Using Parallel Signal Processing in Real-Time Audio Matrix Systems

JIRI SCHIMMEL
Department of Telecommunications FEEC
Brno University of Technology
Purkynova 118, 612 00 Brno
CZECH REPUBLIC
schimmel@feec.vutbr.cz http://www.utko.feec.vutbr.cz/en/

Abstract: - The paper deals with design and performance analysis of algorithms that utilize parallel signal-processing methods and SIMD technology for multiply-and-add algorithm for digital audio signal processing. This algorithm is used for summing the gained input signals on output buses in applications for distributing, mixing, effect-processing, and switching multi-format digital audio signal in an audio signal network on desktop processors platforms. The subjective evaluation of latency caused by principle of the real-time digital audio processing is also studied in the paper Results of an analysis of speed-up and real-time performance of several summing algorithms are presented in the paper as well as subjective evaluation of the latency depending on the audio buffer size.

Key-Words: - Parallel processing, Parallel algorithms, Audio systems, Optimization methods, SIMD, Digital audio processing

1 Introduction

Embedded PCs with desktop processors are versatile, flexible and cost effective solutions for distributing, mixing, effect-processing, and switching multi-format digital audio signals in an audio signal network. Each available input and output can be processed by several audio processing algorithms, and each input signal can be sent to each output.

In an audio signal processing system with many input and output channels, an algorithm performing the summation of input signals at output buses can have high computing demands. As a result, the real-time performance of the system decreases because the summing algorithm restrains the computing power available for incorporated digital audio effects for the processing of input and output signals.

The summing algorithm is a simple loop of iterations that can run simultaneously without interfering with each other. This makes the summing algorithm a perfect tool for parallel processing, e.g. for *parallel for* or *parallel reduce* approaches. However, a parallel loop is generally useful for large-scale vectors and matrixes because it incurs overhead cost for every chunk of work that it schedules. If the chunks are too small, the overhead may exceed the useful work [1]. However, the audio signal buffers must be short for real-time processing, so the parallel processing efficiency may be reduced to nothing by its own overhead.

2 Optimizing the Cross-Point Matrix

Part of an audio signal flow diagram of a digital audio mixing application that performs summing input signals on output buses is often called the *cross-point matrix*. It is a simple multiply-and-add algorithm performed independently on each sample of each input channel for each output channel. It makes the summing algorithm perfect for parallel processing.

2.1 Multiply-and-Add Algorithm

The most time-consuming algorithm, which the cross-point matrix performs, can be expressed using the equation

$$y_o[n] = \sum_{i=0}^{L-1} F_{io}\, g[n] x_i[n] \quad \text{for } o = 0,1,...,M-1, \qquad (1)$$

where $y_o[n]$ is signal of the output bus o, $x_i[n]$ is signal of the input channel i, $g[n]$ is the time-dependent gain function, L is the number of input channels, M is the number of output buses, and

$$F_{io} = \begin{cases} G_{io} & \text{for } pfl_{io} = 0 \\ 1 & \text{for } pfl_{io} = 1, \end{cases} \qquad (2)$$

where G_{io} is the constant gain factor of a given cross point, and pfl_{io} is the pre-fade setting of a given cross point. The $g[n]$ function represents a soft-switch function that uses the fast-fade envelope generator to prevent clicks at step changes of the gain (e.g. pre/post fader switch, mute on/off). In a multi-format digital audio network, no input or output bus is monophonic; it consists of several audio channels (see Fig. 1). Fortunately, every multi-channel node (cross point) can be expanded into a corresponding number of single-

channel nodes (see Fig. 2) if the pointers to audio signal buffers (in Figs 1 and 2 labelled as a, b,..., z, α, β,.., ω) of all buses (in Figs 1 and 2 labelled as A, B, C,..., Δ, Φ, Γ,...) are stored in pointer-to-pointer arrays. In that case, the pointers to audio signal buffers can be remapped accordingly only once – at an application start-up or in runtime when the bus configuration is changed.

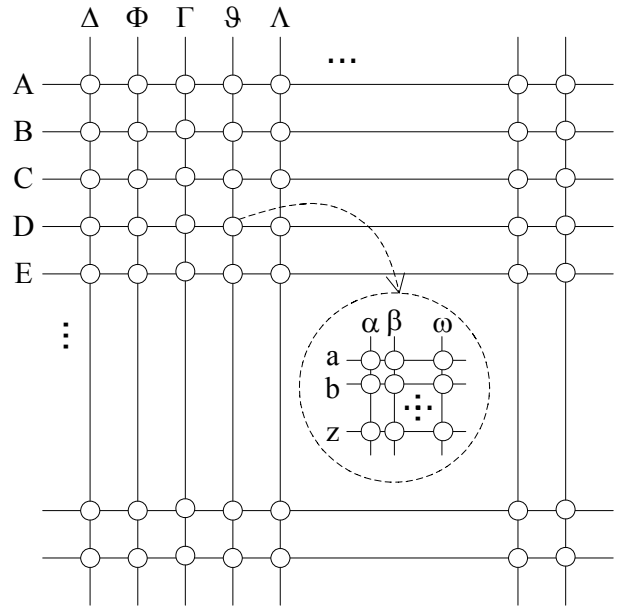

Fig. 1: Symbolic representation of a multi-format cross-point audio matrix.

The most power-consuming situation occurs when a scene is recalled – the summing algorithm must perform the soft-switch on all nodes in which the gain and/or pre/post settings were changed. Equation (1) can be expressed as

$$\left(y_0[n] \;\; y_1[n] \;\; \cdots \;\; y_{M-1}[n]\right) = g[n] \cdot \begin{pmatrix} F_{0,0} & F_{0,1} & \cdots & F_{0,L-1} \\ F_{1,0} & F_{1,1} & \cdots & F_{1,L-1} \\ & & \ddots & \\ F_{M-1,0} & F_{M-1,1} & \cdots & F_{M-1,L-1} \end{pmatrix} \cdot \begin{pmatrix} x_0[n] \\ x_1[n] \\ \vdots \\ x_{L-1}[n] \end{pmatrix}$$

or

$$\mathbf{y}[n] = g[n]\mathbf{F}\mathbf{x}[n], \tag{3}$$

where $\mathbf{y}[n]$ is the vector of output signals at time n, $\mathbf{x}[n]$ is the vector of input signals at time n, $g[n]$ is time-dependent gain function, $g[n]$ is the time-dependent gain function, and $\mathbf{F}$ is the matrix of node gains.

Some solutions of matrix multiplications using parallel processing for the distributed systems and multi-core processors are presented in [2] and [3]. However, equation (3) is special simpler case.

The most direct implementation of equation (3) in the C++ programming language uses nested loops like this:

```
for (int o = 0; o < M; ++o)
  for (int i = 0; i < L; ++i)
    for (int n = 0; n < N; ++n)
      y[o][n] += F[i][o]*g[n]*x[i][n];
```

where N is the number of samples of processed audio signal buffers. However, this is a less effective solution. It is possible to use more effective implementations using one-, two- or three-dimensional scalable parallelism. The speed-up of all implementations mentioned below will be with respect to this nested-loop implementation.

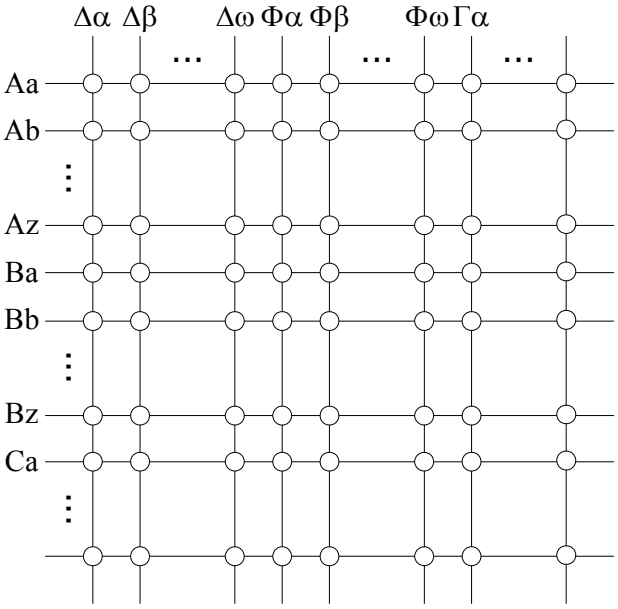

Fig. 2: Symbolic representation of a multi-format cross-point audio matrix with remapped audio channels.

The algorithm is performed independently on each sample of each input channel for each output channel, i.e. the dependencies are not irregular, which simplifies the parallelization. For Transformations techniques for extracting parallelism in irregular nested loops are described in [4].

2.2 Testing Conditions

The testing of the implementation methods was performed on three desktop computers with the Windows XP® operating system (see Tab. 1).

Tab. 1: Testing computers

PC1	Intel® Core™2 T7200 @ 2 GHz
PC2	Intel® Core™2 Quad CPU Q9400 @ 2.66 GHz
PC3	Intel® Core™2 6600 @ 2.4 GHz

All unnecessary operating system services were disabled to minimize the influence of other running processes on the results. For the same reason, the minimal value of the processing time from ten tests was taken as the result. The Microsoft® Visual C++ 2005 was used to implement the testing Win32 console application.

The implementation methods were tested for the IEEE 754 single- and double-precision floating point data format [5]. All variables were defined as local and the overhead of the implementation was minimized.

The input signal buffers $x[n]$ were loaded with random numbers from the range of $\langle -1; 1\rangle$, gain matrix **F** was generated using random numbers from the range of $(0; 1\rangle$, and gain function $g[n]$ was implemented as a look-up table of linear function $g[n] = n/(N-1)$ for $n = 0, ..., N-1$, where N is the buffer length.

The generated random numbers were thresholded to eliminate the occurrence of denormalized numbers in the whole processing and thus reducing computing power of the processor [6]. Denormalized numbers represent an underflow condition and they are computed using the gradual underflow technique [7]. This causes that arithmetical operations with denormalized numbers are much slower than those with normalized numbers [7].

The processing time of each method was measured as the so-called *wall clock time* using the Intel® TBB template *tick_count* class. The resolution of *tick_count* corresponds to the highest resolution timing service on the platform that is valid across all threads in the same process [1].

The cross-point matrix for testing consists of 128 input channels and 128 output channels, which represents a large routing system. A theoretical situation of a scene switch with soft-fade change of all gains was simulated.

2.3 Implementation Methods

A speed-up of implementation methods described below was measured relatively to the wall clock time of the direct implementation described above, which uses nested loops. All compiler optimizations were disabled in order to minimize the compiler influence on the results. The influence of compiler optimizations and real-time performance of implementation methods will be discussed later. Fig. 3 shows the dependence of the speed-up of implementation methods described in the following paragraphs on the size of audio signal buffers for double-precision floating point data at PC1, Fig. 4 at PC2, and Fig. 5 at PC3 (see Tab. 1). Fig. 6 also shows the dependence of the speed-up of implementation methods on the size of audio signal buffers for single-precision floating point data at PC3.

2.3.1 Serial Implementation Using Iterators

The most common serial method of optimization of loops that decreases the algorithm overhead is using the C-language while statement and pointer iterators:

```
py=&y[o]; px=&x[i]; pF=&F[i][o]; pf=&f;
int n=N;
while(--i >= 0)
   *py++ = *pF * *pf++ * *px++;
```

The first method (in Figs 3 to 6 labelled as "Serial") implements all three loops using this method.

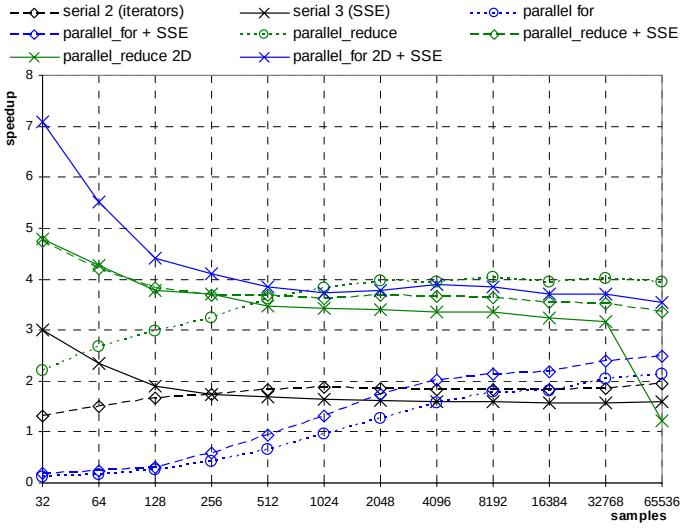

Fig. 3: Speed-up of the cross-point matrix implementation methods for a double-precision floating point data at PC1.

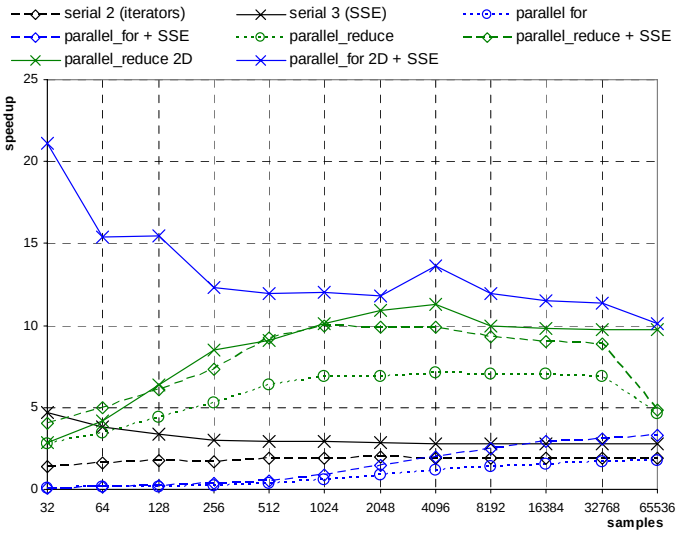

Fig. 4: Speedup of the cross-point matrix implementation methods for a double-precision floating point data at PC2.

2.3.2 Serial Implementation Using SSE

The advantage of the Streaming SIMD Extension [7] implementation is obvious at the first sight. Its performance analysis is presented e.g. in [8]. However, loading and storing the vector data types from and into floating-point unit registers have to be performed with each audio signal sample if we want to use the cross-point matrix algorithm for other-party audio technologies, for example Audio Streaming Input/Output [9] or Virtual Studio Technology [10], which uses the floating-point unit data types. Such overhead reduces the efficiency of the SIMD implementation (in Figs 3 to 6 labelled as "Serial SSE").

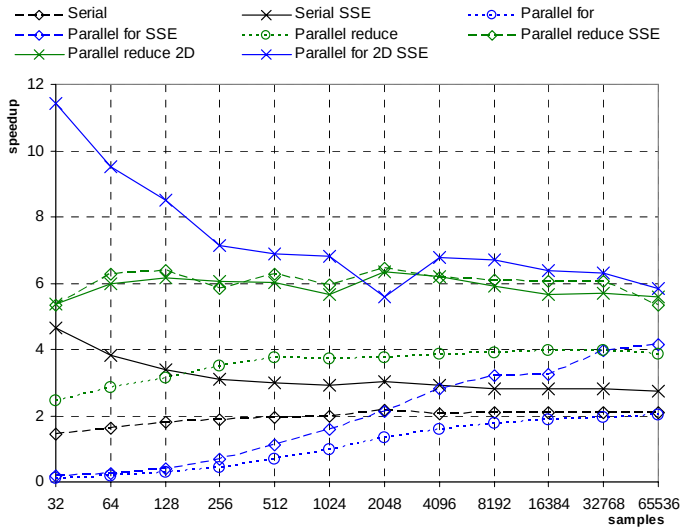

Fig. 5: Speed-up of the cross-point matrix implementation methods for a double-precision floating point data at PC3.

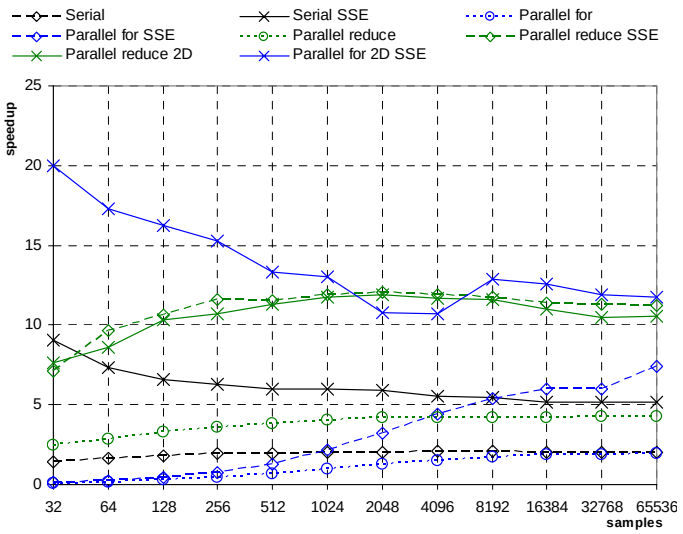

Fig. 6: Speedup of the cross-point matrix implementation methods for a single-precision floating point data at PC3.

2.3.3 Parallel For Implementations

The iteration space of the input buffer index *i*, output buffer index *o* and number of sample frame *n* goes from 0 to $L-1$, $M-1$, or $N-1$, respectively. The Intel® Threading Building Blocks *parallel_for* template [11] was used for one-dimensional division of one of the iteration spaces into chunks and for running each chunk on a separate thread.

The first method (in Figs 3 and 4 labelled as "Parallel for") breaks the *n* iteration space (sample frames) and uses the serial C-language while statement implementation described above for outer loops (output and input buffer indexes).

We can combine the parallel for implementation with the SIMD extension. However, we have to analyze influence of the overhead of both methods. The methodology of combining two types of parallel processing mechanism and study of the performance is presented in [12]. The second method (in Figs 3 to 6 labelled as "Parallel for SSE") uses both approaches and breaks the *i* iteration space (input buffer indexes) and uses the SSE2 implementation for the inner loop (sample frames) and serial C-language while statement implementation for the outer loop (output buffer indexes). The *parallel_for* construct incurs an overhead cost for every chunk of work that it schedules [1]. So the Intel® TBB templates allow controlling the grain size of parallel loop. To ensure sufficient system bandwidth between the processor and the memory, an automatic grain size optimized for cache affinity [11] was chosen. The influence of explicitly defined grain sizes on the algorithm performance will be discussed later.

2.3.4 Parallel Reduce Implementations

The cross-point matrix algorithm is also a typical example of algorithm suitable for the parallel split/join approach [1]. The Intel® TBB *parallel_reduce* template generalizes any associative operation by splitting the iteration space into chunks and performing summation of each chunk on a separate thread. The join method performs the corresponding merges of the results [11].

In the cross-point matrix algorithm, the addition of samples is performed over the *i* iteration space (input buffer index). The first implemented method (in Figs 3 to 6 labelled as "Parallel reduce") breaks the *i* iteration space and uses the serial C-language while statement implementation for the outer loop (output buffer indexes) and the inner loop (sample frames) as well. Second method (in Fig. 3 and 4 labelled as "Parallel reduce SSE") breaks the *i* iteration space and uses the SSE2 implementation for the inner loop and serial C-language while statement implementation for the outer loop. Automatic grain-size optimized for the cache affinity is used again.

2.3.5 Implementations with 2D Iteration Spaces

The iteration spaces *i*, *o*, and *n* are independent and computations can run simultaneously without interfering with each other. So we can break the whole iteration space into two-dimensional chunks. The Intel® TBB *blocked_range2d* template class [11] represents recursively divisible two-dimensional half-open interval. Each axis of the range has its own splitting threshold [1].

The first implemented method (in Figs 3 to 6 labelled as "Parallel reduce 2D") breaks the *n* and *i* iteration spaces (sample frames and input buffer indexes), performs partial summation on each chunk, and uses the serial C-language while statement implementation for the outer loop (output buffer indexes). This implementation method does not use the SSE2 on purpose so that it could be compared with the "Parallel reduce SSE" implementation.

The second implemented method (in Figs 3 to 6 labelled as "Parallel for 2D SSE") breaks the *o* and *i* iteration spaces (buffer indexes) and uses the SSE2 implementation for the inner loop (sample frames).

2.4 Real-Time Algorithm Performance

Only the speed-up of the implementation methods was discussed in the previous section. However, the most important thing from the practical viewpoint is whether the implementation is able to work in real-time with the double-buffering technique. It means that computing all buffer samples by the cross-point matrix algorithm must be finished before next buffers are recorded. The time for computing all buffer samples *t* must fulfil the condition

$$t < NT, \tag{4}$$

where *T* is the sampling period and *N* is the number of buffer sample frames.

Compiler optimization is therefore used to optimize the method overhead and also the algorithm itself. Intel compiler 10.0.654 and Microsoft® Visual C++ 2005 compiler were used with the following settings:

Microsoft® Visual C++ 2005 compiler:
- Maximize speed
- Enable Intrinsic functions
- Whole program optimization

Intel® compiler 10.0.654:
- Maximize Speed
- Enable Intrinsic Functions
- Global Optimization
- Use Intel® Processor Extensions: P4 SSE3
- Enable Parallelization
- Floating-Point Speculation: Fast
- Flush Denormal Results to Zero: No
- Floating-Point Precision Improvement: None

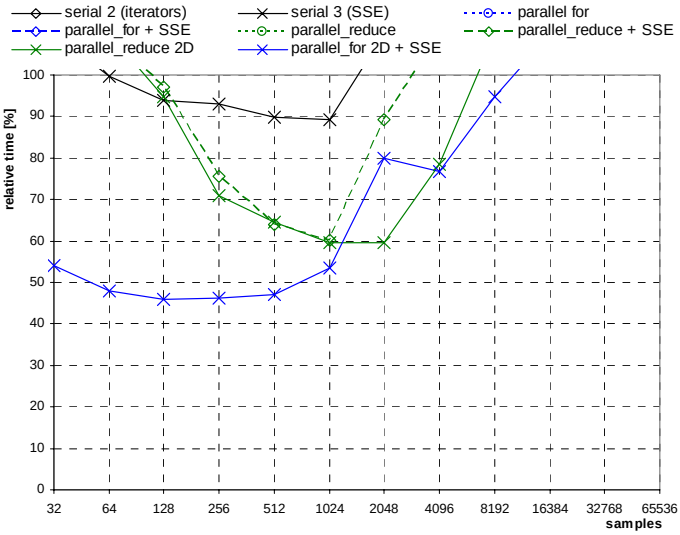

Fig. 7: Relative time of computing cross-point matrix algorithm for double-precision floating point numbers at PC1 with Microsoft® Visual C++ 2005 compiler.

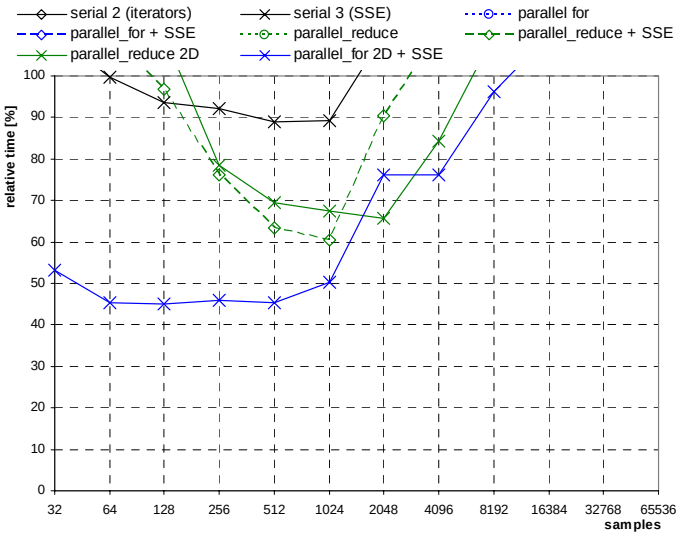

Fig. 8: Relative time of computing cross-point matrix algorithm for double-precision floating point numbers at PC1 with Intel® compiler 10.0.654.

Figs 5 to 12 show the dependence of the relative time *t*/*NT* in percent points on the number of buffer sample frames for double-precision floating point data and sampling frequency of 48 kHz at PC1, PC2, and PC3 (see Tab. 1) when Microsoft® Visual C++ 2005 compiler and Intel® compiler 10.0.654 is used. Figs 13 and 14 also show the dependence for single-precision floating point data at PC3. Relative times for sampling frequency of 96 kHz will be doubled in all cases.

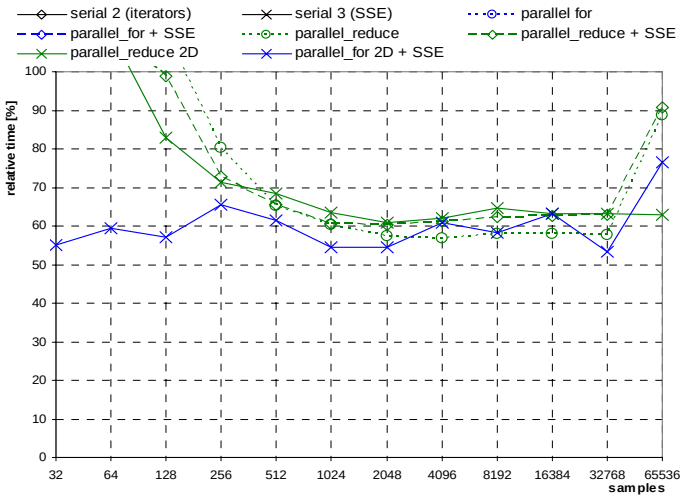

Fig. 9: Relative time of computing cross-point matrix algorithm for double-precision floating point numbers at PC2 with Microsoft® Visual C++ 2005 compiler.

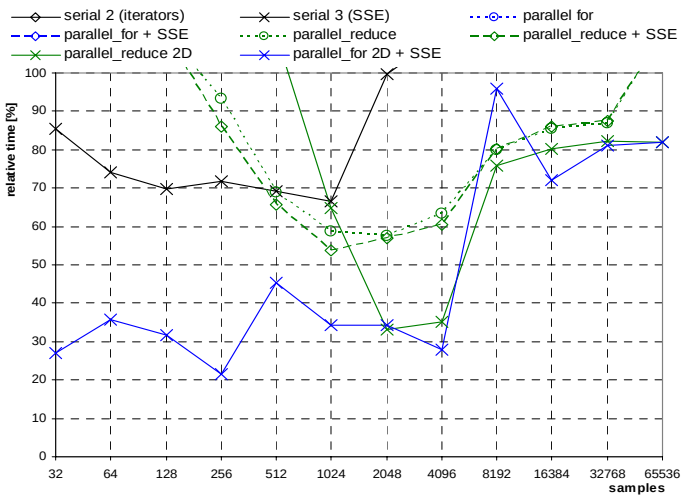

Fig. 10: Relative time of computing cross-point matrix algorithm for double-precision floating point numbers at PC2 with Intel® compiler 10.0.654.

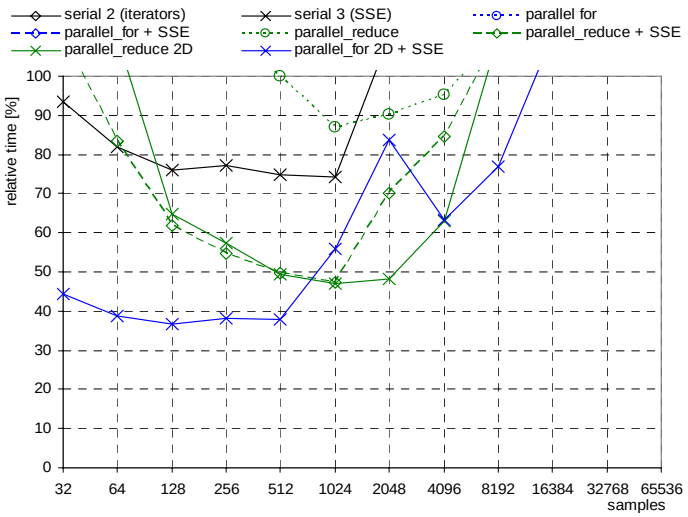

Fig. 11: Relative time of computing cross-point matrix algorithm for double-precision floating point numbers at PC3 with Microsoft® Visual C++ 2005 compiler.

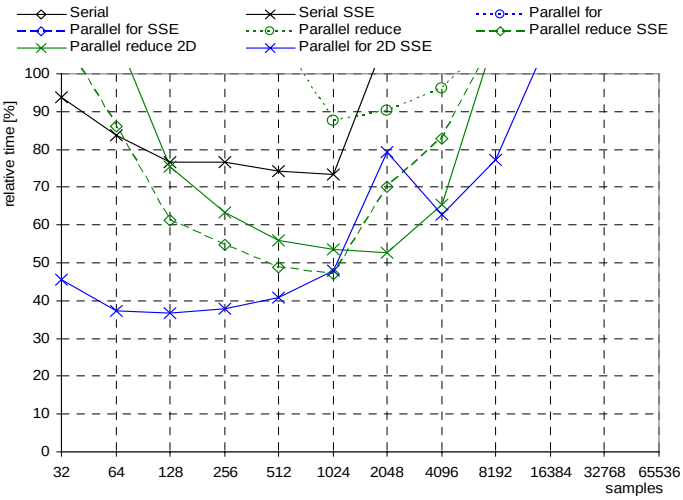

Fig. 12: Relative time of computing the cross-point matrix algorithm for double-precision floating point numbers at PC3 with Intel® compiler 10.0.654.

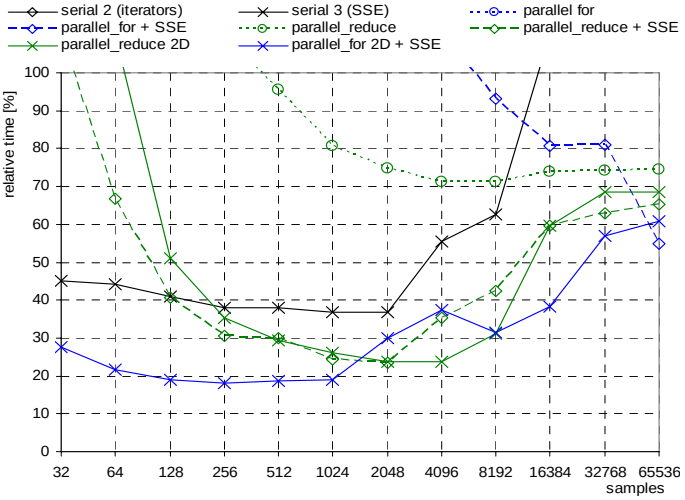

Fig. 13: Relative time of computing cross-point matrix algorithm for single-precision floating point numbers at PC3 with Microsoft® Visual C++ 2005 compiler.

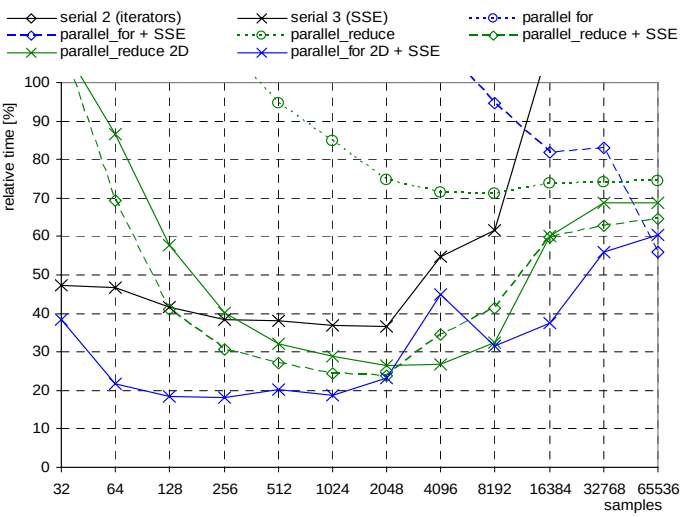

Fig. 14: Relative time of computing the cross-point matrix algorithm for single-precision floating point numbers at PC3 with Intel® compiler 10.0.654.

3 Subjective Evaluation of Latency

This chapter will discuss selection of the buffer size for real-time audio processing with the double-buffering technique from a musician point of view.

3.1 Real-Time Audio Processing Latency

If an in-place processing of audio signal buffer is used, an audio signal latency t_L is introduced between an input and output

$$t_L = NT + Nt_1, \tag{5}$$

where T is the sampling period, N is the number of the buffer sample frames, and t_1 is an average time of processing of one sample frame. Professional real-time audio technologies for desktop computers, for example Audio Streaming Input/Output [9] from Steinberg, use one callback for synchronous buffer switching for input and output buffers. The audio signal latency in such systems is not dependent on computing demands of an algorithm and it equals to

$$t_L = 2NT\ . \tag{6}$$

For real-time audio processing we try to keep the latency as small as possible in all its applications. Live music performance is the application, which is most sensitive to the latency introduced by the real-time digital audio processing.

3.2 Latency in Live Music Performance

If we want to find a maximum latency of real-time audio processing system that is acceptable for a live music performance, we have to investigate the time gap between physical stimulus of sound and its sensation. In a live music performance, the stimulus is action of a musician (hitting drum, pressing key on keyboard, plucking string, etc.) and the sensation is musician's perception of the sound. There are several factors that forms the overall sound latency in real-time processing of digital audio signal. These are shown in Fig. 15.

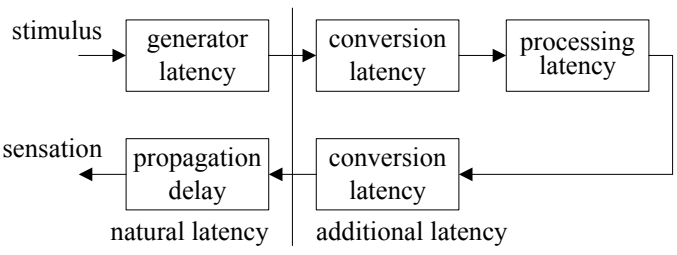

Fig. 15: Overall sound latency in real-time processing of digital audio signal of live electronic music performance.

Latency, which can be named *natural*, consists of latency of a sound generator and of a delay caused by propagation of sound waves from loudspeaker to musician. The former one is typical for electronic music instruments and it is caused by group delay of incorporated audio signal processing algorithms and by latency of control signals. The farther one is about 3 ms per one meter, it means up to 9 ms for a typical distances of floor monitors at the concert stage.

The real-time processing of audio signal introduces *additional* latency, that consists of a processing latency defined by (5) or (6) and conversion latency that is mostly determined by internal buffering in used audio interface. The internal buffering is typical for audio interfaces that use isochronous serial buses. They have to perform the internal buffering when small processing buffers are used by an application in order to synchronize digital audio stream with isochronous timeslots.

3.3 Experiment Design and Results

Two experiments were performed with fifteen musicians playing on electronic instruments in order to find minimum size of the additional latency that is not perceived by the musician. Electronic drums Roland V-Drums, electric guitar, electronic wind instrument AKAI EWI 4000s, and MIDI master keyboard with Roland XV 5050 sound module were played.

Specially designed console application for Windows XP was used to delay the audio signal between input of a PC audio interface and its output. The audio interface used for the experiment allowed minimum additional latency of 4 ms at sampling frequency of 96 kHz and operator could increase this latency by steps of one sample frame, i.e. 10 μs. Headphones were used in order to eliminate the sound propagation delay. The responders were asked to accompany a piece of jazz music with rhythm of 90 BPM.

In first experiment, the hidden reference and anchor methodology was used. Responders were asked whether they can or cannot recognize time gap between their action and perception of the sound in two cases. In one case, the audio signal was delayed with additional latency set by the operator. In the other case, responders were listening to a reference signal with no additional latency (audio interface was bypassed to eliminate conversion delay). The responders did not know in which case they listen to the reference signal and the delayed signal.

In second experiment, the responders was simply asked to set-up such amount of the additional latency, that starts to be uneasy for their performance, e.g. they start to have problems to follow the rhythm of the song that they accompany.

Actual value of the additional latency was measured afterwards using a sweep-sine generator and an audio analyzer. It was determined as mean value of measured group delay. The generator output was connected instead of the electronic instrument output and the analyzer input was connected to the headphone amplifier output. Difference between theoretical value of the additional latency computed from buffer sizes and user delay and its actual measured value was about 0.3 ms and thus negligible.

Fig. 16 to 18 show results of both experiments for all electronic instruments used in the experiment. The box has lines at the lower quartile, median, and upper quartile values. Lines extending from each end of the box show the extent of the rest of the data.

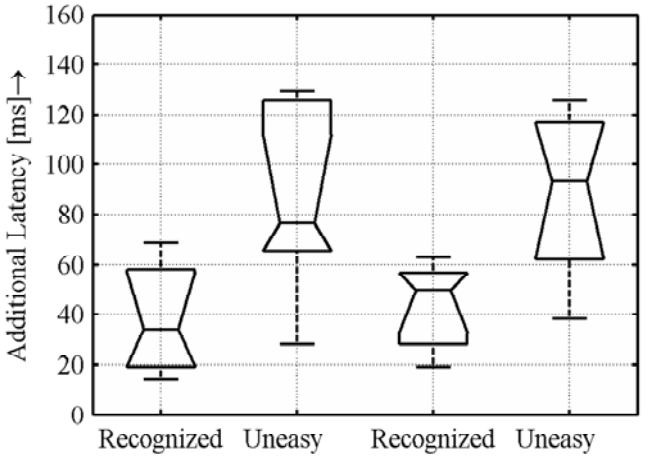

Fig. 16: Perception of additional latency for clean (left) and metal (right) electric guitar.

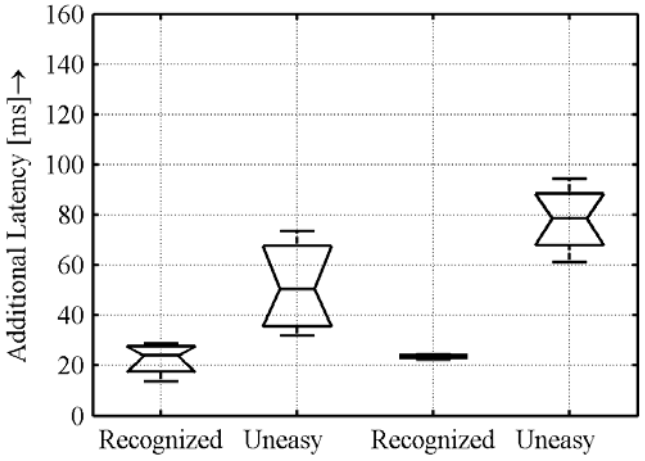

Fig. 17: Perception of additional latency for electronic drums (left) and electronic keyboard (right).

It can be seen that low boundary of just-recognized additional latency is about 20 ms for almost all instruments. Furthermore, the minimum value and the range of just-recognized additional latency depend on the instrument type, strictly speaking on the playing style. For example, minimal value for electrical violin was recognized for a pizzicato. It can be seen from Fig. 17 that variance of just-recognized additional latency for electronic keyboard players is minimal.

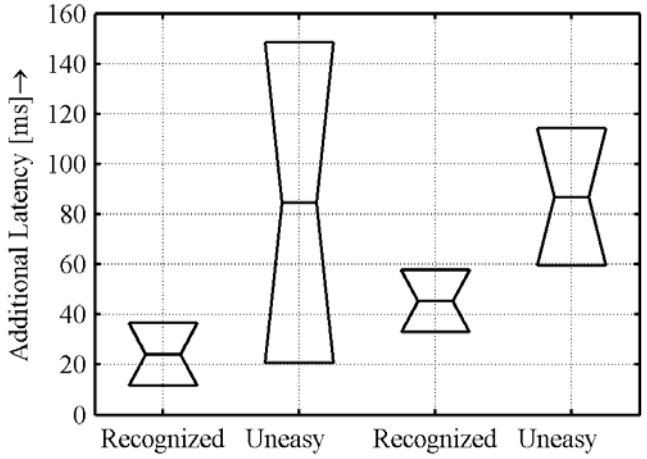

Fig. 18: Perception of additional latency for electronic wind instrument (left) and electronic violin (right).

Since the headphones were used during experiments, the sound propagation delay about 9 ms has to be subtracted from the results in for typical distances of floor monitors at the concert stage or studio.

According to these results, it can be recommended to use maximum audio buffer size for the real-time audio processing with the double-buffering technique from (6); it corresponds to 528 sample frames at sampling frequency of 96 kHz, 264 sample frames at sampling frequency of 48 kHz, and 242 sample frames at sampling frequency of 44.1 kHz.

4 Analyzing Influence of Grain Size

The maximum recommended audio buffer size is determined by the subjective evaluation of additional latency introduced by the real-time digital processing of audio signals, as mentioned in previous chapter.

The minimum recommended audio buffer size can be determined according to the results of analysis of real-time algorithm performance presented in paragraph 2.4. It can be seen form Figs 7 to 14 that both serial SSE implementation and *parallel_for* implementation using SSE and 2D iteration space have best real-time performance for the audio buffer sizes from 64 sample frames. The rest of methods have significant improvement of the real-time performance from the buffer sizes from 128 sample frames.

According to this, it can be recommended to use the buffer sizes of 128 and 256 sample frames for all implementation methods and sampling frequencies up to 48 kHz. The buffer size of 512 sample frames can be used for sampling frequency of 96 kHz and buffer size

of 1024 sample frames for sampling frequency of 192 kHz as well. For further speedup of the parallel implementation methods, we can analyze influence of the grain size for these four buffer sizes.

Figs 19 to 22 show the influence of explicitly defined grain size on the speed-up of the parallel implementation methods for double-precision floating point data type at PC1 (see Tab. 1).

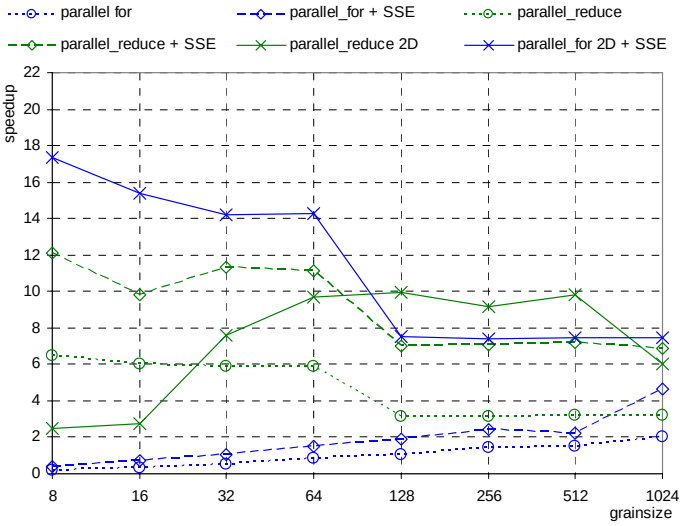

Fig. 19: Influence of grain size on the speed-up of the parallel implementation methods for a buffer size of 1024 sample frames at PC1.

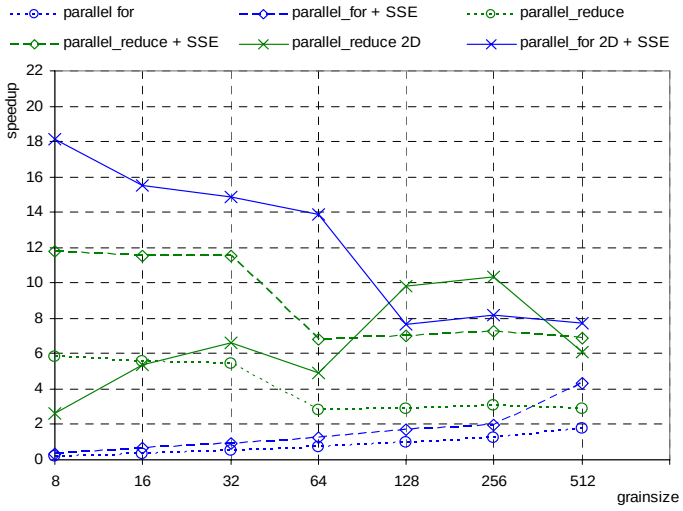

Fig. 20: Influence of grain size on the speed-up of the parallel implementation methods for a buffer size of 512 sample frames at PC1.

The two-dimensional *parallel_for* method gives in most cases better results for grain sizes up to 128 sample frames while the two-dimensional *parallel_reduce* method gives better results for larger grain sizes.

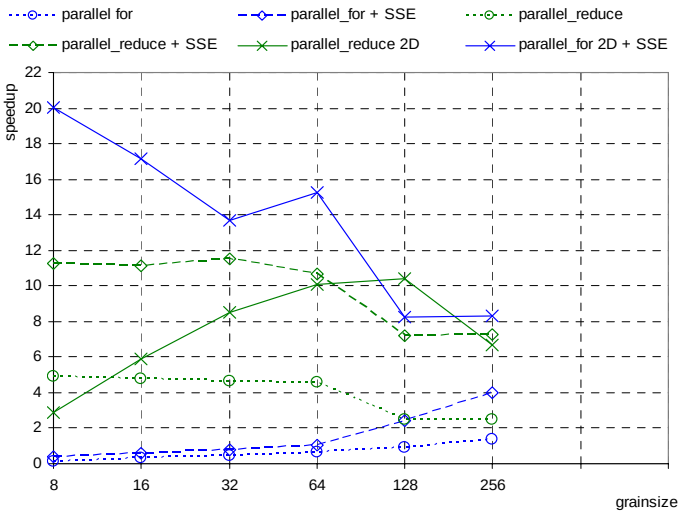

Fig. 21: Influence of grain size on the speed-up of the parallel implementation methods for a buffer size of 256 sample frames at PC1.

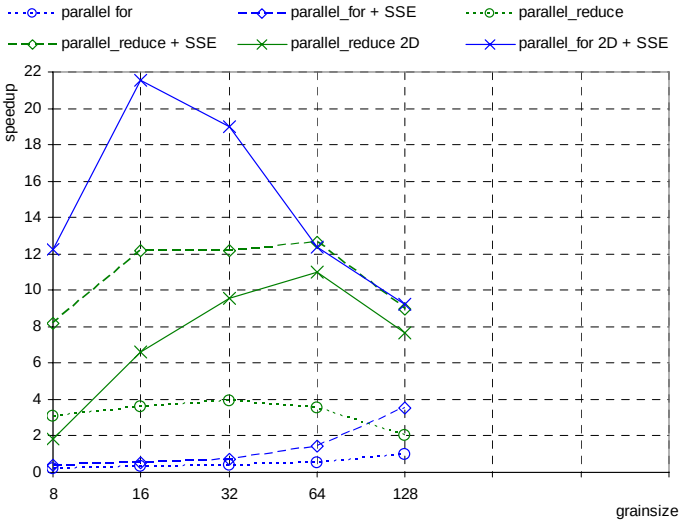

Fig. 22: Influence of grain size on the speed-up of the parallel implementation methods for a buffer size of 128 sample frames at PC1.

5 Conclusion

It can be seen from Figs 3 to 6 that the *parallel_for* approach using two-dimensional iteration space and SSE has the highest speed-up for almost all buffer sizes for both single-precision and double-precision data types. It is the fastest method for small buffer sizes. Its performance is comparable to methods of the two-dimensional *parallel_reduce* and one-dimensional *parallel_reduce* with SSE. These two implementation methods have almost a constant speed-up for all buffer sizes. The two-dimensional *parallel_for* method gives in most cases better results for explicitly defined grain sizes up to 128 sample frames while the two-dimensional *parallel_reduce* method gives better results for larger grain sizes.

Figs 7 to 14 show that not all implementation methods are suitable for real-time processing. The *parallel_for* method using two-dimensional iteration space and SSE has the best performance for real-time processing at all testing desktop computers but it can be used only for buffer sizes up to 8 kB when the double-precision floating-point data type is used.

Together with results of subjective evaluation of the additional latency in live music performance, it can be recommended to use the audio buffer size of 128 and 256 sample frames for sampling frequencies up to 48 kHz. The buffer size of 512 sample frames can be used for sampling frequency of 96 kHz or higher and buffer size of 1024 sample frames for sampling frequency of 192 kHz.

Acknowledgement

The paper was prepared within the framework of project no. 102/07/P505 of the Czech Science Foundation and project no. FT-TA3/010 of the Ministry of Industry and Trade of the Czech Republic.

References:

[1] *Intel® Threading Building Blocks Tutorial.* Document Number 319872-002US, 2009-Jun-25.

[2] Nakhoon Baek, Hwanyong Lee, "Parallelized Matrix Multiplications for the Multi-Core CPU's". *WSEAS TRANSACTIONS on COMPUTERS*, Issue 12, Volume 6, December 2007, pp. 1168-1173. ISSN 1109-2750.

[3] Muhammad Hafeez, Dr. Muhammad Younus, Abdur Rehman, Athar Mohsin, „Optimal Solution to Matrix Parenthesization Problem Employing Parallel Processing Approach". In *Proceedings of the 8th WSEAS International Conference on Evolutionary Computing*, Vancouver, British Columbia, Canada, June 19-21, 2007, pp. 235 – 240, ISSN: 1790-5095, ISBN: 978-960-8457-75-1

[4] Fawzy A. Torkey, Afaf A. Salah, Nahed M. El Desouky, Sahar A. Gomaa, "Transformations Techniques for extracting Parallelism in Non-Uniform Nested Loops". *WSEAS TRANSACTIONS on COMPUTERS*, Issue 9, Volume 7, 2008, pp. 1394 – 1404. ISSN 1109-2750.

[5] *IEEE 754-2008, Standard for Floating-Point Arithmetic*, Aug. 29 2008. ISBN: 978-0-7381-5753-5

[6] Laurent de Soras, *Denormal Numbers in Floating Point Signal Processing Applications* [online]. 2002. http://www.musicdsp.org/files/denormal.pdf

[7] *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1: Basic Architecture*. Intel Corporation, 2008.

[8] Y.F. Fung, M.F. Ercan, W.L. Cheung, T.K. Ho, C.Y. Chung, G. Singh, "Performance analysis of PC based SIMD parallel mechanism". *WSEAS TRANSACTIONS on COMPUTERS*, Issue 2, Volume 2, April 2003, pp. 295 – 298. ISSN 1109-2750

[9] *Audio Streaming Input/Output Software Developer Kit 2.1*. Steinberg Media Technologies GmbH. 2005.

[10] *VST Plug-Ins Software Developer Kit 2.4*. Steinberg Media Technologies GmbH. 2006.

[11] *Intel(R) Threading Building Blocks Reference Manual*. Document Number 315415-003US, 2009-Jul-3

[12] Yu-Fai Fung, Wai-Leung Cheung, Gujit Singh, Muhammet F. Ercan, "An Empirical Study of Bi-level Parallel Computing on a PC". In *Proceedings of the 2nd WSEAS Int. Conf. on ELECTRONICS, CONTROL and SIGNAL PROCESSING (ICECS '03)*, Singapore, December 7 – 9, 2003. ISBN 960-8052-91-2