Implementation of Memory Locality Optimizations in OpenFOAM-based Code

Ilya Popov[1] (ilya.popov@isteq.nl) Dmitry Astakhov[1]

13 July 2023, 18th OpenFOAM Workshop, Genoa, Italy

[1] ISTEQ BV, Eindhoven, the Netherlands, https://isteq.nl

Recap of memory locality techiques

At previous OpenFOAM Workshops we introduced a number of techniques aimed at improving memory locality for CFD code:

- Loop fusion (replacing multiple loops over data with one loop doing several operations at once) - reduces number of memory reads and writes.
- Microdomains: introducing another level of mesh decomposition with domain size corresponding to cache size. This way the data for one microdomain will stay in the cache.
- Switching loop order: making outer loop over microdomains, then loop over species, then loop over cells produced

These techniques together resulted in 2.9× performance increase for a benchmark problem.

However, the code was quite complex.

This time we attempt to make the code using these techniques easier to write and look more familiar to regular OpenFOAM user.

Goals

Memory locality techniques work well, but the code is complex and difficult to write every time

Therefore, we would like to:

- Make the code look similar to regular OpenFOAM code
- Make it easier to write
- Consolidate all the iteration order logic in one place
- Make changing iteration order simpler

Which will allow us to:

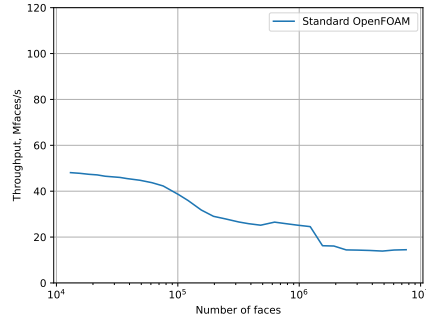
- Experiment with various iteration orders
- Reuse the same logic inside the iterative solver (for example, by computing true (instead of linearized) residual in the GMRES solver we can improve convergence)
- Introduce OpenMP-based multithreading
- Make use of overlapping of parallel exchanges and computation

Case study: viscous term computation

Viscous term flux is an example of a complex expression that is used in real life code:

$$F_{\rho U} = \mu \left[\left(\nabla \vec{U} + (\nabla \vec{U})^{\mathrm{T}} \right) - \frac{2}{3} (\nabla \cdot \vec{U}) \mathbf{I} \right] \cdot \vec{S}_f \qquad (1)$$

This is similar to what is inside `divDevRhoReff` function in standard OpenFOAM solvers.



In OpenFOAM, this expression is written as

```
F_rhoU = (fvc::interpolate(mu) * fvc::dev(fvc::twoSymm(fvc::interpolate(fvc::grad(U)))))
         & mesh.Sf();
```

where every `fvc::` operation and every operator (`*` and `&`) result in a temporary field that has to be written to memory and then read from memory.

Manual loop fusion

```
void Foam::compute_viscous_flux(surfaceVectorField &F_rhoU, const volTensorField &gradU, const volScalarField &nu)
{
    const fvMesh& mesh = gradU.mesh();
    const surfaceVectorField& Sf = mesh.Sf();
    const labelUList& owner = mesh.owner();
    const labelUList& neighbour = mesh.neighbour();
    const surfaceScalarField& weights = mesh.weights();

    for (label facei = 0; facei < mesh.nInternalFaces(); facei++) {
        label own = owner[facei];
        label nei = neighbour[facei];
        scalar w = weights[facei];

        scalar mu_f    = w*mu[own]    + (1.0-w)*mu[nei];
        tensor gradU_f = w*gradU[own] + (1.0-w)*gradU[nei];
        F_rhoU[facei] = (mu_f * dev(twoSymm(gradU_f))) & Sf[facei];
    }

    for (label patchi = 0; patchi < mesh.boundary().size(); patchi++) {
        const fvPatch& patch = mesh.boundary()[patchi];
        const fvPatchTensorField& p_gradU = gradU.boundaryField()[patchi];
        const fvsPatchVectorField& p_Sf = Sf.boundaryField()[patchi];
        const fvPatchScalarField& p_mu = mu.boundaryField()[patchi];
        fvsPatchVectorField& p_F_rhoU = F_rhoU.boundaryFieldRef()[patchi];

        if (!patch.coupled()) {
            label nFaces = patch.size();
            for (label facei = 0; facei < nFaces; facei++) {
                p_F_rhoU[facei] = (p_mu[facei] * dev(twoSymm(p_gradU[facei]))) & p_Sf[facei];
            }
        }
        else {
            auto p_mu_internal = p_mu.patchInternalField();
            auto p_mu_neighbour = p_mu.patchNeighbourField();
            auto p_gradU_internal = p_gradU.patchInternalField();
            auto p_gradU_neighbour = p_gradU.patchNeighbourField();
            const fvsPatchScalarField& p_w = weights.boundaryField()[patchi];

            label nFaces = patch.size();
            for (label facei = 0; facei < nFaces; facei++) {
                scalar w = p_w[facei];
                scalar mu_f    = w*p_mu_internal()[facei]    + (1.0-w)*p_mu_neighbour()[facei];
                tensor gradU_f = w*p_gradU_internal()[facei] + (1.0-w)*p_gradU_neighbour()[facei];
                p_F_rhoU[facei] = (mu_f * dev(twoSymm(gradU_f))) & p_Sf[facei];
            }
        }
    }
}
```

Pros:

- Performance
- Full control over iteration order

Cons:

- Significant amount (~ 50 lines) of code has to be written for every expression.
- Large part of code is just boiler plate, not specific to the expression. It would be nice if this iteration logic can be separated.
- Expression has to be repeated 3 times (highlighted; for inner faces, for regular boundaries and for coupled boundaries)
- Requires deeper knowledge of OpenFOAM internals
- The code will become even more complex when we try to use microdomains (and will need to be done in every such function)

Use lambda to abstract boilerplate?

Abstracting iteration boilerplate

Pass a lambda
and a list of fields here

Call the lambda here
with fileld elements
as arguments

Abstracting iteration boilerplate: for_each_face function

Solution:

- Pass a lambda as argument to the function
- Instead of computing hardcoded expression, let's call the lambda for every face
- Template magic is required to support arbitrary number of fields of arbitrary types
- Additional magic is required to automatically do interpolation of volFields onto faces

The code becomes:

```
volTensorField gradU(fvc::grad(U));
for_each_face_interp(
    [](const tensor& gradU_f, const scalar& mu_f, const vector& s_f, vector& F_rhoU_f) {
        F_rhoU_f = (mu_f * dev(twoSymm(gradU_f))) & s_f;
    },
    gradU, mu, mesh.Sf(), F_rhoU);
```

for_each_face function

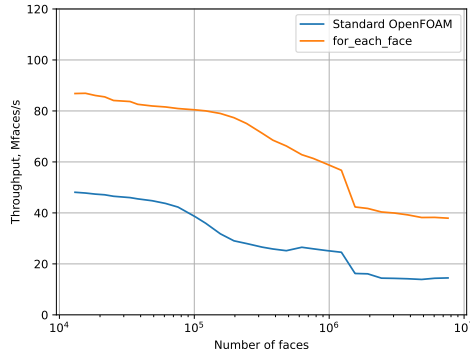
```
volTensorField gradU(fvc::grad(U));
for_each_face_interp(
    [](const tensor& gradU_f, const scalar& mu_f, const vector& s_f, vector& F_rhoU_f) {
        F_rhoU_f = (mu_f * dev(twoSymm(gradU_f))) & s_f;
    }, gradU, mu, mesh.Sf(), F_rhoU);
```

Pros:

- No temporary fields (except for gradient)
- Up to 3× performance gain compared to OpenFOAM (same performance as manual loop fusion)
- All iteration logic in one place - easier to change, less error-prone

Cons:

- Looks quite different to regular OpenFOAM expression
- Requires to use lambda syntax
- Duplication of field names and types
- Gradient still has to be computed separately



Abstracting iteration boilerplate: Expression templates

Standard OpenFOAM:

- **`fvc::`** Immediate computation: produces temporary field `Foam::tmp<Field>`
- **`fvm::`** Produces an `fvMatrix` for implicit solution

We introduce

- **`fve::`** Expression templates: produces an expression, actual computation happens in the assignment operator

Example:

```
scalar_field3 <<= fve::read(vector_field1) & fve::read(vector_field2);
```

Right hand side has type `dot_expr<field_expr<volVectorField>, field_expr<volVectorField>>`.

Operator `<<=` does all the work: it iterates over the cells or faces and calls `operator[]` on the right hand side.

Technical footnotes:

- `fve::read` is needed so the `operator&` is the `fve` version, not the regular OpenFOAM version that operates on fields.
- `operator<<=` is named such because in C++ you can not overload `operator=` for third party types. It also makes clear that this is not just assignment, but makes nontrivial work. Also look similar to math ← notation.

Expression templates

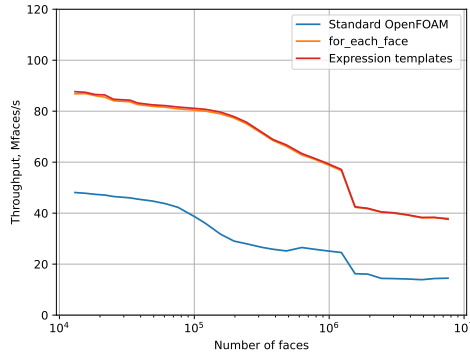
```
volTensorField gradU(fvc::grad(U));
F_rhoU <<= (interpolate(fve::read(mu)) * dev(twoSymm(interpolate(fve::read(gradU)))))
          & fve::read(mesh.Sf());
```

Pros:

- No temporary fields (except for gradient)
- Up to 3× performance gain. Same performance as `for_each_face` or manual looping, compiler manages to unwrap everything.
- Iteration logic is abstracted in one place
- Concise notation similar to standard OpenFOAM

Cons:

- Error messages can be very confusing
- Extending (adding support for more operators)
- Gradient has still to be computed separately



Expression templates: implementation

Create a tree of expressions, where each expression is a simple class containing:

```
struct expression {
    using value_type = ...;
    static constexpr loc location = ...;
    value_type operator[](Foam::label facei) const;
    value_type on_boundary(Foam::label patchi, Foam::label facei) const;
    const Foam::fvMesh& mesh() const;
    Foam::dimensionSet dimensions() const;
};
```

- Construction of an expression is very lightweight
- All the computation happens in the `operator[]`, which is called from the top level `operator<<=`.
- Each expression reports where it is defined (cells or faces), its value type (scalar, vector or tensor), and dimensions

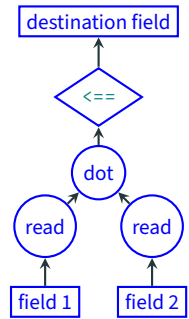
Easy level: element-wise functions: cell → cell, face → face

Expressions that are elementwise: either cell → cell or face → face:

- Unary functions
  - dev, twoSymm, ...
- Unary operator: `-`
- Binary operators
  - `+`, `-`, `*`, `/`
  - dot product (`&`), cross product (`^`), tensor product (`&&`)

All the work is implemented in the `operator[]` of the expression classes

```
auto operator[](label i) -> value_type {
  return expr1[i] op expr2[i];
}
```



Red — defined an faces, blue — defined in cells

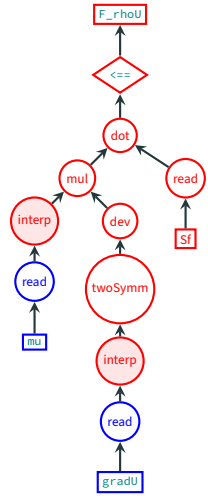
Medium level: two cells → face

These are expressions such as `surfaceInterpolate`, `snGrad`, etc.

The `operator[]` computes nested expression in the owner and neighbor cells and performs interpolation on the face.

```
value_type operator[](Foam::label facei) const {
  auto w = weight[facei];
  auto own = owner[facei];
  auto nei = neighbour[facei];
  return w*nested[own] + (1-w)*nested[nei];
}
```

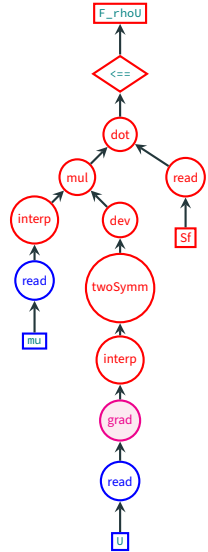
This duplicates computation of nested expression, therefore beneficial only for simple nested expressions.



Red — defined an faces, blue — defined in cells

Hard level: stencil-based operators

- Faces around the cell → cell: `surfaceIntegrate`, `div` of a surface Field
- Neighbour cells → cell: `grad`, `div` of a volume field
- Need to switch between iterating over cells and faces
- Hard to come up with a generic implementation for arbitrary stencil



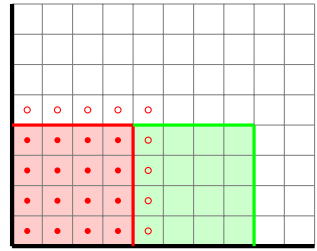
Red — defined an faces, blue — defined in cells

Microdomains

- Introduce another level of decomposition
- Size is such that all data needed for a microdomain can stay in cache
- Introduce an outer loop over microdomains

```
for each microdomain:
    for each internal and boundary face in the microdomain:
        update grad in owner and neighbour cells
    for each internal face in the microdomain:
        compute viscous flux on the face
```

The gradient can stay in the cache between the two stages.



Black lines — domain boundaries, red lines — boundary between block 0 and other blocks

Microdomains: cell ordering

OpenFOAM's renumberMesh utility already has an (underdocumented) option to make this kind of decomposition: it calls scotch internally:

```
renumberMesh -dict system/renumberMeshDict
```

With controlDict:

```
method CuthillMcKee;
blockSize 1000;
writeMaps true;
blockCoeffs {
    method scotch;
}
```

Produces the layout:

```
internal faces of block 0
...
internal faces of block N
faces between block 0 and blocks 1...N
faces between block 1 and blocks 2...N
...
faces between block N-1 and block N
```

Expression templates with gradient

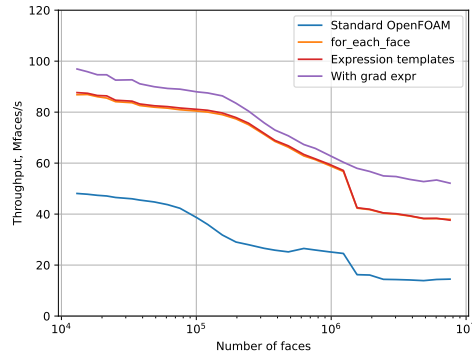
```
F_rhoU <<= (interpolate(fve::read(mu)) * dev(twoSymm(interpolate(grad(gradU, fve::read(U))))))
          & fve::read(mesh.Sf());
```

Pros:

- Better performance than expression templates with separate gradient, especially in RAM region (1.5×)

Cons:

- Only one level of cell–face iteration switch allowed at the moment



Code status

The demo code is available at `https://github.com/ISTEQ-BV/OpenFOAM_Workshop_2023_Demo`

The code is not complete, and is not suitable for real life use yet.

- Implements expression templates for a subset of operators/functions.
- Of stencil operations, only least squares grad is implemented at the moment.
- Code uses MP11 for metaprogramming, used standalone without requiring all of Boost, included as a git submodule.
- Uses Nanobench for benchmarking, included as a git submodule.
- Tested with OpenFOAM-v2212, should be compatible with wide range of OpenFOAM versions.
- Uses C++11, however later versions, such as C++17 or C++20 would allow for simpler implementation.

Next steps

- Complete the set of operations supported by expression templates
- Complete support for parallel execution; allows to use overlapping exchange consistently
- Experiment with more iteration order variations (inter-block boundary faces ordered together with internal cells)
- Can be adapted for use with multi-threading, such as OpenMP etc.

Thank you!

Test machine

- CPU: AMD EPYC 7543 (Milan) 3.8GHz
- L1: 32 kiB / core
- L2: 512 KiB / core
- L3: 32 MiB per CCX (4 cores), 256 per socket