Capítulo 1

Exploração do Paralelismo nas Arquiteturas de Computadores Atuais

Guilherme Galante

Resumo

A exploração do paralelismo em arquiteturas de computadores atuais tem se tornado uma área de extrema relevância e interesse, impulsionada pela necessidade de lidar com cargas de trabalho cada vez mais complexas e exigentes. Com o avanço das tecnologias e a demanda por desempenho computacional escalável, tem-se buscado diferentes formas de aproveitar o paralelismo em diversos níveis, desde instruções vetoriais do processador até a exploração de sistemas de memória distribuída e aceleradores. Essa abordagem permite a execução de múltiplas tarefas simultaneamente, resultando em ganhos significativos de desempenho. Sob este escopo, este minicurso tem o objetivo de oferecer uma visão geral dos diferentes aspectos do paralelismo nas arquiteturas de computadores atuais.

1.1. Introdução

Recentemente, a demanda por capacidades computacionais tem crescido exponencialmente, impulsionada por uma variedade de aplicações e setores. A sociedade atual depende cada vez mais da computação para impulsionar a inovação, resolver problemas complexos e melhorar a eficiência em diversas áreas de conhecimento [Pacheco and Malensek 2021]. Algumas das áreas que demonstram uma crescente demanda por capacidades de computação incluem:

- Ciência de Dados e Inteligência Artificial (IA): Com o aumento da coleta de dados, a análise e o processamento desses dados para extrair informações significativas requerem poder computacional considerável. Algoritmos de IA, como aprendizado de máquina e redes neurais, exigem enormes recursos de computação para treinamento e inferência.
- Pesquisa Científica e Simulações: Disciplinas como física, química, biologia e climatologia dependem fortemente de simulações computacionais para modelar fenô-

menos complexos, prever resultados e entender o mundo natural em um nível fundamental.

- Medicina e Biologia Computacional: A análise de dados genômicos, simulações de processos biológicos e o desenvolvimento de novos medicamentos são algumas das áreas que se beneficiam significativamente do poder computacional para acelerar a descoberta e a inovação.
- Finanças e Mercados: Em mercados financeiros altamente dinâmicos, algoritmos de negociação de alta frequência e modelagem de riscos exigem capacidades computacionais significativas para análise de dados em tempo real e tomada de decisões automatizadas.
- Entretenimento e Mídia: Com o avanço da tecnologia de realidade virtual (VR) e realidade aumentada (AR), assim como a produção de conteúdo multimídia de alta qualidade, a demanda por renderização de gráficos em tempo real e processamento de vídeo aumentou drasticamente.

Essas são apenas algumas das muitas áreas que têm grandes demandas por computação. Em essência, praticamente todos os campos da ciência, da indústria e da sociedade contemporânea dependem, em maior ou menor grau, do poder computacional para avançar em suas respectivas áreas de atuação.

Nesse contexto, uma das alternativas para satisfazer essas demandas é o emprego de computação paralela. Computação paralela é uma abordagem na qual múltiplas tarefas computacionais são executadas simultaneamente, seja dividindo uma única tarefa em partes menores que podem ser executadas em paralelo ou realizando tarefas independentes simultaneamente. Essa técnica é fundamental para lidar com cargas de trabalho intensivas e complexas, proporcionando um aumento significativo no desempenho e na eficiência dos sistemas computacionais.

A computação paralela envolve uma interação complexa entre hardware e software para aproveitar ao máximo os recursos disponíveis. No lado do hardware, empregam-se arquiteturas específicas, como processadores multicore, clusters de computadores e aceleradores especializados, para suportar a execução simultânea de múltiplas tarefas. Por outro lado, no aspecto do software, é fundamental ter algoritmos paralelos eficientes e estratégias de programação que distribuam efetivamente o trabalho entre os diversas unidades de processamento. Isso requer uma compreensão profunda da arquitetura subjacente do hardware e a habilidade de desenvolver código que possa ser executado de forma concorrente e coordenada.

1.2. Explorando o paralelismo nas arquiteturas atuais

O uso de paralelismo em arquiteturas modernas desempenha um papel fundamental no aumento do desempenho e na eficiência dos sistemas computacionais. Em vez de depender exclusivamente do aumento da frequência do clock dos processadores para melhorar o desempenho, como feito no passado, as arquiteturas modernas exploram o paralelismo em vários níveis. Isso inclui a utilização de instruções vetoriais e SIMD (*Single Instruction, Multiple Data*), o uso de GPUs e de arquiteturas paralelas de memória comparti-

lhada e distribuída [Wilkinson and Allen 2005]. Nas próximas seções são apresentados mais detalhes sobre os diferentes níveis de paralelismo nas arquiteturas e os modelos de programação que podem se usados em cada um deles.

1.2.1. Extensões vetoriais

As extensões vetoriais permitem que os processadores modernos realizem operações em paralelo em conjuntos de dados de forma muito mais eficiente do que seria possível com instruções de processamento escalares tradicionais. Existem várias extensões vetoriais disponíveis em processadores modernos, cada uma com suas próprias características e conjuntos de instruções.

Um exemplo são as instruções vetoriais *Advanced Vector Extensions* de 512 bits (AVX-512), introduzidas pela Intel em seus processadores x86. As instruções AVX permitem que um único processador execute operações em paralelo em múltiplos elementos de dados em um vetor, conhecido como vetor SIMD (*Single Instruction, Multiple Data*). Cada vetor SIMD pode conter vários elementos de dados, e uma única instrução AVX é capaz de realizar uma operação em todos esses elementos simultaneamente, resultando em um aumento significativo na taxa de execução de instruções e no desempenho geral do sistema.

As instruções AVX-512 são projetadas para operar em registradores de 512 bits. Isso significa que até 16 números de ponto flutuante de precisão simples (float de 32 bits) ou 8 números de ponto flutuante de dupla precisão (double de 64 bits) podem ser processados em paralelo em uma única instrução AVX. Além disso, as instruções AVX incluem uma ampla variedade de operações aritméticas, lógicas e de manipulação de dados, como adição, subtração, multiplicação, divisão, operações bitwise, mistura de dados e carregamento/armazenamento de dados de memória, o que permite uma ampla gama de operações paralelas em conjuntos de dados de diferentes tipos.

A Figura 1.1 mostra uma operação de soma de vetores usando as instruções vetoriais. Para o Intel AVX-512, pode-se armazenar 8 elementos de ponto flutuante de dupla precisão (*double*) em cada vetor, já que cada elemento desse tipo possui 64 bits e o vetor tem 512 bits, então 512/64 = 8 elementos *double*. A operação de soma é realizada para os 8 elementos usando uma única instrução SIMD. Como resultado, o Intel AVX pode potencialmente ser até 8 vezes mais rápido que a implementação escalar.

Note que ao escrever código que utiliza essas extensões vetoriais, é necessário verificar se o hardware de destino oferece suporte a essas instruções. Por exemplo, em sistemas operacionais Linux é possível verificar o suporte usando o comando:

```
grep avx /proc/cpuinfo
```

Como resposta ao comando exibe-se as flags do processador, destacando as extensões AVX (destacadas em vermelho), como ilustrado na Figura 1.2.

As extensões vetoriais podem ser exploradas de diferentes formas:

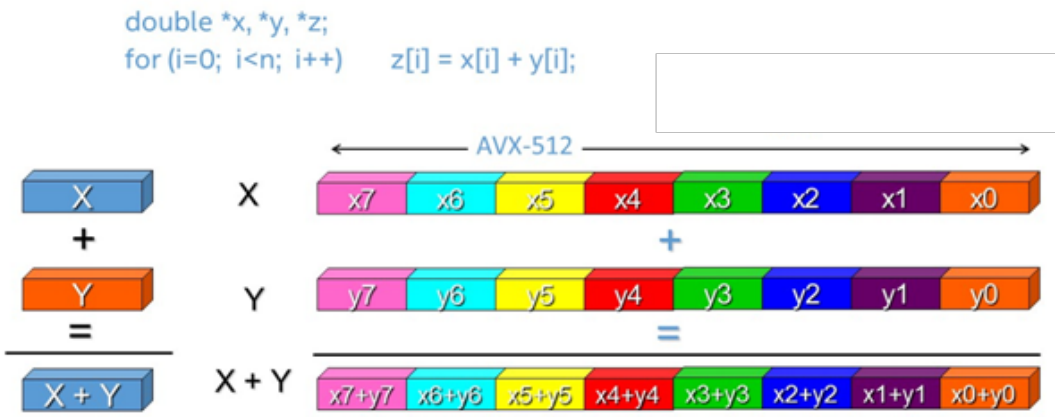

Figura 1.1. Soma de vetores usando AVX-512.

```
flags           : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush dts
 acpi mmx fxsr sse sse2 ss ht tm pbe syscall nx pdpe1gb rdtscp lm constant_tsc art arch_perfmon pebs
bts rep_good nopl xtopology nonstop_tsc cpuid aperfmperf tsc_known_freq pni pclmulqdq dtes64 monitor
ds_cpl vmx est tm2 ssse3 sdbg fma cx16 xtpr pdcm pcid sse4_1 sse4_2 x2apic movbe popcnt tsc_deadline_
timer aes xsave avx f16c rdrand lahf_lm abm 3dnowprefetch cpuid_fault epb cat_l2 invpcid_single cdp_l
2 ssbd ibrs ibpb stibp ibrs_enhanced tpr_shadow flexpriority ept vpid ept_ad fsgsbase tsc_adjust bmi1
 avx2 smep bmi2 erms invpcid rdt_a avx512f avx512dq rdseed adx smap avx512ifma clflushopt clwb intel_
pt avx512cd sha_ni avx512bw avx512vl xsaveopt xsavec xgetbv1 xsaves split_lock_detect dtherm ida arat
 pln pts hwp hwp_notify hwp_act_window hwp_epp hwp_pkg_req vnmi avx512vbmi umip pku ospke avx512_vbmi
2 gfni vaes vpclmulqdq avx512_vnni avx512_bitalg avx512_vpopcntdq rdpid movdiri movdir64b fsrm avx512
```

Figura 1.2. Suporte a extensões vetoriais AVX.

- **Compilação com suporte AVX**: Os compiladores geralmente têm opções para habilitar o suporte a AVX. Ao compilar programas usando esses compiladores, você pode aproveitar automaticamente as instruções AVX.
- **Bibliotecas otimizadas**: Muitas bibliotecas populares de computação numérica, como *Intel Math Kernel Library* (MKL) e *Intel Integrated Performance Primitives* (IPP), são otimizadas para aproveitar as instruções AVX-512 quando disponíveis. Isso significa que, ao usar essas bibliotecas em seu código, você pode obter automaticamente benefícios de desempenho ao executar operações numéricas intensivas.
- **Programação manual**: Para aplicações que exigem otimização extrema ou para desenvolvedores que desejam tirar o máximo proveito das instruções AVX-512, é possível escrever código assembly ou Intrinsics AVX-512 diretamente. Os Intrinsics AVX-512 são funções fornecidas pelo compilador C (como o GCC ou o Clang) que correspondem diretamente às instruções de assembly AVX-512.

A Figura 1.3 mostra um exemplod e soma de vetores em Assembly e usando os C Intrinsics. As instruções AVX estão destacadas em cores. Em ambos os casos, há o carregamento dos vetores para os registradores vetorias (vermelho), a soma é realizada (azul), e então o resultado é atribuído a um vetor (verde). Note que a operação de soma dos vetores é realizada utilizando uma única instrução.

```
section .data
vec1 dd 1.0, 2.0, 3.0, 4.0    ; vetor 1
vec2 dd 4.0, 3.0, 2.0, 1.0    ; vetor 2
result dd 0.0, 0.0, 0.0, 0.0  ; vetor resultado

section .text
global _start

_start:
    ; Carregando os vetores
    vmovups zmm0, [vec1]
    vmovups zmm1, [vec2]

    ; Realizando a operação de soma
    vaddps zmm2, zmm0, zmm1

    ; Armazenando o resultado
    vmovups [result], zmm2
```

```
int main() {
    // Vetores de entrada
    float vec1[] = {1.0f, 2.0f, 3.0f, 4.0f};
    float vec2[] = {4.0f, 3.0f, 2.0f, 1.0f};
    // Vetor de resultado
    float result[16];

    // Carregando os vetores em registradores ZMM
    __m512 vec1_avx = _mm512_loadu_ps(vec1);
    __m512 vec2_avx = _mm512_loadu_ps(vec2);

    // Realizando a operação de soma
    __m512 result_avx = _mm512_add_ps(vec1_avx, vec2_avx);

    // Armazenando o resultado em memória
    _mm512_storeu_ps(result, result_avx);
}
```

Figura 1.3. Programação AVX: Assembly e Intrinsics.

1.2.2. Memória Compartilhada: Multicores e Multiprocessadores

Computadores paralelos de memória compartilhada são sistemas de computação nos quais múltiplos processadores ou núcleos de processamento têm acesso a um único espaço de memória compartilhada. Isso significa que todos os processadores/núcleos podem ler e escrever na mesma área de memória, facilitando a comunicação e o compartilhamento de dados. Os sistemas de memória compartilhada podem variar em escala, desde sistemas com alguns núcleos de processamento (multicore) até sistemas com centenas ou milhares de processadores.

Um processador multicore combina dois ou mais unidades de processador (chamadas núcleos) em uma única peça de silício. Normalmente, cada núcleo consiste em

todos os componentes de um processador independente, como registradores, ULA, hardware de pipeline e unidade de controle, além de cache L1 de instruções e dados. Além dos múltiplos núcleos, os chips multicore contemporâneos também incluem cache L2 e, cada vez mais, cache L3 [Stallings 2015], como ilustrado na Figura 1.4. Todos os núcleos podem acessar diretamente todos os dados armazenados na memória principal. Comparado com a arquitetura tradicional de núcleo único, arquitetura multicore fornece computação muito maior capacidade e é muito mais eficiente em termos energéticos.

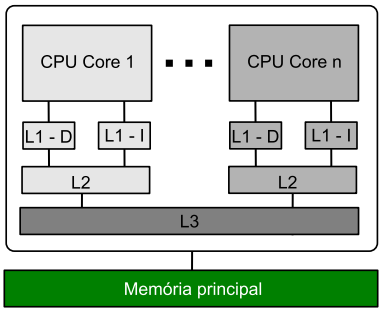

Figura 1.4. Organização de um processador multicore.

Embora os fabricantes de hardware continuem aumentando o número de núcleos em chips de CPU, o número de núcleos não pode ser aumentado ilimitadamente devido a limitações físicas. Para atender a necessidade de computadores com capacidades superiores de processamento, vários chips de CPU são integrados em uma única máquina. Esse tipo de arquitetura é chamada de multiprocessador.

Em sistemas de memória compartilhada com vários processadores, a interconexão pode conectar todos os processadores diretamente à memória principal ou cada processador pode ter uma conexão direta com um bloco de memória principal, e os processadores podem acessar os blocos de memória principal uns dos outros por meio de hardware especial incorporado aos processadores. No primeiro tipo de sistema, o tempo para acessar todos os locais de memória será o mesmo para todos os núcleos, enquanto no segundo tipo, um local de memória ao qual um núcleo está diretamente conectado pode ser acessado mais rapidamente do que um local de memória que precisa ser acessado por meio de outro chip. Assim, o primeiro tipo de sistema de sistema é chamado de sistema de acesso uniforme à memória (*Uniform memory access* - UMA), enquanto o segundo tipo é chamado de sistema de acesso não uniforme à memória (*Non-Uniform memory access* - NUMA).

Em teoria, máquinas UMA podem teoricamente alcançar um grande número de processadores, mas na prática, há limitações físicas e de design que podem restringir o número máximo de processadores em uma única máquina. Geralmente, sistemas UMA podem acomodar dezenas a centenas de processadores em uma única máquina, dependendo do projeto específico e das restrições de hardware. Por sua vez, sistemas NUMA podem escalar para um número significativo de processadores, mas a capacidade de expansão é geralmente mais flexível do que em sistemas SMP, podendo suportar centenas ou até mesmo milhares de processadores. Figura 1.5 ilustra a diferença entre os dois tipos de arquitetura.

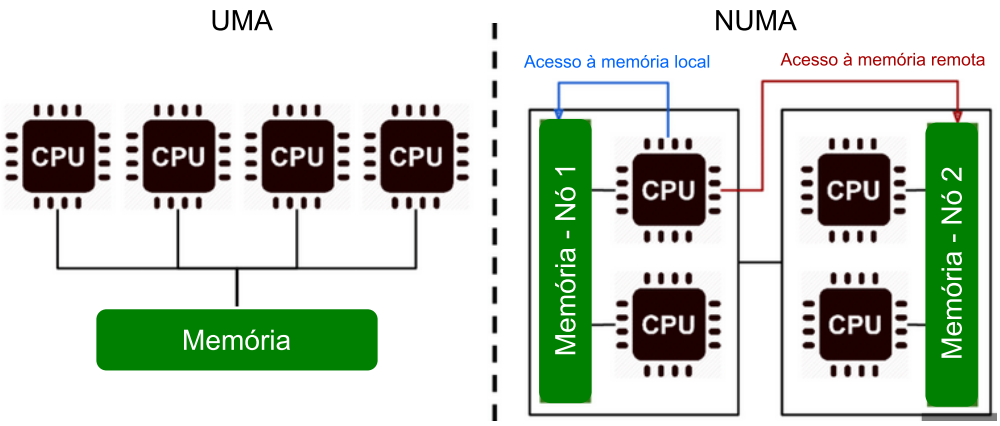

Figura 1.5. UMA vs. NUMA.

Um dos modelos de programação mais comuns para computadores paralelos de memória compartilhada é o modelo de programação paralela com threads. Nesse modelo, cada thread de execução é atribuída a um núcleo de processamento e pode acessar diretamente o espaço de memória compartilhada. Isso permite que as threads cooperem entre si, trocando informações e coordenando suas atividades por meio da memória compartilhada.

OpenMP é uma API (*Application Programming Interface*) de programação paralela que é amplamente utilizada para aproveitar o paralelismo de threads em arquiteturas de memória compartilhada [Chandra et al. 2001]. A interface é definida pela coleção de diretivas de compilador (`#pragmas`), funções de biblioteca, e e variáveis de ambiente. Esses elementos permitem que os usuários indicar a um compilador as partes de um programa sequencial que podem ser executado em paralelo.

Na Figura 1.6 apresenta-se um código-fonte C usando OpenMP para a soma de vetores utilizando múltiplas threads. Este código realiza a soma de dois vetores a e b e armazena o resultado no vetor c. Ele utiliza a diretiva `#pragma omp parallel for` para distribuir o laço for em paralelo entre as threads disponíveis. Para isso, o programa começa a execução com uma única thread, chamada de thread mestre. Quando a primeira região paralela, definida pela diretiva `parallel`, é encontrada, a thread mestre cria um time de threads que executam uma parte das somas dos vetores em diferentes núcleos ou processadores.

Além do OpenMP, existem várias outras APIs e bibliotecas que podem ser utilizadas para programação multithread em diferentes plataformas e linguagens de programação. Algumas das mais comuns incluem *Pthreads*, *Threading Building Blocks* (TBB), *Cilk*, entre outras.

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

#define N 1000000

int main() {
    int i;
    double a[N], b[N], c[N];

    // Inicializando os vetores
    for (i = 0; i < N; i++) {
        a[i] = i; b[i] = i * 2;
    }

    // Paralelizando a soma dos vetores
    #pragma omp parallel for shared(a, b, c) private(i)
    for (i = 0; i < N; i++)
    {
        c[i] = a[i] + b[i];
    }

    return 0;
}
```

Figura 1.6. OpenMP: Soma de vetores.

1.2.3. Memória distribuída: Clusters

Um cluster de computadores é uma arquitetura de sistema distribuído composta por vários computadores interconectados e trabalhando juntos como se fossem uma única entidade de processamento. Geralmente, esses computadores individuais, chamados de nós, são conectados por meio de uma rede de alta velocidade, permitindo comunicação eficiente entre eles, como ilustrado na Figura 1.7. Note que os nós desses sistemas geralmente são sistemas de memória compartilhada com um ou mais processadores de vários núcleos.

Nesses sistemas, cada nó de processamento opera de forma autônoma e pode executar tarefas independentes, ou colaborar com outros nós para realizar tarefas mais complexas. A comunicação entre os nós é realizada por meio de troca de mensagens pela rede de comunicação, e os dados precisam ser explicitamente transferidos entre os nós quando necessário, considerando que cada nó acessa apenas a sua própria memória. Esses sistemas são frequentemente utilizados em ambientes de computação de alto desempenho e em aplicações distribuídas, onde a escalabilidade e a capacidade de processamento em paralelo são essenciais, podendo apresentar dezenas ou até milhares de nós.

Em relação às APIs de programação para clusters, geralmente utilizam-se implementações do padrão de *Message Passing Interface* (MPI). O MPI permite a execução de aplicações paralela tanto em sistemas de memória distribuída quanto em sistemas de memória compartilhada. Tipicamente, uma aplicação MPI consiste em vários processos que podem trocar dados enviando mensagens, seja utilizando primitivas de comunicação ponto a ponto ou primitivas de comunicação coletiva.

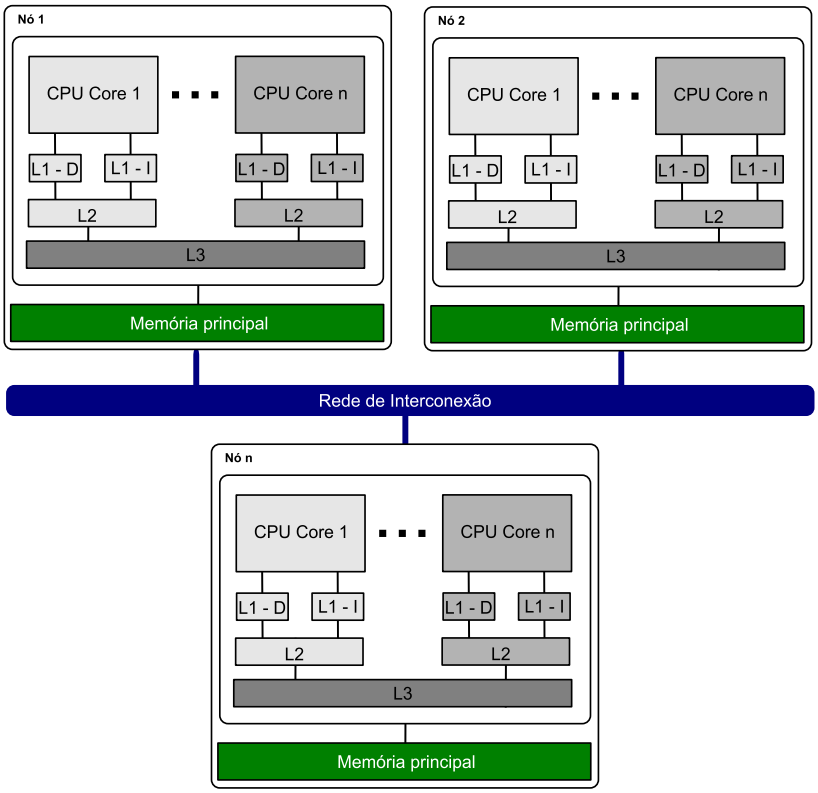

Figura 1.7. Cluster de computadores.

Na Figura 1.8 apresenta-se um trecho de código MPI (em linguagem C) usando as primitivas de comunicação ponto a ponto `Send` e `Recv` para o envio de dados. A operação Send é usada para enviar dados de um processo para outro. Ela inclui o endereço do buffer de dados a ser enviado, o tamanho dos dados a serem enviados, o tipo de dados a serem enviados e o identificador do processo de destino. O processo de destino deve estar pronto para receber a mensagem antes que a operação send seja concluída. Por outro lado, a operação Recv é usada para receber dados de um processo remoto. Ela especifica o endereço do buffer onde os dados recebidos serão armazenados, o tamanho máximo dos dados a serem recebidos, o tipo de dados a serem recebidos e o identificador do processo de origem [Silva et al. 2022].

Essas operações de comunicação são usadas em uma variedade de cenários em MPI. Por exemplo, em programas paralelos simples, um processo pode enviar dados para outro processo para coordenação ou troca de informações. No exemplo, temos dois processos (P0 e P1), sendo que o processo P0 envia um array de 100 elementos do tipo double para o processo P1.

```
//Processo enviando mensagem
double buffer[100];
int to=1, tag=123, error;
error=MPI_Send(buffer, 100, MPI_DOUBLE, to, tag, MPI_COMM_WORLD);


//Processo recebendo a mensagem
double buffer[100];
MPI_Status status;
int from=0, tag=123, error;
error=MPI_Recv(buffer, 100,MPI_DOUBLE, from, tag, MPI_COMM_WORLD, &status);
```

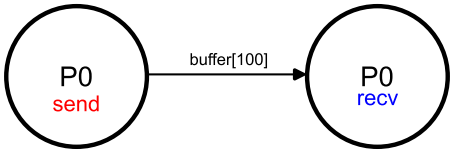

Figura 1.8. MPI: Troca de mensagens ponto a ponto.

1.2.4. GPGPUs

Inicialmente projetadas para renderizar gráficos complexos em jogos e aplicações de modelagem 3D, as GPUs modernas evoluíram para serem utilizadas também em tarefas de computação intensiva, como simulações científicas, aprendizado de máquina, análise de dados e muito mais.

A exploração do paralelismo em placas gráficas, conhecida como GPGPU (*General-Purpose computing on Graphics Processing Units*), é uma técnica que utiliza as unidades de processamento gráfico para realizar tarefas de propósito geral além do processamento gráfico tradicional. As GPUs são altamente paralelas e possuem muitos núcleos de processamento, o que as torna ideais para acelerar uma variedade de aplicações que podem se beneficiar do processamento em paralelo.

Por exemplo, as GPUs da NVIDIA são divididas em vários SMs (*Streaming Multiprocessors*), que por sua vez, executam grupos de threads, chamados de warps. Cada SM possui vários núcleos, chamados de CUDA cores. Cada CUDA core possui pipelines completos de operações aritméticas e de pontos flutuantes. A Figura 1.9 apresenta a arquitetura da GPU NVIDIA A100, equipada com 6.912 (64 por SM) núcleos CUDA memória dedicada de 40GB de memória. A GPU e sua memória associada geralmente são fisicamente separadas da CPU e de sua memória. Na documentação da NVIDIA, a CPU junto com sua memória associada é frequentemente chamada de *host*, e a GPU junto com sua memória é chamada de *device*.

As arquiteturas de computação paralela heterogênea CPU + GPU evoluíram porque a CPU e a GPU possuem atributos complementares que permitem que as aplicações tenham melhor desempenho usando os dois tipos de processadores. Portanto, para um desempenho ideal, é necessário usar CPU e GPU para a aplicação, executando as partes na CPU (sequenciais ou paralelas) e as partes paralelas de dados intensivos na GPU, conforme mostrado na Figura 1.10. Escrever código dessa forma garante que as carac-

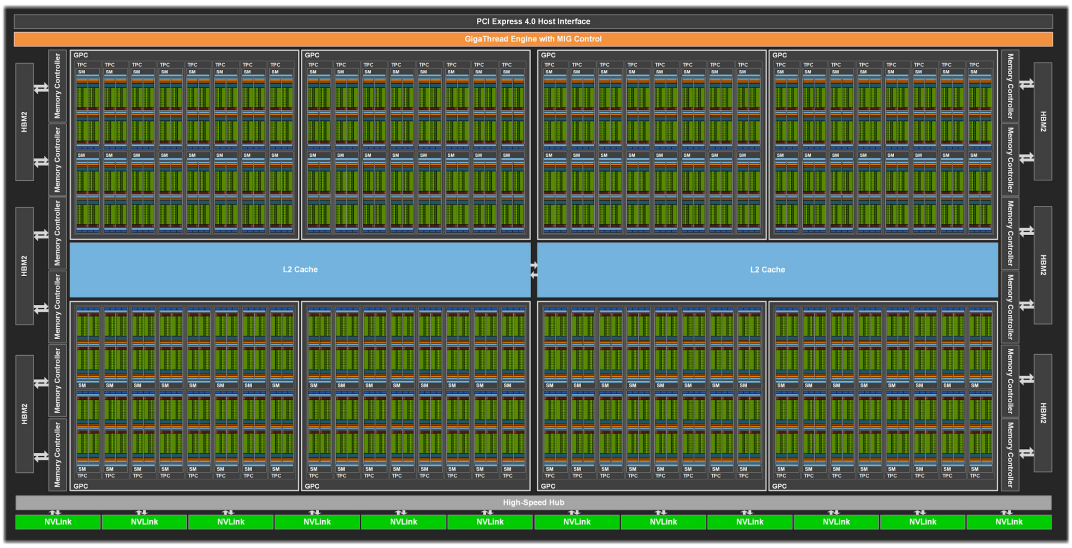

Figura 1.9. Arquitetura NVIDIA A100.

terísticas da GPU e da CPU se complementem, levando à utilização total do poder computacional do sistema combinado. Para suportar a execução conjunta de CPU + GPU de uma aplicação, a NVIDIA projetou um modelo de programação chamado CUDA [Cheng et al. 2014]. CUDA oferece um modelo de programação que permite aos desenvolvedores escreverem código para ser executado nas GPUs NVIDIA usando uma extensão da linguagem de programação C/C++.

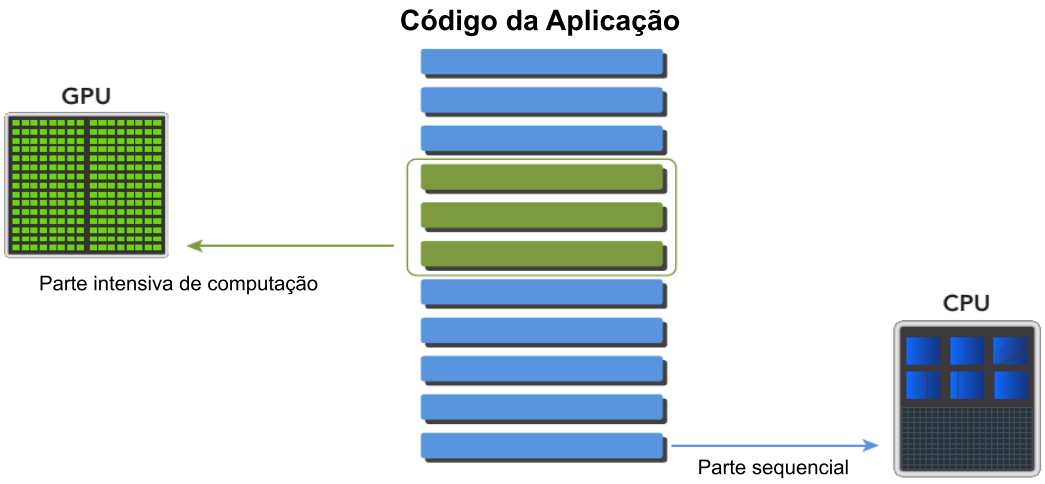

Figura 1.10. Computação heterogênea usando CPU + GPU. Adaptado de [Cheng et al. 2014]

Na Figura 1.11 compara-se um programa hello world em C e CUDA lado a lado. A principal diferença entre a implementação C e CUDA é o especificador __global__ e a sintaxe <<< ... >>>. O especificador __global__ indica uma função executada nna

GPU (device). Essa função pode ser chamada através do código host (a função *main()* no exemplo) e também é conhecida como *kernel*. Quando um kernel é chamado, sua configuração de execução é fornecida através da sintaxe $<<<$ ... $>>>$, por exemplo, *cuda_hello* $<<<1,1>>>()$.

C

```c
void c_hello(){
    printf("Hello World!\n");
}

int main() {
    c_hello();
    return 0;
}
```

CUDA

```c
__global__ void cuda_hello(){
    printf("Hello World from GPU!\n");
}

int main() {
    cuda_hello<<<1,1>>>();
    return 0;
}
```

Figura 1.11. Hello World em C e CUDA.

A curva de aprendizado do CUDA pode ser desafiadora devido à complexidade da programação de GPU e à necessidade de entender detalhes da arquitetura e hierarquia de memória. No entanto, há outras alternativas de mais alto nível, como o OpenACC [Silva et al. 2022]. OpenACC é uma API de programação paralela projetada para simplificar o desenvolvimento de aplicações para GPUs e outros aceleradores. O OpenACC permite adicionar diretivas (#pragmas) ao código existente para indicar onde e como paralelizar, muito semelhante ao OpenMP, como pode-se observar na Figura 1.12.

```c
#include <stdio.h>
#define N 1000000000

int main(void) {
   double pi = 0.0f; long i;
   #pragma acc parallel loop reduction(+: pi)
   for (i=0; i<N; i++)
   {
     double t=(double) ((i+0.5)/N);
     pi += 4.0/(1.0+t*t);
   }
   printf("pi=%f\n",pi/N);
   return 0;
}
```

Figura 1.12. cálculo de PI usando OpenACC.

Neste exemplo, calcula-se Este uma estimativa do valor de π (pi) usando o método da soma de Riemann. Destaca-se a linha "`#pragma acc parallel loop reduction(+: pi)`" que indica a paralelização do loop e a aplicação uma redução à variável pi. Isso significa que cada thread irá calcular sua própria soma parcial de pi e, no final, todas as somas parciais serão somadas para obter o valor final de pi. Note que com uma única linha adicional de código é possível paralelizar código para GPUs.

1.2.5. Cloud Computing

Ambientes de nuvem oferecem uma ampla variedade de recursos e serviços que podem ser utilizados para explorar o paralelismo e executar tarefas computacionais de forma escalável. Essas plataformas permitem que os desenvolvedores implantem e gerenciem sistemas distribuídos e paralelos na nuvem, aproveitando recursos de computação sob demanda para lidar com cargas de trabalho de tamanho e complexidade diversas.

Atualmente, os provedores de nuvem oferecem uma ampla gama de opções de máquinas virtuais e instâncias projetadas especificamente para atender às demandas de processamento de alto desempenho. Essas instâncias não apenas abrangem CPUs com múltiplos núcleos mas também podem incluir GPUs dedicadas. Além disso, a flexibilidade da computação em nuvem permite a criação de clusters sob demanda, onde diversas instâncias podem ser facilmente combinadas para formar um ambiente distribuído de processamento paralelo.

Para exemplificar, as instâncias P3[1] do Amazon EC2 são otimizadas para aplicações HPC e machine learning distribuído. Essas instâncias fornecem até 100 Gbps de taxa de transferência de redes, 96 vCPUs Intel Xeon, 8 GPUs NVIDIA V100 Tensor Core com 32 GiB de memória cada. Outro exemplo, é a ferramenta AWS ParallelCluster para a implantação e o gerenciamento de clusters de computação de alta performance na Amazon. O ParallelCluster[2] usa uma interface gráfica de usuário (GUI) simples ou um arquivo de texto para modelar e provisionar os recursos necessários de forma automatizada.

1.3. Considerações Finais

A exploração de paralelismo em arquiteturas modernas representa uma abordagem fundamental para lidar com os desafios crescentes de processamento de dados em um mundo cada vez mais digital e orientado por dados. A utilização de múltiplos níveis de paralelismo possibilita a realização de tarefas complexas de forma rápida e eficiente. No entanto, é importante reconhecer que a exploração do paralelismo também apresenta desafios, como a necessidade de desenvolver e otimizar software para aproveitar totalmente os recursos paralelos disponíveis. O contínuo avanço e aprimoramento das arquiteturas modernas garantem que o paralelismo continuará desempenhando um papel central na evolução da computação e no enfrentamento dos desafios computacionais do futuro.

Para saber mais sobre arquiteturas paralelas e programação paralela, recomenda-se [Pacheco and Malensek 2021, Wilkinson and Allen 2005, Silva et al. 2022] e demais referências citadas.

[1] `https://aws.amazon.com/pt/ec2/instance-types/p3/`
[2] `https://aws.amazon.com/pt/hpc/parallelcluster/`

Referências

[Chandra et al. 2001] Chandra, R., Dagum, L., Kohr, D., Maydan, D., McDonald, J., and Menon, R. (2001). *Parallel programming in OpenMP*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[Cheng et al. 2014] Cheng, J., Grossman, M., and McKercher, T. (2014). *Professional CUDA C Programming*. Wrox Press.

[Pacheco and Malensek 2021] Pacheco, P. and Malensek, M. (2021). *An introduction to parallel programming*. Morgan Kaufmann, Oxford, England, 2 edition.

[Silva et al. 2022] Silva, G. P., Bianchini, C. P., and Costa, E. B. (2022). *Programação Paralela e Distribuída: com MPI, OpenMP e OpenACC para computação de alto desempenho*. Casa do Código.

[Stallings 2015] Stallings, W. (2015). *Computer Organization and Architecture*. Pearson, Upper Saddle River, NJ, 10 edition.

[Wilkinson and Allen 2005] Wilkinson, B. and Allen, M. (2005). *Parallel programming - techniques and applications using networked workstations and parallel computers (2. ed.)*. Pearson Education.