DEVITO V4.3: PRODUCTION-GRADE MULTI-GPU SUPPORT

Fabio Luporini\textsuperscript{1}, Rhodri Nelson\textsuperscript{2}, George Bisbas\textsuperscript{2}, Italo Assis\textsuperscript{4}, Ken Hester\textsuperscript{3}, Gerard Gorman\textsuperscript{1,2}

1. Devito Codes
2. Imperial College London
3. NVidia
4. Federal University of Rio Grande do Norte
Traditional approach to solving PDEs

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

void kernel(...) {
    ...
    <impenetrable code with aggressive performance optimizations>
    ...
}
Traditional approach to solving PDEs

MATH

CODE

Space = physics $\times$ discretization $\times$ architecture $\times$ language $\times$ developers

Huge space $\Rightarrow$ Huge cost
Raising the level of abstraction

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

void kernel(...) {
  ...
  <impenetrable code with aggressive performance optimizations>
  ...
}

Raising the level of abstraction

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

void kernel(...) {
  ...
  <impenetrable code with aggressive performance optimizations>
  ...
}
Raising the level of abstraction

\[ m \frac{\partial^2 u}{\partial t^2} + \eta \frac{\partial u}{\partial t} - \Delta u = 0 \]
Raising the level of abstraction

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

eqn = m * u.dt2 + eta * u.dt - u.laplace
Raising the level of abstraction

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

eqn = m * u.dt2 + eta * u.dt - u.laplace

void kernel(...) { ... }
Raising the level of abstraction

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

Devito

eqn = m * u.dt2 + eta * u.dt - u.laplace

void kernel(...) { ... }
Devito: a DSL and compiler for explicit finite differences

- **Open source platform** – MIT license.

- **Python** package — easy to learn

- **Devito is a compiler** that generates optimized parallel code.
  - Supported languages:
    - \{C, SIMD, OpenMP, OpenACC\} + MPI
  - Supported architectures:
    - CPUs: Intel, AMD, ARM
    - GPUs: NVidia, AMD

- **Composability: integrate with existing codes and AI/ML**
  - Works out-of-the-box with other popular packages from the Python ecosystem (e.g. PyTorch, NumPy, Dask, TensorFlow)

- **Best practises software engineering** (testing, CI/CD, …)

- **Cloud ready**
Target applications

- **Seismic imaging**
  - FWI, RTM, LS-RTM (TTI, elastic, visco-elastic propagators, etc.)

- Now maturating strong interest in **medical imaging**

- Generation of high performance **neural networks**

- **CFD problems** in renewable energy

- Black-Scholes in **finance**

- Virtually any partial differential equations on structured grids; more generally, any sort of stencil code
Devito on GPUs

• Implementation needs to take into account:
  • Support for multiple target languages
    • OpenMP, OpenACC
    • potentially: CUDA, HIP, SYCL, …
  • Unreliability of the target languages’ software stack
  • Multi-GPU support:
    • Make it possible to run different shots on different GPUs
    • Single-node multi-GPU via domain decomposition
    • Multi-node multi-GPU via domain decomposition
  • Data movement
  • Data streaming
  • Kernel performance (e.g., register optimization)

This is already quite hard…
… But much harder is the **automation**!

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

\[
eqn = m * u.dt2 + \text{eta} * u.dt - u.laplace
\]

**The user expresses the mathematical operators; the same exact DSL code needs to run efficiently on different architectures**
The key is decomposition

- Compilation is a hard problem

- The key to success is decomposition: a hard problem is decomposed into many — more manageable and simpler — subproblems

- Here the hard problem is the generation of efficient GPU code

- The subproblems are a series of compilation passes

- Each compilation pass in isolation doesn’t do much. But altogether they solve the problem while ensuring **maintainability** and **extendibility**.
Example: forward propagation with CPU-GPU data streaming

\[ m \cdot u_{\text{dt}^2} + \eta \cdot u_{\text{dt}} - u_{\text{laplace}} = 0 \]

\[
\ldots
\]

\[
\ldots
\]

\[
\ldots
\]

\[
\text{usave} = u
\]

\[
\ldots
\]
Example: forward propagation with CPU-GPU data streaming

Compiler pass 1: buffering to decouple CPU-GPU execution

\[ m \times u.dt^2 + \eta \times u.dt - u.laplace = 0 \]

\[
\begin{align*}
\text{ubuffer} &= u \\
\text{usave} &= \text{ubuffer}
\end{align*}
\]

Too large for the GPU memory; it will reside on the host
Example: forward propagation with CPU-GPU data streaming

Compiler pass 1: buffering to decouple CPU-GPU execution

\[ m \cdot u \cdot dt^2 + \eta \cdot u \cdot dt - u \cdot \text{laplace} = 0 \]

**GPU**

\text{(thread}_0)\]

\[ \text{ubuffer} = u \]

\[ \text{...} \]

**CPU**

\text{(thread}_1)\]

\[ \text{usave} = \text{ubuffer} \]

\[ \text{...} \]
Example: forward propagation with CPU-GPU data streaming

Compiler pass 2: analysis and placement of synchronizations

\[ m \times u.\text{dt}^2 + \eta \times u.\text{dt} - u.\text{laplace} = 0 \]

**GPU** (thread\(0\))

\[ \text{ubuffer} = u \]

\[ <\text{wait(lock)}> \]

**CPU** (thread\(1\))

\[ \text{usave} = \text{ubuffer} \]

\[ <\text{unset(lock)}> \]
Example: forward propagation with CPU-GPU data streaming

Compiler pass 3: lowering into Abstract Syntax Trees

<loop nest>

GPU (thread_0)
while(lock == 0);

<loop nest>

-----------------------------
while(flag != 0)

CPU (thread_1)
<loop nest>
lock = 2;
Example: forward propagation with CPU-GPU data streaming

Compiler pass 4: specialization for the target language

```c
<loop nest>

GPU (thread0)
while(lock == 0);

<loop nest>

while(flag != 0)

CPU (thread1)
#pragma acc update self(... ubuffer ...)

lock = 2;
```
Performance of iso-acoustic benchmark

• Achieved performance
  • 27 GPoints/s
  • This corresponds to slightly less than 1 Teraflops/s
  • The measured arithmetic intensity is 1.5. This means ~53% of the attainable peak

• Benchmark details:
  • Benchmark: O(2, 8), 512³ grid points, 150 timesteps, single precision, NO data streaming
  • System: NVidia V100, nvc 20.9 compiler, NSight Compute for the roofline
  • Optimization: OpenACC, tuned thread block size, all divisions lifted, all arithmetic redundancies eliminated (factorization, time-invariants, etc), constant folding (where reasonable)

• Bottleneck
  • Register pressure => affects occupancy
  • This is an aggressively optimized implementation with OpenACC; we’ll probably need to use a lower level language to push it even higher on the roofline
Sponsors who supported this work

• DUG
• BP
• Shell
• Microsoft
• NVidia
• Intel

• Thanks to our many collaborators and contributors. For a full list of contributors for each release please see https://github.com/devitocodes/devito/releases
GPU support roadmap

- Support for multiple target languages
  - OpenMP, OpenACC
  - potentially: CUDA, HIP, SYCL, …
- Unreliability of the target languages’ software stack
- Multi-GPU support:
  - Make it possible to run different shots on different GPUs
  - Single-node multi-GPU via domain decomposition
  - Multi-node multi-GPU via domain decomposition
- Data movement (optimized)
- Data streaming (optimized)
- Kernel performance (best so far: 27 GPOINTS on iso-acoustic O(2, 8))

Legend:
- **Done**
- **Nearly done**
- **In progress**
- Potentially later this year