Multiplicación de matrices con Cuda

por Daniel Cerrato y David T. Garitagoitia

Objetivo

La idea básica es la de comparar los tiempos de ejecución de un programa que calcule la matriz resultante de haber multiplicado varias matrices, tanto por la CPU como GPU (con ayuda de CUDA).

Como es de esperar, los tiempos de la GPU deben ser mucho menores que los de la CPU.

Proceso

- **Creación del código básico**
  Creamos el código que multiplica dos matrices, primero pensado para CPU (código genérico) y luego adaptado a GPU
- **Preparación de entorno**
  Generamos el código que engloba la llamada a la función de multiplicación para CPU y GPU.
- **Añadimos funcionalidades**
  La multiplicación de dos matrices es algo sencillo. Podría servir, pero la idea es ir un poco más allá.

Generación de código básico

Primero generamos el código básico para ambas modalidades

```
__global__ void matrixMultiplicationKernel(int* mt1, int* mt2, int* result, int filas1, int columnas1, int columnas2) {

    int fila = blockIdx.y*blockDim.y+threadIdx.y;
    int columna = blockIdx.x*blockDim.x+threadIdx.x;

    //Comprobamos que no se haya salido de la matriz
    if (fila < filas1 && columna < columnas2) {
        float tmpSum = 0;
        // cada thread se encarga de un elemento
        for (int i = 0; i < columnas1; i++) {
            tmpSum += mt1[fila * columnas1 + i] * mt2[i * columnas2 + columna];
        }

      //Guardamos la suma total en su posición
      result[fila * columnas2 + columna] = tmpSum;
    }
}
```

GPU

CPU

```
int *multiplicaMatrices(int *mat1, int *mat2, int filas1, int comp, int columnas2){
  int *matRes = (int *)calloc(filas1 * columnas2, sizeof(int));

  for (int i = 0; i < filas1; i++){
    for (int j = 0; j < columnas2; j++){
      for (int k = 0; k < comp; k++){
        matRes[i * columnas2 + j] += mat1[i * comp + k] * mat2[k * comp + j];
      }
    }
  }
  return matRes;
}
```

Ideas principales

Como indicamos, la idea principal es la de multiplicar varias matrices seguidas.

Tuvimos varias ideas sobre cómo podríamos proceder. Para ambas modalidades, creímos conveniente hacer varios métodos de cálculo:

- Multiplicación directa
- Orden específico según dimensiones
- Multiplicación por pares

MULTIPLICACIÓN DIRECTA

La multiplicación directa es la forma básica y tradicional de multiplicación de matrices, siguiendo el orden en el que están escritas las matrices.

Esta funcionalidad pensamos en usarla con la CPU y con la GPU para confirmar la diferencia de velocidad de cálculo.

```
int *resMatrix = matrices[0];
for (int i = 1; i < numMatrices; i++){
    resMatrix = multiplicaMatrices(resMatrix, matrices[i], arrayFilas[0], arrayColumnas[i-1], arrayColumnas[i]);
}
```

Como se puede observar, el código es muy simple, ya que solo tiene que recorrer las matrices e ir haciendo llamadas a la función de cálculo.

Para la GPU, el código es el mismo pero llamando al kernel.

ORDEN ESPECÍFICO SEGÚN DIMENSIONES

En este caso, queríamos dar algo de velocidad de procesado a través de unos “pre-cálculos”, que nos ayudarían a conocer el orden óptimo para multiplicar las matrices, realizando las menores multiplicaciones internas posibles.

Para ello, modificaríamos el código generado en las prácticas de la asignatura de “Algoritmia y Estructuras de Datos Avanzadas”. El código original calcula la cantidad de multiplicaciones mínimas a realizar dadas las dimensiones de las matrices. Para conseguir nuestro propósito, simplemente necesitamos guardar la posición de los paréntesis que permiten optimizar la multiplicación y devolver el orden en el que se realizan las multiplicaciones.

Dado que nos ha comido el tiempo, esta parte no hemos podido completarla. Pero la idea era usar esta funcionalidad añadida a la básica, mejorando el rendimiento de ambas modalidades.

MULTIPLICACIÓN POR PARES

Otra forma que se nos ocurrió para multiplicar las matrices es la de multiplicarlas dos a dos, reduciendo el número de matrices a la mitad en cada iteración hasta llegar a la matriz final.

```
for ( ; numMatrices > 1; numMatrices /= 2){

    imparFlag = 0;
    if (numMatrices % 2 != 0)
      imparFlag = 1;

    for (int index = imparFlag; index < numMatrices; index += 2){

    .........

    numMatrices += imparFlag;
}
```

Como se aprecia, tuvimos que hacer un pequeño tratamiento al algoritmo para que funcionase correctamente en caso de que el número de matrices fuese impar en algún momento.

La idea es prácticamente la misma que vimos con los “intrinsics” en la práctica 1 al usar funciones como la que sumaba horizontalmente a pares.

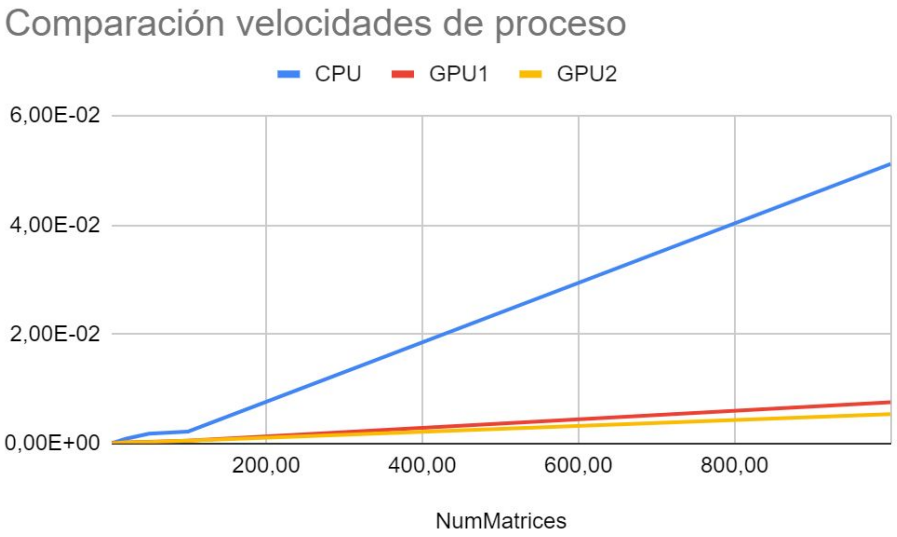
MULTIPLICACIÓN POR PARES II

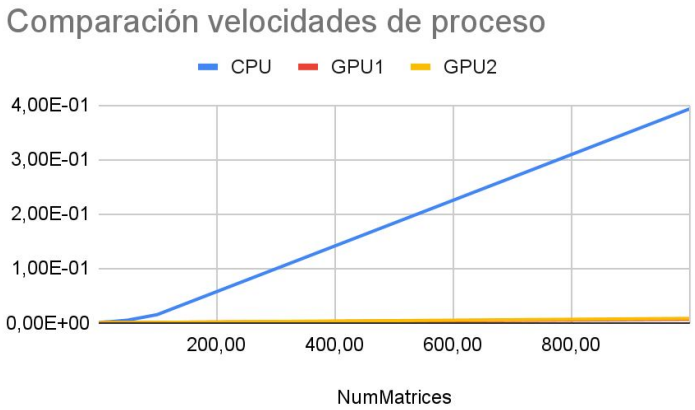
Para este apartado, pensamos en dos vertientes:

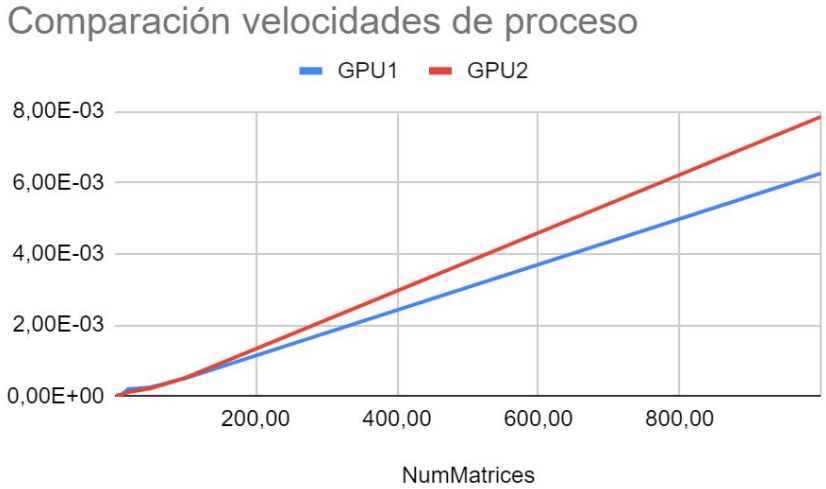
- Serie: Realizar la reducción de matrices por pares atendiendo a los cálculos por iteración par a par. Esta sería la versión más sencilla y a la vez más lenta.

- Paralelo: Utilizar hilos o subprocesos para lanzar varias llamadas a las funciones de cálculo a la vez, simplemente haciendo espera al proceso padre a que terminen todas las llamadas. Esta sería una versión más rápida, por lo menos en GPU, pero también complica ligeramente el código.

Algunos resultados







Muchas gracias por vuestro tiempo