CFD Lab Group E Project Documentation

Yunus Can Cakir
Technical University of Munich

Markus Mühlhäußer
Technical University of Munich

Sheng Chia-Yu
Technical University of Munich

July 2021

1 Introduction

In this CFD project, our main goal was to implement and investigate different turbulent models. Aside from this, we wanted to explore different solvers, parallelization of our code on the GPU with different APIs, preconditioning, adaptive grid refinement and more. Our CFD solver comprises of 3 types of simulations: a CPU, a Vulkan and a CUDA simulation. It also supports MPI on the CPU path. During the course of our project we have implemented K-epsilon, K-omega and K-omega SST turbulent models.

2 Code

Our code diverges from the given skeleton in the sense that we have a single Solver class and depending on the simulation type we have CPUSolver, VulkanSolver or CudaSolver classes. In the Utilities.hpp file, you can change the precision between double and float under the "Real" type. Inside the Simulation.cpp file, the solver loop runs. The solver loop is divided into 3 sub methods that is implemented by each subclass. These are namely,

```
void initialize()
```

This method handles all the initialization specific to chosen path. In Vulkan, this method handles preconditioner calculation, buffer allocation, descriptor binding etc. In CUDA, similarly this method handles preconditioner calculation and buffer allocation.

```
void solve_pre_pressure(Real &dt)
```

That handles the computations up to the pressure solving step. These include dt calculation, F and G calculation, setting up boundary values and other GPU or CPU specific tasks.

```
void solve_pressure(Real &res, uint32_t &it)
```

This method handles the pressure solving step. Depending on the configuration, either an SOR solver or a PCG solver is employed on the GPU or CPU.

```
void solve_post_pressure()
```

This method handles velocity calculation, turbulent viscosity, k/ϵ or ω calculation and copying the field values to the CPU for VTK output, if the GPU path is selected.

2.1 Running the code

The code has been tested on both Linux and Windows. For Linux you can build the code via CMake.

```
mkdir build
cd build
cmake ..
make
```

Make sure you enable CUDA or Vulkan support from the CMake settings.

- Vulkan : cmake .. -DUSE_VULKAN=ON
- CUDA : cmake .. -DUSE_CUDA=ON

Make sure you delete you CMake cache if you change these settings midway through the build. Also note that the Vulkan code is only tested and designed on a RTX 3060 GPU, therefore there may be problems on lower-end hardware.

2.2 Scene configuration

In addition to the usual options from the previous exercises, we have other options.

- model
  - 0 → Turbulence off(default)
  - 1 → K-epsilon model
  - 2 → K-omega model
  - 3 → K-omega SST model
- solver
  - 0 → SOR(default)
  - 1 → PCG
- simulation
  - 0 → CPU(default)
  - 1 → Cuda
  - 2 → Vulkan
- preconditioner
  - -1 → off(default)
  - 0 → AINV
  - 1 → SSOR
  - 2 → Jacobi preconditioner

- refine
  - Number (Grid has the refinement level 2^N

Note that following combinations are not supported:

- PCG - MPI
- Cuda - MPI
- Vulkan - MPI

3 Turbulent Models

In our code we are using RANS-based turbulence modelds. Specifically we are using eddy viscosity models that models the turbulence via a turbulent viscosity term.

3.1 K-epsilon Model

In this model, we model the turbulent viscosity as

$$\mu_t = \rho C_\mu \frac{k^2}{\epsilon} \tag{1}$$

Where k is the kinetic energy and ϵ is the turbulent dissipation rate.
We calculate these quantities via the equations:

$$\frac{\partial(\rho k)}{\partial t} + \frac{\partial\left(\rho k u_i\right)}{\partial x_i} = \frac{\partial}{\partial x_j}\left[\frac{\mu_t}{\sigma_k}\frac{\partial k}{\partial x_j}\right] + 2\mu_t E_{ij}E_{ij} - \rho\varepsilon \tag{2}$$

$$\frac{\partial(\rho \varepsilon)}{\partial t} + \frac{\partial\left(\rho \varepsilon u_i\right)}{\partial x_i} = \frac{\partial}{\partial x_j}\left[\frac{\mu_t}{\sigma_\varepsilon}\frac{\partial \varepsilon}{\partial x_j}\right] + C_{1\varepsilon}\frac{\varepsilon}{k}2\mu_t E_{ij}E_{ij} - C_{2\varepsilon}\rho\frac{\varepsilon^2}{k} \tag{3}$$

Although the K-epsilon model is fairly accurate away from the wall, it is problematic near the wall due to high shear stress, which results in higher μ along the walls.

3.2 K-omega model

Just like the K-epsilon model, the K-omega model is also a two equation model. Here, we reformulate the ω such that:

$$\omega = \frac{\epsilon}{C_\mu k}, C_\mu = 0.09 \tag{4}$$

Then we transport the quantities K and ω as usual.

Compared to K-epsilon model, the K-omega model has much better behaviour near walls, however one still wants the accuracy the K-epsilon model shows away from the walls. Which brings us to another widely used turbulence model.

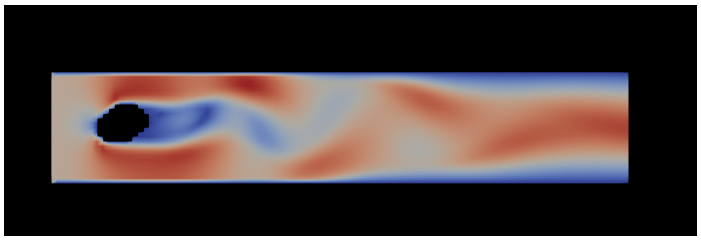

Figure 1: K-omega model on Karman Vortex Street

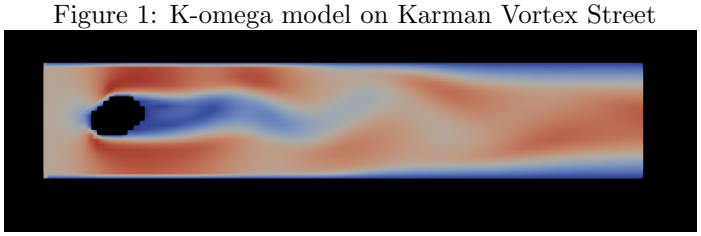

Figure 2: K-omega SST model on Karman Vortex Street

3.3 K-omega SST model

In this model, we "blend" between K-omega and K-epsilon via a constant and limit the turbulent viscosity in the process. In the process, the only difference with the K-omega model is the limitation of the viscosity and the extra term in the ω transport. Namely,

$$\nu_T = \frac{a_1 k}{\max\left(a_1 \omega, S F_2\right)} \tag{5}$$

$$\frac{\partial \omega}{\partial t} + U_j \frac{\partial \omega}{\partial x_j} = \alpha S^2 - \beta \omega^2 + \frac{\partial}{\partial x_j}\left[\left(\nu + \sigma_\omega \nu_T\right)\frac{\partial \omega}{\partial x_j}\right] + 2\left(1 - F_1\right)\sigma_{\omega 2}\frac{1}{\omega}\frac{\partial k}{\partial x_i}\frac{\partial \omega}{\partial x_i} \tag{6}$$

Via constants F_1 and F_2. These are empirically derived constants fitted to a tanh function.

In our observations, we found that SST model is in between the K-epsilon and K-omega model, as suggested by the literature.

4 Pressure Solvers and Preconditioners

Aside from the usual SOR solver, we have added a PCG solver for the CPU, Vulkan and the CUDA path. For reference in the CPU path, we've used a modified version of Robert Bridson's PCG code which also provides a Modified Incomplete Cholesky preconditioner. In the GPU paths, we've implemented our own PCG solver and experimented with different preconditioners. For the preconditioners in the GPU branches, we wanted SPAI preconditioners for fast evaluation on the GPU, otherwise we'd have to solve 2 triangular systems which is a costly operation.

4.1 Jacobi Preconditioner

This is the simplest type of preconditioner. Since in our case the Laplacian matrix has non-zero diagonals, we simply use $P_{ij} = \frac{1}{D_{ij}}$ where D is the diagonal part of the pressure matrix.

4.2 SSOR Preconditioner

This preconditioner [1] is defined by

$$M = KK^T \tag{7}$$

where

$$K = \frac{1}{\sqrt{2-\omega}}(\bar{D} + L)\bar{D}^{-1/2} \tag{8}$$

in which $0 < \omega < 2$ and $\bar{D} = \frac{1}{\omega}D$ In the code, we precompute this matrix from the pressure matrix, then in the CG algorithm, the only extra cost is a sparse matrix vector multiplication.

4.3 Approximate Inverse Preconditioner Variant

This preconditioner [2] approximates the inverse by first choosing an initial preconditioner, in which case we've chosen a Jacobi preconditioner, then approximating the inverse via Neumann series. Assuming we have D_0 as an initial approximation of A^{-1} and a condition

$$\|R_0\| \leq k < 1, \quad R_0 = I - AD_0 \tag{9}$$

Then,

$$\begin{aligned} A^{-1} &= \left(D_0 D_0^{-1}\right) A^{-1} = D_0 \left(D_0^{-1} A^{-1}\right) \\ &= D_0 \left(AD_0\right)^{-1} = D_0 \left(I - R_0\right)^{-1} \\ &= D_0 \left(I + R_0 + R_0^2 + R_0^3 + \ldots\right) \end{aligned} \tag{10}$$

where $R_n = I - AD_n$

This way, we can selectively approximate a finer preconditioner. We have stopped at D_1. As an extra measure, we also interpolate it via a naive Jacobi preconditioner.

Please note that since we are merely approximating a sparse approximate inverse, it isn't guaranteed to exist and more often than not, we need to choose different preconditioners depending on the scene. On some scenes, like the "Fluid Trap", none of the preconditioners work.

5 GPU Solvers

5.1 Vulkan

For our Vulkan solver, we wanted to focus on performance. To maximize the performance we fully solve the simulation on the GPU. Our code supports both double precision and single precision operations. However, for GPU execution the single precision path is recommended. To reduce the CPU overhead, we record all the command buffers beforehand, then we only submit those command buffers during the simulation. To avoid divergent code while handling boundary conditions, we instead encode all the BCs into a sparse matrix then do a sparse matrix multiplication.

For the sparse matrix representation, we use a CSR(Compressed Sparse Row) format. A pseudocode below describes a basic SpMV calculation.

During the PCG algorithm, we also do a sparse matrix vector multiplication. There, we use a diagonal storage format(DIA) which is better suited for compute intensive GPU code.

5.2 CUDA

CUDA calculations are similar to Vulkan, except some parts of the code, like the calculation of dt happens on the CPU.

Algorithm 1 An algorithm for sparse matrix vector calculation

```
if row < size then
    s ← 0
    for j from rowstart[row] to rowstart[row + 1] do
        s ← s + umatrix[j] ∗ u[colidx[j]]
```

6 Grid Refinement

Creation of detailed pgm files can be tedious, but especially when using turbulence models a finer grid can be beneficial. Therefore a refine parameter can be set to computationally create a finer grid in the powers of 2. To accurately compute the k-epsilon model near solid walls, a finer mesh size is needed. To save computation power a method for creating an adaptive grid was developed, but is yet unfinished and therefore on a seperate project branch.

References

[1] R. Helfenstein, J. Koko. 2012. Parallel preconditioned conjugate gradient algorithm on GPU.

[2] I. Labutin, I. Surodina. 1996. Algorithm for Sparse Approximate Inverse Preconditioners in the Conjugate Gradient Method.