Project Assignment in
EDA331, *Computer Systems Engineering*

Optimizing Hardware and Software in Scientific Computing

Department of Computer Science and Engineering
Chalmers University of Technology

April 20, 2016

1. Background

Solving large systems of equations is an important and demanding task in many scientific and technical applications. Gaussian elimination is a commonly used method for solving systems of linear equations, that is, systems of the form **Ax = b**.

The elimination is a two-step process. In the first step, the coefficient matrix **A** is reduced to upper triangular form, a matrix with ones on the main diagonal and zeros in all positions below the main diagonal. The second step uses back substitution to find **x**. The C code in figure 1 performs the first step of Gaussian elimination.

```
/* reduce matrix A to upper triangular form */
 void eliminate (double **A, int N) {
        int i, j, k;
         /* loop over all diagonal (pivot) elements */
        for (k = 0; k < N; k++) {
                /* for all elements in pivot row and right of pivot element */
                for (j = k + 1; j < N; j++) {
                        /* divide by pivot element */
                        A[k][j] = A[k][j] / A[k][k];
                }

                /* set pivot element to 1.0 */
                A[k][k] = 1.0;

                /* for all elements below pivot row and right of pivot column */
                for (i = k + 1; i < N; i++) {
                        for (j = k + 1; j < N; j++) {
                                A[i][j] = A[i][j] - A[i][k] * A[k][j];
                        }
                        A[i][k] = 0.0;
                }
        }       /* end pivot loop */
}        /* end eliminate */
```

Figure 1: Triangularization of matrix A of size N x N

A computer manufacturer, Moon Macrosystems, has decided that their next product must handle this kind of computation more efficiently. (Their customers are just crazy about Gaussian elimination.) They need your help with developing an assembler routine for the elimination algorithm in figure 1 and with tailoring the memory subsystem of their new computer to fit the data access patterns of the algorithm. The goal is to find the best possible trade-off between price and performance.

$$\begin{pmatrix} 57 & 20 & 34 & 59 \\ 104 & 19 & 77 & 25 \\ 55 & 14 & 10 & 43 \\ 31 & 41 & 108 & 59 \end{pmatrix} \qquad \begin{pmatrix} 1.00 & 0.35 & 0.60 & 1.04 \\ 0.00 & 1.00 & -0.86 & 4.73 \\ 0.00 & 0.00 & 1.00 & -0.41 \\ 0.00 & 0.00 & 0.00 & 1.00 \end{pmatrix}$$

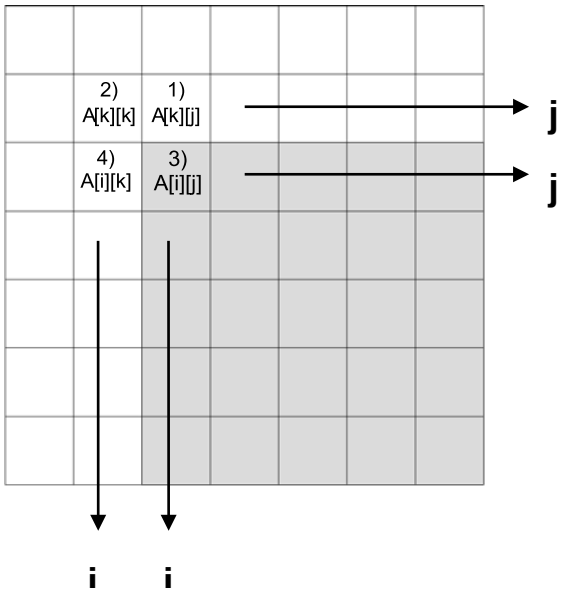
(a) Before elimination (b) After elimination

Figure 2: An example of Gaussian elimination.

2. Algorithm Description

As mentioned in Section 1, the first step of Gaussian elimination is to reduce a matrix to triangular form. In this assignment you don't need to worry about the second step (back substitution). Figure 2 shows an example 4×4 matrix before and after the elimination step.

The elimination routine contains division operations. Thus, the result will not always be integers. If you rounded all results to integers you would end up with unacceptably large errors, especially at the bottom of the matrix where elements are updated many times.



1. First j loop:
   A[k][j] = A[k][j] / A[k][k]
2. After first j loop:
   A[k][k] = 1.0
3. Nested j loop:
   A[i][j] = A[i][j] - A[i][k] * A[k][j]
4. i loop:
   A[i][k] = 0.0

Figure 3: Data access pattern for one iteration of the elimination algorithm (k = 1).

In order to maintain good precision throughout the computation you should use floating point representation for the matrix elements and floating point instructions for all matrix computations. Matrix computations are highlighted with **bold** font in figure 1. Chapter 3.5 of the course book[1] gives more information about floating point representation and MIPS floating point instructions. Single precision floating point representation (32 bits) gives sufficient numeric precision for the application at hand.

Figure 3 is a graphical visualization of the algorithm from figure 1. It is not necessary to understand exactly what the algorithm does, but you will probably find it interesting to analyze the data access patterns. Is it possible to take advantage of locality in order to reduce the number of cache misses?

Optional: An alternate version of the elimination algorithm is given in appendix C. What is the difference between the two versions? Can you think of any situations where the alternative algorithm will perform better?

3. Computer System Description

Processors

The computer system consists of a MIPS-compatible microprocessor with five pipeline stages and delayed branching. There are also two coprocessors, one for memory management and exception handling (coprocessor 0) and one for floating point arithmetic operations (coprocessor 1). The processor core and the coprocessors may not be modified or replaced

Caches and Write Buffer

The memory system includes separate instruction and data caches. The block replacement policy is Least Recently Used (LRU) and the write policy is Write Back. The data cache has an optional write buffer of variable size to reduce the number of stall cycles due to memory writes. Cache size, block size and associativity are configurable parameters and they can be set differently for the two caches. Available configurations are given in figure 4.

Main Memory

Depending on which type of memory module you choose, the main memory has an access time of either 44 ns for the first word in a block and 8 ns for following words, or 30 ns for the first word and 6 ns for following words.

Component costs for the possible hardware configurations are given in appendix A.

[1]Hennessy & Patterson: Computer Organization and Design, the Hardware Software Interface, ed. 5

Caches

Word size:	4 bytes
Cache size:	32, 64 or 128 words
Block size:	1, 2, 4, or 8 words
Associativity:	1, 2 or 4
Replacement policy:	LRU
Write policy:	Write back

Write buffer

Size:	0..12 words

Memory

Access time:	44 ns & 8 ns or 30 ns & 6 ns

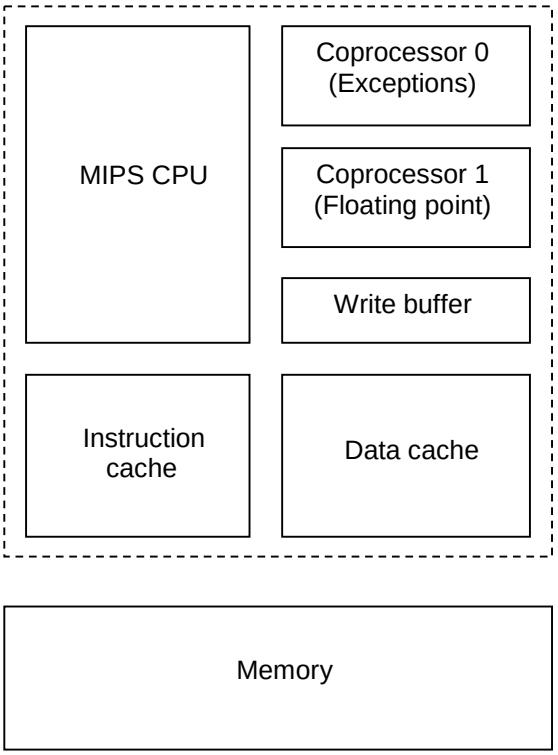


Figure 4: Computer hardware.

4. The Assignments

Assignment 1

Implement the elimination routine in MIPS assembler. Use single precision floating point instructions for all matrix computations. Your program should run in the MIPS simulator MARS (see appendix B).

Assignment 2

Construct a computer system considering both *price* and *performance*. *Performance* is measured as the time it takes your program to triangularize a 24 × 24 reference matrix. *Price* is the sum of the component costs for processor, caches and memory. Your goal is to minimize the product *Price* × *Performance*. The unit of this measure is *µsC$* (microSecondChalmersDollar).

Competition

Who constructs the best system? A scoreboard with the top results will be published on PingPong.

5. Examination

The following is required in order to pass the assignment:

- A live demonstration of your program (during a lab session)
  - It is possible to reach 900 *µsC$* with a reasonable amount of effort, and much better performance is possible. Results equivalent to a metric below 600 *µsC$* have previously been recorded.
- A report **written in English** containing:
  - A description of the computer system and the program. Someone with the appropriate background (for example a course like this) should be able to understand the report even if he or she hasn't read this document.
  - The source code for assignment 1 as a separate file.
  - The answer to assignment 2: what is the best *Price* × *Performance* value you have obtained? A good motivation for your choice of parameters is necessary. A working program alone is not sufficient. We want to know what considerations you made when choosing components and what measures you took to improve the execution time of the program. **Your justifications and explanations are more important than the exact result. You should include data from several test runs with different configurations to show that your solution is good. These tests should be performed in a systematic and well-planned fashion.**
  - To simplify comparison and recreation of results, you should report the following data for each of your test cases:
    - **Configuration:** Cache size, associativity and block size for each of the caches, processor frequency, memory access time and write buffer size.
    - **Results:** The number of clock cycles needed to triangularize the 24 × 24 matrix, execution time (*µs*), total component cost (*C$*) and *Price* × *Performance* (*µsC$*).
  - The length of the report should be 3-4 pages (not counting source code).
  - Plagiarism check: The report and assembly code will be checked with the Urkund and the MOSS tools.

Deadlines for the demonstration and the report are given on PingPong.

6. Hints

- The assembler file gauss_template.s, found on PingPong, contains some code that might help you get started. It also contains the 4 × 4 matrix from figure 2(a) and the 24 × 24 matrix you should use when reporting your results. Matrices are stored in row-major order.
- The assembler routines in gauss_template.s may be changed or not used at all. You are free to write your own helper routines. You may use your own algorithm, as long as the result is correct for any given 24 × 24 matrix, and the numeric precision is no worse than that of the given algorithms.
- It is probably a good idea to get a working program running using the given helper routines before bothering with optimizations. Do note, though, that the given helper routines have not been written with optimization as a goal and calling them often will result in bad performance.
- Use the small matrix when developing and debugging your code. However, you should try larger matrices as well to see how the performance scales with problem size. A 24 × 24 matrix will not fit in the cache, but a 4 × 4 matrix might.
- One tricky thing about the assembler code is that there are several variables to keep track of. Use code comments to document what data you have in which registers. There's nothing wrong with a comment for each line of assembler code!
- For integer arithmetic, use instructions that do not result in overflow exceptions (addu, addiu, subu, subiu). In this particular exercise you may disregard possible exceptions. This also means you do not need to worry about dividing by zero if your subroutine should be given a matrix that is not diagonalizable or would need rows to be switched.

Some Suggestions for Code Optimizations (in Increasing Difficulty)

- Familiarize yourself with the instruction list (appendix A in the course book)
- Fill delay slots with useful instructions and move instructions around to avoid load-use and branch stalls.
- Check whether the function of any pseudo instructions you use can be expressed more efficiently with normal instructions.
- Avoid slow floating point operations whenever possible.
- Avoid performing array index calculations. Instead, maintain pointers to the data to be used.
- Avoid calculating the same value multiple times.
- Step through your program to see which penalties it incurs at which points.
- Do not calculate those values of the result matrix that are known at the start.
- Optimize the code for a 24 × 24 matrix.
- The table in appendix B shows a quirk of the system regarding double-word loads/stores – figure out when to use these.
- Unroll loops when this is beneficial.
- And many more optimizations…

A. Component Prices

In this assignment we use a made-up currency called "Chalmers Dollar" (C$). The component prices below do not correspond to actual prices in any real currency.

CPU

The CPU operates at a clock frequency of 500 MHz. Its price is *2C$*.

Caches

128 byte instruction and data caches with block size of 4 words and associativity 1 are included in the price of the CPU. Larger caches will cost you more. The caches are the slowest components of the system. Hence, your choice of caches will affect the clock frequency of the CPU. The table below shows maximal clock frequencies for the available cache configurations. If you choose different configurations for the instruction and data caches, the slowest component will determine the system clock frequency. Available block sizes are 1, 2, 4 and 8 words. The choice of block size does not affect the price or the clock frequency. Please note that each cache is configured and paid for separately.

Cache Size	***Associativity***	***Max Clock Frequency***	***Additional Cost***
32	1	500	0
32	2	475	0
32	4	450	0
64	1	475	0.25
64	2	450	0.25
64	4	425	0.25
128	1	450	0.5
128	2	425	0.5
128	4	400	0.5

Write Buffer

The cost of buffer space is 0.03 C$/word. You are allowed to use a write buffer of up to 12 words, or none at all.

Memory

There are two types of memory modules to choose from. The table below lists their costs and their latencies for the first word as well as for other words that are accessed from a block.

Type	***1st word access time (ns)***	***Access time, other words (ns)***	***Cost (C$)***
1	44	8	0.5
2	30	6	1.0

B. MIPS Assembler and Runtime Simulator (MARS) and Performance Evaluation Tool (PET)

For this assignment you should use the simulator MARS[2]. Compared to MipsIt (the tool used in the laboratory exercises), MARS is a high-level simulator. Rather than simulating the logic of the CPU, it simulates the behavior of instructions and their effects on registers and memory. This approach has advantages, such as increased simulation speed and generality with respect to actual MIPS implementations. On the other hand, behavioral simulation does not reflect how pipeline hazards and cache memory misses affect the execution time of a program.

Evaluating Performance

Due to these limitations of MARS, we provide a plug-in that allows detailed measurements of program execution times. This utility is launched via the "Tools" menu in MARS (the tool name is "EDA331 Performance Evaluation Tool (PET)"). Note: The performance evaluation tool is not part of the package distributed via the MARS web site. **In order to use the tool, you need to download the MARS .jar archive from PingPong.**

Below are listed some reasons for stalls in the execution of a program and how they are handled by PET.

Floating-point Instructions

PET simulates a simplified model of a floating point coprocessor. The simulated coprocessor is non-pipelined and has multi-cycle instructions. The table below lists cycle counts for floating point operations. After issuing a floating point instruction, the processor will stall until the result is available.

Operation	***Single***	***Double***
move, absolute value,	1	1
negate	2	2
add, compare	4	5
multiply	12	19

Write Buffer

When the write policy of the data cache is set to write-back and a dirty block is evicted from the cache, the block will be put in the write buffer and written back to memory in the background, while the program keeps executing. If the write buffer is too full to contain the block, the processor will stall until enough words from the write buffer have been written back to memory to fit the new block in the buffer. The number of such stall cycles is displayed by PET.

If a block is being written back to memory from the buffer and the running program requires that data or instructions be read from main memory, the write

[2] http://courses.missouristate.edu/KenVollmar/MARS/

from the write buffer will be interrupted, and the current instruction's memory access will take precedence. After this access is done, the write buffer will again start writing to memory, with a new first-word access penalty for the next word written to memory.

Pipeline Stalls

PET is designed to simulate the pipeline depicted in the lab 2 instructions on page 21. This means that there are load-use stalls and branch stalls when there are register conflicts. The performance evaluation tool displays the number of cycles the processor has stalled due to these kinds of hazards.

C. Alternative Algorithm

```
/* reduce matrix A to upper triangular form */
void eliminate (double **A, int N, int B) {
      int i, j, k, I, J;
       /* size of block is (M*M) */
      int M = N/B;

      /* for all block rows */
      for (I = 0; I < B; I++) {
            /* for all block columns */
            for (J = 0; J < B; J++) {
                  /* loop over pivot elements */
                  for (k = 0; k <= Min((I + 1)* M - 1, (J + 1) * M - 1); k++) {
                        /* if pivot element within block */
                        if (k >= I*M && k <= (I + 1) * M - 1) {
                              /* perform calculations on pivot */
                              for (j = Max(k + 1, J * M); j <= (J + 1) * M - 1; j++) {
                                    A[k][j] = A[k][j] / A[k][k];
                              }
                              /* if last element in row */
                              if (j == N) A[k][k] = 1.0;
                        }

                        /* for all rows below pivot row within block */
                        for (i = Max(k + 1, I * M); i <= (I + 1) * M - 1; i++) {
                              /* for all elements in row within block */
                              for (j = Max(k + 1, J * M); j <= (J + 1) * M - 1; j++) {
                                    A[i][j] = A[i][j] - A[i][k] * A[k][j];
                              }
                              /* if last element in row */
                              if (j == N) A[i][k] = 0.0;
                        }
                  }
            }
      }
}                    /* end eliminate */
```

D. Rounded Solution for the 24x24 Matrix

(Note that your solution needs to have higher accuracy than this)

1,00	0,47	0,93	0,95	1,09	0,23	0,39	0,91	0,33	0,65	0,57	0,75	0,43	0,61	1,13	1,09	0,75	0,85	0,16	0,72	0,01	0,28	0,16	0,96
0,00	1,00	-0,05	-0,14	2,51	0,56	1,37	2,12	1,70	0,58	1,68	1,23	1,29	2,68	0,35	1,82	1,23	1,93	1,14	2,38	0,85	0,92	2,36	2,35
0,00	0,00	1,00	0,30	-4,30	0,25	-0,78	-3,27	-3,00	-1,09	-2,20	-2,20	-1,92	-3,02	0,99	-1,75	-1,79	-2,54	-1,35	-2,90	-0,54	-0,39	-3,25	-4,30
0,00	0,00	0,00	1,00	1,60	0,38	-0,37	0,96	4,04	0,29	0,85	3,67	3,20	-0,81	-4,45	0,32	0,53	1,36	3,35	0,68	1,38	1,17	1,93	4,94
0,00	0,00	0,00	0,00	1,00	0,61	-0,47	0,15	2,49	-0,57	0,36	2,10	1,61	0,20	-2,47	0,96	0,32	1,35	2,21	0,99	0,61	0,89	1,48	2,39
0,00	0,00	0,00	0,00	0,00	1,00	-4,72	-1,80	11,27	-3,75	0,07	11,37	7,99	-3,54	-15,51	1,25	0,28	3,28	12,26	0,62	3,88	2,64	5,75	11,70
0,00	0,00	0,00	0,00	0,00	0,00	1,00	-0,11	-2,09	0,71	-0,28	-2,41	-1,50	0,76	3,25	-0,13	-0,41	-0,61	-2,49	0,24	-1,39	0,02	-1,81	-2,58
0,00	0,00	0,00	0,00	0,00	0,00	0,00	1,00	1,02	0,18	1,06	1,66	1,02	-0,39	-2,03	-0,48	0,86	-0,04	1,50	-0,45	1,90	-0,27	2,15	2,31
0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	1,00	0,18	0,29	1,06	1,01	-0,23	-1,27	-0,33	-0,02	0,02	1,03	0,31	0,22	0,69	0,32	1,64
0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	1,00	-0,90	-4,05	2,56	7,08	10,80	2,55	-4,99	8,98	0,51	6,48	-9,81	10,06	-4,08	0,99
0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	1,00	0,98	-0,77	-2,30	-2,33	-0,43	2,02	-1,16	1,32	-3,19	5,15	-2,17	3,59	-1,06
0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	1,00	-0,86	-2,19	-1,86	-0,12	0,42	-0,48	0,04	-2,48	3,01	-1,79	0,80	-1,18
0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	1,00	1,32	-0,86	-0,91	0,54	-2,74	-0,55	2,68	-3,30	-0,33	-1,24	1,90
0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	1,00	4,63	1,70	-3,16	4,01	-2,17	-1,25	-2,39	2,92	-3,36	0,77
0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	1,00	0,67	-0,66	1,64	0,50	-0,15	0,08	0,80	0,29	-0,14
0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	1,00	1,17	9,90	7,48	-1,44	11,56	4,70	8,96	-1,07
0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	1,00	0,18	1,94	-0,26	2,48	-0,74	2,63	0,48
0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	1,00	0,46	-0,20	1,08	0,57	0,50	-0,09
0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	1,00	1,67	-0,63	0,76	1,75	4,98
0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	1,00	0,64	1,76	1,64	7,26
0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	1,00	2,58	1,87	8,87
0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	1,00	0,57	3,10
0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	1,00	10,42
0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	1,00