Performant low-order matrix-free finite element kernels on GPU architectures

Randolph R. Settgast[1], William R. Tobin [2], Yohann Dudouit[3], Nicola Castelletto[1], Sergey Klevtsov[4], Benjamin C. Corbett[1]

[1]Atmospheric, Earth, and Energy Division, [2]Applications, Simulations, and Quality Division, [3]Center for Applied Scientific Computing, Lawrence Livermore National Laboratory, Livermore, CA, USA
[4] Energy Resources Engineering, Stanford University, Stanford, United States

Motivation

- Many engineering application typically contain highly heterogeneous material properties and in-situ state variables with high levels of uncertainty. For example, in subsurface reservoir modeling data are typically provided at a resolution representative of the heterogeneity/natural layering of the geologic formation as determined by a geophysical analysis.
- The evolution of high performance computing architectures towards GPU accelerator based systems has resulted in a focus on high order finite element methods which are more FLOP/s intensive than low order methods.
- In industrial applications containing high levels of uncertainty/heterogeneity, the use of higher-order discretizations does not result in a realizable accuracy advantages over low order discretizations. As such, low order methods dominate some industrial simulation community.
- Application of a low order FE method typically involves the filling and allocation of a global sparse matrix of size $(ndof \cdot nnzr)$, where $ndof$ is the global number of degrees of freedom in the problem, and $nnzr$ is the number of non-zeros in a row of the matrix. For iterative Krylov methods, these large sparse matrices are used as a vector operator through a sparse matrix vector multiply $\mathbf{Ax} = \mathbf{b}$. In contrast, a matrix-free approach eliminates the need to allocate/fill/store the sparse global stiffness matrix, which results in a significant reduction in the amount of memory required to perform a simulation.
- The speed of application of the matrix/matrix-free operator should be considered. In the case of a matrix-based operator, the speed of sparse matrix vector product has a low arithmetic intensity, i.e. is bound by the memory bandwidth (e.g. [**Zardoshti2017AdaptiveMultiplication**]). A matrix-free operator potentially allows a reduction in the bandwidth required to carry out the operator application, which can lead to faster runtime and better utilization of the GPU's capabilities in some cases.
- We present a strategy for implementing highly performant low-order matrix-free FE operator kernels for the case of a Laplace problem and Linear Elasticity. These kernels are applied within an un-preconditioned conjugate gradient (CG) iteration to provide context for the potential performance in an iterative Krylov method.

Model Problems

We focus on the Laplace's and linear elasticity equations. The local matrices associated to an element Ω^e for such equations are defined as:

$$[A^e]_{ij} = \int_{\Omega_e} \nabla\phi_i \cdot \nabla\phi_j \mathrm{d}\boldsymbol{x} \qquad \text{(Laplace)} \qquad (1)$$

$$[A^e]_{ij} = \int_{\Omega_e} \nabla\boldsymbol{\phi}_i : \boldsymbol{\sigma}(\nabla\boldsymbol{\phi}_j) \mathrm{d}\boldsymbol{x} \qquad \text{(linear elasticity)} \qquad (2)$$

where

- $\{\phi_i, \phi_j\}$ and $\{\boldsymbol{\phi}_i, \boldsymbol{\phi}_j\}$ are scalar and vector lowest-order nodal basis functions, respectively;
- $\boldsymbol{\sigma}$ is the second-order stress tensor that depends on the displacement gradient – specifically on the symmetric gradient, i.e. strain tensor – through an appropriate constitutive law;
- ∇ denotes the gradient operator.

Evaluating integrals using a transformation to a reference element and integrating numerically based on Gaussian quadrature, we obtain

$$[\mathbf{A}^e]_{ij} = \sum_q \left(\boldsymbol{J}_e^{-T}(\hat{\boldsymbol{x}}_q)\hat{\nabla}\hat{\phi}_i(\hat{\boldsymbol{x}}_q)\right) \cdot \left(\boldsymbol{J}_e^{-T}(\hat{\boldsymbol{x}}_q)\hat{\nabla}\hat{\phi}_j(\hat{\boldsymbol{x}}_q)\right) w_q \det(\boldsymbol{J}_e(\hat{\boldsymbol{x}}_q)) \qquad \text{(Laplace)} \qquad (3)$$

$$[\mathbf{A}^e]_{ij} = \sum_q \left(\hat{\nabla}\boldsymbol{\phi}_i(\hat{\boldsymbol{x}}_q)\boldsymbol{J}_e^{-1}(\hat{\boldsymbol{x}}_q)\right) : \left(\boldsymbol{\sigma}\left(\hat{\nabla}\boldsymbol{\phi}_j(\hat{\boldsymbol{x}}_q)\boldsymbol{J}_e^{-1}(\hat{\boldsymbol{x}}_q)\right)\right) w_q \det(\boldsymbol{J}_e(\hat{\boldsymbol{x}}_q)) \qquad \text{(linear elasticity)} \qquad (4)$$

where

- $\hat{\boldsymbol{x}}_q$ and w_q denote quadrature point coordinates in the reference element and weights, respectively;
- $\boldsymbol{J}_e$ is the Jacobian transformation matrix from the reference element to element Ω^e;
- $\hat{\nabla}$ denotes the reference space gradient.

Applying matrix $\mathbf{A}$ to a vector $\mathbf{u}$ to compute vector $\mathbf{v}$ in a matrix-free fashion requires merging the matrix assembly into the multiplication and consists of three steps for each element Ω^e:

1. Reading the local components $\mathbf{u}^e$ form the global input vector $\mathbf{u}$;
2. Evaluating the local matrix multiplication $\mathbf{v}^e = \mathbf{A}^e\mathbf{u}^e$;
3. Assembling the local components of $\mathbf{v}^e$ into the global output vector $\mathbf{v}$.

Using expressions (3)-(4) into the local multiplication yields:

$$[\mathbf{v}^e]_i = \sum_q \sum_j \left(\boldsymbol{J}_e^{-T}(\hat{\boldsymbol{x}}_q)\hat{\nabla}\hat{\phi}_i(\hat{\boldsymbol{x}}_q)\right) \cdot \left(\boldsymbol{J}_e^{-T}(\hat{\boldsymbol{x}}_q)\hat{\nabla}\hat{\phi}_j(\hat{\boldsymbol{x}}_q)\right) w_q \det(\boldsymbol{J}_e(\hat{\boldsymbol{x}}_q))[\mathbf{u}^e]_j \qquad \text{(Laplace)} \qquad (5)$$

$$[\mathbf{v}^e]_i = \sum_q \sum_j \left(\hat{\nabla}\boldsymbol{\phi}_i(\hat{\boldsymbol{x}}_q)\boldsymbol{J}_e^{-1}(\hat{\boldsymbol{x}}_q)\right) : \left(\boldsymbol{\sigma}\left(\hat{\nabla}\boldsymbol{\phi}_j(\hat{\boldsymbol{x}}_q)\boldsymbol{J}_e^{-1}(\hat{\boldsymbol{x}}_q)\right)\right) w_q \det(\boldsymbol{J}_e(\hat{\boldsymbol{x}}_q))[\mathbf{u}^e]_j \qquad \text{(linear elasticity)} \qquad (6)$$

In Equations (5)-(6) the order of operations in the summations is optimally rearranged for optimal performance as detailed in Algorithm 2.

Algorithms Matrix Free Kernel Implementation

Algorithm 1 Conjugate Gradient to solve $A\mathsf{x} = \mathsf{b}$

Require:
- b: right-hand side vector
- x_0: initial guess to solution vector

Ensure:
- x: solution vector

Initialize:
  $\mathsf{p}_0 = 0$
  $\mathsf{r}_0 = \mathsf{b} - A(\mathsf{x}_0)$
  $k = 1$
  $\tau_0 = \mathsf{r}_0 \cdot \mathsf{r}_0$
for (k < maxNumberOfIterations) **do**
  if $\mathsf{r}_k \leq absTol$ **then**
    break
  end if
  kernel: action of A on p_k
    $\mathsf{Ap}_k = A(\mathsf{p}_k)$
  $pAp_k = \mathsf{p}_k \cdot \mathsf{Ap}_k$
  $\alpha = \frac{\tau_k}{pAp_k}$
  kernel: update x_{k+1}, r_{k+1}, and $r_{k+1} \cdot r_{k+1}$:
    $\mathsf{x}_{k+1} = \mathsf{x}_k + \alpha_k \mathsf{p}_k$
    $\mathsf{r}_{k+1} = \mathsf{r}_k - \alpha_k \mathsf{Ap}_k$
    $\tau_{k+1} = \mathsf{r}_{k+1} \cdot r_{k+1}$
    $\beta = \frac{\tau_{k+1}}{\tau_k}$
  $\mathsf{p}_{k+1} = \mathsf{r}_{k+1} + \beta \mathsf{p}_k$
end for

Algorithm 2 Matrix-free action of elasticity operator $A(\cdot)$

Require:
- x: global support point coordinates
- u: global input vector
- D: operator specifying the physics

Ensure:
- v: global output vector

templated Class/ templated Method:
  $Class\langle FemType, MatType\rangle::kernelLaunch\langle LaunchPolicy\rangle$
for each element e **do**
  Gather/Load data from x and u into element local storage
    $\mathsf{x}^e = \{\boldsymbol{x}_a\}_{a=0}^{n_{supppts}} \leftarrow \mathsf{x}$,
    $\mathsf{u}^e = \{\boldsymbol{u}_a\}_{a=0}^{n_{supppts}} \leftarrow \mathsf{u}$
  for each integration point q **do**
    Calculate Jacobian transformation at integration point q:
      $\boldsymbol{J}_e = \boldsymbol{J}_e(\hat{\boldsymbol{x}}_q) = \sum_a \boldsymbol{x}_a \otimes \hat{\nabla}\hat{\phi}_a(\hat{\boldsymbol{x}}_q)$
    Calculate inverse Jacobian transformation $\boldsymbol{J}_e^{-1}$ and $\det(\boldsymbol{J}_e)$
    Calculate gradient of input variable
      $\nabla\boldsymbol{u}_e = \nabla\boldsymbol{u}_e(\hat{\boldsymbol{x}}_q) = \left(\sum_a \boldsymbol{u}_a \otimes \hat{\nabla}\hat{\phi}_a(\hat{\boldsymbol{x}}_q)\right)\boldsymbol{J}_e^{-1}$
    Calculate action of stiffness operator (i.e. stress):
      $\boldsymbol{\sigma}_e = D(\nabla\boldsymbol{u}_e)$
    Apply transform and quadrature weight to stress
      $\boldsymbol{P}_e = w_q \det(\boldsymbol{J}_e)\boldsymbol{\sigma}_e\boldsymbol{J}_e^{-T}$
    Apply the gradient of the test functions:
      $\boldsymbol{v}_a \mathrel{+}= \boldsymbol{P}_e \cdot \hat{\nabla}\hat{\phi}_a(\hat{\boldsymbol{x}}_q)$
  end for
  add element local output vector (v^e) to global output vector ($\mathbf{v}$)
    $\mathsf{v}^e \rightarrow \mathsf{v}$ (atomic add)
end for

Matrix-Free Kernel Implementation Details

Focused on maximizing compile time (c.t.) knowledge of kernel

1. Class templated on:
   - Finite element type (e.g.):
     - H1 Hexahedron (number of vertices known at c.t.)
     - Linear Lagrange Basis (parent derivatives...)
     - Gaussian quadrature (at quadrature points known at c.t)
   - Constitutive model type:
     - parameters and state variables known at c.t.
     - action of stiffness operator known at c.t.
2. RAJA kernel launch using $LaunchPolicy$
   - e.g. $cuda/hip/sycl\_exec < BLOCK\_SIZE >$
   - https://github.com/LLNL/RAJA
3. local array storage sizes known at c.t.
4. unroll loop (manually if required)
5. unroll loop over vertices, $\hat{\nabla}\hat{\phi}_a(\hat{\boldsymbol{x}}_q)$ known at c.t.
7. unroll loop over support points. $\hat{\nabla}\hat{\phi}_a(\hat{\boldsymbol{x}}_q)$ known at c.t.
8. operation for $D(\nabla\boldsymbol{u}_e)$ known at c.t.
10. unroll loop over support points. $\hat{\nabla}\hat{\phi}_a(\hat{\boldsymbol{x}}_q)$ known at c.t.
12. addition from local result to global involves hardware atomics through $cuda/hip/sycl\_atomic$ reduction policy.

Hardware specifications and metrics for comparison

The GPU architectures utilized in this study consist of:

- NVIDIA V100-SXM2 on LLNL/Lassen
- NVIDIA A100-SXM on NERSC/Perlmutter
- AMD MI250X on LLNL/Tioga

Table 1: Table of specifications for GPU hardware

	V100SXM2	A100SXM	MI250X
TFLOPs	7.8	9.7	24
Bandwidth (TB/s)	0.90	2.0	1.6
Number of SM | CU	84	128	110
L1 cache (kB/SM|CU)	128	192	16.0
L1 cache (MB)	10.8	24.6	1.76
L2 Cache size(MB)	6.1	40	8
FLOP/Byte	8.67	4.85	14.6

The performance metrics that are utilized in this study are:

- The throughput of $A(\mathsf{x})$.
  The NoOp-SpMV operator -vs- the real world matrix-free operator.
  - In the case of the NoOp-SpMV operator, the throughput are calculated by:
    * estimating the amount of memory in the sparse matrix and input/output vectors,
    * dividing the memory by the architecture bandwidth to attain a data transfer time,
    * and then dividing the number of degrees of freedom by that transfer time.

    These calculations are presented in Table 2.
  - In the case of the matrix-free operator, the number of degrees of freedom are divided by the minimum kernel duration as given by nsight-systems for all kernel launches.
- The conjugate gradient throughput is calculated by dividing the number of degrees of freedom in a problem by the minimum CG iteration time as given by a simple timer instrumented in the CG loop.

Table 2: Table of theoretical throughput for NoOp-SpMV

Quantity	Laplace	Mechanics
number of elements	1,000,000,000	
number of nodes	1,003,003,001	
memory / non-zero (bytes)	16	
num dof/support point	1	3
rows (num dof)	1,003,003,001	3,009,009,003
nnz/row	27	81
matrix memory (GB)	437	3,912
vector memory (GB)	16	49
total memory (GB)	453	3,959
V100-SXM2 transfer time (s)	0.50	4.4
A100-SXM transfer time (s)	0.23	2.0
MI250X transfer time (s)	0.28	2.4
V100-SXM2 throughput (GDof/s)	1.99	0.68
A100-SXM throughput (GDof/s)	4.43	1.52
MI250X throughput (GDof/s)	3.63	1.25

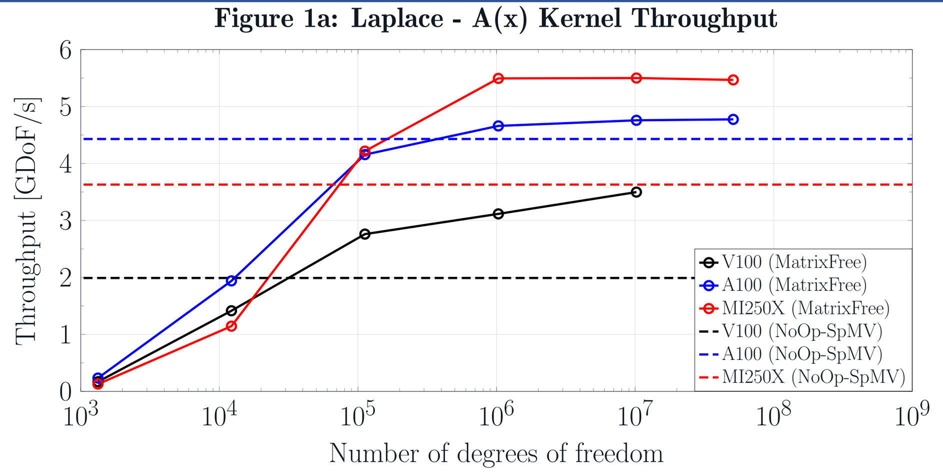
* The NoOp-SpMV throughput is significantly higher than is attainable in practice as it assumes data in the sparse matrix and vectors is transferred from HBM only once, does not include the cost of compute or atomic adds, etc.

Description of Study

- The goal of this study is to evaluate the potential for use of a MF operator in comparison to SpMV on low order finite element discretizations.
- A **single GPU** strong scaling study is conducted on a simple cube, increasing the number of elements by an order of magnitude at each interval in the scaling interval. It should be noted that the inclusion of MPI communications would influence the overall results, but will similarly affect both the MF operator and the SpMV operator as the MPI communication required to conduct operations will be conduced similarly as part of the CG iteration algorithm.
- While the problem mesh is Cartesian, the mesh is assumed to be unstructured and requires transfer of coordinate and mapping information. The result is that the
- The unpreconditioned C.G. shown in algorithm 1 is applied using the matrix free kernel operator described in algorithm 2 for a total of 1000 iterations in each run. Data is extracted from the runs to calculate the throughputs for the MF kernel and the overall CG iteration as previously described.
- The results of the matrix-free kernel operator are plotted against the NoOp-SpMV througput.

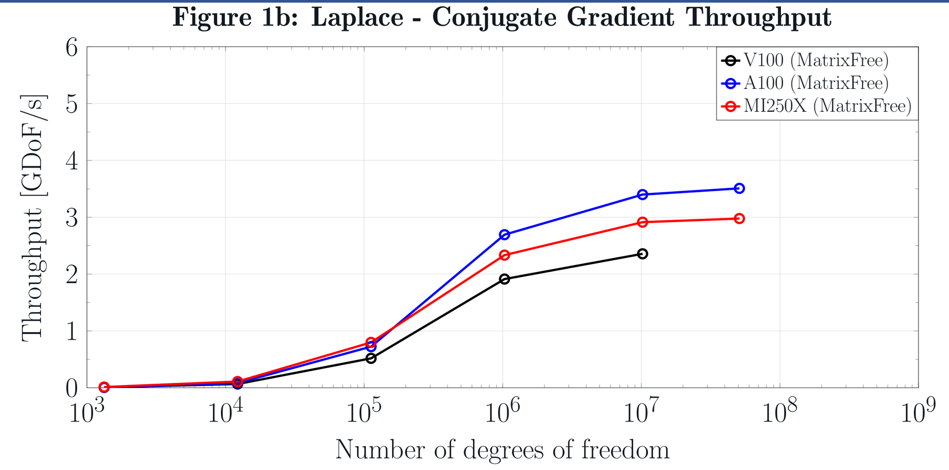
Results

Figure 1a: Laplace - A(x) Kernel Throughput



- Figure 1a shows that the matrix-free operator kernel achieves peak performance once the problem size exceeds 10^6 degrees of freedom.
- Once the matrix-free operator kernel is applied to a problem size where peak performance is achieved, the throughput moderately outperforms the theoretical maximum throughput of SpMV across all architectures. (1x-1.5x)
- There is approximately a 1.4x jump in throughput between the V100 and the A100, and approximately a 1.15x jump in throughput between the A100 and the MI250X. The lack of a large increase in performance for the MI250X over the A100 despite a 2.5x higher dflop capacity is the higher bandwidth of the A100.

Figure 1b: Laplace - Conjugate Gradient Throughput



- Figure 1b shows the throughput for the entire CG iteration reaches peak performance once the problem size reaches 10^7 degrees of freedom.
- The matrix free $A(\mathsf{x})$ operator kernel accounts for roughly 60% of the runtime for of the overall CG iteration.
- The reason for the shift in the peak performance between the kernel and the CG iteration is that the vector/reduction operations in the the CG algorithm achieve scaling efficiency at a higher problem size.
- In the CG throughput, the A100 outperforms the MI250X. This is a result of the bandwidth bound nature of the non-$A(\mathsf{x})$ kernels, as the A100 has a 15% higher bandwidth than the MI250X.

Figure 2a: Isotropic Linear Elasticity - A(x) Kernel Throughput

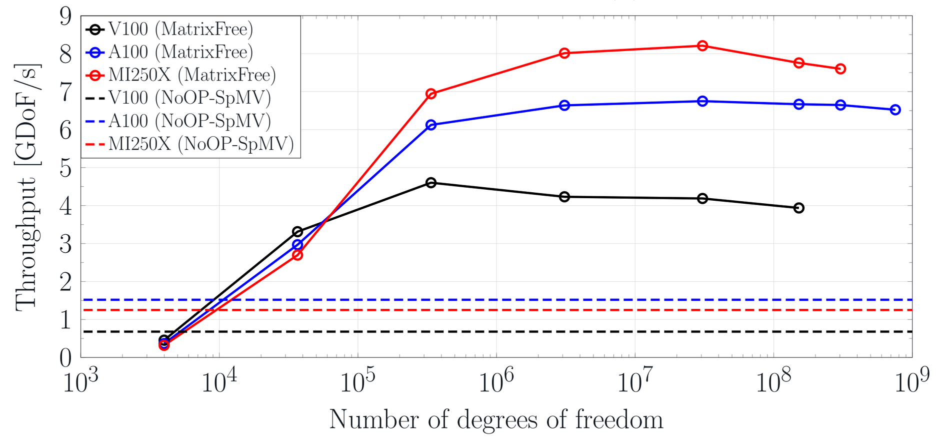
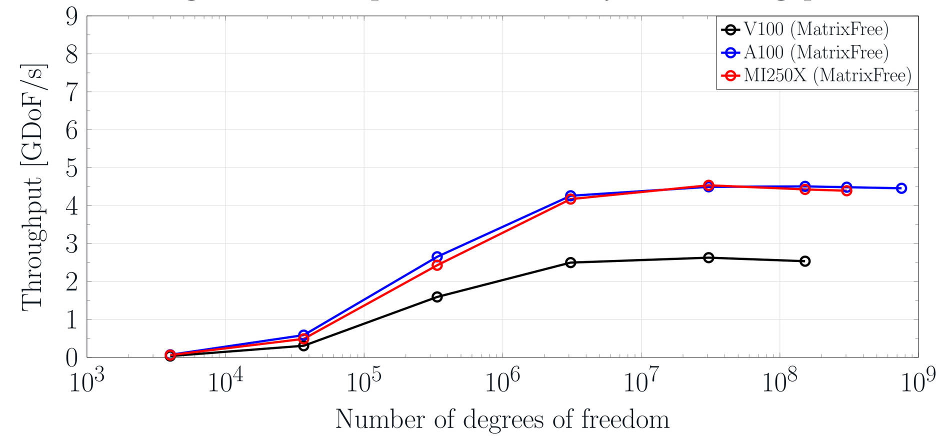


Figure 2b: Isotropic Linear Elasticity - CG Throughput



- Figure 2a shows that the matrix-free operator kernel achieves peak performance once the problem size exceeds 10^6 degrees of freedom.
- Once the matrix-free operator kernel is applied to a problem size where peak performance is achieved, the throughput significantly outperforms the theoretical maximum throughput of SpMV across all architectures. (4x-7x)
- Figure 2b shows the throughput for the entire CG iteration reaches peak performance once the problem size reaches 10^7 degrees of freedom.
- The matrix free $A(\mathsf{x})$ operator kernel accounts for roughly 65% of the runtime for of the overall CG iteration.
- There are similar trends between the architectures as displayed for the Laplace problem, although the heavier weight of the $A(\mathsf{x})$ results in equivalent CG throughput for the A100 and MI250X.

Conclusions & Ongoing Research

- The proposed low order matrix-free fem operator kernel performs well compared to the cost of transferring the data for a sparse matrix, and input/output vectors a single time from HBM. In the case of Laplace's problem the matrix-free operator performance is similar to the NoOp-SpMV, and in the case of linear elasticity the matrix-free operator significantly outperforms NoOp-SpMV.
- Given the performance of the matrix-free operator compared to the NoOp-SpMV and the reduction in memory allocation requirements duo to the lack of a global matrix, the use of a matrix-free appears to have potential as a viable alternative to SpMV.
- By utilizing LLNL's RAJA/CHAI/Umpire platform portability suite, the single kernel performs well on NVIDIA and AMD GPU's.
- Practical use of the proposed matrix-free approach requires use in a preconditioned Krylov method. The authors are currently engaged in development a multi-level strategy for preconditioning that does not require the system matrix to be stored.
- This study will be extended to include a greater range of applications such as compositional multi-phase flow coupled to poroelasticity.
- Study will be extended to include distributed memory parallelism (i.e. MPI), and direct comparison with state-of-the-art SpMV operators.

Acknowledgements and References

- This work was produced under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344.
- This research was supported by the Exascale Computing Project (ECP), Project Number: 17-SC-20-SC.