Estudo extraclasse – Arquitetura de CPUs e GPUs

PSPD - Programação para Sistemas Paralelos e Distribuídos

Prof. Fernando William Cruz

Murilo Perazzo Barbosa Souto

Introdução

As arquiteturas de processadores evoluíram consideravelmente nas últimas décadas, acompanhando o crescimento exponencial da demanda por poder computacional em diversas áreas. Dentre as principais classes de processadores, destacam-se as CPUs (Unidades Centrais de Processamento) e as GPUs (Unidades de Processamento Gráfico), que, apesar de compartilharem o objetivo comum de executar tarefas computacionais, apresentam arquiteturas e propósitos distintos.

Breve histórico das arquiteturas de CPUs e GPUs

A CPU, criada como o "cérebro" dos primeiros computadores, teve sua arquitetura inicialmente focada em executar uma ampla variedade de instruções sequenciais para processamento geral. Desde os primeiros modelos, como o Intel 4004 lançado em 1971, até os modernos processadores multi-core, como os Intel Core e AMD Ryzen, as CPUs evoluíram para incorporar múltiplos núcleos, caches maiores e tecnologias avançadas de previsão de comandos e paralelismo. Esse avanço foi influenciado pela Lei de Moore, que previa a duplicação do número de transistores a cada dois anos, permitindo maior capacidade de processamento.

As GPUs, por sua vez, surgiram a partir da necessidade de acelerar o processamento gráfico para jogos e aplicações visuais. Inicialmente, concentradas em operações específicas de renderização, as GPUs evoluíram para arquiteturas altamente paralelas, capazes de lidar com milhares de threads simultaneamente. Desde a primeira geração de GPUs dedicadas, como as placas produzidas pela NVIDIA e ATI no final dos anos 1990, até as GPUs atuais com suporte a programação paralela por meio de frameworks como CUDA e OpenCL, houve um crescimento exponencial em sua capacidade de processamento paralelo.

Principais aplicações das CPUs e GPUs

As CPUs continuam sendo o componente central para processar tarefas gerais do sistema computacional, incluindo sistemas operacionais, execução de programas diversos, aplicações empresariais e tarefas que requerem baixa latência e alta flexibilidade. Sua arquitetura é ideal para operações sequenciais complexas e controle de fluxo.

As GPUs, graças ao seu design focado em paralelismo massivo, tornaram-se essenciais não apenas na renderização gráfica para jogos, animações e interfaces visuais, mas também em áreas emergentes como inteligência artificial, aprendizado profundo (deep learning), simulações científicas e processamento de grandes volumes de dados. Este uso ampliado das GPUs transformou-as em plataformas computacionais versáteis, capazes de acelerar algoritmos que podem ser paralelizados.

Características gerais e importância da evolução tecnológica

A principal diferença entre CPUs e GPUs está no seu modelo arquitetural: as CPUs possuem poucos núcleos complexos, otimizados para baixa latência e alta flexibilidade, enquanto as GPUs contam com milhares de núcleos simples, que trabalham de maneira colaborativa para executar tarefas altamente paralelas. Essa distinção resulta em diferentes perfis de desempenho e aplicações.

A evolução tecnológica de ambas as arquiteturas tem sido crucial para a transformação da computação moderna. Com o aumento da eficiência energética, maior paralelismo e arquiteturas específicas para tarefas dedicadas, tanto CPUs quanto GPUs possibilitam que novas aplicações, que vão desde a inteligência artificial até a realidade virtual, se tornem viáveis e acessíveis.

Visão geral sobre CPUs

A CPU (Unidade Central de Processamento) desempenha um papel fundamental em qualquer sistema computacional, sendo responsável por interpretar e executar instruções contidas em programas de computador. Sua arquitetura básica consiste em três componentes principais: a unidade de controle, que coordena as operações do processador; a unidade lógica e aritmética (ULA), que realiza operações matemáticas e lógicas; e os registradores, que armazenam temporariamente dados e instruções durante o processamento. Por sua versatilidade e capacidade de executar uma vasta gama de tarefas, as CPUs são conhecidas como processadores de uso geral.

A partir dos anos 2000, passou a ser economicamente e tecnologicamente mais vantajoso investir na construção de processadores multi-core do que simplesmente aumentar a frequência de operação dos núcleos individuais. Isso porque o aumento da frequência causava maior consumo energético e aquecimento, resultando em um limite prático que dificultava ganhos significativos em desempenho quando se utilizava uma única unidade de processamento. Assim, os processadores modernos integram dois ou mais núcleos independentes, capazes de executar múltiplas tarefas simultaneamente, aumentando o throughput e a eficiência no uso do hardware.

Exemplos de arquiteturas modernas

Atualmente, destacam-se três grandes famílias de arquiteturas de CPUs que ilustram as diferentes abordagens comerciais e técnicas no mercado: Intel Core, AMD Ryzen e ARM.

- **Intel Core:** A arquitetura Intel Core, lançada em 2006, representa a evolução dos processadores Intel voltados ao mercado desktop e mobile. Com múltiplos núcleos, a arquitetura Core combina alta frequência de clock, avançadas técnicas de previsão de instruções (branch prediction) e unidades de execução paralelas para melhorar tanto o desempenho em tarefas sequenciais como em

workloads paralelos. A Intel continuamente investe em otimizações para consumo energético e integração de funcionalidades, como suporte nativo à virtualização e instruções para aceleração criptográfica.

- **AMD Ryzen:** A linha Ryzen, introduzida pela AMD em 2017, marca uma mudança significativa no design de CPUs, adotando a microarquitetura “Zen”, que prioriza uma arquitetura multi-core eficiente, com foco em escalabilidade e desempenho por watt. Os processadores Ryzen destacam-se pelo elevado número de núcleos e threads, além de amplo suporte a instruções modernas. Esse modelo trouxe competitividade ao mercado, impulsionando avanços tecnológicos e preços mais acessíveis para soluções high-end e mainstream.
- **Arquitetura ARM:** Originariamente projetada para dispositivos móveis e embarcados devido à sua eficiência energética, a arquitetura ARM fundamenta-se em um conjunto de instruções RISC (Reduced Instruction Set Computing), que favorece a simplicidade e a velocidade de execução. Nos últimos anos, com avanços como a linha ARM Cortex e o design big.LITTLE, que combina núcleos de alto desempenho com núcleos eficientes, a arquitetura ARM expandiu sua presença para laptops, servidores e supercomputadores. Essa diversidade de aplicação mostra uma alternativa robusta e eficiente frente às arquiteturas tradicionais baseadas em x86.

Embora compartilhem o objetivo de executar tarefas computacionais, essas arquiteturas diferem em aspectos como o conjunto de instruções, o número e tipo de núcleos, e as técnicas utilizadas para balancear desempenho e consumo energético. Por exemplo, os processadores Intel Core e AMD Ryzen baseiam-se na arquitetura x86-64, que é mais complexa e oferece uma ampla gama de instruções, enquanto os processadores ARM apostam em um design mais enxuto e eficiente energeticamente.

Evolução e motivação para múltiplos núcleos

A migração dos processadores de núcleo único para múltiplos núcleos não se deveu apenas ao aumento da quantidade de transistores previsto na Lei de Moore, mas também à necessidade de lidar com limites físicos que impedem o crescimento ilimitado da frequência dos processadores, como o aquecimento excessivo e o consumo energético elevado. Através da multiplicação do número de núcleos, os processadores modernos conseguem realizar múltiplas operações paralelas, melhorando significativamente o desempenho em aplicações que podem explorar esse paralelismo, tais como edição de vídeos, simulações, jogos e aplicações científicas.

Além disso, o avanço no desenvolvimento de sistemas operacionais e softwares que suportam multiprocessamento e multithreading foi essencial para que os benefícios dos processadores multi-core fossem plenamente aproveitados. Sem essa adaptação, os múltiplos núcleos poderiam permanecer ociosos, limitando o ganho de performance. Portanto, a evolução das CPUs acompanha não só mudanças na arquitetura física, mas também avanços na camada de software que gerencia a execução das tarefas.

A arquitetura das CPUs modernas é resultado de décadas de inovação que equilibram complexidade, desempenho e eficiência energética, refletindo as demandas

cada vez maiores por capacidade computacional em diferentes contextos de aplicação. A compreensão dessas características é essencial para estudantes e profissionais que buscam entender os fundamentos da computação atual e o futuro das arquiteturas de processadores.

Arquitetura de GPUs

Comparação entre arquiteturas de CPUs e GPUs

As GPUs (Unidades de Processamento Gráfico) possuem uma arquitetura fundamentalmente distinta das CPUs, projetadas para atender a necessidades específicas de processamento paralelo massivo, principalmente em tarefas gráficas e computacionais. Enquanto as CPUs são otimizadas para executar um pequeno número de threads complexos com controle sofisticado de fluxo, as GPUs são construídas para manipular milhares de threads simples simultaneamente, oferecendo uma alta taxa de throughput.

Em termos estruturais, a CPU é composta por poucos núcleos poderosos, cada um capaz de executar múltiplas instruções por ciclo nos seus pipelines complexos, além de suportar predição de salto, processamento fora de ordem e grandes caches para minimizar a latência. Isso torna as CPUs ideais para tarefas sequenciais e com forte dependência de controle lógico.

Por outro lado, a GPU possui centenas ou milhares de núcleos mais simples, organizados em grupos de execução paralela, que são especialmente eficientes para tarefas homogêneas e altamente paralelizáveis, como cálculos matriciais, processamento de pixels e vértices, além de algoritmos em aprendizado de máquina e simulações científicas. Sua arquitetura enfatiza o throughput em vez da baixa latência, sacrificando a complexidade do núcleo individual pela capacidade de processar muitas operações idênticas concorrentemente.

Essa diferença pode ser sintetizada da seguinte forma:

Aspecto	CPU	GPU
Núcleos	Poucos, complexos, alto desempenho por núcleo	Muitos, simples, alto paralelismo
Controle de fluxo	Avançado, com predição e execução fora de ordem	Simples, otimizado para execução em SIMD (Single Instruction Multiple Data)
Cache	Grandes caches para reduzir latência	Caches menores, maior dependência de largura de banda da memória

Aspecto	CPU	GPU
Objetivo	General-purpose: flexibilidade para diversas tarefas	Alto desempenho em paralelismo massivo, especialmente gráfico e científico
Paradigma de programação	Execução sequencial e paralela limitada a poucos threads	Execução massiva e simultânea de milhares de threads

Evolução das arquiteturas de GPUs

Primeira geração: GPUs fixas para renderização

No final dos anos 1990, as primeiras GPUs surgiram para acelerar tarefas específicas de renderização de gráficos 3D, inicialmente com funções fixas e sem possibilidade de programação pelo usuário. Essas placas eram capazes de realizar operações básicas de transformação geométrica, textura e rasterização, liberando a CPU dessas tarefas pesadas.

Segunda geração: GPUs programáveis e shaders

A partir dos anos 2000, com a introdução dos shaders programáveis, as GPUs passaram a utilizar unidades de processamento capazes de programar os estágios do pipeline gráfico. Isso permitiu personalizar comportamentos para sombreamento de vértices (vertex shaders) e pixels (pixel shaders), dando mais flexibilidade e possibilidades visuais para jogos e aplicações gráficas. Essas unidades reconfiguráveis contribuíram para o aumento do paralelismo e o desempenho geral.

Terceira geração: Arquiteturas unificadas e paralelismo massivo

A revolução da NVIDIA na arquitetura CUDA, lançada em meados de 2006 com a geração Tesla, introduziu um modelo de programação paralelo baseado em múltiplos núcleos unificados, capazes de executar operações gerais (GPGPU – General-Purpose GPU Computing). Essa unificação de processamento gráfico e computacional permitiu usar a GPU para aplicações além do gráfico, como simulações científicas e inteligência artificial.

A arquitetura passou a incluir milhares de núcleos CUDA simples organizados em SMs (Streaming Multiprocessors), que executam milhares de threads em paralelo, suportando diversas operações simultâneas e otimizando o uso dos recursos de hardware.

Quarta geração: GPUs modernas com alta eficiência e especialização

As arquiteturas mais recentes, como NVIDIA Ampere e AMD RDNA 2+, incorporam melhorias significativas em eficiência energética, capacidade computacional e sistemas de memória. Essas GPUs apresentam suporte para:

- **Ray tracing em hardware:** para simulação de luz e sombras com realismo.
- **Tensor cores:** unidades especializadas para operações matriciais utilizadas em aprendizado profundo.
- **Memórias de alta largura de banda:** GDDR6, HBM2 ou superiores, garantindo comunicação rápida entre a GPU e a memória global.
- **Suporte a métodos avançados de paralelismo e sincronização:** facilitando a programação eficiente e o escalonamento de milhares de threads.

Essas inovações consolidam as GPUs como plataformas heterogêneas não apenas para gráficos, mas também para computação científica, análise de dados, inteligência artificial e sistemas em nuvem.

Mecanismos de paralelismo e funcionamento interno das GPUs

O principal diferencial das GPUs está em sua capacidade de paralelismo massivo, obtido através da execução simultânea de milhares de threads organizados em blocos. O modelo é frequentemente descrito como SIMT (Single Instruction, Multiple Threads), uma variação do SIMD, onde múltiplas threads seguem a mesma instrução em paralelo, podendo divergir em execuções condicionais.

O pipeline gráfico, que é o coração da GPU, pode ser dividido nas seguintes etapas principais:

- **Input Assembler:** recebe e prepara os dados dos vértices.
- **Vertex Shader:** processa os vértices individualmente aplicando transformações geométricas.
- **Geometry Shader (opcional):** gera ou modifica geometria adicional.
- **Rasterizer:** converte primitivas vetoriais em fragmentos para serem processados.
- **Pixel (Fragment) Shader:** calcula a cor, textura e efeitos de cada fragmento.
- **Output Merger:** combina os fragmentos para produzir a imagem final.

Além do pipeline gráfico, as unidades computacionais da GPU (como os SMs) executam kernels paralelos aplicados a dados não gráficos. Esses kernels são escritos em APIs como CUDA, OpenCL e Vulkan, permitindo que a GPU realize operações massivas em paralelo, aproveitando sua arquitetura.

A arquitetura de memória também é fundamental para o desempenho da GPU. Ela possui uma hierarquia que inclui memória global de alta largura de banda, memória

compartilhada rápida entre threads de um mesmo bloco, caches de textura e constantes, e registradores locais para cada thread. O gerenciamento eficiente dessa memória é essencial para maximizar a utilização da GPU e reduzir os tempos de espera.

Modelo de programação para GPUs

Introdução aos modelos de programação para GPUs

A programação eficiente de GPUs exige o entendimento de modelos específicos que exploram sua arquitetura massivamente paralela. Entre os principais modelos usados estão o CUDA, desenvolvido pela NVIDIA, e o OpenMP, amplamente utilizado para programação paralela em geral, que também pode ser aplicado em GPUs. Esses modelos permitem que desenvolvedores criem programas que aproveitam milhares de threads executando simultaneamente, maximizando o desempenho em aplicações paralelizáveis.

CUDA: Conceitos e estrutura do modelo de programação

CUDA (Compute Unified Device Architecture) é uma plataforma de computação paralela e um modelo de programação criado pela NVIDIA para GPUs. Ele permite a execução de programas chamados *kernels* diretamente na GPU, utilizando milhares de threads organizadas hierarquicamente em blocos e grids.

A unidade básica de execução do CUDA é a **thread**. Threads são agrupadas em **blocks** (blocos) e estes, por sua vez, compõem um **grid**. Esse modelo tridimensional permite a organização dos dados em múltiplas dimensões, facilitando o mapeamento de algoritmos paralelos.

Dimensões e indexação: threadIdx e blockIdx

Cada thread possui índices que indicam sua posição dentro do bloco e do grid, definidos pelas variáveis predefinidas:

- threadIdx.x, threadIdx.y, threadIdx.z: identificação da thread dentro do bloco, em até três dimensões;
- blockIdx.x, blockIdx.y, blockIdx.z: identificação do bloco dentro do grid, também em até três dimensões;
- blockDim.x, blockDim.y, blockDim.z: dimensão total do bloco em cada eixo.

Com esses índices, é possível calcular a posição global da thread e determinar qual parte dos dados ela deve processar, tornando o paralelismo explícito e ordenado.

Exemplo ilustrativo simples em CUDA

```
__global__ void somaVetores(int *a, int *b, int *c, int n) {
  int idx = blockIdx.x * blockDim.x + threadIdx.x;
```

```
  if (idx < n) {
    c[idx] = a[idx] + b[idx];
  }
}

int main() {
  // Inicialização e alocação omitidas para brevidade
  int n = 1024;
  int blockSize = 256;
  int numBlocks = (n + blockSize - 1) / blockSize;
  somaVetores<<<numBlocks, blockSize>>>(a_dev, b_dev, c_dev, n);
  // Sincronização e cópia de dados omitidas
}
```

Neste exemplo, cada thread calcula a soma de um elemento dos vetores a e b, armazenando o resultado em c. Utiliza-se um grid unidimensional, mas esse conceito é facilmente estendido para duas ou três dimensões usando as variáveis threadIdx.y, threadIdx.z e seus equivalentes para blocos.

OpenMP e programação SIMD em GPUs

OpenMP é um modelo de programação paralela baseado em diretivas para linguagens como C, C++ e Fortran. Tradicionalmente usado para paralelismo em CPUs, desde sua versão 4.0 teve adicionado suporte para programação em aceleradores, incluindo GPUs, por meio de *offloading*. Além disso, OpenMP permite explorar o paralelismo SIMD (Single Instruction Multiple Data), muito eficiente para operações vetoriais típicas em GPUs.

As diretivas OpenMP para paralelismo SIMD indicam ao compilador que se pode aplicar uma mesma instrução a múltiplos dados simultaneamente, possibilitando a vetorização pelo hardware do GPU.

Principais diretivas OpenMP para SIMD

- #pragma omp simd: paraleliza loops aplicando instrução única para dados múltiplos;
- #pragma omp parallel for simd: combina execução paralela de múltiplas threads com vetorização SIMD;
- #pragma omp target: indica regiões do código que devem ser executadas na GPU (offloading).

Exemplo ilustrativo com OpenMP SIMD

```
#include <omp.h>
void soma_vetores_simd(float *a, float *b, float *c, int n) {
  #pragma omp target teams distribute parallel for simd
  for (int i = 0; i < n; i++) {
    c[i] = a[i] + b[i];
  }
}
```

Essa função utiliza OpenMP para executar a soma de vetores diretamente na GPU, automatizando a geração do código paralelo e a vetorização, além de realizar o

offload da computação para o dispositivo acelerador. O uso conjunto de parallel for simd possibilita combinar paralelismo em nível de threads com o paralelismo SIMD.

Importância dos modelos CUDA e OpenMP na programação de GPUs

A adoção de modelos como CUDA e OpenMP é crucial para explorar plenamente o potencial de paralelismo das GPUs, que apresentam milhares de núcleos simples, cada um capaz de executar threads simultâneas. Enquanto CUDA oferece um controle preciso e detalhado da arquitetura NVIDIA, permitindo otimizações específicas, OpenMP traz maior portabilidade e facilidade para programadores já habituados ao paralelismo em CPUs.

Esses modelos revolucionaram a forma de desenvolver software para tarefas altamente paralelizáveis, como processamento científico, inteligência artificial, simulações físicas, análise de dados e renderização gráfica avançada, promovendo ganhos expressivos de desempenho que seriam inacessíveis em arquiteturas convencionais.

Assim, dominar essas técnicas e compreender a estrutura subjacente de blocos, threads e diretivas paralelas é fundamental para estudantes e profissionais que buscam inovar e otimizar aplicações em ambientes de computação paralela modernos, especialmente na área de computação acelerada por GPUs.

Referências bibliográficas

- Ducatte, L. C. (2009). *Introdução à arquitetura de GPUs*. Disponível em: https://www.ic.unicamp.br/~ducatte/mo401/1s2009/T2/045116-t2.pdf
- Moraes, L. F. (2018). *Introdução à programação paralela em GPU* (Slides). Disponível em: https://lief.if.ufrgs.br/pub/Cursos/Cuda/aula01.pdf
- Bertoldo, A. L. (2015). *Análise de Desempenho da Arquitetura CUDA utilizando os NAS Parallel Benchmark* (Trabalho de Conclusão de Curso). Universidade Federal do Rio Grande do Sul. Disponível em: https://www.lume.ufrgs.br/bitstream/handle/10183/18536/000730588.pdf?sequence=1
- Ducatte, L. C. (2012). *Arquitetura e programação de GPU Nvidia*. Disponível em: https://ic.unicamp.br/~ducatte/mo401/1s2012/T2/G02-001963-023169-085937-t2.pdf
- Analytics Vidhya. (2023). *CPU vs GPU: Why GPUs are More Suited for Deep Learning?* Disponível em: https://www.analyticsvidhya.com/blog/2023/03/cpu-vs-gpu/
- Spiceworks. (s.d.). *CPU vs. GPU: 11 Key Comparisons*. Disponível em: https://www.spiceworks.com/tech/hardware/articles/cpu-vs-gpu/
- Hennessy, J. L., & Patterson, D. A. (2019). *Computer Architecture: A Quantitative Approach* (6ª ed.). Morgan Kaufmann.