Leandro Moreira Barbosa

Implementação e avaliação de desempenho de algoritmo de criptografia em GPU para o FlexA

Monografia apresentada ao Departamento de Ciências de Computação e Estatística do Instituto de Biociências, Letras e Ciências Exatas da Universidade Estadual Paulista "Júlio de Mesquita Filho", como parte dos requisitos necessários para aprovação na disciplina Projeto Final.

São José do Rio Preto
2013

Leandro Moreira Barbosa

Implementação e avaliação de desempenho de algoritmo de criptografia em GPU para o FlexA

Monografia apresentada ao Departamento de Ciências de Computação e Estatística do Instituto de Biociências, Letras e Ciências Exatas da Universidade Estadual Paulista "Júlio de Mesquita Filho", como parte dos requisitos necessários para aprovação na disciplina Projeto Final.

Orientador:

Prof. Dr. Aleardo Manacero Jr.

São José do Rio Preto
2013

Leandro Moreira Barbosa

Implementação e avaliação de desempenho de algoritmo de criptografia em GPU para o FlexA

Monografia apresentada ao Departamento de Ciências de Computação e Estatística do Instituto de Biociências, Letras e Ciências Exatas da Universidade Estadual Paulista "Júlio de Mesquita Filho", como parte dos requisitos necessários para aprovação na disciplina Projeto Final.

Prof. Dr. Aleardo Manacero Jr. Leandro Moreira Barbosa

Banca Avaliadora:

Prof. Dr. Leandro Alves Neves

Prof. Dr. Norian Marranghello

São José do Rio Preto
2013

Aos meus pais Frank e Silvia,

ao meu irmão Evandro,

aos meus avós paternos Nei e Adelci,

e maternos Antônio e Neide.

"I love deadlines. I like the whooshing sound they make as they fly by."

-Douglas Adams

Agradecimentos

Primeiramente, agradeço à professora Renata por me aceitar no Grupo de Sistemas Paralelos e Distribuídos, por acreditar no meu potencial para realizar um trabalho de pesquisa, por chamar a minha atenção na sala de aula e me incentivar a fazer os exercícios.

Agradeço ao professor Aleardo por confiar no meu trabalho, por me orientar e me apontar a direção correta. Mais do que meu orientador e professor tenho prazer de poder chamá-lo de amigo. Como amigo partilhou suas experiências de vida e tirou dúvidas sobre assuntos além das obrigações como docente. Tudo isso enquanto tomávamos um cafézinho na cantinha, na xícara de cerâmica ou no copinho de plástico.

Agradeço aos meus pais por toda a confiança que depositaram em mim, por todo o carinho e amor. Mesmo que amar tenha significado, muitas vezes, chamar minha atenção e me colocar de volta no caminho certo. Sou grato por todas as condições favoráveis de estudo, por todo o esforço e dedicação que desempenharam esse papel tão importante.

Agradeço à minha grande amiga e meu amor, Marina. Agradeço por estar sempre do meu lado, por me incentivar, se interessar pelo meu trabalho, por me ajudar nos momentos mais difíceis e nunca duvidar do meu potencial.

Agradeço aos amigos do GSPD, pelo companheirismo e pelas risadas. Em especial agradeço ao Danilo, Diogo, Gabriel Saraiva, Gabriel Covello, Leonardo, Matheus e Rafael.

Obrigado a todos por me ajudarem a crescer e tornar-me uma pessoa melhor, por participarem desse ciclo de vida tão importante que foi a graduação. Guardarei a lembrança e as lições de todos vocês com muito carinho para sempre.

Resumo

Dado que a criptografia é necessária em sistemas de arquivos distribuídos para manter a privacidade dos usuários, este trabalho de pesquisa implementou o algoritmo de criptografia AES em ambiente CUDA para tirar vantagem do paralelismo das unidades de processamento gráfico (processadores altamente paralelos) para acelerar o processo de criptografia do sistema de arquivos distribuído FlexA. O módulo implementado conseguiu atingir um *speed-up* de até 3 vezes e um *throughput* de aproximadamente 2230 Mbit/s.

Palavras-chave: programação paralela, sistemas distribuídos, criptografia, CUDA, AES

Abstract

Given that criptography is necessary in distributed filesystems to ensure the privacy of the users, this research work implemented the criptography algorithm AES in CUDA platform in order to use the paralelism provided by the graphical processing units (highly paralell processors) to accelerate the criptography proccess of the distributed filesystem FlexA. The implemented module managed to achieve up to 3x speed-up with a throughput of about 2230 Mbit/s.

Keywords: parallel programming, distributed systems, criptography, CUDA, AES

Sumário

Lista de Figuras

Lista de Tabelas

Lista de Abreviaturas e Siglas

AES *Advanced Encryption Standard*.

CUDA *Compute Unified Device Architecture*.

FlexA *Flexible and Adaptable Distributed File System*.

GP-GPU *General-Purpose Computing on Graphics Processing Units*.
GPU *Graphics Processing Unit*.
GSPD *Grupo de Sistemas Paralelos e Distribuídos.*

NIST *National Institute of Standards and Technology*.

SM *Streamming Multiprocessor*.
SSD *Solid State Drive*.

Capítulo 1

Introdução

1.1 Motivação

Um sistema distribuído é um conjunto de computadores independentes que se apresenta aos usuários como um sistema único e coerente. Desta forma, é intrínseco ao sistema que as diferenças entre seus componentes (tais como as comunicações que ocorrem) fiquem, em grande parte, ocultas ao usuário. Como esses sistemas utilizam as redes de computadores para se comunicar, torna-se fundamental a utilização de criptografia para a proteção da troca de mensagens entre os componentes do sistema (TANENBAUM; STEEN, 2006).

A criptografia possibilita a proteção contra vazamento de dados, modificação de mensagens e inserção de mensagens por um intruso. Especificamente, o uso da criptografia é imprescindível para a proteção dos arquivos dos usuários no sistema de arquivos distribuídos. Usado em praticamente todos os tipos de comunicação de sistemas distribuídos, o processo de criptografia pode se tornar custoso e ocupar uma parte considerável do poder de processamento da CPU dos computadores dos sistemas (TANENBAUM; STEEN, 2006).

Muitos dos algoritmos de criptografia atuais realizam fazendo a criptografia de um bloco de dados de tamanho fixo. A criptografia feita desta forma não impede que arquivos de tamanhos arbitrários sejam cifrados. Para isto, o arquivo é subdividido em blocos, em seguida, esses blocos são cifrados individualmente e concatenados de acordo com um dos cinco modos de criptografia especificados pelo *National Institute of Standards and Technology* (NIST) (DWORKIN, 2001).

O processo de criptografia dos arquivos do usuário é uma tarefa paralelizável, a qual pode ser acelerada utilizando-se *General-Purpose Computing on Graphics Processing Units* (GP-GPU), isto é, uso da *Graphics Processing Unit* (GPU) para executar

tarefas tradicionalmente executadas em CPU. O uso dessa técnica também implica na liberação da CPU para realizar outras tarefas do sistema, o que é interessante para sistemas de arquivos distribuídos, como o *Flexible and Adaptable Distributed File System* (FlexA) (FERNANDES, 2012), desenvolvido nos laboratórios do Grupo de Sistemas Paralelos e Distribuídos (GSPD)[1].

1.2 Objetivos

Os objetivos desse trabalho foram,

- avaliar o desempenho do algoritmo de criptografia utilizado atualmente no FlexA (AES);
- implementar o algoritmo AES na plataforma CUDA;
- integrar essa implementação no FlexA
- avaliar os possíveis ganhos de desempenho da versão paralela em relação à versão sequencial.

1.3 Organização do Texto

O trabalho realizado é descrito em quatro capítulos. No Capítulo 2, o sistema de arquivos distribuídos FlexA é descrito brevemente, a plataforma *Compute Unified Device Architecture* (CUDA), organização do *hardware* em que ela se apoia e, por fim, o algoritmo de criptografia *Advanced Encryption Standard* (AES) são apresentados com o intuito de familiarizar o leitor com as tecnologias e conceitos envolvidos neste projeto. No Capítulo 3, a implementação realizada, a integração com o FlexA e outros detalhes são descritos. No Capítulo 4, são descritos o ambiente de testes, os testes realizados e seus resultados. As implicações dos resultados nos ambientes nos quais o FlexA está inserido também são discutidos. Por fim, no Capítulo 5, são apresentadas as conclusões e as implicações do projeto realizado.

[1] GSPD:http://www.dcce.ibilce.unesp.br/spd/

Capítulo 2

Fundamentação teórica

Nesse capítulo é apresentado o funcionamento básico do sistema de arquivos distribuído FlexA, do algoritmo de criptografia AES e da plataforma CUDA.

2.1 FlexA

O FlexA é um sistema de arquivos distribuído que tem como principais objetivos:

- Permitir que a estação cliente também possa se tornar um servidor do sistema;
- Suporte a *hardware off-the-shelf*, isto é, *hardware* disponível e acessível;
- Facilidade no uso de criptografia;
- Tolerância a falhas utilizando-se redundância;
- Uso de *caches* para reduzir a latência de acesso a arquivos previamente abertos.

No FlexA, há três grupos de computadores semanticamente separados: computador do usuário, servidores primários (grupo de escrita) e servidores secundários (grupo de réplicas). O papel dos servidores primários é receber as conexões dos usuários e administrar os arquivos e as informações sobre os mesmos; dos servidores secundários é servir como *backup* e para balanceamento de carga, ou seja, quando os servidores primários forem sobrecarregados, os usuários podem requisitar os arquivos dos servidores secundários. Na versão atual, utilizam-se três servidores primários e um número arbitrário de servidores secundários (Figura 2.1).

Quando um cliente solicita um arquivo do sistema, é necessário comunicar-se com os três servidores primários para fazer a requisição das porções do arquivo desejado. Para que isto ocorra, os módulos coletor, sincronizador e comunicador são os

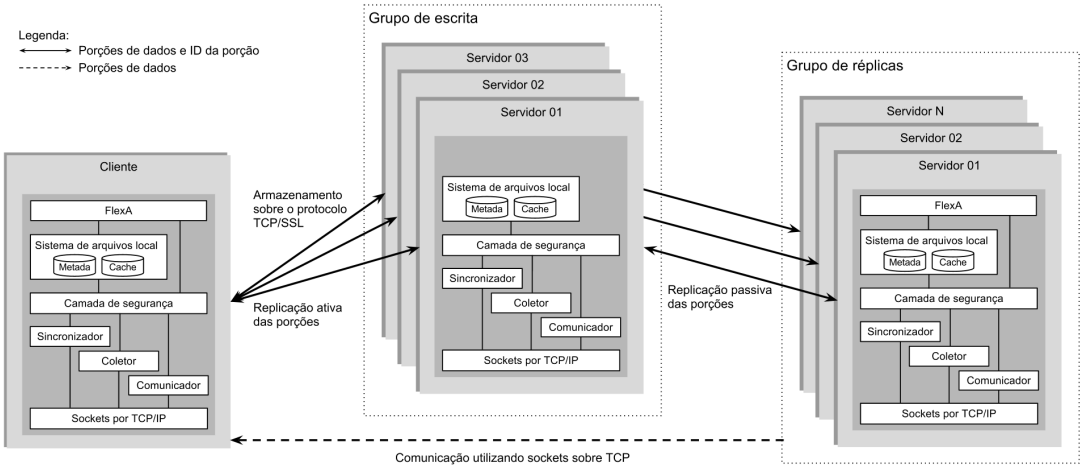

Figura 2.1: Visão geral do FlexA (Fonte: (FERNANDES, 2012))

responsáveis por receber conexões e realizar as operações, informar os outros componentes do sistema sobre mudanças nos dados e buscar componentes ativos do sistema, respectivamente.

Ao ser enviado para um servidor, cada arquivo é dividido no cliente em três partes, denominadas *porções*. Cada servidor primário recebe duas dessas porções, de maneira que se um servidor estiver indisponível, seja possível recuperar o arquivo utilizando as porções dos outros dois remanescentes (Figura 2.2). As porções são replicadas passivamente para os servidores secundários (réplicas).

O controle de acesso do sistema é feito utilizando-se do conceito de função *hash* criptográfica. Essa função é construída de forma que ela não seja inversível, isto é, conhecida a imagem, não é possível conhecer o domínio.

Quando um novo arquivo é criado no sistema, é gerado um número aleatório de uma fonte criptograficamente segura. A partir desse desse número, é gerada uma chave de verificação utilizando um *hash sha384* (FIPS-180-4, 2012), chamada ***verify key***, que é utilizada para a identificação de um arquivo no sistema. Do número aleatório também é gerado um *hash sha256* chamado ***write key*** e, a partir deste, é gerado um outro *hash* chamado ***valid write*** que é utilizado para fazer o controle de escrita do arquivo (Figura 2.3). Deste modo, é possível que o servidor realize autenticação sem possuir a chave de criptografia do arquivo, já que é computacionalmente inviável realizar a inversão de um *hash* e o servidor possui apenas os *hashes*, *verify key* e *valid write*.

Para obter um arquivo do sistema é necessário que o usuário forneça a *verify key*. A partir dessa chave o sistema encontra o arquivo solicitado e envia ao cliente.

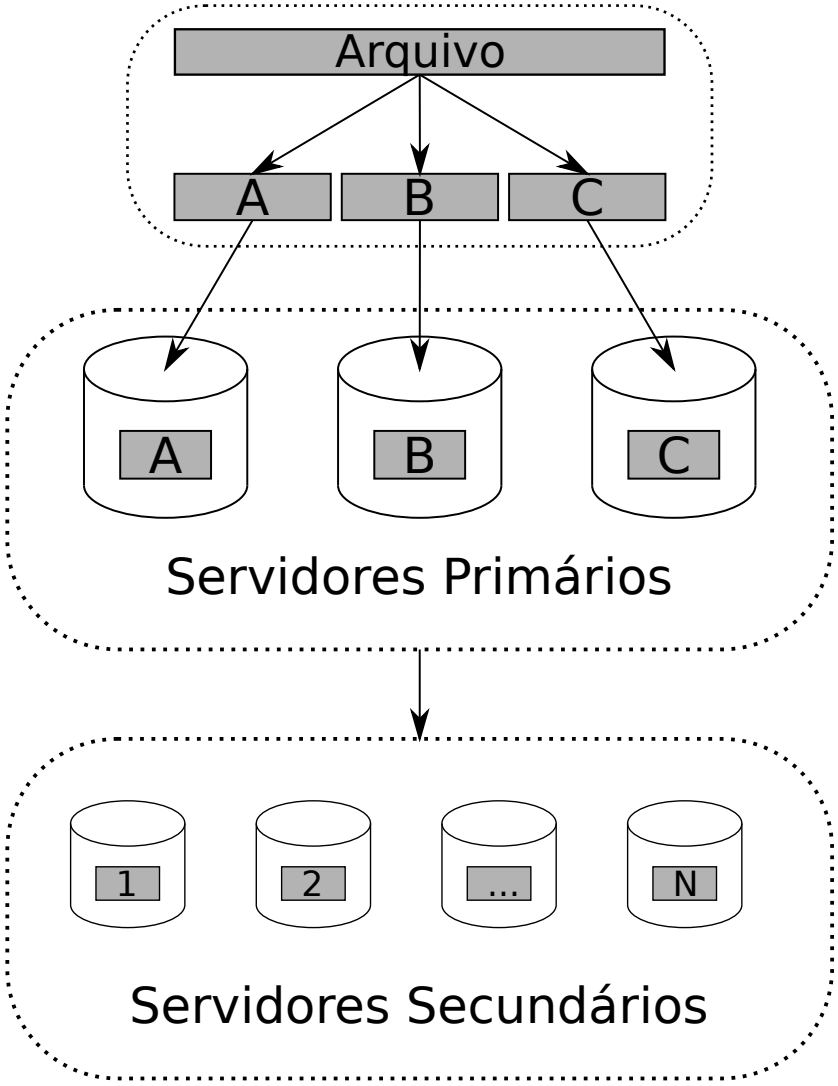

Figura 2.2: Redundância entre servidores primários

Tendo em vista que o sistema de arquivos é distribuído, há a necessidade de assegurar a privacidade dos usuários. Para isto, cada arquivo é cifrado utilizando o algoritmo de criptografia AES no cliente, utilizando como chave criptográfica um *hash* da *write key*, chamado **read key**.

2.2 AES

O algoritmo de criptografia AES consiste em alocar um bloco de 16 bytes de texto em uma matriz chamada de `state` e, em seguida, modificar essa matriz de acordo com as várias transformações especificadas pelo algoritmo até se obter a forma final da matriz — o texto cifrado. O algoritmo realiza quatro operações: `AddRoundKey`,

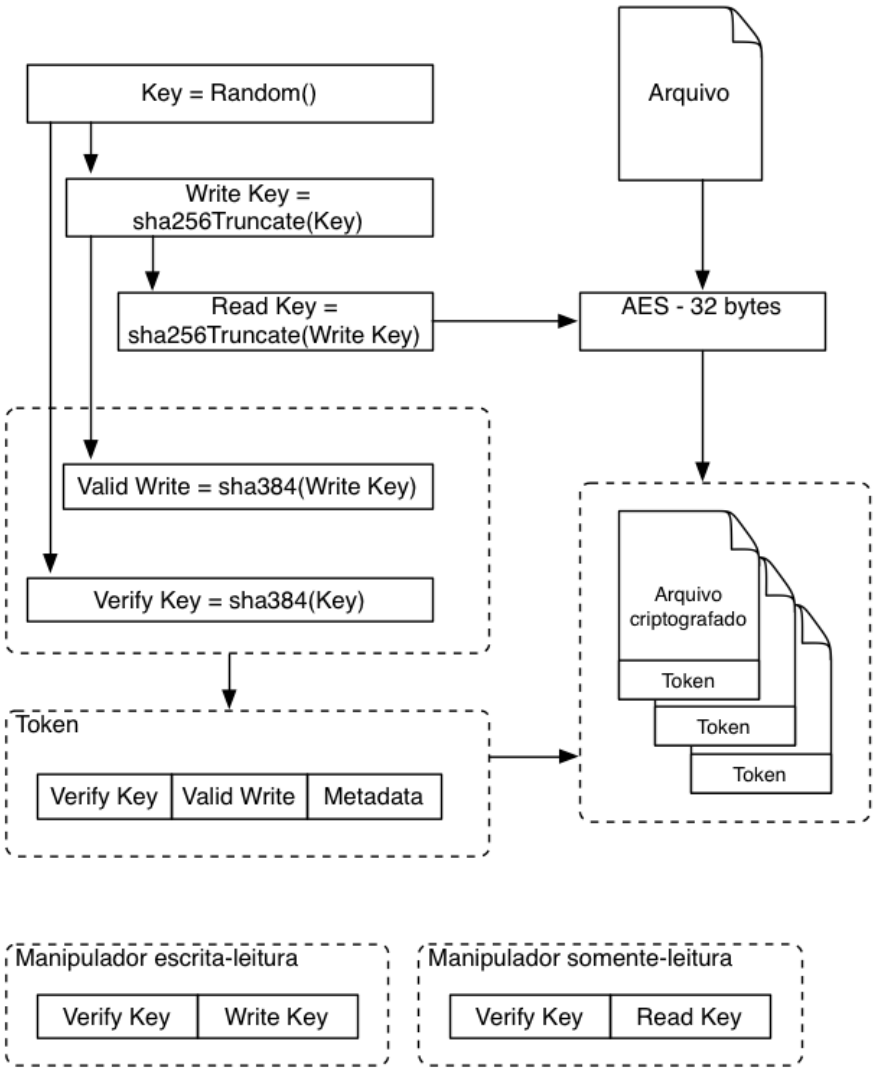

Figura 2.3: Controle de acesso no FlexA (Fonte: (FERNANDES, 2012))

`SubBytes`, `ShiftRows` e `MixColumns`. Essas operações são realizadas repetidas vezes nos chamados **rounds** (ou ciclos) do algoritmo. Para chaves de tamanho 128, 192 e 256 são realizados, respectivamente, 10, 12, 14 *rounds*.

- `AddRoundKey`: OU-exclusivo da chave com o estado (Figura 2.4);
- `SubBytes`: Usa uma tabela de substituição para prover não-linearidade a cifra. Previne ataques às propriedades lineares do resto do algoritmo (Figura 2.5);
- `ShiftRows`: Reorganiza as linhas do estado. Essa operação faz com que o algoritmo não seja linearmente independente (Figura 2.6);
- `MixColumns`: Aplica uma transformação linear ao estado (Figura 2.7).

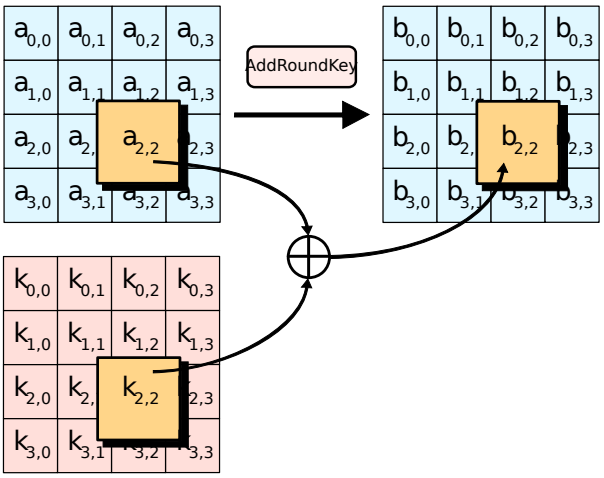

Figura 2.4: Operação AddRoundKeys realiza OU-exclusivo

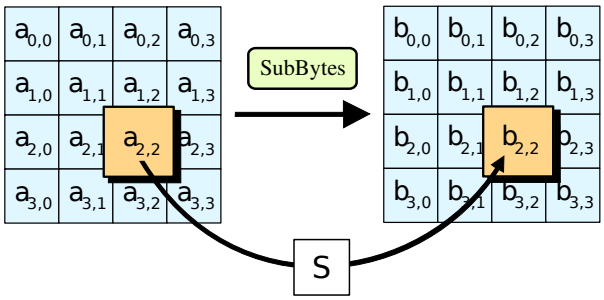

Figura 2.5: Operação SubBytes realiza uma substituição byte a byte

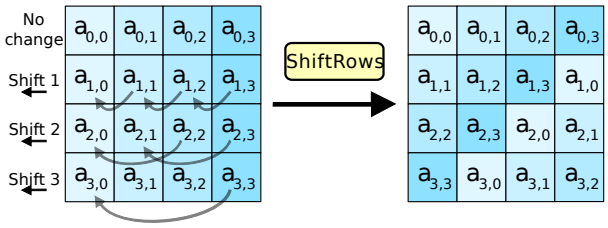

Figura 2.6: Operação ShiftRows reorganiza as linhas

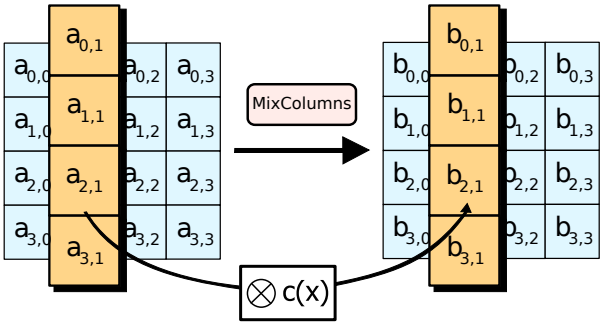

Figura 2.7: Operação MixColumns realiza uma transformação linear

No início do algoritmo, é definido o número de *rounds* a serem realizados baseado no tamanho da chave criptográfica: 10 para chaves de 128 *bits*, 12 para 192 *bits* e 14 para 256 *bits*. Em seguida, realiza-se uma operação `AddRoundKey` e as operações `SubBytes`, `ShiftRows`, `MixColumns` e `AddRoundKey` formam o corpo de um *round* de operações. Esse *round* é executado `NR` vezes. Por fim, é executado mais um *round* com exceção da operação `MixColumns` (Figura 2.8).

```
NR ← 10                                  ▷ 12 ou 14, para chaves de 192 ou 256 bits
i ← 0
ADDROUNDKEY(state)
while i < NR do
    SUBBYTES(state)
    SHIFTROWS(state)
    MIXCOLUMNS(state)
    ADDROUNDKEY(state)
    i ← i + 1
end while
SUBBYTES(state)
SHIFTROWS(state)
ADDROUNDKEY(state)
```

Figura 2.8: Algoritmo AES

2.3 Modo *Counter*

Para a cifragem de um texto de tamanho arbitrário usando o algoritmo AES, é necessário dividir o texto em blocos de 16 bytes. À maneira como esses blocos são utilizados e gerados, dá-se o nome de "modo de criptografia". Para esse trabalho, foi selecionado o modo *Counter* (CTR), devido à sua independencia entre os blocos e, portanto, com grande potencial de paralelização.

Nesse modo, cada bloco é inicializado com um número aleatório chamado **`nonce`**, que preenche metade do bloco. Para a outra metade, são gerados blocos consecutivos utilizando uma função contadora. Comumente, utiliza-se a própria função identidade para realizar a contagem, isto é, $f(x) = x$, onde x é o identificador do bloco. Dessa maneira, os blocos são gerados a partir de números naturais. Os blocos são cifrados e utilizados em uma operação OU-exclusivo com os blocos de texto, resultando nos blocos de textos cifrados (Figura 2.9).

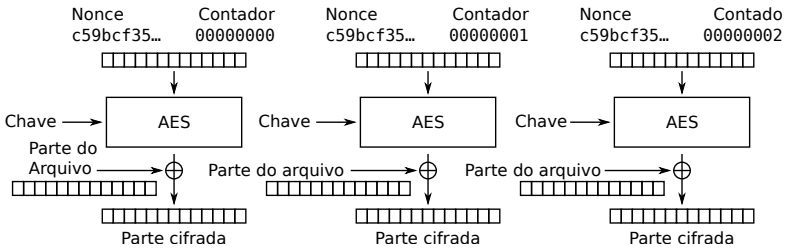

Figura 2.9: Diagrama do modo CTR

2.4 CUDA

CUDA é uma plataforma desenvolvida pela NVIDIA para facilitar o uso do paradigma GP-GPU para as suas placas de vídeo. Ao desenvolver nessa plataforma é preciso levar em consideração as diferenças entre CPU e GPU, incluindo, diferenças na organização dos elementos de processamento, escalonamento das *threads* e hierarquia de memória.

Uma das diferenças entre CPU e GPU é a maneira como os transistores são utilizados na placa. A CPU aloca mais transistores para serem usados como *cache* e com a unidade de controle do que a GPU—esta tem o foco maior nas unidades lógicas e aritméticas. Outra diferença é que a CPU também tem o papel de executar o sistema o sistema operacional que vai controlar os outros dispositivos. A GPU, no entanto, não precisar desempenhar esta função

Para organizar a arquitetura e facilitar a programação, as *threads* são agrupados em blocos, e os blocos por sua vez são agrupados em *grids* (Figura 2.10). Cada bloco de *threads* é executado em um *Streamming Multiprocessor (SM)*. O SM contém 32 **núcleos CUDA** (Figura 2.11). São esses núcleos os responsáveis por todos os cálculos da GPU.

As *threads* são executados paralelamente em grupos de 32, a este grupo se dá o nome de **warp**. Cada SM possui apenas uma unidade de controle (*warp scheduler*) divida em duas partes. Cada parte é responsável por controlar 16 unidades lógicas e aritméticas. A execução segue o modelo de execução *Single Instruction, Multiple Data* (SIMD), isto é, a mesma intrução é enviada para todas as ULAs, que executam-na em dados diferentes.

Para que os dados cheguem até os núcleos, os dados devem ser copiados para a memória global da GPU de maneira explícita. É dessa memória que os dados serão acessados pelas *threads*. A memória global, no entanto, é uma memória de alta latência e o seu uso não é recomendável para compartilhar dados entre *threads* (NVIDIA,

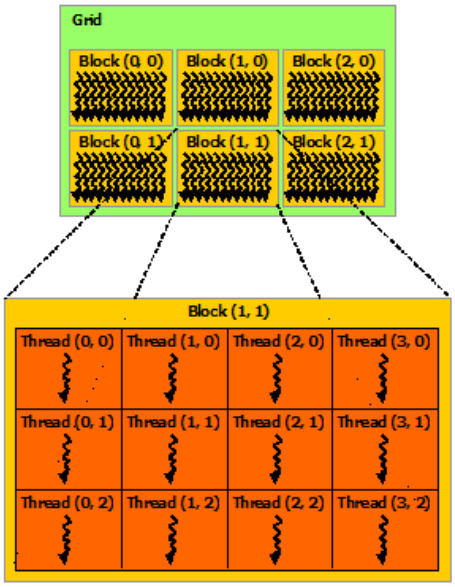

Figura 2.10: Organização das *threads* de uma GPU (Fonte: (NVIDIA, 2013b)

2013a). Para isso, cada SM contém 64Kb de memória, dos quais, 48Kb podem ser usados como memória compartilhada. A única restrição dessa memória é que apenas *threads* do mesmo bloco podem utilizá-la para compartilhar dados entre si (cada bloco executa em um SM apenas).

O gerenciamento da memória da placa é realizada de maneira explícitas. A sua alocação, a cópia dos dados—tanto da memória principal (RAM) para a GPU (cópia dos dados), quanto da GPU para a memória principal (cópia dos resultados)—e a utilização da memória compartilhada, devem ser feitas explicitamente.

2.5 Trabalhos Relacionados

Algumas implementações do algoritmo AES em CUDA existem na literatura. Em todas essas implementações foram utilizadas tabelas de mapeamento para realizar a cifragem. Desta forma, não é necessário realizar cálculos para cifrar um arquivo, apenas é necessário localizar o valor adequado para substituição dos *bytes* do arquivo. Com exceção de Liu, que utilizou apenas uma tabela—sendo necessário a derivação das outras tabelas a partir dessa—as outras implementações utilizaram quatro tabelas. Observa-se que todas essas placas utilizadas possuem um desempenho similar. A Tabela 2.1 aprensenta os resultados obtidos pelos autores dos trabalhos (MANAVSKI, 2007) (LIU et al., 2009) (MEI et al., 2010) (NISHIKAWA et al., 2011).

É necessário notar que essas tabelas de mapeamento devem ser copiadas para a memória da GPU antes de iniciar o algoritmo da cifragem. Nos trabalhos considerados,

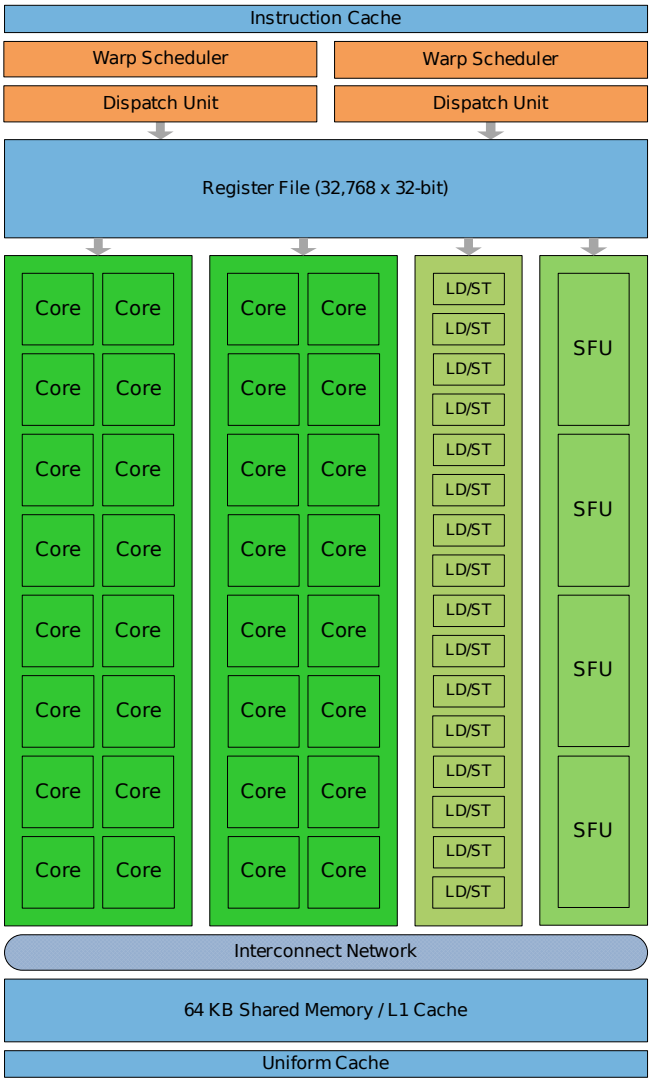

Figura 2.11: Organização de um *streamming multiprocessor*

a utilização da memória compartilhada dos SM para armazenar as tabelas apresentou melhor desempenho.

Tabela 2.1: Desempenho de trabalhos da literatura envolvendo CUDA e AES

Autor	*Throughput* (Mbit/s)	Tabelas	GPU	*Cores*	*Shader Clock* (MHz)
Manavski	6650	4	8800 GTX	128	1350
Liu et al.	4200	1	9800 GTX	128	1688
Mei et al.	6400	4	9200M GS	8	1300
Nishikawa et al.	6250	4	GTX 285	240	1476

Capítulo 3

Implementação e validação do algoritmo

Nesse capítulo será apresentado como o algoritmo AES foi implementado na plataforma CUDA, quais as decisões tomadas para os detalhes de implementação e como o módulo foi desenvolvido para ser integrado ao FlexA. Além disso, as otimizações realizadas também são expostas.

3.1 Implementação do algoritmo

Neste projeto, o algoritmo implementado utiliza chaves de tamanho 128 *bits* (tamanho utilizado pela implementação original do FlexA). Não foi usada nenhuma tabela de mapeamento para o algoritmo, com exceção às tabelas de substituição das funções `SubBytes` e `KeyExpansion`, ambas descritas em (FIPS-197, 2001). As tabelas não foram utilizadas pois a implementação original não as utilizam, dessa forma, o algoritmo é implementado de maneira mais próxima para permitir comparações coerentes entre o tempo de execução dos algoritmos sequencial e paralelo.

A implementação foi feita na plataforma CUDA (NVIDIA, 2013c), com granularidade adotada de 1 bloco de texto (16 *bytes*) para cada *thread* de execução da GPU. Dessa forma, nenhuma sincronização é necessária, fazendo com que cada *thread* seja um trabalhador do modelo *bag of tasks*. Além disso, as GPUs comumente têm muitos registradores disponíveis para cada *warp* (grupo de 32 *threads* em execução em paralelo no mesmo núcleo da GPU), o que acelera a execução pois os dados são armazenados nos registradores e não na memória compartilhada. Como exemplo, no *hardware* testado, o compilador nvcc disponibilizou em torno de 60 registradores para cada *thread* de um *warp*.

Para a cifragem de um arquivo de tamanho arbitrátio, o modo de criptografia

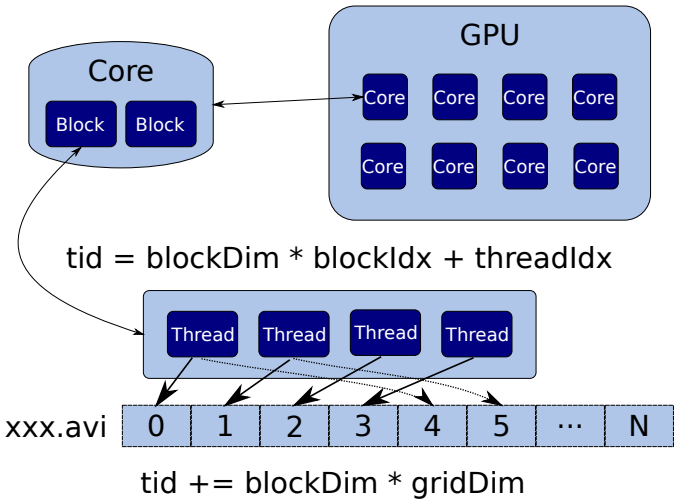

Figura 3.1: Diagrama de mapeamento de *threads* e blocos para a cifragem de um arquivo chamado "xxx.avi".

utilizado é o CTR (*Counter*) devido seu grande potencial de paralelização, porque os blocos podem ser cifrados independemente uns dos outros. Os contadores para esse modo de criptografia são gerados na própria GPU, de maneira que cada *thread* é responsável em gerar o bloco contador respectivo do seu bloco de texto.

Inicialmente, o arquivo a ser cifrado é carregado do disco para a memória RAM e depois copiado para a memória global da GPU. Em seguida, os blocos contadores são gerados em cada *thread*: os 8 primeiros *bytes* dos blocos contadores são inicializados com o `nonce`; os 8 *bytes* finais são inicializados com o `tid` particular da *thread* convertido para base 256, isto é, cada "dígito" do `tid` vai ser representado por 1 *byte*.

Esses blocos são armazenados diretamente nos registradores da GPU, em um vetor chamado `state`. Este vetor simula uma matriz de estados sobre os quais tranformações sucessivas do algoritmo AES vão gerar o bloco de texto cifrado.

Cada *thread* calcula um número de idenficação inicial único dentre todas *threads* em execução na GPU. Esse número, chamado `tid` é calculado com base no índice do bloco em que ela está inserida (*blockIdx*), na dimensão do bloco (*blockDim*, isto é, quantas *threads* ele possui ao todo, e no índice da *thread* dentro do seu bloco (*threadIdx*) (Equação 3.1).

$$tid = blockIdx * blockDim + threadIdx \tag{3.1}$$

A utilização desse `tid` permite mapear cada *thread* para um bloco de 16 *bytes* do texto de maneira única (Figura 3.1). O identificador é único por *thread*, portanto,

nenhuma *thread* vai sobreescrever os resultados da outra.

Após a inicialização do `state`, cada *thread* aplica o algoritmo AES sobre este vetor. Então, cada *thread* realiza a operação XOR (OU-exclusivo) entre o vetor `state` e o respectivo bloco de texto localizado na memória global da GPU. Obtendo assim, um bloco de texto cifrado.

Finalizado o processo de criptografia de um bloco, cada *thread* altera o seu `tid` para ser mapeada para o próximo bloco de texto ainda não cifrado. Essa alteração consiste em somar o produto da dimensão de um bloco (*blockDim*) com a dimensão de um *grid* (*gridDim*), isto é, quantidade total de blocos dentro do *grid* (Equação 3.2).

$$tid += blockDim * gridDim \tag{3.2}$$

Ao final de todo o processo, o arquivo cifrado é copiado de volta para a memória RAM e o ponteiro contendo endereço do arquivo cifrado na memória é devolvido para a função que chamou esta rotina.

3.2 Desenvolvimento do módulo para utilização no FlexA

Como o algoritmo descrito na Seção 3.1 é implementado na plataforma CUDA e o FlexA é implementado na linguagem Python, foi necessário o desenvolvimento de um módulo em Python para disponibilizar os recursos do novo algoritmo desenvolvido em GPU para o sistema de arquivos.

Nesse módulo, chamado `AESCuda`, o usuário conta com duas funções principais: `encrypt()` e `decrypt()`. Os argumentos para ambas as funções são:

- `input`: um vetor que contém o texto a ser cifrado (pode ser lido de um arquivo);
- `key`: chave de criptografia;
- `nonce`: um vetor aleatório de 8 *bytes*;
- `ctr_offset`: parâmetro opcional para indicar o número inicial do contador. Usado para quando o arquivo não cabe inteiro na memória da GPU;
- `gpup`: parâmetro opcional para configurar o número de blocos e *threads* a serem utilizados pela GPU.

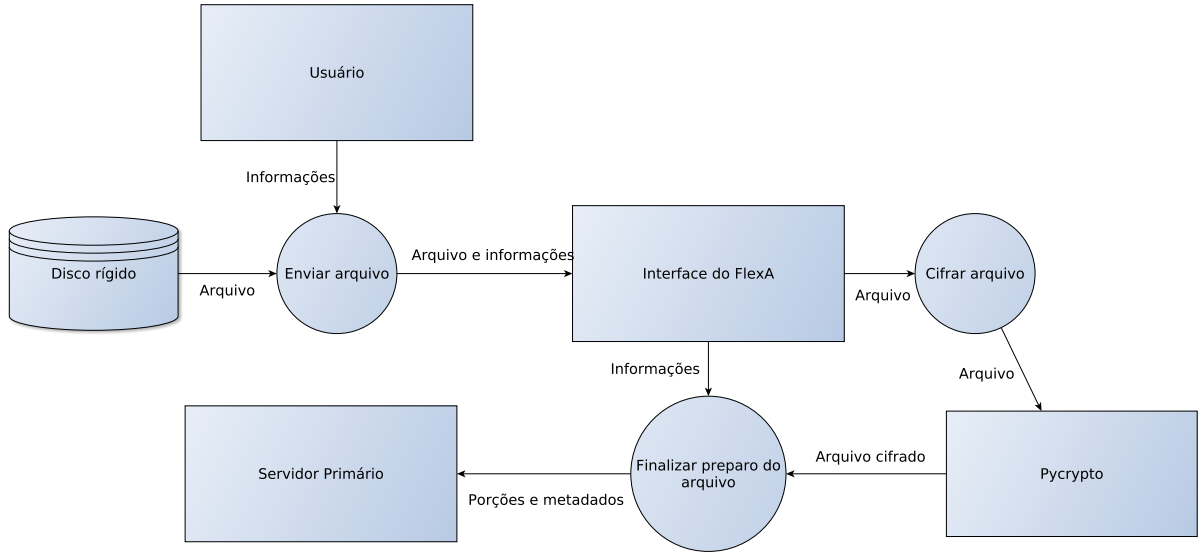

Figura 3.2: Funcionamento do FlexA antes da implementação do AESCuda.

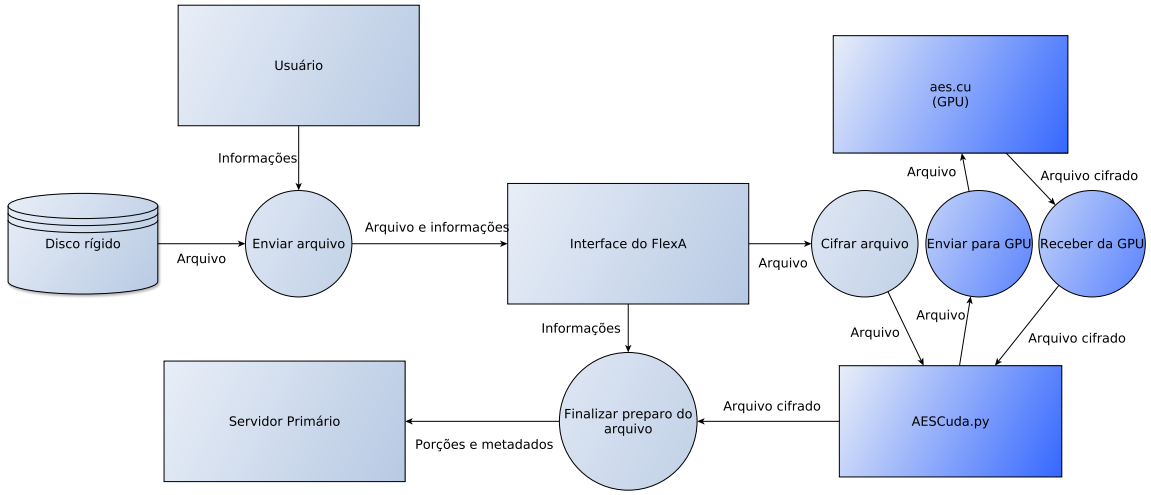

Figura 3.3: Funcionamento do FlexA após a implementação do AESCuda.

Antes da implementação do módulo AESCuda, o FlexA fazia a cifragem de um arquivo a ser enviado utilizando o módulo Pycrypto (Figura 3.2). Após a implementação do módulo, o FlexA passa a utilizá-lo para fazer o uso da implementação do algoritmo AES em CUDA. Deste modo, o arquivo é cifrado utilizando a GPU, como visto na Figura 3.3.

3.3 Otimização da implementação

A implementação do algoritmo seguiu à abordagem indicada pelo NIST (FIPS-197, 2001). A técnia de desenrolamento de laços foi utilizada para acelerar o algoritmo.

A técnica de desenrolamento de laços consiste em alterar um laço para se evitar a atualização constante dos índices. Se o laço for repetido poucas vezes, pode-se evitar a utilização dos índices por completo. O compilador *nvcc* (disponibilizado pela NVIDIA) possui mecanismos para o desenrolamento automático de laços. Em um laço comum, a cada iteração é necessário atualizar o índice e verificar se o laço deve ser executado novamente (Figura 3.4). Em um laço desenrolado, o índice é atualizado menos constantemente, por exemplo, a cada quatro iterações (Figura 3.5).

```
for (int i = 0; i < N; i++) {
  doSomething (&a[i]);
}
```

Figura 3.4: Laço comum

```
for (int i = 0; i < N; i += 4) {
  doSomething (&a[i+0]);
  doSomething (&a[i+1]);
  doSomething (&a[i+2]);
  doSomething (&a[i+3]);
}
```

Figura 3.5: Laço desenrolado

3.4 Validação

A validação do algoritmo foi feita com base nos exemplos de (FIPS-197, 2001) e com base na biblioteca PyCrypto (LITZENBERGER, 2013), utilizada originalmente no FlexA. Os arquivos utilizados nos testes foram cifrados utilizando as duas implementações e foram comparados *byte* a *byte*, para certificar que o algoritmo produz as mesmas saídas produzidas originalmente.

Capítulo 4

Testes, resultados e avaliação

Neste capítulo são apresentados o ambientes de testes e os resultados obtidos.

4.1 Ambiente de testes e parâmetros utilizados

Os testes de desempenho foram conduzidos em *laptop* com sistema operacional Arch Linux (kernel 3.11.5), compilador gcc 4.8.1 e *NVIDIA CUDA Toolkit* versão 5.5.22. O *driver* de vídeo utilizado foi o *driver* proprietário da NVIDIA versão 325.15. O *laptop* possui processador Intel Core i7–740QM@1.73GhHz e com placa de vídeo Nvidia GeForce GTX 460M. O disco rígido utilizado possui uma taxa de leitura sequencial de aproximadamente 90 MB/s.

Para os testes foram gerados arquivos com conteúdo aleatório de tamanhos de 1 até 512 MB. Foi medido o tempo para executar as funções de criptografia do PyCrypto e do módulo AESCuda. Como o módulo AESCuda permite a configuração de blocos e *threads* da GPU, foram feitos testes com tamanhos de bloco de 64 até 512 elementos, contendo de 64 até 512 *threads* cada. Os tempos de leitura do disco não foram contabilizados em nenhum caso.

Para amenizar uma possível influência de outros processos e de condições do sistema operacional, cada teste foi realizado 10 vezes. Os tempos utilizados nos gráficos são as médias dessas execuções. Além disso, também foram calculados os desvios padrões.

4.2 Resultados e avaliação

Primeiramente foram realizados os testes para identificar quais são as melhores configurações de blocos e *threads* CUDA no *hardware* testado. Não houve diferenças

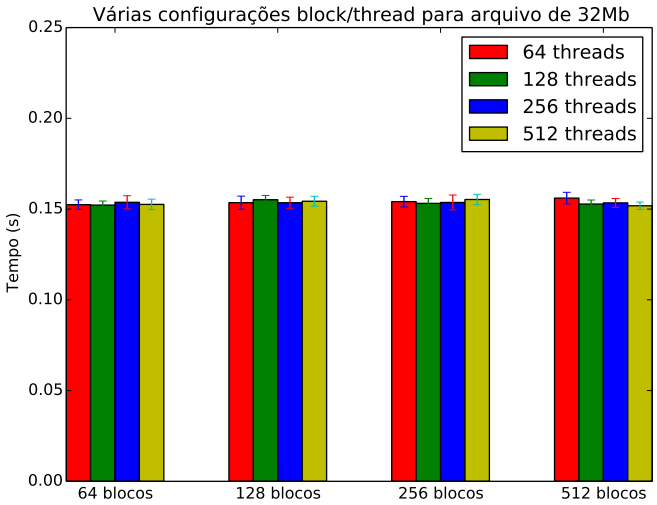

Figura 4.1: Nenhuma diferença significativa entre as várias configurações de blocos e *threads*

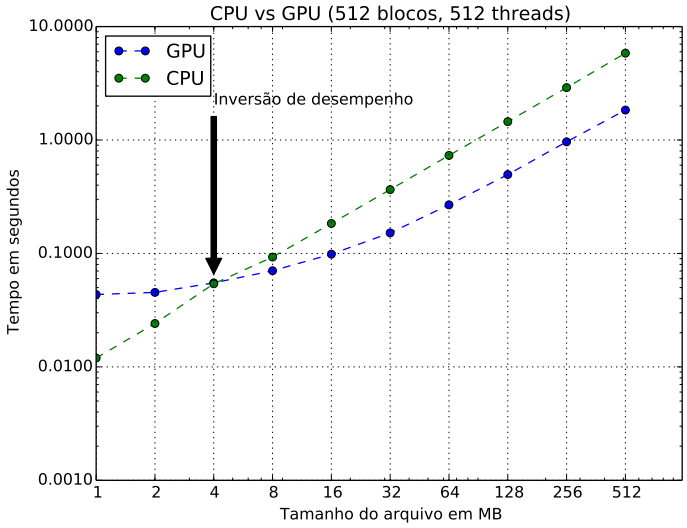

Figura 4.2: Comparação entre tempos de execução da versão sequencial e paralela

significativas entre as configurações testadas (Figura 4.1). Para os testes subsequentes, utilizou-se 512 blocos e 512 *threads*.

Observa-se na Figura 4.2 que o desempenho do módulo AESCuda se torna melhor a partir do tamanho de arquivo de 4 MB, escalando linearmente conforme o tamanho do arquivo aumenta. Observa-se que a diferença não é tão significativa para arquivos pequenos—diferença de menos de 0.09 ms—entretanto, para arquivos maiores a diferença se torna mais significativa. Por exemplo, para os arquivos de 256 MB a diferença de desempenho foi de aproximadamente 4 segundos.

Tabela 4.1: Comparação com os desempenhos da literatura

Autor	*Throughput* (Mbit/s)	Tabelas	GPU	*Cores*	*Shader Clock* (MHz)
AESCuda	2230	0	GTX 460M	192	1350
Manavski	6650	4	8800 GTX	128	1350
Liu et al.	4200	1	9800 GTX	128	1688
Mei	6400	4	9200M GS	8	1300
Nishikawa	6250	4	GTX 285	240	1476

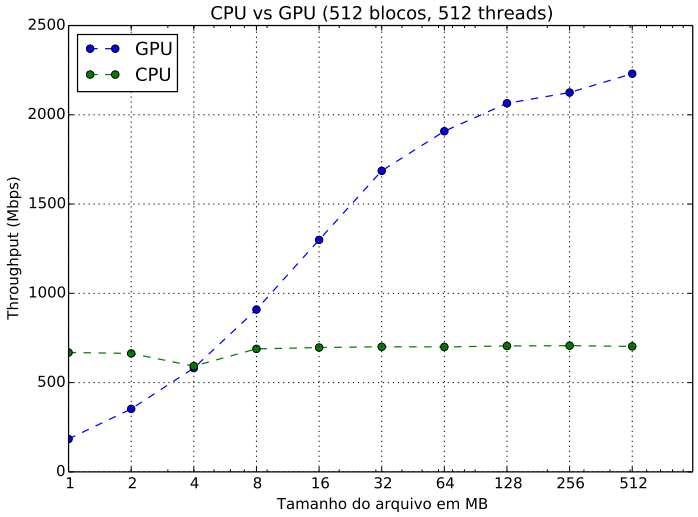

Figura 4.3: Comparação entre *throughput* da versão sequencial e paralela.

Em um gráfico de *throughput* (Figura 4.3), é possível ver que a taxa de velocidade da cifragem do algoritmo sequencial se aproxima de 700 Mbit/s (87 MB/s), ou seja, taxa muito próxima da taxa de transferência de um disco rígido SATA comum. Com o surgimento de novas tecnologias de armazenamento, no entanto, o algoritmo se torna um fator limitante para o desempenho do sistema. Um dispositivo de armazenamento secundário do tipo *Solid State Drive* (SSD), por exemplo, pode obter taxas de leituras de 2240 Mbit/s (280 MB/s) (RYAN, 2013).

Na versão paralela do algoritmo desenvolvida neste projeto, o desempenho chegou a uma taxa de 2230 Mbit/s (278 MB/s), que seria suficiente para saturar a velocidade do melhor dispositivo SSD apresentado em (RYAN, 2013). Essa velocidade é suficiente para tirar proveito dos roteadores sem fio atuais que possuem taxas de transferência de até 1440 Mbit/s (180 MB/s) (WINKLE, 2013). Na Tabela 4.1 mostra-se uma comparação com os resultados encontrados na literatura.

Pelos resultados obtidos neste trabalho, é possível afirmar que os ganhos de desempenho com a implementação paralela foram significativos, chegando a um *spe-*

edup superior a 3.0 para arquivos de tamanho superior a 512 MB. Para esse tamanho de arquivo, obteve-se uma *throughput* de aproximadamente 2230 Mbit/s.

Para os arquivos menores do que 4 MB, nos quais o desempenho foi abaixo da versão sequencial, o tempo gasto com a computação foi menor do que o tempo gasto para a transferência dos dados entre as memórias principal e da GPU. No caso da versão sequencial, esse tempo não existe, pois a CPU trabalha diretamente com a memória principal do computador. O tempo gasto com a transferência entre memórias foi determinante para o desempenho nessa faixa de tamanho de arquivo e essa característica deve ser considerada sempre que a utilização do paradigma GP-GPU for considerada.

Observa-se que isso não ocorre para o caso de a CPU incluir uma GPU dentro do encapsulamento do processador, como é o caso dos modelos de processadores recentes da Intel e da AMD. Nesses casos não é necessário transferir os dados da memória principal para a memória da GPU.

Capítulo 5

Conclusões

Com os resultados obtidos, foi possível observar um tempo de cifragem com um *speed-up* de até três vezes para arquivos de 512 MB, com *throughput* de aproximadamente 2230 Mbps. Com essa velocidade, apesar de estar abaixo dos resultados encontrados na literatura, é possível utilizar a banda completa de um roteador sem fio doméstico *Gigabit Ethernet* utilizando frequencia 5GHz (WINKLE, 2013).

O paradigma de programação GP-GPU mostrou-se eficaz em acelerar o processo de criptografia realizada pelo cliente no FlexA. Em configurações de redes comumente encontradas em laboratórios e domicílios, o processo de criptografia provavelmente não será o fator limitante para a utilização do sistema, desde que a GPU seja usada para realizar essa operação. Portanto, este projeto possibilita que a criptografia seja utilizada sem restrições de desempenho nesses ambientes.

5.1 Dificuldades encontradas

Visto que o paradigma de programação paralela apresenta uma visão diferente da programação sequencial e ainda mais no caso da utilização de GP-GPU — devido às características intrínsicas da GPU, tais como hierarquia de memória e organização do *hardware* — a principal dificuldade encontrada no projeto foi realizar o mapeamento do algoritmo sequencial para a utilização do novo paradigma.

5.2 Trabalhos futuros

Como sugestão para trabalhos futuros, uma possibilidade é implementar o algoritmo utilizando as tabelas de mapeamento. Além disso, a implementação atual não realiza paralelismo entre a cópia entre memórias da GPU e RAM e o processamento da cifra, o que pode ser alterado para obter um desempenho melhor. Com essas duas

modificações acredita-se possível alcançar um ganho de desempenho de até três vezes o ganho atual, tornando a implementação tão boa quanto as apresentadas na literatura.

Referências Bibliográficas

DWORKIN, M. *Recommendation for Block Cipher Modes of Operation*. 2001.

FERNANDES, S. E. *Sistema de Arquivos Distribuído Flexível e Adaptável*. Dissertação (Mestrado) — Universidade Estadual Paulista "Júlio de Mesquita Filho", 2012.

FIPS-180-4. *Secure Hash Standard (SHS)*. 2012. [Online; acessado em: 22-Outubro-2013]. Disponível em: <http://csrc.nist.gov/publications/fips/fips180-4-/fips-180-4.pdf>.

FIPS-197. *Specification for the ADVANCED ENCRYPTION STANDARD (AES)*. 2001. [Online; acessado em: 22-Outubro-2013]. Disponível em: <http://csrc.nist-.gov/publications/fips/fips197/fips-197.pdf>.

LITZENBERGER, D. C. *The Python Crytography Toolkit*. 2013. [Online; acessado em: 22-Outubro-2013]. Disponível em: <https://www.dlitz.net/software/pycrypto>.

LIU, G. et al. A program behavior study of block cryptography algorithms on gpgpu. In: *Frontier of Computer Science and Technology, 2009. FCST '09. Fourth International Conference on*. [S.l.: s.n.], 2009. p. 33–39.

MANAVSKI, S. Cuda compatible gpu as an efficient hardware accelerator for aes cryptography. In: *Signal Processing and Communications, 2007. ICSPC 2007. IEEE International Conference on*. [S.l.: s.n.], 2007. p. 65–68.

MEI, C. et al. Cuda-based aes parallelization with fine-tuned gpu memory utilization. In: *Parallel Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010 IEEE International Symposium on*. [S.l.: s.n.], 2010. p. 1–7.

NISHIKAWA et al. High-performance symmetric block ciphers on cuda. In: *Networking and Computing (ICNC), 2011 Second International Conference on*. [S.l.: s.n.], 2011. p. 221–227.

NVIDIA. *Best Practices Guide*. 2013. Versão: CUDA Toolkit Version 5.5.

NVIDIA. *CUDA C Programming Guide*. 2013. Versão: CUDA Toolkit Version 5.5.

NVIDIA. *Parallel Programming and Computing Platform*. 2013. [Online; acessado em: 22-Outubro-2013]. Disponível em: <http://www.nvidia.com/object-/cuda_home_new.html>.

RYAN, C. *Best SSDs For The Money: October 2013*. 2013. [Online; acessado em: 25-Novembro-2013]. Disponível em: <http://www.tomshardware.com/reviews/ssd-recommendation-benchmark,3269-6.html>.

TANENBAUM, A. S.; STEEN, M. v. *Distributed Systems: Principles and Paradigms (2nd Edition)*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 2006. ISBN 0132392275.

WINKLE, W. V. *Gigabit Wireless? Five 802.11ac Routers, Benchmarked*. 2013. [Online; acessado em: 25-Novembro-2013]. Disponível em: <http://www-.tomshardware.com/reviews/wi-fi-802.11ac-router,3386-11.html>.

Glossário

blockDim tamanho de um bloco em execução na GPU. Conta a quantidade de threads em cada bloco.
blockIdx identificador de um bloco de threads na GPU relativo ao número total de blocks, isto é, da dimensão do grid.

gridDim tamanho do grid em execução na GPU. Conta a quantidade de blocos em execução.

nonce número aleatório de 8 bytes gerado para prover entropia para a geração de blocos contadores.

threadIdx identificador de uma thread na GPU relativo ao bloco em que ela está inserida.
tid identificador global da thread. Usada para mapear uma thread qualquer com o seu respectivo bloco de texto.