Optimizing miniFE an Implicit Finite Element Application on GPUs

Justin Luitjens - NVIDIA
Alan Williams, Mike Heroux - Sandia
The Mantevo Project

Analyzing and improving performance via mini-applications

Project Goals:
- Predict performance of real applications
- Foster communication between developers of applications, libraries and computer systems
- Guide design choices for both computer systems and application software

Mini-applications:
- Represent key performance characteristics of “real” applications
- Small, self-contained, easily ported to new systems
- Freely available as open-source
- Reduces to requirement on domain knowledge

• miniFE
• HPCCG
• miniGhost
• miniMD
• phdMesh
• miniXyce

Mike Heroux et al., Sandia National Laboratories
**CUDA Compute Architecture**

- **SM** – Streaming multi-processors
  - Each SM contains 32 processing cores
  - Execute in a Single Instruction Multiple Thread (SIMT) fashion
  - Up to 16 SMs on a card for a maximum of 512 compute cores
GPU Computing Model

- **Single-Instruction Multiple-Thread (SIMT)**
  - 32 threads execute in lock-step (warp)
- **Optimized for throughput not latency**
  - Hide latencies by having many warps on each SM and context switching when stalled
- **Extremely lightweight threads**
  - Creation and deletion overhead is small
  - Dedicated resources
  - Fast context switching
- **Fine-grained parallelism instead of coarse-grained parallelism**
  - Example: 1 thread per element
  - Coarse-grained parallelism is often used for MPI
“miniFE” is a Finite-Element mini-application

Implements algorithms from an implicit finite-element application

1. Assemble a sparse linear-system from the steady-state conduction equation on a domain of hexahedral elements

2. Solve the linear-system using the Conjugate Gradient algorithm
   - Per iteration:
     - 2 dot-products
     - 3 axpy operations
     - 1 sparse matrix-vector product

\[
\begin{align*}
    r_0 &= b - Ax_0 \\
    \text{Loop} \{ & \\
    & \text{If } k == 1 \\
    & \quad p_1 = r_0 \\
    & \text{else} \\
    & \quad \beta_k = r_{k-1}^T r_{k-1} / r_{k-2}^T r_{k-2} \\
    & \quad p_k = r_{k-1} + \beta_k p_{k-1} \\
    & \quad \alpha_k = r_{k-1}^T r_{k-1} / p_k^T A p_k \\
    & \quad x_k = x_{k-1} + \alpha_k p_k \\
    & \quad r_k = r_{k-1} - \alpha_k A p_k 
\end{align*}
\]
The Solve Phase

Many cuda enabled libraries for sparse-matrix operations already exist
- CUSPARSE, CUSP, CULA, MAGMA, etc.

Data format is key
- Host: Compressed Sparse Row (CSR)
- Device: ELL-Pack
  - [Link](http://www.nvidia.com/object/nvidia_research_pub_001.html)

This was not the focus of this work
- We used CUSP with the ELL format
Matrix Formats

\[
\begin{pmatrix}
  a_{0,0} & a_{0,1} & a_{0,2} & 0 \\
  a_{1,0} & a_{1,1} & 0 & 0 \\
  a_{2,0} & 0 & a_{2,2} & 0 \\
  0 & 0 & a_{3,2} & a_{3,3}
\end{pmatrix}
\]

CSR

row_offsets = [0 3 5 7 9]
column_indices = [0 1 2 0 1 0 2 2 3]
values = [a_{0,0} a_{0,1} a_{0,2} a_{1,0} a_{1,1} a_{2,0} a_{2,2} a_{3,2} a_{3,3}]

ELL

column_indices = [0 0 0 2 1 1 2 3 2 -1 -1 -1]
values = [a_{0,0} a_{1,0} a_{2,0} a_{3,2} a_{0,1} a_{1,1} a_{2,2} a_{3,3} a_{0,2} x x x]

ELL has much better memory access patterns for CUDA and vector operations.
CSR Memory Access Pattern

row_offsets = [0 3 5 7 9]
column_indices = [0 1 2 0 1 0 2 2 3]
values = [a_{0,0} a_{0,1} a_{0,2} a_{1,0} a_{1,1} a_{2,0} a_{2,2} a_{3,2} a_{3,3}]

for each row
for each column

Serial Access Pattern

[ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ]
CSR Memory Access Pattern

row_offsets = [0 3 5 7 9]
column_indices = [0 1 2 0 1 0 2 2 3]
values = [a_{0,0} a_{0,1} a_{0,2} a_{1,0} a_{1,1} a_{2,0} a_{2,2} a_{3,2} a_{3,3}]

for each row
for each column

Serial Access Pattern
CSR Memory Access Pattern

row_offsets = [ 0 3 5 7 9 ]
column_indices = [ 0 1 2 0 1 0 2 2 3 ]
values = [ a_{0,0} a_{0,1} a_{0,2} a_{1,0} a_{1,1} a_{2,0} a_{2,2} a_{3,2} a_{3,3} ]

for each row
for each column
CSR Memory Access Pattern

row_offsets = [ 0 3 5 7 9 ]
column_indices = [ 0 1 2 0 1 0 2 2 3 ]
values = [ $a_{0,0}$ $a_{0,1}$ $a_{0,2}$ $a_{1,0}$ $a_{1,1}$ $a_{2,0}$ $a_{2,2}$ $a_{3,2}$ $a_{3,3}$ ]

for each row

for each column

Serial Access Pattern
CSR Memory Access Pattern

row_offsets = [ 0 3 5 7 9 ]
column_indices = [ 0 1 2 0 1 0 2 2 3 ]
values = [ a_{0,0} a_{0,1} a_{0,2} a_{1,0} a_{1,1} a_{2,0} a_{2,2} a_{3,2} a_{3,3} ]

for each row
for each column
CSR Memory Access Pattern

row_offsets = [0 3 5 7 9]
column_indices = [0 1 2 0 1 0 2 2 3]
values = [a_{0,0} a_{0,1} a_{0,2} a_{1,0} a_{1,1} a_{2,0} a_{2,2} a_{3,2} a_{3,3}]

for each row
for each column

Serial Access Pattern
CSR Memory Access Pattern

row_offsets = [0 3 5 7 9]
column_indices = [0 1 2 0 1 0 2 2 3]
values = [a_{0,0} a_{0,1} a_{0,2} a_{1,0} a_{1,1} a_{2,0} a_{2,2} a_{3,2} a_{3,3}]

for each row
for each column

Serial Access Pattern
CSR Memory Access Pattern

row_offsets = [0 3 5 7 9]
column_indices = [0 1 2 0 1 0 2 2 3]
values = [\(a_{0,0}\), \(a_{0,1}\), \(a_{0,2}\), \(a_{1,0}\), \(a_{1,1}\), \(a_{2,0}\), \(a_{2,2}\), \(a_{3,2}\), \(a_{3,3}\)]

for each row
for each column

Serial Access Pattern
CSR Memory Access Pattern

- row_offsets = [0 3 5 7 9]
- column_indices = [0 1 2 0 1 0 2 2 3]
- values = [$a_{0,0} \ a_{0,1} \ a_{0,2} \ a_{1,0} \ a_{1,1} \ a_{2,0} \ a_{2,2} \ a_{3,2} \ a_{3,3}$]

for each row
for each column

Serial Access Pattern

Ideal for serial codes
CSR Memory Access Pattern

row_offsets = [0 3 5 7 9]
column_indices = [0 1 2 0 1 0 2 2 3]
values = [a_{0,0} a_{0,1} a_{0,2} a_{1,0} a_{1,1} a_{2,0} a_{2,2} a_{3,2} a_{3,3}]

parallel for each row
for each column

Parallel Access Pattern (lock step or vector)
CSR Memory Access Pattern

rowOffsets = [0 3 5 7 9]
columnIndices = [0 1 2 0 1 0 2 2 3]
values = [a0,0 a0,1 a0,2 a1,0 a1,1 a2,0 a2,2 a3,2 a3,3]

parallel for each row
for each column

Parallel Access Pattern (lock step or vector)
CSR Memory Access Pattern

rowOffsets = [0 3 5 7 9]
columnIndices = [0 1 2 0 1 0 2 2 3]
values = [a_{0,0} a_{0,1} a_{0,2} a_{1,0} a_{1,1} a_{2,0} a_{2,2} a_{3,2} a_{3,3}]

parallel for each row
for each column

Parallel Access Pattern (lock step or vector)

Scattered access pattern, inefficient cache line usage
ELL Memory Access Pattern

\[
\text{column_indices} = [0 \ 0 \ 0 \ 2 \ 1 \ 1 \ 2 \ 3 \ 2 \ -1 \ -1 \ -1 ]
\]
\[
\text{values} = [a_{0,0} \ a_{1,0} \ a_{2,0} \ a_{3,2} \ a_{0,1} \ a_{1,1} \ a_{2,2} \ a_{3,3} \ a_{0,2} \ x \ x \ x ]
\]

**parallel** for each row

for each column

Parallel Access Pattern (lock step or vector)
ELL Memory Access Pattern

\[
\text{column\_indices} = [0 \ 0 \ 0 \ 2 \ 1 \ 1 \ 2 \ 3 \ 2 \ -1 \ -1 \ -1]
\]
\[
\text{values} = [a_{0,0} \ a_{1,0} \ a_{2,0} \ a_{3,2} \ a_{0,1} \ a_{1,1} \ a_{2,2} \ a_{3,3} \ a_{0,2} \ x \ x \ x]
\]

parallel for each row
for each column

Parallel Access Pattern (lock step or vector)
**ELL Memory Access Pattern**

Parallel for each row

for each column

column_indices = [0 0 0 2 1 1 2 3 2 -1 -1 -1]
values = [a_{0,0} a_{1,0} a_{2,0} a_{3,2} a_{0,1} a_{1,1} a_{2,2} a_{3,3} a_{0,2} x x x]

**Parallel Access Pattern (lock step or vector)**

Fully coalesced access pattern (Some wasted bandwidth for padding)
Access pattern can also benefit vector processors and many core processors
Linear-System Assembly

1. Get node-ids & coordinates

2. Compute dense element-operators

3. Scatter-assemble into global sparse linear-system

Element-operator computation is perfectly parallel. (Elements are independent.)

Assembly into global sparse matrix has potential race conditions. (Multiple contributions to the same global row.)

Global Matrix
Parallel Matrix Assembly

```c
parallel for each element {
    get coords
    compute elem-operators
    get node ids
    sum into linear-system
}
```

- **1 thread per element**
  - Trivial Parallelism
  - Requires very small changes to the code
    - Mostly added `__device__` to existing functions
  - Sum into linear system using atomics
    - Used ELL format
    - Atomics could become a bottleneck

- **Use a single kernel for all operations**
  - Avoid having to reload data from global memory

- **Separate get coords and get nodes**
Performance Limiters

- Compute element-operator
  - Flop intensive
  - Should be compute bound
  - Profiler shows that it is bandwidth bound
- Sum into linear system
  - Few flops
  - Bandwidth bound
  - Uncoalesced read & write with atomicAdd()
    - Atomics were less than 1 % of the overall run time
- Bandwidth is our limitation
Increasing Bandwidth Utilization

- Make sure accesses are coalesced
  - Assemble is coalesced, scatter is not

- Increase occupancy
  - Occupancy = active warps / max warps
  - Higher occupancy implies more loads and stores in flight
  - Can hide more latency
  - Currently 33%
  - Occupancy Limiters
    - Block size (Number of Blocks)
    - Shared memory usage
    - Register usage
      - It is unlikely that we can decrease register usage enough to increase occupancy

Why is compute element-operator bandwidth bound?
- Register spilling
Register Spilling

Spilling occurs for 2 reasons

1. Auto-spills due to indirect addressing
   • A[i], where i is not known at compile time
   • Unroll loops and inlining can help avoid these
   • Sometimes they are unavoidable

2. More state than registers

Compute element-operator state per thread:

- Node Ids: 32 bytes
- Node Coordinates: 96 bytes
- Diffusion Matrix: 512 bytes
- Source Vector: 64 bytes
- Total: 704 bytes
  • 176 32 bit registers
+ other registers for indexing, addresses, etc.

Max on Fermi is 63 registers per thread

Conclusion: We are going to spill a lot
Register Spilling

- Registers are spilled to L1 -> L2 -> Global Memory
- We are running at 33% occupancy (512 threads)
- L1 size: 16K / 512 threads = 32 bytes/thread
- L2 size: 768K / 512 threads / 16 sms = 96 bytes/thread
- Spilling is likely going all the way to global memory
Reduce Impact of Register Spilling

- Exploit symmetry in the diffusion matrix (-224 bytes)
  - Also pushed into the CPU code
- Utilize shared memory
  - Store source vector in shared memory (-64 bytes)
  - We could place more in shared memory
    - Reduces cache size or occupancy
- Use large 48K cache configuration
  - Reduces shared memory to 16K
- Help the compiler make better optimization choices
  - restrict pointers
  - inline functions
  - Unroll loops (eliminates indexing registers)
    - Sometimes this can hurt performance
- These changes significantly increased performance
  - However, spilling is still significant
    - 512 bytes total
Performance Results

GPU vs. CPU Speedup

- GenStructure
- Assembly
- Solve
- Overall

GPU: M2090
CPU: Xeon E5-2680@2.7 Ghz (8 cores)
Summary

What did NVIDIA learn?
- FE assembly is math heavy but requires a lot of state
  - This leads to large amounts of register spilling
  - This flop-heavy operation is bandwidth bound due to spilling

What could we change to improve FE assembly?
- Increase the maximum registers per thread
- More shared memory per thread
- Better register spilling
  - Smarter compilers
  - Faster register spilling
  - Larger L1/L2 caches per thread
Questions?