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

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

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

Domain:
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
A Brief Overview of the Finite Element Method

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

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:

**Evaluate Integral**

*Gaussian Quadrature*

**Evaluate q at nodes**

**Do element = 1, N**

**Assemble(element)**

**End do**
A Brief Overview of the Finite Element Method

\[ L(u) = q \xrightarrow{\text{Evaluate Integral (Gaussian Quadrature)}} \int_{\Omega} vL(u^\delta) \, dX = \int_{\Omega} vq \, dX \]

Domain:

Compressed Sparse Row Matrix

Evaluate \( q \) at nodes

Do element = 1, N

Assemble(element)

End do

Vector
A Brief Overview of the Finite Element Method

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

Domain:

Evaluate Integral (Gaussian Quadrature)

Evaluate \( q \) at nodes

Compressed Sparse Row Matrix

Do element = 1,N

Assemble(element)

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:

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

Evaluate \( q \) at nodes

Evaluate Integral (Gaussian Quadrature)

\[ Ax = b \]

Solve for \( x \)

(CG, GMRES)

Compressed Sparse Row Matrix
The Test Problem

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

- **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

1 Result

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

Assembly Loop in CUDA:

1 Element

1 Result

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

GPU Memory

Thousands Of Elements
Porting `test_advection_diffusion`

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

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

Start

- Assemble & Solve Advection (x4)
- Assemble & Solve Diffusion
- Transfer initial conditions to GPU
- Assemble & solve advection (x4)

Fast execution
- Assemble & solve diffusion
- Transfer solution to host (optional)

Slow transfers
- Exit
- Exit
Coalescing with Unstructured Meshes

Obtaining coalescing:

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

- One thread per element
- Unstructured meshes

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
Coalescing with Unstructured Meshes

- 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**

![Diagram of matrix sparsity and memory access](image)

- **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
Preliminary results: Overall speedup

- Assembly and solve for advection only, diffusion only, and advection-diffusion for 200 timesteps.
Proportion of GPU Time in each Kernel

Q: Which kernels should we focus on optimising?

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:

\[ \mathbf{M} = \mathbf{A}^T \mathbf{M}^e \mathbf{A} \]

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

More efficiently, on the CPU:

“Addto” in Fluidity:

\[
\begin{align*}
\text{map}[1][i] &= \begin{bmatrix} 1 \\ 2 \end{bmatrix} \\
\text{map}[2][i] &= \begin{bmatrix} 2 \\ 3 \end{bmatrix}
\end{align*}
\]

\[
M = 0 \\
\text{foreach } Element e \text{ do} \\
\quad \text{for } i \leftarrow 1 \text{ to } N_e \text{ do} \\
\quad \quad \text{for } j \leftarrow 1 \text{ to } N_e \text{ do} \\
\quad \quad \quad M[\text{map}[e][i], \text{map}[e][j]] += M^e[i, j] ;
\]
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:

\[
y = \left( A^T \left( M^e (A \, v) \right) \right)
\]

Increase in work ~ 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:

  ![Graph showing simulation times]

  - Addto for Matrix and RHS
  - Stage1/2/3 for Matrix, SpMV for RHS

  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]

- A *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


