Programación SIMD para x86, AltiVec y Cell

José Miguel-Alonso
Carlos Pérez Miguel

Informe Interno
EHU-KAT-IK-02-09
Departamento de Arquitectura y
Tecnología de Computadores
UPV/EHU

Resumen

Desde la aparición de los procesadores con soporte para cálculo vectorial se ha intentado concienciar a los programadores sobre los beneficios de usar dichas técnicas en el desarrollo de sus aplicaciones. La ganacia de rendimiento que puede obtenerse mediante el uso de dichas capacidades es tan espectacular como fácil de usar. Este manual pretende ser una introducción de dichas técnicas para procesadores x86, Altivec y Cell.

1. Uso de SIMD en GCC

Para programación SIMD los compiladores usan el concepto de "intrinsics". Aparentemente, son funciones en C/C++, pero hay una traducción directa (o casi directa) a instrucción máquina.

Los intrinsics son distintos para cada juego de instrucciones. Por ejemplo, los de Altivec (PowerPC), SSE (Intel x86) y SPU (unidad vectorial del Cell), cuyos juegos de instrucciones son diferentes, tienen intrinsics incompatibles. Así, la labor de vectorizar códigos usando intrinsics nos "ata" a una arquitectura.

GCC incluye [1] métodos para escribir código vectorial independiente del juego de instrucciones subyacente. Se parte de una definición de tipos-vector:

```
typedef int v4si __attribute__ ((vector_size(4*sizeof(int))));
```

Una vez definido este tipo, se puede operar con él como si fuese un escalar, aunque realmente no lo es. Algunas limitaciones:

- El tamaño tiene que ser una potencia de dos, multiplicado por un escalar
- Se trata de una extensión de GCC. NO es parte de ANSI C, luego otros compiladores puede que no soporten la extensión.

Veamos un ejemplo. Partimos de una solución tradicional para sumar dos vectores:

```
int foo[4] = {1,5,7,9};
int bar[4] = {2,3,4,5};
```

```
for(unsigned int i=0 ; i<4 ; i++)
{
    bar[i] += foo[i];
}
```

La solución vectorizada quedaría así:

```
typedef int v4si __attribute__ ((vector_size (4*sizeof(int))));
v4si foo = {1,5,7,9};
v4si bar = {2,3,4,5};
bar += foo;
```

Con esto, se pueden usar sobre vectores las operaciones

`*, /, ^, |, &, ~, + y -`

(resta y negativo unario).

Nótese que no hay operaciones como la comparación, el cálculo de un mínimo, etc. Por lo tanto, a veces es imprescindible recurrir a los intrinsics. Un ejemplo de comparación "portable":

```
#include <stdio.h>

typedef int v4si __attribute__ ((vector_size (4*sizeof(int))));
typedef float v4sf __attribute__ ((vector_size (4*sizeof(float))));

#define INLINE inline extern __attribute__((always_inline))

INLINE int equal(v4si v1, v4si v2) {
#if defined(__ALTIVEC__)
        return vec_all_eq((vector int)v1,(vector int)v2);
#elif defined(__SSE__)
        v4si compare = __builtin_ia32_cmpeqps((v4sf)v1,(v4sf)v2);
        return __builtin_ia32_movmskps((v4sf)compare);
#else
        int * s1 = (int*)&v1;
        int * s2 = (int*)&v2;
        return (s1[0] == s2[0] && s1[1] == s2[1]
        && s1[2] == s2[2] && s1[3] == s2[3]);
#endif
}

int main(void) {
        v4si foo = {1,5,7,9};
        v4si bar = {1,5,7,9};
        if(equal(foo, bar)) {
                printf("Foo and bar had the same values.\n");
        }
        Else {
                printf("Foo and bar had different values.\n");
        }
        return 0;
}
```

En el ejemplo se ve cómo en Altivec se usan los tipos "vector xx" (por ejemplo, vector int) y los intrinsics empiezan por "vec_". En el caso de SSE, no

existe una operación de enteros, pero sí de floats (totalmente equivalente, por eso los casts). Con el compilador GCC, los intrinsics para SSE empiezan por "__builtin_ia32_", aunque es habitual que existan macros que nos permita usar "_mm_" (tal como sugiere el compilador de C de Intel).

Otro ejemplo portable Altivec – general (no para SSE):

```
#include <stdio.h>

typedef int v4si __attribute__ ((vector_size (4*sizeof(int))));

#define INLINE inline extern __attribute__((always_inline))
#define MAX(a,b) (((a)>(b))?(a):(b))

INLINE v4si v_max(v4si v1, v4si v2) {
#if defined(__ALTIVEC__)
        printf("Altivec!\n");
        return vec_max((vector int)v1,(vector int)v2);
#else
        printf("Scalar!\n");
        int * s1 = (int*)&v1;
        int * s2 = (int*)&v2;
        v4si max;
        int * smax = (int*)&max;
        smax[0] = MAX(s1[0], s2[0]);
        smax[1] = MAX(s1[1], s2[1]);
        smax[2] = MAX(s1[2], s2[2]);
        smax[3] = MAX(s1[3], s2[3]);
        return max;
#endif
}

int main(void) {
        v4si foo = {1,5,7,9};
        v4si bar = {7,6,5,4};
        bar = v_max(foo,bar);
        int * s_bar = (int*)&bar;
        for(unsigned int i=0; i<4; i++)
        {
                printf("bar: %d\n", s_bar[i]);
        }
        return 0;
}
```

Cuando los tipos que manejamos tienen distintos nombres en función del juego de instrucciones, podemos (debemos) usar unions para operar con ellos, y acceder a elementos. Nótese la forma tan compleja, en el ejemplo anterior de acceder a un elemento de un vector: declaramos un puntero a entero, le asignamos la dirección del vector, y luego indexamos. Se puede hacer de forma más sencilla:

```
#include <stdio.h>

typedef int v4si __attribute__ ((vector_size (4*sizeof(int))));
```

```
typedef union {
        v4si v;
        int s[4];
} vector;

int main(void) {
        vector foo = {{1,5,7,9}};
        vector bar = {{2,3,4,5}};
        bar.v += foo.v;
        for (unsigned int i=0; i<4; i++) {
                printf("bar2: %d\n", bar.s[i]);
        }
        return 0;
}
```

2. SIMD en PowerPC (AltiVec)

2.1. Tipos de datos

La extensión multimedia SIMD (tambien conocida como AltiVec [2, 3]) añade un conjunto de tipos de datos fundamentales, llamados tipos vector. Estos tipos se muestran en el Cuadro 1. Estos tipos son de 128 bits, representando numeros en notación decimal. Dichos registros vectoriales pueden contener:

- Dieciseis valores de 8 bits, con o sin signo (chars).
- Ocho valores de 16 bits, con o sin signo (short, short int).
- Cuatro valores de 32 bits, con o sin signo (int).
- Cuatro valores en coma flotante de precisión simple según el estándar IEE-754 (float).

Los tipos vectoriales usan el prefijo vector delante de uno de los tipos estándar de C –por ejemplo vector **signed int** y vector unigned **short**– Un tipo vectorial representa un vector de tantos elementos como quepan en un registro de 128 bits. Por lo tanto, un vector **signed int** es un operando de 128 bits que contiene cuantro enteros de 32 bits con signo. Del mismo modo, un vector **unsigned int** es un operador de 128 bits que contiene 8 valores sin signo. Notese que ya que la palabra *vector* es un tipo AltiVec, es recomendable no usarla en ningún otra parte del programa.

La introducción de los tipos de datos vectoriales permite al compilador proveer de un tipado mas fuerte y soportar la sobrecarga de operadores sobre tipos vectoriales.

2.2. Vector Intrinsics

Los intrinsics de Altivec (Vector/SIMD Multimedia Extension) están agrupados en *tres* clases:

- *Intrinsics Especificas*: aquellas que están asociadas directamente con una instrucción máquina.

Vector Data Type	Meaning	Values
vector unsigned char	Sixteen 8-bit unsigned values	0 ... 255
vector signed char	Sixteen 8-bit signed values	-128 ... 127
vector bool char	Sixteen 8-bit unsigned boolean	0 (false), 255 (true)
vector unsigned short	Eight 16-bit unsigned values	0 ... 65535
vector unsigned short int	Eight 16-bit unsigned values	0 ... 65535
vector signed short	Eight 16-bit signed values	-32768 ... 32767
vector signed short int	Eight 16-bit signed values	-32768 ... 32767
vector bool short	Eight 16-bit unsigned boolean	0 (false), 65535 (true)
vector bool short int	Eight 16-bit unsigned boolean	0 (false), 65535 (true)
vector unsigned int	Four 32-bit unsigned values	0 ... $2^32 - 1$
vector signed int	Four 32-bit signed values	-2^31 ... $2^31 - 1$
vector bool int	Four 32-bit unsigned values	0 (false), $2^31 - 1$ (true)
vector float	Four 32-bit single precision	IEEE-754 values
vector pixel	Eight 16-bit unsigned values	1/5/5/5 pixel

Cuadro 1: Vector/SIMD Multimedia Extension data types

- *Intrinsics Genéricas*: aquellas que pueden traducirse por una o más instrucciones máquina en función del tipo de los datos de entrada.
- *Intrinsics Predicados*: aquellas que comparan valores y devuelven un intefer que puede ser usado directamente como un valor o condición para un salto condicional.

Las intrinsics y los predicados AltiVec usan el prefijo vec_ delante del nombre mnemotécnico de la operación máquina correspondiente, mientras que los predicados usan los prefijos vec_all y vec_any. Cuando se compilan, los intrinsics generan una o mas instrucciones vectoriales máquina. En los Cuadros 2, 3 y 4 se muestran las distintas intrinsics Altivec.

Cuadro 2: Vector/SIMD Multimedia Extension specific and generic intrinsics

Arithmetic Intrinsics	
d = vec_abs(a)	Vector Absolute Value
d = vec_abss(a)	Vector Absolute Value Saturated
d = vec_add(a,b)	Vector Add
d = vec_addc(a,b)	Vector Add Carryout Unsigned Word
d = vec_adds(a,b)	Vector Add Saturated
d = vec_avg(a,b)	Vector Average
d = vec_madd(a,b,c)	Vector Multiply Add
d = vec_madds(a,b,c)	Vector Multiply Add Saturated
d = vec_max(a,b)	Vector Maximum
d = vec_min(a,b)	Vector Minimum
d = vec_mladd(a,b,c)	Vector Multiply Low and Add Unsigned Half Word
d = vec_mradds(a,b,c)	Vector Multiply Round and Add Saturated
d = vec_msum(a,b,c)	Vector Multiply Sum
d = vec_msums(a,b,c)	Vector Multiply Sum Saturated
d = vec_mule(a,b)	Vector Multiply Even
d = vec_mulo(a,b)	Vector Multiply Odd
d = vec_nmsub(a,b,c)	Vector Negative Multiply Subtract
d = vec_sub(a,b)	Vector Subtract
d = vec_subc(a,b)	Vector Subtract Carryout
d = vec_subs(a,b)	Vector Subtract Saturated
d = vec_sum4s(a,b)	Vector Sum Across Partial (1/4) Saturated
d = vec_sum2s(a,b)	Vector Sum Across Partial (1/2) Saturated
d = vec_sums(a,b)	Vector Sum Saturated
Rounding And Conversion	
d = vec_ceil(a)	
d = vec_ctf(a,b)	Vector Convert from Fixed-Point Word
d = vec_cts(a,b)	Vector Convert to Signed Fixed-Point Word Saturated
d = vec_ctu(a,b)	Vector Convert to Unsigned Fixed-Point Word Saturated
d = vec_floor(a)	Vector Floor
d = vec_trunc(a)	Vector Truncate
Floating-Point Estimate	
d = vec_expte(a)	
d = vec_loge(a)	Vector Log2 Estimate Floating-Point
d = vec_re(a)	Vector Reciprocal Estimate
d = vec_rsqrte(a)	Vector Reciprocal Square Root Estimate
Compare Intrinsics	
d = vec_cmpb(a,b)	
d = vec_cmpeq(a,b)	Vector Compare Equal
d = vec_cmpge(a,b)	Vector Compare Greater Than or Equal
d = vec_cmpgt(a,b)	Vector Compare Greater Than
d = vec_cmple(a,b)	Vector Compare Less Than or Equal
d = vec_cmplt(a,b)	Vector Compare Less Than
Merge Intrinsics	
d = vec_mergeh(a,b)	
d = vec_mergel(a,b)	Vector Merge Low
Permute and Select Intrinsics	
d = vec_perm(a,b,c)	
d = vec_sel(a,b,c)	Vector Select

Cuadro 3: Vector/SIMD Multimedia Extension specific and generic intrinsics

Logical Intrinsics	
d = vec_and(a,b)	
d = vec_andc(a,b)	Vector Logical AND with Complement
d = vec_nor(a,b)	Vector Logical NOR
d = vec_or(a,b)	Vector Logical OR
d = vec_xor(a,b)	Vector Logical XOR
Rotate and Shift Intrinsics	
d = vec_rl(a,b)	
d = vec_round(a)	Vector Round
d = vec_sl(a,b)	Vector Shift Left
d = vec_sld(a,b,c)	Vector Shift Left Double
d = vec_sll(a,b)	Vector Shift Left Long
d = vec_slo(a,b)	Vector Shift Left by Octet
d = vec_sr(a,b)	Vector Shift Right
d = vec_sra(a,b)	Vector Shift Right Algebraic
d = vec_srl(a,b)	Vector Shift Right Long
d = vec_sro(a,b)	Vector Shift Right by Octet
Load and Store Intrinsics	
d = vec_ld(a,b)	
d = vec_lde(a,b)	Vector Load Element Indexed
d = vec_ldl(a,b)	Vector Load Indexed LRU
d = vec_lvlx(a,b)	Load Vector Left Indexed
d = vec_lvlxl(a,b)	Load Vector Left Indexed Last
d = vec_lvrx(a,b)	Load Vector Right Indexed
d = vec_lvrxl(a,b)	Load Vector Right Indexed Last
d = vec_lvsl(a,b)	Vector Load for Shift Left
d = vec_lvsr(a,b)	Vector Load Shift Right
d = vec_stvlx(a,b)	Store Vector Left Indexed
d = vec_stvlxl(a,b)	Store Vector Left Indexed Last
d = vec_stvrx(a,b)	Store Vector Right Indexed
d = vec_stvrxl(a,b)	Store Vector Right Indexed Last
vec_st(a,b,c)	Vector Store Indexed
vec_ste(a,b,c)	Vector Store Element Indexed
vec_stl(a,b,c)	Vector Store Indexed LRU
Pack and Unpack Intrinsics	
d = vec_pack(a,b)	
d = vec_packpx(a,b)	Vector Pack Pixel
d = vec_packs(a,b)	Vector Pack Saturated
d = vec_packsu(a,b)	Vector Pack Saturated Unsigned
d = vec_unpackh(a)	Vector Unpack High Element
d = vec_unpackl(a)	Vector Unpack Low Element
Scalar Intrinsics	
d = vec_extract(a,element)	
d = vec_insert(a,b,element)	Insert Scalar into Specified Vector Element
d = vec_promote(a,element)	Promote Scalar to a Vector
d = vec_splats(a)	Splat Scalar to Vector

Cuadro 4: Vector/SIMD Multimedia Extension specific and generic intrinsics

Stream Intrinsics	
vec_dss(a)	
vec_dssall()	Vector Stream Stop All
vec_dst(a,b,c)	Vector Data Stream Touch
vec_dstst(a,b,c)	Vector Data Stream Touch for Store
vec_dststt(a,b,c)	Vector Data Stream Touch for Store Transient
vec_dstt(a,b,c)	Vector Data Stream Touch Transient
Move Intrinsics	
d = vec_mfvscr	
vec_mtvscr(a)	Vector Move to Vector Status and Control Register
Replicate Intrinsics	
d = vec_splat(a,b)	
d = vec_splat_s8(a)	Vector Splat Signed Byte
d = vec_splat_s16(a)	Vector Splat Signed Half-Word
d = vec_splat_s32(a)	Vector Splat Signed Word
d = vec_splat_u8(a)	Vector Splat Unsigned Byte
d = vec_splat_u16(a)	Vector Splat Unsigned Half-Word
d = vec_splat_u32(a)	Vector Splat Unsigned Word
All Predicates	
d = vec_all_eq(a,b)	
d = vec_all_ge(a,b)	All Elements Greater Than or Equal
d = vec_all_gt(a,b)	All Elements Greater Than
d = vec_all_in(a,b)	All Elements in Bounds
d = vec_all_le(a,b)	All Elements Less Than or Equal
d = vec_all_lt(a,b)	All Elements Less Than
d = vec_all_nan(a)	All Elements Not a Number
d = vec_all_ne(a,b)	All Elements Not Equal
d = vec_all_nge(a,b)	All Elements Not Greater Than or Equal
d = vec_all_ngt(a,b)	All Elements Not Greater Than
d = vec_all_nle(a,b)	All Elements Not Less Than or Equal
d = vec_all_nlt(a,b)	All Elements Not Less Than
d = vec_all_numeric(a)	All Elements Numeric
Any Predicates	
d = vec_any_eq(a,b)	
d = vec_any_ge(a,b)	Any Element Greater Than or Equal
d = vec_any_gt(a,b)	Any Element Greater Than
d = vec_any_le(a,b)	Any Element Less Than or Equal
d = vec_any_lt(a,b)	Any Element Less Than
d = vec_any_nan(a)	Any Element Not a Number
d = vec_any_ne(a,b)	Any Element Not Equal
d = vec_any_nge(a,b)	Any Element Not Greater Than or Equal
d = vec_any_ngt(a,b)	Any Element Not Greater Than
d = vec_any_nle(a,b)	Any Element Not Less Than or Equal
d = vec_any_nlt(a,b)	Any Element Not Less Than
d = vec_any_numeric(a)	Any Element Numeric
d = vec_any_out(a,b)	Any Element Out of Bounds

2.3. Ejemplos

A continuación se muestra un ejemplo de código y compilación sobre la plataforma Cell.

```
#include <stdio.h>
#include <altivec.h>

// Define a type we can look at either as an array of ints or as a vector.
typedef union {
        int iVals[4];
        vector signed int myVec;
} vecVar;

int main() {
        vecVar v1, v2, vConst; // define variables

        // load the literal value 2 into the 4 positions in vConst,
        vConst.myVec = (vector signed int){2, 2, 2, 2};

        // load 4 values into the 4 element of vector v1
        v1.myVec = (vector signed int){10, 20, 30, 40};

        // call vector add function
        v2.myVec = vec_add( v1.myVec, vConst.myVec );

        // see what we got!
        printf("\nResults:\nv2[0] = %d, v2[1] = %d, v2[2] = %d, v2[3] = %d\n\n",
                        v2.iVals[0], v2.iVals[1], v2.iVals[2], v2.iVals[3]);

        return 0;
}
```

En la PlayStation 3 podemos compilar y ejecutar este programa.

```
$ ppu-gcc -maltivec -o vecsum vecsum.c
```

La ejecución genera este resultado:

```
$ ./vecsum
```

Resultados:

```
v2[0] = 12, v2[1] = 22, v2[2] = 32, v2[3] = 42
```

A continuación veremos tres variantes de un código para calcular la suma de un vector de 16 números. Empezamos por una versión no vectorizada y con un bucle sin desenrollar:

```
// 16 iterations of a loop
int rolled_sum(unsigned char bytes[16]) {
        int i;
        int sum = 0;
        for (i = 0; i < 16; ++i) {
                sum += bytes[i];
        }
```

```
        return sum;
}
```

La siguiente versión desenrolla el bucle, haciendo 4 sumas en cada iteración:

```
// 4 iterations of a loop, with 4 additions in each iteration
int unrolled_sum(unsigned char bytes[16]) {
        int i;
        int sum[4] = {0, 0, 0, 0};
        for (i = 0; i < 16; i += 4) {
                sum[0] += bytes[i + 0];
                sum[1] += bytes[i + 1];
                sum[2] += bytes[i + 2];
                sum[3] += bytes[i + 3];
        }
        return sum[0] + sum[1] + sum[2] + sum[3];
}
```

Podría comprobarse que el compilador GCC (con las opciones "-O2 -funroll-loops") es capaz de hacer el desenrollado automáticamente. De hecho, desenrolla tanto que elimina completamente el bucle for.

La última versión usa vectorización. Como los 16 números a sumar caben en un vector, no es necesario bucle.

```
// Vectorized for Vector/SIMD Multimedia Extension
int vectorized_sum(unsigned char bytes[16])
{
  vector unsigned char vbytes;
  union {
    int i[4];
    vector signed int v;
  } sum;
  vector unsigned int zero = (vector unsigned int){0};

  // Perform a misaligned vector load of the 16 bytes.
  vbytes = vec_perm(vec_ld(0, bytes), vec_ld(16, bytes), vec_lvsl(0, bytes));

  // Sum the 16 bytes of the vector
  sum.v = vec_sums((vector signed int)vec_sum4s(vbytes, zero),
    (vector signed int)zero);

  // Extract the sum and return the result.
  return (sum.i[3]);
}
```

Como comentarios a este código, vemos cómo se usan las intrinsics vec_perm(), vec_ld() y vec_lvsl() para cargar los 16 caracteres de bytes[] en el vector vbytes. ¡No es tan sencillo como hacer una asignación!

- con vec_ld(0, bytes) se crea un vector temporal T1 desalineado (se pueden perder bytes de posiciones bajas de memoria)
- con vec_ld(16, bytes) se crea otro vector temporal T2 desalineado (se pueden perder bytes de posiciones altas de memoria). Entre T1 y T2 tenemos todos los bytes, sin pérdidas

- con vec_lvsl(0, bytes) se crea un tercer vector temporal T3, que contiene una lista que indica qué posiciones de T1 son válidas y dónde, y lo mismo sobre T2
- por último, vec_perm(T1, T2, T3) devuelve un vector resultado de combinar adecuadamente T1 y T2, siguiendo lo indicado en T3

Todos estos pasos han sido necesarios porque no hay garantías de que el array bytes[] esté alineado a frontera de 16 bytes. En general, no es conveniente trabajar en AltiVec con datos no alineados.

La suma se hace en dos fases. Primero, con vec_sum4s() se hacen 4 sumas parciales, cada una de cuatro caracteres, que quedan en un vector temporal (de cuatro enteros). Luego, con vec_sums() se suman los cuatro enteros parciales, y se deja el resultado en el vector sum – en realidad, en el último elemento de este vector, ya que se trata de un valor escalar.

3. Instrucciones SIMD en arquitecturas x86

El trabajo realizado por Intel para introducir extensiones SIMD en sus juegos de instrucciones no es tan limpio como el de AltiVec. De hecho, nos podemos encontrar procesadores que soportan

1. MMX – Introducido con el Pentium (1997), usado en el Pentium II.
2. SSE – Introducido con el Pentium III (1999).
3. SSE2 – Introducido con el P4 (2000).
4. SSE3 – Introducido con el P4 (Presscott, 2004).
5. SSE4 – Introducido en la arquitectura Core (2006).

El soporte de un juego de instrucciones supone el soporte de los anteriores (por ejemplo, SSE incluye a MMX, y SSE3 incluye a todos los demás). Los procesadores de AMD incluyen estas extensiones, y a veces algunas otras.

En un manual de Apple, "Introduction to AltiVec/SSE Migration Guide" [], leemos

"AltiVec and SSE are quite similar at the highest levels. They are SIMD vector units with the same vector size (128-bits) and a similar general design. SSE adds several important new features compared to AltiVec. The single and double precision floating point engines are fully IEEE-754 compliant, which means that all four rounding modes, exceptions and flags are available. Misaligned loads and stores are handled in hardware. There is hardware support for floating point division and square root. There is a Sum of Absolute Differences instruction for video encoding. All of the floating point operations provided are available in both scalar and packed variants."

Del mismo manual, vemos qué aporta cada extensión SIMD:

MMX , the first of the vector extensions provides a series of packed integer operators that utilize eight 64-bit registers (that collide with the x87 FPU registers). (. . .) The operations defined by MMX are, generally speaking, also available in a 128-bit format in SSE2. Their use on SSE2 does not

collide with the x87 unit, making SSE2 the generally preferred way to do these sorts of operations. MMX is enabled using the GCC compiler flag -mmmx. MMX is enabled by default on gcc-4.0. If MMX is enabled, the C preprocessor symbol __MMX__ is defined. MMX is disabled using the -mno-mmx flag on GCC.

SSE adds a series of packed and scalar single precision floating point operations, and some conversions between single precision and integer. SSE uses the XMM register file, which is distinct from the MMX register file and does not alias the x87 floating point stack. All operations under SSE are done under the control of the MXCSR, a special purpose control register that contains IEEE-754 flags and mask bits. SSE is enabled using the GCC compiler flag -msse. SSE is enabled by default on gcc-4.0. If SSE is enabled, the C preprocessor symbol __SSE__ is defined.

SSE2 adds a series of packed and scalar double precision floating point operations. Like SSE, SSE2 uses the XMM register file. All floating point operations under SSE2 are also done under the control of the MXCSR to set rounding modes, flags and exception masks. In addition, SSE2 replicates most of the integer operations in MMX, except modified appropriately to fit the 128-bit XMM register size. In addition, SSE2 adds a large number of data type conversion instructions. SSE2 is enabled using the GCC compiler flag -msse2. SSE2 is enabled by default on gcc-4.0. If SSE2 is enabled, the C preprocessor symbol __SSE2__ is defined.

SSE3 adds a small series of instructions mostly geared to making complex floating point arithmetic work better in some data layouts. However, since it is possible to get the same or better performance by repacking data as uniform vectors rather than non-uniform vectors ahead of time, it is not expected that most developers will need to rely on this feature. Finally, it adds a small set of additional permutes and some horizontal floating point adds and subtracts that may be of use to some developers. Further details on SSE3 can be found in the Intel's documentation. SSE3 is enabled using the GCC compiler flag -msse3. SSE3 is an optional hardware feature on MacOS X for Intel and is not enabled by default on gcc-4.0. If SSE3 is turned on, the C preprocessor symbol __SSE3__ is defined.

3.1. API para SSE

El API para SSE (y variantes) se basa, como en el caso de AltiVec, en el uso de intrinsics. Los manuales de Intel definen un conjunto de intrinsics, mientras que GCC ofrece otro. Sin embargo, a través del uso de ficheros de cabecera, es posible usar el modelo de Intel incluso usando GCC.

Los tipos de datos que se utilizan son derivados del tipo básico __m128 (un vector de 4 floats), que ocupa 128 bytes. Se pueden usar las variantes __m128i (un vector de enteros) o __m128d (un vector de dos doubles). Uno de los problemas que surgen es que el tipo __m128i no indica si tenemos 16 chars, 8 shorts ó 4 longs. Las instrucciones MMX, y algunas de SSE, usan el tipo __m64, que ocupa 64 bytes y puede contener dos floats, y el tipo __int64 para un único entero.

Para más claridad, podemos usar definiciones de tipos tal como sugiere GCC.

```
typedef int v4si __attribute__ ((vector_size (16)));
```

Esto nos asegura que el tipo v4si es equivalente a __m128i, pero dejando claro que se trata de 4 enteros con signo. GCC define estos tipos:

- Para MMX:
  - v2si vector de dos enteros de 32 bits.
  - v4hi vector de cuatro enteros de 16 bits.
  - v8qi vector de ocho enteros de 8 bits.
- Para SSE:
  - v4si vector de cuatro enteros de 32 bits – equivalente a __m128i.
  - v4sf vector de cuatro floats (precisión simple) – equivalente a __m128.
  - v2df vector de dos doubles – equivalente a __m128d.

Para complicar más las cosas, Apple define en su "Accelerate Framework" sus propios tipos. En lo posible, nos ceñiremos a los de Intel.

Siguiendo las convenciones de Intel, las intrinsics se invocan como funciones normales (tipo nombre (parámetros)). Los nombres siguen estas convenciones:

```
_mm_<intrin_op>_<suffix>
```

Donde intrin_op indica el nombre de la operación a realizar (por ejemplo, "add" para sumar) y "suffix" indica los tipos de datos sobre los que se opera. Nótese que esto significa que las intrinsics de Intel no están sobrecargadas (mientras que las de AltiVec sí lo estaban). Algunos sufijos válidos son:

- -pi-n: MMX (64-bit) vector containing packed n-bit integers
- -pu-n: MMX (64-bit) vector containing packed n-bit unsigned integers
- -epi-n: XMM (128-bit) vector containing packed n-bit integers
- -epu-n: XMM (128-bit) vector containing packed n-bit unsigned integers
- -ps: XMM (128-bit) vector containing packed single precision floating point values
- -ss: XMM (128-bit) vector containing one single precision floating point value
- -pd: XMM (128-bit) vector containing packed double precision floating point values
- -sd: XMM (128-bit) vector containing one double precision floating point value
- -si64: MMX (64-bit) vector containing a single 64-bit int
- -si128: XMM (128-bit) vector

Debido a que, como hemos dicho, no hay sobrecarga de operaciones, la lista de intrinsics es muy larga, y no merece la pena reproducirla. Lo más razonable es acudir al manual del compilador de Intel. Damos información de algunas, a título de ejemplo:

```
__m128 _mm_setzero_ps(void) // set to (0, 0, 0, 0)
__m128 _mm_set_ps1(float f) // set to (f, f, f, f)
__m128 _mm_set_ps(float w, float x, float y , float z) // set to (z,y,x,w)
__m128 _mm_setr_ps(float w, float x, float y , float z) // set to (w,x,y,z)
__m128 _mm_add_ps(__m128, __m128)
__m128 _mm_sub_ps(__m128, __m128)
__m128 _mm_mul_ps(__m128, __m128)
__m128 _mm_load_ps(float *base_addr) // load 4 floats from base_addr
__m128 _mm_loadr_ps(float *base_addr) // load in reverse order
void _mm_store_ps(float *addr, __m128 val) // write val into location addr
```

Para utilizar las intrinsics en nuestro código, tenemos que incorporar estas cabeceras:

- MMX: mmintrin.h
- SSE: xmmintrin.h
- SSE2: emmintrin.h
- SSE3: pmmintrin.h

3.2. Ejemplo

El siguiente ejemplo está sacado del manual del compilador de Intel. No es particularmente interesante, pero ilustra el uso de diferentes intrinsics. Con GCC, la orden de compilación es "gcc –msse3 –o dot_product dot_product.c".

```
#include <stdio.h>
#include <pmmintrin.h>
#define SIZE 12 //assumes size is a multiple of 4 because MMX and SSE
                //registers will store 4 elements.
float dot_product(float *a, float *b); // Plain C
float dot_product_intrin(float *a, float *b); // SSE intrinsics
short MMX_dot_product(short *a, short *b); // MMX intrinsics

int main() {
  float x[SIZE], y[SIZE];
  short a[SIZE], b[SIZE];
  int i;
  float product;
  short mmx_product;

  for(i=0; i<SIZE; i++) {
    x[i]=i;
    y[i]=i;
    a[i]=i;
    b[i]=i;
  }
  product = dot_product(x, y);
```

```
  printf("Dot␣Product␣computed␣by␣C:␣␣%f\n", product);
  product = dot_product_intrin(x,y);
  printf("Dot␣Product␣computed␣by␣SSE3␣intrinsics:␣␣␣%f\n", product);
  mmx_product = MMX_dot_product(a,b);
  printf("Dot␣Product␣computed␣by␣MMX␣intrinsics:␣␣␣%d\n", mmx_product);
  return 0;
}

float dot_product(float *a, float *b) {
  int i;
  int sum=0;
  for(i=0; i<SIZE; i++) {
    sum += a[i]*b[i];
  }
  return sum;
}

float dot_product_intrin(float *a, float *b) {
  float arr[4];
  float total;
  int i;
  __m128 num1, num2, num3, num4;
  num4= _mm_setzero_ps(); //sets sum to zero
  for(i=0; i<SIZE; i+=4) {
    num1 = _mm_loadu_ps(a+i);
      //loads unaligned array a into num1 num1 = a[3] a[2] a[1] a[0]
    num2 = _mm_loadu_ps(b+i);
      //loads unaligned array b into num2 num2 = b[3] b[2] b[1] b[0]
    num3 = _mm_mul_ps(num1, num2);
      //performs multiplication num3 = a[3]*b[3] a[2]*b[2] a[1]*b[1] a[0]*b[0]
    num3 = _mm_hadd_ps(num3, num3);
      //performs horizontal addition num3 = a[3]*b[3]+ a[2]*b[2]
      // a[1]*b[1]+a[0]*b[0] a[3]*b[3]+ a[2]*b[2] a[1]*b[1]+a[0]*b[0]
    num4 = _mm_add_ps(num4, num3); //performs vertical addition
  }
  num4= _mm_hadd_ps(num4, num4);
  _mm_store_ss(&total,num4);
  return total;
}

//MMX technology cannot handle single precision floats
short MMX_dot_product(short *a, short *b) {
  int i;
  short result, data;
  __m64 num3, sum;
  __m64 *ptr1, *ptr2;

  sum = _mm_setzero_si64(); //sets sum to zero
  for(i=0; i<SIZE; i+=4) {
   ptr1 = (__m64*)&a[i]; //Converts array a to a pointer of type
                         //__m64 and stores four elements into MMX
                         //registers
   ptr2 = (__m64*)&b[i];
   num3 = _m_pmaddwd(*ptr1, *ptr2); //multiplies elements and adds lower
```

```
                                    //elements with lower element and
                                    //higher elements with higher
    sum = _m_paddw(sum, num3);
    }
    data = _m_to_int(sum); //converts __m64 data type to an int
    sum= _m_psrlqi(sum,32); //shifts sum
    result = _m_to_int(sum);
    result= result+data;
    _m_empty(); //clears the MMX registers and MMX state.
    return result;
}
```

Destacamos del ejemplo que SSE se apaña mejor con datos no alineados, comparando con AltiVec.

4. Programación de las SPEs del Cell

Empezamos recordando que el Cell [4, 5] es una máquina heterogénea, que incluye un procesador PPE (PowerPC Procesor Element) y 8 procesadores vectoriales SPE (Synergistic Processor Elements). A veces se emplea PPU/SPU para referirse a las unidades de proceso, dejando de lado la memoria. Nosotros no haremos esas distinciones.

Todo lo dicho sobre AltiVec es aplicable a la PPU. Sin embargo, las SPUs tienen su propio juego de instrucciones que, aunque inspirado en Altivec, no es compatible.

La SPUs son procesadores vectoriales. No se trata de que se haya ampliado un juego de instrucciones escalar con unas cuantas instrucciones SIMD. Todo lo contrario: las instrucciones son SIMD y pueden de manera poco eficiente, trabajar con datos escalares. Todo está preparado para trabajar con datos de 128 bits.

También hay intrinsics para programar las SPUs. Se utilizan tipos de datos similares (si no iguales) a los de AltiVec, lo que simplifica la portabilidad del código. Estos tipos de datos se pueden ver en el Cuadro 5. Es importante destacar

Cuadro 5: Tipos de datos en el Cell

Vector Data Type	**Content**
vector unsigned char	Sixteen 8-bit unsigned chars
vector signed char	Sixteen 8-bit signed chars
vector unsigned short	Eight 16-bit unsigned halfwords
vector signed short	Eight 16-bit signed halfwords
vector unsigned int	Four 32-bit unsigned words
vector signed int	Four 32-bit signed words
vector unsigned long long	Two 64-bit unsigned doublewords
vector signed long long	Two 64-bit signed doublewords
vector float	Four 32-bit single-precision floats
vector double	Two 64-bit double precision floats quadword (16-byte)

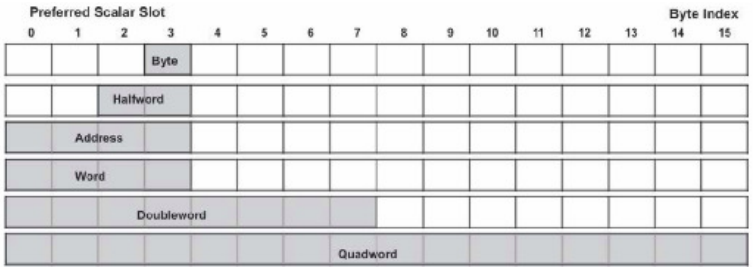

Figura 1: Datos escalares en la memoria del Cell.

que las SPUs incluidas en la PlayStation 3 son muy poco eficientes trabajando con doubles. Esto irá cambiando conforme vayan apareciendo nuevas versiones del CellBE.

Como referencia, incluimos la tabla de el Cuadro 6 con los tipos soportados por la PPU y las SPUs.

Type	SPU/PPU
vector signed char	Both
vector unsigned char	Both
vector signed short	Both
vector unsigned short	Both
vector signed int	Both
vector unsigned int	Both
vector signed long long	SPU
vector unsigned long long	SPU

Cuadro 6: Tipos de datos en las SPU/PPU

Las intrinsics para las SPUs se dividen en tres grupos:

- Específicas. Tienen una traducción una-a-una con instrucciones en ensamblador. Normalmente no se usan – si se precisa ensamblador, GCC permite hacerlo. Empiezan todas con el prefijo "si_".
- Genéricas. Las más habituales. Ofrecen un interfaz como el de las funciones C, pero se traducen directamente a una o más instrucciones en ensamblador. Empiezan todas con el prefijo "spu_".
- Compuestas. Diseñadas para comodidad del programador, se traducen en secuencias de intrinsics específicas y genéricas. También empiezan con "spu_".

Reproducimos ahora la lista de intrinsics (ver Tablas 7 a 10). Nótese que la lista incluye no sólo instrucciones para operar con datos vectoriales, sino también otras para trabajar con canales y el sistema de DMA.

Recordemos que las SPUs trabajan con vectores. Si necesitamos valores escalares, los tenemos que "colocar" dentro de un vector. Eso sí, siguiendo unas convenciones (ver Figura 1).

Intrinsic	Description
Constant Formation Intrinsics	
d = spu_splats(a)	Replicate scalar a into all elements of vector d
Conversion Intrinsics	
d = spu_convtf(a, scale)	Convert integer vector to float vector
d = spu_convts(a, scale)	Convert float vector to signed int vector
d = spu_convtu(a, scale)	Convert float vector to unsigned float vector
d = spu_extend(a)	Sign extend vector
d = spu_rountf(a)	Round double vector to float vector
Arithmetic Intrinsics	
d = spu_add(a, b)	Vector add
d = spu_addx(a, b, c)	Vector add extended
d = spu_genb(a, b)	Vector generate borrow
d = spu_genbx(a, b, c)	Vector generate borrow extended
d = spu_genc(a, b)	Vector generate carry
d = spu_gencx(a, b, c)	Vector generate carry extended
d = spu_madd(a, b, c)	Vector multiply and add
d = spu_mhhadd(a, b, c)	Vector multiply high high and add
d = spu_msub(a, b, c)	Vector multiply and subtract
d = spu_mul(a, b)	Vector multiply
d = spu_mulh(a, b)	Vector multiply high
d = spu_mulhh(a, b)	Vector multiply high high
d = spu_mulo(a, b)	Vector multiply odd
d = spu_mulsr(a, b)	Vector multiply and shift right
d = spu_nmadd(a, b, c)	Negative vector multiply and add
d = spu_nmsub(a, b, c)	Negative vector multiply and subtract
d = spu_re(a)	Vector floating-point reciprocal estimate
d = spu_rsqrte(a)	Vector floating-point reciprocal square root estimate
d = spu_sub(a, b)	Vector subtract
d = spu_subx(a, b, c)	Vector subtract extended
Byte Operation Intrinsics	
d = spu_absd(a, b)	Vector absolute difference
d = spu_avg(a, b)	Vector average
d = spu_sumb(a, b)	Vector sum bytes into shorts
Compare, Branch, and Halt Intrinsics	
d = spu_bisled(func)	Branch indirect and set link if external data
d = spu_cmpabseq(a, b)	Vector compare absolute equal
d = spu_cmpabsgt(a, b)	Vector compare absolute greater than
d = spu_cmpeq(a, b)	Vector compare equal
d = spu_cmpgt(a, b)	Vector compare greater than

Cuadro 7: Lista de las intrinsics genéricas

Intrinsic	Description
(void) spu_hcmpeq(a, b)	Halt if compare equal
(void) spu_hcmpgt(a, b)	Halt if compare greater than
d = spu_testsv(a, values)	Element-wise test for special value
Bit and Mask Intrinsics	
d = spu_cntb(a)	Vector count ones for bytes
d = spu_cntlz(a)	Vector count leading zeros
d = spu_gather(a)	Gather bits from elements
d = spu_maskb(a)	Form select byte mask
d = spu_maskh(a)	Form select halfword mask
d = spu_maskw(a)	Form select word mask
d = spu_sel(a, b, pattern)	Select bits
d = spu_shuffle(a, b, pattern)	Shuffle bytes of a vector
Logical Intrinsics	
d = spu_and(a, b)	Vector bit-wise AND
d = spu_andc(a, b)	Vector bit-wise AND with complement
d = spu_eqv(a, b)	Vector bit-wise equivalent
d = spu_nand(a, b)	Vector bit-wise complement of AND
d = spu_nor(a, b)	Vector bit-wise complement of OR
d = spu_or(a, b)	Vector bit-wise OR
d = spu_orc(a, b)	Vector bit-wise OR with complement
d = spu_orx(a)	Bit-wise OR word elements
d = spu_xor(a, b)	Vector bit-wise exclusive OR
Rotate Intrinsics	
d = spu_rl(a, count)	Element-wise bit rotate left
d = spu_rlmask(a, count)	Element-wise bit rotate left and mask
d = spu_rlmaska(a, count)	Element-wise bit algebraic rotate and mask
d = spu_rlmaskqw(a, count)	Bit rotate and mask quadword
d = spu_rlmaskqwbyte(a, count)	Byte rotate and mask quadword
d = spu_rlmaskqwbytebc(a, count)	Byte rotate and mask quadword using bit rotate count
d = spu_rlqw(a, count)	Bit rotate quadword left
d = spu_rlqwbyte(a, count)	Byte rotate quadword left
d = spu_rlqwbytebc(a, count)	Byte rotate quadword left using bit rotate count
Shift Intrinsics	
d = spu_sl(a, count)	Element-wise bit shift left
d = spu_slqw(a, count)	Bit shift quadword left
d = spu_slqwbyte(a, count)	Byte shift quadword left
d = spu_slqwbytebc(a, count)	Byte shift quadword left using bit shift count
Control Intrinsics	
(void) spu_idisable()	Disable interrupts

Cuadro 8: Lista de las intrinsics genéricas (continuación)

Intrinsic	Description
(void) spu_ienable()	Enable interrupts
(void) spu_mffpscr()	Move from floating-point status and control register
(void) spu_mfspr(register)	Move from special-purpose register
(void) spu_mtfpscr(a)	Move to floating-point status and control register
(void) spu_mtspr(register, a)	Move to special-purpose register
(void) spu_dsync()	Synchronize data
(void) spu_stop(type)	Stop and signal
(void) spu_sync()	Synchronize
Scalar Intrinsics	
d = spu_extract(a, element)	Extract vector element from vector
d = spu_insert(a, b, element)	Insert scalar into specified vector element
d = spu_promote(a, element)	Promote scalar to vector
Channel Control Intrinsics	
d = spu_readch(channel)	Read word channel
d = spu_readchqw(channel)	Read quadword channel
d = spu_readchcnt(channel)	Read channel count
(void) spu_writech(channel, a)	Write word channel
(void) spu_writechqw(channel, a)	Write quadword channel

Cuadro 9: Lista de las intrinsics genéricas (continuación)

Intrinsic	Description
spu_mfcdma32(ls, ea, size, tagid, cmd)	Initiate DMA to or from 32-bit effective address
spu_mfcdma64(ls, eahi, ealow, size, tagid, cmd)	Initiate DMA to or from 64-bit effective address
spu_mfcstat(type)	Read MFC tag status

Cuadro 10: Lista de las intrinsics compuestas

En general, se aconseja no usar escalares, o agruparlos en vectores. En caso de necesidad, se pueden usar las intrinsics spu_insert(), spu_promote() y spu_extract().

4.1. Compatibilidad PPE-SPE

Dado que los juegos de instrucciones AltiVec y SPU son similares, es bastante sencillo traducir de código vectorial AltiVec a código para SPUs. Para simplificar aún más este trabajo, existe un fichero de cabecera, "vmx2spu.h" que nos hace automáticamente esta traducción. En algunos casos se realiza un simple cambio de nombres, pero en otros es necesario algo más.

Por ejemplo, estas sentencias indican que hay una relación directa entre spu_and() y vec_and():

```
static inline vec_uchar16 vec_and(vec_uchar16 a, vec_uchar16 b)
{
  return (spu_and(a, b));
}
```

Sin embargo, estas otras indican que la implementación de una operación equivalente a vec_floor() requiere el uso de más de una docena de instrucciones (intrinsics) en una SPU.

```
static inline vec_float4 vec_floor(vec_float4 a)
{
  vec_int4 exp;
  vec_uint4 mask;

  a = spu_sub(a, (vec_float4)(spu_and(spu_rlmaska((vec_int4)a, -31),
                spu_splats((signed int)0x3F7FFFFF))));
  exp = spu_sub(127, (vec_int4)(spu_and(spu_rlmask((vec_uint4)(a), -23), 0xFF)));
  mask = spu_rlmask(spu_splats((unsigned int)0x7FFFFF), exp);
  mask = spu_sel(spu_splats((unsigned int)0), mask, spu_cmpgt(exp, -31));
  mask = spu_or(mask, spu_xor((vec_uint4)(spu_rlmaska(spu_add(exp, -1), -31)), -1));

  return ((vec_float4)(spu_andc((vec_uint4)(a), mask)));
}
```

4.2. Otro ejemplo, con DMA incluido

Vamos a ver un ejemplo completo de aplicación vectorizada (aunque no paralelizada) para su ejecución en el Cell. Se trata de una aplicación que convierte una cadena de caracteres de minúsculas a mayúsculas, utilizando las intrinsics de las SPU.

Vamos a manejar tres ficheros fuente en C. El primero de ellos es el código para una SPU, que realiza una conversión de minúsculas a mayúsculas. Lo editamos como "convert_buffer_c.c"

```
#include <spu_intrinsics.h>

unsigned char conversion_value = 'a' - 'A';

inline vec_uchar16 convert_vec_to_upper(vec_uchar16 values) {
```

```
        /* Process all characters */
        vec_uchar16 processed_values = spu_absd(values, spu_splats(conversion_value));
        /* Check to see which ones need processing (those between 'a' and 'z')*/
        vec_uchar16 should_be_processed = spu_xor(spu_cmpgt(values, 'a'-1),
        spu_cmpgt(values, 'z'));
        /* Use should_be_processed to select between the original and processed values */
        return spu_sel(values, processed_values, should_be_processed);
}

void convert_buffer_to_upper(vec_uchar16 *buffer, int buffer_size) {
        /* Find end of buffer (must be casted first because size is bytes) */
        vec_uchar16 *buffer_end = (vec_uchar16 *)((char *)buffer + buffer_size);

        while(__builtin_expect(buffer < buffer_end, 1)) {
                *buffer = convert_vec_to_upper(*buffer); buffer++;
                *buffer = convert_vec_to_upper(*buffer); buffer++;
                *buffer = convert_vec_to_upper(*buffer); buffer++;
                *buffer = convert_vec_to_upper(*buffer); buffer++;
        }
}
```

Del fichero anterior podemos destacar que contiene dos funciones. La más básica, convert_vec_to_upper() realiza, de golpe, la conversión de 16 caracteres. Lo hace asegurándose de que sólo se trata con caracteres "convertibles": minúsculas entre la "a" y la "z". La otra función simplemente trocea un buffer de entrada en bloques de 16 caracteres y llama a la conversión. Nótese que hace, en cada iteración del bucle, 4 llamadas a la función de conversión: se trata de "desenrollar" el bucle para obtener mayor eficiencia. Además, la condición de fin del bucle puede resultar extraña: en vez de un simple (buffer < buffer_end) se ha puesto (__builtin_expect(buffer < buffer_end, 1)). Se trata de dar pistas al compilador, para que sepa que, en la evaluación, lo más probable es que el resultado de la comparación sea 1 ("true"). Una optimización adicional en este código es el uso de "inline" en la función convert_vec_to_upper(), que hace que el compilador no use realmente llamadas a la función, sino que copie el código correspondiente.

El siguiente es un "driver" que se encarga de la transferencia de datos, vía DMA, con el PPE. Lo editamos como "convert_driver_c.c"

```
#include <spu_intrinsics.h>
#include <spu_mfcio.h>
typedef unsigned long long uint64;

#define CONVERSION_BUFFER_SIZE 16384
#define DMA_TAG 0

void convert_buffer_to_upper(char *conversion_buffer, int current_transfer_size);

char conversion_buffer[CONVERSION_BUFFER_SIZE];

typedef struct {
        int length __attribute__((aligned(16)));
        uint64 data __attribute__((aligned(16)));
} conversion_structure;
```

```
int main(uint64 spe_id, uint64 conversion_info_ea) {
        conversion_structure conversion_info; /* Information about the data from the PPE */

        /* We are only using one tag in this program */
        mfc_write_tag_mask(1<<DMA_TAG);

        /* Grab the conversion information */
        mfc_get(&conversion_info, conversion_info_ea, sizeof(conversion_info), DMA_TAG, 0, 0);
        spu_mfcstat(MFC_TAG_UPDATE_ALL); /* Wait for Completion */

        /* Get the actual data */
        mfc_get(conversion_buffer, conversion_info.data, conversion_info.length, DMA_TAG, 0, 0);
        spu_mfcstat(MFC_TAG_UPDATE_ALL);

        /* Perform the conversion */
        convert_buffer_to_upper(conversion_buffer, conversion_info.length);

        /* Put the data back into system storage */
        mfc_put(conversion_buffer, conversion_info.data, conversion_info.length, DMA_TAG, 0, 0);
        spu_mfcstat(MFC_TAG_UPDATE_ALL); /* Wait for Completion */
}
```

Este programa contiene el "main" que se ejecuta en una SPU e invoca a la función convert_buffer_to_upper() descrita antes. Lo más interesante de este programa es el movimiento de memoria entre la principal del Cell y la local de la SPU. Utiliza para ello funciones descritas en spu_mfcio.h, que gestionan el controlador de memoria. Todo el movimiento de memoria queda en manos de la SPU. Los pasos básicos que se dan son:

1. Recibir una estructura informativa sobre el buffer a convertir, conversion_info. La dirección en memoria principal de esta estructura la recibe el programa principal como parámetro (conversion_info_ea). El tamaño se calcula con **sizeof**(). Esperar a que la transferencia haya terminado.

2. Recibir los datos a convertir, cuya dirección en memoria principal puede obtenerse de conversion_info.data. El tamaño se encuentra en conversion_info.length. Esperar a que la transferencia haya terminado.

3. Tras realizar las conversiones oportunas, enviar los datos a memoria principal. Se sobrescribirán sobre el lugar del que se leyó. Esperar a que la transferencia haya terminado.

Por último, el código para el PPE ("ppu_dma_main.c"):

```
#include <stdio.h>
#include <libspe.h>
#include <errno.h>
#include <string.h>
#include <malloc.h>

/* embedspu actually defines this in the generated object file,
 we only need an extern reference here */
extern spe_program_handle_t convert_to_upper_handle;
```

```
/* This is the parameter structure that our SPE code expects */
/* Note the alignment on all of the data that will be passed to the SPE is 16-bytes */
typedef struct {
        int length __attribute__((aligned(16)));
        unsigned long long data __attribute__((aligned(16)));
} conversion_structure;

int main() {
        int status = 0;
        /* Pad string to a quadword -- there are 12 spaces at the end. */
        char *tmp_str = "This is the string we want to convert to uppercase.            ";
        /* Copy it to an aligned boundary */
        char *str = (char *) memalign(16, strlen(tmp_str) + 1);
        strcpy(str, tmp_str);
        /* Create conversion structure on an aligned boundary */
        conversion_structure conversion_info __attribute__((aligned(16)));

        /* Set the data elements in the parameter structure */
        conversion_info.length = strlen(str) + 1; /* add one for null byte */
        conversion_info.data = (unsigned long long)str;

        /* Create the thread and check for errors */
        speid_t spe_id = spe_create_thread(0, &convert_to_upper_handle,
        &conversion_info, NULL, -1, 0);
        if(spe_id == 0) {
                fprintf(stderr, "Unable to create SPE thread: errno= %d\n", errno);
                return 1;
        }

        /* Wait for SPE thread completion */
        spe_wait(spe_id, &status, 0);

        /* Print out result */
        printf("The converted string is: %s\n", str);

        return 0;
}
```

El programa principal del PPE se encarga de preparar el buffer a convertir, asegurándose de que se guarda en memoria alineada. Prepara la estructura conversion_info, con la longitud de string a convertir (conversion_info.length) y la dirección en MP de dicho string (conversion_info.data).

Tras preparar los datos, se prepara un thread para su ejecución en una SPU. El código a ejecutar se obtiene a través del "handle" convert_to_upper_handle. De manera externa, con el comando "embedspu", asociaremos ese handle al programa principal de la SPU. Cuando la SPU haya terminado, el PPE continúa y escribe el resultado de la conversión.

Para compilar y ejecutar el programa tenemos que dar estos pasos.

```
$ spu-gcc convert_buffer_c.c convert_driver_c.c -o spe_convert
```

Genera el fichero de salida "spe_convert", un ejecutable para una SPU. No puede usarse directamente, porque espera comunicarse con un programa en el

PPE.

```
$ embedspu.sh -m64 convert_to_upper_handle spe_convert spe_convert_csf.o
```

Genera el fichero "spe_convert_csf.o", con el handle "convert_to_upper_handle". Un programa para el PPE puede invocar dicho handle.

```
$ ppu-gcc spe_convert_csf.o ppu_dma_main.c -lspe -o dma_convert
```

Este genera el ejecutable dma_convert, que incluye tanto el código para el PPE (resultado de compilar ppu_dma_main.c) como el que va a ejecutar el SPE (que está en el fichero spe_convert_csf.o).

Referencias

[1] Vector Programming with GCC por David Chisnall. InformIT Mar 30, 2007 Article is provided courtesy of Prentice Hall Professional. http:/www.informit.com/articles/article.aspx?p=710752.

[2] AltiVec Technology Programming Interface Manua. http://www.freescale.com/files/32bit/doc/ref_manual/ALTIVECPIM.pdf.

[3] Software Development Kit for Multicore Acceleration Version 3.0 – Programming Tutorial.

[4] PPU & SPU C/C++ Language Extension Specification

[5] Programming high-performance applications on the Cell BE processor, Part 5: Programming the SPU in C/C++ J. Bartlet