CAP – Propuesta de examen

I. Un programa de cálculo numérico se ha instrumentalizado mediante la herramienta gprof y el resultado obtenido después de ejecutarlo ha sido el siguiente:

Flat profile:

Each sample counts as 0.04 seconds

% time	acumulative seconds	self seconds	calls	self ms/call	total ms/call	name
54,19	170,52	170,52	12	14210	14210	suma
30,79	267,42	96,9	45	2153,44	2153,44	multi
13,87	311,08	43,66	1	43655	43655	coseno
1,15	314,69	3,61	66	54,71	54,71	resto

1. ¿Qué procedimientos elegiría para mejorar el rendimiento? Justifique su respuesta.

Teniendo en cuenta que el análisis se ha desarrollado sobre las cuatro funciones indicadas, y careciendo de información más concreta sobre las mismas que la que nos proporcionada la tabla (datos tales como el código o la relación de dichas funciones entre sí). Suponemos que la ejecución de dichas funciones se realiza de manera independiente las unas de las otras.

Cunado dos funciones no mantienen dependencias ni en los datos que utilizan ni en las instrucciones que ejecutan, esto quiere decir que ambas pueden ser optimizadas de manera independiente. En base a esto, analizaremos una a una cada una de las cuatro funciones, para lo cual nos apoyaremos en los datos referentes al número de llamadas y los tiempos de ejecución.

- **Función suma:** La función se esta ejecutando la mitad tiempo de ejecución y únicamente es llamada 12 veces. Esto nos indica que se trata de una función muy pesada de ejecutar, la cual seguramente requiera la ejecución de un alto número de operaciones (tales como operar con vectores) o necesite realizar accesos a disco.
  - Suponiendo que se trata de una operación de suma con grandes vectores, una de las opciones más óptimas seria utilizar una combinación del uso de los operadores SIMD de la CPU junto a la operación en paralelo de las diferentes sumas que conforman la suma de vectores. Suponiendo un juego de vectorización AVX con 8 elementos simultáneos y un procesador de 8 núcleos, el speedup total podría llegar a ser de 16x.
  - Por otra parte, si el equipo donde estamos realizando las operaciones cuenta con un acelerador gráfico lo suficientemente potente, otra opción optima sería descargar el espacio de datos necesario para la ejecución de las distintas llamadas de la función suma en el acelerador y dejar que este realizara la ejecución.
  - Cual de las dos opciones sería más eficiente a la hora de aumentar el speedup dependerá, sobre todo, de si el tiempo necesario para transmitir los datos al acelerador gráfico es lo suficientemente pequeño como para que su ejecución en el mismo sea más rentable que el uso de la CPU (SIMD + multihilo).
- **Función multiplicación:** Se trata de la segunda función que más tiempo de computo consume (el cual sigue siendo un porcentaje muy representativo) y se llama un total de 45 veces, las cuales son más del triple que las que se llamaba la función suma. Esto quiere decir

que la ejecución unitaria de la función multiplicación es mucho más liviana que la de la anterior función.

  - Partiendo de este criterio podemos desestimar el uso de un acelerador gráfico, puesto que si cada vez que se llama a la función necesitamos copiar los datos en la memoria del dispositivo, lo más seguro es que este tiempo sea mucho mayor que el empleado si ejecutáramos la función en la CPU de la máquina.

  - Suponiendo que se pueda trata de una función que multiplica dos matrices, podemos emplear al igual que en la función anterior, una combinación del uso de las instrucciones SIMD del procesador con una ejecución multihilo. Volviendo a suponer el uso del juego de instrucciones AVX2 con 8 elementos simultáneos y 8 núcleos, obtendríamos un speedup máximo de 16x, lo que supondría una reducción aproximada de 267s a 16s.

- **Función coseno:** Aunque únicamente representa el 13% del tiempo de cómputo, esta función es únicamente llamada una vez, lo que lo convierte en la función con coste computacional de todas.

  - Teniendo en cuenta esto, y suponiendo que se trata de una función que ejecuta la función conseno sobre una lista de elementos, la opción más lógica es emplear el uso de un acelerador. Esto es debido a que únicamente necesitaremos copiar la información en la memoria del acelerador una vez (ya que no se ejecuta más veces) y sin embargo su coste de computo puede llegar a reducirse enormemente, siendo más eficiente que si lo ejecutáramos en la CPU del equipo.

- **Función resto:** La función representa una parte tan pequeña del tiempo de cómputo y es llamada tantas veces que lo más seguro es que no fuera rentable aplicar técnicas que mejoren el rendimiento.

2. ¿Qué sería más beneficioso a priori suponiendo aceleraciones ideales los casos propuestos?

a) Usar las instrucciones SSE para optimizar la rutina suma si se utilizasen datos flotante sencillos.

El juego de instrucciones SSE utiliza datos en punto flotante sencillos de 32 bits, sin embargo, debido que se trata de un juego más antiguo, el tamaño del vector es de 128 bits, lo que supone que únicamente opera con cuatro datos simultáneos.

Esto conlleva a que el speedup máximo obtenido sería de 4x, aunque teniendo en cuenta que se trata de la función con mayor carga computacional, esto reduciría el tiempo de cómputo aproximado de 170s a 42s (reducción de 120s).

b) Usar las instrucciones vectoriales AVX para optimizar la rutina multi si se utilizasen datos en punto flotante sencillos.

El juego de instrucciones SSE utiliza datos en punto flotante sencillos de 32 bits y tiene un tamaño de vector de 256 bits, de modo que puede operar simultáneamente hasta con 8 datos, pudiendo tener un speedup máximo de hasta 8x.

Teniendo en cuenta que se trata de una función que abarca el 30% del cómputo total, la reducción aproxima del tiempo de cómputo sería de 96s a 12s (reducción de 84s).

c) Paralelizar las rutinas suma y multi con OpenMP en un multicore de 2 cores

Paralelizar las rutinas conlleva que ambas se ejecuten al mismo tiempo, sin embargo, teniendo en cuenta que la función suma abarca mucho más tiempo que la función multiplicación, esto llevaría a que la segunda terminaría de ejecutarse mucho antes que la primera.

En base a esto, el tiempo de computo será el valor maximizo de entre las dos funciones, es decir, los 170s de la función suma, pudiendo solapar los 96 segundos de la función multiplicación. Debido a esto no se podría obtener el speedup ideal de 2x.

Conclusión: Teniendo en cuenta el tiempo ahorrado de forma teórica por los tres métodos, podemos deducir que el que más tiempo de ejecución reduce es la optimización de la función suma mediante el juego de instrucciones SSE.

II. De acuerdo a los siguientes códigos y suponiendo que los arrays se encuentran almacenados en la memoria principal y la variables puede almacenarse en un registro, responder a las siguientes cuestiones:

1A	1B	1C	1D

`double a[], b[];`
`for(i=0; i<N; ++i)`
`  a[i] = a[i] + b[i];`
	`double a[], b[];`
	`for(i=0; i<N; ++i)`
	`  a[i] = a[i]+ s * b[i];`
		`float s=0, a[];`
		`for(i=0; i<N; ++i)`
		`  s = s + a[i] * a[i];`
			`float s=0, a[], b[];`
			`for(i=0; i<N; ++i)`
			`  s = s + a[i] * b[i];`

1. ¿Cual es la intensidad aritmética de cada uno de los códigos? ¿Alguno de los códigos es memory-bound o compute-bound?

El cálculo de la intensidad aritmética (I) de un kernel viene dada por la relación entre el número de operaciones aritméticas que este debe llevar a cabo (Q) y el tráfico en memoria que debe desarrollarse para poder realizar dichas operaciones (W). Esto lo podemos indicar como: **I= W/Q.**

Teniendo en cuenta esta relación, podemos especificar que si el valor de I es mayor que uno, la intensidad aritmética recae en mayor porcentaje sobre las operaciones de computo, luego podríamos clasificar el kernel como *compute-bound*.

En el caso contrario, si el valor de I es mor que uno, entonces hay un mayor coste asociado al tráfico en memoria, luego podríamos clasificar el kernel como memory-*bound*.

Debido a que el enunciado nos indica que las variables escalares se encuentran almacenadas en registros, asumiremos que el coste de tráfico en memoria desarrollado por las mismas (tanto si leen como si se escriben) es despreciable.

Kernel 1A

- Operaciones aritméticas: 1 ADD.
- Operaciones de tráfico en memoria: 2 LOAD (8 bytes), 1 STORE (8 bytes)
- *I = 1 / (2*8+8) = 1/24. Como I<1, el kernel es memory-bound.*

Kernel 1B

- Operaciones aritméticas: 1 ADD, 1 MUL

- Operaciones de tráfico en memoria: 2 LOAD (8 bytes), 1 STORE (8 bytes)
- *I = (1+1) / (2*8+8) = 2/24 = 1/12. Como I<1, el kernel es memory-bound.*

Kernel 1C

- Operaciones aritméticas: 1 ADD, 1 MUL
- Operaciones de tráfico en memoria: 1 LOAD (4 bytes)
- *I = (1+1) / 4 = 2/4 = 1/2. Como I<1, el kernel es memory-bound.*

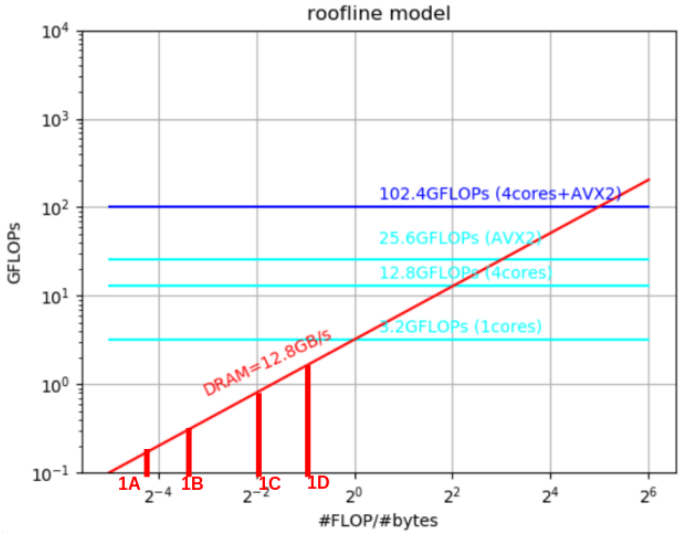
Kernel 1D

- Operaciones aritméticas: 1 ADD, 1 MUL
- Operaciones de tráfico en memoria: 2 LOAD (4 bytes)
- *I = (1+1) / (4+4) = 2/8 = 1/4. Como I<1, el kernel es memory-bound.*

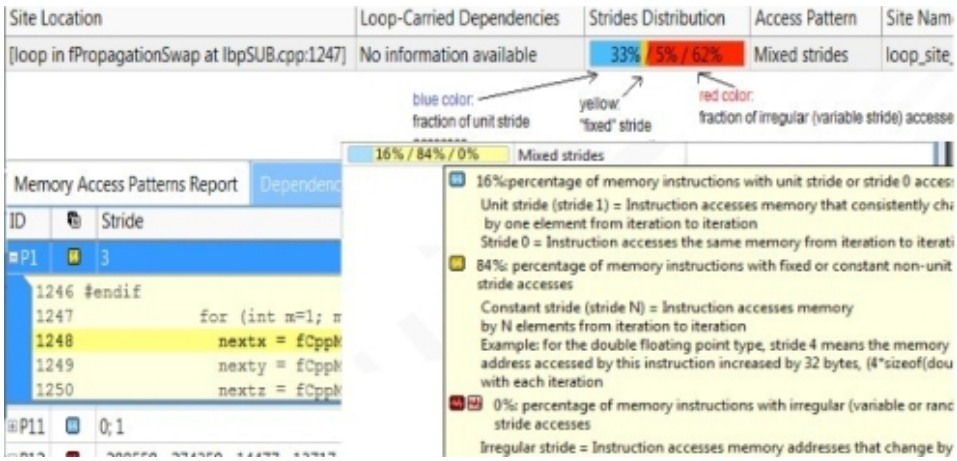
2. De acuerdo al modelo roofline de la máquina disponible dibujar el rendimiento máximo esperable para los códigos 1A, 1B, 1C y 1D con un único core y sin explotar las capacidades SIMD.

Segun los resultados obtenidos en el apartado anterior:

- **Kernel 1A:** I = 1/24; $2^{-4} < I < 2^{-5}$
- **Kernel 1B:** I = 1/12; $2^{-3} < I < 2^{-4}$
- **Kernel 1C:** I = 1/2; $I = 2^{-1}$
- **Kernel 1D:** I = 1/4; $I = 2^{-4}$



III. Evaluando un código desarrollado observamos que no escala como esperamos cuando se hace uso del paralelismo SIMD. Por este motivo, se evalúa su rendimiento mediante la herramienta del Intel Advisor, y poniendo el foco en los patrones de acceso a memoria se observa el siguiente resultados para la rutina *fPropagationSwap*.



El código de colores de la barra Strides Distribution, corresponde al acceso con stride=1 (color azul), stride fijo (amarillo) y stride variable (barra roja).

De acuerdo a este análisis, explicar razonadamente porque el rendimiento de la aplicación no alcanza la aceleración esperada motivada por los patrones de acceso a memoria. Indicar el motivo del overhead o sobrecoste producido de acuerdo a la medida observada en el Intel Advisor.

Para comprender las causas del bajo rendimiento, primero debemos entender que el *stride* hace referencia al acceso a memoria que realiza el computador cuando necesita acceder a la variable que se encuentra en la misma (ya sea para realizar una operación de lectura y escritura). Concretamente, el valor del *stride* nos indica el número de posiciones que el equipo debe recorrer a la hora de realzar dos accesos consecutivos.

Teniendo en cuenta esto, cuando queremos realizar operaciones que requieren de una alta cantidad de accesos a memoria, como puede ser trabajar con vectores o matrices, el valor del *stride* es muy relevante en el rendimiento.

Cunado nos encontramos ante un *stride* con valor 1, indica que los accesos de los diferentes elementos consecutivos en memoria se han producido sin la necesidad de que el computado recorra más posiciones en la misma, es decir, que dichos elementos se encontraban alineados en la memoria uno de detrás de otro (por ejemplo, los elementos de un array de longitud fija). Este se trata del caso más deseable, puesto que el equipo puede obtener elementos de manera simultánea, aumentando enormemente la eficiencia.

En el caso de tener un *stride* fijo y diferente de 1, quiere decir que los elementos no se encuentran de forma consecutiva en la memoria, pero que la distancia entre cada uno es siempre la misma (por ejemplo, el acceso al mismo elemento dentro de un array de estructuras). En este caso el rendimiento se ve ampliamente mermado, puesto que el equipo no puede obtener elementos de manera simultánea.

Finalmente, en el caso de que se produzca un *stride* variable, esto nos indica que no solamente los elementos no se encuentran alineados en memoria, sino que la distancia entre dos de estos elementos es siempre diferente (por ejemplo, los elementos de una lista que se van insertando). Nos encontramos ante una situación que reduce drásticamente el rendimiento.

Como conclusiones podemos razonar que el principal motivo por la escasa escalada en el rendimiento de la aplicación don los accesos desordenados a memoria, y esto es debido a que en el momento de declarar los vectores o matrices utilizados no se ha especificado que se realice de forma ordenada, de modo que el sistema ha asignado los diferentes espacios de memoria sin tener en cuenta esto.

Para mejorar el rendimiento necesitaremos definir los deferentes vectores de manera alineada, utilizando funciones especiales para ello como puede ser *_mm_malloc()*. Además de esto, y teniendo en cuenta que hay un porcentaje del análisis donde nos indica que el *stride* es fijo (lo cual nos indica que seguramente se acceda a los diferentes elementos de una lista de estructuras), deberemos organizar los datos en forma de SoA, haciendo que los accesos entre los diferentes vectores también se produzcan de manera alineada.

IV. Añadir la líneas de código correspondientes a las directivas para desarrollar una implementación de OpenMP (target) de la multiplicación de matrices

Para la realización del presente ejercicio tomamos como referencia el código de multiplicación de matrices que indicado en el ejercicio VI.

Teniendo en cuenta que dicho código únicamente abarca las operaciones aritméticas necesarias para el cálculo de la multiplicación de matrices y pidiendo el ejercicio que utilicemos las pragmas *target* asociadas al uso de dispositivos, lo mejor será utilizar dicho código como el *kernel* a descargar en el dispositivo que vamos a utilizar.

```
#pragma omp target enter data map(to: A[0:Ndim], B[0:Mdim])  map(tofrom: C[0:Ndim])

#pragma omp target teams distribute parallel for collapse(2) private(i, j, k) schedule(static)
for (i=0; i<Ndim; i++){
   for (j=0; j<Mdim; j++){
      tmp = 0.0;
      for(k=0;k<Pdim;k++){
         tmp += *(A+(i*Ndim+k)) * *(B+(k*Pdim+j));      //C(i,j) = sum(over k) A(i,k) * B(k,j)
      }
      *(C+(i*Ndim+j)) = tmp;
   }
}

#pragma omp target enter data map(from: C[0:Ndim])
```

La utilización de las pragmas *omp target enter data* y *omp target exit data* nos permite definir el ámbito en el que los datos serán utilizados por el dispositivo, de modo que cuando este termine, los datos indicados se retornarán a la host y los restantes serán reciclados.

Por otra parte, con la pragma *omp target teams distribute parallel for* nos permite descargar el kernel que realiza las operaciones en el dispositivo. Además, le indicamos que cree los equipos de ejecución correspondientes, distribuimos las iteraciones entre los mismos y cada uno de ellos ejecutará un hilo de ejecución.

De esta manera estamos haciendo que el dispositivo aproveche sus capacidades de paralelismo utilizando varios equipos e hilos con los que ejecutar las diferentes iteraciones del bucle for.

VI. La versión secuencial del código siguiente calcula la multiplicación de matrices de A y B. Se desea realizar una implementación paralela con el paradigma de programación paralelo OpenMP.

```
#pragma omp parallel for \
  schedule(guided) collapse(2) \
  private(i, j, k, tmp) shared(C)
for (i=0; i<Ndim; i++){
   for (j=0; j<Mdim; j++){
      tmp = 0.0;
      for(k=0;k<Pdim;k++){
         tmp += *(A+(i*Ndim+k)) * *(B+(k*Pdim+j));      //C(i,j) = sum(over k) A(i,k) * B(k,j)
      }
      *(C+(i*Ndim+j)) = tmp;
   }
}
```

Podemos ver como el código a ejecutar esta compuesto por tres bucles for anidados, donde hemos utilizado una pragma. La primera de ellas se encuentra en el exterior de los bucles y su función crear una sección paralela que ejecute todas la iteraciones del bucle for externo.

En adición, esta pragma cuenta con la clausula *collapse*, la cual aplica el desenrollado de bucles a los dos bucles externos, haciendo que se comporten como si únicamente fueran uno. Por otra parte, la clausula *schedule* nos permite indicar como se debe realizar la distribución de las ejecuciones, la cual hemos indicado que sea guiada, con el objetivo de que se produzca un menor desbalanceo de carga a medida que se queden menos iteraciones por realizar.

Indicar que no hemos empleado vectorización debido a que las instrucciones donde realizamos el cálculo con los diferentes vectores se guardan o operan con escalares, lo cual rompe el esquema de vectorización al no poder almacenar los resultados correspondientes en espacios de memoria consecutivos.