Barcelona Supercomputing Center
Centro Nacional de Supercomputación

EXCELENCIA SEVERO OCHOA

Formal methods for GPU Software Development and Verification using Ada SPARK: Experiences from Applications in Aerospace

Dr. Leonidas Kosmidis

22/10/2024

esa

AdaCore

Outline

- Introduction and motivation
- Background
  - The GPGPU programming architecture
- Project Contributions
  - Kernel code verification
  - Buffer overflow detection
- Conclusions

Project Objectives

Objectives

O1

Develop an **open source infrastructure** in which **GPU code** (device code and code communicating with the host CPU interface) **annotated with a specification language** (Ada SPARK) for properties like pre-conditions, post-conditions, loop-invariants etc. can be used with an automatic proof system to **prove the correctness of the code** and the **absence of runtime errors**.

O2

Project Objectives

Objectives

O1

O2

Identify limitations of formal methods for GPU code, so that can be addressed by tool vendors and/or the potential GPU users **understand what these tools can and cannot do**.

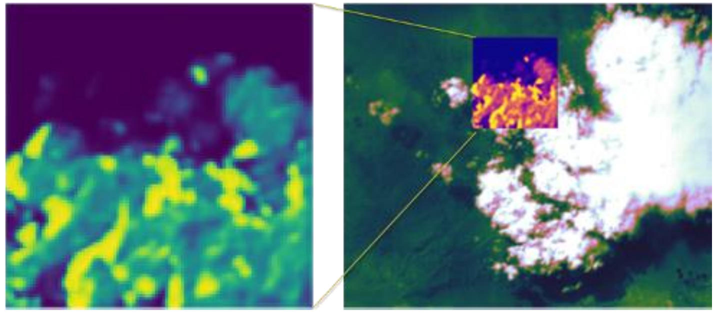
Safety Critical Systems

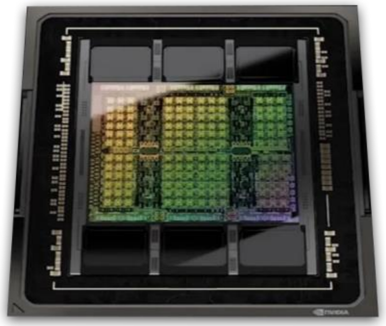
- Used in automotive, avionics and aerospace industries
- Correct execution is of paramount importance
  - Any malfunction may be dangerous
  - Designed to comply with **functional safety standards**:
    - Automotive: ISO 26262, Avionics: DO-178C, Aerospace: ECSS
- Traditionally rely on very old and simple single core processors
  - Cannot provide the performance required for new advanced functionalities

Need for Higher Performance in Aerospace Systems

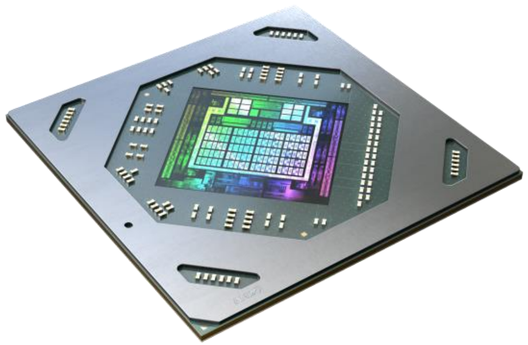
- Airbus: Automatic Taxi, Take-Off and Landing (ATTOL)
- ESA: Φ-Sat-1, OPSAT - AI and automatic cloud screening



Need for Higher Performance in Safety Critical Systems



- Legacy hardware used for safety critical systems cannot provide the required performance
- Embedded Graphics Processing Units (GPUs) are:
  - Designed to comply with safety critical functional safety standards e.g. ISO 26262
  - Very attractive candidate platforms for safety critical systems
  - GPU4S (GPU for Space) project funded by the European Space Agency at BSC has shown very promising performance results on space relevant processing



Need for Safe Programming Models

- The adoption GPU platforms in safety critical systems require not only high performance but also ease of programmability and high assurance
- According to ISO 26262, Automotive functionalities are assigned a criticality level
- Automotive Safety Integrity Level (ASIL)
- Highest Criticality software (ASIL-D) needs to comply with certain rules:
  - Restricted use of Pointers
  - No dynamic memory allocation
  - Static verification of program properties
  - Expensive testing methods like MC/DC (Modified Condition/Decision Coverage)
- Similar requirements found in other safety standards

Need for Safe Programming Models

- The adoption GPU platforms in safety critical systems require not only … high performance but also ease of programmability and high as…
- According to ISO 26262, Automotive functionalities ar… a criticality level
- Automotive Safety Integrity Level (ASIL)
- Highest Criticality software (ASIL … comply with certain rules:
  - Restricted use of Po…
  - No dynam… allocation
  - S… …cation of program properties
  - Expensive testing methods like MC/DC (Modified Condition/Decision Coverage)
- Similar requirements found in other safety standards

Violated by existing low-level parallel programming models, e.g. CUDA [1]

[1] Trompouki and Kosmidis, Brook Auto: High-Level Certification-Friendly Programming for GPU-powered Automotive Systems, DAC 2018

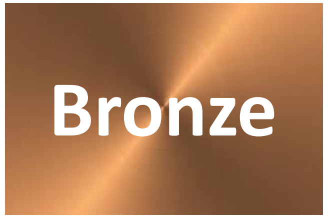
Ada

- High level programming language
- Appropriate for low-level programming like C
  - Similar performance but safer
  - Strong typing, bound checks in arrays
  - Even programming in Ada protects against common C programming mistakes
- Widely used in safety critical systems, especially in the aerospace domain as well as for security
- SPARK is a safe subset of Ada
- Can be used with Formal methods Tools
  - Prove the absence of runtime errors
  - It can formally verify program specifications

SPARK 2014 Expanding the boundaries of safe and secure programming.

Ada SPARK - Adoption Levels

- The code uses only the SPARK executable subset



- Data and flow analysis
- Prevents null dereferences, ensures proper data flow

- Guarantees absence of runtime errors (including buffer and numerical overflow, division-by-zero)

- Proves key integrity properties (e.g. pre/post-conditions)

- Full functional proofs of the requirements

Cost
Effort
Assurance

Ada SPARK - CUDA backend

- On-going collaboration between NVIDIA and AdaCore
  - NVIDIA has adopted SPARK for the development of the secure hypervisor (CPU) for their Embedded GPU platforms
- On-going development of an experimental compiler backend
  - Allows to use Ada instead of CUDA C for programming both the host and the GPU code
  - Currently under closed beta
  - AdaCore donated a license and support for any issues we discovered
- Current version of the tools do not support all language features yet (e.g. shared memory, thread synchronisation)
- The AdaCore CUDA backend is not yet integrated with SPARK tools
  - Figuring out how to use it was part of the project contributions

SPARK 2014 Expanding the boundaries of safe and secure programming.

The GPGPU Programming Architecture

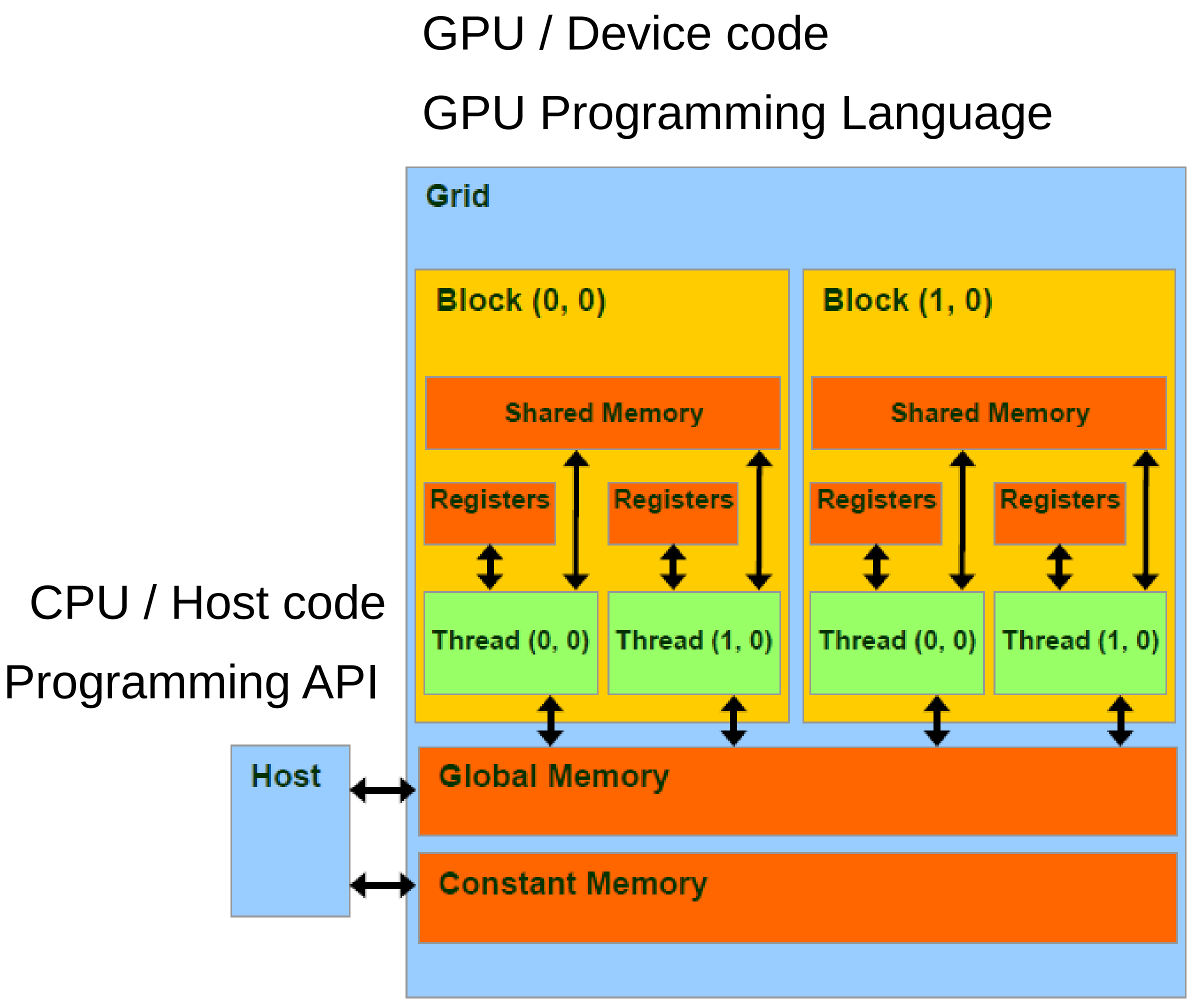


Image credit Wen-mei W. Hwu (UIUC) and David Kirk (NVIDIA)

- Massively Parallel Accelerators
- Single instruction - multiple threads programming model
- Threads organised in up-to 3D groups
  - Unique thread identifiers
- Different address space
- Memory Transfers to/from host CPU
  - Explicit memory allocation and transfers
  - Raw pointers

Example: Vector Addition Kernel in CUDA

(Device Code)

```
__global__ void vecAddKernel(int *A, int *B, int *C, int n) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if(i < n)
        C[i] = A[i] + B[i];
}
```

Example: Vector Addition Kernel Launch in CUDA

(Host Code)

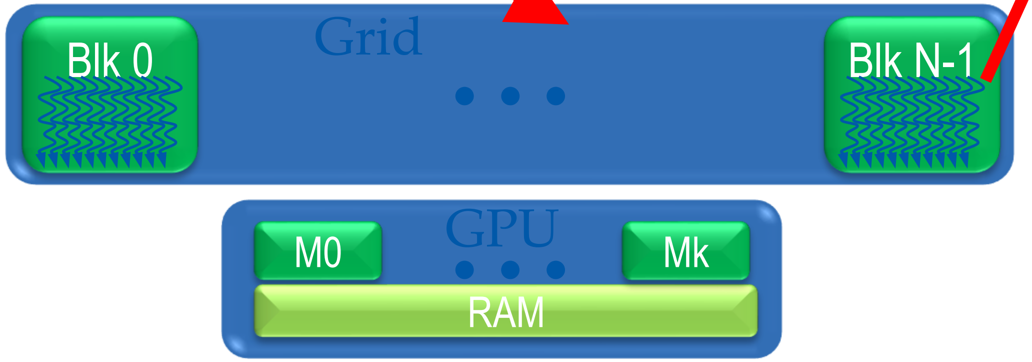
The ceiling expression makes sure that there are enough threads to cover all elements.

```
#define BLOCK_SIZE 16.0

__host__ void vecAdd(int *A, int* B, int* C, int n) {
    int *d_A, *d_B, *d_C;
    cudaMalloc((void **)&d_A, n, * sizeof(int));
    cudaMalloc((void **)&d_B, n, * sizeof(int));
    cudaMalloc((void **)&d_C, n, * sizeof(int));

    dim3 DimBlock(BLOCK_SIZE, 1, 1);
    dim3 DimGrid(ceil(n/BLOCK_SIZE), 1, 1);

    cudaMemcpy(d_A, A, n * sizeof(int), cudaMemcpyHostToDevice);
    cudaMemcpy(d_B, B, n * sizeof(int), cudaMemcpyHostToDevice);

    vecAddKernel<<<DimGrid,DimBlock>>>(d_A, d_B, d_C, n);

    cudaMemcpy(C, d_C, n * sizeof(int), cudaMemcpyDeviceToHost);

    cudaFree(d_A);
    cudaFree(d_B);
    cudaFree(d_C);
}
```

CUDA Kernel Execution in a Nutshell

```
#define BLOCK_SIZE 16.0

__host__ void vecAdd(int *A, int* B, int* C, int n) {
    (...)
    dim3 DimBlock(BLOCK_SIZE, 1, 1);
    dim3 DimGrid(ceil(n/BLOCK_SIZE), 1, 1);
    (...)
    vecAddKernel<<<DimGrid,DimBlock>>>(d_A, d_B, d_C, n);
    (...)
}
```

```
__global__ void vecAddKernel(int *A, int *B, int *C, int n) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if(i < n)
        C[i] = A[i] + B[i];
}
```



Kernel Code Verification

Guarantees from the Typing System

```
type Int_10 is new Integer range -10 .. 10;
type Int_20 is new Integer range -20 .. 20;

type Vector is array (Natural range <>) of Int_10;
type Fat_Vector is array (Natural range <>) of Int_20;
```

```
procedure VectorAdd
  (A, B : not null Vector_Device_Constant_Access;
   C    : not null Fat_Vector_Device_Access)
is
   X : Natural := Cuda_Index (Block_Dim.X, Block_Idx.X, Thread_Idx.X);

   ( ... )
begin
   if X <= A'Last then
      C (X) := Int_20 (A (X) + B (X));
   end if;
end VectorAdd;
```

Using only the default Integer type:

```
kernels.adb:41:25: medium: range check might fail, cannot prove lower bound for A (X) + B (X)
   41 |          C (X) := A (X) + B (X);
      |                   ~~~~~~^~~~~~
  reason for check: result of addition must fit in the target type of the assignment
```

Arithmetic Overflows and Underflows

SPARK verification over the silver level guarantees us freedom of runtime errors:

```
if X ⩽ A'Last then
   C (X) := Int_20 (A (X) + B (X) + 1);
   C (X) := Int_20 (A (X) + B (X) - 1);
end if;
```

Prover output:

```
kernel.adb:30:41: medium: range check might fail, cannot prove lower bound for A (X) + B (X) + 1
   30 |          C (X) := Int_20 (A (X) + B (X) + 1);
      |                           ~~~~~~~~~~~~~~^~~
  reason for check: result of addition must be convertible to the target type of the conversion

kernel.adb:31:41: medium: range check might fail, cannot prove lower bound for A (X) + B (X) - 1
   31 |          C (X) := Int_20 (A (X) + B (X) - 1);
      |                           ~~~~~~~~~~~~~~^~~
  reason for check: result of subtraction must be convertible to the target type of the conversion
```

Division-by-Zero

Divisions are common in kernel computations.

The possibility of dividing by zero is usually left unchecked.

```
if X ≤ A'Last then
   C (X) := A (X) / B (X);
end if;
```

```
kernel.adb:32:25: medium: divide by zero might fail
   32 |          C (X) := A (X) / B (X);
      |                   ~~~~~~^~~~~~
```

SPARK's control flow analysis is able to guarantee absence of the error:

```
if X ≤ A'Last and then B (X) /= 0 then
   C (X) := A (X) / B (X);
end if;
```

Functional Correctness Guarantees

```
procedure VectorMax
  (A : Vector_Device_Access; B : Vector_Device_Access;
   C : Vector_Device_Access)
is
   X : Natural := Cuda_Index (Block_Dim.X, Block_Idx.X, Thread_Idx.X);

   ( … )

   function Max (A, B : Integer) return Integer with
     Post => (Max'Result >= A and Max'Result >= B)
   is
   begin
      if A > B then
         return A;
      else
         return B;
      end if;
   end Max;
begin
   if X <= A'Last then
      C (X) := Max (A (X), B (X));
      pragma Assert (C (X) >= A (X) and C (X) >= B (X));
   end if;
end VectorMax;
```

- Preconditions/Postconditions
- Loop Invariants
- Assertions

Assertions can be evaluated at runtime, but in SPARK code they are also used by the prover.

A typo in the Max function, like A<B instead of A>B, results in this:

```
kernel.adb:29:18: medium: postcondition might fail, cannot prove Max'Result >= A
   29 |        Post => (Max'Result >= A and Max'Result >= B)
      |                 ^~~~~~~~~~~~~~~
```

Fixed-Point Arithmetic

- Floating point numbers are a source of inaccuracies.
- Accumulated errors can result in silent bugs.
- Fixed point types are predictable.
- The prover treats them as scalar integers.

No warnings are reported from the prover on this example:

```
type Grade is delta 0.1 digits 3 range 0.0 .. 10.0;

type Grade_Vector is array (Natural range <>) of Grade;
```

```
procedure AvgGrades
  (A : Grade_Vector_Device_Access; B : Grade_Vector_Device_Access;
   C : Grade_Vector_Device_Access)
is
   X : Integer := Cuda_Index (Block_Dim.X, Block_Idx.X, Thread_Idx.X);

   ( ... )

   type Grade_20 is delta 0.1 digits 3 range 0.0 .. 20.0;
   tmp : Grade_20;
begin
   if X <= A'Last then
      tmp    := Grade_20 (A (X) + B (X));
      tmp    := tmp / 2;
      C (X) := Grade (tmp);
   end if;
end AvgGrades;
```

Buffer Overflow Detection

Detecting Buffer Overflow Errors

- Buffer overflow errors are very common in GPU programming.
- They can result in (possibly silent) bugs.
- Possible bugs include:
  - Dimensions we give at the kernel invocation
  - Indexing inside the kernel
  - Memory transfers
- We need a way to detect such errors.

A Programming Pattern for Buffer Overflow Detection

The prover analyses the host and GPU code in isolation.
There must be consistency between the CPU and GPU code.

Step 01

Construct a wrapper for the CUDA kernel invocation and the data transfers before and after it.

Step 02

Add preconditions in the wrapper's specification that dictate invariants among the vectors' ranges and the given block and grid dimensions.

```
procedure VectorAddWrapper
  (Threads_Per_Block :     Pos3; Blocks_Per_Grid : Pos3; A, B : Vector;
   C                 : out Fat_Vector; Vector_Size : Positive) with
  Pre =>
   Threads_Per_Block.X * Blocks_Per_Grid.X in Positive'Range
   and then
   (A'First = 0 and B'First = 0 and C'First = 0 and
    A'Last = Vector_Size - 1 and B'Last = Vector_Size - 1 and
    C'Last = Vector_Size - 1 and
    A'Last = (Threads_Per_Block.X * Blocks_Per_Grid.X) - 1 and
    B'Last = (Threads_Per_Block.X * Blocks_Per_Grid.X) - 1 and
    C'Last = (Threads_Per_Block.X * Blocks_Per_Grid.X) - 1);
```

A Programming Pattern for Buffer Overflow Detection

Step 03

Reflect the wrapper's preconditions with Ada-SPARK assumptions in the declaration part of the kernel's body.

```
function Cuda_Index
  (Block_Dim, Block_Idx, Thread_Idx : unsigned) return Natural with
  SPARK_Mode => Off
is
begin
   return Natural (Block_Dim * Block_Idx + Thread_Idx);
end Cuda_Index;

procedure VectorAdd
  (A, B : not null Vector_Device_Constant_Access;
   C    : not null Fat_Vector_Device_Access)
is
   -------- Mirror wrapper's precondition semantics with assumptions --------
   X : Natural := Cuda_Index (Block_Dim.X, Block_Idx.X, Thread_Idx.X);

   pragma Assume (A'First = 0 and B'First = 0 and C'First = 0);
   pragma Assume (A'Last = B'Last and then B'Last = C'Last);
   pragma Assume (A'Last <= Integer'Last - 31);

   Max_X : Integer := ((A'Last + 31) / 32) * 32;
   pragma Assume (X in 0 .. Max_X);
   --------------------------------------------------------------------------
begin
   if X <= A'Last then
      C (X) := Int_20 (A (X) + B (X));
   end if;
end VectorAdd;
```

A Programming Pattern for Buffer Overflow Detection

If we forget to check whether the index is within the array boundaries, we'll get an error like this:

```
kernels.adb:41:38: medium: array index check might fail
   41 |          C (X) := Int_20 (A (X) + B (X));
      |                                ^ here
  reason for check: value must be a valid index into the array
```

```ada
function Cuda_Index
  (Block_Dim, Block_Idx, Thread_Idx : unsigned) return Natural with
  SPARK_Mode => Off
is
begin
   return Natural (Block_Dim * Block_Idx + Thread_Idx);
end Cuda_Index;

procedure VectorAdd
  (A, B : not null Vector_Device_Constant_Access;
   C    : not null Fat_Vector_Device_Access)
is
   -------- Mirror wrapper's precondition semantics with assumptions --------
   X : Natural := Cuda_Index (Block_Dim.X, Block_Idx.X, Thread_Idx.X);

   pragma Assume (A'First = 0 and B'First = 0 and C'First = 0);
   pragma Assume (A'Last = B'Last and then B'Last = C'Last);
   pragma Assume (A'Last <= Integer'Last - 31);

   Max_X : Integer := ((A'Last + 31) / 32) * 32;
   pragma Assume (X in 0 .. Max_X);
   --------------------------------------------------------------------------
begin
   if X <= A'Last then
      C (X) := Int_20 (A (X) + B (X));
   end if;
end VectorAdd;
```

GPU4S Benchmark Suite Port (Use cases)

Open Source - Used by the European Space Agency

Seven benchmarks have been ported first to Ada for CPU and then GPU:

- matrix_multiplication_bench (int + float version)
- convolution_2D_bench (int + float version)
- fir_filtering (int + float version)
- max_pooling_bench (int + float version)
- relu_bench (int + float version)
- softmax_bench (int + float version)
- correlation_2D (float only version)
- LRN_bench (float only version)

Even without any specific SPARK verification attempts, the benchmarks reach **stone level** verification.

For all of them, we reached **silver adoption level**.

For several of them, we also reached **gold adoption level**.

GPU4S Bench Identified Issues

```
    unsigned int size_A = arguments_parameters->size;
unsigned int mem_size_A = sizeof(bench_t) * size_A;
    bench_t* A = (bench_t*) malloc(mem_size_A);
    // B output matrix
    unsigned int size_B = arguments_parameters->size + arguments_parameters->kernel_size - 1;
unsigned int mem_size_B = sizeof(bench_t) * size_B;
    bench_t* h_B = (bench_t*) malloc(mem_size_B);
    bench_t* d_B = (bench_t*) malloc(mem_size_B);
```

```
for (int i=0; i<arguments_parameters->size; i++){
    #ifdef INT
    A[i] = rand() % (NUMBER_BASE * 100);

for (int i=0; i<arguments_parameters->size; i++){
    h_B[i] = 0;
    d_B[i] = 0;
```

Incomplete Array Initialisation (use of uninitialised variables)

Conclusions and Contributions

- We found a way to run the Ada-SPARK prover on Ada code for GPUs
- We developed examples showcasing that is possible to run SPARK tools on kernel code
- We constructed a pattern for buffer overflow detection across host and device code
- We ported GPU4S benchmark suite to Ada for GPUs
  - applying our developed methodologies
  - achieving at minimum silver adoption level
  - demonstrated that errors found in C/CUDA version do not exist in the Ada SPARK version
- All our developments are released as open source [1][2]
- For more details check our DATE 2024 and Ada Europe 2024 publications [3][4]

[1] Ada SPARK GPU Examples, https://gitlab.bsc.es/dimitris_aspetakis/ada-spark-gpu
[2] GPU4S Ada SPARK port, https://gitlab.bsc.es/dimitris_aspetakis/gpu4s-bench-ada
[3] Formal Methods for High Integrity GPU Software Development and Verification, DATE 2024
[4] Using AdaCore's GNAT for CUDA for Safety Critical GPU Code Development and Verification, AEiC 2024

Acknowledgements

This work was supported by:

- The European Space Agency (ESA) through the Formal Methods for GPU Software Development and Verification project (ESA STAR AO 2-1856/22/NL/GLC/ov).
- The European Commission through the METASAT Horizon Europe Project under grant agreement number 101069595.
- AdaCore through a license and support for all related tools used in this work.
- The Spanish Ministry of Science and Innovation under the grant IJC2020-045931-I.

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

EXCELENCIA SEVERO OCHOA

Thank you!

esa

AdaCore