ELC139 - Programação Paralela
Trabalho 8: Geração de Imagem em Paralelo com CUDA

Lucas Roges de Araujo[1]

[1]Curso de Ciência da Computação
Universidade Federal de Santa Maria

18 de Junho de 2019

Parte 1

- 1. Implemente um programa nomeado wavecuda1.cu, em que cada frame seja computado em paralelo por uma thread diferente.
  - Paralelização do laço mais externo.
- 2. Analise o desempenho do programa, começando pela medição dos tempos de execução sequencial e paralela para pelo menos 4 diferentes entradas do programa, sendo: a) 1024 100, b) 1024 200 e outras 2 entradas à sua escolha. Compare os tempos obtidos. Use nvprof para analisar o perfil de execução em GPU. Explique seus resultados.

Código

```
// Kernel function to calculate the value of each pixel
__global__
void calculatePixels(int width, int frames, unsigned char* pic)
{
  int index = threadIdx.x; // id da thread
  int offset = blockDim.x; // tamanho do bloco (número total de threads, nesse caso)
  for (int frame = index; frame < frames; frame += offset) {
  //for (int frame = 0; frame < frames; frame++) {
    for (int row = 0; row < width; row++) {
      for (int col = 0; col < width; col++) {
        float fx = col - 1024/2;
        float fy = row - 1024/2;
        float d = sqrtf( fx * fx + fy * fy );
        unsigned char color = (unsigned char) (160.0f + 127.0f *
                                               cos(d/10.0f - frame/7.0f) /
                                               (d/50.0f + 1.0f));

        pic[frame * width * width + row * width + col] = (unsigned char) color;
      }
    }
  }
}
```

```
unsigned char* pic;
cudaMallocManaged(&pic, frames*width*width*sizeof(unsigned char));

// start time
timeval start, end;
gettimeofday(&start, NULL);

calculatePixels<<<1, frames>>>(width, frames, pic);

// Wait for GPU to finish before accessing on host
cudaDeviceSynchronize();
```

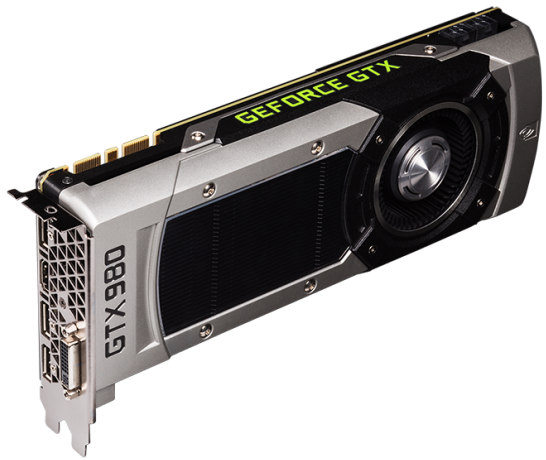
Metodologia

Parâmetros de execução:

- Código:
  - C++ sequencial.
  - CUDA sequencial.
  - CUDA paralelo.
- Largura das imagens:
  - 1024 e 2048.
- Quantidade de *frames*/*threads*:
  - 100, 200 e 400.

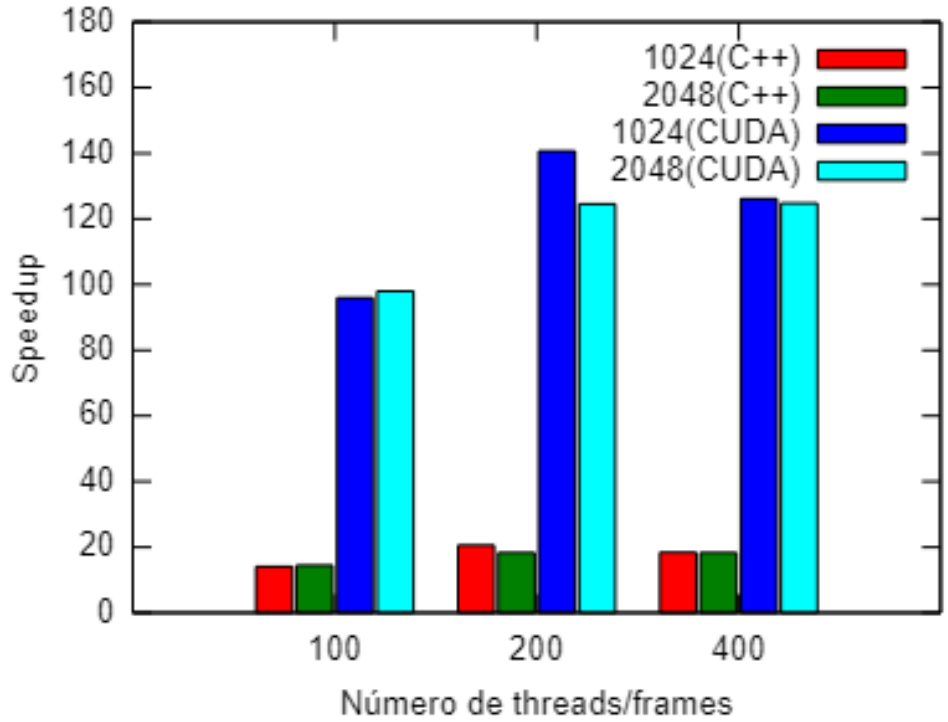
Ambiente de execução:

- *Cluster* CDER (nó cder01).
- GeForce GTX 980
  - 2048 CUDA cores.
  - 1.126 GHz.



Resultados

- Há melhora no tempo de execução para a versão implementada.
  - Esse *speedup* é maior em relação a versão sequencial em CUDA.
    - Frequência.
    - Transferência de dados.



Parte 2

- 1. Implemente um programa nomeado wavecuda2.cu, que aproveite melhor o paralelismo da GPU.
  - Paralelização laço de linhas.
  - Utilização de múltiplos blocos.
- 2. Analise o desempenho do segundo programa.

Código

```
// Kernel function to calculate the value of each pixel
__global__
void calculatePixels(int width, int frames, unsigned char* pic)
{
  int index = blockIdx.x * blockDim.x + threadIdx.x;
  int offset = blockDim.x * gridDim.x;
  for (int frame = 0; frame < frames; frame++) {
    for (int row = index; row < width; row += offset) {
      for (int col = 0; col < width; col++) {
        float fx = col - 1024/2;
        float fy = row - 1024/2;
        float d = sqrtf( fx * fx + fy * fy );
        unsigned char color = (unsigned char) (160.0f + 127.0f *
                                               cos(d/10.0f - frame/7.0f) /
                                               (d/50.0f + 1.0f));
        pic[frame * width * width + row * width + col] = (unsigned char) color;
      }
    }
  }
}
```

```
// allocate picture array
unsigned char* pic;
cudaMallocManaged(&pic, frames*width*width*sizeof(unsigned char));

// start time
timeval start, end;
gettimeofday(&start, NULL);

int blockSize = threads;
int numBlocks = (width + blockSize - 1) / blockSize;

calculatePixels<<<numBlocks, blockSize>>>(width, frames, pic);

// Wait for GPU to finish before accessing on host
cudaDeviceSynchronize();
```

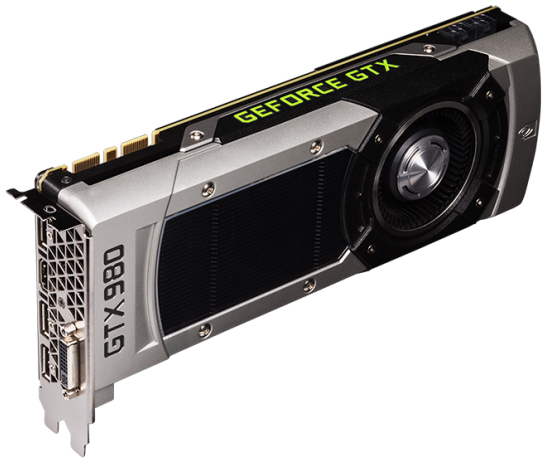
Metodologia

Parâmetros de execução:

- Largura das imagens:
  - 1024 e 2048.
- Quantidade de *frames*:
  - 100, 200 e 400.
- Quantidade de *threads*:
  - 16, 32, 64, 128, 256.

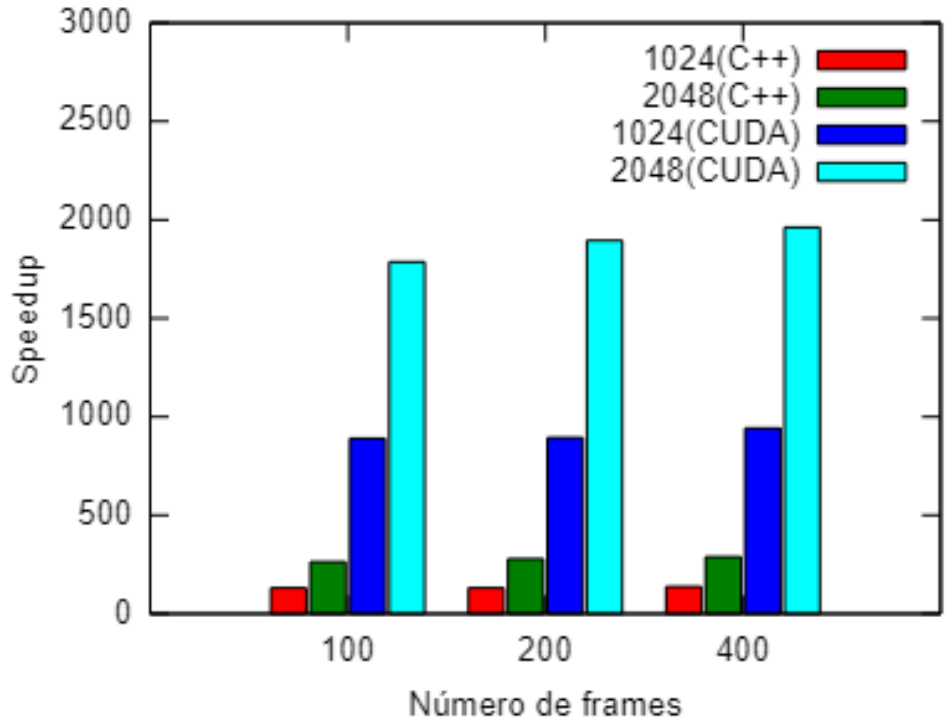
Ambiente de execução:

- *Cluster* CDER.
- GeForce GTX 980 (cder01)
  - 2048 CUDA cores.
  - 1.126 GHz.



Resultados

- Há melhora no tempo de execução para a versão implementada.
- Em geral, o *speedup* tem uma eficiência maior para todos os casos, em relação a eficiência para o caso da parte 1.



Referências


CUDA C Programming Guide
https://docs.nvidia.com/cuda/cuda-c-programming-guide/


Experiência com grids, blocks e threads em CUDA
https://colab.research.google.com/drive/1uSTM6C0p4n4aAuvFksplqFxa4NG87rMp


Unified Memory for CUDA Beginners
https://devblogs.nvidia.com/unified-memory-cuda-beginners/


How to Submit a Job on CDER
https://help.rs.gsu.edu/display/PD/How+to+Submit+a+Job+on+CDER


Boas Práticas para Experimentos Computacionais em Clusters de Alto Desempenho
https://github.com/viniciusvgp/tutorial-mc-erad-2019


GPGPU: How does running sequential code on a GPU compare to running sequential code on a CPU?
https://www.quora.com/GPGPU-How-does-running-sequential-code-on-a-GPU-compare-to-running-sequential-code-on-a-CPU

ELC139 - Programação Paralela
Trabalho 8: Geração de Imagem em Paralelo com CUDA

Lucas Roges de Araujo[1]

[1]Curso de Ciência da Computação
Universidade Federal de Santa Maria

18 de Junho de 2019