VirtualFluids at DTU: Installation and Usage Guide

Author

Nils Horneff, PhD Candidate
DTU Wind and Energy Systems
Copenhagen, Denmark

March 20, 2025

VirtualFluids (VF) is an open-source Computational Fluid Dynamics (CFD) solver based on the Lattice Boltzmann Method (LBM). Developed at the Institute for Computational Modeling in Civil Engineering (iRMB) at TU Braunschweig, it is designed to efficiently simulate turbulent and multi-component flows. This guide provides an overview of VF's file structure, test cases, and installation process. Additionally, it details step-by-step instructions for running test cases locally on Linux (Ubuntu) and remotely on DTU's GBAR cluster. The end of the document provides a performance comparison on using VF on various DTU-related hardware.

1 VirtualFluids Architecture

1.1 Directories

```
/virtualfluids
 3rdparty/          # Third-party libraries (Metis, CUDA samples, MuParser)
 apps/              # Source code for example applications
    cpu/            # CPU-based applications
    gpu/            # GPU-based applications
 CMake/             # CMake configuration scripts
 Containers/        # Docker configurations and images
 docs/              # Documentation
 Python/            # Python applications
 pythonbindings/    # Python bindings for VirtualFluids
 src/               # Core source code
    basics/         # Basic utilities (geometry, VTK writer)
    core/cpu/       # CPU-based data structures and processing
    core/gpu/       # GPU-based data structures and processing
    lbm/            # Lattice Boltzmann Method implementation
    logger/         # Logging utilities
    parallel/       # MPI parallelization utilities
 tests/             # Automated tests to verify and validate code
    unit-tests/         # Unit testing framework (GoogleTest)
    regression-tests/   # Regression testing scripts
    performance-tests/  # Performance benchmarking tests
 .gitlab-ci.yml         # GitLab CI/CD configuration file
```

1.2 Test Cases

CPU	GPU
GyroidsRow	ActuatorLine
LaminarPipeFlow	AtmosphericBoundaryLayer
LaminarPlaneFlow	DrivenCavity
LidDrivenCavity	DrivenCavityMultiGPU
	LaminarPipeFlowGPU
	SphereInChannel
	SphereMultiGPU
	TaylorGreenVortex

Table 1: VirtualFluids test cases for CPU and GPU

2 GPUs at DTU

DTU provides both, regular and interactive GPU nodes. These nodes are grouped in the Load Sharing Facility (LSF) batch system that manages computational workloads across DTU's clusters. The interactive nodes are intended for development, profiling, and short test jobs. For more resource-intensive tasks, it is recommended to submit jobs to the appropriate GPU queues to avoid conflicts with other users. Detailed information can be found here. Below is a short summary of the GPU nodes available at DTU.

2.1 Regular NVIDIA-GPU Nodes at DTU

# GPUs	GPU Types	Alias
40	8 nodes (each 2x Tesla V100 32GB) + 6 nodes (each 2x Tesla V100 16 GB) + 3 nodes (each 4x Tesla V100 32GB with NVLink)	`gpuv100`
20	6 nodes (each 2x Tesla A100 PCIE 80GB) + 4 nodes (each 2x Tesla A100 PCIE 40GB)	`gpua100`
2	1 node (2x Tesla A10 PCIE 24GB)	`gpua10`
2	1 node (2x Tesla A40 48GB with NVLink)	`gpua40`

Table 2: Regular GPU nodes at DTU

2.2 Interactive NVIDIA-GPU Nodes at DTU

# GPUs	GPU Type	Alias
4	Tesla V100-SXM2 32GB	`sxm2sh`
2	Tesla A100 40GB	`a100sh`
2	Tesla V100 16GB	`voltash`

Table 3: Interactive GPU nodes at DTU

3 Running on local CPU (Laptop)

3.1 Installation

1. Update system packages and install required dependencies:

   ```
   sudo apt update
   sudo apt install -y \
   git cmake g++ openmpi-bin libopenmpi-dev paraview
   ```

   Note that for some reason, VF data cannot be opened on certain (older) ParaView versions (recommendation: install ParaView 5.13.2).

2. Clone the VirtualFluids repository:

   ```
   git clone https://git.rz.tu-bs.de/irmb/virtualfluids.git
   ```

3. Navigate into the cloned repository:

   ```
   cd /path/to/virtualfluids
   ```

   Replace `/path/to/virtualfluids` with the actual repository location.

4. Create a build directory, configure compilation for CPU simulation, and compile (using the `make -j$(nproc)` command compiles the code using all available CPU cores):

   ```
   rm -rf build # remove possible existing directory
   mkdir build && cd build
   cmake --preset=make_cpu -DVF_ENABLE_GPU=OFF ..
   make -j$(nproc)
   ```

   The available preset options are:

   - `cpu`
   - `gpu`
   - `python_bindings`
   - `make_all`: build all targets with Make
   - `make_cpu`: build only the CPU targets with Make
   - `make_gpu`: build only the GPU targets with Make

3.2 Running the Lid-Driven Cavity Simulation

1. Copy the configuration file of the test case into the `build/bin/` folder:

```
cd .. # get back to the repository main folder
cp apps/cpu/LidDrivenCavity/LidDrivenCavity.cfg build/bin
```

2. Move to the `build/bin/` folder and run the executable of the lid-driven cavity case (we need to provide the configuration file that specifies the simulation parameters):

```
cd build/bin
./LidDrivenCavityCPU LidDrivenCavity.cfg
```

3.3 Postprocessing with ParaView

1. Open ParaView (recommended version is v5.13.2, as older versions may yield errors):

```
paraview
```

2. Load the result file by clicking: *File* → *Open* and then choose: `/path/to/virtualfluids/build/output/LidDrivenCavity/mq/mq_collection.pvd`. The `.pvd` file is a metadata file referencing time-series files in `.vtu` or `.vtk` format.

3. Click *Apply* to load the data and click the play-button to step through the snapshots of each time-step. It should show something like this:

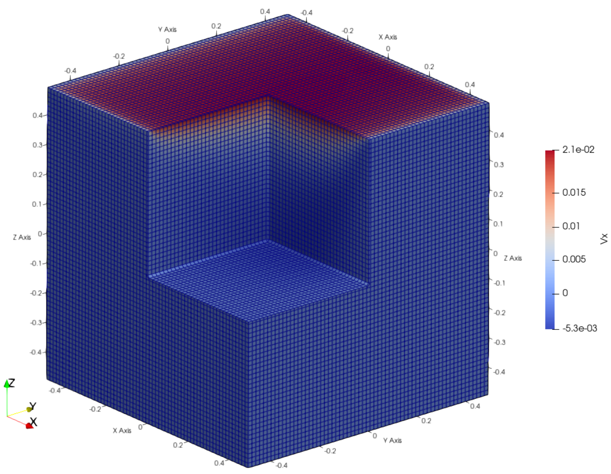

Figure 1: ParaView visualization of the 3D Lid-Driven Cavity test case using CPU-based VF.

3.4 Modify the Simulation Parameters

1. In case we want to adjust the simulation parameters, we need to open the configuration file with a text editor:

   ```
   nano LidDrivenCavity.cfg
   ```

2. Adjust the configuration file as needed. The default file looks like this:

```
# SPDX-License-Identifier: CC-BY-4.0
# SPDX-FileCopyrightText: Copyright @ VirtualFluids (...)
#set your output path here
path = ./output/LidDrivenCavity

L = 1.0
Re = 1000.0
velocity = 1.0
dt = 0.5e-3
nx = 64

timeStepOut = 1000
timeStepEnd = 1000

# Number of OpenMP threads
numOfThreads = 1
```

   Note that choosing e.g., `Re` too high while having `nx` too low, the simulation may become instable. Instabilities show either in the simulation finishing oddly fast or by ParaView displaying nonphysical artifacts.

3. Save changes typing *STRG + X + ENTER*.

4. Now we can repeat the steps of subsection 3.2.

4 Running on external GPU (GBAR)

4.1 Set up GBAR

1. Connect to DTU-network via VPN

   ```
   sudo openconnect --useragent=AnyConnect --os=win \
   --no-dtls https://vpn.dtu.dk
   ```

   Note that connecting to VPN requires 2-factor authentication (requires your phone).

2. Open a new terminal and log into GBAR:

   ```
   ssh DTUuser@login1.gbar.dtu.dk
   ```

   Alternatively you can use `ssh DTUuser@login2.gbar.dtu.dk`. Replace `DTUuser` e.g., with `nluho` for Nils Horneff.

3. Access GPU node by typing one of the three:

   ```
   voltash   # V100 Node (2 GPUs, 16GB VRAM each)
   sxm2sh    # V100-SXM2 Node (4 GPUs, 32GB VRAM each)
   a100sh    # A100 Node (2 GPUs, 40GB VRAM each)
   ```

   check the capacity of the chosen GPU node by typing:

   ```
   nvidia-smi
   ```

   If a GPU is busy, select another available GPU on the chosen node:

   ```
   export CUDA_VISIBLE_DEVICES=0
   ```

   Replace `0` with an available GPU ID (available means here that it has available VRAM). E.g., since `voltash` has 2 GPUs, we can select for this node between the GPU IDs `0` and `1`. If all GPUs on the chosen node are on full throttle, simply switch to another GPU node by typing either `voltash`, `sxm2sh`, or `a100sh`.

4. Clone the VF repository and navigate into it (analogous to section 3):

   ```
   git clone https://git.rz.tu-bs.de/irmb/virtualfluids.git
   ```

5. Navigate into the cloned repository:

   ```
   cd /path/to/virtualfluids
   ```

   Replace `/path/to/virtualfluids` with the actual repository location.

6. Load required packages using `module load`:

   ```
   module load gcc/12.4.0-binutils-2.42
   module load mpi/5.0.6-gcc-12.4.0-binutils-2.42
   module load cuda/12.8.0
   module load cmake/3.31.6
   ```

   Note that each package has various versions available, which can be checked using e.g., `module avail mpi`. Choosing a wrong set of modules (e.g., an old mpi-version with a new cuda version) may lead to a compatibility error from VF. But don't worry, this is not severe. Simply use e.g., `module unload mpi/(...)` to unload a loaded module and choose a version that is compatible with the other modules. Or just use my suggested set of packages from above. You can verify the loading of the modules using: `module list`. Note that after switching to another GPU node, the modules need to be reloaded.

7. Compile VF with GPU preset:

   ```
   rm -rf build # remove possible old build folder
   mkdir build && cd build
   cmake --preset=make_gpu -DCMAKE_CUDA_ARCHITECTURE=70 ..
   make -j 8
   ```

   For optimal GPU perfomance, the option `DCMAKE_CUDA_ARCHITECTURE=*` needs to be adjusted to the GPU-specific CUDA architecture. The default value here is `52` and works okay in general. The Tesla V100-SXM2 32GB and the Tesla V100 16GB are both part of the Volta architecture (7.0) and the Tesla A100 40GB is part of the Ampere architecture (8.0), thus we should choose `70` and `80`, respectively. The make flag `8` allows to speed-up the (time-consuming) make-process: e.g., `make -j 8` runs the make-process on 8 cores. Choosing `make -j$(nproc)` enables automated adjustment of the used number processors to the available number of CPU processors.

4.2 Run the Lid-Driven Cavity Simulation

1. Copy the configuration file of the test case into the `build/bin/` folder:

```
cd .. # navigate back into main repository
cp apps/gpu/DrivenCavity/drivencavity_1level.cfg build/bin/
```

2. Move to the `build/bin/` folder (contains the executables) and run the simulation:

```
cd build/bin/
./DrivenCavity drivencavity_1level.cfg
```

   Once the simulation runs successfully, the interactive node will display the simulation progress to the terminal (other than a regular GPU node).

4.3 Postprocessing with ParaView

1. Open a new terminal and copy the simulation results to the local machine (we need to use a terminal of the local machine, not a gbar terminal, for this):

```
rsync -avz DTUuser@login1.gbar.dtu.dk:\
path/to/virtualfluids/build/bin/output/DrivenCavity_uniform \
path/to/simulation_results_local
```

   Replace `path/to/simulation_results_local` with the actual destination path on the local machine (e.g., laptop) for storing the simulation results. Also replace `DTUuser` with you DTU username.

2. Open paraview (recommended version is v5.13.2, since older versions may not work):

```
paraview
```

3. Load the result file clicking: *File* → *Open* and then select:
   `LidDrivenCavity_bin_ID_0.pvd` within the folder
   `path/to/simulation_results_local/DrivenCavity_uniform/`.
   The `.pvd` file is a metadata file referencing time-series files in `.vtu` or `.vtk` format.

4. Click *Apply* to load and visualize the simulation results. Press the play-button to go through the snapshots for each written time-step. The results should be qualitatively identical to the CPU test-case:

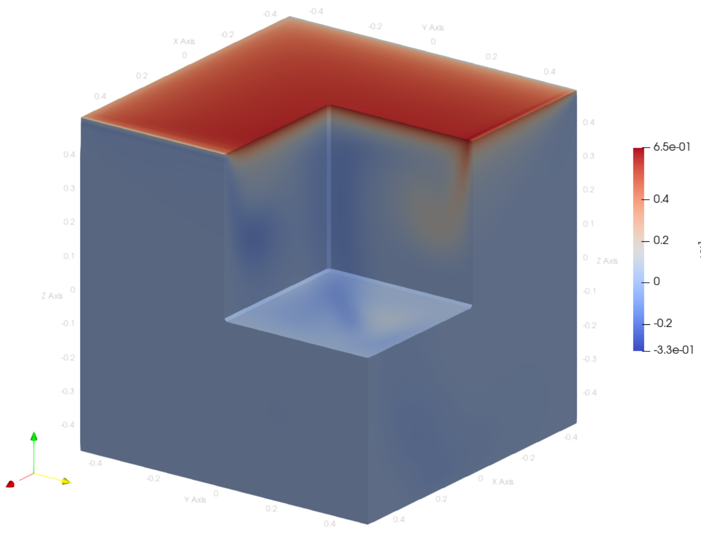

Figure 2: ParaView visualization of the 3D Lid-Driven Cavity test case using GPU-based VF.

4.4 Edit Simulation Parameters

Adjusting the same parameters as in the CPU test-case requires a little more work for the GPU case here. This is because the default configuration file for the GPU lid-driven cavity case only offers to adjust the LBM-velocity:

```
# SPDX—License—Identifier: CC—BY—4.0
# SPDX—FileCopyrightText: Copyright  VirtualFluids Project contributors,
    see AUTHORS.md in root folder
refine = false

output_path = ./output/DrivenCavity_uniform

velocityLB = 0.032
```

Thus, we first shall edit the `.cpp` file that defines how the GPU specific configuration file can be set up. This will enable us to change the desired parameters:

1. Open the `.cpp` file in an editor:

   ```
   nano path/to/virtualfluids/apps/gpu/DrivenCavity/DrivenCavity.cpp
   ```

   Replace `path/to/virtualfluids` with the actual repository location.

2. Declare those variables that you want to add to the configuration file as "writable". E.g., for being able to define the Reynoldsnumber in the configuration file, you have to turn this:

   ```
   const real reynoldsNumber = 1000.0;
   ```

   into this:

   ```
   real reynoldsNumber = 1000.0;
   ```

3. Enable the assignment of parameters to the parameter class, e.g. add:

   ```
   if (config.contains("reynoldsNumber"))
      reynoldsNumber = config.getValue<real>("reynoldsNumber");
   ```

4. Save changes typing *STRG* + *X* + *ENTER*.

5. Since we made changes to source code (we changed a .cpp-file), we now have to go back to the main repository folder and repeat the make process:

   ```
   cd /path/to/virtualfluids
   rm -rf build # remove possible old build folder
   mkdir build && cd build
   cmake --preset=make_gpu -DCMAKE_CUDA_ARCHITECTURE=70 ..
   make -j 8
   ```

6. Now we have to copy the configuration file into the `build/bin/` folder again:

   ```
   cd .. # navigate back into main repository
   cp apps/gpu/DrivenCavity/drivencavity_1level.cfg build/bin/
   ```

7. Now we can add the specified simulation parameters to the configuration file. For this, we need to open the configuration file using an editor:

   ```
   cd /path/to/virtualfluids/build/bin/
   nano drivencavity_1level.cfg
   ```

   Then we can add e.g., the Reynoldsnumber to the changeable parameters:

   ```
      reynoldsNumber = 500.0
   ```

8. Re-running the simulation will now use the Reynoldsnumber of 500 instead of the default of 1000.

Using the same procedure described here, we can analogously add the parameters `length`, `velocity`, `numberOfNodesX`, `timeStepOut`, or `timeStepEnd`, if desired.

4.5 General Debugging Advice

For the sake of clarity, this guide lists long commands across multiple lines using the bash-script line-braker "\". However, when copy-pasting the above listed commands into terminal, the interactive node sometimes comes up with an error. In this case, this can be solved by simply deleting the linebraker and bringing the pasted multiple-line command into one line.

5 Performance Benchmarking

Measuring the performance between the different available CPUs and GPUs, both, locally and remote, reveils the strength of using GPUs for LBM over CPUs. Also, we can see the difference between different GPU types and generations. The performance is measured here in time-so-solution (s) as well as in Node Updates Per Second (NUPs), which is a performance metric commonly used in Lattice Boltzmann Method (LBM) simulations and other computational fluid dynamics (CFD) applications.

Hardware	**Time-to-Solution (s)**	**NUPS (Million)**
CPU: Intel i9 (1 Thread)	620.00	4.16
CPU: AMD Ryzen 9 9950X (1 Thread)	343.00	7.64
CPU: AMD Ryzen 9 9950X (16 Threads)	66.00	39.40
GPU: NVIDIA A100-PCIE-40GB	1.73	1,658.60
GPU: Tesla V100-SXM2-32GB	1.47	1,950.10
GPU: Tesla V100-PCIE-16GB	1.44	2,001.50
GPU: NVIDIA RTX 5090	0.27	10,852.30

Table 4: Performance comparison for $Re = 1000$, grid $63 \times 63 \times 63$, 10,000 steps

Important note: for accurately measuring the Time-to-Solution (TTS), the data output intervall (`timeStepOut` or `tout`) should be chosen bigger than 1,000; ideally 5,000, or 10,000.

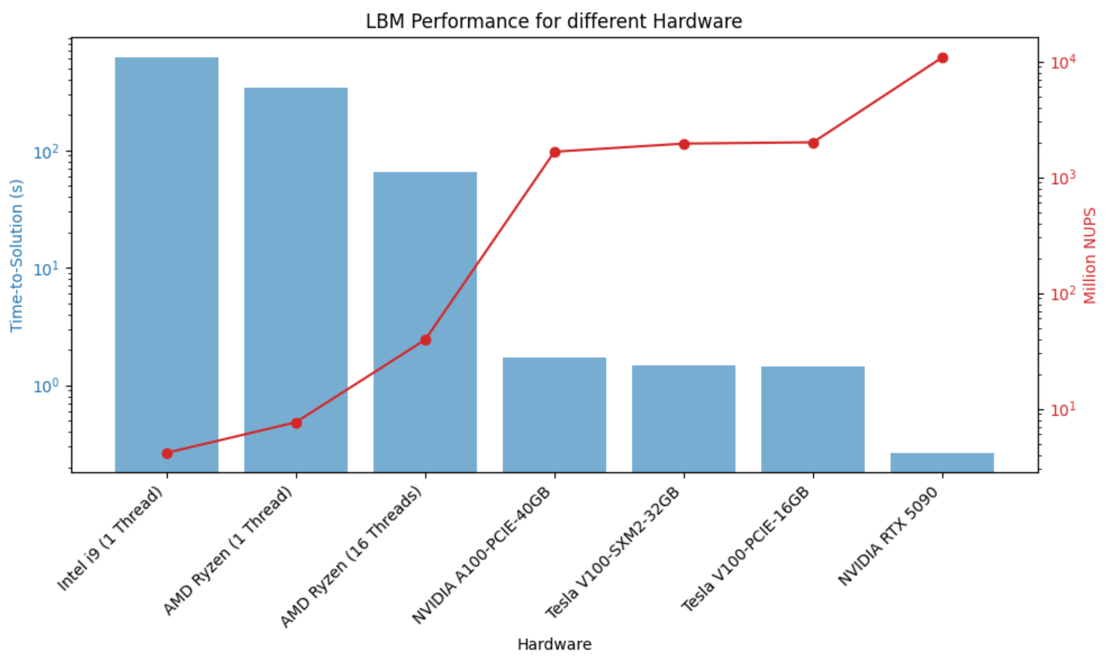

Figure 3: LBM performance using different hardware, i.e., comparing different CPUs with varying number of threads and various GPUs.