Illinois Institute of Technology
8 May 2022

H. E. Greenblatt
A20324648

CS 597 Report

1 Introduction.

The broad purpose of this work is to investigate performance and power profiling on differing architectures for workloads relevant to high-performance scientific computing. A wide variety of architectures are available, including CPUs, GPUs, FPGAs, and special-purpose accelerators; since each requires a specific approach to programming and it is not generally trivial to port simulations from one to another, designers of large-scale computational studies must be able to make intelligent decisions about which is most suited to their workload. Furthermore, once an architecture has been chosen, trade-offs must be made in the design (for example, between computation and I/O, or simulation and analysis) guided by the capabilities of the hardware. All of these decisions must be guided by real-world data. Are the performance gains of offloading a particular computation to an accelerator greater than the cost of moving the data? It depends on the CPU, the accelerator, the nature of the computation, and the problem size; premature optimization is the root of all evil.

The narrow purpose of this work is to analyze the performance and power use of several benchmarks based on a combination of C++ simulation and, via its C API, Python (including both NumPy and TensorFlow). This combination permits significant, flexible data analysis to be done *in situ*, gaining an advantage in the traditional computation-I/O tradeoff in a highly tunable manner[4]; it therefore makes these benchmarks particularly interesting candidates for analysis, especially of data movement through the memory hierarchy and between devices.

Three benchmarks were originally considered:

1. a mini-application[2] combining C++ and Python developed for the Argonne Leadership Computing Facility's "Simulation, Data, and Learning" workshop,
2. TensorFlowFoam[3], a C++-only app integrating TensorFlow directly into OpenFOAM by way of its C API, and
3. PythonFOAM[4], a C++/Python app combining OpenFOAM (a computational fluid dynamics toolbox) and Python.

Analysis of the SDL mini-app was performed mostly by other people on the project, while TensorFlowFoam was judged low-priority due to its limited flexibility and version incompatibility with the available TensorFlow installation. This report therefore concerns the PythonFOAM analysis.

2 History.

Data collection and most development was performed on the Argonne Leadership Computing Facility's ThetaGPU system[5]. ThetaGPU consists of 24 NVIDIA DGX A100 nodes, each with two 64-core AMD Rome CPUs and eight NVIDIA A100 GPUs. Present analysis has been limited to parallelization on single nodes, although PythonFOAM is parallelizable across multiple nodes.

The authors of PythonFOAM provide a Docker container for dependency management; however, this was incompatible with profiling needs. Dependency management was handled with the package manager Spack[6].

Spack is popular in the HPC world for its strong versioning capabilities, compatibility with the Environment Modules[7] system, and ability to run without root.

Some dependencies were not installed with Spack, but supplied by ALCF for ThetaGPU. The ThetaGPU versions of TensorFlow and Open MPI, specifically, are special builds configured for ThetaGPU's hardware. In addition, using the system TensorFlow requires the use of the system conda, and Spack documentation recommends using the system OpenSSL for security reasons (although in fact ThetaGPU's OpenSSL is well behind the current version).

2.1 Building and running PythonFOAM.

For readability, this section will present the *final, ideal* build process. This is what is necessary to produce results; it is not what was done on the first try (or second or third). The mistakes and missteps that actually occurred, their symptoms, and their resolutions will be discussed in the next section.

All of the following instructions should take place within a ThetaGPU job allocation, not on a service node.

2.1.1 Dependency installation.

First, clone Spack from git (as described in its documentation[8]):

```
$ git clone -c feature.manyFiles=true https://github.com/spack/spack.git
```

Next, configure Spack to use the system MPI and OpenSSL, by creating a file `etc/spack/packages.yaml` in the Spack repo with the following contents:

```
packages:
openmpi:
        externals:
                - spec: openmpi@4.1.1
                  modules:
                          - openmpi/openmpi-4.1.1_ucx-1.11.2_gcc-9.3.0
openssl:
        externals:
                - spec: openssl@1.1.1f
                  prefix: /usr
        buildable: False
```

Next, activate Spack and use it to install OpenFOAM:

```
$ source spack/share/spack/setup-env.sh
$ spack install openfoam-org
```

Next, load the newest TensorFlow conda module:

```
$ module load conda/2021-11-30
```

(While the default ThetaGPU `openmpi` module is version 4.0.5, the `conda/2021-11-30` module depends on Open MPI version 4.1.1; activating the `conda` module will automatically change the `openmpi` module version and print a message to this effect. N.B.: ThetaGPU's `conda/tensorflow` module is older, not newer, than the dated modules, and ThetaGPU support staff recommended against using it.)

Finally, activate Conda:

```
$ conda activate
```

All of the Python packages required by PythonFOAM are already installed in the conda environment, so dependency installation is now complete. However, if for some reason it becomes necessary to install additional packages (as it was for the SDL mini-app), this should be done in a Python virtual environment created and activated for the purpose:

```
(conda/2021-11-30/base) $ python -m venv --system-site-packages <virtualenv name>
(conda/2021-11-30/base) $ source <virtualenv name>/bin/activate
```

The `--system-site-packages` flag indicates that the virtualenv should inherit the packages installed in the currently active environment (in this case, conda's).

2.1.2 Environment configuration.

Installation is now complete, and the environment is correctly configured for building and running PythonFOAM. However, on subsequent logins, the environment will need to be reconfigured, as follows:

```
$ module load conda/2021-11-30
$ conda activate
(conda/2021-11-30/base) $ source <virtualenv name>/bin/activate # from the directory
    containing the virtualenv--if applicable
```

2.1.3 A brief aside on OpenFOAM.

OpenFOAM was designed by physicists for physicists. It has a user's guide[9], but this is somewhat opaque. Users who are experienced programmers but not domain experts are likely to be frequently surprised, because OpenFOAM's architecture has limited overlap with the conventions of software development and deployment in Linux (or indeed any other major OS). In particular, to name just a few peculiarities relevant to the following build instructions:

- Rather than installing binaries and shared libraries in the usual manner, or even providing its own module files, OpenFOAM prefers to bundle everything in a single "installation directory" and provide custom shell scripts for wrangling `PATH`, `LD_LIBRARY_PATH`, and a large number of OpenFOAM-specific environment variables. Although the Spack package provides a module which handles setting and unsetting this type of configuration more robustly, some software depends on the shell scripts directly or is otherwise brittle.

- It bundles its own compiler wrapper, `wmake`, for reasons which are not described in the documentation. wmake is among this brittle software; compilation should be performed using the shell scripts and not the module provided by Spack.

- Rather than building in the working directory (or to a specified output), wmake places binaries in the "installation directory" (under Spack, `~/OpenFoam/<username>-8`, although this is not in fact where OpenFOAM is installed).

The bespoke fragility of this architecture disturbs me deeply and I am grateful to Dr. Romit Maulik for his experience and advice.

2.1.4 Building the PythonFOAM example solvers.

Clone PythonFOAM from git:

```
$ git clone https://github.com/argonne-lcf/PythonFOAM.git
```

In the PythonFOAM repo is a script called `prep_env.sh`. This sources OpenFOAM's shell script and exports several variables concerning the relevant Python libraries needed for compilation and must be edited appropriately. On ThetaGPU, this is as follows:

```
source <path to spack repo>/opt/spack/linux-ubuntu20.04-zen2/gcc-9.3.0/openfoam-org
    -8-6taoyscxhdfq2tsuiqbbbrmcduquc7cx/etc/bashrc
export PYTHON_LIB_PATH=/lus/theta-fs0/software/thetagpu/conda/2021-11-30/mconda3/lib
export PYTHON_BIN_PATH=/lus/theta-fs0/software/thetagpu/conda/2021-11-30/mconda3/bin
export PYTHON_INCLUDE_PATH=/lus/theta-fs0/software/thetagpu/conda/2021-11-30/mconda3/
    include/python3.8/
export NUMPY_INCLUDE_PATH=/lus/theta-fs0/software/thetagpu/conda/2021-11-30/mconda3/
    lib/python3.8/site-packages/numpy/core/include
export PYTHON_LIB_NAME=lpython3.8
```

(Since Python and NumPy—the only packages with APIs called in the C++ code—are supplied by the conda env, these paths are correct regardless of whether a virtualenv is used on top.)

Next, actually source (don't run) the script:

```
$ source prep_env.sh
```

Change to the solver directory of choice (`PODFoam`, `APMOSFoam`, or `AEFoam`, under `Solver_Examples` in the PythonFOAM repo) and compile:

```
$ wclean && wmake
```

Binaries will be in `~/OpenFOAM/<username>-8/platforms/<build>/bin/`.

2.1.5 Running the PythonFOAM example solvers.

Source `prep_env.sh`, change to the `Run_Case` directory in the solver of choice (or supply it as `-case`), and run the corresponding binary. Logging information is printed to the console, while results are written to numerically named "time directories" within the case directory.

2.2 Issues.

The following problems are treated in the approximate order of their discovery. Transient errors—due to environmental misconfiguration—can be found in the appendix.

- During the build, 'wmkdepend' printed the error `could not open 'kinematicMomentumTransportModel.H'` for each solver.

A brief investigation concluded (and discussion with Dr. Romit Maulik, the lead PythonFOAM author, confirmed) that this was irrelevant for our purposes.

- AEFoam, although not PODFoam or APMOSFoam, immediately exited with a fatal error:

```
Cannot find file "points" in directory "polyMesh" in times "0" down to constant
```

This message means that the block mesh has not been generated from the case definition. Running the `blockMesh` command provided by OpenFOAM in the case directory resolved the issue.

- PODFoam and APMOSFoam ran successfully for a while, then exited with fatal errors:

```
Wrong token type - expected scalar value, found on line 0: word 'nan'
```

This was evidently a numerical issue.

In the initial builds, the latest version of the `openfoam` Spack package had been installed. This is the version of OpenFOAM commonly referred to as "openfoam.com", which developed and distributed by OpenCFD Limited at that address; there is also an "openfoam.org", developed mostly by CFD Direct and distributed by the OpenFOAM Foundation at *that* address. The latest `openfoam` version was v2112, but reconsulting the PythonFOAM documentation showed that only "openfoam.com" v2012 and v2106 and "openfoam.org" version 8 were supported (or, more specifically, "should compile without changes"). However, downgrading to and recompiling against v2106 failed to resolve the issue.

Communication with Dr. Maulik revealed that PythonFOAM had never been *tested* on any "openfoam.com" version, only the "openfoam.org". Changing to and recompiling against the latest `openfoam-org` Spack package resolved the issue. The solvers could *apparently* now run to completion.

- Time to completion was unacceptably long.

A rough estimate of the total runtime of PODFoam, the simplest of the example solvers (made by linear extrapolation from the last timestep reached in the available job time to the end timestep defined in the problem) put it at approximately 300 days. Even if ThetaGPU jobs were not limited to twelve hours of wall-clock time, this would be wildly impractical to profile.

However, the original PythonFOAM paper described good scaling results, so an attempt to run the PODFoam and APMOSFoam solvers in parallel mode was made. To run an OpenFOAM solver in parallel, it is first necessary to decompose the run case by configuring the `system/decomposeParDict` file in the case directory, then running the `decomposePar` command provided by OpenFOAM. This will write "processor directories" within the case directory. The solver can then be run as

```
$ mpiexec -np <n> <solver> -parallel
```

where *solver* is the binary and n is the number of processors configured for the decomposition. (To actually analyze the data, the processor directories can subsequently be recomposed with the `reconstructPar` command, although this was superfluous for our purposes.)

- Running PODFoam or APMOSFoam in parallel immediately crashed with a symbol lookup error:

```
symbol lookup error: /lus/theta-fs0/software/thetagpu/openmpi/openmpi-4.1.1_ucx
    -1.11.2_gcc-9.3.0/lib/openmpi/mca_coll_han.so: undefined symbol:
    mca_coll_base_colltype_to_str
```

It was immediately apparent that this was an MPI compatibility error. The conda environment used Open MPI version 4.1.1, but in the initial builds, Spack had been configured with the default ThetaGPU Open MPI, 4.0.5. Recompiling the `openfoam-org` package (and its dependencies) with the correct MPI configuration resolved the issue.

- PODfoam and APMOSFoam ran successfully in parallel for a while, then exited when one of the threads aborted:

```
munmap_chunk(): invalid pointer
Collecting snapshots iteration:  73
Collecting snapshots iteration:  73
Collecting snapshots iteration:  73
[thetagpu09:3613607] *** Process received signal ***
[thetagpu09:3613607] Signal: Aborted (6)
[thetagpu09:3613607] Signal code:  (-6)
[thetagpu09:3613607] [ 0] /lib/x86_64-linux-gnu/libc.so.6(+0x46210)[0x7fab586df210]
[thetagpu09:3613607] [ 1] /lib/x86_64-linux-gnu/libc.so.6(gsignal+0xcb)[0
    x7fab586df18b]
[thetagpu09:3613607] [ 2] /lib/x86_64-linux-gnu/libc.so.6(abort+0x12b)[0x7fab586be859
    ]
[thetagpu09:3613607] [ 3] /lib/x86_64-linux-gnu/libc.so.6(+0x903ee)[0x7fab587293ee]
[thetagpu09:3613607] [ 4] /lib/x86_64-linux-gnu/libc.so.6(+0x9847c)[0x7fab5873147c]
[thetagpu09:3613607] [ 5] /lib/x86_64-linux-gnu/libc.so.6(+0x986cc)[0x7fab587316cc]
[thetagpu09:3613607] [ 6] /lus/theta-fs0/software/thetagpu/conda/2021-11-30/mconda3/
    lib/libpython3.8.so.1.0(PyTuple_SetItem+0x46)[0x7fab596065c6]
[thetagpu09:3613607] [ 7] /home/hgreenbl/OpenFOAM/hgreenbl-8/platforms/
    linux64GccDPInt32-spack/bin/PODFoam(+0x31b23)[0x563427bb1b23]
[thetagpu09:3613607] [ 8] /lib/x86_64-linux-gnu/libc.so.6(__libc_start_main+0xf3)[0
    x7fab586c00b3]
[thetagpu09:3613607] [ 9] /home/hgreenbl/OpenFOAM/hgreenbl-8/platforms/
    linux64GccDPInt32-spack/bin/PODFoam(+0x3321e)[0x563427bb321e]
[thetagpu09:3613607] *** End of error message ***
Snapshot collection wall time: 1.19 ms
IO wall time: 0.00 ms
Snapshot collection wall time: 1.31 ms
IO wall time: 0.00 ms
Snapshot collection wall time: 1.31 ms
IO wall time: 0.00 ms
--------------------------------------------------------------------------
Primary job  terminated normally, but 1 process returned
a non-zero exit code. Per user-direction, the job has been aborted.
--------------------------------------------------------------------------
--------------------------------------------------------------------------
orterun noticed that process rank 3 with PID 0 on node thetagpu09 exited on signal 6
    (Aborted).
--------------------------------------------------------------------------
```

This was non-trivial to debug. The crash only occurred in parallel runs, never serial runs. The exact timing was nondeterministic. There were no debug symbols in the executable: OpenFOAM "debug" configuration is largely concerned with runtime logging[10], the wmake compiler wrapper is poorly documented, and while there exists a `WM_COMPILE_OPTION` environment variable it seems to be intended for building OpenFOAM itself.

It was *possible* that this bug was related purely to parallelism. However, the invalid pointer and failure in `PyTuple_SetItem` were suggestive of a segfault in the Python interpreter (which would have sent the SIGABRT to the PODFoam executable). PythonFOAM instantiates a separate Python interpreter for each process, so it seemed possible that the increased memory pressure of a parallel run was triggering garbage collection that did not occur in serial runs and thereby revealing an underlying reference-counting error—particularly as `PyTuple_SetItem` has unusual ownership behavior[11][12].

On observation of the actual code in the solvers, it was clear that memory management was incomplete: most of the Python objects were, apparently deliberately, leaked. The majority of calls to `PyTuple_SetItem` were in a loop: the tuple in question served as the arguments to the main Python 'work' function periodically called from C, and the C code repeatedly pointed the same index of the same tuple to the same Python object. (The Python C API recommends against this.[13]) It's not obvious how `PyTuple_SetItem` behaves if the inserted and discarded objects are the same, and the implementation[14] uses the undocumented[15] XSETREF macro. The obvious tool for diagnosing memory management errors would normally be valgrind, but while Open MPI and Python both[16] have[17] valgrind suppression files (although the accuracy of valgrind through Python's internal memory management, or the reproducibility of any bug under debug-friendlier PYTHONMALLOC[18] configurations, would still be dubious), OpenFOAM does *not* and is decidedly not valgrind-clean.

To recap, this bug was a parallel nondeterministic memory error in leaky C interfacing with a garbage-collected interpreter through an incompletely documented API without debugging symbols. Fortunately, careful thought and print statements were still readily available.

At this point, the project team had the opportunity to meet with Dr. Maulik, who helpfully discussed both benchmarks—including, particularly, the discrepancy between our PythonFOAM results and the ones described in his paper, and the mechanism of OpenFOAM's parallelism. I was concerned that my understanding might have been incomplete, partly due to our poor results and partly because elements of the discussion in the PythonFOAM paper seemed inconsistent with inspection of the code in the PythonFOAM GitHub repository. (In particular, while the paper did not specify total runtimes, job submission scripts in the then-current version of the repo[19] suggested that they should have been two orders of magnitude better than we were getting, and the "Scotch" decomposition method[20] specified in the paper did not match the "simple" method used in the case directories[21].)

Dr. Maulik's group had not experienced the same crashes. While he was therefore uncertain of the etiology, he was able to make several suggestions for workarounds: if I was correct about the bug being triggered by increased memory pressure, then decreasing the `writeInterval` setting[22] of the system/controlDict file in the case directory might avoid it by flushing data to disk more frequently, while either increasing `writeInterval` or reducing `endTime`[23] would reduce time to completion regardless of parallelism. (We had considered reducing the end time, but were held back by lack of domain knowledge of computational fluid dynamics general and OpenFOAM in particular; one of the other topics of discussion in the meeting was a set of crashes in the SDL mini-app which Dr. Maulik explained were caused by physically inappropriate parameter choices. He stated that the example solvers were purely iterative and reducing the end time could not cause computational issues, but we did not discuss specific value ranges—an omission I would later regret.)

Perhaps more importantly, Dr. Maulik confirmed that the code available from GitHub was *not* the original code used in his paper, but a "reduced", "simplified" version of each solver, and kindly sent a copy of the original PODFoam (including its case directories). With a 'good' version of the code to compare against in addition to a good candidate for the bug, it became significantly easier to identify and patch, if not wholly explain, the problem. One of the arguments in the problematic tuple was the process rank, which remained constant throughout the run. In the reduced PODFoam, the rank was stored in a single `PyObject` (specifically, a `PyLongObject`), to which the argument tuple was continually re-pointed; in the original PODFoam, rank was stored instead in a C integer, and a new `PyObject` was created from it for each Python iteration. Duplicating this approach in the reduced PODFoam—thus—

```
    diff --git a/Solver_Examples/PODFoam/PythonComm.H b/Solver_Examples/PODFoam/
PythonComm.H
    index b31660c..566fe86 100644
    --- a/Solver_Examples/PODFoam/PythonComm.H
    +++ b/Solver_Examples/PODFoam/PythonComm.H
    @@ -92,6 +92,7 @@ else

         clock_gettime(CLOCK_MONOTONIC, &tw1); // POSIX
    +    rank_val = PyLong_FromLong(rank);
         PyTuple_SetItem(snapshot_args, 0, array_2d);
         PyTuple_SetItem(snapshot_args, 1, rank_val);
         (void) PyObject_CallObject(snapshot_func, snapshot_args);
    diff --git a/Solver_Examples/PODFoam/PythonCreate.H b/Solver_Examples/PODFoam
/PythonCreate.H
    index 2508ca8..5e06e8b 100644
    --- a/Solver_Examples/PODFoam/PythonCreate.H
    +++ b/Solver_Examples/PODFoam/PythonCreate.H
    @@ -57,7 +57,8 @@ volScalarField vpod_(U.component(vector::Y));
     volScalarField wpod_(U.component(vector::Z));

     // To pass rank to Python interpreter
    -PyObject *rank_val = PyLong_FromLong(Pstream::myProcNo());
    +int rank = Pstream::myProcNo();
    +PyObject *rank_val;
     PyObject *array_2d(nullptr);

     ///int encode_mode = 0;
```

eliminated the crash. (This change should not introduce a memory leak, since `PyTuple_SetItem`'s ownership behavior, as referenced above, discards the tuple's reference to the previous PyObject as each new one is assigned; however, this has not been verified in profiling due to the other leaks in the code.) Similar patches applied to APMOSFoam and AEFoam. At this point, solvers could run to completion in parallel.

- Time to completion was still unacceptably long, and scaling was poor.

The AEFoam solver was set aside at this point due to apparent NaN poisoning and a TensorFlow issue that appeared in parallelized runs. Rough scaling performance data was collected for the other two example solvers. For each scale tested, the case directories were reconfigured and decomposed with the corresponding number of subdomains (using the "simple" decomposition method and incrementing the x and y coefficients by factors of 2 alternately), and the solvers were run on the decomposed cases in ThetaGPU job allocations with fixed wall-clock time limits (10 minutes for PODFoam, 20 minutes for the slower APMOSFoam). Speed estimates were calculated by dividing the simulated time of the last time directory written by the timestep length and the total job time. Results are shown in fig. 1.

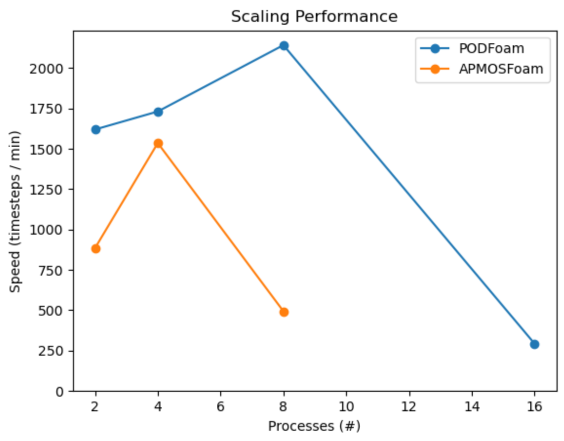

Figure 1: Scaling performance of PODFoam and APMOSFoam on ThetaGPU.

PODFoam ceased showing improvement above only 8 processes; APMOSFoam, only 4. This is in stark contrast with the scaling results presented in the PythonFOAM paper, in which APMOSFoam displayed near-ideal scaling up to the maximum available 128 ranks on multiple CPU architectures. These (7(c) and 8(c) of [1]) are reproduced in fig. 2.

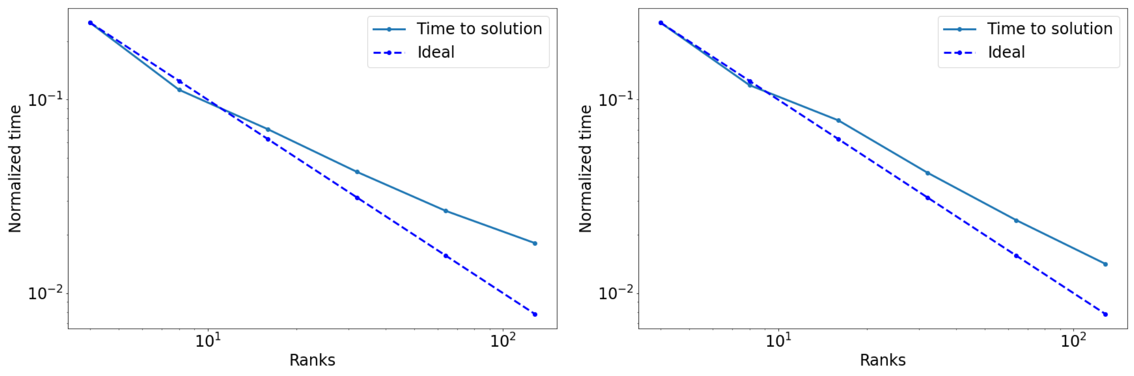

Figure 2: Scaling performance of APMOSFoam on Bebop (BDW and KNL nodes, respectively).

Normally, very bad scaling would suggest that the problem size might be *too* small—so small that the overhead of parallelization overwhelms any computational advantage. In this case, however, the high absolute runtimes seemed to indicate otherwise.

The results from the PythonFOAM paper were not directly comparable to these: they were collected on a different system equipped with different hardware (Bebop, at Argonne's Laboratory Computing Resource Center[24]), they were supplied different case definitions, and they were running different code. The latter factors seemed more likely to account for the differences, but not *certain*. An examination of the `system/blockMeshDict` files[25] of the respective case directories did not reveal any obviously large spatial discrepancies. Therefore, in order to bisect the problem, an attempt was made to run the original PODFoam on ThetaGPU.

- The original code did not build on ThetaGPU.

The original Makefiles (or rather, wmake options) had hard-coded paths to the Python installation used by the author; replacing them wholesale with the reduced Makefiles, and setting the appropriate variables with `prep_env.sh`, resolved the issue. (Compiling the original code produces a large number of warnings about potentially uninitialized variables; these are due to an inexplicable pattern of `#include` usage and are spurious.)

- The original PODFoam almost immediately crashed with a floating point exception.

This was quickly narrowed down to the initialization of NumPy through its C API (a call to `import_array` [26]). Research revealed that OpenFOAM appears to interact poorly with NumPy: initialization (in certain circumstances, on certain architectures) signals a floating point exception which OpenFOAM, by default, traps and dies on. Fortunately, there exists a workaround involving disabling the default OpenFOAM root case initialization.[27] Unfortunately, this does not address the root cause of the problem, and leaves *all* of OpenFOAM's default signal handlers turned off.[28] Fortunately, there exists a narrower workaround that disables only the floating-point signal handler only during NumPy initialization.[29] Unfortunately, while methods to temporarily disable signal handlers are available in the "openfoam.com" library[30], no such functionality exists in the "openfoam.org" library[31], to which this project is restricted.

Implementation of the first workaround was, although not ideal, sufficient to resolve the issue. (It was later incorporated into the reduced PODFoam[32] although the use of `PY_ARRAY_UNIQUE_SYMBOL` is not actually necessary in this case.)

- The original PODFoam almost immediately crashed with a segmentation fault.

This was just as quickly narrowed down to the declaration of the array holding the numerical data sent to the Python function; apparently Bebop's stack is larger than ThetaGPU's. Moving the allocation to the heap—thus—

```
diff --git a/createFields.H b/createFields.H
index 21dda5a..6179228 100644
--- a/createFields.H
+++ b/createFields.H
@@ -87,7 +87,7 @@ printf("Got rank\n");

 // Placeholder to grab data before sending to Python
 int num_cells = mesh.cells().size();
-double input_vals[num_cells][3];
+auto input_vals = new double[num_cells][3];

 // Number of POD modes
 truncation = 5;
```

eliminated the segfault. (The corresponding declaration in the reduced PODFoam[33] did not trigger the segfault because the reduced case definition is smaller.) Finally, the original PODFoam solver appeared to be running correctly.

- The case definition was simply too long.

The original PODFoam ran to completion in under thirty minutes. Inspection of the output time directories revealed–compared to the reduced PODFoam—very few of them at very round simulated times. Inspection of the system/controlDict files revealed that whereas the "reduced" PODFoam had an end time of 10,000 and a timestep length of 1×10^{-5}[34], for a total of *one billion* timesteps, the original PODFoam had an end time of 10 and a timestep length of 0.2, for just 50. The poor scaling was quite likely because the reduced problems were too small—but the high time to completion was because they were much too *long*.

(I should certainly have realized this earlier; getting the original PODFoam running was, although informative, completely unnecessary. I was not primed to notice—the results in the paper are actually for APMOSFoam and AEFoam, not PODFoam, and the exact problem lengths are not given, but the performance figures and timestamp precision strongly imply cases of at least a few thousand timesteps—but it was my own domain ignorance that led me to think that 'one billion' was an appropriate problem size and my own forgetfulness that led me to fail to make this comparison after I had specifically planned to do so.)

The original PODFoam, for what it's worth, has excellent scaling, as shown in fig. 3. Since all runs could be completed, speed estimates were calculated simply by dividing the number of timesteps (50) by the wall-clock time.

There are several possible explanations for the regression at 128 processes (in comparison to the APMOSFoam results from Bebop which showed none). It may simply be the natural limit of this benchmark at this problem size (APMOSFoam possibly being larger); alternatively, it may have something to do with 128 being the maximum number of processes available on a single ThetaGPU node (APMOSFoam likely having been tested on up to 128 nodes with one process each, based on run scripts included with the original code), and/or with slight profiling overhead (since these data were collected under perf).

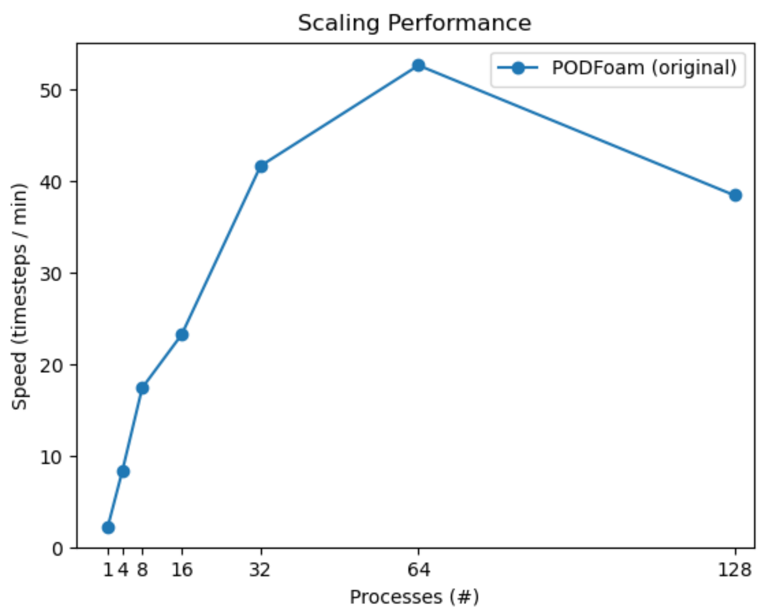

Figure 3: Scaling performance of the *original* PODFoam on ThetaGPU.

2.3 Data collection.

Performance data was collected using Mantis, a test harness[35] for automatically running applications in multiple configurations through multiple profiling tools with minimal overhead, summarizing the results in a single format, and displaying them in a corresponding dashboard[36]. Mantis (which was developed in previous work by members of this project) runs applications through "plugins", each a set of variables and hooks that permit Mantis to run the application and change its parameters and other configuration.

At present, these plugins take the form of Bash scripts. The PODFoam plugin has the following contents:

```
#!/bin/bash

BASE_LOC=/lus/grand/projects/SEEr-planning/pff

TEST_NAME="PODFOAMFULL"
TEST_TYPES=("1" "4" "8" "16" "32" "64" "128")

source mantis_utils.sh

function setgpus() {
        :
}
export setgpus

function setup() {
        export TYPE_BIN=$BASE_LOC/PODFoam
        check_bin $1 $TYPE_BIN
        export WORK_DIR=$BASE_LOC/$1rank
        if [ "$1" == "1" ]; then
```

```
                echo "Using $1 rank"
                export TYPE_ARGS=""
        else
                echo "Using $1 ranks"
                mpi_obj=$TYPE_BIN
                setup_mpi
                export TYPE_ARGS="-np $1 $mpi_obj -parallel"
        fi
        export GPU_RANGE="0"
}
```

It defines seven "types" of test, each a different scale at which to run PODFoam. The setup function is called once for each test type, and the `TEST_NAME`, `TEST_TYPES`, `WORK_DIR`, `TYPE_BIN`, and `TYPE_ARGS` variables configure Mantis's actions in a highly flexible manner—in this case, changing to a case directory pre-decomposed to the appropriate scale and running the PODFoam solver directly or through mpiexec as appropriate.

Mantis was designed with hardware comparisons in mind and treats GPU configuration as a first-class citizen, varying it independently from the test types configured in the plugin; however, PODFoam (like APMOSFoam, but not AEFoam) is CPU-only, so this plugin must include no-op GPU settings. (In addition, the GPU power profiler was disabled in the main body of Mantis for this application.)

Preliminary data has already been collected. A Python pickle file compatible with the Mantis dashboard is attached.

3 Future work.

The immediate next step is to collect full Mantis runs for analysis from all three PythonFOAM example solvers. PODFoam and APMOSFoam will run without further changes (except possibly a more appropriate problem length). Sadly, AEFoam—the most exciting of the three for the purposes of this project, given its use of potentially GPU-accelerated TensorFlow—still requires debugging. This should be the first priority.

Meanwhile, Mantis itself has certain limitations which make it less convenient for automatic data collection than it was designed to be. In particular, it achieves its minimal profiling overhead by running a large number of iterations and collecting only a few counters on each, which results in massively inflated runtimes; this can quickly run up against maximum job durations, even for benchmarks of moderate duration, minimal test configurations, and conservative choices of counters. Version 2 of Mantis is currently in development, and patching it to add a semi-automatic suspend/resume feature would streamline future work enormously. In addition, power profiling—currently not possible on ThetaGPU's AMD CPUs—will soon become available through AMD's µProf. This, too, should be integrated into Mantis.

Once meaningful data has been collected, a wide variety of analyses are possible. Simple proof-of-concept investigations might include fact-checking chip manufacturers' performance claims and calculating performance per watt, ideally on Theta or other systems as well as ThetaGPU. Once AEFoam is working, a more interesting question might be the comparison of GPU-accelerated and CPU-only TensorFlow in terms of power and performance (with special attention to the costs of data movement). Results from these steps would help to refine and expand the choice of profiling measurements for further work, possibly ending in something like exploring hyperparameter space by training multiple deep learning models in parallel on multiple GPUs, predicting the performance costs of increasing neural network size, or investigating the performance benefits of AI acceleration on computational physics solvers.

I hope to present some or all of these results as a master's thesis in Summer 2022. The official Sequence of Events has not yet been posted, but the Graduate Studies Handbook[37] indicates that deadlines will be[38]:

Application for graduation	25 June
Preliminary draft of thesis	26 July
Master's comprehensive exam	5 August
Deposit of thesis & payment of fees	8 August.

Appendix: Table of environment misconfigurations

Symptom: `--> FOAM FATAL ERROR in Foam::findEtcFiles() : could not find mandatory file 'controlDict'`

Diagnosis: The OpenFOAM environment variables have not been set. Source `prep_env.sh` (which sources the OpenFOAM shell script). Loading the OpenFOAM module will also resolve this, but not other issues.

Symptom: `--> FOAM FATAL ERROR: [1] UPstream::init(int& argc, char**& argv) : environment variable MPI_BUFFER_SIZE not defined`

Diagnosis: An environment variable which is not strictly OpenFOAM's but on which OpenFOAM can depend has not been set. Source `prep_env.sh` (which sources the OpenFOAM shell script).

Symptom: `ModuleNotFoundError: No module named 'numpy'`

Diagnosis: The conda Python is not active. Source `prep_env.sh` or load and activate conda.

Symptom: Nondeterministic segfaults with extremely long backtraces mostly in `_PyEval_EvalFrameDefault` (the Python interpreter) during parallel runs.

Diagnosis: Python/MPI version mismatch. `prep_env.sh` has been sourced (adding the conda Python to library and binary PATHs), but the conda module has not been loaded and the underlying OpenMPI version has not been changed. Load the conda module.

Symptom: `RuntimeError: module compiled against API version 0xe but this version of numpy is 0xd`

Diagnosis: There's a virtualenv active which is based on an older conda module with an older numpy installed, but the solver was compiled with `prep_env.sh` pointing to newer ones. Use an up-to-date virtualenv.

Symptom: `error in IOstream "/home/<...>/Run_Case/<...>" for operation Ostream& operator<<( Ostream&, const char)`

Diagnosis: Hit disk quota[39]. Move the case directory out of ~ and into a project directory.

References

[1] https://arxiv.org/abs/2103.09389

[2] https://github.com/argonne-lcf/sdl_ai_workshop/tree/master/05_Simulation_ML/ML_PythonC%2B%2B_Embedding/

[3] https://github.com/argonne-lcf/TensorFlowFoam

[4] https://github.com/argonne-lcf/PythonFOAM

[5] https://www.alcf.anl.gov/support-center/theta/theta-thetagpu-overview

[6] https://spack.io/

[7] http://modules.sourceforge.net/

[8] https://spack.readthedocs.io/en/latest/getting_started.html

[9] https://cfd.direct/openfoam/user-guide-v8/

[10] https://cfd.direct/openfoam/user-guide/v8-compiling-applications/#x10-820003.2.6

[11] https://docs.python.org/3.8/extending/extending.html#ownership-rules

[12] https://docs.python.org/3.8/c-api/tuple.html#c.PyTuple_SetItem

[13] https://github.com/python/cpython/blob/main/Include/tupleobject.h#L9

[14] https://github.com/python/cpython/blob/main/Objects/tupleobject.c#L111

[15] https://bugs.python.org/issue31983

[16] https://github.com/open-mpi/ompi/blob/main/contrib/openmpi-valgrind.supp

[17] https://github.com/python/cpython/blob/main/Misc/valgrind-python.supp

[18] https://docs.python.org/3/using/cmdline.html#envvar-PYTHONMALLOC

[19] https://github.com/argonne-lcf/PythonFOAM/blob/d60f84170f81004aa18dacf7f37eb4e610a97c59/Solver_Examples/PODFoam/Run_Case/runOpenFOAM-PODFoam.sh

[20] https://cfd.direct/openfoam/user-guide/v8-running-applications-parallel/#x12-860003.4.1

[21] https://github.com/argonne-lcf/PythonFOAM/blob/main/Solver_Examples/PODFoam/Run_Case/system/decomposeParDict

[22] https://cfd.direct/openfoam/user-guide/v8-controldict/#x19-1430004.4.2

[23] https://cfd.direct/openfoam/user-guide/v8-controldict/#x19-1420004.4.1

[24] https://www.lcrc.anl.gov/systems/resources/bebop/

[25] https://cfd.direct/openfoam/user-guide/v8-blockMesh/#x26-1850005.3

[26] https://numpy.org/doc/1.16/reference/c-api.array.html#importing-the-api

[27] https://github.com/pybind/pybind11/issues/1889#issuecomment-1029584553

[28] https://github.com/pybind/pybind11/issues/1889#issuecomment-1029754822

[29] https://github.com/pybind/pybind11/issues/1889#issuecomment-1029909985

[30] https://www.openfoam.com/documentation/guides/latest/api/classFoam_1_1sigFpe.html

[31] https://cpp.openfoam.org/v3/a02284.html

[32] https://github.com/argonne-lcf/PythonFOAM/commit/7db6bef1d64bd7370a459252c7c5a960eb4a185c

[33] https://github.com/argonne-lcf/PythonFOAM/blob/main/Solver_Examples/PODFoam/PythonCreate.H#L70

[34] https://github.com/argonne-lcf/PythonFOAM/blob/main/Solver_Examples/PODFoam/Run_Case/system/controlDict#L26

[35] https://gitlab.com/seer-benchmark-research/seer-benchmark-code/-/tree/master/mantis

[36] https://gitlab.com/mdooley11/mantis_dashboard_omp

[37] http://bulletin.iit.edu/grad-handbook/graduation/

[38] https://www.iit.edu/registrar/academic-calendar

[39] https://www.alcf.anl.gov/support-center/theta-and-thetagpu/theta-disk-quota