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 objetivo 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** que tira proveito das unidades de processamento gráfico (GPUs) da **NVIDIA**.

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

1.1 Algoritmo

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 "smothing", 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.

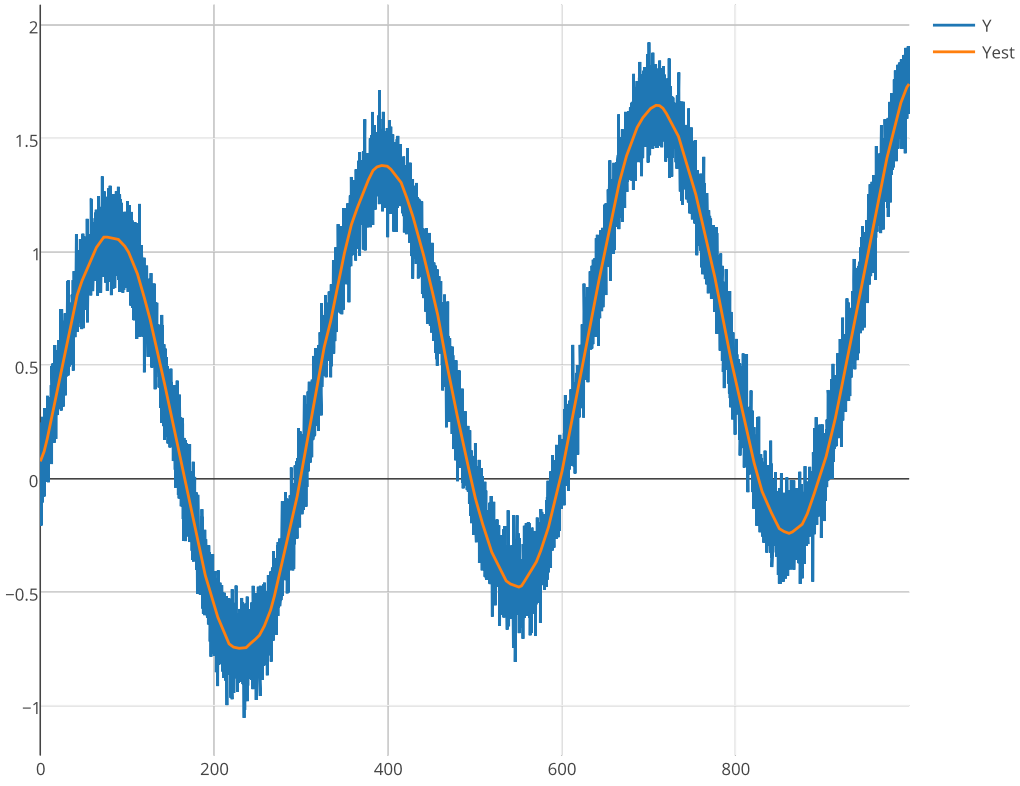
```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <sys/time.h>
#define N       10000
#define smooth  4

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 3,862388 segundos.



Capítulo 3 | Código para o GPU

Para optimizar a aceleração do algoritmo de "smoothing" no **GPU** começámos por analisar as diversas chamadas ao **GPU** e concluímos que as que consomem mais tempo são a inicialização e as transferências de dados entre o **Host** e o **GPU** e entre o **GPU** e o **Host**, sendo que a execução do **Kernel** em si consome uma porção quase negligenciável.

Com estas informações e tendo em conta que as chamadas de inicialização são constantes e não se podem alterar procurámos primeiro otimizar as transferências de dados e só depois otimizar o **Kernel**.

3.1 Kernel

O código do **Kernel** a executar no **GPU** pode ser visto na Secção A.2, e este é chamado várias vezes de modo a calcular uma parte do vetor de resultados.

3.2 Transferências de Dados

Inicialmente são transferidos os vetores de "entrada" (X e Y) necessários para os cálculos dos resultados, em seguida e como descrito na Secção 3.1 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 corretos 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

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

```
#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;
}
```