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 G++ compiler on Linux. **To compile the program go into the build directory and then run make which creates the "ConjGrad" executable.** The CMakeLists.txt includes the OpenMP option, src files, and compiler optimization (as "-03" is not enabled by default on g++) seen below:

```
project(ConjGrad VERSION 0.1.0 LANGUAGES C CXX)
include(CTest)
enable_testing()
set(SOURCES ${CMAKE_SOURCE_DIR}/main.cpp
      ${CMAKE_SOURCE_DIR}/ConjugateGradients.cpp
      ${CMAKE_SOURCE_DIR}/Timer.h
      ${CMAKE_SOURCE_DIR}/ConjugateGradients.h
      ${CMAKE_SOURCE_DIR}/Laplacian.cpp
      ${CMAKE_SOURCE_DIR}/Laplacian.h
      ${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}/Utilities.cpp
      ${CMAKE_SOURCE_DIR}/Utilities.h)
add_executable(ConjGrad ${SOURCES})
find_package(OpenMP)
if(OpenMP_CXX_FOUND)
  target_link_libraries(ConjGrad PUBLIC OpenMP::OpenMP_CXX)
endif()
set(CMAKE_CXX_FLAGS "-O3 -mavx512f")
set(CPACK_PROJECT_NAME ${PROJECT_NAME})
set(CPACK_PROJECT_VERSION ${PROJECT_VERSION})
include(CPack)
```

Testing/Results Task #1

1 Thread W/O Output:

[LaplacianL2 time: 18.3866ms]
[SaxpyL2 time: 17.2262ms]
[NormL2 time: 15.1579ms]
[CopyL4 time: 14.3663ms]
[InnerProductL4 time: 15.8381ms]
[LaplacianL6 Time: 2846.63ms]
[InnerProductL6 Time: 4090.17ms]
[SaxpyL8 Time: 2610.37ms]
[NormL8 Time: 4003.81ms]
[SaxpyL9 Time: 10.0836ms]
[CopyL13 Time: 2486.54ms]
[InnerProductL13 Time: 4079.75ms]
[SaxpyL16A Time: 2589.79ms]
[SaxpyL16B Time: 2541.42ms]
[Total Time: 25345.2ms]

12 Threads: W/O Output

[LaplacianL2 time: 12.8674ms]
[SaxpyL2 time: 18.9731ms]
[NormL2 time: 3.13036ms]
[CopyL4 time: 7.12993ms]
[InnerProductL4 time: 6.84855ms]
[LaplacianL6 Time: 4316.45ms]
[InnerProductL6 Time: 1734.31ms]
[SaxpyL8 Time: 2538.67ms]
[NormL8 Time: 825.029ms]
[SaxpyL9 Time: 10.0642ms]
[CopyL13 Time: 2783.87ms]
[InnerProductL13 Time: 1723.65ms]
[SaxpyL16A Time: 2558.88ms]
[SaxpyL16B Time: 2541.36ms]
[Total Time: 19086.2ms]

From the data, we can see about a 30% speed up by parallelizing with 12 cores. From examining the individual kernel operations, we can see that it varies but most of the time there is around a 100% speedup on the operations, besides Saxpy, which did not seem to get sped up with the additional cores. This makes sense as we did not include OpenMP parallelization for Saxpy. Here is a snippet of the code showing how I collected the timing information:

```
// Algorithm : Line 8
timerSaxpyL8.Restart(); Saxpy(z, r, r, -alpha); timerSaxpyL8.Pause();
timerNormL8.Restart(); nu=Norm(r); timerNormL8.Pause();
```

The timers were reset and initialized before the call to ConjugateGradients in main and the result was printed out after the call to ConjugateGradients. These are the same timers from LaplaceSolver_0_3.

Combined Kernels Task #2:

```
float SaxpyNorm(const float (&x)[XDIM][YDIM][ZDIM], const float
(&y)[XDIM][YDIM][ZDIM],
   float (&z)[XDIM][YDIM][ZDIM],
   const float scale)
{
   float result = 0.;
#pragma omp parallel for reduction(max: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++) {
       z[i][j][k] = x[i][j][k] * scale + y[i][j][k];
       result = std::max(result, std::abs(z[i][j][k]));
   }
   return result;
}
double InnerProductCopy(const float (&x)[XDIM][YDIM][ZDIM], float
(&y)[XDIM][YDIM][ZDIM])
{
double result = 0.;
#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++) {
       y[i][j][k] = x[i][j][k];
       result += (double) x[i][j][k] * (double) y[i][j][k];
   }
   return result;
}
```

I combined the InnerProduct and Copy kernels as well as the Saxpy and Norm kernels with the code above. The reason for doing this was outlined very well on Piazza as “we can avoid streaming memory for the same vector repeatedly from memory to cache.” We can see that this actually results in a 2-second speedup, mostly from the from the InnerProductCopy function now taking half the time of the InnerProduct and Copy functions previously. We can also see the same cost cut in SaxpyNorm, but it is not as pronounced due to it being outside of the loop, and only running once. In both these cases, we were also able to perform the functions simultaneously, with OpenMP parallelization as well.

[LaplacianL2 time: 12.6538ms]
[NEW SaxpyNorm Time: 10.8233ms]

[CopyL4 time: 9.97012ms]
[InnerProductL4 time: 6.83228ms]
[LaplacianL6 Time: 4279.32ms]
[InnerProductL6 Time: 1724.91ms]
[SaxpyL8 Time: 2518.18ms]
[NormL8 Time: 823.469ms]
[SaxpyL9 Time: 9.92884ms]
[NEW InnerProductCopy Time: 2712.91ms]
[SaxpyL16A Time: 2543.83ms]
[SaxpyL16B Time: 2492.57ms]
[Total Time: 17150.7ms]