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 op-mode(s):      32-bit, 64-bit
CPU(s):                12
Vendor ID:             GenuineIntel
  Model name:          11th Gen Intel(R) Core(TM) i5-11500 @ 2.70GHz
    CPU family:        6
    Model:             167
    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 "a1" executable.** The CMakeLists.txt includes the OpenMP option, src files, and compiler optimization (as "-02" is not enabled by default on g++) seen below:

```
cmake_minimum_required(VERSION 3.9.0)
project(a1 VERSION 0.1.0 LANGUAGES C CXX)
include(CTest)
enable_testing()

set(SOURCES
       ${CMAKE_SOURCE_DIR}/main.cpp
       ${CMAKE_SOURCE_DIR}/Laplacian.cpp
       ${CMAKE_SOURCE_DIR}/Timer.h
       ${CMAKE_SOURCE_DIR}/Laplacian.h)
add_executable(a1 ${SOURCES})

find_package(OpenMP)
if(OpenMP_CXX_FOUND)
   target_link_libraries(a1 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

To calculate the effective bandwidth we need to determine the total memory of the 2 arrays we are accessing, u and lu.
Total: 512 float x 512 float x 512 float = 134217728 float x 4 bytes = 0.5 GB * 2 arrays = **1 GB**

No OpenMP:

Running test iteration 1 [Elapsed time : 274.66ms]
Running test iteration 2 [Elapsed time : 130.736ms]
Running test iteration 3 [Elapsed time : 130.469ms]
Running test iteration 4 [Elapsed time : 130.49ms]
Running test iteration 5 [Elapsed time : 130.208ms]
Running test iteration 6 [Elapsed time : 130.254ms]
Running test iteration 7 [Elapsed time : 131.931ms]
Running test iteration 8 [Elapsed time : 130.41ms]
Running test iteration 9 [Elapsed time : 130.187ms]
Running test iteration 10 [Elapsed time : 130.223ms]

1 thread w/ OpenMP:

Running test iteration 1 [Elapsed time : 269.916ms]
Running test iteration 2 [Elapsed time : 130.138ms]
Running test iteration 3 [Elapsed time : 130.194ms]
Running test iteration 4 [Elapsed time : 130.28ms]
Running test iteration 5 [Elapsed time : 130.22ms]
Running test iteration 6 [Elapsed time : 130.544ms]
Running test iteration 7 [Elapsed time : 130.339ms]
Running test iteration 8 [Elapsed time : 130.094ms]
Running test iteration 9 [Elapsed time : 130.192ms]
Running test iteration 10 [Elapsed time : 133.914ms]

12 thread [max] (in order):

Running test iteration 1 [Elapsed time : 72.4163ms]
Running test iteration 2 [Elapsed time : 72.0183ms]
Running test iteration 3 [Elapsed time : 57.469ms]
Running test iteration 4 [Elapsed time : 63.0387ms]
Running test iteration 5 [Elapsed time : 61.9608ms]
Running test iteration 6 [Elapsed time : 59.2857ms]
Running test iteration 7 [Elapsed time : 66.0255ms]
Running test iteration 8 [Elapsed time : 65.3157ms]
Running test iteration 9 [Elapsed time : 60.398ms]
Running test iteration 10 [Elapsed time : 64.2383ms]

Max Threads:
Average time: 64.21663 ms
1 GB / 0.06421663
Total GB/s: 15.572 GB/s about 31% of peak bandwidth

We can see that with no OpenMP enabled time for completion was about the same with 1 thread with OpenMP as OpenMP is unable to parallelize the code with just 1 thread to work with. We can see a significant speed-up when we utilize the full 12 threads of the CPU as the average time reduces down to 64 ms, however when we calculate the practical memory bandwidth we are left with disappointing results as we divide the total memory by the seconds spent executing to get 15.625 GB/s. This is only 31% of the most optimal bandwidth of the CPU, this could be due to the CPU being utilized by other users as this is a shared lab computer where other users are signed on and running programs.

12 thread [max] (opposite order):
Running test iteration 1 [Elapsed time : 1708.4ms]
Running test iteration 2 [Elapsed time : 1495.48ms]
Running test iteration 3 [Elapsed time : 1499.63ms]
Running test iteration 4 [Elapsed time : 1435.93ms]
Running test iteration 5 [Elapsed time : 1466.81ms]
Running test iteration 6 [Elapsed time : 1469.33ms]
Running test iteration 7 [Elapsed time : 1487.55ms]
Running test iteration 8 [Elapsed time : 1444.95ms]
Running test iteration 9 [Elapsed time : 1478.31ms]
Running test iteration 10 [Elapsed time : 1458.96ms]

12 thread [max] (switched Y and Z):
Running test iteration 1 [Elapsed time : 152.165ms]
Running test iteration 2 [Elapsed time : 101.447ms]
Running test iteration 3 [Elapsed time : 125.632ms]
Running test iteration 4 [Elapsed time : 100.582ms]
Running test iteration 5 [Elapsed time : 108.108ms]
Running test iteration 6 [Elapsed time : 112.823ms]
Running test iteration 7 [Elapsed time : 110.698ms]
Running test iteration 8 [Elapsed time : 112.061ms]
Running test iteration 9 [Elapsed time : 105.254ms]
Running test iteration 10 [Elapsed time : 114.245ms]

The 3-dimensional array is stored as a 1-dimensional vector in memory, so consecutive values on the X-axis are right next to each, then a new Y-axis row starts, again adding consecutive X-axis values, then finally a new Z-axis value is introduced with the same process of adding the respective Y axis and X axis to memory.

This means how we access the values matters as we want to optimize for spacial locality as we want as many cache hits as possible, meaning we want consecutive values as that is what is being stored in the cache lines. This is with the data as when we access in X, Y, Z order we see the fastest times because we don't have to reach all the way into memory for most values. This is only slightly slowed down when we switch Y and Z as the cache line is probably large enough to store more than a whole X-axis worth of data, meaning that even though we skip around the memory more, we still hit the cache more often than not. We can see drastic change when we go to the opposite Z, Y, X order as we move so far around memory so often that it causes a bunch of cache misses, thus slowing down the time significantly.

Approximate Time Taken: 3 - 3.5 hrs (Took time understanding how to use CMake, use OpenMP with Cmake, and make sure the compiler was optimized, etc.)