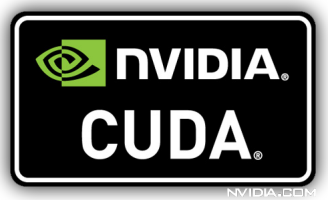
UNIVERSIDAD DE GRANADA

Práctica 4: Inicio *CUDA*

Arquitectura y Computación de altas prestaciones



Luis González Romero

26 de abril de 2019

Índice

Índice de figuras

1. Enunciado

- Crear un proyecto que nos muestre por pantalla las características de la GPU donde vais a ejecutar vuestras prácticas utilizando la función cudaGetDeviceProperties.
- Implementar una versión del problema que multiplique dos vectores, ejecutándose sólo en la CPU.
- Implementar una versión del problema que multiplique dos vectores, ejecutándose la multiplicación en la GPU.

2. cudaGetDeviceProperties

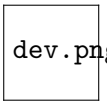


Figura 1: Salida por terminal de cudaGetDeviceProperties

3. Versión CPU

La versión en CPU se trata de un programa simple donde se leen dos ficheros de entrada y se opera con estos.

```
for (int i = 0; i < numElements; ++i)
{
  for (int j=0; j < 1000; j++)
  h_C[i] = h_A[i] * h_B[i] + j;
}
```

Figura 2: Computación en CPU

Se ha fatigado el for que tenía incialmente para que se vea reflejada la diferencia con cuda.

4. Versión GPU

La versión en GPU se trata de una versión alternativa del sample que nos ofrece NVIDIA(sobre vector addition) pero en este caso de multiplicación llevada a cabo de la misma forma que para CPU.

```
__global__ void
vectorMul(const float *A, const float *B, float *C, int numElements)
{
    int i = blockDim.x * blockIdx.x + threadIdx.x;

    if (i < numElements)
    {
      for (int j=0; j < 1000; j++)
        C[i] = A[i] * B[i] + j;
    }
}
```

Figura 3: Función multiplicación CUDA

Se opera de la misma forma que en CPU, invocándose este método con el kernel de CUDA tras hacer los *mallocs* y *memcpy(HostToDevice)* necesarios.

Ya solo queda realizar *memcpy(DeviceToHost)* y liberar todo el espacio con *cudaFree*(también de los ficheros abiertos para los vectores de entrada).

5. Comparación gráfica de las versiones

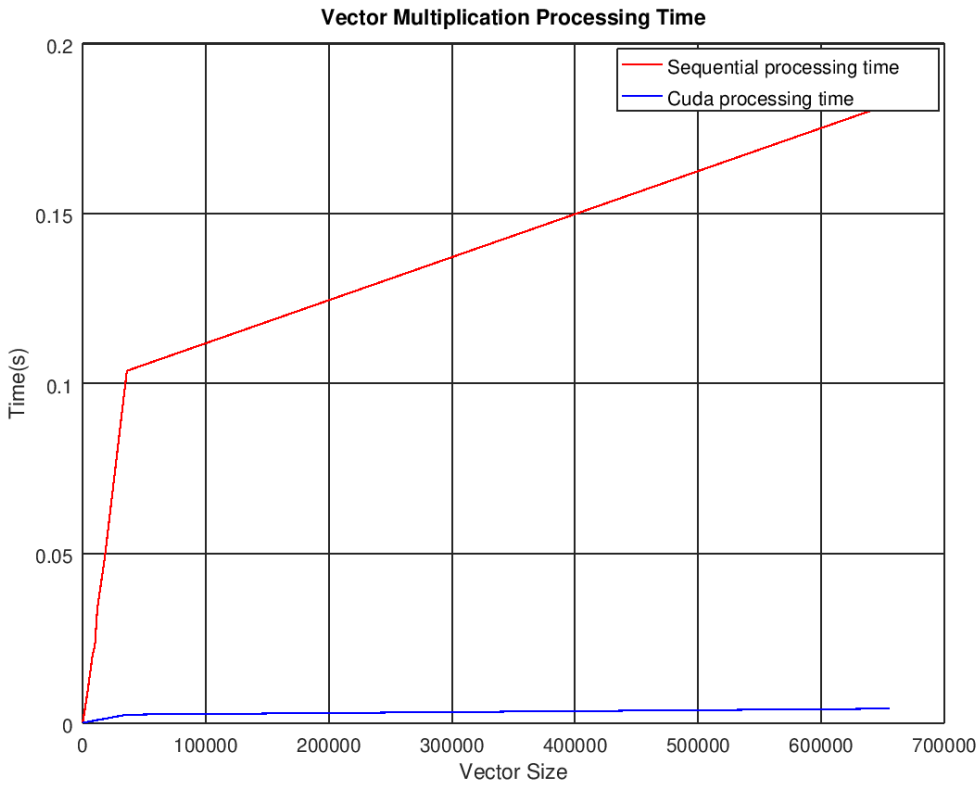


Figura 4: Tiempos de ejecución de ambas versiones

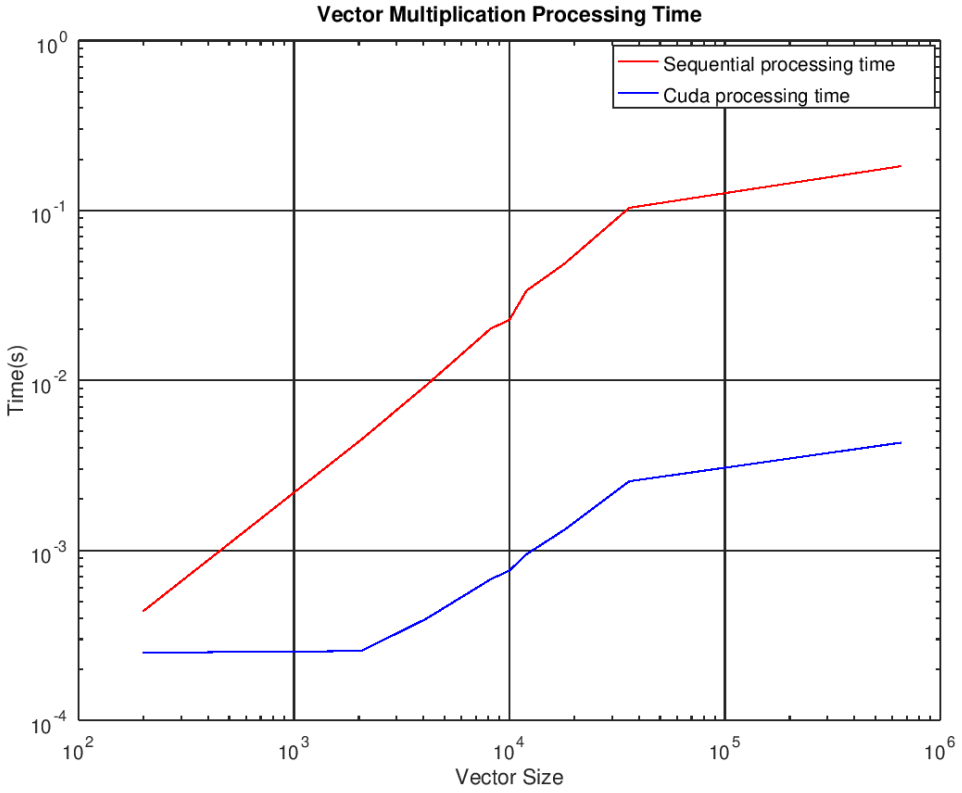


Figura 5: Tiempos de ejecución de ambas versiones con escala logarítmica en ambos ejes

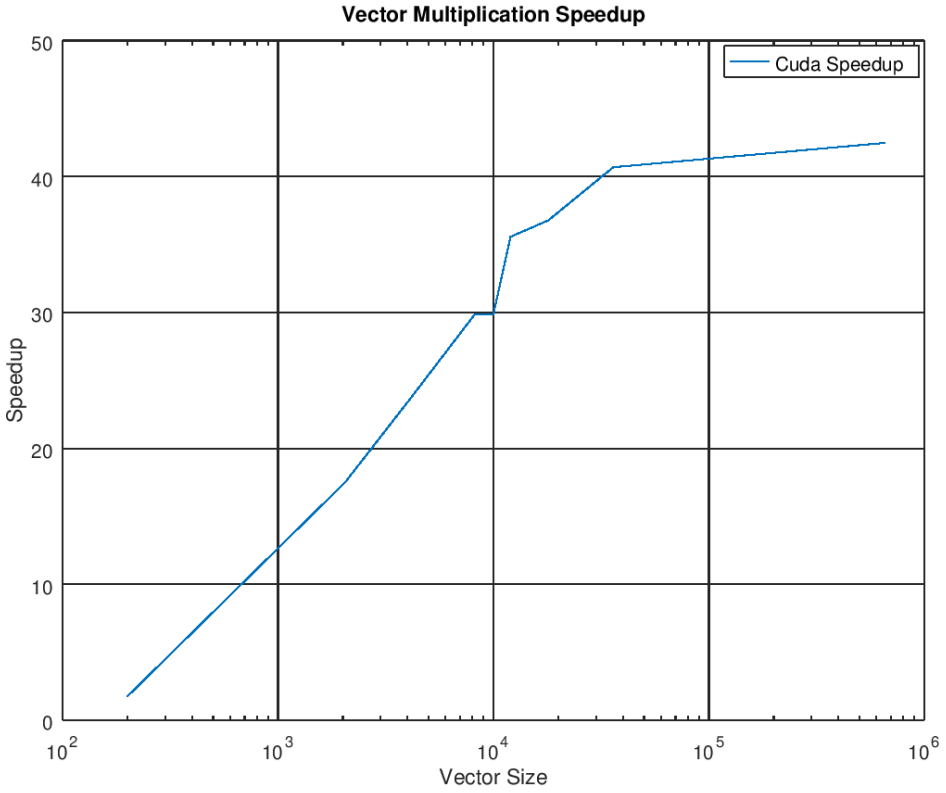


Figura 6: Ganancia obtenida con *CUDA*, escala logarítmica en el eje x

La siguiente tabla muestra los cálculos estadísticos de los tiempos de ejecución obtenidos para ambas versiones.

NumElem	**Media aritmética(μ)**	**Desviación estándar(σ)**
200	0.00043948357142857	5.8422569967693E-6
2050	0.00447632075	1.9223651739062E-5
4096	0.009323553	0.00026463559808249
8192	0.0201927795	0.0017321395403606
10000	0.0227121635	0.00012482941927574
12000	0.0337206876666675	0.00011958691370144
18000	0.04861527	0.00047098291491866
36000	0.10370042466667	0.0005327146236126
65536	0.18217966766667	0.0015538152782421

Cuadro 1: Estadísticos CPU

NumElem	Media aritmética(μ)	Desviación estándar(σ)
200	0.000249930625	1.5099500711427E-5
2050	0.00025567025	5.3467016175874E-6
4096	0.0003946715	1.2280724011638E-5
8192	0.00067617075	2.4856396132332E-5
10000	0.00075986975	1.1397960088871E-5
12000	0.0009481945	2.17670066672E-5
18000	0.00132184475	4.0506911980401E-5
36000	0.00254901975	9.3821985262877E-6
65536	0.00429938375	8.1383175744018E-5

Cuadro 2: Estadísticos GPU

6. Conclusión

Con *CUDA* se obtiene una ganancia muy superior a las conseguidas en otros ámbitos como *OpenMPI*, donde se veía como a lo sumo se limitaba a tender al *-np* usado, mientras que con este ejemplo básico se alcanza una ganancia superior a 40 para un número mediano de elementos fatigado.

Cuando el cómputo empieza a ser costoso se compensa el tiempo gastado en las copias *DeviceToHost* y *HostToDevice* ya que el tiempo de cómputo es ridículo.