Using SSE and SSE2: Misconceptions and Reality

Alex Klimovitski
Software Application Engineer
Tools and Technologies, Europe
Intel GmbH

Table of Contents

(Click on page number to jump to sections)

Using SSE and SSE2: Misconceptions and Reality

Alex Klimovitski
Software Application Engineer
Tools and Technologies, Europe
Intel GmbH

Overview

Single Instruction Multiple Data (SIMD) has industry-wide acceptance as a technology that increase performance on a wide variety of applications. Intel® Streaming SIMD Extensions (SSE) are sets of instructions that Intel introduced with the Intel® Pentium® III processor. This technology was the first major expansion of the Intel® instruction set since the introduction of Intel® MMX™ technology in 1995. With the launch of the Intel® Pentium® 4 platform, SSE was supplemented by SSE2. SSE/SSE2 can dramatically enhance software performance in a wide range of applications, from secure communication to speech recognition and synthesis, and from 3D visualization and video processing to realistic physics modeling.

Programming with SSE/SSE2 is quick, easy, and supported by mature tools. However, when trying to use SIMD in a real-life application, a developer is frequently confronted with a number of tough questions. How do I best exploit the parallelism promised by SIMD? How should I organize my data? And what if the optimal data structure isn't compatible with my algorithm? This article seeks to rectify some misconceptions about using SSE/SSE2 in the "real life" applications and to direct you to techniques and tricks that help achieve the best performance in your code.

SSE/SSE2 and Data Types

Misconception: Intel® SSE/SSE2 technology is applicable only to certain data types (just single precision floating-point, or just 16-bit integer…)
Reality: Intel SSE/SSE2 technology applies to elements of *all* standard data types, both floating-point (FP) and integer, that fit into 16-byte-wide SSE/SSE2 registers (Figure 1).

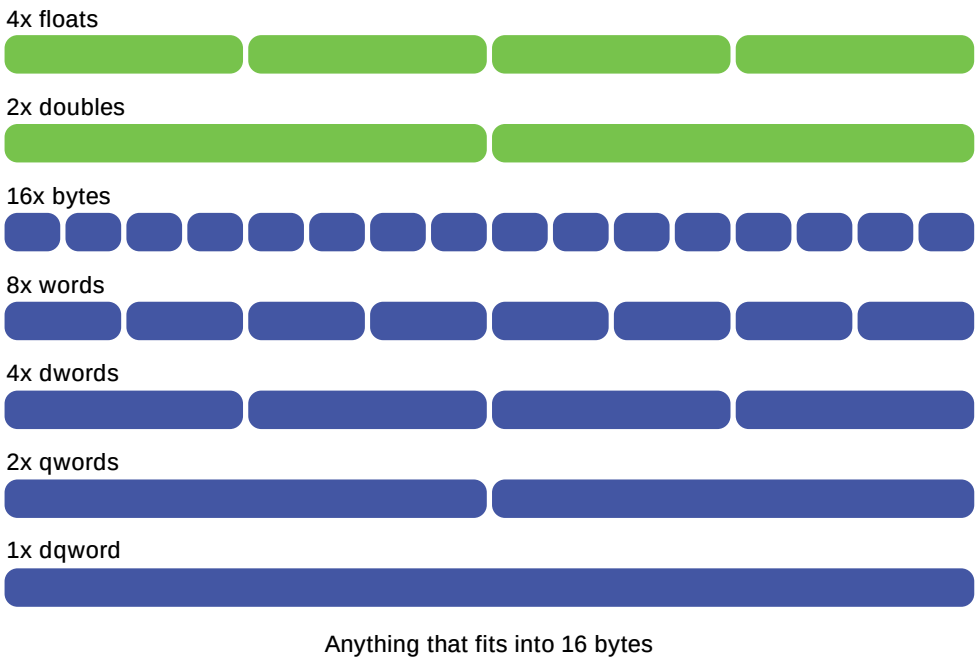


Figure 1. SSE/SSE2 data types

The supported data types include double- and single-precision FP data and all integer types with a length from 8 up to 128 bits. SSE/SSE2 offer a rich set of operations for each of these data types. This means, for example, that with just one instruction you can simultaneously add or multiply two operands of four floats, or of two doubles, or of two 64-bit integers, or of sixteen 8-bit integers, etc. This is how SSE/SSE2 dramatically increase the computational power available to your applications.

SSE/SSE2 and Coding Effort

Misconception: SSE/SSE2 require profound changes to the algorithm and significant additional coding effort.
Reality: Applying SSE/SSE2 lets you maintain the original algorithm. With SSE/SSE2, it can now process several independent data portions at once instead of one at a time. For example, for float data type, four data portions can be processed at once.

If you use C++ Vector Classes, you can also keep your original source code—just change the original scalar data type to the corresponding Vector Class that contains several elements of the type. For example, use F32vec4 Vector Class to keep and process four float elements in one operand.

The key is to keep only homogeneous elements in an SSE/SSE2 operand. For example, when dealing with an array of four-component (x, y, z, w) vectors, resist the temptation to stick all components of a vector into an SSE/SSE2 operand. Instead, work on the four vectors in parallel, and place their four x components in one SSE/SSE2 operand, the four y components into another, as well as four z and w components. Refer to Figure 2 to see how an array of four-component vectors is multiplied by a 4x4 matrix with perspective correction (x, y, and z components of the resulting vector is divided by w).

```
for (int i = 0; i < ARRAY_COUNT; i += 4) {
  F32vec4 x = (F32vec4&)xi[i];
  F32vec4 y = (F32vec4&)yi[i];
  F32vec4 z = (F32vec4&)zi[i];
  F32vec4 w = (F32vec4&)wi[i];

  F32vec4 wr = rcp_nr(x * q[3][0] + y * q[3][1] +
    z * q[3][2] + w * q[3][3]);

  (F32vec4&)xo[i] = wr * (x * q[0][0] + y * q[0][1]
    + z * q[0][2] + w * q[0][3]);
  (F32vec4&)yo[i] = wr * (x * q[1][0] + y * q[1][1]
    + z * q[1][2] + w * q[1][3]);
  (F32vec4&)zo[i] = wr * (x * q[2][0] + y * q[2][1]
    + z * q[2][2] + w * q[2][3]);
  (F32vec4&)wo[i] = wr;
}
```

Figure 2. Four-component vectors multiplied by 4x4 matrix using SSE/SSE2 with vector classes

This approach assumes that you are using SIMD-friendly Structure of Arrays (SoA) or “Hybrid” structures for your data, as shown in Figure 3. These structures allow quick and convenient loading or storing of data elements, e.g. four floats, to/from an SSE/SSE2 operand.

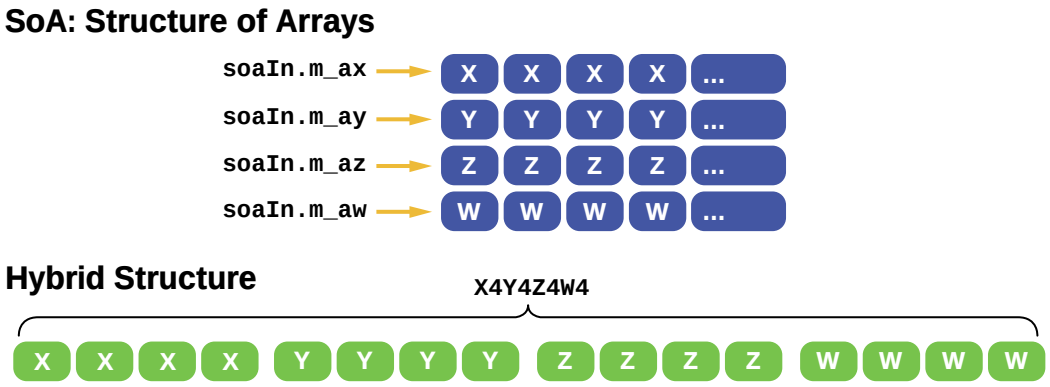
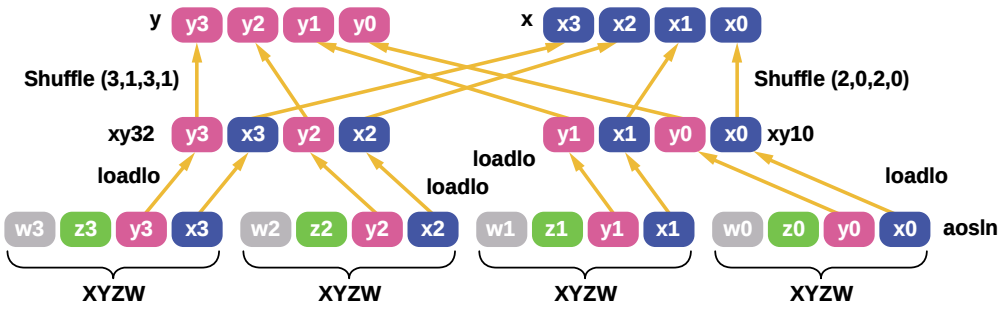


Figure 3. AoS (Array of Structures) hinders SSE/SSE2

SSE/SSE2 and Data Structures

Misconception: Because of interface/API constraints, algorithm logic, or legacy code, one still has to deal with SSE/SSE2-unfriendly data structures such as array of structures (AoS, Figure 1). These prevent any efficient application of SSE/SSE2.

Reality: It is most efficient to transform the data structure into SSE/SSE2-friendly form at design or load time. But if this is impossible due to interface/API constraints, algorithm logic, or legacy code, you can quickly modify the data structure on the fly by using SSE/SSE2. SSE/SSE2 provide a rich set of data transform operations such as half-loads, shuffles, packs, and unpacks. These help to load SSE/SSE2-unfriendly data and quickly "swizzle" it into a form suitable for fast processing with SSE/SSE2. If necessary, SSE/SSE2 also helps "unswizzle" the resulting data into the original form. As an example, Figure 4 shows an algorithm to transform AoS data into an SSE/SSE2-friendly form using SSE/SSE2 data restructuring operations.




```
void XYZWtoF32vec4(F32vec4& x, y, z, w, XYZW* aosln
{
F32vec4 xy10, xy32, zw10, zw32;
xy10 = zw10 = _mm_setzero_ps();
xy10 = _mm_loadl_pi(xy10, (__m64*)&(aosIn[0]).x);
zw10 = _mm_loadl_pi(zw10, (__m64*)&(aosIn[0]).z);
xy10 = _mm_loadh_pi(xy10, (__m64*)&(aosIn[1]).x);
zw10 = _mm_loadh_pi(zw10, (__m64*)&(aosIn[1]).z);
xy32 = zw32 = _mm_setzero_ps();
xy32 = _mm_loadl_pi(xy32, (__m64*)&(aosIn[2]).x);
zw32 = _mm_loadl_pi(zw32, (__m64*)&(aosIn[2]).z);
xy32 = _mm_loadh_pi(xy32, (__m64*)&(aosIn[3]).x);
zw32 = _mm_loadh_pi(zw32, (__m64*)&(aosIn[3]).z);
x = _mm_shuffle_ps(xy10, xy32, SHUFFLE(2,0,2,0));
y = _mm_shuffle_ps(xy10, xy32, SHUFFLE(3,1,3,1));
z = _mm_shuffle_ps(zw10, zw32, SHUFFLE(2,0,2,0));
w = _mm_shuffle_ps(zw10, zw32, SHUFFLE(3,1,3,1));
```

Figure 4. Transforming data from array of structures achieved with similar sequence of operations

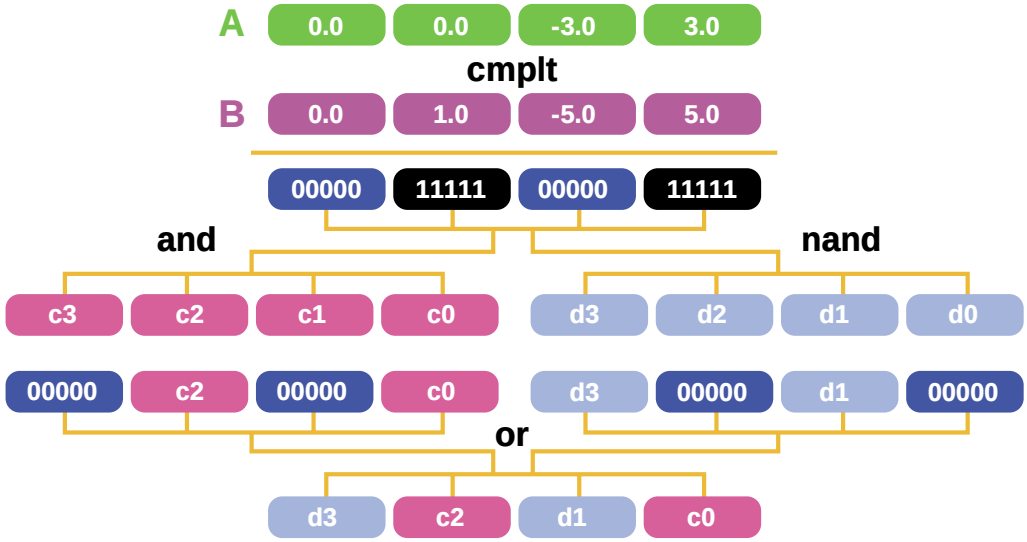
SSE/SSE2 and Conditional Code

Misconception: Algorithms containing conditions are inherently unsuitable for SSE/SSE2. With SSE/SSE2 several data portions are processed, so many different combinations of conditions should be considered. This can be unpractical to implement and inefficient to execute.

Reality: You can successfully apply SSE/SSE2 even to conditional, branchy code and get a significant performance increase. The general approach is to replace the conditional branches with logical operations or computations.

Instead of setting the processor's condition flags as scalar comparisons do, SSE/SSE2 comparisons generate bit masks. All bits of the elements where the comparison yielded true are set to 1. All bits of the elements where the comparison produced false are cleared to 0. The mask can then be processed with SSE/SSE2 bitwise logical operations.

Figure 5 illustrates a general case of selecting the result depending on a comparison. Notice that several selections are performed in parallel—without a single branch. Since conditional branches often reduce code performance, the branchless SSE/SSE2 solution offers a significant performance benefit.




```
// R = (A < B)? C : D

F32vec4 mask = cmplt(a, b);
r = (mask & c) | _mm_nand_ps(mask, d);

// OR, using F32vec4 friend function:
r = select_lt(a, b, c, d);
```

Figure 5. Selecting the result depending on a particular condition

If a special reaction is required for a certain combination of conditions, you can apply a movemask operation to the bit mask produced by the comparison. Movemask generates the hash value of the mask by taking the most significant bit of every element. For a four-element comparison (e.g., for four floats of 32-bit ints), a value in the range 0000b…1111b will be produced. If, for example, a certain code block can be skipped because all element comparisons yielded false, then a single conditional branch does the job: the movemask value is compared against zero and, if equal, it invokes a branch around the block.

Summary

Programming with SSE/SSE2 is quick and easy. It uses mature tools and greatly improves performance of a wide variety of applications. Virtually all critical code can be accelerated with SSE/SSE2 processing. SSE/SSE2 can help transform SIMD-unfriendly data structures on the fly. And SSE/SSE2 comparisons and logic help eliminate branches in conditional code.

More Info

For more information on SIMD, SSE, and SSE2, visit the Intel Developer Web site.

Try out the Intel Compiler Interactive Tutorials in the software products area of the Intel Developer site.

For more information about developing with SSE/SSE2 visit the Intel Developer Services Web site.

Author Bio

Alex Klimovitski has been with Intel GmbH for five years. He develops tools and libraries for the latest Intel® Architecture processors, and assists leading software vendors in porting and enhancing their software for Intel® processors. He shares his experience in numerous presentations, application notes, and training courses. He holds an M.S. in computer science from Berlin Polytechnic.

—End of Intel Developer Update Magazine Article—