Assembly Language in Modern Technologies still Faster than HLL: Myth or Reality

Micic, Milos; Etinski, Maja; and Milutinovic, Veljko

Abstract— ***When technologies change, realities change, as well. Many believe that advances in the technology of optimizing compilers cause the programming in a high-level language to create a faster code (compared to assembly language programming), due to the fact that individual programmers can never optimize their hand-written code as successfully as the optimizing compilers do it. This paper demonstrates that it is not always the case, and clarifies the issue for the case of a selected algorithm of importance for a number of modern applications. Architectural changes demonstrate how powerful can be the usage of advanced assembly instructions [1].***

1. Introduction

1.1 Development and Variety of C Compiler Optimizers

THE development of compilers is focused on optimizations. There are many different goals. The first of them is to speed up execution of generated code. The diminishing of code size is also one of the main goals. So, there are many compiler options that determine whether optimization is needed and which kind of it should be used. Compiling time with optimization takes more than without it. That issue isn't relevant for this paper. Our main aim is to observe the time needed for the execution. For that purpose we choose the newest versions, MS Visual Studio 2005 and older version of GCC compiler – version 3.4.4. These versions were chosen in order to see achievements in the area of code optimization.

1.2 Concrete Platform

Compilers also tend to use advantages of a concrete platform so the generated code could get the maximum from it. Our work is based on Intel Pentium III processor. This processor uses 16 KB of data L1 cache, 16 KB instruction L1 cache and 256 KB of L2 cache. L1 cache is synchronous write-back. Pentium III supports following technologies: MMX (Multimedia extensions) adds instructions for fast execution of typical primitives in multimedia processing such as vector manipulations, matrix manipulations and big block moves; SSE (Streaming SIMD Extensions) expands the SIMD (Single Instruction, Multiple Data) execution model introduced with Intel MMX technology by providing a new set of 128-bit registers and the ability to perform SIMD operations on packed single precision floating point values.

1.3 Fast Fourier Transform

In this work we used an FFT (Fast Fourier Transform) algorithm as an example. It is a fast algorithm (in the sense of algorithm complexity) for computing discrete Fourier transform dividing the problem into two of smaller dimensions.

2. Problem Statement

Because of long lasting efforts to develop better optimizers the speed of generated code becomes better. That arise a question about using the assembly code in order to gain on execution speed. Assembly programmers could be expensive resource comparing to the achievement of code performance that they make. Code written in some HLL could be faster than the one written in assembly language. According to this, some aspects of assembly programming must be carefully considered. Sometimes compilers don't use hardware specific instructions that trained programmer is able to implement. This could be explained by the fact that compilers don't have the capability for intelligent abstract methodologies that could solve the problem in global manner. In such situation inline assembly code could be a good solution. We should observe that on the FFT algorithm example written in C high level language.

3. Existing Techniques

There are different ways for faster implementation of some problem. One way is to find better algorithm. In the case of FFT there are different algorithms developed in last few decades. A lot of effort was given to their comparison and analyzes. The other aspect is implementation of a concrete algorithm.

Manuscript received Jun 1, 2006.

Micic Milos and Milutinovic Veljko are with Faculty of Electrical Engineering, University of Belgrade, Serbia.

Etinski Maja is with the Faculty of Mathematics, University of Belgrade, Serbia.

Performances of implementation depend on hardware and software used for development.

3.1 Categories of Compilers Optimization

Techniques in optimization can be divided into two groups depending on the scope of optimization: global and local. Local techniques are much more common and easier to develop. The main ideas for optimization are based on next tasks: avoiding redundancy, making less code, jump reducing, code locality, extracting information from code etc. They are all based on making a faster code, making less code or both using existing advantages of modern computer technologies such as pipeline and cache. GCC and MS Visual Studio 2005 compiler have both options for enabling different types of these optimizations.

3.2 DSP

The other way to speed up FFT implementation is to use a specific architecture. Digital signal processors (DSP) have important role in a great number of computationally intensive signal processing applications in fields such as communications. Many of today's DSP architectures employ a deep pipeline that supports high-frequency operation and a very long instruction word (VLIW) that enables the execution of multiple instructions in parallel. This kind of architecture is suitable for FFT usage.

4. PROPOSED ANALYSIS

4.1 Different Compilers

Using different compilers to compile the same code of an FFT algorithm we could compare execution time of programs that compilers made. As we mentioned, observed compilers are GCC 3.4.4 and Visual Studio 2005 C compiler. Usually default option is code generation without optimization. It is reasonable because an optimization takes time, makes debugging more difficult and very often isn't necessary. Optimizer efficiency could be discussed if we compare execution speeds without optimization and with compiler optimization.

4.2 Modifications Using Assembler

We are also interested in a rate of improvement gained by the usage of an assembly inline code. Testing compiler optimized code and the new one with inline assembly will give as a result the difference between execution time of C code without and C code with inline assembly. The smaller difference means the better optimizer.

4.3 Possible Optimizations Using Assembler

There is a variety of possible optimization techniques in the context of inline assembly programming. Depending on algorithm structure different ones could be suitable. Sometimes it is possible to reduce size of a part of code in order to be possible putting it in instruction cache at once. The other technique is to gain more from pipeline mechanism changing order of jump instructions and by reducing their number. Of course the code semantics must remain the same. The inline assembly optimization used in this paper is based on arithmetic instructions that provide one kind of parallelism. The used set of instructions is SSE set of MMX instructions. It is used in the most inner loop so, small amount of gained time, multiplied by many times should significantly reduce the execution time.

4.4 Variations of Input Size

Input size variations are included. The size of data cache is limited so at one point data cache won't be enough large to store all program data. The input is generated randomly. The time for the input generation isn't included in execution time. The input presents some values of function whose discrete Fourier transform should be calculated. Sizes of inputs that will be used are from the range: 1KB - 16MB. Value of arithmetical mean carried out from many measurements for the same input size will be presented.

5. EXPERIMENTAL DETAILS

Figure 1 show that GCC compiler is better for compiling pure C code without inline assembler. This achievement is not as significant as the next one shown on figure 2 when using inline assembly.

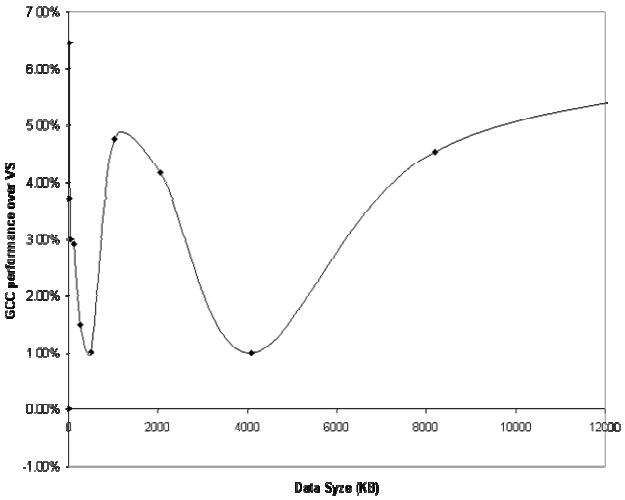

Figure 1: GCC performance over MS Visual Studio 2005 compiler using speed optimization

On the figure 2 we have results for execution times for both GCC and VS compilers before and after inline assembly implementation. Input data sizes are small enough to fit into cache. Figure 3 shows us the same characteristics but for input data that can not fit into cache.

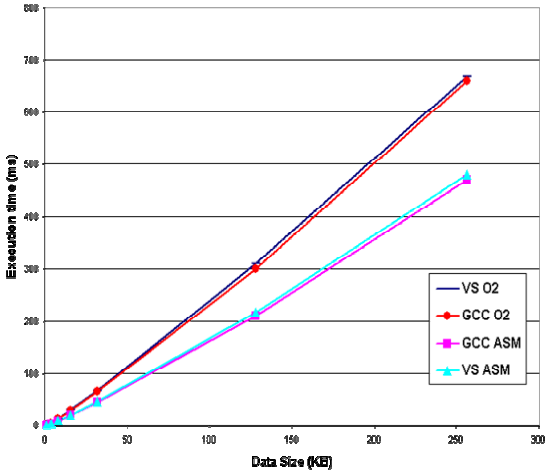

Figure 2: Execution time analysis for cache hit input data sizes

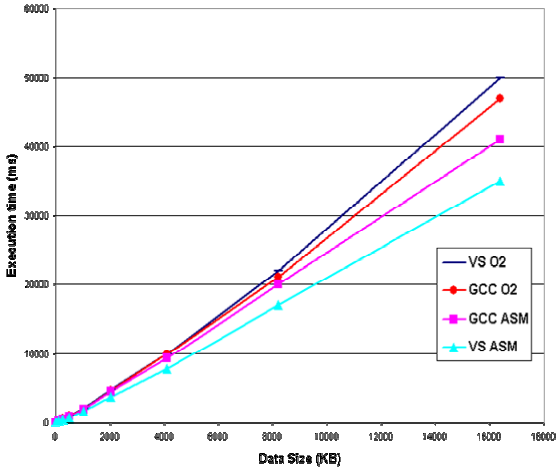

Figure 3: Execution time analysis for cache miss input data sizes

It can be seen that performances are better when using inline assembly as shown on figure 4 and 5. Performance achievement is shown comparing C code without inline assembly. 100 % performance is for pure C code.

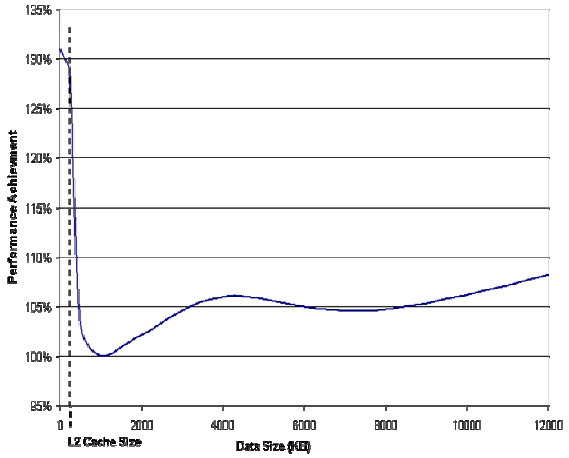

Figure 4: GCC 3.4.4 performance achievement using inline assembly code

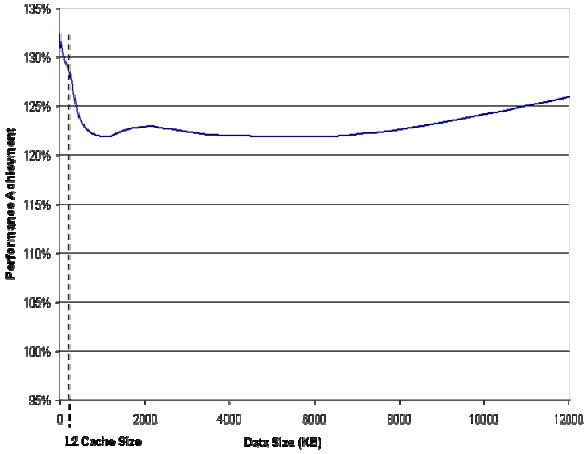

Figure 5: MS Visual Studio 2005 performance achievement using inline assembly code

It can be seen that MS VS 2005 achieves better performance for cache miss data sizes. The maximum of achieved performance is 30 % above the pure C code performance in both cases.

6. Conclusion

6.1 FFT Algorithm Based Conclusion

Due to the nature of FFT algorithm it was possible to accomplish its performance by replacing a part of code from the inner loop. This replacement was based on set of instructions which are independent from each other. That is suitable for parallel implementation with SSE Pentium set of instructions.

6.2 Discussing the Myth

The important question about assembly language today is whether there is a need for its usage at all. In this paper we have showed that there are some situations where assembly inline code could make obvious improvement despite the great set of various compiler optimizers' techniques.

The code replacement that we made isn't possible without reasoning. Simulation of reasoning could be a good development direction for compilers. For now, inline assembly is still reality for achieving greater performance for specific problems on specific platforms.

Appendix with Math Related Details

Evaluating a discrete Fourier transform would take O (n^2) arithmetical operations. Fast Fourier Transform (FFT) is an algorithm which reduces the number of computations from O (n^2) to O (n log n). The idea is to break problem into two smaller using the Danielson Lanczo's lemma (1942.):

$$\sum_{n=0}^{N-1} a_n e^{-2\pi i n k / N} = \sum_{n=0}^{N/2-1} a_{2n} e^{-2\pi i (2n) k / N} + \sum_{n=0}^{N-1} a_{2n} e^{-2\pi i (2n+1) k / N}$$

$$= \sum_{n=0}^{N/2-1} a_n^{even} e^{-2\pi i n k / (N/2)} + e^{-2\pi i k / N} \sum_{n=0}^{N/2-1} a_n^{odd} e^{-2\pi i n k / (N/2)}$$

This is break up of transform of dimension n into transforms of dimension n/2 and that is radix-2 case. It is possible to divide problem of size n=n1n2 into smaller DFTs of sizes n1 and n2 what is called mixed-radix case.

FFT was popularized by a publication of J. W. Cooley and J. W. Tukey in 1965. But similar idea was known to Carl Friedrich Gauss what was discovered later.

There are many FFT algorithms. Some of them are PFA, Rader-Brenner algorithm and Bruun's algorithm.

REFERENCES

[1] McNeley, K., Milutinovic, V., "Emulating a CISC with a RISC," *IEEE Micro*, Vol. 7, No. 1, February 1987, pp. 60-72.

[2] Chung, C., Xiangdong, F., "Achieving Better Code Optimization in DSP Design," 30.12.2005. www.us.design-reuse.com/articles/article8123.html.

[3] Hiroyuki, S., Yutaka, M., "A Type System for Optimization Verifying Compilers," 30.12.2005. www.is.titech.ac.jp/ppl2004/proceedings/p007.pdf.

[4] Milutinović, V., "Surviving The Design of Microprocessor and Multimicroprocessor Systems," *A Wiley Interscience*, New York, USA, 2000.

[5] Rokić, J., "Implementation of the fastest FFT algorithm," Graduate paper, *Faculty of Electrical Engineering, Belgrade*, Serbia, 2004.

[6] "IA-32 Intel Architecture Software Developer's Manual," 30.12.2005, www.intel.com/design/pentium4/manuals/index_new.htm .

[7] Ganapathiraju, A., et. al., "Contemporary View of FFT Algorithms," 30.12.2005, ardra.hpcl.cis.uab.edu/publications/documents/PDSP/ganapath_etal_1998-iasted.pdf.

[8] Balducci, M., et. al., "Benchmarking of FFT Algorithms," *Proceedings of the IEEE*, Blacksburg, VA, USA, April 1997.

[9] Cipra, B., A., "Faster than a Speeding Algorithm," 30.12.2005, www.siam.org/siamnews/11-99/speed.pdf.

[10] Allen, R., Kennedy, K., "Optimizing Compilers for Modern Architectures," *Morgan Kaufmann Publishers Inc*, San Francisco, CA, USA, 2001.

[11] Benjamin, C., "Types and Programming Languages the Next Generation," 30.12.2005, http://www.cis.upenn.edu/~bcpierce/papers/tng-lics2003-slides.pdf.