CS 441 Project 4

Authors

Phillip Sime

2013-11-24

Summary

This software models main memory performance when performing intensive calculations. The two types of calculations used in this program are scalur multiplication and matrix multiplication. This software also incorporates a highly accurate timing mechanism which will output the average time take to perform each calculation in megaFLOPS.

Build

To build this software, simply navigate to the directory containing the Makefile and all other included files on a unix-based machine or virtual machine. Type the command `make` into the terminal to trigger the build. This will cause the gcc build commands listed in the Makefile to execute, compiling the source code into an executable file.

Usage

To use this software, there are two seperate commeands depending on the which type of calculation you would like to use for modeling.

The first command is `./scalarmult <N-limit>` where `<N-limit>` is an optional argument for the size of the NxN matrix used for calculations. N will initially start out at 2 and increase by powers of 2 until `<N-limit>` is reached. If no second argument is provided the program will default to an N of 1024.

The second command is `./matrixmult <N-limit>` where `<N-limit>` is an optional argument for the size of the NxN matrix used for calculations. N will initially start out at 2 and increase by powers of 2 until `<N-limit>` is reached. If no second argument is provided the program will default to an N of 1024.

Test Cases

- Test 1 consisted of running the command `./scalarmult` with an iteration number set at 100,000. This test created an accurate average runtime in megaFLOPS for scalur multiplication from 2 until 1024 increasing by powers of two for `N`.
- Test 2 consisted of running the command `./matrixmult` with an iteration number set at 100. This test created an accurate average runtime in megaFLOPS for matrix multiplication from 2 until 1024 increasing by powers of two for `N`.

- Test 3 consisted of running `./scalarmult <num>` and `./matrixmult <num>` for various values of `N`. These numbers included negative numbers and very large numbers. The program used the proper value for `N` in all cases.
- Test 4 consisted of running `./scalarmult <num>` and `./matrixmult <num>` for various invalid values of `N`. These values consisted of invalid characters and invalid characters mixed with valid characters. The program ignored the invalid input and used a default value of 1024 for `N`.

Examples

The following example shows the output of `shell$ ./scalarmult 2`. Note how each experiment only runs once since a max value for `N` is set to 2.

```
shell$ ./scalarmult 2

--------------------------

Executing: run_experiment_ij()

Matrix Size in bytes: 32
megaFLOPS = 61.662451


--------------------------

Executing: run_experiment_ji()

Matrix Size in bytes: 32
megaFLOPS = 63.291139
```

The following example shows the output of `shell$ ./matrixmult 2`. Note how each experiment only runs once since a max value for `N` is set to 2.

```
shell$ ./matrixmult 2

--------------------------

Executing: run_experiment_ijk()

Matrix Size in bytes: 32
megaFLOPS = 199.975003


--------------------------

Executing: run_experiment_ikj()

Matrix Size in bytes: 32
megaFLOPS = 196.391310


--------------------------
```

```
Executing: run_experiment_kji()

Matrix Size in bytes: 32
megaFLOPS = 194.457949


---------------------------

Executing: run_experiment_kij()

Matrix Size in bytes: 16
megaFLOPS = 203.329521


---------------------------

Executing: run_experiment_jki()

Matrix Size in bytes: 32
megaFLOPS = 199.526126


---------------------------

Executing: run_experiment_jik()

Matrix Size in bytes: 32
megaFLOPS = 162.222448
```

The following example shows the output of `shell$ ./scalarmult invalid input`. Note how the `invalid input` is ignored and the experiments run with a default size of 1024 for `N`.

```
shell$ ./scalarmult invalid input

---------------------------

Executing: run_experiment_ij()

Matrix Size in bytes: 32
megaFLOPS = 49.091802

Matrix Size in bytes: 128
megaFLOPS = 54.624287

Matrix Size in bytes: 512
megaFLOPS = 50.039093

Matrix Size in bytes: 2048
megaFLOPS = 66.913060

Matrix Size in bytes: 8192
megaFLOPS = 67.028516

Matrix Size in bytes: 32768
megaFLOPS = 113.668719
```

```
Matrix Size in bytes: 131072
megaFLOPS = 303.008783

Matrix Size in bytes: 524288
megaFLOPS = 299.837111

Matrix Size in bytes: 2097152
megaFLOPS = 311.276311

Matrix Size in bytes: 8388608
megaFLOPS = 296.175698


---------------------------

Executing: run_experiment_ji()

Matrix Size in bytes: 32
megaFLOPS = 213.675214

Matrix Size in bytes: 128
megaFLOPS = 213.191206

Matrix Size in bytes: 512
megaFLOPS = 135.889760

Matrix Size in bytes: 2048
megaFLOPS = 207.605161

Matrix Size in bytes: 8192
megaFLOPS = 233.463745

Matrix Size in bytes: 32768
megaFLOPS = 277.711861

Matrix Size in bytes: 131072
megaFLOPS = 273.359800

Matrix Size in bytes: 524388
megaFLOPS = 186.629809

Matrix Size in bytes: 2097152
megaFLOPS = 156.660171

Matrix Size in bytes: 8388608
megaFLOPS = 50.882077
```

Known Bugs and Problem Areas

- No bugs or problem areas are known at this time.

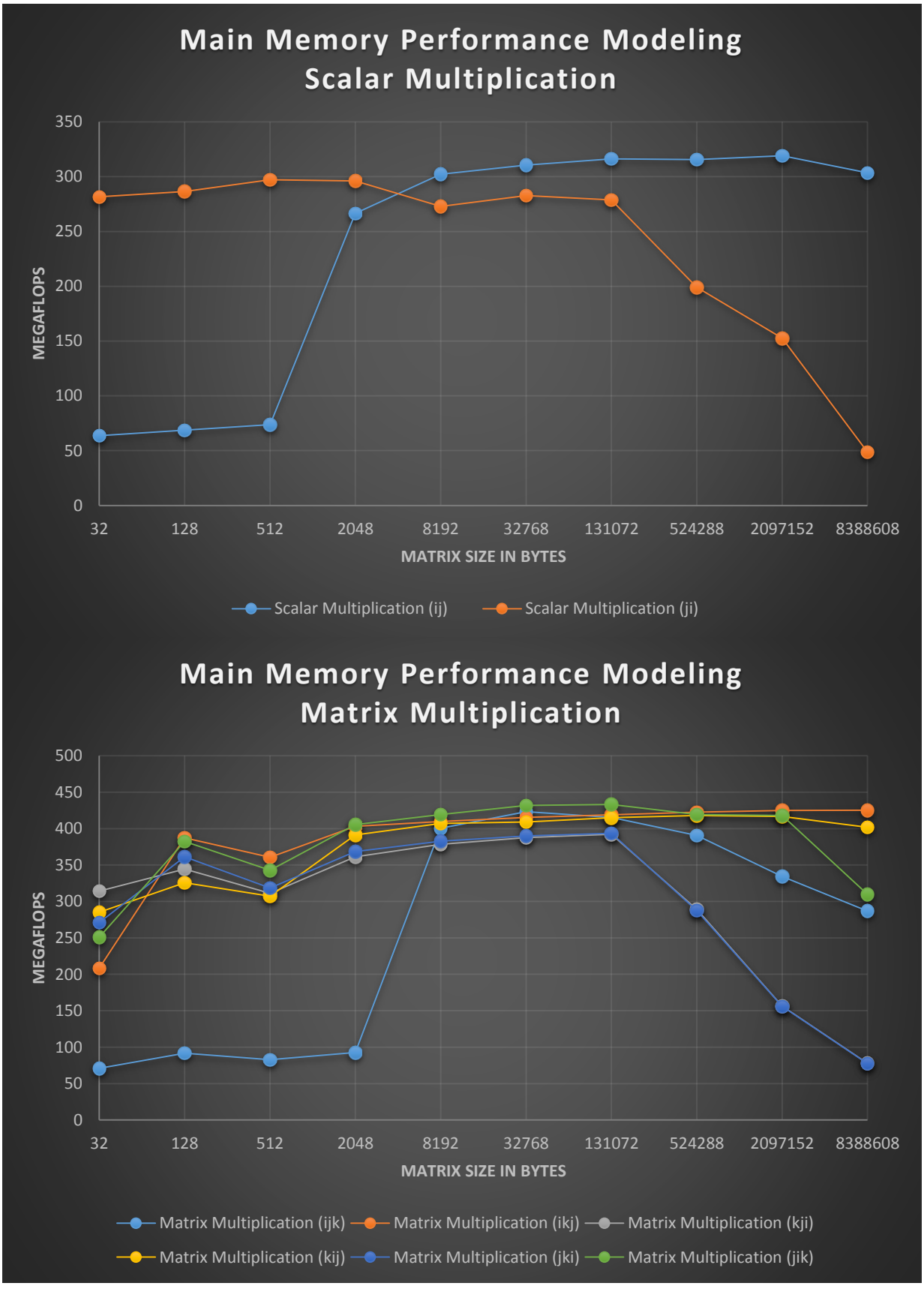
Raw Experimental Data

- For scalar multiplication see `docs/scalarmult_10000.txt`
- For matrix multiplication see `docs/matrixmult_100.txt`

Output From lstopo

- See `docs/hwloc_results.txt`

Graphs


Main Memory Performance Modeling
Scalar Multiplication
MEGAFLOPS
350
300
250
200
150
100
50
0
32
128
512
2048
8192
32768
131072
524288
2097152
8388608
MATRIX SIZE IN BYTES
Scalar Multiplication (ij)
Scalar Multiplication (ji)
Main Memory Performance Modeling
Matrix Multiplication
MEGAFLOPS
500
450
400
350
300
250
200
150
100
50
0
32
128
512
2048
8192
32768
131072
524288
2097152
8388608
MATRIX SIZE IN BYTES
Matrix Multiplication (ijk)
Matrix Multiplication (ikj)
Matrix Multiplication (kji)
Matrix Multiplication (kij)
Matrix Multiplication (jki)
Matrix Multiplication (jik)

Questions and Answers

Scalar Multiplication

Which experiment produces the best overall performance?

- The best overall performing experiment was when we had the `j` value in our outer loop and the `i` value in our inner loop. This would be the `run_experiment_ji()` function. The average megaFLOPS for this experiment was 239 megaFLOPS compared with 234 megaFLOPS for the other experiment called with the `run_experiment_ij()` function.

Describe the cause of the divergence of performance between the experiments.

- The reason there is a difference in the divergence between the two experiments is based on how the matrix is laid out in memory. My computer has an L2 cache with a size of 256KB. You can see from the graph that up until the matrix size reaches 256KB (262144 bytes) the program runs about equal. This is caused by the program using the L1 and L2 to store most of the values. This results in fast lookup times for either experiment. When we reach a matrix size of 512KB (524288 bytes) we see that the performance begins to rapidly decline for the `ji` experiment. The reason this experiment diverges is because the program needs to "hop" around in memory to access each value in the array. Since it can no longer store the values in L1 and L2 it is now relying on L3 cache and main memory for the data it needs. This is where the slowdown comes in. The `ij` experiment is stored in a more continuous chunk of memory which requires far less "hopping" around to get the needed value. That is why we see this experiment continue to run efficiently.

Matrix Multiplication

Which experiment(s) produces the best overall performance? Why?

- The best overall performing experiments would be the `ikj` and `jik` functions. The reason that these two experiments is similar to why the scalar multiplication experiments performed in this manner. Our memory is allocated in a chunk that will be stored in L1, L2, L3, and main memory. Due to how matrices are stored with using C we have our needed values stored in memory in a manner that they are near each other. This results in them all being stored in some form of cache until they are needed by the program. For experiments `jki` and `kji` our values are spread out in memory so much that we cannot efficiently store all the needed values in cache.

Rank the six experiments from best to worst in terms of MFLOPS.

- 1) Matrix Multiplication `ikj`
- 2) Matrix Multiplication `jik`
- 3) Matrix Multiplication `kij`

- 4) Matrix Multiplication `kji`
- 5) Matrix Multiplication `jki`
- 6) Matrix Multiplication `ijk`

Describe the cause of the divergence of performance between the experiments.

- The cause of divergence is the same as the scalar multiplication above. When the program can no longer store all of the needed values in cache it needs to start using main memory to store these values. This will cause the lookup time to be much slower if the needed value does not have a good locality compared to the previously used value. If the locality is good there will be no decrease in performance because the values will be stored in cache.

You may notice that some of the experiments pair up. Why does that happen?

- The graphs will pair up when they are using similar values for calculations. Depending on the experiment running the values needed will have different localities. When a graph pairs up with another graph it means that the locality of values needed is similar to the comparable graph.