UNIVERSIDADE DE BRASÍLIA
Faculdade do Gama

Programação para Sistemas Paralelos e Distribuídos

Atividade Extraclasse

Arquitetura de CPUs e GPUs

André Emanuel Bispo da Silva - 221007813
Vinicius de Oliveira Santos - 202017263

Brasília, DF
2025

1. Introdução

Com o avanço constante da tecnologia da informação, o processamento de dados em larga escala tornou-se uma necessidade crescente em diversas áreas da ciência, da indústria e do cotidiano. Desde os primeiros computadores, construídos com válvulas e transistores simples, até os sofisticados chips atuais compostos por bilhões de transistores, a evolução da arquitetura de processadores tem sido fundamental para a expansão da capacidade computacional.

Dentre os principais componentes responsáveis por essa evolução estão as Unidades Centrais de Processamento (CPUs) e as Unidades de Processamento Gráfico (GPUs). As CPUs, historicamente projetadas para executar tarefas de propósito geral com alta flexibilidade, passaram por um processo contínuo de miniaturização, aumento da frequência e, mais recentemente, multiplicação de núcleos, permitindo o processamento paralelo em arquiteturas multi-core.

Em paralelo, as GPUs, inicialmente desenvolvidas para acelerar renderizações gráficas em tempo real, evoluíram para unidades altamente paralelas capazes de executar milhares de threads simultaneamente. Com isso, tornaram-se essenciais em aplicações como simulações físicas, modelagem científica, mineração de dados e, especialmente, em Inteligência Artificial (IA) e Aprendizado de Máquina (ML), por meio da computação de propósito geral com GPUs (GPGPU).

Essa transformação também impulsionou o surgimento de modelos de programação especializados, como CUDA e OpenMP, que permitem aos desenvolvedores explorar de forma eficiente o paralelismo massivo oferecido pelas GPUs e a programação multithread nas CPUs. Tais modelos são indispensáveis para o desenvolvimento de aplicações de alto desempenho em arquiteturas modernas.

Neste trabalho, será explorada a evolução das arquiteturas de CPUs e GPUs, destacando suas diferenças fundamentais, características estruturais e os modelos de programação que permitem maximizar seu desempenho. O objetivo é fornecer uma visão abrangente sobre como essas tecnologias transformaram o panorama da computação paralela e distribuída, tornando possível resolver problemas cada vez mais complexos de forma eficiente.

2. Visão geral sobre CPUs

A Unidade Central de Processamento (CPU) é o componente mais importante para que um computador possa funcionar, pois ela executa desde operações aritméticas e lógicas básicas até o controle do sistema e o gerenciamento de entrada/saída. As CPUs são projetadas para lidar com uma ampla variedade de tarefas computacionais, oferecendo flexibilidade e desempenho em aplicações gerais.

Inicialmente, as CPUs possuíam apenas um núcleo de processamento. No entanto, com o avanço das tecnologias e as limitações físicas para o aumento da frequência de operação, surgiram as CPUs multi-core, que permitem a execução simultânea de múltiplas tarefas, proporcionando paralelismo em nível de hardware.

Cada núcleo de uma CPU é otimizado para executar um grande volume de instruções com baixa latência, minimizando os atrasos nas transferências de dados. Uma CPU hoje pode ter de 2 a 96 núcleos, dependendo da sua arquitetura.

Essas instruções são executadas dentro do chamado ciclo de busca-decodificação-execução (fetch-decode-execute). Na etapa de fetch, a CPU busca a próxima instrução da memória RAM com base no valor do contador de programa (PC – Program Counter). Em seguida, na etapa de decode, a CPU interpreta (decodifica) a instrução e determina qual operação será realizada e quais operandos serão usados. Por fim, na etapa de execução, a CPU realiza a operação decodificada.

Com o passar do tempo, a evolução das CPUs seguiu a tendência descrita pela Lei de Moore, aumentando o número de transistores nos chips e, consequentemente, sua capacidade de processamento. Isso permitiu avanços significativos no desempenho computacional ao longo das décadas.

Para tornar a evolução das arquiteturas dos processadores mais evidente, a seguir apresenta-se uma tabela com informações sobre CPUs lançadas ao longo dos anos:

Tabela 1: Evolução das CPUs

CPU	Ano	Tecnologia	Arquitetura / Palavras	Frequência (MHz)	Transistores / Componentes	Núcleos
UNIVAC 1103	1953	Válvulas eletrônicas	36 bits	~0.1 MHz	~18.000 válvulas	1
IBM 7090	1959	Transistores	36 bits	~0.5 MHz	~50.000 transistores	1
Intel 4004	1971	LSI (circuito integrado)	4 bits	0.74 MHz	2.300 transistores	1
Intel 8080	1974	NMOS	8 bits	2 MHz	6.000 transistores	1
Intel 8086	1978	HMOS	16 bits	5-10 MHz	29.000 transistores	1
Motorola 68000	1979	HMOS	16/32 bits	8 MHz	68.000 transistores	1

Intel 80386	1985	CMOS	32 bits	16-33 MHz	275.000 transistores	1
Intel Pentium	1993	BiCMOS	32 bits (x86)	60-300 MHz	3,1 milhões de transistores	1
AMD Athlon XP	2001	CMOS	32 bits	1.3–2.3 GHz	~37,5 milhões de transistores	1
Intel Core 2 Duo	2006	65 nm CMOS	64 bits (x86-64)	~2.0 GHz	~291 milhões de transistores	2
Intel i7 (Nehalem)	2008	45 nm CMOS	64 bits, multi-core	2.66–3.33 GHz	~731 milhões de transistores	4-8
Apple M1	2020	5 nm FinFET	64 bits (ARM)	3.2 GHz	16 bilhões de transistores	8 (4P+4E)
AMD Ryzen 9 7950X	2022	5/6 nm	64 bits (x86-64)	Até 5.7 GHz	~13,1 bilhões de transistores	16
Intel Core Ultra 9 (Meteor Lake)	2023	Intel 4 (7nm equiv.)	64 bits, heterogênea	Até 5.5 GHz	~15 bilhões (estimado)	16 (6P+8E +2LPE)

No início, os computadores eram construídos com válvulas eletrônicas e transistores discretos, ocupando salas inteiras e consumindo enormes quantidades de energia. Com o tempo, esses componentes foram substituídos por circuitos integrados e tecnologias cada vez mais miniaturizadas, passando dos micrômetros para os nanômetros. Hoje, CPUs modernas como a Apple M1 e os processadores da linha AMD Ryzen utilizam processos de fabricação de 5 nanômetros ou menos, com bilhões de transistores integrados em chips.

A arquitetura das CPUs também evoluiu significativamente. Enquanto as primeiras máquinas operam com palavras de 36 bits em arquiteturas específicas e inflexíveis, os anos 1970 e 1980 marcaram a consolidação da arquitetura x86, especialmente com o lançamento do Intel 8086, que estabeleceu a base dos computadores pessoais modernos. Posteriormente, surgiram as arquiteturas RISC, como a ARM, que ganharam força nos dispositivos móveis e, mais recentemente, nos desktops com os chips Apple Silicon.

No que diz respeito à frequência de operação, houve um crescimento acelerado entre as décadas de 1970 e 2000. Processadores como o Intel 8086 operam a 5 MHz, enquanto os primeiros Pentium ultrapassaram os 100 MHz e os processadores dos anos 2000 chegaram a 3 GHz. No entanto, a partir da década de 2010, a frequência estagnou em torno de 3 a 5 GHz devido a limitações térmicas e físicas, deslocando o foco do desenvolvimento para eficiência por ciclo e paralelismo. Assim, o desempenho deixou de depender apenas da velocidade do clock e passou a contar com arquiteturas mais inteligentes e multicores.

A quantidade de transistores é outro indicador marcante da evolução. Se o Intel 4004, lançado em 1971, possuía apenas 2.300 transistores, os chips modernos ultrapassam os 15 bilhões. Esse crescimento exponencial, previsto pela Lei de Moore, permitiu a inclusão de caches maiores, núcleos múltiplos, unidades de processamento gráfico (GPUs) integradas e até aceleradores de inteligência artificial, como NPUs.

Por fim, a quantidade de núcleos ilustra bem a transição do paradigma de desempenho. Durante décadas, as CPUs operavam com apenas um núcleo, o que limitava sua capacidade de executar tarefas simultâneas. A partir dos anos 2000, com o Core 2 Duo da Intel, os processadores começaram a adotar múltiplos núcleos como padrão. Hoje, é comum encontrar CPUs com 8, 16 ou até mais núcleos físicos, muitos deles organizados em arquiteturas híbridas, como nos chips da Apple (com núcleos de performance e eficiência) e da Intel (com núcleos P, E e auxiliares). Isso permite maior escalabilidade, melhor uso de energia e desempenho superior em tarefas paralelas ou multitarefa.

3. Arquitetura de GPU

As GPUs, unidades de processamento gráfico, são processadores dedicados para cálculos de pontos flutuantes, otimização de renderização de imagens e objetos de 3 dimensões e é um facilitador para a implementação de Inteligência Artificial (IA) e Aprendizado de Máquina (ML).

Inicialmente, a GPU era um processador de função fixa, construído sobre um pipeline gráfico que se sobressaia apenas em processamento de gráficos em três dimensões (ZANOTTO; FERREIRA; MATSUMOTO, [2012?]). Mas com o passar do tempo esses processadores foram crescendo em números de pipelines, otimização de desempenho e adição de novas tecnologias como pixel shading, onde cada pixel poderia ser processado por um pequeno programa, SLI e CrossFire, que permite a conexão de várias GPUs para funcionamento em conjunto e adição do barramento PCIe que permite uma comunicação mais eficiente entre a GPU, CPU e outros componentes do computador.

Atualmente as GPUs evoluíram de um processador de pipeline simples, para um processador massivamente paralelo permitindo a execução simultânea de milhares de threads, o que as torna ideais para tarefas que demandam alto poder computacional, como simulações físicas, mineração de criptomoedas e, principalmente, aplicações em Inteligência Artificial (IA) e Aprendizado de Máquina (ML).

Com a introdução de arquiteturas programáveis, como CUDA da NVIDIA e OpenCL, as GPUs passaram a ser utilizadas não apenas para gráficos, mas também para computação geral (GPGPU – General Purpose computing on Graphics Processing Units), viabilizando o treinamento de modelos complexos de redes neurais com grandes volumes de dados em tempo viável.

Além disso, fabricantes como NVIDIA, AMD e mais recentemente a Intel, têm investido em otimizações específicas para IA, como núcleos tensor (Tensor Cores), que aceleram operações matriciais fundamentais para o treinamento e a inferência de redes neurais profundas. Com isso, as GPUs deixaram de ser exclusividade do setor gráfico e se tornaram centrais em centros de dados, supercomputadores e ambientes de pesquisa, consolidando-se como um dos principais vetores do avanço tecnológico nas áreas de IA e ciência de dados.

As arquiteturas de GPUs diferente grandemente de cpus devido à sua especialização histórica para trabalhar com gráficos, que envolve processamento massivamente paralelo (como para operações com vetores e matrizes) e alto throughput para lidar com quantidades grandes de dados. Devido à esse workload diferente GPUs colocam um foco muito maior em unidades de lógica e aritmética e muito menor em unidades de controle (ZANOTTO; FERREIRA; MATSUMOTO, [s.d.]), que são mais essenciais para as tarefas de propósito geral de uma cpu. Dessa forma, sem a complexidade de estruturas de controle permite que a GPU possa ser mais eficiente em trocas de contexto, além de liberar espaço para mais núcleos e outros processadores (BABBAGE, 2024). Além disso cpus enfatizam baixa latência, utilizando estruturas como hierarquias de caches com vários níveis para este fim, enquanto GPUs tem foco maior em *throughput (BASUMALLICK, 2013)*, que seria a capacidade de operações por unidade de tempo, permitindo assim que a placa de vídeo consiga lidar com a alta quantidade de dados por segundo necessário para renderização de vídeo em tempo real por exemplo.

GPUs também diferem em que cpus são compostas de uma quantidade pequena de núcleos de uso geral, otimizados para processamento serial, enquanto GPUs têm grandes quantidades de núcleos que tem propósito mais específico e portanto tem menor performance para processamento serial, mas permitindo uma eficiência muito maior em tarefas paralelizáveis, permitindo processamento rápido de instruções em paralelo *(BASUMALLICK, 2013)*.

Figura 1 - Esquemático de um processador da arquitetura Nvidia Hopper H100

Fonte: Babbage, 2024

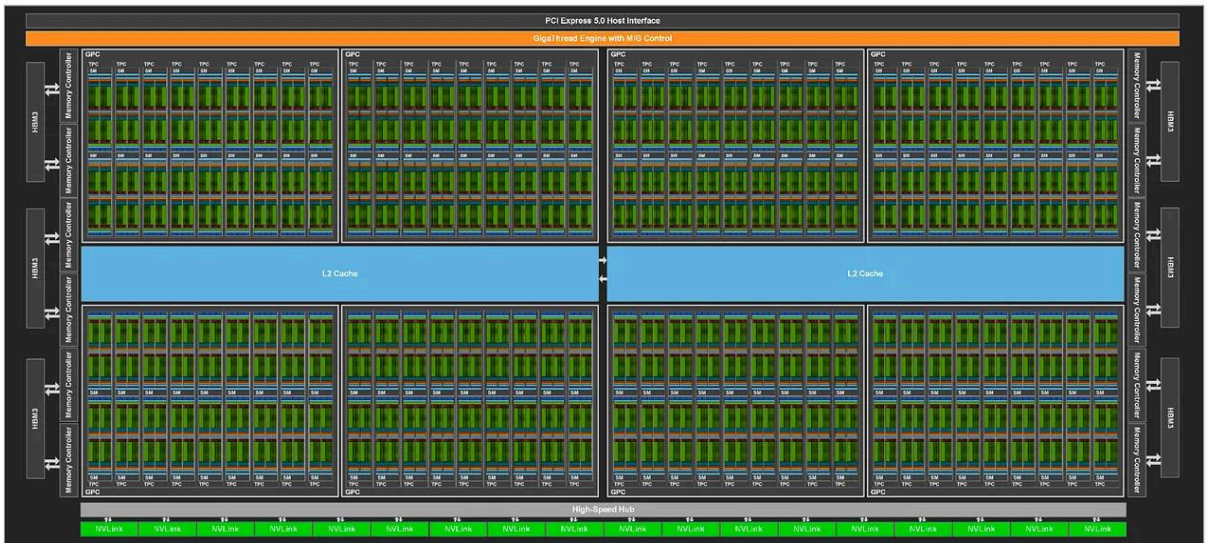
Como visto na imagem acima um “núcleo” de uma GPU contém várias unidades de execução (processamento de inteiros, números de ponto flutuante, carregamento em memória e funções transcendentais) que são executadas em paralelo com uma única instrução similar à execução de instruções SIMD de CPUs (BABBAGE, 2024), é visível a diferença de número de unidades de execução e tamanho de cache quando comparado à um núcleo de cpu, por exemplo. GPUs utilizam uma hierarquia de núcleos, em que Núcleos são agrupados em *Streaming MultiProcessors*, que são agrupados em *Processor Clusters*. Essas hierarquias permitem uma grande densidade de unidades de computação.

Figura 2 - Agrupamento de núcleos em um Streaming Multiprocessor

Fonte: Babbage, 2024

Além disso, GPUs modernas também apresentam hardware específico para aceleração de tarefas de machine learning para aceleração de cálculo tensorial para, como visto no “Núcleo Tensorial” da figura 1.

Figura 3 - Agrupamento de SMs em uma GPU



Fonte: Babbage, 2024

3.1. Evolução de arquiteturas

3.1.1. G80

Foi a primeira geração de placas Nvidia a ter suporte para a plataforma CUDA, com suporte a linguagem C e 128 CUDA cores e 8 SMs (ZANOTTO; FERREIRA; MATSUMOTO, 2025).

3.1.2. Fermi

De acordo com (ZANOTTO; FERREIRA; MATSUMOTO, 2025) a arquitetura fermi trouxe suporte a instruções novas para programação CUDA em C++, como alocação dinâmica de objetos e tratamento de exceções, permitindo maior flexibilidade na programação em GPU.

A microarquitetura Fermi é composta de SMs com 32 Cuda cores, com capacidade de 16 operações de 64 bits por pulso de clock. Além disso, cada SM contém 16 unidades ld/str para cálculo de endereços e 4 Special Function Units para cálculo de funções transcendentais (ZANOTTO; FERREIRA; MATSUMOTO, 2025).

3.1.3. Kepler

Criou um novo modelo de SM com 192 núcleos CUDA, tirando proveito de tecnologias como transistores de 28 nanômetros para reduzir consumo de energia e aumentar a frequência de clock. Cada SMX, como foram chamados, possui 64

unidades de 64 bits, 32 SFUs e 32 unidades ld/str (ZANOTTO; FERREIRA; MATSUMOTO, 2025).

Além disso a arquitetura tem capacidade de paralelismo dinâmico, para reutilização de threads, sem necessidade do overhead de comunicação com a CPU (que é uma grande parte da perda de performance em aplicações CUDA) para reuso de threads (ZANOTTO; FERREIRA; MATSUMOTO, 2025). Também o Hyper-Q das arquiteturas kepler permitem que múltiplos núcleos de uma CPU utilizem o mesmo dispositivo, com até 32 conexões simultâneas, diminuindo tempo ocioso (ZANOTTO; FERREIRA; MATSUMOTO, 2025).

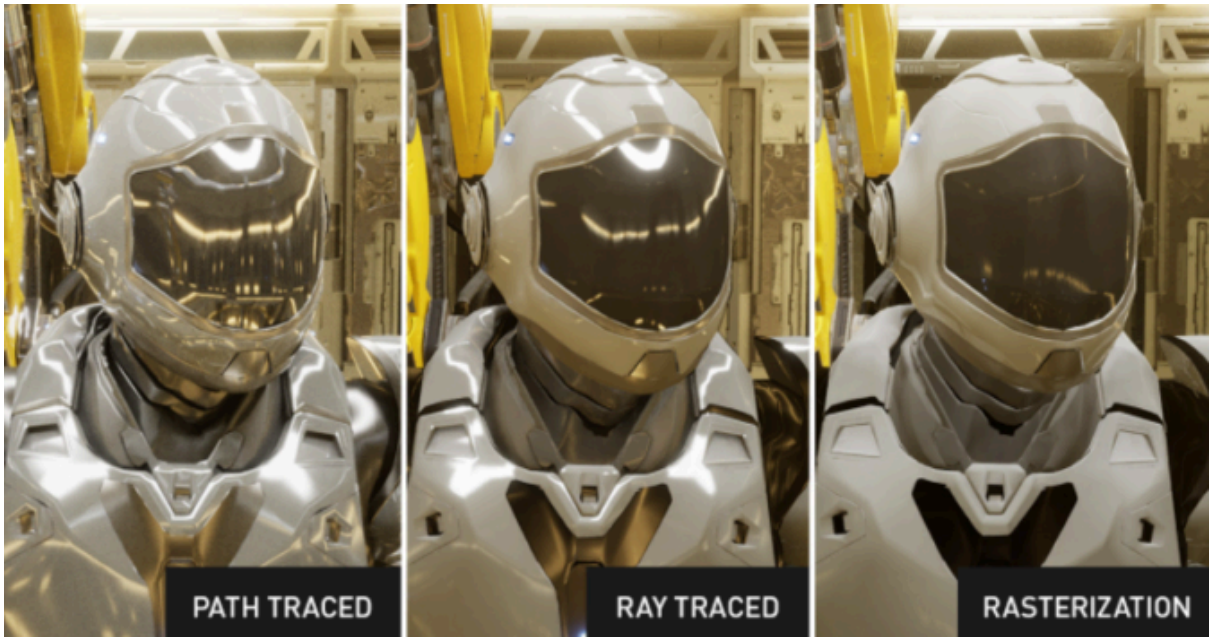
3.1.4. Volta

A arquitetura Volta introduziu os Tensor Cores, permitindo a aceleração de tarefas de treino baseadas em deep learning (NEHALMR, 2024).

3.1.5. Turing/Ampere

Turing foi a arquitetura que introduziu os Tensor Cores de segunda geração, processadores dedicados a operações matriciais para aceleração de modelos de aprendizagem de máquina. Além disso, a arquitetura Turing introduziu os RT Cores, núcleos especializados para ray-tracing em tempo real (NEHALMR, 2024).

Figura 4 - Demonstração do ray tracing em tempo real nas placas RTX

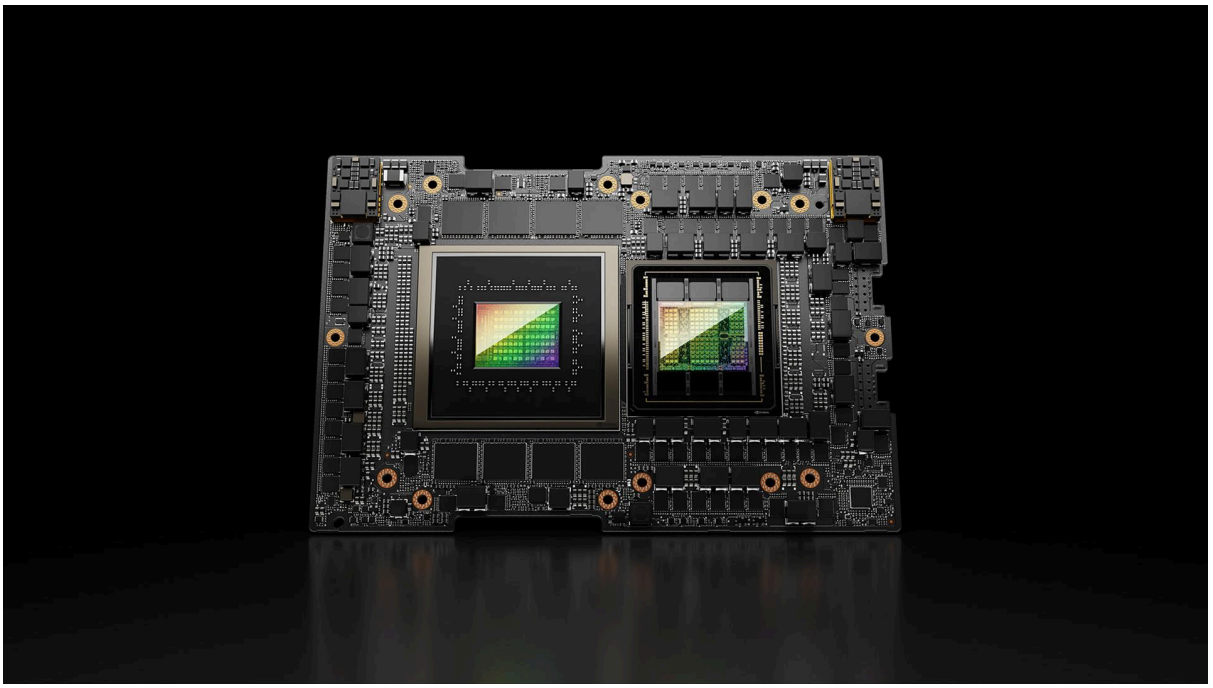


Fonte: Nvidia Corporation, 2022

3.1.6. Grace Hopper

A arquitetura Grace Hopper são diferentes dos designs anteriores, pois são chips desenvolvidos para datacenters, com o superchip Grace Hopper que na verdade é um conjunto CPU+GPU, apresentando velocidades maiores do que são capazes pelo barramento PCIe com velocidades de até 900 GB/s (NVIDIA CORPORATION, [s.d.]), permitindo através dessas melhorias grande capacidade de processamento paralelo de alto throughput, estas características permitem alta performance para tarefas envolvendo Big Data como machine learning.

Figura 5 - Superchip Nvidia Grace Hopper



Fonte: Nvidia Corporation, 2024

4. Modelo de programação para GPUs

Com a crescente evolução arquitetural das GPUs, tornou-se possível explorar de forma mais eficiente o paralelismo de dados, essencial para acelerar cálculos massivos em diversas áreas da computação. A popularização do uso dessas unidades gráficas fora do contexto original de renderização 3D abriu caminho para novos paradigmas de programação, nos quais tarefas intensivas em processamento podem ser executadas de maneira altamente paralela. Nesse cenário, surgem modelos como CUDA e OpenMP, desenvolvidos com o propósito de facilitar a programação voltada para a execução em múltiplos núcleos e milhares de threads simultâneas.

O modelo de programação CUDA, criado pela NVIDIA, é especialmente voltado para exploração das capacidades computacionais de suas GPUs, permitindo ao programador controlar diretamente a execução em blocos e threads. Por outro lado, o OpenMP, originalmente desenvolvido para ambientes de CPU, também passou a oferecer suporte a execução paralela em GPUs por meio de diretrizes específicas, como aquelas voltadas ao modelo SIMD. A compreensão desses modelos é fundamental para tirar proveito total da arquitetura moderna das GPUs e maximizar o desempenho das aplicações que requerem alto poder de processamento.

4.1 CUDA

CUDA (Compute Unified Device Architecture) é uma plataforma de computação paralela criada pela NVIDIA para tirar proveito das capacidades de computação paralela utilizando GPUS, a plataforma tem apoio para linguagens diferentes como fortran, java e python; programação com diretivas e APIs diferentes como DirectCompute e OpenACC (NVIDIA CORPORATION, 2025).

A interface c++ consiste em um conjunto de extensões para a linguagem c++ além de uma biblioteca em tempo de execução responsável por fazer a interação e controle entre a CPU e a(s) GPU(s), tal biblioteca faz uso de um driver de baixo nível para execução de instruções de memória e controle por exemplo. Este driver também pode ser utilizado diretamente, mas as extensões de linguagem fornecem uma API para código mais conciso.

O código cuda é uma mistura de código de CPU e GPU. Na compilação as diretivas são substituídas pela chamadas de função ao Runtime CUDA e o código do kernel é compilado para a GPU, tendo como opções de saída código C++ modificado que pode ser compilado separadamente ou código de objeto. O código de GPU pode ser carregado separadamente usando a API do CUDA, ignorando o código do host.

O modelo de programação CUDA é baseado em “kernels” que são as funções que serão rodadas em paralelo na GPU, elas são declaradas usando o declarador `__global__`, e são chamadas usando a sintaxe `<<<N, M>>>`, para executar a função utilizando M threads por bloco em N blocos. Cada bloco de threads roda em um único SM da GPU, e cada SM tem suporte a 1024 threads ativas.

Figura 6 - Soma vetorial em CUDA com 256 threads

```cpp
#include <iostream>
#include <math.h>

__global__ void add(int n, float *x, float *y) {
  int index = threadIdx.x;
  int stride = blockDim.x;
  for (int i = index; i < n; i += stride)
      y[i] = x[i] + y[i];
}

int main(void) {
  int N = 1 << 20;
  float *x, *y;

  cudaMallocManaged(&x, N * sizeof(float));
  cudaMallocManaged(&y, N * sizeof(float));

  for (int i = 0; i < N; i++) {
    x[i] = 1.0f;
    y[i] = 2.0f;
  }

  add<<<1, 256>>>(N, x, y);

  cudaDeviceSynchronize();

  float maxError = 0.0f;
  for (int i = 0; i < N; i++) {
    maxError = fmax(maxError, fabs(y[i] - 3.0f));
  }
  std::cout << "Max error: " << maxError << std::endl;

  cudaFree(x);
  cudaFree(y);
}
```

Fonte: Harris, 2025

No exemplo acima é implementada uma operação de soma vetorial paralela em 256 threads e 1 bloco. Para que a operação possa ser dividida entre as threads são utilizadas as variáveis threadIdx e blockDim, que indicam o índice da thread em seu bloco e o número de threads em um bloco respectivamente.

Figura 7 - Soma vetorial em CUDA com 256 threads por bloco

```cpp
#include <iostream>
#include <math.h>

__global__ void add(int n, float *x, float *y) {
  int index = blockIdx.x * blockDim.x + threadIdx.x;
  int stride = blockDim.x * gridDim.x;
  for (int i = index; i < n; i += stride)
    y[i] = x[i] + y[i];
}

int main(void) {
  int N = 1 << 20;
  float *x, *y;

  cudaMallocManaged(&x, N * sizeof(float));
  cudaMallocManaged(&y, N * sizeof(float));

  for (int i = 0; i < N; i++) {
    x[i] = 1.0f;
    y[i] = 2.0f;
  }

  int blockSize = 256;
  int numBlocks = (N + blockSize - 1) / blockSize;
  add<<<numBlocks, blockSize>>>(N, x, y);

  cudaDeviceSynchronize();

  float maxError = 0.0f;
  for (int i = 0; i < N; i++) {
    maxError = fmax(maxError, fabs(y[i] - 3.0f));
  }
  std::cout << "Max error: " << maxError << std::endl;

  cudaFree(x);
  cudaFree(y);
}
```

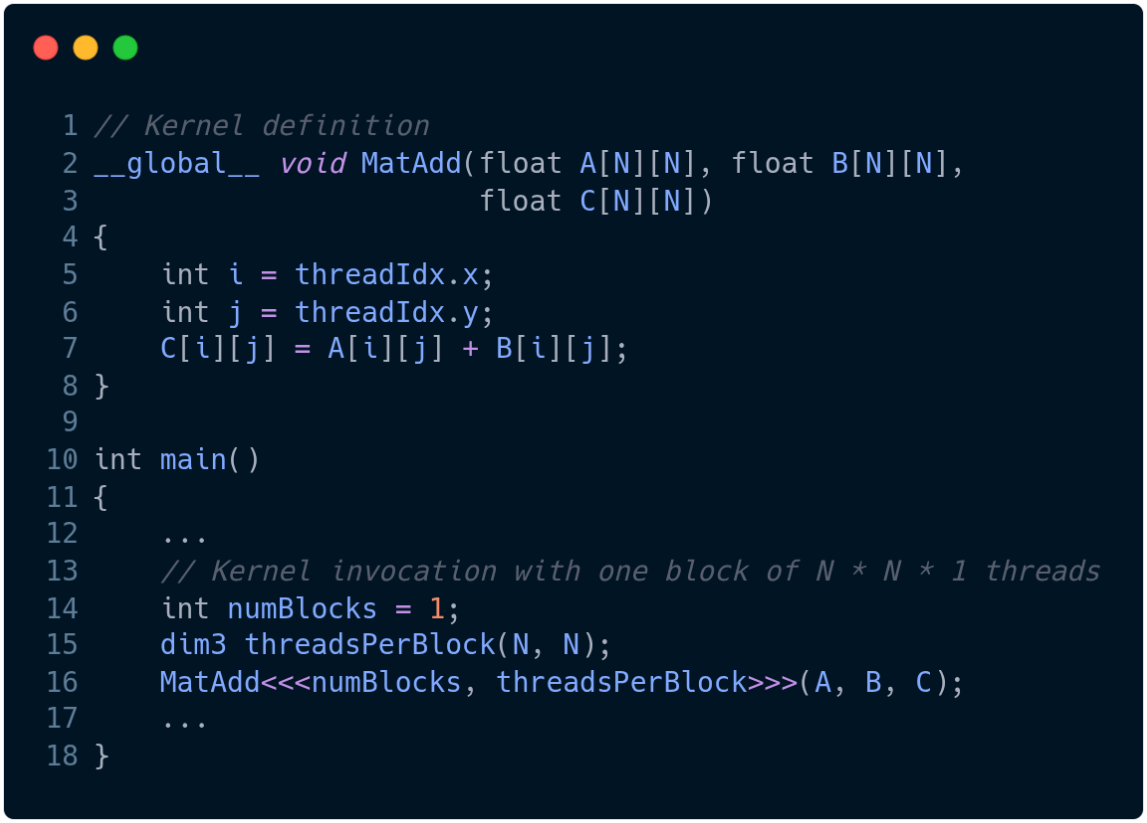
Fonte: Harris, 2025

Este exemplo, por outro lado, faz uso de blocos, utilizando uma expressão para calcular quantos blocos serão necessários para processar todos os elementos. Para a divisão de tarefas, o código utiliza a variável gridDim que representa o tamanho do “grid” que é o agrupamento de todos os blocos sendo utilizados.

Além dos exemplos da API demonstrados anteriormente, as variáveis de Identificação fornecidas pela plataforma são na verdade vetores de 3 componentes

(threadId.x, threadId.y, threadId.z, por exemplo), a fim de facilitar operações com dados de uma, duas e três dimensões com naturalidade (NVIDIA CORPORATION, 2025).

Figura 8 - Soma matricial em cuda com índices bidimensionais



```
// Kernel definition
__global__ void MatAdd(float A[N][N], float B[N][N],
                       float C[N][N])
{
    int i = threadIdx.x;
    int j = threadIdx.y;
    C[i][j] = A[i][j] + B[i][j];
}

int main()
{
    ...
    // Kernel invocation with one block of N * N * 1 threads
    int numBlocks = 1;
    dim3 threadsPerBlock(N, N);
    MatAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
    ...
}
```

Fonte: Nvidia Corporation, 2025

Figura 9 - Soma matricial em cuda com índices tridimensionais

```cuda
// Kernel definition
__global__ void MatAdd(float A[N][N][N], float B[N][N][N], float C[N][N][N])
{
    int i = threadIdx.x;
    int j = threadIdx.y;
    int k = threadIdx.z;

    C[i][j][k] = A[i][j][k] + B[i][j][k];
}

int main()
{
    ...
    // Kernel invocation with one block of N×N×N threads
    dim3 threadsPerBlock(N, N, N);
    dim3 numBlocks(1, 1, 1);
    MatAdd<<<numBlocks, threadsPerBlock>>>(d_A, d_B, d_C);
    ...
}
```

Fonte: André Silva, 2025

Como visto no exemplo acima é possível endereçar diretamente os elementos das matrizes utilizando os índices bidimensionais, sem precisar calcular offsets manualmente, por exemplo para uma thread em um bloco bidimensional de tamanho (Dx, Dy) seu índice seria: $(x, y) = (x + y * Dx)$ e para um bloco tridimensional de tamanho (Dx, Dy, Dz) seu índice seria: $(x, y, z) = (x + y * Dx + z * Dx * Dy)$.

Link para repositório com os códigos:

- https://github.com/ViniciussdeOliveira/PSPD-AE-CPUeGPU

4.2 OpenMP

O OpenMP é uma API para programação paralela de arquiteturas multi-processador/multicore definida inicialmente para ser usada em programas C/C++ ou Fortran sobre plataformas Unix/Linux ou Windows. O OpenMP é um modelo de programação em memória compartilhada de programação portável e escalável que proporciona aos programadores uma interface simples e flexível para o desenvolvimento de aplicações paralelas.

A API OpenMP é baseada no modelo de execução fork-join, no qual uma thread principal (mestre) inicia o programa e, em determinados trechos, cria múltiplas threads auxiliares para executar tarefas em paralelo. O programador define explicitamente quais partes do código devem ser paralelizadas. Assim, o código é executado de forma sequencial pela thread mestre até atingir uma seção paralela.

Essas seções paralelas começam com uma diretiva específica do OpenMP, que indica quando as threads auxiliares devem ser criadas (fork). No caso das linguagens C/C++, essa diretiva é precedida por #pragma omp, e em FORTRAN por !$omp, podendo incluir também atributos adicionais. O bloco de código que se segue será executado simultaneamente pelas threads até o final da seção paralela, que pode ser delimitado de maneira explícita, como com o símbolo } em C/C++, ou !%omp end em FORTRAN, ou ainda de forma implícita, como no caso de um laço for ou do, onde o encerramento do laço também marca o fim da execução paralela. Ao término da área paralela, ocorre o encerramento das threads auxiliares, sua sincronização e a retomada da execução pela thread mestre (join).

A partir da versão 4.0, o OpenMP passou a oferecer suporte a instruções SIMD (Single Instruction, Multiple Data), permitindo que o programador especifique que determinados trechos de código (especialmente laços) sejam vetorizados. Isso significa que uma única instrução pode ser aplicada simultaneamente a múltiplos dados, aproveitando as capacidades de vetorização dos processadores modernos (como SSE, AVX em x86).

Com essa adição, diretivas como #pragma omp simd passaram a ser utilizadas para indicar ao compilador que um determinado laço deve ser vetorizado, ou seja, que ele pode ser transformado para executar múltiplas iterações simultaneamente utilizando instruções SIMD. Isso aumenta significativamente o desempenho de certos trechos computacionalmente intensivos, principalmente os que envolvem operações aritméticas sobre grandes vetores ou matrizes.

Além da diretiva simd, o OpenMP 4.0 também introduziu outras funcionalidades importantes, como:

- Mapeamento para dispositivos (offloading): com a diretiva target, passou a ser possível executar trechos de código em dispositivos aceleradores, como GPUs. Isso estende o modelo de memória compartilhada para arquiteturas heterogêneas.
- Melhor controle de tarefas: melhorias nas diretivas task, taskgroup e dependências entre tarefas (depend), permitindo programação assíncrona mais sofisticada.

- Suporte a declare simd: permite ao programador declarar versões vetorizadas de funções, que podem então ser chamadas dentro de loops simd.

Essas funcionalidades tornam o OpenMP uma ferramenta poderosa não apenas para paralelismo tradicional em CPUs com múltiplos núcleos, mas também para programação em arquiteturas modernas com aceleração por hardware, como as GPUs. A seguir serão apresentados alguns exemplos de uso das diretivas do OpenMP com processamento SIMD.

Combinação de parallel for com simd (CPU com múltiplos núcleos + vetorização)

```c
#include <omp.h>
#include <stdlib.h>
#include <stdio.h>

// Função que multiplica cada elemento de um vetor por um escalar
void multiplicar_escalar(float *vetor, float escalar, int n) {
    // Diretiva OpenMP que combina paralelismo de múltiplas threads (parallel for)
    // com vetorização SIMD (simd) para otimizar a execução da iteração do laço
    #pragma omp parallel for simd
    for (int i = 0; i < n; i++) {
        // Multiplica o elemento i do vetor pelo escalar fornecido
        vetor[i] *= escalar;
    }
}


int main() {
    int n = 1000;
    float escalar = 2.5;
    float *vetor = (float *)malloc(n * sizeof(float));

    // Inicializa o vetor
    for (int i = 0; i < n; i++) {
        vetor[i] = i * 1.0f;
    }

    // Multiplica com OpenMP + SIMD
    multiplicar_escalar(vetor, escalar, n);

    // Mostra os 5 primeiros resultados
    for (int i = 0; i < 5; i++) {
        printf("vetor[%d] = %.2f\n", i, vetor[i]);
    }

    free(vetor);
    return 0;
}
```

Nesse código, a função multiplicar_escalar possui a instrução #pragma omp parallel for simd, que indica ao compilador que o laço for pode ser dividido entre múltiplas threads para execução paralela (parallel for) e, dentro de cada thread, o laço pode ser vetorizado usando instruções SIMD (Single Instruction Multiple Data).

Dessa forma, podemos aproveitar tanto os múltiplos núcleos da CPU quanto às instruções de vetorização para acelerar o processamento.

Offloading para GPU com target (a partir do OpenMP 4.0)

```c
#include <omp.h>
#include <stdlib.h>
#include <stdio.h>

// Função que realiza a soma de dois vetores utilizando GPU com OpenMP
void soma_gpu(float *a, float *b, float *resultado, int n) {

    // Diretiva OpenMP para offload do bloco de código para o dispositivo (ex: GPU)
    // 'map(to: a[0:n], b[0:n])' indica que os vetores 'a' e 'b' devem ser copiados da CPU para a GPU
    // 'map(from: resultado[0:n])' indica que o vetor 'resultado' será copiado da GPU de volta para a CPU após a execução
    #pragma omp target map(to: a[0:n], b[0:n]) map(from: resultado[0:n])

    // Diretiva OpenMP para paralelizar o loop 'for' na GPU
    #pragma omp parallel for
    for (int i = 0; i < n; i++) {
        // Soma elemento a elemento dos vetores 'a' e 'b', armazenando o resultado no vetor 'resultado'
        resultado[i] = a[i] + b[i];
    }
}

int main() {
    int n = 1000000;
    float *a = (float *)malloc(n * sizeof(float));
    float *b = (float *)malloc(n * sizeof(float));
    float *resultado = (float *)malloc(n * sizeof(float));

    // Inicializa os vetores
    for (int i = 0; i < n; i++) {
        a[i] = i * 1.0f;
        b[i] = i * 2.0f;
    }

    // Soma vetores na GPU
    soma_gpu(a, b, resultado, n);

    // Mostra os 5 primeiros resultados
    for (int i = 0; i < 5; i++) {
        printf("resultado[%d] = %.2f\n", i, resultado[i]);
    }

    free(a);
    free(b);
    free(resultado);
    return 0;
}
```

Essa função paraleliza a soma de dois vetores (a e b) utilizando offload para GPU via OpenMP. As diretivas #pragma omp target e #pragma omp parallel for fazem com que o cálculo seja executado paralelamente na GPU, melhorando o desempenho em comparação com uma execução sequencial na CPU, especialmente para grandes vetores.

Vetorização com OpenMP SIMD e collapse(n)

```c
#include <stdio.h>
#include <omp.h>

#define N 4
#define M 4

int main() {
    float A[N][M], B[N][M], C[N][M];

    // Inicializa as matrizes A e B
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < M; j++) {
            A[i][j] = i + j;
            B[i][j] = i - j;
        }
    }

    // Multiplicação elemento a elemento usando OpenMP SIMD com collapse
    #pragma omp simd collapse(2)
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < M; j++) {
            C[i][j] = A[i][j] * B[i][j];
        }
    }

    // Impressão do resultado
    printf("Matriz C (resultado da multiplicação elemento a elemento):\n");
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < M; j++) {
            printf("%6.2f ", C[i][j]);
        }
        printf("\n");
    }

    return 0;
}
```

O código realiza a multiplicação elemento a elemento (também chamada de multiplicação Hadamard) de duas matrizes 2D, armazenando o resultado em uma terceira matriz. Para otimizar a execução, utiliza-se a diretiva #pragma omp simd com a cláusula collapse(2) do OpenMP, permitindo ao compilador aplicar instruções SIMD (Single Instruction, Multiple Data) para acelerar os cálculos.

A cláusula collapse(n) do OpenMP é usada para combinar múltiplos loops aninhados em um único loop, permitindo que o compilador ou o tempo de execução trate todas as iterações como uma sequência linear. Isso é especialmente útil para paralelização e vetorização, pois aumenta a granularidade do paralelismo disponível, facilitando o balanceamento de carga entre as threads ou a aplicação de instruções SIMD. Ao colapsar os loops, o OpenMP transforma uma estrutura aninhada de n níveis em um único espaço de iteração, o que pode melhorar significativamente o desempenho em arquiteturas modernas com suporte a paralelismo e vetorização.

Vetorização com OpenMP SIMD e reduction

```c
#include <stdio.h>
#include <omp.h>

#define TAM 1000

int main() {
    float vetor[TAM];
    float soma = 0.0f;

    // Inicializa o vetor com valores simples
    for (int i = 0; i < TAM; i++) {
        vetor[i] = 1.0f; // Cada elemento vale 1.0
    }

    // Paraleliza a soma com OpenMP SIMD e cláusula reduction
    #pragma omp simd reduction(+:soma)
    for (int i = 0; i < TAM; i++) {
        soma += vetor[i];
    }

    printf("Soma dos elementos: %.2f\n", soma); // Esperado: 1000.00

    return 0;
}
```

Este programa em C demonstra o uso da diretiva #pragma omp simd do OpenMP para aplicar paralelismo do tipo SIMD (Single Instruction, Multiple Data) em um loop que soma os elementos de um vetor. O vetor é inicializado com 1000 elementos, todos com valor 1.0, e a soma é realizada dentro de um laço for. Ao utilizar #pragma omp simd, o compilador é instruído a vetorizar o loop, permitindo que múltiplas iterações sejam executadas simultaneamente por meio de registradores vetoriais da CPU. Essa abordagem melhora o desempenho em processadores modernos que suportam instruções SIMD, reduzindo o tempo necessário para percorrer grandes conjuntos de dados.

A cláusula reduction(+:soma) é essencial para garantir que a operação de soma seja feita corretamente no contexto do paralelismo. Em um loop vetorizado, várias instâncias da variável soma são criadas localmente para cada vetor de execução, evitando condições de corrida (race conditions). Ao final da execução paralela, essas variáveis locais são combinadas (reduzidas) em uma única variável global usando o operador de soma +. Esse processo é chamado de redução e é crucial para manter a correção do resultado ao aplicar paralelismo em operações acumulativas, como somas ou produtos.

Link para repositório com os códigos:

- https://github.com/ViniciussdeOliveira/PSPD-AE-CPUeGPU

5. Conclusão

Em suma, a evolução das arquiteturas de CPUs e GPUs representa um dos pilares fundamentais do avanço tecnológico nas últimas décadas, refletindo diretamente na forma como desenvolvemos aplicações, processamos informações e enfrentamos desafios computacionais cada vez mais complexos. Enquanto as CPUs evoluíram para oferecer maior desempenho em tarefas gerais por meio de múltiplos núcleos e maior eficiência energética, as GPUs se destacaram no processamento massivamente paralelo, expandindo sua atuação para além dos gráficos e tornando-se peças-chave em áreas como inteligência artificial, simulações científicas e computação de alto desempenho. Com o suporte de modelos de programação como CUDA e OpenMP, a capacidade de explorar todo o potencial dessas arquiteturas tornou-se mais acessível, consolidando um cenário no qual a eficiência e a escalabilidade são essenciais para o futuro da computação.

Experiência - André

Com essa pesquisa tive a oportunidade de estudar sobre modelos de programação paralela tanto para CPUs e GPUs em especial a parte sobre CUDA, achei muito importante a parte de arquitetura de GPUs, pois apenas é possível obter os ganhos de performance necessários pela paralelização através do conhecimento de como o sistema computacional realmente funciona.

Experiência - Vinícius

A atividade me forneceu bastante conhecimento sobre as CPUs, GPUs e os modelos de programação para as unidades de processamento gráfico. Percebi que a programação desses componentes requer muita atenção aos detalhes, pois uma declaração na posição errada pode trazer resultados bem diferentes do esperado. Participei da pesquisa sobre CPUs, GPUs, OpenMP e estudei sobre o CUDA. Considero esse trabalho de pesquisa interessante para adquirir mais controle e confiança sobre o programa que estamos desenvolvendo para alcançar o maior desempenho possível.

6. Referências bibliográficas

BABBAGE. Demystifying GPU Compute Architectures. Disponível em: <https://thechipletter.substack.com/p/demystifying-gpu-compute-architectures>. Acesso em: 21 maio. 2025.

BASUMALLICK, C. CPU vs. GPU: 11 Key Comparisons. Disponível em: <https://www.spiceworks.com/tech/hardware/articles/cpu-vs-gpu/>. Acesso em: 21 maio. 2025.

HARRIS, M. An Even Easier Introduction to CUDA. Disponível em: <https://developer.nvidia.com/blog/even-easier-introduction-cuda/>. Acesso em: 23 maio. 2025.

NEHALMR. “The Evolution of NVIDIA Chips: A Journey from Graphics to AI and Beyond” by nehalmr. Disponível em: <https://nehalmr.medium.com/the-evolution-of-nvidia-chips-a-journey-from-graphics-to-ai-and-beyond-by-nehalmr-55de44fadb92>. Acesso em: 23 maio. 2025.

NVIDIA CORPORATION. CUDA C++ Programming Guide. Disponível em: <https://docs.nvidia.com/cuda/cuda-c-programming-guide/>. Acesso em: 22 maio. 2025.

NVIDIA CORPORATION. NVIDIA Grace Hopper Superchip. Disponível em: <https://www.nvidia.com/en-us/data-center/grace-hopper-superchip/>. Acesso em: 23 maio. 2025.

SHAPIRO, A. NVIDIA Grace Hopper Ignites New Era of AI Supercomputing. Disponível em: <https://nvidianews.nvidia.com/news/nvidia-grace-hopper-ignites-new-era-of-ai-supercomputing>. Acesso em: 23 maio. 2025.

ZANOTTO, L.; FERREIRA, A.; MATSUMOTO, M. Arquitetura E Programação De GPU Nvidia. [s.l: s.n.]. Disponível em: <https://ic.unicamp.br/~ducatte/mo401/1s2012/T2/G02-001963-023169-085937-t2.pdf>. Acesso em: 22 maio. 2025.

MADAN, A. CPU vs. GPU: Which to use and when? Disponível em: <https://www.analyticsvidhya.com/blog/2023/03/cpu-vs-gpu/>. Acesso em: 23 maio 2025.

A história das placas de vídeo: Dos primórdios à realidade virtual. Disponível em: <https://epraja.com.br/a-fascinante-jornada-das-placas-de-video-dos-primordios-a-realidade-virtual/>. Acesso em: 23 maio 2025.

FLINDERS, M.; SUSNJARA, S.; SMALLEY, I. O que é uma GPU? Ibm.com, 15 Apr. 2025. Disponível em: <https://www.ibm.com/br-pt/think/topics/gpu>. Acesso em: 23 maio 2025.

Disponível em: <https://mcclanahoochie.com/blog/wp-content/uploads/2011/03/gpu-hist-paper.pdf>. Acesso em: 23 maio 2025.

Departamento de Ciência de Computadores Faculdade de Ciências. Disponível em: <https://www.dcc.fc.up.pt/~ricroc/aulas/1516/cp/apontamentos/slides_openmp.pdf>. Acesso em: 23 maio 2025.

Departamento de Ciência de Computadores Faculdade de Ciências. Disponível em: <https://www.dcc.fc.up.pt/~ricroc/aulas/1516/cp/apontamentos/slides_openmp.pdf>. Acesso em: 23 maio 2025.

Parallel programming: Multithreading (OpenMP). Disponível em: <https://wvuhpc-github-io.translate.goog/2018-Lesson_4/03-openmp/index.html?_x_tr_sl=en&_x_tr_tl=pt&_x_tr_hl=pt&_x_tr_pto=tc>. Acesso em: 23 maio 2025.

Disponível em: <https://www.google.com/url?sa=t&source=web&rct=j&opi=89978449&url=https://books-sol.sbc.org.br/index.php/sbc/catalog/download/119/526/805-1&ved=2ahUKEwjsyp6og7iNAxU9CLkGHY_HH7sQFnoECDgQAQ&usg=AOvVaw1rqpZu8rBNX5dFpBitFtuk>. Acesso em: 23 maio 2025.

TYLERMSFT. Extensão SIMD. Disponível em: <https://learn.microsoft.com/pt-br/cpp/parallel/openmp/openmp-simd?view=msvc-170>. Acesso em: 23 maio 2025.

POWELL, P. A história da unidade central de processamento (CPU). Ibm.com, 9 Oct. 2024. Disponível em: <https://www.ibm.com/br-pt/think/topics/central-processing-unit-history>. Acesso em: 23 may. 2025