CENTRO UNIVERSITÁRIO EURÍPIDES DE MARÍLIA
FUNDAÇÃO DE ENSINO “EURÍPIDES SOARES DA ROCHA”
BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

COMPARAÇÃO DE DESEMPENHO ENTRE DIFERENTES IMPLEMENTAÇÕES DO ALGORITMO KECCAK PARA PLATAFORMAS GPGPUS UTILIZANDO OPENCL

ALLAN MARIANO DE SOUZA

Marília, 2013

CENTRO UNIVERSITÁRIO EURÍPIDES DE MARÍLIA
FUNDAÇÃO DE ENSINO "EURÍPIDES SOARES DA ROCHA"
BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

COMPARAÇÃO DE DESEMPENHO ENTRE DIFERENTES IMPLEMENTAÇÕES DO ALGORITMO KECCAK PARA PLATAFORMAS GPGPUS UTILIZANDO OPENCL

Monografia apresentada ao Centro Universitário Eurípides de Marília como parte dos requisitos necessários para a obtenção do grau de Bacharel em Ciência da Computação.
Orientador: Prof. Fábio Dacêncio Pereira

Marília, 2013

CENTRO UNIVERSITÁRIO EURÍPIDES DE MARÍLIA
BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

TRABALHO DE CONCLUSÃO DE CURSO – AVALIAÇÃO FINAL

Allan Mariano de Souza

Comparação de Desempenho Entre Diferentes Implementações do Algoritmo Keccak para Plataformas GPGPU utilizando OpenCL

Banca examinadora da monografia apresentada ao Curso de Bacharelado em Ciência da Computação do UNIVEM/F.E.E.S.R., para obtenção do Título de Bacharel em Ciência da Computação.

Nota: 10 (Dez)

Orientador: Fábio Dacêncio Pereira

1°. Examinador: Giulianna Marega Marques

2°. Examinador: Ricardo José Sabatine

Marília, 05 de dezembro de 2013.

DEDICATÓRIA

Dedico este trabalho a todos os meus familiares, especialmente aos meus pais, não só por estarem sempre ao meu lado, mas também, pelo apoio, confiança, sacrifícios realizados e a motivação para estar sempre seguindo em frente. Sem eles não teria conseguido mais essa conquista.

Essa conquista não é só minha, é nossa!
Obrigado Ditão e Silviona.

AGRADECIMENTOS

Agradeço a todos os professores do UNIVEM, especialmente ao professor Ms. Rodolfo Barros Chiaramonte e ao professor Ms. Leonardo Castro Botega pelas oportunidades concebidas e, ao professor Dr. Fábio Dacêncio Pereira pelo apoio e orientação nos projetos e pesquisas realizadas e pela confiança e motivação.

Resumo

Usar unidades de processamento gráfico (GPUs) em computação paralela para obter um alto desempenho em aplicações vem se tornando comum, muitas vezes como parte de um sistema heterogêneo. OpenCL é uma API para computação paralela para plataformas heterogêneas, na qual permite ao desenvolvedor executar processos comuns ou massivamente paralelos em GPUs ou qualquer outro tipo de processador. Em 02 de outubro de 2012, o NIST (*National Institute of Standards and Technology*) anunciou o novo algoritmo para a implementação de funções *hash*, o algoritmo Keccak, sendo assim, o novo padrão para geração de funções *hash* (SHA-3). Neste contexto, esse trabalho se propõe a explorar o algoritmo em evidencia realizando diferentes implementações para plataformas heterogenêas (GPGPUs) comparar os resultados das implementações e selecionar uma implementação com melhor desempenho nesta plataforma.

Palavras-chave: OpenCL, Plataformas heterogêneas GPGPUs, Keccak.

Abstract

Using graphics processing units (GPUs) for parallel computing in high performance applications are becoming common, often as part of a heterogeneous system. OpenCL is an API for parallel computing for heterogeneous platforms, which enables developers to perform common or massively parallel processes on GPUs or in any other type of processor. On October 2, 2012, NIST (National Institute of Standards and Technology) announced the new algorithm for implementation of hash functions, the Keccak algorithm, so the new standard for generation of hash functions (SHA-3). In this context, this paper aims to explore the algorithm shows different implementations for heterogeneous platforms (GPGPUs) to compare the results of implementations and select an implementation with better performance on this platform.

Keywords: OpenCL, heterogeneous platforms, GPGPUs, Keccak.

LISTA DE ILUSTRAÇÕES

Figura 1: Comparação da filosofia de design entre CPU e GPU 14

Figura 2: Arquitetura de uma GPU moderna. 16

Figura 3: Comparação de desempenho em GFLOPS entre GPUs e CPUs 16

Figura 4: OpenCL Modelo da plataforma OpenCL.. 20

Figura 5: Modelo de execução OpenCL 21

Figura 6: Modelo de Memória OpenCL 24

Figura 7: Exemplo multiplicação de matrizes + algoritmo 26

Figura 8: Exemplo de cálculo do primeiro elemento da matriz resultante com sub matrizes com work-groups de dimensão 2 x 2 27

Figura 9: As operações das rodadas e suas etapas de Keccak-f[b] 32

Figura 10: Função de Esponja Keccak[r,c] 34

Figura 11: Modelo de execução Keccak em OpenCL das implementações I e II 43

Figura 12: Modelo de execução do Keccak em OpenCL da implementação III 44

Figura 13: Gráfico I, comparação de desempenho parte I 49

Figura 14: Gráfico II, comparação de desempenho parte II 50

Figura 15: Gráfico III, comparação de desempenho parte III 51

LISTA DE TABELAS

Tabela 1: Timeline OpenCL 18
Tabela 2: Regiões de memória – Alocação e acesso 24
Tabela 3: Resultado multiplicação de matrizes em diferentes linguagens 28
Tabela 4: Timeline da competição do NIST 30
Tabela 5: Round Constants de RC[i] 33
Tabela 6: Trabalhos correlatos 36

Sumário

INTRODUÇÃO 11
OBJETIVOS 12
METODOLOGIA 12
CAPÍTULO 1: FUNDAMENTAÇÃO 13
1.1 GPUs e processadores de propósito geral 13
1.2 Arquitetura de uma GPU moderna 15
1.3 GPU de propósito geral e história OpenCL 16
CAPÍTULO 2: OPENCL 19
2.1 Modelo de plataforma 19
2.2 Modelo de execução 20
2.3 Contexto 21
2.4 Filas de comandos 22
2.5 Modelo de memória 23
2.6 Modelo de programação 24
2.7 Aplicação 25
2.7.1 Multiplicação de matrizes 25
CAPÍTULO 3: SHA-3 29
3.1 Competição SHA-3 29
3.2 Timeline da competição 29
3.3 Resultado da competição 31
3.4 Algoritmo Keccak 31
3.5 Função de permutação 31
3.6 Função de esponja 33
3.7 Keccak em OpenCL 35
CAPÍTULO 4: PESQUISA E DESENVOLVIMENTO 36
4.1 Trabalhos Correlatos 36
4.2 Implementações do Keccak em OpenCL 37
4.2.1 Implementação I 38
4.2.2 Implementação II 39
4.2.3 Implementação III 41
4.3 Modelo de execução Keccak em OpenCL 42
4.3.1 Modelo de execução das implementações I e II 42
4.3.2 Modelo de execução da implementação III 44
CAPÍTULO 5: RESULTADOS 46
5.1 Ambiente de teste 46
5.1 Metodologia de testes 46
5.2 Resultados 48
CAPÍTULO 6: CONCLUSÕES 52
6.1 Limitações 53
6.2 Trabalhos futuros 53
REFERÊNCIAS 54
APÊNDICE A: Kernel multiplicação de matrizes 56
APÊNDICE B: Kernel implementação I 58
APÊNDICE C: Kernel implementação II 60
APÊNDICE D: Kernel implementação III 66

INTRODUÇÃO

Nos últimos anos, processadores *multi-core/many-core* estão substituindo os sequenciais. Aumentar paralelismo, ao invés de aumentar a taxa de *clock*, tornou-se a principal forma de aumento de desempenho dos processadores, e essa tendência deverá continuar (GARLAND, 2008).

Particularmente, as GPUs de hoje (*Graphic Processing Units*), vêm superando muitas CPUs, podendo usar centenas de núcleos de processadores paralelos para executar dezenas de milhares de *threads* paralelas (NICKOLLS *et al,* 2010). Pesquisadores e desenvolvedores estão se tornando cada vez mais interessados em aproveitar essa energia para computação de propósito geral, conhecido como GPGPU (*General-Purpose computing on the GPU*) (FANG, 2010), para resolver rapidamente problemas paralelismo.

OpenCL (*Open Computing Language*) é API que permite aos programadores desenvolverem aplicações GPGPU e softwares para processadores massivamente paralelos.

Um dos métodos para garantir a integridade da informação e o uso de funções de *hash*, o qual gera um fluxo de bytes (*hash*) único. Mas a maioria das funções *hash* não podem mais evitar ataques maliciosos e garantir a integridade das informações. A fim de resolver este problema, o Instituto Nacional de Padrões e Tecnologia (NIST) convocou a comunidade cientifica através de um concurso para criar um novo padrão de função *hash*, chamado SHA-3.

O NIST recebeu um *feedback* significativo da comunidade criptográfica. Com base no *feedback* interno e opiniões públicas dos candidatos de segunda ordem, o NIST selecionou cinco finalistas - Blake, Grostl, JH, Keccak e Skein para avançar para a rodada final da competição no dia 9 de dezembro de 2010.

Em 02 de outubro de 2012, o NIST anunciou o vencedor do concurso SHA-3 e o vencedor foi Keccak e agora vai se tornar o oficial algoritmo de *hash* SHA-3 do NIST.

Neste contexto, nesse trabalho de conclusão de curso foi escolhido o algoritmo Keccak vendedor da competição SHA-3, ele apresenta uma estrutura relativamente simples e versátil, o que facilita não só na implementação do mesmo, mas também na compatibilidade de arquiteturas heterogêneas e massivamente paralelas no qual é utilizado no projeto. Este trabalho tem como objetivo estudar o algoritmo vencedor da competição SHA-3 o Keccak e, em seguida, propor e implementar diferentes técnicas de paralelização para sistemas heterogêneos utilizando OpenCL para obter dados de desempenho em GPUs.

OBJETIVOS

Este projeto visa pesquisar e implementar diferentes formas de paralelização do algoritmo SHA-3 Keccak (keccak-f[1600]) para arquiteturas massivamente paralelas GPGPU, utilizando OpenCL para expor o hardware disponível na GPU, buscando explorar seu alto poder de processamento e definir possíveis métricas para a realização de testes apontando assim uma melhor implementação para esse tipo de plataforma. No processo de concepção da aplicação têm-se os seguintes objetivos específicos:

- Estudar arquitetura geral da GPU e consequentemente a arquitetura OpenCL.
- Pesquisar algoritmos e implementações correlatas.
- Estudar e comparar diferentes técnicas de paralelização do algoritmo SHA-3
- Definição da linguagem de implementação.
- Comparação de desempenho entre diferentes implementações.
- Criar uma arquitetura escalável, para otimizar o desempenho.
- Criar exemplos de teste e validação.

METODOLOGIA

O projeto foi divido em quatro etapas principais:

- **Revisão sistemática do tema e pesquisa de trabalhos correlatos**. Foram pesquisados projetos e tecnologias correlatas. E estudado diferentes estruturas e paralelizações do algoritmo em questão para a implementação em GPUs
- **Desenvolvimento e simulação**. Foi definida a linguagem para a implementação do projeto através de medidas de desempenho das linguagens suportadas pela plataforma OpenCL (C/C++, Java, Python), atendendo os requisitos definidos na especificação gerada na primeira etapa.
- **Testes e validação**. Foi selecionada uma arquitetura composta por GPUs e realizada diferentes implementações do algoritmo. Após a implementação métricas para a realização de testes foram definidas e executadas.
- **Estatística de uso.** Foi gerado estatísticas dos resultados obtidos para caracterização das implementações, sendo possível após essa etapa a comparação com outras implementações existentes na literatura.

CAPÍTULO 1: FUNDAMENTAÇÃO

Neste capítulo será apresentada uma introdução sobre GPU (*Graphic Processing Unit*), onde é descrita sua arquitetura, no qual, é feita uma comparação com a arquitetura de uma CPU (*Central Processing Unit*). Neste capítulo também é apresentado e descrito o surgimento das tecnologias GPGPU (*Gerneral Propose computing on GPU*) e um breve histórico da plataforma OpenCL, plataforma que é utilizada neste projeto.

1.1 GPUs e processadores de propósito geral

Unidades de Processamento Gráfico (*Graphic Processing Unit*) são processadores especializados em renderização de gráficos 3D. São usadas em videogames, computadores pessoais, estações de trabalho, chegando a estar presente, hoje em dia, em *smartphones* e *tablets*.

Inicialmente foram desenvolvidas para processar apenas gráficos e por este propósito, são bastante poderosas e eficientes manipulando gráficos computacionais em operações como adição de efeitos, iluminação, suavização de contornos de objetos e criação e processamento de imagens. Devido a não só ao seu alto poder de processamento, mas também por serem massivamente paralelas (*many-core*), as tornam muito mais eficientes do que as CPU's para a execução de algoritmos paralelos, ou, como por exemplo, na otimização de algoritmos genéticos (NVIDIA Corporation, 2010), principalmente quando há grandes quantidades de dados a serem processadas, utilizando a paralelização de dados (*Single Instruction Multiple Data - SIMD*).

Além da facilidade de processar imagens, as GPUs também são usadas em cálculos matemáticos e geométricos, devido a sua capacidade de processar vetores ou matrizes com extrema facilidade. Essa grande velocidade e poder para cálculos matemáticos vêm do fato das GPU's modernas possuírem muito mais processadores, ou *cores*, que as CPUs e executar um grande número de *threads* em paralelo.

A grande diferença de desempenho entre *many-cores* GPUs e *multicore* CPUs de propósito geral, reside na filosofia de design de arquitetura fundamental entre os dois tipos de processadores (HWU *et al*, 2010), como mostrado na Figura 1.

A arquitetura da CPU é otimizada para performance de códigos sequenciais. Ela uma

utiliza lógica de controle para permitir que instruções de uma única *thread* em execução executem em paralelo ou até mesmo fora de sua ordem sequencial, porém, mantendo a aparência de uma execução sequencial (HWU *et al*, 2010). Outro design importante da arquitetura da CPU são as grandes memórias cache, que são proporcionadas para reduzir as latências de instruções e de acesso aos dados de grandes aplicações. Entretanto, não só a lógica de controle, mas também grandes memórias cache não contribuem para o pico de velocidade de cálculo.

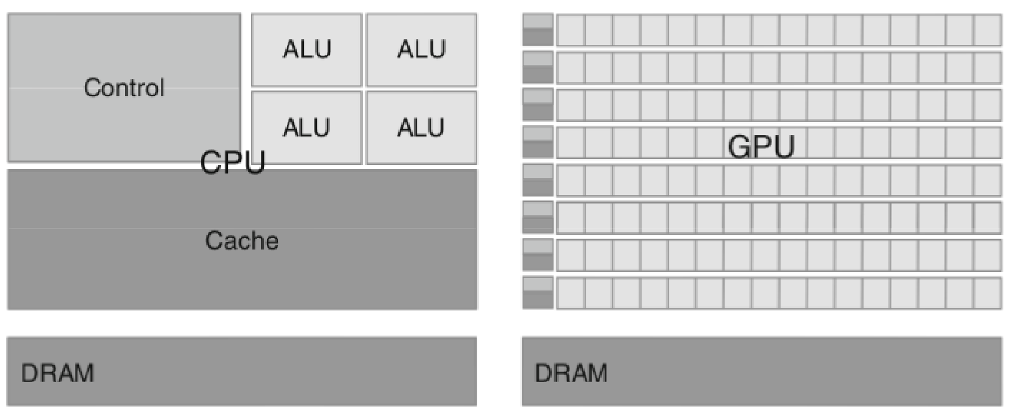

Figura 1: Comparação da filosofia de design entre CPU e GPU

fonte: *HWU* et al, 2010.

O design de arquitetura das GPUs são moldadas pela indústria de videogames em rápido crescimento, que exercem enorme pressão econômica para a capacidade de executar um grande número de cálculos de ponto flutuante por frame de vídeo em jogos avançados. Essa demanda motiva os vendedores GPU à procurar formas de maximizar a área do chip e gerenciamento de energia dedicada a cálculos de ponto flutuante. A solução que prevalece até o momento para otimizar o rendimento, é a de execução de um enorme número de threads em paralelo. O hardware aproveita o grande número de threads em execução para encontrar “trabalho”, quando há threads à espera de acessos à memória de longa latência, minimizando, assim, a lógica de controle necessário para cada thread em execução. Pequenas memórias cache são fornecidos para ajudar a controlar os requisitos de largura de banda desses aplicativos para múltiplas threads que acessam os mesmos dados da memória não precisarem irem todas para a DRAM. Como resultado, a maior área do chip é dedicada aos cálculos de ponto flutuante.

GPUs são concebidas como motores de computação numérica, e não possuem um bom desempenho em algumas tarefas em que CPUs são projetados para obter melhor

desempenho e, portanto, para melhor desempenho de um sistema ou plataforma de computação heterogênea deve-se executar códigos sequenciais na CPU e códigos numericamente intensivos ou paralelos nas GPUs.

1.2 Arquitetura de uma GPU moderna

As GPUs modernas são compostas por grandes quantidades de multiprocessadores ou também conhecidos como *streaming multiprocessors* (SMs), que são formados por pequenos processadores, ou também conhecidos como, *streaming processors* (SPs), organizados em uma série altamente segmentada, onde cada *streaming processor* é capaz de receber uma ou mais *threads* para a execução. Todos os SP (*streaming processor)* em um SM (*streaming multiprocessor)* compartilham lógicas de controle, instruções e cache (HWU *et al*, 2010).

Os multiprocessadores são organizados em blocos, onde o número de multiprocessadores para formar um bloco, pode variar de uma GPU para outra. Os blocos possuem acesso à memória local para compartilhar informações e tomar decisões de controle entre os SMs e, também possuem acesso à memória global (*global memory*), para compartilhar dados com outros blocos.

As GPUs modernas possuem até 6 gigabytes de *graphics double data rate* (GDDR) DRAM ou memória global, porém esta memória é diferente da memória das CPUs, pois a memória da GPU é utilizada apenas como *buffer* dos dados à serem processados pelos *streaming processors*.

A Figura 2 mostra a arquitetura de uma GPU Nvidia G80, o chip possui 128 *streaming processors*, organizados em 16 *streaming multiprocessors*, onde cada SM possui 8 SPs. Os blocos são formados por 2 SMs, somando um total de 8 blocos. Nesta arquitetura cada SMs suporta aproximadamente 768 *threads* somando assim um total de 12.000 *threads,* enquanto uma CPU Intel atual suporta de 2 a 4 *threads* por núcleo, dependendo do modelo de arquitetura (HWU *et al*, 2010).

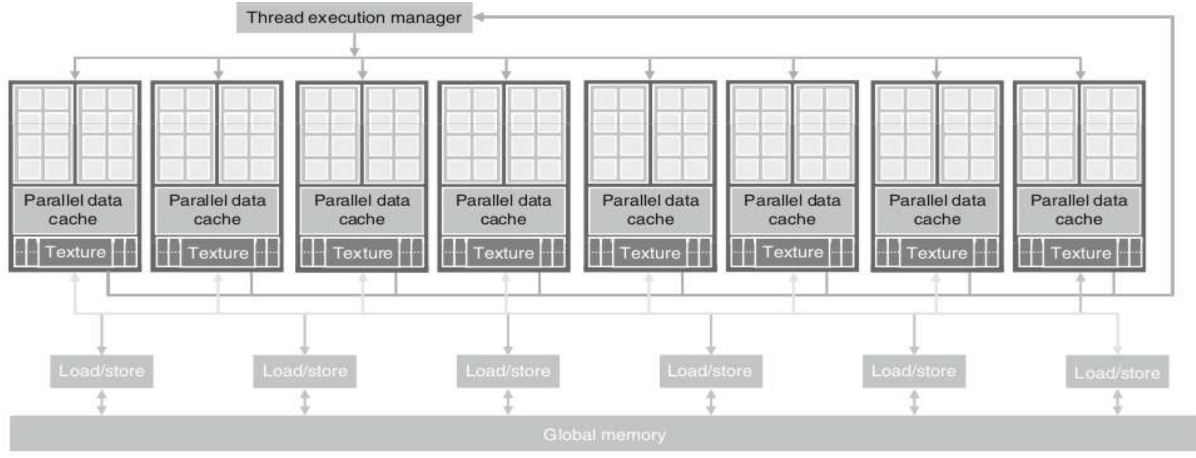

Figura 2: Arquitetura de uma GPU moderna.

fonte: *HWU* et al, 2010.

1.3 GPU de propósito geral e história OpenCL

Impulsionado pela crescente complexidade do processamento gráfico, especificamente em aplicações de jogos, as placas de vídeo passaram por grandes revitalizações tecnológicas e atualmente são massivamente paralelas. A figura **3** exibe uma comparação entre capacidade de processamento de pontos flutuantes (*flops – float points*) das GPUs da NVIDIA, GPUs AMD/ATI e dos processadores da Intel feita em 2009, onde uma GPU Nvidia pode atingir até 1 teraflops (1000 gigaflops), enquanto uma CPU intel atinge apenas 100 gigaflops.

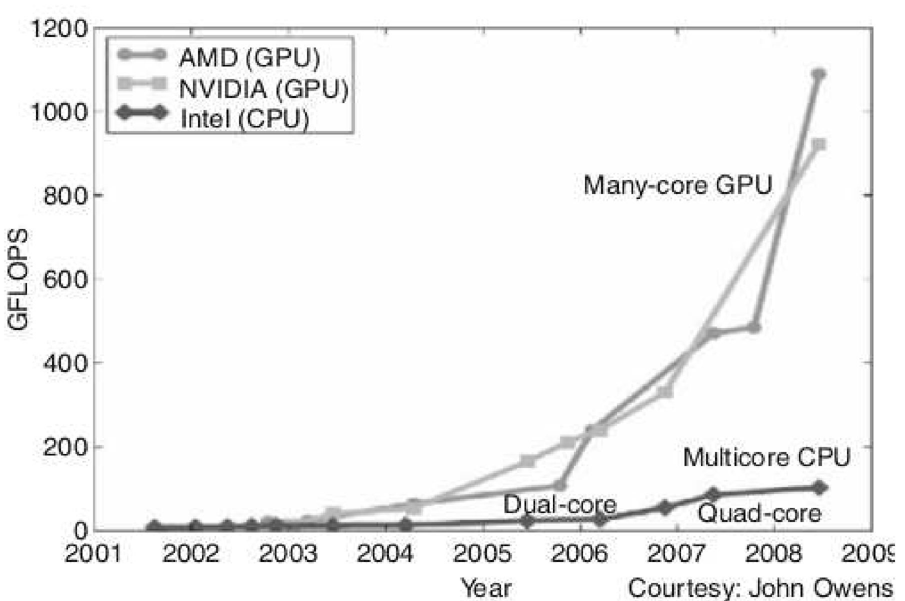

Figura 3: Comparação de desempenho em GFLOPS entre GPUs e CPUs

fonte: *HWU* et al, 2010.

Processadores de múltiplos núcleos, especialmente as GPUs, ou também conhecidas como *many-cores*, lideram a corrida de desempenho de ponto flutuante desde 2003, como visto na figura 3 e incidem sobre o rendimento de execução de aplicações massivamente paralelas (HWU *et al*, 2010). Devido ao seu alto poder de processamento e desempenho as GPUs começaram a adquirir características de propósito geral (GPGPU – *Gerneral Propose computing on GPU*).

A ideia das GPUs adquirirem características de propósito geral surgiu de utilizar seu *hardware* disponível na GPU não só para renderização e processamento de imagens, mas também, para a execução de processos comuns executados na CPU. Com isso foram criadas APIs, plataformas e frameworks para o desenvolvimento dessa tecnologia, assim aliviando a carga de dados à serem processados pela CPU e aumentando o desempenho de aplicações utilizando o alto processamento do hardware disponível na GPU.

De 2003 a 2008, o cenário GPGPU foi fragmentado, com várias soluções proprietárias como, por exemplo, CUDA (*Compute Unified Device Architecture*), que é um modelo de programação paralela criada pela Nvidia, que permite aos desenvolvedores acessarem instruções virtuais e, acesso a memória dos núcleos de processamento paralelos das GPUs Nvidia (NVIDIA, 2010), e AMD/CTM (*Close to Metal*), que proporcionava aos desenvolvedores um acesso completo ao conjunto de instruções nativas e à memória dos elementos de cálculos massivamente paralelos das GPUs AMD (HENSLEY, 2007).

Em 2008 Apple enxerga uma oportunidade, intervem e desenvolve uma interface padronizada para computação GPGPU em diferentes plataformas de hardware, e em julho de 2008, apple propõe uma versão inicial do OpenCL e a submete para o grupo Khronos para a padronização. A Tabela 1 mostra a *timeline* da plataforma OpenCL

Tabela 1: Timeline OpenCL

Data	Versão
Junho 2008	Apple propõe uma versão inicial do OpenCL e a submete ao Khronos para padronização
Dezembro 2008	A especificação OpenCL 1.0 é disponibilizada publicamente
Maio 2009	Testes de conformidade são disponibilizados para homologar implementações
2º Semestre 2009	Diversas implementações são lançadas para uma variedade de plataformas
Junho 2010	OpenCL 1.1 é lançado; uma primeira implementação surge na mesma época
Novembro 2011	É oficialmente disponibilizada a especificação OpenCL 1.2

CAPÍTULO 2: OPENCL

OpenCL (*Open Computing Language*) é um padrão da indústria multi-plataforma e de computação paralela para a programação de aplicações heterogêneas, que podem ser formadas por coleções de CPUs, GPUs e outros dispositivos computacionais organizados em uma única plataforma. OpenCL é um framework para programação paralela e inclui uma linguagem, API e bibliotecas do sistema em tempo de execução para apoiar o desenvolvimento de software (Khronos OpenCL Working Group, 2011).

Programas únicos escritos em OpenCL podem ser executados em uma ampla gama de sistemas, a partir de telefones celulares, computadores portáteis até nós em grandes supercomputadores. Nenhum outro padrão de programação paralela tem um amplo padrão multi-plataforma (MUNSHI A. *et al,* 2011).

A ideia central por trás OpenCL pode ser descrita usando modelos hierárquicos. Modelo de plataforma (apresentado na seção 2.1), modelo de execução (apresentado na seção 2.2), modelo de memória (apresentado na seção 2.5) e modelo de programação (apresentado na seção 2.6).

2.1 Modelo de plataforma

O modelo de plataforma consiste em um *host* que está conectado a um ou mais dispositivos OpenCL (CPUs, GPUs, PDAs), os dispositivos OpenCL são divididos em uma ou mais unidades de computação (UCs), que são divididos em um ou mais elementos de processamento (PEs). Os cálculos executados nos dispositivos OpenCL ocorrem dentro dos elementos de processamento (Khronos OpenCL Working Group, 2011). A Figura 4 ilustra o modelo de plataforma OpenCL que foi descrito.

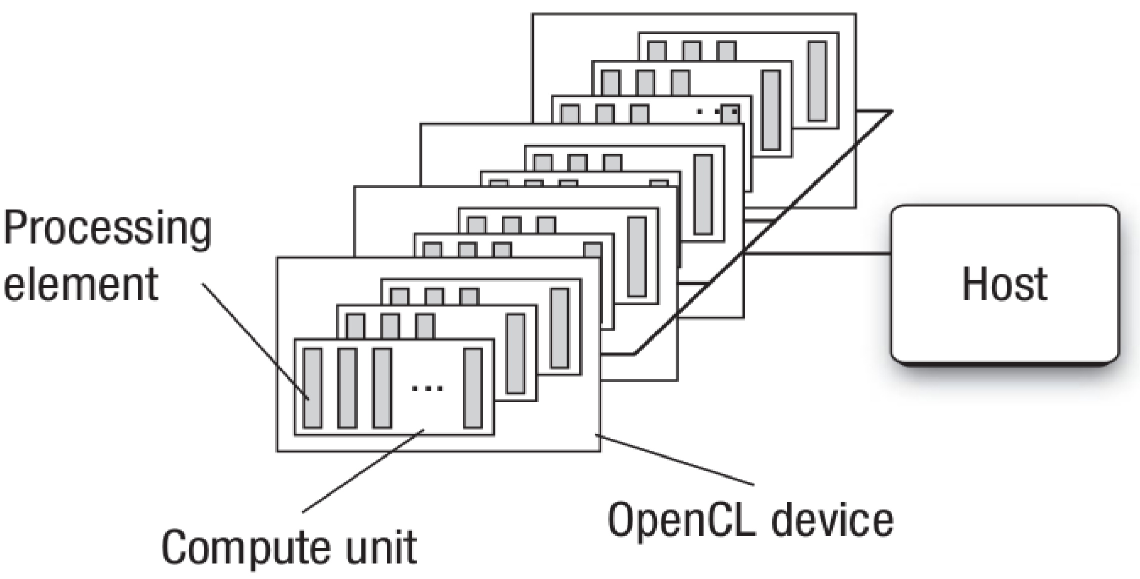

Figura 4: OpenCL Modelo da plataforma OpenCL
fonte: MUNSHI A. *et al*, 2011

2.2 Modelo de execução

A execução de um programa OpenCL ocorre em duas partes: (i) *kernels* que são funções executadas em um ou mais dispositivos OpenCL e (ii) um programa de *host* partes seriais executadas no *host*. O programa de *host* define o contexto e os parâmetros para os *kernels* e também gerencia a sua execução (Khronos OpenCL Working Group, 2011).

O modelo de execução OpenCL é definido pela forma como seus *kernels* são executados. Quando o programa de *host* apresenta um *kernel* para a execução, um espaço de índices é definido chamado *NDRange*, onde estes índices podem ser de uma dimensão (1D), duas dimensões (2D) ou três dimensões (3D). Cada ponto no espaço de índice é chamado de *work-item* e cada *work-item* é uma instância do *kernel* e possuem um índice único (identificador global) para calcular endereços de memória e de tomar decisões de controle.

Os *work-items* são organizados em *work-groups*, fornecem uma decomposição do espaço índice. Aos *work-groups* é atribuído a um identificador de grupo único, com a mesma dimensão do espaço de índice utilizado para os *work-items,* é atribuído também, um identificador único local dentro de cada *work-group* para que cada *work-item* possa ser identificado exclusivamente por seu identificador global ou por uma combinação de seu identificador local e grupo. *Work-items* em um determinado *work-group* executam

concorrentemente sobre os elementos de processamento de uma única unidade de computação (MUNSHI A. *et al,* 2011).

A figura 5 é um exemplo da forma como os identificadores globais, locais e de grupos estão relacionados por um *NDRange* bidimensional. Outros parâmetros do espaço de índice estão definidos na figura. O bloco sombreado tem uma identificação global de (GX, GY) = (6, 5), com um identificação de grupo mais uma identificação local (WX, WY) = (1, 1) e (lx, ly) = (2, 1).

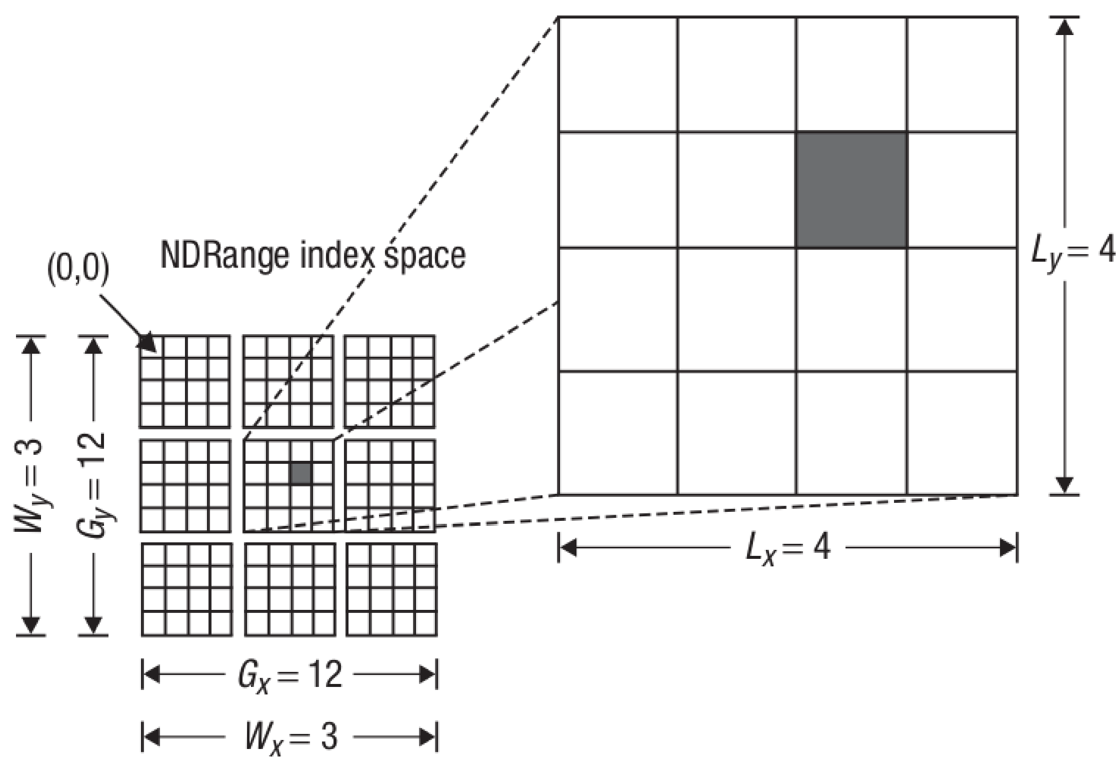

Figura 5: Modelo de execução OpenCL
fonte: MUNSHI A. *et al*, 2011

2.3 Contexto

Todo processamento intensivo ou massivamente paralelo de uma aplicação ocorre no dispositivo OpenCL. Entretanto o *host* possui uma tarefa muito importante durante a implementação e execução de uma aplicação. É no *host* onde são definidos os *kernels* que serão executados pelos dispositivos, onde também é estabelecido um contexto (*context*), um espaço de índice (NDRange) e as filas (*command enqueue*) que controlam os detalhes de como e quando os *kernels* serão executados. Todas estas importantes funções estão definidas na API de especificação OpenCL (SCARPINO, 2012).

A primeira tarefa do programa de *host* é definir o contexto (context) para a aplicação

OpenCL. Como o nome indica, o contexto define o ambiente no qual os *kernels* são definidos e executados. Os termos que definem o contexto são os seguintes:

- ***Devices***: Coleção de dispositivos que serão usados pelo host.
- ***Kernels***: Funções OpenCL que serão executadas nos dispositivos.
- ***Program Objects***: Códigos fonte do programa e executáveis que implementam os *kernels*.
- ***Memory Objects***: Conjunto de objetos na memória que são visíveis para todos os dispositivos OpenCL e contêm valores que podem ser operados por instâncias de um *kernel*

O contexto é criado e manipulado pelo *host* usando funções da API OpenCL (MUNSHI *et al*, 2011).

2.4 Filas de comandos

A interação entre o *host* e o *kernel* é feita através de comandos alocados pelo *host* em um fila de comandos (*command enqueue*). Estes comandos aguardam em uma fila até que sejam executados pelo dispositivo OpenCL. Uma fila de comando (*command enqueue*) é criado pelo *host* porém é vinculada a um único dispositivo após a definição de seu contexto (*context*). O *host* aloca os comandos em uma fila e então, são vinculados para a execução em um dispositivo associado. OpenCL suporta três tipos de comandos:

- **Comandos de execução do *kernel***: Executam um *kernel* sobre os elementos de processamentos (PEs) de um dispositivo OpenCL.
- **Comandos de memória**: Transferem dados entre o *host* e diferentes objetos de memória. Move dados entre os objetos de memória, ou mapeia ou remove objetos do espaço de memória do *host*.
- **Comandos de sincronização**: Colocam restrições sobre a ordem em que os comandos são executados.

Em um programa de *host*, o desenvolvedor define o contexto (*context*), filas de comandos (*command enqueue*), objetos de memória e constrói as estruturas de dados no *host* necessárias para suportar a aplicação. Então o foco mudo para as filas de comandos. Os objetos de memória são movidos a partir do *host* para os dispositivos; argumentos do *kernel*

estão ligados à estes objetos de memória e, em seguida, submetidos para a fila de comandos para a execução. Quando o *kernel* termina seu processamento, objetos de memória são gerados pelos elementos de processamento do dispositivo e podem ser copiados de volta para o *host*, para serem utilizados pela a aplicação (SCARPINO, 2012).

2.5 Modelo de memória

O modelo de memória OpenCL é formado por cinco regiões como: memória de *host*, memória global, memória constante, memória local e memória privada. *Work-items* executando um *kernel* têm acesso à essas cinco regiões de memória (MUNSHI *et al*, 2011).

- **Memória de host:** Esta região de memória só é visível para o *host*. Tal como acontece com a maioria dos detalhes sobre o *host*, OpenCL define apenas como a memória do *host* interage com objetos OpenCL.

- **Memória global:** Esta região de memória permite acesso de leitura e escrita a todos os *work-items* em todos os *work-groups*. Cada *work-item* pode ler ou escrever em qualquer elemento de um objeto de memória. Leitura e escrita para a memória global pode ser armazenada em cache, dependendo das capacidades do dispositivo.

- **Memória constante:** Esta região de memória global permanece constante durante a execução de um *kernel*. O *host* aloca e inicializa objetos de memória colocados na memória constante. *Work-items* têm acesso somente leitura a esses objetos.

- **Memória local:** Esta região de memória é local para cada *work-item*. Esta região de memória pode ser usada para alocar variáveis que são compartilhadas por todos os *work-items* do mesmo *work-group*. Ele pode ser implementado como regiões dedicadas de memória no dispositivo OpenCL. Alternativamente, a região de memória local pode ser mapeado sobre seções da memória global.

- **Memória privada:** Esta região de memória é privada a cada *work-item*. As variáveis definidas na memória particular de um *work-items* não são visíveis para os outros *work-items*.

A Tabela 2, mostra como o host e o kernel alocam e acessam uma região de memória, fazendo uma relação com o tipo de alocação (estática – tempo de compilação vs dinâmica – tempo de execução) e o tipo de acesso permitido, leitura ou escrita (READ/WRITE).

Tabela 2: Regiões de memória – Alocação e acesso

	Global	Constant	Local	Private
Host	Alocação dinâmica. Acesso à leitura e escrita	Alocação dinâmica. Acesso à leitura e escrita	Alocação dinâmica. Sem acesso	Nenhuma alocação. Sem acesso
Kernel	Nenhuma alocação. Acesso à leitura e escrita	Alocação estática. Acesso à leitura	Alocação estática. Acesso à leitura e escrita	Alocação estática. Acesso à leitura e escrita

A Figura 6 mostra um resumo do modelo da memória OpenCL e como as diferentes regiões de memória interagem com o modelo de plataforma.

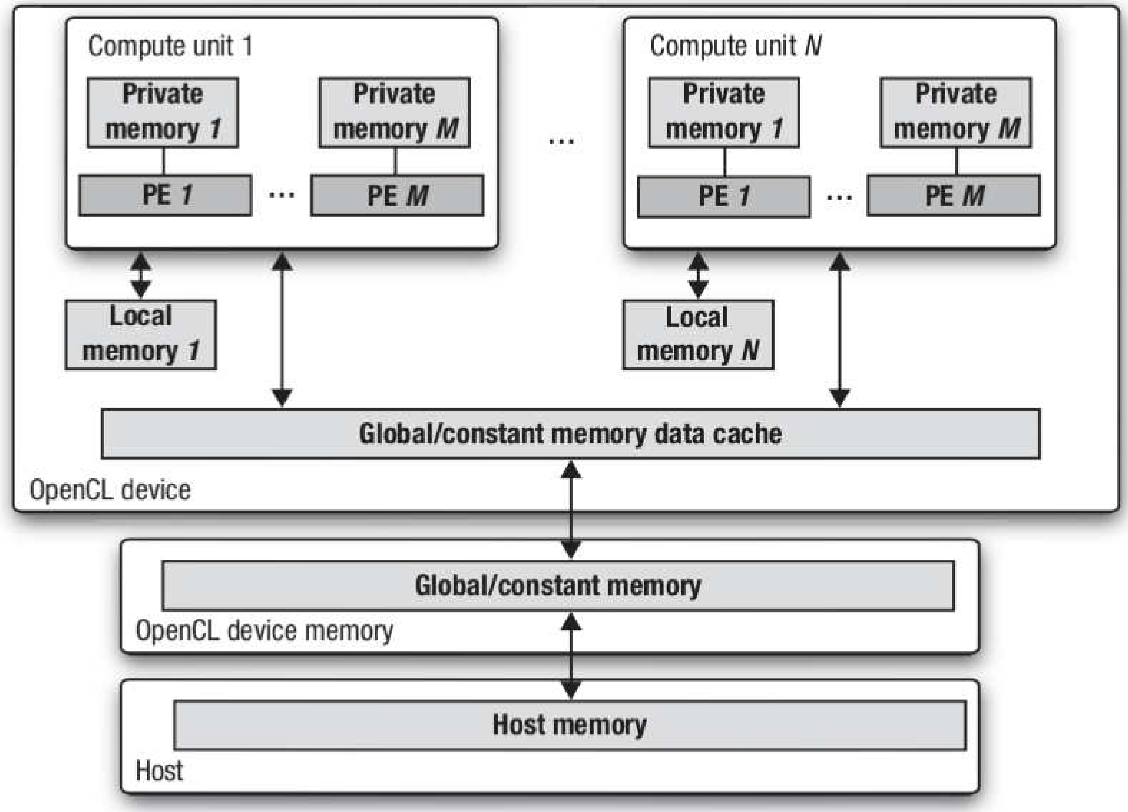

Figura 6: Modelo de Memória OpenCL
fonte: MUNSHI A. *et al*, 2011

2.6 Modelo de programação

OpenCL inclui uma linguagem baseada em C99 para escrever o código do *kernel*, e o programa de *host* pode ser escrito em outras linguagens, tais como: C/C++, Java e Python. O modelo de programação OpenCL suporta paralelismo de dados (*Data Parallel Programing*

Model) e processos (*Task Parallel Programing Model*), bem como implementações híbridas destes dois modelos.

O paralelismo de dados define uma computação em termos de sequencia de instruções aplicadas a múltiplos elementos de um objeto de memória. O espaço de índice associado ao modelo de execução define cada *work-item* e como os dados são mapiados no mesmo.

Neste modelo, cada *work-item* instância um mesmo *kernel* OpenCL, porém os dados acessados e utilizados para as instruções contidas no *kernel* são diferentes para cada *work-item* em execução.

O paralelismo de processos, define um modelo em que uma única instância de um *kernel* OpenCL é executado independente de qualquer espaço de índice. É logicamente equivalente à executar um *kernel* OpenCL é um processador com um *work-group* contendo apenas um work-item. De acordo com esse modelo o paralelismo pode ser utilizado por:

- Usar vetores de tipos de dados implementados pelo dispositivo.
- Enfileirar (*enqueue*) múltiplos processos e / ou.
- Enfileirar (*enqueue*) *kernels* nativos desenvolvidos utilizando um modelo de programação ortogonal ao OpenCL.

2.7 Aplicação

Com o conhecimento adquirido com o estudo da plataforma e de métodos de paralelização, foi feito uma implementação como teste para comparar o desempenho de uma CPU com uma GPU. Para o teste realizado, fez-se uma implementação de uma multiplicação básica de duas matrizes utilizando uma paralelização de dados e comparado o desempenho entre CPU e GPU, a descrição dessa implementação é apresentada na seção 2.7.1

2.7.1 Multiplicação de matrizes

Nesta seção será apresentado um exemplo de multiplicação de duas matrizes em OpenCL sendo A[4][4] * B[4][4] = C[4][4] e também será apresentado um exemplo de paralelização do mesmo.

Ao implementar a multiplicação de matrizes em OpenCL o código é dividido em

duas partes distintas, programa de *host* (parte 1) e *kernel* OpenCL (parte 2). O programa de host é responsável por criar um contexto (*context*) para a execução, configurar a plataforma (*platform device*), definir os objetos de memórias a serem processados pelo *kernel* (neste caso, os valores das matrizes A e B), criar a fila de comandos (*command enqueue*), enviar os objetos de memória para o *kernel* processar e, receber o retorno dos dados kernel (neste caso a matriz resultante C). Já o *kernel* é responsável por pegar os parâmetros atribuídos pelo programa de *host* (*object memories*), executar as instruções implementadas e retornar o resultado para o host.

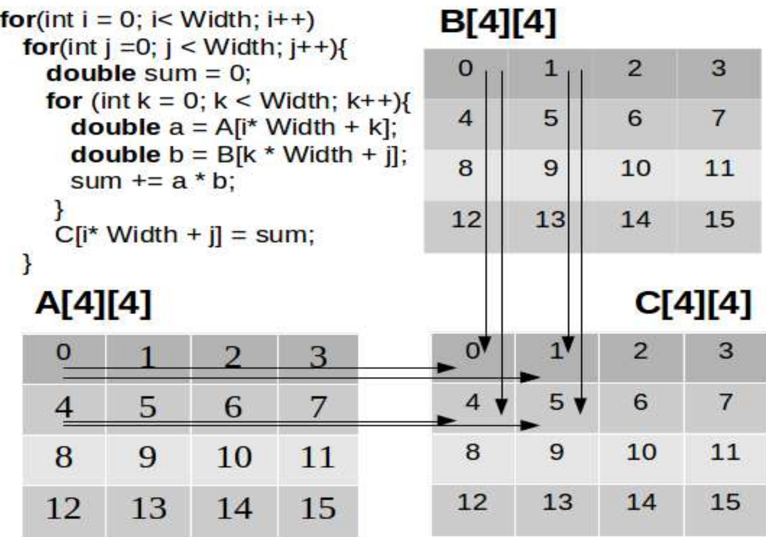

Figura 7: Exemplo multiplicação de matrizes + algoritmo

fonte: Própria

Quando o programa de *host* define os objetos de memória, ou seja, define os valores das matrizes que serão executas pelo *kernel,* no dispositivo OpenCL essas matrizes são vetorizadas, de forma que, uma matriz A[4][4] seja interpretada como um vetor A[16]. A Figura **7** mostra um exemplo de multiplicação comum (linha * coluna) de duas matrizes A[4][4] * B[4][4] = C[4][4], onde C[0] = A[0][0] * B[0][0] + A[0][1] * B[1][0] + A[0][2] * B[2][0] + A[0][3] * B[3][0], na figura também é mostrado um exemplo de algoritmo para a multiplicação de duas matrizes vetorizadas.

Para a paralelização da multiplicação de matrizes A x B = C, cada matriz é dividida em sub-matrizes, ou seja, em uma multiplicação de A[4][4] x B[4][4], gera-se: subMatrizA_0[2][2] = {{A_0, A_1},{A_4, A_5}}, subMatrizA_1[2][2] = {{A_2, A_3},{A_6, A_7}}, subMatrizA_2[2][2] = {{A_8, A_9},{A_{12}, A_{13}}}, subMatrizA_3[2][2] = {{A_{10}, A_{11}},{A_{14}, A_{15}}} e

subMatrizB_0[2][2] = {{B_0, B_1},{B_4, B_5}}, subMatrizB_1[2][2] = {{B_8, B_9},{B_{12}, B_{13}}}, subMatrizB_2[2][2] = {{B_2, B_3},{B_6, B_7}}, subMatrizB_3[2][2] = {{B_{10}, B_{11}},{B_{14}, B_{15}}}.

Desse modo, ao enviar o *kernel* OpenCL para a execução, é criado um *NDRange* de duas dimensões de tamanho 4x4, no qual também são criados *work-groups* de dimensões iguais ao *NDRange* de tamanho 2x2, onde, cada *work-item* instanciado em cada *work-group* calcula as multiplicações referentes ao seus identificadores. A figura 8 demonstra um exemplo do cálculo do primeiro elemento da matriz resultante A x B = C, calculado pelas sub-matrizes geradas, como descritas nesta seção.

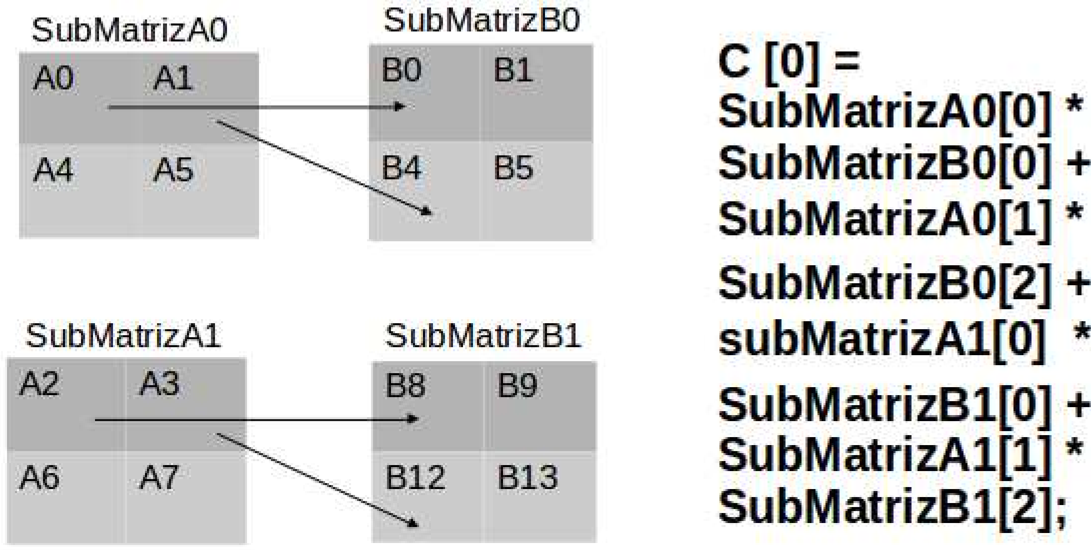

Figura 8: Exemplo de cálculo do primeiro elemento da matriz resultante com sub matrizes com *work-groups* de dimensão 2x2
fonte: Própria

Nesse escopo, foi implementado um *kernel* OpenCL de uma multiplicação de matrizes de 4800 x 4800 elementos com *work-groups* de duas dimensões de tamanho 16 x 16. O programa de *host* foi implementado, nas três linguagens suportadas pela plataforma (C/C++, Java e Python) para comparação de desempenho entre elas. A Tabela 3 apresenta os resultados obtidos dos testes de desempenhos em segundos, executados em uma CPU Intel(R) Core(TM) i5-2410M CPU @ 2.30GHz e em uma GPU AMD/ATI Radeon HD 6400M Grafics Series.

Tabela 3: Resultado multiplicação de matrizes em diferentes linguagens

Implementação	**Tempo em segundos**	
	CPU	**GPU**
Java + OpenCL	859,937	14,397
Python + OpenCL	841,851	14,257
C/C++ + OpenCL	868,585	14,139

Os resultados obtidos mostram um aumento de desempenho de aproximadamente 60 vezes entre GPU e CPU, os resultados também mostram que o tempo de execução das implementações são próximos, indicando assim, que o programa de *host* não influência no desempenho da aplicação, ou seja, o desempenho da aplicação depende do dispositivo no qual está sendo executado o *kernel* OpenCL. O código completo da implementação pode ser observado no Apêndice A.

Neste contexto, os dados de desempenho obtidos nesta aplicação mostraram o grande poder de processamento das GPUs. Sendo assim foi escolhido um algoritmo em evidência para aplicar a paralelização em plataformas heterogêneas e o algoritmo escolhido foi o novo algoritmo padrão de funções *hash* recentemente homologado (SHA-3 – Keccak), onde será aplicado diferentes tipos de paralelização, gerenciamento de memória e dimensão de work-groups, obtendo dados de desempenho e comparando os mesmos para propor uma implementação que obtenha maior desempenho neste tipo de plataforma.

CAPÍTULO 3: SHA-3

Neste capítulo é apresentado o algoritmo Keccak o novo padrão de funções hash SHA-3, onde é apresentada toda a história do algoritmo, iniciando a partir da competição aberta pelo NIST para eleição de um novo padrão de funções hash e qual o motivo de se ter um novo padrão, depois é descrita toda a competição até a escolha dos algoritmos finalistas e a eleição do vencedor, no qual foi o Keccak. Após a apresentação da competição e do algoritmo vencedor é descrito o algoritmo, onde é apresentada a estrutura geral do mesmo e permutações utilizadas pelo algoritmo. Ao final do capítulo é descrito o foco do projeto, a implementação do algoritmo para plataformas GPGPU utilizando OpenCL

3.1 Competição SHA-3

Foram reportadas falhas encontradas nas funções SHA-1 e MD5 pela comunidade científica, que geram ataques de colisão com sucesso. A função SHA-2 atualmente continua segura e inquebrável, porém como o mesmo compartilha de uma herança estrutural similar ao seu antecessor, o SHA-1, o torna suspeito e levanta dúvidas quanto a sua segurança.

Como resposta, em 2007 foi aberta uma competição com o principio de escolher um novo padrão de função de *hash*. Organizado pelo *National Institute of Standards and Technology* (NIST), atualmente a competição se encontra em sua terceira e última etapa, onde os especialistas da área escolheram 5 dos 14 algoritmos presentes na segunda etapa, dos quais haviam 64 inscritos no início da competição. O *timeline* da competição, que contêm informações mais específicas de seu processo se encontra na seção 3.1.

3.2 Timeline da competição

A competição surgiu, de acordo com o NIST (2008a) (tradução nossa),

> "em resposta a uma vulnerabilidade no SHA-1 anunciado em fevereiro de 2005, o NIST realizou um workshop de *hash* criptográfico em 31 de outubro de 2005 para avaliar o estado das suas funções *hash* aprovadas. Enquanto o NIST continuava a recomendar a transição da função SHA-1 para a família de funções aprovadas SHA-2 (SHA-224, SHA-256, SHA-384 e SHA-512), o NIST decidiu também que seria prudente desenvolver a longo prazo uma ou mais funções de *hash* através de um concurso público, semelhante ao processo de desenvolvimento para o *Advanced Encryption Standard* (AES)".

Na Tabela 4 está descrito o *timeline* contendo os períodos da competição proposta

pelo NIST para o desenvolvimento das novas funções de *hash*. Como pode ocorrer imprevistos na competição, este *timeline* é apenas uma tentativa proposta, podendo ser alterada pelo NIST quando houver necessidade (NIST, 2008a).

Tabela 4: Timeline da competição do NIST

2007	
Jan – Mar.	Publicado os requerimentos mínimos preliminares, requerimentos da submissão e critério de avaliação para os comentários públicos.
27/04/07	Período para comentários público terminado.
Abr – Jun	Análise dos comentários.
Out – Dez	Finalizado e publicado os requerimentos mínimos para aceitação, requerimentos de submissão e o critério de avaliação das funções de *hash* candidatas. Requisição para submissão das funções de *hash.*
2008	
Out – Dez	Prazo final para submissão das funções de *hash.*
2009	
Abr – Jun	Revisão das funções submetidas, e seleção dos candidatos que atende os requerimentos básicos da submissão. Primeira conferência para anunciar os candidatos da primeira etapa da competição. Chamada para comentários públicos dos candidatos da primeira etapa.
2010	
Abr – Jun	Período para comentários público terminado.
Abr – Jun	Segunda conferência das funções de *hash* candidatas. Discussão dos resultados das análises dos candidatos. Candidatos podem identificar possíveis melhorias para seus algoritmos.
Jul – Set	Análise dos comentários públicos sobre os candidatos e seleção os finalistas. Preparação do relatório para explicar a seleção. Anúncio dos finalistas e publicação do relatório de seleção.
Out – Dez	Candidatos finalistas podem anunciar qualquer ajuste de seus algoritmos. Última etapa começa.
2011	
Out – Dez	Período para comentários público da etapa final terminado.
2012	
Jan – Mar	Terceira e final conferência das funções de *hash* candidatas. Candidatos finalistas discutem sobre os comentários de suas submissões.
Abr – Jun	Análise dos comentários públicos e seleção do vencedor. Preparação do relatório para descrever a seleção final. Anúncio da nova função de *hash*.
Jul – Set	Projeto revisado do padrão de *hash*. Publicação do padrão de projeto para análise e comentários públicos.
Out – Dec	Período para comentários públicos terminado. Análise destes comentários. Envio do padrão proposto para a *Secretary of Commerce* para assinatura.

3.3 Resultado da competição

Atualmente a competição para a escolha da nova função de *hash* realizada pelo NIST já esta encerrada. A terceira etapa e ultima da competição começou no final de 2010, onde cinco algoritmos finalistas foram escolhidos para esta última etapa, dos 14 algoritmos da segunda etapa.

As funções escolhidas para esta terceira etapa são: BLAKE, Grøstl, JH, Keccak e Skein. Em outubro de 2012 o NIST anunciou o vencedor da competição SHA-3, sendo ele o algoritmo Keccak, desenvolvida por Guido Bertoni, Joan Daemen, Michaël Peeters e Gilles Van Assche que agora é oficialmente o novo padrão de funções *hash* (SHA-3).

3.4 Algoritmo Keccak

O algoritmo Keccak pertence à família de funções de esponja, que nada mais são funções de *hash* que utilizam a construção em esponja para transformação ou permutação de tamanho fixo para construção de uma função que a partir de uma entrada de qualquer tamanho gere um saída de tamanho arbitrário (DAEMEN *el al.*, 2009). Um dos grandes atrativos desta nova função de *hash* é a de que a mesma pode gerar a partir de uma entrada de tamanho variável uma saída de tamanho infinito.

Além disso, funções de esponja possuem segurança comprovada contra todos os ataques genéricos existentes (DAEMEN *et al.*, 2009).

O algoritmo Keccak consiste de duas partes: a função *Keccak-f[b](A,RC),* que realiza as permutações e operações lógicas sobre os dados e a função de esponja *Keccak[r,c](M)*, que organiza e prepara os dados de entrada para realizar a manipulação desses dados pela função de permutação e organiza os valores de saída para gerar o hash.

3.5 Função de permutação

O algoritmo Keccak utiliza técnicas de permutações para gerar o *hash*. Na função de permutação pode ser escolhida uma das sete permutações disponíveis, denotada como *Keccak-f[b]*, onde $b \in \{25, 50, 100, 200, 400, 800, 1600\}$, que representa a largura de permutações. A largura da permutação é também a largura do estado S na função de esponja.

Os valores do estado são organizados em uma matriz A de formato 5x5, que contém

25 posições de tamanho w bits, onde $w \in \{1, 2, 4, 8, 16, 32, 64\}$, tal que $w = b \div 25$, por exemplo, caso b seja igual a 1600, o tamanho de cada posição da matriz irá ter 64 bits.

Esta função realiza um número de rodadas n_r onde em cada rodada são realizadas cinco etapas que realizam operações lógicas e permutações de bits nos blocos de dados contidos na matriz A. O número de rodadas n_r depende da largura de permutação, que é dada por $n_r = 12 + 2*l$, onde $2^l = w$. Seguindo o exemplo acima, se w for igual a 64, o número de rodadas seria igual a 24, pois $2^l = 2^6$, tal que $n_r = 12 + 2 * 6$, ou seja, em cada uma das 24 rodadas seriam realizadas as 5 etapas de operações presentes na função.

As cinco etapas citadas acima que irão manipular os dados são referenciadas com letras gregas, que são θ (theta), ρ (rho), π (pi), χ (chi) e ι (iota). Cada uma delas tem objetivos diferentes e maneiras específicas para a manipulação dos dados da matriz.

A Figura 9 apresenta o pseudocódigo da função de permutação, onde a matriz *A* contendo blocos de informação e *RC* (*round constant*) são parâmetros de entradas, e são realizadas as operações lógicas *XOR*,*NOT* e *AND*, permutações e rotações de bits sobre as informações contidas nas entradas, que geram ao final da rodada uma matriz de saída *A* de tamanho 5x5.

```
Keccak-f[b](A) {
  forall i in 0…nr-1
    A = Round[b](A, RC[i])
  return A
}
Round[b](A,RC) {
  θ step
  C[x] = A[x,0] xor A[x,1] xor A[x,2] xor A[x,3] xor A[x,4],   forall x in 0…4
  D[x] = C[x-1] xor rot(C[x+1],1),                             forall x in 0…4
  A[x,y] = A[x,y] xor D[x],                              forall (x,y) in (0…4,0…4)

  ρ and π steps
  B[y,2*x+3*y] = rot(A[x,y], r[x,y]),                    forall (x,y) in (0…4,0…4)

  χ step
  A[x,y] = B[x,y] xor ((not B[x+1,y]) and B[x+2,y]),     forall (x,y) in (0…4,0…4)

  ι step
  A[0,0] = A[0,0] xor RC

  return A
}
```

Figura 9: As operações das rodadas e suas etapas de Keccak-f[b]
fonte: *DAEMEN* el al., *2009*

O parâmetro RC (definido na Tabela IV) presente no pseudocódigo da função de permutação é um vetor contendo 24 valores no formato hexadecimal que são utilizados na etapa ι.

Tabela 5: Round Constants de RC[i]

RC[0]	0x0000000000000001	**RC[12]**	0x000000008000808B
RC[1]	0x0000000000008082	**RC[13]**	0x800000000000008B
RC[2]	0x800000000000808A	**RC[14]**	0x8000000000008089
RC[3]	0x8000000080008000	**RC[15]**	0x8000000000008003
RC[4]	0x000000000000808B	**RC[16]**	0x8000000000008002
RC[5]	0x0000000080000001	**RC[17]**	0x8000000000000080
RC[6]	0x8000000080008081	**RC[18]**	0x000000000000800A
RC[7]	0x8000000000008009	**RC[19]**	0x800000008000000A
RC[8]	0x000000000000008A	**RC[20]**	0x8000000080008081
RC[9]	0x0000000000000088	**RC[21]**	0x8000000000008080
RC[10]	0x0000000080008009	**RC[22]**	0x0000000080000001
RC[11]	0x000000008000000A	**RC[23]**	0x8000000080008008

Em cada rodada n_r um desses valores é utilizado (o valor é referente ao número da rodada) para realizar a operação lógica XOR no primeiro bloco da matriz A. Esta operação tem como objetivo “quebrar” a simetria, para evitar brechas que podem ser exploradas em ataques contra o algoritmo (DAEMEN *et al,* 2011b).

3.6 Função de esponja

A função *Keccak[r,c]* utiliza a construção em esponja, que recebe um valor de entrada de tamanho variável e gera uma saída de tamanho arbitrário (DAEMEN *et al.,* 2011b). A função recebe dois parâmetros, onde: *r* é o parâmetro de *bitrate* e *c* é o parâmetro de capacidade.

O parâmetro *r* define o tamanho de que cada bloco irá ter depois da informação ser quebrada em P pedaços de tamanho *r*. É necessário quebrar a informação pois o algoritmo não pode ser aplicado (ou não desejável por questões de segurança) em uma informação inteira caso ela seja muito grande, mas sim em pequenos pedaços dela.

O parâmetro *c* afeta no desempenho do algoritmo e na segurança do *hash* gerado, onde quanto maior o valor de *c*, mais seguro o *hash,* porém exige mais desempenho da máquina.

A soma dos parâmetros *r* e *c* define o número da largura da permutação escolhida, por exemplo, na permutação 1600, um dos valores adotados adotado é r = 1024 e c = 576, onde r + c = 1600. Na competição do NIST, os desenvolvedores do Keccak submeteram quatro propostas do tamanho em bits *n* do *hash* gerado e dos parâmetros r e c (DAEMEN *et.*

al, 2011c), que são:

- n = 224: Keccak[r = 1152, c = 448]
- n = 256: Keccak[r = 1088, c = 512]
- n = 384: Keccak[r = 832, c = 768]
- n = 512: Keccak[r = 576, c = 1024]

O algoritmo possui as fases de inicialização, *padding*, absorção e compressão, que podem ser vistas no pseudocódigo representado na Figura 10.

```
Keccak[r,c](M) {
  Initialization and padding
  S[x,y] = 0,                               forall (x,y) in (0…4,0…4)
  P = M || 0x01 || 0x00 || … || 0x00
  P = P xor (0x00 || … || 0x00 || 0x80)

  Absorbing phase
  forall block Pi in P
    S[x,y] = S[x,y] xor Pi[x+5*y],          forall (x,y) such that x+5*y < r/w
    S = Keccak-f[r+c](S)

  Squeezing phase
  Z = empty string
  while output is requested
    Z = Z || S[x,y],                        forall (x,y) such that x+5*y < r/w
    S = Keccak-f[r+c](S)

  return Z
}
```

Figura 10: Função de Esponja Keccak[r,c]
fonte: *DAEMEN* el al., *2009*

Na fase de inicialização é criada uma matriz de estado S de tamanho 5x5. Na fase de *padding* ocorre o preenchimento da mensagem *M* com um padrão de 0x01 (valor hexadecimal), seguido de 0x00 necessários seguido de 0x80 final, para tornar a mensagem *M* múltiplo do parâmetro *r*.

O preenchimento da mensagem é necessário pois o algoritmo disponibiliza a opção da mensagem de entrada ter um tamanho variável que nem sempre dispõe de um tamanho aceitável para realizar as operações de permutação.

Na fase de absorção (*absorbing*), a mensagem preenchida que foi armazenada numa variável *P* é quebrada em pedaços *Pi* de tamanho *r* e depois inseridos na matriz de estado *S*. Após a inserção dos valores na matriz são realizados as permutações, ou seja, a função de

permutação *Keccak-f[b](S)* é aplicada dos valores de *S*. Esta fase é realizada enquanto existir blocos *Pi* que ainda não foram aplicados pela função de permutação.

Por fim, na fase de compressão (*squeezing*) é definido *hLength,* tamanho do hash que será gerado, e então é realizada a concatenação dos blocos de dados já permutados da matriz *S*, que gera uma "string" *Z* que é o valor *hash* (com formato hexadecimal) da informação de entrada que tem tamanho em bits determinado por *n*. A fase de compressão é realizada enquanto $hLength - r > 0$, que executa novamente a função de permutação sobre a matriz *S* e concatena os valores em *Z* novamente caso a condição seja verdadeira. Isto faz com que a geração do *hash* seja ainda mais aleatório, prevenindo ataques que geram colisões (DAEMEN *et al,* 2011b).

3.7 Keccak em OpenCL

Neste projeto foi escolhido o algoritmo Keccak vendedor da competição SHA-3, ele apresenta uma estrutura relativamente simples e versátil, o que facilita não só na implementação do mesmo, mas também na compatibilidade de arquiteturas heterogêneas e massivamente paralelas no qual é utilizado no projeto. Este trabalho tem como objetivo estudar o algoritmo vencedor da competição SHA-3 o Keccak e, em seguida, propor e implementar diferentes técnicas de paralelização para sistemas heterogêneos utilizando OpenCL para obter dados de desempenho em GPUs

A função principal (*core*) executada pelo algoritmo é a keccak-f[b] composta por quatro etapas (Θ, $\rho\pi$, χ, ι), porem cada etapa possui dependência de dados de primeiro nível, isto é, o passo de corrente depende apenas do resultado do passo anterior. Este recurso permite explorar técnicas de paralelismo em sistemas heterogêneos

CAPÍTULO 4: PESQUISA E DESENVOLVIMENTO

Neste capítulo são apresentados cinco trabalhos correlatos encontrados na literatura, sendo que dois deles foram implementados em GPUS utilizando placas gráficas Nvidia (HOFFMANN, 2011) e (SEVESTRE, 2011), dois deles implementados em FPGAs (GUO *et al,* 2010) e (PEREIRA, *et* al, 2011), e um implementado em ASIC (GUO *et al,* 2010).

4.1 Trabalhos Correlatos

Gerhard Hoffmann em seu trabalho apresenta uma implementação da família função *hash* Keccak em placas gráficas, usando o framework CUDA da NVIDIA. Essa implementação permite escolher uma função da família *hash* e processar documentos arbitrários. Além disso, ele apresenta a primeira aplicação para utilização de modo árvore do Keccak que é ainda mais adequado para paralelização (HOFFMANN, 2011).

Guillaume Sevestre apresenta uma implementação do Keccak em *Graphics Processing Unit*, em um modo de árvore paralelo explorando a capacidade de computação paralela das placas gráficas usando CUDA (SEVESTRE, 2011).

Em seu trabalho de Xu Guo descreveu uma abordagem consistente e sistemática para mover um processo de referência SHA-3 a partir de hardware FPGA prototipagem para a implementação ASIC (GUO *et al,* 2010).

Fábio Pereira apresenta uma implementação do Keccak em FPGA usando arquitetura de pipeline com intuito de obter dados de desempenho (PEREIRA, *et* al, 2011).

Tabela 6: Trabalhos correlatos

Autores	Título	Plataforma
HOFFMANN	GPU Implementation of the Keccak Hash Function Family	NVIDIA GTX 295 GPU
SEVESTRE	Implementation of Keccak hash function in Tree mode on Nvidia GPU	Core i5-750 2.6 Ghz Nvidia GTS 250
PEREIRA	Pipeline architecture	Virtex 5
XU GUO	Fair and Comprehensive Performance Evaluation of 14 Second Round SHA-3 ASIC implementation	FPGA implementation
		ASIC implementation

A Tabela 6 mostra os trabalhos correlatos encontrados na literatura organizados por

autor, título do trabalho e plataforma de implementação. Entretanto não foi encontrado nenhum trabalho publicado na literatura sobre a implementação do algoritmo Keccak implementado em OpenCL.

Com a apresentação dos trabalhos correlatos encontrados na literatura, é importante salientar que não é possível uma comparação direta entre eles, pois são implementados em arquiteturas diferentes com tecnologias e ambientes diferentes, porém, é possível classifica-los com relação ao seu desempenho.

4.2 Implementações do Keccak em OpenCL

Nesta seção são apresentadas as implementações realizadas neste projeto e também serão descritas as técnicas de paralelização utilizadas para a implementação das mesmas.

Foram realizadas três implementações, sendo elas, a primeira implementação adaptada para OpenCL da referência oficial disponível do Keccak, a segunda implementação otimizada da implementação anterior e a terceira implementação paralelizada, utilizando uma paralelização proposta por Gerhard Hoffmann (HOFFMANN, 2011) implementada em CUDA, que para a implementação deste projeto foi adaptada para OpenCL. Todos os *kernels* das três implementações foram adaptados para suportar o paralelismo de dados (SIMD – *Single Instruction Multiple Data*).

Para as implementações o programa de *host* foi desenvolvido em C/C++, onde foi configurada toda a plataforma, objetos de memória e parâmetros de execução para o *kernel*. Já o *kernel* foi implementado na linguagem da API OpenCL (C99), onde foi implementado a principal função executada pelo algoritmo, ou o core do mesmo.

A principal função executada pelo algoritmo é a keccak-f [b], onde b {25, 50, 100, 200, 400, 800, 1600} é a largura de a permutação. E recebe como entrada uma matriz 5x5 (*state*), com comprimento w {1, 2, 4, 8, 16, 32, 64} (b = 25 W), onde se aplicam os quatro passos do algoritmo (Θ, $\rho\pi$, χ, ι) e retorna uma matriz de 5x5 com comprimento w como saída.

4.2.1 Implementação I

Nesta primeira implementação foi utilizado como base a implementação de referência disponível no site oficial do Keccak (http://keccak.noekeon.org/), porém com algumas adaptações para ser implementada em OpenCL (segmentar o código em duas partes: programa de *host* e *kernel*) e otimizar o seu desempenho em GPUs. O programa de *host* foi implementado em C/C++ como descrito na seção anterior onde foi configurada a plataforma e definido um vetor múltiplo de 25 como parâmetro do *kernel* para aplicar o *core* do algoritmo.

O *kernel* foi implementado de forma que, para cada 25 elementos do vetor recebido ele instancie um *work-item* para processar o algoritmo Keccak, ou seja, se for definido um vetor de 1000 elementos, será instanciado 40 (1000/25) *work-items* para processar 40 blocos diferentes do vetor de entrada, de forma que, cada *work-item* processe um bloco diferente do vetor de entrada paralelamente e gere uma saída após aplicar o *core* do algoritmo (keccak-f).

No código 1 é descrito um pseudocódigo do *core* do algoritmo Keccak que mostra um exemplo do *kernel* descrito anteriormente.

Código 1

```
Entrada: data, Saída: out
  0: #define ROUNDS 24
  1: #define rot(x,y) ((x << y) ^ (x >> (64-y))
  2: #define getIndice(i,j) (((i)%5)+5*((j)%5))
  3: int id = get_global_id(0);
  4: __local unsigned long A[25], B[25], C[5], D[5];
  5:  for i←0,  j← id*25 to b < (id*25)+25 and i < 25
  6:        A[i]←data[j];
7: end for
  8:  for x←0 to ROUNDS
  9:       for i ← 0 to 5
10:           C[i]←A[i][0]^A[i][1]^A[i][2]^A[i][3]^A[i][4];
11:       end for
12:       for i ← 0 to 5
13:           D[i]←C[(i-1)%5] ^ rot(C[(i+1)%5],1);
14:       end for
15:       for i ← 0 to 5
16:            for j ← 0 to 5
17:               A[getIndice(i,j)]←A[getIndice(i,j)] ^ D[i];
18:            end for
19:       end for
20:       for i ← 0 to 5
21:           for j ← 0 to 5
22:             B[getIncide(i, (2*i+3*j)%5)] ←rot(A[getIndice(i,j)], r[getIndice(i,j)]);
23:           end for
24:       end for
25:       for i ← 0 to 5
26:           for j ← 0 to 5
```

```
27:             A[getIndice(i,j)]←B[getIndice(i,j)] ^ ((~B[getIndice((i+1)%5], j) & B[getIndice((i+2)%5], j);
28:           end for
29:       end for
30:       A[0]←A[0] ^ RC[x];
31: end for
32: for i←0,  j← id*25 to j < (id*25)+25 and i < 25
33:      out[j]←A[i];
34: end for
```

Na linha 0 é definido uma constante que define o número de iterações que será das pelo algoritmo, logo em seguida na linha 1 é definida a permutação de rotação utilizada pelo algoritmo e na linha 2 é definida uma função que recebe como parâmetro dois números (i,j) retorna o um índice que é utilizado para acessar uma posição do vetor (A[getIncide(i,j)]).

Na linha 3 é utilizada uma função nativa do OpenCL que retorna o id do *work-item* em execução na dimensão x (get_global_id(0)). Linhas 5 – 7 é descrito como cada *work-item* copia um bloco de 25 elementos para sua memória local para aplicar as permutações, ou seja, o *work-item* de id = 0 ira copiar um bloco de 0 – 24, o *work-item* de id = 1 ira copiar um bloco de 25 – 49 e assim sucessivamente.

Nas linhas 9 – 19 é aplicada a permutação Θ (Theta), nas linhas 20 – 24 é aplicada as permutações ρπ (Rho and Pi), nas linhas 25 – 29 é aplicado a permutação χ (Chi), na linha 30 é aplicada a permutação ι (Iota) e nas linhas 32 – 34 o *state* resultante após a aplicação da função keccak-f, é copiada de volta para o buffer de saída de modo que cada *work-item* copie seu bloco processado.

O código completo descrito anteriormente implementado em OpenCL pode ser observado no APENDICE B.

4.2.2 Implementação II

Nesta segunda implementação foi utilizado como base a implementação anterior, que utiliza como base a implementação de referência disponível no site oficial do Keccak (http://keccak.noekeon.org/) Entretanto nesta implementação, foram feitas algumas alterações nas funções do *core* do algoritmo para otimizar o desempenho.

As alterações realizadas foram feitas nas funções: θ (theta), ρ (rho), π (pi), χ (chi) e ι (iota), de modo que, todas as permutações realizadas nestas funções foram implementadas de maneira estática (*hardcoded*), ou seja, todas as permutações que utilizavam a instrução **for,** foram adaptadas para executar linhas de código (instruções) pré-definidas, pois as instruções

for, geram *jumps* e acesso excessivo na memória, atuando assim de maneira inversamente proporcional no desempenho da aplicação.

Foram feitos alguns testes e comparações de desempenho entre as duas implementações (I e II) para comparar se houve otimização na mesma e aumento de desempenho entre as duas implementações. Os resultados dos testes e das comparações podem ser observados no Capítulo 5.

No código 2 é descrito um pseudocódigo da primeira permutação realizada pelo algoritmo mostrando as alterações realizadas podem ser comparadas com o código1.

Código 2: Exemplo da primeira permutação após a otimização

```
Entrada: data
 0: #define rot(x,y) ((x << y) ^ (x >> (64-y))
 1: int id = get_global_id(0);
 2:    local unsigned long A[25], B[25], C[5], D[5];
 3: A[0] = data[id * 25 + 0];
 4: A[1] = data[id * 25 + 1];
 5:  . . .
 6: A[24] = data[id * 25 + 24];
 7: C[0] = 0;
 8: C[0] = C[0] ^ A[0];
 9: C[0] = C[0] ^ A[5];
10: C[0] = C[0] ^ A[10];
11: C[0] = C[0] ^ A[15];
12: C[0] = C[0] ^ A[20];
13: D[0] = rot(C[0], 1);
14: C[1] = 0;
15: C[1] = C[1] ^ A[1];
16: C[1] = C[1] ^ A[6];
17: C[1] = C[1] ^ A[11];
18: C[1] = C[1] ^ A[16];
19: C[1] = C[1] ^ A[21];
20: D[1] = rot(C[1], 1);
21: C[2] = 0;
22: C[2] = C[2] ^ A[2];
23: C[2] = C[2] ^ A[7];
24: C[2] = C[2] ^ A[12];
25: C[2] = C[2] ^ A[17];
26: C[2] = C[2] ^ A[22];
27: D[2] = rot(C[2], 1);
28: C[3] = 0;
29: C[3] = C[3] ^ A[3];
30: C[3] = C[3] ^ A[8];
31: C[3] = C[3] ^ A[13];
32: C[3] = C[3] ^ A[18];
33: C[3] = C[3] ^ A[23];
34: D[3] = rot(C[3], 1);
35: C[4] = 0;
36: C[4] = C[4] ^ A[4];
37: C[4] = C[4] ^ A[9];
38: C[4] = C[4] ^ A[14];
39: C[4] = C[4] ^ A[19];
40: C[4] = C[4] ^ A[24];
41: D[4] = rot(C[4], 1);
42:  . . .
```

No código 2, na linha 0 é definida a permutação de rotação utilizada no algoritmo,

igual no código 1, Na linha 2 é utilizada uma função nativa do OpenCL que retorna o id do *work-item* em execução na dimensão x (get_global_id(0)). Nas linhas 3 – 6 é descrito como cada *work-item* copia um bloco de 25 elementos para sua memória local para aplicar as permutações, essas linhas são equivalentes as linhas 3 – 7 do código 1.

Nas linhas 7 – 41 é descrita as alterações feitas na primeira e segunda parte da permutação Θ (Theta), no qual todas as iterações realizadas na função **for** foram implementadas de maneira estática, essas mesmas permutações são equivalente as linhas 9 – 14 do código 1. O código completo do kernl implementado em OpenCL pode ser observado no APENDICE C.

4.2.3 Implementação III

Nesta terceira implementação foi utilizado como referência, uma paralelização proposta por Gerhard Hoffmann. O programa de *host* foi implementado em C/C++ como descrito na seção onde foi configurada a plataforma e definido uma matriz com altura e largura múltiplas de 5 (largura%5 == 0 e altura%5 == 0) como parâmetro do *kernel* para aplicar o *core* do algoritmo.

Para a implementação do *kernel*, foi definido um espaço de índice de duas dimensões (2D) de tamanho equivalente ao parâmetro de entrada do *kernel*, onde este espaço de índice foi subdividido em *work-groups* de 5x5, de modo que, cada *work-group* aplique o core do algoritmo Keccak e cada *work-item* instanciado neste *work-group* processe um elemento da matriz resultante equivalente ao seu id local.

No Código 3 é descrito um pseudocódigo da terceira implementação realizada, onde a mesma pode ser comparada com as duas implementações anteriores.

Código 3

```
Entrada: data
Saída: out
  0: #define ROUNDS 24
  1: #define rot(a,b,c) (((a) << b) ^ ((a) >> c))
  2:     local unsigned long A[5*5], C[5*5], D[5*5];
  3: int t = get_local_id(0), s = get_local_id(0)%5;
  4: if t < 25 then
  5:        A[t] ← data[t];
  6:        for i ←0 to ROUNDS
  7:              C[t] ← A[s] ^ A[s+5] ^ A[s+10] ^ A[s+15] ^ A[s+20];
  8:              D[t] ← C[b[20+s]] ^ rot(C[b[5+s]],1,63);
  9:           if t == 0
 10:               C[0] ←  rot(A[0] ^ D[0], 0,64);
 11:           end if
 12:           else
 13:               C[t] ←  rot(A[a[t]] ^ D[b[t]], ro[t][0], ro[t][1]);
```

```
14:        end else
15:           A[d[t]] ← C[c[t][0]] ^ ((~C[c[t][1]]) & C[c[t][2]]);
16:           A[t] ← A[t] ^ rc[(t==0)? 0: 1][i];
17:        end for
18: end if
```

No código 3, na linha 0 é definido uma constante com o numero de iterações que será executada pelo algoritmo, logo em seguida na linha 1 é definida a permutação de rotação utilizada no mesmo. Na linha 5 pode ser observado a cópia de um elemento do *state* de entrada para o *state* no qual será aplicado a permutação keccak-f, equivalente as linhas 3 – 7 do código 1 e linhas 3 – 6 do código 2.

As linhas 7 – 8 do Código 3 descrevem a permutação Θ (Theta) (equivalente linhas 9 – 19 do Código 1 e linhas 7 – 41 do Código 2). Nas linhas 9 -14 do Código 3 são aplicadas a permutações ρπ (Rho and Pi) referentes as linhas 20 – 24 do Código 1, na linha 15 do Código 3 é aplicada a permutação χ (chi), referente as linhas 25 – 29 do Código 1 e por fim a linha 16 do Código 3 aplica a permutação ι (Iota) equivalente a linha 30 do Código 1. O código completo implementado em OpenCL pode ser observado no APENDICE D.

4.3 Modelo de execução Keccak em OpenCL

Nesta seção são apresentados os modelos de execução das implementações realizadas, também são apresentadas as distribuições de dados entre os *work-items* juntamente com as técnicas de paralelizações utilizadas para implementar os *kernels*.

4.3.1 Modelo de execução das implementações I e II

Para a primeira e segunda implementação a estrutura original do Keccak foi quase totalmente mantida, mesmo que tenha sido feito alguns ajustes para a implementação e OpenCL e maximizar o desempenho em GPU. Sendo assim para implementar a paralelização de dados, na função OpenCL, é recebido como entrada um vetor de X elementos múltiplo de 25 (X%25 = 0), onde será instanciada em um espaço de índice (*NDRange)* de tamanho X/25 *work-items*, cada *work-item* receberá como entrada espaços diferentes do vetor de entrada de modo que cada entrada recebida forme blocos de 25 elementos, formando assim diferentes *states* para aplicar a função principal do algoritmo.

A Figura 11 mostra o modelo de distribuição dados e execução do *kernel* OpenCL conforme foi descrito acima. De modo que recebe como entrada um vetor de tamanho x onde x é múltiplo de 25 e é instanciada um espaço de índice (*NDRange)* com x/25 *work-item.* Cada *work-item* instância um *kernel* OpenCL, copia um espaço do vetor de entrada para sua memória local formando um *state* e, aplica a permutação keccak-f nesse *state,* onde depois é copiado para o vetor de saída.

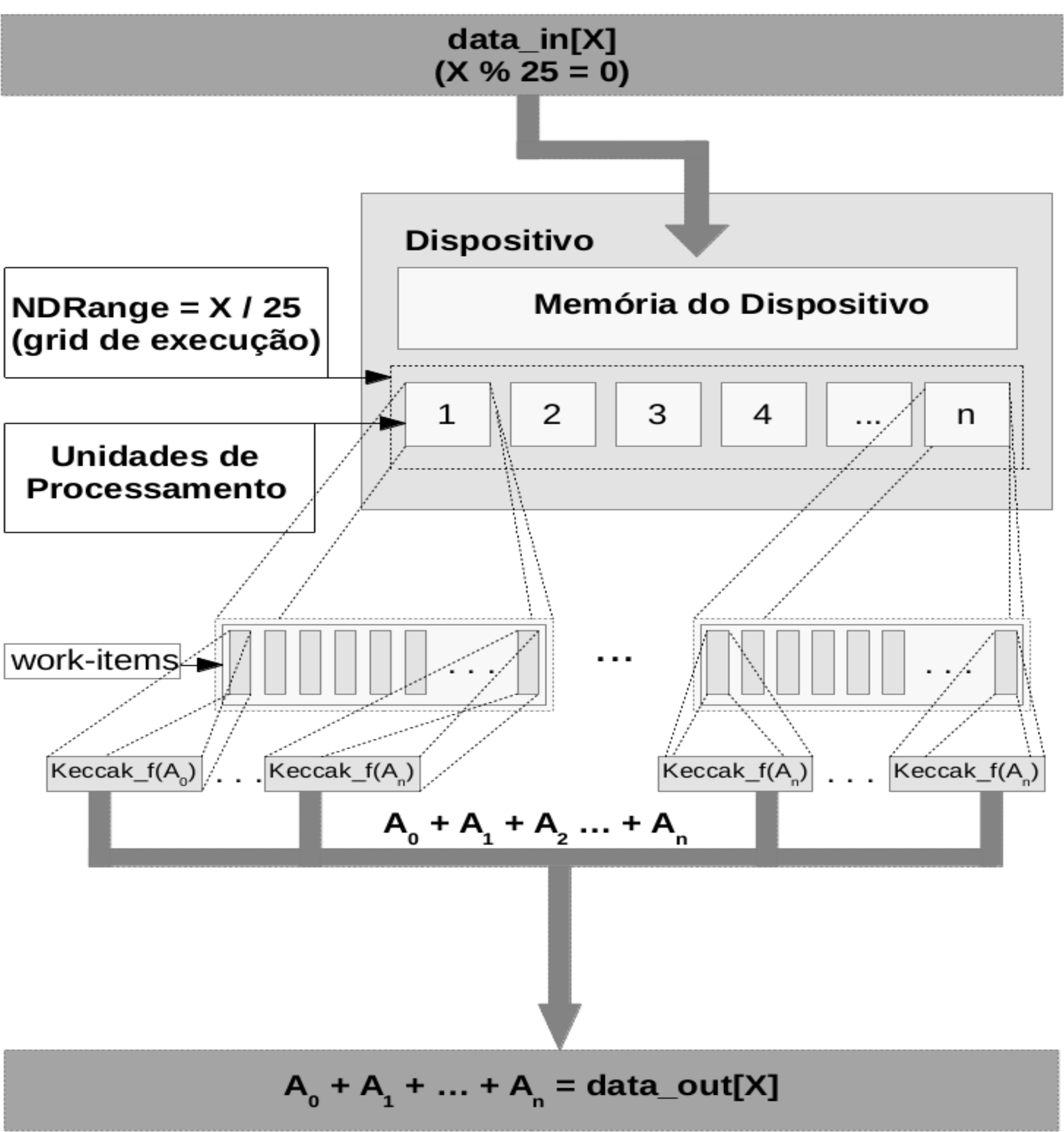

Figura 11: Modelo de execução Keccak em OpenCL das implementações I e II
Fonte: Própria

4.3.2 Modelo de execução da implementação III

Para a terceira implementação a estrutura do Keccak foi alterada para realizar a paralelização do algoritmo segundo a paralelização proposta por HOFFMANN. Sendo assim para implementar a paralelização de dados é recebido como entrada uma matriz com largura e altura múltiplas de 5 (altura%5 == 0 e largura%5 ==0), onde é instanciado um espaço índice (*NDRange*) de duas dimensões (2D) de tamanho equivalente, no qual é subdividido em *work-groups* de dimensões 5 por 5 (5x5), de modo que cada *work-group* calcule um bloco (*state*) diferente da entrada e cada *work-item* instanciado em cada *work-group* aplique a permutação keccak-f paralelamente, de modo que, cada *work-item* calcule um elemento da matriz resultante referente ao seu id local.

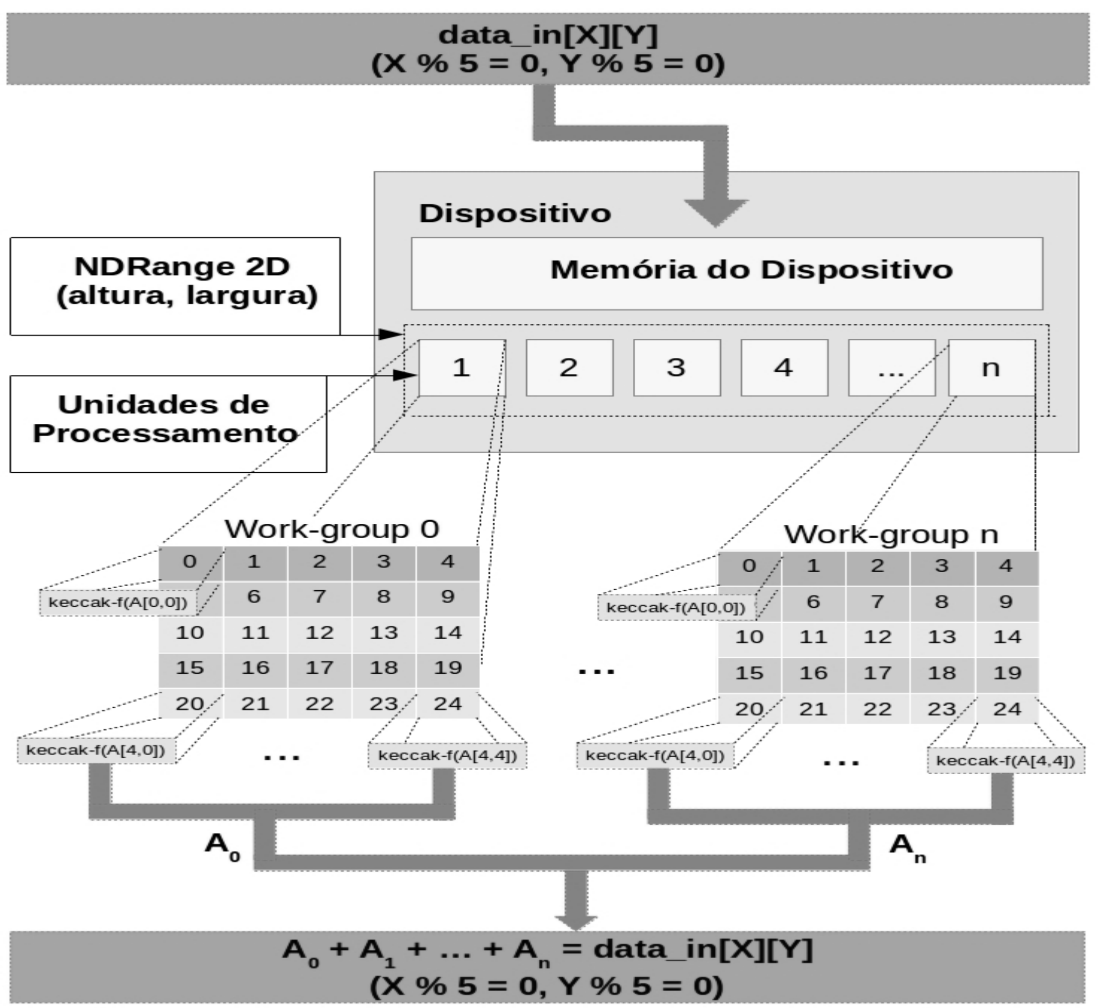

Figura 12: Modelo de execução do Keccak em OpenCL da implementação III
fonte: Própria

A Figura 12 mostra o modelo de distribuição dos dados e de execução do *kernel* OpenCL conforme descrito nesta seção. Na Figura pode ser observado como os *work-groups* são organizados e também como cada *work-item* em cada *work-group* aplica a permutação keccak-f(A) em cada posição do *state* que o *work-group* esta processando.

CAPÍTULO 5: RESULTADOS

Neste capítulo é apresentado o ambiente onde foram realizados os testes, descrevendo arquiteturas dos dispositivos utilizados, sistema operacional e versões de SDK, compiladores e bibliotecas utilizadas no projeto. Após a apresentação do ambiente é descrito a metodologia de testes utilizada no projeto e consequentemente os resultados obtidos com as implementações e comparação dos resultados. E por fim é descrito as conclusões dos resultados e do trabalho.

5.1 Ambiente de teste

Os testes foram realizados no sistema operacional Linux Mint 14 com versão do *kernel 3.2.0-23-generic* com CPU Intel core i5 e uma GPU Nvidia GeForce GTX 650 ti, compilados com o compilador nativo do sistema operacional *g++ versão 4.6* e utilizando as bibliotecas da versão da API OpenCL suportada pela placa de gráfica utilizada neste projeto, que no caso é a versão 1.2.

A GPU utilizada para a execução dos testes (Nvidia GeForce GTX 650 ti) possui uma memória global de 2GB (GDDR5) e 768 *stream processors* com clock de 1032 MHz.

A descrição de como foram executados os testes e resultados das comparações entre as três implementações podem ser vista nas próximas seções (5.1 e 5.2)

5.1 Metodologia de testes

Nesta seção será descrito como os testes foram realizados, juntamente com a execução e coleta dos dados.

Após a compilação das implementações com o compilador g++ (descrito na seção anterior) é gerado um executável, onde o mesmo recebe dois parâmetros para a execução, onde o primeiro parâmetro é um arquivo (.cl) que contem o *kernel* implementado e o segundo parâmetro é a quantidade de blocos que serão passados para o *kernel* para aplicar as instruções implementadas (keccak_f). Dessa forma para a execução é utilizado a seguinte linha de comando.

./nome-do-executavel arquivo-com-o-código-do-kernel.cl quantidade-de-blocos
./Implementacao1 kernelImplementacao1.cl 100000

Os testes realizados foram para calcular o desempenho entre as três implementações, de modo que, ao executar o teste foi passado diferentes quantidade de bloco à serem processados pelos *kernels,* onde foi calculado o tempo que cada *kernel* levou para executar aquela quantidade de blocos.

Para medir o tempo de execução de cada *kernel*, foi utilizado primitivas da API OpenCL que permite coletar esse tipo de dado. A execução do *kernel* OpenCL é baseado em eventos alocados em uma fila de comandos (*command enqueue*), dessa forma, ao submeter uma fila de comandos para a execução é coletado um tempo T1 e ao término da execução da fila de comandos é coletado outro tempo T2, o tempo de execução de cada *kernel* é dados pela diferença desses dois tempos coletados (T2 – T1).

Para a execução dos testes foi desenvolvido um *shell* script para automatizar a execução e captura os dados das implementações. O Código 4 descreve o código de implementação do *shell scrtipt* utilizado.

Código 4

```
0: #!/bin/bash
1: if [ -f $1.csv ]; then
2:     rm -fr $1.csv
3: fi
4: for ((i=1; i<100; i=$((i+10))));
5:   do
6:      out=$(./$1 $1'.cl' $i | head -n1 | awk '{print $5}')
7:       echo $i';'$out >> $1.csv
8:   done
```

A linha 0 indica o caminho de onde se encontra o *bash* utilizado para execução do script. Nas linhas 1 – 3 é realizada a verificação de existência do arquivo de saída que será gerado ($1 – significa o primeiro parâmetro passado para o script, nesse caso é passado o nome do executável), no caso se ele existir o mesmo é excluído.

Na linha 4 é definido o laço para a repetição, onde é informada a quantidade de blocos a serem processadas inicialmente (variável $i), a condição de parada do mesmo (i<100) e, de quanto em quanto será incrementado o tamanho desses blocos ($((i+10))).

Na linha 6 é executado o teste com a quantidade de blocos do valor da variável $i, e é

coletado o ultimo valor da saída do teste, que nesse caso é o tempo que o teste levou para ser executado, no qual esse valor é copiado para a variável $out. Já na linha 7, é criado um arquivo de saída com node do primeiro parâmetro passado para o script e extensão .csv (para ser aberto em um editor de planilhas), nesse arquivo é gravado todos os testes executados pelo script, onde é gravado o numero de blocos processados e o tempo com que o teste levou para ser executado.

Foram executados diversos testes com diferentes quantidades de blocos a serem processados. Os resultados dos testes e comparação entre as implementações pode ser visto na seção seguinte (seção 5.2).

5.2 Resultados

Com a apresentação do ambiente de testes e a descrição da metodologia utilizada os resultados dos testes e a comparação de desempenho entre as três implementações são apresentados.

Para a realização dos testes foi iniciado com a execução de 1 bloco até 4000000 (quarto milhões) de blocos, de modo que, foi tirado o tempo de com as mesmas quantidades de blocos entre as três implementações.

Para melhor visualização dos resultados os testes foram divididos em três partes, sendo eles, parte I: comparação de desempenho de 0 a 435600 blocos, parte II: Comparação de desempenho de 435600 a 1742400 blocos e parte III: Comparação de desempenho de 1742400 a 4000000 blocos, que é a quantidade máxima de blocos suportada pela memória da placa de vídeo utilizada no projeto.

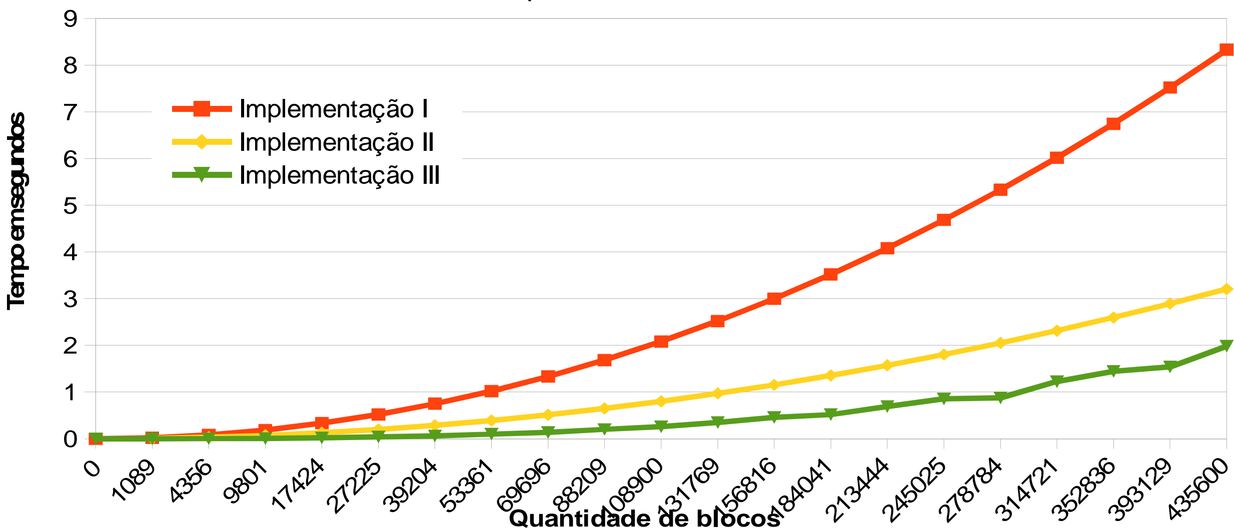

Figura 13: Gráfico I, comparação de desempenho parte I.
Fonte: Própria

O Gráfico I, mostra o resultado da comparação dos testes do primeiro *range* (de 0 a 435600), onde pode ser observado que a implementação I tem pior desempenho em relação as outras duas implementações e a Implementação III tem melhor desempenho em relação a implementação I e II.

Como a Implementação I é a que teve menor desempenho as medidas de aumento de desempenho foram tiradas baseadas nela. Sendo assim, o aumento de desempenho da Implementação II para a implementação I é de aproximadamente 61,5%, já o aumento de desempenho da Implementação III para a Implementação I é de aproximadamente 76% e o aumento de desempenho em relação à Implementação II é de aproximadamente 38%.

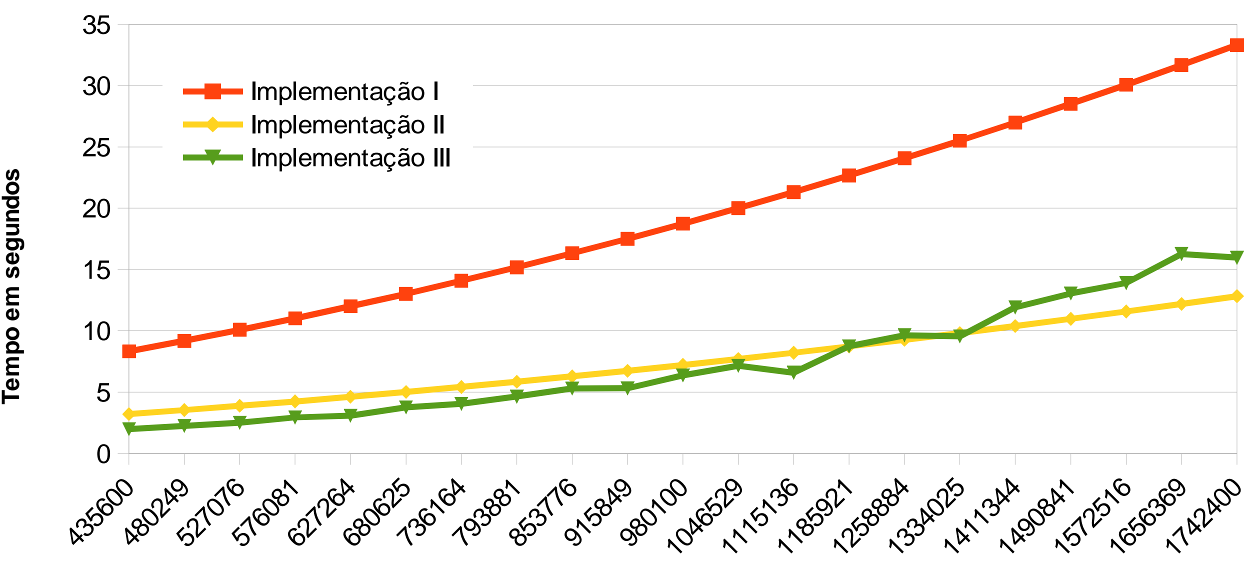

Figura 14: Gráfico II, comparação de desempenho parte II
fonte: Própria

O Gráfico II, mostra os resultados da comparação dos testes do segundo *range* (de 435600 a 1742400), onde pode ser observado que até aproximadamente 1350000 blocos, a Implementação III tem melhor desempenho sobre as outras duas implementações. Entretanto a Implementação III atinge seu ponto de saturação, onde não é mais viável a paralelização do algoritmo, ou seja, o desempenho da Implementação III será inversamente proporcional a quantidade de blocos instanciados a partir desse ponto.

Então a partir deste ponto que pode ser observado no Gráfico, a Implementação III começa a ter uma queda no seu desempenho. Sendo assim, a Implementação II começa a ter um melhor desempenho sobre a Implementação I e III. Ainda pode ser observado que a Implementação I ainda possui um pior desempenho em relação as outras implementações.

Ao final desse segundo teste, pode-se notar que, a implementação II tem um ganho de desempenho de aproximadamente 25% em relação a implementação III e aproximadamente 60% em relação a implementação I. Já a implementação III ainda tem um ganho de desempenho de aproximadamente 49% em relação à implementação I.

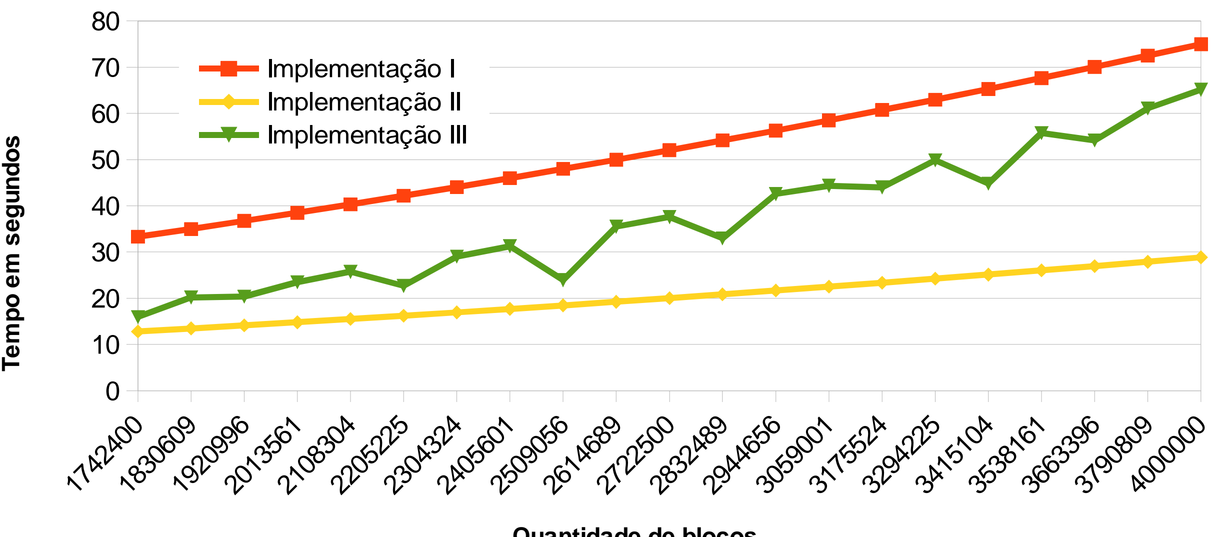

Figura 15: Gráfico III, comparação de desempenho parte III
fonte: Própria

O Gráfico III, mostra os resultados da comparação dos testes do terceiro *range* (1742400 a 4000000), onde pode-se notar que a o desempenho da Implementação III continua a cair após a saturação e a Implementação II continua tendo um melhor desempenho sobre as outras implementações, já a implementação I continua tendo um desempenho inferior as outras implementações.

No final deste terceiro teste, a diferença do aumento de desempenho da Implementação II para Implementação III que no teste anterior era de aproximadamente 25% agora atinge aproximadamente 56%, já a diferença de desempenho da Implementação II para a Implementação I continua com aproximadamente 60%. Entretanto a diferença de desempenho em relação as implementações I e III que no teste anterior era de aproximadamente 49% agora neste ultimo teste chega a aproximadamente apenas 13%.

CAPÍTULO 6: CONCLUSÕES

Este trabalho de conclusão de curso apresenta uma visão geral sobre o uso da GPU para acelerar o processamento do algoritmo Keccak.

Com os resultados obtidos é possível afirmar que os objetivos foram alcançados, com o estudo da arquitetura de uma GPU, foi possível entender como é a execução e a distribuição dos núcleos de uma arquitetura *multithreading* e como esses *threads* executam paralelamente para aumentar o desempenho utilizando a plataforma OpenCL, uma plataforma GPGPU utilizada neste trabalho.

Foram apresentados quatro trabalhos correlatos encontrados na literatura, com o estudo desses trabalhos foi possível analisar diferentes formas de paralelização, sendo elas: uma paralelização de dados e uma paralelização do algoritmo.

Para a definição da linguagem de programação que foi utilizada neste projeto, no caso a linguagem C/C++, foi feito primeiramente um teste básico de uma aplicação de multiplicação de matrizes, onde após a execução do teste, foi possível concluir que as linguagens suportadas pela API OpenCL, não têm influencia sobre o desempenho da aplicação, sendo que o desempenho da mesma depende apenas do hardware utilizado para executar a aplicação.

Com o estudo das possíveis técnicas de paralelização do algoritmo ou da execução do mesmo e, a definição da linguagem de programação, foram realizadas três implementações para a comparação de desempenho entre elas e apontar qual possui um melhor desempenho na plataforma utilizada no projeto. Para cada uma das implementações, foi aplicado uma paralelização de dados, com o intuito de criar uma arquitetura escalável, a fim de aumentar o desempenho das implementações.

Sendo assim, com as implementações realizadas, foram feitos testes de desempenho para validar as implementações e compará-las para saber qual implementação tem melhor desempenho na arquitetura testada.

Com os resultados obtidos é possível concluir que a Implementação III tem maior desempenho sobre as outras duas implementações chegando a até 78% de ganho de desempenho comparando com a Implementação I e 38% de ganho de desempenho comparando com a Implementação II, porém esse aumento de desempenho só é obtido até sua saturação, onde a paralelização do algoritmo não se torna mais viável, que no caso da GPU (Nvidia GeForce GTX 650 ti) utilizada neste projeto é de aproximadamente 1350000 (um

milhão e trezentos e cinquenta mil) blocos.

Após sua saturação, ao final dos testes a Implementação II tem maior desempenho sobre as outras duas implementações, chegando a aproximadamente um aumento de desempenho de 56% em relação à Implementação III e aproximadamente 60% de desempenho em relação a Implementação I. Entretanto diferença de desempenho em relação às implementações III e I que antes da saturação era de 78%, ao final dos testes é de apenas 13%, onde se pode concluir que, com o aumento da quantidade de blocos a Implementação III terá desempenho inferior às outras duas implementações.

No entanto os resultados também mostram que as três implementações obtiveram um excelente desempenho em GPUs. Sendo assim, como o objetivo do projeto era propor uma melhor implementação para esse tipo de plataforma, todas as três implementações tem um alto desempenho, porém, dependendo da quantidade de blocos a serem processados, ou seja, se a quantidade de blocos for inferior a 1350000 blocos a implementação III tem melhor desempenho sobre as outras duas implementações. Mas se a quantidade de blocos for maior, a Implementação II tem melhor desempenho nesta plataforma. É importante salientar que para outras plataformas e ambientes, novos testes deverão ser realizados.

6.1 Limitações

Para o desenvolvimento do projeto foram encontradas algumas limitações, uma delas foi à infraestrutura, pois para o desenvolvimento do projeto e realização dos testes só havia um único ambiente. Sendo assim, os resultados obtidos no projeto são específicos para o ambiente testado e, não foi possível comparar o desempenho das implementações com diferentes GPUs (AMD/ATI e Nvidia) ou outros ambientes.

6.2 Trabalhos futuros

Os trabalhos futuros desse trabalho de conclusão de curso são: (i) melhorar o desempenho das implementações gerenciando memórias locais e privadas dos *work-items* ou gerenciando a distribuição dos *work-items* e *work-groups*, (ii) comparar diferentes versões do keccak, para obter dados de qual versão tem melhor desempenho em GPUs, (iii) comparar o desempenho das implementações em diferentes versões da plataforma OpenCL, (iv) realizar os testes em outra GPU “compatível” (AMD) para comparação, (v) implementar e comparar testes na plataforma CUDA, (vi) escrita de artigos com os resultados obtidos.

REFERÊNCIAS

GARLAND M. , *et al*, **Parallel Computing Experiences with CUDA**, IEEE Micro, vol. 28, pp. 13–27, July 2008.

NICKOLLS J. ; DALLY W. J., **The GPU Computing Era**, IEEE Micro, vol. 30, pp. 56–69, March 2010. 2. Oxford: Clarendon, 1892, pp.68-73.

FANG J. *et al*, **A Comprehensive Performance Comparison of CUDA and OpenCL**, International Conference on Parallel Processing , 2011

HWU W. ; KIRK B. **Programming massively parallel processors: A hands-on approach**, 2010.

Khronos OpenCL Working Group, **The OpenCL Specification**, 2011.

MUNSHI A. *et al.* **OpenCL Programming Guide**. Pearson Education Inc., 2011

SCARPINO M. **OpenCL in Action**. Manning Publications Co., 2012

NVIDIA Corporation. **NVIDIA CUDA Developer Guide for NVIDIA Optimus Systems**, 2010.

HENSLEY J. *et al.* **Close to Metal**. Special Interest Group on Computer Graphics and Interactive Techniques, 2007.

HOFFMANN G.; CAYREL P. L., **GPU Implementation of the Keccak Hash Function Family**, SERSC International Journal of Security and Its Applications Vol. 5 No 4, October, 2011.

SEVESTRE G., **Implementation of Keccak hash function in Tree mode on Nvidia GPU**, 2011.

Pereira, F. D. ; Ordonez, E. D. M.; Sakai, I. D. **Hash function keccak: exploring parallelism with pipeline**. In: PDCS- Parallel and Distributed Computing and Systems, 2011.

GUO X. ; HUANG S., NAZHANDALI L. and SXHAUMONT P., **Fair and Comprehensive Performance Evaluation of 14 Second Round SHA-3 ASIC Implementations**, Second SHA-3 Candidate Conference, 2010.

DAEMEN, Joan *et al.* **Cryptographic Sponges.** 2009. Disponível em <http://sponge.noekeon.org/>. Acesso em 20 abril. 2013.

DAEMEN, Joan *et al.* **Keccak Implementation Overview.** Version 3.1, 2011a. Disponível em < http://keccak.noekeon.org/Keccak-implementation-3.1.pdf>. Acesso em 20 ago. 2013.

DAEMEN, Joan *et al.* **The Keccak Reference.** Version 3, 2011b. Disponível em <http://keccak.noekeon.org/Keccak-reference-3.0.pdf>. Acesso em 10 maio. 2013.

DAEMEN, Joan *et al.* **The Keccak SHA-3 Submission.** Version 3, 2011c. Disponível em <http://keccak.noekeon.org/Keccak-submission-3.pdf>. Acesso em 20 maio. 2013.

G. Bertoni, J. Daemen, M. Peeters and G. Van Assche, **The Keccak reference**, 2011.

G. Bertoni, J. Daemen, M. Peeters and G. Van Assche, **The Keccak SHA-3 submission**, 2011.

NATIONAL INSTITUTE OF STANDARDS AND TECHNOLOGY (NIST). Cryptographic Hash Algorithm Competition. 2008b Disponível em: <http://csrc.nist.gov/groups/ST/hash/sha-3/index.html>. Acesso em: 11 jun. 2013.

PEREIRA, Fábio Dacêncio ; ORDONEZ, Edward David Moreno; SOUZA, A. M. **Exploiting Heterogeneous Systems: Keccak on OpenCL**. PDPTA'13 - Int'l Conf on Parallel & Distributed Processing Techniques & Applications, July 22-25, 2013, Las Vegas, USA.

PEREIRA, Fábio Dacêncio ; SOUZA, A. M. ; ORDONEZ, Edward David Moreno . **OpenCL: uma Alternativa para o Ensino de Arquitetura de Computadores**. International Journal of Computer Architecture Education, v. 2, p. 1-4, 2013.

PEREIRA, Fábio Dacêncio ; ORDONEZ, Edward David Moreno ; SOUZA, A. M. . **Exploiting parallelism on Keccak: FPGA and GPU Comparison**. Parallel & Cloud Computing, v. 2, p. 1-6, 2013.

APÊNDICE A: Kernel multiplicação de matrizes

```
#define HB WA  // Matrix B height
#define WC WB  // Matrix C width
#define HC HA  // Matrix C height

#define AS(j, i) As[i + j * BLOCK_SIZE]
#define BS(j, i) Bs[i + j * BLOCK_SIZE]

__kernel __attribute__((reqd_work_group_size(BLOCK_SIZE,BLOCK_SIZE,1)))
Void MultMatrix( __global float* A, __global float* B, __global float* C){

    __local float As[BLOCK_SIZE*BLOCK_SIZE];
    __local float Bs[BLOCK_SIZE*BLOCK_SIZE];

    // Block index
    int bx = get_group_id(0);
    int by = get_group_id(1);

    // Thread index
    int tx = get_local_id(0);
    int ty = get_local_id(1);

    // index do primeiro sub-Bloco A
    int aBegin = WA * BLOCK_SIZE * by;

    // Index do ultimo sub-Bloco A
    int aEnd   = aBegin + WA - 1;

    // incremento para as iteracoes dos sub-Blocos A
    int aStep  = BLOCK_SIZE;

    // index do primeiro sub-Bloco A
    int bBegin = BLOCK_SIZE * bx;

    // incremento para as iteracoes dos sub-Blocos B
    int bStep  = BLOCK_SIZE * WB;

    //guarda o resultado da multiplicacao
    float sum = 0.0f;

    //Carrega os Valores para os sub-Blocos
    for (int a = aBegin, b = bBegin; a <= aEnd; a += aStep, b += bStep) {
        // Carregas as matrizes da memoria do dispositivo
        // para compartilhar memoria; cada thread carrega um elemento
        //printf("AS[%d] = A[%d] e BS[%d] = B [%d]", (tx + ty * BLOCK_SIZE), (a + WA * ty + tx), (tx + ty *
        BLOCK_SIZE), (b + WB * ty + tx)  );

        AS(ty, tx) = A[a + WA * ty + tx];
        BS(ty, tx) = B[b + WB * ty + tx];

        // Sincroniza as threads para ter certeza que os dados foram carregados
        barrier(CLK_LOCAL_MEM_FENCE);
```

```
        // Multiplica as matrizes juntas
        // cada thread computa um elemento do sub-Bloco
        for (int k = 0; k < BLOCK_SIZE; ++k){
            sum += AS(ty, k) * BS(k, tx);
            //printf("AS[%d] * BS[%d]", (k + ty * BLOCK_SIZE), (tx + k * BLOCK_SIZE));
            }

        // Sincroniza as threads para ter certerza
        // que os calculos foram feitos antes de carregas dois novos sub-Blocos
        barrier(CLK_LOCAL_MEM_FENCE);
    }
    C[get_global_id(1) * get_global_size(0) + get_global_id(0)] = sum;
}
```

APÊNDICE B: Kernel implementação I

```
typedef unsigned char  uint8_t;
typedef unsigned long uint64_t;

#define index(x, y) (((x)%5)+5*((y)%5))
uint64_t ROL64(uint64_t data, unsigned int offset)
{
    const int _offset = offset;
    return ((offset != 0) ? ((data << _offset) ^ (data >> (64-offset))) : data);
}

__constant static const uint64_t RC[24]=
                    {0x0000000000000001,0x0000000000008082, 0x800000000000808A,
                    0x8000000080008000, 0x000000000000808B, 0x0000000080000001,
                    0x8000000080008081, 0x8000000000008009, 0x000000000000008A,
                    0x0000000000000088, 0x0000000080008009, 0x000000008000000A,
                    0x000000008000808B, 0x800000000000008B, 0x8000000000008089,
                    0x8000000000008003, 0x8000000000008002, 0x8000000000000080,
                    0x000000000000800A, 0x800000008000000A, 0x8000000080008081,
                    0x8000000000008080, 0x0000000080000001, 0x8000000080008008};

__constant static const int r[25]=
                   {0, 1, 62, 28, 27,
                   36, 44, 6, 55, 20,
                   3, 10, 43, 25, 39,
                   41, 45, 15, 21, 8,
                   18, 2, 61, 56, 14};


__kernel void implementacao_I(__global uint64_t *data),  __global uint64_t *out){

        int t = get_global_id(0);
        __local uint64_t A[25], B[25], C[5], D[5];
        unsigned int x, y;

        // montagem dos blocos
         for(int a = 0, b = t*25; b < (t*25)+25 , a<25;b++, a++){
                        A[a] =  data[b];
            }

        for(int i = 0; i < 24; i++) {
            // Theta
            for(x=0; x<5; x++) {
                        C[x] = 0;
                        for(y=0; y<5; y++)
                                C[x] ^= A[index(x, y)];
                        D[x] = ROL64(C[x], 1);
            }
            for(x=0; x<5; x++)
                        for(y=0; y<5; y++)
                                A[index(x, y)] ^= D[(x+1)%5] ^ C[(x+4)%5];
            // Rho
            for(x=0; x<5; x++)
                        for(y=0; y<5; y++)
                                A[index(x, y)] = ROL64(A[index(x, y)], r[index(x, y)]);
```

```
        // Pi
        for(x=0; x<5; x++) for(y=0; y<5; y++)
                B[index(x, y)] = A[index(x, y)];

        for(x=0; x<5; x++) for(y=0; y<5; y++)
                A[index(0*x+1*y, 2*x+3*y)] = B[index(x, y)];



        // Chi
        for(y=0; y<5; y++) {
                for(x=0; x<5; x++)
                        C[x] = A[index(x, y)] ^ ((~A[index(x+1, y)]) & A[index(x+2, y)]);
                for(x=0; x<5; x++)
                        A[index(x, y)] = C[x];
        }

        // Iota
        A[0] ^= RC[i];
    }

      for(int i=0, b=t*25; i < 25, b<t*25+25; i++,b++)
           data[b] = A[i];
}
```

APÊNDICE C: Kernel implementação II

```
typedef unsigned char  uint8_t;
typedef unsigned long uint64_t;

uint64_t ROL64(uint64_t data, unsigned int offset)
{
   const int _offset = offset;
   return ((offset != 0) ? ((data << _offset) ^ (data >> (64-offset))) : data);
}

__constant static const uint64_t RC[24]=
                        {0x0000000000000001,0x0000000000008082, 0x800000000000808A,
                        0x8000000080008000, 0x000000000000808B, 0x0000000080000001,
                        0x8000000080008081, 0x8000000000008009, 0x000000000000008A,
                        0x0000000000000088, 0x0000000080008009, 0x000000008000000A,
                        0x000000008000808B, 0x800000000000008B, 0x8000000000008089,
                        0x8000000000008003, 0x8000000000008002, 0x8000000000000080,
                        0x000000000000800A, 0x800000008000000A, 0x8000000080008081,
                        0x8000000000008080, 0x0000000080000001, 0x8000000080008008};

__constant static const int r[25]=
                      {0, 1, 62, 28, 27,
                      36, 44, 6, 55, 20,
                      3, 10, 43, 25, 39,
                      41, 45, 15, 21, 8,
                      18, 2, 61, 56, 14};

__kernel
void implementacao_II(__global uint64_t *data), __global uint64_t *out){

        int x = get_global_id(0);
        __local uint64_t A[25], B[25], C[5], D[5];
        int offset = x * 25;

        A[0] = data[offset+0];
        A[1] = data[offset+1];
        A[2] = data[offset+2];
        A[3] = data[offset+3];
        A[4] = data[offset+4];

        A[5] = data[offset+5];
        A[6] = data[offset+6];
        A[7] = data[offset+7];
        A[8] = data[offset+8];
        A[9] = data[offset+9];

        A[10] = data[offset+10];
        A[11] = data[offset+11];
        A[12] = data[offset+12];
        A[13] = data[offset+13];
        A[14] = data[offset+14];

        A[15] = data[offset+15];
        A[16] = data[offset+16];
        A[17] = data[offset+17];
```

```
A[18] = data[offset+18];
A[19] = data[offset+19];

A[20] = data[offset+20];
A[21] = data[offset+21];
A[22] = data[offset+22];
A[23] = data[offset+23];
A[24] = data[offset+24];


for(int i =0; i < 24; i++){
C[0] = 0;
C[0] = C[0] ^ A[0];
C[0] = C[0] ^ A[5];
C[0] = C[0] ^ A[10];
C[0] = C[0] ^ A[15];
C[0] = C[0] ^ A[20];
D[0] = ROL64(C[0], 1);
//x = 1
C[1] = 0;
C[1] = C[1] ^ A[1];
C[1] = C[1] ^ A[6];
C[1] = C[1] ^ A[11];
C[1] = C[1] ^ A[16];
C[1] = C[1] ^ A[21];
D[1] = ROL64(C[1], 1);
// x = 2
C[2] = 0;
C[2] = C[2] ^ A[2];
C[2] = C[2] ^ A[7];
C[2] = C[2] ^ A[12];
C[2] = C[2] ^ A[17];
C[2] = C[2] ^ A[22];
D[2] = ROL64(C[2], 1);
// x = 3
C[3] = 0;
C[3] = C[3] ^ A[3];
C[3] = C[3] ^ A[8];
C[3] = C[3] ^ A[13];
C[3] = C[3] ^ A[18];
C[3] = C[3] ^ A[23];
D[3] = ROL64(C[3], 1);
// x = 4
C[4] = 0;
C[4] = C[4] ^ A[4];
C[4] = C[4] ^ A[9];
C[4] = C[4] ^ A[14];
C[4] = C[4] ^ A[19];
C[4] = C[4] ^ A[24];
D[4] = ROL64(C[4], 1);

A[0] = A[0] ^ D[1] ^ C[4];
A[5] = A[5] ^ D[1] ^ C[4];
A[10] = A[10] ^ D[1] ^ C[4];
A[15] = A[15] ^ D[1] ^ C[4];
A[20] = A[20] ^ D[1] ^ C[4];
```

```
A[1] = A[1] ^ D[2] ^ C[0];
A[6] = A[6] ^ D[2] ^ C[0];
A[11] = A[11] ^ D[2] ^ C[0];
A[16] = A[16] ^ D[2] ^ C[0];
A[21] = A[21] ^ D[2] ^ C[0];

A[2] = A[2] ^ D[3] ^ C[1];
A[7] = A[7] ^ D[3] ^ C[1];
A[12] = A[12] ^ D[3] ^ C[1];
A[17] = A[17] ^ D[3] ^ C[1];
A[22] = A[22] ^ D[3] ^ C[1];

A[3] = A[3] ^ D[4] ^ C[2];
A[8] = A[8] ^ D[4] ^ C[2];
A[13] = A[13] ^ D[4] ^ C[2];
A[18] = A[18] ^ D[4] ^ C[2];
A[23] = A[23] ^ D[4] ^ C[2];

A[4] = A[4] ^ D[0] ^ C[3];
A[9] = A[9] ^ D[0] ^ C[3];
A[14] = A[14] ^ D[0] ^ C[3];
A[19] = A[19] ^ D[0] ^ C[3];
A[24] = A[24] ^ D[0] ^ C[3];

 A[0] = ROL64(A[0], r[0]);
A[5] = ROL64(A[5], r[5]);
A[10] = ROL64(A[10], r[10]);
A[15] = ROL64(A[15], r[15]);
A[20] = ROL64(A[20], r[20]);

A[1] = ROL64(A[1], r[1]);
A[6] = ROL64(A[6], r[6]);
A[11] = ROL64(A[11], r[11]);
A[16] = ROL64(A[16], r[16]);
A[21] = ROL64(A[21], r[21]);

A[2] = ROL64(A[2], r[2]);
A[7] = ROL64(A[7], r[7]);
A[12] = ROL64(A[12], r[12]);
A[17] = ROL64(A[17], r[17]);
A[22] = ROL64(A[22], r[22]);

A[3] = ROL64(A[3], r[3]);
A[8] = ROL64(A[8], r[8]);
A[13] = ROL64(A[13], r[13]);
A[18] = ROL64(A[18], r[18]);
A[23] = ROL64(A[23], r[23]);

A[4] = ROL64(A[4], r[4]);
A[9] = ROL64(A[9], r[9]);
A[14] = ROL64(A[14], r[14]);
A[19] = ROL64(A[19], r[19]);
A[24] = ROL64(A[24], r[24]);

B[0] = A[0];
B[5] = A[5];
B[10] = A[10];
B[15] = A[15];
B[20] = A[20];
```

```
B[1] = A[1];
B[6] = A[6];
B[11] = A[11];
B[16] = A[16];
B[21] = A[21];

B[2] = A[2];
B[7] = A[7];
B[12] = A[12];
B[17] = A[17];
B[22] = A[22];

B[3] = A[3];
B[8] = A[8];
B[13] = A[13];
B[18] = A[18];
B[23] = A[23];

B[4] = A[4];
B[9] = A[9];
B[14] = A[14];
B[19] = A[19];
B[24] = A[24];

A[0] = B[0];
A[16] = B[5];
A[7] = B[10];
A[23] = B[15];
A[14] = B[20];

A[10] = B[1];
A[1] = B[6];
A[17] = B[11];
A[8] = B[16];
A[24] = B[21];

A[20] = B[2];
A[11] = B[7];
A[2] = B[12];
A[18] = B[17];
A[9] = B[22];

A[5] = B[3];
A[21] = B[8];
A[12] = B[13];
A[3] = B[18];
A[19] = B[23];

A[15] = B[4];
A[6] = B[9];
A[22] = B[14];
A[13] = B[19];
A[4] = B[24];

C[0] = A[0] ^ ((~A[1]) & A[2]);
C[1] = A[1] ^ ((~A[2]) & A[3]);
C[2] = A[2] ^ ((~A[3]) & A[4]);
C[3] = A[3] ^ ((~A[4]) & A[0]);
C[4] = A[4] ^ ((~A[0]) & A[1]);
```

```
A[0] = C[0];
A[1] = C[1];
A[2] = C[2];
A[3] = C[3];
A[4] = C[4];

C[0] = A[5] ^ ((~A[6]) & A[7]);
C[1] = A[6] ^ ((~A[7]) & A[8]);
C[2] = A[7] ^ ((~A[8]) & A[9]);
C[3] = A[8] ^ ((~A[9]) & A[5]);
C[4] = A[9] ^ ((~A[5]) & A[6]);

A[5] = C[0];
A[6] = C[1];
A[7] = C[2];
A[8] = C[3];
A[9] = C[4];

C[0] = A[10] ^ ((~A[11]) & A[12]);
C[1] = A[11] ^ ((~A[12]) & A[13]);
C[2] = A[12] ^ ((~A[13]) & A[14]);
C[3] = A[13] ^ ((~A[14]) & A[10]);
C[4] = A[14] ^ ((~A[10]) & A[11]);

A[10] = C[0];
A[11] = C[1];
A[12] = C[2];
A[13] = C[3];
A[14] = C[4];

C[0] = A[15] ^ ((~A[16]) & A[17]);
C[1] = A[16] ^ ((~A[17]) & A[18]);
C[2] = A[17] ^ ((~A[18]) & A[19]);
C[3] = A[18] ^ ((~A[19]) & A[15]);
C[4] = A[19] ^ ((~A[15]) & A[16]);

A[15] = C[0];
A[16] = C[1];
A[17] = C[2];
A[18] = C[3];
A[19] = C[4];

C[0] = A[20] ^ ((~A[21]) & A[22]);
C[1] = A[21] ^ ((~A[22]) & A[23]);
C[2] = A[22] ^ ((~A[23]) & A[24]);
C[3] = A[23] ^ ((~A[24]) & A[20]);
C[4] = A[24] ^ ((~A[20]) & A[21]);

A[20] = C[0];
A[21] = C[1];
A[22] = C[2];
A[23] = C[3];
A[24] = C[4];

A[0] = A[0] ^ (RC[i] << 64);

}
```

```
        out[offset+0] = A[0];
        out[offset+1] = A[1];
        out[offset+2] = A[2];
        out[offset+3] = A[3];
        out[offset+4] = A[4];

        out[offset+5] = A[5];
        out[offset+6] = A[6];
        out[offset+7] = A[7];
        out[offset+8] = A[8];
        out[offset+9] = A[9];

        out[offset+10] = A[10];
        out[offset+11] = A[11];
        out[offset+12] = A[12];
        out[offset+13] = A[13];
        out[offset+14] = A[14];

        out[offset+15] = A[15];
        out[offset+16] = A[16];
        out[offset+17] = A[17];
        out[offset+18] = A[18];
        out[offset+19] = A[19];

        out[offset+20] = A[20];
        out[offset+21] = A[21];
        out[offset+22] = A[22];
        out[offset+23] = A[23];
        out[offset+24] = A[24];

    }
```

APÊNDICE D: Kernel implementação III

```
#define ROUNDS 24
#define R64(a,b,c) (((a) << b) ^ ((a) >> c))
#define AS(j, i) A[i + j * BLOCK_SIZE]

typedef unsigned int uint32_t;
typedef unsigned long uint64_t;

__constant unsigned long rc[5][ROUNDS] = {
    {0x0000000000000001, 0x0000000000008082, 0x800000000000808A,
     0x8000000080008000, 0x000000000000808B, 0x0000000080000001,
     0x8000000080008081, 0x8000000000008009, 0x000000000000008A,
     0x0000000000000088, 0x0000000080008009, 0x000000008000000A,
     0x000000008000808B, 0x800000000000008B, 0x8000000000008089,
     0x8000000000008003, 0x8000000000008002, 0x8000000000000080,
     0x000000000000800A, 0x800000008000000A, 0x8000000080008081,
     0x8000000000008080, 0x0000000080000001, 0x8000000080008008},
    {0, 0 , 0 , 0 , 0 , 0 , 0 , 0 ,
     0, 0 , 0 , 0 , 0 , 0 , 0 , 0 ,
     0, 0 , 0 , 0 , 0 , 0 , 0 , 0 },
    {0, 0 , 0 , 0 , 0 , 0 , 0 , 0 ,
     0, 0 , 0 , 0 , 0 , 0 , 0 , 0 ,
     0, 0 , 0 , 0 , 0 , 0 , 0 , 0 },
    {0, 0 , 0 , 0 , 0 , 0 , 0 , 0 ,
     0, 0 , 0 , 0 , 0 , 0 , 0 , 0 ,
     0, 0 , 0 , 0 , 0 , 0 , 0 , 0 },
    {0, 0 , 0 , 0 , 0 , 0 , 0 , 0 ,
     0, 0 , 0 , 0 , 0 , 0 , 0 , 0 ,
     0, 0 , 0 , 0 , 0 , 0 , 0 , 0 }};

__constant unsigned int ro[25][2] = {
         /*y=0*/          /*y=1*/          /*y=2*/          /*y=3*/          /*y=4*/
/*x=0*/{ 0,64}, /*x=1*/{44,20}, /*x=2*/{43,21}, /*x=3*/{21,43}, /*x=4*/{14,50},
/*x=1*/{ 1,63}, /*x=2*/{ 6,58}, /*x=3*/{25,39}, /*x=4*/{ 8,56}, /*x=0*/{18,46},
/*x=2*/{62, 2}, /*x=3*/{55, 9}, /*x=4*/{39,25}, /*x=0*/{41,23}, /*x=1*/{ 2,62},
/*x=3*/{28,36}, /*x=4*/{20,44}, /*x=0*/{ 3,61}, /*x=1*/{45,19}, /*x=2*/{61, 3},
/*x=4*/{27,37}, /*x=0*/{36,28}, /*x=1*/{10,54}, /*x=2*/{15,49}, /*x=3*/{56, 8}};

__constant unsigned int a[25] = {
    0,  6, 12, 18, 24,
    1,  7, 13, 19, 20,
    2,  8, 14, 15, 21,
    3,  9, 10, 16, 22,
    4,  5, 11, 17, 23};

__constant unsigned int b[25] = {
    0,  1,  2,  3, 4,
    1,  2,  3,  4, 0,
    2,  3,  4,  0, 1,
    3,  4,  0,  1, 2,
    4,  0,  1,  2, 3};
```

```
__constant unsigned int c[25][3] = {
   { 0, 1, 2}, { 1, 2, 3}, { 2, 3, 4}, { 3, 4, 0}, { 4, 0, 1},
   { 5, 6, 7}, { 6, 7, 8}, { 7, 8, 9}, { 8, 9, 5}, { 9, 5, 6},
   {10,11,12}, {11,12,13}, {12,13,14}, {13,14,10}, {14,10,11},
   {15,16,17}, {16,17,18}, {17,18,19}, {18,19,15}, {19,15,16},
   {20,21,22}, {21,22,23}, {22,23,24}, {23,24,20}, {24,20,21}};

__constant unsigned int d[25] = {
         0,  1,  2,  3,  4,
        10, 11, 12, 13, 14,
        20, 21, 22, 23, 24,
         5,  6,  7,  8,  9,
        15, 16, 17, 18, 19};

unsigned long ROL64(unsigned long a, unsigned int offset)
{
   const int _offset = offset;
   return ((offset != 0) ? ((a << _offset) ^ (a >> (64-offset))) : a);
}

__kernel __attribute__ ((reqd_work_group_size(BLOCK_SIZE,BLOCK_SIZE,1)))
void implementacao_III(__global const unsigned long *d_data, __global unsigned long *d_out) {

   __local unsigned long A[BLOCK_SIZE*BLOCK_SIZE];
   __local unsigned long C[BLOCK_SIZE*BLOCK_SIZE];
   __local unsigned long D[BLOCK_SIZE*BLOCK_SIZE];

   // Block index
   int bx = get_group_id(0);
   int by = get_group_id(1);

   // Thread index
   int tx = get_local_id(0);
   int ty = get_local_id(1);

   // index do primeiro sub-Bloco A
   int aBegin = WIDTH * BLOCK_SIZE * by;

   //Index do ultimo sub-Bloco A
   int aEnd   = aBegin + WIDTH - 1;

   // incremento para as iteracoes dos sub-Blocos A
   int aStep  = BLOCK_SIZE;

   int t = tx + ty * BLOCK_SIZE;
   int s = t%5;

   for (int a = aBegin; a <= aEnd; a += aStep) {
      AS(ty, tx) = d_data[a + WIDTH * ty + tx];
  }
      barrier(CLK_LOCAL_MEM_FENCE);
```

```
 for(int i=0;i<ROUNDS;++i) {
        C[t] = A[s]^A[s+5]^A[s+10]^A[s+15]^A[s+20];
        D[t] = C[b[20+s]] ^ R64(C[b[5+s]],1,63);
        if(t==0)
            C[0] = ROL64(A[0] ^ D[0], 0);
        else
            C[t] = R64(A[a[t]]^D[b[t]], ro[t][0], ro[t][1]);

        A[d[t]] = C[c[t][0]] ^ ((~C[c[t][1]]) & C[c[t][2]]);
        A[t] = A[t]^rc[(t==0)?0:1][i];
    }

  barrier(CLK_LOCAL_MEM_FENCE);

  d_out[get_global_id(1) * get_global_size(0) + get_global_id(0)] = A[t];
}
```