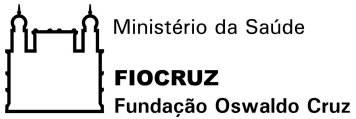
SYCL

Amanda S. Dufek
asdufek@lbl.gov

Douglas A. Augusto
daa@fiocruz.br



2021-01-20

Material online

Exemplos:

- github.com/codeplaysoftware/computecpp-sdk/tree/master/samples
- github.com/alcf-perfengr/sycltrain
- github.com/Apress/data-parallel-CPP/tree/main/samples

Documentação:

- sycl.tech
- www.khronos.org/sycl/
- github.com/codeplaysoftware/syclacademy/tree/main/Lesson_Materials
- Livro *Mastering DPC++ for Programming of Heterogeneous Systems using C++ and SYCL*
  - www.apress.com/gp/book/9781484255735

Especificação:

- www.khronos.org/registry/SYCL/specs/sycl-1.2.1.pdf
- www.khronos.org/registry/SYCL/specs/sycl-2020-provisional.pdf
- www.khronos.org/files/sycl/sycl-121-reference-guide.pdf

“SYCL é o mais *elegante*, *sofisticado* e *integrado* modelo portável de programação paralela heterogênea em C++ moderno”

“SYCL é o mais *elegante*, *sofisticado* e *integrado* modelo portável de programação paralela heterogênea em C++ moderno”

- *Elegância*:
  - sintaxe C++ pura, simples e coesa;
  - dedução automática de dependências
- *Sofisticação*:
  - paralelismo incremental, do simples com pouco controle ao complexo com total controle;
  - múltiplos dispositivos simultaneamente
- *Integração*:
  - fiel à sintaxe do C++ moderno;
  - SYCL e C++ estão coevoluindo na mesma direção
- *Portabilidade*:
  - vários dispositivos e backends;
  - incita portabilidade funcional/desempenho pela abordagem mais descritiva

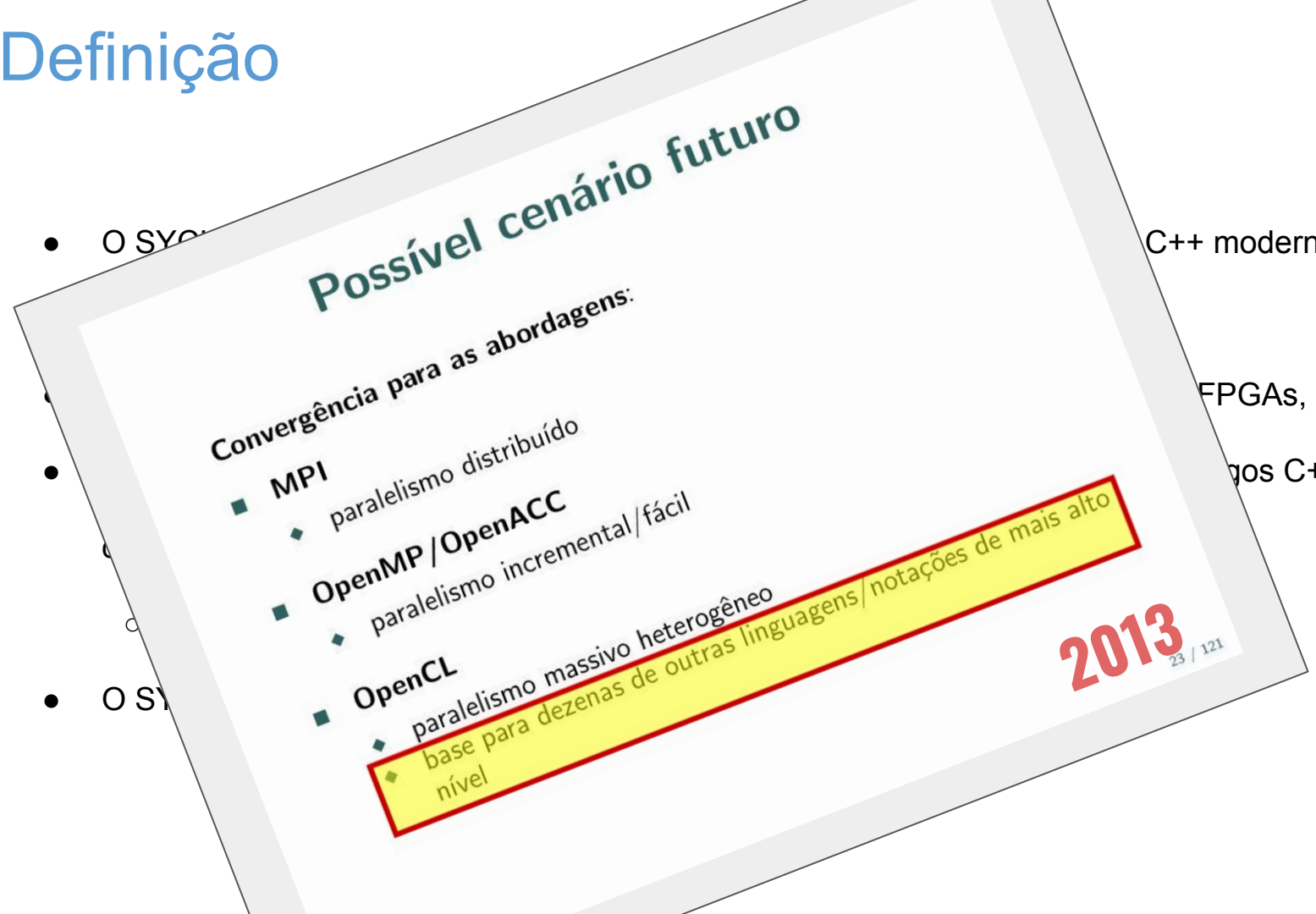
Definição

- O SYC
- O SY

C++ moderno e OpenCL

FPGAs, etc.)

gos C++ sequenciais



Definição

- O SYCL é um modelo de programação paralela heterogênea baseado em C++ moderno e OpenCL
- Códigos em SYCL seguem estritamente a sintaxe C++
- São compatíveis com (quase) todos os tipos de CPUs e aceleradores (GPUs, FPGAs, etc.)
- O SYCL une a vantagem da implementação incremental do paralelismo em códigos C++ sequenciais com o desempenho de linguagens de mais baixo nível, como o OpenCL e CUDA
    - produz códigos mais enxutos
- O SYCL é um padrão com especificação aberta, gerido pelo grupo Khronos

Ementa (1º dia)

- **Contextualização**
- **A linguagem SYCL**
  - Introdução
  - Filas de comando e dispositivos
  - *Unified Shared Memory* explícita
  - *Unified Shared Memory* implícita
  - Kernels: forma básica
  - Sincronismo no hospedeiro
  - Sincronismo no hospedeiro usando eventos
  - Plataformas e dispositivos
- **Exercícios**

Ementa (2º dia)

- **A linguagem SYCL**
  - Plataformas e dispositivos
  - *Buffers* e *accessors*
  - Modelo de execução
  - Modelo de memória
  - Kernels: forma *ND-Range*
    - Barreiras
    - *Local accessors*
- **Exercícios**

Algumas linguagens de programação paralela

- **MPI**
- **OpenMP**
- CUDA
- **OpenCL**
- Kokkos, Raja
- ***SYCL***

Desempenho

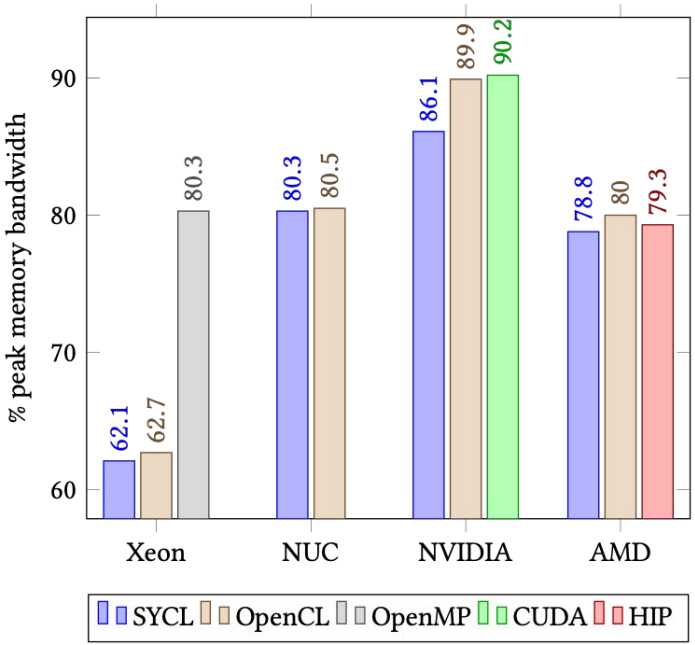


Figure 1: BabelStream Triad results

Paper: https://dl.acm.org/doi/abs/10.1145/3388333.3388643

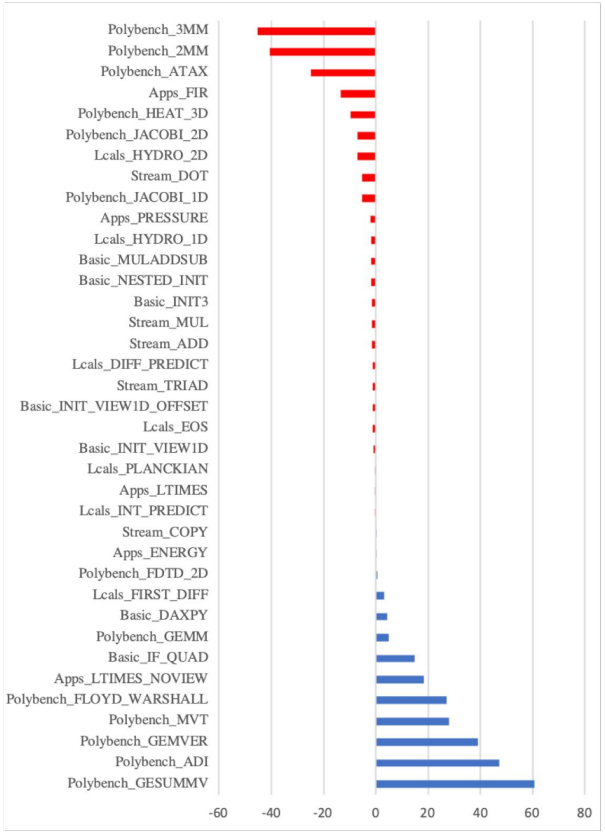


Figure 2: Percent speedup of SYCL kernels relative to CUDA kernels

Paper: https://dl.acm.org/doi/abs/10.1145/3388333.3388660

Ruptura nas arquiteturas HPC

- **Heterogeneidade**: tendência dos supercomputadores se tornarem cada vez mais heterogêneos
  - Diversidade de fabricantes e arquiteturas
- **Computadores de propósito específico**: computadores especializados para certas cargas de trabalho serão cada vez mais empregados
  - Exemplo: aprendizagem profunda

Rompimento da tradição

CPU IBM/Intel + GPU NVIDIA

Top 500, nov/2020

1	**Supercomputer Fugaku** - Supercomputer Fugaku, A64FX 48C 2.2GHz, Tofu interconnect D, Fujitsu RIKEN Center for Computational Science Japan	7,630,848	442,010.0	537,212.0	29,899

Green 500, jun/2020

1	393	**MN-3** - MN-Core Server, Xeon 8260M 24C 2.4GHz, MN-Core, RoCEv2/MN-Core DirectConnect, Preferred Networks Preferred Networks Japan	2,080	1,621.1	77	21.108

Green 500, nov/2019

1	159	**A64FX prototype** - Fujitsu A64FX, Fujitsu A64FX 48C 2GHz, Tofu interconnect D, Fujitsu Fujitsu Numazu Plant Japan	36,864	1,999.5	118	16.876

Green 500, nov/2018

1	375	**Shoubu system B** - ZettaScaler-2.2, Xeon D-1571 16C 1.3GHz, Infiniband EDR, PEZY-SC2, PEZY Computing / Exascaler Inc. Advanced Center for Computing and Communication, RIKEN Japan	953,280	1,063.3	60	17.604

Supercomputadores *exascale* (2021+)

As primeiras máquinas exascale previstas para 2021+ não possuem GPUs da NVIDIA.

CPU Intel + GPU Intel

CPU AMD + GPU AMD

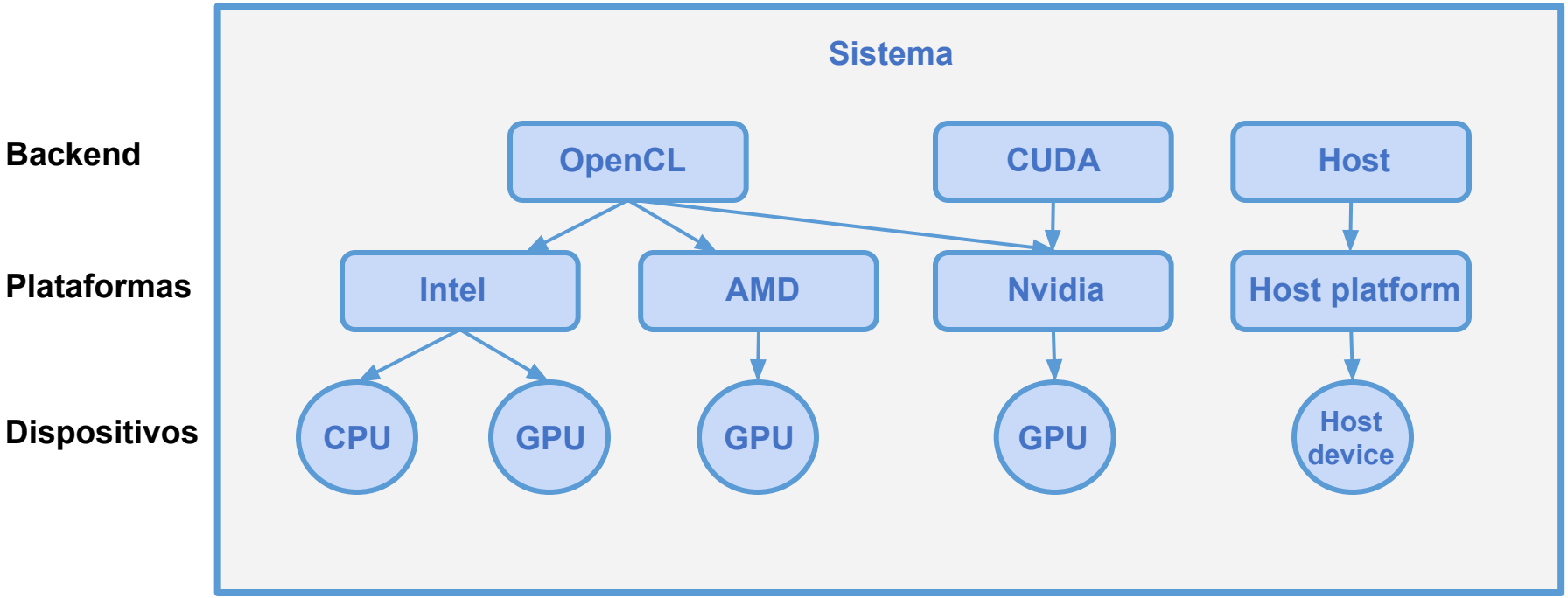
CPU AMD + GPU AMD

Supercomputadores têm vida útil de ~5 anos; não é admissível que softwares científicos sejam sentenciados ao mesmo período

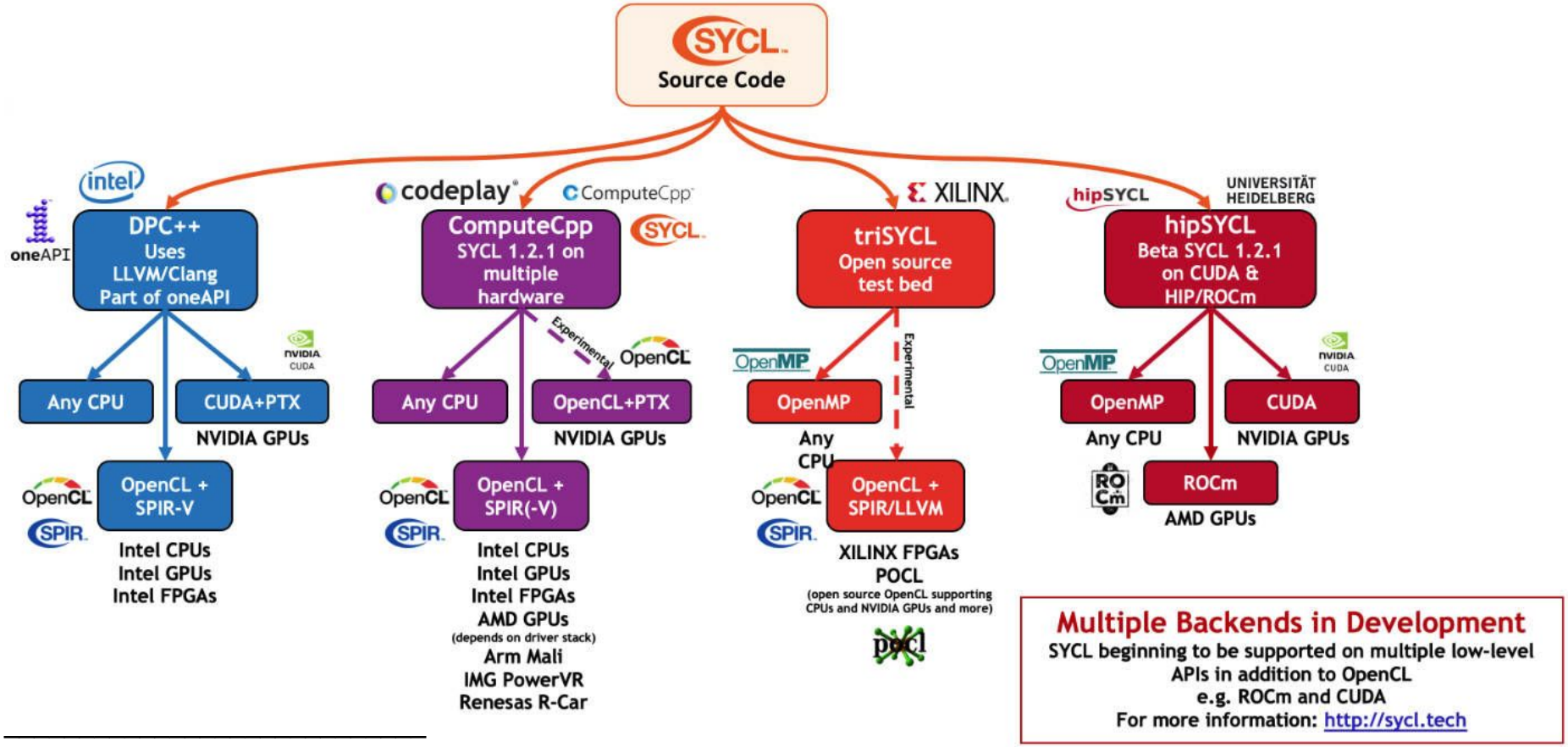
Outras informações

- SYCL não é um acrônimo; é apenas um nome
- Versões: SYCL 1.2 (2015), SYCL 1.2.1 (2017) e SYCL 2020 (2020/2021)
- SYCL 2020 suporta C++17 (C++ puro)
- Vários compiladores disponíveis
- Um único código, dois alvos (hospedeiro e dispositivo)
- Provê interface *homogênea* para a exploração da computação paralela em sistemas *heterogêneos*
- Código portável entre arquiteturas e gerações
- Backends: Host, OpenCL, CUDA, OpenMP, Vulkan, TBB…

Plataformas e dispositivos



Implementações



Créditos: Khronos groups (https://www.khronos.org/sycl/)

Comparação

Solução paralela via SYCL:

```
#include <CL/sycl.hpp>
#include <math.h>
using namespace sycl;

constexpr int N = 42;

int main() {
  queue Q;

  auto *array = malloc_shared<float>(N, Q);
  for (int i=0; i<N; i++) { array[i] = i; }

  Q.submit([&](handler &h) {
    h.parallel_for(N, [=] (id<1> i) {
      array[i] = sqrt(array[i]);
    });
  });
  Q.wait();

  free(array, Q);
  return 0;
}
```

Solução sequencial:

```
#include <stdlib.h>
#include <math.h>


constexpr int N = 42;

int main() {


  float *array = (float*)malloc(N*sizeof(float));
  for(int i=0; i<N; i++) { array[i] = i; }



  for(int i=0; i<N; i++) { array[i] = sqrt(array[i]); }




  free(array);
  return 0;
}
```

Cabeçalho

```
#include <CL/sycl.hpp>
using namespace cl::sycl;

int main() {
  ...
  return 0;
}
```

Define todos os construtores SYCL. Contém o API de execução.
Versão: SYCL 1.2.1

Para não ter que escrever cl::sycl:: o tempo todo.

Cabeçalho

```
#include <SYCL/sycl.hpp>
using namespace sycl;

int main() {
  ...
  return 0;
}
```

Define todos os construtores SYCL. Contém o API de execução.
Versão: SYCL 2020

Para não ter que escrever sycl:: o tempo todo.

Cabeçalho

```
#include <CL/sycl.hpp>
using namespace sycl;

int main() {
  ...
  return 0;
}
```

Passos gerais para a programação

1. Criando as filas de comandos para os dispositivos
2. Escolhendo os dispositivos
3. Transferência dos dados para o dispositivo
4. Criando o kernel
5. Execução do kernel
6. Transferência dos resultados para o hospedeiro
7. Sincronismo
8. Liberação de memória

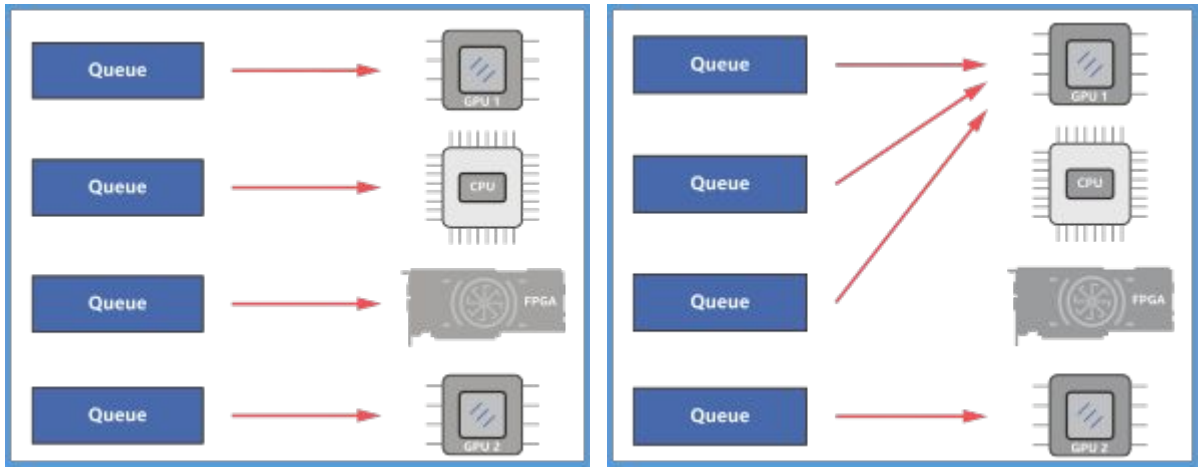
Passos gerais para a programação

1. Criando as filas de comandos para os dispositivos
2. Escolhendo os dispositivos
3. Transferência dos dados para o dispositivo
4. Criando o kernel
5. Execução do kernel
6. Transferência dos resultados para o hospedeiro
7. Sincronismo
8. Liberação de memória

Filas de comando

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

int main() {
  sycl::queue Q;
  ...
  return 0;
}
```

Criando a fila de comandos para um dispositivo.
As filas permitem a comunicação do programa com os dispositivos.



Créditos: Mastering DPC++ for Programming of Heterogeneous Systems using C++ and SYCL

Passos gerais para a programação

1. Criando as filas de comandos para os dispositivos
2. Escolhendo os dispositivos
3. Transferência dos dados para o dispositivo
4. Criando o kernel
5. Execução do kernel
6. Transferência dos resultados para o hospedeiro
7. Sincronismo
8. Liberação de memória

Escolhendo os dispositivos

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

int main() {
  sycl::queue Q;
  ...
  return 0;
}
```

Escolhendo os dispositivos

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

int main() {
  sycl::queue Q{sycl::default_selector{}};
  ...
  return 0;
}
```

Escolhendo os dispositivos

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

int main() {
  sycl::queue Q(sycl::default_selector{});
  ...
  return 0;
}
```

Escolhendo os dispositivos

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

int main() {
  sycl::queue Q{sycl::host_selector{}};
  ...
  return 0;
}
```

Classe ***host_selector***

- Seleciona o dispositivo *host* (sempre disponível).
- É geralmente uma CPU.
- Ideal para a depuração do código (SYCL kernel). Nesse caso, você pode fazer uso do gdb, por exemplo.

host_selector
default_selector (implícito)
cpu_selector
gpu_selector
accelerator_selector

Passos gerais para a programação

1. Criando as filas de comandos para os dispositivos
2. Escolhendo os dispositivos
3. Transferência dos dados para o dispositivo (USM explícita)
4. Criando o kernel
5. Execução do kernel
6. Transferência dos resultados para o hospedeiro (USM explícita)
7. Sincronismo
8. Liberação de memória (explícita)

Grupo de comando

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

int main() {
  sycl::queue Q;

  Q.submit([&](handler &h) {
    //grupo de comando
  });
  ...
  return 0;
}
```

Um **grupo de comando** pode conter:

- Um comando SYCL (e.g. SYCL kernel)
- Range de execução
- *Accessors*

Para a submissão de trabalho na fila usa-se a função membro ***submit***.

Apenas um comando SYCL por grupo de comando pode ser submetido na fila (e.g. operações explícitas de memória e execuções de kernels).

Gerenciamento de dados: USM explícito

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

constexpr int N = 42;

int main() {
  std::array<int,N> host_array;
  for (int i=0; i<N; i++) { host_array[i] = i; }

  sycl::queue Q;

  auto *device_array = sycl::malloc_device<int>(N, Q);

  Q.submit([&](handler &h) {
    h.memcpy(device_array, &host_array[0], N*sizeof(int));
  });

  ...

  Q.submit([&](handler &h) {
    h.memcpy(&host_array[0], device_array, N*sizeof(int));
  });

  ...
  sycl::free(device_array, Q);
  return 0;
}
```

Unified Shared Memory (USM) é um modelo baseado em ponteiro. Nem todos os dispositivos suportam todos os tipos de alocação USM.

Unified Shared Memory (USM): ***device allocation***

- A alocação de memória acontece no dispositivo.
- O hospedeiro não pode acessar os dados.
- Transferência explícita de dados para a memória do dispositivo.

Gerenciamento de dados: USM explícito

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

constexpr int N = 42;

int main() {
  std::array<int,N> host_array;
  for (int i=0; i<N; i++) { host_array[i] = i; }

  sycl::queue Q;

  auto *device_array = sycl::malloc_device<int>(N, Q);

  Q.memcpy(device_array, &host_array[0], N*sizeof(int));



  ...

  Q.memcpy(&host_array[0], device_array, N*sizeof(int));



  ...
  sycl::free(device_array, Q);
  return 0;
}
```

Unified Shared Memory (USM) é um modelo baseado em ponteiro. Nem todos os dispositivos suportam todos os tipos de alocação USM.

Unified Shared Memory (USM): ***device allocation***

- A alocação de memória acontece no dispositivo.
- O hospedeiro não pode acessar os dados.
- Transferência explícita de dados para a memória do dispositivo.

Não precisamos usar grupos de comando quando usamos USM.

Passos gerais para a programação

1. Criando as filas de comandos para os dispositivos
2. Escolhendo os dispositivos
3. Transferência dos dados para o dispositivo (USM implícita)
4. Criando o kernel
5. Execução do kernel
6. Transferência dos resultados para o hospedeiro (USM implícita)
7. Sincronismo
8. Liberação de memória (explícita)

Gerenciamento de dados: USM implícito

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

constexpr int N = 42;

int main() {

  sycl::queue Q;

  auto *shared_array = sycl::malloc_shared<int>(N, Q);



  ...

  sycl::free(shared_array, Q);
  return 0;
}
```

Unified Shared Memory (USM) é um modelo baseado em ponteiro. Nem todos os dispositivos suportam todos os tipos de alocação USM.

Unified Shared Memory (USM): ***shared allocation***

- Alocação de memória compartilhada entre hospedeiro e dispositivo.
- Hospedeiro e dispositivo podem acessar os dados.
- Transferência implícita de dados para a memória do dispositivo ou do hospedeiro.
- Os dados migram livremente entre hospedeiro e dispositivo sem a nossa interferência.

Gerenciamento de dados: USM implícito

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

constexpr int N = 42;

int main() {

  sycl::queue Q;

  auto *host_array = sycl::malloc_host<int>(N, Q);

  for (int i=0; i<N; i++) { host_array[i] = i; }

  ...

  sycl::free(host_array, Q);
  return 0;
}
```

Unified Shared Memory (USM) é um modelo baseado em ponteiro. Nem todos os dispositivos suportam todos os tipos de alocação USM.

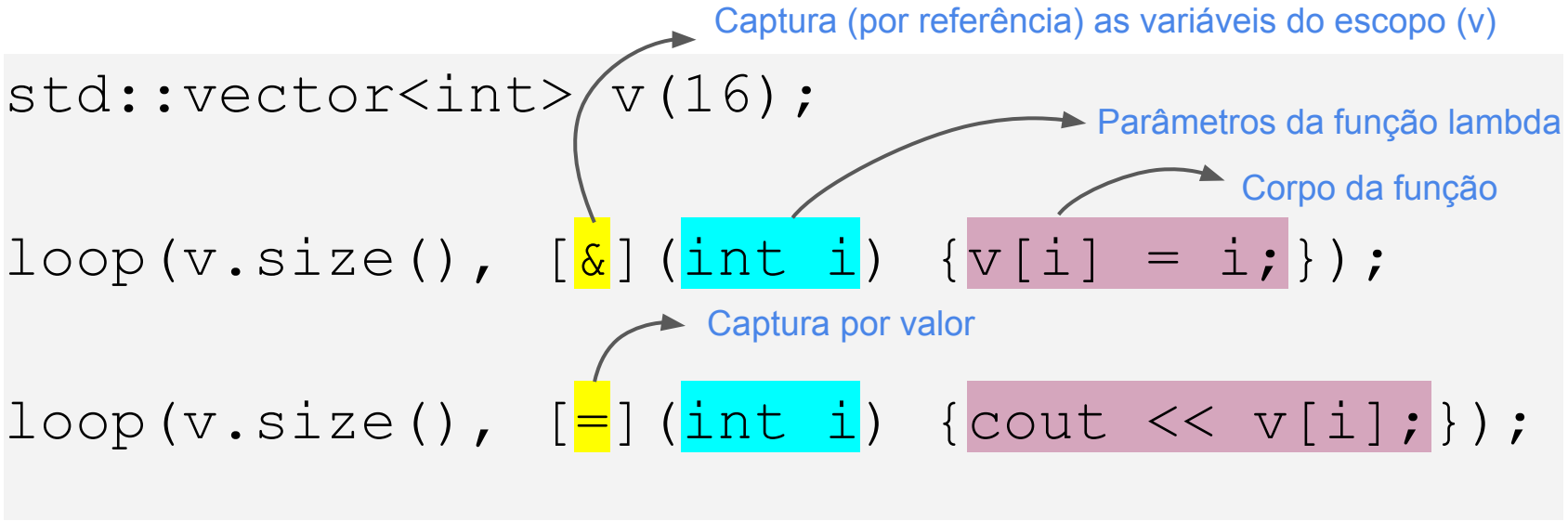
Unified Shared Memory (USM): ***host allocation***

- A alocação de memória acontece no hospedeiro.
- Hospedeiro e dispositivo podem acessar os dados.
- Os dados **NÃO** são transferidos para a memória do dispositivo; quando necessário os dados trafegam através do *PCI-Express bus*.
- Pode ser usada para dados pouco acessados ou para grandes conjuntos de dados que não cabem na memória do dispositivo.

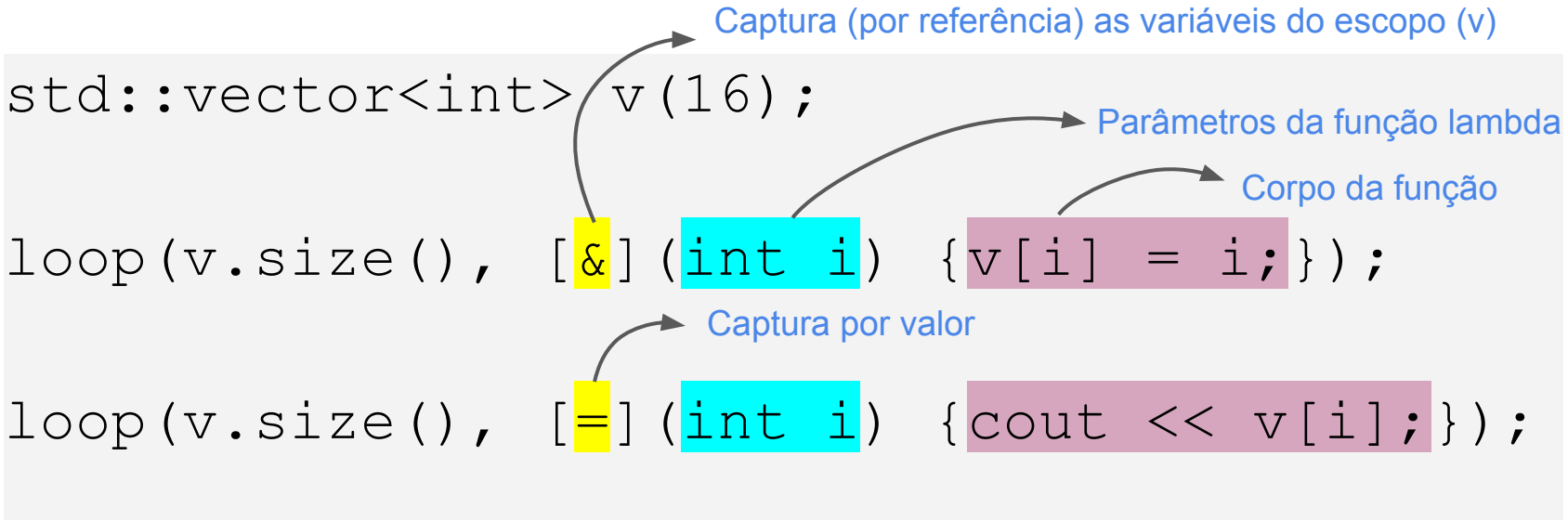
Passos gerais para a programação

1. Criando as filas de comandos para os dispositivos
2. Escolhendo os dispositivos
3. Transferência dos dados para o dispositivo
4. Criando o kernel
5. Execução do kernel
6. Transferência dos resultados para o hospedeiro
7. Sincronismo
8. Liberação de memória

Funções *lambda* (C++ moderno)



Funções *lambda* (C++ moderno)

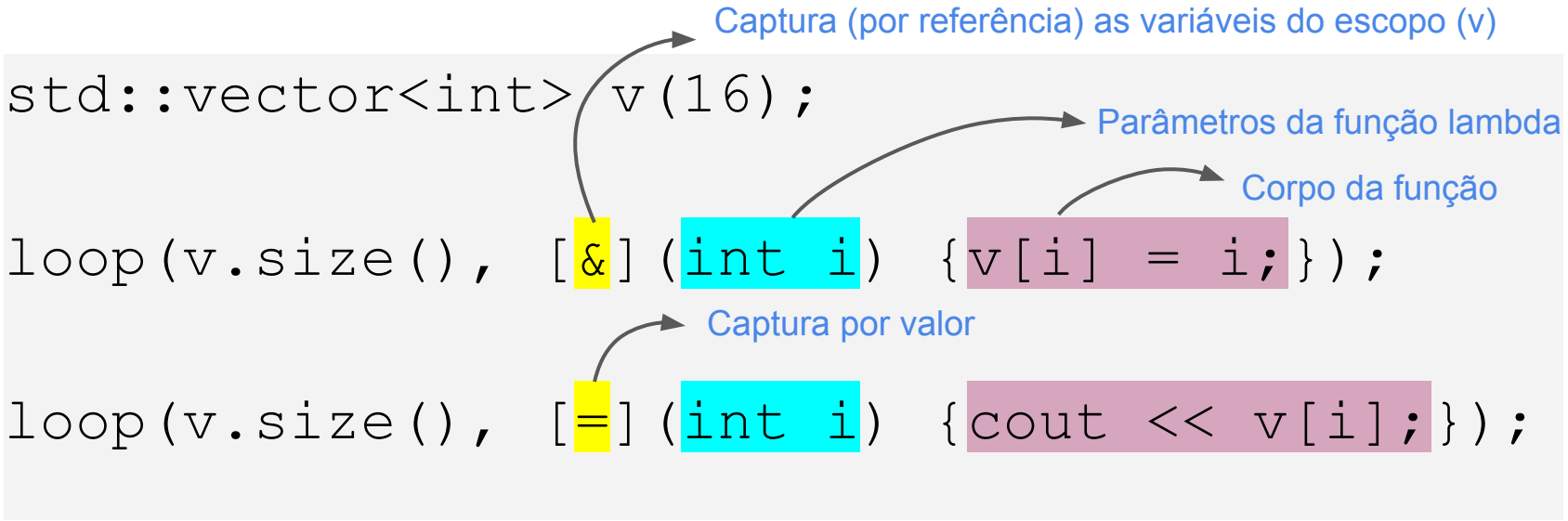


Implementação da função `loop`:

```
template<typename Func> void loop(int n, Func f) {
   for(int i=0; i<n; i++) f(i);
}
```

Chama a lambda (função `f`) para cada valor de `i`, uma por vez

Funções *lambda* (C++ moderno)




```
std::vector<int> v(16);

loop(v.size(), [&](int i) {v[i] = i;});

loop(v.size(), [=](int i) {cout << v[i];});
```

Outra possível implementação da função `loop`:

```
template<typename Func> void loop(int n, Func f) {
   #pragma omp parallel for
   for(int i=0; i<n; i++) f(i);
}
```

Chama a lambda (função `f`) para cada valor de `i`, em paralelo via OpenMP!

Kernels: forma básica

Solução sequencial:

```
for(int i=0; i<N; i++) {
  c[i] = a[i] + b[i];
}
```

Solução paralela via SYCL:

```
h.parallel_for(N, [=] (id<1> i) {c[i] = a[i] + b[i];});
```

Captura por **valor** as variáveis do escopo

Restrições:
- Alocação dinâmica
- Polimorfismo dinâmico
- Ponteiros para funções
- Recursividade

Item de trabalho (***work-item***):
- Os kernels são executados por itens de trabalho.
- Cada item de trabalho é executado por um elemento de processamento.
- O número total de itens de trabalho é chamado de ***global range***.

range: especifica o espaço de iteração (*global_range*). O *local_range* é definido pela aplicação.

função lambda: representa o início do kernel. O parâmetro **id** descreve a iteração corrente sendo executada; é um índice global.

offset global: parâmetro opcional
Para N = 1024 e offset = 512
-> idx = (512, 1536)

Kernels: forma básica

```
h.parallel_for(sycl::range{N}, [=] (sycl::id<1> idx) {
  c[idx] = a[idx] + b[idx];
});

h.parallel_for(sycl::range{N}, [=] (sycl::id<1> {offset}, sycl::id<1> idx)
{
  c[idx] = a[idx] + b[idx];
});
```

Item de trabalho (***work-item***):

- Os kernels são executados por itens de trabalho.
- Cada item de trabalho é executado por um elemento de processamento.
- O número total de itens de trabalho é chamado de ***global range***.

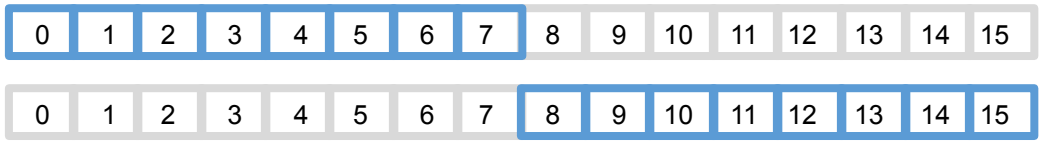
range: especifica o espaço de iteração (*global_range*). O *local_range* é definido pela aplicação.

função lambda: representa o início do kernel. O parâmetro **id** descreve a iteração corrente sendo executada; é um índice global.

offset global: parâmetro opcional
Para N = 1024 e offset = 512
-> idx = (512, 1536)

Kernels: forma básica

```
h.parallel_for(sycl::range{N}, [=] (sycl::id<1> idx) {
  c[idx] = a[idx] + b[idx];
});

h.parallel_for(sycl::range{N}, [=] (sycl::id<1> {offset}, sycl::id<1> idx)
{
  c[idx] = a[idx] + b[idx];
});
```



global range = 16
N = 8 e offset = 0 -> idx = (0,7)
N = 8 e offset = 8 -> idx = (8,15)

Kernels: forma básica

```
h.parallel_for(sycl::range{N}, [=] (sycl::id<1> idx) {
  c[idx] = a[idx] + b[idx];
});

h.parallel_for<kernel>(sycl::range<1>{N}, [=] (sycl::id<1> idx) {
  c[idx] = a[idx] + b[idx];
});
```

SYCL 1.2.1

Kernels: forma básica

Solução sequencial:

```
for (int i=0; i<N; i++) {        //nlin
  for (int j=0; j<M, j++) {      //ncol
    c[i][j] = a[i][j] + b[i][j];
}
```

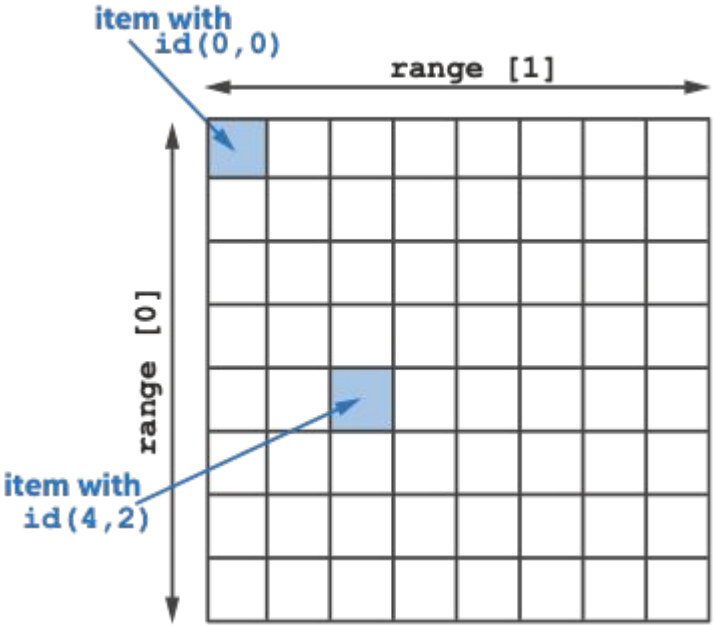
Solução paralela via SYCL:

```
h.parallel_for(range{N,M}, [=] (id<2> idx) {
  c[idx] = a[idx] + b[idx];
});
```

range pode ter 1, 2 ou 3 dimensões.

ou

```
h.parallel_for(range{N,M}, [=] (id<2> idx) {
  int i = idx[0];  //nlin
  int j = idx[1];  //ncol
  c[i][j] = a[i][j] + b[i][j];
});
```



Créditos: Mastering DPC++ for Programming of Heterogeneous Systems using C++ and SYCL

Passos gerais para a programação

1. Criando as filas de comandos para os dispositivos
2. Escolhendo os dispositivos
3. Transferência dos dados para o dispositivo
4. Criando o kernel
5. Execução do kernel
6. Transferência dos resultados para o hospedeiro
7. Sincronismo
8. Liberação de memória

Kernels: forma básica

```
#include <CL/sycl.hpp>
#include <math.h>
using namespace sycl;

constexpr int N = 42;

int main() {
  queue Q;

  auto *array = malloc_shared<float>(N, Q);
  for (int i=0; i<N; i++) { array[i] = i; }

  Q.submit([&](handler &h) {
    h.parallel_for(N, [=] (id<1> i) { array[i] = sqrt(array[i]); });
  });
  Q.wait();

  free(array, Q);
  return 0;
}
```

Barreira explícita: o objeto fila irá esperar até a conclusão de todos os trabalhos na fila.

Kernels: forma básica

```
#include <CL/sycl.hpp>
#include <math.h>
using namespace sycl;

constexpr int N = 42;

int main() {
  queue Q;

  auto *array = malloc_shared<float>(N, Q);
  for (int i=0; i<N; i++) { array[i] = i; }

  Q.parallel_for(N, [=] (id<1> i) { array[i] = sqrt(array[i]); });
  Q.wait();



  free(array, Q);
  return 0;
}
```

Não precisamos usar grupos de comando quando usamos USM.

Kernels: forma básica

```
#include <CL/sycl.hpp>
#include <math.h>
using namespace sycl;

constexpr int N = 42;

int main() {
  queue Q;

  auto *array = malloc_shared<float>(N, Q);
  for (int i=0; i<N; i++) { array[i] = i; }

  Q.submit([&](handler &h) {
    h.parallel_for(N, [=] (id<1> i) {
      array[i] = sqrt(array[i]);
    });
  });
  Q.wait();

  free(array, Q);
  return 0;
}
```

Hospedeiro

Dispositivo

Hospedeiro

Passos gerais para a programação

1. Criando as filas de comandos para os dispositivos
2. Escolhendo os dispositivos
3. Transferência dos dados para o dispositivo
4. Criando o kernel
5. Execução do kernel
6. Transferência dos resultados para o hospedeiro
7. Sincronismo
8. Liberação de memória

Sincronismo no hospedeiro

```
sycl::queue Q;

auto *device_array = sycl::malloc_device<int>(N, Q);

Q.submit([&](handler &h) {
  h.memcpy(device_array, &host_array[0], N*sizeof(int));
});

Q.submit([&](handler &h) {
  h.parallel_for(N, [=] (id<1> idx) {
    device_array[idx]++;
  });
});

Q.submit([&](handler &h) {
  h.memcpy(&host_array[0], device_array, N*sizeof(int));
});


sycl::free(device_array, Q);
```

Sincronismo no hospedeiro: fila em ordem

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

int main() {
  sycl::queue Q{sycl::property::queue::in_order()};
  ...
  return 0;
}
```

Objetos do tipo fila são **fora de ordem** por ***default***. Numa fila em ordem, as tarefas são executadas na ordem em que foram submetidas.

Sincronismo no hospedeiro: fila em ordem

```
sycl::queue Q{property::queue::in_order()};

auto *device_array = sycl::malloc_device<int>(N, Q);

Q.submit([&](handler &h) {
  h.memcpy(device_array, &host_array[0], N*sizeof(int));
});

Q.submit([&](handler &h) {
  h.parallel_for(N, [=] (id<1> idx) {
    device_array[idx]++;
  });
});

Q.submit([&](handler &h) {
  h.memcpy(&host_array[0], device_array, N*sizeof(int));
});
Q.wait();

sycl::free(device_array, Q);
```

Criando a fila de comandos para um dispositivo

Transferência dos dados para o dispositivo

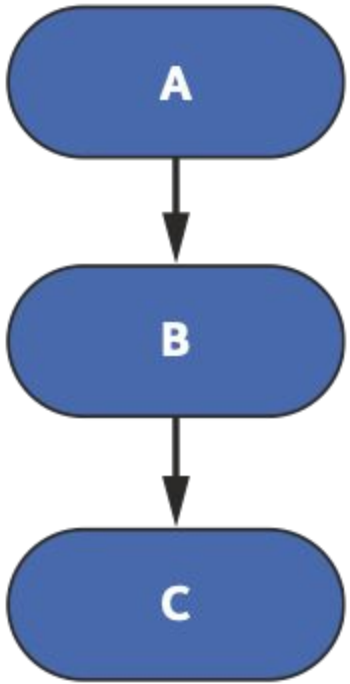
Execução do kernel

Transferência dos resultados para o hospedeiro

Liberação de memória

Sincronismo no hospedeiro: fila em ordem

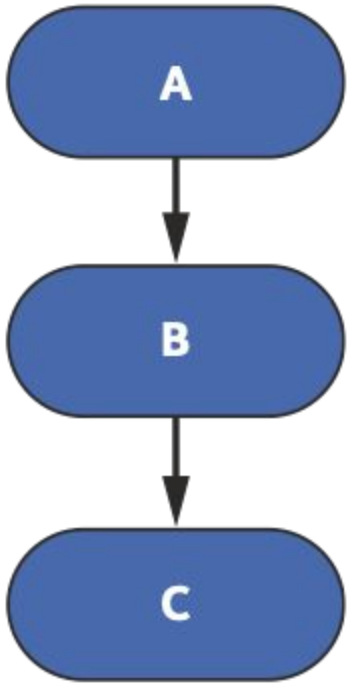
```
sycl::queue Q{property::queue::in_order()};

Q.submit([&](handler& h) {
  h.parallel_for(N, [=] (id<1> i) { /*...*/ });
});

Q.submit([&](handler& h) {
  h.parallel_for(N, [=] (id<1> i) { /*...*/ });
});

Q.submit([&](handler& h) {
  h.parallel_for(N, [=] (id<1> i) { /*...*/ });

});
```



Créditos: Mastering DPC++ for Programming of Heterogeneous Systems using C++ and SYCL

Sincronismo no hospedeiro: barreiras explícitas

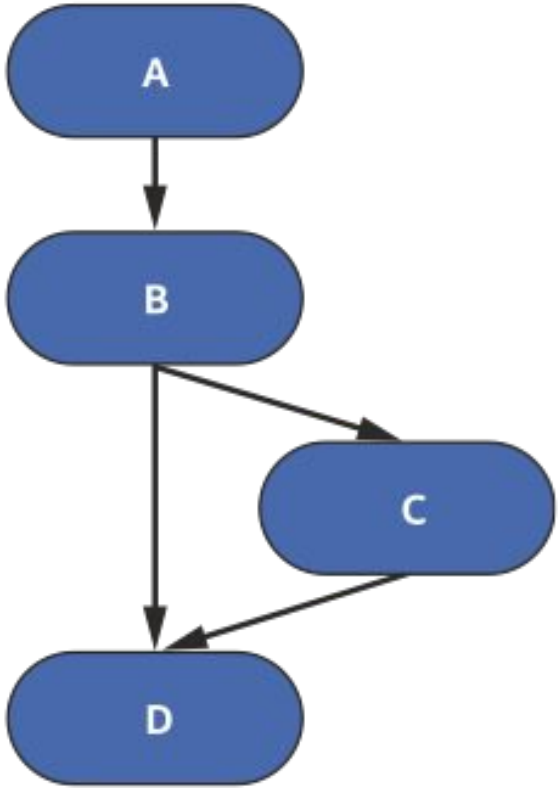
```
sycl::queue Q;

Q.submit([&](handler& h) {
  h.parallel_for(N, [=] (id<1> i) { /*...*/ });
});
Q.wait();
Q.submit([&](handler& h) {
  h.parallel_for(N, [=] (id<1> i) { /*...*/ });
});
Q.wait();
Q.submit([&](handler& h) {
  h.parallel_for(N, [=] (id<1> i) { /*...*/ });
Q.wait();
});
```



Créditos: Mastering DPC++ for Programming of Heterogeneous Systems using C++ and SYCL

Sincronismo no hospedeiro: eventos

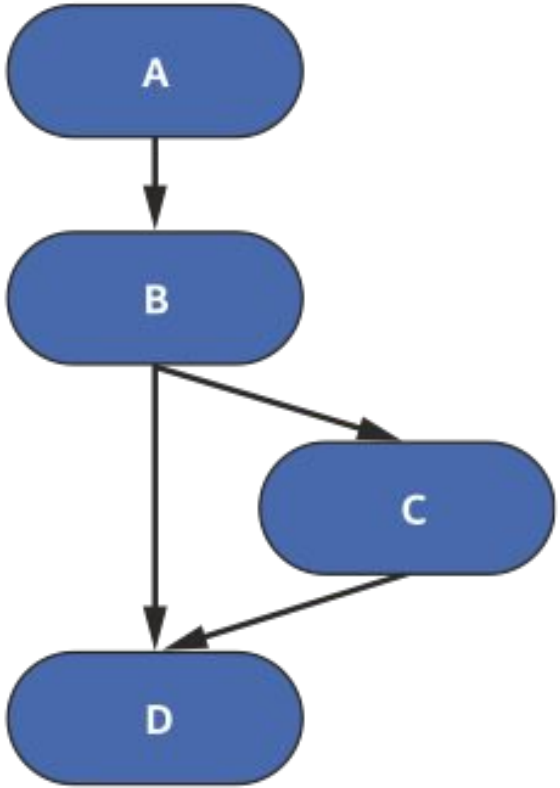
```
sycl::queue Q;

auto eA = Q.submit([&](handler &h) {
  h.parallel_for(N, [=] (id<1> i) { /*...*/ });
});
eA.wait();

auto eB = Q.submit([&](handler &h) {
  h.parallel_for(N, [=] (id<1> i) { /*...*/ });
});

auto eC = Q.submit([&](handler &h) {
  h.depends_on(eB);
  h.parallel_for(N, [=] (id<1> i) { /*...*/ });
});

auto eD = Q.submit([&](handler &h) {
  h.depends_on({eB,eC});
  h.parallel_for(N, [=] (id<1> i) { /*...*/ });

});
eD.wait();
```



Créditos: Mastering DPC++ for Programming of Heterogeneous Systems using C++ and SYCL

Sincronismo no hospedeiro: eventos

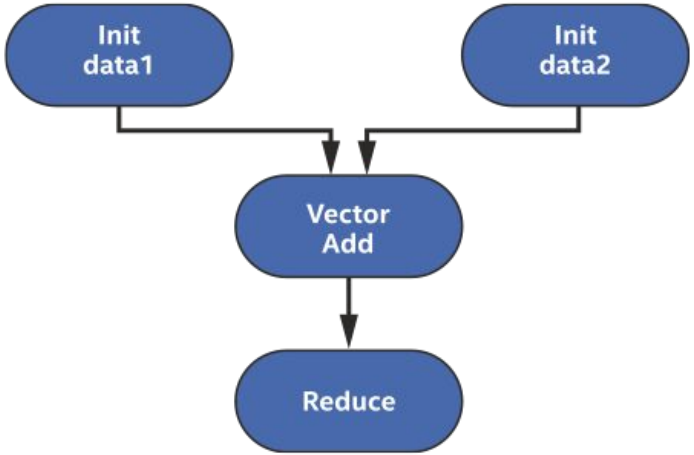
```
sycl::queue Q;

auto eA = Q.submit([&](handler &h) {
  h.parallel_for(N, [=] (id<1> i) { /*...*/ });
});
eA.wait();

auto eB = Q.submit([&](handler &h) {
  h.parallel_for(N, [=] (id<1> i) { /*...*/ });
});

auto eC = Q.submit([&](handler &h) {

  h.parallel_for(N, {eB}, [=] (id<1> i) { /*...*/ });
});

auto eD = Q.submit([&](handler &h) {

  h.parallel_for(N, {eB,eC}, [=] (id<1> i) { /*...*/ });

});
eD.wait();
```



Créditos: Mastering DPC++ for Programming of Heterogeneous Systems using C++ and SYCL

Sincronismo no hospedeiro: eventos

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

int main() {

  sycl::queue Q;

  auto *data1 = sycl::malloc_shared<int>(N, Q);
  auto *data2 = sycl::malloc_shared<int>(N, Q);

  auto e1 = Q.parallel_for(N, [=] (id<1> idx) { data1[idx] = 1; });
  auto e2 = Q.parallel_for(N, [=] (id<1> idx) { data2[idx] = 2; });
  auto e3 = Q.parallel_for(range{N}, {e1,e2}, [=] (id<1> idx) {
                           data1[idx] += data2[idx]; });

  Q.single_task(e3, [=] () {
    for (int i=1; i<N; i++)
      data1[0] += data1[i];
  });
  Q.wait();

  sycl::free(data1, Q);
  sycl::free(data2, Q);
  return 0;
}
```



Créditos: Mastering DPC++ for Programming of Heterogeneous Systems using C++ and SYCL

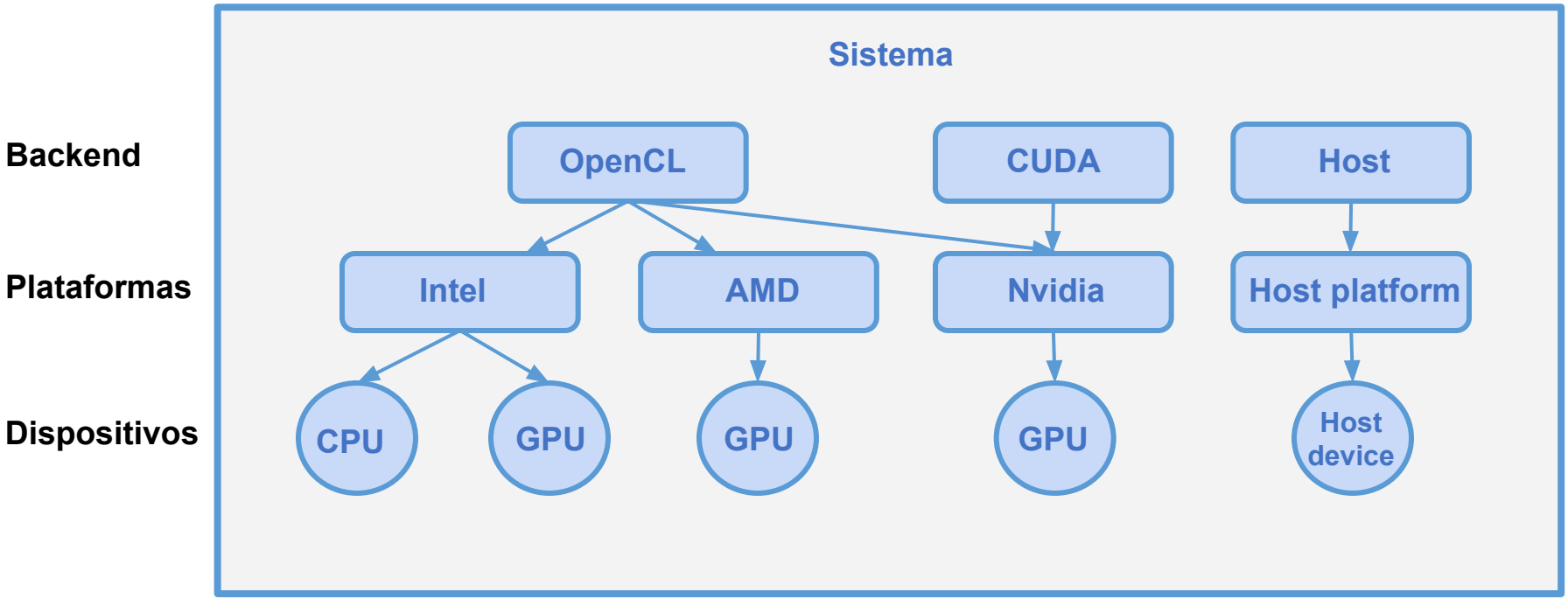
Tomada de tempo

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

int main() {

  sycl::queue Q{sycl::property::queue::enable_profiling()};

  auto *data1 = sycl::malloc_shared<int>(N, Q);

  auto e1 = Q.parallel_for(N, [=] (id<1> idx) { data1[idx] = 1; });

  auto ns = e1.get_profiling_info<sycl::info::event_profiling::command_end>()
          - e1.get_profiling_info<sycl::info::event_profiling::command_start>();

  return 0;
}
```

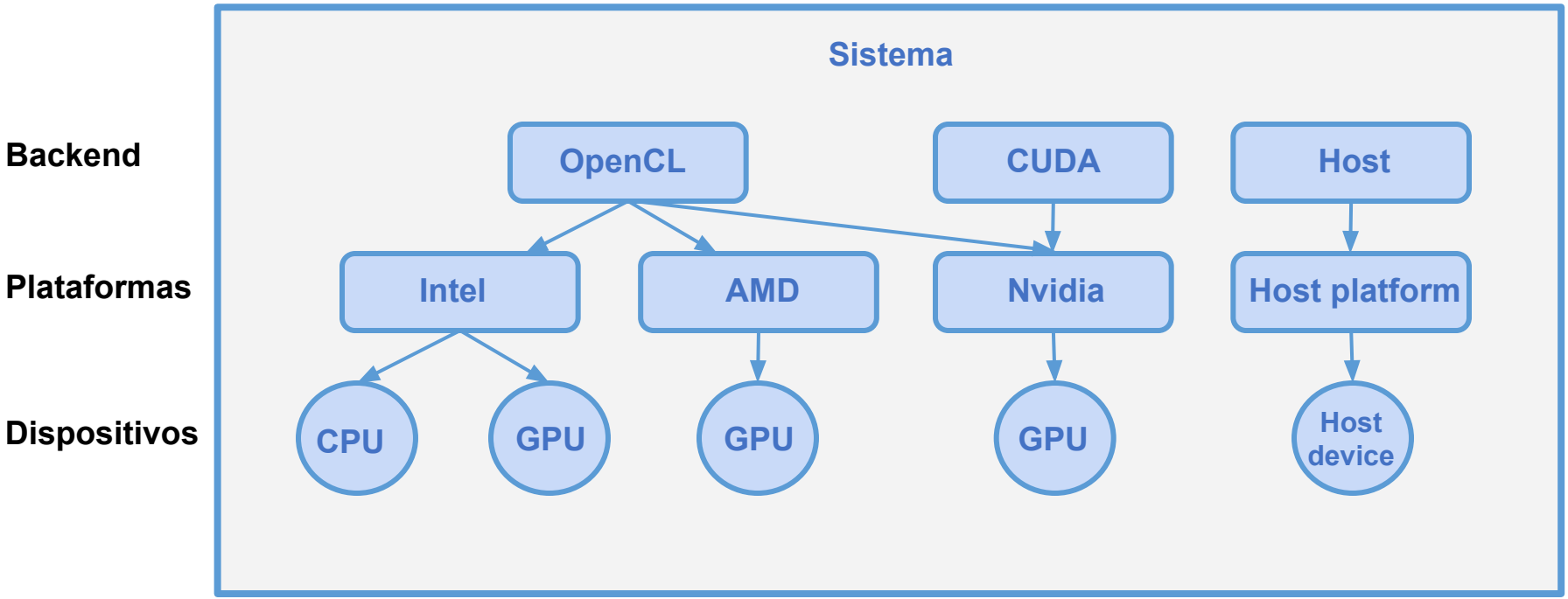
Para obter informações de *profiling* como, por exemplo, tempo de execução.

Barreira implícita aqui.

Plataformas e dispositivos

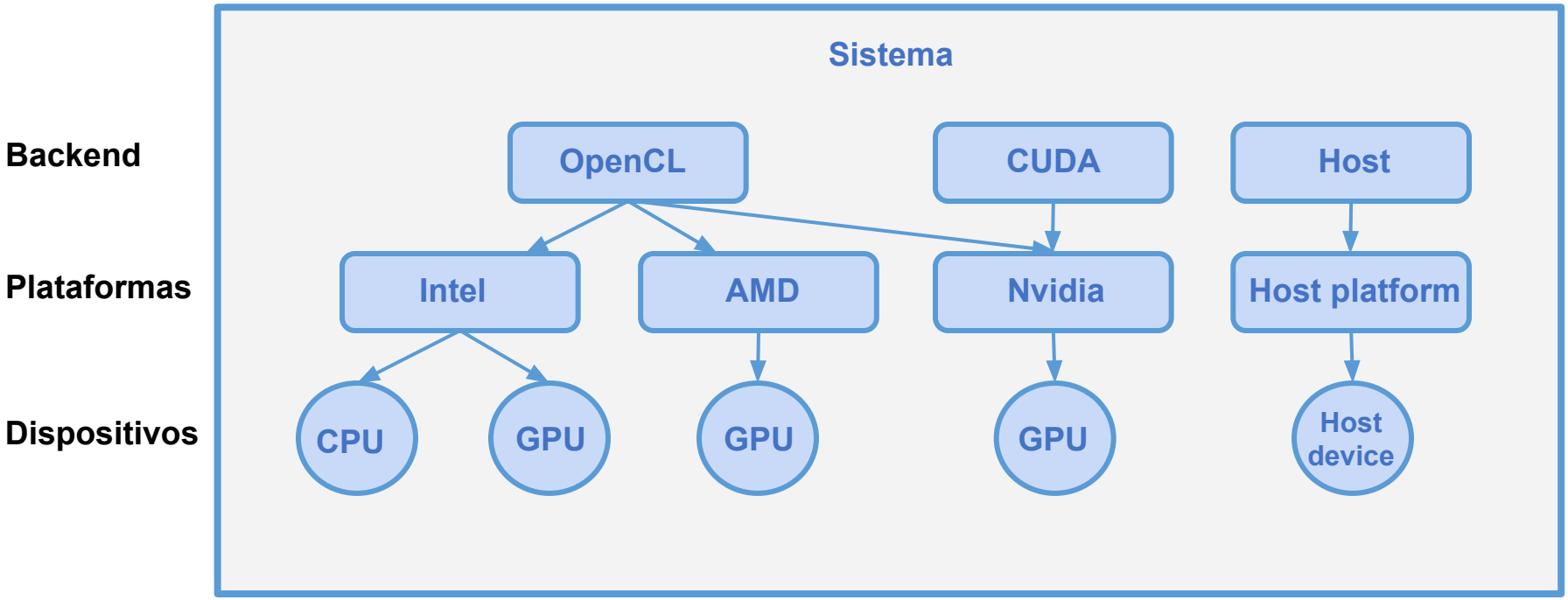


Plataformas e dispositivos



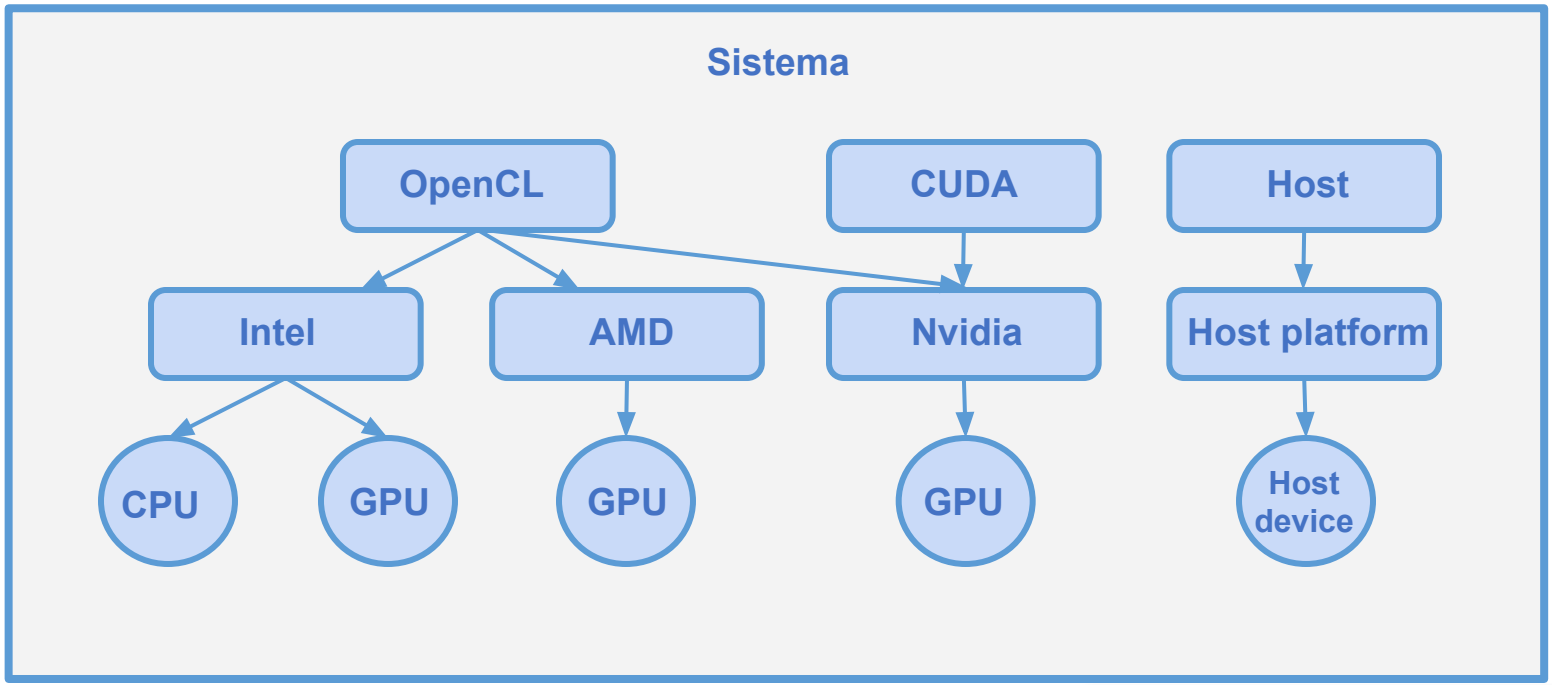

```
auto platforms = platform::get_platforms();

auto platName  = platforms[0].get_info<info::platform::name>();
                 platforms[0].get_info<info::platform::version>();
                 platforms[0].get_info<info::platform::vendor>();
                 platforms[0].get_info<info::platform::profile>();
```

Plataformas e dispositivos




```
auto platforms = platform::get_platforms();
auto devices   = platforms[0].get_devices();

auto devName = devices[0].get_info<info::device::name>();
               devices[0].get_info<info::device::version>();
               devices[0].get_info<info::device::vendor>();
               devices[0].get_info<info::device::device_type>();
```

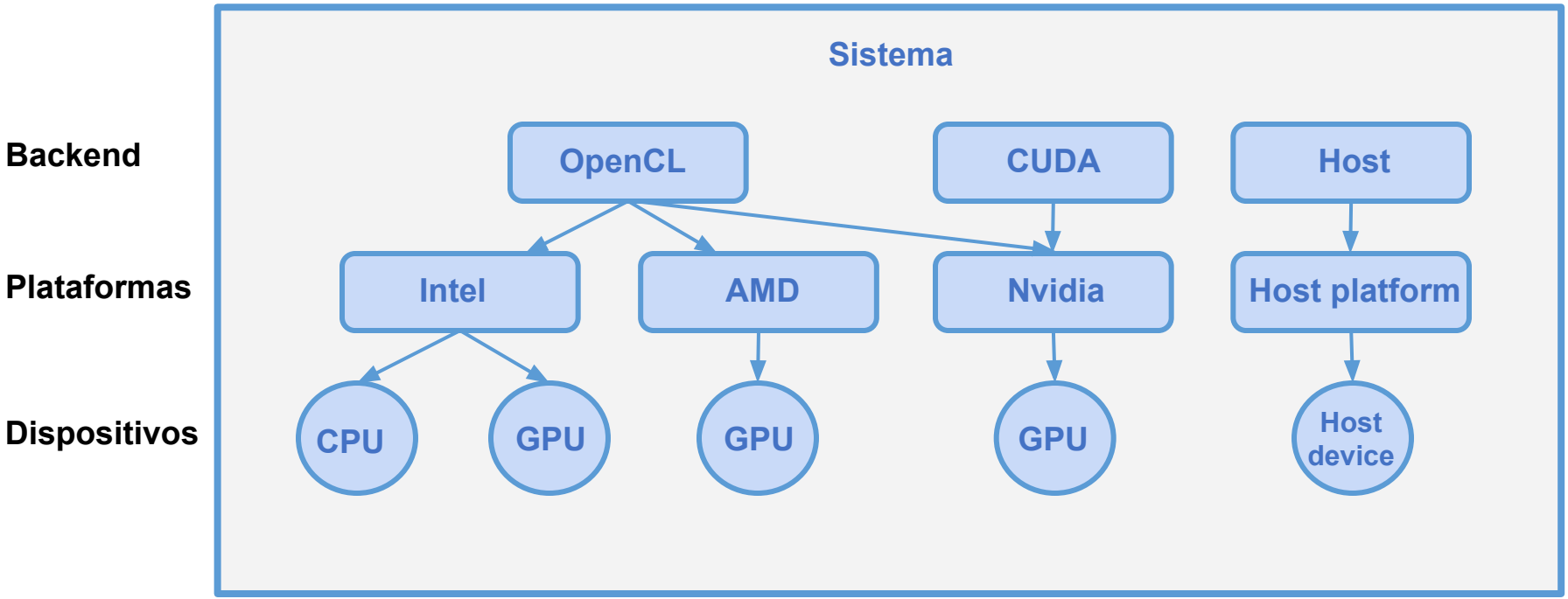
Plataformas e dispositivos




```
auto platforms = platform::get_platforms();
auto devices   = platforms[0].get_devices();

auto devName = devices[0].get_info<info::device::name>();
               devices[0].get_info<info::device::version>();
               devices[0].get_info<info::device::vendor>();
               devices[0].get_info<info::device::device_type>();
```

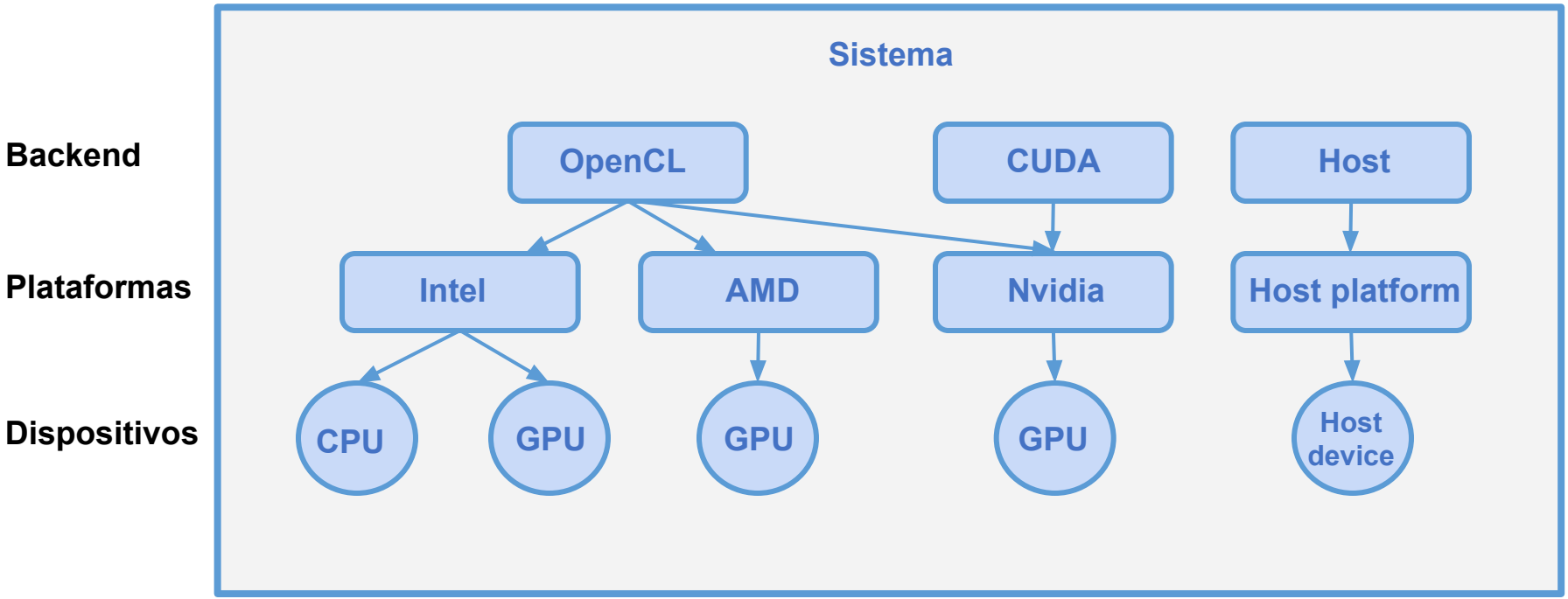
driver_version, double_fp_config, max_work_item_dimensions, max_work_group_size, max_compute_units, local_mem_size, mem_base_addr_align, partition_max_sub_devices

Plataformas e dispositivos




```
auto devices = device::get_devices();

auto devName = devices[0].get_info<info::device::name>();
               devices[0].get_info<info::device::version>();
               devices[0].get_info<info::device::vendor>();
               devices[0].get_platform().get_info<info::platform::vendor>()
```

Plataformas e dispositivos




```
auto devName = Q.get_device().get_info<info::device::name>();
               Q.get_device().get_info<info::device::version>();
               Q.get_device().get_info<info::device::vendor>();
               Q.get_device().get_info<info::device::device_type>();

auto devName = device{default_selector{}}.get_info<info::device::name>();
```

Plataformas e dispositivos

```
#include <CL/sycl.hpp>
#include <iostream>
using namespace sycl;

int main() {

  // Loop through the available platforms
  for(auto const& this_platform :platform::get platforms()) {
    std::cout << "Platform: " << this platform.get info<info::platform::name>()   << "\n";
    std::cout << "          " << this platform.get info<info::platform::vendor>() << "\n";
    std::cout << "          " << this_platform.get_info<info::platform::version>()<< "\n";

    // Loop through the devices available in this platform
    for(auto const& this_device :this platform.get devices()) {
      std::cout << "Device: "  <<this device.get info<info::device::name>()  << "\n";
      std::cout << "        "  <<this_device.get_info<info::device::vendor>()<< "\n";
      std::cout << "        is_host(): "        << (this device.is host()           ? "Yes" : "No") << "\n";
      std::cout << "        is_cpu(): "         << (this device.is cpu()            ? "Yes" : "No") << "\n";
      std::cout << "        is_gpu(): "         << (this device.is gpu()            ? "Yes" : "No") << "\n";
      std::cout << "        is_accelerator(): " << (this_device.is_accelerator()   ? "Yes" : "No") << "\n";
    }
    std::cout << "\n";
  }
  return 0;
}
```

Exercícios

1. Consultar propriedades das plataformas e dos dispositivos

2. Calcular a raiz quadrada de cada elemento de um vetor
   2.1. Usando *Unified Shared Memory* implícito
      2.1.1. Escolher outros dispositivos
   2.2. Usando *Unified Shared Memory* explícito
      2.2.1. Adicionar eventos
      2.2.2. Medir o tempo de execução através de eventos

3. Calcular a soma de dois vetores
   3.1. Usando *Unified Shared Memory* implícito

Instalação, compilação e execução

- A Intel lidera o desenvolvimento de um compilador SYCL 2020, o DPC++
  - Versão código aberto (*github*, há opção para suporte a GPUs NVIDIA):
    - github.com/intel/llvm
    - Adicione a flag `--cuda` durante o *configure* para suportar GPUs NVIDIA (tinyurl.com/y4xocyut)
  - Versão incluída no ecossistema *oneAPI* (comercial porém gratuita):
    - software.intel.com/content/www/us/en/develop/tools/oneapi/base-toolkit/download.html

- Compilação:
  - `source /opt/intel/oneapi/setvars.sh` (uma única vez por sessão)
  - `dpcpp codigo.cpp -o codigo`

- Execução:
  - `./codigo`