Teaching the SIMD Execution Model: Assembling a Few Parallel Programming Skills

Ariel Ortiz
Computer Science Department
Instituto Tecnológico y de Estudios Superiores de Monterrey
Campus Estado de México
ariel.ortiz@itesm.mx

Abstract

This paper gives an overview of what the SIMD (single-instruction/multiple-data) parallel execution model is, and provides an approach for presenting it to undergraduate students. We specifically propose a few assembly language idioms and programming projects which we have successfully used in the past to teach this non-trivial topic.

Categories & Subject Descriptors

K.3 [**Computers & Education**]: Computer & Information Science Education – *Computer Science Education*.

General Terms

Performance, Design, Languages.

Keywords

SIMD, Assembly Language, x86, Computer Architecture, Parallel Computing.

1 Introduction

For more than sixteen years now, the Monterrey Institute of Technology and Higher Education has offered an assembly language course for PCs based on the x86 family of microprocessors. This is a mandatory sophomore course for undergraduate computer science and computer engineering students. Typically, many students have perceived assembly language as an intricate and archaic subject. We wanted them to realize that assembly language programming can dwell in the context of state of the art technology. In order to do this, we decided to introduce SIMD programming as one of the final topics of the course.

The SIMD (single-instruction/multiple-data) execution model allows several data elements to be processed simultaneously. For example, one SIMD instruction can add sixteen pairs of integer bytes simultaneously. This contrasts with the conventional scalar execution model (also known as SISD, single-instruction/single-data) in which only one pair of data elements are dealt with at a time. Programs that use SIMD instructions have the advantage of being considerably faster than their scalar counterparts. Unfortunately, this does not come without a price: designing SIMD enabled programs is considerably harder.

Applications that benefit from using SIMD technology include those that do many parallelizable computations over large amounts of data. Examples of these kinds of applications are 2-D/3-D graphics, motion video, image processing, speech recognition, virtual reality, audio synthesis and data compression [1].

Practically all commercial x86 microprocessors produced since the late 1990s support some sort of SIMD technology, the most important ones being the following:

- **MMX**: 64-bit integer SIMD (Intel's Pentium II, Pentium with MMX and above)
- **SSE**: 128-bit floating point SIMD (Intel's Pentium III and above)
- **SSE2**: Extended 128-bit floating point and integer SIMD (Intel's Pentium 4)
- **3DNow!**: 64-bit floating point SIMD (AMD's K6-2 and above)
- **3DNow! Professional**: 128-bit floating point and integer SIMD (AMD's Athlon XP)

The 64-bit SIMD instructions can be used from assembly language programs running on any operating system. The 128-bit SIMD instruction require a more recent operating system (such as Windows 2000/Me/XP or Linux with kernel 2.4). An alternative to using assembly language is using the MS Visual C++ compiler "intrinsics", which allow the SIMD instructions to be issued from C/C++ code. Of course, this option is only useful if using a Microsoft environment.

As instructors, the question that crops up is: How do we teach students SIMD programming skills? This is indeed a challenging topic, and unfortunately most contemporary x86 assembly language textbooks don't discuss the pragmatics of the SIMD execution model.

The remainder of this paper describes some of the most relevant SIMD elements supported by the Intel Pentium 4 processor (the most recent and powerful member of the Intel x86 family of microprocessors at the time of this writing), as well as a series of

SIGCSE'03, February 19-23, 2003, Reno, Nevada, USA.

programming idioms and programming projects that are useful for demonstrating to students how to build simple, yet interesting, SIMD enabled programs.

Please note that, for the sake of brevity, we specifically discuss the Pentium 4 processor, yet many of the things mentioned here apply as well to other processors of recent previous generations. It would be too cumbersome to explicitly state what is available or not for every individual processor type.

All assembly language code presented in this paper was designed for the Netwide Assembler (NASM) [2].

2 SIMD Support for the Pentium 4

The Pentium 4 has eight 128-bit registers. These registers are named XMM0 up to XMM7 as seen in Figure 1.

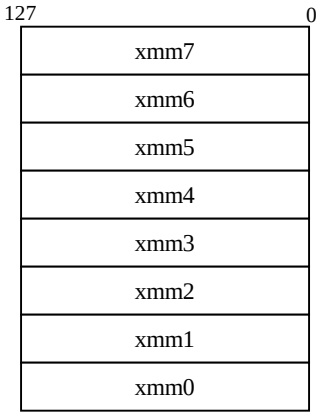

Figure 1: XMM Registers

Depending on which instructions we use, registers and memory locations may be interpreted as follows (see Figure 2):

- Sixteen packed 8-bit bytes.
- Eight packed 16-bit words.
- Four packed 32-bit double words.
- Two packed 64-bit quad words.
- One 128-bit double quad word.
- Four packed 32-bit single precision floating point numbers (C language `float` data type).
- Two packed 64-bit double precision floating point numbers (C language `double` data type).

The Pentium 4 SIMD instruction set comprises several categories of instructions, including those for arithmetic, logical, comparison, conversion, and data transfer operations. The syntax for these instructions is similar to other x86 instructions:

```
OP destination, source
```

This is interpreted as:

```
destination ← destination OP source
```

where **`OP`** stands for the specific operation. Except for the data transfer instructions, the destination operand must always be any XMM register. The source operand may be a datum stored in a memory location or in a XMM register. A few specific instructions will be discussed further on.

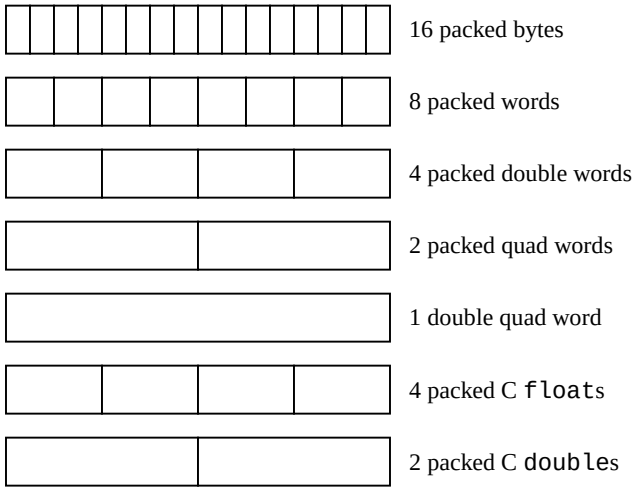

Figure 2: Pentium 4 SIMD Data Types

3 A Simple Example

This following example, originally presented by the author in [3], demonstrates the essentials of SIMD programming. Suppose we have a gray-scale bitmap image, like the one in Figure 3, and we wish to make it brighter. Each pixel is stored in one unsigned 8-bit byte contained in an array. Smaller numbers represent darker tones of gray, while larger numbers represent brighter tones. Numbers 0 and 255 represent the pure black and white colors, respectively.

To make the image brighter, we just need to add a positive integer (let's say 64 hexadecimal) to each of its pixels. In C, we would have something like this:

```
#define BRIGHTENING_K 0x64

unsigned char bitmap[BITMAP_SIZE];

size_t i;

/* Load image somehow ... */
for(i = 0; i < BITMAP_SIZE; i ++)
    bitmap[i] += BRIGHTENING_K;
```

Figure 3: Original Gray Scale Image

Unfortunately, we end up with the undesired image shown in Figure 4. This happens because of wraparound; if the result of the addition overflows (i.e., exceeds 255, which is the upper unsigned 8-bit byte limit), the result is truncated so that only the lower (least significant) bits are considered.

What we require is that whenever an addition exceeds the maximum limit, the result should saturate (clipped to a predefined data-range limit). In this case, the saturation value is 255, which represents pure white. The following C fragment takes care of saturation:

```
for(i = 0; i < BITMAP_SIZE; i ++) {

    int sum = bitmap[i] + BRIGHTENING_K;

    /* UCHAR_MAX is defined in
     * <limits.h> and is equal to 255u
     */
    if(sum > UCHAR_MAX)
        bitmap[i] = UCHAR_MAX;
    else
        bitmap[i] = (unsigned char) sum;
}
```

Now we obtain the image shown in Figure 5, which is correctly brightened as expected.

Figure 4:
Brightened Image Using Wraparound Arithmetic

Figure 5:
Brightened Image Using Saturated Arithmetic

SIMD technology allows us to do this saturated arithmetic addition on sixteen unsigned bytes in parallel using just one instruction: `paddusb`. The pseudocode for doing this would be:

1. Pack the same brightening constant byte sixteen times into the XMM0 register.
2. Repeat (BITMAP_SIZE / 16) times:
    a. Copy the next sixteen bytes from the bitmap array into the XMM1 register.
    b. Add the sixteen packed unsigned bytes contained in XMM0 to the sixteen packed unsigned bytes in XMM1. Use saturation.
    c. Copy the result of the XMM1 register back to the bitmap array from where it was originally taken.
    d. Advance bitmap array index register.

A simplified x86 assembly language version of the previous pseudocode, adapted to our example, would be as follows (comments indicate the part of the pseudocode that is being implemented):

```
section .data
    brightening_k dd 0x64646464
                  dd 0x64646464
                  dd 0x64646464
                  dd 0x64646464

    ;;; other data definitions

section .text
    movdqu  xmm0, [brightening_k] ; 1.
    mov     eax, bitmap
    mov     ecx, [bitmap_size]    ; 2.
    shr     ecx, 4
.repeat
    movdqu  xmm1, [eax]           ; 2.a
    paddusb xmm1, xmm0            ; 2.b
    movdqu  [eax], xmm1           ; 2.c
    add     eax, 16               ; 2.d
    dec     ecx
    jnz     .repeat
```

Comparing the SIMD example program to a pure C language version, the speed improvements can be quite impressive when using SIMD instructions. It's a good exercise to let students benchmark both kinds of programs so they can conclude if writing SIMD enhanced programs in assembly language is worth all the trouble.

4 SIMD Idioms

We started teaching SIMD programming in our assembly language course back in 1998, using at that time 64-bit MMX technology. From the beginning we realized that just presenting to our students a description of a SIMD instruction set was not enough to get them started building SIMD programs. We needed them to learn how to combine several instructions and data declarations in order to solve recurrent problems when designing SIMD code. At that point, we weren't too sure what these recurring problems were, but over the time we've been able to identify a few of them, and propose possible solutions.

Language specific low-level solution patterns are usually known as *idioms*. The following assembly language SIMD idioms have been designed so that students may understand the typical use of SIMD instructions. We believe that knowing and applying these idioms is a fundamental exercise for those interested in developing SIMD enhanced applications.

The following list of idioms is not exhaustive. We only included the five idioms we considered most general.

4.1 Parallel Arithmetic Idiom

The most common use of SIMD instructions is doing plain arithmetic in parallel over large amounts of data. This was exemplified in the previous section with the image brightening program. The Pentium 4 allows operating over collections of integer and floating point values. In order to perform parallel arithmetic, the program must:

1. Load into SIMD registers the multiple data values that will be operated on.
2. Perform the arithmetic operation on the registers.

3. If required, load the results from the SIMD registers back into memory.
4. Go back to 1 if more data must be processed.

Fitting more data values into the SIMD registers in point 1 generates a better overall performance because more data is processed at the same time in fewer iterations.

Point 3 is optional because under some circumstances the problem may only need to accumulate a result in a single register (for example in a vector dot product).

This same idiom may be applied in the same way to logical operations (XOR, AND, OR, etc.).

4.2 Clear (All Zeros) Register Idiom

Often, a SIMD register must be initialized with all its bits in zero. The Pentium 4 does not support an immediate (literal) data value transfer to a SIMD register. The zero value could be copied directly from memory, but a zero initialized 128-bit variable must be explicitly declared.

An XOR operation in which both source and destination operands are the same register does the trick. This is probably one of the oldest assembly language idioms still in use. So, for example, if we wish to clear the XMM1 register, the required code would be something like:

```
pxor xmm1, xmm1
```

4.3 Set (All Ones) Register Idiom

Sometimes we require a SIMD register to have all its bits set to one. This is the opposite of the previous idiom. We could try using a NOT operation over a previously cleared register. Unfortunately, the Pentium 4 does not have a SIMD NOT operation. But it does have some interesting compare instructions that come in useful for this idiom. The compare-equal instruction compares element-wise two vectors. If two elements in a certain position happen to be equal, all the bits in the corresponding position of the destination register are set to one, otherwise all those same bits are cleared (set to zero). So, if we compare a register with itself, we should have all ones as a result. For example:

```
pcmpeqb xmm2, xmm2
```

In this case, every byte sized element of XMM2 is compared to itself, and because they're equal, in that byte position all bits will be set to one.

4.4 One's Complement Idiom

As stated before, the Pentium 4 does not have a SIMD NOT instruction. But what happens if you actually do need one? A one's complement can be achieved using the XOR operation, because we know that the result of XORing any bit value *x* with a 1, is equal to NOT *x*. So if we want to obtain the NOT value of the XMM0 register (using XMM1 as an auxiliary register) we would have something like:

```
pcmpeqb xmm1, xmm1
pxor    xmm0, xmm1
```

The PANDN, described shortly, is an other common (and maybe more obvious) alternative for solving this problem.

4.5 The Mask and Merge Idiom

Suppose we have some double word size elements stored in the XMM3 register. We want to replace every occurrence of the value 7, overwriting it with the value 21. We want all other values to remain unchanged. How can we do this using SIMD instructions? The previously discussed compare-equal instruction is useful in cases like this one. It's just a matter of comparing the original values with a vector filled with 7's. Doing this produces a mask with ones and zeros in the appropriate places that later can be used together with ANDs, ORs, NOTs, and a vector filled with 21's to generate the expected result.

Basically, this idiom can be summarized as:

1. Create a mask using a compare-equal (or compare-greater-than) instruction.
2. Apply an AND to the mask in order to create a vector with the new values in the desired places, and zeros everywhere else.
3. Apply an AND to the inverse mask in order to create a vector with the original values we want to keep in place, and zeros everywhere else.
4. Apply an OR to merge the vectors produced in steps 2 and 3.

The following code fragment specifically solves the stated problem (XMM0 and XMM1 are used as auxiliary registers):

```
section .data
    k_7  dd 7, 7, 7, 7
    k_21 dd 21, 21, 21, 21
section .text
    movdqu  xmm0, xmm3
    pcmpeqd xmm3, [k_7]    ; Step 1
    movdqu  xmm1, xmm3
    pand    xmm3, [k_21]   ; Step 2
    pandn   xmm1, xmm0     ; Step 3
    por     xmm3, xmm1     ; Step 4
```

Step 3 (the inverse mask) is usually done with a PANDN instruction. It works the following way:

```
destination ← NOT(destination) AND source
```

This is where commonly a NOT operation would be used, and the reason why there's no explicit SIMD NOT instruction.

5 Course Organization and Pedagogical Approach

We've found that the basic SIMD programming skills can be taught in our assembly language course using six class hours (equivalent to two weeks out of a 16-week semester). The following table summarizes how we organize our lecture time during a typical semester.

Lecture Hours	Topic	Description
1	Introduction	In this lecture, we present a general overview of SIMD technology in the context of the x86 processor architecture (basically the same information found in sections 1 and 2 of this paper).
1	Arithmetic, Logic and Comparison Instructions	During this second lecture, we go on to explain the difference between saturated and wraparound (truncated) arithmetic, as well as how and when to use it. This corresponds to the material covered in section 3. We also review the main categories of SIMD instructions and their general usage.
1	Simple Idioms	With the information our students have from the two previous lectures, they are now able to deduce by their own the idioms in section 4.1, 4.2, 4.3 and 4.4. One at a time, we present them a general problem statement. Working collaboratively in pairs, students are usually able to arrive to a correct solution in just a few minutes. At the end of the lecture we summarize what has been achieved so far.
1	Mask and Merge Idiom	Having solved the first four idioms, students should be able to deduce as well the "mask and merge" idiom (section 4.5) on their own. Once again, we work with our students in the same way as in the previous lecture. It's very important to discuss at the end of the class the generalization of the idiom. This is because it may seem at first that a particular solution was found to a specific problem, instead of arriving to a new idiom.
2	Non-Trivial Example	The last two lectures can be dedicated to design from scratch an instructor-guided complete programming example. We also spend some time explaining how to time our routines, so we can compare the performance of regular scalar and SIMD code.

6 Programming Projects

Once students have grasped the general ideas on SIMD instructions and idioms, they're ready to solve relatively easy programming problems. Students are required to program their solutions using both scalar (using C/C++) and SIMD (using assembly language) instructions. Then, execution time is measured so they can contrast their results.

Here's a list of programming projects that we have assigned to our students in the past. Complete specifications for these projects and possible solutions are available at the following URL:

http://simd.arielortiz.com/

- **Vector Dot Product:** Two vectors are multiplied term by term, and the products are added into one final result. Applies the parallel arithmetic idiom using packed multiplication with a simultaneous addition, as well as the clear register idiom.
- **Composing Background and Foreground images:** A background is combined with some foreground details producing a new image. This is the "blue screen" effect used in the entertainment business. It uses the mask and merge idiom.
- **Alpha Dissolution Image Combination Algorithm:** Two colored images are combined into one image. Applies the parallel arithmetic idiom using packed arithmetic, shifts, and data unpacking.
- **Vigenere Cipher:** An input file is read and encrypted, resulting in a new output file. Uses the parallel arithmetic and mask and merge idioms.
- **Inverse Vigenere Cipher:** The inverse operation of the previous project. Requires also the parallel arithmetic and mask and merge idioms.
- **Gauss-Jordan Method:** Solve a system of three linear equations with three unknowns. Involves using shuffle instructions and the parallel arithmetic idiom on floating point values.

7 Conclusion

Even though it's a challenging topic, most students can learn quite satisfactorily the intricacies of SIMD programming. It's just a matter of presenting them an adequate set of idioms and allowing them to have some hands-on experience writing simple SIMD enhanced applications.

Teaching SIMD programming has been an effective way of updating our x86 assembly language course. We perceive that our students have received this topic quite positively because of its novelty.

SIMD technology is already available on any recent PC. If it's there, why not show our students how to use it? Parallel programming skills are considered essential for any computer science graduate. Teaching SIMD programming is a step forward in that direction.

8 Acknowledgements

Over the past five years, several people have provided us with the necessary software and hardware infrastructure to make SIMD teaching possible at our campus. We would like to thank particularly the support and contributions made by the following people: Maricela Quintana López, Alejandra Ocariz Villaseñor, Eduardo García García, Francisco Camargo Santacruz, José Manuel Franco Melgar and Armando García Díaz.

References

[1] Intel. *IA-32 Intel Architecture Software Developer's Manual*, Volumes I and II. Intel Corporation, 2000.

[2] NASM Development Team. *NASM — The Netwide Assembler*. 2002.

[3] Ortiz, A. An Overview of Intel's MMX Technology. *Linux Journal*, Issue #61, May 1999.