Computação de Alto Desempenho (HPC)

A Computação de Alto Desempenho (HPC) é a utilização de **recursos computacionais poderosos para resolver problemas complexos**. Abrange não apenas a arquitetura de hardware, mas também as ferramentas de software, plataformas de programação e paradigmas de programação paralela associados. O seu objetivo principal é **oferecer alta taxa de transferência (throughput) e eficiência para tarefas computacionalmente intensivas**.

Importância e Aplicações:
A HPC é crucial para resolver problemas de larga escala em diversas áreas, incluindo:

- Cosmologia, astrofísica e astronomia.
- Química computacional, biologia e engenharia.
- Ciência da computação, IA/Aprendizagem de Máquina (ML).
- Ciências da Terra e materiais.
- Previsão meteorológica.
- Ciência e tecnologia de informação geográfica.
- Segurança global e fusão nuclear.

Permite a inovação e impulsiona avanços em hardware e software.

Evolução e Tendências:

- A HPC inicial era baseada em sistemas de processador único e processadores vetoriais.
- Houve uma **mudança de paradigma com o surgimento de arquiteturas de computação heterogéneas**, combinando CPUs e GPUs. As GPUs são altamente eficazes para certos tipos de computação, especialmente aquelas com altos níveis de paralelismo de dados.
- Atualmente, os sistemas HPC incluem **computação exascale** (10^18 cálculos por segundo), integração de IA/ML, e HPC baseada na nuvem para acessibilidade. A ênfase é na **eficiência energética e escalabilidade**.
- Os principais supercomputadores em novembro de 2024 incluem o El Capitan, Frontier e Aurora nos Estados Unidos, e o Supercomputer Fugaku no Japão.

PROF

Arquitetura de Máquinas Paralelas:

- Os sistemas HPC modernos operam principalmente como **arquiteturas paralelas de memória distribuída**, organizadas em clusters extensos de nós de processamento (PN) interligados.
- Um **nó de processamento** possui CPUs multi-core e GPUs many-core, permitindo a computação heterogénea. A CPU gere o ambiente, código e dados antes de descarregar tarefas computacionalmente pesadas para a GPU. As GPUs complementam, não substituem, as CPUs.
- **Topologias de interconexão** são cruciais para a comunicação entre nós. Exemplos incluem:
    - **Mesh Topology:** Nós dispostos numa estrutura de grade, ideal para comunicação localizada.
    - **Torus Topology:** Extensão da mesh com ligações nas extremidades, reduzindo a distância de comunicação.

    - **Hypercube Topology:** Nós em uma estrutura de cubo binário, altamente escalável e eficiente para dados intensivos.
    - **Tree Topology:** Hierárquica, simplifica o encaminhamento e eficiente para comunicação coletiva.
    - **Fully Connected Topology:** Cada nó diretamente ligado a todos os outros, máximo throughput para pequenos sistemas, impraticável para grandes.
    - **Fat-Tree Topology:** Variação da árvore com largura de banda adicional nos níveis superiores para evitar gargalos, equilibra escalabilidade e largura de banda.
    - **Star Topology:** Todos os nós ligados a um hub central, simples para pequenos sistemas.
    - **Ring Topology:** Nós ligados em círculo, simples e económico para sistemas de tamanho moderado.
- Mesh, torus e fat-tree são prevalentes em supercomputadores modernos.

Decomposição Paralela:

- Tipicamente **orientada a dados**.
- A granularidade dos algoritmos paralelos é classificada em:
    - **Fine-grained (Granularidade Fina):** Operações em múltiplos dados ao nível da variável (SIMD - Single Instruction, Multiple Data).
    - **Medium-grained (Granularidade Média):** Paralelismo ao nível do thread (MIMD - Multiple Instruction, Multiple Data, com memória partilhada).
    - **Coarse-grained (Granularidade Grosseira):** Paralelismo ao nível do processo (MIMD, com memória distribuída).
- Algoritmos HPC frequentemente combinam todos os três níveis para desempenho ótimo.

Lei de Amdahl:

- A Lei de Amdahl (1967) estima o ganho de desempenho, mostrando que o **speedup é limitado pela fração de tempo de execução afetada pelo modo mais rápido**.
- A fórmula é $S_{overall} = 1 / ((1 - P) + P/N)$, onde P é a fração paralelizável e N é o fator de speedup.
- Em sistemas de memória distribuída, a **sobrecarga de comunicação (C)** limita ainda mais o speedup, levando a retornos decrescentes: $S_{overall} = 1 / ((1 - P) + C + P/N)$.
- A lei define um **limite para o speedup**, pois a fração não paralela restringe o desempenho.

PROF

Ferramentas para Programação Paralela:
As aplicações paralelas são escritas em C/C++ usando:

- `std::thread`**:** Para multithreading em memória partilhada (paralelismo de granularidade média).
- **MPI (Message Passing Interface):** Para multiprocessamento em memória distribuída (paralelismo de granularidade grosseira).
- **CUDA C/C++:** Para paralelismo de granularidade fina em arquiteturas CPU-GPU.

Concorrência

A concorrência refere-se à **capacidade de um sistema executar múltiplas tarefas ao mesmo tempo**, seja em paralelo (execução simultânea) ou de forma intercalada (aparente simultaneidade).

Arquitetura de Computadores (Revisão):

- Um sistema computacional básico é composto por: Processador (CPU), Memória Principal (RAM), Memória de Massa (armazenamento), Dispositivos de E/S (Input/Output) e Interconexão de Sistema.
- A **arquitetura de programa armazenado (von Neumann)** permite que as instruções sejam armazenadas na memória principal junto com os dados.
- O processador alterna continuamente entre os estados de **busca de instrução** e **execução de instrução**.
- O processador contém uma **unidade de controlo**, uma **unidade aritmética/lógica (ALU)**, um **banco de registos** e **buffers de E/S**.
- **Pipelining:** Técnica que divide a execução de uma tarefa em sub-tarefas independentes (estágios de pipeline) que operam simultaneamente em objetos sucessivos. Aumenta o throughput e, idealmente, uma pipeline de N estágios oferece um speedup de N vezes.
- **Instruction-Level Parallelism (ILP):** Paralelismo ao nível da instrução, alcançado por mecanismos como *multiple-issue*, *pipelining*, *branch prediction*, *speculative execution*, *out-of-order execution* e *prefetching*.
- **Hierarquia de Memória:** Inclui registos, caches (L1, L2, L3) e memória principal (RAM), com diferentes latências e tamanhos. A **coerência de cache** é um desafio em processadores multicore quando há cópias do mesmo bloco de memória em caches de diferentes cores, exigindo políticas como *write-through* para garantir consistência.

Programa vs. Processo:

- Um **programa** é uma sequência de instruções que descreve uma tarefa.
- Um **processo** é a execução de um programa, uma instância ativa caracterizada por:
    - Espaço de endereçamento (código e variáveis).
    - Contexto do processador (estado dos registos).
    - Contexto de E/S (dados a serem transferidos).
    - Estado de execução (status do processo no ciclo de execução).
- Um processo pode estar nos estados: **Running** (executando), **Ready-to-run** (aguardando atribuição do processador) ou **Blocked** (aguardando evento externo).
- O **scheduler** (agendador) do sistema operativo gere as transições de estado dos processos.

Processos vs. Threads:

- Um processo possui **propriedade de recursos** (espaço de endereçamento privado, canais de comunicação) e uma **linha de execução** (program counter, registos, stack).
- Uma **thread** (ou processo lightweight - LWP) é uma entidade executável independente dentro do contexto de um único processo.
- **Multithreading** permite a criação de múltiplas threads de execução dentro do mesmo processo, possibilitando a execução concorrente e partilha eficiente de recursos.
- **Vantagens do Multithreading**:
    - Simplifica a decomposição da solução e melhora a modularidade.
    - Melhora a gestão de recursos ao partilhar o espaço de endereçamento e contexto de E/S.

    - Aumenta a eficiência e velocidade de execução, pois as operações de thread (criação, terminação, troca de contexto) são mais leves que as de processo.
    - Permite execução paralela em sistemas SMP (Symmetric Multiprocessing).
- As threads são geralmente associadas à execução de uma função, e uma **estrutura de dados global serve como espaço partilhado**.
- **Tipos de Threads**:
    - **User-Level Threads (ULTs):** Implementadas por uma biblioteca de nível de utilizador sem envolvimento do kernel. Versátil e portátil, mas ineficiente se uma thread bloquear todo o processo.
    - **Kernel-Level Threads (KLTs):** Geridas diretamente pelo kernel. Permitem que outras threads no processo continuem a executar se uma bloquear, e possibilitam a execução paralela em processadores multicore.

Interação de Processos:

- **Processos Independentes:** Não interagem explicitamente, mas competem por recursos.
- **Processos Cooperantes:** Partilham informação ou comunicam explicitamente. Podem usar **memória partilhada** ou **comunicação inter-processos (IPC)** via passagem de mensagens (pipes, filas de mensagens, sockets, buffers de memória partilhada).

Região Crítica e Exclusão Mútua:

- A **região crítica** é o código de acesso a um recurso ou região partilhada que deve ser executado de forma a **prevenir condições de corrida** (race conditions) e perda de informação.
- A **exclusão mútua** é o princípio de que **apenas um processo de cada vez** deve ter acesso a um recurso partilhado.

Condições de Corrida (Racing Conditions):
A imposição de exclusão mútua pode levar a:

- **Deadlock / Livelock:** Dois ou mais processos esperam indefinidamente por acesso às suas regiões críticas.
- **Indefinite Postponement (Starvation):** Um ou mais processos são repetidamente impedidos de aceder a uma região crítica devido ao afluxo contínuo de novos processos competidores.
- O objetivo é garantir a **propriedade de vivacidade (liveness)**, ou seja, que os processos progridam e concluam a execução.
- Uma solução para a exclusão mútua deve garantir:
    - Exclusão mútua eficaz.
    - Independência da velocidade de execução ou número de processos.
    - Não interferência de processos externos.
    - Ausência de adiamento indefinido (starvation freedom).
    - Tempo de execução limitado dentro da região crítica.

Recursos e Deadlock:

- **Recursos** são componentes que um processo requer para execução. Classificam-se em:
    - **Físicos:** Hardware (processadores, memória, E/S).

    - **Lógicos:** Estruturas geridas pelo SO ou aplicações (tabelas de controlo de processos, canais IPC, estruturas de dados partilhadas).
- **Tipos de Alocação de Recursos**:
    - **Recursos Preemptíveis:** Podem ser reassignados à força (ex: processadores, regiões de memória principal).
    - **Recursos Não Preemptíveis:** Não podem ser reassignados à força sem causar erros (ex: impressoras, estruturas de dados partilhadas).
- **Condições Necessárias para Deadlock** (todas devem estar presentes):
    1. **Exclusão Mútua:** Recurso atribuído exclusivamente a um processo.
    2. **Manter e Esperar (Hold and Wait):** Um processo mantém recursos previamente alocados enquanto espera por novos.
    3. **Não Preempção (No Preemption):** Recurso alocado não pode ser tirado à força.
    4. **Espera Circular (Circular Wait):** Cadeia circular de processos onde cada um espera por um recurso detido pelo próximo.
- **Prevenção de Deadlock:** Eliminar pelo menos uma das condições necessárias. Geralmente foca-se em negar "manter e esperar", "não preempção" ou "espera circular", pois a exclusão mútua é essencial para recursos não preemptíveis.
    - **Negar "manter e esperar":** Processo deve requisitar todos os recursos de uma vez. Pode levar a starvation se não houver políticas justas.
    - **Negar "não preempção":** Se um processo não conseguir todos os recursos, deve libertar os que tem e recomeçar. Pode levar a starvation.
    - **Negar "espera circular":** Estabelecer uma ordem linear estrita para os recursos e exigir que as requisições sigam essa ordem. Pode levar a starvation.

Thread Pools (Conjuntos de Threads):

- Um **thread pool é uma coleção de threads de trabalho que gerem eficientemente a execução de tarefas**. Em vez de criar e destruir threads frequentemente, um conjunto fixo é reutilizado.
- **Vantagens**:
    - Reduz a sobrecarga de criação e destruição de threads.
    - Agenda e executa tarefas de forma eficiente.
    - Evita a sobrecarga do sistema limitando o número de threads ativas.

PROF

- **Funcionamento**:
    - Componentes: **Worker Threads** (threads persistentes à espera de tarefas), **Task Queue** (fila onde as tarefas são armazenadas), **Thread Manager** (atribui tarefas às threads), e **Synchronization Primitives** (mutexes e variáveis de condição).
    - Um ciclo principal envolve tarefas submetidas à fila, um worker ocioso escolhe uma tarefa, executa-a, e depois retorna ao pool.
- **Uso:** Aplicações de alto desempenho como servidores web, computações paralelas, processamento de Big Data (MapReduce), e motores de jogos.

MPI (Message Passing Interface)

MPI é uma **especificação de biblioteca que suporta o modelo de programação de passagem de mensagens**, onde os dados são explicitamente transferidos entre processos através de operações cooperativas. **Não é uma linguagem de programação**, mas uma API padronizada.

Características Gerais:

- Permite a comunicação inter-processos movendo dados de um espaço de memória para outro.
- Oferece uma base portátil e eficiente para implementações otimizadas, muitas vezes com aceleração de hardware.
- Todas as constantes e funções MPI são prefixadas com `MPI_` (ex: `mpi.h` em C/C++).

Anatomia de um Programa MPI:

- Todo programa MPI deve incluir `mpi.h`.
- Começa sempre com `MPI_Init(&argc, &argv)` e termina com `MPI_Finalize()`.
- `MPI_Comm_rank(MPI_COMM_WORLD, &rank)`: Obtém o **rank** (número de identificação) do processo atual dentro do comunicador.
- `MPI_Comm_size(MPI_COMM_WORLD, &size)`: Obtém o **número total de processos** no comunicador.
- Compilação com `mpic++` e execução com `mpiexec -n <num_processes>`.

Tratamento de Erros:

- Todas as funções MPI retornam um código de erro.
- `MPI_ERRORS_ARE_FATAL` **(padrão):** Aborta todos os processos em caso de erro.
- `MPI_ERRORS_RETURN`**:** Retorna o código de erro ao utilizador, permitindo que o programa continue.

Tipos de Dados:

- MPI define um conjunto de **tipos de dados predefinidos** (ex: `MPI_INT`, `MPI_FLOAT`) que mapeiam para tipos de linguagens como C/C++ para garantir a segurança de tipos e portabilidade.
- **Tipos Especiais:**
  - `MPI_BYTE`**:** Representa um byte bruto de 8 bits.
  - `MPI_PACKED`**:** Usado para empacotamento manual de estruturas de dados complexas e não contíguas.

PROF

Conceito de Comunicador:

- Um **comunicador define um contexto de comunicação** – um "universo" isolado onde a comunicação ocorre. As mensagens são correspondidas apenas dentro do mesmo comunicador, prevenindo interferência.
- Inclui um **grupo de processos**, onde cada um tem um **rank único**.
- `MPI_COMM_WORLD`**:** O comunicador predefinido que inclui todos os processos lançados na tarefa MPI.

Mensagens MPI:

- Cada mensagem MPI inclui um **cabeçalho (envelope)** com metadados para correspondência de mensagens: Contexto de Comunicação, Identificação da Origem, Identificação do Destino e uma **Tag** (rótulo inteiro para distinguir tipos de mensagens).

- O **conteúdo da mensagem (payload)** é descrito por: Tipo de Dados, Referência do Buffer e Contagem (número de elementos).

Comunicação Ponto-a-Ponto:

- Ocorre quando um processo (origem) envia uma mensagem para outro (destino).
- `MPI_Send` **e** `MPI_Recv` são as formas mais básicas e são **operações de bloqueio (blocking)** por padrão:
  - `MPI_Send` bloqueia até que a mensagem seja enviada com segurança (tipicamente, até ser recebida ou armazenada em buffer).
  - `MPI_Recv` bloqueia até que a mensagem seja recebida e o buffer preenchido.
- `MPI_Recv` permite especificar `MPI_ANY_SOURCE` ou `MPI_ANY_TAG` para receber de qualquer origem ou com qualquer tag, respetivamente.
- Um `MPI_Status` struct pode ser usado para obter informações sobre a mensagem recebida (origem, tag, erro).

Comunicação Coletiva:

- Envolve múltiplos processos trabalhando juntos num grupo de comunicação.
- `MPI_Bcast` **(Broadcast):** Um processo raiz envia a mesma mensagem para todos os outros processos no grupo. É uma operação de bloqueio.
- `MPI_Scatter` **(Scatter):** Um processo raiz distribui segmentos distintos de dados para cada processo no grupo. Por padrão, é bloqueante. `MPI_Scatterv` suporta tamanhos de mensagem não uniformes.
- `MPI_Gather` **(Gather):** Cada processo no grupo envia os seus dados para o processo raiz, que os recolhe numa única estrutura. É uma operação de bloqueio. `MPI_Gatherv` é para recolha de dados não uniformes.
- `MPI_Reduce` **(Reduce):** Processos realizam uma computação global elemento a elemento nos seus dados e entregam o resultado final a um processo raiz designado. Usa operadores como `MPI_SUM`, `MPI_MAX`, `MPI_MIN`, etc.. É bloqueante.

Comunicação Não Bloqueante:

PROF

- `MPI_Isend` **e** `MPI_Irecv`**:** Iniciam operações de envio e receção, respetivamente, e **retornam imediatamente sem esperar pela conclusão da comunicação**.
- Para monitorizar e garantir a conclusão, usam-se:
  - `MPI_Wait`**:** Bloqueia o chamador até que a comunicação esteja completa.
  - `MPI_Test`**:** Verifica se a comunicação foi concluída, mas não bloqueia se não o foi.
- Permite **sobrepor computação e comunicação**, melhorando o desempenho.

Sincronização Coletiva:

- `MPI_Barrier`**:** Garante que todos os processos num comunicador esperem uns pelos outros para atingir o mesmo ponto na execução do programa antes de prosseguirem. O último processo a chegar à barreira liberta todos os outros.

Ordenação com MPI (Exemplo de Aplicação):

- Para ordenar grandes conjuntos de dados em paralelo, o MPI é usado em três etapas: **Distribuir o conjunto de dados, Ordenar localmente em cada processo, e Combinar ou redistribuir os pedaços ordenados** para obter um resultado globalmente ordenado.
- Exemplos de técnicas: Merge Sort Paralelo, Sample Sort, Bitonic Sort, Bucket Sort.
- O **Parallel Merge Sort** envolve a distribuição de dados (ex: com `MPI_Scatter`), ordenação local (ex: com `std::sort`), e várias rondas de fusão (`merge`) entre pares de processos, que podem ser seguidas por um `MPI_Gather` opcional no processo raiz.

CUDA (Compute Unified Device Architecture)

CUDA é uma **plataforma de computação paralela de propósito geral e modelo de programação desenvolvida pela NVIDIA**, que aproveita o poder de processamento paralelo das GPUs NVIDIA para resolver eficientemente problemas computacionalmente intensivos.

Visão Geral:

- A plataforma CUDA é acessível via bibliotecas aceleradas por CUDA, diretivas do compilador e APIs, e extensões para linguagens padrão da indústria como C, C++, Fortran, Java e Python.
- **CUDA C** é uma extensão do ANSI C com recursos de linguagem para programação heterogénea e APIs para gerir dispositivos, memória e outras tarefas de GPU.

Níveis de API:

- CUDA fornece dois níveis de API:
  - **CUDA Driver API:** API de baixo nível que oferece controlo granular sobre o comportamento da GPU. Mais difícil de programar, mas oferece mais flexibilidade.
  - **CUDA Runtime API:** API de alto nível construída sobre a Driver API. Mais fácil de usar, com cada função runtime a mapear para uma ou mais operações de nível de driver.
- Não há diferença significativa de desempenho entre as duas APIs; a organização dos threads e o uso da memória impactam mais o desempenho. São mutuamente exclusivas, não podendo ser misturadas no mesmo código.

PROF

Modelo de Compilação:

- O compilador **nvcc** da CUDA separa o **código do host** (CPU, C/C++ padrão) e o **código do dispositivo** (GPU, CUDA C com palavras-chave especiais para kernels) durante a compilação.
- Durante a fase de ligação, as bibliotecas CUDA são ligadas para permitir chamadas de kernel e gestão explícita da GPU. O **CUDA Toolkit** inclui o compilador `nvcc`, bibliotecas matemáticas e ferramentas de depuração/otimização.

Abstrações Principais:

- CUDA expõe três abstrações principais através de extensões de linguagem mínimas:
  - **Hierarquia de grupos de threads**.
  - **Memória partilhada**.
  - **Sincronização de barreira**.
- Estas abstrações suportam **paralelismo de dados e threads de granularidade fina**, aninhados em paralelismo de dados e tarefas de granularidade grosseira.

- A **decomposição do problema** envolve particionar em subproblemas resolvidos por blocos de threads, que são subdivididos e resolvidos cooperativamente por threads dentro de um bloco.
- Os blocos de threads são unidades de execução independentes, permitindo a **execução escalável** em qualquer GPU, independentemente do número de multiprocessadores.

Arquitetura de Hardware da GPU:

- Uma GPU consiste num **array de Streaming Multiprocessors (SMs)**.
- Internamente, são organizadas como uma **topologia MIMD de processadores SIMD**.
- Um programa CUDA multithreaded é dividido em **blocos de threads**, que são executados independentemente.
- GPUs com mais multiprocessadores podem executar mais blocos **concorrentemente**, levando a uma execução mais rápida sem alterações no código.

Estrutura de um Programa CUDA:

1. Alocar memória na GPU.
2. Copiar dados da memória da CPU (host) para a memória da GPU (device).
3. Invocar o kernel CUDA para realizar o cálculo.
4. Copiar os resultados de volta da memória da GPU para a memória da CPU.
5. Libertar a memória da GPU.

Modelo de Programação CUDA:

- **Kernels e Modelo de Threading:**
  - **Kernels** são funções especiais executadas concorrentemente por `N` threads CUDA.
  - Definidos com o qualificador `__global__`.
  - Lançados com uma **configuração de execução** usando a sintaxe `<<<gridDim, blockDim>>>`, que define uma **grid de blocos de threads**.
  - Cada thread tem um **ID único** dentro do seu bloco (`threadIdx`), possui os seus registos e memória privada, e é responsável por calcular e produzir resultados.
- **Blocos de Threads e Indexação:**
  - `threadIdx` é um vetor de 3 componentes, permitindo que as threads sejam indexadas em 1D, 2D ou 3D. Ideal para cálculos sobre vetores, matrizes ou volumes.
  - Restrições de hardware: Um bloco de threads pode ter até 1024 threads (GPUs atuais). Todas as threads num bloco residem no mesmo SM e partilham recursos de memória (registos, memória partilhada) desse SM.
  - A escalabilidade é obtida lançando um kernel com múltiplos blocos: `total de threads = threads por bloco × número de blocos`.
- **Grid de Blocos de Threads:**
  - Organizados numa grid 1D, 2D ou 3D.
  - O número de blocos por grid é determinado pelo tamanho do conjunto de dados ou pelo número de processadores GPU disponíveis.
  - Dentro de um kernel, cada bloco é identificado por `blockIdx` (índice 1D, 2D ou 3D) e o tamanho do bloco é dado por `blockDim`.
- **Independência e Sincronização de Blocos de Threads:**
  - Os blocos de threads devem ser executados independentemente (podem ser agendados em qualquer ordem, em paralelo ou em série), o que permite a escalabilidade.

PROF

    - As threads dentro de um bloco podem cooperar através de **memória partilhada** e **sincronização de barreira** usando `__syncthreads()`, que garante que todas as threads no bloco atinjam este ponto antes de prosseguir.
- **Warps e Execução:**
    - **Warps** são a unidade de execução real na GPU. Um warp consiste em **32 threads** agrupadas pelo agendador do GPU.
    - Um warp executa instruções em modo **SIMD (Single Instruction, Multiple Data)**, também conhecido como **SIMT (Single Instruction, Multiple Threads)**.
    - Execução em *lock-step*: Todas as threads num warp executam a mesma instrução concorrentemente em dados diferentes. Se as threads divergem (branch divergence), o warp serializa esses caminhos, reduzindo a eficiência.
    - O agendador de warps intercala warps prontos para executar para esconder a latência. Usar múltiplos de 32 threads por bloco é frequentemente ideal para evitar warps parciais.

Modelo de Memória:

- As threads CUDA acedem a múltiplos espaços de memória:
    - **Local memory:** Privada a cada thread.
    - **Shared memory:** Acessível por todas as threads num bloco, com tempo de vida do bloco.
    - **Global memory:** Acessível por todas as threads em todos os blocos.
    - **Constant memory:** Apenas de leitura, acessível por todas as threads.
    - **Texture memory:** Apenas de leitura, suporta endereçamento especial e filtragem de dados.
- A memória global, constante e de textura são otimizadas para diferentes padrões de uso e são persistentes entre lançamentos de kernel na mesma aplicação.
- **Memória Host vs. Device:** CUDA assume espaços de memória separados para CPU (host) e GPU (device). Requer gestão de memória explícita (alocação/dealocação de memória do dispositivo, transferências de dados entre host e device) via CUDA Runtime API.
- **Unified Memory (Memória Gerida):** Fornece um espaço de memória único e coerente partilhado entre CPU e GPU, permitindo migração automática de dados, sobreposição da memória do dispositivo e programação simplificada (não há necessidade de cópia manual de memória).

PROF

Qualificadores de Tipo de Função CUDA:

- `__global__`**:** Executado no Device, Chamável do Host. Usado para kernels.
- `__device__`**:** Executado no Device, Chamável do Device. Para funções apenas do dispositivo.
- `__host__`**:** Executado no Host, Chamável do Host. Opcional; padrão para código do host.
- `__device__` e `__host__` podem ser usados em conjunto para compilar uma função para uso tanto no host quanto no device.

Restrições de Kernel:

- Devem aceder apenas à memória do dispositivo.
- Devem ter tipo de retorno `void`.
- Não suportam listas de argumentos variáveis, variáveis estáticas ou ponteiros de função.
- Os lançamentos de kernel são **assíncronos** (o controlo retorna ao host antes da conclusão do kernel).