Chapter 1

Pablo Alessandro Santos Hugen

Abstract

This meta-paper describes the style to be used in articles and short papers for SBC conferences. For papers in English, you should add just an abstract and for the papers in Portuguese, we also ask for an abstract in Portuguese ("resumo"). In both cases, abstracts should not have more than 10 lines and must be in the first page of the paper.

Resumo

Este meta-artigo descreve o estilo a ser usado na confecção de artigos e resumos de artigos para publicação nos anais das conferências organizadas pela SBC. É solicitada a escrita de resumo e abstract apenas para os artigos escritos em português. Artigos em inglês, deverão possuir apenas abstract. Nos dois casos, o autor deve tomar cuidado para que o resumo (e o abstract) não ultrapassem 10 linhas cada, sendo que ambos devem estar na primeira página do artigo.

1.1. Introdução

Nos últimos anos, a Computação de Alto Desempenho (*HPC – High-Performance Computing*) emergiu como uma ferramenta fundamental para resolver problemas em diversos campos, desde a simulação de fenômenos físicos [Hong et al., 2022] até a sua aplicação em vários subcampos da Inteligencia artificial, como a análise de dados e a aprendizagem de máquina Elsebakhi et al. [2015]. Paralelamente, **?** reitera que as Unidades de Processamento Gráfico de Proposito Geral (*GPGPUs, General Purpose Graphics Processing Units*) têm desempenhado um papel cada vez mais crítico nesta arena, oferecendo capacidades de processamento massivamente paralelo. Por conseguinte, a complexidade de problemas modernos estão cada vez mais expandindo os limites da computação de alto desempenho. Como exemplo disso, temos o recente uso da aceleração de múltiplas *GPGPUs* no treinamento de Modelos de Linguagem Massivos (*LLM, Large Language*

Models), como pode ser visto no trabalho de narayanan2021efficient, que utiliza técnicas de paralelismo de dados a fim de escalar o treinamento de modelos desse tipo para *clusters* de *GPUs*.

Consoante a isso, a utilização de técnicas refinadas de *HPC* e a utilização de *GPUs* para processamento vem se tornado proeminente no campo de simulações computacionais compartimentais multiagente [Kabiri Chimeh et al., 2019]. Com isso, Vynnycky2010 citam que estas simulações, que utilizam os chamados Modelos Baseados em Agentes (*ABMs*), são essenciais para modelar dinâmicas complexas, como a propagação de patógenos em populações, permitindo análises detalhadas de cenários epidemiológicos. Eles fazem uso de uma especificação individualizada dos agente – que são categorizadas em compartimentos, geralmente com base em seu estado de saúde em relação a uma dada doença Alves and Gagliardi [2006] – e uma análise granular das interações espaço-temporais, incluindo as transições de estado desses agentes, possibilitando uma compreensão mais profunda sobre a disseminação de epidemias e a eficácia de intervenções de saúde pública, incorporando a natureza estocástica dos processos epidêmicos Russel and Norvig [2013].

Além disso, a capacidade de processamento paralelo das *GPUs* proporciona uma melhora significativa no tempo de execução desses modelos, tornando a simulação de grandes populações viável. Trabalhos como o de [Holvenstot et al., 2014] e o de aaby2010efficient mostram a importância do ajuste de técnicas de *HPC* e *Multi-GPGPU* no que tange a simulações *Multiagente* eficientes computacionalmente. Entretanto, a medida que o número de agentes e a tamanho do ambiente simulado aumenta ainda mais, **sofremos do problema da escala**, onde o tempo de execução dessas simulações de torna tão inviável, que em um contexto prático são infactíveis. Sendo assim, o uso de técnicas de computação paralela adequadas torna-se imprescindível para garantir a escalabilidade, precisão e a eficiência na modelagem de fenômenos epidemiológicos em larga escala, onde a interação entre milhões de agentes deve ser computacionalmente gerenciável e precisa.

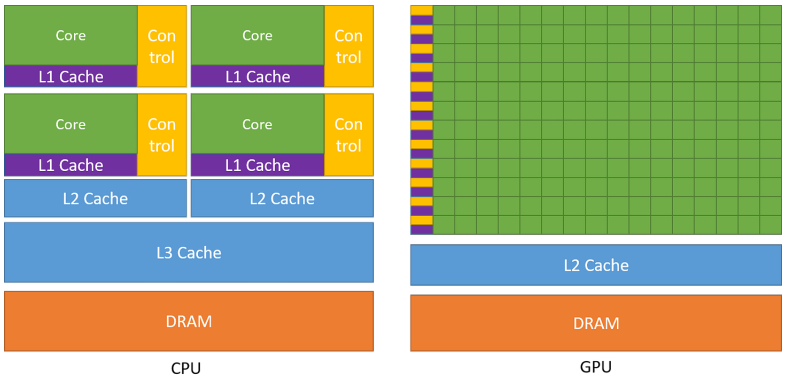
1.2. Modelo de computação das GPUs

A principal diferença entre *CPUs* e *GPUs* reside em seus objetivos de design. As *CPUs* foram projetadas para executar instruções sequenciais. Para aprimorar o desempenho dessa execução sequencial, diversas funcionalidades foram incorporadas ao design das *CPUs* ao longo dos anos [Hennessy and Patterson, 1990]. O foco tem sido a redução da latência na execução de instruções, de modo que as *CPUs* possam processar uma sequência de instruções o mais rápido possível. Como destacado por hennessy2018new, isso inclui características como: pipeline de instruções, execução fora de ordem, execução especulativa e caches multinível.

Por outro lado, as *GPUs* foram desenvolvidas visando níveis massivos de paralelismo e alto *throughput*, ainda que isso signifique uma latência média a alta na execução de instruções [Owens, 2007]. Essa direção no design foi influenciada por sua utilização em jogos eletrônicos, gráficos, computação numérica e, mais recentemente, em aprendizado profundo. Todas essas aplicações exigem a realização de uma grande quantidade de cálculos de álgebra linear e computações numéricas em uma velocidade muito elevada, o que levou a um foco significativo no aprimoramento do *throughput* desses dispositivos

Owens [2007]. Como exemplificado na Figure 1.1, as *CPUs* dedicam uma quantidade significativa de área do chip para recursos que reduzem a latência das instruções, como caches grandes, possuem menos *ULAs* (Unidades Logica e Aritmética) e mais unidades de controle. Em contraste, as *GPUs* utilizam um grande número de *ULAs* para maximizar seu poder de cálculo e taxa de transferência. Eles usam uma quantidade muito pequena da área do chip para caches e unidades de controle, decisão de projeto que aumenta a latência média de instruções nas *GPUs*.

Figure 1.1. Comparativo arquitetural entre *CPU* (esquerda) e *GPU* (direita)



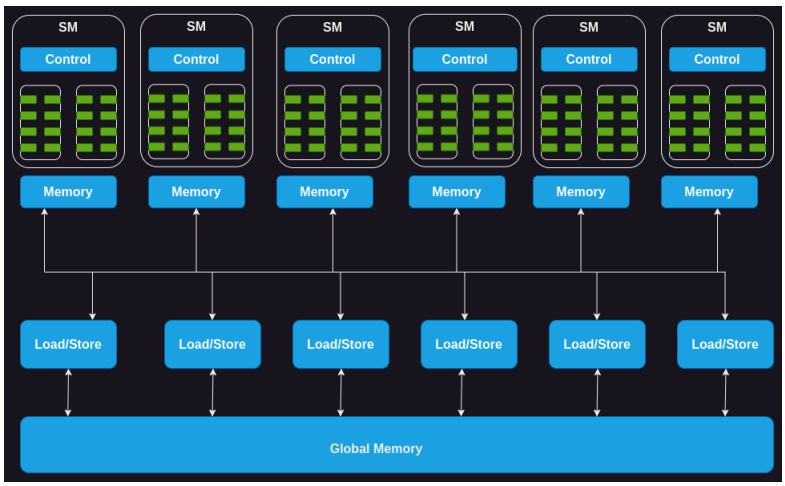
Fonte: [NVIDIA, 2023a]

1.2.0.1. Arquitetura do Modelo de Computação

Portanto, evidencia-se que as *GPUs* são capazes de tolerar altas latências, mantendo um desempenho superior, graças à sua habilidade de combinar um elevado número de *threads*. Embora as instruções individuais possam apresentar altas latências, as *GPUs* têm a habilidade de escalonar as *threads* para execução de forma eficiente, assegurando a utilização constante do poder computacional disponível. Em seu texto, [Owens et al., 2008] reforça que enquanto determinadas *threads* estão em um estado de espera pelo resultado de uma instrução específica, a *GPU* efetua a transição para a execução de outras *threads* que não estão sujeitas a essa espera. Tal estratégia assegura que as unidades de processamento do *GPU* estejam operando em sua máxima capacidade constantemente, resultando em uma elevada taxa de transferência. Na Figure 1.2, é possível observar como as diferentes unidades de processamento dentro de uma *GPU* são organizadas e como operam em paralelo.

Internamente, a Unidade de Processamento Gráfico é composta por um conjunto de Multiprocessadores de Fluxo (*SMs, Streaming Multiprocessors*) [Owens, 2007], com cada *SM* possuindo uma quantidade de memória compartilhada. Esses *SMs* são fundamentais para o desempenho paralelo da *GPU*, permitindo o processamento simultâneo massivo. Por sua vez, wittenbrink2011fermi esclarecem que cada *SM* é composto por vários núcleos de processamento. Estes núcleos, frequentemente referidos como *threads*, operam de maneira paralela. Importante destacar que dentro de cada *SM*, todas essas

Figure 1.2. Arquitetura do Modelo de computação das *GPUs*



Fonte: [NVIDIA, 2023a]

threads compartilham memória e outros recursos de processamento, facilitando a execução de tarefas paralelas de forma eficiente.

1.2.0.2. Arquitetura Geral de Memória

Nesse contexto, [Mei and Chu, 2016] citam que as *GPUs* apresentam uma hierarquia complexa de memórias, cada qual com suas funções específicas. Essa hierarquia é crucial para o desempenho desse tipo de acelerador, como ilustrado na Figure 1.3, que demonstra a organização da memória em um único *SM* na *GPU*, com os registradores sendo um componente chave nessa arquitetura. Durante a execução, os registradores reservados a uma *thread* são privados, ou seja, inacessíveis a outras *threads* Mei and Chu [2016]. A próxima camada consiste nos *caches* de constantes, utilizados para armazenar dados constantes requisitados pelo código executado no *SM*. Para otimizar o uso desses caches, os programadores devem declarar explicitamente os objetos como constantes no código, permitindo que a *GPU* os armazene nesses caches.

Figure 1.3. Arquitetura de memoria das *GPUs*

Imagens/Revisaobibliografica/gpu_mem_arch.

Fonte: [NVIDIA, 2023a]

Cada *Streaming Multiprocessor* tem uma memória compartilhada (*i.e. scratch-pad*), que é uma pequena quantidade de memória *SRAM* programável, rápida e de baixa latência, localizada no chip [Owens, 2007]. Essa memória é ideal para ser usada por um grupo de *threads* que estão rodando no *SM*, ajudando a diminuir operações de carga desnecessárias da memória global e aumentando a eficiência do desempenho do *kernel*. Além disso, essa memória compartilhada ajuda na sincronização entre *threads* dentro de um bloco.

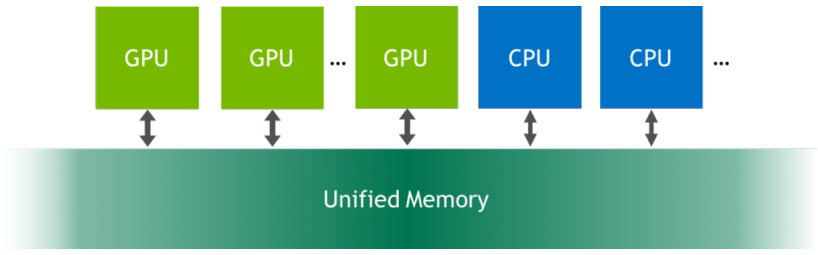
Cada SM também tem um cache L1, usado para armazenar dados que são acessados frequentemente do cache L2. Por sua vez, o cache L2 é compartilhado por todos os SMs [Picchi and Zhang, 2015], e ajuda a reduzir a latência de acesso à memória global. Com isso, mei2016dissecting ressaltam que os processos relacionados aos caches *L1* e *L2* são transparentes para o *SM*, ou seja, são percebidos como parte da memória global. Esse mecanismo é semelhante ao funcionamento dos caches *L1*, *L2* e *L3* em *CPUs*.

Por último, as *GPUs* têm uma memória global externa ao chip, como a memória *DRAM* de alta capacidade e largura de banda encontrada, por exemplo, na mais recente arquitetura *hopper* da *Nvidia* [Choquette, 2023], encontrada em placas como a *H100*. Apesar da alta latência devido à distância dos SMs, a presença de várias camadas de memória adicionais no chip e um grande número de unidades de computação contribuem para minimizar essa latência.

1.2.0.3. Arquitetura de Memória Unificada

A Memória Unificada é um espaço único de endereçamento de memória acessível de qualquer processador no sistema. Esta tecnologia permite que aplicações aloquem dados que podem ser lidos ou escritos por códigos executados tanto em *CPUs* quanto em *GPUs* [NVIDIA, 2023b]. Complementarmente, chien2019performance enfatizam que quando uma rotina acessa dados alocados dessa maneira, o driver *CUDA* e o *hardware* cuidam da migração das páginas de memória para a memória do processador que está acessando de maneira automática.

Figure 1.4. Arquitetura de Memória Unificada CUDA



Fonte: [NVIDIA, 2023a]

Essa paginação sob demanda pode ser particularmente benéfica para aplicações que acessam dados com um padrão disperso. Em algumas aplicações, não se sabe de antemão quais endereços de memória específicos um determinado processador irá acessar. Sem suporte a falhas de página no *hardware*, as aplicações só podem pré-carregar arrays inteiros ou sofrer o custo de acessos de alta latência fora do dispositivo (também conhecido como *"Zero Copy"*). Mas as falhas de página significam que apenas as páginas que o *kernel* acessa precisam ser migradas.

1.3. Modelos de programação paralela em GPUs disponíveis no C++

1.3.1. Compute Shaders

1.3.2. CUDA/HIP

1.3.3. OpenCL

1.3.4. OpenMP

1.3.5. SYCL

1.3.6. Bibliotecas de alto nível

1.3.7. stdexec

1.4. stdexec: Modelo assíncrono/heterogêneo de execução de tarefas em GPUs

1.5. References

Bibliographic references must be unambiguous and uniform. We recommend giving the author names references in brackets, e.g. [Knuth 1984], [Kernighan and Ritchie 1990]; or dates in parentheses, e.g. Knuth (1984), Sederberg and Zundel (1989,1990).

References

Domingos Alves and HF Gagliardi. Técnicas de modelagem de processos epidêmicos e evolucion á rios. *Notas em Matemática Aplicada*, 26:92, 2006.

Jack Choquette. Nvidia hopper h100 gpu: Scaling performance. *IEEE Micro*, 2023.

Emad Elsebakhi, Frank Lee, Eric Schendel, Anwar Haque, Nagarajan Kathireason, Tushar Pathare, Najeeb Syed, and Rashid Al-Ali. Large-scale machine learning based on functional networks for biomedical big data with high performance computing platforms. *Journal of Computational Science*, 11:69–81, 2015.

John L Hennessy and David A Patterson. Computer architecture. 1990.

Peter Holvenstot, Diana Prieto, and Elise de Doncker. Gpgpu parallelization of self-calibrating agent-based influenza outbreak simulation. In *2014 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–6. IEEE, 2014.

Sumin Hong, Ganghee Jang, and Won-Ki Jeong. Mg-fim: A multi-gpu fast iterative method using adaptive domain decomposition. *SIAM Journal on Scientific Computing*, 44(1):C54–C76, 2022.

Mozhgan Kabiri Chimeh, Peter Heywood, Marzio Pennisi, Francesco Pappalardo, and Paul Richmond. Parallelisation strategies for agent based simulation of immune systems. *BMC bioinformatics*, 20:1–14, 2019.

Xinxin Mei and Xiaowen Chu. Dissecting gpu memory hierarchy through microbenchmarking. *IEEE Transactions on Parallel and Distributed Systems*, 28(1):72–86, 2016.

NVIDIA. CUDA C++ programming guide. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html, 2023a. [Acessado em 19/01/2024].

NVIDIA. CUDA C++ unified memory guide. https://developer.nvidia.com/blog/unified-memory-cuda-beginners/, 2023b. [Acessado em 19/01/2024].

John Owens. Gpu architecture overview. In *ACM SIGGRAPH 2007 courses*, pages 2–es. 2007.

John D Owens, Mike Houston, David Luebke, Simon Green, John E Stone, and James C Phillips. Gpu computing. *Proceedings of the IEEE*, 96(5):879–899, 2008.

John Picchi and Wei Zhang. Impact of l2 cache locking on gpu performance. In *SoutheastCon 2015*, pages 1–4. IEEE, 2015.

Stuart J Russel and Peter Norvig. Inteligência artificial. [sl], 2013.