Report

Home Assignment, DST15_035

Dennis Bennhage & Hampus Lidin

May 14th, 2015

1 System Overview

The system we have constructed is intended to perform the first step in *Gauss elimination* on any square matrix, while it has been specifically optimized for any 24x24 square matrix. It is optimized to yield the most efficient execution, whilst keeping the price as low as possible. Therefore the performance has often been compromised in favour of cheaper components.

The software will reduce the matrix to *upper triangle form*, which is characterized as a matrix with it's diagonal containing only ones, and all elements below the diagonal containing only zeros. The second step in the elimination process, *back substitution*, is not handled by this system.

2 Optimization

Both software and hardware have been in consideration when optimizing the system. The hardware is heavily dependent on how the program is executed, therefore the program have been written in such way that it would be beneficial for an optimal hardware configuration. We will also see that the software has to adjust for the hardware component costs sometimes as well.

2.1 Software

The algorithm iterates through the matrix in several different loops. The outermost loop iterates along the main diagonal (i.e. the *pivot elements*) of the matrix. In each loop, we do the two following operations; we minimize the current row (let's call it the top row) by dividing each element with the pivot element, and then we subtract each underlying row with their respective pivot elements multiplied by the top row. By the end of the first outmost loop, we end up with a matrix that has a computed top row, and thus its pivot element being a one and all the elements below the pivot being zeros. Then successively after each row gets computed, we will end up with our result matrix.

The first approach for writing the program was to directly translate an elimination algorithm written in C-code to Assembler, and not bother with any optimizations. After confirming that we had a working program, we started to find places in the code where we could do something to speed up the performance. We realized that since we had many loops in our program, we'd want to push out or eliminate as many instructions from the loops as possible. One way we had done this, was to convert the arrays into pointers instead. With arrays, you have to calculate each new address for every iteration of the loop. This will increase the number of instructions executed in each loop (the *overhead*). When using deep-nested loops as we do in our algorithm, it will add up to a lot of instructions, or clock cycles if you will, in the end. By using pointers instead, we could omit the use of indices of the arrays, and instead only increment the address by one byte. This minimized the overhead of the loops, since we only had to keep track of the address of the current element in the array.

After running the program with the provided *Performance Evaluation Tool* (PET), we could analyze if we had any *branch hazards* or *load-use hazards.* Branch hazards occur when an instruction calculates a value for a conditional branch, that is directly subsequent. The CPU will then have to stall one clock cycle due to the delay in the pipeline, which is a waste of performance. By reordering the instructions, for example inserting a completely independent instruction in between, we could avoid this type of hazard. Load-use hazards work in a similar way; when loading a value to a register, then immediately using it in another instruction, the CPU will have to stall one clock cycle.

Utilizing all the registers that were at hand was also a great way to optimize the performance. By having commonly calculated addresses and values stored in registers (such as the loop limit address), we could peel of some instructions from the code. Our goal was to reduce the program to only consist of the absolutely necessary instructions, and make sure that values used often only got calculated once. We found out later that it would be very important to make sure that all instructions, that were contained in the outermost loop, could fit inside the *Instruction Cache* memory unit (I-Cache). If we only could reduce the instructions to fit in the cache, we would almost exclusively get a 100% *hit rate.* High hit rates greatly improve performance by only wasting relatively few clock cycles for fetching instructions from the main memory.

To achieve this goal of going under the limit, we had to often think of ways to execute one function of the program in a different way, whilst still getting an identical result. A good source of improvement was to look at how the *branching* of the program was executed. In the beginning we had two branch-instructions in each loop; one for checking a condition for exiting the loop, and one to jump to the beginning of the loop. The reason we initially designed it like this was that sometimes we didn't wanted to enter the loop at all, if the condition was met (for exiting the loop) at the start. If we hadn't checked in the beginning of the loop, we would've had unwanted execution of the program. They way we improved this was to move the condition branch outside the loop instead, and checked the reverse condition (the condition to branch to the next iteration) in the end of the loop.

By following these optimization guidelines, we were able to construct a program that would fit in an optimal configuration of the I-Cache. But sometimes it was not always good to shorten the program too much. We actually had to restrict the program to a certain structure, as we only had limited component specifications. We found out that if we would read/write data from/to the memory too often, we would fill the memory *write buffer* too fast, resulting in the program having to wait for the memory to finish. By reducing the loops, we not only made the loop execute faster, we also brought the read- and write-instructions closer. This created a bottleneck, which outweighed the possible performance boost that we would get. For that reason we left out some optimization opportunities in favour for an optimal hardware configuration. But what was our optimal configuration? That is what we will discuss in the next chapter.

2.2 Hardware

When we were choosing the components we had to consider several things. Since the goal of this project was to get the best performance for as cheap as possible, we could not simply try to get the fastest execution time possible. With this in mind, we tested several different configurations. In the following subsections we describe our thought process when deciding which hardware configuration to use.

2.2.1 Memory

Since the entire 24x24 matrix is stored in memory we have to read memory every time we need a new value from the matrix, and write to memory every time we want to update a value in the matrix. This adds up to a lot of memory accesses, so a fast memory is going to greatly improve the execution time.

2.2.2 I-Cache

There are not many instructions in the loops in our program. A large I-Cache is not necessary to store the instructions of each loop while it is running, which means that a larger cache is not going to be a big enough performance increase to make the price increase worth it.

2.2.3 D-Cache

We had to hit a balance between price and performance here. Larger cache gives better execution times, since there is such a large amount of memory accesses, but the price increase is steeper than the performance increase.

2.3 Results

Hr lgger vi in alla testresultat och visar vilken som r bst etc.

After trying a lot of different configuration, this was the best one we could find:

References

[1] Monthly observations of precipitation 2002-2014, SMHI. `http://data.smhi.se/met/climate/time_series/month/vov_pdf/`, visited 2014-10-12.

[2] Climate Data and Data Related Products, WMO. `http://www.wmo.int/pages/themes/climate/climate_data_and_products.php`, visited 2014-10-11.

Configuration	
I-Cache size(words)	32
I-Cache associativity	1
I-Cache block size(words)	4
D-Cache size(words)	64
D-Cache associativity	2
D-Cache block size(words)	4
Processor frequency(MHz)	450
Memory access time(ns, first/other)	30 / 6
Write buffer Size(words)	4
Results	
Clock cycles	140 314
Execution time(μs)	311.809
Total component cost	3.37
Price x performance($\mu s C\$$)	1050.796

	Gothenburg						**Stockholm**					
Year	**Jan**	**Feb**	**Mar**	**Apr**	**May**	**Jun**	**Jan**	**Feb**	**Mar**	**Apr**	**May**	**Jun**
2002	153	95	48	27	71	116	65	42	28	8	34	94
2003	71	34	21	96	77	64	19	21	2	44	58	44
2004	65	38	61	36	27	83	37	29	25	13	47	108
2005	126	43	41	25	39	73	44	38	14	11	56	72
2006	66	41	52	94	71	42	10	29	28	28	46	32
2007	159	28	81	40	57	130	73	26	19	17	25	67
2008	166	89	108	33	11	67	69	23	41	28	17	56
2009	57	31	48	11	80	58	21	24	36	5	25	80
2010	30	48	46	32	31	53	26	29	22	23	31	33
2011	72	65	40	34	67	58	33	38	11	10	34	57
2012	88	55	9	75	74	95	42	45	18	62	20	160
2013	44	17	3	46	73	139	29	30	3	27	17	46
2014	66	107	32	47	81	52	39	35	43	25	35	28
Average	89	53	45	46	58	79	39	31	22	23	34	67
Normal value (1961-1990)	61	40	49	41	49	59	39	27	26	30	30	45

Year	**Jul**	**Aug**	**Sep**	**Oct**	**Nov**	**Dec**	**Jul**	**Aug**	**Sep**	**Oct**	**Nov**	**Dec**
2002	101	35	17	86	82	16	114	4	13	45	32	9
2003	143	32	25	101	76	119	75	53	20	32	43	78
2004	98	128	100	115	82	76	55	49	39	45	47	40
2005	107	83	60	75	93	53	122	65	11	28	44	54
2006	78	190	55	202	179	193	33	146	23	98	43	34
2007	129	56	121	47	82	108	51	17	74	39	48	45
2008	57	167	81	142	100	37	42	154	27	64	63	50
2009	170	79	56	84	128	53	89	54	32	68	52	45
2010	159	118	74	130	80	48	74	105	73	34	58	45
2011	121	156	129	107	39	146	13	90	60	44	14	75
2012	97	88	146	146	92	78	61	121	58	72	56	66
2013	51	49	56	106	72	154	43	55	39	58	61	51
2014	39	142	36				44	115	83			
Average	104	102	74	103	85	83	63	79	42	48	43	46
Normal value (1961-1990)	68	75	80	83	82	72	72	66	55	50	53	46

Table 1: Precipitation, mm/month

[3] Student t-Value Calculator, Dr. Daniel Soper, 2014.
`http://www.danielsoper.com/statcalc3/calc.aspx?id=10`, visited 2014-10-12.