PRACTICAL EFFICIENCY OF OPTIMIZING COMPILERS IN PARALLEL SCIENTIFIC APPLICATIONS

Bogdanov A.V.[1], Gankevich I.G.[2]

[1]Saint-Petersburg State University, Russia
bogdanov@csa.ru
[2]Saint-Petersburg State University, Russia
gig.spb@gmail.com

Optimizing compilers are essential for building any scientific application, however they are not general purpose tools. Although, many compilers offer similar functionality, different optimization strategies as well as code structure can lead to different performance results [1]. Additionally, modern scientific applications often solve large-scale problems concurrently on a set of processors thus demanding not only serial but parallel code optimizations. So, choosing the right compiler for a particular problem is a topical question of today.

In the paper a variety of commercial and open source optimizing compilers are compared in terms of their functionality. Then their relative performance is measured benchmarking different sets of algorithms and scientific applications classified by their problem domains. The final results are presented as a cumulative compiler rating scored in a particular problem domain. Based on this rating conclusions are made.

1 Introduction

History of optimizing compilers implementing parallelization can be divided into three major milestones showing emergence of different parallel technologies. The first standardized parallel technology introduced in 1994 was MPI (Message Passing Interface) exposing coarse-grained parallelism of a homogeneous cluster. The second one standardized in 1995 was Pthreads (POSIX Threads) exposing fine-grained parallelism of a shared memory multi-processor or multi-core homogeneous system. The most recent standard released in 2008 was OpenCL (Open Computing Language) that can be used to exploit parallelism of a heterogeneous hybrid system consisting of multiple CPUs and GPUs. These standards are bundled into compilers either as libraries, as a set of directives or as means of automatic parallelization.

Analysis of compiler output was traditionally performed using sophisticated instrumenting tools, however better understanding of compiler's work can be obtained using optimization reporting options. These options make compiler produce valuable information on succeeded and failed optimizations showing corresponding lines of code. Optimization reports work in harmony with conventional instrumentation tools (e.g. Valgrind, Oprofile) providing justification of compiler decisions and optimization hints and are especially useful when translating parallel code.

Compilers have different level of support for introduced parallel technologies and current top level of each technology can be outlined as follows (Figure 1):

1. MPI library support,
2. Pthreads auto-parallelization support,
3. directive-based GPU parallelization support.

Library-level support for open standards (unlike the proprietary CUDA) is the default level for any compiler and at this level there is not much compiler can optimize. The efficiency of low-level system parallel library is dependent on its implementation and not the compiler used to link it to a program, not to mention that MPI library is distributed also by “non-compiler” vendors (Platform MPI, OpenMPI). Therefore, it leaves discussion of library-level support for MPI out of topic.

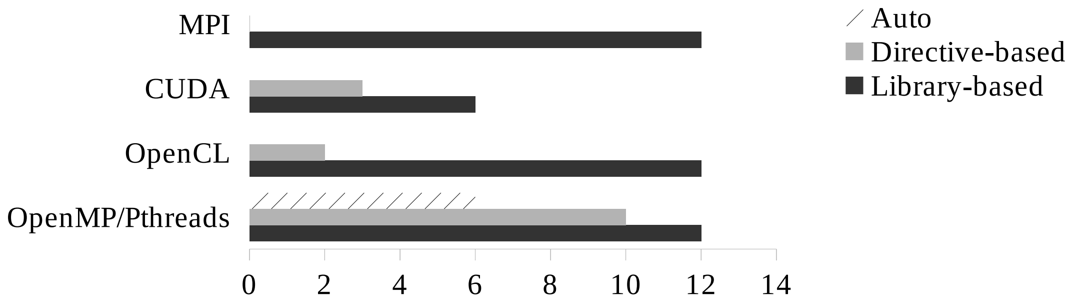

Fig. 1: Different levels of support for parallel standards (both open and proprietary). The numbers show amount of compilers supporting technology at a specified level.

Directive-based parallelization and auto-parallelization support are the levels where compiler has the most control over parallelization process and they are the most useful levels for compiler benchmarking purposes. Directive-based approach may include optimizations involving reduction of redundant data movement, elimination of unneeded synchronization points and others. Auto-parallelization level adds an heuristic algorithm to determine the most suitable program part to perform parallelization. All in all, benchmarking directive-based and automatic parallelization of compilers may give insight into their overall efficiency.

To summarize, from the parallel applications developer point of view the most useful compiler facilities aiding in parallelization process are support for compiler-based parallelization and optimization reporting. In the first case, parallelization reveals compiler's ability to optimize program in multi-threaded environment and for benchmarking purposes both CPU and GPU parallelization should be considered (Section 2 and 3). Finally, optimization reports are useful to reveal compiler's decisions in problematic lines of a source code (Section 4). So, compiler's ability to parallelize on par with optimization reports can enhance program performance.

2 Compiler-based parallelization efficiency

Compiler-based parallelization is the simplest approach to optimize and boost resource-intensive programs, and other than that, it is also a good way to show efficiency of compiler's optimizations with a view to parallel code. One of the most useful optimizations for scientific applications are those involving loops such as loop tiling, loop unrolling and loop interchange, as they maximize cache usage when processing large arrays of data [1]. They often work in harmony with other kinds of optimizations such as vectorization and invariant code motion that further improve resulting program performance. Thus, auto-parallelization can reveal the level of compiler's “skills” with a view to these optimizations.

Benchmark was carried out on the basis of two BLAS (Basic Linear Algebra Subprograms) library implementations and in a similar manner both for CPU and GPU parallelization. The first implementation is non-optimized reference BLAS implementation [2] compiled with a maximum optimization level and auto-parallelization options. The second one is hand-tuned GotoBLAS library [3,4] compiled with self-chosen set of options. In case of GPU OpenACC [4] directives were added before each outermost *for* loop inside BLAS routines so that it can be parallelized by the compiler. So, each compiler built auto-parallelized and hand-tuned versions of the library and PGI also built GPU version of the library.

BLAS library was chosen because it is de facto standard in scientific software development and it is also contains the simplest algorithms to parallelize. Algorithms consist of vector-vector, vector-matrix and matrix-matrix operations that correspond to the Level 1, 2 and 3 of library routines respectively. Each algorithm includes no more than a triple of nested loops assembled exclusively for different argument variations (in case of Level 3 – matrix transpositions) [2]. The two inner loops can be used to expose both coarse-grained parallelism of multiple processor cores and fine-grained parallelism of vector registers. So, the choice of BLAS library was based on a desire to test compilers on the algorithms that are easy to parallelize.

Benchmarking strategy consisted of running a subset of Level 3 routines with a variable set of actual arguments' values and using different number of threads. Arguments' variations included different matrix sizes, transposed/non-transposed cases, upper/lower triangular matrices and left/right sided equations. Thread number varied from 2 to 12 threads that correspond to the total of 12 machine cores. Benchmark results were summarized in one table containing approximately 2500 rows for subsequent analysis.

Average performance of auto-parallelized BLAS code is lesser than of hand-tuned BLAS library (Table 2), and further investigation shows significant variations in efficiency when using transposed and non-transposed matrices (Table 1). In case of *SGEMM* the source code for *SGEMM(A^T, B)* and *SGEMM(A^T, B^T)* differs only in one index (Figure 2), and their relative performance differs by an order of magnitude. Naturally, this index prevents loop vectorization and leads to ineffective cache utilization due to non-unit stride memory access pattern. As matrix operations such as *SGEMM* are not arithmetically intensive involving no more than floating point additions and multiplications, it is essential to optimize CPU cache usage and GPU loads/stores coalescing to achieve scalable performance. However, it is hard for compilers to do so and matrix transpositions lead to performance degradation.

SGEMM(A^T, B)

```
DO J = 1,N
  DO I = 1,M
    TEMP = ZERO
    DO L = 1,K
      TEMP = TEMP + A(L,I)*B(L,J)
    CONTINUE
    IF (BETA.EQ.ZERO) THEN
      C(I,J) = ALPHA*TEMP
    ELSE
      C(I,J) = ALPHA*TEMP +
BETA*C(I,J)
    END IF
  CONTINUE
CONTINUE
```

SGEMM(A^T, B^T)

```
DO J = 1,N
  DO I = 1,M
    TEMP = ZERO
    DO L = 1,K
      TEMP = TEMP + A(L,I)*B(J,L)
    CONTINUE
    IF (BETA.EQ.ZERO) THEN
      C(I,J) = ALPHA*TEMP
    ELSE
      C(I,J) = ALPHA*TEMP +
BETA*C(I,J)
    END IF
  CONTINUE
CONTINUE
```

Fig. 2: Source code for *SGEMM(A^T, B)* and *SGEMM(A^T, B^T)* differs only in one index and performance differs by an order of magnitude (see Table 1).

GCC inferior performance compared to other compilers seems to be attributed to the restriction of the compiler to parallelize only innermost loops [5]. Since benchmarked Level 3 routines contain loops nested up to the third level it is inefficient to parallelize only innermost loop as it may involve synchronization overheads and also as it constitutes only a part of a whole problem. As GCC does not produce any optimization report during compilation there is no easy way to give a reliable explanation of low performance.

		Performance, Mflops				PGI SGEMM cache statistics ×10^9			
op(A)	*op(B)*	Intel	PGI	GCC	PGI CUDA	Reads	Writes	Misses	Instructions
A^T	B	19108	10193	1683	1326	137.4	0.016	0.008	412
A	B^T	17935	9738	2982	1341	137.5	68	0.039	481
A	B	16824	9453	3110	2335	137.5	68	0.008	481
A^T	B^T	3424	946	199	2407	137.4	0.016	0.104	549

Table 1. The effect of matrix transposition on the performance of auto-parallelized BLAS library.

GPU parallelization which was carried out on the basis of directive-based PGI CUDA compiler shows inferior performance compared to other compilers, however, matrix transpositions have an opposite effect on the efficiency (Table 1). The effect. which consists of GPU code having the best performance in cases where CPU code having the worst, is attributed to CUDA storing matrices in a row-major order in contrast to Fortran storing them in a column-major order. Differences of matrix storage scheme affects memory access pattern which in turn affects CPU cache utilization and GPU memory loads/stores coalescing and finally defines efficiency of arithmetically non-intensive code. Although, GPU is perfectly suitable for matrix operations it is inherently parallel device in contrast to CPU and the best performance is generally achieved by using specific algorithm and implementation. So, to show comparable performance GPU compiler should rewrite a whole routine and not just parallelize existing CPU code. All in all, the best results are achieved when parallelizing code on CPU.

Routine	Hand-tuned (CPU)			Parallelized (CPU)			Parallelized (GPU)
	Intel	PGI	GCC	Intel	PGI	GCC	PGI
SGEMM	111974	**130197**	57816	**16173**	7333	1824	1939
SSYMM	108611	**130466**	60525	9089	**12123**	2390	1632
SSYR2K	103813	**112828**	60230	**18157**	13728	3136	1917
SSYRK	95721	**106000**	43239	**14562**	10027	1811	1218
STRMM	99073	**115354**	43204	**14423**	7148	1990	2102
STRSM	97371	**115009**	41857	**11381**	6295	1745	1998

Table 2: Compiler parallelization efficiency of Intel, PGI and GCC compilers in Mflops. Target platform: HP SL390s G7, 2x Intel X5650 2.67 Ghz (12 cores total), 96 Gb RAM, 3x NVIDIA Tesla M2050.

To summarize, benchmarks show that average performance of hand-tuned optimized BLAS library is an order of magnitude higher than of automatically parallelized library. The largest performance variations appear when performing operations on transposed matrices that affect efficient CPU cache usage and GPU memory operations coalescing. Finally, as GPU and CPU vastly differ in their architectures and programming style, the best GPU performance is achieved when implementing algorithm from a scratch and not adapting CPU source code. All in all, compiler auto-parallelization facilities can be seen as a means of making the first parallel program prototype which is incrementally optimized by hand to reach its best performance.

3 Compiler-based parallelization overhead

Overheads of compiler-based parallelization are defined by a corresponding runtime library implementation and are imposed by usage of directives. These include thread scheduling algorithms overhead, parallel region entering overhead and synchronization constructs overhead. Analysis of those overheads can be performed using micro benchmarks specifically designed to measure efficiency of directives.

Benchmark, which was carried out via EPCC benchmark suite [6] developed to analyze multi-processor machine efficiency, showed that it can also be used to compare compiler's OpenMP runtime library performance. In the original paper this suite is used to show variations in directive overheads running benchmarks on different hardware platforms, however, if the source code is translated by different compilers and run on a single machine then it shows representable performance of a particular OpenMP runtime library. So, EPCC benchmark suite was used to measure overhead of OpenMP directives using a set of supporting compilers.

Results of the benchmarks revealed most directives having similar overheads in case of PGI and Intel compilers and larger overheads in case of GCC, however, there are also some directives with different overhead pattern. These are *omp reduction* and *omp schedule(dynamic)* showing inferior performance of PGI and *omp schedule(guided)* showing inferior performance of both PGI and Intel compilers (Figure 4). In addition, these directives also account for no more than 5% of a total count of *omp for* directives used in examined scientific applications (Figure 5). All in all, commercial compilers have more efficient OpenMP runtime library implementations than open source GCC compiler does.

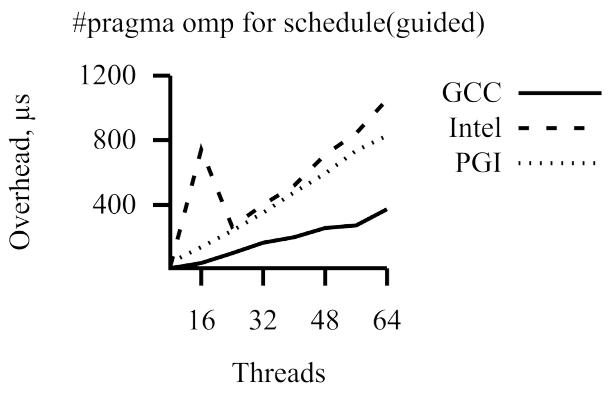

Fig. 4: Overhead of *omp schedule(guided)* directive showing inferior performance of commercial PGI and Intel compilers compared to open source GCC. Target platform: HP Proliant DL980, 8x Intel X7560 2.2 Ghz (64 cores), 512 Gb RAM.

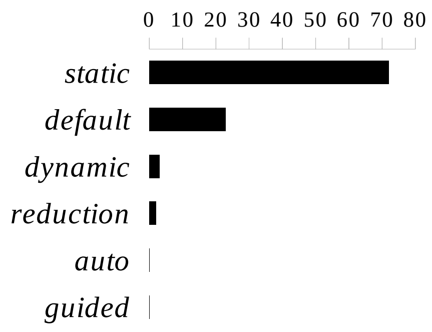

Fig. 5: Distribution of scheduling and reduction directives among all *omp for* directives in scientific software packages WRF (Weather Research Forecasting) and WaveWatch3.

4 Optimization reports

Optimization report is a special type of compiler's output telling developer about compiler's decisions in optimization process. Although not standardized, this output often includes decisions on loop vectorization, loop tiling, data distribution during loop parallelization and also decisions on an alternative loop code generation. The information is presented in a form of hints about why particular optimization technique was or was not employed, thus providing developer with a way of optimizing a source code for a given compiler.

Optimization hints may be useful, however, it is hard to measure their impact on an application performance. On one hand, they provide user with a knowledge of compiler's inner workings relieving him from tedious machine code analysis. On the other hand, following optimization hints of one compiler not always results in an optimal code for another compiler. So, this limits usefulness of optimization reports only to the most simple cases of inefficient code structure considering target's machine architecture.

Conclusion

In conclusion, practical efficiency of optimizing compilers is dependent on many factors. Intel compiler lets developer achieve easy parallelization of a program prototype, but reduced efficiency of a hand-tuned source code version. In contrast to this, PGI compiler offers quite an opposite: a prototype is slow but a final version is highly optimized. In either case, usage of non-commercial GCC compiler degrades performance of both a prototype and a final application. From a different point of view, one can use the most efficient compiler on a corresponding development stage and also for a particular target platform. All in all, final program performance is dependent on many factors and compiler's impact shall not be considered in isolation.

Acknowledgements

The research was carried out using computational resources of Resource Center Computational Center of Saint-Petersburg State University (T-EDGE96 HPC-0011828-001) that provided commercial compiler's licenses and access to high-performance machines for the scientific work.

References

[1] Bacon D.F., Graham, S.L., Sharp O.J. Compiler transformations for high-performance computing. ACM Computing Surveys (CSUR). Vol. 26:4, pp. 345-420, 1994

[2] J. J. Dongarra, J. Du Croz, I. S. Duff, and S. Hammarling, A set of Level 3 Basic Linear Algebra Subprograms, ACM Trans. Math. Soft., 16 (1990), pp. 1-17.

[3] Kazushige Goto and Robert A. van de Geijn. "Anatomy of High-Performance Matrix Multiplication," ACM Transactions on Mathematical Software 34(3): Article 12, 25 pages, May 2008.

[4] Kazushige Goto and Robert van de Geijn. "High-Performance Implementation of the Level-3 BLAS." ACM Transactions on Mathematical Software 35(1): Article 4, 14 pages, July 2008.

[5] http://gcc.gnu.org/wiki/Graphite/Parallelization. GCC Graphite Parallelizer official WIKI page.

[6] J. m. Bull. Measuring Synchronisation and Scheduling Overheads in OpenMP. 1999.