UNIVERSIDAD DE OVIEDO
E.P. INGENIERÍA DE GIJÓN

Programación Concurrente y Paralela
15 de diciembre de 2022, 11:00

Normas del examen:

- Tiempo total: **110 minutos.**
- Tiene dos partes, y cada parte 2 ejercicios. Contestar a cada ejercicio en los cuadros preparados para ello.

Datos personales:

Apellidos: Firma:

Nombre:

DNI/Pasaporte:

Parte 1: OpenMP (5 p)

Ejercicio 1 (2,5 p)

Partiendo del siguiente código:

```
void main(){
#pragma omp parallel sections num_threads(2)
        {
                printf("%d\n",omp_get_thread_num());
                #pragma omp section
                #pragma omp parallel
                printf("-%d\n",omp_get_thread_num());
        }
}
```

Sólo 2 hilos. Anidamiento por defecto desactivado

Primera sección, implícita (solo un hilo)

Parallel que solo renumera los 2 hilos que hay

Ambas secciones corren concurrentemente

Escribir TODAS las posibles combinaciones de salida que dicho código puede producir en su ejecución, según la especificación formal de OpenMP.

Respuesta:

Opción 1:	Opción 2:	Opción 3:	Opción 4:
0	1	-0	-0
-0	-0	0	1

Ejercicio 2 (2,5 p)

Se dispone de un vector de enteros de longitud N que ha sido inicializado con distintos números. Se pide implementar, haciendo uso de OpenMP, una función paralela que devuelva cuántos números impiden que dicho vector sea capicua. Por ejemplo:

[1 2 3 4 5 4 3 2 1]	La función debe devolver 0, pues es capicua.
[0 1 3 4 4 3 2 1]	La función debe devolver 2, pues dos números impiden que sea capicua. Dichos números han sido marcados en rojo para mayor claridad.

Respuesta:

```
int capicua(int* v, int N){
	int i;
	int l=floor(N/2);
	int num=0;
	#pragma omp parallel for
	for(i=0;i<l;i++)
		if(v[i] != v[N-i-1])
			#pragma omp atomic
			num+=1;
	return num;
}
```

También se da por buena la solución:

```
int capicua(int* v, int N){
	int i;
	int l=floor(N/2);
	int num=0;
	#pragma omp parallel for reduction(+:num)
	for(i=0;i<l;i++)
		if(v[i] != v[N-i-1])
			num+=1;
	return num;
}
```

Parte 2: CUDA (5 p)

Ejercicio 3 (2,5 p)

Dado un vector *v* de 1984 elementos enteros, se quiere mapear sobre él la función $f(x)=x^x$ con x natural usando una GPU, de forma que, tras el procesado, *v[0]=0, v[1]=1, v[2]=2, v[3]=27....* Hacer un kernel sin saltos divergentes que permita realizar este cálculo y dar la configuración de ejecución que lo hace funcionar.

Respuesta:

```
__global__ void kernel(int* v){
        int i=threadIdx.x+blockIdx.x*blockDim.x;
        v[i]=pow(i,i);
        if(i==0) v[i]=0;
}
```

Estrictamente hablando, pow está definido como pow(double, int), por lo que su llamada debería ser:
v[i]=pow((double)i,i);
No obstante, se da por bueno el primer código.
No se contarán como saltos divergentes la gestión de la posición v[0] cuando se haya procesado específicamente. Se da por buena también si no se ha procesado la posición 0.

También se da por bueno el código que lo calcule con un bucle for

Llamada al kernel:

```
kernel<<<62,32>>>(v);
```

O bien:

```
kernel<<<31,64>>>(v);
```

Ejercicio 4 (2,5 p)

Pasar el Ejercicio 2 a CUDA. Como el kernel no puede devolver ninguna variable, este valor será pasado por referencia (llamado *resultado*), quedando la cabecera del kernel como sigue:

```
__global__ void capicua(int* v, int N, int* resultado);
```

Dar, asímismo, la configuración de ejecución necesaria para un tamaño arbitrario N del vector.

Soluciones no coalescentes contarán sobre 1 punto.

Respuesta:

```
__global__ void capicua(int* v, int N, int* resultado){
        int bd=blockDim.x;
        int i=threadIdx.x+blockIdx.x*bd;
        int l=(N/2);
        extern __shared__ int vs[];
        if(i<l){
        vs[i]=v[N-(blockIdx.x+1)*bd+i];
        __syncthreads();
        if(v[i] != vs[bd-i-1])
                atomicAdd(&(num[0]),1);
        }
}

 numBlocks = (N/2 + ThPerBlk - 1) / ThPerBlk;
 capicua<<<numBlocks, ThPerBlk, ThPerBlk*sizeof(int)>>>(Devi_x, N, resultado);
```

Notas:
- Basta con la mitad de hilos pues cada uno debe comparar dos números.
- El uso de shared permite leer siempre coalescentemente la memoria principal. De lo contrario, las lecturas sobre la memoria principal con índices [N-…] rompen la coalescencia.
- Hay que cuidar que funciona para cualquier número de bloques.
- El uso de atomicAdd elimina la condición de carrera.