Experiments in Unstructured Mesh Finite Element CFD Using CUDA

Graham Markall

Software Performance Optimisation Group
Imperial College London
http://www.doc.ic.ac.uk/~grm08
grm08@doc.ic.ac.uk
Joint work with David Ham and Paul Kelly

December 2009
Problem statement:

**Fluidity** – Finite Element Unstructured Mesh CFD Code
- Applied Modelling & Computation Group, Dept. of Earth Science & Engineering, Imperial College
- Used in Imperial College Ocean Model (ICOM)
- Fortran/C++ source... `wc -l`: >340672

How do we exploit multicore architectures to improve the performance of Finite Element Assembly?

1. Write code for today’s multicore architecture (CUDA)
2. Test, debug, optimise...
3. New multicore architecture: go to Step 1
Proposed Solution:

- Provide **hardware-independent abstraction** for the specification of finite element methods. Advantages:
  - Future proofing of code
  - Easier development
  - Faster!

- A two-part study (so far):
  1. CUDA Implementation of test problems – this talk
  2. Implementing a *Domain Specific Language* compiler

- Talk structure:
  1. The Finite Element Method
  2. Test Problems
  3. Translation Methodology
  4. Performance Results
  5. Optimisation
A Brief Overview of the Finite Element Method

\[ L(u) = q \quad \int_{\Omega} vL(u^\delta) \, dX = \int_{\Omega} vq \, dX \]
A Brief Overview of the Finite Element Method

\[ L(u) = q \int_{\Omega} vL(u^\delta) \, dX = \int_{\Omega} vq \, dX \]

Domain:
A Brief Overview of the Finite Element Method

\[ L(u) = q \quad \int_{\Omega} vL(u^\delta) \, dX = \int_{\Omega} vq \, dX \]

Domain:

Do element \( = 1, N \)
Assemble(element)
End do
A Brief Overview of the Finite Element Method

\[ L(u) = q \int_{\Omega} vL(u^\delta) \, dX = \int_{\Omega} vq \, dX \]

Domain:

Do element = 1, N
Assemble(element)
End do

Evaluate Integral
(Gaussian Quadrature)
A Brief Overview of the Finite Element Method

\[ L(u) = q \quad \int_{\Omega} vL(u^\delta) \, dX = \int_{\Omega} vq \, dX \]

Domain:

- Do \( 1 \leq \text{element} \leq N \)
- Assemble(\text{element})
- Evaluate Integral (Gaussian Quadrature)
- Evaluate \( q \) at nodes
- End do
A Brief Overview of the Finite Element Method

\[ L(u) = q \rightarrow \int_{\Omega} vL(u^\delta) \, dX = \int_{\Omega} vq \, dX \]

Domain:

Evaluated at nodes

Evaluate Integral (Gaussian Quadrature)

Compressed Sparse Row Matrix

Vector
A Brief Overview of the Finite Element Method

\[ L(u) = \int_{\Omega} vL(u^\delta) \, dX = \int_{\Omega} vq \, dX \]

Domain:

Evaluate Integral (Gaussian Quadrature)

Compressed Sparse Row Matrix

Evaluate q at nodes

Do element = 1, N

Assemble(element)

End do
A Brief Overview of the Finite Element Method

\[ L(u) = q \quad \int_{\Omega} vL(u^\delta) \, dX = \int_{\Omega} vq \, dX \]

Domain:

\[ \text{Do element } i = 1, N \]
\[ \text{Assemble(element)} \]
\[ \text{End do} \]

Evaluate Integral (Gaussian Quadrature)

Evaluate \( q \) at nodes

Compressed Sparse Row Matrix

\[ Ax = b \]

Solve for \( x \)

(CG, GMRES)
The Test Problem

- Test_advection_diffusion:
  \[
  \frac{\partial T}{\partial t} + \mathbf{u} \cdot \nabla T = \nabla \cdot \bar{\mathbf{u}} \cdot \nabla T
  \]

- Solved using a split scheme:
  - Advection: Explicit RK4
  - Diffusion: Implicit theta scheme

- Linear Solver:
  - CUDA Conjugate Gradient Solver [1]
Porting FE Assembly

An Assembly Loop in Fortran:

1 Element

Transform reference element → Compute Local Matrix → Local mat. add to Global mat. → Compute Local Vector → Local vec. add to Global vec.

1 Result

Assembly Loop in CUDA:

Transform reference element → Compute Local Matrix → Local mat. add to Global mat. → Compute Local Vector → Local vec. add to Global vec.

Thousands Of Elements

GPU Memory
Porting \texttt{test_advection_diffusion}

- **Timestepping loop:**
  - Output of solve input to next Assemble

- **CUDA:**
  - Upload initial conditions
  - Iterate timestepping loop
  - Transfer solution when required

- Assemble & Solve Advection (x4)
- Assemble & Solve Diffusion
- Start
- Exit

- Transfer initial conditions to GPU
- Assemble & solve advection (x4)
- Fast execution
- Slow transfers
- Transfer solution to host (optional)
- Exit
Obtaining coalescing:

- One thread per element
- Unstructured meshes
Coalescing with Unstructured Meshes

- Obtaining coalescing:
- One thread per element
- Unstructured meshes
Coalescing with Unstructured Meshes

Obtaining coalescing:

- One thread per element
- Unstructured meshes

![Diagram showing coalescing with unstructured meshes](image)
Coalescing with Unstructured Meshes

- Obtaining coalescing:

  - One thread per element
  - Unstructured meshes
Coalescing with Unstructured Meshes

- Obtaining coalescing:
  - One thread per element
  - Unstructured meshes

- "Side-by-side" layout
- Achieves coalescing
- Destroys temporal locality
- Expansion 2D: 6, 3D: 24
- Alternative: partitioning
Performance Optimisations

- Texture Memory for matrix sparsity

- Specialisation of Kernels (reduced register usage)

```c
for(int x=0; x<nodes; x++) {
    for(int y=0; y<nodes; y++) {
        ...;
    }
}
```

```c
for(int x=0; x<3; x++) {
    for(int y=0; y<3; y++) {
        ...;
    }
}
```
Performance testing setup

- Nvidia 280GTX – 1GB RAM (use Tesla C1060 for 4GB)
- Intel Core 2 Duo E8400 @ 3.00GHz
- Double Precision arithmetic
- Run problem for 200 timesteps

- 2GB RAM in host machine
- Intel C++ and Fortran Compilers V10.1 - V11.0 suffers from bugs and cannot compile Fluidity
- Five runs of each problem - averages reported
- CPU Implementations compiled with -O3 flags
- CUDA Implementation compiled using NVCC 2.2
- Increasingly finer meshes with increasing element count
Preliminary Results: Assembly time

- Assembly phase for advection-diffusion for 200 timesteps

![Graph showing assembly time for different number of elements and number of cores.](image-url)
Preliminary results: Overall speedup

- Assembly and solve for advection only, diffusion only, and advection-diffusion for 200 timesteps
### Q: Which kernels should we focus on optimising?

<table>
<thead>
<tr>
<th>Kernel</th>
<th>GPU Time (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>matrix_addto</td>
<td>50-40</td>
</tr>
<tr>
<td>matrix_addto_sum_diag</td>
<td>20-30</td>
</tr>
<tr>
<td>vector_addto</td>
<td>10-20</td>
</tr>
<tr>
<td>matrix_addto_diffusion</td>
<td>5-10</td>
</tr>
<tr>
<td>dshape_dot_vector_shape</td>
<td>5-10</td>
</tr>
<tr>
<td>shape_shape</td>
<td>5-10</td>
</tr>
<tr>
<td>transform_to_physical</td>
<td>5-10</td>
</tr>
<tr>
<td>advection_rhs</td>
<td>5-10</td>
</tr>
<tr>
<td>scatter_rhs_values</td>
<td>5-10</td>
</tr>
<tr>
<td>dshape_tensor_dshape</td>
<td>5-10</td>
</tr>
<tr>
<td>diffusion_rhs</td>
<td>5-10</td>
</tr>
<tr>
<td>daxpy</td>
<td>5-10</td>
</tr>
<tr>
<td>memcpyDtoD_aligned</td>
<td>5-10</td>
</tr>
<tr>
<td>gather_rhs_values</td>
<td>5-10</td>
</tr>
</tbody>
</table>

### A: Addto kernels: 84% of execution time
Impact of Atomic Operations

Colouring in a high-order FE earthquake simulation on CUDA: [2]
Can we do any better? (1/5)

Global Assembly:

\[
M = A^T M^e A \\
b = A^T b^e
\]
Can we do any better? (2/5)

- More efficiently, on the CPU:

```
1 2 3
\Omega^1 \Omega^2

1 2
\Omega^1 \Omega^2
```

- "Addto" in Fluidity:

```
M = 0;
foreach Element e do
  for i ← 1 to Ne do
    for j ← 1 to Ne do
      M[map[e][i], map[e][j]] += M^e[i, j];
```
Can we do any better? (3/5)

- Why do we assemble M?
  - SpMV: \( y = Mv \)

- Don’t do global assembly: when the solver needs the SpMV product, compute:
  \[
  y = \left( A^T \left( M^e (A \nu) \right) \right)
  \]

- Increase in work \( \sim \) multiplicity of nodes

- We still need global assembly of \( b \):
  \[
  b = A^T b^e
  \]
Can we do any better? (4/5)

- Effect on the assembly time/SpMV time:

<table>
<thead>
<tr>
<th>Elements</th>
<th>$A^T M^e A v$</th>
<th>$M v$</th>
</tr>
</thead>
<tbody>
<tr>
<td>28710</td>
<td>$1.91 \times 10^6$</td>
<td>$8.48 \times 10^5$</td>
</tr>
<tr>
<td>331714</td>
<td>$2.45 \times 10^7$</td>
<td>$9.85 \times 10^6$</td>
</tr>
</tbody>
</table>

- Assembly: 4x faster - SpMV: 2.5x slower
Can we do any better? (5/5)

Total simulation times:

Conclusion: Don’t do global assembly on the GPU (in 2D)!
Conclusions & Further Work

- Up to **12x** overall speedup over dual-core machine
- Choice of algorithm is **device dependent**
- Further performance gains from:
  - Fusing kernels [3] – **which kernels?**
  - Mesh reordering & Partitioning [4]
  - Integration with a better CUDA SpMV [5]

- **Unified Form Language** Compiler, supporting:
  - Automatic generation of CUDA kernels
  - Automatic generation of marshalling code
  - Automated exploration of optimisations
  - Multiple Backends

**Manifesto**: Integration of UFL code into the Fluidity codebase, to generate **highly optimised code** for solving complex multiphase problems
References


Spare Slides

Equation: $\Delta u = f$

Weak form: $\int_{\Omega} \nabla v \cdot \nabla u \, dX = \int_{\Omega} vf \, dX$

/* P is a finite element space */

v=TestFunction(P)
u=TrialFunction(P)
f=Function(Psi, "sin(x[0])+cos(x[1])")
A=dot(grad(v),grad(u))*dx
RHS=v*f*dx
P = Solve(A,RHS)

- Ease of use – close to math. notation
- “What”, not “how” - future proof
- Code generation:
  - Multiple backends
  - Choice of algorithm/optimisation

Prototype Implementation

- " Parses" UFL using \texttt{ufl.algorithms} package (FEniCS)
- Creates of a graph representing the assembly:

\begin{figure}
\centering
\begin{tikzpicture}
  % Graph nodes
  \node (ufl) at (0,0) {UFL};
  \node (frontend) at (2,0) {Frontend};
  \node (backend) at (4,0) {Backend};
  \node (cuda) at (6,0) {CUDA};

  % Edges
  \draw[->] (ufl) -- (frontend) node[midway,above] {\texttt{IR}};
  \draw[->] (frontend) -- (backend);
  \draw[->] (backend) -- (cuda);

  % Subgraph
  \node (v) at (2,-2) {$v$};
  \node (u) at (4,-2) {$u$};
  \node (divv) at (2,-4) {$\nabla v$};
  \node (divu) at (4,-4) {$\nabla u$};
  \node (divv_divu) at (3,-6) {$\nabla v \cdot \nabla u$};
  \node (a) at (3,-8) {A};

  % Edges in subgraph
  \draw[->] (v) -- (divv);
  \draw[->] (u) -- (divu);
  \draw[->] (divv) -- (divv_divu);
  \draw[->] (divu) -- (divv_divu);
  \draw[->] (divv_divu) -- (a);
\end{tikzpicture}
\end{figure}
Prototype testing

Input:

```python
P = state.scalar_fields["psi"]
v = TestFunction(P)
u = TrialFunction(P)
f = Function(P)
f.name="shape_rhs"
A = dot(grad(v),grad(u))*dx
solve(P, A, f)
```

CUDA Kernel reference

- **Bodge**: the prototype could not output code to perform arbitrary functions: relied on a library of kernels
- **Upside**: Allows “low-level” non-standard operations to be implemented (femtools, CUDA, Fortran, C++, Python?)
What did we learn from the prototype?

- Generation of CUDA from UFL is feasible

**Wishlist:**
- Automatic generation of CUDA kernels
- Automatic generation of marshalling code
- Automated exploration of optimisations
  - e.g. choice of algorithm
- Other backends should be possible
  - Cell, multicore x86/SSE, Larrabee etc.

**Idea:**
- IR should capture **expressions** (declarative)
- Not **instructions** (imperative)
\[
\int_\Omega \nabla v \cdot \nabla u + \lambda vu \; d\Omega = \int_\Omega vf \; d\Omega
\]
\[ \int_{\Omega} (\nabla v \cdot \nabla u + \lambda vu) \, dX = \int_{\Omega} vf \, dX \]
\[ \int_\Omega \nabla v \cdot \nabla u + \lambda vu \, dX = \int_\Omega vf \, dX \]
\[ \int_\Omega \nabla v \cdot \nabla u + \lambda uv \, dX = \int_\Omega vf \, dX \]
\[ \int_{\Omega} \nabla v \cdot \nabla u + \lambda vu \, dX = \int_{\Omega} vf \, dX \]
\[ \int_\Omega \nabla v \cdot \nabla u + \lambda vv \, dX = \int_\Omega vf \, dX \]
Intermediate Representation

\[ \int_{\Omega} \nabla v \cdot \nabla u + \lambda v u \, dX = \int_{\Omega} v f \, dX \]
\[ \int_{\Omega} \nabla v \cdot \nabla u + \lambda uv \, dX = \int_{\Omega} v f \, dX \]

Diagram:
- Begin
  - Solve: 1
    - dX
    - Return 1 into psi
  - Product
    - Test Function: psi
    - Function 0: psi
  - Sum
    - dX
  - Product
    - Test Function: psi
    - Trial Function: psi
    - Grad
      - Test Function: psi
    - Trial Function: psi
    - Grad
      - Trial Function: psi
    - Product
      - 20
      - Test Function: psi
\[ M = p \cdot q \, dx \]

\[ \text{rhs} = dt \cdot \text{dot} (\text{grad}(q), u) \cdot t \cdot dx \]
\[ t_1 = \text{solve} (M, \text{rhs}) \]

\[ \text{rhs} = dt \cdot \text{dot} (\text{grad}(q), (0.5 \cdot u + 0.5 \cdot \text{unew})) \cdot (t + 0.5 \cdot t_1) \cdot dx \]
\[ t_2 = \text{solve} (M, \text{rhs}) \]

\[ \text{rhs} = dt \cdot \text{dot} (\text{grad}(q), (0.5 \cdot u + 0.5 \cdot \text{unew})) \cdot (t + 0.5 \cdot t_2) \cdot dx \]
\[ t_3 = \text{solve} (M, \text{rhs}) \]

\[ \text{rhs} = dt \cdot \text{dot} (\text{grad}(q), (0.5 \cdot u + 0.5 \cdot \text{unew})) \cdot (t + t_3) \cdot dx \]
\[ t_4 = \text{solve} (M, \text{rhs}) \]

\[ t_a = t + 1.0/6.0 \cdot t_1 + 1.0/3.0 \cdot t_2 + 1.0/3.0 \cdot t_3 + 1.0/6.0 \cdot t_4 \]

\[ d = -\text{grad}(q)[i] \cdot z[i,j] \cdot \text{grad}(p)[j] \cdot dx \]
\[ A = M - 0.5 \cdot d \]
\[ \text{rhs} = \text{action} (M + 0.5 \cdot d, t_a) \]
\[ t["n+1"] = \text{solve} (A, \text{rhs}) \]
Interface to state

# Tracer concentration – scalar
T = state.scalar_fields["Tracer"]

# Velocity – vector
V = state.vector_fields["Velocity"]

# Diffusivity – tensor
Mu = state.tensor_fields["Diffusivity"]

# Tracer concentration – scalar
Conc = Function(T)

# Assemble using field at timestep n
# Solve problem for timestep n+1
RHS = v*Conc["n"]*dx
Conc["n+1"] = solve(LHSMatrix, RHS)
Conclusions & Ongoing Work

- Multicore architectures provide *high performance* FE assembly & solution
- Rewriting code for every architecture is *low performance* software development

- Getting the best performance is hard:
  - Algorithmic choices (local/global assembly)
  - Implementation choices (e.g. fusions)

- Current work in progress:
  - CUDA, x86 multicore SSE & Aecute code generation
  - Integration with a faster SpMV implementation [6]

Spare Slides
$L(u) = q$

$R(u^\delta) = L(u^\delta) - q$

$\int_\Omega vR(u^\delta) dX = \int_\Omega vL(u^\delta) dX - \int_\Omega vq dX.$

$\int_\Omega vL(u^\delta) dX = \int_\Omega vq dX.$
FE Discretisation

\[ u^\delta = \sum_{i=0}^{N-1} \hat{u} \Phi_i \]

\[ \nu = \sum_{j=0}^{N-1} \hat{\nu} \Phi_j. \]

\[ \Phi_i(x_i) = 1, \text{ and } \forall k : i \neq k, \Phi_i(x_k) = 0 \]

\[ \Phi_i = \begin{cases} 
\frac{x-x_{k-1}}{x_k-x_{k-1}} & \text{if } x \in [x_{k-1}, x_k] \\
\frac{x_k-x_{k-1}}{x_k-x_{k+1}} & \text{if } x \in [x_k, x_{k+1}] \\
0 & \text{otherwise.} 
\end{cases} \]
### NVIDIA Tesla Architecture & CUDA

#### GT200 Architecture
- 10 TPCs
- 8 Banks of DRAM: 1-4GiB

#### CUDA Kernel:

```c
void daxpy(double a, double* x, double* y, int n)
{
    for (int i=0; i<n; i++)
        y[i] = y[i] + a*x[i];
}
```

```c
#define T_ID __global__ void daxpy(double a, double* x, double* y, int n)
{
    for (int i=T_ID; i<n; i+=T_COUNT)
        y[i] = y[i] + a*x[i];
}
```
Eliminating repeated (redundant) data
- 3 quadrature points; all equally weighted
- Reduce from 3-entry vector to scalar per element
- Loop fusions & interchanges, etc.

Constant folding:

```
for(int x=0; x<nodes; x++) {
    for(int y=0; y<nodes; y++) {
        ...
    }
}
```

Texture memory for matrix sparsity (random-access)
- Transposition
  - Achieve coalescing
  - Vital to performance
Proportion of GPU Time in each Kernel

Orange: Using Atomic operations
Blue: Using non-atomic operations

- matrix_addto
- matrix_addto_sum_diag
- vector_addto
- matrix_addto_diffusion
- dshape_dot_vector_shape
- shape_shape
- transform_to_physical
- advection_rhs
- scatter_rhs_values
- dshape_tensor_dshape
- diffusion_rhs
- daxpy
- memcpyD_toD_aligned
- gather_rhs_values
Memory Bandwidth Utilisation

Orange: Using Atomic operations
Blue: Using non-atomic operations
Testing

Backend (example):

```cpp
stringList *params = new stringList();
(*params).push_back(string("val"));
(*params).push_back(string("size_val"));
(*params).push_back(string("ele_psi"));
(*params).push_back(string("lmat"));
(*params).push_back(string("n"));
launchList.push_back(
    kernelLaunch("matrix_addto",params));
```

Frontend:

```cpp
psi = state.scalar_fields("psi")
v = TestFunction(P)
u = TrialFunction(P)
f = Function(P)
f.name="shape_rhs"
A = dot(grad(v),grad(u))*dx
solve(P, A, f)
```

Result: Code generated is equivalent to `test_laplacian`

- manual translation

- Helmholtz equation:

\[ \nabla^2 u - \lambda u = f \]

- Weak form:

\[ \int_{\Omega} \nabla v \cdot \nabla u dX + \lambda \int_{\Omega} vu dX = - \int_{\Omega} vf \]

- \[ A = (\text{dot}(\text{grad}(v), \text{grad}(u)) + (20) \times \text{dot}(v,u)) \times dx \]
Testing - continued

- Implement a Helmholtz solver in FEniCS Dolfin [7], and compare output with generated code
- Frontend didn’t parse modified code.
- Alternative: add extra calls to `shape_shape` and `matrix_addto` to backend test code

FEniCS Dolfin solution:  
Generated code solution:
Variable naming

How do we ensure the output of a kernel is correctly input to successive kernels? Consistently invent names.

```python
Psi = state.scalar_fields("psi")
v=TestFunction(Psi)
u=TrialFunction(Psi)
f=Function(Psi, "sin(x[0])+cos(x[1])")
A=dot(grad(v),grad(u))*dx
RHS=v*f*dx
Solve(Psi,A,RHS)
```
Code Generation

- List of variables, kernels and parameters passed to *backend*.
- Using the ROSE Compiler Infrastructure [6].

**Initialisation**
- `cudaMalloc()`
- `cudaBindTexture()`
- `cudaMemcpy()`

**Assembly**
- `kernel<<<.>>>()`

**Finalisation**
- `cudaFree()`
- `cudaUnbindTexture()`

**Declarations**
- `Int, double, ...`

- CUDA Keywords (``global``, `<<<...>>>` notation) inserted as arbitrary strings.

**Streaming**
- `cudaMemcpy()`
Memory Bandwidth Utilisation

Memory bandwidth (GB/s)

0 20 40 60 80 100 120 140

- matrix_addto
- matrix_addto_sum_diag
- vector_addto
- matrix_addto_diffusion
- dshape_dot_vector_shape
- shape_shape
- transform_to_physical
- advection_rhs
- scatter_rhs_values
- dshape_tensor_dshape
- diffusion_rhs
- daxpy
- memcpyDtoD_aligned
- gather_rhs_values
Assembly Throughput

![Graph showing assembly throughput for different systems and configurations.](image-url)
Helmholtz equation:
\[ \nabla^2 u - \lambda u = f \]

Weak form:
\[ \int_{\Omega} \nabla v \cdot \nabla u dX + \lambda \int_{\Omega} vu dX = -\int_{\Omega} vf \]

\[ A = \text{dot}(\text{grad}(v), \text{grad}(u)) + (20) \times \text{dot}(v,u) \times dx \]

Add extra calls to `shape_shape` and `matrix_addto`

FEniCS Dolfin solution:

Generated code solution:
We present a pilot study into using the *Unified Form Language* to generate CUDA code.
Intermediate Representation

\[
\int_{\Omega} \nabla v \cdot \nabla u dX + \lambda \int_{\Omega} v u dX = -\int_{\Omega} v f
\]
NVIDIA Tesla GT200 Architecture

- Cost: £300
- 1-4GiB RAM
- PCI-E interface (4GB/s)
- Made of many “stupid” processors

- Run the same code on many items of data

- For high performance:
  - Use many threads (10000+)
  - Coalescing:

  64B window
  16 threads (half-warp)
About Me:

- PhD Student in Department of Computing
  - On NERC Doctoral Training Account studentship

- Member of Software Performance Optimisation Group
  - Led by Paul Kelly

- Background: BSc & MSc Computing

- Supervisors:
  - Paul Kelly (Computing)
  - David Ham (AMCG, Grantham Institute)
Can we do any better? (5/6)

- The effect on the SpMV computation:

<table>
<thead>
<tr>
<th>Elements</th>
<th>$\mathbf{A}^T \mathbf{M}^e \mathbf{A} \mathbf{v}$</th>
<th>$\mathbf{M} \mathbf{v}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>28710</td>
<td>$1.91 \times 10^6$</td>
<td>$8.48 \times 10^5$</td>
</tr>
<tr>
<td>331714</td>
<td>$2.45 \times 10^7$</td>
<td>$9.85 \times 10^6$</td>
</tr>
</tbody>
</table>

- Times for the largest & smallest meshes over the whole simulation

- Much slower; more work to do! 2.5 times slower

- What is the overall performance?
Further work

On the UFL Compiler:
- Support for a more complete subset of UFL
- Development of a more expressive intermediate representation
  - Facilitates the development of other backends
- Generation of kernels from IR
- Automatic tuning

On the Conjugate Gradient Solver:
- Integration with better SpMV implementation [6]
  - Expect: further performance improvements

Preliminary results: Speedup

- Assembly phase for advection only, diffusion only, and advection-diffusion