ARQUITECTURA Y ORGANIZACIÓN DE COMPUTADORES

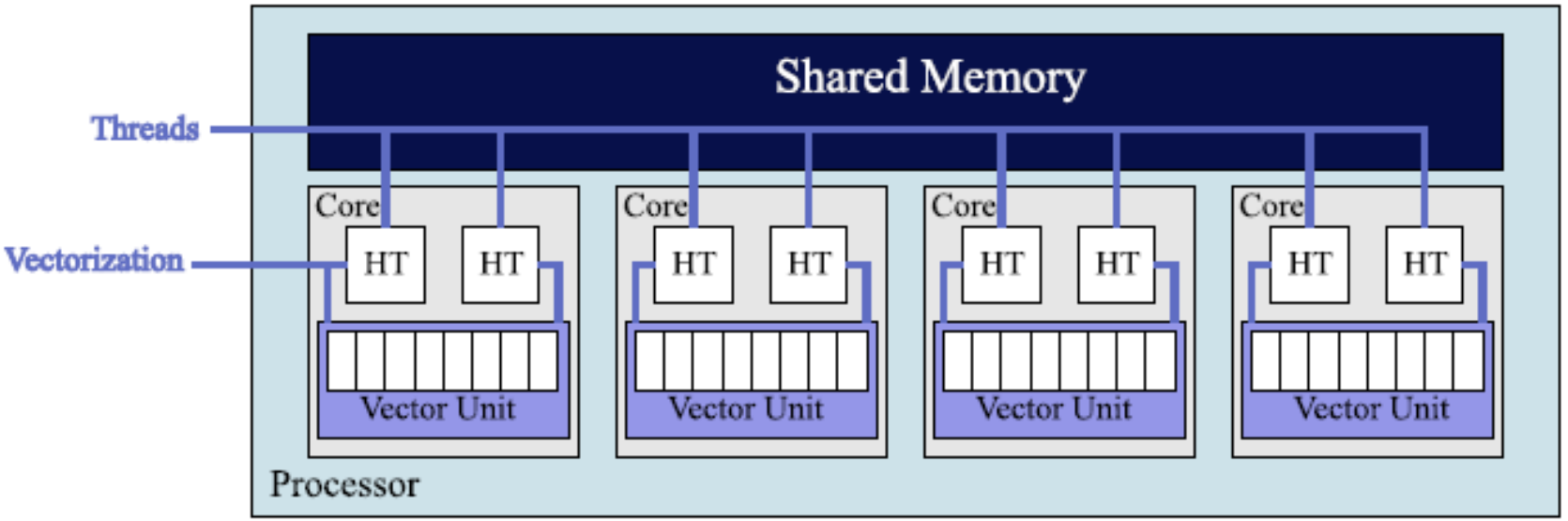
3er curso

Práctica 2.Procesadores CMP: Usando la vectorización y la paralelización

Departamento de Ingeniería y Tecnología de Computadores

Universidad de Murcia

Introducción: Vectorización y Paralelización

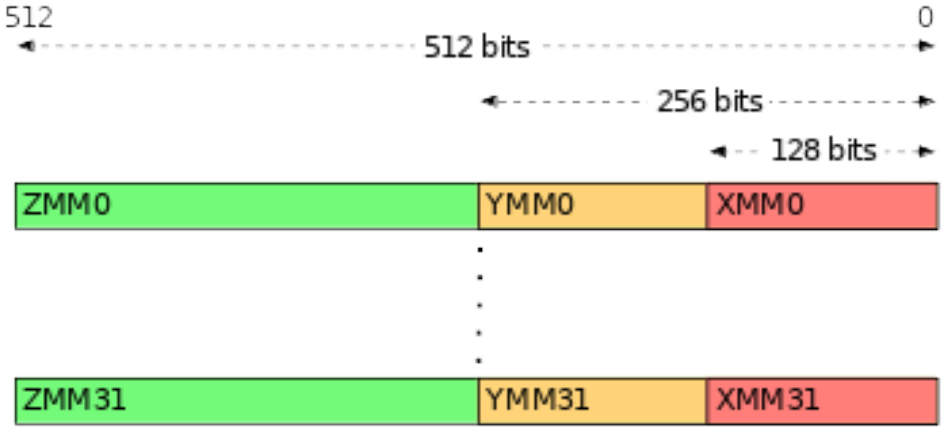


Explotar el paralelismo usando:

- Núcleos: Ejecutan múltiples hilos/procesos (MIMD)
- Vectores: Cada hilo emite instrucciones vectoriales (SIMD)

Introducción: Vectorización

- Paralelismo de tipo SIMD (*Single Instruction Multiple Data*)
- Extensiones vectoriales en x86:



SSE (128 bits)
AVX (256 bits)
AVX-512 (512 bits)

- ¿Cómo vectorizamos un código?
  1. Programando en ensamblador
  2. Utilizando los *intrinsics*
  3. Vectorización automática usando el compilador
  4. Usar una librería optimizada

Introducción: Paralelización

- Paralelizar consiste en dividir un programa en varios hilos, que se ejecutan de forma simultanea.
- En el caso ideal, podemos obtener un *speedup* proporcional al número de núcleos de la CPU
- Dependiendo de las características de la aplicación, el *speedup* gracias a la paralelización puede ser muy variable. Algunos factores son:
  - La memoria (caches, ancho de banda a memoria).
  - La sobrecarga introducida por las sincronizaciones entre hilos.
  - El grado de paralelismo de la aplicación (unas aplicaciones sacan más provecho de la paralelización que otras debido a su comportamiento).
  - ¡Cuidado con las condiciones de carrera!

Objetivos

• Vectorizar:

- Aprender a vectorizar con los compiladores de Intel (**icc**) y GNU (**gcc**)
- Aprender técnicas de vectorización automática, como la regularización de bucles, *hints* del compilador...

• Paralelizar:

- Aprender las características básicas de OpenMP
- Paralelización de tareas y de bucles

• Evaluar el rendimiento obtenido por las diferentes formas de paralelización

IMPORTANTE: Seguiremos usando la imagen de Docker de la práctica anterior

Vectorización automática (I)

En el directorio **vectorization/vectorization1** tenemos un programa que mide el rendimiento de una suma de dos arrays usando vectorización.

- Tenemos 4 implementaciones para realizar la suma de arrays:
  - 1. Mediante un bucle for (*sum_vectors_vectorized*)
  - 2. Igual que 1, pero usando una directiva para desactivar la vectorización (*sum_vectors_no_vectorized*)
  - 3. Igual que 1, indicando la palabra clave **restrict**, que avisar al compilador de que no existe *aliasing* entre los punteros (*sum_vectors_vectorized_restrict*).
  - 4. Igual que 1, indicando al compilador que los datos a los que apuntan los punteros están alineados, usando **std::assume_aligned** (*sum_vectors_vectorized_aligned*)

Rendimiento de la vectorización (I)

En la máquina **compiladorintel.inf.um.es**, usando el compilador gcc, y con un tamaño de array de 512 elementos, obtenemos:

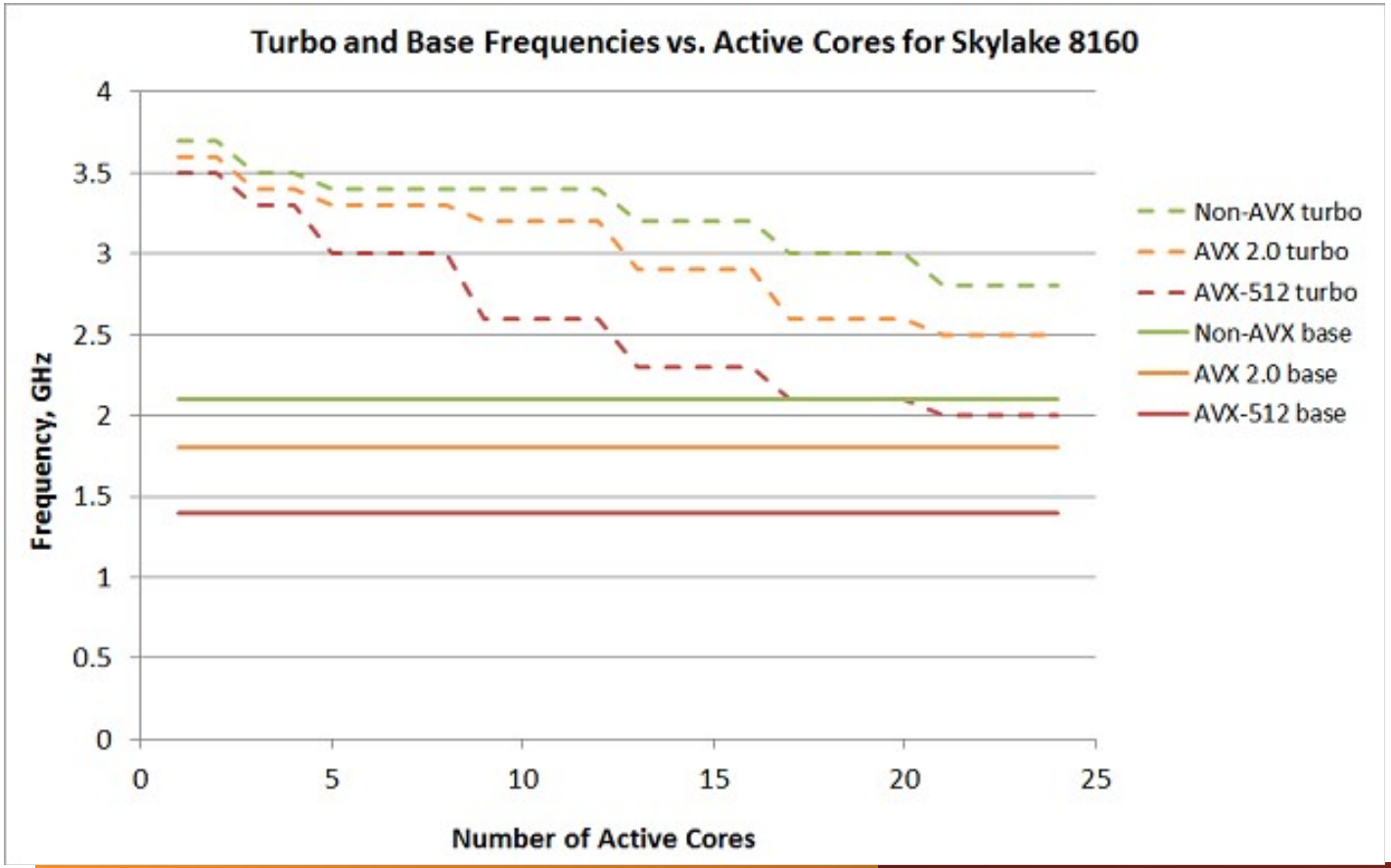
- Vectorización desactivada (*sum_vectors_no_vectorized*): **2.12 GFLOP/s**
- Vectorización activada (*sum_vectors_vectorized*): **15.5 GFLOP/s**

Gracias a la vectorización, obtenemos un *speedup* de **7.31x**, lo que nos da un **91.7%** de eficiencia vectorial (teniendo en cuenta que esta CPU soporta AVX y estamos trabajando en simple precisión).

Dependiendo del tamaño del array, el *speedup* obtenido gracias a la vectorización varía notablemente debido al tiempo de acceso a memoria.

Rendimiento de la vectorización (II)

Hay que tener en cuenta que al usar todos los núcleos y vectorizar, el procesador baja automática la frecuencia de los núcleos, por lo que el rendimiento obtenido es menor que el que cabría esperar al vectorizar.



Vectorización automática (II)

En el directorio **vectorization/vectorization2** tenemos un programa que suma dos arrays de 3 formas diferentes; 1) Usando un bucle, 2) Usando una función escalar que suma únicamente 2 elementos y 3) Usando una función que suma todos los elementos

1. Es interesante generar un informe de vectorización mediante el compilador, lo que nos ayudará a entender cómo ha vectorizado.
2. Podemos forzar al compilador a no vectorizar un bucle con **#pragma novector**

```
pablo@compiladorintel:~/materiales/vectorization$ make
icc -c -g -std=c++20 -Wall -O3 -restrict -xCORE-AVX2 -qopt-report -qopt-report-
file=vectorization.optrpt vectorization.cpp
icc -c -g -std=c++20 -Wall -O3 -restrict -xCORE-AVX2 -qopt-report -qopt-report-
file=worker.optrpt worker.cpp
icc -Wall -o vectorization-icc vectorization.o worker.o
pablo@compiladorintel:~/materiales/vectorization$ cat vectorization.optrpt
...
INLINE REPORT: (main()) [1] vectorization.cpp(22,11)

    Report from: Loop nest, Vector & Auto-parallelization optimizations [loop, vec, par]
...
```

Vectorización automática (III)

3. ¿Qué sucede si intentamos vectorizar un bucle en el que no se usan todos los elementos del vector? Uso de operaciones enmascardas (*masked*)

```
for(int i = 0 ; i < n ; i++) {
  if(i % 2 == 0) {
     b[i]= a[i] + b[i];
  }
}
```

4. Podemos vectorizar la llamada a una función mediante **#pragma simd** antes de la llamada a la función.

5. También es posible vectorizar una función añadiendo **__attribute__((vector))** delante de la declaración de la función.

Al hacer esto, el compilador usa *Multiversioning* para producir dos implementaciones; una vectorizada y una que no lo está, para el caso en el que haya **pointer aliasing**

Podemos evitar el *Multiversioning* con **#pragma ivdep**

Introducción a OpenMP

- Corregir programa **hello.cpp**, (directorio OpenMP-basics)

```
pablo@compiladorintel:~/materiales/OpenMP-basics$ make
g++ -std=c++20 -Wall -O3 -march=native -mtune=native -fopenmp
hello.cpp -o hello
icc -std=c++20 -Wall -O2 -xCORE-AVX2 -qopenmp hello.cpp -o hello-
icc
pablo@compiladorintel:~/materiales/OpenMP-basics$ ./hello
There are -1 available threads.
Hello world from thread -1
```

- Usar omp_get_max_threads() y omp_get_thread_num()
- Usar cláusula #pragma omp parallel
- Usar num_threads(N)
- Usar variable de entorno OMP_NUM_THREADS

Hello world OpenMP program

```
1  #include <omp.h>
2  #include <cstdio>
3
4  int main(){
5    // This code is executed by 1 thread
6    const int nt=omp_get_max_threads();
7    printf("OpenMP with %d threads\n", nt);
8
9  #pragma omp parallel
10   { // This code is executed in parallel
11     // by multiple threads
12     printf("Hello World from thread %d\n",
13                          omp_get_thread_num());
14   }
15 }
```

OpenMP: paralelismo de datos

- Trabajamos con **integral.cpp** (directorio OpenMP-reduction)

$$I(a,b) = \int_0^a \frac{1}{\sqrt{x}} dx$$

Rectangle method:

$$\begin{aligned} \Delta x &= \frac{a}{n}, \\ x_i &= (i+1)\Delta x, \\ I(a,b) &= \sum_{i=0}^{n-1} \frac{1}{\sqrt{x_i}} \Delta x + O(\Delta x). \end{aligned}$$

```
1  float Integrate(const float a,
2                  const int N) {
3    const float dx = a/float(n);
4    float S = 0.0f;
5    for (int i = 0; i < n; i++) {
6      const float xi = dx*float(i+1);
7      S += 1.0f/sqrtf(xi) * dx;
8    }
9    return S;
10 }
```

OpenMP: paralelismo de datos

- Obtenemos el resultado correcto pero ¿cómo paralelizamos este código?
- ¿Qué sucede si ponemos #pragma omp parallel?
- ¿Y con `#pragma omp parallel for`?
  - Encontramos una **condición de carrera** en la variable integral
- ¿Cómo solucionamos el problema?
  - 1º Opción: Utilizamos **`#pragma omp atomic`** y hacemos la reducción de forma manual.
  - 2º Opción: Utilizamos la cláusula de OpenMP **`reduction`** dentro del `#pragma omp parallel for` y OpenMP hará la reducción de forma automática.

OpenMP: paralelismo de tareas

- Trabajamos con **integral.cpp** (directorio OpenMP-tasks)

- Utilizamos **#pragma omp task** en la primera llamada recursiva
- Necesitamos la clausula **shared** para asegurar que la variable *child_integral* es compartida por las tareas
- Solucionamos la condición de carrera en la variable *integral* utilizando **#pragma omp taskwait**
- La creación de tareas debe estar dentro de una región paralela de OpenMp, por lo que la añadimos en el main.
- Para que la primera llamada sólo la ejecute un hilo tenemos que añadir además **#pragma omp master** o **#pragma omp single**

Rendimiento de la paralelización

En el directorio **OpenMP-time** tenemos los programas de la integral paralelizados utilizando los dos enfoques que hemos visto. Se han aumentado las iteraciones a 10e10. En la máquina **compiladorintel.inf.um.es**, usando el compilador de Intel y con 4 hilos, obtenemos:

- Paralelización por datos: 4.34 seg.
- Paralelización por tareas: 5.67 seg.

La aplicación secuencial tarda 15.32 seg. Por lo tanto, la paralelización por datos obtiene un *speedup* de 3.52x respecto a la aplicación secuencial y la paralelización por tareas un 2.70x, siendo en este caso la paralelización por datos un 30% más rápida que la de tareas.

Ejercicios pedidos (vectorización)

- En el directorio **vectorización** hay un programa que tiene 6 bucles:
- Para cada bucle del programa, identifica las dependencias entre iteraciones y explica cómo afectarían a una vectorización automática
- Compila el programa y genera el informe de vectorización (usando el Makefile). Analiza el informe y explica, para cada bucle, por qué ha sido vectorizado o por qué no se ha podido vectorizar. Para los bucles en los que se hubieran detectado dependencias entre iteraciones en el punto anterior:
  - Si ha sido vectorizado, explica qué transformaciones ha aplicado automáticamente el compilador.
  - Si no ha sido vectorizado, explica si crees que hay alguna transformación manual posible para que se pueda vectorizar

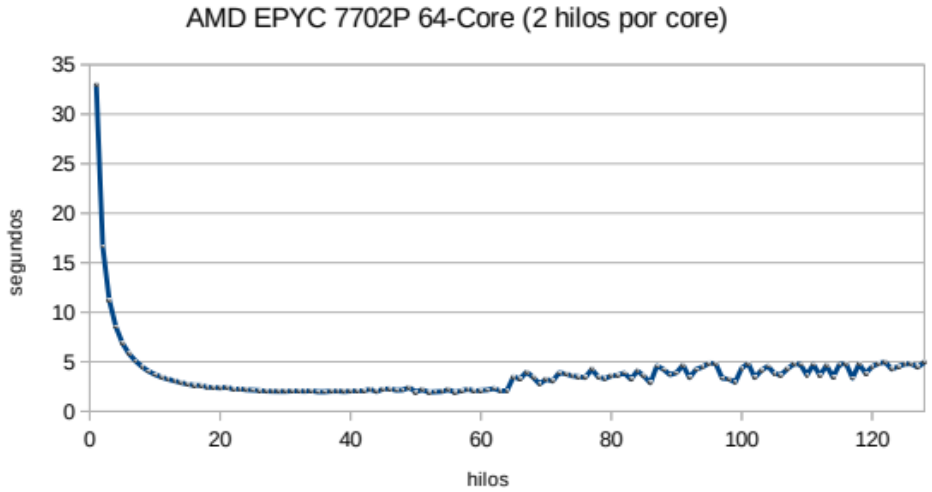
Ejercicios pedidos (paralelización)

En el directorio **paralelización** hay un programa que simula la propagación del calor en una superficie rectangular. De los 3 ficheros, solo es necesario modificar **heat.cpp**. Se pide:

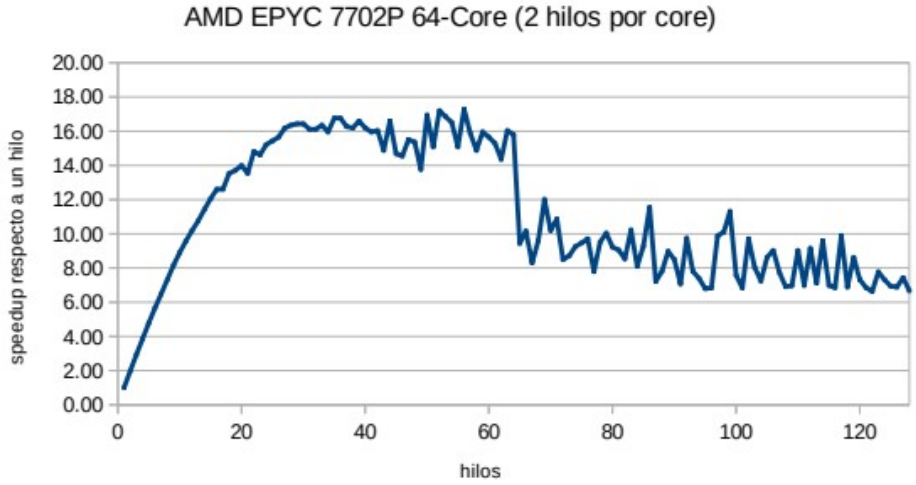
- El procedimiento **solve** de **heat.cpp** tiene 3 bucles. Para cada uno de ellos, explica si es candidato para ser paralelizado usando OpenMP. Indica en cada caso qué *pragma* (o *pragmas*) sería necesario añadir.
- Prueba a paralelizar cada uno de los bucles paralelizables por separado. Prueba también a paralelizar todos los bucles paralelizables a la vez. Para cada versión resultante del programa, mide el tiempo de ejecución variando el número de hilos de cada uno de los 3 casos de prueba. Representa los resultados en una gráfica, mostrando los tiempos de ejecución y la escalabilidad obtenida (similar a las figuras de la siguiente diapositiva). Comenta los resultados.

Se recomienda el uso de un ordenador que tenga 4 cores o más

Ejercicios pedidos (paralelización)



Evaluación del tiempo de ejecución de una aplicación en función del N.º de hilos



Evaluación de la escalabilidad de una aplicación en función del N.º de hilos

Entrega

- Tras finalizar los ejercicios anteriores, se pide entregar lo siguiente:
  - Un documento en formato PDF que será el documento principal utilizado a la hora de evaluar la práctica. Dicho fichero debe incluir:
    - Información sobre la autoría de la práctica.
    - Las respuestas a los ejercicios solicitados.
    - Un breve informe comentando los aspectos positivos de la práctica, así como cualquier aspecto negativo y cosas que has echado en falta en la misma. La longitud del informe no debe exceder las 2000 palabras
  - El código fuente de cualquier programa o script desarrollado (o modificado) para la realización de la práctica. Estos ficheros se mirarán opcionalmente para comprobar cualquier aspecto que no esté claro en el documento PDF.
  - Un fichero de texto llamado README identificando todos los ficheros fuente incluidos con instrucciones claras para su compilación.

Criterios de evaluación

- Esta práctica se evaluará teniendo en cuenta los siguientes criterios de evaluación (sobre 10 puntos):
  - Realización y corrección de los apartados pedidos (6 puntos)
  - Presentación y claridad de las explicaciones (1 punto)
  - Aportación de ideas originales en las explicaciones a los apartados realizados (1 punto)
  - Realización de alguna actividad extra relacionada con los ejercicios de la práctica demostrando curiosidad (1 punto)
  - Comentarios sobre la práctica, incluyendo aspectos negativos y positivos (1 punto)

ARQUITECTURA Y ORGANIZACIÓN DE COMPUTADORES

3er curso

Práctica 2.Procesadores CMP: Usando la vectorización y la paralelización

Departamento de Ingeniería y Tecnología de Computadores

Universidad de Murcia