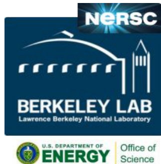
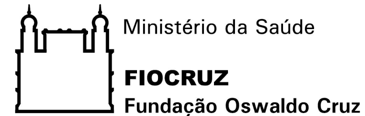
SYCL (II)

Amanda S. Dufek
asdufek@lbl.gov



Douglas A. Augusto
daa@fiocruz.br



2021-01-21

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**

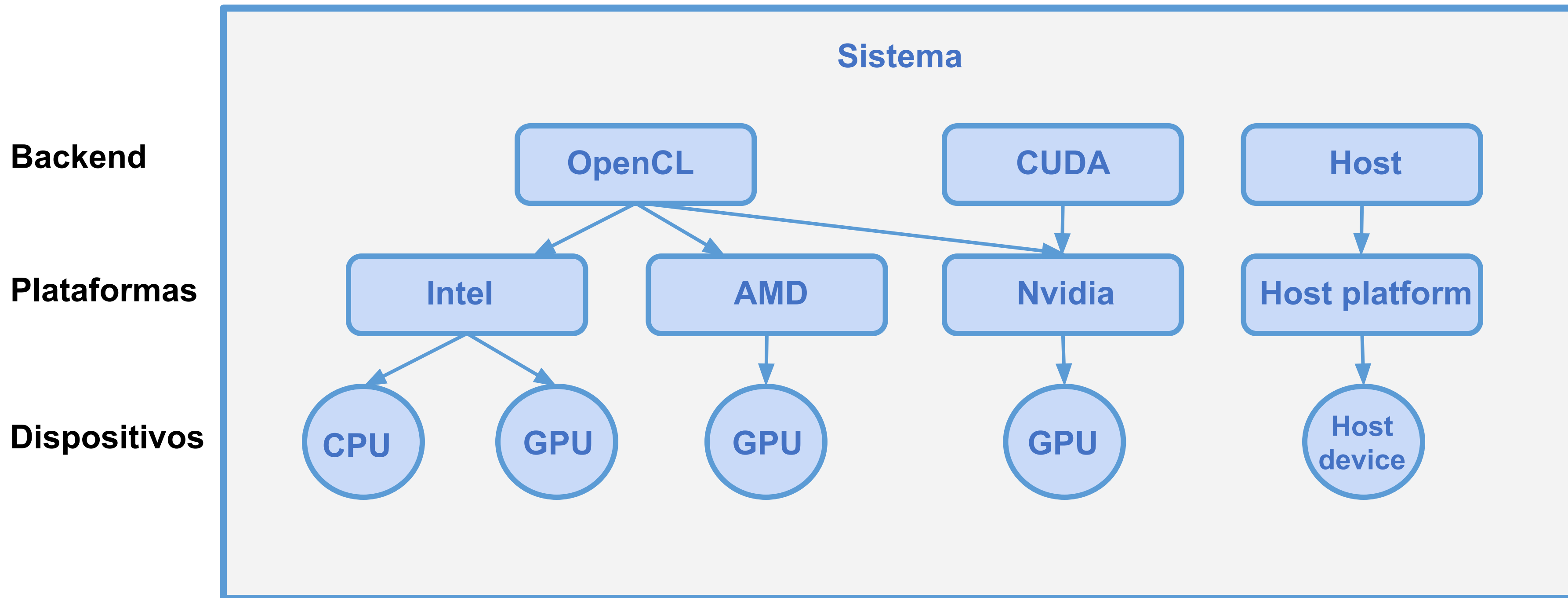
Ementa (2º dia)

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

Passos gerais para a programação

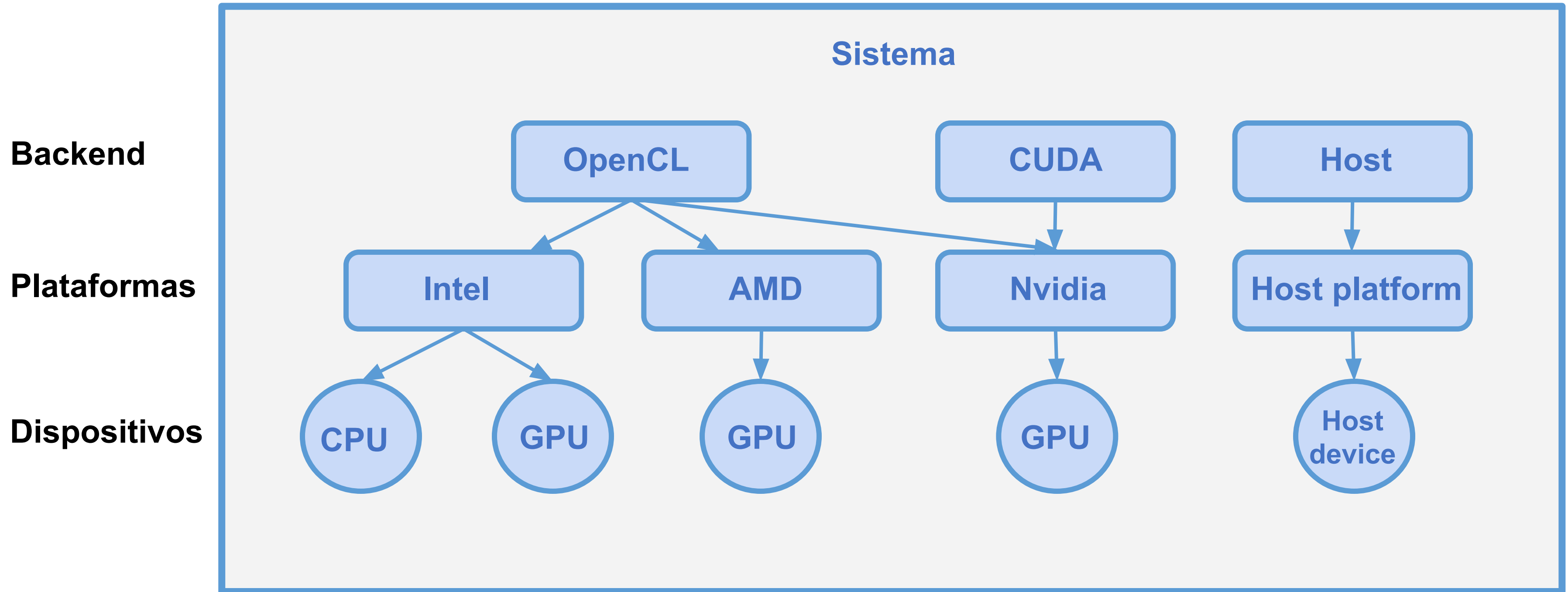
1. Escolhendo as plataformas e os dispositivos
2. Criando as filas de comandos para os dispositivos
3. Transferência dos dados para o dispositivo
4. Criando o kernel
5. Execução do kernel
6. Transferência dos resultados para o hospedeiro
7. Sincronismo
8. Liberação de memória

Plataformas e dispositivos



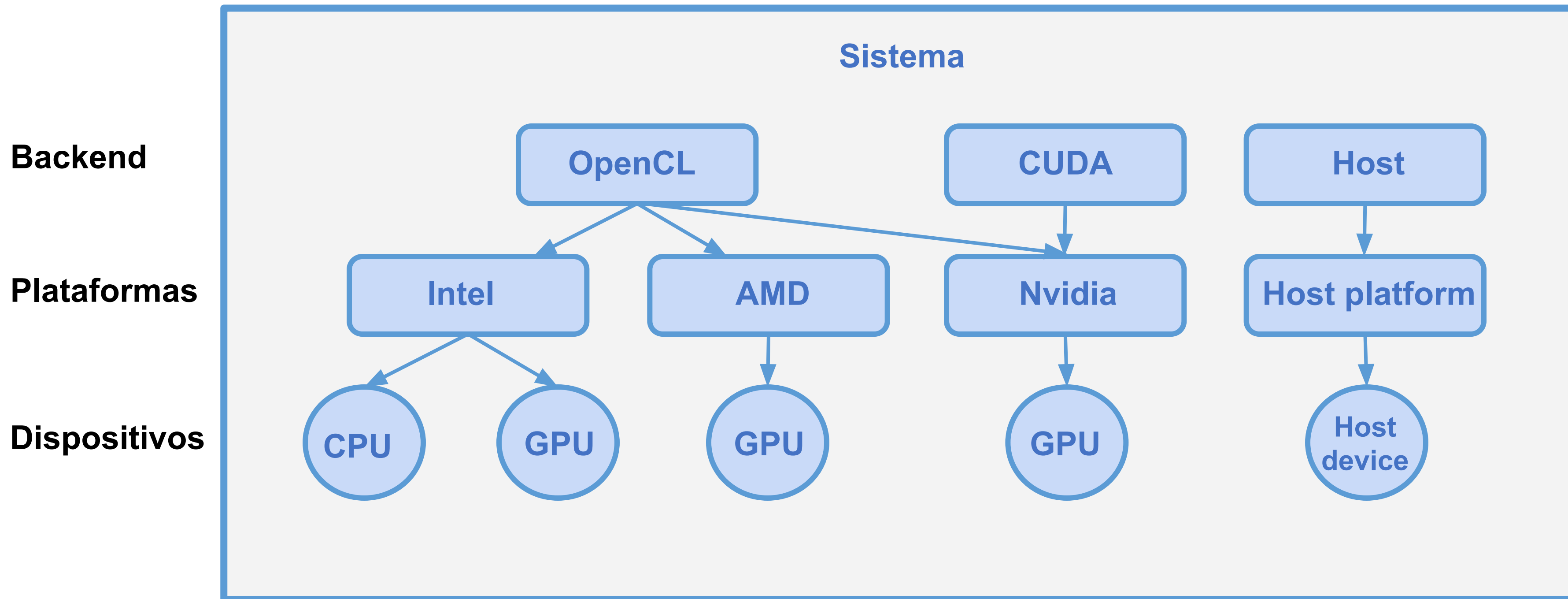

```
auto platforms = platform::get_platforms();

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



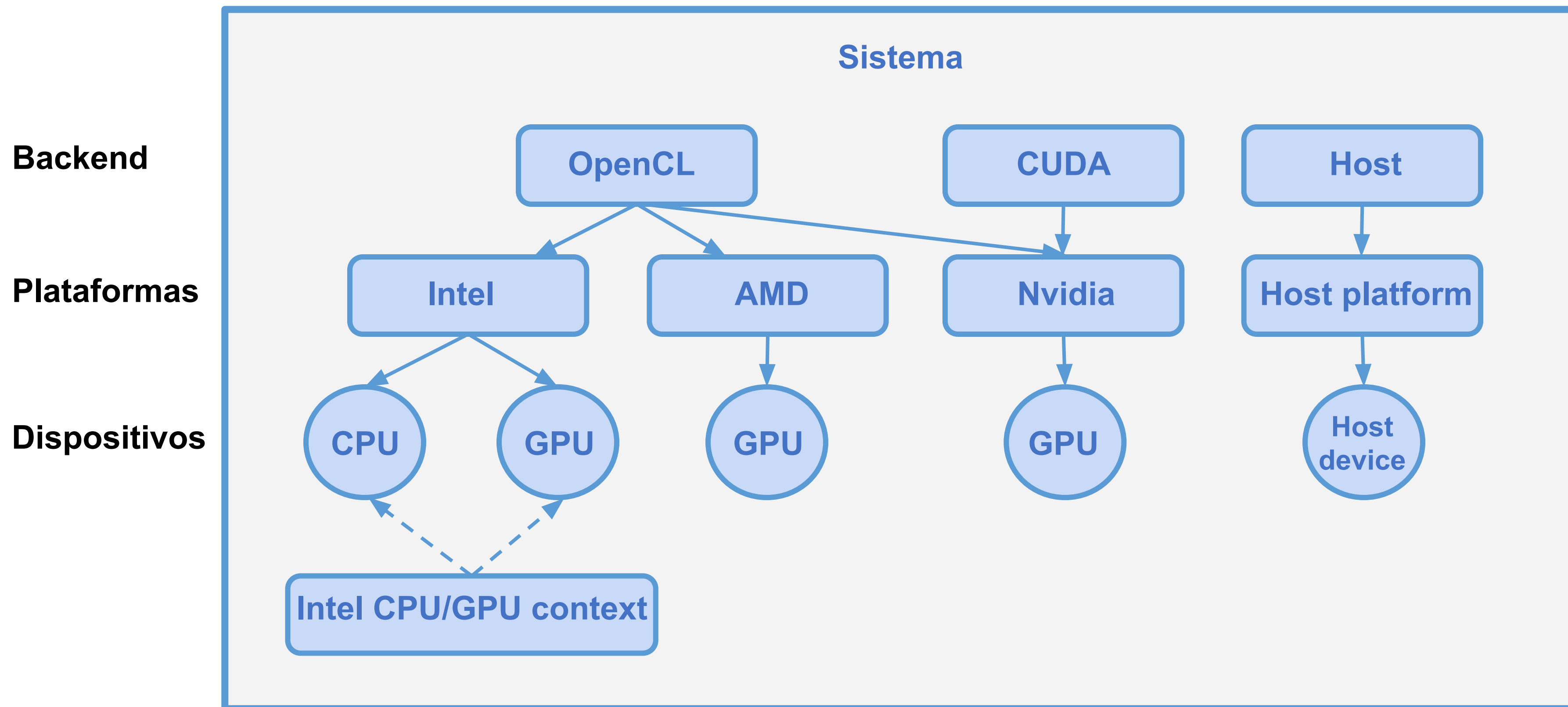

```
auto platforms = platform::get_platforms();
auto devices   = platforms[0].get_devices();

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



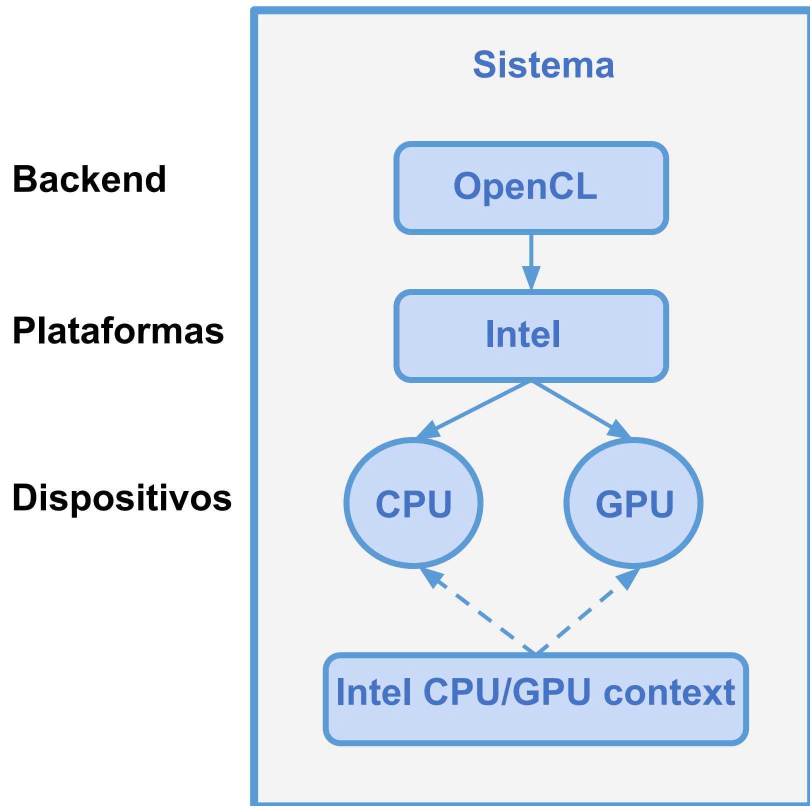

```
auto devName = Q.get_device().get_info<info::device::name>();
               Q.get_device().get_info<info::device::version>();
               Q.get_device().get_info<info::device::vendor>();
               Q.get_device().get_info<info::device::device_type>();

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

Plataformas e dispositivos



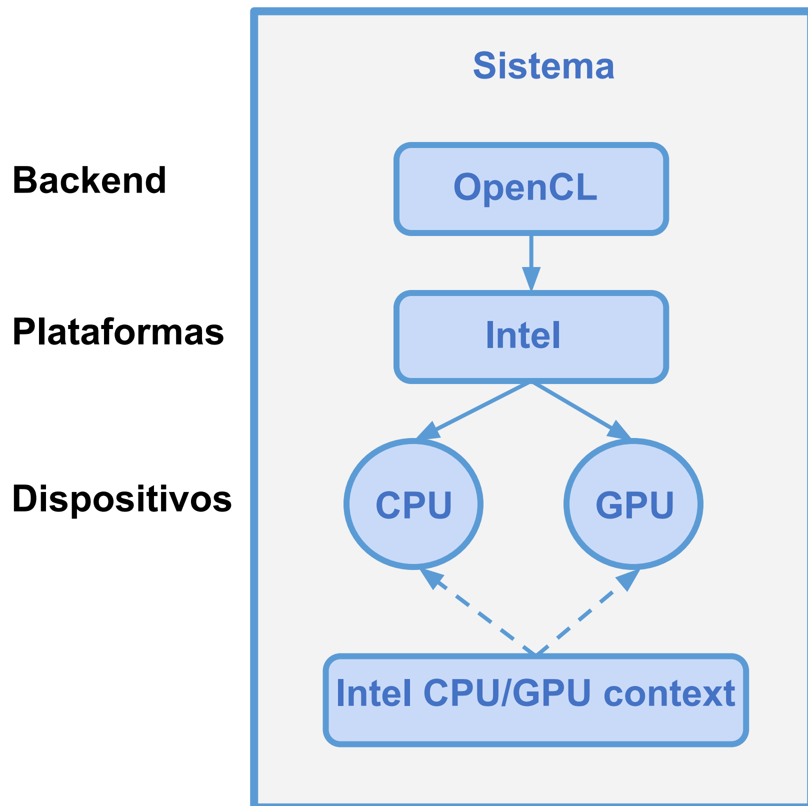
Um contexto representa um ou mais dispositivos pertencentes a uma mesma plataforma.
Uma fila está associada a um dispositivo de um contexto.
O contexto é criado de forma implícita.

Plataformas e dispositivos




```
queue cpu_fila{cpu_selector{}};
queue gpu_fila{gpu_selector{}};
```

Plataformas e dispositivos




```
queue cpu_fila{cpu_selector{}};
queue gpu_fila{cpu_fila.get_context(),gpu_selector{}};

std::cout << cpu_fila.get_context().get() << std::endl;
std::cout << gpu_fila.get_context().get() << std::endl;
```

Plataformas e dispositivos

```
class custom_selector : public device_selector {
  public:
    custom_selector() : device_selector() {}

    int operator()(const device& device) const override {

      std::string vendor_name = device.get_platform().get_info<info::platform::vendor>();

      //Selecting Intel platform
      if (vendor_name.find("Intel") != std::string::npos)

         //Selecting GPU device
         if (device.get_info<info::device::device_type>() == info::device_type::gpu) { return 100; }

      return -1;

    }
};


int main()
{
   queue gpu_fila{custom_selector{}};

   ...

}
```

Passos gerais para a programação

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

Gerenciamento de dados: *buffers* implícito

- ***Buffers*** são objetos de memória criados para gerenciar a movimentação de dados entre hospedeiro e dispositivo.
- Inicialmente o *buffer* apenas aponta para a memória do hospedeiro.
- *Buffers* podem ser criados a partir de dados já existentes ou não no hospedeiro.

Créditos: https://github.com/codeplaysoftware/syclacademy/tree/main/Lesson_Materials

Gerenciamento de dados: *buffers* implícito

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

int main() {

  sycl::queue Q;

  sycl::buffer<int> buffer{sycl::range{N}};

  ...
  return 0;
}
```

Gerenciamento de dados: *buffers* implícito

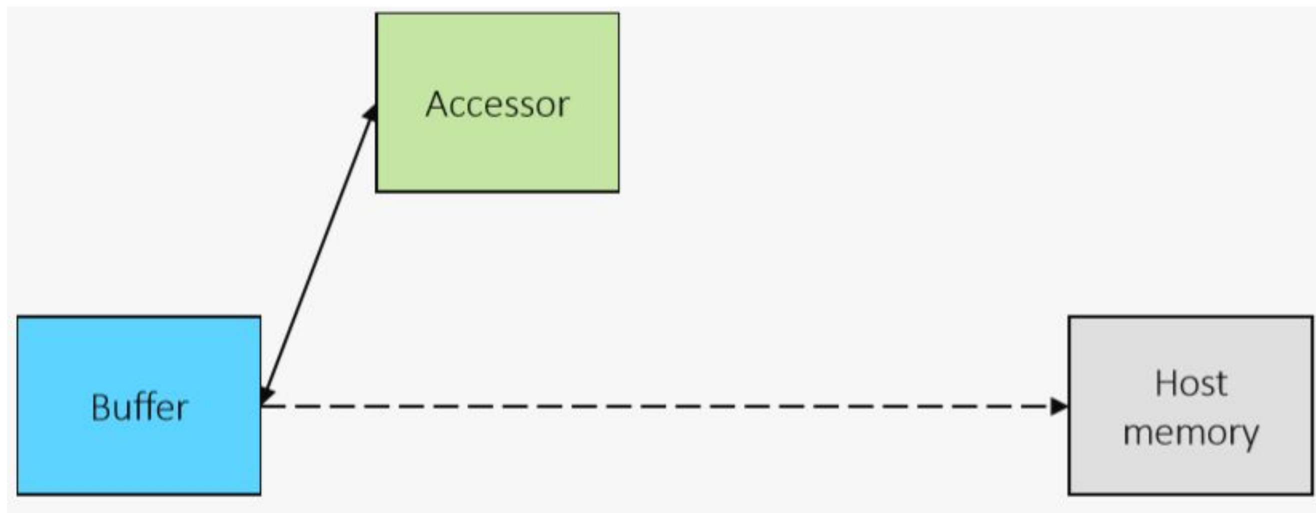
```
#include <CL/sycl.hpp>
#include<array>

int main() {

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

  sycl::queue Q;

  sycl::buffer buffer{array};

  ...
  return 0;
}
```

Gerenciamento de dados: *buffers* implícito

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

int main() {

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

  sycl::queue Q;

  sycl::buffer<int,1> buffer{ array.data(), sycl::range<1>{array.size()} };

  ...
  return 0;
}
```

SYCL 1.2.1

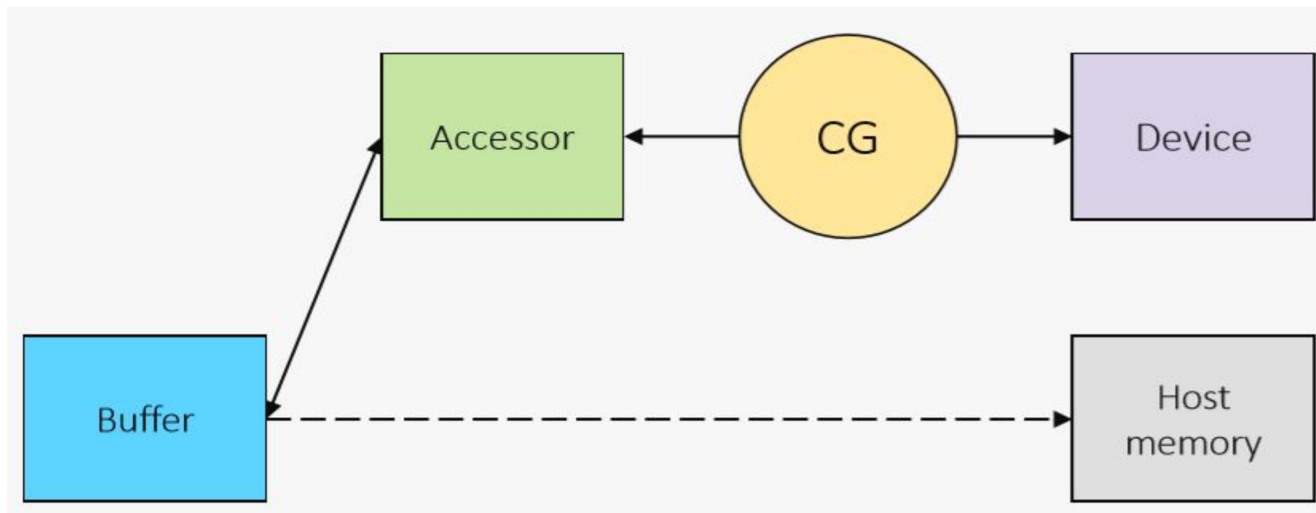
Gerenciamento de dados: *buffers* implícito



- *Buffers* não podem ser acessados diretamente pelo hospedeiro nem pelo dispositivo, porém são acessíveis no hospedeiro ou em qualquer dispositivo através dos objetos ***accessors***.
- ***Accessors*** solicitam o acesso aos *buffers*, permitindo assim a leitura e escrita nos *buffers*.

Créditos: https://github.com/codeplaysoftware/syclacademy/tree/main/Lesson_Materials

Gerenciamento de dados: *buffers* implícito



Créditos: https://github.com/codeplaysoftware/syclacademy/tree/main/Lesson_Materials

Gerenciamento de dados: *buffers* implícito

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

int main() {

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

  sycl::queue Q;

  {
    sycl::buffer buffer{array};

    Q.submit([&](handler &h) {
      sycl::accessor accessor{buffer, h};
      ...
    });
  }

  ...
  return 0;
}
```

O modo de acesso ***default*** é ***read_write***.

Gerenciamento de dados: *buffers* implícito

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

int main() {

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

  sycl::queue Q;

  {
    sycl::buffer buffer{array};

    Q.submit([&](handler &h) {
      sycl::accessor accessor{buffer, h, read_only,  global_buffer};
      ...
    });
  }

  ...
  return 0;
}
```

read_write (implícito)
read_only
write_only

global_buffer (implícito)
constant_buffer
local
host_buffer

Gerenciamento de dados: *buffers* implícito

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

int main() {

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

  sycl::queue Q;

  {
    sycl::buffer buffer{array};

    Q.submit([&](handler &h) {
      sycl::accessor<int, 1, sycl::access::mode::read,
                     sycl::access::target::global_buffer> accessor(buffer, h);
      ...
    });
  }

  ...
  return 0;
}
```

read_write (implícito)
read
write

global_buffer (implícito)
constant_buffer
local
host_buffer

SYCL 1.2.1

Gerenciamento de dados: *buffers* implícito

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

int main() {

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

  sycl::queue Q;

  {
    sycl::buffer buffer{array};

    Q.submit([&](handler &h) {
      auto accessor = buffer.get_access<read_write, global_buffer>(h);

        ...
    });
    ...
  }

  ...
  return 0;
}
```

read_write (implícito)
read_only
write_only

global_buffer (implícito)
constant_buffer
local
host_buffer

Gerenciamento de dados: *buffers* implícito

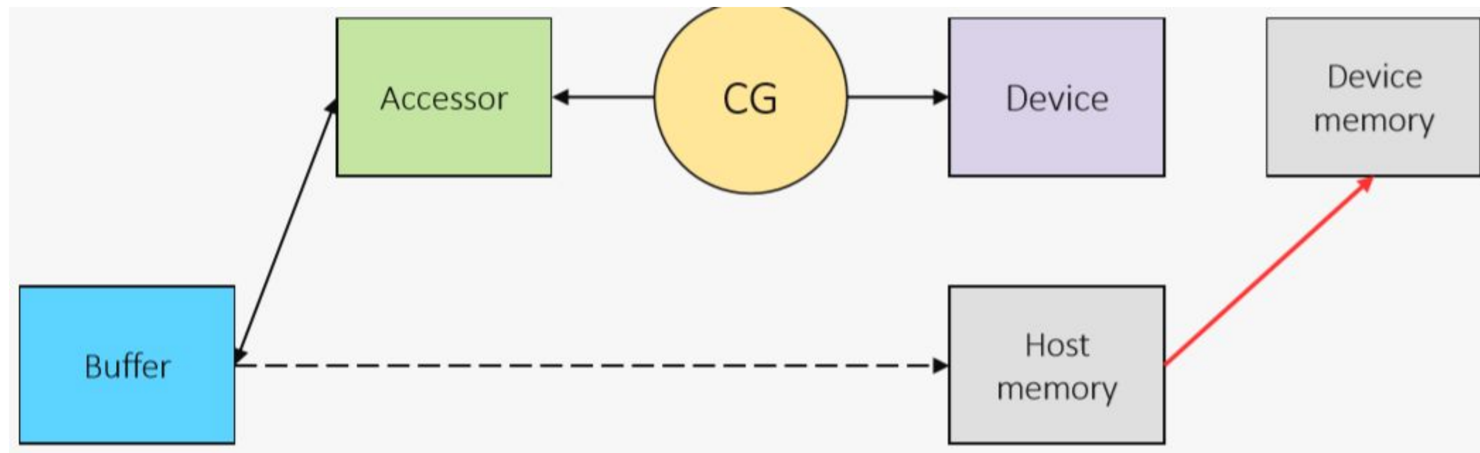
```
#include <CL/sycl.hpp>
#include<array>

int main() {

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

  sycl::queue Q;

  {
    sycl::buffer buffer{array};

    Q.submit([&](handler &h) {
      auto accessor = buffer.get_access<sycl::access::mode::read_write,
                                        sycl::access::target::global_buffer>(h);

      ...
    });
    ...
  }

  ...
  return 0;
}
```

read_write (implícito)
read
write

global_buffer (implícito)
constant_buffer
local
host_buffer

SYCL 1.2.1

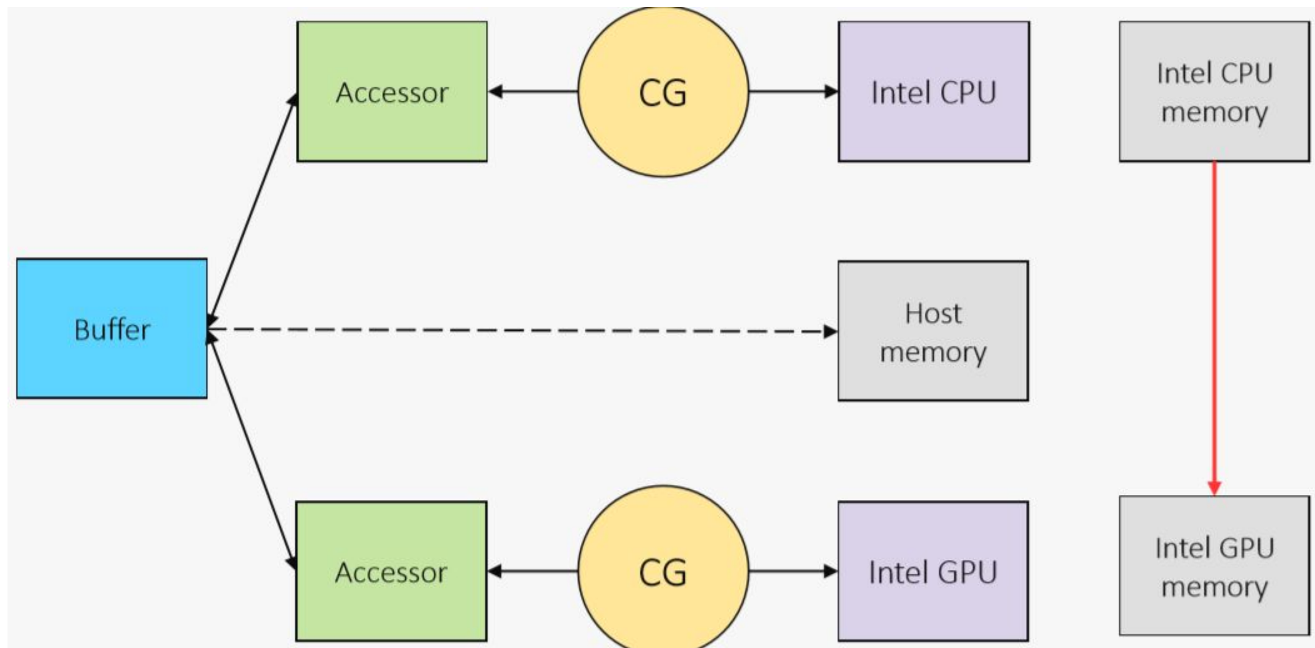
Gerenciamento de dados: *buffers* implícito



Os dados permanecerão no dispositivo até que outro grupo de comando solicite acesso aos dados em um outro dispositivo ou no hospedeiro.

Créditos: https://github.com/codeplaysoftware/syclacademy/tree/main/Lesson_Materials

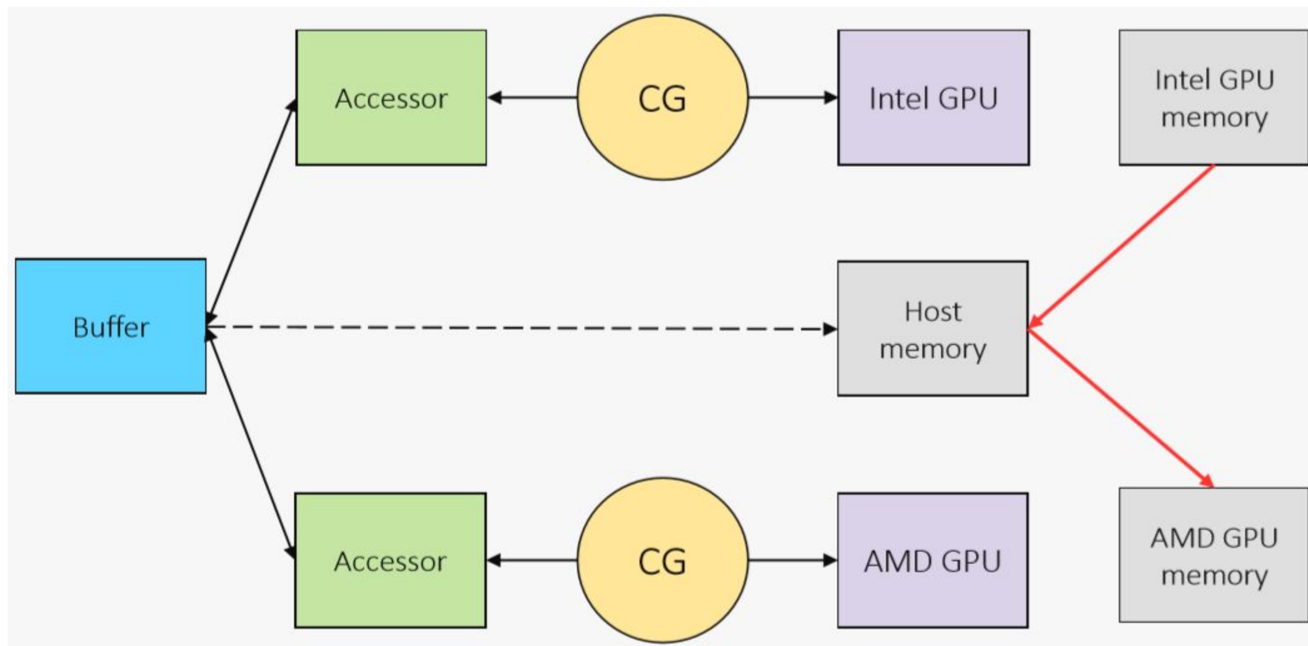
Gerenciamento de dados: *buffers* implícito



Se os dispositivos pertencem ao mesmo contexto, a cópia é direta.

Créditos: https://github.com/codeplaysoftware/syclacademy/tree/main/Lesson_Materials

Gerenciamento de dados: *buffers* implícito



Caso contrário, a cópia acontece via memória do hospedeiro.

Créditos: https://github.com/codeplaysoftware/syclacademy/tree/main/Lesson_Materials

Gerenciamento de dados: *buffers* implícito

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

int main() {

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

  sycl::queue Q;

  {
    sycl::buffer buffer{array};

    Q.submit([&](handler &h) {
      sycl::accessor accessor{buffer, h};
      ...
    });
    sycl::host_accessor host_accessor{buffer};
    ...
  }

  ...
  return 0;
}
```

Host accessors são bloqueantes: a execução no hospedeiro não prossegue enquanto o dado não estiver disponível no hospedeiro, e o *buffer* não pode ser usado em nenhum dispositivo enquanto o *host_accessor* não for destruído.

Gerenciamento de dados: *buffers* implícito

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

int main() {

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

  sycl::queue Q;

  {
    sycl::buffer buffer{array};

    Q.submit([&](handler &h) {
      sycl::accessor accessor{buffer, h};
      ...
    });
    sycl::host_accessor host_accessor{buffer, read_only};
    ...
  }

  ...
  return 0;
}
```

read_write (implícito)
read_only
write_only

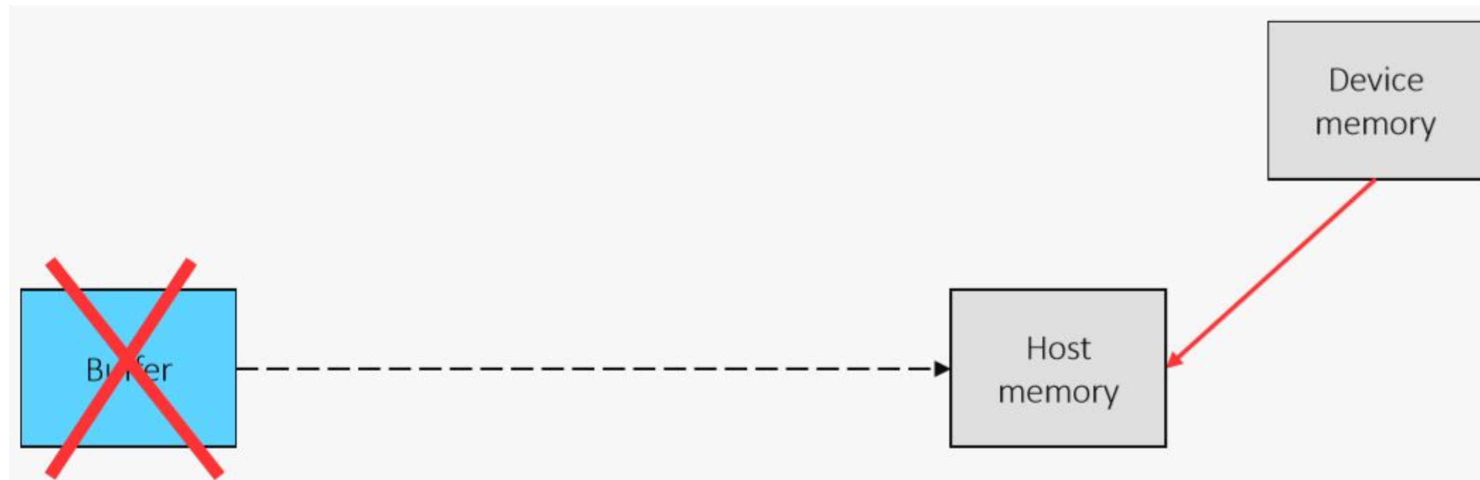
Gerenciamento de dados: *buffers* implícito

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

int main() {

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

  sycl::queue Q;

  {
    sycl::buffer buffer{array};

    Q.submit([&](handler &h) {
      sycl::accessor accessor{buffer, h};
      ...
    });
    sycl::accessor<int, 1, sycl::access::mode::read_write,
                           sycl::access::target::host_buffer> host_accessor(buffer);
    ...
  }

  ...
  return 0;
}
```

read_write (implícito)
read
write

SYCL 1.2.1

Gerenciamento de dados: *buffers* implícito



- O objeto *buffer* só será destruído após o término de todos os trabalhos atrelados aos dados e, antes da destruição, os dados serão copiados para a memória original do hospedeiro.
- A destruição dos *buffers* gera um ponto de sincronização.

Créditos: https://github.com/codeplaysoftware/syclacademy/tree/main/Lesson_Materials

Gerenciamento de dados: *buffers* implícito

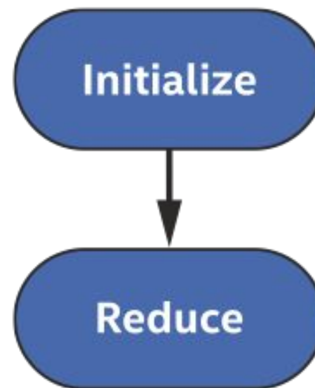
```
#include <CL/sycl.hpp>
#include<array>

int main() {

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

  sycl::queue Q;

  {
    sycl::buffer buffer{array};

    Q.submit([&](handler &h) {
      sycl::accessor accessor{buffer, h};
      ...
    });
    sycl::host_accessor host_accessor{buffer};
    ...
  }

  ...
  return 0;
}
```

Todos os *buffers* sincronizam e são destruídos ao término do escopo. Os dados serão transferidos para o vetor *array* ao término do escopo.

Análise de dependências

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

int main() {

  sycl::queue Q;

  sycl::buffer<int> buffer{range{N}};

  Q.submit([&] (handler &h) {
                sycl::accessor   a{buffer,   h,
write_only};
    h.parallel_for(N, [=] (id<1> idx) {
      a[idx] = 1;
    });
  });

  Q.submit([&] (handler &h) {
                sycl::accessor   a{buffer,   h,
read_write};
    h.single_task([=] () {
      for (int i=1; i<N; i++)
        a[0] += a[i];
    });
  });

  sycl::host_accessor h_a{buffer};
  return 0;
}
```

Executa kernel **"Initialize"**	Executa kernel **"Reduce"**	Copia buffer para o hospedeiro



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

Análise de dependências

```
sycl::queue Q;

sycl::buffer<int> data1{range{N}};
sycl::buffer<int> data2{range{N}};

Q.submit([&] (handler &h) {
  sycl::accessor a{data1, h, write_only};
  h.parallel_for(N, [=] (id<1> idx) { a[idx] = 1; });
});

Q.submit([&] (handler &h) {
  sycl::accessor b{data2, h, write_only};
  h.parallel_for(N, [=](id<1> idx) { b[idx] = 2; });
});

Q.submit([&] (handler &h) {
  sycl::accessor a{data1, h, read_write};
  sycl::accessor b{data2, h, read_only};
   h.parallel_for(N, [=](id<1> idx) { a[idx] += b[idx];
});
});

Q.submit([&] (handler &h) {
  sycl::accessor a{data1, h, read_write};
  h.single_task([=] () {
    for (int i=1; i<N; i++)
      a[0] += a[i];
  });
});
sycl::host_accessor h_a{data1};
```

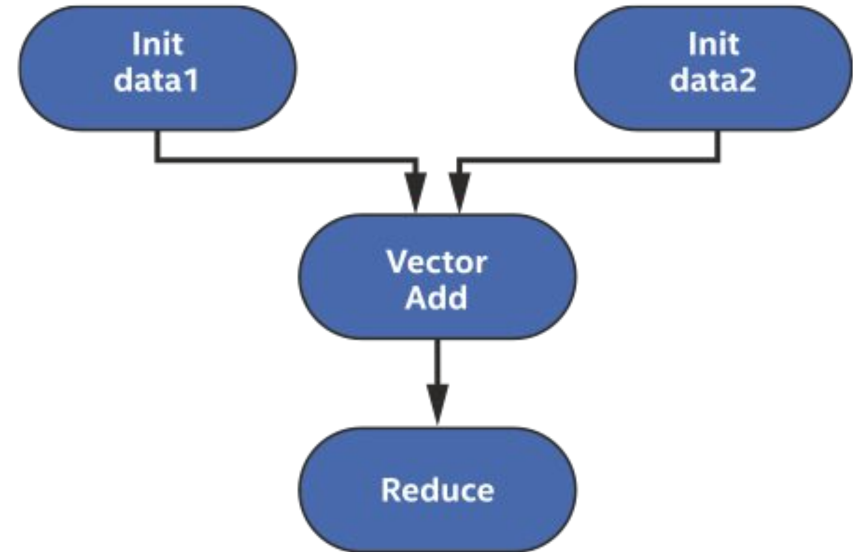
Executa kernel “Init data1”

Executa kernel “Init data2”

Executa kernel “Vector Add”

Executa kernel “Reduce”

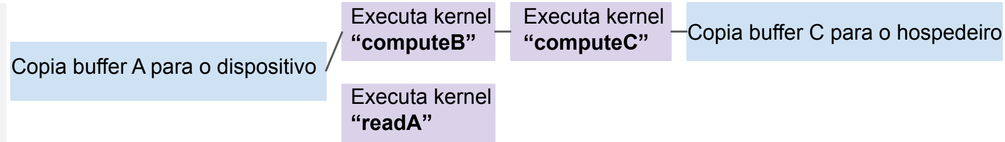
Copia buffer para o hospedeiro

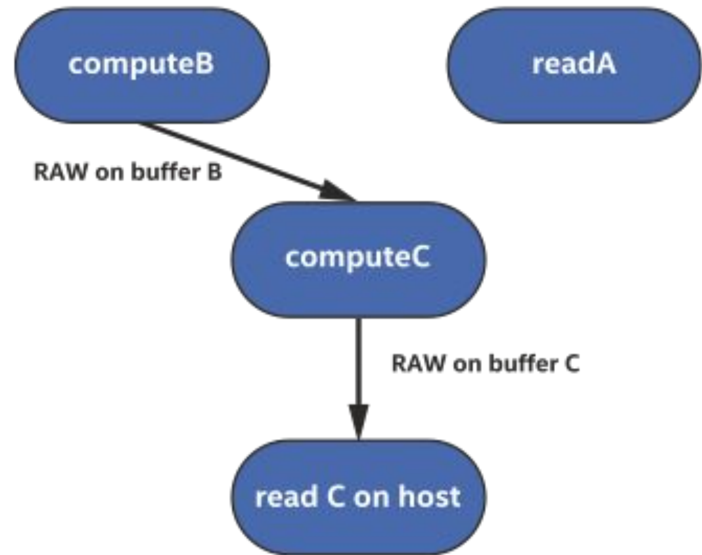


Créditos: Mastering DPC++ for Programming of Heterogeneous Systems using C++ and SYCL

Análise de dependências

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

int main() {

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

  sycl::queue Q;

  sycl::buffer A{a}, B{b}, C{c};

  Q.submit([&](handler &h) {
      sycl::accessor accA(A, h, read_only);
      sycl::accessor accB(B, h, write_only);
      h.parallel_for(N, [=] (id<1> i) { accB[i] = accA[i] + 1;
});
  });

  Q.submit([&](handler &h) {
      sycl::accessor accA(A, h, read_only);
      h.parallel_for(N, [=] (id<1> i) {  int data = accA[i]; });
  });

  Q.submit([&](handler &h) {
      sycl::accessor accB(B, h, read_only);
      sycl::accessor accC(C, h, write_only);
      h.parallel_for(N, [=] (id<1> i) { accC[i] = accB[i] + 2;
});
  });

  sycl::host_accessor host_accC(C, read_only);

  return 0;
```

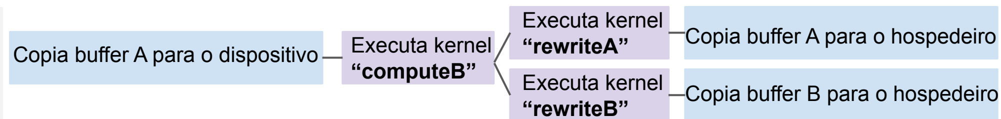


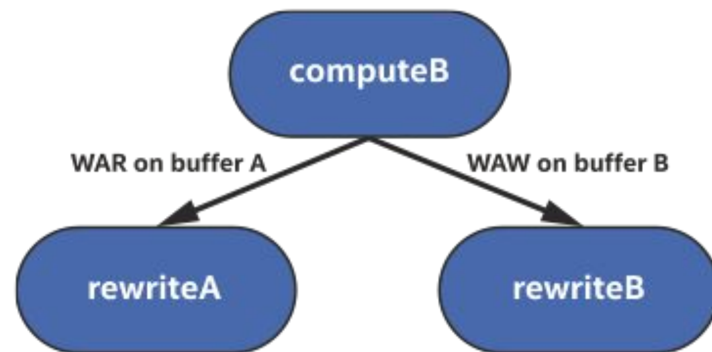


Créditos: Mastering DPC++ for Programming of Heterogeneous Systems using C++ and SYCL

Análise de dependências

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

int main() {

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

  sycl::queue Q;

  sycl::buffer A{a}, B{b};

  Q.submit([&](handler &h) {
      sycl::accessor accA(A, h, read_only);
      sycl::accessor accB(B, h, write_only);
      h.parallel_for(N, [=] (id<1> i) { accB[i] = accA[i] + 1;
});
  });

  Q.submit([&](handler &h) {
      sycl::accessor accA(A, h, write_only);
      h.parallel_for(N, [=] (id<1> i) {  accA[i] = 42; });
  });

  Q.submit([&](handler &h) {
      sycl::accessor accB(B, h, write_only);
      h.parallel_for(N, [=] (id<1> i) { accB[i] = 52; });
  });

  Q.wait();

  return 0;
}
```





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

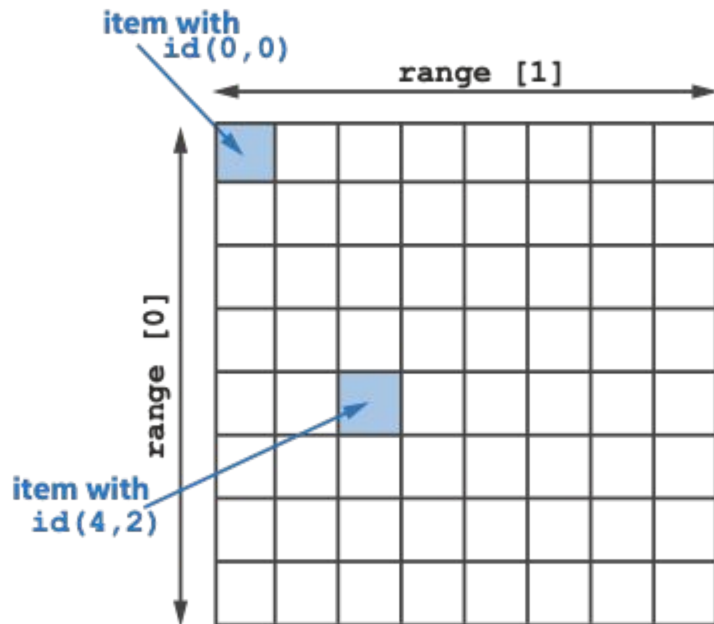
Modelo de execução

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

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];
});
```

Item de trabalho (***work-item***): o número total de itens de trabalho é chamado de ***global range***.



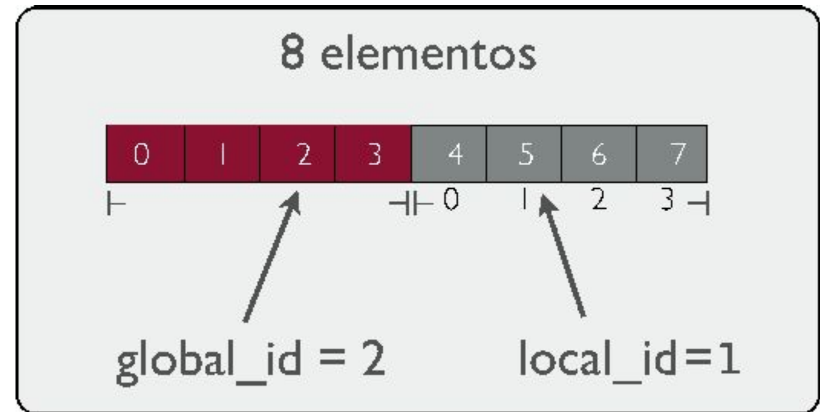
Créditos: Mastering DPC++ for Programming of Heterogeneous Systems using C++ and SYCL

Modelo de execução

Grupos de trabalho (***work-group***):

- Os itens de trabalho podem ser divididos em grupos de trabalho.
- O número de itens de trabalho de um mesmo grupo é chamado de ***local range***.
- Itens de trabalho de um mesmo grupo podem se comunicar eficientemente e sincronizar.
- Diferentes grupos de trabalho são executados independentemente.

Cada item de trabalho tem um identificador local e global.

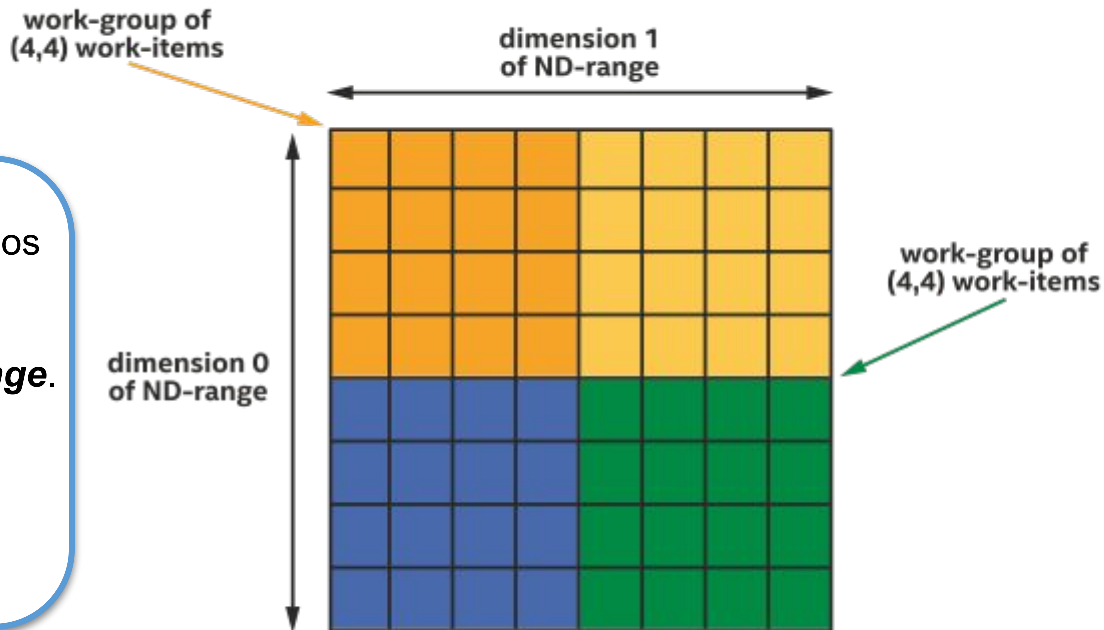


global range = 8
local range ou *group size* = 4
group range ou *num groups* = *global range* / *local range* = 2

Modelo de execução

Grupos de trabalho (***work-group***):

- Os itens de trabalho podem ser divididos em grupos de trabalho.
- O número de itens de trabalho de um mesmo grupo é chamado de ***local range***.
- Itens de trabalho de um mesmo grupo podem se comunicar eficientemente e sincronizar.
- Diferentes grupos de trabalho são executados independentemente.

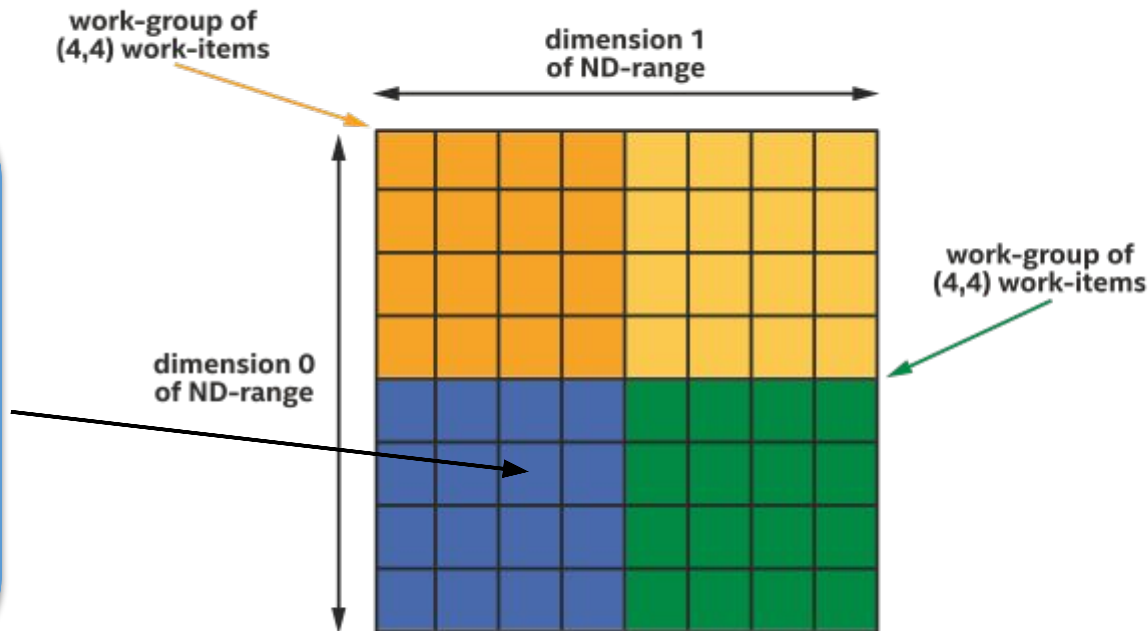


global range = (8,8)
local range ou *group size* = (4,4)
group range ou *num groups* = *global range* / *local range* = (2,2)

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

Modelo de execução

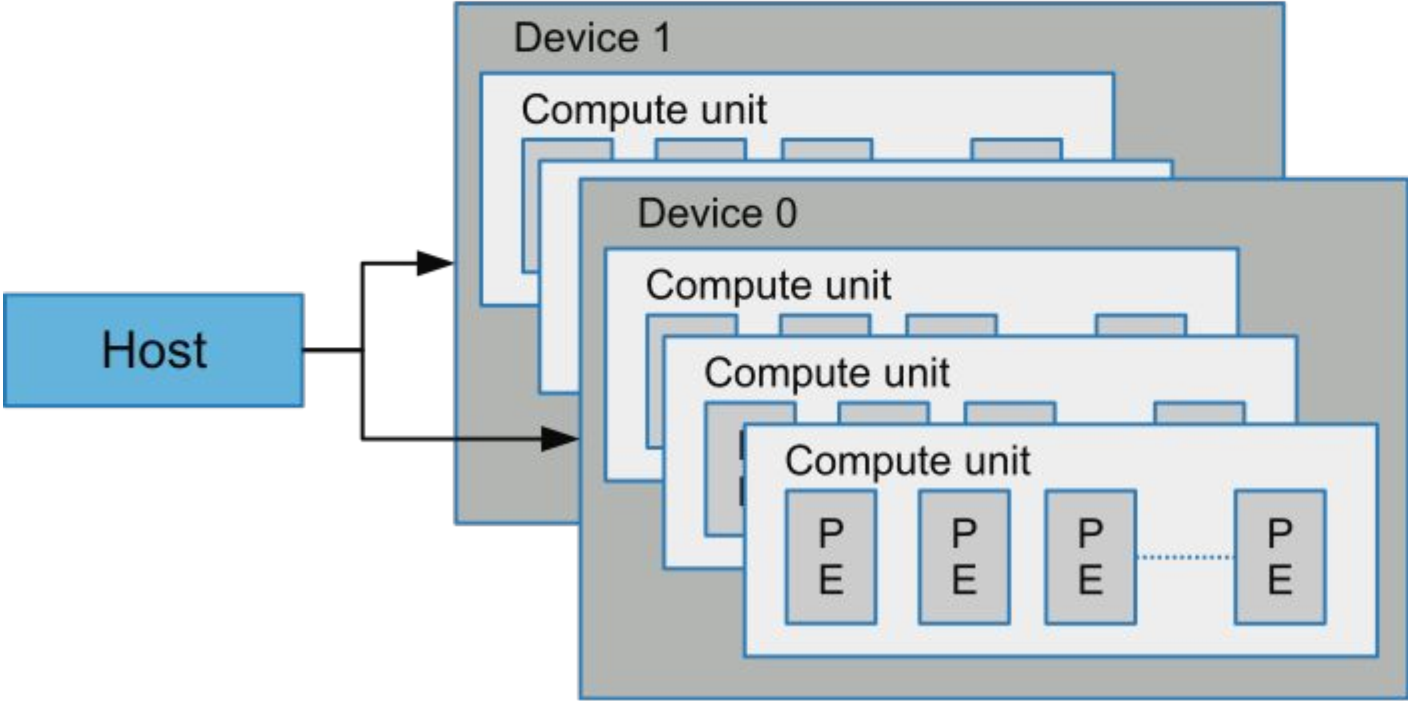
get_global_range(0): 8
get_global_range(1): 8
get_global_id(0): 5
get_global_id(1): 2
get_local range(0): 4
get_local range(1): 4
get_local_id(0): 1
get_local_id(1): 2
get_group_range(0): 2
get_group_range(1): 2
get_group(0): 1
get_group(1): 0



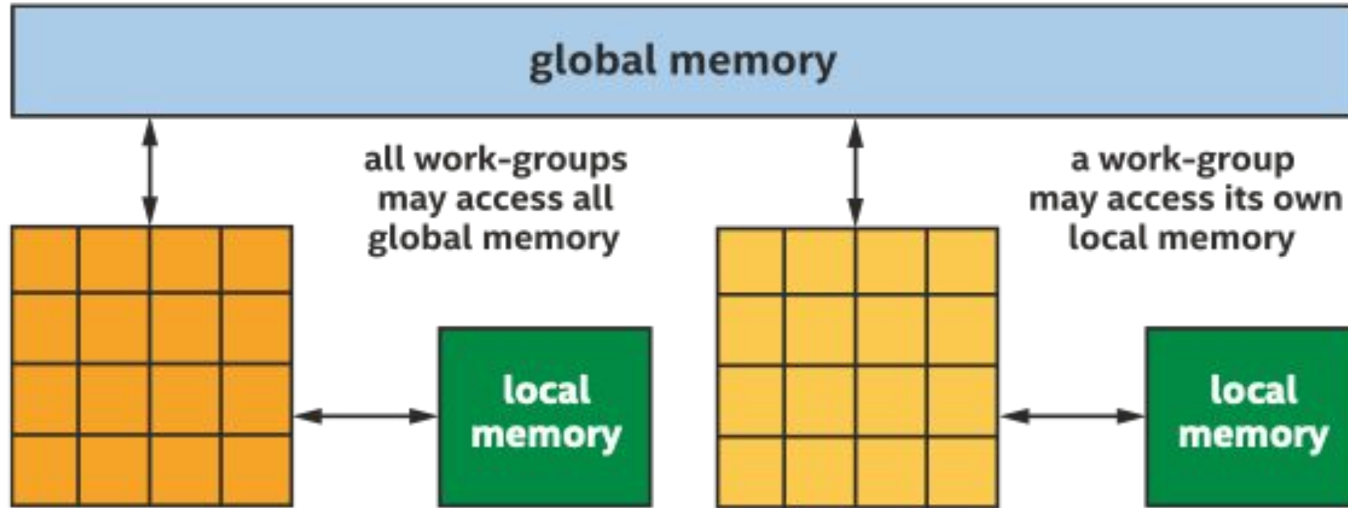
Cada item de trabalho e cada grupo de trabalho pode ser identificado dentro do kernel.

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

Modelo de execução



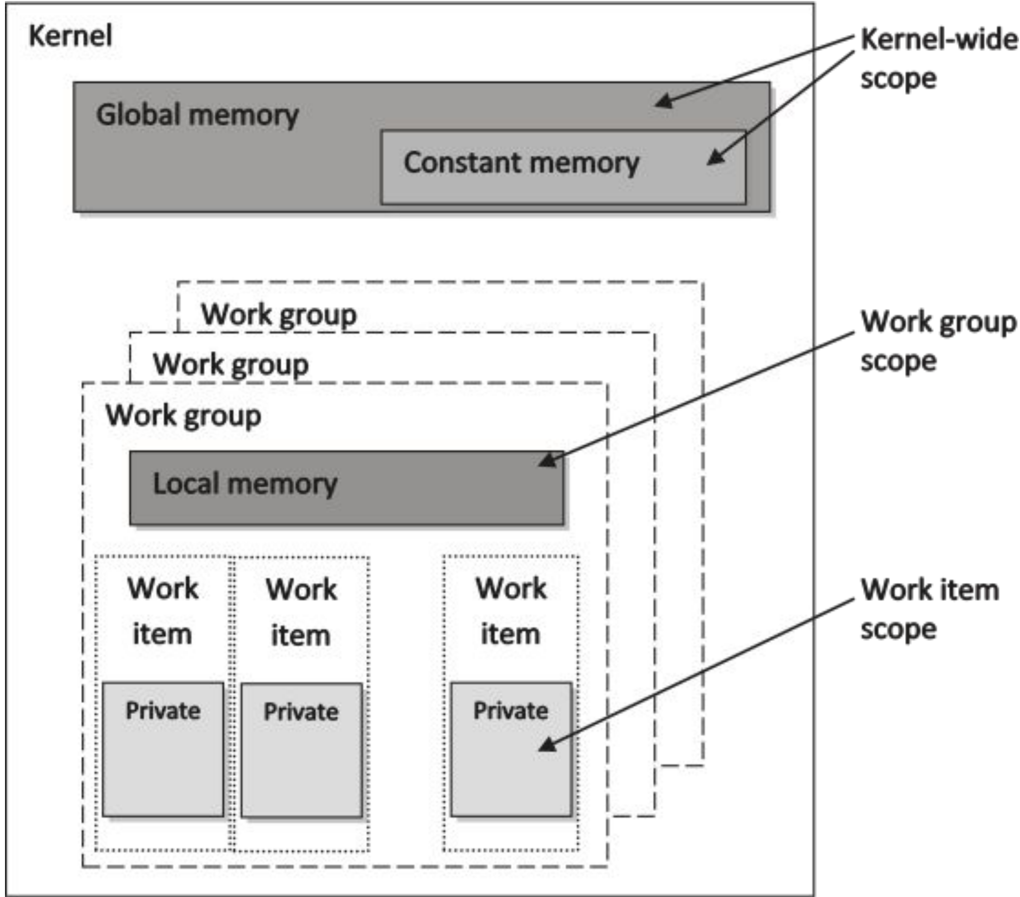
Modelo de memória



Cada item de trabalho tem a sua própria memória privada.

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

Modelo de memória



Passos gerais para a programação

1. Escolhendo as plataformas e os dispositivos
2. Criando as filas de comandos para 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 *ND-Range*

Solução sequencial:

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

Solução paralela via SYCL: forma básica

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

Solução paralela via SYCL: forma *ND-Range*

```
sycl::range global_range{N};
sycl::range local_range{NL};
h.parallel_for(sycl::nd_range{global_range, local_range}, [=] (sycl::nd_item<1> idx) {
  int i = idx.get_global_id(0);
  c[i] = a[i] + b[i];
});
```

nd_range pode ter 1, 2 ou 3 dimensões.

Kernels: forma *ND-Range*

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: forma básica

```
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];
});
```

Solução paralela via SYCL: forma *ND-Range*

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

Kernels: forma *ND-Range*

```
sycl::range global_range{N};
sycl::range local_range{NL};
h.parallel_for(sycl::nd_range{global_range, local_range}, [=] (sycl::nd_item<1> idx) {
  int i = idx.get_global_id(0);
  c[i] = a[i] + b[i];
});

sycl::range global_range{N};
sycl::range local_range{NL};
h.parallel_for(sycl::nd_range{global_range, local_range, sycl::id<1> {offset}}, [=] (sycl::nd_item<1> idx) {
  int i = idx.get_global_id(0);
  c[i] = a[i] + b[i];
});
```

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

Kernels: forma *ND-Range*

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

range global_range{N,M};
range local_range{NL,ML};
h.parallel_for(nd_range{global_range, local_range, id<2> {NO,MO}}, [=] (nd_item<2> idx) {
  int i = idx.get_global_id(0);  //nlin
  int j = idx.get_global_id(1);  //ncol
  c[i][j] = a[i][j] + b[i][j];
});
```

Kernels: forma *ND-Range*

```
sycl::range global_range{N};
sycl::range local_range{NL};
h.parallel_for(sycl::nd_range{global_range, local_range}, [=] (sycl::nd_item<1> idx) {
  int i = idx.get_global_id(0);
  c[i] = a[i] + b[i];
});

sycl::range<1> global_range{N};
sycl::range<1> local_range{NL};
h.parallel_for<kernel>(sycl::nd_range<1>{global_range, local_range}, [=] (sycl::nd_item<1> idx) {
  int i = idx.get_global_id(0);
  c[i] = a[i] + b[i];
});
```

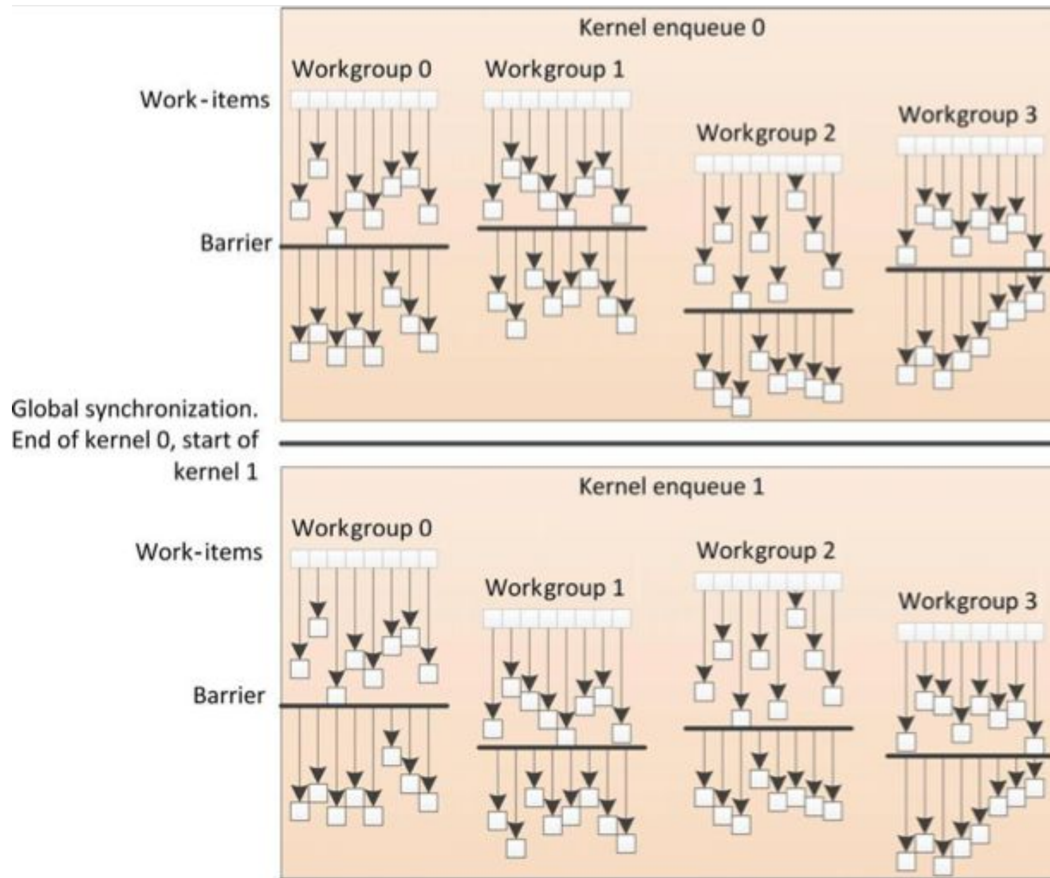
SYCL 1.2.1

Kernels: forma *ND-Range*

```
sycl::range global_range{N,M};
sycl::range local_range{NL,ML};
h.parallel_for(sycl::nd_range{global_range, local_range}, [=] (sycl::nd_item<2> idx) {
  int i = idx.get_global_id(0);  //nlin
  int j = idx.get_global_id(1);  //ncol
  c[i][j] = a[i][j] + b[i][j];
});

sycl::range<2> global_range{N,M};
sycl::range<2> local_range{NL,ML};
h.parallel_for<kernel>(sycl::nd_range<2>{global_range, local_range}, [=] (sycl::nd_item<2> idx) {
  int i = idx.get_global_id(0);  //nlin
  int j = idx.get_global_id(1);  //ncol
  c[i][j] = a[i][j] + b[i][j];
});
```

SYCL 1.2.1

Kernels: forma *ND-Range* − barreiras

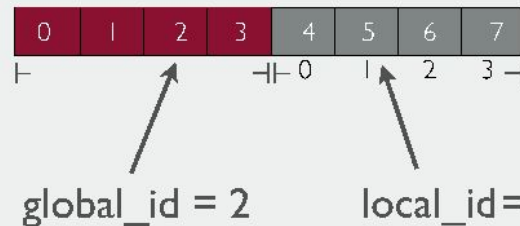


Kernels: forma *ND-Range* – *local accessors*

```
Q.submit([&] (handler &h) {

  sycl::accessor dataAcc{dataBuf, h};

  sycl::local_accessor<int> localIntAcc{4, h};


  h.parallel_for(nd_range{8,4}, [=] (nd_item<1> idx) {
    auto global_index = idx.get_global_id();
    auto local_index  = idx.get_local_id();

    localIntAcc[local_index] = dataAcc[global_index] + 1;
    dataAcc[global_index] = localIntAcc[local_index];
  });

});
```

Local Accessors:

- Não são criados a partir de *buffers*.
- São sempre *read_write*.



Kernels: forma *ND-Range* – *local accessors*

SYCL 1.2.1

```
Q.submit([&] (handler &h) {

  sycl::accessor dataAcc{dataBuf, h};

  auto localIntAcc = sycl::accessor<int,1,sycl::access::mode::read_write,
                                          sycl::access::target::local>(4, h);

  h.parallel_for(nd_range{8,4}, [=] (nd_item<1> idx) {
    auto global_index = idx.get_global_id();
    auto local_index  = idx.get_local_id();

    localIntAcc[local_index] = dataAcc[global_index] + 1;
    dataAcc[global_index] = localIntAcc[local_index];
  });

});
```

Local Accessors:

- Não são criados a partir de *buffers*.
- São sempre *read_write*.

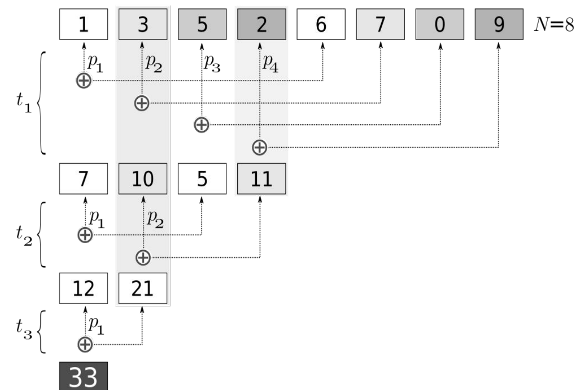
8 elementos

global_id = 2

local_id=1

Exemplo: redução

```
const unsigned N = 16;
range global_range{N};
range local_range{N/2};

Q.submit([&](handler &h) {

  local_accessor<int> SP{local_range, h};


  h.parallel_for(nd_range{global_range, local_range}, [=] (nd_item<1> idx) {

    // Copiar elementos da memória global para o array local SP

    int local_id = idx.get_local_id(0);
    for (int s = idx.get_local_range(0)/2; s > 0; s /= 2) {
      idx.barrier(sycl::access::fence_space::local_space);
      if (local_id < s) { SP[ local_id ] += SP[ local_id + s ]; }
    }

    // Atribuir somatório parcial do grupo (SP[0]) para memória global
  });

});
```

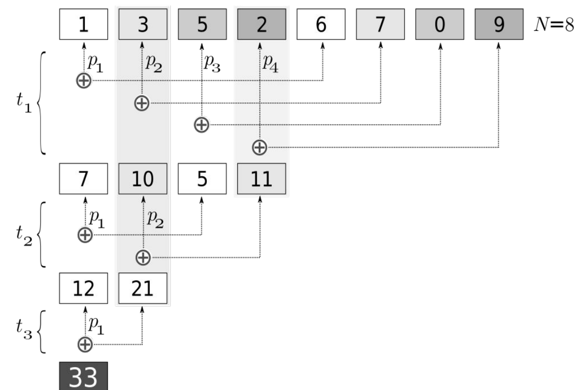


local_space
global_space
global_and_local (implícito)

Exemplo: redução

```
const unsigned N = 16;
range global_range{N};
range local_range{N/2};

Q.submit([&](handler &h) {

  auto SP = accessor<float,1,sycl::access::mode::read_write,
                     sycl::access::target::local>(local_range, h);

  h.parallel_for(nd_range{global_range, local_range}, [=] (nd_item<1> idx) {

    // Copiar elementos da memória global para o array local SP

    int local_id = idx.get_local_id(0);
    for (int s = idx.get_local_range(0)/2; s > 0; s /= 2) {
      idx.barrier(sycl::access::fence_space::local_space);
      if (local_id < s) { SP[ local_id ] += SP[ local_id + s ]; }
    }

    // Atribuir somatório parcial do grupo (SP[0]) para memória global
  });

});
```

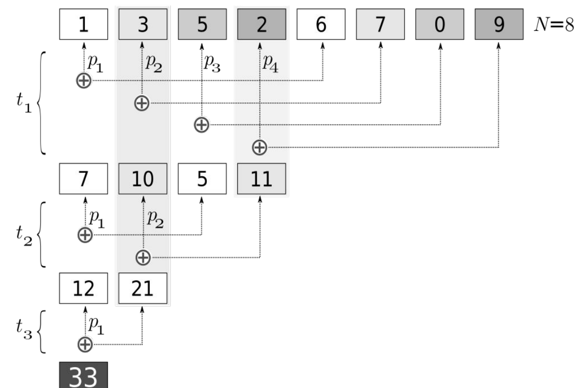
SYCL 1.2.1



local_space
global_space
global_and_local (implícito)

Exemplo: redução

```
const unsigned N = 16;
range global_range{N};
range local_range{N/2};

Q.submit([&](handler &h) {

  local_accessor<int> SP{local_range, h};


  h.parallel_for(nd_range{global_range, local_range}, [=] (nd_item<1> idx) {

    // Copiar elementos da memória global para o array local SP

    int local_id = idx.get_local_id(0);
    for (int s = idx.get_local_range(0)/2; s > 0; s /= 2) {
      idx.barrier(sycl::access::fence_space::local_space);
      if (local_id < s) { SP[ local_id ] += SP[ local_id + s ]; }
    }

    // Atribuir somatório parcial do grupo (SP[0]) para memória global
  });

});
```



local_id = 0 s = 4
SP[0] += SP[0+4] -> SP[0] = 1+6 = 7
local_id = 1 s = 4
SP[1] += SP[1+4] -> SP[1] = 3+7 = 10
local_id = 2 s = 4
SP[2] += SP[2+4] -> SP[2] = 5+0 = 5
local_id = 3 s = 4
SP[3] += SP[3+4] -> SP[3] = 2+9 = 11

local_id = 0 s = 2
SP[0] += SP[0+2] -> SP[0] = 7+5 = 12
local_id = 1 s = 2
SP[1] += SP[1+2] -> SP[1] = 10+11 = 21

local_id = 0 s = 1
SP[0] += SP[0+1] -> SP[0] = 12+21 = 33

Kernels

```
h.single_task([=] () {
  // O kernel é executado uma única vez (um item de trabalho)
});

h.parallel_for(range(global_range), [=] (id<1> it) {
  // O kernel é executado por todos os itens de trabalho
});

h.parallel_for(nd_range(global_range,local_range), [=] (nd_item<1> it) {
  // O kernel é executado por todos os itens de trabalho
});

h.parallel_for_work_group(range(num_groups), [=] (group<1> gp) {
  // O kernel é executado uma única vez por grupo de trabalho (um item de trabalho por grupo)
  // As variáveis são alocadas na memória local

  gp.parallel_for_work_item(group_size, [=] (h_item<1> it) {
    // O kernel é executado por todos os itens de trabalho do grupo
    // As variáveis são alocadas na memória privada
  });
});

h.host_task([=] () {
  // Código C++ nativo. Dependências baseadas em host accessors
  // Não há restrições de linguagem como aquelas aplicadas aos kernels
  // (por exemplo: recursividade, alocação dinâmica, etc.)
});
```

Kernels

```
sycl::buffer buf{array};

Q.submit([&](handler &h) {
  auto var = buf.get_access<sycl::access::mode::read_write, sycl::access::target::host_buffer>(h);
  h.host_task([=] () {
    //algum código em c++ nativo usando a variável var
  });
});
```

SYCL 1.2.1

```
sycl::buffer buf{array};

Q.submit([&](handler &h) {
  auto var = buf.get_host_access(h);
  h.host_task([=] () {
    //algum código em c++ nativo usando a variável var
  });
});
```

Kernels: hierárquicos

```
range group_size{16};
range num_groups = N/group_size; // N é um múltiplo de group_size

h.parallel_for_work_group(num_groups, [=] (group<1> group) {

   int localIntArr[16]; // variável alocada na memória local

   // barreira implícita
   group.parallel_for_work_item(group_size, [&] (h_item<1> item) {

      auto global_index = item.get_global_id(); // variável alocada na memória privada
      auto local_index  = item.get_local_id(); // variável alocada na memória privada

      localIntArr[local_index] = global_index + 1;
      data_acc[global_index] =localIntArr[local_index];
   });
});
```

Kernels: hierárquicos

```
range group_size{16};
range num_groups = N/group_size; // N é um múltiplo de group_size

h.parallel_for_work_group(num_groups, group_size [=] (group<1> group) {

   int localIntArr[16]; // variável alocada na memória local

   //barreira implícita
   group.parallel_for_work_item([&] (h_item<1> item) {

      auto global_index = item.get_global_id(); // variável alocada na memória privada
      auto local_index  = item.get_local_id(); // variável alocada na memória privada

      localIntArr[local_index] = global_index + 1;
      data_acc[global_index] =localIntArr[local_index];
   });
});
```

Exercícios

1. **Calcular a raiz quadrada de cada elemento de um vetor usando *buffers***
2. **Calcular a soma de dois vetores usando *buffers***
3. Computação heterogênea
4. Redução

Exemplo: computação heterogênea

Computar:

$$x = 3x - \sqrt{x}$$

Tempo 1:

CPU computa paralelamente a multiplicação **cpu_y = 3x**

GPU computa paralelamente a raiz quadrada **gpu_y = √x**

Tempo 2:

CPU computa paralelamente a subtração **x = cpu_y − gpu_y**

Exemplo: computação heterogênea

```
const unsigned N = 42;
std::array<float,N> X, cpuY, gpuY;
for (int i=0; i<N; i++) X[i] = i;

queue cpu_fila{cpu_selector{}};
queue gpu_fila{gpu_selector{}};
{
   buffer bufferX{X};
   buffer cpu_bufferY{cpuY};
   buffer gpu_bufferY{gpuY};

   cpu_fila.submit([&](handler &h) {
     accessor x{bufferX, h, read_only};
     accessor cpu_y{cpu_bufferY, h,
write_only};
     h.parallel_for(N, [=] (id<1> i) {
       cpu_y[i] = 3 * x[i];
     });
   });

   gpu_fila.submit([&](handler &h) {
     accessor x{bufferX, h, read_only};
     accessor gpu_y{gpu_bufferY, h,
write_only};
     h.parallel_for(N, [=] (id<1> i) {
       gpu_y[i] = sqrt(x[i]);
     });
   });

   cpu_fila.submit([&](handler &h) {
     accessor x{bufferX, h, write_only};
     accessor cpu_y{cpu_bufferY, h, read_only};
     accessor gpu_y{gpu_bufferY, h, read_only};
     h.parallel_for(N, [=] (id<1> i) {
       x[i] = cpu_y[i] - gpu_y[i];
     });
  });
}
```

Copia bufferX para a CPU | Executa **“3x”** na CPU | Executa a **subt.** na CPU | Copia bufferX para o hosp.

Copia bufferX para a GPU | Executa **“√x”** na GPU | Copia gpu_bufferY para a CPU