Capítulo 1

Programação de Alto Desempenho em GPUs com C++

Pablo Alessandro Santos Hugen

Abstract

Heterogeneous parallel programming has become an essential approach in the field of HPC*, especially with the growing use of* GPGPUs*. In this context, the* C++ *programming language plays a central role. However, the diversity and complexity of the available models and tools present a significant barrier for beginners in the field. Therefore, the goal of this minicourse is to provide an overview of the main tools and parallel programming models for GPUs in* C++*, highlighting their characteristics, peculiarities, and applicability.*

Resumo

A programação paralela heterogênea tem se consolidado como uma abordagem essencial na área de HPC*, especialmente com o uso crescente de* GPGPUs*. Nesse cenário, a linguagem de programação* C++ *desempenha um papel central. Contudo, a diversidade e a complexidade dos modelos e ferramentas disponíveis representam uma barreira significativa para iniciantes na área. Assim, o objetivo deste minicurso é fornecer uma visão geral das principais ferramentas e modelos de programação paralela para GPUs em* C++*, destacando suas características, peculiaridades e aplicabilidades.*

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 na linguagem. A Seção 1.2 explica brevemente o modelo de computação das *GPUs* e a Seção 1.3 enumera as alternativas em C++ para programação desses aceleradores.

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].

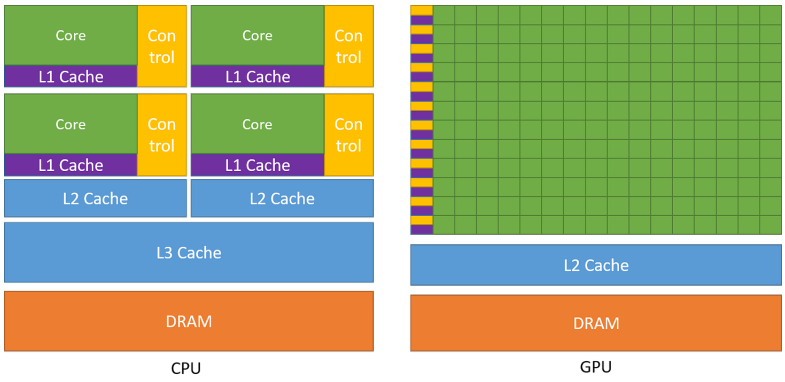

Figura 1.1: Comparativo arquitetural entre *CPUs* e *GPUs*. Fonte: [NVIDIA, 2023a]

Assim, as *GPUs* toleram altas latências mantendo desempenho superior ao escalonar 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.

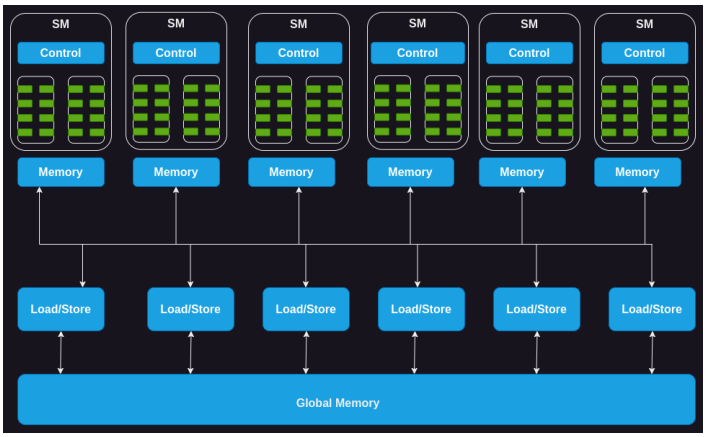

Figura 1.2: Arquitetura do Modelo de computação das *GPUs*. Fonte: [NVIDIA, 2023a].

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.

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.

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.

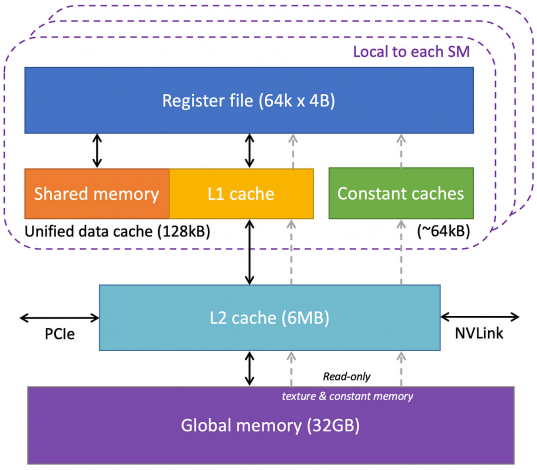

Figura 1.3: Arquitetura de memória das *GPUs*. Fonte: [NVIDIA, 2023a]

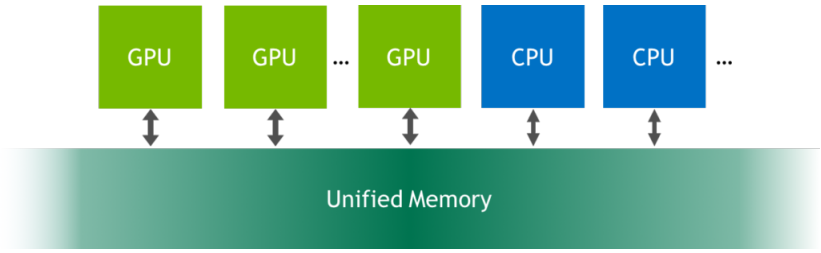

Figura 1.4: Arquitetura de Memória Unificada CUDA. Fonte: [NVIDIA, 2023a]

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 portáveis 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.

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.

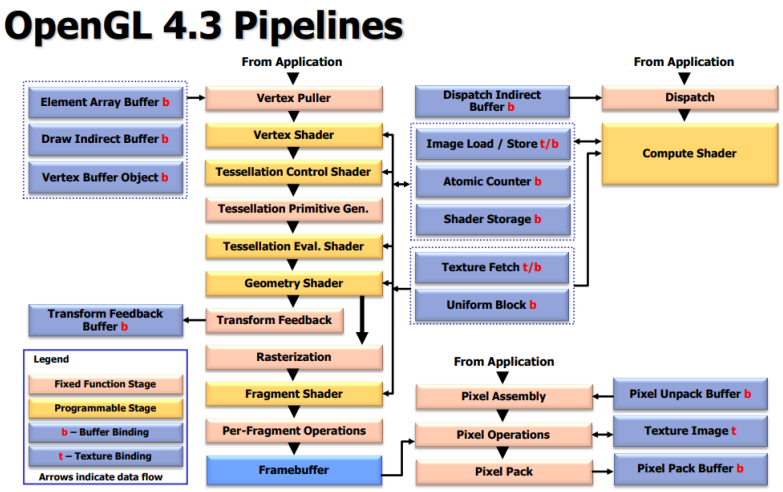

Figura 1.5: Estrutura do *pipeline* gráfico e *compute shaders*. Fonte: [?]

No *OpenGL*, os *compute shaders* diferem dos demais por não terem entradas ou saídas fixas, exigindo que busquem e armazenem dados manualmente (em texturas, images 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 (como demonstrado no Código 1). 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

```glsl
#version 460

layout(local_size_x = 16, local_size_y = 16) in;
layout(binding = 0, rgba8) uniform readonly image2D inputImage;
layout(binding = 1, rgba8) uniform writeonly image2D outputImage;

void main() {
    ivec2 pixelCoords = ivec2(gl_GlobalInvocationID.xy);
    vec4 pixelColor = imageLoad(inputImage, pixelCoords);
    vec4 invertedColor = vec4(vec3(1.0) - pixelColor.rgb, pixelColor.a);
    imageStore(outputImage, pixelCoords, invertedColor);
}
```

```glsl
GLuint createComputeShader() {/*...*/ }

auto main() -> int {
    /*...*/
    GLuint computeShaderProgram = createComputeShader();
    /*...*/
    glUseProgram(computeShaderProgram);
    glDispatchCompute(16, 16, 1);
    glMemoryBarrier(GL_SHADER_IMAGE_ACCESS_BARRIER_BIT);
    /*...*/
}
```

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

um *ID* único, acessível dentro do próprio *kernel*.

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*. O Código 2 exemplifica um *kernel* simples de adição de matrizes em *CUDA*. Nele, 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

```
__global__ void matadd(float A[N][N], float B[N][N],
float C[N][N])
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int j = blockIdx.y * blockDim.y + threadIdx.y;
    if (i < N && j < N)
        C[i][j] = A[i][j] + B[i][j];
}

int main()
{
    /*...*/
    dim3 threadsPerBlock(16, 16);
    dim3 numBlocks(N / threadsPerBlock.x, N / threadsPerBlock.y);
    matadd<<<numBlocks, threadsPerBlock>>>(A, B, C);
    /*...*/
}
```

Código 2: Kernel de adição de matrizes em *CUDA*.

novas arquiteturas. O uso do *nvcc* simplifica esse processo, fornecendo uma interface unificada para a compilação e a geração de código otimizado para *GPUs*.

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 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 3 mostra a soma de arrays paralela utilizando *OpenACC* e *OpenMP*.

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

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

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: `-fopenacc` e `-fopenmp`, respectivamente.

1.3.4. OpenCL

A Open Computing Language (*OpenCL*) é uma especificação aberta para programação paralela heterogênea desenvolvida pela *Khronos Group*, permitindo a execução de código em *CPUs*, *GPUs*, e vários outros aceleradores. Desenvolvida para alto desempenho, a *API* oferece portabilidade e controle de baixo nível entre diferentes arquiteturas de hardware e é amplamente utilizada em aplicações que exigem processamento massivo de dados, como simulações científicas e processamento de imagens. O Código 4 ilustra o controle de baixo nível ao implementar um kernel para a redução de arrays em *OpenCL*.

O objetivo dessa *API* é fornecer uma camada de abstração próxima ao hardware, suficientemente flexível para ser utilizada tanto no desenvolvimento direto quanto como destino de compilação por compiladores e *frameworks* de mais alto nível. A Figura 1.6 ilustra a importância dessa *API* como destino de *frameworks* como o *SYCL* ou até mesmo como alvo de compiladores como o *clang*, por meio da representação intermediária *SPIR*.

```
__kernel void sum(
    __global const double *input,
    __global double *partialSums,
    __local double *localSums) {
  uint local_id = get_local_id(0);
  uint group_size = get_local_size(0);
  localSums[local_id] = input[get_global_id(0)];

  for (uint stride = group_size / 2; stride > 0; stride /= 2) {
    barrier(CLK_LOCAL_MEM_FENCE);

    if (local_id < stride)
        localSums[local_id] += localSums[local_id + stride];
  }

  if (local_id == 0) partialSums[get_group_id(0)] = localSums[0];
}
```

Código 4: Kernel de redução (soma) de arrays para *OpenCL*.

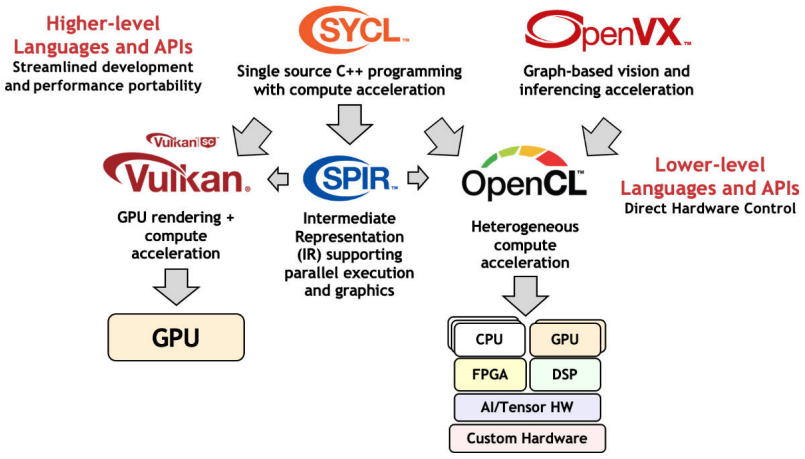

Figura 1.6: Ecossistema do *OpenCL*. Fonte: [?].

1.3.5. SYCL

Também desenvolvida pelo *Khronos Group*, o *SYCL* é uma especificação multi-plataforma focada na abstração de algoritmos em diferentes tipos de aceleradores, com alta expressividade com mínima modificação no código. Delegar parte do código para aceleradores específicos é comum na computação de alto desempenho, mas exige conhecimento de bibliotecas e modelos de programação específicos para cada hardware. O *SYCL* resolve isso ao fornecer uma abstração comum para a programação heterogênea, maximizando a conformidade com o padrão *C++*. Sua popularidade resultou no surgimento de implementações como **Open SYCL**, **neoSYCL**, **triSYCL** e **Intel® oneAPI tools**.

Essas implementações são na maioria das vezes compatíveis, mas variam em recursos devido a diferenças no desenvolvimento e foco arquitetural, porém todas suportam *CPUs* populares. O Código 5 mostra um exemplo de adição de arrays paralelo em *GPU* usando o *SYCL* (implementação *oneAPI*). O uso de *filas* de trabalho e *seletores de dispositivo* deixa o códio genérico ao ponto de, se mudarmos o seletor para `cpu_selector_v` ou `fpga_selector_v` a função executará no respectivo dispositivo (não necessáriamente com o mesmo desempenho, levando em conta as características de cada um).

```
#include <sycl/sycl.hpp>

using namespace sycl;

auto vecadd(queue &q, const int *a, const int *b, int *sum, size_t size)
    -> void {
  range<1> num_items{size};
  auto e = q.parallel_for(
        num_items,
        [=](auto i) { sum[i] = a[i] + b[i]; }
   );
  e.wait();
}

auto main(int argc, char *argv[]) -> int {
  constexpr auto array_size = 10000;
  auto selector = gpu_selector_v;

  sycl::queue q(selector);

  int *a = malloc_shared<int>(array_size, q);
  int *b = malloc_shared<int>(array_size, q);
  int *sum = malloc_shared<int>(array_size, q);

  vecadd(q, a, b, sum, array_size);

  free(a, q);
  free(b, q);
  free(sum, q);
}
```

Código 5: Exemplo de adição de arrays paralelo usando *SYCL*.

O *SYCL* vem evoluindo com a intenção de influenciar a direção do *ISO C++* em torno da computação heterogênea, criando pontos de prova e features que podem ser considerados no contexto da evolução e adoção na linguagem.

1.3.6. std::exec: Execução de tarefas assíncronas/heterogêneas diretamente no C++

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* [**?**]. Um exemplo de execução assíncrona nesse modelo pode ser visto no Código 6.

```
const auto fn1 = [...](auto i) noexcept {/*...*/};
const auto fn2 = [...](auto i) noexcept {/*...*/};

const auto work = stdexec::when_all(
    stdexec::just() | exec::on(gpu.get_scheduler(),
                        stdexec::bulk(N, insert_s_human)),
    stdexec::just() | exec::on(cpu.get_scheduler(),
                        stdexec::bulk(N, insert_s_mosquito)))
stdexec::sync_wait(work);
```

Código 6: Exemplo de execução assíncrona com std::exec.

Nas abordagens tradicionais de programação paralela e concorrente, utiliza-se threads para paralelismo e primitivas de sincronização (como variáveis atômicas e *mutexes*) para evitar interações incorretas, como *race conditions* e *deadlocks*. Podemos pensar nesse modelo como "desestruturado", onde é difícil ter um raciocínio local sobre uma tarefa, pois é preciso considerar o programa como um todo para determinar a sincronização necessária. Com isso, esse *framework* adota o conceito de concorrência estruturada, onde um programa pode ser decomposto em tarefas independentes, executadas concorrentemente e coordenadas de forma assíncrona, abstraindo a programação paralela heterogênea dentro de um mesmo conjunto de abstrações. A Figura 1.7 ilustra os principais conceitos disponíveis no *framework*.

Um *sender* é uma entidade que descreve uma tarefa concorrente. Essa tarefa possui um único ponto de entrada e um ponto de saída, com três possíveis resultados de finalização: sucesso (com retorno de um valor), erro (exceção) ou cancelamento (sem retorno). Os schedulers abstraem o local de execução dessas tarefas, possibilitando que sejam realizadas em *CPUs*, *GPUs* ou outros dispositivos, otimizando o uso dos recursos

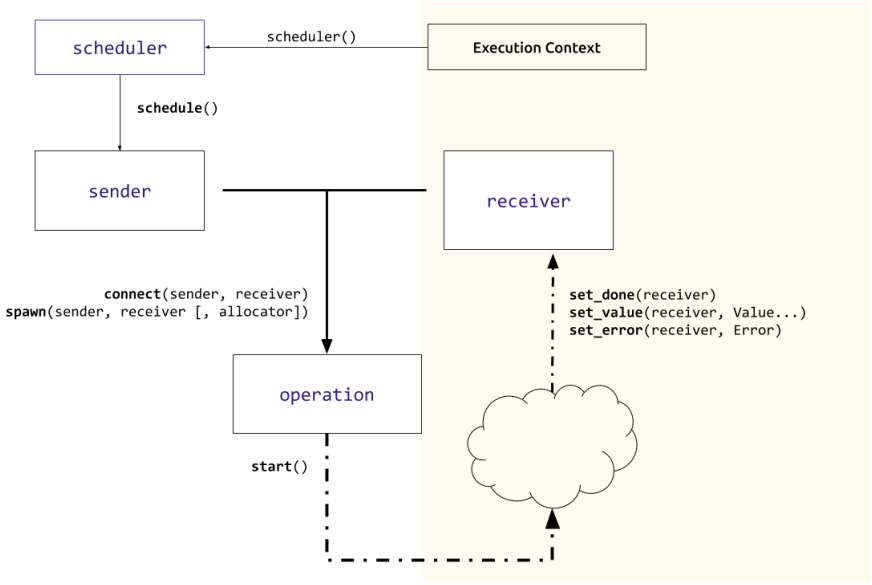

Figura 1.7: Framework *std::exec*. Fonte: [?]

disponíveis. Por fim, os receivers são responsáveis por capturar o resultado de um sender e processá-lo conforme o tipo de conclusão. Com esses blocos de construção, é possível desenvolver programas paralelos e concorrentes de forma estruturada, clara e eficiente. A implementação atual do *framewrk* é fornecida pela *NVIDIA* no compilador *nvc++*, parte do *NVIDIA HPC SDK*. Ela inclui *schedulers* otimizados para as suas *GPUs* proprietárias, inclusive com suporte a *multi-GPUs*.

1.4. Considerações finais

A programação paralela em *GPUs* com *C++* é essencial para o desenvolvimento de aplicações de alto desempenho que aproveitem o máximo do hardware moderno, mas requer o domínio de diversos modelos e ferramentas. Este minicurso apresentou as principais abordagens, desde *APIs* de baixo nível, como *CUDA* e *OpenCL*, até soluções mais portáveis, como *SYCL* e *OpenMP*. Cada modelo tem suas vantagens e desafios, permitindo a escolha mais adequada para cada caso de uso específico. Com esse conhecimento, espera-se que os participantes adquiram uma base sólida sobre as alternativas para explorar e desenvolver aplicações eficientes em *GPUs*. Para aprofundar o entendimento sobre o tema, recomenda-se a leitura do clássico trabalho de Barlas [2014] e das outras referências citadas, além das páginas de comunidade e documentação das ferramentas apresentadas.

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.

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.

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.