3BA26 : Concurrent Systems I: SSE Programming

David Gregg

Department of Computer Science
Trinity College Dublin

April 15, 2009

Flynn's Taxonomy

Flynn's Taxonomy. Professor Michael J. Flynn, Stanford University, 1966.

Classifies computer architectures into four types:

- SISD
- SIMD - Single Instructions Mulptiple Datastreams
- MISD
- MIMD

SIMD

SIMD (Single Instruction Multiple Data streams).

Single instructions operate on multiple data streams.

Examples include GPUs, Modern vector processors with SIMD extensions eg SSE, Altivec.

MMX

Intel first introduced SIMD to the desktop CPU in 1997 with MMX.

8 64 bit registers, MM0 to MM7 new names for the existing Floating Point registers.

Cannot interleave FP and MMX code easily because same registers are used for both. Need to save and restore registers.

Integer only instructions.

Supports packed instructions. (2x32bit, 4*16bit, 8*8bit).

Benchmark Results (MMX v's C)

Title	Speed-up
Audio Echo Effects	5.9x
G.728 Code Book Search	2.7x
Efficient Vector/Matrix Multiply Routine	14.6x
Matrix Transpose	2x
Dot Product - 16x16 -> 32	5x
Real FIR - 16 bit	5x
Vector Arithmetic and Logic Operations	6x
Fractals with MMX Technology	1.5x
Advanced Procedural Texturing	10x
3D Bilinear Texture Mapping	7x
3D Transform	3.1x
2X 8-bit Image Scaling	13.5x
Bilinear Interpolation	3.9x

Introducing SSE

8 New 128bit registers XMM0 to XMM7.

No overlapping of existing registers. This makes programming much simpler and more efficient.

With EM64T/x86_64/AMD64 now 16 registers (XMM0 to XMM15).

Floating point support.

SSE2: Major Upgrade allowing use of integer instructions in XMM registers, more packing options (Doubles)

SSE3,4: Minor upgrades with some notable instructions such as horizontal ops, a dot product and lots of integer instructions.

AVX: Major upgrade adding 256bit registers, encryption, 3 address instructions, fused multiply-add.

Intrinsics

Programming SSE.

We could write the assembly directly. However, such code can be difficult to maintain. We therefore use *intrinsics*. Intrinsics are special functions for which the compiler generally has a specific optimisation path. They generally encode to a specific small list of sequential instructions.

Speed ↑	Assembly Intrinsics C, C++	↓ Ease of Use

Intrinsics

- Intrinsics represent a good middle ground and with modern optimising compilers we can get results nearly as good as the best hand tuned ASM.
- Code readability is not compromised much compared to to assembly language.
- We do not have to worry about register management as this is handled by the compiler

Types

The SSE intrinsics and types are defined in the `xmmintrin.h` header file. Simply `#include` this header. No special compiler options are needed.

Types (All 128 bits wide)

__m128	4 32bit single precision floats
__m128i	4 32bit integers
__m128d	2 64bit double precision floats

Load Instructions

```
__m128 _mm_load_ps(float *src)
```
Load 4 floats from a 16-byte aligned address.

```
__m128 _mm_loadu_ps(float *src)
```
Load from an unaligned address (4x slower!)

```
__m128 _mm_load1_ps(float *src)
```
Load 1 float into all 4 fields of an __m128

```
__m128 _mm_setr_ps(float a,float b,float c,float d)
```
Load 4 floats from parameters into an __m128

```
__m128 _mm_set1_ps(float w);
```
Load 1 float into all 4 fields of an __m128

Store Instructions

`void _mm_store_ps(float *dest,__m128 src)`
Store 4 floats to an aligned address.

`void _mm_storeu_ps(float *dest,__m128 src)`
Store 4 floats to unaligned address.

Aligned stores/loads must operate on a 16-byte aligned address. Some `malloc()` implementations will provide memory allocated on the correct boundary. Your compiler may also need special commandline options or `#pramga` directives.

Attempting to use an aligned operation on a non 16-byte boundary will result in a segfault. If you must use non aligned addresses, use the unaligned intrinsics. Unaligned stores/loads are however, much slower.

Arithmetic Instructions

```
__m128 _mm_add_ps(__m128 a,__m128 b)
```
Add corresponding floats (also "sub")

```
__m128 _mm_mul_ps(__m128 a,__m128 b)
```
Multiply corresponding floats (also "div")

```
__m128 _mm_min_ps(__m128 a,__m128 b)
```
Take corresponding minimum (also "max")

Other Instructions

`__m128 _mm_sqrt_ps(__m128 a)`
Take square roots of 4 floats (slow like divide)

`__m128 _mm_rcp_ps(__m128 a)`
Compute rough (12-bit accuracy) reciprocal of all 4 floats (fast as an add!)

`__m128 _mm_rsqrt_ps(__m128 a)`
Rough (12-bit) reciprocal-square-root of all 4 floats (fast)

Precision and Speed

We might see some slight difference in the values produced by simple C code and the SIMD code.

The reason for this is the reduced accuracy of some of the SSE instructions. In particular any instruction generating a reciprocal suffers from low precision. In many cases full precision is not needed. If full precision is needed, a Newton Rhapson Iteration can be performed, increasing precision (More on this later).

Because of this small precision sacrifice, a reciprocal executes as fast as an ADD instruction in SSE!

Relative Speeds

Time taken to execute an instruction:

Instruction	Speed
ADD	Fast
RCP	Fast
MUL	Slow
DIV	Slower
SQRT	Very Slow

In many cases we may find ourselves dividing by the same variable throughout our code. By first getting the reciprocal and then changing the divides to multiplies, we can gain an appreciable amount of speed in our computations. Care must be taken though if precision is necessary especially when taking the reciprocal of very large numbers.

Usage

```
/* c = a+b */
__m128 a = _mm_setr_ps(1.0f, 2.0f, 3.0f, 4.0f);
__m128 b = _mm_set1_ps(5.0f);
__m128 c = _mm_add_ps(a, b);

/* c = {6.0f, 7.0f, 8.0f, 9.0f} */
```

Original C SISD code

```
#define SIZE 4096

float vals[SIZE];
float a, b;

int main()
{
    load_vals();

    for (int i=0; i<SIZE; i++) {
        vals[i]=vals[i]*a+b;
    }

    return 0;
}
```

SIMD Code

```
#include<xmmintrin.h>
#define SIZE 4096

float vals[SIZE];
float a, b;

int main()
{

    load_vals();

    __m128 va=_mm_set1_ps(a); /* va contains 4 copies of a */
    __m128 vb=_mm_set1_ps(b); /* vb contains 4 copies of b */

    for (int i=0; i<SIZE; i+=4) { /* careful!  SIZE must be multiple of 4! */
        __m128 v=_mm_load_ps(&vals[i]); /* careful about alignment! */
        v=_mm_mul_ps(v,va);
        v=_mm_add_ps(v,vb);
        _mm_store_ps(&vals[i],v);
   }
   return 0;
}
```

Data Dependance

The following code cannot be vectorized. Why?

```
#define SIZE 4096

float vals[SIZE];
float a, b;

int main()
{
    load_vals();

    for (int i=1; i<SIZE; i++) {
        vals[i]=vals[i-1]*a+b;
    }

    return 0;
}
```