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: 16
megaFLOPS = 61.662451


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

Executing: run_experiment_ji()

Matrix Size in bytes: 16
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: 16
megaFLOPS = 199.975003


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

Executing: run_experiment_ikj()

Matrix Size in bytes: 16
megaFLOPS = 196.391310


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

```
Executing: run_experiment_kji()

Matrix Size in bytes: 16
megaFLOPS = 194.457949


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

Executing: run_experiment_kij()

Matrix Size in bytes: 16
megaFLOPS = 203.329521


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

Executing: run_experiment_jki()

Matrix Size in bytes: 16
megaFLOPS = 199.526126


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

Executing: run_experiment_jik()

Matrix Size in bytes: 16
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`.

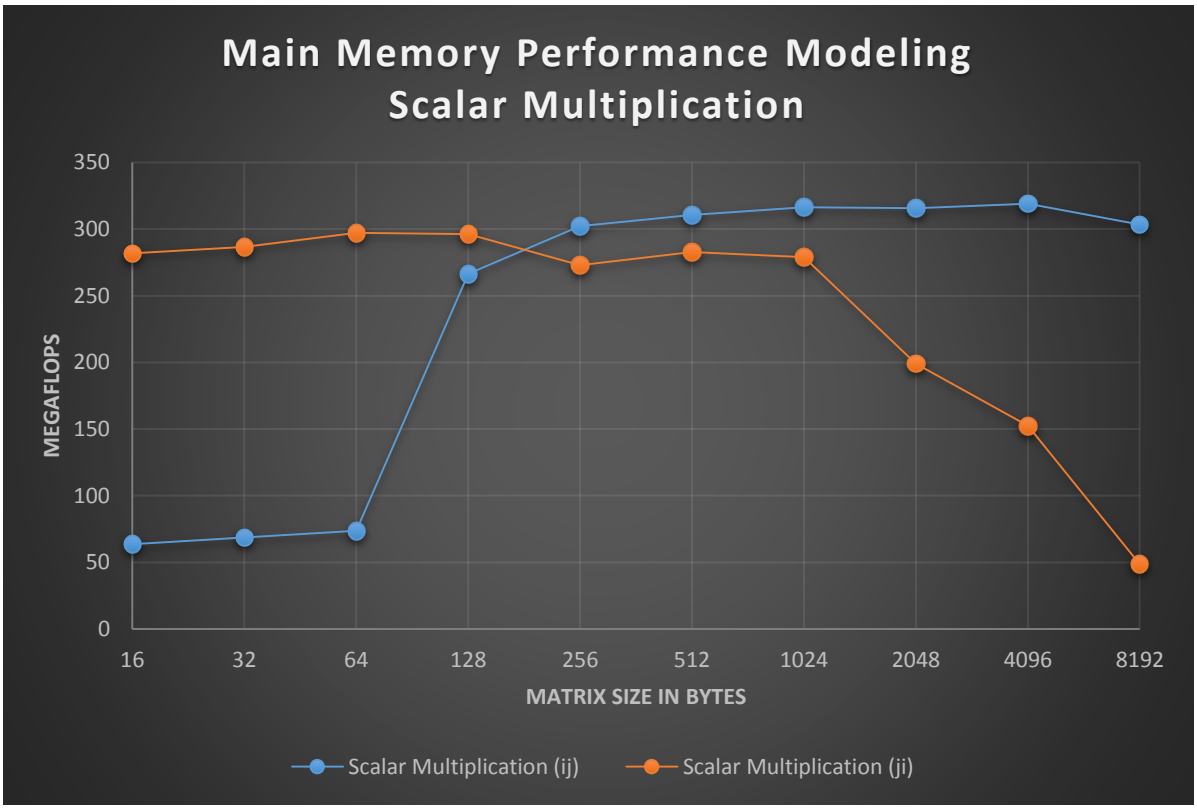
```
shell$ ./scalarmult invalid input

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

Executing: run_experiment_ij()

Matrix Size in bytes: 16
megaFLOPS = 49.091802

Matrix Size in bytes: 32
megaFLOPS = 54.624287

Matrix Size in bytes: 64
megaFLOPS = 50.039093

Matrix Size in bytes: 128
megaFLOPS = 66.913060

Matrix Size in bytes: 256
megaFLOPS = 67.028516

Matrix Size in bytes: 512
megaFLOPS = 113.668719
```

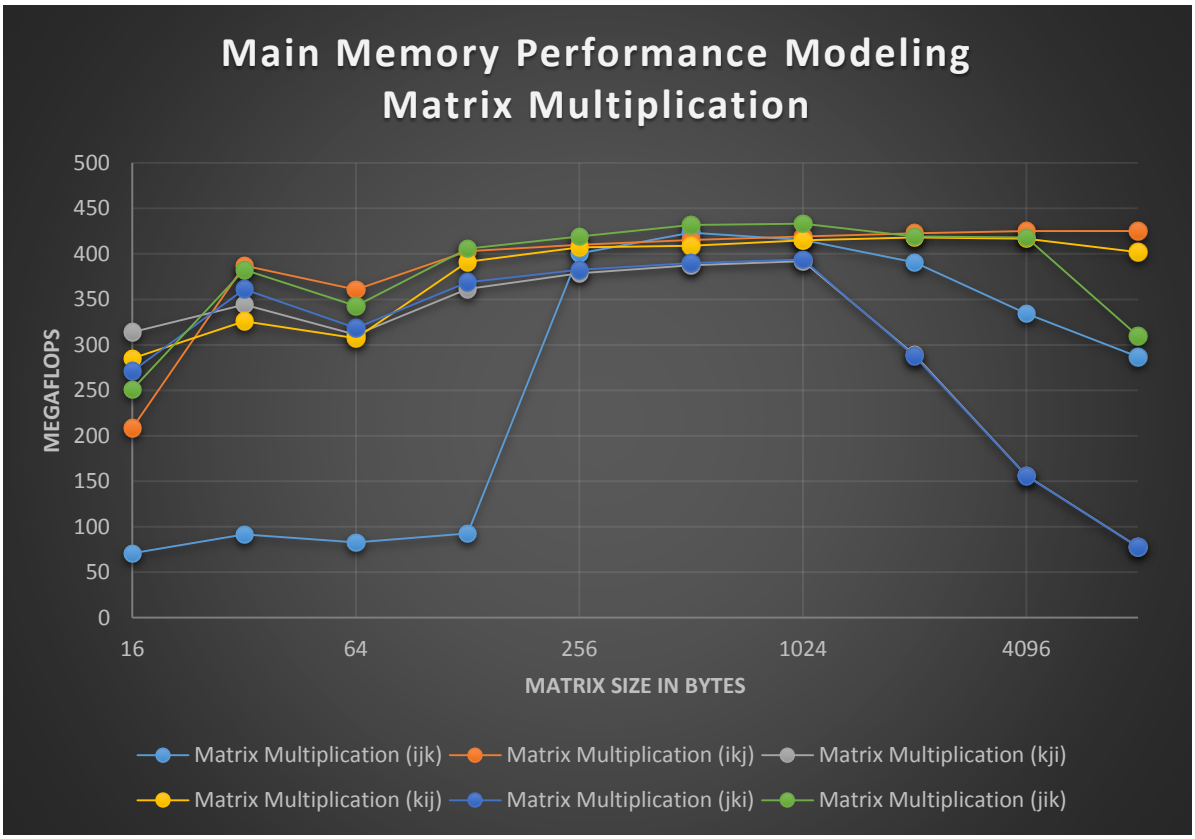
```
Matrix Size in bytes: 1024
megaFLOPS = 303.008783

Matrix Size in bytes: 2048
megaFLOPS = 299.837111

Matrix Size in bytes: 4096
megaFLOPS = 311.276311

Matrix Size in bytes: 8192
megaFLOPS = 296.175698


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

Executing: run_experiment_ji()

Matrix Size in bytes: 16
megaFLOPS = 213.675214

Matrix Size in bytes: 32
megaFLOPS = 213.191206

Matrix Size in bytes: 64
megaFLOPS = 135.889760

Matrix Size in bytes: 128
megaFLOPS = 207.605161

Matrix Size in bytes: 256
megaFLOPS = 233.463745

Matrix Size in bytes: 512
megaFLOPS = 277.711861

Matrix Size in bytes: 1024
megaFLOPS = 273.359800

Matrix Size in bytes: 2048
megaFLOPS = 186.629809

Matrix Size in bytes: 4096
megaFLOPS = 156.660171

Matrix Size in bytes: 8192
megaFLOPS = 50.882077
```

Known Bugs and Problem Areas

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

Graphs


Main Memory Performance Modeling
Scalar Multiplication
MEGAFLOPS
350
300
250
200
150
100
50
0
16
32
64
128
256
512
1024
2048
4096
8192
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
16
64
256
1024
4096
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 `i` value in our outer loop and the `j` value in our inner loop. This would be 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 expirments is based on how memory is managed by the system. The divergence in the graph is caused by the values needed not being stored in a quick access part of memory.

Matrix Multiplication

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

- The best overall performing experiments would be the `kji` and `jki` functions. Initially they take ~300 megaFLOPS to execute, but as the matrix size increases the runtime actually decreases. This is caused by the values being stored in memory which allows a faster lookup time.

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

- Matrix Multiplication `ijk`
- Matrix Multiplication `jki`
- Matrix Multiplication `kji`
- Matrix Multiplication `kij`
- Matrix Multiplication `jik`
- Matrix Multiplication `ikj`

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

- There is a relatively equal divergence pattern for all of the experiments except for `ijk`. The cause of the divergence is caused by main memory not having quick access to the value needed for the next calculation. `ijk` starts out much better than the rest because it, initially, has quick reference to the values needed for the calculations as they are the first values of the matrix as it resides in memory.

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

- The majority of the graphs initialy start out at around the same MFLOPS. The only exception to this would be the `ijk` experiment. This graph starts at a much lower MFLOPS because the neede values are stored in location of memory that does not require

an extensive lookup. The other graphs follow similiar paths as eachother as they load/access values into memory.