Homework 4: Single-node Optimizations

Due: 11:55 P.M. on Friday, March 17, 2017

In this assignment, you will practice single-node optimization. As usual, you may work in teams of two. Be sure to indicate with whom you worked by creating a `README` file as part of your submission.

Part 0: Getting the scaffolding code

The tarball containing the code for this homework/lab is available at this URL: `https://github.com/EECS117/hw4.git`. To get it, use the same git-clone procedure from previous labs.

If it worked, you'll have a new directory called **hw4**.

Part 1 (in class): Saavedra-Barrera benchmark

Last week in class, we looked at the Saavedra-Barrera (SB) benchmark. It works by first creating an array of length n, and then stepping through the array with stride s, reading an element at each step. It repeats this process many times and reports the average time to read 1 element.

Lets compile and run this benchmark, `sb.cc`:

```
# Compile
g++ -O3 -o sb sb.cc

# Request an interactive node
qrsh -q eecs117

    ...

# Once logged into the interactive node, run the program:
./sb 2048
```

The benchmark starts with an array of size $n = 1024$ and runs the Saavedra-Barrera scheme of varying strides up to $1024/2 = 512$. It then doubles the array size and repeats this procedure, up to the maximum array size that you specify. (In this example, specifying 2048 on the command-line will cause the benchmark to try $n = 1024$ and $n = 2048$.)

Q1. Run the benchmark to collect data for sizes up to 64 MB, i.e., array sizes up to 16777216 (since each element of the array is 4 bytes). Submit both the raw data and a plot of the data. You can use any program (e.g., gnuplot, Excel, MATLAB) to draw the plot. Below, we explain how to use `gnuplot` for this purpose.

To collect the data, please use the following procedure. First, log into an interactive node. Run the `sb` program twice as follows:

```
# First, be sure you are on an interactive node.
# Then, start with a "warm-up" run on a small array:

./sb 512

# Now do the real run on 64 MB (i.e., array of length 2^24):

./sb 16777216 | tee run.txt
```

The first command is a short warm-up run. Doing this first will help you get stable timings for the real run, which is the second command. The second command collects data by sending (piping) anything that the program writes to standard output to a file called `run.txt`. (Standard output refers to the `cout` file variable in `sb.cc`.)

The `run.txt` file will have output that looks something like:

```
1024 1 1.30666e-09
1024 2 1.30656e-09
1024 4 1.3064e-09
1024 8 1.30646e-09
1024 16 1.30645e-09
1024 32 1.30657e-09
1024 64 1.30648e-09
1024 128 1.30642e-09
1024 256 1.3063e-09
1024 512 1.30668e-09


2048 1 1.30649e-09
2048 2 1.30643e-09
2048 4 1.30658e-09
2048 8 1.30639e-09
2048 16 1.30654e-09
2048 32 1.30668e-09
2048 64 1.30643e-09
2048 128 1.30661e-09
2048 256 1.30647e-09
2048 512 1.30639e-09
2048 1024 1.30663e-09


4096 1 1.30648e-09
4096 2 1.3066e-09
4096 4 1.30652e-09
4096 8 1.30644e-09
  ...
```

The values mean the following:

- Column 1 is the length of the input array, in 4 byte words. That is, a value of 1024 means an array with 1024 elements x 4 bytes per element = 4096 bytes.

- Column 2 is the stride, again in 4 byte words. A value of 1 means every consecutive word, a value of 2 means every other word.
- Column 3 is the average time (seconds) per read.

This output is designed to be easy to plot using the `gnuplot` utility, though you are welcome to use any other plotting utility (such as Excel, MATLAB, or R) and modify the program output accordingly. If you wish to use `gnuplot`, the following command will run a `gnuplot` script we have provided and generate an output file called `sb.png`.

```
gnuplot < sb.gp
```

To view the plot, run the following command:

```
display sb.png
```

Q2. From your plot, try to deduce the following:

1. How many levels of cache do there appear to be?
2. For each level of cache, what is its capacity (size)? Line size?

Again, please submit your answers, raw data, and plot.

Part 2 (after class): Cache blocked vectorized matrix multiply

In class, we covered memory hierarchies in which we learned that CPUs have fast memory, or caches, that are used to augment the slower main memory. Caches are much smaller in size but faster to access in comparison to main memory and are used to temporarily store data that may be re-used during the course of a program's execution.

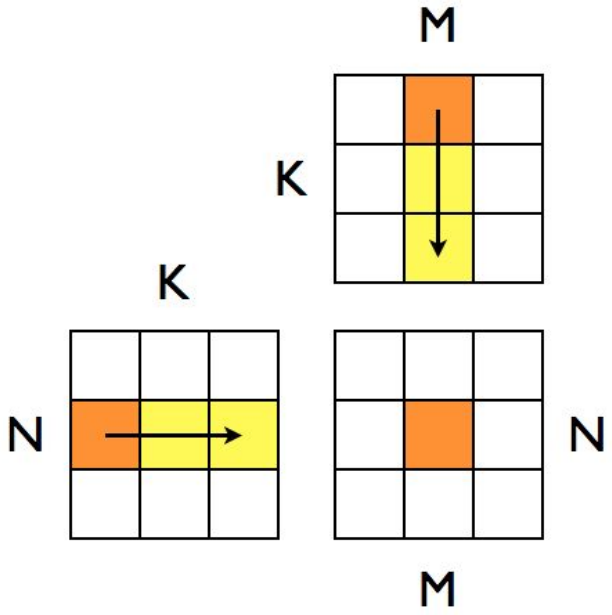
For the after-class part of this homework, you will implement a matrix-matrix multiply program that will try to take advantage of the cache and SIMD registers available on the HPC cluster to improve performance.

A naive matrix-matrix multiply implementation is provided as a reference. To compile and run this code, do the following:

```
# Compile
g++ -O3 -o mm mm.cc

# Request an interactive node
qrsh -q eecs117

...

# Once logged into the interactive node, run the program:
# ./mm <N> <K> <M>
./mm 1024 1024 1024
```

Q3. Run the program several times and record the smallest execution time. And then, compute the performance in GFLOP/s for the naive kernel using this time.

Q4. Implement a cache-blocked version of matrix-matrix multiply. This is essentially a blocked/tiled implementation where you compute the matrix in smaller sub-block increments. This is shown in the figure below. You can also refer to lecture slides for more information and the psuedo code. In your implementation, make sure that the size of the sub-block can be varied.



Q5. Try varying the block size and measure/report the performance of your code in terms of GFLOP/s. Use this information to determine the approximate size of the cache in the system. Justify your reasoning.

Q6. Implement a SIMD vectorized version of your cache blocked matrix-matrix multiply. Report the performance of your code in terms of GFLOP/s.

Turn in the tarball of your code and results by uploading it to eee.uci.edu dropbox.

Good luck, and have fun!