Optimizing Digital Audio Cross-Point Matrix for Desktop Processors Using Parallel Processing and SIMD Technology

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 optimizing the performance of a 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 paper presents results of an analysis of speed-up and real-time performance of several summing algorithms, which use parallel processing with Intel® Threading Building Blocks (TBB) and Streaming SIMD Extension 2 (SSE2) technologies.

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 matrices 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

\[ y_o[n] = \sum_{i=0}^{L-1} F_{io} g[n] x_i[n] \]  \text{for } o = 0,1,\ldots,M - 1 ,

(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

\[ F_{io} = \begin{cases} G_{io} & \text{for } pfl_{io} = 0 \\ 1 & \text{for } pfl_{io} = 1 , \end{cases} \]

(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.

![Diagram](image)

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

\[
\begin{pmatrix}
 y_0[n] \\
 y_1[n] \\
 \vdots \\
 y_{N-1}[n]
\end{pmatrix}
= g[n] 
\begin{pmatrix}
 F_{00} & F_{01} & \cdots & F_{0L-1} \\
 F_{10} & F_{11} & \cdots & F_{1L-1} \\
 \vdots & \vdots & \ddots & \vdots \\
 F_{N-1,0} & F_{N-1,1} & \cdots & F_{N-1,L-1}
\end{pmatrix}
\begin{pmatrix}
 x_0[n] \\
 x_1[n] \\
 \vdots \\
 x_{N-1}[n]
\end{pmatrix}
\]

or

\[
y[n] = g[n] F x[n],
\]

where \( y[n] \) is the vector of output signals at time \( n \), \( x[n] \) 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 \( F \) is the matrix of node gains.

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

```c++
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.

![Diagram](image)

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

2.2 Testing Conditions

The testing of the implementation methods was performed on three desktop computers with the Windows XP® operating system (Intel® Core™ T7200 @ 2 GHz, Intel® Core™2 Quad CPU Q9400 @ 2.66 GHz, and 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 [2]. 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 \((-1; 1)\), gain matrix \( F \) was generated using random numbers from the range of \((0; 1)\), 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 [3] in the whole processing. Denormalized numbers represent an underflow condition and they are computed using the gradual underflow technique [4]. This causes that arithmetical operations with denormalized numbers are much slower than those with normalized numbers [4].

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. Figs 3 and 4 show the dependence of the speed-up of implementation methods described in the following paragraphs on the size of audio signal buffers for desktop computer with Intel® Core™ 2 6600 @ 2.4 GHz processor.

#### 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:

```c
int n=N;
while(--i >= 0) {
    *py++ = *pF * *pf++ * *px++;
```

The first method (in Figs 3 and 4 labelled as “Serial”) implements all three loops using this method.

#### 2.3.2 Serial Implementation Using SSE

The advantage of the Streaming SIMD Extension [4] implementation is obvious at the first sight. 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 [5] or Virtual Studio Technology [6], which uses the floating-point unit data types. Such overhead reduces the efficiency of the SIMD implementation (in Figs 3 and 4 labelled as “Serial SSE”).

#### 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 [7] 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). The second method (in Figs 3 and 4 labelled as “Parallel for SSE”) 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 [7] 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 [7].

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 and 4 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 [7] 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 and 4 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 and 4 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 fulfill the condition \( t < NT \), 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. 5: Relative time of computing the cross-point matrix algorithm for double-precision floating point numbers (Intel compiler).](image)

![Fig. 6: Relative time of computing the cross-point matrix algorithm for double-precision floating point numbers (MS compiler).](image)
Figs 5 to 8 show the dependence of the relative time \( t/NT \) in percent points on the number of buffer sample frames for a sampling frequency of 48 kHz.

Fig. 7: Relative time of computing the cross-point matrix algorithm for single-precision floating point numbers (Intel compiler).

Fig. 8: Relative time of computing the cross-point matrix algorithm for single-precision floating point numbers (MS compiler).

4 Conclusion

Results of the performance analysis obtained with the other two desktop computers are similar to the results obtained with the Intel® Core™2 6600 @ 2.4 GHz processor presented in this paper. It can be seen from Figs 3 and 4 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.

It can be seen from Figs 5 to 8 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 but it can be used only for buffer sizes up to 8 kB when the double-precision floating-point data type is used.

Fig. 9 shows the influence of explicitly defined grain size on the speed-up of the parallel implementation methods for a buffer size of 1024 sample frames, double-precision floating point data type, and no compiler optimization. The two-dimensional parallel_for method gives better results for smaller grain sizes while the two-dimensional parallel_reduce method gives better results for larger grain sizes.

Fig. 9: Influence of grain size on the speed-up of the parallel implementation methods for a buffer size of 1024 sample frames.

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: