Tema 6.2 Programación Aceleradores por medio de Directivas (OpenMP)

Computación de Altas Prestaciones

Carlos García Sánchez

UCM

28 de octubre de 2022

- "OpenMP 5.2", https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5-2.pdf
- "Ejemplo de OpenMP 5.2", https://www.openmp.org/wp-content/uploads/openmp-examples-5-2.pdf

Outline

1 Intro

2 GPUs

3 OpenMP offload

4 OpenMP Data

5 Paralelismo en OpenMP-offload

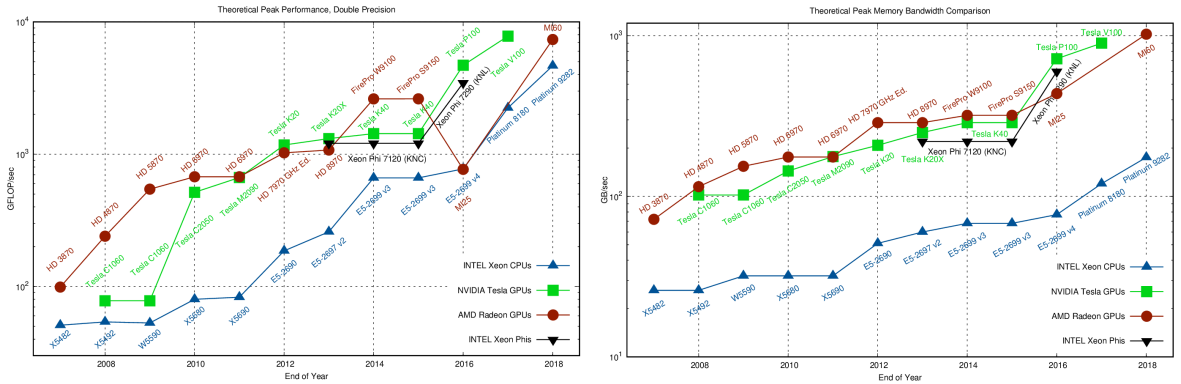
6 Otros aspectos

Motivación

- CPU
  - Propósito general
  - Bueno para el procesamiento en serie
  - Excelente para el paralelismo de tareas
  - Baja latencia por subproceso
  - Caché y control dedicados de área grande
- GPU
  - Altamente especializado para el paralelismo
  - Ideal para paralelismo de datos
  - alto rendimiento
  - Cientos de unidades de ejecución de punto flotante

Motivación

- GPU vs CPU (rendimiento)
  - GPU vista como co-procesadores de la CPU (**alto parelelismo**)
  - Descargan cómputo lanzado en *kernel* (lanzado asíncronamiente)
  - CPU y GPU puede trabajar concurrentemente



Motivación

- Desafíos: muchos modelos de programación
  - Más abstración vs Más Rendimiento

+++ Abstraction					+++ Performance
python	C/C++ Fortran	OpenMP (Cores)	OpenMP target OpenACC	OpenCL CUDA	Vector Intrs. GPU Intrinsics

Retos

1. (Variedad): Muchos lenguajes con sus toolchains, versión a mantener e integrar
2. (Performance): Desarrollo de app con alto rendimiento habitualmente conlleva desarrolladores especializados
3. (Porting): Algunas abstracciones ofrecen soluciones para alto-rendimiento en diferentes arquitecturas

Algunas comparaciones

- Algunos modelos de programación para aceleradores
- OpenMP permite aplicar técnicas de programación incremental para sistemas heterogeneos (ej: accelerator offloading, tasks...)
- Lista de compiladores que soportan OpenMP-offload

	CUDA	OpenACC	OpenMP (5.0)	SYCL
Language	C/C++	C/C++ Fortran	C/C++ Fortran	C/C++
Prog. Style		pragmas	pragmas	C++11 lambdas
Parallelism	SIMT	SIMD, Fork/join CUDA	SPMD, SIMD Tasks, Fork/join, CUDA	OpenCL
Licensing	Proprietary	Few comp.	Open-source	Open-source
Abstraction	Low	High	High	Medium

Un poco de historia

- Modelos de programación basados en directivas en **aceleradores**
  - Anotaciones en el código mediante directiva para indicar código a descargar en el acelerador

OpenACC

- Creado en 2011. Última versión 3.1 (Nov20)
- Mayoritario en GPUs de NVIDIA
- Porland Group Compiler (PGI) comienza con soporte para OpenACC
  - NVIDIA compra PGI en 2013

Un poco de historia

- Modelos de programación basados en directivas en **aceleradores**
  - Anotaciones en el código mediante directiva para indicar código a descargar en el acelerador

OpenMP

- Paralelismo multithreading
- Suporte inicial para aceleradores en v4.0 (2013)
  - Mejoras y extensiones en v4.5 (2015), 5.0 (2018), 5.1 (2020 y 5.2 (2021)
- Razonable rendimiento con esfuerzo de código moderado

Un poco de historia

- A principios de la década de 1990, los proveedores de sistemas de memoria compartida suministraron sus propias extensiones de programación basadas en directivas:
  - El usuario especifica los bucles a paralelizar con directivas
  - Los compiladores de los fabricantes son los responsables de paralelizar los bucles de forma automática en los sistemas SMP
- En 1997 el estándar OpenMP aparece [1]

[1] OpenMP standard https://www.openmp.org/

Versiones

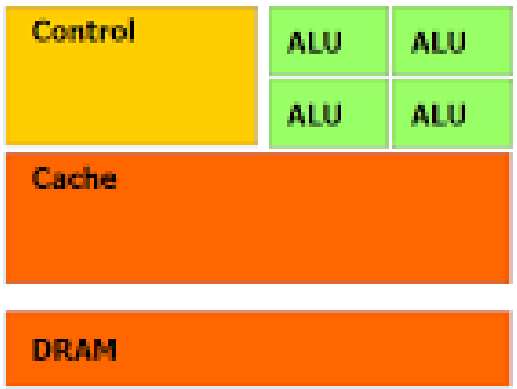
Date	Version
Oct 1997	Fortran 1.0
Oct 1998	C/C++ 1.0
...	...
Jul 2013	OpenMP 4.0
Nov 2015	OpenMP 4.5
Nov 2018	OpenMP 5.0
Nov 2020	OpenMP 5.1
Nov 2022	OpenMP 5.2

OpenMP Application Programming Interface (API)

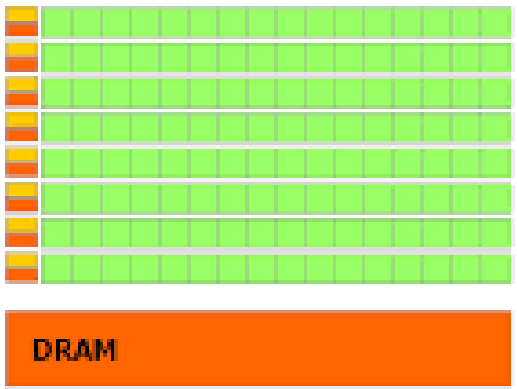
- Standard de-facto, OpenMP API version 5.0 (released in Nov18)
- API para C/C++ y Fortran para programación paralela en sistemas de memoria compartida
- Basado en directivas (pragmas en C/C++)
- Portable entre las diferentes plataformas y fabricantes
- Soporta varios tipos de paralelismo (threads)
  - For-loop: asigna grupo de iteraciones a los hilos
  - Sections: las secciones son ejecutadas por un único hilo
  - Tasks: un hilo crea una nueva tarea (nested parallelism)
  - **Heterogenous**: descarga de trabajo en el acelerador

Rendimiento GPU

- Por qué usar una GPU?
- La tendencia actual es que las GPUs ofrecen más rendimiento paralelo que los GPP
- Muchísimos cores sencillos vs pocos cores complejos (CPU)
  - E.g.: NVIDIA Volta ofrece 4.4× de ancho de banda y 1.9× de rendimiento en FLOPSs que un Intel Xeon dual socket (Skylake)



CPU

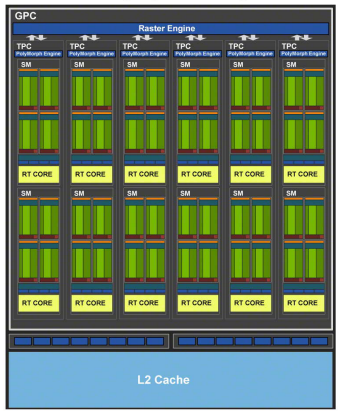


GPU

GPU performance

- GPUs hechas con muchísimos cores. NVIDIA los denomina Streaming Multiprocessors (SMs):
  - V100 tiene 80 SMs
  - P100 tiene 56 SMs
  - A100 tiene 128 SMs
- GPUs de Intel se denominan Execution Units (EU) que a su vez incorporan unidades SIMD

Nvidia Turing Graphics Processing Cluster



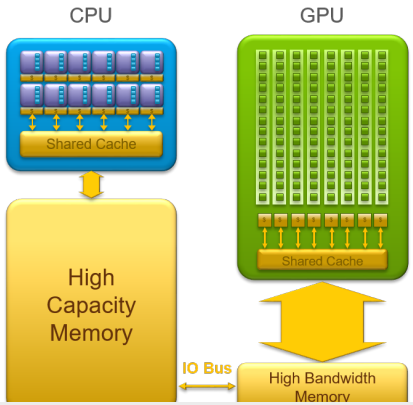
Intel Gen11 Graphics Slice

GPU architecture

- En el caso de Intel, las GPUs son modulares agrupadas estructuras de Slice y Subslice
  - Ej: Intel Gen9 Slice=3 subslices, Gen11 Slice=8 subslices
    - 1 Subslice = 8 EUs
  - Cada subslice contiene un Unit-Dispacher y una Shared Local Memory (SLM) de 64KB
  - Cada EU tiene 2 ALUs SIMD-128bits:
    - 16xFP32 simultaneamente en cada EU: 2ALUs*SIMD-4* 2 Ops (Add+Mul)
    - 32xFP16 en cada EU: 2ALUs*SIMD-8* 2 Ops (Add+Mul)

GPU vs CPU

- GPUs están *optimizadas para el throughput*, CPUs están *optimizadas para latencia*
- Throughput optimised también llamado como **latency tolerant** = **high latency hidding**
- GPUs ejecutan muchas operaciones al mismo tiempo (on-flight)
- Lo que significa que necesitan muchas (pero muchas) operaciones en paralelo...

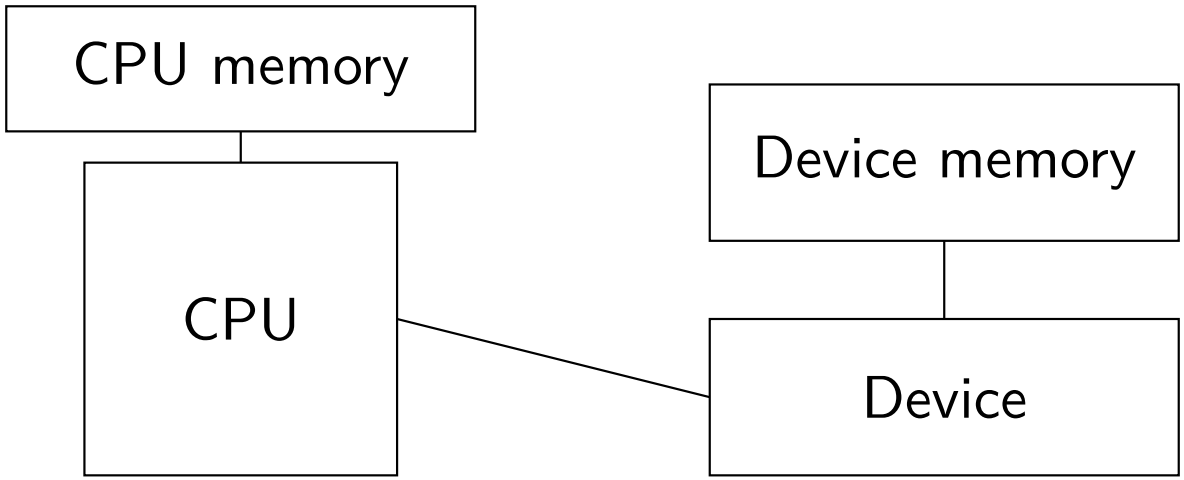


GPU vs CPU

- GPUs están *optimizadas para el throughput*, CPUs están *optimizadas para latencia*
- Throughput optimised también llamado como **latency tolerant** = **high latency hidding**
  - GPUs procesan muchas operaciones al mismo tiempo (on-flight)
  - Lo que significa que necesitan muchas (pero muchas) operaciones ...
- **Alto grado de paralelismo es obligado**

Modelo dispositivo

- Desde OpenMP 4.0/4.5 la API tiene soporte para aceleradores/coprocesadores
- Modelo de dispositivo:
  - Un host "tradicional"
  - Múltiples aceleradores/coprocesadores del mismo tipo donde descargar el trabajo
  - Dispositivos tienen su propio espacio de memoria



Modelo de ejecución

- La ejecución comienza en el host comúnmente conocido como CPU
  - ¡Espacio de memoria *no* compartido!
- .... y una parte de código (*kernels*) es descargado en el device

OpenMP offload region

- `#pragma omp target`
  - La clausula target descarga el código del kernel en el device

Modelo de ejecución

- Contrucción `target`

omp_target.c

```
void axpy_target()
{
    double x[SIZE];
    double y[SIZE];
    double a=1.0;

    double t0 = get_time();                          } ⟶ Host

    #pragma omp target device(0)
    #pragma omp parallel for firstprivate(a)         } ⟶ Device
    for (int i=0; i<N; i++)
        y[i] = a*x[i]+y[i];

    double t1 = get_time();
    printf("Time of kernel:%lf s\n", t1-t0);         } ⟶ Host
}
```

Construcción target

- Lleva el código del kernel y lo descarga en el device
  - OJO: necesita otras #pragmas adicionales para expresar el paralelismo dentro del dispositivo
- Otras clausulas como el movimiento de datos son también necesarias

C syntax

```
#pragma omp target [clause]
```

Fortran syntax

```
!$omp target [clause]
```

Contrucción `target`

- Ejecutar el kernel en el dispositivo o *device*

omp target

#pragma omp target [clause] new-line

- Clause:
  - device(...)
  - map(...)
  - if(...)

target if

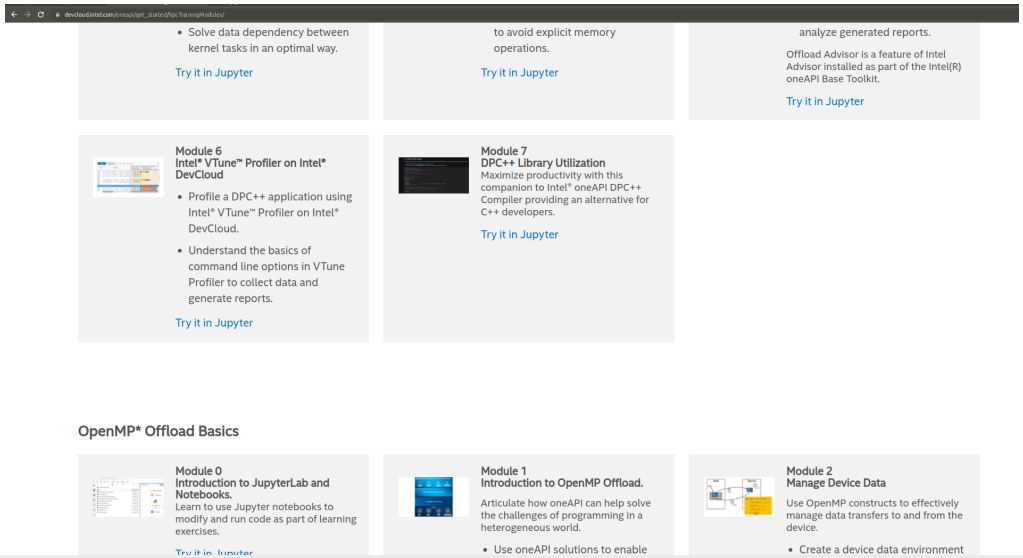
- La clausula `if` en la construcción `target` indica si el kernel es ejecutado en el *host* o en el *device*

omp_target_if.c

```
#define THRESHOLD1 1000000
#define THRESHOLD2 1000
extern void init(float*, float*, int);
extern void output(float*, int);

void vec_mult(float *p, float *v1, float *v2, int N)
{
    int i;
    init(v1, v2, N);
    #pragma omp target if(N>THRESHOLD1) map(to: v1[0:N], v2[:N])\
        map(from: p[0:N])
    #pragma omp parallel for if(N>THRESHOLD2)
    for (i=0; i<N; i++)
        p[i] = v1[i] * v2[i];

    output(p, N);
}
```

Hand-on

1. Conéctate al Intel DevCloud
2. ... pero vamos a trabajar con **OpenMP Offload Basics**
   - https://devcloud.intel.com/oneapi/get_started/hpcTrainingModules/
3. Selecciona el cuaderno de jupyter **Module 1 Introduction to OpenMP Offload.**



Movimiento de datos

Recuerda

- La memoria *no* está compartida entre el host y el device
- OpenMP usa una combinación de movimiento de datos implício y explícito
- NOTA: el moviento de datos es un destructor de rendimiento
  - V100 tiene 900 GB/s de ancho de banda con memoria
  - El PCIe Host-Device logra 32 GB/s de pico
  - **¡¡¡Minimiza la transferencia entre memorias!!!**

Regiones de datos

- Los datos deben de ser mapeado/copiado/transferidos entre las memoria del host y el device
- Puede existir variables con el mismo nombre que se refieran al espacio de memoria del host y del device
  - El compilador averigua donde están emplazadas y las usa

```
#pragma omp target
for (int i=0; i<N; i++)
    y[i] = a*x[i]+y[i];
```

- El runtime de OpenMP y el compilador deben de trabajar con copias de `x, y` en el espacio de memoria del host y del device
- En realidad duplica las variables y las "renombra": `x_host, y_host` y `x_device, y_device` en el host y en el device respectivamente

Movimiento de datos

- El mapeo/copia/transferencia de datos se produce entre el host y el device cuando
  - A la entrada y salida de una región `target`: conlleva un movimiento de datos implícito
  - Cuando aparece la pragma `target enter/exit data`
  - En la construcción `target data map`
  - En la construcción `update`

Construcción target enter data

- Datos no-estructurados pueden crearse/eliminarse en el dispositivo en cierto puntos
  - Como en los esquema pila (e.j. para los métodos constructores/destructores de C++)
- El constructor crea un vector con `target enter data`. No hay copia desde el host
- El destructor elimina los datos del dispositivo con `target exit data`

omp_target_exit.c

```
#include <stdlib.h>
typedef struct {
  double *A;
  int N;
} Matrix;

void init_matrix(Matrix *mat, int n)
{
   mat->A = (double *)malloc(n*sizeof(double));
   mat->N = n;
   #pragma omp target enter data map(alloc:mat->A[:n])
}
void free_matrix(Matrix *mat)
{
   #pragma omp target exit data map(delete:mat->A[:mat->N])
   mat->N = 0;
   free(mat->A);
   mat->A = NULL;
}
```

Clausula map

- Especifica la transferencia de datos entre el host y el acelerador en la región `target`
  - El tamaño de los arrays deben especificarse para conocer la cantidad de información a copiar

C syntax

```
#pragma target map(...)
```

Fortran syntax

```
!$omp target map(...)
   ...
```

Clausula map

- La dirección está definida desde el punto de vista del *host*
- `map(to: x)`: en la entrada de la región, copia datos desde el host al device
- `map(from: x)`: en la salida de la región, copia datos del device al host
- `map(tofrom: x)`: ambas `map(to: ...)` y `map(from: ...)`
- `map(alloc: x)`: reserva espacio en la memoria del dispostivo **sin** copiar datos
- `map(delete: x)`: libera espacio en el dispositivo (cuando el contador de referencias llega a 0)
- `map(release: x)`: decrementa el contador de referencias de una variable

Clausula map

- La dirección está definida desde el punto de vista del *host* [2]

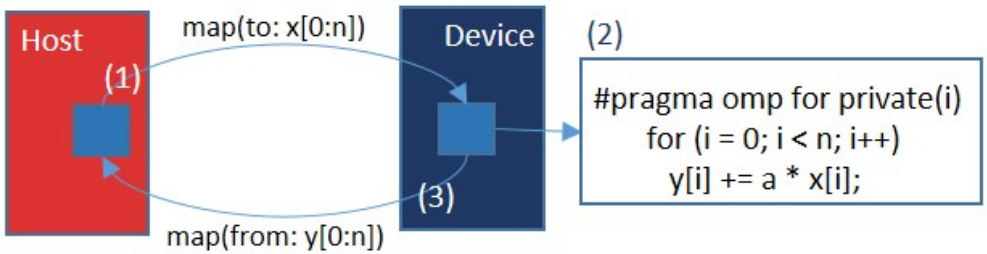
map-type	target	target data	target enter data	target exit data
alloc	X	X	X	
to	X	X	X	
from	X	X		X
tofrom	X	X		
delete				X
release				X

[2] Map Clause https://www.openmp.org/spec-html/5.0/openmpsu109.html#x142-6190002.19.7.1

Clausula map

omp_target.c

```
    #pragma omp target data map(to:x[0:n]) map(tofrom:y[0:n])
    {
    #pragma omp target
    #pragma omp parallel for
    for (int i = 0; i < n; ++i){
        y[i] = a*x[i] + y[i];
    }
}
```



Clausula map

omp_target_unstructured_data.c

```
    #pragma omp target enter data map(to:x[0:n])
    #pragma omp target enter data map(to:y[0:n])

    {
    #pragma omp target
    #pragma omp parallel for
    for (int i = 0; i < n; ++i){
        y[i] = a*x[i] + y[i];
    }
    #pragma omp target update from(y[0:n])

    #pragma omp target exit data map(release:x[0:n])
    #pragma omp target exit data map(release:y[0:n])
}
```

Construcción update

update

#pragma omp target update [clause]

- Clause:
  - to(...)
  - from(...)
  - device(...)
  - if(...)

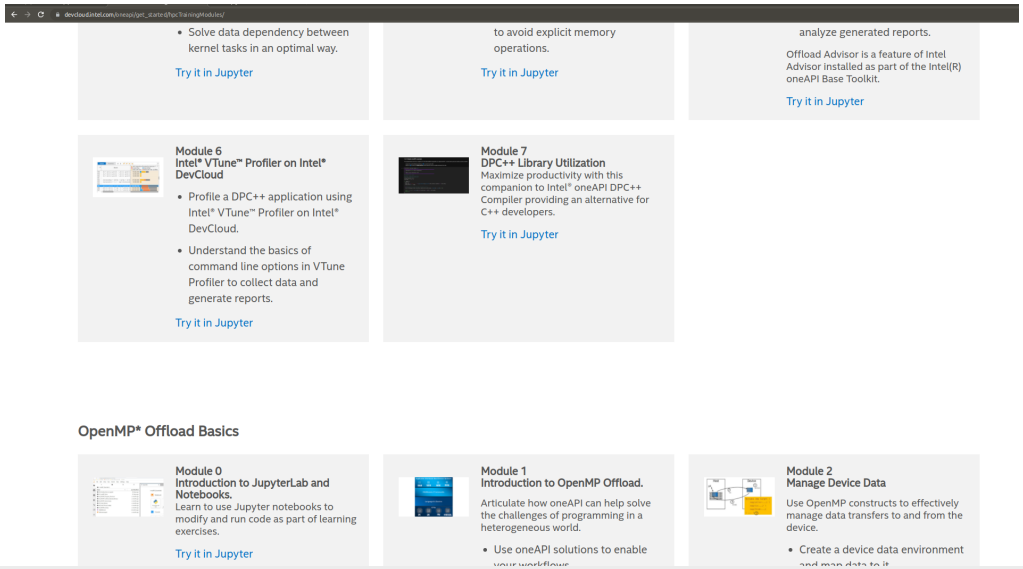
Construcción update

omp_update.c

```
extern void init(float *, float *, int);
extern void init_again(float *, float *, int);
extern void output(float *, int);

void vec_mult(float *p, float *v1, float *v2, int N)
{
    int i;
    init(v1, v2, N);
    #pragma omp target data map(to: v1[:N], v2[:N]) map(from: p[0:N])
    {
        #pragma omp target
        #pragma omp parallel for
        for (i=0; i<N; i++)
            p[i] = v1[i] * v2[i];
        init_again(v1, v2, N);
        #pragma omp target update to(v1[:N], v2[:N])
        #pragma omp target
        #pragma omp parallel for
        for (i=0; i<N; i++)
            p[i] = p[i] + (v1[i] * v2[i]);
    }
    output(p, N);
}
```

Hand-on

1. Conéctate al Intel DevCloud
2. ... pero vamos a trabajar con **OpenMP Offload Basics**
   - https://devcloud.intel.com/oneapi/get_started/hpcTrainingModules/
3. Selecciona el cuaderno de jupyter **Module 2 Manage Device Data.**



Datos no estructurados

- Regiones de datos no estructurados permiten manejar casos en los que la asignación y la liberación se realizan en un ámbito diferente
  - `enter data` define el comienzo de región de datos no-estructurados
    - C/C++: `#pragma omp enter data [clauses]`
  - `exit data` define salida de datos nos estructurados
    - C/C++: `#pragma omp exit data [clauses]`

Datos no estructurados

unstructure_data.c

```
typedef struct {
   double *A;
   int N;
} Matrix;

void init_matrix(Matrix *mat, int n)
{
   mat->A = (double *)malloc(n*sizeof(double));
   mat->N = n;
   #pragma omp target enter data map(alloc:mat->A[:n])
}

void free_matrix(Matrix *mat)
{
   #pragma omp target exit data map(delete:mat->A[:mat->N])
   mat->N = 0;
   free(mat->A);
   mat->A = NULL;
}
```

Datos no estructurados

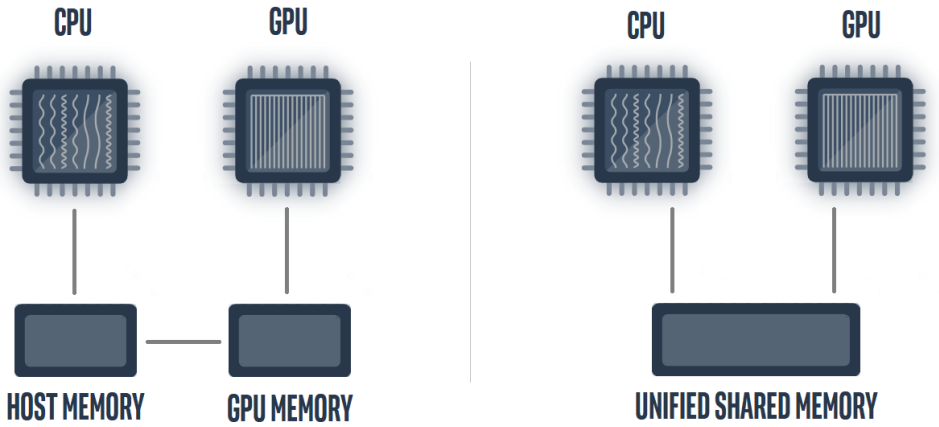
- Clausulas del `enter data`
  - `map(alloc:var-list)`: reserva espacio en el dispositivo
  - `map(to:var-list)`: reserva espacio y copia desde el host al dispositivo
- Clausulas del `exit data`
  - `map(delete:var-list)`: libera espacio en el dispositivo
  - `map(from:var-list)`: libera espacio previa copia de datos desde el dispositivo

Declaración `target`

- Hace una variable residente en el acelerador
- Se debe indicar en la declaración de la variable
- El tiempo de vida en el acelerador es implicito al uso de la variable
  - C/C++: `#pragma omp declare target [clauses]`

Memoria unificada

- Añadida en el estandard en la versión OpenMP 5.0
  - Se asume que la memoria es accesible desde el host y device



Memoria unificada

Type	Location	Accessible From	Allocation Routine
Host	Host	Host or Device	omp_target_alloc_host(size,device_num)
Device	Device	Device	omp_target_alloc_device(size,device_num)
Shared	Host or Device	Host or Device	omp_target_alloc_shared(size,device_num)

Memoria unificada

usm.c

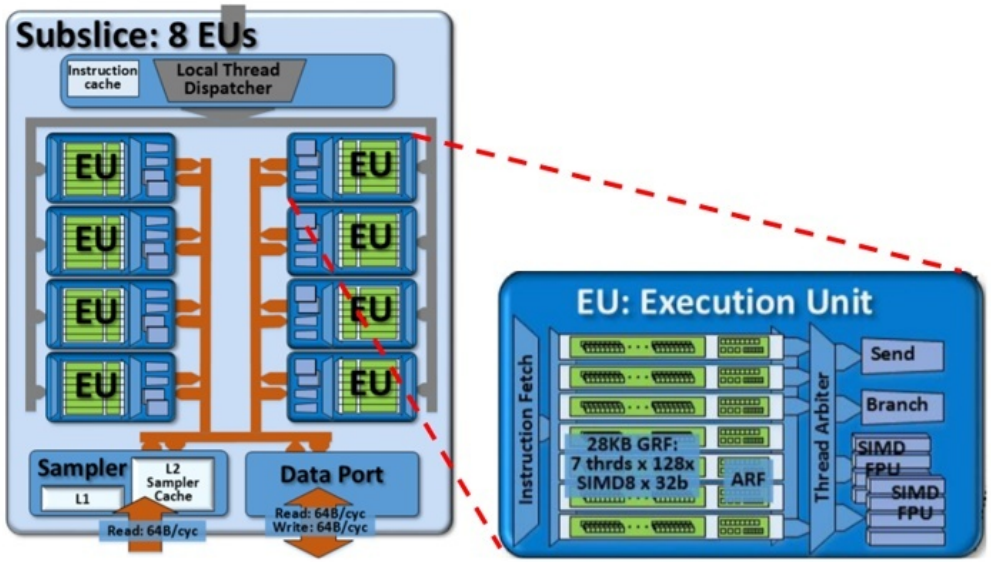
```
#include <omp.h>
#define SIZE 1024

#pragma omp requires unified_shared_memory
int main() {
  int deviceId = (omp_get_num_devices() > 0) ? omp_get_default_device()
       : omp_get_initial_device();

  int *a = (int *)omp_target_alloc_shared(SIZE, deviceId);
  int *b = (int *)omp_target_alloc_shared(SIZE, deviceId);

  for (int i = 0; i < SIZE; i++) {
    a[i] = i; b[i] = SIZE - i;
  }

  #pragma omp target parallel for
  for (int i = 0; i < SIZE; i++) {
    a[i] += b[i];
  }

  ...
  omp_target_free(a, deviceId);
  omp_target_free(b, deviceId);
...
}
```

Arquitectura GPU

- GPU tiene varios niveles de paralelismo a explotar

Recuerda en la Gen11 de Intel

- 1 Slice × 8 subslices × 8 EUs × 2 ALUs SIMD × SMT



Arquitectura GPU

- La construcción `omp target` descarga el kernel en el dispositivo...
- ... pero, sacamos el máximo partido al device?
  - solo un SMs es utilizado!!

¡¡Presta atención!!

¿Cómo se puede distribuir la carga de trabajo entre los Slice, Subslices, EUs, ALUs...?

Modelo de ejecución: teams

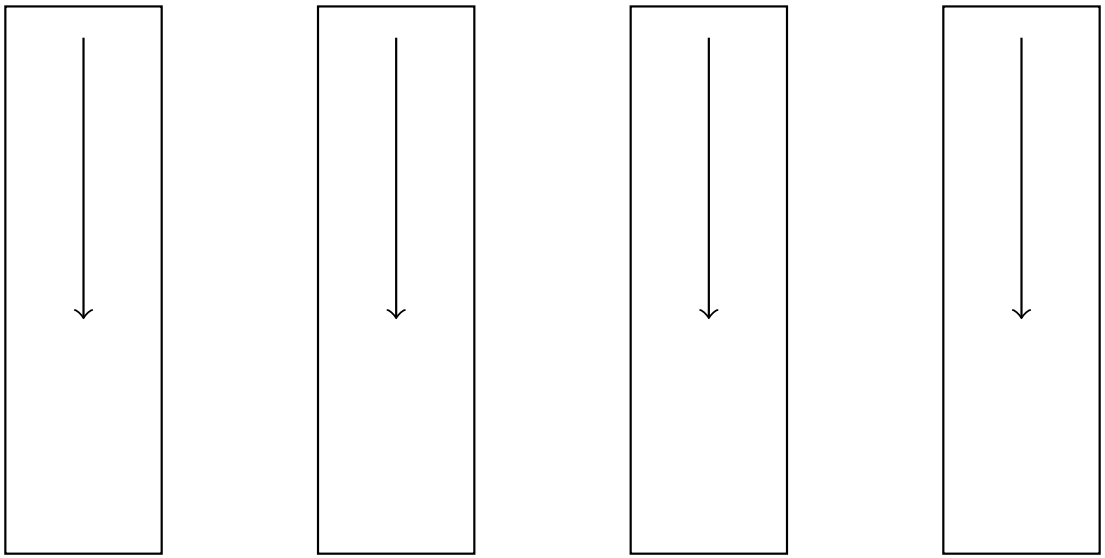
- Los **threads** de OpenMP en el device se agrupan en *teams*
- Los hilos de un team pueden sincronizarse
  - Pero **no pueden** sincronizarse entre hilos de otros teams
- Grupos de teams se denominan una *league*
- La construcción `target` descarga la ejecución al dispositivo pero de forma **secuencial**
  - La construcción `teams` crea una league de teams
  - El hilo master en *cada* team ejecuta el código

C syntax

```
#pragma omp target teams
```

Modelo de ejecución: teams

- La construcción `target teams` crea un número de teams en la GPU que contienen un solo hilo (*League and teams of threads*)
- Todos los hilos ejecutan el mismo bloque de código



Modelo de ejecución: distribute

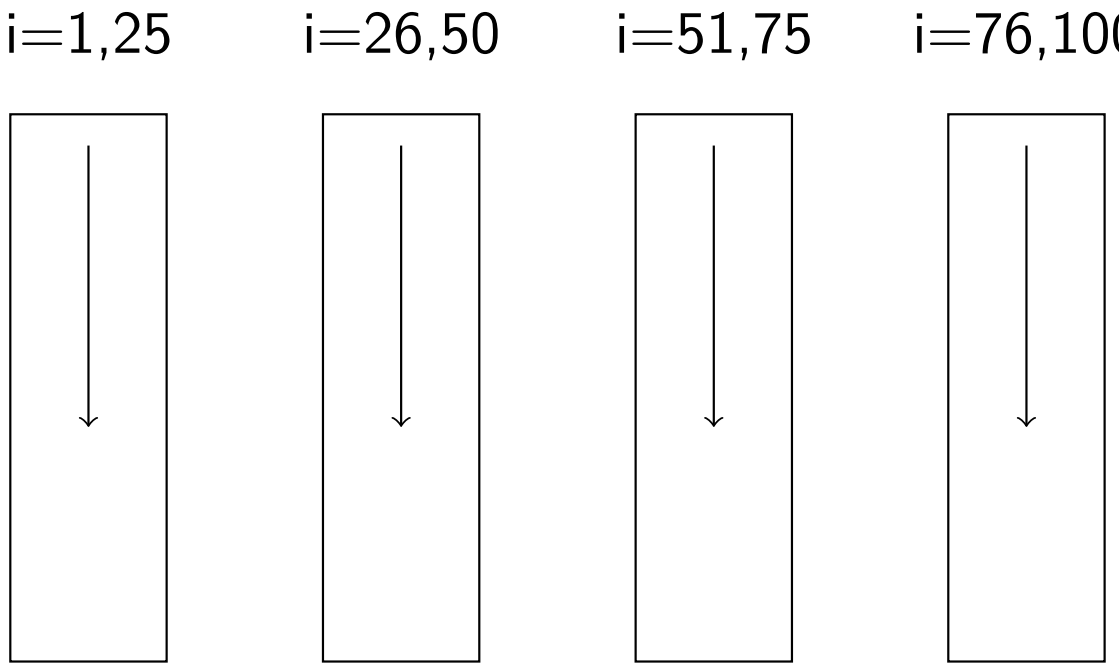
- Las iteraciones del bucle se distribuyen entre los teams
- Cada teams tiene un conjunto de iteraciones a procesar
- La asignación normalmente se realiza *estáticamente*, aunque la clausula `dist_schedule` permite controlar la distribución de iteraciones de forma parecida a la construcción `#pragma omp parallel for schedule (...)`
- Sin embargo, el hilo maestro de un team sigue ejecutando el código él solo, aún no hay más hilos que el maestro en el team

omp_distributed.c

```
#pragma omp target teams distribute
for(int i=1; i<N; i++)
{
   ...
}
```

Modelo de ejecución: distribute

- La construcción `target teams distribute` distribuye las iteracione del bucle entre los teams
- Recuerda: cada teams solo tiene un único hilo (master)
- Cada team computa un rango distinto de iteraciones



Modelo de ej.: parallel for

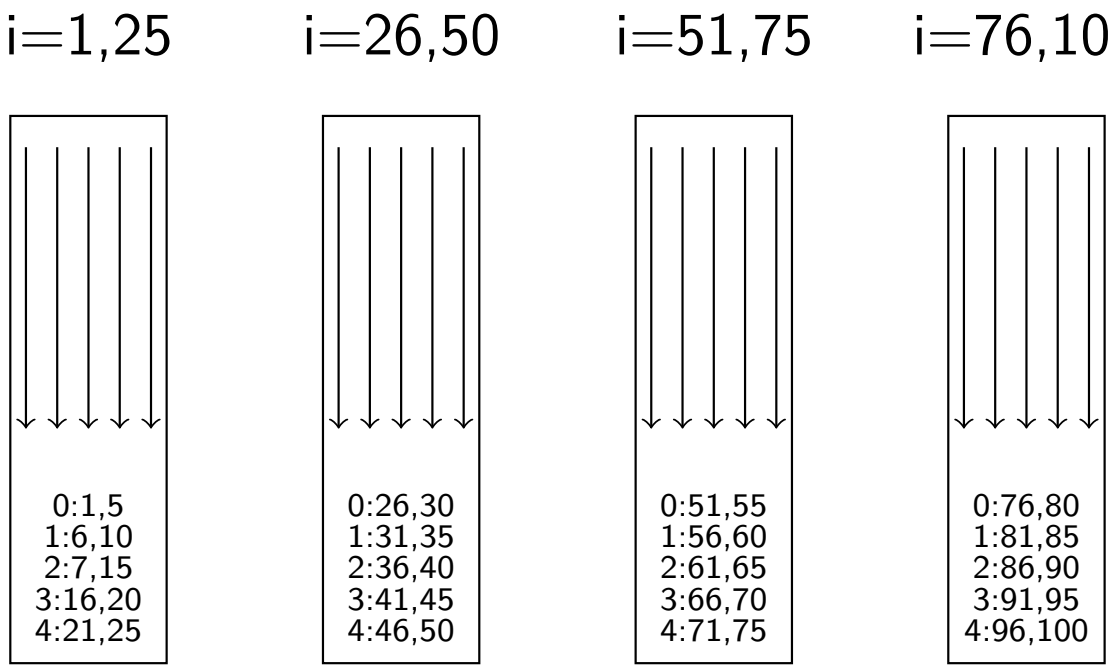
- Misma semántica que para CPU!
- Los hilos son creados (fork-join) y asociado a un team, y se distribuyen las iteraciones asignadas al team entre todos estos nuevos hilos
- NOTA: las iteraciones asignadas a cada team con la construcción `distribute` son ahora distribuidas entre los hilos
- También puede usarse la clausula `dist_schedule` en este contexto para especificar la política de distribución de iteraciones

omp_distributed_parallelfor.c

```
#pragma omp target teams distribute parallel for
for(int i=1; i<N; i++)
{
   ...
}
```

Modelo ejecución: parallel do

- La construcción `target teams distribute parallel for` lanza hilos en cada team
- Las iteraciones asignadas al team son distribuidas entre los hilos creados, controlados con `dist_schedule`



Construcción SIMD

- La construcción `simd` es también válida con la construcción `parallel for`
- Instrucciones vectoriales/SIMD también son generadas

Modelo de ej. en Intel-GPUs

- `target`: descarga el kernel en la GPU
- `teams`: un hilo por sub-slice
- `distributed`: distribuye "trozos" (1024 en el ejemplo) de iteraciones por team
- `parallel for`: crea hilos dentro del sub-slice
- `simd`: explota las capacidades vectoriales a nivel de Execute-Unit

omp_intel.c

```
#pragma omp target map(to:x[0:sz]) map(tofrom(y[0:sz])
{
  #pragma omp teams num_teams(16)
  {
      #pragma omp distribute dist_schedule(static, 1024)
      for (ib = 0; ib < sz; ib += num_blocks) {
         #pragma omp parallel for simd schedule(static,64)
         for (int i = ib; i < ib + num_blocks; i++) {
            y[i] = a * x[i] + y[i];
         }
      }
  }
}
```

Modelo de ej. en Intel-GPUs

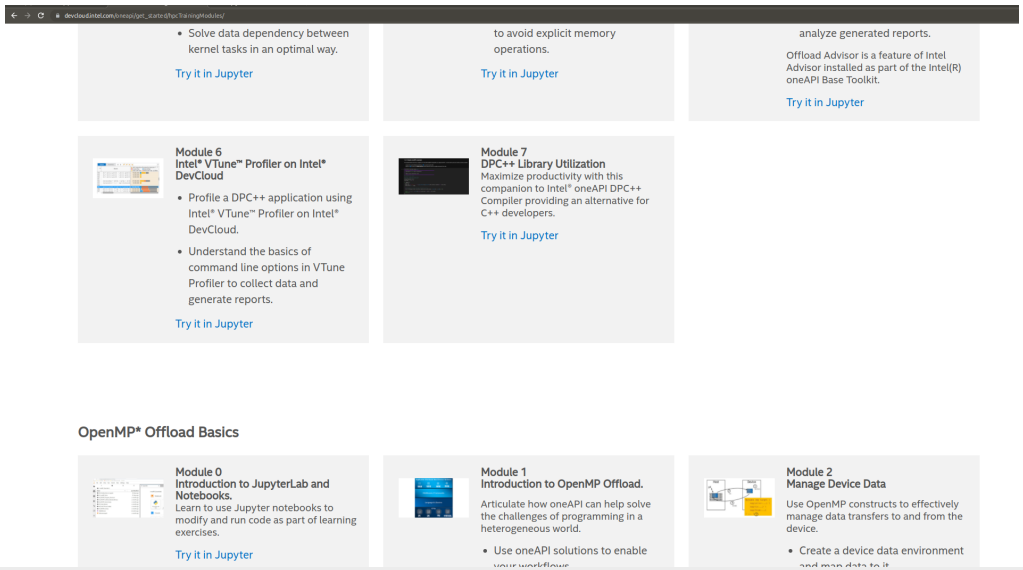
- `target`: descarga el kernel en la GPU
- `teams`: un hilo por sub-slice
- `distributed`: distribuye "trozos" de iteraciones por team
- `parallel for`: crea hilos dentro del sub-slice
- `simd`: explota las capacidades vectoriales a nivel de Execute-Unit

omp_intel.c

```
void saxpy(float a, float* x, float* y, int sz) {
#pragma omp target teams distributed parallel for simd \
   num_teams(num_blocks) map(to:x[0:sz]) map(tofrom(y[0:sz])
{
   for (int i = 0; i < sz; i++) {
      y[i] = a * x[i] + y[i];
   }
}
```

Hand-on

1. Conéctate al Intel DevCloud
2. ... pero vamos a trabajar con **OpenMP Offload Basics**
   - https://devcloud.intel.com/oneapi/get_started/hpcTrainingModules/
3. Selecciona el cuaderno de jupyter **Module 3 OpenMP* Device Parallelism**



Funciones interesantes de la API

- `omp_is_initial_device()` : devuelve True/False (cuando estás en el host o dispositivo)
- `omp_get_num_devices()` : número de dispositivos disponibles
- `omp_get_device_num()` : número del dispositivo desde donde se invoca
- `omp_get_default_device` : dispositivo por defecto
- `omp_set_default_device` : fija dispositivo por defecto

reduction

- La construcción `parallel for` soporta clausulas de reducción[3]

omp.c

```
#ifdef TARGET_GPU
   #pragma omp target teams distribute reduction(max:error)
#endif
for( int j = 1; j < n-1; j++) {
   #pragma omp parallel for reduction(max:error)
  for( int i = 1; i < m-1; i++ ) {
      Anew[j][i] = 0.25f * ( A[j][i+1] + A[j][i-1]
                           + A[j-1][i] + A[j+1][i]);
      error = fmaxf( error, fabsf(Anew[j][i]-A[j][i]));
  }
}
```

[3]

declare target

- Cuando una función es invocada desde el código del device

omp_declare_target.c

```
#pragma omp declare target
extern void fib(int N);
#pragma omp end declare target

#define THRESHOLD 1000000
void fib_wrapper(int n)
{
    #pragma omp target if(n > THRESHOLD)
    {
    fib(n);
    }
}
```

Más paralelismo

- Para bucles anidados, la clausula `collapse(N)` fusiona los N loops en uno solo
- Este aspecto permite disponer de más paralelismo para poder ser distribuido

omp_collapse.c

```
#pragma omp target data map(fromto:A[0:N*M])
{
    #pragma omp target
    #pragma omp parallel for collapse(2)
    for(i=0;i<N; i++){
        for(j=0;j<M; j++){
            foo(A,i,j);
        }
    }
}
```

Llamadas a funciones desde región `target`

- A menudo, es útil llamar funciones para mejorar la legibilidad y código modular
  - Por defecto, OpenMP no crea regiones aceleradas que contengan llamadas a funciones
  - Indicar al compilador que compile una versión para device de la función
- `#pragma omp declare target` y `#pragma omp end declare target`

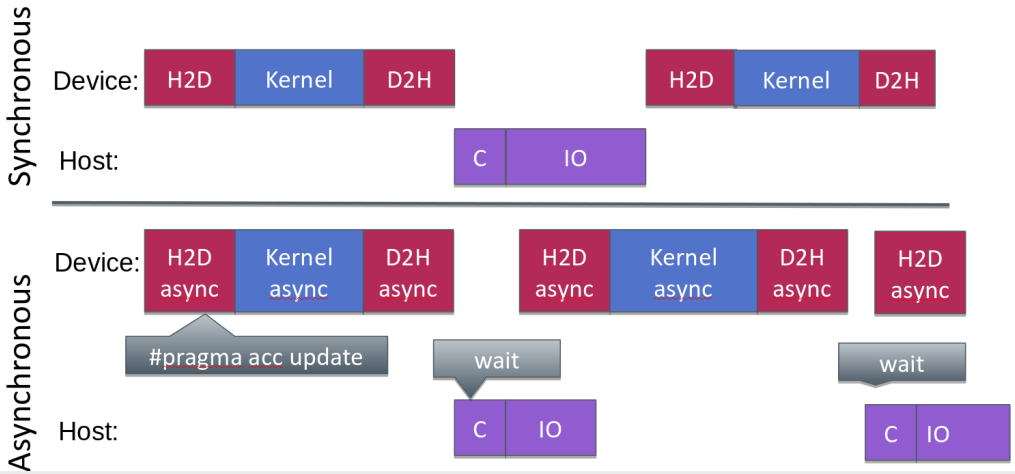
Llamadas a funciones

omp_declare_target.c

```
#pragma omp declare target
void foo(float* v, int i, int n) {
...
}
#pragma omp end declare target

void main()
{
   ...
   #pragma omp target teams parallel for
   for (int i=0; i<n; ++i) {
      foo(v,i); // executed on the device
   }
   ...
}
```

Modelo ejecución task

- Por defecto el modelo de ejecución *target* es síncrono: (host espera finalización del kernel)
  - ... aunque implicitamente la construcción *target* conlleve una tarea
- Además OpenMP soporta el modelo **task** asíncrono
  - Directivas del tipo `task`, `taskloop` no llevan barrera implícita y que se han de sincronizar con `taskwait`



Modelo ejecución task

- Ej: de modelo asíncrono con *target*

omp_task_target.c

```
#pragma omp target nowait
{process_in_device();}

process_in_host();
#pragma omp taskwait
```

Dependencias con tareas

- El modelo *task* conlleva un flujo de tareas con sus dependencias (input/output)

omp_task_depend.c

```
#pragma omp task depend(out: A)
{Code A}

#pragma omp target depend(in: A) depend(out: B) nowait
{Code B}

#pragma omp target depend(in: B) depend(out: C) nowait
{Code C}

#pragma omp target depend(in: B) depend(out: D) nowait
{Code D}

#pragma omp task depend(in: A) depend(in: A)
{Code E}

#pragma omp task depend(in: C,D,E)
{Code F}
```

Dependencias con tareas

- El modelo *task* conlleva un flujo de tareas con sus dependencias (input/output)
  - Se puede controlar la granilaridad más fina

omp_task_depend.c

```
// Preocessing array in blocks
for (int ib = 0; ib < n; ib += bf) {
  #pragma omp ... depend(out:A[ib*bf]) nowait
  {Processing step 1}
  #pragma omp ... depend(in:A[ib*bf]) nowait
  {Processing step 2}
}
```

Interoperabilidad

- OpenMP *target* o *offloading* tambien soporta interación de forma nativa con lenguajes de más bajo nivel como CUDA o HIP, o incluso MPI
- Las construcciones:
  - `omp target data use_device_ptr(var-list)` : define un puntero disponible en el host
  - `omp target data use_device_addr(var-list)` : permite que las variables en *var-list* sean accesibles en el dispositivo

Interoperabilidad

- Ejemplo de interacción con la librería nativa de cuBLAS para GPUs de NVIDIA:

omp_interop_cublas.c

```
cublasInit();
double *x, *y;
//Allocate x and y, and initialise x
#pragma omp target data map(to:x[:n]), map(from:y[:n]))
{
    #pragma omp target data use_device_ptr(x, y) {
        cublasDaxpy(n, a, x, 1, y, 1);
    }
}
```

Compilador (NVIDIA)

- Soportado por el NVIDIA HPC
- Reportes controlados por el flag de compilación `-Minfo[=option]`
- Otras opciones interesantes:
  - `mp` – activación de OpenMP
  - `all` – imprime todos los mensajes del compilador
  - `intensity` – muestra información relacionada con la intensidad computacional

Compilador (NVIDIA)

- Ejemplo de opción: `-Minfo`

Terminal #1

```
user@lab:~$ nvc++ -O3 -mp=gpu -gpu=cc80 -c -Minfo=mp,intensity core.cpp
evolve:
     63, #omp target teams distribute parallel for
         63, Generating Tesla and Multicore code
             Generating "nvkernel_evolve_F1L63_1" GPU kernel
         68, Loop parallelized across teams and threads, schedule(static)
     69, Intensity = 19.00
```

Compilador (ICX)

- icx/icpx para C/C++ incluido en la suite oneAPI Toolkit HPC
- Soporte OpenMP:
  - `-fiopenmp`: compila y reconoce las directivas OpenMP (tanto multithreading como SIMD)
  - `-fopenmp-targets=spir64`: necesaria para activar el soporte de OpenMP v4.5/5.0 con directivas *target*

Terminal #1

```
user@lab:~$ icx -fiopenmp -fopenmp-targets=spir64
user@lab:~$ icpx -fiopenmp -fopenmp-targets=spir64
```

Compiladores de Intel C++

- Compatibles los binario y linkables https://www.intel.com/content/www/us/en/developer/articles/tool/oneapi-standalone-components.html

Compilador	target	OpenMP	OpenMP-offload	Toolkit
Intel C++ Compiler ILO **icc/icpc/icl**	CPU	SI	No	HPC
Intel® oneAPI DPC++ Compiler, **dpcpp**	CPU, GPU, FPGA	SI	SI	Base
Intel® oneAPI C++ Compiler, **icx/icpx**	CPU, GPU	SI	SI	Base

Resumen de las versiones OpenMP

- OpenMP 4.0
  - target [data]
  - declare target
  - simd
  - teams
  - distributed parallel for
  - target teams
  - ... otras llamadas a la API
- OpenMP 5.0
  - taskloop
  - taskloop simd
  - target enter/exit data
  - ... otras llamadas a la API
- OpenMP 5.1
  - allocate
  - declare mapper
  - unified memory
  - parallel loop/teams loop
  - ... otras llamadas a la API

Más info

- OpenMP: www.openmp.org
- Reference Guides: https://www.openmp.org/resources/refguides/
- Tutorials:
  - SC 2020, 2019... https://www.openmp.org/resources/openmp-presentations/
- Compilers supported https://www.openmp.org/resources/openmp-compilers-tools/