> **Warning** This artifact matches the originally submitted version of the paper, the published version of the paper had additional figures added during the review process resulting in some of the figure numbers not match exactly, namely Figure 10 here is Figure 13 in the paper, Figure 12 here is Figure 14 in the paper and Figure 13 here is Figure 16 in the paper.

This artifact generates figures of the submitted draft of “Register Tiling for Unstructured Sparsity in Neural Network Inference”. The artifact generates graphs of the paper for an intel processor with 20 cores and AVX512VL (and 2 FMA units), namely a Xeon(R) Gold 6248 CPU. The tested dataset is a collection of matrices from deep learning matrix collection (DLMC).

The list of claims: This artifact includes all scripts required to reproduce the graphs on the Intel processor. In particular, this artifact should be evaluated for the following figures/claims on the Intel XEON Gold processor:

- Figure 7 compares the performance of sparse register tiling with the intel MKL library.
- Figure 8 compares the performance of sparse register tiling with the Intel MKL library and ASpt across DLMC matrices between 60-95% sparsity ratio.
- Figure 9 compares the memory usage efficiency of the compression phase of the sparse register tiling with the CSR compression.
- Figure 10 shows how much the two steps of sparse register tiling , i.e., unroll-and-sparsejam and data compression, contribute to the overall performance (MKL GEMM is also shown as a baseline).
- Figure 12 analyzes the effect of memory loads and redundant FLOPs on the performance of sparse register tiling.
- Figure 13 analyzes the effect of code size (number of patterns) and the performance of sparse register tiling.

Some claims/figures are excluded from the artifact for the following reasons:

- This artifact does not reproduce Figure 14 and Table 2 since they require the Raspberry Pie board.
- This artifact does not generate figure 11 because obtaining enough profiling info for the figure is time-consuming, for each performance counter, the program should run several times.
- We set a 6 hours time-out to cap the evaluation time thus, the generated results do not include all DLMC matrices between 60-95% but it should be enough to reproduce the trend.

Running containerized artifact (this is by far the easiest way)

install singularity: https://docs.sylabs.io/guides/3.0/user-guide/installation.html

ensure your machine has avx512vl

```
lscpu | grep avx512vl
```

if running local, run:

```
s_setup.sh
s_run.sh
s_plot.sh
```

if running a cluster (niagara) with slurm, run:

```
s_setup.sh
s_run.sh

# Wait for all the jobs to finish then run:

s_plot.sh
```

Building and running from source

ensure your machine has avx512vl

```
lscpu | grep avx512vl
```

Download DLMC

Download the dlmc dataset, this will download it to `../dlmc`

```
sh download_dlmc.sh
```

Install python dependencies

```
pip3 install torch
pip3 install -e .
export PYTHONPATH=$(pwd):$PYTHONPATH
```

Install MKL 2021.4.0

```
wget https://apt.repos.intel.com/intel-gpg-keys/GPG-PUB-KEY-INTEL-SW-PRODUCTS.PUB &&
    add-apt-repository -y "deb https://apt.repos.intel.com/oneapi all main" && \
    apt-key add GPG-PUB-KEY-INTEL-SW-PRODUCTS.PUB && \
    rm GPG-PUB-KEY-INTEL-SW-PRODUCTS.PUB && \
    apt-get update -y && \
    apt-get install -y intel-oneapi-mkl-devel-2021.4.0
```

Generating executor/scheduler pairs used in the paper

from `spmm-nano-kernels`, run the following:

```
cd spmm-nano-kernels
python3 -m codegen.generate_ukernels
cd ..
```

Building the demo code

```
mkdir release-build
cmake -Brelease-build -DCMAKE_BUILD_TYPE=Release -DENABLE_AVX512=True .
make -j 16 -Crelease-build SPMM_demo
```

Running a single matrix

```
python3 artifact/run_matrix.py -m ../dlmc/transformer/magnitude_pruning/0.8/body_dec
```

see:

```
python3 artifact/run_matrix.py --help
```

for more details

Code overview

- `cpp_testbed/demo/SPMM_demo.cpp` : benchmarking code (best to use the `python3 artifact/run_matrix.py` wrapper)
- `spmm_nano_kernels/codegen/generate_ukernels.py` : entry point for generating scheduler/executor pairs
- `spmm_nano_kernels/codegen/base_ukernel_codegen.py` : code for generating mini-register tiles, this code in conjunction with `spmm_nano_kernels/include/Executor.h` forms the executor code
- `spmm_nano_kernels/include/MicroKernelPacker.h` : templated scheduler + data-compressor code
- `spmm_nano_kernels/include/MatMulSpecialized.h` : wrapper that contains a scheduler/executor pair
- `sbench/ilp_pad/nano_solver.py` contains the code for the mathematical model