CAGE – Grupo de Investigación

@sc3uis

SC3 – Super Cómputo y Cálculo Científico – UIS

Programación de GPU AMD con ROCm y HIP

ROCm, HIP y portabilidad entre GPU

Módulo 7 – Semillero CAGE

Luis Alejandro Torres Niño | sc3.operaciones@uis.edu.co

Objetivos del módulo

- Conocer la plataforma **ROCm** y la arquitectura de las GPU AMD (*compute units*, *wavefronts*, LDS).
- Escribir y compilar programas con **HIP**: *kernels*, memoria, *events* y manejo de errores.
- **Portar** código CUDA a HIP (módulo 6) y entender qué cambia y qué no.
- Explorar la portabilidad: HIP, OpenMP *offload* y SYCL (AdaptiveCpp).
- Usar bibliotecas (rocBLAS, rocThrust) y herramientas (`rocminfo`, `rocm-smi`, `rocprof`).
- Ejecutar en la partición `amd` del cluster y combinar **MPI + HIP** con OpenMPI (*ROCm-aware*: por confirmar en el cluster).

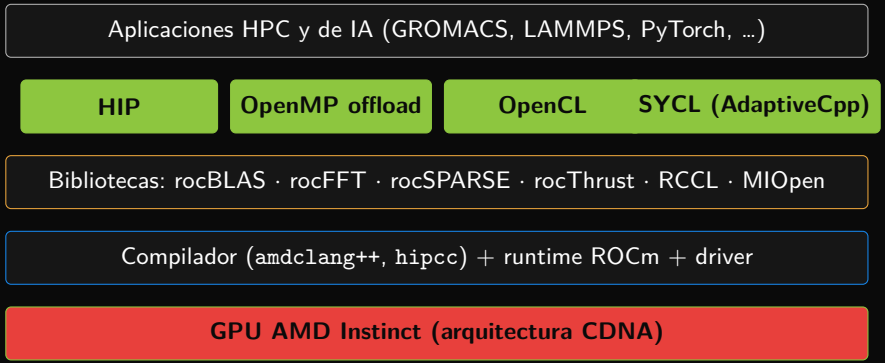
Agenda

1. ROCm, HIP, arquitectura y entorno en el cluster
2. La API de HIP y de CUDA a HIP
3. Kernels y grid: suma de vectores
4. Memoria: LDS, *reduction* y matmul
5. Portabilidad: hipify, OpenMP offload y SYCL
6. Bibliotecas y herramientas
7. MPI + HIP en el cluster
8. Retos individuales y reto grupal

1

ROCm, HIP y la arquitectura de las GPU AMD

ROCm: la plataforma abierta de AMD

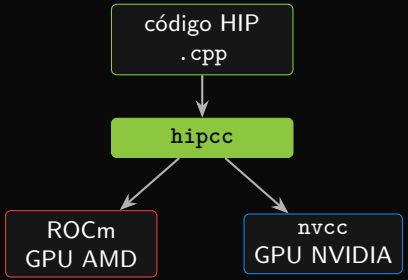


➔ **ROCm** es el stack de software **abierto** de AMD para cómputo en GPU: varios lenguajes (HIP, OpenMP, OpenCL, Python) y bibliotecas matemáticas y de comunicación.

Fuente: amd.com/en/products/software/rocm

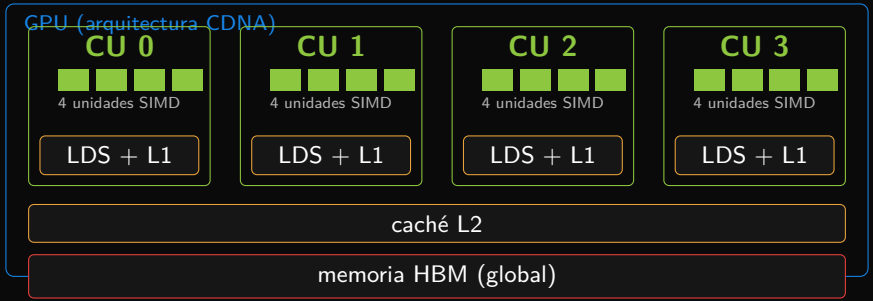
HIP: Heterogeneous-compute Interface for Portability

- API de C++ y *kernel language* para escribir aplicaciones que corren en **GPU AMD** y en **GPU NVIDIA**.
- **Open source**.
- Sintácticamente **casi idéntica a CUDA**: la mayoría de llamadas se convierten cambiando solo el prefijo: `cuda` → `hip`.
- Soporta un subconjunto amplio del runtime de CUDA.



Fuente: *Intro to AMD GPU Programming with HIP* (material AMD).

La GPU AMD por dentro



- Una GPU AMD se compone de varias **CU** (*compute units*); cada CU tiene unidades SIMD, registros y **LDS** (*Local Data Share*).
- Un *work-group* (bloque) se ejecuta **completo en una CU**.
- Modelo, número de CU, memoria y arquitectura (`gfx90a`, ...) dependen de la GPU: consúltalos con `rocminfo`.

Diccionario CUDA ↔ HIP ↔ hardware AMD

CUDA (NVIDIA)	HIP (código)	Hardware AMD
`threadIdx`, `blockIdx`	igual	work-item, work-group
SM	–	CU (*compute unit*)
warp (32 hilos)	`warpSize`	**wavefront** (64 en CDNA)
`__shared__`	igual	**LDS**
memoria global (VRAM)	igual	HBM
`cudaMalloc`	`hipMalloc`	–
`nvcc`	`hipcc`	`amdclang++`

Wavefront: warp de 64 (y a veces 32)



- En las GPU de cómputo AMD Instinct (CDNA) el *wavefront* es de **64 hilos**; algunas GPU de la serie RDNA usan 32 (o 64).
- **No asumas 32**: pregunta `warpSize` (o `props.warpSize`) en tiempo de ejecución.
- Todo lo dicho sobre *divergence* y *coalescing* (módulo 6) **aplica igual**: los 64 hilos del wavefront ejecutan la misma instrucción.
- Elige bloques **múltiplos de 64** y, en general, de 128 a 256 hilos.

El entorno de ROCm en el cluster

```
ssh usuario@toctoc.sc3.uis.edu.co          # bastion
ssh guane                                   # frontend (sin GPU)

sinfo -p amd                                # nodos con GPU AMD (exadell, smexa)
module avail rocm                           # rocm/system, ...
module load rocm/system
hipcc --version

srun -p amd --gres=gpu:1 --pty bash         # shell en un nodo con 1 GPU
rocminfo | grep gfx                         # arquitectura: gfx90a, gfx942, ...
rocm-smi                                    # estado de las GPU
```

- El **frontend no tiene GPU**: ejecuta en un nodo de la partición `amd`.
- Confirma el nombre del recurso con `scontrol show node exadell` (puede no ser `gpu`).

Compilar con hipcc

```
hipcc -O2 hola.cpp -o hola                                  # arquitectura por
    defecto
hipcc -O2 --offload-arch=gfx90a hola.cpp -o hola            # para una GPU concreta
```

- ➔ Los archivos son `.cpp` normales: no hay extensión `.hip` obligatoria.
- ➔ `hipcc` es un envoltorio de clang: separa host y device como `nvcc`.
- ➔ `--offload-arch=gfxXXXX` fija el código de device para la GPU.

★ Arquitectura (gfx)

Mira el valor con `rocminfo | grep gfx`. `gfx90a` es el **ejemplo** usado en las diapositivas de AMD: usa el que corresponda a tu nodo.

Hola mundo en HIP

```cpp
#include <hip/hip_runtime.h>
#include <cstdio>

__global__ void hola() {
    printf("Hola desde el bloque %d, hilo %d\n", (int)blockIdx.x, (int)threadIdx.
        x);
}

int main() {
    hola<<<2, 4>>>();               // misma sintaxis que CUDA
    (void)hipDeviceSynchronize();  // espera al kernel
    return 0;
}
```

```
hipcc -o hola hola.cpp
srun -p amd --gres=gpu:1 ./hola
```

Cambia a `<<<1,8>>>`, `<<<8,1>>>` y `<<<2,64>>>`: observa que el orden de impresión **no está garantizado**.

Reto individual 1: conoce tu GPU AMD

■ Reto 1 – Hola GPU AMD

1. Compila y ejecuta `hola.cpp` con `hipcc` en un nodo de la partición `amd`.
2. Ejecuta `rocminfo` y `rocm-smi`: anota modelo, arquitectura (`gfx`), memoria y número de CU.
3. Escribe un programa que use `hipGetDeviceProperties` e imprima nombre, arquitectura, CU, *wavefront size*, memoria global, LDS por bloque e hilos máximos por bloque.
4. Compara con la GPU NVIDIA del módulo 6: ¿qué cambia y qué se mantiene?

2
La API de HIP y de CUDA a HIP

La API de HIP

Área	Funciones
Dispositivos	`hipSetDevice`, `hipGetDevice`, `hipGetDeviceProperties`
Memoria	`hipMalloc`, `hipMemcpy`, `hipMemcpyAsync`, `hipFree`
Streams	`hipStreamCreate`, `hipStreamSynchronize`, `hipDeviceSynchronize`, `hipStreamDestroy`
Eventos	`hipEventCreate`, `hipEventRecord`, `hipStreamWaitEvent`, `hipEventElapsedTime`
Kernels	`__global__`, `__device__`, `hipLaunchKernelGGL`
Código de device	`threadIdx`, `blockIdx`, `blockDim`, `__shared__`
Errores	`hipGetLastError`, `hipGetErrorString`

Más de 200 funciones matemáticas de device cubren la biblioteca matemática de CUDA.

Fuente: material *Introduction to AMD Programming with ROCm – HIP*.

De CUDA a HIP: la conversión

CUDA

```
#include <cuda_runtime.h>

cudaMalloc(&d_a, bytes);
cudaMemcpy(d_a, h_a, bytes,
           cudaMemcpyHostToDevice);
suma<<<bloques, hilos>>>(d_a, d_b, d_c, n);
cudaDeviceSynchronize();
cudaFree(d_a);

cudaError_t e = cudaGetLastError();
cudaEvent_t t0;  cudaEventCreate(&t0);
```

HIP

```
#include <hip/hip_runtime.h>

hipMalloc(&d_a, bytes);
hipMemcpy(d_a, h_a, bytes,
          hipMemcpyHostToDevice);
suma<<<bloques, hilos>>>(d_a, d_b, d_c, n);
hipDeviceSynchronize();
hipFree(d_a);

hipError_t e = hipGetLastError();
hipEvent_t t0;  hipEventCreate(&t0);
```

- El **código del kernel no cambia**; cambian el `#include` y el prefijo `cuda` → `hip`.
- La herramienta **HIPIFY** (`hipify-perl`, `hipify-clang`) hace la conversión automáticamente.

Lanzar kernels: dos formas

```
// Forma 1: la misma de CUDA (hipcc la acepta)
suma<<<bloques, hilos>>>(d_a, d_b, d_c, n);

// Forma 2: macro de HIP
//                          kernel  grid     bloque   shmem  stream  argumentos...
hipLaunchKernelGGL(suma, dim3(bloques), dim3(hilos), 0, 0, d_a, d_b, d_c, n);

// con shared memory (LDS) dinamica y un stream:
suma<<<bloques, hilos, bytes_lds, stream>>>(d_a, d_b, d_c, n);
```

- ➔ `hipLaunchKernelGGL` es la forma **canónica** de HIP; `<<<>>>` se acepta por compatibilidad con CUDA.
- ➔ Los kernels son `void`; los resultados salen por punteros.

Manejo de errores: HIP_CHECK

```
#define HIP_CHECK(llamada)                                                  \
    do { hipError_t e = (llamada);                                          \
         if (e != hipSuccess) {                                             \
             std::fprintf(stderr, "HIP error %s en %s:%d\n",                \
                          hipGetErrorString(e), __FILE__, __LINE__);        \
             std::exit(1); } } while (0)

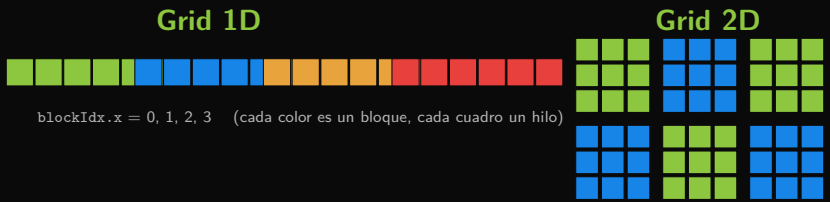
HIP_CHECK(hipMalloc(&d_a, bytes));
suma<<<bloques, hilos>>>(d_a, d_b, d_c, n);
HIP_CHECK(hipGetLastError());          // error de lanzamiento
HIP_CHECK(hipDeviceSynchronize());     // error de ejecucion
```

❄ El compilador te avisa

Con `-Wall`, `hipcc` marca `warning: ignoring return value … nodiscard`: las funciones HIP devuelven un `hipError_t` que **no debes ignorar**. Usa siempre `HIP_CHECK`.

3
Kernels y grid: suma de vectores

El grid en 1D y en 2D



`blockIdx.x`, `blockIdx.y`	bloque dentro del grid (1D o 2D).
`threadIdx.x`, `threadIdx.y`	hilo dentro de su bloque.
`blockDim.x`, `gridDim.x`	hilos por bloque / bloques en el grid.

El concepto es el **mismo** en 1D y 2D: en 2D cada bloque y cada hilo tienen un índice de dos dimensiones.

Kernels

- Una función de device que se lanza desde el host es un **kernel**: se declara con `__global__` y debe devolver `void`.
- **Todos los hilos** ejecutan el cuerpo del kernel "simultáneamente" (SIMT).
- Cada hilo usa su **ID de bloque e hilo** para calcular su **índice global**.
- Puede haber **más hilos** que N en el grid: protege con `if (i < n)`.

```
__global__ void suma(const float* a, const float* b, float* c, int n) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;      // indice global
    if (i < n) c[i] = a[i] + b[i];
}
```

Exactamente el mismo kernel del módulo 6: en HIP **no cambia una línea**.

Suma de vectores en HIP (1/2)

```
const int n = 1 << 24;  const size_t bytes = n * sizeof(float);
float *h_a = (float*)malloc(bytes), *h_b = (float*)malloc(bytes), *h_c = (float*)
    malloc(bytes);
for (int i = 0; i < n; ++i) { h_a[i] = 1.0f; h_b[i] = 2.0f; }

float *d_a, *d_b, *d_c;
HIP_CHECK(hipMalloc(&d_a, bytes));                    // 1) reservar en la GPU
HIP_CHECK(hipMalloc(&d_b, bytes));
HIP_CHECK(hipMalloc(&d_c, bytes));
HIP_CHECK(hipMemcpy(d_a, h_a, bytes, hipMemcpyHostToDevice));    // 2) host ->
    device
HIP_CHECK(hipMemcpy(d_b, h_b, bytes, hipMemcpyHostToDevice));
```

➔ Los mismos **seis pasos** del módulo 6: reservar, copiar a la GPU, lanzar, sincronizar, copiar de vuelta, liberar.

Suma de vectores en HIP (2/2)

```
int hilos = 256, bloques = (n + hilos - 1) / hilos;          // division techo
hipEvent_t t0, t1;
HIP_CHECK(hipEventCreate(&t0));  HIP_CHECK(hipEventCreate(&t1));
HIP_CHECK(hipEventRecord(t0));
suma<<<bloques, hilos>>>(d_a, d_b, d_c, n);                  // 3) kernel
HIP_CHECK(hipEventRecord(t1));
HIP_CHECK(hipGetLastError());
HIP_CHECK(hipEventSynchronize(t1));                          // 4) sincronizar
float ms = 0;  HIP_CHECK(hipEventElapsedTime(&ms, t0, t1));

HIP_CHECK(hipMemcpy(h_c, d_c, bytes, hipMemcpyDeviceToHost)); // 5) device ->
    host
// verificar en la CPU
HIP_CHECK(hipFree(d_a)); HIP_CHECK(hipFree(d_b)); HIP_CHECK(hipFree(d_c));   //
    6) liberar
```

Mide con `hipEvent` el tiempo del kernel y compáralo con el tiempo total (con copias): el **costo de transferir** sigue dominando en operaciones simples.

Reto individual 2: portar a mano

■ Reto 2 – De CUDA a HIP

1. Toma tu `vecadd.cu` del módulo 6 y conviértelo a HIP **a mano** (cambia el `#include` y los prefijos).
2. Compílalo con `hipcc -Wall` y corrige los avisos de `nodiscard` con `HIP_CHECK`.
3. Ejecútalo en la partición `amd` y verifica el resultado contra la CPU.
4. Mide el kernel con `hipEvent` y compáralo con el de la GPU NVIDIA (módulo 6): ¿qué ancho de banda efectivo logras en cada una?

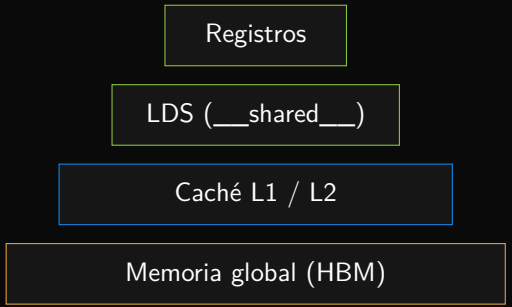
Reto individual 3: índices 2D

■ Reto 3 – Kernel 2D

1. Escribe un kernel **2D** con `dim3 bloque(16,16)` que llene una matriz $A[i][j] = i \cdot N + j$.
2. Usa `hipLaunchKernelGGL` en vez de `<<<>>>` y comprueba que el resultado es igual.
3. Verifica en la CPU para $N = 1000$ (no múltiplo de 16): ¿qué pasa si quitas la guarda?
4. Cambia el bloque a 8×8, 32×8 y 64×4: ¿cambia el tiempo? ¿Qué relación tiene con el *wavefront*?

4
Memoria: LDS, *reduction* y matmul

La jerarquía de memoria



	Alcance	Velocidad
Registros	un work-item	más rápida
LDS (__shared__)	un work-group	muy rápida (en la CU)
Caché L1 / L2	automática	rápida
Memoria global (HBM)	todos + host	lenta, mucho ancho de banda

Misma idea que en NVIDIA: **coalescing** en la memoria global (los hilos consecutivos acceden a direcciones consecutivas) y reutilización de datos en LDS. La *shared memory* (LDS) se declara igual: `__shared__`.

Reduction con LDS

```
__global__ void reduce_suma(const float* in, float* out, int n) {
    __shared__ float tile[256];                    // LDS del bloque
    int tid = threadIdx.x;
    int i = blockIdx.x * blockDim.x + tid;
    tile[tid] = (i < n) ? in[i] : 0.0f;
    __syncthreads();

    for (int s = blockDim.x / 2; s > 0; s >>= 1) {  // reduction en arbol
        if (tid < s) tile[tid] += tile[tid + s];
        __syncthreads();
    }
    if (tid == 0) atomicAdd(out, tile[0]);          // suma parcial del bloque
}
```

Código **idéntico** al de CUDA. `atomicAdd` sobre `float` está disponible en HIP; su rendimiento depende de la GPU y la arquitectura. Con bloques de 256 hay 4 *wavefronts* de 64 por bloque.

matmul con tiles en HIP

```
#define TILE 16
__global__ void matmul_tiled(const float* A, const float* B, float* C, int n) {
    __shared__ float As[TILE][TILE], Bs[TILE][TILE];
    int fila = blockIdx.y * TILE + threadIdx.y;
    int col  = blockIdx.x * TILE + threadIdx.x;
    float s = 0.0f;
    for (int t = 0; t < (n + TILE - 1) / TILE; ++t) {
        int a_col = t * TILE + threadIdx.x, b_fila = t * TILE + threadIdx.y;
        As[threadIdx.y][threadIdx.x] = (fila < n && a_col < n) ? A[fila*n + a_col] : 0.0f;
        Bs[threadIdx.y][threadIdx.x] = (b_fila < n && col < n) ? B[b_fila*n + col] : 0.0f;
        __syncthreads();                          // tile completo
        for (int k = 0; k < TILE; ++k) s += As[threadIdx.y][k] * Bs[k][threadIdx.x];
        __syncthreads();                          // nadie sobreescribe antes
    }
    if (fila < n && col < n) C[fila*n + col] = s;
}
```

Lanzamiento: `dim3 bloque(TILE,TILE); dim3 grid((n+TILE-1)/TILE, (n+TILE-1)/TILE);`. Con $16 \times 16 = 256$ hilos por bloque; prueba `TILE` 8 y 32.

Memoria administrada (managed memory)

```
float* a;
hipMallocManaged(&a, n * sizeof(float));      // accesible desde CPU y GPU
for (int i = 0; i < n; ++i) a[i] = 1.0f;
kernel<<<bloques, hilos>>>(a, n);
hipDeviceSynchronize();                       // antes de leer en la CPU
hipFree(a);
```

- ➔ Equivalente a `cudaMallocManaged` del módulo 6: menos código, el driver migra datos bajo demanda.
- ➔ El soporte y el rendimiento **dependen de la GPU y de la configuración** del driver (por ejemplo, la migración de páginas puede requerir `XNACK`). Verifícalo en tu nodo antes de depender de él.
- ➔ Para rendimiento predecible: `hipMalloc` + `hipMemcpy` explícitos.

Reto individual 4: reducción y wavefront

Reto 4 – Suma de 2^{24} elementos en AMD

1. Porta tu reducción del módulo 6 a HIP; verifica contra la CPU.
2. Prueba tamaños de bloque 64, 128, 256, 512 y 1024 (ajusta el tamaño de `tile`). ¿Cuál es mejor? ¿Por qué conviene un múltiplo de 64?
3. Cambia el `atomicAdd` por sumas parciales por bloque y una segunda reducción. Compara el tiempo.
4. Compara con `std::accumulate`, la versión OpenMP (módulo 4) y la de CUDA (módulo 6).

Reto individual 5: matmul AMD vs NVIDIA

■ Reto 5 – De CUDA a HIP: matmul

1. Porta `matmul_naive` y `matmul_tiled` a HIP y verifícalos contra la CPU ($n = 512$).
2. Mide GFLOP/s ($2n^3/t$) para $n = 512, 1024, 2048$ con `TILE` 8, 16 y 32.
3. Compara los resultados con los de NVIDIA (módulo 6), con OpenMP (módulo 4) y con la CPU secuencial.
4. *Extra:* compara con `rocblas_sgemm`/`hipblasSgemm`: ¿qué fracción del rendimiento de la biblioteca alcanzas?

5

Portabilidad: hipify, OpenMP offload y SYCL

HIPIFY: convertir CUDA a HIP

```
# conversion rapida basada en reemplazos (perl)
hipify-perl vecadd.cu > vecadd_hip.cpp

# conversion basada en el compilador (clang) : mas precisa
hipify-clang vecadd.cu -o vecadd_hip.cpp

hipcc -O2 vecadd_hip.cpp -o vecadd_hip
```

- ➔ `hipify-perl`: rápido, sin dependencias, reemplaza nombres de la API.
- ➔ `hipify-clang`: analiza el código con clang y es más fiable en proyectos grandes.
- ➔ Siempre **revisa el resultado**: las funciones sin equivalente aparecen marcadas con comentarios.
- ➔ Compila en una máquina con ROCm y valida contra el resultado original.

Comprueba con `module avail` y `which hipify-perl` si las herramientas están instaladas en el cluster.

Un mismo código HIP en AMD y NVIDIA

```
# en un nodo AMD (ROCm)
module load rocm/system
hipcc --offload-arch=gfx90a prog.cpp -o
    prog

# en un nodo NVIDIA (CUDA): hipcc usa nvcc
module load cuda/11.8
HIP_PLATFORM=nvidia hipcc prog.cpp -o prog
```

★ Requisitos

Compilar para NVIDIA requiere una instalación de HIP con soporte NVIDIA y un compilador del host compatible con `nvcc` (módulo 6). Verifica en el cluster antes de planearlo.

- ➔ HIP **no es un traductor en tiempo de ejecución**: el código se compila para la plataforma elegida.
- ➔ Lo que no es portable: instrucciones específicas de una arquitectura, *warp size* fijo, *intrinsics* propietarios.

OpenMP offload: el mismo bucle del módulo 4

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

```
amdclang++ -O2 -fopenmp --offload-arch=gfx90a saxpy_omp.cpp -o saxpy_omp   # clang de ROCm
```

- El bucle se **descarga** a la GPU con directivas: sin escribir kernels.
- `map(to:…)`, `map(tofrom:…)` controlan las copias host ↔ device.
- Mismo código fuente para otras GPU si el compilador lo soporta.
- Menos control fino que HIP/CUDA, pero mucho menos esfuerzo de portado.

SYCL con AdaptiveCpp

```
#include <sycl/sycl.hpp>

int main() {
    const int n = 1 << 20;
    sycl::queue q;                                          // elige un
        dispositivo disponible
    float* a = sycl::malloc_shared<float>(n, q);
    float* b = sycl::malloc_shared<float>(n, q);
    for (int i = 0; i < n; ++i) { a[i] = 1.0f; b[i] = 2.0f; }

    q.parallel_for(sycl::range<1>(n), [=](sycl::id<1> i) {   // el "kernel" es
        una lambda
        b[i] = a[i] + b[i];
    }).wait();

    sycl::free(a, q);  sycl::free(b, q);
}
```

En el cluster: módulos `adaptivecpp/amd/25.02.0` y `adaptivecpp/nvidia/25.02.0`.

Compila con `acpp`; revisa `acpp --help` para elegir el objetivo (por ejemplo `--acpp-targets=hip:gfx90a`).

¿Qué modelo elegir?

Modelo	Ventajas	Cuidados
HIP	Control total; casi igual a CUDA; AMD y NVIDIA.	Hay que escribir y optimizar kernels.
CUDA	Ecosistema maduro en NVIDIA.	Solo NVIDIA.
OpenMP offload	Poco cambio sobre código CPU.	Menos control; depende del compilador.
SYCL (AdaptiveCpp)	C++ estándar; varios fabricantes.	Menos bibliotecas y herramientas propias.

★ Criterio

Código nuevo con rendimiento crítico en AMD: **HIP**. Ya tienes CUDA: **hipify**. Ya tienes OpenMP en CPU: prueba **offload**. Necesitas varios fabricantes con un solo C++: **SYCL**.

Reto individual 6: un mismo kernel, tres modelos

■ Reto 6 – Portabilidad

1. Convierte tu `saxpy.cu` del módulo 6 con `hipify-perl` (o a mano) y compílalo con `hipcc`.
2. Escribe la versión con `#pragma omp target` y la versión SYCL con AdaptiveCpp.
3. Mide los tres con $n = 2^{26}$ en la partición `amd`; calcula ancho de banda efectivo en GB/s.
4. Compara líneas de código, facilidad de escribir y rendimiento. ¿Cuál elegirías para un proyecto nuevo? Justifica.

6
Bibliotecas y herramientas

Bibliotecas de ROCm

- **rocBLAS** y **hipBLAS**: álgebra lineal densa (`rocblas_sgemm`, `hipblasSgemm`).
- **rocFFT**, **rocSPARSE**, **rocRAND**.
- **rocThrust** y **hipCUB**: algoritmos tipo STL en GPU.
- **RCCL**: comunicación colectiva entre GPU (equivalente a NCCL).
- **MIOpen**: redes neuronales.

★ ro* vs hip*

Las bibliotecas `roc*` son las implementaciones nativas de AMD; las `hip*` son capas portables que usan `roc*` en AMD y `cu*` en NVIDIA.

Usa una biblioteca siempre que exista: está optimizada para cada arquitectura. Escribe tu kernel para aprender o cuando la operación no esté cubierta.

Herramientas de diagnóstico

- ➔ `rocminfo`: agentes (CPU/GPU), arquitectura `gfx`, unidades de cómputo.
- ➔ `rocm-smi`: estado de las GPU (uso, memoria, temperatura, potencia).
- ➔ `rocprof` (o `rocprofv3`, según la versión): perfilado de kernels y trazas.
- ➔ `hipEvent`: mediciones dentro del código.
- ➔ `-fsanitize=address`/`valgrind` en la parte del host para memoria de CPU.

```
module load rocm/system
rocminfo | grep -E "Name|gfx|Compute Unit"
rocm-smi --showuse --showmeminfo vram
rocprof --stats ./vecadd          # tiempos por kernel
```

Comprueba con `which rocprof rocprofv3` qué versión tienes antes de planear un análisis.

Errores frecuentes en AMD/HIP

- **Arquitectura equivocada** (`--offload-arch`): el kernel no corre o da `invalid device function`. Revisa `rocminfo`.
- **Asumir** `warpSize=32`: en CDNA es 64 y rompe la lógica de *reductions* por *warp*.
- **Ignorar los códigos de error** (`nodiscard`): usa `HIP_CHECK`.
- **Leer en la CPU sin sincronizar** un resultado del kernel (los lanzamientos son asíncronos).
- **Mezclar módulos**: `rocm`, el MPI ROCm-aware y el compilador deben ser compatibles entre sí.
- **GPU no visible**: falta `--gres` en SLURM; revisa `HIP_VISIBLE_DEVICES` / `ROCR_VISIBLE_DEVICES`.

7
MPI + HIP en el cluster

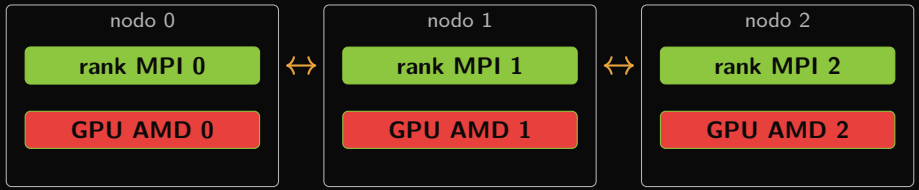
Script SLURM para HIP

```
#!/bin/bash
#SBATCH --job-name=hip_vecadd
#SBATCH --partition=amd
#SBATCH --nodes=1
#SBATCH --ntasks=1
#SBATCH --gres=gpu:1                    # 1 GPU
#SBATCH --time=00:10:00
#SBATCH --output=hip_%j.out

module load rocm/system
rocm-smi                                # que GPU nos toco
./vecadd
```

- `--gres=gpu:N` reserva *N* GPU (confirma el nombre del recurso con `scontrol show node`).
- Compila con la `--offload-arch` de la GPU del nodo; si compilas en el frontend, indícala explícitamente.

MPI + HIP: un rank por GPU




```
int rank, ngpu;
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
hipGetDeviceCount(&ngpu);
hipSetDevice(rank % ngpu);          // una GPU distinta por rank
```

Módulo `mpi/openmpi-rocm/4.1.6`: OpenMPI compilado con soporte ROCm (**por confirmar** en el cluster). Con un MPI *ROCm-aware* puedes pasar **punteros de device** a `MPI_Send`/`MPI_Recv`; si no, copia primero al host. Compila con `mpicxx` y enlaza la biblioteca HIP, o con `hipcc` y las rutas de MPI.

Script SLURM: MPI + HIP

```
#!/bin/bash
#SBATCH --job-name=mpi_hip
#SBATCH --partition=amd
#SBATCH --nodes=1
#SBATCH --ntasks-per-node=2          # 2 ranks MPI
#SBATCH --gres=gpu:2                 # 2 GPU: una por rank
#SBATCH --time=00:10:00
#SBATCH --output=mpihip_%j.out

module load rocm/system mpi/openmpi-rocm/4.1.6

mpirun -np $SLURM_NTASKS ./mpi_hip
# cada rank imprime: rank R usa la GPU R de N
```

- ➔ Confirma cuántas GPU tiene el nodo antes de pedir `gpu:2` (`scontrol show node`, `rocm-smi`).
- ➔ Para halos entre GPU: `hipMemcpy` a buffers del host y `MPI_Sendrecv`, o `MPI_Sendrecv` directo con punteros de device si el MPI es ROCm-aware.

8
Reto grupal

Reto grupal: el mismo problema en cinco modelos

★ Reto grupal – Equipos de 3 o 4 personas

Resuelvan **una difusión de calor 2D (Jacobi)** con los modelos vistos:

1. **CPU secuencial** y **OpenMP** (módulo 4) como línea base.
2. **MPI** (módulo 5) en 1 a 4 nodos, con halos.
3. **CUDA** (módulo 6) en `gpu_titan`.
4. **HIP** (este módulo): porten su kernel CUDA con `hipify` o a mano y ejecútenlo en `amd`.
5. Misma verificación numérica y mediciones repetidas en todos.
6. *Extra:* **MPI + HIP** con dos GPU, o versión SYCL (AdaptiveCpp), u OpenMP *offload*.

Entregables y criterios

Entregables

- Repositorio con las versiones (CPU, OpenMP, MPI, CUDA, HIP) y `Makefile`.
- Scripts `sbatch` de cada partición.
- Una gráfica comparativa: tiempo, GFLOP/s y ancho de banda.
- Informe: qué modelo eligen y por qué.
- Exposición del equipo.

Criterios

- Todas las versiones dan el mismo resultado.
- Mediciones justas (mismo tamaño, repeticiones, nodos exclusivos).
- Análisis con Amdahl, ancho de banda y jerarquía de memoria.
- Calidad del código y reproducibilidad (módulos, flags).

Referencias

- AMD, *ROCm Documentation* y *HIP Programming Guide*: rocm.docs.amd.com.
- AMD, *ROCm for HPC and Supercomputing*: amd.com/en/products/software/rocm.
- AMD, *Intro to AMD GPU Programming with HIP* (material de capacitación).
- HIPIFY: github.com/ROCm/HIPIFY. HIP: github.com/ROCm/HIP.
- AdaptiveCpp: github.com/AdaptiveCpp/AdaptiveCpp.
- Material del semillero: *Introduction to AMD Programming with ROCm – HIP* (L. A. Torres Niño, SC3-UIS) y módulos 3 a 6.

Cierre

★ HIP en una línea

HIP lleva el modelo de CUDA a las GPU AMD (y NVIDIA): casi el mismo código, con *wavefronts*, LDS y ROCm por debajo. **Verifica** siempre arquitectura, módulos y recursos en tu nodo.

¡Gracias!

sc3.operaciones@uis.edu.co