Performance of Parallel Algorithms Using OpenMP

Roman Mego
and Tomas Fryza
Department of Radio Electronics
Brno University of Technology
Brno, Czech Republic
roman.mego@phd.feec.vutbr.cz

Abstract — **This paper describes tests performed by parallel signal processing algorithms on Texas Instruments 8-core digital signal processor and Intel Core i7. Tests reveal how the change of number of threads and size of input data influences the performance compared with sequential code. The result of the test is that the relative speedup is highly dependent on type of the algorithm and the amount of processed data.**

Keywords — digital signal processing, OpenMP, parallel, TMS320C6678, Intel Core i7, FIR, DFT, FFT

I. INTRODUCTION

The increasing of computing performance has taken a direction to the parallelization in last 10 years. This trend is not connected only with the general purpose processors and personal computers, but with the embedded devices as well (e.g. tablets, smartphones). Many of the developers have their own software libraries which are used in more than one application. If these libraries are desired to run on different platforms, they are probably written in C. For this reason, it could happen that the final program cannot take benefits, which the new hardware offers, including multi-core processors. One of the solutions how to make old code to run on multi-core processor is to use OpenMP. This paper describes the OpenMP, how to simply localize possible parallelism and its impact on the performance.

The rest of the paper is divided as follows. Section 2 describes the OpenMP and which types of parallelism supports. Section 3 shows implemented algorithms and where is the parallel execution. Section 4 compares platforms on which test was performed. Section 5 deals with the measured execution times.

II. OPENMP DESCRIPTION

OpenMP is an application program interface (API), which provides a portable, scalable model for shared-memory programming. First specification of OpenMP was defined in 1997 for FORTRAN by major hardware and software vendors. One year later OpenMP was defined for C/C++. In these days, the specification of version 4.0 is prepared [1].

OpenMP uses thread based parallelism with fork-join model. This means, that application start in one thread and if it come to parallel section, it creates another threads. When this team of threads completes their work, they synchronize and terminate except master thread. These threads can be section work-sharing and loop work-sharing [2].

A. Section work-sharing

This type of work-sharing can be used for independent pieces of code which can run in parallel. Parallelism of this type is similar to creating threads through standard libraries provided by operating system. It can be used to pipeline the processing.

Figure 1 shows the example of section work-sharing. The original algorithm consists of 4 steps and is performed sequentially. Steps 2 and 3 are independent and can be performed in different order or in parallel.

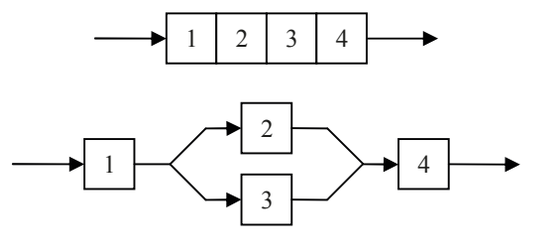
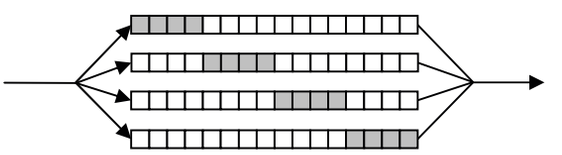


Figure 1. Example of section work-sharing

B. Loop work-sharing

The loop work-sharing is the common way how to increase the performance of the application. For-loops are primary targets in parallelization. They can be used if iterations have no dependencies between each other.

On figure 2 is shown a for-loop parallel execution, which is processing an array with length of 16. It is divided into 4 threads, where each of them process 4 values. Parts of array which are processed are marked in grey colour.



This paper was supported by the internal grant of BUT project FEKT-S-11-12 (MOBYS). The described research was performed in laboratories supported by the SIX project; the registration number CZ.1.05/2.1.00/03.0072, the operational program Research and Development for Innovation.

978-1-4673-5517-9 ©2013 IEEE

Figure 2. For-loop parallel execution

In this case the processed data are shared among all cores. If the algorithm uses some auxiliary variables and they are used by all threads, the code will possibly return wrong result. These variables must be defined as private, which means that there is created local copy in memory for every core.

III. ALGORITHMS PARALLELIZATION

This part is dealing with a parallelization of selected signal processing algorithms. It is especially finite impulse response (FIR) filter, discrete Fourier transform (DFT) and Fast Fourier transform (FFT). Parallelization of code is realized with OpenMP directives. During this process, it is important to take care of which variable is shared between threads and which must be created as private for each thread.

A. FIR Filter

FIR filter is implemented according to (1) from [3]. This type of filter was selected, because it does not require feedback, which could not be simply parallelized. Final code contains 2 nested for-loops, but only outer loop is parallel. However OpenMP support nested parallelism, inner loop is performed sequentially. It is because the number of physical cores is less than number of signal samples and there is no space where to execute other threads.

$$y_n = \sum_{k=0}^{N-1} x_{n-k} h_k \tag{1}$$

B. Discrete Fourier Transform

Structure of the DFT implementation is similar to the FIR filtration. It consists of 2 nested for-loops. The difference is that there are complex calculations and the inner loop goes through full length of the signal. This means, that the amount of processed data is much higher in compared to the FIR filter. The definition of DFT (2) was taken from [3].

$$X_k = \sum_{n=0}^{N-1} x_n e^{\frac{-i2\pi kn}{N}} \tag{2}$$

C. Fast Fourier Transform

For the demonstration of FFT, the Cooley-Tukey algorithm was chosen. The structure is different from the previous implementations. Figure 3 schematically shows progress of used loops in algorithm. The outer loop iterations, which represent the stage in FFT, obviously depend on each other, so it cannot be executed in parallel. The middle, representing group of butterflies, and the inner loop, representing butterfly processing, are independent in each of its iteration. For simplicity, only middle loop was chosen for parallelism even if last stages will not benefit from this.

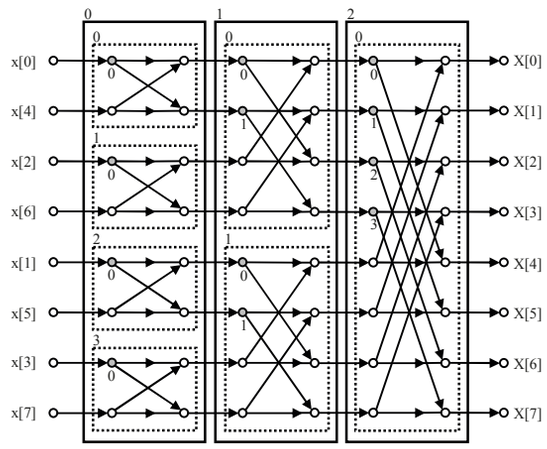


Figure 3. FFT radix-2 with highlighted loop iterations

IV. TESTED PLATFORMS

The final code was compiled for 2 different platforms. The first is personal computer with Intel Core i7-2670QM. Second is development board with multi-core digital signal processor (DSP) from Texas Instruments (TI) TMS320C6678.

A. Intel Core i7-2670QM

The Intel Core i7-2670QM is quad-core processor for mobile products. It offers hyper-thread technology which means that every physical core has two architectural states with general-purpose registers and control registers. Therefore one physical core can be used as two logical cores, where the execution unit, cache memory and buses are shared. This processor also includes smart-cache, which means that the last level cache memory is shared among all cores. More information can be found in [4].

The source code was compiled with compiler provided with Intel C++ Studio XE 2013. The advantage is that it supports all streaming SIMD extensions (SSE) and advanced vector extensions (AVX) in instruction set.

B. Texas Instruments TMS320C6678

The TMS320C6678 is fixed and floating-point DSP with 8 C66x CorePacs. Each core consists of 8 functional units, 2 sets of register files and 2 data paths. Functional units are divided in 2 equal groups with .M, .L, .D and .S unit. The .D unit is primary used for loading and storing data from memory, .M unit is capable of multiply operations in single and double precision, .L and .S units performs general logic, arithmetic and branch functions. More information about structure of the DSP can be found in [5].

Final parallel code cannot run without operating system, which controls threads. TI provides real-time kernel called SYS/BIOS. It is designed to use in embedded applications which requires real-time scheduling.

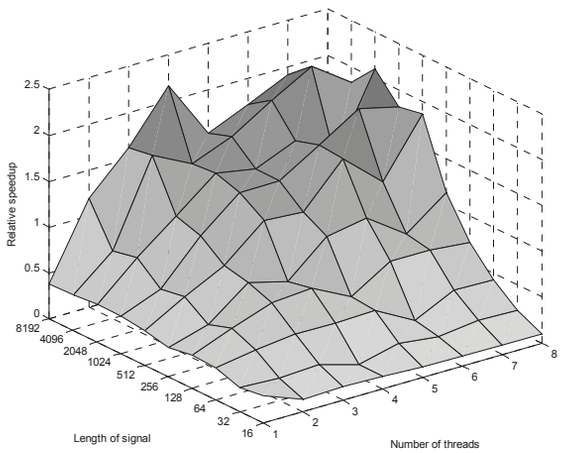


Figure 4. Relative speedup of FIR filter on CPU

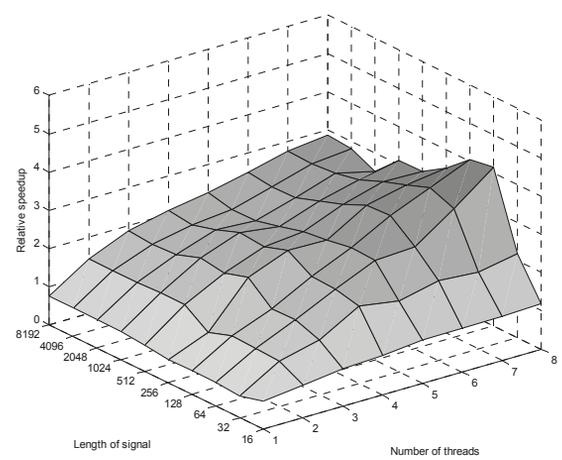


Figure 5. Relative speedup of DFT on CPU

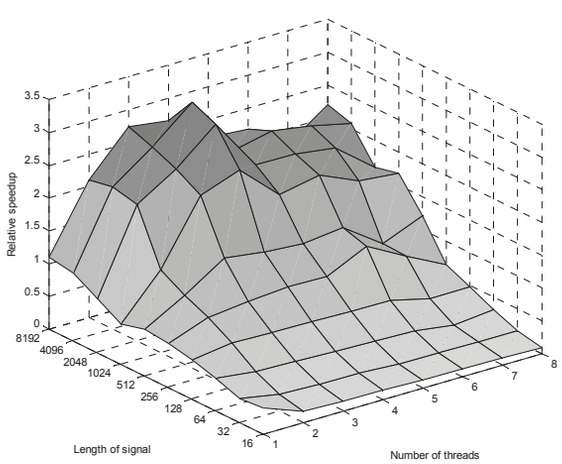


Figure 6. Relative speedup of FFT on CPU

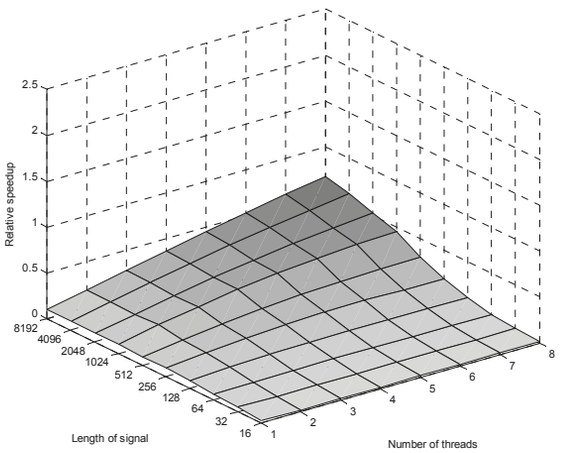


Figure 7. Relative speedup of FIR fiter on DSP

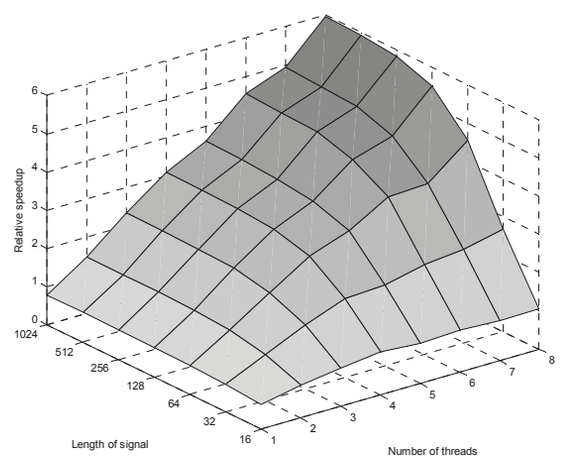


Figure 8. Relative speedup of DFT on DSP

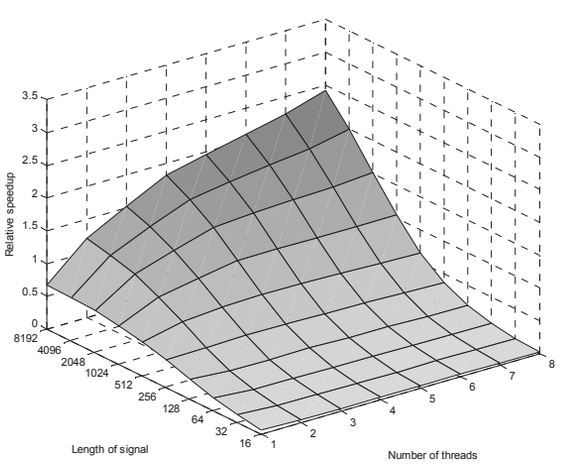


Figure 9. Relative speedup of FFT on DSP

TABLE I. COMPARISON OF SELECTED PLATFORMS

	Core i7-2670QM	TMS320C6678
Clock Speed	2.2 GHz	1 GHz
L1P Memory	32 kB/Core	32 kB/Core
L1D Memory	32 kB/Core	32 kB/Core
L2 Memory	256 kB/Core	512 kB/Core*
L3 Memory	6 MB Shared	-
GFLOPS	70.4	128
Thermal Design Power	45 W	17 W

*TMS320C6678 have in addition 4 MB shared L2

V. MEASURED PERFORMANCE

The execution time of whole function call represents the performance of implemented algorithms. Dependence of execution time on number of created threads and length of input signal was measured. For determining how the performance of algorithms was influenced with changing of these parameters and by the OpenMP runtime, the execution time of sequential versions (without OpenMP pragmas) of algorithms was chosen as reference.

Figure 4 to 6 shows the relative increase of performance on Intel CPU and figures 7 to 9 are characteristics measured on TI DSP. Reference times in microseconds are shown in table 2.

TABLE II. MEASURED REFERENCE TIME [µS]

Length	Core i7-2670QM			TMS320C6678		
	FIR	*DFT*	*FFT*	*FIR*	*DFT*	*FFT*
16	0.148	5.74	0.837	2	157	16
32	0.369	22.8	2.15	4	605	39
64	0.819	91.8	5.25	8	2457	93
128	1.57	367	12.6	16	9911	215
256	2.82	1455	28.6	32	39832	491
512	6.48	5738	65.9	63	159782	1105
1024	13.5	22949	145	126	640102	2456
2048	26.7	91556	322	252	-	5405
4096	51.9	364197	706	507	-	11810
8192	96.1	1489188	1547	1025	-	25791

From graphs can be seen, that performance of all algorithms with OpenMP directives are slower when there is only master thread. It is because the process of thread creating is still active, even if the maximum number of threads is set to 1. It is the same reason why the relative speedup is not the same as the number of created threads. In addition, threads are communicating with each other and accessing to the same memory, because inputs and outputs are defined as shared variables.

Table 3 shows the measured times that are needed to create a new threads. On FIR filter and DFT algorithm, it is created only once. When program compute FFT, the parallel region is created regularly depended on length of input array.

The shape of the graphs representing performance increases of TI DSP are as was expected. It is increasing along the both axes.

Situation with Intel CPU is different. The characteristics do not grow with every increase of input parameters. Expected behaviour is shown on figure 6 where performance is increasing with length of input signal, but the maximum is when the program runs on 4 threads, which is number of physical cores. The decrease when program runs on 5 and more threads can be caused by switching between states of logical cores for one physical unit.

Similar effect is also on figures 4 and 5. The local maximums are on line, when 4 threads are created. After changing the number of threads to 5, the slight decrease can be observed, but then the performance continues in growing. Really unexpected is behaviour on figure 5 where global maximum is with small size of input data.

TABLE III. TIME NEEDED TO CREATE PARALLEL REGION

Number of threads	Core i7-2670QM	TMS320C6678
1	0.371 µs	17 µs
2	0.446 µs	34 µs
3	0.518 µs	36 µs
4	0.663 µs	39 µs
5	0.705 µs	42 µs
6	0.734 µs	45 µs
7	0.776 µs	48 µs
8	0.793 µs	52 µs

VI. CONCLUSION

In this paper, the behaviour of different parallel algorithms was shown on different platforms. If the computing is very simple or the length of processed data is short, it does not worth it to parallelize the loops. It is because the time required for creating threads and time while these threads communicate with each other can be approximately the same or bigger than the execution time of the actual time of calculation. Next what should be considered is the usage hyper-thread technology, because some algorithms can be degraded.

REFERENCES

[1] The OpenMP API specification for parallel programming [online]. 2013 [Cited 26. 2. 2013]. <http://www.openmp.org/>.

[2] BLAISE, Barney. OpenMP [online]. 2013 [Cited 26. 2. 2013]. <https://computing.llnl.gov/tutorials/openMP/>.

[3] JAN, Jiří. Číslicová filtrace, analýza a restaurace signálů. 2nd ed. Brno: VUTIUM, 2002. ISBN 80-214-1558-4.

[4] INTEL CORPORATION. 2nd Generation Intel Core Mobile Processor Datasheet, Vol. 1 [online]. 2012 [Cited 7. 1. 2013]. <http://www.intel.com/content/www/us/en/processors/core/2nd-gen-core-family-mobile-vol-1-datasheet.html>.

[5] TEXAS INSTRUMENTS. TMS320C6678 Multicore Fixed and Floating-Point Digital Signal Processor [online]. 2012 [Cited 7. 1. 2013]. <http://www.ti.com/lit/gpn/tms320c6678>.

[6] INTEL CORPORATION. Intel® Core i7-2600 Mobile Processor Series [online]. 2012 [Cited 27. 2. 2013]. <http://download.intel.com/-support/processors/corei7/sb/core i7-2600_m.pdf>