Geoffrey Corey
March 19, 2014
CS 480 Final

Loop Optimization[1]

Overview

In the article "A compiler optimization to reduce execution time of loop nest" the authors Oh-Young Kwon, Gi-Ho Park, and Tack-Don Han discuss altering loop tiling to not tile the innermost loop, reducing the number of instructions for loop control and increasing performance of matrix multiplication. the focus of this alteration is to maximize spatial reuse of the innermost loop to stay within the CPU's cacheline.

The article first goes over a formal definition of how the algorithm works and then explains what each of the type of vectors means in terms of loop nesting and spacial reuse. In the testing section of the article, the authors list what steps they took to run the code and how they went about comparing the results of the tests. In the experiments, the authors compared their loop tiling permutation that would take advantage of cache reuse and compared it to the conventional loop tiling algorithm for matrix multiplication.

The results of the tests were shown with 2 different CPU architectures: DEC 5000/25 and a SPARC 10. When only using the permutation algorithm, there was 1.44 and 2.61 fold computation time improvement on each architecture respectively. However, when applying the tiling technique, the execution time improvement is only 1.92 and 1.08 fold respectively.

Critique

This article is aimed at people who know more about loop nesting and know more about loop nesting and using loop tiling to decrease execution time of nested loops. The authors do a decent job of trying to explain what loop tiling is though and they do a even better job of explaining the shortcomings of the standard loop tiling algorithm.

In the algorithm explanation, the authors introduces the term "cacheline", but does not seem to define to the reader what a CPU chaceline is. The only thing the reader can concretely draw from this term is that it has something to do with the CPU's cache, but without the definition, it can obfuscate what the advantages of what spatial and temporal reuse in the algorithm can actually accomplish in speeding up the execution of the matrix multiplication example. It is not until later in Section 4 that the authors describe what a "cacheline" is, but even then they still do not relate it back to the term.

For the proof of the effectiveness of the algorithm, the authors does not show the average time of execution for X number of tests. They also do not detail how they generated the numbers to fill the matrices for the matrix multiplication. However, the authors did keep timings of the execution times as they increased the size of the arrays to track whether the observed execution time decrease had a plateau.

Overall, the authors do clearly show that their nested loop modification does increase the performance of matrix multiplication and they do relate the results of the timings back to the reason of taking advantage of CPU cache access time.

Function Inlining and Loop Unrolling[2]

Overview

In the article "Function Inlining and Loop Unrolling for Loop Acceleration in Reconfigurable Processors", the authors begin by introducing the platform for which they will be studying their chosen compiler optimization. They also introduce the optimizations that the platform can make as well as the limits to what

the compiler can do to optimize loops on the hardware. The goal of the writer is to exploit the compiler's loop parallelization techniques as well as exploit the pipelining of the platform to minimize the number of required to execute a for loop.

In this article, the authors propse that manually unrolling loops and inlining function in loops will increase the amount of optimization the compiler can do int terms of loop parallelization which becomes is turned off if while inside a loop, other functions are called.

With only just inlining of the Sobel Filter function, there was some increase in performance observed, 24%. The shortcoming of just the inlining were the inability to exploit the Course Grained Reconfigurable Array (CGRA) in the loop execution. However, when unrolling the nested loops they observed a 93% reduction in cycles when compared to no optimization and a 91% reduction in cycles when compare to just inlining functions. Another benefit to the loop unrolling is the ability to utilize efficiently the CGRA for loop parallelization.

For the Luma Deblock filter, the results of only using inlining show a cycle reduction of only 1%. Inlinig the code in the inner most loop of the deblocking filter allows mapping of the loop to the CGRA which reduces the number of cycles by 6%. When the level of loop unrolling is increased to a factor of 2, there is a reduction in the number of cycles by 11% to just inlining and unrolling the inner most loop. However, the gains of inlining and unrolling for this function are less because the inner most loop of the deblocking filter have too many operations for the CGRA's array size.

Critique

The authors of this article do a good job of trying to explain all the terminology introduced in this article as well as explaining what platform they will be using for the tests and how the platform works. They also do a good job at explaining the shortcomings of the way the current code is analyzed and compiled on the platform.

When it comes to explaining what the goal of this article is, the authors make it very clear what it is they are trying to accomplish and how they are going about accomplishing their goal. They even justify their relevance by comparing what they are expanding upon to the current optimization techniques used by compilers today. In order to make this as clear as possible, they even provide the original code side by side with the modified code to help the reader see what is changing and how it is changing.

The authors also provide quite a bit of detail on how compilers work with loop structures and how they go about optimizing the number of cycles needed for the execution, including Instruction Level Parallelism, Loop Level Parallelism, how to exploit these optimizations while writing code as well as noting the limitation to these optimizations by standard compilers.

In terms of the hardware platform, the authors do a good job of explaining how the hardware works and how to exploit the various parts of the hardware when optimizing loop operations and reducing the number of cycles required to execute the program.

for the results, the authors compare the various combinations function inlining and loop unrolling and even compare (when possible) the performance of various levels of loop unrolling. This method allows the read to gauge the the effects of the various levels of loop optimization on the number of cycles in the program. In the deblocking algorithm, the authors also note the lower than expected increase in cycle count by addressing the fact that the code could not exploit fully the hardware, but were still able to compare the differences in loop unrolling and function inlining to that of the original algorithm.

References

[1] Oh-Young Kwon, Gi-Ho Park, and Tack-Don Han. A compiler optimization to reduce execution time of loop nest. *SIGARCH Comput. Archit. News*, 24(1):6–11, March 1996.

[2] Narasinga Rao Miniskar, Pankaj Shailendra Gode, Soma Kohli, and Donghoon Yoo. Function inlining and loop unrolling for loop acceleration in reconfigurable processors. In *Proceedings of the 2012 International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, CASES '12, pages 101–110, New York, NY, USA, 2012. ACM.