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

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.

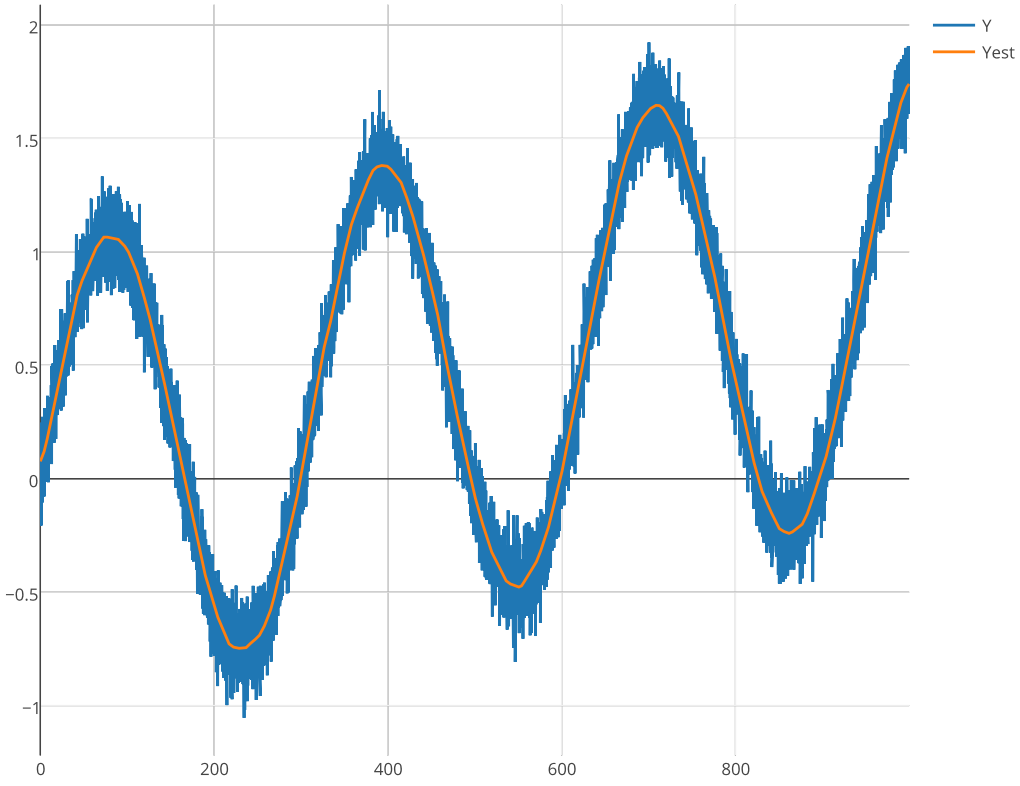
```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <sys/time.h>
#define N        10000
#define smooth   4

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 Transferências de Dados

3.2 Kernel

Capítulo 4 | Resultados

Capítulo 5 | Conclusão