Instituto Superior Técnico

Mestrado Integrado em Engenharia Electrotécnica e de Computadores

Arquiteturas Avançadas de Computadores

Paralisação e aceleração de um programa

Guilherme Branco Teixeira	n.º 70214
Maria Margarida Dias dos Reis	n.º 73099
Nuno Miguel Rodrigues Machado	n.º 74236

Lisboa, 1 de Junho de 2015

Índice

1 Introdução

Pretende-se paralisar e acelerar um algoritmo de *smoothing* usando um processador gráfico como ferramenta de computação paralela. Neste relatório esta demonstrado o algoritmo em CPU, as optimizações necessárias e paralisações essenciais de forma a ter os melhores resultados no GPU.

2 Implementação no CPU

Inicialmente o algoritmo proposto foi implementado para correr só no CPU. Para isso, foi transcrito para o código C. O algoritmo divide-se em duas partes fundamentais:

- Criação do sinal amostrado;
- *Smoothing* do sinal amostrado;

2.1 Criação do sinal amostrado

O sinal a ser processado é composto pela soma de dois sinais sinusoidais mais um erro com uma amplitude máxima de $0,1$. Todos os calculos são feitos em `float`, em primeiro lugar é realizado a alocação da memória de todas as variáveis necessárias para o calculo do sinal de entrada. De seguida está representado o código C em detalhe.

```
#define N 10000;
  ...

  int main() {

  float *x, *y, *yest_cpu, *randomArray;
  ...

  /*Alocacao de memoria*/
  x = (float *)malloc(N*sizeof(float));
  y = (float *)malloc(N*sizeof(float));
  yest_cpu = (float *)malloc(N*sizeof(float));
  randomArray = (float *)malloc(N*sizeof(float));
  ...
  exit(0);
  }
```

A implementação do algoritmo do sinal de entrada é composta por um ciclo que itera o numero de amostras do sinal pretendido. Em primeiro lugar é necessário gerar os valores ao sinal que vai ser processado, estes valores são gerados pela seguinte equação

$$X = i/10; \tag{2.1}$$

de seguida é gerado um valor aleatório entre 1 e -1, simulando o ruído resultante da amostragem do sinal. Este valor é gerado pela função `randn()`, o código da função está representado de seguida, é de salientar que o código foi obtido da Internet, onde este simula a função `randn()` do MatLab :

```
float randn()
{
  float x1, x2, w, y1;
  do
  {
    x1 = (float)(2.0 * rand() / RAND_MAX - 1.0);
    x2 = (float)(2.0 * rand() / RAND_MAX - 1.0);
    w = x1 * x1 + x2 * x2;
  } while (w >= 1.0);

  w = (float)sqrt((-2.0 * log(w)) / w);
  y1 = x1 * w;
  return y1;
}
```

Depois de obter os valores de `X` e do valor aleatório pode-se iterar os valores das amostras do sinal a ser processado. O código de seguida representa o ciclo que itera as amostras de `X`, do valor aleatório, `randomArray` e o sinal a ser processado, `Y`.

```
 int main(){
   for (int i = 0; i < N; ++i) {
     x[i] = (float)i / 10;
     randomArray[i] = randn();
     y[i] = function((float)x[i], (float)randomArray[i]);
   }
...
exit(0);
}
```

2.2 *Smoothing* do sinal amostrado

O algoritmo de *Smooth* para o anulamento do ruído resultante da amostragem do sinal é aplicado segundo a expressão seguinte:

$$y_{est} = \sum_{i=0}^{N-1} \frac{\sum_{k=0}^{N-1} Kb(x_i, x_k) y_k}{\sum_{k=0}^{N-1} Kb(x_i, x_k)}; \tag{2.2}$$

$$K_b(x_i, x_k) = e^{-\frac{(x_i - x_k)^2}{2b^2}}; \tag{2.3}$$

A implementação do algoritmo em C baseia-se na utilização de um ciclo para o somatório exterior e outro ciclo para o somatório interior, as funções exponencial `expf` e potencia de base 2, `powf`, pertence á biblioteca, `math.h`. O código seguinte demonstra a utilização dos dois ciclos como tambem a das funções para o calculo do *smoothing*:

```
int main(){
  float sumA, sumB;
```

```
  ...

  for (int i = 0; i < N; ++i) { //percorrer o yest
    sumA = 0;
    for (int j = 0; j < N; ++j) { //percorer o input dataset
      sumA = sumA + ((expf(-powf((x[i] - x[j]), 2) / (2 * powf(SMOOTH, 2)))) * y[j
  ]);
    }
    sumB = 0;
    for (int j = 0; j < N; ++j) { //percorer o input dataset
      sumB = sumB + expf(-powf((x[i] - x[j]), 2) / (2 * powf(SMOOTH, 2)));
    }
    yest_cpu[i] = sumA / sumB;
  }

  ...
  exit(0);
}

```

3 Implementação no GPU

A implementação em GPU é dividia em 4 partes:

- Alocação da memória;
- Envio dos dados do CPU para o GPU;
- Iniciação do algoritmo em GPU;
- Envio dos dados do GPU para o CPU;

3.1 Alocação da memória

No inicio da implementação da paralisação em CUDA é necessário alocar a memória total a ser enviada do CPU para o GPU. Neste caso também foi alocado a memória total necessária para guardar o resultado do *smoothing*. De seguida apresenta-se o código para alocação da memória do *device* que com tem o GPU:

```
int main(){
  float *d_x, *d_y, *d_yest;
  ...
  cudaMalloc(&d_x, N*sizeof(float));
  cudaMalloc(&d_y, N*sizeof(float));
  cudaMalloc(&d_yest, N *sizeof(float));
  ...
  exit(0);
}

```

3.2 Envio dos dados do CPU/GPU ou GPU/CPU

Com a memória alocada , o passo seguinte é transferir os dados obtidos na secção 2.1, `X` e `Y`, para o *device*. É utilizado a função `cudaMemcpy`, que pertence à biblioteca `cuda.h`, recebe o ponteiro de destino e o ponteiro onde está a memória a ser transferida, é necessário definir a dimensão de dados a ser transferidos e por fim é necessário definir o sentido da transferência usando as seguintes mascaras, `cudaMemcpyHostToDevice` sentido do CPU para o GPU e `cudaMemcpyDeviceToHost` sentido GPU para o CPU. No código seguinte está implementado a função descrita:

```
int main(){
  float *x, *y, *yest_cpu,*yest_gpu, *randomArray;
  float *d_x, *d_y, *d_yest;
  ...
  /*Envio de dados para o device*/
  cudaMemcpy(d_x, x, N*sizeof(float), cudaMemcpyHostToDevice);
  cudaMemcpy(d_y, y, N*sizeof(float), cudaMemcpyHostToDevice);
  ...
  /*Envio de dados para o CPU*/
  cudaMemcpy(yest_gpu, d_yest, N*sizeof(float), cudaMemcpyDeviceToHost);
  ...
exit(0);
}

```

3.3 Iniciação do algoritmo em GPU

Depois de enviar todos os dados para o GPU o *kernel* está pronto para ser invocado. O *kernel* representa o código que vai ser executado pela GPU, este é definido pela declaração `__global__` antes da função C. Ver código seguinte,

```
__global__ void funtion_smooth(float *x, float *y, float *yest, int n){
...
}

```

Com o *kernel* definido este é executado usando a seguinte configuração,

$$NomeDaFuncao <<< NB, NT >>> \tag{3.1}$$

Onde `NB` é o numero de blocos a ser lançados no GPU e `NT` o numero de *threads* por bloco.

4 Técnicas de aceleração e optimização

O código a ser paralisado é referente à segunda secção, *Smoothing* do sinal amostrado, do capítulo, Implementação no CPU. Analisou-se a estrutura do algoritmo e verificou-se a possibilidade de optimização e paralisação dos ciclos.

4.1 Optimização

Analisando o algoritmo proposto identificou-se duas situações principais de optimização, número de ciclos e acesso á memória. Começou-se então por reduzir o número de ciclos do algoritmo, os dois ciclos interiores podem ser reduzidos a um só, com esta alteração verificou-se que se podia reduzir o acesso á memória. Isto é, como as variáveis `sumA` e `sumB` calculam-se da mesma forma tirando a diferença de `sumA` ser multiplicada por `y[i]`. Fez-se a seguinte alteração, a parte comum é calculada em primeira instância e o resultado é guardado numa variável auxiliar, `sum`. Assim para o calculo de `sumB` é só necessário aceder a cache e obter os valores `sumB` e `sum` e para o calculo de `sumA` acede de igual forma, vai à cache retirar os valores de `sumA` e `sum` e um acesso à memória global, `y[j]`. O código de seguida demonstra a explicação feita anteriormente:

```
for (int i = 0; i < N; ++i) {
  for (int j = 0; j < n ;j++){
    sum = (expf(-powf((x[i] - x[j]), 2) / (2 * powf(SMOOTH, 2))));
    sumA = sumA +  sum* y[j];
    sumB = sumB + sum;
    }
  yest[i] = sumA / sumB;
}
```

4.2 Paralisação dos ciclos

Analisando a optimização do código anterior verificou-se que pode ser paralisado em duas situações:

- Paralisação do ciclo externo;
- Paralisação do ciclo externo e interno;

Para melhor compreensão, no código seguinte está representado qual o ciclo externo e qual o ciclo interno.

```
for (int i = 0; i < N; ++i) {/*Ciclo Externo*/
  for (int j = 0; j < n ;j++){/*Ciclo Interno*/
    sum = (expf(-powf((x[i] - x[j]), 2) / (2 * powf(SMOOTH, 2))));
    sumA = sumA +  sum* y[j];
    sumB = sumB + sum;
  }
  yest[i] = sumA / sumB;
}
```

4.2.1 Paralisação do ciclo externo

Analisando o ciclo externo verificou-se que não existe dependências de dados sendo por isso a primeira implementação a ser realizada. Assim sendo cada iteração do ciclo externo é executada pelas *threads* dos blocos chamados. Como o GPU utilizado é composto por 1024 *threads* por bloco, quando se pretende processar `N` iterações é necessário saber o número de blocos a serem chamados para isso

divide-se o número máximo de iterações pelo número máximo de *threads* por blocos. De seguida está implementado o o código do *kernel* que corre no GPU e a função de chamada pelo CPU.

```
1
2  __global__ void funtion_smooth(float *x, float *y, float *yest, int n){
3    int i = blockIdx.x* blockDim.x + threadIdx.x;
4    float sumA=0.0, sumB=0.0,sum=0.0;
5
6    if (i < n){
7      for (int j = 0; j < n ;j++){
8        sum = (expf(-powf((x[i] - x[j]), 2) / (2 * powf(SMOOTH, 2))));
9        sumA = sumA +  sum* y[j];
10       sumB = sumB + sum;
11     }
12     yest[i] = sumA / sumB;
13   }
14 }
15 int main() {
16   ...
17   dim3 dimBlock(BLOCK_SIZE, 1, 1);
18   dim3 dimGrid(N / BLOCK_SIZE + 1, 1, 1);
19   funtion_smooth <<< dimGrid, dimBlock >>>(d_x, d_y, d_yest, N);
20   ...
21   exit(0);
22 }
```

O *kernel* a ser executado é composto pelo ciclo interno, ou seja, cada *thread* executa o *smoothing* de cada amostra do sinal. É de salientar que a variável `i` identifica qual a iteração que o ciclo exterior executava, para isso seguiu-se o seguinte esquema,

Figura 1: Esquema para indexação única.

Como o número de identificação de cada *thread* só identifica a *thread* dentro de cada bloco é necessário obter uma indexação única e igual à do ciclo exterior paralisado. Assim sendo, e analisando a figura anterior obteve-se a seguinte expressão para o calculo do de `i`, `i = blockIdx.x*blockDim.x + threadIdx.x;`. Onde o `blockIdx.x` é o identificador do bloco, `blockDim.x` dimensão de cada bloco e `threadIdx.x` é o identificador de cada *thread*.

4.2.2 Paralisação do ciclo interno

Ao contrário do ciclo externo o ciclo interno tem dependência de dados dificultando assim a paralisação desse mesmo. Para a paralisação interna optou-se por dividir o número máximo de iterações

em quatro partes, reduzindo assim o tempo de processamento em cada *thread*. O código seguinte representa a implementação referida:

```
__global__ void funtion_smooth(float *x, float *y, float *yest, int n){
  int i = blockIdx.x* blockDim.x + threadIdx.x;
  int j = 0;
  float sumA=0.0, sumB=0.0,sumA_Total=0.0, sumB_Total=0.0, sum = 0.0;
  __shared__ float sumA_partial[BLOCK_SIZE];
  __shared__ float sumB_partial[BLOCK_SIZE];

  if ((i/MULTHREADS) < n){
    if(!(threadIdx.x&0x02)){
      if(!(threadIdx.x&0x01)){
        for (j = 0; j < (n/MULTHREADS); j++){
          sum = (expf(-powf((x[i>>2] - x[j]), 2) / (2 * powf(SMOOTH, 2))));
          sumA = sumA + sum* y[j];
          sumB = sumB + sum;
        }
      }else{
        for (j = (n/MULTHREADS); j < 2*(n/MULTHREADS); j++){
          sum = (expf(-powf((x[i>>2] - x[j]), 2) / (2 * powf(SMOOTH, 2))));
          sumA = sumA + sum* y[j];
          sumB = sumB + sum;
        }
      }
    }else{
      if(!(threadIdx.x&0x01)){
        for (j = 2*(n/MULTHREADS); j < 3*(n/MULTHREADS); j++){
          sum = (expf(-powf((x[i>>2] - x[j]), 2) / (2 * powf(SMOOTH, 2))));
          sumA = sumA + sum* y[j];
          sumB = sumB + sum;
        }
      }else{
        for (j = 3*(n/MULTHREADS); j < 4*(n/MULTHREADS); j++){
          sum = (expf(-powf((x[i>>2] - x[j]), 2) / (2 * powf(SMOOTH, 2))));
          sumA = sumA + sum* y[j];
          sumB = sumB + sum;
        }
    }
  }
  sumA_partial[threadIdx.x]= sumA;
  sumB_partial[threadIdx.x]= sumB;
  ...
}

```

Como se pode verificar no código anterior, existe quatro ciclos com um quarto do número máximo de iterações. Ao dividir o processamento em quatro *threads* é necessário ter uma memória partilhada:

```
__global__ void funtion_smooth(float *x, float *y, float *yest, int n){
...
  __shared__ float sumA_partial[BLOCK_SIZE];
  __shared__ float sumB_partial[BLOCK_SIZE];
...
}
```

Para que a organização de cada *thread* seja executa de forma correcta é necessário organizar as *threads* de forma coerente, assim sendo é verificado os últimos dois *bits* menos significativos da variável `thread.idx` de forma a executar o ciclo correcto, por exemplo, se a variável `thread.idx` for igual a 1 então executa o 2 ciclo.

Com os dados guardados na memória partilhada é necessário sincronizar todo de forma a obter o resultado acumulado final. Para a sincronização é o utilizado a função de sistema `__syncthreads();`, que sincroniza todas as *threads* do mesmo bloco. Depois da sincronização pode-se acumular os resultados de todas as quatro *threads* para isso é utilizado a *thread* que tem os dois *bits* menos significativos a zero, ou seja, a primeira *thread* de cada grupo de quatro. De seguida está demonstrado o código:

```
  ...
  if(!(threadIdx.x&0x03)){
    for(int l=0 ; l<MULTHREADS;l++){
    sumA_Total += sumA_partial[threadIdx.x + l];
    sumB_Total += sumB_partial[threadIdx.x + l];
  }
  yest[i>>2] = sumA_Total / sumB_Total;
}
```

5 Secção de resultados

Depois da implementação anteriormente referida foi analisado os tempos de execução

5.1 Tempos

Para o cálculo dos tempos foi usado a função `clock_gettime()` e a função `timeDiff()`. Todas as estruturas e funções não representadas estão na biblioteca `sys/time.h`.

```
  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;
  }
```

Para o calculo do tempo de execução em CPU foi calculado o intervalo de tempo desde a primeira iteração do ciclo exterior até á ultima tiração.

```
int main(){
  ...
  struct timespec timeVect[7];
  double timeCPU;
  ...
  clock_gettime(CLOCK_REALTIME, &timeVect[0]);/*Inicio da contagem do tempo*/

  for (int i = 0; i < N; ++i) { //percorrer o yest
    sumA = 0;
    for (int j = 0; j < N; ++j) { //percorer o input dataset
      sumA = sumA + ((expf(-powf((x[i] - x[j]), 2) / (2 * powf(SMOOTH, 2)))) * y[j]);
    }
    sumB = 0;
    for (int j = 0; j < N; ++j) { //percorer o input dataset
      sumB = sumB + expf(-powf((x[i] - x[j]), 2) / (2 * powf(SMOOTH, 2)));
    }
    yest_cpu[i] = sumA / sumB;
  }
  clock_gettime(CLOCK_REALTIME, &timeVect[1]);/*Fim da contagem do tempo*/
  timeCPU = timeDiff(timeVect[0], timeVect[1]);
  printf("    ... execution took %.6f seconds\n", timeCPU);
  ...
  exit(0);
}
```

O calculo do tempo de execução do GPU referido foi obtido somando o tempo de alocação de memória, o tempo de transferência de dados do CPU para o GPU, o tempo de execução do *kernel* e a transferência dos resultados do GPU para o CPU.

```
int main(){
...
  struct timespec timeVect[7];
  double timeCPU,timeGPU[7];
  ...
  clock_gettime(CLOCK_REALTIME, &timeVect[1]);/*Inicializacao do contador*/

  cudaMalloc(&d_x, N*sizeof(float));
  cudaMalloc(&d_y, N*sizeof(float));
  cudaMalloc(&d_yest, N *sizeof(float));
  clock_gettime(CLOCK_REALTIME, &timeVect[2]);/*Tempo de alocacao de memoria*/

  cudaMemcpy(d_x, x, N*sizeof(float), cudaMemcpyHostToDevice);
  cudaMemcpy(d_y, y, N*sizeof(float), cudaMemcpyHostToDevice);
  clock_gettime(CLOCK_REALTIME, &timeVect[3]);/*Tempo de transferencia de dados CPU/
    GPU*/

```

```
17    dim3 dimBlock(BLOCK_SIZE, 1, 1);
18    dim3 dimGrid(N / BLOCK_SIZE + 1, 1, 1);
19
20    funtion_smooth <<< dimGrid, dimBlock >>>(d_x, d_y, d_yest, N);
21    clock_gettime(CLOCK_REALTIME, &timeVect[4]);/*Tempo de execucao do kernell*/
22
23    cudaMemcpy(yest_gpu, d_yest, N*sizeof(float), cudaMemcpyDeviceToHost);
24    clock_gettime(CLOCK_REALTIME, &timeVect[5]);/*Tempo de transferencia de dados GPU/
        CPU*/
25    ...
26    exit(0);
27 }
```

5.1.1 Paralisação do ciclo externo

De seguida está implementado uma tabela com os resultados dos tempos de execução da paralisação do ciclo externo em função do número de amostras. Também se encontra representado na tabela o valor do *speed-up* obtido com a implementação do GPU. O *speed-up* é calculado pela seguinte equação, tmp_{cpu} tempo de execução no CPU e tmp_{gpu} tempo de execução no GPU.

$$speedup = \frac{tmp_{cpu}}{tmp_{gpu}}; \tag{5.1}$$

Tabela 1: Valores dos tempos de execução no CPU e GPU para diferentes valores de amostras.

Nº Amostras	Tempo no CPU (segundos)	Tempo no GPU (segundos)	SpeedUp
1000	0,058	0,0014	40,2
10000	3,94	0,0197	200,4
20000	15,184	0,0753	201,7
30000	33,757	0,1496	225,7
40000	59,644	0,2167	275,3

De seguida também está implementado o gráfico do *speed-up* em função do número de amostras.

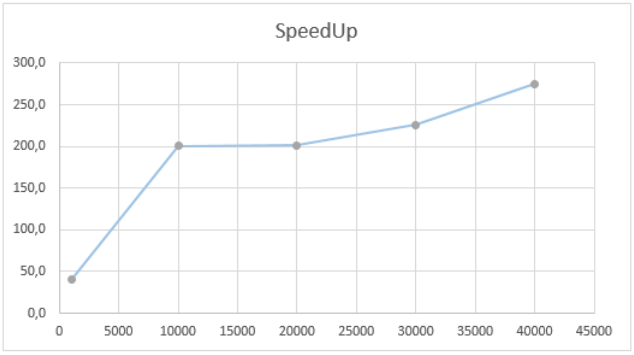

Figura 2: Gráfico do *speed-up* em função das amostragem.

5.1.2 Paralisação do ciclo externo e interno

Está representado os resultados dos tempos de execução referentes á paralisação do ciclo externo e interno. Também se encontra representado na tabela o valor do *speed-up* obtido com a implementação do GPU. O *speed-up* é calculado pela equação anterior 5.1.

Tabela 2: Valores dos tempos de execução no CPU e GPU para diferentes valores de amostras.

Nº Amostras	Tempo no CPU (segundos)	Tempo no GPU (segundos)	SpeedUp
1000	0,061	0,0015	41,7
10000	3,94	0,0580	67,9
20000	15,19	0,2287	66,4
30000	33,731	0,4822	70,0
40000	59,621	0,8121	73,4

De seguida também está implementado o gráfico do *speed-up* em função do número de amostras.

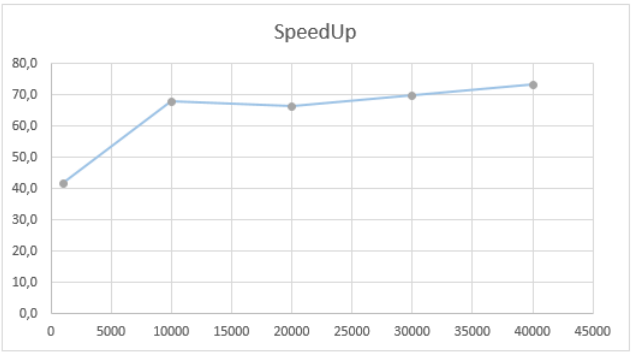

Figura 3: Gráfico do *speed-up* em função das amostragem.

5.2 Resultados do *Smooth*

Está representado nos gráficos seguintes os resultados das amostras processadas pelo CPU e GPU com 20000 amostras.

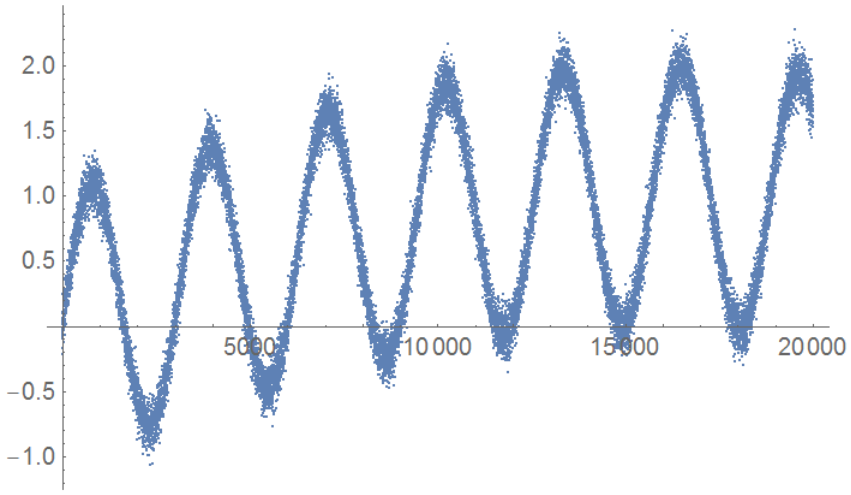

Figura 4: Gráfico do sinal amostrado com ruído.

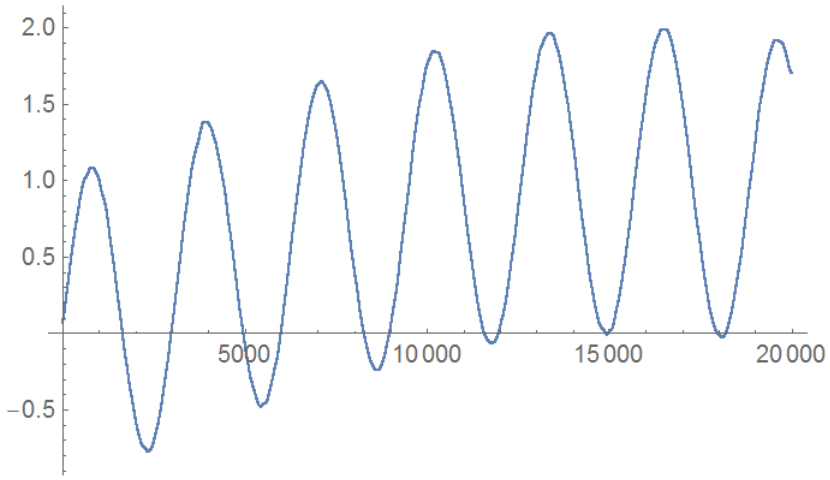

Figura 5: Gráfico do sinal amostrado com *smoothing* do ruído implementado em CPU.

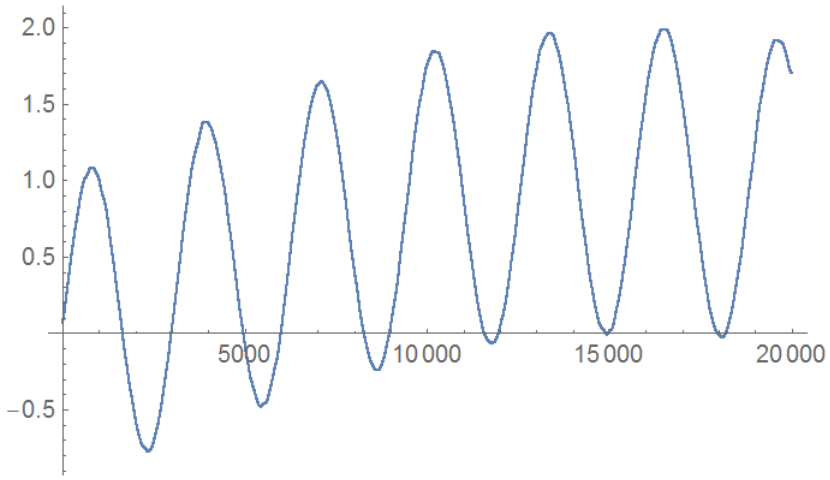

Figura 6: Gráfico do sinal amostrado com *smoothing* do ruído implementado em GPU.

6 Conclusões

A opitmização escolhida para este algoritmo foi a implementação da paralisação só no ciclo externo, esta escolha é referente aos resultados dos *speed-ups* anteriormente referidos, pois existe um *speed-up* três vezes superior ao da paralisação do ciclo externo e interno. Estes resultado era de esperar devido ao *overhead* que existe na sincronização das*threads* e acesso á memória partilhada. Mesmo com os maus resultados obtidos foi importante implementar esta segunda paralisação devido à complexidade e a utilização de diversas funções do CUDA.

7 Anexos

Descrição dos ficheiros em anexo: `Smooth_1v.cu` ficheiro que contem o código da primeira paralisação, paralisação do ciclo externo. `Smooth_2V.cu` ficheiro que contem o código da segunda para-

lisação, paralisação do ciclo externo e interno.

Na pasta `Datasets` estão os resultados obtidos para 1000, 10000 e 20000 amostras.