Cristina María Garrido López

Universidad de Granada

Ingeniería de Computadores

Arquitectura y Computación de Altas Prestaciones

Práctica 4 y 5. Inicio en CUDA

En esta práctica se ejecutará un programa que implementa la suma de dos vectores, primero se lanzará en secuencial y después en paralelo en la GPU. En nuestro caso vamos a utilizar la GPU proporcionada por la profesora accediendo a través de SSH. Vemos las características de esta con el siguiente programa que encontramos en la página que se indica en las diapositivas de la práctica.

```
#include <stdio.h>
#include <stdlib.h>


int main(void){

    int deviceCount;
    cudaGetDeviceCount(&deviceCount);
    int device;
    for (device = 0; device < deviceCount; ++device) {
        cudaDeviceProp deviceProp;
        cudaGetDeviceProperties(&deviceProp, device);
        printf("Device %d has compute capability %d.%d.\n",
               device, deviceProp.major, deviceProp.minor);
    }
}
```

La salida es la siguiente:

```
cristgl2017@genmagic:~$ ./properties
Device 0 has compute capability 3.0.
cristgl2017@genmagic:~$
```

Ahora ejecutamos el siguiente programa a través de SSH, este suma dos vectores primero en secuencial y después en paralelo midiendo el tiempo ocupado por cada una de las ejecuciones.

```
#include <stdio.h>
#include <stdlib.h>
#include <fstream>
#include <iostream>
#include <string>
#include <vector>
#include "cuda.h"


using namespace std;
```

```
#define BILLION 1E9;

__global__ void vecAddKernel(float *A, float *B, float *C, int n){
       int i = threadIdx.x+blockDim.x*blockIdx.x;
       if(i<n) C[i] = A[i]+B[i];
}

void vecAddK(float *A, float *B, float *C, int n){
       for(int i=0; i<n; i++)
              C[i] = A[i]+B[i];
}

void vecAdd(float *hA, float *hB, float *hC, int n){
    struct timespec requestStart, requestEnd, requestS, requestE;
    int size = n*sizeof(float);
    float dA[size], dB[size], dC[size];

    clock_gettime(CLOCK_REALTIME, &requestStart);
    vecAddK(hA,hB,dC,n);
    cout << dC[0] <<" " << dC[n-1] << endl;
    clock_gettime(CLOCK_REALTIME, &requestEnd);

    cudaMalloc((void **) &dA, size);
    cudaMemcpy(dA, hA, size, cudaMemcpyHostToDevice);
    cudaMalloc((void **) &dB, size);
    cudaMemcpy(dB, hB, size, cudaMemcpyHostToDevice);
    cudaMalloc((void **) &dC, size);

    dim3 DimGrid(((n-1)/256)+1,1,1);
    dim3 DimBlock(256,1,1);
    clock_gettime(CLOCK_REALTIME, &requestS);
    vecAddKernel<<<DimGrid,DimBlock>>>(dA,dB,dC,n);
    cudaMemcpy(hC, dC, size, cudaMemcpyDeviceToHost);
    cout << dC[0] <<" " << dC[n-1] << endl;
    clock_gettime(CLOCK_REALTIME, &requestE);
```

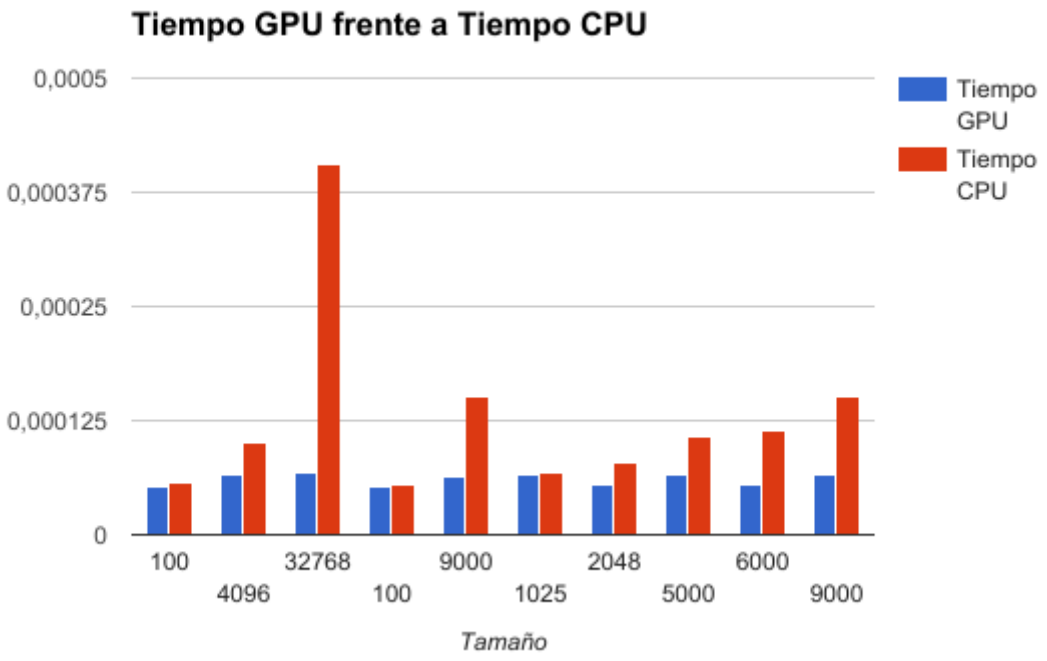
```
    double accum = (double) (requestEnd.tv_sec - requestStart.tv_sec) +
(requestEnd.tv_nsec - requestStart.tv_nsec)/BILLION;

    printf( "CPU %lf\n", accum );

    double accumu = (double) (requestE.tv_sec - requestS.tv_sec) + (requestE.tv_nsec -
requestS.tv_nsec) / BILLION;

       printf( "GPU %lf\n", accumu );


    cudaFree(dA); cudaFree(dB); cudaFree(dC);

    cudaError_t err = cudaMalloc((void **) &dA, size);


       if(err != cudaSuccess){

              printf("%s en %s en línea %d\n",
cudaGetErrorString(err),__FILE__,__LINE__);

              exit(EXIT_FAILURE);

       }

}

int main(){

       int n, i=0;

       string line;

       ifstream fA("data/0/input0.raw");

       ifstream fB("data/0/input1.raw");

       ifstream fC("data/0/output.raw");


       if(fA){

              getline(fA,line);

              n = stoi(line,NULL,0);

       }


       float hA[n];

       float hB[n];

       float hC[n];


       if(fA){

              while(getline(fA,line)){

                     float f = stof(line,NULL);

                     hA[i] = f;

                     i++;

              }
```

```
            fA.close();
        }
        i=0;
        if(fB){
            getline(fB,line);
            n = stoi(line,NULL,0);
            while(getline(fB,line)){
                float f = stof(line,NULL);
                hB[i] = f;
                i++;
            }
            fB.close();
        }
        vecAdd(hA, hB, hC, n);
}
```

El programa nos indicará el tiempo que se ha tardado en ejecutar la suma en la CPU y en la GPU además de mostrar el primer y último elemento del vector C en el que se ha introducido el resultado para que no lo borre automáticamente el compilador. Debemos compilar con la orden `/usr/local/cuda-8.0/bin/nvcc -m64 -I/usr/local/cuda-8.0/include -o sumaVec sumaVec.cu -std=c++11` y a continuación ejecutamos normalmente. En la siguiente tabla vemos los resultados para cada una de las carpetas seleccionadas.

Tamaño	Tiempo GPU	Tiempo CPU
100	0,000053	0,000058
4096	0,000066	0,000101
32768	0,000067	0,000406
100	0,000052	0,000055
9000	0,000064	0,000152
1025	0,000065	0,000067
2048	0,000054	0,000079
5000	0,000065	0,000107
6000	0,000054	0,000115
9000	0,000065	0,000151

Realizamos una gráfica para apreciar mejor la diferencia.



En ella vemos que el tiempo secuencial siempre es mayor que el tiempo ocupado por la GPU, ya que se ha llamado al kernel que lo realiza de forma paralela.