

Programação de Alto Desempenho em GPUs com C++

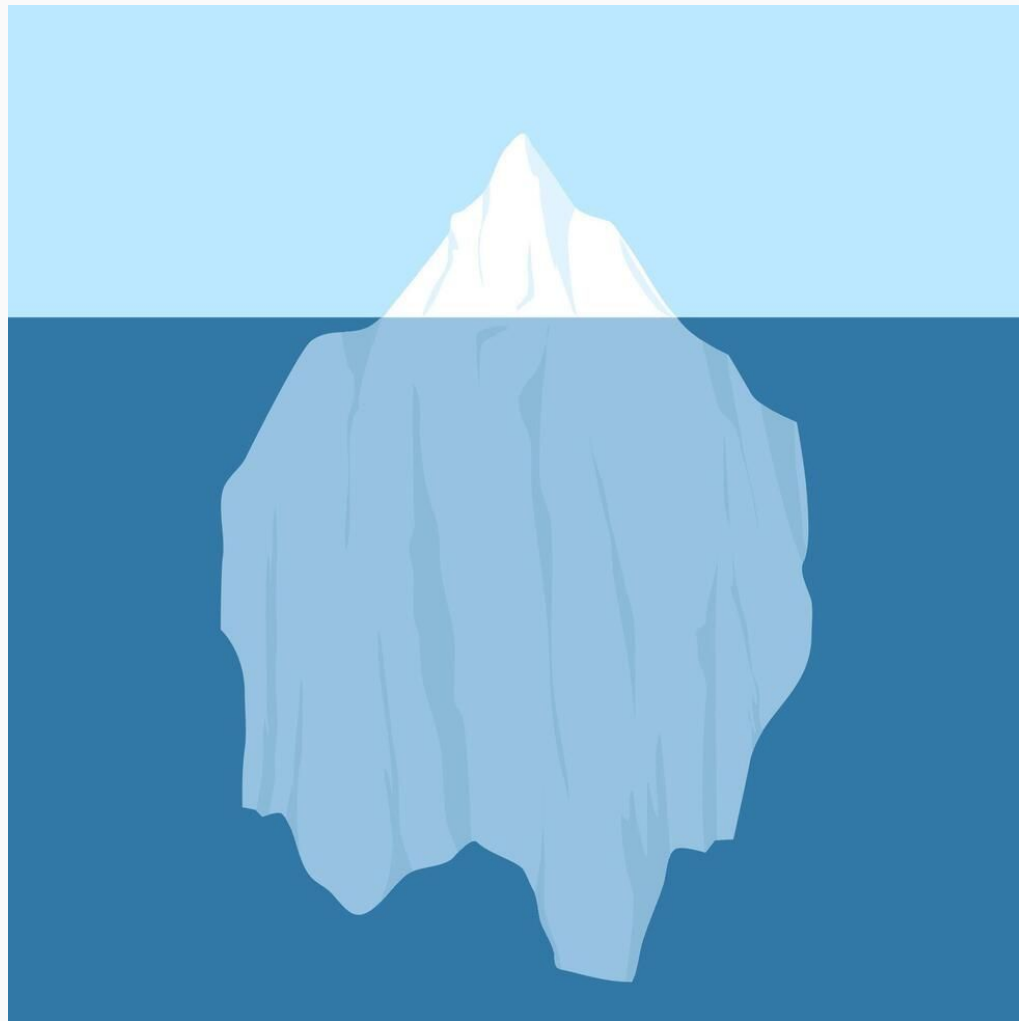
Pablo Hugen @ erad/rs 2025

Objetivo

Apresentar os principais modelos de programação paralela em GPU disponíveis no C++.

Ter um viés prático:

- Perguntem, **nenhuma** dúvida é trivial
- Interajam e compartilhem, torna o minicurso menos maçante.



Overview

- ***Introdução***
- GPU Architecture 101
- Prática 0: Configuração do Ambiente
- ***Computer Shaders***
Prática 1
- ***CUDA/HIP***
Prática 2
- ***OpenMP/OpenACC***
Prática 3
- ***OpenCL, SYCL***
Prática 4
- ***std::exec***
Prática 5
- Considerações Finais

Pesquisa

Quantos de vocês sabem o básico de:

- C++
- Git
- Docker
- “Se virar” no terminal (compilar, rodar scripts, ...)

Introdução

Nos últimos anos o uso de GPUs cresce exponencialmente

O principal responsável está sendo os Large Language Models, porém vários outros campos contribuem para essa estatística:

- Processamento de Imagens e Vídeo
- Simulações Científicas
- Bioinformática e Genômica
- Jogos (Óbvio)

LLMs: Treinamento e Inferência

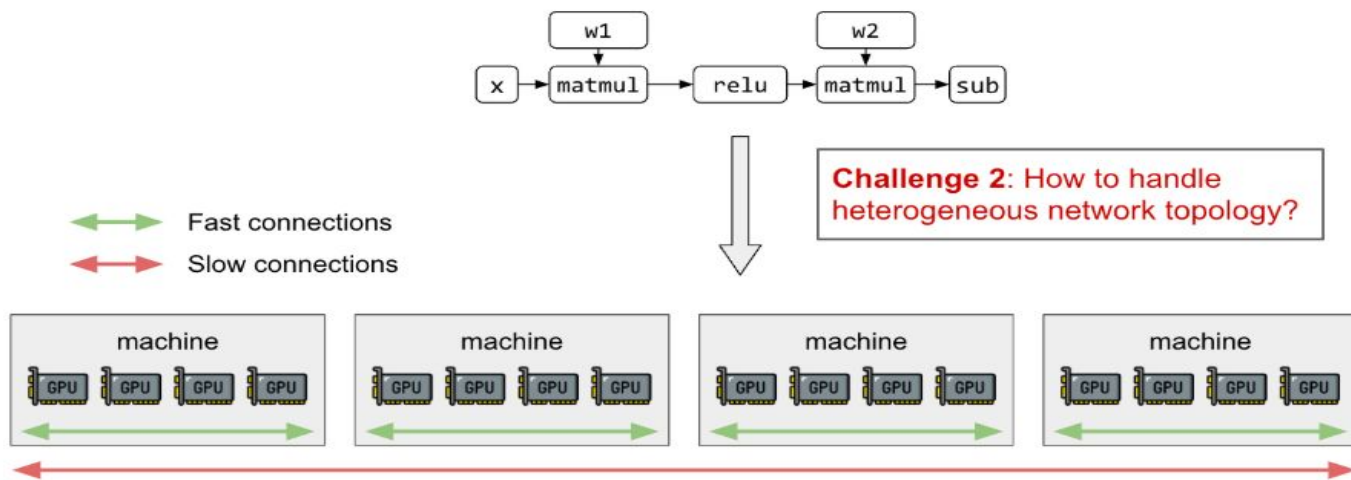




Figure 2. Network topology of GPU clusters

<https://developer.nvidia.com/blog/efficiently-scale-llm-training-across-a-large-gpu-cluster-with-alpa-and-ray/>

LLMs: Treinamento e Inferência



Sam Altman  

@sama

it's super fun seeing people love images in chatgpt.

but our GPUs are melting.

Simuladores Multi-Agente

Figura 14 – Ambiente pequeno



Figura 15 – Ambiente médio

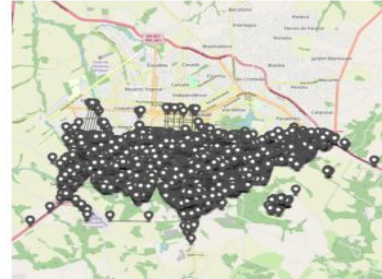


Figura 16 – Ambiente grande

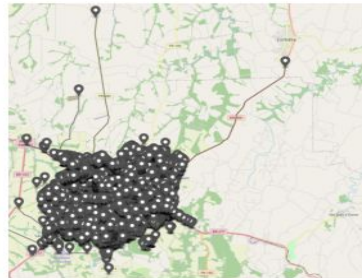
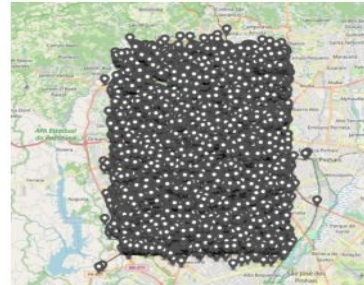


Figura 17 – Ambiente muito grande



Fonte: Autor.

Introdução

C++ sempre se mantém entre as linguagens mais usadas ano após ano (por bem ou por mal).

Usada em vários campos:

- Systems Software
- Gamedev
- IoT/Embarcados
- HPC
- ...

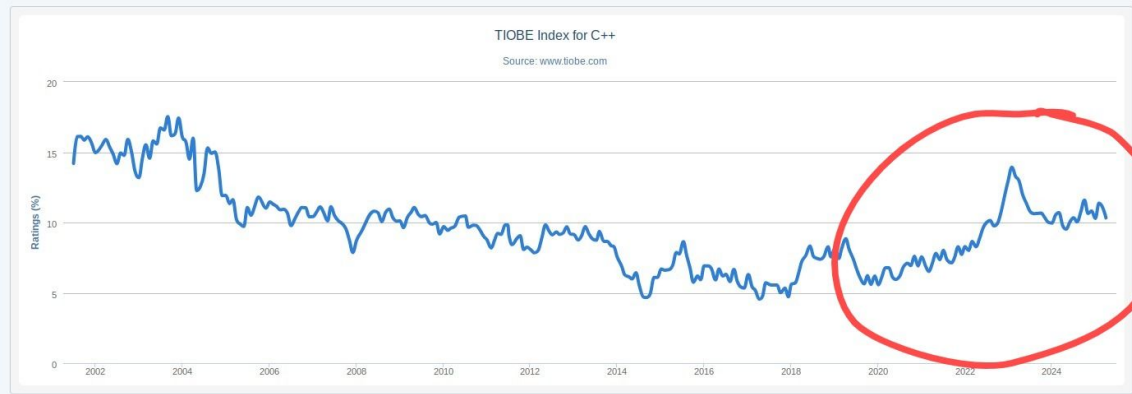
The C++ Programming Language

Some information about C++:

📈 Highest Position (since 2001): #2 in Apr 2025

📉 Lowest Position (since 2001): #5 in Feb 2008

🏆 Language of the Year: 2003, 2022



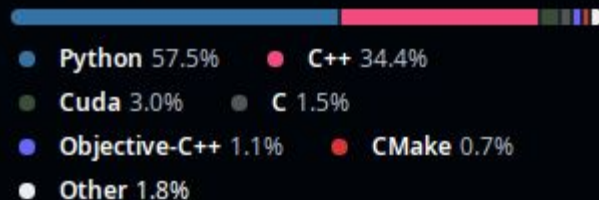
<https://www.tiobe.com/tiobe-index/cplusplus/>

Introdução

Dentro do contexto de HPC usando GPUs, o C++ é onipresente:

- Modelos: CUDA, HIP
- Frameworks: Thrust, FlameGPU
- Bibliotecas de linguagens interpretadas (principalmente python): Pytorch, OpenCV, TensorFlow

Languages

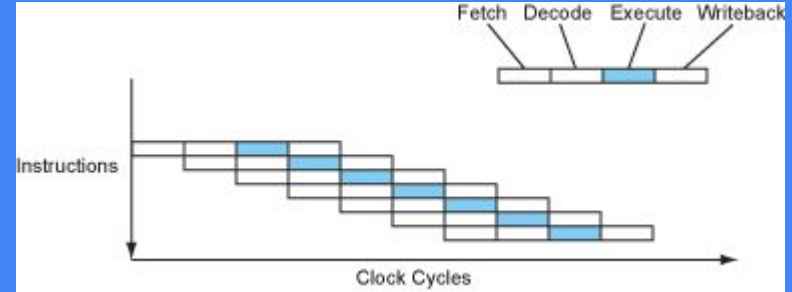


GPU Architecture 101



GPU 101

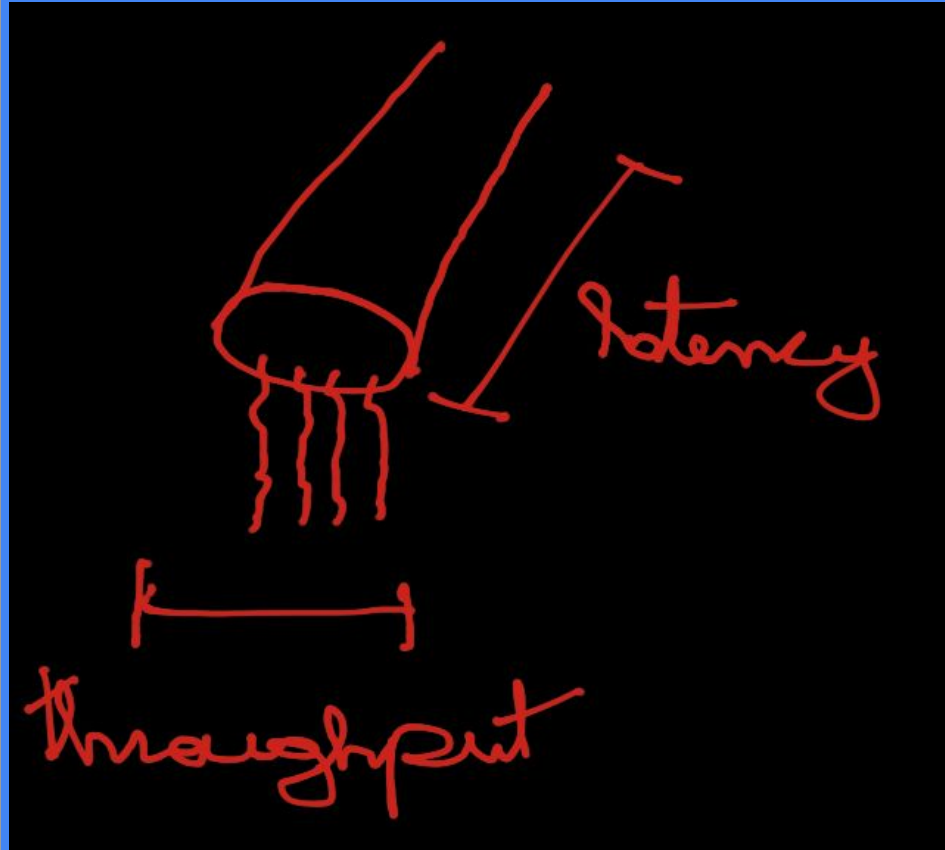
- CPUs são feitas para execução sequencial:
 - Cores mais poderosos
 - Redução de latência: Pipelining, Hierarquia de caches.
- Já as GPUs priorizam o alto throughput e paralelismo massivo, mesmo com uma maior latência na execução de instruções:
 - Menos área no chip p/ caches e UC
 - Grande número de ULAs



GPU 101

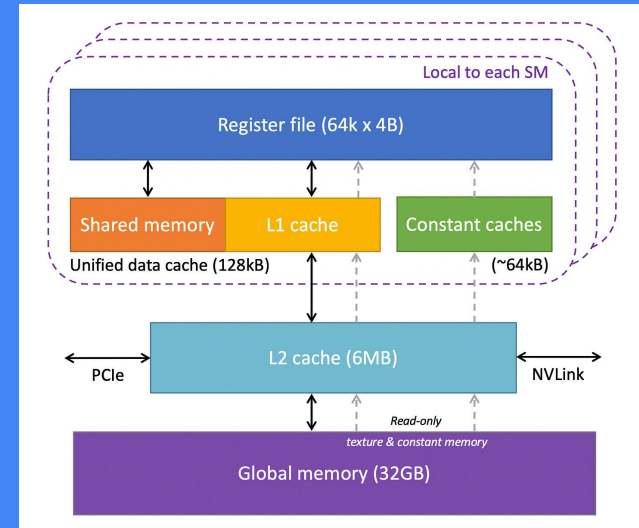
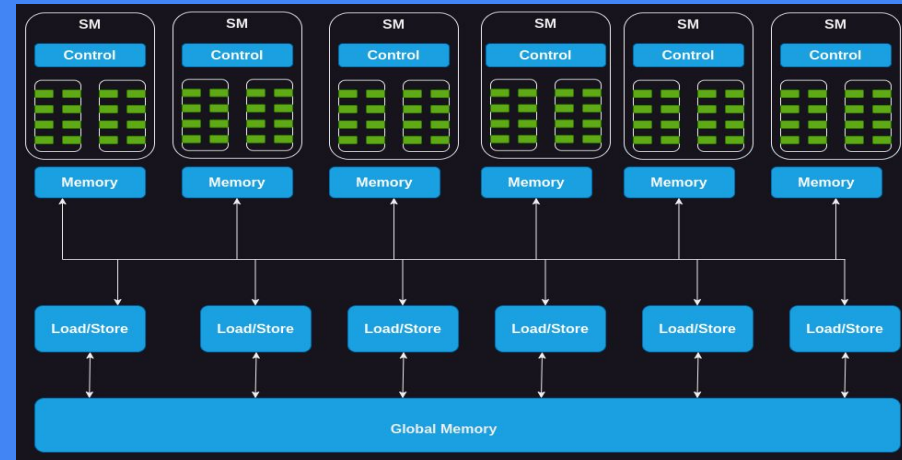
Latency vs. throughput

- Instruções individuais são lentas (alta latência)
- Porém se executarmos essa instrução em uma maior quantidade de dados, a quantidade de resultados (throughput) compensa a latência.

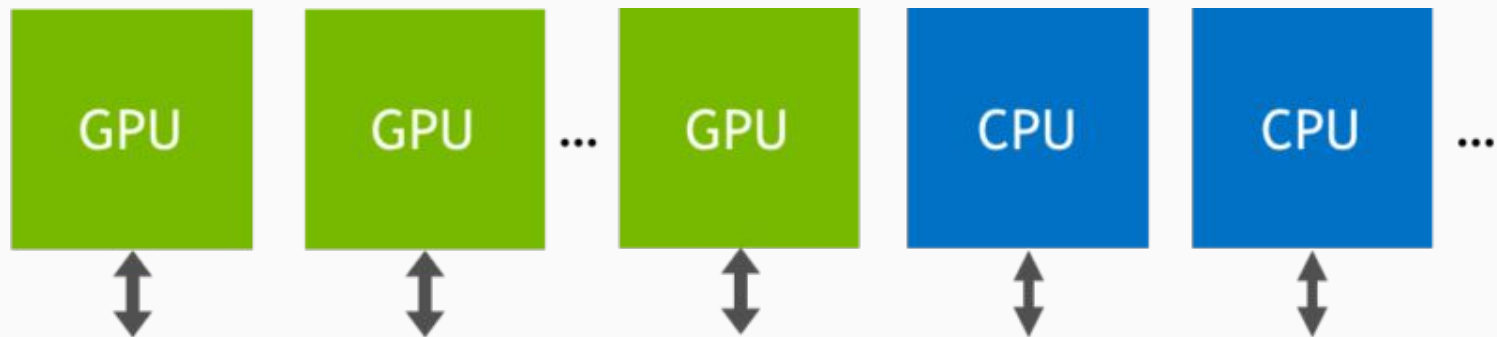


GPU 101

- A GPU é composta por um conjunto de *Streaming Multiprocessors*
- Cada SM possui um conjunto de *threads*, e uma memória compartilhada entre elas



GPU 101



Unified Memory

Prática 0: Configuração do Ambiente

git clone https://github.com/HpcResearchLaboratory/gpu_cpp_minicourse

Instruções no Readme: Containers

Compute Shaders

- Shaders são programas executados na GPU em estágios específicos do pipeline gráfico:
 - Vertex
 - Fragment
 - ...
- APIs gráficas antigas (👉 OpenGL) tinham o pipeline fixo e não eram nada flexíveis.

Compute Shaders

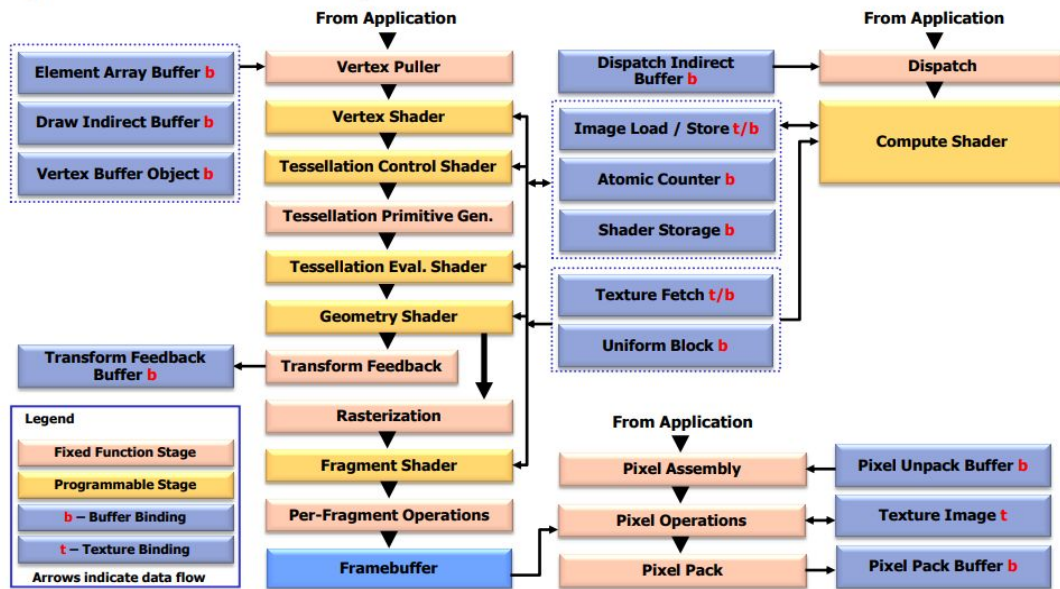
Versões modernas incluem os *compute shaders*: uso da GPU para processamento genérico.

Os dados de entrada são passados via uniforms

Executam em *workgroups* em uma ordem não definida. Cada *workgroup* invoca esse shader N vezes.

Comunicação apenas intra grupos.

OpenGL 4.3 Pipelines



Prática 1: Compute Shader N-Body Simulation (~15 min)

Instruções no Readme: Compute Shaders

CUDA

API Principal da NVIDIA para uso de suas GPUs.

Baseada em C++

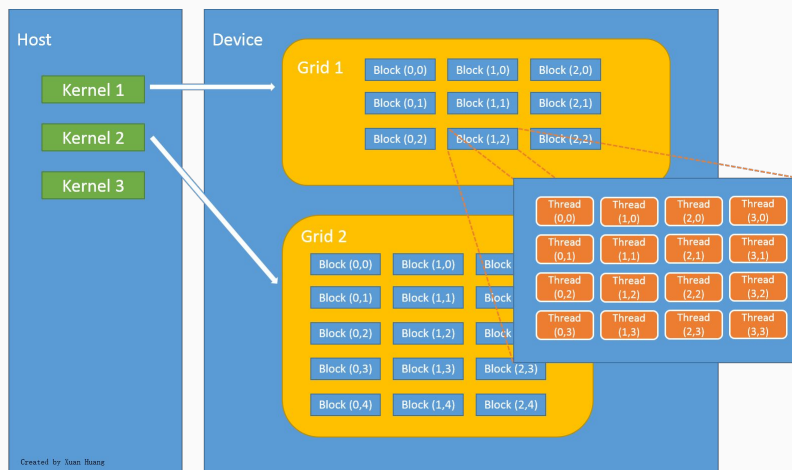
AMD: HIP (tentativa deles de “Eliminar” o vendor lock-in na NVIDIA.

CUDA

- API é baseada em *kernels*
- O trabalho é dividido entre os blocos
- O número de threads por bloco é parametrizável
- O kernel calcula as posições específicas baseando-se nos índices do block/thread atual.

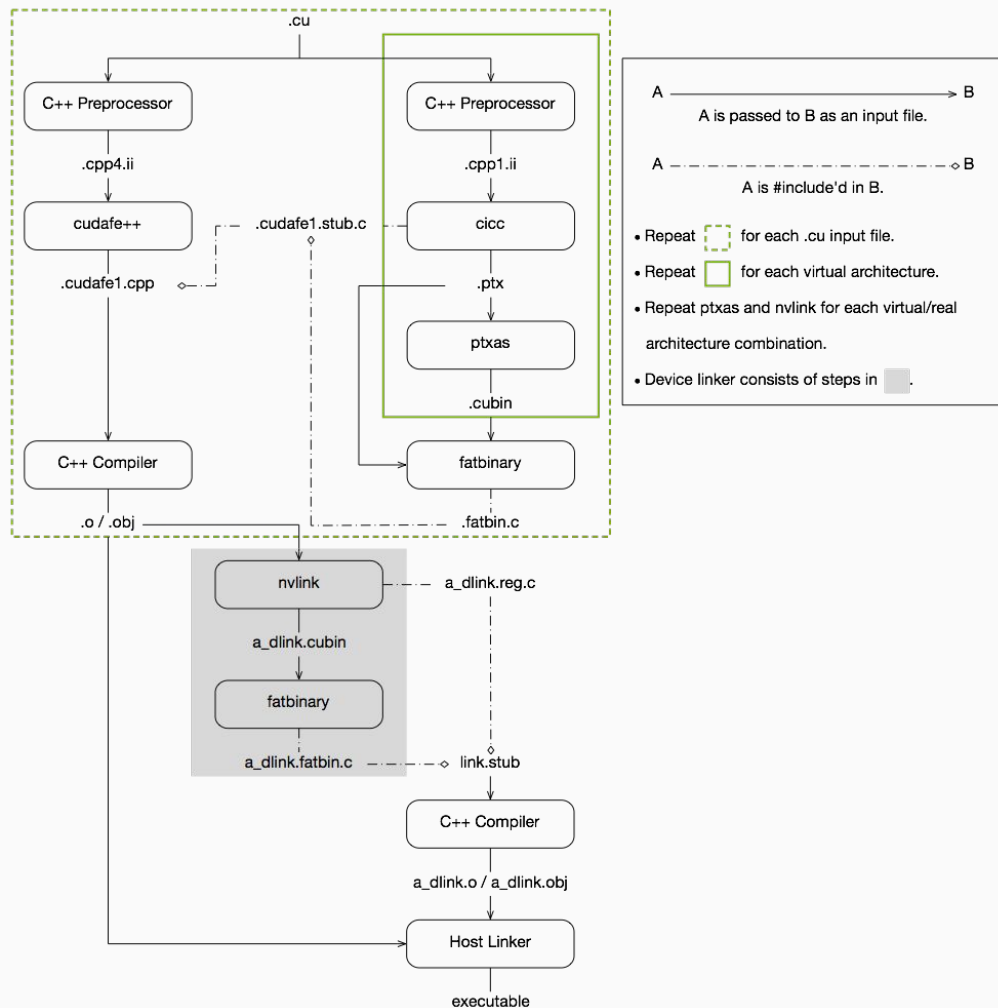
```
1  __global__ void matadd(float A[N][N], float B[N][N],
2  float C[N][N])
3  {
4      int i = blockIdx.x * blockDim.x + threadIdx.x;
5      int j = blockIdx.y * blockDim.y + threadIdx.y;
6      if (i < N && j < N)
7          C[i][j] = A[i][j] + B[i][j];
8  }
9
10 int main()
11 {
12     /*...*/
13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



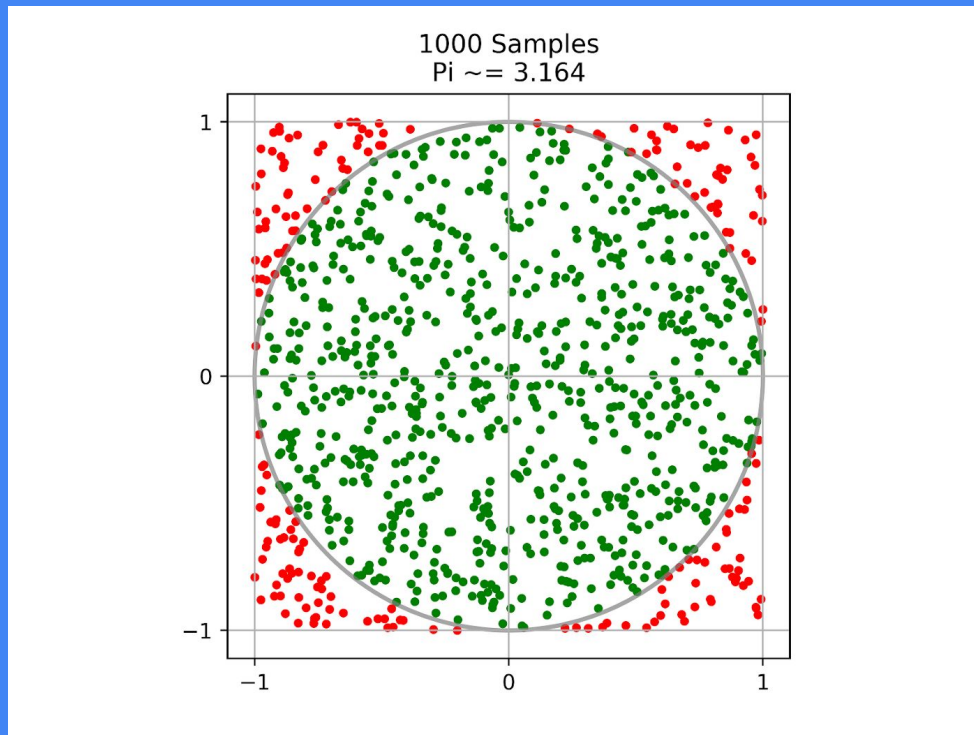
CUDA

- O compilador *nvcc* (proprietário) é usado para compilar CUDA.
- O principal trabalho dele é compilar separadamente o código que vai rodar na GPU (device) em PTX ou cubin e o código da CPU (host) que vai gerenciar as chamadas dos kernels.
- Vários outros compiladores modernos tem suporte a codegen CUDA (clang, zig cc, ...), porém o *nvcc* continua sendo a opção mais fácil.



Prática 2: CUDA Monte Carlo PI (~15 min)

Instruções no Readme: CUDA



OpenMP/OpenACC

- Ambas são APIs baseadas em diretivas de compilação (pragmas)
- OpenACC foi projetado desde o começo com foco em múltiplos aceleradores (“Open Accelerators”)
- Já o OpenMP ganhou suporte a GPUs somente a partir da versão 4
- O OpenACC também tem o foco de facilitar o uso de programação paralela para a comunidade científica (slogan “more science, less programming”)

OpenMP/OpenACC

- Muito úteis nos casos de paralelizar aplicações científicas sequenciais no modelo “loop => data transform”.
- Em muitos casos somente marcar loops com diretivas paralelas (map, reduce, loop, ...) e gerenciar o particionamento de dados.

```
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*.

Prática 3: OpenACC/OpenMP saxpy (~15 min)

Instruções no Readme: OpenACC

$$z = ax + y$$

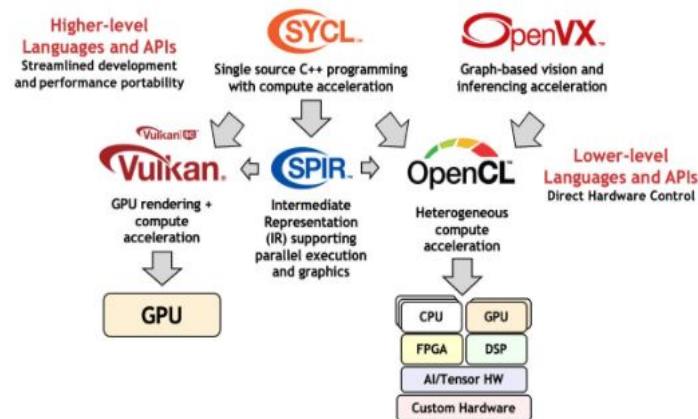
OpenCL

- O OpenCL é uma API Criada pelo *Khronos Group*, com foco em suporte multi-plataforma e multi-acelerador
- Controle de Baixo nível, porém suficientemente flexível para ser usada no desenvolvimento direto.

OpenCL

- Como programador você vai precisar ter um cuidado maior com detalhes de baixo nível, como ordenamento e barreiras de memória
- Geralmente é usada como destino de compilação de frameworks e compiladores de mais alto nível

```
1 __kernel void sum(  
2     __global const double *input,  
3     __global double *partialSums,  
4     __local double *localSums) {  
5     uint local_id = get_local_id(0);  
6     uint group_size = get_local_size(0);  
7     localSums[local_id] = input[get_global_id(0)];  
8  
9     for (uint stride = group_size / 2; stride > 0; stride /= 2) {  
10        barrier(CLK_LOCAL_MEM_FENCE);  
11  
12        if (local_id < stride)  
13            localSums[local_id] += localSums[local_id + stride];  
14    }  
15  
16    if (local_id == 0) partialSums[get_group_id(0)] = localSums[0];  
17 }
```



SYCL

- Já o foco do SYCL é ser uma abstração de alto nível para programação heterogênea
- Também desenvolvida pelo *Khronos Group*
- Funciona mais como um “padrão”, com cada vendor implementando sua API SYCL:
 - Intel oneAPI
 - openSYCL
 - triSYCL
 - ...

SYCL

- Utiliza a ideia de filas de trabalho
- Sendo possível enviar o mesmo kernel para diferentes filas em diferentes dispositivos (CPU, GPU, FPGA, ...)
- Porém, há impactos de performance (cópia de dados em cada execução para cada dispositivo e os kernels são compilados JIT)

```
sycl::queue q4(d_selector);  
sycl::queue q5(d_selector);  
sycl::queue q6(d_selector);  
VectorAdd(q4, q5, q6, a, b);
```

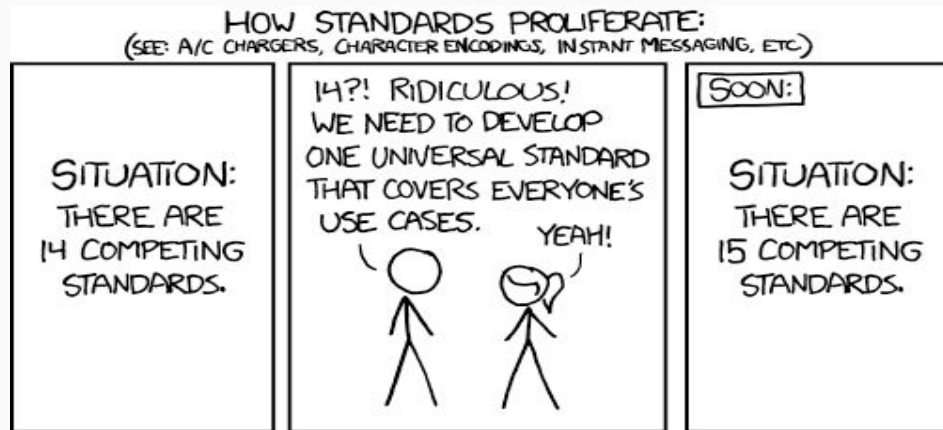
Prática 4: OpenCL, SYCL soma de vetores (~15 min)

Instruções no Readme: ***OpenCL, SYCL***

std::exec

Esforço da NVIDIA para padronizar no C++ um “framework” de:

- Computação paralela
- Computação heterogênea
- Computação assíncrona



std::exec

- Similar a ideia do SYCL:
Combinar a semântica/facilidade do C++ moderno com a programação heterogênea.
- Paralelismo estruturado:
 - Sender: Descreve uma tarefa concorrente.
 - Schedulers: Abstraem o local que essas tarefas vão ser executadas(CPU, GPU)
 - Receivers: Capturam os resultados e processam de acordo com o tipo de conclusão (sucesso, erro, cancelamento).

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

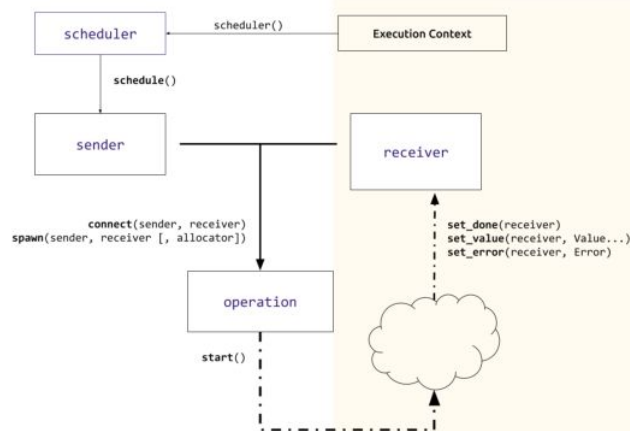


Figura 1.7: Framework std::exec. Fonte: [Jabot, 2019]

Prática 5: std::exec (?) (~15 min)

Instruções no Readme: ***std::exec***

Tempo livre: Sem exercício específico.

Mudem o código, compilem, façam perguntas, ...

Conclusão

- Espero que todos tenham tido uma ideia das opções disponíveis para programação em GPU usando o C++.
- Quem quiser se aprofundar siga os links no Readme do repositório e/ou as referências da versão em PDF do minicurso:

<https://cradrs.github.io/eradrs2025/pdfs/minicursos/MC1.pdf>