
Instituto Superior Técnico

Mestrado Integrado em Engenharia Electrotécnica e de Computadores


# Arquiteturas Avançadas de Computadores

# Paralelização e aceleração de um programa


| | |
|---|---|
| Guilherme ..... Maria Margarida Dias dos Reis | n.º 73099 |
| Nuno Miguel Rodrigues Machado | n.º 74236 |


Lisboa, 3 de Maio de 2015

# Índice

# 1 Introdução

# 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:

```
__global__ void funtion_smooth(float *x, float *y, float *yest, int n){
  int i = blockIdx.x* blockDim.x + threadIdx.x;
  float sumA=0.0, sumB=0.0,sum=0.0;

  if (i < n){
  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

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