Practical Intel® AVX Optimization on 2nd generation Intel® Core™ Processors

Introduction
The introduction of the 2nd generation Intel® Core™ processor family (Intel® microarchitecture code name Sandy Bridge) includes a tremendous opportunity to boost application performance. To many developers, those opportunities might be hidden under the hood, as they might not know how to utilize this huge performance treasure. The first effective way to boost performance is by utilizing multi-core CPUs. Another opportunity is Intel® Advanced Vector Extension (Intel® AVX), the new vectorization capabilities for floating point operations. Intel AVX is a new 256-bit instruction set extension to Intel® Streaming SIMD Extensions (Intel® SSE). It was released as part of the Intel microarchitecture code name Sandy Bridge and will be present in platforms ranging from notebooks to servers. The ability to work with 256-bit vectors enables operating on 8 float elements per iteration instead of a single element. This data parallelism can give great computation power to boost application programmers who know how to use it. The following article describes a simple methodology to utilize AVX inside the 2nd generation Intel Core processor family (Intel architecture codename Sandy Bridge). The paper shares insights and methods gained during a shared project by Waves (http://www.waves.com/) and Intel Software and Services Group on optimizing a Linear Phase FIR Filter using Intel AVX. It focuses on how Intel AVX technology can be exploited to achieve a significant performance boost. The paper includes concrete code examples to demonstrate the optimization.

Identify your hotspots
A hotspot is where the CPU spends significant amount of time in an application. The first step of optimization work must be to find the hotspots. Don’t skip this stage; otherwise you might optimize parts of the code which have almost no impact on the total application performance. Shortly before the introduction of the 2nd generation Intel® Core™ microprocessor, Intel released Intel® VTune™ Amplifier XE 2011 as part of the Intel® Parallel Studio XE 2011 tool suite. Intel Parallel Studio XE includes all the capabilities of Intel® Parallel Amplifier, plus a number of advanced capabilities for developers who need to dig deeper. Intel VTune Amplifier XE can pinpoint your hotspots, provide insights on CPU utilization, and show you where your parallelism is poor. There is no need for guessing about the location of performance bottlenecks in the application anymore.

One important way to optimize the hotspot is by multi-threading. If your desktop CPU has 4 cores (Quad Core) and supports Intel® Hyper-Threading Technology, it actually means you can run 8 simultaneous threads. If your application is still single-threaded, you have a theoretical opportunity to boost your performance by 8x. Learn more about parallel programming at http://software.intel.com/en-us/parallel/.

A new capability just introduced with the 2nd generation Intel® Core™2 Duo processor is Intel® Advanced Vector Extensions (Intel® AVX). This article describes how to use Intel AVX to improve performance.

SIMD (Single Instruction, Multiple Data) with Intel® AVX implementation
Before Intel microarchitecture code name Sandy Bridge, SIMD vectorization was provided by Intel® Streaming SIMD Extensions (Intel® SSE). Intel SSE is a technique for micro level data-level parallelism on x86 architecture. Intel SSE instructions use eight 128-bit registers, XMM0 through XMM7, where uniform type data can be packed. Intel SSE instructions operate on all data items in parallel. Intel SSE with 128-bit vectors enables operating on 4 float elements per iteration instead of a single element.

Figure 1: Going from Scalar to Intel® SSE can provide up to 4x faster performance
Intel® AVX brings new vectorization capabilities for floating point operations. The new ability to work with 256-bit vectors enables operating on 8 float elements per iteration instead of a single element.

It is easier to understand when looking at an example. Following is a representative inner loop from a Linear Phase FIR Filter that is often a hotspot. Let’s implement it with Intel SSE first and then Intel AVX. In this case, it was possible to obtain up to a 2.1x increase in speed when using Intel AVX over Intel SSE.

**Implementing A Linear Phase FIR Filter using Intel® AVX**

The following code example demonstrates the use of a specific linear phase audio FIR filter. A FIR (Finite Impulse Response) filter is a vector of filter taps values multiplied and accumulated (convolution) with history input signal samples, which may be used, for example, to achieve EQ effect on an audio signal.

A linear phase filter is characterized by a symmetric shape of the filter taps values. Since the filter length is odd, the symmetry is around a single center tap. In this example, we use a double precision floating point implementation. Recognizing that floating point values are subject to large errors when a large value is summed to a much smaller value, it is more accurate to sum the values by their expected magnitude, from the smallest to the largest. In the code for filter implementation, this is approximately true when we sum the values from the edges towards the center. Given the symmetric nature of the filter, it is better to read only the first half of the filter taps - where the second half taps are mirrored.

Filtering an audio signal with a FIR filter requires the convolution of the audio signal with the FIR filter. The convolution is implemented by iteration, one iteration per input sample, i.e., for each input sample, N filter taps are dot-multiplied by the last N input signal samples and summed to an accumulator. Infinite length input signal processors (such as in real-time processing) must use a memory FIFO buffer retaining old input samples. A FIFO buffer (First-In-First-Out) data structure is implemented using an array and mask. The mask is of 2^M-1 value (e.g., 0x00FFF) and is used to compute the FIFO access index using bit-wise AND operation rather than conditions, as the bit-wise AND is a faster implementation.

The basic algorithm works like this:

For each input signal sample:
- A. Insert sample to buffer, Advance the read/write offsets using a mask operation.
- B. Loop by half of the filter length
  1. Load 4 filter taps.
  2. Load 4 input samples from buffer.
  3. Multiply and accumulate (1)+(2).
  4. Mirror the 4 filter taps.
  5. Load 4 new input samples from buffer.
  6. Multiply and accumulate (4)+(5).
  7. Advance memory buffer access indexes.
- C. Multiply and accumulate center tap.
**Original C implementation**

The code example below shows only the implementation of the internal loop (B). It loads one filter tap per iteration.

```
for (int j=0 ; j<firHalfLength; j++) // firHalfLength is 1023
{
    dFirCoefs = pFIRBuf[j];
    acc1 += pDllBuf[lFirIndex]*dFirCoefs; //acc1 is accumulator for Index
    acc2 += pDllBuf[lFirIndexRev]*dFirCoefs; //acc2 is accumulator for IndexRev
    lFirIndex =(lFirIndex -1)&lMask; //dec backward index (modulo operation)
    lFirIndexRev = (lFirIndexRev+1)&lMask;
}
```

*Note: In this example, firHalfLength equals 1023 defining the iterations of the internal loop. In addition, the internal loop was nested with an external loop of 1,000,000 iterations in order to get meaningful numbers in the standalone measurements.*

The original C code above runs in ~0.63 sec on Intel® Core™ i7-2820QM CPU @2.3 GHz, RAM 4GB, Windows® 7 Ultimate with Service Pack 1.

**Intel® SSE 128-bit implementation**

Converting the code into Intel SSE loads two filter taps per iteration. The new code looks like this:

```
__m128d DllVal, FIRCoef, mulVal;

for (int i = 0; i < firHalfLength; i += 2) //Operate on 2 elements at a time
{
    FIRCoef = _mm_load_pd(pFIRBuf+i);

    //acc1
    DllVal = _mm_load_pd(pDllBuf+lFIRIndexRev);
    mulVal = _mm_mul_pd(FIRCoef, DllVal);
    acc1 = _mm_add_pd(acc1, mulVal);

    //acc2
    DllVal = _mm_load_pd(pDllBuf+lFIRIndex);
    DllVal = _mm_shuffle_pd(DllVal, DllVal, 0x1);
    mulVal = _mm_mul_pd(FIRCoef, DllVal);
```

*Note: Data Type is Double (64bit)*
acc2 = _mm_add_pd(acc2, mulVal);

lFIRIndex -= 2;
lFIRIndex = (lFIRIndex & lMask);
lFIRIndexRev += 2;
lFIRIndexRev = (lFIRIndexRev & lMask);
}

The SSE 128-bit code runs in ~0.40 sec on the exact same hardware as used to run the original code. This is already a speedup of 1.58x over the original code. However, Intel AVX can provide an additional speedup!

**Intel® AVX Implementation**

**Key Intel® AVX Features**

<table>
<thead>
<tr>
<th>Key Features</th>
<th>Benefits</th>
</tr>
</thead>
<tbody>
<tr>
<td>Wider Vectors</td>
<td>Up to 2x peak floating point operations per second (FLOPs) output with good power efficiency</td>
</tr>
<tr>
<td>- Increased from 128 to 256 bit</td>
<td></td>
</tr>
<tr>
<td>- Two 128-bit load ports</td>
<td></td>
</tr>
<tr>
<td>Enhanced Data Rearrangement</td>
<td>Organize, access and pull only necessary data more quickly and efficiently</td>
</tr>
<tr>
<td>- Use the new 256 bit primitives to broadcast, mask loads and permute data</td>
<td></td>
</tr>
<tr>
<td>Three and four Operands</td>
<td>Fewer register copies, better register use for both vector and scalar code</td>
</tr>
<tr>
<td>- Non Destructive Syntax for both 128 bit and 265 bit Intel AVX instructions</td>
<td></td>
</tr>
<tr>
<td>Flexible unaligned memory access support</td>
<td>More opportunities to fuse load and compute operations</td>
</tr>
<tr>
<td>Extensible new opcode (VEX)</td>
<td>Code size reduction</td>
</tr>
</tbody>
</table>

Intel® AVX is a general purpose architecture.

**Figure 3: Benefits of Intel® AVX.**

This implementation uses basic Intel AVX intrinsics code. The use of Intel AVX 256-bit vectors enables twice the number of elements processed per iteration in the loop - thus operating on 4 elements at a time. The main difference when comparing the Intel SSE intrinsics to the new Intel AVX intrinsics is clear when looking at the instruction names, e.g., _mm_load_pd ➔ _mm256_load_pd. Unlike Intel SSE intrinsics, the Intel AVX intrinsics can support up to 4 operands. In addition, there are different registers type __m128 ➔ __m256 with different width (128bit ➔ 256bit). The example uses the basic Intel AVX instructions such as _mm256_mul_pd and _mm256_add_pd.

**Intel® AVX - Arranging the Data**

In this example we also use “cross-lane” shuffle “mm256_permute2f128_pd” to arrange the data and eliminate the number of loads. More details on the “lane” concept will be discussed below. Since the index is decremented by (-4) in each iteration, we need to arrange each of the 4 elements being loaded in an opposite consecutive order, as described below:
**Intel® AVX - Implementation**

```c
__m256d DllVal, FIRCoef, mulVal;
__m128d tmph,tmpl,tmplsh,tmphsh;

for (int i = 0; i < firHalfLength; i += 4) //Operate on 4 elements at a time
{
    FIRCoef = _mm256_load_pd(pFIRBuf+i);

    //acc1
    DllVal = _mm256_load_pd(pDllBuf+lFIRIndexRev);
    mulVal = _mm256_mul_pd(FIRCoef, DllVal);
    acc1 = _mm256_add_pd(acc1, mulVal);

    //acc2
    DllVal = _mm256_load_pd(pDllBuf+lFIRIndex);
    DllVal = _mm256_permute2f128_pd (DllVal,DllVal ,0x1); // Cross lane shuffle
    DllVal = _mm256_permute_pd(DllVal, 0x5);
    mulVal = _mm256_mul_pd(FIRCoef, DllVal);
    acc2 = _mm256_add_pd(acc2, mulVal);

    lFIRIndex += 4;
    lFIRIndex = (lFIRIndex & lMask);
    lFIRIndexRev += 4;
    lFIRIndexRev = (lFIRIndexRev & lMask);
}
```

The Intel® AVX 256-bit code runs in ~0.19 sec on the exact same hardware as used to run the original code. Thus, the Intel AVX code shows a 2.1x improvement over the Intel SSE code.

**Intel® AVX Lanes**

Intel AVX implements 256-bit registers, divided into two lanes of 128 bits: a low lane and a high lane. Most of the 256-bit Intel AVX instructions are defined as in-lane: the destination elements in each lane are calculated using source elements only from the same lane. There are only a few cross-lane instructions such as the cross-lane” shuffle “mm256_permute2f128_pd” we used in the example.
Intel AVX Lanes

- Simple in-lane instructions
  - 2 lanes, 128 bit each
  
  \[
  \text{vAddPS dest, src1, src2}
  \]

- New in-lane PS and PD Permutest
  - Permute controlled via immediate
  
  \[
  \text{vPerm1PS dest, src, imm}
  \]

- New 128-bit permutes
  - Useful for lane-crossing operations
  
  \[
  \text{vPerm2F128 dest, src1, src2, imm}
  \]

**Figure 4:** Use of Intel® AVX lanes

**Strided Load**

The strided load technique is a programming method that uses Intel AVX instructions and is useful for algorithms that involve unsupported cross-lane shuffles. The method describes how to arrange data to avoid cross-lane shuffles. The main idea is to use 128-bit loads in a way that mimics the corresponding Intel SSE algorithm, and enables the 256-bit Intel AVX instructions to execute iterations i of the loop in the low lanes and the iteration and i+1 in the high lanes. This method eases the porting of Intel SSE algorithms, since 128 bits is still a natural data type for many algorithms, such as RGBA in single precision and Real and Imaginary in double precision complex. Intel AVX adds new in-lane data movement instructions such as \_mm_shuffle_pd, \_mm256_insertf128_pd and \_mm256_castpd128_pd256 to convert 256-bit to 128-bit width. See examples of Strided load in the presentation: [http://software.intel.com/file/29515](http://software.intel.com/file/29515) slide #59

**Cache alignment**

Aligning data to vector length is always recommended. When using Intel SSE and Intel SSE2 instructions, loaded data should be aligned to 16 bytes. Similarly, to achieve best results use Intel AVX instructions on 32-byte vectors that are 32-byte aligned. The use of Intel AVX instructions on unaligned 32-byte vectors means that every second load will be across a cache-line split, since the cache line is 64 bytes. This doubles the cache line split rate compared to Intel SSE code that uses 16-byte vectors. A high cache-line split rate in memory-intensive code is extremely likely to cause performance degradation. For that reason, it is highly recommended to align the data to 32 bytes for use with Intel AVX.
It is important to make every effort to align the data to 32 bytes in advance to avoid potential performance degradation. However, there are instances where the programmer is not able to force the data structure to be 32-byte alignment. For example, this can happen when working with third-party code. In cases where 32-byte alignment is not possible, change the load instruction to unaligned load “_mm256_loadu_pd”. This works, but it is clearly a compromise on performance. When we ran the sample Intel® AVX implementation of the linear phase audio FIR filter, we noticed a penalty degradation of more than 20% when working with misaligned data and using the loadu and storeu instructions.

**Intel® Compilers**

The Intel® C++ Compiler version 11.1 supports Intel® AVX both for Windows* and Linux*. In many cases (if there are not too many dependencies) it can generate Intel AVX instructions automatically. The Intel Composer XE (Intel® C++ Compiler version 12), part of Intel® Parallel Studio XE, has additional improvements for Intel AVX. On Windows*, use the command line switch /QxAVX. On Linux*, use –xavx. The switches /QxAVX (Windows*) and –xavx (Linux*) may be used to build applications that will take advantage of Intel AVX instructions on Intel® processor-based systems that support these, but will use only Intel SSE instructions on other systems.

The Intel® C++ Compiler supports Intel AVX-based intrinsics via the header file immintrin.h. To see which loops have been vectorized, use the switch /Qvec-report1 (windows) or –vec-report1 (Linux).

**Important notice for using Intel® AVX:**

1. In order to take advantage of Intel AVX on Windows* 7, Service Pack 1 must have been installed. Microsoft* Windows* 7 SP1 Beta is available from Microsoft at: [http://technet.microsoft.com/hi-in/evalcenter/ff183870.aspx](http://technet.microsoft.microsoft.com/hi-in/evalcenter/ff183870.aspx)
   
   Upon upgrade to Windows* 7 SP1, Intel AVX optimized software will automatically “opt-in” and take advantage of Intel AVX.

2. The Intel® Integrated Performance Primitives (Intel® IPP) library has been optimized for a variety of SIMD instruction sets. Automatic "dispatching" detects the SIMD instruction set that is available on the running processor and selects the optimal SIMD instructions for that processor. It is recommended to review Intel® IPP functions which are already optimized for Intel AVX. This can speed up the optimization and the development cycle. Below is a link for Intel IPP version 7 supporting Intel AVX that may be useful for optimization: [http://software.intel.com/en-us/articles/intel-ipp-functions-optimized-for-intel-avx-intel-advanced-vector-extensions/](http://software.intel.com/en-us/articles/intel-ipp-functions-optimized-for-intel-avx-intel-advanced-vector-extensions/)


Summary and Results

Intel® AVX provides an opportunity to boost application performance on 2nd generation Intel® Core™ processor-based platforms. First, the main hotspots of the application must be identified. Next, Intel AVX vectorization provides an optimization opportunity which is done per core. This optimization has a great impact when using all cores/threads with Intel AVX. Before using direct Intel AVX coding, it is recommended to review the great variety of relevant Intel® IPP version 7 functions which are already optimized for many of existing processors and can speed up the development cycle.

Wave’s Director of Research and Development Itai Neoran, said of AVX: “Similar double-precision FIR filters are widely used by Waves in real-time sound processing applications, in both music production studios and live-concert sound-processing. Since extensive sound processing is needed on each and every instrumental/vocal track, and since typical applications run hundreds such tracks in real-time, AVX parallelization allows sound engineers worldwide to use more simultaneous tracks, which until now could only be achieved with dedicated DSP hardware.”

In this article we showed how to use direct Intel AVX coding on a simple loop which was identified as a hotspot. Moving from original C implementation through Intel® SSE 128-bit to the Intel® microarchitecture code name Sandy Bridge 256-bit implementation, we were able to get impressive improvement. We’ve seen a 2.1x improvement from Intel SSE 128-bit code to Intel AVX 256-bit code for the hotspot that enabled the speedup of the entire FIR Filter application from Wave.

<table>
<thead>
<tr>
<th>Original C code</th>
<th>SSE 128 bit code</th>
<th>AVX 256 bit code</th>
</tr>
</thead>
<tbody>
<tr>
<td>~0.63 sec</td>
<td>~0.40 sec</td>
<td>~0.19 sec</td>
</tr>
</tbody>
</table>

Table 1: Intel® AVX code vs. Intel® SSE: 2.1x Improvement

About the Authors

Guy Ben-Haim:
Guy Ben-Haim is a senior application engineer in Intel corporation at the EMEA Software and Services Group (SSG). Guy works on optimizing applications to take advantage of the latest Intel software and hardware innovations. Guy provides technical training, consultation, and hands-on assistance to SW developers in areas of software optimization and parallel programming. Guy has been working in Intel for the last 10 years as a senior CPU designer where he participated in the development of the first dual core processor and other Centrino mobile processors. Prior to Intel, Guy worked for several startup up companies in the fields of communication, computers HW and semiconductors. Guy holds a B.Sc. from Tel Aviv University and M.B.A from Technion - Israel Institute of Technology.

Itai Neoran:
Itai Neoran is the director of research and development in Waves Audio Ltd. (Waves), Israel, a Grammy-awarded maker of software and hardware tools for digital audio processing in the music industry. During the last 17 years, Itai is leading a group of researchers and engineers who develop audio processing algorithms for professional and home users, and implement them in real-time, particularly on Intel CPUs. Itai also develops methods for acceleration of audio algorithms using parallel computation such as SSE and AVX, while keeping high bit accuracy and low input to output latency. Prior to Waves, Itai worked as a researcher in several companies employing signal processing and image processing algorithms. Itai holds an M.Sc in Electrical Engineering from the Technion Institute, Israel, and a B.Sc. in Computer Science from The Hebrew University of Jerusalem.

Ishay Tubi:
Ishay Tubi is a senior DSP engineer at Waves Audio Ltd., Israel. For the past 5 years, Ishay has been developing professional high-end audio signal processing algorithms. His command of FIR filters and Intel's parallel programing facilitates acceleration on various Waves Audio Plugins. Ishay holds a B.Sc. degree in computer science from Tel-Hai Academic College and has 15 years of experience in computer programming. Ishay is also the CTO and co-founder of MixBox Audio - an iOS Interactive music applications software company who's goal is to enable consumers without a musical background to mix and play music.