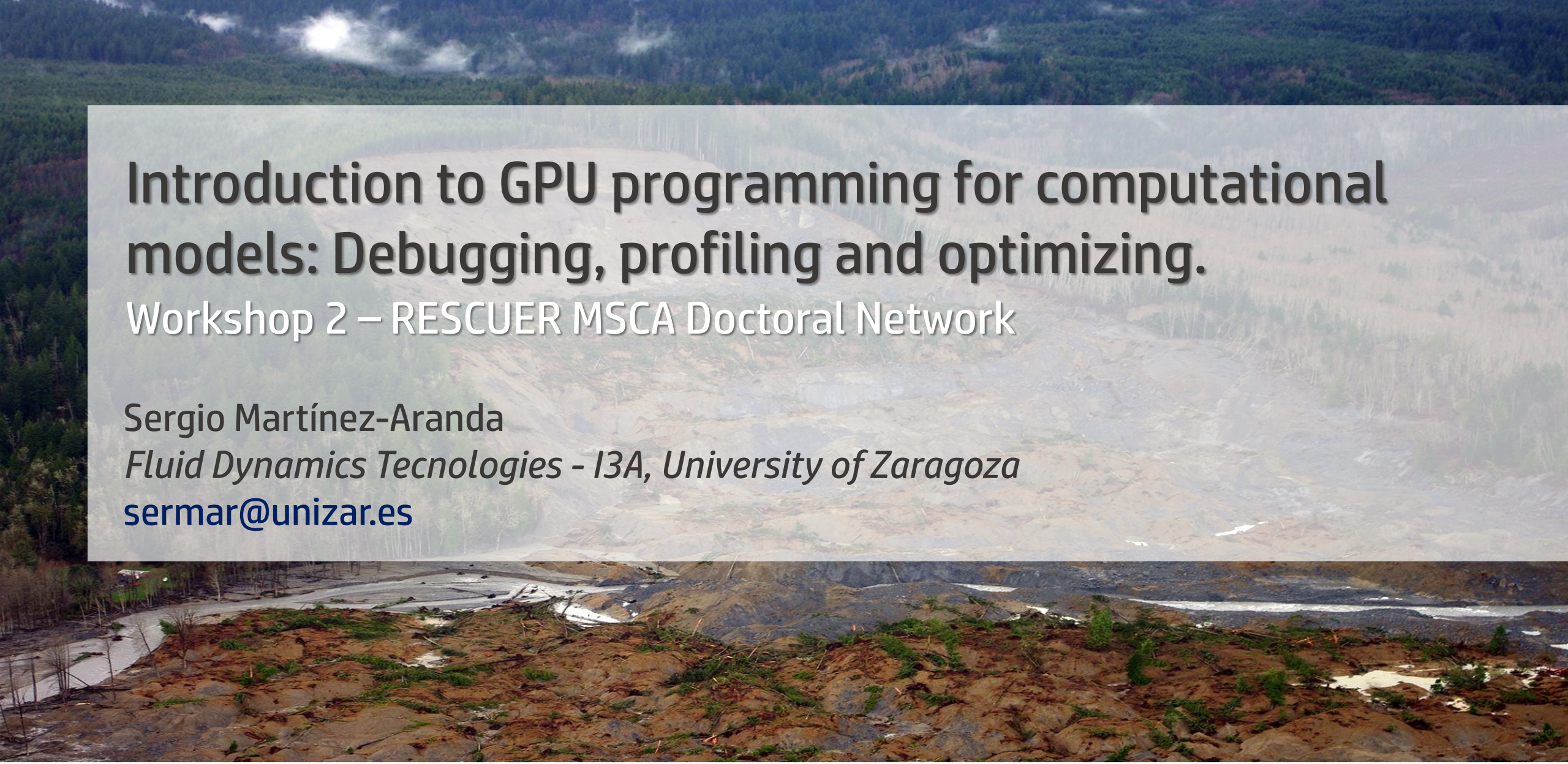


Introduction to GPU programming for computational models: Debugging, profiling and optimizing.

Workshop 2 – RESCUER MSCA Doctoral Network

Sergio Martínez-Aranda
Fluid Dynamics Tecnologies - I3A, University of Zaragoza
sermar@unizar.es

Universidad Zaragoza
1542

Escuela de Ingeniería y Arquitectura Universidad Zaragoza

Instituto Universitario de Investigación en Ingeniería de Aragón Universidad Zaragoza

Module content

1. Optimizing tools
2. Debugging code with Compute Sanitizer
3. Profiling code with Nsight System

Module content

1. Optimizing tools
2. Debugging code with Compute Sanitizer
3. Profiling code with Nsight System

CUDA Toolkit optimization

1. CUDA-GDB

A GPU-aware debugger based on GDB for debugging CUDA applications.

- Debug both host (CPU) and device (GPU) code.
- Set breakpoints, inspect variables, and step through CUDA kernels.
- Examine CUDA threads, blocks, and grids.

(NVIDIA GeForce 256, 1999)

2. Nsight Systems

A system-wide performance analysis tool for CUDA applications.

- Timeline view of CPU, GPU, and memory activities.
- Profiling of CUDA kernels, memory transfers, and API calls.
- Identify performance bottlenecks.

(NVIDIA GeForce RTX 5090, 2025)

CUDA Toolkit optimization

3. Nsight Compute

An interactive kernel profiler for CUDA applications.

- Detailed metrics for CUDA kernels (e.g., warp execution efficiency, memory bandwidth).
- Source-level analysis and optimization hints.
- Customizable profiling sessions.

(NVIDIA GeForce 256, 1999)

4. Compute Sanitizer

A suite of tools for detecting and diagnosing issues in CUDA applications.

- Memory checking (e.g., out-of-bounds access, memory leaks).
- Race condition detection.
- Initialization checking

(NVIDIA GeForce RTX 5090, 2025)

Module content

1. Optimizing tools
2. Debugging code with Compute Sanitizer
3. Profiling code with Nsight System

Debugging with Compute Sanitizer

Compilation flags

- Include –g: Add information for the host code.
- Include –G: Add information for the device code.
- Disable compiler optimizations.

Check for memory errors

Memory errors include out-of-bounds accesses, memory leaks, and illegal memory operations.

Check for memory initialization

Detects uninitialized memory accesses

Check for race conditions

Multiple threads access shared data concurrently without proper synchronization.

Check for synchronization conditions

Detects synchronization errors, such as incorrect use of barriers or warps

Check for memory errors

Memory-related issues during execution can range from out-of-bounds memory access, memory leaks and other illegal memory operations.

Execution

```
>> compute-sanitizer --tool memcheck ./my_cuda_app app_command_line_options
```

Optional flags

--leak-check full	Perform a detailed memory leak check
--track-unused-memory yes	Track memory that is allocated but never used.
--log-file <file>	Save the output to a file.
--verbose	Enable verbose output for more detailed information.

Check for memory errors

>> compute-sanitizer --tool memcheck ./my_cuda_app app_command_line_options

```
7
8   __global__ void outOfBoundsKernel(int *array, int N) {
9       int idx = blockIdx.x * blockDim.x + threadIdx.x;
10
11      array[idx] = idx;
12  }
13
14  int main() {
15
16      // Allocate device memory
17      int N = 100;
18      int *d_array;
19      cudaMalloc((void**) &d_array, N * sizeof(int));
20
21      ...
22      // Run CUDA kernel
23      int threadsPerBlock = 256;
24      int blocksPerGrid = N/threadsPerBlock + 1;
25      outOfBoundsKernel <<< blocksPerGrid, threadsPerBlock >>> (d_array, N);
26      ...
27  }
```

Threads with idx>=N will access to an out-of-bound memory position

d_array allocate 100 integer elements in the global memory

The grid for the kernel has

1 block * 256 threads/block = 256 threads

Check for memory errors

Memcheck output message

```
========= Invalid __global__ write of size 4 bytes
=========     at 0x00000120 in my_kernel
=========     by thread (1, 0, 0) in block (0, 0, 0)
=========     Address 0x10000000 is out of bounds
=========     Saved host backtrace up to driver entry point at kernel launch time
=========     Host Frame: [0x00007f8e1a2b3e45]
=========                 in /lib/x86_64-linux-gnu/libc.so.6
=========     Host Frame: [0x00007f8e1a2b3e45]
=========                 in /lib/x86_64-linux-gnu/libc.so.6
=========
========= Leaked 256 bytes at 0x20000000
=========     Allocated in my_kernel at 0x00000150
=========     Saved host backtrace up to driver entry point at memory allocation time
=========     Host Frame: [0x00007f8e1a2b3e45]
=========                 in /lib/x86_64-linux-gnu/libc.so.6
=========
========= ERROR SUMMARY: 2 errors
```

Error type

- Invalid __global__ write
- Invalid __global__ read
- Leak
- Misaligned address
- Illegal memory operation

Check for memory errors

Memcheck output message

```
========= Invalid __global__ write of size 4 bytes
=========     at 0x00000120 in my_kernel
=========     by thread (1, 0, 0) in block (0, 0, 0)
=========     Address 0x10000000 is out of bounds
=========     Saved host backtrace up to driver entry point at kernel launch time
=========     Host Frame: [0x00007f8e1a2b3e45]
=========                 in /lib/x86_64-linux-gnu/libc.so.6
=========     Host Frame: [0x00007f8e1a2b3e45]
=========                 in /lib/x86_64-linux-gnu/libc.so.6
=========
========= Leaked 256 bytes at 0x20000000
=========     Allocated in my_kernel at 0x00000150
=========     Saved host backtrace up to driver entry point at memory allocation time
=========     Host Frame: [0x00007f8e1a2b3e45]
=========                 in /lib/x86_64-linux-gnu/libc.so.6
=========
========= ERROR SUMMARY: 2 errors
```

Location:

- CUDA kernel name
- Thread Idx (x,y,z)
- Block Idx (x,y,z)

Check for memory errors

Memcheck output message

```
========= Invalid __global__ write of size 4 bytes
=========     at 0x00000120 in my_kernel
=========     by thread (1, 0, 0) in block (0, 0, 0)
=========     Address 0x10000000 is out of bounds
=========     Saved host backtrace up to driver entry point at kernel launch time
=========     Host Frame: [0x00007f8e1a2b3e45]
=========                 in /lib/x86_64-linux-gnu/libc.so.6
=========     Host Frame: [0x00007f8e1a2b3e45]
=========                 in /lib/x86_64-linux-gnu/libc.so.6
=========
========= Leaked 256 bytes at 0x20000000
=========     Allocated in my_kernel at 0x00000150
=========     Saved host backtrace up to driver entry point at memory allocation time
=========     Host Frame: [0x00007f8e1a2b3e45]
=========                 in /lib/x86_64-linux-gnu/libc.so.6
=========
========= ERROR SUMMARY: 2 errors
```

Error type

Location:

Address:

Address

Check for memory errors

Memcheck output message

```
========= Invalid __global__ write of size 4 bytes
=========     at 0x00000120 in my_kernel
=========     by thread (1, 0, 0) in block (0, 0, 0)
=========     Address 0x10000000 is out of bounds
=========     Saved host backtrace up to driver entry point at kernel launch time
=========     Host Frame: [0x00007f8e1a2b3e45]
=========                 in /lib/x86_64-linux-gnu/libc.so.6
=========     Host Frame: [0x00007f8e1a2b3e45]
=========                 in /lib/x86_64-linux-gnu/libc.so.6
=========
========= Leaked 256 bytes at 0x20000000
=========     Allocated in my_kernel at 0x00000150
=========     Saved host backtrace up to driver entry point at memory allocation time
=========     Host Frame: [0x00007f8e1a2b3e45]
=========                 in /lib/x86_64-linux-gnu/libc.so.6
=========
========= ERROR SUMMARY: 2 errors
```

Call stack

- Sequence of function calls leading to the error origin.

Check for memory errors

Invalid __global__ write Writing to an invalid memory location, such as out-of-bounds or freed memory.

Invalid __global__ read Reading from an invalid memory location.

- Ensure all memory accesses are within the allocated bounds.
- Check array indices and pointer arithmetic

Leak Memory was allocated but not freed.

- Free all allocated memory using cudaFree or cudaFreeHost.

Misaligned address Accessing memory with improper alignment, such as accessing a float2 at an odd address.

- Ensure memory is properly aligned for the data type being accessed

Illegal memory operation Performing an illegal operation, such as accessing freed memory.

- Avoid accessing memory after it has been freed.
- Use proper synchronization to prevent race conditions.

Check for memory initialization

Memory allocated on the device (GPU) or host (CPU) is not properly initialized before being used by API functions or in CUDA kernels.

Execution

>> compute-sanitizer --tool initcheck ./my_cuda_app app_command_line_options

Optional flags

--track-unused-memory yes	Track memory that is allocated but never used.
--log-file <file>	Save the output to a file.
--verbose	Enable verbose output for more detailed information.

Check for memory initialization

>> compute-sanitizer --tool initcheck ./my_cuda_app app_command_line_options

```
58
59  int main() {
60      // Allocate device memory
61      int N = 100
62
63      ...
64
65      int *h_array = (int*) malloc(N * sizeof(int));
66      cudaMemcpy(d_array, h_array, size, cudaMemcpyHostToDevice);
67
68      ...
69
70      int *d_array;
71      cudaMalloc((void**) &d_array, N * sizeof(int));
72      kernel <<< blocks, threads>>> (d_array, N);
73
74      ...
75
76      cudaMemcpy(d_array, h_array, N/2 * sizeof(int), cudaMemcpyHostToDevice);
77      cudaMemset(d_array, 0, N/2 * sizeof(int));
78
79      ...
80  }
```

Host memory is not initialized before cudaMemcpy

Device global memory is not initialized before launch kernel

Partial initialization with cudaMemcpy or cudaMemset

Check for memory initialization

Initcheck output message

```
========= Uninitialized __global__ read of size 4 bytes
=========     at 0x00000120 in init_kernel
=========     by thread (1, 0, 0) in block (0, 0, 0)
=========     Address 0x10000000 is uninitialized
=========     Saved host backtrace up to driver entry point at kernel launch time
=========     Host Frame: [0x00007f8e1a2b3e45]
=========                 in /lib/x86_64-linux-gnu/libc.so.6
=========     Host Frame: [0x00007f8e1a2b3e45]
=========                 in /lib/x86_64-linux-gnu/libc.so.6
=========
========= ERROR SUMMARY: 1 error
```

Memory sise

Location

- Kernel
- Thread and block

Address

Memory address being accessed.

Call stack

Initializing the memory

- API functions: cudaMemcpy, cudaMemset, etc.
- Use initialization kernels.

Check for race conditions

A race condition in CUDA occurs when multiple threads access and modify shared memory or global memory concurrently without proper synchronization.

Execution

>> compute-sanitizer --tool racecheck ./my_cuda_app app_command_line_options

Optional flags

--track-unused-memory yes	Track memory that is allocated but never used.
--log-file <file>	Save the output to a file.
--verbose	Enable verbose output for more detailed information.

Check for race conditions

>> compute-sanitizer --tool racecheck ./my_cuda_app app_command_line_options

```
32
33  __global__ void raceConditionKernel(int *array, int N) {
34      int idx = blockIdx.x * blockDim.x + threadIdx.x;
35      if (idx < N) {
36          array[0] += idx;
37      }
38  }
39
40  int main() {
41      // Allocate device memory
42      int N = 100;
43      int *d_array;
44      cudaMalloc((void**) &d_array, N * sizeof(int));
45
46      ...
47      // Run CUDA kernel
48      int threadsPerBlock = 256;
49      int blocksPerGrid = N/threadsPerBlock + 1;
50      raceConditionKernel <<< blocksPerGrid, threadsPerBlock >>> (d_array, N);
51      ...
52  }
```

Multiple threads access simultaneusly to array[0] position

d_array allocate 100 integer elements in the global memory

kernel with a grid of 256 threads in 1 block

Check for race conditions

Racecheck output message

```
========= WARNING: Potential RAW race condition on address 0x10000000 at kernel my_kernel
=========     Write at 0x00000120 in my_kernel by thread (1, 0, 0) in block (0, 0, 0)
=========     Read at 0x00000130 in my_kernel by thread (2, 0, 0) in block (0, 0, 0)
=========     Address 0x10000000 is shared memory
=========     Saved host backtrace up to driver entry point at kernel launch time
=========     Host Frame: [0x00007f8e1a2b3e45]
=========                 in /lib/x86_64-linux-gnu/libc.so.6
=========     Host Frame: [0x00007f8e1a2b3e45]
=========                 in /lib/x86_64-linux-gnu/libc.so.6
=========
========= RACE SUMMARY: 1 race condition detected
```

Kernel

Location

- Thread that wrote to the shared memory.
- Thread reads from the share memory.

Address

- The shared memory address involves.

Call stack

Check for race conditions

Synchronization primitives

- Add syncronitazion barriers: __syncthreads()
- Use atomic operations for fine synchronization: atomicAdd, atomicExch, ...

Re-designing the kernel

- Avoid algorithms where multiple threads write to the same memory location.
- Modify the memory allocation to avoid shared memory use.

Check for synchronization conditions

Synchronization errors occur when threads in a CUDA kernel do not properly synchronize their access to shared or global memory, leading to race conditions and deadlocks.

Execution

```
>> compute-sanitizer --tool synccheck  ./my_cuda_app  app_command_line_options
```

Optional flags

--track-unused-memory yes	Track memory that is allocated but never used.
--log-file <file>	Save the output to a file.
--verbose	Enable verbose output for more detailed information.

Check for synchronization conditions

>> compute-sanitizer --tool synccheck ./my_cuda_app app_command_line_options

```
83
84  __global__ void warpDivergenceKernel(int *input, int *output, int N) {
85      int idx = blockIdx.x * blockDim.x + threadIdx.x;
86      if (idx < N) {
87          if (threadIdx.x % 2 == 0) {
88              output[idx] = input[idx] * 2; // Even threads
89          } else {
90              output[idx] = input[idx] + 1; // Odd threads
91          }
92      }
93  }
94
95  int main() {
96      // Allocate device memory
97      int N = 100
98
99      int *d_input, *d_output;
100     cudaMalloc(&d_input, N * sizeof(int));
101     cudaMalloc(&d_output, N * sizeof(int));
102
103     ...
104     // Run CUDA kernel
105     int threadsPerBlock = 32; // 1 warp per block
106     int blocksPerGrid = N/threadsPerBlock + 1;
107     warpDivergenceKernel <<<blocksPerGrid, threadsPerBlock>>> (d_input, d_output, N);
108     ...
109 }
```

Warp divergence: threads in the same warp take different paths.

kernel with a grid of 32 threads (1 warp) blocks

Check for synchronization conditions

Synccheck output message

```
========= Barrier divergence detected at 0x00000120 in sync_kernel
=========     by thread (1, 0, 0) in block (0, 0, 0)
=========     Saved host backtrace up to driver entry point at kernel launch time
=========     Host Frame: [0x00007f8e1a2b3e45]
=========                 in /lib/x86_64-linux-gnu/libc.so.6
=========     Host Frame: [0x00007f8e1a2b3e45]
=========                 in /lib/x86_64-linux-gnu/libc.so.6
=========
========= ERROR SUMMARY: 1 error
```

Error type

- Barrier divergence
- Warp divergence

Adress

Kernel

Location

Thread and block leading to the problem.

Call stack

Check for synchronization conditions

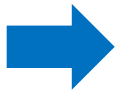
Using synchronization barriers properly

- Use __syncthreads() to synchronize all threads in a block
- Use atomic operations for fine synchronization: atomicAdd, atomicExch, ...

Avoiding warp divergence

- Minimize conditional statements that cause threads in a warp to follow different execution paths.

```
__global__ void warpDivergenceKernel(int *input, int *output, int N) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if (idx < N) {
        // Warp divergence: threads in the same warp take different paths
        if (threadIdx.x % 2 == 0) {
            output[idx] = input[idx] * 2; // Even threads
        } else {
            output[idx] = input[idx] + 1; // Odd threads
        }
    }
}
```



```
__global__ void noWarpDivergenceKernel(int *input, int *output, int N) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if (idx < N) {
        // No warp divergence: all threads in a warp take the same path
        output[idx] = (idx % 2 == 0) ? (input[idx] * 2) : (input[idx] + 1);
    }
}
```

- Avoid situations where threads need to synchronize in complex ways.

Module content

1. Optimizing tools
2. Debugging code with Compute Sanitizer
3. Profiling code with Nsight System

Profiling with Nsight System

Compilation flags

- Include –g: Add information for the host code.
- Include –G: Add information for the device code.
- Include -03: High-level compiler optimizations

Running Nsight System

>> nsys-ui

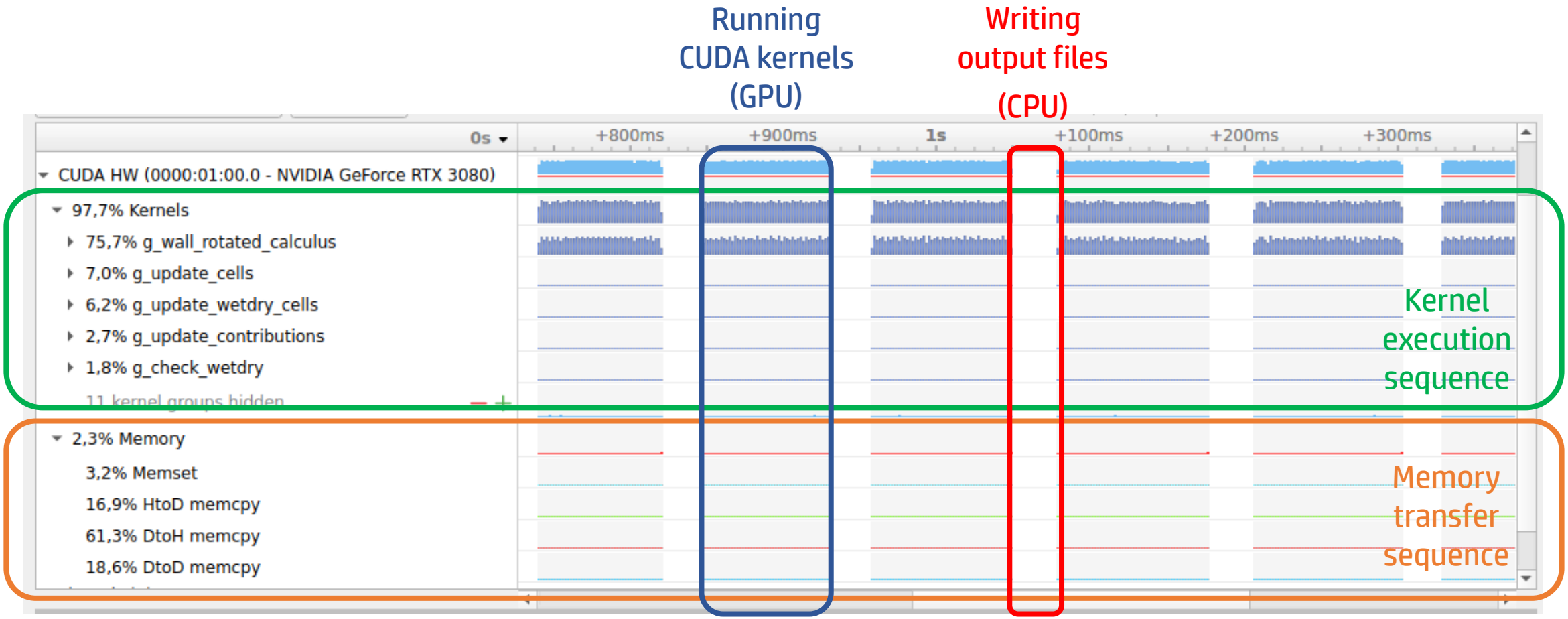
Timeline View

- Execution of the CPU and GPU activities over time.
- Sequential calls to CUDA kernels, memory transfers and CUDA API calls.

GPU activity statistics

- Detailed metrics of the kernel execution time, the block/grid dimensions, and memory usage.
- Time spent on data transfers between the host and device.

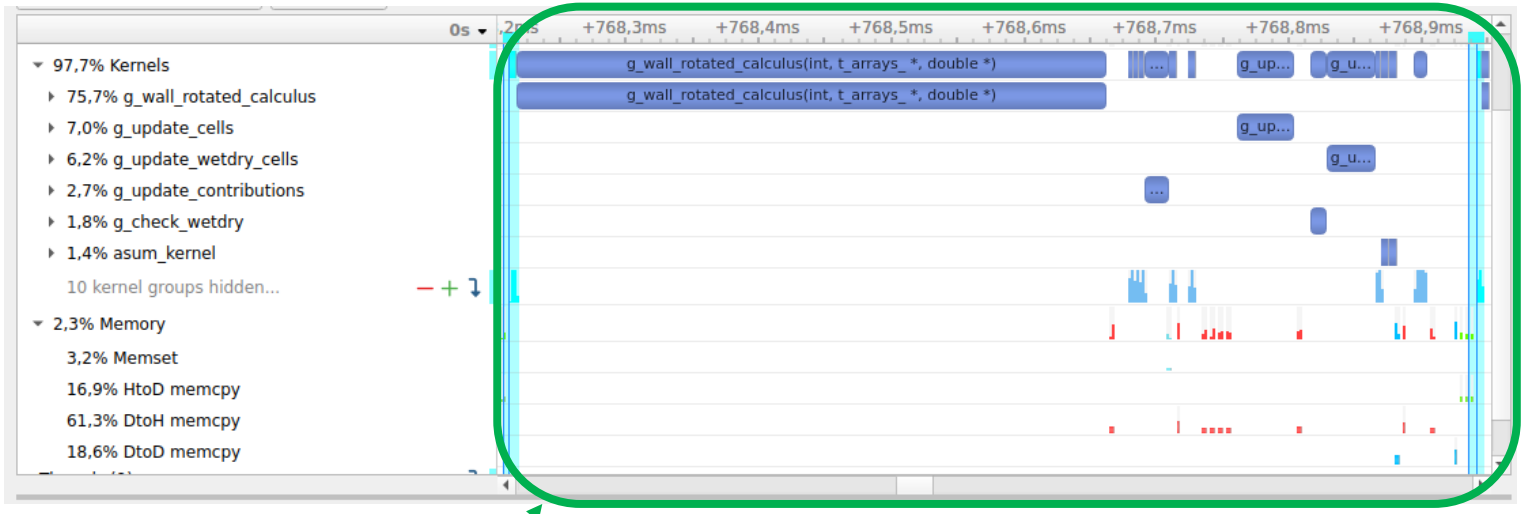
Timeline view

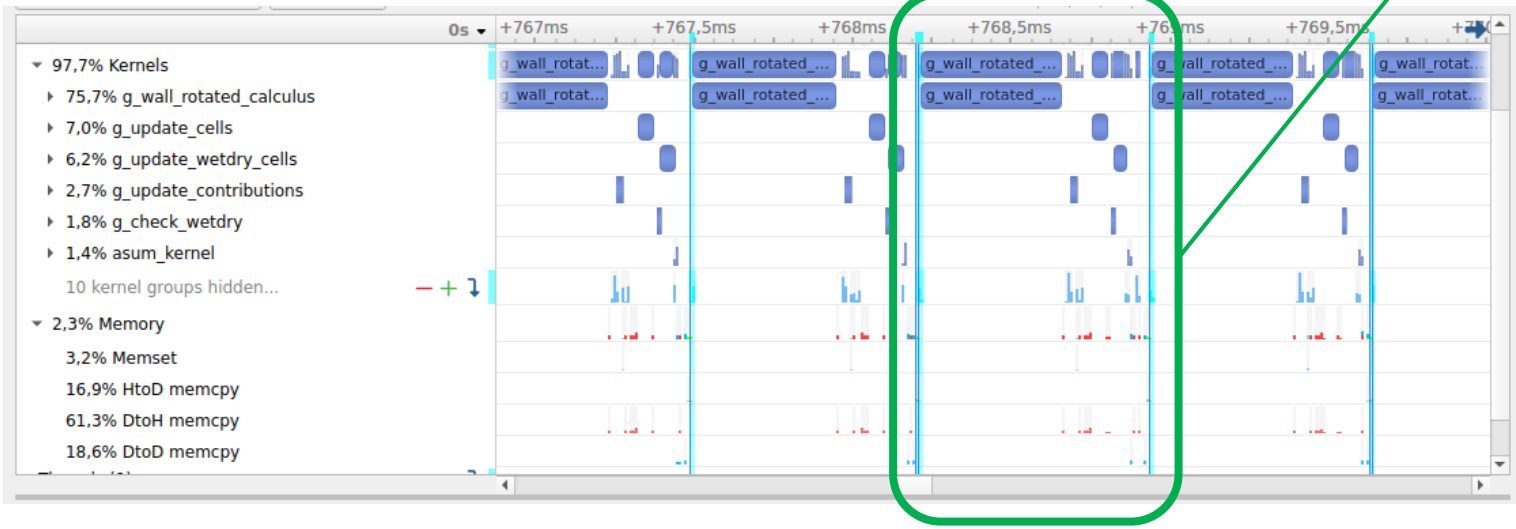


Timeline view

Time step analysis

Fine-grained analysis of the kernel launches and the memory transfer during each time step.





Statistics and summary

Kernel execution

Relative/Total execution time

Number of calls

Execution statistics

Grid and block size

CUDA kernel

CUDA API Summary
CUDA API Trace
CUDA GPU Kernel Summary
CUDA GPU Kernel/Grid/Block Summary
CUDA GPU MemOps Summary (by Size)
CUDA GPU MemOps Summary (by Time)
CUDA GPU Summary (Kernels/MemOps)
CUDA GPU Trace
CUDA Kernel Launch & Exec Time Summary
CUDA Kernel Launch & Exec Time Trace
CUDA Summary (API/Kernels/MemOps)
DX11 PIX Range Summary
DX12 GPU Command List PIX Ranges Summary
DX12 PIX Range Summary
MPI Event Trace
NVTX GPU Projection Summary
NVTX GPU Projection Trace
NVTX Push/Pop Range Summary
NVTX Push/Pop Range Trace
NVTX Range Kernel Summary

CLI command::
nsys stats -r cuda_gpu_kern_gb_sum /
root/.nsightsystems/Projects/Project 1/report2.sqlite

Time	Total Time	Instances	Avg	Med	Min	Max	StdDev	GridXYZ	BlockXYZ	Name
75.0%	505,598 ms	1213	416,816 µs	412,320 µs	384,160 µs	445,345 µs	9,401 µs	387 1 1	128 1 1	g_wall_rotated_calculus(int, t_arrays_ *, double *)
7.0%	46,975 ms	1213	38,726 µs	38,816 µs	35,968 µs	43,104 µs	1,379 µs	196 1 1	128 1 1	g_update_cells(int, t_arrays_ *)
6.0%	41,640 ms	1213	34,328 µs	33,760 µs	32,832 µs	36,960 µs	1,268 µs	196 1 1	128 1 1	g_update_wetdry_cells(int, t_arrays_ *)
2.0%	17,987 ms	1213	14,828 µs	14,848 µs	14,016 µs	16,320 µs	454 ns	196 1 1	128 1 1	g_update_contributions(int, t_arrays_ *)
1.0%	12,003 ms	1213	9,895 µs	9,888 µs	9,408 µs	10,816 µs	259 ns	387 1 1	128 1 1	g_check_wetdry(int, t_arrays_ *)
0.0%	5,918 ms	1214	4,874 µs	4,769 µs	4,703 µs	5,248 µs	183 ns	816 1 1	128 1 1	void asum_kernel<int, double, double>(cublasAsumParams<
0.0%	5,321 ms	1214	4,382 µs	4,416 µs	3,936 µs	4,896 µs	204 ns	196 1 1	128 1 1	g_compute_cell_mass(int, t_arrays_ *, double *)
0.0%	5,209 ms	1213	4,294 µs	4,256 µs	3,839 µs	4,800 µs	174 ns	68 1 1	128 1 1	void iamin_kernel<int, double, double, (int)128>(cublasIamii
0.0%	5,193 ms	1213	4,280 µs	4,256 µs	3,904 µs	5,568 µs	159 ns	782 1 1	128 1 1	g_initialize_delta(int, t_arrays_ *)
0.0%	5,058 ms	1213	4,169 µs	4,160 µs	3,967 µs	4,640 µs	94 ns	196 1 1	128 1 1	g_checkpos_h(int, t_arrays_ *, int *)
0.0%	4,718 ms	1213	3,889 µs	3,776 µs	3,712 µs	5,024 µs	198 ns	1 1 1	1 1 1	g_set_new_dt(t_arrays_ *)
0.0%	4,149 ms	1213	3,420 µs	3,392 µs	3,264 µs	3,713 µs	104 ns	1 1 1	128 1 1	void iamin_kernel<int, double, double, (int)128>(cublasIamii
0.0%	3,475 ms	1214	2,862 µs	2,816 µs	2,752 µs	3,073 µs	106 ns	1 1 1	128 1 1	void asum_kernel<int, double, double>(cublasAsumParams<
0.0%	3,109 ms	1213	2,563 µs	2,528 µs	2,464 µs	2,784 µs	88 ns	1 1 1	1 1 1	g_get_dtmin(t_arrays_ *, double *, int *)

Statistics and summary

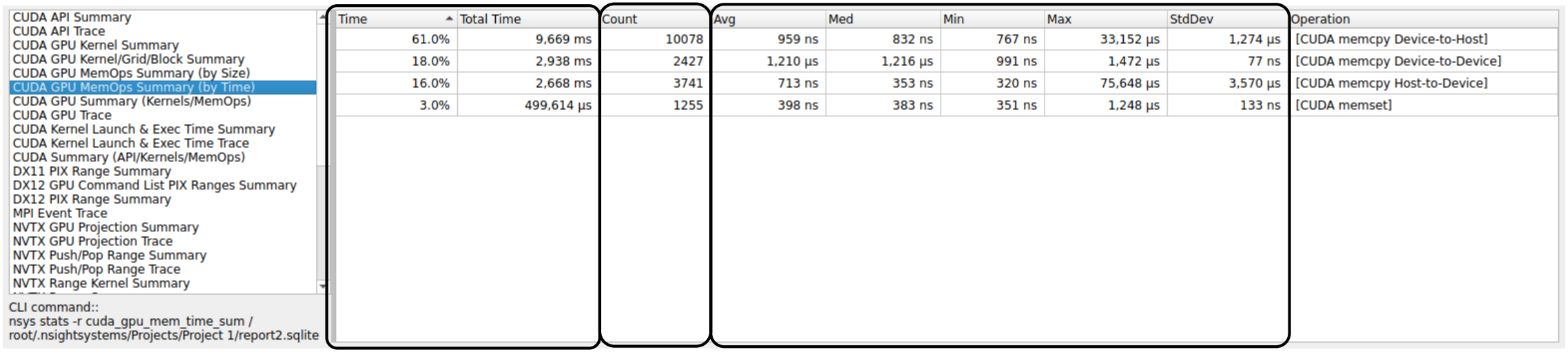
Memory operations

Relative/Total execution time

Number of calls

Execution statistics

Memory operation type



CUDA API Summary
CUDA API Trace
CUDA GPU Kernel Summary
CUDA GPU Kernel/Grid/Block Summary
CUDA GPU MemOps Summary (by Size)
CUDA GPU MemOps Summary (by Time)
CUDA GPU Summary (Kernels/MemOps)
CUDA GPU Trace
CUDA Kernel Launch & Exec Time Summary
CUDA Kernel Launch & Exec Time Trace
CUDA Summary (API/Kernels/MemOps)
DX11 PIX Range Summary
DX12 GPU Command List PIX Ranges Summary
DX12 PIX Range Summary
MPI Event Trace
NVTX GPU Projection Summary
NVTX GPU Projection Trace
NVTX Push/Pop Range Summary
NVTX Push/Pop Range Trace
NVTX Range Kernel Summary

CLI command::
nsys stats -r cuda_gpu_mem_time_sum /
root/.nsightsystems/Projects/Project 1/report2.sqlite

Time	Total Time	Count	Avg	Med	Min	Max	StdDev	Operation
61.0%	9,669 ms	10078	959 ns	832 ns	767 ns	33,152 µs	1,274 µs	[CUDA memcpy Device-to-Host]
18.0%	2,938 ms	2427	1,210 µs	1,216 µs	991 ns	1,472 µs	77 ns	[CUDA memcpy Device-to-Device]
16.0%	2,668 ms	3741	713 ns	353 ns	320 ns	75,648 µs	3,570 µs	[CUDA memcpy Host-to-Device]
3.0%	499,614 µs	1255	398 ns	383 ns	351 ns	1,248 µs	133 ns	[CUDA memset]

References

Wu, W. (2007). Computational River Dynamics. NetLibrary, Inc, CRC Press .

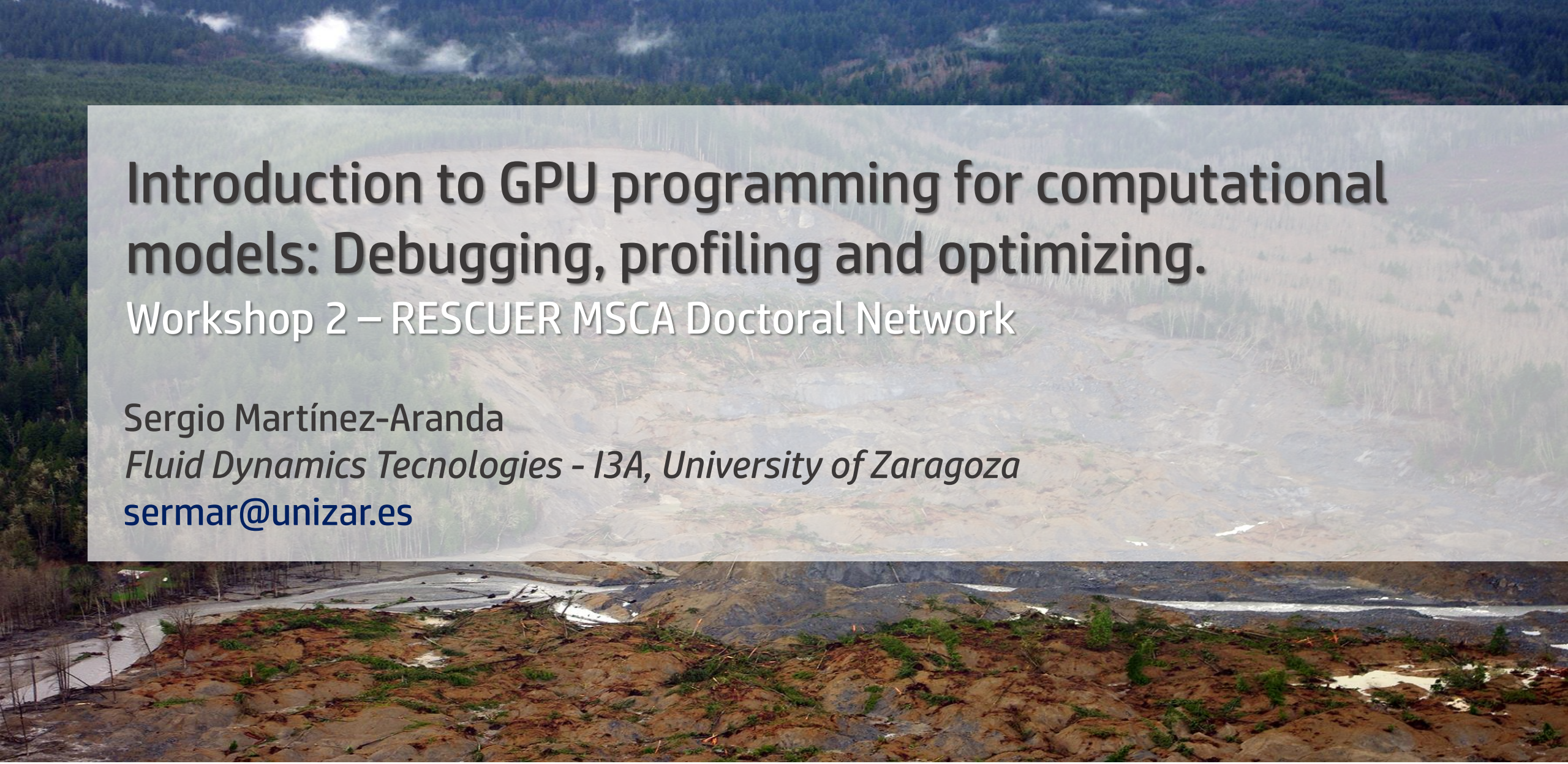
Fernández-Pato J., et al., 2025. Acceleration of pipeline analysis for irrigation networks through parallelisation in Graphic Processing Units. Biosystems Engineering 252, pp. 1-14.

An, S., and Seo, SC., 2020. Efficient Parallel Implementations of LWE-Based Post-Quantum Cryptosystems on Graphics Processing Units. Mathematics 8(10) pp. 1781.

NVIDIA developers, 2025. https://developer.nvidia.com/

CUDA Toolkit Documentation v12.8 Update 1, 2025. https://docs.nvidia.com/cuda/

Aragón Institute of Engineering Research I3A, 2025. https://i3a.unizar.es/es



Introduction to GPU programming for computational models: Debugging, profiling and optimizing.

Workshop 2 – RESCUER MSCA Doctoral Network

Sergio Martínez-Aranda
Fluid Dynamics Tecnologies - I3A, University of Zaragoza
sermar@unizar.es

Universidad Zaragoza
1542

Escuela de Ingeniería y Arquitectura Universidad Zaragoza

Instituto Universitario de Investigación en Ingeniería de Aragón Universidad Zaragoza