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 dos dados de entrada;
- Obtenção do *Smooth* com processamento dos dados de entrada;

2.1 Criação dos dados de entrada

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 Obtenção do *Smooth* com processamento dos dados de entrada

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 para o GPU

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`, função que pertence à biblioteca `cuda.h`. Função de e

3.3 Iniciação do algoritmo em GPU

3.4 Envio dos dados do GPU para o CPU

4 Técnicas de aceleração e otimização

5 Secção de resultados

5.1 Tempos

5.2 Resultados do *Smooth*

6 Conclusões