



UNIVERSIDAD DEL BÍO-BÍO
FACULTAD DE CIENCIAS EMPRESARIALES

Introducción a DPC++

Computación Heterogénea

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

Ayudantes:

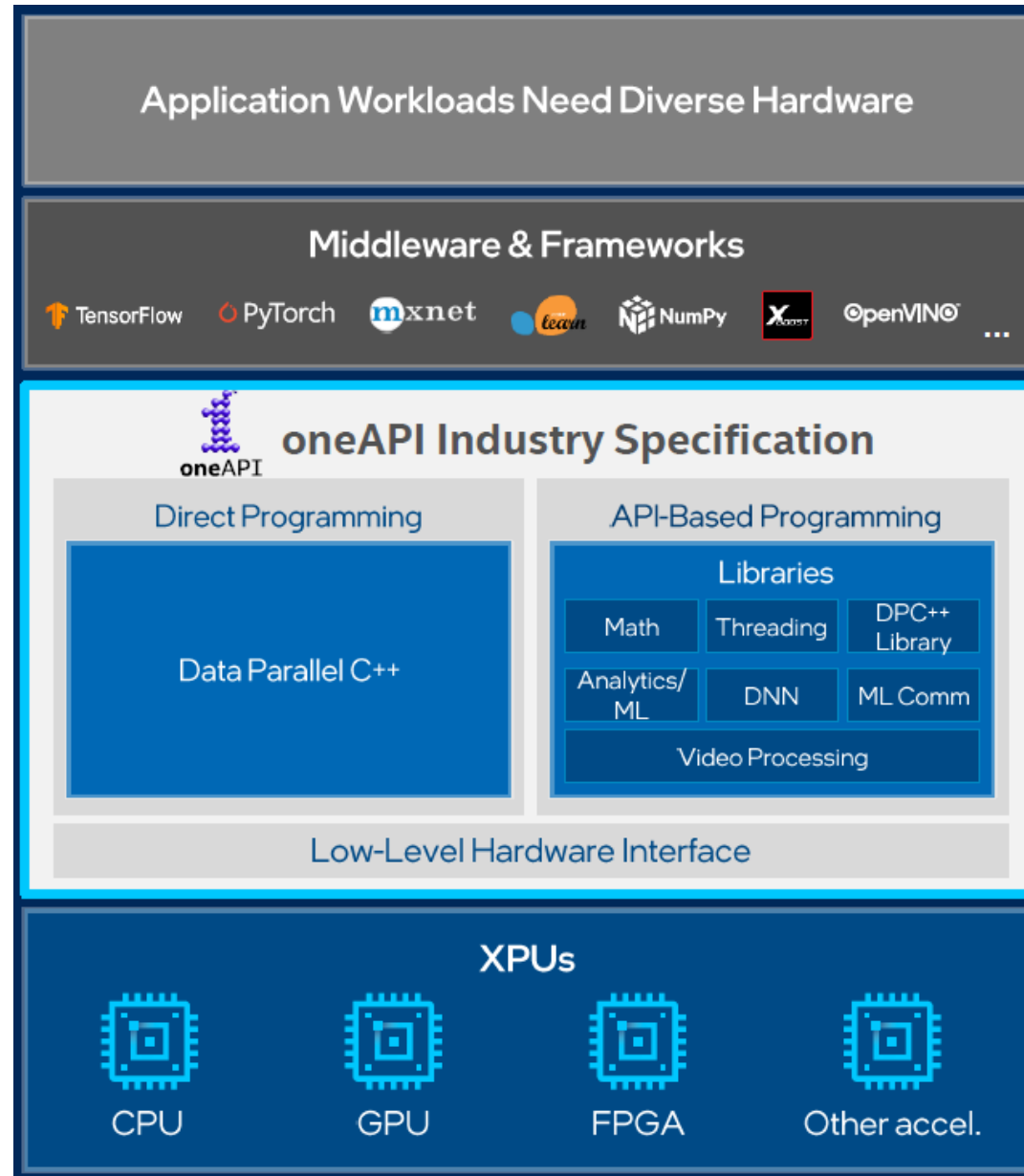
- Daniel López - daniel.lopez1701@alumnos.ubiobio.cl
- Sebastián González - sebastian.gonzalez1801@alumnos.ubiobio.cl

Página web del curso: <http://www.face.ubiobio.cl/~jfuentes/classes/ch>

DPC++

- SYCL es un modelo de programación de alto nivel para procesadores heterogéneos.
- SYCL define un estándar para que diferentes organizaciones realicen implementaciones.
- DPC++ es una implementación de SYCL.
- DPC++ es parte de la suite OneAPI

OneAPI



Léxica y semántica similar a C++

Ejemplo:

<code>#include <stdio.h></code>	comandos del preprocesador
---------------------------------------	----------------------------

<code>typedef float balance;</code>	definiciones de tipos
-------------------------------------	-----------------------

<code>void imprime();</code>	prototipos de funciones
------------------------------	-------------------------

<code>int cont=3; balance b;</code>	variable globales
-------------------------------------	-------------------

<pre>int main(void) { b=1000; imprime(); return 0; } void imprime(){ printf("cont = %d /n", &cont); printf("balance = %f /n", &b); }</pre>	funciones
---	-----------

Tipos de datos

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

Compilación

- Dpcpp es el compilador para DPC++.
- Puede funcionar como compilador cruzado, generando Código para plataformas distintas a la de desarrollo).
- Uso:
 - `$ dpcpp [opciones] [archivos] -lsycl`
- Opciones más usadas:

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

Programa en DPC++

- Código fuente único
 - Código del host y kernels de aceleradores heterogéneos puede ser incluidos en los mismos archivos fuentes.
- Principales funcionalidades:
 - `queue`
 - Trabajo objetivo
 - `malloc_shared`
 - Administración de datos
 - `parallel_for`
 - Paralelismo

host {
dispositivo {
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;
}
```

Programa en 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. Crear cola dispositivo
2. Crear buffers
3. Encolar kernel
4. Especificar parametros del kernel para su ejecución
5. Crear accesadores
6. Especificar función del kernel

Estructura de un programa en DPC++

- Aspectos a considerar en programas DPC++
 1. Decidir dónde el código se ejecutará
 2. Decidir modelo de ejecución paralela
 3. Transferencia de datos y sincronización

Principales clases en DPC++

Clase	Funcionalidad
<code>sycl::device</code>	Representa una CPU, GPU, FPGA u otro dispositivo específico que puede ejecutar SYCL kernels.
<code>sycl::queue</code>	Representa una cola en la cual kernels pueden ser agregados para su ejecución.
<code>sycl::buffer</code>	Encapsula buffer de memore que puede ser transferida entra host y dispositivo
<code>sycl::handler</code>	Usada para definir comandos de grupos que conectan buffers con kernels.
<code>sycl::accessor</code>	Usada para definir requerimientos de acceso desde kernels (sólo lectura, sólo escritura, etc).
<code>sycl::range</code> , <code>sycl::nd_range</code> <code>sycl::id</code> , <code>sycl::item</code> , <code>sycl::nd_item</code>	Representación de ejecución por rangos y ejecución individual en rangos.

Colas de ejecución en dispositivo (Queue)

- Las colas (queue) nos permiten conectar con los dispositivos (device).
- Con ellas enviamos kernels para ejecutar trabajo y mover datos.
- Se declara como: ***queue nombreCola;***

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

    return 0;
}
```

También se puede especificar dónde ejecutar 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

- Permite la declaración de memoria para ser utilizada por host y dispositivos.
- La forma más fácil de declararlos es indicando en su constructor la fuente de datos: arreglo, vector, puntero, etc.
- Se declara como: ***buffer nombreBuffer(data);***

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

    return 0;
}
```

Accessors

- Permite definir el tipo de acceso que se realizará a un buffer dentro de un kernel.
- El acceso puede ser sólo de lectura, sólo de escritura, de lectura y escritura.
- Se declara como: ***accessor nombreAccesor(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	Acceso solo de lectura
write_only	Acceso solo de escritura
read_write	Accesi de lectura y escritura

Accessors

- Permite definir el tipo de acceso que se realizará a un buffer dentro de un kernel.
- El acceso puede ser sólo de lectura, sólo de escritura, de lectura y escritura.
- Se declara como: ***accessor nombreAccesor(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	Acceso solo de lectura
write_only	Acceso solo de escritura
read_write	Acceso de lectura y escritura

Estructuras para Paralelismo

Kernel básico

- Un kernel que ejecute una sola tarea o hilo puede expresarse mediante el método `single_task`.

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


Kernel paralelo básico

- La funcionalidad de un kernel paralelo es expuesta mediante las clases range, id, e item.
- **Range**: usada para describir el espacio de iteración de la ejecución paralela
- **Id**: usado como índice de una instancia individual de un kernel en la ejecución paralela
- **Item**: representa una instancia individual de una función kernel. Expone funciones adicionales sobre las propiedades de la ejecución (rango, id, etc).

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

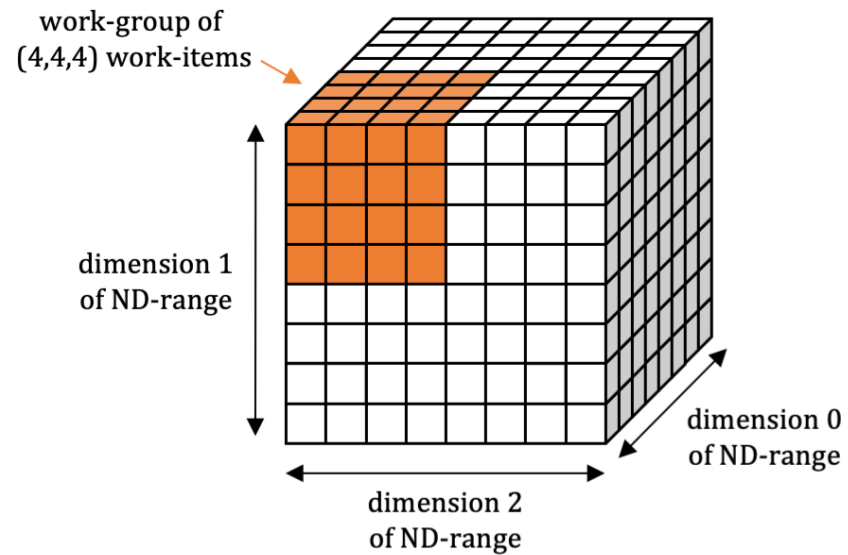
Parallel_for

- Permite definir un bucle paralelo que será ejecutado por muchos hilos dependiendo del número de iteraciones.
- En su versión simplificada, el compilador realiza la división del trabajo (iteraciones) en los hilos.
- Se declara como: **h.parallel_for(range{N}, [=](id<1> idx) {**
- N es el tamaño del rango, y id sirve para definir las dimensiones del trabajo paralelo
- En el siguiente ejemplo N representa el número de hilos

```
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 es una forma de expresar paralelismo con alto nivel de precisión.
- Sirve para mapear operaciones a unidades de cómputo e hilos.
- Permite definir el trabajo paralelo en múltiples dimensiones.
- Ejemplo de 3D-range:



ND-Range

ND-Range

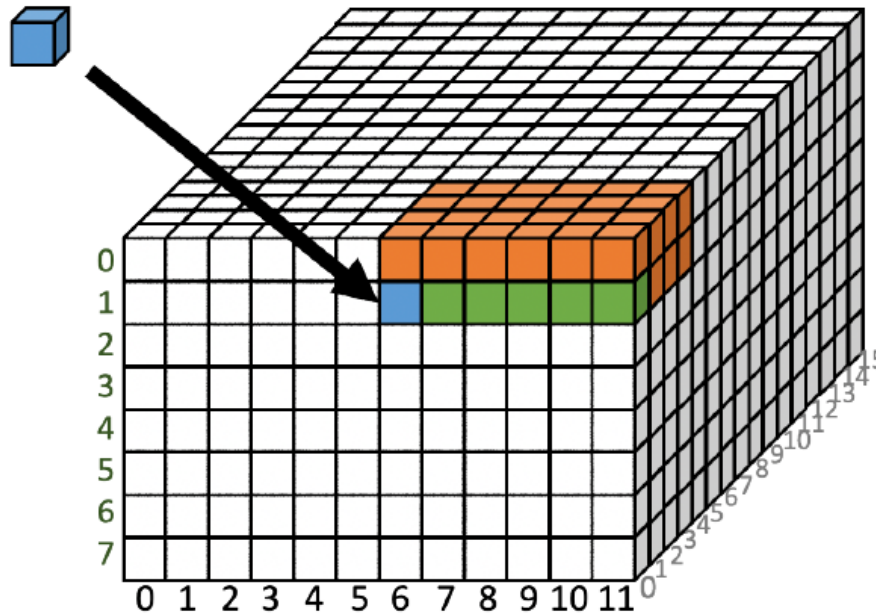
- La funcionalidad de ND-Range se expresa mediante las clases `nd_range` y `nd_item`.
- **`nd_range`**: representa un la configuración del grupo de ejecución usando rango global y local para cada work-group.
- **`nd_item`**: representa una instancia individual de una función kernel. Expone funciones adicionales sobre las propiedades del la ejecución (rango, id, etc).



```
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

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

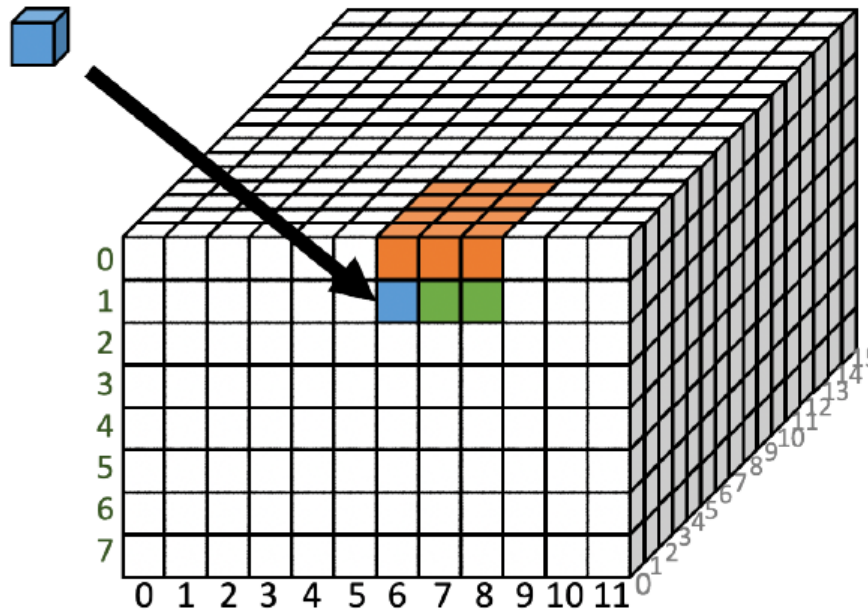


```
nd_range<3>({12,8,16},{6,2,4})
global range      {12, 8, 16}
global id         {6, 1, 0}
global linear id  784
group range       {2, 4, 4}
group             {1, 0, 0}
group linear id   16
local range       {6, 2, 4}
local id          {0, 1, 0}
local linear id   4

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}
```

ND-Range

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

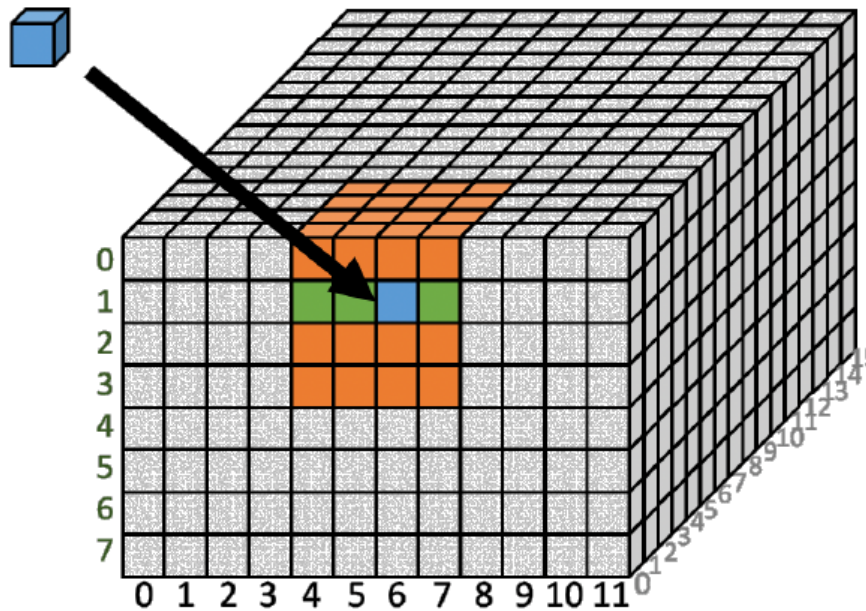


```
nd_range<3>({12,8,16},{3,2,4})
global range      {12, 8, 16}
global id         {6, 1, 0}
global linear id  784
group range       {4, 4, 4}
group             {2, 0, 0}
group linear id   32
local range       {3, 2, 4}
local id          {0, 1, 0}
local linear id   4

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}
```

ND-Range

- Ejemplo usando: `nd_range<3>(range<3>({12,8,16}), range<3>({4,4,4}))`
- El `group range` multiplicado por el `local range` da el `global range`

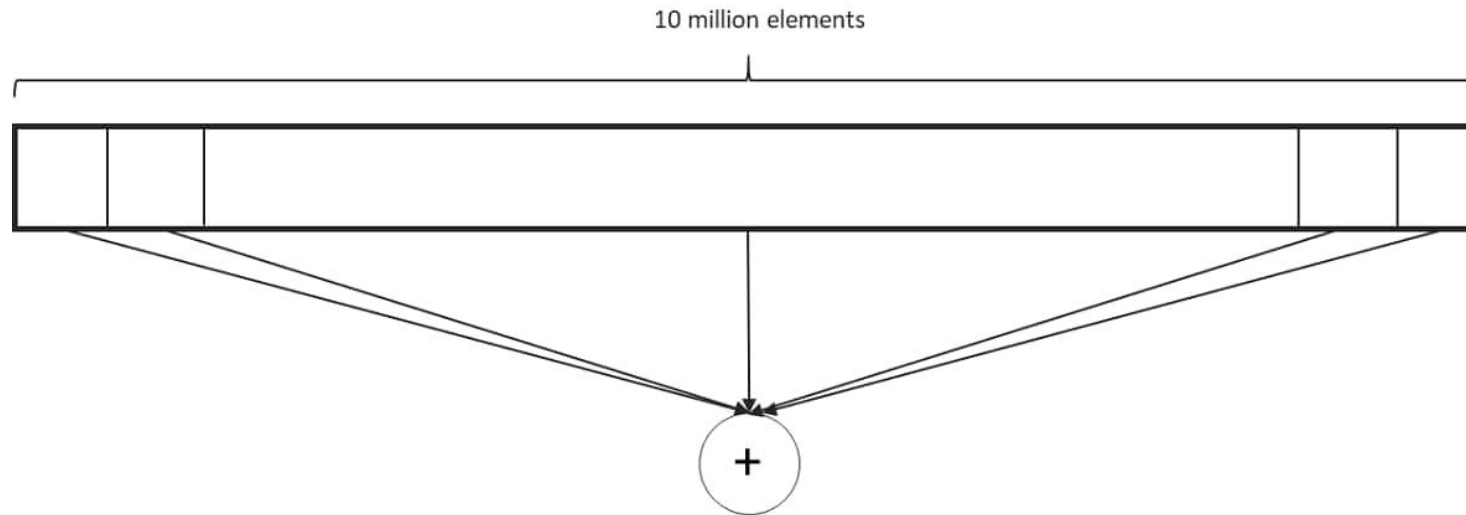


```
nd_range<3>({12,8,16},{4,4,4})
global range      {12, 8, 16}
global id         {6, 1, 0}
global linear id  784
group range       {3, 2, 4}
group             {1, 0, 0}
group linear id   8
local range       {4, 4, 4}
local id          {2, 1, 0}
local linear id   36

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}
```

Reducciones

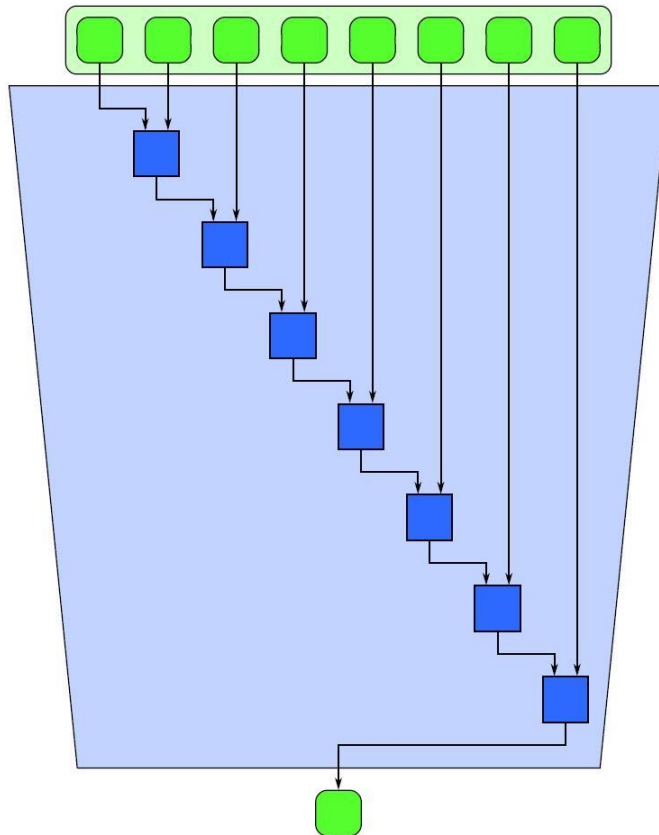
- **Reducción:** Combina cada elemento en una colección usando una “función de combinación”
- Diferentes órdenes de la reducción son posibles.
- Ejemplos de funciones de combinación: add, mul, max, min, AND, OR, y XOR
- Ejemplo: suma de todos los elementos de un vector



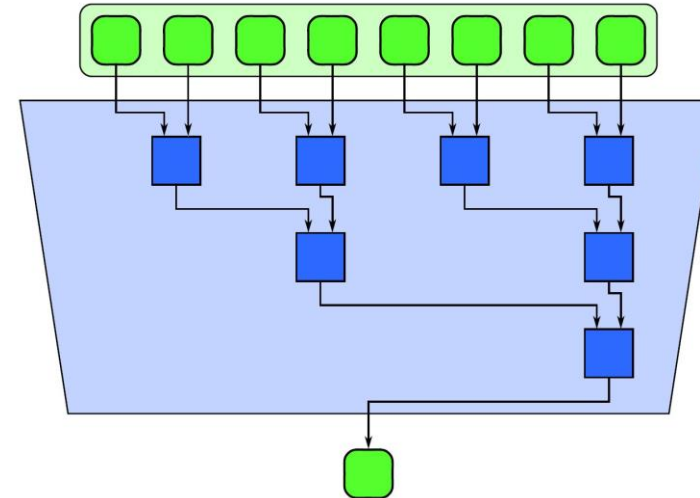
Reducciones

- Secuencial vs paralelo:

Reducción Serial

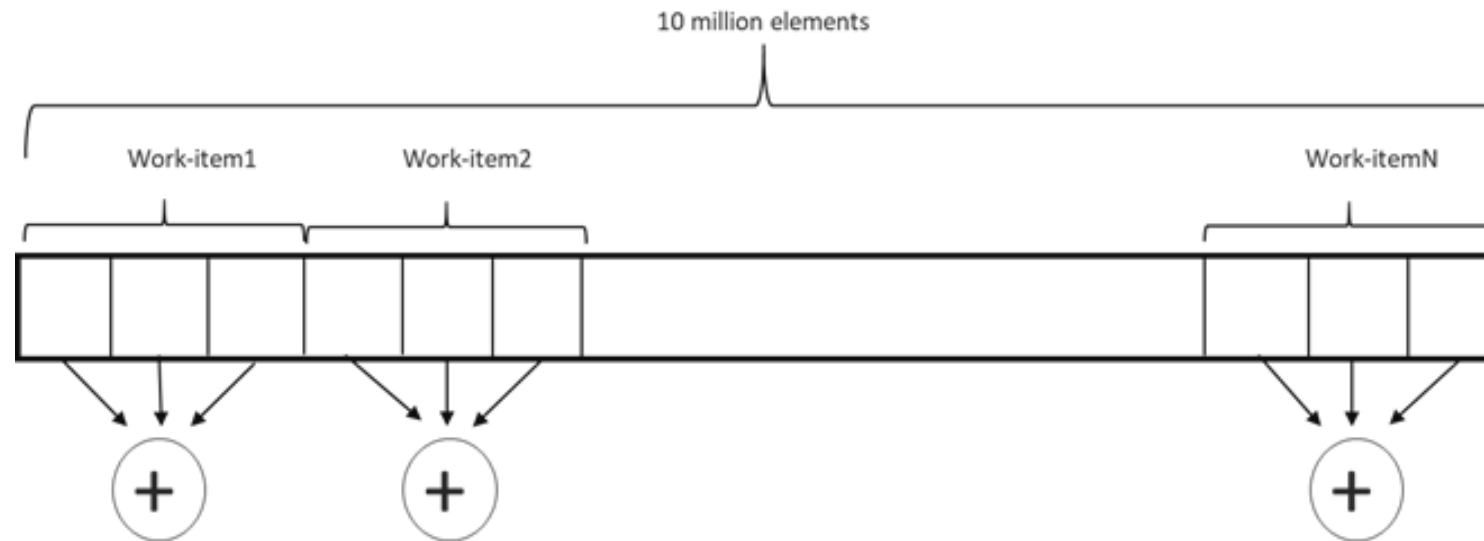


Reducción Paralela



Reducciones

- Un enfoque simple para reducciones es utilizar una variable atómica y que todos los hilos (o work-ítem) realicen cambios directos. Sin embargo esta no es una solución muy eficiente ya que produce **alta contención** y **serialización de instrucciones**.
- Una solución es dividir el trabajo en sub-grupos, realizando calculos parciales para finalmente combinar estos resultados.
- Ejemplo:



Reducciones

- En DPC++ existen dos formas de definir reducciones cuando múltiples hilos modifican una variable o posición de memoria compartida: `reduction` y `reduce_over_group`.
- 1. Usando `reduction`:
 - Estructura: `reduction(buffer, handler, operation);`
 - `buffer`: donde se guardará la reducción
 - `handler`: manejador del kernel
 - `operation`: operación a ser realizada en la reducción. Ejemplo: plus, minimum, maximum, etc.
- Ejemplo: realizar la reducción de suma

```
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]);  
    ...  
});
```

Reducciones

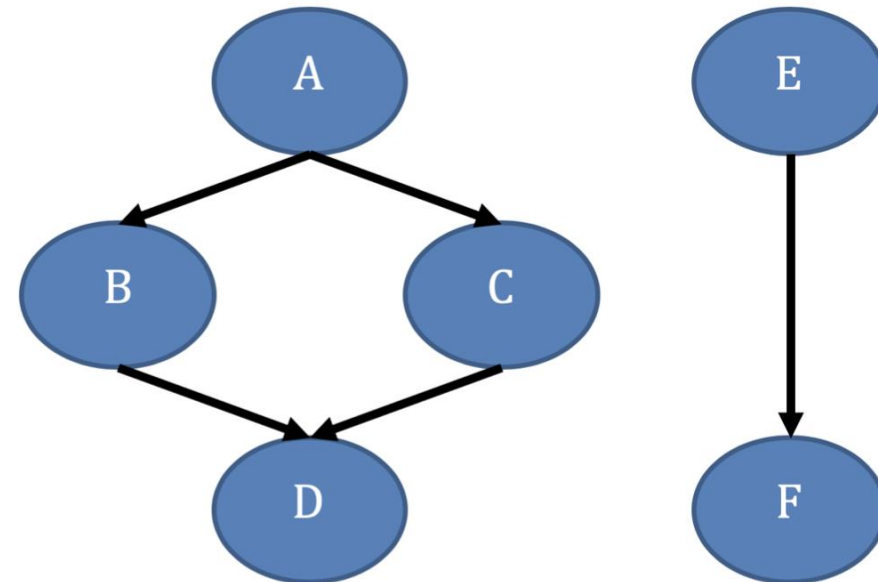
- 2. Usando **reduce_over_group**:
 - Estructura: `reduce_over_group(work-group, data, operation);`
 - **work-group**: número identificador del work-group o subgroup
 - **data**: variable a ser procesada por la reducción
 - **operation**: operación a ser realizada en la reducción. Ejemplo: plus, minimum, maximum, etc.
- Ejemplo: realizar la reducción de suma a través de 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<>());  
    ...  
});
```

- Ver ejemplos en Jupyter: https://devcloud.intel.com/oneapi/get_started/baseTrainingModules/

Dependencias

- El orden en el que se ejecutan los kernels pueden resultar cruciales para la correctitud de programas.
- El orden de ejecución está directamente ligado a dependencias
- Ejemplos:
 - A debe ejecutarse antes que B y C.
 - B y C requieren datos procesados por A.
 - D debe ejecutarse después de B y C.
 - E debe ejecutarse antes que F.



Dependencias

- En DPC++ el orden en el cual se ejecutan kernels puede ser definido explícitamente.
 - Wait: Espera por un kernel particular que termine

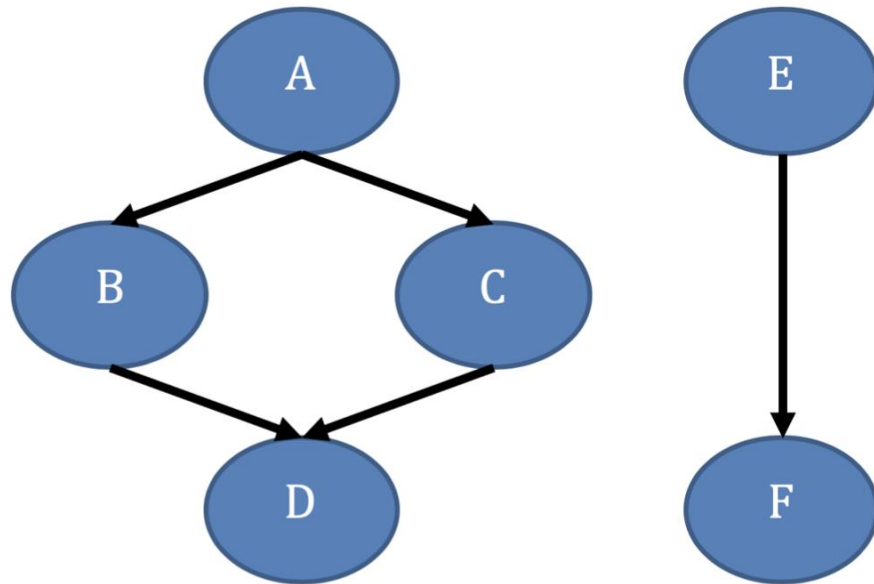
```
int main(){  
    queue q;  
    auto kernel = q.submit([&](handler& h){  
        . . .  
    });  
    kernel.wait();  
}
```

- Depends_on: Indica que el kernel actual depende de otro que debe ejecutarse primero

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

Dependencias

- Ejemplo con 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);
    });
}
```

Referencias

- 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.