IMPERIAL

DEVITO

Automated MPI-X code generation for scalable finite difference solvers

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

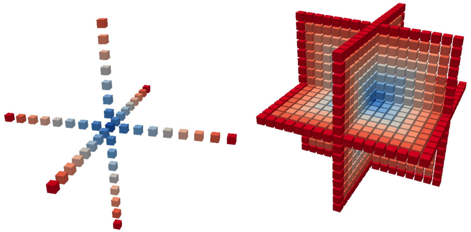
[1]*Imperial College London, Dept. of Computing*
[2]*Imperial College London, Dept. of Earth Sciences and Engineering*
[3]*Devito Codes, UK*

IPDPS 2025 • Milano, Italy

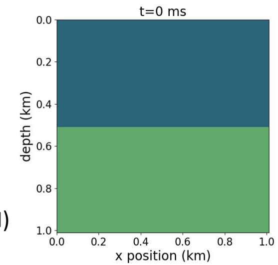
39th IEEE International Parallel & Distributed Processing Symposium June 3-7, 2025

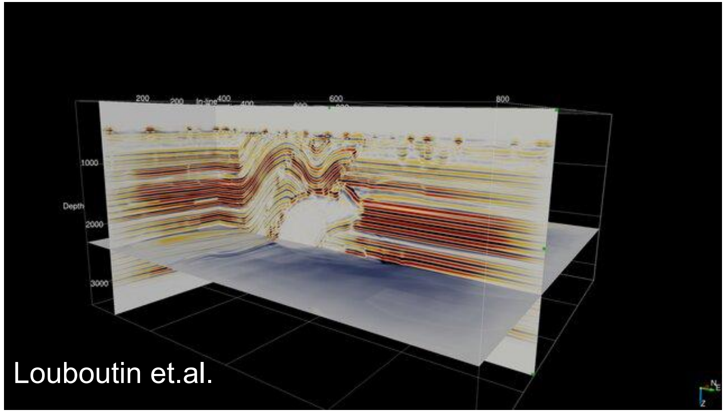
Motivation

- **PDEs** everywhere in **scientific modeling**
- Challenge: Building scalable, accurate, and performant HPC solvers is **complex & time-consuming**.
- Goal: **Automate** codegen for HPC-ready solvers. **Abstractions** for the win!



Isotropic acoustic and anisotropic acoustic (TTI)





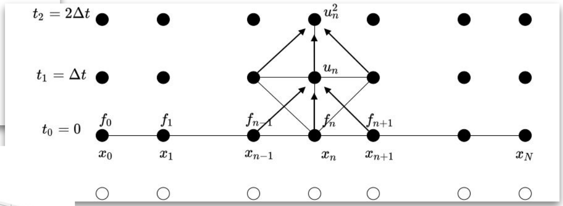
Louboutin et.al.

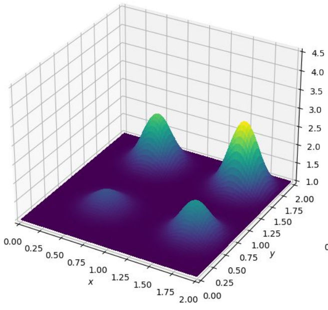
$$\frac{\partial^2 u}{\partial t^2} = c^2 \frac{\partial^2 u}{\partial x^2} \quad 0 < x < L$$

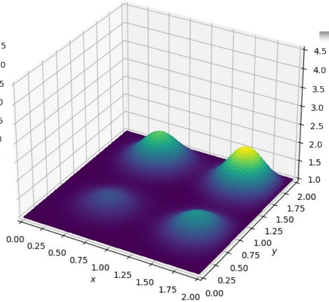
$$\text{Boundary Conditions :} \quad u(0,t) = 0; \quad u(L,t) = 0$$

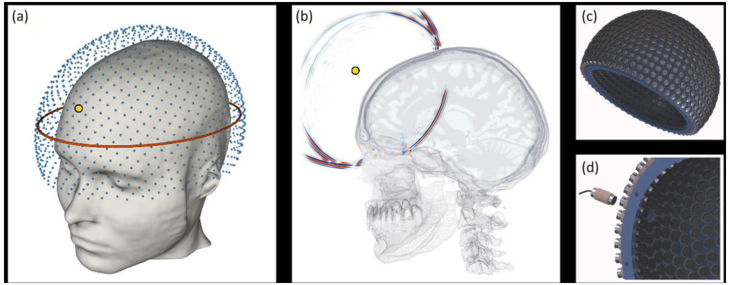
$$\text{Initial Conditions :} \quad u(x,0) = f(x)$$

$$\frac{\partial u}{\partial t}(x,0) = g(x)$$









Cueto et.al. (2022)

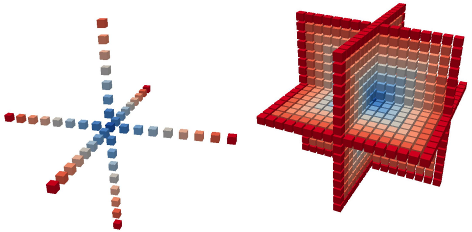
The Devito DSL and Compiler Framework

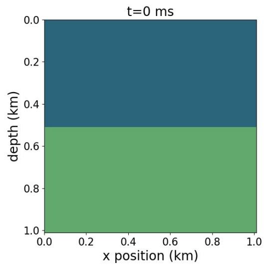
- **Devito is OSS, Python-embedded and** solved PDEs using the FD-method for structured grids
- Lots of users from **academia** and **industry**, interdisciplinary dev team and user-base, join our SLACK!
- Users leverage **high-level DSL** using symbolic math abstraction, and the **compiler auto-generates HPC optimized code.** Path to **Productivity**, **Performance** and **Portability**
- Real-world problem simulations (CFD, seismic/medical imaging, finance, tsunamis, planetary exploration, agriculture)
- High quality testing, performance regression, docker-ready (install now!)

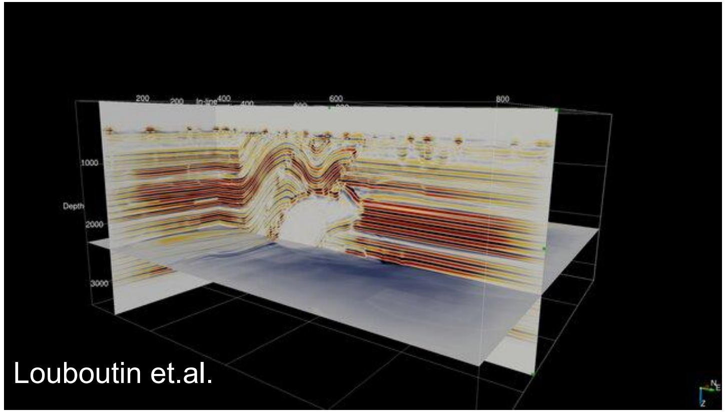
CI-core passing | CI-mpi passing | CI-gpu passing

codecov 92% | chat on slack | benchmarked by asv

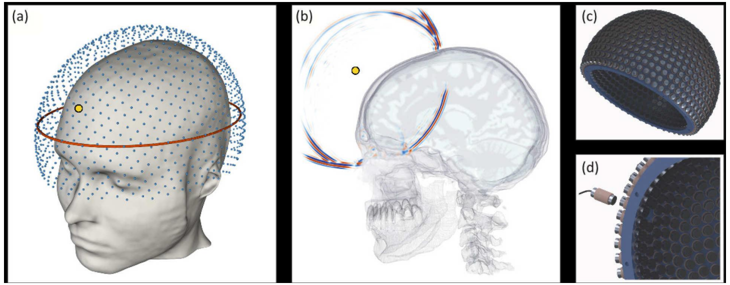
pypi package 4.8.16 | launch binder | docker images


Isotropic acoustic and anisotropic acoustic (TTI)





Louboutin et.al.



Cueto et.al. (2022)

Contributions

- **This paper is the result of the development, maintenance, optimization, and evaluation over 6 years of effort**

- We contribute abstractions for a **novel end-to-end automated MPI code generation** for real-world FD stencil computations. **NOT** toy benchmarks!

- Seamless integration of MPI with:
  - **OpenMP/OpenACC/(CUDA/HIP/SYCL in PRO)**
  - advanced **cache-blocking**, **flop-reduction and many other** optimizations.

- Comprehensive benchmarking:
  - Four wave propagators with varying memory and computational needs
  - Strong and weak scaling on 128 CPU nodes (16384 cores) and 128 GPUs

$$m\frac{\partial^2 u}{\partial t^2} + \eta\frac{\partial u}{\partial t} - \Delta u = 0$$

User writes:

```
eqn = M * u.dt2 + eta
* u.dt - u.laplace
```

DEVITO

```
void kernel(...)
{...}
```

A full PDE solver in a few lines of Devito DSL

```
# No-MPI
$ python myscript.py
# With-MPI (2 ranks)
$ DEVITO_MPI=basic mpirun -n 2 python myscript.py
# MPI + GPU ready
# …add DEVITO_PLATFORM=nvidia DEVITO_COMPILER=nvc
```

```
nx, ny = 4, 4                  # Define the structured grid
nu = .5
dx, dy = 2. / (nx - 1), 2. / (ny - 1)
sigma = .25
dt = sigma * dx * dy / nu
grid = Grid(shape=(nx, ny), extent=(2., 2.))

u = TimeFunction(name="u", grid=grid, space_order=2)
u.data[1:-1,1:-1] = 1

eq = Eq(u.dt, u.laplace)
stencil = solve(eq, u.forward)       # Define the equations to be solved
eq_stencil = Eq(u.forward, stencil)


op = Operator([eq_stencil])          # Generate C-code using the Devito compiler,
op.apply(time_M=1, dt=dt)            # JIT-compiler and run
```

(Dense) Data Access: Support for distributed NumPy arrays

```
nx, ny = 4, 4
nu = .5
dx, dy = 2. / (nx - 1), 2. / (ny - 1)
sigma = .25
dt = sigma * dx * dy / nu
grid = Grid(shape=(nx, ny), extent=(2., 2.))
```

Define the structured grid

```
u = TimeFunction(name="u", grid=grid, space_order=2)
u.data[1:-1,1:-1] = 1

eq = Eq(u.dt, u.laplace)
stencil = solve(eq, u.forward)
eq_stencil = Eq(u.forward, stencil)
```

Define the equations to be solved

```
op = Operator([eq_stencil])
op.apply(time_M=1, dt=dt)
```

Generate C-code using the Devito compiler, JIT-compiler and run

- The data is **physically distributed**, but from the user's perspective, it remains a **logically centralized** entity!
- User interaction with data using **familiar indexing schemes** (e.g., slicing) without concern about the underlying layout.
- All works via global-to-local index conversion.

```
[stdout:0]              [stdout:1]
[[0.00 0.00]            [[0.00 0.00]
 [0.00 1.00]]            [1.00 0.00]]

[stdout:2]              [stdout:3]
[[0.00 1.00]            [[1.00 0.00]
 [0.00 0.00]]            [0.00 0.00]]
```

(Dense) Data Access: Support for distributed NumPy arrays

```
nx, ny = 4, 4          # Define the structured grid
nu = .5
dx, dy = 2. / (nx - 1), 2. / (ny - 1)
sigma = .25
dt = sigma * dx * dy / nu
grid = Grid(shape=(nx, ny), extent=(2., 2.))

u = TimeFunction(name="u", grid=grid, space_order=2)
u.data[1:-1,1:-1] = 1

eq = Eq(u.dt, u.laplace)          # Define the equations to be solved
stencil = solve(eq, u.forward)
eq_stencil = Eq(u.forward, stencil)


op = Operator([eq_stencil])       # Generate C-code using the Devito compiler,
op.apply(time_M=1, dt=dt)         # JIT-compiler and run
```

- The data is **physically distributed**, but from the user's perspective, it remains a **logically centralized** entity!
- User interaction with data using **familiar indexing schemes** (e.g., slicing) without concern about the underlying layout.
- All works via global-to-local index conversion.

```
[stdout:0]              [stdout:1]
[[0.50 -0.25]           [[-0.25 0.50]
 [-0.25 0.50]]           [0.50 -0.25]]

[stdout:2]              [stdout:3]
[[-0.25 0.50]           [[0.50 -0.25]
 [0.50 -0.25]]           [-0.25 0.50]]
```

(Sparse) Data Access: Support for distributed NumPy arrays

```
nx, ny = 4, 4
nu = .5
dx, dy = 2. / (nx - 1), 2. / (ny - 1)
sigma = .25
dt = sigma * dx * dy / nu
grid = Grid(shape=(nx, ny), extent=(2., 2.))

u = TimeFunction(name="u", grid=grid, space_order=2)
u.data[1:-1,1:-1] = 1

eq = Eq(u.dt, u.laplace)
stencil = solve(eq, u.forward)
eq_stencil = Eq(u.forward, stencil)
sf = SparseFunction(name="sf", grid=grid,...)

op = Operator([eq_stencil])
op.apply(time_M=1, dt=dt)
```

Define the structured grid

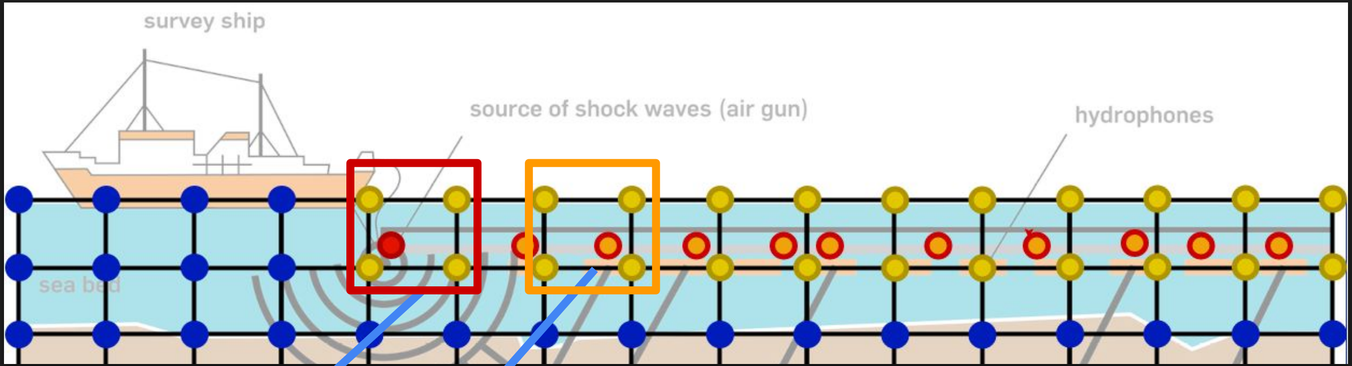
Define the equations to be solved

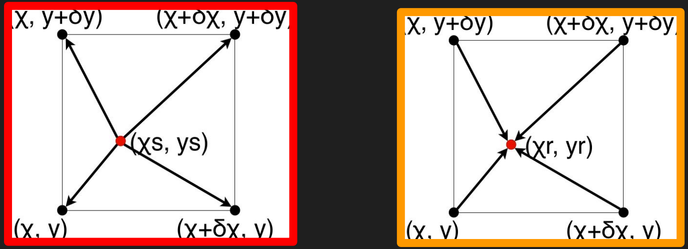
It is more than “just” stencils

Generate C-code using the Devito compiler, JIT-compiler and run

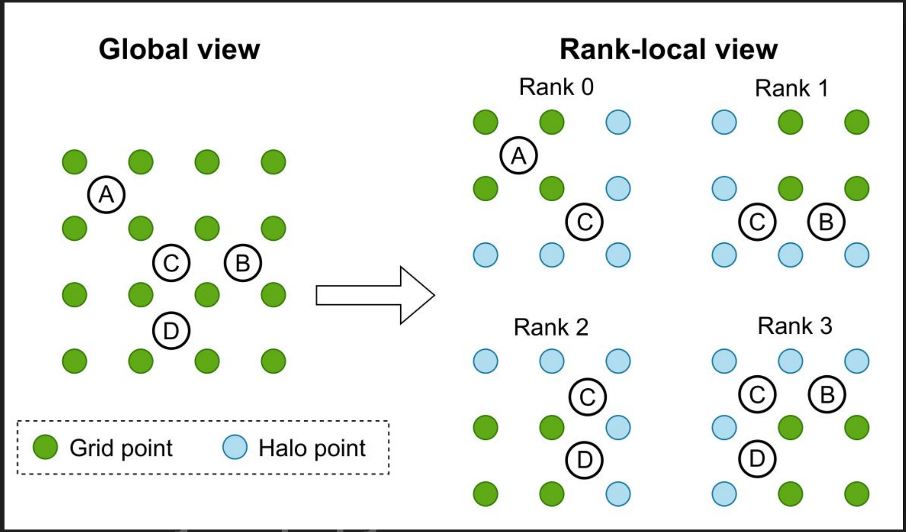
- Handling **--non-aligned to the FD-grid--** data
- scatter/gather operations with dependencies spanning over different ranks
- Sources/Receivers
- Boundary conditions
- Subdomains
- More complex geometries
- **We handle more than “just” FD-stencils!**

(Sparse) Data Access: Support for distributed NumPy arrays





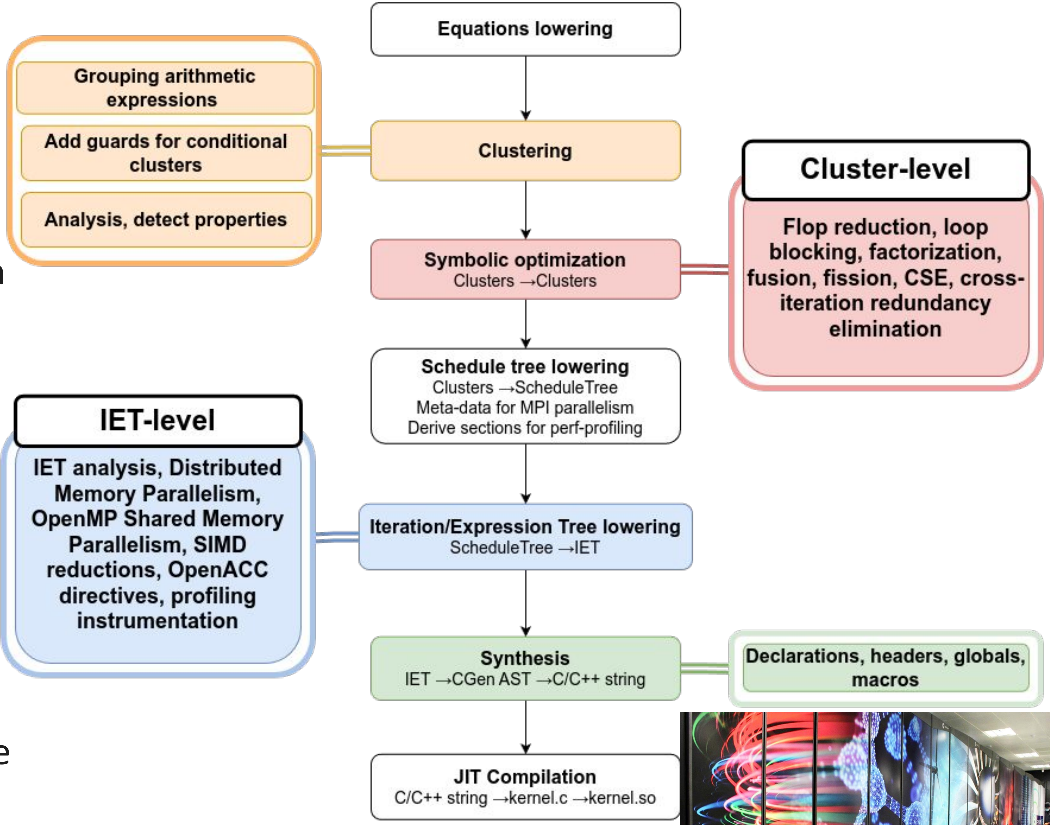
Source injection/Receiver interpolation



The multi-step compilation process

```
eqn = M * u.dt2 + eta
* u.dt - u.laplace
```

- From symbolic representation to HPC code
- Optimizations at different Intermediate Representation (IR) levels.
- **Cluster-level IR:**
  - "Cluster"s symbolic expressions based on computational properties.
  - Advanced data dependence analysis. Incorporates DMP/MPI <u>analysis</u>
  - Reduces arithmetic intensity via loop motion, blocking, factorization, etc.
- **Iteration/Expression Tree (IET) IR:**
  - Establishes control flow (loops and expressions).
  - Optimizations tailored to target hardware (SIMD, OpenMP, OpenACC).
  - Incorporates DMP/MPI <u>synthesis</u>



Halo exchanges: an **Analysis** and Synthesis approach

```
<Callable Kernel>
 <Expression r0 = 1/dt>
 <Expression r1 = 1/(h_x*h_x)>
 <Expression r2 = 1/(h_y*h_y)>

 <Iteration time...>


   <Iteration x...>
    <Iteration y...>
     <Expression r3 = -2.0*u[t0,x + 2,y + 
2]>
     <Expression u[t1, x + 2, y + 2] =
            dt*(r0*u[t0, x + 2, y + 2] +
...)>
```

- Check expression accesses (reads/writes)

  $u \Rightarrow W : (t1, x+2, y+2)$

  $R : (t0, x+1, y+2)$
  $(t0, x+2, y+3)$
  $(t0, x+2, y+2)$
  $(t0, x+2, y+1)$
  $(t0, x+3, y+2)$

  Are exchanges required? Where?

- Place “exchange hints” at the IR
- Optimize communications (**drop**, **merge**, or **move** HaloSpots)
- Lower HaloSpots to MPI Calls using their metadata

Halo exchanges: an Analysis and **Synthesis** approach

```
<Callable Kernel>
 <Expression r0 = 1/dt>
 <Expression r1 = 1/(h_x*h_x)>
 <Expression r2 = 1/(h_y*h_y)>

 <Iteration time...>

  <HaloSpot(u)>
   <Iteration x...>
    <Iteration y...>
     <Expression r3 = -2.0*u[t0,x + 2,y +
2]>
     <Expression u[t1, x + 2, y + 2] =
              dt*(r0*u[t0, x + 2, y + 2] +
...)>
```

- Check expression accesses (reads/writes)

  u => W : (t1, x + 2, y + 2)
  R : (t0, x + 1, y + 2)
  (t0, x + 2, y + 3)
  (t0, x + 2, y + 2)
  (t0, x + 2, y + 1)
  (t0, x + 3, y + 2)

  Are exchanges required? Where?

- Place “exchange hints” at the IR
- Optimize communications (**drop**, **merge**, or **move** HaloSpots)
- Lower HaloSpots to MPI Calls using their metadata

Halo exchanges: an Analysis and **Synthesis** approach

```
<Callable Kernel>
 <Expression r0 = 1/dt>
 <Expression r1 = 1/(h_x*h_x)>
 <Expression r2 = 1/(h_y*h_y)>

 <Iteration time...>

  <HaloUpdateCall>
   <Iteration x...>
    <Iteration y...>
     <Expression r3 = -2.0*u[t0,x + 2,y +
2]>
     <Expression u[t1, x + 2, y + 2] =
              dt*(r0*u[t0, x + 2, y + 2] +
...)>
```

- Check expression accesses (reads/writes)

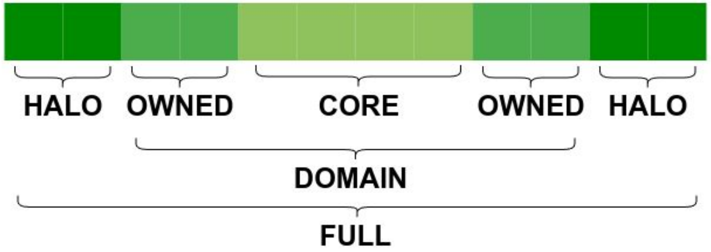
  $u \Rightarrow\ W: (t1, x+2, y+2)$
  $R: (t0, x+1, y+2)$
  $(t0, x+2, y+3)$
  $(t0, x+2, y+2)$
  $(t0, x+2, y+1)$
  $(t0, x+3, y+2)$

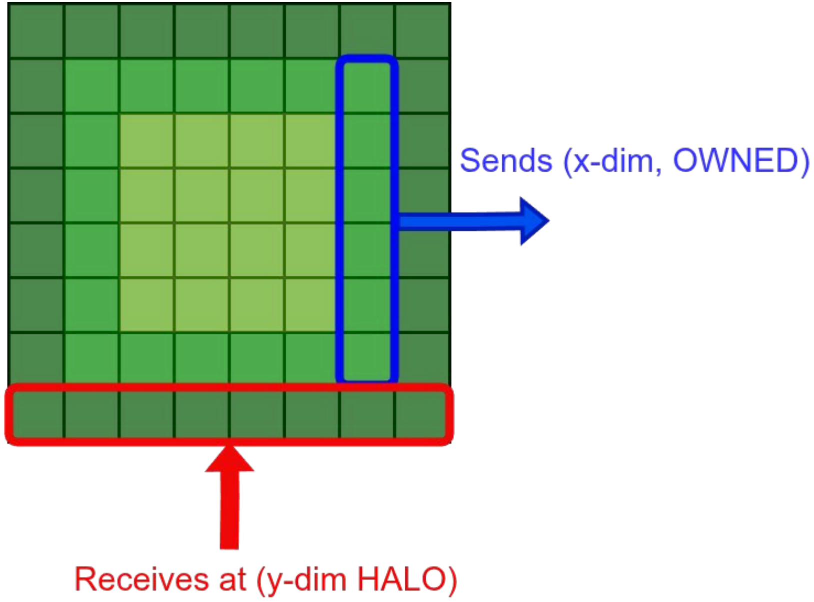
  Are exchanges required? Where?

- Place “exchange hints” at the IR
- Optimize communications (**drop**, **merge**, or **move** HaloSpots)
- Lower HaloSpots to MPI Calls using their metadata

Composing computation/communication patterns

- **Aliases** for data regions help reason about WHO (ranks) send/receive WHAT (data) to who
  1D example:





- **Neighborhoods** help to form the communication grid (left/right/diagonal) communications

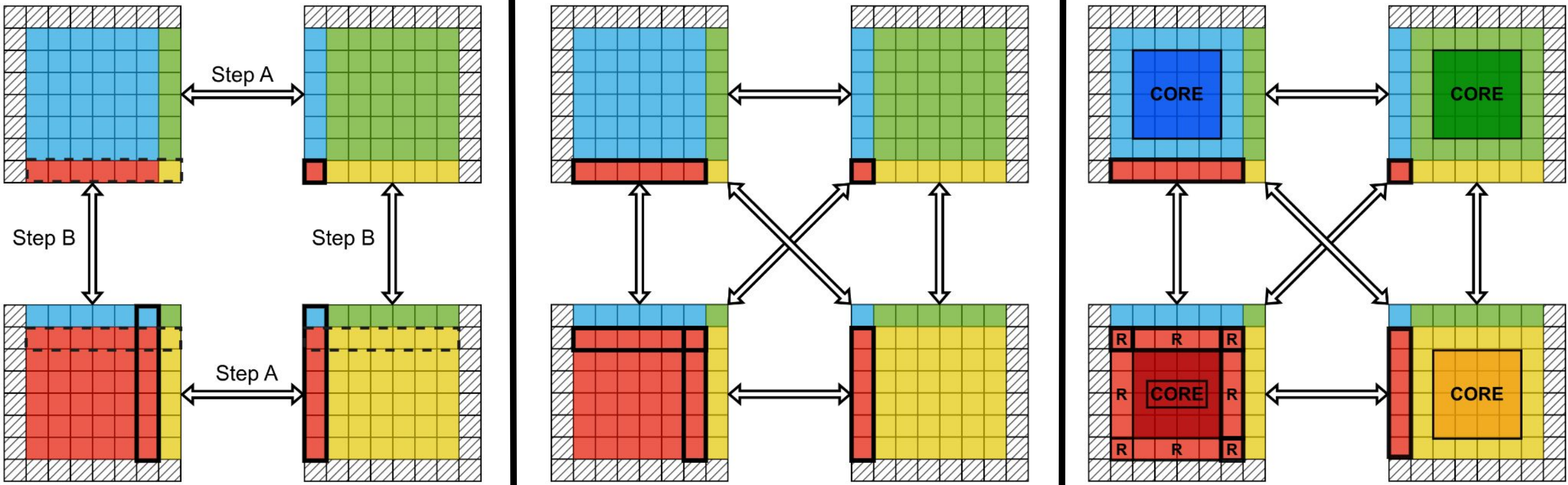
- Help in easily parametrizing MPICall codegen by determining **message size, sender, recipient etc..**

Figure style influenced from Li et.al, ICPP 2021

Supported computation/communication patterns

Users only have to add: “ *DEVITO_MPI=<mode> mpirun -n #nranks* python my_devitoscript.py ”

MPI mode	Target	Communication	Message batches	#messages (3D)	Buffer allocation
Basic	CPU, GPU	Sync, No comp overlap	Multi-step	6	runtime (C/C++)
Diagonal	CPU	Sync, No comp overlap	Single-step	26	pre-alloc (Python)
Full	CPU	ASync, comp overlap	Single-step	26	pre-alloc (Python)



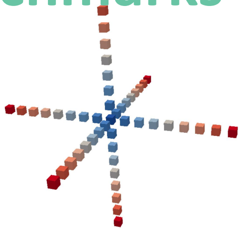
Basic

Diagonal

Full (Com(m/p)) overlap)

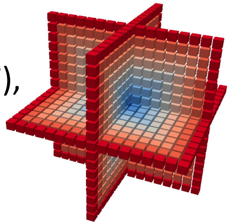
Performance evaluation: the benchmarks

- **Isotropic Acoustic**
  Low-cost (OI: 2.64), low communication needs (5 fields)

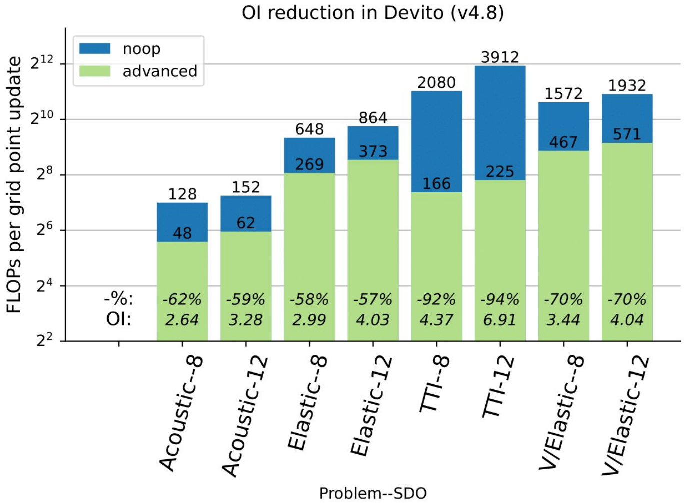


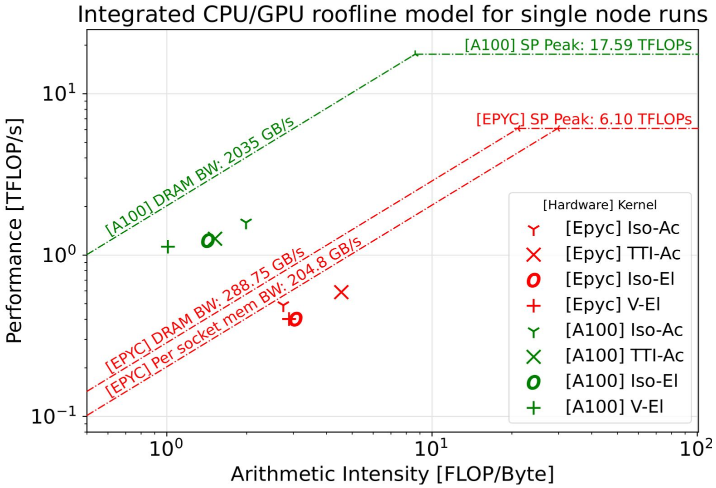
- **Isotropic Elastic**
  Higher-flops (OI: 2.99), increased data movement, high flops (22 fields)

- **Anisotropic Acoustic (aka TTI, Zhang-Louboutin variation)**
  Industrial applications, highest arithmetic intensity (OI: 4.37), (12 fields)



- **Isotropic elastic with viscosity:**
  High fidelity modelling, (OI:3.44) the highest memory footprint (36 fields)

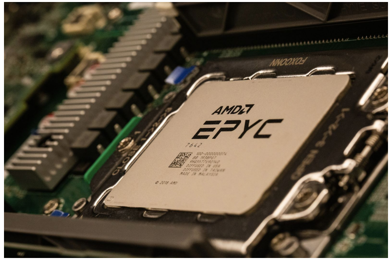




🚧 NOT the typical jacobian low-order stencils!
✅ Significantly reduced operational intensity!

Performance evaluation: CPU strong scaling (Archer2-UK HPC)

- Dual-socket AMD Zen2 EPYC 7742 (64 cores, 2.25 GHz)
- 128 cores per node (8 NUMA regions, 16 cores/NUMA)
- 32KB L1, 512KB L2 cache/core, 16MB L3 cache/4 cores
- HPE Slingshot interconnect (200 Gb/s, dragonfly topology)
- Cray Clang 11.0.4, Cray MPICH
- 8 MPI ranks/node, 16 OpenMP workers/rank (128 threads/node)
- Strong/weak scaling up to 128 nodes (16,384 cores)



- Space discretization order: 8
- Largest models fitting single-node memory
- 512 ms simulation time
- Metric: Throughput (GPts/s)

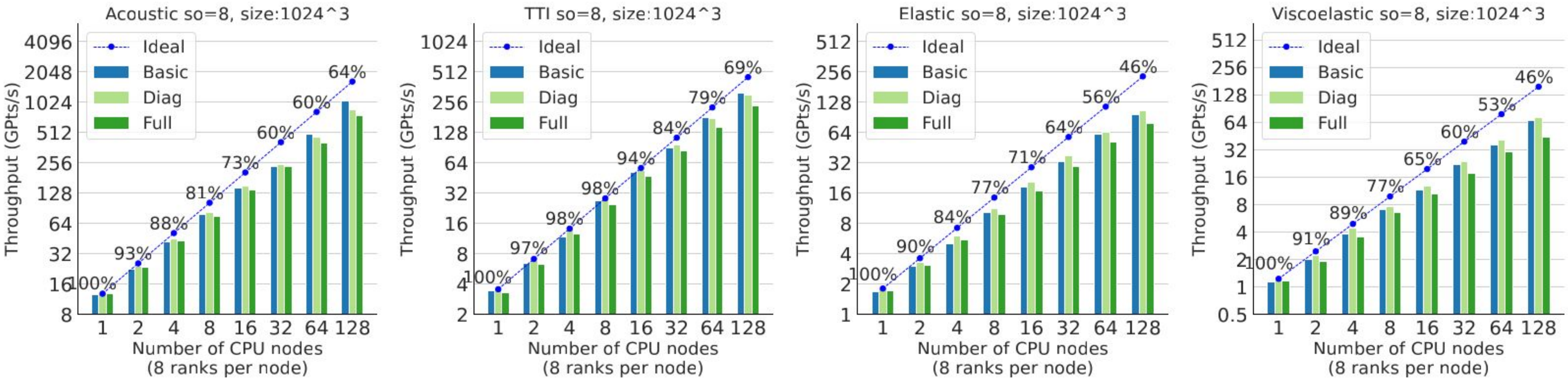
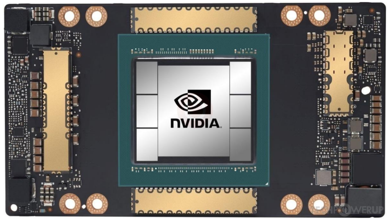


Figure 3: Numbers on the ideal line show the percentage of the achieved ideal efficiency (Gpts/s for N nodes)/((Gpts/s for 1 node) * N).

Performance evaluation: GPU strong scaling (DiRAC: Tursa-EPCC)

- 2x AMD 7413 EPYC 24c processor
- 4x NVIDIA Ampere A100-80 GPUs with NVLink
- Peak FP32: 19.5 TFLOPS, 80GB HBM2e memory per GPU
- 4x 200 Gbps NVIDIA Infiniband interfaces
- nvc++ 23.5-0 compiler
- Strong/weak scaling up to 32 nodes (128 A100-80 GPUs)



- Space discretization order: 8
- Largest models fitting single-node memory
- 512 ms simulation time
- Metric: Throughput (GPts/s)

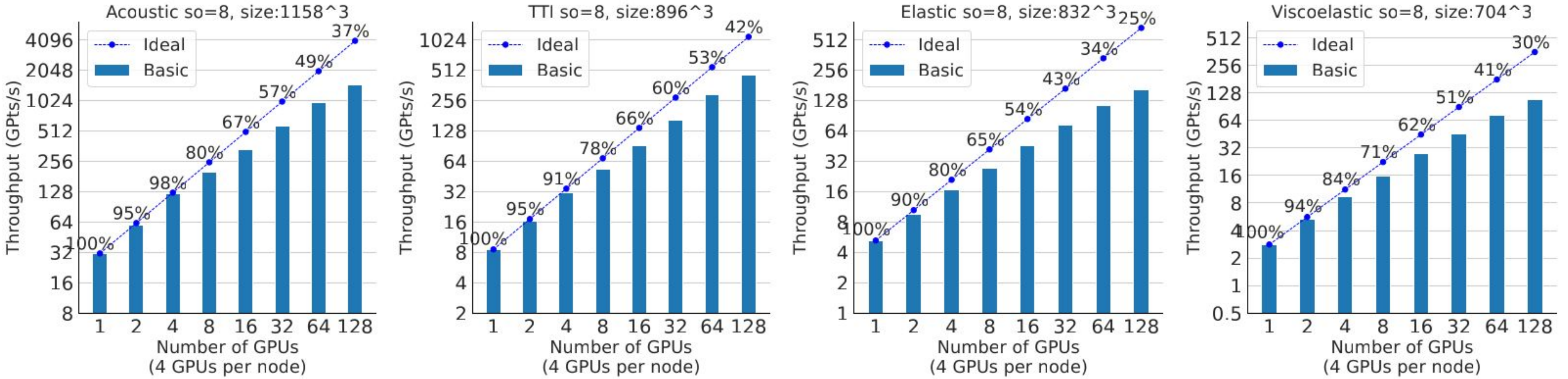


Figure 4: Numbers on the ideal line show the percentage of the achieved ideal efficiency (Gpts/s for N GPUs)/((Gpts/s for 1 GPU) * N).

Conclusions - Limitations - Future work

- We show abstractions that generate speed-of-light HPC code
  - Enables complex simulations with high-level symbolic math
  - Compiler approach automates MPI code generation for PDEs
  - Seamless portability to HPC clusters
  - Competitive throughput and scaling on CPU/GPU clusters
- Future work:
  - Maintenance
  - Performance Optimization
- Check the full paper for more details!
- Nominated for the IPDPS 2025 Open Source Contribution Award

- Website
- Slack
- Code

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.

- Bisbas G., Luporini F., Louboutin M., Nelson R., Gorman G., and Kelly P. H.J. (2021) Temporal blocking of finite-difference stencil operators with sparse” off-the-grid” sources. IEEE International Parallel and Distributed Processing Symposium (IPDPS) (pp. 497-506). IEEE.

- Example code available: demo_laplace.c

- MPI tutorial: MPI Jupyter Notebook