Aneesh Pandoh

Configuration

Utilized the Linux Lab machines. The lscpu command reveals that the machine has an 11th-gen core i5 processor with a base clock of 2.70 GHz and 6 cores with 2 threads each meaning **12 total threads**. Based on the Intel product page, **the peak memory bandwidth of the processor is 50 GB/s.**

Architecture: x86_64
CPU(s): 12
Model name: 11th Gen Intel(R) Core(TM) i5-11500 @ 2.70GHz
Thread(s) per core: 2
Core(s) per socket: 6

[HOW TO RUN]

Need CMake to build the project configured for the icc compiler on Linux. **To compile the program go into the build directory and then run make which creates the "Homework3" executable, make sure you have set source /s/intelcompilers/bin/iccvars.sh intel64.** The CMakeLists.txt includes the OpenMP & MKL option, src files, and compiler optimization (as "-03" is not enabled by default on g++) seen below:

```
cmake_minimum_required(VERSION 3.5)
#make sure you set the source env
set(CMAKE_C_COMPILER "icc")
set(CMAKE_CXX_COMPILER "icpc")
project(Homework3 VERSION 0.1.0 LANGUAGES C CXX)
include(CTest)
enable_testing()

set(SOURCES
    ${CMAKE_SOURCE_DIR}/ConjugateGradients.cpp
    ${CMAKE_SOURCE_DIR}/ConjugateGradients.h
    ${CMAKE_SOURCE_DIR}/CSRMatrix.h
    ${CMAKE_SOURCE_DIR}/CSRMatrixHelper.h
    ${CMAKE_SOURCE_DIR}/Laplacian.cpp
    ${CMAKE_SOURCE_DIR}/Laplacian.h
    ${CMAKE_SOURCE_DIR}/main.cpp
    ${CMAKE_SOURCE_DIR}/MatVecMultiply.cpp
    ${CMAKE_SOURCE_DIR}/Parameters.h
    ${CMAKE_SOURCE_DIR}/PointwiseOps.cpp
    ${CMAKE_SOURCE_DIR}/PointwiseOps.h
    ${CMAKE_SOURCE_DIR}/Reductions.cpp
    ${CMAKE_SOURCE_DIR}/Reductions.h
    ${CMAKE_SOURCE_DIR}/Timer.h
```

```
    ${CMAKE_SOURCE_DIR}/Utilities.cpp
    ${CMAKE_SOURCE_DIR}/Utilities.h
)
add_executable(Homework3 ${SOURCES})
set(CMAKE_CXX_FLAGS "-O3 -mavx512f -mkl")

find_package(OpenMP)
if(OpenMP_CXX_FOUND)
  target_link_libraries(Homework3 PUBLIC OpenMP::OpenMP_CXX
  )
endif()

set(CPACK_PROJECT_NAME ${PROJECT_NAME})
set(CPACK_PROJECT_VERSION ${PROJECT_VERSION})

include(CPack)
```

Replaced Kernels

Copy Kernel:

```
void Copy(const float (&x)[XDIM][YDIM][ZDIM], float (&y)[XDIM][YDIM][ZDIM])
{
#ifdef DO_NOT_USE_MKL
  #pragma omp parallel for
    for (int i = 1; i < XDIM-1; i++)
    for (int j = 1; j < YDIM-1; j++)
    for (int k = 1; k < ZDIM-1; k++)
      y[i][j][k] = x[i][j][k];
#else
  int totalElements = (XDIM) * (YDIM) * (ZDIM);
  cblas_scopy(totalElements, &x[0][0][0], 1, &y[0][0][0], 1);
#endif
}
```

Norm Kernel:

```
float Norm(const float (&x)[XDIM][YDIM][ZDIM])
{
  float result = 0.;
#ifdef DO_NOT_USE_MKL
#pragma omp parallel for reduction(max:result)
  for (int i = 0; i < XDIM; i++)
  for (int j = 0; j < YDIM; j++)
```

```
  for (int k = 0; k < ZDIM; k++)
    result = std::max(result, std::abs(x[i][j][k]));
#else
  int totalElements = (XDIM) * (YDIM) * (ZDIM);
  int maxIndex = cblas_isamax(totalElements, &x[0][0][0], 1);
  result = std::abs(x[(maxIndex / ((YDIM)*(ZDIM)))]
                    [((maxIndex % ((YDIM)*(ZDIM))) / (ZDIM))]
                    [(maxIndex % (ZDIM))]);
#endif
  return result;
}
```

Inner Product Kernel:

```
float InnerProduct(const float (&x)[XDIM][YDIM][ZDIM], const float (&y)[XDIM][YDIM][ZDIM])
{
  double result = 0.;
#ifdef DO_NOT_USE_MKL
#pragma omp parallel for reduction(+:result)
  for (int i = 1; i < XDIM-1; i++)
  for (int j = 1; j < YDIM-1; j++)
  for (int k = 1; k < ZDIM-1; k++)
    result += (double) x[i][j][k] * (double) y[i][j][k];
#else
  int totalElements = (XDIM) * (YDIM) * (ZDIM);
  result = cblas_sdot(totalElements, &x[0][0][0], 1, &y[0][0][0], 1);
#endif
  return (float) result;
}
```

MKL

Conjugate Gradients terminated after 256 iterations; residual norm (nu) = 0.000974735
[Total Norm Time: 796.141ms]
[Total Copy Time: 1695.49ms]
[Total InnerProduct Time: 3128.71ms]

NOT MKL

Conjugate Gradients terminated after 256 iterations; residual norm (nu) = 0.00097589
[Total Norm Time: 853.909ms]
[Total Copy Time: 2770.1ms]
[Total InnerProduct Time: 3534.79ms]

Results:

The implementation is configured properly with pretty much equivalent residual norm values and iterations. We can clearly see a significant speedup in the copy kernel at around 39%, and still a decent ~10% speedup in the norm and inner product kernel. When implementing I changed the functions to go through 0 to DIM instead of 1 to DIM - 1 as explained on piazza.