Matrix-Free Finite Element Computations on Graphics Processors

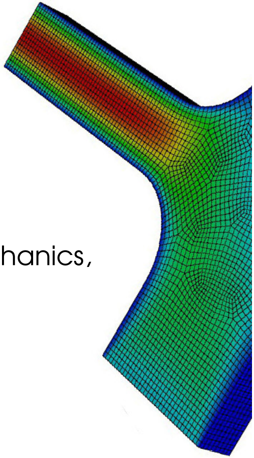
Karl Ljungkvist

BIT Circus
26 August 2015, Umeå

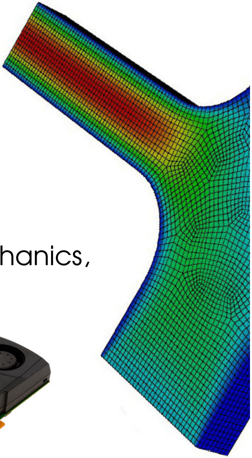
Motivation

Finite Element Methods:

- Unstructured mesh
- Flexibility wrt. geometry
- Adaptive mesh refinement
- Applications: fluid dynamics, structural mechanics, electromagnetics, etc.



Motivation



Finite Element Methods:

- Unstructured mesh
- Flexibility wrt. geometry
- Adaptive mesh refinement
- Applications: fluid dynamics, structural mechanics, electromagnetics, etc.

Graphics Processors:

- Lots of compute power & bandwidth
- Efficient (Gflop/Watt)
- Cheap (Gflop/$)

Finite Element Method

Linear system:

$$Au = b$$

where

$$A_{ij} = \sum_{K \in \mathcal{K}} \int_K \nabla \varphi_i \cdot \nabla \varphi_j \mathrm{d}x$$

$$b_i = \sum_{K \in \mathcal{K}} \int_K \varphi_i f(x) \mathrm{d}x$$

Finite Element Method

Linear system:

$$Au = b$$

where

$$A_{ij} = \sum_{K \in \mathcal{K}} \int_K \nabla \varphi_i \cdot \nabla \varphi_j \mathrm{d}x$$

$$b_i = \sum_{K \in \mathcal{K}} \int_K \varphi_i f(x) \mathrm{d}x$$

Two phases:

- Assembly
- Solving system

Finite Element Method

Solving the system:

- Large and sparse
- Iterative Krylov method
- Sparse Matrix-Vector Product (SpMV)

Finite Element Method

Solving the system:

- Large and sparse
- Iterative Krylov method
- Sparse Matrix-Vector Product (SpMV)

Problems:

- SpMV inefficient on modern processors (e.g. GPUs):
  - Bandwidth intensive – low $\frac{\text{Flops}}{\text{Byte}}$
  - GPUs need $\sim 5\frac{\text{Flops}}{\text{Byte}}$ to be fully utilized
  - SpMV $\sim 0.2\frac{\text{Flops}}{\text{Byte}}$
- Assembly takes time (often $>$30%)
- Non-linear problems need reassembly

Finite Element Method

Solving the system:

- Large and sparse
- Iterative Krylov method
- Sparse Matrix-Vector Product (SpMV)

Problems:

- SpMV inefficient on modern processors (e.g. GPUs):
  - Bandwidth intensive – low $\frac{\text{Flops}}{\text{Byte}}$
  - GPUs need $\sim 5\frac{\text{Flops}}{\text{Byte}}$ to be fully utilized
  - SpMV $\sim 0.2\frac{\text{Flops}}{\text{Byte}}$
- Assembly takes time (often $>$30%)
- Non-linear problems need reassembly

Idea: Matrix-free approach

- Merge assembly and solution phases

Matrix-free approach

Exploit structure of A:

$$A = \sum_{k \in \mathcal{K}} P_k a_k P_k^T$$

$$(a_k)_{i_{loc}, j_{loc}} = \int_{\Omega_k} \nabla \varphi_{i_{loc}} \cdot \nabla \varphi_{j_{loc}}$$

- P_k : local-to-global mapping

Matrix-free approach

Matrix-free application:

- $Au = \left(\sum_k P_k a_k P_k^T\right) u = \sum_k \left(P_k a_k P_k^T u\right)$
- Many small and dense MxVs

Matrix-free approach

Matrix-free application:

- $Au = \left(\sum_k P_k a_k P_k^T\right) u = \sum_k \left(P_k a_k P_k^T u\right)$
- Many small and dense MxVs

Parallel algorithm

- For each element / thread
  - Fetch local DoFs

$$u_{loc} := P_k^T u$$

  - Perform local product

$$v_{loc} := a_k u_{loc}$$

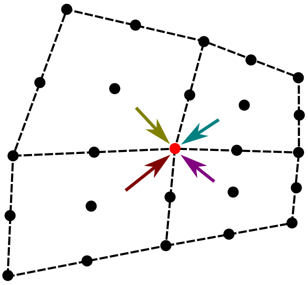
  - Add to global vector

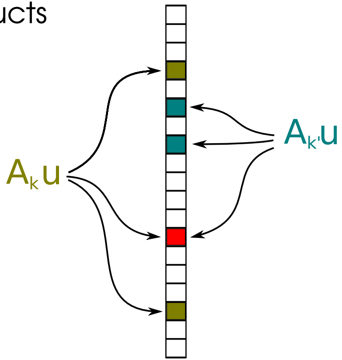
$$v := v + P_k v_{loc}$$

Conflicting updates

Issue:

- A node is updated by sub-products of all neighboring elements
- Race condition





Conflict handling

Solution: Atomic updates

- Built-in CUDA function `atomicAdd`
- Thread-safe update of memory
- Overhead – conflicts must be rare

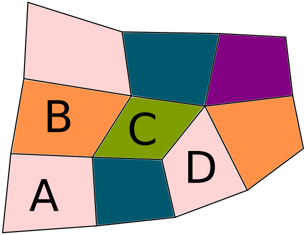
Conflict handling

Solution: Atomic updates

- Built-in CUDA function `atomicAdd`
- Thread-safe update of memory
- Overhead – conflicts must be rare

Alternative Solution: Graph coloring

- Color elements such that no pair of elements of the same color share a vertex.
- Elements within a color processed in parallel



Local product

Operation:

$$u_{loc} = a_k v_{loc}$$

Problem:

- For general meshes,
  - a_k not constant
  - Too much memory to store individually
- For Cartesian meshes
  - a_k constant
  - Might still not fit in GPU cache

Local product

Operation:

$$u_{loc} = a_k v_{loc}$$

Problem:

- For general meshes,
  - a_k not constant
  - Too much memory to store individually
- For Cartesian meshes
  - a_k constant
  - Might still not fit in GPU cache

Idea:

- Same approach – matrix-free local product

Structure of Local Matrix

Local (mass) matrix:

$$a_{ij}^k = \int_{\Omega_k} \varphi_i^k \varphi_j^k \mathbf{dx}$$

Structure of Local Matrix

Local (mass) matrix:

$$a_{ij}^k = \int_{\Omega_k} \varphi_i^k \varphi_j^k \mathbf{dx}$$

Transformation to reference element:

$$a_{ij}^k = \int_{\hat{\square}} \hat{\varphi}_i \, \hat{\varphi}_j \, |J_k| \mathrm{d}\boldsymbol{\xi}$$

Structure of Local Matrix

Local (mass) matrix:

$$a_{ij}^k = \int_{\Omega_k} \varphi_i^k \varphi_j^k \, \mathrm{d}\mathbf{x}$$

Transformation to reference element:

$$a_{ij}^k = \int_{\hat{\square}} \hat{\varphi}_i \, \hat{\varphi}_j \, |J_k| \mathrm{d}\boldsymbol{\xi}$$

Numerical quadrature:

$$a_{ij}^k = \sum_{q=1}^{N_q} \hat{\varphi}_i(\boldsymbol{\xi}_q) \, \hat{\varphi}_j(\boldsymbol{\xi}_q) |J_k(\boldsymbol{\xi}_q)| w_q$$

Local Product

Matrix-free approach II:

$$u_i = \sum_{q=1}^{n_q} \hat{\varphi}_i(\boldsymbol{\xi}_q) \left[\sum_{j=1}^{n_p} \hat{\varphi}_j(\boldsymbol{\xi}_q) v_j \right] |J_k(\boldsymbol{\xi}_q)| w_q$$

Local Product

Matrix-free approach II:

$$u_i = \sum_{q=1}^{n_q} \hat{\varphi}_i(\boldsymbol{\xi}_q) \left[\sum_{j=1}^{n_p} \hat{\varphi}_j(\boldsymbol{\xi}_q) v_j \right] |J_k(\boldsymbol{\xi}_q)| w_q$$

Local product:

$$v_q = \sum_{j=1}^{n_p} \hat{\varphi}_j(\boldsymbol{\xi}_q) v_j$$

$$u_i = \sum_{q=1}^{n_q} \hat{\varphi}_i(\boldsymbol{\xi}_q) v_q |J_k(\boldsymbol{\xi}_q)| w_q$$

Tensor structure

Tensor product elements:

$$\varphi_i(\boldsymbol{\xi}) = \psi_\mu(\xi_1)\psi_\nu(\xi_2)\psi_\sigma(\xi_3)\,, \quad i \sim (\mu, \nu, \sigma)$$

Tensor structure

Tensor product elements:

$$\varphi_i(\boldsymbol{\xi}) = \psi_\mu(\xi_1)\psi_\nu(\xi_2)\psi_\sigma(\xi_3)\,, \quad i \sim (\mu, \nu, \sigma)$$

Quadrature points:

$$\boldsymbol{\xi}_q = (\xi^\alpha, \xi^\beta, \xi^\gamma)\,, \quad q \sim (\alpha, \beta, \gamma) \qquad \text{also: } \psi_\mu^\alpha := \psi_\mu(\xi^\alpha)$$

Tensor structure

Tensor product elements:

$$\varphi_i(\boldsymbol{\xi}) = \psi_\mu(\xi_1)\psi_\nu(\xi_2)\psi_\sigma(\xi_3)\,, \quad i \sim (\mu, \nu, \sigma)$$

Quadrature points:

$$\boldsymbol{\xi}_q = (\xi^\alpha, \xi^\beta, \xi^\gamma)\,, \quad q \sim (\alpha, \beta, \gamma) \qquad \text{also: } \psi_\mu^\alpha := \psi_\mu(\xi^\alpha)$$

Algorithm:

$$v^{\alpha\beta\gamma} = \sum_\mu \psi_\mu^\alpha \sum_\nu \psi_\nu^\beta \sum_\sigma \psi_\sigma^\gamma v_{\mu\nu\sigma}$$

$$u_{\mu'\nu'\sigma'} = \sum_\alpha \psi_{\mu'}^\alpha \sum_\beta \psi_{\nu'}^\beta \sum_\gamma \psi_{\sigma'}^\gamma v^{\alpha\beta\gamma} w^{\alpha\beta\gamma} |J_k^{\alpha\beta\gamma}|$$

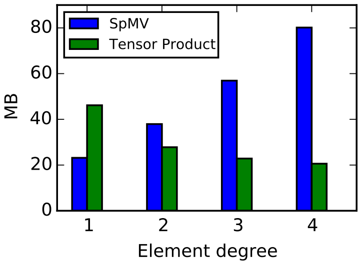
Note:

- Tensor contractions $\sim$ Dense MxM
- Only $J_k^{\alpha\beta\gamma}$ depends on k

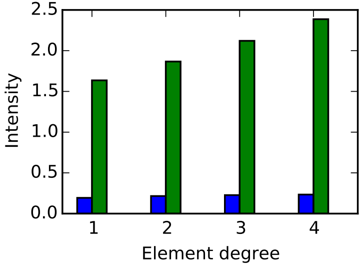
Tensor structure

2D

Data usage



Computational intensity

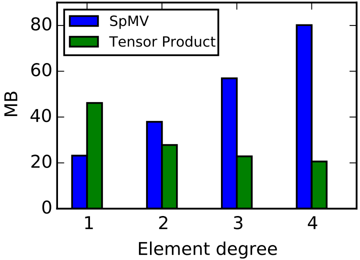


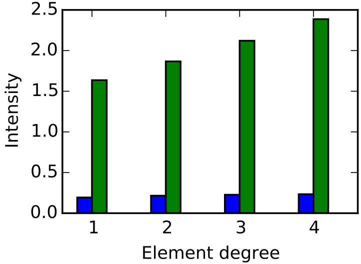
Tensor structure

Data usage

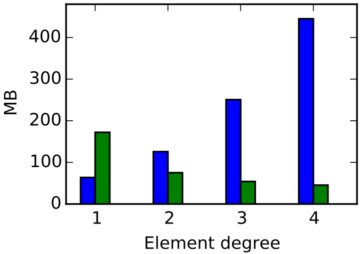
Computational intensity

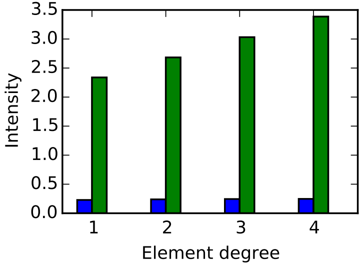
2D





3D





Experiment

Problem:

- Poisson equation

$$-\nabla(A(x)\nabla u) = f$$

- 2D and 3D
- Elements of order $1, 2$ and 4
- Conjugate Gradient with Chebyshev preconditioner

Experiment

Problem:

- Poisson equation

$$-\nabla(A(x)\nabla u) = f$$

- 2D and 3D
- Elements of order 1, 2 and 4
- Conjugate Gradient with Chebyshev preconditioner

Implementation:

- Preliminary code using `Deal.II` + CUDA
- Structure-of-array data layout
- Compared to highly optimized multicore/vectorization version

Experiment

Problem:

- Poisson equation

$$-\nabla(A(x)\nabla u) = f$$

- 2D and 3D
- Elements of order 1, 2 and 4
- Conjugate Gradient with Chebyshev preconditioner

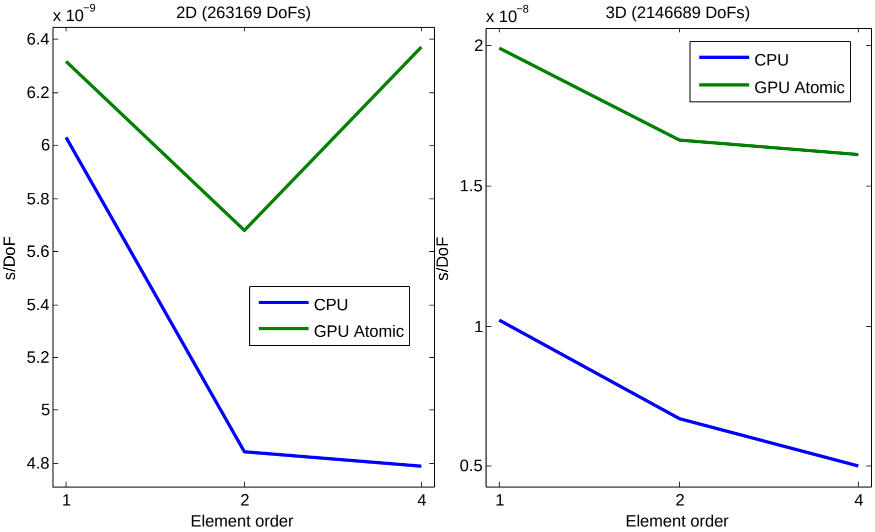
Implementation:

- Preliminary code using `Deal.II` + CUDA
- Structure-of-array data layout
- Compared to highly optimized multicore/vectorization version

System:

- 2 x Intel Xeon E5-2680 (8 cores, 172 GFlops)
- 64 GB DRAM
- Nvidia Tesla K20c GPU (2496 cores, 5 GB, 1173 GFlops, 208 GB/s)
- CUDA 7.5

Results


x 10^−9
2D (263169 DoFs)
6.4
6.2
6
5.8
5.6
5.4
5.2
5
4.8
s/DoF
CPU
GPU Atomic
1
2
4
Element order
x 10^−8
3D (2146689 DoFs)
2
1.5
1
0.5
s/DoF
CPU
GPU Atomic
1
2
4
Element order

Closing

Conclusion:

- Matrix-free method promising
- Help leveraging GPUs to make FEM computations faster / more efficient
- Still improvements to be done (register usage & memory access pattern)

Closing

Conclusion:

- Matrix-free method promising
- Help leveraging GPUs to make FEM computations faster / more efficient
- Still improvements to be done (register usage & memory access pattern)

Ongoing work:

- Graph Coloring
- Tiling & multiple threads / element
- Multigrid
- Solve more complicated problems
- Get stuff into `Deal.II`