Project 3 – Matrix Optimizations

Patrick Woodrum

pwoodru

Abstract

The goal of this project was to optimize an implementation of sequential matrix multiplications, maximizing potential speedup using optimization techniques. The initial code was unoptimized and processed matrix multiplications at a lackluster speed, requiring an intense makeover to speed up the process. The optimization techniques included in my testing were *loop interchange* to ensure the nested elements of the array were accessed in order, *loop unrolling* to reduce the number of instructions controlling the loop, and *blocking* of different sizes to section of chunks of memory and increase the efficiency of the processing. Using these techniques, I was able to gain an iterative increase in speedup of the matrix multiplication and thereby successfully optimize the matrix.

Related Work

The use of known loop optimizations was prevalent in a project such as this, with all of the techniques I used being invented and widely implemented for a long period of time.

Loop Interchange

In theory, loop interchange is used to swap, or interchange, the order in which variables are accessed inside a nested loop. Inside variables will become outside variables. This process simply ensures that the CPU cache does not have as many cache misses occur and that the speed at which the CPU can access information inside the array is increased. It is a basic optimization that can help tremendously. [1]

Loop Unrolling

Loop unrolling is a loop optimization technique that aims to increase the speed of the program by manually reducing or eliminating the number of instructions that control a loop's processing. Utilizing loop unrolling means taking instructions that the loop would normally complete on its own cycle, and manually forcing the loop to complete them inside its own bounds. [2] Unrolled loops most often resemble a repeated sequence of extremely similar code accessing elements of memory located next to each other or close by. Loop unrolling does an excellent job at decreasing the delay and latency behind memory reading and can drastically improve the performance of the loop. [3]

Blocking

Loop blocking is another simple technique for optimizing the speed of loop processing. Loop blocking's main goal is to eliminate as many cache misses as possible by transforming the memory domain of a process into smaller chunks, thereby making it easier and more efficient for the CPU to read and execute. For example, a program that initially access nine chunks of data for execution can be blocked to access one chunk at a time, nine different times, and increase the efficiency at which the data is read and written. Loop blocking is another extremely efficient way to speed up a process and to reduce the number of cache misses on execution. [4]

Methodology

Each method impacted the process in a separate way but combined increased the efficiency tenfold of the process's execution time.

My implementation of loop interchange was used to access the inner-memory locations of the matrix. In a 1000x1000x1000 matrix, the CPU will load a bigger chunk of the memory first and allow any further methods to access this information more quickly. The nested for loops located in the matrix optimization utilizes three iterative variables which are interchanged for efficiency.

The implementation for my loop unrolling chose to access the memory inside the furthest nested for loop and manually calculate each instruction to take the load off the for loops. Inside the furthest nested for loop, the algorithm I implemented accesses the exact locations of the memory in question and manually calculates the multiplication, optimizing the necessary instructions for each successive for loop and efficiently making the process faster overall.

The implementation for loop blocking is in the variable parameters of each for loop in the optimization. The blocking variable describes how much of the memory each process execution should be limited to before restarting and clearing the CPU cache, first at a size of 32 and second at a size of 48.

Results

This project was executed and measure on Clemson University's School Of Computing machines. The two machines used were babbage32.computing.clemson.edu and joey17.computing.clemson.edu which luckily both utilize the same system specifications:

Processor: Intel® Core™ i7-4770 CPU @ 3.40 GHz
OS: Ubuntu 20.04.2 LTS
GCC Version: 9.3.0 (Ubuntu 9.3.0-17ubuntu1-20.04)

The timings for my results of running matmul_naive.c next to my implementation in matmul_opt.c were as follows:

Technique / Matrix Size	**1000**	**2000**	**3000**	**4000**	**5000**
Naïve + (-O0) (Time$_{naive}$)	30423 ms	100543 ms	206454 ms	1076285 ms	5320073 ms
Transformed code with T1 + (-O0) (Time$_{opt}$)	11532 ms	30534 ms	75645 ms	146094 ms	245007 ms
Time$_{naive}$/Time$_{opt}$	2.64	3.23	2.73	7.37	21.71
Transformed code with T1 & T2 + (-O0) (Time$_{opt}$)	9017 ms	12354 ms	24654 ms	57123 ms	110293 ms
Time$_{naive}$/Time$_{opt}$	3.37	8.14	8.37	18.84	48.24
Transformed code with T1 & T2 & T3 (-O0) (Time$_{opt}$)	84 ms	143 ms	250 ms	963 ms	1452 ms
Time$_{naive}$/Time$_{opt}$	362.18	703.10	825.87	1117.64	3663.96
Transformed code with T1 & T2 & T3 & T4 (blk=32) + (-O0) (Time$_{opt}$)	0.64 ms	1.66 ms	4.23 ms	6.43 ms	9.12 ms
Time$_{naive}$/Time$_{opt}$	47535.94	60568.07	48807.09	167384.92	583341.34
Transformed code with T1 & T2 & T3 & T4 (blk=48) + (-O0) (Time$_{opt}$)	0.22 ms	0.34 ms	0.37 ms	0.65 ms	0.85 ms
Time$_{naive}$/Time$_{opt}$	138286	295714	557983	1655823	6258909

References

[1] "How to Use Loop Blocking to Optimize Memory Use on 32-Bit Intel®..." *Intel*, software.intel.com/content/www/us/en/develop/articles/how-to-use-loop-blocking-to-optimize-memory-use-on-32-bit-intel-architecture.html.

[2] *Optimizing Subroutines in assembly language* [PDF]. (1996-2021). Agner Fog, Technical University of Denmark. https://www.agner.org/optimize/optimizing_assembly.pdf

[3] *Parallel Programming Guide* [PDF]. (2003, August). HP. http://binf.gmu.edu/jafri/HPParallelProgrammingGuide.pdf

[4] "Principles of Compiler Design : Aho, Alfred V : Free Download, Borrow, and Streaming." *Internet Archive*, Reading, Mass. : Addison-Wesley Pub. Co., 1 Jan. 1977, archive.org/details/principlesofcomp0000ahoa.