UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Escola Tècnica Superior d'Enginyeria Informàtica

etsinf

Práctica 2

Parte 1: Compilando Programas en C

Máster Universitario en Ingeniería de Computadores y Redes

1 de diciembre de 2025

Pablo González Segarra

Índice

Ejercicio 1 y 2

Debido a la similitud entre los ejercicios 1 y 2, se presentan juntos en esta sección. A continuación se muestra el código modificado necesario para ejecutar el programa en la GPU utilizando CUDA y comprobar que los resultados son correctos.

```
#define N 100000000

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

__global__ void vector_add(float *out, float *a, float *b, int n) {
    for(int i = 0; i < n; i++){
        out[i] = a[i] + b[i];
    }
}

int main(){
    float *a, *b, *out;

    // Allocate memory
    a   = (float*)malloc(sizeof(float) * N);
    b   = (float*)malloc(sizeof(float) * N);
    out = (float*)malloc(sizeof(float) * N);

    // Initialize array
    for(int i = 0; i < N; i++){
        a[i] = 1.0f; b[i] = 2.0f;
    }

    float *a_cuda = NULL, *b_cuda = NULL, *out_cuda = NULL;

    // Allocate device memory
    cudaMalloc((void**)&a_cuda, sizeof(float) * N);
    cudaMalloc((void**)&b_cuda, sizeof(float) * N);
    cudaMalloc((void**)&out_cuda, sizeof(float) * N);

    // Copy inputs to device
    cudaMemcpy(a_cuda, a, sizeof(float) * N, cudaMemcpyHostToDevice);
    cudaMemcpy(b_cuda, b, sizeof(float) * N, cudaMemcpyHostToDevice);

    // Main function
    vector_add<<<1,1>>>(out_cuda, a_cuda, b_cuda, N);

    cudaMemcpy(out, out_cuda, sizeof(float) * N,
    cudaMemcpyDeviceToHost);

    cudaDeviceSynchronize();

    // Verify result
    const float MAX_ERR = 1e-6f;
    for(int i = 0; i < N; i++){
        if(fabsf(out[i] - 3.0f) > MAX_ERR){
            printf("Error at index %d: %f (diff=%f)\n", i, out[i],
    fabsf(out[i] - 3.0f));
```

```
            return -1;
        }
    }

    printf("Success! All values are correct.\n");

    // Free memory
    cudaFree(a_cuda);
    cudaFree(b_cuda);
    cudaFree(out_cuda);

    free(a);
    free(b);
    free(out);
}
```

Listing 1: Código CUDA modificado para ejecutar en GPU

Ejercicio 3

El código modificado para poder obtener los tiempos de ejecución es el siguiente:

```
#define N 100000000

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

__global__ void vector_add(float *out, float *a, float *b, int n) {
    for(int i = 0; i < n; i++){
        out[i] = a[i] + b[i];
    }
}

int main(){
    float *a, *b, *out;

    // Allocate memory
    a   = (float*)malloc(sizeof(float) * N);
    b   = (float*)malloc(sizeof(float) * N);
    out = (float*)malloc(sizeof(float) * N);

    // Initialize array
    for(int i = 0; i < N; i++){
        a[i] = 1.0f; b[i] = 2.0f;
    }

    float *a_cuda = NULL, *b_cuda = NULL, *out_cuda = NULL;

    // Create events for timing
    cudaEvent_t start, end;
    cudaEventCreate(&start);
    cudaEventCreate(&end);

```

```
34      // Start timing
35      cudaEventRecord(start);
36
37      // Allocate device memory
38      cudaMalloc((void**)&a_cuda, sizeof(float) * N);
39      cudaMalloc((void**)&b_cuda, sizeof(float) * N);
40      cudaMalloc((void**)&out_cuda, sizeof(float) * N);
41
42      // Copy inputs to device
43      cudaMemcpy(a_cuda, a, sizeof(float) * N, cudaMemcpyHostToDevice);
44      cudaMemcpy(b_cuda, b, sizeof(float) * N, cudaMemcpyHostToDevice);
45
46      // Main function
47      vector_add<<<1,1>>>(out_cuda, a_cuda, b_cuda, N);
48
49      cudaMemcpy(out, out_cuda, sizeof(float) * N,
        cudaMemcpyDeviceToHost);
50
51      // End timing
52      cudaEventRecord(end);
53      cudaEventSynchronize(end);
54      float milliseconds = 0;
55      cudaEventElapsedTime(&milliseconds, start, end);
56      printf("Time elapsed: %f ms\n", milliseconds);
57
58      // Verify result
59      const float MAX_ERR = 1e-6f;
60      for(int i = 0; i < N; i++){
61          if(fabsf(out[i] - 3.0f) > MAX_ERR){
62              printf("Error at index %d: %f (diff=%f)\n", i, out[i],
        fabsf(out[i] - 3.0f));
63              return -1;
64          }
65      }
66
67      printf("Success! All values are correct.\n");
68
69      // Free memory
70      cudaFree(a_cuda);
71      cudaFree(b_cuda);
72      cudaFree(out_cuda);
73
74      free(a);
75      free(b);
76      free(out);
77  }
```

Listing 2: Código CUDA modificado para medir tiempos de ejecución

Se ha ejecutado 5 veces para evitar variaciones ajenas al rendimiento del código. El tiempo de La ejecución tiene en cuenta tanto la transferencia de datos entre host y device como la ejecución del kernel.

Resultado de las ejecuciones:

1. 5887.4 ms
2. 5864.1 ms

3. 5918.6 ms
4. 5856.9 ms
5. 5868.1 ms

Siendo el mejor tiempo 5856.9 ms.

Ejercicio 4

Para comparar la aceleración obtenida al ejecutar el código en la GPU frente a la CPU, primero se ha compilado y ejecutado la versión original en CPU, obteniendo un tiempo mejor de 992.6 ms.

Para obtener los tiempos de ejecución en GPU con distintas configuraciones, número de bloques y hilos por bloque, se ha modificado el código del ejercicio 3 para variar estos parámetros. Todas las configuraciones se han ejecutado múltiples veces y se ha tomado el mejor tiempo de ejecución. En la tabla 1 se muestran los resultados obtenidos.

Versión	**Tiempo de ejecución (ms)**	**Speedup GPU vs CPU**
CPU (original)	992.6	1.00
1 thread / block	532.3	1.86
16 threads / block	438.2	2.27
32 threads / block	431.8	2.30
256 threads / block	428.4	2.32

Tabla 1: Tiempos de ejecución y speedup para distintas configuraciones de hilos/ bloque

Como se puede observar, la paralelización en GPU ofrece una mejora significativa en el tiempo de ejecución en comparación con la versión en CPU. El mejor rendimiento se obtiene con 256 hilos por bloque, logrando un speedup de aproximadamente 2.32 veces respecto a la ejecución en CPU. Sin embargo, la mejora a partir de 16 hilos por bloque es marginal.

A continuación se muestra el código modificado para variar el número de bloques e hilos por bloque:

```
#define N 100000000

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

__global__ void vector_add(float *out, float *a, float *b, int n) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i < n) {
        out[i] = a[i] + b[i];
    }
}

int main(int argc, char **argv){

```

```
17      float *a, *b, *out;
18
19      // Allocate memory
20      a   = (float*)malloc(sizeof(float) * N);
21      b   = (float*)malloc(sizeof(float) * N);
22      out = (float*)malloc(sizeof(float) * N);
23
24      // Initialize array
25      for(int i = 0; i < N; i++){
26          a[i] = 1.0f; b[i] = 2.0f;
27      }
28
29      float *a_cuda = NULL, *b_cuda = NULL, *out_cuda = NULL;
30
31      // Create events for timing
32      cudaEvent_t start, end;
33      cudaEventCreate(&start);
34      cudaEventCreate(&end);
35
36      // Start timing
37      cudaEventRecord(start);
38
39      // Allocate device memory
40      cudaMalloc((void**)&a_cuda, sizeof(float) * N);
41      cudaMalloc((void**)&b_cuda, sizeof(float) * N);
42      cudaMalloc((void**)&out_cuda, sizeof(float) * N);
43
44      // Copy inputs to device
45      cudaMemcpy(a_cuda, a, sizeof(float) * N, cudaMemcpyHostToDevice);
46      cudaMemcpy(b_cuda, b, sizeof(float) * N, cudaMemcpyHostToDevice);
47
48      // Main function
49      // threadsPerBlock is taken from argv[1] (assume valid integer
        provided)
50      int threadsPerBlock = atoi(argv[1]);
51      int blocks = (N + threadsPerBlock - 1) / threadsPerBlock;
52      // Print configuration
53      printf("Using %d blocks of %d threads\n", blocks, threadsPerBlock
        );
54      vector_add<<<blocks, threadsPerBlock>>>(out_cuda, a_cuda, b_cuda,
        N);
55
56      cudaMemcpy(out, out_cuda, sizeof(float) * N,
        cudaMemcpyDeviceToHost);
57
58      // End timing
59      cudaEventRecord(end);
60      cudaEventSynchronize(end);
61      float milliseconds = 0;
62      cudaEventElapsedTime(&milliseconds, start, end);
63      printf("Time elapsed: %f ms\n", milliseconds);
64
65      // Verify result
66      const float MAX_ERR = 1e-6f;
67      for(int i = 0; i < N; i++){
68          if(fabsf(out[i] - 3.0f) > MAX_ERR){
69              printf("Error at index %d: %f (diff=%f)\n", i, out[i],
        fabsf(out[i] - 3.0f));
```

```
70                 return -1;
71             }
72         }
73
74         printf("Success! All values are correct.\n");
75
76         // Free memory
77         cudaFree(a_cuda);
78         cudaFree(b_cuda);
79         cudaFree(out_cuda);
80
81         free(a);
82         free(b);
83         free(out);
84 }
```

Listing 3: Código CUDA modificado para variar bloques e hilos por bloque

Ejercicio 5

En este ejercicio se ha creado un kernel para la función `void heavy_cpu(float* data, int n)`, el código. se puede ver a continuación:

```
1  __global__ void heavy_cpu(float* data, int n) {
2      int i = blockIdx.x * blockDim.x + threadIdx.x;
3
4      if (i < n) {
5          float x = data[i];
6          for (int j = 0; j < 10000; j++) {
7              x = sinf(x) * 1.00001f + cosf(x) * 0.99999f;
8          }
9          data[i] = x;
10     }
11 }
```

Listing 4: Código CUDA del kernel para la función heavy_cpu

Ejercicio 6

Para este ejercicio se ha usado la misma metodología que en los ejercicios anteriores, pero esta vez se ha medido solo el tiempo de ejecución del kernel, sin contar las transferencias de datos entre host y device. Se ha ejecutado la función en CPU y GPU, obteniendo los siguientes tiempos:

- CPU: 224,69 s
- GPU: 10,33 s

Suponiendo que la función `heavy_cpu` contiene 5 operaciones de coma flotante por iteración del bucle, y considerando que el bucle se ejecuta 10.000 veces por cada elemento del array, podemos calcular el número total de operaciones de coma flotante realizadas

durante la ejecución de la función. Dado que el array tiene un tamaño de N = 1.000.000, el número total de operaciones de coma flotante es:

Total de operaciones = $N \times 10{,}000 \times 5 = 1{,}000{,}000 \times 10{,}000 \times 5 = 5 \times 10^{10}$ operaciones

Con estos datos, podemos calcular el rendimiento en GFLOPS para ambas ejecuciones:

- CPU: $\frac{5 \times 10^{10} \text{ operaciones}}{224{,}69 \text{ s}} \approx 0{,}2226$ GFLOPS
- GPU: $\frac{5 \times 10^{10} \text{ operaciones}}{10{,}33 \text{ s}} \approx 4{,}8385$ GFLOPS

Estos resultados muestran una mejora significativa en el rendimiento al ejecutar la función en la GPU en comparación con la CPU. Obteniendo una aceleración de aproximadamente 21.6 veces.

Ejercicio 7

Cada hilo ejecuta el procesamiento de un elemento del array, es decir, obtiene su índice, lee `data[i]`, aplica el bucle de 10.000 iteraciones con el grueso de la carga computacional, y escribe el resultado de vuelta.

En cuanto al paralelismo, se podría mejorar ligeramente retocando algunos detalles como los parámetros de threads per block o blocks, pero en general no es posible aumentar significativamente el paralelismo porque cada iteración del bucle interno depende del valor anterior, por lo que esta dependencia impide paralelizar el bucle.

Ejercicio 8