CINECA

Basics of GPU programming

OpenACC & CUDA

Mattia Mencagli
m.mencagli@cineca.it

19/04/2024

OpenACC

Similarly to **OpenMP**, OpenACC does not change the code, but adds **directives** in order to convey the execution on the GPU.

OpenACC

FIRST OF ALL

Git: `git clone https://github.com/mattiamencagli/gpus_course.git`

Folder: `gpus_course/openacc`

Allocate a node (modify your `allocation_script.sh` ACCOUNT variable with your account): `source allocation_script.sh`

Load module: `module load nvhpc/23.11`

Check the GPUs on the node: `nvidia-smi`

Documentation: `OpenACC-3.3-MANUAL.pdf`

Lexicon:

- Device, Accelerator: GPU
- Host: CPU

OpenACC

Standard

Easy: Directives are the easy path to accelerate compute intensive applications.

Open: OpenACC is an open GPU directives standard, making GPU programming straightforward and portable across accelerators.
[It is well implemented on NVIDIA compilers BUT not so well on GNU compiler]

Powerful: GPU Directives allow complete access to the massive parallel power of GPUs.
[But never as much powerful as CUDA]

OpenACC

Directives

Directives are added to serial source code.

- Manage loop **parallelization** and **data transfer** between CPU and GPU memory;
- Works with **C**, **C++**, or **Fortran;**
  - Can be combined with explicit CUDA C/Fortran usage;
- Directives are **formatted as comments** (as in OpenMP);
  - They don't interfere with serial execution;
  - Maintains portability of original code.
- Directives Syntax:

C, C++	**#pragma acc** directives clause(…) … { … } (optional)
Fortran	**!$acc** directive clause(…) … … **!$acc end** directive

OpenACC

Parallel and Loop Directives

```
#pragma acc parallel
{
double a = 3.14;
#pragma acc loop
for(int i=0; i<N; ++i)
    y[i] = a * x[i] + y[i];
...
}
```

The parallel region is executed sequentially on the GPU.

The Loop directives enable the parallelization of the immediately follow for loop.

```
double a = 3.14;
#pragma acc parallel loop
for(int i=0; i<N; ++i)
    y[i] = a * x[i] + y[i];
```

Compact version.

OpenACC

Kernels and Independent Directives

```
#pragma acc kernels
{
for(int i=0; i<N; ++i)
    y[i] = a * x[i] + y[i];
...
for(int i=0; i<N; ++i)
    y[i] = b * y[i-1];
}
```

The compiler try to parallelize all the loops in the kernels region.

Non-independent loop!

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

Help the compiler identifying independent loops.

OpenACC

Data Directives

- `copyin(x[0:N], … )` : Allocates memory on GPU and copy data to GPU memory on entering the region.
- `copyout(x[0:N], … )` : Allocates memory on GPU and copy data back to CPU memory on exiting the region.
- `create(x[0:N], … )` : Allocates memory on GPU without transferring data.
- `delete(x[0:N], … )` : Deallocates memory on GPU without transferring data.
- `update host/device(x[0:N], … )` : transfer data to the host or to the device.
- `present (x, … )` : specifies that the variables are already present in the current GPU.

OpenACC

Data Directives examples

```
#pragma acc data copyin(x[:N], y[:N]) copyout(y[:N])
{
#pragma acc parallel loop present(x, y)
for(int i=0; i<N; ++i)
    y[i] = a * x[i] + y[i];
}
```

```
#pragma acc enter data copyin(x[:N], y[:N])
#pragma acc parallel loop present(x, y)
for(int i=0; i<N; ++i)
    y[i] = a * x[i] + y[i];
#pragma acc exit data copyout(y[:N]) delete(x[:N])
```

```
#pragma acc parallel loop data copyin(x[:N], y[:N]) copyout(y[:N])
for(int i=0; i<N; ++i)
    y[i] = a * x[i] + y[i];
```

OpenACC

A starting point

EX 1

```
#include <openacc.h>
int main(){
...
#pragma acc enter data copyin(x[:N], y[:N])
...
#pragma acc parallel loop present(x, y)
for(int i=0; i<N; ++i)
    y[i] = a * x[i] + y[i];
...
#pragma acc data copyout/update/copyin(x[:N])
...
#pragma acc exit data delete(x[:N]) copyout(y[:N])
...
}
```

Include the necessary headers.

Allocation on GPU and copy of data on it.

Simple parallelization on the GPU threads.

You can move data within a enter-exit region.

Copy of the needed data on CPU memory, and deletion of unnecessary data on GPU.

Compilation (see `Makefile`): `nvc++ … -acc -Minfo=acc -gpu=managed,rdc -cudalib=nvtx3 -cudalib=nccl`

OpenACC

Loop optimization

EX 2

- The **seq** clause specifies that the associated loops will be executed sequentially on the accelerator;

- The **collapse(n_loops)** clause allows for extending loop to tightly nested loops.

Non-independent loop!

```
#pragma acc parallel
...
#pragma acc loop seq
for(int i=0; i<N; ++i)
    y[i] = a * x[i] + y[i-1];
```

```
#pragma acc parallel
...
#pragma acc loop
for(int j=0; j<Nj; ++j){
    int off = j*Ni;
    #pragma acc loop
    for(int i=0; i<Ni; ++i)
        y[i+off] = a * x[i+off] + y[i+off];
}
```

```
#pragma acc parallel
...
#pragma acc loop collapse(2)
for(int j=0; j<Nj; ++j)
    for(int i=0; i<Ni; ++i){
        int index = i + j * Ni;
        y[index] = a * x[index] + y[index];
    }
```

OpenACC

Reduction clause

EX 3

The **reduction** clause on a loop specifies a reduction operator on one scalar variable

- A private copy of the variable is created for each thread of the loop;
- At the end of the loop, the values for each thread are combined using the reduction operator;
- Common operators are supported: **+**, *****, **max**, **min**, …

```
#pragma acc parallel
...
double sum = 0.0;
#pragma acc loop reduction(+:sum)
for(int i=0; i<N; ++)
    sum += x[i];
```

OpenACC

Atomic clause

The **atomic** clause ensures that a variable is accessed and/or updated atomically.

- It prevents simultaneous reading or writing by threads;
- a private copy of the variable is created for each thread of the loop;
- atomic-clause is one of: **read**, **write**, **update**, or **capture**.

```
#pragma acc parallel
int x_crit_num = 0;
#pragma acc loop
for(int i=0; i<N; ++)
    ...
    if(x[i]>critical_value){
        #pragma acc atomic update
        x_crit_num++;
    }
```

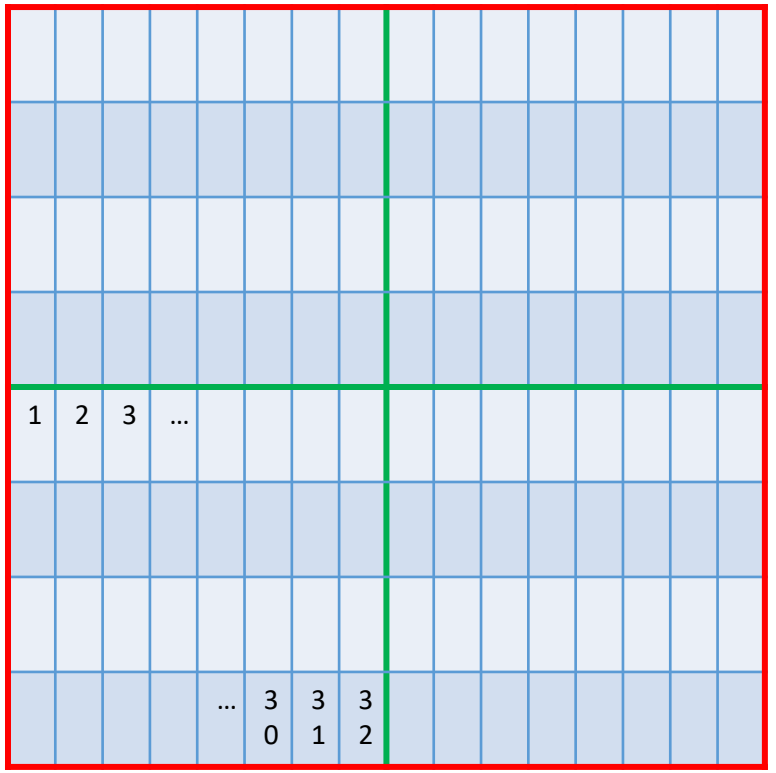
OpenACC

Levels of parallelism

In **for loop** directives, you can specify the level of parallelism:

- **Gang** : loose parallelism, gangs can work independently without synchronization;
- **Worker** : tighter parallelism, workers may share data and/or coordinate within a common gang;
- **Vector** : tightest parallelism, a vector instruction may be used across multiple data;

The equivalent of these 3 levels In CUDA are **blocks**, **warps** and **threads**.



- Gang, or block
- Worker, or warp
- Vector, or threads

OpenACC

Levels of parallelism

```
#pragma acc parallel loop gang
for(int j=0; j<Nj; ++j)
    if( j%2 == 0 ){ //j is even
        #pragma acc loop vector
        for(int i=0; i<Ni; ++i)
            A[i + j*Ni] = x[j];
    } else {
        #pragma acc loop vector
        for(int i=0; i<Ni; ++i)
            A[i + j*Ni] = 0;
    }
```

In this way we avoid **warp divergence**!

You can specify the length of vector within the gang, the number of workers within the gang and the number of gang using: **num_gangs()**, **num_worker()**, **vector_length()**.

OpenACC

Asynchronous parallelism

You can superpose multiple GPU operations at once, or superposing GPU operations with CPU operations using "async".

```
...
#pragma acc parallel loop present(x, y) async
for(int i=0; i<N; ++i)
    y[i] = a * x[i] + y[i];
int z = c[0]; // CPU operations
#pragma acc wait
#pragma acc exit data delete(x[:N]) copyout(y[:N]) async
...
```

```
#pragma acc enter data copyin (z[:N]) async(1)
#pragma acc parallel loop present(x, y) async(2)
for(int i=0; i<N; ++i)
    y[i] = a * x[i] + y[i];
...
#pragma acc parallel loop present(z) async(1)
for(int i=0; i<N; ++i)
    z[i] = a;
    z[i] = a * y[i];
```

Stream 1

Stream 2

Stream 1

Each **stream** has his sequence of operations that execute in issue-order on the GPU.

OpenACC

Routines

EX 4

In order to use a function within a parallelized loop you have to mark that function with `#pragma acc routine`

```
#pragma acc routine
double add(double a, double b){
    return a + b;
}
...
#pragma acc parallel loop
for(int i=0; j<N; ++i)
    y[i] = add(x[i], a);
```

```
#pragma acc routine seq
double add(double a, double b){
    return a + b;
}
...
#pragma acc parallel loop
for(int i=0; j<N; ++i)
    y[i] = add(x[i], a);
```

CUDA

CUDA (Compute Unified Device Architecture) is a general-purpose parallel computing platform and programming model to code on GPUs created by Nvidia.

CUDA

FIRST OF ALL

Folder: `gpus_course/CUDA`

Allocate a node: `source allocation_script.sh`

Load module: `module load nvhpc/23.11`

Check the GPUs on the node: `nvidia-smi`

Documentation: CUDA Runtime API :: CUDA Toolkit Documentation (nvidia.com)

Lexicon:

- Device, Accelerator: GPU
- Host: CPU

CUDA

Standard

Not that easy: the code must be almost completely refactored. However, it is mostly as using C, steep learning curve but doable.

Private: it is proprietary to Nvidia GPUs, applications developed using CUDA are limited to Nvidia hardware. However, AMD is catching up with HIP.

[cudaMemcpy(dst, src, count, cudaMemcpyDeviceToHost) → hipMemcpy(dst, src, count, hipMemcpyDeviceToHost)]

Very powerful: with CUDA you can really take complete advantage of the massive parallelism that a GPU can give you.

CUDA

Kernels

A function which runs on a GPU is called "**kernel**".

- when a kernel is launched on a GPU thousands of threads will execute its code;
- programmer must choose the number of **threads** and **blocks** to run;
- each thread acts on a different data element independently.

```
example_kernel<<< 100, 128 >>>( N, a, b, c )
```

1° parameter:
defines the number of blocks to use.

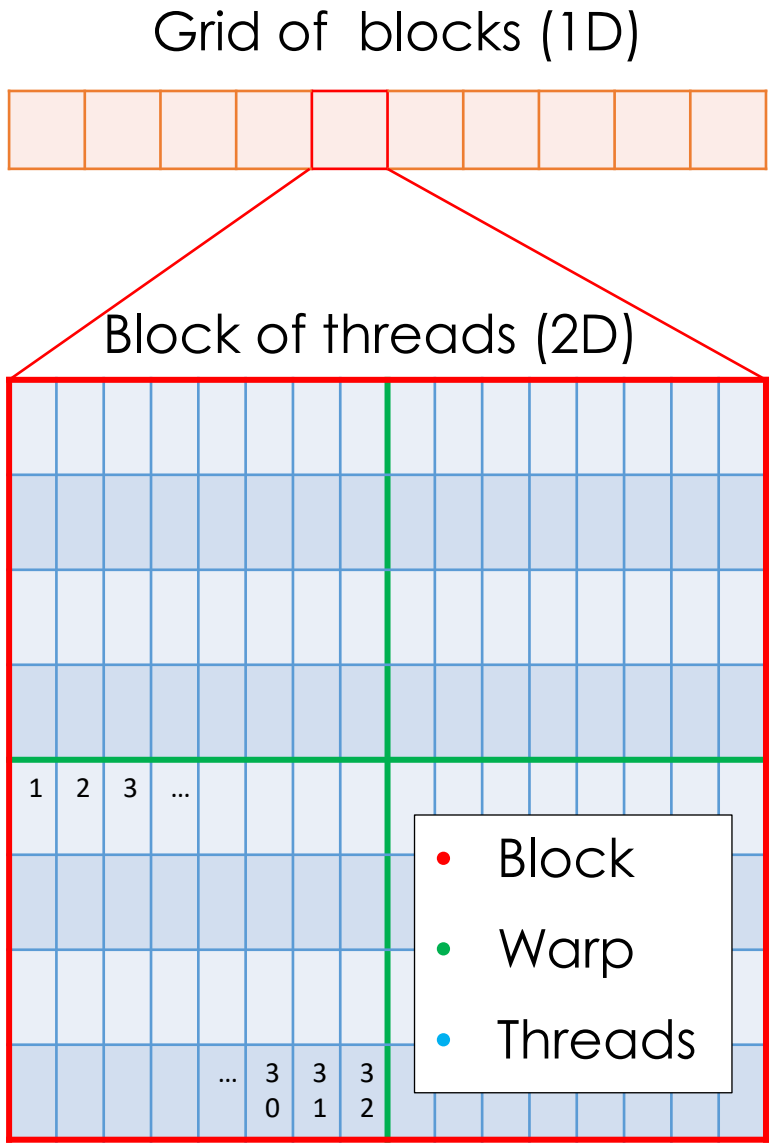
2° parameter:
defines the number of threads per block.
(you want a multiple of **32**)

CUDA

Thread Hierarchy

- Threads are organized into blocks of threads
  *[blocks structure can be **1D**, 2D, 3D sized in threads]*
- Blocks are organized in a grid of blocks
  *[blocks can be organized into a **1D**, 2D, 3D grid of blocks]*
- each block or thread has a unique ID
  *[use **.x**, .y, .z to access its components]*

threadIdx: thread coordinates inside the block
blockIdx: block coordinates inside the grid
blockDim: block dimensions in thread units
gridDim: grid dimensions in block units



CUDA

Kernels launch

```
#include <cuda.h>
#include <cuda_runtime.h>
...
__device__ double add (double *A, double *B, int i)
    return A[i] + B[i];
...
__global__ void vecAddGPU (int N, double *A,
                           double *B, double *C) {
     int i = blockIdx.x * blockDim.x + threadIdx.x;
     if (i < N) C[i] = add(A, B, i);
}
...
int N = 130;
dim3 threads( 128, 1, 1 ); // 4 warps
dim3 blocks ( (N - 1) / threads.x + 1, 1, 1);
vecAddGPU<<<blocks, threads>>>( N, a, b, c );
```

Basic CUDA headers.

Kernels with "__device__" are callable only from GPU kernels.

Kernels with "__global__" are callable from CPU.

Global index computation, where the offset is: `blockIdx.x*blockDim.x`

dim3 is a CUDA type.

We use blocks with **128** threads (4 warp).

We add one extra block for reminders.

CUDA

Memory accesses and Indexing

for each block:
threadIdx.x = { 0, 1, 2, … , 127 }

```
__device__ double add (double *A, double *B, int i)
    return A[i] + B[i];
...
__global__ void vecAddGPU (int N, double *A,
                           double *B, double *C) {
     int i = blockIdx.x * blockDim.x + threadIdx.x;
     if (i < N) C[i] = add(A, B, i);
}
...
int N = 130;
dim3 threads( 128, 1, 1 ); // 4 warps
dim3 blocks ( (N - 1) / threads.x + 1, 1, 1);
vecAddGPU<<<blocks, threads>>>( N, a, b, c );
```

for **blockIdx.x** = 0
i = 0 * 128 + threadIdx.x = { 0, … , 127 }
for **blockIdx.x** = 1
i = 1 * 128 + threadIdx.x = { **128**, **129**, … , 255 }

The if condition ensure we do not try to access non-allocated areas of memory.

Accelerator Fatal Error: ... returned error 700: Illegal address during kernel execution

We have **warp divergence** just in the last block!

CUDA

Data movement

```
int N = 130;
int size = N * sizeof(double);
double *h_a, *d_a; //host and device
cudaMallocHost(&h_a, size);
cudaMalloc(&d_a, size);
...
cudaMemcpy(a_d, a_h, size, cudaMemcpyHostToDevice);
...
dim3 threads( 128, 1, 1 ); // 4 warps
dim3 blocks ( (N - 1) / threads.x + 1, 1, 1);
vecAddGPU<<<blocks, threads>>>( N, d_a, d_b, d_c );
...
cudaMemcpy(h_c, d_c, size, cudaMemcpyDeviceToHost);
...
cudaFree(d_a);
cudaFreeHost(h_a);
```

One pointer for the **host** (CPU) and on for the **device** (GPU).
Host memory Allocation.
Device memory Allocation.

CPU → GPU memory transfer.

GPU → CPU memory transfer.

Deallocation.

CUDA

Compilation

EX 1

Compilation (see `Makefile`): `nvcc … -arch=sm_80`

The number next to "`-arch=sm_`"
is the Compute Capability of the GPU

	Pascal 100	Tesla 100	Amper 100	Hopper 100
Comp. Cap.	6.0	7.0	8.0	9.0

Gallielo100
Marconi100

Leonardo

```
#define CUDA_SAFE_CALL(ans) { gpuAssert((ans), __FILE__, __LINE__); }

inline void gpuAssert(cudaError_t code, const char *file, int line) {
    if (code != cudaSuccess) {
        fprintf(stderr, "GPUassert: %s %s %d\n", cudaGetErrorString(code), file, line);
        exit(code);
    }
}
```

CUDA

Multidimensional Blocks and Threads

EX 2

```
__global__ void matmulGPU (int N, double *A, double *B, double *C) {
    int i = blockIdx.x * blockDim.x + threadIdx.x; // col
    int j = blockIdx.y * blockDim.y + threadIdx.y; // raw
        if( i < N && j < N ) {
            ...
        }
}
...
int N = 1000; // square matrix NxN
dim3 threads( 16, 16, 1 ); // 8 warps
dim3 blocks ( (N - 1) / threads.x + 1, (N - 1) / threads.y + 1, 1);
matmulGPU<<<blocks, threads>>>( N, d_a, d_b, d_c );
...
```

Double index,
“equivalent” to a double for loop

Simple example:
Square blocks, and square grid.

CUDA

Asynchronous operations and streams

Each **stream** has his sequence of operations that execute in issue-order on the GPU.

```
...
cudaStream_t stream_1, stream_2;
cudaStreamCreate(&stream_1);
cudaStreamCreate(&stream_2);
...
cudaMemcpyAsync(z_d, z_h, size,
                    cudaMemcpyHostToDevice, stream_1);
...
vecAddGPU<<<blocks, threads, 0, stream_2>>>(N, d_a, d_b, d_c);
cudaStreamSyncronize(stream_2);
vecAddGPU<<<blocks, threads, 0, stream_1>>>(N, d_c, d_z, d_w);
cudaDeviceSynchronize();
...
```

Declare and create streams.

Async mem transfer on stream_1.

Async kernel on stream_2.

Wait for work on stream_2 to end.

Wait for all streams on GPU to end their work.

CUDA

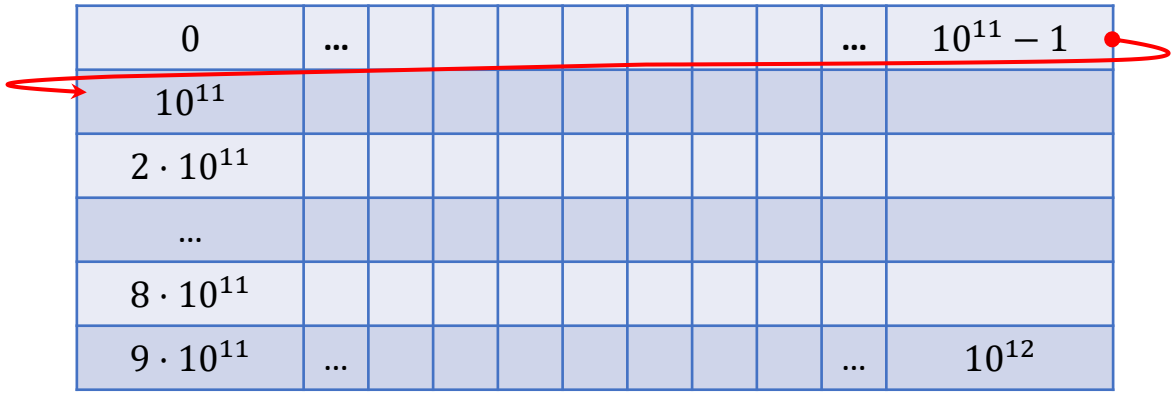
Strides

```
__global__ void vecAddGPU (int N, double *A,
                           double *B, double *C) {
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  int stride = gridDim.x * blockDim.x;

  for (size_t j = i; j < N; j += stride)
      C[j] = A[j] + B[j];
}
...
size_t N = 1e12;
size_t Nthreads = N / 10; // 1e11
dim3 threads( 128, 1, 1 ); // 4 warps
dim3 blocks ( (Nthreads - 1) / threads.x + 1, 1, 1);
vecAddGPU<<<blocks, threads>>>( N, d_a, d_b, d_c );
```

Grid of blocks (1D), 781.250.000 blocks.

Block of threads (1D),128 threads.



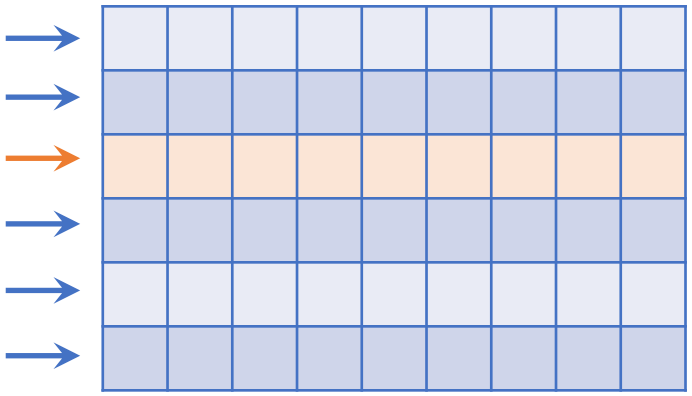
Optimal pattern to access the GPU global memory.

CUDA

Contiguous vs. Cached memory access

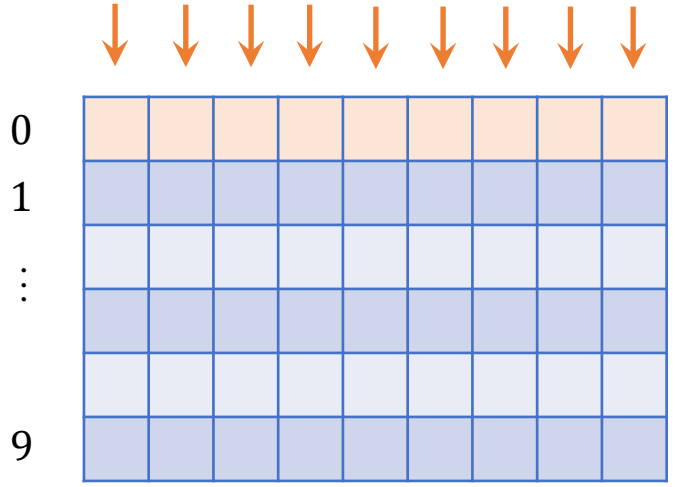
Cached: good on CPU

Example: OpenMP or MPI tasks. Each task get its portion of the arrays.



Contiguous : good on GPU

Example: 32 threads of a warp. Each thread has an element, and then you offset the entire warp.



CUDA

Unified memory (or managed memory)

EX 3

```
double *a;
cudaMallocManaged(&a, size);
... // Initialization on CPU.
int deviceID, numberOfSMs;
cudaGetDevice(&deviceID);
cudaDeviceGetAttribute(&numberOfSMs,
            cudaDevAttrMultiProcessorCount, deviceId);
cudaMemPrefetchAsync(a, size, deviceID);
...
dim3 threads( 128, 1, 1 ); // 4 warps
dim3 blocks ( numberOfSMs * 16 , 1, 1);
GPU_func<<<blocks, threads>>>( N, a, b, c );
cudaDeviceSynchronize();
...
printf(a[0]);
```

One single pointer for **host** and **device**.

Each GPU has its own ID, which can be used to extract information on the GPU hardware.

Memory transfer optimization.

While using *strides* you can choose an optimal number of blocks.

Automatic CPU → GPU memory transfer.
CUDA barrier (crucial in this case).

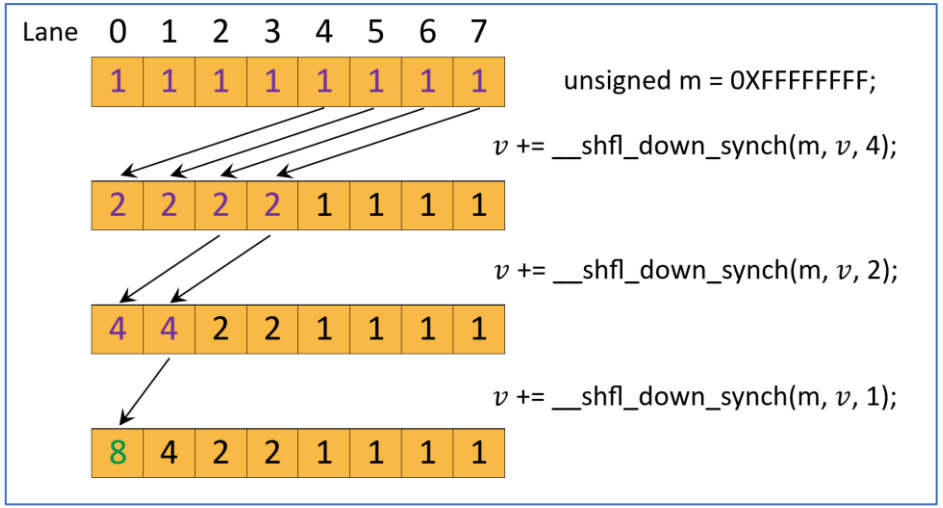
Automatic GPU → CPU memory transfer.

CUDA

Reduction

EX 4

```
__device__ float warpReduce (float val){
for(int k=16; k>0; k/=2)
    val += __shfl_down_sync(0xFFFFFFFF, val, k, 32);
return val;
}
__device__ float blockReduce(float val){
    static __shared__ float shared[32]; // Shared mem for 32 partial sums
    int threads_localwarp_id = threadIdx.x % 32; // Lane
    int warp_id = threadIdx.x / 32;
    val = warpReduce(val); // Each warp performs partial reduction
    if (threads_localwarp_id == 0)
        shared[warp_id] = val; // Write reduced value to shared memory
    __syncthreads(); // Wait for all partial reductions
    //read from shared memory
    val = (threadIdx.x < blockDim.x / 32) ? shared[threads_localwarp_id] : 0.0;
    if (warp_id == 0)
        val = warpReduce(val); //Final reduce within first warp
    return val;
}
```



The "first thread" of each warp will have the partial reduction of "v" on the warp.

```
Value = blockReduce (Value);
```

EX 4

CUDA

Reduction with CUDA UnBound (CUB) library

CUB library: CUB :: CUDA Toolkit Documentation (nvidia.com)

Parallel primitives:

- **Warp-wide** "collective" primitives
  - Cooperative warp-wide prefix scan, **reduction**, etc.
  - Safely specialized for each underlying CUDA architecture
- **Block-wide** "collective" primitives
  - Cooperative I/O, sort, scan, **reduction**, histogram, etc.
  - Compatible with arbitrary thread block sizes and types
- **Device-wide** primitives
  - Parallel sort, prefix scan, **reduction**, histogram, etc.
  - Compatible with CUDA dynamic parallelism

CUDA

Multiple GPUs

EX 5

```
int MPI_RANK, NumberOfProcessors;
MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &MyRank);
MPI_Comm_size(MPI_COMM_WORLD, &NumberOfProcessors);
...
int Ngpus;
cudaGetDeviceCount(&Ngpus);
...
cudaSetDevice(MPI_RANK);
GPU_func<<<blocks, threads>>>( N, a, b, c, f(MPI_RANK) );
cudaDeviceSynchronize();
...
```

MPI initialization.

Get total number of GPUs (However sometimes it is better to use `NumberOfProcessors`).

Uses a different GPU for each MPI rank.

When using multiple GPUs you will need an **MPI library** (OpenMPI, NCCL, SpectrumMPI, …)
In order to take care of communication between GPUs (ex: reduction).

CINECA

END

Mattia Mencagli
m.mencagli@cineca.it