Fast Matrix Multiply

Asignment 7

Author:

Patricio Beltran

patriciobeltran@utexas.edu

PMB784

Useful information:

The program comes with a couple of features that should be helpful for proving correctness of the algorithm and showing the output after it ran.

It has been tested with `gcc` and `icc` compilers. To compile just:

`ICC Compiler`

```
$ icc -O3 -fp-model precise matrixmultiply.c -o mmm
```

`GCC Compiler`

```
$ gcc matrixmultiply.c -o mmm -O3
```

After compiled it can be run by doing `./mmm`. The program expects a size as an input and it will generate two random square matrices and multiply them. If no input is given to the program it will display how to use it.

```
Usage: ./mmm <matrix size> [flags]

Flags:
  -d:   Print debugging info. This includes both A and B matrices as well as the
        result of their multiplication (C matrix).
  -t:   Run Fast Matrix Multiply (save resulting matrix in mine.txt) and the
        simple ijk algorithm (save resulting matrix in correct.txt). Useful
        for proving correctness of Fast Matrix Multiply.
```

If program is ran without any flags (only with size) it will only print the size, the time spent in matrix multiply and the GFLOPS achieved.

The `testCorrectness.bash` script is intended to help prove correct the algorithm. I runs the program for a range of inputs with the `-t` flag (they are initialy set to 1-400, but can be modified). If the script finds that for one input the result is not correct it will output a message to the screen. This script is not intended to be fast, it intends to show that the `fast matrix multiply` is correct by comparing it to the simple `ijk` algorithm.

What's included

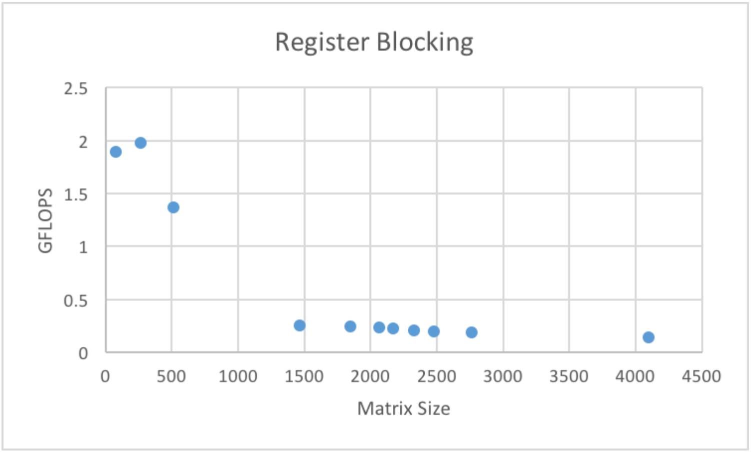
In the project directory you will find:

```
 - matrixmultiply.c         Fastest Matrix Multiply code, includes cache
blocking,
                           register blocking, data copying and vectorization.
 - testCorrectness.bash    Script that matches the code from the fast matrix
                           multiply with a simple ijk matrix multiply for a
number
                           of different inputs.
 - README.pdf              Information on the project.
```

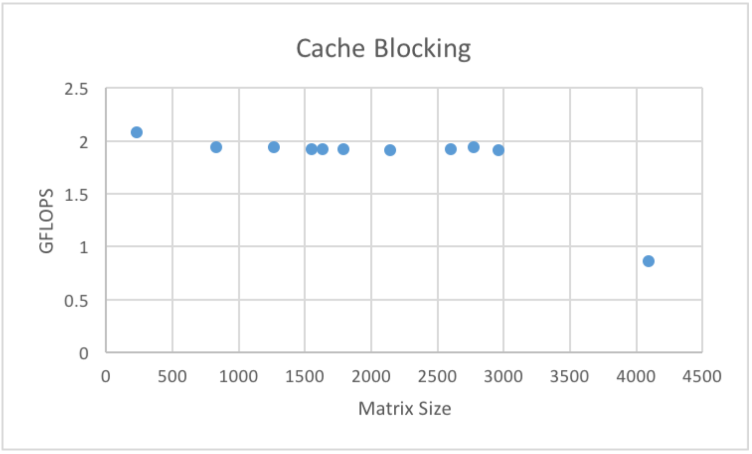
Plots and analysis

Below are shown some plots that show the relationship between the matrix sizes and the GFLOPS. For each optimization the code was run 11 times with random values for the matrix sizes in the range of `1 - 3000` for register and cache blocking (because of the runtime) and `1 - 5000` for data copying and vectorization.

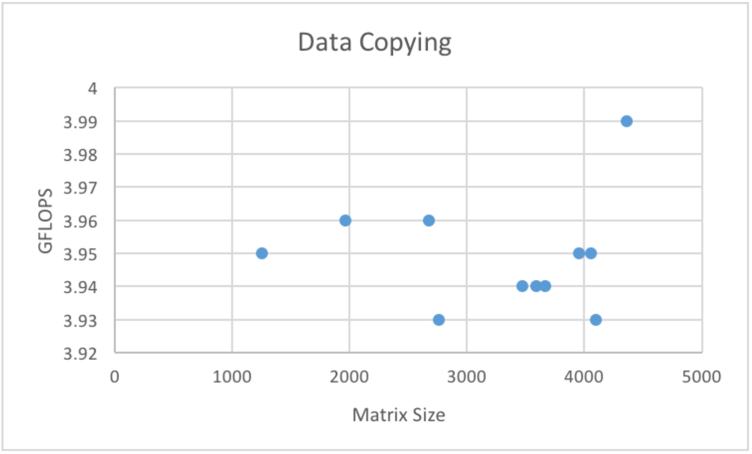
Graphs are shown below:



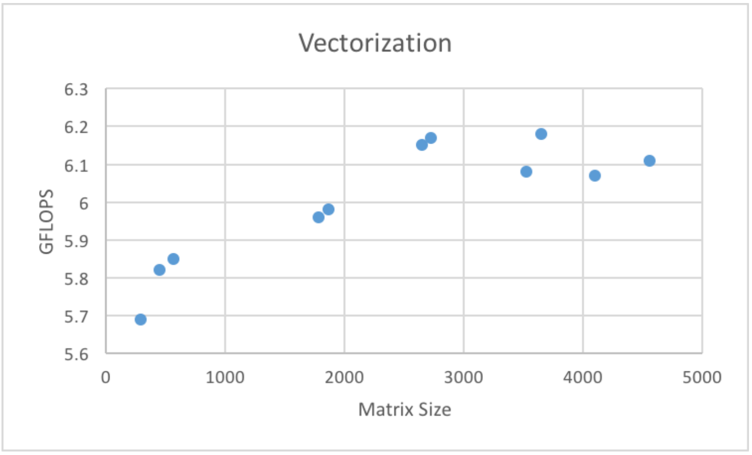
We can see how in small values the `GFLOPS` tend to go higher. This might be because of the use of floating point arithmetic and because the time spent on small matrices tends to be zero, then the `GFLOPS` will tend to go higher.



The performance gain after `Cache blocking` is notizable but not significant. It has a consistent performance behaviour for most matrix size, but it drops for the matrix of size `4096`. Considering it takes this program a couple of minutes to run on matrices bigger than 4000, there could be external factors that make it slow.



Data copying was the most difficult optimization of them all and it is definetly one bottleneck on the performance. It improved performance by a factor of two over `Cache Blocking`. But there could be a more efficient way to calculate the indexes of the matrices copied. Also it makes matrices bigger, wasting time on unnecesary calculations and wasting space on unused elements on the copy matrix. This could be made more efficient by using a smarter copying technique.



We can clearly see that vectorization brought an important increase in the performance of the program. This could be higher if the architecture used for testing was able to use certain features of intrinsics like `_mm_256 registers`. Unfortunately this was not tested within the reach of this project.

Below are shown the means of the test ran with every optimization. Therefore we claim that the program performs close to these values depending on the amount of optimizations used.

```
Vectorization:          6.01
Data Copying:           3.95
Cache Blocking:         1.84
Register Blocking:      0.63
```