Intel oneAPI DPC++ Library (oneDPL)

Programmation parallèle performante

OneDPL

- Programmation hétérogène (CPU, GPU, FPGA) multiplate-forme
- Basée sur les standards connus tel que C++ STL, Boost.Compute, SYCL
- Algorithmes C++ optimisés et parallélisés via SYCL, OpenMP et oneTBB
- Interopérabilité totale avec les autres librairies oneAPI

Utilisation de la librairie oneDPL

- La version minimale du standard C++ est C++17
- Les fichiers d’entête commencent par oneapi/dpl

  #include <oneapi/dpl/...>
- Utilisation du namespace oneapi::dpl

  alias court prédéfini : namepace dpl=oneapi::dpl ;

Parallélisme en C++17/20

- C++17 permet de définir des politiques d'exécution des algorithmes standards
  - std::execution::seq
  - std::execution::par
  - std::execution::unseq
  - std::execution::par_unseq

std::execution::seq

- Exécution séquentiel explicite

```
std::vector<int> x{5, 7, 6, 4, 8, 2};
std::sort(std::execution::seq, x.begin(), x.end());
std::copy(x.begin(), x.end(), std::ostream_iterator<int>{std::cout, " "});
std::cout << '\n';
```

équivalant à

```
std::sort(v.begin(),v.end()) ;
```

std::execution::par

- Exécution de parallèle de l’algorithme

```
std::vector<int> x{5, 7, 6, 4, 8, 2};
std::sort(std::execution::seq, x.begin(), x.end());
std::copy(x.begin(), x.end(), std::ostream_iterator<int>{std::cout, " "});
std::cout << '\n';
```

std::execution::unseq

- Permet la vectorisation si possible

```
std::vector<int> x{5, 7, 6, 4, 8, 2};
std::sort(std::execution::unseq, x.begin(), x.end());
std::copy(x.begin(), x.end(), std::ostream_iterator<int>{std::cout, " "});
std::cout << '\n';
```

std::execution::par_unseq

- Permet la parallélisation et la vectorisation si possible

```
std::vector<int> x{5, 7, 6, 4, 8, 2};
std::sort(std::execution::par_unseq, x.begin(), x.end());
std::copy(x.begin(), x.end(), std::ostream_iterator<int>{std::cout, " "});
std::cout << '\n';
```

Extensions oneDPL

- oneDPL étend l’utilisation de la plupart des algorithmes standard pour permettre l’utilisation de SYCL.
- L’exécution peut alors avoir lieu sur CPU, GPU où FPGA
- dpl::execution::dpcpp_default
  - Utilise la queue par défaut de SYCL

Utilisation explicite du GPU

```
#include <oneapi/dpl/execution>
#include <oneapi/dpl/algorithm>
int main()
{
    std::vector<int> vec(1000);
    dpl::fill(dpl::execution::dpcpp_default,vec.begin(), vec.end(), 0);
    return 0;
}
```

Utilisation de « buffers » Sycl

dpl::begin et dpl::end sont des fonctions qui permettent l'utilisation d'objets buffer de syscl avec les algorithmes.

Utilisées avec un buffer Sycl, ces fonctions retournent un type non spécifié ayant certaines des propriétés des objets « random access iterator »

- Ils peuvent être copier où passé à un constructeur
- Ils peuvent être comparés (== et !=)
- Ils peuvent être utilisés dans des expressions

Exemple d’utilisation de buffers

```
#include <oneapi/dpl/execution>
#include <oneapi/dpl/algorithm>
#include <oneapi/dpl/iterator>
#include <CL/sycl.hpp>
int main()
{
sycl::buffer<int> buf{ 1000 };
dpl::fill(dpl::execution::dpcpp_default, dpl::begin(buf),dpl::end(buf), 1234);
auto result = dpl::find(dpl::execution::dpcpp_default,dpl::begin(buf),
    dpl::end(buf), 1234);
return 0;
}
```

Le buffer est réutilisé, les données sont dans la mémoire du GPU

Gestion des modes d’accès

Il est possible de spécifier le mode d’accès aux régions de la mémoire représentées par un buffer

```
auto start=dpl::begin(buf) ;
dpl::fill(dpl::execution::dpcpp_default,start,start+100,0) ;
```

```
auto start=dpl::begin(buf,sycl::write_only, syscl::noinit) ;
dpl::fill(dpl::execution::dpcpp_default,start,start+100,0) ;
```

Utilisation de la librairie standard

- Le code s’exécutant sur le CPU peut utiliser toute la librairie standard du c++
- GPU/FPGA
  - Certaines fonctions et classes C++ ne peuvent être utilisées à cause des limitations de Sycl.
    - Fonctions utilisant les exceptions
    - Allocation dynamique de la mémoire
    - Les fonctions virtuelles

Les API standards testées

- Près de 120 API standards ont été testées
  - Utilisables dans les kernels DPC++
  - Transfert de données entre la RAM et le GPU
- Testés avec les 3 implémentations majeurs
  - libstdc++ (GNU)
  - libc++ (LLVM/Clang)
  - Miscrosoft STL

Utilisation des nombres complexes

```
#include <oneapi/dpl/execution>
#include <oneapi/dpl/algorithm>
#include <oneapi/dpl/iterator>
#include <sycl/sycl.hpp>
#include <complex>

int main() {
    const size_t n = 100;
    sycl::buffer<std::complex<double>> c_buf{n};
    auto idx = dpl::counting_iterator<int>(0);
    dpl::transform(dpl::execution::dpcpp_default, idx, idx+n, dpl::begin(c_buf),[=](auto i) {
        const double v = fabs(static_cast<double>(n)/2.0 – static_cast<double>(i));
        return std::complex<double>{v, v};
    });
    return 0;
}
```

Iterateurs

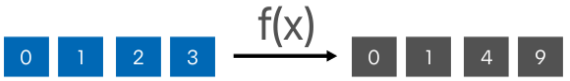
- oneDPL fournit un certain nombre d’itérateurs facilitant la parallélisation
  - counting_iterator
  - zip_iterator
  - transform_iterator
  - permutation_iterator
  - discard_iterator

counting_iterator

- Représente une séquence linéaire d’entiers
- Utilisé comme un index
- N’est pas en mémoire

transform_iterator

Un itérateur défini sur un autre itérateur dont la valeur déréférencée est le résultat d'une fonction appliquée à l'élément correspondant de l'itérateur original.



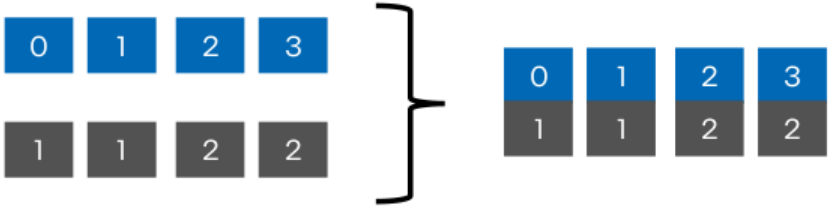

```
using namespace oneapi;
dpl::counting_iterator<int> first(0);
dpl::counting_iterator<int> last(10);
auto transform_first = dpl::make_transform_iterator(first, std::negate<int>());
auto transform_last = transform_first + (last – first);
auto sum = std::reduce(dpl::execution::dpcpp_default,
                       transform_first, transform_last);
```

zip_iterator

Un itérateur construit avec un ou plusieurs itérateurs en entrée.

Le déréférencement d'un zip_iterator est un objet de type tuple d'un type non spécifié qui contient les valeurs retournées par le déréférencement des itérateurs membres, que le zip_iterator enveloppe. Les opérations arithmétiques effectuées sur une instance de zip_iterator sont également appliquées à chacun des itérateurs membres.

```
using namespace oneapi;
auto zipped_begin = dpl::make_zip_iterator(sequence1.begin(), sequence2.begin(), sequence3.begin());
auto found = std::find(dpl::execution::dpcpp_default, zipped_begin, zipped_begin + n,
[](auto tuple_like_obj) {
      auto [e1, e2, e3] = tuple_like_obj;
      return e1 == e2 && e1 == e3;
    }
)
```



Pour aller plus loin

- Spécification oneDPL
  - https://spec.oneapi.io/versions/latest/elements/oneDPL/source/index.html
- Guide de référence
  - https://docs.oneapi.io/versions/latest/onedpl/index.html
- Code source
  - https://github.com/oneapi-src/oneDPL