



Using Intel® Streaming SIMD Extensions and Intel® Integrated Performance Primitives to Accelerate Algorithms

Thu, Feb 9, 2012

By Sam Siewert

Algorithm Acceleration Using Single Instruction Multiple Data

Computing architecture can be described at the highest level using Flynn's architecture classification scheme as single instruction, single data (SISD); multiple instruction, single data (MISD); single instruction, multiple data (SIMD); or multiple instruction, multiple data (MIMD) systems. General-purpose computing most often employs simple SISD processing units, even if they are multi-core, super-scalar, or hyper-threaded; but the addition of vector instructions that can operate on multiple data words in parallel with a single instruction (SIMD) can provide powerful acceleration for data-intensive algorithms. Image processing; Digital Signal Processing (DSP); simulations employing grid data, graphics, and rendering; and many scientific and numerical algorithms can benefit from SIMD instructions to provide data processing parallelism. Intel introduced an instruction set extension with the Intel® Pentium® III processor called *Intel® Streaming SIMD Extensions* (Intel® SSE), which was a major re-design of an earlier SIMD instruction set called *MMX™* introduced with the Pentium® processor. Intel has updated the Intel® SSE instruction set extension based on user feedback as the Pentium® processor has progressed, and the latest Intel® SSE revision (version 4.2) can be found in the Intel® Core™i7 processor. Many systems employing the Intel® Core™2 processor family will have Supplemental Streaming SIMD Extension 3 (SSSE3), the generation of SIMD acceleration available prior to SSSE4.x.

This paper examines how to use Intel® SSE to accelerate an image processing application, how to identify which Intel® SSE instruction set your computing system has available, how to take advantage of it using compiler directives for optimal SIMD code generation, and how to employ Intel® Integrated Performance Primitives (Intel® IPP) to help fully exploit SIMD acceleration on the Intel® x86 architecture. The article examines optimization with Intel® SSE instructions using both the Intel® C++ Compiler and the GNU compiler collection (gcc) running on Fedora Core Linux*. Methods to verify speed-up, code generation, and how to employ Intel® IPP functions in applications are reviewed. Finally, the article looks at how SIMD acceleration architectures are evolving with Larrabee general-purpose graphics processing units (GPUs) and how SIMD technology-both integrated into general-purpose computing (GPC) cores and general-purpose graphics processing units (GPUs) like Larrabee-are enabling the implementation of many new algorithms with software that previously required exotic custom Application-Specific Integrated Circuit (ASIC) or Field Programmable Gate Array (FPGA) designs. The continued advancement of SIMD architecture with Intel® SSE 4.x and Larrabee provide exciting opportunities for system design and applications that are much more flexible, lower cost, and more software-driven.

Algorithms That Can Benefit from Intel® SSE

Algorithms that can benefit from Intel® SSE include those that employ logical or mathematical operations on data sets larger than a single 32-bit or 64-bit word. The Intel® SSE uses vector instructions, or SIMD architecture, to complete operations like bitwise XOR, integer or floating-point multiply-and-accumulate and scaling in a single clock cycle for multiple 32-bit or 64-bit words. Speed-up comes from the parallel operation of operations and the size of the vector (multi-word data) to which each mathematical or logical operator is applied. Examples of algorithms that significantly benefit from SIMD vector instructions include:

- **Image processing and graphics** - Both scale in terms of resolution (pixels per unit area) and the pixel encoding (bits per pixel to represent intensity and color) and both benefit from speed-up relative to processing frame rates.
- **Digital Signal Processing (DSP)** - Samples digitized from sensors and instrumentation have resolution like images as well as data acquisition rates. Often, a time series of digitized data that is one dimensional will still be transformed using algorithms like a DFT (Discrete Fourier Transform) that operate over a large number of time series samples.
- **Digest, hashing, and encoding** - Algorithms used for security, data corruption protection, and data loss protection such as simple parity, CRC (Cyclic Redundancy Check), MD5 (Message Digest Algorithm 5), SHA (Secure Hash Algorithm), Galois math, Reed-Solomon encoding, and Cypher-Block-Chaining all make use of logical and mathematical operators over blocks of data, often many Kilobytes in size.
- **Simulation of physical systems** - Most often simulations involve data transformation over time and can include grids of data that are transformed. For example, in physical thermodynamic, mechanical, fluid-dynamic or electrical-field models a grid of floating point values will be used to represent the physical fields as finite elements. These finite element grids will then be updated through mathematical transformations over time to simulate a physical process.

To better understand Intel® SSE instructions, you can look at a simple edge-enhancing image processing algorithm that uses multiply-and-accumulate operations on every pixel in a single image. The algorithm applies scaling to neighboring pixels and the current pixel in order to emphasize boundaries where pixel intensity changes occur. The next section examines how this works and how much Intel® SSE can speed up the process.

Image Processing

Images can be enhanced with sharper edges using an image processing algorithm that employs a Point Spread Function (PSF) on each pixel and a Red, Green, Blue (RGB) color band in an uncompressed Portable Pixel Map (PPM) image, as shown in Figures 1 and 2. Figure 1 shows the original image (a digital photo of a northern California sunset I took on vacation); Figure 2 shows the edge-enhanced version of the same PPM.



Figure 1. Original sunset digital photo in PPM uncompressed binary format

The original photo was a Joint Photographic Expert's Group (JPEG) image taken with a Sony* digital camera. I scaled the image down to 320×240 pixels and converted it into uncompressed PPM format with 24-bit RGB pixel encoding using [IrfanView \(http://www.irfanview.com/\)](http://www.irfanview.com/)* for Windows.



Figure 2. Edge-enhanced sunset digital photo in PPM uncompressed binary format

The edge-enhanced PPM in the same resolution in Figure 2 was created by running the PSF over all interior pixels (excluding top and bottom rows and left and right pixel columns) using the following "sharpen.c" C code block (see [downloads \(http://ecee.colorado.edu/~siewerts/sse/sharpen.c\)](http://ecee.colorado.edu/~siewerts/sse/sharpen.c) for the full source):

```
01 // Skip first and last row, no neighbors to convolve with
02 for(i=1; i>239; i++)
03 {
04
05     // Skip first and last column, no neighbors to convolve with
```

```

06     for(j=1; j>319; j++)
07     {
08         temp=0;
09         temp += (PSF[0] * (FLOAT)R[((i-1)*320)+j-1]);
10         temp += (PSF[1] * (FLOAT)R[((i-1)*320)+j]);
11         temp += (PSF[2] * (FLOAT)R[((i-1)*320)+j+1]);
12         temp += (PSF[3] * (FLOAT)R[(i)*320+j-1]);
13         temp += (PSF[4] * (FLOAT)R[(i)*320+j]);
14         temp += (PSF[5] * (FLOAT)R[(i)*320+j+1]);
15         temp += (PSF[6] * (FLOAT)R[(i+1)*320+j-1]);
16         temp += (PSF[7] * (FLOAT)R[(i+1)*320+j]);
17         temp += (PSF[8] * (FLOAT)R[(i+1)*320+j+1]);
18         if(temp<0.0) temp=0.0;
19         if(temp>255.0) temp=255.0;
20         convR[(i*320)+j]=(UINT8)temp;
21
22         temp=0;
23         temp += (PSF[0] * (FLOAT)G[((i-1)*320)+j-1]);
24         temp += (PSF[1] * (FLOAT)G[((i-1)*320)+j]);
25         temp += (PSF[2] * (FLOAT)G[((i-1)*320)+j+1]);
26         temp += (PSF[3] * (FLOAT)G[(i)*320+j-1]);
27         temp += (PSF[4] * (FLOAT)G[(i)*320+j]);
28         temp += (PSF[5] * (FLOAT)G[(i)*320+j+1]);
29         temp += (PSF[6] * (FLOAT)G[(i+1)*320+j-1]);
30         temp += (PSF[7] * (FLOAT)G[(i+1)*320+j]);
31         temp += (PSF[8] * (FLOAT)G[(i+1)*320+j+1]);
32         if(temp<0.0) temp=0.0;
33         if(temp>255.0) temp=255.0;
34         convG[(i*320)+j]=(UINT8)temp;
35
36         temp=0;
37         temp += (PSF[0] * (FLOAT)B[((i-1)*320)+j-1]);
38         temp += (PSF[1] * (FLOAT)B[((i-1)*320)+j]);
39         temp += (PSF[2] * (FLOAT)B[((i-1)*320)+j+1]);
40         temp += (PSF[3] * (FLOAT)B[(i)*320+j-1]);
41         temp += (PSF[4] * (FLOAT)B[(i)*320+j]);
42         temp += (PSF[5] * (FLOAT)B[(i)*320+j+1]);
43         temp += (PSF[6] * (FLOAT)B[(i+1)*320+j-1]);
44         temp += (PSF[7] * (FLOAT)B[(i+1)*320+j]);
45         temp += (PSF[8] * (FLOAT)B[(i+1)*320+j+1]);
46         if(temp<0.0) temp=0.0;
47         if(temp>255.0) temp=255.0;
48         convB[(i*320)+j]=(UINT8)temp;
49     }
50 }

```

The code simply scales each pixel R, G, and B value by the PSF, which has nine elements, with a central value that amplifies the value of the current pixel using PSF[4] and de-emphasizes the pixel according to the value of the nearest neighbor pixels in the same color band using PSF[0,1,2,3,5,6,7,8]. The resulting R, G, or B pixel value is then range limited to 0.0-255.0 and type converted back to an 8-bit unsigned integer. The theory behind how this serves to enhance edges as well as how PSF can be used for a wider variety of image enhancements can be found in Steven W. Smith's book, *The Scientist and Engineer's Guide to Digital Signal Processing* (<http://www.dspguide.com> (<http://www.dspguide.com>)). The key concept is that PSF convolution of images requires a large number of multiply and accumulate operations in floating point driven by the resolution of the image and pixel encoding (76,800 3-byte pixels in this example).

Edge enhancement is often used as a first image processing step in computer vision prior to segmenting scenes, finding target-of-interest centers, and using this information for higher-level navigation or remote sensing operations. Most often, this data will be acquired at high frame rates—for example, 29.97 frames per second for National Television Standards Committee (NTSC) video. So, the processing of each frame must be done in less than 33.37 milliseconds overall—a deadline that becomes critical in this example.

Invoking SSE to Accelerate the Image-Processing Example

The PSF code, `sharpen.c`, is first compiled using `gcc` as follows:

```
1 [root@localhost code]# make -f Makefile.gcc
2 cc -O0 -c sharpen.c
3 cc -O0 -o sharpen sharpen.o
```

No optimization was used, and the SSSE3 instructions available on my Intel® Core™ 2 Duo laptop were not employed in the code generation. This resulted in a run time on Fedora Core 9 Linux* 2.6.25 as follows:

```
1 [root@localhost code]# ./sharpen sunset.ppm
2 Cycle Count=1729087321
3 Based on usleep accuracy, CPU clk rate = 1729087321 clks/sec, 1729.1 MHz
4 Convolution time in cycles=24274250, rate=1729, about 14 millisecs
5 [root@localhost code]#
```

The time of 14 milliseconds on my dual-core 1.7-GHz Intel® Core™ 2 Duo is dangerously close to the deadline of 33 milliseconds per frame and leaves little time for any additional image processing to segment the scene or find targets of interest. Knowing that the Intel® Core™ 2 Duo processor has the SSSE3 instruction set extension and quickly referencing x86 `gcc` compiler directives, the code was re-compiled with level-3 (`-O3`) optimization and directed to use SSSE3 architecture-specific code generation (`-mssse3`), which is critical on 32 bit architectures for optimal double precision word alignment for cache and memory interfaces:

```
1 [root@localhost code]# make -f Makefile.gcc
2 cc -O3 -mssse3 -malign-double -c sharpen.c
3 cc -O3 -mssse3 -malign-double -o sharpen sharpen.o
4 [root@localhost code]# ./sharpen sunset.ppmCycle Count=1729178581
5 Based on usleep accuracy, CPU clk rate = 1729178581 clks/sec, 1729.2 MHz
6 Convolution time in cycles=9850750, rate=1729, about 5 millisecs
7 [root@localhost code]#
```

You can see that the processing time was significantly reduced to almost a third of the original time. Being curious how much of the acceleration came from SSSE3 instructions compared to general compiler code optimization, I re-compiled again without the SSSE3 directive:


```

1 [root@localhost code]# make -f Makefile.gcc
2 cc -O3 -c sharpen.c
3 cc -O3 -o sharpen sharpen.o
4 [root@localhost code]# ./sharpen sunset.ppm
5 Cycle Count=1729176397
6 Based on usleep accuracy, CPU clk rate = 1729176397 clks/sec, 1729.2 MHz
7 Convolution time in cycles=12817857, rate=1729, about 7 millisecs
8 [root@localhost code]#

```

So, you can see that the SSSE3 instructions used for multiply and accumulate specifically provide a 28.6% speed-up compared to the un-accelerated time of 7 milliseconds.

Without tools like Intel® IPP and the Intel® C++ Compiler, I was forced to write some assembly code to sample the Intel® Pentium® time stamp counter (TSC) in order to accurately time execution of this PSF code block. The assembly for this can be found in the sharpen.c full source. Clearly, SSSE3 accelerates image processing, so I became curious about how I could use the Intel tools on Linux to better understand my system and to simplify more significant image processing code development. Luckily, Intel provides its tools for Linux* for academic purposes free of charge from its [Non-Commercial Software Download site \(/en-us/articles/non-commercial-software-download\)](https://software.intel.com/en-us/articles/non-commercial-software-download).

Using the Intel® C/C++ Compiler and Intel® IPP Tools

Downloading the tools from the Non-Commercial Software Download site, I found that I could easily install the Intel® C/C++ Compiler, the Intel® IPP library, and the Intel® Math Kernel Library (Intel® MKL) all in one download for IA32 or IA64 architectures. Installation on my Fedora Core 9 Linux* 2.6.25 laptop was as simple as extracting the archive and running the install.sh script. The Intel® C++ Compiler Professional Edition installation, in fact, installed the compiler, Intel® IPP, and Intel® MKL. To verify the Intel® IPP installation, I ran the built-in test programs and compiled the examples.

Compiling on Linux Using the Intel® C++ Compiler

Using Intel® IPP, I modified the example code, sharpen.c, to use Intel® IPP core library functions to positively identify the host processor-Intel® Core™2 Duo, in my case-and to provide the TSC timing and core cycle rate instead of the custom assembly I had to write before. So, Intel® IPP not only provides core functions for DSP, image processing, vector/matrix math, and cryptography functions but also provides convenient host processor utilities to work with the wide variety of Intel® SSE instruction set extensions found on the x86 architecture.

Taking this updated code and recompiling it using the Intel® C++ Compiler, I found similar results to the gcc when I first built the code without the directive to employ SSSE3 in code generation:

```

1 [root@localhost code]# make -f Makefile.icc
2 icc -O3 -c sharpen.c
3 icc -O3 -o sharpen sharpen.o
4 [root@localhost code]# ./sharpen sunset.ppm
5 Cycle Count=1729177684
6 Based on usleep accuracy, CPU clk rate = 1729177684 clks/sec, 1729.2 MHz
7 Convolution time in cycles=12858508, rate=1729, about 7 millisecs
8 [root@localhost code]#

```

Re-compiling again with the SSSE3-specific directive with the Intel® C++ Compiler yielded SIMD accelerated code:

```

1 [root@localhost code]# make -f Makefile.icc
2 icc -O3 -mssse3 -align -xssse3 -axssse3 -c sharpen.c
3 icc -O3 -mssse3 -align -xssse3 -axssse3 -o sharpen sharpen.o
4 [root@localhost code]# ./sharpen sunset.ppm
5 Cycle Count=1729154843
6 Based on usleep accuracy, CPU clk rate = 1729154843 clks/sec, 1729.2 MHz
7 Convolution time in cycles=10226502, rate=1729, about 5 millisecs
8 [root@localhost code]#

```

From this comparison of gcc and the Intel® C++ Compiler, you can see that it is the SSSE3 instruction set that provides the speed-up, and both compilers are able to take good advantage of the extensions. While Intel® MKL and Intel® IPP can be used with either GCC or the Intel C++ Compiler, I used the Intel® C++ Compiler to link in Intel® IPP functions. Overall, the Intel® C Compiler, Intel® IPP, and Intel® MKL were well integrated, easy to use, and yielded optimized code, but it's nice to know that open source tools also work well with Intel® SSE.

Linking in the Intel® IPP Libraries

Linking in Intel® IPP using the Intel® C Compiler is simple. On my Intel® Core™2 Duo laptop, both GCC and the Intel C++ Compiler generated equally fast and efficient code using the SSSE3 instructions. To test this yourself, start by using the core library with any code you have to identify your hardware capabilities and to start making use of the TSC timing utilities. The example code used in this article can be downloaded from my website [<http://ecee.colorado.edu/~siewerts/sse/> (<http://ecee.colorado.edu/~siewerts/sse/>)]. A few simple make file directives and C flags added to a standard make file are all that you need to link in the static core library and employ Intel® SSE:

```

01 INCLUDE_DIRS =
02 LIB_DIRS = /opt/intel/Compiler/11.0/083/ipp/ia32/lib
03
04 CFLAGS= -O3 -mssse3 -align -xssse3 -axssse3 $(INCLUDE_DIRS) $(CDEFS)
05 LIBS= -lippcore
06
07 PRODUCT=sharpen
08
09 HFILES=
10 CFILES= sharpen.c
11
12 SRCS= ${HFILES} ${CFILES}
13 OBJS= ${CFILES:.c=.o}
14 CC= icc
15
16 all:    ${PRODUCT}
17
18 clean:
19     -rm -f *.o *.NEW *~
20     -rm -f ${PRODUCT} ${DERIVED} ${GARBAGE}
21
22 ${PRODUCT}: ${OBJS}
23     $(CC) $(LDFLAGS) $(CFLAGS) -o $@ $(OBJS) $(LIBS)
24
25 depend:

```

```
26  
27 .c.o:  
28 $(CC) $(CFLAGS) -c $>
```

Intel® Core™ i7 SSE4.2 and Larrabee

The Intel® instruction set extensions for DSP, image processing, and vector math were introduced with the MMX™ extensions to the original Intel® Pentium® ISA. Since this time, Intel® SSE was introduced and continued innovation in SIMD hardware acceleration instructions with Intel® SSE2, SSE3, SSSE3, and finally SSE4.2 with the Intel® Core™ i7 architecture. In all cases, the SIMD processing was provided as an extension to the x86 GPC processor architecture, making it convenient to accelerate existing code bases with simple re-compilation and to improve or design code by linking in Intel® IPP functions for commonly used DSP, image processing, vector math, and encryption functions in applications.

Intel provided additional graphics performance and flexibility with Larrabee, a many-core offload architecture for visual computing that can be integrated with x86 GPC architectures or stand on its own. The Larrabee architecture was first unveiled by Intel in detail at SIGGRAPH 2008 (see the paper, "[Larrabee: A Many-Core x86 Architecture for Visual Computing](/sites/default/files/m/d/4/1/d/8/larrabee_manycore.pdf)" /sites/default/files/m/d/4/1/d/8/larrabee_manycore.pdf). The details of Larrabee go beyond the scope of this paper, but the SIGGRAPH paper provides an excellent overview of how Intel intends to provide GP-GPU SIMD in addition to Intel® SSE.

Summary

Both built-in SIMD instruction set extensions to GPC and GPGPU offload SIMD are critical to next-generation applications. The built-in SIMD instructions provide acceleration to all existing GPC applications with simple re-compilation. Likewise, with a bit more effort, existing applications can be re-worked to link in Intel® IPP functions to go beyond simple instruction set-level acceleration. New applications can start out using Intel® IPP and leverage the rich code base of DSP, image processing, and vector math functions from the start. For many applications, the built-in acceleration is optimal, because it does not require I/O to employ the SIMD acceleration but rather in-memory processing and is simple to integrate. Looking forward, in-data-path processing employing Larrabee as an extension to general-purpose x86 CPU's greatly enhances the x86 architecture and application performance.

About the Author

Dr. Sam Siewert is the chief technology officer (CTO) of Atrato, Inc. and has worked as a systems and software architect in the aerospace, telecommunications, digital cable, and storage industries. He also teaches as an Adjunct Professor at the University of Colorado at Boulder in the Embedded Systems Certification Program, which he co-founded in 2000. His research interests include high-performance computing and storage systems, digital media, and embedded real-time systems.

Categories: [Game Development \(/en-us/search/site/language/en?query=Game%20Development\)](/en-us/search/site/language/en?query=Game%20Development), [Graphics \(/en-us/search/site/language/en?query=Graphics\)](/en-us/search/site/language/en?query=Graphics), [Intel® Integrated Performance Primitives \(/en-us/search/site/language/en?query=Intel%C2%AE%20Integrated%20Performance%20Primitives\)](/en-us/search/site/language/en?query=Intel%C2%AE%20Integrated%20Performance%20Primitives), [Intel® Streaming SIMD Extensions \(/en-us/search/site/language/en?query=Intel%C2%AE%20Streaming%20SIMD%20Extensions\)](/en-us/search/site/language/en?query=Intel%C2%AE%20Streaming%20SIMD%20Extensions), [Desktop \(/en-us/search/site/language/en?query=Desktop\)](/en-us/search/site/language/en?query=Desktop), [Laptop \(/en-us/search/site/language/en?query=Laptop\)](/en-us/search/site/language/en?query=Laptop)

[query=Laptop\)](#)


Tags: [visual computing \(/en-us/search/site/field_tags/visual-computing-17238/language/en?query\)](#), [image processing \(/en-us/search/site/field_tags/image-processing-17985/language/en?query\)](#), [Intel SSE \(/en-us/search/site/field_tags/intel-sse-21971/language/en?query\)](#), [vcsource type techarticle \(/en-us/search/site/field_tags/vcsource-type-techarticle-17309/language/en?query\)](#), [vcsource domain media \(/en-us/search/site/field_tags/vcsource-domain-media-17243/language/en?query\)](#), [vcsource os windows \(/en-us/search/site/field_tags/vcsource-os-windows-17246/language/en?query\)](#), [vcsource platform desktoplaptop \(/en-us/search/site/field_tags/vcsource-platform-desktoplaptop-17245/language/en?query\)](#), [vcsource product icc \(/en-us/search/site/field_tags/vcsource-product-icc-17859/language/en?query\)](#), [vcsource os linux \(/en-us/search/site/field_tags/vcsource-os-linux-17773/language/en?query\)](#), [vcsource index \(/en-us/search/site/field_tags/vcsource-index-20510/language/en?query\)](#)

For more complete information about compiler optimizations, see our [Optimization Notice](#).

Comments (5)

[^Top](#)

said on Wed, 11/04/2009 - 23:40

Great intro to SIMD, thanks! But I think there's an error in the pixel convolution algorithm in the first code block. You intend to saturate the accumulated value between 0 and 255, but with the code shown, it will have maximum 0.0 with unbounded minimum: 


```
if(temp>0.0) temp=0.0;  
if(temp>255.0) temp=255.0;
```

It should be changed to:

```
if(temp<0.0) temp=0.0;  
if(temp>255.0) temp=255.0;
```

This error occurs 3 times, once for each color channel. Just thought you might like to know...

said on Sat, 11/28/2009 - 20:49

reed-solomon? Do you have any sample code for SSE-accelerated Galois field multiply? esp. GF(2¹⁶) as used in par2. 

GF mult is log/antilog table-lookup driven, and SSE can't accelerate that much/at all.

Please email me if you reply, if the web site doesn't auto-notify. peter@cordes.ca

said on Mon, 12/21/2009 - 01:45

Andrew,



Thanks for pointing this error out!

It looks like the "<" got flipped to ">" somehow in the inline code text since the download code and the code I compiled and tested is correct. The image would have been severely messed up rather than sharpened with the incorrect comparison that you point out.

<http://avatar.colorado.edu/sse/sharpen.c> -- this code is correct, so please download from here.

The lines 18, 32, and 46 should match the code from the download link above.

Perhaps the editors can fix this in the paper?

Sam

said on Mon, 12/21/2009 - 01:48

Peter - Agreed on the lookup aspect of GF math.



The main acceleration I would expect is for the XOR operations, of which there are many.

Sam

[Lexi S \(Intel\)](#) said on Mon, 12/21/2009 - 10:10

Sam + Andrew: Fixed.



Add a Comment

[^Top](#)

(For technical discussions visit our [developer forums](#). For site or software product issues [contact support](#).)

Please [sign in](#) to add a comment. Not a member?

[Join today >](#)

[Terms of Use](#)

[*Trademarks](#)

[Privacy](#)

[Cookies](#)

Look for us on:



[English >](#)