Exploiting SIMD Instructions

Felix von Leitner
CCC Berlin
felix-simd@fefe.de

August 2003

Abstract

General purpose CPUs have become powerful enough to decode and even encode MPEG audio and video in real time. These tasks have previously been the domain of special purpose hardware, DSPs and FPGAs. These feats would in general not be possible without SIMD instructions.

Agenda

1. How do CPUs work
2. Making a CPUs that is good at number crunching
3. Typical vector instruction sets: MMX, SSE
4. Vectorizing code

How do CPUs work (VAX, 386)

1. read one instruction from memory (FETCH)
2. decode instruction (DECODE)
3. load prerequisite data (READ)
4. execute instruction (EXEC)
5. write back results (WRITE)
6. repeat from step 1

How do CPUs work (VAX, 386)

```
+===+===+===+===+===+
| F | D | R | E | W |
+===+===+===+===+===+
                     +===+===+===+===+===+
                     | F | D | R | E | W |
                     +===+===+===+===+===+
```

Problem: slow.

How do CPUs work (486, VIA C3, early RISC)

```
+===+===+===+===+===+
| F | D | R | E | W |
+===+===+===+===+===+
    +===+===+===+===+===+
    | F | D | R | E | W |
    +===+===+===+===+===+
        +===+===+===+===+===+
        | F | D | R | E | W |
        +===+===+===+===+===+
```

Problem: needs good compiler, bad for architectures with few registers.

How do CPUs work (Pentium, later RISC)

- Two pipelines.
- One general purpose pipeline
- One special purpose, only executes some instructions

Problem: needs good compiler, bad for architectures with few registers.

How do CPUs work (current x86 or RISC)

- Fetch many instructions
- Calculate dependency graph
- Send instructions to functional units
- Each functional unit has a wait queue
- If you need more floating point performance, add another FP unit
- Has shadow registers, renames internally

Problem: expensive, complex. State of the art.

Register renaming

Register renaming makes it possible for a CPU to execute several iterations of the same loop in parallel.

However, the iterations must be independent.

If the architecture has few registers, it may not be possible to formulate the loop so that the iterations are independent.

Adding registers is usually not an option (breaks assembler, compiler).

Simulations have shown that adding more than 32 registers yields very little improvement, so most RISC architectures have 32 integer and 32 floating point registers. But x86 has only 7 general purpose registers!

How to make a CPU good for number crunching

There are several ways to improve floating point performance:

- Use low latency FP units
  Has lower bound. Not useful for rare insns like sqrt.

- Reduce work of FP units
  Alpha defers normalization, no exception handling.

- Add more FP units
  Useless on stack arch (387), only helps for parallel algorithms, expensive.

Out of order execution has excellent FP performance even without hand-optimized assembly core routines.

How to make a CPU good for number crunching

The number 1 performance killer in modern CPUs are branches.

Because CPUs fetch instructions and then distribute them to functional units, it needs to know which instructions to fetch and distribute. Branches make that impossible.

So CPUs either have elaborate guesswork to guess the outcome of a branch, or they execute both outcomes and then discard the wrong one (IA64).

It turns out that floating point and DSP code usually has comparatively few branches, so it is actually easier to build a CPU that is a good number cruncher than to make one that is good at interpreting perl or Java.

Why add a vector extension at all?

Many operations can be done in constant time, no matter how wide the machine word is. Examples are: XOR, AND, OR, shifting, adding.

Partitioning a large word (say, 128 bits) into 4 32-bit words is a no-op for XOR, AND, OR. Shifting needs very few additional hardware, adding and negating simply need new hardware to not carry over the carry bit at the partitions.

Subtracting can be done by adding the negated value, and negating is a XOR and an ADD.

These instructions can all easily be done in 1 machine cycle.

Historical perspective

The first general purpose CPU to add word partitioning so that one could add 4 bytes in a 32-bit register was… does anyone know?

Historical perspective

No, it’s not the Pentium MMX, that was announced in 1996 but shipped in 1997.

Sun UltraSPARC vector extension is called VIS (Visual Instruction Set) and shipped in 1995.

The HP PA-RISC vector extension is called MAX (Multimedia Acceleration eXtensions) and also shipped in 1995. HP is widely credited as being the first desktop CPU vendor to ship vector extensions.

However, the first CPU with partitioning came from Intel: the i860 (1989), a little known but very innovative CPU. The i860 was the first superscalar CPU, first multiply-add, manual pipelining… cool chip! Unfortunately, it failed in the market, because Intel’s compiler sucked.

Why add a vector extension at all?

The application that benefits obviously and trivially from integer vector instructions is image operations like alpha channel calculations.

These operations often also need saturation, i.e. ”add 5, but don’t wrap around” (adding 5 to 253 yields 255). Some vector units offer normal add and add-with-saturation instructions.

Number crunching with x86

When MMX was added, the worst bottleneck was graphics speed, in particular drawing shaded textures in VR apps and games. 3dfx didn't exist yet, quake was not on the market yet.

So MMX is an integer vector unit, it is useless for 3d calculations and number crunching.

Before modern RISC, floating point math was generally much slower than integer math. So an established optimization technique was to express and algorithm like "mp3 decoding" in integer math. On some embedded architectures this is still useful (ARM for example).

Number crunching with x86

It turns out that MPEG2 video decoding can be done completely in integer math. And like most imaging algorithms it is highly parallelizable.

These days non-vector floating point is faster than non-vector integer math on most architectures.

Still, SSE2 has 128-bit registers for floating point and integer math. floats have 32 bits. If you can write the algorithm with 16-bit integer math, you can put twice the data in each vector.

The different x86 vector extensions

MMX does integer vector operations on 8 64-bit vector registers that are mapped over the 387 floating point registers (using 8-, 16- and 32-bit integers as elements).

3dnow! is MMX plus instructions to use the MMX registers as vectors of 2 32-bit floats.

SSE adds 8 separate 128-bit registers that can be used as vectors of 4 32-bit floats or 2 64-bit doubles. SSE also adds a few more integer instructions on the MMX registers.

SSE2 adds integer vector instructions on the 128-bit registers from SSE.

The different RISC vector extensions

Altivec operates on 32 128-bit vectors of floats, 32-, 16- or 8-bit integers. Altivec is very versatile and supports most every operation.

Alpha MAX supports only integers in its normal 64-bit registers, and can only compare, logical, absolute difference, min/max and pack/unpack.

HP MAX2 operates on 16-bit values in 64-bit registers, can add, shift, saturating shift-and-add (to roll your own multiply-by-constant), logical, pack and shuffle.

SPARC VIS can add (no saturation), compare, logical, absolute difference, pack and unpack on some subsets of 64-bit registers.

Which SIMD instructions are important?

About the only ones actually used are Altivec and the various x86 extensions.

I will focus on these from now on.

If you want to see how these are used in the real world, the best place to find code using them is `ffmpeg`, in particular the platform dependent dsputils.c implementation.

ffmpeg has platform specific routines for alpha, arm, i386 (MMX, SSE), SPARC VIS (using mlib, Sun's abstraction library), Altivec, the Playstation 2 (I'm not kidding!) and SH-4. Arm and SH-4 do not actually have a vector extension, but this is a treasure trove for SIMD hackers.

What can you typically do with SIMD?

- Add and substract vectors of bytes, shorts or ints
- Same, but with saturation (”don’t go over 255 or below 0”)
- logical bitwise operations (obviously): AND, OR, XOR, NOT
- compare (element in result vector set to all-1 or all-0)
- shift (bits inside each element are shifted, not the elements themselves)
- multiply-and-add

What can you typically do with SIMD?

- absolute difference (i.e. abs(a-b)) (Altivec has this)
- min, max (more useful that you'd think!)
- packing (e.g. 2 words $\rightarrow$ 2 bytes)
- unpacking (e.g. 2 bytes $\rightarrow$ 2 words)
- shuffle (permute the elements in a vector)

What can you typically NOT do with SIMD?

- look up value in table using vector elements as index
- multiply the second word with the third word in same vector
- get sum of all 8 bytes in vector

The first one is the most serious, because often you vectorize code that has already been optimized, and people often put statically computable values in tables and later look them up.

With SIMD it's often faster to compute the value than look it up.

What does SIMD code look like?

This is actually used in MPEG encoding:

```
int sum(uint8_t* pix,int width) {
  int s,i,j;
  for (s=i=0, i<16; ++i) {
    for (j=0; j<16; ++j)
      s+=pix[j];
    pix+=width;
  }
  return s;
}
```

Poor man's SIMD

```
int sum(uint8_t* pix,int width) {
  int i,j;
  uint32_t v=0;
  for (s=i=0, i<16; ++i) {
    uint32_t x
    for (j=0; j<16; j+=4) {
      x=*(uint32_t*)pix;
      v+=((x & 0xff00ff00)>>8) + (x & 0x00ff00ff);
    }
    pix+=width;
  }
  return (v & 0xffff0000) + (v >> 16);
}
```

What does SIMD code look like?

Let's vectorize this!

```
#include <mmintrin.h>
int sum(uint8* pix,int width) {
  int i;
  __m64 x=_mm_add_pi8(*(__m64*)pix,*(__m64*)(pix+8));
  for (i=1; i<16; ++i) {
    pix+=width;
    x=_mm_add_pi8(x,*(__m64*)pix);
    x=_mm_add_pi8(x,*(__m64*)(pix+8));
  }
  /* there is no instruction to sum the elements */
} /* problem: overflow! */
```

Uh... What now?

First we need to make sure there are no overflows.

We read 8-bit values. We need to add them as 16-bit values. So first, we have to promote them.

1. copy vector
2. shift right 16-bit vector elements in copy by 8
3. AND original with `0x00ff00ff00ff00ff`
4. 16-bit vector add both

Yuck!

Run that by me again

```
0. read           (1,2,3,4,5,6,7,8)
1. copy           (1,2,3,4,5,6,7,8)      (1,2,3,4,5,6,7,8)
2. 16-bit shr 8   (0,1,0,3,0,5,0,7)      (1,2,3,4,5,6,7,8)
3. and 00ff00ff   (0,1,0,3,0,5,0,7)      (0,2,0,4,0,6,0,8)
4. sum            (1+3,3+4,5+6,7+8)
```

Now we have 4 16-bit values.

More problems

The next floating point operation after using MMX will generate an error (usually NaN). That is because MMX shares the floating point registers and sets a bit that this register was used for MMX.

You are required to run the instruction `EMMS` after the MMX code. This instructions takes 6 cycles on Pentium 3, 12 cycles on Pentium 4 (2 on Athlon). This alone can eat up all your MMX savings!

3dnow! defines a `FEMMS` ("fast EMMS") instruction that will not zero the register but just clear the MMX bit. Still, this sucks, and it is a major source of ridicule in Apple's Altivec PR.

Shuffling

Permutating the elements of a vector is impossible in MMX. You would have to write the vector to memory, permutate manually, and then read the vector back. This is prohibitively slow, don't even think about it.

SSE adds a few integer instructions as MMX extensions as well, most importantly PSHUFW. pshufw can only shuffle 16-bit elements. It takes three arguments: an immediate byte constant, a destination register, and a source register or memory location.

Every two bits in the immediate byte value are taken as index in the source vector. So, to turn (1,2,3,4) into (4,3,2,1), the immediate would be `00011011` or `0x1b`. Use `0x00` to turn (1,2,3,4) into (1,1,1,1).

Integer multiplication

Multiplying two n bit integers yields a $2n$ bit result. MMX handles this by having two multiply instructions – one for the upper n bits of the result, one for the lower n bits of the result.

This is handy because typically integer multiplication happens in ported floating point code. DSP style floating point code works on values between 0 and 1, so you work on the most significant bits of the fractional part of the number.

This means that after a multiplication you would normally shift right the result to get the most significant bits. The SSE multiplication means you don't have to shift.

What now?

As you can see, there is major bit fiddling involved in porting even this trivial function to MMX.

Even worse, many typical operations just are not there for vectors. I find it intellectually stimulating to find SIMD ways to do things.

For example, to set an MMX register to 0, you XOR it with itself. This is a well-known trick from non-MMX.

To set an MMX register to all-1, you… well, there is no decrement for SIMD. So what you do is to compare a vector with itself for equality. The result will always be true, so the result vector is all-1.

Other interesting ways to do stuff

To exchange two vectors: XOR a,b; XOR b,a; XOR a,b.

To calculate abs() on a float vector: AND each element with `0x7fffffff`.

To make a vector with 0x7fffffff: make an all-1 vector and shr by 1.

To make a vector with 0x80000000: make an all-1 vector and shl by 31.

Imaging people often need the absolute sum of differences of two images. There is no abs() for MMX. The solution is to calculate a-b and b-a... with saturation! So one is 0 and the other is positive. OR them and you are done.

What else?

SSE has 128-bit vectors, i.e. 4 floats.

There are two load instructions, a fast one that only works if the memory location is 16-byte aligned, and a slow one for the general case.

If you use gcc to write your SSE code, and you declare an `__m128` variable on the stack, gcc will not make sure it is 16-byte aligned, but will generate the aligned load and store instructions. The result is a core dump (I reported this bug a few months ago).

The problem is: gcc without `-O` generates tons of temporaries on the stack. So you will notice that suddenly, when you start compiling with `-g` to find that bug, your SSE code will start segfaulting all around you. Don't be surprised, it's a known bug.

What about floating point vectors?

Partitioning floating point words is not as easy as partitioning an integer word. CPUs with floating point vector math usually add several math units. So, normal math and vector math go to different units (on x86, the normal math unit can do stuff like `sin()` and `log()`, the vector units can only do basic arithmetic.

If you interleave vector floating point code with scalar code, you can be even faster than just the vector code!

Funky floating point math

3dnow! and SSE offer instructions to calculate an approximation of $1/x$. The background is that multiplying is always faster than dividing, so multiplying by $1/x$ is often faster than straight division, in particular if you do several divisions with the same divisor.

People often need square roots. 3dnow! and SSE do not offer square roots, but they have instructions to calculate 1/sqrt(x).

3dnow! is even funkier than SSE in that they offer two functions each, one quick one and one that you can call additionally if you need more precision.

Problem cases

In many real-life functions you end up shuffling your vectors around all the time.

For example, many imaging algorithms do a zig-zag traversal of the image, i.e. first scan line from left to right, next scan line from right to left.

One particularly nasty function is a part of MPEG 4 where they compare neighbouring pixels, but if your vector leaves the 16x16 tile to the left or right, you don't take the pixels from the neightbouring tile, but you "reflect" off the border. This is a major switch statement with a lot of evil shuffling to get it done with vector instructions at all.

Every if, switch or shuffle hurts performance.

Case study

I recently spent some time with libvorbis and an SSE manual.

I used gcov to find lines of code that are executed particularly often.

I found about 10 hot spots in the encoder code (psy.c).

Three of them looked like using vectors might be beneficial. I converted those three.

The speed-up was 25%.

It took me about a week to learn SSE enough to do this.

Go forth and multiply!

If I can do it, so can you.

Send questions to `felix-simd@fefe.de`!

BTW: If you want to learn more about computer architecture, go to a good book store and order ”the Hennessy-Patterson”. If they are any good, they’ll know.