ALGORITMOS PARALELOS

EJERCICIO 3.1

June 11, 2017

Norman Patrick Harvey Arce
Universidad Nacional de San Agustín - Arequipa
Ciencia de la Computación

EJERCICIO 3.1

A matrix addition takes two input matrices A and B and produces one output matrix C. Each element of the output matrix C is the sum of the corresponding elements of the input matrices A and B, i.e., C[i][j] = A[i][j] + B[i][j]. For simplicity, we will only handle square matrices whose elements are single-precision floating-point numbers. Write a matrix addition kernel and the host stub function that can be called with four parameters: pointer-to-the-output matrix, pointer-to-the-first-input matrix, pointer-to-the-second-input matrix, and the number of elements in each dimension. Follow the instructions below:

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

__global__ void Suma_matrices(float A[], float B[], float C[], int n)
{
   int my_ij = blockDim.x * blockIdx.x + threadIdx.x;

   if (blockIdx.x < n && threadIdx.x < n)
      C[my_ij] = A[my_ij] + B[my_ij];
}

void leer_matriz(float A[], int n)
{
   int i, j;

   for (i = 0; i < n; i++)
      for (j = 0; j < n; j++)
         scanf("%f", &A[i*n+j]);
}

void imprimir_matriz(char title[], float A[], int n)
{
   int i, j;

   printf("%s\n", title);
   for (i = 0; i < n; i++)
   {
      for (j = 0; j < n; j++)
         printf("%.1f ", A[i*n+j]);
      printf("\n");
```

```
    }
}


int main(int argc, char* argv[]) {
    int n;
    float *h_A, *h_B, *h_C;
    float *d_A, *d_B, *d_C;
    size_t size;

    /* Get size of matrices */
    if (argc != 2) {
        fprintf(stderr, "uso: %s <dimension>\n", argv[0]);
        exit(0);
    }
    n = strtol(argv[1], NULL, 10);
    printf("n = %d\n", n);
    size = n*n*sizeof(float);

    h_A = (float*) malloc(size);
    h_B = (float*) malloc(size);
    h_C = (float*) malloc(size);

    printf("Ingrese las matrices A y B\n");
    leer_matriz(h_A, n);
    leer_matriz(h_B, n);

    imprimir_matriz("A=", h_A, n);
    imprimir_matriz("B=", h_B, n);

    /* Asigna memoria para las matrices en el dispostivo */
    cudaMalloc(&d_A, size);
    cudaMalloc(&d_B, size);
    cudaMalloc(&d_C, size);

    /* Copia las matrices del host al dispositivo */
    cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);

    /* Invoca el kernel con n bloques de hilos    */
    /* los cuales contienen n hilos               */
    Suma_matrices<<<n, n>>>(d_A, d_B, d_C, n);
```

```
    /* Espera que finalice el dispositivo de realizar las operaciones*/
    cudaThreadSynchronize();

    /* Copiar los resultados del dispositivo al host */
    cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);

    imprimir_matriz("La suma es: ", h_C, n);

    /* Liberar memoria en el dispositivo */
    cudaFree(d_A);
    cudaFree(d_B);
    cudaFree(d_C);

    /* Liberar memoria en el host */
    free(h_A);
    free(h_B);
    free(h_C);

    return 0;
}
```

A. Write the host stub function by allocating memory for the input and output matrices, transferring input data to device; launch the kernel, transferring the output data to host and freeing the device memory for the input and output data. Leave the execution configuration parameters open for this step.

```
    h_A = (float*) malloc(size);
    h_B = (float*) malloc(size);
    h_C = (float*) malloc(size);

    cudaMalloc(&d_A, size);
    cudaMalloc(&d_B, size);
    cudaMalloc(&d_C, size);

    cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);

    Suma_matrices<<<n, n>>>();

    cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);
```

```
   cudaFree(d_A);
   cudaFree(d_B);
   cudaFree(d_C);

   free(h_A);
   free(h_B);
   free(h_C);
```

B. Write a kernel that has each thread to produce one output matrix element. Fill in the execution configuration parameters for this design.

```
Suma_matrices<<<n,1>>>(d_A, d_B, d_C, n);

__global__ void Mat_add(float A[], float B[], float C[], int n)
{
 int tx = threadId.x;
 int ty = threadId.y;

 float elemento_C = 0;
 for (int k = 0; k < n; ++k)
        {
        float elemento_A = A[ty * n + k];
        float elemento_B = B[k * n + tx];
        elemento_C += elemento_A * elemento_B;
        }

 C[ty*n + tx ] = elemento_C;
}
```

C. Write a kernel that has each thread to produce one output matrix row. Fill in the execution configuration parameters for the design.

```
una_fila<<<n,n>>>(d_A, d_B, d_C, n);

__global__ void una_fila(float A[], float B[], float C[], int n)
{
   int fila_A = blockIdx.x + threadIdx.y;
   int fila_B = blockIdx.y + threadIdx.y;
   float sum = 0;
   for (int k = 0; k < n; k++)
     {
      sum += A[fila_A*n+k]+B[fila_B*n+k];
```

```
            C[n+k] = sum;
        }
}
```

D. Write a kernel that has each thread to produce one output matrix column. Fill in the execution configuration parameters for the design.

```
una_columna<<<n,n>>>(d_A, d_B, d_C, n);

__global__ void una_columna(float A[], float B[], float C[], int n)
{
   int fila_A = blockIdx.y + threadIdx.y;
   int fila_B = blockIdx.x + threadIdx.x;
   float sum = 0;
   for (int k = 0; k < n; k++)
     {
      sum += A[fila_A*n+k]+B[fila_B*n+k];
          C[n+k] = sum;
     }
}
```

E. Analyze the pros and cons of each kernel design above.

Ya que sólo trabajamos con matrices cuadradas, los casos donde son diferentes el número de filas y columnas no serán analizados.

Pros

Las ventajas de utilizar hilos para programar la suma de matrices son muchas que van desde el tiempo de ejecución hasta usar menos recursos computacionales

- Trabajo en paralelo
- Menos coste computacional
- Menor tiempo computacional

Cons

Las contras de estos tipos de Kernel son las siguientes:

- Para el caso B, el hecho que produzca sólo un elemento por hilo será parecido a un algoritmo serial
- Para C y D, el hecho de producir una fila o una columna estaría demorando el trabajo que puede ser simplificado en una sóla función