KHRONOS®
GROUP

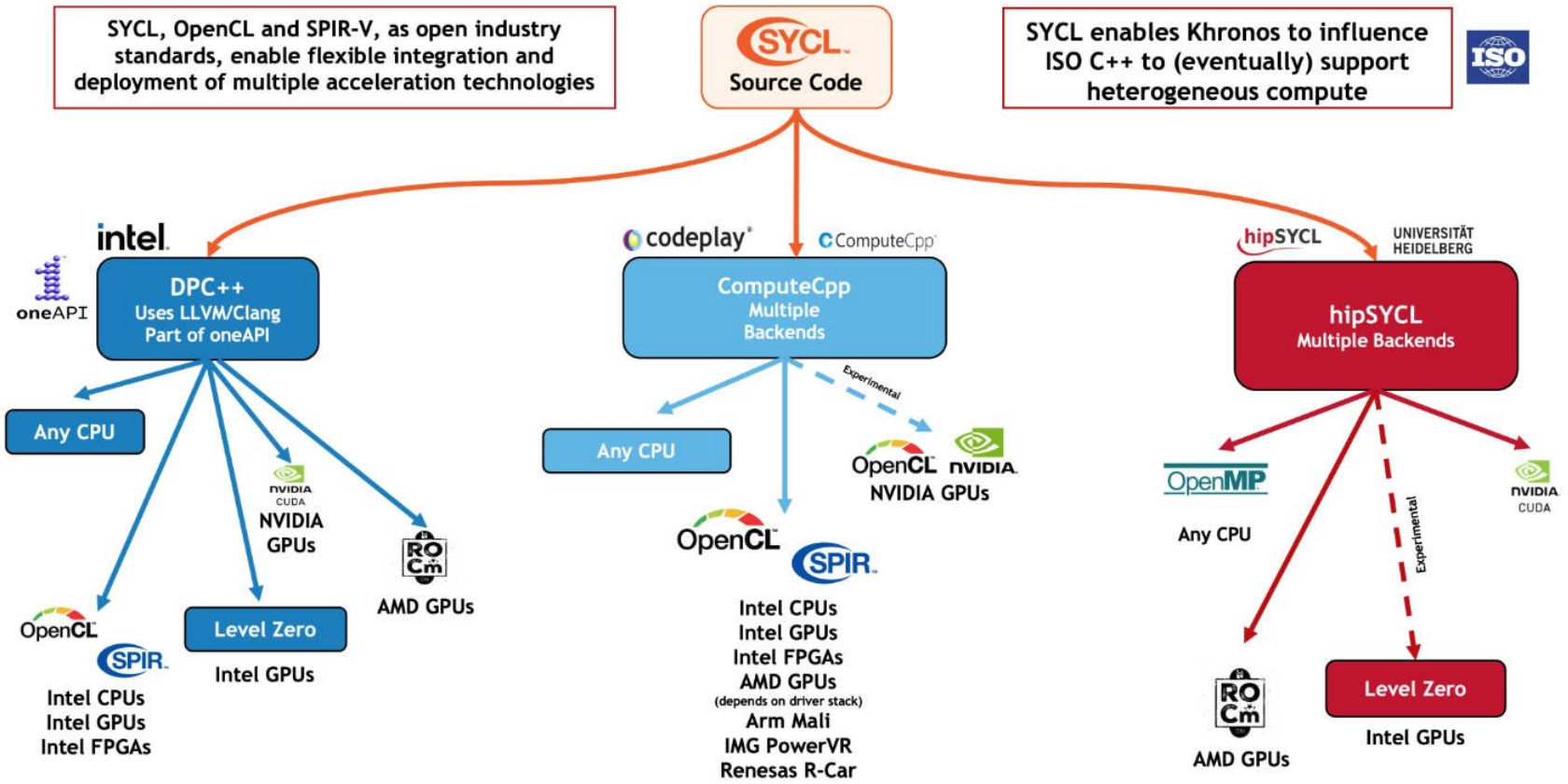
SYCL

- Couche d’abstraction multi-plateforme
- Maintenue et développée par Khronos Group
- Standard ouvert
- C++

SYCL

- Multiples implémentations
  - Intel oneAPI Data Parallel C++ (DPC++)
  - Codeplay ComputeCPP (Nvidia, Intel, OpenCL)
  - HipSYCL (Nvidia, AMD, Intel GPU)
  - neoSYCL (NEC)

SYCL


SYCL, OpenCL and SPIR-V, as open industry standards, enable flexible integration and deployment of multiple acceleration technologies
SYCL
Source Code
SYCL enables Khronos to influence ISO C++ to (eventually) support heterogeneous compute
ISO
intel
oneAPI
DPC++
Uses LLVM/Clang
Part of oneAPI
Any CPU
NVIDIA
CUDA
NVIDIA GPUs
AMD GPUs
OpenCL
SPIR
Level Zero
Intel GPUs
Intel CPUs
Intel GPUs
Intel FPGAs
codeplay
ComputeCpp
ComputeCpp
Multiple Backends
Experimental
Any CPU
OpenCL
NVIDIA
NVIDIA GPUs
OpenCL
SPIR
Intel CPUs
Intel GPUs
Intel FPGAs
AMD GPUs
(depends on driver stack)
Arm Mali
IMG PowerVR
Renesas R-Car
hipSYCL
UNIVERSITÄT HEIDELBERG
hipSYCL
Multiple Backends
OpenMP
Any CPU
NVIDIA
CUDA
Experimental
AMD GPUs
Level Zero
Intel GPUs

Programmation hétérogène

Structure typique d'un programme SYCL

La plus grande partie du programme s'exécute sur le CPU. Seule une partie du code (les calculs) est exécutée par un accélérateur (GPU, FPGA...etc)

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

int main() {
  constexpr int size=16;
  std::array<int, size> data;

  // Create queue on implementation-chosen default device
  queue Q;

  // Create buffer using host allocated "data" array
  buffer B { data };

  Q.submit([&](handler& h) {
    accessor A{B, h};
    h.parallel_for(size , [=](auto& idx) {
        A[idx] = idx;
      });
    });

  // Obtain access to buffer on the host
  // Will wait for device kernel to execute to generate data
  host_accessor A{B};
  for (int i = 0; i < size; i++)
    std::cout << "data[" << i << "] = " << A[i] << "\n";

  return 0;
}
```

Host code

Device code

Host code

SYCL

- Le code sur accélérateur est exécuté de manière asynchrone.
- Il existe des restrictions sur les accélérateurs
  - Pas d’allocation mémoire
  - Pas de RTTI
  - Pas de gestion des exceptions C++
    - Mais SYCL permet quand même la gestion des erreurs via les exceptions du C++

Device

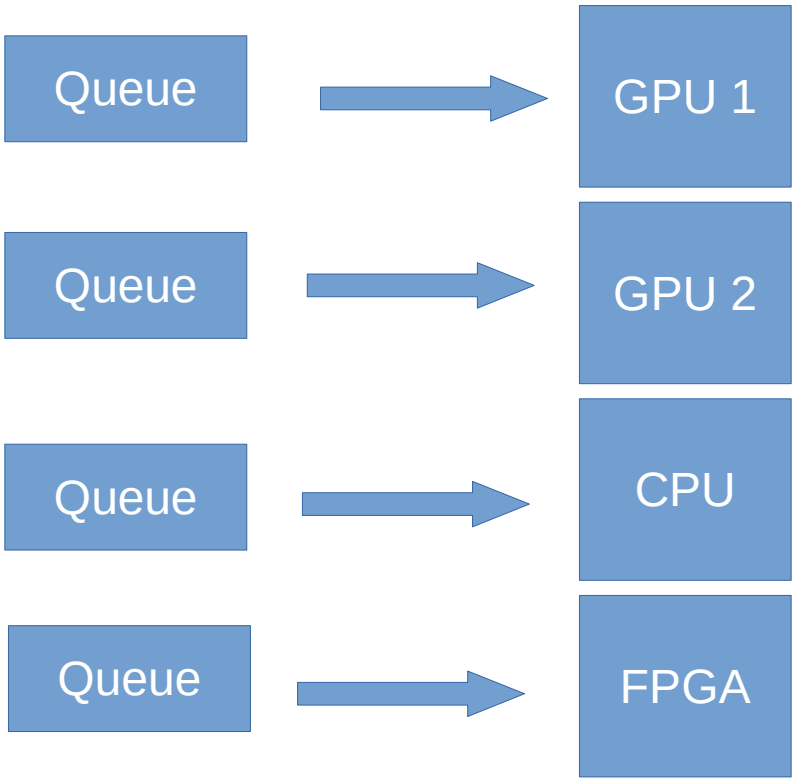
- Un device est le matériel qui va exécuter les commandes SYCL ainsi que les codes de calcul.
- Un device est :
  - Le CPU
  - Un GPU (intégré ou discret(e))
  - Un FPGA
- Le développeur décide quel device exécute le code.
- Il est possible de faire exécuter du code par plusieurs devices en même temps.

Queue

- Une queue permet d’envoyer des actions (commandes et kernels) vers un device.
- Une queue est rattachée à un device au moment de sa création.
- Il n’est possible d’envoyer et d’exécuter qu’une commande à la fois via une queue.

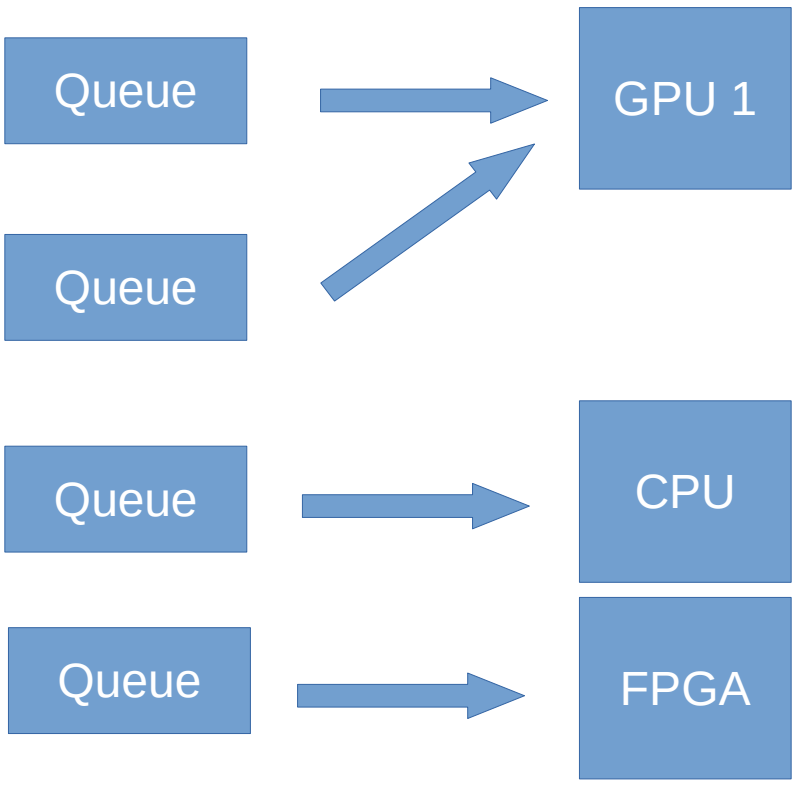
Queue

Un programme peut créer plusieurs queues.
Chaque queue est rattachée à un device.
L'envoi et l'exécution des commandes étant asynchrone, il est possible de faire exécuter un même kernel sur plusieurs devices en même temps



Queues

Il est possible de rattacher plusieurs queues à un même device.



Queue

```
1   #include <CL/sycl.hpp>
2   #include <iostream>
3   using namespace sycl;
4
5   int main()
6   {
7       // Création d'une queue. L'implémentation choisie le device
8       queue Q;
9       std::cout << "Selected device: " << Q.get_device().get_info<info::device::name>() << "\n";
10      return 0;
11  }
12
```

Si aucun device n’est spécifié lors de la création de la queue, l’implémentation en choisit une automatiquement.
Ci-dessous une sortie possible de ce programme.

```
Selected device: Intel(R) UHD Graphics 630 [0x3e9b]
```

Queue

Afin de spécifier à quel device nous désirons rattacher une queue, il faut utiliser un objet du type « device_selector »

Par défaut, il existe 5 objets du type « device_selector »

- **gpu_selector**
  - Sélectionne un GPU parmi les GPU disponibles
- **cpu_selector**
  - Sélectionne le CPU
- **default_selector**
  - Sélectionne automatiquement un device, cela peut-être le CPU, un GPU ou une carte FPGA.
- **host_selector**
  - Revient à utiliser cpu_selector
- **accelerator_selector**
  - Sélectionne une carte FPGA

Si aucun device_selector n’est spécifié, l’objet default_selector est utilisé.

Queue

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

int main()
{
    // Création d'une queue. L'implémentation choisie le device
    queue Q(cpu_selector{});
    std::cout << "Selected device: " << Q.get_device().get_info<info::device::name>() << "\n";
    return 0;
}

```

Initialisation d’une queue en utilisant un device_selector. Ici le CPU

```
Selected device: Intel(R) Core(TM) i5-9300H CPU @ 2.40GHz
```

SYCL garantit l’existence d’au moins un device (le CPU)

Premier kernel

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

int main()
{
    queue Q(gpu_selector{});
    std::cout << "Selected device: " << Q.get_device().get_info<info::device::name>() << "\n";
    std::vector<float> vec{1, 2, 3, 4, 5, 6, 7, 8};
    {
        buffer buff(vec.data(), range<1>(vec.size()));
        Q.submit([&](handler &h){
        accessor acc(buff,h);
        h.parallel_for(vec.size(),[=](auto i){
            acc[i]=acc[i]+1;
        });
        });
    }
    for (auto v : vec)
    {
        std::cout << v << " ";
    }
    std::cout << std::endl;
    return 0;
}
```

Parallélisme

```
61      h.parallel_for(range(M, P), [=](auto index) {
62        int row = index[0];
63        int col = index[1];
64        float sum = 0.0f;
65        // Compute the result of one element of c
66        for (int i = 0; i < width_a; i++) {
67          sum += a[row][i] * b[i][col];
68        }
69        c[index] = sum;
70      });
71    });
```

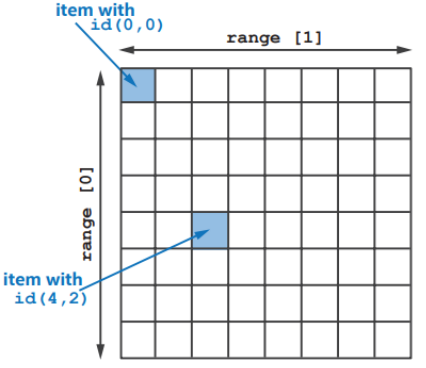
Ceci n'est pas une boucle (enfin, pas tout à fait)

Sur un GPU (ou un CPU) chaque coeur va se voir assigner une instance du kernel

L'exécution de chaque instance est parallèle et est contrôlée par SYCL

La répartition entre les coeurs est automatique

index représente le numéro de l'instance du kernel. Il s'agit de la classe « id »
De manière simple, ce numéro d'instance est transposé en l'index de la donnée à traiter
Conceptuellement, l'identifiant du kernel est un tableau. Chaque élément du tableau est l'indice d'une des dimensions de la donnée.



Buffer

- L’objet « Buffer » est une couche d’abstraction par dessus des données sur lesquelles nous voulons travailler.
- Un « Buffer » gère automatiquement les transferts de données entre la mémoire centrale et la mémoire du device.
- Les données gérées par un buffer ne sont recopiées dans la mémoire du CPU qu’au moment de la destruction de l’objet.

Accessors

- L’objet « accessor » permet l’accès aux données qui sont gérées par un Buffer.
- L’objet « accessor » est uniquement utilisable dans un kernel.
- L’objet « host_accessor » permet d’accéder aux données d’un buffer en dehors d’un kernel.

TP 1

- En partant de l’exemple précédant, écrire un programme qui permet d’additionner les données de deux vecteurs et stocker le résultat dans un troisième vecteur.

USM (unified shared memory)

- Gestion de la mémoire basée sur l’utilisation de pointeurs.
- Allocation mémoire explicite
- Mouvement des données manuelle où automatique.

USM

- malloc_host<T>
  - Alloue de la mémoire dans la mémoire du CPU
- malloc_shared<T>
  - Alloue de la mémoire partagée par le CPU et le GPU
- malloc_device<T>
  - Alloue de la mémoire directement dans la mémoire du GPU, il n’est pas possible d’accéder à cette mémoire depuis le CPU
- free(pointer,queue)
  - Libère la mémoire allouée par l’une des trois fonctions d’allocation mémoire

USM

- Lors de l’utilisation du pointeur par le CPU ou le GPU, SYCL copie automatiquement.
- SYCL s’assure que les données sont disponibles pour l’exécution du kernel en copiant les données au bon moment.
- Facilite le portage et l’interopérabilité avec du code existant

USM

Exemple :

```
float* a = malloc_shared<float>(NBITEMS,Q);
float *b = malloc_host<float>(NBITEMS,Q) ;
…..
   h.parallel_for(NBITEMS,[=](auto i){
        a[i]=a[i]+b[i];
    });
….
    for (int i=0; i<NBITEMS; i++) {
        std::cout << a[i] << " " ;
}
free(a,Q);
```

TP 2

Adaptez le programme d’addition de vecteurs afin d’utiliser malloc_host<T> et malloc_shared<T>

USM

- Lorsque de la mémoire est allouée explicitement sur le device grâce à malloc_device il est possible de copier manuellement les données entre le CPU et le GPU

Example :

```
Q.submit([&](handler &h){
    h.memcpy(device_array,host_array,nbitems) ;
});
Q.wait() ; // Attend que l’opération soit effectuée
```

USM ou Buffer ?

- Question de choix personnel
- Contraintes liées au portage de codes
- L’utilisation de buffer permet à SYCL d’utiliser un arbre de dépendance qui ordonne automatiquement l’exécution des kernels

Données multidimensionnelles

- range<> permet de spécifier la dimension des données (1D,2D et 3D)
- Utilisé avec les Buffers et parallel_for
- Multiplication de matrices...etc

Multiplication de matrices

```
 9  // Matrix size constants.
10  constexpr int m_size = 150 * 8;
11  constexpr int M = m_size;
12  constexpr int N = m_size;
13  constexpr int P = m_size;
```

```
26      buffer<float, 2> a_buf(range(M, N));
27      buffer<float, 2> b_buf(range(N, P));
28      buffer c_buf(reinterpret_cast<float *>(c_back), range(M, P));
```

```
61        h.parallel_for(range(M, P), [=](auto index) {
62          int row = index[0];
63          int col = index[1];
64          float sum = 0.0f;
65          // Compute the result of one element of c
66          for (int i = 0; i < width_a; i++) {
67            sum += a[row][i] * b[i][col];
68          }
69          c[index] = sum;
70        });
71      });
72    } catch (sycl::exception const &e) {
73      cout << "An exception is caught while multiplying matrices.\n";
74      terminate();
75    }
```

Gestion d’erreurs

- SYCL permet de reporter les éventuelles erreurs sous forme d’exceptions C++
- sycl::exception
- Une exception peut être levée par
  - les queues lors de l’exécution d’une commande
  - un handler lors de l’exécution d’un kernel

Questions