + b$) while the host language provides the meta programming of the kernel ($c \leftarrow a + b$). Operator overloading of the host language will provide the familiar syntax programmers are accustomed to. The host language will also provide easy interfacing with libraries we might need during the development of our framework.

It was also important to have our constructs like *for* loops to have a syntax approaching those commonly found in programming languages. To this end, we needed a language which could seamlessly pass a block of code to a function. Ruby [20] is one such language. It has deep introspection capabilities as well. This is the main reason why it was selected.

4.1.1 BOAST Keywords

In order to clearly differentiate what is going to be generated from what is related to manipulations in the host language four keywords were defined. They are *decl*, *pr*, *opn* and *close*. As the language is an EDSL, these four keywords are methods in the BOAST namespace. Sample usage of these keywords will be found in the next Figures.

The *decl* method is used to declare variables or procedures and functions.

```
i = BOAST::Int("i") # or BOAST::Variable::new("i", Int)
k = BOAST::Int("k", :size => 8)
l = BOAST::Real("l", :dim => [ BOAST::Dim(-5, 21) ], :local => true )
BOAST::decl i, k, l
BOAST::pr i === 5
j = i + 5
BOAST::pr k === j * 2
BOAST::pr l[k] === 1.0
BOAST::register_funccall("sin")
BOAST::pr l[k+1] === BOAST::sin(j)
```

Listing 6: BOAST code

```
integer(kind=4) :: i
integer(kind=8) :: k
real(kind=8), dimension(-5:21) :: l
i = 5
k = (i + 5) * (2)
l(k) = 1.0_wp
l(k + 1) = sin(i + 5)
```

Listing 7: Fortran output.

```
int32_t i;
int64_t k;
double l[27];
i = 5;
k = (i + 5) * (2);
l[k - (-5)] = 1.0;
l[k + 1 - (-5)] = sin(i + 5);
```

Listing 8: C output.

Figure 2: BOAST Code Snippet for Variables and Expressions

The *pr* method calls the public *pr* method of Objects it is called on. Each BOAST object is responsible for printing itself correctly depending on the BOAST configuration at the time the print public method is called. Calling directly the print method of a BOAST object yields the same result.

The *opn* method can be used to print the beginning of a control structure without an associated code block.

The *close* method is the counterpart to the *opn* method. It is used to close a control structure without an associated code block.

4.1.2 BOAST Abstractions

BOAST defines several classes that are used to represent the structure of the code. These classes can be sorted in two groups, algebraic related and control flow related:

Algebra The first and most fundamental abstraction is named Variable. Variables have a type, a name and a set of named attributes. The existing attributes are mainly inspired from Fortran. Those attributes are not limited and can be arbitrarily enriched, allowing a lot of flexibility in Variable management.

The second abstraction is named Expression. It combines variables into algebraic or logic expressions. Most of the classical operators are overloaded for those two abstractions and thus the syntax of the expressions are rather straightforward. The exception is the assignment operator as it is important to differentiate between assigning an Expression or a Variable in the Ruby context and the assignment operator in the context of a BOAST Expression. Thus the assignment in a BOAST Expression is represented as the === operator, while the classical assignment is kept as the = operator. Function calls (FuncCall) are also abstracted and can be used in Expressions.

Figure 2 shows some basic usage of both Variables and Expressions as well as the *pr* and *decl* keywords. For clarity we stayed out of BOAST namespace so BOAST related class and methods are prefixed with *BOAST::*. Listing 6 shows the BOAST code that produces the

```
1 BOAST::register_funccall("modulo")
2 i = BOAST::Int("i")
3 j = BOAST::Int("j")
4 BOAST::pr j === 0
5 BOAST::pr BOAST::For( i, 0, 100 ) {
6   BOAST::pr BOAST::If( BOAST::modulo(i,7) == 0 ) {
7     BOAST::pr j === j + 1
8   }
9 }
```

Listing 9: BOAST code

```
1 j = 0
2 do i = 0, 100, 1
3   if (modulo(i, 7) == 0) then
4     j = j + 1
5   end if
6 end do
```

Listing 10: Fortran output.

```
1 j = 0;
2 for (i = 0; i <= 100; i += 1) {
3   if (modulo(i, 7) == 0) {
4     j = j + 1;
5   }
6 }
```

Listing 11: C output.

Figure 3: BOAST Code Snippet for control structures

Fortran (Listing 7) and C (Listing 8) output. At first we define 2 Variables i and k (note that k is 64 bit integer). The third variable named l is a one dimensional local array of length 27, it is indexed in the range -5 to 21. All those Variables are affected to Ruby variables of the corresponding name.

On Line 4 we declare those three variables. Variable i is then affected the value 5. On Line 6 the j Ruby variable is used to store the BOAST expression $i + 5$. This variable will the be used transparently through the rest of the program.

On Line 8 we use the k variable to index into the array l using the bracket operator. Would the array be multidimensional the index would be comma separated, similar to Fortran notation.

On the last line, a call to the *sin* function is made through the creation of a FuncCall object. The possibility to use this method is declared using the *register_funccall* method.

Control Structures The classical control structures are implemented. *If*, *For*, *While*, *Case* are abstractions in BOAST matching the behavior of corresponding control structures in other languages. An exception is the For in BOAST more closely matches the for in Fortran than the one in C.

Figure 3 shows some basic usages of the control structures. The example shows a C macro or function that behaves similarly to the Fortran modulo intrinsic. The sample script (inefficiently) computes and stores in j the number of multiples of 7 in the 0 to 100 range. It uses the For and If control structures. A Ruby block is passed to each of those constructs. This block is evaluated at the time the construct is printed. If several such constructs are needed (in an *if elsif else* case, for instance) they can be explicitly passed as parameters using the *lambda* Ruby keyword.

The last control structure is Procedure. It describes either procedures or functions. Figure 4 presents the use of this abstraction. It illustrates the signature of a real kernel from BigDFT. This kernel uses an input array x and an output array y of double precision numbers. Those arrays have two dimensions which depend on input variables n and *ndat*. We can see here the use of the *opn* and *close* keywords that are used to print a control structure without an associated Ruby block. This time we placed ourselves inside of BOAST namespace.

The generated outputs in Fortran (Listing 13) and C (Listing 14) show the difference in

```
1 n = Int("n", :dir => :in)
2 ndat = Int("ndat", :dir => :in)
3 x = Real("x", :dir => :in, :dim => [ Dim(0, n-1), Dim(ndat) ])
4 y = Real("y", :dir => :out, :dim => [ Dim(ndat), Dim(0, n-1) ])
5 p = Procedure("magicfilter", [n, ndat, x, y])
6 opn p
7 close p
```

Listing 12: BOAST code

```
1 SUBROUTINE magicfilter(n, ndat, x, y)
2   integer(kind=4), intent(in) :: n
3   integer(kind=4), intent(in) :: ndat
4   real(kind=8), intent(in), dimension(0:n - (1), ndat) :: x
5   real(kind=8), intent(out), dimension(ndat, 0:n - (1)) :: y
6 END SUBROUTINE magicfilter
```

Listing 13: Fortran output.

```
1 void magicfilter(const int32_t n, const int32_t ndat,
2                  const double * x, double * y) {
3 }
```

Listing 14: C output.

Figure 4: BOAST Code Snippet for Procedure

meta-information that is kept between both versions.

4.2 BOAST Run-time

In the previous Section we presented BOAST's language. This allows us to describe procedures and functions and to meta-program them using Ruby. Each version has to be compiled, linked and executed to assess its performance in order to find the best version of a computing kernel. This can be very time consuming if this process cannot be automated. By enabling more versions for evaluation, the automation will bring improved portability, better performance, and in the end will improve the productivity of the developer.

In this section we will present the different aspects of BOAST run-time that allow this automation. Those aspects include:

- multi target language generation (performance, portability)
- compilation (productivity, performance)
- execution (productivity, performance)
- tracer, dumper and replay for non regression tests (productivity)

4.2.1 Multi-target Language Generation

Language availability and performance varies between platforms. It is thus important to express computing kernels in different languages, based on the availability and their respective merits on the target platform. Some languages have additional features, such as languages that target GPUs (OpenCL, CUDA). The developer should be given tools to determine what kind of language is currently used in order to be able to use those additional features.

Similarly the target language must be changed with ease in order to compare different alternatives. Two methods are dedicated to this task: *set_lang* and *get_lang*. The target language can also be set through an environment variable before launching BOAST, allowing for easy command line scripting.

4.2.2 Compilation

Compilation of the generated kernels must also be very flexible because HPC application developers may encounter platforms with very diverse compilation environments. Proprietary and dedicated compilers are common on HPC infrastructures. Thus BOAST build system exhibits similar behavior to common build systems. Compilers and their compile/build options can be specified at several places. The list by increasing order of precedence includes: BOAST configuration file, environment variables, and at kernel build time.

This way the framework to test different compiler optimizations is completely available and performance study can include both kernel related parameters and compilation related parameters. This behavior is contained in the *CKernel* class of BOAST. When instantiating this class a BOAST *Procedure* representing the entry point of the kernel is specified.

4.2.3 Execution

The next logical step is to benchmark the built kernel. BOAST offers a simple way to run a kernel that was successfully built while staying within BOAST. A built BOAST *CKernel* exposes a run method that accepts arguments corresponding to the BOAST *Procedure* used to instantiate the kernel. Arrays must be instances of *NArray* which are numerical arrays that use C arrays underneath.

Arrays which correspond to output parameters will be modified during the execution of the kernel so results can be checked. This allows for easy non regression testing. The run method also returns information (and result for kernels that are functions as well as output scalars) about the run. For instance, one such information includes the run time of the kernel and it is obtained using the system-wide real-time clock. Other probes can be inserted at compile-time, if needed. BOAST also supports PAPI [21] to capture hardware performance counters during each kernel execution.

4.2.4 Kernel Replay

The non regression methodology presented before is viable provided input data can be generated at run-time. For instance, in the case of the Laplace kernel, generating an input image and the corresponding reference output image, using a reference implementation, is easy. Unfortunately it is not always possible, some applications have complex data patterns that are difficult to synthesize without running the full application. Thus BOAST offers a way to load binary data from the file system and use them as inputs of a kernel. Outputs can also be checked against those binary data thus enabling (almost) data oblivious non regression testing.

In order to use this methodology one has to be able to trace an application to get input data. Such a tracer, dedicated to CUDA and OpenCL, will be presented in the next subsection.

4.3 Non Regression Testing Using Trace Debugging

Debugging applications running on GPU environments is well-recognized as a hard and time-consuming activity. In complement with BOAST, we designed a trace-based debugging tool that simplifies this porting operation. The tool relies on BOAST support of multi-target code

generation (Section 4.2.1), used to validate an application from one GPU-programming framework to another or between different implementations using the same programming model. A case-study of the port of a CUDA application to OpenCL is presented in Section 5.3.

The idea behind the tool is based on the assumption that the different GPU ports of the code should do the very same operations, at least at the logical level (the APIs will have *implementation* differences, but they should offer nonetheless same functionalities). The usage of BOAST framework sustains this assumption, as both sets of kernels should be generated from the same source code.

Hence, the verification and validation of the new port can be narrowed down to asserting that both codes apply the same operations on the GPU. And the debugging part will consist in understanding what diverges. To that purpose, we developed `GPUTrace`, inspired by `strace` and `ltrace` tools: `GPUTrace` dynamically preloads a library between the application and the GPU library, and collects the function name, execution range and argument values (input *and* output) of the relevant function calls. These information are traced in a unified format for all the APIs. This is a custom trace format. Listing 15 presents a sample output generated during SPECFEM 3D tracing.

However, in contrast with `strace` and `ltrace`, `GPUTrace` has to be *state-full*. Indeed, most of API parameters are handles to opaque types. So, in order to generate meaningful traces, `GPUTrace` gathers information about these objects at creation time (handle value, buffer creation size and attributes, kernel name and prototype, etc.) and re-injects this information when the objects used. A state-less implementation of `GPUTrace` would highly rely on the GPU libraries introspection capabilities, which seem not possible at the moment.

```
1  New kernel: update_potential_kernel
2  ...
3  New buffer #94, 2Mb, READ_WRITE (0x74d0420)
4  New buffer #95, 2Mb, READ_WRITE (0x74d08d0)
5  New buffer #96, 2Mb, READ_WRITE (0x74d0d80)
6  ...
7  Buffer #94 written, 2314764b at +0b
8  Buffer #95 written, 2314764b at +0b
9  Buffer #96 written, 2314764b at +0b
10 ...
11 update_disp_veloc_kernel<128,1><4522,1>(
12       float *displ=<buffer #94 1.00e-24, 1.00e-24, 1.00e-24, 1.00e-24, ...>
13       float *veloc=<buffer #95 0.00e+00, 0.00e+00, 0.00e+00, 0.00e+00, ...>
14       float *accel=<buffer #96 0.00e+00, 0.00e+00, 0.00e+00, 0.00e+00, ...>
15       const int size=<578691>
16       const float deltat=<1.365914e-04>
17       const float deltatsqover2=<9.328606e-09>
18       const float deltatover2=<6.829570e-05>
19       ----
20       <out> float *displ=<buffer #94 1.000e-24, 1.00e-24, 1.00e-24, 1.00e-24, ...>
21       <out> float *veloc=<buffer #95 0.000e+00, 0.00e+00, 0.00e+00, 0.00e+00, ...>
22       <out> float *accel=<buffer #96 0.000e+00, 0.00e+00, 0.00e+00, 0.00e+00, ...>
23 );
```

Listing 15: GPUTrace sample trace

Once two call traces have been generated by `GPUTrace`, the user can compare them with a graphical `diff` tool, and spot the different porting mistakes: two parameters reversed, an offset incorrectly applied, etc.

By default, `GPUTrace` only prints a unique identifier (the creation index) for the memory buffers. Additionally, `GPUTrace` supports several modifier flags. One flag can be activated to append the first bits of the buffer to the trace, printed in the right format, for visual inspection. Another flag can be set to dump the whole content of the buffer into a file, for a full inspection.

This last option is also useful for generating *replay buffers* for BOAST kernel execution. With a set of filters based on kernel names and execution counters, developers can precisely select

which kernel execution parameters should be dumped, for further reuse as real-case benchmarks and non-regression testing.

5 BOAST Use Cases

In this section we will present the benefits of using BOAST on the Laplace motivating example as well as two scientific applications that use BOAST. The first one, BigDFT [11], uses BOAST in order to develop new functionalities with performance portability in mind. The second one, SPECFEM3D [18], uses BOAST to factorize OpenCL and CUDA development, while having robust non regression tests.

5.1 Laplace Filter Kernel

Section 3 presented the Laplace motivating example. From this section we know that a number of optimizations can have an impact on the performance of the kernel on the Mali architecture. But, what is the impact in other architectures? And, are there additional optimizations that can impact the performance?

5.1.1 Optimization Space

The list of already identified optimizations are:

- vectorization,
- intermediary data type,
- number of pixels processed,
- and synthesizing loads.

To create our generic implementation we decided to work at the component level rather than the pixel level. This approach leads to more flexibility and genericity when applying optimizations. We also decided to study the impact of another parameter which is the number of components to process on the column direction. This leads to being able to process tiles instead of only rows.

Here are the parameters we finally selected for our kernel optimization and their possible values:

- *x_component_number*: a positive integer
- *y_component_number*: a positive integer
- *vector_length*: 1, 2, 4, 8 or 16
- *temporary_size*: 2 or 4
- *synthesize_loads*: *true* or *false*
- *vector_recompute*: *true* or *false*

Image Size	Naive (s)	Best (s)	Acceleration	BOAST (s)	Acceleration
768 x 432	0.0107	0.00669	x1.6	0.000639	x16.7
2560 x 1600	0.0850	0.0137	x6.2	0.00687	x12.4
2048 x 2048	0.0865	0.0149	x5.8	0.00715	x12.1
5760 x 3240	0.382	0.0449	x8.5	0.0325	x11.8
7680 x 4320	0.680	0.0747	x9.1	0.0581	x11.7

Table 1: Best performance of ARM Laplace kernel.

The last parameter is used when *x_component_number* is not divisible by *vector_length*. Two solutions are possible then, divide the remainder of the division in vectors of smaller sizes (*vector_recompute* = *false*) or load more data and compute useless values in vectors of the specified size (*vector_recompute* = *true*). This last option mimics the behavior of the ARM implementation, although when working at the component level it may not be a valuable thing to try.

5.1.2 Performance Results

Table 1 show the best results obtained by ARM on different images compared to the naive implementation and to the best version version BOAST found. It shows that the generated version systematically outperforms the hand optimized version. As far as the optimization options are concerned, the results are disappointing: the same kernel configuration is the best for all image sizes. This kernel uses *x_component_number* = 16, *y_component_number* = 1, *vector_length* = 16, *temporary_size* = 2 and *synthesize_loads* = false. *vector_recompute* because *x_component_number* == *vector_length*. Those results show that, when working on full vectors, synthesizing the loads is harmful to performance and the programmer is better of trusting the cache to load each vector in one cycle without compromising the bandwidth. The results shown here are the best of four runs for each configuration.

But what if we run our benchmark on other architectures? Table 2 shows the results obtained when running our BOAST implementation on other architectures. The chosen architectures include an Intel i7-2760QM CPU (Sandy Bridge architecture) that supports OpenCL 1.2 and a system with an NVIDIA gtx680 GPU that supports OpenCL 1.1. We see that the performance ratio between the different architectures is stable across image sizes.

The optimization parameters selected are not the same for those architectures. Indeed, the Intel CPU favors kernels that have the parameters: *x_component_number* = 16, *vector_length* = 8, *temporary_size* = 2 and *synthesize_loads* = false. Once again *vector_recompute* does not apply. *y_component_number* varies from 4 to 2 when image size increase, thus decreasing task granularity as the global work size increase. This result is unexpected and understanding why this happen could be interesting. The NVIDIA GPU favors processing square tiles: *x_component_number* = 4, *y_component_number* = 4, *vector_length* = 4, *temporary_size* = 2 and *synthesize_loads* = false. Once more, *vector_recompute* has no meaning.

5.1.3 Laplace Conclusion

In this Subsection we have shown the interest of BOAST in optimizing a well-known algorithm across different architectures. Chosen optimization combinations are highly dependent on the targeted architecture. In the next Subsections we will show that our methodology also applies to real applications.

Image Size	BOAST ARM (s)	BOAST Intel	Ratio	BOAST NVIDIA	Ratio
768 x 432	0.000639	0.000222	x2.9	0.0000715	x8.9
2560 x 1600	0.00687	0.00222	x3.1	0.000782	x8.8
2048 x 2048	0.00715	0.00226	x3.2	0.000799	x8.9
5760 x 3240	0.0325	0.0108	x3.0	0.00351	x9.3
7680 x 4320	0.0581	0.0192	x3.0	0.00623	x9.3

Table 2: Best performance of Laplace Kernel on several architectures.

5.2 Creating an Auto-Tuned Convolution Library for BigDFT using BOAST

In 2005, the EU FP6-STREP-NEST BigDFT [11] project funded a consortium of four European laboratories (L_Sim - CEA Grenoble, Basel University - Switzerland, Louvain-la-Neuve University - Belgium and Kiel University - Germany), with the aim of developing a novel approach for DFT calculations based on Daubechies wavelets. Rather than simply building a DFT code from scratch, the objective of this three-years project was to test the potential benefit of a new formalism in the context of electronic structure calculations.

As a matter of fact, Daubechies wavelets exhibit a set of properties which make them ideal for a precise and optimized DFT approach. In particular, their systematicity allows to provide a reliable basis set for high-precision results, whereas their locality (both in real and reciprocal space) is highly desired to improve the efficiency and the flexibility of the processing. Indeed, a localized basis set allows to optimize the number of degrees of freedom for a required accuracy [11], which is highly desirable given the complexity and inhomogeneity of the systems under investigation nowadays.

Despite the application is mainly written in Fortran, it currently includes 360 kLOC and 70kLOC of Fortran and C languages, respectively, accounting for more than 50% of the code base. It is a parallel application based on the standards MPI [1] and OpenMP [6]. It also supports CUDA [22] and OpenCL [17]. In the recent years this code has been used for many scientific applications, and its development and user consortium is continuously growing. Massively parallel computations are routinely executed with the BigDFT code, either in homogeneous or hybrid architectures. In 2009, the French Bull-Fourier award was attributed for the implementation of the hybrid version of BigDFT [12].

5.2.1 BigDFT

In the Kohn-Sham (KS) formulation of DFT, the electrons are associated to wavefunctions (orbitals), which are represented arrays of floating point numbers. In wavelets formalism, the operators are written via convolutions with short, separable filters. The detailed description of how these operations are defined is beyond the scope of this report and can be found in the BigDFT reference paper [11].

The CPU convolutions of BigDFT have thus been thoroughly optimized. In a recent paper [27], the optimization of the CPU convolutions of BigDFT has been extensively considered. One example of a specific convolution, called MagicFilter [13], can be seen in Listing 16. It applies a filter $filt$ to the data set *in* and then stores the result in the data set *out* with a transposition [14].

```
1 double filt[16] = {F0, F1, ... , F15};
2 void magicfilter(int n, int ndat, double* in, double* out){
3   double temp;
```

```
4    int m;
5    for( j = 0; j < ndat; j++) {
6      for( i = 0; i < n; i++) {
7        temp = 0;
8        for( k = 0; k < 16; k++) {
9          m = (i-7+k)%n
10         temp += in[m + j*n] * filt[k];
11       }
12       out [j + i*ndat] = temp ;
13     }
14   }
15 }
```

Listing 16: MagicFilter.

As we can see, there are three nested loops working on arrays whose sizes vary. Various optimizations can be applied to this treatment and may focus on the loop structure, as well as on the size of the data.

5.2.2 A Generic Convolution Library

The number of convolution kernels needed in BigDFT has been continuously growing in the recent years. Various boundary conditions and functionalities have been added, making the BigDFT more and more powerful in terms of scientific applications. However, the cost of the maintenance and of development of the convolutions is always a delicate point to be considered while including a new functionality. The convolution patterns are usually rather similar, leading to code duplication and difficulties in code maintainability.

It appears therefore very interesting to benefit from an automatic tool to drive the implementation and the generation of new convolutions. This would lead to an optimized code, adapted to different computing platforms, that is optimally factorized. In addition to this point, the help of such code generator is also important to build *new science*: the cost of implementing new convolutions would become so little that other functionalities (for example, the generalization of the BigDFT convolutions to Neumann boundary conditions or the usage of wavelet-on-the-interval basis) can be added with limited manpower.

For these reasons, a convolution library has been engineered with the help of BOAST. The detailed API of the library will be described elsewhere. The spirit is similar to BLAS-LAPACK API, where low-level operations are scheduled and called from high-level operations. The basic blocks will be comprised of unidimensional wavelet transforms and convolutions applied to multidimensional arrays. Combining those blocks will yield multidimensional transforms.

Each of the building blocks of these convolution libraries is optimally tuned by BOAST by choosing the sources providing the optimal kernel for the chosen computing platform. The sources of these kernels are then collected and compiled to meet the API of the library.

A library written in this way might have an impact going largely beyond the community of BigDFT developers. Indeed, convolutions are basic operations of lots of scientific application codes, like for example finite differences approaches, which are universally used in computational physics.

The interest in having a robust and optimally tuned library in this scientific field is therefore evident. Techniques are under investigation to provide also fine-tuned binaries to the end-user rather than the code sources, such that more aggressive inter procedural optimization can be performed. Indeed, BOAST finds the optimal source code for a given kernel and compiler configuration but one could imagine using binary optimizer to the compiled binaries. Those optimizers could be coupled to BOAST and the output of the process would be the final binary rather than the source code. This experience therefore would be a first step toward the release of a tunable optimised convolution library oriented to computational physics communities.

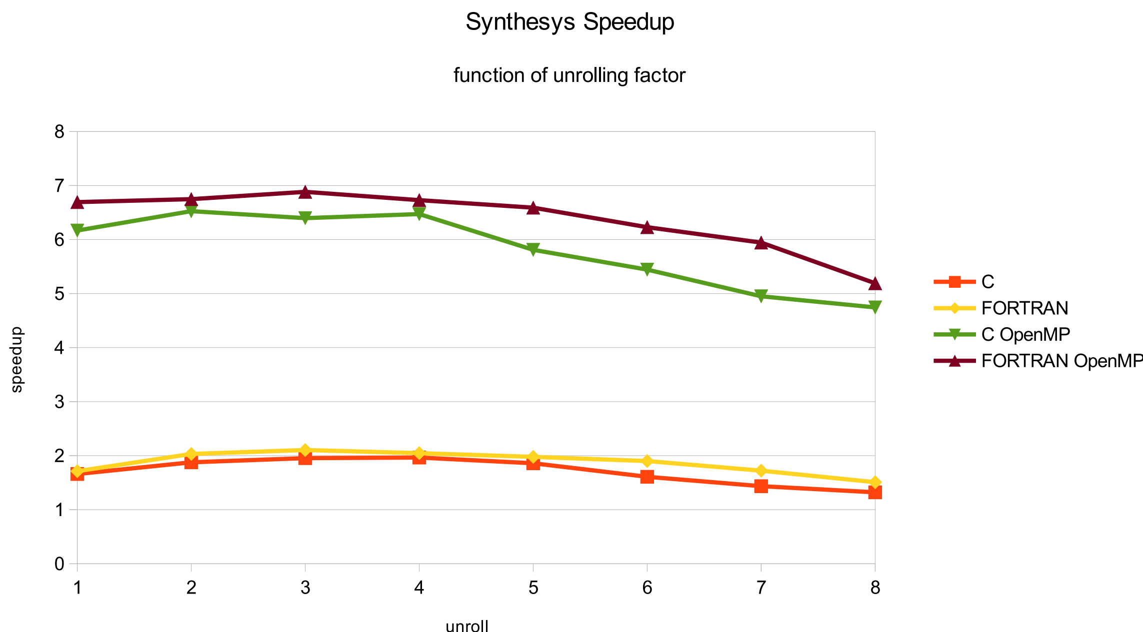

Figure 5: Impact of Unrolling, Language and OpenMP on a Wavelet Transform

5.2.3 Performance report

Several kernels have already been implemented in BOAST for the convolution library. Figure 5 shows the performance of the wavelet transform operation as a factor of the unrolling length of the outer loop, the language used to implement it as well as the activation or not of the OpenMP parallelization. Results are given as a speedup compared to the sequential hand tuned implementation that can be found in BigDFT. These tests were run on the Intel Xeon X5550 that was used to hand optimize the code.

We can see that this function is better optimized using Fortran and small unrolling factors. In the hand optimized version the unrolling factor was chosen much too high (a factor of 12 was used). This factor might have been optimal at the time the procedure was optimized (compiler version changed in the meantime) but since then the environment changed. Other optimizations have also been incorporated in the BOAST sources, like the systematic inner loop unrolling, and those could also help increase performance while limiting the interest of the outer loop unrolling.

Nonetheless what is interesting from the physicist point of view is that the generated source will give its better performance on a whole range of architectures/compilers combinations than that of the hand tuned code.

5.3 Porting SPECFEM3D to OpenCL using BOAST

In this last subsection, we study a free seismic wave propagation simulator, Specfem 3D, in its Globe flavor[1]. Specfem 3D simulates seismic wave propagation at the local or regional scale based upon spectral-element method (SEM), with very good accuracy and convergence properties. It is a reference application for super computer benchmarking, thanks to its good scaling capabilities.

[1]Specfem3d Globe — CIG `http://www.geodynamics.org/cig/software/specfem3d-globe`

5.3.1 SPECFEM3D

When we started to work on the project (version v2.1 of July 2013) it supported graphics card GPU acceleration through NVidia Cuda. This GPU support came in addition to the MPI support implemented to enable multi-CPU parallel computing. Most of Specfem 3D code-base is written in Fortran2003, only the GPU related parts are written in C. The split between CPU and GPU code was done at a rather fine grain, as the application counted more than 40 GPU kernels. Some of them were quite simple (e.g., performing a few vector operations, but at massively parallel scale), while at the other side of the spectrum, some complex kernels took more than 80 parameters and perform very specific physical transformations.

Because of the complexity of wave propagation (and of the application architecture), it is hard to impossible to define unit tests. Hence, the application is validated by the accuracy of the results it produces, that is, the accuracy of seismograms of the simulated earthquakes, in comparison with the actual ones. Non-regression testing (after new developments) is also based on these seismograms, with measurements of the relative error between two identical simulations.

As part of the Mont-Blanc project, we had to port Specfem 3D to OpenCL, so that it could be used to benchmark Mont-Blanc HPC platforms.

5.3.2 Porting to OpenCL

Porting kernels to BOAST Nvidia Cuda and OpenCL are based on the same programming model: a massively parallel accelerator running in disjoint, non addressable memory environment. Thanks to that proximity, we have been able to carry out most of the porting task with only a limited knowledge about Specfem 3D internal physics.

This lack of Specfem 3D internal knowledge led us to be particularly careful to the path we undertook for the port, as we would have been unable to understand how and why the application was not operating properly, if it was to fail.

Hence, our first milestone in the port was the translation of Specfem 3D's Cuda kernels into BOAST EDSL. This way, we could ask BOAST framework to generate a CUDA version of the kernels, plug them back into Specfem 3D and get (after fixing compile-time errors—prototypes and naming mistakes mainly) a first set of Specfem 3D seismograms.

As we had expected, the seismograms were erroneous. But with the help of shell scripts and BOAST framework ability to store and provide the kernels' original source code, we built a set of Specfem 3D binaries including only *one* BOAST-generated kernel, with all others reference-kernels. Running and validating all these binaries enabled use to pinpoint the misbehaving kernels. We finished the debugging with a side-by-side comparison that highlighted the coding mistakes.

Porting run-time to OpenCL The second part of the port consisted in the translation of the CPU-side of the application, from Cuda API to OpenCL API. Most of the functions of the interfaces are very similar, with only naming-convention and data-structure distinctions. Hence, it was clear that automatic rewriting tools (namely `sed` regexp and `emacs-lisp` functions) could be useful. To give an idea of the cost of a *manual* rewriting, we can count (in # of OpenCL API function calls): 70 kernel "function calls", 790 arguments to set, 230 memory transfers, 160 buffer creations, and 270 releases.

Once the transformations were applied, compilation errors fixed, and OpenCL unsuccessful function calls solved, the application managed to complete its execution and generate a first set of seismograms. And again, as expected and feared, these seismograms were not valid as their shape was completely different from the reference ones.

As we had already validated BOAST-generated kernels (and trusted CUDA and OPENCL versions to be semantically identical), we knew that the bugs were now in the usage of the run-time, and we had to find a way to understand where SPECFEM 3D's CUDA version of the code diverged from its OPENCL counterpart. To help us in that purpose, we had a strong assumption: both versions of the code were supposed to perform exactly the same operations, with the same "logical" parameters (the APIs have *implementation* differences, for instance OPENCL has two memory transfer functions, `clEnqueueReadBuffer`, `clEnqueueWriteBuffer`, whereas CUDA has only one, with a direction parameter `cudaMemcpy(..., dir)`, but above that, it is the same functionality).

Hence, our idea for locating the execution problems was to make sure that both execution actually did the same thing. As the OPENCL results were invalid, we knew the executions would diverge at one or several points.

Debugging OpenCL Execution: `GPUTrace` With the help of `GPUTrace` (Section 4.3), we could confront CUDA and OPENCL execution traces with a graphical `diff` tool, and spot the different porting mistakes: some parameters reversed, offsets incorrectly applied, etc..

One last problem remained, clearly highlighted by the seismograms not matching perfectly (they had a similar shape, but with a reduced intensity). We added more verbosity to `GPUTrace` output: first the initial bits of the GPU memory buffers, then their full content. The drift was visible in the trace, but it was nonetheless unclear where it started. We finally got it after hours of code review of BOAST kernels and OPENCL code. One kernel was *three*-dimensional, whereas the others were two-dimensional. But for all of them, only two dimensions were passed, and one was missing.

Evaluation Our OPENCL/BOAST port of SPECFEM 3D is now merged in SPECFEM 3D's development tree and under test and extension by different research teams. On a platform with two K40x GPUs and N Intel Xeon processors, we measured similar results between the original CUDA version and our BOAST-CUDA version. With the same set of optimization flags, BOAST CUDA and OPENCL version reported similar execution time spans. The best execution speed was achieved with CUDA version though (25% higher than OPENCL), as one optimization parameter (`CUDA_LAUNCH_BOUND`) cannot be passed, to OPENCL run-time, as of version 1.1. This parameter, in addition to specifying the work group size (which can be done in OPENCL), also constrains the number of work group that must run in parallel on a multiprocessor. This value is set to 7 in CUDA. This means that the compiler must be very conservative on register usage in order to allow this parallelism which allows better overlapping of communications end computations. This functionality is not supported in OPENCL.

By refactoring GPU kernel code in BOAST EDSL, the size of kernel code shrank by a factor of 1.8 (from 7500 to less than 4000 LOC, partly because of code duplication, also removing manually unrolled loops). This is beneficial for SPECFEM 3D as it improves the readability and maintainability of its source code.

We have also been able to enhance SPECFEM 3D's non-regression test-suite by adding per-kernel non-regression tests. This was done with the help of `GPUTrace`, that we used to capture all the input parameters of a particular *valid* kernel execution, as well as the output values. Then, during the non-regression testing, BOAST framework loads these buffer files, allocates GPU memory and initializes it through CUDA or OPENCL run-time, and triggers the kernel execution. A comparison of the output values (for instance against a maximal error level) validates the non-regression.

In the same mindset, we provided SPECFEM 3D test-suite with kernel performance evaluation mechanisms. These tests will help developers to try new optimizations in kernel code and

measure their impact, without executing the whole application.

6 Related Work

Code generation and auto-tuning techniques are not new. Nonetheless, with recent developments in hardware, and the HPC landscape being as diverse as it is now, there is a renewed interest in the field. This related work Section is split in three parts, focusing first on Auto-tuning frameworks. Tools that provide a DSL to describe computing kernels will then be presented. Last, optimization space pruners and their ties to auto-tuning will be introduced.

6.1 Application Auto-Tuning

The most convenient way to obtain an application that can be auto-tuned on a given platform is to base this application on a widely used computing library. BLAS [9] and LAPACK [4] are such libraries. These libraries are either hand tuned for selected platforms or have auto-tuned implementations. Atlas [28] is an auto-tuned implementation of BLAS/LAPACK. ATLAS authors defined the *Automated Empirical Optimization of Software* methodology that we implemented with BOAST. Their kernel generation is done using macro-functions in C.

Nonetheless many application formalisms cannot be reduced to standardized library or border on what could be considered edge cases for those libraries and not as optimized as more general cases. Orio [15] is an auto-tuning framework that has an approach close of that of BOAST but they are based on an annotated DSL describing loop transformation rather than a more generic EDSL. Halide [24] is an auto-tuning framework dedicated to image processing. It can also be used to describe other operations on memory buffers. Those two frameworks propose automated search space exploration to find the best version of a kernel. LGen [26] is a compiler that generates linear algebra programs for small fixed size problems. Knowing the problem size it fully unrolls and vectorizes loops, yielding better performance than state of the art generic implementations.

6.2 Kernel Description DSL

The idea to describe computing kernels using a DSL has been already explored. SPIRAL [23] is a decade old generation framework for signal processing. It uses a proprietary DSL, SPL (Signal Processing Language), to describe a DSP algorithm. This DSL is then transformed into efficient programs in high level languages like C or Fortran. POET [32] also uses a DSL to describe custom code transformations, like loop unrolling, loop blocking and loop interchange. Those transformation can be parametrized in order to tune the application. Orio [15] can be compared to POET as it aims at describing possible code transformations using a DSL. All those approaches are very different from our as they put the emphasis on compilation techniques whereas BOAST relies on the user to express the different optimizations.

Halide [24] is closer in some ways to our approach as it uses an embedded C++ DSL to describe image processing algorithm. This DSL allows decoupling the algorithm description from its scheduling. Each pixel in the resulting image has a completely defined dependency tree with regard to pixels in the input image (and intermediary results). During generation, depending on memory and computing cost, some values are recomputed rather than fetched from memory. We used Halide to implement the *magicfilter* of BigDFT, but unfortunately results were 4 to 5 times slower than the one we obtained with BOAST. We speculate that the

three-dimensional nature of our convolutions, as well as the filter length, can be considered edge cases in Halide and quite far from the intended target.

6.3 Optimization Space Pruner

Once auto-tuning techniques are used, the parameter space explodes and the systematic sampling rapidly becomes impossible to achieve. The cost of finding the optimal kernel parameters and environment parameters (compiler flags, used language, ...) is rapidly prohibitive. Dedicated frameworks have been developed to address this problem. Adaptive Sampling Kit (ASK) [7] is one such tool. It reduces the number of samples needed by creating a model of the performance and by minimizing the number of samples needed to find the parameters of this model. Several models and sampling techniques are implemented.

Collective Mind [10] proposes similar techniques to solve the problem but also stores and the results and complete experimental setups in databases for future reference and reproducibility. This database approach also enables easy parallelization of the experimental process. Collective Mind also proposes collections of flags for many available compilers/versions easing the exploration process.

7 Conclusion and Future Works

In this report we presented the BOAST infrastructure as well as its application to three use cases from the Mont-Blanc project. Results are encouraging as BOAST proved to be a powerful and flexible tool that allowed gains in performance and performance portability.

Future development will focus on three goals. Interfacing with binary analysis tools like MAQAO [8] in order to build a feedback loop to guide optimization. Interfacing with search space modellers/pruners in order to optimize the search of the optimal version of a kernel. Continue working on vector code. For instance producing a collection of small to medium useful vector patterns (transposition, for instance) in BOAST could really help develop vectorized version of our algorithm.

References

[1] The message passing interface (MPI) standard, 2012. `http://www.mcs.anl.gov/research/projects/mpi/`.

[2] Chris Adeniyi-Jones. Optimal Compute on ARM Mali GPUs. `http://www.cs.bris.ac.uk/home/simonm/montblanc/OpenCL_on_Mali.pdf`.

[3] John R Allen and Ken Kennedy. Automatic loop interchange. In *ACM SIGPLAN Notices*, volume 19, pages 233–246. ACM, 1984.

[4] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. *LAPACK Users' Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, third edition, 1999.

[5] Johan Cronsioe, Brice Videau, and Vania Marangozova-Martin. BOAST: Bringing optimization through automatic source-to-source transformations. In *Embedded Multicore SoCs (MCSoC), 2013 IEEE 7th International Symposium on*, pages 129–134. IEEE, 2013.

[6] L. Dagum and R. Menon. OpenMP: an industry standard API for shared-memory programming. *Computational Science & Engineering, IEEE*, 5(1):46–55, Jan-Mar 1998.

[7] Pablo de Oliveira Castro, Eric Petit, Asma Farjallah, and William Jalby. Adaptive sampling for performance characterization of application kernels. *Concurrency and Computation: Practice and Experience*, 25(17):2345–2362, 2013.

[8] Lamia Djoudi, Denis Barthou, Patrick Carribault, Christophe Lemuet, Jean-Thomas Acquaviva, and William Jalby. Exploring application performance: a new tool for a static/-dynamic approach. In *Proceedings of the 6th LACSI Symposium*, 2005.

[9] Jack J Dongarra, Jeremy Du Croz, Sven Hammarling, and Iain S Duff. A set of level 3 basic linear algebra subprograms. *ACM Transactions on Mathematical Software (TOMS)*, 16(1):1–17, 1990.

[10] Grigori Fursin, Renato Miceli, Anton Lokhmotov, Michael Gerndt, Marc Baboulin, D. Malony, Allen, Zbigniew Chamski, Diego Novillo, and Davide Del Vento. Collective mind: Towards practical and collaborative auto-tuning. *Scientific Programming*, 22(4):309–329, July 2014.

[11] Luigi Genovese, Alexey Neelov, Stefan Goedecker, Thierry Deutsch, Seyed Alireza Ghasemi, Alexander Willand, Damien Caliste, Oded Zilberberg, Mark Rayson, Anders Bergman, et al. Daubechies wavelets as a basis set for density functional pseudopotential calculations. *The Journal of chemical physics*, 129(1):014109, 2008.

[12] Luigi Genovese, Matthieu Ospici, Thierry Deutsch, Jean-François Méhaut, Alexey Neelov, and Stefan Goedecker. Density functional theory calculation on many-cores hybrid central processing unit-graphic processing unit architectures. *The Journal of chemical physics*, 131(3):034103, 2009.

[13] Luigi Genovese, Brice Videau, Matthieu Ospici, Thierry Deutsch, Stefan Goedecker, and Jean-François Méhaut. Daubechies wavelets for high performance electronic structure calculations: the BigDFT project. In *Compte-Rendu de l'Académie des Sciences, Calcul Intensif.* Académie des Sciences, 2010.

[14] Stefan Goedecker. Rotating a three-dimensional array in an optimal position for vector processing: Case study for a three-dimensional fast Fourier transform. *Computer Physics Communications*, 76(3):294–300, 1993.

[15] Albert Hartono, Boyana Norris, and Ponnuswamy Sadayappan. Annotation-based empirical performance tuning using Orio. In *Proceedings of the 23rd IEEE International Parallel & Distributed Processing Symposium*, Rome, Italy, 2009. Also available as Preprint ANL/MCS-P1556-1008.

[16] Paul Hudak. Building domain-specific embedded languages. *ACM Computing Surveys (CSUR)*, 28(4es), 1996.

[17] Khronos OpenCL consortium. OpenCL: Open Computing Language. `http://www.khronos.org/opencl/`.

[18] Dimitri Komatitsch. Fluid-solid coupling on a cluster of GPU graphics cards for seismic wave propagation. *Comptes Rendus Mécanique*, 339(2):125–135, 2011.

[19] 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.

[20] Yukio Matsumoto and K Ishituka. Ruby programming language, 2002.

[21] P.J. Mucci, S. Browne, C. Deane, and G. Ho. PAPI: A Portable Interface to Hardware Performance Counters. In *Proc. Dept. of Defense HPCMP Users Group Conference*, pages 7–10. Citeseer, 1999.

[22] NVIDIA. NVIDIA Compute Unified Device Architecture, 2011. `http://www.nvidia.com/object/cuda_home_new.html`.

[23] Markus Püschel, José MF Moura, Bryan Singer, Jianxin Xiong, Jeremy Johnson, David Padua, Manuela Veloso, and Robert W Johnson. SPIRAL: A generator for platform-adapted libraries of signal processing algorithms. *International Journal of High Performance Computing Applications*, 18(1):21–45, 2004.

[24] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. *ACM SIGPLAN Notices*, 48(6):519–530, 2013.

[25] Mostafa I Soliman. Performance evaluation of multi-core Intel Xeon processors on basic linear algebra subprograms. *Parallel Processing Letters*, 19(01):159–174, 2009.

[26] Daniele G Spampinato and Markus Püschel. A basic linear algebra compiler. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, page 23. ACM, 2014.

[27] Brice Videau, Vania Marangozova-Martin, Luigi Genovese, and Thierry Deutsch. Optimizing 3D convolutions for wavelet transforms on CPUs with SSE units and GPUs. In *Euro-Par 2013 Parallel Processing*, pages 826–837. Springer, 2013.

[28] R. Clint Whaley and Antoine Petitet. Minimizing development and maintenance costs in supporting persistently optimized BLAS. *Software: Practice and Experience*, 35(2):101–121, February 2005.

[29] Michael E Wolf and Monica S Lam. A loop transformation theory and an algorithm to maximize parallelism. *Parallel and Distributed Systems, IEEE Transactions on*, 2(4):452–471, 1991.

[30] Michael Wolfe. Loops skewing: The wavefront method revisited. *International Journal of Parallel Programming*, 15(4):279–293, 1986.

[31] Dong Ye, Alexey Titov, Volodymyr Kindratenko, Ivan Ufimtsev, and Todd Martinez. Porting optimized GPU kernels to a multi-core CPU. In *Symposium on Application Accelerators in High-Performance Computing*, 2012.

[32] Qing Yi, Keith Seymour, Haihang You, Richard Vuduc, and Dan Quinlan. POET: Parameterized optimizations for empirical tuning. In *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, pages 1–8. IEEE, 2007.

Annexes

This section will present some simple examples to familiarize the user with BOAST. More samples can be found with source code from the program in the git repository: `https://github.com/Nanosim-LIG/boast`.

Installation

BOAST is Ruby based, so Ruby needs to be installed on the machine. Installation of boast can be done using the Ruby built-in package manager: *gem*. See Listing 17 for reference.

```
1  sudo apt-get install ruby ruby-dev
2  gem install --user-install BOAST
```

Listing 17: BOAST installation

Variable and Procedure Declaration

The following samples are presented using *irb* Ruby interactive interpreter. It can be launched using the *irb* command in a terminal. Listing 18 shows the declaration of two variables of different kind.

```
1  irb(main):001:0> require 'BOAST'
2  => true
3  irb(main):002:0> a = BOAST::Int "a"
4  => a
5  irb(main):003:0> b = BOAST::Real "b"
6  => b
7  irb(main):004:0> BOAST::decl a, b
8  integer(kind=4) :: a
9  real(kind=8) :: b
10 => [a, b]
```

Listing 18: Variable Declaration

Listing 19 shows the declaration of a procedure using the two previous variables as parameters. For clarity, *irb* echoes have been suppressed.

```
1  005:0> p = BOAST::Procedure( "test_proc", [a,b] )
2  006:0> BOAST::opn p
3  SUBROUTINE test_proc(a, b)
4    integer, parameter :: wp=kind(1.0d0)
5    integer(kind=4) :: a
6    real(kind=8) :: b
7  007:0> BOAST::close p
8  END SUBROUTINE test_proc
```

Listing 19: Procedure Declaration

Switching Language

Listing 20 shows how to switch BOAST to C. Available languages are *FORTRAN*, *C*, *CUDA* and *CL*.

```
1  008:0> BOAST::lang = BOAST::C
2  009:0> BOAST::opn p
3  void test_proc(int32_t a, double b){
4  010:0> BOAST::close p
5  }
```

Listing 20: Switching Language

Defining a Complete Procedure

Listing 21 shows how to define a procedure and the associated code. Note that here the parameters of the procedure have been associated a direction: one, *a*, is an input parameter while the other is an output parameter.

```
1  011:0> BOAST::lang = BOAST::FORTRAN
2  012:0> a = BOAST::Real( "a", :dir => :in)
3  013:0> b = BOAST::Real( "b", :dir => :out)
4  014:0> p = BOAST::Procedure( "plus_two", [a,b] ) {
5  015:1*     BOAST::pr b === a + 2
6  016:1> }
7  017:0> BOAST::pr p
8  SUBROUTINE plus_two(a, b)
9    integer, parameter :: wp=kind(1.0d0)
10   real(kind=8), intent(in) :: a
11   real(kind=8), intent(out) :: b
12   b = a + 2
13 END SUBROUTINE plus_two
14 018:0> BOAST::lang = BOAST::C
15 019:0> BOAST::pr p
16 void plus_two(const double a, double * b){
17   (*b) = a + 2;
18 }
```

Listing 21: Complete Procedure

Creating, Building and Running a Computing Kernel

Listing 22 shows how to create a Computing kernel (*CKernel*) and build it. Once a computing kernel is instantiated the output of BOAST will be redirected to the computing kernel source code. Line 4 sets the entry point of the computing kernel to the procedure we just defined. By default compilation commands are not shown unless an error occurs. This behavior can be changed by switching to verbose mode.

When running the kernel all the arguments have to be specified. Running a kernel returns a hash table containing information about the procedure execution. In this simple case two informations are returned, first the value of the output parameter *b* and second the time the kernel execution took.

```
1  020:0> BOAST::lang = BOAST::FORTRAN
2  021:0> k = BOAST::CKernel::new
3  022:0> BOAST::pr p
4  023:0> k.procedure = p
5  024:0> puts k
6  SUBROUTINE plus_two(a, b)
7    integer, parameter :: wp=kind(1.0d0)
8    real(kind=8), intent(in) :: a
9    real(kind=8), intent(out) :: b
10   b = a + 2
11 END SUBROUTINE plus_two
12 025:0> k.build
13 026:0> BOAST::verbose = true
14 027:0> k.build
15 gcc -O2 -Wall -fPIC -I/usr/lib/x86_64-linux-gnu/ruby/2.1.0 -I/usr/include/ruby-2.1.0 -I/
       usr/include/ruby-2.1.0/x86_64-linux-gnu -I/usr/include/x86_64-linux-gnu/ruby/2.1.0 -
       I/var/lib/gems/2.1.0/gems/narray-0.6.1.1 -DHAVE_NARRAY_H -c -o /tmp/
       Mod_plus_two20150309_4611_5a129k.o /tmp/Mod_plus_two20150309_4611_5a129k.c
```

```
16 gfortran -O2 -Wall -fPIC -c -o /tmp/plus_two20150309-4611-5a129k.o /tmp/plus_two20150309
       -4611-5a129k.f90
17 gcc -shared -o /tmp/Mod_plus_two20150309_4611_5a129k.so /tmp/
       Mod_plus_two20150309_4611_5a129k.o /tmp/plus_two20150309-4611-5a129k.o -Wl,-
       Bsymbolic-functions -Wl,-z,relro -rdynamic -Wl,-export-dynamic -L/usr/lib -lruby-2.1
        -lrt
18 028:0> r = k.run(5,0)
19 029:0> puts r
20 {:reference_return=>{:b=>7.0}, :duration=>5.84e-07}
```

Listing 22: Computing Kernel

Using Arrays in Procedures

Most computing kernels don't work on scalar values but rather on arrays of data. Listing 23 shows how to use arrays in computing kernels. In this case we place ourselves in BOAST namespace to reduce the syntax overhead. Variables *a* and *b* are one-dimensional arrays of size *n*. Arrays in BOAST start at index 1 unless specified otherwise. For instance `Dim(0,n-1)` would have created a dimension starting at 0. Array bounds can also be negative and several dimensions can be specified to obtain muti-dimensional arrays. For self contained procedures/kernels one can use the shortcut written on line 13 to create a CKernel object. As we are not specifying build options the build command can also be omitted and will be automatically called when running the kernel the first time. Lines 17 to 19 are used to check the result of the kernel.

```
1  001:0> require 'BOAST'
2  002:0> require 'narray'
3  003:0> include BOAST
4  004:0> n = Int( "n", :dir => :in )
5  005:0> a = Real( "a", :dir => :in,  :dim => [Dim(n)] )
6  006:0> b = Real( "b", :dir => :out, :dim => [Dim(n)] )
7  007:0> p = Procedure( "plus_two", [n, a, b] ) {
8  008:1*   decl i = Int( "i" )
9  009:1>   pr For( i, 1, n ) {
10 010:2*     pr b[i] === a[i] + 2.0
11 011:2>   }
12 012:1> }
13 013:0> k = p.ckernel
14 014:0> input  = NArray.float(1024).random
15 015:0> output = NArray.float(1024)
16 016:0> k.run(input.length, input, output)
17 017:0> (output - input).each { |val|
18 018:1*   raise "Error!" if (val-2).abs > 1e-15
19 019:1> }
20 020:0> stats = k.run(input.length, input, output)
21 021:0> puts "Success, duration: #{stats[:duration]} s"
22 Success, duration: 3.79e-06 s
```

Listing 23: Array Usage

The Canonical Case: Vector Addition

Listing 24 shows the addition of two vectors in a third one. Here BOAST is configured to have arrays starting at 0 and to use single precision reals by default (Lines 5 and 6). The kernel declaration is encapsulated inside a method to avoid cluttering the global namespace. Line 15 the expression `c[i] === a[i] + b[i]` is stored inside a variable *expr* for later use. Lines 16 to 23 show that the kernel differs depending on the target language, in CUDA and OpenCL each thread will process one element.

```
1 require 'narray'
2 require 'BOAST'
```

```
3  include BOAST
4
5  set_array_start(0)
6  set_default_real_size(4)
7
8  def vector_add
9    n = Int("n",:dir => :in)
10   a = Real("a",:dir => :in, :dim => [ Dim(n)] )
11   b = Real("b",:dir => :in, :dim => [ Dim(n)] )
12   c = Real("c",:dir => :out, :dim => [ Dim(n)] )
13   p = Procedure("vector_add", [n,a,b,c]) {
14     decl i = Int("i")
15     expr = c[i] === a[i] + b[i]
16     if (get_lang == CL or get_lang == CUDA) then
17       pr i === get_global_id(0)
18       pr expr
19     else
20       pr For(i,0,n-1) {
21         pr expr
22       }
23     end
24   }
25   return p.ckernel
26 end
```

Listing 24: Vector Addition Declaration

Listing 25 shows the a way to check the validity of the previous kernel over the available range of languages. The options that are passed to run are only relevant for GPU languages and are thus ignored in Fortran and C (Line 16). Success is only printed if results are validated, else an exception is raised (Lines 17 to 20).

```
1  n = 1024*1024
2  a = NArray.sfloat(n).random
3  b = NArray.sfloat(n).random
4  c = NArray.sfloat(n)
5
6  epsilon = 10e-15
7
8  c_ref = a + b
9
10 [:FORTRAN, :C, :CL, :CUDA].each { |l|
11   set_lang( BOAST.const_get(l) )
12   puts "#{l}:"
13   k = vector_add
14   puts k.print
15   c.random!
16   k.run(n, a, b, c, :global_work_size => [n,1,1], :local_work_size => [32,1,1])
17   diff = (c_ref - c).abs
18   diff.each { |elem|
19     raise "Warning: residue too big: #{elem}" if elem > epsilon
20   }
21 }
22 puts "Success!"
```

Listing 25: Vector Addition Test