BOAST

Performance Portability Using Meta-Programming and Auto-Tuning

Brice Videau [1], Kevin Pouget [1], Luigi Genovese [2], Thierry Deutsch [2], Dimitri Komatitsch [3], Jean-François Méhaut [1]

[1]INRIA - Corse, [2]CEA - L_Sim, [3]CNRS

Workshop HOSCAR
September 24, 2015

Scientific Application Portability

Limited Portability

- Huge codes (more than 100 000 lines), Written in FORTRAN or C++
- Collaborative efforts
- Use many different programing paradigms

But Based on Computing Kernels

- Well defines part of a program
- Compute intensive
- Prime target for optimization

Kernels Should Be Written:

- In a portable manner
- In a way that raises developer productivity
- To present good performance

HPC Architecture Evolution

Very Rapid and Diverse, 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)

Tomorrow?

- ARM + DSP?
- Intel Atom + FPGA?
- Quantum computing?

How to write kernels that could adapt to those architectures?
(well maybe not quantum computing...)

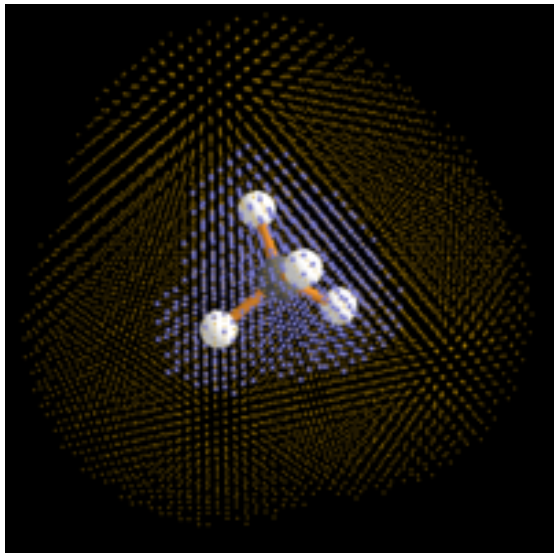
BigDFT a Tool for Nanotechnologies

Ab initio simulation:

- Simulates the properties of crystals and molecules,
- Computes the electronic density,
- Based on Daubechie wavelet.

The formalism was chosen because it is fit for HPC computations:

- Each orbital can be treated independently most of the time,
- Operator on orbitals are simple and straightforward.



Electronic density around a methane molecule.

3D convolutions

Operators can be expressed as 3D convolutions :

- Wavelet Transform
- Potential Energy
- Kinetic Energy

These convolutions are separable and filter are short (around 16 elements).
Can take up to 80% of the computation time on some systems.

Case Study: the MagicFilter

The simplest convolution found in BigDFT, corresponds to the potential operator.

Characteristics

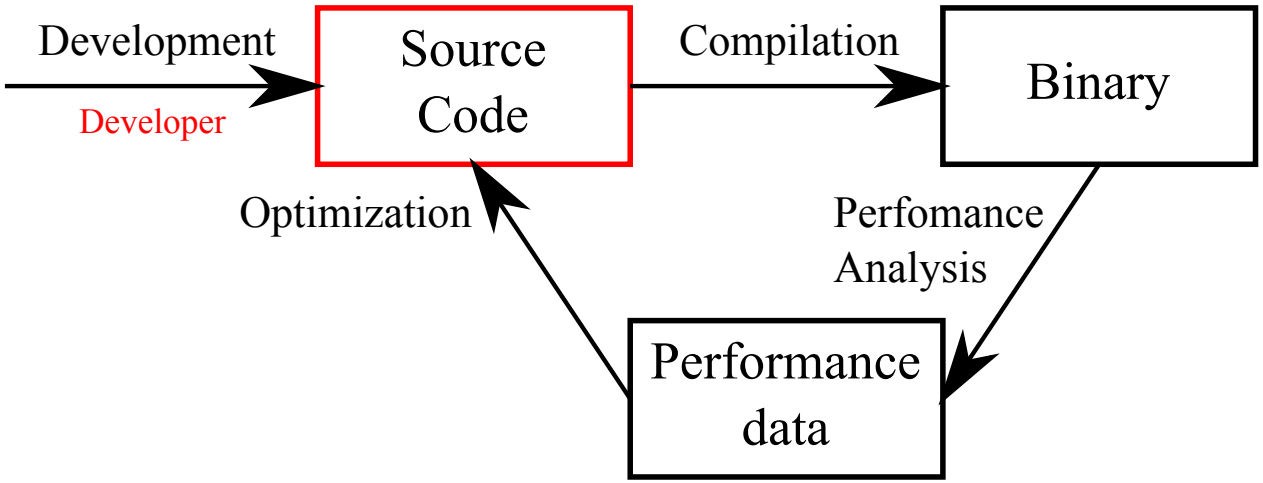
- Separable,
- Filter length 16,
- Transposition,
- Periodic,
- Only 32 operations per element.

Pseudo code

```
1  double filt[16] = {F0, F1, ... , F15};
2  void magicfilter(int n, int ndat,
3                   double *in, double *out){
4    double temp;
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          temp+= in[ ((i-7+k)%n) + j*n]
10                  * filt[k];
11       }
12       out[j + i*ndat] = temp;
13 } } }
```

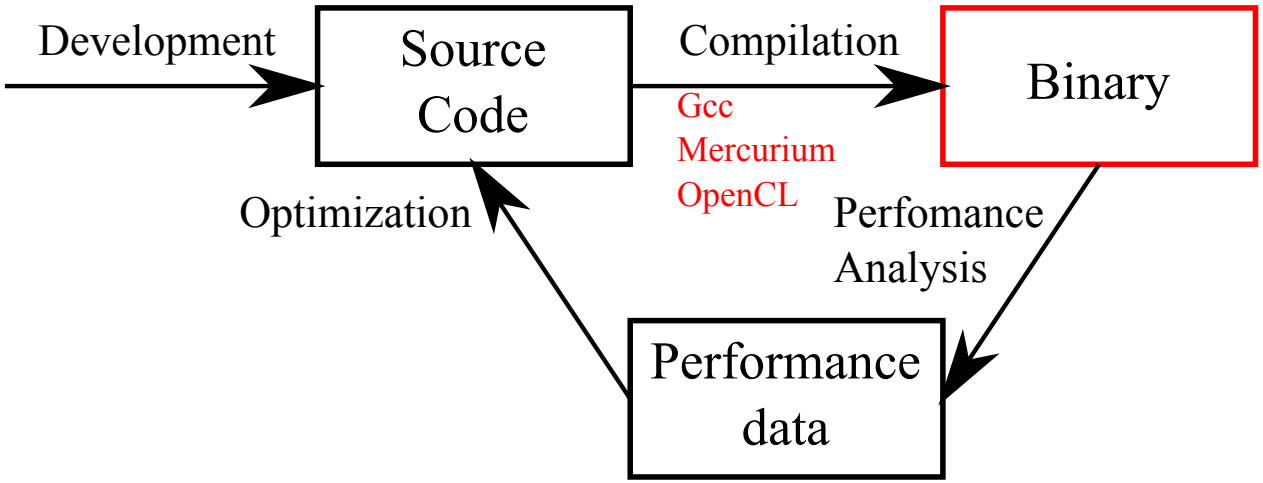
Talk Outline

Classical Tuning of Computing Kernels



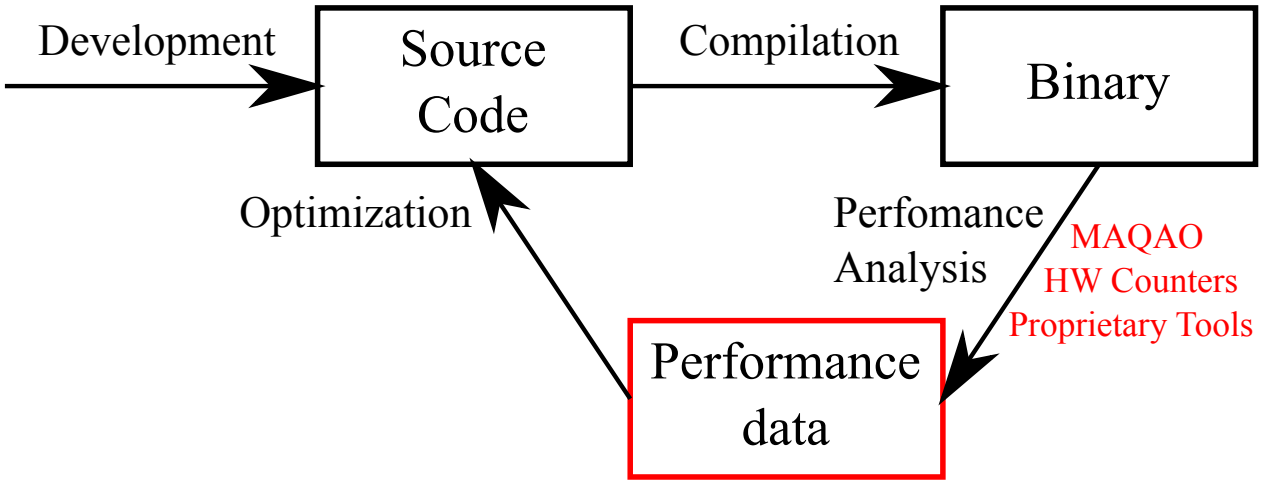
- Kernel optimization workflow
- Usually performed by a knowledgeable developer

Classical Tuning of Computing Kernels



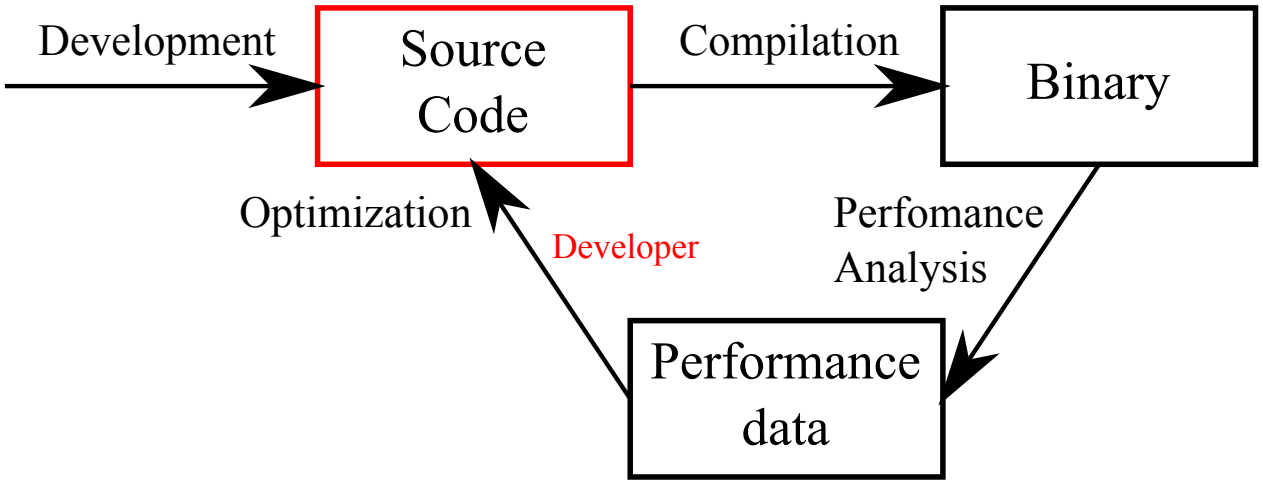
- Compilers perform optimizations
- Architecture specific or generic optimizations

Classical Tuning of Computing Kernels



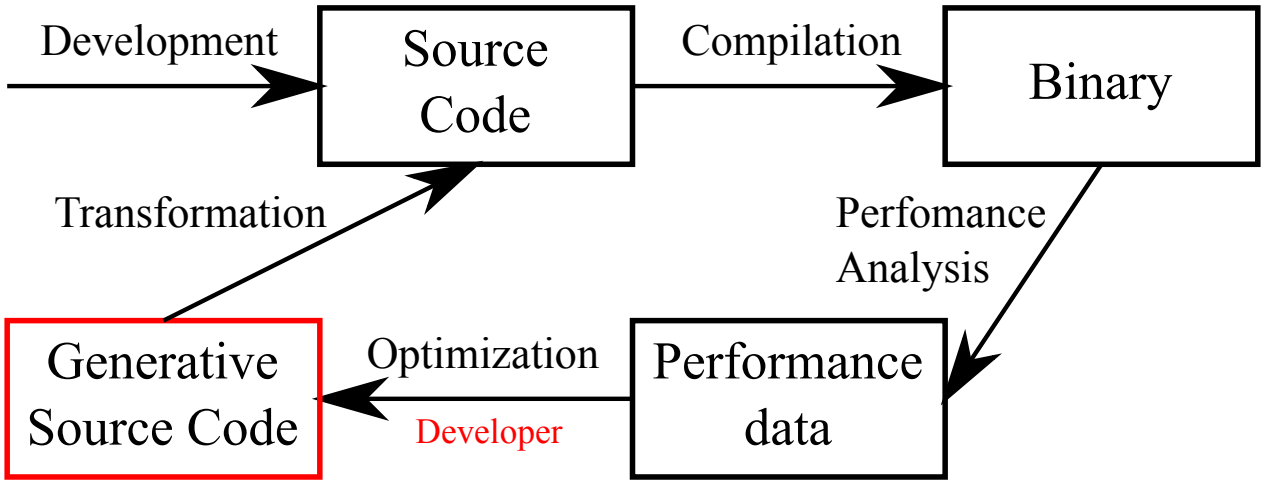
- Performance data hint at source transformations
- Architecture specific or generic hints

Classical Tuning of Computing Kernels



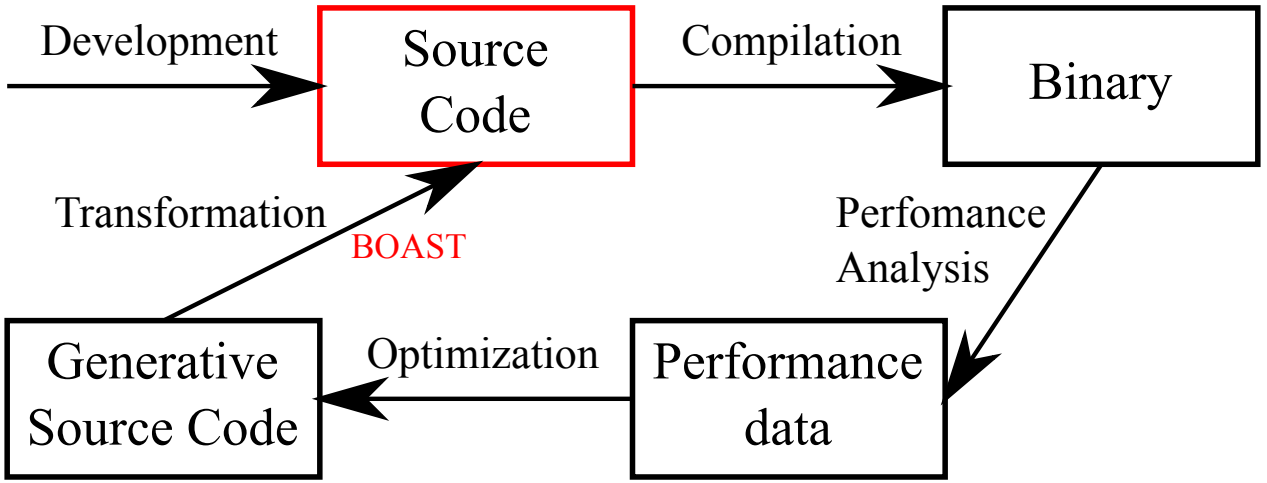
- Multiplication of kernel versions or loss of versions
- Difficulty to benchmark versions against each-other

BOAST Workflow



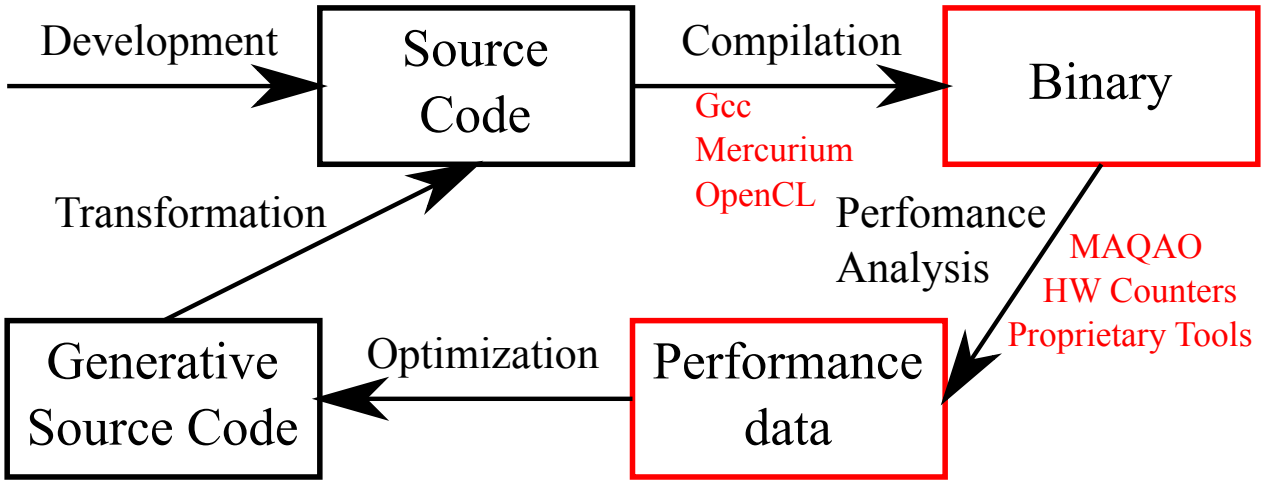
- Meta-programming of optimizations in BOAST
- High level object oriented language

BOAST Workflow



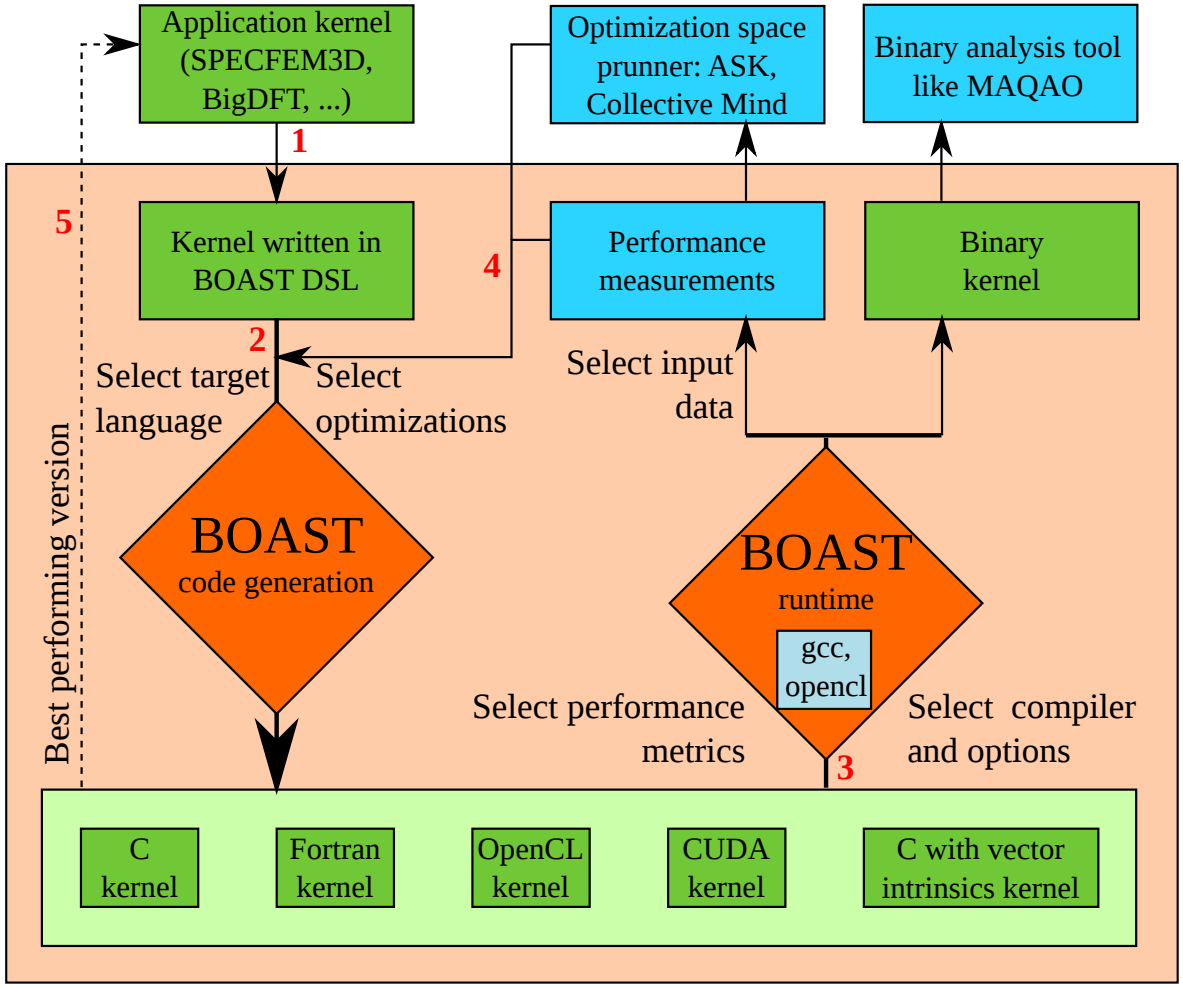
- Generate combination of optimizations
- C, OpenCL, FORTRAN and CUDA are supported

BOAST Workflow



- Compilation and analysis are automated
- Selection of best version can also be automated

BOAST Architecture



Back to the use case: BigDFT

Parameters arising in a convolution:

- Filter: length, values, center.
- Direction: forward or inverse convolution.
- Boundary conditions: free or periodic.
- Unroll factor: arbitrary.

How are those parameters constraining our tool?

Features required

Unroll factor:

- Create an arbitrary number of temporary variables,
- Create loops with variable steps.

Boundary conditions:

- Manage arrays with parametrized size.

Filter and convolution direction:

- Transform arrays.

And of course be able to describe convolutions and output them in different languages.

Proposed Generator

Idea: use a high level language with support for operator overloading to describe the structure of the code, rather than trying to transform a decorated tree.
Define several abstractions:

- Variables: type (array, float, integer), size...
- Operators: affect, multiply...
- Procedure and functions: parameters, variables...
- Constructs: if, case, for, while...

Sample Code: Variables and Parameters

```
#simple Variable
i = Int("i")
#simple constant
lowfil = Int("lowfil", :constant => 1-center)
#simple constant array
fil = Real("fil", :const => arr,
                  :dim => [ Dim(lowfil,upfil) ])
#simple parameter
ndat = Int("ndat", :direction => :in)
#multidimensional array, an output parameter
y = Real("y", :dir => :out,
              :dim => [ Dim(ndat),
                        Dim(dim_out_min, dim_out_max) ] )
```

Variables and Parameters are objects with a name, a type, and a set of named properties.

Sample Code: Procedure Declaration

The following declaration:

```
p = Procedure("magic_filter", [n,ndat,x,y], [lowfil,upfil])
opn p
close p
```

Outputs Fortran:

```
1  SUBROUTINE magicfilter(n, ndat, x, y)
2    integer(kind=4), parameter :: lowfil = -8
3    integer(kind=4), parameter :: upfil = 7
4    integer(kind=4), intent(in) :: n
5    integer(kind=4), intent(in) :: ndat
6    real(kind=8), intent(in), dimension(0:n-1, ndat) :: x
7    real(kind=8), intent(out), dimension(ndat, 0:n-1) :: y
8  END SUBROUTINE magicfilter
```

Or C:

```
1  void magicfilter(const int32_t n, const int32_t ndat, const double * x, double * y){
2    const int32_t lowfil = -8;
3    const int32_t upfil = 7;
4  }
```

Sample Code: Constructs and Arrays

The following declaration:

```
unroll = 5
tt = (0...unroll).collect { |t| Real("tt#{t}") }
pr For(j, 1, ndat-(unroll-1), unroll){
  pr For( i, 0, n-1) {
    (0...unroll).each { |u| pr tt[u] === 0 }
    For(l, -8, 7) {
      pr k === modulo(i + l, n)
      (0...unroll).each { |u|
        pr tt[u] === tt[u] + x[k,j+u]*fil[l]
      }
    }.unroll
    (0...unroll).each { |u| pr y[j+u,i] === tt[u] }
  }
}
```

Outputs Fortran:

```
1  do j=1, ndat-4, 5
2    do i=0, n-1
3      !......
4      k = modulo(i+2, n)
5      tt0=tt0+x(k,j+0)*fil(2)
6      tt1=tt1+x(k,j+1)*fil(2)
7      !......
8    enddo
9  enddo
```

Or C:

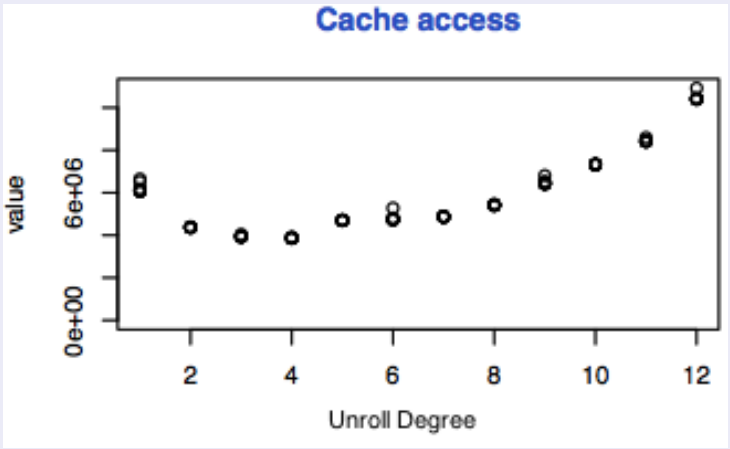
```
1  for(j=1; j<=ndat-4; j+=5){
2    for(i=0; i<=n-1; i+=1){
3  /*..........*/
4      k = modulo(i+2, n)
5      tt0=tt0+x[k-0+(j+0-1)*(n-1-0+1)]*fil[2-lowfil];
6      tt1=tt1+x[k-0+(j+1-1)*(n-1-0+1)]*fil[2-lowfil];
7  /*..........*/
8    }
9  }
```

Generator Evaluation

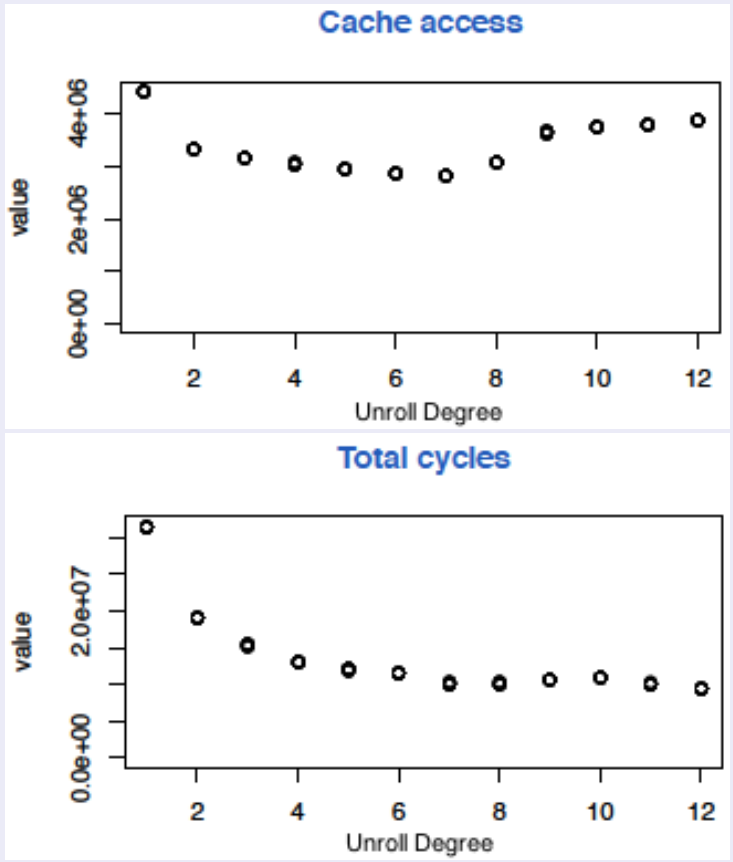
BigDFT: Magicfilter

The generator was used to unroll the Magicfilter an evaluate it's performance on an ARM processor and an Intel processor.

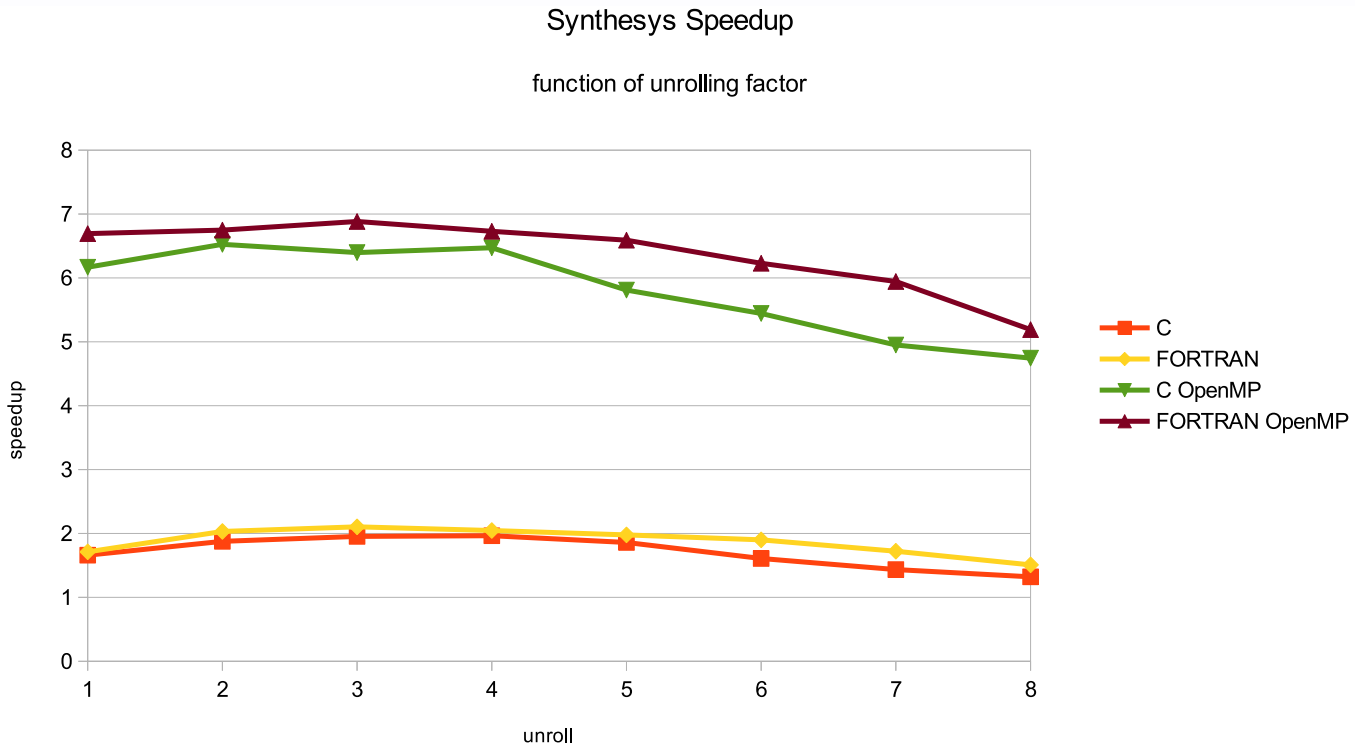
Tegra2



Intel T7500



BigDFT: Synthesis



- Reference is hand tuned code on target architecture
- Toward a BLAS-like library fo wavelets

Example: Laplace Kernel from ARM

```
1   void laplace(const int width,
2                const int height,
3                const unsigned char src[height][width][3],
4                      unsigned char dst[height][width][3]){
5     for (int j = 1; j < height-1; j++) {
6       for (int i = 1; i < width-1; i++) {
7         for (int c = 0; c < 3; c++) {
8           int tmp = -src[j-1][i-1][c] -   src[j-1][i][c] - src[j-1][i+1][c]\
9                     - src[j  ][i-1][c] + 9*src[j  ][i][c] - src[j  ][i+1][c]\
10                    - src[j+1][i-1][c] -   src[j+1][i][c] - src[j+1][i+1][c];
11          dst[j][i][c] = (tmp < 0 ? 0 : (tmp > 255 ? 255 : tmp));
12        }
13      }
14    }
15  }
```

- C reference implementation
- Many opportunities for improvement

Example: Optimization Summary

- Very complex process
- Intimate knowledge of the architecture required
- Numerous versions to be benchmarked
- Difficult to test combination of optimizations:
  - Vectorization,
  - Intermediary data type,
  - Number of pixels processed,
  - Synthesizing loads.
- Can we use BOAST to automate the process?

Example: Laplace Kernel with BOAST

- Based on components instead of pixel
- Use tiles rather than only sequence of elements
- Parameters used in the BOAST version:
  - 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
  - synthesizing_loads: true or false

Example: ARM results

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

- Optimal parameter values:
  - x_component_number: 16
  - y_component_number: 1
  - vector_length: 16
  - temporary_size: 2
  - synthesizing_loads: false
- Close to what ARM engineers found

Example: Performance Portability

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

- Optimal parameter values Intel:
  - x_component_number: 16
  - y_component_number: 4..2
  - vector_length: 8
  - temporary_size: 2
  - synthesizing_loads: false
- Optimal parameter values nVidia:
  - x_component_number: 4
  - y_component_number: 4
  - vector_length: 4
  - temporary_size: 2
  - synthesizing_loads: false

Performance portability among several different architectures.

Real Applications: SPECFEM3D

- SPECFEM3D ported to OpenCL using BOAST
  - Unified code base (CUDA/OpenCL)
  - Refactoring: kernel code base reduced by 40%
  - Similar performance on NVIDIA Hardware
  - Non regression test for GPU kernels
- On the Mont-Blanc prototype:
  - OpenCL+MPI runs
  - Speedup of 3 for the GPU version

Installing BOAST

Install ruby, version >= 1.9.3
On recent debian-based distributions:

```
1 sudo apt-get install ruby ruby-dev
```

And then install the BOAST gem (ruby module):

```
1 sudo gem install BOAST
```

If on a cluster frontend:

```
1 gem install --user-install BOAST
```

First interactive steps

Interactive Ruby:

```
1 irb
```

Simple BOAST commands:

```
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]
```

Creating a Computing Kernel

```
n = BOAST::Int(  "n", :dir => :in )
a = BOAST::Real( "a", :dir => :in,  :dim => [BOAST::Dim(n)] )
b = BOAST::Real( "b", :dir => :out, :dim => [BOAST::Dim(n)] )
p = BOAST::Procedure( "test_proc", [n, a, b] ) {
  BOAST::decl i = BOAST::Int( "i" )
  BOAST::pr BOAST::For( i, 1, n ) {
    BOAST::pr b[i] === a[i] + 2
  }
}
k = BOAST::CKernel::new
BOAST::pr p
k.procedure = p
k.build
BOAST::verbose = true
k.build
> gcc -O2 -Wall -fPIC -I/usr/lib/ruby/1.9.1/x86_64-linux -I/usr/include/ruby-1.9.1 -I/us
> gfortran -O2 -Wall -fPIC -c -o /tmp/test_proc20140624-19378-1qdep6u.o /tmp/test_proc20
> gcc -shared -o /tmp/Mod_test_proc20140624_19378_1qdep6u.so /tmp/Mod_test_proc20140624_
```

Running a Computing Kernel

```
require 'narray'
n = BOAST::Int(  "n", :dir => :in )
a = BOAST::Real( "a", :dir => :in,  :dim => [BOAST::Dim(n)] )
b = BOAST::Real( "b", :dir => :out, :dim => [BOAST::Dim(n)] )
p = BOAST::Procedure( "test_proc", [n, a, b] ) {
  BOAST::decl i = BOAST::Int( "i" )
  BOAST::pr BOAST::For( i, 1, n ) {
    BOAST::pr b[i] === a[i] + 2
  }
}
k = BOAST::CKernel::new
BOAST::pr p
k.procedure = p

input  = NArray.float(1024).random
output = NArray.float(1024)
k.run(input.length, input, output)
(output - input).each { |val|
  raise "Error!" if (val-2).abs > 1e-15
}
stats = k.run(input.length, input, output)
puts "#{stats[:duration]} s"
> 4.911e-06 s
```

Conclusions

- BOAST v1.0 is released
- BOAST language features:
  - Unified C and FORTRAN with OpenMP support,
  - Unified OpenCL and CUDA support,
  - Support for vector programming.
- BOAST runtime features:
  - Generation of parametric kernels,
  - Parametric compilation,
  - Non-regression testing of kernels,
  - Benchmarking capabilities (PAPI support)

Perspectives

- Find and port new kernels to BOAST
- Couple BOAST with other tools:
  - Parametric space pruners (speed up optimization),
  - Binary analysis (guide optimization),
  - Source to source transformation (improve optimization),
  - Binary transformation (improve optimization).
- Improve BOAST:
  - Improve the eDSL to make it more intuitive,
  - Better vector support,
  - Gather feedback.

Question?

Example: Laplace in OpenCL

```
1   kernel laplace(const int width,
2                  const int height,
3                  global const uchar *src,
4                  global       uchar *dst){
5     int i = get_global_id(0);
6     int j = get_global_id(1);
7     for (int c = 0; c < 3; c++) {
8       int tmp = -src[3*width*(j-1) + 3*(i-1) + c]\
9                 - src[3*width*(j-1) + 3*(i  ) + c]\
10                - src[3*width*(j-1) + 3*(i+1) + c]\
11                - src[3*width*(j  ) + 3*(i-1) + c]\
12              + 9*src[3*width*(j  ) + 3*(i  ) + c]\
13                - src[3*width*(j  ) + 3*(i+1) + c]\
14                - src[3*width*(j+1) + 3*(i-1) + c]\
15                - src[3*width*(j+1) + 3*(i  ) + c]\
16                - src[3*width*(j+1) + 3*(i+1) + c];
17      dst[3*width*j + 3*i + c] = clamp(tmp, 0, 255);
18    }
19  }
```

- OpenCL reference implementation
- Outer loops mapped to threads

Example: Vectorizing

```
 1  kernel laplace(const int width,
 2                 const int height,
 3                 global const uchar *src,
 4                 global       uchar *dst){
 5    int i = get_global_id(0);
 6    int j = get_global_id(1);
 7    uchar16 v11_ = vload16( 0, src + 3*width*(j-1) + 3*5*i - 3 );
 8    uchar16 v12_ = vload16( 0, src + 3*width*(j-1) + 3*5*i     );
 9    uchar16 v13_ = vload16( 0, src + 3*width*(j-1) + 3*5*i + 3 );
10    uchar16 v21_ = vload16( 0, src + 3*width*(j  ) + 3*5*i - 3 );
11    uchar16 v22_ = vload16( 0, src + 3*width*(j  ) + 3*5*i     );
12    uchar16 v23_ = vload16( 0, src + 3*width*(j  ) + 3*5*i + 3 );
13    uchar16 v31_ = vload16( 0, src + 3*width*(j+1) + 3*5*i - 3 );
14    uchar16 v32_ = vload16( 0, src + 3*width*(j+1) + 3*5*i     );
15    uchar16 v33_ = vload16( 0, src + 3*width*(j+1) + 3*5*i + 3 );
16    int16 v11 = convert_int16(v11_);
17    int16 v12 = convert_int16(v12_);
18    int16 v13 = convert_int16(v13_);
19    int16 v21 = convert_int16(v21_);
20    int16 v22 = convert_int16(v22_);
21    int16 v23 = convert_int16(v23_);
22    int16 v31 = convert_int16(v31_);
23    int16 v32 = convert_int16(v32_);
24    int16 v33 = convert_int16(v33_);
25    int16 res = v22 * (int)9 - v11 - v12 - v13 - v21 - v23 - v31 - v32 - v33;
26          res = clamp(res, (int16)0, (int16)255);
27    uchar16 res_ = convert_uchar16(res);
28    vstore8(res_.s01234567, 0, dst + 3*width*j + 3*5*i);
29    vstore8(res_.s89ab,     0, dst + 3*width*j + 3*5*i + 8);
30    vstore8(res_.scd,       0, dst + 3*width*j + 3*5*i + 12);
31    dst[3*width*j + 3*5*i + 14] = res_.se;
32  }
```

- Vectorized OpenCL implementation
- 5 pixels (15 compoenets)

Example: Synthesizing Vectors

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

Becomes

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

- Synthesizing loads should save bandwidth
- Could be pushed further

Example: Reducing Variable Size

```
 1   int16 v11 = convert_int16(v11_);
 2   int16 v12 = convert_int16(v12_);
 3   int16 v13 = convert_int16(v13_);
 4   int16 v21 = convert_int16(v21_);
 5   int16 v22 = convert_int16(v22_);
 6   int16 v23 = convert_int16(v23_);
 7   int16 v31 = convert_int16(v31_);
 8   int16 v32 = convert_int16(v32_);
 9   int16 v33 = convert_int16(v33_);
10   int16 res = v22 * (int)9 - v11 - v12 - v13 - v21 - v23 - v31 - v32 - v33;
11         res = clamp(res, (int16)0, (int16)255);
```

Becomes

```
 1   short16 v11 = convert_short16(v11_);
 2   short16 v12 = convert_short16(v12_);
 3   short16 v13 = convert_short16(v13_);
 4   short16 v21 = convert_short16(v21_);
 5   short16 v22 = convert_short16(v22_);
 6   short16 v23 = convert_short16(v23_);
 7   short16 v31 = convert_short16(v31_);
 8   short16 v32 = convert_short16(v32_);
 9   short16 v33 = convert_short16(v33_);
10   short16 res = v22 * (short)9 - v11 - v12 - v13 - v21 - v23 - v31 - v32 - v33;
11           res = clamp(res, (short16)0, (short16)255);
```

- Using smaller intermediary types could save registers

SPECFEM3D
assemble_boundary_potential_on_device : Reference

```
1   typedef float realw;
2   __global__ void assemble_boundary_potential_on_device(realw* d_potential_dot_dot_acousti
3                                                         realw* d_send_potential_dot_dot_bu
4                                                         int num_interfaces,
5                                                         int max_nibool_interfaces,
6                                                         int* d_nibool_interfaces,
7                                                         int* d_ibool_interfaces) {
8
9     int id = threadIdx.x + blockIdx.x*blockDim.x + blockIdx.y*gridDim.x*blockDim.x;
10    int iglob,iloc;
11
12    for( int iinterface=0; iinterface < num_interfaces; iinterface++) {
13      if(id < d_nibool_interfaces[iinterface]) {
14
15        iloc = id + max_nibool_interfaces*iinterface;
16
17        iglob = d_ibool_interfaces[iloc] - 1;
18
19        // assembles values
20        atomicAdd(&d_potential_dot_dot_acoustic[iglob],d_send_potential_dot_dot_buffer[ilo
21      }
22    }
23  }
```

SPECFEM3D assemble_boundary_potential_on_device : BOAST (1)

```
def BOAST::assemble_boundary_potential_on_device(ref = true)
  push_env( :array_start => 0 )
  kernel = CKernel::new
  function_name = "assemble_boundary_potential_on_device"
  num_interfaces                    = Int("num_interfaces",                    \
                                          :dir => :in)
  max_nibool_interfaces             = Int("max_nibool_interfaces",             \
                                          :dir => :in)
  d_potential_dot_dot_acoustic      = Real("d_potential_dot_dot_acoustic",     \
                                          :dir => :out,:dim => [ Dim() ])
  d_send_potential_dot_dot_buffer = Real("d_send_potential_dot_dot_buffer", \
                                          :dir => :in, :dim => [ Dim(num_interfaces*max_ni
  d_nibool_interfaces               = Int("d_nibool_interfaces",               \
                                          :dir => :in, :dim => [ Dim(num_interfaces) ])
  d_ibool_interfaces                = Int("d_ibool_interfaces",                \
                                          :dir => :in, :dim => [ Dim(num_interfaces*max_ni
  p = Procedure(function_name, [d_potential_dot_dot_acoustic,d_send_potential_dot_dot_bu
```

SPECFEM3D assemble_boundary_potential_on_device : BOAST (2)

```
if(get_lang == CUDA and ref) then
  @@output.print File::read("specfem3D/#{function_name}.cu")
elsif(get_lang == CUDA or get_lang == CL) then
  opn p
  id          = Int("id")
  iglob       = Int("iglob")
  iloc        = Int("iloc")
  iinterface  = Int("iinterface")
  decl id, iglob, iloc, iinterface
  pr id === get_global_id(0) + get_global_size(0)*get_global_id(1)
  pr For(iinterface, 0, num_interfaces-1) {
    pr If(id<d_nibool_interfaces[iinterface]) {
      pr iloc === id + max_nibool_interfaces*iinterface
      pr iglob === d_ibool_interfaces[iloc] - 1
      pr atomicAdd(d_potential_dot_dot_acoustic + iglob, \
                   d_send_potential_dot_dot_buffer[iloc])
    }
  }
  close p
else
  raise "Unsupported language!"
end
pop_env( :array_start )
kernel.procedure = p
return kernel
```

SPECFEM3D assemble_boundary_potential_on_device : Generated CUDA

```
1   __global__ void assemble_boundary_potential_on_device(float * d_potential_dot_dot_acoust
2     int id;
3     int iglob;
4     int iloc;
5     int iinterface;
6     id = threadIdx.x + ((blockIdx.x) * (blockDim.x)) + (((gridDim.x) * (blockDim.x)) * (th
7     for(iinterface=0; iinterface<=num_interfaces - (1); iinterface+=1){
8       if(id < d_nibool_interfaces[iinterface - 0]){
9         iloc = id + ((max_nibool_interfaces) * (iinterface));
10        iglob = d_ibool_interfaces[iloc - 0] - (1);
11        atomicAdd(d_potential_dot_dot_acoustic + (iglob), d_send_potential_dot_dot_buffer[
12      }
13    }
14  }
```

SPECFEM3D assemble_boundary_potential_on_device : Generated OpenCL

```
1   kernel void assemble_boundary_potential_on_device(global float * d_potential_dot_dot_aco
2     int id;
3     int iglob;
4     int iloc;
5     int iinterface;
6     id = get_global_id(0) + ((get_global_size(0)) * (get_global_id(1)));
7     for(iinterface=0; iinterface<=num_interfaces - (1); iinterface+=1){
8       if(id < d_nibool_interfaces[iinterface - 0]){
9         iloc = id + ((max_nibool_interfaces) * (iinterface));
10        iglob = d_ibool_interfaces[iloc - 0] - (1);
11        atomicAdd(d_potential_dot_dot_acoustic + (iglob), d_send_potential_dot_dot_buffer[
12      }
13    }
14  }
```