UNIVERSIDAD DEL BÍO-BÍO
FACULTAD DE CIENCIAS EMPRESARIALES

Introduction to DPC++ Heterogeneous Computing

Professor: Dr. Joel Fuentes - jfuentes@ubiobio.cl

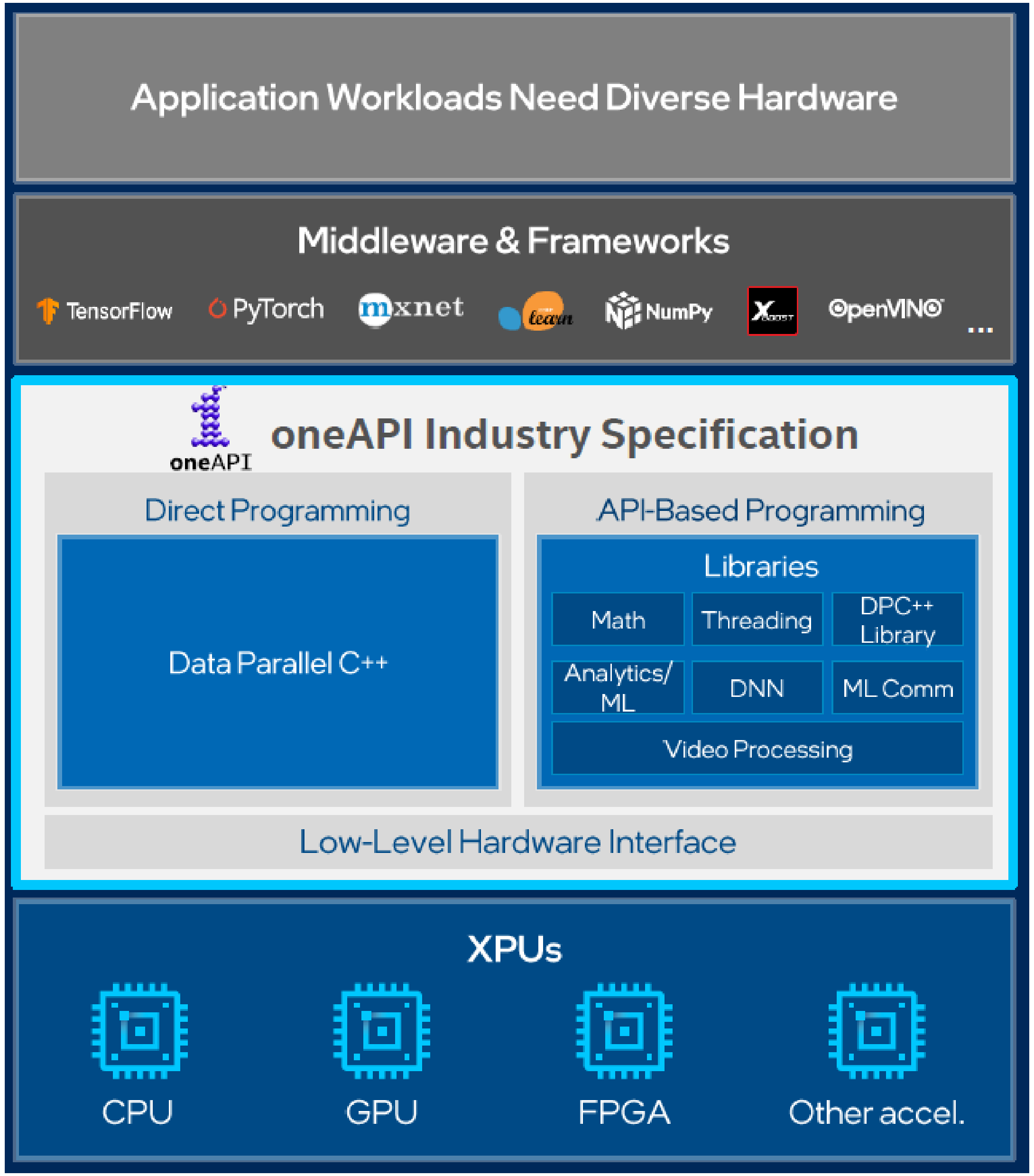
Teaching Assistants:
Daniel López - daniel.lopez1701@alumnos.ubiobio.cl
Sebastián González - sebastian.gonzalez1801@alumnos.ubiobio.cl

Course website: http://www.face.ubiobio.cl/~jfuentes/classes/ch

DPC++

- SYCL is a high-level programming model for heterogeneous processors.
- SYCL defines a standard for different organizations to perform deployments.
- DPC++ is an implementation of SYCL.
- DPC++ is part of the OneAPI suite

OneAPI



Application Workloads Need Diverse Hardware
Middleware & Frameworks
TensorFlow
PyTorch
mxnet
NumPy
OpenVINO
...
oneAPI
oneAPI Industry Specification
Direct Programming
Data Parallel C++
API-Based Programming
Libraries
Math
Threading
DPC++ Library
Analytics/ ML
DNN
ML Comm
Video Processing
Low-Level Hardware Interface
XPUs
CPU
GPU
FPGA
Other accel.

Lexis and semantic similar to C++

- Example:

`#include <stdio.h>`	comandos del preprocesador
`typedef float balance;`	definiciones de tipos
`void imprime();`	prototipos de funciones
`int cont=3; balance b;`	variable globales

```
int main(void) {
  b=1000;
  imprime();                         funciones
  return 0;
}

void imprime(){
  printf("cont = %d /n", &cont);
  printf("balance = %f /n", &b);
}
```

Data types

Tipo	N° de bits	Rango
char	8	-128 a 127
unsigned char	8	0 a 255
signed char	8	-128 a 127
int	16/32	-32.768 a 32.767
unsigned int	16	0 a 65.535
signed int	16	-32.768 a 32.767
short int	16	-32.768 a 32.767
unsigned short int	16	0 a 65.535
signed short int	16	-32.768 a 32.767
long int	32	-2.147.483.648 a 2.147.483.647
signed long int	32	-2.147.483.648 a 2.147.483.647
float	32	3,4 E -38 a 3,4 E +38
double	64	1,7 E -308 a 1,7 E +308
long double	64	1,7 E -308 a 1,7 E +308

Compilation

- Dpcpp is the compiler for DPC++.
- It can function as a cross-compiler, generating code for platforms other than development.
- Use:
- $ dpcpp [options] [files] -lsycl
- Most used options:

--help:	mostrar la ayuda de gcc.
-o [archivo]:	permite colocar un nombre al archivo ejecutable generado en la compilación. (por defecto es a.out)
-Wall:	muestra todos los errores y advertencias al compilar.
-g:	incluye en el archivo ejecutable información necesaria para usar posteriormente un depurador (por ejemplo gdb)
-O [nivel]:	permite optimizar el código (el nivel va de 0 a 3). Obs: no use la opción –O cuando use la opción –g.
-E:	sólo realiza la fase del preprocesador, no compila ni ensambla.
-S:	Genera sólo el archivo ensamblador del programa
-c:	Genera sólo código objeto

Program in DPC++

- Unique source code
- Host code and heterogeneous accelerator kernels can be included in the same source files.
- <u>Main features</u>:
    - Queue
        - For target work
    - malloc_shared
        - Data Management
    - parallel_for
        - Parallelism

host

device

host

```
#include <CL/sycl.hpp>
constexpr int N=16;
using namespace sycl;
int main(){
    queue q;
    int *data = malloc_shared<int>(N,q);
    q.parallel_for(N, [=](auto i){
       data[i] = i;
    }).wait();
    for (int i=0; i<N; i++)
        std::cout << data[i] << "\n";
    free(data,q);
    return 0;
}
```

Program in DPC++

```
#include <CL/sycl.hpp>
constexpr int N=16;
using namespace sycl;
int main(){
    queue q;
    std::vector<int> v(N);
    buffer buff(v);
    q.submit([&](handler& h){
        accessor a(buff, h, write_only);
        h.parallel_for(N,[=](auto i) {
            a[i] = i;
        });
    }).wait();
    for (int i=0; i<N; i++)
         std::cout << v[i] << "\n";
     return 0;
}
```

1. Create device queue
2. Create buffers
3. Enqueue kernel
4. Specify kernel parameters for execution
5. Create accessors
6. Specify kernel function

Structure of a program in DPC++

Aspects to consider in DPC++ programs

1. Decide where the code will run
2. Decide the parallel execution model
3. Data transfer and synchronization

Main classes in DPC++

Class	Functionality
sycl::device	It represents a CPU, GPU, FPGA, or other specific device that can run SYCL kernels.
sycl::queue	Represents a queue in which kernels can be added for execution.
sycl::buffer	Encapsulates memory buffer that can be transferred between host and device
sycl::handler	Used to define group commands that connect buffers to kernels.
sycl::accessor	Used to define access requirements from kernels (read-only, write-only, etc.).
sycl::range, sycl::nd_rangesycl::id, sycl::item, sycl::nd_item	Representation of execution by ranges and individual execution in ranges.

On-device execution queues (Queue)

- The queues allow us to connect with the devices (device).
- With them we send kernels to execute work and move data.
- It is declared as: queue nameQueue;

```
#include <CL/sycl.hpp>
using namespace sycl;
int main(){
    queue q;
    std::vector<int> v(N);
    buffer buff(v);
     …

     return 0;
}
```

You can also specify where to run kernel:

queue q(sycl::host_selector{}); // run on the CPU without a runtime (i.e., no OpenCL)
queue q(sycl::cpu_selector{}); // run on the CPU with a runtime (e.g., OpenCL)
queue q(sycl::gpu_selector{}); // run on the GPU
queue q(sycl::accelerator_selector{}); // run on an FPGA or other accelerator

Buffers

- Allows memory declaration to be used by host and devices.
- The easiest way to declare them is to indicate in their constructor the data source: array, vector, pointer, etc.
- It is declared as: ***buffer nameBuffer(data);***

```
#include <CL/sycl.hpp>
using namespace sycl;
int main(){
    queue q;
    std::vector<int> v(N);
    buffer buff(v);
     …

     return 0;
}
```

Accessors

- Allows you to define the type of access that will be made to a buffer within a kernel.
- Access can be read-only, write-only, read-write-only.
- It is declared as: ***accessor nameAccesor(buffer, handler, permiso);***

```
int main(){
    queue q;
    std::vector<int> v(N);
    buffer buff(v);
    q.submit([&](handler& h){
        accessor a(buff, h, write_only);
        h.parallel_for(N,[=](auto i) {
            a[i] = i;
        });
    }).wait();
}
```

Tipo de acceso	Descripción
read_only	Read-only access
write_only	Write-only access
read_write	Read and write access

Structures for Parallelism

Basic kernel

- A kernel that runs a single task or thread can be expressed using single_task.

```
h.single_task([=]() {
            ...
});
```

Basic parallel kernel

- The functionality of a parallel kernel is exposed by the range, id, and item classes.
- Range: Used to describe the iteration space of parallel execution
- ID: Used as an index of an individual instance of a kernel in parallel execution
- Item: Represents an individual instance of a kernel function. Exposes additional functions on the properties of the execution (range, id, etc).

```
h.parallel_for(range<1>(1024), [=](item<1> i) {
            auto idx = i.get_id();
            auto R = i.get_range();
            ...
});
```

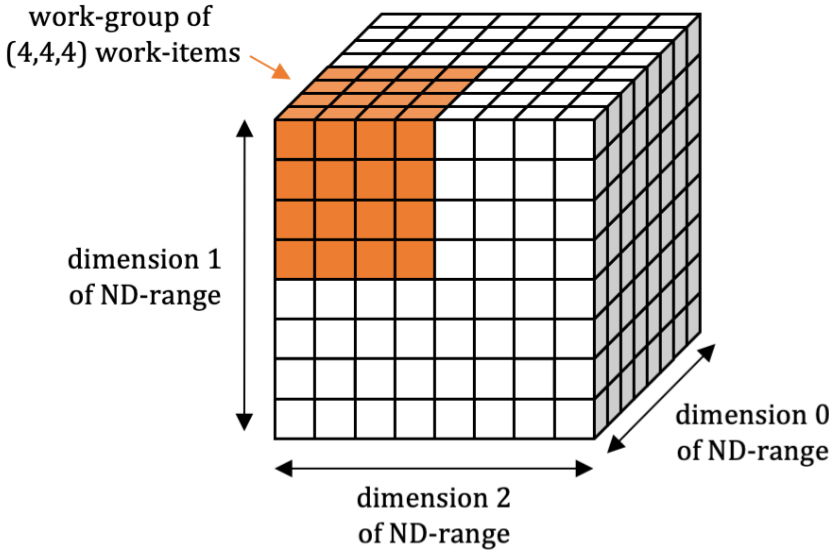
Parallel_for

- It allows to define a parallel loop that will be executed by many threads depending on the number of iterations.
- In its simplified version, the compiler performs the division of work (iterations) among threads.
- Declared as: h.parallel_for(range{N} , [=](id<1> idx) {
- N is the size of the range, and id is used to define the dimensions of the parallel work
- in the following example n represents the number of threads

```
int main(){
    queue q;
    buffer buff(v);
    q.submit([&](handler& h){
        accessor a(buff, h, write_only);
        h.parallel_for(N,[=](auto i) {
            a[i] = i;
        });
    }).wait();
}
```

ND-Range

- ND-Range it is a way of expressing parallelism with a high level of precision.
- It is used to map operations to compute units and threads.
- It allows to define the parallel work in multiple dimensions.
- Example of 3D-range:
-



ND-Range

ND-Range

- The functionality of ND-Range is expressed by the nd_range and nd_item classes.
- nd_range: Represents the group configuration using global and local range for each work-group.
- nd_item: Represents an individual instance of a kernel function. Exposes additional functions on the properties of the execution (range, id, etc).
-

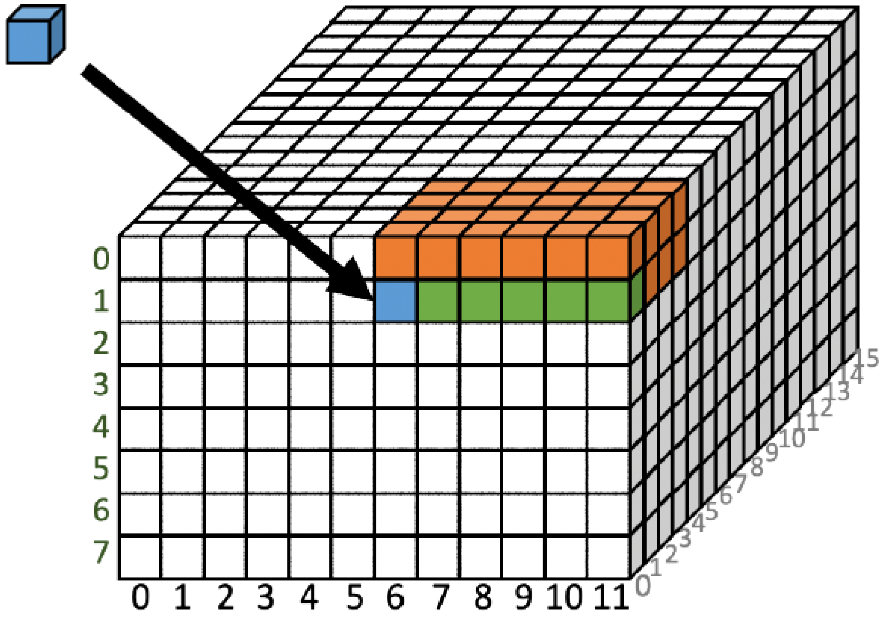
Global size

work-group size

```
h.parallel_for(nd_range<1>(range<1>(1024), range<1>(64)), [=](nd_item<1> item) {
            auto global_ = item.get_global_id();
            auto local_id = item.get_local_id();
            ...
});
```

ND-Range

- Example using: `nd_range<3>(range<3>({12,8,16}), range<3>({6,2,4}))`



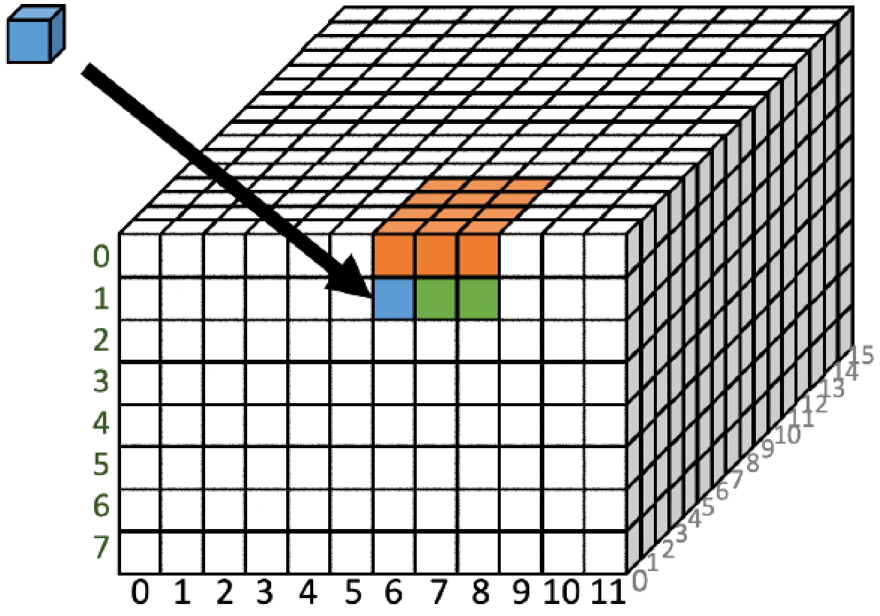

```
nd_range<3>({12,8,16}{6,2,4})
global range          {12, 8, 16}
global id             {6, 1, 0}
global linear id      784
group range           {2, 4, 4}
group                 {1, 0, 0}
group linear id       16
local range           {6, 2, 4}
local id              {0, 1, 0}
local linear id       4

subgroup group range          3
subgroup group id             {0}
subgroup local range          {16}
subgroup local id             {6}
subgroup uniform group range  3
subgroup max local range      {16}
```

Source: DPC++ Fundamentals Part 1. Colfax Research 2020

ND-Range

- Example using: `nd_range<3>(range<3>({12,8,16}), range<3>({3,2,4}))`



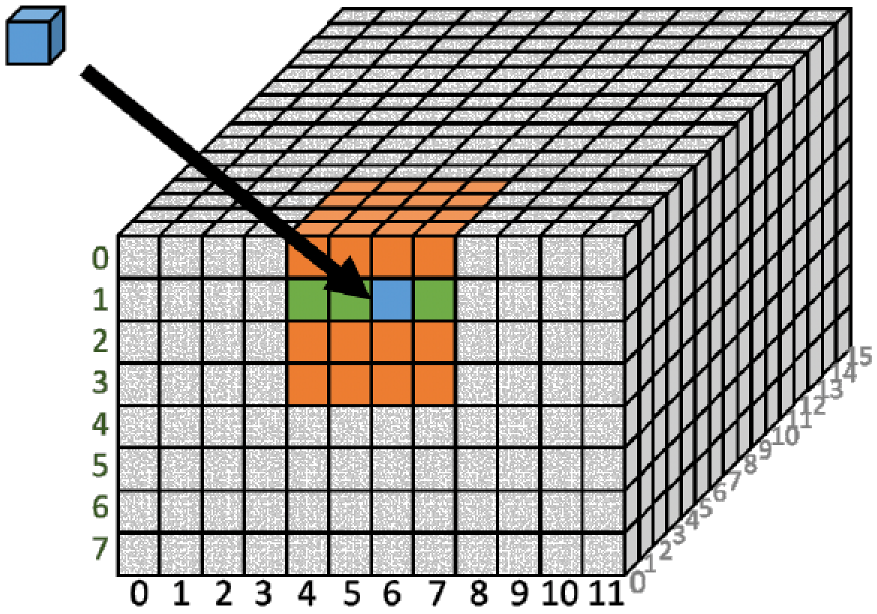

```
nd_range<3>({12,8,16}{3,2,4})
global range         {12, 8, 16}
global id            {6, 1, 0}
global linear id     784
group range          {4, 4, 4}
group                {2, 0, 0}
group linear id      32
local range          {3, 2, 4}
local id             {0, 1, 0}
local linear id      4

subgroup group range          2
subgroup group id             {0}
subgroup local range          {16}
subgroup local id             {3}
subgroup uniform group range  2
subgroup max local range      {16}
```

Source: DPC++ Fundamentals Part 1. Colfax Research 2020

ND-Range

- Example using: `nd_range<3>(range<3>({12,8,16}), range<3>({4,4,4}))`
- `group range` multiplied by the `local range` = `global range`

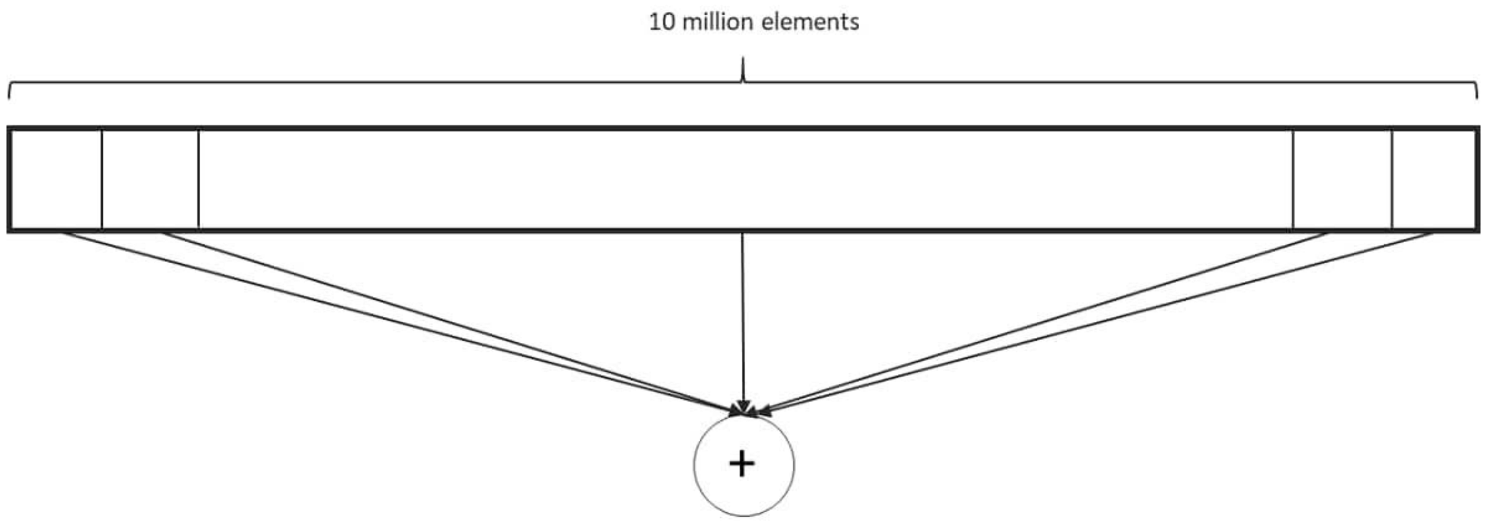



```
nd_range<3>({12,8,16}{4,4,4})
global range        {12, 8, 16}
global id           {6, 1, 0}
global linear id    784
group range         {3, 2, 4}
group               {1, 0, 0}
group linear id     8
local range         {4, 4, 4}
local id            {2, 1, 0}
local linear id     36

subgroup group range          4
subgroup group id             {0}
subgroup local range          {16}
subgroup local id             {6}
subgroup uniform group range  4
subgroup max local range      {16}
```

Source: DPC++ Fundamentals Part 1. Colfax Research 2020

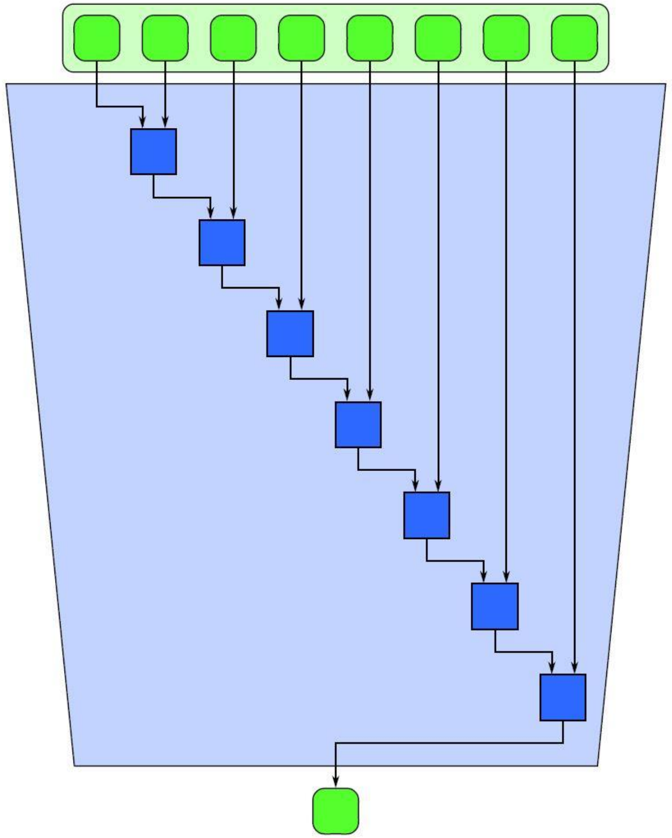
Reductions

- **Reduction:** Combines each item into a collection using a "combination function"
- Different orders of reduction are possible.
- Examples of combination function functions: add, mul, max, min, AND, OR, and XOR
- Example: Sum of all the elements of a vector
- 

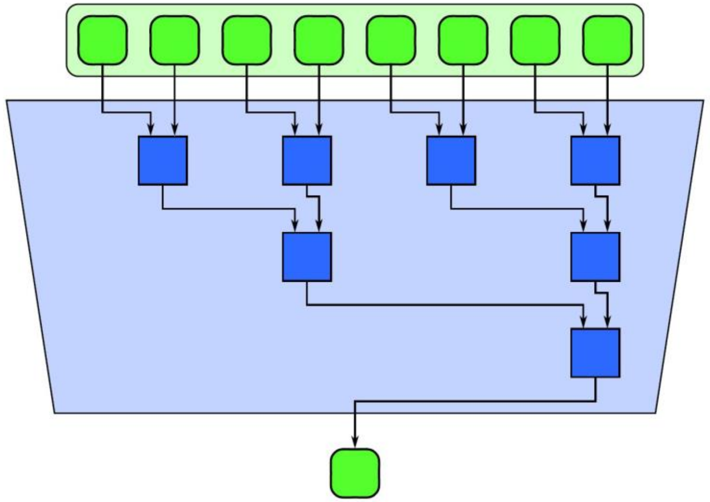
Reductions

- Sequential vs parallel:
-

Serial Reduction

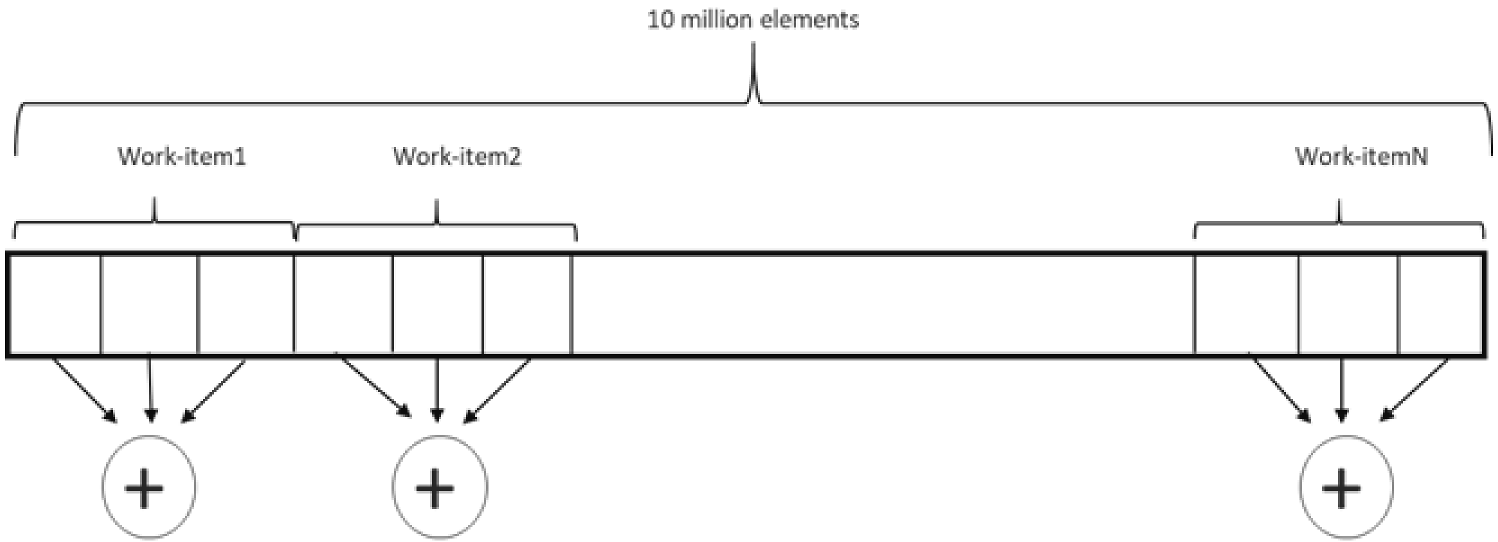


Parallel Reduction



Reductions

- A simple approach for reductions is to use an atomic variable and have all threads (or work-items) make direct changes. However, this is not a very efficient solution since it produces high containment and serialization of instructions.
- One solution is to divide the work into sub-groups, performing partial calculations to finally combine these results.
- Example:
-



Reductions

- In DPC++ there are two ways to define reductions when multiple threads modify a variable or shared memory position: reduction y reduce_over_group.
- 1. Using reduction:
  - Structure: reduction(buffer, handler, operation);
  - *buffer*: where the reduction will be saved
  - *handler*: kernel handler
  - *operation*: operation to be performed in the reduction. Example: plus, minimum, maximum, etc.
  -
  - Example: Perform sum reduction
  -

```
auto sumReduction = reduction(buffer, h, plus<>());
h.parallel_for(nd_range<1>(N, B), sumReduction [=](nd_item<1> item, auto& sum) {
            auto i = item.get_global_id();
            sum.combine(data[i]);
            ...
});
```

Reductions

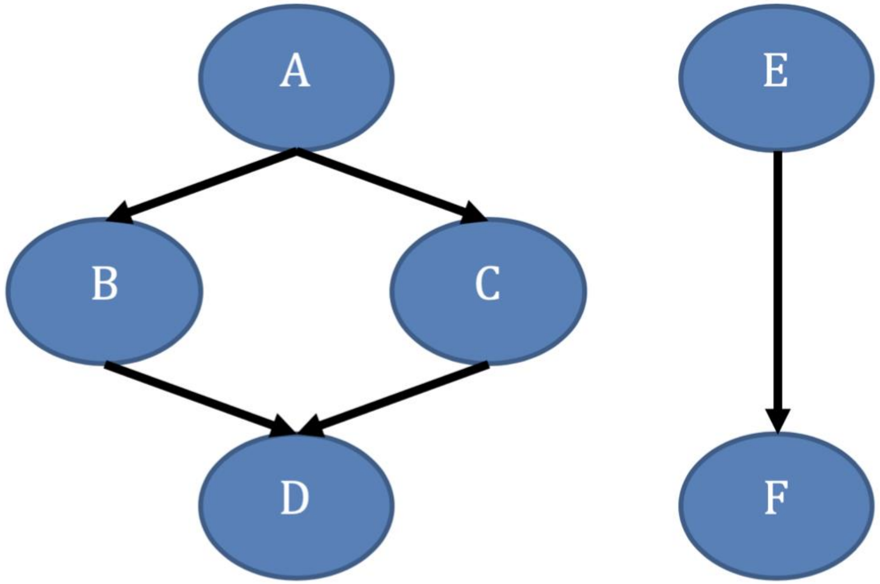
- 2. Using reduce_over group:
  - Structure: *reduce_over_group(work-group, data, operation);*
  - *work-group*: identifier number of the work-group or subgroup
  - *data*: variable to be processed by the reduction
  - *operation*: operation to be performed in the reduction. Example: plus, minimum, maximum, etc.
  -
  - Example: perform the sum reduction through work-groups.
  -

```
h.parallel_for(nd_range<1>(N, B), [=](nd_item<1> item) {
            auto work_group = item.get_group();
            auto i = item.get_global_id();
            int sum_wg = reduce_over_group(work_group, data[i], plus<>());
            ...
});
```

- See examples in Jupyter: https://devcloud.intel.com/oneapi/get_started/baseTrainingModules/

Dependencies

- The order in which kernels are executed can be crucial to the correctness of programs.
- The order of execution is directly linked to dependencies
- Examples:
  - A must be executed before B and C.
  - B and C require data processed by A.
  - D must be executed after B and C.
  - E must be executed before F.



Dependencies

- In DPC++ the order in which kernels are executed can be explicitly defined.
- **Wait**: Wait for a particular kernel to finish

```
int main(){
    queue q;
    auto kernel = q.submit([&](handler& h){
          . . .
    });
    kernel.wait();
}
```

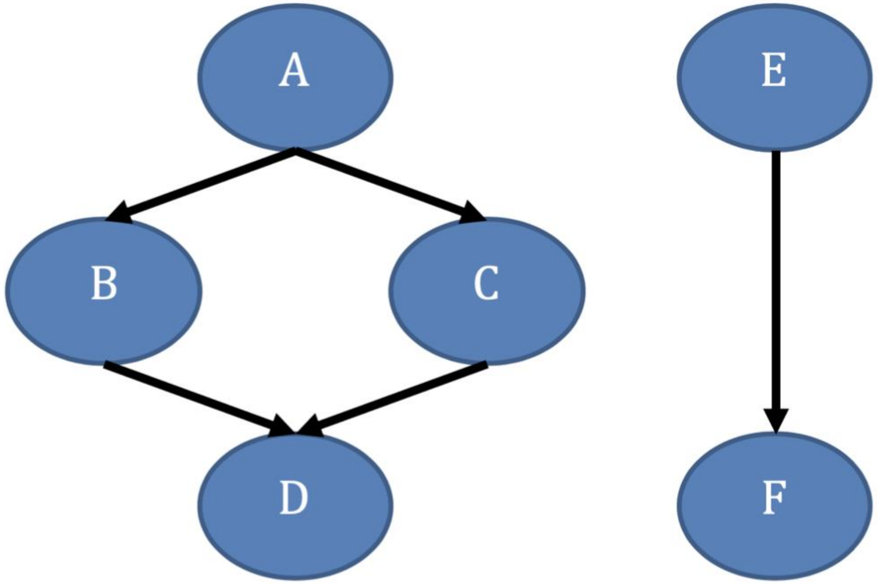
- **Depends_on**: Indicates that the current kernel depends on another kernel that must be run first
-

```
int main(){
    queue q;
    auto kernel1 = q.submit([&](handler& h){ … });

    auto kernel2 = q.submit([&](handler& h){
         h.depends_on(kernel1);
    });
}
```

Dependencies

- Example with depends_on:




```
int main(){
    queue q;
    auto A = q.submit([&](handler& h){  });

    auto B = q.submit([&](handler& h){
            h.depends_on(A);
    });

    auto C = q.submit([&](handler& h){
            h.depends_on(A);
    });

    auto D = q.submit([&](handler& h){
            h.depends_on(B);
            h.depends_on(C);
    });

    auto E = q.submit([&](handler& h){  });

    auto F = q.submit([&](handler& h){
            h.depends_on(E);
    });
}
```

References

- Intel Corp. Training for OneAPI https://www.intel.com/content/www/us/en/developer/tools/oneapi/training/overview.html
- Colfax. DPC++ Fundamentals https://www.colfax-intl.com/downloads/oneAPI_module03_DPCplusplusFundamentals1of2.pdf
- Reinders, J., Ashbaugh, B., Brodman, J., Kinsner, M., Pennycook, J., & Tian, X. (2021). Data Parallel C++: Mastering DPC++ for Programming of Heterogeneous Systems using C++ and SYCL (p. 548). Springer Nature.