Paralelização do Algoritmo AES e Análise Sobre GPGPU

Douglas Tybusch[1], Gustavo Stangherlin Cantarelli[1]

[1]Centro Universitário Franciscano Santa Maria – RS – Brasil

douglastybusch@hotmail.com, gus.cant@gmail.com

Abstract. *This paper deals with the parallelized implementation of the AES (Advanced Encryption Standard) cryptographic algorithm using the CUDA technology to obtain greater speeds in both the encryption and decryption processes. This implementation's purpose is to create a speed comparison between the processing capabilities of a CPU (Central Processing Unit) and a GPU (Graphical Processing Unit). This comparison has become necessary due to breakthrough technologies in general processing of the latest generations of GPUs, as well as due to the increased use of encryption and decryption in data transfer and storage.*

Resumo. *Este trabalho trata da implementação do algoritmo criptográfico AES (Advanced Encryption Standard) de forma paralelizada, utilizando a tecnologia CUDA para a obtenção de maior velocidade nos processos de criptografia e descriptografia. Esta implementação tem como objetivo criar um comparativo de velocidade entre as capacidades de processamento de uma CPU (Central Processing Unit) e uma GPU (Graphical Processing Unit). Este comparativo se mostra necessário devido ao grande avanço nas tecnologias de processamento geral das últimas gerações de GPU, como também ao aumento da utilização de criptografia e descriptografia na transferência e armazenamento de dados.*

1. Introdução

No passado, as ferramentas utilizadas para desenvolver softwares para GPUs eram muito específicas, limitando-se a renderização de gráficos. Em 2006 a desenvolvedora de chips NVIDIA lançou com sua nova geração de placas de vídeo, o CUDA (*Compute Unified Architecture)*, uma arquitetura de computação unificada, como uma extensão para a linguagem C, onde permitia a "conversão" de programas *single-thread* em programas paralelos a serem executadas em suas GPUs [LUKEN, OUYANG, e DESOKY, 2009]. O CUDA serviu de base para a criação de novas tecnologias, como o OpenCL.

Com as crescentes demandas de transferências de dados, surge a necessidade de armazenar dados de forma segura, e acessá-los de forma rápida. Embora o método de criptografia AES seja seguro e eficiente, a quantidade de dados a serem criptografados simultaneamente é grande e crescente, gerando tempos de resposta cada vez mais baixos [LUKEN, OUYANG e DESOKY, 2009].

Este trabalho tem por objetivo desenvolver um *software* para a criptografia e descriptografia de dados de forma paralela, a fim de gerar um comparativo de performance entre o modo clássico de processamento via CPU com o método de processamento paralelo através do uso de GPGPU (Computação de Propósito Geral na Unidade de Processamento Gráfico), método resultante da evolução de tecnologias como o CUDA e o OpenCL em conjunto com as últimas gerações de GPUs.

Os objetivos deste trabalho são:

•Implementar e paralelizar o algoritmo AES para criptografia e descriptografia de arquivos;

•Efetuar testes de performance comparativos do algoritmo AES implementado tanto para processamento via CPU quanto via GPU.

Assim, a partir do desenvolvimento deste *software*, analisar e documentar os problemas e dificuldades encontradas durante o processo de paralelização do algoritmo para execução em GPGPU, como também a realização de uma análise dos diferentes tipos de testes de performance realizados.

2. CUDA e arquitetura de programação GPGPU

CUDA é uma plataforma computacional e modelo de programação desenvolvido pela NVIDIA, com o objetivo de facilitar o acesso ao processamento paralelo via GPU, através de linguagens como C, C++ ou Fortran, permitindo controle e paralelização de GPUs NVIDIA antes somente possíveis através de linguagens de baixo nível, como o Assembly [NVIDIA, 2014a].

O CUDA inclui uma linguagem de programação e sintaxe baseada no C, utilizando seu próprio compilador, o NVCC, baseado no compilador C/C++, capaz de executar a grande maioria de códigos C/C++, e para o desenvolvimento dos *kernels* a serem executados na GPU [NVIDIA, 2014b].

Para satisfazer as necessidades de programação paralela e utilização de *kernels* do CUDA, a linguagem foi estendida com diversos recursos, como funções para sincronização de *threads*, paralelização de operações e vetores e maior estruturação na alocação de dados na memória. Também foram criadas funções para determinação de características arquitetônicas de um dispositivo, como capacidade de memória, e quantidade de núcleos e *threads* de uma GPU a ser utilizada [NVIDIA, 2014c].

2.1 Estrutura de memórias

A estrutura de memórias estão representadas pelas seguintes regiões apresentadas na Figura 1, conforme a estrutura de uma placa de vídeo:

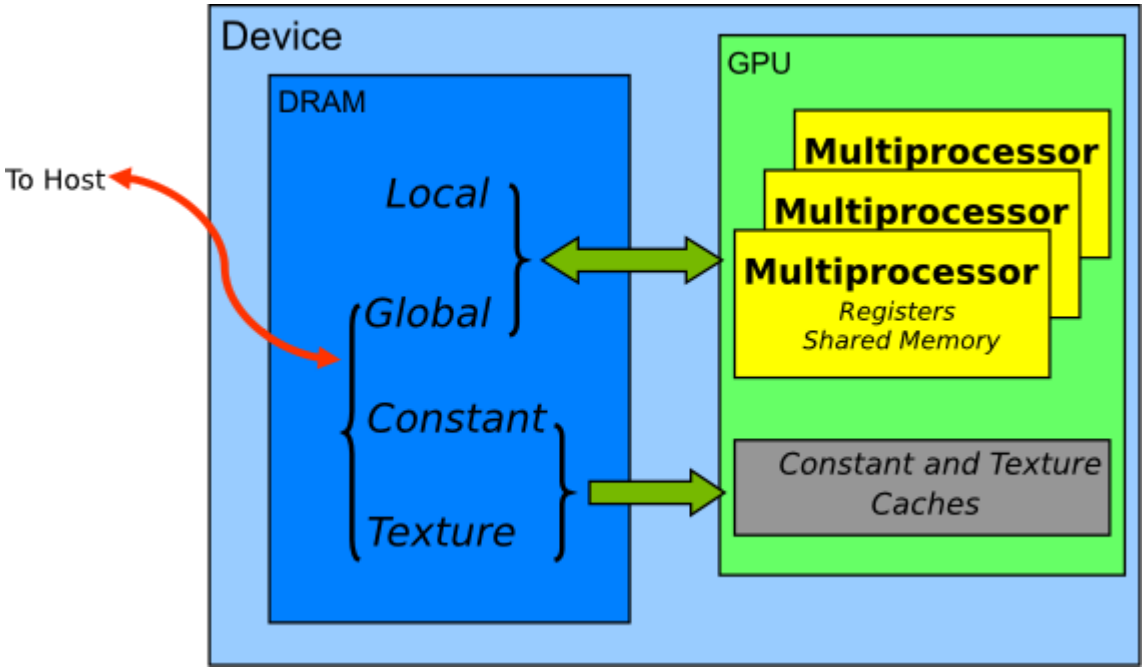

Figura 1. Tipos e relacionamentos de memórias pelo CUDA [NVIDIA, 2014c]

Memória global: O espaço de maior capacidade de um dispositivo, onde os dados são visíveis a todos os *threads/kernels* alocados. É também, a região de acesso mais lento, devido a cuidados como a sincronização e ao acesso de dados.

Memória Constante: Armazenada pelo mesmo banco de memórias da memória global, é utilizada apenas para leitura, sendo otimizada através de cache para uma maior velocidade de acesso.

Memória Local: Também armazenada pelo mesmo banco de memórias da memória global, é utilizada para escrita e leitura dentro do escopo de um *thread.*

Registradores: Armazenada no próprio *chip*, são memórias de acesso extremamente rápido e de baixa quantidade de armazenamento. São memórias de uso exclusivo de suas *threads*, e armazena dados como endereços de variáveis e ponteiros. Seu uso excessivo pode gerar a realocação de dados para outras regiões mais lentas.

Memória Compartilhada: Armazena dados a serem compartilhados entre *threads* de um mesmo bloco, afim de reduzir o acesso a dados da memória global. Esta memória é armazenada no próprio chip gráfico.

2.2 Grupos de Trabalho e Sincronização

No CUDA, os dados são localizados e associados como *CUDA-cores*, ou blocos. *CUDA-cores* são a menor entidade de execução no CUDA, sendo equivalentes a *threads*, as quais pertencem a um bloco (um grupo *threads)* onde todos *threads* pertencentes a um mesmo bloco executam um *kernel* em comum [NVIDIA, 2014c]. A vantagem da organização por blocos se deve a habilidade de sincronia e compartilhamento de dados de forma eficiente entre os *threads* agrupados, como o uso de memória compartilhada.

A diferença de *kernels* para blocos ou cuda-cores pode ser dada como:

Kernels: Representam códigos e funções diferentes a serem executadas;

Blocos: Agrupamento de instâncias de um mesmo *kernel*, executando um código comum, agrupados de forma a compartilhar recursos;

Cuda-cores: *Threads* de execução instanciados a partir de um *kernel*.

Na Figura 2 é exibida a organização entre os *threads* e blocos, onde *grid* representa o dispositivo GPU e seu total de *CUDA-cores* e *block* representa um bloco de *threads*:

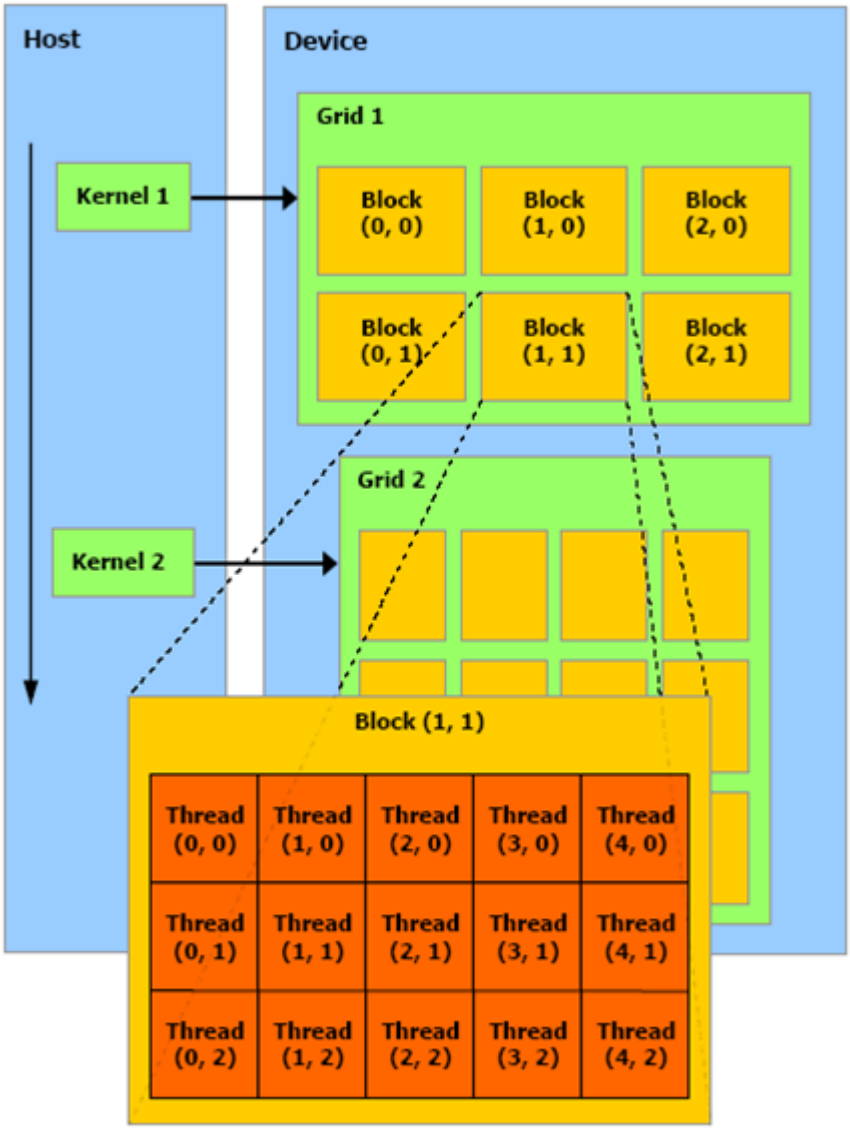

Figura 2. Organização e sincronia de threads em CUDA [NVIDIA, 2014c]

No CUDA, devido à forma como seu compilador funciona, é possível re-ordenar operações de memória, para utilizar de forma otimizada a arquitetura do dispositivo em uso, assim, operações de acesso a memória podem não ocorrer de forma sequencial e ordenada conforme o código fonte [NVIDIA, 2014c].

A sincronia de memória pelo CUDA pode ser feita pela utilização de três diferentes funções: *threadfence_block()*, *threadfence()* e threadfence_system(), onde:

threadfence_block(): Garante que todas escritas na memória compartilhada e global feitas pela *thread* estejam em sincronia com outras ações feitas por *threads* agrupadas em um mesmo bloco;

threadfence(): Assim como a função *threadfence_block()*, garante sincronia entre os dados armazenados na memória global e compartilhada, porém entre todas as *threads* do dispositivo de GPGPU.

threadfence_system(): Assim como a função *threadfence()*, garante sincronia entre os dados armazenados na memória global e compartilhada, porém entre todas as *threads* do dispositivo de GPGPU, do dispositivo de *host* (CPU) e também outros dispositivos GPU.

No CUDA, também são disponíveis funções para a sincronização de *threads,* como a função *syncthreads()*, a qual garante que todas *threads* do bloco tenham alcançado determinado ponto de execução do *kernel*, antes de executar o restante do código.

3. *Advanced Encryption Standard*

*Advanced Encryption Standard (*AES) é uma especificação para criptografia de dados eletrônicos estabelecido pelo National Institute of Standards and Technology (NIST) [NIST, 2001]. A criptografia constitui de técnicas para converter informações e dados de sua forma original, para formatos ilegíveis, de forma a proteger estas informações, e serem recuperadas apenas por detentores da chave criptográfica.

Em 2001, o NIST selecionou e aprovou como o AES, o algoritmo Rijndael, nome derivado de seus criadores belgas, Vincent Rijmen e Joan Daemen [NIST, 2001]. Primeiramente publicado em 1998, o algoritmo Rijndael é um algoritmo de bloco de cifras simétrico que permite criptografar e descriptografar informações. Por simétrico, entende-se que é um algoritmo que utiliza da mesma chave criptográfica para ambos os processos de criptografia de descriptografia de dados, e por bloco de cifras, que utiliza de blocos de dados de tamanho fixado de bits [NIST, 2001].

Com a crescente demanda de coleta e armazenamento de dados, surge a necessidade de armazenar dados de forma segura, e a necessidade de acessá-los de forma rápida. Embora o método de criptografia AES, seja bastante seguro e eficiente, a quantidade de dados a serem criptografados simultaneamente é significativamente grande, e pode gerar tempos de resposta baixos e lentidões em serviços [LUKEN, OUYANG e DESOKY, 2009].

Este é um dos problemas de possível solução através da utilização de GPGPU para a computação de algoritmos de criptografia, podendo gerar ganhos na performance e velocidade no processamento destes dados.

3.1 Funcionamento do AES

A especificação AES baseada no algoritmo Rijndael, possui algumas diferenças do algoritmo original para assegurar níveis de segurança requeridos pelo governo norte

americano, tais como a fixação do tamanho do bloco de dados em 128 bits, e chaves criptográficas de 128, 192 ou 256 bits, onde o algoritmo original poderia utilizar chaves e blocos de qualquer número múltiplo de 32 bits entre 128 e 256 bits [DAEMEN e RIJMEN, 2002].

No AES, os dados a serem tratados são processados como uma matriz de tamanho 4x4 *bytes*, totalizando assim 16 *bytes*, ou 128 bits. O processo de criptografia ou descriptografia de dados se dá a partir da passagem do bloco de dados por diversas operações, as quais realizam a conversão do bloco, e são descritas a seguir:

Expansão da Chave: Processo em que são geradas diversas *round-keys* conforme o algoritmo de *Rijndael's key schedule*, onde a primeira *round-key* é derivada diretamente da chave criptográfica, e cada *round-key* subsequente baseada na *round-key* anterior;

AddRoundKey: Processo onde cada *byte* do bloco de dados é combinado com o respectivo byte da *round-key* (demais rounds) através da operação X-OR a nível de bits;

ShiftRows: Processo onde os *bytes* de cada linha são deslocados de forma cíclica N elementos a esquerda. Com N começando em 0 (sem substituições), e aumentando em 1 a cada linha do bloco de dados;

Mixcolumns: Função onde os *bytes* de uma coluna são combinados utilizando uma transformação linear invertível, multiplicando os dados desta coluna por uma matriz pré-estabelecida seguindo os princípios de *Rijndael's Galois Field*. Sua implementação pode ser reduzida à substituição dos valores destes *bytes*, por valores em tabelas pré-estabelecidas;

SubBytes: É feita uma operação de substituição, onde cada byte (em formato hexadecimal) é substituído por outro a partir da utilização de uma tabela pré-estabelecida chamada *S-box*;

Para os processos de criptografia e descriptografia, os blocos de dados passam pelas mesmas fases de conversão, apesar de utilizarem as operações em ordens diferentes. Na fase incial é feita a expansão da chave e a passagem do bloco pela operação de AddRoundKey. Em seguida, o bloco é processado por diversas iterações sobre as operações de SubBytes, ShiftRows, MixColumns e AddRoundKey. Este número de iterações a serem repetidas, e quantidade de *round-keys* a serem geradas, são diretamente relacionadas ao tamanho da chave criptográfica:

Chave de 128 bits: 10 ciclos de repetição e *round-keys;*

Chave de 192 bits: 12 ciclos de repetição e *round-keys;*

Chave de 256 bits: 14 ciclos de repetição e *round-keys*.

Após a realização destes processos, o bloco de dados passa por uma etapa final composta apenas das operações de SubBytes, ShiftRows e AddRoundKey, excluindo-se o uso da operação MixColumns. Na Figura 3 é visto de maneira geral os processos para criptografia e descriptografia de um bloco de dados com chave de 128 bits.

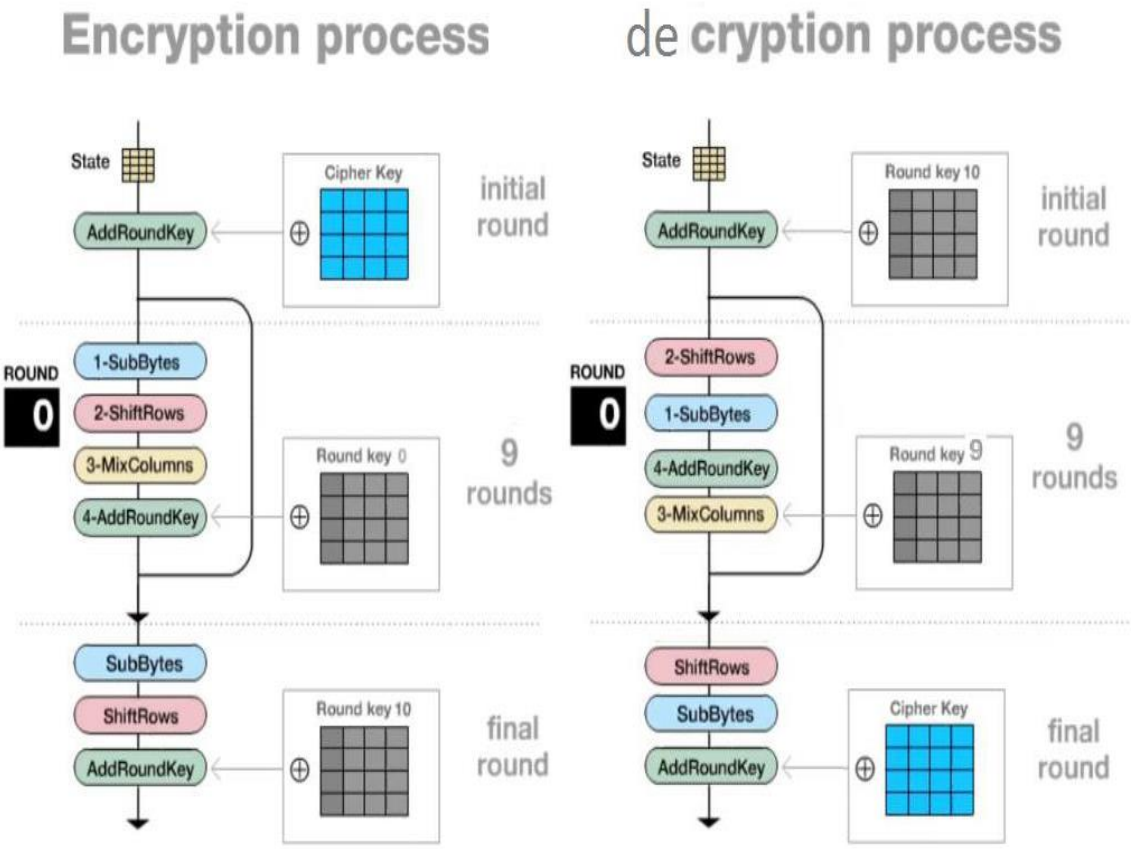

Figura 3. Visualização geral do processo de criptografia e descriptografia, adaptado de Zabala (2013).

4. Trabalhos Relacionados

No artigo *AES and DES Encryption With GPU* de Luken et all (2009), foram implementados 3 tipos diferentes de criptografia AES e DES: onde os algoritmos são executados apenas através da CPU; onde são executados exclusivamente pela GPU; e onde todos os passos são feitos pela GPU, exceto o escalonamento da chave criptográfica. Também foram implementadas versões para diferentes tamanhos da chave, de 16, 24 e 32 *bytes* respectivamente [LUKEN, OUYANG e DESOKY, 2009].

Para comparativo de performance entre as diferentes implementações, foram gerados *benchmarks* com arquivos de tamanho entre 1KB e até 100MB, utilizando chaves criptográficas com tamanho de 16, 24 e 32 *bytes*, mensurando o tempo total de execução em milissegundos. Os resultados para os *benchmarks* podem ser vistos na Tabela 2:

Tabela 1. Comparativo de performance entre CPU e GPU, para cada tamanho de chave [LUKEN, OUYANG e DESOKY, 2009].

File Size	1K	10K	100K	1M	10M	100M
16-CPU	.039	.351	3.56	36.1	358	3596
16-GPU GPU	.503	1.58	2.24	11.7	102	1006
16-GPU CPU	.480	1.55	2.22	11.5	101	1003
24-CPU	.044	.419	4.21	42.9	425	4226
24-GPU GPU	.607	1.69	2.42	13.3	119	1177
24-GPU CPU	.553	1.65	2.39	13.3	117	1169
32-CPU	.052	.488	4.91	49.9	498	5010
32-GPU GPU	.687	1.74	2.60	15.1	136	1349
32-GPU CPU	.623	1.85	2.67	14.9	134	1337

5. Projeto e Implementação

Este projeto tem por objetivo desenvolver um aplicativo de interface básica, que será executado através de *prompt* de comando (Aplicativo de criptografia e descriptografia de dados), criado a partir de C/C++ em conjunto da API CUDA. E possui como principais funções o carregamento de arquivos para a memória, a execução de criptografia e descriptografia dos arquivos, opção para armazenamento dos arquivos criptografados e/ou descriptografados, e também a seleção dos modos de criptografia ou descriptografia dos arquivos:

- Via CPU (*single-thread* e *multi-thread*)
- Via GPGPU

Onde o modo de execução por CPU realiza todas as operações de criptografia e descriptografia e o carregamento de dados apenas pela utilização da CPU de um computador. O modo CPU *single-thread* representa um modelo de funcionamento onde o arquivo é processado de modo sequencial em blocos de 16 *bytes*, enquanto que o modo *multi-thread* permite a inicialização de N *threads*, onde cada *thread* realiza o processamento de diferentes blocos do arquivo, de modo paralelo.

No modo de execução por GPU, é realizado o carregamento de arquivos pela CPU, alocação e transferência dos arquivos para a GPU, e a execução das operações de criptografia ou descriptografia pela GPU de forma parecida com o modo CPU *multi-thread*.

Foi implementado também um modo de *benchmark* na utilização deste aplicativo. O modo *benchmark*, funciona de modo a efetuar o carregamento do(s)

arquivo(s) selecionado(s), e mensurar o tempo de processamento destes arquivos em diferentes pontos de execução para efeitos comparativos e para análise de performance.

O processo de paralelização do algoritmo AES no modo GPGPU, constitui-se da criação de *kernels* através da utilização da API CUDA, que executam as funções de criptografia e descriptografia do algoritmo AES, contendo pontos de sincronia como a expansão da chave criptográfica, alocação de tabelas (S-box) em memória compartilhada, alocação e transferência de arquivo(s) na memória global da GPU, a partir da memória RAM do computador.

5.1 Tecnologias Utilizadas

As tecnologias utilizadas neste projeto são listadas a seguir:

- Linguagem de programação C/C++ (Visual C++ 11.0): Utilizada para o desenvolvimento do aplicativo de criptografia e descriptografia AES *single-thread* e *multi-thread* pela sua eficiência e implementação de *threads* em ambiente *Windows*;
- Linguagem de programação CUDA 6.0: Utilizada para o desenvolvimento do aplicativo paralelizado via GPGPU de criptografia e descriptografia AES para a criação e utilização de *kernels* alocados em GPUs NVIDIA;
- CUDA *Compute Capability 3.0+*: Padrão que define capacidades computacionais (como quantidade de *threads*, memória compartilhada, quantidade de registradores) para placas gráficas NVIDIA.

5.2. Paralelização do Algoritmo Rijndael

O algoritmo para criptografia e descriptografia de dados foi baseado em um pacote de arquivos e funções referentes aos processos de criptografia do algoritmo Rijndael [Erdelsky, 2002]. O pacote foi criado e implementado por Vincent Rijmen, Antoon Bosselaers e Paulo Barreto e serviu como para implementação base do padrão AES pelo NIST (rijndael-api-fst.c). A diferença da implementação oficial padronizada pelo NIST se dá pelo fato de utilizar um tamanho fixo de blocos de 16 *bytes*, e chaves de 16, 24 ou 32 *bytes*.

5.2.1 Modo CPU *Single-Thread* e *Multi-Thread*

Inicialmente implementado para a criptografia ou descriptografia de *strings* em blocos fixados de 16 *bytes* (funções *rijndaelEncrypt* e *rijndaelDecrypt* respectivamente), foram necessárias algumas modificações nas funções do algoritmo original, para o processamento de arquivos e sua paralelização.

Para o processamento de arquivos, foram desenvolvidas funções para o carregamento de arquivos locais, onde os mesmos são alocados para uma variável do tipo *unsigned char** através de alocação dinâmica, assim armazenados na memória RAM do computador, para uso das funções de processamento de dados. Como o algoritmo realiza a criptografia em blocos de 16 *bytes*, o tamanho do arquivo alocado é arredondado para o próximo valor múltiplo de 16 (em *bytes*), durante o processo de alocação.

As funções originais *rijndaelEncrypt* e *rijndaelDecrypt* foram modificadas de forma a receber um ponteiro do arquivo alocado, como também um índice indicando o bloco/posição do arquivo a ser processado

Para o modo *single-thread* via CPU, após o carregamento do arquivo, é feito a operação selecionada de criptografia ou descriptografia do arquivo, através um laço de repetição que percorre a variável e processando-a em blocos de 16 em 16 *bytes*.

Para a paralelização do algoritmo AES em modo CPU *multi-thread*, foram desenvolvidas funções para a inicialização de N *threads* paralelos, onde o valor de N é baseado no número de núcleos da CPU do computador.

Após definido o número de *threads* a ser utilizado, são criados *structs* do tipo secao_Dados, os quais armazenam o endereco do arquivo a ser utilizado (*unsigned char* *arquivo), como também índices de início e fim, representando seções do arquivo a serem processados pela *thread*.

Os valores de início e fim são encontrados a partir da divisão do tamanho do arquivo em *bytes*, pela quantidade de *threads* a serem inicializadas, também de modo a gerar seções de tamanhos múltiplos de 16.

Após a definição destas *structs*, é feito então a inicialização das *threads* para processamento do arquivo, recebendo como parâmetro o endereço do arquivo, e sua respectiva seção a ser processada, existindo assim, múltiplas *threads* em paralelo processando o mesmo arquivo em diferentes seções.

5.2.2 Paralelização Modo GPU

Para a paralelização do algoritmo AES no modo GPU, foram portadas as principais funções do algoritmo AES (*rijndaelEncrypt* e *rijndaelDecrypt*), para código CUDA, tornando-se *cuda_rijndaelEncrypt* e *cuda_rijndaelDecrypt* respectivamente.

As principais alterações no código, foram no modo de declaração das funções de *void*, para *__global__ void*, indicando se tratar de um *kernel* de acesso global, e a alteração das tabelas S-box (Te0, Te1, Te2, Te3 e Te4) e S-box inverso (Td0, Td1, Td2, Td3 e Td4) do tipo static const u32 para __device__ u32, assim alocando estas tabelas na memória global da GPU.

As funções *rijndaelEncrypt* e *rijndaelDecrypt* funcionam de forma a receber o endereço do arquivo e o índice do bloco de 16 *bytes* a ser processado, porém no modelo de programação CUDA, os *kernels* são inicializados a partir de grupos de *kernels*, onde deve-se definir a quantidade de grupos a ser inicializado, e a quantidade de *threads* por grupo. Esta forma de inicialização impossibilita assim o envio de parâmetros diferenciados para cada *thread*, logo, não sendo possível o envio do parâmetro de índice do bloco a ser processado para cada *thread*.

Para contornar este problema, o índice do bloco a ser processado é encontrado a partir da seguinte formula:

*Indice = ((blockIdx.x * blockDim.x)*16) + (threadIdx.x * 16);*

Onde *blockIdx*, *blockDim*, *threadIdx* são variáveis especiais CUDA, indicando respectivamente o ID do bloco atual do *kernel* em execução, a quantidade total de

blocos inicializados, e o id do *thread* atual. A partir desta fórmula, é possível com que cada *thread* possa calcular o índice correto do bloco a ser processado, de forma independente.

Outra mudança e forma de otimização utilizada, foi a transferência das tabelas *S-box* e *S-box* inverso da memória global da GPU, para a memória compartilhada do grupo de *threads*, gerando um acesso consideravelmente mais rápido pelos *threads* as tabelas necessárias. Esta transferência é feita através do *kernel* de criptografia ou descriptografia, onde no início da execução, cada *thread* é responsável por copiar um valor da memória global para a memória compartilhada do grupo.

Kernels CUDA também não possuem acesso direto à memória RAM do computador, sendo necessária a alocação dos arquivos na memória global do dispositivo GPU. Existem diversas tipos funções para a transferência destes dados, como o tipo *pinned allocation* que funciona como um método *malloc* de variáveis, ou métodos *zero-copy* que funcionam como uma forma a realizar *streaming* de dados diretamente da memória RAM do computador, para a variável no dispositivo GPU conforme o uso.

Nesta implementação, a transferência do arquivo para a GPU foi feita a partir dos métodos *cudaMalloc()* e *cudaMemcpy()*, onde primeiramente é alocado o espaço do arquivo na memória global da GPU, e logo em seguida é copiado o arquivo para a variável alocada.

Após a alocação do arquivo na memória da GPU, é possível então a inicialização dos *kernels* adaptados para o processamento do arquivo.

6. Resultados e Benchmarks

Nesta seção serão analisados os resultados de performance obtidos a partir da paralelização do algoritmo AES. Os testes de performance foram realizados a partir do tempo da execução dos processos de criptografia e descriptografia em arquivos de diferentes tamanhos: 1KB, 1MB, 10MB, 100MB e 1000MB.

A configuração do computador onde os testes de performance foram realizados:

- Processador Core i5 750 com *clock* de 3.8ghz;
- Memória 8GB DDR3 com *clock* de 1800mhz;
- Placa de vídeo GTX680 (1536 *cuda-cores* e *clock* dinâmico de 915mhz);
- SSD 128GB Crucial C300 (leitura sequencial de 250MB/s)

6.1 Performance no processo de criptografia de Dados

Neste teste, foi analisada apenas a performance do processo criptográfico dos arquivos selecionados, ignorando-se tempos como carregamento de arquivos do HD para a memória RAM, e transferência de arquivos da memória RAM principal para a memória global da GPU.

Os testes realizados mensuraram o tempo necessário para processamento pelos algoritmos de CPU *single-thread* e *multi-thread*, e GPU para arquivos de 1KB, 1MB, 10MB, 100MB e 1000MB, conforme Tabela 2:

Tabela 2. Tempo em segundos para o processamento de dados nas diferentes implementações (* indica um valor não significativo)

Tam. do arquivo	Tempo de processamento CPU (*Single-Thread)*	Tempo de processamento CPU(4 *Threads)*	Tempo de processamento GPU (2048 *Threads)*
1KB	*	*	*
1MB	0.015	*	*
10MB	0.093	0.046	0.016
100MB	0.922	0.281	0.078
1000MB	9.172	2.656	0.625

A partir destes resultados é possível notar um aumento de performance escalável em relação ao número de núcleos de um processador no algoritmo de execução em modo CPU *multi-thread* em relação ao modo *single-thread*. Para o modo CPU *multi-thread* também foram realizados testes de performance com um número de *threads* maior que a quantidade de núcleos de CPU disponíveis no computador de teste, mas não apresentaram qualquer aumento de performance, indicando o uso completo de processamento de cada núcleo por sua respectiva *thread*.

Com estes resultados, é visto também um claro aumento de performance no algoritmo de execução em GPU em relação a ambos os algoritmos CPU *single-thread* e *multi-thread*, obtendo uma performance de até 424% mais rápida no tempo de processamento em comparação ao algoritmo de execução CPU *multi-thread*, no arquivo de 1000MB.

6.2 Tempos de Transferência de Arquivos

Nesta bateria de testes foram considerados apenas os tempos de carregamento dos arquivos do HD para a memória RAM do computador (evento realizado antes de ambos os processos de processamento na CPU e GPU), e também os tempos de transferência dos arquivos da memória RAM para a memória global do dispositivo GPU e vice-versa, eventos efetuados antes e após os processos de criptografia ou descriptografia do arquivo na GPU, conforme Tabela 3:

Tabela 3. Tempo em segundos para carregamento e transferência de arquivos (* indica um valor não significativo)

Tam. Do Arquivo	Tempo (s) de Carregamento para memória RAM	Tempo (s) de Transferência CPU → GPU e GPU → CPU
1KB	*	0.062
1MB	0.015	0.062
10MB	0.046	0.093
100MB	0.421	0.157
1000MB	4.390	0.922

Com estes resultados é possível perceber que o carregamento e transferência de arquivos, se torna o processo mais custoso entre todos os eventos para a criptografia ou descriptografia de um arquivo. Um processo ligado diretamente à velocidade de leitura do HD, com uma banda efetiva de 227MB/s em um arquivo de 1000MB.

Também analisando os valores, pode-se notar que a transferência de arquivos de tamanhos pequenos (1MB ou menos) podem gerar uma quantidade grande de *overhead* na transferência de arquivos CPU → GPU, assim gerando uma performance geral, muito inferior para o processamento de arquivos pequenos na GPU.

Uma solução para contornar este problema, pode ser a transferência em blocos (ou em massa), carregando e transferindo diversos arquivos em uma única variável, assim ignorando os custos de *overhead* das diversas micro transferências.

6.3 Tempo Total do Processo de Criptografia ou Descriptografia

Nestes últimos testes foram levados em conta o tempo total para o processamento de um arquivo, contabilizando-se os tempos de carregamento do arquivo, e o tempo processamento de dados no caso de algoritmos via CPU, ou o tempo de carregamento do arquivo, a sua transferência CPU→GPU, o seu tempo de processamento via GPU, e também o tempo de transferência GPU→CPU, conforme Tabela 4.

Tabela 4. Tempo em segundos para o processo completo de criptografia (* indica um valor não significativo)

Tam. Arquivo	Tempo (s) CPU (*single-thread*)	Tempo (s) CPU (4 *threads)*	Tempo (s) GPU (2048 *threads)*
1KB	*	*	0.062
1MB	0.03	0.015	0.077
10MB	0.139	0.092	0.155
100MB	1.343	0.702	0.656
1000MB	13.562	7.046	5.937

Levando em conta todos os processos para a criptografia ou descriptografia de arquivos, nota-se que o algoritmo de CPU *multi-thread* apresenta uma melhora geral de desempenho em todos os tamanhos de arquivos, enquanto que o algoritmo GPU se torna mais eficiente apenas em arquivos de tamanho maior a 100mb.

6.4 Análise dos Resultados

Analisando os resultados, é possível perceber um grande potencial de processamento paralelo a partir da utilização de placas de vídeo, e também a escalabilidade do algoritmo AES Rijndael em termos de paralelização, onde o algoritmo paralelizado em GPU obteve um resultado de até 424% mais velocidade de processamento em relação ao algoritmo CPU *multi-thread*.

Porém, apesar da grande capacidade de processamento, existe um notável gargalo nos processos de criptografia e descriptografia de arquivos, capaz de negativar em parte, os ganhos de processamento via GPU, que é a velocidade de transferência de arquivos. Problema que afeta especialmente arquivos de tamanhos pequenos, que

possuem baixos tempos de processamento, levando diversas vezes mais tempo para alocar o mesmo no dispositivo de GPU, do que realizando o próprio processamento.

7. Conclusões

Neste trabalho foi apresentado como é realizado o processo de criptografia e descriptografia de dados utilizando o padrão AES, bem como a importância de sua utilização e problemas quanto a performance. Também foi visto que por ser um algoritmo repetitivo e linear, é passível de paralelização com o objetivo de aumentar sua performance.

Foram vistos também as características básicas da utilização da tecnologia CUDA, como sua hierarquia de memórias e threads, assim como a percepção de sua eficácia para a paralelização e otimização de algoritmos utilizando esta tecnologia.

Com o desenvolvimento deste projeto, foi criada uma análise sobre a paralelização do algoritmo AES, como dificuldades e problemas encontrados relativos a sua performance e processo de paralelização.

Porém, apesar das modificações feitas no algoritmo, ainda existem alguns pontos plausíveis de melhorias, a ser estudados no futuro, como:

- Realização de testes com métodos otimizados para transferência e manipulação dos arquivos, como o uso de funções de *zero-copy d*e dados e *streaming d*e eventos (agrupando chamadas de escrita e leitura na memória global);
- Alocação temporária dos blocos de dados sendo processados, para a memória compartilhada da GPU ou registradores, reduzindo acessos a memória global ou local (também armazenada na memória global);
- Outra forma de otimização, pode se dar na forma de como os dados alocados na memória global são acessados, sendo alinhados em 256 *bytes*, é recomendado que grupos de *threads* manipulem seções da memória global de maneira sequencial, de 256 em 256 *bytes*;
- Possibilidade de expansão da paralelização do algoritmo para múltiplas GPUs em paralelo (função *cudaSetDevice()*);

Referências

Cook, Shade (2013), CUDA Programming, Morgan Kaufman, 1st edition.

Daemen, J.; Rijmen, (2002), V. The Design of Rijndael, Springer, 1st edition.

Erdelsky, Philip J., (2002), "Rijndael Encryption Algorithm", http://www.efgh.com/software/rijndael.htm, Maio 2014.

Luken, B. P.; Ouyang, M.; Desoky, A. H. (2009), "AES and DES Encryption with GPU", researchgate.net/publication/220922662_AES_and_DES_Encryption_with_GPU/file/d912f5062ed3d4c86b.pdf, Julho 2013.

National Institute Of Standards And Technology (2001), “Advanced Encryption Standard (AES)”, http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf, Julho 2013.

NVIDIA, (2014a), “CUDA LLVM Compiler”,

https://developer.nvidia.com/cuda-llvm-compiler, Maio 2014.

NVIDIA, (2014b), “Parallel Programing and Computing Platform”, http://www.nvidia.com/object/cuda_home_new.html, Junho 2014.

NVIDIA (2014c), “CUDA Toolkit Documentation”, http://docs.nvidia.com/cuda/cuda-c-programming-guide/#cuda-c-runtime, Maio 2014.

Smith, A. and Jones, B. (1999). On the complexity of computing. In *Advances in Computer Science*, pages 555–566. Publishing Press.

Zabala, (2013) E. Boston College, http://www.cs.bc.edu/~straubin/cs381-05/blockciphers/rijndael_ingles2004.swf, Junho 2013.