MEMORIA PRAC05

Ejercicio 1:

2. Ejecutar “./VecAdd1 –2000 32 2000 21”. No olvidar el signo menos del primer 2000.
¿Qué observa?

> En CPU directamente no entre por la dimension negativa. En el caso de CUDA Nos da un tiempo pero no hace nada (no entra al kernel), pero no lo sabemos porque no hemos hecho CHEKNULL.

3. Edite VecAdd1. Busque el comentario “Paso 1 º”. Borre las 4 líneas que siguen al comentario “Paso 1 º” y quite los comentarios a las 4 líneas que siguen a las borradas. Compilar de igual forma que en el punto 1. Ejecutar en las mismas condiciones que en el punto 2. ¿Qué observa?

> Da error a la hora de reservar memoria, ya que ahora vemos con CHECKNULL que la dimension pasada es negativa (no sirve).

4. Ahora ejecutar “./VecAdd1 2000 32 2000 21”. ¿Qué observa?

> Ahora da un tiempo mayor en la CPU y menor en el kernel CUDA con un error de 1,07E+00 (bastante grande) pero ejecuta en ambos.

5. Edite VecAdd1. Busque el comentario “Paso 2 º”. Quite el comentario para habilitar las 7 líneas que le siguen. Compilar de igual forma que en el punto 1 y ejecutar en las mismas condiciones que en el punto 4. ¿Qué observa?

> La version de CUDA del equipo es insuficiente para ejecutar la orden `CUDAERR(cudaGetDeviceCount(&ndev));`

6. Crear un trabajo y ejecutar en las mismas condiciones que el punto 5. ¿Qué observa?

> Ahora el error es 0, y los tiempos obtenidos en la version CPU son menores que los de CUDA, esto nos muestra que es importante la implementacion de CUDA para sacarle partido, en este caso estamos siendo menos eficientes que usando la CPU sin más.

7. Cambie el script para ejecutar “VecAdd1 2000 2048 2000 21”. ¿Qué observa?

> Nos da un error de configuracion debido a que 2048 es un número demasiado grande de hilos por cada bloque.

8. Busque en VecAdd1 el resto de comentarios del estilo “Paso x”. Quite los comentarios a la (las) línea (líneas) que le siguen y borre, si existen, la siguientes que hacen lo mismo sin control de errores. Ejecute. ¿Qué observa?

De nuevo, el número de hilos por bloque es demasiado grande y nos sale un error grande, pese a que el código fuente funcione. El problema está en el número de hilos por bloque que se le pasa como parámetro.

9. Prepare un trabajo con las siguientes órdenes de ejecución “VecAdd1 10000 4 1000
2121” y “VecAdd1 1000000 1024 100 2121”. ¿Qué observa?

Ahora cambiamos el número de hilos por bloque a 4 y 1024. En el primer caso los tiempo de CUDA son aproximadamente 5 veces mayores que los de la CPU, CUDA no esta siendo la más óptima ya que 4 hilos por bloque tal vez sean muy pocos (hemos pasado de un extremo a otro). En el segundo caso esto se invierte, y los tiempos de la CPU son 5 o 6 veces mayores que los de CUDA, ahora si que estamos optimizando los tiempos mucho con CUDA pasandole un mejor valor de hilos por bloque (1024)

Ejercicio 2:

1. Preparar un trabajo y ejecutar “VecAdd2 1000000 1024 100 2121”. ¿Qué observa en los tiempos obtenidos?

Obtenemos dos tiempos para la CPU y dos tiempos para CUDA. Basicamente utiliza dos funciones de CUDA para medir los tiempos.

Ejercicio 3:

1. Preparar un trabajo y ejecutar “VecAdd3 1000000 1024 100 2121”. ¿Qué observa en lostiempos obtenidos?

De nuevo nos da un error de configuración, indicando que 1024 son demasiados hilos por bloque para el caso de la GPU2D. Esto no se debe a que 1024 en si sean demasiados, sino a que en la llamada de GPU2D, se pasa como parámetro de hilos por bloque 1024*1024, que si se pasa. Por eso solo falla en GPU2D, ya que para las otras dos implementaciones el número de hilos por bloque sigue siendo 1024, que entra dentro del rango.

2. Ahora ejecute el trabajo con la orden “VecAdd3 1000000 32 100 2121”. ¿Qué observa en los tiempos obtenidos?

En este caso el argumento de hilos por bloque es válido y ejecuta sin problema ya que para GPU2D, el número de hilos por bloque es 32+32 = 1024 (entra dentro del rango). Respecto a los tiempos obtenidos, el más lento es la CPU. La GPU1D es mas eficiente y obtiene unos tiempos 5 veces menores que su anterior. La GPU2D es la implementación mas eficiente, obteniendo un orden de magnitud menor.

3. Ahora ejecute el trabajo con la orden “VecAdd3 1000000 16 100 2121”. ¿Qué observa
en los tiempos obtenidos?

Los tiempos son practicamente iguales a la version anterior en el caso de la CPU y de la GPU2D. En el caso de la GPU1D, esta nueva implementacion da unos tiempos mayores. Esto se debe a que los hilos se reparte de 32 en 32. De esta forma, a 2D le da igual 16 hilos por bloque que 32 ya que para su llamada usara 16*16 y 32*32 hilos, y podra repartirlos de forma correcta en grupos de 32. En el caso de 1D es diferente ya que en su llamada usa el número de hilos que le pasas como argumento directamente, de forma que si le pasas 32 podra repartirlos de la forma correcta (de 32 en 32), pero si le pasas 16, pasará grupos de hilos de 16 en 16, de forma que la mitad del tiempo el hilo estará en desuso (ya que 16=32/2).

Ejercicio 4:

1. ¿Qué ha cambiado?

Cambia la forma de reservar y liberar memoria.

2. Preparar un trabajo y ejecutar “VecAdd4 1000000 1024 100 2121”. ¿Qué observa en los
tiempos obtenidos?

Los tiempos resptecto a la version anterior (con la forma anterior de reservar y liberar memoria) son un orden de magnitud mayores. Además hay menos diferencia entre la version CPU y GPU.

Ejercicio 5:

1. ¿Qué ha cambiado?

De nuevo es otra forma más de reservar y liberar memoria distinta a las 2 anteriores.

2. Preparar un trabajo y ejecutar “VecAdd5 1000000 1024 100 2121”. ¿Qué observa en los
tiempos obtenidos?

En este caso se mejoran los tiempos respecto al ejercicio 4. Comparandolo con la primera forma, vemos que en CPU da unos tiempos parecidos (ligeramente menores en este caso) y en GPU da tiempos el doble de grandes (esto se debe a que en la primera version no entra la parte de reserva de memoria en el contador y en esta si, en realidad deberia dar tiempos muy parecidos).

Ejercicio 6:

6.1:

```
__global__ void kernel6_1(double *v, const double *x, const double *y, const int size){

int tid = blockIdx.x * blockDim.x + threadIdx.x;

if(tid<size){
v[tid] = y[tid]*y[tid]+x[tid];
}

}
```

6.2:

```
__global__ void kernel6_2(double *v, const double *x, const double *A, const int n){

int tid = blockIdx.x * blockDim.x + threadIdx.x;

if(tid < n){
v[i]=0.0;
for(int j=0; j<n; j++){
v[i] += A[tid*n+j] * x[j];
}
}

}
```