

CENTRO DE INVESTIGACIÓN EN MATEMÁTICAS (CIMAT).
UNIDAD MONTERREY
PROGRAMACIÓN Y ANÁLISIS DE ALGORITMOS

Tarea 2 de Cómputo Paralelo

Gustavo Hernández Angeles

24 de noviembre de 2024

1 Problema 1

Dado un vector de números reales V de tamaño N , programar lo siguiente:

- a) $S_1[i] = V[i] + V[i + 1]$ para $i = 0, \dots, N - 2$, con S_1 otro vector de tamaño $N - 1$.
- b) $S_2[i] = \frac{V[i+1] + V[i-1]}{2}$ para $i = 1, \dots, N - 2$, con S_2 otro vector de tamaño $N - 2$.

SOLUCIÓN

Inciso a)

Código secuencial

En la tarea previa pudimos realizar este mismo ejercicio tanto en su versión paralela (usando OpenMP) como en su versión secuencial. Podemos extraer entonces el mismo código secuencial.

```
void inciso_a_s(float *V, float *A, long int N)
{
    long int i;
    for (i = 0; i < N - 1; i++)
    {
        A[i] = V[i] + V[i + 1];
    }
}
```

Donde simplemente iteramos sobre el tamaño del vector resultante, y guardamos los valores de acuerdo a la definición del problema.

Código paralelo

En la versión paralela hacemos uso de el índice tanto de los bloques como de los hilos para calcular el índice del vector resultante. Únicamente validamos que el índice sea menor a $N-1$ para evitar acceder a memoria no asignada.

```
--global-- void incisoA(float *s_d, float *v_d, int N)
{
    int idx = blockDim.x * blockIdx.x + threadIdx.x;
    if (idx < N - 1)
    {
        s_d[idx] = v_d[idx] + v_d[idx + 1];
    }
}
```

De igual forma, hacemos que cada hilo almacene en el vector resultante los valores de acuerdo a su definición e índice.

Inciso b)

Código secuencial

Podemos extraer de la tarea pasada el código secuencial para este problema en forma de una función.

```
void inciso_b_s(float *V, float *A, long int N)
{
    long int i;
    for (i = 1; i < N - 1; i++)
    {
        A[i - 1] = (V[i - 1] + V[i + 1]) / 2;
    }
}
```

Note que en el ciclo el índice donde se almacena cada valor en A es $i-1$ mientras que se utilizan los índices $i-1$ y $i+1$ para acceder al vector V. Por lo que no se deja el primer elemento libre.

Código paralelo

Calculamos el índice en función de los índices del bloque y del hilo que estamos ejecutando. Y después hacemos la validación de que el índice debe ser mayor a 0 y menor a $N-1$, de acuerdo a la definición de S_2 .

```
--global-- void incisoB(float *s_d, float *v_d, int N)
{
    int idx = blockDim.x * blockIdx.x + threadIdx.x;
    if (idx > 0 && idx < N - 1)
    {
        s_d[idx - 1] = (v_d[idx + 1] + v_d[idx - 1]) / 2;
    }
}
```

Secuencial vs Paralelo

Hemos visto que las implementaciones del problema tanto en paralelo como en secuencial han sido muy similares, cambiando únicamente en cómo accedemos al vector para almacenar sus valores. Sin embargo, podremos ver que hay cambios importantes en el tiempo de ejecución entre ambos métodos.

Para medir el tiempo de ejecución de funciones en el CPU utilizamos la librería **chrono** con la función **now()** antes y después de llamar a la función. Por ejemplo

```
auto inicio = chrono::high_resolution_clock::now();

for (k = 0; k < n_iteraciones; k++)
    inciso_a_s(v_h, s1_h, N);

auto fin = chrono::high_resolution_clock::now();
chrono::duration<double> duracion = (fin - inicio) / n_iteraciones;
cout << "Tiempo en secuencial (segundos):\t" << duracion.count() << endl;
```

Para medir el tiempo de ejecución de funciones en la GPU utilizamos la API CUDA Event. Esto nos permite crear eventos y calcular el tiempo transcurrido entre cada uno. Guardando el tiempo de eventos antes y después del lanzamiento del kernel proveerán el tiempo de ejecución en la GPU. Aquí hay un ejemplo.

```
float tiempo_paralelo = 0;
cudaEvent_t inicio_p, fin_p; // Declaramos los eventos
cudaEventCreate(&inicio_p); // Los creamos
cudaEventCreate(&fin_p);

cudaEventRecord(inicio_p); // Guardamos tiempo antes

for (k = 0; k < n_iteraciones; k++)
    incisoA<<<n_blocks, block_size>>>(s1_d, v_d, N);

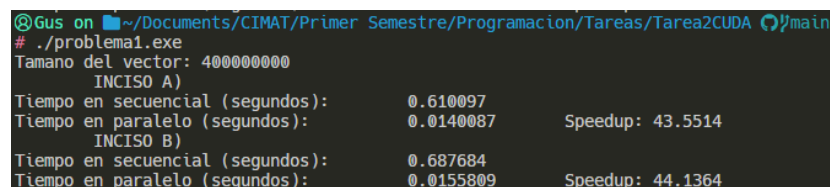
cudaEventRecord(fin_p); // Guardamos tiempo despues

cudaEventSynchronize(fin_p);
cudaEventElapsedTime(&tiempo_paralelo, inicio_p, fin_p);
// Milisegundos a segundos.
tiempo_paralelo = tiempo_paralelo / (1000 * n_iteraciones);
cout << "Tiempo en paralelo (segundos):\t\t" << tiempo_paralelo << "\t"
    << "Speedup:-" << duracion.count() / tiempo_paralelo << endl;
```

Donde `cudaEventRecord(a)` crea una marca de tiempo para el evento `a`, la función para sincronizar `cudaEventSynchronize(a)` detiene la ejecución CPU hasta que se registre el evento `a`, y `cudaEventElapsedTime(&t, a, b)` calcula el tiempo transcurrido entre el registro del evento `a` y el registro del evento `b`. *Nota:* Ambas mediciones se hicieron calculando el tiempo en el que las funciones se ejecutan `n_iteraciones` veces, y el tiempo resultante se divide entre este número de iteraciones. Este número varía con cada problema por su complejidad computacional.

Resultados

Debemos tener en cuenta nuestro hardware para decidir el tamaño máximo de los arreglos que pretendemos alojar en la memoria. En mi caso, tengo una tarjeta gráfica con 8 GB de VRAM y mi equipo cuenta con 32 GB de RAM. Entonces 1.5 GB de RAM para cada vector puede ser una buena prueba, esto se consigue con $N = 4 \times 10^8$ haciendo que los vectores contengan datos de tipo `float`. Adicionalmente, los parámetros del lanzamiento del kernel también pueden tener un impacto en el rendimiento del mismo, en este caso utilizamos la capacidad máxima de hilos por bloque en mi GPU (1024) y el número de bloques lo definimos como $(N + \text{block_size} - 1) / \text{block_size}$. Finalmente, se muestran los resultados calculando los tiempos de ejecución y el Speedup. Los resultados muestran que las funciones ejecutadas en paralelo son, como mínimo, 40 veces más rápidas para ambos incisos.



```
@Gus on ~/Documents/CIMAT/Primer Semestre/Programacion/Tareas/Tarea2CUDA [main]
# ./problema1.exe
Tamano del vector: 400000000
INCISO A)
Tiempo en secuencial (segundos):    0.610097
Tiempo en paralelo (segundos):      0.0140087    Speedup: 43.5514
INCISO B)
Tiempo en secuencial (segundos):    0.687684
Tiempo en paralelo (segundos):      0.0155809    Speedup: 44.1364
```

2 Problema 2

Dadas dos matrices A y B de tamaño $N \times M$ con valores enteros positivos, programar lo siguiente:

- a) $C_1(i, j) = A(i, j) + B(N - i - 1, M - j - 1)$ para $i = 0, \dots, N - 1$ y $j = 0, \dots, M - 1$, con C_1 otra matriz de tamaño $N \times M$.
- b) $C_2(i, j) = \alpha A(i, j) + (1 - \alpha)B(i, j)$ para $i = 0, \dots, N - 1$ y $j = 0, \dots, M - 1$, con α un valor real constante entre $[0, 1]$ que podemos pasar como parámetro a la función *kernel*, y C_2 otra matriz de tamaño $N \times M$.

SOLUCIÓN

Inciso a)

Código secuencial

Comenzamos con la declaración de variables, la función toma como entrada las matrices A , B y C de tipo `unsigned int` por tener elementos enteros no negativos (asumimos que C también, ya que es la suma de los elementos de A y B). Y el tamaño de las matrices M y N .

La idea es iterar sobre todas las posiciones que puede contener C , en cada iteración calcular el índice `idx` para las matrices A y B , y el índice `idx_b` para la matriz B . Entonces se inicializa el valor del elemento (i, j) de la matriz C como se define en el problema.

```
void incisoA_secuencial(unsigned int *A, unsigned int *B,
                        unsigned int *C, long int N, long int M)
{
    long int i, j, idx, idx_b;

    for (i = 0; i < N; i++)
    {
        for (j = 0; j < M; j++)
        {
            idx = i * M + j;
            idx_b = (N - i - 1) * M + (M - j - 1);
            C[idx] = A[idx] + B[idx_b];
        }
    }
}
```

Código paralelo

La configuración que se propone para el lanzamiento del kernel es de una malla de bloques de 32×32 hilos, tomando su capacidad máxima. De esta forma, se mapea cada hilo con un correspondiente elemento de la matriz C . Primero determinamos el índice i con `idy` y el índice j con `idx`. Después, verificamos que los índices no estén fuera del rango de nuestra matriz; `idx < M` y `idy < N` para entonces darle el valor correspondiente al elemento (i, j) de la matriz C .

```
--global-- void incisoA(unsigned int *A, unsigned int *B,
                        unsigned int *C, long int N, long int M)
```

```

{
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    int idy = blockIdx.y * blockDim.y + threadIdx.y;
    int id = idy * M + idx;
    int id_B = (N - idy - 1) * M + (M - idx - 1);
    if (idx < M && idy < N)
        C[id] = A[id] + B[id_B];
}

```

Inciso b)

Nuevamente, en este inciso las dos implementaciones de la solución al problema son muy similares, particularmente ambas toman como valores de entrada las matrices de enteros no negativos A y B , la matriz de salida de tipo flotante C , la constante α de tipo flotante, y el tamaño de las matrices N y M .

Código secuencial

El código en secuencial tiene una complejidad de $O(n^2)$, iterando sobre todos los elementos de la matriz C para darles un valor. La asignación se realiza mediante la definición del problema.

```

void incisoB_secuencial(unsigned int *A, unsigned int *B,
                        float *C, float a, long int N, long int M)
{
    long int i, j, idx;

    for (i = 0; i < N; i++)
    {
        for (j = 0; j < M; j++)
        {
            idx = i * M + j;
            C[idx] = a * A[idx] + (1 - a) * B[idx];
        }
    }
}

```

Código paralelo

Al igual que en el inciso a) de este problema, la configuración de la malla de bloques es de 32×32 hilos, asignando a cada elemento de la matriz un hilo. A simple vista, el código parece ser de complejidad $O(1)$ aunque se debe tomar en cuenta la concurrencia de los bloques, el hardware utilizado y otras razones por las que su tiempo de ejecución no resulta ser constante.

```

--global-- void incisoB(unsigned int *A, unsigned int *B,
                        float *C, float a, long int N, long int M)
{
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    int idy = blockIdx.y * blockDim.y + threadIdx.y;
    int id = idy * M + idx;

    if (idx < M && idy < N)
        C[id] = a * A[id] + (1 - a) * B[id];
}

```

Hacemos de igual forma el mapeo de los índices tanto de bloques como de hilos a los índices de la matriz, nos aseguramos que esté dentro de los límites de la matriz, y realizamos la asignación al elemento mediante la definición.

Secuencial vs Paralelo

Justo como en el problema 1, utilizamos la librería `chrono` y los `cudaEvents` para medir los tiempos de ejecución en CPU y GPU respectivamente. También dentro de cada problema se incluye la verificación de los cálculos realizados por la GPU y el CPU. Un ejemplo de la verificación es el siguiente bloque de código.

```
/*          VERIFICACION          */
cudaMemcpy(verificacion_C1, C_1d, size, cudaMemcpyDeviceToHost);

for (i = 0; i < N * M; i++)
{
    if (fabs(verificacion_C1[i] - C_1h[i]) > 1e-3)
    {
        printf("Inciso A--Error en el indice-%d\n", i);
        printf("%d!=%d\n", verificacion_C1[i], C_1h[i]);
        exit(EXIT_FAILURE);
    }
}
```

Resultados

En este problema realicé la comparación de los tiempos de ejecución para matrices de 12300×12300 elementos. Sin embargo, encontré que el rendimiento varía significativamente en cada ejecución del código incluso cuando se dan los mismos valores de entrada. Honestamente no entiendo por qué, pero creo que es bueno mencionarlo. El número de iteraciones por función se mantiene constante y es igual a 10.

```
tmpxft_00000e00_00000000-10_problema2.cudafe1.cpp
Creando biblioteca problema2.lib y objeto problema2.exp
Tamano de las matrices NxM: 12300 12300
Valor de alfa: 0.123
Inciso A):
Tiempo en secuencial (segundos): 0.432051
Tiempo en paralelo (segundos): 0.021392 Speedup: 20.1969
Inciso B):
Tiempo en secuencial (segundos): 0.409028
Tiempo en paralelo (segundos): 0.030711 Speedup: 13.3186
@Gus on ~\Documents\CIMAT\Primer Semestre\Programacion\Tareas\Tarea2CUDA ~\main
# ./problema2.exe
Tamano de las matrices NxM: 12300 12300
Valor de alfa: 0.123
Inciso A):
Tiempo en secuencial (segundos): 0.431215
Tiempo en paralelo (segundos): 0.00896952 Speedup: 48.0756
Inciso B):
Tiempo en secuencial (segundos): 0.408989
Tiempo en paralelo (segundos): 0.02284 Speedup: 17.9067
@Gus on ~\Documents\CIMAT\Primer Semestre\Programacion\Tareas\Tarea2CUDA ~\main
# ./problema2.exe
Tamano de las matrices NxM: 12300 12300
Valor de alfa: 0.123
Inciso A):
Tiempo en secuencial (segundos): 0.431555
Tiempo en paralelo (segundos): 0.0370601 Speedup: 11.6447
Inciso B):
Tiempo en secuencial (segundos): 0.407593
Tiempo en paralelo (segundos): 0.0319931 Speedup: 12.74
```

En conclusión, el Speedup suele ser del orden de 10 así que sigue siendo una gran mejora a comparación del código secuencial.

3 Problema 3

Dada una matriz A de tamaño $N \times K$ y una matriz B de tamaño $K \times M$ con valores en punto flotante de 64 bits (*double*), programar lo siguiente:

1. La multiplicación de las matrices A y B usando memoria global (GM).
2. La multiplicación de las matrices A y B usando memoria compartida (SM).
3. Evaluar el tiempo de procesamiento entre las versiones Serial, Paralelo usando GM y Paralelo usando SM, para diferentes tamaños de las matrices, por ejemplo:

$(512 \times 1024, 1024 \times 2048), (2048 \times 1024, 1024 \times 2048), (2048 \times 8192, 8192 \times 2048).$

SOLUCIÓN

Inciso a)

Para una implementación de la multiplicación de matrices utilizando la memoria global utilicé el algoritmo más común, es decir, para cada elemento c_{ij} de la nueva matriz se calcula el producto punto del i -ésimo renglón de A con la j -ésima columna de B mediante un ciclo **for**. El código tiene la siguiente forma.

```
// Inciso A
--global-- void mult_matrices_GM(double *A, double *B, double *C,
                                long int N, long int K, long int M)
{
    long int idx, idy, k, index;
    idx = blockDim.x * blockIdx.x + threadIdx.x;
    idy = blockDim.y * blockIdx.y + threadIdx.y;
    index = idy * M + idx;
    if (idy < N && idx < M)
    {
        double sum = 0.0;
        for (k = 0; k < K; k++)
        {
            sum += A[idy * K + k] * B[k * M + idx];
        }
        C[index] = sum;
    }
}
```

Inciso b)

La implementación en paralelo utilizando la memoria compartida se hace mediante bloques de matrices. Asignamos a cada bloque de nuestra malla una submatriz del resultado C , entonces se realizarán las multiplicaciones de submatrices de A y B para obtener el valor de la submatriz en C . El uso de la memoria compartida se aprovecha al hacer que los cálculos de cada hilo contribuyan al resultado de la submatriz correspondiente a su bloque, mejorando significativamente los tiempos de ejecución como se verá en la siguiente sección.

```

// Inciso B
--global-- void mult_matrices_SM(double *A, double *B, double *C,
                                long int N, long int K, long int M)
{
    // Block index
    int bx = blockIdx.x;
    int by = blockIdx.y;

    // Thread index
    int tx = threadIdx.x;
    int ty = threadIdx.y;

    // Indices de inicio , final de la submatriz de A
    // Pasos de la iteracion
    int aBegin = K * BLOCK_SIZE * by;
    int aEnd = aBegin + K - 1;
    int aStep = BLOCK_SIZE;

    // Indice de inicio de la submatriz de B
    // junto a sus pasos
    int bBegin = BLOCK_SIZE * bx;
    int bStep = BLOCK_SIZE * M;

    // Valor en cada posicion
    double Csub = 0;

    for (int a = aBegin, b = bBegin;
        a <= aEnd;
        a += aStep, b += bStep)
    {
        // Se utiliza la memoria compartida
        // para guardar las respectivas
        // submatrices de A y B.
        --shared-- double As[BLOCK_SIZE][BLOCK_SIZE];
        --shared-- double Bs[BLOCK_SIZE][BLOCK_SIZE];

        As[ty][tx] = A[a + K * ty + tx];
        Bs[ty][tx] = B[b + M * ty + tx];

        __syncthreads();

#pragma unroll

        for (int k = 0; k < BLOCK_SIZE; ++k)
        {
            Csub += As[ty][k] * Bs[k][tx];
        }

        __syncthreads();
    }

    int c = M * BLOCK_SIZE * by + BLOCK_SIZE * bx;
    C[c + M * ty + tx] = Csub;
}

```

Inciso c)

Se ejecutó el programa con los tamaños de matrices sugeridos en el documento de la tarea. Los resultados se muestran a continuación. Podemos observar que el incremento de velocidad con el algoritmo paralelo son enormes a comparación de un algoritmo ejecutado secuencialmente. Este incremento de velocidad es del orden de 500x, mostrando la necesidad de la programación paralela para funciones cuyo costo computacional crece de forma polinomial ($O(n^3)$ para la multiplicación de matrices).

```
(base) gus@DESKTOP-45AMP03:/mnt/c/Users/Gus/Documents/CINAT/Primer Semestre/Programacion/Tareas/Tarea2CUDA$ ./problema3
Tamano de las matrices 'N K M': 512 1024 512
Tiempo en secuencial (segundos): 0.886373
Tiempo en paralelo GM (segundos): 0.0310559 Speedup: 28.5412
Tiempo en paralelo SM (segundos): 0.0140968 Speedup: 62.8776
Resultado verificado!
(base) gus@DESKTOP-45AMP03:/mnt/c/Users/Gus/Documents/CINAT/Primer Semestre/Programacion/Tareas/Tarea2CUDA$ ./problema3
Tamano de las matrices 'N K M': 2048 1024 2048
Tiempo en secuencial (segundos): 16.6645
Tiempo en paralelo GM (segundos): 0.0709214 Speedup: 234.971
Tiempo en paralelo SM (segundos): 0.035669 Speedup: 467.197
Resultado verificado!
(base) gus@DESKTOP-45AMP03:/mnt/c/Users/Gus/Documents/CINAT/Primer Semestre/Programacion/Tareas/Tarea2CUDA$ ./problema3
Tamano de las matrices 'N K M': 2048 8192 2048
Tiempo en secuencial (segundos): 152.819
Tiempo en paralelo GM (segundos): 0.330618 Speedup: 462.222
Tiempo en paralelo SM (segundos): 0.29102 Speedup: 525.114
Resultado verificado!
```

Los códigos de todos los programas se encuentran en el archivo `T2CUDA_HernandezGustavo.zip`.