Exploiting the Power of Modern Processors and Architecture-Specific Compiling Techniques in Numerical Algorithms

Nino Arsov*, Vladimir Zdraveski† and Boro Jakimovski‡
Faculty of Computer Science and Engineering
Ss. Cyril and Methodius University
Skopje, Republic of Macedonia
Email: *arsov.nino@students.finki.ukim.mk, †vladimir.zdraveski@finki.ukim.mk, ‡boro.jakimovski@finki.ukim.mk

***Abstract*—Today's modern processors have developed rapidly since their early 1980s microprocessor days. Multiple cores, out-of-order execution and speculation, multithreading and support for single-instruction multiple data (SIMD) belong to the ordinary characteristics of most, if not every modern processor. "State of the art" numerical algorithms for problems in linear algebra have been developed to take advantage of specific matrix properties, but have they taken modern processors' capabilities into account? In this paper we examine two different algorithms to solve a system of linear equations - Gauss-Jordan elimination and Doolittle's LU factorization algorithms. The latter requires roughly 2.5 times less operations, but modern compilers offer effortless optimization techniques to exploit processor-specific characteristics. The experiments show that a theoretically slower algorithm by a factor of approximately 2.5 can be optimized solely by compiler techniques, so that it outperforms its theoretically faster counterpart.**

***Keywords*—computer architecture, compiler, processor scheduling, numerical algorithm, performance analysis**

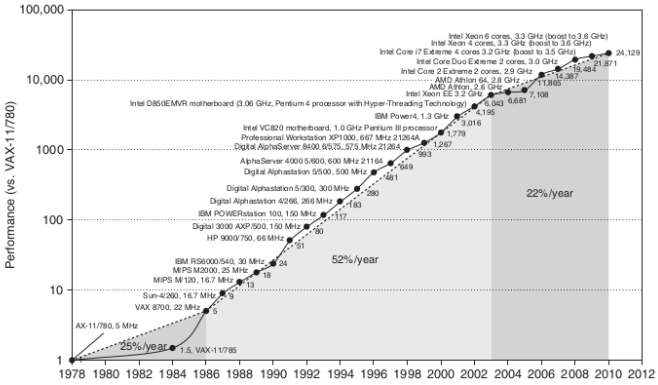

Fig. 1: A timeline showing the performance improvements of different processors in the past four decades [1, Figure 1.1]

I. Introduction

Computer architecture principles have been the driving force of performance leaps in modern processor technologies and modern computing since the late 1970s. Since then, many computer science researchers and authors have proposed a vast number of algorithms, from which the majority are based on their theoretical complexity and its improvements. Obviously, there have been significant advances in these areas. On the other hand, the gap between memory and processor performance, particularly their speed, has contributed to the advancement of fast cache memory and DRAM technologies in order to narrow the gap, but the overall processor performance is what comes to the fronts of modern computing. The performance timeline shown in Figure 1 gives a very clear view of what has been achieved up to 2012 with the latest Intel Core i7 and Xeon technologies. It can be seen that modern processors have a performance boost of at least 10,000 times more than their predecessors in the early 1980s. The trend of multicore processors has also contributed to this number by allowing processors to handle more work in parallel. When running multiple programs, it has been observed that dual-core processors perform about 1.5 times faster compared to a single-core chip [2]. Today, multicore processors are available to a wide range of electronic devices, ranging from servers to modern ultrabooks and smartphones, and the number of cores on a single chip goes up to eight or even ten cores. Processors capable to execute program instructions out of their original order, also known as out-of-order execution (OoOE) processors, have been more widely available on the market since the 1990s. This paradigm of instruction scheduling exploits instruction-level parallelism and is often implemented using Tomasulo's hardware scheme since IBM's 360/91 in 1996 [3]. Furthermore, hardware multithreading (MT), exploiting thread-level parallelism, facilitates the expansion of logical cores by adding additional hardware in each physical core to allow simultaneous execution of multiple threads. The most common hardware MT implementation is two threads per core, which results in a twofold increase of logical cores. The most efficient form of hardware MT is simultaneous multithreading (SMT), where it is possible to issue multiple instructions from different threads at each cycle. SMT provides more speedup than other MT techniques [4, Figure 4]. It is implemented by multiplicated OoOE hardware, and therefore it exploits both instruction-level and thread-level parallelism [5]. It effectively increases the overall processor efficiency and hides the increased memory latencies and stalls caused by interthread interference in the memory system and branch prediction hardware, respectively [4]. Multicore processors, OoOE and SMT are responsible for most leaps in processor performance in Figure 1.

More than fifteen years ago, professor David A. Patter-

son, a prominent author of computer architecture works, has stepped up with compelling arguments regarding the amount of effort put into development of algorithms and data structures with low theoretical time bounds both then and presumably nowadays [6]. The basic idea is that the authors of algorithms and data structures books, and programmers themselves, sometimes go too far in forging algorithms in terms of the required resources to implement them. This arguably refers to an extended amount of time required for the algorithms to be implemented correctly and all the difficulties that come along the way in order to achieve lower theoretical time bounds. For example, trade-offs between exactness and speed are used to develop theoretically faster algorithms that yield approximate, rather than exact results. The underlying argument is that the process of developing new algorithms seems to treat processors as if they were still stuck in their early 1980s microprocessor days [6]. In fact, this is certainly not the case today, when modern processors are much more powerful than before and it might be the case that straightforward implementations of theoretically faster algorithms, in practice, might run slower due to inadequate processor utilization and inadequate exploitation of the capabilities of modern compilers.

For instance, in most cases a programmer would simply choose the faster algorithm and implement it straightforwardly. But performance improvements like cache-friendly code or exploitation of the dedicated registers for SIMD operations, such as the 128-bit and 256-bit SIMD registers in recent typical Intel Core i7 chips that provide efficient vectorization are usually left out in an every-day development scenario. Moreover, compiler optimization techniques such as loop unrolling and generation of much shorter assembly code contribute to drastic performance boosts.

Our goal in this paper is to show that the power of modern processors and compilers can scale down the actual running time of an algorithm significantly below its theoretical complexity and examine whether cubic algorithms are really cubic in practice. The rest of this paper is organized as follows: Section II studies two different numerical algorithms and their theoretical complexities, Section III examines optimization techniques of slow cubic algorithms, Section IV presents our results and Section V gives a final conclusion of the work in this paper and suggestions for future work.

II. Case Study: Numerical Algorithms for Solving Systems of Linear Equations

The focus of this section is a typical scenario in many engineering applications - finding the solution of a system of N linear equations with N unknowns of the form

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_N &= b_1 \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_N &= b_2 \\ &\vdots \\ a_{N1}x_1 + a_{N1}x_2 + \cdots + a_{NN}x_N &= b_N, \end{aligned} \tag{1}$$

or simply, the system represented in in matrix form as below.

$$\mathbf{A}\mathbf{x} = \mathbf{b} \tag{2}$$

Using matrices to represent systems of equations is the usual fashion. Many different numerical methods have been developed for solving systems of equations, and in this paper we analyse two very well-known methods for this: Gauss-Jordan elimination and Doolittle's algorithm for LU factorization with forward elimination and backsubstitution. LU factorization is used as the standard method to solve systems of linear equations due to the numerical instability of the Gauss-Jordan method, but heuristic techniques such as partial pivoting help to overcome this issue and eliminate suspicions of instability [7], [8]. In the following two subsections, the complexity of both algorithms is expressed through the total number of required Arithmetic Logical Units (ALUs) operations including both integer and floating point comparisons, additions, subtractions, multiplications and divisions. The calculations are based on our C++ implementations in [9].

A. Gauss-Jordan Elimination

The Gauss-Jordan elimination method transforms the coefficient matrix $\mathbf{A}$ into reduced row-echelon form (RREF) with zeros below and above each leading non-zero element of each row. The solutions $\mathbf{x}$ are then easily obtained from the RREF of $\mathbf{A}$ by applying the same matrix transformations to the right-hand side vector $\mathbf{b}$. This procedures also applies to r right-hand side vectors, while the coefficient matrix $\mathbf{A}$ remains unchanged. The algorithms involves partial pivoting to ensure its numerical stability. Partial pivoting is done by searching for the row that contains the absolute maximum element in the column below the leading non-zero element of the current row. It is repeated N times throughout the elimination method. The total number of arithmetic operations t^r for r right-hand side vectors is given below:

$$\begin{aligned} t^r_{gj} &= \sum_{i=1}^{N}\Big[3(N-i) + 2 + 2N + 2r + \sum_{j=1}^{i}(3 + 4(N-j) + 4r) \\ &\quad + (N-i)(3 + 4N + 4r)\Big] \\ &= \sum_{i=1}^{N}\Big[4N^2 + 8N + 4rN + 2r + 2 - 2i^2 - 5i\Big] \\ &= 3.33N^3 + (4.5 + 4r)N^2 + (2r - 0.83)N. \end{aligned} \tag{3}$$

Thus, for $r = 1$, i.e. a single right-hand side, the calculation yields

$$t^1_{GJ} = 3.33N^3 + 8.5N^2 + 1.17N \tag{4}$$

arithmetic operations required to calculate the solution to a system of N linear equations with N unknowns, or approximately $3.33N^3 + O(N^2)$ operations.

B. LU Factorization by Doolittle's Algorithm

LU factorization is the standard choice of solving a system of linear equations. It represents a matrix $\mathbf{A}$ as a product of a lower-triangular and an upper-triangular matrices $\mathbf{L}$ and $\mathbf{U}$. Given the LU decomposition of the coefficient matrix $\mathbf{A} =$

$\mathbf{LU}$, the solution of the system $\mathbf{Ax} = \mathbf{b}$ can be obtained in two steps with a substitution, as shown in Equation 5.

$$\begin{aligned} \mathbf{LUx} &= \mathbf{b} \quad (\text{substitute } \mathbf{Ux} = \mathbf{y}) \\ \mathbf{Ly} &= \mathbf{b} \quad (\text{forward elimination for } \mathbf{y}) \\ \mathbf{Ux} &= \mathbf{y} \quad (\text{backsubstitution for } \mathbf{x}) \end{aligned} \tag{5}$$

The Doolittle algorithm to find the LU decomposition of a square matrix is considered to be more suitable than Gauss-Jordan elimination for solving a system with r different right-hand side vectors, because it remains unchanged and can be reused in each different forward elimination and backsubstitution [8], [10]. The total number of required operations for the LU factorization is

$$\begin{aligned} t_{lu}^r &= \sum_{i=1}^{N} \Big[2 + (N-i+1)(3+4i) + (N-i)(4+4i) \Big] \\ &= \sum_{i=1}^{N} \Big[7N + 8Ni + 8 + -8i^2 - 3i \Big] \\ &= 1.33N^3 + 5.5N^2 + 5.17N \end{aligned} \tag{6}$$

Forward elimination and backsubstitution to obtain a single solution vector both require $1+\sum_{i=1}^{N-1}(4i+4) = 2N^2+2N-3$ operations and a total of $4N^2 + 4N - 6$ operations together. The total number of required operations for this algorithm is the

$$t_{lu}^r = 1.33N^3 + (5.5 + 4r)N^2 + (5.17 + 4r)N - 6r, \tag{7}$$

and for $r = 1$ it comes down to

$$t_{lu}^1 = 1.33N^3 + 9.5N^2 + 9.17N - 6. \tag{8}$$

arithmetic operations to solve a system of N linear equations with N unknowns and is approximated at $1.33N^3 + O(N^2)$ operations in total.

Generally, Gauss-Jordan elimination requires 50% to 250% more operations in total, or as it follows from Equations 4 and 8, the LU factorization method is at roughly 2.5 times faster. This is obvious from the running time approximations in Big-O notation as well.

III. Optimization of The Gauss-Jordan Method

In the previous section, as a result of the complexity analysis of our C++ implementations of the two methods for solving a system of linear equations, it was shown that the Gauss-Jordan method is roughly 2.5 times slower than LU factorization. But the discussion in I brings forward the question: Can Gauss-Jordan actually run (much) faster than Doolittle's algorithm in practice?

For convenience and clarity, we assume that no changes to the C++ code are going to be made by hand, except for adding only a few *pragma* compiler directives to ignore dependencies and anti-dependencies between instructions within loops or between different loop cycles. The independence of instructions cannot be always detected immediately from the code (e.g. using the same variable across all loop cycles). Of course, if the code runs on a 1980s microprocessor, the palette of optimizations would be exceptionally narrowed, considering the limited extent of incorporated microprocessor technologies then. In such cases, the most advantageous choice of optimization would be to manually rewrite the code in order to decrease the number of arithmetic operations, or even implement a totally different algorithm for the same purpose. But, today's modern processors and modern compilers can do a lot more than earlier, without having to change a single line of code. The nature of the algorithm and the code itself provide much more opportunities than before.

For the purpose of the results in the following section, let us consider the specifications of the host Intel processor to execute the code and how can one take advantage of its features. The specifications are outlined in Table I. We use the latest 2015 version of the Intel C++ Compiler to compile our code. The processor supports OoOE and speculative execution by incorporation of Tomasulo's scheme and reorder buffers. The Gauss-Jordan and Doolittle's LU factorization methods can take advantage of this feature to achieve IPC greater than 1.0.

TABLE I: The specifications of the Intel Core i7 4500U

Feature	Value
Model	Intel Core i7 4500U
Frequency	1800 MHz
Turbo frequency	up to 3000 MHz
Data width	64 bit
Cores	2
Threads	4
L1 Cache	2 x 32KB 8-way set associative data cache 2 x 32KB 8-way set associative instruction cache
L2 Cache	2 x 256KB 8-way set associative cache
L3 Cache	4MB 16-way set associative cache
SIMD	SSE4.2 and AVX2

A. Exploiting the Power of the Processor

The Intel Core i7 4500U supports dynamic branch prediction, but this aspect is not analysed because the code does not involve any branches that change often, except for the loops, where special registers are used for that purpose. Thus, the penalty of branch misprediction is left out of our analysis.

We consider single-threaded execution on a single core, so all optimizations related to thread-level parallelism, such as SMT, are not examined in this paper. The reason for this is the different underlying nature of the algorithms; the two phases (factorization and substitution) of Doolittle's algorithm and Gauss-Jordan elimination are not equally efficient for parallel implementation, while the latter is more suitable for parallel architectures [11]. This observation applies to modern processor architectures as well, although algorithms using a similar, Doolittle-like reduction seem to be suitable for parallel implementation [12]. We use 64 bit floating point numbers with double precision for the coefficient matrix $\mathbf{A}$. The first optimization comes from the private L2 caches and the shared L3 cache. Since Gauss-Jordan deals with row transformations like swapping, multiplying and adding rows together it can make use of the cache since C++ uses row-major matrix storage. Since there are 256KB of L2 cache available, a system of order 180 could easily fit inside. The shared L3 cache of 4 MB can store a system of order up to roughly 600. For systems of higher order, single rows can safely fit in the L3

cache, since Gauss-Jordan elimination involves only row-wise matrix operations. Except for the partial pivoting repeated N times, i.e. only once before N^2 accesses to rows, the rest of the method consists of row operations. In contrast, Doolittle's LU factorization method accesses both rows and columns in the main loop.

A second optimizations of the Gauss-Jordan method lies in the possibility to vectorize loops. The processor has 256-bit SIMD registers provided by the Advanced Vector Extensions (AVX) technology. For double precision floating point numbers, a loop speed-up of 4 times can be achieved since 4 floating point vector operations can be executed at the same time. This is very handy for the most inner loops of the code, where row operations such as row swapping or addition takes place.

The third major optimization to be considered is a very basic one, namely loop unrolling. This provides even more optimization as additional branching instructions during the loop counting can be scaled down to only a few branches. This can be done not only for the most inner loops, but for the outer loops as well. There are a few other optimizations, such as aggressive instruction prefetching, floating point speculation and automatic inlining of functions where possible.

B. Implementation of the Optimization Techniques

With the previously provided information, one might argue how to implement all these optimization techniques. A programmer in the typical scenario would not want to spend additional time for implementing optimizations in the code, especially SIMD instructions and loop unrolling. Modern C++ compilers, such as the Intel C++ Compiler are very sophisticated in this manner. It provides optimized matrix multiplications to fully exploit the available cache memory, such as working over blocks of matrices, rather than rows or columns. It allows one to set the instruction prefetching to an aggressive mode, speculation of floating point instructions and automatic inlining of functions without having to specify the *inline* directive in the code. The compiler also supports automatic code generation, particularly AVX instructions that exploit the SIMD registers, without having to change a single line of the original C++ code. This way, many loops can be automatically vectorized to gain even more performance. The third major optimization with loop unrolling is supported in most compilers. For our short code, we can safely instruct the compiler to unroll the loops 500 times. This technique eliminates 500 branching instructions in the assembly code. The following section summarizes our results regarding these optimizations of the C++ code for the Gauss-Jordan elimination method.

IV. Experiments

This section examines our initial hypothesis that exploiting the power of modern processors and compiling techniques can provide a very significant boost the running time of numerical algorithms much more than implementing a completely different algorithm which theoretically is several times faster. Our goal is to show that if one makes real use of the processor, the running time of an easy-to-implement algorithm can be very satisfactory, without having to put additional time and resources in a more sophisticated algorithm that takes more knowledge and patience to be correctly implemented. The Gauss-Jordan method to solve a system of linear equations is what everyone is taught at school and is very easy to implement. LU factorization requires more knowledge of linear algebra and matrix properties, and different algorithms that take specific properties of matrices have been proposed. The purpose is to show that it is very important to be aware of the capabilities of modern processors.

The experiments were conducted on a machine with Intel Core i7 4500U (see Table I), with 8 GB DDR3 DRAM and running Ubuntu 15.04 as the host operating system. In each run of the compiled code, the coefficient matrix $\mathbf{A}$ is randomly initialized to double precision floating point numbers up to 100. Note that random matrices are very rarely singular (non-invertible), but only non-singular matrices were considered. We have used Intel Performance Counter Monitor C++ API to get precise measurements from the processor's own performance counters [13].

A. Execution Time Analysis

The experiments to analyse the execution time of the algorithms have been performed independently for gradually increasing values of N, from 5 to 3,000 equations. Throughout the tests a single right-hand side vector b was used because r, the number of right-hand side vectors, has no influence in the dominant cubic factor in the complexity, as shown in Equations 3 and 7. The results are summarized in Figure 2. The

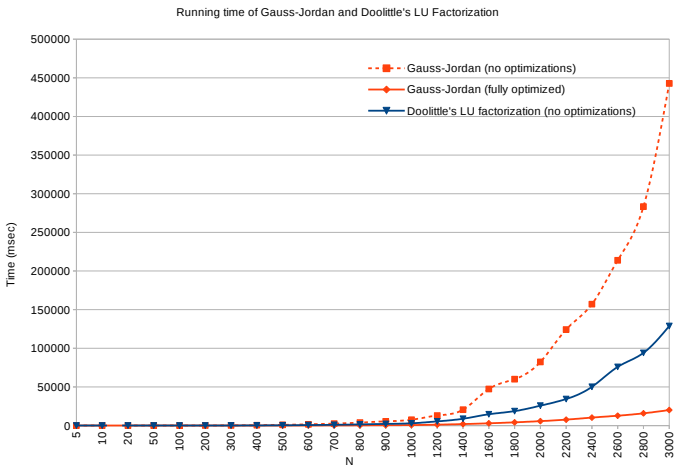

Fig. 2: The execution time of Gauss-Jordan elimination method versus Doolittle's LU factorization.

difference between the execution times of the three algorithms in Figure 2 is mostly emphasized for $N > 1000$. It can be seen that the graphs of the non-optimized Gauss-Jordan algorithm and Doolittle's algorithm start to grow very fast, in tens of thousands of milliseconds. Moreover, they precisely reflect the ratio of their theoretical complexities. The former is about two to three times slower. In contrast, the optimized Gauss-Jordan algorithm by Intel's C++ compiler is much faster than Doolittle's algorithm. In fact, it is more than six times faster for $N = 3000$.

The shape of the bottom curve visually resembles a much slower-paced growth rate of a quadratic polynomial with very small coefficient values, while the others could be a polynomial with much greater coefficients or even of a higher degree. Note that this is only a visual analysis, since the axes do not correspond to a equally-scaled coordinate system. The same analysis also holds for $N \leq 1000$.

1) Vectorization: Modern compilers like Intel's C++ compiler have the ability to generate processor specific code, particularly for the dedicated SIMD registers. One can insert such instructions manually into the code, but sophisticated compilers can do that automatically. The fully optimized version of the Gauss-Jordan elimination algorithm in Figure 2 uses this optimization too, but let us see how it improves the overall performance. Figure 3 shows the performance gain in terms of reduced execution time by automatic vectorization of loops. The results show that at least 10% of the execution time

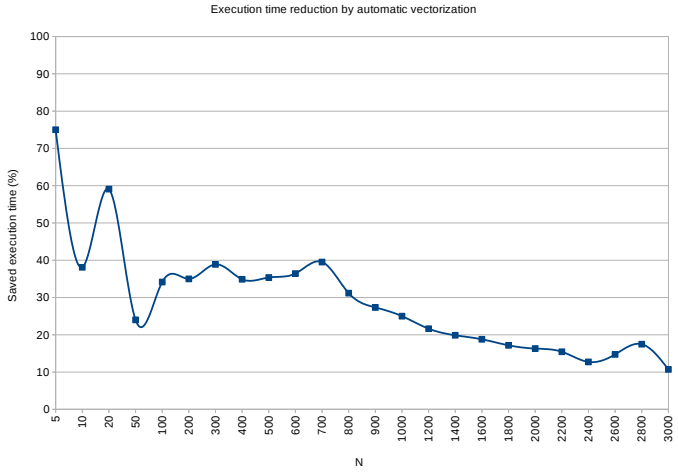

Fig. 3: Reduction of the execution time by automatic vectorization, expressed as a fraction (in percent) of the execution time of the optimized code where vectorization has been left out.

can be saved with vectorization. The peak is at 75% savings for $N = 5$, which is expected because of the low order of the matrix. For N below 1000, there is 30% to 40% reductions, and for N greater than 1000 these savings range from 10.7% to 21.6%, which is certainly not insignificant. This comes from the fact that when N is a multiple of 4, there are no loop remainders to be additionally executed after the vectorized portion of the original loop. But when N is not a multiple of 4, this disadvantage can be eliminated because remainder loops are not longer than 3 iterations. Generally, we do not have to worry about this, and such conditions can be ignored. It is notable that a positive amount of reduction is always present as it never gets below zero, and thus, vectorization is practical for real-world engineering applications with no more than 3000 equations, as the results show.

B. Processor Performance Metrics

In this part we examine the processor's performance in terms of L2 and L3 cache hit rates, instructions per cycle (IPC) and the number of retired instructions.

Since we have 256KB L2 cache and 4MB shared L3 cache available, it is very likely that linear equation systems of order $N < 180$ can fit in the private L2 cache, but considering other OS environmental factors, we do not expect the whole L2 private cache to be fully available. On the other hand, the shared L3 cache can fit a coefficient matrix of order $N < 620$. Doolittle's algorithm accesses both rows and columns of the matrix. For system of order 100, Doolittle's algorithm achieved a maximum L2 hit rate of 92.3%, while Gauss-Jordan had 92.6%. For a system of order 500, the hit rates in the L3 shared cacahe are 95.3% and 83.7% respectively. It turns out that generally, the hit rates are close to each other for systems that can fit entirely in the cache. But for $N = 1000$, Doolittle's algorithm has 88.9% L2 hit rate and 73% L3 hit rate, while Gauss-Jordan has much lower hit rates of 3.6% for the L2 cache and 5.03% for the L3 cache. Yet, it executes roughly five times faster than Doolittle's algorithm. To conclude, the cache efficiency of the algorithms does not contribute to the performance gain between the two algorithms.

The maximum IPC on the processor is 4, and Doolittle's algorithm is much better than Gauss-Jordan optimized by the compiler. Its average IPC is usually above 2.0, ranging from 2.0 to 2.27 for $N > 1000$. In contrast, Gauss-Jordan's IPC is generally below 2.0, ranging from 1.39 to 1.44. Thus, Doolittle's algorithm utilizes the processor far better than Gauss-Jordan's algorithm. But, where does this come from? The explanation is rather simple. We observed that Gauss-Jordan takes 4.7 times less CPU cycles and retires 7.77 times less instructions, which is 1.65 times less IPC than Doolittle's algorithm. This is why making conclusions about the processor's performance based on the IPC can be very misleading.

Finally, let us consider the number of retired instructions by both algorithms. This is the essence of the performance gain. Figure 4 illustrates the number of retired instructions, or the number of instructions generated by the compiler for the two algorithms. As the figure shows, the number of instructions generated by Intel's C++ compiler is dramatically lower than the number of instructions for Doolittle's algorithms. This is even more exaggerated in the case of a non-optimized Gauss-Jordan elimination algorithm. For a linear equation system of order 1000, the non-optimized Gauss-Jordan algorithm is 13.4 times worse, while Doolittle's algorithm is 8.52 times worse. These numbers range from 11.2 to 13.4 and 7.25 to 8.52 for the non-optimized Gauss-Jordan algorithm and Doolittle's algorithm, respectively. These results are crucial to the understanding of the compiler's optimization techniques and the consequent performance gains. To summarize, this section shows that an algorithm that is roughly 2.5 times slower can be a lot more efficient and faster than its theoretically faster counterpart.

V. Conclusion and Future Work

In this paper we showed how a relatively high degree of optimization can be achieved by exploiting the power of modern processors and modern sophisticated compilers, such as the Intel C++ Compiler. Our hypothesis that without changing a single line of the source code and being aware of the host processor capabilities, advanced compiling techniques can deliver speed-ups that completely change the way we look

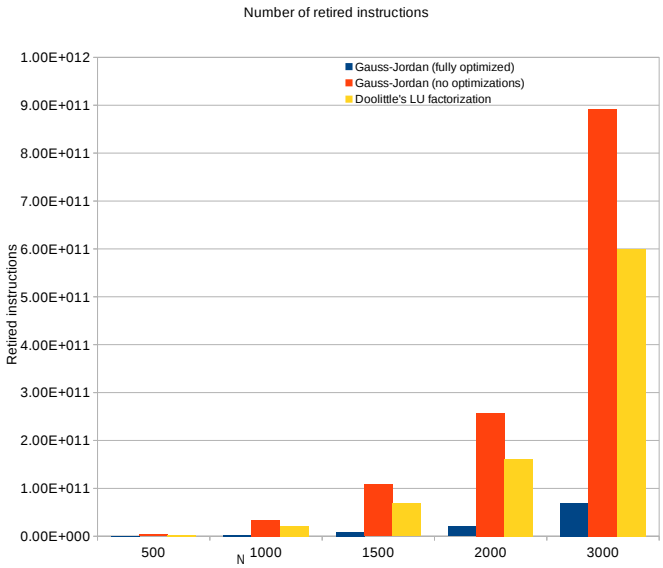

Fig. 4: Number of instructions retired for each algorithm as inspected by the Intel Performance Counter Monitor API.

at algorithms. The choice of an algorithm based solely on its theoretical complexity can be very misleading if straightforward (or optimization-free) implementation and compilation is being used. The results proved that modern compilers can yield much shorter and faster code besides a demotivating theoretical complexity.

The two algorithms examined in this paper come from numerical linear algebra, so a suggestion for further research is to test similar hypotheses on other numerical algorithms, such QR decomposition, eigenvalues calculation and so on. The results are also a motivation to examine more convenient algorithms such as quick sort and merge sort, or even algorithms for matrix multiplication. To sum up, we believe that many algorithmic techniques need a refreshment to meet today's modern processors' capabilities

REFERENCES

[1] J. L. Hennessy and D. A. Patterson, *Computer architecture: a quantitative approach*. Elsevier, 2011.

[2] D. Geer, "Chip makers turn to multicore processors," *Computer*, vol. 38, no. 5, pp. 11–13, 2005.

[3] R. M. Tomasulo, "An efficient algorithm for exploiting multiple arithmetic units," *IBM Journal of research and Development*, vol. 11, no. 1, pp. 25–33, 1967.

[4] J. L. Lo, J. S. Emer, H. M. Levy, R. L. Stamm, D. M. Tullsen, and S. J. Eggers, "Converting thread-level parallelism to instruction-level parallelism via simultaneous multithreading," *ACM Trans. Comput. Syst.*, vol. 15, no. 3, pp. 322–354, Aug. 1997. [Online]. Available: http://doi.acm.org/10.1145/263326.263382

[5] S. J. Eggers, J. S. Emer, H. M. Leby, J. L. Lo, R. L. Stamm, and D. M. Tullsen, "Simultaneous multithreading: A platform for next-generation processors," *Micro, IEEE*, vol. 17, no. 5, pp. 12–19, 1997.

[6] "Advanced computer architecture project suggestions," 1998, accessed 2015-07-19. [Online]. Available: http://www.cs.berkeley.edu/~pattrsn/252S98/projects.s98.html

[7] G. Peters and J. H. Wilkinson, "On the stability of gauss-jordan elimination with pivoting," *Communications of the ACM*, vol. 18, no. 1, pp. 20–24, 1975.

[8] W. H. Press, *Numerical recipes 3rd edition: The art of scientific computing*. Cambridge university press, 2007.

[9] "Advancedcomputerarchitecture," accessed 2015-07-22. [Online]. Available: https://github.com/ninoarsov/AdvancedComputerArchitecture

[10] G. E. Forsythe and C. B. Moler, *Computer solution of linear algebraic systems*. Prentice-Hall Englewood Cliffs, NJ, 1967, vol. 7.

[11] R. Melhem, "Parallel gauss-jordan elimination for the solution of dense linear systems," *Parallel Computing*, vol. 4, no. 3, pp. 339–343, 1987.

[12] D. Kaya and K. Wright, "Parallel algorithms for lu decomposition on a shared memory multiprocessor," *Applied mathematics and computation*, vol. 163, no. 1, pp. 179–191, 2005.

[13] T. Willhalm, R. Dementiev, and P. Fay, "Intel performance counter monitor - a better way to measure cpu utilization," 2012, accessed 2015-07-19. [Online]. Available: www.intel.com/software/pcm