Segmentação de Imagem

*Desempenhos variados conforme execução sequencial ou em GPU*

*Gabriel do Vale Rios, Engenharia de Computação. Insper-2018.*

***Introdução***

Seja por motivos profissionais, ou apenas um *hobby*, é necessário que exista meios de se destacar áreas específicas em uma imagem. Destacar no caso seria identificar uma região da imagem e destacá-la, a fim de se trabalhar somente com o que é necessário.

Assim, este trabalho busca comparar o desempenho de segmentação de imagem sequencial x paralela, utilizando da técnica de filtragem de borda, para maior precisão e melhor desempenho.

***Implementação***

Todo código citado neste documento pode ser baixado e executado a partir do endereço abaixo:

<https://github.com/GabrielValeRios/superComp2018/tree/master/projeto4>

Para a implementação e gestão de todo o código feito para esta simulação, se utilizou a linguagem C++, python e Cmake. O hardware utilizado para compilação e execução das simulações segue abaixo, junto de suas informações:

* Processador: Intel® Core™ i7-4500U CPU @ 1.80GHz 2.40GHz
* Memória (RAM): 16.0 GB
* Tipo de Sistema: Sistema Operacional de 64 bits, processador com base em x64, 4 cores
* Sistema Operacional utilizado: Distribuição Linux (Ubuntu 18.04 LTS)
* Placa Nvidia GeForce - 840M

O código busca segmentar uma imagem a partir de *Seeds*, que são pontos específicos na imagem que servem para diferenciar a região que se quer destacar do plano de fundo. Assim, podem ser fornecidas múltiplas seeds do tipo f\_g, para destacar diferentes regiões, e múltiplas seeds do tipo b\_g, para informar diferentes regiões que pertencem ao plano de fundo. Com a diferenciação da região desejada do plano de fundo, é utilizado o algoritmo de Dijkstra, para de fato separar as regiões. Tal algoritmo se traduz no trabalho como SSSP (*Single Source Shortest Path)*

Como imagem *source*, pode ser dado uma imagem com extensão .pgm. Está imagem, para que a separação das regiões seja facilitada, pode passar por uma filtragem de borda, onde a imagem resultante é igual à de entrada, porém com bordas que definem melhor regiões diferentes. Para isso, foi utilizada a biblioteca “thrust”, responsável por transferir os dados a serem calculados para a GPU, e junto de ferramentas do CUDA, criar um um kernel dedicado para executar a função de bordas em si. A execução de dados em GPU é mais rápida, justificando o aumento de performance pelas múltiplas threads que podem ser geradas na GPU. O ganho de tempo é justificado pela quantidade massiva de threads em si, e não pela performance individual de cada thread.

Link: https://www.magazineluiza.com.br/bateria-e-percussao/instrumentos-musicais/s/im/baoa/

***- Arquivos em C++***

Os arquivos deste projeto para gerar os executáveis, em GPU, são:

* Imagem.cpp, responsável pela leitura de uma imagem, geração de uma nova imagem e fornecer diversos dados sobre a imagem (tamanho, pixels, etc)
* Main.cpp, responsável por executar o algoritmo de SSSP da Nvidia nas imagens, medições de tempo para comparação de desempenho e montar os vetores necessários para o código da Nvidia.
* Border\_filter.cu, para transformar uma imagem .pgm em outra imagem .pgm, com bordas.

Para gerar o executável sequencial:

* Imagem.cpp, idem.
* Main.cpp, responsável por executar o algoritmo de SSSP sequencialmente.

A ideia é compara o quão mais rápido o processamento em GPU é mais rápido do que o processamento sequencial. Para isso, serão medidos os tempos de cálculo do SSSP e o tempo de montagem da imagem resultante. Tal imagem irá destacar as áreas selecionadas através das seeds f\_g.

***- Arquivos em Python***

Para que a simulação dos testes fosse feita de maneira menos tediosa, existem arquivos em python que ajudam a realizar os testes:

* runSEQ\_SEQ.py, para rodar a versão sequencial do SSSP
* installCuda.py, para instalar dependências
* runBorderFile.py, para gerar a imagem com bordas
* runGPU\_SEG.py, para rodar a versão em GPU do SSSP

NOTA: Mais informações de como realizar os testes e usar seus resultados para interpretação podem ser encontradas em README, no link:

<https://github.com/GabrielValeRios/superComp2018/tree/master/projeto4>

**Simulação e Comparação**

Para a simulação e análise de resultados, foi escolhida a categoria “Bateria e Percussão”, de instrumentos musicais. O link para acesso é:

<https://www.magazineluiza.com.br/bateria-e-percussao/instrumentos-musicais/s/im/baoa/>

Ao fim de cada simulação, é gerado um arquivo de texto chamado “testes.txt”, e nele estão contidos:

* Tempo de cada produto
* Tempo total de simulação (TotalTime)
* Tempo total ocioso (WaitTime)

Para realizar as simulações, foi fixado um número de 2 processos por máquina. Este número foi escolhido baseado no número de produtos no link de exemplo usado, para evitar erros de execução. Para outros links, é possível que o número de processos por máquina seja maior.

O tempo total de simulação foi reduzido ao processo mais demorado dentre os existentes, pois, por estar rodando em um cluster, a latência entre mensagens é desprezível. Mais informações sobre latência podem ser conferidas no link abaixo.

https://computing.llnl.gov/mpi/mpi\_benchmarks.html

Assim, para um número constante de 2 processos por máquina, com um número crescente de máquinas (até 7 máquinas rodando simultaneamente no cluster), foi feito uma análise de desempenho do algoritmo. Abaixo, segue o resultado de desempenho (tempo total de execução) conforme o número de máquinas rodando no cluster.

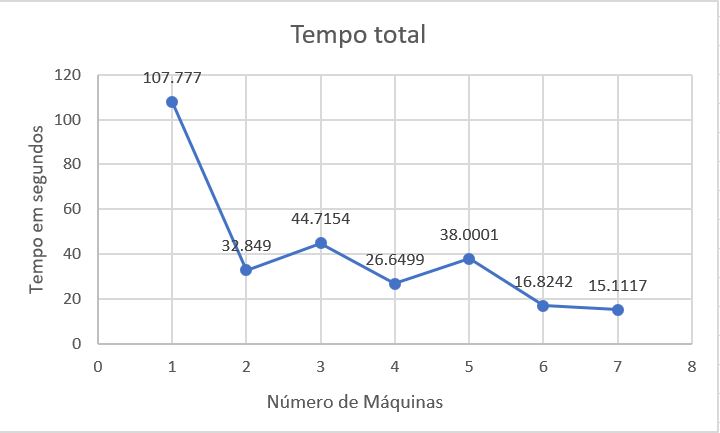


Figura Desempenho por máquina no cluster

**Análise e Conclusão**

A partir do gráfico analisado, pode-se perceber um grande aumento de desempenho apenas ao se adicionar 2 máquinas ao cluster ao invés de 1 só. Isso era esperado, pois com 1 máquina não existe divisão de trabalho, e com 2 já é possível distribuir a carga de trabalho. Conforme se aumenta o número de máquinas, é visto que existe um aumento de desempenho em relação a 1 máquina só. Porém, para os casos onde o número de máquinas é igual a 3 e 5, existe um pequeno aumento no tempo de execução. Porém no geral, houve uma grande baixa no tempo de execução do crawler, de até 7,12 vezes no caso em que o número de máquinas é igual a 7, por exemplo.

Conclui-se, portanto, que, conforme o cluster possui mais máquinas, o desempenho aumenta consideravelmente. O tempo de execução, para esse teste, mostrou uma melhora de até 7,12 vezes. Existiram alguns casos onde o aumento de máquinas aumentou um pouco o tempo da simulação anterior, porém o aumento de tempo não gerou um desempenho tão destoante dos demais.