DeVito: Fast Finite Difference Computation

Marcos de Aguiar [1] Gerard Gorman [2] Felix J. Herrmann [3] Navjot Kukreja [1] Michael Lange [2] Mathias Louboutin [3] Felippe Vieira Zacarias [1]

[1]SENAI CIMATEC, Salvador, Brazil

[2]Imperial College, London, United Kingdom

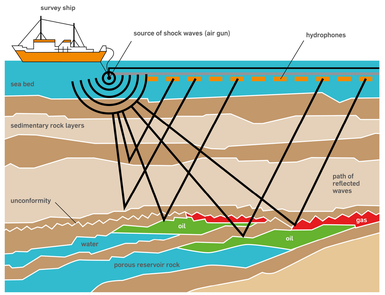
[3]Seismic Laboratory of Imaging and Modelling, University of British Columbia

Motivation

Seismic imaging, used in energy exploration, is arguably the most compute and data intensive application in the private sector. The bulk of the computational cost relates to simulating the propagation of waves through the subsurface. The conventional approach to writing optimised code for these applications would involve multiple man-years of effort that would need to be repeated every time a new development needs to be factored in – like using the elastic wave equation instead of the acoustic equation, or to incorporate some hardware specific optimisation like cache-blocking or auto-tuning.

Computer architectures are rapidly evolving and diversifying to continue delivering performance increases within tight energy envelopes. These changes offer opportunities — but also demand disruptive software changes to harness the full hardware potential. There have been multiple explorations into this problem of achieving performance portability across diverse, rapidly evolving architectures while maintaining a common code-base. This is usually solved by building code optimisation tools. An important question to be addressed while designing these tools is the abstraction level at which the application developer interacts with the tool. In the case of a code-to-code translation, for example, a large majority of the implementation details that could be used for optimisation are already frozen as implicit assumptions in code. Therefore it becomes imperative to have the application developer write as less code as possible - so that the rest of the code can be customised to a particular hardware platform.

Here we propose *DeVito* as a solution to this problem. DeVito leverages domain-specific languages (DSL), code generation and Just-In-Time(JIT) compilation to delay the code generation until the properties of the target platform are known. At the highest level of abstraction application developers define the operations to be performed in symbolic math, while at the lower levels Devito generates native code that is highly optimised to one of several available target platforms. This effectively decouples the domain knowledge – that is expressed as high level symbolic math to be more intuitive to domain experts, from the platform specific optimisations that code-tuning experts can focus on. DeVito's significant advantage over other tools that take a similar approach is the fact that it can easily be integrated into a larger workflow involving complex programs running on multiple nodes of a cluster.



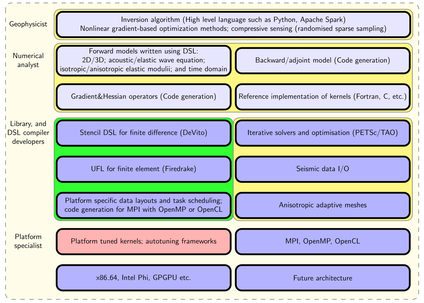
Full Waveform Inversion

- Given a sound source, an array of hydrophones, and a model for wave propagation, use a wiggle for wiggle comparison to infer the structure of the subsurface.
- The Subsurface velocity image is built up in an iterative process by running the wave propagation model forwards and backwards, in order to minimise the difference between the results measured at the hydrophones and the simulated data.
- The dominant computational expense here is running the wave propagation model iteratively for each set of data.
- The wavefield(s) to be propagated can easily reach Petabytes in memory.
- A major bottleneck in the development of more complex FWI models is the fact that writing code optimised for performance gets error-prone as the wave propagation model is made with increasingly detailed representation of the physical properties of the earth.

Opportunities & challenges

- Regular grids with finite difference is the standard of the O&G industry.
  - Higher order methods can achieve greater accuracy at lower cost:
    - Coarser resolution (fewer grid points).
    - Larger time step.
    - Easier to write compute bound code-thereby increasing the maximum attainable performance.
- Implementation complexity!
  - Implementing higher order methods is much more involved and error prone.
- Hardware continues to track Moore's Law.
- Software to utilise this additional capacity is getting increasingly complex to write.
  - Various parallel programming models (MPI, threads, etc.)
  - Deep memory hierarchy, data locality.
  - SIMD vectorisation.
  - Heterogeneous computing Intel® Xeon®, Intel® Xeon Phi™.
- Increasingly difficult for domain specialists to implement HPC software.
- Domain Specific Languages (DSL's) offer a route to bridge the divide between domain specialists (often the application developers) and parallel programming specialists (in this case compiler writers).

Architectural layout



Finite difference using symbolic mathematics and code generation

```python
class ForwardOperator(Operator):
    def __init__(self, m, src, damp, rec, u, time_order=4, spc_order=12, **kwargs):
        assert(m.shape == damp.shape)
        u.pad_time = False
        # Set time and space orders
        u.time_order = time_order
        u.space_order = spc_order
        # Derive stencil from symbolic equation
        eqn = m * u.dt2 - u.laplace + damp * u.dt
        stencil = solve(eqn, u.forward)[0]
        # Add substitutions for spacing (temporal and spatial)
        s, h = symbols('s h')
        subs = {s: src.dt, h: src.h}
        super(ForwardOperator, self).__init__(src.nt, m.shape, stencils=Eq(u.forward, stencil),
                                              substitutions=subs, spc_border=spc_order/2,
                                              time_order=time_order, forward=True, dtype=m.dtype,
                                              **kwargs)

        # Insert source and receiver terms post-hoc
        self.input_params += [src, rec]
        self.propagator.time_loop_stencils_a = src.add(m, u) + rec.read(u)
        self.propagator.add_devito_param(src)
        self.propagator.add_devito_param(rec)
```

```
#include <iostream>
#include <cstdlib>
#include <cstdio>
#include <iostream>
#include <fstream>
#include <sstream>
#include <omp>
#include <string>
#include <sys/time.h>
#include <sys/time.h>
#include <math.h>
extern "C" int ForwardOperator(float *m_vec, float *src_vec, float *damp_vec, float *rec_vec, float *u_vec, float *s_vec, float *s_vec, float *h_vec, float *ih_vec, float *pb_vec
  {
  float (*m)[16][16] = (float (*)[16][16][16]) m_vec;
  float (*u)[2][16][16][16] = (float (*)[2][16][16][16]) u_vec;
  float (*damp)[16][16][16] = (float (*)[16][16][16]) damp_vec;
  float (*src)[16] = (float (*)[16]) src_vec;
  float (*rec)[16][16][16] = (float (*)[16][16][16]) rec_vec;
  float (*s)[16] = (float (*)[16]) s_vec;
  float (*h)[16] = (float (*)[16]) h_vec;
  float (*ih)[16] = (float (*)[16]) ih_vec;
  float (*pb)[16][16] = (float (*)[16][16]) pb_vec;
  int x0, y0, z0;
  int x1, y1, z1;
  int t, p;
  int t0 = (1ul);
  int t1 = (1ul);
  int p0 = (0);
  int p1 = (1);
  int time = t0, t1;
  int sf = (1);
  int si = (1);
  int ix, iy, iz;
  int ix0, iy0, iz0;
  int ix1, iy1, iz1;
  int ix2, iy2, iz2;
  int ix3, iy3, iz3;
  int ix4, iy4, iz4;
  int ix5, iy5, iz5;
  int ix6, iy6, iz6;
  int ix7, iy7, iz7;
  int ix8, iy8, iz8;
  int ix9, iy9, iz9;
  int
```