Paralelização do Algoritmo AES em CPU e GPU Utilizando HIP sobre ROCm

Fernando Torres Moreira, Ana Paula Canal
Curso de Ciência da Computação
UFN - Universidade Franciscana
Santa Maria - RS
fernando.moreira@ufn.edu.br, apc@ufn.edu.br

***Resumo*—A crescente demanda por segurança no processamento de grandes volumes de dados torna a criptografia *Advanced Encryption Standard* (AES) um possível gargalo de desempenho em sistemas sequenciais baseados em CPU. Este trabalho apresenta a paralelização do algoritmo AES em Unidades de Processamento Gráfico (GPUs) utilizando a plataforma aberta *Radeon Open Compute* (ROCm) e a interface de portabilidade HIP. O estudo comparou o desempenho de uma versão serial em C++ executada na CPU com a implementação paralela na GPU, utilizando cargas de trabalho variando de 1 KB a 1 GB. Os resultados comprovaram que a paralelização resolve o gargalo do processador em arquivos grandes, alcançando acelerações de até 52 vezes na descriptografia e 36 vezes na criptografia. Para arquivos menores que 1 MB, o custo de transferência de dados pelo barramento PCIe torna a execução sequencial mais vantajosa. Conclui-se que o uso do paradigma GPGPU é uma solução eficiente para acelerar a proteção de grandes conjuntos de dados.**

***Palavras-chave*-Criptografia; Computação de Alto Desempenho; GPGPU; Linguagem de Programação C++; Aceleração de Hardware:**

I. Introdução

O avanço contínuo da Computação de Alto Desempenho (*High Performance Computing* — HPC) tem impulsionado o desenvolvimento de novas estratégias de processamento capazes de lidar com volumes crescentes de dados e demandas computacionais cada vez maiores [1]. Diante desse cenário, a paralelização surge como abordagem para explorar de forma eficiente os recursos disponíveis em arquiteturas modernas.

Entre as diversas soluções existentes, o uso de unidades de processamento gráfico (GPUs) para fins gerais — paradigma conhecido como GPGPU (*General-Purpose computing on Graphics Processing Units*) — consolidou-se como uma alternativa para acelerar aplicações que exigem alta capacidade de processamento. Esse modelo vem sendo adotado em áreas como aprendizado de máquina, simulações científicas e criptografia.

Plataformas abertas como o ROCm (*Radeon Open Compute*) e sua interface de portabilidade HIP (*Heterogeneous-compute Interface for Portability*), possibilitam o desenvolvimento de aplicações de alto desempenho em um ecossistema livre e multiplataforma.

Nesse cenário, algoritmos criptográficos como o *Advanced Encryption Standard* (AES) apresentam potencial de otimização quando executados de forma paralela em GPUs. A exploração dessa abordagem pode contribuir para reduzir gargalos de desempenho sem comprometer a segurança, o que a torna relevante para sistemas que processam grandes volumes de dados.

A. Justificativa

A criptografia de grandes volumes de dados é uma necessidade fundamental em sistemas de *Big Data* e HPC. No entanto, a elevada demanda computacional de algoritmos como o AES pode criar gargalos de desempenho em CPUs. Embora trabalhos anteriores, como o de Tybusch [2], já tenham demonstrado ganhos de desempenho significativos ao paralelizar o AES em GPUs com a plataforma proprietária CUDA, ainda existe uma lacuna na literatura referente a plataforma ROCm e HIP [3].

Com a crescente adoção de plataformas de computação de alto desempenho, como o ROCm da AMD, que integra sistemas como os supercomputadores Frontier e El Capitan, líderes do ranking mundial na lista dos maiores supercomputadores do mundo [4], e interfaces de portabilidade como o HIP, surge a necessidade de investigar a viabilidade e a eficiência dessas novas ferramentas para tarefas de criptografia. Este trabalho se justifica por buscar preencher essa lacuna, aplicando os conceitos de paralelização do AES em um ecossistema de *hardware* e *software* aberto, e comparando seus resultados com a abordagem sequencial.

B. Objetivos

O objetivo geral deste trabalho é desenvolver e avaliar a paralelização do algoritmo AES em uma arquitetura de GPGPU, utilizando a plataforma ROCm e a interface HIP, visando uma análise de desempenho comparativa com a execução serial em CPU.

Para alcançar esse objetivo, este trabalho define os seguintes objetivos específicos:

- Realizar uma revisão bibliográfica sobre os conceitos de HPC, GPGPU, o algoritmo AES, as tecnologias ROCm e HIP, e a metodologia Scrum.

- Levantar e examinar estudos recentes que explorem a aceleração do AES em GPU, destacando abordagens, resultados e limitações.
- Implementar uma versão serial *single-threaded* do AES em C++
- Implementar uma versão paralela do AES para GPU utilizando a linguagem C++ e a interface HIP.
- Avaliar e comparar o desempenho (Tempo de Execução, *Throughput* e *Speedup*) de ambas as versões em diferentes cargas de trabalho, quantidade de *threads* e execução em CPU/GPU.

II. Referencial Teórico

Esta seção apresenta a fundamentação teórica e tecnológica necessária para a compreensão e o desenvolvimento deste trabalho. Inicialmente, são explorados os conceitos de Computação de Alto Desempenho (HPC) e o paradigma GPGPU, essenciais para entender a utilização de placas de vídeo em processamentos de propósito geral. Na sequência, é apresentado o ecossistema de *software* selecionado, englobando a plataforma aberta ROCm, a interface de portabilidade HIP e a linguagem de programação C++. São descritos também a estrutura matemática e o fluxo de execução do algoritmo criptográfico AES, o objeto central de paralelização deste estudo. Por fim, são apresentados os princípios do *framework* ágil Scrum, adotado para gerenciar e estruturar o ciclo de desenvolvimento de *software*.

A. Computação de Alto Desempenho (HPC) e o Paradigma GPGPU

A Computação de Alto Desempenho, ou HPC (*High Performance Computing*), refere-se ao uso de sistemas computacionais avançados para solucionar problemas complexos que seriam inviáveis, em tempo ou escala, para computadores convencionais. Caracterizados pelo elevado poder de processamento, grande capacidade de memória e redes de interconexão de alta velocidade, os sistemas de HPC viabilizam a execução de grandes volumes de dados em processamento [5]. Geralmente, essas arquiteturas utilizam supercomputadores ou clusters compostos por milhares de nós, cada um equipado com CPUs *multi-core* de alto desempenho ou GPUs, interconectados por redes de baixa latência. A arquitetura típica consiste em nós que executam tarefas de forma coordenada sob um agendador central, acelerando o processamento de grande volume de dados e permitindo a resolução de problemas como simulações científicas e algoritmos de inteligência artificial [6].

A base de todo esse poder de processamento é a paralelização. Em vez de executar uma instrução por vez como na abordagem sequencial, a execução paralela distribui o trabalho e realiza múltiplas operações ao mesmo tempo [7]. Para que as placas de vídeo consigam explorar esse potencial de *hardware*, é indispensável utilizar plataformas e interfaces de *software* específicas, como o ROCm e o HIP, que assumem o controle do *hardware* gráfico e otimizam a carga de trabalho [7].

O uso contínuo de placas de vídeo para essas tarefas gerais popularizou o paradigma GPGPU (*General-Purpose computing on Graphics Processing Units*). Enquanto as CPUs são projetadas para entregar baixa latência em processos sequenciais, as GPUs possuem uma arquitetura focada em vazão (*throughput*). Elas utilizam o modelo SIMD (*Single Instruction, Multiple Data*), onde múltiplas unidades de processamento executam a mesma instrução em diferentes dados simultaneamente [7].

Na prática, a computação ocorre em um modelo heterogêneo: a CPU atua como *host* gerenciando a aplicação, e a GPU atua como *device* executando as funções mais pesadas, conhecidas como *kernels*. Para mascarar o tempo de acesso à memória e manter os núcleos ocupados, as GPUs gerenciam milhares de *threads* concorrentes. Como consequência, o desenvolvedor precisa controlar manualmente a transferência de dados entre a memória RAM e a placa de vídeo [8].

Justamente pela exigência desse controle manual da memória e dos núcleos gráficos, o desenvolvimento em GPGPU depende de ecossistemas de *software* robustos, como a plataforma ROCm detalhada a seguir.

B. ROCm

ROCm (*Radeon Open Compute Platform*) é a plataforma de *software* de código aberto da AMD, projetada para viabilizar a computação de alto desempenho (HPC) e aplicações de inteligência artificial em suas GPUs. O ROCm funciona como uma alternativa completa ao ecossistema proprietário CUDA da NVIDIA, oferecendo não apenas um conjunto de ferramentas, mas uma pilha de *software* completa [9].

Em sua base, a plataforma inclui *drivers* de baixo nível que fazem a interface direta com o *hardware* da GPU. Acima dessa camada, há um compilador baseado em LLVM *(Low Level Virtual Machine)*, que é responsável por traduzir o código escrito em linguagens de alto nível, como HIP, para o código de máquina executado pela GPU. A interação entre a aplicação e o dispositivo é gerenciada pelo HIP Runtime, uma API que fornece as funções essenciais para alocação de memória na GPU, transferência de dados e o lançamento dos *kernels* [9].

O ecossistema ROCm foi arquitetado e centralizado para ambientes Linux, visando supercomputadores e *clusters* de alta densidade. Contudo, para democratizar o desenvolvimento em estações de trabalho convencionais, a AMD expandiu a sua compatibilidade para o sistema operacional Windows através do lançamento do AMD HIP SDK. Este pacote atua como um subconjunto da plataforma ROCm, fornecendo aos desenvolvedores em ambiente Windows o compilador HIP-Clang (hipcc) e as

bibliotecas de runtime essenciais para a tradução e execução nativa de códigos GPGPU nas placas gráficas Radeon, mantendo a interoperabilidade e os ganhos de desempenho da arquitetura [9].

C. HIP

HIP (*Heterogeneous-compute Interface for Portability*) é a interface de programação (API) e linguagem de *kernel* em C++ utilizada no ecossistema ROCm. O principal objetivo do HIP é fornecer uma camada de portabilidade para o código GPGPU. Com ele, é possível escrever um único código-fonte que pode ser compilado para rodar tanto em GPUs AMD (através do ROCm) quanto em GPUs NVIDIA (onde as chamadas HIP são traduzidas para a API do CUDA). A sintaxe do HIP é intencionalmente quase idêntica à do CUDA, facilitando a migração de aplicações existentes e o desenvolvimento de *software* heterogêneo [10].

O modelo de programação do HIP baseia-se no paradigma *Single Instruction, Multiple Threads* (SIMT). Na prática, isso significa que uma mesma instrução é enviada para ser executada simultaneamente por um grande número de *threads*. A aplicação funciona em um modelo heterogêneo com duas partes centrais: o *host* (geralmente a CPU), que orquestra o fluxo do programa e a distribuição de dados, e o *device* (a GPU), que atua executando os trechos de alto custo computacional, conhecidos como *kernels*. No código, o desenvolvedor utiliza os qualificadores `__host__` e `__device__` para sinalizar onde cada função será processada [10].

Como a CPU e a GPU possuem espaços de memória físicos separados, cabe ao programador gerenciar explicitamente a movimentação de informações entre elas. Para reservar espaço na placa de vídeo, utiliza-se a função de alocação `hipMalloc`. O envio dos dados entre a memória RAM e a memória da GPU é realizado pelo comando `hipMemcpy`, que exige parâmetros para indicar a direção exata da cópia (do *host* para o *device*, ou o contrário). Em situações específicas, o HIP oferece também o recurso de memória *zero-copy* (via `hipHostMalloc`), permitindo que a GPU acesse a memória física do *host* diretamente pelo barramento [10].

A arquitetura das GPUs AMD permite o gerenciamento de milhares de *threads* leves diretamente pelo *hardware*, operando de forma independente do escalonador do Sistema Operacional. O lançamento de um *kernel* a partir do *host* é realizado através da sintaxe de *chevrons* «< »> ou pela função de runtime `hipLaunchKernelGGL` [10].

Para viabilizar a execução paralela, o HIP organiza as *threads* em uma estrutura hierárquica: as *threads* são agrupadas em blocos (*blocks*), que por sua vez compõem uma grade de computação (*grid*). A configuração desta geometria lógica é definida pelo desenvolvedor através da estrutura vetorial `dim3`. Durante a execução no *device*, cada *thread* mapeia sua porção específica de dados utilizando variáveis nativas de indexação, como `threadIdx` (índice da *thread* dentro do bloco) e `blockIdx` (índice do bloco dentro da grade) [10].

Por ser uma arquitetura assíncrona, o HIP exige sincronização cuidadosa. Para que a CPU aguarde a finalização dos cálculos na GPU, utiliza-se o comando `hipDeviceSynchronize()`. Dentro do *device*, para garantir que todas as *threads* de um bloco alcancem o mesmo ponto antes de prosseguir, utiliza-se a barreira `__syncthreads()` [10].

Durante a execução paralela, pode ocorrer o acesso a uma seção crítica, onde múltiplas *threads* tentam atualizar o mesmo espaço de memória simultaneamente. Para resolver isso, o HIP oferece operações atômicas baseadas em *hardware*, como o comando `atomicAdd`, que garante que a leitura, modificação e escrita sejam feitas de forma ininterrupta, impedindo conflitos de acesso [10].

D. Linguagem de programação C++

C++ é uma linguagem de programação de propósito geral e alto desempenho desenvolvida por Bjarne Stroustrup no início da década de 1980 como uma extensão da linguagem C. Sua principal característica é combinar a eficiência e a proximidade com o *hardware* herdadas do C com recursos avançados de abstração, incluindo conceitos de programação orientada a objetos, como classes, herança e polimorfismo. Essa versatilidade permite combinar programação estruturada e orientada a objetos, tornando-a adequada para sistemas complexos em que desempenho e gerenciamento de recursos são essenciais. Ao abstrair detalhes de baixo nível sem comprometer a velocidade de execução, o C++ consolidou-se como uma ferramenta fundamental no desenvolvimento de sistemas operacionais, simuladores e aplicações de alto desempenho [11].

Com as bases de *hardware* e de linguagem de programação devidamente definidas, o foco direciona-se para o problema prático a ser resolvido pelo paralelismo: a aceleração da criptografia de dados, utilizando o algoritmo AES.

E. Algoritmo AES

O *Advanced Encryption Standard* (AES) é um algoritmo de criptografia estabelecido pelo *National Institute of Standards and Technology* (NIST) dos Estados Unidos em 2001, como resultado de um processo público que durou anos para encontrar um sucessor para o antigo *Data Encryption Standard* (DES). O DES, com sua chave de 56 bits, tornou-se vulnerável a ataques de força bruta devido ao avanço do poder computacional [12].

O AES é uma cifra de bloco simétrica, o que significa que utiliza a mesma chave tanto para cifrar quanto para decifrar os dados. O algoritmo opera em blocos de tamanho fixo de 128 bits (16 bytes) e pode utilizar chaves de 128, 192 ou 256 bits. A segurança do AES deriva da aplicação de uma série

de transformações matemáticas sobre os dados, repetidas em múltiplos ciclos conhecidos como "rodadas" (*rounds*). Neste trabalho, foi adotado o modo de operação *Electronic Codebook* (ECB), no qual cada bloco é processado de forma independente, favorecendo a paralelização em GPU [12].

Internamente, o AES representa cada bloco de 128 bits como uma matriz de 4x4 bytes, chamada de Estado (*State*). Todas as operações do algoritmo são realizadas sobre essa matriz. Antes do início do ciclo de criptografia, a chave original passa por um processo de Expansão de Chave (*Key Expansion*), que gera um conjunto de sub-chaves, uma para cada rodada do algoritmo [12].

O processo de criptografia de um bloco de dados pode ser observado na figura 1. Ele inicia com uma operação `AddRoundKey` preliminar. Em seguida, o Estado passa por um ciclo de rodadas principais, onde cada rodada (com exceção da última) é composta por quatro etapas distintas, aplicadas em sequência:

1) `SubBytes`: Uma etapa de substituição não linear que visa introduzir "confusão" no processo. Cada byte do Estado é substituído por um novo valor, consultado a partir de uma tabela predefinida de 256 bytes chamada S-Box.
2) `ShiftRows`: Uma etapa de permutação que promove a difusão dos dados. As três últimas linhas da matriz do Estado são rotacionadas ciclicamente para a esquerda por um número diferente de posições para cada linha, espalhando a influência de cada byte para outras colunas.
3) `MixColumns`: Uma operação de mistura que também contribui para a difusão. Cada coluna da matriz do Estado é transformada através de uma multiplicação por uma matriz fixa em um corpo finito (*Galois Field*), combinando os quatro bytes da coluna de forma complexa.
4) `AddRoundKey`: Nesta etapa final da rodada, o Estado é combinado com a sub-chave correspondente àquela rodada por meio de uma operação XOR a nível de bits.

Após a execução de todas as rodadas principais, uma rodada final é realizada. Ela é idêntica a uma rodada comum, mas omite a etapa `MixColumns` para garantir que o processo seja reversível na decifragem [12].

O processo de descriptografia do AES, por ser uma cifra simétrica, utiliza a mesma chave criptográfica e o processo inverso das operações de criptografia. As operações são realizadas em ordem inversa àquela aplicada na criptografia, começando com a sub-chave final e aplicando as funções inversas em cada etapa [2]. As funções inversas são: `InvShiftRows` (inversa de `ShiftRows`), `InvSubBytes` (inversa de `SubBytes`), e `InvMixColumns` (inversa de `MixColumns`). A operação `AddRoundKey` é sua própria inversa, pois o XOR com a mesma chave duas vezes retorna ao estado original. A descriptografia passa por um estágio inicial (`AddRoundKey` com a sub-chave final), o ciclo de rodadas principais (com as operações inversas em ordem revertida) e uma rodada final, seguindo o padrão inverso da criptografia [12].

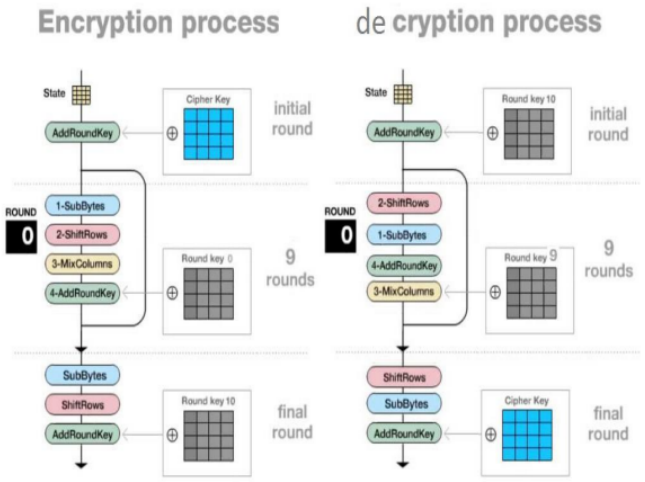

Figura 1. Visualização do processo de Criptografia e Descriptografia do Algoritmo AES [2].

A estrutura lógica detalhada anteriormente é sintetizada no pseudocódigo da função `Cipher`, apresentado na Figura 2. O algoritmo recebe como entrada um bloco de dados de 128 bits (`in`), a chave expandida (`w`) e o número de rodadas (N_r), e produz como saída o bloco cifrado (`out`). Após a cópia inicial do bloco para a matriz de Estado, ocorre a primeira operação `AddRoundKey`. O ciclo subsequente executa as transformações `SubBytes`, `ShiftRows`, `MixColumns` e `AddRoundKey` repetidamente até a penúltima rodada ($N_r - 1$). Na rodada final, a etapa de mistura `MixColumns` é omitida para garantir que o processo seja reversível durante a descriptografia, finalizando a operação com o mapeamento do Estado para o bloco de saída.

```
Cipher(byte in[4*Nb], byte out[4*Nb], word w[Nb*(Nr+1)])
begin
    byte state[4, Nb]
    state = in
    AddRoundKey(state, w[0, Nb-1])
    for round = 1 step 1 to Nr-1
        SubBytes(state)
        ShiftRows(state)
        MixColumns(state)
        AddRoundKey(state, w[round*Nb, (round+1)*Nb-1])
    end for
    SubBytes(state)
    ShiftRows(state)
    AddRoundKey(state, w[Nr*Nb, (Nr+1)*Nb-1])
    out = state
end
```

Figura 2. Pseudocódigo do processo de criptografia [12].

Para o desenvolvimento da paralelização, foi adotada uma metodologia ágil de gerenciamento para organizar as etapas de desenvolvimento do projeto.

F. Metodologia de Gerenciamento Ágil - Scrum

O Scrum é um *framework* para o gerenciamento de projetos complexos, que estrutura o desenvolvimento em ciclos iterativos e incrementais chamados *Sprints* [13].

O *framework* é composto por três elementos centrais. O primeiro é o *Time Scrum*, que consiste em três papéis: o *Product Owner*, responsável por maximizar o valor do produto e gerenciar o *Product Backlog*; o *Development Team*, composto pelos profissionais que realizam o trabalho de criar um Incremento; e o *Scrum Master*, responsável por garantir que o Scrum seja compreendido e aplicado [13].

O segundo elemento são os eventos do Scrum, usados para criar regularidade. Os eventos são: o *Sprint*, o ciclo de trabalho, a *Sprint Planning*, onde o trabalho é planejado, a *Daily Scrum*, que é uma reunião curta de alinhamento, a *Sprint Review*, onde o resultado é inspecionado e a *Sprint Retrospective*, onde a equipe reflete sobre melhorias [13].

O terceiro elemento são os artefatos do Scrum, que representam o trabalho. Os artefatos principais são: o *Product Backlog*, uma lista ordenada de tudo o que é necessário para o produto, o *Sprint Backlog*, o conjunto de itens selecionados para um *Sprint*) e o *Incremento*, a soma de todos os itens concluídos [13].

III. Trabalhos Correlatos

Nesta seção, são apresentados os estudos que servem de base para este projeto, abrangendo desde a viabilidade de paralelizar o AES até o uso de ferramentas de portabilidade em plataformas abertas.

O ponto de partida deste trabalho é a demonstração de que a execução do algoritmo AES em GPUs traz ganhos de desempenho significativos. Isso foi evidenciado por Tybusch [2], que alcançou acelerações de até 424% utilizando a plataforma proprietária CUDA. Esta pesquisa avança nessa ideia ao transpor o problema para o ecossistema aberto ROCm, da AMD. Outro ponto importante é reforçado por Praxedes [14], que mostra como o paradigma GPGPU é uma ferramenta indispensável para otimização de performance, o que justifica buscar um controle do *hardware* por meio da interface HIP.

A escolha do HIP como ferramenta principal também se apoia nos conceitos discutidos por Carvalho Junior [15]. O estudo explica que, devido à crescente variedade de *hardwares*, é essencial adotar estratégias de *platform-aware programming* para criar códigos que sejam modulares e fáceis de portar. Além disso, a eficiência dessa escolha é confirmada na prática por Farina [3]. Em seu estudo sobre o método de Monte Carlo, a migração de CUDA para ROCm utilizando HIP apresentou uma perda de desempenho mínima, ficando abaixo de 1% em cenários de alta demanda computacional.

Em resumo, este TCC une essas referências: utiliza os testes de desempenho de Tybusch [2] como motivação, fundamenta-se na necessidade de portabilidade discutida por Carvalho Junior [15] e busca alcançar a eficiência de implementação observada por Farina [3], focando especificamente na paralelização do AES dentro da plataforma ROCm.

IV. Metodologia

A metodologia para a execução prática deste projeto consistiu na implementação e análise de desempenho das solução de *software* propostas. Para o gerenciamento do ciclo de desenvolvimento, foi utilizado o framework ágil Scrum.

A. Gerenciamento do Projeto com Scrum

Os papéis no projeto foram definidos da seguinte forma:

- *Product Owner:* este papel foi desempenhado pela professora orientadora, responsável por definir os objetivos, priorizar as funcionalidades e validar as entregas ao final de cada ciclo.
- *Development Team:* este papel foi desempenhado pelo acadêmico, responsável por realizar as tarefas de pesquisa, implementação, teste e escrita do trabalho.

Os artefatos do Scrum foram utilizados para gerenciar o trabalho e o progresso:

- *Product Backlog:* Foi criada uma lista ordenada de todas as atividades necessárias para a conclusão do projeto. Para este trabalho, o *backlog* detalha as funcionalidades e entregáveis (Quadro I).

Quadro I
Product Backlog das tarefas.

ID	Item do Backlog (Tarefa)	Sprint
PB-01	Ambiente de dev. (ROCm/HIP) configurado	1
PB-02	Validação da GPU com "Hello World" HIP	1
PB-03	Versão serial (CPU) do AES	2
PB-04	Kernel HIP do AES para GPU	3
PB-05	Lógica Host-Device (transferência de dados)	4
PB-06	Scripts de automação de testes	5
PB-07	Coleta de métricas (Tempo, Throughput)	5
PB-08	Cálculo de Speedup e geração de gráficos	6
PB-09	Análise de resultados e conclusões	6

- *Sprint Backlog:* No início de cada ciclo, um conjunto de itens do *Product Backlog* (Quadro I) foi selecionado para formar o *Sprint Backlog*, representando o plano de trabalho de cada *Sprint*.

B. Plano de Sprints para o Desenvolvimento

O desenvolvimento do projeto foi executado ao longo de seis *Sprints*, cada uma com uma meta específica para garantir a entrega de avanços no projeto.

- Sprint 1: Configuração e Validação do Ambiente
  - Meta: Preparar e validar todo o ambiente de desenvolvimento.
  - Atividades: Instalação do Windows 11 e da plataforma ROCm por meio do HIP SDK versão 7.1.1. O *sprint* foi concluído com a execução bem-sucedida de um programa de teste em HIP para confirmar a comunicação com a GPU.
- Sprint 2: Implementação da versão de CPU Serial
  - Meta: Criar uma versão funcional e correta do algoritmo AES para CPU.
  - Atividades: Desenvolvimento em C++ da lógica completa do AES (AES-128) para criptografar e descriptografar arquivos. Foi adotado o modo de operação ECB (*Electronic Codebook*), que processa cada bloco de forma independente.
- Sprint 3: Desenvolvimento do Kernel para GPU
  - Meta: Portar a lógica de criptografia do AES para um *kernel* executável na GPU.
  - Atividades: Adaptação do código C++ para a sintaxe do HIP.
- Sprint 4: Integração *Host-Device*
  - Meta: Construir a aplicação completa que gerencia a execução do *kernel* na GPU.
  - Atividades: Implementação no código *host* (CPU) da lógica para alocação de memória no *device* (GPU), a transferência de dados, o lançamento do *kernel* e a cópia dos resultados de volta.
- Sprint 5: Automação de Testes e Coleta de Dados
  - Meta: Preparar e executar a estrutura para a avaliação de desempenho.
  - Atividades: Desenvolvimento de scripts para automatizar a execução das duas versões (CPU e GPU) com diferentes tamanhos de arquivos, coletando e armazenando as métricas de tempo de execução.
- Sprint 6: Análise de Resultados e Redação Final
  - Meta: Concluir a análise dos dados e a documentação do trabalho.
  - Atividades: Tabulação dos dados coletados, geração dos gráficos de desempenho (Tempo, *Throughput* e *Speedup*), e redação das seções de análise dos resultados e das conclusões finais do TCC.

C. Ambiente de Desenvolvimento e Teste

As especificações do ambiente onde o projeto foi executado são:

- *Hardware*: Processador AMD Ryzen 5 5600G (6 núcleos, 12 *threads*), Placa de Vídeo XFX Speedster SWFT210 Radeon RX 7600 (8GB GDDR6), 16GB de Memória RAM DDR4 3000MHz e SSD Kingston A400 de 480GB.
- *Software*: Sistema Operacional Windows 11, editor de código Visual Studio Code, e compilador G++ (GCC) para a versão serial em CPU. Para a execução paralela em GPU, utilizou-se a linguagem HIP através do pacote oficial AMD HIP SDK para Windows, que fornece o compilador HIP-Clang (hipcc) e as bibliotecas de runtime necessárias para interfacear a aplicação com a placa gráfica.

D. Procedimentos de Teste e Avaliação

A avaliação de desempenho das duas implementações foi conduzida com:

- Cenários de Teste: Foram utilizados como entrada arquivos de dados com tamanhos variando entre 1 KB e 1 GB permitindo analisar o comportamento dos algoritmos tanto em cenários limitados pela latência da transferência de dados quanto pela capacidade de processamento.
- Métricas de Desempenho: Para cada execução, foram coletadas as seguintes métricas: o Tempo de Execução (em segundos); o *Throughput* (em MB/s); e o *Speedup*.

O *Speedup* (Aceleração) foi calculado pela razão de desempenho, conforme a Equação 1:

$$\text{Speedup} = \frac{\text{Tempo da Versão Serial}}{\text{Tempo da Versão Paralela}} \tag{1}$$

Os resultados foram tabulados e plotados em gráficos para permitir uma análise comparativa clara, buscando identificar o ponto de inflexão a partir do qual a implementação em GPU se torna mais vantajosa e quantificar o impacto do *overhead* da comunicação entre CPU e GPU.

Por fim, além de medir o desempenho, a implementação passou por uma validação funcional para garantir que os cálculos do AES estavam exatos. Para atestar essa conformidade matemática, os testes foram executados utilizando os vetores de teste oficiais fornecidos na própria documentação do algorítmo AES [12]. A verificação confirmou que o conteúdo final descriptografado voltou a ser exatamente igual ao arquivo original, comprovando que tanto a versão em CPU quanto a versão em GPU processam os dados sem erros.

V. Desenvolvimento

Nessa seção, detalha-se o processo de implementação das duas versões do algoritmo AES-128: a abordagem sequencial e a abordagem paralela utilizando a arquitetura GPGPU.

A. *Implementação do Algoritmo Sequencial*

A versão sequencial do AES-128 foi desenvolvida em C++ de forma a servir como base de controle (*baseline*) para a avaliação de desempenho. Em conformidade com as especificações do padrão FIPS 197 detalhadas no referencial teórico, a implementação adotou o modo de operação *Electronic Codebook* (ECB). A escolha deste modo foi feita exclusivamente por conta da sua característica de independência estrita entre os blocos onde cada bloco de 16 bytes é processado sem depender do bloco anterior. Essa característica matemática favorece testar o paralelismo massivo em GPUs, que é o objetivo central de avaliação deste trabalho.

A lógica de criptografia foi estruturada para refletir o pseudocódigo do algoritmo (apresentado anteriormente na Figura 2). A execução ocorre inteiramente na Unidade Central de Processamento (CPU), onde as tabelas de substituição (S-Box) são alocadas na memória RAM principal.

Como demonstrado na Figura 3, a função principal itera sobre as 10(Nr) rodadas do AES-128. Após a operação inicial de `AddRoundKey`, o bloco de estado de 16 bytes passa sequencialmente pelas transformações não-lineares (`SubBytes`), permutações (`ShiftRows`) e mistura linear (`MixColumns`), finalizando com a adição da sub-chave da rodada atual.

```
void Cipher(uint8_t* state) {
    // AddRoundKey inicial (rodada 0)
    for (int i = 0; i < 16; ++i)
        state[i] ^= RoundKey[i];

    // Ciclo de rodadas principais (1 a nr)
    for (uint8_t round = 1; round <= Nr; ++round) {
        // SubBytes (Substituicao via S-Box)
        for (int i = 0; i < 16; ++i)
            state[i] = sbox[state[i]];

        shiftRows(state); // Permutacao

        // MixColumns
        if (round < Nr) mixColumns(state);

        // AddRoundKey
        for (int i = 0; i < 16; ++i)
            state[i] ^= RoundKey[round * 16 + i];
    }
}
```

Figura 3. Trecho da implementação da criptografia do AES em C++.

Apesar de funcional, essa abordagem evidencia o limite do processamento para arquivos de grandes dimensões, a CPU processa um bloco de cada vez de forma iterativa, resultando em um tempo de execução diretamente proporcional e linear ao tamanho da carga de trabalho.

Foi implementada a descriptografia tanto na versão sequencial quanto na versão paralela, utilizando as transformações inversas definidas pelo padrão AES para restaurar os dados originais a partir do texto cifrado.

B. *Projeto e Implementação do Algoritmo Paralelo*

Para a versão paralela, o paradigma adotado foi o Paralelismo de Dados (*Data Parallelism*), aproveitando a independência dos blocos no modo ECB. A arquitetura mapeou exatamente um bloco de 16 bytes do AES para uma *thread* individual da GPU, permitindo que milhões de blocos sejam cifrados simultaneamente pelos processadores de fluxo (*stream processors*).

Foi feita uma separação das responsabilidades entre o *Host* (CPU) e o *Device* (GPU). A expansão de chaves, sendo um processo sequencial, rápido e de execução única, foi mantida na CPU. Apenas as sub-chaves resultantes e a carga de dados massiva são transferidas para a VRAM*(Video Random Access Memory)* através do barramento PCIe.

Para a execução do *kernel* na placa de vídeo, a malha de computação (*grid*) foi configurada de forma dinâmica com base no tamanho do arquivo de entrada. Adotou-se a divisão lógica em blocos unidimensionais compostos por 256 *threads* cada, uma métrica padrão na programação heterogênea voltada a maximizar a ocupação dos multiprocessadores de fluxo da arquitetura da placa de vídeo sem esgotar o limite físico de registradores por bloco.

Para maximizar o *throughput* e diminuir a latência inerente ao acesso à memória de vídeo global, aplicou-se a otimização da hierarquia de memória no interior do *kernel*. A tabela S-Box foi declarada como constante na memória (`__constant__`), permitindo o acesso via *cache*. Adicionalmente, conforme ilustrado na Figura 4, as matrizes de estado foram carregadas para variáveis locais. Isso força o *hardware* a realizar as operações aritméticas diretamente nos registradores físicos de cada *thread*.

```
__global__ void aes_encrypt_kernel(uint8_t* d_data, const uint8_t* d_RoundKey, size_t num_blocks) {
    size_t idx = blockIdx.x * blockDim.x + threadIdx.x;
    if (idx >= num_blocks) return;

    size_t offset = idx * 16;
    uint8_t state[16]; // Matriz local (Registradores)

    // Copia da memoria global para mitigar latencia
    for (int i = 0; i < 16; ++i) state[i] = d_data[offset + i];

    // AddRoundKey Inicial
    for (int i = 0; i < 16; ++i) state[i] ^= d_RoundKey[i];

    // Execucao das 10 rodadas independentes por thread
    for (uint8_t round = 1; round <= Nr; ++round) {
        // SubBytes via Memoria Constante (__constant__)
        for (int i = 0; i < 16; ++i) state[i] = d_sbox[state[i]];

        shiftRows_device(state);
        if (round < Nr) mixColumns_device(state);

        for (int i = 0; i < 16; ++i)
            state[i] ^= d_RoundKey[round * 16 + i];
    }

    // Escrita do bloco cifrado de volta na memoria global
    for (int i = 0; i < 16; ++i) d_data[offset + i] = state[i];
}
```

Figura 4. Kernel HIP mapeando as etapas do AES para execução em registradores locais da GPU.

Essa reestruturação garante que a execução das etapas principais ocorra de forma otimizada na arquitetura da GPU,

preservando a integridade matemática do algoritmo enquanto explora a vazão extrema do *hardware* gráfico.

C. Automação dos Testes e Coleta de Métricas

Para automatizar a coleta de dados e evitar erros de medição manual, foi desenvolvido um *script* em Python, responsável por gerenciar a execução dos testes de desempenho.

O *script* cria automaticamente os arquivos de teste nos tamanhos definidos, variando de 1 KB a 1 GB. Utilizando a biblioteca `subprocess`, o código executa os programas (`aes_serial.exe` e `aes_hip.exe`) e passa os parâmetros de linha de comando para iniciar a criptografia (`-e`) e, em seguida, a descriptografia (`-d`).

As informações exibidas no terminal durante a execução são lidas pelo Python e processadas usando expressões regulares (biblioteca `re`) para extrair exatamente o Tempo de Execução e o *Throughput* (MB/s). A partir do tempo de execução, o *script* calcula o *Speedup* de cada cenário.

Por fim, os dados coletados são organizados e plotados usando a biblioteca `matplotlib`. Ela gera automaticamente os painéis de gráficos, possibilitando a comparação visual entre o rendimento da CPU e da placa de vídeo em diferentes tamanhos de arquivo. O código-fonte completo do projeto está disponível publicamente no GitHub [16].

VI. Resultados

Para garantir uma avaliação justa e completa, os testes de desempenho foram executados no ambiente computacional descrito anteriormente na Seção Metodologia, em Procedimento de Testes e Avaliação. Essa variação foi escolhida com o propósito de observar como o sistema se comporta tanto em tarefas rápidas quanto em cargas extremas de processamento.

Durante as execuções, o sistema monitorou três métricas principais. A primeira foi o Tempo de Execução, para saber a duração total do processo. A segunda foi o *Throughput* (Vazão), que mede a velocidade contínua de transferência de dados em MB/s. A terceira foi o *Speedup* (Aceleração), que serve para comparar de forma direta quantas vezes a placa de vídeo conseguiu ser mais rápida do que o processador central para a mesma tarefa.

Um ponto é que os tempos medidos representam a execução completa (do tipo fim-a-fim). Isso significa que, na versão em HIP, a contagem de tempo inclui o custo (*overhead*) de transferir os dados da memória RAM para a placa de vídeo através do barramento PCIe, e depois trazer de volta. Avaliar o processo considerando esse tempo de comunicação garante que os resultados mostrem o desempenho real que um usuário teria na prática, sem mascarar as desvantagens da GPU.

A Figura 5 apresenta a saída gerada pelo *script* de automação no terminal do sistema. Essa imagem mostra todas as métricas exatas coletadas em ambas as baterias de testes, servindo como a base de dados para a geração dos gráficos analisados a seguir.

MÉTRICAS DE CRIPTOGRAFIA (-e)

Tamanho	Tempo CPU (s)	Tempo GPU (s)	Throughput CPU	Throughput GPU	Speedup
1KB	0.000016	0.010311	61.60 MB/s	0.10 MB/s	0.00x
100KB	0.001445	0.010304	67.60 MB/s	9.48 MB/s	0.14x
1MB	0.015255	0.013248	65.55 MB/s	75.48 MB/s	1.15x
10MB	0.159771	0.014648	62.59 MB/s	682.69 MB/s	10.91x
100MB	1.504089	0.046850	66.49 MB/s	2134.46 MB/s	32.10x
500MB	7.838482	0.215342	63.79 MB/s	2321.89 MB/s	36.40x
1GB	15.749577	0.396890	65.02 MB/s	2580.06 MB/s	39.68x

MÉTRICAS DE DESCRIPTOGRAFIA (-d)

Tamanho	Tempo CPU (s)	Tempo GPU (s)	Throughput CPU	Throughput GPU	Speedup
1KB	0.000021	0.014820	45.63 MB/s	0.07 MB/s	0.00x
100KB	0.002034	0.011269	48.00 MB/s	8.67 MB/s	0.18x
1MB	0.020050	0.011008	49.87 MB/s	90.85 MB/s	1.82x
10MB	0.201599	0.014468	49.60 MB/s	691.17 MB/s	13.93x
100MB	2.027018	0.046702	49.33 MB/s	2141.22 MB/s	43.40x
500MB	10.067386	0.203674	49.67 MB/s	2454.90 MB/s	49.43x
1GB	20.576688	0.403454	49.77 MB/s	2538.08 MB/s	51.00x

Figura 5. Saída do terminal exibindo as métricas brutas coletadas para os processos de criptografia e descriptografia.

Para facilitar a análise detalhada e a visualização gráfica, os resultados foram discutidos separadamente entre os processos de criptografia e descriptografia.

A. Avaliação do Processo de Criptografia

A Figura 6 apresenta os resultados de *throughput* obtidos durante a etapa de criptografia. A versão executada na CPU manteve uma média estável de aproximadamente 68 MB/s em todos os tamanhos de arquivo testados. Por outro lado, a versão paralela na GPU demonstrou um crescimento rápido na vazão de dados, estabilizando em torno de 2500 MB/s para os arquivos maiores.

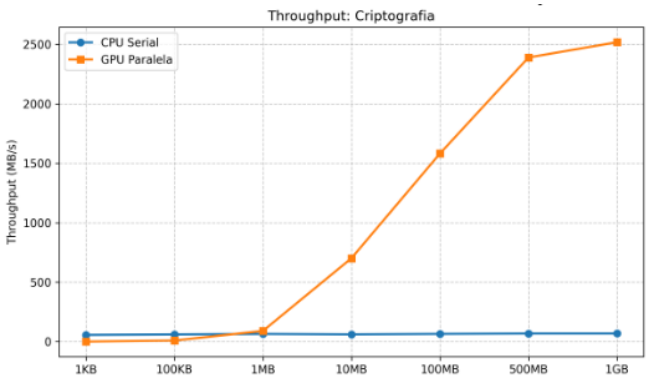

Figura 6. Throughput efetivo (MB/s) registrado durante a operação de Criptografia.

Essa diferença no volume de dados processados reflete diretamente na métrica de *Speedup*, conforme mostra a Figura 7. O gráfico ilustra um comportamento já esperado: em arquivos pequenos (de 1 KB a 100 KB), a versão sequencial (CPU) tem um desempenho melhor. Isso acontece porque o tempo gasto para transferir os dados entre as

memórias é maior do que o tempo de processamento em si, gerando uma aceleração inferior a 1 (GPU mais lenta que a CPU).

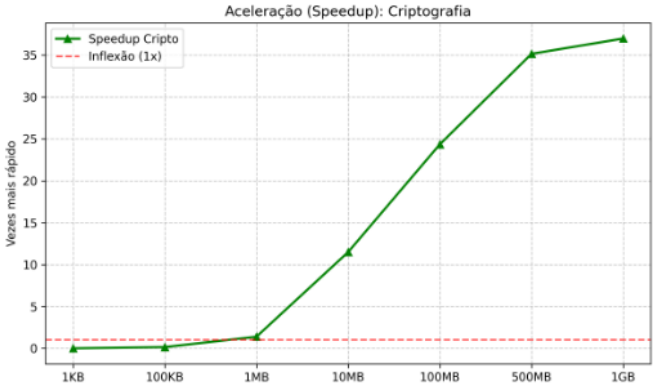

Figura 7. Curva de Aceleração (Speedup) e ponto de inflexão para a Criptografia.

No entanto, a partir de arquivos de 1 MB, ocorre o ponto de inflexão (*Crossing Point*). O custo de transferência se torna pequeno em relação ao tempo total de cálculo. No teste com o arquivo de 1 GB, a implementação na placa de vídeo foi cerca de 36 vezes mais rápida que a da CPU.

B. Avaliação do Processo de Descriptografia

Em relação ao processo de descriptografia, os resultados indicaram uma mudança no comportamento da versão sequencial. Como pode ser visto na Figura 8, enquanto a GPU manteve o seu rendimento na faixa de 2500 MB/s, o *throughput* da CPU caiu para aproximadamente 48 MB/s.

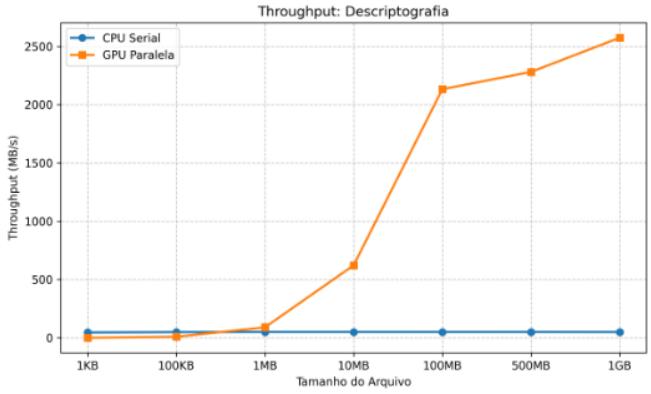

Figura 8. Throughput efetivo (MB/s) registrado durante a operação de Descriptografia.

Essa queda de desempenho na CPU ocorre devido à complexidade matemática da descriptografia do algoritmo AES. A etapa `InvMixColumns` exige operações de multiplicação mais complexas no corpo de Galois em comparação com a etapa de criptografia.

Devido a essa diminuição de desempenho na CPU e à estabilidade da GPU, os valores de *Speedup* na descriptografia foram ainda maiores. A Figura 9 mostra que, para o arquivo de 1 GB, a implementação em HIP conseguiu executar a tarefa de forma 52 vezes mais rápida do que a versão sequencial em C++.

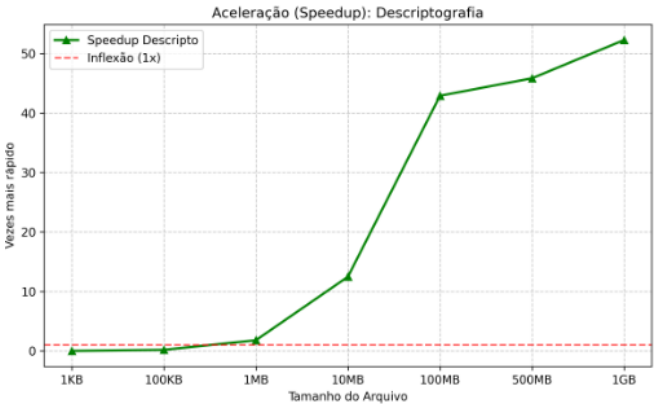

Figura 9. Curva de Aceleração (Speedup) e ponto de inflexão para a Descriptografia.

C. Síntese dos Resultados

A partir da análise das métricas coletadas, é possível destacar dois comportamentos centrais do sistema. O primeiro é que o custo de transferência de dados pelo barramento PCIe cria um limite prático na marca de 1 MB, abaixo do qual o tempo de comunicação acaba penalizando o uso da GPU. O segundo é a alta capacidade da placa de vídeo visto o aumento da complexidade matemática na etapa `InvMixColumns`. Enquanto a CPU sofreu uma queda perceptível no seu rendimento durante a descriptografia, a GPU conseguiu manter uma vazão constante na casa dos 2500 MB/s.

Dessa forma, os dados comprovam o ganho real da paralelização no ambiente testado. A implementação com HIP superou os gargalos da CPU e os custos de tráfego de memória, alcançando uma aceleração de até 52 vezes nas cargas de trabalho mais pesadas.

VII. Conclusões e Trabalhos Futuros

Este estudo comprovou a eficácia da paralelização do AES-128 em GPUs utilizando HIP sobre a plataforma ROCm. A implementação alcançou acelerações de até 52 vezes na descriptografia e 36 vezes na criptografia para arquivos de 1 GB, mantendo throughput próximo de 2500 MB/s. Entretanto, o custo de transferência pelo barramento PCIe estabelece um ponto de inflexão em torno de 1 MB, abaixo do qual a execução sequencial permanece mais eficiente.

Os resultados obtidos também evidenciam a evolução das tecnologias de computação paralela ao longo da última década. Em comparação ao trabalho de Tybusch [2], que utilizou OpenCL e uma GPU Radeon HD5870, obtendo

ganhos médios entre 3 e 5 vezes no desempenho, a presente implementação alcançou acelerações consideravelmente superiores utilizando HIP sobre ROCm e uma GPU Radeon RX 7600. Essa diferença demonstra não apenas os avanços do hardware gráfico moderno, mas também a maturidade das ferramentas de desenvolvimento para computação heterogênea.

Como limitações, o trabalho utilizou exclusivamente o modo ECB (Electronic Codebook), adequado para avaliação de paralelismo, mas inadequado para aplicações modernas. Além disso, os experimentos foram realizados em uma única plataforma (RX 7600) e com cargas sintéticas, restringindo a generalização dos resultados.

Como trabalhos futuros, pretende-se expandir o kernel HIP para suportar os modos CTR (Counter Mode) e GCM (Galois/Counter Mode), validar a solução em diferentes ambientes HPC e implementar suporte às variantes AES-192 e AES-256.

Referências

[1] Amit Mehta. "High Performance Computing: Trends, Technologies and Applications". **in***International Journal of Advanced Intelligent Systems and Machine Learning*: 2.1 (2021), **pages** 20–27. URL: https://ijaidsml.org/index.php/ijaidsml/article/download/26/24.

[2] Douglas Tybusch **and** Gustavo Stangherlin Cantarelli. *Paralelização do Algoritmo AES e Análise Sobre GPGPU*. https://tfgonline.lapinf.ufn.edu.br/media/midias/Douglas_Tybusch.pdf. Acesso em 2 de setembro de 2025.

[3] Bruno Longo Farina. *Análise da Portabilidade de Desempenho do Método de Monte Carlo em GPUs*. Monografia (Bacharelado em Ciência da Computação) — Instituto de Informática, Universidade Federal do Rio Grande do Sul. Disponível em: https://lume.ufrgs.br/bitstream/handle/10183/294746/001289974.pdf?sequence=1\&isAllowed=y. Acesso em 3 de setembro de 2025. 2025.

[4] *TOP500 Supercomputer List – June 2025*. https://www.top500.org/lists/top500/2025/06/highs. Acesso em 26 de agosto de 2025.

[5] Victor Eijkhout. *Introduction to High-Performance Scientific Computing*. https://freecomputerbooks.com/Introduction-to-High-Performance-Scientific-Computing.html. Acesso em 18 de setembro de 2025. 2015.

[6] IBM. *O que é computação de alto desempenho (HPC)?* https://www.ibm.com/br-pt/think/topics/hpc. Acesso em 18 de setembro de 2025.

[7] Peter S. Pacheco. *An Introduction to Parallel Programming*. 1 **edition**. Disponível em: https://vdoc.pub/download/an-introduction-to-parallel-programming-jjknbg681cg0. Burlington, MA: Morgan Kaufmann, 2011. ISBN: 978-0-12-374260-5.

[8] David B. Kirk **and** Wen-mei W. Hwu. *Programming Massively Parallel Processors: A Hands-on Approach*. 1 **edition**. Disponível em: https://www.cse.iitd.ac.in/~rijurekha/col730_2022/cudabook.pdf. Burlington, MA: Morgan Kaufmann, 2010. ISBN: 978-0-12-381472-2.

[9] Inc. Advanced Micro Devices. *ROCm Platform – Documentation*. Online at https://rocm.docs.amd.com/en/latest/. Acesso em: 22 out. 2025. 2025.

[10] Inc. Advanced Micro Devices. *HIP: Heterogeneous-compute Interface for Portability — Documentation*. Online at https://rocm.docs.amd.com/projects/HIP/en/latest/. Acesso em: 22 out. 2025. 2025.

[11] Alexander Volkov. *C++ Tutorial*. Acessado em: 6 abr. 2026. n.d. URL: https://www.idpoisson.fr/volkov/C++.pdf.

[12] National Institute of Standards and Technology. *Advanced Encryption Standard (AES)*. Federal Information Processing Standards Publication FIPS 197-upd1. Updated May 9, 2023. Gaithersburg, MD: U.S. Department of Commerce, 2001. DOI: 10.6028/NIST.FIPS.197-upd1. URL: https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.197-upd1.pdf.

[13] Ken Schwaber **and** Jeff Sutherland. *O Guia do Scrum: O Guia Definitivo para o Scrum: As Regras do Jogo*. https://scrumguides.org/docs/scrumguide/v2020/2020-Scrum-Guide-PortugueseBR-3.0.pdf. Novembro de 2020. Versão em Português do Brasil. 2020.

[14] Vinícius Henrique Almeida Praxedes. *Paralelizacao do algoritmo K-means em GPUs NVIDIA utilizando a biblioteca Numba*. https://repositorio.ufu.br/bitstream/123456789/43738/1/ParalelizacaoAlgoritmoKmeans.pdf. Trabalho de Conclusão de Curso (Bacharelado em Ciência da Computação), Universidade Federal de Uberlândia, abril de 2024. Acesso em 2 de setembro de 2025.

[15] Francisco H. de Carvalho-Junior **andothers**. "Structured platform-aware programming". **in***Anais do Simpósio de Sistemas Computacionais de Alto Desempenho (SSCAD)*: Acesso em 2 de setembro de 2025. Sociedade Brasileira de Computação (SBC), 2023.

[16] Fernando Moreira. *TCC-AES: Implementação Paralela do AES-128 utilizando HIP/ROCm*. https://github.com/FernandoTMor/TCC-AES. Repositório GitHub. Acesso em: 12 jun. 2026. 2026.