Universidade de São Paulo
Instituto de Matemática e Estatística
Departamento de Ciência da Computação

MAC0219: Introdução à Programação Concorrente, Paralela e Distribuída

EP2: Criptografia em GPUs usando CUDA

Professor: Alfredo Goldman

Monitor: Pedro Bruel

Bárbara Fernandes - 7577351
Duílio Elias - 6799722
Eduardo Bier - 8536148

São Paulo - SP, 12 de junho de 2017

Sumário

Lista de Figuras

1. Introdução

Este relatório faz parte do segundo exercício-programa da matéria de Introdução à Programação Concorrente, Paralela e Distribuída, lecionada pelo professor Alfredo Goldman. Neste exercício, foram realizados experimentos utilizando **CUDA** na implementação de algoritmos simples de encriptação e geração de *hashes* de arquivos. Como proposto, foram escolhidos três algoritmos de encriptação: **ROT13**, **Base64** e **RC4**, que serão executados em *GPUs* da NVIDIA. Nosso objetivo é mostrar o quanto se pode melhorar o desempenho da geração de *hashes* de arquivos com o uso de GPUs e com a paralelização do algoritmo usando CUDA, analisar o efeito tamanho do arquivos, trabalhar com GPUs e, por fim, verificar se há divergência entre as versões sequenciais e paralelas dos algoritmos de encriptação e qual o impacto do *I/O* no tempo de execução.

2. Criptografia

Para testarmos a diferença de desempenho entre o programa sequencial e o paralelizado foram escolhidos três algoritmos de encriptação. Dadas as suas implementações sequenciais, solicitou-se que implementássemos as versões paralelizadas utilizando a ferramenta CUDA. Os três algoritmos criptográficos de nossa escolha foram ROT13, Base64 e Blowfish.

2.1. ROT13

ROT13 é um algoritmo simples de criptografia em que cada caractere de um texto é substituído pelo caractere 13 letras depois no alfabeto. Assim, este algoritmo não fornece nenhum tipo de segurança criptográfica e pode ser decodificado por qualquer um. Por causa disso, seu uso é limitado a apenas dificultar a leitura de um texto, sendo usado, por exemplo, em fóruns para esconder *spoilers*.

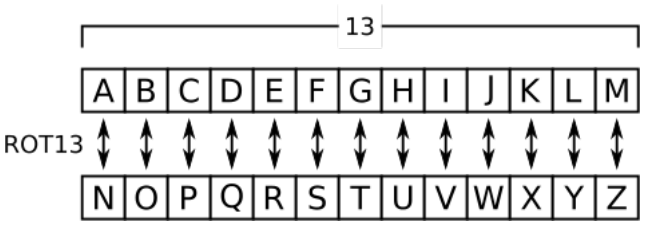


Figura 1: Codificação usando ROT13

A codificação usada nesse algoritmo é extremamente simples e rápida. Cada letra de um texto é substituída pela letra 13 posições depois no alfabeto:

$$\text{ROT13}(c) \equiv c + 13 \bmod 26$$

Pela equação 1, vê-se que a função ROT13 é o inverso dela mesma e portanto, a decodificação usa o mesmo algoritmo da codificação.

$$\begin{aligned} ROT13(ROT13(c)) &\equiv ROT13(c) + 13 \bmod 26 \\ &\equiv (c + 13 \bmod 26) + 13 \bmod 26 \\ &\equiv c + 26 \bmod 26 \\ &\equiv c + 0 \bmod 26 \\ &\equiv c \end{aligned} \tag{1}$$

2.2. Base64

Base64 é um tipo de codificação que representa *bytes* como uma *string* ASCII. Nessa codificação, cada três *bytes* do arquivo original são transformados em quatro caracteres no arquivo codificado. A codificação acontece da seguinte forma: grupos de três *bytes* são concatenados de modo a formar uma *string* de 24 *bits*, que é então divida em grupos de 6 *bits*. Cada um desses novos grupos é então convertido em um caractere usando uma tabela (com 64 valores possíveis). Repetindo-se esse processo para todos os *bytes* do arquivo, obtém-se a codificação Base64. Caso o arquivo tenha um número de *bytes* não divisível por 4, um preenchimento é adicionado para completar os *bytes* restantes.

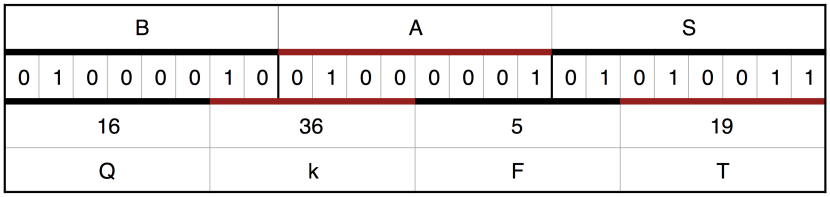


Figura 2: Processo de codificação da string "BAS" usando Base64.

Para decodificar um arquivo, o processo contrário precisa ser feito. Primeiro, cada caractere é convertido em um número usando a tabela inversa usada para a codificação. Grupos de 4 caracteres convertidos são concatenados formando uma *string* de 24 *bits* que é então separada em grupos de 8 *bits*. Cada um desses grupos de 8 *bits* representa 1 *byte* do arquivo original.

2.3. RC4

RC4 é um algoritmo de encriptação formulado para ser rápido, eficiente e de fácil implementação. Sua segurança varia conforme o tamanho de sua

chave. Aumentando-se o tamanho de sua chave, que geralmente tem entre 1 a 256 *bytes*, garantimos uma maior segurança, mas seu desempenho será penalizado. O programador deve, então, encontrar o equilíbrio desejado entre segurança e velocidade.

Apesar de podermos melhorar sua segurança através do aumento do tamanho de sua chave, foram encontradas múltiplas vulnerabilidades neste algoritmo, recomendando-se, então, o uso de esquemas de encriptação mais seguros conforme a necessidade de sigilo requerida pelas informações a serem transmitidas.

Este algoritmo de encriptação funciona em duas fases: a configuração da chave e a encriptação. A configuração da chave é a primeira fase deste algoritmo. Nela, uma chave de tamanho variável (entre 1 e 256 *bytes*) é utilizada para inicializar um vetor de estados de 256 *bytes*.

Esse vetor irá conter uma permutação de todos os números de 8 bits, de 0 a 255. Para a encriptação e decriptação, um *byte* k é gerado selecionando uma das 255 entradas do vetor de estados. De acordo com o valor de k gerado, os elementos são novamente permutados. Cada um de seus elementos é trocado pelo menos uma vez. Por fim, é feita uma operação *XOR* do vetor de permutações com o arquivo que queremos encriptar.

Para a decriptação o procedimento é análogo: fazemos um *XOR* do texto encriptado com o vetor de permutações para obter o texto original.

3. Tecnologias

3.1. NVIDIA

NVIDIA é a empresa líder mundial no mercado de GPUs (*Graphics Processing Units*). Parte de seu renome deve-se aos seus produtos direcionados ao mercado de jogos de computadores. Apesar de ser considerada por muitos como a inventora da GPU, a companhia teve sua patente rejeitada em um caso judicial onde acusava a Samsung de violações de direitos autorais.

3.2. GPU

GPUs (*Graphics Processing Unit*) são processadores multicore de alto desempenho capazes de computações bastante complexas e de elevadas taxas de tranferência de dados. As GPUs da NVIDIA são construídas sob o que é conhecido como **arquitetura CUDA**.

3.3. CUDA

CUDA (*Compute Unified Device Architecture*) é uma arquitetura de computação paralela e um modelo de programação que inclui uma linguagem de montagem (PTX) e uma tecnologia de compilação que é a base na qual

várias interfaces de linguagens paralelas e de APIs são construídas nas GPUs da NVIDIA. Através dela, GPUs podem realizar tanto tarefas clássicas de renderização gráfica quanto tarefas de propósito geral.

Para programar GPUs em CUDA utiliza-se uma linguagem conhecida como CUDA C, que nada mais é do que a linguagem C com diversas extensões, de forma a permitir a programação de máquinas massivamente paralelas como as GPUs da NVIDIA.

As mais importantes características trazidas por essa nova tecnologia são a memória compartilhada, que pode melhorar enormemente o desempenho de aplicações de banda limitada; aritmética de precisão dupla de ponto flutuante; e um modelo de memória de carregamento/armazenamento arbitrário, que permite programarem-se muitos algoritmos novos que anteriormente era difíceis ou impossíveis de implementar na GPU.

4. Desenvolvimento

4.1. Tamanho do block e grid

Para determinar o tamanho de cada bloco foi utilizado o "CUDA: Occupancy Calculator", uma tabela de Excel disponível no CUDA Toolkit que ajuda desenvolvedores a saber se o tamanho dos blocos sendo utilizados pelos *kernels* está em um valor ótimo. Isso acontece quando o número de *warps* ativos atinge um valor máximo, aproximando-se do número de *warps* disponíveis.

Para usar essa tabela, quatro informações são necessárias: a computabilidade da GPU, o número de *threads* em cada bloco, o número de registradores por *thread* e a memória compartilhada usada por bloco. A computabilidade da GPU está disponível tanto no site oficial da NVIDIA quanto usando-se o *script queryDevice* do CUDA Toolkit. O número de *threads* em cada bloco, por sua vez, é definido pelo desenvolvedor ao se chamar um *kernel*. Os últimos dois atributos são obtidos utilizando a *flag –ptxas-options=-v* ao compilar-se o programa.

Assim, o tamanho ótimo de cada bloco não depende só de qual GPU está sendo utilizada, mas também da implementação do programa. Nas seções a seguir, serão discutidas as implementações de cada um dos algoritmos, incluindo a definição dos tamanhos dos blocos e *grids*.

4.2. Implementação do ROT13

Com a descrição do algoritmo ROT13 da seção anterior pode-se perceber que este algoritmo é facilmente paralelizado, uma vez que cada caractere depende apenas dele mesmo para sua codificação. Assim, cada *kernel* é

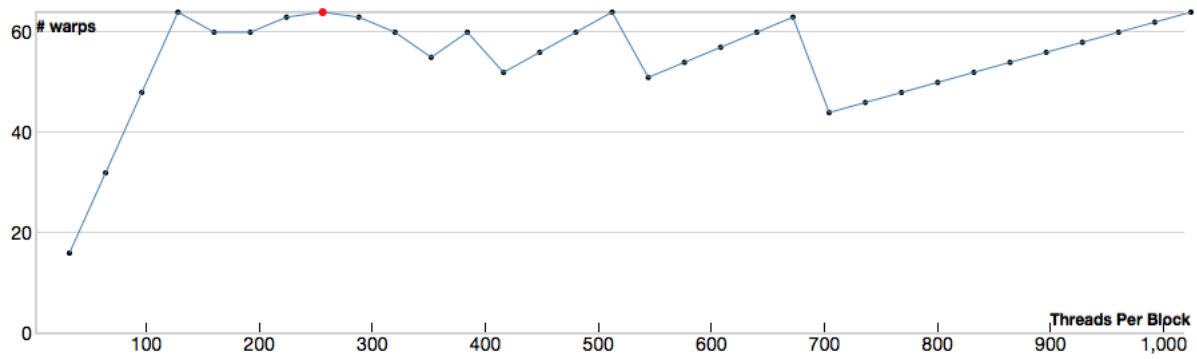


Figura 3: Impacto do número de *threads* por bloco sobre o número de *warps* ativos para o ROT13. O ponto vermelho representa 256 *threads* por bloco.

responsável apenas pela codificação de um caractere e pode, portanto, ser executado em qualquer ordem.

Por se tratar de um problema simples, tanto o *grid* quanto os blocos são unidimensionais. A escolha do tamanho dos blocos foi feita usando o procedimento descrito anteriormente, o que levou a criação de blocos com 256 *threads*. O gráfico na Figura 3 mostra como tal escolha é adequada, apesar de outros tamanhos também serem possíveis, por exemplo, 128, 512 e 1024. Os outros dois gráficos gerados pela Occupancy Calculator, apresentados nas Figuras 4 e 5, também mostram a maximização do número de *warps*. Isso garante que o número de *warps* ativos é máximo, o que melhora o desempenho do algoritmo.

O tamanho do *grid*, por sua vez, foi calculado usando a equação 2, onde g é o tamanho do *grid*, N o número de *kernels* necessários para a entrada, t o número de *threads* por bloco e n o tamanho da entrada:

$$g = \frac{N + t - 1}{t} = \frac{n - 255}{256} \tag{2}$$

Essa equação garante que o *grid* tenha um tamanho suficientemente grande para a entrada e ao mesmo tempo não ultrapasse muito a quantidade de blocos necessária. Neste caso, $N = n$, já que cada *kernel* codificará apenas um caractere da entrada.

4.3. Implementação do Base64

No caso do Base64, cada conjunto de 3 caracteres é codificado independentemente dos outros. Isso significa que, assim como o ROT13, este algoritmo também pode ser facilmente paralelizado. Cada *kernel* é responsável pela

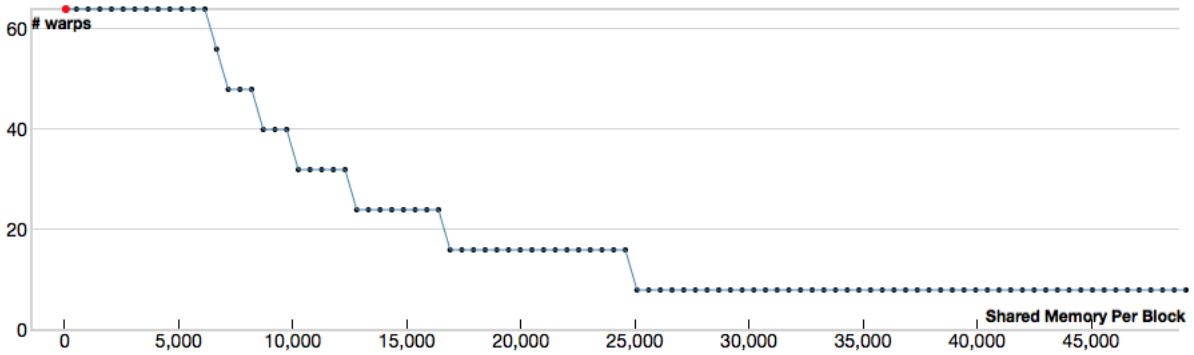


Figura 4: Impacto da quantidade de memória compartilhada por bloco sobre o número de *warps* ativos para o ROT13. O ponto vermelho representa a quantidade gasta pelo algoritmo: 48 *bytes*.

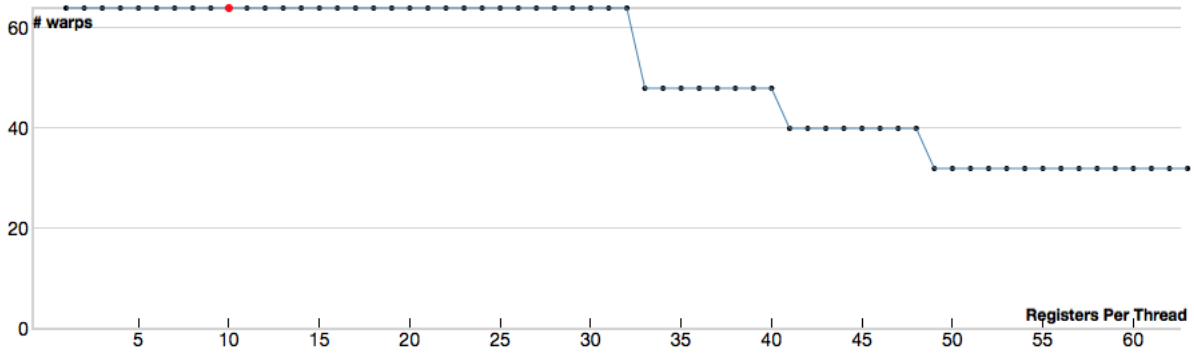


Figura 5: Impacto do número de *threads* por bloco sobre o número de *warps* ativos para o ROT13. O ponto vermelho representa 10 registradores.

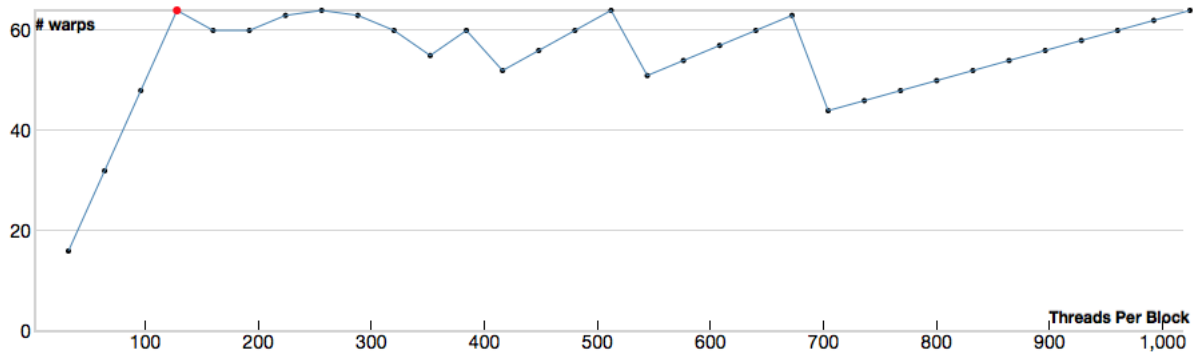


Figura 6: Impacto do número de *threads* por bloco sobre o número de *warps* ativos para o Base64. O ponto vermelho representa 128 *threads* por bloco.

codificação de 3 caracteres seguidos e trata cada um deles conforme necessário. Note que, ao contrário do ROT13, o tamanho da entrada e da saída são diferentes, o que faz com que seja necessário o uso de vetores distintos para codificar/decodificar uma entrada.

Foram escolhidos blocos e *grids* unidimensionais devido à simplicidade do problema. O número de *threads* em cada bloco foi definido como 128, usando o mesmo procedimento usado no ROT13. Pelas figuras 6, 7 e 8, vê-se que o tamanho de bloco escolhido está adequado. Já o tamanho do *grid* foi calculado usando a mesma equação do ROT13, mas desta vez $N = \frac{n}{3}$, já que cada *kernel* é responsável por 3 caracteres. Logo, aplicando a equação 3, o tamanho do *grid* é $\frac{\frac{n}{3}-127}{128}$, onde n é o tamanho da entrada.

$$g = \frac{N + t - 1}{t} = \frac{\frac{n}{3} - 127}{128} \tag{3}$$

4.4. Implementação do Blowfish

Escrever aqui sobre a implementação do algoritmo Blowfish.

5. Experimentos

5.1. Hardware utilizado

Os experimentos foram executados em uma GPU *NVIDIA GeForce GTX 680MX* com 2GB de memória VRAM, 1536 CUDA cores e capacidade CUDA 3.0. Além disso, foi utilizada a versão 8.0 do CUDA.

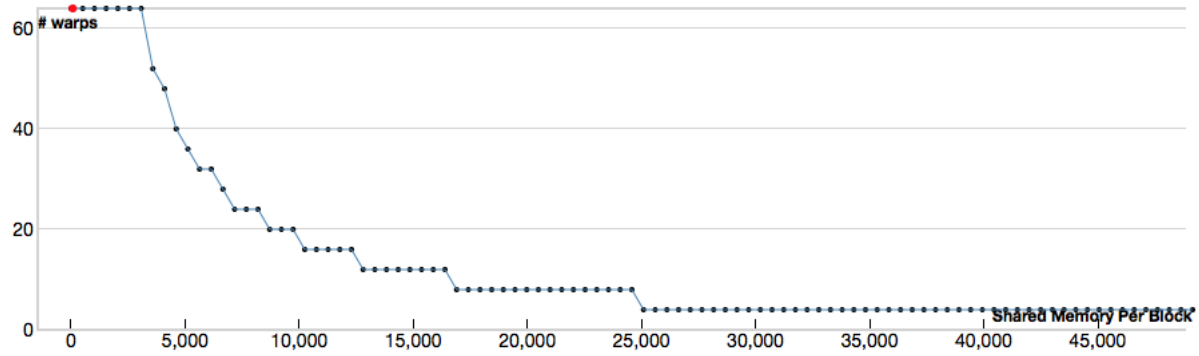


Figura 7: Impacto da quantidade de memória compartilhada por bloco sobre o número de *warps* ativos para o Base64. O ponto vermelho representa a quantidade gasta pelo algoritmo: 48 *bytes*.

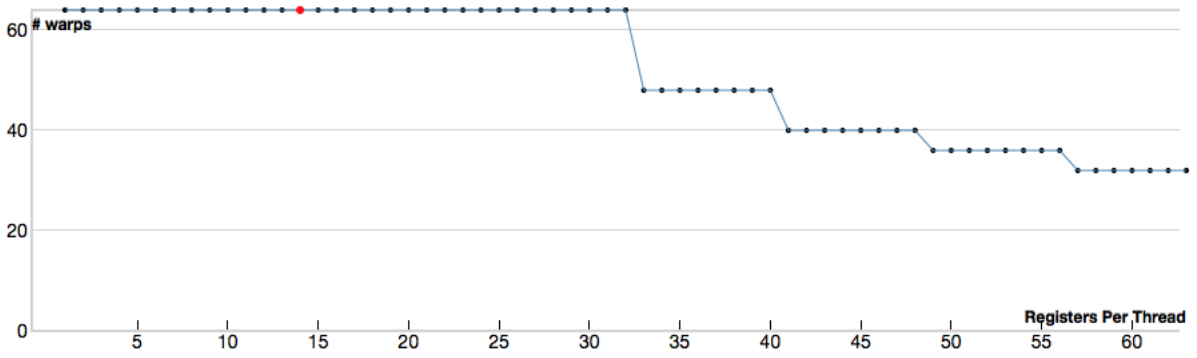


Figura 8: Impacto do número de *threads* por bloco sobre o número de *warps* ativos para o Base64. O ponto vermelho representa 14 registradores.

5.2. Script

Os experimentos foram rodados usando um *script* escrito em Python 3. Cada algoritmo de encriptação é rodado com cada um dos arquivos de teste dez vezes, incluindo as versões sequenciais dos algoritmos. Para medir o desempenho das versões paralelas dos algoritmos, foi usada a ferramenta **nvprof**, que fornece uma série de informações detalhadas sobre a execução do programa. Os resultados são colocados em uma estrutura de dados que facilita seu uso na etapa de criação de gráficos, que é feita usando a biblioteca **matplotlib**. Para instalá-la, basta executar:

```
pip3 install matplotlib
```

O *script* gera gráficos comparando o tempo de execução de um mesmo algoritmo para diferentes arquivos e gráficos comparando o tempo de execução das versões sequenciais *vs* versões usando CUDA. Os gráficos gerados são salvos na pasta '../graphs/'. Para rodar o *script*, basta executar:

```
python3 experiments.py
```

```
from subprocess import getoutput
from textwrap import fill
import subprocess
import re
import pprint as pp
import numpy
import time
import matplotlib.pyplot as plt
import matplotlib.patches as mpatches
import os

def unit_to_expo_notation(x):
    return {
        'm': "E-3",
        'u': "E-6",
        'n': "E-9"
    }.get(x, x)

def seconds_to_float(s):
    if not s.endswith("s"): return None
    s = s.replace("s", "")
    return float(s.replace(s[-1], unit_to_expo_notation(s[-1])))

def get_total_time(times):
    total = 0
    for t in times.keys():
        total += times[t].time
    return total

def get_total_IO_time(times):
    total = 0
    for t in times.keys():
        if (times[t].name == 'cudaMalloc' or times[t].name == 'cudaFree'
            or times[t].name == 'cudaMemcpy'):
            total += times[t].time
    return total
```

```
class Line_Result:

    def __init__(self, s):
        l = line.split()
        if len(l) < 7: print("SOMETHING'S WRONG")
        self.time_percent = float(l[0].strip("%"))
        self.time = seconds_to_float(l[1])
        self.avg = seconds_to_float(l[3])
        self.min = seconds_to_float(l[4])
        self.max = seconds_to_float(l[5])
        self.name = " ".join(l[6:])
        self.count = 1

    def __str__(self):
        return " ".join([self.name, str(self.time_percent) + "%",
            str(self.time), str(self.avg), str(self.min), str(self.max)])

    def __repr__(self):
        return " ".join([self.name, str(self.time_percent) + "%",
            str(self.time), str(self.avg), str(self.min), str(self.max)])

p = re.compile('^[ ]*\d+\.?\d+?')
algorithms = ['rot_13', 'base64', 'arcfour']
values = [True, False]
test_files = ['tale_of_two_cities.txt', 'moby_dick.txt',  'ulysses.txt',
    'king_james_bible.txt',
 'hubble_2.png', 'mercury.png', 'hubble_3.tif', 'hubble_1.tif']

sequential_results = {}
cuda_results = {}
io_results = {}

for a in algorithms:
    cuda_results[a] = {}
    sequential_results[a] = {}
    io_results[a] = {}
    for f in test_files:
        for sequential in values:
            total_times = []
            total_IO_percents = []
            times = {}
            for i in range(0, 10):
                if not sequential:
                    output = getoutput("nvprof ./encode_cuda
                        sample_files/" + f + " " + a).split("\n")
                    for line in output:
                        if (p.match(line)):
                            t = Line_Result(line)
                            times[t.name] = t
                    tt = get_total_time(times)
                    if (tt != 0):
                        total_times.append(tt)
                    io = get_total_IO_time(times) / total_times[-1] * 100
                    if (io != 0):
                        total_IO_percents.append(io)
                else:
                    start = time.time()
                    subprocess.call(["./encode_cuda","sample_files/" + f,
                        a, "-s"])
                    total_times.append(time.time() - start)

            if (sequential):
```

```
                sequential_results[a][f] = total_times
            else:
                cuda_results[a][f] = total_times
                io_results[a][f] = total_IO_percents
            print("\nAlgorithm: {} seq = {}, file: {}".format(a,
                sequential, f))
            print("Average: {} seconds".format(numpy.mean(total_times)))
            print("Standard deviation:
                {}\n".format(numpy.std(total_times)))

# File Impact
for seq in values:
    for a in algorithms:
        if (a == 'rot_13'): continue
        plt.figure()
        data_to_plot = []
        for f in test_files:
            if (seq): data_to_plot.append(sequential_results[a][f])
            else: data_to_plot.append(cuda_results[a][f])
        plt.boxplot(data_to_plot, meanline=True)
        plt.xlabel("Arquivo")
        plt.ylabel("Tempo de execucao (s)")
        plt.xticks(list(range(1, len(test_files) + 1)), test_files,
            rotation=45)
        if not seq:
            plt.title(fill("Tempo de execucao usando " + a + " para
                diferentes arquivos", 45))
        else:
            plt.title("Tempo de execucao usando " + a + "_seq" + " para
                diferentes arquivos")
        name = a
        if (seq): name += "_seq"
        name += ".pdf"
        plt.tight_layout()
        try:
            plt.savefig('./graphs/file_impact/' + name)
        except FileNotFoundError:
            os.makedirs('./graphs/file_impact/')
            plt.savefig('./graphs/file_impact/' + name)

for seq in values:
    a = 'rot_13'
    plt.figure()
    data_to_plot = []
    for f in test_files:
        if ((f == 'hubble_1.tif' or f == 'hubble_2.png' or
            f == 'hubble_3.tif' or f == 'mercury.png') and a == 'rot_13'):
                continue
        if (seq): data_to_plot.append(sequential_results[a][f])
        else: data_to_plot.append(cuda_results[a][f])
    plt.boxplot(data_to_plot, meanline=True)
    plt.xlabel("Arquivo")
    plt.ylabel("Tempo de execucao (s)")
    plt.xticks(list(range(1, len(test_files) - 3)), test_files,
        rotation=45)
    if not seq:
        plt.title(fill("Tempo de execucao usando " + a + " para diferentes
            arquivos", 45))
    else:
        plt.title("Tempo de execucao usando " + a + "_seq" + " para
            diferentes arquivos")
    name = a
```

```
    if (seq): name += "_seq"
    name += ".pdf"
    plt.tight_layout()
    try:
        plt.savefig('./graphs/file_impact/' + name)
    except FileNotFoundError:
        os.makedirs('./graphs/file_impact/')
        plt.savefig('./graphs/file_impact/' + name)

# CUDA impact
for f in test_files:
    for a in algorithms:
        if ((f == 'hubble_1.tif' or f == 'hubble_2.png' or
            f == 'hubble_3.tif' or f == 'mercury.png') and a == 'rot_13'):
                continue
        data_to_plot = []
        for seq in values:
            if (seq): data_to_plot.append(sequential_results[a][f])
            else: data_to_plot.append(cuda_results[a][f])

        plt.figure()
        plt.boxplot(data_to_plot, meanline=True)
        plt.xlabel("Implementacao")
        plt.ylabel("Tempo de execucao (s)")
        plt.xticks([1, 2], ["Sequencial", "CUDA"])
        plt.title(fill("Comparacao do tempo de execucao para encriptar {}
            com a versao usando CUDA e a sequencial de {}".format(f, a),
            45))
        name = f + ".pdf"
        plt.tight_layout()
        try:
            plt.savefig('./graphs/cuda_impact/' + a + "/" + name)
        except FileNotFoundError:
            os.makedirs('./graphs/cuda_impact/' + a)
            plt.savefig('./graphs/cuda_impact/' + a + "/" + name)
        plt.close()

# IO percentage
for a in algorithms:
    plt.figure()
    data_to_plot = []
    for f in test_files:
        data_to_plot.append(io_results[a][f])
    plt.boxplot(data_to_plot, meanline=True)
    plt.xlabel("Arquivo")
    plt.ylabel("Porcentagem do tempo gasto com IO")
    plt.xticks(list(range(1, len(test_files) + 1)), test_files,
        rotation=45)
    plt.title(fill("Porcentagem do tempo gasto com IO usando " + a + "
        para diferentes arquivos", 45))
    name = a
    name += ".pdf"
    plt.tight_layout()
    try:
        plt.savefig('./graphs/io_percentage/' + name)
    except FileNotFoundError:
        os.makedirs('./graphs/io_percentage/')
        plt.savefig('./graphs/io_percentage/' + name)
```

6. Resultados

6.1. Impacto do I/O e alocação de memória

Os resultados demonstram claramente que o tempo gasto com as operações de entrada e saída e de alocação de memória é bastante significativo. De fato, nos melhores casos, esse tipo de operação gastou 75% do tempo de execução total dos algoritmos, chegando a gastar mais de 97% nos piores casos.

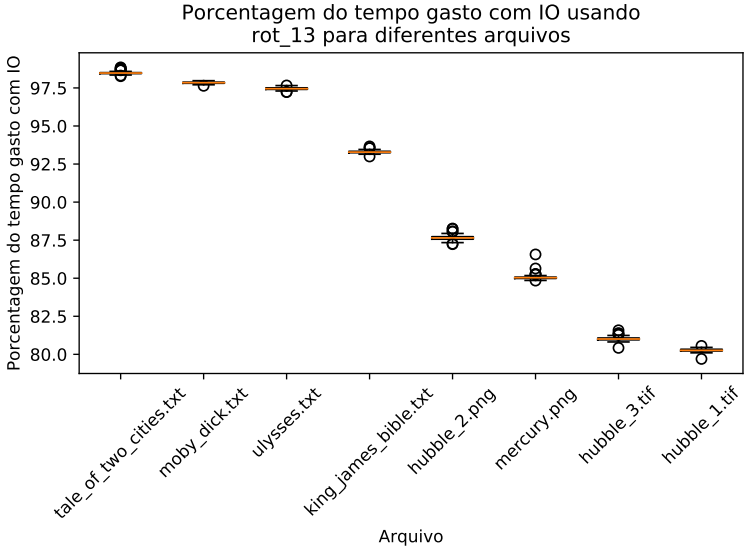


Figura 9: Gráfico da porcentagem de tempo gasta com operações de I/O e alocação de memória usando o algoritmo ROT13. Os arquivos estão em ordem crescente de tamanho.

Além disso, pode se perceber uma forte relação entre a porcentagem de tempo gasta com alocação de memória e o tamanho do arquivo a ser encriptado. Os gráficos 9, 10 e 11 mostram que independente do algoritmo utilizado, quanto maior for o arquivo, menor será a porcentagem de tempo gasta com alocação de memória. Isso significa que os programas gastam uma fatia maior de seu tempo realmente aplicando o algoritmo quando o arquivo é grande e acabam tendo um *overhead* menor de alocação de memória.

6.2. Versão paralelizada do ROT13

O algoritmo sequencial de ROT13 apresentou um desempenho superior à versão usando CUDA para todos os arquivos. As figuras 12 e 13 representam os casos extremos observados e mostram como o comportamento foi similar para todos os arquivos testados. Note que este algoritmo só foi rodado usando

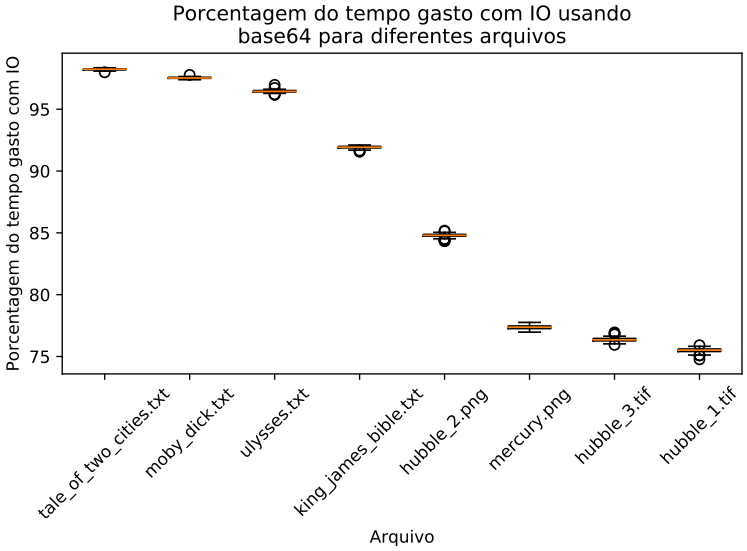


Figura 10: Gráfico da porcentagem de tempo gasta com operações de I/O e alocação de memória usando o algoritmo Base64. Os arquivos estão em ordem crescente de tamanho.

arquivos de texto, uma vez que ele é projetado apenas para esse tipo de arquivo.
Essa diferença na performance entre a versão CUDA e sequencial pode ser explicada pelo fato de haver um grande *overhead* de alocação de memória e transferência de dados para a GPU, como visto na seção anterior. Assim, a versão paralelizada gasta muito tempo com os preparativos do algoritmo e os arquivos de texto não são grandes o bastante para compensar isso, o que dá vantagem ao sequencial.

6.3. Versão paralelizada do Base64

Para arquivos de texto, o Base64 paralelo apresentou um desempenho inferior à versão sequencial, como mostra a figura 14. No entanto, no caso das imagens, a versão sequencial se mostrou mais lenta do que a versão em CUDA. Como pode se ver na figura 15, a diferença entre o tempo de execução da versão sequencial e paralela aumenta conforme o tamanho da imagem aumenta, dando uma vantagem cada vez maior para a versão em CUDA.
Essa diferença pode ser explicada devido ao *overhead* mencionado na seção anterior. No caso do Base64, esse overhead é ainda um pouco maior do que na versão sequencial, uma vez que um vetor de saída deve ser alocado pois, ao contrário dos outros dois algoritmos, o vetor de saída tem um tamanho diferente do de entrada. Apesar disso, o overhead é compensado pelo tempo economizado com a paralelização da criptografia de um arquivo suficiente-

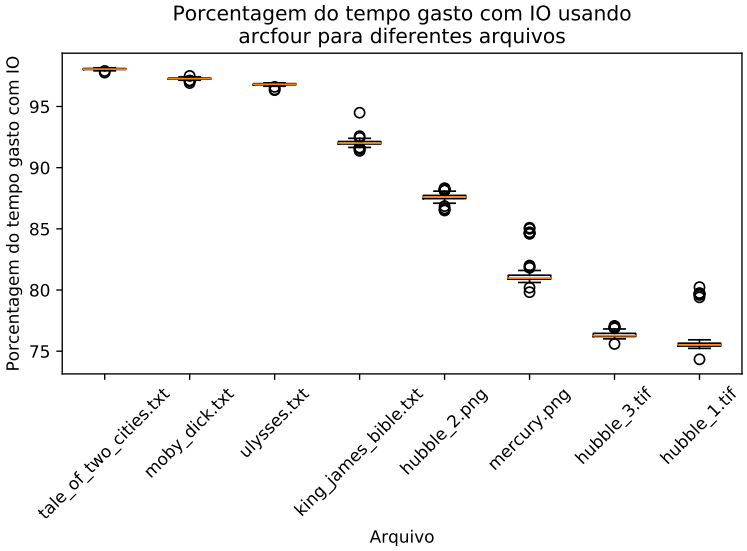


Figura 11: Gráfico da porcentagem de tempo gasta com operações de I/O e alocação de memória usando o algoritmo Arcfour. Os arquivos estão em ordem crescente de tamanho.

mente grande e o algoritmo em CUDA se mostra superior ao sequencial uma vez atingido esse tamanho mínimo.

6.4. Versão paralelizada do Arcfour

Tal como o Base64, o Arcfour paralelo se mostrou mais rápido do que sua versão sequencial apenas ao codificar arquivos menores. No entanto, a figura 16 mostra que o algoritmo em CUDA já supera o algoritmo sequencial no caso do arquivo King James Bible, ao contrário do Base64. Isso se deve ao fato de que o Base64 tem um overhead maior de alocação de memória, conforme mencionado anteriormente.
Ao criptografar imagens, por sua vez, o algoritmo em CUDA foi sempre mais rápido, como mostra a figura 17. A razão para isso é a mesma dos casos anteriores: o *overhead* de alocação de memória é compensado pela rápida codificação trazida pelo paralelismo do programa.

6.5. Tamanho do arquivo

Conforme esperado, arquivos maiores levam mais tempo para serem criptografados usando tanto a versão sequencial quanto a versão em CUDA de qualquer um dos algoritmos. No entanto, os gráficos 19 e 18 revelam uma diferença interessante entre a versão sequencial e a versão paralelizada. Enquanto que na versão sequencial a diferença de tempo de execução entre os arquivos varia de uma forma significativa conforme o tamanho deles, no caso

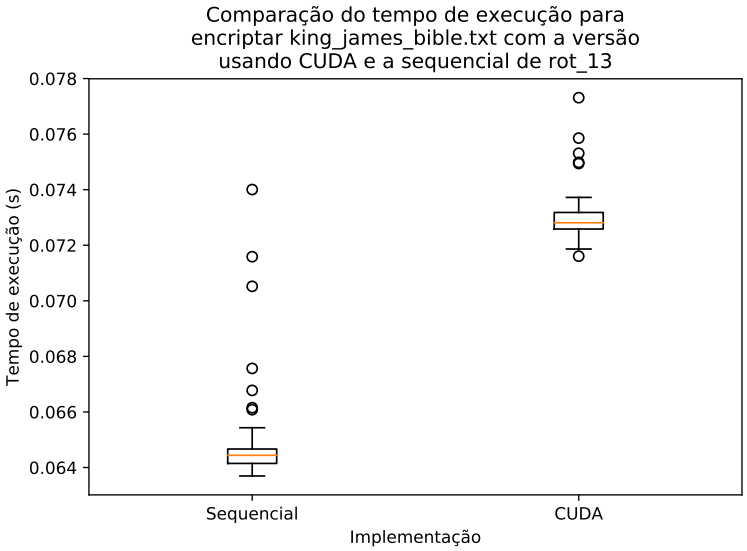


Figura 12: Gráfico do tempo de execução do algoritmo ROT13 paralelizado usando CUDA e usando a versão sequencial para o arquivo king_james_bible.txt

paralelizado isso acontece de uma forma bem menos expressiva. Isso fica bastante claro ao comparar o salto entre o tempo de execução de *ulysses.txt* e *king_james_bible.txt*, cujos tamanhos são, respectivamente, 1.6MB e 4.5MB, nas figuras 19 e 18. No caso sequencial, o algoritmo demorou quase o triplo do tempo para encriptar *king_james_bible.txt* do que para encriptar *ulysses.txt*. Já no caso, paralelizado, o tempo de execução para o *king_james_bible.txt* é apenas em torno de 10% maior do que para o *ulysses.txt*. Isso significa que o algoritmo paralelizado usando CUDA sofre muito menos do que o algoritmo sequencial com o aumento do tamanho do arquivo. Esse mesmo fenômeno pode ser observado no caso do Base64 e Arcfour, como mostram as figuras 20, 21, 22 e 23.
Isso pode ser facilmente explicado devido a paralelização dos algoritmos. A versão sequencial do algoritmo deve percorrer todo o arquivo de forma sequencial, o que leva um tempo muito maior do que a paralelização desse processo, em que várias partes do arquivo vão sendo criptografadas ao mesmo tempo. Isso faz com que o tamanho do arquivo tenha uma influência consideravelmente menor no tempo de execução do programa paralelo.

7. Conclusões

O uso da GPU para a execução e paralelização de algoritmos pode trazer muitos benefícios para o desempenho de um programa. No entanto,

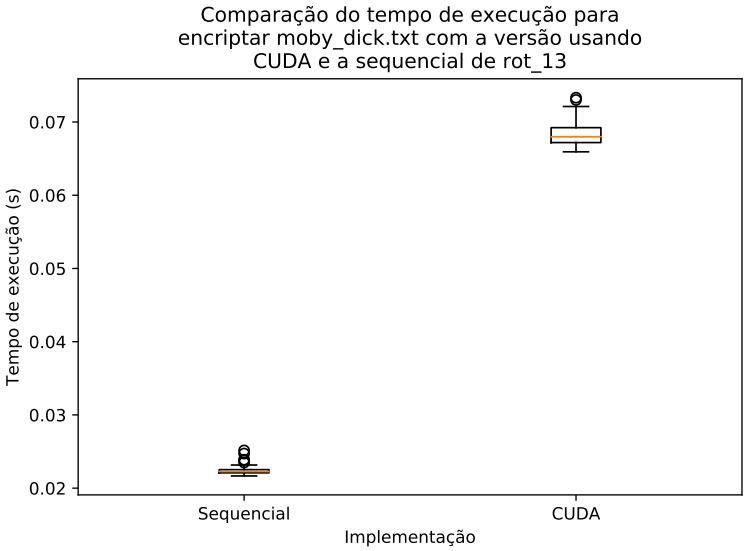
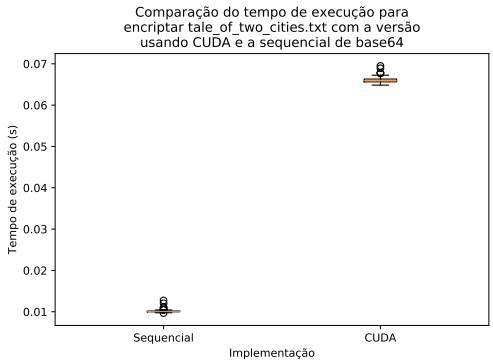
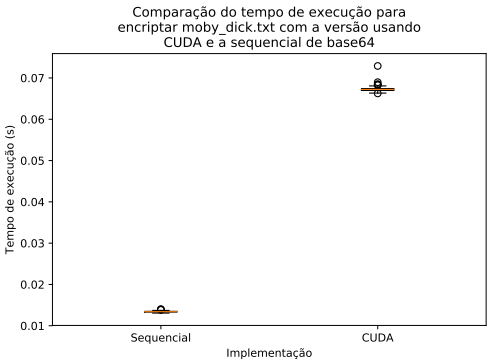


Figura 13: Gráfico do tempo de execução do algoritmo ROT13 paralelizado usando CUDA e usando a versão sequencial para o arquivo moby_dick.txt.

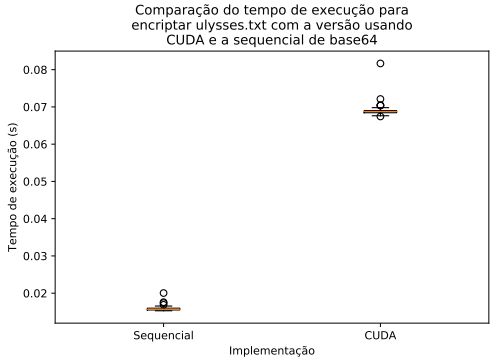
para que isso aconteça, o programa deve ser cuidadosamente desenvolvido, levando em conta as peculiaridades do algoritmo sendo implementado e as características da GPU que será usada. Além disso, deve-se levar em conta o tamanho dos problemas que o algoritmo será usado para resolver. Os resultados dos experimentos mostraram que para entradas pequenas, os algoritmos de criptografia que foram implementados tinham desempenho pior quando eram paralelizados usando CUDA. Assim, antes de implementar um algoritmo usando CUDA, é necessário avaliar se de fato haverá algum benefício em fazê-lo, ainda mais considerando a curva de aprendizado envolvida no processo. Os resultados também mostraram o enorme impacto que as alocações de memória e transferências de dados para a GPU tem sobre o tempo de execução de um programa.



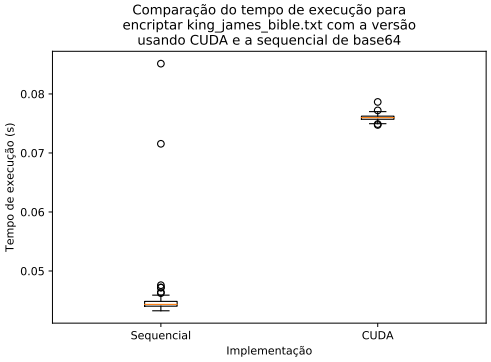
(a) Tale of Two Cities



(b) Moby Dick

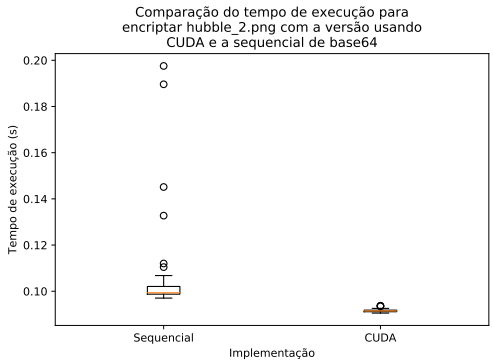


(a) Ulysses

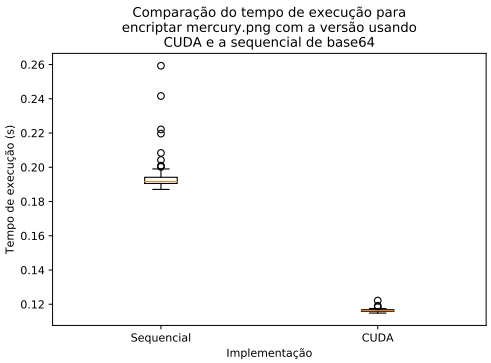


(b) King James Bible

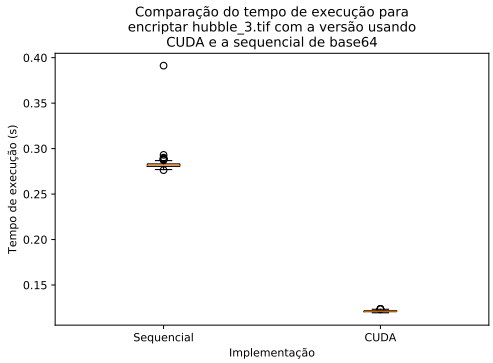
Figura 14: Gráficos comparando o tempo de execução da versão paralela e sequencial do Base64 para arquivos de texto. Os arquivos estão em ordem crescente de tamanho.



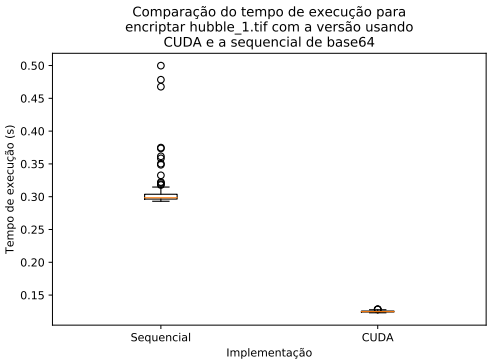
(a) Hubble 2



(b) Mercury

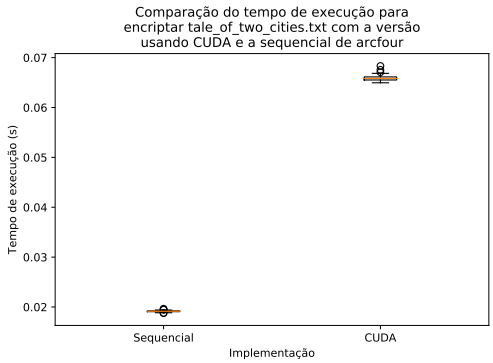


(a) Hubble 3

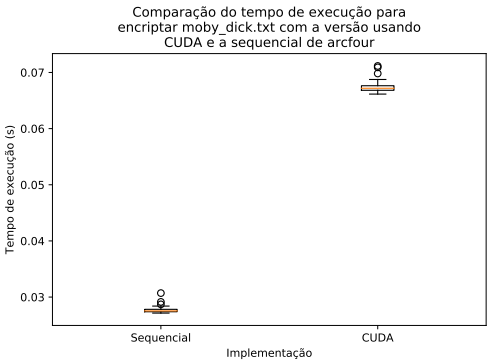


(b) Hubble 1

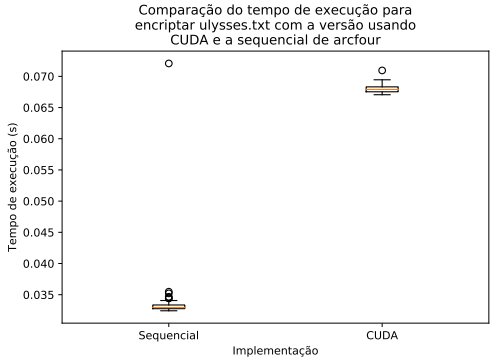
Figura 15: Gráficos comparando o tempo de execução da versão paralela e sequencial do Base64 para arquivos de imagem. Os arquivos estão em ordem crescente de tamanho.



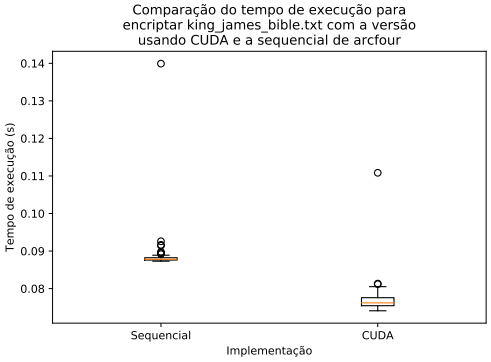
(a) Tale of Two Cities



(b) Moby Dick

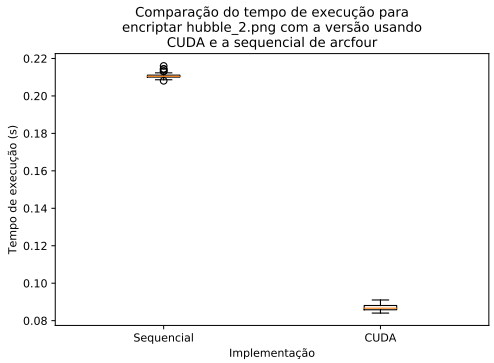


(a) Ulysses

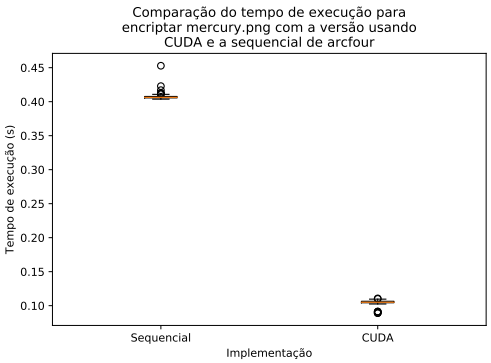


(b) King James Bible

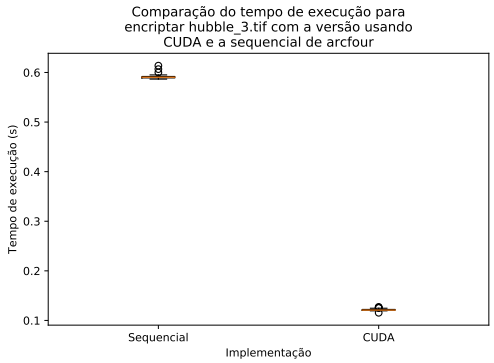
Figura 16: Gráficos comparando o tempo de execução entre a versão paralela e sequencial do Arcfour para arquivos de texto. Os arquivos estão em ordem crescente de tamanho.



(a) Hubble 2



(b) Mercury



(a) Hubble 3

(b) Hubble 1

Figura 17: Gráficos comparando o tempo de execução da versão paralela e sequencial do Arcfour para arquivos de imagem. Os arquivos estão em ordem crescente de tamanho.

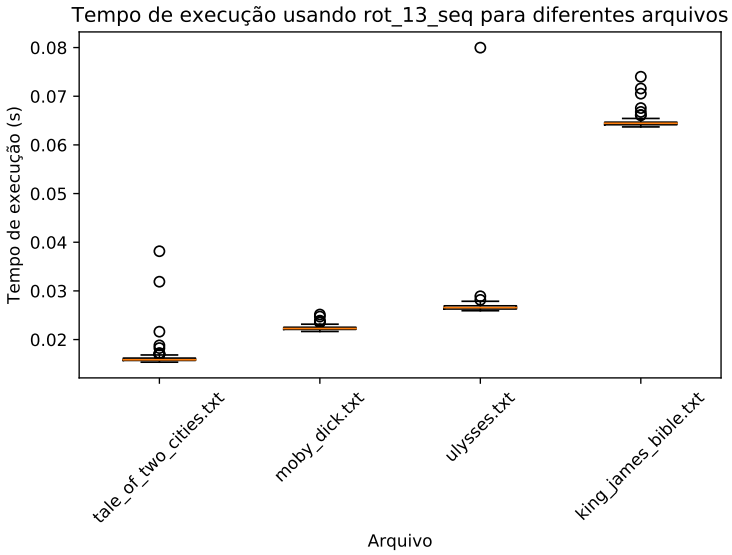


Figura 18: Gráficos do tempo de execução do algoritmo ROT13 sequencial para diferentes arquivos. Os arquivos estão em ordem crescente de tamanho.

Referências

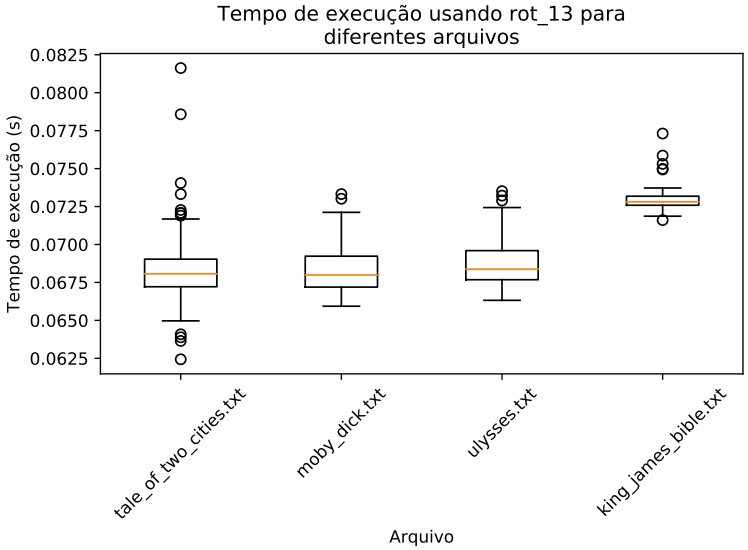


Figura 19: Gráficos do tempo de execução do algoritmo ROT13 paralelizado usando CUDA para diferentes arquivos. Os arquivos estão em ordem crescente de tamanho.

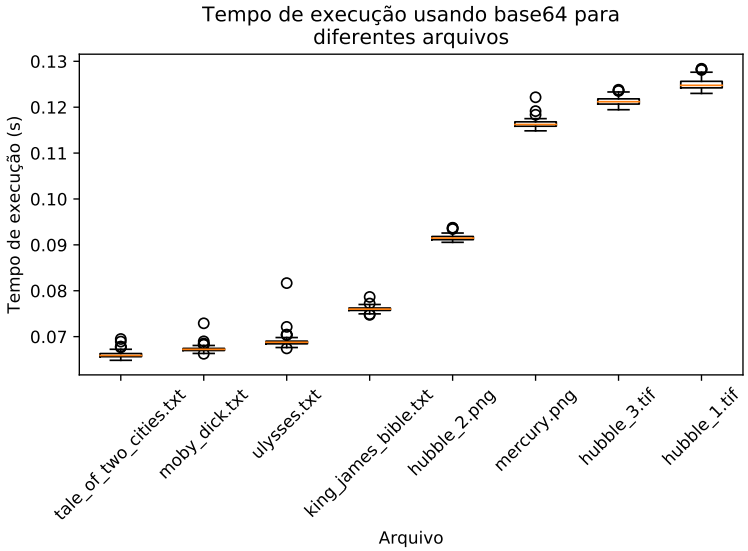


Figura 20: Gráficos do tempo de execução do algoritmo Base64 paralelizado usando CUDA para diferentes arquivos. Os arquivos estão em ordem crescente de tamanho.

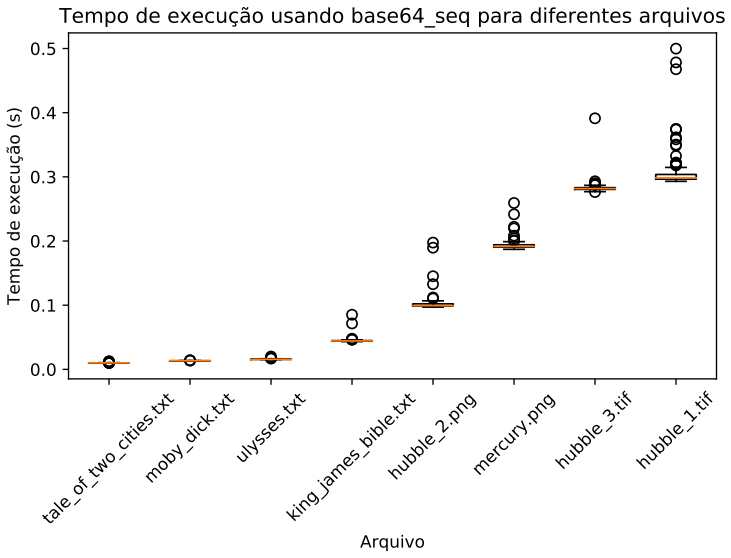


Figura 21: Gráficos do tempo de execução do algoritmo Base64 sequencial para diferentes arquivos. Os arquivos estão em ordem crescente de tamanho.

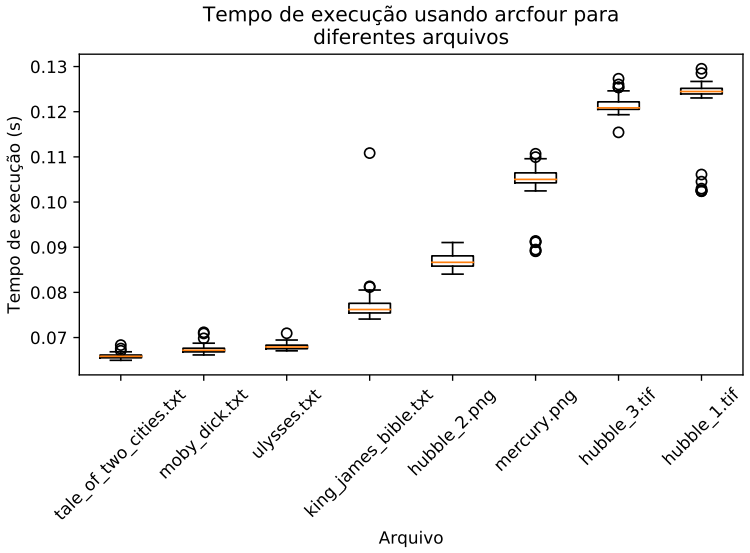


Figura 22: Gráficos do tempo de execução do algoritmo Arcfour paralelizado usando CUDA para diferentes arquivos. Os arquivos estão em ordem crescente de tamanho.

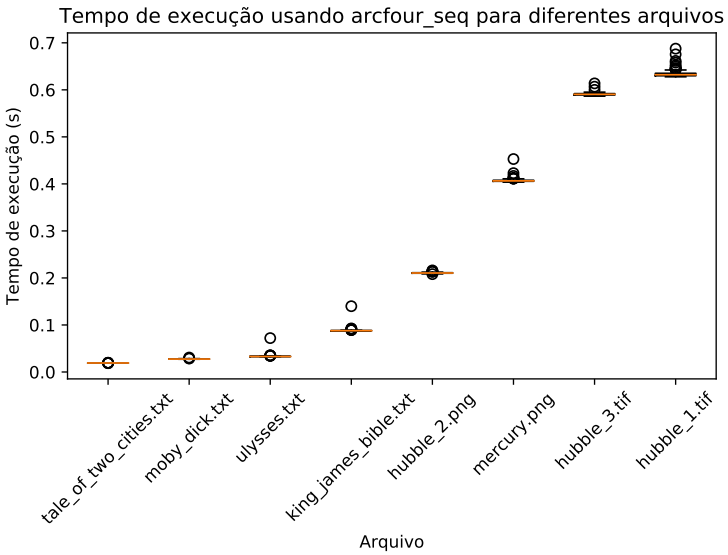


Figura 23: Gráficos do tempo de execução do algoritmo Arcfour sequencial para diferentes arquivos. Os arquivos estão em ordem crescente de tamanho.