Check for updates

Research Paper

International Journal of HIGH PERFORMANCE COMPUTING APPLICATIONS

The International Journal of High Performance Computing Applications
2018, Vol. 32(1) 28–44
© The Author(s) 2017
Reprints and permissions:
sagepub.co.uk/journalsPermissions.nav
DOI: 10.1177/1094342017718068
journals.sagepub.com/home/hpc
SAGE

BOAST: A metaprogramming framework to produce portable and efficient computing kernels for HPC applications

Brice Videau[1,2], Kevin Pouget[1,3], Luigi Genovese[3,4], Thierry Deutsch[3,4], Dimitri Komatitsch[5,6,7], Frédéric Desprez[1,8] and Jean-François Méhaut[1,3]

Abstract

The portability of real high-performance computing (HPC) applications on new platforms is an open and very delicate problem. Especially, the performance portability of the underlying computing kernels is problematic as they need to be tuned for each and every platform the application encounters. This article presents BOAST, a metaprogramming framework dedicated to computing kernels. BOAST allows the description of a kernel and its possible optimizations using a domain-specific language. BOAST runtime will then compare the different versions'performance as well as verify their exactness. BOAST is applied to three use cases: a Laplace kernel in OpenCL and two HPC applications BigDFT (electronic density computation) and SPECFEM3D (seismic and wave propagation).

Keywords

Code generation, portability, genericity, autotuning, nonregression, testing, productivity and software design, high-performance computing

1. Introduction

Porting and tuning high-performance computing (HPC) applications to new platforms is tedious and costly in terms of human resources. Many developer teams are more concentrated on solving the physical problems faster and better with new algorithms and simulating new physics. This is why optimization, while important, is a task which is sometimes neglected or hidden for many developers. In the same way, domain-specific language (DSLs) have exactly this goal, the complete separation of the high-level code dedicated to simulate new physics and the low-level one which is more dependent on the HPC architecture. Nevertheless, decoupling the high and low levels does not avoid the need of optimizations which become more and more unavoidable with new architectures based on many cores and accelerators.

Unfortunately, portability efforts of the low-level codes (i.e., computing kernels) 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.

The primary goal of the BOAST metaprogramming framework is to simplify the optimization of computing kernels for HPC applications. The main idea is to use a high-level language to define an application program interface (API) of computing kernels, that is, subroutines and their related arguments, but also describe the way to optimize it (unrolling, vectorization, parametrization, templating, and tiling). BOAST will then test automatically the many possibilities of optimizations and choose

[1]Laboratoire d'Informatique de Grenoble, Saint-Martin-d'Hères, France
[2]Centre National de la Recherche Scientifique, Paris, France
[3]Université Grenoble Alpes, Grenoble, France
[4]CEA/INAC/MEM/L_Sim, Grenoble, France
[5]Aix Marseille University, Marseille, France
[6]CNRS, Centrale Marseille, Marseille, France
[7]LMA, Marseille, France.
[8]Inria

Corresponding author:
Frédéric Desprez, Laboratoire d'Informatique de Grenoble, Inria, France.
Email: frederic.desprez@inria.fr

the best one after validation from a reference version. Then BOAST can generate pieces of Fortran, C, Open Computing Language (OpenCL), or CUDA codes, test them for a specific target architecture.

With the definition of an API, BOAST can hide the implementation details at a high level but also is a way to factorize different kernels depending only on parameters such as data types, length of filters, or boundary conditions. The maintainability and genericity of HPC codes is often limited. One of the reasons is that producing generic code in Fortran 90/95 is difficult as the language does not really fit for it. Sometimes, adding genericity or functionalities can be quite costly and may degrade performance as optimization opportunities that come from over-specification are lost. BOAST can provide a solution to handle genericity and add new functionalities at low levels.

The article is organized as follows. First, in Section 2 we further motivate the need for metaprogramming frameworks. Second, in Section 3 we illustrate the optimization problems related to scientific computing with the Laplace equation. Third, we present our solution, called BOAST, its architecture, and its functionalities in Section 4. Fourth, we apply BOAST to three use cases: the Laplace equation motivating example, short multidimensional convolutions from the BigDFT quantum chemistry application, and the seismic wave propagation simulator SPECFEM3D where 80 GPU (graphics processing unit) computing kernels have been ported improving considerably the maintainability and providing a single code base for OpenCL and CUDA kernel versions. Finally, before a conclusion and future works, we compare BOAST with other related works in Section 5.

2. Background and motivation

The motivation of the BOAST project comes from the need of the HPC communities which have full and large applications. These applications are difficult to port and maintain for emerging architectures, especially since new architectures are based on accelerators, longer vector units, or low power processors.

2.1. Evolution of HPC architectures

Evolution of HPC architectures is rapid and also diverse. In the last 7 years, no less than seven architectures have been number 1 in the Top 500:

- Sunway processor (TaihuLight, 2016)
- Intel Processor + Xeon Phi (Tianhe-2, 2013)
- AMD Processor + NVIDIA GPU (Titan, 2012)
- IBM BlueGene/Q (Sequoia, 2012)
- Fujitsu SPARC64 (K computer, 2011)
- Intel Processor + NVIDIA GPU (Tianhe-1, 2010)
- AMD Processor (Jaguar, 2009)

Being able to efficiently use those architectures on such a small period is challenging for scientists and application developers.

The race to exascale is not going to simplify the environment. All of the above machine architectures can be considered to build these supercomputers and other will appear soon. For instance, European FP7 project DEEP considers using Accelerators (XEON Phi), while the European FP7 project Mont-Blanc considers using the low-power embedded processor with integrated GPU.

Running existing applications on these 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, 6 applications were selected for porting and optimizing, 11 were selected in the Mont-Blanc project.

2.2. Scientific computing applications

Scientific computing applications are usually developed by other science researchers such as physicists, chemists, or meteorologists. 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 optimizations.

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 first available while Fortran may sometimes come later.

In Section 4, 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.

2.3. How should computing kernels be written?

The problem here is to obtain computing kernels that present good performance on the architectures encountered by the application while staying 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 nonregression 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 they must present good *performance*.

3. Basic 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 studies was a Laplace filter. 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 target architecture.

3.1. The Laplace filter

Listing 1 shows a simple implementation of the Laplace filter in C99. For the sake of 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 on the Mali architecture. This is done using the native OpenCL support for vectorization. For instance, *vload16* loads 16 consecutive elements from memory and stores it in a vector. Here, the loaded vector is of type *uchar16*, meaning that it is a vector of 16 consecutive bytes. In order to do computations without overflowing, those vectors have to be converted (using *convert_int16*) to a signed and bigger integer type. Here, they are converted to signed 4 bytes integers, leading to a vector size of 64 bytes.

Once those vector are loaded they can be used in Single Instruction Multiple Data paradigm. Operations are done component-wise on the vector. For instance, *v22 * (int16)9* multiplies each component of the vector by 9. The same thing goes for the difference operator. Once computed, the result is clamped and converted back to bytes using *convert_uchar16*.

Using vectors of 16 elements, 5 pixels (15 components) can be simultaneously loaded and can be used in computation. Here, the developer chose a pixel-based approach and thus decided to only save the 15 relevant components. These components are saved in a vector manner using the power of two decomposition of 15. The first eight components (0–7) are saved using *vstore8* then the next four (8,9,a,b) using *vstore4* and so on.

This optimization yields speedups between 1.5 and 6 depending on the image size.

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.

Listing 4. Laplace OpenCL Filter Vectorized with Synthesized Loads.

```
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 * (int16)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);
  vstore4(res_.s89ab,     0, dst + 3*width*j + 3*5*i + 8);
  vstore2(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.

```
  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.

```
  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*.

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

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 Adeniyi-Jones. Others consist in tiling the problem space or improving the loads synthesis.

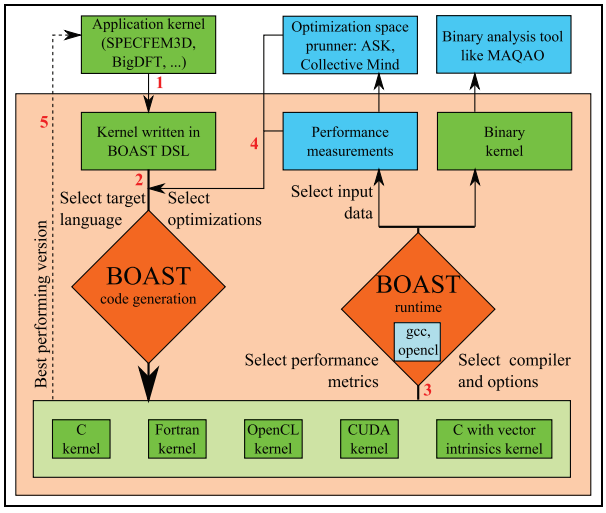

Figure 1. Structure and workflow of the BOAST framework.

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 (like the number of pixels chosen for instance). Testing the different combinations of those optimizations and the different parameters would be very costly in developer time and, moreover, like all repetitive and tedious tasks, error prone. Thus, every created kernel would have to be thoroughly tested to ensure that no error was done.

In the next section, we will present our answer to these issues: 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 (Cronsioe et al., 2013).

Figure 1 illustrates the workflow and program structure. The user starts from an application kernel (either designed or implemented), and writes it in a dedicated

language (Step 1). The language provides enough flexibility for the kernel to be metaprogrammed with several orthogonal optimizations. From this set of optimizations, possible languages targets, and compilation options, the user can design an optimization space to explore. This optimization space can contain rules to remove infeasible candidates. BOAST provides the mechanisms to specify those optimization spaces and enforce the users rules.

Once this optimization space is designed, the user selects an optimization strategy, brute force and genetic algorithms are provided by BOAST, or can design his own. BOAST will then evaluate the different candidates generated by the optimization program. The candidate's parameters define the output source code that will be generated by BOAST (Step 2). The resulting code source is then built according to the specified compiler and options (Step 3). The kernel can be benchmarked and tested for nonregression. Based on the results, other optimizations can be selected (Step 4). 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).

Of course, this workflow is not the only possible use of BOAST. Especially, during the kernel description phase the user can use the BOAST framework to test his existing optimizations and analyze the results in order to design new optimizations. He can possibly do it using external tools.

In order to achieve those results, three aspects should be considered: code description, code generation, and kernel execution runtime.

4.1. Kernel description language

Usually, computing kernels are hotspots of an HPC application, and they take most of the time based on loop nests. 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 recognize the opportunity.

There are many different loop optimization techniques (Wolf and Lam, 1991). We can cite loop skewing (Wolfe, 1986; derives nested loops wavefronts) or loop interchange (Allen and Kennedy, 1984; loop variables change places). The importance of correct loop imbrication on Basic Linear Algebra Subprograms (BLAS; Lawson et al., 1979) operations is studied by Soliman (2009), and shows performance increase of a factor up to 5 when using correct loop imbrication. The importance of code transformation is stressed in Ye et al. (2012), where a selection of GPU kernels are ported to central processing unit (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, etc.).
- Procedures have to be abstracted (reunite Fortran and C like languages, attributes, etc.).
- Functions must be available.
- Variables, constants and functions should be able to be composed in complex expressions.
- Basic control structures (for, while, if/else, etc.) have to be abstracted.
- Powerful array management features (allowing several dimensions, transformations, indexing, etc.).

In order to manipulate those abstractions, we want to have a syntax similar to 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 (EDSL) approach was selected (Hudak, 1996). This feature allows for the coexistence of two languages: the host language and the DSL. In our case, DSL allows the description of the kernel ($c = a + b$), while the host language provides the metaprogramming of the kernel ($c \leftarrow a + b$). Operator overloading of the host language will provide the familiar syntax programmers are accustomed to.

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 (Matsumoto and Ishituka, 2002) is one such language. It has deep introspection capabilities as well. This is the main reason why it was selected for BOAST.

One of the added advantages of using a high-level scripting language like Ruby as the host language is its interfacing capabilities, providing an easy way to use the libraries needed during the development of the framework.

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.

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

Listing 6. BOAST code.

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

Listing 7. Fortran output.

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

Listing 8. C output.

Figure 2. BOAST code snippet for variables and expressions.

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

4.1.3. 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 is 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 Fortran (Listing 7) and C (Listing 8) output. First, we define two 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.

Listing 6. BOAST code.
Listing 7. Fortran output.
Listing 8. C output.

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

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

```
BOAST::register_funccall("modulo")
i = BOAST::Int("i")
j = BOAST::Int("j")
BOAST::pr j === 0
BOAST::pr BOAST::For( i, 0, 100 ) {
  BOAST::pr BOAST::If( BOAST::modulo(i,7) == 0 ) {
    BOAST::pr j === j + 1
  }
}
```

Listing 9. BOAST code.

```
j=0
do i=0, 100, 1
  if(modulo(i, 7)==0) then
    j=j+1
  end if
end do
```

Listing 10. Fortran output.

```
j=0;
for(i=0; i<=100; i+=1){
  if(modulo(i, 7)==0){
    j=j+1;
  }
}
```

Listing 11. C output.

Figure 3. BOAST code snippet for control structures.

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.

4.1.4. 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 that matches more closely 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 0–100 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.

Listing 9. BOAST code.
Listing 10. Fortran output.
Listing 11. C output.

The last control structure is Procedure. It describes either procedures or functions. Code in 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, both composed of double precision numbers. Those arrays have two dimensions which depend on input variables n and

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

Listing 12. BOAST code.

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

Listing 13. Fortran output.

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

Listing 14. C output.

Figure 4. BOAST code snippet for procedure.

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 the BOAST namespace.

Listing 12. BOAST code.
Listing 13. Fortran output.
Listing 14. C output.

The generated outputs in Fortran (Listing 13) and C (Listing 14) show the difference in metainformation that is kept between both versions.

4.2. BOAST runtime

In the previous section, we presented the BOAST's language. This allows us to describe procedures and

functions and to metaprogram 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 runtime that allow this automation. Those aspects are multitarget language generation (*performance, portability*), kernel compilation (*productivity, performance*), kernel execution (*productivity, performance*), and last, kernel tracer, dumper and replay for nonregression tests (*productivity*).

4.2.1. Multitarget 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), or languages that support multithreading paradigms (OpenMP). 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 without the need to fork another process. 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 nonregression 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 runtime of the kernel and it is obtained using the system-wide real-time clock. Other probes can be inserted at compiletime, if needed. BOAST also supports PAPI (Mucci et al., 1999) to capture hardware performance counters during each kernel execution.

4.2.4. Kernel replay. The nonregression 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, because 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 nonregression 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. Nonregression 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 multitarget 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 casestudy of the port of a CUDA application to OpenCL is presented in Section 5.3.

Listing 15. GPUTrace sample trace.

The idea behind this tool is based on the assumption that the different GPU ports of the code should perform the very same operations, at least at the logical level (the APIs will have *implementation* differences, but they should offer nonetheless same functionalities).

```
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.

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 bystrace andltrace 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 SPECFEM3D tracing.

However, in contrast withstrace andltrace, 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.

Once two call traces have been generated by GPU Trace, the user can compare them with a graphicaldiff 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 to generate *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 nonregression testing.

5. BOAST use cases

In this section, we will present the benefits of using BOAST on the Laplace motivating example as well as on two scientific applications. The first one, BigDFT (Genovese et al., 2008), uses BOAST in order to develop new functionalities with performance portability in mind. The second one, SPECFEM3D (Komatitsch, 2011), uses BOAST to factorize OpenCL and CUDA development, while having robust nonregression 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 any 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*

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.

Table 1. Best performance of ARM Laplace kernel.

Image size	Naive (s)	Best (s)	Accel.	BOAST (s)	Accel.
768 × 432	0.0107	0.00669	×1.6	0.000639	×16.7
2560×1600	0.0850	0.0137	×6.2	0.00687	×12.4
2048×2048	0.0865	0.0149	×5.8	0.00715	×12.1
5760×3240	0.382	0.0449	×8.5	0.0325	×11.8
7680×4320	0.680	0.0747	×9.1	0.0581	×11.7

Table 2. Best performance of Laplace kernel on several architectures.

Image size	ARM	Intel	Ratio	NV	Ratio
768×432	0.000639	0.000222	×2.9	0.0000715	×8.9
2560×1600	0.00687	0.00222	×3.1	0.000782	×8.8
2048×2048	0.00715	0.00226	×3.2	0.000799	×8.9
5760×3240	0.0325	0.0108	×3.0	0.00351	×9.3
7680×4320	0.0581	0.0192	×3.0	0.00623	×9.3

5.1.2. Performance results. Table 1 shows the best results obtained by ARM on different images compared to the naive implementation and to the best 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 increases, thus decreasing task granularity as the global work size increases. This result is interesting because its unexpected nature strongly backs the use of autotuning. 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 target architecture. But BOAST simplifies and speeds up the identification of the correct combination at the cost of metaprogramming. In the next subsections, we will show that our methodology also applies to real applications.

5.2. Creating an autotuned convolution library for BigDFT using BOAST

In 2005, the EU FP6-STREP-NEST BigDFT (Genovese et al., 2008) 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-year 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 one 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 one to optimize the number of degrees of freedom for a required accuracy (Genovese et al., 2008), which is highly desirable given the complexity and inhomogeneity of the systems under investigation nowadays.

Despite that the application is mainly written in Fortran (360 kLOC of Fortran), it currently also includes 70 kLOC of C languages, accounting for more than 50% of the code base. It is a parallel application based on the standards MPI (2012) and OpenMP (Dagum and Menon, 1998). It also supports CUDA (NVIDIA, 2011) and OpenCL. 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 (Genovese et al., 2009).

5.2.1. BigDFT. In the Kohn-Sham formulation of Density Functional Theory (DFT), the electrons are associated to wavefunctions (orbitals), which are represented by 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 article and can be found in the BigDFT reference paper (Genovese et al., 2008). Convolutions are basic operations of lots of scientific application codes, for example, finite differences approaches, which are universally used in computational physics.

The CPU convolutions of BigDFT have thus been thoroughly optimized. In a recent paper (Videau et al., 2013), the optimization of the CPU convolutions of BigDFT has been extensively considered. One example of a specific convolution, called MagicFilter (Genovese et al., 2010), 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 (Goedecker 1993).

```
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 maintenance and 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.

Therefore, it appears 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 (e.g., 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 is beyond the scope of this paper. 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 composed 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.

The interest in having a robust and optimally tuned library in this scientific field is therefore evident. Techniques are under investigation to also provide end users with fine-tuned binaries rather than the source codes, such that more aggressive interprocedural 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.

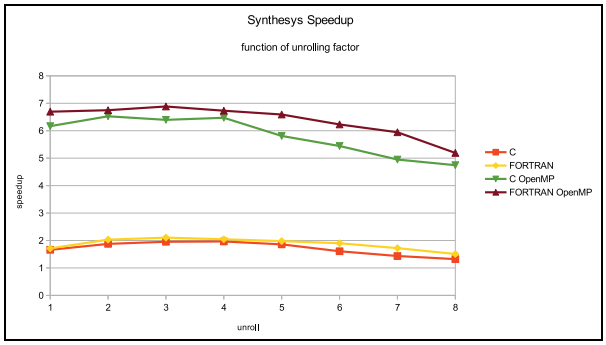

Figure 5. Impact of unrolling, language used and OpenMP on a wavelet transform code.

This experience therefore would be a first step toward the release of a tunable optimized convolution library oriented to computational physics communities.

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 application kernels: From CUDA to OpenCL using BOAST

In this last subsection, we present an alternative use case of the BOAST framework, focused on portability. As part of the Mont-Blanc project, we ported a scientific application so that it could be use to benchmark the Mont-Blanc HPC cluster. It was composed of 40 complex and hand-tuned CUDA kernels, that we ported to the OpenCL framework. This scenario also highlights the factorization and maintainability improvements of the BOAST framework, as we obtained a single BOAST implementation of the kernels (instead of two, CUDA and OpenCL). This version of the code is still used and updated in the upstream project.

The challenges we faced in this case study were more related to the complexity of the scientific application and its kernels than to the BOAST framework.

5.3.1. Specfem3D. SPECFEM3D GLOBE[1] is a free seismic wave propagation simulator. It simulates seismic wave propagation at the local or regional scale based upon spectral element method, with very good accuracy and convergence properties. It is a reference application for supercomputer benchmarking, thanks to its good scaling capabilities.

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 SPECFEM3D code base is written in Fortran 2003 and 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 performed very specific physical transformations.

Because of the complexity of the wave propagation kernels, it was impossible for us to understand the kernel's source. Hence, we treated them as black-boxes, and aimed at obtaining identical performances. Furthermore, the kernels' complexity prevents the specification of unit tests, thus the application is validated by the accuracy of the results it produces (the seismograms of the simulated earthquakes), against the actual ones. Nonregression testing (after new developments) is also based on these seismograms, with measurements of the relative error between two identical simulations.

5.3.2. Porting to OpenCL. (a) *Porting kernels to BOAST*: NVidia CUDA and OpenCL are based on the same programming model: a massively parallel accelerator running in disjoint, nonaddressable memory environment. Thanks to that proximity, we have been able to carry out most of the porting task with only a limited knowledge about SPECFEM3D internal physics.

This lack of SPECFEM3D 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 SPECFEM3D's CUDA kernels into BOAST EDSL. This way, we could ask the BOAST framework to generate a CUDA version of the kernels, plug them back into SPECFEM3D and get (after fixing compile-time errors—prototypes and naming mistakes mainly) a first set of SPECFEM3D 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 SPECFEM3D 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.

(b) *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. 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 SPECFEM3D'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, clEnqueueWrite Buffer, 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.

(c) *Debugging OpenCL Execution: GPU Trace*: With the help of GPU Trace (described in Section 4.3), we could confront CUDA and OpenCL execution traces with a graphicaldiff 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.

(d) *Evaluation*: Our OpenCL/BOAST port of SPECFEM3D is now merged in SPECFEM3D's development tree and under test and extension by different research teams. On a platform with two K40x GPUs and 24 Intel Xeon processors, we measured identical performance between the original CUDA version and our BOAST-CUDA version. This result was expected, as the BOAST-generated version is identical to the original, except from naming differences.

With the same set of optimization flags, BOAST CUDA and OpenCL versions 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 the 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.

Thanks to the porting of the SPECFEM3D GPU kernels to BOAST EDSL, the size of the kernels' source code shrank by a factor of 1.8 (from 7500 to less than 4000 LOC, mainly because of the removal of code duplication and manually unrolled loops). This is beneficial for SPECFEM3D as it improves the readability and maintainability of its source code. This gain was confirmed by the SPECFEM3D community as the subsequent developments have been carried out on the BOAST version of the kernels, and not on the CUDA or OpenCL generated code.

We have also been able to enhance SPECFEM3D's nonregression test-suite by adding per-kernel

nonregression 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 nonregression testing, the 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 nonregression.

In the same mindset, we provided SPECFEM3D test-suite with kernel performance evaluation mechanisms. These tests will help developers to try new optimizations in kernels' code and measure their impact, without executing the whole application.

6. Related work

Code generation and autotuning 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 into three parts, focusing first on autotuning frameworks. Tools that provide a DSL to describe computing kernels will then be presented. Last, optimization space pruners and their ties to autotuning will be introduced.

6.1. Application autotuning

The most convenient way to obtain an application that can be autotuned on a given platform is to base this application on a widely used computing library. BLAS (Dongarra et al., 1990) and LAPACK (Anderson et al., 1999) are such libraries. These libraries are either hand tuned for selected platforms or have autotuned implementations. Atlas (Whaley and Petitet,2005) is an autotuned implementation of BLAS/LAPACK. ATLAS authors defined the *Automated Empirical Optimization of Software* methodology that we implemented with BOAST. Their kernel generation is performed 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 (Hartono et al., 2009) is an autotuning framework that has an approach close of that of BOAST but they are based on an annotated DSL describing loop transformations rather than a more generic EDSL. Halide (Ragan-Kelley et al., 2013) is an autotuning 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 (Spampinato and Püschel, 2015) 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 (Püschel et al., 2004) is a decade old generation framework for signal processing. It uses a proprietary DSL called SPL (Signal Processing Language), to describe a DSP algorithm. This DSL is then transformed into efficient programs in high-level languages such asC or Fortran. POET (Yi et al., 2007) also uses a DSL to describe custom code transformations, such asloop unrolling, loop blocking, and loop interchange. Those transformations can be parametrized in order to tune the application. Orio (Hartono et al., 2009) can be compared to POET as it aims at describing possible code transformations using a DSL. All those approaches are very different from ours as they put the emphasis on compilation techniques whereas BOAST relies on the user to express the different optimizations.

Halide (Ragan-Kelley et al., 2013) is closer in some ways to our approach as it uses an embedded C++ DSL to describe the 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 four to five 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.

BEAST (Anzt et al., 2015) is an autotuning framework that uses macro-processing in source files. Inside the macros, BEAST offers a DSL that can be used to specify values derived from a set of iterators. Those iterators explore the search space and can be constrained. Great care has been used to ensure that the iterators enumeration as well as the pruning are as efficient as possible.

Heterogeneous Programming Library (HPL; Viñas et al., 2013)is may be the most similar to BOAST. It is an autotuning kernel library in C++ that targets OpenCL. It uses an EDSL in C++ to describe generic computing kernels. Like when using BOAST, optimization is left to the users and written directly using the EDSL. The syntax is very close to real C++ with the classical control structures having a trailing underscore.

But whereas BOAST re-interprets the same code while changing the environment, in HPL each expression is stored in a tree and the tree is reevaluated in a different environment.

6.3. Optimization space pruners

Once autotuning 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, etc.) is rapidly prohibitive. Dedicated frameworks have been developed to address this problem. Adaptive Sampling Kit (ASK; de Oliveira Castro et al., 2013) 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 (Fursin et al., 2014) 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

Application portability is an important issue that should be solved efficiently, especially given the large number of different processors now available for today's supercomputers. The work needed to get performance portability is a tedious task, even for experienced programmers. The availability of semiautomatic tools is therefore mandatory for the development of large simulation applications. Computing kernels' identification and optimization has to be carefully performed as they usually consume most of the computing resources.

In this article, we presented the BOAST infrastructure (DLS and runtime) that aims at describing kernels in a high-level language and allows the comparison of the performance of different versions of the code in a simple and seamless way. We described 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 compared to hand-tuned codes. Performance portability of those codes is also improved.

Future development will focus on three main goals. First, we find interesting to interface it with binary analysis tools like MAQAO (Djoudi et al., 2005) in order to build a feedback loop to guide optimization. On the autotuning side, interfacing with search space modellers/pruners in order to optimize the search of the optimal version of a kernel will allow us to gain some time in the optimization process. Finally, work should also continue on improving the support for vector codes. For instance, producing a collection of small to medium useful vector patterns (transposition, for instance) in BOAST could really help users develop vectorized version of their algorithm. Other computing kernels and applications will also be ported on various architectures.

Acknowledgements

The research leading to these results has received funding from the European Community's Seventh Framework Programme [FP7/2007-2013] under grant agreements 288777 and 610402.

Declaration of Conflicting Interests

The author(s) declared no potential conflicts of interest with respect to the research, authorship, and/or publication of this paper.

Funding

The author(s) received no financial support for the research, authorship, and/or publication of this paper.

Note

1. SPECFEM3D GLOBE—CIG http://www.geodynamics.org/cig/software/specfem3d-globe.

References

Adeniyi-Jones C. Optimal Compute on ARM Mali GPUs. Available at: http://www.cs.bris.ac.uk/home/simonm/montblanc/OpenCL on Mali.pdf (accessed 1 December 2016).

Allen JR and Kennedy K (1984, June) Automatic loop interchange. In: *ACM SIGPLAN Notices*, vol. 19(6), pp. 233–246. Montreal, Canada: ACM.

Anderson E, Bai Z, Bischof C, et al. (1999) *LAPACK Users' Guide*, 3rd ed. Philadelphia: Society for Industrial and Applied Mathematics.

Anzt H, Haugen B, Kurzak J, et al. (2015) Experiences in autotuning matrix multiplication for energy minimization on GPUs. *Concurrency and Computation: Practice and Experience* 27(17): 5096–5113. DOI:10.1002/cpe.3516.

Cronsioe J, Videau B and Marangozova-Martin V (2013) BOAST: bringing optimization through automatic source-to-source transformations. In: Tomohiro Y (ed) *2013 IEEE 7th International Symposium on Embedded Multicore SoCs (MCSoC)*, Tokyo, Japan, September, pp. 129–134. IEEE.

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

de Oliveira Castro P, Petit E, Farjallah A, et al. (2013) Adaptive sampling for performance characterization of application kernels. *Concurrency and Computation: Practice and Experience* 25(17): 2345–2362.

Djoudi L, Barthou D, Carribault P, et al. (2005) Exploring application performance: a new tool for a static/dynamic approach. In: *Proceedings of the 6th LACSI Symposium*, October 2005.

Dongarra JJ, Du Croz J, Hammarling S, et al. (1990) A set of level 3 basic linear algebra subprograms. *ACM Transactions on Mathematical Software (TOMS)* 16(1): 1–17.

Fursin G, Miceli R, Lokhmotov A, et al. (2014) Collective Mind: towards practical and collaborative auto-tuning. *Scientific Programming* 22(4): 309–329. Available at: https://hal.inria.fr/hal-01054763.

Genovese L, Neelov A, Goedecker S, et al. (2008) Daubechies wavelets as a basis set for density functional pseudopotential calculations. *The Journal of Chemical Physics* 129(1): 2008.

Genovese L, Ospici M, Deutsch T, et al. (2009) Density functional theory calculation on many-cores hybrid central processing unit-graphic processing unit architectures. *The Journal of Chemical Physics* 131(3): 2009.

Genovese L, Videau B, Ospici M, et al. (2010) Daubechies wavelets for high performance electronic structure calculations: the BigDFT project. In: Catherine B (ed) *Compte-Rendu de l'Académie des Sciences, Calcul Intensif*. France: Académie des Sciences.

Goedecker S (1993) 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.

Hartono A, Norris B and Sadayappan P (2009) Annotation-based empirical performance tuning using orio. In: *Proceedings of the 23rd IEEE International Parallel & Distributed Processing Symposium*, Rome, Italy, Preprint ANL/MCS-P1556-1008. Available at: http://www.mcs.anl.gov/uploads/cels/papers/P1556.pdf

Hudak P (1996) Building domain-specific embedded languages. *ACM Computing Surveys* 28(4): 1–6.

Khronos OpenCL consortium. OpenCL: Open Computing Language. Available at: http://www.khronos.org/opencl/ (accessed 1 December 2016).

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

Lawson CL, Hanson RJ, Kincaid DR, et al. (1979) Basic linear algebra subprograms for FORTRAN usage. *ACM Transactions on Mathematical Software (TOMS)* 5(3): 308–323.

Matsumoto Y and Ishituka K (2002) *Ruby Programming Language*. Boston, Massachusetts: Addison Wesley Publishing Company.

MPI (2012) The Message Passing Interface (MPI) Standard. Available at: http://www.mcs.anl.gov/research/projects/mpi/ (accessed 1 December 2016).

Mucci P, Browne S, Deane C, et al. (1999) PAPI: a portable interface to hardware performance counters. In: *Proceeding Dept of Defense HPCMP Users Group Conference*, pp. 7–10. Citeseer.

NVIDIA (2011) NVIDIA Compute Unified Device Architecture. Available at: http://www.nvidia.com/object/cuda home new.html (accessed 1 December 2016).

Püschel M, Moura JM, Singer B, et al. (2004) SPIRAL: a generator for platform-adapted libraries of signal processing algorithms. *International Journal of High Performance Computing Applications* 18(1): 21–45.

Ragan-Kelley J, Barnes C, Adams A, et al. (2013) Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. *ACM SIGPLAN Notices* 48(6): 519–530.

Soliman MI (2009) Performance evaluation of multi-core intel xeon processors on basic linear algebra subprograms. *Parallel Processing Letters* 19(01): 159–174.

Spampinato DG and Püschel M (2014) A basic linear algebra compiler. In: David RK and Tipp M (eds) *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, Orlando, FL, USA, February 2014, p. 23. ACM.

Top500.Org. Top500. Available at: http://www.top500.org (accessed 1 December 2016).

Videau B, Marangozova-Martin V, Genovese L, et al. (2013) Optimizing 3D convolutions for wavelet transforms on CPUs with SSE units and GPUs. In: *Euro-Par 2013 Parallel Processing*, Aachen, Germany, August 2013, pp. 826–837. Springer.

Viñas M, Bozkus Z and Fraguela BB (2013) Exploiting heterogeneous parallelism with the heterogeneous programming library. *Journal of Parallel and Distributed Computing* 73(12): 1627–1638.

Whaley RC and Petitet A (2005) Minimizing development and maintenance costs in supporting persistently optimized BLAS. *Software: Practice and Experience* 35(2): 101–121.

Wolf ME and Lam MS (1991) A loop transformation theory and an algorithm to maximize parallelism. *IEEE Transactions on Parallel and Distributed Systems* 2(4): 452–471.

Wolfe M (1986) Loops skewing: the wavefront method revisited. *International Journal of Parallel Programming* 15(4): 279–293.

Ye D, Titov A, Kindratenko V, et al. (2012) Porting optimized GPU kernels to a multi-core CPU. In: Gregory P (ed) *Symposium on Application Accelerators in High-Performance Computing*, Knoxville, TN, USA, September 2012, IEEE.

Yi Q, Seymour K, You H, et al. (2007) POET: Parameterized optimizations for empirical tuning. In: Timothy MP (ed) *Parallel and Distributed Processing Symposium 2007. IPDPS 2007*, Long Beach, CA, USA, March 2007, IEEE International, pp. 1–8. IEEE.

Author biographies

Brice Videau received the M.Eng. degree in chemistry from ENSCM, Montpellier, France, in 2003, the M.Eng. degree in computer science from ENSIMAG, Grenoble, France, in 2004, and the Ph.D. degree in computer science from UJF Grenoble 1, Grenoble, in 2009. Since 2010, he has been a Postdoctoral Fellow with the L_Sim Laboratory, INAC, CEA and with the CNRS at Grenoble. He is working on code generation, optimization and auto-tuning for High Performance

Computing and does so in several European projects (Mont-BLanc, HPC4E, EoCoE).

Kevin Pouget received his Computer Science Ph.D. degree from the University of Grenoble in 2014, for his work on interactive debugging for multicore and embedded systems. He now pursues his research work at the University as a Post-doctoral fellow, with a focus on OpenMP debugging. Through his experience, he got an advanced knowledge of the low-level aspects of programming languages, runtime libraries and their interactions with the operating systems.

Luigi Genovese is a Computational Physicist in the domain of Material Sciences, with education in Theoretical High Energy Physics. He is presently Researcher at the Laboratoire de Simulation Atomistique in CEA Grenoble, His research interests are related to the conception, development, and implementation of new theoretical algorithms and methods exploiting state-of-the-art computing resources, enabling large-scale computations in diverse areas of Solid-State physics, Quantum Chemistry and Electronic Structure calculations.

Thierry Deutsch is a Research Director at the institute Nanosciences and Cryogeny (INAC, CEA) in Grenoble. He is a specialist of high performance computing in the field of solid state physics. He has developed some softwares as BigDFT and CPMD based on Schrodinger equations and the Density Functional Theory to calculate the electronic structure of materials or molecules.

Frédéric Desprez is a Chief Senior Research Scientist at Inria and holds a position at the LIG laboratory (UGA, Grenoble, France) in the Corse research team. He is also Deputy Scientific Director at Inria. He received his PhD in C.S. from Institut National Polytechnique de Grenoble, France, in 1994 and his MS in C.S. from ENS Lyon in 1990. His research interests include parallel high performance computing algorithms and scheduling for large scale distributed platforms. He leads the Grid'5000 project, which offers a platform to evaluate large scale algorithms, applications, and middleware systems. See https://fdesprez.github.io/ for further information.

Jean-Francois Mehaut is professor of Computer Science at Universite Grenoble Alpes (UGA). His fields of interest in research are high performance computing, runtime systems and debugging tools. In particular, Jean-Francois Mehaut is interested in alternative and low power architectures to build the future exascale platforms. He was involved during 6 years in the European Mont-Blanc projects. Jean-Francois Mehaut has also several scientific collaborations with Brazilian Universities (UFRGS, USP, PUC, UFSC, LNCC). These collaborations are developed and funded by several scientific projects (H2020 Europe Brazil, Capes, CNPq).