Programación y Análisis de Algoritmos | Tarea Vectores y Matrices con CUDA

Aguirre Calzadilla César Miguel

19 de noviembre de 2024

Códigos

Todo el código escrito para esta tarea será anexado en un archivo .zip. Dentro se encuentran las rutinas escritas para la tarea así como comentarios sobre las mismas.

El objetivo de la tarea es implementar un código secuencial, y otro paralelo, para resolver los siguientes problemas.

Problema 1

Dado un vector de números reales V de tamaño N, programa lo siguiente.

(a) $S_1[i] = V[i] + V[i+1]$ **para** $i = 0, .., N-2$ **con** S_1 **otro vector de tamaño** $N-1$**.**

La meta de este problema es construir un vector, digamosle S_1, a partir de la suma de dos elementos consecutivos dentro de un vector V previamente definido. Se trata de un problema matemático aparentemente sencillo y sin mucho trasfondo, pero que puede ser de utilidad para problemas donde se realicen operaciones repetitivas sobre elementos consecutivos de un mismo conjunto de datos. Esto podría verse dentro de áreas como las series temporales o suavizado de imágenes.

La diferencia clave entre la representación serial y la paralelizada por CUDA es justo el tratamiento que se le da a los datos y al cómputo de los mismos. El modelo secuencial el código se encarga de realizar cada cálculo uno tras otro, es decir, la suma de los $S_1[i]$ uno por uno tomando los valores de V.

Por otro lado, el modelo de CUDA paraleliza esa tarea y en lugar de realizar el cómputo en una sola dirección, se ejecutan diversos hilos (conocidos en la comunidad angloparlante como *threads*) simultáneamente dentro de una GPU. CUDA permite dividir el trabajo, y así cada cómputo de los $S_1[i]$ se realiza en la cantidad de hilos definida por el programa. Los hilos se definen dentro de bloques, los cuales ejecutan el cómputo a través de cada hilo contenido en el *grid* (bloque).

El tratamiento secuencial es quizás el más sencillo de determinar. Podemos ver que en nuestro código se construye a S_1 de la siguiente manera:

```
// Construccion de S1 a traves de la definicion de V

for (int i = 0; i < N - 1; i++) {
S1[i] = V[i] + V[i + 1];
}
```

Listing 1: Código secuencial en C++.

Podemos ver cómo esta línea de código es de ejecución lineal y, por lo tanto, secuencial. Visto de otra manera, cada iteración del cómputo espera a que la anterior termine de realizarse para continuar hasta terminar la construcción del nuevo vector.

Por otra parte, el modelo paralelo formado utilizando CUDA es el siguiente:

```
// Construccion de S1 a traves de la definicion de V (usando CUDA)

__global__ void calculateS1(const double* V, double* S1, int N) {
int idx = blockIdx.x * blockDim.x + threadIdx.x;
if (idx < N - 1) {
    S1[idx] = V[idx] + V[idx + 1];
}
}
```

Listing 2: Código paralelo en C++ usando CUDA.

Vale la pena detenernos un poco en esta parte. La función *kernel* **calculareS1** se ejecuta en paralelo para diversos hilos dentro del a GPU. A cada *thread* se le asigna un índice definido como **idx**. Este se calcula combinando **blockIdx.x**, o identificador de bloque, y del hilo dentro del mismo bloque, llamado **threadIdx.x**.

Además del *kernel*, un apartado fundamental para programar con CUDA es la configuración del *grid* y la cantidad de *threads* que se utilizarán para la tarea. En nuestro caso tenemos algo de la siguiente manera:

```
// Configuracion del bloque (grid) y n mero de hilos (threads)

int threadsPerBlock = 256;
int blocksPerGrid = (N + threadsPerBlock - 1) / threadsPerBlock;
calculateS1<<<blocksPerGrid, threadsPerBlock>>>(d_V, d_S1, N);
```

Listing 3: Código paralelo en C++ usando CUDA.

Veamos qué nos está diciendo esta sección del código. En primera instancia, la declaración **threadsPerBlock** define el número de hilos por bloque. Por otra parte, la sentencia **blocksPerGrid** define la cantidad de bloque a utilizarse para cubrir todos los índices del vector S_1.

Con el fin de que esta sección quede más clara para todos (y que me sirva a mí para cuando revisite estas tareas), procederé a explicar de una manera más sencilla cómo es que funciona este tipo de paralelización. Una opción es explicar el funcionamiento de **threadsPerBlock** como en una cantidad definida de personas que están realizando un proyecto juntos. Pensemos en que dicho proyecto es construir una torre cuadrada hecha de LEGO.

Al utilizar **threadsPerBlock** podemos definir a, digamos, 256 personas que quieren construir un LEGO de 50,000 piezas. Entonces, podemos dividir esas 50,000 piezas

entre los 256 participantes construyendo la torre, con las dimensiones bien definidas, a esto se le conoce como **threadsPerBlock**. Entonces, en lugar de ir poniendo una por una cada pieza, se reparten entre todos y al final se coloca cada fragmento para dar pie a la torre.

Cada hilo es una tarea a realizar, digamos, colocar una pieza en la torre. De ese modo, cada bloque es el conjunto de tareas a realizar por persona, es decir, la cantidad de piezas que construirá una persona por su cuenta. De ese modo, la GPU es la que organiza todo para que cada persona haga su parte de manera simultánea, ahorrando el tiempo de construcción.

Además de todo ello, hay una sección crucial para lograr trabajar con CUDA: la comunicación entre la CPU y la GPU.

Veámos el siguiente código:

```
// Reservar memoria en la GPU

double* d_V;
double* d_S1;
cudaMalloc(&d_V, N * sizeof(double));
cudaMalloc(&d_S1, (N - 1) * sizeof(double));

// Copiar datos de la CPU a la GPU
cudaMemcpy(d_V, V.data(), N * sizeof(double),
cudaMemcpyHostToDevice);

// Configurar dimensiones del grid y bloque
int threadsPerBlock = 256;
int blocksPerGrid = (N + threadsPerBlock - 1) / threadsPerBlock;

// Ejecutar kernel
calculateS1<<<blocksPerGrid, threadsPerBlock>>>(d_V, d_S1, N);

// Esperar a que los calculos terminen
cudaDeviceSynchronize();

// Copiar resultados de vuelta a la CPU
cudaMemcpy(S1.data(), d_S1, (N - 1) * sizeof(double),
cudaMemcpyDeviceToHost);

// Liberar memoria en la GPU
cudaFree(d_V);
cudaFree(d_S1);
}
```

Listing 4: Comunicación CPU-GPU.

Como ya adelantamos, este fragmento de código es el encargado de establecer la comunicación entre la CPU y la GPU.

Veamos qué hacen paso a paso. Primero que nada, está la sección de **cudaMalloc(&d_V, N * sizeof(double));** y **cudaMalloc(&d_S1, (N - 1) * sizeof(double));**. La primera declaración utiliza **cudaMalloc** para reservar N elementos de tipo *double* en la memoria de nuestra GPU. Esta se almacena en d_V (bajo este contexto un puntero), de ese modo, d_V apunta al inicio del arreglo creado en la GPU. De manera similar, la segunda declaración **cudaMalloc** se encarga de reservar N-1 elementos de tipo *double*

en la GPU, pero ahora en dirección al puntero d_S1. De esa manera aseguramos que los datos sean cargados a la memoria de la GPU.

Ahora bien, en cuando a **cudaMemcpy(d_V, V.data(), N * sizeof(double), cudaMemcpyHostToDevice);** . En esta línea lo que se hace es copiar los datos del vector V, dede la memoria de la CPU a la memoria reservada para la GPU, es decir, d_V.

La declaración **calculateS1«<blocksPerGrid, threadsPerBlock»>(d_V, d_S1, N);** es la que inicializa la ejecución del *kernel*. Es decir, esta líne aes la encargada de dar luz verde para que la GPU comience a trabajar de manera paralela según lo definido para cada bloque e hilo. **cudaDeviceSynchronize();** simplemente bloquea la ejecución en la CPU hasta que la GPU termine de realizar los cálculos. Esta línea es necesaria para que los resultados estén listos antes de roceder con la siguiente sección de código.

Finalmente, **cudaMemcpy(S1.data(), d_S1, (N - 1) * sizeof(double), cudaMemcpyDeviceToHost);** copia los resultados calculados para S1 desde la GPU hacia la CPU. De memoria a memoria. Los resultados deben volver a la CPU para su manipulación y visualización. **cudaFree(d_V);** y **cudaFree(d_S1);** simplemente liberan la memoria que fue reservada para la GPU, es decir, d_V y d_S1.

En pocas palabras, lo que sucede después de todo el último bloque de código es:

1. Se reserva la memoria en la GPU
2. Transfiere los datos de la CPU a la GPU.
3. Organiza las tareas para que la GPU las procese en paralelo
4. Recupera los datos desde la GPU.
5. Se limpia la memoria GPU.

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

Este problema es muy similar al anterior. Sin embargo, en esta ocasión se nos solicita calcular un nuevo vector llamado S_2 construido a partir de otro vector llamado V. Los elementos de S_2 se obtienen a través de la ecuación mostrada en el enunciado (b).

La parte secuencial se calcula simplemente como:

```
// Construcci n de S2
std::vector<double> S2(N - 2);

for (int i = 1; i < N - 1; i++) {
    S2[i - 1] = ((V[i + 1] + V[i - 1]) / 2.0);
}
```

Listing 5: Código secuencial en C++.

Por otra parte, la sección paralela para esta sección es la siguiente:

```
// Kernel para S2
__global__ void calculateS2(const double* V, double* S2, int N) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if (idx >= 1 && idx < N - 1) {
        S2[idx - 1] = (V[idx + 1] + V[idx - 1]) / 2.0;
    }
}
```

Listing 6: Código Paralelo con CUDA.

Asimismo, en esta sección tenemos las siguientes declaraciones para la interacción entre CPU y GPU:

```
// Reservar memoria de la GPU
double* d_V;
double* d_S2;
cudaMalloc(&d_V, N * sizeof(double));
cudaMalloc(&d_S2, (N-2) * sizeof(double));

// Copiar datos del host (la CPU) a la GPU
cudaMemcpy(d_V, V.data(), N * sizeof(double),
cudaMemcpyHostToDevice);

// Configurar el grid y bloque
int threadsPerBlock = 256;
int blocksPerGrid = (N + threadsPerBlock - 1) / threadsPerBlock;

// Ejecutar kernel
calculateS2<<<blocksPerGrid, threadsPerBlock>>>(d_V, d_S2, N);

// Sincronizar
cudaDeviceSynchronize();

// Copiar resutlados de la GPU al host (la CPU)
cudaMemcpy(S2.data(), d_S2, (N-2) * sizeof(double),
cudaMemcpyDeviceToHost);

// Liberar memoria de la GPU
cudaFree(d_V);
cudaFree(d_S2);
```

Listing 7: Comunucación CPU-GPU.

Problema 2 | Matrices con CUDA

Dadas dos matrices A y B de tamaño NxM con valores enteros positivos, programar lo siguiente.

(a) Sea $C_1(i,j) = A(i,j) + B(N-i-1, M-j-1)$ para $i = 0,..,N-1$ y $j = 0,..,M-1$ con C_1 otra matriz de tamaño NxM

En este caso, el problema es construir una rutina para que pueda calcular, a partir de dos matrices NxM, una tercera matriz llamada C_1 definida como:

$$C_1 = A(i,j) + B(N-i-1, M-j-1)$$

El código secuencial para ello se soporta sobre el siguiente bloque:

```
for (int i = 0; i < N; i++) {
 for (int j = 0; j < M; j++) {
     C1[i][j] = A[i][j] + B[N - i - 1][M - j - 1];
 }
}
```

Listing 8: Código secuencial para constuir C_1

Por otra parte, el enfoque paralelo usando CUDA utiliza lo siguiente:

```
// Kernel CUDA para calcular C1
__global__ void calculateC1(const int* A, const int* B, int* C1,
 int N, int M) {
     int row = blockIdx.y * blockDim.y + threadIdx.y;  // Indice de
 fila global
     int col = blockIdx.x * blockDim.x + threadIdx.x;  // Indice de
 columna global

     if (row < N && col < M) {
         C1[row * M + col] = A[row * M + col] + B[(N - row - 1) * M
 + (M - col - 1)];
     }
}
```

Listing 9: Código CUDA para constuir C_1

El código anterior es el encargado de construir a la matriz C_1 usando CUDA.

Esta rutina también implementa declaraciones para la comunicación entre la CPU y la GPU. Tal como en la manipulación de vectores que vimos anteriormente.

```

// Reservar memoria en GPU
int* d_A;
int* d_B;
int* d_C1;
cudaMalloc(&d_A, N_large * M_large * sizeof(int));
cudaMalloc(&d_B, N_large * M_large * sizeof(int));
cudaMalloc(&d_C1, N_large * M_large * sizeof(int));

// Copiar datos del host (CPU) a la GPU
```

```
11 cudaMemcpy(d_A, A_large.data(), N_large * M_large * sizeof(int),
   cudaMemcpyHostToDevice);
12 cudaMemcpy(d_B, B_large.data(), N_large * M_large * sizeof(int),
   cudaMemcpyHostToDevice);
13
14 // Configurar dimensiones del grid y threads
15 dim3 threadsPerBlock(16, 16);
16 dim3 blocksPerGrid((M_large + threadsPerBlock.x - 1) /
   threadsPerBlock.x,
17                    (N_large + threadsPerBlock.y - 1) /
   threadsPerBlock.y);
18
19 // Ejecutar kernel
20 calculateC1<<<blocksPerGrid, threadsPerBlock>>>(d_A, d_B, d_C1,
   N_large, M_large);
21
22 // Sincronizar GPU
23 cudaDeviceSynchronize();
24
25 // Copiar resultados de la GPU al host (CPU)
26 cudaMemcpy(C1_large.data(), d_C1, N_large * M_large * sizeof(int),
   cudaMemcpyDeviceToHost);
27
28 // Liberar memoria en GPU
29 cudaFree(d_A);
30 cudaFree(d_B);
31 cudaFree(d_C1);
```

Listing 10: Comunicación CPU-GPU.

En este caso, también se hacen reservas de memoria, para las variables de cada matriz A, B y C_1. En este caso, los hilos por bloque. Otra cosa que podría saltar a la vista es la declaración:

```
1 dim3 threadsPerBlock(16, 16);
2 dim3 blocksPerGrid((M_large + threadsPerBlock.x - 1) /
   threadsPerBlock.x,
3                    (N_large + threadsPerBlock.y - 1) /
   threadsPerBlock.y);
```

Aunque se utilice la variable **dim3**, no hay que confundirnos y pensar que estamos trabajando directamente con vectores de tres dimensiones. En realidad solo estamos utilizando elementos en X y Y. Aunque podríamos utilizar dicha declaración para vectores (x, y, z), **dim3** permite manejarnos solo en dos dimensiones.

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

Similar al inciso anterior, en este caso se nos pode construir una matriz nueva a partir de otras dos definidas y una multiplicación escalar también definida. La diferencia entre el desarrollo paralelo y el secuencial es que, mientras que con CUDA podemos implementar operaciones de manera paralela dentro de la GPU, en el modelo secuencial todo se realiza iteración tras iteración.

La clave en este caso, así como en el otro, es paralelizar la combinación lineal de un par de matrices para construir una nueva.

En el caso secuencial, la parte determinante del código es la siguiente:

```
1
2  for (int i = 0; i < N; i++) {
3      for (int j = 0; j < M; j++) {
4          C2[i][j] = static_cast<int>(alpha * A[i][j] + (1 - alpha) *
    B[i][j]);
5      }
6  }
```

Listing 11: Combinación lineal 2 - Secuencial

Esa sección de código es la encargada de procesar la combinación lineal de manera matemática. Podemos ver cómo es una línea de código secuencial, que no implica ninguna declaración para paralelizar el proceso. Se utiliza un ciclo for para calcular iteración por iteración.

Como hemos visto en ejemplos anteriores, podemos paralelizar utilizando CUDA:

```
1
2  // Funci n de CUDA para calcular la matriz C2
3  __global__ void calcular_C2_kernel(int* A, int* B, int* C2, double
    alpha, int N, int M) {
4      int i = blockIdx.x * blockDim.x + threadIdx.x;
5      int j = blockIdx.y * blockDim.y + threadIdx.y;
6
7      if (i < N && j < M) {
8          C2[i * M + j] = static_cast<int>(alpha * A[i * M + j] + (1
    - alpha) * B[i * M + j]);
9      }
10 }
```

Listing 12: Combinación lineal 2 - CUDA

Este bloque del código se encarga de paralelizar el proceso de cálculo de la combinación lineal. Podemos ver cómo se le asignan threads a cada elemento (i, j) de nuestras matrices y cómo se realiza la operación utilizando CUDA.

De igual forma, y como en los códigos anteriores, tenemos declaraciones para la comunicación entre la CPU y la GPU para realizar la tarea:

```
1
2  // Asignar memoria en GPU
3  cudaMalloc(&d_A, N * M * sizeof(int));
4  cudaMalloc(&d_B, N * M * sizeof(int));
```

```
5  cudaMalloc(&d_C2, N * M * sizeof(int));
6
7  // Copiar datos desde la memoria de host a la memoria de GPU
8  cudaMemcpy(d_A, A[0].data(), N * M * sizeof(int),
   cudaMemcpyHostToDevice);
9  cudaMemcpy(d_B, B[0].data(), N * M * sizeof(int),
   cudaMemcpyHostToDevice);
10
11 // Definir el tama o de los bloques y la cuadr cula
12 dim3 blockSize(16, 16);
13 dim3 gridSize((N + blockSize.x - 1) / blockSize.x, (M + blockSize.y
    - 1) / blockSize.y);
14
15 // Ejecutar el kernel CUDA
16 calcular_C2_kernel<<<gridSize, blockSize>>>(d_A, d_B, d_C2, alpha,
   N, M);
17
18 // Esperar que todos los hilos terminen
19 cudaDeviceSynchronize();
20
21 // Copiar los resultados de la memoria de GPU a la memoria de host
   (CPU)
22 cudaMemcpy(C2[0].data(), d_C2, N * M * sizeof(int),
   cudaMemcpyDeviceToHost);
23
24 // Liberar memoria de la GPU
25 cudaFree(d_A);
26 cudaFree(d_B);
27 cudaFree(d_C2);
```

Listing 13: Comunicación CPU-GPU

Problema 3 | Más Matrices

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.

Creo que antes de mostrar las soluciones, vale la pena mencionar las diferencias entre el cómputo usando Memoria Global (GM, por sus siglas en inglés), y usando Memoria Compartida (SM, por sus siglas en inglés).

Cuando estamos trabajando con CUDA, hay dos tipos diferentes de memoria en la arquitectura de la GPU. Uno es conocido como *Global Memory* (GM), y se trata de un tipo de memoria disponible para todos los hilos de todos los bloques de un mismo bloque (grid). Es decir, es accesible para cualquier thread. En general, se considera que es un tipo de memoria con mayor latencia, i.e. el tiempo que la GPU tarda en manipular los vectores y producir un output.

Por otra parte, la memoria compartida se limita a los hilos dentro de un mismo grid. De acuerdo con información del blog para desarrolladores de NVIDIA, esta es, en teoría, la técnica más rápida de las dos. En una de las entradas del blog mencionado se indica que la latencia de la memoria compartida es al menos 100 más baja que la de la memoria global. Así, la SM permite la rápida comunicación entre hilos dentro de un blqoue, algo realmente eficiente para optimizar cálculos paralelos.

(a) La multiplicación de las matrices A y B usando memoria global (GM).

La multiplicación de matrices fue un problema que resolvimos la tarea pasada. El fragmento de código secuencial que establece la multiplicación utilizando sub bloques es el siguiente:

```
1  // Multiplicaci n de matrices en bloques
2  for (int i = 0; i < N; i += BS) {
3      for (int j = 0; j < M; j += BS) {
4          for (int k = 0; k < K; k += BS) {
5              // Multiplicaci n dentro de cada bloque
6              for (int ii = i; ii < std::min(i + BS, N); ii++) {
7                  for (int jj = j; jj < std::min(j + BS, M); jj++) {
8                      for (int kk = k; kk < std::min(k + BS, K); kk
   ++) {
9                          C[ii * M + jj] += A[ii * K + kk] * B[kk * M
    + jj];
10                     }
11                 }
12             }
13         }
14     }
15 }
```

Listing 14: Multiplicación de matrices - Serial

Podemos ver cómo es un ciclo de ciclos for, que trabajan iterativamente. Por otra parte, la pararelización utilizando CUDA se ve de la siguiente manera:

```
1
2  // Funcion para realizar la multiplicacion de matrices usando CUDA
3  __global__ void multiplicarMatricesCUDA(const double* A, const
   double* B, double* C, int N, int K, int M, int BS) {
```

```
    // Indices de fila y columna de la matriz C
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;

    if (row < N && col < M) {
        double sum = 0.0;
        for (int k = 0; k < K; ++k) {
            sum += A[row * K + k] * B[k * M + col];
        }
        C[row * M + col] = sum;
    }
}
```

Listing 15: Multiplicación de matrices - CUDA

De igual forma, tenemos las líneas para comunicarnos entre CPU y GPU:

```
// Asignar memoria en la GPU
double *d_A, *d_B, *d_C;
cudaMalloc(&d_A, N * K * sizeof(double));
cudaMalloc(&d_B, K * M * sizeof(double));
cudaMalloc(&d_C, N * M * sizeof(double));

// Copiar matrices A y B desde la memoria principal a la GPU
cudaMemcpy(d_A, A.data(), N * K * sizeof(double),
cudaMemcpyHostToDevice);
cudaMemcpy(d_B, B.data(), K * M * sizeof(double),
cudaMemcpyHostToDevice);

// Definir el tama o del bloque y la grilla
dim3 blockDim(BS, BS);
dim3 gridDim((M + BS - 1) / BS, (N + BS - 1) / BS);

// Medir el tiempo de ejecucion de la multiplicacion
auto start = std::chrono::high_resolution_clock::now();

// Llamar al kernel de CUDA para multiplicar las matrices
multiplicarMatricesCUDA<<<gridDim, blockDim>>>(d_A, d_B, d_C, N, K,
 M, BS);

// Esperar a que la GPU termine
cudaDeviceSynchronize();

auto end = std::chrono::high_resolution_clock::now();
std::chrono::duration<double> duration = end - start;  // Calcular
la duracion

// Copiar el resultado de C desde la GPU a la memoria principal
cudaMemcpy(C.data(), d_C, N * M * sizeof(double),
cudaMemcpyDeviceToHost);

// Liberar memoria de la GPU
cudaFree(d_A);
cudaFree(d_B);
cudaFree(d_C);
```

Listing 16: Comunicación CPU-GPU

(b) La multiplicación de las matrices A y B usando memoria compartida (SM).

El código escencial para lograr memoria compartida y multiplicar matrices se ve así:

```
1
2  // Funci n para realizar la multiplicaci n de matrices usando
   CUDA con memoria compartida
3__global__ void multiplicarMatricesCUDA(const double* A, const double*
   B, double* C, int N, int K, int M, int BS) {
4  //   ndices  de fila y columna de la matriz C
5  int row = blockIdx.y * blockDim.y + threadIdx.y;
6  int col = blockIdx.x * blockDim.x + threadIdx.x;
7
8  // Definir memoria compartida para submatrices A y B
9  __shared__ double s_A[32][32];  // Tama o de submatriz A por
   bloque
10 __shared__ double s_B[32][32];  // Tama o de submatriz B por
   bloque
11
12 double sum = 0.0;
13
14 for (int k = 0; k < (K + BS - 1) / BS; ++k) {
15     // Cargar los elementos de la matriz A y B en memoria
   compartida
16     if (row < N && k * BS + threadIdx.x < K) {
17         s_A[threadIdx.y][threadIdx.x] = A[row * K + k * BS +
   threadIdx.x];
18     } else {
19         s_A[threadIdx.y][threadIdx.x] = 0.0;
20     }
21
22     if (col < M && k * BS + threadIdx.y < K) {
23         s_B[threadIdx.y][threadIdx.x] = B[(k * BS + threadIdx.y) *
   M + col];
24     } else {
25         s_B[threadIdx.y][threadIdx.x] = 0.0;
26     }
27
28     // Sincronizar los hilos para asegurar que la carga de datos en
    memoria compartida termine
29     __syncthreads();
30
31     // Realizar la multiplicaci n de submatrices
32     for (int n = 0; n < BS; ++n) {
33         sum += s_A[threadIdx.y][n] * s_B[n][threadIdx.x];
34     }
35
36     // Sincronizar antes de cargar los siguientes bloques
37     __syncthreads();
38 }
39
40 // Almacenar el resultado en la matriz C
41 if (row < N && col < M) {
42     C[row * M + col] = sum;
43 }
```

```
}
```

Listing 17: Multiplicación de matrices con memoria compartida.

Podemos ver como aquí, a diferencia del caso de GM, la memoria compartida se define con la palabra clave **__shared__**, y se utiliza dentro de cada bloque de hilos para almacenar temporalmente los datos que se pasarán por los hilos dentro de dicho grid. También podemos ver la presencia de **__syncthreads()**, función que se utiliza para sincronizar a los hilos después de cargar los datos en la SM. Esa declaración es esencial pues es la encargada de verificar que todos los hilos dentro de un bloque puedan acceder a los datos de la SM antes de entrar de lleno a los cálculos.

Simado a ello, podemos ver que los hilos cargan parte de las matrices A y B en la SM con un tamaño de 32×32. De esa manera, se permite que cada hilo acceda de manera sencilla y eficiente a los datos requeridos para nuestro computo.

De una manera muy similar al caso anterior se implementa la comunicación entre CPU y GPU:

```
1
2  // Llenar matrices A y B con valores aleatorios
3  llenarMatrizConAleatorios(A, N, K, 0.0, 10.0);  // Valores
   aleatorios entre 0.0 y 10.0
4  llenarMatrizConAleatorios(B, K, M, 0.0, 10.0);  // Valores
   aleatorios entre 0.0 y 10.0
5
6  // Asignar memoria en la GPU
7  double *d_A, *d_B, *d_C;
8  cudaMalloc(&d_A, N * K * sizeof(double));
9  cudaMalloc(&d_B, K * M * sizeof(double));
10 cudaMalloc(&d_C, N * M * sizeof(double));
11
12 // Copiar matrices A y B desde la memoria principal a la GPU
13 cudaMemcpy(d_A, A.data(), N * K * sizeof(double),
   cudaMemcpyHostToDevice);
14 cudaMemcpy(d_B, B.data(), K * M * sizeof(double),
   cudaMemcpyHostToDevice);
15
16 // Definir el tama o del bloque y la grilla
17 dim3 blockDim(BS, BS);
18 dim3 gridDim((M + BS - 1) / BS, (N + BS - 1) / BS);
19
20 // Medir el tiempo de ejecuci n de la multiplicaci n
21 auto start = std::chrono::high_resolution_clock::now();
22
23 // Llamar al kernel de CUDA para multiplicar las matrices
24 multiplicarMatricesCUDA<<<gridDim, blockDim>>>(d_A, d_B, d_C, N, K,
    M, BS);
25
26 // Esperar a que la GPU termine
27 cudaDeviceSynchronize();
28
29 auto end = std::chrono::high_resolution_clock::now();
30 std::chrono::duration<double> duration = end - start;  // Calcular
   la duraci n
31
32 // Copiar el resultado de C desde la GPU a la memoria principal
```

```
33  cudaMemcpy(C.data(), d_C, N * M * sizeof(double),
    cudaMemcpyDeviceToHost);
34
35  // Liberar memoria de la GPU
36  cudaFree(d_A);
37  cudaFree(d_B);
38  cudaFree(d_C);
```

Listing 18: Comunicación CPU-GPU

(c) Evaluar el tiempo de procesamiento entre las versiones Serial, Paralelo usando GM y Paralelo usando SM, para diferentes tamaños de las matrices, por ejemplo:

$512 \times 1024, 1024 \times 512$

- Modelo Secuencial: 7.98478 s
- Memoria Global: 0.010536 s
- Memoria Compartida: 0.0106128 s

$2048 \times 1024, 1024 \times 2048$

- Modelo Secuencial: 44.2253 s
- Memoria Global: 0.116366 s
- Memoria Compartida: 0.122704 s

$2048 \times 8192, 8192 \times 2048$

- Modelo Secuencial: 288.717 s
- Memoria Global: 0.785954 s
- Memoria Compartida: 0.776135 s