ACCELERATOR BASED PROGRAMMING
UPPSALA UNIVERSITY
FALL 2023

ASSIGNMETN 4: KOKKOS

This assignment is due October 10th at 11:59 pm. Upload your code and a report on Studium.

1. Benchmark description. When starting this assignment it is assumed that you have completed the two posted exercise/toturial sheets on studium.

For this assignment, we implement a typical computational kernel with Kokkos, namely the computation of the element matrices of a 3D finite element discretization of the Laplacian on a mesh of tetrahedral elements (simplices). Finite element methods are used to approximately solve partial differential equations, where the matrices computed in this exercise are later assembled into a sparse matrix, which will eventually be used to solve a linear system of equations to determine unknown solution coefficients of a piecewise linear approximation. On tetrahedral elements, the 4×4 element matrix on an element (e) can be computed by the following formula:

$$A_{ij}^{(e)} = \int_{\hat{E}} (\hat{\nabla}\varphi_i)^T (\mathbf{J}^{(e)})^{-1} (\mathbf{J}^{(e)})^{-T} \hat{\nabla}\varphi_j \det(\mathbf{J}^{(e)}) \mathrm{d}\mathbf{x} \tag{1.1}$$

where $\hat{E}$ denotes the reference tetrahedron with vertices in $(0,0,0)$,$(1,0,0)$, $(0,1,0)$, $(0,0,1)$, $\mathbf{J}^{(e)}$ is the 3×3 Jacobian matrix of the transformation from reference to real coordinates (with columns $\mathbf{v}_i - \mathbf{v}_0, i = 1,2,3$, for the four vertices of the tetrahedron in real space), and φ are the basis functions. In our program, we hardcode the result of the above multiplication and basis functions by some mathematical transformations, which eventually give the following code:

```
double C0 = J(1, 1) * J(2, 2) - J(1, 2) * J(2, 1);
double C1 = J(1, 2) * J(2, 0) - J(1, 0) * J(2, 2);
double C2 = J(1, 0) * J(2, 1) - J(1, 1) * J(2, 0);
double inv_J_det = J(0, 0) * C0 + J(0, 1) * C1 + J(0, 2) * C2;
double d = (1./6.) / inv_J_det;
double G0 = d * (J(0,0) * J(0,0) + J(1,0) * J(1,0) + J(2,0) * J(2,0));
double G1 = d * (J(0,0) * J(0,1) + J(1,0) * J(1,1) + J(2,0) * J(2,1));
double G2 = d * (J(0,0) * J(0,2) + J(1,0) * J(1,2) + J(2,0) * J(2,2));
double G3 = d * (J(0,1) * J(0,1) + J(1,1) * J(1,1) + J(2,1) * J(2,1));
double G4 = d * (J(0,1) * J(0,2) + J(1,1) * J(1,2) + J(2,1) * J(2,2));
double G5 = d * (J(0,2) * J(0,2) + J(1,2) * J(1,2) + J(2,2) * J(2,2));

A(0, 0) = G0;
A(0, 1) = A(1, 0) = G1;
A(0, 2) = A(2, 0) = G2;
A(0, 3) = A(3, 0) = -G0 - G1 - G2;
A(1, 1) = G3;
A(1, 2) = A(2, 1) = G4;
A(1, 3) = A(3, 1) = -G1 - G3 - G4;
A(2, 2) = G5;
A(2, 3) = A(3, 2) = -G2 - G4 - G5;
A(3, 3) = G0 + 2 * G1 + 2 * G2 + G3 + 2 * G4 + G5;
```

In this code, the 3×3 matrix `J` stores the inverse and transpose Jacobian matrix, $(\mathbf{J}^{(e)})^{-T}$, of the mathematical formula above.

2. Assumptions. In this assignment, we assume that the array of inverse Jacobian matrices has already been computed from the geometric information of the finite element vertices. You can assume the entries to be given, e.g. $[3,1,1;1,3,1;1,1,3]$ on **all** elements. It should be stored in a Kokkos array of type `Kokkos::View<double*[3][3]>`, where the compile-time dimension is equal to the number of elements. Furthermore, the result matrix A will be stored in a `Kokkos::View<double*[4][4]>` object, with the first array index associated to different elements. Note that you will need to modify the above code to take these three-dimensional data structures properly into account.

The tasks are as follows:

1. Write a program that implements the above algorithm in a portable manner, targeting both the CPU and the GPU, fulfilling the following requirements:
   (a) The program should explicitly transfer the data between the host and device, with the inverse Jacobians filled on the host and the final matrices shipped to the host. You can study the tutorial 04 of Kokkos regarding the necessary commands, https://github.com/kokkos/kokkos-tutorials/tree/main/Exercises/04/Solution.
   (b) You should provide implementations both with float and double numbers. It is recommended to use a template parameter Number for the benchmark function, and switch between the two instanti- ations of float and double in the main function.
   (c) Make sure to synchronize the CUDA device around the section with timers, using the code `Kokkos::fence()` before you measure the time.
   (d) The program should be capable of changing the memory layout type from `Kokkos::LayoutLeft` to `Kokkos::LayoutRight`.
2. Measure the achieved performance with float and double numbers over a range of elements from around 1000 to $4 \cdot 10^7$. The application performance should be expressed by the number of million elements computed per second, accompanied by secondary metrics regarding the memory bandwidth (GB/s) and floating point operations (GFlop/s). Report results from both the CPU and the GPU. To get more insight into the performance, you can also use the output from running the application with `nvprof ./executable -Nele 1000000`.

3. Predict the achievable performance from the available floating point performance and memory bandwidth in the data sheet https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/tesla-t4/t4-tensor-core-datasheet-951643.pdf, using the `float` type. What is the limiting resource, and what possibilities do you see to improve the performance? How about the CPU?

4. Compare the performance for the left and right layout types for both the CPU and the GPU. Discuss the results and give recommendations regarding the optimal data layout for this kind of algorithm on the CPU and the GPU.

5. Measure the time it takes to transfer the Jacobians from the host to the device, and the time it takes to transfer the resulting matrices from the device to the host. Compare this to the time it takes to perform the computations.

6. Starting from the observations in the previous task, discuss the achieved performance and give recommendations in case a full finite element pipeline is run, i.e.,
   - the Jacobians are computed from the vertices and connectivity information,
   - the element matrices are computed as above,
   - the element matrices are assembled into a sparse matrix, and
   - the linear system is solved with an iterative solver.

   What kinds of operations should be run on the GPU and which oper- ations should be run on the host? You can assume each step is done once, except for the iterative solvers that makes 50 passes through the matrix and some auxiliary data structures like vectors.

7. (Optional.) Have a look at the performance profile with `nsys` and assess the time spent in various sections.