

Introduction to GPU Accelerators and CUDA Programming



25th Summer School
on Parallel Computing

11-22 July 2016

Sergio Orlandini
s.orlandini@ Cineca.it

- Hands on
 - *Chunk execution*
 - *Naive version*
 - *Stream version*
 - *Multi-GPU version*
 - *Profiling session*

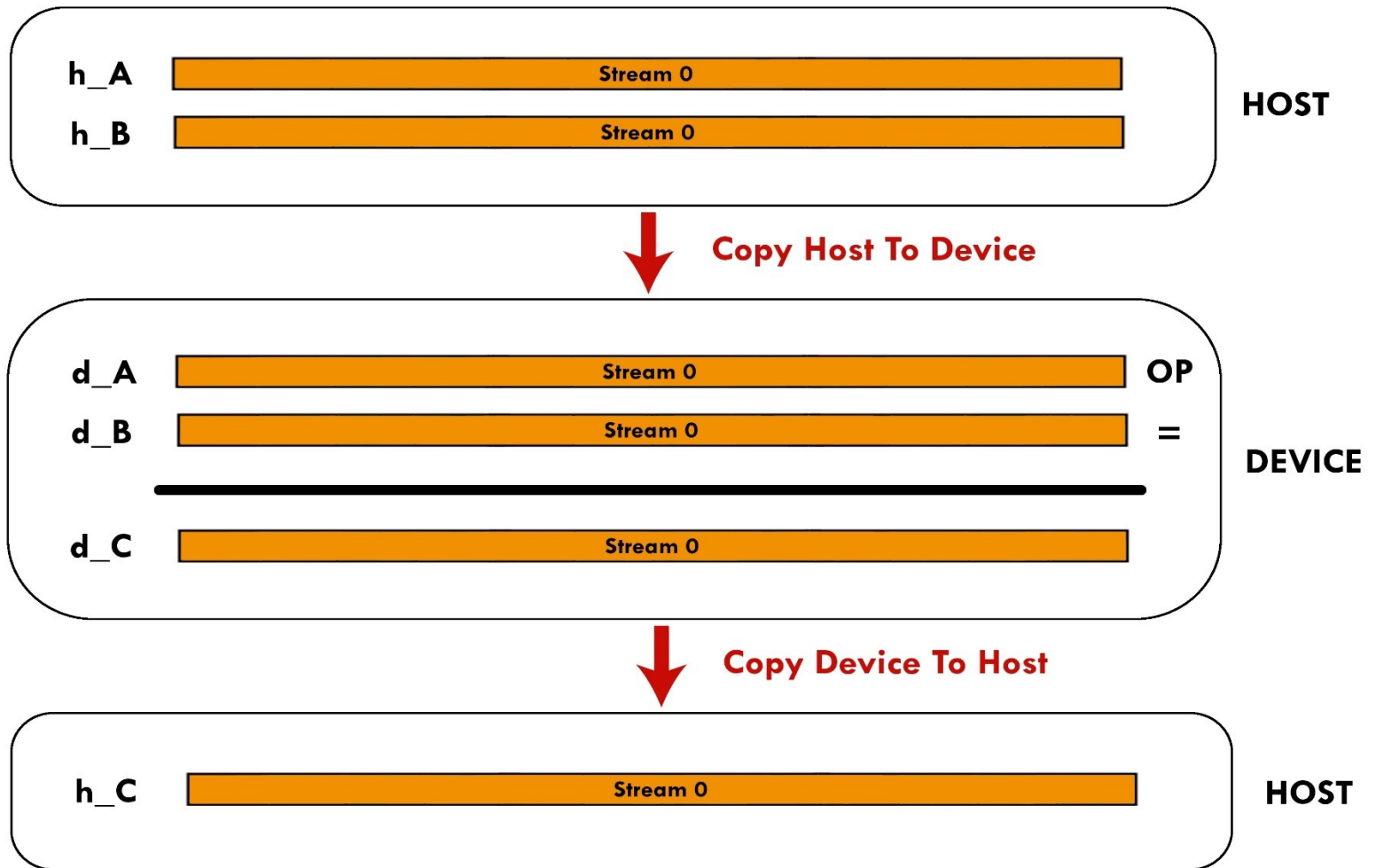


Hands-on Streams: naive version

Write a C or F90 program which performs the following operations:

- Allocate `h_A`, `h_B`, `h_C` single precision arrays of `nSize` elements on *host*
- Initialize `h_A` and `h_B` arrays using the `initArrayData()` function in C or the `RANDOM_NUMBER()` subroutine in F90
- Allocate `d_A`, `d_B`, `d_C` single precision arrays on the *device*
- Transfer data from `h_A` and `h_B` arrays on the `d_A` and `d_B` arrays
- Launch the `arrayFunc()` kernel which combine data from `d_A` and `d_B` and write results onto array `d_C`
- Copy back `d_C` array from *device* in `h_C` array on *host*
- Measure the total elapsed time to perform both kernel and memory transfers using `cudaEvents`
- Execute the `funcArrayCPU()` function which replicates the same CUDA kernel on host for result comparison
- Measure the elapsed time of the `funcArrayCPU()` function
- Compute the Speed Up of GPU implementation as CPU time / GPU time

Hands-on Streams: naive version

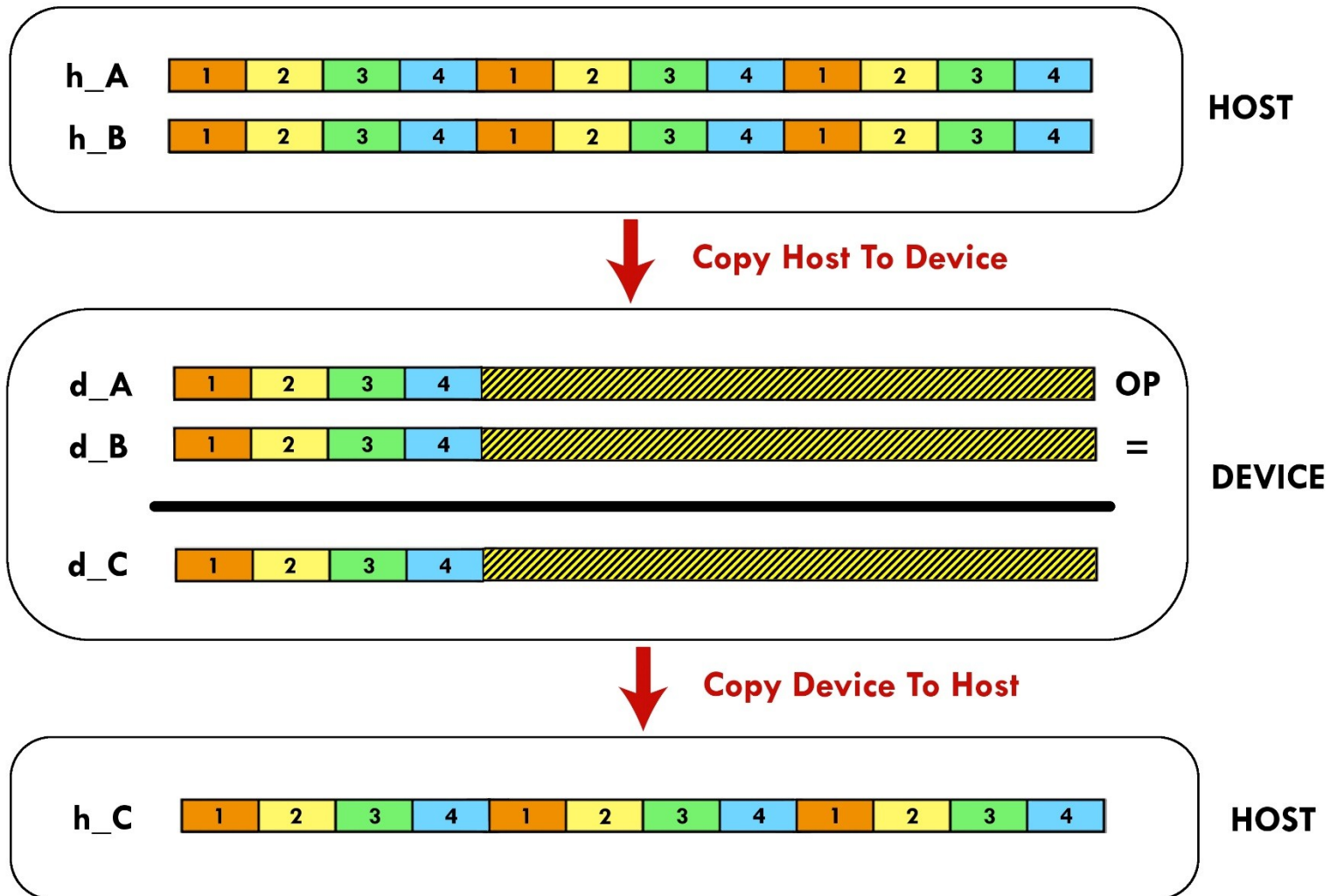


Hands-on Streams: using cudaStreams

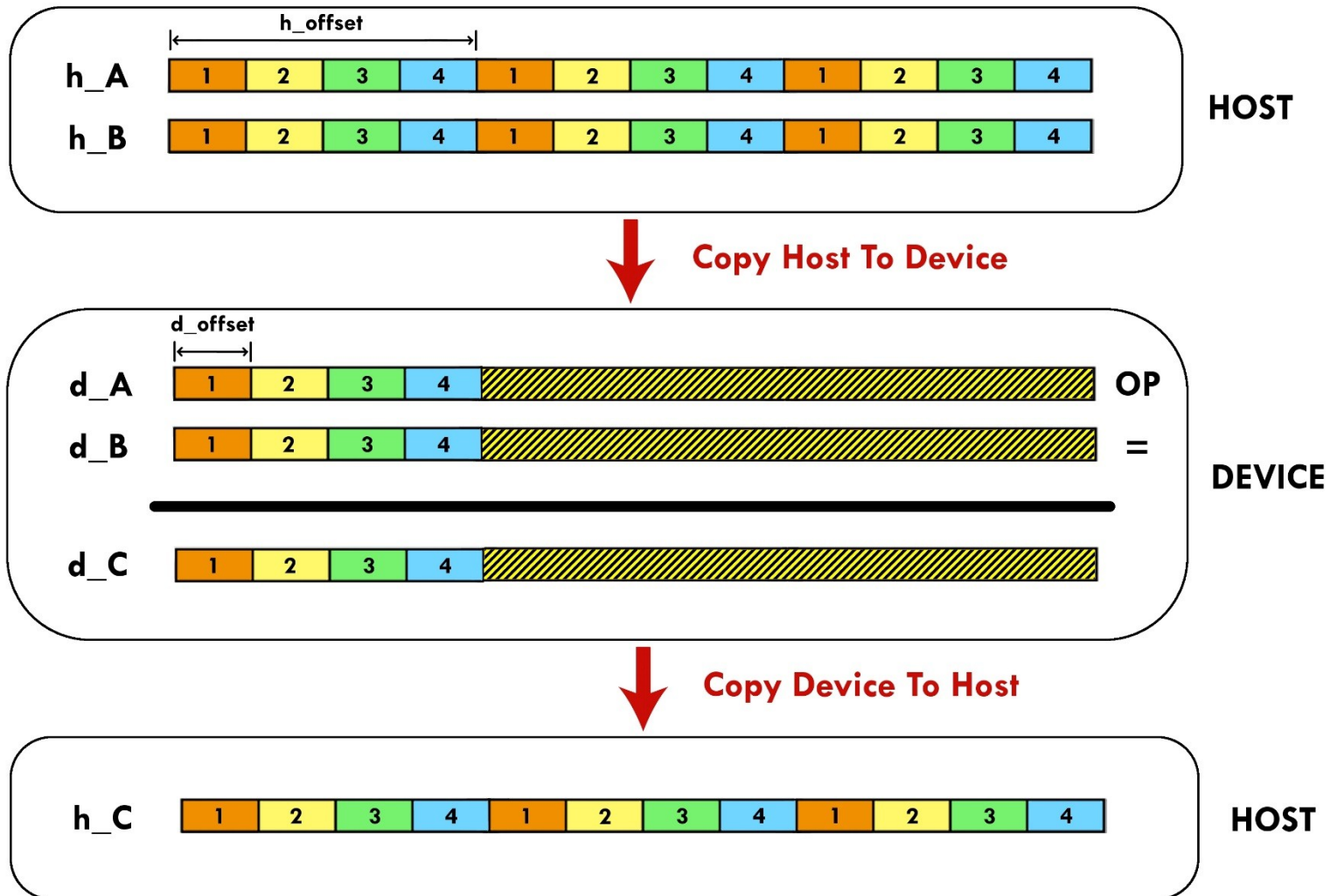
Write a C or F90 program which performs the following operations:

- Allocate `h_A`, `h_B`, `h_C` single precision arrays of `nSize` elements on *host*
- Initialize `h_A` and `h_B` arrays using the `initArrayData()` function in C or the `RANDOM_NUMBER()` subroutine in F90
- Split the elaboration of `h_A`, `h_B` arrays into chunks of `chunk_size` size elements
- Create `streams_number` of `cudaStream`
- Allocate `d_A`, `d_B`, `d_C` of `chunk_size * streams_number` size on the *device*
- Assign to each `cudaStream` the elaboration of each chunk. Each stream will:
 - copy a chunk of data from `h_A` and `h_B` on `d_A` and `d_B` buffers
 - Launch the kernel `arrayFunc`
 - Copy back to *host* the results from `d_C` into `h_C`
- Measure execution time and compare the speedup with respect *naïve* implementation
 - Try to change the number of active streams, the chunk size, etc...

Hands-on Streams: using cudaStreams



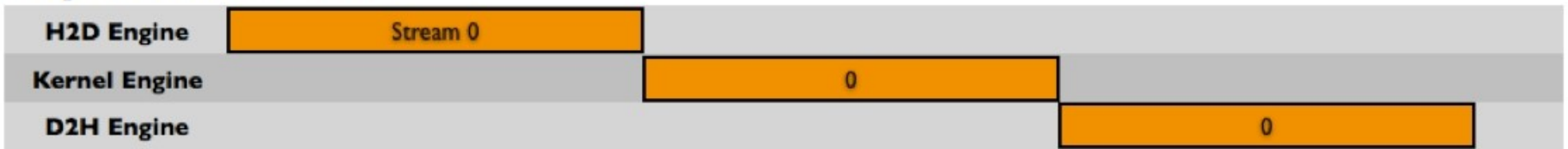
Hands-on Streams: using cudaStreams



Hands-on Streams: using cudaStreams

Execution Time Lines

Sequential Version



Asynchronous Versions



Time →

Hands-on Streams: using cudaStreams

CUDA Runtime functions to implement the code (C for CUDA):

- `cudaError_t cudaStreamCreate(cudaStream_t *stream)`
- `cudaError_t cudaStreamDestroy(cudaStream_t stream)`
- `cudaError_t cudaDeviceSynchronize(void)`
- `cudaError_t cudaMemcpyAsync(void* dst, void* src, size_t nbyte, enum cudaMemcpyKind kind, cudaStream_t stream)`

CUDA Runtime functions to implement the code (CUDA FORTRAN):

- `integer function cudaStreamCreate(stream)`
`integer :: stream`
- `integer function cudaStreamDestroy(stream)`
`integer :: stream`
- `integer function cudaDeviceSynchronize()`
- `integer function cudaMemcpyAsync(dst, src, nelements, kind, stream)`

Hands-on Streams: cudaStreams and Multi-GPU

Write a C or F90 program which performs the following operations:

- Allocate `h_A`, `h_B`, `h_C` single precision arrays of `nSize` elements on *host*
- Initialize `h_A` and `h_B` arrays using the `initArrayData()` function in C or the `RANDOM_NUMBER()` subroutine in F90
- Split the elaboration of `h_A`, `h_B` arrays into chunks of `chunk_size` size elements
- Assign to each available GPU device a balanced number of chunks to process
- Create `streams_number` of `cudaStream`
- Allocate `d_A`, `d_B`, `d_C` of `chunk_size * streams_number` size on the *device*
- Assign to each `cudaStream` the elaboration of each chunk. Each stream will:
 - copy a chunk of data from `h_A` and `h_B` on `d_A` and `d_B` buffers
 - Launch the kernel `arrayFunc`
 - Copy back to *host* the results from `d_C` into `h_C`
- Measure execution time and compare the speedup with respect single GPU implementation