BOAST

Porting HPC applications to the Mont-Blanc prototype using BOAST

Kevin Pouget, **Brice Videau**, Jean-François Méhaut
(INRIA Grenoble - LIG - NANOSIM)

Tutorial BOAST
June 25, 2014

Scientific Application Optimization

- In the past, waiting for a new generation of hardware was enough to obtain performance gains.
- Nowadays, architecture are so different that performance regress when a new architecture is released.
- Sometimes the code is not fit anymore and cannot be compiled.
- Few applications can harness the power of current computing platforms.
- Thus, optimizing scientific application is of paramount importance.

Multiplicity of Architectures

A High performance Computing application can encounter several types of architectures :

- Generalist Multicore CPU (AMD, Intel, PowerPC...)
- Graphical Accelerators (ATI, NVIDIA...)
- Computing Accelerators (CELL, MIC...)
- **Low power CPUs (ARM...)**

Those architectures can present drastically different characteristics.

Architectures Comparison

Architecture	AMD/Intel CPUs	ARM	GPUs	Xeon Phi
Cores	4-12	2-4	512-2496	60
Cache	3l	2l	2l incoherent	2l
Memory (GiB)	2-4 (per core)	1 (per core)	2-6	6-16
Vector Size	2-4	1-2	1-2	8
Peak GFLOPS	10-20 (per core)	2-4 (per core)	500-1500	1000
Peak GiB/s	20-40	2.5-5	150-250	200
TDP W	75	5	200	300
GFLOPS/W	1-3	2-4	2-7	3

Table : Comparison between commonly found architectures in HPC.

Exploiting Various Architectures

Usually work is done on a class of architecture (CPUs or GPUs or accelerators).

Well Known Examples

- Atlas (Linear Algebra CPU)
- PhiPAC (Linear Algebra CPU)
- Spiral (FFT CPU)
- FFTW (FFT CPU)
- NukadaFFT(FFT GPU)

No work targeting several class of architectures. What if the application is not based on a library ?

The Mont-Blanc European Project

European project :

- Develop prototypes of HPC clusters using low power commercially available embedded technology (ARM CPUs, low power GPUs...).
- Design the next generation in HPC systems based on embedded technologies and experiments on the prototypes.
- Develop a portfolio of existing applications to test these systems and optimize their efficiency, using BSC's OmpSs programming model (11 existing applications were selected for this portfolio).

Prototype : based on Exynos 5250 : dual core Cortex A15 with T604 Mali GPU (OpenCL)

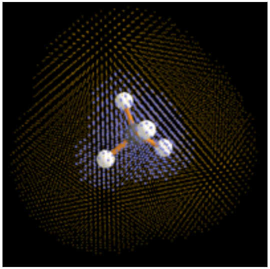
BigDFT a Tool for Nanotechnologies

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.

BigDFT as an HPC application

Implementation details :

- 200,000 lines of Fortran 90 and C
- Supports MPI, OpenMP, CUDA and OpenCL
- Uses BLAS
- Scalability up to 16000 cores of Curie and 288GPUs

Operators can be expressed as 3D convolutions :

- Wavelet Transform
- Potential Energy
- Kinetic Energy

These convolutions are separable and filter are short (16 elements).
Can take up to 90% of the computation time on some systems.

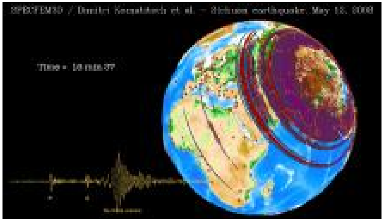
SPECFEM3D a tool for wave propagation research

Wave propagation simulation :

- Used for geophysics and material research,
- Accurately simulate earthquakes,
- Based on spectral finite element.

Developped all around the world :

- Marseilles (CNRS),
- Switzerland (ETH Zurich) CUDA,
- United States (Princeton) Networking,
- Grenoble (LIG/CNRS) OpenCL.


SPECFEM3D / Dimitri Komatitsch et al. - Sichuan earthquake, May 12, 2008
Time = 10 min 37

Sichuan earthequake.

SPECFEM3D as an HPC application

Implementation details :

- 80,000 lines of Fortran 90
- Supports MPI, CUDA, OpenCL and an OMPSs + MPI miniapp
- Scalability up to 693,600 cores on IBM BlueWaters

Talk Outline

2 Case Study

3 A Parametrized Generator

4 Evaluation

5 Conclusions and Future Work

Case Study 1 : BigDFT's 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

```
double filt[16] = {F0, F1, ... , F15};
void magicfilter(int n, int ndat,
                 double *in, double *out){
  double temp;
  for(j=0; j<ndat; j++) {
    for(i=0; i<n; i++) {
       temp = 0;
       for(k=0; k<16; k++) {
         temp+= in[ ((i-7+k)%n) + j*n]
                  * filt[k];
       }
       out[j + i*ndat] = temp;
} } }
```

Case study 2 : SPECFEM3D port to OpenCL

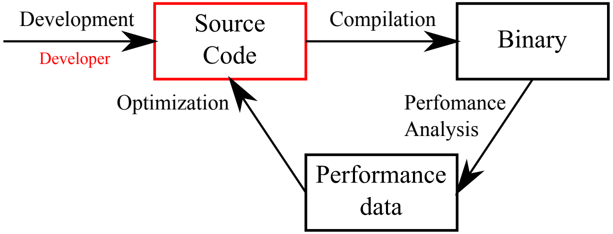
Existing CUDA code :

- 42 kernels and 15000 lines of code
- kernels with 80+ parameters
- $\sim$ 7500 lines of cuda code
- $\sim$ 7500 lines of wrapper code

Objectives :

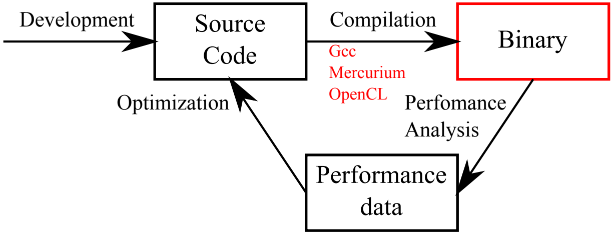
- Factorize the existing code,
- Single OpenCL and CUDA description for the kernels,
- Validate without unit tests, comparing native Cuda to generated Cuda executions
- Keep similar performances.

Classical Workflow



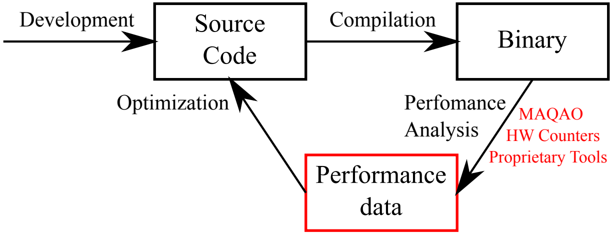
- Kernel optimization workflow
- Usually performed by a knowledgeable developer

Classical Workflow



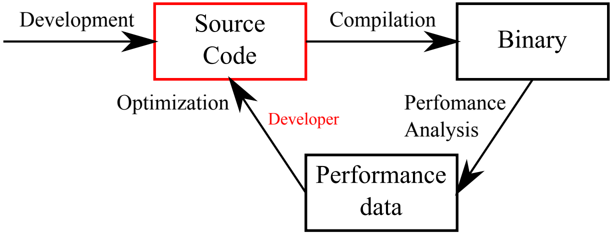
- Compilers perform optimizations
- Architecture specific or generic optimizations

Classical Workflow



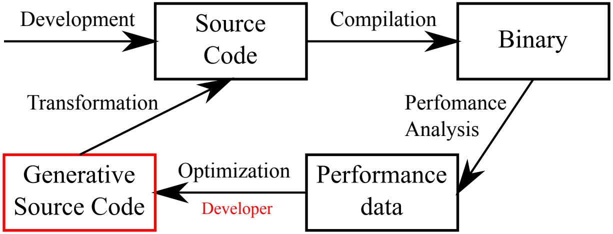
- Performance data hint at source transformations
- Architecture specific or generic hints

Classical Workflow



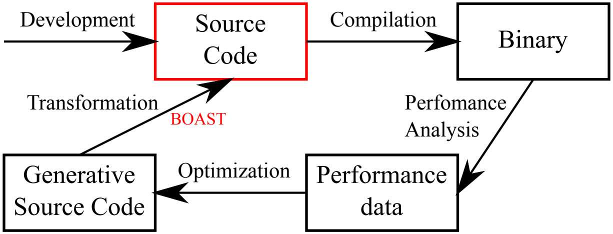
- Multiplication of kernel versions or loss of versions
- Difficulty to benchmark versions against each-other

BOAST Workflow



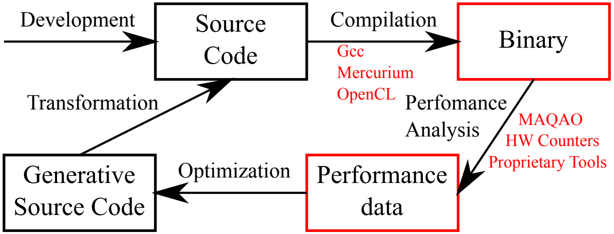
- Meta-programming of optimizations in BOAST
- High level object oriented language

BOAST Workflow



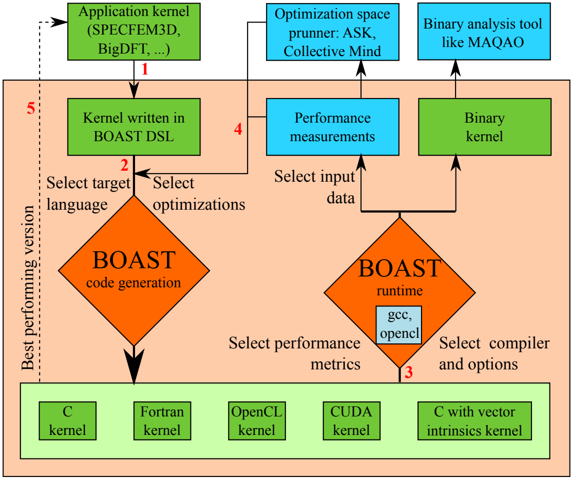
- Generate combination of optimizations
- C, OpenCL, FORTRAN and CUDA are supported

BOAST Workflow



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

BOAST


Application kernel (SPECFEM3D, BigDFT, ...)
Optimization space prunner: ASK, Collective Mind
Binary analysis tool like MAQAO
1
5
Kernel written in BOAST DSL
4
Performance measurements
Binary kernel
2
Select target language
Select optimizations
Select input data
Best performing version
BOAST code generation
BOAST runtime
gcc, opencl
Select performance metrics
Select compiler and options
3
C kernel
Fortran kernel
OpenCL kernel
CUDA kernel
C with vector intrinsics kernel

Use Case Driven

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 and manipulate an unknown number of 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 : for, while...

Sample Code : Variables and Parameters

```
#simple Variable
i = Int "i"
#simple constant
lowfil = Int( "lowfil", :const => 1-center )
#simple constant array
fil = Real("fil", :const => arr, :dim => [ Dim(lowfil,upfil) ])
#simple parameter
ndat = Int("ndat", :dir => :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])
decl p
```

Outputs Fortran :

```
subroutine magicfilter(n, ndat, x, y)
  integer(kind=4), parameter :: lowfil = -8
  integer(kind=4), parameter :: upfil = 7
  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
```

Or C :

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

Sample Code : Constructs and Arrays

The following declaration :

```
unroll = 5
print For(j,1,ndat-(unroll-1), unroll) {
#.....
  (tt2 === tt2 + x[k,j+1]*fil[l]).print
#.....
}
```

Outputs Fortran :

```
do j=1, ndat-4, 5
  !......
  tt2=tt2+x(k,j+1)*fil(l)
  !......
enddo
```

Or C :

```
for(j=1; j<=ndat-4; j+=5){
/*...........*/
  tt2=tt2+x[k-0+(j+1-1)*(n-1-0+1)]*fil[l-lowfil];
/*...........*/
}
```

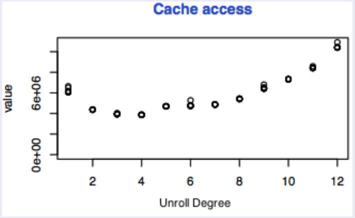
Generator Evaluation

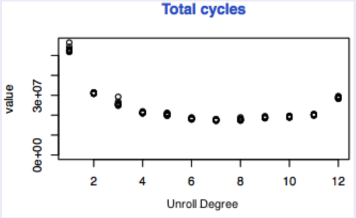
Back to the test cases :

- The generator was used to unroll the Magicfilter an evaluate it's performance on an ARM processor and an Intel processor.
- The generator was used to describe SPECFEM3D kernel.

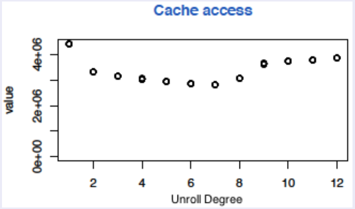
Performance Results

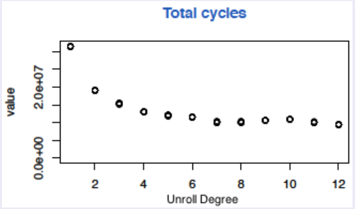
Tegra2





Intel T7500





Improvement for BigDFT

- Most of the convolutions have been ported to BOAST.
- Results are encouraging : on the hardware BigDFT was hand optimized for, convolutions gained on average between 30 and 40% of performance.
- MagicFilter OpenCL versions tailored for problem size by BOAST gain 10 to 20% of performance.

SPECFEM3D OpenCL port

Fully ported to OpenCL with comparable performances (using the `global_s362ani_small` test case) :

- On a 2*6 cores (E5-2630) machine with 2 K40, using 12 MPI processes :
  - OpenCL : 4m15s
  - CUDA : 3m10s
- On an 2*4 cores (E5620) with a K20 using 6 MPI processes :
  - OpenCL : 12m47s
  - CUDA : 11m23s

Difference comes from the capacity of cuda to specify the minimum number of blocks to launch on a multiprocessor. Less than 4000 lines of BOAST code (7500 lines of cuda originally).

Conclusions

Generator has been used to test several loop unrolling strategies in BigDFT.
Highlights :

- Several output languages.
- All constraints have been met.
- Automatic benchmarking framework allows us to test several optimization levels and compilers.
- Automatic non regression testing.
- Several algorithmically different versions can be generated (changing the filter, boundary conditions...).

Future Works and Considerations

Future work :

- Produce an autotuning convolution library.
- Implement a parametric space explorer or use an existing one (ASK : Adaptative Sampling Kit, Collective Mind...).
- Vector code is supported, but needs improvements.
- Test the OpenCL version of SPECFEM3D on the Mont-Blanc prototype.

Question raised :

- Is this approach extensible enough ?
- Can we improve the language used further ?

BOAST

Using BOAST: an Introduction

Brice Videau
(LIG - NANOSIM)

BOAST Tutorial
June 25, 2014

Installing BOAST

Install ruby, version >= 1.9.3

On recent debian-based distributions:

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

And then install the BOAST gem (ruby module):

```
sudo gem install BOAST
```

If on a cluster frontend:

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

First interactive steps

Interactive Ruby:

```
irb
```

Simple BOAST commands:

```
irb(main):001:0> require 'BOAST'
=> true
irb(main):002:0> a = BOAST::Int "a"
=> a
irb(main):003:0> b = BOAST::Real "b"
=> b
irb(main):004:0> BOAST::decl a, b
integer(kind=4) :: a
real(kind=8) :: b
=> [a, b]
```

Defining a Procedure

Simple BOAST Procedure:

```
05:0> p = BOAST::Procedure( "test_proc", [a,b] )
06:0> BOAST::decl p
SUBROUTINE test_proc(a, b)
  integer, parameter :: wp=kind(1.0d0)
  integer(kind=4) :: a
  real(kind=8) :: b
007:0> BOAST::lang = BOAST::C
008:0> BOAST::decl p
void test_proc(int32_t a, double b){
009:0> BOAST::close p
}
```

Available languages are FORTRAN, C, CUDA, CL (OpenCL)

Defining a Full Procedure:

```
010:0> a = BOAST::Real( "a", :dir => :in)
011:0> b = BOAST::Real( "b", :dir => :out)
012:0> p = BOAST::Procedure( "test_proc", [a,b] ) { BOAST::
013:0> BOAST::lang = BOAST::FORTRAN
014:0> BOAST::print p
SUBROUTINE test_proc(a, b)
    integer, parameter :: wp=kind(1.0d0)
    real(kind=8), intent(in) :: a
    real(kind=8), intent(out) :: b
    b = a + 2
END SUBROUTINE test_proc
```

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::For( i, 1, n ) {
    BOAST::print b[i] === a[i] + 2
  }.print
}
k = BOAST::CKernel::new
BOAST::print 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_
-Wl,-Bsymbolic-functions -Wl,-z,relro -rdynamic -Wl,-export-dynamic  -L/usr/lib -lruby-1
```

Running a Computing Kernel

```
require 'narray'
input  = NArray.float(1024).random
output = NArray.float(1024)
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::For( i, 1, n ) {
    BOAST::print b[i] === a[i] + 2
  }.print
}
k = BOAST::CKernel::new
BOAST::print p
k.procedure = p
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
```

The Canonic Case: Vector Addition Kernel

```
def BOAST::vector_add
  kernel = CKernel::new
  function_name = "vector_add"
  n = Int("n",{:dir => :in, :signed => false})
  a = Real("a",{:dir => :in, :dim => [ Dim(0,n-1)] })
  b = Real("b",{:dir => :in, :dim => [ Dim(0,n-1)] })
  c = Real("c",{:dir => :out, :dim => [ Dim(0,n-1)] })
  i = Int("i",{:signed => false})
  print p = Procedure(function_name, [n,a,b,c]) {
    decl i
    if [CL, CUDA].include?(get_lang) then
      print i === get_global_id(0)
      print c[i] === a[i] + b[i]
    else
      print For(i,0,n-1) {
        print c[i] === a[i] + b[i]
      }
    end
  }
  kernel.procedure = p
  return kernel
end
```

Running the Kernel

```
n = 1024*1024
a = NArray.float(n).random
b = NArray.float(n).random
c = NArray.float(n)
c_ref = NArray.float(n)

epsilon = 10e-15

set_lang( FORTRAN )
k = vector_add
k.run(n,a,b,c_ref)

[C, CL, CUDA].each { |lang|
  set_lang( lang )
  c.random
  k = vector_add
  case lang
  when CL
    k.run(n,a,b,c, :global_work_size => [rndup(n,32), 1,1], :local_work_size => [32,1,1]
  when CUDA
    k.run(n,a,b,c, :block_number => [rndup(n,32)/32, 1,1], :block_size => [32,1,1] )
  else
    k.run(n,a,b,c)
  end
  (c_ref - c).abs.each { |diff|
    raise "Warning: residue too big: #{elem}" if elem > epsilon
  }
}
```

Building Kernels

- Running a kernel builds it (if it is not already built).
- Kernels can be built beforehand.
- Usual build parameters can be specified.

Sample:

```
k.build({:FC => 'gfortran', :CC => 'gcc',\
         :FCFLAGS => "-O2", :LDFLAGS => ""})
k.build({:FC => 'gfortran', :CC => 'gcc',\
         :FCFLAGS => "-O2 -fopenmp",:LDFLAGS => "-fopenmp"})
k.build({:FC => 'ifort',    :CC => 'icc',
         :FCFLAGS => "-O2 -openmp" ,:LDFLAGS => "-openmp",:LD => "ifort"})
```

- Probes can be inserted at compile time.
- Default: high resolution timer.

```
stats = k.run(...)
puts stats[:duration]+" s"
```

SPECFEM3D assemble_boundary_potential_on_device : Reference

```
typedef float realw;
__global__ void assemble_boundary_potential_on_device(realw* d_potential_dot_dot_acousti
                                                      realw* d_send_potential_dot_dot_bu
                                                      int num_interfaces,
                                                      int max_nibool_interfaces,
                                                      int* d_nibool_interfaces,
                                                      int* d_ibool_interfaces) {

  int id = threadIdx.x + blockIdx.x*blockDim.x + blockIdx.y*gridDim.x*blockDim.x;
  int iglob,iloc;

  for( int iinterface=0; iinterface < num_interfaces; iinterface++) {
    if(id < d_nibool_interfaces[iinterface]) {

      iloc = id + max_nibool_interfaces*iinterface;

      iglob = d_ibool_interfaces[iloc] - 1;

      // assembles values
      atomicAdd(&d_potential_dot_dot_acoustic[iglob],d_send_potential_dot_dot_buffer[ilo
    }
  }
}
```

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
  decl p
  id         = Int("id")
  iglob      = Int("iglob")
  iloc       = Int("iloc")
  iinterface = Int("iinterface")
  decl id, iglob, iloc, iinterface
  print id === get_global_id(0) + get_global_size(0)*get_global_id(1)
  print For(iinterface, 0, num_interfaces-1) {
    print If(id<d_nibool_interfaces[iinterface]) {
      print iloc === id + max_nibool_interfaces*iinterface
      print iglob === d_ibool_interfaces[iloc] - 1
      print 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
```

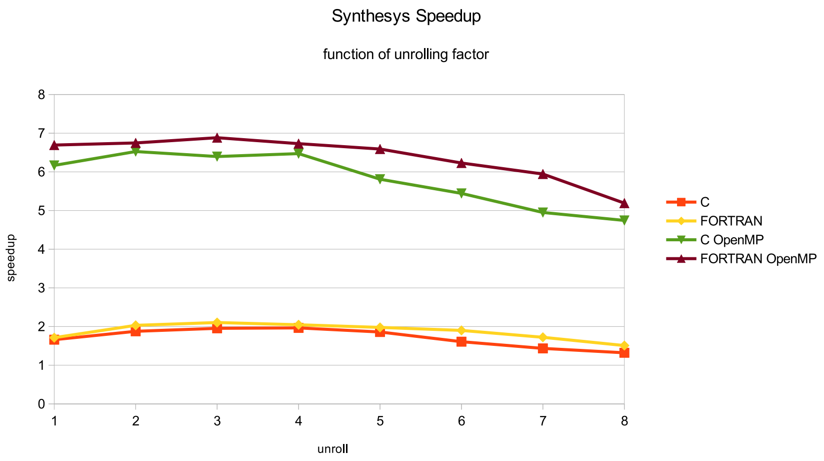
SPECFEM3D assemble_boundary_potential_on_device : Generated CUDA

```
__global__ void assemble_boundary_potential_on_device(float * d_potential_dot_dot_acoust
  int id;
  int iglob;
  int iloc;
  int iinterface;
  id = threadIdx.x + ((blockIdx.x) * (blockDim.x)) + (((gridDim.x) * (blockDim.x)) * (th
  for(iinterface=0; iinterface<=num_interfaces - (1); iinterface+=1){
    if(id < d_nibool_interfaces[iinterface - 0]){
      iloc = id + ((max_nibool_interfaces) * (iinterface));
      iglob = d_ibool_interfaces[iloc - 0] - (1);
      atomicAdd(d_potential_dot_dot_acoustic + (iglob), d_send_potential_dot_dot_buffer[
    }
  }
}
```

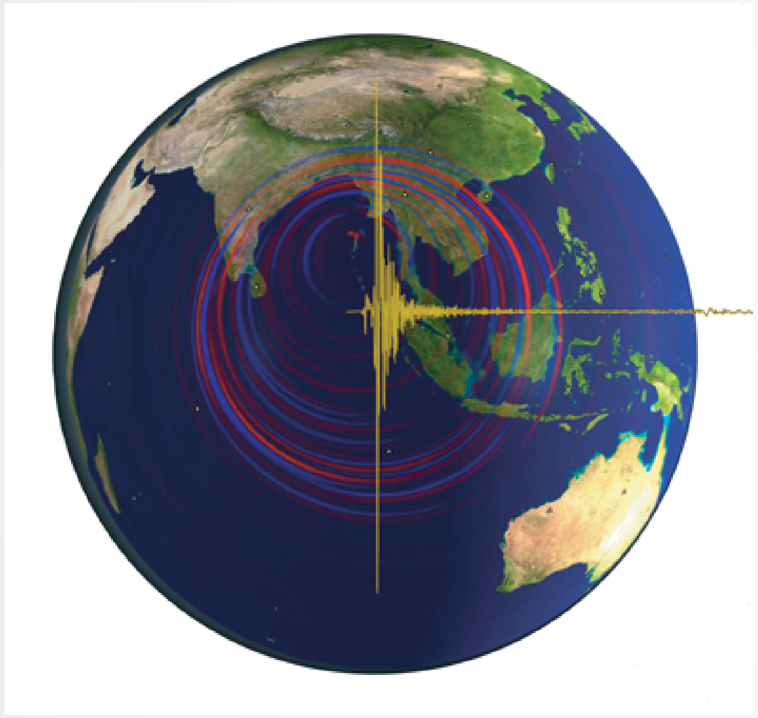
SPECFEM3D assemble_boundary_potential_on_device : Generated OpenCL

```
__kernel void assemble_boundary_potential_on_device(__global float * d_potential_dot_dot
  int id;
  int iglob;
  int iloc;
  int iinterface;
  id = get_global_id(0) + ((get_global_size(0)) * (get_global_id(1)));
  for(iinterface=0; iinterface<=num_interfaces - (1); iinterface+=1){
    if(id < d_nibool_interfaces[iinterface - 0]){
      iloc = id + ((max_nibool_interfaces) * (iinterface));
      iglob = d_ibool_interfaces[iloc - 0] - (1);
      atomicAdd(d_potential_dot_dot_acoustic + (iglob), d_send_potential_dot_dot_buffer[
    }
  }
}
```

BigDFT Synthesis Kernel



Porting SPECFEM3D to OpenCL



Kevin Pouget, Brice Videau

OpenCL vs. Cuda

- GPU programming frameworks
- Same programming model:
  - massively parallel accelerator
  - computing kernels and memory buffers
- But not the same platform targets:
  - Cuda for Nvidia GPUs only
  - OpenCL for **any accelerator processor** (GPU, MPSoC, CPU)

From Cuda to OpenCL

- Some straightforward translations:
  - clCreateBuffer (context, flags, size, ...)
  - cudaMalloc (devPtr, size)

  - clEnqueueWriteBuffer(buffer, size, ptr, ...)
  - cudaMemcpy (src, dst, count, cpyHostToDevice)

- just reorder the parameters (sed regex)

From Cuda to OpenCL

- Some translations more complex:
  - crust_mantle_kernel<<<grid,threads>>>(nb_block_to_compute, d_ibool, ...);

  - clSetKernelArg (crust_mantle_kernel, idx++, &nb_blocks_to_compute);
  - clSetKernelArg (crust_mantle_kernel, idx++, &d_ibool);
  - clEnqueueNDRangeKernel (command_queue, crust_mantle_kernel, ...);

  (no 'preprocessor' in OpenCL)

- more advanced reordering (emacs macro)

From Cuda to OpenCL

- Some translations completely different:
  - ptr_offset = ptr + offset;

  - region_type.origin = _offset_ * sizeof(CL_FLOAT);
  - region_type.size = size;
  - bf_offset = clCreateSubBuffer (bf, CREATE_TYPE_REG, region_type, ...);
  - clReleaseMemObject(bf_offset);

  (no pointer arithmetics in OpenCL)

- manual rewriting, with preproc macro functions

Cuda to OpenCL cohabitation

(Cuda, OpenCL and Cuda+OpenCL compilations)

```
#ifdef USE_OPENCL
    if (run_opencl) {   ...   }
#endif


#ifdef USE_CUDA
    if (run_cuda) {  ...   }
#endif
```

- ./configure --with-opencl --with-cuda
- DATA/Par_file: GPU_RUNTIME = {0 compile-time, 1 cuda, 2 opencl}

Cuda to OpenCL cohabitation

(Cuda, OpenCL and Cuda+OpenCL compilations)

```
#ifdef USE_OPENCL
    if (run_opencl) {   ...   }
#endif


#ifdef USE_CUDA
    if (run_cuda) {  ...   }
#endif
```

```
typedef union {
#ifdef USE_OPENCL
  cl_mem ocl;
#endif
#ifdef USE_CUDA
  realw *cuda;
#endif
} gpu_realw_mem;
```

- ./configure --with-opencl --with-cuda
- DATA/Par_file: GPU_RUNTIME = {0 compile-time, 1 cuda, 2 opencl}

Debugging the execution

- The execution completes*, but the results are wrong.

What do we do now?

--> find a way to debug!

* after fixing compilation problems and OpenCL invalid return codes

Debugging the execution

How to debug OpenCL port?

--> by comparing its execution with Cuda version

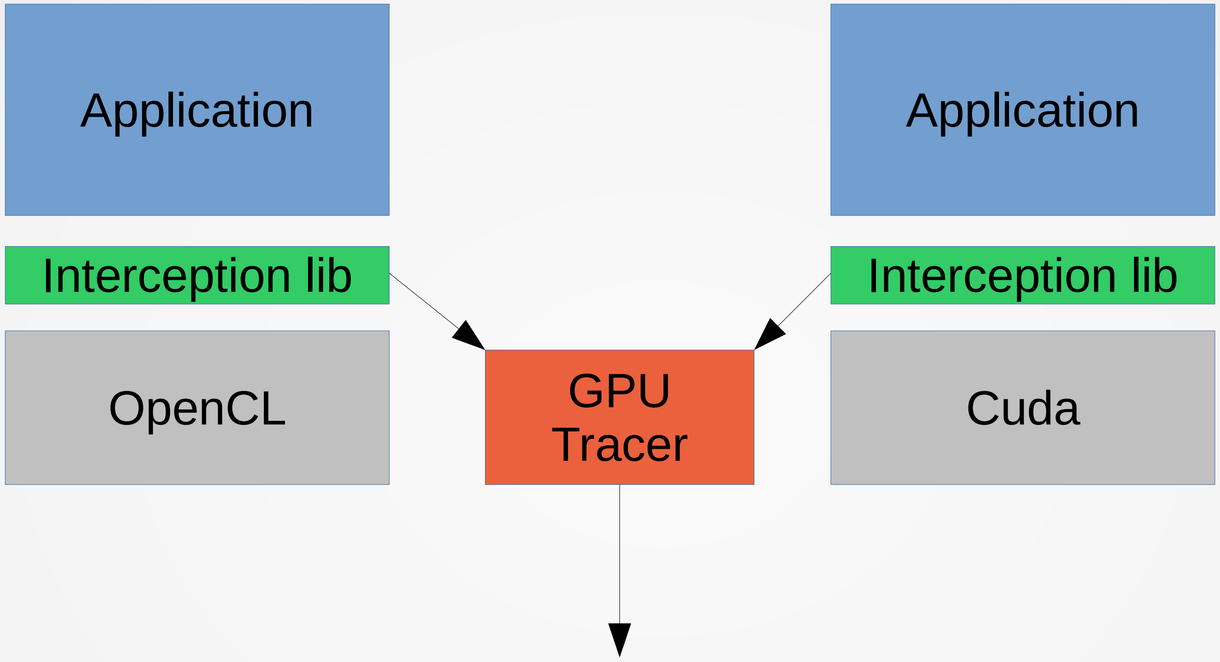
But how?

--> by making sure that both runtimes do the same thing

But how do we do that?

--> by tracing and interpreting the interactions between the application and the GPU runtime

GPU Tracing



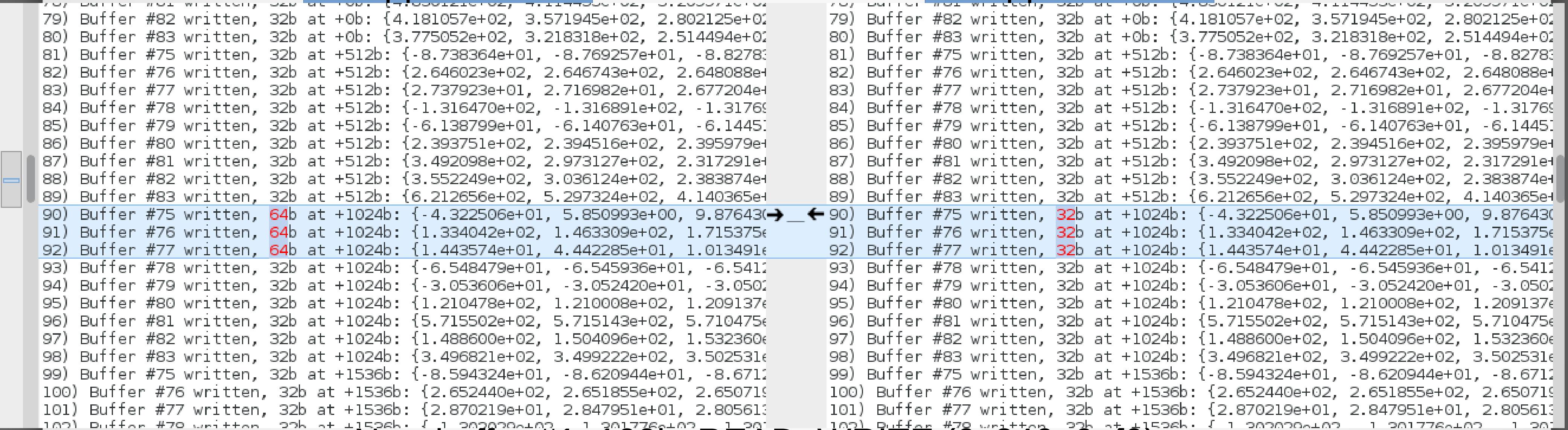
```
New buffer #1, 100b, READ_WRITE (0x246e0ef0)
Buffer #1 written, 32b at +0b: {-5.000000e+00, ... }
```

- Are buffers created with the right size?
- Are they correctly read and written?
  (only the first bits are printed)

GPU Tracing

Application

Application



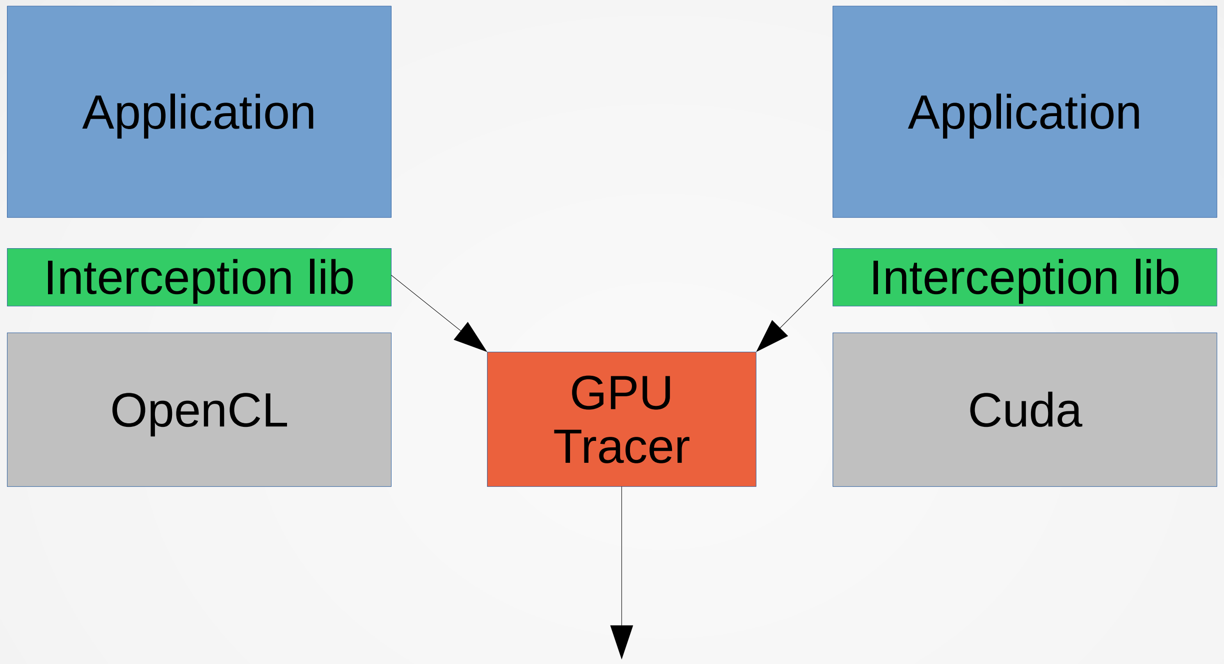
```
79) Buffer #82 written, 32b at +0b: {4.181057e+02, 3.571945e+02, 2.802125e+02
80) Buffer #83 written, 32b at +0b: {3.775052e+02, 3.218318e+02, 2.514494e+02
81) Buffer #75 written, 32b at +512b: {-8.738364e+01, -8.769257e+01, -8.8278
82) Buffer #76 written, 32b at +512b: {2.646023e+02, 2.646743e+02, 2.648088e+
83) Buffer #77 written, 32b at +512b: {2.737923e+01, 2.716982e+01, 2.677204e+
84) Buffer #78 written, 32b at +512b: {-1.316470e+02, -1.316891e+02, -1.3176
85) Buffer #79 written, 32b at +512b: {-6.138799e+01, -6.140763e+01, -6.1445
86) Buffer #80 written, 32b at +512b: {2.393751e+02, 2.394516e+02, 2.395979e+
87) Buffer #81 written, 32b at +512b: {3.492098e+02, 2.973127e+02, 2.317291e+
88) Buffer #82 written, 32b at +512b: {3.552249e+02, 3.036124e+02, 2.383874e+
89) Buffer #83 written, 32b at +512b: {6.212656e+02, 5.297324e+02, 4.140365e+
90) Buffer #75 written, 64b at +1024b: {-4.322506e+01, 5.850993e+00, 9.87643
91) Buffer #76 written, 64b at +1024b: {1.334042e+02, 1.463309e+02, 1.715375e
92) Buffer #77 written, 64b at +1024b: {1.443574e+01, 4.442285e+01, 1.013491e
93) Buffer #78 written, 32b at +1024b: {-6.548479e+01, -6.545936e+01, -6.541
94) Buffer #79 written, 32b at +1024b: {-3.053606e+01, -3.052420e+01, -3.050
95) Buffer #80 written, 32b at +1024b: {1.210478e+02, 1.210008e+02, 1.209137e
96) Buffer #81 written, 32b at +1024b: {5.715502e+02, 5.715143e+02, 5.710475e
97) Buffer #82 written, 32b at +1024b: {1.488600e+02, 1.504096e+02, 1.532360e
98) Buffer #83 written, 32b at +1024b: {3.496821e+02, 3.499222e+02, 3.502531e
99) Buffer #75 written, 32b at +1536b: {-8.594324e+01, -8.620944e+01, -8.671
100) Buffer #76 written, 32b at +1536b: {2.652440e+02, 2.651855e+02, 2.65071
101) Buffer #77 written, 32b at +1536b: {2.870219e+01, 2.847951e+01, 2.80561
```

```
79) Buffer #82 written, 32b at +0b: {4.181057e+02, 3.571945e+02, 2.802125e+02
80) Buffer #83 written, 32b at +0b: {3.775052e+02, 3.218318e+02, 2.514494e+02
81) Buffer #75 written, 32b at +512b: {-8.738364e+01, -8.769257e+01, -8.8278
82) Buffer #76 written, 32b at +512b: {2.646023e+02, 2.646743e+02, 2.648088e+
83) Buffer #77 written, 32b at +512b: {2.737923e+01, 2.716982e+01, 2.677204e+
84) Buffer #78 written, 32b at +512b: {-1.316470e+02, -1.316891e+02, -1.3176
85) Buffer #79 written, 32b at +512b: {-6.138799e+01, -6.140763e+01, -6.1445
86) Buffer #80 written, 32b at +512b: {2.393751e+02, 2.394516e+02, 2.395979e+
87) Buffer #81 written, 32b at +512b: {3.492098e+02, 2.973127e+02, 2.317291e+
88) Buffer #82 written, 32b at +512b: {3.552249e+02, 3.036124e+02, 2.383874e+
89) Buffer #83 written, 32b at +512b: {6.212656e+02, 5.297324e+02, 4.140365e+
90) Buffer #75 written, 32b at +1024b: {-4.322506e+01, 5.850993e+00, 9.87643
91) Buffer #76 written, 32b at +1024b: {1.334042e+02, 1.463309e+02, 1.715375e
92) Buffer #77 written, 32b at +1024b: {1.443574e+01, 4.442285e+01, 1.013491e
93) Buffer #78 written, 32b at +1024b: {-6.548479e+01, -6.545936e+01, -6.541
94) Buffer #79 written, 32b at +1024b: {-3.053606e+01, -3.052420e+01, -3.050
95) Buffer #80 written, 32b at +1024b: {1.210478e+02, 1.210008e+02, 1.209137e
96) Buffer #81 written, 32b at +1024b: {5.715502e+02, 5.715143e+02, 5.710475e
97) Buffer #82 written, 32b at +1024b: {1.488600e+02, 1.504096e+02, 1.532360e
98) Buffer #83 written, 32b at +1024b: {3.496821e+02, 3.499222e+02, 3.502531e
99) Buffer #75 written, 32b at +1536b: {-8.594324e+01, -8.620944e+01, -8.671
100) Buffer #76 written, 32b at +1536b: {2.652440e+02, 2.651855e+02, 2.65071
101) Buffer #77 written, 32b at +1536b: {2.870219e+01, 2.847951e+01, 2.80561
```

```
New buffer #1, 100b, READ_WRITE (0x246e0ef0)
Buffer #1 written, 32b at +0b: {-5.000000e+00, ... }
```

- Are buffers created with the right size?
- Are they correctly read and written?
  (only the first bits are printed)

GPU Tracing



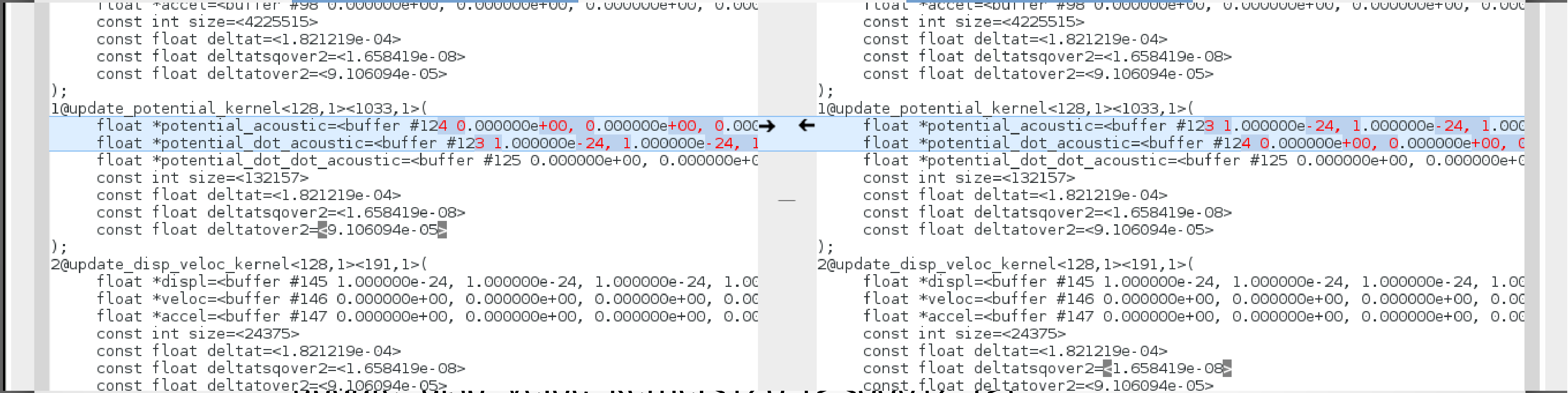
```
update_disp_veloc_kernel<128,1><33012,1>(
    float *displ=<buffer #96 1.000000e-24, 1.000000e-24, ...
    const int size=<4225515>
    const float deltat=<1.821219e-04>
);
```

- Are kernels called in the same order?
- Do we pass the right parameters, with the same values? (again, only the first bits are printed)

GPU Tracing

Application

Application



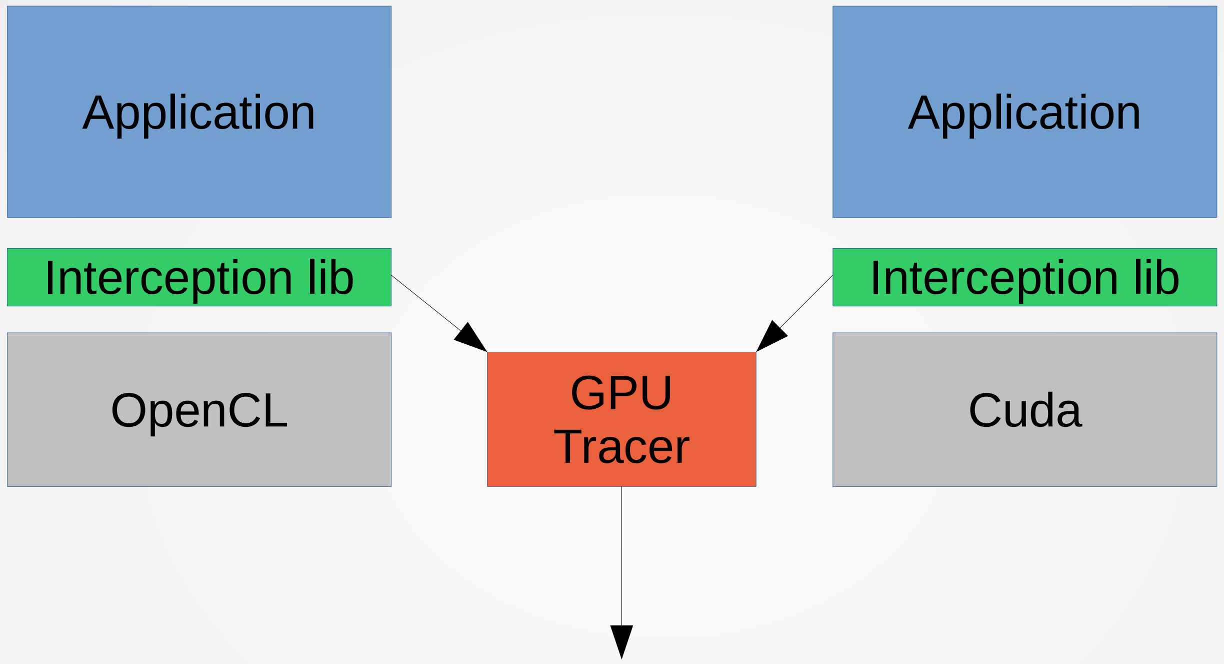
```
    float *accel=<buffer #98 0.000000e+00, 0.000000e+00, 0.000000e+00, 0.000
    const int size=<4225515>
    const float deltat=<1.821219e-04>
    const float deltatsqover2=<1.658419e-08>
    const float deltatover2=<9.106094e-05>
);
1@update_potential_kernel<128,1><1033,1>(
    float *potential_acoustic=<buffer #124 0.000000e+00, 0.000000e+00, 0.000
    float *potential_dot_acoustic=<buffer #123 1.000000e-24, 1.000000e-24, 1
    float *potential_dot_dot_acoustic=<buffer #125 0.000000e+00, 0.000000e+0
    const int size=<132157>
    const float deltat=<1.821219e-04>
    const float deltatsqover2=<1.658419e-08>
    const float deltatover2=<9.106094e-05>
);
2@update_disp_veloc_kernel<128,1><191,1>(
    float *displ=<buffer #145 1.000000e-24, 1.000000e-24, 1.000000e-24, 1.00
    float *veloc=<buffer #146 0.000000e+00, 0.000000e+00, 0.000000e+00, 0.00
    float *accel=<buffer #147 0.000000e+00, 0.000000e+00, 0.000000e+00, 0.00
    const int size=<24375>
    const float deltat=<1.821219e-04>
    const float deltatsqover2=<1.658419e-08>
    const float deltatover2=<9.106094e-05>
```

```
    float *accel=<buffer #98 0.000000e+00, 0.000000e+00, 0.000000e+00, 0.000
    const int size=<4225515>
    const float deltat=<1.821219e-04>
    const float deltatsqover2=<1.658419e-08>
    const float deltatover2=<9.106094e-05>
);
1@update_potential_kernel<128,1><1033,1>(
    float *potential_acoustic=<buffer #123 1.000000e-24, 1.000000e-24, 1.000
    float *potential_dot_acoustic=<buffer #124 0.000000e+00, 0.000000e+00, 0
    float *potential_dot_dot_acoustic=<buffer #125 0.000000e+00, 0.000000e+0
    const int size=<132157>
    const float deltat=<1.821219e-04>
    const float deltatsqover2=<1.658419e-08>
    const float deltatover2=<9.106094e-05>
);
2@update_disp_veloc_kernel<128,1><191,1>(
    float *displ=<buffer #145 1.000000e-24, 1.000000e-24, 1.000000e-24, 1.00
    float *veloc=<buffer #146 0.000000e+00, 0.000000e+00, 0.000000e+00, 0.00
    float *accel=<buffer #147 0.000000e+00, 0.000000e+00, 0.000000e+00, 0.00
    const int size=<24375>
    const float deltat=<1.821219e-04>
    const float deltatsqover2=<1.658419e-08>
    const float deltatover2=<9.106094e-05>
```

```
    float *displ=<buffer #96 1.000000e-24, 1.000000e-24, ...
    const int size=<4225515>
    const float deltat=<1.821219e-04>
);
```

- Are kernels called in the same order?
- Do we pass the right parameters, with the same values? (again, only the first bits are printed)

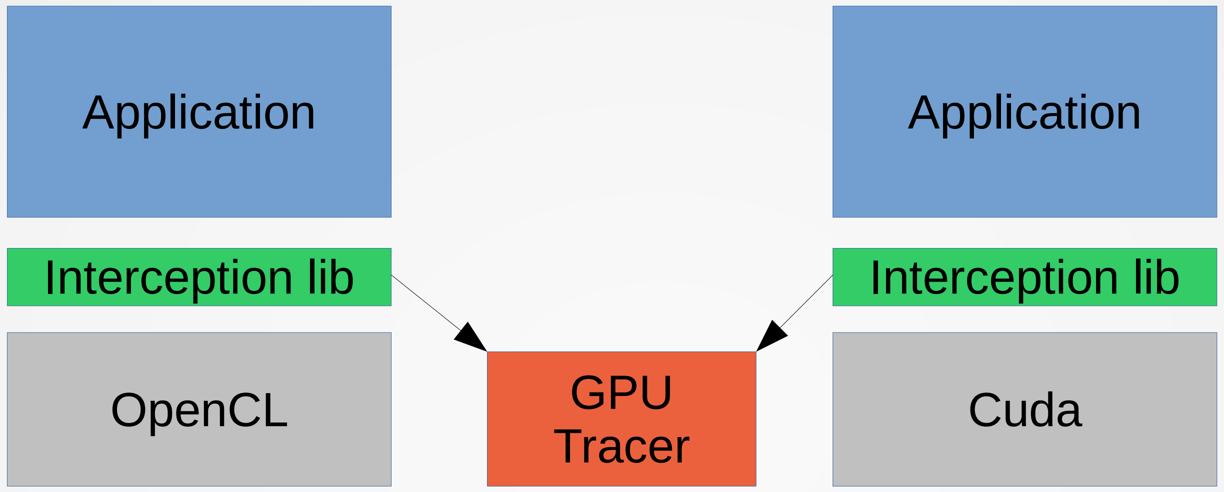
GPU Tracing



```
update_disp_veloc_kernel<128,1><33012,1>(
    float *displ=<buffer #96 1.000000e-24, 1.000000e-24, ...
    const int size=<4225515>
    const float deltat=<1.821219e-04>
    ----
    <out> float *displ=<buffer #96 1.000000e-24, ...
);
```

- Do kernels produce the same values?

GPU Tracing



- ... until both traces were identical, but the results still wrong.

- so we added full buffer printouts
  - their values slowly drifted, but impossible to say where it started...

- ... until Brice got a clue: there is **one** 3-dim kernel, among 2-dim others
  ... and I did not consider that, neither in the appli nor in the tracer:

  compute_add_sources_kernel<5,5><1,1>(...) instead of
  compute_add_sources_kernel<5,5,**5**><1,1,1>(...)

GPU Tracing: going further

- Object leak detection:

  clCreate* without clRelease*
  --> last patch commited in git repository

- Execution profiling
- Memory usage (for Montblanc boards with low memory)

SPECFEM port: future steps

- non-regression tests / build bot tests
  - on its way, will be quick as soon as I get an example
- reduce code duplication
  - generic (OpenCL+cuda) API to avoid many #ifdef in the code
- reduce code duplication
  - factorize very-similar code blocks
    - x, y, z
    - xx, yy, xz, yz
    - crust mantle (cm), outer core (oc), inner core (ic)