My Implementation Is Faster Than Yours

Zack Tillotson
Department of Electrical Engineering and Computer Science
Colorado School of Mines

Abstract

In this demonstration report it is shown that deliberate instruction level utilization of advanced computer architecture knowledge can lead to significant improvement in algorithm execution speed. The test bed computer system and algorithm is first introduced. Next an alternative approach, compiler level optimization, is shown to gain only modest speedup results. A series of instruction level changes to the given test algorithm are made which achieve significant speedup. Knowledge of advanced computer architecture is therefore shown to be relevant even in the modern computing environments.

Introduction

Modern computers are extremely powerful but also increasingly difficult to take advantage of, from a programmers point of view. Despite the fact that the hardware has grown exponentially fast in terms of processor speed and memory size, modern computational needs have also grown at least as fast.

There have been many efforts in the compiler community to implement this need optimization. These efforts have achieved some gains, but it will be shown that modern complex architectures require more logic than is available to the compiler.

A common scientific computing task is matrix multiplication. In this two large matrices are multiplied together to produce another large matrix. The basic code to accomplish this task is given in Figure 1.

Figure 1: Square Matrix Multiplication

```
for i = 1 to S
   for k = 1 to S
      for j = 1 to S
         C[j,i] = C[j,i] + B[i,k] * A[k, j]
```

This code is obviously very simple, for each location in the results matrix C we iterate through each row of matrix A and column of matrix B, summing their values. In our work these three matrices A, B, and C are square matrices of size varying from 500 by 500 to 5000 by 5000. In general these matrices do not need to be square, but it makes for easier analysis without loss of generality. The largest size matrices, 5000 by 5000 has 25,000,000 elements. If each element is stored as a double precision floating point number using the standard 8 bytes, then each matrix uses nearly 200 megabytes of memory, approaching the limit of what can be stored in main memory of the computer. For simplicity and ease of analysis our task is run in a single threaded manner.

Computing Environment

The target machines for these optimization techniques are given in Figure 2.

Figure 2: Machine Specifications

Processor	
Type	i7-2600
Number of Cores	4
Clock Rate	3.4GHz
Core Flop Per Cycle	4
Core Speed	13.6 Gflops
Memory	
L1 Cache	64 KB
L2 Cache	256 KB
Main Memory	8GB
Software	
Program	C++
Compiler	gcc 4.4.3

The processor is the Intel i7-2600. Each core runs normally at 3.6 GHz, though a core can be dynamically over clocked to 3.8 GHz in situations of intensive CPU utilization. Each i7 core is capable of 4 floating point operations per second (flops), therefore the maximum theoretical throughput per core is 13.6 Gigaflops. This 13.6 Gflops will be our benchmark to compare real world results against.

As will be shown, the size of each memory cache in this processor will have a large impact on the overall performance. The i7 has a 32 KB L1 data cache and a separate 32 KB L1 instruction cache. Above that is a 256KB unified L2 cache and there is no L3 cache below the main memory bus.

Our program is compiled with the Gnu C++ compiler gcc 4.4.3, which supports many optimization techniques as seen in the next section.

Compiler Optimization

Many optimization steps can be taken by the compiler. gcc supports the use of several levels of optimization as specified with the “-O” flag. Each level turns on a set of options depending on the constraints of that level. The default optimization level is -O0, no optimization. Each statement of the original code is kept atomic. This allows for ease of debugging, as the machine code for any line in the original code can be located, stopped at, modified, and the continued past.

The first level of true optimization is -O1. At this level optimization techniques are turned on as long as they do not increase compile times significantly. Some of the individual optimization options that are turned on at this level are “defer-pop”, “omit-frame-pointer”, and “guess-branch-probability”. defer-pop is an option used when making multiple function calls in succession. Normally the compiler will pop off the method parameters from the stack as soon as the function returns, in cases where several function calls are made in a row this option allows the compiler to wait and pop them all of at once. omit-frame-pointer is an option to not save the frame pointer in a register for methods that don’t need it. This frame pointer is most often used in debugging, so turning this option on removes the ability to debug these methods. guess-branch-probability is an optimization where the compiler uses heuristics to predict when a branch will happen.

The second level of optimization, -O2, turns on options which might require longer to run during compilation, but which won’t normally increase the binary executable size. Some of the optimization techniques enabled at this level are “align-loops”, “strict-overflow”, “gcse”, and “cse-follow-jumps”. Align loops is a technique where loop boundaries are forced to align with powers of two. For example if the loop boundary was 10 bytes, it would be forced up to 16 bytes. This allows for the prefetcher to realize that the

loop is ending more quickly, without forcing a instruction fetch miss when the branch occurs. strict-overflow allows the compiler to declare arithmetic overflow operations as undefined. For example the statement ‘i + 10 < i’ is always false when arithmetic overflow is undefined. gcse stands for Global Common Sub Expression, when gcse is turned on the compiler goes over the entire code and replaces common sub expressions with their computed values. cse-follow-jumps is a modification to gcse which turns on the option to have the compiler follow jumps during its sub expression elimination.

The final basic level of optimization, -O3, does away with the executable file size constraint and turns on a new set of options. These options include “inline-functions” and “unswitch-loops”. The inline-functions options tells the compiler to consider all functions for inlining, instead of those explicitly declared inline. The compiler uses a heuristic to decide if a function would be worth using inline. unswitch-loops is an option to move if statements with loop invariant conditions outside of the loop with duplicate loops on either branch.

One final option was tested, “unroll-loops”. This option tells the compiler to attempt to unroll loops at compile time. This option is not included in any of the three basic optimization levels because it increases the size of the executable and often does not increase the execution speed. In order to achieve speed gains with this option very specific memory management techniques must be used, as will be shown in the next section.

The speedup achieved by these compiler optimization techniques are good, but as will be shown in the next section, thoughtful programming with an eye on the computers architecture can produce even better results.

Programmatic Optimization

Step 1 - L1 Cache

Many of the bottlenecks associated with this matrix multiplication algorithm are directly related to specific pieces of the computer architecture. The first bottleneck is the L1 cache. As the base program executes it loops over the matrices, loading them from memory. Each memory access will pull a page of the memory into the L1 cache. Because of the way the memory accesses are happening this cache has an extremely high miss rate. Using a technique called “2D tiling” we are better able to take advantage of this cache.

In 2D tiling the memory is only accessed in small chunks which can be stored in the L1 cache. Instead of looping through an entire row or column of A and B, we loop through small sections of them at a time. The same work is being done, but using small sub matrices of each large matrix which better correspond to the size of the L1 cache.

Step 2 - TLB

Some of the gains of 2D tiling are offset because the smaller loops cause a higher rate of Transition Lookaside Buffer (TLB) misses. Because the outside loop i has such a large amount of work done per iteration the i TLB entry has a higher chance of being flushed. By changing the 2D tiling from step 1 to a 3D tiling in step 2 we reduce the number of TLB misses which still gaining from the higher cache hit rates of the previous step.

Step 3 - Store Queue

The store queue is used when a write operation is sent to memory. Because of the memory intensive nature of the matrix

multiplication algorithm there are many store requests being sent. When a store is still queued and that location is read from the read operation fails and must wait and be retried. This is called a replay trap. The occurrence of replay traps in our algorithm in high because of how often we need to store values in C. To fix this the algorithm is modified so that the innermost loop is out, then the value being stored is saved in a temporary value during the new innermost loop. This allows for one store command to C per innermost loop, greatly reducing the number of number of store commands and reducing replay traps.

Step 4 - Instruction Level Parallelism

The i7-2600 processor has many 64 bit (double size) floating point registers and the matrix multiplication algorithm only uses a few. A technique called loop unrolling is used to take advantage of more of them. In loop unrolling a loop can be "flattened" by manually storing several iterations worth of variables in one iteration, and then jumping over all of the iterations done. In our case we are unrolling the innermost loop eight times.

Step 5 - L0 Cache

I call the way we use the registers in this step the L0 cache. By unrolling some of the other, higher loops we can save values into registers. By keeping these values in registers rather than having to do memory lookups we further decrease the amount of time we are memory bound.

Results

For comparisons sake the results of these tests are given in Figure 3 and Figure 4. Figure 3 shows the time taken to solve the matrix multiplication algorithm on a 1500 x 1500 matrix for each program version.

Figure 3: Time Taken By Program Version For Size 1500 Matrix

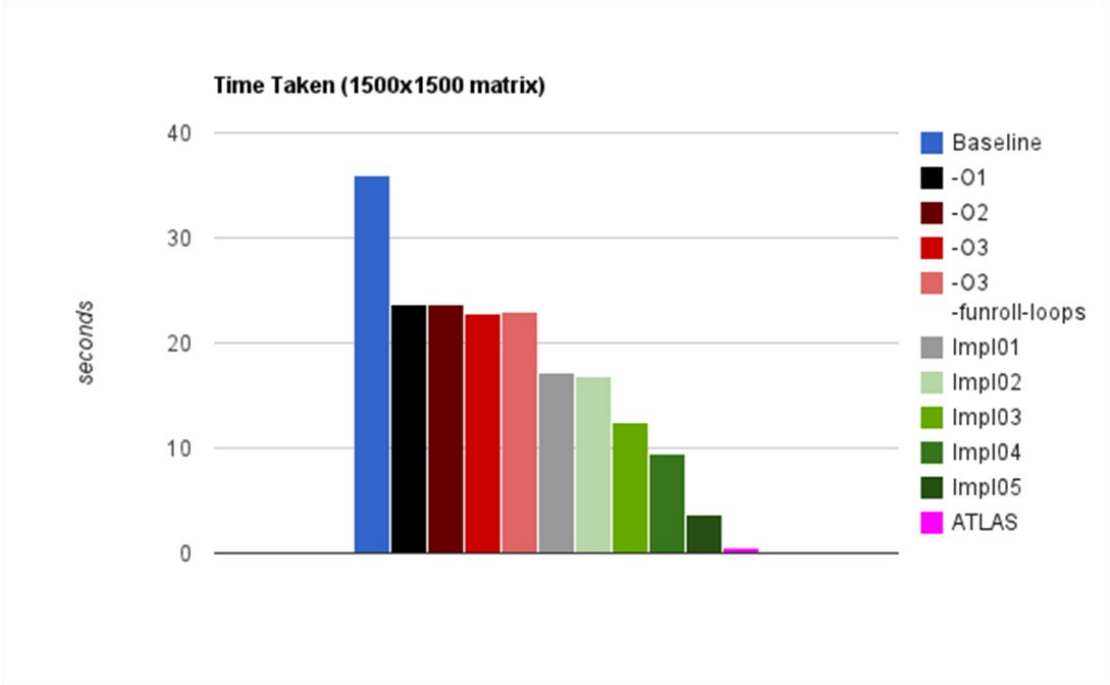
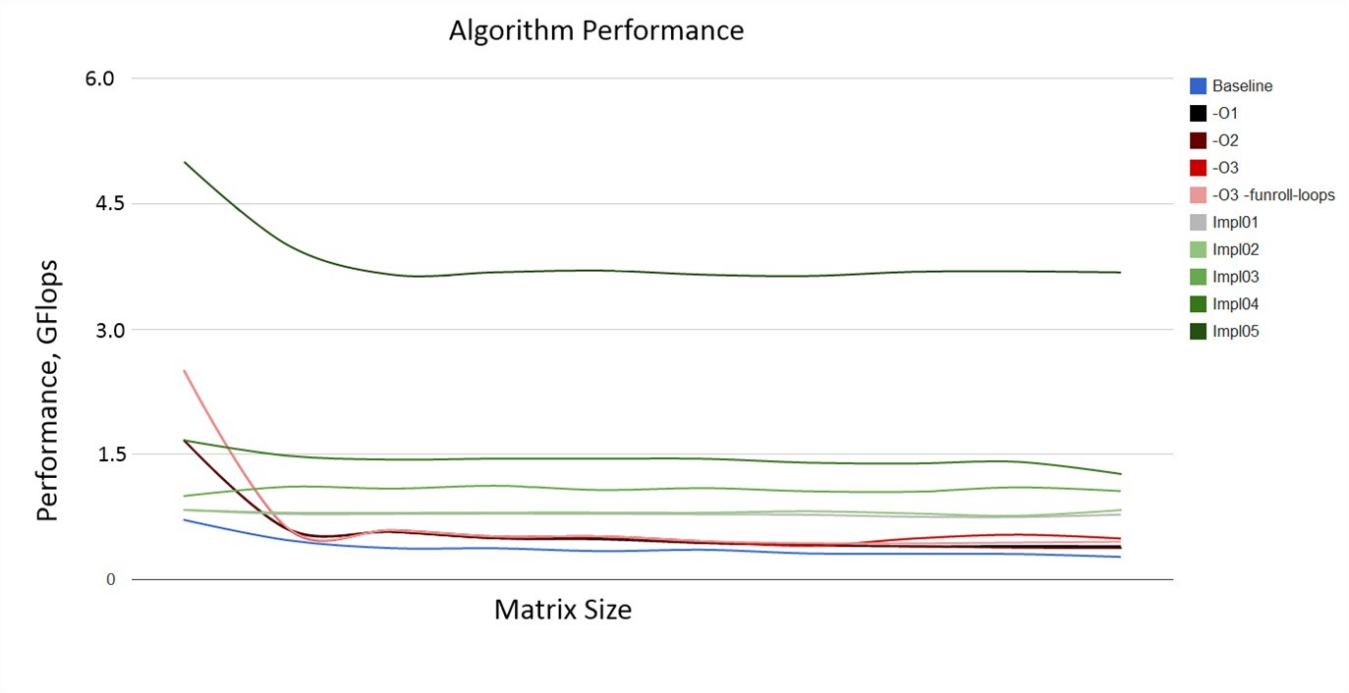


Figure 4 shows the performance in Gflops of the different versions of the program. Compared to the baseline, compiler level optimization techniques (shown in shades of red) only offer a slight improvement. The instruction level changes (show in shades of green), however, show ever more significant improvement as more architectural elements are consider. The final step 5 approach cumulates in an impressive 13.54 speed up over the baseline

Figure 4: Performance By Program Version



Conclusion

As has been demonstrated in this series of experiments, while modern computer architectures are powerful, the ability to fully utilize this power is not a given. Even utilizing advanced compiler optimization techniques we were unable to approach the theoretical maximum speeds of the given tested computing system. By deliberately modifying a simple algorithm to take advantage of the available resources and also reduce disadvantageous utilization of the same resources we were able to see a significant improvement in execution speed, even approaching a significant fraction of the theoretical maximum throughput of the given processor.

References

1. D. Parello, O. Temam, and J.-M. Verdun, On Increasing Architecture Awareness in Program Optimizations to Bridge the Gap between Peak and Sustained Processor Performance -
Matrix-Multiply Revisited, SuperComputing 2002, Baltimore, Maryland (November 2002).