Relatório final CI853

Aluno: Renan de Souza Polisciuc
Título: Implementação do algoritmo de criptografia AES-128 utilizando a GPU

Introdução

O algoritmo AES (Advanced Encryption Standard) tem como objetivo criptografar/descriptografar dados utilizando uma chave de forma que somente quem possui-la possa acessar os dados protegidos. O AES possui opções de chaves de tamanhos 128 *bits* (16 *bytes*), 192 *bits* (24 *bytes*) e 256 *bits* (32 *bytes*), que garantem confiabilidade proporcional ao tamanho da chave.

O AES realiza um conjunto de operações(permutações, substituições, etc.) sobre os **n** bytes da entrada, que são divididos e processados em blocos de tamanho **16**. Se **n** não for múltiplo de 16, são adicionados **x** bytes de maneira que **(n + x)** seja múltiplo, o que garante que o algoritmo sempre processe grupos de 16 bytes.

Cada bloco do AES é independente de todos os outros blocos, permitindo que eles sejam processados em **paralelo**. Sendo assim, será proposto neste trabalho uma implementação do algoritmo AES utilizando **CUDA** e os conhecimentos de processamento paralelo obtidos na disciplina.

A ferramenta implementada neste trabalho fará apenas a criptografia dos dados utilizando uma chave de 128 bits. Como o objetivo do trabalho é verificar o ganho de desempenho e não o nível de segurança, o algoritmo de 128 bits já é suficiente (as versões com 192 e 256 bits seriam apenas mais lentas).

O algoritmo de descriptografia não será implementado, pois seu objetivo é apenas inverter as operações realizadas na criptografia. Porém, para garantir a corretude da ferramenta, foram utilizadas algumas ferramentas online que conferem se a criptografia está correta[1].

AES-128 - GPU

O algoritmo AES é composto basicamente por 4 funções principais que são aplicadas sobre um vetor de bytes de tamanho 16 que será chamado “estado”. As funções são especificadas abaixo na sequência em que são executadas:

- *AddRoundKey* (estado[16], chave[16]) : xor byte a byte do estado com a chave
- *SubBytes*(estado[16]): substitui cada byte do estado por um valor equivalente da S-BOX[2] (uma “tabela mágica” com 256 inteiros)
- *ShiftRows(*estado[16]*):* Rotaciona os bytes do estado
- *MixColumns(*estado[16]*):* Galois fields

A saída de cada função é entrada para a outra na sequência, sendo que esse processo é executado 10 vezes para cada bloco de 16 bytes. Para cada rodada do AES, uma chave de 16 bytes diferente é usada. Para que isso seja possível, o AES realiza uma expansão da chave inicial de 16 bytes em um vetor de 176 bytes(10 * 16 + a chave original, que é utilizada antes da primeira rodada). A expansão de chaves é executada apenas uma vez no início do algoritmo.

Implementação

O programa implementado lê bytes da entrada em blocos de tamanho no máximo 512 MB, que chamaremos de *buffSize.* Essa estratégia foi adotada para tentar diminuir a probabilidade de que arquivos muito grandes ocupem toda a memória RAM ou toda a memória da GPU. Esse tamanho pode ser expandido utilizando *padding*, caso o tamanho do arquivo/bloco lido seja menor que 512 MB e não seja múltiplo de 16.

Cada thread utilizada no kernel processa um bloco de 16 bytes (estado) do arquivo. Sendo assim, um arquivo de 512MB precisa de 33554432 threads para ser completamente processado. Em uma GPU com máximo de 1024 threads por bloco, é necessário 32768 blocos de 1024 threads para processar todo o arquivo.

```
//Número de rodadas do AES 128 bits
#define R_ROUNDS 10

//Implementação do Kernel do AES
__global__
void aes(unsigned char * in_bytes, unsigned char * keys, int nBlocks) {
  int id = blockIdx.x * blockDim.x + threadIdx.x;

  if (id <= nBlocks) {

    unsigned char state[16];
    //Copia os primeiros 16 bytes para a memoria
    for(int i = 0; i < 16; i++)
      state[i] = in_bytes[id * 16 + i];

    //Adiciona a primeira chave
    addRoundKey(state, keys);

    //N-1 rodadas
    for(int i = 0; i < (R_ROUNDS -1); i++) {
      subBytes(state);
      shiftRows(state);
      mixColumns(state);

      //Seleciona a próxima chave
      addRoundKey(state, keys + (16 * (i + 1)));
    }

    //Última rodada
    subBytes(state);
    shiftRows(state);
    addRoundKey(state, keys + 160);

    //Copia a resposta para a memória
    for(int i = 0; i < 16; i++)
      in_bytes[id * 16 + i] = state[i];
  }
}
```

O trecho de código acima é a implementação do kernel do AES deste trabalho. No final do kernel, o estado processado pela thread é copiado para a memória global substituindo o estado inicial.

```
...
cudaMemcpy(buffGPU, buffer, sizeof(unsigned char) * buffSize, cudaMemcpyHostToDevice)
...
int nBlocks = 1;
int nTh = buffSize / 16;

if (nTh > MAX_THR_PBLK) {
  nBlocks = (nTh / MAX_THR_PBLK) + 1;
  nTh = MAX_THR_PBLK;
}
aes<<<nBlocks, nTh>>>(buffGPU, keysGPU, buffSize / 16);
...
cudaMemcpy(buffer, buffGPU, sizeof(unsigned char) * buffSize, cudaMemcpyDeviceToHost)
....
```

O trecho de código acima mostra a chamada do kernel do AES. A variável *buffSize* é o total de bytes lidos do arquivo, sendo que seu valor máximo é *MAX_BUFFER_SIZE (512MB)*. Se o número de threads necessárias para processar o arquivo for maior que o número máximo de threads, então aumenta o número de blocos. No fim, o buffer da GPU (*buffGPU*) é copiado sob o *buffer* da CPU e enviado para a saída de dados. Caso o arquivo não tenha sido processado por inteiro, novas chamadas do kernel são executadas na sequência.

Otimização

O seguinte trecho de código, localizado em gpu_src/teste_gpu.cu, identifica o número máximo de threads por bloco da GPU e atualiza a variável MAX_THR_PBLK:

```
int countDevices = 0;
cudaGetDeviceCount(&countDevices);
for(int i = 0; i < countDevices; i++) {
  cudaDeviceProp prop;
  cudaGetDeviceProperties(&prop, i);
  MAX_THREADS = prop.maxThreadsPerBlock;
  break;
}
```

Como executar o programa

Nos arquivos fontes do trabalho encontra-se um arquivo **Makefile**, que é responsável por compilar os arquivos e gerar dois executáveis: **run** (CPU) e **run_gpu** (GPU). Para que o programa fosse testado mais facilmente, a chamada dos executáveis exige apenas o caminho do arquivo de origem. A chave de criptografia e o arquivo de saída foram fixados para os testes não se tornarem complexos. Sendo assim, os programas podem ser executados da seguinte maneira

./run <caminho_arquivo_entrada>
./run_gpu <caminho_arquivo_entrada>

A execução na CPU irá gerar o arquivo **cpu.out**, com o arquivo criptografado, já a execução na GPU irá gerar o arquivo **gpu.out**. A chave de criptografia é um vetor de 16 bytes com números de 0x1 até 0x10.

Junto com os fontes do trabalho, encontra-se um arquivo chamado **run_tests.py** que executa os programas usando todos os arquivos que estão na pasta **amostras**. No fim da execução, dois gráficos são gerados: **velocidade.pdf** e **aceleracao.pdf**, que são os gráficos apresentados neste trabalho. Além disso, existe uma pasta chamada **output_tests**, que contém os resultados das execuções das amostras. Essa pasta contém arquivos com nome no formato **{cpu, gpu}_{número da amostra}_{número da execução}**. Dentro de cada arquivo, existe um texto padrão no seguinte formato: **total_bytes_lidos|tempo_milisegundos**. Se a pasta output_tests não estiver vazia, run_tests.py irá gerar os mesmos gráficos, o que significa que para executar novamente todas as amostras, esse diretório deve estar vazio.

Testes

Gerar amostras:
python run_tests.py 2

Limpar logs antigos e rodar:
python run_tests.py 1

Rodar testes (se não for feito clean (passo anterior), apenas irá gerar os gráficos):
python run_tests.py

Avaliação

Ambiente de teste

Dados da CPU:

```
Arquitetura:           x86_64
Modo(s) operacional da CPU:32-bit, 64-bit
Byte Order:            Little Endian
CPU(s):                4
On-line CPU(s) list:   0-3
Thread(s) per núcleo  1
Núcleo(s) por soquete:4
Soquete(s):            1
Nó(s) de NUMA:        1
ID de fornecedor:      GenuineIntel
Família da CPU:       6
Modelo:                158
Model name:            Intel(R) Core(TM) i5-7400 CPU @ 3.00GHz
Step:                  9
CPU MHz:               1137.451
CPU max MHz:           3500,0000
CPU min MHz:           800,0000
BogoMIPS:              6000.00
Virtualização:       VT-x
cache de L1d:          32K
cache de L1i:          32K
cache de L2:           256K
cache de L3:           6144K
NUMA node0 CPU(s):     0-3
```

Dados da GPU:

```
CUDA Device #0
Major revision number:         6
Minor revision number:         1
Name:                          GeForce GTX 1060 6GB
Total global memory:           2069495808
Total shared memory per block: 49152
Total registers per block:     65536
Warp size:                     32
Maximum memory pitch:          2147483647
Maximum threads per block:     1024
Maximum dimension 0 of block:  1024
Maximum dimension 1 of block:  1024
Maximum dimension 2 of block:  64
Maximum dimension 0 of grid:   2147483647
Maximum dimension 1 of grid:   65535
Maximum dimension 2 of grid:   65535
Clock rate:                    1784500
Total constant memory:         65536
Texture alignment:             512
Concurrent copy and execution: Yes
Number of multiprocessors:     10
Kernel execution timeout:      Yes
```

Total de bytes do arquivo

O total de bytes do arquivo foi calculado através de um acumulador, que somava a quantidade de bytes lidos do arquivo a cada iteração necessária.

Cálculo do tempo

Na implementação da GPU, o tempo foi calculado utilizando as funções cudaEventCreate, cudaEventRecord, cudaEventSynchronize, cudaEventElapsedTime e cudaEventDestroy. Essas funções calculam a diferença de tempo em milissegundos entre o início da simulação e fim da mesma. Tempo gasto com entrada/saída de dados e transferência de dados entre CPU e GPU foram considerados. Na CPU o tempo foi calculado utilizando a biblioteca std::chrono, considerando entrada e saída de dados

Implementação na CPU

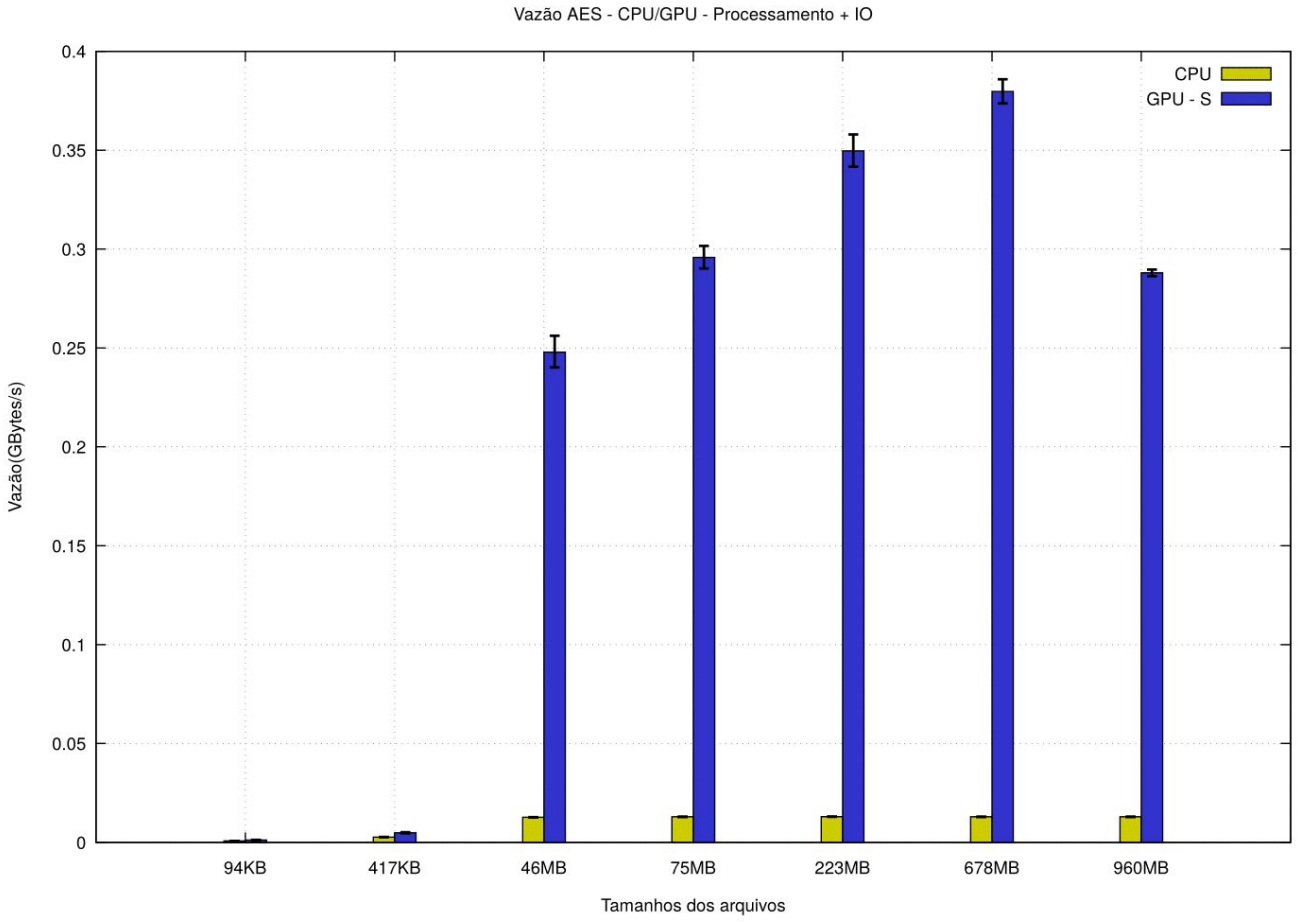
A implementação do AES na CPU faz parte deste trabalho. Foi implementada de maneira sequencial e sem usar bibliotecas externas com funções do AES.

Resultados - Velocidade

O gráfico a seguir mostra os resultados obtidos quando avaliado a velocidade de processamento dos algoritmos na CPU e na GPU. No eixo horizontal temos os tamanhos

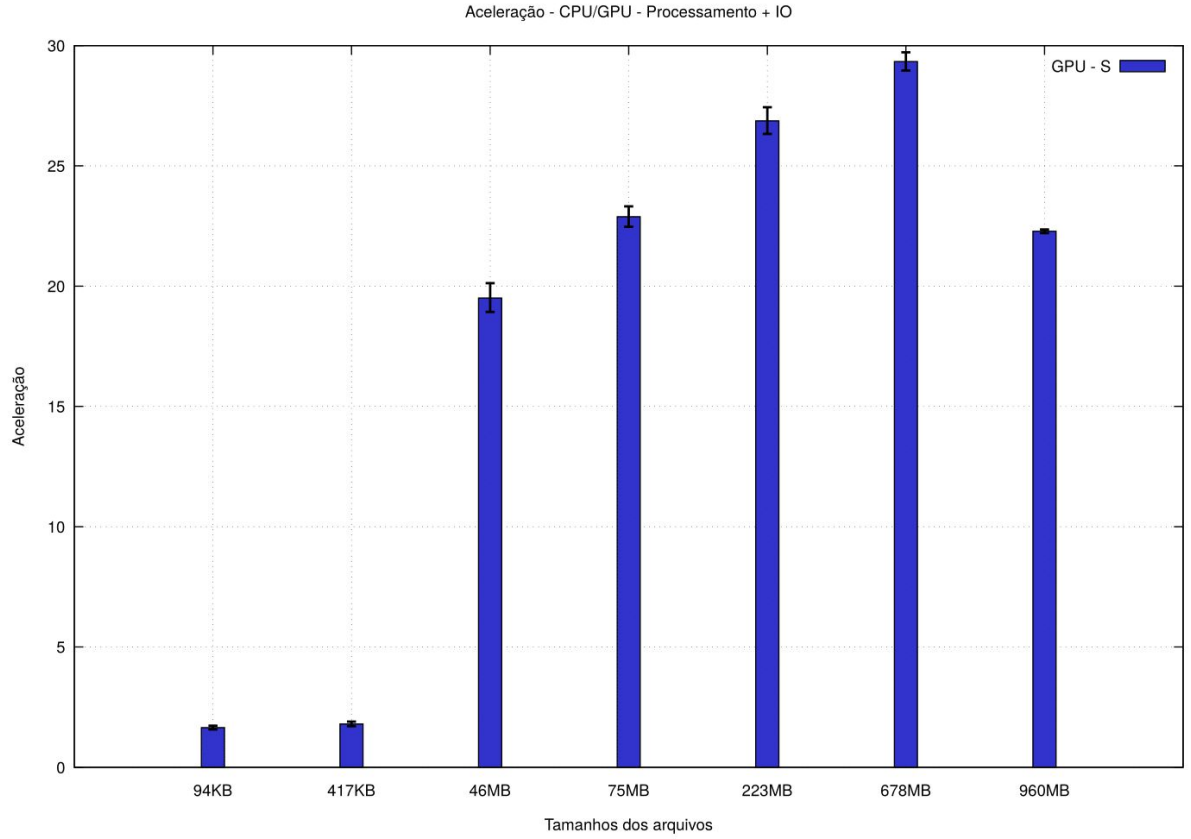
dos arquivos ordenados de maneira crescente, e no eixo vertical temos a velocidade de processamento em GBytes/s.

O tempo em cada execução se inicia após a leitura dos primeiros *buffSize* bytes do arquivo e termina quando todos os bytes do arquivo foram processados. Esse tempo inclui a transição de dados do disco para a memória RAM e da memória RAM para a memória da GPU. Todos os experimentos foram executados 20 vezes, e os resultados apresentados no gráfico são uma média das 20 execuções com intervalo de confiança de 95%.

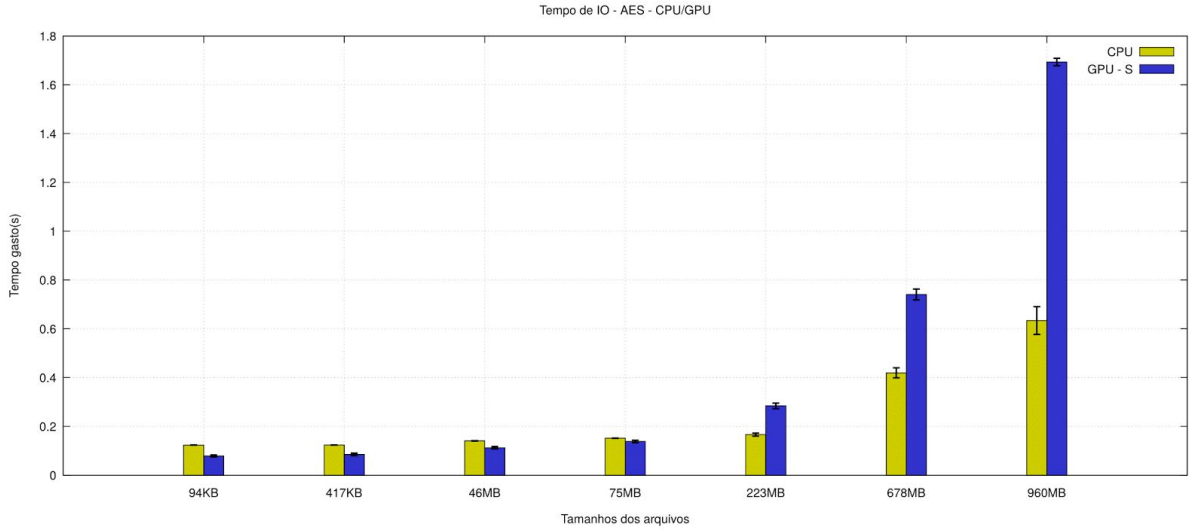


No gráfico de velocidade, podemos observar que o algoritmo AES na GPU foi amplamente mais rápido que a versão sequencial. No entanto, nota-se que a partir da amostra de 678MB, há uma tendência de a velocidade cair gradativamente. Isso ocorre porque acima de 512MB, é necessário ler mais de uma vez os arquivos do disco e trazê-los para a memória RAM e em sequência para a GPU. Podemos notar também que essa queda desempenho afeta mais a GPU do que a CPU, justamente pelo fato de ter que copiar os dados de/para GPU.

No gráfico a seguir, é mostrado a aceleração obtida com a implementação na GPU em relação a CPU. Podemos ver que houve um aumento próximo a 30 vezes no caso da amostra de 223MB. Além disso, podemos notar mais nitidamente a queda de desempenho do algoritmo quando as amostras ultrapassam o tamanho de *buffSize*.

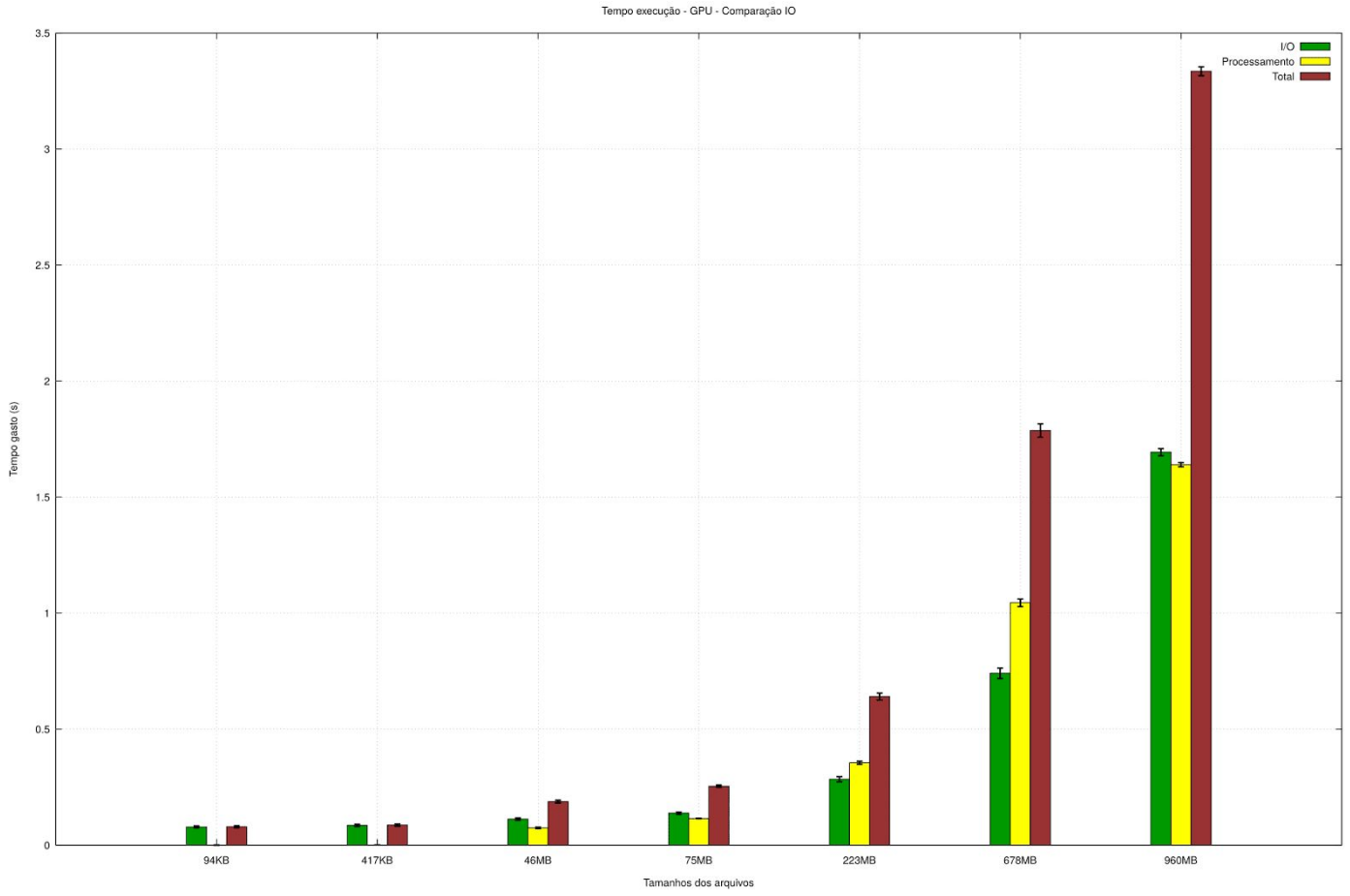


A seguir fizemos uma análise do tempo de simulação médio gasto com IO. O tempo de IO da CPU é apenas o tempo de ler/escrever dados no disco. O tempo de IO da GPU é o tempo de ler/escrever dados no disco mais o tempo gasto transferindo os dados lidos para a GPU.



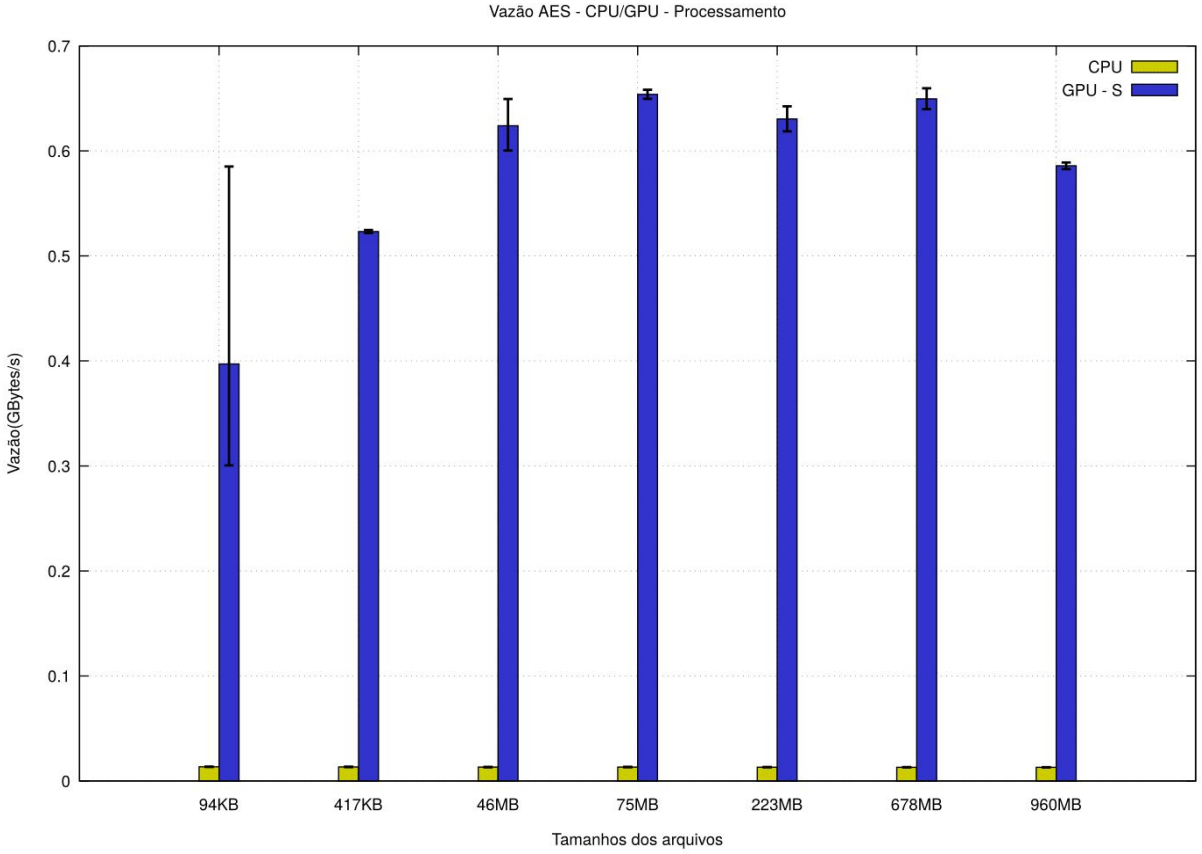
Curiosamente, podemos ver que para instâncias pequenas a CPU levou mais tempo que a GPU para fazer IO. Podemos dizer que essa anomalia ocorre porque o ambiente não estava 100% isolado no momento das simulações, além disso, a diferença de tempo nessas instâncias foi muito inferior a 0.1s. Para instâncias maiores, a GPU chegou a gastar quase 2 segundos com I/O, o que mostra que o I/O afetou o tempo de execução do programa.

No gráfico a seguir, podemos ver uma comparação entre os tempos de IO, processamento e total do programa:



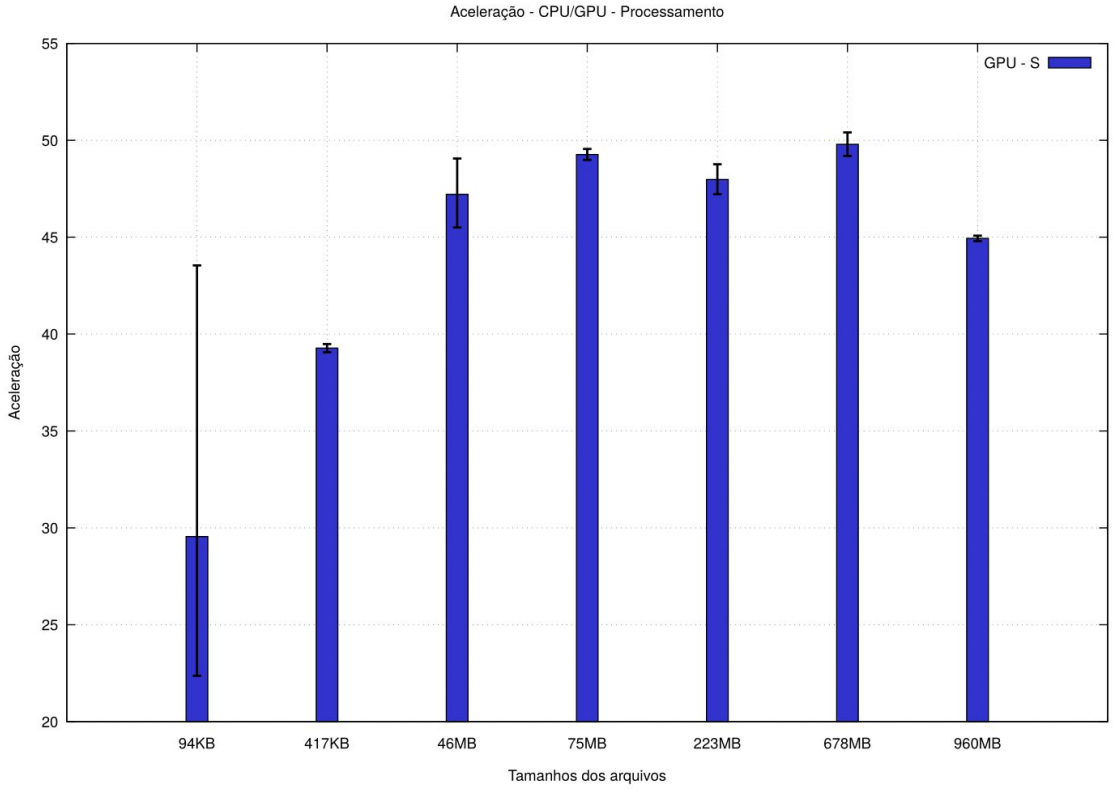
Como pode ser notado, em muitos casos o tempo de IO foi maior que o tempo de processamento, como era de se esperar. Em instâncias muito pequenas, o tempo de IO foi mais de 90% do tempo gasto pelo algoritmo.

Considerando apenas tempo de processamento, obtivemos os seguintes resultados para a vazão:



Podemos ver pelo gráfico acima que a diferença de vazão da GPU para a CPU foi ainda maior se desconsiderarmos operações de I/O. O arquivo de 94KB teve uma grande variação em sua vazão, e atribuímos esse comportamento ao fato de o ambiente de simulação não estar 100% isolado no momento da simulação.

Para a aceleração, obtivemos os seguintes resultados:



Podemos notar que se considerarmos apenas o tempo de processamento, a solução paralela usando GPU pode ser até 52 vezes mais rápida que uma versão sequencial na CPU.

Conclusão

Concluiu-se neste trabalho que a implementação do algoritmo AES é eficiente na GPU. Os resultados obtidos mostram que a GPU foi superior à CPU quanto à velocidade de execução das amostras utilizadas e que podemos obter uma aceleração de 30 vezes quando consideramos IO. Quando o IO não é considerado, a aceleração foi acima de 50 vezes. Além disso, foi possível observar que o I/O chegou a tomar 90% do tempo de execução do programa para a algumas instâncias.

Trabalhos futuros

Para um trabalho futuro, pretendo refazer a implementação utilizando a shared memory. Uma ideia inicial é dividir os 49152 bytes da shared memory entre as 1024 threads do bloco. Dessa maneira, cada thread processaria 48 bytes que são equivalentes a 3 estados, aproveitando toda a shared memory e trabalhando de maneira mais persistente do que a implementação atual. Quanto aos resultados, seria interessante comparar o desempenho deste trabalho com a versão utilizando shared-memory, a CPU em sequencial e a CPU em paralelo, utilizando o OpenMP, por exemplo. Por fim, poderia testar as implementações variando o valor de *MAX_BUFFER_SIZE* para verificar se há um impacto significativo na velocidade de processamento da amostra muito grandes.

Referências

[1]:http://aes.online-domain-tools.com/
[2]: https://en.wikipedia.org/wiki/Rijndael_S-box