Report

Home Assignment, DST15_035

Dennis Bennhage & Hampus Lidin

May 17th, 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 for the Gauss elimination 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 section.

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 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 Data Cache

With the D-Cache, we had to hit a balance between price and performance. 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. Therefore we had to find a point when the price would drain the possible performance gain that we would get. The main cost for the caches are prices for larger ones, so we had to find other ways to improve performance than just having a large cache. Associativity is a great hardware implementation that makes the miss rate drop down greatly, but unfortunately at the cost of slower CPU clock speed. Nevertheless, a 2-way associativity of the D-Cache has been proven to be the best configuration. As for block sizes, we would want to keep as much recent data in the cache for as long as possible, while consulting the main memory as little as possible. In other words, we want to find a good balance between the *temporal locality and spatial locality* usage.

2.2.2 Instruction Cache

The I-Cache only has to be as big as the largest loop in the program, or the longest streak of different instructions. Since our program managed to compress the largest loop down to under 32 instructions, a 32-word I-Cache will suffice. A larger cache would be pointless, since price would go up and performance in terms of CPU clock frequency would go down. Additionally, a direct-mapped associativity is the best choice, since we have already such a high hit rate, and a higher associativity would also slow down the clock frequency.

Finally, we wanted the block size of the cache to be as large as possible. Large blocks exploit the spatial locality of the memory and end up with fewer cache misses. However, our code needed to change in order to adopt to the large block size. Consider this, if we are in the first iteration of the outermost loop of our program, when we would get near the end, we might store some instructions outside the loop, that are only necessary at the end of the program. This would mean that the *Least Recently Used* (LRU) replacement policy of the cache would replace the early instructions of the loop with the instructions outside. Then each time the program gets to the end, we would get a read miss, and we'd start over again. This is not good, since it will generate unnecessary cache misses. The solution would either be to change to another block size, or to pad the program with NOP-instructions, to synchronize the cache writes better. That's what we did, and we ended up with fewer cache misses, for the same price of a same size cache with smaller blocks.

2.2.3 Memory

The main memory will have all the instructions and data stored for the program to access during its execution time. Since the program is calculating a matrix with hundreds of elements, we'd want a fast memory. The entire 24x24 matrix is stored in the main memory, so we have to read the memory every time we need a new value from the matrix. The same way we have to write back to the 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.

The memory can also support a write buffer, which is beneficial when writing a lot of data in a short period of time. However, we can't max it out with a giant buffer; we need to have a good flow of data. This is due to the fact that first time accesses in one write session is longer than subsequent accesses. For this reason we want to have the buffer busy almost all the time, i.e. increase the *utilization* as much as possible, so that the shorter access time is being exploited. The sweet spot for our program landed at a size of 4 words, presumably because of the data cache block size being 4 words, where most of the writing occurs. Each time a write miss occurs in the D-Cache, 4 subsequent data words in memory will be written back.

2.3 Results

The final form of the system consist of one CPU, one main memory and two caches. The I-Cache has a size of 32 words, with 8-word blocks and a direct-mapped associativity. The D-Cache is configured with a larger 64 word data space, with the smaller 4-word blocks and a 2-way associativity. Both caches implement the LRU replacement policy. The main memory has a first access time of 30 ns, with subsequent accesses of 6 ns. The memory also has a write buffer of 4 words, to accommodate for the frequent data writes. All specifications are listed in Table 2.3.

Configuration	
I-Cache size	32 words
I-Cache associativity	Direct-mapped
I-Cache block size	8 words
D-Cache size	64 words
D-Cache associativity	2-way
D-Cache block size	4 words
Processor frequency	450 MHz
Memory access time (first / others)	30 ns / 6 ns
Write buffer Size	4 words
Results	
Clock cycles	132 100
Execution time	293.556 μs
Total component cost	3.37 $C\$$
Price $\times$ performance	989.282 $\mu s C\$$

Table 1: A compilation of the system's specifications.