COMPUTACIÓN DE ALTAS PRESTACIONES

TEMA 3

Vectorización, paralelismo de datos

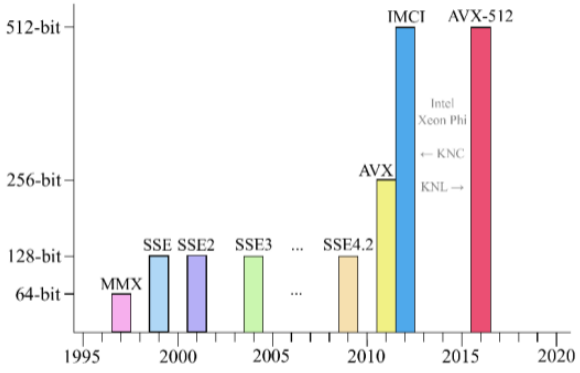
1º Las arquitecturas SIMD

- Los procesadores SIMD se basan en realización de una misma operación de forma simultánea sobre un conjunto de datos, en lugar de realizarla secuencialmente, como ocurre en los procesadores escalares. Estos surgen para el procesamiento de aplicaciones multimedia, las cuales tienen asociada forma intrínseca una gran cantidad de paralelismo en sus cálculos.

- El crecimiento en el tamaño en bits del tipo de datos utilizados para este tipo de aplicaciones se ve trincado por las propias limitaciones del ser humano, ya que al ser productos centrados en al interacción con las personas, no tiene sentido utilizar precisiones no perceptibles por las mismas.

- Este tipo de aplicaciones explotan enormemente la localidad temporal, ya que se suele operar con estructuras que tienen datos consecutivos en memoria, como pueden ser matrices. Sin embargo, no explotan la localidad temporal, ya que lo normal es que una vez se ha operado con un conjunto de datos, este no vuelva a ser utilizado en el corto plazo.

- La implementación de operaciones SIMD fue implementada por primera vez mediante el juego de instrucciones MMX de Intel. Su funcionalidad se basaba en la idea de que el uso de aplicaciones multimedia requería de datos con baja precisión, por lo que dividirían las unidades aritméticas de la CPU, las cuales empleaban 32 bits, para operar en su lugar con un total de 4 datos de 8 bits de manera simultánea.



- Para poder operar de esta manera con un conjunto de datos, las unidades aritméticas necesitaban conmutan su modo de funcionamiento, lo cual requería guardar los datos procesados en ese momento en la memoria caché. Esto suponía una gran perdida de aceleración debido a los múltiples cambios entre las ejecuciones secuenciales y las vectoriales.

1.1 Características de algunas arquitecturas SIMD

- **SSE:** Tamaño máximo del vector 128 bits.
  - Trabajo con enteros de 8, 16 , 34 o 64 bits.
  - Trabajo en punto flotante con 32 y 64 bits.
  - Dependiendo de los datos de trabajo, el vector puede ser de longitud 2, 4, 8 o 16 elementos.

- **AXV:** Tamaño máximo del vector 256 bits.
  - Trabajo con enteros de 8, 16 , 32 o 64 bits.
  - Trabajo en punto flotante con 32 y 64 bits.
  - Dependiendo de los datos de trabajo, el vector puede ser de longitud 4, 8, 16 o 32 elementos.

- **Intel MIC:** Tamaño máximo del vector 512 bits.
  - Trabajo con enteros de 32 o 64 bits.
  - Trabajo en punto flotante con 32 y 64 bits.
  - Dependiendo de los datos de trabajo, el vector puede ser de longitud 8 o 16 elementos.

2º Vectorización automática

– En la vectorización automática, el compilador genera código vectorial sin la necesidad de que el programador indique nada sobre el mismo. Este tipo de vectorización se centra en la ejecución paralela de las diferentes iteraciones de un bucle, sin embargo, se deben cumplir unos requisitos:

  - Únicamente funciona para los bucles internos.
  - El número de iteraciones del bucle debe ser bien conocida, y mejor si la variable de control se incrementa o decrementa de forma consecutiva.
  - No deben existir interdependencias entre las distintas iteraciones del bucle.
  - En el caso de vectorizar funciones, esto debe ser especificado.

– Para poder operar con la vectorización de elementos, vamos a utilizar el compilador de icc de Intel, cuyo comando de compilación más básico es: ***icc -o "ficheroCompilacion" "ficheroFuente.c"***.

– El compilador trae consigo una gran cantidad de parámetros mediante los cuales se podremos modificar la compilación que vamos a realizar. Algunos de ellos son:

  - **-x[code]:** Código de la arquitectura SIMD específica para la cual vamos a generar la compilación, de modo que la compilación sera óptima para el uso de la misma. La opción *host* nos permite indicar al compilador que utilice aquella versión que considera más apropiada para el equipo.

code	Target Architecture
MIC-AVX512	Intel Xeon Phi (KNL)
CORE-AVX512	Intel Xeon Proc. AVX512
CORE-AVX2	Intel Xeon Proc. E3/E5/E7 v3, v4
AVX	Intel Xeon Proc. E3/E5 and E3/E5/E7 v2
SSE4.2	Intel Xeon Proc. 55XX, 56XX, 75XX y E7
host	where is compiled

  - **-gopt-report=[nivel]:** Generamos un fichero .optrpt donde se nos muestra información a cerca de la vectorización de bucles realizada en la compilación. Podemos elegir el nivel de la información (0-5), siendo 0 sin diagnosticar y 5 el análisis más completo.

2.1 Elementos a tener en cuenta

– El uso de punteros dentro de un bucle suele inhibir la vectorización automática. Esto es debido a que el compilador no conoce el valor referenciado por el puntero, sino únicamente la dirección del mismo, por lo que detecta una posible interdependencia y decide no vectorizarlo.

– Las conversiones entre distintos tipos de datos afectan gravemente al rendimiento, ya sea entre enteros y flotantes o entre simples y dobles precisiones. Deberemos intentar utilizar datos que al operar entre sí no precisen de conversiones.

– Utilizar la opción de compilación para la arquitectura SIMD más avanzada que soporta el procesador supone un gran aumento en el rendimiento de la vectorización.

– El uso de Arrays de Estructuras (AoS) en lugar de una Estructura de Array (SoA) puede facilitar el acceso alineado a los datos con los que se va a operar en la vectorización, por lo que aumenta el rendimiento de la misma.

– Declarar los arrays de forma alineada mediante el uso de intrínsecas dedicadas a ello hace que el acceso a dichas variables pueda realizarse obteniendo tantos datos como el tamaño máximo de vectorización permite. Utilizando esto tendremos que realizar un menor número de accesos.

3º Vectorización guiada

– La vectorización guiada se emplea cuando el código consta de alguna parte que puede ser electrizable pero que el compilador no vectoriza por que no cumple alguna de las condiciones necesarias para que se aplique la vectorización automática.

example_simd.c

```
for (int i = 1; i < n; i++)
   a[i] += a[i-1]; // dependence on the previous element
```

example_simd.c

```
for (int i = 0; i < n-1; i++)
   a[i] += a[i+1]; // no dependence on the previous element
```

– Esta vectorización se realiza en base a especificar una serie de etiquetas en el código denominadas pragmas, las cuales tienen diversas funcionalidades y guían al compilador para llevar a cabo la vectorización del código.

– Estas pragmas afectan a los bucles situados en la instrucción situada justamente después y se especifican utilizando el carácter #, de modo que si el código se compila sin utilizar los compiladores de Intel para vectorización, estas no producen ningún problema. Algunas de ellas son:

- **#pragma omp simd:** Transforma un bucle o función sencilla en vectorial, de modo que fuerza al compilador a vectorizarla. Esto hace que se puedan vectorizar bucles que no cumplían con las condiciones de la vectorización automática, sin embargo, el compilador puede inhibirlo.
- **#pragma vector always:** Indica al compilador que solo aplique las políticas de vectorización para datos altamente alineados.
- **#pragma vector alined:** Indica que el bucle se vectorice para datos alineados.
- **#pragma vector unalined:** Indica que el bucle se vectorice para datos no alineados.
- **#pragma vector nontemporal:** Indica al compilador que no almacene la variable en memoria caché. Se emplea si la variable no va a ser utilizada en un breve espacio de tiempo.
- **#pragma vector temporal:** Indica al compilador que no almacene la variable en memoria caché. Se emplea si la variable va a volver a ser utilizada en un breve espacio de tiempo.
- **#pragma novector:** Indicamos que el bucle nunca debe ser vectorizado.
- **#pragma ivdep:** Indicamos al compilador que ignore las supuestas dependencias del bucle.
- **#pragma loop_count:** Indicamos al compilador el número mínimo, máximo y medio de iteraciones del bucle, de la manera min(), max() y avg() respectivamente.

4º Vectorización por intrínsecas

- Este tipo de vectorización se basa en la especificación directa del uso de los registros y operaciones vectoriales por parte del programador, lo cual se hace mediante el empleo de intrínsecas de programación. Empleando esto, el compilador no necesitar predecir nada sobre la posible vectorización del programa, sino que todo viene expresado en el código.
- Para el empleo de las intrínsecas es necesario incluir la cabecera específica del tipo de arquitectura de nuestro procesador.

```
#include <xmmintrin.h> //(for SSE)
#include <emmintrin.h> //(for SSE2)
#include <pmmintrin.h> //(for SSE3)
#include <smmintrin.h> //(for SSE4)
```

- Se puede emplear diferentes tipos de datos, lo cual se especifica en el momento de la declaración de los mismos. Algunos de ellos son:
  - **__m128:** Datos empaquetados de flotantes de simple precisión.
  - **__m128d:** Datos empaquetados de flotantes de doble precisión.
  - **__m128i:** Datos empaquetados de enteros de 32 bits.
- Las operaciones intrínsecas tienen el formato ***_mm_"instruccion"_"sufijo"()***, de modo que estas se pueden operar con distintos tipos de datos. El tipo de dato a operar se expresa mediante el sufijo, el cual puede ser:
  - **ss:** Un único flotante.
  - **ps:** Un vector empaquetado de flotantes.
  - **sd:** Un único flotante de doble precisión.
  - **pd:** Un vector empaquetados de flotantes de doble precisión.
  - **si#:** Un vector de enteros (8, 16, 32, 64 o 128 bits).
  - **su#:** Un vector de enteros sin signo (8, 16, 32, 64 o 128 bits).
- La cantidad total de intrínsecas es enorme y por ello requiere consultarse las guías oficiales de Intel, sin embargo, algunas funciones interesantes puede ser:
  - **_mm_malloc():** Generamos una reserva de memoria alineada.
- Las operaciones vectoriales siempre operan llenando completamente los registros vectoriales que utilizan, de modo que si una CPU puede operar con un total de cuatro datos simultáneamente, todas las operaciones vectoriales trabajarán siempre con conjuntos de cuatro.
- En el caso de no querer obtener el resultado de los cuatro datos, podemos utilizar una máscara AND sobre el resultado, de modo que podamos especificar los datos que queremos preservar de los mismos.
- Si el vector con el que queremos operar no es múltiplo del número de datos con los que operan los registros vectoriales de la CPU, podemos realzar operaciones vectoriales y el remanente operarlo de forma secuencial.