Instituto de Matemática e Estatística

EP1 - MAC0219

Implementação de Algoritmos de Criptografia e Codificação Paralelos Usando CUDA

Professor: Alfredo Goldman

Alunos: Bruno Sesso

Gustavo Estrela de Matos

São Paulo, 4 de Junho de 2017

Conteúdo

1 Introdução

Este trabalho tem como objetivo a paralelização e análise de desempenho de algoritmos de criptografia e codificação para serem rodados em GPUs. O código desenvolvido utilizará a biblioteca *Compute Unified Device Architecture* (CUDA), e portanto será compatível apenas com GPUs NVIDIA.

Pensando na arquitetura *multiple instruction single data* (MISD), escolhemos três algoritmos em que os dados não possuem grandes dependência entre si, e portanto podem ser separados mais facilmente para serem processados em paralelo. Os três algoritmos escolhidos foram:

- **Base64**: um algoritmo de codificação;
- **Rot13**: também um algoritmo de codificação;
- **Vigenere**: um algoritmo de cifração.

Para analisar a paralelização dos algoritmos a nossa principal métrica foi o tempo de execução ao processar arquivos de texto. Os três algoritmos escolhidos não tem grandes dependências ao conteúdo lido, portanto escolhemos arbitrariamente uma versão em texto puro da Bíblia para medir tempos de execução. Para garantir significância estatística utilizamos o programa *perf*, que nos permite apresentar resultados médios de rodadas do algoritmo.

O computador utilizado para realizar os testes foi o *terra* (terra.eclipse.ime.usp.br), que conta com uma GPU GeForce GTX 750, rodando CUDA 8.0, e 4 Multiprocessors de 128 CUDA cores cada.

2 Algoritmos Escolhidos

2.1 Base64

2.2 Rot13

Rot13 é um algoritmo bastante simples em que para cada letra é trocada por uma 13 indices a frente. Por exemplo: a letra "a"é substituida pela letra "n", a letra "b"por "o"e assim por diante. Para implementar uma versão que possa ser rodada em uma GPU separamos a simples operação de somar 13 para cada caractere e a passamos como kernel.

2.3 Vigenere

3 Os experimentos

Para cada um dos algoritmos analisados realizamos uma série de testes. Como queríamos analisar o desempenho dos algoritmos em relação ao tamanho de entrada utilizamos o texto da Bíblia em diversos tamanhos. O texto da Bíblia tem 4452070 caracteres. A partir desse texto, criamos arquivos com o mesmo texto repetido 1, 2, 4 e 8 vezes. Isto é, nossos arquivos de testes são o textos de tamanho exponencial.

Nosso primeiro teste verifica o tempo necessário para um alogritmo de encriptação implementado sequencialmente processar os arquivos de testes (as Bíblias).

Para o algoritmo implementado para rodar na GPU, verificamos não somente o tempo em relação ao tamanho dos arquivos de entrada como também em relação ao número de threads por bloco. Como esse número deve ser múltiplo de 32, verificamos para todos os múltiplos de 32 a 1024 (máximo de threads por bloco).

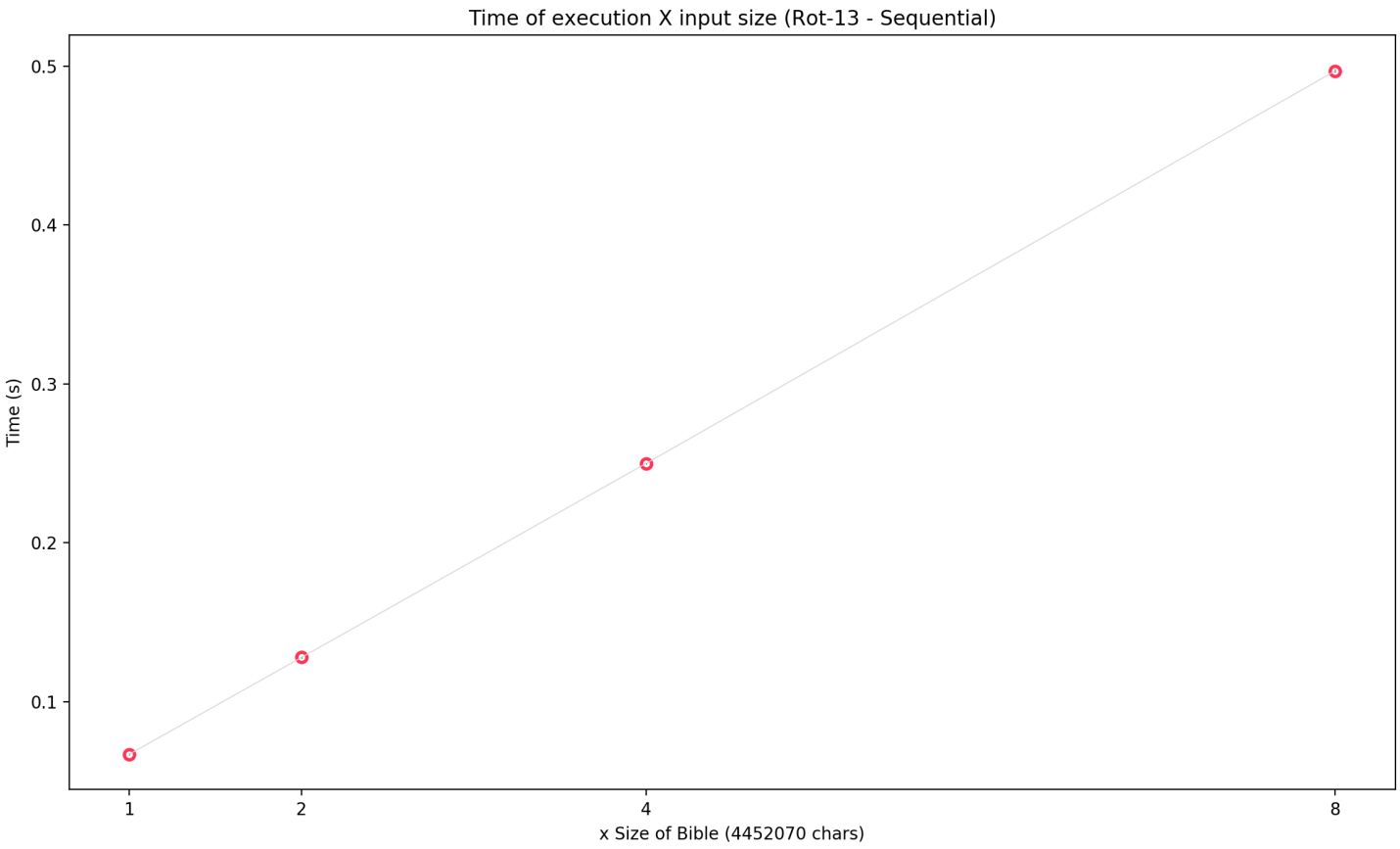
Por fim verificaremos também o tempo levado para se fazer a leitura dos arquivos. A leitura é feita igualmente para ambas as implementçaões de cada algoritmo de encriptação, portanto queremos verificar se o tempo levado para a execução de cada é relacionado às operações de leitura.

4 Discussão dos Resultados

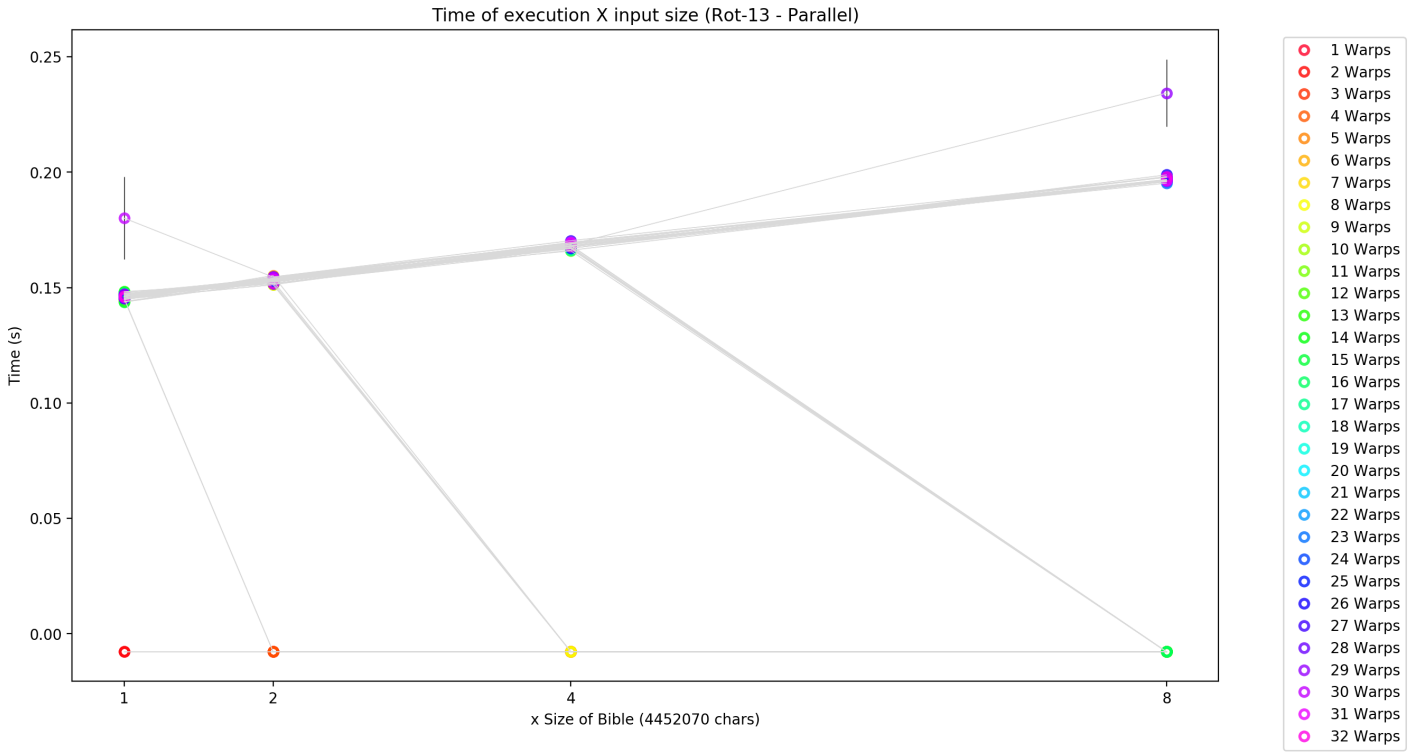
4.1 Base64

4.2 Rot13

Primeiramente analisemos o tempo levado da implementação sequencial. Sabemos que o programa realiza uma operação igual para cada caractere. Portanto esperamos que o tempo de um algoritmo sequencial seja linear para o tamanho do array de caracteres. Observamos tal comportamento no seguinte gráfico:

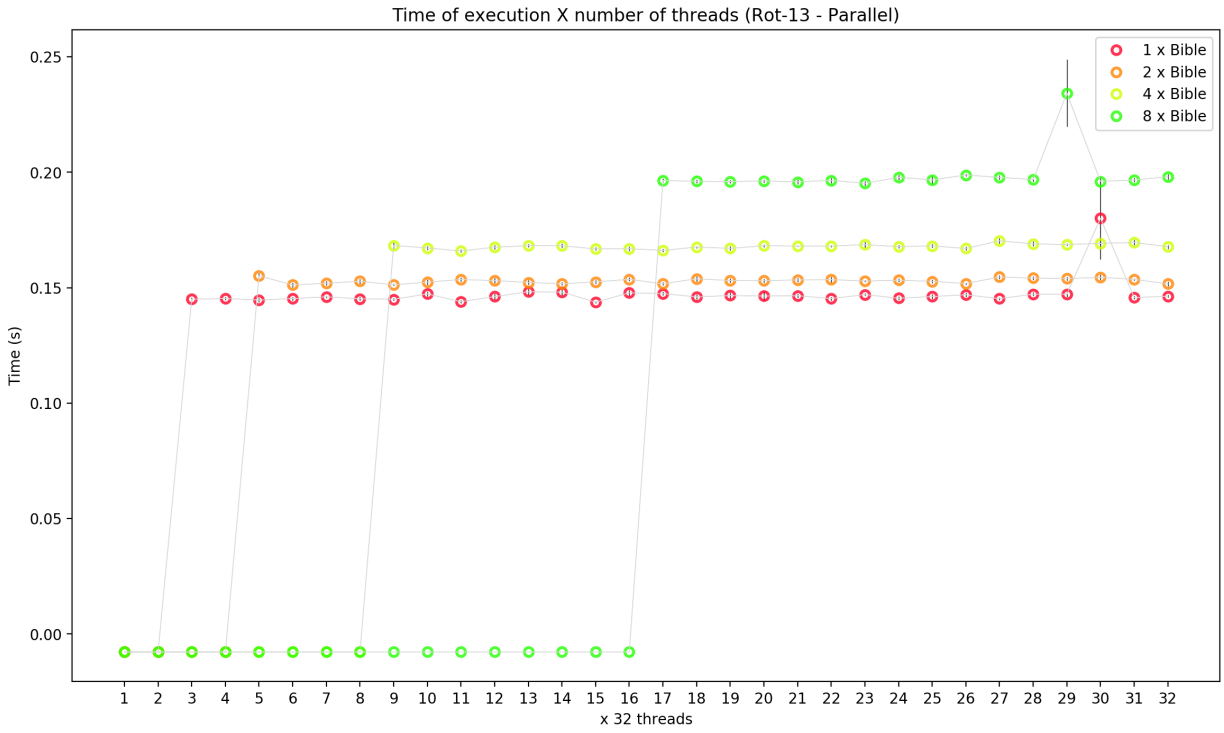


Apesar de 8 vezes o tamanho da biblia (35.616.560 caracteres) ser um número bastante grande, até mesmo o algoritmo sequencial o realiza em apenas meio segundo. Esperamos que a implementação paralela, realize os calculos pelo menos mais rápido que a implementação sequencial:



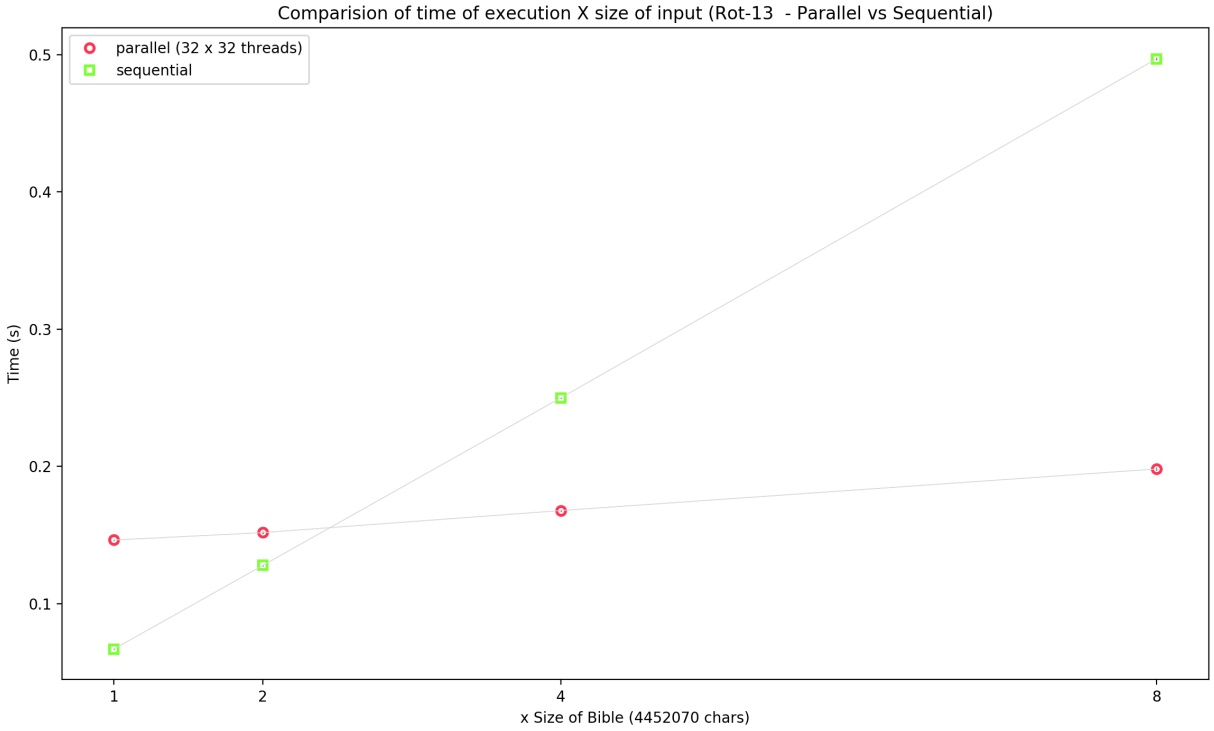
Como esse gráfico possui mais informações analizemos-o com mais cuidado. O eixo Y representa o tempo levado para o algoritmo ser executado. O eixo X representa o tamanho dos arquivos de entrada. Em cada execução do programa, utilizamos um valor de threads por bloco. 1 warp representa 32 threads. Para cada valor de 1 a 32 warps, uma cor é utilizada no gráfico. Além disso para algumas combinações de número de threads e de tamanho de entrada, há falha ao passar tais argumentos para a GPU. Nesses casos representamos o tempo do algoritmo por um valor um pouco abaixo de 0 no gráfico. Comecemos a analizar o gráfico.

Para cores roxas e mais azuladas, notamos que elas seguem o mesmo padrão linear de aproximadamente 0,15s a 0,18s. Os valores aonde há erro ocorrem, pois o número de blocos necessários para a combinação ultrapassaria o número máximo de blocos da GPU. Por exemplo para 1 warp (32 threads) e tamanho de entrada 1 Bíblia (4.452.070 caracteres) há a necessidade de 139.128 blocos, que ultrapassa o número máximo de blocos da GPU (65535). Ou seja, quando aumentamos o tamanho do arquivo de entrada, algum dos valores de número de warps por bloco passam a ser inválidos, pois iriam requerir blocos demais. É interessante notar que para os valores adequados de warps por blocos o tempo de execução do programa é o mesmo:



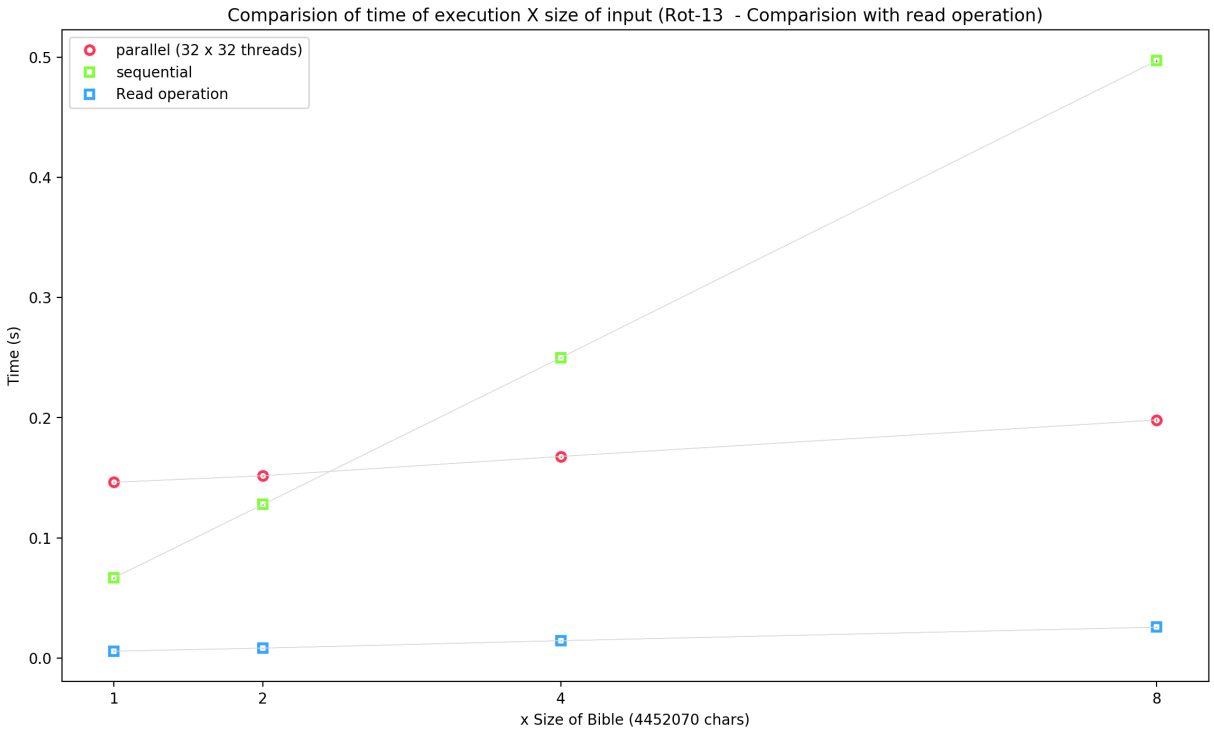
Nesse gráfico o eixo Y representa o tempo de execução do programa enquanto o eixo X representa o número de threads por bloco. Para cada tamanho de entrada uma cor é dada. Nesse gráfico fica claro como para o mesmo tamanho de entrada, independente do número de threads por bloco o tempo para a execução é o mesmo. No entanto o tempo cresce de acordo com o tamanho da entrada. Esse é um resultado inesperado, pois como os valores são processados em paralelo não esperavamos que o tempo fosse aumentar.

Faremos agora uma comparação entre a implementação sequencial e a paralela:



Como discutido anteriormente, como o tempo de execução indepente do número de threads por bloco, escolhemos 32 x 32 threads, pois garantimos que o número de blocos necessários serão validos.

Notamos um comportamento interessante, onde para 1 e 2 vezes a Bíblia, o algoritmo sequencial é mais rápido que o paralelo. No entanto, para valores um pouco maior que 2 vezes o tamanho da Bíblia a implementação paralela é mais rápida. Talvez isso ocorra, devido á transfrencia de dados para a GPU, que leva um tempo extra em relação à implementação sequencial, porém como a os dados são processados paralelamente, a taxa de crescimento do tempo em relação ao tamanho dos arquivos de entrada da implementação paralela é menor do que o da sequencial. De fato, talvez a implementação paralela só cresça devido a operação de leitura de arquivo que ocorre sequencialmente para ambos os algoritmos. Comparemos com o seguinte gráfico:



Para gerar os dados dos pontos azuis, executamos um programa que somente lê um arquivo e o guarda em um vetor de caracteres. Essa é a mesma função utilizada nas duas implementações do algoritmo rot13.

Nota-se que a inclinação do tempo de leitura é bastante similar com a da implementação paralela. Portanto se pudessemos retirar a leitura do algoritmo paralelo, o tempo para encriptação seria o mesmo independente do tamanho de entrada. É claro que isso se limita aos valores máximos de threads por bloco e de blocos por GPU.

4.3 Vigenere

5 Conclusão