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: $\frac{5\times10^{10} \text{ operaciones}}{224{,}69 \text{ s}} \approx 0{,}2226$ GFLOPS
- GPU: $\frac{5\times10^{10} \text{ operaciones}}{10{,}33 \text{ s}} \approx 4{,}8385$ 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

En primer lugar, se ha medido el rendimiento ejecutando la funcion en la CPU. Para esto se ha usado N = 1000000, para permitir que el procesador termine en un tiempo razonable. Se ha obtenido un tiempo de ejecución de 324 segundos, lo que da un rendimiento de 16,158 GFLOPS calculado de la siguiente manera:

$$\text{FLOPS} = \frac{2N^2}{\text{tiempo}} = \frac{2 \cdot (10^6)^2}{324} \approx 6,158 \text{ GFLOPS}$$

La primera optimización que se ha realizado es el uso de la GPU, por lo que se ha implementado el kernel siguiente:

```
__global__ void vector_pro(float *out, float *a, float *b, int n) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i < n) {
        out[i] = 0;
        for (int j= 0; j<n; j++){
            out[i] += a[i]*b[j];
        }
    }
}
```

Listing 5: Código CUDA del kernel para la función vector_pro

Con la que se obtiene un timepo de ejecución 4389 ms, para 128 threads per block, lo que da un rendimiento 455 GFLOPS, calculado de la siguiente manera:

$$\text{FLOPS} = \frac{2N^2}{\text{tiempo}} = \frac{2 \cdot (10^6)^2}{4{,}389} \approx 455 \text{ GFLOPS}$$

y una aceleración de 73.99x respecto a la CPU.

Analizando algebráicamente la operación, se puede observar que la funcion ejecuta la siguiente operación matemática:

$$out[i] = \sum_{j=0}^{N-1} a[i] * b[j]$$

esta operacion se puede simplificar algebráicamente precalculando la suma de b, lo que reduce la complejidad de la operación de $O(N^2)$ a $O(2N)$, de forma que obtenemos la siguiente operacion:

$$out[i] = a[i] * \left( \sum_{j=0}^{N-1} b[j] \right)$$

Lo que permite implementar un kernel mucho más eficiente:

```
__global__ void vector_pro(float *out, float *a, float sum_b, int n)
    {
    int i = blockIdx.x * blockDim.x + threadIdx.x;

    if (i < n) {
        out[i] = a[i] * sum_b;
    }
}
```

Listing 6: Código CUDA del kernel optimizado para la función vector_pro

La suma de b se calcula previamente en la CPU y se pasa como parámetro al kernel. De esta forma se obtiene un tiempo de ejecución de 0.9048 ms, para 256 threads per block, lo que representa una aceleración de 359122.17x respecto a la CPU, lo que es realmente sorprendente. Sin embargo al calcular el rendimiento en GFLOPS obtenemos un valor de 22,9 GFLOPS, que es mucho menor que el obtenido en la versión anterior. El valor se ha calculado de la siguiente manera:

$$\text{FLOPS} = \frac{2N}{\text{tiempo}} = \frac{2 \cdot (10^6)}{0{,}0009048} \approx 22,9 \text{ GFLOPS}$$

Mi suposicion sobre el valore bajo de GFLOPS es que la GPU no está siendo suficientemente utilizada, ya que la operación es muy simple y no requiere muchos recursos computacionales (o lo estoy calculando mal, que siempre es una opcion :)). Por lo tanto, aunque el tiempo de ejecución es muy bajo, el número de operaciones por segundo no es tan alto como en la versión anterior.

Puesto que utilizar valores de N=1.000.000 es demasiado bajo para la optimizacion los valores tan bajos pueden estar siendo afectados por el ruido de la medicion de forma

significativa, por lo que se ha aumentado el valor de N a 100.000.000. Adicionalmente, es evidente que paralelizar la suma de b en la GPU puede aportar una mejora adicional, por lo que se ha implementado una version que paraleliza la suma de b en la GPU.

Se han obtenido los siguientes resultados para 256 threads per block(la mejor configuración):

- Versión secuencial: 133.5 ms
- Versión paralelizada (2 threads [1]): 69.5 ms

Vemos como la versión paralelizada ofrece una aceleración de 1.916x respecto a la versión que realiza la suma de b de forma secuencial en la CPU. Esto hace pensar que el cuello de botella en la versión optimizada es la suma de b, por lo que se hize un análisis más detallado de los tiempos de ejecución, cullos resultados se muestran a continuación:

| **Fase** | **Tiempo (ms)** | **Porcentaje del total** |
|---|---|---|
| Suma paralela (CPU) | 69.15 | 95.2 % |
| Ejecución kernel (GPU) | 3.50 | 4.8 % |
| **Total** | 72.66 | 100 % |

Tabla 2: Desglose de tiempos de ejecución para N=100.000.000, suma paralela en CPU y kernel en GPU.

De todos los resultado obtenidos, es trivial observar que la suma de b es el cuello de botella y que paralelizarla en la CPU aporta una mejora significativa. Por lo que se podría intentar paralelizar la suma de b en la GPU para intentar mejorar aún más el rendimiento.

Para eso se ha implementado el siguiente codigo:

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>

// Kernel for sum: each thread sums a portion of the vector and
    accumulates with atomicAdd
__global__ void sum_kernel(float *b, double *result, int n, int
    elements_per_thread) {
    int tid = blockIdx.x * blockDim.x + threadIdx.x;
    int start = tid * elements_per_thread;

    // Early exit if this thread has no work to do
    if (start >= n) return;

    int end = start + elements_per_thread;
```

[1]Se han usado 2 threads en la CPU para paralelizar la suma de b, ya que el equipo solo dispone de 2 núcleos físicos. Se han realizando experimentos con 4, 8 y 12 threads, pero el rendimiento empeoraba debido al overhead de gestión de los hilos.

```
17      if (end > n) end = n;
18
19      float sum = 0.0;
20      for (int i = start; i < end; i++) {
21          sum += (float)b[i];
22      }
23
24      // Accumulate partial sum in the global result using atomicAdd
25      atomicAdd(result, sum);
26  }
27
28  // Main kernel: out[i] = a[i] * sum(b)
29  __global__ void vector_pro(float *out, float *a, double* sum_b, int n
        ) {
30      int i = blockIdx.x * blockDim.x + threadIdx.x;
31
32      if (i < n) {
33          out[i] = a[i] * (float)(*sum_b);
34      }
35  }
36
37  int main(int argc, char **argv){
38
39      float *a, *b, *out;
40
41      // Allocate memory
42      a   = (float*)malloc(sizeof(float) * N);
43      b   = (float*)malloc(sizeof(float) * N);
44      out = (float*)malloc(sizeof(float) * N);
45
46      // Initialize array
47      for(int i = 0; i < N; i++){
48          a[i] = 1.0f; b[i] = 2.0f;
49      }
50
51      float *a_cuda = NULL, *b_cuda = NULL, *out_cuda = NULL;
52
53      // Allocate device memory
54      cudaMalloc((void**)&a_cuda, sizeof(float) * N);
55      cudaMalloc((void**)&b_cuda, sizeof(float) * N);
56      cudaMalloc((void**)&out_cuda, sizeof(float) * N);
57
58      // Copy inputs to device
59      cudaMemcpy(a_cuda, a, sizeof(float) * N, cudaMemcpyHostToDevice);
60      cudaMemcpy(b_cuda, b, sizeof(float) * N, cudaMemcpyHostToDevice);
61
62      // threadsPerBlock for vector_pro from argv[1]
63      int threadsPerBlock = atoi(argv[1]);
64
65      // threadsPerBlock for sum kernel from argv[2]
66      int sumThreadsPerBlock = atoi(argv[2]);
67
68      // Calculate total threads and elements per thread for sum
69      int sum_blocks = atoi(argv[3]);
70      int total_sum_threads = sumThreadsPerBlock * sum_blocks;
71      int elements_per_thread = (N + total_sum_threads - 1) /
        total_sum_threads;
72
```

```
73      int blocks = (N + threadsPerBlock - 1) / threadsPerBlock;
74
75      // Allocate memory for the result (single double) on GPU
76      double *result_cuda = NULL;
77      cudaMalloc((void**)&result_cuda, sizeof(double));
78      cudaMemset(result_cuda, 0, sizeof(double));
79
80      // Print configuration
81      printf("Using %d blocks of %d threads for vector_pro\n", blocks,
        threadsPerBlock);
82      printf("Using %d blocks of %d threads for sum kernel\n",
        sum_blocks, sumThreadsPerBlock);
83      printf("Elements per thread: %d\n", elements_per_thread);
84
85      // Create events for timing
86      cudaEvent_t start, end;
87      cudaEventCreate(&start);
88      cudaEventCreate(&end);
89
90      // Start timing
91      cudaEventRecord(start);
92
93      // Compute sum of b on GPU using atomicAdd
94      sum_kernel<<<sum_blocks, sumThreadsPerBlock>>>(b_cuda,
        result_cuda, N, elements_per_thread);
95
96      printf("Sum kernel launched\n");
97
98      // Compute out[i] = a[i] * sum_b
99      vector_pro<<<blocks, threadsPerBlock>>>(out_cuda, a_cuda,
        result_cuda, N);
100
101     // End timing
102     cudaEventRecord(end);
103     cudaEventSynchronize(end);
104     float milliseconds = 0;
105     cudaEventElapsedTime(&milliseconds, start, end);
106     printf("Time elapsed: %f ms\n", milliseconds);
107
108     cudaMemcpy(out, out_cuda, sizeof(float) * N,
        cudaMemcpyDeviceToHost);
109
110     // Verify first result: out[0] = a[0] * sum(b)
111     // Copy result back to host to verify
112     double sum_b = 0.0;
113     for (long long i = 0; i < (long long)N; i++) {
114         sum_b += b[i];
115     }
116     // Compare with gpu calculated value
117     double gpu_sum_b;
118     cudaMemcpy(&gpu_sum_b, result_cuda, sizeof(double),
        cudaMemcpyDeviceToHost);
119     printf("CPU sum: %f, GPU sum: %f\n", sum_b, gpu_sum_b);
120
121     float expected = a[0] * sum_b;
122     printf("Expected: %f, Got: %f\n", expected, out[0]);
123     if (fabsf(out[0] - expected) / expected > 1e-6f) {
124         printf("Error: result mismatch!\n");
```

```
125             return -1;
126         }
127         printf("Success!\n");
128
129         // Free memory
130         cudaFree(a_cuda);
131         cudaFree(b_cuda);
132         cudaFree(out_cuda);
133         cudaFree(result_cuda);
134
135         free(a);
136         free(b);
137         free(out);
138     }
```

Listing 7: Código CUDA para paralelizar la suma de b en GPU

Como en los otros casos se han probado un amplio rango de configuraciones, obteniendo el mejor resultado con 2048 bloques de sumas parciales y 256 threads per block, lo que nos da 191 elementos por hilo. Se han obtenidos los siguientes resultados:

| **Configuración** | **Tiempo (ms)** | **GFLOPS** |
|---|---|---|
| 2048 bloques, 256 threads/block, 191 elem/thread | 8.645 | 23,168 |
| **Speedup** | **Respecto a** | **Factor** |
| Suma secuencial en CPU | 133.53 ms | 15.44x |
| Suma paralelizada en CPU | 69.15 ms | 8.00x |

Tabla 3: Mejor configuración para la suma en GPU con atomicAdd, N=100.000.000