Capítulo 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 a resolução de problemas em diversos campos, desde a simulação de fenômenos físicos [Hong et al., 2022] até sua aplicação em vários subcampos da Inteligência Artificial, como a análise de dados e a aprendizagem de máquina [Elsebakhi et al., 2015]. Paralelamente, Barlas [2014] reitera que as Unidades de Processamento Gráfico de Propósito Geral (*GPGPUs, General Purpose Graphics Processing Units*) têm desempenhado um papel cada vez mais crítico nessa área, oferecendo capacidades de processamento massivamente paralelo. Consequentemente, a complexidade dos problemas modernos tem expandido continuamente os limites da computação de alto desempenho. Como exemplo, destaca-se o uso recente da aceleração de múltiplas *GPGPUs* no treinamento de Modelos de Linguagem Massivos (*LLMs, Large Language Models*), que utiliza técnicas de paralelismo de dados para escalar o treinamento desses modelos em *clusters* de *GPUs*.

Nesse contexto, a linguagem de programação C++ desempenha um papel central no desenvolvimento de aplicações para HPC, oferecendo um equilíbrio entre abstração de alto nível e controle eficiente sobre os recursos de hardware. Especialmente no caso das *GPUs*, a linguagem serviu como base para a criação de todo o ecossistema de programação paralela para esses aceleradores. Assim, o objetivo deste minicurso é apresentar os principais modelos de programação paralela em *GPUs* disponíveis para C++, com destaque para o modelo assíncrono e heterogêneo **stdexec**. A Seção 1.2 explica brevemente o modelo de computação das *GPUs*; a Seção 1.3 enumera as alternativas em C++ para programação desses aceleradores; e, por fim, a Seção 1.4 concentra-se no modelo mais recente, o *stdexec*.

1.2. Modelo de computação das GPUs

Um pré requisito básico para a escrita de algoritmos e estruturas de dados eficientes na programação para *GPUs* é entender a sua arquitetura, e como ela difere do modelo tradicional. 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 de máquina. 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 Figura 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*.

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]

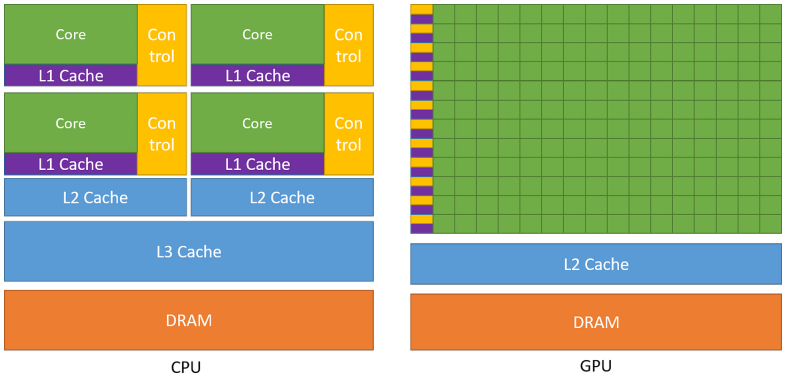

Figura 1.1. Comparativo arquitetural entre *CPU* (esquerda) e *GPU* (direita). Fonte: [NVIDIA, 2023a]

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 Figura 1.2, é possível observar como as diferentes unidades de processamento dentro de uma *GPU* são organizadas e como operam em paralelo.

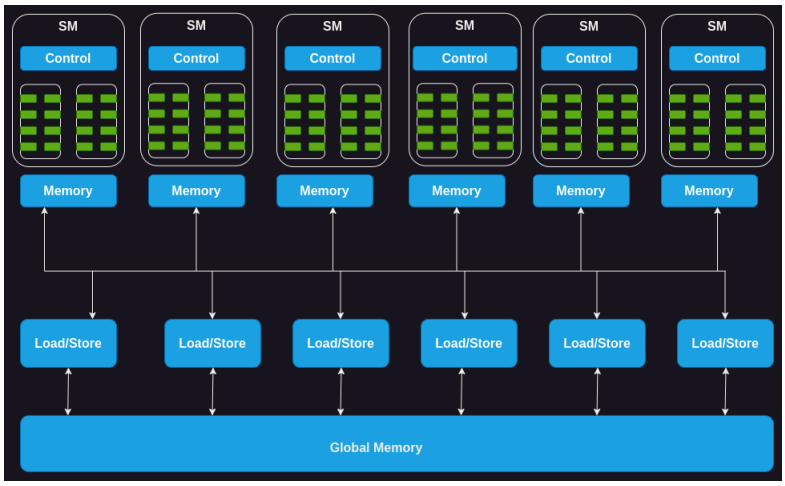

Figura 1.2. Arquitetura do Modelo de computação das *GPUs*. Fonte: [NVIDIA, 2023a].

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 *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

As *GPUs* possuem uma hierarquia complexa de memórias, essencial para seu desempenho [Mei and Chu, 2016]. A Figure 1.3 ilustra a organização da memória em um *SM*, destacando os registradores, que são privados a cada *thread*. Os *caches* de constantes armazenam dados constantes, exigindo que programadores declarem explicitamente objetos como constantes para otimização. Cada *SM* tem uma memória compartilhada (*scratchpad*) de baixa latência, ideal para reduzir acessos à memória global e sincronizar *threads* [Owens, 2007]. O cache L1 armazena dados frequentemente acessados do cache L2, que é compartilhado entre os *SMs* [Picchi and Zhang, 2015], reduzindo a latência de acesso à memória global. Esses processos são transparentes para o *SM*, semelhante aos caches em *CPUs* [Mei and Chu, 2016]. A memória global externa ao chip, como a *DRAM* de alta largura de banda encontrada na arquitetura *Hopper* da Nvidia [Choquette, 2023], apresenta alta latência. No entanto, as camadas adicionais de memória e o grande número de unidades de cálculo ajudam a minimizar esse impacto.

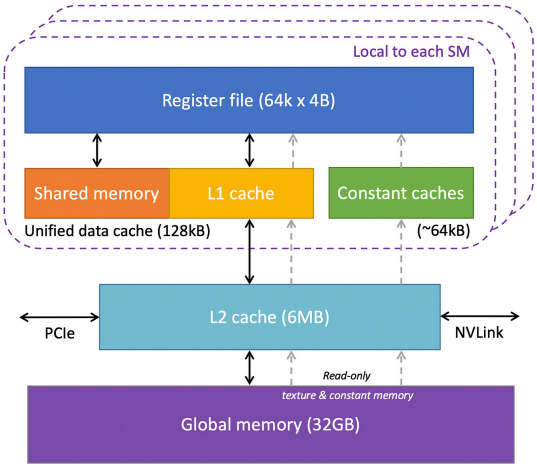

Figura 1.3. Arquitetura de memória das *GPUs*. Fonte: [NVIDIA, 2023a]

A Memória Unificada permite que *CPUs* e *GPUs* acessem um mesmo endereço de memória [NVIDIA, 2023b]. O driver *CUDA* e o *hardware* gerenciam automaticamente a migração das páginas de memória quando necessário chien2019performance. Esse mecanismo é vantajoso para aplicações com padrões de acesso dispersos, evitando a necessidade de pré-carregar grandes arrays ou recorrer a acessos de alta latência fora do dispositivo (*Zero Copy*). Com suporte a falhas de página, apenas as páginas necessárias são migradas, otimizando o desempenho.

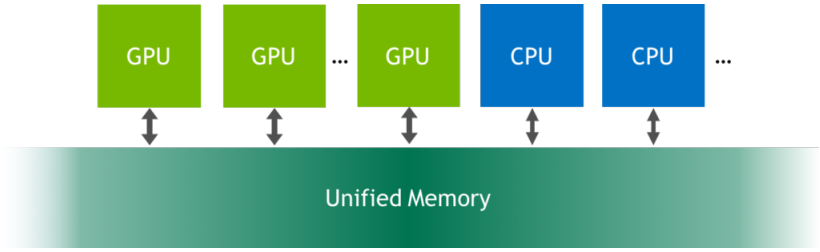

Figura 1.4. Arquitetura de Memória Unificada CUDA. Fonte: [NVIDIA, 2023a]

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

Referências

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

Gerassimos Barlas. *Multicore and GPU Programming: An integrated approach*. Elsevier, 2014.

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.