Imperial College London

DEVITO

The Devito DSL and Compiler Framework: From Symbolic PDEs to HPC Code

George Bisbas[1,2]**,** Fabio Luporini[3], Mathias Louboutin[4], Rhodri Nelson[2], Ed Caunt[2], Gerard Gorman[2,3], Paul H.J. Kelly[1]

[1]Imperial College London, Dept. of Computing
[2]Imperial College London, Dept. of Earth Sciences and Engineering
[3]Devito Codes, UK
[4]Georgia Institute of Technology, Atlanta, USA

MS307: PDE Simulations with High-Productivity Languages at the Dawn of Exascale, SIAM CSE23, Amsterdam

Scientific simulations are demanding

🚧 **Very complex to model** (complicated PDEs, BCs, external factors, complex geometries)

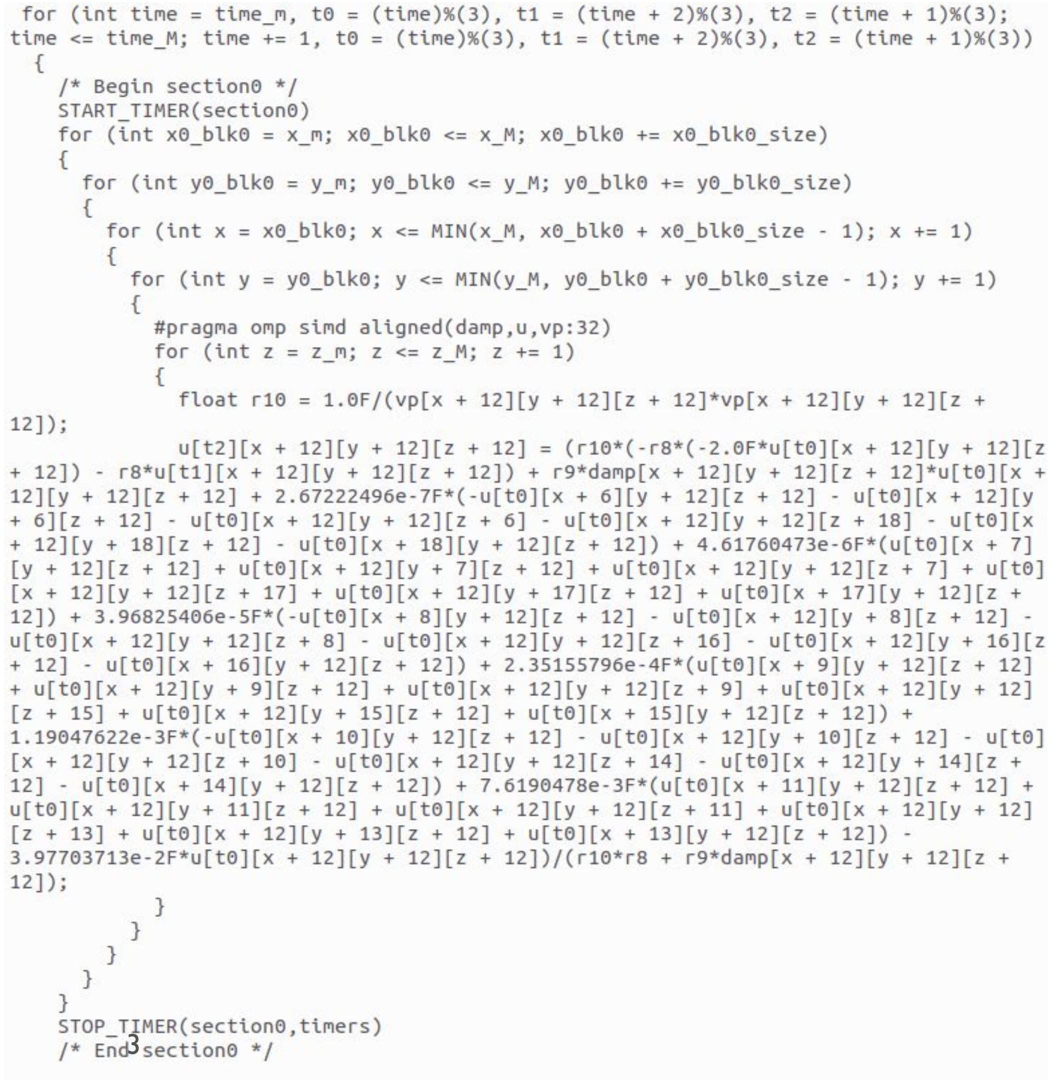
✅ Software offering **high-level, high-productivity DSLs**
✅ Let **domain experts** navigate their design space

🚧 **Resource-demanding** (O(10^3) FLOPs per loop iteration, high memory pressure, 3D grids with > 10^9 grid points, often O(10^3) time steps, inverse problems, ≈O(billions) TFLOPs. Which means days, or weeks, or months on supercomputers!

✅ **Offer** automated optimisations and **efficient codegen for HPC** workloads
✅ **Higher resolution in space and time** opens up compelling new applications
✅ **Unlocks** ever-increasing application **value**

- Complex FD-stencil
- Just a part of these codes!
- No one wants to write it
- No one wants to optimise it
- No one wants to debug

```
void kernel(…) {
  …
  <impenetrable code with
aggressive performance
optimizations, manually
applied, full-time human
resources, less
reproducibility, debugging
nightmares>
  …}
```

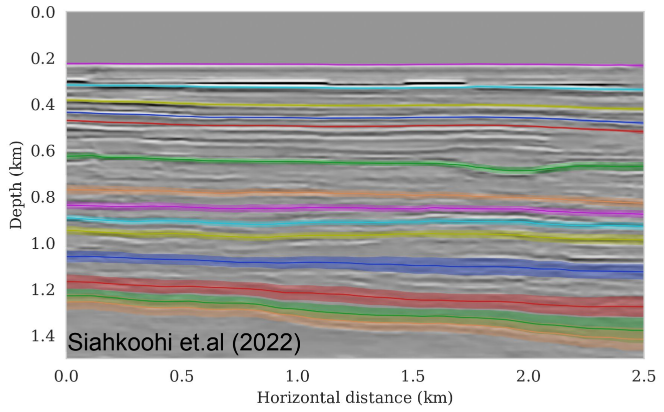


```
  for (int time = time_m, t0 = (time)%(3), t1 = (time + 2)%(3), t2 = (time + 1)%(3);
time <= time_M; time += 1, t0 = (time)%(3), t1 = (time + 2)%(3), t2 = (time + 1)%(3))
  {
    /* Begin section0 */
    START_TIMER(section0)
    for (int x0_blk0 = x_m; x0_blk0 <= x_M; x0_blk0 += x0_blk0_size)
    {
      for (int y0_blk0 = y_m; y0_blk0 <= y_M; y0_blk0 += y0_blk0_size)
      {
        for (int x = x0_blk0; x <= MIN(x_M, x0_blk0 + x0_blk0_size - 1); x += 1)
        {
          for (int y = y0_blk0; y <= MIN(y_M, y0_blk0 + y0_blk0_size - 1); y += 1)
          {
            #pragma omp simd aligned(damp,u,vp:32)
            for (int z = z_m; z <= z_M; z += 1)
            {
              float r10 = 1.0F/(vp[x + 12][y + 12][z + 12]*vp[x + 12][y + 12][z +
12]);
              u[t2][x + 12][y + 12][z + 12] = (r10*(-r8*(-2.0F*u[t0][x + 12][y + 12][z
+ 12]) - r8*u[t1][x + 12][y + 12][z + 12]) + r9*damp[x + 12][y + 12][z + 12]*u[t0][x +
12][y + 12][z + 12] + 2.67222496e-7F*(-u[t0][x + 6][y + 12][z + 12] - u[t0][x + 12][y
+ 6][z + 12] - u[t0][x + 12][y + 12][z + 6] - u[t0][x + 12][y + 12][z + 18] - u[t0][x
+ 12][y + 18][z + 12] - u[t0][x + 18][y + 12][z + 12]) + 4.61760473e-6F*(u[t0][x + 7]
[y + 12][z + 12] + u[t0][x + 12][y + 7][z + 12] + u[t0][x + 12][y + 12][z + 7] + u[t0]
[x + 12][y + 12][z + 17] + u[t0][x + 12][y + 17][z + 12] + u[t0][x + 17][y + 12][z +
12]) + 3.96825406e-5F*(-u[t0][x + 8][y + 12][z + 12] - u[t0][x + 12][y + 8][z + 12] -
u[t0][x + 12][y + 12][z + 8] - u[t0][x + 12][y + 12][z + 16] - u[t0][x + 12][y + 16][z
+ 12] - u[t0][x + 16][y + 12][z + 12]) + 2.35155796e-4F*(u[t0][x + 9][y + 12][z + 12]
+ u[t0][x + 12][y + 9][z + 12] + u[t0][x + 12][y + 12][z + 9] + u[t0][x + 12][y + 12]
[z + 15] + u[t0][x + 12][y + 15][z + 12] + u[t0][x + 15][y + 12][z + 12]) +
1.19047622e-3F*(-u[t0][x + 10][y + 12][z + 12] - u[t0][x + 12][y + 10][z + 12] - u[t0]
[x + 12][y + 12][z + 10] - u[t0][x + 12][y + 12][z + 14] - u[t0][x + 12][y + 14][z +
12] - u[t0][x + 14][y + 12][z + 12]) + 7.6190478e-3F*(u[t0][x + 11][y + 12][z + 12] +
u[t0][x + 12][y + 11][z + 12] + u[t0][x + 12][y + 12][z + 11] + u[t0][x + 12][y + 12]
[z + 13] + u[t0][x + 12][y + 13][z + 12] + u[t0][x + 13][y + 12][z + 12]) -
3.97703713e-2F*u[t0][x + 12][y + 12][z + 12])/(r10*r8 + r9*damp[x + 12][y + 12][z +
12]);
            }
          }
        }
      }
    }
    STOP_TIMER(section0,timers)
    /* End section0 */
```

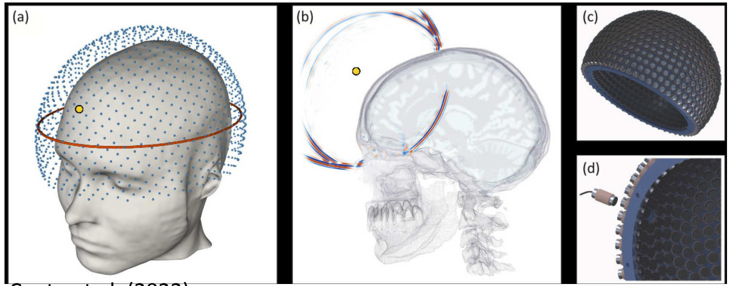
Introducing Devito

- **Devito is a DSL and compiler framework** for finite difference and stencil computations
- **Solving PDEs** using the **finite-difference method for structured grids** (but not limited to this!)
- Users model in the high-level DSL using symbolic math abstraction, and the compiler auto-generates HPC optimized code
- Inter(-national, -institutional,-disciplinary), lots of users from academia and industry
- Real-world problem simulations! (CFD, seismic/medical imaging, finance, tsunamis)

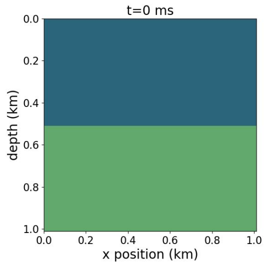
CI-core passing | CI-mpi passing | CI-gpu passing | codecov 88% | chat on slack | benchmarked by asv | pypi package 4.8.0 | launch binder | docker images

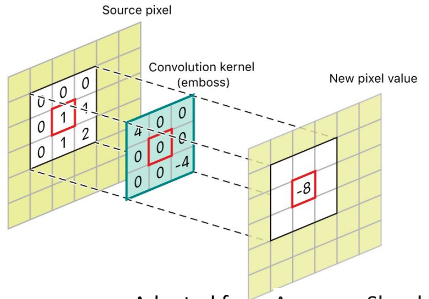


Siahkoohi et.al (2022)



Cueto et.al. (2022)





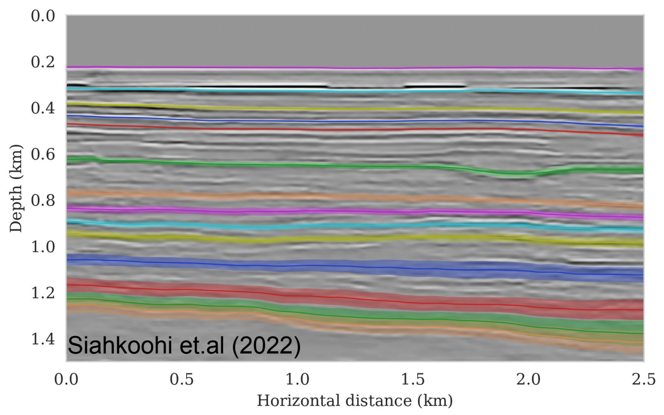
Adapted from Aryaman Sharda

Introducing Devito

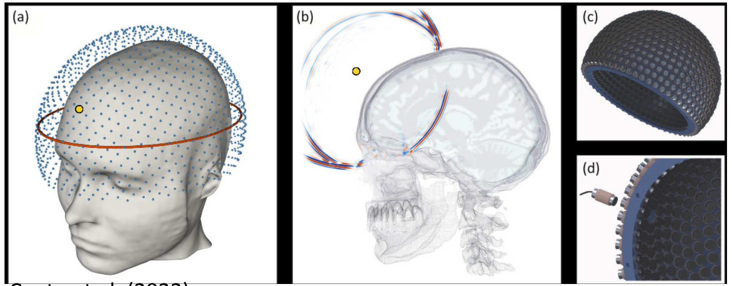
- **Open source** - MIT lic. - Try now! https://github.com/devitocodes/devito
- **Compose with** packages from the Python ecosystem (e.g. PyTorch, NumPy, Dask, TensorFlow)
- Best practices in **software engineering**: extensive software testing, code verification, CI/CD, regression tests, documentation, tutorials and PR code review
- Actual compiler technology (not a S2S translator or templates!)

CI-core passing | CI-mpi passing | CI-gpu passing | codecov 88% | chat on slack | benchmarked by asv | pypi package 4.8.0

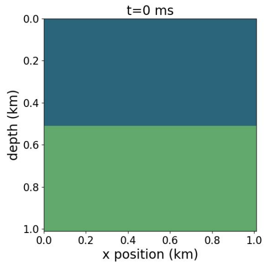
launch binder | docker images

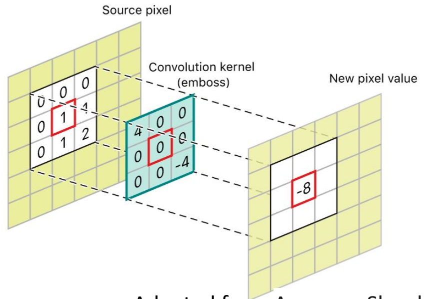


Siahkoohi et.al (2022)



Cueto et.al. (2022)

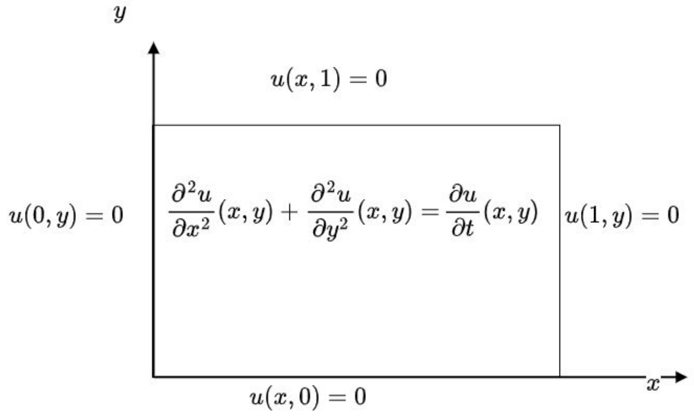


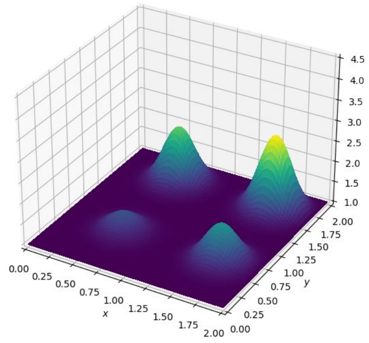


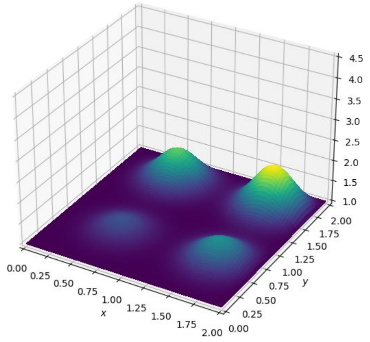
Adapted from Aryaman Sharda

An example from textbook maths to via Devito DSL

2D Heat diffusion modelling


y
$u(x,1) = 0$
$u(0,y) = 0$
$\frac{\partial^2 u}{\partial x^2}(x,y) + \frac{\partial^2 u}{\partial y^2}(x,y) = \frac{\partial u}{\partial t}(x,y)$
$u(1,y) = 0$
$u(x,0) = 0$
x





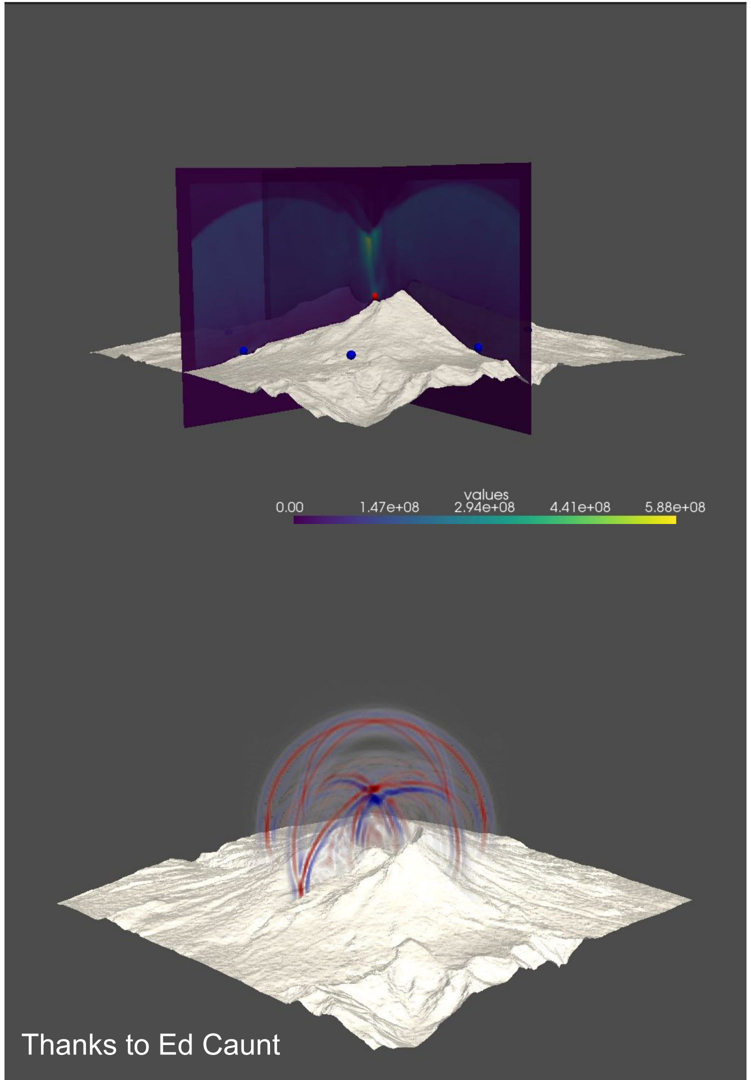
```
from devito import Eq, Grid, TimeFunction, Operator, solve

# Define a structured grid
nx, ny = 10, 10
grid = Grid(shape=(10, 10))

# Define a field on the structured grid
u = TimeFunction(name='u', grid=grid, space_order=2)

# Define a forward time-stepping symbolic equation
eqn = Eq(u.dt, u.laplace)
eqns = [Eq(u.forward, solve(eqn, u.forward))]

# Define boundary conditions
x, y = grid.dimensions
t = grid.stepping_dim

bc_left = Eq(u[t + 1, 0, y], 0.)
bc_right = Eq(u[t + 1, nx-1, y], 0.)
bc_top = Eq(u[t + 1, x, ny-1], 0.)
bc_bottom = Eq(u[t + 1, x, 0], 0.)

eqns += [bc_left, bc_bottom, bc_right, bc_top]
op = Operator(eqns)

# Compute for 3 timesteps
op.apply(time_M=3, dt=0.1)
```

Devito’s API is much richer though…

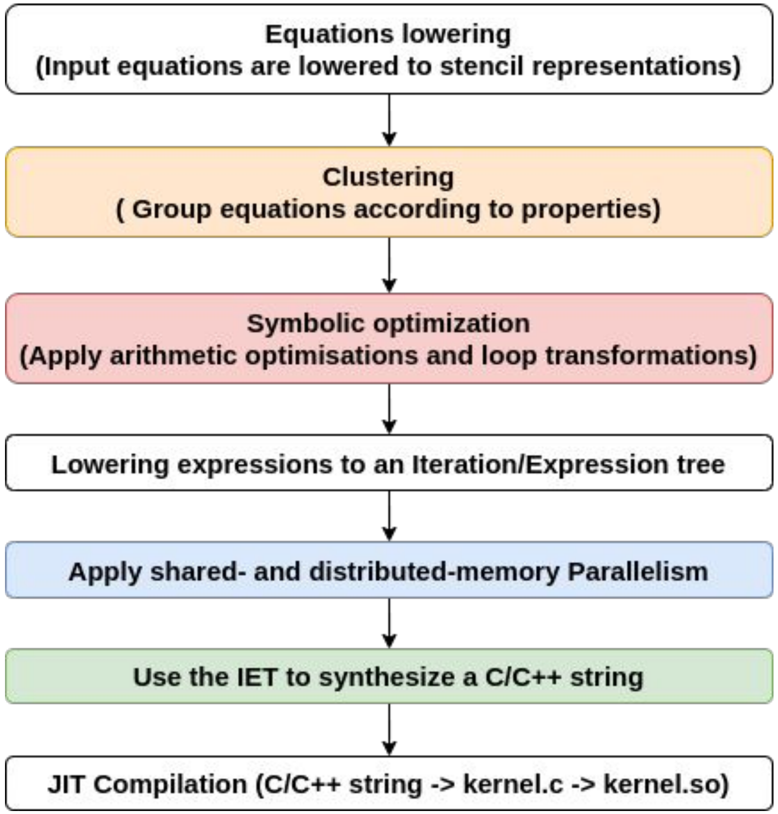
- Any PDE simulations
- Boundary conditions
- Sparse off-grid operations (interpolations)
- Subdomains
- Immersed boundaries (WIP)

Examples available with:

- *CFD* (convection/diffusion/cavity flow/shallow waters (tsunami) /Darcy flow)
- *Wave propagators* ((Visco-)Acoustic/Elastic, TTI)
- *Seismic/Medical Imaging* (FWI/RTM)
- *Finance*



The Devito compiler automatically applies lots of optimisations


Equations lowering
(Input equations are lowered to stencil representations)
Clustering
(Group equations according to properties)
Symbolic optimization
(Apply arithmetic optimisations and loop transformations)
Lowering expressions to an Iteration/Expression tree
Apply shared- and distributed-memory Parallelism
Use the IET to synthesize a C/C++ string
JIT Compilation (C/C++ string -> kernel.c -> kernel.so)

- Loop Fusion
- Hoist and optimize Dimension-invariant sub-expressions
- Loop blocking
- Factorization
- Loop Fission
- Optimize powers
- Cross-iteration redundancies elimination
- Eliminate redundant array expressions
- Common sub-expressions elimination

- Flush denormals
- Distributed Memory parallelism
- SIMD vectorization
- Shared memory parallelism
- Profiling instrumentation

+ advanced combinations of them!
+ heuristics to tune them more!

Write once,

Run everywhere!

- Serial C/CPP code
- OpenMP parallel code
- MPI (+ OpenMP)
- OpenMP 5 GPU offloading via Clang
- OpenACC GPU offloading

Math-related optimisations -- Reducing OI/AI of stencil kernels

- **Isotropic Acoustic**

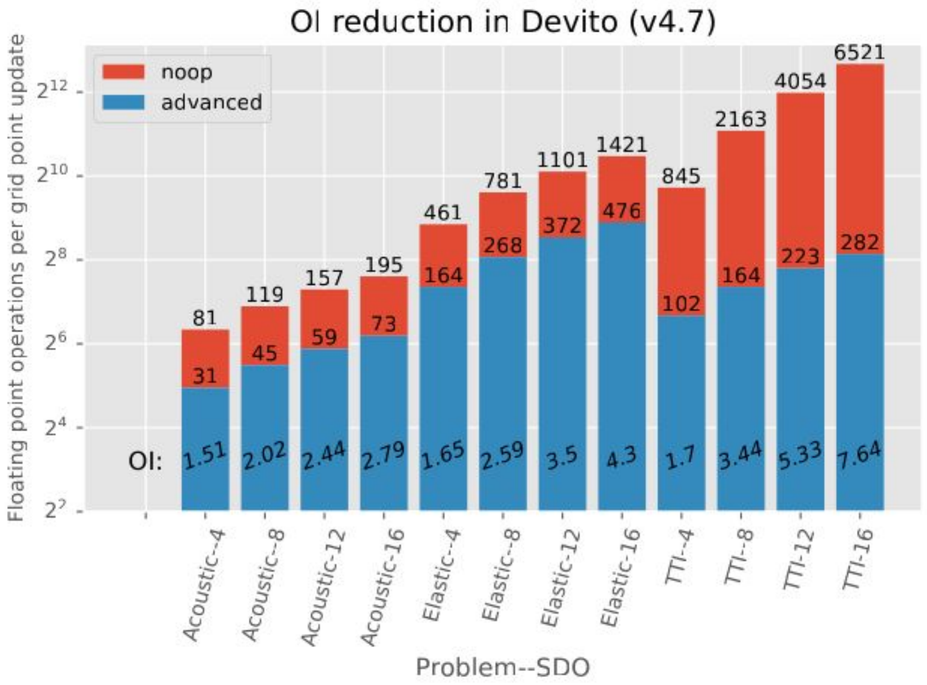
  Generally known, single scalar PDE, laplacian like, low cost

- **Isotropic Elastic**

  Coupled system of a vectorial and tensorial PDE, explosive source, increased data movement, first order in time, cross-loop data dependencies

- **Anisotropic Acoustic (aka TTI, Zhang-Louboutin variation)**

  Industrial applications, rotated laplacian, coupled system of two scalar PDEs, several variations based on variable or constant density



Problem--SDO	noop	advanced	OI
Acoustic--4	81	31	1.51
Acoustic--8	119	45	2.02
Acoustic-12	157	59	2.44
Acoustic-16	195	73	2.79
Elastic--4	461	164	1.65
Elastic--8	781	268	2.59
Elastic-12	1101	372	3.5
Elastic-16	1421	476	4.3
TTI--4	845	102	1.7
TTI--8	2163	164	3.44
TTI-12	4054	223	5.33
TTI-16	6521	282	7.64

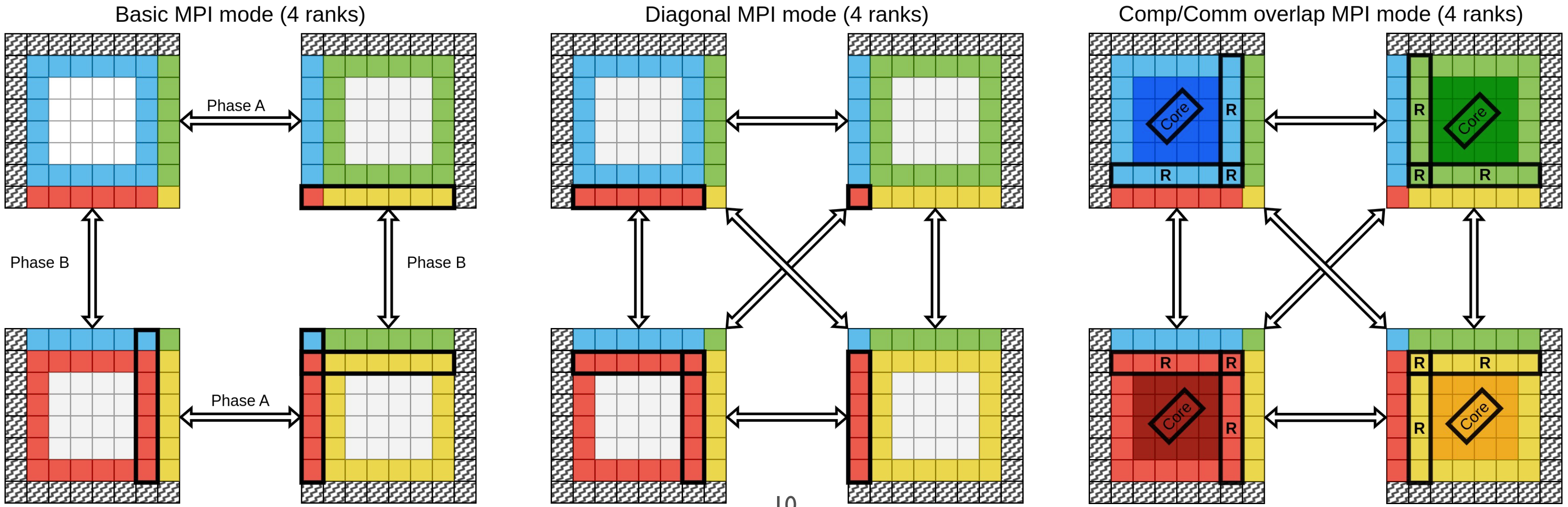
🚧 Not the typical memory-bw bound stencils!

✅ Significantly reduced operational intensity!

Write once, run everywhere!

Devito offers automated MPI-openmp code generation, taking advantage of several optimised communication/computation patterns

User only has to use : “ *DEVITO_MPI=<mode> mpirun -n 2* python my_devitoscript.py ”



Basic MPI mode (4 ranks)
Phase A
Phase B
Phase B
Phase A
Diagonal MPI mode (4 ranks)
Comp/Comm overlap MPI mode (4 ranks)
Core
R

Figure style influenced from Li et.al, ICPP 2021

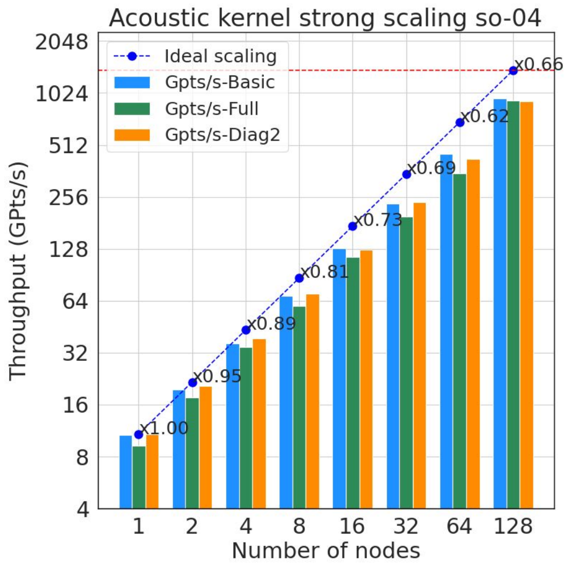
Performance evaluation: Strong scaling on Archer2

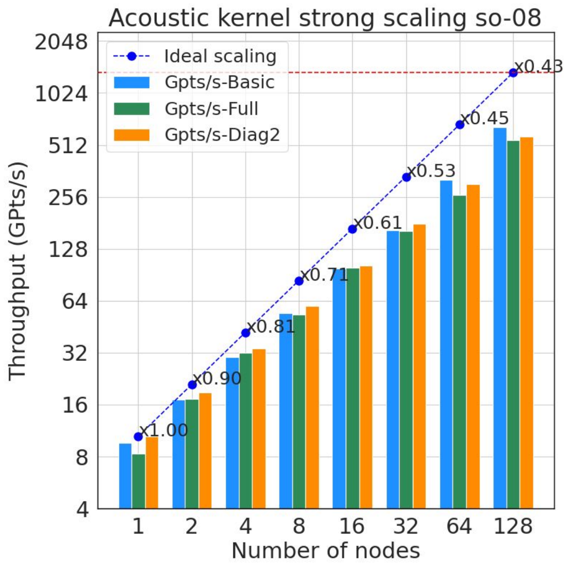
- Archer2 HPE Cray EX Supercomputer
- 128 dual AMD EPYC 7742 64-core 2.25GHz nodes
- 8 NUMA regions per node (16 cores per NUMA region)
- HPE Slingshot interconnect with 200 Gb/s signalling
- 8 MPI-ranks per node and 16 openmp workers per MPI rank, total of 128 cores per node
- Strong scaling up to 16384 cores

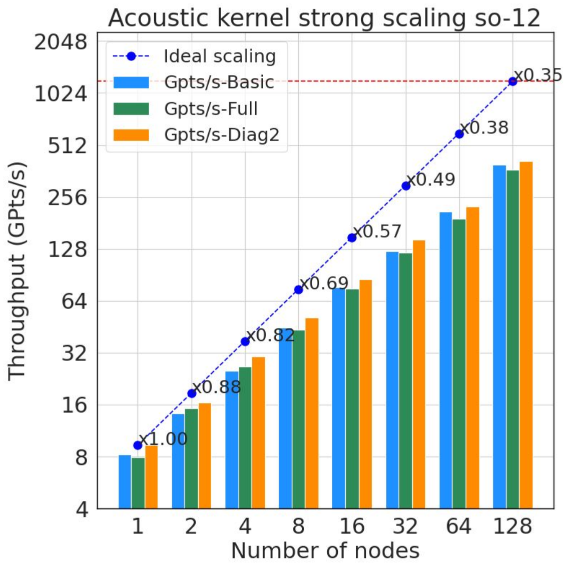
Isotropic acoustic wave propagation kernel

shape=[1024 1024 1024], timesteps=512,

with sponge-layers BCs



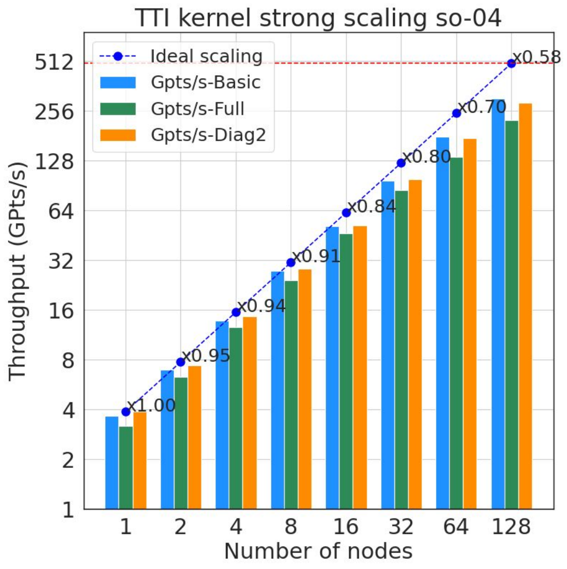


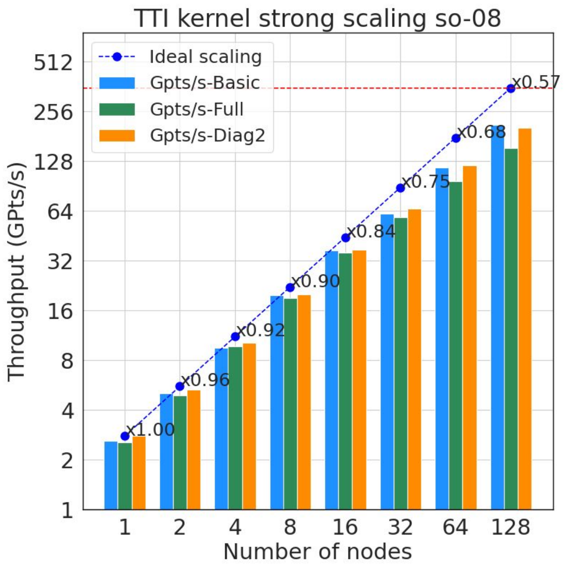


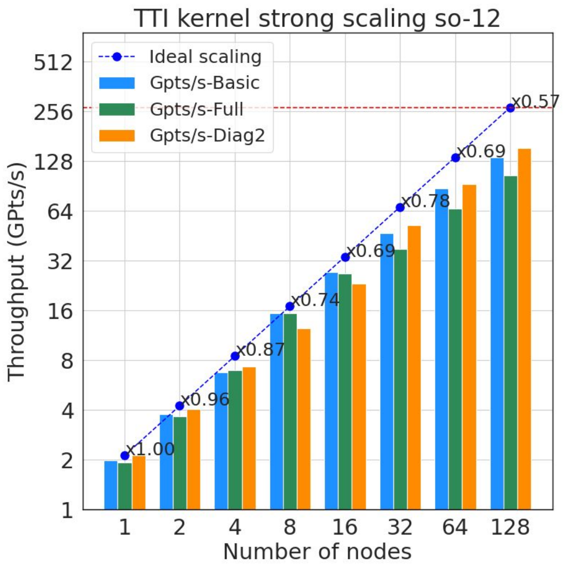
Performance evaluation: Strong scaling on Archer2

- Archer2 HPE Cray EX Supercomputer
- 128 dual AMD EPYC 7742 64-core 2.25GHz nodes
- 8 NUMA regions per node (16 cores per NUMA region)
- HPE Slingshot interconnect with 200 Gb/s signalling
- 8 MPI-ranks per node and 16 openmp workers per MPI rank, total of 128 cores per node
- Strong scaling up to 16384 cores

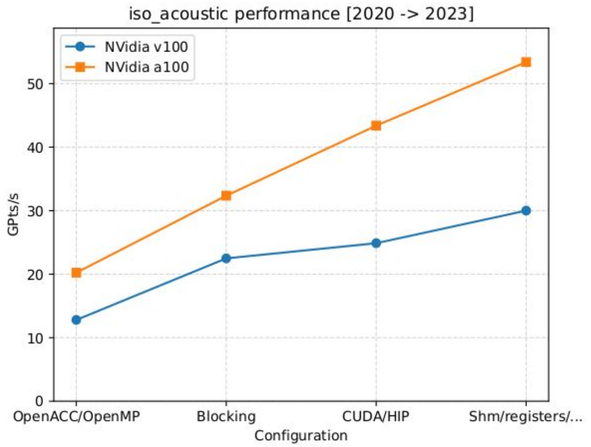
Anisotropic acoustic wave propagation kernel (TTI) (Zhang/Louboutin), shape=[1024 1024 1024] timesteps=512, with sponge-layers BCs

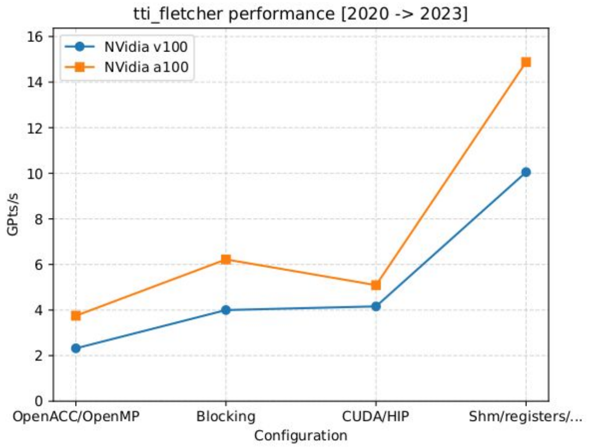


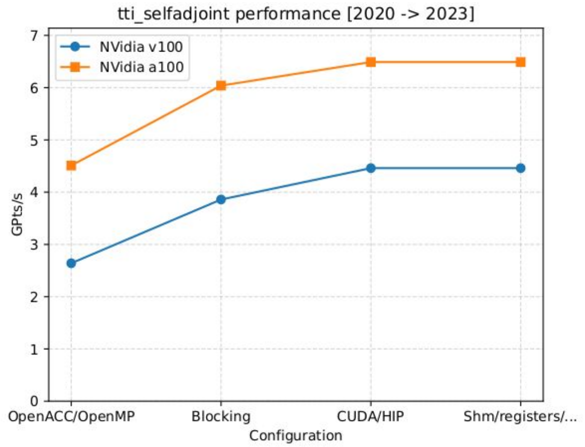


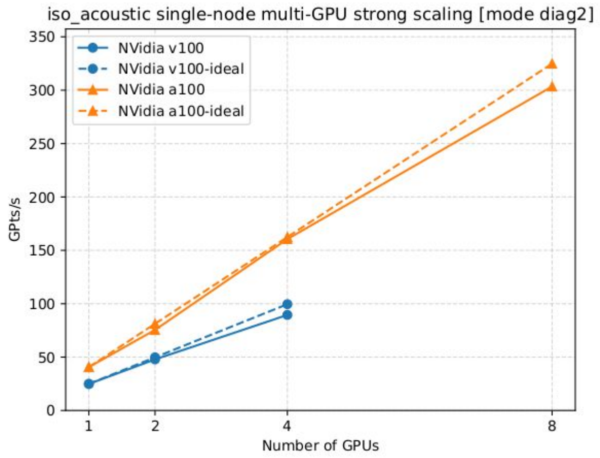


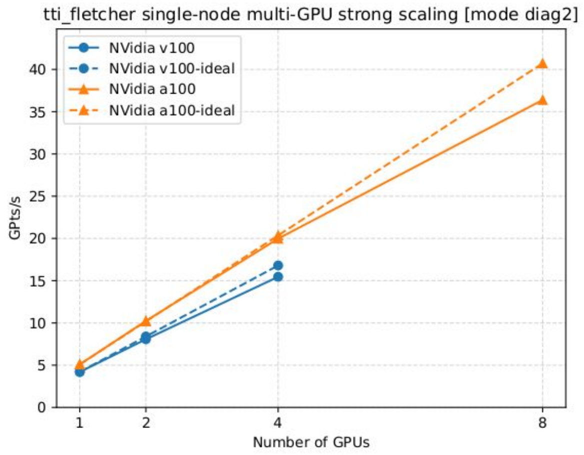
Performance evaluation: GPU results

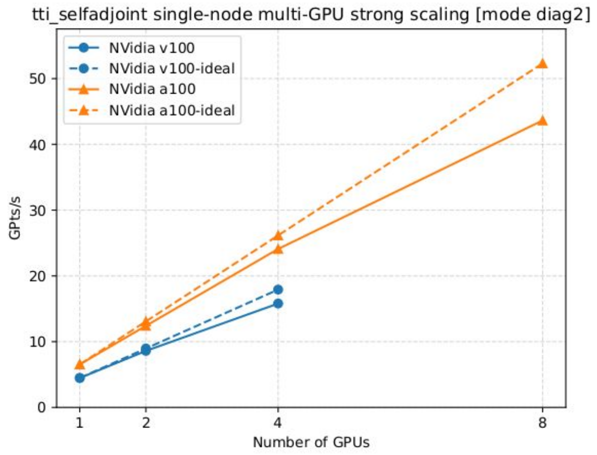












Conclusions

- We presented the **Devito DSL and Compiler framework** for stencil computation for **solving PDEs** using the **FD method on structured grids** (but not limited to them!)

- The Devito compiler supports a great variety of optimisations for stencil kernels, and support for shared- and distributed memory parallelism…all that…automatically and automagically!

- Performance results on UK's strongest SC show competitive strong scaling!

- Preliminary performance benchmarking on MPI+GPUs

- **Future work:**
  - further improve our MPI implementations for better scaling
  - Multi-node multi-GPU

- Website
- Slack
- Code

DEVITOPROJECT

Imperial College London

EPSRC Engineering and Physical Sciences Research Council

HiPEDS

DEVITO

References

- Luporini, F., Lange, M., Louboutin, M., Kukreja, N., Hückelheim, J., Yount, C., Witte, P.A., Kelly, P.H., Gorman, G., & Herrmann, F. (2020). Architecture and Performance of Devito, a System for Automated Stencil Computation. ACM Transactions on Mathematical Software (TOMS), 46, 1 - 28.

- Louboutin, M., M., Lange, F., Luporini, N., Kukreja, P. A., Witte, F. J., Herrmann, P., Velesko, and G. J., Gorman. "Devito (v3.1.0): an embedded domain-specific language for finite differences and geophysical exploration".Geoscientific Model Development 12, no.3 (2019): 1165–1187.

Appendix

From high to low…

```
# High-level DSL syntax
from devito import Eq, Grid,
TimeFunction, Operator

grid = Grid(shape=(4, 4))
u = TimeFunction(name='u', grid=grid,
space_order=2)
u.data[:] = 1

eq = Eq(u.forward, u.laplace + 1)
op = Operator(eq)
op.apply(time_M=3)
```

```
Groups of expressions,
Cluster-level
(Cluster([Eq(u[t1, x + 2, y + 2],
u[t0, x + 1, y + 2]/h_x**2 -
2.0*u[t0, x + 2, y + 2]/h_x**2 +
u[t0, x + 3, y + 2]/h_x**2 +
u[t0, x + 2, y + 1]/h_y**2 -
2.0*u[t0, x + 2, y + 2]/h_y**2 +
u[t0, x + 2, y + 3]/h_y**2 +
1)]),)
```

```
Groups of expressions,
Cluster-level (Optimized)
[Cluster([Eq(r0, 1/(h_x*h_x))
          Eq(r1, 1/(h_y*h_y))]),
Cluster([Eq(r2, -2.0*u[t0, x + 2,
y + 2])
          Eq(u[t1, x + 2, y + 2],
r0*r2 + r0*u[t0, x + 1, y + 2] +
r0*u[t0, x + 3, y + 2] + r1*r2 +
r1*u[t0, x + 2, y + 1] + r1*u[t0,
x + 2, y + 3] + 1)])]
```

```
<Callable Kernel>
  <CallableBody <allocs=0, casts=0, maps=0> <unmaps=0, frees=0>>
    <List (4, 0, 0)>
      <C.Comment /* Flush denormal numbers to zero in hardware */>
      <C.Statement
_MM_SET_DENORMALS_ZERO_MODE(_MM_DENORMALS_ZERO_O
N);>
      <C.Statement
_MM_SET_FLUSH_ZERO_MODE(_MM_FLUSH_ZERO_ON);>
      <C.Line >
    <List (0, 2, 0)>

      <ExpressionBundle (2)>

        <Expression r0 = 1/(h_x*h_x)>
        <Expression r1 = 1/(h_y*h_y)>

      <[affine,sequential] Iteration time::time::(time_m, time_M, 1)>
        <Section (section0)>

          <OverlappableHaloSpot(u)>
            <OmpRegion (1, 1, 0)>
              <C.Pragma #pragma omp parallel num_threads(nthreads)>
              <ParallelTree (0, 1, 0)>

                <[affine,collapsed[1],parallel] Iteration x::x::(x_m, x_M, 1)>
                  <[affine,parallel,vector-dim] Iteration y::y::(y_m, y_M, 1)>
                    <ExpressionBundle (2)>

                      <Expression r2 = -2.0*u[t0, x + 2, y + 2]>
                      <Expression u[t1, x + 2, y + 2] = r0*r2 + r0*u[t0, x + 1, y + 2]
+ r0*u[t0, x + 3, y + 2] + r1*r2 + r1*u[t0, x + 2, y + 1] + r1*u[t0, x + 2, y +
3] + 1>
```

Mapping from IET level to c-code

```
<Callable Kernel>
  <CallableBody <allocs=0, casts=0, maps=0> <unmaps=0,
frees=0>>
    <List (0, 2, 0)>

      <ExpressionBundle (2)>

        <Expression r0 = 1/(h_x*h_x)>
        <Expression r1 = 1/(h_y*h_y)>

      <[affine,sequential] Iteration time::time::(time_m, time_M, 1)>
        <Section (section0)>

          <HaloSpot(u)>
            <[affine,parallel] Iteration x::x::(x_m, x_M, 1)>
              <[affine,parallel] Iteration y::y::(y_m, y_M, 1)>
                <ExpressionBundle (2)>

                  <Expression r2 = -2.0*u[t0, x + 2, y + 2]>
                  <Expression u[t1, x + 2, y + 2] = r0*r2 + r0*u[t0, x + 1, y
+ 2] + r0*u[t0, x + 3, y + 2] + r1*r2 + r1*u[t0, x + 2, y + 1] + r1*u[t0, x
+ 2, y + 3] + 1>
```

```
int Kernel(const float h_x, const float h_y, struct dataobj *restrict u_ve
const int time_M, const int time_m, const int x_M, const int x_m, const
int y_M, const int y_m)
{
  r0 = 1.0F/(h_x*h_x);
  r1 = 1.0F/(h_y*h_y);

  for (int time = time_m, t0 = (time)%(2), t1 = (time + 1)%(2); time <=
time_M; time += 1, t0 = (time)%(2), t1 = (time + 1)%(2))
  {
    /* Begin section0 */
    for (int x = x_m; x <= x_M; x += 1)
    {
      for (int y = y_m; y <= y_M; y += 1)
      {
        r2 = -2.0F*u[t0][x + 2][y + 2];
        u[t1][x + 2][y + 2] = r0*r2 + r0*u[t0][x + 1][y + 2] + r0*u[t0][x +
3][y + 2] + r1*r2 + r1*u[t0][x + 2][y + 1] + r1*u[t0][x + 2][y + 3] + 1;
      }
    }
    /* End section0 */
  }
}
```

Mapping from IET level to c-code - Add denormals

```
<Callable Kernel>
  <CallableBody <allocs=0, casts=0, maps=0> <unmaps=0, frees=0>>
    <List (4, 0, 0)>
      <C.Comment /* Flush denormal numbers to zero in hardware */>
      <C.Statement
_MM_SET_DENORMALS_ZERO_MODE(_MM_DENORMALS_ZERO_ON);>
      <C.Statement _MM_SET_FLUSH_ZERO_MODE(_MM_FLUSH_ZERO_ON);>
      <C.Line >

    <List (0, 2, 0)>

      <ExpressionBundle (2)>

        <Expression r0 = 1/(h_x*h_x)>
        <Expression r1 = 1/(h_y*h_y)>

      <[affine,sequential] Iteration time::time::(time_m, time_M, 1)>
        <Section (section0)>

          <HaloSpot(u)>
            <[affine,parallel] Iteration x::x::(x_m, x_M, 1)>
              <[affine,parallel] Iteration y::y::(y_m, y_M, 1)>
                <ExpressionBundle (2)>

                  <Expression r2 = -2.0*u[t0, x + 2, y + 2]>
                  <Expression u[t1, x + 2, y + 2] = r0*r2 + r0*u[t0, x + 1, y + 2] + r0*u[t0, x
+ 3, y + 2] + r1*r2 + r1*u[t0, x + 2, y + 1] + r1*u[t0, x + 2, y + 3] + 1>
```

```
int Kernel(const float h_x, const float h_y, struct dataobj *restrict u_vec, const
int time_M, const int time_m, const int x_M, const int x_m, const int y_M,
const int y_m)
{
  /* Flush denormal numbers to zero in hardware */
  _MM_SET_DENORMALS_ZERO_MODE(_MM_DENORMALS_ZERO_ON);
_MM_SET_FLUSH_ZERO_MODE(_MM_FLUSH_ZERO_ON);

  r0 = 1.0F/(h_x*h_x);
  r1 = 1.0F/(h_y*h_y);

  for (int time = time_m, t0 = (time)%(2), t1 = (time + 1)%(2); time <= time_M;
time += 1, t0 = (time)%(2), t1 = (time + 1)%(2))
  {
    /* Begin section0 */
    for (int x = x_m; x <= x_M; x += 1)
    {
      for (int y = y_m; y <= y_M; y += 1)
      {
        r2 = -2.0F*u[t0][x + 2][y + 2];
        u[t1][x + 2][y + 2] = r0*r2 + r0*u[t0][x + 1][y + 2] + r0*u[t0][x + 3][y + 2] +
r1*r2 + r1*u[t0][x + 2][y + 1] + r1*u[t0][x + 2][y + 3] + 1;
      }
    }
    /* End section0 */
  }
}
```

Mapping from IET level to c-code - Add parallelism

```
<Callable Kernel>
  <CallableBody <allocs=0, casts=0, maps=0> <unmaps=0, frees=0>>
    <List (4, 0, 0)>
      <C.Comment /* Flush denormal numbers to zero in hardware */>
      <C.Statement
_MM_SET_DENORMALS_ZERO_MODE(_MM_DENORMALS_ZERO_ON);>
      <C.Statement _MM_SET_FLUSH_ZERO_MODE(_MM_FLUSH_ZERO_ON);>
      <C.Line >
    <List (0, 2, 0)>

      <ExpressionBundle (2)>

        <Expression r0 = 1/(h_x*h_x)>
        <Expression r1 = 1/(h_y*h_y)>

      <[affine,sequential] Iteration time::time::(time_m, time_M, 1)>
        <Section (section0)>

          <OverlappableHaloSpot(u)>
            <OmpRegion (1, 1, 0)>
              <C.Pragma #pragma omp parallel num_threads(nthreads)>
              <ParallelTree (0, 1, 0)>

                <[affine,collapsed[1],parallel] Iteration x::x::(x_m, x_M, 1)>
                  <[affine,parallel,vector-dim] Iteration y::y::(y_m, y_M, 1)>
                    <ExpressionBundle (2)>

                      <Expression r2 = -2.0*u[t0, x + 2, y + 2]>
                      <Expression u[t1, x + 2, y + 2] = r0*r2 + r0*u[t0, x + 1, y + 2] +
r0*u[t0, x + 3, y + 2] + r1*r2 + r1*u[t0, x + 2, y + 1] + r1*u[t0, x + 2, y + 3] + 1>
```

```
int Kernel(...)
{
  /* Flush denormal numbers to zero in hardware */
  _MM_SET_DENORMALS_ZERO_MODE(_MM_DENORMALS_ZERO_ON);
  _MM_SET_FLUSH_ZERO_MODE(_MM_FLUSH_ZERO_ON);

  r0 = 1.0F/(h_x*h_x);
  r1 = 1.0F/(h_y*h_y);

  for (int time = time_m, t0 = (time)%(2), t1 = (time + 1)%(2); time <= time_M; time +=
1, t0 = (time)%(2), t1 = (time + 1)%(2))
  {
    /* Begin section0 */
    #pragma omp parallel num_threads(nthreads)
    {
      #pragma omp for collapse(1) schedule(dynamic,1)
      for (int x = x_m; x <= x_M; x += 1)
      {
        #pragma omp simd aligned(u:32)
        for (int y = y_m; y <= y_M; y += 1)
        {
          r2 = -2.0F*u[t0][x + 2][y + 2];
          u[t1][x + 2][y + 2] = r0*r2 + r0*u[t0][x + 1][y + 2] + r0*u[t0][x + 3][y + 2] + r1*r2 +
r1*u[t0][x + 2][y + 1] + r1*u[t0][x + 2][y + 3] + 1;
        }
      }
    }
    /* End section0 */
  }
}
```

Mapping from IET level to c-code - Add parallelism

```
<Callable Kernel>
 <CallableBody <allocs=0, casts=0, maps=0> <unmaps=0, frees=0>>
  <List (4, 0, 0)>
   <C.Comment /* Flush denormal numbers to zero in hardware */>
   <C.Statement
_MM_SET_DENORMALS_ZERO_MODE(_MM_DENORMALS_ZERO_ON);>
   <C.Statement _MM_SET_FLUSH_ZERO_MODE(_MM_FLUSH_ZERO_ON);>
   <C.Line >
  <List (0, 2, 0)>

   <ExpressionBundle (2)>

    <Expression r0 = 1/(h_x*h_x)>
    <Expression r1 = 1/(h_y*h_y)>

   <[affine,sequential] Iteration time::time::(time_m, time_M, 1)>
    <Section (section0)>

     <OverlappableHaloSpot(u)>
      <OmpRegion (1, 1, 0)>
       <C.Pragma #pragma omp parallel num_threads(nthreads)>
       <ParallelTree (0, 1, 0)>

        <[affine,collapsed[1],parallel] Iteration x::x::(x_m, x_M, 1)>
         <[affine,parallel,vector-dim] Iteration y::y::(y_m, y_M, 1)>
          <ExpressionBundle (2)>

           <Expression r2 = -2.0*u[t0, x + 2, y + 2]>
           <Expression u[t1, x + 2, y + 2] = r0*r2 + r0*u[t0, x + 1, y + 2] +
r0*u[t0, x + 3, y + 2] + r1*r2 + r1*u[t0, x + 2, y + 1] + r1*u[t0, x + 2, y + 3] + 1>
```

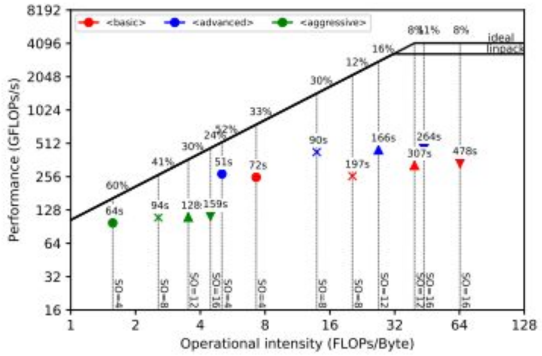
```
int Kernel(...)
{
 /* Flush denormal numbers to zero in hardware */
 _MM_SET_DENORMALS_ZERO_MODE(_MM_DENORMALS_ZERO_ON);
 _MM_SET_FLUSH_ZERO_MODE(_MM_FLUSH_ZERO_ON);

 r0 = 1.0F/(h_x*h_x);
 r1 = 1.0F/(h_y*h_y);

 for (int time = time_m, t0 = (time)%(2), t1 = (time + 1)%(2); time <= time_M; time +=
1, t0 = (time)%(2), t1 = (time + 1)%(2))
 {
  /* Begin section0 */
  #pragma omp parallel num_threads(nthreads)
  {
   #pragma omp for collapse(1) schedule(dynamic,1)
   for (int x = x_m; x <= x_M; x += 1)
   {
    #pragma omp simd aligned(u:32)
    for (int y = y_m; y <= y_M; y += 1)
    {
     r2 = -2.0F*u[t0][x + 2][y + 2];
     u[t1][x + 2][y + 2] = r0*r2 + r0*u[t0][x + 1][y + 2] + r0*u[t0][x + 3][y + 2] + r1*r2 +
r1*u[t0][x + 2][y + 1] + r1*u[t0][x + 2][y + 3] + 1;
    }
   }
  }
  /* End section0 */
 }
}
```

Pipeline for each target: CPU/OpenMP

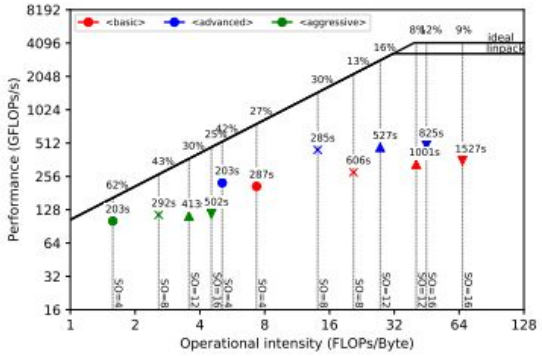
```c
int Kernel(const float h_x, const float h_y, struct dataobj *restrict u_vec, const int time_M, const int time_m, const int x_M, const int x_m, const int y_M, const int y_m, const int nthreads, struct profiler * timers)
{
  float (*restrict u)[u_vec->size[1]][u_vec->size[2]] __attribute__ ((aligned (64))) = (float (*)[u_vec->size[1]][u_vec->size[2]]) u_vec->data;

  /* Flush denormal numbers to zero in hardware */
  _MM_SET_DENORMALS_ZERO_MODE(_MM_DENORMALS_ZERO_ON);
  _MM_SET_FLUSH_ZERO_MODE(_MM_FLUSH_ZERO_ON);

  float r0 = 1.0F/(h_x*h_x);
  float r1 = 1.0F/(h_y*h_y);

  for (int time = time_m, t0 = (time)%(2), t1 = (time + 1)%(2); time <= time_M; time += 1, t0 = (time)%(2), t1 = (time + 1)%(2))
  {
    /* Begin section0 */
    START_TIMER(section0)
    #pragma omp parallel num_threads(nthreads)
    {
      #pragma omp for collapse(1) schedule(dynamic,1)
      for (int x = x_m; x <= x_M; x += 1)
      {
        #pragma omp simd aligned(u:32)
        for (int y = y_m; y <= y_M; y += 1)
        {
          float r2 = -2.0F*u[t0][x + 2][y + 2];
          u[t1][x + 2][y + 2] = r0*r2 + r0*u[t0][x + 1][y + 2] + r0*u[t0][x + 3][y + 2] + r1*r2 + r1*u[t0][x + 2][y + 1] + r1*u[t0][x + 2][y + 3] + 1;
        }
      }
    }
    STOP_TIMER(section0,timers)
    /* End section0 */
  }

  return 0;
}
```

Pipeline for each target: GPU/OpenACC

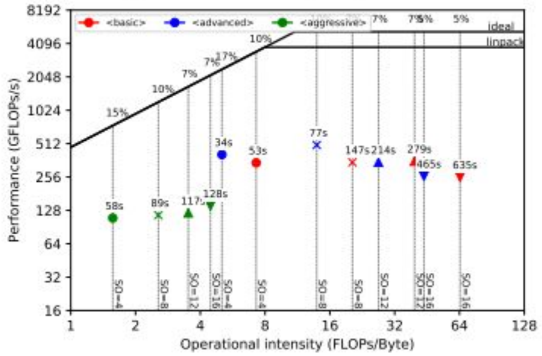
```c
int Kernel(const float h_x, const float h_y, struct dataobj *restrict u_vec, const int time_M, const int time_m, const int x_M, const int x_m, const int y_M, const int
y_m, const int deviceid, const int devicerm, struct profiler * timers)
{
  /* Begin of OpenACC setup */
  acc_init(acc_device_nvidia);
  if (deviceid != -1)
  {
    acc_set_device_num(deviceid,acc_device_nvidia);
  }
  /* End of OpenACC setup */

  float (*restrict u)[u_vec->size[ 1]][u_vec->size[ 2]] __attribute__ ((aligned ( 64))) = (float (*)[u_vec->size[ 1]][u_vec->size[ 2]]) u_vec->data;

  #pragma acc enter data copyin(u[0:u_vec->size[0]][0:u_vec->size[1]][0:u_vec->size[2]])

  float r0 = 1.0F/(h_x*h_x);
  float r1 = 1.0F/(h_y*h_y);

  for (int time = time_m, t0 = (time)%( 2), t1 = (time + 1)%(2); time <= time_M; time += 1, t0 = (time)%( 2), t1 = (time + 1)%(2))
  {
    /* Begin section0 */
    START_TIMER(section0)
    #pragma acc parallel loop collapse(2) present(u)
    for (int x = x_m; x <= x_M; x += 1)
    {
      for (int y = y_m; y <= y_M; y += 1)
      {
        float r2 = -2.0F*u[t0][x + 2][y + 2];
        u[t1][x + 2][y + 2] = r0*r2 + r0*u[t0][x + 1][y + 2] + r0*u[t0][x + 3][y + 2] + r1*r2 + r1*u[t0][x + 2][y + 1] + r1*u[t0][x + 2][y + 3] + 1;
      }
    }
    STOP_TIMER(section0,timers)
    /* End section0 */
  }

  #pragma acc exit data copyout(u[0:u_vec->size[0]][0:u_vec->size[1]][0:u_vec->size[2]])
  #pragma acc exit data delete(u[0:u_vec->size[0]][0:u_vec->size[1]][0:u_vec->size[2]])    if(devicerm)

  return 0;
}
```



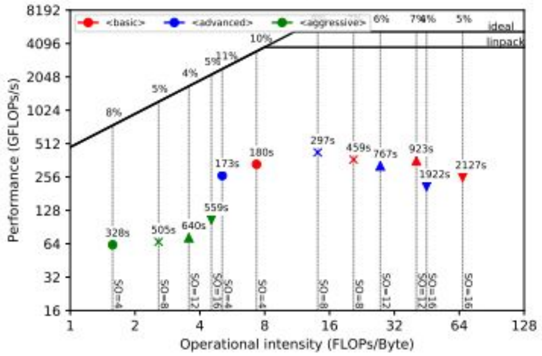
(a) skl8180, 512^3 grid points.



(b) skl8180, 768^3 grid points.



(c) knl7250, 512^3 grid points.



(d) knl7250, 768^3 grid points.

Fig. 6. Performance of `tti` on core for different architectures and grids.

Experimental evaluation: the models

- **Isotropic Acoustic**

  Generally known, single scalar PDE, laplacian like, low cost

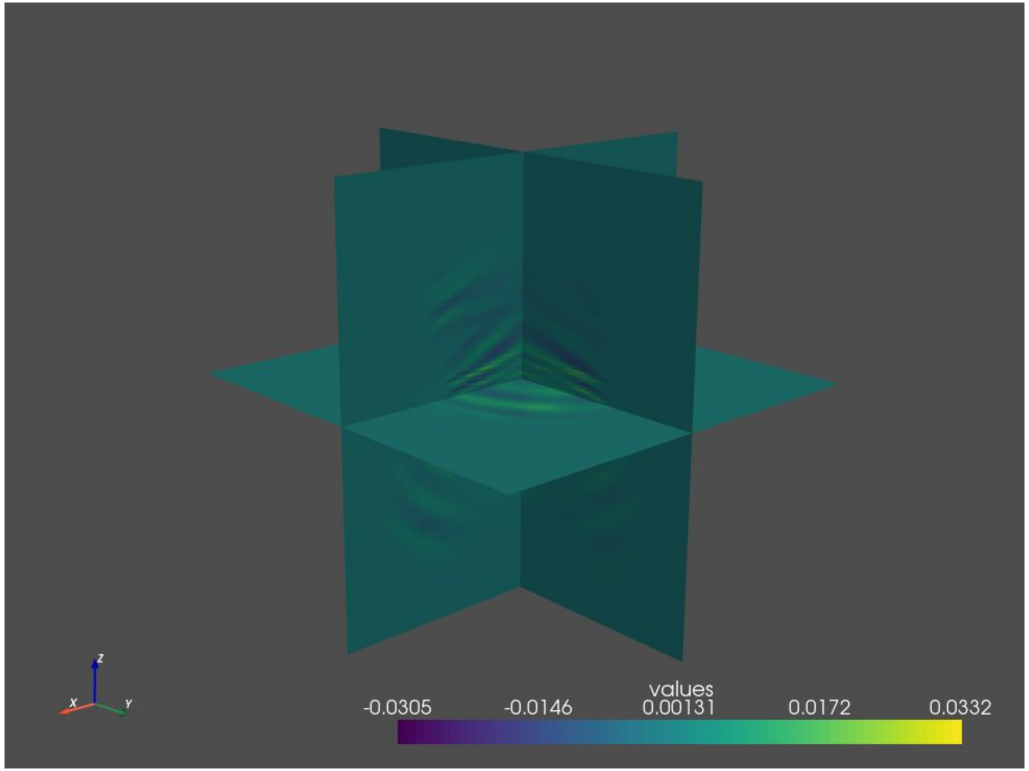
- **Isotropic Elastic**

  Coupled system of a vectorial and tensorial PDE, explosive source, increased data movement, first order in time, cross-loop data dependencies

- **Anisotropic Acoustic (aka TTI)**

  Industrial applications, rotated laplacian, coupled system of two scalar PDEs

Industrial-level, 512^3 grid points, 512ms simulation time, damping fields ABCs



Velocity field, TTI wave propagation after 512ms

Cache aware roofline model

From here: https://crd.lbl.gov/departments/computer-science/par/research/roofline/introduction/

Effects of Cache Behavior on Arithmetic Intensity
The Roofline model requires an estimate of total data movement. On cache-based architectures, the 3C's cache model highlights the fact that there can be more than simply compulsory data movement. Cache capacity and conflict misses can increase data movement and reduce arithmetic intensity. Similarly, superfluous cache write-allocations can result in a doubling of data movement. The vector initialization operation x[i]=0.0 demands one write allocate and one write back per cache line touched. The write allocate is superfluous as all elements of that cache line are to be overwritten. Unfortunately, the presence of hardware stream prefetchers can make it very difficult to quantify how much beyond compulsory data movement actually occurred.