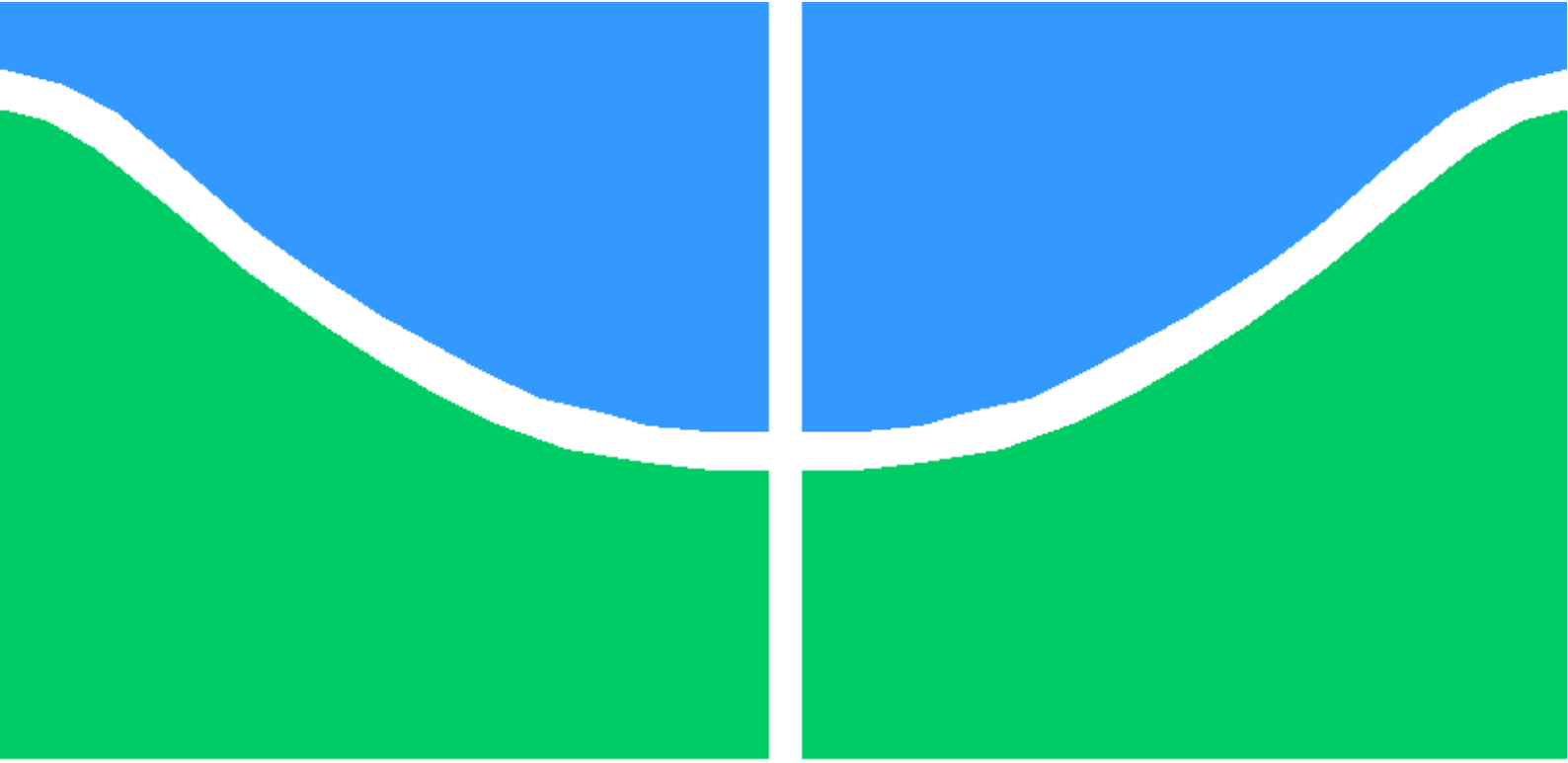


IMPLEMENTAÇÃO DO AES NA PLATAFORMA CUDA

MARCEL AUGUSTUS BARBOSA CARVALHO

DISSERTAÇÃO DE MESTRADO EM ENGENHARIA ELÉTRICA
DEPARTAMENTO DE ENGENHARIA ELÉTRICA

FACULDADE DE TECNOLOGIA
UNIVERSIDADE DE BRASÍLIA

UNIVERSIDADE DE BRASÍLIA
FACULDADE DE TECNOLOGIA
DEPARTAMENTO DE ENGENHARIA ELÉTRICA

IMPLEMENTAÇÃO DO AES NA PLATAFORMA CUDA

MARCEL AUGUSTUS BARBOSA CARVALHO

ORIENTADOR: ANDERSON CLAYTON ALVES NASCIMENTO

DISSERTAÇÃO DE MESTRADO EM ENGENHARIA ELÉTRICA

PUBLICAÇÃO: PPGENE.DM - 490 A/12

BRASÍLIA/DF: SETEMBRO - 2012.

UNIVERSIDADE DE BRASÍLIA
FACULDADE DE TECNOLOGIA
DEPARTAMENTO DE ENGENHARIA ELÉTRICA

IMPLEMENTAÇÃO DO AES NA PLATAFORMA CUDA

MARCEL AUGUSTUS BARBOSA CARVALHO

DISSERTAÇÃO DE MESTRADO SUBMETIDA AO DEPARTAMENTO DE ENGENHARIA ELÉTRICA DA FACULDADE DE TECNOLOGIA DA UNIVERSIDADE DE BRASÍLIA, COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE MESTRE EM ENGENHARIA ELÉTRICA.

APROVADA POR:

Prof. Dr. Anderson Clayton Alves Nascimento (ENE-UnB)
(Orientador)

Prof. Dr. Rafael Timóteo de Sousa Jr. (ENE-UnB)
(Examinador Interno)

Prof. Dr. Diego de Freitas Aranha (CIC-UnB)
(Examinador Externo)

BRASÍLIA/DF, 17 DE SETEMBRO DE 2012.

FICHA CATALOGRÁFICA

CARVALHO, MARCEL AUGUSTUS BARBOSA
Implementação do AES na Plataforma CUDA.
[Distrito Federal] 2012.
xvii, 144p., 297 mm (ENE/FT/UnB, Mestre, Engenharia Elétrica, 2012).
Dissertação de Mestrado - Universidade de Brasília.
Faculdade de Tecnologia.
Departamento de Engenharia Elétrica.
1. Criptografia 2. AES
3. GPGPU 4. CUDA
I. ENE/FT/UnB II. Título (série)

REFERÊNCIA BIBLIOGRÁFICA

CARVALHO, M. A. B. (2012). Implementação do AES na plataforma CUDA. Dissertação de Mestrado em Engenharia Elétrica, Publicação PPGENE.DM - 490 A/12, Departamento de Engenharia Elétrica, Universidade de Brasília, Brasília, DF, 144p.

CESSÃO DE DIREITOS

NOME DO AUTOR: Marcel Augustus Barbosa Carvalho.

TÍTULO DA DISSERTAÇÃO DE MESTRADO: Implementação do AES na plataforma CUDA

GRAU / ANO: Mestre / 2012

Marcel Augustus Barbosa Carvalho
Universidade de Brasília - Faculdade de Tecnologia
Departamento de Engenharia Elétrica
70910-900 Brasília-DF Brasil.

AGRADECIMENTOS

Agradeço à Professora Doutora Alba Cristina Magalhães Alves de Melo por gentilmente ceder as instalações do Laboratório de Sistemas Integrados e Concorrentes (LAICO) do Departamento de Ciência da Computação da Universidade de Brasília, para a mensuração dos resultados descritos nesta dissertação em hardwares de ponta.

Agradeço também ao Chefe do Centro de Desenvolvimento de Sistemas do Exército Brasileiro, General de Brigada Bráulio de Paula Machado, pelo total apoio e forte incentivo para a finalização deste trabalho.

MARCEL AUGUSTUS BARBOSA CARVALHO

Dedicatória

Dedico este trabalho a minha amada esposa Jaqueline Lima e minhas lindas filhas Camille Pietra e Nicole Jolie que tanto me apoiaram nesse árduo caminho mesmo em detrimento da minha constante ausência do convívio familiar.

MARCEL AUGUSTUS BARBOSA CARVALHO

RESUMO

IMPLEMENTAÇÃO DO AES NA PLATAFORMA CUDA

Autor: Marcel Augustus Barbosa Carvalho

Orientador: Anderson Clayton Alves Nascimento

Programa de Pós-graduação em Engenharia Elétrica

Brasília, setembro de 2012

Compute Unified Device Architecture (CUDA) é uma plataforma de computação paralela de propósito geral que tira proveito das unidades de processamento gráfico (GPU) NVIDIA para resolver problemas computacionais que possam ser paralelizáveis.

No campo da criptografia já foram realizados esforços no uso de GPUs com algoritmos criptográficos simétricos e assimétricos e mais recentemente com as funções de *hash*. Este trabalho realiza uma revisão das implementações anteriores do AES sobre GPUs e implementa o algoritmo AES para cifração e decifração com chaves de 128, 192 e 256 *bits* no modo ECB com *padding*, com variações no uso dos recursos disponíveis nas GPUs CUDA.

Como resultado final chegou-se a implementação em CUDA cuja configuração de recursos levou a ganhos no tempo total de cifração/decifração de até 32,7 vezes comparados à versão em CPU usada como referência.

ABSTRACT

IMPLEMENTATION OF AES ON THE CUDA PLATAFORM

Author: Marcel Augustus Barbosa Carvalho

Supervisor: Anderson Clayton Alves Nascimento

Programa de Pós-graduação em Engenharia Elétrica

Brasília, september of 2012

Compute Unified Device Architecture (CUDA) is a platform for general purpose parallel computing that takes advantage of NVIDIA Graphic Processing Units (GPU) to solve parallelizable computational problems.

In the field of the cryptography efforts have been made in the use of GPUs with asymmetric and symmetric cryptographic algorithms more recently with hash functions. This paper conducts a review of previous implementations of AES on GPU and implements the AES algorithm for encryption and decryption with keys of 128, 192 and 256 bits in ECB mode with padding, with variations in the use of available resources in CUDA GPUs.

As a final result, a CUDA implementation was obtained with a resource configuration providing gains in total time of encryption / decryption of up to 32,7 times compared to the used CPU version.

SUMÁRIO

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE CÓDIGOS FONTES

LISTA DE SÍMBOLOS, NOMENCLATURAS E ABREVIAÇÕES

ABNT	Associação Brasileira de Normas Técnicas
AES	*Advanced Encryption Standard*
AES-128	*Advanced Encryption Standard* com chave de 128 *bits*
AES-192	*Advanced Encryption Standard* com chave de 192 *bits*
AES-256	*Advanced Encryption Standard* com chave de 256 *bits*
AES-NI	*Intel Advanced Encryption Standard New Instructions*
ALU	*Arithmetic and Logic Unit*
API	*Application Program Interface*
CBC	*Cipher Block Chaining*
CDT	*C/C++ Development Tooling*
CFB	*Cipher Feedback*
CTR	*Counter*
CUDA	*Compute Unified Device Architecture*
CPU	*Central Processing Unit*
DDR3	*Double Data Rate Type Three*
DES	*Data Encryption Standard*
DRAM	*Dynamic Random-Access Memory*
ECB	*Eletronic Code Book*
FIPS	*Federal Information Processing Standards*
Gbps	*Gigabit* por segundo (10^9 *bits* por segundo)
Gibps	*Gibibit* por segundo (2^{30} *bits* por segundo)
GB	*Gigabyte* (10^9 *bytes*)
GiB	*Gibibyte* (2^{30} *bytes*)
GHz	*GigaHertz* (10^9 repetições por segundo)
GCC	*GNU Compiler Collection*
GF	Corpo de Galois
GPU	*Graphic Processinc Unit*
GPGPU	*General-purpose Computations on GPU*
IDE	*Integrated Development Environment*
ISA	*Instruction Set Architecture*
JIT	Compilador *Just-In-Time*
KB	*KiloByte* (1000 *bytes*)
KiB	*KibiByte* (1024 *bytes*)
LTS	*Long Term Support*

ms	milissegundo
MB	*MegaByte* (1000*1000 *bytes*)
MiB	*MebiByte* (1024*1024 *bytes*)
Mbps	*Megabit* por segundo(10^6 *bits* por segundo)
Mibps	*Mebibit* por segundo(2^{20} *bits* por segundo)
N_b	tamanho do bloco de entrada do AES em palavras de 32 *bits*
N_k	tamanho da chave do AES em palavras de 32 *bits*
N_r	número de rodadas (*rounds*) do AES
NIST	*National Institute of Standards and Technology*
OFB	*Output Feedback*
OpenGL	*Open Graphics Library*
OpenSSL	*Open Secure Sockets Layer*
PCI	*Peripheral Component Interconnect Express*
PTX	*Parallel Thread eXecution*
PUBS	*Publications*
S-Box	Caixa de substituição do AES
SHA-3	*Secure Hash Algorithm* versão 3
SIMD	*Single Instruction Stream - Multiple Data Stream*
SIMT	*Single Instruction Stream - Multiple Threads*
SM	*Streaming Multiprocessor*
SSL	*Secure Sockets Layer*
TLS	*Transport Layer Security*
TI	Tecnologia da Informação
U.S	*United States*
UnB	Universidade de Brasília
XOR	Operação booleana Ou-Exclusivo

Capítulo 1

Introdução

Este capítulo apresenta a motivação deste trabalho, elenca os objetivos a serem alcançados e por fim, mostra a estruturação do manuscrito.

1.1 Contextualização

Graphics Processing Units (GPUs) são coprocessadores que tradicionalmente realizam a renderização de informações bidimensionais e tridimensionais para exibição para o usuário. O crescimento da indústria de jogos e o aumento da demanda por gráficos mais realistas e com renderização em tempo real forçou as GPUs a proverem unidades de processamento paralelas mais rápidas. Este esforço levou ao surgimento de dispositivos que sobrepujaram as CPUs em aplicações específicas, como no caso dos jogos, com a relação custo-benefício aceitável devido a produção em escala destes dispositivos.

Não demorou até a comunidade científica e os programadores perceberem que este poder computacional poderia ser utilizado para outras funções além das originais de computação gráfica. Em Harris (2003) foi introduzido o termo *General-purpose Computations on GPUs* (GPGPU) para se referir a estas aplicações não gráficas que usavam o poder computacional das GPUs . Naquele momento, programar GPGPU significava expressar seus algoritmos em termos de operações sobre dados gráficos, *pixels* e vetores. Este paradigma mudou quando dois importantes fabricantes de GPUs, NVIDIA e AMD, alteraram a arquitetura de *hardware* dos seus dispositivos gráficos para uma plataforma de computação *multi-core*, com a introdução das *unified processing units* nas placas gráficas. As GPU puderam então suportar um conjunto de instruções de propósito genérico, promovendo o surgimento de linguagens de programação, ferramentas e *frameworks* para usufruir desse potencial computacional.

Os benefícios do uso de GPUs são vários, mas pode-se destacar sua fácil integração em plataformas distintas como *desktops*, *notebooks*, servidores e até mesmo dispositivos móveis, sem quaisquer alterações fundamentais no *hardware*. GPUs estão disponíveis no mercado em uma vasta gama de níveis de desempenho, pode-se escolher uma de acordo com suas necessidades. Quando não estão sendo utilizadas para jogos ou na renderização de interfaces gráficas elegantes ficam ociosas, fazendo sentido usá-las como coprocessadores de operações custosas.

No campo da criptografia já foram realizados esforços no uso de GPUs com algoritmos criptográficos simétricos e assimétricos e mais recentemente com as funções de *hash*. Este trabalho realiza uma revisão das implementações anteriores do AES sobre GPUs e implementa o algoritmo AES para cifração e decifração com chaves de 128, 192 e 256 *bits* no modo ECB com *padding*, com variações no uso dos recursos disponíveis nas GPUs CUDA.

1.2 Definição do problema

Usar o poder computacional provido pelas GPUs CUDA no uso do algoritmo simétrico AES, atentando para as diferenças arquiteturais entre GPU CUDA e CPU e buscando identificar a combinação de recursos e funcionalidades disponíveis nas placas CUDA para obter o melhor desempenho do AES.

1.3 Objetivos do projeto

Realizar um estudo da arquitetura CUDA da NVIDIA, focando nos recursos e funcionalidades que possam ser explorados na implementação do algoritmo AES. Revisitar os trabalhos anteriores sobre o uso do algoritmo simétrico AES sobre dispositivos CUDA. Realizar a implementação do AES sobre esta arquitetura, elencando possibilidades de ganho de desempenho em comparação com uma implementação em CPU.

1.4 Apresentação do manuscrito

No Capítulo 2, é feita uma descrição da arquitetura CUDA, focando nos seus recursos e funcionalidades. No Capítulo 3, é apresentada uma revisão sobre o AES e alguns trabalhos anteriores que fizeram uso do AES em GPUs. Em seguida, o Capítulo 4 descreve a metodologia empregada no desenvolvimento do projeto e a implementação do AES. Resultados experimentais são discutidos no Capítulo 5, e por fim as conclusões no Capítulo 6.

Capítulo 2

CUDA - *Compute Unified Device Architecture*

Este capítulo tem a função de apresentar a arquitetura CUDA, modelo de memória, modelo de execução e seus paradigmas de implementação

2.1 Introdução

Compute Unified Device Architecture (CUDA) é uma plataforma de computação paralela de propósito geral que tira proveito das unidades de processamento gráfico (GPU) NVIDIA para resolver problemas computacionais que possam ser paralelizáveis. Possui um conjunto de instruções chamado CUDA ISA (*Instruction Set Architecture*) e o mecanismo de computação paralela na GPU. Para programar segundo a arquitetura CUDA, os desenvolvedores utilizam C com um conjunto de extensões, o chamado CUDA *Toolkit*, que pode ser então executado em um processador compatível com CUDA. Outras linguagens também são admitidas, como C++ e FORTRAN. Em versões antigas do CUDA *Toolkit*, era possível desenvolver mesmo sem ter o *hardware* necessário, usando um emulador, mas a partir da versão 3.0, essa funcionalidade não é mais suportada.

2.2 Histórico

As primeiras GPUs foram projetadas como aceleradores gráficos, suportando somente *pipelines* com funções fixas específicas[1]. Fazer uso das GPU para outros propósitos além das gráficas, demandava o uso de linguagens de programação gráficas como a OpenGL[2]. Os desenvolvedores tinham que mapear cálculos científicos a problemas que poderiam ser representados por triângulos e polígonos.

Em Buck et al. (2004) foi apresentado um sistema de computação de propósito geral em

[1] Etapas pelas quais um dado gráfico é processado por uma GPU, tais como, operações com vértices, montagem de primitivas, rasterização, operação com fragmentos e composição.

[2] OpenGL (*Open Graphics Library*) é uma API livre utilizada na computação gráfica, para desenvolvimento de aplicativos gráficos, ambientes 3D, jogos, entre outros.(WIKIPÉDIA, 2012b)

hardwares gráficos programáveis, chamado Brook. Este sistema estendia a linguagem C para incluir construções de paralelismo de dados, permitindo o uso das GPUs como coprocessadores de fluxo[3]. Neste trabalho foi proposto um compilador e um *runtime* que abstraía e virtualizava muitos aspectos dos dispositivos gráficos.

A NVIDIA sabia que um *hardware* extremamente rápido tinha que ser combinado a ferramentas intuitivas de *software* e *hardware*, e por isso convidou Ian Buck para juntar-se à empresa e começar a desenvolver uma solução para executar o C na GPU de forma fluida. Juntando o *software* e o *hardware*, a NVIDIA apresentou o CUDA em 2006, a primeira solução do mundo para computação de propósito geral em GPUs (NVIDIA, 2012c).

Atualmente todas as GPUs da NVIDIA suportam CUDA. CUDA não é uma arquitetura computacional no sentido de um conjunto de instruções e um conjunto de registradores específicos. Binários compilados para uma GPU CUDA não necessariamente executarão em todas as GPUs. A NVIDIA definiu o termo *"compute capability"* para descrever as funcionalidades suportadas por um *hardware* CUDA. A primeira GPU CUDA, GeForce 8800, possui a *compute capability* 1.0, e em 2011 a NVIDIA lançou GPUs com a *compute capability* 2.x, também conhecidas como arquitetura "Fermi", veja Figura 2.1. Mais detalhes sobre as funcionalidades correspondentes a cada *compute capability* encontram-se em NVIDIA (2012a, Apêndice F, pág. 135).

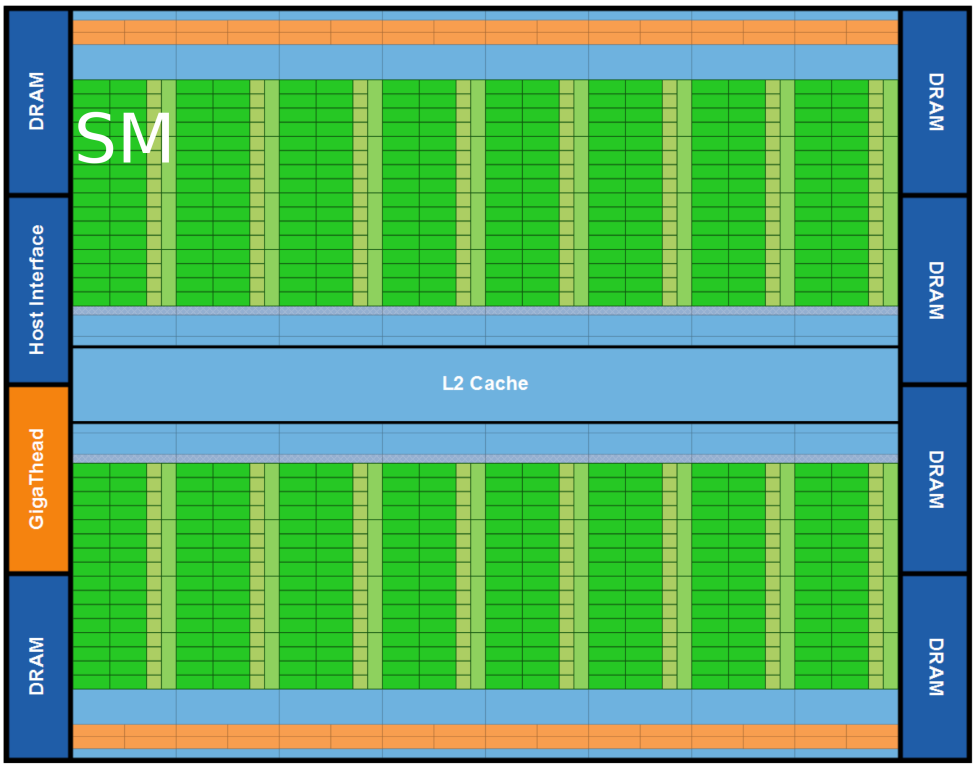


Figura 2.1: Fermi - Visão Geral da Arquitetura. (NVIDIA, 2011c)

Uma GPU CUDA consiste de múltiplos *streaming multiprocessor* (SMs), no caso da arquitetura

[3]Unidade de processamento que sobre um conjunto de dados (fluxo) aplica um série de operações (funções de *kernel*). Esta relacionado ao paradigma de programação *"single instruction, multiple data"* (SIMD).

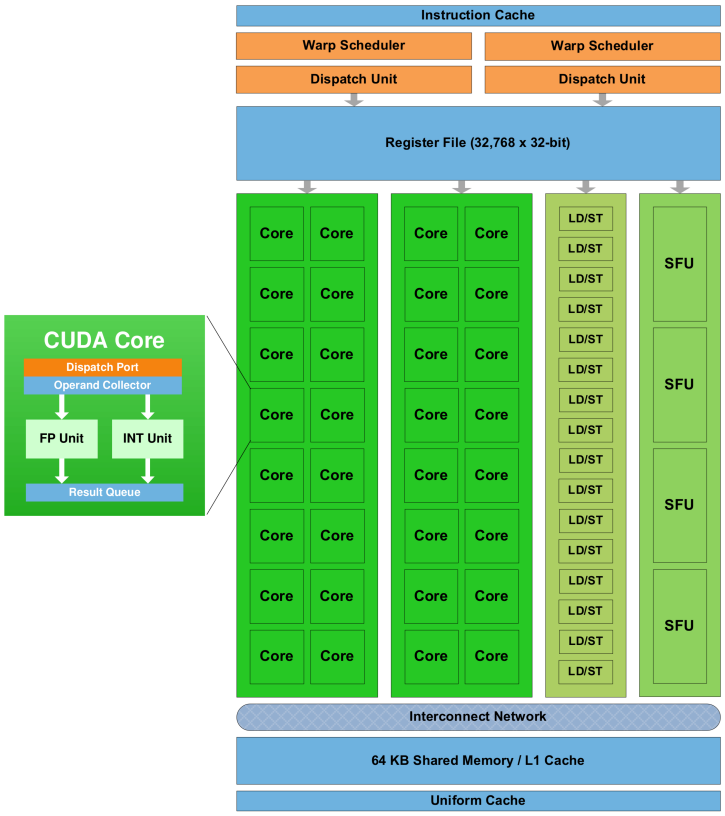


Figura 2.2: Fermi - Streaming Multiprocessor (SM)(NVIDIA, 2011c)

Fermi são 16 SMs, conforme exemplificado na Figura 2.1. Cada SM contém vários núcleos (*core*) CUDA, 8 *cores* por SM nas placas com *compute capability* 1.x, 32 *cores* por SM nas placas com *compute capability* 2.0 e 48 *cores* por SM nas placas com *compute capability* 2.1, cada *core* é o responsável por executar os cálculos aritméticos. Na Figura 2.2 está exemplificado um SM na arquitetura Fermi, contendo 32 CUDA *cores*.

2.3 CPU x GPU

De forma simplista as GPUs são projetadas com mais transistores dedicados ao processamento de dados, as chamadas unidades lógicas aritméticas (ALU), do que para *caching* e controle de fluxo, visando executar vários fluxos de instruções o mais rápido possível, enquanto CPUs gastam seus transistores com a finalidade de aumentar a velocidade de execução de um fluxo de instruções.

CPUs suportam uma ou duas *threads* por *core*, enquanto GPUs CUDA suportam até 1024 *threads* por *streaming multiprocessor*. O custo de troca (*switch*) de uma *thread* na CPU é de algumas centenas de ciclos, enquanto as GPUs não tem custo nas trocas de *threads*, na realidade as *threads* são trocadas a cada ciclo de relógio.

As GPUs usam o conceito de *"Single Instruction Stream - Multiple Data Stream"* (SIMD) para explorar o poder computacional destes ALUs, executando um único fluxo de instruções em múltiplos fluxos de dados independentes (FLYNN, 1966). Em GPUs CUDA ocorre um variação

na abordagem SIMD, um conjunto de *threads* executa o mesmo fluxo de instruções em diferentes conjuntos de dados, NVIDIA chama essa abordagem de *"Single Instruction Stream - Multiple Threads"* (SIMT) (NVIDIA, 2012a, pág.61).

Essa comparação arquitetural simplificada pode ser exemplificada pela Figura 2.3.

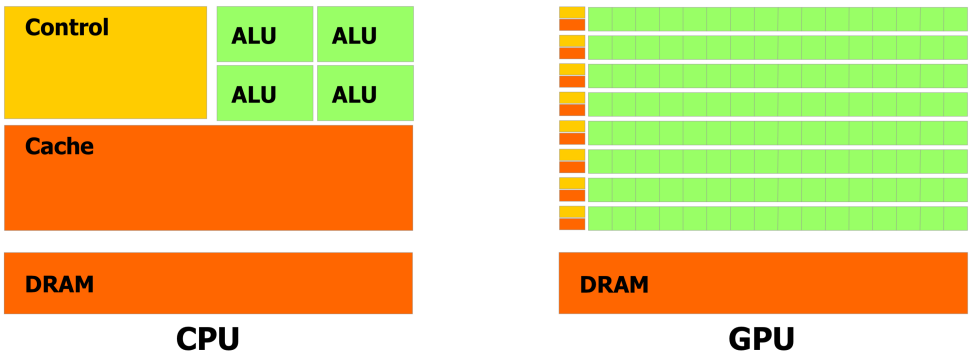


Figura 2.3: CPU x GPU (NVIDIA, 2012a)

Mais especificamente, a GPU é especialmente adequada para resolver os problemas que podem ser expressos como o mesmo programa sendo executado em muitos elementos de dados em paralelo, programas com uma alta intensidade de cálculos aritméticos comparados com as operações de acesso à memória. Como o mesmo programa é executado para cada elemento de dados, há uma menor necessidade de controle de fluxo sofisticado, e como é executado em muitos elementos de dados e tem intensidade aritmética elevada, a latência de acesso de memória pode ser ocultada com o uso cálculos intensos em vez de grandes *caches* de dados como ocorre nas CPUs.

2.4 Modelo de programação

O modelo de programação paralela CUDA baseia-se em três abstrações: uma hierarquia de grupos de *threads*, hierarquia de memórias compartilhadas, e barreiras de sincronização. Essas abstrações são disponibilizadas ao programador como um conjunto mínimo de extensões da linguagem de programação.

Essas abstrações guiam o programador a dividir o problema em grandes sub-problemas que podem ser resolvidos de forma independente em paralelo por blocos de *threads*, e cada sub-problema em pedaços menores que podem ser resolvidos de forma cooperativa em paralelo por todos as *threads* dentro do bloco.

Essa decomposição preserva a finalidade inicial do programa permitindo que *threads* cooperem na resolução dos sub-problemas, e ao mesmo tempo habilitando automaticamente a escalabilidade do problema. Cada bloco de *threads* pode ser designado para ser executado em qualquer multiprocessador disponível dentro de uma GPU, e em qualquer ordem, concorrentemente ou sequencialmente.

Este modelo de programação escalável permite que um programa CUDA seja executado em

uma variedade de *hardwares* CUDA, desde placas de alto-desempenho até placas mais baratas, simplesmente deixando com o *runtime* CUDA a responsabilidade de determinar o total de multi-processadores fisicamente disponíveis em um *hardware*. Como pode ser exemplificado na Figura 2.4.

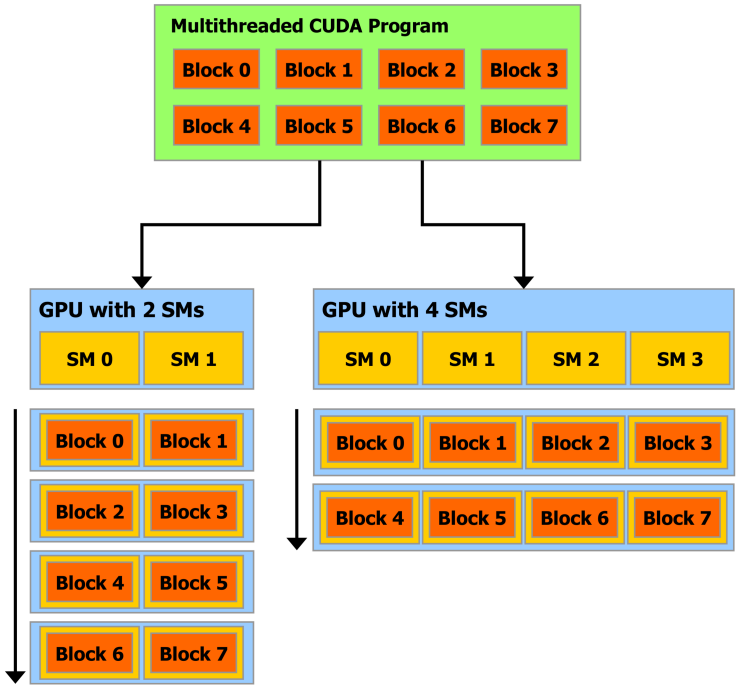


Figura 2.4: Escalabilidade Automática (NVIDIA, 2012a)

2.4.1 *Kernels* e Hierarquia de *Threads*

No modelo de programação CUDA o conjunto de instruções que serão executados devem ser distinguidos e agrupados por funções. Existem funções que são executadas na CPU, ou *host*, funções que são executadas na GPU, ou *device*. A função que o *host* chama para dar início a uma execução no *device* é chamada de *kernel*. A distinção entre as funções é feita por qualificadores explicados no Tópico 2.5.3.

Um *kernel* é mapeado por um conjunto de *threads*, que são agrupadas em blocos (*blocks*), e estes blocos são agrupados em um *grid*. Essa configuração contendo o número de blocos em cada *grid* e o número de *threads* em cada bloco é passada na chamada do *kernel*, conforme explicado no Tópico 2.5.5. *Threads* pertencentes a um bloco rodam em um mesmo SM, mas uma SM pode rodar múltiplos blocos concorrentemente. Internamente ao *device*, os blocos são divididos em grupos de 32 *threads*, chamados *warps*[4], as *threads* pertencentes a um *warp* são executadas em sincronia.

As dimensões do *grid* e dos blocos podem ser unidimensionais, bidimensionais ou tridimensionais, no exemplo da Figura 2.5 está definido um *grid* bidimensional, formado por blocos também bidimensionais. Dentro de cada *thread* estão disponíveis variáveis *built-in* que designam unica-

[4]Este termo vem da tecelagem, considerado pela NVIDIA a primeira tecnologia *multi-thread*.

mente a qual bloco ela pertence e o seu próprio identificador, além das dimensões dos blocos e *grids*, conforme explicado no Tópico 2.5.4.

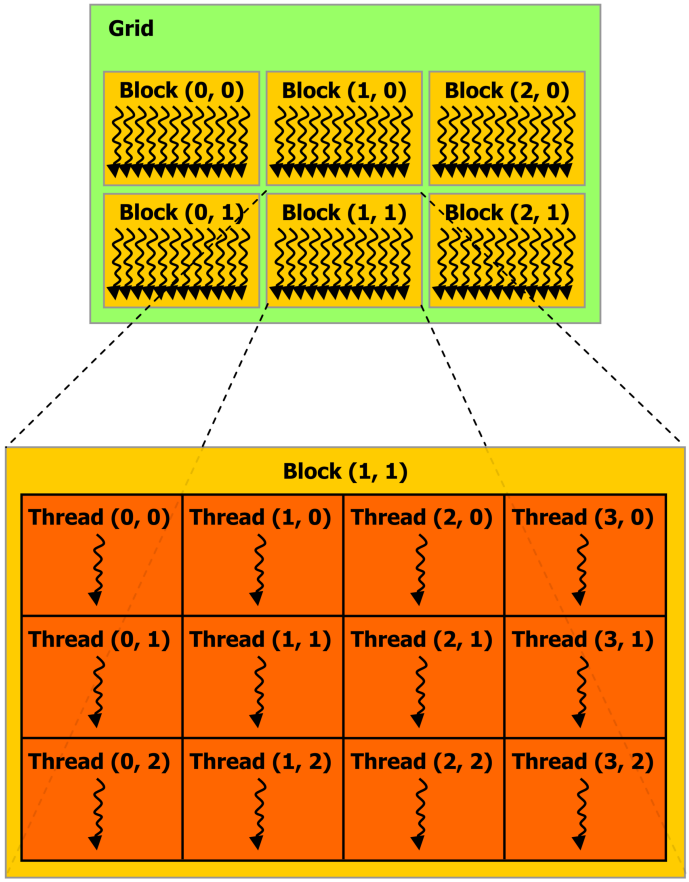


Figura 2.5: *Grid* formado por blocos de *threads* (NVIDIA, 2012a)

2.4.2 Hierarquia de Memória

Otimizar o desempenho de aplicações CUDA geralmente envolve otimizações no acesso de dados, que inclui o uso apropriado dos diferentes tipos de memória. Cada tipo de memória dentro de uma GPU possui uma forma distinta de leitura e escrita, tempo de vida, escopo e tempo de acesso. A Tabela 2.1 apresenta os diferentes tipos de memória presentes em um *device* CUDA.

Os bancos de registradores possuem o melhor tempo de acesso porém quanto mais *threads* são executadas menos registradores ficam disponíveis para as *threads*. Encontrar o número ótimo de *threads* rodando concorrentemente em um SM é um passo crucial para atingir-se um bom desempenho.

Cada SM possui vários KB de memória compartilhada (*shared*) de acesso rápido, acessível por todas as *threads* em um SM. Esta memória serve para troca de dados entre *threads* de um bloco com latência de acesso tão baixa quanto a dos registradores, mas a vazão (*throughput*) depende do padrão de acesso. A memória compartilhada é organizada em 16 bancos de memória, se duas *threads* pertencentes a um mesmo *half-warp* (16 *threads*) leem ou armazenam em diferentes

Tabela 2.1: Tipos de memórias

Memória	***on/off chip***	**Acesso**	**Escopo**	**Tempo de Vida**
Registrador	*on*	leitura/escrita	*thread*	*thread*
Local	*off*	leitura/escrita	*thread*	*thread*
Compartilhada	*on*	leitura/escrita	bloco	bloco
Global	off	leitura/escrita	*grids* + *host*	alocação do *host*
Constante	off	leitura	*grids* + *host*	alocação do *host*
Textura	off	leitura	*grids* + *host*	alocação do *host*

endereços dentro do mesmo banco de memória na mesma instrução, então essas requisições são serializadas. Este evento é chamado de conflito de banco (*bank conflicts*).

2.4.3 Transferência de dados entre GPU e CPU

Ambos *host* e *device* possuem suas próprias DRAM[5]. Todo o gerenciamento de memória realizado por um aplicativo (global, constantes e texturas), passa pelo *runtime* do CUDA, incluindo alocação, desalocação, e transferência de dados entre as memórias do *host* e do *device*.

Comunicação entre CPU e GPU é realizada pela transferência de dados entre a memória do *host* e a memória do *device* ou mapeando uma memória *page-locked* (também chamada de *pinned*) no espaço de endereços da GPU. Transferência de dados assíncrona entre memória *page-locked* do *host* e a memória do *device* pode ser sobreposta (*overlap*) com execuções na CPU, e em dispositivos CUDA desde a *compute capability* 1.1 essas transferências podem ser sobrepostas com execuções na GPU.

2.4.4 Barreira de Sincronização

Se uma *thread* necessita visualizar o valor de uma variável *shared* que foi escrita por outra *thread*, pode ocorrer uma inconsistência, pois a *thread* com a leitura pode acontecer antes da *thread* de escrita. Para evitar esse problema é necessário usar uma barreira de sincronização antes da leitura da variável, desta forma, quando a execução do programa encontrar essa barreira de sincronização ele aguardará a chegada de todas as *threads* do bloco que estão em execução neste ponto para poder prosseguir com a execução do programa, e consequentemente com a leitura da variável.

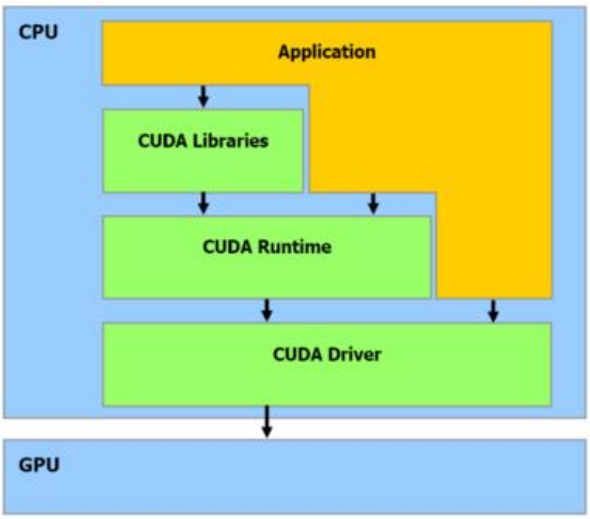


Figura 2.6: Pilha de *Software* CUDA (NVIDIA, 2008)

2.5 CUDA C

2.5.1 Pilha de *software*

A pilha de *software* CUDA (Figura 2.6) é composta por:

- Bibliotecas CUDA: conjunto de bibliotecas de uso genérico como CUBLAS (implementação da *Basic Linear Algebra Subprograms*), CUFFT (implementação da Transformada Rápida de Fourier), etc;
- *Runtime* CUDA: API de alto-nível que facilita a gerência de código para o *device* provendo inicialização implícita, gerência de contexto e gerência de módulos;
- *Driver* CUDA: API de baixo-nível, requer mais codificação, é mais complexa de programar e depurar, mas prove um nível melhor de controle e é independente de linguagem pois trata somente com binários CUDA.

O programador deve decidir qual API utilizar pois são mutuamente excludentes (NVIDIA, 2008).

2.5.2 Compilação

Primeiramente, o compilador separa as partes do programa que rodam no *host* das partes que rodam no *device*. As partes que rodam no *host* são compiladas com os compiladores C ou C++ para a respectiva arquitetura do *host*. As partes que rodam no *device* são traduzidas para uma linguagem intermediária chamada *Parallel Thread eXecution*(PTX). Esse código PTX é compatível entre as revisões menores de uma determinada *compute capability*. O *driver* da GPU contém um

[5] *Dynamic random-access memory*

compilador *JIT*[6] para essa linguagem intermediária. Códigos que precisam rodar em *hardwares* com diferentes *compute capability* precisam gerar diferentes versões desse código PTX, a compilação final para o código binário será executada pelo respectivo *driver*. Esta última compilação pode também ser realizada *offline*, para produzir binários para uma arquitetura específica da GPU.

2.5.3 Qualificadores de funções

Especificam onde uma função é executada e de onde pode ser chamada (NVIDIA, 2012a, pág. 81):

- __device__ : declara funções que são executadas no *device*, e só podem ser chamadas a partir do *device*;
- __global__ : declara funções *kernel*, são executadas no *device*, e só podem ser chamadas a partir do *host*. Devem apresentar retorno do tipo *void*. Deve ser especificada a sua configuração de execução, conforme Tópico 2.5.5;
- __host__ : declara funções que são executadas no *host*, e só podem ser chamadas a partir do *host*.

Caso a função seja declarada sem nenhum qualificador, ela será considerada do tipo __host__ por padrão. Nos casos, em que a função deve ser compilada para o *device* e para o *host* os qualificadores __host__ e __device__ são usados combinados. Já os qualificadores __global__ e __host__ não podem ser usados juntos.

2.5.4 Variáveis *Built-in*

Servem para especificar o *grid*, a dimensão dos blocos e os índices das *threads*. Sendo válidas apenas dentro das funções que são executadas pelo *device*(GPU) (NVIDIA, 2012a, pág. 86). Essas variáveis são:

- *gridDim*: Variável do tipo *dim3* e corresponde a dimensão do *grid*;
- *blockIdx*: Variável do tipo *uint3* e corresponde ao índice do bloco no *grid*;
- *blockDim*: Variável do tipo *dim3* e corresponde a dimensão do bloco;
- *threadIdx*: Variável do tipo *uint3* e corresponde ao índice da *thread* no bloco;
- *warpSize*: Variável do tipo básico *int* e contém o tamanho do *warp* em *threads*.

O tipo *uint3* é utilizado para designar um vetor contendo três elementos do tipo básico *int*, cada um dos elementos é acessado pela notação *var.x*, *var.y* e *var.z*. O tipo *dim3* corresponde

[6]Compilador *just-in-time* é um tradutor que converte, em tempo de execução, instruções de um formato para outro, por exemplo, no caso da linguagem Java, de *bytecode* para código de máquina.(WIKIPÉDIA, 2012a)

a um vetor de inteiros do tipo *uint3* especializado para a definição de dimensões, os elementos não especificados no vetor são inicializados com o valor 1. Ambos os tipos estão disponíveis na biblioteca de *runtime* do CUDA. Para nenhuma das variáveis *built-in* é possível realizar atribuição de valores.

2.5.5 Chamada do Kernel

As funções declaradas com o qualificador __global__ representam o *kernel* e na sua chamada devem ser definidos alguns parâmetros para a execução da função (NVIDIA, 2012a, pág. 111). Essa configuração é expressa da seguinte forma $\lll Dg, Db, NS, S \ggg$, onde :

- Dg: variável do tipo *dim3* e especifica a dimensão e o tamanho do *grid*;
- Db: variável do tipo *dim3* e especifica a dimensão e o tamanho de cada bloco;
- NS: variável do tipo $size\_t$ e especifica o tamanho, em *bytes*, do espaço que será alocado dinamicamente na memória compartilhada por bloco, esse argumento é opcional e seu valor padrão é 0;
- S: variável do tipo $cudaStream\_t$ e especifica o *stream* associado ao *kernel*, esse argumento é opcional e seu valor padrão é 0.

2.5.6 Qualificadores de Variáveis

Especificam a localização na memória de uma variável. Os seguintes qualificadores de variáveis estão disponíveis (NVIDIA, 2012a, pág. 82):

- __device__ : declara variáveis que devem residir no *device* e na memoria global;
- __constant__ : declara variáveis que devem residir no espaço de memoria de constante;
- __shared__ : declara variáveis que devem residir no espaço de memória de compartilhado.

2.5.7 Sincronização de *Threads*

A sincronização de *threads* explícita é realizada pelo comando __syncthreads(), esta é uma operação que pode impactar no desempenho, pois força o SM a ficar ocioso aguardando as outras *threads* chegarem ao ponto de sincronização.

Como um *warp* executa uma instrução simples por vez e *threads* dentro de um *warp* são implicitamente sincronizadas, isto pode ser usado para omitir as instrução __syncthreads() visando melhorar o desempenho (NVIDIA, 2012a, pág. 77).

Capítulo 3

AES - *Advanced Encryption Standard*

Neste capítulo é apresentado algoritmo criptográfico AES e alguns aspectos de sua implementação em software.

3.1 Introdução

Em Criptografia, o *Advanced Encryption Standard* (AES, ou Padrão de Criptografia Avançada), também conhecido por Rijndael[1], é uma cifra de bloco adotada como padrão de criptografia pelo governo dos Estados Unidos. O AES foi anunciado pelo NIST (Instituto Nacional de Padrões e Tecnologia dos EUA) como U.S. FIPS PUB (FIPS 197) em 26 de Novembro de 2001, depois de 5 anos de um processo de padronização (NIST, 2001a). Tornou-se um padrão efetivo em 26 de Maio de 2002. É amplamente utilizado em todo o ecossistema de *hardware* e *software* para proteger o tráfego de rede, dados pessoais, e infraestrutura corporativa de TI.

3.2 Descrição da Cifra

Todos os *bytes* no algoritmo AES são interpretados como elementos de um corpo finito. Esses elementos são adicionados ou multiplicados seguindo as definições matemáticas descritas em NIST (2001a, pág. 10-13). Os seguintes operadores são definidos, $\oplus$ para adição e $\bullet$ para multiplicação.

O AES opera sobre um arranjo bidimensional de *bytes* com 4x4 posições, denominado de estado (*state*).

O tamanho do bloco de entrada, bloco de saída e do estado é de 128 *bits*. Este tamanho é representado por $N_b = 4$, que reflete o número de palavras de 32 *bits* que o compõe, ou o número de colunas da visão bidimensional do estado.

O tamanho da chave de cifração/decifração é de 128, 192 ou 256 *bits*. Este tamanho é representado por $N_k = 4$, 6, ou 8, que reflete o número de palavras de 32 *bits* que o compõe. É comum denominar o AES dependentemente do tamanho de sua chave, surgindo os nomes AES-128, AES-192 e AES-256.

[1] Nome formado da fusão dos nomes de seus criadores Vincent Rijmen e Joan Daemen

O número de rodadas (*rounds*) a ser executado sobre um estado é dependente do tamanho da chave (Tabela 3.1). Este número é representado por $N_r = 10$, 12 ou 14.

Tabela 3.1: AES: Combinações de chave, bloco e rodadas

	Tamanho da Chave (N_k)	Tamanho do Bloco (N_b)	Número de rodadas (N_r)
AES-128	4	4	10
AES-192	6	4	12
AES-256	8	4	14

O processo de cifração, descrito no Algoritmo 3.1, faz uso da chave de cifração expandida, conforme o Algoritmo 3.4, e de 4 transformações que são aplicadas sobre o estado, conforme Tópicos 3.2.1, 3.2.2, 3.2.3 e 3.2.4. Em NIST (2001a) esse processo é denominado *"Cipher"*.

O processo de decifração, descrito no Algoritmo 3.2, faz uso da mesma chave de cifração expandida gerada pelo Algoritmo 3.4 e das transformações inversas da cifração que são aplicadas sobre o estado, conforme Tópicos 3.2.5, 3.2.6, 3.2.7 e 3.2.8. Em NIST (2001a) esse processo é denominado *"Inverse Cipher"*.

Um processo equivalente para a decifração é fornecido em NIST (2001a). Este processo (Algoritmo 3.3) oferece uma estrutura mais eficiente do que o processo descrito no Algoritmo 3.2, porém exige uma alteração na chave de cifração expandida, aplicando os Algoritmos 3.4 e 3.5 em sequência para gerar as chaves de rodada que serão usadas neste algoritmo. Em NIST (2001a) esse processo é denominado *"Equivalent Inverse Cipher"*.

Código fonte 3.1: Pseudocódigo para cifração do AES (NIST, 2001a)

```
Cipher(byte in[4*Nb], byte out[4*Nb], word w[Nb*(Nr+1)])
begin
  byte state[4,Nb]

  state = in

  AddRoundKey(state, w[0, Nb - 1])

  for round = 1 step 1 to Nr - 1
    SubBytes(state)
    ShiftRows(state)
    MixColumns(state)
    AddRoundKey(state, w[round*Nb, (round + 1)*Nb - 1])
  end for

  SubBytes(state)
  ShiftRows(state)
  AddRoundKey(state, w[Nr*Nb, (Nr + 1)*Nb - 1])

  out = state
end
```

Código fonte 3.2: Pseudocódigo para decifração do AES (NIST, 2001a)

```
InvCipher(byte in[4*Nb], byte out[4*Nb], word w[Nb*(Nr+1)])
```

```
begin
  byte state[4,Nb]

  state = in

  AddRoundKey(state, w[Nr*Nb, (Nr+1)*Nb-1])

  for round = Nr-1 step -1 downto 1
    InvShiftRows(state)
    InvSubBytes(state)
    AddRoundKey(state, w[round*Nb, (round+1)*Nb-1])
    InvMixColumns(state)
  end for

  InvShiftRows(state)
  InvSubBytes(state)
  AddRoundKey(state, w[0, Nb-1])

  out = state
end
```

Código fonte 3.3: Pseudocódigo equivalente para decifração do AES (NIST, 2001a)

```
EqInvCipher(byte in[4*Nb], byte out[4*Nb], word dw[Nb*(Nr+1)])
begin
  byte state[4,Nb]
  state = in
  AddRoundKey(state, dw[Nr*Nb, (Nr+1)*Nb-1])

  for round = Nr-1 step -1 downto 1
    InvSubBytes(state)
    InvShiftRows(state)
    InvMixColumns(state)
    AddRoundKey(state, dw[round*Nb, (round+1)*Nb-1])
  end for

  InvSubBytes(state)
  InvShiftRows(state)
  AddRoundKey(state, dw[0, Nb-1])

  out = state
end
```

3.2.1 Transformação AddRoundKey

Nesta transformação o estado é combinado com a chave de rodada, usando uma operação XOR *byte* a *byte*.

3.2.2 Transformação SubBytes

Nesta transformação cada *byte* do estado é substituído por outro de acordo com uma tabela de referência, denominada S-Box (Figura 3.1). A tabela S-Box é inversível e é construída calculando o inverso multiplicativo em $GF(2^8)$ para cada um dos *bytes*, sendo que *"00"* é mapeado em si mesmo, e então aplicando uma transformação afim em $GF(2)$, descrita em NIST (2001a).

		y															
		0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
x	0	63	7c	77	7b	f2	6b	6f	c5	30	01	67	2b	fe	d7	ab	76
	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 3.1: Tabela S-BOX: valores de substituição para o *byte* xy (no formato hexadecimal) (NIST, 2001a)

3.2.3 Transformação ShiftRows

Nesta transformação são alteradas as linhas do estado, deslocando os *bytes* em cada linha de um determinado número de posições. A primeira linha fica inalterada. Cada *byte* da segunda linha é rotacionado à esquerda de uma posição. Similarmente, a terceira e quarta linhas são rotacionadas à esquerda de duas e de três posições respectivamente.

3.2.4 Transformação MixColumns

Nesta transformação cada coluna do estado é combinada usando uma transformação linear inversível. Cada coluna é tratada como um polinômio em $GF(2^8)$ e é então multiplicado em módulo $x^4 + 1$ pelo polinômio fixo $a(x) = \{03\}x^3 + \{01\}x^2 + \{01\}x + \{02\}$, sendo os coeficientes de $a(x)$ *bytes* no formato hexadecimal no corpo finito $GF(2^8)$.

3.2.5 Transformação Inversa da AddRoundKey

A transformação inversa da AddRoundKey é ela mesma, pois trata-se de uma simples operação XOR *byte* a *byte*.

3.2.6 Transformação invSubBytes

Nesta transformação cada *byte* do estado é substituído por outro de acordo com uma tabela inversa da tabela referência S-Box(Figura 3.2). Esta tabela é obtida pela inversa da transformação afim em $GF(2)$ descrita em NIST (2001a) seguida pelo cálculo do inverso multiplicativo em $GF(2^8)$.

		y															
		0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
x	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
	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 3.2: Tabela Inversa da S-BOX: valores de substituição para o *byte* xy (no formato hexadecimal)(NIST, 2001a)

3.2.7 Transformação invShiftRows

Nesta transformação são alteradas as linhas do estado, deslocando os *bytes* em cada linha de um determinado número de posições. A primeira linha fica inalterada. Cada *byte* da segunda linha é rotacionado à direita de uma posição. Similarmente, a terceira e quarta linhas são rotacionadas à direita de duas e de três posições respectivamente.

3.2.8 Transformação invMixColumns

Nesta transformação cada coluna do estado é combinada usando uma transformação linear inversível. Cada coluna é tratada como um polinômio em $GF(2^8)$ e é então multiplicado em módulo x^4+1 pelo polinômio fixo $a^{-1}(x) = \{0b\}x^3 + \{0d\}x^2 + \{09\}x + \{0e\}$, sendo os coeficientes de $a^{-1}(x)$ *bytes* no formato hexadecimal no corpo finito $GF(2^8)$.

3.2.9 Expansão da Chave

A chave de cifração é expandida usando-se o Algoritmo 3.4, neste processo são gerados um total de $N_b * (N_r + 1)$ palavras de 32 *bits*.

A função SubWord() recebe 4 *bytes* e aplica a tabela S-BOX (Figura 3.1) em cada um dos *bytes* para gerar sua saída.

A função RotWord() recebe uma palavra de 4 *bytes* e a rotaciona à esquerda para gerar sua saída.

O vetor constante de rodada, $Rcon[i]$, contém os valores $[\{02\}^{i-1}, \{00\}, \{00\}, \{00\}]$, onde as operações de multiplicação ocorrem no corpo $GF(2^8)$ módulo o polinômio $m(x) = x^8 + x^4 + x^3 + x + 1$, e o índice i inicia-se em 1.

Código fonte 3.4: Pseudocódigo para expansão de chave de cifração do AES (NIST, 2001a)

```
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
```

A chave de decifração usada no Algoritmo 3.3 é gerada aplicando primeiramente o Algoritmo 3.4 e em seguida o Algoritmo 3.5. Cabe ressaltar uma mudança na transformação InvMixColumns usada nesse algoritmo, a transformação original descrita no Tópico 3.2.8 opera sobre um vetor bidimensional, o estado, no Algoritmo 3.5 a transformação InvMixColumns opera sobre um vetor unidimensional.

Código fonte 3.5: Pseudocódigo para expansão da chave da decifração(equivalente) AES (NIST, 2001a)

```
for i = 0 step 1 to (Nr+1)*Nb-1
  dw[i] = w[i]
end for

for round = 1 step 1 to Nr-1
  InvMixColumns(dw[round*Nb, (round+1)*Nb-1])
end for
```

3.3 Modos de Operação

Existem cinco modos de operação para as cifras de bloco recomendados em NIST (2001b): *Electronic Codebook* (ECB), *Cipher Block Chaining* (CBC), *Cipher Feedback* (CFB), *Output Feedback* (OFB) e *Counter* (CTR). Para as implementações do algoritmo AES que fazem uso de paralelismo são mais apropriados os seguintes modos de operação, ECB e CTR para cifração e ECB, CBC, CFB e CTR para decifração. Cabe ressaltar que o modo de operação ECB não é seguro para aplicações práticas, contudo é adequado para fins de medição de desempenho de cifração/decifração.

3.3.1 *Padding*

Os modos de operação ECB e CBC requerem que a sua entrada seja um múltiplo do tamanho do bloco. Se o tamanho do texto a ser cifrado não é um múltiplo do tamanho bloco é necessário completar o bloco antes da cifração (*padding*), e no momento da decifração esse *padding* deve ser removido após o final do processo.

Em Kaliski (2000) e Kaliski (1998) é proposto o seguinte método de *padding*:

1. Determinar TAM, o tamanho do texto a ser cifrado;
2. Determinar PAD, o quanto falta para TAM ser um múltiplo do tamanho do bloco (16 *bytes* no caso do AES);
3. Completar o último bloco do texto a ser cifrado com PAD *bytes* de valor PAD;
4. Caso TAM seja um múltiplo de 16 (dezesseis), um bloco adicional contendo 16 *bytes* de valor 16 é concatenado ao texto a ser cifrado .

3.4 *Table Look-up*

Em Daemen e Rijmen (1999) no tópico que trata sobre aspectos de implementação do AES em processadores de 32 *bits* é sugerida a combinação dos diferentes passos de uma rodada do AES em um conjunto de tabelas. São definidas 4 tabelas, T_0, T_1, T_2 e T_3 (Figura 3.3), cada uma contendo 256 palavras de 4 *bytes* totalizando 4KiB de espaço total. De modo análogo ao processo de cifração, as tabelas da Figura 3.4 representam as *table look-up* para a decifração. Nas Figuras 3.3 e 3.4 a função $S\,[a]$ é a aplicação da transformação SubBytes, a função $iS\,[a]$ é a aplicação da transformação invSubBytes e o operador $\bullet$ é a multiplicação no corpo finito $GF(2^8)$ definido em NIST (2001a, pág. 10-13).

$$T_0\,[a] = \begin{bmatrix} S\,[a] \bullet 02 \\ S\,[a] \bullet 01 \\ S\,[a] \bullet 01 \\ S\,[a] \bullet 03 \end{bmatrix} \quad T_1\,[a] = \begin{bmatrix} S\,[a] \bullet 03 \\ S\,[a] \bullet 02 \\ S\,[a] \bullet 01 \\ S\,[a] \bullet 01 \end{bmatrix}$$

$$T_2\,[a] = \begin{bmatrix} S\,[a] \bullet 01 \\ S\,[a] \bullet 03 \\ S\,[a] \bullet 02 \\ S\,[a] \bullet 01 \end{bmatrix} \quad T_3\,[a] = \begin{bmatrix} S\,[a] \bullet 01 \\ S\,[a] \bullet 01 \\ S\,[a] \bullet 03 \\ S\,[a] \bullet 02 \end{bmatrix}$$

Figura 3.3: AES cifração *Table Look-up* (DAEMEN; RIJMEN, 1999)

Considerando $a_{i,j}$ o estado na entrada de uma rodada, $b_{i,j}$ o estado na saída de uma rodada e k_j a chave expandida da rodada, sendo $0 <= i, j < 4$ com i representando a linha do estado e j a coluna, e usando essas tabelas (Figura 3.3), tem-se que a transformação desta rodada pode ser expressa pela Figura 3.5.

$$Td_0[a] = \begin{bmatrix} iS[a] \bullet 0e \\ iS[a] \bullet 09 \\ iS[a] \bullet 0d \\ iS[a] \bullet 0b \end{bmatrix} \quad Td_1[a] = \begin{bmatrix} iS[a] \bullet 0b \\ iS[a] \bullet 0e \\ iS[a] \bullet 09 \\ iS[a] \bullet 0d \end{bmatrix}$$

$$Td_2[a] = \begin{bmatrix} iS[a] \bullet 0d \\ iS[a] \bullet 0b \\ iS[a] \bullet 0e \\ iS[a] \bullet 09 \end{bmatrix} \quad Td_3[a] = \begin{bmatrix} iS[a] \bullet 09 \\ iS[a] \bullet 0d \\ iS[a] \bullet 0b \\ iS[a] \bullet 0e \end{bmatrix}$$

Figura 3.4: AES decifração *Table Look-up* (DAEMEN; RIJMEN, 1999)

$$\begin{bmatrix} b_{0,j} \\ b_{1,j} \\ b_{2,j} \\ b_{3,j} \end{bmatrix} = T_0[a_{0,j}] \oplus T_1\left[a_{1,(j-1) \mod 4}\right] \oplus T_2\left[a_{2,(j-2) \mod 4}\right] \oplus T_3\left[a_{3,(j-3) \mod 4}\right] \oplus k_j$$

Figura 3.5: Rodada da cifração do AES usando *Table Look-up* (DAEMEN; RIJMEN, 1999)

Pode-se observar que as entradas $T_0[a]$, $T_1[a]$, $T_2[a]$ e $T_3[a]$ são versões rotacionadas uma das outras, para todos os valores de a. Consequentemente, ao custo de três rotações adicionais para cada coluna por rodada, a implementação de *table look-up* pode ser realizada com somente uma tabela. Ainda é possível realizar a geração dessas tabelas durante a execução ao invés de inseri-las no código.

Na rodada final, a transformação MixColumns não é utilizada, isso significa que deve ser utilizada a tabela SBox ao invés das tabelas T. A necessidade de uma tabela adicional pode ser eliminada pela extração da tabela SBOX a partir da tabela T com uso de mascaramento.

3.5 *Bit-Slicing*

Bit-Slicing é uma técnica proposta por Biham (1997) para o algoritmo DES[2]. Esta técnica enxerga um processador de W *bits* como um computador paralelo SIMD capaz de executar W operações sobre 1 *bit* simultaneamente, para isso um operando deve conter W *bits* de W diferentes entradas. Esses operandos são montados da seguinte forma, W diferentes entradas são selecionadas e arranjadas sendo que a primeira palavra do rearranjo contém o primeiro *bit* de cada uma das W entradas, a segunda palavra contém o segundo *bit* de cada uma das W entradas, e assim por diante. Após esse rearranjo das entradas, os operandos passam por um conjunto de circuitos lógicos que representam as transformações ocorridas *bit* a *bit*.

Rebeiro e Devi (2006), Matsui e Nakajima (2007) e Kasper e Schwabe (2009) propõem o uso da técnica de *bit-slicing* para o algoritmo AES. Desta forma as transformações do AES, são descritas na forma de circuitos lógicos que operam *bit* a *bit* dos estados. Esta implementação é imune aos

[2] *Data Encryption Standard*, algoritmo antecessor do AES

ataques de *cache-timing*, diferentemente das implementações usando *table look-up* sugeridas em (DAEMEN; RIJMEN, 1999) que são suscetíveis a esses ataques (BERNSTEIN, 2005; OSVIK; TROMER, 2005).

Kasper e Schwabe (2009) apresentam a implementação estado da arte do AES em processadores *Intel Core 2*. Neste trabalho são reportados os desempenhos de uma implementação de tempo constante do AES no modo CTR na cifração de pacotes de 40, 576, 1500 e 4096 *bytes* em 3 CPUs diferentes. A Tabela 3.2 exibe o desempenho do AES no modo CTR no processo de cifração de um pacote de 4096 *bytes* com uma chave de 128 *bits*, em todos os resultados foram utilizados somente um núcleo(*core*) da CPU e estão incluídos os tempos de transformação dos dados de entrada para o formato *bit-sliced* e a transformação da saída para o formato padrão.

Tabela 3.2: Resultados de Kasper e Schwabe (2009) para cifração AES-CTR (128 bits) de pacotes de 4096 *bytes*

CPU	**Frequência da CPU (MHz)**	**Desempenho da cifração (ciclos/*byte*)**	**Desempenho da cifração (Gbit/s)**
Intel Core 2 Quad Q6600	2404,102	9,32	2,063
Intel Core 2 Quad Q9550	2833	7,59	2,986
Intel Core i7 920	2668	6,92	3,084

3.6 AES-NI

Em 2010, a Intel lançou uma nova família de processadores, codinome *Westmere*, que inclui um conjunto de novas instruções denominadas *Intel® Advanced Encryption Standard (AES) New Instructions* (AES-NI). Essas instruções implementam todas as etapas do algoritmo AES em *hardware*, resultando em uma implementação do AES usando menos ciclos de relógio (*clock*) do que uma solução somente baseada em *software* (INTEL, 2010b).

O conjunto de instruções AES-NI é composto de seis instruções, quatro delas usadas para executar uma rodada da cifração ou decifração e as outras duas são usadas para geração de chaves de rodada. Segue uma descrição mais detalhada dessas novas instruções:

- AESENC: Esta instrução implementa uma rodada do processo de cifração, combinando as quatro transformações do algoritmo AES (ShiftRows, SubBytes, MixColumns e AddRoundKey) em uma instrução;
- AESENCLAST: Esta instrução implementa a última rodada do processo de cifração do algoritmo AES, combinando as transformações ShiftRows, SubBytes e AddRoundKey em uma instrução;
- AESDEC: Esta instrução implementa uma rodada do processo de decifração, combinando

as quatro transformações do algoritmo AES (InvShiftRows, InvSubBytes, InvMixColumns e AddRoundKey) em uma instrução;

- AESDECLAST: Esta instrução implementa a última rodada do processo de decifração do algoritmo AES, combinando as transformações InvShiftRows, InvSubBytes e AddRoundKey em uma instrução;
- AESKEYGENASSIST: Esta instrução implementa o processo de expansão da chave de cifração, gerando as chaves de rodada;
- AESIMC: Esta instrução é usada para converter as chaves de rodada usadas no processo de cifração para uma forma utilizável no processo de equivalente de decifração (*"Equivalent Inverse Cipher"*).

As novas instruções além de proverem desempenho, fornecem importantes benefícios de segurança. Executando em tempo igual independentemente do dado de entrada e não se utilizando de tabelas, são eliminados os principais ataques baseados em tempo e *cache* (BERNSTEIN, 2005; OSVIK; TROMER, 2005), ataques esses que assolam as implementações do AES que usam *table look-up* (INTEL, 2010b).

Em (INTEL, 2010a) são apresentados resultados de uma implementação do AES, sobre um núcleo da CPU, nos modos de operação ECB, CBC e CTR para os processos de cifração/decifração de um bloco de 1KiB usando as instruções AES-NI num processador da família *Westmere* com frequência de CPU de 2.67 GHz, conforme Tabela 3.3. Neste trabalho também é apresentado o desempenho do OpenSSL[3] versão 0.9.8h, sobre o mesmo processador, para o processo de cifração do AES nos modos ECB, CBC e CTR para 128 *bits* de chave, porém sem usar as instruções AES-NI, conforme Tabela 3.4.

3.7 AES e CUDA

Já foram realizados vários esforços na implementação do algoritmo AES na plataforma CUDA. Esta seção aborda algumas dessas tentativas visando levantar os caminhos trilhados e os resultados alcançados, que servirão de guia para os capítulos seguintes. Cabe salientar que nos vários trabalhos que tratam de AES e CUDA constantemente são utilizadas as expressões **tempo de cifração** e **tempo de cifração total**, a primeira referindo-se ao tempo que a placa CUDA demora para cifrar um texto e a segunda referindo-se ao tempo de cifração do texto mais o tempo das transferências de dados entre a CPU e a GPU. Esses dois tempos são ressaltados pois, como será visto, o tempo gasto nas transferências de dados entre CPU e GPU impacta fortemente no desempenho final do processo de cifração usando as placas CUDA.

[3]OpenSSL é uma implementação de código aberto dos protocolos *Secure Socket Layer* (SSL v2/v3) e *Transport Layer Security* (TLS v1), fornecendo também uma biblioteca escrita na linguagem C que implementa as funções básicas de criptografia e várias funções utilitárias.

Tabela 3.3: Resultados de INTEL (2010a) para cifração/decifração AES de um bloco de 1024 *bytes*

Modo de Operação	**Cifração / Decifração**	**Tamanho da Chave (bits)**	**Desempenho (ciclos/*byte*)**	**Desempenho (Gbit/s)**
ECB	Cifração	128	1,28	16,687
ECB	Decifração	128	1,26	16,952
CBC	Cifração	128	4,15	5,146
CBC	Decifração	128	1,30	16,430
CTR	Cifração	128	1,38	15,478
CTR	Decifração	128	1,38	15,478
ECB	Cifração	192	1,53	13,960
ECB	Decifração	192	1,51	14,145
CBC	Cifração	192	4,91	4,350
CBC	Decifração	192	1,53	13,960
CTR	Cifração	192	1,61	13,267
CTR	Decifração	192	1,61	13,267
ECB	Cifração	256	1,76	12,136
ECB	Decifração	256	1,76	12,136
CBC	Cifração	256	5,65	3,780
CBC	Decifração	256	1,78	12
CTR	Cifração	256	1,88	11,361
CTR	Decifração	256	1,88	11,361

Tabela 3.4: Resultados de INTEL (2010a) para cifração AES-128 usando OpenSSL versão 0.9.8h sem AES-NI

Modo de Operação	**Desempenho da cifração (ciclos/*byte*)**	**Desempenho da cifração (Gbit/s)**
ECB	15,38	1,388
CBC	17,66	1,209
CTR	19,60	1,089

3.7.1 Manavski (2007)

Considerado o primeiro trabalho com o uso de CUDA e o AES. Faz uso de uma implementação com tabelas de *look-up* armazenadas na memória constante da placa CUDA, cada bloco contém 256 *threads* e é responsável por processar 1024 *bytes*. Os dados de entrada são armazenados inicialmente na memória global e posteriormente guardados na memória compartilhada. As Tabelas 3.5 e 3.6 apresentam os tempos e desempenhos obtidos neste trabalho.

Apesar de não ser citado no documento o modo de operação utilizado, supõem-se que seja uma implementação usando ECB dado que o objetivo do trabalho era a mensuração do máximo desempenho.

O *hardware* utilizado foi a NVIDIA 8800 GTX, com 128 *cores*, 384 *bits* de interface de memória e 1350 MHz de frequência de operação.

Tabela 3.5: Resultados de Manavski (2007)

Algoritmo	**Tamanho Arquivo (MiB)**	**Tempo de cifração GPU (ms)**	**Desempenho da cifração (Gibit/s)**	**Tempo Total de cifração GPU (ms)**	**Desempenho da cifração total (Gibit/s)**
AES-128	1	1,04	7,512	3,50	2,232
AES-128	4	3,80	8,224	13,00	2,404
AES-128	8	7,55	8,278	25,00	2,500
AES-256	1	1,31	5,964	3,74	2,089
AES-256	4	4,78	6,538	13,90	2,248
AES-256	8	9,39	6,656	27,34	2,286

Tabela 3.6: Resultados de Manavski (2007) com desempenho em Gbit/s

Algoritmo	**Tamanho Arquivo (MiB)**	**Desempenho da cifração (Gbit/s)**	**Desempenho da cifração total (Gbit/s)**
AES-128	1	8,066	2,397
AES-128	4	8,830	2,581
AES-128	8	8,889	2,684
AES-256	1	6,404	2,243
AES-256	4	7,020	2,414
AES-256	8	7,147	2,455

3.7.2 Harrison e Waldron (2008)

Apresentam a implementação do AES no modo CTR, usando *table look-up* na memória compartilhada. Cada *thread* processa um bloco AES, usa memória *page-locked* para os dados de entrada/saída. Cada bloco contém 256 *threads* e cada uma é responsável por trazer da memória global para a compartilhada a tabela de *look-up*.

São apresentados os desempenhos máximos de 15423 Mbps para o processo de cifração e 6914 Mbps para o processo de cifração total, incluindo transferências de dados entre CPU e GPU. Na Figura 3.6 são apresentados outros resultados de desempenho da cifração AES em modo CTR para vários tamanhos de arquivo.

Apesar de não ser citado no documento o tamanho da chave utilizada, supõem-se que seja uma implementação de 128 *bits* dado que o objetivo do trabalho era a mensuração do máximo desempenho. Para os desempenhos máximos citados também não são apresentados os tamanhos dos dados, supõem-se pela Figura 3.6 que sejam para valores maiores que 64 MiB.

O *hardware* utilizado foi a NVIDIA 8800 GTX, com 128 *cores*, 384 *bits* de interface de memória e 1350 MHz de frequência de operação.

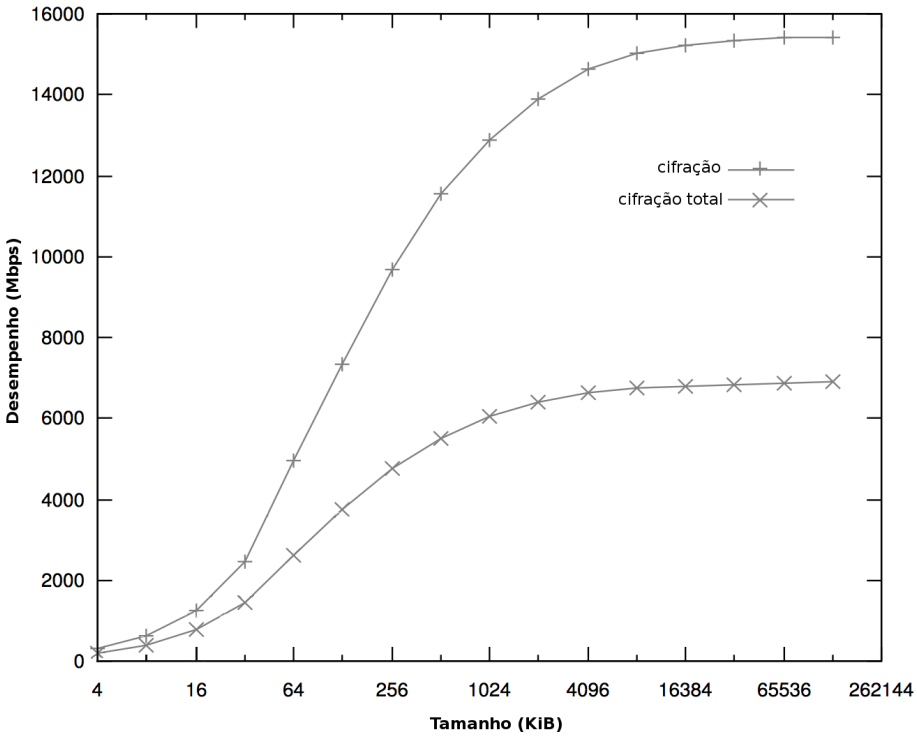


Figura 3.6: Resultados de Harrison e Waldron (2008) com desempenho em Mbit/s (adaptado)

3.7.3 Biagio et al. (2009)

Apresentam quatro implementações do AES no modo CTR, combinando duas variações de projeto do AES com duas possibilidades no local de armazenamento das tabelas de *look-up*.

As duas variações de projeto do AES são: a implementação denominada *fine-grained*, que explora o paralelismo interno de cada rodada do AES, porém necessita de uma barreira de sincronização entre as rodadas, e a implementação denominada *coarse-grained*, que ignora o paralelismo interno e foca no paralelismo provido pelo modo de operação, como ECB e CTR.

As duas possibilidades de armazenamento das tabelas de *look-up* são: na memória constante e na memória compartilhada.

Apesar de não ser citado no documento o tamanho da chave utilizada, supõem-se que seja uma implementação de 128 *bits* dado que o objetivo do trabalho era a mensuração do máximo desempenho.

Foram utilizadas duas placas gráficas para a mensuração dos resultados: NVIDIA 8800 GT com 112 *cores*, 256 *bits* de interface de memória e 1500 MHz de frequência de operação e a NVIDIA 8400 GS com 16 *cores*, 64 *bits* de interface de memória e 900 MHz frequência de operação. Somente serão exibidos os resultados da primeira placa gráfica, pois esta possui capacidade computacional análoga aos *hardwares* utilizados nos outros trabalhos aqui descritos.

Na Figura 3.7 são apresentados os resultados de desempenho da cifração AES em modo CTR para vários tamanhos de arquivo, para cada uma das implementações. Na Tabela 3.7 são apresentados os melhores resultados das quatro implementações para vários tamanhos de arquivos.

Para os desempenhos apresentados não fica explícito se foi considerado o processo de cifração ou o processo de cifração total, incluindo transferências de dados entre CPU e GPU, porém no Tópico "Trabalhos Relacionados" do referido artigo é feita uma comparação com o resultado obtido na implementação *coarse-grained* com uso de memória compartilha com o desempenho do processo de cifração citado em Harrison e Waldron (2008), supõem-se então que os desempenhos apresentados sejam para o processo de cifração, sem transferências de dados entre CPU e GPU.

Tabela 3.7: Resultados de Biagio et al. (2009) com o melhor desempenho em Mbit/s

Tamanho do Arquivo	**Desempenho da cifração (Gbit/s)**
32 KiB	2,917
128 KiB	6,591
32 MiB	12,075
128 MiB	12,412

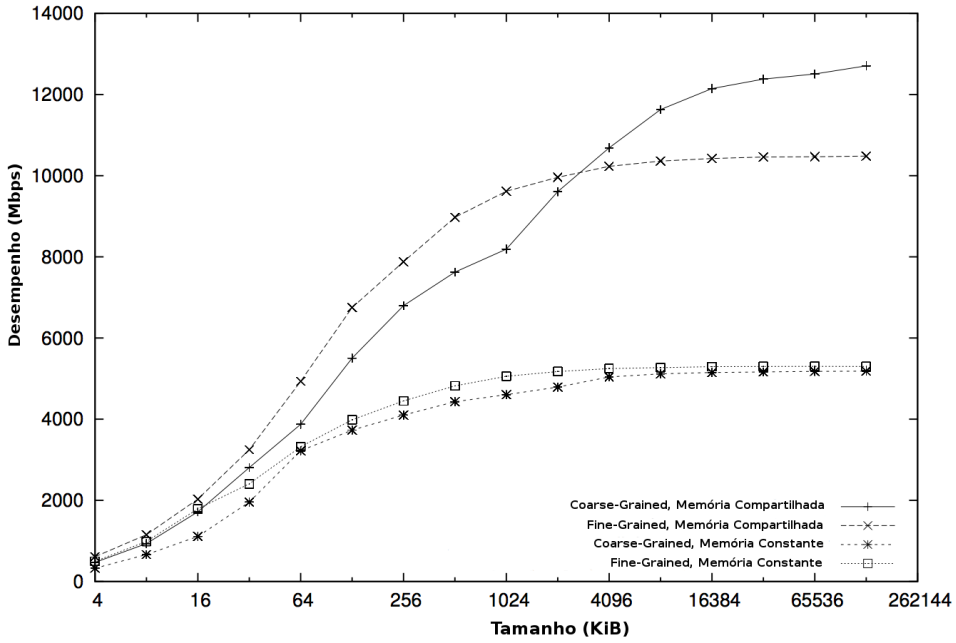


Figura 3.7: Resultados de Biagio et al. (2009) com desempenho em Mbit/s (adaptado)

3.7.4 Jang et al. (2011)

Apresenta uma implementação de um *proxy*[4] SSL usando GPU para acelerar os processos criptográficos. Na implementação do AES neste *proxy*, foi utilizado o modo de operação CBC para cifração/decifração com chave de 128 *bits*.

Esta implementação faz uso de tabelas de *look-up* armazenadas inicialmente na memória constante e cada *thread* ao iniciar a sua execução copia a tabela para a memória compartilhada. A chave de rodada é expandida quando utilizada, ao invés de usar as chaves pré-calculadas armazenadas na memória global, causando um aumento no processamento porém reduzindo o número de acessos à memória.

Somente o desempenho do AES no modo CBC para decifração foi reportado nesta dissertação, pois este processo possui ganho com a paralelização, diferentemente do modo CBC para cifração.

São apresentados os desempenhos máximos de 33,9 Gbps para o processo de cifração e 10 Gbps para o processo de cifração total, incluindo transferências de dados entre CPU e GPU, para um tamanho de arquivo de 64 MiB (conforme citado no trabalho, 4096 *flows* de 16 KiB).

O *hardware* utilizado foi a NVIDIA GTX 580, com 512 *cores*, 384 *bits* de interface de memória e 1544 MHz de frequência de operação.

No tópico "Discussão e Trabalhos Relacionados" Jang et al. (2011), é citado o desempenho de 32,8 Gbps para o modo de operação ECB com chave de 128 *bits*, sem informar o tamanho do arquivo de entrada, supõem-se que seja para o maior tamanho usado no trabalho pois este apresentaria o

[4] *Proxy* é um servidor intermediário que atende as requisições dos clientes, podendo realizar operações como busca em *cache* de páginas *Web*, e sendo necessário encaminha a requisição ao servidor principal.(WIKIPÉDIA, 2012d)

melhor resultado neste modo de operação (64 MiB), porém esse valor de desempenho não se refere ao *hardware* usado no trabalho e sim a uma única execução sobre uma NVIDIA GTX 285, com 240 *cores*, 512 *bits* de interface de memória e 1476 MHz de frequência de operação, visando prover um resultado para comparação com as implementações anteriores.

Capítulo 4

Desenvolvimento

Neste capítulo são apresentados a metodologia usada no desenvolvimento do trabalho, os utilitários que foram criados, e as implementações para CPU e GPU com suas descrições.

4.1 Introdução

Este trabalho objetiva em primeira instância comparar os tempos de execução do processo de cifração/decifração do algoritmo AES, para chaves de 128,192 e 256 *bits*, usando uma implementação de referência para CPU e sua versão portada para uma GPU CUDA.

A versão portada para a plataforma CUDA na realidade é representada por um conjunto de implementações que variam no uso dos recursos disponíveis na GPU, como quais memórias utilizar para armazenamento dos dados de entrada do algoritmo AES, configurações de chamadas do *kernel* e métodos de transferência de dados entre CPU e GPU. A análise comparativa entre essas várias versões e a seleção da versão com melhor desempenho representa um objetivo intermediário para a consecução do objetivo principal.

Para cada uma das plataformas, CPU e GPU, foram criadas versões que exploram características específicas destas plataformas (detalhadas nos tópicos a frente), cada uma destas versões representa uma biblioteca contendo as funções de expansão de chave, cifração e decifração para cada um dos tamanhos possíveis de chave do AES (128, 192 e 256 *bits*). Um conjunto de cinco utilitários se ligam a cada uma dessas bibliotecas para fornecer as interfaces que acionarão as suas funções.

As implementações que trabalham sobre arquivos, usando mais de um bloco do AES, usam o modo de operação ECB, por ser o modo mais simples paralelizável e que apresenta o menor tempo nas operações de cifração/decifração, e fazem uso de *padding*, seguindo o definido no Tópico 3.3.1.

4.2 Utilitários

Foram desenvolvidos cinco utilitários para explorar as funcionalidades providas pelas várias bibliotecas disponíveis.

4.2.1 aes_key

Aplicativo que recebe como entrada uma cadeia de caracteres (*string*) que representa a chave de cifração/decifração no formato hexadecimal, e retorna a expansão desta chave, com os tempos gastos para realizar as operações (Figura 4.1, Código fonte para CPU II.1 e CUDA II.8).

```
# aes_key 000102030405060708090a0b0c0d0e0f
Encryption key

00. 00010203        01. 04050607        02. 08090A0B        03. 0C0D0E0F
04. D6AA74FD        05. D2AF72FA        06. DAA678F1        07. D6AB76FE
08. B692CF0B        09. 643DBDF1        10. BE9BC500        11. 6830B3FE
12. B6FF744E        13. D2C2C9BF        14. 6C590CBF        15. 0469BF41
16. 47F7F7BC        17. 95353E03        18. F96C32BC        19. FD058DFD
20. 3CAAA3E8        21. A99F9DEB        22. 50F3AF57        23. ADF622AA
24. 5E390F7D        25. F7A69296        26. A7553DC1        27. 0AA31F6B
28. 14F9701A        29. E35FE28C        30. 440ADF4D        31. 4EA9C026
32. 47438735        33. A41C65B9        34. E016BAF4        35. AEBF7AD2
36. 549932D1        37. F0855768        38. 1093ED9C        39. BE2C974E
40. 13111D7F        41. E3944A17        42. F307A78B        43. 4D2B30C5
time(enc_key_128):        0.026587 ms


Decryption key

00. 13111D7F        01. E3944A17        02. F307A78B        03. 4D2B30C5
04. 13AA29BE        05. 9C8FAFF6        06. F770F580        07. 00F7BF03
08. 1362A463        09. 8F258648        10. 6BFF5A76        11. F7874A83
12. 8D82FC74        13. 9C47222B        14. E4DADC3E        15. 9C7810F5
16. 72E3098D        17. 11C5DE5F        18. 789DFE15        19. 78A2CCCB
20. 2EC41027        21. 6326D7D2        22. 6958204A        23. 003F32DE
24. A8A2F504        25. 4DE2C7F5        26. 0A7EF798        27. 69671294
28. C7C6E391        29. E54032F1        30. 479C306D        31. 6319E50C
32. A0DB0299        33. 2286D160        34. A2DC029C        35. 2485D561
36. 8C56DFF0        37. 825DD3F9        38. 805AD3FC        39. 8659D7FD
40. 00010203        41. 04050607        42. 08090A0B        43. 0C0D0E0F
time(dec_key_128):        0.003173 ms
```

Figura 4.1: Utilitário aes_key

4.2.2 aes_enc

Aplicativo que recebe como entrada uma cadeia de caracteres que representa a chave de cifração no formato hexadecimal, uma cadeia de caracteres que representa o bloco a ser cifrado no formato hexadecimal e retorna o bloco cifrado, com os tempos gastos para realizar as operações (Figura 4.2, Código fonte para CPU II.2 e CUDA II.9).

```
# aes_enc 00000000000000000000000000000000 00000000000000000000000000000000
time(enc_key_128):        0.004505 ms
time(enc_128):                0.005252 ms
66E94BD4EF8A2C3B884CFA59CA342B2E
```

Figura 4.2: Utilitário aes_enc

4.2.3 aes_dec

Aplicativo que recebe como entrada uma cadeia de caracteres que representa a chave de decifração no formato hexadecimal, uma cadeia de caracteres que representa o bloco a ser decifrado no formato hexadecimal e retorna o bloco decifrado, com os tempos gastos para realizar as operações (Figura 4.3, Código fonte para CPU II.3 e CUDA II.10).

```
# aes_dec 00000000000000000000000000000000 66E94BD4EF8A2C3B884CFA59CA342B2E
time(dec_key_128):        0.006781 ms
time(dec_128):                0.029226 ms
00000000000000000000000000000000
```

Figura 4.3: Utilitário aes_dec

4.2.4 aes_enc_file

Na versão para CPU, o aplicativo recebe como entrada uma cadeia de caracteres que representa o caminho do arquivo contendo a chave de cifração no formato binário, uma cadeia de caracteres que representa o caminho do arquivo de entrada a ser cifrado (*PLAINTEXT*) no formato binário e uma cadeia de caracteres que representa o caminho do arquivo de saída cifrado (*CIPHERTEXT*) no formato binário, exibe o tempo para realizar a operação (Figura 4.4, Código fonte II.4).

Na versão para CUDA, o aplicativo recebe como entrada uma cadeia de caracteres que representa o caminho do arquivo contendo a chave de cifração no formato binário, uma cadeia de caracteres que representa o caminho do arquivo de entrada a ser cifrado (*PLAINTEXT*) no formato binário, uma cadeia de caracteres que representa caminho do arquivo de saída cifrado (*CIPHERTEXT*) no formato binário, exibe os tempos para realizar as operações (enc_cuda: tempo para

a cifração na placa CUDA, enc_cuda_cp: tempo da cópia entre os dispositivos, enc_cuda_tt: tempo total com cifração e cópia entre os dispositivos) . Esta versão se subdivide em três implementações (Figura 4.5, Código fonte II.11) que utilizam diferentes propriedades da placa CUDA:

- aes_enc_file : uso normal de transferências de dados entre as memórias da CPU e GPU, além dos parâmetros anteriores recebe o número de *threads* por bloco;
- aes_enc_file_p : uso de memória *pinned* nas transferências de dados entre as memórias da CPU e GPU, além dos parâmetros anteriores recebe o número de *threads* por bloco;
- aes_enc_file_a : uso de memória *pinned* nas transferências de dados entre as memórias da CPU e GPU, uso de transferências assíncronas de blocos de dados com *overlap* de execução usando *streams*, além dos parâmetros anteriores recebe o número de *threads* por bloco, o número de blocos do AES por *streams* e o número de *streams*.

```
# aes_enc_file ../test_files/key128.bin ../test_files/8MBP.bin ../test_files/8MBP_1.bin
time(enc_128):                324.937592 ms        8388608 bytes
```

Figura 4.4: Utilitário aes_enc_file

```
# aes_enc_file ../test_files/key128.bin ../test_files/8MBP.bin ../test_files/8MBP_1.bin 256
time(enc_cuda_128):                4.462080 ms         256 threads        8388608 bytes
time(enc_cuda_cp_128):            41.846111 ms         256 threads        8388608 bytes
time(enc_cuda_tt_128):            44.219425 ms         256 threads        8388608 bytes

# aes_enc_file_p ../test_files/key128.bin ../test_files/8MBP.bin ../test_files/8MBP_1.bin
 256
time(enc_cuda_128):         4.456096 ms         256 threads        8388608 bytes
time(enc_cuda_cp_128):         7.306816 ms         256 threads        8388608 bytes
time(enc_cuda_tt_128):         9.509152 ms         256 threads        8388608 bytes

# aes_enc_file_a ../test_files/key128.bin ../test_files/8MBP.bin ../test_files/8MBP_1.bin
 256 1000 3
time(enc_cuda_tt_128):         8.958368 ms        256 threads        1000 blocos
                                              3 streams        8388608 bytes
```

Figura 4.5: Utilitário CUDA aes_enc_file

4.2.5 aes_dec_file

Na versão para CPU, o aplicativo recebe como entrada uma cadeia de caracteres que representa o caminho do arquivo contendo a chave de decifração no formato binário, uma cadeia de

caracteres que representa o caminho do arquivo de entrada a ser decifrado (*CIPHERTEXT*) no formato binário e uma cadeia de caracteres que representa o caminho do arquivo de saída decifrado (*PLAINTEXT*) no formato binário, exibe o tempo para realizar a operação (Figura 4.6, Código fonte II.5).

Na versão para CUDA, o aplicativo recebe como entrada uma cadeia de caracteres que representa o caminho do arquivo contendo a chave de decifração no formato binário, uma cadeia de caracteres que representa o caminho do arquivo de entrada a ser decifrado (*CIPHERTEXT*) no formato binário, uma cadeia de caracteres que representa caminho do arquivo de saída decifrado (*PLAINTEXT*) no formato binário, exibe os tempos para realizar as operações (dec_cuda: tempo para a decifração na placa CUDA, dec_cuda_cp: tempo da cópia entre os dispositivos, dec_cuda_tt: tempo total com decifração e cópia entre os dispositivos) . Esta versão se subdivide em três implementações (Figura 4.7, Código fonte II.12) que utilizam diferentes propriedades da placa CUDA:

- aes_dec_file : uso normal de transferências de dados entre as memórias da CPU e GPU, além dos parâmetros anteriores recebe o número de *threads* por bloco;
- aes_dec_file_p : uso de memória *pinned* nas transferências de dados entre as memórias da CPU e GPU, além dos parâmetros anteriores recebe o número de *threads* por bloco;
- aes_dec_file_a : uso de memória *pinned* nas transferências de dados entre as memórias da CPU e GPU, uso de transferências assíncronas de blocos de dados com *overlap* de execução usando *streams*, além dos parâmetros anteriores recebe o número de *threads* por bloco, o número de blocos do AES por *streams* e o número de *streams*.

```
# aes_dec_file ../test_files/key128.bin ../test_files/8MBP.bin ../test_files/8MBP_1.bin
time(enc_128):                324.937592 ms        8388608 bytes
```

Figura 4.6: Utilitário aes_dec_file

4.3 Plataforma

Os testes foram rodados em duas CPUs *Intel Core i7*, sem uso de *threading* ou instruções especiais de baixo nível do processador, e nas placas gráficas NVIDIA GeForce GT 460M e GTX 580. Detalhes das plataformas de *softwares* e *hardwares* utilizadas para a implementação das versões do AES para CPU e GPU são descritas no Anexo I.

Todos os códigos em CUDA foram compilados para a arquitetura *sm_20* (as placas gráficas usadas nos testes possuíam *compute capability* maior ou igual a 2.0), para evitar o atraso da compilação *JIT*, para maiores detalhes ver NVIDIA (2011a, Pág. 25).

```
# aes_dec_file ../test_files/key128.bin ../test_files/8MBP_1.bin ../test_files/8MBP_2.bin 256
time(dec_cuda_128):                    4.373856 ms         256 threads        8388608 bytes
time(dec_cuda_cp_128):           41.361088 ms         256 threads        8388608 bytes
time(dec_cuda_tt_128):           43.729218 ms         256 threads        8388608 bytes

# aes_dec_file_p ../test_files/key128.bin ../test_files/8MBP_1.bin ../test_files/8MBP_2.bin
 256
time(dec_cuda_128):                    4.368128 ms         256 threads        8388608 bytes
time(dec_cuda_cp_128):            7.183232 ms         256 threads        8388608 bytes
time(dec_cuda_tt_128):            9.379808 ms         256 threads        8388608 bytes

# aes_dec_file_a ../test_files/key128.bin ../test_files/8MBP_1.bin ../test_files/8MBP_2.bin
 256 1000 3
time(dec_cuda_tt_128):            8.829024 ms         256 threads        1000 blocos
                                                        3 streams        8388608 bytes
```

Figura 4.7: Utilitário CUDA aes_dec_file

4.4 Implementação para CPU

Na implementação em CPU foi realizada uma adaptação da versão de Barreto (2007). O código original está implementado em C++, utiliza *table look-up* para a implementação das rodadas do AES e permite escolher uma versão com o uso de laços (*for*) ou o seu desenrolamento (*unroll*), eliminando as instruções *for*. Na versão adaptada, o código foi migrado para C, os processos de expansão da chave de rodada, cifração e decifração foram divididos em implementações específicas para cada tamanho de chave, evitando testes desnecessários e facilitando o processo de desenrolamento (eliminação dos laços). Duas versões foram desenvolvidas: uma usando laços (Código fonte II.13) e outra com todos os laços desenrolados (*unroll*) (Código fonte II.14).

Esta implementação de Barreto (2007) foi utilizada como referência por não fazer uso de instruções de baixo nível do processador.

4.5 Implementação para GPU CUDA

Nas implementações em GPU CUDA foram realizadas várias combinações dentre as funcionalidades disponíveis, visando determinar qual a melhor versão. Elas se baseiam na implementação para CPU, variando as memórias utilizadas para o armazenamento da chave de cifração/decifração e das tabelas para *look-up* e a configuração de execução no particionamento do *grid* e blocos.

Todas as configurações foram implementadas com o uso da memória para texto de entrada/saída *pageable* e *pinned* e com transferência do texto de entrada/saída entre *host*/*device* completa, para o uso de memória *pageable* e *pinned*, e particionada, para o uso de memória *pinned* com vários *streams*, sobrepondo cópia na memória e execução (*overlap*). Os laços (*loops*) foram todos expandidos (*unroll*), com exceção da implementação cuda_aes_nn_global_roll.

O processo de expansão da chave de cifração é realizado pela CPU, por não representar ganho na sua execução na GPU. As chaves de rodada são transferidas da CPU para a memória global, constante ou de textura da GPU.

4.5.1 cuda_aes_nn_global_roll

Esta implementação (Código fonte II.15) possui as seguintes características:

- configuração de execução: a partir do número T, *threads* por bloco, recebido na chamada da função, o processamento é particionado em um *grid* com N blocos e cada bloco contendo T *threads*, sendo cada *thread* responsável por processar um bloco do AES;
- memória para *table look-up*: usada a memória global do dispositivo;
- memória para chave de rodada: usada a memória global do dispositivo;
- laços (*loops*): mantidos os laços na expansão da chave e nos processos de cifração/decifração.

4.5.2 cuda_aes_nn_global_unroll

Esta implementação (Código fonte II.16) possui as seguintes características:

- configuração de execução: a partir do número T, *threads* por bloco, recebido na chamada da função, o processamento é particionado em um *grid* com N blocos e cada bloco contendo T *threads*, sendo cada *thread* responsável por processar um bloco do AES;
- memória para *table look-up*: usada a memória global do dispositivo;
- memória para chave de rodada: usada a memória global do dispositivo;
- laços (*loops*): removidos os laços na expansão da chave e nos processos de cifração/decifração.

4.5.3 cuda_aes_1b_nt_const_const

Esta implementação (Código fonte II.17) possui as seguintes características:

- configuração de execução: alocado um *grid* com somente um bloco, e cada bloco contendo T *threads*, sendo cada *thread* responsável por processar vários blocos do AES;
- memória para *table look-up*: usada a memória constante do dispositivo;
- memória para chave de rodada: usada a memória constante do dispositivo.

4.5.4 cuda_aes_nb_nt_const_const

Esta implementação (Código fonte II.18) possui as seguintes características:

- configuração de execução: a partir do número T, *threads* por bloco, recebido na chamada da função, o processamento é particionado em um *grid* com N blocos e cada bloco contendo T *threads*, sendo cada *thread* responsável por processar um bloco do AES;
- memória para *table look-up*: usada a memória constante do dispositivo;
- memória para chave de rodada: usada a memória constante do dispositivo.

4.5.5 cuda_aes_1b_nt_const_texture

Esta implementação (Código fonte II.19) possui as seguintes características:

- configuração de execução: alocado um *grid* com somente um bloco, e cada bloco contendo T *threads*, sendo cada *thread* responsável por processar vários blocos do AES;
- memória para *table look-up*: usada a memória constante do dispositivo;
- memória para chave de rodada: usada a memória de textura do dispositivo.

4.5.6 cuda_aes_nb_nt_const_texture

Esta implementação (Código fonte II.20) possui as seguintes características:

- configuração de execução: a partir do número T, *threads* por bloco, recebido na chamada da função, o processamento é particionado em um *grid* com N blocos e cada bloco contendo T *threads*, sendo cada *thread* responsável por processar um bloco do AES;
- memória para *table look-up*: usada a memória constante do dispositivo;
- memória para chave de rodada: usada a memória de textura do dispositivo.

4.5.7 cuda_aes_1b_nt_shared_const

Esta implementação (Código fonte II.21) possui as seguintes características:

- configuração de execução: alocado um *grid* com somente um bloco, e cada bloco contendo T *threads*, sendo cada *thread* responsável por processar vários blocos do AES;
- memória para *table look-up*: inicialmente usada a memória constante do dispositivo e posteriormente copiada para a memória compartilhada;
- memória para chave de rodada: usada a memória constante do dispositivo.

4.5.8 cuda_aes_nb_nt_shared_const

Esta implementação (Código fonte II.22) possui as seguintes características:

- configuração de execução: a partir do número T, *threads* por bloco, recebido na chamada da função, o processamento é particionado em um *grid* com N blocos e cada bloco contendo T *threads*, sendo cada *thread* responsável por processar um bloco do AES;
- memória para *table look-up*: inicialmente usada a memória constante do dispositivo e posteriormente copiada para a memória compartilhada;
- memória para chave de rodada: usada a memória constante do dispositivo.

4.5.9 cuda_aes_1b_nt_shared_texture

Esta implementação (Código fonte II.23) possui as seguintes características:

- configuração de execução: alocado um *grid* com somente um bloco, e cada bloco contendo T *threads*, sendo cada *thread* responsável por processar vários blocos do AES;
- memória para *table look-up*: inicialmente usada a memória constante do dispositivo e posteriormente copiada para a memória compartilhada;
- memória para chave de rodada: usada a memória de textura do dispositivo.

4.5.10 cuda_aes_nb_nt_shared_texture

Esta implementação (Código fonte II.24) possui as seguintes características:

- configuração de execução: a partir do número T, *threads* por bloco, recebido na chamada da função, o processamento é particionado em um *grid* com N blocos e cada bloco contendo T *threads*, sendo cada *thread* responsável por processar um bloco do AES;
- memória para *table look-up*: inicialmente usada a memória constante do dispositivo e posteriormente copiada para a memória compartilhada;
- memória para chave de rodada: usada a memória de textura do dispositivo.

Capítulo 5

Resultados Experimentais

Este capítulo elenca os passos realizados para a seleção da melhor implementação em GPU CUDA, exibindo os resultados intermediários e os pontos de decisão.

5.1 Introdução

Foram realizados uma série de experimentos visando determinar a implementação com maior ganho na relação tempo de cifração/decifração da CPU e da GPU. Alguns passos intermediários foram necessários, como a determinação de uma implementação com ou sem laços (*roll* ou *unroll*), escolha da configuração de chamada do *kernel*, seleção de quais memórias do dispositivo CUDA utilizar e os métodos de transferência de dados entre CPU e GPU.

5.2 Avaliação *roll* ou *unroll*

Neste experimento foram utilizadas as implementações descritas na Tabela 5.1 e os resultados estão apresentados nas Tabelas 5.2 e 5.3, para as máquinas dos Tópicos I.1 e I.2 respectivamente. Nas versões para GPU foram utilizadas várias chamadas com o valor de *threads* por bloco variando de 128 até 768, sendo que para cada tamanho de bloco foram realizadas 100 rodadas e o tempo médio foi auferido. Para cada uma das implementações o menor tempo médio por tamanho de chave está representado nas Tabelas 5.2 e 5.3. O tamanho do arquivo de entrada foi de 8MiB, usando o processo de cifração com chaves de 128, 192 e 256 *bits*.

Neste experimento pode ser observado o ganho na versão *unroll* comparado com a versão *roll*, tanto na implementação em CPU quanto para GPU. Este resultado determinou as implementações seguintes em GPU, onde todas utilizam uma implementação *unroll* do AES.

Pode ser observado, porém ainda em estágio inicial, o ganho na implementação em GPU quando comparado com a implementação em CPU. Este resultado somente está avaliando o tempo para a operação de cifração, sendo desconsiderado neste momento os tempos de cópia dos textos de entrada/saída entre *CPU* e *GPU*.

Tabela 5.1: Descrição das implementações utilizadas no experimento do Tópico 5.2

Implementação	**Dispositivo**	**Roll / Unroll**
reference (Código fonte II.13)	*CPU*	*roll*
reference_unroll (Código fonte II.14)	*CPU*	*unroll*
cuda_aes_nn_global_roll (Código fonte II.15)	*GPU*	*roll*
cuda_aes_nn_global_unroll (Código fonte II.16)	*GPU*	*unroll*

Tabela 5.2: Tempos de execução em milissegundos no experimento do Tópico 5.2 na máquina do Tópico I.1

Algoritmo	**Chave** (***bits***)	**Roll** (**ms**)	**Unroll** (**ms**)	***Relação*** *(Roll/Unroll)*
CPU	128	320,0029	318,6677	1,0042
GPU	128	16,7104	16,7019	1,0005
Relação *CPU/GPU*		19,1499	19,0797	
CPU	192	378,6409	371,1960	1,0201
GPU	192	19,9016	19,8982	1,0002
Relação *CPU/GPU*		19,0257	18,6548	
CPU	256	434,8714	426,7436	1,0190
GPU	256	23,1007	23,0861	1,0006
Relação *CPU/GPU*		18,8250	18,4849	

Tabela 5.3: Tempos de execução em milissegundos no experimento do Tópico 5.2 na máquina do Tópico I.2

Algoritmo	**Chave** (***bits***)	**Roll** (**ms**)	**Unroll** (**ms**)	***Relação*** *(Roll/Unroll)*
CPU	128	37,7775	33,5040	1,1276
GPU	128	1,8332	1,8272	1,0033
Relação *CPU/GPU*		20,6074	18,3363	
CPU	192	42,3018	41,0028	1,0317
GPU	192	2,1807	2,1739	1,0031
Relação *CPU/GPU*		19,3983	18,8614	
CPU	256	48,8614	46,0824	1,0603
GPU	256	2,5293	2,5221	1,0029
Relação *CPU/GPU*		19,3182	18,2714	

5.3 Avaliação 1 bloco ou N blocos

Neste experimento foram utilizadas as implementações descritas na Tabela 5.4 e os resultados estão apresentados nas Figuras 5.1 e 5.2 para a máquina do Tópico I.1 e nas Figuras 5.3 e 5.4 para a máquina do Tópico I.2.

Nas versões para GPU foram utilizadas várias chamadas com o valor de *threads* por bloco variando de 128 até 1024. Cada uma das implementações foi executada 100 vezes e o tempo médio foi auferido.

O tamanho do arquivo de entrada foi de 8MiB, usando o processo de cifração com chave de 128 *bits*.

A ideia deste experimento era verificar se o uso de 1 bloco com *threads* processando mais de um bloco do AES era mais vantajoso que N blocos com *threads* processando somente um bloco do AES, forçando que as *threads* na implementação de 1 bloco executassem mais cálculos e evitasse a troca de contexto dos blocos.

Pode ser observado nas Figuras 5.1 e 5.2 para a máquina do Tópico I.1 e nas Figuras 5.3 e 5.4 para a máquina do Tópico I.2 que as implementações de N blocos possuem tempos de cifração melhores que as suas versões de 1 bloco.

Tabela 5.4: Descrição das implementações utilizadas no experimento do Tópico 5.3

Implementação (Código Fonte)	**Abreviação**	**Blocos**	**Tabelas de *look-up***	**Chave de Cifração**
cuda_aes_1b_nt_const_const (Código fonte II.17)	1B CONST CONST	1	Constante	Constante
cuda_aes_1b_nt_const_texture (Código fonte II.19)	1B CONST TEXTURE	1	Constante	Textura
cuda_aes_1b_nt_shared_const (Código fonte II.21)	1B SHARED CONST	1	Compartilhada	Constante
cuda_aes_1b_nt_shared_texture (Código fonte II.23)	1B SHARED TEXTURE	1	Compartilhada	Textura
cuda_aes_nb_nt_const_const (Código fonte II.18)	NB CONST CONST	N	Constante	Constante
cuda_aes_nb_nt_const_texture (Código fonte II.20)	NB CONST TEXTURE	N	Constante	Textura
cuda_aes_nb_nt_shared_const (Código fonte II.22)	NB SHARED CONST	N	Compartilhada	Constante
cuda_aes_nb_nt_shared_texture (Código fonte II.24)	NB SHARED TEXTURE	N	Compartilhada	Textura

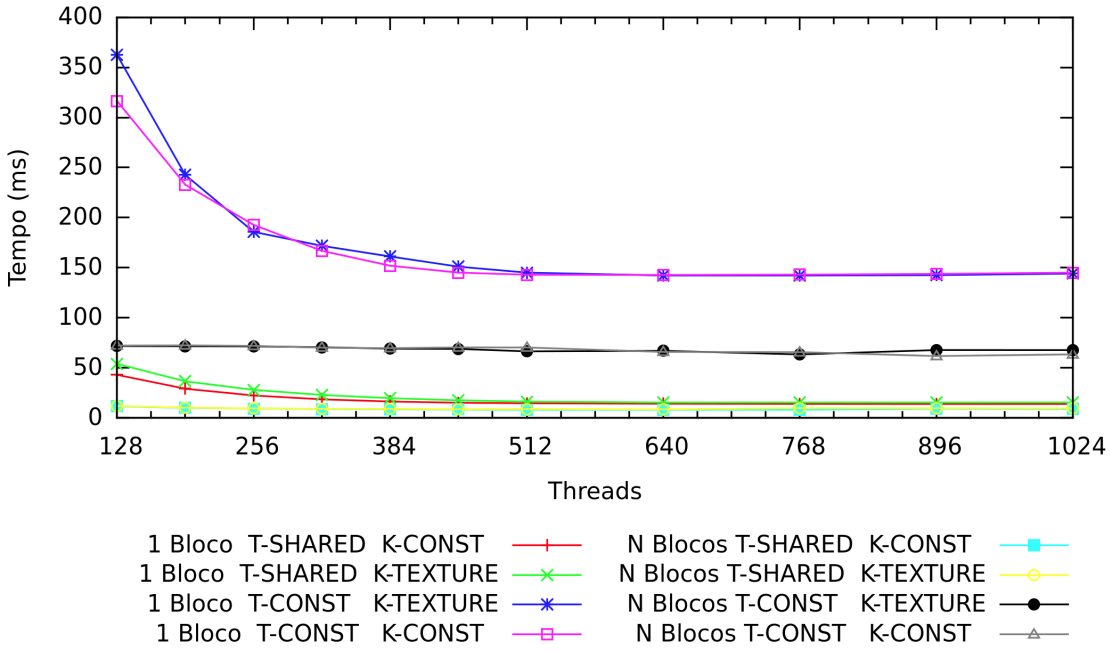
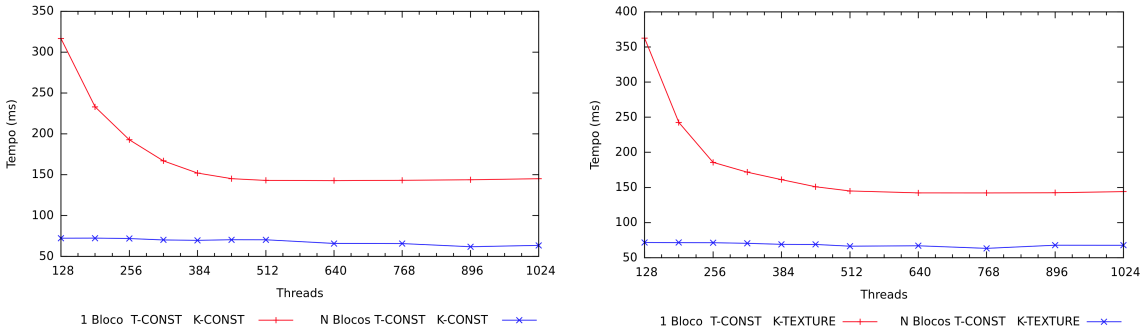
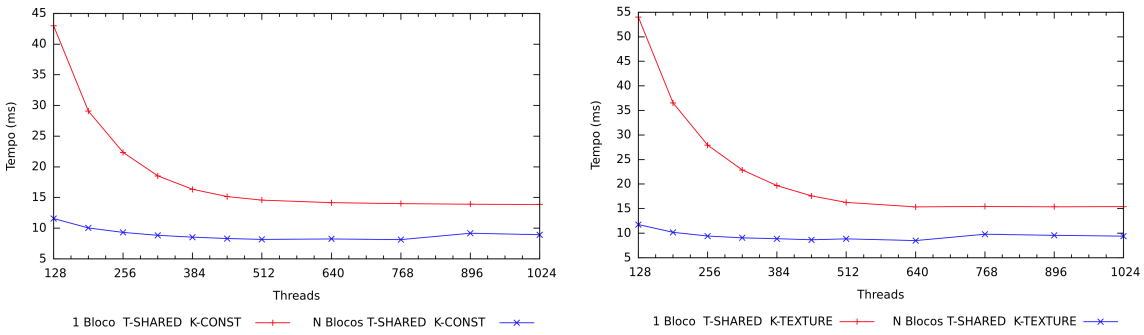


Figura 5.1: Tempos de execução em milissegundos no experimento do Tópico 5.3 para a máquina do Tópico I.1



(a) T-CONST K-CONST (b) T-CONST K-TEXTURE



(c) T-SHARED K-CONST (d) T-SHARE K-TEXTURE

Figura 5.2: Tempos de execução em milissegundos por implementação no experimento do Tópico 5.3 para a máquina do Tópico I.1

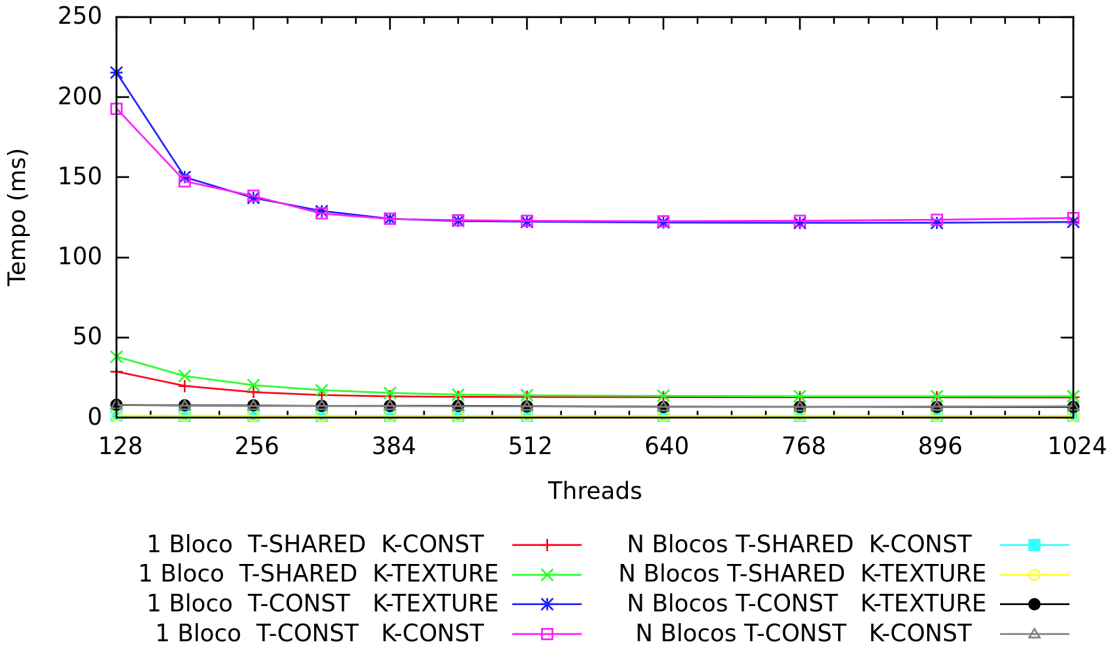
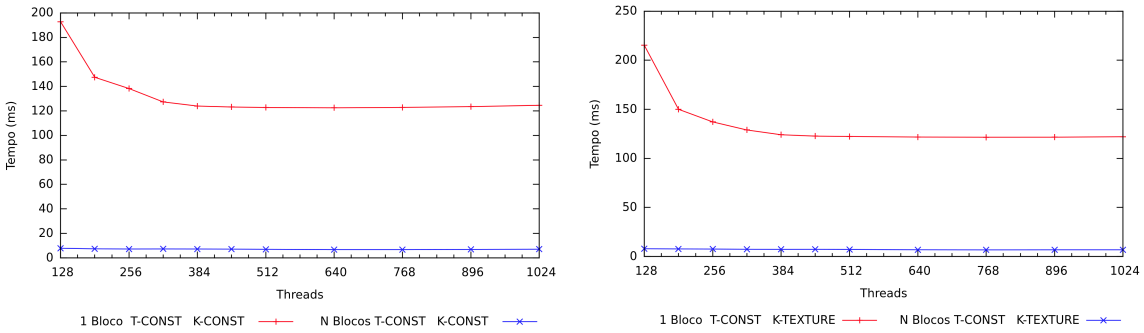
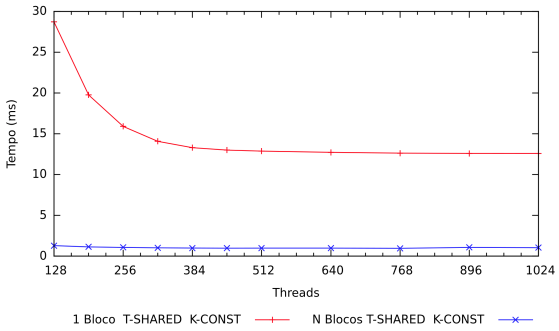


Figura 5.3: Tempos de execução em milissegundos no experimento do Tópico 5.3 para a máquina do Tópico I.2

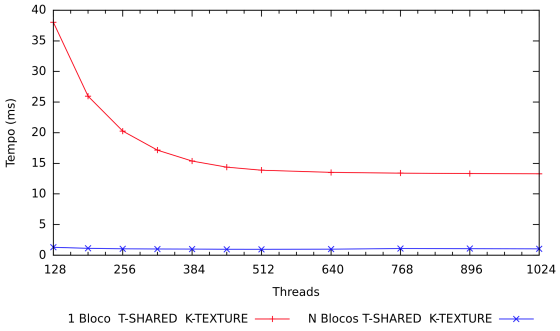


(a) T-CONST K-CONST

(b) T-CONST K-TEXTURE



(c) T-SHARED K-CONST



(d) T-SHARE K-TEXTURE

Figura 5.4: Tempos de execução em milissegundos por implementação no experimento do Tópico 5.3 para a máquina do Tópico I.2

5.4 Avaliação das implementações de N blocos

Neste experimento foram utilizadas as implementações descritas na Tabela 5.5 e os resultados estão apresentados nas Figura 5.5 e 5.6, para as máquinas dos Tópicos I.1 e I.2 respectivamente.

Nas versões para GPU foram utilizadas várias chamadas com o valor de *threads* por bloco variando de 128 até 1024. Cada uma das implementações foi executada 100 vezes e o tempo médio foi auferido.

O tamanho do arquivo de entrada foi de 8MiB, usando o processo de cifração com chave de 128 *bits*.

A ideia deste experimento era verificar as várias combinações no uso das memórias da GPU, colocando as tabelas de *look-up* nas memórias constante e compartilhada e as chaves de rodada nas memórias constante e de textura.

Pode ser observado nas Figuras 5.5 e 5.6 que as implementações de N blocos que usam a memória compartilhada para armazenar as tabelas de *look-up* vão diminuindo o seu tempo de cifração a medida que se aumenta o número de *threads* por bloco, valor esse esperado pois a memória compartilhada se torna mais efetiva quando atende a um número maior de *threads*.

As implementações que usam a memória compartilhada para armazenar as tabelas de *look-up* apresentaram o melhor tempo de execução, pode ser visto ainda uma pequena melhora usando

esta implementação quando as chaves de rodada são armazenadas na memória constante, conforme Figuras 5.7 e 5.8.

Cabe destacar ainda nas Figuras 5.7 e 5.8 uma grande oscilação nos tempos quando o número *threads* por bloco está entre 448 e 1024, isso ocorre pois o *kernel* estourou o espaço de registradores (memória mais rápida) disponível causando um armazenamento na memória global (memória mais lenta) do excedente.

Tabela 5.5: Descrição das implementações utilizadas no experimento do Tópico 5.4

Implementação (Código Fonte)	**Abreviação**	**Blocos**	**Tabelas de *look-up***	**Chave de Cifração**
cuda_aes_nb_nt_const_const (Código fonte II.18)	NB CONST CONST	N	Constante	Constante
cuda_aes_nb_nt_const_texture (Código fonte II.20)	NB CONST TEXTURE	N	Constante	Textura
cuda_aes_nb_nt_shared_const (Código fonte II.22)	NB SHARED CONST	N	Compartilhada	Constante
cuda_aes_nb_nt_shared_texture (Código fonte II.24)	NB SHARED TEXTURE	N	Compartilhada	Textura

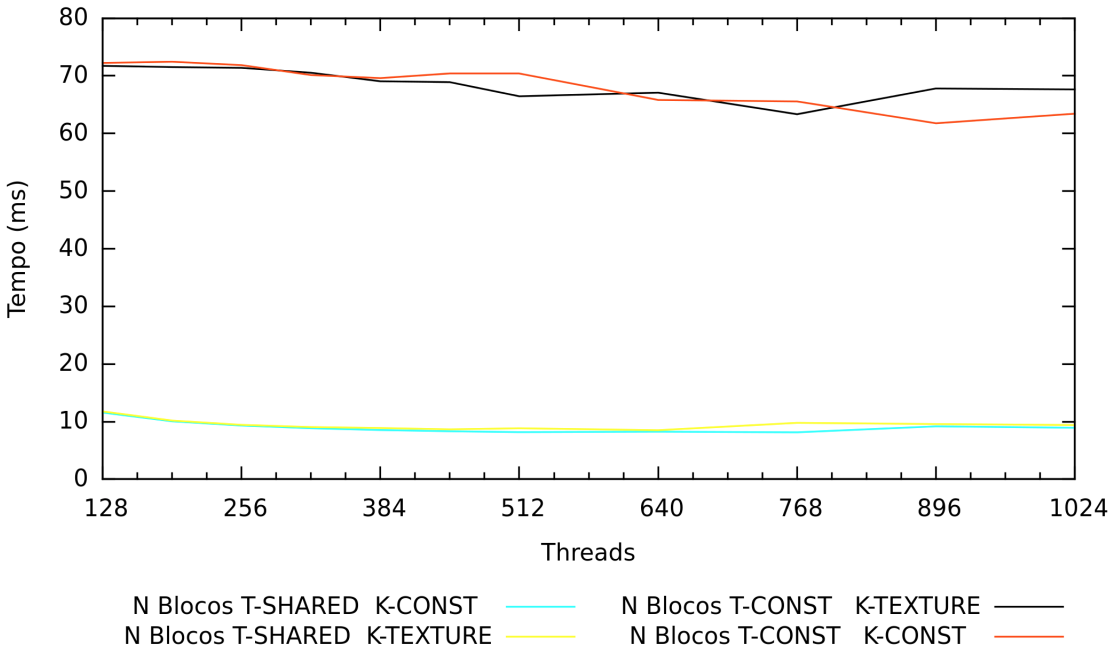


Figura 5.5: Tempos de execução em milissegundos no experimento do Tópico 5.4 para a máquina do Tópico I.1

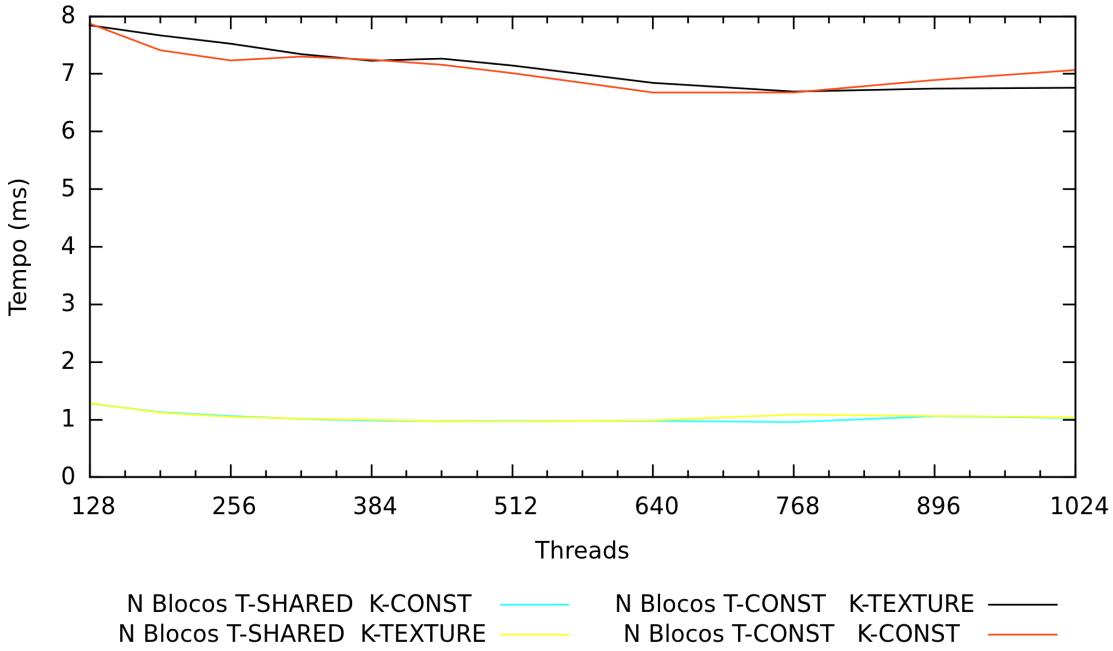


Figura 5.6: Tempos de execução em milissegundos no experimento do Tópico 5.4 para a máquina do Tópico I.2

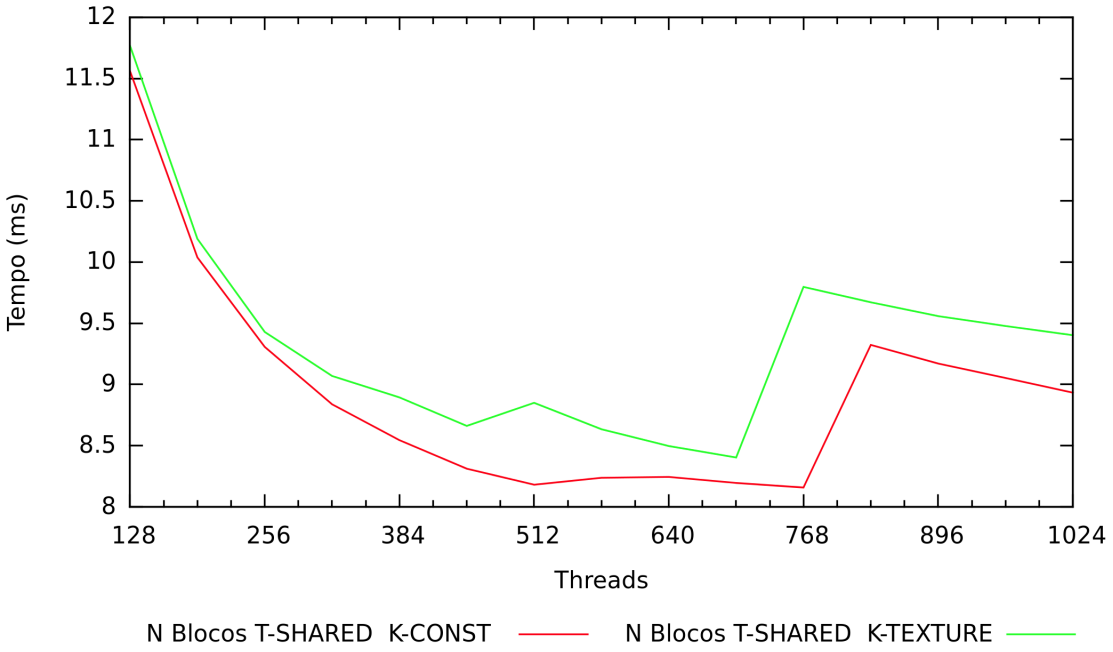


Figura 5.7: Tempos de execução em milissegundos no experimento do Tópico 5.4 para implementações com tabelas na memória compartilhada para a máquina do Tópico I.1

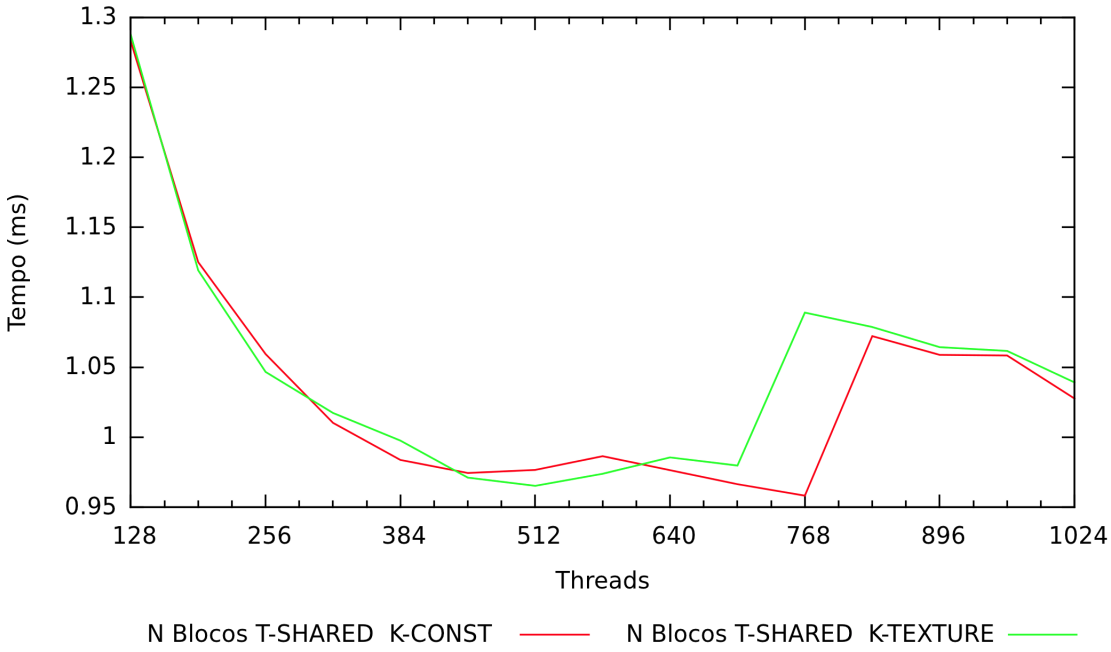


Figura 5.8: Tempos de execução em milissegundos no experimento do Tópico 5.4 para implementações com tabelas na memória compartilhada para a máquina do Tópico I.2

5.5 Avaliação das implementações de memória *pinned*

Neste experimento foram utilizadas as implementações descritas na Tabela 5.6, nas versões *pageable* e *pinned*, e os resultados estão apresentados nas Figuras 5.9, 5.10, 5.11 e 5.12.

Nas versões para GPU foram utilizadas várias chamadas com o valor de *threads* por bloco variando de 128 até 1024. Cada uma das implementações foi executada 100 vezes e os tempos médios de cifração foram representados nas Figuras 5.9 e 5.10, para as máquinas dos Tópicos I.1 e I.2 respectivamente. Posteriormente, os tempos totais médios de cifração, envolvendo a operação de cifração e as transferências entre CPU e GPU, são apresentados nas Figuras 5.11 e 5.12, para as máquinas dos Tópicos I.1 e I.2 respectivamente.

O tamanho do arquivo de entrada foi de 8MiB, usando o processo de cifração com chave de 128 *bits*.

A ideia deste experimento é fazer uso da memória *pinned* ou *page-locked* para o texto de entrada/saída nas implementações que obtiveram os melhores tempos no experimento do Tópico 5.4, ou seja, nas implementações com N blocos e tabelas de *look-up* na memória compartilhada.

Pode ser observado na Figura 5.9, dados relativos à máquina do Tópico I.1, que o tempo de cifração não variou entre as versões *pageable* e *pinned*. Nos dados relativos a máquina do Tópico I.2, representados na Figura 5.10, o tempo de cifração apresentou um pequena melhora na versão *pageable* para chaves de rodada na memória de textura e não variou entre as versões *pageable* e *pinned* para chaves de rodada na memória constante.

Com relação ao tempo total de cifração, representado nas Figuras 5.11 e 5.12, dados das

máquinas dos Tópicos I.1 e I.2 respectivamente, pode ser observado uma melhora significativa na versão *pinned* comparada com a versão *pageable*.

Destaca-se que o menor tempo de cifração ocorreu na implementação de N blocos, com tabelas de *look-up* na memória compartilhada e chaves de rodada na memória constante com 768 *threads* por bloco e o menor tempo de cifração total ocorreu na mesma implementação com o número de *threads* por bloco entre 384 e 768.

Tabela 5.6: Descrição das implementações *pageable* e *pinned* utilizadas no experimento do Tópico 5.5

Implementação (Código Fonte)	**Abreviação**	**Blocos**	**Tabelas de *look-up***	**Chave de Cifração**
cuda_aes_nb_nt_shared_const (Código fonte II.22)	NB SHARED CONST	N	Compartilhada	Constante
cuda_aes_nb_nt_shared_texture (Código fonte II.24)	NB SHARED TEXTURE	N	Compartilhada	Textura

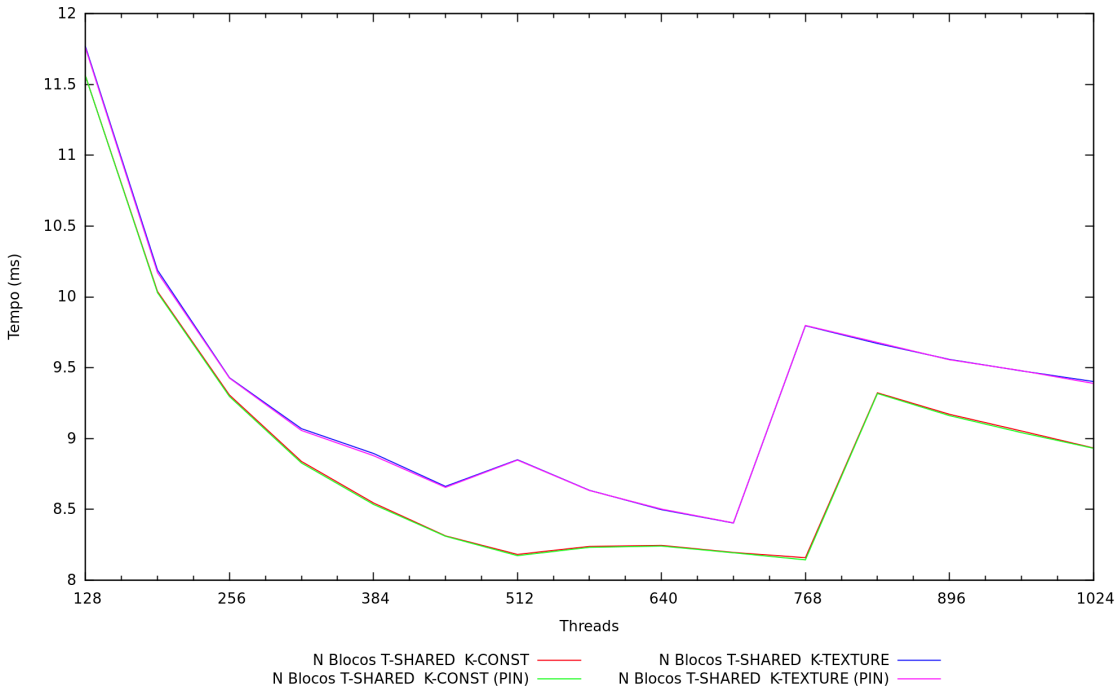


Figura 5.9: Tempos de execução em milissegundos do experimento do Tópico 5.5 para a máquina do Tópico I.1

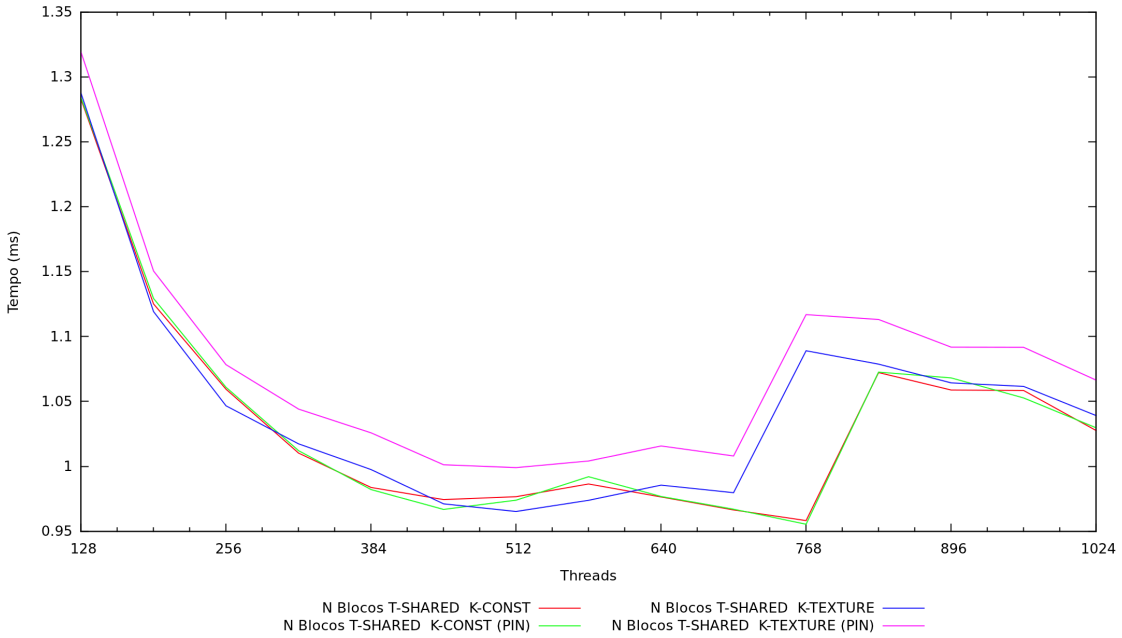


Figura 5.10: Tempos de execução em milissegundos do experimento do Tópico 5.5 para a máquina do Tópico I.2

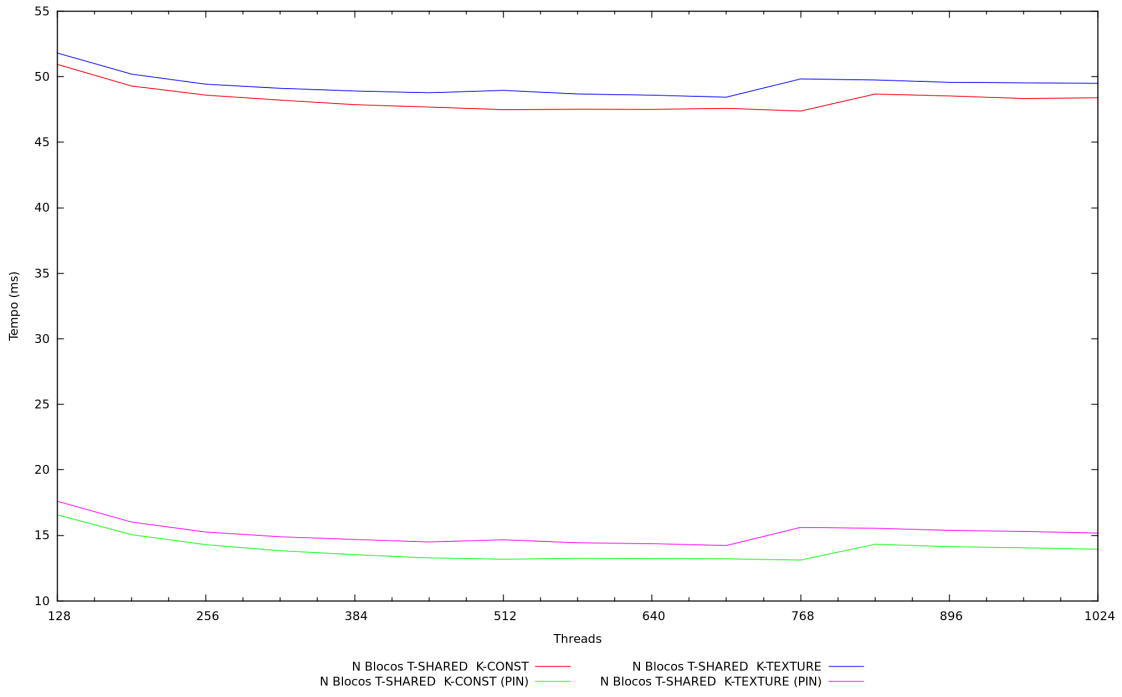


Figura 5.11: Tempos totais de execução em milissegundos do experimento do Tópico 5.5 para a máquina do Tópico I.1

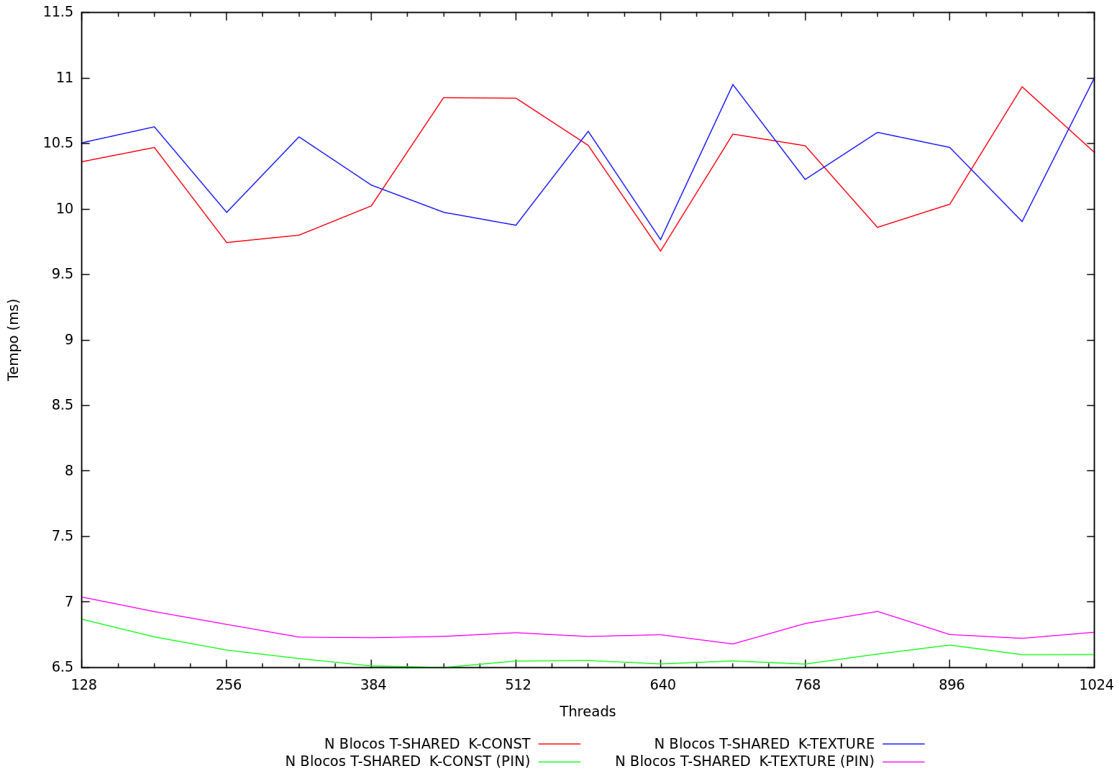


Figura 5.12: Tempos totais de execução em milissegundos do experimento do Tópico 5.5 para a máquina do Tópico I.2

5.6 Avaliação das implementações assíncronas

Neste experimento foi utilizada a implementação com N blocos, tabelas *look-up* na memória compartilhada e chave de cifração na memória constante e os resultados estão apresentados nas Figuras 5.13 e 5.14, para as máquinas dos Tópicos I.1 e I.2 respectivamente. Foram utilizadas várias chamadas com o valor de *threads* por bloco variando de 128 até 1024, tamanho de bloco enviado em cada *stream* variando de 32 até 1024 e total de *streams* variando de 2 até 4. Cada uma das implementações foi executada 10 vezes e o menor tempo médio total de cifração foi auferido.

O tamanho do arquivo de entrada foi de 8MB, usado o processo de cifração com chave de 128 *bits*.

A ideia deste experimento é fazer uso da memória *pinned* ou *page-locked* para o texto de entrada/saída, usando a melhor implementação selecionada do experimento do Tópico 5.5, porém usando transferência de dados assíncronas entre as memórias da CPU e GPU e fazendo uso de *streams*, intercalando transferência de dados e processamento na GPU.

Pode ser observado que o tempo total sofreu uma redução significativa comparado ao experimentos do Tópico 5.5. O tempo de cada operação de cifração não foi mensurado pois essa operação ocorre várias vezes para cada bloco de dados de entrada, porém espera-se que também não se altere como ocorreu no experimento do Tópico 5.5.

A melhor configuração na execução assíncrona deu-se para transferências de blocos contendo

128 blocos do AES, com 3 *streams* e 512 *threads* por bloco para a máquina do Tópico I.1 e com transferências de blocos contendo 32 blocos do AES, com 3 *streams* e 768 *threads* por bloco para a máquina do Tópico I.2, essas características dependem da placa gráfica disponível.

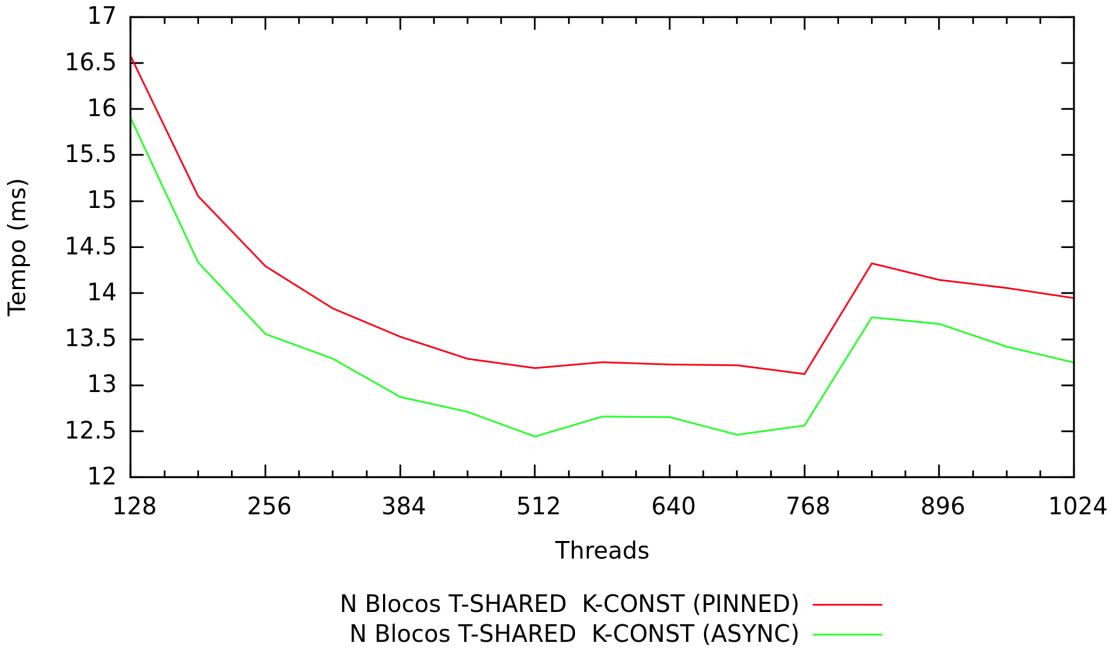


Figura 5.13: Tempos totais de execução em milissegundos do experimento do Tópico 5.6 para a máquina do Tópico I.1

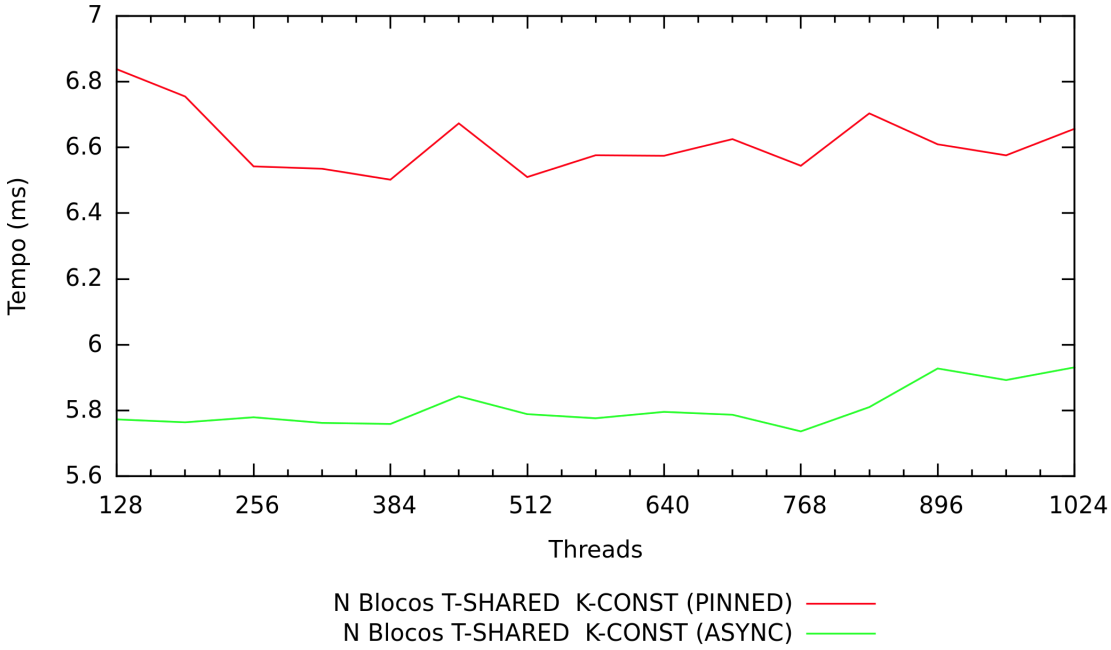


Figura 5.14: Tempos totais de execução em milissegundos do experimento do Tópico 5.6 para a máquina do Tópico I.2

5.7 Avaliação final

Na avaliação final foram determinados os tempos médios de cifração e os tempos totais médios de cifração para o AES com chaves de 128, 192 e 256 *bits*, para os tamanhos de arquivo de 1MiB, 4MiB, 8MiB, 16MiB, 64MiB, 128MiB e 256MiB, usando a melhor implementação para GPU. São apresentados também os desempenhos em Gbps para os dois tempos visando permitir comparação com outros trabalhos. Esses resultados são exibidos nas Tabelas 5.7, 5.8 e 5.9, para a máquina do Tópico I.1 e nas Tabelas 5.13, 5.14 e 5.15 para a máquina do Tópico I.2.

Cada configuração de teste foi rodada 100 vezes, usando tamanhos de blocos de 128 até 1024 *threads*, tamanho de bloco enviado em cada *stream* variando de 32 até 1024 e total de *streams* variando de 2 até 4, e selecionando o melhor tempo médio de cifração e cifração total por tamanho de arquivo e tamanho de chave do AES.

Para representar o ganho obtido na implementação em GPU, são apresentados nas Tabelas 5.10, 5.11 e 5.12 para a máquina do Tópico I.1 e nas Tabelas 5.16, 5.17 e 5.18 para a máquina do Tópico I.2 os tempos totais de cifração e o desempenho em Gbps para o AES com chaves de 128, 192 e 256 *bits*, para os tamanhos de arquivos de 1MiB, 4MiB, 8MiB, 16MiB, 64MiB, 128MiB e 256MiB, usando a melhor implementação para CPU, e a comparação entre os tempos totais de cifração da CPU e GPU [1].

Na comparação com a versão de referência para CPU e a melhor implementação para GPU, foram obtidos ganhos de no máximo 32,7 vezes mais na GPU na máquina do Tópico I.1 e de 8,7 vezes mais na GPU na máquina do Tópico I.2.

Tabela 5.7: Resultados do experimento do Tópico 5.7 da GPU para AES 128 *bits* na máquina do Tópico I.1

Tamanho Arquivo (MiB)	**Tempo de cifração (ms)**	**Desempenho da cifração (Gbps)**	**Tempo de cifração total (ms)**	**Desempenho cifração total (Gbps)**
1	1,1411	7,3516	3,2441	2,5858
4	4,1281	8,1283	7,1144	4,7164
8	8,1135	8,2713	12,4044	5,4101
16	16,0728	8,3506	22,9325	5,8527
64	63,7551	8,4208	86,0360	6,2401
128	127,3936	8,4285	169,9271	6,3188
256	254,5561	8,4362	338,1119	6,3514

[1] Valor da divisão do tempo de cifração total na CPU pelo valor do tempo de cifração total na GPU, indicando quantas vezes mais o primeiro é maior que o segundo.

Tabela 5.8: Resultados do experimento do Tópico 5.7 da GPU para AES 192 *bits* na máquina do Tópico I.1

Tamanho Arquivo (MiB)	**Tempo de cifração (ms)**	**Desempenho da cifração (Gbps)**	**Tempo de cifração total (ms)**	**Desempenho cifração total (Gbps)**
1	1,2934	6,4857	3,4136	2,4574
4	4,7512	7,0623	7,8257	4,2877
8	9,3533	7,1749	13,6902	4,9020
16	18,5526	7,2344	25,4940	5,2647
64	73,6770	7,2868	96,0417	5,5900
128	147,1834	7,2953	189,9174	5,6537
256	294,1711	7,3001	377,7917	5,6843

Tabela 5.9: Resultados do experimento do Tópico 5.7 da GPU para AES 256 *bits* na máquina do Tópico I.1

Tamanho Arquivo (MiB)	**Tempo de cifração (ms)**	**Desempenho da cifração (Gbps)**	**Tempo de cifração total (ms)**	**Desempenho cifração total (Gbps)**
1	1,4425	5,8152	3,6105	2,3234
4	5,3811	6,2356	8,4238	3,9833
8	10,5849	6,3400	14,9551	4,4873
16	21,0243	6,3839	27,8673	4,8163
64	83,5394	6,4266	105,7864	5,0750
128	166,9730	6,4306	209,6714	5,1211
256	333,7800	6,4338	416,7997	5,1523

Tabela 5.10: Resultados do experimento do Tópico 5.7 da CPU e comparação com a GPU para AES 128 *bits* na máquina do Tópico I.1

Tamanho Arquivo (MiB)	**Tempo de cifração (ms)**	**Desempenho da cifração (Gbps)**	**Comparação CPU/GPU (vezes)**
1	39,68066	0,2114	12,23
4	159,087021	0,2109	22,36
8	317,276794	0,2115	25,58
16	636,607361	0,2108	27,76
64	2544,4021	0,2110	29,57
128	5066,789551	0,2119	29,82
256	10134,1543	0,2119	29,97

Tabela 5.11: Resultados do experimento do Tópico 5.7 da CPU e comparação com a GPU para AES 192 *bits* na máquina do Tópico I.1

Tamanho Arquivo (MiB)	**Tempo de cifração (ms)**	**Desempenho da cifração (Gbps)**	**Comparação CPU/GPU (vezes)**
1	46,350266	0,1810	13,58
4	185,77211	0,1806	23,74
8	371,477783	0,1807	27,13
16	740,146301	0,1813	29,03
64	2967,556396	0,1809	30,90
128	5933,62207	0,1810	31,24
256	11863,09277	0,1810	31,40

Tabela 5.12: Resultados do experimento do Tópico 5.7 da CPU e comparação com a GPU para AES 256 *bits* na máquina do Tópico I.1

Tamanho Arquivo (MiB)	**Tempo de cifração (ms)**	**Desempenho da cifração (Gbps)**	**Comparação CPU/GPU (vezes)**
1	53,355797	0,1572	14,78
4	213,490753	0,1572	25,34
8	426,653473	0,1573	28,53
16	852,833496	0,1574	30,60
64	3410,360596	0,1574	32,24
128	6819,180176	0,1575	32,52
256	13629,34375	0,1576	32,70

Tabela 5.13: Resultados do experimento do Tópico 5.7 para AES 128 *bits* na máquina do Tópico I.2

Tamanho Arquivo (MiB)	**Tempo de cifração (ms)**	**Desempenho da cifração (Gbps)**	**Tempo de cifração total (ms)**	**Desempenho cifração total (Gbps)**
1	0,1443	58,1379	0,9376	8,9472
4	0,4980	67,3719	2,9639	11,3210
8	0,9508	70,5803	5,6802	11,8145
16	1,8657	71,9398	11,0442	12,1528
64	7,3372	73,1716	43,3835	12,3750
128	14,6340	73,3732	86,4958	12,4138
256	29,2372	73,4503	172,7090	12,4341

Tabela 5.14: Resultados do experimento do Tópico 5.7 para AES 192 *bits* na máquina do Tópico I.2

Tamanho Arquivo (MiB)	**Tempo de cifração (ms)**	**Desempenho da cifração (Gbps)**	**Tempo de cifração total (ms)**	**Desempenho cifração total (Gbps)**
1	0,1630	51,4714	0,9281	9,0382
4	0,5722	58,6386	2,9642	11,3198
8	1,0955	61,2575	5,6834	11,8079
16	2,1550	62,2818	11,0444	12,1525
64	8,4864	63,2623	43,3991	12,3706
128	16,9340	63,4074	86,5170	12,4108
256	33,8359	63,4676	172,7118	12,4339

Tabela 5.15: Resultados do experimento do Tópico 5.7 para AES 256 *bits* na máquina do Tópico I.2

Tamanho Arquivo (MiB)	**Tempo de cifração (ms)**	**Desempenho da cifração (Gbps)**	**Tempo de cifração total (ms)**	**Desempenho cifração total (Gbps)**
1	0,1804	46,4959	0,9381	8,9417
4	0,6433	52,1602	2,9646	11,3183
8	1,2385	54,1844	5,6769	11,8214
16	2,4396	55,0159	11,0472	12,1494
64	9,6179	55,8202	43,3996	12,3704
128	19,1981	55,9296	86,5218	12,4101
256	38,3564	55,9876	172,7584	12,4306

Tabela 5.16: Resultados do experimento do Tópico 5.7 da CPU e comparação com a GPU para AES 128 *bits* na máquina do Tópico I.2

Tamanho Arquivo (MiB)	**Tempo de cifração (ms)**	**Desempenho da cifração (Gbps)**	**Comparação CPU/GPU (vezes)**
1	4,0704	2,0609	4,34
4	22,9372	1,4629	7,74
8	33,4471	2,0064	5,89
16	64,9942	2,0651	5,88
64	258,5443	2,0765	5,96
128	516,4587	2,0790	5,97
256	1032,3069	2,0803	5,98

Tabela 5.17: Resultados do experimento do Tópico 5.7 da CPU e comparação com a GPU para AES 192 *bits* na máquina do Tópico I.2

Tamanho Arquivo (MiB)	**Tempo de cifração (ms)**	**Desempenho da cifração (Gbps)**	**Comparação CPU/GPU (vezes)**
1	4,7517	1,7654	5,12
4	21,4540	1,5640	7,24
8	40,5723	1,6541	7,14
16	75,9523	1,7671	6,88
64	303,0099	1,7718	6,98
128	605,3896	1,7736	7,00
256	1209,9767	1,7748	7,01

Tabela 5.18: Resultados do experimento do Tópico 5.7 da CPU e comparação com a GPU para AES 256 *bits* na máquina do Tópico I.2

Tamanho Arquivo (MiB)	**Tempo de cifração (ms)**	**Desempenho da cifração (Gbps)**	**Comparação CPU/GPU (vezes)**
1	5,4788	1,5311	5,84
4	23,8296	1,4081	8,04
8	44,8700	1,4956	7,90
16	87,6158	1,5319	7,93
64	348,8948	1,5388	8,04
128	697,2134	1,5400	8,06
256	1393,8160	1,5407	8,07

5.8 Comparação com trabalhos anteriores

Na Tabela 5.19 são apresentados os melhores tempos de cifração e cifração total para o AES 128 *bits* dos vários trabalhos referenciados com implementação do AES em CUDA e dos resultados obtidos neste trabalho. Devem ser levados em conta o *hardware* utilizado em cada um dos trabalhos, sendo que os principais fatores que impactam no desempenho são: o número de *cores*, o tamanho da interface de transferência com a memória e frequência de operação dos processadores.

Tabela 5.19: Comparação com trabalhos anteriores para AES 128 *bits*

Referência	**Modelo da GPU**	**Modo de Operação**	**Tamanho Arquivo (MiB)**	**Desempenho da cifração (Gbps)**	**Desempenho da cifração total (Gbps)**
Manavski (2007)	8800 GTX	ECB	8	8,8890	2,6840
Harrison e Waldron (2008)	8800 GTX	CTR	128	15,4230	6,9140
Biagio et al. (2009)	8800 GT	CTR	128	12,4120	-
Jang et al. (2011)	GTX 580	CBC-DEC	64	33,9000	10,0000
Jang et al. (2011)	GTX 295	ECB	64	32,8000	-
Máquina do Tópico I.1	GT 540M	ECB	8	8,2713	5,4101
Máquina do Tópico I.1	GT 540M	ECB	64	8,4208	6,2401
Máquina do Tópico I.1	GT 540M	ECB	128	8,4285	6,3188
Máquina do Tópico I.2	GTX 580	ECB	8	70,5803	11,8145
Máquina do Tópico I.2	GTX 580	ECB	64	73,1716	12,3750
Máquina do Tópico I.2	GTX 580	ECB	128	73,3732	12,4138

Capítulo 6

Conclusões

Neste trabalho foram apresentadas algumas características da plataforma CUDA, em especial, aquelas de interesse para o problema descrito na introdução, portar uma implementação em CPU do algoritmo AES para uma GPU CUDA.

O produto final deste trabalho são várias implementações do algoritmo AES para cifração e decifração, para tamanhos de chaves de 128, 192 e 256 *bits*, com o modo de operação ECB, e usando o esquema de *padding* proposto em Kaliski (2000).

Foi apresentado o caminho seguido até a seleção do melhor algoritmo que proveu ganhos de até 32,7 vezes sobre sua implementação em CPU. Também foram exibidos desempenhos equivalentes na cifração e superiores na cifração total comparados aos resultados de Manavski (2007) e desempenhos equivalentes na cifração total comparados aos resultados de Harrison e Waldron (2008), mesmo com um *hardware* bem inferior na máquina do Tópico I.1. No caso da máquina do Tópico I.2 todos os desempenhos foram superiores aos trabalhos anteriores, mesmo quando comparados ao trabalho de Jang et al. (2011), que utilizou o mesmo *hardware* na suas medições.

Os resultados alcançados estão limitados aos *hardwares* disponíveis para este trabalho, porém a metodologia aplicada aqui pode ser estendida a novos dispositivos CUDA, a medida que estes apareçam, como é o caso da plataforma *Kepler* anunciada em Maio deste ano, onde ganhos maiores poderão acontecer. (NVIDIA, 2012b)

Ganhos maiores são aguardados com o uso da tecnologia *PCI Express 3.0*[1], pois esta impactará no maior gargalo do uso das GPUs, a transferência de dados entre CPU e GPU.

Um questionamento pode ser levantado sobre as novas instruções destinadas ao AES providas nos novos processadores (AES-NI) e o porquê de implementar o mesmo em uma GPU CUDA respalda-se no fato que a migração para a GPU de cálculos como o algoritmo AES não limita-se a este somente, iniciativas de migração dos novos algoritmos propostos para o SHA-3 e até implementações de cifras assimétricas, por si só justificam o estudo e o uso desta plataforma neste e em outros trabalhos.

[1] *PCI-Express* (*Peripheral Component Interconnect Express*) é o padrão de soquetes criada para placas de expansão utilizadas em computadores pessoais para transmissão de dados.(WIKIPÉDIA, 2012c)

Como proposta de trabalho futuro fica o provimento de uma biblioteca criptográfica com algoritmos simétricos, assimétricos e funções de *hash* que contemplem o uso de processamento em dispositivos CUDA, quando estes estiverem disponíveis em um computador, ou sua versão em CPU quando não houver essa disponibilidade. Esse era o meu projeto ambicioso para este mestrado, porém por obstáculos encontrados, como a dificuldade no entendimento dos vários algoritmos em uma ambiente paralelo, a minha falta de experiência na arquitetura CUDA e seus aspectos arquiteturais e a premissa do tempo para conclusão deste trabalho, forçaram o direcionamento do escopo desta dissertação para o algoritmo AES, o que facilitou o estudo da arquitetura CUDA por já existirem outros trabalhos semelhantes e coube no tempo disponível para conclusão, contudo essa ideia ainda permanece válida e, na minha opinião, de grande interesse para a comunidade. Cito como exemplo a implementação em CUDA do *proxy* para o protocolo SSL em Jang et al. (2011), onde são feitas implementações dos algoritmos criptográficos usados neste protocolo na plataforma CUDA.

Trabalhos como Bernstein et al. (2012), também indicam um caminho para explorar mais a fundo a arquitetura CUDA, chegando ao nível de "instruções de máquina" para obter ganhos de desempenho.

E como proposta final de trabalho futuro fica a sugestão do uso de uma implementação que utilize múltiplas GPUs numa mesma máquina usando a tecnologia *Nvidia GPUDIRECT*. (NVIDIA, 2011b)

REFERÊNCIAS BIBLIOGRÁFICAS

BARRETO, P. S. L. M. **The AES Block Cipher, the GCM and EAX Authenticated Encryption with Associated Data Modes of Operation, and the CMAC (Originally OMAC) Message Authentication Code, in C++ and Java**. 2007. Disponível em: http://www.larc.usp.br/~pbarreto/AES++.zip. Acesso em: 20 jun. 2012.

BERNSTEIN, D. J. **Cache-timing Attacks on AES**. 2005. Disponível em: http://cr.yp.to/antiforgery/cachetiming-20050414.pdf.

BERNSTEIN, D. J. et al. Usable Assembly Language for GPUs: A Success Story. In: **Workshop records of Special-Purpose Hardware for Attacking Cryptographic Systems – SHARCS 2012**. [s.n.], 2012. p. 169–178. Disponível em: http://cryptojedi.org/papers/gpuasm-20120313.pdf. Acesso em: 20 jun. 2012.

BIAGIO, A. D. et al. **Design of a Parallel AES for Graphics Hardware Using the CUDA Framework**. 2009. 1–8 p. Disponível em: http://home.dei.polimi.it/barenghi/files/IPDPS2009.pdf. Acesso em: 20 jun. 2012.

BIHAM, E. A Fast New DES Implementation in Software. In: SPRINGER. **Fast Software Encryption**. [S.l.], 1997. p. 260–272.

BUCK, I. et al. Brook for GPUs: Stream Computing on Graphics Hardware. In: ACM. **ACM Transactions on Graphics (TOG)**. 2004. v. 23, n. 3, p. 777–786. Disponível em: http://dx.doi.org/10.1145/1015706.1015800. Acesso em: 20 jun. 2012.

DAEMEN, J.; RIJMEN, V. **AES proposal: Rijndael**. 1999. Disponível em: http://www.nic.funet.fi/index/crypt/cryptography/symmetric/aes/nist/Rijndael.pdf. Acesso em: 20 jun. 2012.

FLYNN, M. Very High-Speed Computing Systems. **Proceedings of the IEEE**, IEEE, v. 54, n. 12, p. 1901–1909, 1966. Disponível em: http://dx.doi.org/10.1109/PROC.1966.5273. Acesso em: 20 jun. 2012.

HARRIS, M. **Real-time Cloud Simulation and Rendering**. Tese (Doutorado) — University of North Carolina, 2003. Disponível em: http://www.markmark.net/dissertation/harrisDissertation.pdf. Acesso em: 20 jun. 2012.

HARRISON, O.; WALDRON, J. Practical Symmetric Key Cryptography on Modern Graphics Hardware. **USENIX Security Symposium**, p. 195–210, 2008. Disponível em: http://www.scss.tcd.ie/publications/tech-reports/reports.08/TCD-CS-2008-20.pdf. Acesso em: 20 jun. 2012.

INTEL. **Intel® Advanced Encryption Standard (AES) Instructions Set - Rev 3**. Janeiro 2010. Disponível em: http://software.intel.com/file/24917. Acesso em: 20 jun. 2012.

_____. **Intel® Advanced Encryption Standard Instructions (AES-NI)**. Outubro 2010. Disponível em: http://software.intel.com/en-us/articles/intel-advanced-encryption-standard-instructions-aes-ni/. Acesso em: 20 jun. 2012.

JANG, K. et al. SSLShader: Cheap SSL Acceleration with Commodity Processors. 2011. Disponível em: http://static.usenix.org/events/nsdi11/tech/full_papers/Jang.pdf. Acesso em: 20 jun. 2012.

KALISKI, B. **PKCS# 7**: Cryptographic Message Syntax Version 1.5. RFC 2315. [S.l.], 1998. Disponível em: http://tools.ietf.org/html/rfc2315.

_____. **PKCS# 5**: Password-Based Cryptography Specification Version 2.0. RFC 2898. [S.l.], 2000. Disponível em: http://tools.ietf.org/html/rfc2898.

KASPER, E.; SCHWABE, P. **Faster and Timing-Attack Resistant AES-GCM**. Junho 2009. Cryptology ePrint Archive, Report 2009/129. Disponível em: http://eprint.iacr.org/2009/129.pdf. Acesso em: 20 jun. 2012.

MANAVSKI, S. A. CUDA Compatible GPU as an Efficient Hardware Accelerator for AES Cryptography. **Proceedings of the IEEE International Conference on Signal Processing and Communication**, p. 65–68, 2007. Disponível em: http://www.manavski.com/downloads/PID505889.pdf. Acesso em: 20 jun. 2012.

MATSUI, M.; NAKAJIMA, J. **On the Power of Bitslice Implementation on Intel Core2 Processor**. 2007. Disponível em: http://dx.doi.org/10.1007/978-3-540-74735-2_9. Acesso em: 20 jun. 2012.

NIST. **FIPS 197**: Advanced Encryption Standard (AES). [S.l.], 2001. Disponível em: http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf. Acesso em: 20 jun. 2012.

_____. **SP 800-38A**: Recommendation for Block Cipher Modes of Operation. [S.l.], 2001. Disponível em: http://csrc.nist.gov/publications/nistpubs/800-38a/sp800-38a.pdf. Acesso em: 20 jun. 2012.

NVIDIA. **NVIDIA CUDA C Programming Guide Version 2.0**. [S.l.], Julho 2008. Disponível em: http://moss.csc.ncsu.edu/~mueller/cluster/nvidia/2.0/Programming_Guide_2.0beta2.pdf. Acesso em: 20 jun. 2012.

_____. **The CUDA Compiler Driver NVCC**. [S.l.], 2011. Disponível em: http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/nvcc.pdf. Acesso em: 20 jun. 2012.

_____. **NVIDIA GPUDIRECT**. 2011. Disponível em: http://developer.nvidia.com/cuda/nvidia-gpudirect. Acesso em: 20 jun. 2012.

_____. **NVIDIA's Next Generation CUDA Compute Architecture: Fermi V1.1**. [S.l.], 2011. Disponível em: http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf. Acesso em: 20 jun. 2012.

_____. **NVIDIA CUDA C Programming Guide Version 4.2**. [S.l.], Abril 2012. Disponível em: http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_C_Programming_Guide.pdf. Acesso em: 20 jun. 2012.

_____. **NVIDIA Kepler Compute Architecture**. 2012. Disponível em: http://www.nvidia.com/object/nvidia-kepler.html. Acesso em: 20 jun. 2012.

_____. **What is CUDA?** 2012. Disponível em: http://www.nvidia.com/object/cuda_home_new.html. Acesso em: 20 jun. 2012.

OSVIK, A. S. D. A.; TROMER, E. **Cache Attacks and Countermeasures: the Case of AES**. 2005. Disponível em: http://eprint.iacr.org/2005/271.pdf. Acesso em: 20 jun. 2012.

REBEIRO, D. S. C.; DEVI, A. **Bitslice Implementation of AES**. 2006. Disponível em: http://dx.doi.org/10.1007/11935070_14. Acesso em: 20 jun. 2012.

WIKIPÉDIA. **JIT**. 2012. Disponível em: http://pt.wikipedia.org/wiki/JIT. Acesso em: 20 jun. 2012.

_____. **OpenGL**. 2012. Disponível em: http://pt.wikipedia.org/wiki/OpenGL. Acesso em: 20 jun. 2012.

_____. **PCI Express**. 2012. Disponível em: http://en.wikipedia.org/wiki/PCI_Express. Acesso em: 20 jun. 2012.

_____. **Proxy Server**. 2012. Disponível em: http://en.wikipedia.org/wiki/Proxy_server. Acesso em: 20 jun. 2012.

ANEXOS

I PLATAFORMA DE TESTES

Este anexo descreve o ambiente de testes que foi utilizado para o desenvolvimento e execução dos programas descritos no Capítulo 4.

I.1 Máquina 1

I.1.1 *Software*

Segue descrição dos *softwares* utilizados:

- sistema operacional: Ubuntu 12.04 LTS *"Precise Pangolin"* 64 *bits*, disponível em http://www.ubuntu.com/download/desktop;
- GCC: versão 4.6.3;
- IDE: Eclipse CDT Indigo;
- CUDA *Toolkit*: versão 4.2.9, 64 *bits*, Ubuntu 11.04, disponível em http://developer.download.nvidia.com/compute/cuda/4_2/rel/toolkit/cudatoolkit_4.2.9_linux_64_ubuntu11.04.run;
- *Driver* CUDA: versão 295.41, 64 *bits*, disponível em http://developer.download.nvidia.com/compute/cuda/4_2/rel/drivers/devdriver_4.2_linux_64_295.41.run;
- CUDA SDK: versão 4.29, disponível em http://developer.download.nvidia.com/compute/cuda/4_2/rel/sdk/gpucomputingsdk_4.2.9_linux.run.

I.1.2 *Hardware*

Segue descrição dos *hardwares* utilizados:

- computador: *Notebook* LG A520-T.BE53P1(6000);
- processador: *Intel Core i7* - 2630QM (2.00GHz, *Cache* 6MB);
- memória: 6GB DDR3;
- Placa Gráfica: NVIDIA GeForce GT540M com 96 *cores*, 128 *bits* de interface de memória, 1344 MHz de frequência de operação, 2GB de memória dedicada e com tecnologia *Optimus*;

Segue descrição das capacidades da placa gráfica (Figura I.1), conforme saída do programa *getDeviceProperties* (Código Fonte II.6 e II.7)

I.2 Máquina 2

I.2.1 *Software*

Segue descrição dos *softwares* utilizados:

- sistema operacional: Ubuntu 11.04 *"Natty Narwhal"* 64 *bits*, disponível em http://releases.ubuntu.com/natty/;
- GCC: versão 4.5.2;
- CUDA *Toolkit*: versão 4.2.9, 64 *bits*, Ubuntu 11.04, disponível em http://developer.download.nvidia.com/compute/cuda/4_2/rel/toolkit/cudatoolkit_4.2.9_linux_64_ubuntu11.04.run;
- *Driver* CUDA: versão 295.41, 64 *bits*, disponível em http://developer.download.nvidia.com/compute/cuda/4_2/rel/drivers/devdriver_4.2_linux_64_295.41.run;
- CUDA SDK: versão 4.29, disponível em http://developer.download.nvidia.com/compute/cuda/4_2/rel/sdk/gpucomputingsdk_4.2.9_linux.run.

I.2.2 *Hardware*

Segue descrição dos *hardwares* utilizados:

- computador: *Desktop*;
- processador: *Intel Core i7* - 2600K (3.40GHz, *Cache* 8MB);
- memória: 8GB DDR3;
- Placa Gráfica: NVIDIA GeForce GTX580 com 512 *cores*, 384 *bits* de interface de memória e 1544 MHz de frequência de operação;

Segue descrição das capacidades da placa gráfica (Figura I.2), conforme saída do programa *getDeviceProperties* (Código Fonte II.6 e II.7)

```
name:                          GeForce GT 540M
compute capability:            2.1
clockRate:                     1344000
integrated:                    0
asyncEngineCount:              1
kernelExecTimeoutEnabled:      0
canMapHostMemory:              1
computeMode:                   0
concurrentKernels:             1

Memory information
totalGlobalMem:                2147155968
totalConstMem:                 65536
memPitch:                      2147483647
l2CacheSize:                   131072
textureAlignment:              512
maxTexture1D:                  65536
maxTexture2D:                  (65536, 65535)
maxTexture3D:                  (2048, 2048, 2048)
memoryBusWidth:                128 bits
memoryClockRate:               900000 KHz
unifiedAddressing:             1

MP information
multiProcessorCount:           2
sharedMemPerBlock:             49152
regsPerBlock:                  32768
warpSize:                      32
maxThreadsPerBlock:            1024
maxThreadsDim:                 (1024, 1024, 64)
maxGridSize:                   (65535, 65535 ,65535)
```

Figura I.1: Propriedades da placa gráfica 1

```
name:                          GeForce GTX 580
compute capability:            2.0
clockRate:                     1544000
integrated:                    0
asyncEngineCount:              1
kernelExecTimeoutEnabled:      0
canMapHostMemory:              1
computeMode:                   0
concurrentKernels:             1

Memory information
totalGlobalMem:                1610153984
totalConstMem:                 65536
memPitch:                      2147483647
l2CacheSize:                   786432
textureAlignment:              512
maxTexture1D:                  65536
maxTexture2D:                  (65536, 65535)
maxTexture3D:                  (2048, 2048, 2048)
memoryBusWidth:                384 bits
memoryClockRate:               2004000 KHz
unifiedAddressing:             1

MP information
multiProcessorCount:           16
sharedMemPerBlock:             49152
regsPerBlock:                  32768
warpSize:                      32
maxThreadsPerBlock:            1024
maxThreadsDim:                 (1024, 1024, 64)
maxGridSize:                   (65535, 65535 ,65535)
```

Figura I.2: Propriedades da placa gráfica 2

II.1 UTIL

Código fonte II.1: util/aes_key.c

```
1  #include "aes.h"
2  #include "util.h"
3
4  #include <time.h>
5  #include <sys/time.h>
6  #include <unistd.h>
7
8  #include <stdint.h>
9
10 #include <math.h>
11
12 int main(int argc, char **argv)
13 {
14   struct timespec start, stop;
15   float eventTime=0;
16
17   if (argc < 2) {
18     printf("USAGE: aes_key KEY\n");
19     return 1;
20   }
21   unint *ek;
22   unint *rk;
23
24   unint nk = stringToUnintArray(argv[1], &ek);
25
26   unint nr = 0;
27
28   switch (nk) {
29   case 4:
30     nr = 10;
31     break;
32   case 6:
33     nr = 12;
34     break;
35   case 8:
36     nr = 14;
37     break;
38   default:
39     printf("Invalid AES key size. %d\n", nk);
40     return 1;
41   }
42   printf("Encryption key\n");
43   rk = (unint *) malloc(4 * (nr + 1) * sizeof(unint));
44
45   eventRecord(&start);
46   AES_EncKey(ek,rk,nk);
47   eventRecord(&stop);
48
49   printHexArray(rk, 4 * (nr + 1));
50
```

```
51    eventElapsedTime(&eventTime,&start,&stop);
52
53    fprintf(stderr,"time(enc_key_%d):\t%f ms\n",nk*4*8,eventTime);
54
55    free(rk);
56    rk = NULL;
57
58    printf("\n\nDecryption key\n");
59    rk = (unint *) malloc(4 * (nr + 1) * sizeof(unint));
60
61    eventRecord(&start);
62    AES_DecKey(ek, rk,nk);
63    eventRecord(&stop);
64
65    printHexArray(rk, 4 * (nr + 1));
66
67    eventElapsedTime(&eventTime,&start,&stop);
68
69    fprintf(stderr,"time(dec_key_%d):\t%f ms\n",nk*4*8,eventTime);
70
71    free(rk);
72    rk = NULL;
73
74    free(ek);
75    ek = NULL;
76
77    return 0;
78  }
```

Código fonte II.2: util/aes_enc.c

```
1   #include "aes.h"
2   #include "util.h"
3
4   #include <time.h>
5   #include <sys/time.h>
6   #include <unistd.h>
7
8   #include <stdint.h>
9   #include <stdio.h>
10  #include <stdlib.h>
11  #include <string.h>
12
13  int main(int argc, char **argv)
14  {
15    struct timespec start, stop;
16    float eventTime;
17
18    if (argc < 3) {
19      printf("USAGE: aes_enc KEY PLAINTEXT\n");
20      return 1;
21    }
22
23    unint *ek, *rk;
24    unint *ct, *pt;
25    unint nk = stringToUnintArray(argv[1], &ek);
26    unint nb = stringToUnintArray(argv[2], &pt);
27
28    unint nr = 0;
29
```

```
30    switch (nk) {
31    case 4:
32      nr = 10;
33      break;
34    case 6:
35      nr = 12;
36      break;
37    case 8:
38      nr = 14;
39      break;
40    default:
41      printf("Invalid AES key size. %d\n", nk);
42      return 1;
43    }
44
45    if (nb != 4) {
46      printf("Invalid AES block size.\n");
47      return 1;
48    }
49
50    rk = (unint *) malloc(4 * (nr + 1) * sizeof(unint));
51
52    eventRecord(&start);
53    AES_EncKey(ek, rk, nk);
54    eventRecord(&stop);
55
56    eventElapsedTime(&eventTime,&start,&stop);
57
58    fprintf(stderr,"time(enc_key_%d):\t%f ms\n",nk*4*8,eventTime);
59
60    ct = (unint *) malloc(4 * sizeof(unint));
61
62    eventRecord(&start);
63    AES_Enc(pt, ct, rk, nk);
64    eventRecord(&stop);
65
66    eventElapsedTime(&eventTime,&start,&stop);
67
68    fprintf(stderr,"time(enc_%d):\t\t%f ms\n",nk*4*8,eventTime);
69
70    printHexArrayLine(ct, 4);
71
72    free(rk);
73    rk = NULL;
74
75    free(pt);
76    pt = NULL;
77
78    free(ct);
79    ct = NULL;
80    return 0;
81  }
```

Código fonte II.3: util/aes_dec.c

```
1  #include "aes.h"
2  #include "util.h"
3
4  #include <time.h>
5  #include <sys/time.h>
```

```
6  #include <unistd.h>
7
8  #include <stdint.h>
9  #include <stdio.h>
10 #include <stdlib.h>
11 #include <string.h>
12
13 int main(int argc, char **argv)
14 {
15   struct timespec start, stop;
16   float eventTime;
17
18   if (argc < 3) {
19     printf("USAGE: aes_dec KEY CYPHERTEXT\n");
20     return 1;
21   }
22
23   unint *dk, *rk;
24   unint *ct, *pt;
25   unint nk = stringToUnintArray(argv[1], &dk);
26   unint nb = stringToUnintArray(argv[2], &ct);
27
28   unint nr = 0;
29
30   switch (nk) {
31   case 4:
32     nr = 10;
33     break;
34   case 6:
35     nr = 12;
36     break;
37   case 8:
38     nr = 14;
39     break;
40   default:
41     printf("Invalid AES key size. %d\n", nk);
42     return 1;
43   }
44
45   if (nb != 4) {
46     printf("Invalid AES block size.\n");
47     return 1;
48   }
49
50   rk = (unint *) malloc(4 * (nr + 1) * sizeof(unint));
51
52   eventRecord(&start);
53   AES_DecKey(dk, rk, nk);
54   eventRecord(&stop);
55
56   eventElapsedTime(&eventTime,&start,&stop);
57
58   fprintf(stderr,"time(dec_key_%d):\t%f ms\n",nk*4*8,eventTime);
59
60
61   pt = (unint *) malloc(4 * sizeof(unint));
62
63   eventRecord(&start);
64   AES_Dec(ct, pt, rk, nk);
65   eventRecord(&stop);
```

```
66
67    eventElapsedTime(&eventTime,&start,&stop);
68
69    fprintf(stderr,"time(dec_%d):\t\t%f ms\n",nk*4*8,eventTime);
70
71    printHexArrayLine(pt, 4);
72
73    free(rk);
74    rk = NULL;
75
76    free(pt);
77    pt = NULL;
78
79    free(ct);
80    ct = NULL;
81    return 0;
82  }
```

Código fonte II.4: util/aes_enc_file.c

```
1   #include "aes.h"
2   #include "util.h"
3
4   #include <time.h>
5   #include <sys/time.h>
6   #include <sys/stat.h>
7   #include <unistd.h>
8
9   #include <stdint.h>
10  #include <stdio.h>
11  #include <stdlib.h>
12  #include <string.h>
13
14  #include "util.h"
15
16  int main(int argc, char **argv) {
17    unint i;
18    struct timespec start, stop;
19    float eventTime;
20
21    if (argc < 4) {
22      printf(
23          "USAGE: aes_enc_file KEY_FILE PLAINTEXT_FILE CYPHERTEXT_FILE\n");
24      return 1;
25    }
26    unint *ek, *rk;
27    unint *ct, *pt;
28
29    char* ek_file_name = argv[1];
30    char* pt_file_name = argv[2];
31    char* ct_file_name = argv[3];
32
33    FILE * ek_file;
34    FILE * pt_file;
35    FILE * ct_file;
36
37    struct stat ek_stat;
38    struct stat pt_stat;
39
40    // KEY FILE
```

```
41    ek_file = fopen(ek_file_name, "rb");
42    if (ek_file == NULL) {
43      fprintf(stderr, "KEY_FILE READ ERROR [%s]", ek_file_name);
44      return EXIT_FAILURE;
45    }
46
47    if (stat(ek_file_name, &ek_stat)) {
48      fclose(ek_file);
49      fprintf(stderr, "KEY_FILE STAT ERROR [%s]", ek_file_name);
50      return EXIT_FAILURE;
51    }
52
53    off_t ek_file_size = ek_stat.st_size;
54
55    unint nr = 0;
56    unint nk = 0;
57
58    switch (ek_file_size) {
59    case 16:
60      nk = 4;
61      nr = 10;
62      break;
63    case 24:
64      nk = 6;
65      nr = 12;
66      break;
67    case 32:
68      nk = 8;
69      nr = 14;
70      break;
71    default:
72      fprintf(stderr, "KEY_FILE Invalid AES key size [%lu bytes] [%s] \n",
73          ek_file_size, ek_file_name);
74      fclose(ek_file);
75      return EXIT_FAILURE;
76    }
77
78    if ((ek = (unint *) malloc(ek_file_size)) == NULL) {
79      fclose(ek_file);
80      fprintf(stderr, "KEY ERROR MALLOC\n");
81      return EXIT_FAILURE;
82    }
83
84    if (fread(ek, 1, ek_file_size, ek_file) < (unsigned) ek_file_size) {
85      fclose(ek_file);
86      fprintf(stderr, "KEY_FILE READING ERROR [%s] \n", ek_file_name);
87      return EXIT_FAILURE;
88    }
89
90    fclose(ek_file);
91
92    // PLAINTEX FILE
93
94    pt_file = fopen(pt_file_name, "rb");
95    if (pt_file == NULL) {
96      fprintf(stderr, "PLAINTEXT_FILE READ ERROR [%s]", pt_file_name);
97      return EXIT_FAILURE;
98    }
99
100   if (stat(pt_file_name, &pt_stat)) {
```

```
101     fclose(pt_file);
102     fprintf(stderr, "PLAINTEXT_FILE STAT ERROR [%s]", pt_file_name);
103     return EXIT_FAILURE;
104   }
105
106   off_t pt_file_size = pt_stat.st_size;
107
108   /// determinar padding
109   unint pt_file_size_mod16 = pt_file_size % 16;
110
111   char pt_size_pad = 16;
112   if (pt_file_size_mod16 != 0)
113     pt_size_pad = 16 - pt_file_size_mod16;
114
115   unint pt_size = pt_file_size + pt_size_pad;
116
117   if ((pt = (unint *) malloc(pt_size)) == NULL) {
118     fclose(pt_file);
119     fprintf(stderr, "PLAINTEXT ERROR MALLOC\n");
120     return EXIT_FAILURE;
121   }
122
123   if (fread(pt, 1, pt_file_size, pt_file) < (unsigned) pt_file_size) {
124     fclose(pt_file);
125     fprintf(stderr, "PLAINTEXT_FILE READING ERROR [%s] \n", pt_file_name);
126     return EXIT_FAILURE;
127   }
128
129   fclose(pt_file);
130   char * ptb = (char *) pt;
131   for (i = 0; i < (unsigned) pt_size_pad; ++i)
132     ptb[pt_file_size + i] = pt_size_pad;
133
134   ct_file = fopen(ct_file_name, "wb");
135   if (ct_file == NULL) {
136     fprintf(stderr, "CYPHERTEXT_FILE WRITE ERROR [%s]", ct_file_name);
137     return EXIT_FAILURE;
138   }
139
140 // Chave de round
141   rk = (unint *) malloc(4 * (nr + 1) * sizeof(unint));
142
143   AES_EncKey(ek, rk, nk);
144
145   ct = (unint *) malloc(pt_size);
146
147   eventRecord(&start);
148   AES_Enc(pt, ct, rk, nk,pt_size);
149   eventRecord(&stop);
150
151   eventElapsedTime(&eventTime,&start,&stop);
152
153   fprintf(stderr,"time(enc_%d):\t\t%f ms\t%u bytes\n",nk*4*8,eventTime,pt_size);
154
155   if (fwrite(ct, 1, pt_size, ct_file) < pt_size) {
156     fclose(ct_file);
157     fprintf(stderr, "CYPHERTEXT_FILE WRITING ERROR [%s]", ct_file_name);
158     return EXIT_FAILURE;
159
160   }
```

```
161
162   fclose(ct_file);
163
164   if (ek != NULL) {
165     free(ek);
166     ek = NULL;
167   }
168
169   if (rk != NULL) {
170     free(rk);
171     rk = NULL;
172   }
173
174   if (pt != NULL) {
175     free(pt);
176     pt = NULL;
177   }
178
179   if (ct != NULL) {
180     free(ct);
181     ct = NULL;
182   }
183
184   return 0;
185 }
```

Código fonte II.5: util/aes_dec_file.c

```
1  #include "aes.h"
2  #include "util.h"
3
4  #include <time.h>
5  #include <sys/time.h>
6  #include <sys/stat.h>
7  #include <unistd.h>
8
9  #include <stdint.h>
10 #include <stdio.h>
11 #include <stdlib.h>
12 #include <string.h>
13
14 #include "util.h"
15
16 int main(int argc, char **argv) {
17   struct timespec start, stop;
18   float eventTime;
19
20   if (argc < 4) {
21     printf("USAGE: aes_dec_file KEY_FILE CYPHERTEXT_FILE PLAINTEXT_FILE\n");
22     return 1;
23   }
24   unint *ek, *rk;
25   unint *ct, *pt;
26
27   char* ek_file_name = argv[1];
28   char* ct_file_name = argv[2];
29   char* pt_file_name = argv[3];
30
31   FILE * ek_file;
32   FILE * pt_file;
```

```
33    FILE * ct_file;
34
35    struct stat ek_stat;
36    struct stat ct_stat;
37
38    // KEY FILE
39    ek_file = fopen(ek_file_name, "rb");
40    if (ek_file == NULL) {
41      fprintf(stderr, "KEY_FILE READ ERROR [%s]", ek_file_name);
42      return EXIT_FAILURE;
43    }
44
45    if (stat(ek_file_name, &ek_stat)) {
46      fclose(ek_file);
47      fprintf(stderr, "KEY_FILE STAT ERROR [%s]", ek_file_name);
48      return EXIT_FAILURE;
49    }
50
51    off_t ek_file_size = ek_stat.st_size;
52
53    unint nr = 0;
54    unint nk = 0;
55
56    switch (ek_file_size) {
57    case 16:
58      nk = 4;
59      nr = 10;
60      break;
61    case 24:
62      nk = 6;
63      nr = 12;
64      break;
65    case 32:
66      nk = 8;
67      nr = 14;
68      break;
69    default:
70      fprintf(stderr, "KEY_FILE Invalid AES key size [%lu bytes] [%s] \n",
71          ek_file_size, ek_file_name);
72      fclose(ek_file);
73      return EXIT_FAILURE;
74    }
75
76    if ((ek = (unint *) malloc(ek_file_size)) == NULL) {
77      fclose(ek_file);
78      fprintf(stderr, "KEY ERROR MALLOC\n");
79      return EXIT_FAILURE;
80    }
81
82    if (fread(ek, 1, ek_file_size, ek_file) < (unsigned) ek_file_size) {
83      fclose(ek_file);
84      fprintf(stderr, "KEY_FILE READING ERROR [%s] \n", ek_file_name);
85      return EXIT_FAILURE;
86    }
87
88    fclose(ek_file);
89
90    // CYPHERTEXT FILE
91
92    ct_file = fopen(ct_file_name, "rb");
```

```
93    if (ct_file == NULL) {
94      fprintf(stderr, "CYPHERTEXT_FILE READ ERROR [%s]", ct_file_name);
95      return EXIT_FAILURE;
96    }
97
98    if (stat(ct_file_name, &ct_stat)) {
99      fclose(ct_file);
100     fprintf(stderr, "CYPHERTEXT_FILE STAT ERROR [%s]", ct_file_name);
101     return EXIT_FAILURE;
102   }
103
104   off_t ct_file_size = ct_stat.st_size;
105
106   /// determinar padding correto
107   unint ct_file_size_mod16 = ct_file_size % 16;
108
109   if (ct_file_size_mod16 != 0) {
110     fclose(ct_file);
111     fprintf(stderr, "CYPHERTEXT_FILE SIZE PADDING ERROR [%s]",
112         ct_file_name);
113     return EXIT_FAILURE;
114   }
115
116   if ((ct = (unint *) malloc(ct_file_size)) == NULL) {
117     fclose(ct_file);
118     fprintf(stderr, "CYPHERTEXT ERROR MALLOC\n");
119     return EXIT_FAILURE;
120   }
121
122   if (fread(ct, 1, ct_file_size, ct_file) < (unsigned)ct_file_size) {
123     fclose(ct_file);
124     fprintf(stderr, "CYPHERTEXT_FILE READING ERROR [%s] \n", ct_file_name);
125     return EXIT_FAILURE;
126   }
127
128   fclose(ct_file);
129
130   /// PLAINTEXT FILE
131   pt_file = fopen(pt_file_name, "wb");
132   if (pt_file == NULL) {
133     fprintf(stderr, "PLAINTEXT_FILE WRITE ERROR [%s]", pt_file_name);
134     return EXIT_FAILURE;
135   }
136
137   /// Chave de round
138   rk = (unint *) malloc(4 * (nr + 1) * sizeof(unint));
139
140
141   AES_DecKey(ek, rk, nk);
142
143   pt = (unint *) malloc(ct_file_size);
144
145   eventRecord(&start);
146   AES_Dec(ct, pt, rk, nk, ct_file_size);
147   eventRecord(&stop);
148
149   eventElapsedTime(&eventTime, &start, &stop);
150
151   fprintf(stderr, "time(dec_%d):\t\t%f ms\t%lu bytes\n", nk * 4 * 8,
        eventTime,ct_file_size);
```

```
152
153   char * ptb = (char *) pt;
154   char pad = ptb[ct_file_size - 1];
155
156   if (fwrite(pt, 1, ct_file_size - pad, pt_file) < (ct_file_size - pad)) {
157     fclose(pt_file);
158     fprintf(stderr, "PLAINTEXT_FILE WRITING ERROR [%s]", pt_file_name);
159     return EXIT_FAILURE;
160   }
161
162   fclose(pt_file);
163
164   if (ek != NULL) {
165     free(ek);
166     ek = NULL;
167   }
168
169   if (rk != NULL) {
170     free(rk);
171     rk = NULL;
172   }
173
174   if (pt != NULL) {
175     free(pt);
176     pt = NULL;
177   }
178
179   if (ct != NULL) {
180     free(ct);
181     ct = NULL;
182   }
183
184   return 0;
185 }
```

II.2 CUDA_UTIL

Código fonte II.6: cuda_util/getDeviceProperties.cu

```
1 #include "cuda_util.h"
2 #include <stdio.h>
3 #include <stdlib.h>
4
5 int main(int argc, char **argv)
6 {
7   printDeviceProperties();
8   return EXIT_SUCCESS;
9 }
```

Código fonte II.7: cuda_util/cuda_util.cu

```
1 #include "cuda_util.h"
2 #include <stdio.h>
3 #include <stdlib.h>
4
5 int printDeviceProperties()
6 {
```

```
7
8   cudaDeviceProp prop;
9
10  int count = 0;
11
12  HANDLE_ERROR(cudaGetDeviceCount(&count));
13
14  if (count == 0) {
15    printf("No CUDA device founded\n");
16    return EXIT_FAILURE;
17  }
18
19  for (int i = 0; i < count; i++) {
20    cudaGetDeviceProperties(&prop, i);
21    printf("Device %d\n\n", i);
22    printf("name:                          %s\n", prop.name);
23    printf("compute capability:            %d.%d\n", prop.major,
24           prop.minor);
25    printf("clockRate:                     %d\n", prop.clockRate);
26    printf("integrated:                    %d\n", prop.integrated);
27    printf("asyncEngineCount:              %d\n",
28           prop.asyncEngineCount);
29    printf("kernelExecTimeoutEnabled:   %d\n",
30           prop.kernelExecTimeoutEnabled);
31    printf("canMapHostMemory:              %d\n",
32           prop.canMapHostMemory);
33    printf("computeMode:                   %d\n", prop.computeMode);
34    printf("concurrentKernels:             %d\n",
35           prop.concurrentKernels);
36
37    printf("\nMemory information\n");
38    printf("totalGlobalMem:                %ld\n",
39           prop.totalGlobalMem);
40    printf("totalConstMem:                 %ld\n", prop.totalConstMem);
41    printf("memPitch:                      %ld\n", prop.memPitch);
42    printf("l2CacheSize:                   %d\n", prop.l2CacheSize);
43
44    printf("textureAlignment:              %ld\n",
45           prop.textureAlignment);
46    printf("maxTexture1D:                  %d\n", prop.maxTexture1D);
47    printf("maxTexture2D:                  (%d, %d)\n",
48           prop.maxTexture2D[0], prop.maxTexture2D[1]);
49    printf("maxTexture3D:                  (%d, %d, %d)\n",
50           prop.maxTexture3D[0],
51           prop.maxTexture3D[1], prop.maxTexture3D[2]);
52    printf("memoryBusWidth:                %d bits\n",
53           prop.memoryBusWidth);
54    printf("memoryClockRate:               %d KHz\n",
55           prop.memoryClockRate);
56    printf("unifiedAddressing:             %d\n",
57           prop.unifiedAddressing);
58
59    printf("\nMP information\n");
60    printf("multiProcessorCount:           %d\n",
61           prop.multiProcessorCount);
62    printf("sharedMemPerBlock:             %ld\n",
63           prop.sharedMemPerBlock);
64    printf("regsPerBlock:                  %d\n", prop.regsPerBlock);
65    printf("warpSize:                      %d\n", prop.warpSize);
66    printf("maxThreadsPerBlock:            %d\n",
```

```
67              prop.maxThreadsPerBlock);
68      printf("maxThreadsDim:                 (%d, %d, %d)\n",
69              prop.maxThreadsDim[0], prop.maxThreadsDim[1],
70              prop.maxThreadsDim[2]);
71      printf("maxGridSize:                   (%d, %d ,%d)\n\n",
72              prop.maxGridSize[0], prop.maxGridSize[1],
73              prop.maxGridSize[2]);
74    }
75
76    return EXIT_SUCCESS;
77  }
78
79  void HandleError(cudaError_t err, const char *file, int line)
80  {
81    if (err != cudaSuccess) {
82      printf("%s in %s at line %d\n", cudaGetErrorString(err),
83              file, line);
84      exit(EXIT_FAILURE);
85    }
86    return;
87  }
88
89      //Create thread
90  CUTThread start_thread(CUT_THREADROUTINE func, void *data)
91  {
92    pthread_t thread;
93    pthread_create(&thread, NULL, func, data);
94    return thread;
95  }
96
97      //Wait for thread to finish
98  void end_thread(CUTThread thread)
99  {
100   pthread_join(thread, NULL);
101 }
102
103     //Destroy thread
104 void destroy_thread(CUTThread thread)
105 {
106   pthread_cancel(thread);
107 }
108
109     //Wait for multiple threads
110 void wait_for_threads(const CUTThread * threads, int num)
111 {
112   for (int i = 0; i < num; i++)
113     end_thread(threads[i]);
114 }
```

Código fonte II.8: cuda_util/aes_key.cu

```
1  #include "aes.h"
2  #include "util.h"
3
4  #include <time.h>
5  #include <sys/time.h>
6  #include <unistd.h>
7
8  #include <stdint.h>
9
```

```
10 #include <math.h>
11
12 int main(int argc, char **argv)
13 {
14   if (argc < 2) {
15     printf("USAGE: aes_key KEY\n");
16     return 1;
17   }
18   unint *ek;
19   unint *rk;
20
21   unint nk = stringToUnintArray(argv[1], &ek);
22
23   unint nr = 0;
24
25   switch (nk) {
26   case 4:
27     nr = 10;
28     break;
29   case 6:
30     nr = 12;
31     break;
32   case 8:
33     nr = 14;
34     break;
35   default:
36     printf("Invalid AES key size. %d\n", nk);
37     return 1;
38   }
39   printf("Encryption key\n");
40   rk = (unint *) malloc(4 * (nr + 1) * sizeof(unint));
41
42   AES_EncKey(ek,rk,nk);
43
44   printHexArray(rk, 4 * (nr + 1));
45
46   free(rk);
47   rk = NULL;
48
49   printf("\n\nDecryption key\n");
50   rk = (unint *) malloc(4 * (nr + 1) * sizeof(unint));
51
52   AES_DecKey(ek, rk,nk);
53
54   printHexArray(rk, 4 * (nr + 1));
55
56   free(rk);
57   rk = NULL;
58
59   free(ek);
60   ek = NULL;
61
62   return 0;
63 }
```

Código fonte II.9: cuda_util/aes_enc.cu

```
1 #include "aes.h"
2 #include "util.h"
3
```

```c
#include <time.h>
#include <sys/time.h>
#include <unistd.h>

#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(int argc, char **argv)
{
  if (argc < 3) {
    printf("USAGE: aes_enc KEY PLAINTEXT\n");
    return 1;
  }

  unint *ek, *rk;
  unint *ct, *pt;
  unint nk = stringToUnintArray(argv[1], &ek);
  unint nb = stringToUnintArray(argv[2], &pt);

  unint nr = 0;

  switch (nk) {
  case 4:
    nr = 10;
    break;
  case 6:
    nr = 12;
    break;
  case 8:
    nr = 14;
    break;
  default:
    printf("Invalid AES key size. %d\n", nk);
    return 1;
  }

  if (nb != 4) {
    printf("Invalid AES block size.\n");
    return 1;
  }

  rk = (unint *) malloc(4 * (nr + 1) * sizeof(unint));

  AES_EncKey(ek, rk, nk);

  ct = (unint *) malloc(4 * sizeof(unint));

  AES_Enc(pt, ct, rk, nk);

  printHexArrayLine(ct, 4);

  free(rk);
  rk = NULL;

  free(pt);
  pt = NULL;

  free(ct);
```

```
64    ct = NULL;
65    return 0;
66  }
```

Código fonte II.10: cuda_util/aes_dec.cu

```
1   #include "aes.h"
2   #include "util.h"
3
4   #include <time.h>
5   #include <sys/time.h>
6   #include <unistd.h>
7
8   #include <stdint.h>
9   #include <stdio.h>
10  #include <stdlib.h>
11  #include <string.h>
12
13  int main(int argc, char **argv)
14  {
15
16    if (argc < 3) {
17      printf("USAGE: aes_dec KEY CYPHERTEXT\n");
18      return 1;
19    }
20
21    uint *dk, *rk;
22    uint *ct, *pt;
23    uint nk = stringToUintArray(argv[1], &dk);
24    uint nb = stringToUintArray(argv[2], &ct);
25
26    uint nr = 0;
27
28    switch (nk) {
29    case 4:
30      nr = 10;
31      break;
32    case 6:
33      nr = 12;
34      break;
35    case 8:
36      nr = 14;
37      break;
38    default:
39      printf("Invalid AES key size. %d\n", nk);
40      return 1;
41    }
42
43    if (nb != 4) {
44      printf("Invalid AES block size.\n");
45      return 1;
46    }
47
48    rk = (uint *) malloc(4 * (nr + 1) * sizeof(uint));
49
50    AES_DecKey(dk, rk, nk);
51
52    pt = (uint *) malloc(4 * sizeof(uint));
53
54    AES_Dec(ct, pt, rk, nk);
```

```
55
56    printHexArrayLine(pt, 4);
57
58    free(rk);
59    rk = NULL;
60
61    free(pt);
62    pt = NULL;
63
64    free(ct);
65    ct = NULL;
66    return 0;
67  }
```

Código fonte II.11: cuda_util/aes_enc_file.cu

```
1  #include "aes.h"
2  #include "util.h"
3
4  #include <time.h>
5  #include <sys/time.h>
6  #include <sys/stat.h>
7  #include <unistd.h>
8
9  #include <stdint.h>
10 #include <stdio.h>
11 #include <stdlib.h>
12 #include <string.h>
13
14 #include <cuda.h>
15 #include <cuda_runtime.h>
16 #include <device_launch_parameters.h>
17
18 #include "cuda_util.h"
19 #include "util.h"
20
21 int main(int argc, char **argv) {
22   unint i;
23   cudaDeviceProp prop;
24   HANDLE_ERROR( cudaGetDeviceProperties( &prop,0));
25 #ifdef ASYNC
26   if (argc < 5) {
27     printf(
28         "USAGE: aes_enc_file KEY_FILE PLAINTEXT_FILE CYPHERTEXT_FILE THREADS_PER_BLOCK
               BLOCKS STREAMS\n");
29     return 1;
30   }
31   unint BLOCOS = atoi(argv[5]);
32   unint STREAMS = atoi(argv[6]);
33
34   if (!prop.deviceOverlap) {
35     printf("Overlap nao disponivel. Nao e possivel rodar ASYNC\n");
36     return EXIT_FAILURE;
37   }
38 #else
39   if (argc < 3) {
40     printf(
41         "USAGE: aes_enc_file KEY_FILE PLAINTEXT_FILE CYPHERTEXT_FILE THREADS_PER_BLOCK\n");
42     return 1;
43   }
```

```
44 #endif
45
46 #ifdef ZEROCOPY
47   if (prop.canMapHostMemory != 1) {
48     printf("Memoria mapeada nao disponivel. Nao e possivel rodar ZEROCOPY\n");
49     return EXIT_FAILURE;
50   }
51   HANDLE_ERROR( cudaSetDeviceFlags( cudaDeviceMapHost ) );
52 #endif
53
54   unint *ek, *rk;
55   unint *ct, *pt;
56
57   char* ek_file_name = argv[1];
58   char* pt_file_name = argv[2];
59   char* ct_file_name = argv[3];
60   unint threads_per_block = atoi(argv[4]);
61
62 //  fprintf(stderr,"KEY_FILE:              %s\n",ek_file_name);
63 //  fprintf(stderr,"PLAINTEXT_FILE:        %s\n",pt_file_name);
64 //  fprintf(stderr,"CYPHERTEXT_FILE:       %s\n",ct_file_name);
65 //  fprintf(stderr,"THREADS_PER_BLOCK:     %d\n\n",threads_per_block);
66
67   FILE * ek_file;
68   FILE * pt_file;
69   FILE * ct_file;
70
71   struct stat ek_stat;
72   struct stat pt_stat;
73
74   // KEY FILE
75   ek_file = fopen(ek_file_name, "rb");
76   if (ek_file == NULL) {
77     fprintf(stderr, "KEY_FILE READ ERROR [%s]", ek_file_name);
78     return EXIT_FAILURE;
79   }
80
81   if (stat(ek_file_name, &ek_stat)) {
82     fclose(ek_file);
83     fprintf(stderr, "KEY_FILE STAT ERROR [%s]", ek_file_name);
84     return EXIT_FAILURE;
85   }
86
87   off_t ek_file_size = ek_stat.st_size;
88
89   unint nr = 0;
90   unint nk = 0;
91
92   switch (ek_file_size) {
93   case 16:
94     nk = 4;
95     nr = 10;
96     break;
97   case 24:
98     nk = 6;
99     nr = 12;
100    break;
101  case 32:
102    nk = 8;
103    nr = 14;
```

```
104       break;
105     default:
106       fprintf(stderr, "KEY_FILE Invalid AES key size [%lu bytes] [%s] \n",
107           ek_file_size, ek_file_name);
108       fclose(ek_file);
109       return EXIT_FAILURE;
110     }
111
112     if ((ek = (unint *) malloc(ek_file_size)) == NULL) {
113       fclose(ek_file);
114       fprintf(stderr, "KEY ERROR MALLOC\n");
115       return EXIT_FAILURE;
116     }
117
118     if (fread(ek, 1, ek_file_size, ek_file) < (unsigned) ek_file_size) {
119       fclose(ek_file);
120       fprintf(stderr, "KEY_FILE READING ERROR [%s] \n", ek_file_name);
121       return EXIT_FAILURE;
122     }
123
124     fclose(ek_file);
125
126     // PLAINTEX FILE
127
128     pt_file = fopen(pt_file_name, "rb");
129     if (pt_file == NULL) {
130       fprintf(stderr, "PLAINTEXT_FILE READ ERROR [%s]", pt_file_name);
131       return EXIT_FAILURE;
132     }
133
134     if (stat(pt_file_name, &pt_stat)) {
135       fclose(pt_file);
136       fprintf(stderr, "PLAINTEXT_FILE STAT ERROR [%s]", pt_file_name);
137       return EXIT_FAILURE;
138     }
139
140     off_t pt_file_size = pt_stat.st_size;
141
142     /// determinar padding
143     unint pt_file_size_mod16 = pt_file_size % 16;
144
145     unsigned char pt_size_pad = 16;
146     if (pt_file_size_mod16 != 0)
147       pt_size_pad = 16 - pt_file_size_mod16;
148
149     unint pt_size = pt_file_size + pt_size_pad;
150 #ifdef ZEROCOPY
151     HANDLE_ERROR(cudaHostAlloc((void **)&pt,
          pt_size,cudaHostAllocWriteCombined|cudaHostAllocMapped));
152 #elif PINNED
153     HANDLE_ERROR(cudaHostAlloc((void **)&pt, pt_size,cudaHostAllocDefault));
154 #else
155     if ((pt = (unint *) malloc(pt_size)) == NULL) {
156       fclose(pt_file);
157       fprintf(stderr, "PLAINTEXT ERROR MALLOC\n");
158       return EXIT_FAILURE;
159     }
160 #endif
161
162     if (fread(pt, 1, pt_file_size, pt_file) < (unsigned) pt_file_size) {
```

```
163       fclose(pt_file);
164       fprintf(stderr, "PLAINTEXT_FILE READING ERROR [%s] \n", pt_file_name);
165       return EXIT_FAILURE;
166     }
167
168     fclose(pt_file);
169     char * ptb = (char *) pt;
170     for (i = 0; i < pt_size_pad; ++i)
171       ptb[pt_file_size + i] = pt_size_pad;
172
173     ct_file = fopen(ct_file_name, "wb");
174     if (ct_file == NULL) {
175       fprintf(stderr, "CYPHERTEXT_FILE WRITE ERROR [%s]", ct_file_name);
176       return EXIT_FAILURE;
177     }
178
179   // Chave de round
180   #if defined PINNED || defined ZEROCOPY
181     HANDLE_ERROR(cudaHostAlloc((void **)&rk, 4 * (nr + 1) *
          sizeof(unint),cudaHostAllocDefault));
182   #else
183     rk = (unint *) malloc(4 * (nr + 1) * sizeof(unint));
184   #endif
185
186     AES_EncKey(ek, rk, nk);
187
188   #ifdef ZEROCOPY
189     HANDLE_ERROR(cudaHostAlloc((void **)&ct, pt_size,cudaHostAllocMapped));
190   #elif PINNED
191     HANDLE_ERROR(cudaHostAlloc((void **)&ct, pt_size,cudaHostAllocDefault));
192   #else
193     ct = (unint *) malloc(pt_size);
194   #endif
195
196   #ifdef ZEROCOPY
197     AES_Enc_ZeroCopy(pt, ct, rk, nk, pt_size, threads_per_block);
198   #elif ASYNC
199     AES_Enc_Async(pt, ct, rk, nk, pt_size, threads_per_block,BLOCOS,STREAMS);
200   #else
201     AES_Enc(pt, ct, rk, nk, pt_size, threads_per_block);
202   #endif
203
204     if (fwrite(ct, 1, pt_size, ct_file) < pt_size) {
205       fclose(ct_file);
206       fprintf(stderr, "CYPHERTEXT_FILE WRITING ERROR [%s]", ct_file_name);
207       return EXIT_FAILURE;
208
209     }
210
211     fclose(ct_file);
212
213     if (ek != NULL) {
214       free(ek);
215       ek = NULL;
216     }
217
218   #if defined PINNED || defined ZEROCOPY
219     if (rk != NULL) {
220       HANDLE_ERROR(cudaFreeHost(rk));
221       rk = NULL;
```

```
222     }
223
224     if (pt != NULL) {
225       HANDLE_ERROR(cudaFreeHost(pt));
226       pt = NULL;
227     }
228
229     if (ct != NULL) {
230       HANDLE_ERROR(cudaFreeHost(ct));
231       ct = NULL;
232     }
233
234 #else
235     if (rk != NULL) {
236       free(rk);
237       rk = NULL;
238     }
239
240     if (pt != NULL) {
241       free(pt);
242       pt = NULL;
243     }
244
245     if (ct != NULL) {
246       free(ct);
247       ct = NULL;
248     }
249 #endif
250
251     return 0;
252 }
```

Código fonte II.12: cuda_util/aes_dec_file.cu

```
1  #include "aes.h"
2  #include "util.h"
3
4  #include <time.h>
5  #include <sys/time.h>
6  #include <sys/stat.h>
7  #include <unistd.h>
8
9  #include <stdint.h>
10 #include <stdio.h>
11 #include <stdlib.h>
12 #include <string.h>
13
14 #include <cuda.h>
15 #include <cuda_runtime.h>
16 #include <device_launch_parameters.h>
17
18 #include "cuda_util.h"
19 #include "util.h"
20
21 int main(int argc, char **argv) {
22
23 #ifdef ASYNC
24     if (argc < 5) {
25       printf(
```

```
26          "USAGE: aes_dec_file KEY_FILE PLAINTEXT_FILE CYPHERTEXT_FILE THREADS_PER_BLOCK
                BLOCKS STREAMS\n");
27      return 1;
28    }
29    unint BLOCOS = atoi(argv[5]);
30    unint STREAMS = atoi(argv[6]);
31  #else
32    if (argc < 3) {
33      printf(
34          "USAGE: aes_dec_file KEY_FILE CYPHERTEXT_FILE PLAINTEXT_FILE THREADS_PER_BLOCK\n");
35      return 1;
36    }
37  #endif
38
39    unint *ek, *rk;
40    unint *ct, *pt;
41
42    char* ek_file_name = argv[1];
43    char* ct_file_name = argv[2];
44    char* pt_file_name = argv[3];
45    unint threads_per_block = atoi(argv[4]);
46
47    FILE * ek_file;
48    FILE * pt_file;
49    FILE * ct_file;
50
51    struct stat ek_stat;
52    struct stat ct_stat;
53
54    // KEY FILE
55    ek_file = fopen(ek_file_name, "rb");
56    if (ek_file == NULL) {
57      fprintf(stderr, "KEY_FILE READ ERROR [%s]", ek_file_name);
58      return EXIT_FAILURE;
59    }
60
61    if (stat(ek_file_name, &ek_stat)) {
62      fclose(ek_file);
63      fprintf(stderr, "KEY_FILE STAT ERROR [%s]", ek_file_name);
64      return EXIT_FAILURE;
65    }
66
67    off_t ek_file_size = ek_stat.st_size;
68
69    unint nr = 0;
70    unint nk = 0;
71
72    switch (ek_file_size) {
73    case 16:
74      nk = 4;
75      nr = 10;
76      break;
77    case 24:
78      nk = 6;
79      nr = 12;
80      break;
81    case 32:
82      nk = 8;
83      nr = 14;
84      break;
```

```
85    default:
86      fprintf(stderr, "KEY_FILE Invalid AES key size [%lu bytes] [%s] \n",
87          ek_file_size, ek_file_name);
88      fclose(ek_file);
89      return EXIT_FAILURE;
90    }
91
92    if ((ek = (unint *) malloc(ek_file_size)) == NULL) {
93      fclose(ek_file);
94      fprintf(stderr, "KEY ERROR MALLOC\n");
95      return EXIT_FAILURE;
96    }
97
98    if (fread(ek, 1, ek_file_size, ek_file) < (unsigned) ek_file_size) {
99      fclose(ek_file);
100     fprintf(stderr, "KEY_FILE READING ERROR [%s] \n", ek_file_name);
101     return EXIT_FAILURE;
102   }
103
104   fclose(ek_file);
105
106   // CYPHERTEXT FILE
107
108   ct_file = fopen(ct_file_name, "rb");
109   if (ct_file == NULL) {
110     fprintf(stderr, "CYPHERTEXT_FILE READ ERROR [%s]", ct_file_name);
111     return EXIT_FAILURE;
112   }
113
114   if (stat(ct_file_name, &ct_stat)) {
115     fclose(ct_file);
116     fprintf(stderr, "CYPHERTEXT_FILE STAT ERROR [%s]", ct_file_name);
117     return EXIT_FAILURE;
118   }
119
120   off_t ct_file_size = ct_stat.st_size;
121
122   /// determinar padding correto
123   unint ct_file_size_mod16 = ct_file_size % 16;
124
125   if (ct_file_size_mod16 != 0) {
126     fclose(ct_file);
127     fprintf(stderr, "CYPHERTEXT_FILE SIZE PADDING ERROR [%s]",
128         ct_file_name);
129     return EXIT_FAILURE;
130   }
131
132 #ifdef PINNED
133   HANDLE_ERROR(cudaHostAlloc((void **)&ct, ct_file_size,cudaHostAllocDefault));
134 #else
135   if ((ct = (unint *) malloc(ct_file_size)) == NULL) {
136     fclose(ct_file);
137     fprintf(stderr, "CYPHERTEXT ERROR MALLOC\n");
138     return EXIT_FAILURE;
139   }
140 #endif
141
142   if (fread(ct, 1, ct_file_size, ct_file) < (unsigned) ct_file_size) {
143     fclose(ct_file);
144     fprintf(stderr, "CYPHERTEXT_FILE READING ERROR [%s] \n", ct_file_name);
```

```
145       return EXIT_FAILURE;
146     }
147
148     fclose(ct_file);
149
150     /// PLAINTEXT FILE
151     pt_file = fopen(pt_file_name, "wb");
152     if (pt_file == NULL) {
153       fprintf(stderr, "PLAINTEXT_FILE WRITE ERROR [%s]", pt_file_name);
154       return EXIT_FAILURE;
155     }
156
157     // Chave de round
158 #ifdef PINNED
159     HANDLE_ERROR(cudaHostAlloc((void **)&rk, 4 * (nr + 1) *
          sizeof(unint),cudaHostAllocDefault));
160 #else
161     rk = (unint *) malloc(4 * (nr + 1) * sizeof(unint));
162 #endif
163
164     AES_DecKey(ek, rk, nk);
165
166 #ifdef PINNED
167     HANDLE_ERROR(cudaHostAlloc((void **)&pt, ct_file_size,cudaHostAllocDefault));
168 #else
169     pt = (unint *) malloc(ct_file_size);
170 #endif
171
172 #ifdef ASYNC
173     AES_Dec_Async(ct, pt, rk, nk, ct_file_size, threads_per_block,BLOCOS,STREAMS);
174 #else
175     AES_Dec(ct, pt, rk, nk, ct_file_size, threads_per_block);
176 #endif
177
178
179
180     char * ptb = (char *) pt;
181     char pad = ptb[ct_file_size - 1];
182
183     if (fwrite(pt, 1, ct_file_size - pad, pt_file) < (ct_file_size - pad)) {
184       fclose(pt_file);
185       fprintf(stderr, "PLAINTEXT_FILE WRITING ERROR [%s]", pt_file_name);
186       return EXIT_FAILURE;
187     }
188
189     fclose(pt_file);
190
191     if (ek != NULL) {
192       free(ek);
193       ek = NULL;
194     }
195
196 #ifdef PINNED
197     if (rk != NULL) {
198       HANDLE_ERROR(cudaFreeHost(rk));
199       rk = NULL;
200     }
201
202     if (pt != NULL) {
203       HANDLE_ERROR(cudaFreeHost(pt));
```

```
204     pt = NULL;
205   }
206
207   if (ct != NULL) {
208     HANDLE_ERROR(cudaFreeHost(ct));
209     ct = NULL;
210   }
211
212 #else
213   if (rk != NULL) {
214     free(rk);
215     rk = NULL;
216   }
217
218   if (pt != NULL) {
219     free(pt);
220     pt = NULL;
221   }
222
223   if (ct != NULL) {
224     free(ct);
225     ct = NULL;
226   }
227 #endif
228
229   return 0;
230 }
```

II.3 REFERENCE

Código fonte II.13: reference/aes.c

```
1  void AES_EncKey128(const unint * ck, unint * rk)
2  {
3    unint i = 0;
4    unint temp;
5
6    rk[0] = ck[0];
7    rk[1] = ck[1];
8    rk[2] = ck[2];
9    rk[3] = ck[3];
10
11   for (;;) {
12     temp = rk[3];
13     rk[4] = rk[0] ^
14         (Te4[(temp >> 16) & 0xff] & 0xff000000) ^
15         (Te4[(temp >> 8) & 0xff] & 0x00ff0000) ^
16         (Te4[(temp) & 0xff] & 0x0000ff00) ^
17         (Te4[(temp >> 24)] & 0x000000ff) ^ rcon[i];
18     rk[5] = rk[1] ^ rk[4];
19     rk[6] = rk[2] ^ rk[5];
20     rk[7] = rk[3] ^ rk[6];
21     if (++i == 10)
22       return;
23     rk += 4;
24   }
25
```

```
}
void AES_Enc128(unint * pt, unint * ct, unint * rek)
{
  unint s0, s1, s2, s3, t0, t1, t2, t3;

  s0 = pt[0] ^ rek[0];
  s1 = pt[1] ^ rek[1];
  s2 = pt[2] ^ rek[2];
  s3 = pt[3] ^ rek[3];
  uint r = 5;
  for (;;) {
    t0 = Te0[(s0 >> 24)] ^
        Te1[(s1 >> 16) & 0xff] ^
        Te2[(s2 >> 8) & 0xff] ^ Te3[(s3) & 0xff] ^ rek[4];
    t1 = Te0[(s1 >> 24)] ^
        Te1[(s2 >> 16) & 0xff] ^
        Te2[(s3 >> 8) & 0xff] ^ Te3[(s0) & 0xff] ^ rek[5];
    t2 = Te0[(s2 >> 24)] ^
        Te1[(s3 >> 16) & 0xff] ^
        Te2[(s0 >> 8) & 0xff] ^ Te3[(s1) & 0xff] ^ rek[6];
    t3 = Te0[(s3 >> 24)] ^
        Te1[(s0 >> 16) & 0xff] ^
        Te2[(s1 >> 8) & 0xff] ^ Te3[(s2) & 0xff] ^ rek[7];
    rek += 8;
    if (--r == 0) {
      break;
    }
    s0 = Te0[(t0 >> 24)] ^
        Te1[(t1 >> 16) & 0xff] ^
        Te2[(t2 >> 8) & 0xff] ^ Te3[(t3) & 0xff] ^ rek[0];
    s1 = Te0[(t1 >> 24)] ^
        Te1[(t2 >> 16) & 0xff] ^
        Te2[(t3 >> 8) & 0xff] ^ Te3[(t0) & 0xff] ^ rek[1];
    s2 = Te0[(t2 >> 24)] ^
        Te1[(t3 >> 16) & 0xff] ^
        Te2[(t0 >> 8) & 0xff] ^ Te3[(t1) & 0xff] ^ rek[2];
    s3 = Te0[(t3 >> 24)] ^
        Te1[(t0 >> 16) & 0xff] ^
        Te2[(t1 >> 8) & 0xff] ^ Te3[(t2) & 0xff] ^ rek[3];
  }

  ct[0] =
      (Te4[(t0 >> 24)] & 0xff000000) ^
      (Te4[(t1 >> 16) & 0xff] & 0x00ff0000) ^
      (Te4[(t2 >> 8) & 0xff] & 0x0000ff00) ^
      (Te4[(t3) & 0xff] & 0x000000ff) ^ rek[0];
  ct[1] =
      (Te4[(t1 >> 24)] & 0xff000000) ^
      (Te4[(t2 >> 16) & 0xff] & 0x00ff0000) ^
      (Te4[(t3 >> 8) & 0xff] & 0x0000ff00) ^
      (Te4[(t0) & 0xff] & 0x000000ff) ^ rek[1];
  ct[2] =
      (Te4[(t2 >> 24)] & 0xff000000) ^
      (Te4[(t3 >> 16) & 0xff] & 0x00ff0000) ^
      (Te4[(t0 >> 8) & 0xff] & 0x0000ff00) ^
      (Te4[(t1) & 0xff] & 0x000000ff) ^ rek[2];
  ct[3] =
      (Te4[(t3 >> 24)] & 0xff000000) ^
      (Te4[(t0 >> 16) & 0xff] & 0x00ff0000) ^
      (Te4[(t1 >> 8) & 0xff] & 0x0000ff00) ^
```

```
86        (Te4[(t2) & 0xff] & 0x000000ff) ^ rek[3];
87
88    return;
89  }
90  void AES_EncKey(const unint * ck, unint * rk, unint nk)
91  {
92    switch (nk) {
93    case 4:
94      AES_EncKey128(ck, rk);
95      break;
96    case 6:
97      AES_EncKey192(ck, rk);
98      break;
99    case 8:
100     AES_EncKey256(ck, rk);
101     break;
102   default:
103     return;
104   }
105 }
106 void AES_Enc(unint * pt, unint * ct, unint * rk, unint nk,unint pt_size)
107 {
108   unint i;
109   switch (nk) {
110   case 4:
111     for (i=0;i<pt_size/16;i++)  AES_Enc128(pt+4*i, ct+4*i, rk);
112     break;
113   case 6:
114     for (i=0;i<pt_size/16;i++)  AES_Enc192(pt+4*i, ct+4*i, rk);
115     break;
116   case 8:
117     for (i=0;i<pt_size/16;i++)  AES_Enc256(pt+4*i, ct+4*i, rk);
118     break;
119   default:
120     return;
121   }
122 }
```

II.4 REFERENCE_UNROLL

Código fonte II.14: reference_unroll/aes.c

```
1  void AES_EncKey128(const unint * ck, unint * rk)
2  {
3
4    rk[0] = ck[0];
5    rk[1] = ck[1];
6    rk[2] = ck[2];
7    rk[3] = ck[3];
8
9    rk[4] = rk[0] ^
10       (Te4[(rk[3] >> 16) & 0xff] & 0xff000000) ^
11       (Te4[(rk[3] >> 8) & 0xff] & 0x00ff0000) ^
12       (Te4[(rk[3]) & 0xff] & 0x0000ff00) ^
13       (Te4[(rk[3] >> 24)] & 0x000000ff) ^ rcon[0];
14
15   rk[5] = rk[1] ^ rk[4];
```

```
16    rk[6] = rk[2] ^ rk[5];
17    rk[7] = rk[3] ^ rk[6];
18    rk[8] = rk[4] ^
19        (Te4[(rk[7] >> 16) & 0xff] & 0xff000000) ^
20        (Te4[(rk[7] >> 8) & 0xff] & 0x00ff0000) ^
21        (Te4[(rk[7]) & 0xff] & 0x0000ff00) ^
22        (Te4[(rk[7] >> 24)] & 0x000000ff) ^ rcon[1];
23
24    rk[9] = rk[5] ^ rk[8];
25    rk[10] = rk[6] ^ rk[9];
26    rk[11] = rk[7] ^ rk[10];
27    rk[12] = rk[8] ^
28        (Te4[(rk[11] >> 16) & 0xff] & 0xff000000) ^
29        (Te4[(rk[11] >> 8) & 0xff] & 0x00ff0000) ^
30        (Te4[(rk[11]) & 0xff] & 0x0000ff00) ^
31        (Te4[(rk[11] >> 24)] & 0x000000ff) ^ rcon[2];
32
33    rk[13] = rk[9] ^ rk[12];
34    rk[14] = rk[10] ^ rk[13];
35    rk[15] = rk[11] ^ rk[14];
36    rk[16] = rk[12] ^
37        (Te4[(rk[15] >> 16) & 0xff] & 0xff000000) ^
38        (Te4[(rk[15] >> 8) & 0xff] & 0x00ff0000) ^
39        (Te4[(rk[15]) & 0xff] & 0x0000ff00) ^
40        (Te4[(rk[15] >> 24)] & 0x000000ff) ^ rcon[3];
41
42    rk[17] = rk[13] ^ rk[16];
43    rk[18] = rk[14] ^ rk[17];
44    rk[19] = rk[15] ^ rk[18];
45    rk[20] = rk[16] ^
46        (Te4[(rk[19] >> 16) & 0xff] & 0xff000000) ^
47        (Te4[(rk[19] >> 8) & 0xff] & 0x00ff0000) ^
48        (Te4[(rk[19]) & 0xff] & 0x0000ff00) ^
49        (Te4[(rk[19] >> 24)] & 0x000000ff) ^ rcon[4];
50
51    rk[21] = rk[17] ^ rk[20];
52    rk[22] = rk[18] ^ rk[21];
53    rk[23] = rk[19] ^ rk[22];
54    rk[24] = rk[20] ^
55        (Te4[(rk[23] >> 16) & 0xff] & 0xff000000) ^
56        (Te4[(rk[23] >> 8) & 0xff] & 0x00ff0000) ^
57        (Te4[(rk[23]) & 0xff] & 0x0000ff00) ^
58        (Te4[(rk[23] >> 24)] & 0x000000ff) ^ rcon[5];
59
60    rk[25] = rk[21] ^ rk[24];
61    rk[26] = rk[22] ^ rk[25];
62    rk[27] = rk[23] ^ rk[26];
63    rk[28] = rk[24] ^
64        (Te4[(rk[27] >> 16) & 0xff] & 0xff000000) ^
65        (Te4[(rk[27] >> 8) & 0xff] & 0x00ff0000) ^
66        (Te4[(rk[27]) & 0xff] & 0x0000ff00) ^
67        (Te4[(rk[27] >> 24)] & 0x000000ff) ^ rcon[6];
68
69    rk[29] = rk[25] ^ rk[28];
70    rk[30] = rk[26] ^ rk[29];
71    rk[31] = rk[27] ^ rk[30];
72    rk[32] = rk[28] ^
73        (Te4[(rk[31] >> 16) & 0xff] & 0xff000000) ^
74        (Te4[(rk[31] >> 8) & 0xff] & 0x00ff0000) ^
75        (Te4[(rk[31]) & 0xff] & 0x0000ff00) ^
```

```
76        (Te4[(rk[31] >> 24)] & 0x000000ff) ^ rcon[7];
77
78    rk[33] = rk[29] ^ rk[32];
79    rk[34] = rk[30] ^ rk[33];
80    rk[35] = rk[31] ^ rk[34];
81    rk[36] = rk[32] ^
82        (Te4[(rk[35] >> 16) & 0xff] & 0xff000000) ^
83        (Te4[(rk[35] >> 8) & 0xff] & 0x00ff0000) ^
84        (Te4[(rk[35]) & 0xff] & 0x0000ff00) ^
85        (Te4[(rk[35] >> 24)] & 0x000000ff) ^ rcon[8];
86
87    rk[37] = rk[33] ^ rk[36];
88    rk[38] = rk[34] ^ rk[37];
89    rk[39] = rk[35] ^ rk[38];
90    rk[40] = rk[36] ^
91        (Te4[(rk[39] >> 16) & 0xff] & 0xff000000) ^
92        (Te4[(rk[39] >> 8) & 0xff] & 0x00ff0000) ^
93        (Te4[(rk[39]) & 0xff] & 0x0000ff00) ^
94        (Te4[(rk[39] >> 24)] & 0x000000ff) ^ rcon[9];
95
96    rk[41] = rk[37] ^ rk[40];
97    rk[42] = rk[38] ^ rk[41];
98    rk[43] = rk[39] ^ rk[42];
99
100   return;
101 }
102 void AES_Enc128(unint * pt, unint * ct, unint * rek)
103 {
104   unint s0, s1, s2, s3, t0, t1, t2, t3;
105
106   s0 = pt[0] ^ rek[0];
107   s1 = pt[1] ^ rek[1];
108   s2 = pt[2] ^ rek[2];
109   s3 = pt[3] ^ rek[3];
110
111   /* round 1: */
112   t0 = Te0[s0 >> 24] ^ Te1[(s1 >> 16) & 0xff] ^ Te2[(s2 >> 8) & 0xff] ^
113       Te3[s3 & 0xff] ^ rek[4];
114   t1 = Te0[s1 >> 24] ^ Te1[(s2 >> 16) & 0xff] ^ Te2[(s3 >> 8) & 0xff] ^
115       Te3[s0 & 0xff] ^ rek[5];
116   t2 = Te0[s2 >> 24] ^ Te1[(s3 >> 16) & 0xff] ^ Te2[(s0 >> 8) & 0xff] ^
117       Te3[s1 & 0xff] ^ rek[6];
118   t3 = Te0[s3 >> 24] ^ Te1[(s0 >> 16) & 0xff] ^ Te2[(s1 >> 8) & 0xff] ^
119       Te3[s2 & 0xff] ^ rek[7];
120
121   /* round 2: */
122   s0 = Te0[t0 >> 24] ^ Te1[(t1 >> 16) & 0xff] ^ Te2[(t2 >> 8) & 0xff] ^
123       Te3[t3 & 0xff] ^ rek[8];
124   s1 = Te0[t1 >> 24] ^ Te1[(t2 >> 16) & 0xff] ^ Te2[(t3 >> 8) & 0xff] ^
125       Te3[t0 & 0xff] ^ rek[9];
126   s2 = Te0[t2 >> 24] ^ Te1[(t3 >> 16) & 0xff] ^ Te2[(t0 >> 8) & 0xff] ^
127       Te3[t1 & 0xff] ^ rek[10];
128   s3 = Te0[t3 >> 24] ^ Te1[(t0 >> 16) & 0xff] ^ Te2[(t1 >> 8) & 0xff] ^
129       Te3[t2 & 0xff] ^ rek[11];
130
131   /* round 3: */
132   t0 = Te0[s0 >> 24] ^ Te1[(s1 >> 16) & 0xff] ^ Te2[(s2 >> 8) & 0xff] ^
133       Te3[s3 & 0xff] ^ rek[12];
134   t1 = Te0[s1 >> 24] ^ Te1[(s2 >> 16) & 0xff] ^ Te2[(s3 >> 8) & 0xff] ^
135       Te3[s0 & 0xff] ^ rek[13];
```

```
136     t2 = Te0[s2 >> 24] ^ Te1[(s3 >> 16) & 0xff] ^ Te2[(s0 >> 8) & 0xff] ^
137         Te3[s1 & 0xff] ^ rek[14];
138     t3 = Te0[s3 >> 24] ^ Te1[(s0 >> 16) & 0xff] ^ Te2[(s1 >> 8) & 0xff] ^
139         Te3[s2 & 0xff] ^ rek[15];
140 
141     /* round 4: */
142     s0 = Te0[t0 >> 24] ^ Te1[(t1 >> 16) & 0xff] ^ Te2[(t2 >> 8) & 0xff] ^
143         Te3[t3 & 0xff] ^ rek[16];
144     s1 = Te0[t1 >> 24] ^ Te1[(t2 >> 16) & 0xff] ^ Te2[(t3 >> 8) & 0xff] ^
145         Te3[t0 & 0xff] ^ rek[17];
146     s2 = Te0[t2 >> 24] ^ Te1[(t3 >> 16) & 0xff] ^ Te2[(t0 >> 8) & 0xff] ^
147         Te3[t1 & 0xff] ^ rek[18];
148     s3 = Te0[t3 >> 24] ^ Te1[(t0 >> 16) & 0xff] ^ Te2[(t1 >> 8) & 0xff] ^
149         Te3[t2 & 0xff] ^ rek[19];
150 
151     /* round 5: */
152     t0 = Te0[s0 >> 24] ^ Te1[(s1 >> 16) & 0xff] ^ Te2[(s2 >> 8) & 0xff] ^
153         Te3[s3 & 0xff] ^ rek[20];
154     t1 = Te0[s1 >> 24] ^ Te1[(s2 >> 16) & 0xff] ^ Te2[(s3 >> 8) & 0xff] ^
155         Te3[s0 & 0xff] ^ rek[21];
156     t2 = Te0[s2 >> 24] ^ Te1[(s3 >> 16) & 0xff] ^ Te2[(s0 >> 8) & 0xff] ^
157         Te3[s1 & 0xff] ^ rek[22];
158     t3 = Te0[s3 >> 24] ^ Te1[(s0 >> 16) & 0xff] ^ Te2[(s1 >> 8) & 0xff] ^
159         Te3[s2 & 0xff] ^ rek[23];
160 
161     /* round 6: */
162     s0 = Te0[t0 >> 24] ^ Te1[(t1 >> 16) & 0xff] ^ Te2[(t2 >> 8) & 0xff] ^
163         Te3[t3 & 0xff] ^ rek[24];
164     s1 = Te0[t1 >> 24] ^ Te1[(t2 >> 16) & 0xff] ^ Te2[(t3 >> 8) & 0xff] ^
165         Te3[t0 & 0xff] ^ rek[25];
166     s2 = Te0[t2 >> 24] ^ Te1[(t3 >> 16) & 0xff] ^ Te2[(t0 >> 8) & 0xff] ^
167         Te3[t1 & 0xff] ^ rek[26];
168     s3 = Te0[t3 >> 24] ^ Te1[(t0 >> 16) & 0xff] ^ Te2[(t1 >> 8) & 0xff] ^
169         Te3[t2 & 0xff] ^ rek[27];
170 
171     /* round 7: */
172     t0 = Te0[s0 >> 24] ^ Te1[(s1 >> 16) & 0xff] ^ Te2[(s2 >> 8) & 0xff] ^
173         Te3[s3 & 0xff] ^ rek[28];
174     t1 = Te0[s1 >> 24] ^ Te1[(s2 >> 16) & 0xff] ^ Te2[(s3 >> 8) & 0xff] ^
175         Te3[s0 & 0xff] ^ rek[29];
176     t2 = Te0[s2 >> 24] ^ Te1[(s3 >> 16) & 0xff] ^ Te2[(s0 >> 8) & 0xff] ^
177         Te3[s1 & 0xff] ^ rek[30];
178     t3 = Te0[s3 >> 24] ^ Te1[(s0 >> 16) & 0xff] ^ Te2[(s1 >> 8) & 0xff] ^
179         Te3[s2 & 0xff] ^ rek[31];
180 
181     /* round 8: */
182     s0 = Te0[t0 >> 24] ^ Te1[(t1 >> 16) & 0xff] ^ Te2[(t2 >> 8) & 0xff] ^
183         Te3[t3 & 0xff] ^ rek[32];
184     s1 = Te0[t1 >> 24] ^ Te1[(t2 >> 16) & 0xff] ^ Te2[(t3 >> 8) & 0xff] ^
185         Te3[t0 & 0xff] ^ rek[33];
186     s2 = Te0[t2 >> 24] ^ Te1[(t3 >> 16) & 0xff] ^ Te2[(t0 >> 8) & 0xff] ^
187         Te3[t1 & 0xff] ^ rek[34];
188     s3 = Te0[t3 >> 24] ^ Te1[(t0 >> 16) & 0xff] ^ Te2[(t1 >> 8) & 0xff] ^
189         Te3[t2 & 0xff] ^ rek[35];
190 
191     /* round 9: */
192     t0 = Te0[s0 >> 24] ^ Te1[(s1 >> 16) & 0xff] ^ Te2[(s2 >> 8) & 0xff] ^
193         Te3[s3 & 0xff] ^ rek[36];
194     t1 = Te0[s1 >> 24] ^ Te1[(s2 >> 16) & 0xff] ^ Te2[(s3 >> 8) & 0xff] ^
195         Te3[s0 & 0xff] ^ rek[37];
```

```
196   t2 = Te0[s2 >> 24] ^ Te1[(s3 >> 16) & 0xff] ^ Te2[(s0 >> 8) & 0xff] ^
197        Te3[s1 & 0xff] ^ rek[38];
198   t3 = Te0[s3 >> 24] ^ Te1[(s0 >> 16) & 0xff] ^ Te2[(s1 >> 8) & 0xff] ^
199        Te3[s2 & 0xff] ^ rek[39];
200
201   ct[0] =
202        (Te4[(t0 >> 24)] & 0xff000000) ^
203        (Te4[(t1 >> 16) & 0xff] & 0x00ff0000) ^
204        (Te4[(t2 >> 8) & 0xff] & 0x0000ff00) ^
205        (Te4[(t3) & 0xff] & 0x000000ff) ^ rek[40];
206   ct[1] =
207        (Te4[(t1 >> 24)] & 0xff000000) ^
208        (Te4[(t2 >> 16) & 0xff] & 0x00ff0000) ^
209        (Te4[(t3 >> 8) & 0xff] & 0x0000ff00) ^
210        (Te4[(t0) & 0xff] & 0x000000ff) ^ rek[41];
211   ct[2] =
212        (Te4[(t2 >> 24)] & 0xff000000) ^
213        (Te4[(t3 >> 16) & 0xff] & 0x00ff0000) ^
214        (Te4[(t0 >> 8) & 0xff] & 0x0000ff00) ^
215        (Te4[(t1) & 0xff] & 0x000000ff) ^ rek[42];
216   ct[3] =
217        (Te4[(t3 >> 24)] & 0xff000000) ^
218        (Te4[(t0 >> 16) & 0xff] & 0x00ff0000) ^
219        (Te4[(t1 >> 8) & 0xff] & 0x0000ff00) ^
220        (Te4[(t2) & 0xff] & 0x000000ff) ^ rek[43];
221
222   return;
223 }
```

II.5 cuda_aes_nn_global_roll

Código fonte II.15: cuda_aes_nn_global_roll/aes.cu

```
1  __global__ void AES_Enc128(unint * pt, unint * ct, unint * rek, unint N) {
2
3    unint tid = blockIdx.y * gridDim.x * blockDim.x + blockIdx.x * blockDim.x
4        + threadIdx.x;
5
6    if (tid > (N - 1))
7      return;
8
9    unint s0, s1, s2, s3, t0, t1, t2, t3;
10   s0 = pt[4 * tid] ^ rek[0];
11   s1 = pt[4 * tid + 1] ^ rek[1];
12   s2 = pt[4 * tid + 2] ^ rek[2];
13   s3 = pt[4 * tid + 3] ^ rek[3];
14
15   uint r = 5;
16   for (;;) {
17     t0 = d_Te0[(s0 >> 24)] ^ d_Te1[(s1 >> 16) & 0xff]
18          ^ d_Te2[(s2 >> 8) & 0xff] ^ d_Te3[(s3) & 0xff] ^ rek[4];
19     t1 = d_Te0[(s1 >> 24)] ^ d_Te1[(s2 >> 16) & 0xff]
20          ^ d_Te2[(s3 >> 8) & 0xff] ^ d_Te3[(s0) & 0xff] ^ rek[5];
21     t2 = d_Te0[(s2 >> 24)] ^ d_Te1[(s3 >> 16) & 0xff]
22          ^ d_Te2[(s0 >> 8) & 0xff] ^ d_Te3[(s1) & 0xff] ^ rek[6];
23     t3 = d_Te0[(s3 >> 24)] ^ d_Te1[(s0 >> 16) & 0xff]
24          ^ d_Te2[(s1 >> 8) & 0xff] ^ d_Te3[(s2) & 0xff] ^ rek[7];
```

```
25      rek += 8;
26      if (--r == 0) {
27        break;
28      }
29      s0 = d_Te0[(t0 >> 24)] ^ d_Te1[(t1 >> 16) & 0xff]
30          ^ d_Te2[(t2 >> 8) & 0xff] ^ d_Te3[(t3) & 0xff] ^ rek[0];
31      s1 = d_Te0[(t1 >> 24)] ^ d_Te1[(t2 >> 16) & 0xff]
32          ^ d_Te2[(t3 >> 8) & 0xff] ^ d_Te3[(t0) & 0xff] ^ rek[1];
33      s2 = d_Te0[(t2 >> 24)] ^ d_Te1[(t3 >> 16) & 0xff]
34          ^ d_Te2[(t0 >> 8) & 0xff] ^ d_Te3[(t1) & 0xff] ^ rek[2];
35      s3 = d_Te0[(t3 >> 24)] ^ d_Te1[(t0 >> 16) & 0xff]
36          ^ d_Te2[(t1 >> 8) & 0xff] ^ d_Te3[(t2) & 0xff] ^ rek[3];
37    }
38
39    ct[4 * tid] = (d_Te4[(t0 >> 24)] & 0xff000000)
40        ^ (d_Te4[(t1 >> 16) & 0xff] & 0x00ff0000)
41        ^ (d_Te4[(t2 >> 8) & 0xff] & 0x0000ff00)
42        ^ (d_Te4[(t3) & 0xff] & 0x000000ff) ^ rek[0];
43    ct[4 * tid + 1] = (d_Te4[(t1 >> 24)] & 0xff000000)
44        ^ (d_Te4[(t2 >> 16) & 0xff] & 0x00ff0000)
45        ^ (d_Te4[(t3 >> 8) & 0xff] & 0x0000ff00)
46        ^ (d_Te4[(t0) & 0xff] & 0x000000ff) ^ rek[1];
47    ct[4 * tid + 2] = (d_Te4[(t2 >> 24)] & 0xff000000)
48        ^ (d_Te4[(t3 >> 16) & 0xff] & 0x00ff0000)
49        ^ (d_Te4[(t0 >> 8) & 0xff] & 0x0000ff00)
50        ^ (d_Te4[(t1) & 0xff] & 0x000000ff) ^ rek[2];
51    ct[4 * tid + 3] = (d_Te4[(t3 >> 24)] & 0xff000000)
52        ^ (d_Te4[(t0 >> 16) & 0xff] & 0x00ff0000)
53        ^ (d_Te4[(t1 >> 8) & 0xff] & 0x0000ff00)
54        ^ (d_Te4[(t2) & 0xff] & 0x000000ff) ^ rek[3];
55
56    return;
57  }
58
59  void AES_Enc(unint * pt, unint * ct, unint * rk, unint nk, unint pt_size,
60      unint threads_per_block) {
61  #ifdef TEMPO
62    cudaEvent_t tstart, tstop;
63    HANDLE_ERROR(cudaEventCreate(&tstart));
64    HANDLE_ERROR(cudaEventCreate(&tstop));
65
66    cudaEvent_t cstart, cstop;
67    HANDLE_ERROR(cudaEventCreate(&cstart));
68    HANDLE_ERROR(cudaEventCreate(&cstop));
69
70    cudaEvent_t start, stop;
71    HANDLE_ERROR(cudaEventCreate(&start));
72    HANDLE_ERROR(cudaEventCreate(&stop));
73
74    float elapsedTime;
75  #endif
76
77    unint *d_pt, *d_ct, *d_rk;
78
79    unint N = pt_size / 16;
80    unint nr = NR(nk);
81    unint nkx = 4 * (nr + 1);
82    unint nkxb = nkx * sizeof(unint);
83
84    cudaDeviceProp prop;
```

```
85    HANDLE_ERROR( cudaGetDeviceProperties( &prop,0));
86
87    if (threads_per_block > prop.maxThreadsPerBlock)
88      threads_per_block = prop.maxThreadsPerBlock;
89
90    unint blocks = (unint) N / threads_per_block;
91
92    if (N % threads_per_block != 0)
93      blocks++;
94    unint blocks1 = blocks;
95    unint blocks2 = 1;
96
97    // Blocks transformado em bidimensional se ultrapassar maximo
98    unint maxgridsize = prop.maxGridSize[0];
99    if (blocks > maxgridsize) {
100     unint max = (unint) sqrt(blocks);
101     blocks1 = blocks;
102     blocks2 = 1;
103     unint i = 0;
104     // procura divisores de blocks
105     while (primo[i] <= max) {
106       if (blocks1 % primo[i] == 0) {
107         blocks1 = blocks1 / primo[i];
108         blocks2 = blocks2 * primo[i];
109         if (blocks1 > maxgridsize)
110           continue;
111         break;
112       }
113       i++;
114     }
115     //se nao encontrar divisores tentar minimizar blocks1 e blocks2
116     if ((blocks1 > maxgridsize) || (blocks2 > maxgridsize)) {
117       blocks1 = max;
118       blocks2 = max;
119       while (blocks1 * blocks2 * threads_per_block < N)
120         blocks2++;
121     }
122   }
123   dim3 BLOCKS(blocks1, blocks2, 1);
124
125 #ifdef TEMPO
126   HANDLE_ERROR(cudaEventRecord(tstart, 0));
127 #endif
128   HANDLE_ERROR(cudaMalloc((void **)&d_pt, pt_size));
129   HANDLE_ERROR(cudaMalloc((void **)&d_ct, pt_size));
130   HANDLE_ERROR(cudaMalloc((void **)&d_rk, nkxb));
131
132 #ifdef TEMPO
133   HANDLE_ERROR(cudaEventRecord(cstart, 0));
134 #endif
135   HANDLE_ERROR(cudaMemcpy(d_pt, pt, pt_size, cudaMemcpyHostToDevice));
136   HANDLE_ERROR(cudaMemcpy(d_rk, rk, nkxb, cudaMemcpyHostToDevice));
137
138 #ifdef TEMPO
139   HANDLE_ERROR(cudaEventRecord(start, 0));
140 #endif
141   switch (nk) {
142   case 4:
143     AES_Enc128 <<< BLOCKS, threads_per_block >>> (d_pt, d_ct, d_rk,N);
144     break;
```

```
145     case 6:
146       AES_Enc192 <<< BLOCKS, threads_per_block >>> (d_pt, d_ct, d_rk,N);
147       break;
148     case 8:
149       AES_Enc256 <<< BLOCKS, threads_per_block >>> (d_pt, d_ct, d_rk,N);
150       break;
151     default:
152       return;
153     }
154 #ifdef TEMPO
155     HANDLE_ERROR(cudaEventRecord(stop, 0));
156 #endif
157 
158     HANDLE_ERROR(cudaMemcpy(ct, d_ct, pt_size, cudaMemcpyDeviceToHost));
159 
160 #ifdef TEMPO
161     HANDLE_ERROR(cudaEventRecord(cstop, 0));
162 #endif
163 
164     HANDLE_ERROR(cudaFree(d_pt));
165     HANDLE_ERROR(cudaFree(d_ct));
166     HANDLE_ERROR(cudaFree(d_rk));
167 
168 #ifdef TEMPO
169     HANDLE_ERROR(cudaEventRecord(tstop, 0));
170     HANDLE_ERROR(cudaEventSynchronize(tstop));
171 
172     HANDLE_ERROR(cudaEventElapsedTime(&elapsedTime, start, stop));
173     fprintf(stderr, "time(enc_cuda_%d):\t\t%12.6f ms\t%4d threads\t%u bytes\n",
174         (4 * nk * 8), elapsedTime, threads_per_block,pt_size);
175 
176     HANDLE_ERROR(cudaEventElapsedTime(&elapsedTime, cstart, cstop));
177     fprintf(stderr, "time(enc_cuda_cp_%d):\t%12.6f ms\t%4d threads\t%u bytes\n",
178         (4 * nk * 8), elapsedTime, threads_per_block,pt_size);
179 
180     HANDLE_ERROR(cudaEventElapsedTime(&elapsedTime, tstart, tstop));
181     fprintf(stderr, "time(enc_cuda_tt_%d):\t%12.6f ms\t%4d threads\t%u bytes\n",
182         (4 * nk * 8), elapsedTime, threads_per_block,pt_size);
183 
184     HANDLE_ERROR(cudaEventDestroy(start));
185     HANDLE_ERROR(cudaEventDestroy(stop));
186 
187     HANDLE_ERROR(cudaEventDestroy(cstart));
188     HANDLE_ERROR(cudaEventDestroy(cstop));
189 
190     HANDLE_ERROR(cudaEventDestroy(tstart));
191     HANDLE_ERROR(cudaEventDestroy(tstop));
192 #endif
193 }
194 
195 void AES_Enc_Async(unint * pt, unint * ct, unint * rk, unint nk, unint pt_size,
196     unint threads_per_block, unint BLOCOS, unint STREAMS) {
197 #ifdef TEMPO
198     cudaEvent_t tstart, tstop;
199     HANDLE_ERROR(cudaEventCreate(&tstart));
200     HANDLE_ERROR(cudaEventCreate(&tstop));
201 
202     float elapsedTime;
203 #endif
204 
```

```
205   unint nb = 4;
206   unint nbb = nb * sizeof(unint);
207
208   unint N = pt_size / nbb;
209   unint nr = NR(nk);
210   unint nkx = 4 * (nr + 1);
211   unint nkxb = nkx * sizeof(unint);
212
213   cudaDeviceProp prop;
214   HANDLE_ERROR( cudaGetDeviceProperties( &prop,0));
215
216   if (!prop.deviceOverlap) {
217     printf("Overlap nao disponivel. Nao e possivel rodar ASYNC\n");
218     return;
219   }
220
221   unint NN;
222   NN = BLOCOS * threads_per_block;
223
224   unint NS[STREAMS];
225   cudaStream_t stream[STREAMS];
226
227   int M = N;
228   unint *d_pt, *d_ct, *d_rk;
229
230   int i, j;
231 #ifdef TEMPO
232   HANDLE_ERROR(cudaEventRecord(tstart, 0));
233 #endif
234   // criar streams
235   for (i = 0; i < STREAMS; i++) {
236     HANDLE_ERROR( cudaStreamCreate( &stream[i] ));
237   }
238
239   HANDLE_ERROR(cudaMalloc((void **)&d_pt, pt_size));
240   HANDLE_ERROR(cudaMalloc((void **)&d_ct, pt_size));
241   HANDLE_ERROR(cudaMalloc((void **)&d_rk, nkxb));
242   HANDLE_ERROR(cudaMemcpy(d_rk, rk, nkxb, cudaMemcpyHostToDevice));
243
244   i = 0;
245   while (M > 0) {
246     // determinar N para cada stream
247     for (j = 0; j < STREAMS; j++) {
248       if ((M / NN) >= 1) {
249         NS[j] = NN;
250       } else {
251         NS[j] = M % NN;
252       }
253       M -= NS[j];
254     }
255     // copiar async host to device para cada stream
256     for (j = 0; j < STREAMS; j++) {
257       if (NS[j] > 0) {
258         HANDLE_ERROR(
259             cudaMemcpyAsync(d_pt+(i+j)*NN*nb, pt+(i+j)*NN*nb, NS[j]*nbb,
                  cudaMemcpyHostToDevice, stream[j]));
260       }
261     }
262
263     // inciar processamento para cada stream
```

```
264     switch (nk) {
265     case 4:
266       for (j = 0; j < STREAMS; j++) {
267         if (NS[j] > 0) {
268 AES_Enc128        <<< BLOCOS, threads_per_block, 0 , stream[j] >>> (d_pt+(i+j)*NN*nb,
        d_ct+(i+j)*NN*nb, d_rk, NS[j]);
269         }
270     }
271     break;
272     case 6:
273     for (j=0;j<STREAMS;j++) {
274       if (NS[j] > 0) {
275         AES_Enc192<<< BLOCOS, threads_per_block, 0 , stream[j] >>> (d_pt+(i+j)*NN*nb,
              d_ct+(i+j)*NN*nb, d_rk, NS[j]);
276       }
277     }
278     break;
279     case 8:
280     for (j=0;j<STREAMS;j++) {
281       if (NS[j] > 0) {
282         AES_Enc256<<< BLOCOS, threads_per_block, 0 , stream[j] >>> (d_pt+(i+j)*NN*nb,
              d_ct+(i+j)*NN*nb, d_rk, NS[j]);
283       }
284     }
285     break;
286     default:
287     return;
288   }
289   // copiar async device to host para cada stream
290     for (j = 0; j < STREAMS; j++) {
291       if (NS[j] > 0) {
292         HANDLE_ERROR(
293              cudaMemcpyAsync(ct+(i+j)*NN*nb, d_ct+(i+j)*NN*nb, NS[j]*nbb,
                  cudaMemcpyDeviceToHost, stream[j]));
294       }
295     }
296
297     i += STREAMS;
298   }
299
300   // sincronizar streams
301   for (j = 0; j < STREAMS; j++) {
302     HANDLE_ERROR( cudaStreamSynchronize( stream[j]));
303   }
304
305   HANDLE_ERROR(cudaFree(d_pt));
306   HANDLE_ERROR(cudaFree(d_ct));
307   HANDLE_ERROR(cudaFree(d_rk));
308
309 #ifdef TEMPO
310   HANDLE_ERROR(cudaEventRecord(tstop, 0));
311   HANDLE_ERROR(cudaEventSynchronize(tstop));
312
313   HANDLE_ERROR(cudaEventElapsedTime(&elapsedTime, tstart, tstop));
314   fprintf(stderr, "time(enc_cuda_tt_%d):\t%12.6f ms\t%4d threads\t%4d blocos\t%3d
        streams\t%u bytes\n",
315       (4 * nk * 8), elapsedTime, threads_per_block, BLOCOS, STREAMS, pt_size);
316
317   HANDLE_ERROR(cudaEventDestroy(tstart));
318   HANDLE_ERROR(cudaEventDestroy(tstop));
```

```
#endif
}
```

II.6 cuda_aes_nn_global_unroll

Código fonte II.16: cuda_aes_nn_global_unroll/aes.cu

```
__global__ void AES_Enc128(unint * pt, unint * ct, unint * d_rk, unint N) {

  unint tid = blockIdx.y * gridDim.x * blockDim.x + blockIdx.x * blockDim.x
      + threadIdx.x;

  if (tid > (N - 1))
    return;

  unint s0, s1, s2, s3, t0, t1, t2, t3;

  s0 = pt[4 * tid] ^ d_rk[0];
  s1 = pt[4 * tid + 1] ^ d_rk[1];
  s2 = pt[4 * tid + 2] ^ d_rk[2];
  s3 = pt[4 * tid + 3] ^ d_rk[3];

  /* round 1: */
  t0 = d_Te0[s0 >> 24] ^ d_Te1[(s1 >> 16) & 0xff] ^ d_Te2[(s2 >> 8) & 0xff]
      ^ d_Te3[s3 & 0xff] ^ d_rk[4];
  t1 = d_Te0[s1 >> 24] ^ d_Te1[(s2 >> 16) & 0xff] ^ d_Te2[(s3 >> 8) & 0xff]
      ^ d_Te3[s0 & 0xff] ^ d_rk[5];
  t2 = d_Te0[s2 >> 24] ^ d_Te1[(s3 >> 16) & 0xff] ^ d_Te2[(s0 >> 8) & 0xff]
      ^ d_Te3[s1 & 0xff] ^ d_rk[6];
  t3 = d_Te0[s3 >> 24] ^ d_Te1[(s0 >> 16) & 0xff] ^ d_Te2[(s1 >> 8) & 0xff]
      ^ d_Te3[s2 & 0xff] ^ d_rk[7];

  /* round 2: */
  s0 = d_Te0[t0 >> 24] ^ d_Te1[(t1 >> 16) & 0xff] ^ d_Te2[(t2 >> 8) & 0xff]
      ^ d_Te3[t3 & 0xff] ^ d_rk[8];
  s1 = d_Te0[t1 >> 24] ^ d_Te1[(t2 >> 16) & 0xff] ^ d_Te2[(t3 >> 8) & 0xff]
      ^ d_Te3[t0 & 0xff] ^ d_rk[9];
  s2 = d_Te0[t2 >> 24] ^ d_Te1[(t3 >> 16) & 0xff] ^ d_Te2[(t0 >> 8) & 0xff]
      ^ d_Te3[t1 & 0xff] ^ d_rk[10];
  s3 = d_Te0[t3 >> 24] ^ d_Te1[(t0 >> 16) & 0xff] ^ d_Te2[(t1 >> 8) & 0xff]
      ^ d_Te3[t2 & 0xff] ^ d_rk[11];

  /* round 3: */
  t0 = d_Te0[s0 >> 24] ^ d_Te1[(s1 >> 16) & 0xff] ^ d_Te2[(s2 >> 8) & 0xff]
      ^ d_Te3[s3 & 0xff] ^ d_rk[12];
  t1 = d_Te0[s1 >> 24] ^ d_Te1[(s2 >> 16) & 0xff] ^ d_Te2[(s3 >> 8) & 0xff]
      ^ d_Te3[s0 & 0xff] ^ d_rk[13];
  t2 = d_Te0[s2 >> 24] ^ d_Te1[(s3 >> 16) & 0xff] ^ d_Te2[(s0 >> 8) & 0xff]
      ^ d_Te3[s1 & 0xff] ^ d_rk[14];
  t3 = d_Te0[s3 >> 24] ^ d_Te1[(s0 >> 16) & 0xff] ^ d_Te2[(s1 >> 8) & 0xff]
      ^ d_Te3[s2 & 0xff] ^ d_rk[15];

  /* round 4: */
  s0 = d_Te0[t0 >> 24] ^ d_Te1[(t1 >> 16) & 0xff] ^ d_Te2[(t2 >> 8) & 0xff]
      ^ d_Te3[t3 & 0xff] ^ d_rk[16];
  s1 = d_Te0[t1 >> 24] ^ d_Te1[(t2 >> 16) & 0xff] ^ d_Te2[(t3 >> 8) & 0xff]
      ^ d_Te3[t0 & 0xff] ^ d_rk[17];
```

```
    s2 = d_Te0[t2 >> 24] ^ d_Te1[(t3 >> 16) & 0xff] ^ d_Te2[(t0 >> 8) & 0xff]
        ^ d_Te3[t1 & 0xff] ^ d_rk[18];
    s3 = d_Te0[t3 >> 24] ^ d_Te1[(t0 >> 16) & 0xff] ^ d_Te2[(t1 >> 8) & 0xff]
        ^ d_Te3[t2 & 0xff] ^ d_rk[19];

    /* round 5: */
    t0 = d_Te0[s0 >> 24] ^ d_Te1[(s1 >> 16) & 0xff] ^ d_Te2[(s2 >> 8) & 0xff]
        ^ d_Te3[s3 & 0xff] ^ d_rk[20];
    t1 = d_Te0[s1 >> 24] ^ d_Te1[(s2 >> 16) & 0xff] ^ d_Te2[(s3 >> 8) & 0xff]
        ^ d_Te3[s0 & 0xff] ^ d_rk[21];
    t2 = d_Te0[s2 >> 24] ^ d_Te1[(s3 >> 16) & 0xff] ^ d_Te2[(s0 >> 8) & 0xff]
        ^ d_Te3[s1 & 0xff] ^ d_rk[22];
    t3 = d_Te0[s3 >> 24] ^ d_Te1[(s0 >> 16) & 0xff] ^ d_Te2[(s1 >> 8) & 0xff]
        ^ d_Te3[s2 & 0xff] ^ d_rk[23];

    /* round 6: */
    s0 = d_Te0[t0 >> 24] ^ d_Te1[(t1 >> 16) & 0xff] ^ d_Te2[(t2 >> 8) & 0xff]
        ^ d_Te3[t3 & 0xff] ^ d_rk[24];
    s1 = d_Te0[t1 >> 24] ^ d_Te1[(t2 >> 16) & 0xff] ^ d_Te2[(t3 >> 8) & 0xff]
        ^ d_Te3[t0 & 0xff] ^ d_rk[25];
    s2 = d_Te0[t2 >> 24] ^ d_Te1[(t3 >> 16) & 0xff] ^ d_Te2[(t0 >> 8) & 0xff]
        ^ d_Te3[t1 & 0xff] ^ d_rk[26];
    s3 = d_Te0[t3 >> 24] ^ d_Te1[(t0 >> 16) & 0xff] ^ d_Te2[(t1 >> 8) & 0xff]
        ^ d_Te3[t2 & 0xff] ^ d_rk[27];

    /* round 7: */
    t0 = d_Te0[s0 >> 24] ^ d_Te1[(s1 >> 16) & 0xff] ^ d_Te2[(s2 >> 8) & 0xff]
        ^ d_Te3[s3 & 0xff] ^ d_rk[28];
    t1 = d_Te0[s1 >> 24] ^ d_Te1[(s2 >> 16) & 0xff] ^ d_Te2[(s3 >> 8) & 0xff]
        ^ d_Te3[s0 & 0xff] ^ d_rk[29];
    t2 = d_Te0[s2 >> 24] ^ d_Te1[(s3 >> 16) & 0xff] ^ d_Te2[(s0 >> 8) & 0xff]
        ^ d_Te3[s1 & 0xff] ^ d_rk[30];
    t3 = d_Te0[s3 >> 24] ^ d_Te1[(s0 >> 16) & 0xff] ^ d_Te2[(s1 >> 8) & 0xff]
        ^ d_Te3[s2 & 0xff] ^ d_rk[31];

    /* round 8: */
    s0 = d_Te0[t0 >> 24] ^ d_Te1[(t1 >> 16) & 0xff] ^ d_Te2[(t2 >> 8) & 0xff]
        ^ d_Te3[t3 & 0xff] ^ d_rk[32];
    s1 = d_Te0[t1 >> 24] ^ d_Te1[(t2 >> 16) & 0xff] ^ d_Te2[(t3 >> 8) & 0xff]
        ^ d_Te3[t0 & 0xff] ^ d_rk[33];
    s2 = d_Te0[t2 >> 24] ^ d_Te1[(t3 >> 16) & 0xff] ^ d_Te2[(t0 >> 8) & 0xff]
        ^ d_Te3[t1 & 0xff] ^ d_rk[34];
    s3 = d_Te0[t3 >> 24] ^ d_Te1[(t0 >> 16) & 0xff] ^ d_Te2[(t1 >> 8) & 0xff]
        ^ d_Te3[t2 & 0xff] ^ d_rk[35];

    /* round 9: */
    t0 = d_Te0[s0 >> 24] ^ d_Te1[(s1 >> 16) & 0xff] ^ d_Te2[(s2 >> 8) & 0xff]
        ^ d_Te3[s3 & 0xff] ^ d_rk[36];
    t1 = d_Te0[s1 >> 24] ^ d_Te1[(s2 >> 16) & 0xff] ^ d_Te2[(s3 >> 8) & 0xff]
        ^ d_Te3[s0 & 0xff] ^ d_rk[37];
    t2 = d_Te0[s2 >> 24] ^ d_Te1[(s3 >> 16) & 0xff] ^ d_Te2[(s0 >> 8) & 0xff]
        ^ d_Te3[s1 & 0xff] ^ d_rk[38];
    t3 = d_Te0[s3 >> 24] ^ d_Te1[(s0 >> 16) & 0xff] ^ d_Te2[(s1 >> 8) & 0xff]
        ^ d_Te3[s2 & 0xff] ^ d_rk[39];

    ct[4 * tid] = (d_Te4[(t0 >> 24)] & 0xff000000)
        ^ (d_Te4[(t1 >> 16) & 0xff] & 0x00ff0000)
        ^ (d_Te4[(t2 >> 8) & 0xff] & 0x0000ff00)
        ^ (d_Te4[(t3) & 0xff] & 0x000000ff) ^ d_rk[40];
    ct[4 * tid + 1] = (d_Te4[(t1 >> 24)] & 0xff000000)
```

```
      ^ (d_Te4[(t2 >> 16) & 0xff] & 0x00ff0000)
      ^ (d_Te4[(t3 >> 8) & 0xff] & 0x0000ff00)
      ^ (d_Te4[(t0) & 0xff] & 0x000000ff) ^ d_rk[41];
  ct[4 * tid + 2] = (d_Te4[(t2 >> 24)] & 0xff000000)
      ^ (d_Te4[(t3 >> 16) & 0xff] & 0x00ff0000)
      ^ (d_Te4[(t0 >> 8) & 0xff] & 0x0000ff00)
      ^ (d_Te4[(t1) & 0xff] & 0x000000ff) ^ d_rk[42];
  ct[4 * tid + 3] = (d_Te4[(t3 >> 24)] & 0xff000000)
      ^ (d_Te4[(t0 >> 16) & 0xff] & 0x00ff0000)
      ^ (d_Te4[(t1 >> 8) & 0xff] & 0x0000ff00)
      ^ (d_Te4[(t2) & 0xff] & 0x000000ff) ^ d_rk[43];

  return;
}
```

II.7 cuda_aes_1b_nt_const_const

Código fonte II.17: cuda_aes_1b_nt_const_const/aes.cu

```
__constant__ __device__ unint d_rk[60];
__global__ void AES_Enc128(unint * pt, unint * ct, unint N) {

  unint tid = blockIdx.x * blockDim.x + threadIdx.x;

  unint s0, s1, s2, s3, t0, t1, t2, t3;

  while (tid < N) {

    s0 = pt[4 * tid] ^ d_rk[0];
    s1 = pt[4 * tid + 1] ^ d_rk[1];
    s2 = pt[4 * tid + 2] ^ d_rk[2];
    s3 = pt[4 * tid + 3] ^ d_rk[3];

    /* round 1: */
    t0 = d_Te0[s0 >> 24] ^ d_Te1[(s1 >> 16) & 0xff]
        ^ d_Te2[(s2 >> 8) & 0xff] ^ d_Te3[s3 & 0xff]
        ^ d_rk[4];
    t1 = d_Te0[s1 >> 24] ^ d_Te1[(s2 >> 16) & 0xff]
        ^ d_Te2[(s3 >> 8) & 0xff] ^ d_Te3[s0 & 0xff]
        ^ d_rk[5];
    t2 = d_Te0[s2 >> 24] ^ d_Te1[(s3 >> 16) & 0xff]
        ^ d_Te2[(s0 >> 8) & 0xff] ^ d_Te3[s1 & 0xff]
        ^ d_rk[6];
    t3 = d_Te0[s3 >> 24] ^ d_Te1[(s0 >> 16) & 0xff]
        ^ d_Te2[(s1 >> 8) & 0xff] ^ d_Te3[s2 & 0xff]
        ^ d_rk[7];

    /* round 2: */
    s0 = d_Te0[t0 >> 24] ^ d_Te1[(t1 >> 16) & 0xff]
        ^ d_Te2[(t2 >> 8) & 0xff] ^ d_Te3[t3 & 0xff]
        ^ d_rk[8];
    s1 = d_Te0[t1 >> 24] ^ d_Te1[(t2 >> 16) & 0xff]
        ^ d_Te2[(t3 >> 8) & 0xff] ^ d_Te3[t0 & 0xff]
        ^ d_rk[9];
    s2 = d_Te0[t2 >> 24] ^ d_Te1[(t3 >> 16) & 0xff]
        ^ d_Te2[(t0 >> 8) & 0xff] ^ d_Te3[t1 & 0xff]
        ^ d_rk[10];
```

```
    s3 = d_Te0[t3 >> 24] ^ d_Te1[(t0 >> 16) & 0xff]
        ^ d_Te2[(t1 >> 8) & 0xff] ^ d_Te3[t2 & 0xff]
        ^ d_rk[11];

    /* round 3: */
    t0 = d_Te0[s0 >> 24] ^ d_Te1[(s1 >> 16) & 0xff]
        ^ d_Te2[(s2 >> 8) & 0xff] ^ d_Te3[s3 & 0xff]
        ^ d_rk[12];
    t1 = d_Te0[s1 >> 24] ^ d_Te1[(s2 >> 16) & 0xff]
        ^ d_Te2[(s3 >> 8) & 0xff] ^ d_Te3[s0 & 0xff]
        ^ d_rk[13];
    t2 = d_Te0[s2 >> 24] ^ d_Te1[(s3 >> 16) & 0xff]
        ^ d_Te2[(s0 >> 8) & 0xff] ^ d_Te3[s1 & 0xff]
        ^ d_rk[14];
    t3 = d_Te0[s3 >> 24] ^ d_Te1[(s0 >> 16) & 0xff]
        ^ d_Te2[(s1 >> 8) & 0xff] ^ d_Te3[s2 & 0xff]
        ^ d_rk[15];

    /* round 4: */
    s0 = d_Te0[t0 >> 24] ^ d_Te1[(t1 >> 16) & 0xff]
        ^ d_Te2[(t2 >> 8) & 0xff] ^ d_Te3[t3 & 0xff]
        ^ d_rk[16];
    s1 = d_Te0[t1 >> 24] ^ d_Te1[(t2 >> 16) & 0xff]
        ^ d_Te2[(t3 >> 8) & 0xff] ^ d_Te3[t0 & 0xff]
        ^ d_rk[17];
    s2 = d_Te0[t2 >> 24] ^ d_Te1[(t3 >> 16) & 0xff]
        ^ d_Te2[(t0 >> 8) & 0xff] ^ d_Te3[t1 & 0xff]
        ^ d_rk[18];
    s3 = d_Te0[t3 >> 24] ^ d_Te1[(t0 >> 16) & 0xff]
        ^ d_Te2[(t1 >> 8) & 0xff] ^ d_Te3[t2 & 0xff]
        ^ d_rk[19];

    /* round 5: */
    t0 = d_Te0[s0 >> 24] ^ d_Te1[(s1 >> 16) & 0xff]
        ^ d_Te2[(s2 >> 8) & 0xff] ^ d_Te3[s3 & 0xff]
        ^ d_rk[20];
    t1 = d_Te0[s1 >> 24] ^ d_Te1[(s2 >> 16) & 0xff]
        ^ d_Te2[(s3 >> 8) & 0xff] ^ d_Te3[s0 & 0xff]
        ^ d_rk[21];
    t2 = d_Te0[s2 >> 24] ^ d_Te1[(s3 >> 16) & 0xff]
        ^ d_Te2[(s0 >> 8) & 0xff] ^ d_Te3[s1 & 0xff]
        ^ d_rk[22];
    t3 = d_Te0[s3 >> 24] ^ d_Te1[(s0 >> 16) & 0xff]
        ^ d_Te2[(s1 >> 8) & 0xff] ^ d_Te3[s2 & 0xff]
        ^ d_rk[23];

    /* round 6: */
    s0 = d_Te0[t0 >> 24] ^ d_Te1[(t1 >> 16) & 0xff]
        ^ d_Te2[(t2 >> 8) & 0xff] ^ d_Te3[t3 & 0xff]
        ^ d_rk[24];
    s1 = d_Te0[t1 >> 24] ^ d_Te1[(t2 >> 16) & 0xff]
        ^ d_Te2[(t3 >> 8) & 0xff] ^ d_Te3[t0 & 0xff]
        ^ d_rk[25];
    s2 = d_Te0[t2 >> 24] ^ d_Te1[(t3 >> 16) & 0xff]
        ^ d_Te2[(t0 >> 8) & 0xff] ^ d_Te3[t1 & 0xff]
        ^ d_rk[26];
    s3 = d_Te0[t3 >> 24] ^ d_Te1[(t0 >> 16) & 0xff]
        ^ d_Te2[(t1 >> 8) & 0xff] ^ d_Te3[t2 & 0xff]
        ^ d_rk[27];

```

```
99      /* round 7: */
100     t0 = d_Te0[s0 >> 24] ^ d_Te1[(s1 >> 16) & 0xff]
101         ^ d_Te2[(s2 >> 8) & 0xff] ^ d_Te3[s3 & 0xff]
102         ^ d_rk[28];
103     t1 = d_Te0[s1 >> 24] ^ d_Te1[(s2 >> 16) & 0xff]
104         ^ d_Te2[(s3 >> 8) & 0xff] ^ d_Te3[s0 & 0xff]
105         ^ d_rk[29];
106     t2 = d_Te0[s2 >> 24] ^ d_Te1[(s3 >> 16) & 0xff]
107         ^ d_Te2[(s0 >> 8) & 0xff] ^ d_Te3[s1 & 0xff]
108         ^ d_rk[30];
109     t3 = d_Te0[s3 >> 24] ^ d_Te1[(s0 >> 16) & 0xff]
110         ^ d_Te2[(s1 >> 8) & 0xff] ^ d_Te3[s2 & 0xff]
111         ^ d_rk[31];
112
113     /* round 8: */
114     s0 = d_Te0[t0 >> 24] ^ d_Te1[(t1 >> 16) & 0xff]
115         ^ d_Te2[(t2 >> 8) & 0xff] ^ d_Te3[t3 & 0xff]
116         ^ d_rk[32];
117     s1 = d_Te0[t1 >> 24] ^ d_Te1[(t2 >> 16) & 0xff]
118         ^ d_Te2[(t3 >> 8) & 0xff] ^ d_Te3[t0 & 0xff]
119         ^ d_rk[33];
120     s2 = d_Te0[t2 >> 24] ^ d_Te1[(t3 >> 16) & 0xff]
121         ^ d_Te2[(t0 >> 8) & 0xff] ^ d_Te3[t1 & 0xff]
122         ^ d_rk[34];
123     s3 = d_Te0[t3 >> 24] ^ d_Te1[(t0 >> 16) & 0xff]
124         ^ d_Te2[(t1 >> 8) & 0xff] ^ d_Te3[t2 & 0xff]
125         ^ d_rk[35];
126
127     /* round 9: */
128     t0 = d_Te0[s0 >> 24] ^ d_Te1[(s1 >> 16) & 0xff]
129         ^ d_Te2[(s2 >> 8) & 0xff] ^ d_Te3[s3 & 0xff]
130         ^ d_rk[36];
131     t1 = d_Te0[s1 >> 24] ^ d_Te1[(s2 >> 16) & 0xff]
132         ^ d_Te2[(s3 >> 8) & 0xff] ^ d_Te3[s0 & 0xff]
133         ^ d_rk[37];
134     t2 = d_Te0[s2 >> 24] ^ d_Te1[(s3 >> 16) & 0xff]
135         ^ d_Te2[(s0 >> 8) & 0xff] ^ d_Te3[s1 & 0xff]
136         ^ d_rk[38];
137     t3 = d_Te0[s3 >> 24] ^ d_Te1[(s0 >> 16) & 0xff]
138         ^ d_Te2[(s1 >> 8) & 0xff] ^ d_Te3[s2 & 0xff]
139         ^ d_rk[39];
140
141     ct[4 * tid] = (d_Te4[(t0 >> 24)] & 0xff000000)
142         ^ (d_Te4[(t1 >> 16) & 0xff] & 0x00ff0000)
143         ^ (d_Te4[(t2 >> 8) & 0xff] & 0x0000ff00)
144         ^ (d_Te4[(t3) & 0xff] & 0x000000ff) ^ d_rk[40];
145     ct[4 * tid + 1] = (d_Te4[(t1 >> 24)] & 0xff000000)
146         ^ (d_Te4[(t2 >> 16) & 0xff] & 0x00ff0000)
147         ^ (d_Te4[(t3 >> 8) & 0xff] & 0x0000ff00)
148         ^ (d_Te4[(t0) & 0xff] & 0x000000ff) ^ d_rk[41];
149     ct[4 * tid + 2] = (d_Te4[(t2 >> 24)] & 0xff000000)
150         ^ (d_Te4[(t3 >> 16) & 0xff] & 0x00ff0000)
151         ^ (d_Te4[(t0 >> 8) & 0xff] & 0x0000ff00)
152         ^ (d_Te4[(t1) & 0xff] & 0x000000ff) ^ d_rk[42];
153     ct[4 * tid + 3] = (d_Te4[(t3 >> 24)] & 0xff000000)
154         ^ (d_Te4[(t0 >> 16) & 0xff] & 0x00ff0000)
155         ^ (d_Te4[(t1 >> 8) & 0xff] & 0x0000ff00)
156         ^ (d_Te4[(t2) & 0xff] & 0x000000ff) ^ d_rk[43];
157     tid += blockDim.x;
158 }
```

```
159   return;
160 }
161 void AES_Enc(unint * pt, unint * ct, unint * rk, unint nk, unint pt_size,
162     unint threads_per_block) {
163 #ifdef TEMPO
164   cudaEvent_t tstart, tstop;
165   HANDLE_ERROR(cudaEventCreate(&tstart));
166   HANDLE_ERROR(cudaEventCreate(&tstop));
167 
168   cudaEvent_t cstart, cstop;
169   HANDLE_ERROR(cudaEventCreate(&cstart));
170   HANDLE_ERROR(cudaEventCreate(&cstop));
171 
172   cudaEvent_t start, stop;
173   HANDLE_ERROR(cudaEventCreate(&start));
174   HANDLE_ERROR(cudaEventCreate(&stop));
175 
176   float elapsedTime;
177 #endif
178 
179   unint *d_pt, *d_ct;
180 
181   unint N = pt_size / 16;
182   unint nr = NR(nk);
183   unint nkx = 4 * (nr + 1);
184   unint nkxb = nkx * sizeof(unint);
185 
186 #ifdef TEMPO
187   HANDLE_ERROR(cudaEventRecord(tstart, 0));
188 #endif
189 
190   HANDLE_ERROR(cudaMalloc((void **)&d_pt, pt_size));
191   HANDLE_ERROR(cudaMalloc((void **)&d_ct, pt_size));
192 
193 #ifdef TEMPO
194   HANDLE_ERROR(cudaEventRecord(cstart, 0));
195 #endif
196 
197   HANDLE_ERROR(cudaMemcpy(d_pt, pt, pt_size, cudaMemcpyHostToDevice));
198   HANDLE_ERROR(cudaMemcpyToSymbol(d_rk, rk, nkxb));
199 
200 #ifdef TEMPO
201   HANDLE_ERROR(cudaEventRecord(start, 0));
202 #endif
203   switch (nk) {
204   case 4:
205     AES_Enc128 <<< 1, threads_per_block >>> (d_pt, d_ct,N);
206     break;
207   case 6:
208     AES_Enc192 <<< 1, threads_per_block >>> (d_pt, d_ct,N);
209     break;
210   case 8:
211     AES_Enc256 <<< 1, threads_per_block >>> (d_pt, d_ct,N);
212     break;
213   default:
214     return;
215   }
216 #ifdef TEMPO
217   HANDLE_ERROR(cudaEventRecord(stop, 0));
218 #endif
```

```
219
220   HANDLE_ERROR(cudaMemcpy(ct, d_ct, pt_size, cudaMemcpyDeviceToHost));
221
222 #ifdef TEMPO
223   HANDLE_ERROR(cudaEventRecord(cstop, 0));
224 #endif
225
226   HANDLE_ERROR(cudaFree(d_pt));
227   HANDLE_ERROR(cudaFree(d_ct));
228
229 #ifdef TEMPO
230   HANDLE_ERROR(cudaEventRecord(tstop, 0));
231   HANDLE_ERROR(cudaEventSynchronize(tstop));
232
233   HANDLE_ERROR(cudaEventElapsedTime(&elapsedTime, start, stop));
234   fprintf(stderr, "time(enc_cuda_%d):\t\t%12.6f ms\t%d threads\t%u bytes\n",
235       (4 * nk * 8), elapsedTime,threads_per_block,pt_size);
236
237   HANDLE_ERROR(cudaEventElapsedTime(&elapsedTime, cstart, cstop));
238   fprintf(stderr, "time(enc_cuda_cp_%d):\t%12.6f ms\t%d threads\t%u bytes\n",
239       (4 * nk * 8), elapsedTime,threads_per_block,pt_size);
240
241   HANDLE_ERROR(cudaEventElapsedTime(&elapsedTime, tstart, tstop));
242   fprintf(stderr, "time(enc_cuda_tt_%d):\t%12.6f ms\t%d threads\t%u bytes\n",
243       (4 * nk * 8), elapsedTime,threads_per_block,pt_size);
244
245   HANDLE_ERROR(cudaEventDestroy(start));
246   HANDLE_ERROR(cudaEventDestroy(stop));
247
248   HANDLE_ERROR(cudaEventDestroy(cstart));
249   HANDLE_ERROR(cudaEventDestroy(cstop));
250
251   HANDLE_ERROR(cudaEventDestroy(tstart));
252   HANDLE_ERROR(cudaEventDestroy(tstop));
253 #endif
254 }
```

II.8 cuda_aes_nb_nt_const_const

Código fonte II.18: cuda_aes_nb_nt_const_const/aes.cu

```
1  __constant__ __device__ unint d_rk[60];
2  void AES_Enc(unint * pt, unint * ct, unint * rk, unint nk, unint pt_size,
3      unint threads_per_block) {
4  #ifdef TEMPO
5    cudaEvent_t tstart, tstop;
6    HANDLE_ERROR(cudaEventCreate(&tstart));
7    HANDLE_ERROR(cudaEventCreate(&tstop));
8
9    cudaEvent_t cstart, cstop;
10   HANDLE_ERROR(cudaEventCreate(&cstart));
11   HANDLE_ERROR(cudaEventCreate(&cstop));
12
13   cudaEvent_t start, stop;
14   HANDLE_ERROR(cudaEventCreate(&start));
15   HANDLE_ERROR(cudaEventCreate(&stop));
16
```

```
17    float elapsedTime;
18  #endif
19    unint *d_pt, *d_ct;
20
21    unint N = pt_size / 16;
22    unint nr = NR(nk);
23    unint nkx = 4 * (nr + 1);
24    unint nkxb = nkx * sizeof(unint);
25
26    cudaDeviceProp prop;
27    HANDLE_ERROR( cudaGetDeviceProperties( &prop,0));
28
29    if (threads_per_block > prop.maxThreadsPerBlock)
30      threads_per_block = prop.maxThreadsPerBlock;
31
32    unint blocks = (unint) N / threads_per_block;
33
34    if (N % threads_per_block != 0)
35      blocks++;
36    unint blocks1 = blocks;
37    unint blocks2 = 1;
38
39    // Blocks transformado em bidimensional se ultrapassar maximo
40    unint maxgridsize = prop.maxGridSize[0];
41    if (blocks > maxgridsize) {
42      unint max = (unint) sqrt(blocks);
43      blocks1 = blocks;
44      blocks2 = 1;
45      unint i = 0;
46      // procura divisores de blocks
47      while (primo[i] <= max) {
48        if (blocks1 % primo[i] == 0) {
49          blocks1 = blocks1 / primo[i];
50          blocks2 = blocks2 * primo[i];
51          if (blocks1 > maxgridsize)
52            continue;
53          break;
54        }
55        i++;
56      }
57      //se nao encontrar divisores tentar minimizar blocks1 e blocks2
58      if ((blocks1 > maxgridsize) || (blocks2 > maxgridsize)) {
59        blocks1 = max;
60        blocks2 = max;
61        while (blocks1 * blocks2 * threads_per_block < N)
62          blocks2++;
63      }
64    }
65    dim3 BLOCKS(blocks1, blocks2, 1);
66
67  #ifdef TEMPO
68    HANDLE_ERROR(cudaEventRecord(tstart, 0));
69  #endif
70    HANDLE_ERROR(cudaMalloc((void **)&d_pt, pt_size));
71    HANDLE_ERROR(cudaMalloc((void **)&d_ct, pt_size));
72  #ifdef TEMPO
73    HANDLE_ERROR(cudaEventRecord(cstart, 0));
74  #endif
75    HANDLE_ERROR(cudaMemcpy(d_pt, pt, pt_size, cudaMemcpyHostToDevice));
76    HANDLE_ERROR(cudaMemcpyToSymbol(d_rk, rk, nkxb));
```

```
77
78 #ifdef TEMPO
79   HANDLE_ERROR(cudaEventRecord(start, 0));
80 #endif
81   switch (nk) {
82   case 4:
83     AES_Enc128 <<< BLOCKS, threads_per_block >>> (d_pt, d_ct,N);
84     break;
85   case 6:
86     AES_Enc192 <<< BLOCKS, threads_per_block >>> (d_pt, d_ct,N);
87     break;
88   case 8:
89     AES_Enc256 <<< BLOCKS, threads_per_block >>> (d_pt, d_ct,N);
90     break;
91   default:
92     return;
93   }
94 #ifdef TEMPO
95   HANDLE_ERROR(cudaEventRecord(stop, 0));
96 #endif
97
98   HANDLE_ERROR(cudaMemcpy(ct, d_ct, pt_size, cudaMemcpyDeviceToHost));
99 #ifdef TEMPO
100   HANDLE_ERROR(cudaEventRecord(cstop, 0));
101 #endif
102
103   HANDLE_ERROR(cudaFree(d_pt));
104   HANDLE_ERROR(cudaFree(d_ct));
105
106 #ifdef TEMPO
107   HANDLE_ERROR(cudaEventRecord(tstop, 0));
108   HANDLE_ERROR(cudaEventSynchronize(tstop));
109
110   HANDLE_ERROR(cudaEventElapsedTime(&elapsedTime, start, stop));
111   fprintf(stderr, "time(enc_cuda_%d):\t\t%12.6f ms\t%4d threads\t%u bytes\n",
112       (4 * nk * 8), elapsedTime, threads_per_block,pt_size);
113
114   HANDLE_ERROR(cudaEventElapsedTime(&elapsedTime, cstart, cstop));
115   fprintf(stderr, "time(enc_cuda_cp_%d):\t%12.6f ms\t%4d threads\t%u bytes\n",
116       (4 * nk * 8), elapsedTime, threads_per_block,pt_size);
117
118   HANDLE_ERROR(cudaEventElapsedTime(&elapsedTime, tstart, tstop));
119   fprintf(stderr, "time(enc_cuda_tt_%d):\t%12.6f ms\t%4d threads\t%u bytes\n",
120       (4 * nk * 8), elapsedTime, threads_per_block,pt_size);
121
122   HANDLE_ERROR(cudaEventDestroy(start));
123   HANDLE_ERROR(cudaEventDestroy(stop));
124
125   HANDLE_ERROR(cudaEventDestroy(cstart));
126   HANDLE_ERROR(cudaEventDestroy(cstop));
127
128   HANDLE_ERROR(cudaEventDestroy(tstart));
129   HANDLE_ERROR(cudaEventDestroy(tstop));
130 #endif
131 }
132 __global__ void AES_Enc128(unint * pt, unint * ct, unint N) {
133
134   unint tid = blockIdx.y * gridDim.x * blockDim.x + blockIdx.x * blockDim.x
135       + threadIdx.x;
136
```

```
137   if (tid > (N - 1))
138     return;
139
140   unint s0, s1, s2, s3, t0, t1, t2, t3;
141
142   s0 = pt[4 * tid] ^ d_rk[0];
143   s1 = pt[4 * tid + 1] ^ d_rk[1];
144   s2 = pt[4 * tid + 2] ^ d_rk[2];
145   s3 = pt[4 * tid + 3] ^ d_rk[3];
146
147   /* round 1: */
148   t0 = d_Te0[s0 >> 24] ^ d_Te1[(s1 >> 16) & 0xff] ^ d_Te2[(s2 >> 8) & 0xff]
149       ^ d_Te3[s3 & 0xff] ^ d_rk[4];
150   t1 = d_Te0[s1 >> 24] ^ d_Te1[(s2 >> 16) & 0xff] ^ d_Te2[(s3 >> 8) & 0xff]
151       ^ d_Te3[s0 & 0xff] ^ d_rk[5];
152   t2 = d_Te0[s2 >> 24] ^ d_Te1[(s3 >> 16) & 0xff] ^ d_Te2[(s0 >> 8) & 0xff]
153       ^ d_Te3[s1 & 0xff] ^ d_rk[6];
154   t3 = d_Te0[s3 >> 24] ^ d_Te1[(s0 >> 16) & 0xff] ^ d_Te2[(s1 >> 8) & 0xff]
155       ^ d_Te3[s2 & 0xff] ^ d_rk[7];
156
157   /* round 2: */
158   s0 = d_Te0[t0 >> 24] ^ d_Te1[(t1 >> 16) & 0xff] ^ d_Te2[(t2 >> 8) & 0xff]
159       ^ d_Te3[t3 & 0xff] ^ d_rk[8];
160   s1 = d_Te0[t1 >> 24] ^ d_Te1[(t2 >> 16) & 0xff] ^ d_Te2[(t3 >> 8) & 0xff]
161       ^ d_Te3[t0 & 0xff] ^ d_rk[9];
162   s2 = d_Te0[t2 >> 24] ^ d_Te1[(t3 >> 16) & 0xff] ^ d_Te2[(t0 >> 8) & 0xff]
163       ^ d_Te3[t1 & 0xff] ^ d_rk[10];
164   s3 = d_Te0[t3 >> 24] ^ d_Te1[(t0 >> 16) & 0xff] ^ d_Te2[(t1 >> 8) & 0xff]
165       ^ d_Te3[t2 & 0xff] ^ d_rk[11];
166
167   /* round 3: */
168   t0 = d_Te0[s0 >> 24] ^ d_Te1[(s1 >> 16) & 0xff] ^ d_Te2[(s2 >> 8) & 0xff]
169       ^ d_Te3[s3 & 0xff] ^ d_rk[12];
170   t1 = d_Te0[s1 >> 24] ^ d_Te1[(s2 >> 16) & 0xff] ^ d_Te2[(s3 >> 8) & 0xff]
171       ^ d_Te3[s0 & 0xff] ^ d_rk[13];
172   t2 = d_Te0[s2 >> 24] ^ d_Te1[(s3 >> 16) & 0xff] ^ d_Te2[(s0 >> 8) & 0xff]
173       ^ d_Te3[s1 & 0xff] ^ d_rk[14];
174   t3 = d_Te0[s3 >> 24] ^ d_Te1[(s0 >> 16) & 0xff] ^ d_Te2[(s1 >> 8) & 0xff]
175       ^ d_Te3[s2 & 0xff] ^ d_rk[15];
176
177   /* round 4: */
178   s0 = d_Te0[t0 >> 24] ^ d_Te1[(t1 >> 16) & 0xff] ^ d_Te2[(t2 >> 8) & 0xff]
179       ^ d_Te3[t3 & 0xff] ^ d_rk[16];
180   s1 = d_Te0[t1 >> 24] ^ d_Te1[(t2 >> 16) & 0xff] ^ d_Te2[(t3 >> 8) & 0xff]
181       ^ d_Te3[t0 & 0xff] ^ d_rk[17];
182   s2 = d_Te0[t2 >> 24] ^ d_Te1[(t3 >> 16) & 0xff] ^ d_Te2[(t0 >> 8) & 0xff]
183       ^ d_Te3[t1 & 0xff] ^ d_rk[18];
184   s3 = d_Te0[t3 >> 24] ^ d_Te1[(t0 >> 16) & 0xff] ^ d_Te2[(t1 >> 8) & 0xff]
185       ^ d_Te3[t2 & 0xff] ^ d_rk[19];
186
187   /* round 5: */
188   t0 = d_Te0[s0 >> 24] ^ d_Te1[(s1 >> 16) & 0xff] ^ d_Te2[(s2 >> 8) & 0xff]
189       ^ d_Te3[s3 & 0xff] ^ d_rk[20];
190   t1 = d_Te0[s1 >> 24] ^ d_Te1[(s2 >> 16) & 0xff] ^ d_Te2[(s3 >> 8) & 0xff]
191       ^ d_Te3[s0 & 0xff] ^ d_rk[21];
192   t2 = d_Te0[s2 >> 24] ^ d_Te1[(s3 >> 16) & 0xff] ^ d_Te2[(s0 >> 8) & 0xff]
193       ^ d_Te3[s1 & 0xff] ^ d_rk[22];
194   t3 = d_Te0[s3 >> 24] ^ d_Te1[(s0 >> 16) & 0xff] ^ d_Te2[(s1 >> 8) & 0xff]
195       ^ d_Te3[s2 & 0xff] ^ d_rk[23];
196
```

```
197     /* round 6: */
198     s0 = d_Te0[t0 >> 24] ^ d_Te1[(t1 >> 16) & 0xff] ^ d_Te2[(t2 >> 8) & 0xff]
199         ^ d_Te3[t3 & 0xff] ^ d_rk[24];
200     s1 = d_Te0[t1 >> 24] ^ d_Te1[(t2 >> 16) & 0xff] ^ d_Te2[(t3 >> 8) & 0xff]
201         ^ d_Te3[t0 & 0xff] ^ d_rk[25];
202     s2 = d_Te0[t2 >> 24] ^ d_Te1[(t3 >> 16) & 0xff] ^ d_Te2[(t0 >> 8) & 0xff]
203         ^ d_Te3[t1 & 0xff] ^ d_rk[26];
204     s3 = d_Te0[t3 >> 24] ^ d_Te1[(t0 >> 16) & 0xff] ^ d_Te2[(t1 >> 8) & 0xff]
205         ^ d_Te3[t2 & 0xff] ^ d_rk[27];
206 
207     /* round 7: */
208     t0 = d_Te0[s0 >> 24] ^ d_Te1[(s1 >> 16) & 0xff] ^ d_Te2[(s2 >> 8) & 0xff]
209         ^ d_Te3[s3 & 0xff] ^ d_rk[28];
210     t1 = d_Te0[s1 >> 24] ^ d_Te1[(s2 >> 16) & 0xff] ^ d_Te2[(s3 >> 8) & 0xff]
211         ^ d_Te3[s0 & 0xff] ^ d_rk[29];
212     t2 = d_Te0[s2 >> 24] ^ d_Te1[(s3 >> 16) & 0xff] ^ d_Te2[(s0 >> 8) & 0xff]
213         ^ d_Te3[s1 & 0xff] ^ d_rk[30];
214     t3 = d_Te0[s3 >> 24] ^ d_Te1[(s0 >> 16) & 0xff] ^ d_Te2[(s1 >> 8) & 0xff]
215         ^ d_Te3[s2 & 0xff] ^ d_rk[31];
216 
217     /* round 8: */
218     s0 = d_Te0[t0 >> 24] ^ d_Te1[(t1 >> 16) & 0xff] ^ d_Te2[(t2 >> 8) & 0xff]
219         ^ d_Te3[t3 & 0xff] ^ d_rk[32];
220     s1 = d_Te0[t1 >> 24] ^ d_Te1[(t2 >> 16) & 0xff] ^ d_Te2[(t3 >> 8) & 0xff]
221         ^ d_Te3[t0 & 0xff] ^ d_rk[33];
222     s2 = d_Te0[t2 >> 24] ^ d_Te1[(t3 >> 16) & 0xff] ^ d_Te2[(t0 >> 8) & 0xff]
223         ^ d_Te3[t1 & 0xff] ^ d_rk[34];
224     s3 = d_Te0[t3 >> 24] ^ d_Te1[(t0 >> 16) & 0xff] ^ d_Te2[(t1 >> 8) & 0xff]
225         ^ d_Te3[t2 & 0xff] ^ d_rk[35];
226 
227     /* round 9: */
228     t0 = d_Te0[s0 >> 24] ^ d_Te1[(s1 >> 16) & 0xff] ^ d_Te2[(s2 >> 8) & 0xff]
229         ^ d_Te3[s3 & 0xff] ^ d_rk[36];
230     t1 = d_Te0[s1 >> 24] ^ d_Te1[(s2 >> 16) & 0xff] ^ d_Te2[(s3 >> 8) & 0xff]
231         ^ d_Te3[s0 & 0xff] ^ d_rk[37];
232     t2 = d_Te0[s2 >> 24] ^ d_Te1[(s3 >> 16) & 0xff] ^ d_Te2[(s0 >> 8) & 0xff]
233         ^ d_Te3[s1 & 0xff] ^ d_rk[38];
234     t3 = d_Te0[s3 >> 24] ^ d_Te1[(s0 >> 16) & 0xff] ^ d_Te2[(s1 >> 8) & 0xff]
235         ^ d_Te3[s2 & 0xff] ^ d_rk[39];
236 
237     ct[4 * tid] = (d_Te4[(t0 >> 24)] & 0xff000000)
238         ^ (d_Te4[(t1 >> 16) & 0xff] & 0x00ff0000)
239         ^ (d_Te4[(t2 >> 8) & 0xff] & 0x0000ff00)
240         ^ (d_Te4[(t3) & 0xff] & 0x000000ff) ^ d_rk[40];
241     ct[4 * tid + 1] = (d_Te4[(t1 >> 24)] & 0xff000000)
242         ^ (d_Te4[(t2 >> 16) & 0xff] & 0x00ff0000)
243         ^ (d_Te4[(t3 >> 8) & 0xff] & 0x0000ff00)
244         ^ (d_Te4[(t0) & 0xff] & 0x000000ff) ^ d_rk[41];
245     ct[4 * tid + 2] = (d_Te4[(t2 >> 24)] & 0xff000000)
246         ^ (d_Te4[(t3 >> 16) & 0xff] & 0x00ff0000)
247         ^ (d_Te4[(t0 >> 8) & 0xff] & 0x0000ff00)
248         ^ (d_Te4[(t1) & 0xff] & 0x000000ff) ^ d_rk[42];
249     ct[4 * tid + 3] = (d_Te4[(t3 >> 24)] & 0xff000000)
250         ^ (d_Te4[(t0 >> 16) & 0xff] & 0x00ff0000)
251         ^ (d_Te4[(t1 >> 8) & 0xff] & 0x0000ff00)
252         ^ (d_Te4[(t2) & 0xff] & 0x000000ff) ^ d_rk[43];
253 
254     return;
255 }
```

II.9 cuda_aes_1b_nt_const_texture

Código fonte II.19: cuda_aes_1b_nt_const_texture/aes.cu

```
texture<unsigned, 1, cudaReadModeElementType> rk_tex;
__global__ void AES_Enc128(unint * pt, unint * ct, unint N) {

  unint tid = blockIdx.x * blockDim.x + threadIdx.x;

  unint s0, s1, s2, s3, t0, t1, t2, t3;

  while (tid < N) {

    s0 = pt[4 * tid] ^ tex1Dfetch(rk_tex, 0);
    s1 = pt[4 * tid + 1] ^ tex1Dfetch(rk_tex, 1);
    s2 = pt[4 * tid + 2] ^ tex1Dfetch(rk_tex, 2);
    s3 = pt[4 * tid + 3] ^ tex1Dfetch(rk_tex, 3);

    /* round 1: */
    t0 = d_Te0[s0 >> 24] ^ d_Te1[(s1 >> 16) & 0xff]
        ^ d_Te2[(s2 >> 8) & 0xff] ^ d_Te3[s3 & 0xff]
        ^ tex1Dfetch(rk_tex, 4);
    t1 = d_Te0[s1 >> 24] ^ d_Te1[(s2 >> 16) & 0xff]
        ^ d_Te2[(s3 >> 8) & 0xff] ^ d_Te3[s0 & 0xff]
        ^ tex1Dfetch(rk_tex, 5);
    t2 = d_Te0[s2 >> 24] ^ d_Te1[(s3 >> 16) & 0xff]
        ^ d_Te2[(s0 >> 8) & 0xff] ^ d_Te3[s1 & 0xff]
        ^ tex1Dfetch(rk_tex, 6);
    t3 = d_Te0[s3 >> 24] ^ d_Te1[(s0 >> 16) & 0xff]
        ^ d_Te2[(s1 >> 8) & 0xff] ^ d_Te3[s2 & 0xff]
        ^ tex1Dfetch(rk_tex, 7);

    /* round 2: */
    s0 = d_Te0[t0 >> 24] ^ d_Te1[(t1 >> 16) & 0xff]
        ^ d_Te2[(t2 >> 8) & 0xff] ^ d_Te3[t3 & 0xff]
        ^ tex1Dfetch(rk_tex, 8);
    s1 = d_Te0[t1 >> 24] ^ d_Te1[(t2 >> 16) & 0xff]
        ^ d_Te2[(t3 >> 8) & 0xff] ^ d_Te3[t0 & 0xff]
        ^ tex1Dfetch(rk_tex, 9);
    s2 = d_Te0[t2 >> 24] ^ d_Te1[(t3 >> 16) & 0xff]
        ^ d_Te2[(t0 >> 8) & 0xff] ^ d_Te3[t1 & 0xff]
        ^ tex1Dfetch(rk_tex, 10);
    s3 = d_Te0[t3 >> 24] ^ d_Te1[(t0 >> 16) & 0xff]
        ^ d_Te2[(t1 >> 8) & 0xff] ^ d_Te3[t2 & 0xff]
        ^ tex1Dfetch(rk_tex, 11);

    /* round 3: */
    t0 = d_Te0[s0 >> 24] ^ d_Te1[(s1 >> 16) & 0xff]
        ^ d_Te2[(s2 >> 8) & 0xff] ^ d_Te3[s3 & 0xff]
        ^ tex1Dfetch(rk_tex, 12);
    t1 = d_Te0[s1 >> 24] ^ d_Te1[(s2 >> 16) & 0xff]
        ^ d_Te2[(s3 >> 8) & 0xff] ^ d_Te3[s0 & 0xff]
        ^ tex1Dfetch(rk_tex, 13);
    t2 = d_Te0[s2 >> 24] ^ d_Te1[(s3 >> 16) & 0xff]
        ^ d_Te2[(s0 >> 8) & 0xff] ^ d_Te3[s1 & 0xff]
        ^ tex1Dfetch(rk_tex, 14);
    t3 = d_Te0[s3 >> 24] ^ d_Te1[(s0 >> 16) & 0xff]
        ^ d_Te2[(s1 >> 8) & 0xff] ^ d_Te3[s2 & 0xff]
        ^ tex1Dfetch(rk_tex, 15);

```

```
57      /* round 4: */
58      s0 = d_Te0[t0 >> 24] ^ d_Te1[(t1 >> 16) & 0xff]
59          ^ d_Te2[(t2 >> 8) & 0xff] ^ d_Te3[t3 & 0xff]
60          ^ tex1Dfetch(rk_tex, 16);
61      s1 = d_Te0[t1 >> 24] ^ d_Te1[(t2 >> 16) & 0xff]
62          ^ d_Te2[(t3 >> 8) & 0xff] ^ d_Te3[t0 & 0xff]
63          ^ tex1Dfetch(rk_tex, 17);
64      s2 = d_Te0[t2 >> 24] ^ d_Te1[(t3 >> 16) & 0xff]
65          ^ d_Te2[(t0 >> 8) & 0xff] ^ d_Te3[t1 & 0xff]
66          ^ tex1Dfetch(rk_tex, 18);
67      s3 = d_Te0[t3 >> 24] ^ d_Te1[(t0 >> 16) & 0xff]
68          ^ d_Te2[(t1 >> 8) & 0xff] ^ d_Te3[t2 & 0xff]
69          ^ tex1Dfetch(rk_tex, 19);
70
71      /* round 5: */
72      t0 = d_Te0[s0 >> 24] ^ d_Te1[(s1 >> 16) & 0xff]
73          ^ d_Te2[(s2 >> 8) & 0xff] ^ d_Te3[s3 & 0xff]
74          ^ tex1Dfetch(rk_tex, 20);
75      t1 = d_Te0[s1 >> 24] ^ d_Te1[(s2 >> 16) & 0xff]
76          ^ d_Te2[(s3 >> 8) & 0xff] ^ d_Te3[s0 & 0xff]
77          ^ tex1Dfetch(rk_tex, 21);
78      t2 = d_Te0[s2 >> 24] ^ d_Te1[(s3 >> 16) & 0xff]
79          ^ d_Te2[(s0 >> 8) & 0xff] ^ d_Te3[s1 & 0xff]
80          ^ tex1Dfetch(rk_tex, 22);
81      t3 = d_Te0[s3 >> 24] ^ d_Te1[(s0 >> 16) & 0xff]
82          ^ d_Te2[(s1 >> 8) & 0xff] ^ d_Te3[s2 & 0xff]
83          ^ tex1Dfetch(rk_tex, 23);
84
85      /* round 6: */
86      s0 = d_Te0[t0 >> 24] ^ d_Te1[(t1 >> 16) & 0xff]
87          ^ d_Te2[(t2 >> 8) & 0xff] ^ d_Te3[t3 & 0xff]
88          ^ tex1Dfetch(rk_tex, 24);
89      s1 = d_Te0[t1 >> 24] ^ d_Te1[(t2 >> 16) & 0xff]
90          ^ d_Te2[(t3 >> 8) & 0xff] ^ d_Te3[t0 & 0xff]
91          ^ tex1Dfetch(rk_tex, 25);
92      s2 = d_Te0[t2 >> 24] ^ d_Te1[(t3 >> 16) & 0xff]
93          ^ d_Te2[(t0 >> 8) & 0xff] ^ d_Te3[t1 & 0xff]
94          ^ tex1Dfetch(rk_tex, 26);
95      s3 = d_Te0[t3 >> 24] ^ d_Te1[(t0 >> 16) & 0xff]
96          ^ d_Te2[(t1 >> 8) & 0xff] ^ d_Te3[t2 & 0xff]
97          ^ tex1Dfetch(rk_tex, 27);
98
99      /* round 7: */
100     t0 = d_Te0[s0 >> 24] ^ d_Te1[(s1 >> 16) & 0xff]
101         ^ d_Te2[(s2 >> 8) & 0xff] ^ d_Te3[s3 & 0xff]
102         ^ tex1Dfetch(rk_tex, 28);
103     t1 = d_Te0[s1 >> 24] ^ d_Te1[(s2 >> 16) & 0xff]
104         ^ d_Te2[(s3 >> 8) & 0xff] ^ d_Te3[s0 & 0xff]
105         ^ tex1Dfetch(rk_tex, 29);
106     t2 = d_Te0[s2 >> 24] ^ d_Te1[(s3 >> 16) & 0xff]
107         ^ d_Te2[(s0 >> 8) & 0xff] ^ d_Te3[s1 & 0xff]
108         ^ tex1Dfetch(rk_tex, 30);
109     t3 = d_Te0[s3 >> 24] ^ d_Te1[(s0 >> 16) & 0xff]
110         ^ d_Te2[(s1 >> 8) & 0xff] ^ d_Te3[s2 & 0xff]
111         ^ tex1Dfetch(rk_tex, 31);
112
113     /* round 8: */
114     s0 = d_Te0[t0 >> 24] ^ d_Te1[(t1 >> 16) & 0xff]
115         ^ d_Te2[(t2 >> 8) & 0xff] ^ d_Te3[t3 & 0xff]
116         ^ tex1Dfetch(rk_tex, 32);
```

```
117     s1 = d_Te0[t1 >> 24] ^ d_Te1[(t2 >> 16) & 0xff]
118         ^ d_Te2[(t3 >> 8) & 0xff] ^ d_Te3[t0 & 0xff]
119         ^ tex1Dfetch(rk_tex, 33);
120     s2 = d_Te0[t2 >> 24] ^ d_Te1[(t3 >> 16) & 0xff]
121         ^ d_Te2[(t0 >> 8) & 0xff] ^ d_Te3[t1 & 0xff]
122         ^ tex1Dfetch(rk_tex, 34);
123     s3 = d_Te0[t3 >> 24] ^ d_Te1[(t0 >> 16) & 0xff]
124         ^ d_Te2[(t1 >> 8) & 0xff] ^ d_Te3[t2 & 0xff]
125         ^ tex1Dfetch(rk_tex, 35);
126
127     /* round 9: */
128     t0 = d_Te0[s0 >> 24] ^ d_Te1[(s1 >> 16) & 0xff]
129         ^ d_Te2[(s2 >> 8) & 0xff] ^ d_Te3[s3 & 0xff]
130         ^ tex1Dfetch(rk_tex, 36);
131     t1 = d_Te0[s1 >> 24] ^ d_Te1[(s2 >> 16) & 0xff]
132         ^ d_Te2[(s3 >> 8) & 0xff] ^ d_Te3[s0 & 0xff]
133         ^ tex1Dfetch(rk_tex, 37);
134     t2 = d_Te0[s2 >> 24] ^ d_Te1[(s3 >> 16) & 0xff]
135         ^ d_Te2[(s0 >> 8) & 0xff] ^ d_Te3[s1 & 0xff]
136         ^ tex1Dfetch(rk_tex, 38);
137     t3 = d_Te0[s3 >> 24] ^ d_Te1[(s0 >> 16) & 0xff]
138         ^ d_Te2[(s1 >> 8) & 0xff] ^ d_Te3[s2 & 0xff]
139         ^ tex1Dfetch(rk_tex, 39);
140
141     ct[4 * tid] = (d_Te4[(t0 >> 24)] & 0xff000000)
142         ^ (d_Te4[(t1 >> 16) & 0xff] & 0x00ff0000)
143         ^ (d_Te4[(t2 >> 8) & 0xff] & 0x0000ff00)
144         ^ (d_Te4[(t3) & 0xff] & 0x000000ff) ^ tex1Dfetch(rk_tex, 40);
145     ct[4 * tid + 1] = (d_Te4[(t1 >> 24)] & 0xff000000)
146         ^ (d_Te4[(t2 >> 16) & 0xff] & 0x00ff0000)
147         ^ (d_Te4[(t3 >> 8) & 0xff] & 0x0000ff00)
148         ^ (d_Te4[(t0) & 0xff] & 0x000000ff) ^ tex1Dfetch(rk_tex, 41);
149     ct[4 * tid + 2] = (d_Te4[(t2 >> 24)] & 0xff000000)
150         ^ (d_Te4[(t3 >> 16) & 0xff] & 0x00ff0000)
151         ^ (d_Te4[(t0 >> 8) & 0xff] & 0x0000ff00)
152         ^ (d_Te4[(t1) & 0xff] & 0x000000ff) ^ tex1Dfetch(rk_tex, 42);
153     ct[4 * tid + 3] = (d_Te4[(t3 >> 24)] & 0xff000000)
154         ^ (d_Te4[(t0 >> 16) & 0xff] & 0x00ff0000)
155         ^ (d_Te4[(t1 >> 8) & 0xff] & 0x0000ff00)
156         ^ (d_Te4[(t2) & 0xff] & 0x000000ff) ^ tex1Dfetch(rk_tex, 43);
157     tid += blockDim.x;
158   }
159   return;
160 }
161 void AES_Enc(unint * pt, unint * ct, unint * rk, unint nk, unint pt_size,
162     unint threads_per_block) {
163 #ifdef TEMPO
164   cudaEvent_t tstart, tstop;
165   HANDLE_ERROR(cudaEventCreate(&tstart));
166   HANDLE_ERROR(cudaEventCreate(&tstop));
167
168   cudaEvent_t cstart, cstop;
169   HANDLE_ERROR(cudaEventCreate(&cstart));
170   HANDLE_ERROR(cudaEventCreate(&cstop));
171
172   cudaEvent_t start, stop;
173   HANDLE_ERROR(cudaEventCreate(&start));
174   HANDLE_ERROR(cudaEventCreate(&stop));
175
176   float elapsedTime;
```

```
177 #endif
178
179   unint *d_pt, *d_ct, *d_rk;
180
181   unint N = pt_size / 16;
182   unint nr = NR(nk);
183   unint nkx = 4 * (nr + 1);
184   unint nkxb = nkx * sizeof(unint);
185
186 #ifdef TEMPO
187   HANDLE_ERROR(cudaEventRecord(tstart, 0));
188 #endif
189
190   HANDLE_ERROR(cudaMalloc((void **)&d_pt, pt_size));
191   HANDLE_ERROR(cudaMalloc((void **)&d_ct, pt_size));
192   HANDLE_ERROR(cudaMalloc((void **)&d_rk, nkxb));
193
194 #ifdef TEMPO
195   HANDLE_ERROR(cudaEventRecord(cstart, 0));
196 #endif
197
198   HANDLE_ERROR(cudaMemcpy(d_rk, rk, nkxb, cudaMemcpyHostToDevice));
199   HANDLE_ERROR(cudaMemcpy(d_pt, pt, pt_size, cudaMemcpyHostToDevice));
200   HANDLE_ERROR(cudaBindTexture(NULL, rk_tex, d_rk,nkxb));
201
202 #ifdef TEMPO
203   HANDLE_ERROR(cudaEventRecord(start, 0));
204 #endif
205   switch (nk) {
206   case 4:
207     AES_Enc128 <<< 1, threads_per_block >>> (d_pt, d_ct,N);
208     break;
209   case 6:
210     AES_Enc192 <<< 1, threads_per_block >>> (d_pt, d_ct,N);
211     break;
212   case 8:
213     AES_Enc256 <<< 1, threads_per_block >>> (d_pt, d_ct,N);
214     break;
215   default:
216     return;
217   }
218 #ifdef TEMPO
219   HANDLE_ERROR(cudaEventRecord(stop, 0));
220 #endif
221
222   HANDLE_ERROR(cudaMemcpy(ct, d_ct, pt_size, cudaMemcpyDeviceToHost));
223
224 #ifdef TEMPO
225   HANDLE_ERROR(cudaEventRecord(cstop, 0));
226 #endif
227
228   HANDLE_ERROR(cudaUnbindTexture(rk_tex));
229   HANDLE_ERROR(cudaFree(d_rk));
230
231   HANDLE_ERROR(cudaFree(d_pt));
232   HANDLE_ERROR(cudaFree(d_ct));
233
234
235 #ifdef TEMPO
236   HANDLE_ERROR(cudaEventRecord(tstop, 0));
```

```
237   HANDLE_ERROR(cudaEventSynchronize(tstop));
238
239   HANDLE_ERROR(cudaEventElapsedTime(&elapsedTime, start, stop));
240   fprintf(stderr, "time(enc_cuda_%d):\t\t%12.6f ms\t%d threads\t%u bytes\n",
241       (4 * nk * 8), elapsedTime,threads_per_block,pt_size);
242
243   HANDLE_ERROR(cudaEventElapsedTime(&elapsedTime, cstart, cstop));
244   fprintf(stderr, "time(enc_cuda_cp_%d):\t%12.6f ms\t%d threads\t%u bytes\n",
245       (4 * nk * 8), elapsedTime,threads_per_block,pt_size);
246
247   HANDLE_ERROR(cudaEventElapsedTime(&elapsedTime, tstart, tstop));
248   fprintf(stderr, "time(enc_cuda_tt_%d):\t%12.6f ms\t%d threads\t%u bytes\n",
249       (4 * nk * 8), elapsedTime,threads_per_block,pt_size);
250
251   HANDLE_ERROR(cudaEventDestroy(start));
252   HANDLE_ERROR(cudaEventDestroy(stop));
253
254   HANDLE_ERROR(cudaEventDestroy(cstart));
255   HANDLE_ERROR(cudaEventDestroy(cstop));
256
257   HANDLE_ERROR(cudaEventDestroy(tstart));
258   HANDLE_ERROR(cudaEventDestroy(tstop));
259 #endif
260 }
```

II.10 cuda_aes_nb_nt_const_texture

Código fonte II.20: cuda_aes_nb_nt_const_texture/aes.cu

```
1  texture<unsigned, 1, cudaReadModeElementType> rk_tex;
2  __global__ void AES_Enc128(unint * pt, unint * ct, unint N) {
3
4    unint tid = blockIdx.y * gridDim.x * blockDim.x + blockIdx.x * blockDim.x
5        + threadIdx.x;
6
7    if (tid > (N - 1))
8      return;
9
10   unint s0, s1, s2, s3, t0, t1, t2, t3;
11
12   s0 = pt[4 * tid] ^ tex1Dfetch(rk_tex, 0);
13   s1 = pt[4 * tid + 1] ^ tex1Dfetch(rk_tex, 1);
14   s2 = pt[4 * tid + 2] ^ tex1Dfetch(rk_tex, 2);
15   s3 = pt[4 * tid + 3] ^ tex1Dfetch(rk_tex, 3);
16
17   /* round 1: */
18   t0 = d_Te0[s0 >> 24] ^ d_Te1[(s1 >> 16) & 0xff] ^ d_Te2[(s2 >> 8) & 0xff]
19       ^ d_Te3[s3 & 0xff] ^ tex1Dfetch(rk_tex, 4);
20   t1 = d_Te0[s1 >> 24] ^ d_Te1[(s2 >> 16) & 0xff] ^ d_Te2[(s3 >> 8) & 0xff]
21       ^ d_Te3[s0 & 0xff] ^ tex1Dfetch(rk_tex, 5);
22   t2 = d_Te0[s2 >> 24] ^ d_Te1[(s3 >> 16) & 0xff] ^ d_Te2[(s0 >> 8) & 0xff]
23       ^ d_Te3[s1 & 0xff] ^ tex1Dfetch(rk_tex, 6);
24   t3 = d_Te0[s3 >> 24] ^ d_Te1[(s0 >> 16) & 0xff] ^ d_Te2[(s1 >> 8) & 0xff]
25       ^ d_Te3[s2 & 0xff] ^ tex1Dfetch(rk_tex, 7);
26
27   /* round 2: */
28   s0 = d_Te0[t0 >> 24] ^ d_Te1[(t1 >> 16) & 0xff] ^ d_Te2[(t2 >> 8) & 0xff]
```

```
29          ^ d_Te3[t3 & 0xff] ^ tex1Dfetch(rk_tex, 8);
30      s1 = d_Te0[t1 >> 24] ^ d_Te1[(t2 >> 16) & 0xff] ^ d_Te2[(t3 >> 8) & 0xff]
31          ^ d_Te3[t0 & 0xff] ^ tex1Dfetch(rk_tex, 9);
32      s2 = d_Te0[t2 >> 24] ^ d_Te1[(t3 >> 16) & 0xff] ^ d_Te2[(t0 >> 8) & 0xff]
33          ^ d_Te3[t1 & 0xff] ^ tex1Dfetch(rk_tex, 10);
34      s3 = d_Te0[t3 >> 24] ^ d_Te1[(t0 >> 16) & 0xff] ^ d_Te2[(t1 >> 8) & 0xff]
35          ^ d_Te3[t2 & 0xff] ^ tex1Dfetch(rk_tex, 11);
36
37      /* round 3: */
38      t0 = d_Te0[s0 >> 24] ^ d_Te1[(s1 >> 16) & 0xff] ^ d_Te2[(s2 >> 8) & 0xff]
39          ^ d_Te3[s3 & 0xff] ^ tex1Dfetch(rk_tex, 12);
40      t1 = d_Te0[s1 >> 24] ^ d_Te1[(s2 >> 16) & 0xff] ^ d_Te2[(s3 >> 8) & 0xff]
41          ^ d_Te3[s0 & 0xff] ^ tex1Dfetch(rk_tex, 13);
42      t2 = d_Te0[s2 >> 24] ^ d_Te1[(s3 >> 16) & 0xff] ^ d_Te2[(s0 >> 8) & 0xff]
43          ^ d_Te3[s1 & 0xff] ^ tex1Dfetch(rk_tex, 14);
44      t3 = d_Te0[s3 >> 24] ^ d_Te1[(s0 >> 16) & 0xff] ^ d_Te2[(s1 >> 8) & 0xff]
45          ^ d_Te3[s2 & 0xff] ^ tex1Dfetch(rk_tex, 15);
46
47      /* round 4: */
48      s0 = d_Te0[t0 >> 24] ^ d_Te1[(t1 >> 16) & 0xff] ^ d_Te2[(t2 >> 8) & 0xff]
49          ^ d_Te3[t3 & 0xff] ^ tex1Dfetch(rk_tex, 16);
50      s1 = d_Te0[t1 >> 24] ^ d_Te1[(t2 >> 16) & 0xff] ^ d_Te2[(t3 >> 8) & 0xff]
51          ^ d_Te3[t0 & 0xff] ^ tex1Dfetch(rk_tex, 17);
52      s2 = d_Te0[t2 >> 24] ^ d_Te1[(t3 >> 16) & 0xff] ^ d_Te2[(t0 >> 8) & 0xff]
53          ^ d_Te3[t1 & 0xff] ^ tex1Dfetch(rk_tex, 18);
54      s3 = d_Te0[t3 >> 24] ^ d_Te1[(t0 >> 16) & 0xff] ^ d_Te2[(t1 >> 8) & 0xff]
55          ^ d_Te3[t2 & 0xff] ^ tex1Dfetch(rk_tex, 19);
56
57      /* round 5: */
58      t0 = d_Te0[s0 >> 24] ^ d_Te1[(s1 >> 16) & 0xff] ^ d_Te2[(s2 >> 8) & 0xff]
59          ^ d_Te3[s3 & 0xff] ^ tex1Dfetch(rk_tex, 20);
60      t1 = d_Te0[s1 >> 24] ^ d_Te1[(s2 >> 16) & 0xff] ^ d_Te2[(s3 >> 8) & 0xff]
61          ^ d_Te3[s0 & 0xff] ^ tex1Dfetch(rk_tex, 21);
62      t2 = d_Te0[s2 >> 24] ^ d_Te1[(s3 >> 16) & 0xff] ^ d_Te2[(s0 >> 8) & 0xff]
63          ^ d_Te3[s1 & 0xff] ^ tex1Dfetch(rk_tex, 22);
64      t3 = d_Te0[s3 >> 24] ^ d_Te1[(s0 >> 16) & 0xff] ^ d_Te2[(s1 >> 8) & 0xff]
65          ^ d_Te3[s2 & 0xff] ^ tex1Dfetch(rk_tex, 23);
66
67      /* round 6: */
68      s0 = d_Te0[t0 >> 24] ^ d_Te1[(t1 >> 16) & 0xff] ^ d_Te2[(t2 >> 8) & 0xff]
69          ^ d_Te3[t3 & 0xff] ^ tex1Dfetch(rk_tex, 24);
70      s1 = d_Te0[t1 >> 24] ^ d_Te1[(t2 >> 16) & 0xff] ^ d_Te2[(t3 >> 8) & 0xff]
71          ^ d_Te3[t0 & 0xff] ^ tex1Dfetch(rk_tex, 25);
72      s2 = d_Te0[t2 >> 24] ^ d_Te1[(t3 >> 16) & 0xff] ^ d_Te2[(t0 >> 8) & 0xff]
73          ^ d_Te3[t1 & 0xff] ^ tex1Dfetch(rk_tex, 26);
74      s3 = d_Te0[t3 >> 24] ^ d_Te1[(t0 >> 16) & 0xff] ^ d_Te2[(t1 >> 8) & 0xff]
75          ^ d_Te3[t2 & 0xff] ^ tex1Dfetch(rk_tex, 27);
76
77      /* round 7: */
78      t0 = d_Te0[s0 >> 24] ^ d_Te1[(s1 >> 16) & 0xff] ^ d_Te2[(s2 >> 8) & 0xff]
79          ^ d_Te3[s3 & 0xff] ^ tex1Dfetch(rk_tex, 28);
80      t1 = d_Te0[s1 >> 24] ^ d_Te1[(s2 >> 16) & 0xff] ^ d_Te2[(s3 >> 8) & 0xff]
81          ^ d_Te3[s0 & 0xff] ^ tex1Dfetch(rk_tex, 29);
82      t2 = d_Te0[s2 >> 24] ^ d_Te1[(s3 >> 16) & 0xff] ^ d_Te2[(s0 >> 8) & 0xff]
83          ^ d_Te3[s1 & 0xff] ^ tex1Dfetch(rk_tex, 30);
84      t3 = d_Te0[s3 >> 24] ^ d_Te1[(s0 >> 16) & 0xff] ^ d_Te2[(s1 >> 8) & 0xff]
85          ^ d_Te3[s2 & 0xff] ^ tex1Dfetch(rk_tex, 31);
86
87      /* round 8: */
88      s0 = d_Te0[t0 >> 24] ^ d_Te1[(t1 >> 16) & 0xff] ^ d_Te2[(t2 >> 8) & 0xff]
```

```
89          ^ d_Te3[t3 & 0xff] ^ tex1Dfetch(rk_tex, 32);
90    s1 = d_Te0[t1 >> 24] ^ d_Te1[(t2 >> 16) & 0xff] ^ d_Te2[(t3 >> 8) & 0xff]
91          ^ d_Te3[t0 & 0xff] ^ tex1Dfetch(rk_tex, 33);
92    s2 = d_Te0[t2 >> 24] ^ d_Te1[(t3 >> 16) & 0xff] ^ d_Te2[(t0 >> 8) & 0xff]
93          ^ d_Te3[t1 & 0xff] ^ tex1Dfetch(rk_tex, 34);
94    s3 = d_Te0[t3 >> 24] ^ d_Te1[(t0 >> 16) & 0xff] ^ d_Te2[(t1 >> 8) & 0xff]
95          ^ d_Te3[t2 & 0xff] ^ tex1Dfetch(rk_tex, 35);
96
97    /* round 9: */
98    t0 = d_Te0[s0 >> 24] ^ d_Te1[(s1 >> 16) & 0xff] ^ d_Te2[(s2 >> 8) & 0xff]
99          ^ d_Te3[s3 & 0xff] ^ tex1Dfetch(rk_tex, 36);
100   t1 = d_Te0[s1 >> 24] ^ d_Te1[(s2 >> 16) & 0xff] ^ d_Te2[(s3 >> 8) & 0xff]
101         ^ d_Te3[s0 & 0xff] ^ tex1Dfetch(rk_tex, 37);
102   t2 = d_Te0[s2 >> 24] ^ d_Te1[(s3 >> 16) & 0xff] ^ d_Te2[(s0 >> 8) & 0xff]
103         ^ d_Te3[s1 & 0xff] ^ tex1Dfetch(rk_tex, 38);
104   t3 = d_Te0[s3 >> 24] ^ d_Te1[(s0 >> 16) & 0xff] ^ d_Te2[(s1 >> 8) & 0xff]
105         ^ d_Te3[s2 & 0xff] ^ tex1Dfetch(rk_tex, 39);
106
107   ct[4 * tid] = (d_Te4[(t0 >> 24)] & 0xff000000)
108         ^ (d_Te4[(t1 >> 16) & 0xff] & 0x00ff0000)
109         ^ (d_Te4[(t2 >> 8) & 0xff] & 0x0000ff00)
110         ^ (d_Te4[(t3) & 0xff] & 0x000000ff) ^ tex1Dfetch(rk_tex, 40);
111   ct[4 * tid + 1] = (d_Te4[(t1 >> 24)] & 0xff000000)
112         ^ (d_Te4[(t2 >> 16) & 0xff] & 0x00ff0000)
113         ^ (d_Te4[(t3 >> 8) & 0xff] & 0x0000ff00)
114         ^ (d_Te4[(t0) & 0xff] & 0x000000ff) ^ tex1Dfetch(rk_tex, 41);
115   ct[4 * tid + 2] = (d_Te4[(t2 >> 24)] & 0xff000000)
116         ^ (d_Te4[(t3 >> 16) & 0xff] & 0x00ff0000)
117         ^ (d_Te4[(t0 >> 8) & 0xff] & 0x0000ff00)
118         ^ (d_Te4[(t1) & 0xff] & 0x000000ff) ^ tex1Dfetch(rk_tex, 42);
119   ct[4 * tid + 3] = (d_Te4[(t3 >> 24)] & 0xff000000)
120         ^ (d_Te4[(t0 >> 16) & 0xff] & 0x00ff0000)
121         ^ (d_Te4[(t1 >> 8) & 0xff] & 0x0000ff00)
122         ^ (d_Te4[(t2) & 0xff] & 0x000000ff) ^ tex1Dfetch(rk_tex, 43);
123
124   return;
125 }
126 void AES_Enc(unint * pt, unint * ct, unint * rk, unint nk, unint pt_size,
127      unint threads_per_block) {
128 #ifdef TEMPO
129   cudaEvent_t tstart, tstop;
130   HANDLE_ERROR(cudaEventCreate(&tstart));
131   HANDLE_ERROR(cudaEventCreate(&tstop));
132
133   cudaEvent_t cstart, cstop;
134   HANDLE_ERROR(cudaEventCreate(&cstart));
135   HANDLE_ERROR(cudaEventCreate(&cstop));
136
137   cudaEvent_t start, stop;
138   HANDLE_ERROR(cudaEventCreate(&start));
139   HANDLE_ERROR(cudaEventCreate(&stop));
140
141   float elapsedTime;
142 #endif
143
144   unint *d_pt, *d_ct, *d_rk;
145
146   unint N = pt_size / 16;
147   unint nr = NR(nk);
148   unint nkx = 4 * (nr + 1);
```

```
149   unint nkxb = nkx * sizeof(unint);
150
151   cudaDeviceProp prop;
152   HANDLE_ERROR( cudaGetDeviceProperties( &prop,0));
153
154   if (threads_per_block > prop.maxThreadsPerBlock)
155     threads_per_block = prop.maxThreadsPerBlock;
156
157   unint blocks = (unint) N / threads_per_block;
158
159   if (N % threads_per_block != 0)
160     blocks++;
161   unint blocks1 = blocks;
162   unint blocks2 = 1;
163
164   // Blocks transformado em bidimensional se ultrapassar maximo
165   unint maxgridsize=prop.maxGridSize[0];
166   if (blocks > maxgridsize) {
167     unint max = (unint) sqrt(blocks);
168     blocks1 = blocks;
169     blocks2 = 1;
170     unint i = 0;
171     // procura divisores de blocks
172     while (primo[i] <= max) {
173       if (blocks1 % primo[i] == 0) {
174         blocks1 = blocks1 / primo[i];
175         blocks2 = blocks2*primo[i];
176         if (blocks1 > maxgridsize) continue;
177         break;
178       }
179       i++;
180     }
181     //se nao encontrar divisores tentar minimizar blocks1 e blocks2
182     if ((blocks1 > maxgridsize)||(blocks2 > maxgridsize)) {
183       blocks1 = max;
184       blocks2 = max;
185       while (blocks1 * blocks2 * threads_per_block < N)
186         blocks2++;
187     }
188   }
189   dim3 BLOCKS(blocks1, blocks2, 1);
190
191 #ifdef TEMPO
192   HANDLE_ERROR(cudaEventRecord(tstart, 0));
193 #endif
194
195   HANDLE_ERROR(cudaMalloc((void **)&d_pt, pt_size));
196   HANDLE_ERROR(cudaMalloc((void **)&d_ct, pt_size));
197   HANDLE_ERROR(cudaMalloc((void **)&d_rk, nkxb));
198
199 #ifdef TEMPO
200   HANDLE_ERROR(cudaEventRecord(cstart, 0));
201 #endif
202
203   HANDLE_ERROR(cudaMemcpy(d_rk, rk, nkxb, cudaMemcpyHostToDevice));
204   HANDLE_ERROR(cudaMemcpy(d_pt, pt, pt_size, cudaMemcpyHostToDevice));
205   HANDLE_ERROR(cudaBindTexture(NULL, rk_tex, d_rk,nkxb));
206
207 #ifdef TEMPO
208   HANDLE_ERROR(cudaEventRecord(start, 0));
```

```
209 #endif
210   switch (nk) {
211   case 4:
212     AES_Enc128 <<< BLOCKS, threads_per_block >>> (d_pt, d_ct,N);
213     break;
214   case 6:
215     AES_Enc192 <<< BLOCKS, threads_per_block >>> (d_pt, d_ct,N);
216     break;
217   case 8:
218     AES_Enc256 <<< BLOCKS, threads_per_block >>> (d_pt, d_ct,N);
219     break;
220   default:
221     return;
222   }
223 #ifdef TEMPO
224   HANDLE_ERROR(cudaEventRecord(stop, 0));
225 #endif
226
227   HANDLE_ERROR(cudaMemcpy(ct, d_ct, pt_size, cudaMemcpyDeviceToHost));
228
229 #ifdef TEMPO
230   HANDLE_ERROR(cudaEventRecord(cstop, 0));
231 #endif
232
233   HANDLE_ERROR(cudaUnbindTexture(rk_tex));
234   HANDLE_ERROR(cudaFree(d_pt));
235   HANDLE_ERROR(cudaFree(d_ct));
236   HANDLE_ERROR(cudaFree(d_rk));
237
238 #ifdef TEMPO
239   HANDLE_ERROR(cudaEventRecord(tstop, 0));
240   HANDLE_ERROR(cudaEventSynchronize(tstop));
241
242   HANDLE_ERROR(cudaEventElapsedTime(&elapsedTime, start, stop));
243   fprintf(stderr, "time(enc_cuda_%d):\t\t%12.6f ms\t%4d threads\t%u bytes\n",
244       (4 * nk * 8), elapsedTime, threads_per_block,pt_size);
245
246   HANDLE_ERROR(cudaEventElapsedTime(&elapsedTime, cstart, cstop));
247   fprintf(stderr, "time(enc_cuda_cp_%d):\t%12.6f ms\t%4d threads\t%u bytes\n",
248       (4 * nk * 8), elapsedTime, threads_per_block,pt_size);
249
250   HANDLE_ERROR(cudaEventElapsedTime(&elapsedTime, tstart, tstop));
251   fprintf(stderr, "time(enc_cuda_tt_%d):\t%12.6f ms\t%4d threads\t%u bytes\n",
252       (4 * nk * 8), elapsedTime, threads_per_block,pt_size);
253
254   HANDLE_ERROR(cudaEventDestroy(start));
255   HANDLE_ERROR(cudaEventDestroy(stop));
256
257   HANDLE_ERROR(cudaEventDestroy(cstart));
258   HANDLE_ERROR(cudaEventDestroy(cstop));
259
260   HANDLE_ERROR(cudaEventDestroy(tstart));
261   HANDLE_ERROR(cudaEventDestroy(tstop));
262 #endif
263 }
```

II.11 cuda_aes_1b_nt_shared_const

Código fonte II.21: cuda_aes_1b_nt_shared_const/aes.cu

```
__constant__ __device__ unint d_rk[60];
__global__ void AES_Enc128(unint * pt, unint * ct, unint N) {
unint tid = blockIdx.x * blockDim.x + threadIdx.x;
  unint tx = threadIdx.x;

  __shared__ unint s_Te0[256 + 1];
  __shared__ unint s_Te1[256 + 1];
  __shared__ unint s_Te2[256 + 1];
  __shared__ unint s_Te3[256 + 1];
  __shared__ unint s_Te4[256 + 1];

  for (int i = tx; i < 64; i += blockDim.x) {
    s_Te0[4 * i] = d_Te0[4 * i];
    s_Te0[4 * i + 1] = d_Te0[4 * i + 1];
    s_Te0[4 * i + 2] = d_Te0[4 * i + 2];
    s_Te0[4 * i + 3] = d_Te0[4 * i + 3];

    s_Te1[4 * i] = d_Te1[4 * i];
    s_Te1[4 * i + 1] = d_Te1[4 * i + 1];
    s_Te1[4 * i + 2] = d_Te1[4 * i + 2];
    s_Te1[4 * i + 3] = d_Te1[4 * i + 3];

    s_Te2[4 * i] = d_Te2[4 * i];
    s_Te2[4 * i + 1] = d_Te2[4 * i + 1];
    s_Te2[4 * i + 2] = d_Te2[4 * i + 2];
    s_Te2[4 * i + 3] = d_Te2[4 * i + 3];

    s_Te3[4 * i] = d_Te3[4 * i];
    s_Te3[4 * i + 1] = d_Te3[4 * i + 1];
    s_Te3[4 * i + 2] = d_Te3[4 * i + 2];
    s_Te3[4 * i + 3] = d_Te3[4 * i + 3];

    s_Te4[4 * i] = d_Te4[4 * i];
    s_Te4[4 * i + 1] = d_Te4[4 * i + 1];
    s_Te4[4 * i + 2] = d_Te4[4 * i + 2];
    s_Te4[4 * i + 3] = d_Te4[4 * i + 3];
  }

  __syncthreads();

  unint s0, s1, s2, s3, t0, t1, t2, t3;

  while (tid < N) {

    s0 = pt[4 * tid] ^ d_rk[0];
    s1 = pt[4 * tid + 1] ^ d_rk[1];
    s2 = pt[4 * tid + 2] ^ d_rk[2];
    s3 = pt[4 * tid + 3] ^ d_rk[3];

    /* round 1: */
    t0 = s_Te0[s0 >> 24] ^ s_Te1[(s1 >> 16) & 0xff]
        ^ s_Te2[(s2 >> 8) & 0xff] ^ s_Te3[s3 & 0xff]
        ^ d_rk[4];
    t1 = s_Te0[s1 >> 24] ^ s_Te1[(s2 >> 16) & 0xff]
        ^ s_Te2[(s3 >> 8) & 0xff] ^ s_Te3[s0 & 0xff]
        ^ d_rk[5];
```

```
    t2 = s_Te0[s2 >> 24] ^ s_Te1[(s3 >> 16) & 0xff]
        ^ s_Te2[(s0 >> 8) & 0xff] ^ s_Te3[s1 & 0xff]
        ^ d_rk[6];
    t3 = s_Te0[s3 >> 24] ^ s_Te1[(s0 >> 16) & 0xff]
        ^ s_Te2[(s1 >> 8) & 0xff] ^ s_Te3[s2 & 0xff]
        ^ d_rk[7];

    /* round 2: */
    s0 = s_Te0[t0 >> 24] ^ s_Te1[(t1 >> 16) & 0xff]
        ^ s_Te2[(t2 >> 8) & 0xff] ^ s_Te3[t3 & 0xff]
        ^ d_rk[8];
    s1 = s_Te0[t1 >> 24] ^ s_Te1[(t2 >> 16) & 0xff]
        ^ s_Te2[(t3 >> 8) & 0xff] ^ s_Te3[t0 & 0xff]
        ^ d_rk[9];
    s2 = s_Te0[t2 >> 24] ^ s_Te1[(t3 >> 16) & 0xff]
        ^ s_Te2[(t0 >> 8) & 0xff] ^ s_Te3[t1 & 0xff]
        ^ d_rk[10];
    s3 = s_Te0[t3 >> 24] ^ s_Te1[(t0 >> 16) & 0xff]
        ^ s_Te2[(t1 >> 8) & 0xff] ^ s_Te3[t2 & 0xff]
        ^ d_rk[11];

    /* round 3: */
    t0 = s_Te0[s0 >> 24] ^ s_Te1[(s1 >> 16) & 0xff]
        ^ s_Te2[(s2 >> 8) & 0xff] ^ s_Te3[s3 & 0xff]
        ^ d_rk[12];
    t1 = s_Te0[s1 >> 24] ^ s_Te1[(s2 >> 16) & 0xff]
        ^ s_Te2[(s3 >> 8) & 0xff] ^ s_Te3[s0 & 0xff]
        ^ d_rk[13];
    t2 = s_Te0[s2 >> 24] ^ s_Te1[(s3 >> 16) & 0xff]
        ^ s_Te2[(s0 >> 8) & 0xff] ^ s_Te3[s1 & 0xff]
        ^ d_rk[14];
    t3 = s_Te0[s3 >> 24] ^ s_Te1[(s0 >> 16) & 0xff]
        ^ s_Te2[(s1 >> 8) & 0xff] ^ s_Te3[s2 & 0xff]
        ^ d_rk[15];

    /* round 4: */
    s0 = s_Te0[t0 >> 24] ^ s_Te1[(t1 >> 16) & 0xff]
        ^ s_Te2[(t2 >> 8) & 0xff] ^ s_Te3[t3 & 0xff]
        ^ d_rk[16];
    s1 = s_Te0[t1 >> 24] ^ s_Te1[(t2 >> 16) & 0xff]
        ^ s_Te2[(t3 >> 8) & 0xff] ^ s_Te3[t0 & 0xff]
        ^ d_rk[17];
    s2 = s_Te0[t2 >> 24] ^ s_Te1[(t3 >> 16) & 0xff]
        ^ s_Te2[(t0 >> 8) & 0xff] ^ s_Te3[t1 & 0xff]
        ^ d_rk[18];
    s3 = s_Te0[t3 >> 24] ^ s_Te1[(t0 >> 16) & 0xff]
        ^ s_Te2[(t1 >> 8) & 0xff] ^ s_Te3[t2 & 0xff]
        ^ d_rk[19];

    /* round 5: */
    t0 = s_Te0[s0 >> 24] ^ s_Te1[(s1 >> 16) & 0xff]
        ^ s_Te2[(s2 >> 8) & 0xff] ^ s_Te3[s3 & 0xff]
        ^ d_rk[20];
    t1 = s_Te0[s1 >> 24] ^ s_Te1[(s2 >> 16) & 0xff]
        ^ s_Te2[(s3 >> 8) & 0xff] ^ s_Te3[s0 & 0xff]
        ^ d_rk[21];
    t2 = s_Te0[s2 >> 24] ^ s_Te1[(s3 >> 16) & 0xff]
        ^ s_Te2[(s0 >> 8) & 0xff] ^ s_Te3[s1 & 0xff]
        ^ d_rk[22];
    t3 = s_Te0[s3 >> 24] ^ s_Te1[(s0 >> 16) & 0xff]
```

```
        ^ s_Te2[(s1 >> 8) & 0xff] ^ s_Te3[s2 & 0xff]
        ^ d_rk[23];

    /* round 6: */
    s0 = s_Te0[t0 >> 24] ^ s_Te1[(t1 >> 16) & 0xff]
        ^ s_Te2[(t2 >> 8) & 0xff] ^ s_Te3[t3 & 0xff]
        ^ d_rk[24];
    s1 = s_Te0[t1 >> 24] ^ s_Te1[(t2 >> 16) & 0xff]
        ^ s_Te2[(t3 >> 8) & 0xff] ^ s_Te3[t0 & 0xff]
        ^ d_rk[25];
    s2 = s_Te0[t2 >> 24] ^ s_Te1[(t3 >> 16) & 0xff]
        ^ s_Te2[(t0 >> 8) & 0xff] ^ s_Te3[t1 & 0xff]
        ^ d_rk[26];
    s3 = s_Te0[t3 >> 24] ^ s_Te1[(t0 >> 16) & 0xff]
        ^ s_Te2[(t1 >> 8) & 0xff] ^ s_Te3[t2 & 0xff]
        ^ d_rk[27];

    /* round 7: */
    t0 = s_Te0[s0 >> 24] ^ s_Te1[(s1 >> 16) & 0xff]
        ^ s_Te2[(s2 >> 8) & 0xff] ^ s_Te3[s3 & 0xff]
        ^ d_rk[28];
    t1 = s_Te0[s1 >> 24] ^ s_Te1[(s2 >> 16) & 0xff]
        ^ s_Te2[(s3 >> 8) & 0xff] ^ s_Te3[s0 & 0xff]
        ^ d_rk[29];
    t2 = s_Te0[s2 >> 24] ^ s_Te1[(s3 >> 16) & 0xff]
        ^ s_Te2[(s0 >> 8) & 0xff] ^ s_Te3[s1 & 0xff]
        ^ d_rk[30];
    t3 = s_Te0[s3 >> 24] ^ s_Te1[(s0 >> 16) & 0xff]
        ^ s_Te2[(s1 >> 8) & 0xff] ^ s_Te3[s2 & 0xff]
        ^ d_rk[31];

    /* round 8: */
    s0 = s_Te0[t0 >> 24] ^ s_Te1[(t1 >> 16) & 0xff]
        ^ s_Te2[(t2 >> 8) & 0xff] ^ s_Te3[t3 & 0xff]
        ^ d_rk[32];
    s1 = s_Te0[t1 >> 24] ^ s_Te1[(t2 >> 16) & 0xff]
        ^ s_Te2[(t3 >> 8) & 0xff] ^ s_Te3[t0 & 0xff]
        ^ d_rk[33];
    s2 = s_Te0[t2 >> 24] ^ s_Te1[(t3 >> 16) & 0xff]
        ^ s_Te2[(t0 >> 8) & 0xff] ^ s_Te3[t1 & 0xff]
        ^ d_rk[34];
    s3 = s_Te0[t3 >> 24] ^ s_Te1[(t0 >> 16) & 0xff]
        ^ s_Te2[(t1 >> 8) & 0xff] ^ s_Te3[t2 & 0xff]
        ^ d_rk[35];

    /* round 9: */
    t0 = s_Te0[s0 >> 24] ^ s_Te1[(s1 >> 16) & 0xff]
        ^ s_Te2[(s2 >> 8) & 0xff] ^ s_Te3[s3 & 0xff]
        ^ d_rk[36];
    t1 = s_Te0[s1 >> 24] ^ s_Te1[(s2 >> 16) & 0xff]
        ^ s_Te2[(s3 >> 8) & 0xff] ^ s_Te3[s0 & 0xff]
        ^ d_rk[37];
    t2 = s_Te0[s2 >> 24] ^ s_Te1[(s3 >> 16) & 0xff]
        ^ s_Te2[(s0 >> 8) & 0xff] ^ s_Te3[s1 & 0xff]
        ^ d_rk[38];
    t3 = s_Te0[s3 >> 24] ^ s_Te1[(s0 >> 16) & 0xff]
        ^ s_Te2[(s1 >> 8) & 0xff] ^ s_Te3[s2 & 0xff]
        ^ d_rk[39];

    ct[4 * tid] = (s_Te4[(t0 >> 24)] & 0xff000000)
```

```
177         ^ (s_Te4[(t1 >> 16) & 0xff] & 0x00ff0000)
178         ^ (s_Te4[(t2 >> 8) & 0xff] & 0x0000ff00)
179         ^ (s_Te4[(t3) & 0xff] & 0x000000ff) ^ d_rk[40];
180     ct[4 * tid + 1] = (s_Te4[(t1 >> 24)] & 0xff000000)
181         ^ (s_Te4[(t2 >> 16) & 0xff] & 0x00ff0000)
182         ^ (s_Te4[(t3 >> 8) & 0xff] & 0x0000ff00)
183         ^ (s_Te4[(t0) & 0xff] & 0x000000ff) ^ d_rk[41];
184     ct[4 * tid + 2] = (s_Te4[(t2 >> 24)] & 0xff000000)
185         ^ (s_Te4[(t3 >> 16) & 0xff] & 0x00ff0000)
186         ^ (s_Te4[(t0 >> 8) & 0xff] & 0x0000ff00)
187         ^ (s_Te4[(t1) & 0xff] & 0x000000ff) ^ d_rk[42];
188     ct[4 * tid + 3] = (s_Te4[(t3 >> 24)] & 0xff000000)
189         ^ (s_Te4[(t0 >> 16) & 0xff] & 0x00ff0000)
190         ^ (s_Te4[(t1 >> 8) & 0xff] & 0x0000ff00)
191         ^ (s_Te4[(t2) & 0xff] & 0x000000ff) ^ d_rk[43];
192     tid +=  blockDim.x;
193   }
194   return;
195 }
196 void AES_Enc(unint * pt, unint * ct, unint * rk, unint nk, unint pt_size,
197     unint threads_per_block) {
198 #ifdef TEMPO
199   cudaEvent_t tstart, tstop;
200   HANDLE_ERROR(cudaEventCreate(&tstart));
201   HANDLE_ERROR(cudaEventCreate(&tstop));
202 
203   cudaEvent_t cstart, cstop;
204   HANDLE_ERROR(cudaEventCreate(&cstart));
205   HANDLE_ERROR(cudaEventCreate(&cstop));
206 
207   cudaEvent_t start, stop;
208   HANDLE_ERROR(cudaEventCreate(&start));
209   HANDLE_ERROR(cudaEventCreate(&stop));
210 
211   float elapsedTime;
212 #endif
213 
214   unint *d_pt, *d_ct;
215 
216   unint N = pt_size / 16;
217   unint nr = NR(nk);
218   unint nkx = 4 * (nr + 1);
219   unint nkxb = nkx * sizeof(unint);
220 
221 #ifdef TEMPO
222   HANDLE_ERROR(cudaEventRecord(tstart, 0));
223 #endif
224 
225   HANDLE_ERROR(cudaMalloc((void **)&d_pt, pt_size));
226   HANDLE_ERROR(cudaMalloc((void **)&d_ct, pt_size));
227 
228 #ifdef TEMPO
229   HANDLE_ERROR(cudaEventRecord(cstart, 0));
230 #endif
231 
232   HANDLE_ERROR(cudaMemcpy(d_pt, pt, pt_size, cudaMemcpyHostToDevice));
233   HANDLE_ERROR(cudaMemcpyToSymbol(d_rk, rk, nkxb));
234 
235 #ifdef TEMPO
236   HANDLE_ERROR(cudaEventRecord(start, 0));
```

```
237 #endif
238   switch (nk) {
239   case 4:
240     AES_Enc128 <<< 1, threads_per_block >>> (d_pt, d_ct,N);
241     break;
242   case 6:
243     AES_Enc192 <<< 1, threads_per_block >>> (d_pt, d_ct,N);
244     break;
245   case 8:
246     AES_Enc256 <<< 1, threads_per_block >>> (d_pt, d_ct,N);
247     break;
248   default:
249     return;
250   }
251 #ifdef TEMPO
252   HANDLE_ERROR(cudaEventRecord(stop, 0));
253 #endif
254
255   HANDLE_ERROR(cudaMemcpy(ct, d_ct, pt_size, cudaMemcpyDeviceToHost));
256
257 #ifdef TEMPO
258   HANDLE_ERROR(cudaEventRecord(cstop, 0));
259 #endif
260
261   HANDLE_ERROR(cudaFree(d_pt));
262   HANDLE_ERROR(cudaFree(d_ct));
263
264 #ifdef TEMPO
265   HANDLE_ERROR(cudaEventRecord(tstop, 0));
266   HANDLE_ERROR(cudaEventSynchronize(tstop));
267
268   HANDLE_ERROR(cudaEventElapsedTime(&elapsedTime, start, stop));
269   fprintf(stderr, "time(enc_cuda_%d):\t\t%12.6f ms\t%d threads\t%u bytes\n",
270       (4 * nk * 8), elapsedTime,threads_per_block,pt_size);
271
272   HANDLE_ERROR(cudaEventElapsedTime(&elapsedTime, cstart, cstop));
273   fprintf(stderr, "time(enc_cuda_cp_%d):\t\t%12.6f ms\t%d threads\t%u bytes\n",
274       (4 * nk * 8), elapsedTime,threads_per_block,pt_size);
275
276   HANDLE_ERROR(cudaEventElapsedTime(&elapsedTime, tstart, tstop));
277   fprintf(stderr, "time(enc_cuda_tt_%d):\t%12.6f ms\t%d threads\t%u bytes\n",
278       (4 * nk * 8), elapsedTime,threads_per_block,pt_size);
279
280   HANDLE_ERROR(cudaEventDestroy(start));
281   HANDLE_ERROR(cudaEventDestroy(stop));
282
283   HANDLE_ERROR(cudaEventDestroy(cstart));
284   HANDLE_ERROR(cudaEventDestroy(cstop));
285
286   HANDLE_ERROR(cudaEventDestroy(tstart));
287   HANDLE_ERROR(cudaEventDestroy(tstop));
288 #endif
289 }
```

II.12 cuda_aes_nb_nt_shared_const

Código fonte II.22: cuda_aes_nb_nt_shared_const/aes.cu

```
__constant__ __device__ unint d_rk[60];
__global__ void AES_Enc128(unint * pt, unint * ct, unint N) {
  unint tid = blockIdx.y * gridDim.x * blockDim.x + blockIdx.x * blockDim.x
      + threadIdx.x;
  unint tx = threadIdx.x;

  __shared__ unint s_Te0[256 + 1];
  __shared__ unint s_Te1[256 + 1];
  __shared__ unint s_Te2[256 + 1];
  __shared__ unint s_Te3[256 + 1];
  __shared__ unint s_Te4[256 + 1];

  for (int i = tx; i < 64; i += blockDim.x) {
    s_Te0[4 * i] = d_Te0[4 * i];
    s_Te0[4 * i + 1] = d_Te0[4 * i + 1];
    s_Te0[4 * i + 2] = d_Te0[4 * i + 2];
    s_Te0[4 * i + 3] = d_Te0[4 * i + 3];

    s_Te1[4 * i] = d_Te1[4 * i];
    s_Te1[4 * i + 1] = d_Te1[4 * i + 1];
    s_Te1[4 * i + 2] = d_Te1[4 * i + 2];
    s_Te1[4 * i + 3] = d_Te1[4 * i + 3];

    s_Te2[4 * i] = d_Te2[4 * i];
    s_Te2[4 * i + 1] = d_Te2[4 * i + 1];
    s_Te2[4 * i + 2] = d_Te2[4 * i + 2];
    s_Te2[4 * i + 3] = d_Te2[4 * i + 3];

    s_Te3[4 * i] = d_Te3[4 * i];
    s_Te3[4 * i + 1] = d_Te3[4 * i + 1];
    s_Te3[4 * i + 2] = d_Te3[4 * i + 2];
    s_Te3[4 * i + 3] = d_Te3[4 * i + 3];

    s_Te4[4 * i] = d_Te4[4 * i];
    s_Te4[4 * i + 1] = d_Te4[4 * i + 1];
    s_Te4[4 * i + 2] = d_Te4[4 * i + 2];
    s_Te4[4 * i + 3] = d_Te4[4 * i + 3];
  }
  __syncthreads();

  unint s0, s1, s2, s3, t0, t1, t2, t3;

  if (tid < N) {

    s0 = pt[4 * tid] ^ d_rk[0];
    s1 = pt[4 * tid + 1] ^ d_rk[1];
    s2 = pt[4 * tid + 2] ^ d_rk[2];
    s3 = pt[4 * tid + 3] ^ d_rk[3];

    /* round 1: */
    t0 = s_Te0[s0 >> 24] ^ s_Te1[(s1 >> 16) & 0xff]
        ^ s_Te2[(s2 >> 8) & 0xff] ^ s_Te3[s3 & 0xff]
        ^ d_rk[4];
    t1 = s_Te0[s1 >> 24] ^ s_Te1[(s2 >> 16) & 0xff]
        ^ s_Te2[(s3 >> 8) & 0xff] ^ s_Te3[s0 & 0xff]
        ^ d_rk[5];
    t2 = s_Te0[s2 >> 24] ^ s_Te1[(s3 >> 16) & 0xff]
        ^ s_Te2[(s0 >> 8) & 0xff] ^ s_Te3[s1 & 0xff]
        ^ d_rk[6];
```

```
    t3 = s_Te0[s3 >> 24] ^ s_Te1[(s0 >> 16) & 0xff]
        ^ s_Te2[(s1 >> 8) & 0xff] ^ s_Te3[s2 & 0xff]
        ^ d_rk[7];

    /* round 2: */
    s0 = s_Te0[t0 >> 24] ^ s_Te1[(t1 >> 16) & 0xff]
        ^ s_Te2[(t2 >> 8) & 0xff] ^ s_Te3[t3 & 0xff]
        ^ d_rk[8];
    s1 = s_Te0[t1 >> 24] ^ s_Te1[(t2 >> 16) & 0xff]
        ^ s_Te2[(t3 >> 8) & 0xff] ^ s_Te3[t0 & 0xff]
        ^ d_rk[9];
    s2 = s_Te0[t2 >> 24] ^ s_Te1[(t3 >> 16) & 0xff]
        ^ s_Te2[(t0 >> 8) & 0xff] ^ s_Te3[t1 & 0xff]
        ^ d_rk[10];
    s3 = s_Te0[t3 >> 24] ^ s_Te1[(t0 >> 16) & 0xff]
        ^ s_Te2[(t1 >> 8) & 0xff] ^ s_Te3[t2 & 0xff]
        ^ d_rk[11];

    /* round 3: */
    t0 = s_Te0[s0 >> 24] ^ s_Te1[(s1 >> 16) & 0xff]
        ^ s_Te2[(s2 >> 8) & 0xff] ^ s_Te3[s3 & 0xff]
        ^ d_rk[12];
    t1 = s_Te0[s1 >> 24] ^ s_Te1[(s2 >> 16) & 0xff]
        ^ s_Te2[(s3 >> 8) & 0xff] ^ s_Te3[s0 & 0xff]
        ^ d_rk[13];
    t2 = s_Te0[s2 >> 24] ^ s_Te1[(s3 >> 16) & 0xff]
        ^ s_Te2[(s0 >> 8) & 0xff] ^ s_Te3[s1 & 0xff]
        ^ d_rk[14];
    t3 = s_Te0[s3 >> 24] ^ s_Te1[(s0 >> 16) & 0xff]
        ^ s_Te2[(s1 >> 8) & 0xff] ^ s_Te3[s2 & 0xff]
        ^ d_rk[15];

    /* round 4: */
    s0 = s_Te0[t0 >> 24] ^ s_Te1[(t1 >> 16) & 0xff]
        ^ s_Te2[(t2 >> 8) & 0xff] ^ s_Te3[t3 & 0xff]
        ^ d_rk[16];
    s1 = s_Te0[t1 >> 24] ^ s_Te1[(t2 >> 16) & 0xff]
        ^ s_Te2[(t3 >> 8) & 0xff] ^ s_Te3[t0 & 0xff]
        ^ d_rk[17];
    s2 = s_Te0[t2 >> 24] ^ s_Te1[(t3 >> 16) & 0xff]
        ^ s_Te2[(t0 >> 8) & 0xff] ^ s_Te3[t1 & 0xff]
        ^ d_rk[18];
    s3 = s_Te0[t3 >> 24] ^ s_Te1[(t0 >> 16) & 0xff]
        ^ s_Te2[(t1 >> 8) & 0xff] ^ s_Te3[t2 & 0xff]
        ^ d_rk[19];

    /* round 5: */
    t0 = s_Te0[s0 >> 24] ^ s_Te1[(s1 >> 16) & 0xff]
        ^ s_Te2[(s2 >> 8) & 0xff] ^ s_Te3[s3 & 0xff]
        ^ d_rk[20];
    t1 = s_Te0[s1 >> 24] ^ s_Te1[(s2 >> 16) & 0xff]
        ^ s_Te2[(s3 >> 8) & 0xff] ^ s_Te3[s0 & 0xff]
        ^ d_rk[21];
    t2 = s_Te0[s2 >> 24] ^ s_Te1[(s3 >> 16) & 0xff]
        ^ s_Te2[(s0 >> 8) & 0xff] ^ s_Te3[s1 & 0xff]
        ^ d_rk[22];
    t3 = s_Te0[s3 >> 24] ^ s_Te1[(s0 >> 16) & 0xff]
        ^ s_Te2[(s1 >> 8) & 0xff] ^ s_Te3[s2 & 0xff]
        ^ d_rk[23];

```

```
120     /* round 6: */
121     s0 = s_Te0[t0 >> 24] ^ s_Te1[(t1 >> 16) & 0xff]
122         ^ s_Te2[(t2 >> 8) & 0xff] ^ s_Te3[t3 & 0xff]
123         ^ d_rk[24];
124     s1 = s_Te0[t1 >> 24] ^ s_Te1[(t2 >> 16) & 0xff]
125         ^ s_Te2[(t3 >> 8) & 0xff] ^ s_Te3[t0 & 0xff]
126         ^ d_rk[25];
127     s2 = s_Te0[t2 >> 24] ^ s_Te1[(t3 >> 16) & 0xff]
128         ^ s_Te2[(t0 >> 8) & 0xff] ^ s_Te3[t1 & 0xff]
129         ^ d_rk[26];
130     s3 = s_Te0[t3 >> 24] ^ s_Te1[(t0 >> 16) & 0xff]
131         ^ s_Te2[(t1 >> 8) & 0xff] ^ s_Te3[t2 & 0xff]
132         ^ d_rk[27];
133
134     /* round 7: */
135     t0 = s_Te0[s0 >> 24] ^ s_Te1[(s1 >> 16) & 0xff]
136         ^ s_Te2[(s2 >> 8) & 0xff] ^ s_Te3[s3 & 0xff]
137         ^ d_rk[28];
138     t1 = s_Te0[s1 >> 24] ^ s_Te1[(s2 >> 16) & 0xff]
139         ^ s_Te2[(s3 >> 8) & 0xff] ^ s_Te3[s0 & 0xff]
140         ^ d_rk[29];
141     t2 = s_Te0[s2 >> 24] ^ s_Te1[(s3 >> 16) & 0xff]
142         ^ s_Te2[(s0 >> 8) & 0xff] ^ s_Te3[s1 & 0xff]
143         ^ d_rk[30];
144     t3 = s_Te0[s3 >> 24] ^ s_Te1[(s0 >> 16) & 0xff]
145         ^ s_Te2[(s1 >> 8) & 0xff] ^ s_Te3[s2 & 0xff]
146         ^ d_rk[31];
147
148     /* round 8: */
149     s0 = s_Te0[t0 >> 24] ^ s_Te1[(t1 >> 16) & 0xff]
150         ^ s_Te2[(t2 >> 8) & 0xff] ^ s_Te3[t3 & 0xff]
151         ^ d_rk[32];
152     s1 = s_Te0[t1 >> 24] ^ s_Te1[(t2 >> 16) & 0xff]
153         ^ s_Te2[(t3 >> 8) & 0xff] ^ s_Te3[t0 & 0xff]
154         ^ d_rk[33];
155     s2 = s_Te0[t2 >> 24] ^ s_Te1[(t3 >> 16) & 0xff]
156         ^ s_Te2[(t0 >> 8) & 0xff] ^ s_Te3[t1 & 0xff]
157         ^ d_rk[34];
158     s3 = s_Te0[t3 >> 24] ^ s_Te1[(t0 >> 16) & 0xff]
159         ^ s_Te2[(t1 >> 8) & 0xff] ^ s_Te3[t2 & 0xff]
160         ^ d_rk[35];
161
162     /* round 9: */
163     t0 = s_Te0[s0 >> 24] ^ s_Te1[(s1 >> 16) & 0xff]
164         ^ s_Te2[(s2 >> 8) & 0xff] ^ s_Te3[s3 & 0xff]
165         ^ d_rk[36];
166     t1 = s_Te0[s1 >> 24] ^ s_Te1[(s2 >> 16) & 0xff]
167         ^ s_Te2[(s3 >> 8) & 0xff] ^ s_Te3[s0 & 0xff]
168         ^ d_rk[37];
169     t2 = s_Te0[s2 >> 24] ^ s_Te1[(s3 >> 16) & 0xff]
170         ^ s_Te2[(s0 >> 8) & 0xff] ^ s_Te3[s1 & 0xff]
171         ^ d_rk[38];
172     t3 = s_Te0[s3 >> 24] ^ s_Te1[(s0 >> 16) & 0xff]
173         ^ s_Te2[(s1 >> 8) & 0xff] ^ s_Te3[s2 & 0xff]
174         ^ d_rk[39];
175
176     ct[4 * tid] = (s_Te4[(t0 >> 24)] & 0xff000000)
177         ^ (s_Te4[(t1 >> 16) & 0xff] & 0x00ff0000)
178         ^ (s_Te4[(t2 >> 8) & 0xff] & 0x0000ff00)
179         ^ (s_Te4[(t3) & 0xff] & 0x000000ff) ^ d_rk[40];
```

```
180     ct[4 * tid + 1] = (s_Te4[(t1 >> 24)] & 0xff000000)
181         ^ (s_Te4[(t2 >> 16) & 0xff] & 0x00ff0000)
182         ^ (s_Te4[(t3 >> 8) & 0xff] & 0x0000ff00)
183         ^ (s_Te4[(t0) & 0xff] & 0x000000ff) ^ d_rk[41];
184     ct[4 * tid + 2] = (s_Te4[(t2 >> 24)] & 0xff000000)
185         ^ (s_Te4[(t3 >> 16) & 0xff] & 0x00ff0000)
186         ^ (s_Te4[(t0 >> 8) & 0xff] & 0x0000ff00)
187         ^ (s_Te4[(t1) & 0xff] & 0x000000ff) ^ d_rk[42];
188     ct[4 * tid + 3] = (s_Te4[(t3 >> 24)] & 0xff000000)
189         ^ (s_Te4[(t0 >> 16) & 0xff] & 0x00ff0000)
190         ^ (s_Te4[(t1 >> 8) & 0xff] & 0x0000ff00)
191         ^ (s_Te4[(t2) & 0xff] & 0x000000ff) ^ d_rk[43];
192   }
193   return;
194 }
195 void AES_Enc(unint * pt, unint * ct, unint * rk, unint nk, unint pt_size,
196     unint threads_per_block) {
197 #ifdef TEMPO
198   cudaEvent_t tstart, tstop;
199   HANDLE_ERROR(cudaEventCreate(&tstart));
200   HANDLE_ERROR(cudaEventCreate(&tstop));
201 
202   cudaEvent_t cstart, cstop;
203   HANDLE_ERROR(cudaEventCreate(&cstart));
204   HANDLE_ERROR(cudaEventCreate(&cstop));
205 
206   cudaEvent_t start, stop;
207   HANDLE_ERROR(cudaEventCreate(&start));
208   HANDLE_ERROR(cudaEventCreate(&stop));
209 
210   float elapsedTime;
211 #endif
212 
213   unint *d_pt, *d_ct;
214 
215   unint N = pt_size / 16;
216   unint nr = NR(nk);
217   unint nkx = 4 * (nr + 1);
218   unint nkxb = nkx * sizeof(unint);
219 
220   cudaDeviceProp prop;
221   HANDLE_ERROR( cudaGetDeviceProperties( &prop,0));
222 
223   if (threads_per_block > prop.maxThreadsPerBlock)
224     threads_per_block = prop.maxThreadsPerBlock;
225 
226   unint blocks = (unint) N / threads_per_block;
227 
228   if (N % threads_per_block != 0)
229     blocks++;
230   unint blocks1 = blocks;
231   unint blocks2 = 1;
232 
233   // Blocks transformado em bidimensional se ultrapassar maximo
234   unint maxgridsize=prop.maxGridSize[0];
235   if (blocks > maxgridsize) {
236     unint max = (unint) sqrt(blocks);
237     blocks1 = blocks;
238     blocks2 = 1;
239     unint i = 0;
```

```
240     // procura divisores de blocks
241     while (primo[i] <= max) {
242       if (blocks1 % primo[i] == 0) {
243         blocks1 = blocks1 / primo[i];
244         blocks2 = blocks2*primo[i];
245         if (blocks1 > maxgridsize) continue;
246         break;
247       }
248       i++;
249     }
250     //se nao encontrar divisores tentar minimizar blocks1 e blocks2
251     if ((blocks1 > maxgridsize)||(blocks2 > maxgridsize)) {
252       blocks1 = max;
253       blocks2 = max;
254       while (blocks1 * blocks2 * threads_per_block < N)
255         blocks2++;
256     }
257   }
258   dim3 BLOCKS(blocks1, blocks2, 1);
259
260
261 #ifdef TEMPO
262   HANDLE_ERROR(cudaEventRecord(tstart, 0));
263 #endif
264
265   HANDLE_ERROR(cudaMalloc((void **)&d_pt, pt_size));
266   HANDLE_ERROR(cudaMalloc((void **)&d_ct, pt_size));
267
268 #ifdef TEMPO
269   HANDLE_ERROR(cudaEventRecord(cstart, 0));
270 #endif
271   HANDLE_ERROR(cudaMemcpy(d_pt, pt, pt_size, cudaMemcpyHostToDevice));
272   HANDLE_ERROR(cudaMemcpyToSymbol(d_rk, rk, nkxb));
273 #ifdef TEMPO
274   HANDLE_ERROR(cudaEventRecord(start, 0));
275 #endif
276   switch (nk) {
277   case 4:
278     AES_Enc128 <<< BLOCKS, threads_per_block >>> (d_pt, d_ct,N);
279     break;
280   case 6:
281     AES_Enc192 <<< BLOCKS, threads_per_block >>> (d_pt, d_ct,N);
282     break;
283   case 8:
284     AES_Enc256 <<< BLOCKS, threads_per_block >>> (d_pt, d_ct,N);
285     break;
286   default:
287     return;
288   }
289   HANDLE_ERROR(cudaGetLastError());
290 #ifdef TEMPO
291   HANDLE_ERROR(cudaEventRecord(stop, 0));
292 #endif
293
294   HANDLE_ERROR(cudaMemcpy(ct, d_ct, pt_size, cudaMemcpyDeviceToHost));
295
296 #ifdef TEMPO
297   HANDLE_ERROR(cudaEventRecord(cstop, 0));
298 #endif
299
```

```
300   HANDLE_ERROR(cudaFree(d_pt));
301   HANDLE_ERROR(cudaFree(d_ct));
302
303 #ifdef TEMPO
304   HANDLE_ERROR(cudaEventRecord(tstop, 0));
305
306   HANDLE_ERROR(cudaEventSynchronize(tstop));
307
308   HANDLE_ERROR(cudaEventElapsedTime(&elapsedTime, start, stop));
309   fprintf(stderr, "time(enc_cuda_%d):\t\t%12.6f ms\t%4d threads\t%u bytes\n",
310       (4 * nk * 8), elapsedTime,threads_per_block,pt_size);
311
312   HANDLE_ERROR(cudaEventElapsedTime(&elapsedTime, cstart, cstop));
313   fprintf(stderr, "time(enc_cuda_cp_%d):\t%12.6f ms\t%4d threads\t%u bytes\n",
314       (4 * nk * 8), elapsedTime,threads_per_block,pt_size);
315
316   HANDLE_ERROR(cudaEventElapsedTime(&elapsedTime, tstart, tstop));
317   fprintf(stderr, "time(enc_cuda_tt_%d):\t%12.6f ms\t%4d threads\t%u bytes\n",
318       (4 * nk * 8), elapsedTime,threads_per_block,pt_size);
319
320   HANDLE_ERROR(cudaEventDestroy(start));
321   HANDLE_ERROR(cudaEventDestroy(stop));
322
323   HANDLE_ERROR(cudaEventDestroy(cstart));
324   HANDLE_ERROR(cudaEventDestroy(cstop));
325
326   HANDLE_ERROR(cudaEventDestroy(tstart));
327   HANDLE_ERROR(cudaEventDestroy(tstop));
328 #endif
329 }
```

II.13 cuda_aes_1b_nt_shared_texture

Código fonte II.23: cuda_aes_1b_nt_shared_texture/aes.cu

```
1  texture<unsigned, 1, cudaReadModeElementType> rk_tex;
2  __global__ void AES_Enc128(unint * pt, unint * ct, unint N) {
3    unint tid = blockIdx.x * blockDim.x + threadIdx.x;
4    unint tx = threadIdx.x;
5
6    __shared__ unint s_Te0[256 + 1];
7    __shared__ unint s_Te1[256 + 1];
8    __shared__ unint s_Te2[256 + 1];
9    __shared__ unint s_Te3[256 + 1];
10   __shared__ unint s_Te4[256 + 1];
11
12   for (int i = tx; i < 64; i += blockDim.x) {
13     s_Te0[4 * i] = d_Te0[4 * i];
14     s_Te0[4 * i + 1] = d_Te0[4 * i + 1];
15     s_Te0[4 * i + 2] = d_Te0[4 * i + 2];
16     s_Te0[4 * i + 3] = d_Te0[4 * i + 3];
17
18     s_Te1[4 * i] = d_Te1[4 * i];
19     s_Te1[4 * i + 1] = d_Te1[4 * i + 1];
20     s_Te1[4 * i + 2] = d_Te1[4 * i + 2];
21     s_Te1[4 * i + 3] = d_Te1[4 * i + 3];
22
```

```
23      s_Te2[4 * i] = d_Te2[4 * i];
24      s_Te2[4 * i + 1] = d_Te2[4 * i + 1];
25      s_Te2[4 * i + 2] = d_Te2[4 * i + 2];
26      s_Te2[4 * i + 3] = d_Te2[4 * i + 3];
27
28      s_Te3[4 * i] = d_Te3[4 * i];
29      s_Te3[4 * i + 1] = d_Te3[4 * i + 1];
30      s_Te3[4 * i + 2] = d_Te3[4 * i + 2];
31      s_Te3[4 * i + 3] = d_Te3[4 * i + 3];
32
33      s_Te4[4 * i] = d_Te4[4 * i];
34      s_Te4[4 * i + 1] = d_Te4[4 * i + 1];
35      s_Te4[4 * i + 2] = d_Te4[4 * i + 2];
36      s_Te4[4 * i + 3] = d_Te4[4 * i + 3];
37    }
38
39    __syncthreads();
40
41    uint s0, s1, s2, s3, t0, t1, t2, t3;
42
43    while (tid < N) {
44
45      s0 = pt[4 * tid] ^ tex1Dfetch(rk_tex, 0);
46      s1 = pt[4 * tid + 1] ^ tex1Dfetch(rk_tex, 1);
47      s2 = pt[4 * tid + 2] ^ tex1Dfetch(rk_tex, 2);
48      s3 = pt[4 * tid + 3] ^ tex1Dfetch(rk_tex, 3);
49
50      /* round 1: */
51      t0 = s_Te0[s0 >> 24] ^ s_Te1[(s1 >> 16) & 0xff]
52          ^ s_Te2[(s2 >> 8) & 0xff] ^ s_Te3[s3 & 0xff]
53          ^ tex1Dfetch(rk_tex, 4);
54      t1 = s_Te0[s1 >> 24] ^ s_Te1[(s2 >> 16) & 0xff]
55          ^ s_Te2[(s3 >> 8) & 0xff] ^ s_Te3[s0 & 0xff]
56          ^ tex1Dfetch(rk_tex, 5);
57      t2 = s_Te0[s2 >> 24] ^ s_Te1[(s3 >> 16) & 0xff]
58          ^ s_Te2[(s0 >> 8) & 0xff] ^ s_Te3[s1 & 0xff]
59          ^ tex1Dfetch(rk_tex, 6);
60      t3 = s_Te0[s3 >> 24] ^ s_Te1[(s0 >> 16) & 0xff]
61          ^ s_Te2[(s1 >> 8) & 0xff] ^ s_Te3[s2 & 0xff]
62          ^ tex1Dfetch(rk_tex, 7);
63
64      /* round 2: */
65      s0 = s_Te0[t0 >> 24] ^ s_Te1[(t1 >> 16) & 0xff]
66          ^ s_Te2[(t2 >> 8) & 0xff] ^ s_Te3[t3 & 0xff]
67          ^ tex1Dfetch(rk_tex, 8);
68      s1 = s_Te0[t1 >> 24] ^ s_Te1[(t2 >> 16) & 0xff]
69          ^ s_Te2[(t3 >> 8) & 0xff] ^ s_Te3[t0 & 0xff]
70          ^ tex1Dfetch(rk_tex, 9);
71      s2 = s_Te0[t2 >> 24] ^ s_Te1[(t3 >> 16) & 0xff]
72          ^ s_Te2[(t0 >> 8) & 0xff] ^ s_Te3[t1 & 0xff]
73          ^ tex1Dfetch(rk_tex, 10);
74      s3 = s_Te0[t3 >> 24] ^ s_Te1[(t0 >> 16) & 0xff]
75          ^ s_Te2[(t1 >> 8) & 0xff] ^ s_Te3[t2 & 0xff]
76          ^ tex1Dfetch(rk_tex, 11);
77
78      /* round 3: */
79      t0 = s_Te0[s0 >> 24] ^ s_Te1[(s1 >> 16) & 0xff]
80          ^ s_Te2[(s2 >> 8) & 0xff] ^ s_Te3[s3 & 0xff]
81          ^ tex1Dfetch(rk_tex, 12);
82      t1 = s_Te0[s1 >> 24] ^ s_Te1[(s2 >> 16) & 0xff]
```

```
        ^ s_Te2[(s3 >> 8) & 0xff] ^ s_Te3[s0 & 0xff]
        ^ tex1Dfetch(rk_tex, 13);
    t2 = s_Te0[s2 >> 24] ^ s_Te1[(s3 >> 16) & 0xff]
        ^ s_Te2[(s0 >> 8) & 0xff] ^ s_Te3[s1 & 0xff]
        ^ tex1Dfetch(rk_tex, 14);
    t3 = s_Te0[s3 >> 24] ^ s_Te1[(s0 >> 16) & 0xff]
        ^ s_Te2[(s1 >> 8) & 0xff] ^ s_Te3[s2 & 0xff]
        ^ tex1Dfetch(rk_tex, 15);

    /* round 4: */
    s0 = s_Te0[t0 >> 24] ^ s_Te1[(t1 >> 16) & 0xff]
        ^ s_Te2[(t2 >> 8) & 0xff] ^ s_Te3[t3 & 0xff]
        ^ tex1Dfetch(rk_tex, 16);
    s1 = s_Te0[t1 >> 24] ^ s_Te1[(t2 >> 16) & 0xff]
        ^ s_Te2[(t3 >> 8) & 0xff] ^ s_Te3[t0 & 0xff]
        ^ tex1Dfetch(rk_tex, 17);
    s2 = s_Te0[t2 >> 24] ^ s_Te1[(t3 >> 16) & 0xff]
        ^ s_Te2[(t0 >> 8) & 0xff] ^ s_Te3[t1 & 0xff]
        ^ tex1Dfetch(rk_tex, 18);
    s3 = s_Te0[t3 >> 24] ^ s_Te1[(t0 >> 16) & 0xff]
        ^ s_Te2[(t1 >> 8) & 0xff] ^ s_Te3[t2 & 0xff]
        ^ tex1Dfetch(rk_tex, 19);

    /* round 5: */
    t0 = s_Te0[s0 >> 24] ^ s_Te1[(s1 >> 16) & 0xff]
        ^ s_Te2[(s2 >> 8) & 0xff] ^ s_Te3[s3 & 0xff]
        ^ tex1Dfetch(rk_tex, 20);
    t1 = s_Te0[s1 >> 24] ^ s_Te1[(s2 >> 16) & 0xff]
        ^ s_Te2[(s3 >> 8) & 0xff] ^ s_Te3[s0 & 0xff]
        ^ tex1Dfetch(rk_tex, 21);
    t2 = s_Te0[s2 >> 24] ^ s_Te1[(s3 >> 16) & 0xff]
        ^ s_Te2[(s0 >> 8) & 0xff] ^ s_Te3[s1 & 0xff]
        ^ tex1Dfetch(rk_tex, 22);
    t3 = s_Te0[s3 >> 24] ^ s_Te1[(s0 >> 16) & 0xff]
        ^ s_Te2[(s1 >> 8) & 0xff] ^ s_Te3[s2 & 0xff]
        ^ tex1Dfetch(rk_tex, 23);

    /* round 6: */
    s0 = s_Te0[t0 >> 24] ^ s_Te1[(t1 >> 16) & 0xff]
        ^ s_Te2[(t2 >> 8) & 0xff] ^ s_Te3[t3 & 0xff]
        ^ tex1Dfetch(rk_tex, 24);
    s1 = s_Te0[t1 >> 24] ^ s_Te1[(t2 >> 16) & 0xff]
        ^ s_Te2[(t3 >> 8) & 0xff] ^ s_Te3[t0 & 0xff]
        ^ tex1Dfetch(rk_tex, 25);
    s2 = s_Te0[t2 >> 24] ^ s_Te1[(t3 >> 16) & 0xff]
        ^ s_Te2[(t0 >> 8) & 0xff] ^ s_Te3[t1 & 0xff]
        ^ tex1Dfetch(rk_tex, 26);
    s3 = s_Te0[t3 >> 24] ^ s_Te1[(t0 >> 16) & 0xff]
        ^ s_Te2[(t1 >> 8) & 0xff] ^ s_Te3[t2 & 0xff]
        ^ tex1Dfetch(rk_tex, 27);

    /* round 7: */
    t0 = s_Te0[s0 >> 24] ^ s_Te1[(s1 >> 16) & 0xff]
        ^ s_Te2[(s2 >> 8) & 0xff] ^ s_Te3[s3 & 0xff]
        ^ tex1Dfetch(rk_tex, 28);
    t1 = s_Te0[s1 >> 24] ^ s_Te1[(s2 >> 16) & 0xff]
        ^ s_Te2[(s3 >> 8) & 0xff] ^ s_Te3[s0 & 0xff]
        ^ tex1Dfetch(rk_tex, 29);
    t2 = s_Te0[s2 >> 24] ^ s_Te1[(s3 >> 16) & 0xff]
        ^ s_Te2[(s0 >> 8) & 0xff] ^ s_Te3[s1 & 0xff]
```

```
143         ^ tex1Dfetch(rk_tex, 30);
144     t3 = s_Te0[s3 >> 24] ^ s_Te1[(s0 >> 16) & 0xff]
145         ^ s_Te2[(s1 >> 8) & 0xff] ^ s_Te3[s2 & 0xff]
146         ^ tex1Dfetch(rk_tex, 31);
147 
148     /* round 8: */
149     s0 = s_Te0[t0 >> 24] ^ s_Te1[(t1 >> 16) & 0xff]
150         ^ s_Te2[(t2 >> 8) & 0xff] ^ s_Te3[t3 & 0xff]
151         ^ tex1Dfetch(rk_tex, 32);
152     s1 = s_Te0[t1 >> 24] ^ s_Te1[(t2 >> 16) & 0xff]
153         ^ s_Te2[(t3 >> 8) & 0xff] ^ s_Te3[t0 & 0xff]
154         ^ tex1Dfetch(rk_tex, 33);
155     s2 = s_Te0[t2 >> 24] ^ s_Te1[(t3 >> 16) & 0xff]
156         ^ s_Te2[(t0 >> 8) & 0xff] ^ s_Te3[t1 & 0xff]
157         ^ tex1Dfetch(rk_tex, 34);
158     s3 = s_Te0[t3 >> 24] ^ s_Te1[(t0 >> 16) & 0xff]
159         ^ s_Te2[(t1 >> 8) & 0xff] ^ s_Te3[t2 & 0xff]
160         ^ tex1Dfetch(rk_tex, 35);
161 
162     /* round 9: */
163     t0 = s_Te0[s0 >> 24] ^ s_Te1[(s1 >> 16) & 0xff]
164         ^ s_Te2[(s2 >> 8) & 0xff] ^ s_Te3[s3 & 0xff]
165         ^ tex1Dfetch(rk_tex, 36);
166     t1 = s_Te0[s1 >> 24] ^ s_Te1[(s2 >> 16) & 0xff]
167         ^ s_Te2[(s3 >> 8) & 0xff] ^ s_Te3[s0 & 0xff]
168         ^ tex1Dfetch(rk_tex, 37);
169     t2 = s_Te0[s2 >> 24] ^ s_Te1[(s3 >> 16) & 0xff]
170         ^ s_Te2[(s0 >> 8) & 0xff] ^ s_Te3[s1 & 0xff]
171         ^ tex1Dfetch(rk_tex, 38);
172     t3 = s_Te0[s3 >> 24] ^ s_Te1[(s0 >> 16) & 0xff]
173         ^ s_Te2[(s1 >> 8) & 0xff] ^ s_Te3[s2 & 0xff]
174         ^ tex1Dfetch(rk_tex, 39);
175 
176     ct[4 * tid] = (s_Te4[(t0 >> 24)] & 0xff000000)
177         ^ (s_Te4[(t1 >> 16) & 0xff] & 0x00ff0000)
178         ^ (s_Te4[(t2 >> 8) & 0xff] & 0x0000ff00)
179         ^ (s_Te4[(t3) & 0xff] & 0x000000ff) ^ tex1Dfetch(rk_tex, 40);
180     ct[4 * tid + 1] = (s_Te4[(t1 >> 24)] & 0xff000000)
181         ^ (s_Te4[(t2 >> 16) & 0xff] & 0x00ff0000)
182         ^ (s_Te4[(t3 >> 8) & 0xff] & 0x0000ff00)
183         ^ (s_Te4[(t0) & 0xff] & 0x000000ff) ^ tex1Dfetch(rk_tex, 41);
184     ct[4 * tid + 2] = (s_Te4[(t2 >> 24)] & 0xff000000)
185         ^ (s_Te4[(t3 >> 16) & 0xff] & 0x00ff0000)
186         ^ (s_Te4[(t0 >> 8) & 0xff] & 0x0000ff00)
187         ^ (s_Te4[(t1) & 0xff] & 0x000000ff) ^ tex1Dfetch(rk_tex, 42);
188     ct[4 * tid + 3] = (s_Te4[(t3 >> 24)] & 0xff000000)
189         ^ (s_Te4[(t0 >> 16) & 0xff] & 0x00ff0000)
190         ^ (s_Te4[(t1 >> 8) & 0xff] & 0x0000ff00)
191         ^ (s_Te4[(t2) & 0xff] & 0x000000ff) ^ tex1Dfetch(rk_tex, 43);
192     tid += blockDim.x;
193   }
194   return;
195 }
196 void AES_Enc(unint * pt, unint * ct, unint * rk, unint nk, unint pt_size,
197     unint threads_per_block) {
198 #ifdef TEMPO
199   cudaEvent_t tstart, tstop;
200   HANDLE_ERROR(cudaEventCreate(&tstart));
201   HANDLE_ERROR(cudaEventCreate(&tstop));
202 
```

```
203   cudaEvent_t cstart, cstop;
204   HANDLE_ERROR(cudaEventCreate(&cstart));
205   HANDLE_ERROR(cudaEventCreate(&cstop));
206
207   cudaEvent_t start, stop;
208   HANDLE_ERROR(cudaEventCreate(&start));
209   HANDLE_ERROR(cudaEventCreate(&stop));
210
211   float elapsedTime;
212 #endif
213
214   unint *d_pt, *d_ct, *d_rk;
215
216   unint N = pt_size / 16;
217   unint nr = NR(nk);
218   unint nkx = 4 * (nr + 1);
219   unint nkxb = nkx * sizeof(unint);
220
221 #ifdef TEMPO
222   HANDLE_ERROR(cudaEventRecord(tstart, 0));
223 #endif
224
225   HANDLE_ERROR(cudaMalloc((void **)&d_pt, pt_size));
226   HANDLE_ERROR(cudaMalloc((void **)&d_ct, pt_size));
227   HANDLE_ERROR(cudaMalloc((void **)&d_rk, nkxb));
228
229 #ifdef TEMPO
230   HANDLE_ERROR(cudaEventRecord(cstart, 0));
231 #endif
232
233   HANDLE_ERROR(cudaMemcpy(d_rk, rk, nkxb, cudaMemcpyHostToDevice));
234   HANDLE_ERROR(cudaMemcpy(d_pt, pt, pt_size, cudaMemcpyHostToDevice));
235   HANDLE_ERROR(cudaBindTexture(NULL, rk_tex, d_rk,nkxb));
236
237 #ifdef TEMPO
238   HANDLE_ERROR(cudaEventRecord(start, 0));
239 #endif
240   switch (nk) {
241   case 4:
242     AES_Enc128 <<< 1, threads_per_block >>> (d_pt, d_ct,N);
243     break;
244   case 6:
245     AES_Enc192 <<< 1, threads_per_block >>> (d_pt, d_ct,N);
246     break;
247   case 8:
248     AES_Enc256 <<< 1, threads_per_block >>> (d_pt, d_ct,N);
249     break;
250   default:
251     return;
252   }
253 #ifdef TEMPO
254   HANDLE_ERROR(cudaEventRecord(stop, 0));
255 #endif
256
257   HANDLE_ERROR(cudaMemcpy(ct, d_ct, pt_size, cudaMemcpyDeviceToHost));
258 #ifdef TEMPO
259   HANDLE_ERROR(cudaEventRecord(cstop, 0));
260 #endif
261   HANDLE_ERROR(cudaUnbindTexture(rk_tex));
262   HANDLE_ERROR(cudaFree(d_pt));
```

```
263   HANDLE_ERROR(cudaFree(d_ct));
264   HANDLE_ERROR(cudaFree(d_rk));
265
266 #ifdef TEMPO
267   HANDLE_ERROR(cudaEventRecord(tstop, 0));
268   HANDLE_ERROR(cudaEventSynchronize(tstop));
269
270   HANDLE_ERROR(cudaEventElapsedTime(&elapsedTime, start, stop));
271   fprintf(stderr, "time(enc_cuda_%d):\t\t%12.6f ms\t%d threads\t%u bytes\n",
272       (4 * nk * 8), elapsedTime,threads_per_block,pt_size);
273
274   HANDLE_ERROR(cudaEventElapsedTime(&elapsedTime, cstart, cstop));
275   fprintf(stderr, "time(enc_cuda_cp_%d):\t%12.6f ms\t%d threads\t%u bytes\n",
276       (4 * nk * 8), elapsedTime,threads_per_block,pt_size);
277
278   HANDLE_ERROR(cudaEventElapsedTime(&elapsedTime, tstart, tstop));
279   fprintf(stderr, "time(enc_cuda_tt_%d):\t%12.6f ms\t%d threads\t%u bytes\n",
280       (4 * nk * 8), elapsedTime,threads_per_block,pt_size);
281
282   HANDLE_ERROR(cudaEventDestroy(start));
283   HANDLE_ERROR(cudaEventDestroy(stop));
284
285   HANDLE_ERROR(cudaEventDestroy(cstart));
286   HANDLE_ERROR(cudaEventDestroy(cstop));
287
288   HANDLE_ERROR(cudaEventDestroy(tstart));
289   HANDLE_ERROR(cudaEventDestroy(tstop));
290 #endif
291 }
```

II.14 cuda_aes_nb_nt_shared_texture

Código fonte II.24: cuda_aes_nb_nt_shared_texture/aes.cu

```
1  texture<unsigned, 1, cudaReadModeElementType> rk_tex;
2  __global__ void AES_Enc128(unint * pt, unint * ct, unint N) {
3    unint tid = blockIdx.y * gridDim.x * blockDim.x + blockIdx.x * blockDim.x
4        + threadIdx.x;
5    unint tx = threadIdx.x;
6
7    __shared__ unint s_Te0[256 + 1];
8    __shared__ unint s_Te1[256 + 1];
9    __shared__ unint s_Te2[256 + 1];
10   __shared__ unint s_Te3[256 + 1];
11   __shared__ unint s_Te4[256 + 1];
12
13   for (int i = tx; i < 64; i += blockDim.x) {
14     s_Te0[4 * i] = d_Te0[4 * i];
15     s_Te0[4 * i + 1] = d_Te0[4 * i + 1];
16     s_Te0[4 * i + 2] = d_Te0[4 * i + 2];
17     s_Te0[4 * i + 3] = d_Te0[4 * i + 3];
18
19     s_Te1[4 * i] = d_Te1[4 * i];
20     s_Te1[4 * i + 1] = d_Te1[4 * i + 1];
21     s_Te1[4 * i + 2] = d_Te1[4 * i + 2];
22     s_Te1[4 * i + 3] = d_Te1[4 * i + 3];
23
```

```
24      s_Te2[4 * i] = d_Te2[4 * i];
25      s_Te2[4 * i + 1] = d_Te2[4 * i + 1];
26      s_Te2[4 * i + 2] = d_Te2[4 * i + 2];
27      s_Te2[4 * i + 3] = d_Te2[4 * i + 3];
28
29      s_Te3[4 * i] = d_Te3[4 * i];
30      s_Te3[4 * i + 1] = d_Te3[4 * i + 1];
31      s_Te3[4 * i + 2] = d_Te3[4 * i + 2];
32      s_Te3[4 * i + 3] = d_Te3[4 * i + 3];
33
34      s_Te4[4 * i] = d_Te4[4 * i];
35      s_Te4[4 * i + 1] = d_Te4[4 * i + 1];
36      s_Te4[4 * i + 2] = d_Te4[4 * i + 2];
37      s_Te4[4 * i + 3] = d_Te4[4 * i + 3];
38    }
39    __syncthreads();
40
41    uint s0, s1, s2, s3, t0, t1, t2, t3;
42
43    if (tid < N) {
44
45      s0 = pt[4 * tid] ^ tex1Dfetch(rk_tex, 0);
46      s1 = pt[4 * tid + 1] ^ tex1Dfetch(rk_tex, 1);
47      s2 = pt[4 * tid + 2] ^ tex1Dfetch(rk_tex, 2);
48      s3 = pt[4 * tid + 3] ^ tex1Dfetch(rk_tex, 3);
49
50      /* round 1: */
51      t0 = s_Te0[s0 >> 24] ^ s_Te1[(s1 >> 16) & 0xff]
52          ^ s_Te2[(s2 >> 8) & 0xff] ^ s_Te3[s3 & 0xff]
53          ^ tex1Dfetch(rk_tex, 4);
54      t1 = s_Te0[s1 >> 24] ^ s_Te1[(s2 >> 16) & 0xff]
55          ^ s_Te2[(s3 >> 8) & 0xff] ^ s_Te3[s0 & 0xff]
56          ^ tex1Dfetch(rk_tex, 5);
57      t2 = s_Te0[s2 >> 24] ^ s_Te1[(s3 >> 16) & 0xff]
58          ^ s_Te2[(s0 >> 8) & 0xff] ^ s_Te3[s1 & 0xff]
59          ^ tex1Dfetch(rk_tex, 6);
60      t3 = s_Te0[s3 >> 24] ^ s_Te1[(s0 >> 16) & 0xff]
61          ^ s_Te2[(s1 >> 8) & 0xff] ^ s_Te3[s2 & 0xff]
62          ^ tex1Dfetch(rk_tex, 7);
63
64      /* round 2: */
65      s0 = s_Te0[t0 >> 24] ^ s_Te1[(t1 >> 16) & 0xff]
66          ^ s_Te2[(t2 >> 8) & 0xff] ^ s_Te3[t3 & 0xff]
67          ^ tex1Dfetch(rk_tex, 8);
68      s1 = s_Te0[t1 >> 24] ^ s_Te1[(t2 >> 16) & 0xff]
69          ^ s_Te2[(t3 >> 8) & 0xff] ^ s_Te3[t0 & 0xff]
70          ^ tex1Dfetch(rk_tex, 9);
71      s2 = s_Te0[t2 >> 24] ^ s_Te1[(t3 >> 16) & 0xff]
72          ^ s_Te2[(t0 >> 8) & 0xff] ^ s_Te3[t1 & 0xff]
73          ^ tex1Dfetch(rk_tex, 10);
74      s3 = s_Te0[t3 >> 24] ^ s_Te1[(t0 >> 16) & 0xff]
75          ^ s_Te2[(t1 >> 8) & 0xff] ^ s_Te3[t2 & 0xff]
76          ^ tex1Dfetch(rk_tex, 11);
77
78      /* round 3: */
79      t0 = s_Te0[s0 >> 24] ^ s_Te1[(s1 >> 16) & 0xff]
80          ^ s_Te2[(s2 >> 8) & 0xff] ^ s_Te3[s3 & 0xff]
81          ^ tex1Dfetch(rk_tex, 12);
82      t1 = s_Te0[s1 >> 24] ^ s_Te1[(s2 >> 16) & 0xff]
83          ^ s_Te2[(s3 >> 8) & 0xff] ^ s_Te3[s0 & 0xff]
```

```
84          ^ tex1Dfetch(rk_tex, 13);
85      t2 = s_Te0[s2 >> 24] ^ s_Te1[(s3 >> 16) & 0xff]
86          ^ s_Te2[(s0 >> 8) & 0xff] ^ s_Te3[s1 & 0xff]
87          ^ tex1Dfetch(rk_tex, 14);
88      t3 = s_Te0[s3 >> 24] ^ s_Te1[(s0 >> 16) & 0xff]
89          ^ s_Te2[(s1 >> 8) & 0xff] ^ s_Te3[s2 & 0xff]
90          ^ tex1Dfetch(rk_tex, 15);
91
92      /* round 4: */
93      s0 = s_Te0[t0 >> 24] ^ s_Te1[(t1 >> 16) & 0xff]
94          ^ s_Te2[(t2 >> 8) & 0xff] ^ s_Te3[t3 & 0xff]
95          ^ tex1Dfetch(rk_tex, 16);
96      s1 = s_Te0[t1 >> 24] ^ s_Te1[(t2 >> 16) & 0xff]
97          ^ s_Te2[(t3 >> 8) & 0xff] ^ s_Te3[t0 & 0xff]
98          ^ tex1Dfetch(rk_tex, 17);
99      s2 = s_Te0[t2 >> 24] ^ s_Te1[(t3 >> 16) & 0xff]
100         ^ s_Te2[(t0 >> 8) & 0xff] ^ s_Te3[t1 & 0xff]
101         ^ tex1Dfetch(rk_tex, 18);
102     s3 = s_Te0[t3 >> 24] ^ s_Te1[(t0 >> 16) & 0xff]
103         ^ s_Te2[(t1 >> 8) & 0xff] ^ s_Te3[t2 & 0xff]
104         ^ tex1Dfetch(rk_tex, 19);
105
106     /* round 5: */
107     t0 = s_Te0[s0 >> 24] ^ s_Te1[(s1 >> 16) & 0xff]
108         ^ s_Te2[(s2 >> 8) & 0xff] ^ s_Te3[s3 & 0xff]
109         ^ tex1Dfetch(rk_tex, 20);
110     t1 = s_Te0[s1 >> 24] ^ s_Te1[(s2 >> 16) & 0xff]
111         ^ s_Te2[(s3 >> 8) & 0xff] ^ s_Te3[s0 & 0xff]
112         ^ tex1Dfetch(rk_tex, 21);
113     t2 = s_Te0[s2 >> 24] ^ s_Te1[(s3 >> 16) & 0xff]
114         ^ s_Te2[(s0 >> 8) & 0xff] ^ s_Te3[s1 & 0xff]
115         ^ tex1Dfetch(rk_tex, 22);
116     t3 = s_Te0[s3 >> 24] ^ s_Te1[(s0 >> 16) & 0xff]
117         ^ s_Te2[(s1 >> 8) & 0xff] ^ s_Te3[s2 & 0xff]
118         ^ tex1Dfetch(rk_tex, 23);
119
120     /* round 6: */
121     s0 = s_Te0[t0 >> 24] ^ s_Te1[(t1 >> 16) & 0xff]
122         ^ s_Te2[(t2 >> 8) & 0xff] ^ s_Te3[t3 & 0xff]
123         ^ tex1Dfetch(rk_tex, 24);
124     s1 = s_Te0[t1 >> 24] ^ s_Te1[(t2 >> 16) & 0xff]
125         ^ s_Te2[(t3 >> 8) & 0xff] ^ s_Te3[t0 & 0xff]
126         ^ tex1Dfetch(rk_tex, 25);
127     s2 = s_Te0[t2 >> 24] ^ s_Te1[(t3 >> 16) & 0xff]
128         ^ s_Te2[(t0 >> 8) & 0xff] ^ s_Te3[t1 & 0xff]
129         ^ tex1Dfetch(rk_tex, 26);
130     s3 = s_Te0[t3 >> 24] ^ s_Te1[(t0 >> 16) & 0xff]
131         ^ s_Te2[(t1 >> 8) & 0xff] ^ s_Te3[t2 & 0xff]
132         ^ tex1Dfetch(rk_tex, 27);
133
134     /* round 7: */
135     t0 = s_Te0[s0 >> 24] ^ s_Te1[(s1 >> 16) & 0xff]
136         ^ s_Te2[(s2 >> 8) & 0xff] ^ s_Te3[s3 & 0xff]
137         ^ tex1Dfetch(rk_tex, 28);
138     t1 = s_Te0[s1 >> 24] ^ s_Te1[(s2 >> 16) & 0xff]
139         ^ s_Te2[(s3 >> 8) & 0xff] ^ s_Te3[s0 & 0xff]
140         ^ tex1Dfetch(rk_tex, 29);
141     t2 = s_Te0[s2 >> 24] ^ s_Te1[(s3 >> 16) & 0xff]
142         ^ s_Te2[(s0 >> 8) & 0xff] ^ s_Te3[s1 & 0xff]
143         ^ tex1Dfetch(rk_tex, 30);
```

```
144     t3 = s_Te0[s3 >> 24] ^ s_Te1[(s0 >> 16) & 0xff]
145         ^ s_Te2[(s1 >> 8) & 0xff] ^ s_Te3[s2 & 0xff]
146         ^ tex1Dfetch(rk_tex, 31);
147 
148     /* round 8: */
149     s0 = s_Te0[t0 >> 24] ^ s_Te1[(t1 >> 16) & 0xff]
150         ^ s_Te2[(t2 >> 8) & 0xff] ^ s_Te3[t3 & 0xff]
151         ^ tex1Dfetch(rk_tex, 32);
152     s1 = s_Te0[t1 >> 24] ^ s_Te1[(t2 >> 16) & 0xff]
153         ^ s_Te2[(t3 >> 8) & 0xff] ^ s_Te3[t0 & 0xff]
154         ^ tex1Dfetch(rk_tex, 33);
155     s2 = s_Te0[t2 >> 24] ^ s_Te1[(t3 >> 16) & 0xff]
156         ^ s_Te2[(t0 >> 8) & 0xff] ^ s_Te3[t1 & 0xff]
157         ^ tex1Dfetch(rk_tex, 34);
158     s3 = s_Te0[t3 >> 24] ^ s_Te1[(t0 >> 16) & 0xff]
159         ^ s_Te2[(t1 >> 8) & 0xff] ^ s_Te3[t2 & 0xff]
160         ^ tex1Dfetch(rk_tex, 35);
161 
162     /* round 9: */
163     t0 = s_Te0[s0 >> 24] ^ s_Te1[(s1 >> 16) & 0xff]
164         ^ s_Te2[(s2 >> 8) & 0xff] ^ s_Te3[s3 & 0xff]
165         ^ tex1Dfetch(rk_tex, 36);
166     t1 = s_Te0[s1 >> 24] ^ s_Te1[(s2 >> 16) & 0xff]
167         ^ s_Te2[(s3 >> 8) & 0xff] ^ s_Te3[s0 & 0xff]
168         ^ tex1Dfetch(rk_tex, 37);
169     t2 = s_Te0[s2 >> 24] ^ s_Te1[(s3 >> 16) & 0xff]
170         ^ s_Te2[(s0 >> 8) & 0xff] ^ s_Te3[s1 & 0xff]
171         ^ tex1Dfetch(rk_tex, 38);
172     t3 = s_Te0[s3 >> 24] ^ s_Te1[(s0 >> 16) & 0xff]
173         ^ s_Te2[(s1 >> 8) & 0xff] ^ s_Te3[s2 & 0xff]
174         ^ tex1Dfetch(rk_tex, 39);
175 
176     ct[4 * tid] = (s_Te4[(t0 >> 24)] & 0xff000000)
177         ^ (s_Te4[(t1 >> 16) & 0xff] & 0x00ff0000)
178         ^ (s_Te4[(t2 >> 8) & 0xff] & 0x0000ff00)
179         ^ (s_Te4[(t3) & 0xff] & 0x000000ff) ^ tex1Dfetch(rk_tex, 40);
180     ct[4 * tid + 1] = (s_Te4[(t1 >> 24)] & 0xff000000)
181         ^ (s_Te4[(t2 >> 16) & 0xff] & 0x00ff0000)
182         ^ (s_Te4[(t3 >> 8) & 0xff] & 0x0000ff00)
183         ^ (s_Te4[(t0) & 0xff] & 0x000000ff) ^ tex1Dfetch(rk_tex, 41);
184     ct[4 * tid + 2] = (s_Te4[(t2 >> 24)] & 0xff000000)
185         ^ (s_Te4[(t3 >> 16) & 0xff] & 0x00ff0000)
186         ^ (s_Te4[(t0 >> 8) & 0xff] & 0x0000ff00)
187         ^ (s_Te4[(t1) & 0xff] & 0x000000ff) ^ tex1Dfetch(rk_tex, 42);
188     ct[4 * tid + 3] = (s_Te4[(t3 >> 24)] & 0xff000000)
189         ^ (s_Te4[(t0 >> 16) & 0xff] & 0x00ff0000)
190         ^ (s_Te4[(t1 >> 8) & 0xff] & 0x0000ff00)
191         ^ (s_Te4[(t2) & 0xff] & 0x000000ff) ^ tex1Dfetch(rk_tex, 43);
192   }
193   return;
194 }
195 void AES_Enc(unint * pt, unint * ct, unint * rk, unint nk, unint pt_size,
196     unint threads_per_block) {
197 #ifdef TEMPO
198   cudaEvent_t tstart, tstop;
199   HANDLE_ERROR(cudaEventCreate(&tstart));
200   HANDLE_ERROR(cudaEventCreate(&tstop));
201 
202   cudaEvent_t cstart, cstop;
203   HANDLE_ERROR(cudaEventCreate(&cstart));
```

```
204   HANDLE_ERROR(cudaEventCreate(&cstop));
205
206   cudaEvent_t start, stop;
207   HANDLE_ERROR(cudaEventCreate(&start));
208   HANDLE_ERROR(cudaEventCreate(&stop));
209
210   float elapsedTime;
211 #endif
212
213   unint *d_pt, *d_ct, *d_rk;
214
215   unint N = pt_size / 16;
216   unint nr = NR(nk);
217   unint nkx = 4 * (nr + 1);
218   unint nkxb = nkx * sizeof(unint);
219
220   cudaDeviceProp prop;
221   HANDLE_ERROR( cudaGetDeviceProperties( &prop,0));
222
223   if (threads_per_block > prop.maxThreadsPerBlock)
224     threads_per_block = prop.maxThreadsPerBlock;
225
226   unint blocks = (unint) N / threads_per_block;
227
228   if (N % threads_per_block != 0)
229     blocks++;
230   unint blocks1 = blocks;
231   unint blocks2 = 1;
232
233   // Blocks transformado em bidimensional se ultrapassar maximo
234   unint maxgridsize=prop.maxGridSize[0];
235   if (blocks > maxgridsize) {
236     unint max = (unint) sqrt(blocks);
237     blocks1 = blocks;
238     blocks2 = 1;
239     unint i = 0;
240     // procura divisores de blocks
241     while (primo[i] <= max) {
242       if (blocks1 % primo[i] == 0) {
243         blocks1 = blocks1 / primo[i];
244         blocks2 = blocks2*primo[i];
245         if (blocks1 > maxgridsize) continue;
246         break;
247       }
248       i++;
249     }
250     //se nao encontrar divisores tentar minimizar blocks1 e blocks2
251     if ((blocks1 > maxgridsize)||(blocks2 > maxgridsize)) {
252       blocks1 = max;
253       blocks2 = max;
254       while (blocks1 * blocks2 * threads_per_block < N)
255         blocks2++;
256     }
257   }
258   dim3 BLOCKS(blocks1, blocks2, 1);
259
260 #ifdef TEMPO
261   HANDLE_ERROR(cudaEventRecord(tstart, 0));
262 #endif
263
```

```
264   HANDLE_ERROR(cudaMalloc((void **)&d_pt, pt_size));
265   HANDLE_ERROR(cudaMalloc((void **)&d_ct, pt_size));
266   HANDLE_ERROR(cudaMalloc((void **)&d_rk, nkxb));
267
268 #ifdef TEMPO
269   HANDLE_ERROR(cudaEventRecord(cstart, 0));
270 #endif
271   HANDLE_ERROR(cudaMemcpy(d_rk, rk, nkxb, cudaMemcpyHostToDevice));
272   HANDLE_ERROR(cudaMemcpy(d_pt, pt, pt_size, cudaMemcpyHostToDevice));
273   HANDLE_ERROR(cudaBindTexture(NULL, rk_tex, d_rk,nkxb));
274
275 #ifdef TEMPO
276   HANDLE_ERROR(cudaEventRecord(start, 0));
277 #endif
278   switch (nk) {
279   case 4:
280     AES_Enc128 <<< BLOCKS, threads_per_block >>> (d_pt, d_ct,N);
281     break;
282   case 6:
283     AES_Enc192 <<< BLOCKS, threads_per_block >>> (d_pt, d_ct,N);
284     break;
285   case 8:
286     AES_Enc256 <<< BLOCKS, threads_per_block >>> (d_pt, d_ct,N);
287     break;
288   default:
289     return;
290   }
291 #ifdef TEMPO
292   HANDLE_ERROR(cudaEventRecord(stop, 0));
293 #endif
294
295   HANDLE_ERROR(cudaMemcpy(ct, d_ct, pt_size, cudaMemcpyDeviceToHost));
296
297 #ifdef TEMPO
298   HANDLE_ERROR(cudaEventRecord(cstop, 0));
299 #endif
300
301   HANDLE_ERROR(cudaUnbindTexture(rk_tex));
302   HANDLE_ERROR(cudaFree(d_pt));
303   HANDLE_ERROR(cudaFree(d_ct));
304   HANDLE_ERROR(cudaFree(d_rk));
305
306 #ifdef TEMPO
307   HANDLE_ERROR(cudaEventRecord(tstop, 0));
308
309   HANDLE_ERROR(cudaEventSynchronize(tstop));
310
311   HANDLE_ERROR(cudaEventElapsedTime(&elapsedTime, start, stop));
312   fprintf(stderr, "time(enc_cuda_%d):\t\t%12.6f ms\t%4d threads\t%u bytes\n",
313       (4 * nk * 8), elapsedTime,threads_per_block,pt_size);
314
315   HANDLE_ERROR(cudaEventElapsedTime(&elapsedTime, cstart, cstop));
316   fprintf(stderr, "time(enc_cuda_cp_%d):\t%12.6f ms\t%4d threads\t%u bytes\n",
317       (4 * nk * 8), elapsedTime,threads_per_block,pt_size);
318
319   HANDLE_ERROR(cudaEventElapsedTime(&elapsedTime, tstart, tstop));
320   fprintf(stderr, "time(enc_cuda_tt_%d):\t%12.6f ms\t%4d threads\t%u bytes\n",
321       (4 * nk * 8), elapsedTime,threads_per_block,pt_size);
322
323   HANDLE_ERROR(cudaEventDestroy(start));
```

```
324     HANDLE_ERROR(cudaEventDestroy(stop));
325
326     HANDLE_ERROR(cudaEventDestroy(cstart));
327     HANDLE_ERROR(cudaEventDestroy(cstop));
328
329     HANDLE_ERROR(cudaEventDestroy(tstart));
330     HANDLE_ERROR(cudaEventDestroy(tstop));
331 #endif
332 }
```