Renan Corrêa Detomini

Exploração de Paralelismo em Criptografia Utilizando GPUs

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
2010

Renan Corrêa Detomini

Exploração de Paralelismo em Criptografia Utilizando GPUs

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.

Orientadora: Profa. Dra. Renata Spolon Lobato

São José do Rio Preto
2010

Renan Corrêa Detomini

Exploração de Paralelismo em Criptografia Utilizando GPUs

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.

Profa. Dra. Renata Spolon Lobato

Renan Corrêa Detomini

Banca Examinadora:
Prof. Dr. Aleardo Manacero Junior
Prof. Dr. José Márcio Machado

São José do Rio Preto
2010

Para meus pais, Wanderley
e Elza.

AGRADECIMENTOS

Aos meus pais Wanderley e Elza, por nunca deixarem de me apoiar, mesmo nos momentos difíceis.

Ao meu irmão Vitor por estar sempre ao meu lado.

Aos meus amigos de faculdade, especialmente os que conviveram comigo durante muitos anos do curso.

Ao meu amigo Vinícius Godoy Contessoto por ter cedido o hardware necessário para a execução do projeto.

Ao meu primo Luiz Carlos Bernardo Vessosa Júnior, pelos vários anos de amizade e troca de conhecimentos em informática, além da ajuda no entendimento do algoritmo AES e sua implementação.

A minha orientadora Profa. Dra. Renata Spolon Lobato, por ter confiado em mim e no meu trabalho mesmo depois de tantos problemas.

RESUMO

Este trabalho consiste na exploração do poder computacional das unidades de processamento gráfico (GPUs) da NVIDIA em criptografia através do uso da tecnologia *Compute Unified Device Architecture* (CUDA), que foi criada para facilitar o desenvolvimento de computação de propósito geral utilizando o processamento paralelo presente nas GPUs. Para isso, apresenta-se a arquitetura das GPUs NVIDIA e a tecnologia CUDA, além de conceitos e algoritmos de criptografia. Adicionalmente, é feita a comparação do desempenho da versão executada em CPU com a versão paralela dos algoritmos de criptografia *Advanced Encryption Standard* (AES) e *Message-digest Algorithm 5* (MD5) escritos em CUDA. Por fim, apresentam-se os resultados obtidos para estas implementações.

ABSTRACT

This work consists in the exploration of the computational power of the NVIDIA's GPUs in cryptography through the use of the CUDA technology, which was created to ease the development of general purpose computation utilizing the parallel processing presented on GPUs. To do this, it is presented the NVIDIA's GPUs architecture and CUDA technology, besides some concepts and algorithms of cryptography. Additionally, a comparison between the CPU and parallel version written in CUDA of the AES and MD5 cryptography algorithms is made. At last, the results obtained with these implementations are presented.

ÍNDICE

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

AES: Advanced Encryption Standard

AGP: Accelerated Graphics Port

AMD: Advanced Micro Devices

API: Application Programming Interface

CBC: Cipher Block Chaining

CFB: Cipher Feedback

CPU: Central Processing Unit

CUDA: Compute Unified Device Architecture

CTR: Counter

DES: Data Encryption Standard

DRAM: Dynamic Random Access Memory

ECB: Electronic Codebook

FLOPS: Floating Point Operations per Second

FPU: Float Point Unit

FSB: Front Side Bus

GPGPU: General Purpose computing on Graphics Processing Units

GPU: Graphics Processing Unit

MD5: Message-Digest Algorithm 5

NIST: National Institute of Standards and Technology

OFB: Output Feedback

P2P: Peer-2-peer

PCI: Peripheral Component Interconnect

SIMT: Single Instruction Multiple Thread

ULA: Unidade Lógica Aritmética

USB: Universal Serial Bus

VI: Vetor de Inicialização

XOR: Exclusive-or

Capítulo 1 – Introdução

1.1 Considerações iniciais

Computadores domésticos são utilizados para realizar inúmeras tarefas, tais como transações financeiras, sejam elas bancárias ou mesmo compra de produtos e serviços; comunicação, por exemplo, através de e-mails; armazenamento de dados, etc. Portanto, é necessário prover segurança ao usuário. Um dos meios de prover segurança ao usuário é a utilização de criptografia.

Criptografia é o estudo de princípios e técnicas pelas quais a informação pode ser transformada de sua forma original em algo ilegível, de forma que só o remetente e o destinatário saibam o conteúdo da informação, através do uso de uma chave secreta (KNUDSEN 1998). Criptografia é um processo matemático intenso e alguns algoritmos de criptografia podem ser paralelizados.

Com o passar do tempo, a maioria dos desenvolvedores tem se baseado no aumento da velocidade dos processadores para aumentar a produtividade de suas aplicações. No entanto, a partir de 2003, devido ao aumento do consumo de energia e aumento do aquecimento dos novos processadores, o aumento na velocidade dos processadores diminuiu bastante. Assim, os fabricantes começaram a desenvolver processadores com vários núcleos, aumentando assim o poder de processamento, dependendo da aplicação. Com isso, aumentou o interesse dos desenvolvedores em aprender sobre processamento paralelo (KIRK e HWU 2008).

Impulsionado pela crescente complexidade de processamento gráfico, especialmente em aplicações de jogos 3D, as unidades de processamento gráfico

(GPUs) passaram por um tremendo progresso tecnológico. Atualmente, as GPUs apresentam um elevado grau de paralelismo e superam os processadores modernos em operações de ponto flutuante. Além disso, a largura de banda de memória é maior nas GPUs (NVIDIA 2009). Devido a esse enorme poder computacional, surgiu uma nova tendência de processamento paralelo chamada *General Purpose computing on Graphics Processing Units* (GPGPU), que utiliza a GPU não somente para aplicações gráficas, mas também para fazer computação em geral, normalmente associada a *Central Processing Unit* (CPU) (OWENS et al. 2007). Sendo assim, em 2007 a NVIDIA tornou pública uma nova arquitetura de processamento paralelo chamada *Compute Unified Device Architecture* (CUDA), que é fortemente baseada em C e algumas extensões (NVIDIA, 2008). Essa arquitetura pode ser usada em vários campos, tais como: bioinformática, criptografia, dinâmica de fluídos, matemática, processamento de áudio e vídeo, etc (AKOGLU e STRIEMER 2008, GARLAND et al. 2008, SILBERSTEIN et al. 2008, MANAVSKI 2007, STRIPPGEN e NAGEL 2009, TOLKE 2008).

1.2 Motivação e escopo

O processamento paralelo nas GPUs é um campo em pleno desenvolvimento. As GPUs podem ser utilizadas em várias aplicações, além do processamento em aplicações gráficas. A criptografia é uma área bastante importante hoje em dia, visto que a cada dia surgem novas aplicações para Internet que, se mal utilizadas, podem colocar em risco a segurança dos usuários. Assim, é necessário aumentar o desempenho dos algoritmos de criptografia. Para isso, pode ser utilizado o processamento paralelo presente nas GPUs em conjunto com a arquitetura CUDA.

1.3 Objetivos e metodologia

O principal objetivo do trabalho é utilizar o poder computacional das GPUs junto com a plataforma CUDA e a possível paralelização dos algoritmos de criptografia para a realização de processos de encriptação e desencriptação. Como

são processos pesados para o processamento, a utilização de paralelismo pode aumentar o desempenho. Além disso, é possível utilizar o poder de processamento das *GPUs* para tentar ‘quebrar’ o texto cifrado, obtendo assim o texto original, processo chamado de *cracking*.

A versão sequencial do algoritmo *Advanced Encryption Standard* (AES), programado em linguagem C, junto com sua versão programada em CUDA, serão mostradas para efeitos de comparação de desempenho. Além disso, será comparado também o desempenho da versão sequencial com a versão em CUDA do algoritmo de quebra de cifra do *Message-Digest algorithm 5* (MD5).

1.4 Organização da monografia

O trabalho desenvolvido foi distribuído em quatro capítulos, descritos brevemente a seguir:

- Capítulo 2: apresenta toda a base teórica necessária para a compreensão e desenvolvimento do trabalho. São abordados temas tais como arquitetura das GPUs, CUDA e Criptografia.
- Capítulo 3: descreve o trabalho desenvolvido e seus objetivos, especificando detalhes da implementação desenvolvida para CPU e GPU.
- Capítulo 4: mostra os testes realizados, seus resultados e a avaliação desses resultados em relação ao trabalho proposto.
- Capítulo 5: traz as principais conclusões obtidas com o trabalho e possíveis propostas para trabalhos futuros relativos à área.

Capítulo 2 – Fundamentação Teórica

2.1 Considerações Iniciais

Este capítulo traz a introdução teórica necessária para o desenvolvimento do projeto proposto. São apresentados conceitos de computação de alto desempenho, unidades de processamento gráfico, CUDA, programação paralela, criptografia e segurança.

2.2 Computação de alto desempenho

A principal motivação do desenvolvimento dos sistemas de computação é a tentativa de obtenção máxima de desempenho utilizando cada vez menos recursos. As tecnologias e mecanismos criados para esse fim diferem muito entre si, principalmente se forem comparadas as unidades de processamento autônomas (processadores e unidades auxiliares locais) com as redes de processadores autônomas (*Clusters*, *Grids* e *Clouds*).

Será mostrado um pouco das características técnicas das mais diversas tecnologias, a fim de tornar mais claro que quando se compara CPUs com GPUs, redes de baixa latência com redes *Ethernet*, barramentos *Peripheral Component Interconnect* (PCI) com PCI-Express, não se deixa de comparar grandezas relacionadas e que os problemas, restrições e soluções em determinado nível podem constituir também problemas, restrições e soluções em diversos outros níveis.

2.2.1 Maximização do desempenho

Embora as áreas de investigação em arquiteturas intra-computador e inter-computadores sejam vastas, há um objetivo que pode ser considerado comum a todos, que é o de aumentar a capacidade de computação recorrendo a um determinado número de unidades de processamento e de memórias interligadas por barramentos de comunicação. O desempenho depende de cada um desses três fatores: no caso de um processador, existem as Unidades Lógicas Aritméticas (ULA), memórias cache e o barramento interno; no caso de um sistema autônomo existem os diversos processadores, os barramentos (*Front Side Bus* (FSB), PCI, PCI-Express, *Universal Serial Bus* (USB)); no caso de um cluster existem os nós de computação, as memórias distribuídas e as redes de interligação.

No caso dos processadores importa saber quanto tempo demora a execução de um determinado conjunto de instruções. No caso da memória interessa saber se o seu desempenho e capacidade são suficientes para suportar o programa com seus dados e no caso das comunicações qual a largura de banda, latência e o atraso que estas impõem no cálculo e acesso a memória.

2.2.2 Desenvolvimento nos microprocessadores

A evolução nos microprocessadores passa cada vez mais pelo aumento no número de núcleos e da memória cache ao invés de aumento do desempenho dos núcleos individuais. Com isso, os programadores ou compiladores das aplicações se tornam responsáveis por adaptarem os programas para explorar todo o potencial dos recursos computacionais apresentados nas mais diferentes formas, mesmo para aplicações que se destinam a ser utilizadas em um único computador.

O desempenho do sistema passou a ser avaliado também em relação ao consumo de energia e consequentemente a capacidade de resfriamento. Esse aspecto passou a ter suma importância assim que foi verificado que as gerações de processadores recentes aumentavam o consumo de eletricidade para valores impensáveis há poucos anos atrás, em particular devido às frequências elevadas e o aumento no número de transistores. Os ganhos recentes tem sido significativos e cita-

se como exemplo um processador Xeon da geração NetBurst de núcleo único de 3,6 GHz, que consome 110W para produzir 7,2 bilhões de operações de ponto flutuante (GFLOPS), enquanto um processador Xeon Quadcore baseado na arquitetura Core2 consome menos de 80W para produzir 48 GFLOPS, um aumento da ordem de eficiência da ordem de uma grandeza em apenas 2 anos (WECHSLER 2006).

2.2.3 Processamento com coprocessadores

A utilização de coprocessadores para aumentar as capacidades dos processadores genéricos está longe de ser nova, sendo os *Float Point Unit* (FPU) os mais antigos. Estas funcionalidades básicas da FPU foram incluídas nas CPUs atuais, mas existem certas situações em que é necessário o uso de coprocessadores, especialmente nos campos de processamento de sinal, processamento gráfico e criptografia (KANT et. al. 2006, GOVINDAJARU et. al. 2006, TRICHINA et. al. 2005).

O que distingue um processador de um coprocessador é que o primeiro pode funcionar sem o segundo, mas o inverso não é verdadeiro. Com os primeiros processadores aritméticos, era comum o uso de um *slot* dedicado para sua instalação junto das CPUs. Atualmente, os coprocessadores são disponibilizados na forma de placas de expansão, que é uma forma mais flexível e econômica de integrar os componentes. No entanto, a utilização de um barramento de expansão implica em limitações de largura de banda e a comunicação com atrasos significativos. Assim, como forma de contornar esse problema, tentou-se adequar o barramento às características dos coprocessadores, que deu origem a barramentos dedicados como o *Accelerated Graphics Port* (AGP), que possui uma largura de banda superior ao PCI (INTEL 2002). Além disso, mais recentemente criaram-se mecanismos que permitem integrar os coprocessadores diretamente nos barramentos dos processadores, tal como a iniciativa Torrenza, da *Advanced Micro Devices* (AMD) (AMD 2006).

2.3 As unidades de processamento gráfico

Na seção anterior pretendeu-se apresentar uma panorâmica do estado atual dos sistemas de computação para tentar contextualizar devidamente o processamento em GPUs. Nesta seção introduz-se a GPU como unidade de processamento auxiliar e apontam-se suas vantagens e desvantagens.

O processamento em tempo real envolvendo compressão de imagens, processamento de sinais, cálculos para gráficos 3D e codificação de vídeo é bastante complexo, requerendo cálculos da ordem de bilhões de operações por segundo. O custo dessas operações faz com que diversos fabricantes escolham desenvolver processadores específicos para mídia. O custo e esforço de desenvolvimento envolvidos são altos e o hardware não é flexível, ou seja, não se adapta facilmente a evolução de novos algoritmos e programas de processamento de mídia. Com o objetivo de atender a esta demanda por flexibilidade, surge a motivação para o uso de *Stream Processors* programáveis (VENKATASUBRAMANIAN 2003).

Com a introdução dos *Stream Processors* programáveis surgiu a necessidade de criar linguagens que pudessem possibilitar ao programador sua utilização de maneira simples. Essas linguagens foram evoluindo junto com as GPUs. Atualmente duas empresas chamadas ATI e NVIDIA dominam o mercado de desenvolvimento de jogos e agora disputam um lugar no mercado de computação de alto desempenho.

A NVIDIA tem comercializado GPUs que são o estado da arte na implementação de *Stream Processors*. Trata-se das novas placas gráficas a partir das séries 8, Tesla e algumas Quadro. A nova tecnologia apresentada pela NVIDIA vem acompanhada de um novo modelo de arquitetura e programação chamada CUDA.

A pilha de software (Figura 2.1) do CUDA envolve uma *Application Programming Interface* (API) com suporte direto a várias funções matemáticas, primitivas de computação gráfica, bibliotecas, suporte ao *runtime* e ao *driver*, que otimiza e gerencia a utilização de recursos diretamente com a GPU.

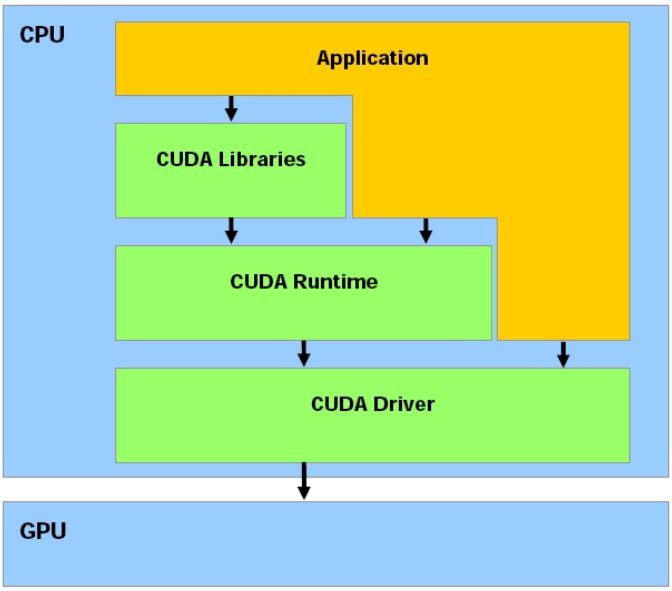

Figura 2.1 Pilha de software CUDA (NVIDIA 2009).

Com esse avanço das GPUs, o número de operações de ponto flutuante desses processadores, quando comparados com as CPUs convencionais, cresce mais rápido ao longo dos anos (Figura 2.2). A razão disso é que a GPU dedica o uso de seus transistores ao processamento de dados ao invés de controle de fluxo e memória cache, como visto na figura 2.3.

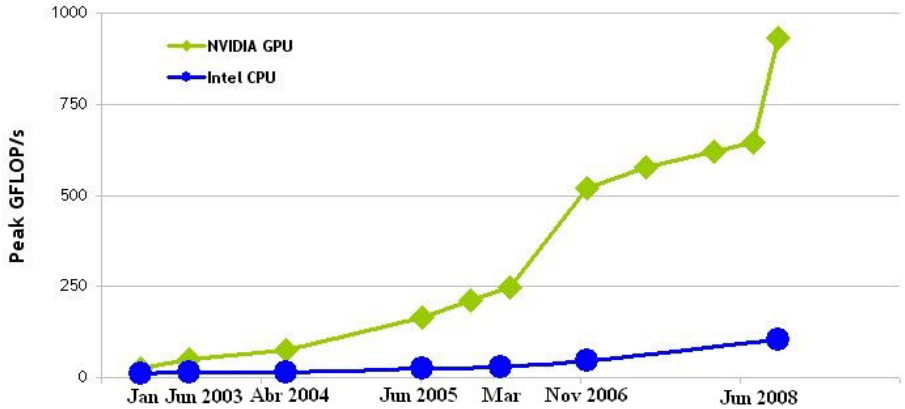

Figura 2.2 Comparação de GFLOPS entre CPU e GPU (NVIDIA 2009).

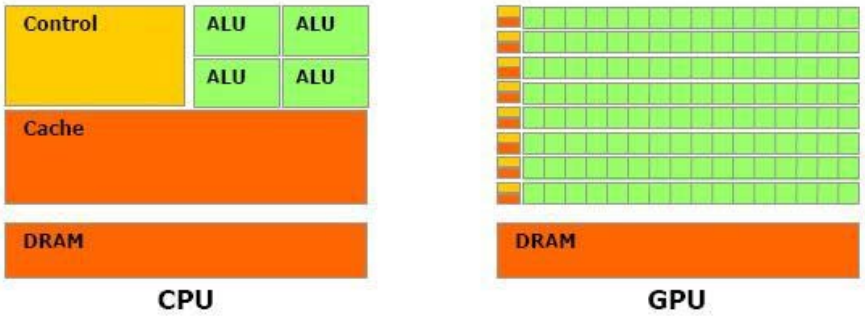

Figura 2.3 Transistores em CPUs e GPUs (NVIDIA 2009).

A grande novidade é a maneira eficiente com que a arquitetura CUDA juntamente com a arquitetura das GPUs possibilita o desenvolvimento de aplicações que podem explorar ao máximo o paralelismo dos dados. Em uma dada aplicação, um mesmo trecho de código é executado em paralelo para pequenos blocos de dados, com a existência de várias *caches* e níveis de hierarquia de memória que escondem a latência de acesso a esses blocos. Essas *caches* são também chamadas de memórias *on-chip* compartilhadas e possuem acesso rápido de leitura e escrita. Estas memórias são parte da hierarquia de memória que diminui o acesso às memórias externas, reduzindo assim o tempo de execução dos aplicativos (Figura 2.4).

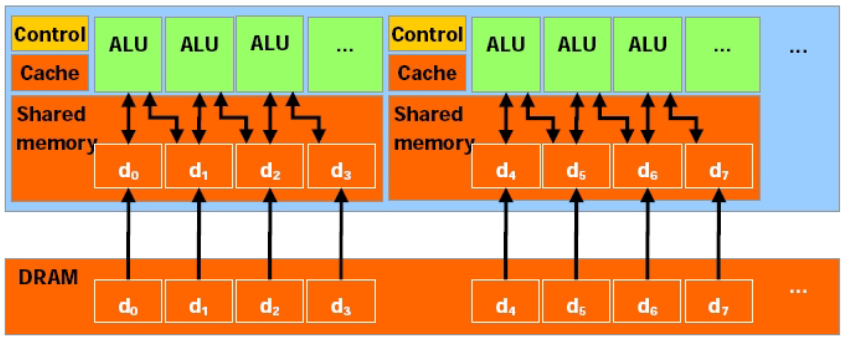

Figura 2.4 Memória Compartilhada (NVIDIA 2009).

A arquitetura CUDA possibilita também o acesso à memória de maneira que era somente suportada pela CPU, como as operações de *gather* e *scatter* com a memória *Dynamic Random Access Memory* (DRAM) da GPU (figura 2.5).

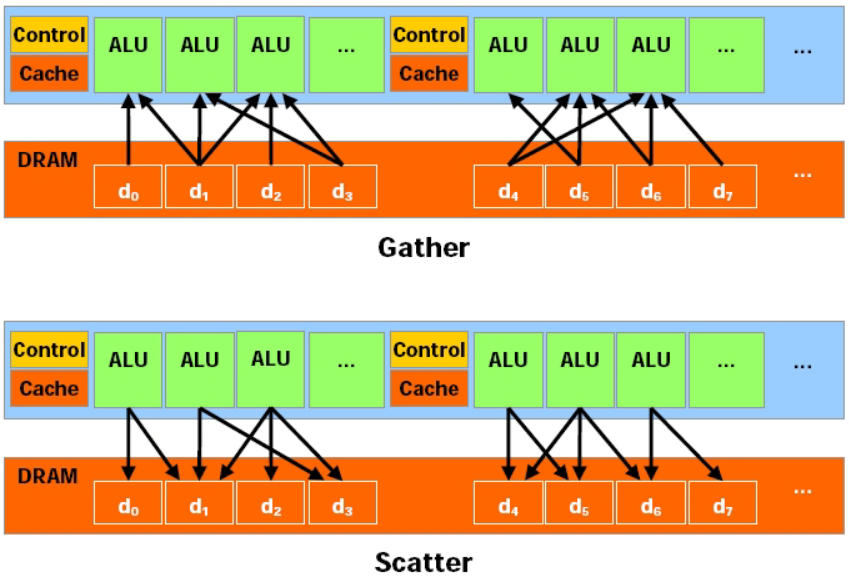

Figura 2.5 Operações *gather* e *scatter* (NVIDIA 2009).

2.3.1 A arquitetura da série 8 da NVIDIA

Um dos objetivos da arquitetura CUDA foi adicionar o suporte a computação paralela de alto desempenho e isso pode ser encontrado nas GPUs a partir da série 8, Tesla e Quadro. Nesta nova série de GPUs, a NVIDIA considera seus *shaders stream* ou *thread processors*, como visto na figura 2.6.

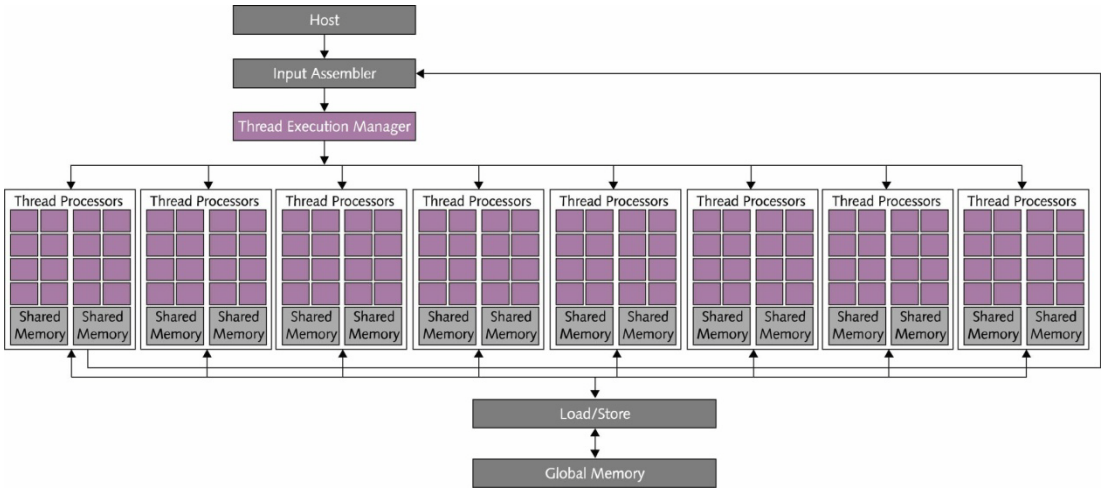

Figura 2.6 Arquitetura NVIDIA série 8 (HALFHILL 2008).

Quando a CPU invoca a execução de um *kernel*, que é um código que será

executado na GPU sobre um conjunto de informações, os blocos de *threads* são executados sobre o conjunto de dados descrito pelo programa. Na figura 2.6 pode-se perceber que os *thread processors* ou *stream processors* são agrupados em grupos de 8, sendo que cada um desse grupo recebe o nome de *multiprocessor*.

Um *multiprocessor* também tem uma memória compartilhada. Ele é responsável por gerenciar, criar e executar as *threads* em *hardware*. O gerenciamento das *threads* é feito por uma nova arquitetura chamada *Single Instruction Multiple Thread* (SIMT). A unidade SIMT realiza todas as tarefas inerentes ao gerenciamento de *threads*, dividindo-as em grupos de 32, chamados *warps*. Quando um *multiprocessor* tem que executar um grid ou um bloco de *threads*, elas são divididas em *warps* que são escalonados pelo SIMT (NVIDIA 2009). Cada *multiprocessor* possui quatro tipos de memória *on-chip*, como mostrado na figura 2.7:

- Uma *cache* constante compartilhada entre os *thread processors*.
- Uma *cache* de textura para otimizar o acesso a memória de texturas.
- Um conjunto de registradores por *thread processor*.
- Memória compartilhada entre os *thread processors*.

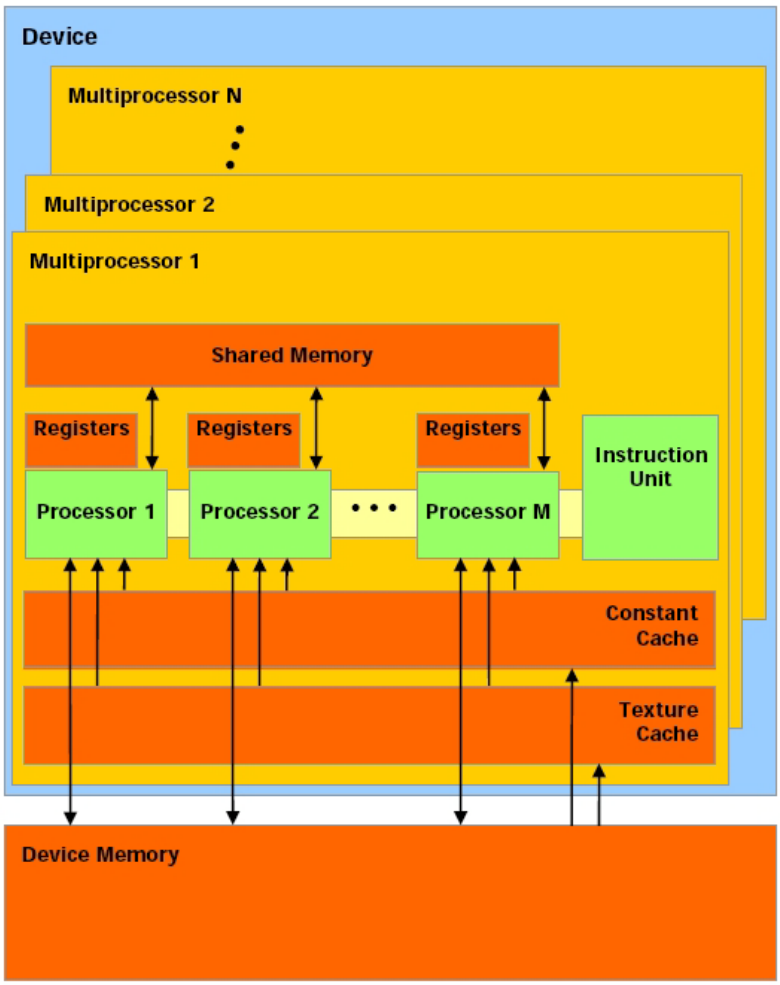

Figura 2.7 Hierarquia de memória (NVIDIA 2009).

A execução de um aplicativo que utiliza CUDA é realizada tanto na GPU (*device)* quanto na CPU (*host*). Apenas o código que envolve diretivas especiais é executado somente na GPU.

2.3.2 Hierarquia de memória

A abordagem do CUDA divide o conjunto de dados em pedaços menores na memória *on-chip* permitindo que várias threads compartilhem cada um destes pedaços. Ao guardar os dados localmente reduz a necessidade de acesso à memória *off-chip*, melhorando o desempenho visto que essa última memória tem latência alta em relação às memórias *on-chip*. Este comportamento é ilustrado na figura 2.8.

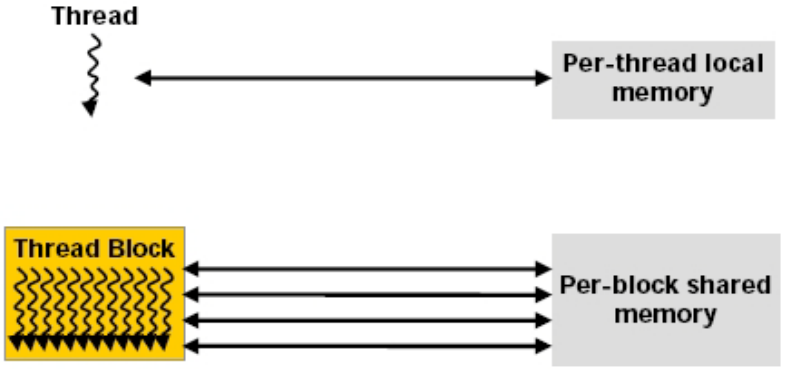

Figura 2.8 Relacionamento entre memórias e threads (NVIDIA 2009).

Além da área global de dados, há ainda outros dois espaços de memória que podem ser acessados pelas *threads*: o espaço constante e de textura. Ambos são otimizados para diferentes usos de memória e os dados lá contidos não desaparecem quando um *kernel* termina.

Tanto o *host* quanto o *device* possuem sua própria DRAM (memória global), que é a memória de alta latência e normalmente com maior capacidade de armazenamento. Todo o gerenciamento de memória realizado por um aplicativo passa pela *runtime* do CUDA, incluindo alocação e desalocação e transferência de dados entre as memórias do *host* e do *device*. O relacionamento entre os *grids* de *threads* e a memória global é ilustrado na figura 2.9.

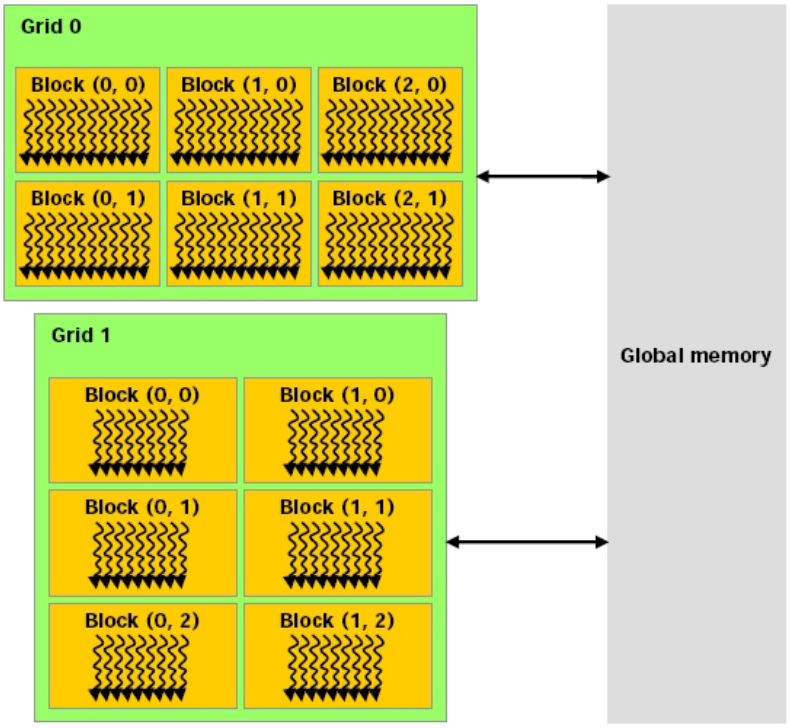

Figura 2.9 Memória global e grids (NVIDIA 2009).

2.3.3 Gerenciamento de *threads*

Os acessos à memória *off-chip* geralmente não geram *stalls* em um *stream processor*. Durante a requisição de dados, uma *thread* entra na fila de inatividade e é substituída por outra pronta pra entrar em atividade. Assim que o dado se tornar disponível, a primeira volta à fila de *threads* disponíveis. Os grupos de threads executam de maneira *round-robin*, garantindo que cada thread tenha seu tempo de execução sem prejudicar as outras.

Outro recurso importante de CUDA é que as *threads* não precisam ser explicitamente escritas pelos desenvolvedores, pois um *thread manager* que gerencia as *threads* é implementado em *hardware*. Toda dificuldade inerente à criação e divisão de blocos de *threads* é feita pelo *hardware*. Cada thread, da maneira convencional, possui pilha, registradores e memória local própria. A NVIDIA alega que CUDA elimina completamente *deadlocks* através da função __syncthreads(), que está presente na API e é convertida pelo compilador para uma instrução da GPU.

2.3.4 Escolha de dados

Para utilizar CUDA de maneira a obter o melhor desempenho, programadores devem escolher a melhor maneira de dividir os dados em blocos menores.

Há o desafio de achar o número ótimo de *threads* e blocos que manterão a GPU totalmente ocupada. Os fatores para análise incluem o tamanho do conjunto global de dados; a máxima quantidade de dados locais que um bloco de *threads* pode compartilhar; o número de *stream processors* da GPU e o tamanho das memórias *on-chip*.

2.4 Paradigma de programação

A NVIDIA esconde a complexidade da arquitetura CUDA através de uma API (figura 2.10), assim os programadores não precisam escrever diretamente na placa e utilizar funções gráficas pré-definidas na API para operar a GPU. A primeira vantagem desse modelo é que os programadores não precisam se preocupar com

detalhes complexos de *hardware* da GPU. Outra vantagem é a flexibilidade, permitindo que a NVIDIA mude bastante a arquitetura de sua GPU sem tornar a API obsoleta e quebrar *softwares* já existentes. Nota-se que apesar de não tornar a API obsoleta, será necessário que os programadores de novos modelos utilizem os recursos introduzidos a fim de obter um *software* otimizado para desempenho.

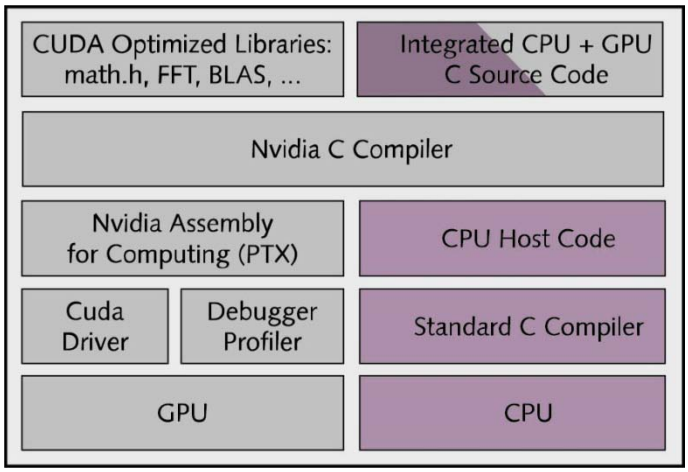

Figura 2.10 API CUDA (HALFHILL 2008).

A API consiste em um conjunto de marcadores e diretivas especiais para linguagem C que permitem que o compilador identifique se o código em questão é para GPU ou CPU. Além disso, possui uma biblioteca para *runtime*, que é dividida em componentes que gerenciam operações do *host* (e principalmente comunicação desse com o *device*) e de componentes comuns, como tipos adicionais e subconjunto de funções da biblioteca padrão C que podem ser utilizados tanto no *host* quanto no *device*.

2.4.1 Extensões

Para escrever um código em CUDA, o programador pode utilizar tanto C quanto C++ (NVIDIA 2009). Para tornar possível o modelo de programação, o CUDA adiciona diversos marcadores especiais a essas linguagens. Estes marcadores, junto com o conjunto de funções fornecidas pela API, completam o suporte necessário para o desenvolvimento de aplicativos em CUDA.

O entendimento desses marcadores especiais é crucial para entender o modelo de programação. Dentre os mais importantes existe o marcador *__global__*, que se

uma função é declarada com esse marcador, então essa função é considerada um *kernel*, sendo acessível de todo o programa e deve ser compilada para GPU e não CPU.

```
//Definição de um kernel
__global__ void VectorSubtract (float *A, float *B)
{
…
}
```

Figura 2.11 Declaração de um *kernel* (NVIDIA 2009).

Considerando a função *VectorSubtract* acima, que possui a diretiva *__global__*, é possível especificar também o número de blocos de *threads* e o número de *threads* por bloco que executarão a mesma. Isto é feito através da diretiva <<<número de blocos, número de threads por bloco>>>. No código abaixo, temos um bloco com N threads, onde cada uma das N *threads* executará paralelamente a função *VectorSubtract*.

```
int main ()
{
//Chamar o kernel
VectorSubtract <<<1,N>>>(A,B);
}
```

Figura 2.12 Especificando blocos e *threads* (NVIDIA 2009).

Na figura 2.13 mostra-se um exemplo completo utilizando os marcadores descritos acima.

```
Computing y _ ax + y with a serial loop:
void saxpy_serial(int n, float alpha, float *x, float *y)
{
    for(int i = 0; i<n; ++i)
        y[i] = alpha*x[i] + y[i];
}
// Invoke serial SAXPY kernel
saxpy_serial(n, 2.0, x, y);

Computing y _ ax + y in parallel using CUDA:
__global__
void saxpy_parallel(int n, float alpha, float *x, float *y)
{
    int i = blockIdx.x*blockDim.x + threadIdx.x;
    if( i<n )  y[i] = alpha*x[i] + y[i];
}
// Invoke parallel SAXPY kernel (256 threads per block)
int nblocks = (n + 255) / 256;
saxpy_parallel<<<nblocks, 256>>>(n, 2.0, x, y);
```

Figura 2.13 Exemplo de código (HALFHILL 2008).

Na primeira parte do código há um código sequencial, onde a função *saxpy_serial* é chamada por apenas uma thread em uma CPU normal. A segunda versão é escrita utilizando CUDA, sendo a função *saxpy_parallel* executada por *nblocks* com 256 *threads* cada.

É possível identificar qual é o índice da *thread* atual de dentro de um *kernel* através da variável *built-in threadIdx*. Reescrevendo o código da figura 2.11, é possível especificar que cada *thread* executada calcule a subtração de um par de posições, como mostrado na figura 2.14.

```
//Definição de um kernel
__global__ VectorSubtract (float *A, float *B)
{
int idx = threadIdx.x;
C[idx] = A[idx] – B[idx];
}
```

Figura 2.14 Mostrando índice de *threads* (NVIDIA 2009).

Em CUDA é possível especificar blocos multidimensionais de *threads*.

O caso mais comum *multithread* é utilizar uma sequência de uma única dimensão de *threads*, como por exemplo, um conjunto de *threads* de 1 a N. Com blocos que possuem *threads* com mais de uma dimensão é possível que cada *thread* do bloco acesse uma posição diferente em uma matriz multidimensional, cobrindo no total todas as posições, como mostrado na figura 2.15.

```
__global__ void matrixAdd (float A[N][N], float B[N][N], float C[N][N])
{
int i = threadIdx.x;
int j = threadIdx.y;
C [i] [j] = A [i] [j] + B [i] [j];
}
int main()
{
...
// Kernel invocation
dim3 dimBlock (N, N);
matrixAdd<<<1, dimBlock>>>(A, B, C);
}
```

Figura 2.15 Blocos multidimensionais de *threads* (NVIDIA 2009).

O tipo *dim3* determina um vetor de inteiros e é utilizado para especificar dimensões. No trecho acima, um bloco contêm N*N *threads* que podem ser acessadas como se estivessem em uma matriz. No escopo do *kernel*, a posição multidimensional é obtida através da variável *threadIdx*, especificando-se a dimensão desejada, como por exemplo *threadIdx.x*, *threadIdx.y* e *threadIdx.z*.

Outra abordagem possível é a utilização de *grids*. Os diversos blocos de *threads* são organizados em um nível acima de abstração, em *grids* com uma ou duas dimensões, onde cada bloco de *threads* se comporta como se fosse um elemento de uma matriz. O acesso aos blocos é realizado utilizando a variável *built-in blockIdx*, que permite a indexação utilizando *blockIdx.x* ou *blockIdx.y*, como mostrado na figura 2.16.

```
__global__ void matAdd (float A [N] [N], float B [N] [N], float C [N] [N])
{
int i = blockIdx.x * blockDim.x + threadIdx.x;
int j = blockIdx.y * blockDim.y + threadIdx.y;
if (i < N && j < N)
C [i] [j] = A [i] [j] + B [i] [j]; }
int main()
{
// Kernel invocation
dim3 dimBlock(16, 16);
dim3 dimGrid((N + dimBlock.x { 1) / dimBlock.x,
(N + dimBlock.y { 1) / dimBlock.y);
matAdd<<<dimGrid, dimBlock>>>(A, B, C); }
```

Figura 2.16 Especificando *grids* (NVIDIA 2009).

Outra variável *built-in blockDim* obtém a dimensão de cada bloco de *threads*. A figura 2.17 contém um exemplo visual dos *grids* de *threads*.

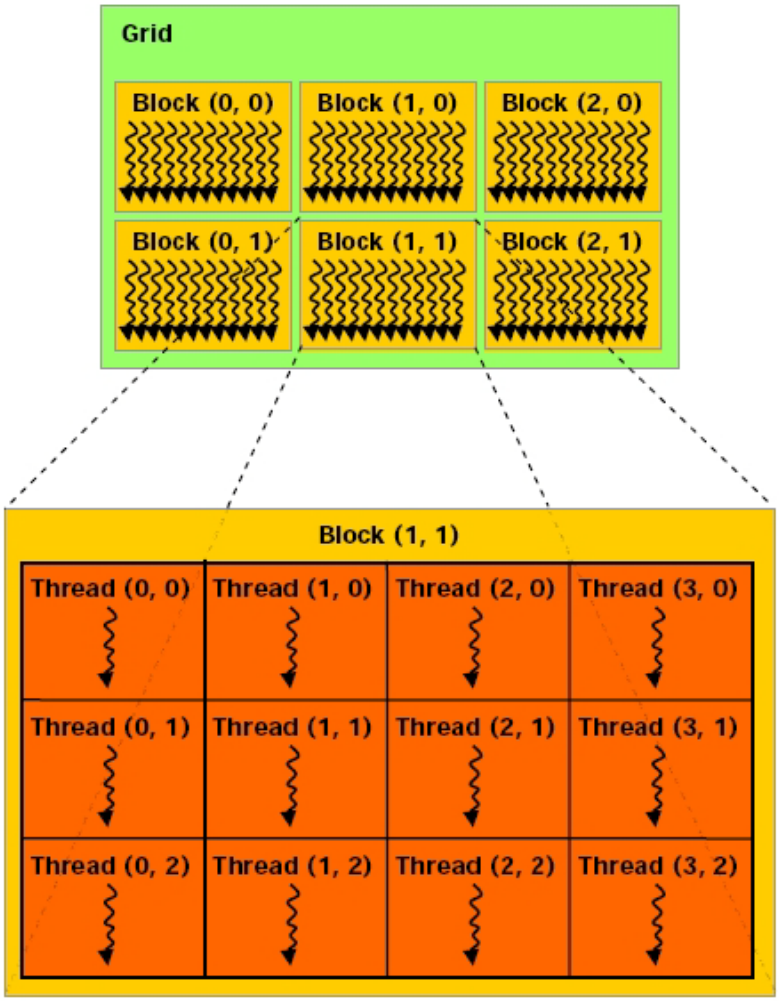

Figura 2.17 Blocos e *grids* (NVIDIA 2009).

2.4.2 Grupos de extensões

As extensões são divididas em alguns grupos:

- Qualificadores de variáveis: especificam em que tipo de memória do *device* a variável deve ser alocada.
- Qualificadores de funções: especificam se a função é executada no *host* ou no *device* e se pode ser chamada a partir de um deles.
- Uma diretiva que especifica como um *kernel* é executado no *device* a partir do *host*. Isto é chamado também de configuração de execução.
- Quatro variáveis *built-in* que especificam as dimensões de *grids* e

blocos.

Os qualificadores utilizados em funções são:

- __device__: declara uma função que é executada no *device* e pode ser chamada somente a partir do mesmo.
- __global__: especifica um *kernel* que é executado no *device* e chamado somente a partir do *host*.
- __host__: declara uma função que só pode ser chamada e executada no host.

Se uma função não possui nenhum qualificador, ela é considerada do tipo *__host__* por padrão. O qualificador *__global__* possui as seguintes restrições:

- Deve obrigatoriamente retornar *void*.
- São assíncronas e devem obrigatoriamente especificar uma configuração de execução.
- Os parâmetros são passados através de memória compartilhada, com limite de 256 bytes.

Os qualificadores utilizados em variáveis são:

- __device__: reside no *device*. Se nenhum outro qualificador for usado, a variável reside no espaço de memória global, tem o tempo de vida da aplicação e é acessível por todas as *threads* dentro de um *grid* e pelo *host* através da biblioteca de *runtime*.
- __constant__: reside no espaço de memória constante, tem o tempo de vida de uma aplicação e possui a mesma acessibilidade do __device__.
- __shared__: reside no espaço de memória compartilhada de um bloco de *threads*, tem o tempo de vida de um bloco e é acessível a todas as *threads* de um bloco.

Uma escrita a variável *__shared__* só pode ser visualizada por outra *thread* após a execução de *__syncthreads*.

Na última seção foi apresentada a diretiva que deve ser utilizada durante a

invocação de um *kernel* a fim de especificar tamanho de blocos, *grids*, etc. Em qualquer chamada a uma função __*global*__ deve ser especificada uma configuração de execução da forma <<<Dg, Db, Ns, S>>>, onde cada argumento tem os detalhes especificados na tabela 2.1:

Tabela 2.1 Parâmetros de configuração de execução (NVIDIA 2009).

Parâmetro	**Tipo**	**Obrigatório**	**Função**
Dg	dim3	Sim	Dimensão e tamanho do grid
Db	dim3	Sim	Dimensão e tamanho do bloco
Ns	size_t	Não	Número de bytes na memória compartilhada alocada dinamicamente
S	cudaStream_t	Não	Especifica um *stream* adicional

É importante notar que Dg.x × Dg.y é o número total de blocos e Db.x × Db.y × Db.z é o número de *threads* por bloco.

Por último temos as variáveis *built-in*, especificadas na tabela 2.2:

Tabela 2.2 Variáveis *built-in* (NVIDIA 2009).

Variáveis *built-in*	**Tipo**	**Especificação**
gridDim	dim3	Dimensão do *grid*
blockIdx	uint3	Índice do bloco no *grid*
blockDim	dim3	Dimensão do bloco
threadIdx	uint3	Índice da *thread* no bloco
warpSize	int	Tamanho do *warp* em *threads*

2.5 Introdução à criptografia

A arte da escrita cifrada é muito antiga. Bastou o homem inventar o alfabeto e desenvolver a escrita que surgiu a necessidade de escrever textos secretos. A evolução na maneira de se escrever esses textos secretos evoluiu lentamente.

Cripto vem do grego kryptós e significa escondido. Também do grego, logia significa falar, estudo. Logo, criptologia é o estudo de tudo que envolve escrita cifrada e pode ser dividida em duas áreas chamadas criptografia, a escrita secreta, e

criptoanálise, a análise do segredo.

A criptografia é a ciência de escrever mensagens que ninguém poderia ler, exceto o remetente e o destinatário. A criptoanálise é a ciência de tentar quebrar o método utilizado e decifrar e ler a mensagem cifrada.

As palavras, caracteres ou letras da mensagem original são chamadas de mensagem ou *plaintext*, enquanto que as respectivas da mensagem cifrada são chamadas mensagem cifrada ou *ciphertext*. O processo de converter um *plaintext* em *ciphertext* é chamado de cifragem e o inverso é chamado de decifragem. É importante ressaltar que não existem na língua portuguesa palavras como encriptar, desencriptar, cifragem, com sentidos relevantes à criptografia. Mesmo assim, serão usados esses termos no decorrer do trabalho. Na prática, qualquer mensagem cifrada é resultado de um algoritmo criptográfico associado a uma chave específica.

2.5.1 Visão histórica

O estudo da criptografia cobre bem mais do que apenas cifragem e decifragem. É um ramo especializado da teoria da informação com muitas contribuições de outros campos da matemática e do conhecimento.

Os primeiros relatos sobre ocultação de mensagens provêm das histórias de Heródoto, filósofo e historiador, que narrou as guerras e conflitos entre a Grécia e a Pérsia, durante o século V antes de Cristo (KAHN 1996).

Uma das primeiras técnicas criptográficas que surgiu foi a cifra de César, usada pelo imperador romano Júlio César para transmitir mensagens aos seus exércitos. A técnica consistia em substituir letras do alfabeto por símbolos ou por outras letras e são chamadas de cifras monoalfabéticas.

Por volta do século VIII os árabes desenvolveram uma técnica para verificar a autenticidade dos textos sagrados de Maomé. Esta técnica foi utilizada posteriormente para criptoanalisar mensagens criptografadas com os métodos de substituição monoalfabéticas. A técnica consiste na análise estatística da frequência com que as letras de um determinado idioma aparecem.

No ano de 1563 foi criado por Blaise de Viginére um novo sistema que inicialmente não podia ser criptoanalisado por análise de frequência. O sistema era

baseado em mais de um alfabeto e conhecido como método de cifragem polialfabético. Em 1854 a cifra de Viginére foi quebrada por Charles Babbage e por muitos anos nenhum outro método de criptografia foi desenvolvido apresentando relativa segurança (SINGH 2000).

Com o desenvolvimento dos computadores mecânicos no início do século XX, máquinas mecânicas de criptografia surgiram e tornaram os processos de criptografia por substituição e transposição mais complexos. Antes do início da Segunda Guerra Mundial, a Alemanha havia desenvolvido uma máquina de criptografia mecânica batizada de Enigma que utilizava métodos já conhecidos de substituição e transposição tão complexos que a criptoanálise manual de suas mensagens era quase impraticável.

Após a Segunda Guerra Mundial e com a evolução da informática, computadores foram desenvolvidos que possibilitaram a criação de algoritmos de substituição e transposição ainda mais complexos. Uma chave era fornecida ao destinatário e em seguida um algoritmo era utilizado para criptografia. A segurança do sistema era baseada no fato da chave encontrar-se segura. Esse sistema recebeu o nome de algoritmos simétricos, pois a mesma chave pra encriptar e desencriptar a mensagem transmitida era utilizada.

2.5.2 Objetivos da criptografia

A criptografia tem principalmente quatro objetivos:

1. Confidencialidade: só o destinatário original da mensagem deve ser capaz de extrair o conteúdo original de sua forma cifrada.
2. Integridade: o destinatário deve ser capaz de verificar se o conteúdo da mensagem foi alterado.
3. Autenticação: o destinatário deve ser capaz de identificar o remetente e verificar se foi ele mesmo quem enviou a mensagem.
4. Irretratabilidade: o emissor da mensagem não poderá negar o envio da mesma.

Nem todos os sistemas criptográficos atingem todos os objetivos citados.

Normalmente existem algoritmos específicos para cada uma das funções.

2.5.3 Tipos de criptografia

A criptografia simétrica foi o primeiro tipo de criptografia criado e é usado principalmente para confidencialidade da informação. Funciona transformando um *plaintext* em texto cifrado, utilizando uma chave secreta. O processo inverso pode ser realizado utilizando a mesma chave. Este método é relativamente rápido e necessita que o emissor e receptor tenham conhecimento da chave. O ponto crítico da criptografia simétrica parte do fato de que ambos precisam ter conhecimento da chave, o que inviabiliza grande parte das transações via Internet. O algoritmo simétrico mais utilizado hoje em dia é o AES, e este é o algoritmo utilizado para a realização desse trabalho.

Na criptografia assimétrica são utilizadas duas chaves para o processo de encriptação e desencriptação, a chave pública e a privada. A chave pública pode ser livremente distribuída, enquanto a chave privada deve ser conhecida somente pelo emissor da mensagem. Neste tipo de criptografia, uma mensagem criptografada com a chave pública somente pode ser descriptografada pela chave privada correspondente, ou seja, um emissor A usa a chave pública do recipiente B para encriptar a mensagem e o recipiente B usa sua chave privada para desencriptar a mensagem. É relativamente lento e principalmente usado para autenticidade e confidencialidade. O algoritmo de criptografia assimétrica mais usado atualmente é o RSA.

2.5.4 Função criptográfica *hash*

Um *hash*, também chamado de *message digest* ou *checksum*, é uma espécie de assinatura ou impressão digital que representa o conteúdo de um fluxo de dados. Uma função *hash* recebe dados de comprimento arbitrário e retorna um número fixo de *bits*, chamado de valor *hash*. Qualquer mudança na mensagem original muda também o valor *hash*. Se uma função *hash* satisfizer requisitos adicionais, ela pode

ser usada em aplicações criptográficas, como por exemplo, proteger a autenticidade de mensagens enviadas através de canais inseguros. Uma função *hash* ideal precisa satisfazer os seguintes requisitos:

1. Fácil de calcular o *hash* de qualquer mensagem.
2. Inviável encontrar a mensagem através do *hash*.
3. Inviável modificar a mensagem sem modificar o *hash*.
4. Inviável encontrar duas mensagens diferentes com o mesmo *hash*.

Uma aplicação importante dos *hashes* é a verificação de integridade da mensagem. Para determinar se qualquer mudança em uma mensagem ou arquivo foi realizada, basta comparar o *hash* da mensagem antes e depois da transmissão. Se qualquer bit for diferente, então a mensagem ou arquivo foi alterado. Uma aplicação relacionada é a armazenagem de senhas. Geralmente, por questões de segurança, as senhas não são armazenadas em *plaintext*, mas sim em sua forma de *hash*. Assim, para realizar a autenticação do usuário, é calculado o *hash* da senha apresentada pelo usuário e comparado com o *hash* armazenado em tempo real, realizando assim a autenticação.

2.6 O algoritmo AES

O atual padrão de criptografia simétrica dos EUA se originou em um concurso lançado em 1997 pelo *National Institute of Standards and Technology* (NIST). Havia a necessidade de escolher um algoritmo mais seguro e eficiente para substituir o algoritmo *Data Encryption Standard* (DES), que apresentou fragilidades. O novo algoritmo deveria atender a certos pré-requisitos como ser divulgado publicamente; não possuir patentes; cifrar em blocos de 128 bits usando chaves de 128, 192 ou 256 bits; possibilidade de ser implementado tanto em hardware quanto em software; ser mais rápido que o 3DES, uma variação do DES. Em 1998, na primeira conferência dos candidatos AES, 15 candidatos foram apresentados. Um ano depois, na segunda conferência, cinco candidatos se tornaram finalistas. Em 2000, o algoritmo Rijndael se tornou vencedor. Esse algoritmo, criado pelos belgas Vincent Rijmen e Joan Daemen, foi escolhido com base em qualidades como

segurança, flexibilidade e bom desempenho em software e hardware.

2.6.1 Aspectos principais

A única diferença entre o AES e o Rijndael é que este suporta tamanhos de chave e bloco variando de 128 a 256 bits, com incrementos de 32 bits, enquanto que o AES suporta apenas blocos de tamanho fixo de 128 bits e chaves de 128, 192 ou 256 bits. Ou seja, os dois possuem o mesmo funcionamento, mas o Rijndael pode utilizar uma variação maior de tamanhos de chave e bloco. Neste trabalho serão utilizados blocos e chaves de 128 bits.

Serão definidos agora alguns termos utilizados com frequência no algoritmo. *Estado* é uma matriz de bytes que inicialmente contém a mensagem e que é modificada após cada etapa. *Mensagem* é o texto antes de ser criptografado e cujo conteúdo deve ser acessível apenas ao destinatário. Para manter a segurança, é preciso tornar a mensagem um texto ilegível, também chamado de *texto cifrado,* e isso é feito através da criptografia.

No AES, o tamanho do estado vai depender do tamanho do bloco utilizado, sendo composto de 4 linhas e *Nb* colunas, onde *Nb* é o número é o número de bits do bloco dividido por 32. O algoritmo possui rodadas, também chamadas de iterações, que possuem 4 etapas: *AddRoundKey, SubBytes, ShiftRows* e *MixColumns*. Na última rodada a operação *MixColumns* não é realizada. A sigla *Nr* será usada para designar o número de iterações do algoritmo. O número de iterações varia de acordo com o tamanho da chave usada, sendo 10 para chaves de 128 bits, 12 para chaves de 192 bits e 14 para chaves de 256 bits. O número de colunas da chave será designado por *Nk*, que varia de acordo com o tamanho da chave, sendo o tamanho da chave dividido por 32, ou seja, sendo 4 para chaves de 128 bits, 6 para chaves de 192 bits e 8 para chaves de 256 bits. O algoritmo possui uma chave principal e a partir dela serão geradas *Nr + 1* chaves, chamadas de *chaves da rodada*, pois cada uma será usada em uma rodada diferente. A chave principal é alocada em uma matriz de 4 linhas e *Nk* colunas e cada chave de rodada é agrupada da mesma maneira que o bloco de dados.

Os processos de cifragem e decifragem do AES são mostrados na figura 2.18.

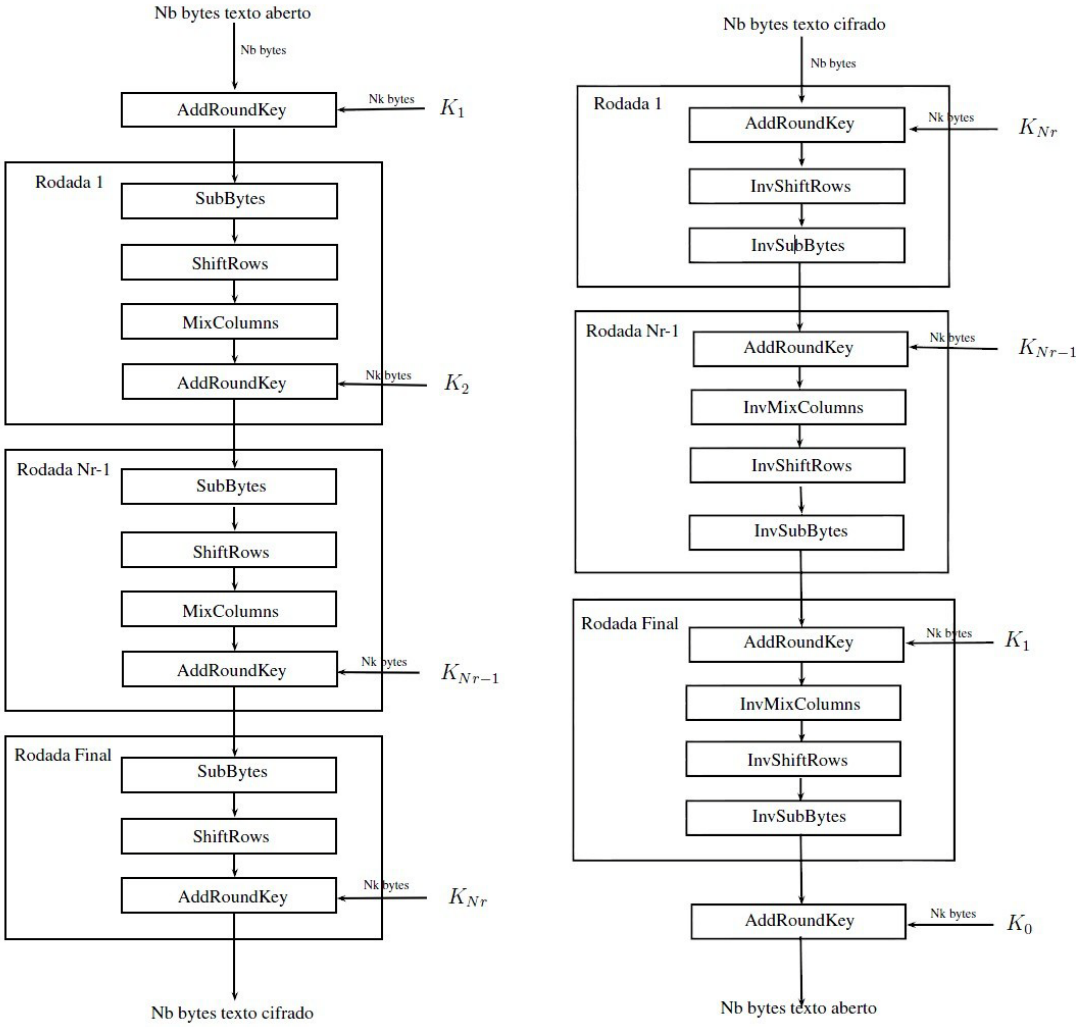

Figura 2.18 Processos de cifragem e decifragem do AES.

2.6.2 Transformação SubBytes

Nesta etapa, cada byte do estado é substituído por outro em uma *S-box* (caixa de substituição), denotada por S_{RD}. Todos os valores dessa caixa são em hexadecimal. Os quatro primeiros e os quatro últimos bits do byte a ser substituído representam em hexadecimal, respectivamente, a linha e a coluna onde se encontra o novo byte. Por exemplo, o valor hexadecimal 6a deverá ser substituído pelo byte que se encontra na linha 6 e na coluna a da S_{RD}, que é o valor 02. A S_{RD} é gerada a partir da composição de duas funções f e g constituídas sobre GF (2^8). Se $a = a_0a_1a_2a_3a_4a_5a_6a_7$, temos que

$$g(a) = a^{-1},$$

onde a^{-1} é o inverso multiplicativo de a em GF (2^8) e

$$f(a) = b = b_0b_1b_2b_3b_4b_5b_6b_7,$$

cujos valores estão mostrados na figura 2.19.

$$b = \begin{bmatrix} b_7 \\ b_6 \\ b_5 \\ b_4 \\ b_3 \\ b_2 \\ b_1 \\ b_0 \end{bmatrix} = \begin{bmatrix} 11111000 \\ 01111100 \\ 00111110 \\ 00011111 \\ 10001111 \\ 11000111 \\ 11100011 \\ 11110001 \end{bmatrix} \times \begin{bmatrix} a_7 \\ a_6 \\ a_5 \\ a_4 \\ a_3 \\ a_2 \\ a_1 \\ a_0 \end{bmatrix} \oplus \begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \end{bmatrix}.$$

Figura 2.19 Valor de b em *SubBytes*.

A figura 2.20 mostra a *S-box* usada no AES. A inversa da operação *SubBytes* chama-se *InvSubBytes* e usa uma *S-box* inversa, denotada por S_{RD}^{-1}, que usa a composição de funções

$$S_{RD}^{-1}[b] = g^{-1}(f^{-1}(b)) = g(f^{-1}(b)).$$

Aplicando a *S-box* no valor 6a obtém-se o valor 02. Logo, aplicando a *S-box* inversa a 02 obtém-se o valor 6a. A figura 2.21 mostra a *S-box* inversa.

		x															
		0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
	0	63	7c	77	7b	f2	6b	6f	c5	30	01	67	2b	fe	d7	ab	76
y	1	ca	82	c9	7d	fa	59	47	f0	ad	d4	a2	af	9c	a4	72	c0
	2	b7	fd	93	26	36	3f	f7	cc	34	a5	e5	f1	71	d8	31	15
	3	04	c7	23	c3	18	96	05	9a	07	12	80	e2	eb	27	b2	75
	4	09	83	2c	1a	1b	6e	5a	a0	52	3b	d6	b3	29	e3	2f	84
	5	53	d1	00	ed	20	fc	b1	5b	6a	cb	be	39	4a	4c	58	cf
	6	d0	ef	aa	fb	43	4d	33	85	45	f9	02	7f	50	3c	9f	a8
	7	51	a3	40	8f	92	9d	38	f5	bc	b6	da	21	10	ff	f3	d2
	8	cd	0c	13	ec	5f	97	44	17	c4	a7	7e	3d	64	5d	19	73
	9	60	81	4f	dc	22	2a	90	88	46	ee	b8	14	de	5e	0b	db
	a	e0	32	3a	0a	49	06	24	5c	c2	d3	ac	62	91	95	e4	79
	b	e7	c8	37	6d	8d	d5	4e	a9	6c	56	f4	ea	65	7a	ae	08
	c	ba	78	25	2e	1c	a6	b4	c6	e8	dd	74	1f	4b	bd	8b	8a
	d	70	3e	b5	66	48	03	f6	0e	61	35	57	b9	86	c1	1d	9e
	e	e1	f8	98	11	69	d9	8e	94	9b	1e	87	e9	ce	55	28	df
	f	8c	a1	89	0d	bf	e6	42	68	41	99	2d	0f	b0	54	bb	16

Figura 2.20 S-box do AES.

	x															
		0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
	0	52	09	6a	d5	30	36	a5	38	bf	40	a3	9e	81	f3	d7	fb
	1	7c	e3	39	82	9b	2f	ff	87	34	8e	43	44	c4	de	e9	cb
	2	54	7b	94	32	a6	c2	23	3d	ee	4c	95	0b	42	fa	c3	4e
	3	08	2e	a1	66	28	d9	24	b2	76	5b	a2	49	6d	8b	d1	25
	4	72	f8	f6	64	86	68	98	16	d4	a4	5c	cc	5d	65	b6	92
	5	6c	70	48	50	fd	ed	b9	da	5e	15	46	57	a7	8d	9d	84
	6	90	d8	ab	00	8c	bc	d3	0a	f7	e4	58	05	b8	b3	45	06
y	7	d0	2c	1e	8f	ca	3f	0f	02	c1	af	bd	03	01	13	8a	6b
	8	3a	91	11	41	4f	67	dc	ea	97	f2	cf	ce	f0	b4	e6	73
	9	96	ac	74	22	e7	ad	35	85	e2	f9	37	e8	1c	75	df	6e
	a	47	f1	1a	71	1d	29	c5	89	6f	b7	62	0e	aa	18	be	1b
	b	fc	56	3e	4b	c6	d2	79	20	9a	db	c0	fe	78	cd	5a	f4
	c	1f	dd	a8	33	88	07	c7	31	b1	12	10	59	27	80	ec	5f
	d	60	51	7f	a9	19	b5	4a	0d	2d	e5	7a	9f	93	c9	9c	ef
	e	a0	e0	3b	4d	ae	2a	f5	b0	c8	eb	bb	3c	83	53	99	61
	f	17	2b	04	7e	ba	77	d6	26	e1	69	14	63	55	21	0c	7d

Figura 2.21 S-box inversa do AES.

2.6.3 Transformação ShiftRows

Esta operação consiste em rotacionar à esquerda as linhas do estado, trocando assim a posição dos bytes. O número de posições a serem rotacionadas depende da linha e do tamanho do bloco que está sendo utilizado. Na figura 2.22 C_i é o número de posições a serem rotacionadas na linha *i* de um bloco com *Nb* colunas. O comportamento dessa transformação é exemplificado na figura 2.23. A operação inversa chama-se *InvShiftRows* e consiste apenas em fazer a mesma rotação, porém à direita.

Nb	$C0$	$C1$	$C2$	$C3$
4	0	1	2	3
6	0	1	2	3
8	0	1	3	4

Figura 2.22 Deslocamento em função de Nb e C_i.

24	40	78	68
6e	63	96	48
13	b0	8e	53
ae	59	01	ee

$\xrightarrow{ShiftRows}$

24	40	78	68
63	96	48	6e
8e	53	13	b0
ee	ae	59	01

Figura 2.23 Exemplo da transformação ShiftRows.

2.6.4 Transformação MixColumns

Nesta etapa, o resultado da operação em uma determinada coluna não influencia o resultado das demais. Porém, a mudança de um byte em uma determinada coluna influencia o resultado na coluna inteira. Os bytes do estado são tratados como polinômios sobre o corpo finito GF (2^8). Essa transformação pode ser representada por uma multiplicação de matrizes. Seja S' o estado após essa transformação e $\odot$ o produto matricial em GF (2^8), então S' será o resultado da multiplicação de uma matriz fixa C pela matriz S que representa o estado, como mostrado na figura 2.24.

$$\begin{bmatrix} S'_{1,1} & S'_{1,2} & S'_{1,3} & S'_{1,4} \\ S'_{2,1} & S'_{2,2} & S'_{2,3} & S'_{2,4} \\ S'_{3,1} & S'_{3,2} & S'_{3,3} & S'_{3,4} \\ S'_{4,1} & S'_{4,2} & S'_{4,3} & S'_{4,4} \end{bmatrix} = \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \odot \begin{bmatrix} S_{1,1} & S_{1,2} & S_{1,3} & S_{1,4} \\ S_{2,1} & S_{2,2} & S_{2,3} & S_{2,4} \\ S_{3,1} & S_{3,2} & S_{3,3} & S_{3,4} \\ S_{4,1} & S_{4,2} & S_{4,3} & S_{4,4} \end{bmatrix}$$

Figura 2.24 Estado em MixColumns.

A inversa dessa operação é chamada *InvMixColumns* e é uma multiplicação que usa uma matriz fixa B, mostrada na figura 2.25, inversa da matriz C, mostrada na figura 2.26, usada na cifragem. Assim, para encontrar o estado inicial S, anterior à operação *MixColumns*, é necessário calcular B $\odot$ S' = S, como visto na figura 2.27.

$$\begin{bmatrix} 0e & 0b & 0d & 09 \\ 09 & 0e & 0b & 0d \\ 0d & 09 & 0e & 0b \\ 0b & 0d & 09 & 0e \end{bmatrix}$$

Figura 2.25 Matriz B.

$$\begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix}$$

Figura 2.26 Matriz C.

$$\begin{bmatrix} 0e & 0b & 0d & 09 \\ 09 & 0e & 0b & 0d \\ 0d & 09 & 0e & 0b \\ 0b & 0d & 09 & 0e \end{bmatrix} \odot \begin{bmatrix} S'_{1,1} & S'_{1,2} & S'_{1,3} & S'_{1,4} \\ S'_{2,1} & S'_{2,2} & S'_{2,3} & S'_{2,4} \\ S'_{3,1} & S'_{3,2} & S'_{3,3} & S'_{3,4} \\ S'_{4,1} & S'_{4,2} & S'_{4,3} & S'_{4,4} \end{bmatrix} = \begin{bmatrix} S_{1,1} & S_{1,2} & S_{1,3} & S_{1,4} \\ S_{2,1} & S_{2,2} & S_{2,3} & S_{2,4} \\ S_{3,1} & S_{3,2} & S_{3,3} & S_{3,4} \\ S_{4,1} & S_{4,2} & S_{4,3} & S_{4,4} \end{bmatrix}.$$

Figura 2.27 InvMixColumns.

2.6.5 Transformação AddRoundKey

Esta é uma operação XOR byte a byte entre o estado e a chave da rodada. Logo, essa transformação opera em cada byte individualmente. Basicamente, se $s_{x,y}$ é um byte do estado e $k_{x,y}$ um byte da chave, então o byte $s_{x,y}$' do novo estado é igual a $s_{x,y} \oplus k_{x,y}$. Como (a $\oplus$ b) $\oplus$ b = a, a transformação *AddRoundKey* é sua própria inversa.

2.6.6 Expansão de chave

Como dito anteriormente, as chaves utilizadas em cada rodada são geradas a partir da chave principal. O algoritmo usado gera Nr + 1 chaves, pois antes da primeira rodada é feita uma operação *AddRoundKey*. A geração de chaves, também conhecida como expansão de chave, resulta em um vetor com palavras de 4 bytes.

Seja cada palavra denotada por w_i, onde *i* é a posição da palavra no vetor, então um exemplo de vetor está na figura 2.28.

w_0	w_1	w_2	w_3	w_4	w_5	w_6	w_7	w_8	w_9	w_{10}	...
chave da rodada 0				chave da rodada 1				...			

Figura 2.28 Vetor composto pelas chaves da rodada.

Inicialmente, as Nk primeiras palavras do vetor são completadas com os bytes da chave principal. Se *i* não é múltiplo de Nk, w_i será obtida através de uma operação XOR entre temp = $w_{[i-1]}$ e $w_{[i-Nk]}$. Caso *i* seja múltiplo de Nk, as seguintes funções são usadas (NIST 2001):

1. *RotWord* – Essa função rotaciona a palavra uma posição à esquerda.
2. *SubWord* – Similar à *SubBytes*, substitui cada byte da palavra pelo byte correspondente na *S-box*.
3. Rcon (j) – É uma constante diferente a cada rodada j. Essa constante é dada por Rcon (j) = (RC [j], 00, 00, 00), onde RC [1] = 1 e RC [j] = 2 × RC [j-1], com a multiplicação sobre GF (2^8). A figura 2.29 mostra o valor de RC [j] a cada rodada.

j	1	2	3	4	5	6	7	8	9	10
RC[j]	01	02	04	08	10	20	40	80	1B	36

Figura 2.29 RC [j] em função da rodada j.

A expansão da chave está escrita em pseudocódigo na figura 2.30.

```
KeyExpansion (byte key [4*Nk], word w [Nb*(Nr+1)], Nk)
begin
        word temp

        i = 0
        while (I < Nk)
                w[i] = word(key[4*i], key[4*i+1], key[4*i+2], key[4*i+3])
                i = i+1
        end while
        I = Nk
        while (i < Nb * (Nr+1)]
                temp = w [i-1]
                if (i mod Nk = 0)
                        temp = SubWord(RotWord(temp)) xor Rcon [i/Nk]
                else if (Nk > 6 and i mod Nk = 4)
                        temp = SubWord(temp)
                end if
                w[i] = w [i-Nk] xor temp
                i = i + 1
        end while
end
```

Figura 2.30 Pseudocódigo da expansão de chave (NIST 2001).

Suponha-se que a chave principal tenha 128 bits e a chave da rodada 4 seja 25 f6 3a c5 78 bb de 47 14 f5 23 d7 a8 cf e2 35. A figura 2.31 exemplifica o funcionamento do algoritmo para o cálculo da primeira palavra da chave da rodada 5.

i	20
temp = w[i-1]	a8cfe235
RotWord	cfe235a8
SubWord	8a9896c2
Rcon(5)	10000000
temp = SubWord ⊕ Rcon(5)	9a9896c2
w[i-Nk]	2f563ac5
w[i] = temp ⊕ w[i-Nk]	b5ceac07

Figura 2.31 Exemplo de expansão de chave.

2.7 Modos de operação do AES

Como o algoritmo AES opera em tamanhos de bloco fixos, para encriptar mensagens maiores que o tamanho do bloco é necessário a utilização de um modo de operação. Vários modos de operação foram criados para prover confidencialidade para mensagens de qualquer tamanho. Os seguintes modos de operação podem ser utilizados:

- *Eletronic Codebook (ECB).*
- *Cipher Block Chaining (CBC).*
- *Cipher Feedback (CFB).*
- *Output Feedback (OFB).*
- *Counter (CTR).*

A entrada para os processos CBC, CFB e OFB requer, além do *plaintext*, um bloco de dados chamado Vetor de Inicialização (VI), que é um bloco de bits necessário aos modos de operação para a produção de um bloco único independente

de outros blocos produzidos pela mesma chave. O VI é utilizado como um passo inicial nos processos de encriptação e desencriptação da mensagem.

O VI não precisa ser secreto. No entanto, para os modos CBC e CFB, o VI deve ser imprevisível. Para o modo OFB, VIs únicos devem ser utilizados para cada execução do processo de encriptação (NIST 2001).

2.7.1 Modo ECB

É o modo de encriptação mais simples. A mensagem é dividida em blocos e cada bloco é encriptado separadamente. Enquanto isso leva a um amplo paralelismo, blocos de *plaintext* idênticos são criptografados em blocos de *ciphertext* idênticos; logo, este modo não esconde o padrão dos dados bem. Este modo é ilustrado na figura 2.32.

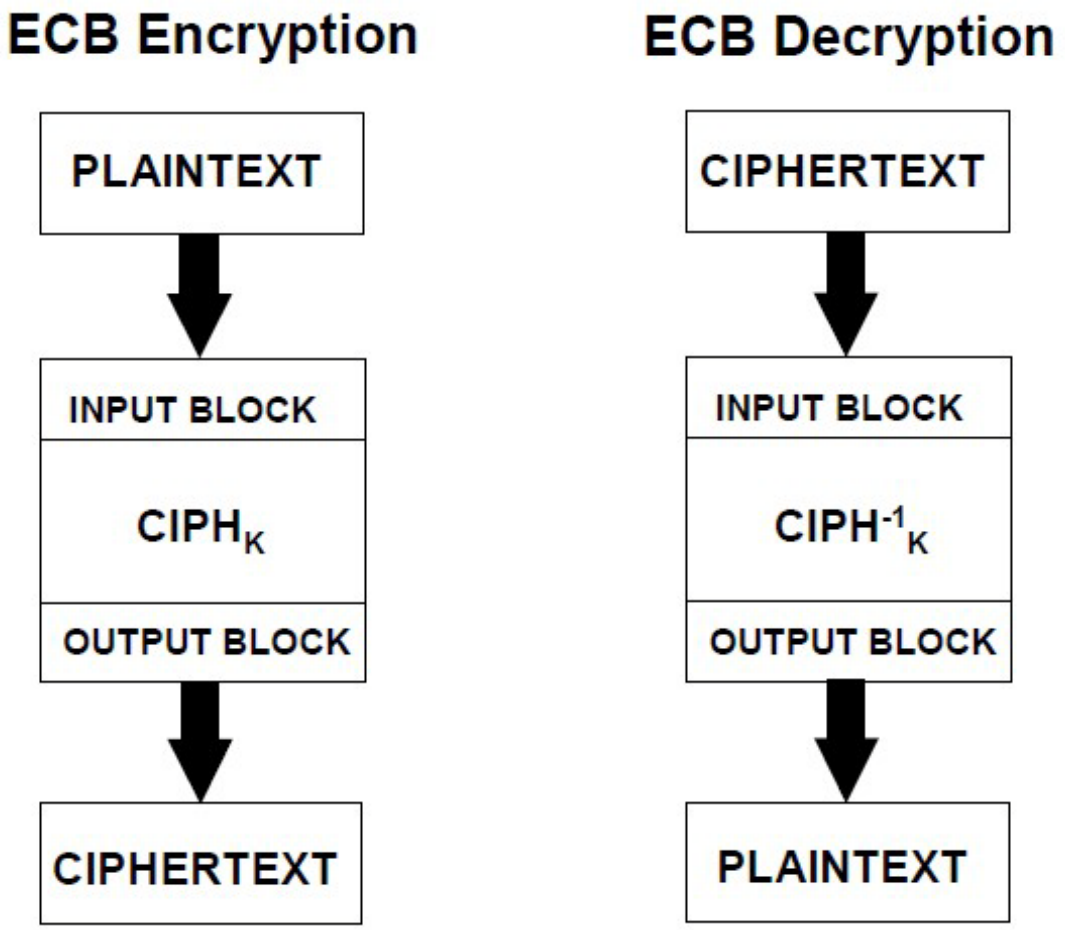

Figura 2.32 Modo ECB (NIST 2001).

2.7.2 Modo CBC

Neste modo, cada bloco de *plaintext* recebe a operação XOR (ou-exclusivo) com o *ciphertext* do bloco anterior antes de ser encriptado. Deste modo, cada bloco de *ciphertext* é dependente de todos os blocos de *plaintext* processados até aquele ponto. Além disso, para tornar cada mensagem única, um VI deve ser utilizado no primeiro bloco.

No processo de encriptação, o primeiro bloco de entrada é formado através da realização de um XOR do primeiro bloco de *plaintext* com o VI. A cifra de encriptação é então aplicada nesse primeiro bloco de entrada, produzindo assim o primeiro bloco de *ciphertext*. Esse bloco também recebe um XOR com o segundo bloco de *plaintext* para produzir o segundo bloco de entrada. A cifra é aplicada novamente, produzindo assim o segundo bloco de *ciphertext*, e assim por diante. Cada bloco de *plaintext* recebe a operação XOR com o bloco de *ciphertext* anterior para produzir um novo bloco de entrada.

No processo de desencriptação, o algoritmo de desencriptação é aplicado para o primeiro bloco de *ciphertext* e o bloco resultante recebe um XOR com o VI para produzir o primeiro bloco de *plaintext*. O algoritmo de desencriptação é aplicado também no segundo bloco de *ciphertext*, e o bloco resultante recebe um XOR com o primeiro bloco de *ciphertext*, produzindo assim o segundo bloco de *plaintext*. Em geral, para recuperar qualquer bloco de *plaintext*, com exceção do primeiro, o algoritmo de desencriptação é aplicado no bloco de *ciphertext* correspondente e o bloco resultante recebe um XOR com o bloco de *ciphertext* anterior.

Como no processo de encriptação cada operação depende do resultado da anterior, então ele não é paralelizável. No entanto, o processo de desencriptação precisa apenas dos blocos de *ciphertext* que já estão disponíveis, então várias operações de desencriptação podem ser realizadas em paralelo.

Este modo está ilustrado na figura 2.33.

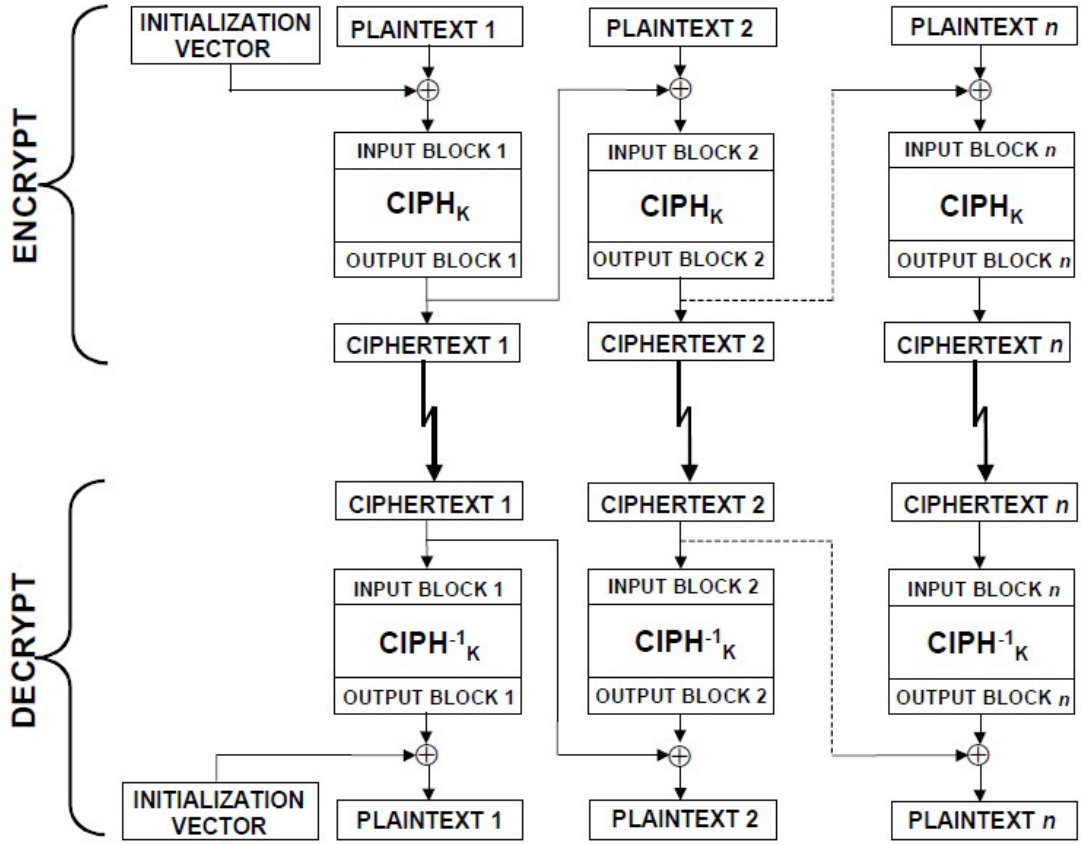

Figura 2.33 Modo CBC (NIST 2001).

2.7.3 Modo CFB

Este modo requer um inteiro *s* tal que tem $1 \leq s \leq b$, onde *b* é o tamanho do bloco, em bits.

No processo de encriptação CFB, o primeiro bloco de entrada é o VI e o algoritmo é aplicado no VI para produzir o primeiro bloco de saída. O primeiro segmento de *ciphertext* é produzido ao realizar um XOR entre o primeiro segmento de *plaintext* e os *s* bits mais significantes do primeiro bloco de saída. O restante dos bits é descartado. Os *b-s* bits menos significantes do VI são então concatenados com os s bits do primeiro segmento de *ciphertext* para formar o segundo bloco de entrada. O processo é repetido com os sucessivos blocos de entrada até que um segmento de *ciphertext* é produzido de cada segmento de *plaintext*.

Em geral, cada bloco de entrada é cifrado para produzir um bloco de saída. Os *s* bits mais significantes de cada bloco de saída recebem um XOR com o

segmento de *plaintext* correspondente para formar um segmento de *ciphertext*.

Na desencriptação, o VI é o primeiro bloco de entrada e cada bloco de entrada sucessivo é formado como na encriptação, concatenando os *b-s* bits menos significativos do bloco de entrada anterior com os *s* bits mais significativos do *ciphertext* anterior. O algoritmo é então aplicado em cada bloco de entrada para produzir os blocos de saída. Os *s* bits mais significativos dos blocos de saída recebem um XOR com os segmentos de *ciphertext* correspondentes para recuperar os segmentos de *plaintext* correspondentes.

Este modo, como no CBC, não permite a utilização de paralelismo na encriptação, somente na desencriptação. O funcionamento do modo está ilustrado na figura 2.34.

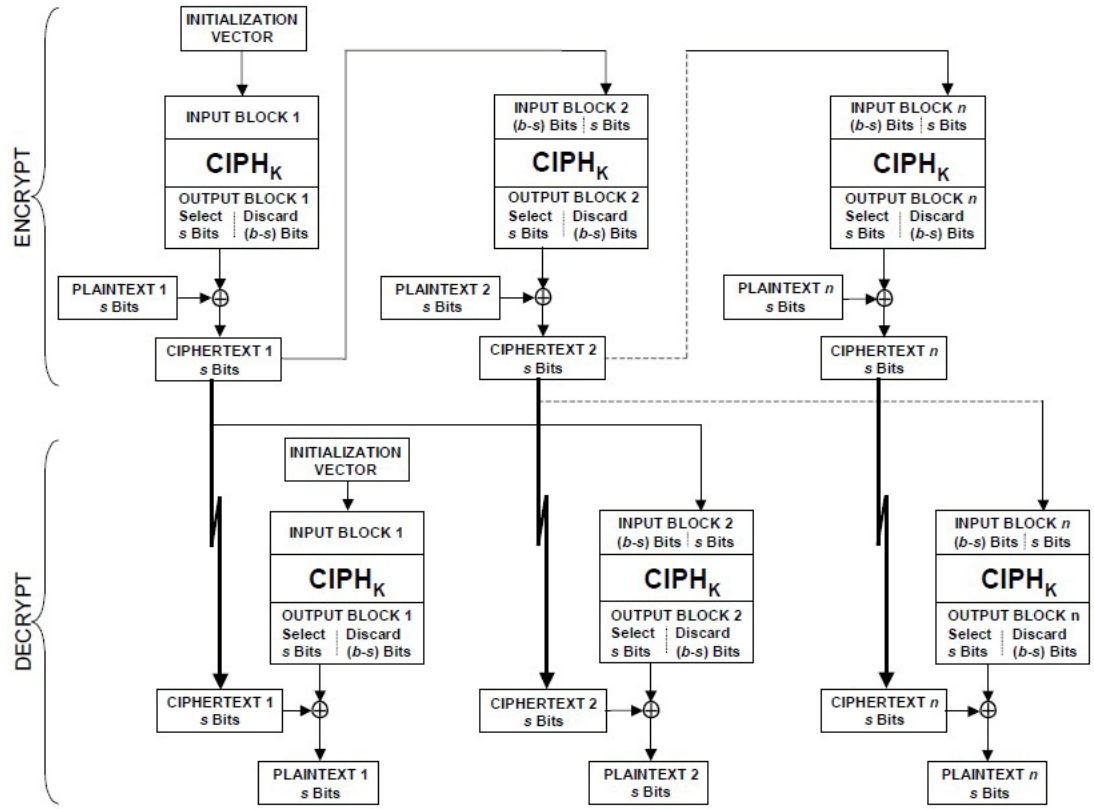

Figura 2.34 Modo CFB (NIST 2001).

2.7.4 Modo OFB

Este é um modo de confidencialidade que retrata a iteração do algoritmo de cifragem em um VI para gerar uma sequência de blocos de saída em que são

realizados XOR com o *plaintext* para gerar o *ciphertext*, e vice-versa. Este modo requer que o VI seja único em cada execução do modo sob certa chave.

No processo de encriptação, o VI é transformado pelo algoritmo para produzir o primeiro bloco de saída. O primeiro bloco de saída recebe um XOR com o primeiro bloco de *plaintext* para produzir o primeiro bloco de *ciphertext*. O algoritmo é invocado no primeiro bloco de saída para produzir o segundo bloco de saída. O segundo bloco de saída recebe um XOR com o segundo bloco de *plaintext* para produzir o segundo bloco de *ciphertext*, e assim por diante. Logo, os sucessivos blocos de saída são produzidos ao aplicar o algoritmo ao bloco de saída anterior, e os blocos de saída recebem um XOR com os blocos de *plaintext* correspondentes para produzir os blocos de *ciphertext*.

Na decriptação, o VI é transformado pelo algoritmo no primeiro bloco de saída. Este bloco então recebe um XOR com o primeiro bloco de *ciphertext* para produzir o primeiro bloco de *plaintext*. Logo, os sucessivos blocos de saída são produzidos ao aplicar o algoritmo nos blocos de saída anteriores, e os blocos de saída recebem um XOR com os blocos de *ciphertext* correspondentes para recuperar os blocos de *plaintext*.

Este modo não pode ser paralelizável. Este modo está ilustrado na figura 2.35.

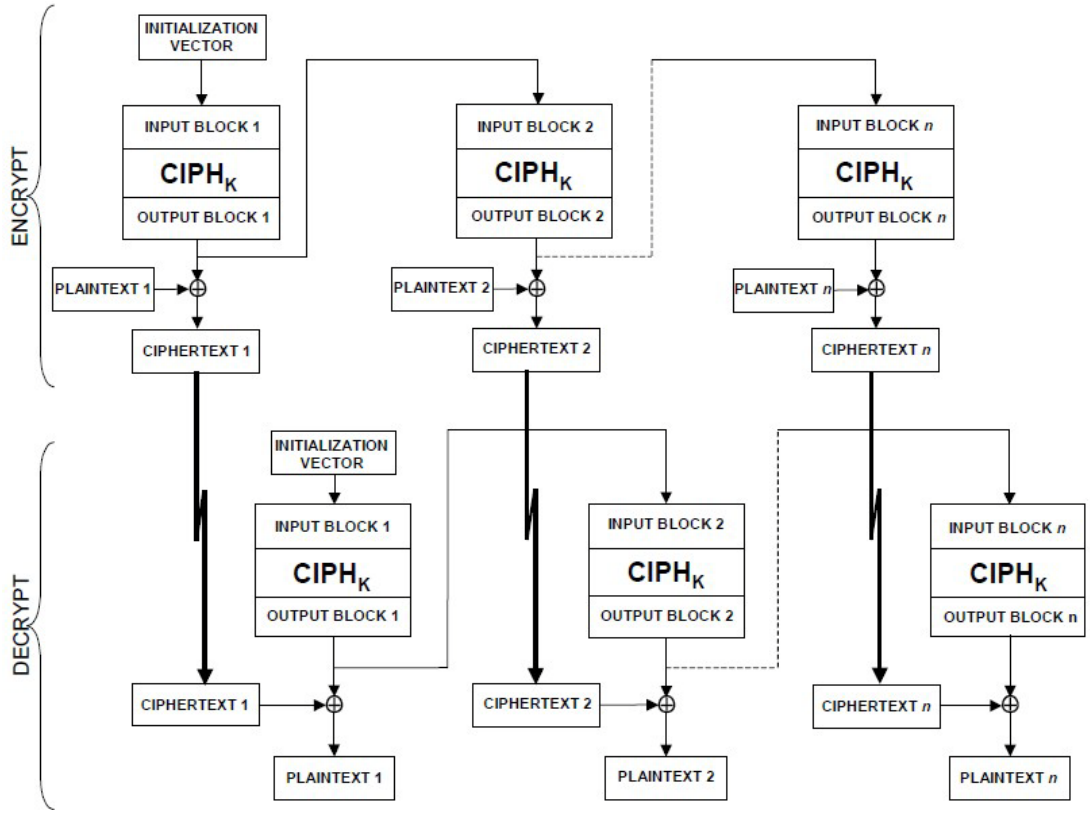

Figura 2.35 Modo OFB (NIST 2001).

2.7.5 Modo CTR

Este modo utiliza o algoritmo em um conjunto de blocos de entrada, chamados contadores, para produzir uma sequência de blocos de saída que então recebem um XOR com o *plaintext* para produzir o *ciphertext* e vice-versa. A sequência de contadores deve ter a propriedade de que cada bloco na sequência é diferente de todos os outros blocos. Esta condição não é restrita a uma única mensagem, ou seja, em todas as mensagens que são encriptadas usando certa chave, os contadores devem ser diferentes.

No processo de encriptação, o algoritmo é chamado em cada bloco de contador, e os blocos resultantes recebem XOR com os blocos correspondentes de *plaintext* para produzirem os blocos de *ciphertext*.

Na decriptação, o algoritmo é chamado em cada bloco de contador, e os blocos resultantes recebem XOR com os blocos correspondentes de *ciphertext* para produzirem os blocos de *ciphertext.*

Tanto no processo de encriptação quanto na desencriptação, o algoritmo pode

ser executado em paralelo. O bloco de *plaintext* que corresponde a um particular bloco de *ciphertext* pode ser recuperado independentemente de outros blocos de *plaintext* se o bloco de contador correspondente for determinado.

Este modo é altamente paralelizável e é o modo empregado neste trabalho. Ele está ilustrado na figura 2.36.

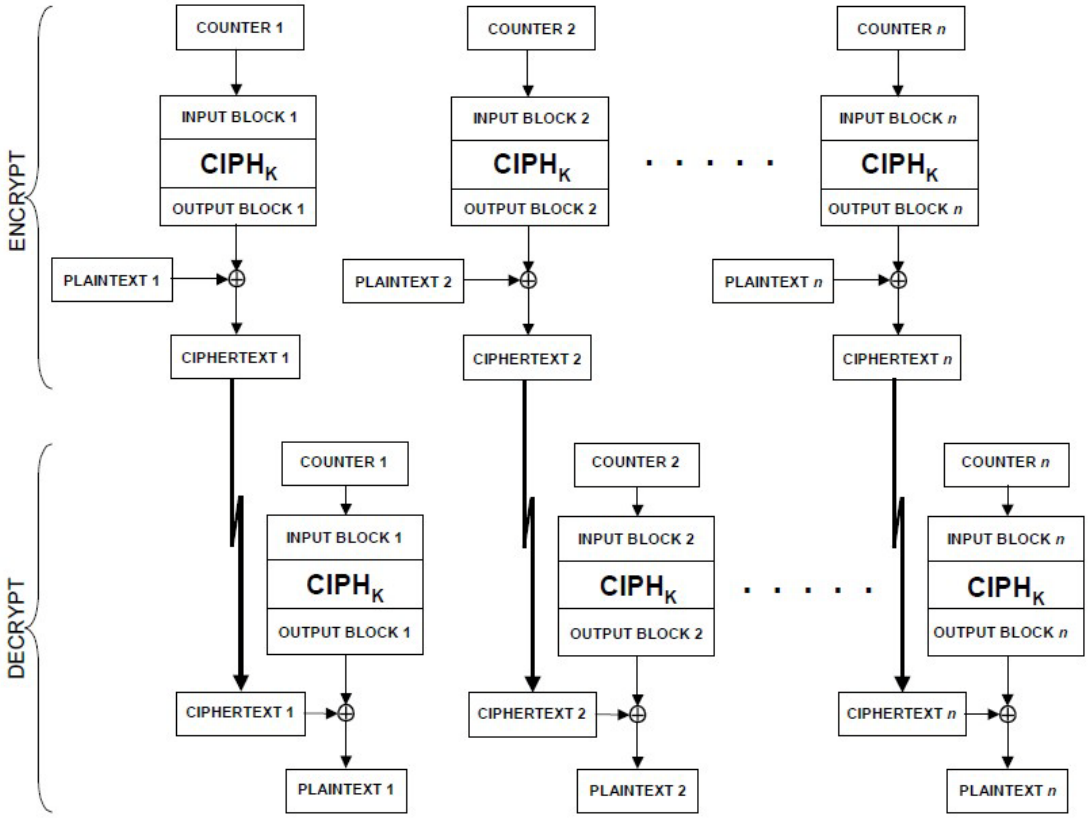

Figura 2.36 Modo CTR (NIST 2001).

2.8 O algoritmo MD5

O MD5 é uma função criptográfica *hash* que recebe uma mensagem de valor arbitrário e retorna um *hash* de 128 bits. É muito utilizado na verificação de *logins* e senhas, integridade de arquivos, sistemas Peer-2-peer (P2P), etc.

A entrada do algoritmo é dividida em pedaços de 512 bits, sendo cada pedaço dividido em 16 blocos de 32 bits. Se o tamanho da mensagem não for divisível por 512, então um complemento é utilizado. Primeiro um bit 1 é adicionado no fim da mensagem. Em seguida, são adicionados quantos bits 0 forem necessários para tornar

a mensagem até 64 bits menor do que um múltiplo de 512. Os 64 bits restantes são preenchidos com um inteiro representando o tamanho da mensagem original.

O algoritmo funciona em um estado de 128 bits, dividido em quatro palavras de 32 bits denominadas A, B, C e D. O algoritmo funciona em cada bloco de 512 bits de uma vez, sendo que cada bloco modifica o estado. O processamento de um bloco consiste em quatro fases, sendo que cada fase é composta de 16 operações similares baseadas em funções não lineares, adição modular e rotação de bits à esquerda. A figura 2.37 ilustra uma operação dentro de uma fase, onde $<<<_s$ denota rotação de bits à esquerda com *s* variando em cada operação, M_i representa um bloco de 32 bits, F representa uma função não linear, K_i denota uma constante 32 bits diferente para cada operação e ⊞ representa adição modular 2^{32}.

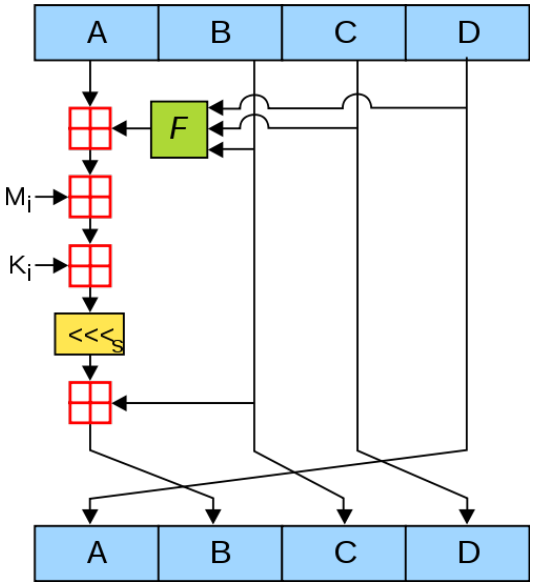

Figura 2.37 Uma operação MD5.

As seguintes funções não lineares são utilizadas uma para cada fase:

- Fase 1: F (X,Y,Z) = (X ^ Y) v (¬X ^ Z)
- Fase 2: G (X,Y,Z) = (X ^ Z) v (Y ^ ¬Z)
- Fase 3: H (X,Y,Z) = X xor Y xor Z
- Fase 4: I (X,Y,Z) = Y xor (X v ¬Z)

Segue a descrição do funcionamento do algoritmo:

1. Vetor de inicialização: quatro variáveis A, B, C e D de 32 bits são

inicializadas com os seguintes valores determinados pela definição do algoritmo:

A. 0x01234567

B. 0x89abcdef

C. 0xfedcba98

D. 0x76543210

2. Verificação do tamanho de entrada e complementação, se necessário.

3. Laço principal é executado uma vez em cada bloco M de 512 bits.

Segue abaixo o pseudocódigo do algoritmo MD5:

```
//Definir r como o seguinte, sendo que r especifica o deslocamento de bits
//por fase
var int[64] r, k
r[ 0..15] := {7, 12, 17, 22,  7, 12, 17, 22,  7, 12, 17, 22,  7, 12, 17, 22}
r[16..31] := {5,  9, 14, 20,  5,  9, 14, 20,  5,  9, 14, 20,  5,  9, 14, 20}
r[32..47] := {4, 11, 16, 23,  4, 11, 16, 23,  4, 11, 16, 23,  4, 11, 16, 23}
r[48..63] := {6, 10, 15, 21,  6, 10, 15, 21,  6, 10, 15, 21,  6, 10, 15, 21}

//Utilizar a parte inteira dos senos de inteiros como constantes:
for i from 0 to 63
    k[i] := floor(abs(sin(i + 1)) * 2 pow 32)

//Iniciar as variáveis:
var int h0 := 0x67452301
var int h1 := 0xEFCDAB89
var int h2 := 0x98BADCFE
var int h3 := 0x10325476

//Pre-processamento:
append "1" bit to message
append "0" bits until message length in bits ≡ 448 (mod 512)
append bit length of message as 64-bit little-endian integer to message

//Processar a mensagem em pedaços sucessivos de 512-bits:
for each 512-bit chunk of message
    break chunk into sixteen 32-bit little-endian words w(i), 0 ≤ i ≤ 15

    //Inicializar o valor do hash para este pedaço:
    var int a := h0
    var int b := h1
    var int c := h2
    var int d := h3
```

```
//Loop principal:
for i from 0 to 63
    if 0 ≤ i ≤ 15 then
        f := (b and c) or ((not b) and d)
        g := i
    else if 16 ≤ i ≤ 31
        f := (d and b) or ((not d) and c)
        g := (5*i + 1) mod 16
    else if 32 ≤ i ≤ 47
        f := b xor c xor d
        g := (3*i + 5) mod 16
    else if 48 ≤ i ≤ 63
        f := c xor (b or (not d))
        g := (7*i) mod 16

    temp := d
    d := c
    c := b
    b := b + leftrotate((a + f + k[i] + w[g]) , r[i])
    a := temp

//Adicionar este pedaço do hash ao resultado:
h0 := h0 + a
h1 := h1 + b
h2 := h2 + c
h3 := h3 + d

var int digest := h0 append h1 append h2 append h3 //expressado como
//little-endian
```

Para demonstrar que uma pequena mudança no valor de entrada altera o valor *hash* gerado, foi utilizado o algoritmo MD5 para gerar o *hash* da palavra Ibilce, gerando assim o *hash* 5c7a0e154a18a483c1630931594202ac. Trocando apenas a letra l pela vogal u, ficando assim a palavra Ibiuce, o *hash* se torna d85aeec150acf1569fd7e0e3c665e4ce, totalmente diferente do original.

2.9 Considerações finais

Neste capítulo apresentou-se uma introdução teórica sobre os principais conceitos relacionados ao projeto, tais como criptografia, computação de alto desempenho, CUDA e unidades gráficas de processamento.

Capítulo 3 – Projeto Desenvolvido

3.1 Considerações iniciais

Este capítulo tem como objetivo mostrar o desenvolvimento do projeto que foi elaborado utilizando os conceitos apresentados no capítulo anterior. Primeiramente é feita uma descrição geral do projeto e os trabalhos relacionados. Uma descrição detalhada do problema é mostrada, assim como as soluções implementadas.

3.2 Descrição do projeto

O projeto consiste em adaptar os algoritmos executados de forma sequencial na CPU para execução paralela na GPU. Alguns algoritmos de criptografia são paralelizáveis e pode haver ganho de desempenho na utilização de GPUs.

3.2.1 AES na GPU

O algoritmo de criptografia AES é baseado em blocos e a utilização dele em tamanhos de mensagens grandes é computacionalmente intensivo e altamente paralelizável, dependendo do modo de operação utilizado. Assim, um algoritmo paralelizável foi escolhido e sua implementação foi feita em CUDA, linguagem

própria para GPUs da NVIDIA. Além disso, a utilização dos recursos da GPU como coprocessadores permite uma melhor utilização da CPU. A implementação do algoritmo AES em CUDA utilizando uma GPU Geforce 8600GT 256MB da NVIDIA obteve um desempenho aproximadamente 3,2 vezes superior para o maior tamanho de mensagem que a mesma implementação em uma CPU Athlon64 3500+ de 2 GHz, enquanto que a utilização de uma GPU mais moderna Geforce GT 240 obteve um desempenho superior a 19 vezes em relação a sua implementação na CPU.

3.2.2 MD5 na GPU

Para a verificação de desempenho do algoritmo MD5, foi implementado um sistema de força bruta para verificação de senha. Na autenticação de senha utilizando MD5, o sistema armazena o hash da senha. Desde modo, a autenticação se dá ao transformar em hash a entrada do usuário e comparando com a armazenada pelo sistema. O sistema de força bruta trabalha exatamente da mesma maneira, tentando todas as senhas possíveis, sendo que uma hora a senha correspondente será encontrada.

Para realizar o teste, foi utilizada a mesma CPU do teste utilizado com o AES. Porém, o teste do MD5 foi realizado somente na GPU Geforce GT 240 por indisponibilidade de outro hardware. O desempenho da versão em CUDA chega a ser mais de 60 vezes maior que a versão sequencial.

3.3 Trabalhos relacionados

Um dos artigos mais antigos que envolvem criptografia e GPU é datado de 1999 (KEDEM e ISHIHARA 1999). Ele utilizou uma GPU experimental chamada *PixelFlow* que consistia de vários processadores de 8 bit rodando a 100 MHz. Os autores estudaram um método de quebrar as senhas de UNIX e conseguiram quebrar a maioria das senhas em dois ou três dias.

As GPUs antigas não eram adequadas para trabalho em criptografia devido à

falta de programabilidade e suporte de processamento de inteiros. Isso mudou nos últimos anos com as ultimas GPUs feitas pela NVIDIA e pela ATI. Vários artigos recentes lidam com a implementação do AES em GPUs da NVIDIA, porém nem todos utilizaram a arquitetura CUDA (HARRISON e WALDRON 2007, FIORESE e BUDAK 2007, MANAVSKI 2007). Todos os artigos se concentraram em implementar o núcleo do algoritmo na GPU, deixando o processo de expansão da chave na CPU. Além disso, todos os artigos utilizaram modos paralelos do AES, como o ECB e o CTR.

3.4 Implementação do algoritmo

Esta seção mostrará como os problemas foram resolvidos, mostrando o funcionamento das soluções e o modo como foram implementadas.

3.4.1 Algoritmo AES

Já que o objetivo era comparar o desempenho do algoritmo executado na CPU e na GPU, então um algoritmo sequencial foi adaptado para GPU, para ter uma real comparação do desempenho entre a versão CPU e GPU. A implementação sequencial usa orientação por bytes e é apropriada para um paralelismo em nível de byte na GPU. Além disso, ela utiliza 9 tabelas de tamanho 16x16 para poupar tempo de computação, utilizando assim 2304 bytes de memória.

A implementação tem duas funções principais chamadas encriptar e desencriptar. Estas funções recebem um bloco de 128 bits de *plaintext* ou um *ciphertext*, uma chave expandida e geram como saída um bloco de 128 bits encriptado ou desencriptado. Em alto nível, as funções recebem uma chave de 128 bits e uma mensagem de tamanho variável. Elas dividem a mensagem em blocos de 128 bits e passam os blocos para as funções Encriptar e Desencriptar. Isto naturalmente leva a um alto paralelismo na implementação da GPU, já que as threads da GPU trabalham em um subconjunto de dados e não há dependência entre as threads, executando as funções Encriptar ou Desencriptar em paralelo, como visto na

figura 3.1.

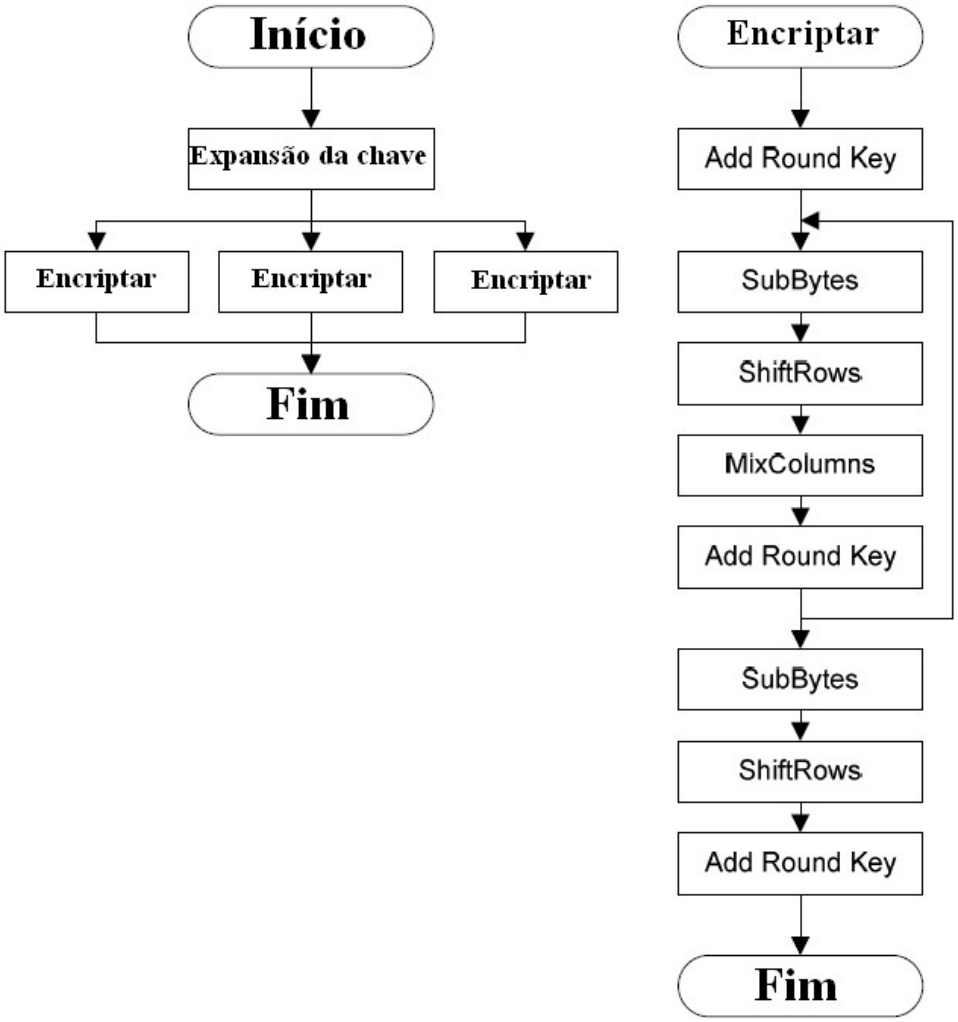

Figura 3.1 Algoritmo AES paralelo.

A expansão da chave é feita na CPU e a chave expandida é passada a GPU como parâmetro. Isto acontece porque a operação é serial e é aplicada em todas as threads igualmente.

Para otimizar o acesso a memória global, é necessário agrupar o acesso aos dados para permitir que a GPU faça leituras de memória amplas. Para alcançar isso, a execução das threads é feita em três estágios, como mostrado na figura 3.2.

Figura 3.2 Estágios de execução das threads.

Já que todas as threads acessam os dados da memória global antes de processá-los, é fácil ordenar os acessos de modo que o uso mais eficiente da largura de banda de memória é utilizada. Isto é feito de uma maneira orientada aos dados, de maneira que qualquer mudança na estrutura dos grids ou blocos não requer qualquer mudança no código. Uma vez que os dados são carregados na memória compartilhada, eles podem ser acessados centenas de vezes mais rapidamente do que na memória global. Já que cada thread carrega um conjunto de dados da memória global que não necessariamente será utilizado, todas as threads devem sincronizar antes que a memória compartilhada possa ser usada. A mesma coisa acontece ao escrever de volta da memória compartilhada pra memória global.

Outra otimização que é útil é a utilização da memória constante. A versão de CPU utilizada utiliza várias tabelas de 16x16 bytes para realizar pesquisas. Já que essas tabelas são constantes e em comum para todas as threads, é possível carregar elas na memória constante da GPU. Esta memória utiliza um cache e ela pode acomodar todos os bytes das tabelas. Uma vez na memória constante, quase não há

penalidade de tempo em acessar os dados.

3.4.2 Algoritmo MD5

O funcionamento de um sistema de força bruta para recuperação de senhas é muito simples. Basta percorrer todas as palavras possíveis de certo alfabeto, transformando as palavras em hashes e comparando com o hash desejado. Este tipo de problema é adaptável à programação paralela em CUDA, já que cada thread pode fazer esse trabalho individualmente, tornando assim o desempenho muito superior. Sendo assim, N threads checam N senhas simultaneamente, sem necessidade de comunicação entre as threads. Tal funcionamento é mostrado na figura 3.3.

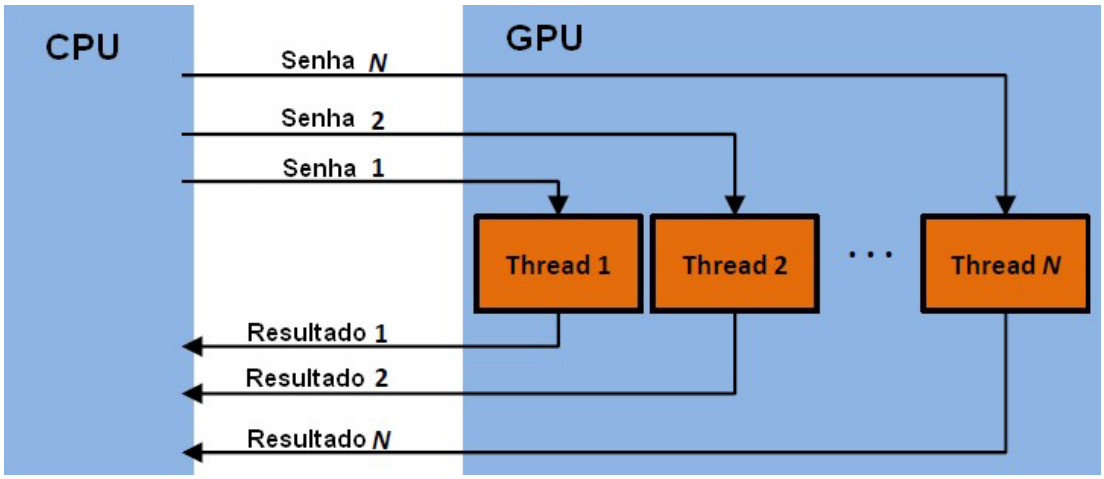

Figura 3.3 Funcionamento do MD5 na GPU.

Assim sendo, o que a N-ésima thread faz é:

1. Gerar a N-ésima senha.
2. Transformar essa senha em um hash MD5.
3. Se o hash estiver correto, retornar N para a CPU.

Nota-se que a CPU não precisa realizar nenhuma tarefa com relação ao funcionamento do MD5; tudo é feito na GPU. Os dados necessários para o funcionamento do algoritmo são copiados para a memória constante da GPU, tornando o acesso aos dados muito mais rápidos. Na implementação do programa foram utilizados 128 threads por bloco e 8192 blocos.

Para um sistema de força bruta funcionar, é necessário ter um conjunto de

caracteres, números ou símbolos que possam formar a palavra original. No caso dessa implementação, foram utilizadas apenas as letras minúsculas e maiúsculas do alfabeto. Em sistemas mais sofisticados realmente voltados para o propósito de recuperação de senhas, é útil adicionar os números de 0 a 9. Esse conjunto de letras que formam as palavras é chamado de alfabeto. Nesse caso, o alfabeto consiste no conjunto “abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ”. Outra variável utilizada na implementação é o tamanho possível da palavra original. Foram utilizadas palavras com tamanho mínimo de quatro caracteres e máximo de oito. É possível a utilização de qualquer tamanho de palavra, mas com esses números já é possível analisar o desempenho.

O programa também necessita do hash da palavra. Assim, o programa é baseado em três dados de entrada: alfabeto, tamanho da palavra e hash da palavra. Todos esses dados são passados na hora da compilação.

Supondo que a palavra consista de quatro caracteres. Sendo assim, o programa percorre todas as combinações possíveis de palavras com quatro caracteres, utilizando todas as letras do alfabeto. Portanto, começa com a palavra aaaa, depois baaa, caaa e assim percorre até chegar à palavra ZZZZ.

Toda a implementação do algoritmo MD5 foi tirada diretamente da descrição do algoritmo MD5 (RFC 1321 1992).

3.5 Considerações finais

Neste capítulo foram mostrados alguns trabalhos relacionados na área, além da descrição do projeto e as soluções propostas. Viu-se que os problemas têm soluções viáveis e mostrou-se que as implementações satisfazem o propósito do trabalho.

Capítulo 4 – Testes, resultados e avaliação dos resultados

4.1 Considerações iniciais

Neste capítulo serão mostrados os testes realizados para verificar se os algoritmos estão funcionando corretamente, avaliar seus desempenhos e compará-los, a fim de verificar se a solução proposta oferece alguma vantagem em relação à versão de CPU.

4.2 Hardware e Software

Para a realização dos testes foi utilizado um computador com processador AMD Athlon64 3500+ de 2 GHz com 3 GB de memória RAM. O sistema operacional utilizado foi o Windows 7 x64 e o software utilizado para desenvolvimento foi o Visual Studio 2008.

Para os testes do algoritmo AES duas GPUs foram utilizadas: Geforce 8600 GT e Geforce GT 240. A figura 4.1 mostra algumas das características da GPU Geforce 8600 GT enquanto que a figura 4.2 mostra as características da Geforce GT 240. É possível ver que os *clocks* da GPU são similares, 1.36GHz e 1.34GHz, ou seja, são mais lentas que a CPU utilizada. No entanto, o desempenho das GPUs é maior devido ao paralelismo utilizado, já que a 8600 GT possui 4 multiprocessadores com 8 *cores* cada, totalizando 32 *cores*, enquanto que a GT 240, mais moderna,

possui 12 multiprocessadores com 8 *cores* cada, totalizando 96 *cores*.

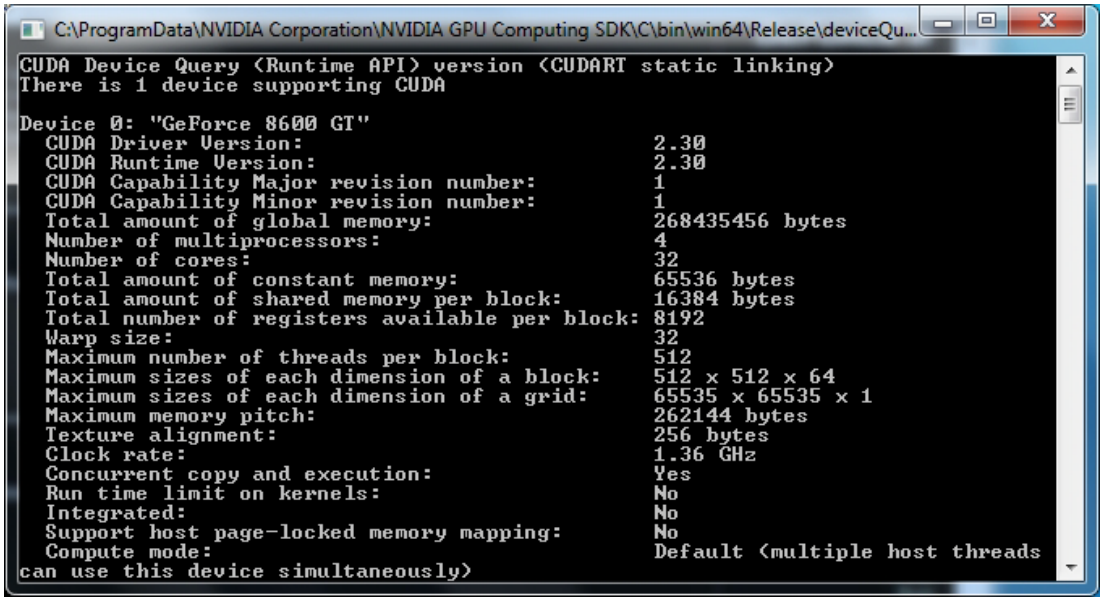

Figura 4.1 Características da GPU Geforce 8600 GT.

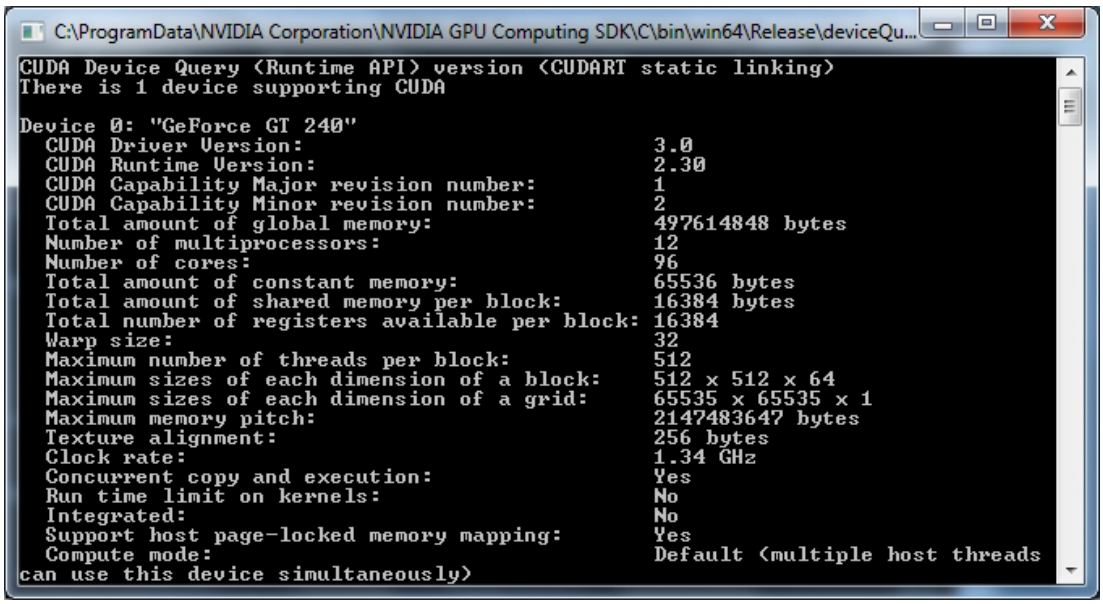

Figura 4.2 Características da GPU Geforce 240 GT.

4.3 Testes realizados

Esta seção mostra os resultados das implementações propostas, tanto do algoritmo AES quanto do MD5. É a maneira mais plausível de analisar os resultados obtidos com as implementações.

4.3.1 Testes do AES

O primeiro aspecto a ser verificado é a corretude do algoritmo. Para assegurar que o algoritmo não foi modificado e está funcionando como deveria em ambas as implementações de CPU e GPU foram testadas de acordo com os seguintes aspectos:

1. Ter certeza que encriptando uma string conhecida gera um texto cifrado também conhecido. O processo de desencriptação também é verificado. O texto e o texto cifrado para testes são encontrados em (NIST 2009).
2. Assegurar que desencriptando um texto cifrado gera um texto original.

Após a verificação da corretude, o algoritmo verifica o desempenho. Já que o propósito de qualquer cifra é rapidamente encriptar dados de entrada, então a métrica utilizada para desempenho é o tempo total de execução do algoritmo.

Para comparar a versão da CPU com a versão da GPU, foi dividido o tempo da CPU com o tempo da GPU. Embora seja um bom indicador dos benefícios que a GPU pode trazer, é necessário ser cauteloso ao comparar o resultado obtido com outras publicações. Variações no hardware, renovações das GPUs, diferenças nas implementações de CPU e GPU provocarão mudanças na razão do tempo de execução entre GPU e CPU. As figuras 4.3, 4.4, 4.5 e 4.6 ilustram o programa em execução utilizando a GPU Geforce 8600 GT.

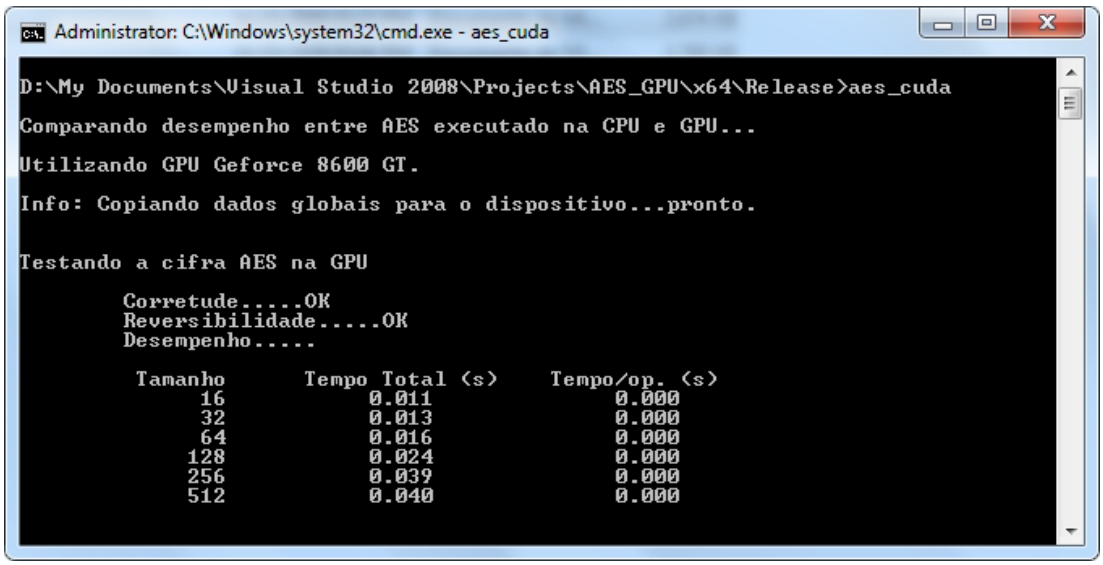

Figura 4.3 Programa em execução testando a GPU Geforce 8600 GT.

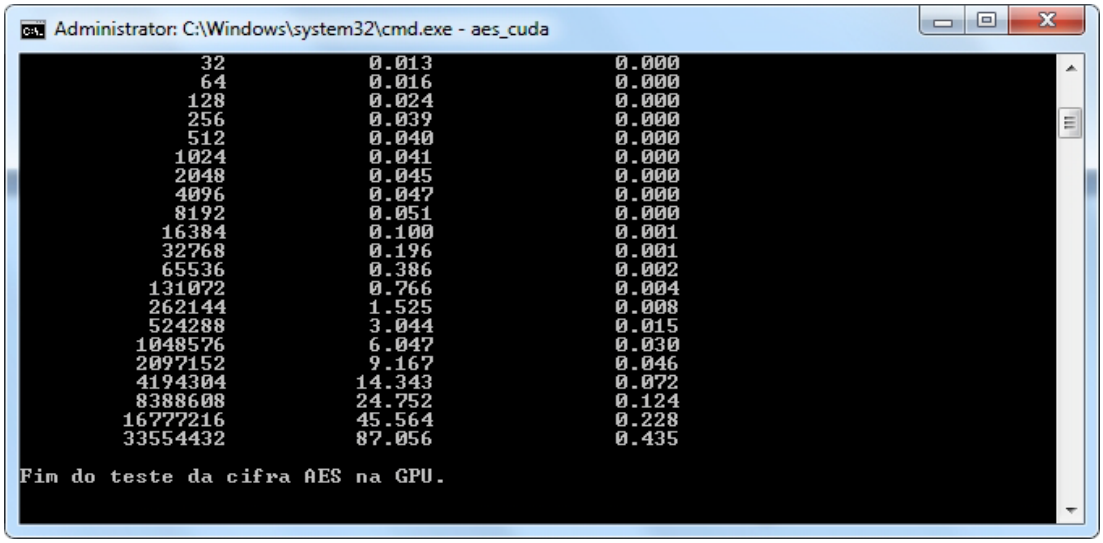

Figura 4.4 Terminando o teste da GPU Geforce 8600 GT.

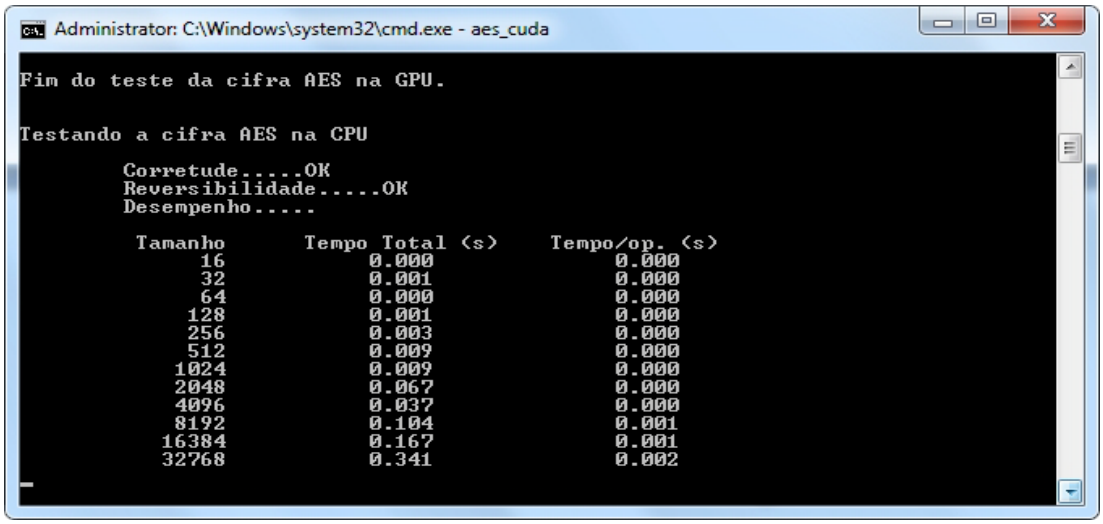

Figura 4.5 Iniciando o teste na CPU.

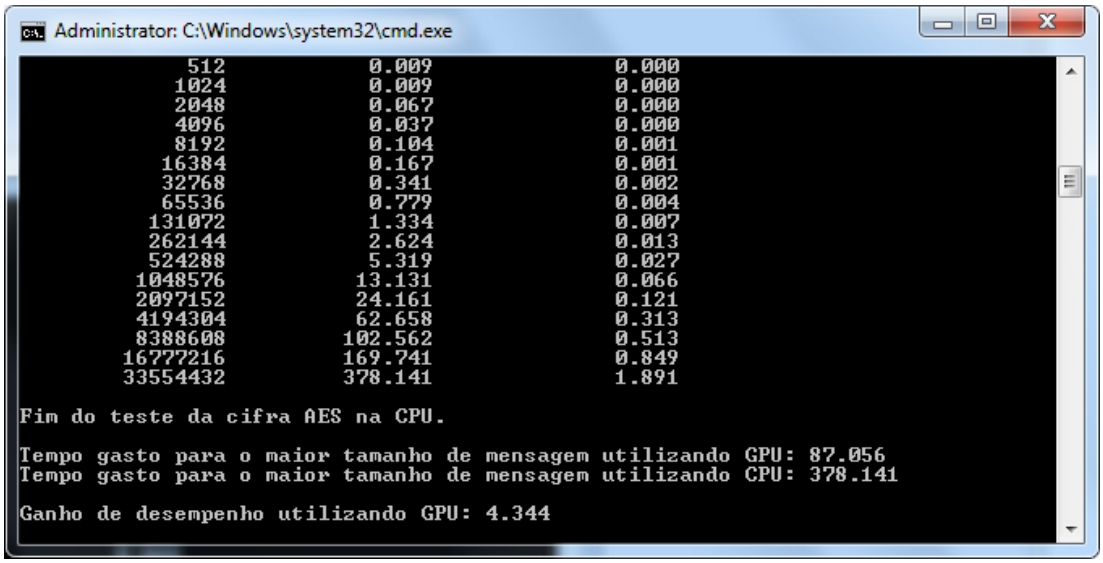

Figura 4.6 Fim da execução do programa.

As figuras 4.7, 4.8, 4.9 e 4.10 ilustram o desempenho do programa sendo executado na GPU Geforce GT 240.

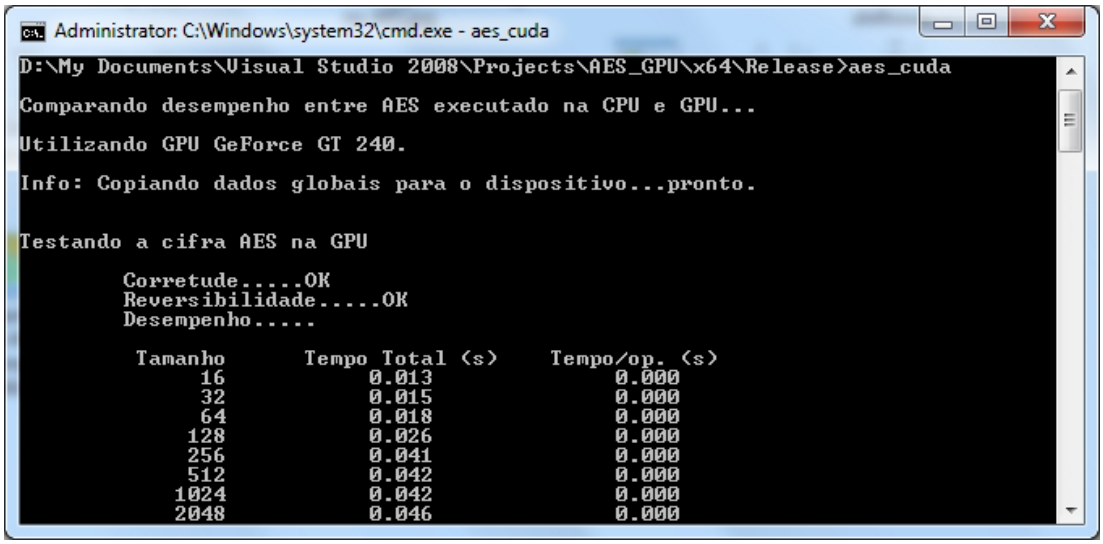

Figura 4.7 Programa em execução testando a GPU Geforce GT 240.

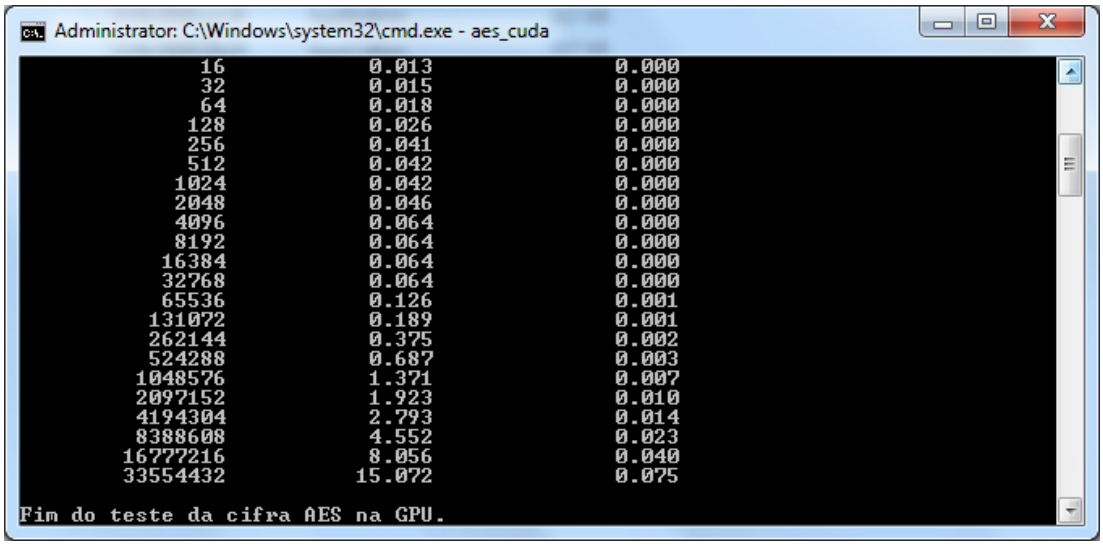

Figura 4.8 Terminando o teste da GPU Geforce GT 240.

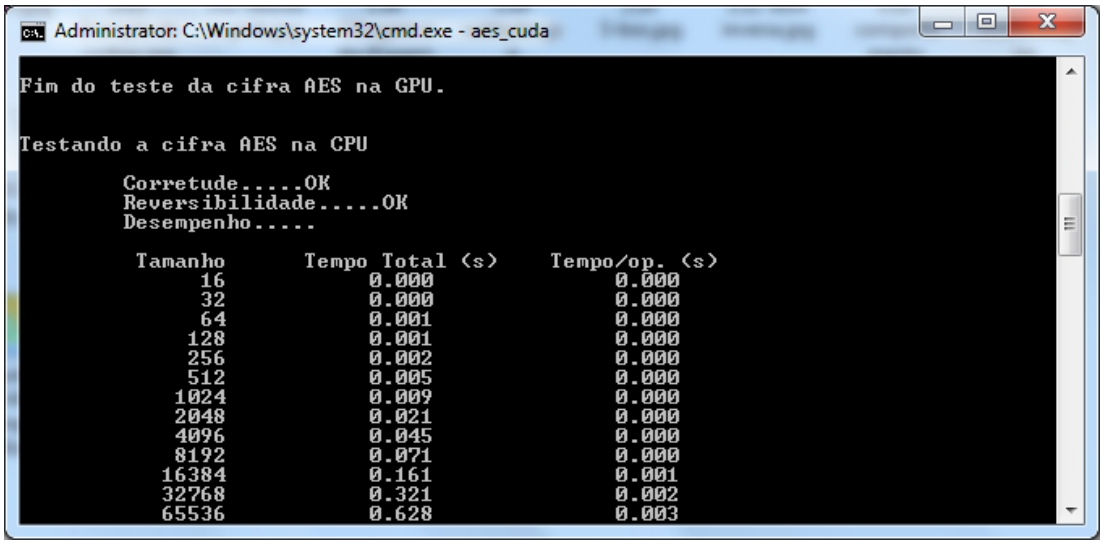

Figura 4.9 Iniciando o teste na CPU.

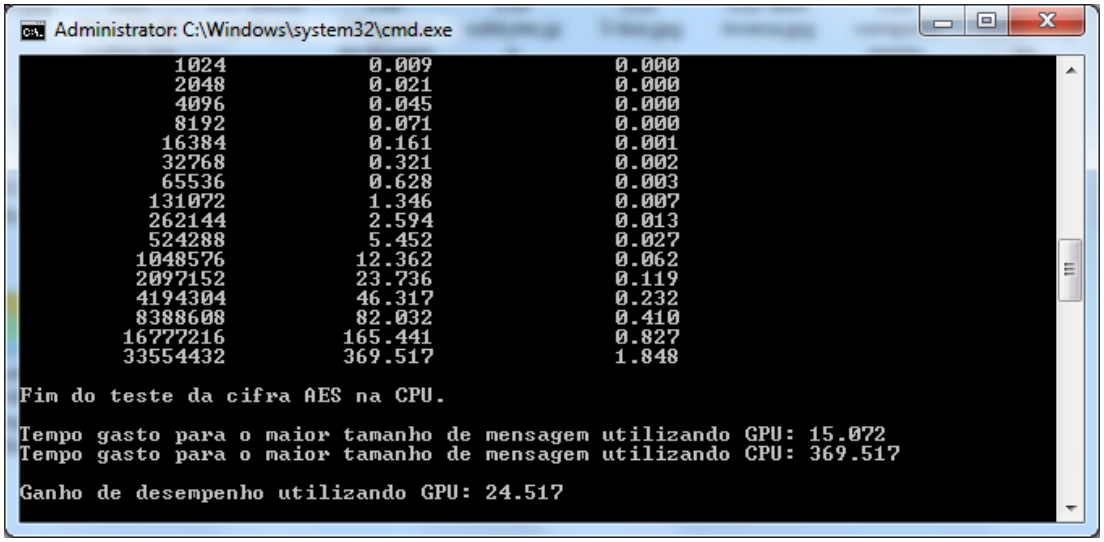

Figura 4.10 Fim da execução do programa.

Apenas lembrando que o tempo é medido em segundos. Para a captura dessas telas foi necessário utilizar a CPU para algumas aplicações. Portanto, o tempo de execução do algoritmo na CPU mostrado nas figuras é um pouco maior do que o analisado, devido ao fato da utilização da CPU para outras aplicações enquanto o programa era executado. Isso prova que o algoritmo é sensível à CPU, utilizando 100% do processamento todo o tempo, não deixando a CPU para outras possíveis aplicações, aumentando ainda mais a diferença dos tempos de execução entre a CPU e a GPU.

4.3.2 Testes do MD5

Os testes do algoritmo MD5 consistem no sistema de força bruta de recuperação da palavra original. Palavras de quatro a oito caracteres foram utilizadas e os tempos de execução para chegar nessas palavras a partir de seus respectivos hashes foram medidos em segundos. As palavras e seus respectivos hashes são mencionados a seguir:

- SPFC - 6f72c9342c9bfd7930c17d12914e2f77
- Unesp - 353bbd9e86c0d7c32bf8ec2144fbc883
- Ibilce - 5c7a0e154a18a483c1630931594202ac
- Palavra - 02d310ea805e1e08f4d76cf709a08f18
- ZyXwDCba - fc42bad0981f824b502706de21240aab

A figura 4.11 mostra o funcionamento do programa para a palavra SPFC.

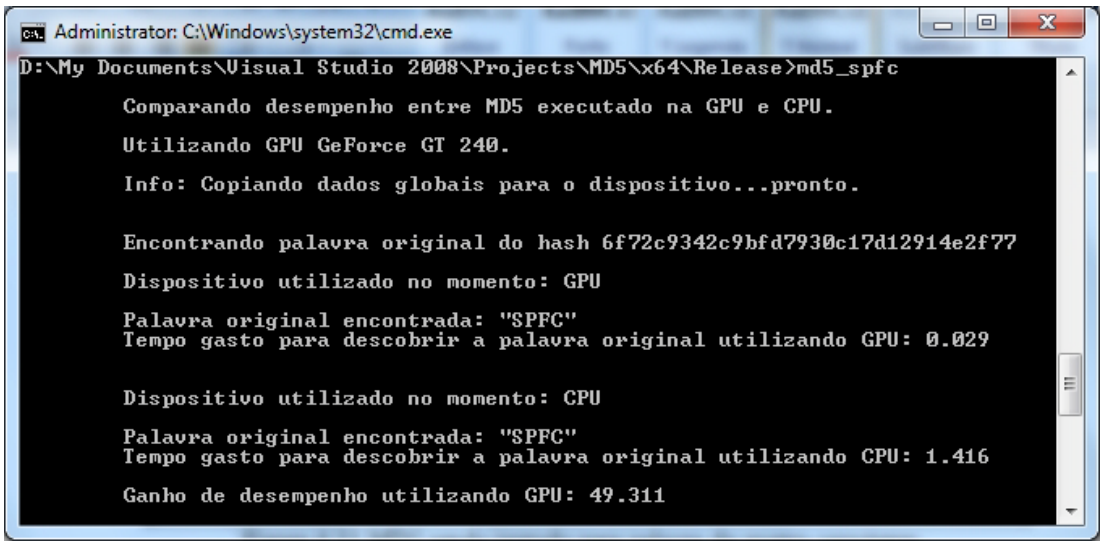

Figura 4.11 MD5 sendo testado para palavra de quatro caracteres.

A figura 4.12 mostra o funcionamento do programa para a palavra Unesp.

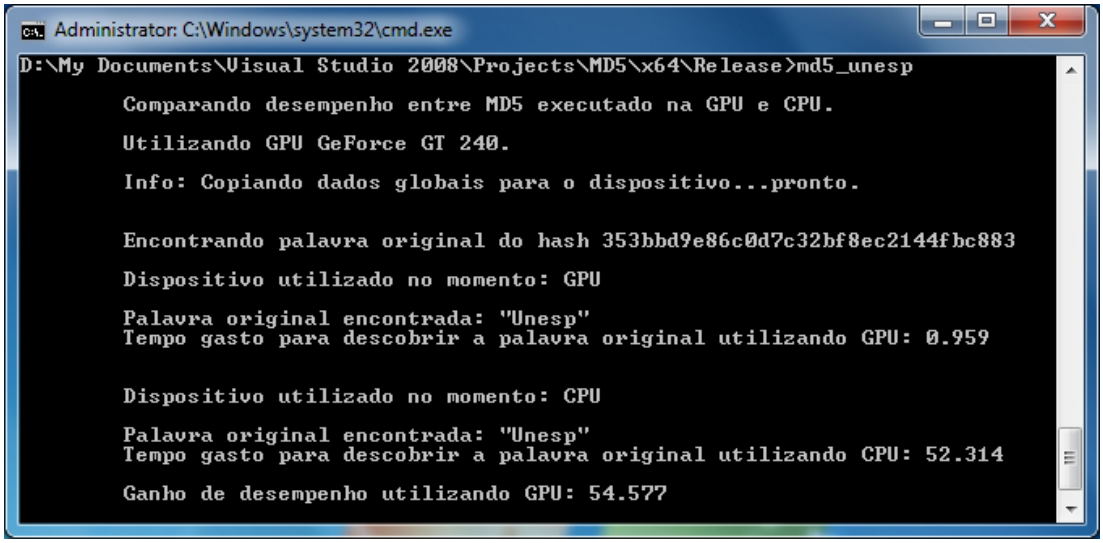

Figura 4.12 MD5 sendo testado para palavra de cinco caracteres.

A figura 4.13 mostra o funcionamento do programa para a palavra Ibilce.

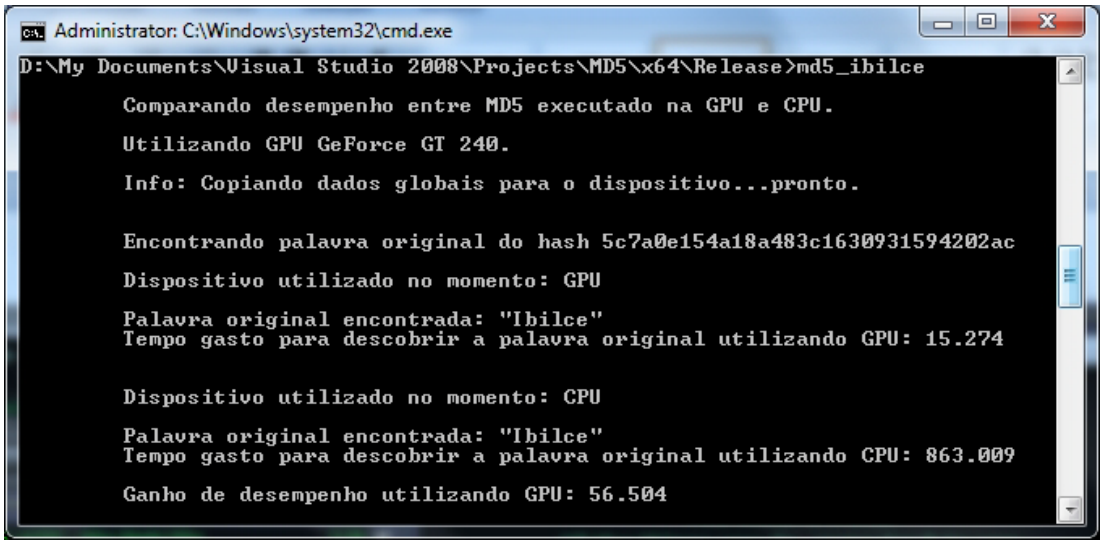

Figura 4.13 MD5 sendo testado para palavra de seis caracteres.

A figura 4.14 mostra o funcionamento do programa para a palavra Palavra.

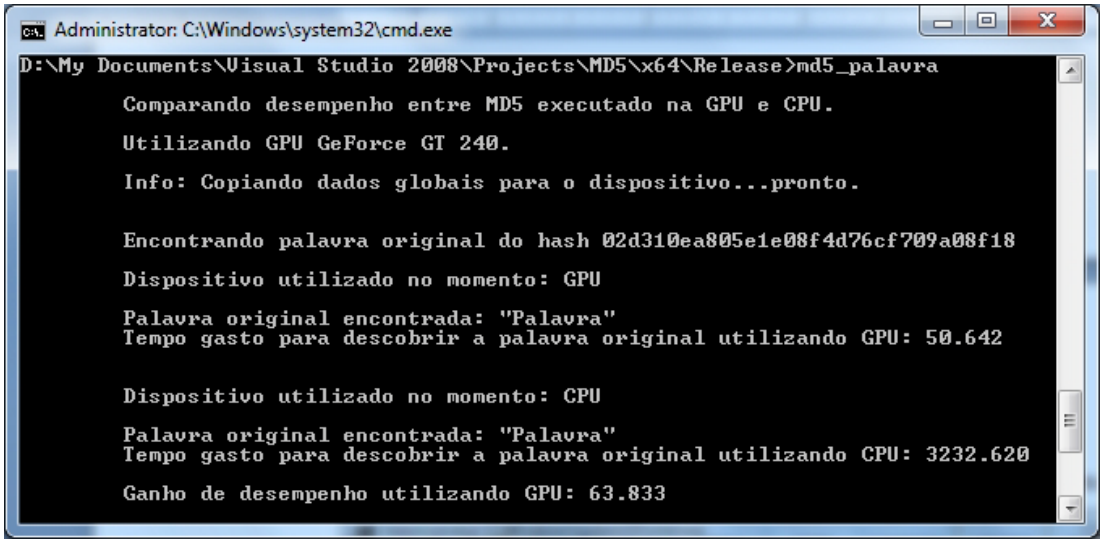

Figura 4.14 MD5 sendo testado para palavra de sete caracteres.

A figura 4.15 mostra o funcionamento do programa para a palavra ZyXwDCba.

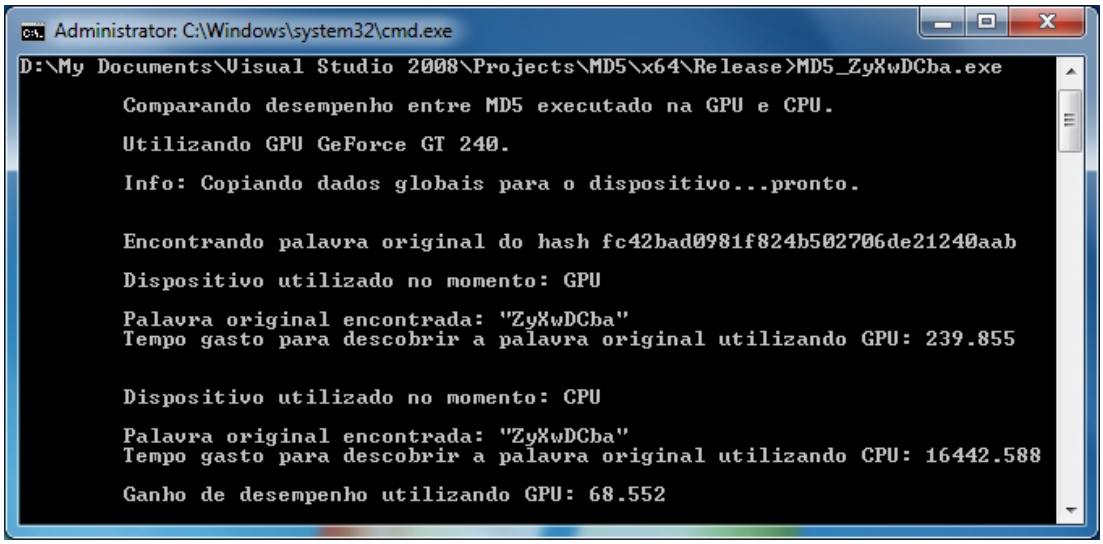

Figura 4.15 MD5 sendo testado para palavra de oito caracteres.

4.4 Comparação de desempenho

Esta seção compara os resultados obtidos em todos os testes realizados e os analisa.

4.4.1 Desempenho do algoritmo AES

Os resultados foram obtidos ao encriptar e desencriptar um conjunto aleatório de dados 100 vezes. A saída do processo de desencriptação foi comparada com a entrada do processo de encriptação para verificação de corretude. O tempo de verificação não foi incluído no tempo total de execução do algoritmo. O tamanho da mensagem começa com 16 bytes, o menor tamanho possível de um bloco, e dobra de tamanho até 32MB, permitindo ver o comportamento do algoritmo conforme o problema aumenta de tamanho.

Todos os testes realizados foram feitos com 128 threads por bloco. O número de blocos era variável, sendo o valor da divisão do tamanho da mensagem por 2048, para mensagens iguais ou maiores que 2048. Enquanto o programa era executado, não havia mais nenhum aplicativo em execução.

Os resultados obtidos na utilização da GPU Geforce 8600 GT estão presentes na tabela 4.1. Os tempos são medidos em segundos.

Tabela 4.1 Resultados AES Geforce 8600 GT.

Tamanho da Mensagem	Tempo GPU	Tempo CPU	Razão CPU/GPU
16	0,011	0	0
32	0,013	0	0
64	0,016	0	0
128	0,024	0	0
256	0,039	0	0
512	0,04	0	0
1024	0,041	0	0
2048	0,045	0,015	0,333333333
4096	0,047	0,047	1
8192	0,051	0,063	1,235294118
16384	0,1	0,125	1,25
32768	0,195	0,265	1,358974359
65536	0,385	0,531	1,379220779
131072	0,765	1,064	1,390849673
262144	1,523	2,14	1,405121471
524288	3,037	4,36	1,435627264
1048576	6,034	8,736	1,447795824
2097152	9,128	16,956	1,857581069
4194304	14,236	33,877	2,379671256
8388608	24,438	67,783	2,77367215
16777216	44,883	135,632	3,021901388
33554432	85,755	273,013	3,183639438

Nota-se que para mensagens pequenas a versão CPU é mais rápida que a GPU. No entanto, a partir de 4 Kb a versão em CUDA começa a ter melhor desempenho que a versão em CPU, chegando a ser 3,18 vezes mais rápida no maior tamanho de mensagem suportado pelo hardware usado. A figura 4.16 ilustra bem o comportamento das duas versões, para mensagens a partir de 256K.

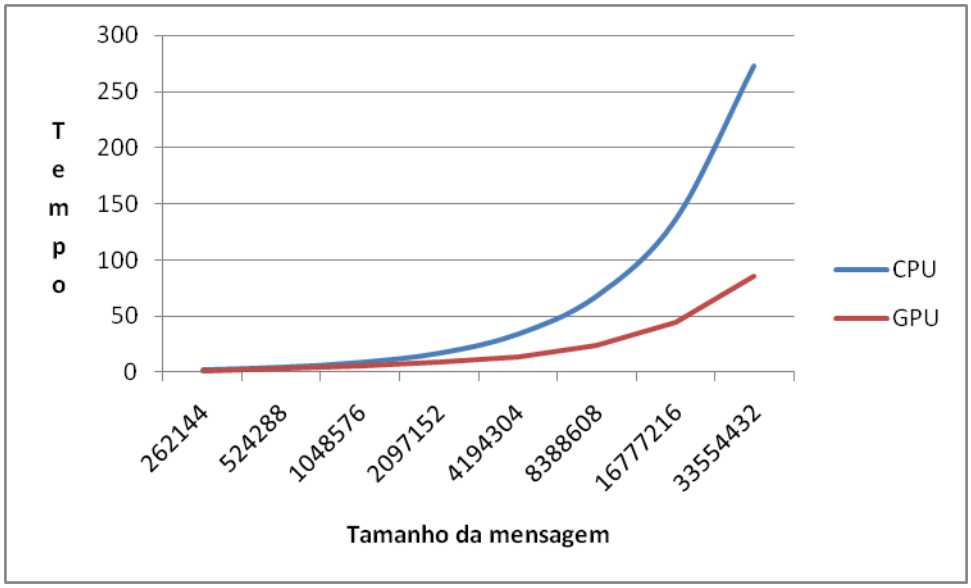

Figura 4.16 Resultados do teste de desempenho utilizando Geforce 8600 GT.

Nota-se que conforme o tamanho da mensagem aumenta, o tempo necessário para encriptar e desencriptar a mensagem aumenta muito mais para a CPU do que para a GPU. O maior uso efetivo da GPU é quando o tamanho da mensagem é suficientemente grande para aproveitar o paralelismo da arquitetura e amenizar o tempo gasto com transferências de dados entre as memórias. Da tabela 4.1 percebe-se que isto acontece quando o tamanho da mensagem é de pelo menos 8192, ou seja, com 4 blocos em execução e 128 threads por bloco, totalizando 512 threads ativas.

Para a GPU Geforce GT 240, os resultados obtidos estão presentes na tabela 4.2, com os tempos medidos em segundos.

Tabela 4.2 Resultados AES Geforce GT 240.

Tamanho da mensagem	Tempo GPU	Tempo CPU	Razão CPU/GPU
16	0,013	0	0
32	0,015	0	0
64	0,018	0	0
128	0,026	0	0
256	0,041	0	0
512	0,042	0	0

1024	0,042	0	0
2048	0,046	0,016	0,347826087
4096	0,046	0,031	0,673913043
8192	0,046	0,172	3,739130435
16384	0,05	0,218	4,36
32768	0,063	0,391	6,206349206
65536	0,126	0,547	4,341269841
131072	0,188	1,234	6,563829787
262144	0,375	2,172	5,792
524288	0,685	4,406	6,432116788
1048576	1,368	8,986	6,56871345
2097152	1,912	17,766	9,291841004
4194304	2,778	35,706	12,85313175
8388608	4,51	71,279	15,80465632
16777216	7,969	142,767	17,91529678
33554432	14,886	282,107	18,95116217

Assim como na outra GPU testada, para tamanho de mensagem superior a 4096 a GPU começa a ter desempenho superior à CPU. Devido ao maior número de *cores* da GT 240, o desempenho para o maior tamanho de mensagem é muito superior, chegando a ser aproximadamente 19 vezes melhor que a versão CPU. O comportamento do tempo é demonstrado na figura 4.17. Nota-se que conforme aumenta o tamanho da mensagem, a GPU pouco sofre para manter o desempenho enquanto que o tempo utilizado pela CPU aumenta vertiginosamente.

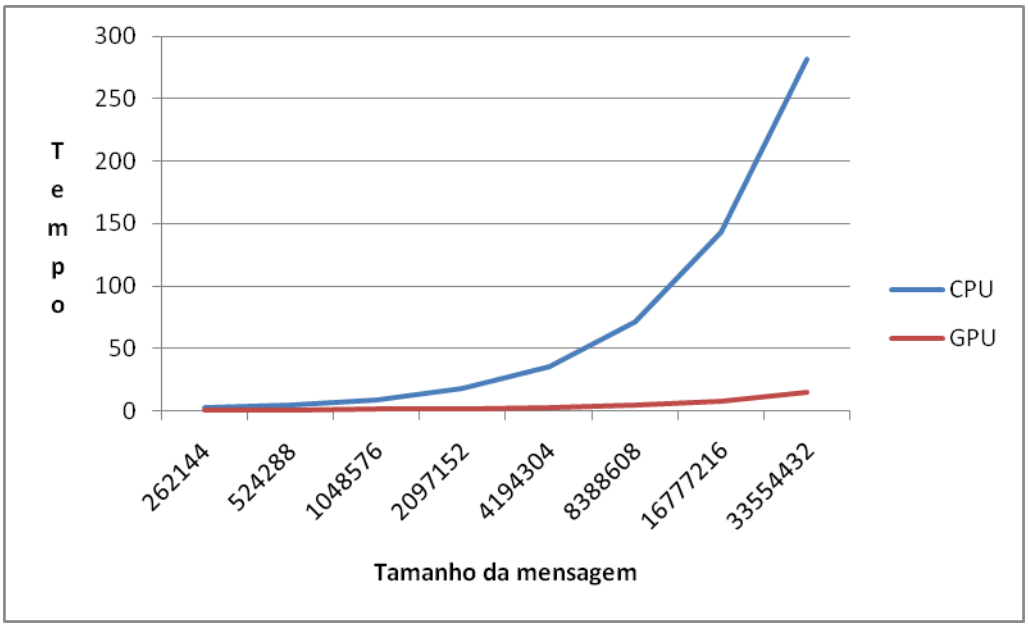

Figura 4.17 Resultados do teste de desempenho utilizando Geforce GT 240.

4.4.2 Desempenho do algoritmo MD5

Todos os testes do algoritmo MD5 na GPU foram feitos com 128 threads por bloco e 8192 blocos. Mediu-se o tempo, em segundos, para encontrar as palavras originais a partir dos hashes utilizando CPU e GPU. Para analisar o desempenho conforme o problema aumenta, as palavras variam de quatro a oito caracteres.

Como o programa precisa percorrer todas as palavras, transformando-as em hashes e comparando com o hash original, então mesmo em palavras pequenas como SPFC, a versão executada na GPU tem desempenho muito melhor que a CPU devido à capacidade de cada thread conseguir verificar uma palavra diferente, sem depender de outras threads, alcançando assim um alto grau de paralelismo.

A tabela 4.3 mostra um sumário do teste realizado.

Tabela 4.3 Resultados MD5.

Palavra utilizada	Tempo de execução GPU	Tempo de execução CPU	Razão CPU/GPU
SPFC	0,029	1,416	48,82758621
Unesp	0,959	52,314	54,55057351
Ibilce	15,274	863,009	56,50183318
Palavra	50,642	3232,62	63,83278701
ZyXwDCba	239,855	16442,588	68,55220029

Conforme é possível ver na tabela, o desempenho do algoritmo executado na GPU chega a ser quase 70 vezes superior ao algoritmo executado na CPU para a palavra com oito caracteres. A figura 4.18 mostra como a versão da CPU demanda muito mais tempo para chegar à palavra original conforme ela aumenta.

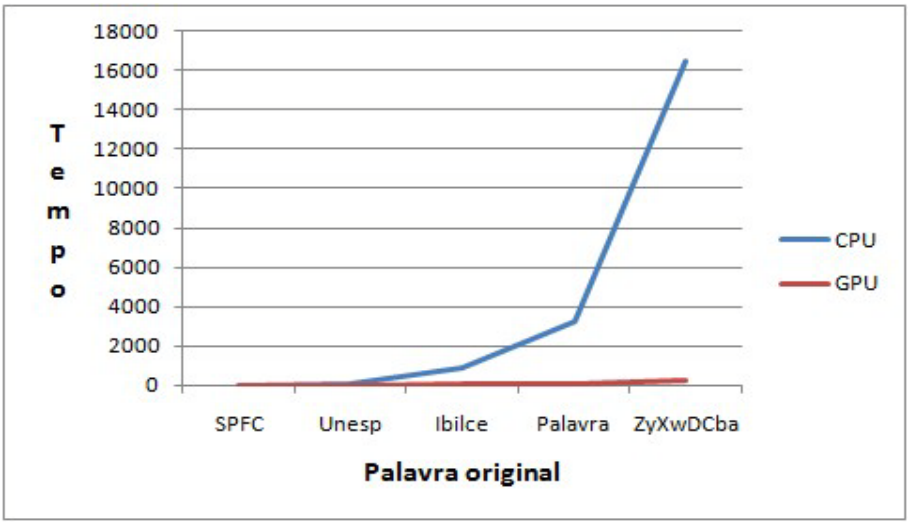

Figura 4.18 Resultados do teste de desempenho do algoritmo MD5.

4.5 Considerações finais

Neste capítulo foram apresentadas as soluções encontradas para implementar as funcionalidades propostas pelo projeto e os testes realizados para verificar se todas as funcionalidades funcionam como deveriam.

Os testes mostraram que os algoritmos funcionam corretamente tanto na CPU quanto na GPU, comprovando que o sistema foi implementado de maneira satisfatória. Além disso, os testes mostram que o desempenho na GPU de ambos os algoritmos é muito superior ao desempenho na CPU, mostrando que pode ser viável a implementação desses algoritmos em GPU para aumentar o desempenho.

O processo de recuperação de palavras a partir de um hash mostrou-se muito custoso a CPU, necessitando de algumas horas para obter sucesso em palavras de apenas oito caracteres, enquanto que apenas alguns minutos eram necessários para o sucesso utilizando GPU. Com o aumento do tamanho da palavra, a utilização de CPU para recuperação de senhas fica inviável, tornando assim a utilização de GPUs ainda mais importante.

Capítulo 5 – Conclusão

5.1 Introdução

A computação paralela é provavelmente o destino da evolução dos sistemas de computação de alto desempenho. Há alguns anos, a Intel apontava para um futuro com processadores com apenas um núcleo funcionando com *clocks* cada vez mais elevados, atingindo 10 GHz em 2011 (GEEK 2000). Naquela época a única medida de desempenho era os Hz, enquanto agora a proliferação de múltiplos núcleos parece ser o caminho a ser seguido.

As GPUs constituem o primeiro dispositivo de computação paralela de baixo custo. Atualmente, a utilização das GPUs para cálculos genéricos parece ser um ato de racionalidade. As dificuldades de programabilidade diminuem conforme plataformas como CUDA são lançadas no mercado.

5.2 Conclusões

Neste trabalho apresentei uma implementação do AES em GPU que chega a ser 19 vezes mais rápida que a implementação em CPU. Com isso, esse trabalho pode ser considerado uma prova de que a GPU pode ser utilizada como um coprocessador para criptografia, deixando a CPU para outros cálculos e aplicações.

É importante ressaltar que o hardware utilizado é considerado modesto e os resultados podem ser ainda melhores. Por isso, os resultados obtidos neste trabalho

não esgotam as possibilidades de exploração de CUDA. A utilização de máquinas com múltiplas GPUs representa um potencial enorme a ser explorado na incessante busca por desempenho.

Além disso, a GPU pode ser utilizada para propósito de recuperação de senhas. No caso, implementei o algoritmo MD5 tanto em CPU quanto em GPU e mostrei que é viável a obtenção da palavra original a partir de um hash, provando que o algoritmo não é tão seguro. Contudo, com o uso de senhas mais complexas, com mais caracteres, números ou até símbolos, a utilização de CPU para recuperação de senhas fica inviável, aumentando em muito o tempo de execução do algoritmo conforma a palavra ou o alfabeto utilizado aumenta. No entanto, uma solução plausível para esse problema é a utilização de GPUs que abrevia o tempo de recuperação para questão de minutos ou algumas horas, dependendo do tamanho e da complexidade da palavra.

5.3 Dificuldades encontradas

A primeira dificuldade encontrada é em relação à CUDA, que apresenta um novo conceito de paralelismo e é uma tecnologia relativamente nova, embora já exista uma ampla base bibliográfica para estudo. Embora CUDA se utilize basicamente da linguagem C e algumas extensões, há todo um novo paradigma de programação e meios de otimização do código que dificultaram o aprendizado. É importante relatar que antigamente era muito mais complicado trabalhar com programação em GPUs e CUDA trouxe um grande benefício aos programadores, tornando mais fácil a programação nas GPUs.

Outro aspecto importante foi todo o estudo dos algoritmos de criptografia AES e MD5, conhecimento esse não visto durante a faculdade.

A adaptação do ambiente Visual Studio para funcionar com CUDA também foi um pouco complicada, pois não há um passo a passo disponível sobre como integrar a arquitetura CUDA no ambiente Visual Studio.

5.4 Trabalhos Futuros

É possível utilizar a arquitetura em outros algoritmos criptográficos para aumentarem seus desempenhos. Outros algoritmos simétricos podem ser implementados também, junto com algoritmos de *hash* e algoritmos assimétricos, a fim de criar um *framework* de algoritmos criptográficos acelerados pela GPU pela utilização de CUDA. Além disso, é viável a adaptação do código para ser executado em outro sistema operacional como o Linux para verificação de desempenho.

A arquitetura CUDA também pode ser utilizada para testar a segurança de outros algoritmos, testando vulnerabilidades que antes eram inviáveis com a utilização de CPUs convencionais.

Referências Bibliográficas

KNUDSEN, J; “Java Cryptography”, Beijing: O´Reilly, 1998.

KIRK, D.; HWU, W.; “Programming Massively Parallel Processors”. Disponível em: http://courses.ece.illinois.edu/ece498/al/textbook/Chapter1-Introduction.pdf, 2008.

NVIDIA; “NVIDIA CUDA Programming Guide”. Disponível em: http://developer.download.nvidia.com/compute/cuda/2_3/toolkit/docs/NVIDIA_CUDA_ProgrammingGuide_2.3.pdf, 2009.

OWENS, J.; LUEBKE, D.; GOVINDAJARU, N.; HARRIS, M.; KRUGER, J.; LEFOHN, A.; PURCELL, T.; “A Survey of General-Purpose Computing on Graphics Hardware”, Computer Graphics Forum, 2007.

AKOGLU, A.; STRIEMER, G.; “Scalable and Highly Parallel Implementation of Smith-Waterman on graphics processing unit using CUDA”, Springer, 2009.

TOLKE, J.; “Implementation of a Lattice Boltzmann kernel using the Compute Unified Device Architecture developed by nVIDIA”, Springer, 2008.

STRIPPGEN, D.; NAGEL, K.; “Using common graphics hardware for multi-agent traffic simulation with CUDA”, ACM, 2009.

GARLAND, M.; GRAND, S.; NICKOLLS, J.; ANDERSON, J.; HARDWICK, J.; MORTON, S.; PHILLIPS, E.; ZHAN, Y.; VOLKOV, V.; “Parallel Computing Experiences with Cuda”, IEEE, 2008.

MANAVSKI, S. A.; "Cuda Compatible GPU as an Efficient Hardware Accelerator for AES Cryptography", IEEE ICSPC, 2007.

SILBERSTEIN, M.; SCHUSTER, A.; GEIGER, D.; PATNEY, A.; OWENS, J.; “Efficient Computation of Sum-Products on GPU through Software-Managed Cache”, ACM, 2008.

WECHSLER, O.; “Inside Intel Core Microarchitecture Setting New Standards for Energy-Efficient Performance”, Tecnology@Intel, 2006.

KANT, S.; MITHUN, U.; GUPTA, P. S. S. B. K.; "Real time H.264 video encoder implementation on a programmable DSP processor for videophone applications", Consumer Electronics ICCE, 2006.

TRICHINA, E.; KORKISHKO, T.; LEE, K. H.; "Small Size, Low Power, Side Channel-Immune AES Coprocessor: Design and Synthesis Results", Lecture Notes on Computer Science, 2005.

GOVINDAJARU, N.; GRAY, J.; KUMAR, R.; MANOCHA, D.; "GPUTeraSort: high performance graphics co-processor sorting for large database management", ACM SIGMOD, 2006.

INTEL; "AGP V3.0 Interface Specification", Intel Corporation, 2002.

AMD; "Torrenza Technology", Advanced Micro Devices, 2006.

VENKATASUBRAMANIAN, S.; "The Graphics Card as a Stream Computer", AT&T Labs Research, 2003.

HALFHILL, T. R.; "Parallel processing with cuda", Microprocessor Report, 2008 .

KAHN, D.; "The Codebreakers: The Comprehensive History of Secret Communication from Ancient Times to the Internet", Scribner, 1996.

NIST; "Specification for the Advanced Encryption Standard (AES)", FIPS 197, 2001.

KEDEM, G.; ISHIHARA, Y.; "Brute Force Attack on Unix Passwords with SIMD Computer", Proceedings of the 8th USENIX Security Symposium, 1999.

HARRISON, O.; WALDRON, J.; "AES Encryption Implementation and Analysis on Commodity Graphics Processing Units", Lecture Notes on Computer Science, 2007.

FIORESE, C.; BUDAK, C.; "AES on GPU: a CUDA Implementation", UCSB, 2008.

NIST; "AES Known Answer Test (KAT) Vectors", disponível em http://csrc.nist.gov/groups/STM/cavp/documents/aes/KAT_AES.zip. Acessado em 30/11/2009.

GEEK, "Intel predicts 10GHz chips by 2011", disponível em http://www.geek.com/intel-predicts-10ghz-chips-by-2011. Acessado em 30/11/2009.