Introduction to GPU programming for computational models: Architectures, memory and management.

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. Introduction to GPU architecture
2. CUDA development platform
3. CUDA programming examples
4. Starting with *swe2d* to *swe2d_gpu* migration

GPUs and parallel computing

- In the **1990s**, companies like NVIDIA and AMD pioneered the development of dedicated graphics cards. **Accelerate rendering for computer graphics** for video games and visual simulations.
- In the **2000s**, **programmable shaders** allowed GPUs to execute custom code for rendering effects.
- In **2006**, NVIDIA launched **CUDA (Compute Unified Device Architecture)**, a programing platform that enabled developers to use GPUs for **general-purpose computing (GPGPU)**. CUDA unlocked the massive parallel processing power of GPUs, making them ideal for tasks like scientific simulations, data analysis, and machine learning.
- In the **2010s**, GPU-accelerated computational models appeared for solving geophysical, hydraulic and atmospheric flows in the Earth surface. Computing simultaneously in multiple GPU devices (**High-Performance-Computing or HPC**) allows to solve very-large-scale problems with affordable simulation times.
- **Nowadays**, GPUs are the **cutting-edge technology for scientist computing**, AI research, cryptocurrency mining, data centers, autonomous vehicles, and even quantum computing simulations. **Cloud computing** has further enabled GPU access to researchers and developers without owning physical hardware.

(NVIDIA GeForce 256, 1999)

(NVIDIA GeForce RTX 5090, 2025)

GPU architecture vs CPU Architecture

CPU (Central Processing Unit)

- Designed for general-purpose computing and **sequential task execution**.
- Focuses on **low-latency processing** and optimizing for single-threaded performance.
- Features **fewer, more powerful cores** with complex control logic and large caches to handle diverse tasks efficiently.

(AMD Ryzen Threadripper PRO 7995WX, 2023)

GPU (Graphics Processing Unit)

- Specialized for **parallel processing** and high-throughput computations.
- Optimized for handling **thousands of lightweight threads simultaneously**.
- Comprises **thousands of smaller, simpler cores** (CUDA cores in NVIDIA GPUs or Stream Processors in AMD GPUs) designed for specific tasks like rendering graphics or performing matrix operations.

(NVIDIA GeForce RTX 5090, 2025)

GPU architecture vs CPU Architecture

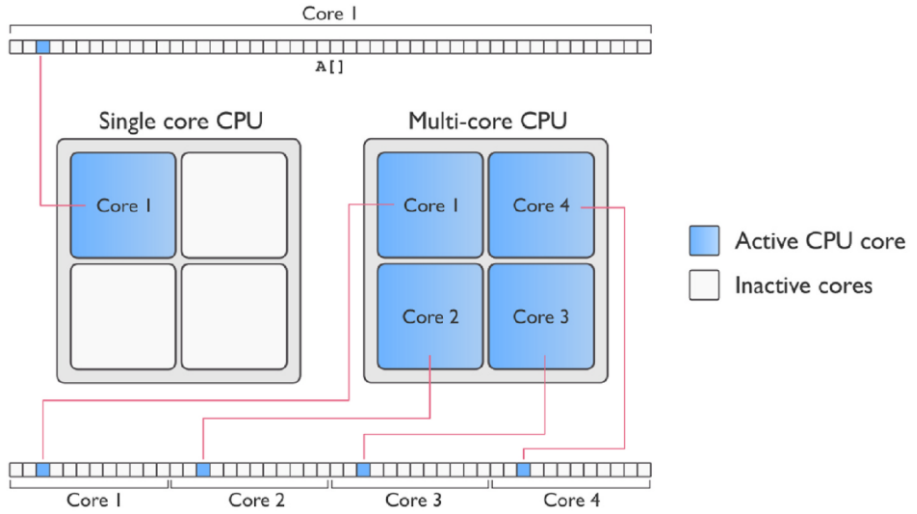
CPU processors

Single-thread execution

- Each task waits for the completion of the previous task, i.e. sequential processing.

Simultaneous Multi-Threading (SMT)

- Physical core has a full set of execution units and caches
- Physical cores are divided into two logical cores for the operating system, which shares the core resources.
- Tasks are divided and executed simultaneously across multiple threads in a multi-core CPU node.



(Reproduced from Fernández-Pato, 2025)

GPU architecture vs CPU Architecture

GPU processors

Thread

- The thread is the smallest unit of execution.
- Thousands of threads can run simultaneously on the GPU kernels.

Block

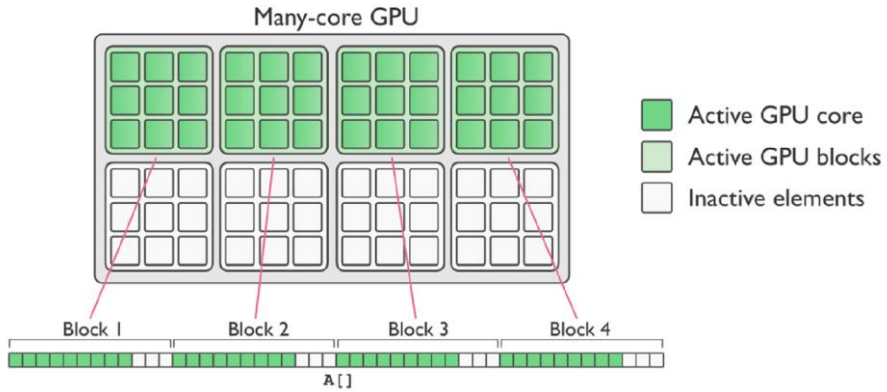
- Threads are grouped into blocks.
- Collections of threads that can synchronize and communicate using shared memory.

Grid

- The problem is divided into a grid, i.e. collection of blocks that execute the same kernel function.
- Threads in a grid can access to the same (global) memory.

Specialized Units

- Dedicated units for specific tasks, such as RT cores for ray tracing or Tensor cores for AI acceleration.



(Reproduced from Fernández-Pato, 2025)

GPU architecture vs CPU Architecture

CPU memory hierarchy

RAM memory

- Large memory shared across all the CPU cores.
- Optimized for random access patterns and complex control flows.
- Primary working memory for I/O communications.

GPU memory hierarchy

Global Memory

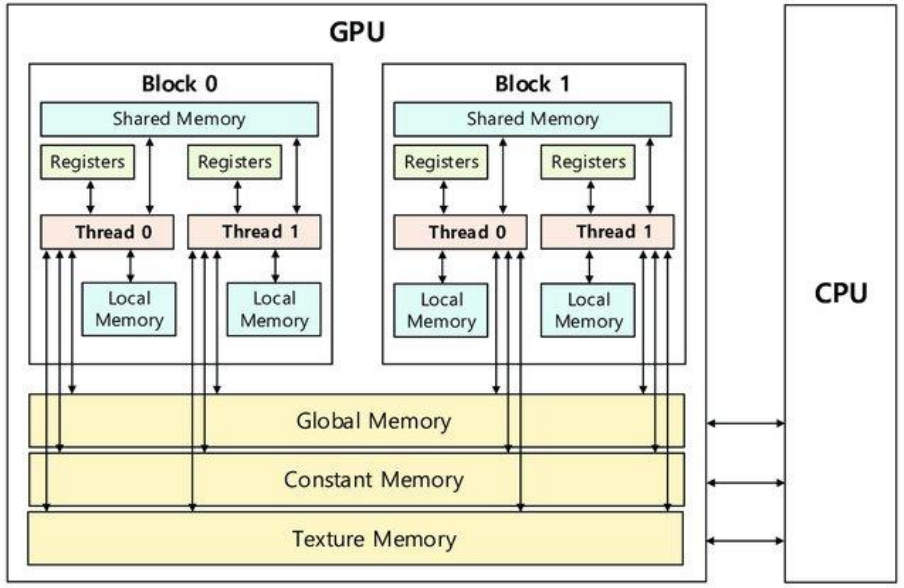
- High-capacity memory but with higher latency.
- Accessible by all the threads in the grid.

Shared Memory

- Faster, on-chip memory shared among the treads of each blocks.
- Limited to 48KB per block in most of GPUs.

Registers

- Ultra-fast memory dedicated to individual threads.
- Limited to 255 registers per thread.



(Reproduced from An and Seo, 2020)

GPU architecture vs CPU Architecture

CPU parallelism

Single-thread execution

- Each task waits for the completion of the previous task, i.e. sequential processing.

Thread-based parallelism (OpenMP)

- Tasks are divided and executed simultaneously across multiple threads in a multi-core CPU node.
- Shared memory, i.e. all threads can access the same memory.
- Easy to implement with compiler directives, such as #pragma omp.

Process-based parallelism (Message Passing Interface)

- The problem can be divided into smaller sub-problems and they are solved simultaneously in independent CPU nodes.
- Distributed memory, i.e. each sub-problem accesses its own memory.
- Scalable performance in clusters but it requires explicit message passing between sub-problems.

GPU architecture vs CPU Architecture

CPU parallelism

Single-thread execution
Thread-based parallelism (OpenMP)
Process-based parallelism (Message Passing Interface)

GPU parallelism

SIMT (Single Instruction, Multiple Threads):
- Blocks are scheduled in Streaming Multiprocessors (SMs).
- Threads run in warps, i.e. a group of 32 threads (NVIDIA GPUs) that execute together lockstep.
- Multiple threads execute the same instruction on different data points simultaneously.

Thread divergence:
- Threads can diverge, i.e. follow different execution paths with different execution times.
- This reduces efficiency.

Memory coalescing:
- Threads access to consecutive memory locations.

GPU development platforms

There are several GPU development platforms available, each with its own strengths and target use cases.

- **CUDA:** The most widely used platform, especially in for scientific computing and AI.
- **OpenCL:** Popular for cross-platform and heterogeneous computing.
- **HIP:** Gaining traction for AMD GPUs and CUDA compatibility.
- **SYCL:** Emerging as a modern C++ alternative for heterogeneous computing.

CUDA platform

https://developer.nvidia.com/

- Developer: NVIDIA
- Platform Support: Primarily NVIDIA GPUs
- Ease of Use: More user-friendly, extensive documentation, and mature ecosystem
- Performance: Optimized for NVIDIA hardware, often faster on NVIDIA GPUs
- API and Ecosystem: Rich ecosystem with libraries (cuBLAS, cuDNN, etc.) and tools (Nsight)
- Portability: Limited to NVIDIA GPUs
- Adoption: Widely used in scientific computing, AI and deep learning.
- Programming Model: SIMT (Single Instruction, Multiple Threads)
- Community Support: Large community, strong industry support

Module content

1. Introduction to GPU architecture
2. CUDA development platform
3. CUDA programming examples
4. Starting with *swe2d* to *swe2d_gpu* migration

CUDA/C++ platforms

The NVIDIA® CUDA® Toolkit provides a comprehensive development environment for C/C++ developers building GPU-accelerated applications.

NVIDIA driver: Enable the operating system and applications to communicate effectively with NVIDIA GPUs.

Compiler: The NVIDIA CUDA Compiler (nvcc) which compiles CUDA/C++ code into applications executable on GPUs.

Runtime: The CUDA API provides functions to manage devices, memory, and execute kernels.

Developing tools: Tools for debugging, profiling, and optimizing CUDA applications.

- nvidia-smi
- compute-sanitizer

Libraries: A set of highly optimized libraries for various domains.

- cuBLAS library is an implementation of Basic Linear Algebra Subprograms (BLAS) on the NVIDIA CUDA runtime.
- cuSPARSE library contains subroutines for handling sparse matrices implemented on the NVIDIA CUDA runtime.

Additional tools: nvtop, VScode, etc.

CUDA/C++ code workflow

GPU-accelerated apps include **parallelized CUDA kernels** running into a **sequential workflow** running in the CPU core.

```
23
24  // Driven fucntion exected in CPU
25  int main()
26  {
27      ...
28      RAM memory allocation and initialization
29      ...
30      GPU selection and setup
31      ...
32      CUDA global memory allocation
33      ...
34      RAM --> CUDA data transfer
35      ...
36      // Kernel invocation with multi-threads
37      say_hi <<<numBlocks, threadsPerBlock>>> ();
38      ...
39      CUDA --> RAM memory transfer
40      ...
41      Write output files
42      ...
43  }
44
```

Sequential workflow

CUDA kernel parallelized in the GPU device.

CUDA/C++ code workflow

GPU-accelerated apps include **parallelized CUDA kernels** running into a **main driven function** running in the CPU core.

```
23
24  // Driven fucntion exected in CPU
25  int main()
26  {
27      ...
28      RAM memory allocation and initialization
29      ...
30      GPU selection and setup
31      ...
32      CUDA global memory allocation
33      ...
34      RAM --> CUDA data transfer
35      ...
36      // Kernel invocation with multi-threads
37      say_hi <<<numBlocks, threadsPerBlock>>> ();
38      ...
39      CUDA --> RAM memory transfer
40      ...
41      Write output files
42      ...
43  }
44
```

Sequential workflow

RAM memory allocation is required.

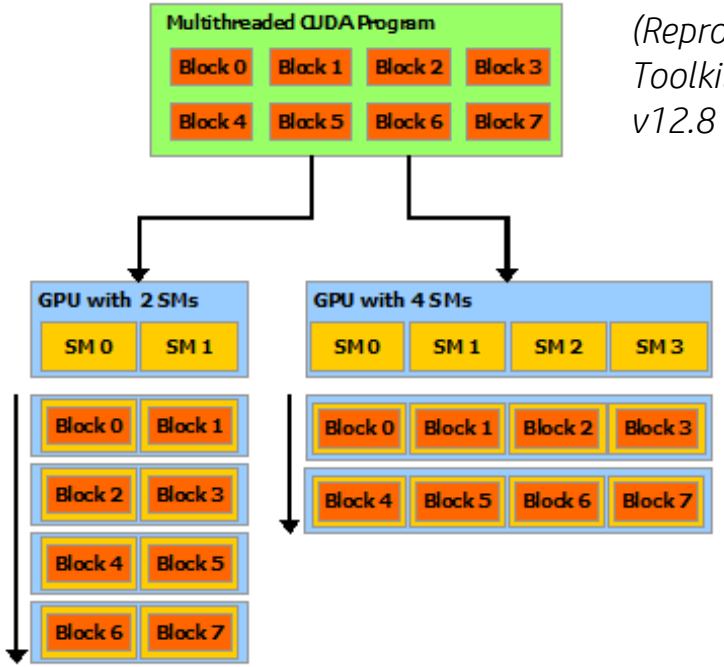
CUDA memory structure is the key factor to achieve high performance in the CUDA kernel execution.

Barriers for thread synchronization and memory transfers RAM-GPU represent critical points for the code performance.

CUDA/C++ parallel kernel

- The multi-threaded kernel is partitioned into **thread blocks**, that execute independently from each other.
- Thread blocks are scheduled in an array of **Streaming Multiprocessors** (SMs).
- GPUs with more multiprocessors execute the program in less time than a GPU with fewer multiprocessors.

```
2
3   // Kernel definition
4   __global__ void say_hi()
5   {
6       int idx = (blockIdx.x * blockDim.x) + threadIdx.x;
7       printf("Hi from Thread %d\n", idx);
8   }
9
10  // Driven fucntion exected in CPU
11  int main()
12  {
13      ...
14      // Kernel invocation with multi-threads
15      say_hi <<<numBlocks, threadsPerBlock>>> ();
16      ...
17  }
18
```



(Reproduced from CUDA Toolkit Documentation v12.8 Update 1, 2025)

CUDA/C++ compilation and execution

- CUDA/C++ code is written in app_code.cu files, including the header:

  #include <cuda_runtime.h>

- NVCC compiler is used to create executable files:

  nvcc -arch=sm_XX -O3 -G -o exe_app app_code.cu –lXXXXX

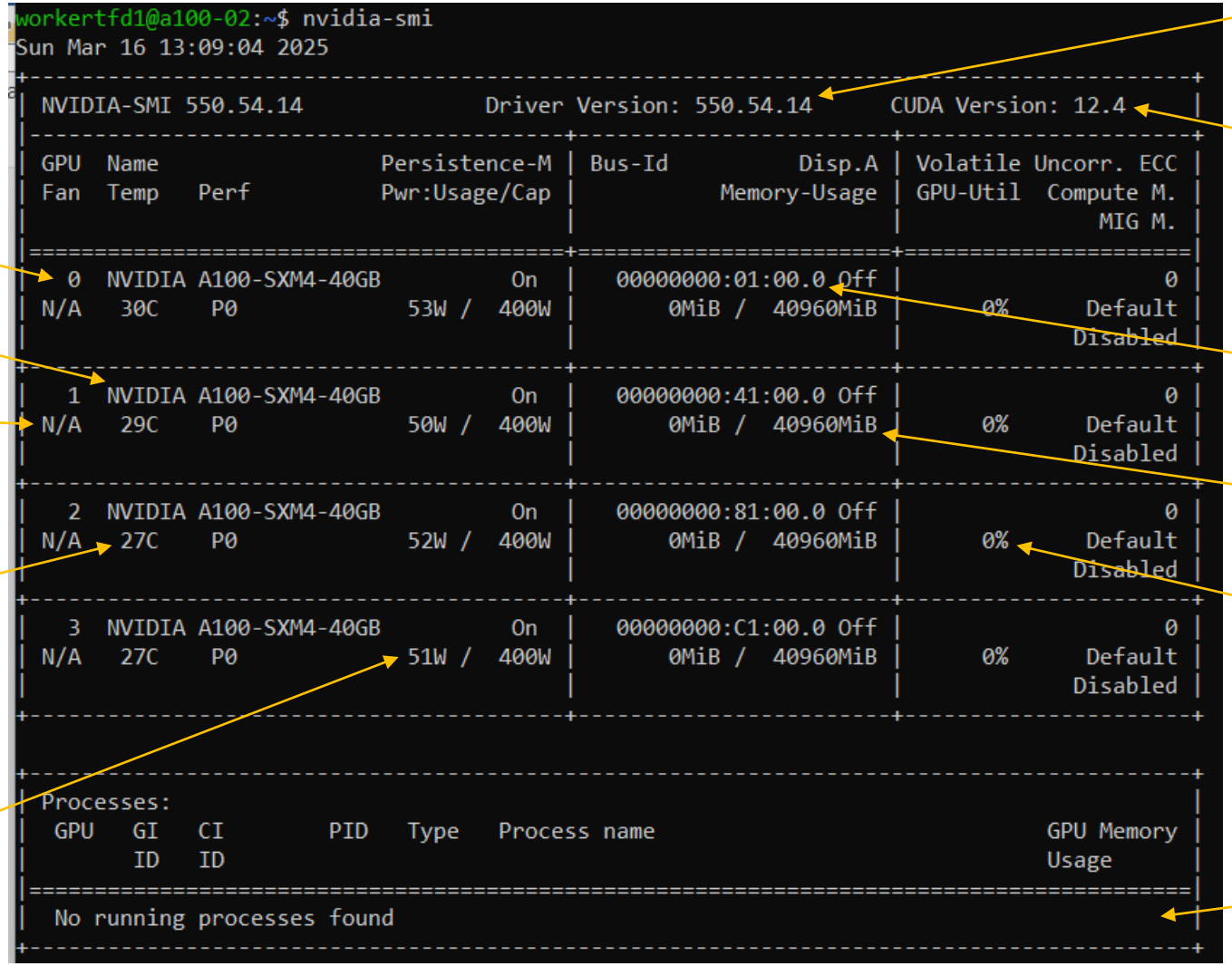
- Additional flags for nvcc compiler:
  -arch=sm_XX to specify the compute capability of the GPU device
  -G to include memory check information for debugging tools.
  -O3 to enable high-level optimizations
  -lXXXXX to include external libraries, such as –lcudart or -lcublas.

- The compiled application runs with the command:

  ./exe_app

- nvidia-smi command provides performance data of the GPU during the execution of the application.

CUDA/C++ compilation and execution



Module content

1. Introduction to GPU architecture
2. CUDA development platform
3. CUDA programming examples
4. Starting with *swe2d* to *swe2d_gpu* migration

Working in the HERMES supercomputing cluster

HERMES supercomputing cluster is the scientific infrastructure dedicated for researchers at the Institute of Engineering Research (I3A), University of Zaragoza. HERMES has over 3,260 CPU cores, with 5,920 parallel processing threads and 26 TB of RAM. HERMES features 251,392 CUDA cores (including NVIDIA A100 GPUs) and 720 GB of dedicated memory.

Instituto Universitario de Investigación
en Ingeniería de Aragón
Universidad Zaragoza

CUDA hands-on session during W2
Dedicated computation node with 4 x NVIDIA A100 devices.
20 working user profiles: 4 groups of 5 users each.

- Remote access by Ubuntu terminal:
  ssh workertfdX@155.210.134.18 -p 4001

- **User:** workertfdX **with X=2,…,20**
  Note: workertfd1 **is reserved for professors.**

- **Password:** Crumb-Submitter-Thread-Bling

Main step in CUDA/C++ code workflow

Sequential workflow

GPU selection and setup

CUDA memory allocation

CPU-GPU data transfer

CUDA kernel execution

GPU selection and setup

```
48
49  int numGPUs = 0;
50  cudaGetDeviceCount(&numGPUs);
51
```

Get the number of available GPUs with computing capabilities.

```
51
52   int gpu_id = 0;
53   cudaDeviceProp props;
54   cudaGetDeviceProperties(&props, gpu_id);
55
```

Get the specific properties of the selected GPU device. The input gpu_id is type integer. The output is class-type cudaDeviceProp.

GPU selection and setup

```
48
49  int numGPUs = 0;
50  cudaGetDeviceCount(&numGPUs);
51
```

Get the number of available GPUs with computing capabilities.

```
51
52   int gpu_id = 0;
53   cudaDeviceProp props;
54   cudaGetDeviceProperties(&props, gpu_id);
55
```

Get the specific properties of the selected GPU device. The input gpu_id is type integer. The output is class-type cudaDeviceProp.

```
55
56   int selectedGPU = 0;
57   cudaSetDevice(selectedGPU);
58
```

Select a specific GPU device to run the parallel CUDA kernels.

```
58
59  size_t freeMemory, totalMemory;
60  cudaMemGetInfo(&freeMemory, &totalMemory);
61
```

Get the total available memory and the free memory for a specific GPU device.

```
61
62  cudaDeviceSynchronize();
63
```

It ensures all the CUDA tasks on the GPU device have been completed before the host (CPU) continues execution.

Main step in CUDA/C++ code workflow

Sequential workflow

GPU selection and setup

CUDA memory allocation

CPU-GPU data transfer

CUDA kernel execution

CUDA memory allocation: Global memory

```
66
67  int main() {
68
69      // Host memory
70      size_t size = N*sizeof(double);
71      double *h_A;
72      h_A = (double*) malloc(size);
73
74      ...
75      double *d_A;
76      cudaMalloc((void**) &d_A, size);
77
78
79      int threadsPerBlock = 256;
80      int blocksPerGrid = N/threadsPerBlock+1;
81
82      memoryCheck
83          <<<blocksPerGrid, threadsPerBlock>>>
84          (N, size, d_A);
85
86
87      cudaFree(d_A);
88      ...
89  }
90
```

Dynamic allocation of computation arrays in the host memory

Dynamic allocation of computation arrays in the GPU global memory

Parallelized kernel executed in the CUDA grid

Free GPU global memory

CUDA memory allocation: Thread register memory

```
89
90  __global__ void registerMemoryAllocate(int N, size_t size, double *d_A) {
91
92      double rA;
93
94      int idx = ( blockIdx.x * blockDim.x ) + threadIdx.x;
95      if (idx < N) {
96          rA = (double)(idx);
97          d_A[idx] = rA;
98      }
99  }
100
101 int main() {
102     ...
103     int threadsPerBlock = 256;
104     int blocksPerGrid = N/threadsPerBlock+1;
105
106     registerMemoryAllocate
107         <<<blocksPerGrid, threadsPerBlock>>>
108         (N, size, d_A);
109     ...
110 }
111
```

Static register memory allocation

Parallel computation of the register at each thread and output to the global memory array.

Parallelized kernel executed in the CUDA grid

CUDA memory allocation: Block shared memory

```
118
119 __global__ void sharedMemoryAllocate(int N, size_t size, double *d_A) {
120
121     int ithread = threadIdx.x;
122     int idx = ( blockIdx.x * blockDim.x ) + threadIdx.x;
123
124     extern __shared__ double shared_A[];
125     if (idx < N) {
126         shared_A[ithread] = (double)(idx);
127     }
128     __syncthreads();
129
130     if (idx < N) {
131         d_A[idx] = shared_A[ithread];
132     }
133 }
134
135 int main() {
136     ...
137     int threadsPerBlock = 256;
138     int blocksPerGrid = N/threadsPerBlock+1;
139     size_t requiredSharedMemory = threadsPerBlock*sizeof(double);
140
141     sharedMemoryAllocate
142         <<<blocksPerGrid, threadsPerBlock, requiredSharedMemory>>>
143         (N, size, d_A);
144     ...
145 }
146
```

Dynamical allocation of the shared memory. The idx thread computes to the ithread position in the shared array of the blockIdx.x block. __syncthreads() ensures that all threads have finished writing the shared array.

Parallel output to the global memory array.

Parallelized kernel executed in the CUDA grid. The size of the shared memory should be provided in the CUDA grid definition.

Main step in CUDA/C++ code workflow

Sequential workflow

GPU selection and setup

CUDA memory allocation

CPU-GPU data transfer

CUDA kernel execution

CPU-GPU data transfer

```
153
154  int main() {
155
156      size_t size = N*sizeof(double); // arrays size
157
158      double *h_A = (double*) malloc(size); // host array
159
160      double *d_A;
161      cudaMalloc((void**) &d_A, size); // device array
162
163      ...
164      cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
165
166
167      vectorAdd <<<blocksPerGrid, threadsPerBlock>>> (N, d_A, d_B);
168
169
170      cudaMemcpy(h_B, d_B, size, cudaMemcpyDeviceToHost);
171      ...
172  }
173
```

Copy array from host to global memory in the device

Copy array from device global memory back to host memory

CPU-GPU data transfer

```
153
154 int main() {
155
156     size_t size = N*sizeof(double); // arrays size
157
158     double *h_A = (double*) malloc(size); // host array
159
160     double *d_A;
161     cudaMalloc((void**) &d_A, size); // device array
162
163     ...
164     cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
165
166
167     vectorAdd <<<blocksPerGrid, threadsPerBlock>>> (N, d_A, d_B);
168
169
170     cudaMemcpy(h_B, d_B, size, cudaMemcpyDeviceToHost);
171     ...
172 }
173
```

Copy array from host to global memory in the device

Copy array from device global memory back to host memory

```
149
150 cudaMemset(d_B, 0, size);
151 cudaMemset(d_D, 0xFF, N*sizeof(int)); //integer array
152
```

Set global memory to specific values (byte level).

Main step in CUDA/C++ code workflow

Sequential workflow

GPU selection and setup

CUDA memory allocation

CPU-GPU data transfer

CUDA kernel execution

CUDA kernel execution: Grid setup

```
177
178 __global__ void arrayComputation(int N, double *A) {
179
180     int idx = ( blockIdx.x * blockDim.x ) + threadIdx.x;
181     if (idx < N) {
182         A[idx] = ...
183     }
184 }
185
186 int main() {
187
188     int N = 1000;
189
190     double *d_A;
191     cudaMalloc((void**) &d_A, N*sizeof(double)); // device array
192
193     ...
194     arrayComputation <<<1, N>>> (N, d_A);
195
196
197
198     int threadsPerBlock = 256;
199     int blocksPerGrid = N/threadsPerBlock + 1;
200
201     arrayComputation <<<blocksPerGrid, threadsPerBlock>>> (N, d_A);
202     ...
203 }
204
```

__global__ function executed in the CUDA grid.

Number of tasks to execute in the parallel kernel, i.e. number of threads in the CUDA grid.

Single block execution.

CUDA kernel execution: Grid setup

```
177
178 __global__ void arrayComputation(int N, double *A) {
179
180     int idx = ( blockIdx.x * blockDim.x ) + threadIdx.x;
181     if (idx < N) {
182         A[idx] = ...
183     }
184 }
185
186 int main() {
187
188     int N = 1000;
189
190     double *d_A;
191     cudaMalloc((void**) &d_A, N*sizeof(double)); // device array
192
193     ...
194     arrayComputation <<<1, N>>> (N, d_A);
195
196
197
198     int threadsPerBlock = 256;
199     int blocksPerGrid = N/threadsPerBlock + 1;
200
201     arrayComputation <<<blocksPerGrid, threadsPerBlock>>> (N, d_A);
202     ...
203 }
204
```

__global__ function executed in the CUDA grid.

Number of tasks to execute in the parallel kernel, i.e. number of threads in the CUDA grid.

Multiple block execution.

CUDA kernel execution: Device functions

```
238
239 __device__ void addDoubleData(double *A, double *B, double *C) {
240     (*C) = (*A) + (*B);
241 }
242
243 __global__ void deviceVectorAdd(int N, double *A, double *B, double *C) {
244
245     int idx = ( blockIdx.x * blockDim.x ) + threadIdx.x;
246     if (idx < N) {
247         addDoubleData( &(A[idx]), &(B[idx]), &(C[idx]) );
248     }
249 }
250
251 int main() {
252
253     int N = 1000;
254
255     double *d_A;
256     cudaMalloc((void**) &d_A, N*sizeof(double)); // device array
257
258     ...
259     int threadsPerBlock = 256;
260     int blocksPerGrid = N/threadsPerBlock + 1;
261
262     deviceVectorAdd <<<blocksPerGrid, threadsPerBlock>>> (N, d_A, d_B);
263     ...
264 }
265
```

__device__ function for algebraic in-thread operations.

addDoubleData is executed in each thread.

Multiple blocks parallel kernel

GPU kernel execution: Coalescent memory access

```
209
210  __global__ void nonCoalescentVectorAdd(int N, double *A, double *B) {
211
212      int idx = ( blockIdx.x * blockDim.x ) + threadIdx.x;
213      if (idx < N) {
214          A[idx] = B[idx*stride];
215      }
216  }
217
218  int main() {
219
220      int N = 1000;
221      int stride = 8;
222
223      double *d_A;
224      cudaMalloc((void**) &d_A, N*sizeof(double)); // device array
225
226      double *d_B;
227      cudaMalloc((void**) &d_B, stride*N*sizeof(double)); // device array
228
229      ...
230      int threadsPerBlock = 256;
231      int blocksPerGrid = N/threadsPerBlock + 1;
232
233      nonCoalescentVectorAdd <<<blocksPerGrid, threadsPerBlock>>> (N, d_A, d_B);
234      ...
235  }
236
```

Coalescent memory access to A array
Non-coalescent memory access to B array.

Coalescent array

Non-coalescent array

Multiple blocks parallel kernel

Module content

1. Introduction to GPU architecture
2. CUDA development platform
3. CUDA programming examples
4. Starting with *swe2d* to *swe2d_gpu* migration

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: Architectures, memory and management.

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