Instituto Superior Técnico

MEEC

2º Semestre 2014/2015

Arquitecturas Avançadas de Computadores

Projecto 3

Paralelização e aceleração de um programa em CUDA

Trabalho realizado por:
João Baúto Nº 72856
João Severino Nº 73608

Professor: Leonel Sousa

Índice

Capítulo 1 | Introdução

O objectivo deste terceiro trabalho de laboratório é a aceleração de um algoritmo de *smoothing* utilizando as propriedades de computação paralela em GPUs.

Para tal recorreu-se à plataforma CUDA^{TM}que tira proveito das unidades de processamento gráfico (GPUs) da NVIDIA^{TM}.

Procura-se tirar proveito da arquitectura dos GPUs para maximizar o desempenho de um algoritmo de *smoothing* recorrendo ao Paralelismo de Dados.

1.1 Algoritmo

O algoritmo de *smoothing* é um algoritmo bastante utilizado no processamento de imagem bem como análise estatística com o objectivo de criar uma função aproximada dos dados de entrada mantendo pontos importantes nos dados enquanto que reduz qualquer tipo de ruído associado aos dados.

Para modelar os dados o algoritmo observa um ponto individual bem como os imediatamente adjacentes e caso o ponto a observar seja maior em valor que os seus adjacentes é suavizado diminuindo o seu valor. Se o valor for menor que os seus adjacentes é entre elevado o seu valor em relação à sua volta. Isto assume que estas elevações bruscas de valores deve-se a ruído no sinal.

O algoritmo que pretendemos paralelizar consiste em:

$$\hat{y}_i = \frac{\sum_{k=0}^{N-1} K_b(x_i, x_k) y_k}{\sum_{k=0}^{N-1} K_b(x_i, x_k)}$$

com

$$K_b(x, x_k) = exp\left(-\frac{(x - x_k)^2}{2b^2}\right)$$

onde

x

Domínio do sinal a ser filtrado

y

Sinal observado e que contém ruído e do qual pretendemos obter a versão sem ruído

$\hat{y}$

Sinal obtido pela passagem do sinal y pela função de *smoothing*

b

Parâmetro de *smoothing*, no nosso caso foi utilizado o valor 4 para este parâmetro

Capítulo 2 | Código C

Para servir como ponto de partida e de comparação com os resultados obtidos quando utilizado o GPU foi desenvolvido este código em C com base no exemplo dado no enunciado.

```c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <sys/time.h>
#define N       10000
#define smooth  4

double randn (double mu, double sigma){
  double U1, U2, W, mult;
  static double X1, X2;
  static int call = 0;

  if (call == 1){
    call = !call;
    return (mu + sigma * (double) X2);
  }
  do{
    U1 = -1 + ((double) rand () / RAND_MAX) * 2;
    U2 = -1 + ((double) rand () / RAND_MAX) * 2;
    W = pow (U1, 2) + pow (U2, 2);
  }while (W >= 1 || W == 0);
```

```
  mult = sqrt ((-2 * log (W)) / W);
  X1 = U1 * mult;
  X2 = U2 * mult;
  call = !call;
  return (mu + sigma * (double) X1);
}

double f_x(double x){
  return sin(0.02 * x) + sin(0.001 * x) + 0.1 * randn(0, 1);
}

double timeDiff(struct timespec tStart,struct timespec tEnd){
   struct timespec diff;

   diff.tv_sec=tEnd.tv_sec-tStart.tv_sec-(tEnd.tv_nsec<tStart
      .tv_nsec?1:0);
   diff.tv_nsec=tEnd.tv_nsec-tStart.tv_nsec+(tEnd.tv_nsec<
      tStart.tv_nsec?1000000000:0);
   return ((double) diff.tv_sec)+((double) diff.tv_nsec)/1e9;
}

void main(){
  double x[N];
  double y[N];
  double yest[N];
  int i,j;
  double sumA, sumB;
  struct timespec timeVect[2];
    double timeCPU;
  FILE* fp;

  for(i=0;i<N;i++){
    x[i]=(i*1.0)/10;
    y[i]=f_x(x[i]);
```

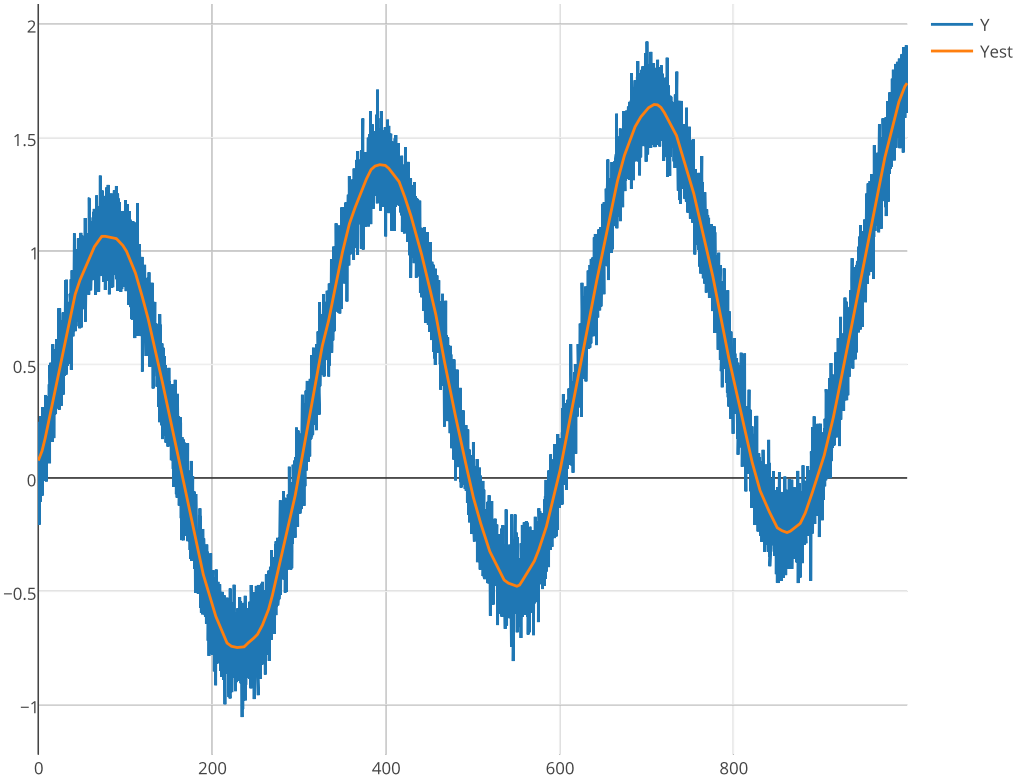
```
  }

  clock_gettime(CLOCK_REALTIME, &timeVect[0]);
  for(i=0;i<N;i++){
    sumA=0;
    sumB=0;
    for(j=0;j<N;j++){
      sumA = sumA + exp((-pow((x[i]-x[j]),2))/(2*pow(smooth
          ,2)))*y[j];
      sumB = sumB + exp((-pow((x[i]-x[j]),2))/(2*pow(smooth
          ,2)));
    }
    yest[i] = sumA / sumB;
  }
  clock_gettime(CLOCK_REALTIME, &timeVect[1]);
  timeCPU = timeDiff(timeVect[0],timeVect[1]);
  printf("CPU execution took %.6f seconds\n", timeCPU );

  fp=fopen("output.csv", "w");
  fwrite("X,Y,Yest\n",9,sizeof(char),fp);
  for(i=0;i<N;i++){
      fprintf(fp,"%f,%f,%f\n", x[i], y[i], yest[i]);
  }
  fclose(fp);
}
```

Este código executa o algoritmo e calcula e imprime para o terminal o tempo que demora a fazê-lo, e em seguida imprime os dados para um ficheiro de saída.

Um gráfico exemplificativo do funcionamento deste código apresenta-se a seguir, o código foi executado na máquina Diana que nos foi disponibilizada e onde demorou um tempo mediano de 5,7345560 segundos quando N igual a 10000. Apenas foi contabilizado o tempo de execução do algoritmo em si tendo sido ignorado o tempo de alocação de recursos e inicialização destes.


Y
Yest
2
1.5
1
0.5
0
−0.5
−1
0
200
400
600
800

Capítulo 3 | Código para o GPU

Para optimizar a aceleração do algoritmo de *smoothing* no *GPU* foram implementadas três diferentes versões cada uma explorando a arquitectura do *GPU* de forma diferente.

3.1 Kernel

3.1.1 Versão A

Numa primeira abordagem ao algoritmo, foi implementado um *Kernel* que executava o *for loop* interior do algoritmo em que cada *thread* computava uma iteração do *for loop* exterior. Desta forma eram necessárias N threads sendo N o número de iterações de cada *loop*. Esta implementação consegue tirar partido do conceito básico por trás dos *GPUs* contudo não o faça da forma óptima e apresenta maus resultados ao nível da transferência de dados quando N é elevado.

3.1.2 Versão B

Uma segunda versão permitia obter o máximo de paralelismo possível do *GPU* criando uma *grid* com as seguintes dimensões,

$$grid.x = \sqrt{\frac{N \cdot N}{maxThreadsPerBlock}} + 1$$
$$grid.y = \sqrt{\frac{N \cdot N}{maxThreadsPerBlock}} + 1$$

e dimensões de bloco,

$$block.x = \sqrt{maxThreadsPerBlock}$$

$$block.y = \sqrt{maxThreadsPerBlock}$$

Desta forma é garantido que existem threads suficientes para o cálculo do algoritmo em que cada thread executa o cálculo de uma exponencial e por fim N threads para o cálculo da soma de cada A e B e da divisão final.

Esta versão apresenta limitações ao nível da memória existente para alocar recursos uma vez que era necessário a alocação de um vector de $N \cdot N \cdot 2$ e uma vez que o *GPU* apenas tem 2GB de memória disponível apenas é possível aplicar o algoritmo para,

$$N = \sqrt{\frac{2048 \cdot 1024 \cdot 1024}{2 \cdot 4}} = 16384$$

O principal factor que inviabilizou esta versão foi o tempo de tranferência dos dados do *GPU* para *Host* que para N igual a 10000 era de aproximadamente 0.25 segundos. Uma hipótese é que o tempo de transferência de dados não depende somente do tamanho do vector a tranferir mas também do total de dados alocados no *GPU* em que para N igual 10000 correspondiam a cerca de 800 MB.

3.1.3 Versão C

Esta última versão segue o mesmo príncipio que a versão A contudo é definido um valor máximo para o tamanho dos vectores de saída, ou seja, se N for igual a 50000 o tamanho máximo do vector *Yest* é 10000 e o *Kernel* é executado 5 vezes. Esta versão afasta-se um pouco do conceito de paralelismo tentando minimizar o tempo de transferência de dados, tempo que se apresentou ser o de maior impacto.

Comparando a versão A e C relativamente à última afirmação, tem-se um tempo de 0.361632 segundos para transferir 80000 elementos para A enquanto que para C tem-se 0,070261 segundos. Esta alteração permite reduzir o tempo de transferência em 5.15 vezes.

3.2 Transferências de Dados

Inicialmente são transferidos os vectores de "entrada"(X *e* Y) necessários para os cálculos dos resultados, em seguida e como descrito na Secção **??** o *Kernel* é executado várias vezes para calcular uma porção dos resultados e após cada chamada ao *Kernel* é transferido para o *Host* os resultados estimados (Yest) acabados de calcular.

Escolhemos fazer deste modo para permitir minimizar o impacto da transferência de informação do *GPU* para o *Host* pois era essa comunicação que ditava o desempenho do programa.

3.3 Critério de aceitação de resultados

Os resultados provenientes do *GPU* são considerados como correctos se diferirem dos resultados obtidos no *CPU* menos do que 1×10^{-6}.

Existe sempre uma ligeira diferença nos resultados devido ao facto de se tratarem de unidades aritméticas diferentes sem precisão infinita.

Capítulo 4 | Resultados

Na tabela 4.1 são apresentados os resultados obtidos para o algoritmo de *smoothing*. Foram feitos testes para valores de N até 80000 sendo feita uma descrição detalhada do tempo para cada secção do programa desenvolvido. Nas tabelas 4.2 e 4.3 é feita uma previsão dos tempos quer para o CPU quer para o GPU (previsão feita com base nas linhas de tendência dos valores até 80000).

	N							
	10000	20000	30000	40000	50000	60000	70000	80000
First Call (segs)	0,413513	0,413302	0,415701	0,418801	0,417367	0,419858	0,418481	0,420071
Malloc Memory (segs)	0,000225	0,000397	0,000234	0,000233	0,000350	0,000404	0,000394	0,000404
Copy - H to D (segs)	0,000096	0,000178	0,000139	0,000175	0,000283	0,000338	0,000393	0,000444
Kernel Execution (segs)	0,000026	0,000058	0,000060	0,000068	0,000074	0,000081	0,000085	0,000094
Copy - D to H (segs)	0,008782	0,017558	0,026343	0,035125	0,043910	0,052692	0,061484	0,070261
Free Memory (segs)	0,000098	0,000120	0,000120	0,000121	0,000120	0,000120	0,000120	0,000120
Total CPU (segs)	5,734556	20,934259	45,695911	78,612154	120,727607	172,362745	233,990434	304,218101
Total GPU (segs)	0,422739	0,431612	0,442597	0,454522	0,462104	0,473492	0,480956	0,491394
Total s/ First Call e Free Memory (segs)	0,009129	0,018191	0,026776	0,035600	0,044617	0,053514	0,062355	0,071203
Speedup Total	13,565240	48,502551	103,244963	172,955852	261,256637	364,025004	486,511609	619,092643
Speedup Kernel	628,2035384	1150,834721	1706,599604	2208,237574	2705,895947	3220,890701	3752,552867	4272,57611

Figura 4.1: Resultados Obtidos.

	N							
	90000	100000	110000	120000	130000	140000	150000	160000
Total GPU (segs)	0,081300	0,090300	0,099300	0,108300	0,117300	0,126300	0,135300	0,144300
Total CPU (segs)	371,732800	458,862800	555,792800	662,702800	779,772800	907,182800	1045,112800	1193,742800
Speedup	4572,359164	5081,537099	5597,107754	6119,139428	6647,679454	7182,761679	7724,410939	8272,645877

Figura 4.2: Previsão de Resultados.

	N							
	170000	180000	190000	200000	210000	220000	230000	240000
Total GPU (segs)	0,153300	0,162300	0,171300	0,180300	0,189300	0,198300	0,207300	0,216300
Total CPU (segs)	1353,252800	1523,822800	1705,632800	1898,862800	2103,692800	2320,302800	2548,872800	2789,582800
Speedup	8827,480757	9388,926679	9956,992411	10531,684969	11113,010037	11700,972264	12295,575494	12896,822931

Figura 4.3: Previsão de Resultados (continuação).

Com os valores obtidos através de baterias de 30 execuções para cada N foram elaborados os gráficos com o tempo de execução do CPU, do GPU, do Kernel e por fim da relação N - Speedup.

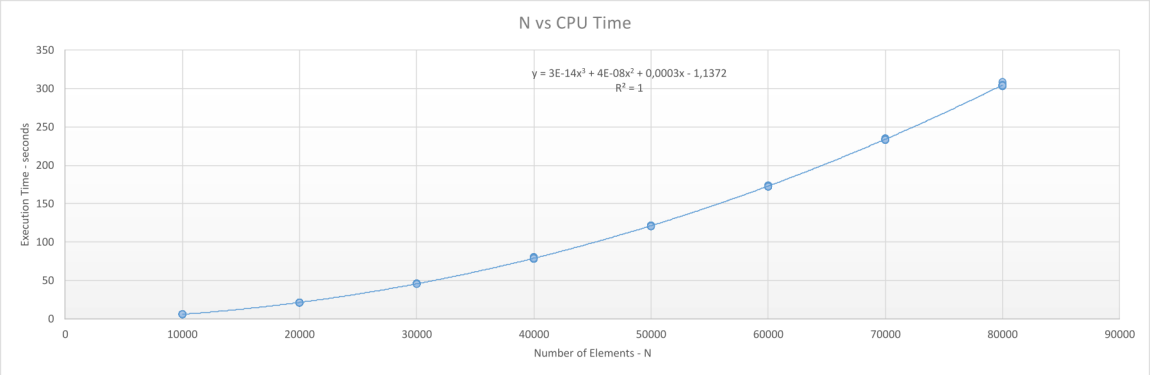


Figura 4.4: Relação N - Tempo de CPU.

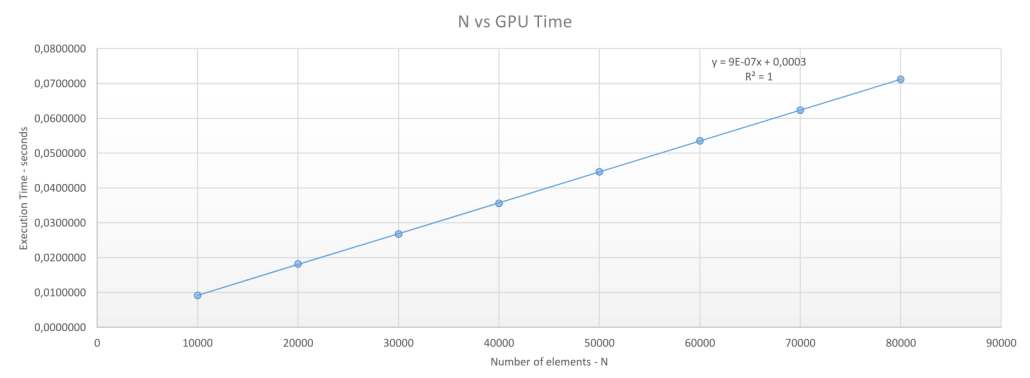


Figura 4.5: Relação N - Tempo de GPU.

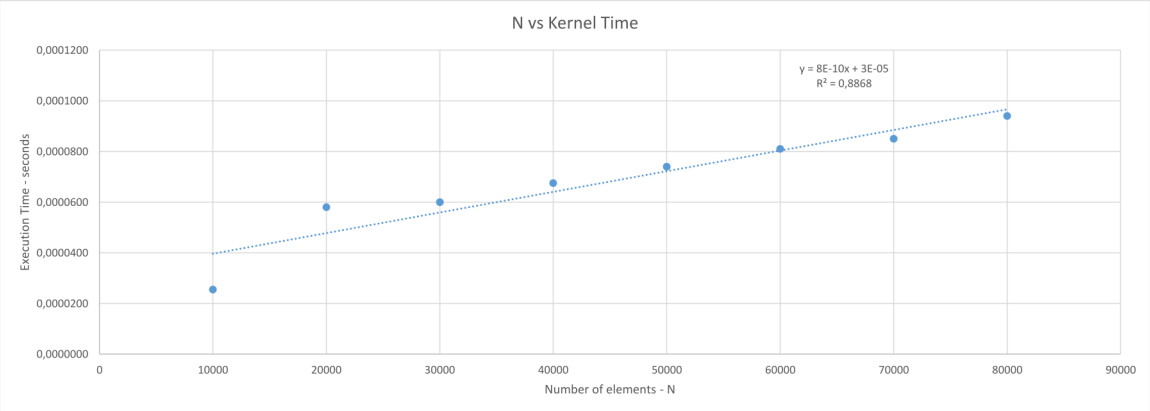


Figura 4.6: Relação N - Tempo de Kernel.

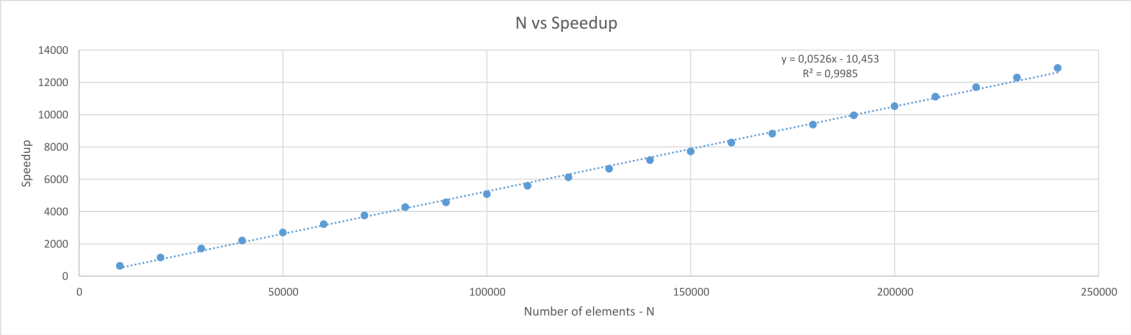


Figura 4.7: Relação N - Speedup.

Por observação dos gráficos 4.4 e 4.5 é óbvio a utilização de *GPUs* para este tipo de algoritmo. No caso do *CPU* para N elevados o tempo de execução começa-se a obter uma função do tipo exponencial ao contrário do *GPU* que mantêm um comportamento linear com crescimento imposto pelo tempo de tranferência de dados do *Device* para o *Host*.

Apesar de não serem apresentados valores de tempo para N inferior a 10000, tal como verificado na demonstração do programa na aula, o *GPU* apresenta valores de Speedup inferior a 0.3 para N igual a 1000 e desta forma para valores de N baixos não se justifica a utilização de *GPU* para este tipo de algoritmo.

Capítulo 5 | Conclusão

Capítulo A | Anexos

A.1 CPU

```
printf("Performing␣the␣computation␣on␣the␣CPU...\n");
clock_gettime(CLOCK_REALTIME, &timeVect[0]);
for(i=0;i<MAX;i++){
  sumA=0;
  sumB=0;
  for(j=0;j<MAX;j++){
    sumA = sumA + exp((-pow((h_X[i]-h_X[j]),2))/(2*pow(
       smooth,2)))*h_Y[j];
  }
  for(j=0;j<MAX;j++){
    sumB = sumB + exp((-pow((h_X[i]-h_X[j]),2))/(2*pow(
       smooth,2)));
  }
  yest_cpu[i] = sumA / sumB;
}
clock_gettime(CLOCK_REALTIME, &timeVect[1]);
timeCPU = timeDiff(timeVect[0],timeVect[1]);
printf("␣␣␣␣...␣execution␣took␣%.6f␣seconds\n", timeCPU );
```

A.2 Kernel GPU

```
/**
 * CUDA Kernel Device code
 */
__global__ void calcy(float *X, float *Y, float *Yest,int
    indice) {
  int i = blockDim.x * blockIdx.x + threadIdx.x;
  int j;

  float smoothing = (float) smooth;
  float A=0, B=0, tmp;
  float smth = 2*pow(smoothing,2);

  for(j=0;j<N;j++){
    tmp = exp((-pow((X[i+indice*N]-X[j+indice*N]),2))/(smth))
        ;
    A = A + tmp*Y[j+indice*N];
    B = B + tmp;
  }

  Yest[i] = A/B;
}
```

A.3 Código final - Versão A

```
#include "cuda_runtime.h"
#include "device_launch_parameters.h"

#include <stdio.h>
#include <sys/time.h>

#define N 10000
#define Streams 10
#define DIMENSION 2
#define smooth  4
#define twoBB   2*smooth*smooth
float randn(float mu, float sigma);
float f_x(float x);

double timeDiff(struct timespec tStart, struct timespec tEnd)
   {
   struct timespec diff;

   diff.tv_sec  = tEnd.tv_sec  - tStart.tv_sec  - (tEnd.
      tv_nsec<tStart.tv_nsec?1:0);
   diff.tv_nsec = tEnd.tv_nsec - tStart.tv_nsec + (tEnd.
      tv_nsec<tStart.tv_nsec?1000000000:0);

   return ((double) diff.tv_sec) + ((double) diff.tv_nsec)/1
      e9;
}

float randn(float mu, float sigma){
  float U1, U2, W, mult;
  static float X1, X2;
  static int call = 0;
```

```
  if (call == 1){
    call = !call;
    return (mu + sigma * (float)X2);
  }

  do{
    U1 = -1 + ((float)rand() / RAND_MAX) * 2;
    U2 = -1 + ((float)rand() / RAND_MAX) * 2;
    W = pow(U1, 2) + pow(U2, 2);
  } while (W >= 1 || W == 0);

  mult = sqrt((-2 * log(W)) / W);
  X1 = U1 * mult;
  X2 = U2 * mult;

  call = !call;

  return (mu + sigma * (float)X1);
}

float f_x(float x){
  return sin(0.02 * x) + sin(0.001 * x) + 0.1 * randn(0, 1);
}



__global__ void Kernel(float *x, float *y, float *expo, float
    *sumB, float *div)
{

    float twosmoothsqr = (float) 2*smooth*smooth;
```

```
float soma1_16 = 0, soma2_16 = 0, soma3_16 = 0, soma4_16
   = 0;
float soma5_16 = 0,soma6_16 = 0,soma7_16 = 0, soma8_16 =
   0;
float soma9_16 = 0,soma10_16 = 0,soma11_16 = 0, soma12_16
    = 0;
float soma13_16 = 0,soma14_16 = 0,soma15_16 = 0,
   soma16_16 = 0;
float soma = 0;
int index_x = blockIdx.x * blockDim.x + threadIdx.x;
int index_y = blockIdx.y * blockDim.y + threadIdx.y;
int grid_width = gridDim.x * blockDim.x;

int blockId = blockIdx.x + blockIdx.y  * gridDim.x;
int index = blockId * (blockDim.x *blockDim.y) + (
   threadIdx.y*blockDim.x) + threadIdx.x;

if(index_x < N && index_y < N){
    expo[index_y + index_x * N] = exp((-pow((x[index_x]-x
       [index_y]),2))/twosmoothsqr);
    expo[index_y + index_x * N + N*N] = exp((-pow((x[
       index_x]-x[index_y]),2))/twosmoothsqr)*y[index_y];
}
__syncthreads();

/*if(index < 2*N){
    for(int i = 0; i< N ; i++){
        soma += expo[i + index*N];
    }
    sumB[index] = soma;
}*/

if(index < 2*N){
    for(int i = 0; i< N/16 ; i++){
```

```
            soma1_16 += expo[i + index*N];
            soma2_16 += expo[i + N/16 + index*N];
            soma3_16 += expo[i + 2*(N/16) + index*N];
            soma4_16 += expo[i + 3*(N/16) + index*N];
            soma5_16 += expo[i + 4*(N/16) + index*N];
            soma6_16 += expo[i + 5*(N/16) + index*N];
            soma7_16 += expo[i + 6*(N/16) + index*N];
            soma8_16 += expo[i + 7*(N/16) + index*N];
            soma9_16 += expo[i + 8*(N/16) + index*N];
            soma10_16 += expo[i + 9*(N/16) + index*N];
            soma11_16 += expo[i + 10*(N/16) + index*N];
            soma12_16 += expo[i + 11*(N/16) + index*N];
            soma13_16 += expo[i + 12*(N/16) + index*N];
            soma14_16 += expo[i + 13*(N/16) + index*N];
            soma15_16 += expo[i + 14*(N/16) + index*N];
            soma16_16 += expo[i + 15*(N/16) + index*N];
        }
        sumB[index] = soma1_16+soma2_16+soma3_16+soma4_16+
           soma5_16+soma6_16+soma7_16+soma8_16+soma9_16+
           soma10_16+soma11_16+soma12_16+soma13_16+soma14_16+
           soma15_16+soma16_16;
    }

    __syncthreads();

    if(index < N){
        div[index] = sumB[index + N] / sumB[index];
    }

}

int main()
{
```

```
int device;
int maxThreadsPerBlock;
int blocksPerGrid;
int Xsize, Ysize, exposize, sumBsize, Divsize;
  int sharedmem;
float x[N], y[N], expo[N], Div[N];
float *d_X = NULL ,*d_Y = NULL,*d_expo = NULL,*d_sumA =
   NULL, *d_sumB = NULL,*d_Div = NULL;
  FILE *fp;
  struct timespec timeVect[20];
  double timeCPU, timeGPU[20];
  cudaError_t err[] = { cudaSuccess , cudaSuccess ,
     cudaSuccess };
  cudaError_t mem = cudaSuccess;
  if (err[0] != cudaSuccess)
  {
      fprintf(stderr, "Failed␣to␣deinitialize␣the␣device!␣
         error=%s\n", cudaGetErrorString(err[0]));
      exit(EXIT_FAILURE);
  }
  clock_gettime(CLOCK_REALTIME, &timeVect[9]);
  cudaGetDevice(&device);
  clock_gettime(CLOCK_REALTIME, &timeVect[10]);
  cudaFree(0);

   float sumA, sumB, yest_cpu[N];

  // Allocate the host
  float *h_X = (float *)malloc(N * sizeof(float));
  float *h_Y = (float *)malloc(N * sizeof(float));
  float *yest = (float *)malloc(N * sizeof(float));

  // Verify that allocations succeeded
  if (h_X == NULL || h_Y == NULL || yest == NULL )
```

```
    {
        fprintf(stderr, "Failed to allocate host data!\n");
        exit(EXIT_FAILURE);
    }

    // Initialize the host input data
    for (int i = 0; i < N ; i++ ){
        h_X[i] = (i*1.0)/10;
        h_Y[i] = f_x(h_X[i]);
        yest[i] = 1;
    }

    // Compute expected result
    printf("Performing the computation on the CPU...\n");
    clock_gettime(CLOCK_REALTIME, &timeVect[0]);
    for(int i=0;i<N;i++){
        sumA=0;
        sumB=0;
        for(int j=0;j<N;j++){
            sumA = sumA + exp((-pow((h_X[i]-h_X[j]),2))/(2*
               pow(smooth,2)))*h_Y[j];
            sumB = sumB + exp((-pow((h_X[i]-h_X[j]),2))/(2*
               pow(smooth,2)));
        }
        yest_cpu[i] = sumA / sumB;
    }
    clock_gettime(CLOCK_REALTIME, &timeVect[1]);
    timeCPU = timeDiff(timeVect[0],timeVect[1]);
Xsize = N * sizeof(float);
Ysize = N * sizeof(float);
    exposize = N * sizeof(float);
    sumBsize = N * sizeof(float);
    Divsize = N * sizeof(float);
```

```
  struct cudaDeviceProp props;
cudaGetDeviceProperties(&props, device);

  size_t uCurAvailMemoryInBytes;
  size_t uTotalMemoryInBytes;
  cudaMemGetInfo( &uCurAvailMemoryInBytes, &
     uTotalMemoryInBytes );
  int mem_av = uTotalMemoryInBytes;
printf("Device␣Number:␣%d\n", device);
printf("\tDevice␣name:␣%s\n", props.name);
printf("\tDevice␣max␣threads␣per␣block:␣%d\n", props.
   maxThreadsPerBlock);
printf("\tMemory␣Clock␣Rate␣(KHz):␣%d\n", props.
   memoryClockRate);
printf("\tMemory␣Bus␣Width␣(bits):␣%d\n",props.
   memoryBusWidth);
  printf("\tTotal␣Memory␣Available␣(MB):␣%d\n",
     uTotalMemoryInBytes / ( 1024 * 1024 ));
  printf("\tShared␣memory:␣%d\n",props.sharedMemPerBlock);

  sharedmem = props.sharedMemPerBlock;
maxThreadsPerBlock = props.maxThreadsPerBlock;
  // create 2d 2x2 thread block
  dim3 block_size;
  block_size.x = sqrt(maxThreadsPerBlock);
  block_size.y = sqrt(maxThreadsPerBlock);
  printf("Creating␣thread␣block␣of␣%dx%d␣threads...\n",
     block_size.x,block_size.y);
  //configure 2d grid
  dim3 grid_size;
  grid_size.x = sqrt((N*N) / maxThreadsPerBlock) + 1;
  grid_size.y = sqrt((N*N) / maxThreadsPerBlock) + 1;
```

```
printf("Creating␣grid␣of␣%dx%d␣blocks...\n",grid_size.x,
   grid_size.y);
clock_gettime(CLOCK_REALTIME, &timeVect[0]);
err[0] = cudaMalloc((void**)&d_X, Xsize);
if (err[0] != cudaSuccess) {
    fprintf(stderr, "cudaMalloc␣failed␣X!");
    return 0;
}
err[0] = cudaMalloc((void**)&d_Y, Ysize);
if (err[0] != cudaSuccess) {
    fprintf(stderr, "cudaMalloc␣failed␣Y!");
    return 0;
}
err[0] = cudaMalloc((void**)&d_expo,exposize*N*2);
if (err[0] != cudaSuccess) {
    fprintf(stderr, "cudaMalloc␣failed␣expo!");
    return 0;
}

err[0] = cudaMalloc((void**)&d_sumB,sumBsize*2);
if (err[0] != cudaSuccess) {
    fprintf(stderr, "cudaMalloc␣failed␣sumB!");
    return 0;
}
err[0] = cudaMalloc((void**)&d_Div,Divsize);
if (err[0] != cudaSuccess) {
    fprintf(stderr, "cudaMalloc␣failed␣sumB!");
    return 0;
}
clock_gettime(CLOCK_REALTIME, &timeVect[1]);
cudaMemGetInfo( &uCurAvailMemoryInBytes, &
   uTotalMemoryInBytes );
int mem_curr = uCurAvailMemoryInBytes;
int mem_all = (mem_av - mem_curr)/(1024*1024);
```

```
printf("Allocating␣%d␣MB␣of␣memory...\n",mem_all);

printf("Copying␣from␣CPU␣to␣GPU...\n");
err[0] = cudaMemcpy(d_X, h_X, Xsize ,
   cudaMemcpyHostToDevice);
err[1] = cudaMemcpy(d_Y, h_Y, Ysize ,
   cudaMemcpyHostToDevice);

clock_gettime(CLOCK_REALTIME, &timeVect[2]);
if ((err[0] != cudaSuccess) || (err[1] != cudaSuccess) ||
    (err[2] != cudaSuccess)){
    fprintf(stderr, "Failed␣to␣allocate␣device␣values␣X␣
       or␣Y!␣Error␣codes␣are:\n");
    fprintf(stderr, "\t␣Allocation␣of␣%d␣Bytes␣for␣value␣
       X:␣%s\n", Xsize , cudaGetErrorString(err[0]) );
    fprintf(stderr, "\t␣Allocation␣of␣%d␣Bytes␣for␣value␣
       Y:␣%s\n", Ysize , cudaGetErrorString(err[1]) );
    fprintf(stderr, "\t␣Allocation␣of␣%d␣Bytes␣for␣value␣
       Yest:␣%s\n", Ysize*N , cudaGetErrorString(err[2])
       );
    exit(EXIT_FAILURE);
}

clock_gettime(CLOCK_REALTIME, &timeVect[3]);
Kernel<<<grid_size, block_size>>>(d_X,d_Y,d_expo,d_sumB,
   d_Div);
clock_gettime(CLOCK_REALTIME, &timeVect[4]);

float somaA[N], somaB[2*N];

/*err[0] = cudaMemcpy(somaA, d_sumA, N*sizeof(float) ,
   cudaMemcpyDeviceToHost);
if (err[0] != cudaSuccess){
```

```
    fprintf(stderr, "Failed to copy Yest from device to
       host (error code %s)!\n", cudaGetErrorString(err
       [0]));
     exit(EXIT_FAILURE);*/
clock_gettime(CLOCK_REALTIME, &timeVect[5]);
err[0] = cudaMemcpy(Div, d_Div, N*sizeof(float),
   cudaMemcpyDeviceToHost);
clock_gettime(CLOCK_REALTIME, &timeVect[6]);
if (err[0] != cudaSuccess){
    fprintf(stderr, "Failed to copy Yest from device to
       host (error code %s)!\n", cudaGetErrorString(err
       [0]));
     exit(EXIT_FAILURE);
}
fp=fopen("output.out", "w");
for(int j=0;j<N;j++){
    fprintf(fp,"%f\n",Div[j]);
}
fclose(fp);

clock_gettime(CLOCK_REALTIME, &timeVect[7]);
cudaFree(d_X);
cudaFree(d_Y);
cudaFree(d_expo);
cudaFree(d_sumB);
cudaFree(d_Div);
clock_gettime(CLOCK_REALTIME, &timeVect[8]);

timeGPU[0] = timeDiff(timeVect[0],timeVect[1]);
timeGPU[1] = timeDiff(timeVect[1],timeVect[2]);
timeGPU[2] = timeDiff(timeVect[3],timeVect[4]);
timeGPU[3] = timeDiff(timeVect[5],timeVect[6]);
timeGPU[4] = timeDiff(timeVect[7],timeVect[8]);
timeGPU[5] = timeDiff(timeVect[0],timeVect[8]);
```

```
    timeGPU[6] = timeDiff(timeVect[9],timeVect[10]);

    printf("␣␣␣␣...␣execution␣took␣%.6f␣seconds␣(speedup=%.3f
       ),␣corresponding␣to:\n",timeGPU[5]+timeGPU[6],timeCPU
       /(timeGPU[5]+timeGPU[6]));
    printf("␣␣␣␣␣␣␣␣␣␣-␣first␣call␣to␣the␣device␣␣␣␣␣␣␣␣␣␣->
       ␣%.6f␣seconds\n",timeGPU[6]);
    printf("␣␣␣␣␣␣␣␣␣␣-␣allocation␣of␣memory␣on␣the␣device␣->
       ␣%.6f␣seconds\n",timeGPU[0]);
    printf("␣␣␣␣␣␣␣␣␣␣-␣copying␣data␣from␣host␣to␣device␣␣␣->
       ␣%.6f␣seconds\n",timeGPU[1]);
    printf("␣␣␣␣␣␣␣␣␣␣-␣kernel␣execution␣on␣the␣device␣␣␣␣␣->
       ␣%.6f␣seconds\n",timeGPU[2]);
    printf("␣␣␣␣␣␣␣␣␣␣-␣copying␣data␣from␣device␣to␣host␣␣␣->
       ␣%.6f␣seconds\n",timeGPU[3]);
    printf("␣␣␣␣␣␣␣␣␣␣-␣freeing␣data␣on␣the␣device␣␣␣␣␣␣␣␣->
       ␣%.6f␣seconds\n",timeGPU[4]);
    printf("
       ----------------------------------------------------------------
       n");

    int i=0,j=0;

    for (i = 0, j = 0; i < N; i++)
    {
        if (fabs(Div[i]-yest_cpu[i]) > 1e-3)
        {
            //fprintf(stderr, "Result verification failed at
               element %d => CPU returns %f while GPU returns
                %f\n", i, yest_cpu[i],Div[i]);
            j++;
        }
    }
    if (j>0) {
```

```
        printf("%d errors found!\n",j);
        exit(EXIT_FAILURE);
    }
    printf("Test PASSED\n");

    printf("Done\n");


    return 0;

}
```

A.4 Código final - Versão C

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <sys/time.h>
#define N   10000
#define smooth  4
#define THREADS_PER_BLOCK 1000

// For the CUDA runtime routines (prefixed with "cuda_")
#include <cuda_runtime.h>

/**
 * CUDA Kernel Device code
 */
__global__ void calcy(float *X, float *Y, float *Yest,int
   indice) {
  int i = blockDim.x * blockIdx.x + threadIdx.x;
  int j;
```

```
  float smoothing = (float) smooth;
  float A=0, B=0, tmp;
  float smth = 2*pow(smoothing,2);

  for(j=0;j<N;j++){
    tmp = exp((-pow((X[i+indice*N]-X[j+indice*N]),2))/(smth))
       ;
    A = A + tmp*Y[j+indice*N];
    B = B + tmp;
  }

  Yest[i] = A/B;
}

/**
 * timeDiff
 *
 * Computes the difference (in ns) between the start and end
    time
 */
double timeDiff(struct timespec tStart, struct timespec tEnd)
   {
  struct timespec diff;

  diff.tv_sec  = tEnd.tv_sec  - tStart.tv_sec  - (tEnd.
     tv_nsec<tStart.tv_nsec?1:0);
  diff.tv_nsec = tEnd.tv_nsec - tStart.tv_nsec + (tEnd.
     tv_nsec<tStart.tv_nsec?1000000000:0);
  return ((double) diff.tv_sec) + ((double) diff.tv_nsec)/1e9
     ;
}

/**
 * randn
```

```
 *
 * Computes a random value with a gaussian distribuition
 */
double randn (double mu, double sigma){
  double U1, U2, W, mult;
  static double X1, X2;
  static int call = 0;

  if (call == 1){
    call = !call;
    return (mu + sigma * (double) X2);
  }

  do{
    U1 = -1 + ((double) rand () / RAND_MAX) * 2;
    U2 = -1 + ((double) rand () / RAND_MAX) * 2;
    W = pow (U1, 2) + pow (U2, 2);
  }while (W >= 1 || W == 0);

  mult = sqrt ((-2 * log (W)) / W);
  X1 = U1 * mult;
  X2 = U2 * mult;

  call = !call;

  return (mu + sigma * (double) X1);
}

/**
 * f_x
 *
 * Computes y = f(x)
 */
float f_x(float x){
```

```
  return sin(0.02 * x) + sin(0.001 * x) + 0.1 * randn(0, 1);
}

/**
 * Host main routine
 */
int main(int argc, char **argv) {
  // Error code to check return values for CUDA calls
  unsigned i,j;
  struct timespec timeVect[7];
  double timeCPU, timeGPU[7];
  cudaError_t err[] = { cudaSuccess , cudaSuccess ,
     cudaSuccess };
  if(argc < 2){
    printf("N indefinido\n");
    exit(-1);
  }
  int MAX = atoi(argv[1]);
  //cpu variables
  float sumA, sumB, yest_cpu[MAX];
  FILE* fp;

  for(int i = 0; i <7; i++)
    timeGPU[i] = 0;

  // Allocate the host
  float *h_X = (float *)malloc(MAX * sizeof(float));
  float *h_Y = (float *)malloc(MAX * sizeof(float));
  float *yest = (float *)malloc(MAX * sizeof(float));

  // Verify that allocations succeeded
  if (h_X == NULL || h_Y == NULL || yest == NULL )
  {
    fprintf(stderr, "Failed to allocate host data!\n");
```

```
    exit(EXIT_FAILURE);
}

// Initialize the host input data
for (i = 0; i < MAX ; i++ ){
    h_X[i] = (i*1.0)/10;
    h_Y[i] = f_x(h_X[i]);
    yest[i] = 0;
}

// Compute expected result
printf("Performing␣the␣computation␣on␣the␣CPU...\n");
clock_gettime(CLOCK_REALTIME, &timeVect[0]);
for(i=0;i<MAX;i++){
    sumA=0;
    sumB=0;
    for(j=0;j<MAX;j++){
        sumA = sumA + exp((-pow((h_X[i]-h_X[j]),2))/(2*pow(
            smooth,2)))*h_Y[j];
    }
    for(j=0;j<MAX;j++){
        sumB = sumB + exp((-pow((h_X[i]-h_X[j]),2))/(2*pow(
            smooth,2)));
    }
    yest_cpu[i] = sumA / sumB;
}
clock_gettime(CLOCK_REALTIME, &timeVect[1]);
timeCPU = timeDiff(timeVect[0],timeVect[1]);
printf("␣␣␣␣...␣execution␣took␣%.6f␣seconds\n", timeCPU );

// Compute on the GPU
printf("
    ----------------------------------------------------------------
    n");
```

```
printf("Performing␣the␣computation␣on␣the␣GPU...\n");

// initialize the device (just measure the time for the
   first call to the device)
//cudaSetDevice(0);
//cudaDeviceReset();
clock_gettime(CLOCK_REALTIME, &timeVect[0]);
cudaFree(0);
clock_gettime(CLOCK_REALTIME, &timeVect[1]);

// Allocate memory on the device
printf("␣...␣Allocation␣of␣memory␣on␣the␣Device␣...\n");
float *d_X = NULL , *d_Y = NULL , *d_yest = NULL;
err[0] = cudaMalloc( (void **) &d_X , MAX * sizeof(float) )
   ;
err[1] = cudaMalloc( (void **) &d_Y , MAX * sizeof(float) )
   ;
err[2] = cudaMalloc( (void **) &d_yest , N * sizeof(float)
   );

if ((err[0] != cudaSuccess) || (err[1] != cudaSuccess) || (
   err[2] != cudaSuccess))
{
  fprintf(stderr, "Failed␣to␣allocate␣device␣memory!␣Error␣
     codes␣are:\n");
  fprintf(stderr, "\t␣Allocation␣of␣%d␣Bytes␣for␣X:␣%s\n",
     N * sizeof(float) , cudaGetErrorString(err[0]) );
  fprintf(stderr, "\t␣Allocation␣of␣%d␣Bytes␣for␣Y:␣%s\n",
     N * sizeof(float) , cudaGetErrorString(err[1]) );
  fprintf(stderr, "\t␣Allocation␣of␣%d␣Bytes␣for␣yest:␣%s\n
     ", N * sizeof(float) , cudaGetErrorString(err[2]) );
  exit(EXIT_FAILURE);
}
```

```
// Copy the host input data to the device memory
clock_gettime(CLOCK_REALTIME, &timeVect[2]);
printf("␣...␣Copying␣input␣data␣from␣the␣host␣memory␣to␣the
   ␣CUDA␣device␣...\n");
err[0] = cudaMemcpy(d_X, h_X, MAX * sizeof(float) ,
   cudaMemcpyHostToDevice);
err[1] = cudaMemcpy(d_Y, h_Y, MAX * sizeof(float) ,
   cudaMemcpyHostToDevice);
clock_gettime(CLOCK_REALTIME, &timeVect[3]);
timeGPU[2] = timeDiff(timeVect[2],timeVect[3]);

if ((err[0] != cudaSuccess) || (err[1] != cudaSuccess))
{
  fprintf(stderr, "Failed␣to␣copy␣data␣to␣the␣device!␣Error
     ␣codes␣are:\n");
  fprintf(stderr, "\t␣X:␣%s\n", cudaGetErrorString(err[0])
     );
  fprintf(stderr, "\t␣Y:␣%s\n", cudaGetErrorString(err[1])
     );
  exit(EXIT_FAILURE);
}


// Launch the CUDA Kernel

dim3 tpb (THREADS_PER_BLOCK);
dim3 bpg (N/THREADS_PER_BLOCK);


printf("␣...␣CUDA␣kernel␣launch␣...\n");
for(int i = 0 ; i < ceil(MAX/N) ; i++){
  clock_gettime(CLOCK_REALTIME, &timeVect[3]);
  calcy<<<bpg, tpb>>>(d_X, d_Y, d_yest,i);
```

```
    clock_gettime(CLOCK_REALTIME, &timeVect[4]);
    err[0] = cudaGetLastError();

    if (err[0] != cudaSuccess)
    {
      fprintf(stderr, "Failed to launch kernel (error code %s
          )!\n", cudaGetErrorString(err[0]));
      exit(EXIT_FAILURE);
    }
    timeGPU[3] = timeGPU[3] + timeDiff(timeVect[3],timeVect
        [4]);

    // Copy the result back to host memory
    clock_gettime(CLOCK_REALTIME, &timeVect[4]);
    err[0] = cudaMemcpy(yest+i*N, d_yest, N * sizeof(float) ,
         cudaMemcpyDeviceToHost);
    clock_gettime(CLOCK_REALTIME, &timeVect[5]);
    timeGPU[4] = timeGPU[4] + timeDiff(timeVect[4],timeVect
        [5]);
    printf("Copy output data from the CUDA device to the host
       memory in %.6f seconds\n",timeDiff(timeVect[4],
       timeVect[5]));
    if (err[0] != cudaSuccess)
    {
      fprintf(stderr, "Failed to copy result from device to
          host (error code %s)!\n", cudaGetErrorString(err[0])
          );
      exit(EXIT_FAILURE);
    }
  }
  clock_gettime(CLOCK_REALTIME, &timeVect[5]);
  err[0] = cudaFree(d_X);
  err[1] = cudaFree(d_Y);
  err[2] = cudaFree(d_yest);
```

```
clock_gettime(CLOCK_REALTIME, &timeVect[6]);

if ((err[0] != cudaSuccess) || (err[1] != cudaSuccess) || (
    err[2] != cudaSuccess))
{
  fprintf(stderr, "Failed␣to␣free␣device␣memory!\n");
  fprintf(stderr, "\t␣X:␣%s\n", cudaGetErrorString(err[0])
     );
  fprintf(stderr, "\t␣Y:␣%s\n", cudaGetErrorString(err[1])
     );
  fprintf(stderr, "\t␣d_yest:␣%s\n", cudaGetErrorString(err
     [2]) );
  exit(EXIT_FAILURE);
}
clock_gettime(CLOCK_REALTIME, &timeVect[6]);
timeGPU[0] = timeDiff(timeVect[0],timeVect[1]);
timeGPU[1] = timeDiff(timeVect[1],timeVect[2]);
//timeGPU[3] = timeDiff(timeVect[3],timeVect[4]);
//timeGPU[4] = timeDiff(timeVect[4],timeVect[5]);
timeGPU[5] = timeDiff(timeVect[5],timeVect[6]);
timeGPU[6] = timeGPU[1] + timeGPU[2] + timeGPU[3] + timeGPU
    [4];
printf("␣␣␣␣...␣execution␣took␣%.6f␣seconds,␣corresponding␣
    to:\n",timeGPU[6]);
printf("␣␣␣␣␣␣␣␣␣␣-␣first␣call␣to␣the␣device␣␣␣␣␣␣␣␣␣␣->␣
    %.6f␣seconds\n",timeGPU[0]);
printf("␣␣␣␣␣␣␣␣␣␣-␣allocation␣of␣memory␣on␣the␣device␣->␣
    %.6f␣seconds\n",timeGPU[1]);
printf("␣␣␣␣␣␣␣␣␣␣-␣copying␣data␣from␣host␣to␣device␣␣␣->␣
    %.6f␣seconds\n",timeGPU[2]);
printf("␣␣␣␣␣␣␣␣␣␣-␣kernel␣execution␣on␣the␣device␣␣␣␣␣->␣
    %.6f␣seconds\n",timeGPU[3]);
printf("␣␣␣␣␣␣␣␣␣␣-␣copying␣data␣from␣device␣to␣host␣␣␣->␣
    %.6f␣seconds\n",timeGPU[4]);
```

```
printf("␣␣␣␣␣␣␣␣␣␣-␣freeing␣data␣on␣the␣device␣␣␣␣␣␣␣␣->␣
   %.6f␣seconds\n",timeGPU[5]);
printf("
   ----------------------------------------------------------------------
   n");
printf("␣␣␣␣...␣overall␣speedup=%.3f␣and␣kernel␣only␣
   execution␣speedup=%.3f\n",timeCPU/timeGPU[6], timeCPU/
   timeGPU[3]);
printf("
   ----------------------------------------------------------------------
   n");

//write data to file
fp=fopen("enunciado.txt", "a");
fwrite("X,Y,CPU,GPU\n",12,1,fp);
fprintf(fp,"%.6f,%.6f,%.6f,%.6f,%.6f,%.6f,%.6f,%.6f\n",
   timeCPU, timeGPU[6], timeGPU[0],timeGPU[1], timeGPU[2],
   timeGPU[3], timeGPU[4],timeGPU[5]);
fclose(fp);

// Free host memory
free(h_X);
free(h_Y);
free(yest);

// Reset the device and exit
err[0] = cudaDeviceReset();

if (err[0] != cudaSuccess)
{
  fprintf(stderr, "Failed␣to␣deinitialize␣the␣device!␣error
     =%s\n", cudaGetErrorString(err[0]));
  exit(EXIT_FAILURE);
}
```

```
  // Verify that the result matrix is correct
  for (i = 0, j = 0; i < MAX; i++)
  {
    if (fabs(yest[i]-yest_cpu[i]) > 1e-6)
    {
      j++;
    }
  }
  float erro = j/MAX;
  if (j>0) {
    printf("%d errors found! --- %f of %d Elements\n",j,erro,
       MAX);
    exit(EXIT_FAILURE);
  }
  printf("Test PASSED\n");
  printf("Done\n");
  return 0;
}
```