eman ta zabal zazu

Universidad del País Vasco
Euskal Herriko Unibertsitatea

Facultad de Informática

Grado de Ingeniería Informática

Ingeniería de Computadores

Extensiones multimedia Intel

Autor: Mikel Sagastibelza Azcarate

Tutor: Clemente Rodríguez Lafuente

San Sebastián, Junio 2014

Resumen

Desde que se inventó el primer ordenador, uno de los objetivos ha sido que el ordenador fuese capaz de ejecutar más y más rápido, para poder así solucionar problemas más complejos.

La primera solución fue aumentar la potencia de los procesadores, pero las limitaciones físicas impuestas por la velocidad de los componentes electrónicos han obligado a buscar otras formas de mejorar el rendimiento.

Desde entonces, ha habido muchos tipos de tecnologías para aumentar el rendimiento como los multiprocesadores, las arquitecturas MIMD… pero nosotros analizaremos la arquitectura SIMD.

Este tipo de procesadores fue muy usado en los supercomputadores de los años 80 y 90, pero el progreso de los microprocesadores hizo que esta tecnología quedara en un segundo plano.

Hoy en día la todos los procesadores tienen arquitecturas que implementan las instrucciones SIMD (*Single Instruction, Multiple Data*). En este documento estudiaremos las tecnologías de SIMD de Intel SSE, AVX y AVX2 para ver si realmente usando el procesador vectorial con las instrucciones SIMD, se obtiene alguna mejora de rendimiento.

Hay que tener en cuenta que AVX solo está disponible desde 2011 y AVX2 no ha estado disponible hasta el 2013, por lo tanto estaremos trabajando con nuevas tecnologías. Además este tipo de tecnologías tiene el futuro asegurado, al anunciar Intel su nueva tecnología, AVX-512 para 2015.

Índice

Índice de figuras

Índice de tablas

Índice de gráficos

Capítulo 1

1 Planificación

1.1 Fases del proyecto

En el siguiente diagrama se puede ver de manera clara y concisa cuales son las fases que se han realizado para poder llevar a cabo el proyecto de manera exitosa.

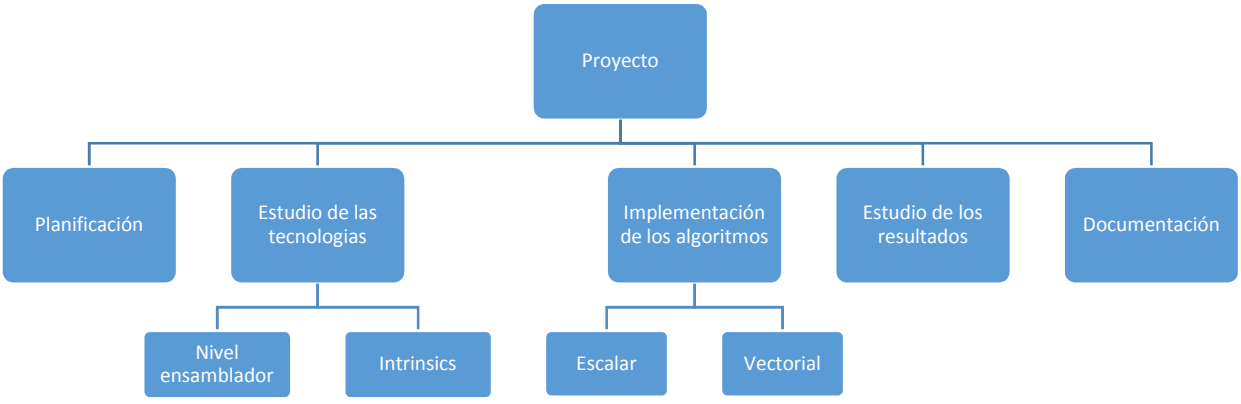


Figura 1.1: Fases del proyecto

Explicación de las diferentes fases del proyecto:

- **Planificación**: En esta fase planificaremos cuales son las tareas a realizar durante el proyecto para cumplir todos los objetivos. Se le asignará a cada tarea una estimación del tiempo que utilizaremos en ellas.
- **Estudio de las tecnologías:** Utilizando diferente bibliografía y enlaces de internet, intentaremos comprender el funcionamiento de las tecnologías. Una vez terminada es ta fase podré usar las *intrinsics* que Intel ofrece.
- **Implementación de los algoritmos:** Implementaremos unos algoritmos, donde usaré los tipos de *intrinsics* mas fundamentales. Los implementaré de dos formas, en escalar y en vectorial.
- **Estudio de los resultados:** Ejecutaré los programas implementados en la fase interior, obteniendo los resultados. Veré si la implementación vectorial realmente es más eficiente que la escalar.
- **Documentación:** Plasmaré en un documento el trabajo realizado durante todo el proyecto.

1.2 Estimación de los tiempos

He estimado los tiempos de cada fase en horas, con lo que calcularemos la estimación total del proyecto:

Fases	Estimación
Planificación	10h
Estudio de la tecnología	100h
Implementación	80h
Estudio de los resultados	15h
Documentación	100h
Total	305h

Tabla 1.1 : Estimación de los tiempos

1.3 Desarrollo en el tiempo

Comenzaré a trabajar en el proyecto en Enero al acabar los exámenes del primer cuatrimestre. En este diagrama se verá mejor:

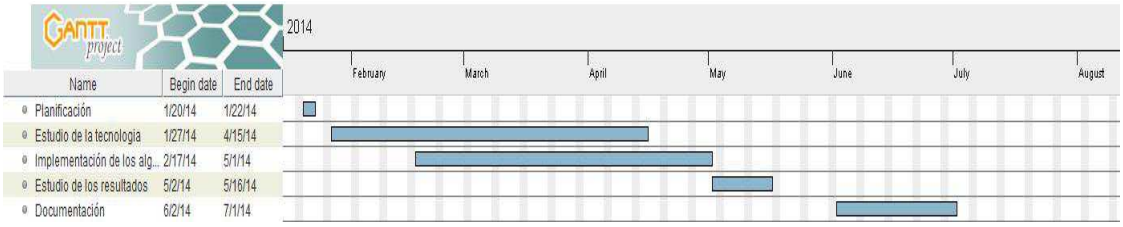


Figura 1.2: Diagrama de Gannt

En la figura 1.2 se ve como primero realizamos la planificación. Durante el segundo cuatrimestre, realizaremos las 3 siguientes fases. La segunda y la tercera fase no son paralelas al principio, pero cuando comience a entender la teoría podré empezar a implementar los primeros algoritmos, a la vez que sigo estudiando la tecnología. Igualmente la implementación

de los algoritmos me ayudará en la comprensión de la teoría. Una vez terminadas las implementaciones recabaré los resultados y los estudiaré. Es este periodo la carga de trabajos será muy irregular debido a las clases.

Finalmente en Junio al terminar los exámenes empezaré a realizar la documentación donde plasmaremos todo el trabajo realizado.

Las horas dedicadas al proyecto variarán mucho durante el segundo cuatrimestre, al tener que compaginarlo con las clases, pero al terminar las clases podré dedicarle todas las horas necesarias.

1.4 Comunicación

La comunicación se realizará principalmente en dos maneras: mediante reuniones o mediante el correo electrónico.

- Reuniones: No están planificadas ni las realizaremos con una periodicidad concreta, sino que las realizaremos cada vez que veamos necesario, por ejemplo para planificar la siguiente tarea o para resolver dudas. Además de estas reuniones, he podido ir al despacho de tutor en cualquier momento para resolver dudas.
- Email: Hemos usado el correo electronico para planificar las reuniones, para resolver dudas y para intercambiar el trabajo realizado.

Capítulo 2

2 Trabajando con vectores

2.1 Introducción

Streamig SIMD Extension, SSE, es una extensión del grupo de extensiones MMX (*MultiMedia eXtension*), para realizar operaciones vectoriales con conjuntos de números de coma flotante.

Esta tecnología fue introducida por primera vez en los procesadores Pentium III de Intel en febrero de 1999. A partir de este momento crearon diferentes versiones de SSE: SSE2, SSE3… (a partir de ahora llamaremos al conjunto de estas tecnologías SSE) con el fin de completar las operaciones que se podían realizar, incluyendo operaciones con diferentes tipos de números, como los enteros o números de coma flotante de diferente precisión. Las principales características de SSE y sus versiones es que posee 8 registros cuya longitud es de 128 bits, y que las instrucciones de nivel ensamblador son de 2 operandos: el registro destino es uno de los operandos fuentes.

Por el otro lado están las extensiones vectoriales avanzadas, AVX de sus siglas en inglés, esta tecnología fue creada en 2008, pero no fue hasta 2011, cuando estuvo disponible para el público en los procesados de arquitectura *Sandy Bridge* de Intel. La principal novedad respecto a SSE es el aumento del tamaño del vector, siendo el doble que en SSE, es decir 256 bits. También aumenta la cantidad de registros que tiene, 16. Otra diferencia respecto a SSE, es el uso de operaciones de nivel ensamblador de tres operandos: el registro destino puede ser diferente de los dos operandos fuentes. Aunque no sean iguales AVX nos permite trabajar conjuntamente con SSE.

Aun así AVX no contiene todas las operaciones disponibles en SSE. Sólo contiene las operaciones realizadas con números de coma flotante. Por esa razón se creó la versión AVX2 en el 2013. Esta versión está disponible en los procesadores de arquitectura *Ivy Bridge* de Intel. Completa AVX, incluyendo todas las operaciones con números enteros. Por lo tanto, se podrán realizar las mismas operaciones que en SSE, pero con vectores el doble de grandes.

2.2 Tipos de datos vectoriales

Los tipos de datos vectoriales utilizados en las extensiones multimedia están empaquetados en 128 o en 256 bits. Estos paquetes están formados por dos tipos de datos, números de coma flotante y números enteros. A su vez existen subdivisiones: por ejemplo en los números de coma flotante pueden ser de simple precisión o de doble precisión.

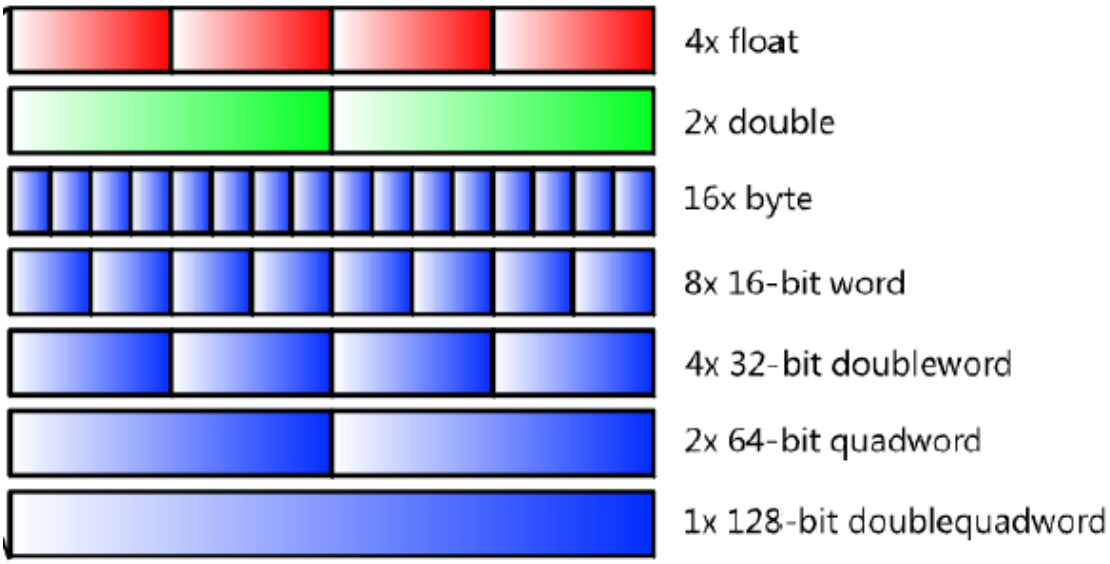


Figura 2.1: Ejemplo de tipos de datos en vectores de 128 bits

En la figura 2.1, hay 2 tipos de paquetes con números de coma flotantes y 5 tipos con números enteros. En SSE estarán disponibles todos los tipos de datos formados por números enteros y por números de coma flotante. En AVX en cambio solo se podrán usar números de coma flotante. En AVX2 solamente se usaran números enteros. Como ya se verá más adelante, no se pueden realizar todas las operaciones con todos los tipos de datos.

Intel proporciona un conjunto de primitivas, *intrinsics* en su jerga, para realizar las operaciones que se deseen con datos vectoriales. Se utilizan sufijos para informar con qué tipo de datos se trabaja. En la tabla 2.1 se relacionan los sufijos con los tipos de datos:

Sufijo	Tipo	Tamaño
s	Precisión simple	32 bits
d	Precisión doble	64 bits
i8	Entero	8 bit
i16	Entero	16 bit
i32	Entero	32 bit
i64	Entero	64 bit
i128	Entero	128 bit

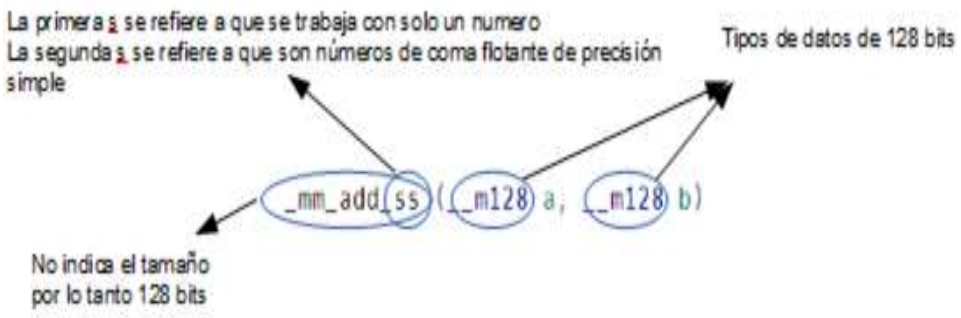
Tabla 2.1: Tipos de sufijos-

También existen otros sufijos para poder especificar cómo se trabaja: en vectorial o escalar.

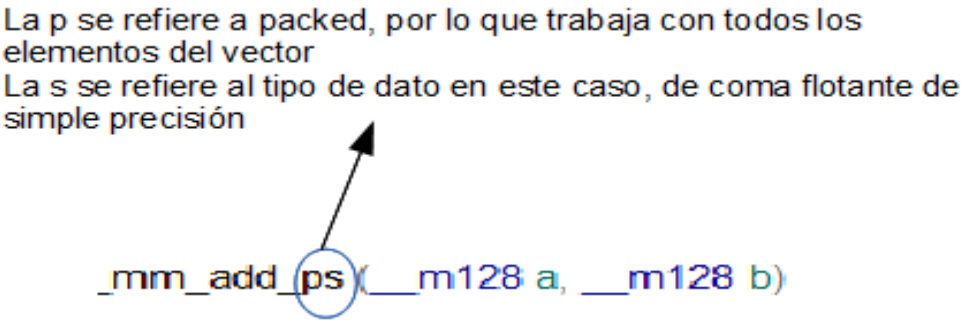
- p -> *packed*: vectorial (el número de elementos depende del tipo de dato y del tamaño total).
- s -> *single*: escalar (únicamente trabaja con el elemento guardado en los bits de menos precisión).

- u -> *unaligned* (se utiliza para trabajar con componentes que no estén alienados en 128 o 256 bit con la memoria).

En las *intrinsic* también se especifica con que tecnología se trabaja con SSE o AVX, AVX2. Esto se especifica al principio de la *intrinsic*, en SSE no se pondrá nada. En AVX y en AVX2 en cambio, tenemos un 256 que nos dice que estamos trabajando con esa tecnología.



Resultado: xmm0				32.0



Resultado: xmm0	23.0	435.0	435.0	324.0

Figura 2.2: Trabajar de forma escalar o de forma vectorial

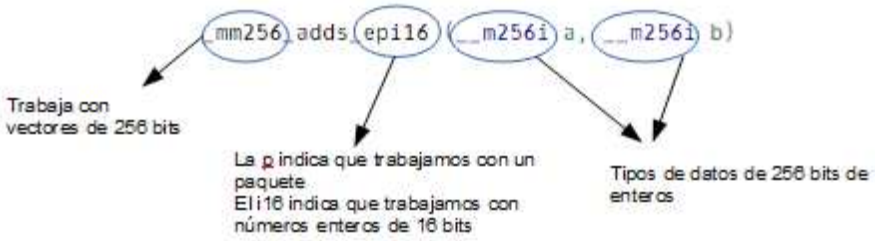


Figura 2.3: Ejemplo de intrinsic de AVX

En la figura 2.2, se ve la diferencia entre *single* y *packed*: en el primero hay solamente un número en el vector, en *packed* en cambio, el vector está lleno. En la figura 2.3, vemos la diferencia de trabajar con diferente tamaño o tipo de datos: el vector es de 256 bit y el tamaño del elemento de 16 bit, es decir, tiene 16 elementos.

Tanto SSE, AVX y AVX2 tienen registros propios, SSE dispone de 8 registros (xmm0,…, xmm7) de 128 bits cada uno, en los que se guardaran las variables que usemos, AVX y AVX2, en cambio, dispone de 16 registros (ymm0, .., ymm15) de 256 bits.

La variable a de la figura 2.4 está declarada de esta forma _m128i:

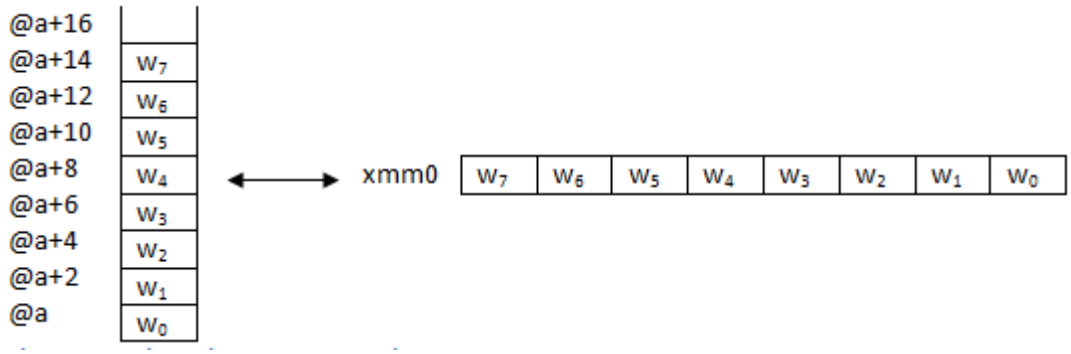


Figura 2.4: funcionamiento de la memoria

En la figura 2.4 hay un vector compuesto por 8 números enteros de 16 bits. Se puede ver como las direcciones de la memoria aumentan de dos en dos byte. Lo más importante es el orden en que guarda los números en el registro: el primer número de la memoria en los bits de menor peso del registro y así sucesivamente, hasta guardar el último número de la memoria en los bits de más peso. A este sistema se le llama *Little Endian Order.*

En las versiones de SSE, se utilizan solamente dos registros en cada operación de nivel ensamblador, por lo que por ejemplo en una división se guardara el resultado en una de las variables utilizadas, véase la figura 2.5.

```
divps    xmm0, xmm1
```

Figura 2.5:-Ejemplo dos operandos

En las versiones de AVX en cambio se utilizaran tres registros en cada operación de nivel de ensamblador, por lo tanto, en una división utilizaremos un registro diferente para guardar el resultado, véase la figura 2.6.

```
vdivps  ymm0, ymm1, ymm2
```

Figura 2.6: Ejemplo tres operandos

2.3 *Intrinsics*

Intel además del hardware necesario, también ha creado las *intrinsics* para poder trabajar más fácilmente. Estas *intrinsics* están en unas cabeceras que facilita Intel, cada versión de las extensiones tiene su propia cabecera.

SSE *<xmmintrin.h>*

SSE2 *<emmintrin.h>*

SSE3 *<pmmintrin.h>*

AVX y AVX2 *<immintrin.h>*

En estas cabeceras se encontraran funciones de todo tipo, con las que realizar cualquier operación. De todas las funciones que existen se expondrán las más utilizadas explicando para que sirven, dando un ejemplo y explicando cómo se especifica en el código.

Estas *intrinsics* están disponibles en la página web de Intel: https://software.intel.com/sites/landingpage/IntrinsicsGuide/ donde se pueden encontrar todas las *intrinsics* de todas las tecnologías disponibles hoy en día y las del futuro.

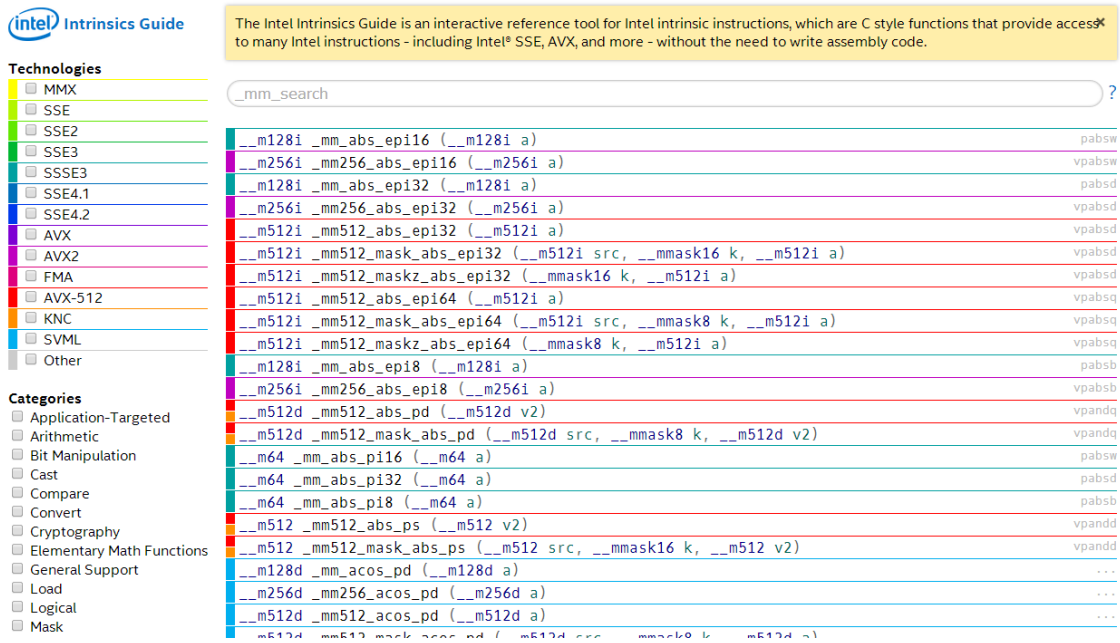


Figura 2.7: Guia de las intrinsicas

En la figura 2.7, hay a la derecha una lista de *intrinsics* de diferentes tecnologías. En la izquierda tenemos dos menús, donde se puede elegir el tipo de *intrinsics* que se muestran teniendo en cuenta la tecnología, o la categoría.

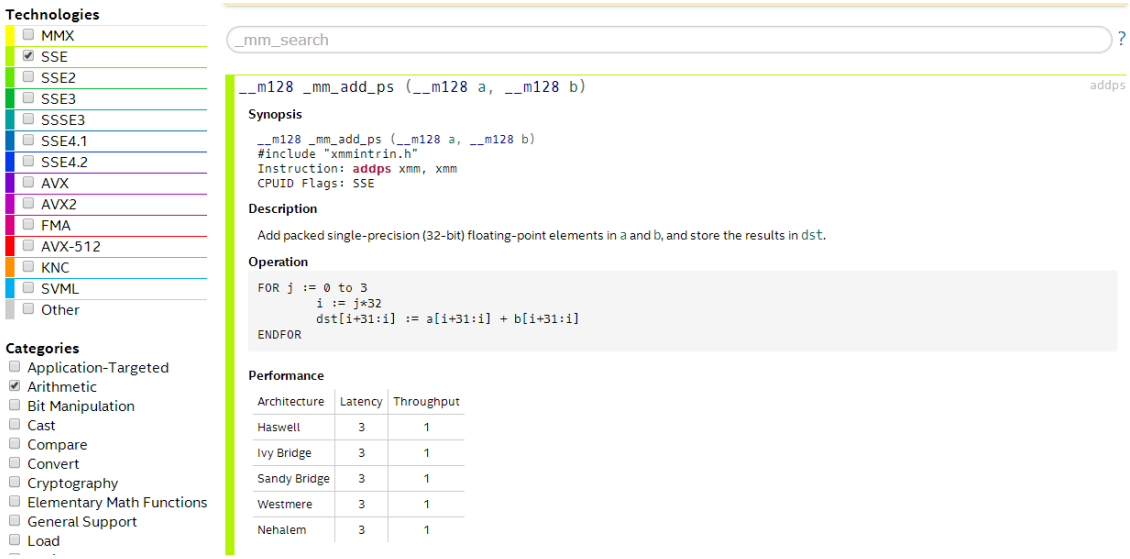


Figura 2.8: Especificación de una intrinsica

En la figura 2.8, se han elegido una tecnología (SSE) y una categoría (aritméticas), además de una intrínseca, en este caso una que suma dos vectores de 128 bits. Se puede ver como la página web nos ofrece mucha información sobre la *intrinsic*, desde la instrucción de nivel de ensamblador, hasta al código escalar de la operación.

Después de realizar unos cuantos programas de prueba para probar diferentes aspectos de las extensiones multimedia, se ha concluido que las que se explicaran a continuación son las más importantes, pudiendo realizar la mayoría de los algoritmos con ellas. También se ha podido comprobar que aunque los procesadores cumplían los requisitos requeridos para usar AVX2 (arquitectura *Ivy Bridge*), finalmente no pueden ejecutar programas que incluyan *intrinsics* de esta tecnología. Por lo tanto todas las explicaciones sobre AVX2 que se den serán teóricas.

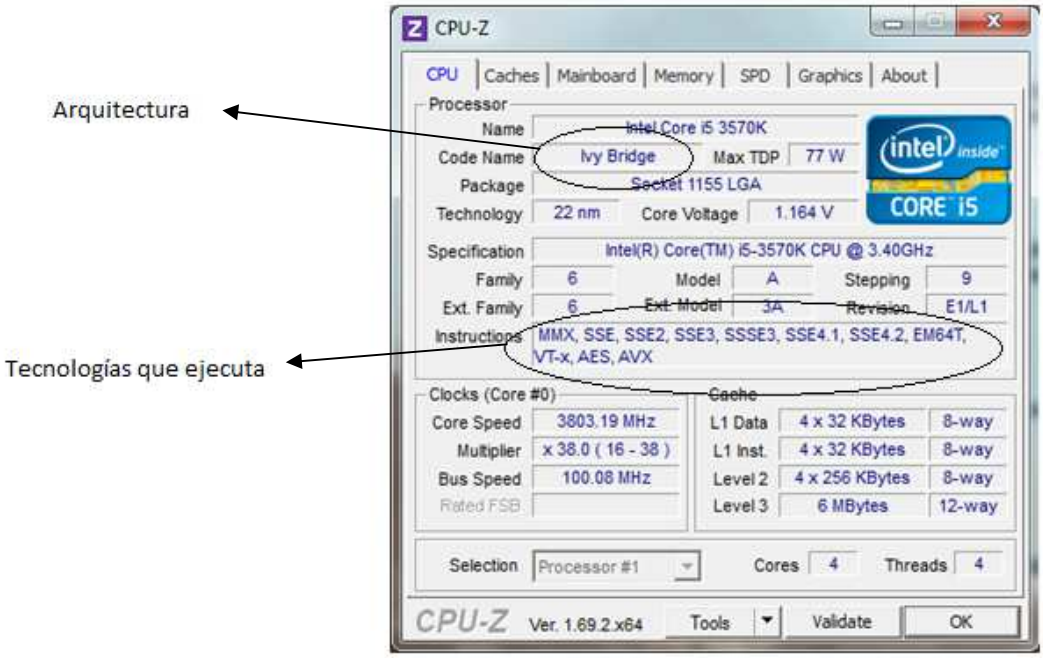


Figura 2.9: ¿Ejecuta AVX2?

En la figura 2.9 vemos como el procesador tiene la arquitectura *Ivy Bridge*, y también las tecnologías que ejecuta. Como se puede ver en ellas no está incluida AVX2

Estas son las funciones que se explicarán:

Declarar las variables, Load/Store, iniciación de variables, aritméticas, lógicas, condicionales y reducciones

En todas ellas se verá un ejemplo usando vectores de 4 unidades de números de coma flotantes de simple precisión. Por último se explicara cómo se utiliza cada tipo de función, en cada tecnología.

Para ello se utilizará un pequeño algoritmo donde se pueden apreciar todas los tipos de función. De este algoritmo se sacaran los ejemplos para poder explicar mejor como funciona cada *intrinsic*.

```
data_t ejemplo ( data_t *va,data_t *resultado ,float b, int n )
{
    float lag[N],suma;
    int i;
    for(i=0;i<n;i++){
        lag[i]=va[i]+b;
        if(lag[i]>20)
            lag[i]=0.0;
            resultado[i]=lag[i];
        }
    suma=0.0;
    for(i=0;i<n;i++)
        suma+=resultado[i];
    return suma;
}
```

Al pasar de código escalar al código vectorial se han fusionado los dos *for* que se muestran en el código escalar convirtiéndolo en un solo *for*. Este sería el código vectorizado del algoritmo con *intrinsics* SSE y vectores de números de coma flotante

```
data_t ejemplo ( data_t *va,data_t *resultado ,float b, int n ){
    int i;
    float suma;
    __m128 vsuma, comparar, lag, cond,a,v_b;
    vsuma=_mm_setzero_ps ();
    comparar=_mm_set_ps(20.0,20.0,20.0,20.0);
    v_b= _mm_set1_ps (b);
    for(i=0;i<n;i+=4){
        a= _mm_load_ps (va+i);
        lag= _mm_add_ps (a,v_b);
        cond = _mm_cmple_ps (lag,comparar);
        lag = _mm_and_ps (lag,cond);
        _mm_store_ps (&resultado[i], lag);
        vsuma= _mm_add_ps (lag,vsuma);
    }
    vsuma = _mm_hadd_ps (vsuma,vsuma);
    vsuma = _mm_hadd_ps (vsuma,vsuma);
    _mm_store_ss (&suma, vsuma);
    return suma;
}
```

Este sería el código vectorizado del algoritmo con *intrinsics* AVX y vectores de números de coma flotante:

```
data_t ejemplo_vec ( data_t *va,data_t *resultado ,float b, int
n ){
    int i;
    float suma;
    __m256 vsuma, comparar, lag, cond,a,v_b;
    vsuma=_mm256_setzero_ps ();

comparar=_mm256_set_ps(20.0,20.0,20.0,20.0,20.0,20.0,20.0,20.0);
    v_b= _mm256_set1_ps (b);
    for(i=0;i<n;i+=8){
        a= _mm256_load_ps (va+i);
        lag= _mm256_add_ps (a,v_b);
        cond = _mm256_cmp_ps (lag,comparar,18);
        lag = _mm256_and_ps (lag,cond);
        _mm256_store_ps (&resultado[i], lag);
        vsuma= _mm256_add_ps (lag,vsuma);
    }
    vsuma = _mm256_hadd_ps (vsuma,vsuma);
    vsuma = _mm256_hadd_ps (vsuma,vsuma);
    _mm_store_ss (&suma, _mm256_castps256_ps128 (vsuma));
    return suma;
}
```

Antes de explicar cada tipo de función se revisará cuales de ese tipo aparecen en el ejemplo, poniendo una flecha grande a la derecha, las funciones tendrán una flecha pequeña a la izquierda, si ya se han explicado; las que están por explicar no tendrán ninguna flecha.

Aquí están todos los tipos de *intrinsics* utilizadas:

2.3.1 Declarar las variables

En estos programas se usarán dos tipos de variable totalmente nuevas. Son los vectores de 128 y 256 bits que se utilizarán en los programas. Así se declaran los vectores de 128 bits:

```
__m128 vsuma, comparar, lag, cond,a,v_b;
```

Y así los de 256 bits:

```
__m256 vsuma, comparar, lag, cond,a,v_b;
```

2.3.2 Load/Store

Estas funciones son las que se usan para leer y escribir en la memoria, por lo que son imprescindibles en cualquier programa.

La función load sirve para leer desde la memoria y guardarla en una variable; la función store, en cambio, vale para guardar el contenido de una variable en la memoria.

Con la función load se leerá o se escribirá un rango de 128 o 256 bits de la memoria, pudiendo ser, cuatro o ocho números de coma flotante y se guardará en un vector. En el caso de la función store, se realizará justo lo contrario, leer de una variable y escribir en la memoria.

Esta es la *intrinsic* utilizada para la figura 2.10: a = _mm_load_ps (v) o _mm_store_ps (v, a), donde ***v*** es la dirección de la memoria de donde se lee/escribe y ***a*** la variable donde se escribe/lee. En los dos casos los elementos tendrán un valor de 3.0:

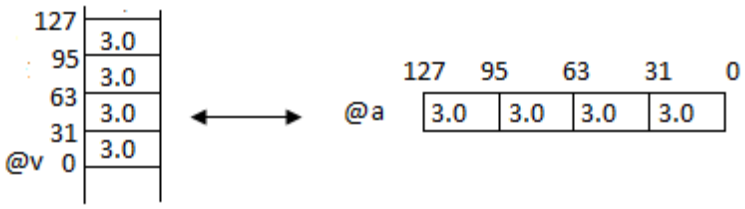


Figura 2.10: Ejemplo Load/Store

En la figura 2.10 se mueve de la memoria a la variable, o viceversa, el vector compuesto por treses, utilizando el ya mencionado *Little Endian order.*

Ahora se explicara la especificación de las dos operaciones en cada tecnología.

2.3.2.1 Load/Store en SSE

Así es como se especifica la operación en la guía que Intel ofrece:

```
__m128 _mm_load_ps (float const* mem_addr)
void _mm_store_ps (float* mem_addr, __m128 a)
```

Así se usan en el ejemplo:

```
data_t ejemplo_vec ( data_t *va,data_t *resultado ,float b,
int n ){
    int i;
    float suma;
    __m128 vsuma, comparar, lag, cond,a,v_b;
    vsuma=_mm_setzero_ps ();
    comparar=_mm_set_ps(20.0,20.0,20.0,20.0);
    v_b= _mm_set1_ps (b);
    for(i=0;i<n;i+=4){
        a= _mm_load_ps (va+i);                    <----------
        lag= _mm_add_ps (a,v_b);
        cond = _mm_cmple_ps (lag,comparar);
        lag = _mm_and_ps (lag,cond);
        _mm_store_ps (&resultado[i], lag);        <----------
        vsuma= _mm_add_ps (lag,vsuma);
    }
    vsuma = _mm_hadd_ps (vsuma,vsuma);
    vsuma = _mm_hadd_ps (vsuma,vsuma);
    _mm_store_ss (&suma, vsuma);                  <----------
    return suma;
```

Se necesita una dirección de la memoria y una variable en los dos casos. En el load se utiliza la variable para guardar el contenido de la memoria, y en el store se guardara el contenido de la variable en la memoria. Este es el código equivalente que muestra de una forma escalar lo que hace la *intrinsic* vectorial:

```
a=_mm_load_ps (va+i);
for(i=0;i<4;i++){
      a[i]=va[i];
}
_mm_store_ps (&resultado[i], lag);
for(i=0;i<4;i++){
      resultado[i]=lag[i];
}
_mm_store_ss (&suma, vsuma);
suma=vsuma[0];
```

2.3.2.2 Load/Store en AVX

Así es como se especifica la operación en la guía que Intel ofrece:

```
__m256 _mm256_load_ps (float const * mem_addr)
void _mm256_store_ps (float * mem_addr, __m256 a)
```

Así se usan en el ejemplo:

```
data_t ejemplo_vec ( data_t *va,data_t *resultado ,float b, int n ){
    int i;
    float suma;
    __m256 vsuma, comparar, lag, cond,a,v_b;
    vsuma=_mm256_setzero_ps ();
    comparar=_mm256_set_ps(20.0,20.0,20.0,20.0,20.0,20.0,20.0,20.0);
    v_b= _mm256_set1_ps (b);
    for(i=0;i<n;i+=8){
        a= _mm256_load_ps (va+i);
        lag= _mm256_add_ps (a,v_b);
        cond = _mm256_cmp_ps (lag,comparar,18);
        lag = _mm256_and_ps (lag,cond);
        _mm256_store_ps (&resultado[i], lag);
        vsuma= _mm256_add_ps (lag,vsuma);
    }
    vsuma = _mm256_hadd_ps (vsuma,vsuma);
    vsuma = _mm256_hadd_ps (vsuma,vsuma);
    _mm_store_ss (&suma, _mm256_castps256_ps128 (vsuma));
    return suma;
}
```

Se necesitan una dirección de memoria y una variable en los dos casos en el load se utilizará la variable para guardar el contenido de la memoria, y en el store se guardará el contenido de la variable en la memoria.

En el segundo store, ocurre un caso especial, queremos usar un tipo de store solo disponible en SSE, por lo tanto se utiliza una *intrinsic* de casting para poder usar ese tipo de store. Este es el código equivalente que nos muestra de una forma escalar lo que hace la *intrinsic* vectorial:

```
a= _mm256_load_ps (va+i);
for(i=0;i<8;i++){
      a[i]=va[i];
}
_mm256_store_ps (&resultado[i], lag);
for(i=0;i<8;i++){
      resultado[i]=lag[i];
}
_mm_store_ss (&suma, _mm256_castps256_ps128 (vsuma));
suma=vsuma[0];
```

En el código se puede ver que la única diferencia con SSE, reside en el tamaño de la variable, es decir, en el número de elementos del vector

2.3.2.3 Load/Store en AVX2

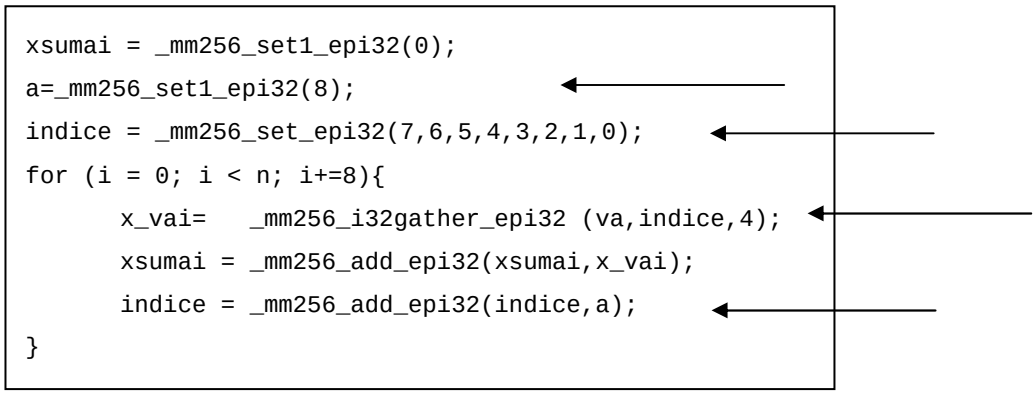
Así es como se especifica la operación en la guía que Intel ofrece:

```
__m256i _mm256_i32gather_epi32 (int const* base_addr, __m256i vindex, const int scale)
        void _mm256_maskstore_epi32 (int* mem_addr, __m256i mask, __m256i a)
```

Y así como se especificaría la operación para usarla en el algoritmo:

```
a= _mm256_i32gather_epi32  (v,indice,4)
_mm256_store_ps (v, maskara, resultado)
```

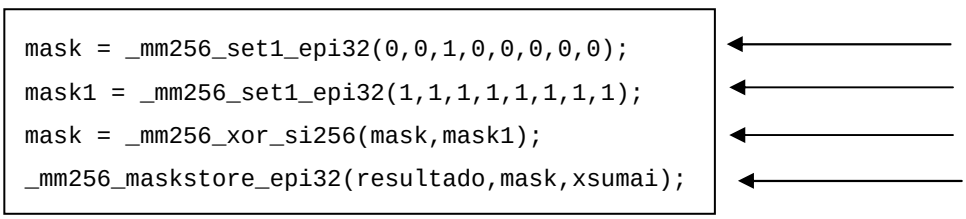
Estas *intrinsics* son completamente diferentes a las utilizadas en SSE o AVX. En la función load se puede ver que además de la dirección de memoria de la que queramos leer y la variable de destino tenemos otras dos variables de entrada. Con la primera, ***indice***, se puede elegir los ocho elementos que se cargarán en la variable. Por ejemplo, si se completa la variable ***indice*** con los valores 0, 1, 2, 3, 4, 5, 6 y 7 se escribirán los primeros ocho elementos de la memoria en la variable. Esta opción da la posibilidad de leer elementos de forma no consecutiva jugando con el vector, en nuestro ejemplo de nombre índice. Por último también hay un entero, este representará el tamaño del elemento que se quiere cargar, en este caso es un 4 por que se quieren cargar enteros de 32 bits que son 4 bytes. Todo esto se puede ver en este ejemplo:



Con la flecha en la derecha, están todas las *intrinsics* que se han usado para realizar el load. Antes del bucle se inicializan dos variables que se usarán para elegir que variables cargar en la memoria, en la variable ***indice*** se tienen los valores del 7 al 0, porque se quieren leer posiciones consecutivas de la memoria, la variable a servirá para ir recorriendo el vector incrementando los valores de índice, como se puede ver en la última línea del código.

Finalmente se utiliza la *intrinsic* de *gathering*, donde se usan como variables de entrada, la dirección de la que se quiere leer, ***va*** en este caso, la variable ***indice*** que ira indicando que elementos leer, y finalmente el 4.

La *intrinsic* para store también es diferente. Esta *intrinsic* dará la opción de guardar en la memoria un 0 en vez del elemento de la variable. Funciona con una máscara: si esta tiene el valor de una seguida de unos, guardará el elemento correspondiente, pero si tiene el valor de 0, guardará un 0. Aquí hay un ejemplo en código:



Estas son las *intrinsics* que se necesitan para usar el store de AVX2. Como se ha comentado se necesita una máscara. Se ha creado la máscara usando la operación lógica *xor*. En este caso se creara una máscara con dos tipos de valores: 0xfff…ff y 0x00…00. La máscara tendrá el primer tipo de valor en todos los elementos del vector menos en el tercero, donde tendrá el segundo tipo de valor.

Se usa la máscara para guardar los elementos consecutivos de la variable resultado. La *intrinsic* guardará los elementos en la memoria si en la posición correspondiente de la máscara hay una seguida de unos, sino, guardara un cero en esa posición, es este caso guardará todos los elementos menos el tercero.

2.3.3. Inicializar variables

Para inicializar variables, además de leer los datos de la memoria, se pueden elegir los valores para inicializar las variables. Estas *intrinsics* tienen 3 opciones principales: expansión escalar, inicializar el vector dando todos los valores y inicializar el vector con todo ceros. Ahora se verá un ejemplo de expansión escalar para ver cómo funciona:

Esta es la *intrinsic* utilizada en el ejemplo: v_b= _mm_set1_ps (b), donde b tiene un valor de 3,0.

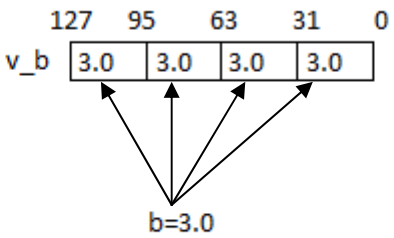


Figura 2.11: ejemplo inicialización

En la figura 2.11 hay un ejemplo de cómo inicializamos un vector con un solo número. En este y los demás casos, también se puede elegir el tipo de dato. Ahora se explicará la especificación de la operación en cada tecnología:

2.3.3.1 Inicializar variables en SSE

Así es como se especifica la operación en la guía que Intel ofrece:

```
__m128 _mm_set1_ps (float a)
__m128 _mm_setzero_ps (void)
__m128 _mm_set_ps (float e3, float e2, float e1, float e0)
```

Así se usan en el ejemplo:

```
data_t ejemplo_vec ( data_t *va,data_t *resultado ,float b,
int n ){
    int i;
    float suma;
    __m128 vsuma, comparar, lag, cond,a,v_b;
    vsuma=_mm_setzero_ps ();
    comparar=_mm_set_ps(20.0,20.0,20.0,20.0);
    v_b= _mm_set1_ps (b);
    for(i=0;i<n;i+=4){
        a= _mm_load_ps (va+i);
        lag= _mm_add_ps (a,v_b);
        cond = _mm_cmple_ps (lag,comparar);
        lag = _mm_and_ps (lag,cond);
        _mm_store_ps (&resultado[i], lag);
        vsuma= _mm_add_ps (lag,vsuma);
    }
    vsuma = _mm_hadd_ps (vsuma,vsuma);
    vsuma = _mm_hadd_ps (vsuma,vsuma);
    _mm_store_ss (&suma, vsuma);
    return suma;
}
```

Se explicarán los tipos de set de uno en uno:

Set1

Para realizar esta operación se necesita una variable de destino y algo con que llenar esa variable, en este caso la variable escalar ***b***. Este es el código equivalente que muestra de una forma escalar lo que hace la *intrinsic* vectorial:

```
v_b= _mm_set1_ps (b);
for(i=0,i<4,i++)
        v_b[i]=b;
```

En el código se ve como se le asigna a todas las posiciones del vector el valor ***b***.

Setzero

Para realizar esta operación solamente se necesitará una variable de destino. Esa es la variable que se inicializará con todo ceros. Este es el código equivalente que muestra de una forma escalar lo que hace la *intrinsic* vectorial:

```
vsuma=_mm_setzero_ps ();
for(i=0,i<4,i++)
        vsuma[i]=0;
```

Set

Con esta tecnología se usa este tipo de set, aunque no sea necesario es este caso, ni lo más adecuado.

Para realizar esta operación se necesitarán los valores de cada elemento con el que se desea inicializar el vector. Se inicializará cada posición del vector con uno de esos valores. Este es el código equivalente que muestra de una forma escalar lo que hace la *intrinsic* vectorial:

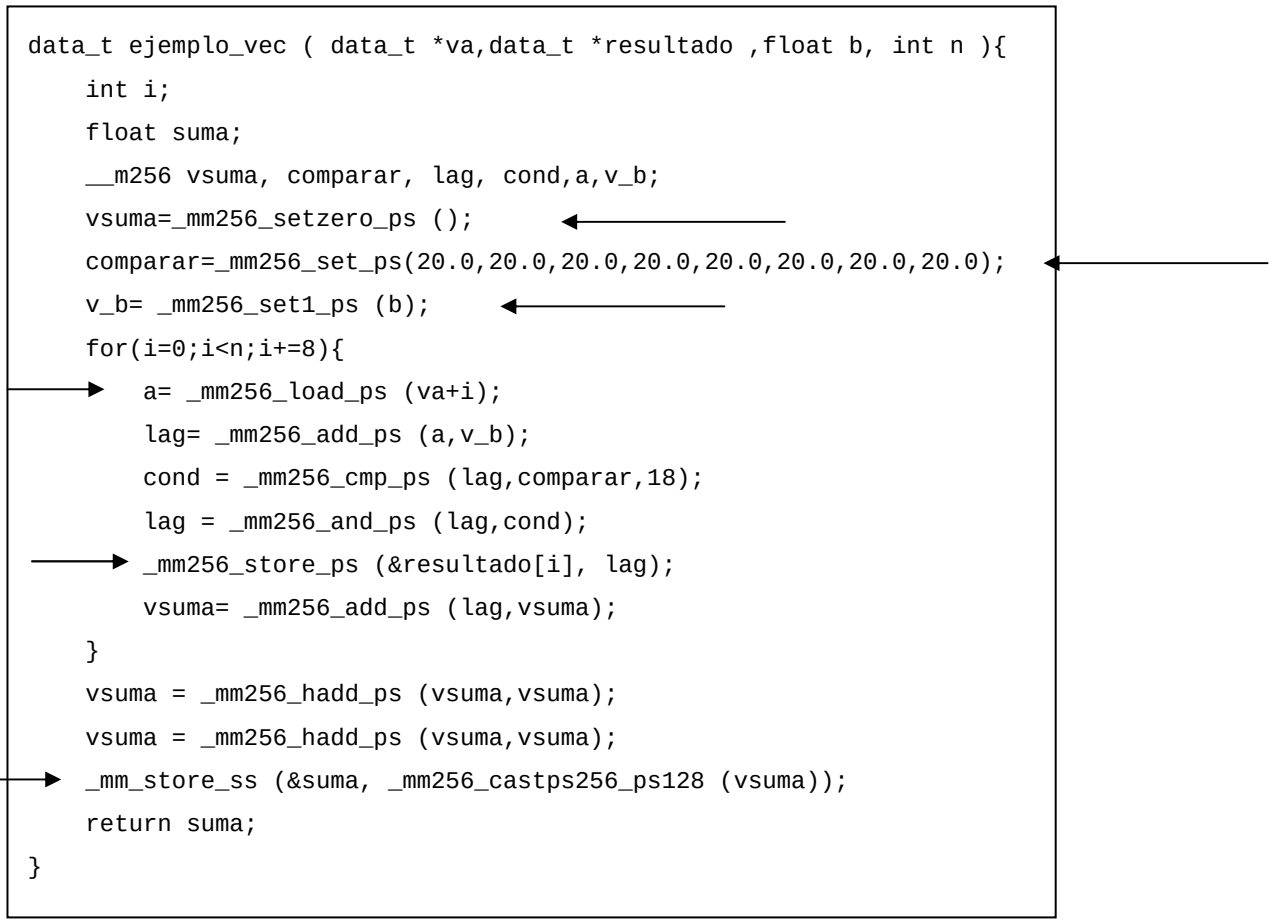
```
comparar=_mm_set_ps(20.0,20.0,20.0,20.0);
for(i=0,i<4,i++)
        comparar[i]=20.0;
```

2.3.3.2 Iniciar variables en AVX

Así es como se especifica la operación básica en la guía que Intel ofrece:

```
__m256 _mm256_set1_ps (float a)
__m256 _mm256_setzero_ps (void)
__m256 _mm256_set_ps (float e7, float e6, float e5, float e4, float e3, float e2, float e1,float e0)
```

Así se usan en el ejemplo:



```
data_t ejemplo_vec ( data_t *va,data_t *resultado ,float b, int n ){
    int i;
    float suma;
    __m256 vsuma, comparar, lag, cond,a,v_b;
    vsuma=_mm256_setzero_ps ();
    comparar=_mm256_set_ps(20.0,20.0,20.0,20.0,20.0,20.0,20.0,20.0);
    v_b= _mm256_set1_ps (b);
    for(i=0;i<n;i+=8){
        a= _mm256_load_ps (va+i);
        lag= _mm256_add_ps (a,v_b);
        cond = _mm256_cmp_ps (lag,comparar,18);
        lag = _mm256_and_ps (lag,cond);
        _mm256_store_ps (&resultado[i], lag);
        vsuma= _mm256_add_ps (lag,vsuma);
    }
    vsuma = _mm256_hadd_ps (vsuma,vsuma);
    vsuma = _mm256_hadd_ps (vsuma,vsuma);
    _mm_store_ss (&suma, _mm256_castps256_ps128 (vsuma));
    return suma;
}
```

Todas estas operaciones son igual que en SSE, solo cambia el tamaño del vector y por lo tanto el número de elementos.

Set1

Este es el código equivalente que nos muestra de una forma escalar lo que hace la *intrinsic* vectorial:

```
v_b= _mm256_set1_ps (b);
for(i=0,i<8,i++)
       v_b[i]=b;
```

Setzero

Este es el código equivalente que nos muestra de una forma escalar lo que hace la *intrinsic* vectorial:

```
vsuma=_mm256_setzero_ps ();
for(i=0,i<8,i++)
       vsuma[i]=0;
```

Set

Con esta tecnología también se usa este tipo de set, aunque no sea necesario ni lo más adecuado. Como todos los elementos son iguales seria más eficiente usar la *intrinsic* set1.

Este es el código equivalente que nos muestra de una forma escalar lo que hace la *intrinsic* vectorial:

```
comparar=_mm256_set_ps(20.0,20.0,20.0,20.0,20.0,20.0,20.0,20.0);
for(i=0,i<8,i++)
       comparar[i]=0;
```

2.3.3.3 Inicializar variables en AVX2

Para esta tecnología no hay ninguna *intrinsic* específica, pero podemos utilizar las *intrinsics* de AVX con las que se puede inicializar los vectores de 256 bits con cualquier tipo de dato.

2.3.4 Operaciones aritméticas

Estas son las operaciones que se pueden realizar en cada tecnología:

SSE

- Suma, resta, multiplicación, división, y cambiar el signo

AVX

- Suma, resta, multiplicación, y división

AVX2

- Suma, resta, multiplicación, y **falta división**

Para ver cómo funcionan se verá un ejemplo de una suma, las demás operaciones funcionan de forma similar.

Se sumarán dos vectores de 128 bits, compuestos por 4 números de coma flotante de simple precisión, aquí tenemos el ejemplo:

Esta es la *intrinsic* utilizada en el ejemplo: c = _mm_add_ps (a,b)

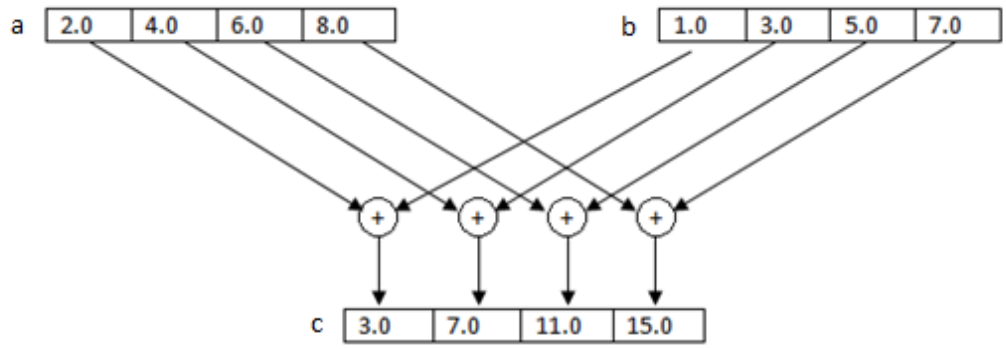


Figura 2.12: Ejemplo suma

En la figura 2.12 se sumarán uno a uno los elementos del vector de forma simultánea, con los que estén en la misma posición en el otro vector y se sumarán en el registro de la variable destino.

Por último se explica la especificación de una suma con cada tecnología (para el resto de operaciones las especificaciones son las mismas).

2.3.4.1 Suma en SSE

Así es como se especifica la operación en la guía que Intel nos ofrece:

```
__m128 _mm_add_ps (__m128 a, __m128 b)
```

Así se usan en el ejemplo:

```
data_t ejemplo_vec ( data_t *va,data_t *resultado ,float b,
int n )
{
    int i;
    float suma;
    __m128 vsuma, comparar, lag, cond,a,v_b;
    vsuma=_mm_setzero_ps ();
    comparar=_mm_set_ps(20.0,20.0,20.0,20.0);
    v_b= _mm_set1_ps (b);
    for(i=0;i<n;i+=4){
        a= _mm_load_ps (va+i);
        lag= _mm_add_ps (a,v_b);
        cond = _mm_cmple_ps (lag,comparar);
        lag = _mm_and_ps (lag,cond);
        _mm_store_ps (&resultado[i], lag);
        vsuma= _mm_add_ps (lag,vsuma);
    }
    vsuma = _mm_hadd_ps (vsuma,vsuma);
    vsuma = _mm_hadd_ps (vsuma,vsuma);
    _mm_store_ss (&suma, vsuma);
    return suma;
}
```

Como se ve la *intrinsic* necesita dos vectores de 128 bits y devolverá la respuesta en otro vector. Este es el código equivalente que muestra de una forma escalar lo que hace la *intrinsic* vectorial:

```
lag= _mm_add_ps (a,v_b);
for(i=0;i<4;i++)
        lag[i]=a[i]+v_b[i];
```

El bucle trabaja con cuatro elementos del vector. Aunque en el código el vector de salida sea distinto, a nivel de ensamblador el de salida sería el mismo que uno de entrada. Es decir, tiene dos operandos. Al ser vectorial este se puede realizar de forma simultánea o paralela.

Se estudiará el código de nivel ensamblador, para comprobar que si se pone una tercera variable como variable de salida, como en el ejemplo, al ser de dos operandos usará un comando para sumar las variable y luego otro para guardarlo en la variable que se quiera.

```
addps       xmm1, xmm3
movaps      xmm0, xmm1
```

Figura 2.13: Suma tres variables con dos operandos

En la figura 2.13 hay dos comandos, con el primero se suma usando dos registros y con el segundo se pasa el resultado de la suma a otro registro.

2.3.4.2 Suma en AVX

Así es como se especifica la operación en la guía que Intel ofrece:

```
__m256 _mm256_add_ps (__m256 a, __m256 b)
```

Así se usan en el ejemplo:

```
data_t ejemplo_vec ( data_t *va,data_t *resultado ,float b, int n ){
    int i;
    float suma;
    __m256 vsuma, comparar, lag, cond,a,v_b;
→   vsuma=_mm256_setzero_ps ();
→   comparar=_mm256_set_ps(20.0,20.0,20.0,20.0,20.0,20.0,20.0,20.0);
→   v_b= _mm256_set1_ps (b);
    for(i=0;i<n;i+=8){
→       a= _mm256_load_ps (va+i);
        lag= _mm256_add_ps (a,v_b);        ←————————
        cond = _mm256_cmp_ps (lag,comparar,18);
        lag = _mm256_and_ps (lag,cond);
→       _mm256_store_ps (&resultado[i], lag);
        vsuma= _mm256_add_ps (lag,vsuma);        ←————————
    }
    vsuma = _mm256_hadd_ps (vsuma,vsuma);
    vsuma = _mm256_hadd_ps (vsuma,vsuma);
→   _mm_store_ss (&suma, _mm256_castps256_ps128 (vsuma));
    return suma;
}
```

En la *intrinsic* se ve como se necesitan dos vectores de 256 bits y como se devuelve la respuesta en otro vector de 256 bits. Este es el código equivalente que muestra de una forma escalar lo que hace la *intrinsic* vectorial:

```
lag= _mm256_add_ps (a,v_b);
for(i=0;i<8;i++)
        lag[i]=a[i]+v_b[i];
```

El bucle trabaja con ocho elementos del vector, los guarda en el vector de salida, que es uno distinto a los dos de entrada. Es decir tiene tres operandos. Al ser vectorial este se puede realizar de forma simultánea o paralela. Ahora veremos como hace la operacion en nivel ensamblador:

```
vaddps      ymm1, ymm3, YMMWORD PTR [ecx+eax*4]
```

Figura 2.14: Suma tres operandos

2.3.4.3 Suma en AVX2

Así es como se especifica la operación en la guía que Intel ofrece:

```
__m256i_mm256_add_epi32(__m256i a,__m256i b)
```

Y así como se especifica la operación para usarla en el algoritmo:

```
lag=_mm256_add_epi32(a, b)
```

En la *intrinsic* se ve como se necesitan dos vectores de 256 bits, aunque en este caso tienen el sufijo i, por lo tanto, los vectores tienen que estar compuestos por enteros. También se ve como devuelve la respuesta en otro vector de 256 bits. Hay que tener en cuenta que AVX2 no deja hacer divisiones.

Este es el código equivalente que muestra de una forma escalar lo que hace la *intrinsic* vectorial:

```
lag=_mm256_add_epi32(a, b)
for(i=0;i<8;i++)
        lag[i]=a[i]+b[i];
```

Se puede apreciar que el código es el mismo que en AVX, la única diferencia es el tipo de dato.

2.3.5 Reducciones

Estas funciones dan la opción de sumar o restar los elementos del vector entre sí. La opción más usada es la de la suma, porque permite sumar todos los elementos del vector y guardarlo en la primera posición. Las reducciones funcionan así:

Esta es la *intrinsic* que se ha utilizado para el ejemplo:

suma = _mm_hadd_ps (resultado,resultado)

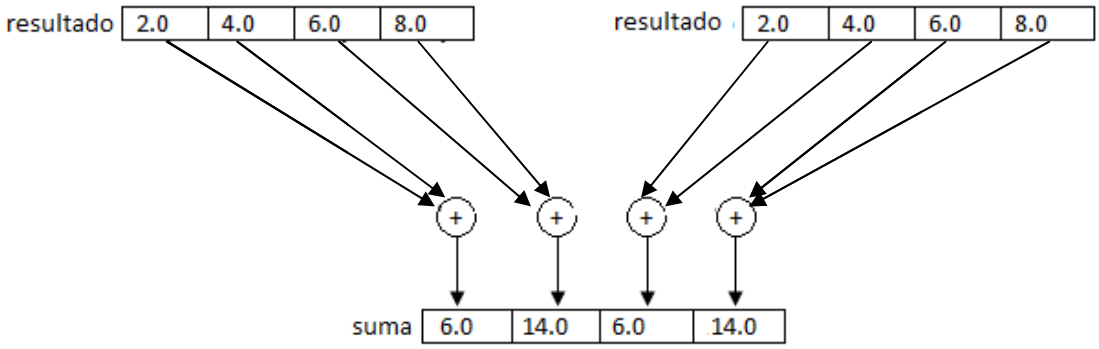


Figura 2.15: Ejemplo reducción

En la figura 2.15 se suman el primer y el segundo elemento del vector por un lado y el tercero y cuarto por otro. Aunque haya dos entradas no son necesarias por lo que la primera entrada solamente se usa para “rellenar”. Usando una vez solo la operación se puede ver que no se consigue lo esperado al no tener la suma de los cuatro elementos en ningún lado. Para conseguir la suma completa hace falta realizar otra vez la operación. Así se consigue la suma de todos los elementos en la primera posición del vector.

2.3.5.1 Reducciones en SSE

Así es como se especifica la operación en la guía que Intel nos ofrece:

```
__m128 _mm_hadd_ps (__m128 a, __m128 b)
```

Así se usan en el ejemplo:

```
data_t ejemplo_vec ( data_t *va,data_t *resultado ,float b, int n
){
    int i;
    float suma;
    __m128 vsuma, comparar, lag, cond,a,v_b;
    vsuma=_mm_setzero_ps ();
    comparar=_mm_set_ps(20.0,20.0,20.0,20.0);
    v_b= _mm_set1_ps (b);
    for(i=0;i<n;i+=4){
        a= _mm_load_ps (va+i);
        lag= _mm_add_ps (a,v_b);
        cond = _mm_cmple_ps (lag,comparar);
        lag = _mm_and_ps (lag,cond);
        _mm_store_ps (&resultado[i], lag);
        vsuma= _mm_add_ps (lag,vsuma);
    }
    vsuma = _mm_hadd_ps (vsuma,vsuma);
    vsuma = _mm_hadd_ps (vsuma,vsuma);
    _mm_store_ss (&suma, vsuma);
    return suma;
}
```

Tiene dos vectores de entrada que se sumaran como muestra la figura 2.16, dando como respuesta otro vector del mismo tipo. Este sería el código equivalente de forma escalar para mostrar lo que hace la *intrinsic* vectorial:

```
vsuma = _mm_hadd_ps (vsuma,vsuma);
vsuma[0]= vsuma[0]+vsuma[1]
vsuma[1]= vsuma[2]+vsuma[3]
vsuma[2]= vsuma[0]+vsuma[1]
vsuma[3]= vsuma[2]+vsuma[3]
```

2.3.5.2 Reducciones en AVX

Así es como se especifica la operación en la guía que Intel nos ofrece:

```
__m256 _mm256_hadd_ps (__m256 a, __m256 b)
```

Así se usan en el ejemplo:

```
data_t ejemplo_vec ( data_t *va,data_t *resultado ,float b, int n ){
    int i;
    float suma;
    __m256 vsuma, comparar, lag, cond,a,v_b;
    vsuma=_mm256_setzero_ps ();
    comparar=_mm256_set_ps(20.0,20.0,20.0,20.0,20.0,20.0,20.0,20.0);
    v_b= _mm256_set1_ps (b);
    for(i=0;i<n;i+=8){
        a= _mm256_load_ps (va+i);
        lag= _mm256_add_ps (a,v_b);
        cond = _mm256_cmp_ps (lag,comparar,18);
        lag = _mm256_and_ps (lag,cond);
        _mm256_store_ps (&resultado[i], lag);
        vsuma= _mm256_add_ps (lag,vsuma);
    }
    vsuma = _mm256_hadd_ps (vsuma,vsuma);
    vsuma = _mm256_hadd_ps (vsuma,vsuma);
    vsuma = _mm256_hadd_ps (vsuma,vsuma);
    _mm_store_ss (&suma, _mm256_castps256_ps128 (vsuma));
    return suma;
}
```

Igual que en SSE se sumarán de dos en dos los elementos del vector, pero en este caso al tener vectores de ocho elementos, se tendrá que realizar la operación 3 veces para conseguir que la suma de todos elementos este en la primera posición. Este sería el código equivalente de forma escalar para mostrar lo que hace la *intrinsic* vectorial:

```
vsuma = _mm256_hadd_ps (vsuma,vsuma);
vsuma[0]= vsuma[0]+ vsuma[1]
vsuma[1]= vsuma[2]+ vsuma[3]
vsuma[2]= vsuma[4]+ vsuma[5]
vsuma[3]= vsuma[6]+ vsuma[7]
vsuma[4]= vsuma[0]+ vsuma[1]
vsuma[5]= vsuma[2]+ vsuma[3]
vsuma[6]= vsuma[4]+ vsuma[5]
vsuma[7]= vsuma[6]+ vsuma[7]
```

2.3.5.3 Reducciones en AVX2

Así es como se especifica la operación en la guía que Intel ofrece:

```
__m256i _mm256_hadd_epi32 (__m256i a, __m256i b)
```

Y así como la utilizaríamos nosotros en el ejemplo:

```
vsuma = _mm256_hadd_epi32 (vsuma,vsuma)
```

Esta *intrinsic* es igual que la de AVX, solamente cambiará el tipo de dato del vector, de números de coma flotante a números enteros. Este sería el código equivalente de forma escalar para mostrar lo que hace la *intrinsic* vectorial:

```
vsuma = _mm256_hadd_epi32 (vsuma,vsuma)
vsuma[0]= vsuma [0]+ vsuma [1]
vsuma[1]= vsuma [2]+ vsuma [3]
vsuma[2]= vsuma [4]+ vsuma [5]
vsuma[3]= vsuma [6]+ vsuma [7]
vsuma[4]= vsuma [0]+ vsuma [1]
vsuma[5]= vsuma [2]+ vsuma [3]
vsuma[6]= vsuma [4]+ vsuma [5]
vsuma[7]= vsuma [6]+ vsuma [7]
```

2.3.6 Condicionales

Con estas funciones se compararan los elementos de dos vectores. Estas son las operaciones disponibles en cada tecnología:

SSE y AVX

- Igual, no igual
- Más grande que, más grande o igual que
- No más grande que, no más grande o igual que
- Más pequeño que, más pequeño o igual que
- No más pequeño que, no más pequeño o igual que

Solo en SSE

- Mirar si no en un número

AVX2

- Igual
- Más grande que

En AVX además de estas opciones se pueden elegir otras dos opciones dentro de la comparación. La primera se llama *signaling*, si escogemos esta opción se tendrán en cuenta los signos en la comparación en cambio si se elige *non-signaling*, se compararan los valores absolutos.

La segunda opción es *ordered,* usando esta opción se comprueba antes de hacer la comparación si los dos elementos a comparar son números, si no lo son devolverá false, sino hará la comparación. Con la opción *non-ordered* en cambio uno de los dos elementos debe no ser un número sino la comparación tendrá como resultado false.

La forma de trabajar con las *intrinsics* es diferente en AVX. En SSE y AVX2 hay una *intrinsic* diferente por cada comparación que se quiera hacer, en AVX en cambio, solamente tenemos una *intrinsic*. Se elegirá que comparación queremos realizar mediante un número entero.

Ahora tenemos un ejemplo de cómo funciona este tipo de operaciones:

Esta es la *intrinsic* utilizada en el ejemplo: cond = _mm_cmple_ps (lag, comparar)

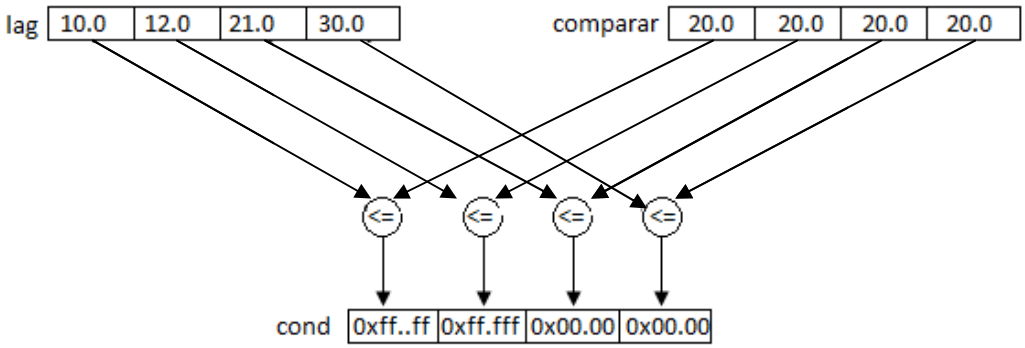


Figura 2.16: Ejemplo condición

En la figura 2.16 se ve como este tipo de operaciones tiene una característica diferente a cuando se hace de modo escalar. En la salida cada elemento está compuesto de ceros, cuando la condición no se cumple, o seguidas de unos, cuando la condición se cumple.

2.3.6.1 Condicionales en SSE

Así es como se especifica la operación en la guía que Intel ofrece:

```
__m128 _mm_cmple_ps (__m128 a, __m128 b)
```

Así se usan en el ejemplo:

```
data_t ejemplo_vec ( data_t *va,data_t *resultado ,float b, int n ){
    int i;
    float suma;
    __m128 vsuma, comparar, lag, cond,a,v_b;
    vsuma=_mm_setzero_ps ();
    comparar=_mm_set_ps(20.0,20.0,20.0,20.0);
    v_b= _mm_set1_ps (b);
    for(i=0;i<n;i+=4){
        a= _mm_load_ps (va+i);
        lag= _mm_add_ps (a,v_b);
        cond = _mm_cmple_ps (lag,comparar);
        lag = _mm_and_ps (lag,cond);
        _mm_store_ps (&resultado[i], lag);
        vsuma= _mm_add_ps (lag,vsuma);
    }
    vsuma = _mm_hadd_ps (vsuma,vsuma);
    vsuma = _mm_hadd_ps (vsuma,vsuma);
    _mm_store_ss (&suma, vsuma);
    return suma;
}
```

Necesita dos vectores de entrada y tiene uno de salida con la característica diferente que hemos comentado. Este sería el código equivalente de forma escalar para mostrar lo que hace la *intrinsic* vectorial:

```
cond = _mm_cmple_ps (lag,comparar);
comparer=20.0;
for(i=0,i<4,i++){
if(lag[i]<=comaprar)
       cond[i]=0xfff..fff
else
       cond[i]=0x00..000
}
```

2.3.6.2 Condicionales en AVX

Así es como se especifica la operación en la guía que Intel ofrece:

```
__m256 _mm256_cmp_ps (__m256 a, __m256 b, const int imm)
```

Así se usan en el ejemplo:

```
data_t ejemplo_vec ( data_t *va,data_t *resultado ,float b, int n ){
    int i;
    float suma;
    __m256 vsuma, comparar, lag, cond,a,v_b;
    vsuma=_mm256_setzero_ps ();
    comparar=_mm256_set_ps(20.0,20.0,20.0,20.0,20.0,20.0,20.0,20.0);
    v_b= _mm256_set1_ps (b);
    for(i=0;i<n;i+=8){
      a= _mm256_load_ps (va+i);
      lag= _mm256_add_ps (a,v_b);
        cond = _mm256_cmp_ps (lag,comparar,18);
        lag = _mm256_and_ps (lag,cond);
      _mm256_store_ps (&resultado[i], lag);
      vsuma= _mm256_add_ps (lag,vsuma);
    }
    vsuma = _mm256_hadd_ps (vsuma,vsuma);
    vsuma = _mm256_hadd_ps (vsuma,vsuma);
    vsuma = _mm256_hadd_ps (vsuma,vsuma);
    _mm_store_ss (&suma, _mm256_castps256_ps128 (vsuma));
    return suma;
}
```

En este caso hay una variable as de entrada, un entero (18). Este entero sirve para elegir la operación que se llevará a cabo, en este caso, más pequeño o igual. Se puede encontrar la tabla con las equivalencias en la guía de Intel. Además tiene dos vectores de entrada y uno de salida. Este sería el código equivalente de forma escalar para mostrar lo que hace la *intrinsic* vectorial:

```
cond = _mm256_cmp_ps (lag,comparar,18);
comparar=20.0;
for(i=0,i<8,i++){
if(lag[i]<=comparar)
       cond[i]=0xfff..fff
else
       cond[i]=0x00..000
}
```

En el código se puede apreciar como es el mismo que en SSE. Solo aumenta el número de iteraciones, por que el vector es más grande.

2.3.6.3 Condicionales en AVX2

Al no tener esta comparación en esta tecnología utilizaremos la comparación de más grande que del ejemplo.

Así es como se especifica la operación en la guía que Intel ofrece:

```
__m256i _mm256_cmpgt_epi32 (__m256i a, __m256i b)
```

Y así como la se utilizaría en el ejemplo:

```
cond = _mm256_cmpgt_epi32  (lag, comparar)
```

Al elegir la comparación más grande que, aunque no hay ningún problema, hay que tener en cuenta que, además de esta operación, AVX2 solo dispone la comparación de igualdad. Por lo que solo se podrán realizar directamente esas comparaciones en vectores de 256 bits con números enteros, sino se deberían realizar más operaciones.

Este sería el código equivalente de forma escalar para mostrar lo que hace la *intrinsic* vectorial:

```
cond =_mm256_cmpgt_epi32 (lag,comparar)
comparar=20.0
for(i=0,i<8,i++){
if(lag[i]>comparar)
       cond[i]=0xfff..fff
else
       cond[i]=0x00..000
}
```

Como en los casos anteriores este es igual al código de AVX.

2.3.7 Operaciones lógicas

Con estas operaciones se realizan las operaciones lógicas AND, OR, ANDNOT y XOR. Estas operaciones se realizaran a nivel de bits. Ahora se verá un ejemplo de la operación AND para ver cómo funciona, aunque las demás operaciones funcionan igual. Este es el ejemplo utilizado: lag = _mm_and_ps (lag, cond)

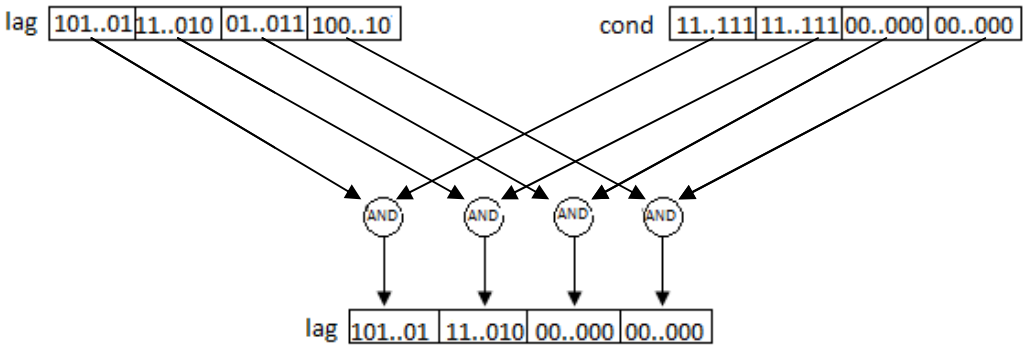


Figura 2.17: Ejemplo AND

En la figura 2.17 se realiza la operación entre los elementos a nivel de bit. En este ejemplo se ve la utilidad que se ha comentado antes; si se supone que la en la variable ***cond*** esta la salida de la operación condicional, y se quiere condicionar la variable ***lag***. Se ve como en la respuesta los elementos que no cumplían con la condición se convierten en 0 en la salida.

Por último se especificará el uso de las operaciones en todas las tecnologías, teniendo que cuenta que todas se utilizan igual solo explicaremos la operación AND.

2.3.7.1 AND en SSE

Así es como se especifica la operación en la guía que Intel nos ofrece:

```
__m128 _mm_and_ps (__m128 a, __m128 b)
```

Así se usan en el ejemplo:

```
data_t ejemplo_vec ( data_t *va,data_t *resultado ,float b, int n ){
    int i;
    float suma;
    __m128 vsuma, comparar, lag, cond,a,v_b;
    vsuma=_mm_setzero_ps ();
    comparar=_mm_set_ps(20.0,20.0,20.0,20.0);
    v_b= _mm_set1_ps (b);
    for(i=0;i<n;i+=4){
        a= _mm_load_ps (va+i);
        lag= _mm_add_ps (a,v_b);
        cond = _mm_cmple_ps (lag,comparar);
        lag = _mm_and_ps (lag,cond);
        _mm_store_ps (&resultado[i], lag);
        vsuma= _mm_add_ps (lag,vsuma);
    }
    vsuma = _mm_hadd_ps (vsuma,vsuma);
    vsuma = _mm_hadd_ps (vsuma,vsuma);
    _mm_store_ss (&suma, vsuma);
    return suma;
}
```

Hay 2 variables de entrada y una de salida, esta última se consigue realizando la operación AND entre las dos de entrada, por lo tanto, es una operación muy parecida a las aritméticas.

Este sería el código equivalente de forma escalar para mostrar lo que hace la *intrinsic* vectorial:

```
lag = _mm_and_ps (lag,cond);
for(i=0,i<4,i++){
       lag[i]=lag[i] && cond[i]
}
```

2.3.7.2 AND en AVX

Así es como se especifica la operación en la guía que Intel ofrece:

```
__m256 _mm256_and_ps (__m256 a, __m256 b)
```

Así se usan en el ejemplo:

```
data_t ejemplo_vec ( data_t *va,data_t *resultado ,float b, int n ){
    int i;
    float suma;
    __m256 vsuma, comparar, lag, cond,a,v_b;
    vsuma=_mm256_setzero_ps ();
    comparar=_mm256_set_ps(20.0,20.0,20.0,20.0,20.0,20.0,20.0,20.0);
    v_b= _mm256_set1_ps (b);
    for(i=0;i<n;i+=8){
        a= _mm256_load_ps (va+i);
        lag= _mm256_add_ps (a,v_b);
        cond = _mm256_cmp_ps (lag,comparar,18);
        lag = _mm256_and_ps (lag,cond);
        _mm256_store_ps (&resultado[i], lag);
        vsuma= _mm256_add_ps (lag,vsuma);
    }
    vsuma = _mm256_hadd_ps (vsuma,vsuma);
    vsuma = _mm256_hadd_ps (vsuma,vsuma);
    vsuma = _mm256_hadd_ps (vsuma,vsuma);
    _mm_store_ss (&suma, _mm256_castps256_ps128 (vsuma));
    return suma;
}
```

El código muestra como tiene dos vectores de entrada de 256 bits y otro de salida de 256 bits.

Este sería el código equivalente de forma escalar para mostrar lo que hace la *intrinsic* vectorial:

```
lag = _mm256_and_ps (lag,cond);
for(i=0,i<8,i++){
      lag[i]=lag[i] && cond[i]
}
```

En el código se ve que la única diferencia con SSE vuelve a ser el número de iteraciones, esto es, porque los vectores usados son el doble de largos.

2.3.7.3 AND en AVX2

Esta tecnología también tiene operaciones lógicas, pero no se han podido utilizarlas al trabajar con números enteros de 32 bits.

Esta es la especificación que Intel da en su guía:

```
__m256i _mm256_and_si256 (__m256i a, __m256i b)
```

Esta *intrinsic* trabaja con vectores compuestos por enteros de 256 bits que no se utilizan. Por lo tanto, si se quieren utilizar las operaciones lógicas con números de 32 bits se debe convertir el vector a elementos de números de coma flotante y utilizar las operaciones de AVX que se han comentado en punto anterior.

2.4 Cálculo de los resultados

2.4.1 Recursos utilizados

Estos son los elementos que se han utilizado para sacar adelante el trabajo. Se dividirán los elementos utilizados en hardware y software.

El único elemento de hardware especial que se ha utilizado han sido el procesador y la memoria. En este caso se han utilizado dos procesadores para hacer las pruebas, un Intel IT-2520M 2,5Ghz (Procesador 1) y un Intel Core i5-3570K 3.4Ghz (Procesador 2). Los dos tienen una memoria de 8GB. Solo el primero de ellos tiene arquitectura Ivy Bridge, donde deberían funcionar todas las tecnologías, pero como ya hemos comentado que no es así, por lo tanto podremos usar los dos procesadores para hacer todas las pruebas.

Estos han sido los elementos de software que se han utilizado. Las pruebas se han realizado sobre Windows 7. Se ha usado el entorno de programación CodeBlocks para realizar el código en C. Y por último, se ha usado el compilador gcc para compilar los programas. Para usar gcc en Windows se ha tenido que descargar, e instalar el programa mingw.

Se han ejecutado cada programa de tres formas diferentes, usando las opciones de gcc, la compilación escalar con diferentes niveles de optimización, la compilación vectorial y compilar las *intrinsics*. Se han utilizado estas opciones:

Compilación escalar optimizada: -o0, -o1, -o2 y –o3

Compilación vectorial y *intrinsics*: SSE-march=core2, AVX -march=corei7-avx

2.4.2 Resultados ejemplo

Como ya se ha comentado no hemos podido ejecutar los programas realizados para las *intrinsics* AVX2. Aun así, nos hemos dado cuenta que usando la compilación de AVX2 sobre los programas de AVX, crea código que se puede ejecutar en algunos casos, aun así no se tendrán en cuenta estos resultados.

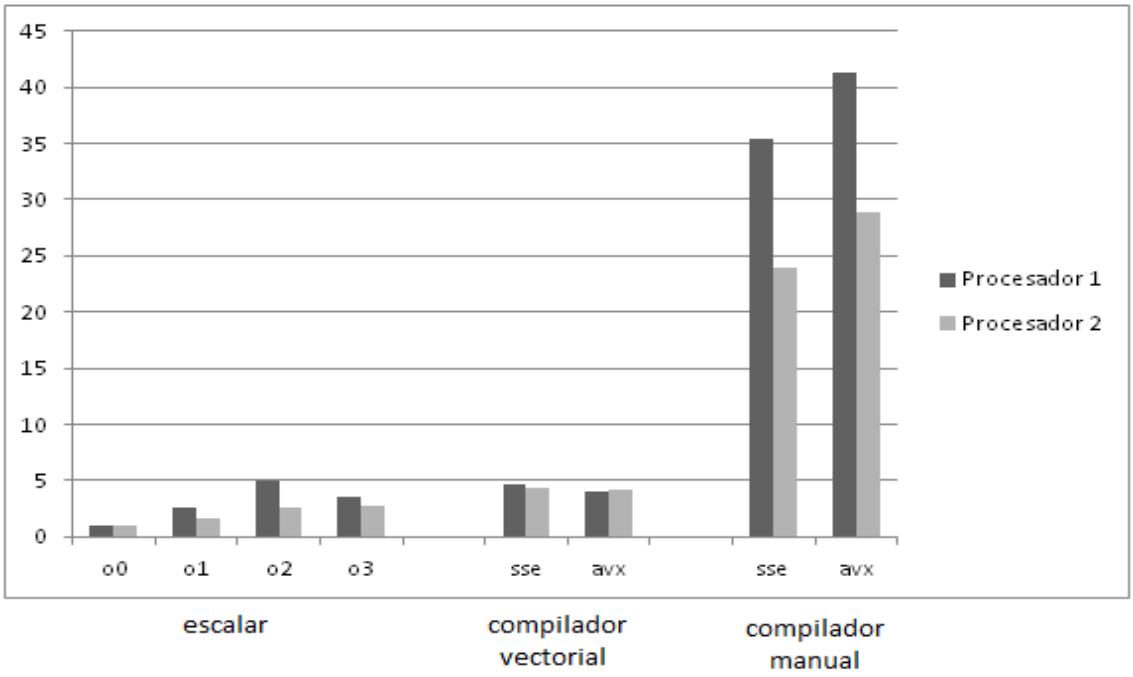


Gráfico 2.1: Resultados ejemplo

En el gráfico 2.1 están los *speed-up* conseguidos con todas las opciones y con los dos procesadores. Para calcular la duración de las ejecuciones se han utilizados los *ticks* del procesador.

Se ha utilizado este método en vez de usar el tiempo, porque algunas ejecuciones de programas son demasiado cortas como para medirlas con el tiempo. Con los *ticks* en cambio no hay ningún problema.

En el grafico 2.1 lo que se ve es el potencial de mejora de cada procesador, en ningún caso se quieren comparar los resultados de los dos procesadores. Aunque si se intentará explicar el porqué de las diferencias en los *speed-up* de los dos procesadores.

En este caso se puede ver como solamente usando las diferentes opciones de optimización se ha podido conseguir un *speed-up* de 3, más o menos, llegando en el caso del primer procesador a un *speed-up* de 5 con la opción de optimización -o2.

Con el código escalar, pero compilación vectorial se obtiene un *speed-up* un poco mejor que con las opciones anteriores estando siempre cerca del 5.

Estos *speed-up* se consiguen porque el compilador al optimizar elimina el *if*. En lugar de eso realiza una instrucción de comparación y luego otra de AND, como hacemos nosotros para vectorizar el *if*. Este *if* es de estructura simple: *if then*.

Finalmente están los *speed-up* conseguidos por el código vectorial. Los *speed-up* son mayores que lo esperado solamente por usar computación vectorial, que son 4 en el caso de SSE, y 8 en el caso de AVX. Aun así, se obtiene unos *speed-up* muy grandes, y como es normal en el caso de AVX en mayor que en el de SSE.

Como se ha comentado en este programa al vectorizarlo fusionábamos dos bucles, ese es el por qué de obtener un tan buen *speed-up*. Esto sucede porque al fusionar no se deben realizar todas las llamadas a la memoria que se realizan en el segundo bucle del código escalar.

En casi todas las opciones se ve que el primer procesador obtiene un mejor *speed-up*.

En el siguiente capítulo se presentaran diferentes algoritmos. Primero se explicará el funcionamiento de cada algoritmo. Se mostrará el código escalar, y los códigos vectoriales con SSE y AVX. También se mostrará los resultados como en este caso.

Capítulo 3

3 Algunos resultados

Estos son los algoritmos utilizados para probar los tipos de *intrinsics* que se han presentado en el capítulo anterior.

3.1 Caso Div

Este algoritmo es el más simple, hay un vector que dividiremos por el valor de la posición del elemento. Este es el código escalar:

```
void Vector_plus_i ( data_t *va, int n )
{
	int i;
	for (i = 0; i < n; i++)
			va[i]=va[i]/(float)(i+1);
}
```

En el código se ve como en escalar solo se necesita un *for* donde se recorre el vector donde se hace la división con el índice del *for* y se guarda el resultado en la variable de entrada. El código vectorial del algoritmo será muy parecido al de escalar. Se presentan dos versiones vectoriales:

```
void Vector_plus_i_vec_1 ( data_t *va, int n ){
       int i;
       __m128 v_indice, x_va;
       for (i = 0; i < n; i+=4)
       {
         x_va = _mm_load_ps(va+i);
         v_indice = _mm_set_ps(i+4,i+3,i+2,i+1);
         x_va = _mm_div_ps(x_va,v_indice);
         _mm_store_ps(va+i,x_va);
       }
}
 void Vector_plus_i_vec_2 ( data_t *va, int n )
{
       int i;
       __m128 v_indice, x_va, v_incr;

       v_indice = _mm_set_ps(4,3,2,1);
       v_incr   = _mm_set1_ps(4);
       for (i = 0; i < n/4; i++)
       {
         x_va = _mm_load_ps(va+i*4);
         x_va = _mm_div_ps(x_va,v_indice);
         _mm_store_ps(va+i*4,x_va);
         v_indice = _mm_add_ps(v_indice,v_incr);
       }
}
```

La principal diferencia es que hay que usar *intrinsics* específicas para leer y escribir en la memoria. La otra diferencia es la cantidad que se suma en el índice del *for*, al ir haciendo las divisiones de 4 en cuatro solo se necesita que el *for* haga una cuarta parte de las iteraciones.

Las versiones se diferencian en la forma de calcular el valor de i, que en este caso al ser las ir las iteraciones de cuatro en cuatro no se puede coger su valor directamente. En la primera, diferenciada con flechas en la izquierda, se rellena en cada iteración un vector con los valores de i necesarios, y en la segunda, con flechas en la derecha, se inicializara un vector con los cuatro primeros valores y irá sumándole 4 a cada elemento para conseguir los valores de i.

Este es el código de nivel ensamblador de la carga del índice:

```
mov   DWORD PTR [esp+28], ebx
fild  DWORD PTR [esp+28]
fstp  DWORD PTR [esp+12]
lea   ebx, [edx+2]
mov   DWORD PTR [esp+28], ebx
fild  DWORD PTR [esp+28]
fstp  DWORD PTR [esp+16]
add   edx, 3
mov   DWORD PTR [esp+28], edx
fild  DWORD PTR [esp+28]
fstp  DWORD PTR [esp+20]
mov   DWORD PTR [esp+28], eax
fild  DWORD PTR [esp+28]
fstp  DWORD PTR [esp+24]
```

Figura 3.1: cargar diferentes elementos de i

En la figura 3.1 se cargan los elementos de uno en uno. Por lo tanto en esa parte, trabaja de forma escalar. En la figura 3.2 se ve como al final lo guarda en un registro xmm, de uno en uno y mediante varios comandos se termina guardando en un solo registro, el xmm1. Finalmente se realiza la división:

```
movss xmm1, DWORD PTR [esp+20]
movss xmm3, DWORD PTR [esp+24]
unpcklps    xmm1, xmm3
movaps      xmm2, xmm1
movss xmm4, DWORD PTR [esp+12]
movss xmm3, DWORD PTR [esp+16]
unpcklps    xmm4, xmm3
movaps      xmm1, xmm4
movlhps     xmm1, xmm2
divps       xmm0, xmm1
```

Figura 3.2: guardar los elementos en el registro -

En cambio en el segundo ejemplo se ejecutan todas las operaciones de forma vectorial, por lo tanto, el segundo método será mucho más rápido.

```
movaps      xmm0, XMMWORD PTR [eax]
divps       xmm0, xmm1
movaps      XMMWORD PTR [eax], xmm0
addps       xmm1, xmm2
```

Figura 3.3: Bucle del segundo método

En la figura 3.3 se hace la división con el segundo comando, y el incremento de i con el último comando.

Las dos versiones también se diferencian en la forma de hacer el bucle. La primera de ellas irá sumando i de cuatro en cuatro, realizando n iteraciones. La segunda en cambio irá sumando i de uno en uno, pero solamente se realizaran n/4 iteraciones. Como se puede ver, los dos modos hacen la misma cantidad de iteraciones solamente cambia el valor con el que llamaremos a la memoria.

Ahora tenemos la versión del programa para AVX:

```
void Vector_plus_i_vec_1 ( data_t *va, int n ){
	int i;
	__m256 v_indice, x_va;
	for (i = 0; i < n; i+=8){
		x_va = _mm256_load_ps(va+i);
		v_indice = _mm256_set_ps(i+8,i+7,i+6,i+5,i+4,i+3,i+2,i+1);
		x_va = _mm256_div_ps(x_va,v_indice);
		_mm256_store_ps(va+i,x_va);
	}
}
 void Vector_plus_i_vec_2 ( data_t *va, int n ){
	int i;
	__m256 v_indice, x_va, v_incr;
	v_indice = _mm256_set_ps(8,7,6,5,4,3,2,1);
	v_incr   = _mm256_set1_ps(8);
	for (i = 0; i < n/8; i++){
		x_va = _mm256_load_ps(va+i*8);
		x_va = _mm256_div_ps(x_va,v_indice);
		 _mm256_store_ps(va+i*8,x_va);
		 v_indice = _mm256_add_ps(v_indice,v_incr);
	}
}
```

Con AVX también hay dos versiones (que se distinguen como en el ejemplo anterior) donde se trabaja de forma diferente con la variable i. Todas las diferencias que hay entre los dos códigos son por el uso de vectores más grandes: en el bucle se hacen n/8 en vez de n/4 iteraciones, y al inicializar la variable ***v_indice*** se usan ocho valores en vez de 4.

3.1.1 Resultados

Este es el gráfico con los resultados:

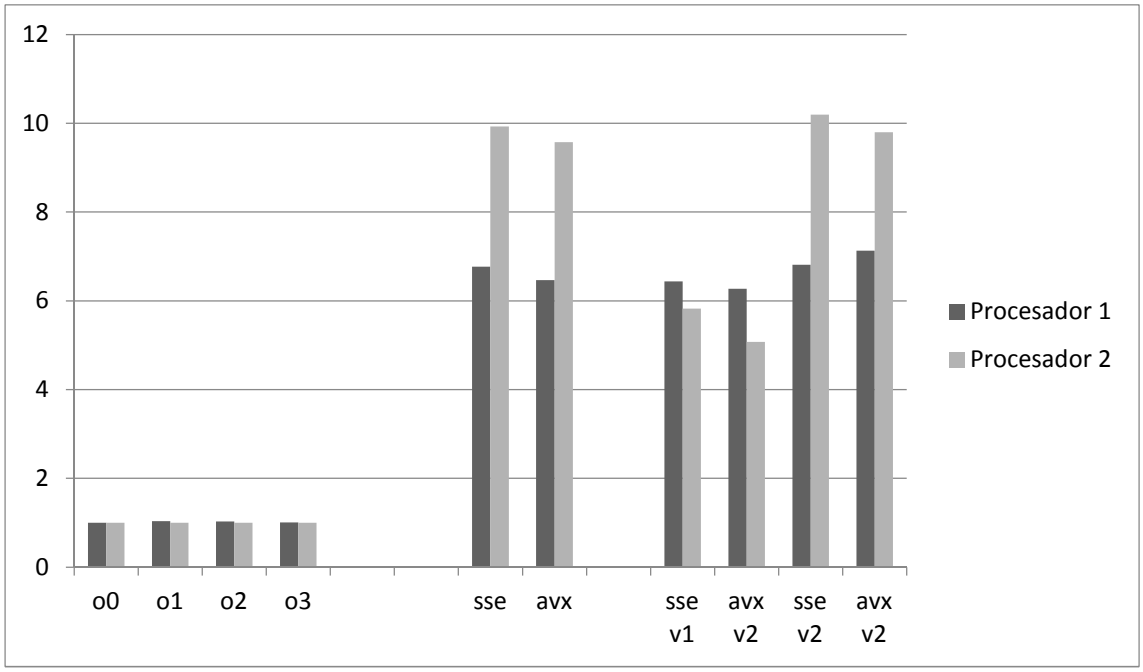


Gráfico 3.1: Resultados del caso Div

Se puede ver como el compilador, aun con optimizaciones no se consigue que el *speed-up* aumente siendo en todos los casos muy cercano a uno.

Esto sucede porque hay una limitación en el acceso de la memoria, aunque la latencia de la división sea bastante grande. Esto nos impide conseguir ninguna mejora de rendimiento.

Aunque el *speed-up* se mantiene en las ejecuciones hechas con el primer procesador cambia en las segundas. Primero se pensara que el procesador crea el código vectorial muy similar al de la segunda versión, por lo que el compilador ha hecho muy bien su trabajo. Esto sucede por la latencia de la función división, que al ser alta, consigue disminuir los problemas de memoria.

La diferencia entre los *speed-up* de los procesadores que se aprecia en el código vectorizado por el compilador, y en el de la segunda versión, puede ser porque el segundo procesador tiene una memoria más rápida, consiguiendo esquivar la limitación de memoria y dependiendo más de la latencia de la división que de la memoria.

3.2 Caso IF

En este algoritmo se usará el condicional por primera vez. Tenemos dos vectores los cuales se dividirán entre ellos, pero para controlar la indeterminación de dividir por 0 usaremos un *if* para solo hacerlo cuando el valor del segundo vector no sea 0. En cuyo caso se asignará al resultado un 0 como valor en esa posición. Finalmente devolverá un vector con las divisiones.

Este es el código escalar:

```
void dividir_vectores ( data_t *a, data_t *b, int n)
{
	int i;
	for (i = 0;i < n; i++)
		if (b[i]!=0) a[i]= a[i]/ b[i];
		else a[i] = 0;
}
```

Y este sería el código vectorial con la tecnología SSE:

```
#define data_t   float

void dividir_vectores ( data_t *pa, data_t *pb, int n){
	int i;
	__m128 a1, b1, c1, cero, condi_eq,condi_neq,div;
	cero=_mm_setzero_ps ();
	for (i = 0;i < n; i+=4)
	{
			a1 = _mm_load_ps(&pa[i]);
			b1 = _mm_load_ps(&pb[i]);
			condi_eq  = _mm_cmpeq_ps(b1,cero);
			condi_neq = _mm_cmpneq_ps(b1,cero);
			b1 = _mm_or_ps(b1, condi_eq);
			div = _mm_div_ps(a1,b1);
			div = _mm_and_ps(div,condi_neq);
			_mm_store_ps(&pa[i], div);
	}
}
```

Y este con la tecnología AVX:

```
#define data_t   float

void dividir_vectores ( data_t *pa, data_t *pb, int n){
       int i;
       __m256 a1, b1, c1, cero, condi_eq,condi_neq,div;
       cero=_mm256_setzero_ps ();
       for (i = 0;i < n; i+=8)
       {
              a1 = _mm256_load_ps(&pa[i]);
              b1 = _mm256_load_ps(&pb[i]);
              condi_eq  = _mm256_cmp_ps(b1,cero,0);
              condi_neq = _mm256_cmp_ps(b1,cero,4);
              b1 = _mm256_or_ps(b1, condi_eq);
              div = _mm256_div_ps(a1,b1);
              div = _mm256_and_ps(div,condi_neq);
              _mm256_store_ps(&pa[i], div);
       }
}
```

En este caso hay muchas diferencias entre el código escalar y el vectorial, esto sucede por el especial funcionamiento que tienen los *if* en el código vectorial. Los *if* se realizan usando máscaras que mediante operaciones lógicas se combinan con los vectores de entrada consiguiendo el resultado deseado. Quitando la forma de realizar el *if* hay las mismas diferencias que en programa anterior, se necesitan *intrinsics* que leen y escriben en la memoria y en los bucles se incrementa el valor de i de cuatro en cuatro en SSE y de ocho en ocho en AVX.

3.2.1 Resultados

Este es el grafico con los resultados:

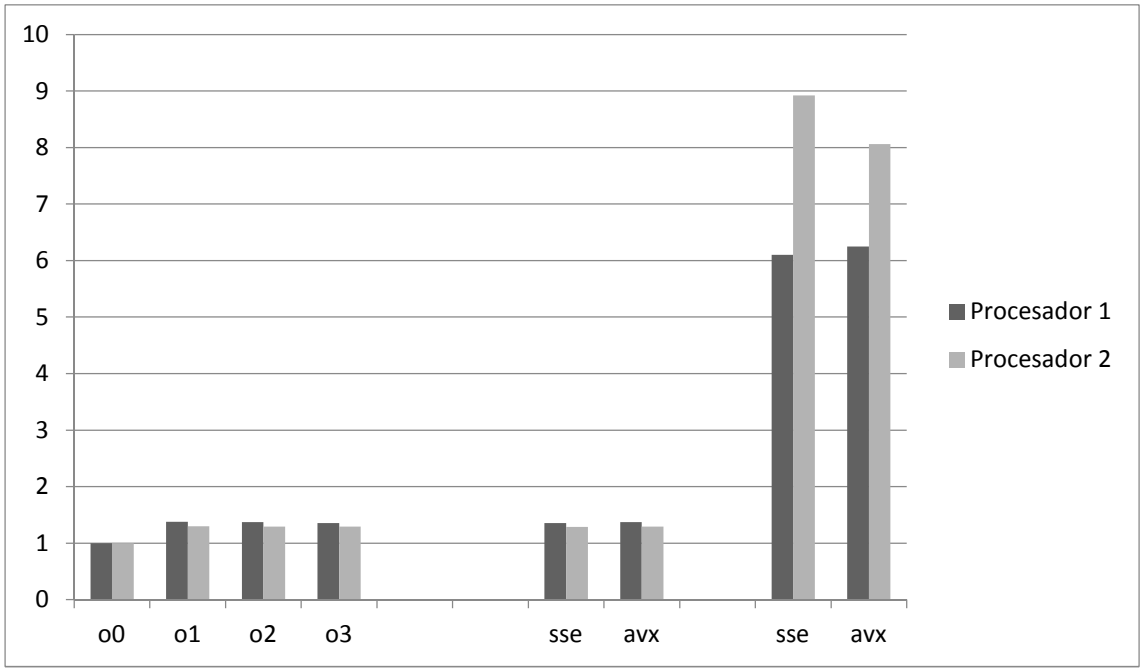


Gráfico 3.2: Resultados caso IF

Se puede ver como solo obtenemos mejoras usando las *intrinsics*. Estas son las causas: como se ha visto en el ejemplo del apartado 2.4.2, el compilador era capaz de no usar el *if*, y realizar la operación mediante máscaras, cosa que no ha hecho en este caso porque la estructura del *if* es más compleja al ser un *if then else.*

Usando las instrinsics si se ha visto una gran mejora al no realizar los *if.* En este caso se puede apreciar cómo con en el segundo procesador tiene un mejor *speed-up* SSE que AVX.

3.3 Caso Reducción

En este algoritmo se suman todos los elementos del vector. Este es un algoritmo simple de forma escalar, pero no tanto de forma vectorial. Al usar vectores en vez de elementos, no se puede obtener el resultado de la suma en un solo elemento, se tiene que usar una función especial para conseguirlo. Esa función especial es la razón por la que tenemos este ejemplo. En el programa se sumarán todos los elementos del vector en una variable y se devolverá esta como salida. Este es el código escalar:

```
#define data_t   float


void reduccion ( data_t *va, data_t *vb, float a,
int n ){
       int i;
       for (i = 0; i < n; i++)
              a=va[i]+a;
}
```

Este es el código vectorial con la tecnología SSE:

```
#define data_t   float

static data_t reduccion_vec (data_t *va, int n ){
       int i;
       __m128  x_va, xsuma;
       data_t a;
       xsuma = _mm_set_ps1(0.0);
       for (i = 0; i < n; i+=4){
         x_va = _mm_load_ps(va+i);
         xsuma = _mm_add_ps(xsuma,x_va);
       }
    xsuma=_mm_hadd_ps (xsuma,xsuma);
    xsuma=_mm_hadd_ps (xsuma,xsuma);
    _mm_store_ss(&a, xsuma);
    return a;
}
```

Este es el código vectorial con la tecnología AVX:

```
#define data_t   float

static data_t reduccion_vec (data_t *va, int n ){
       int i;
       float b;
       __m256  x_va, xsuma;
       xsuma = _mm256_set1_ps(0.0);
       for (i = 0; i < n; i+=8){
         x_va = _mm256_load_ps(va+i);
       xsuma = _mm256_add_ps(xsuma,x_va);
       }
    xsuma=_mm256_hadd_ps (xsuma,xsuma);
    xsuma=_mm256_hadd_ps (xsuma,xsuma);
    xsuma=_mm256_hadd_ps (xsuma,xsuma);
    _mm256_store_ps (&b,xsuma);
    return b;
}
```

No hay mucha diferencia entre el código escalar y el código vectorial, en los dos hay un bucle que suma, en el caso del escalar los elementos, y en el caso del vectorial los vectores de elementos. La diferencia radica en que, en el escalar ya tenemos el resultado, en el vectorial en

cambio, tenemos un vector. La suma de los elementos de ese vector es lo que se busca. Para sumar los elementos del vector tenemos la *intrinsic* especial *hadd*, que va sumando los elementos del vector horizontalmente. En el código se puede ver que se usa esa *intrinsic* es SSE y AVX, pero solamente dos veces en SSE y tres en AVX. Esto sucede, porque, SSE tiene vectores de cuatro elementos por lo tanto para conseguir la suma de todos solamente necesitaremos 2 operaciones. AVX, en cambio, tiene vectores de 8 elementos necesitando ejecutar la *intrinsic* 3 veces para conseguir la suma total.

Una vez que se tiene en la primera posición del vector ***xsuma***, la suma total de los elementos del vector, se pasará a guardarla en una variable escalar. Para ello, se utilizará la *intrinsic* especial de SSE *_mm_store_ss*, que permite guardar solamente el primer elemento del vector en una variable escalar. Esa *intrinsic* solo está disponible en SSE, por lo tanto en AVX no tenemos la *intrinsic* equivalente. Para poder utilizar la *intrinsic* de SSE con el vector de 256 bits se usará una *intrinsic* que hace un casting, convirtiendo el vector de 256 bits en uno de 128bits, y así poder utilizar la *intrinsic*.

3.3.1 Resultados

Aquí está el gráfico con los resultados:

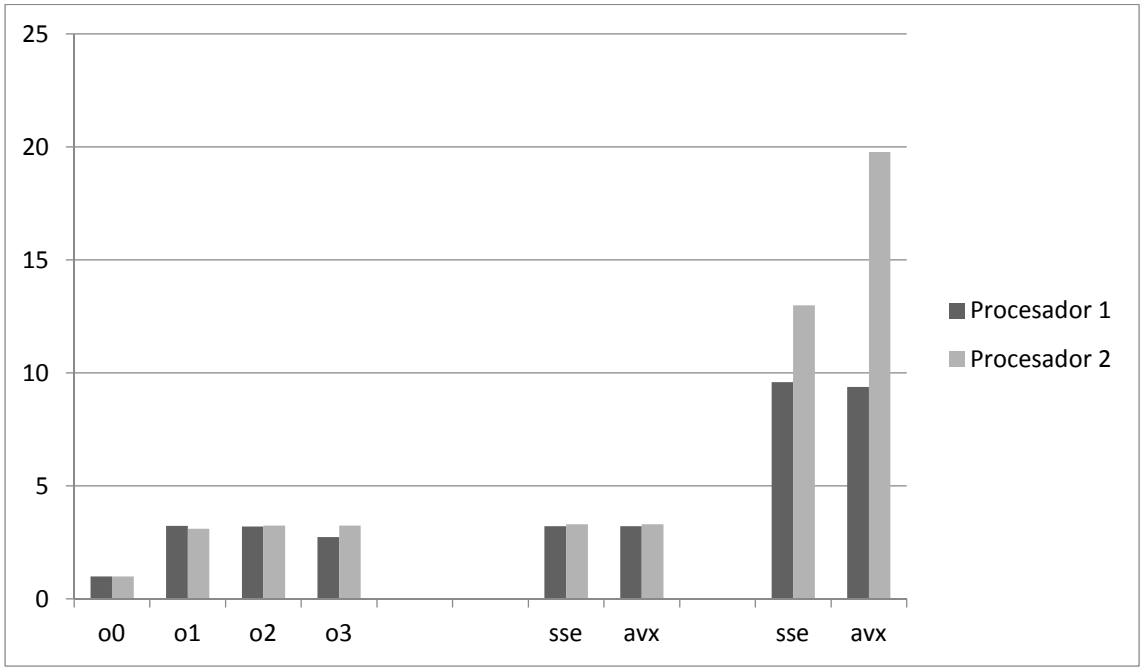


Gráfico 3.3: Resultados caso Reduccion

Se ve como obtenemos un pequeño *speed-up* optimizando el código escalar, o dejándole al compilador vectorizar el código sin que haya una gran diferencia. Estos resultados son malos por la limitación en el acceso de la memoria

Se puede ver como usando las *intrinsics* se consigue un *speed-up* de 10 con el primer procesador con las dos tecnologías. Con el segundo procesador en cambio obtenemos mejores *speed-up* uno de 12 con SSE y otro de casi 20 con AVX.

Como en los otros programas, la diferencia entre los *speed-up* de los programas puede ocurrir por la velocidad de la memoria. Al tener una memoria más veloz con el segundo procesador, este consigue saltarse la limitación de memoria sobre todo en AVX.

Una vez que se controlan los programas más elementales se han implementado otro tipo de programas orientados a obtener mejor rendimiento, una vez que ya se dominan las funciones

básicas del procesamiento vectorial. Para ello se han utilizado unos programas conocidos como BLAS.

3.4 BLAS (Basic Linear Algebra Subprograms)

Son subrutinas de bajo nivel que programan operaciones de algebra lineal básicas como: escalar vectores, multiplicar vectores y matrices, y multiplicar matrices. Estos programas suelen usarse como subrutinas de librerías de un nivel más alto.

Al ser estos subprogramas muy usados, es beneficioso obtener una versión muy eficiente de estos.

BLAS se divide en tres niveles, estos son los tres niveles

- 1º nivel – Consiste en multiplicar un vector por un numero escalar
- 2º nivel – Consiste en multiplicar un vector con una matriz
- 3º nivel – Consiste en multiplicar dos matrices

Ahora explicaremos cada nivel de uno en uno:

3.4.1 Primer nivel

Esta es la fórmula del primer nivel: **y= αx+y**

Siendo **α** un numero escalar y **x** e **y** vectores. Primero se multiplica el número escalar por los elementos del vector **x** y luego se sumarán los elementos del vector **y**. Este es el código escalar de la función:

```
#define data_t   float


void blas1 ( data_t *va, data_t *vb, float a, int n
){
       int i;
       for (i = 0; i < n; i++)
                     va[i]=va[i]*a+vb[i];
}
```

Se realizan todas las operaciones en una sola línea de código. Aquí tenemos el código vectorial con la tecnología SSE:

```
#define data_t   float

static void blas1_vec (data_t *va,data_t *vb,float a, int n ){
       int i;
       __m128 v_indice, x_va, x_vb, x_vl;
    v_indice = _mm_set_ps1 (a);
       for (i = 0; i < n; i+=4){
         x_va = _mm_load_ps(va+i);
         x_vb = _mm_load_ps(vb+i);
          x_va = _mm_mul_ps(x_va,v_indice);
         x_va = _mm_add_ps(x_va,x_vb);
         _mm_store_ps(va+i,x_va);
       }
}
```

Y aquí tenemos el código con la tecnología AVX:

```
#define data_t   float

static void blas1_vec (data_t *va,data_t *vb,float a, int n ){
       int i;
       __m256 v_indice, x_va, x_vb, x_vl;
       v_indice = _mm256_set1_ps(a);
       for (i = 0; i < n; i+=8)
       {
         x_va = _mm256_load_ps(va+i);
         x_vb = _mm256_load_ps(vb+i);
         x_va = _mm256_mul_ps(x_va,v_indice);
         x_va = _mm256_add_ps(x_va,x_vb);
         _mm256_store_ps(va+i,x_va);
       }
}
```

En el código vectorial, se realizan todas las operaciones que se hacen en una línea en el código escalar, pero de una en una. Primero se cargan los dos vectores, se multiplica el primero por la variable, ***v_indice***, la cual antes del bucle se ha inicializado con el número correspondiente. Finalmente se sumará el resultado de la multiplicación con el segundo vector, y se guardarán los valores en la memoria. Se puede ver como no hay ninguna diferencia entre el código de SSE y el de AVX. Estos son los resultados obtenidos:

3.4.1.1 Resultados

Este es el gráfico con los resultados:

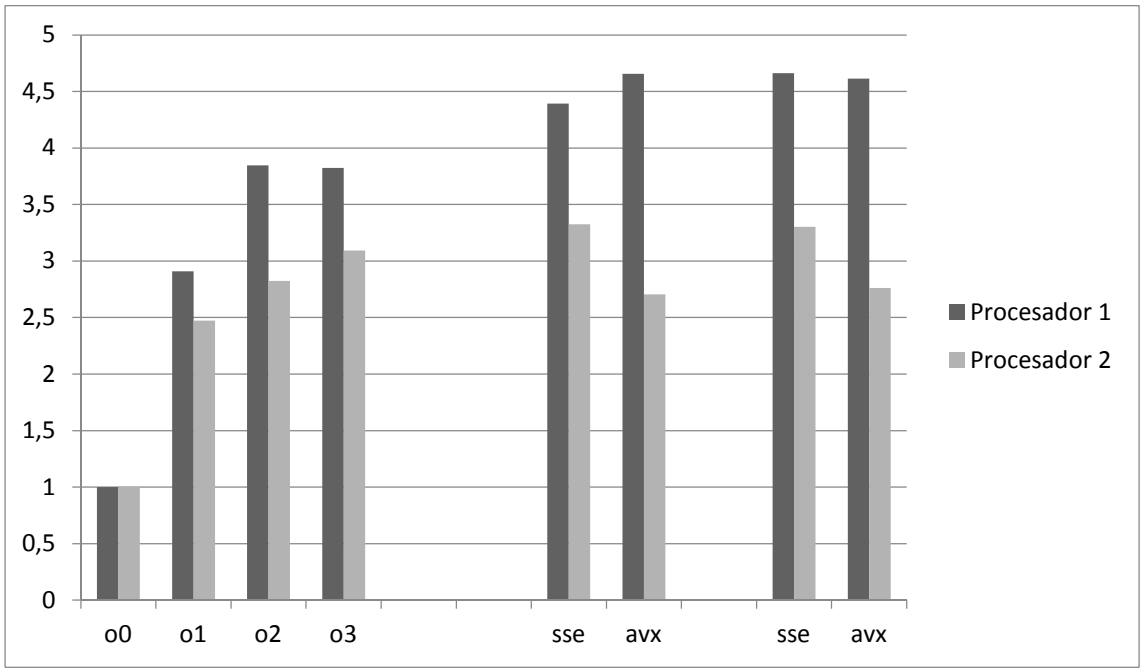


Gráfico 3.4: Resultados BLAS1

Se puede ver como en ningún caso obtenemos una mejora muy grande y también se ve como no hay mucha diferencia entre las diferentes compilaciones.

El motivo de estos *speed-up* es otra vez la limitación en el acceso de la memoria que aunque en un ejemplo anterior, la latencia de la operación realizada disminuía su efecto, sobre todo al ejecutar con *intrinsics*, en este caso al ser una suma y una multiplicación no tienen una latencia suficientemente alta como para paliar los efectos de la limitación de memoria.

3.4.2 Segundo nivel

Esta es la fórmula del segundo nivel: ***y = αAx + yβ***

Donde ***α*** y ***β*** son números escalares, ***x*** e ***y*** son vectores, y ***A*** es una matriz. En este programa se multiplica una matriz por un vector consiguiendo un vector como resultado (x=Ax). Se multiplica uno de esos vectores por un numero escalar (x=αx), finalmente se suman los dos vectores, una vez que se multiplique ***y*** por el escalar ***β*** (y= x+y*β). Este es el código escalar del algoritmo:

```
void Vector_plus_i ( data_t pa[][N], data_t *vx,data_t *vy,data_t *ve, int
n,float a, float b){
        int i,j;
        float bat,bat1;
        for(i=0; i<n; i++){
             bat=0;
             for(j=0; j<n; j++){
                   bat += pa[i][j]*vx[j];
             }
             ve[i]=bat*a+vy[i]*b;
        }
}
```

En el segundo *for* se calcula la multiplicación de la matriz con el vector. Cada vez que se termine la multiplicación de una columna de la matriz y el vector se le multiplica su escalar (a) y se le suma la multiplicación del segundo vector por su escalar (vy[i]*b).

Este es el código vectorial con SSE, se realizara la operación anterior en dos partes:

```
static void Blas2_vec ( data_t * pa,data_t * vx,data_t * vy, int n,float a,float
b){
       int j,k,i;
       __m128 a1, a2, a3, b1, c1, lag, lag2;
       float w[N];
    lag = _mm_set_ps1 (a);
    lag2 = _mm_set_ps1 (b);
    for (j = 0;j < n; j++){
        a1=_mm_setzero_ps();
        for (k = 0; k < n/4; k++){
            c1=_mm_load_ps(&pa[4*k]);
            b1=_mm_load_ps(&vx[4*k]);
            c1=_mm_mul_ps(b1,c1);
            a1=_mm_add_ps(a1,c1);
        }
        a1 = _mm_hadd_ps(a1,a1);
        a1 = _mm_hadd_ps(a1,a1);
        a1=_mm_mul_ps(a1,lag);
        _mm_store_ss(&w[j], a1);
    }
    for(i=0;i<n/4;i++){
       a3=_mm_load_ps(&w[4*i]);
       a2=_mm_load_ps(&vy[4*i]);
       a2=_mm_mul_ps(a2,lag2);
       a3=_mm_add_ps(a3,a2);
       _mm_store_ps(&vy[i*4], a3);
    }
}
```

Y aquí tenemos el código con la tecnología AVX:

```
static void blas2_vec ( data_t * pa,data_t * vx,data_t * vy, int
n,float a,float b){
       int j,k,i;
       __m256 a1, a2, b1, c1, lag, lag2;
       float w[N];
    lag = _mm256_set1_ps (a);
    lag2 = _mm256_set1_ps (b);
    for (j = 0;j < n; j++){
        a1=_mm256_setzero_ps();
        for (k = 0; k < n/8; k++)
        {
            c1=_mm256_load_ps(&pa[8*k]);
            b1=_mm256_load_ps(&vx[8*k]);
            c1=_mm256_mul_ps(b1,c1);
            a1=_mm256_add_ps(a1,c1);
        }
        a1 = _mm256_hadd_ps(a1,a1);
        a1 = _mm256_hadd_ps(a1,a1);
        a1 = _mm256_hadd_ps(a1,a1);
        a1=_mm256_mul_ps(a1,lag);
        _mm_store_ss(&w[j], _mm256_castps256_ps128 (a1));
    }

    for(i=0;i<n/8;i++){
       a1=_mm256_load_ps(&w[8*i]);
       a2=_mm256_load_ps(&vy[8*i]);
       a2=_mm256_mul_ps(a2,lag2);
       a1=_mm256_add_ps(a1,a2);
       _mm256_store_ps(&vy[i*8], a1);
    }
}
```

En la primera parte (los dos *for*) se realiza la multiplicación de la matriz con el vector. Antes de guardar en la memoria el valor de la multiplicación matriz y vector, también se multiplica por el escalar. Así se obtiene la primera parte de la formula, ***α*A*x***.

Para realizar la segunda parte se usara un bucle, en el cual se cargarán los dos vectores, pero antes de sumarlos se multiplicará el segundo vector por el escalar.

3.4.2.1 Resultados

Aquí está el gráfico con los resultados:

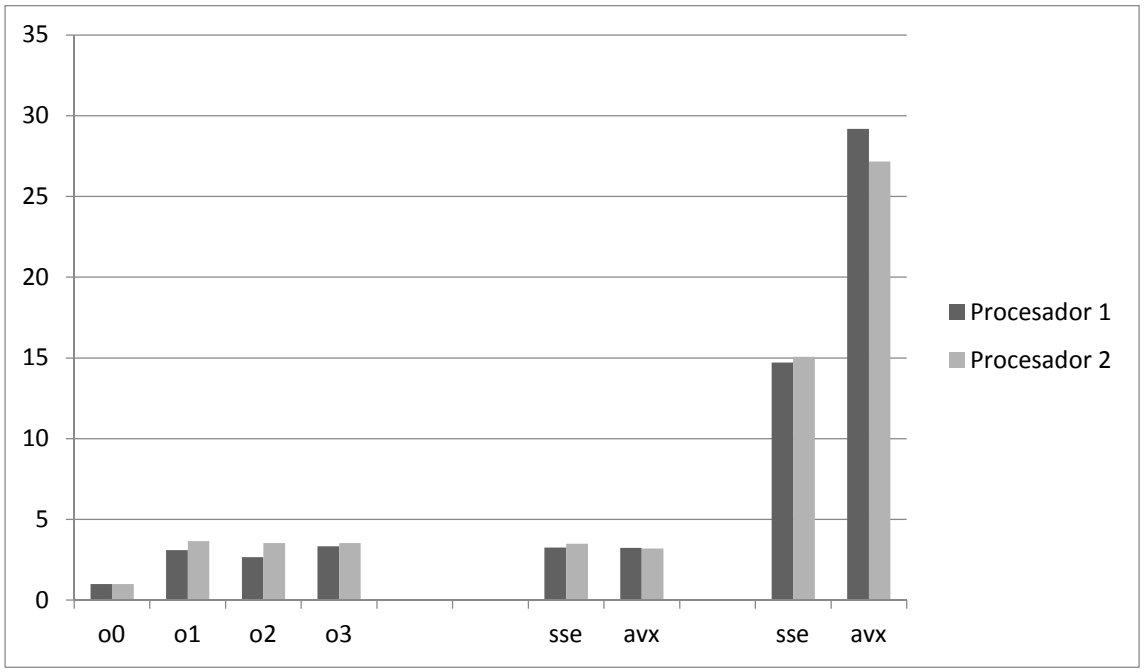


Gráfico 3.5: Resultados BLAS2

En el gráfico se observa cómo no se obtiene una gran mejora con las optimizaciones del código escalar y la vectorización del compilador. Esto ocurre por la limitación en el acceso de la memoria.

En cambio con las *intrinsic* se ve como si se consigue un buen *speed-up* de 15 con SSE y unos 27,5 con AVX. Usando las *intrinsic* hemos conseguido el código más óptimo posible, se concluye esto porque el *speed-up* que se consigue es, más o menos el doble, por lo que la mejora obtenida con AVX respecto a SSE es por el tamaño de los vectores.

3.4.3 Tercer nivel

Esta es la fórmula del segundo nivel: $\boldsymbol{C = \alpha AB + \beta C}$

Donde $\boldsymbol{A}$, $\boldsymbol{B}$, y $\boldsymbol{C}$ son matrices y, $\boldsymbol{\alpha}$ y $\boldsymbol{\beta}$ son números escalares. Primero multiplicaremos las matrices $\boldsymbol{A}$ y $\boldsymbol{B}$ multiplicando el resultado por $\boldsymbol{\alpha}$.

Esta operación se realiza de dos maneras. El primero con los bucles en este orden “ijk”, y la segunda con los bucles en este orden “ikj”. Se verá como el segundo método es más eficiente.

```
void producto_matriz ( data_t ma[][N],data_t mb[][N],data_t mc[][N],
float a,float b ){
    int i,j,k;
       float w;
       for (i = 0;i < N; i++)
              for (j = 0;j < N; j++){
                     w=0.0f;
                     for (k = 0;k < N; k++)
                            w=(w+ma[i][k]*mb[k][j])*a;
           mc[i][j] = mc[i][j]*b+w;
        }
}
```

Este es el código escalar del segundo método:

```
void producto_matriz_escalar ( data_t  ma[][N],data_t  mb[][N],data_t
mc[][N],float a, float b ){
       int i,j,k;
       float t;
       for (i = 0;i < N; i++) {
              for (j = 0;j < N; j++) mc[i][j]*=b;
              for (k = 0;k < N; k++){
                     t = ma[i][k];
                     for (j = 0;j < N; j++){
                     mc[i][j]+=(t*mb[k][j])*a;
              }
         }
    }
}
```

La diferencia entre los dos métodos es el *stride* del recorrido del vector, es decir la diferencia entre las posiciones leídas de la memoria en cada iteración. En el primer método hay un *stride* de N, en el segundo en cambio solo hay uno de 1.

Esto también pasara en el código vectorizado, aquí está el código con *intrinsics* SSE, del primer método:

```
static void producto_matriz_ijk ( data_t * pa,data_t * pb,data_t * pc,float a,
float b)

{

       int i,j,k,indb,inda,indc;

       float w;

       __m128 a1, b1, c1, lag,lag1, a2, d2;

       lag = _mm_set_ps1(a);

       lag1 = _mm_set_ps1(b);

       for (i = 0,inda=indb=0;i < N; i++,indb+=N,inda+=N){

         for (j = 0;j < N; j++){

                a1=_mm_setzero_ps();

                for (k = 0, indc=0; k < N/4; k++,indc+=(4*N)){

                     c1=_mm_set_ps(pb[indc+j],pb[indc+N+j],pb[indc+2*N+j],

                            pb[indc+3*N+j]);

                     b1=_mm_load_ps(&pa[indb+4*k]);

                     c1=_mm_mul_ps(b1,c1);

                     a1=_mm_add_ps(a1,c1);

                     }

                a1 = _mm_hadd_ps(a1,a1);

                a1 = _mm_hadd_ps(a1,a1);

                a1=_mm_mul_ps(a1,lag);

                _mm_store_ss(&w, a1);

                pc[indb+j]= pc[indb+j]*b+w;

         }

       }

}
```

Se ve cómo aunque multipliquemos de forma vectorial las matrices en el tercer *for*, la operación de sumar el resultado y la matriz C se hace de forma escalar, como se ve en la última línea de código.

Y aquí el código del segundo método:

```
static void producto_matriz_ikj ( data_t * pa,data_t * pb,data_t * pc,  float a,
float b){
	int i,j,k,indb,inda,indc;
	__m128 a1, b1, c1,lag,lag1,a2,d2;
	lag = _mm_set_ps1(a);
	lag1 = _mm_set_ps1(b);
	for (i = 0,inda=indb=0 ;i < N; i++,inda+=N, indb+=N){
		for (j=0;j<N;j+=4){
			c1=_mm_load_ps(&pc[indb+j]);
			a1=_mm_mul_ps(c1,lag1);
			_mm_store_ps(&pc[indb+j], a1);
		}
		for (k = 0, indc=0;k < N; k++,indc+=N){
			b1=_mm_load1_ps(&pa[indb+k]);
			for (j = 0;j < N; j+=4){
				c1=_mm_load_ps(&pb[indc+j]);
				a1=_mm_load_ps(&pc[inda+j]);
				c1=_mm_mul_ps(b1,c1);
				a1=_mm_add_ps(a1,c1);
				a1=_mm_mul_ps(a1,lag);
				_mm_store_ps(&pc[inda+j], a1);
			}
		}
	}
}
```

En estos programas se ve mucho más claro el *stride* que tiene cada método. En el primer método se realiza un set donde cargamos números con saltos en la memoria de N, en el segundo en cambio, solo se realizan lecturas a direcciones de memoria consecutivas, por lo tanto tendrán un *stride* de 4 u 8.

Ahora se verá como sucede igual con AVX, este es el código del primer método:

```
static void producto_matriz_ijk ( data_t * pa,data_t * pb,data_t *
pc,float a, float b){
       int i,j,k,indb,inda,indc;
       float w;
       __m256 a1, b1, c1, lag,lag1, a2, d2;
       lag = _mm256_set1_ps(a);
       lag1 = _mm256_set1_ps(b);
       for (i = 0,inda=indb=0;i < N; i++,indb+=N,inda+=N){
         for (j = 0;j < N; j++){
                a1=_mm256_setzero_ps();
                for(k = 0, indc=0; k < N/8; k++,indc+=(8*N)){
                     c1=_mm256_set_ps(pb[indc+j],pb[indc+N+j],
                     pb[indc+2*N+j],pb[indc+3*N+j],pb[indc+4*N+j],
                     pb[indc+5*N+j], pb[indc+6*N+j],pb[indc+7*N+j]);
                     b1=_mm256_load_ps(&pa[indb+8*k]);
                     c1=_mm256_mul_ps(b1,c1);
                     a1=_mm256_add_ps(a1,c1);
                }
                a1 = _mm256_hadd_ps(a1,a1);
                a1 = _mm256_hadd_ps(a1,a1);
                a1 = _mm256_hadd_ps(a1,a1);
                a1=_mm256_mul_ps(a1,lag);
                _mm_store_ss(&w,_mm256_castps256_ps128 (a1));
                pc[indb+j]= pc[indb+j]*b+w;
         }
       }
 }
```

Y este el del segundo código:

```
static void producto_matriz_ikj ( data_t * pa,data_t * pb,data_t * pc,  float
a, float b){
       int i,j,k,indb,inda,indc;
       __m256 a1, b1, c1,lag,lag1,a2,d2;
       __m128 b2;
       float l;
    lag = _mm256_set1_ps(a);
    lag1 = _mm256_set1_ps(b);
       for (i = 0,inda=indb=0 ;i < N; i++,inda+=N, indb+=N){
         for (j=0;j<N;j+=8){
              c1=_mm256_load_ps(&pc[indb+j]);
              a1=_mm256_mul_ps(c1,lag1);
              _mm256_store_ps(&pc[indb+j], a1);
         }
         for (k = 0, indc=0;k < N; k++,indc+=N){
              b2=_mm_load1_ps(&pa[indb+k]);
              _mm_store_ss(&l,b2);
              b1 = _mm256_set1_ps(l);
              for (j = 0;j < N; j+=8){
                     c1=_mm256_load_ps(&pb[indc+j]);
                     a1=_mm256_load_ps(&pc[inda+j]);
                     c1=_mm256_mul_ps(b1,c1);
                     a1=_mm256_add_ps(a1,c1);
                     a1=_mm256_mul_ps(a1,lag);
                     _mm256_store_ps(&pc[inda+j], a1);
              }
         }
       }
}
```

Respecto a la versión de SSE tiene un pequeño cambio al inicializar la variable ***b1***. En SSE usamos una *intrinsic* que no tiene su equivalente en AVX, por lo tanto tenemos que usar unas cuantas *intrinsics* más, que están señaladas con flechas.

3.4.3.1 Resultados

Aquí está el gráfico con los resultados del primer método:

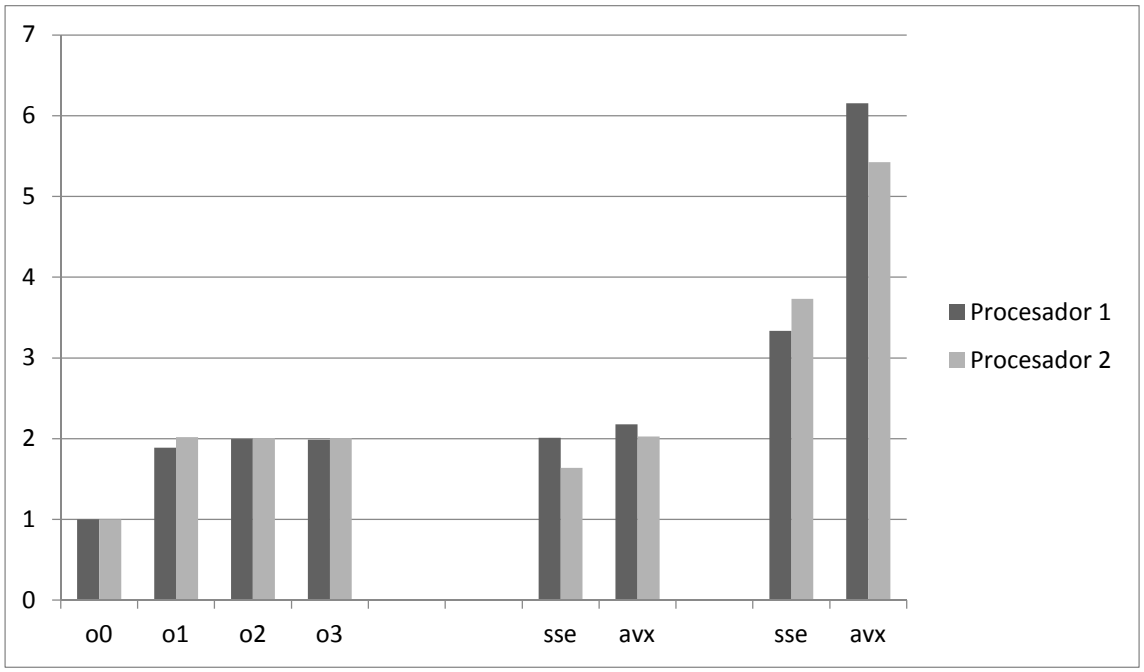


Gráfico 3.6: Resultados BLAS3 ijk

Se ve que como no se obtienen muy buenos resultados compilando el código con optimizaciones o dejándole vectorizar el código al compilador. Se consiguen mejores speed-up con las *intrinsics.*

Aquí está el gráfico con los resultados del segundo método:

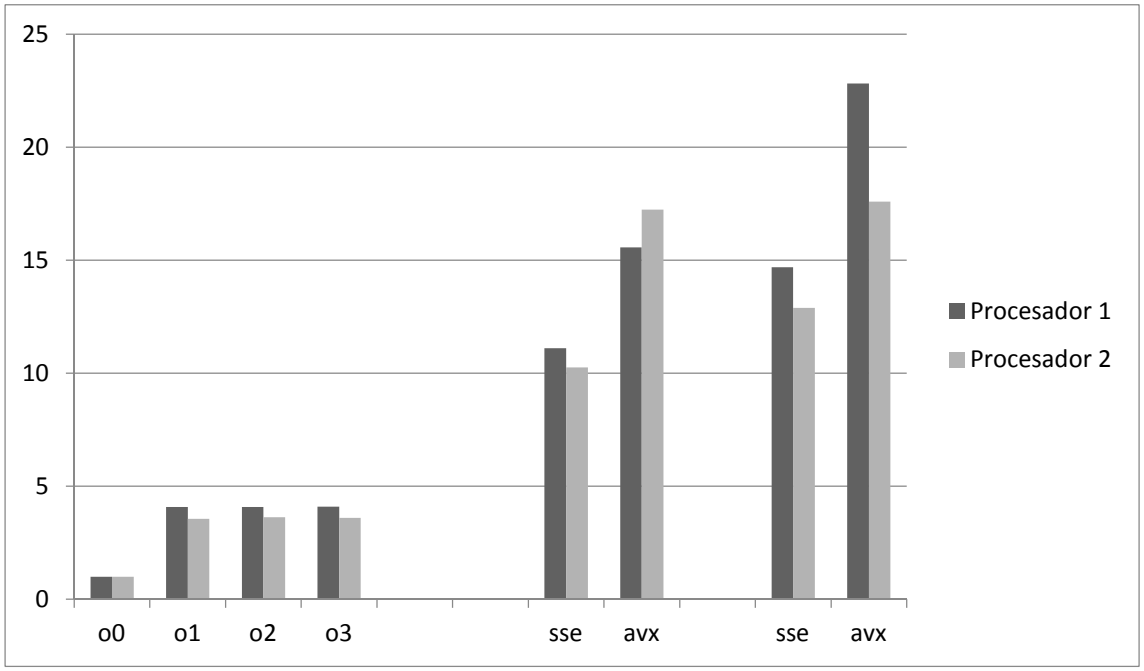


Gráfico 3.7: resultados BLAS3 ikj

Se puede observar cómo se obtienen mejores *speed-ups* con este método. Además, también mejora con el código vectorizado por el compilador consiguiendo con el segundo procesador prácticamente las mismas mejoras.

En cambio con el primer procesador mejora mucho con el uso de *intrinsics*.

Capítulo 4

4 Conclusiones

4.1 Resumen de los resultados obtenidos

	ejemplo		div		if		reducción		BLAS1		BLAS2		BLAS3ijk		BLAS3ikj	
Procesador	P1	P2	P1	P2	P1	P2	P1	P2	P1	P2	P1	P2	P1	P2	P1	P2
Speed-up (max)	41	28	7	10	6	9	9	19	4	3	29	27	6	5	22	17
SSE-AVX	0,11x	0.25x	≈		≈		≈	1,5x	≈		2x		2x	1,5x	1,5x	
Comp: manual Intrinsics	10x	7x	≈		4x	6x	3x	6x	≈		10x	9x	3x	2x	1,5x	≈

Tabla 4.1: Resumen speed-up

Se puede ver como en todos los programas hemos obtenido *speed-up*, aunque haya mucha diferencia entre el *speed-up* máximo (41) y mínimo (3).

Se ven claramente dos tipos de programa, unos tienen limitación de memoria, en estos el segundo procesador consigue un rendimiento mejor, sobre todo en AVX, al tener una memoria más rápida pudiendo esquivar mejor esa limitación que el primer procesador con su memoria más lenta.

El otro grupo (los de BLAS, y el ejemplo) reutilizan más los recursos obteniendo mejores *speed-up* en general. En estos programas el primer procesador obtiene mejores resultados porque la ejecución escalar del mismo tiene más *ticks*, por lo tanto las mejoras son más altas, aunque los *ticks* sean parecidos en los dos procesadores con las *intrinsic*.

También se ve como en los programas que hay limitación de memoria se consigue el mismo *speed-up* con las dos tecnologías cuando supuestamente el *speed-up* de AVX debería ser el

doble que el de SSE. Se consigue que sea el doble en los programas que se reutilizan los recursos, aunque no siempre se llegue a ese límite del doble.

Se ha visto que dejándole vectorizar al compilador se han obtenido buenas mejoras, aunque como se puede ver en la última línea de la tabla, las conseguidas con las *intrinsics* son mejores en la mayoría de los programas por lo tanto si quiere conseguir el rendimiento máximo se deberán usar las *intrinsics*.

4.2 Conclusiones

No se han conseguido todos los objetivos al no poder ejecutar los programas creados en AVX2, aunque sí se ha estudiado su funcionamiento teórico. Quitando eso se han conseguido todos los demás objetivos.

Se ha conseguido ejecutar los programas con las tecnologías SSE y AVX, y se ha visto que se consigue un gran rendimiento en casi todos los casos, llegando en un caso a ejecutar el programa 41 veces más rápido.

Se ha visto que solamente dejándole al compilador realizar el trabajo "sucio" de vectorizar, en la mayoría de programas se consigue un buen *speed-up* llegando a ejecutar 17 veces más rápido en el mejor de los casos. En los programas que no se ha conseguido un *speed-up* grande se ha estado limitado por la memoria, por lo tanto se puede suponer que el compilador ha hecho un buen trabajo.

Con la apuesta de Intel por este tipo de tecnología, podemos creer que cumplirá su promesa de ir doblando el tamaño de los registros, aunque no sea cada dos años como prometieron en un principio.

4.3 Desarrollo del proyecto en el futuro

Como principal trabajo para el futuro es realizar las mismas pruebas que se han realizado con estas dos tecnologías con AVX2 y comparar los resultados para ver si mejora el rendimiento respecto a las anteriores tecnologías.

Además de eso, siendo las extensiones multimedia una tecnología en constante actualización con su siguiente versión (AVX-512), también se podrá probar esta nueva tecnología y comparar los resultados con los obtenidos en estas pruebas.

Además de las diferentes tecnologías se pueden realizar más tipos de algoritmos. Especialmente interesantes son los algoritmos que trabajen con gráficos o matrices.

5 Enlaces y bibliografía mínima

Enlaces:

Guia de intrinsics (https://software.intel.com/sites/landingpage/IntrinsicsGuide/)

en.wikipedia.org/

Introduction to Intel® Advanced Vector Extensions (https://software.intel.com/en-us/articles/introduction-to-intel-advanced-vector-extensions)

Significado elementos comparación AVX (http://stackoverflow.com/questions/16988199/how-to-choose-avx-compare-predicate-variants)

Bibliografia:

Aart J.C. Bik. The Software Vectorization Handbook:Applying Multimedia Extensions for Maximum Perfomance. Intel Press, 2004.

6 Apéndice

Apéndice A: Tabla de ticks

Apéndice B Tabla de speed-up

Apéndice A

A Tabla de ticks

		div		if		ejemplo	
escalar							
	o0	19464216	15520391	59288	45442	3365141	2470363
	o1	18755833	15497268	42950	35064	1260380	1464897
	o2	18879815	15471857	43127	35085	683111	972690
	o3	19376471	15572076	43662	35085	964629	886599
vectorial							
compilador							
	sse	2874430	1563835	43709	35280	714759	568509
	avx	3011906	1620570	43224	35138	831503	589584
manual							
v1	sse	3024868	2665780	9718	5094	95215	102958
	avx	3105056	3056857	9485	5638	81547	85509
v2	sse	2856429	1522407				
	avx	2730011	1583617				

Tabla A.1: Tabla ticks 1

		blas1		blas2		producto ijk	
escalar							
	o0	8421318	6818886	34041099	36769992	516354204	457996958
	o1	2895141	2758630	10985777	10062223	273716617	226904528
	o2	2189253	2415127	12816768	10403468	258634596	228238544
	o3	2202735	2204814	10189359	10398415	259701932	228130347
vectorial							
compilador							
	sse	1916532	2050437	10464091	10521084	256545348	279789509
	avx	1808953	2521839	10512397	11472537	236919315	225868200
manual							
v1	sse	1806870	2064240	2312882	2440591	154746503	122705562
	avx	1824997	2469402	1166329	1354247	83934800	84423192

Tabla A.2: Tabla ticks 2

		producto ikj		reducción	
escalar					
	o0	414731400	321141417	8398054	8628704
	o1	101569593	90189931	2598494	2776913
	o2	101620851	88331335	2616309	2656177
	o3	101331140	89304770	3067681	2655367
vectorial					
compilador					
	sse	37338360	31306089	2609877	2612400
	avx	26638088	18626309	2611882	2607395
manual					
v1	sse	28226557	24906363	875288	664336
	avx	18183853	18259431	895106	436551

Tabla A.3: Tabla ticks 3

Apéndice B

B Tabla con los speed-ups

Speed-up		div		if		ejemplo	
	o1	1,0377687	1,0014921	1,3803958	1,2959731	2,669941605	1,686373172
	o2	1,0309537	1,0031369	1,3747304	1,2951974	4,926199402	2,539722831
	o3	1,0045284	0,9966809	1,3578856	1,2951974	3,488533934	2,786336326
vectorial							
compilador							
	sse	6,7715046	9,9245707	1,3564255	1,2880385	4,708077828	4,345336661
	avx	6,4624248	9,5771185	1,3716454	1,2932438	4,047058159	4,190010245
manual							
v1	sse	6,4347324	5,8220825	6,1008438	8,920691	35,34255107	23,99389071
	avx	6,2685555	5,0772382	6,2507116	8,0599503	41,26627589	28,89009344
v2	sse	6,8141781	10,19464				
	avx	7,1297207	9,8005964				

Tabla B.1: Tabla speed-up 1

Speed-up		BLAS1		BLAS2		producto ijk	
	o1	2,908776	2,471838	3,0986519	3,6542613	1,88645545	2,01845667
	o2	3,846663	2,823407	2,6559815	3,5343976	1,99646224	2,00665913
	o3	3,823119	3,092726	3,3408479	3,5361151	1,98825708	2,00761084
vectorial							
compilador							
	sse	4,39404	3,325577	3,2531348	3,4948863	2,01272098	1,63693399
	avx	4,655355	2,703934	3,2381862	3,2050445	2,1794517	2,02771775
manual							
v1	sse	4,660722	3,30334	14,718044	15,06602	3,33677462	3,73248735
	avx	4,614428	2,761351	29,186532	27,151614	6,15184886	5,42501352

Tabla B.2: Tabla speed-up 2

Speed-up		producto ikj		Reducción	
	o1	4,083224	3,56072361	3,231893	3,107301
	o2	4,08116441	3,63564546	3,209886	3,248543
	o3	4,09283267	3,59601639	2,73759	3,249533
vectorial					
compilador					
	sse	11,1073813	10,2581136	3,217797	3,30298
	avx	15,5691129	17,2412804	3,215327	3,30932
manual					
v1	sse	14,6929503	12,8939507	9,594618	12,98846
	avx	22,8076745	17,5877012	9,382189	19,76563

Tabla B.3: Tabla speed-up 3