

Capítulo

1

Pablo Alessandro Santos Huguen

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 essencial para escrever algoritmos e estruturas de dados eficientes em *GPUs* é compreender sua arquitetura e como ela difere do modelo tradicional das *CPUs*. Estas, projetadas para execução sequencial, priorizam a redução da latência por meio de otimizações como pipeline de instruções, execução fora de ordem, especulativa e caches multinível [Hennessy and Patterson, 1990, 2018]. Por outro lado, as *GPUs* foram projetadas para alto *throughput* e paralelismo massivo, mesmo com maior latência na execução de instruções [Owens, 2007]. Esse design, influenciado por aplicações em gráficos, computação numérica e aprendizado de máquina, prioriza cálculos de álgebra linear em alta velocidade. Como visto na Figura 1.1, enquanto *CPUs* dedicam mais área do chip a caches e *UCs* para reduzir a latência, *GPUs* maximizam o poder de cálculo com um grande número de *ULAs*, sacrificando latência para otimizar *throughput* [Owens, 2007].

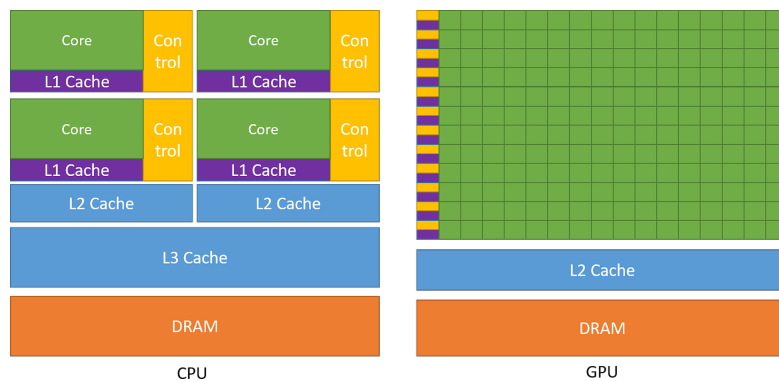


Figura 1.1: Comparativo arquitetural entre CPUs e GPUs. Fonte: [NVIDIA, 2023a]

Assim, as *GPUs* toleram altas latências mantendo desempenho superior ao escalar milhares de *threads* em paralelo. Embora instruções individuais possam ser lentas, a alternância eficiente entre *threads* garante o uso contínuo dos recursos computacionais. Como destacado por Owens et al. [2008], enquanto algumas *threads* aguardam resultados, outras são executadas, maximizando a ocupação das unidades de processamento e o *throughput*. A Figura 1.2 ilustra essa organização e a execução paralela dentro da *GPU*. 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, 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.

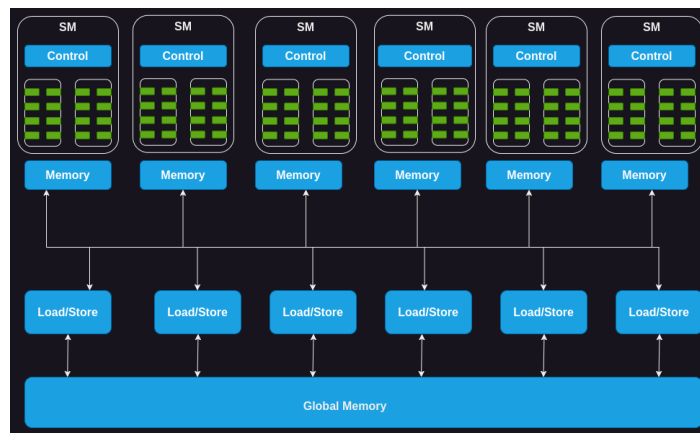


Figura 1.2: Arquitetura do Modelo de computação das *GPUs*. Fonte: [NVIDIA, 2023a].

Esses aceleradores possuem uma hierarquia de memórias complexa e otimizada para desempenho [Mei and Chu, 2016]. A Figura 1.3 ilustra a organização da memória em um *SM*, incluindo registradores privados por *thread* e *caches* de constantes, que exigem declaração explícita para otimização. Cada *SM* possui uma memória compartilhada (*scratchpad*) de baixa latência, útil para reduzir acessos à memória global e sincronizar *threads* [Owens, 2007]. O cache L1 armazena dados acessados com frequência do cache L2, que é compartilhado entre os *SMs* [Picchi and Zhang, 2015]. A memória global externa, como a *DRAM* de alta largura de banda na arquitetura *Hopper* da Nvidia [Choquette, 2023], tem alta latência, mas seu impacto é reduzido pelo escalonamento eficiente das *threads* e pelo uso de caches.

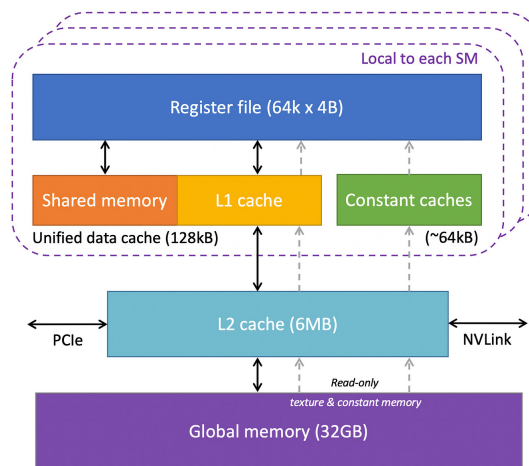


Figura 1.3: Arquitetura de memória das *GPUs*. Fonte: [NVIDIA, 2023a]

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. 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, onde apenas as páginas necessárias são migradas.

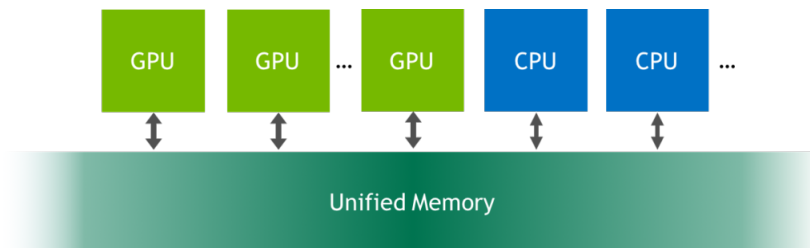


Figura 1.4: Arquitetura de Memória Unificada CUDA. Fonte: [NVIDIA, 2023a]

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

As *GPUs*, inicialmente projetadas para processamento gráfico, evoluíram para se tornar componentes essenciais em diversas aplicações computacionais. Esse avanço acelerado gerou um ecossistema fragmentado, repleto de modelos de programação paralela, compiladores e ferramentas, cada um oferecendo diferentes níveis de abstração e otimização.

Nesse contexto, a linguagem C++ se firmou como a principal escolha para o desenvolvimento em *GPUs*, fornecendo suporte a uma ampla gama de modelos de programação paralela. Sua combinação de flexibilidade e alto desempenho possibilitou a criação de APIs e frameworks que vão desde abordagens de baixo nível, como *Compute Shaders* e *CUDA*, até soluções mais portáteis e genéricas, como *OpenCL*, *OpenMP* e *SYCL*. Essa diversidade reflete a necessidade de equilibrar controle granular sobre o hardware com portabilidade e facilidade de uso, permitindo atender a diferentes domínios computacionais. As próximas seções exploram esses modelos em detalhes, destacando suas características, vantagens e casos de uso.

1.3.1. Compute Shaders

Os *shaders* são pequenos programas executados diretamente na *GPU*, projetados para processar e manipular dados em estágios específicos do *pipeline* gráfico. Os *vertex shaders* transformam as coordenadas de vértices em um espaço tridimensional, enquanto os *fragment shaders* determinam a cor, textura e iluminação de cada pixel antes da renderização final. No entanto, o *pipeline* gráfico tradicional nem sempre é suficiente para atender a demandas computacionais que vão além da renderização, como simulações físicas e processamento de grandes volumes de dados. Para suprir essa necessidade, APIs gráficas modernas introduziram os *Compute Shaders*, que operam fora do *pipeline* e permitem a execução de cálculos arbitrários diretamente na *GPU* usando a mesma API gráfica. A introdução desse tipo de shader possibilitou o uso da *GPU* para tarefas gerais, expandindo suas aplicações para além da renderização gráfica. A Figura 1.5 ilustra a estrutura do *pipeline* gráfico do *OpenGL* e a integração dos *compute shaders* em diferentes estágios.

OpenGL 4.3 Pipelines

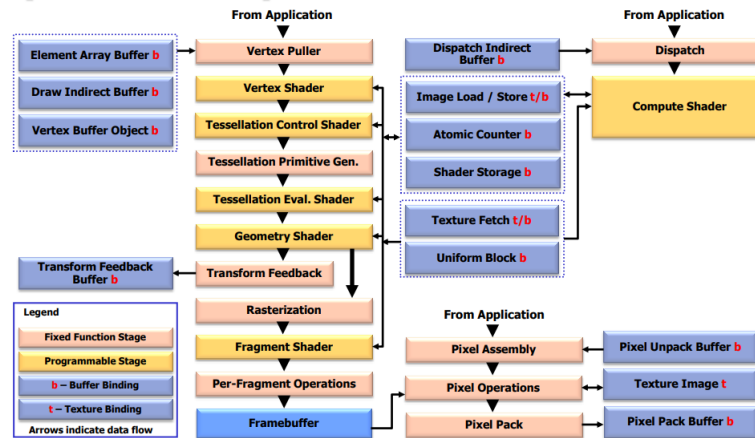


Figura 1.5: Estrutura do *pipeline* gráfico e *compute shaders*. Fonte: [Lichtenbelt, 2012]

No *OpenGL*, os *compute shaders* diferem dos demais por não terem entradas ou saídas fixas, exigindo que busquem e armazenem dados manualmente (em texturas, imagens ou *uniforms*). Eles operam divididos em *work groups*, definidos pelo usuário na execução. Cada grupo contém várias invocações do shader (definidas pelo *local size*) que podem se comunicar entre si, mas não com outros grupos. Como a ordem de processamento é indefinida, não há garantia de sequência fixa. Essa estrutura é ideal para tarefas como processamento de imagens, onde cada grupo lida paralelamente com regiões da imagem. Para executar um *compute shader* em C++, é necessário ativá-lo com `glUseProgram` ou `glBindProgramPipeline` e então despachar a computação com `glDispatchCompute`. Esse comando define quantos *work groups* serão executados nas três dimensões. Também é possível despachar a computação de forma indireta, lendo os valores do número de grupos a partir de um *Buffer Object*, usando `glDispatchComputeIndirect`.

1.3.2. CUDA e HIP

Lançada pela *NVIDIA* em novembro de 2006, *CUDA* (*Compute Unified Device Architecture*) é uma plataforma de computação paralela que permite o uso de suas *GPUs* para processamento geral. Ela fornece uma API baseada em C/C++ para que desenvolvedores possam escrever programas que executem de forma massivamente paralela, aproveitando milhares de núcleos disponíveis nas *GPUs* modernas. Já o *HIP* (*Heterogeneous-Compute Interface for Portability*), é a solução equivalente da *AMD*, que além de permitir a programação de *GPUs* da própria *AMD*, também fornece uma camada de compatibilidade para código *CUDA*, facilitando a portabilidade entre os dois fornecedores de *hardware*.

Essas APIs são baseadas em *kernels* (código executado na *GPU*), que são funções chamadas pelo código *host* (código executado na *CPU*) e executadas *N* vezes em paralelo por *N* diferentes *threads* *CUDA*. Um *kernel* é definido usando o especificador de declaração `__global__`, e o número de *threads* que o executam para uma determinada chamada é especificado por meio de uma sintaxe específica (`<<< ... >>>`). Cada *thread* recebe um *ID* único, acessível dentro do próprio *kernel*.

```

1  #version 460
2
3  layout(local_size_x = 16, local_size_y = 16) in;
4  layout(binding = 0, rgba8) uniform readonly image2D inputImage;
5  layout(binding = 1, rgba8) uniform writeonly image2D outputImage;
6
7  void main() {
8      ivec2 pixelCoords = ivec2(gl_GlobalInvocationID.xy);
9      vec4 pixelColor = imageLoad(inputImage, pixelCoords);
10     vec4 invertedColor = vec4(vec3(1.0) - pixelColor.rgb, pixelColor.a);
11     imageStore(outputImage, pixelCoords, invertedColor);
12 }

```

```

1  GLuint createComputeShader() { /*...*/ }
2
3  auto main() -> int {
4      /*...*/
5      GLuint computeShaderProgram = createComputeShader();
6      /*...*/
7      glUseProgram(computeShaderProgram);
8      glDispatchCompute(16, 16, 1);
9      glMemoryBarrier(GL_SHADER_IMAGE_ACCESS_BARRIER_BIT);
10     /*...*/
11 }

```

Código 1: Exemplo de *compute shader (glsl)* para inverter cores de imagens.

Como discutido anteriormente na Seção 1.2, o conceito de *hierarquia de threads e memória* na arquitetura das *GPUs* é essencial para obter um bom desempenho na execução de *kernels*. No Código 2, podemos observar a organização das *threads* em blocos e sua indexação dentro da *grid* de execução. Cada bloco de *threads* é identificado por um índice (*blockIdx.x* e *blockIdx.y*), enquanto cada *thread* dentro do bloco possui um índice local (*threadIdx.x* e *threadIdx.y*), permitindo calcular sua posição global na grade ao combiná-los com o tamanho do bloco (*blockDim.x* e *blockDim.y*).

O *nvcc* é o compilador *CUDA* proprietário da *NVIDIA* responsável por traduzir código-fonte contendo *kernels* em código executável para a *GPU*. Ele separa o código *host* do código *device*, compilando o código *device* para *PTX* ou *cubin* e modificando o código *host* para gerenciar as chamadas aos *kernels*. A compilação pode ser feita de forma *offline*, gerando código binário antecipadamente, ou *just-in-time*, onde o código *PTX* é compilado pelo driver durante a execução da aplicação, garantindo compatibilidade com novas arquiteturas. O uso do *nvcc* simplifica esse processo, fornecendo uma interface unificada para a compilação e a geração de código otimizado para *GPUs*.

```

1  __global__ void matadd(float A[N][N], float B[N][N],
2  float C[N][N])
3  {
4      int i = blockIdx.x * blockDim.x + threadIdx.x;
5      int j = blockIdx.y * blockDim.y + threadIdx.y;
6      if (i < N && j < N)
7          C[i][j] = A[i][j] + B[i][j];
8  }
9
10 int main()
11 {
12     /*...*/
13     dim3 threadsPerBlock(16, 16);
14     dim3 numBlocks(N / threadsPerBlock.x, N / threadsPerBlock.y);
15     matadd<<<numBlocks, threadsPerBlock>>>(A, B, C);
16     /*...*/
17 }

```

Código 2: Kernel de adição de matrizes em CUDA.

Todo o runtime *CUDA* exposto pelo compilador é baseado em uma *API* de baixo nível escrita em C, que também pode ser acessada pela aplicação. Essa interface oferece um controle adicional ao expor conceitos como *contextos*, semelhantes a processos do sistema operacional – e *módulos*, que funcionam como bibliotecas carregadas dinamicamente no dispositivo. A maioria das aplicações não utiliza essa *API* de baixo nível, pois não necessita dessa quantidade extra de controle, o que deixa o código mais simples.

1.3.3. OpenMP e OpenACC

O OpenMP (*Open Multi-Processing*) é uma *API* para programação paralela em sistemas de memória compartilhada, como *multi-core CPUs*. Por meio de diretivas de compilador (*pragmas*), ele facilita a paralelização de loops e seções de código com mínima modificação no código sequencial, permitindo distribuir o trabalho entre múltiplas *threads*. Além disso, o OpenMP simplifica o acesso à memória compartilhada, um desafio comum em outros ambientes, onde pode levar a *data races* e problemas semelhantes. Versões recentes também introduziram suporte para offloading para GPUs e outros aceleradores, expandindo suas capacidades além das *CPUs*.

Inicialmente desenvolvido pela empresas *Portland Group (PG)*, *Cray* e *NVIDIA*, o OpenACC (*Open Accelerators*), assim como o OpenMP, utiliza diretivas de compilador para paralelizar o código, mas é focado em aceleração com GPUs e outros dispositivos especializados. Com o OpenACC, os desenvolvedores podem facilmente transferir trechos de código e para diferentes tipos de aceleradores, aproveitando as especialidades e poder de computação paralela de cada um deles. Embora também ofereça uma abordagem de alto nível, o OpenACC é especialmente voltado para quem deseja acelerar aplicações sem precisar se aprofundar em conceitos de muito baixo nível.

Apesar de ambas as especificações definirem uma *API* declarativa de alto nível baseada em diretivas de compilação, o OpenACC foi projetado desde o início para plataformas heterogêneas, como GPUs e outros aceleradores, enquanto o OpenMP passou a suportar esses dispositivos apenas a partir da versão 4. Também, o *OpenACC* foca em fornecer computação de alto desempenho de maneira simples e abstraída, especialmente para a comunidade científica. A seguir, o código mostra a soma de arrays paralela utilizando OpenMP e OpenACC.

```
1 template <typename T, size_t N>
2 auto vector_add(const std::array<T, N>& A, const std::array<T, N>& B,
3               std::array<T, N>& C) -> void {
4     #pragma acc parallel loop copyin(A, B) copyout(C)
5     for (auto i = 0ull; i < N; ++i) C[i] = A[i] + B[i];
6 }
```

```
1 template <typename T, size_t N>
2 auto vector_add(const std::array<T, N>& A, const std::array<T, N>& B,
3               std::array<T, N>& C) -> void {
4     #pragma omp target teams distribute parallel for map(to : A, B) map(from : C)
5     for (auto i = 0ull; i < N; ++i) C[i] = A[i] + B[i];
6 }
```

Código 3: Exemplo de adição de arrays paralelo usando *OpenMP* e *OpenACC*.

Ambos os códigos são compilados com compiladores comuns como o *GCC* e o *Clang*, utilizando as flags específicas: `-fopenmp` e `-fopenacc`, respectivamente.

1.3.4. OpenCL

A Open Computing Language (*OpenCL*) é uma especificação aberta para programação paralela heterogênea, permitindo a execução de código em *CPUs*, *GPUs*, *DSPs*, *FPGAs* e outros aceleradores. Desenvolvida para alta performance, a *API* oferece portabilidade entre diferentes arquiteturas de hardware e é amplamente utilizada em aplicações que exigem processamento massivo de dados, como simulações científicas e processamento de imagens. Por meio dessa *API*, é possível delegar trechos do código mais intensivos em computação para dispositivos de aceleração adequados.

1.3.5. SYCL

1.3.6. std::exec

A maturidade do *C++* no ecossistema de *HPC* levou ao desenvolvimento de modelos de programação paralela de alto nível diretamente baseados na linguagem, reduzindo o tempo e o esforço necessários para manter e implantar aplicações, ao mesmo tempo em que garante portabilidade e desempenho com uma única base de código [Deakin and McIntosh-Smith, 2020]. Esses modelos têm como objetivo possibilitar a programação

paralela heterogênea com desempenho próximo ao nativo [Breyer et al., 2022].

O padrão *C++17* introduziu algoritmos paralelos síncronos (bloqueantes) na biblioteca padrão de *C++*, com suporte para *CPUs* multi-core e *GPUs* com desempenho competitivo [Lin et al., 2022]. Esse modelo é inerentemente síncrono, bloqueando a execução do programa até que as tarefas paralelas sejam concluídas, limitando o paralelismo concorrente [Lin et al., 2022]. Para resolver isso, os esforços estão focados no desenvolvimento do modelo de programação paralela assíncrona (não bloqueante) `std::exec` para o próximo padrão *C++26*, facilitando a execução assíncrona flexível e eficiente de tarefas usando abstrações como *senders*, *receivers* e *schedulers* [Dominiak et al., 2024].

```
1  const auto fn1 = [...] (auto i) noexcept { /* ... */ };
2  const auto fn2 = [...] (auto i) noexcept { /* ... */ };
3
4  const auto work = stdexec::when_all(
5      stdexec::just() | exec::on(gpu.get_scheduler(),
6                              stdexec::bulk(N, insert_s_human)),
7      stdexec::just() | exec::on(cpu.get_scheduler(),
8                              stdexec::bulk(N, insert_s_mosquito)))
9  stdexec::sync_wait(work);
```

Código 4: Exemplo de execução assíncrona com `std::exec`.

1.4. `std::exec`: Execução de tarefas assíncronas/heterogêneas no C++

Referências

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

Marcel Breyer, Alexander Van Craen, and Dirk Pflüger. A comparison of sycl, opencl, cuda, and openmp for massively parallel support vector machine classification on multi-vendor hardware. pages 1–12, 2022.

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

Tom Deakin and Simon McIntosh-Smith. Evaluating the performance of hpc-style sycl applications. pages 1–11, 2020.

Michał Dominiak, Georgy Evtushenko, Lewis Baker, Lucian Radu Teodorescu, Lee Howes, Kirk Shoop, Michael Garland, Eric Niebler, and Bryce Adelstein Lelbach. P2300r10: `std::execution`, June 2024. URL <https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2024/p2300r10.html>. ISO/IEC JTC1/SC22/WG21.

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 Hennessy and David Patterson. A new golden age for computer architecture: domain-specific hardware/software co-design, enhanced. 2018.

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

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.

Barthold Lichtenbelt. Latest news and features in opengl. Presented at SIGGRAPH 2012, Los Angeles, CA, USA, August 2012. URL <https://www.khronos.org/assets/uploads/developers/library/2012-siggraph-opengl-bof>.

Wei-Chen Lin, Tom Deakin, and Simon McIntosh-Smith. Evaluating iso c++ parallel algorithms on heterogeneous hpc systems. pages 36–47, 2022.

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. pages 1–4, 2015.