The MILC Benchmark

1. Introduction

Lattice QCD (LQCD) applications are parts of OLCF workloads. In a typical year there would be 2-3 large (and occasionally some small) LQCD projects running at OLCF. From a computational perspective LQCD applications tend to fall into two groups: gauge generation---which is a big hybrid molecular dynamics Monte Carlo process---and analysis---where the gauge configurations generated in the previous step are processed. Typically this involves the calculation of hadronic observables constructed from quark propagators on each configuration.

In general gauge-configuration generation tends to be a strong scaling problem, spread over as many nodes as is practicable. As the node count increases, the surface to volume ratio of the local problem decreases, and the problem becomes increasingly communications bound. Factors affecting the communication can include communications latency and bandwidth. Due to the recent decrease in the ratio of memory and network bandwidths, it is common for strong scaling to top out at O(100)-O(1000) nodes leading recently to a tendency to run multiple streams of gauge generation in a single job; essentially in the style of a small ensemble (e.g. 4-streams of 512 nodes or similar).

In contrast analysis makes use of the independence of the gauge configurations and is virtually almost run in an ensemble manner. The ensemble member sizes typically are chosen to have a sufficiently large local volume per node to keep strong scaling effects to a minimum. For hadronic observables, generally the lattice Dirac Equation needs to be solved which is stencil-like and is generally memory bandwidth bound. Analysis stages may also face limitations from I/O depending on whether they write out the solutions of the dirac matrix, or its eigenspectrum.

1.1 The MILC Benchmark

The MIMD Lattice Collaboration (MILC) code has been a workhorse of LQCD calculations utilizing Highly Improved Staggered (HiSQ) quarks. The code is written in C and can utilize libraries for performance improvement including the QUDA library for GPUs and the QPhiX library for CPUs. This benchmark is adapted from the NERSC-10 Latice QCD Workflow Benchmark (https://gitlab.com/NERSC/N10-benchmarks/lattice-qcd-workflow) which alternates the running of gauge generation using the `su3_rhmd_hisq` application and hadron spectroscopy calculations using the `ks_spectrum_hisq` application.

In the OLCF-6 benchmark however, we care solely on the gauge generation part with the `su3_rhmd_hisq` application. The `ks_spectrum_hisq` benchmark is not part of the OLCF-6 benchmark.

2. Run Rules

The **OLCF-6 Benchmarks Run Rules** are to be followed. Offeror can enable and utilize existing performance optimization libraries such as QUDA on GPUs (and potentially QPhiX on AVX—capable CPUs) which are already integrated into the library (so no source code changes are needed). One note is that the QUDA library employs autotuning. Autotuning of a library is permitted. Although we are primarily interested in post-tunings timings, pre-tuning timings are useful to gauge tuning overhead.

Ported results require the use of the `milc_qcd` source code in this benchmark tarball. Ported results using QUDA require the use of `quda` in this benchmark tarball unless this version is too old to compile or function on the proposed system, in which case a more recent version of QUDA may be used. In this instance we require the offeror to provide the git-hash of the version of QUDA used and the reasons why the older version was not suitable. *Optimized* results can use any version of QUDA. For systems where ported results had to also use the newer QUDA, it is permitted for 'ported' and 'optimized' results to be identical.

3. Building MILC

This benchmark, like the NERSC-10 benchmark uses the public MILC code from the MILC Github Repository with Git-Hash `35b57df9f`. Additionally, NERSC-10 specifies two files to be patched to output additional timing information. In the version of the MILC code included here, these patches have already been applied for convenience.

Instructions for building on Frontier are provided in Section 5 of this file, or in the `BuildingOnFrontier.md` file in this benchmark tarball.

4. Benchmarks Details

4.1 Input Lattices

Similar to NERSC-10 benchmark, five concrete lattice sizes are provided as enumerated in the table below

Class (Name)	Lattice Size (lattice sites)	File Size
Tiny	48^3x64	1.9 GB (x2 files)
Small	64^3x96	6.8 GB (x2 files)
Medium	96^3x192	46 GB (x2 files)
Reference	144^3x288	231 GB (x2 files)
Target	192^3x384	730 GB (x1 file)

We use the same input lattice configurations as provided with NERSC-10 benchmark.

4.2 Validating results

In the `utils` directory of the benchmark tarball, one can find the `check-generation.awk` script. This script can be used to validate the benchmark and provide timings. In addition it will print the `OLCF-FOM` metric, which is simply the number of 80-step trajectories a single replica of the gauge generation is expected to produce per hour. This FOM uses the second trajectory of the run, extrapolated to an 80 step case.

4.2 Benchmark Requirements from Offerors

This is a *strong scaling* benchmark. As such it is not necessarily expected to scale to the full system the offeror is proposing. The benchmark is the number of trajectories per hour that the system is able to produce running either a single instance of the Gauge Generation algorithm, or via multiple instances running in parallel (i.e. made up of a number of replicas, also sometimes called streams). The Figure of Merit (FOM) of the benchmark is a tradeoff between the number of repicas and the speed with which each replica runs. Thus the FOM is increased as the time per trajectory is decreased and/or as the number of replicas increases.

To measure the capabilities of the full system the offeror is required to:

- identify a node count for a single instance of the benchmark (this may be the minimum number of nodes that the problem fits on, or the point at which strong scaling breaks down) which will optimize the final FOM
- For this node count the offeror shall measure or project the `OLCF-FOM` metric provided by the `check-generation.awk` script.
- Multiply this single replica OLCF-FOM metric by the number of replicas that are able to fit on the proposed system, to produce a final FOM.
- This final FOM is to be reported on the OLCF-6 Benchmark Spreadsheet

As an example: Assume that the offeror is proposing a system with `N` nodes, and the optimal benchmark size uses `n` nodes. The `OLCF FOM` reported by the `check-generation.awk` script. In this instance we would require the vendor to report:

- node count `n`
- the number of possible replicas: `R = floor( N / n )`
- The `Step Time` metric reported by `check-generation.awk`
- The `OLCF-FOM` metric reported by `check-generation.awk`
- The Final FOM as defined by `Final_FOM = R x OLCF-FOM`

System / Size	Ported node count: n	Ported number of replicas: R	Ported Step Time (sec)	Ported OLCF-FOM (traj/hour)	Ported Final FOM: R x OLCF-FOM (traj/hour)
MILC: Problem Size					

Optimized results may also be reported:

System/Size	Optimized node count: n	Optimized number of replicas: R	Optimized Step Time (sec)	Optimized OLCF-FOM (traj/hour)	Optimized Final FOM: R x OLCF-FOM (traj/hour)
MILC: Problem Size					

We provide below results from Frontier (system size: 9472 nodes) for the "Reference" and "Target" latice sizes. The offeror is only required to provide or project the FOM for the "Target" problem size.

Frontier, 9472 nodes	node count: n	number of replicas: R	Step Time (sec)	OLCF-FOM (traj/hour)	Final FOM: R x OLCF-FOM (traj/hour)
MILC: Reference	64	148	264.32	1.34	198.32
MILC: Target	144	65	430.22	0.829	53.88

N.B: The FOM is larger for the smaller lattice. This reflects taht that lattice is easier to generate (it is smaller so more replicas with smaller node counts are possible)

5. Bulding MILC on Frontier

This section describes how to build MICL on Frontier with QUDA using ROCm-5.5.1.

On GPU based systems the MILC code is built together with the QUDA library. In this case QUDA should be built first.

It is helpful to identify the top-level directory for the benchmark and consider paths relative to it. In our case we will use an variable called `BENCH_TOPDIR` and relative to this we will have `build` and `install` directories. E.g.

```
export BENCH_TOPDIR=<path>/OLCF-6_MILC_benchmark
cd ${BENCH_TOPDIR}
mkdir build install
export BUILD_DIR=${BENCH_TOPDIR}/build
cd ${BUILD_DIR}

cp -r ${BENCH_TOPDIR}/milc_qcd .
cp -r ${BENCH_TOPDIR}/quda .
export QUDA_DIR="${BUILD_DIR}/quda"
```

NB: The NERSC-10 benchmark requires specific QUDA and MILC versions and patches 2 files in the MILC source code. In our tarball the `quda` and `milc_qcd` branches use those repository versions, and the `milc_qcd` directory is already patched appropriately for convenience. However, one should have identical results following the NERSC-10 benchmark build instructions if one wishes.

Setting up the environment

Setting up the environment can be done in many ways. The key aspect is to ensure that the relevant compilers and libraries are available. On Frontier these libraries can be set by loading and unloading environment variables. The example below is somewhat complicated since it *does not* utilize the HPE/Cray wrappers. As such we specify compiler flags directly. A simple way to set up the environment is through a file which we will

call `env.sh`. We list our `env.sh` file below:

```
# Set up ROCm-5.5.1
# Accound for the shortcomings of the wrappers
module load cpe/23.05
module load craype-accel-amd-gfx90a
module load gcc-mixed/12.2.0
module load PrgEnv-amd
module load amd/5.5.1
module load cmake
module unload darshan-runtime
module list

### NO USER SERVICEABLE PARTS BELOW THIS LINE
export BUILDROOT=${BENCH_TOPDIR}/build
export INSTALLROOT=${BENCH_TOPDIR}/install


#Fixup LDpath because of packaging issues
export LD_LIBRARY_PATH=$CRAY_LD_LIBRARY_PATH:$LD_LIBRARY_PATH

# I need these flags because the Cray Wrappers are broken
# And I want to ensure I use the right versions of all the libraries
#
export MPICH_ROOT=/opt/cray/pe/mpich/8.1.26
export GTL_ROOT=/opt/cray/pe/mpich/8.1.26/gtl/lib
export MPICH_DIR=${MPICH_ROOT}/ofi/amd/5.0

## These must be set before running
export TARGET_GPU=gfx90a

# Set CFLAGS because I am not using a wrapper
export MPI_CFLAGS="${CRAY_XPMEM_INCLUDE_OPTS} -I${MPICH_DIR}/include "
export MPI_LDFLAGS=" ${CRAY_XPMEM_POST_LINK_OPTS} -lxpmem  -Wl,-rpath=${MPICH_DIR}/lib \
-L${MPICH_DIR}/lib -lmpi -Wl,-rpath=${GTL_ROOT} -L${GTL_ROOT} -lmpi_gtl_hsa \
-L${ROCM_PATH}/llvm/lib -Wl,-rpath=${ROCM_PATH}/llvm/lib"

# Set pore paths and LDPATHs
export PATH=${ROCM_PATH}/llvm/bin:$PATH
export LD_LIBRARY_PATH=${ROCM_PATH}/llvm/lib:${ROCM_PATH}/lib:${LD_LIBRARY_PATH}
export LD_LIBRARY_PATH=${INSTALLROOT}/quda/lib:${LD_LIBRARY_PATH}

export MPICH_GPU_SUPPORT_ENABLED=1
```

To load the settings in the `env.sh` we need to `source` the file

```
source ./env.sh
```

NOTE: It is important to source the `env.sh` file when building MILC later as it uses makefiles and various environment variables need to be explicitly exported to be picked up by make.

Building QUDA

To build QUDA for Frontier please run the following script in the `${BUILD_DIR}`:

```
source ${BENCH_TOPDIR}/env.sh

if [ -d ./build_quda ];
then
  rm -rf ./build_quda
fi

mkdir  ./build_quda
cd ./build_quda

export QUDA_GPU_ARCH=${TARGET_GPU}

cmake ${BUILDROOT}/quda \
       -G "Unix Makefiles" \
       -DQUDA_TARGET_TYPE="HIP" \
       -DQUDA_GPU_ARCH=${TARGET_GPU} \
       -DROCM_PATH=${ROCM_PATH} \
       -DQUDA_DIRAC_DEFAULT_OFF=ON \
       -DQUDA_DIRAC_STAGGERED=ON \
       -DQUDA_FORCE_GAUGE=ON \
       -DQUDA_FORCE_HISQ=ON \
       -DQUDA_INTERFACE_MILC=ON \
       -DQUDA_MPI=ON \
       -DQUDA_DOWNLOAD_USQCD=ON \
       -DQUDA_DOWNLOAD_EIGEN=ON \
       -DCMAKE_INSTALL_PREFIX=${INSTALLROOT}/quda \
       -DCMAKE_BUILD_TYPE="Release" \
       -DCMAKE_CXX_COMPILER="amdclang++" \
       -DCMAKE_HIP_COMPILER="amdclang++" \
       -DCMAKE_C_COMPILER="amdclang" \
       -DBUILD_SHARED_LIBS=ON \
       -DQUDA_BUILD_SHAREDLIB=ON \
       -DQUDA_BUILD_ALL_TESTS=ON \
       -DQUDA_CTEST_DISABLE_BENCHMARKS=ON \
       -DCMAKE_C_STANDARD=99 \
       -DCMAKE_CXX_FLAGS="${MPI_CFLAGS} --offload-arch=gfx90a" \
       -DCMAKE_C_FLAGS="${MPI_CFLAGS} --offload-arch=gfx90a" \
       -DCMAKE_SHARED_LINKER_FLAGS="${MPI_LDFLAGS} --offload-arch=gfx90a" \
       -DCMAKE_EXE_LINKER_FLAGS="${MPI_LDFLAGS} --offload-arch=gfx90a"


cmake --build . -j 32  -v
cmake --install .
```

Notes:

1. Please the requirement to explicitly specify the `CMAKE_CXX_FLAGS`, `CMAKE_C_FLAGS` and the `CMAKE_SHARED_LINKER_FLAGS` and the `CMAKE_EXE_LINKER_FLAGS` both to provide the MPI related flags defined in the `env.sh` file as well as adding explicitly the `--offload-arch` option. This is primarily because we are not using the Cray/HPE wrappers.
2. This method of building QUDA will try to automatically Eigen-3.4.0. If Internet access is not available during building Eigen can be downloaded manually on a different host from : The Eigen Web Page (https://gitlab.com/libeigen/eigen/-/archive/3.4.0/eigen-3.4.0.tar.gz). One then needs to install it (unzip) it, and tell the QUDA build-system to use the downloaded version with additional CMake command line arguments:

```
-DQUDA_DOWNLOAD_EIGEN=OFF \
-DEigen3_DIR=<Eigen install>/cmake \
```

or

```
 -DQUDA_DOWNLOAD_EIGEN=OFF \
 -DEIGEN_INCLUDE_DIRS=< .. >
```

The script should install QUDA in the `${BENCH_TOPDIR}/install/quda` directory. For other ways to build QUDA (e.g. for CUDA) please refer to the instructions in the NERSC-10 Lattice QCD Workflow benchmark (https://gitlab.com/NERSC/N10-benchmarks/lattice-qcd-workflow#12-building-quda-optional-for-gpus-only-) or to the QUDA Wiki (https://github.com/lattice/quda/wiki).

Building the MILC Code.

Building the MILC code requires manual setting of options in the `Makefiles` provided with MILC. In this example below we follow the approach in the NERSC-10 Lattice QCD Workflow benchmark (https://gitlab.com/NERSC/N10-benchmarks/lattice-qcd-workflow#13-building-milc).

In the `${BUILD_DIR}/milc_qcd` directory one needs to set several options. The crucial ones relate to timing, and precision. In this `Makefile` these can all be set up up-front at the top of the `Makefile`. Subsequent lower sections of the Makefiles make decisions but will not override the settings that come before them (they are set using the =? Makefile assignment below ours). One exception to this is in a dependent makefile called `libraries/Make_vanilla` which concretely sets variables and may override our choice of C compiler. Thus we will need to edit both the topelevel `Makefile` and the `libraries/Make_vanilla` file.

The crucial timing and precision settings are as follows:

```
CTIME = -DCGTIME -DFFTIME -DFLTIME -DGFTIME -DIOTIME
PRECISION=1
WANT_MIXED_PRECISION_GPU=0
```

however, since this version of the MILC code does not yet know much about ROCM/Clang/etc. we have to set a whole bunch of other options. Paste the following options into the `Makefile` in the toplevel `milc_qcd` directory, at about line 10, below the line `MAKEFILE = Makefile`:

```
MY_CC = amdclang
MY_CXX = amdclang++
MPP = true
OMP = true
PRECISION = 1
WANT_MIXED_PRECISION_GPU = 0
CGEOM = -DFIX_NODE_GEOM -DFIX_IONODE_GEOM
CTIME = -DCGTIME -DFFTIME -DFLTIME -DGFTIME -DIOTIME
KSCGMULTI = -DKS_MULTICG=HYBRID -DMULTISOURCE
CUDA_HOME =
QUDA_HOME = ${INSTALLROOT}/quda
WANTQUDA = true
WANT_FN_CG_GPU = true
WANT_GF_GPU = true
WANT_GA_GPU = true
WANT_FL_GPU = true
WANT_FF_GPU = true
OCFLAGS = -std=c99 ${MPI_CFLAGS}
OCFLAGS += -Wall -Wno-unused-variable -Wno-unused-but-set-variable -Wno-unknown-pragmas -Wno-unused-function
OCFLAGS += -fopenmp
OCXXFLAGS = -std=c++17 ${MPI_CFLAGS}
LDFLAGS += -fopenmp  -L${ROCM_PATH}/lib -lhipfft -lhipblas -lrocblas -lamdhip64 ${MPI_LDFLAGS}
COMPILER = clang
OFFLOAD = rocm
```

The next step is to change the `CC` compiler in the `libraries/Make_vanilla` Makefile. On line 39, add the `else` clause:

```
else ifeq ($(strip ${COMPILER}),clang)
  CC = amdclang
```

so that the file looks like:

```
#------------------------------------------------------------------------
#  User choices - edit to suit
#------------------------------------------------------------------------
# 1. Compiler

ifeq ($(strip ${COMPILER}),intel)
  CC = icx
else ifeq ($(strip ${COMPILER}),gnu)
  CC = gcc
else ifeq ($(strip ${COMPILER}),cray-intel)
  CC = icc
else ifeq ($(strip ${COMPILER}),ibm)
  CC = xlc_r
else ifeq ($(strip ${COMPILER}),clang)
  CC = amdclang
endif

# Override
# CC = # ( cc89 gcc xlc gcc pgcc cl g++ )
```

Once these changes are made, ensure that the `env.sh` file has been sourced, and then return to the toplevel `milc_qcd` directory.

Building `su3_rhmd_hisq`

At this point (toplevel `milc_qcd` directory) the process of building `su3_rhmd_hisq` is as follows:

```
cd ks_imp_rhmc
cp ../Makefile .
make clean
make su3_rhmd_hisq
```

The process should finish leaving the `su3_rhmd_hisq` in the current directory.

6. Running the Benchmarks

6.1 Generic run instructions

To run the `su3_rhmd_hisq` code one needs an input parameter file and an input lattice file. The input parameter file describes the number of MPI processes needed for the code to run, and the location of the input lattice file. Generically one just needs to run the command:

```
su3_rhmd_hisq inputFile
```

The input problem lattice needs to be in a subdirectory called `lattices` of the run directory.

Of course to run this in an MPI environment one needs to invoke an MPI runner (e.g. `srun` or `mpirun`) with appropriate parameters and process binging. This benchmark contains sample run scripts for both the reference and target problem sizes on Frontier. These can be found respectively in:

- `benchmarks/reference/generation/frontier/run_frontier-gpu-512.sh` for the 'reference' problem (64 nodes, 512 GCDs)
- `benchmarks/target/generation/frontier/run_frontier-gpu-1152.sh` for the target problem (144 nodes, 1152 GCDs)

6.2 Changing the number of MPI tasks (`node_geometry`)

The MILC code assumes that the MPI processes make up a 4-dimensional MPI-Process grid. The dimensions of these are specfied in the input parameter file. For example in the case of the `target` problem this file can be found in `benchmarks/target/generation/input_192384` in this tarball. The first few lines of the input file look like:

```
prompt 0
nx 192
ny 192
nz 192
nt 384
node_geometry 1 4 12 24
ionode_geometry 1 4 12 24
```

The `nx`, `ny`, `nz`, `nt` parameters denote the global lattice size. and the `node_geometry` denotes the sizes of the MPI process grid. In this case it is a 1x4x12x24 grid which uses 1152 MPI processes.

One can change the grid-size in the `node_geometry` e.g. a 2304 MPI process could be encoded as:

```
node_geometry 1 8 12 24
```

In this case the local volume per node would be:

```
local volume = (192/1) x (192/8) x (192/12) x (384/24) = 192 x 24 x 16 x 16 sites
```

Restrictions are:

- The *local* volume has to be even in all dimensions
- The minimum *local* volume is 4x2x2x2

6.3 Changing the I/O node_geometry (ionode_geometry)

The number of processes taking place in the I/O can affect the run-time of the benchmarks, although this is not something we care about in the OLCF benchmark. Still it may assist in overall benchmark execution. Each node in the node_geometry can be labeled an I/O process. However we can have only a subset of the nodes take part in the I/O. The ionode_geometry option in the input parameter file controls this subset. If the ionode_geometry is the same as the node_geometry every MPI rank is an I/O processor. Otherwise the I/O node geometry divides the node geometry into sub-hypercubes. E.g. in the example:

```
node_geometry node_geometry 1 4 12  24
ionode_geometry 1 2 6 12
```

The ionode_geometry divides the node_geometry in to 1x2x6x12 hypercubes of size 1x2x2x2 processes. The root process of each hypercube will perform I/O collecting or distributing the data to each of the 8 processes in its hypercube.

6.4 Changing the location of the lattices

If it proves inconvnient to copy or symlink the lattices into a directory called lattices in the run-directory, an alternate location can be specified in the input file using the reload parallel and save_parallel commands. In the input_192384 parameter file these lines are at the end of the file:

```
reload_parallel lattices/l192384f211b728m000415m01129m1329a.23
save_parallel           l192384f211b728m000415m01129m1329a.23x
```

these paths are relative to the current run directory, but can also be replaced with appropriate full paths.