Noah Storms

12/09/2023

Report and Log

**Team Composition**

Noah Storms - Code Development

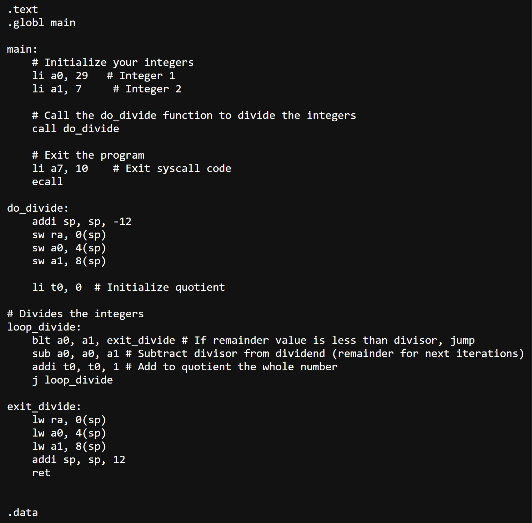
Teammate 2 – Performance Analysis

Teammate 3 – Performance Analysis

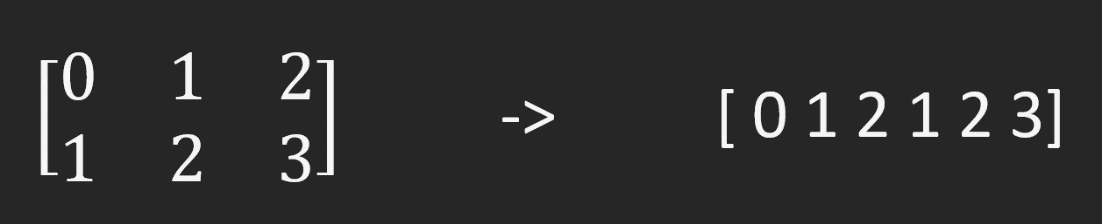
Our team setup was pretty straightforward – Noah took the reins on coding development, while the other teammates would develop the performance analysis of the written code. This approach made sense as too many people writing code could lead to possible loss in translation amongst each iteration of code updated. Nevertheless, this didn't mean we operated in complete isolation. Ideas were tossed around, and updates were announced in the group on the progress of their work. The other teammate’s joint focus on analysis played a crucial role in creating a dynamic environment for comparisons and discussions that strengthened our conclusions. It also allowed one to focus solely on the writing of the analysis and the other to focus on making the data look pretty for the reader. It could be said that the team composition was successful amongst us.

**Method of implementation of unsigned column average**

Our team was tasked with developing an algorithm to compute the average of each column in a matrix. The unique challenge presented was to implement this algorithm without employing a division operation, due to limitations in the simulation. Our first step was to implement this division function and have it ready for the main code. The initial code for this division function had incorporated the handling of negative numbers and the development of addition and subtraction functions from scratch. However, a subsequent realization occurred when management communicated that specific pseudo instructions for addition and subtraction were at our disposal. It was also informed that the matrix inputs would exclusively consist of positive integers. In response to this clarification, we proceeded to refine our code, adhering strictly to the parameters provided. The final version of our algorithm reflects a more simplified approach in accordance with the project requirements. It essentially checks if the dividend is less than the divisor. If so, it subtracts the dividend by the divisor and adds 1 to the quotient. This will loop until the dividend is less than the divisor.



With the completion of the division function, it was time to develop the core algorithm for computing the average of each column in the matrix. The first step was figuring out how to create a matrix in this simulation. That's when the idea popped into our heads to put it all in one array. Each row would be taken and placed at the end of the previous as so:



Once written, our team needed to figure out how to grab each integer from the same column. Knowing the memory address goes up by 0x4, we could multiply that by the number of columns we have. This tells how many elements to jump in memory to get to each integer in our current "n" column.

Another problem arose which was how our pointer would reset to the start of the array and then move to the next column. Knowing the memory address goes up by 0x4, we could just go up 0x4k times depending on column "k". The reset of the pointer was easily done, by saving the initial array address in a register.

The rest of the algorithm was exactly as told by our company. This was to load the sum into a register, exit the loop, divide that sum by the number of rows using our division function, and store the now average column value into our new avg array.

Thinking the team was done, management presented another algorithm which was to be believed to be more efficient. The only changes needed for this algorithm was to now store in the avg array when inside the inner loop. Once the inner and outer loop finished, the avg array, stored with the sums of each column, would then be loaded back into the simulator, divided by number of rows, and then stored back into that same index of the avg array.

**Analysis of Performance**

**Analyzing the differences between part 1 and part 2 implementations.**

The table and graph below show the differences between the two implementations. You can see the size of the matrix used and the number of cycles each implementation took to complete.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Part 1 vs.  Part 2** | **2x3** | **8x8** | **100x100** | **200x200** | **256x256** |
| Cycles Part 1 | 860 | 2,234 | 1,214,494 | 6,694,202 | 11,165,506 |
| Cycles Part 2 | 840 | 1123 | 1,373,121 | 7,564,137 | 12,859,225 |

A graph of different colored bars

Description automatically generated

**Speed Up % differences between Implementation 1 and 2**

The graphs below show the speed up percent differences between implementation 1 and implementation 2. The results below show that there is a slightly higher improvement when implementing divu instruction.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Speedup Part 1 vs.  Part 2** | **2x3** | **8x8** | **100x100** | **200x200** | **256x256** |
| Block = 16 Assoc. = 2 | 2.38% | 98.93% | -11.55% | -11.50% | -13.17% |

A graph with numbers and a bar

Description automatically generated

**Part 1 vs. Part 2 Using the divu instruction**

For bigger datasets likely having bigger matrix sizes we recommend that using the implementation used in part 1 over part two even when implementing division in hardware. As presented in the table below you can see that there is a slightly higher speed up improvement when using the first implementation.

|  |  |  |  |
| --- | --- | --- | --- |
| **Implementation** | **Matrix Size** | **Cycle Count** | **Speed Up %** |
| Part 1 | 256x256 | 11,152,866 | 0.11% |
| Part 2 | 256x256 | 12,846,889 | 0.10% |
| Part 1 Speed Up vs Part 2 |  |  | -13.19% |

**Repeating the analysis of steps 1 and 2 with four times as much data cache.**

The last test experiment used a 256x256 size matrix with different cache settings for the machine to see how the cycle count was affected to see if it would result in a speedup or slowdown in overall execution.

|  |  |  |  |
| --- | --- | --- | --- |
| **Speedup using do\_div and 256x256 matrix** | **Part 1 control vs. Part 1 with increased cache** | **Part 2 control  vs.  Part 2 with increased cache** | **Part 1 with increased cache  vs.  Part 2 with increased cache** |
| Block = 64 Associativity = 2 | -9.67% | -73.84% | -72.78% |
| Block = 16 Associativity = 8 | -12.95% | 0% | 0.26% |
| Block = 32 Associativity = 4 | -10.84% | -48.46% | -47.08% |

|  |  |  |  |
| --- | --- | --- | --- |
| **Speedup using divu and 256x256 matrix with varying cache block sizes and associativity** | **Part 1 control vs. Part 1 with increased cache** | **Part 2 control  vs.  Part 2 with increased cache** | **Part 1 with increased cache  vs.  Part 2 with increased cache** |
| Block = 64 Associativity = 2 | -9.76% | 0.14%% | -72.78% |
| Block = 16 Associativity = 8 | -12.96% | 0.11% | 0.26% |
| Block = 32 Associativity = 4 | -10.89% | 0.10% | -47.08% |

A graph of different colored lines

Description automatically generated

A graph of different types of data

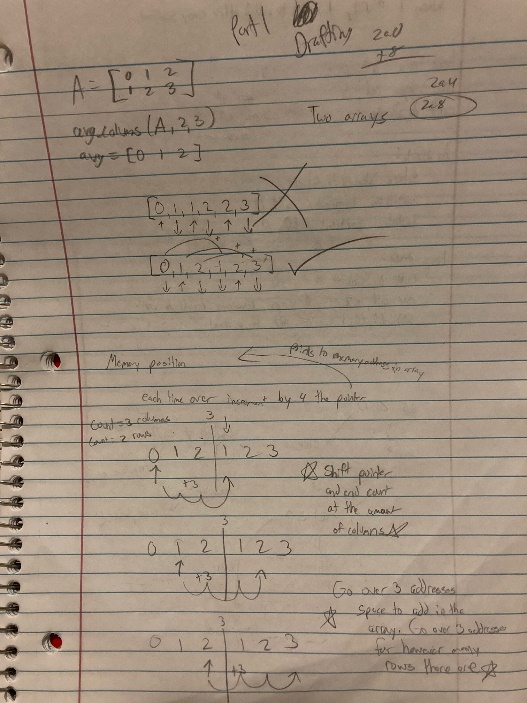
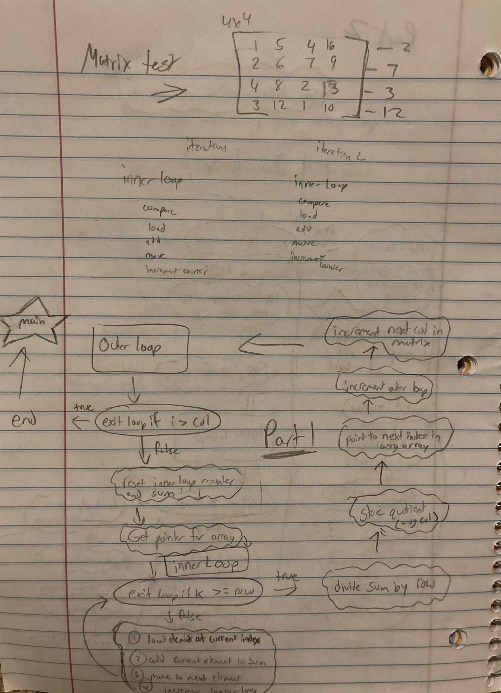
Description automatically generated with medium confidence

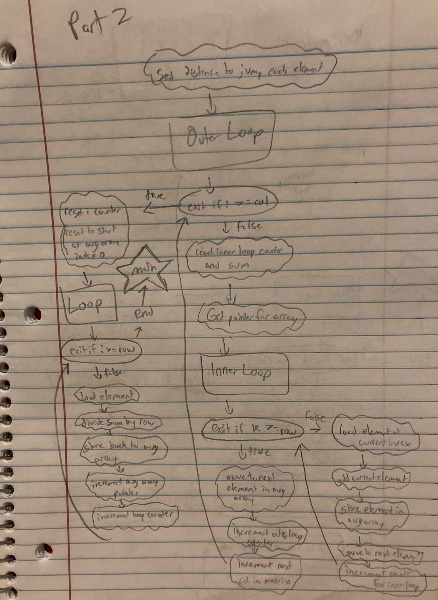
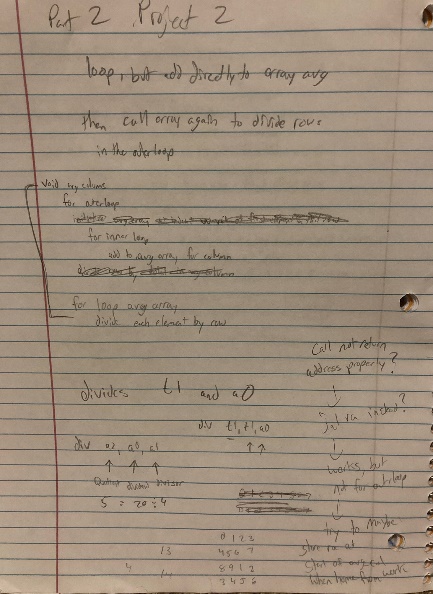
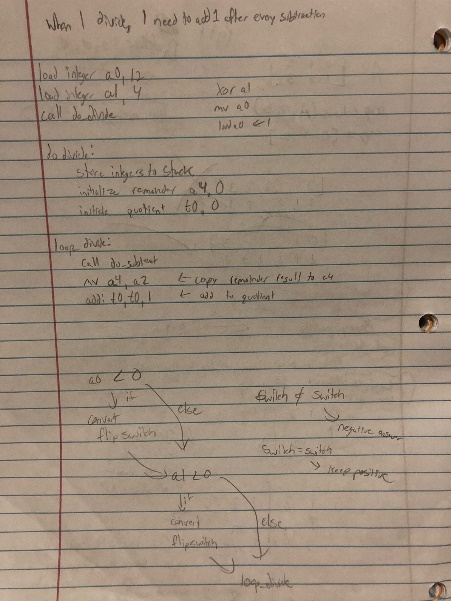
**Recommendations of for adding more cache vs. implementing division in hardware**

Based on the results of the data in our project increasing the cache block size and associativity was very detrimental to the efficiency of the program. When we did our tests with increased cache block size every time it resulted in higher cycle count therefore making the program take longer to complete. Based on the data and tests, our recommendation is to focus on implementing division in hardware if you are focusing on the maximum efficiency of a program.

**Additional comments**

Rough Note Sketches for the Development of the code:

Challenges:

There were some notable challenges while creating this algorithm. The first was being able to understand and apply a matrix into qtrvsim. Initially, I started to create multiple rows inside of a single array (.words) but was getting multiple issues such as limited number of registers to work with. This led to that new implementation of using a single array. This was quickly easy to understand from getting from point A to point B. However, it took a couple attempts and breaks to get the whole idea of skipping by 0x4 in memory and not by 0x1.

The other challenge was the division function created. There was a point where the return call was infinitely looping in the outer loop. Took some debugging and understanding what was really happening to be able to develop the idea of just storing the return address on the stack of each function and then grabbing it when it was time to return to the previous location in which a function was called.