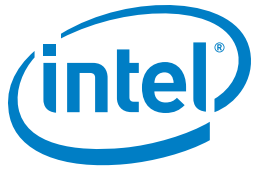


Signal Processing on Intel® Architecture: Performance Analysis using Intel® Performance Primitives

White Paper

Intel® Advanced Vector Extensions (Intel® AVX)

Signal Processing

Embedded Computing

Engineers can quickly determine whether Intel® processor-based platforms with Intel® Advanced Vector Extensions (Intel® AVX) satisfy signal processing requirements

Signal processing functions have often required special-purpose hardware such as DSPs and FPGAs. However, recent enhancements to Intel® architecture processors are providing developers an alternative: execute signal processing workloads on an Intel® processor.

Signal processing on the latest Intel processors is now a viable option due to continued improvements in multi-core architectures. The increased parallelism from vector instructions, along with other continuing performance improvements, enables the efficient execution of data parallel workloads such as digital transforms and filters. Additionally, by consolidating signal processing functions with other workloads on a multi-core Intel processor, it is possible to save hardware cost, simplify the application development environment and reduce time to market. This approach can be applied to many applications in aerospace (radar, sonar), communications infrastructure (baseband processing, transcoding) and healthcare (medical imaging).

This paper describes an easy process that allows developers to quickly determine how fast 2nd generation Intel® Core™ i7-2710QE processor will execute their signal processing algorithms, based on performance data[1] that is relatively easy to obtain. Developers can complete the process in a straightforward manner, as demonstrated with two simple examples in this paper: fast convolution and amplitude demodulation. The paper concludes by reviewing some of the development tools available to developers to conduct their own evaluations.

Umberto Santoni

Platform Architect, Embedded Communications Group

Thomas Long

Software Engineer, Embedded Communications Group

Table of Contents

Why Intel® Architecture for Signal Processing 3

SIMD Instructions Enhanced By Intel® Advanced Vector Extensions (Intel® AVX) 3

The Process for Evaluating Signal Processing Performance 4

Signal Processing Performance Data 4

Overview of benchmark data 5

A) Forward and inverse Fast Fourier Transform (FFT) 5

B) 2D Complex to Complex FFT Throughput (GFLOPS/s and absolute time) 6

C) Filter Execution Times 7

D) Discrete Hilbert Transform 7

E) Discrete Cosine Transform 7

Speedup with Intel® Advanced Vector Extensions (Intel® AVX) 6

Two Signal Processing Workload Examples 8

Example 1: Fast Convolution using FFT 8

Example 2: Discrete Envelope Detection / Amplitude Demodulation 9

Floating Point Speeds Development 9

Development Tools Overview 14

Intel® C++ Compiler 14

Intel® Math Kernel Library (Intel® MKL) 14

Intel® Integrated Performance Primitives (Intel® IPP) 14

Intel® VTune™ Performance Analyzer 14

Intel® Application Debugger 14

Eclipse*-based Integrated Development Environment 14

Consider Intel® Architecture Processors for Signal Processing 14

Appendix A: Test Configuration 15

Acronyms 15

Why Intel® Architecture for Signal Processing

There is a natural tendency to assume that just about any signal processing application requires a DSP, FPGA or ASP. That's because traditionally, it was necessary to use specialized hardware in order to satisfy performance objectives. However, for hybrid designs utilizing a mix of specialized signal processing algorithms and a broader set of applications, implementing two separate computing architectures may pose some significant disadvantages, such as:

- Hardware and board space for two computing systems: higher product cost
- Multiple tool chains: additional technical training and project management complexity
- Multiple code bases: larger software management effort
- Power consumption for two computing systems: more expensive thermal design
- Intersystem communication: greater design complexity or possibility for bottlenecks
- Time to market: extra time needed to design, validate and integrate subsystems
- Two development teams: unique communication challenges (e.g., silos)

One alternative is to perform signal processing workloads on an existing Intel architecture processor in the system. Workload consolidation is a powerful concept that has been delivering significant payoffs in data-centers with respect to reduced server cost, power consumption and footprint. This is made possible by multi-core processors with scalable, efficient performance, coupled with significant memory and I/O bandwidth. This consolidation approach can be equally powerful in embedded systems, addressing issues around cost, software complexity, power, time and communication.

Often, performance efficiency is foremost on the minds of embedded system developers when running signal processing workloads. This is discussed in the next section, which presents performance data for key signal processing kernels running on 2nd generation Intel® Core™ i7 processors. Yet, for most embedded applications, raw performance isn't the only factor; it is also necessary to meet overall system cost goals, and the highly scalable family of embedded Intel architecture processors helps to do just that.

The embedded Intel processor roadmap gives developers a wide choice with respect to the number of cores, cache and system memory size, I/O and footprint. In addition, there are many other technologies available for enhancing system capabilities, like virtualization technology, remote management and various security features. Nevertheless, it is the enhanced vector single-instruction, multiple-data (SIMD) instructions that open the door to using Intel architecture processors for signal processing.

More specifically, the Intel® Advanced Vector Extensions (Intel® AVX) – available for the first time with 2nd generation Intel Core i7 processors– provide significantly improved floating point performance (see sidebar).

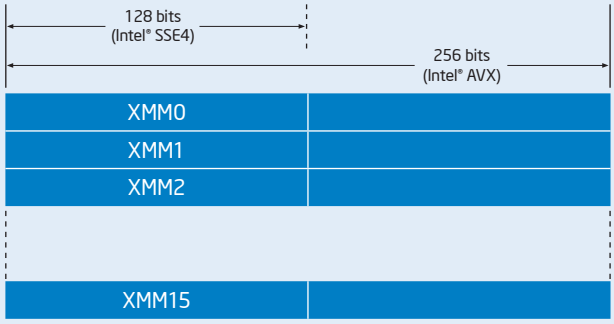
Engineers who code floating point algorithms for Intel architecture processors can leverage a mature software ecosystem that offers a very wide breadth and depth of development tools. Also available are Intel development tools and libraries that employ Intel AVX and Intel® Streaming SIMD Extensions 4 (Intel® SSE4) instructions. Equipment manufacturers can choose from many hardware vendors supplying commercial off-the-shelf (COTS) embedded boards and systems that support embedded lifecycles and benefit from the economics of the PC/server supply chain.

SIMD Instructions Enhanced By Intel® Advanced Vector Extensions (Intel® AVX)

Many signal processing applications are highly parallel, performing the same arithmetic operation on large number sets. Speeding up these workloads, single-instruction, multiple-data (SIMD) instructions were introduced in the mid 1990's, and they perform the same operation on multiple data elements simultaneously, as illustrated below.

SMID	2	3	5	11	20
+	9	11	2	1	5
=	11	14	7	12	25

The throughput of a SIMD instruction is a function of register size because larger registers translate into greater throughput. With the introduction of 2nd generation Intel® Core™ i7 processors, the size of the 16 registers available for floating point operations doubles, increasing from 128 bits to 256 bits. Additionally, new three and four operand instructions establish a destination argument that results in fewer register copies, better register usage, faster execution and smaller code size. These are just some of the recent architectural enhancements, called Intel® Advanced Vector Extensions (Intel® AVX).



The Process for Evaluating Signal Processing Performance

Requiring different levels of effort, there are a number of ways to evaluate the performance of Intel processors, such as using Intel® C++ and Intel® Fortran Compilers, calling optimized performance libraries, or coding optimizations in assembly language and employing compiler intrinsics. The approach presented in this paper strikes a balance between effort and optimization to quickly achieve good estimates for the performance of signal processing algorithms. This is done in two ways. The first is to utilize the Intel® Integrated Performance Primitives (Intel® IPP) library. It provides a quick way to assess the performance of hundreds of algorithms and math functions optimized for Intel architecture. Furthermore, the signal processing portion of the Intel IPP library includes over 250 functions, each often supporting multiple data types and in-place versus not in-place variants. There are also many image processing calls that are useful for DSP applications.

The Intel IPP library distribution includes a performance tool (with documentation) that allows developers to obtain performance metrics for all the functions in the library. For example, Figure 1 shows the results generated by a shell script written to automatically collect performance data on the number of clocks, execution time, and in some cases MFLOPs, for a function. For Figure 2, an .ini file was used in conjunction with the shell script, to report 2D FFT performance for specific input sizes.

```
#!/bin/bash
source /opt/intel/composerxe/bin/compilervars.sh intel64
DATE=`date +%Y-%m-%d`

OUTPUT_DIR=results_avx_${DATE}
OUTPUT_PATH=${PWD}/${OUTPUT_DIR}
PERF_TOOL_DIR=${IPPROOT}/tools/intel64/perfsys
FILE_EXT_1=_lin_avx_1
FILE_EXT_2=_lin_avx_2

mkdir ${OUTPUT_DIR}

#IPPS
${PERF_TOOL_DIR}/ps_ipps -r${OUTPUT_PATH}/ipps${FILE_
EXT_1}.csv -o${OUTPUT_PATH}/ipps${FILE_EXT_1}.txt -N1
-YHIGH -TAVX -B
${PERF_TOOL_DIR}/ps_ipps -r${OUTPUT_PATH}/ipps${FILE_
EXT_2}.csv -o${OUTPUT_PATH}/ipps${FILE_EXT_2}.txt -N1
-YHIGH -TAVX -B

#2DFFT
${PERF_TOOL_DIR}/ps_ippi -r${OUTPUT_PATH}/2dfft${FILE_
EXT_1}.csv -o${OUTPUT_PATH}/2dfft${FILE_EXT_1}.txt -N1
-YHIGH -TAVX -B -i${PWD}/2dfft.ini -fippiFFTFwd_CToC_32fc_C
${PERF_TOOL_DIR}/ps_ippi -r${OUTPUT_PATH}/2dfft${FILE_
EXT_2}.csv -o${OUTPUT_PATH}/2dfft${FILE_EXT_2}.txt -N1
-YHIGH -TAVX -B -i${PWD}/2dfft.ini -fippiFFTFwd_CToC_32fc_C
```

Figure 1. Sample shell script running IPP performance tool

```
[Perf System]

FFT_OrderXY=4x4; 5x5; 6x5; 6x6; 7x4; 7x5; 7x7; 8x3; 8x4; 8x6;
8x7; 8x8; 9x3; 9x5; 9x6; 9x8; 9x9;10x4; 10x5; 10x7; 10x8;
10x10; 11x3; 11x4; 11x6; 11x7; 11x11; 11x12; 11x13; 11x14;
11x15;12x3; 12x5; 12x6; 12x12; 13x4; 13x5; 13x13; 14x3; 14x4;
15x3; 15x4; 16x4; 17x3; 17x4;
```

Figure 2. Sample .ini file generating 2D FFT performance data

Another way to estimate signal processing performance is to focus the performance assessment on key kernels that are often used in signal processing workloads. By selecting a subset of the signal processing functions, developers can produce a manageable set of data that contains the most relevant functions and provides a reference with which to estimate the performance of other functions. For instance, this can be done by choosing forward and inverse FFTs of various sizes – both complex and real – along with FIR and IIR filters of varying complexities, and other useful functions such as discrete cosine and Hilbert transforms. Developers may certainly need data on functions other than the ones covered in this paper; in those cases, the Intel IPP performance tool can be used to gather the necessary data.

In summary, this process for evaluating the signal processing performance of Intel architecture utilizes Intel-collected performance data and gives developers a straightforward method to quickly estimate performance for their own workloads. Although the data provided is on 2nd generation Intel Core i7-2710QE processor, the methods described here are extensible to the full range of Intel processors. With a manageable effort, this process gives developers a quick readout of the signal processing performance of next generation Intel processors and provides an estimate of how much general-purpose computing headroom is available for other applications. The next section reviews the performance data collected and demonstrates the process using two examples.

It is important to note that although using Intel IPP to assess the signal processing performance of Intel processors provides a good starting point that balances effort and optimization, it need not be the endpoint. Going beyond Intel IPP, it may be possible to capture significant performance improvements for specific algorithms through the use of compiler optimizations, primitives and assembly language programming. The flexibility of Intel architecture and its supporting software infrastructure provides developers with all of these degrees of freedom that can ultimately identify the most appropriate tradeoff between performance and effort.

Signal Processing Performance Data

The following lists a sample of the signal processing performance data[1,2] collected by Intel on 2nd Intel Core i7-2710QE processors. The algorithms were run on a single execution thread, on Linux* (Fedora* 13 distribution), and repeated until the results of iterations were within 5 percent accuracy. Developers can create results for their own algorithms and functions of interest using an Intel® compiler and the Intel IPP package,

which includes sample code and the Intel IPP performance test tool. More details about the test configuration are provided in Appendix A.

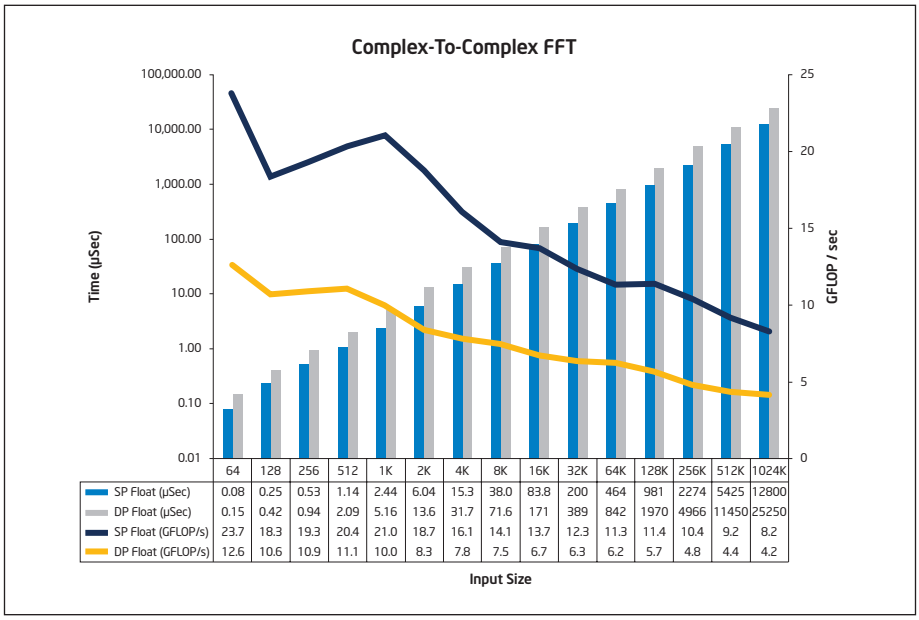
This single threaded execution environment provides an approximation of the performance of a single core and an indication of available computing headroom. Processor cores not used for signal processing can be targeted at other algorithms or applications running on the system. Intel® development tools and libraries also provide extensive support for development, validation and performance tuning of multi-threaded applications.

Overview of benchmark data

A. Forward and inverse Fast Fourier Transform (FFT).
   Format: Complex-to-Complex, Real-to-Complex Conjugate Symmetric
   Input range: 64B to 1024KB
   Data type: Single and double precision floating point
   Purpose: Decompose a discrete sequence into a set of frequencies.

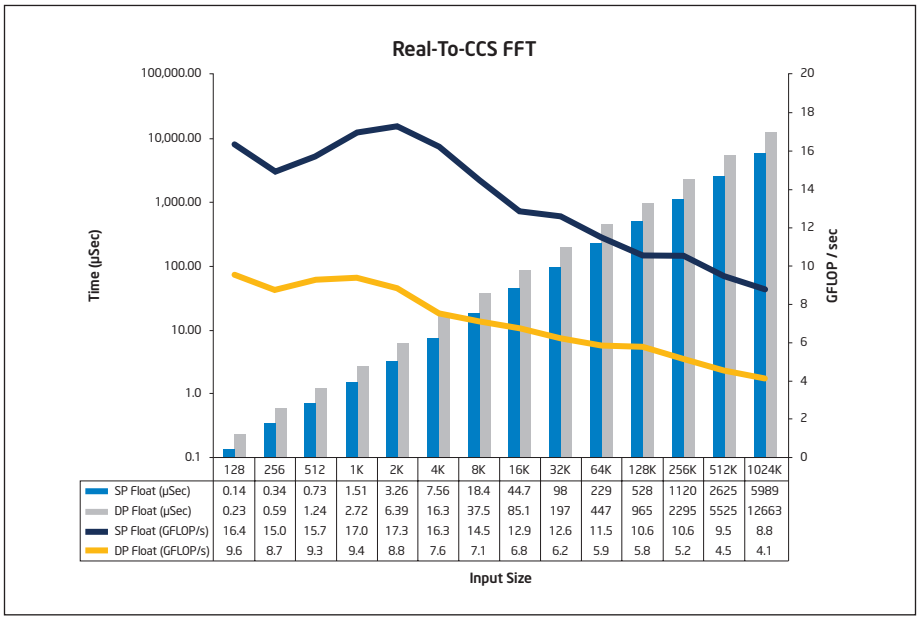
Complex-to-complex FFT performance of the IPP library is in the range of 16.1 – 23.7 single precision GFLOPs / sec for sizes between 64B and 4KB. Similarly, the performance of real-to-complex conjugate FFT is in the range of 15 – 17.3 single precision GFLOPs/sec for sizes between 64B and 4KB.

For larger sizes, FFT performance declines as the execution of the algorithm becomes less compute-bound and more memory-bound. Additionally, the FFT throughput of double precision floating point is approximately half that of single precision, and it scales with input size similarly to single precision.



	64	128	256	512	1K	2K	4K	8K	16K	32K	64K	128K	256K	512K	1024K
SP Float (µSec)	0.08	0.25	0.53	1.14	2.44	6.04	15.3	38.0	83.8	200	464	981	2274	5425	12800
DP Float (µSec)	0.15	0.42	0.94	2.09	5.16	13.6	31.7	71.6	171	389	842	1970	4966	11450	25250
SP Float (GFLOP/s)	23.7	18.3	19.3	20.4	21.0	18.7	16.1	14.1	13.7	12.3	11.3	11.4	10.4	9.2	8.2
DP Float (GFLOP/s)	12.6	10.6	10.9	11.1	10.0	8.3	7.8	7.5	6.7	6.3	6.2	5.7	4.8	4.4	4.2

Figure 3. Complex-to-Complex FFT and Inverse FFT



	128	256	512	1K	2K	4K	8K	16K	32K	64K	128K	256K	512K	1024K
SP Float (µSec)	0.14	0.34	0.73	1.51	3.26	7.56	18.4	44.7	98	229	528	1120	2625	5989
DP Float (µSec)	0.23	0.59	1.24	2.72	6.39	16.3	37.5	85.1	197	447	965	2295	5525	12663
SP Float (GFLOP/s)	16.4	15.0	15.7	17.0	17.3	16.3	14.5	12.9	12.6	11.5	10.6	10.6	9.5	8.8
DP Float (GFLOP/s)	9.6	8.7	9.3	9.4	8.8	7.6	7.1	6.8	6.2	5.9	5.8	5.2	4.5	4.1

Figure 4. Real-to-CCS (Complex Conjugate Symmetric) FFT

B. 2D Complex to Complex FFT Throughput (GFLOPS/s and absolute time)
Format: Complex-to-Complex, 2 dimensional array data
Input range: Various sizes ranging from 64B x 64B to 128KB x 16B
Data type: Single precision floating point
Purpose: Decompose a two dimensional array of discrete sequences into a set of frequencies.

Two dimensional complex-to-complex FFT performance of the Intel IPP library is in the range of 12.9 – 17.2 for sizes ranging from 64B x 64B to 4KB x 64B.

As with the one dimensional FFT case, 2D FFT performance declines for larger sizes as the FFT becomes more memory-bound. Table 1 contains sample data points of interest, and it is possible to generate 2D FFT throughput data for other sizes using the Intel IPP library.

Input Size	64 X 64	128 x 128	256 x 64	256 x 256	512 x 64	1K x 256	2K x 128	4K x 64	8K x 32	16K x 16	32K x 16	128K x 16
GFLOP/S	17.2	13.6	14.9	12.9	14.2	13.6	13.0	12.9	11.3	10.9	8.5	7.4
Time (µSec)	14.3	84.5	76.9	407	173	1744	1823	1834	2088	2163	5893	29838

Table 1. 2D Complex to Complex FFT Throughput (GFLOPS/s)

Speedup with Intel® Advanced Vector Extensions (Intel® AVX)

The improved performance from Intel® Advanced Vector Extensions (Intel® AVX) is illustrated in Figure 5, which shows the speed up compared to the prior generation Intel® Streaming SIMD Extensions 3 (Intel® SSE3) instructions. The comparison is for Complex-to-Complex FFT and Inverse FFT functions, which are averaged together and charted for both single and double precision floating point routines. For smaller input sizes, the speedup is over two times, and for very large inputs, the speedup is around 20 percent.

The floating point performance improvements for 2nd generation Intel® Core™ i7 processors are the result of architectural enhancements, which enable the processor to:

- Retire one floating point instruction per CPU clock cycle
- Dispatch up to 4 floating point instructions per CPU clock cycle

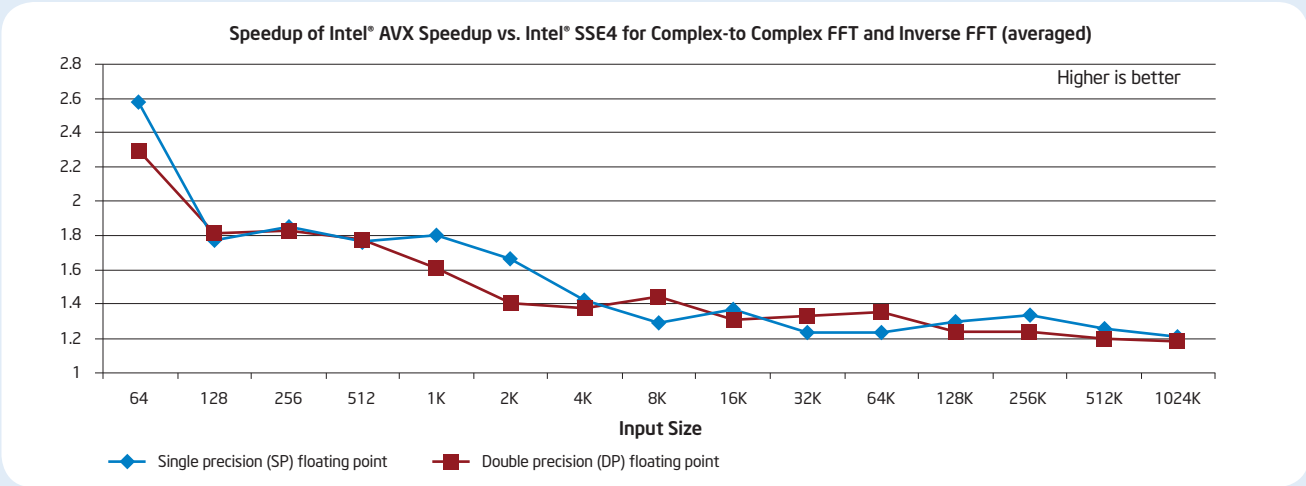


Figure 5. Intel® Advanced Vector Extensions (Intel® AVX) Speedup over Intel® Streaming SIMD Extensions 4 (Intel® SSE4)

C. Filter Execution Times

Format: Single precision floating point complex data, complex coefficients.

Finite Impulse Response Filter: 8, 32, and 128 taps

Infinite Impulse Response Filter: Orders ranging from 2 – 12 taps.

Inputs: Complex data ranging from 32B – 32KB in size.

Purpose: Suppress unwanted components from a discrete-time series.

	Input Size / Execution Time (Microseconds)					
	32 input	**128 input**	**512 input**	**2K**	**8K input**	**32K input**
8 Tap Fir	0.2	0.5	1.8	7.0	28.2	113.0
32 Tap FIR	0.6	2.0	4.2	15.3	59.4	244.0
128 Tap FIR	2.1	7.7	6.2	18.6	67.6	276.0
Order 2 IIR	0.2	0.7	2.4	9.7	38.5	156.0
Order 3 IIR	0.3	0.9	3.5	13.7	54.4	221.5
Order 4 IIR	0.4	1.1	3.8	14.9	59.4	239.0
Order 6 IIR	0.6	1.4	4.5	17.4	69.5	277.5
Order 7 IIR	0.7	1.6	5.2	20.3	81.3	330.0
Order 8 IIR	1.1	1.7	5.3	20.4	80.8	324.5
Order 10 IIR	1.1	2.1	6.3	24.2	97.0	387.0
Order 12 IIR	1.2	2.5	7.3	27.5	112.0	445.0
Order 11 IIR	1.2	2.3	7.0	26.6	107.0	427.0

Table 2. Filter Execution Time

D. Discrete Hilbert Transform

Format: Single precision floating point complex data.

Inputs: Complex data ranging from 128B – 32KB in size.

Purpose: Create analytic representation of a real-valued discrete signal.

	Input Size / Execution Time (Microseconds)				
	128	**512**	**2K**	**8K**	**32K**
INT16 to Complex Short FP	0.7	2.7	13.0	74.2	385.3
IN16 to Complex SP FP	0.6	2.4	11.5	64.1	353.6
SP Float to Complex SP Float	0.6	2.3	11.0	62.8	341.0

Table 3. Hilbert Transform Execution Times

E. Discrete Cosine Transform

Format:

Inputs: 128B – 32KB

Purpose: Express a discrete signal as a series of cosine frequencies that can be used for lossy signal compression.

	Input Size / Execution Time (Microseconds)				
	128	**512**	**2K**	**8K**	**32K**
SP Float Forward	0.8	3.7	20.6	111.8	563.8
SP Float Inverse	0.8	3.7	20.6	111.0	567.3
DP Float Forward	0.6	2.3	11.1	60.0	300.5
DP Float Inverse	0.6	2.3	11.0	58.9	301.8

Table 4. Discrete Cosine Transform Execution Times

Two Signal Processing Workload Examples

In this section, two generic signal processing workload examples are presented, and the performance of the 2nd Intel Core i7-2710QE processor is estimated in two ways, according to the methods described earlier. The first method is a simple manual approximation that adds the performance data of underlying functions obtained from the Intel IPP performance tool. Interpolation is used where measured data is not available. Though this is a rough approximation, it produces a quick performance estimate and a directional check of whether the system performance and headroom is adequate. The second method estimates the performance of the algorithms by coding them in C++ using Intel IPP and measuring their performance directly with hardware counters. In the following examples, the results from both methods are presented, which provides an indication of how well the manual method approximates actual measured results. The objective of this exercise is to produce a reasonable estimate of performance in order to determine where effort is best applied during optimization. Ultimately, detailed design work may be needed to complete product development.

Example 1: Fast Convolution using FFT

The following example performs a fast convolution of two discrete signals, x(n) and y(n) shown in Figure 6. The example is also a frequency domain FIR filter when one of the input sequences represents the transfer function of an FIR filter. A sample C-code snipet using Intel IPPs is provided in Figure 7.

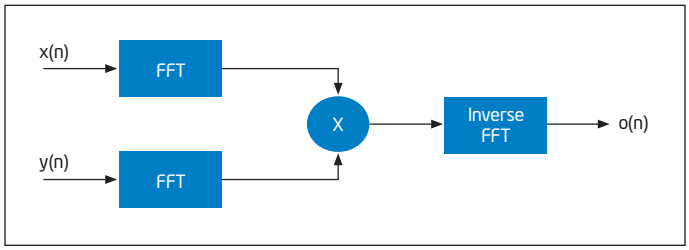


Figure 6. Fast Convolution using FFT Example

Inputs: x(n), y(n): 16KB input size
Output: o(n): 16KB output size
Operations: Single precision floating point in-place FFT, Complex Multiply, Inverse FFT

Sample C-code snipet for Example 1 using Intel® Integrated Performance Primitives (Intel® IPP):

```
/* allocate and initialize specification structures */
ippsFFTInitAlloc_C_32fc(&FFTspec1_p, order, IPP_FFT_DIV_
FWD_BY_N,  ippAlgHintFast);
ippsFFTGetBufSize_C_32fc(FFTspec1, &BufSize);
Buf1_p = (Ipp8u *) ippsMalloc_32sc(BufSize*sizeof(Ipp8u));
...
/* compute in-place FFTs of input sequences*/
ippsFFTFwd_CToC_32fc_I(x_p, FFTspec1_p, Buf1_p);
ippsFFTFwd_CToC_32fc_I(y_p, FFTspec1_p, Buf1_p);

/* perform complex multiplication and inverse FFT*/
ippsMul_32fc( x_p, y_p, o_p, veclength);
ippsFFTInv_CToC_32fc_I(o_p,FFTspec1_p, Buf1_p);
...
/* free specification structures */
ippsFFTFree_C_32fc( FFTSpec1_p);
ippsFree(Buf1_p);
```

Figure 7. Sample C-code Snipet for Example 1

Table 5 summarizes Fast Convolution execution times, calculated and measured, for various data sizes. The calculated times sum the execution times of individual functions, using times for Intel IPP signal processing functions obtained from the Intel IPP performance test tool running in batch mode. The measured times were generated by running the entire algorithm (Figure 6), which was coded in C++ and used Intel IPP, and by calculating the elapsed time based on the hardware clock count. The runtimes were averaged across 10,000 runs.

For comparison, the calculated times were within 16 percent of the measured results. However, the calculated times took a few hours of unattended run time (no human effort aside from installing the Intel IPP and running the aforementioned shell script) and less than an hour of calculating results in a spreadsheet. The measured results took an engineer familiar with the Intel IPP and C++ programming a couple

Size	FFT		iFFT		Complex Mul		Fast Convolution Calculated		Fast Convolution Measured		Delta
	µsec	clocks	µsec	clocks	µsec	clocks	µsec	clocks	µsec	clocks	
64	7.96E-02	1.67E+02	8.02E-02	1.68E+02	5.25E-02	1.10E+02	2.92E-01	6.13E+02	2.99E-01	6.28E+02	-2%
128	2.30E-01	4.83E+02	2.30E-01	4.83E+02	1.05E-01	2.21E+02	7.95E-01	1.67E+03	8.06E-01	1.69E+03	-1%
256	5.04E-01	1.06E+03	4.92E-01	1.03E+03	1.84E-01	3.87E+02	1.68E+00	3.54E+03	1.69E+00	3.55E+03	0%
512	1.07E+00	2.25E+03	1.04E+00	2.18E+03	3.17E-01	6.66E+02	3.50E+00	7.34E+03	3.80E+00	7.99E+03	-8%
1024	2.32E+00	4.87E+03	2.27E+00	4.77E+03	6.12E-01	1.29E+03	7.52E+00	1.58E+04	8.33E+00	1.75E+04	-10%
2048	5.91E+00	1.24E+04	5.93E+00	1.25E+04	1.18E+00	2.48E+03	1.89E+01	3.98E+04	2.11E+01	4.42E+04	-10%
4096	1.49E+01	3.13E+04	1.49E+01	3.13E+04	2.55E+00	5.34E+03	4.72E+01	9.92E+04	5.22E+01	1.10E+05	-10%
8192	3.65E+01	7.67E+04	3.60E+01	7.56E+04	5.46E+00	1.15E+04	1.14E+02	2.40E+05	1.28E+02	2.70E+05	-11%
16384	8.13E+01	1.71E+05	8.19E+01	1.72E+05	1.19E+01	2.50E+04	2.56E+02	5.38E+05	2.96E+02	6.21E+05	-13%
32768	2.00E+02	4.20E+05	2.02E+02	4.24E+05	2.57E+01	5.40E+04	6.28E+02	1.32E+06	7.47E+02	1.57E+06	-16%
65536	4.54E+02	9.53E+05	4.53E+02	9.51E+05	5.14E+01	1.08E+05	1.41E+03	2.97E+06	1.63E+03	3.43E+06	-14%

Table 5. Fast Convolution Execution Times

of days for coding, debugging and executing the runtime. From this example, it is clear the simple manual approximation method (i.e., calculated times) delivers a performance estimate at a fraction of the effort required for the coding approach, and it provides an early read on where to invest optimization effort. Still, coding the algorithm using Intel IPP took significantly less effort than the alternative, which is to manually optimize custom libraries for each generation of Intel processor.

One possible optimization is to parallelize the algorithm by going beyond vector instructions and executing the two FFTs on separate threads. Further parallelizing the execution, the threads can be dispatched to two processor cores and the results combined back to a single thread for the complex multiply and inverse FFT.

Floating Point Speeds Development

Voice quality is still a major concern for countless wireless, cable and internet service providers relying on Voice over IP (VoIP) to deliver telephony services. To ensure quality is on par with the Public Switched Telephone Network (PSTN), service providers require an economical and automatic means to continuously test calls in real time. A commonly used family of standards is PESQ[3] (Perceptual Evaluation of Speech Quality), which defines a MOS voice quality score that closely correlates to human listening experience, as shown in Figure 8. The algorithms perform voice encoding and measurements related to jitter, packet loss, time-clipping and channel errors. Many of the algorithms are computationally intensive and use floating point fast Fourier transforms.

For manufacturers of voice quality test equipment, passing industry conformance tests demands 32-bit float-like behavior throughout the application. This is exceptionally difficult to achieve with integer CPUs or FPGAs without suffering dire performance consequences. Likewise, fixed point math – add, multiply or divide – is not an option because of the loss of accuracy every time an instruction throws out a remainder. With a fixed point integer type, intermediate results of a multiply or add can grow beyond the fixed point type, causing overflow and truncation errors. This can make the result shrink below its fractional component and lead to incorrect results, like a PESQ score of 3.5 instead of 4.2.

Ixia*, a leading supplier of test and measurement equipment, decided to use multi-core Intel® processors with high performance floating point units because they could run the PESQ code as-is, hence minimal migration effort. The processors delivered accurate PESQ results and proved to have floating pointing performance on par with, and even superior to, many floating point DSPs. "Using Intel, we were able to get near-final performance numbers in just a few days, significantly lowering our project risk," says Bryan Rittmeyer, System Architect at Ixia.

Figure 8. MOS Diagram

Example 2: Discrete Envelope Detection / Amplitude Demodulation

The following example performs a fast convolution of two discrete signals, x(n) and y(n) shown in Figure 6. The example is also a frequency domain FIR filter when one of the input sequences represents the transfer function of an FIR filter. A sample C-code snipet using Intel IPPs is provided in Figure 7.

The second example is an envelope detector for a discrete time sequence, shown in Figure 9. The Hilbert transform produces the analytic representation of the signal, whose magnitude is obtained in order to generate the envelope of the signal, which is then downsampled. The downsampling is done in two stages since the carrier is operating at 200x the frequency of the message bandwidth. This keeps the FIRs to reasonable sizes. Finally, the DC component is removed from the discrete output sequence. Figure 10 contains a code snipet of the MATLAB* model for the envelope detector and Figure shows the results.

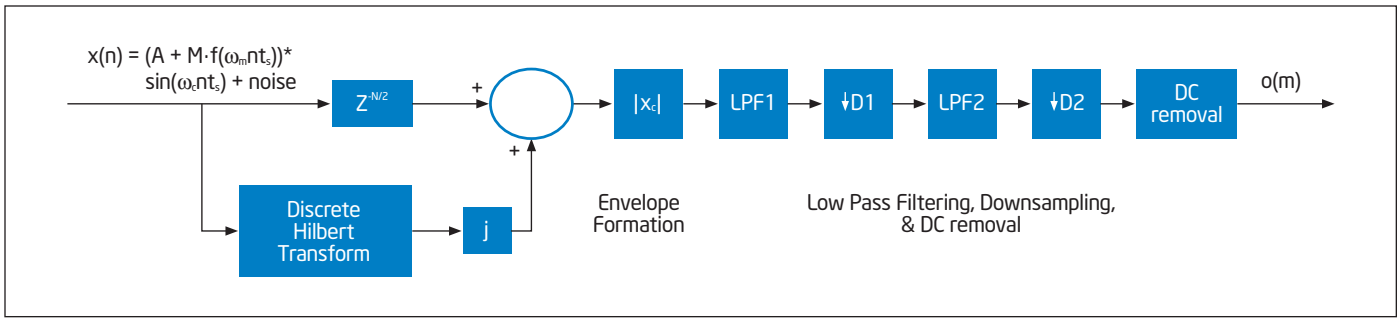


Figure 9. Discrete Envelope Detection / Amplitude Demodulation Example

Input: Amplitude modulated message. Message bandwidth = 5KHz.
Carrier frequency: 1000KHz. **Input sampling frequency:** 2200KHz.
Output sampling frequency: 11KHz
LPF1: 128 tap FIR, 44KHz cutoff frequency. **D1:** Downsampling by 25.
LPF2: 128 tap FIR, 5.5KHz cutoff frequency. **D2:** Downsampling by 8.

```
%Form analytic signal for envelope
inenv = abs(hilbert(in));

%Downsample, take out DC and LPF envelope
lpf1 = fir1(lpf1tap,cutoff1/(Fsi/2),'low',chebwin(lpf1tap+1));
out1 = fftfilt(lpf1,inenv);
out1 = downsample(out1,D1);
tout1 = downsample(tin,D1);

%Stage 2 Downsample & LPF envelope
lpf2 = fir1(lpf2tap,cutoff2/(Fs1/2),'low',chebwin(lpf2tap+1));
out = fftfilt(lpf2,out1);
out = downsample(out,D2);
out = out - mean(out);
out = out(17:length(out));
tout = downsample(tout1,D2);
tout = tout(17:length(tout));
```

Figure 10. Snipet of MATLAB* model

Figure 11 shows the MATLAB results for a simulated noisy AM signal containing messages centered at 1KHz, 3KHz, and 4.75KHz. The figure also shows the single sided spectrum magnitude of the input, after LPF1, after LPF2 and downsampling D2, and at the final output. Superimposed to the frequency spectrum is the frequency response of the LPFs (thin blue line).

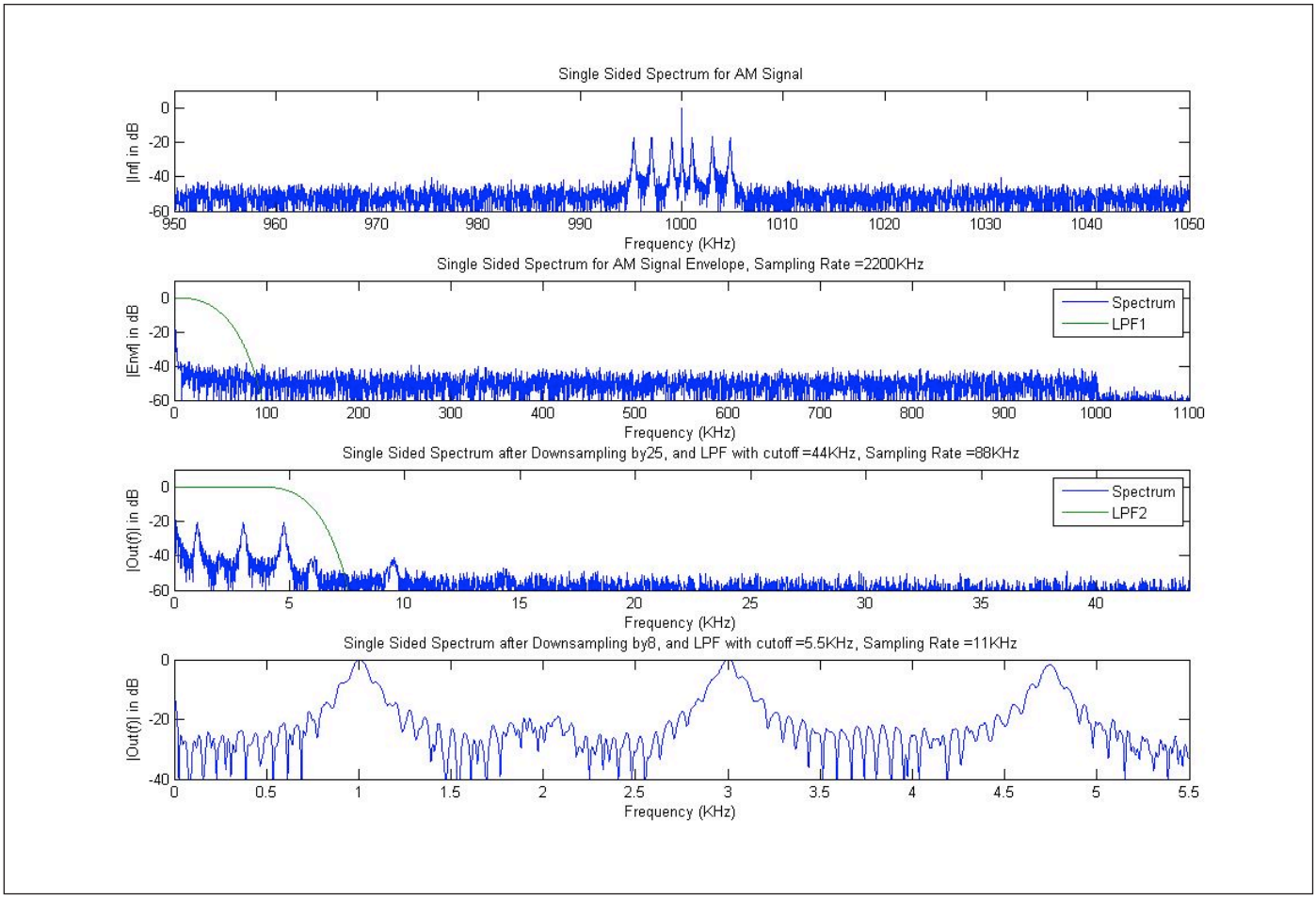


Figure 11. MATLAB* Results

Table 6 summarizes the execution times for the Amplitude Demodulation for various data sizes. Similar to Example 1, the calculated times summed the execution times of the individual functions, and the measured times were generated from hardware clock count measurements collected while the algorithm executed (see Figure 10 for the code snipet). The runtimes were averaged across 10,000 runs.

The calculated times are within 11 percent of the measured results. Here again, the calculated times took a few hours of unattended run time and less than an hour of spreadsheet calculation time. An engineer familiar with Intel IPP, C++ and the algorithm generated the measured results in three days, which included coding using Intel IPP functions, debugging and executing the runtime. This example is similar to Example 1, in that the manual approximation method offers a good compromise between accuracy and effort, and it provides a relatively quick indication of performance and focus areas for further optimization. As in the prior example, going to the next step of coding the algorithm using Intel IPP took an acceptable amount of effort, given the degree of optimization and compared to manually optimizing libraries to Intel processors.

Input Size	Hilbert Tran sf (32f - 32fc)		Magnitude		128 tap FIR #1		DownSampling by 8		DownSampling by 25	
	µsec	clocks	µsec	clocks	µsec	clocks	µsec	clocks	µsec	clocks
32	1.41E-01	2.96E+02	4.05E-02	8.51E+01	4.49E-01	9.43E+02	2.46E-02	5.17E+01	7.69E-02	1.61E+02
128	5.82E-01	1.22E+03	1.40E-01	2.94E+02	1.63E+00	3.42E+03	2.25E-02	4.73E+01	7.03E-02	1.48E+02
512	2.33E+00	4.89E+03	5.52E-01	1.16E+03	6.39E+00	1.34E+04	3.78E-02	7.94E+01	1.18E-01	2.48E+02
2048	1.09E+01	2.29E+04	2.20E+00	4.62E+03	2.63E+01	5.52E+04	9.67E-02	2.03E+02	3.02E-01	6.35E+02
8192	6.17E+01	1.30E+05	8.90E+00	1.87E+04	7.61E+01	1.60E+05	6.34E-01	1.33E+03	1.98E+00	4.16E+03
9000	7.09E+01	1.49E+05	9.91E+00	2.08E+04	8.10E+01	1.70E+05	7.41E-01	1.56E+03	2.31E+00	4.86E+03
18000	1.74E+02	3.64E+05	2.12E+01	4.45E+04	1.36E+02	2.85E+05	1.93E+00	4.05E+03	6.03E+00	1.27E+04
27000	2.76E+02	5.80E+05	3.25E+01	6.82E+04	1.90E+02	3.99E+05	3.12E+00	6.55E+03	9.74E+00	2.05E+04
32768	3.42E+02	7.18E+05	3.97E+01	8.34E+04	2.25E+02	4.73E+05	3.88E+00	8.15E+03	1.21E+01	2.55E+04
36000	3.76E+02	7.89E+05	4.36E+01	9.16E+04	2.47E+02	5.19E+05	4.26E+00	8.95E+03	1.33E+01	2.80E+04

Input Size	128 tap FIR #2		DownSampling by 8	
	µsec	clocks	µsec	clocks
327.68	5.69E+00	1.20E+04	3.56E-02	7.47E+01
360	6.09E+00	1.28E+04	3.68E-02	7.74E+01
720	1.57E+01	3.30E+04	6.54E-02	1.37E+02
1080	2.04E+01	4.28E+04	7.92E-02	1.66E+02
1310.72	2.34E+01	4.91E+04	8.81E-02	1.85E+02
1440	2.51E+01	5.26E+04	9.30E-02	1.95E+02

Input Size	Demod Calculated		Demod Measured		Delta
	µsec	clocks	µsec	clocks	
8192	1.54E+02	3.24E+05	1.59E+02	3.34E+05	-3%
9000	1.70E+02	3.58E+05	1.75E+02	3.67E+05	-3%
18000	3.52E+02	7.39E+05	3.68E+02	7.72E+05	-4%
27000	5.29E+02	1.11E+06	5.97E+02	1.25E+06	-11%
32768	6.42E+02	1.35E+06	7.18E+02	1.51E+06	-11%
36000	7.05E+02	1.48E+06	7.86E+02	1.65E+06	-10%

Measured Data Interpolation From Measured Data Calculated Data

Table 6. Amplitude Demodulation Execution Times

```
/* allocate and initialize specification structures */
ippsHilbertInitAlloc_32f32fc(&HilbertSpec_p, insmpl, ippAlgHintNone);
ippsFIRInitAlloc_32f(&LPF1FIRState_p, (Ipp32f *)lpf1taps, LPF1TAPCNT, NULL);
ippsFIRInitAlloc_32f(&LPF2FIRState_p, (Ipp32f *)lpf2taps, LPF2TAPCNT, NULL);
...
/* Form the analytic signal and its envelope */
ippsHilbert_32f32fc((Ipp32f *)in_p, inenv_p, HilbertSpec_p);
ippsMagnitude_32fc(inenv_p, inenvabs_p, insmpl);

/* First stage of LPF and downsampling */
ippsFIR_32f_I(inenvabs_p, insmpl, LPF1FIRState_p);
ippsSampleDown_32f((Ipp32f *)inenvabs_p, insmpl, outD1_p, &D1smpl, D1, &phm);

/* Second stage of LPF and downsampling */
ippsFIR_32f_I(outD1_p, D1smpl, LPF2FIRState_p);
ippsSampleDown_32f(outD1_p, D1smpl, outD2_p, &D2smpl, D2, &phm);

/* Remove DC */
ippsMean_32f(outD2_p, D2smpl, &dcval, ippAlgHintNone);
ippsSubC_32f_I((Ipp32f)dcval, outD2_p, D2smpl);
...
/* free states */
ippsHilbertFree_32f32fc(HilbertSpec_p);
ippsFIRFree_32f(LPF1FIRState_p);
ippsFIRFree_32f(LPF2FIRState_p);
```

Figure 12. Sample C-code Snipet for Example 2 using Intel® Integrated Performance Primitives (Intel® IPP)

The single-sided spectrum magnitude output of the Intel IPP implementation is shown in Figure 13 and indicates the resulting envelope and message recovery at the 11KHz sampling frequency.

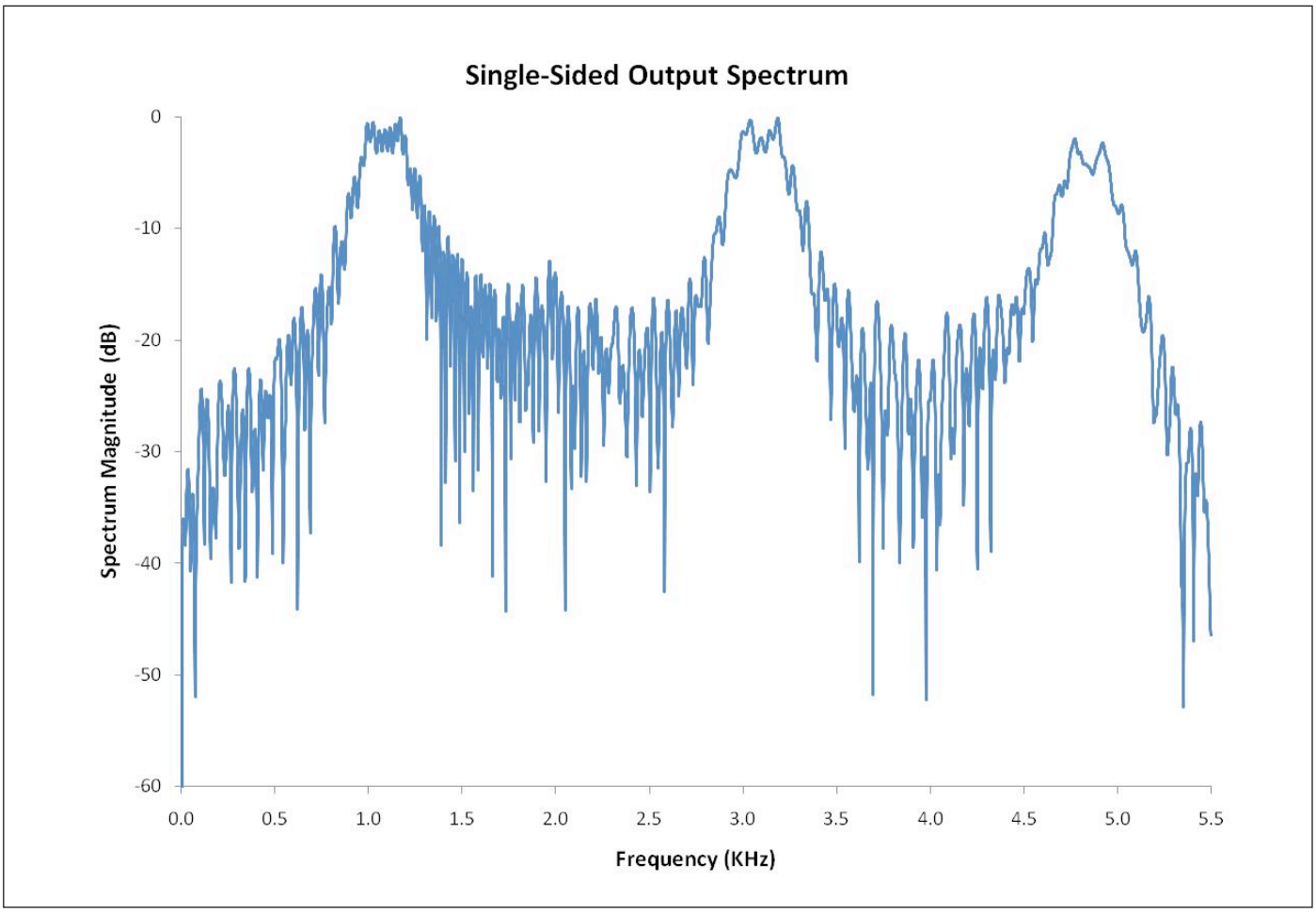


Figure 13. Output Spectrum Magnitude for Intel® Integrated Performance Primitives (Intel® IPP) implementation.

Table 7 summarizes the measured execution time for various lengths of input sample sequences. Of note is the column labeled Utilization Rate. This is the algorithm's execution time divided by the duration of the input sample, which provides a measure of core utilization over the time interval of the algorithm (i.e., before the next set of input samples need to be processed). It is an indication of the amount of headroom the core has available for additional signal processing functions, or perhaps, for other applications.

Input Size in Samples	AVX Speedup Over SSE	AVX Time (µsec)	Utilization Rate (%)
9000	1.32x	174.70	4.30%
18000	1.30x	367.64	4.50%
27000	1.27x	597.04	4.90%
36000	1.29x	785.95	4.80%
45000	1.26x	1,048.31	5.10%
54000	1.27x	1,223.71	5.00%
63000	1.25x	1,485.68	5.20%
72000	1.26x	1,659.40	5.10%
81000	1.25x	1,986.38	5.40%
90000	1.26x	2,155.75	5.30%

Table 7. Measured Execution Time for Envelope Detector

Development Tools Overview

Developers of signal processing applications have a wide choice of development tools from Intel and the broad Intel ecosystem. The benefits of using these comprehensive tool suites are many and impact every phase of the software development process.

Intel® C++ Compiler

The Intel C++ Compilers for Linux and Microsoft* Windows* operating systems are optimized to harness key properties of Intel architecture processors and deliver optimal performance. They take advantage of a complex set of heuristics to decide which assembly instructions can best optimize the performance in various area, including memory access, branch prediction, vectorization and floating point operations.

Intel® Math Kernel Library (Intel® MKL)

Intel® Math Kernel Library (Intel® MKL) is a library of highly optimized, extensively threaded math routines that rely heavily on floating point computations for maximum performance. Core math functions include BLAS, LAPACK, ScaLAPACK, Sparse Solvers, Fast Fourier Transforms, Vector Math and more.

Intel® Integrated Performance Primitives (Intel® IPP)

Intel IPP offers a rich set of library functions and codecs capable of speeding up the development of highly optimized routines for the handling of multimedia formats and data of any kind. They have been hand optimized at a low level to provide maximum performance and ease of use with Intel architecture processor-based platforms.

Intel® VTune™ Performance Analyzer

Designed to help developers find bottlenecks in their applications, the tool profiles how the application is using CPU time and computing platform resources throughout the code.

Intel® Application Debugger

A rich and user friendly Eclipse* RCP-based graphical user interface, combined with OS signal and thread awareness, enable developers to cross-debug more easily by finding coding issues that affect application runtime behavior.

Eclipse*-based Integrated Development Environment

Intel® software development products can be used with the Eclipse Integrated Development Environment (IDE).

Consider Intel® Architecture Processors for Signal Processing

Although today's Intel architecture processors are already being used for signal processing workloads, the release of 2nd generation Intel Core i7 processors with Intel AVX makes this approach much more compelling. Intel AVX delivers over twice the performance[1] for some floating point-based workloads compared to prior generation Intel SSE instructions. It is relatively straightforward for developers to evaluate the signal processing performance of next generation Intel architecture processors using the data available collected with Intel® tools and libraries.

Appendix A: Test Configuration

- Single thread execution
- Emerald Lake Platform (Fab A)
  - BIOS – American Megatrends 4.6.3.2 (Project Version – ASNBCPT1.86C. 0054.P00)
  - CPU: 2nd generation Intel® Core™ i7-2710QE processor (4 core, 2.1GHz, 6MB LLC, Intel® Hyper-Threading Technology off)
  - PCH: Mobile Intel® QM67 Chipset, B0 stepping.
  - 2 GB RAM (2x1GB Samsung DIMM DDR3 1333, dual rank, PN: M471B2874EH1-CH9)
  - Western Digital 160GB HDD (WD1600AAJS)
- Fedora* 13 Linux* 2.6.33.3-85.fc13.x86_64 operating system
- Intel® Composer XE 2011
  - Intel® C++ Compiler Pro, version 12.0.1, build 107.
  - Intel® Integrated Performance Primitives (Intel® IPP) version 7.0, build 205.23, September 2, 2010 (libippse9.so.7.0)
  - Intel IPP performance tool version 7.0 (part of the Intel IPP package)
- All individual Intel IPP measurements were taken using the Intel IPP performance test tool. Standard batch mode (-B) input was used. The automatic timing mode with default accuracy was used. The tests were run with high priority (Y=HIGH) and on one thread only (N=1). More information on the command line parameters can be obtained by running the performance applications with the –hh switch
- Frequency domain FIR was compiled in release mode (Release x64) with the Intel C++ Compiler. The cache is warmed before the test. Optimizations are enabled using the /O3 , -xHost, and –std=c99 compiler flags. FDFIR data averaged among in place, fast, and no divide by N options
- Other data averaged among in place and not in place, fast & accurate switches, divide by N, divide by sqrt(n), and no divide by N, as applicable to each algorithm
- Data is at fixed CPU clock frequency and may change with Intel® Turbo Boost Technology enabled.
- Software libraries, drivers, operating systems, and compilers used are not fully tuned for performance and additional performance gains may be possible.

Acronyms

ASIC	Application-specific integrated circuit
ASP	Application-specific processor
DSP	Digital signal processor
FIR	Finite impulse response
FFT	Fast Fourier transform
FPGA	Field-programmable gate array
IIR	Infinite impulse response
SIMD	Single-instruction, multiple data

[1] Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products.

[2] For more information go to http://www.intel.com/performance

[3] Source: PESQ website at http://www.pesq.org/

*Other names and brands may be claimed as the property of others.

Printed in USA 0111/S2D/BM/XX/PDF Please Recycle 324910-001US