Formal Methods for GPU Software Development Using Ada SPARK

Leonidas Kosmidis, Dimitris Aspetakis, Matina Maria Trompouki

ESA STAR AO 2-1856/22/NL/GLC/ov

ESA Technical Officer: Gabor Marosy

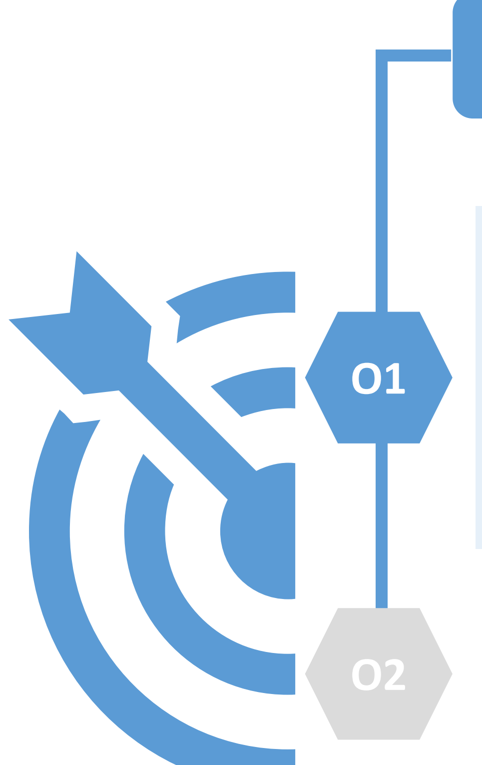
28/09/2023

Software Product Assurance Workshop 2023

Outline

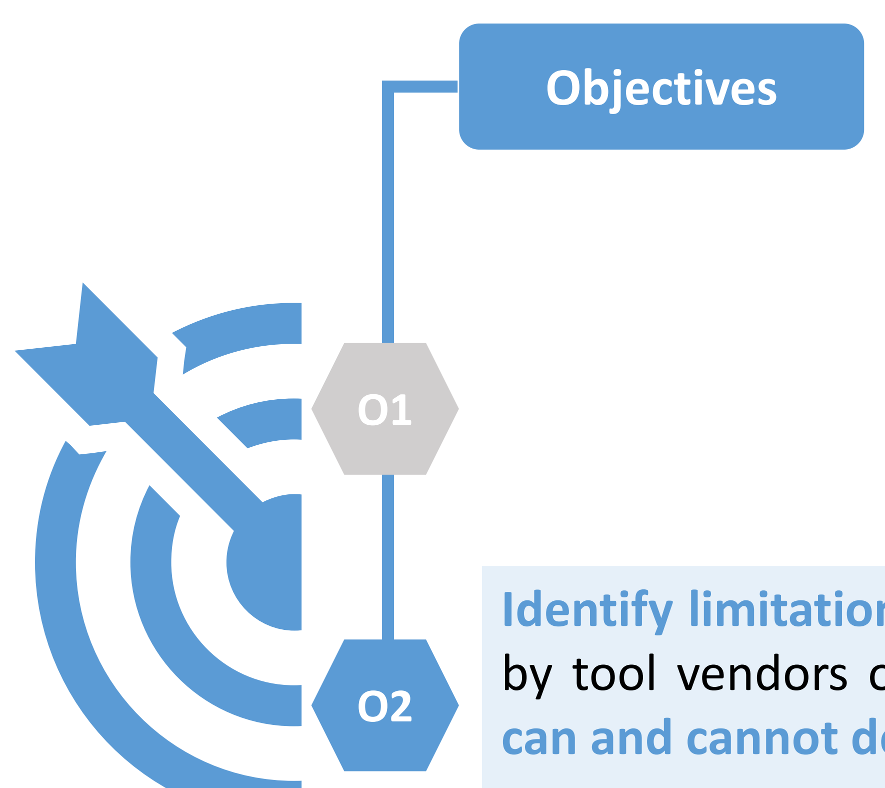
- Introduction and motivation
- Background
  - The GPGPU programming architecture
- Project Contributions
  - Device code verification
  - Buffer overflow detection
- Conclusions

Project Objectives



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**

Project Objectives



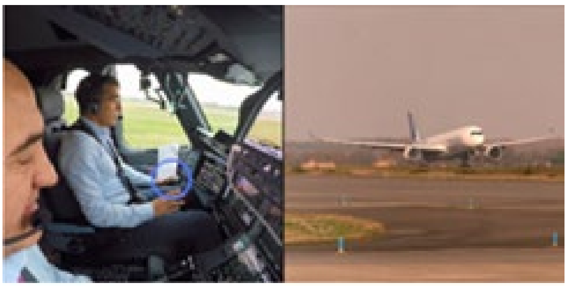
Identify limitations of formal methods for GPU code, so that can be addressed by tool vendors or/and the potential GPU users **understand what these tools can and cannot do**

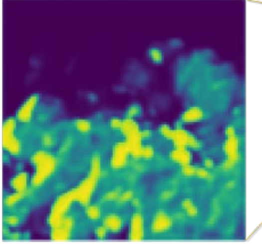
Safety Critical Systems

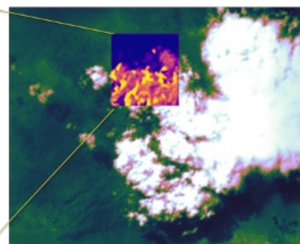
- Used in avionics, automotive and aerospace industries
- Correct execution is of paramount importance
  - Any malfunction may be dangerous
  - Designed to comply with **functional safety standards**:
    - Automotive: ISO 26262, Avionics: DO-178C, Aerospace ECSS
- Traditionally rely on very old and 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 shown very promising performance results on space relevant processing, especially on NVIDIA GPUs like NVIDIA Xavier and TX2

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 …
- Automotive Safety Integrity Level (ASIL)
- Highest Criticality software (ASIL-D) needs to … certain rules:
  - Restricted use of Pointers
  - No dynamic memor…
  - Static ve… program properties
  - … testing methods like MC/DC (Modified Condition/Decision Coverage)
- …ilar 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

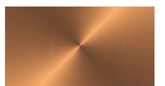
Ada SPARK

- 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

Ada SPARK — Adoption Levels

- Stone

  - The code uses only the SPARK executable subset



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

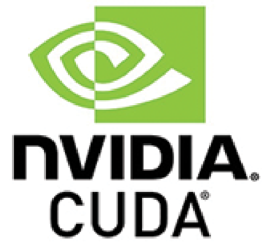
- Silver
  - Guarantees absence of runtime errors (including buffer and numerical overflow, division-by-zero)

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

- Platinum
  - Full functional proofs of the requirements

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 a CUDA Backend for Ada
  - 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



The GPGPU Programming Architecture

GPUs: Programmer's View

GPU / Device code

GPU Programming Language

CPU / Host code

Programming API

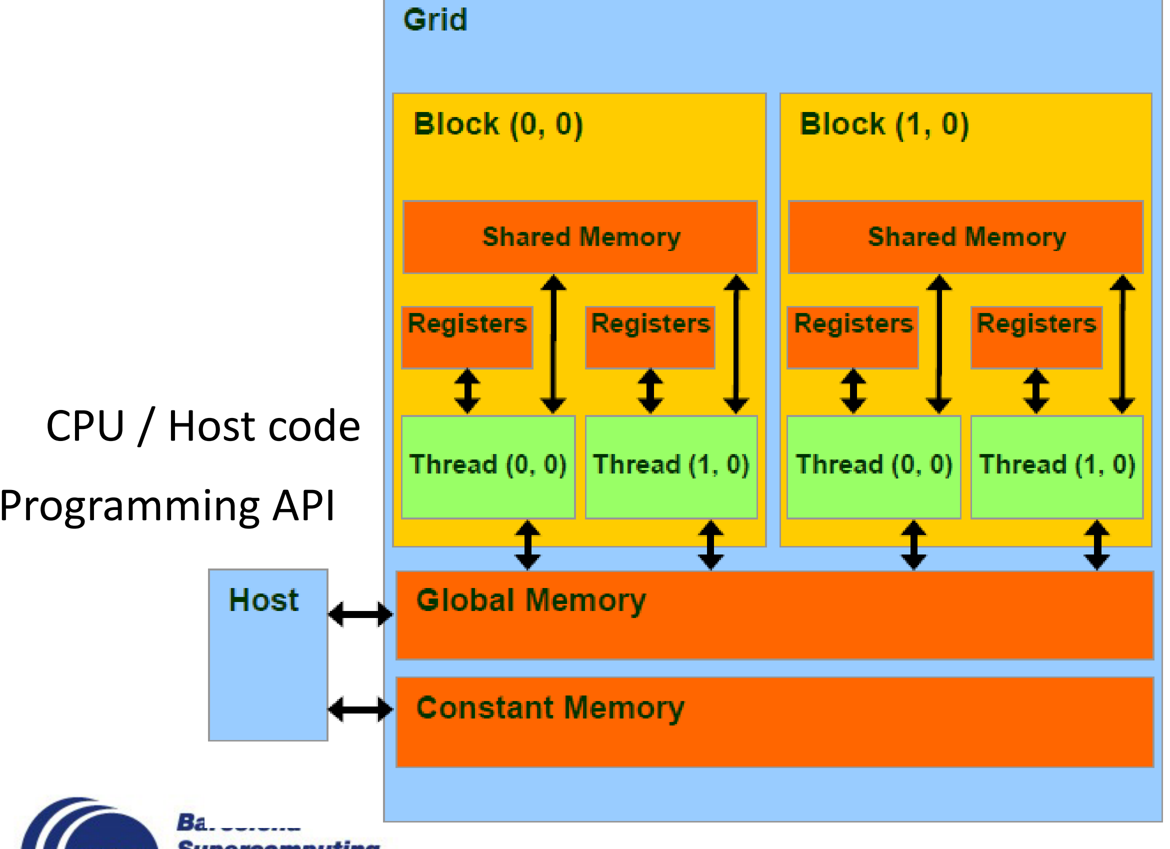


Image credit Wen-mei W. Hwu (UIUC) and David Kirk (NVIDIA)

- Single instruction – multiple threads
- Threads organised in up-to 3D groups
  - Unique thread identifiers
- Several memory types
- Memory Transfers to/from host CPU
  - DMA accesses over PCIe
  - Explicit memory allocation and transfers
  - Raw pointers

Example: Vector Addition Kernel in CUDA

Device Code

```
__host__
void vecAddKernel(int* A, int* B, int* C, int n) {
  int i = threadIdx.x + blockDim.x * blockIdx.x

  if (i < n) {
    C[i] = A[i] + B[i];
  };
}
```

Example: Vector Addition Kernel Launch in CUDA (Host Code)

Host Code

The ceiling function 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];
}
```

Grid

Blk 0 • • • Blk N-1

GPU

M0 • • • Mk

RAM

Project Contributions on GPU Software Verification

Methodology

- Created small examples containing specific classes of GPU programming mistakes
- Examined whether the SPARK formal tools are able to identify them
- Identified how the programmer can guide the tools to detect them
- Developed programming methodologies for Ada SPARK GPU programming
- Developed case studies applying our methodology:
  - Max
  - Histogram
  - Ported GPU4S Bench benchmarks to Ada SPARK GPU, achieving at minimum stone level of SPARK adoption and in several cases bronze level

Device code verification

Guarantees from the Typing System

```ada
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;
```

```ada
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:

```ada
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
```

Arithmetic case study from GPU4S Bench

- GR740’s FPU does not support denormal numbers as input
- Underflow exception raised under RTEMS

```
for (int i=0; i < size_k; ++i)
{
        #ifdef INT
        kernel[i] =  rand() % (NUMBER_BASE * 100);
        #else
        kernel[i] = (bench_t)rand()/(bench_t)(RAND_MAX/NUMBER_BASE);
        #endif
}
```

Currently working on statically detecting of this issue in Ada SPARK

Division-by-Zero

Divisions are common in kernel computations.

The possibility of dividing with zero is usually left unchecked.

We might get runtime errors if coverage is not sufficient.

```
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 as static proof targets.

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 result in silent bugs.
- Fixed point types are predictable.
- The prover treats them as scaled 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 kernels.
- They can result in (possibly silent) bugs.
- There may be a problem inside the kernel with indexing.
- There might also be a problem with the dimensions we give at the kernel invocation.
- We need a way to detect such errors.

GPU4S Bench Identified Issues

- Incomplete initialisation due to wrong size

```
38        unsigned int size_A = arguments_parameters->size;
39    unsigned int mem_size_A = sizeof(bench_t) * size_A;
40        bench_t* A = (bench_t*) malloc(mem_size_A);
41        // B output matrix
42        unsigned int size_B = arguments_parameters->size + arguments_parameters->kernel_size - 1;
43    unsigned int mem_size_B = sizeof(bench_t) * size_B;
44        bench_t* h_B = (bench_t*) malloc(mem_size_B);
45        bench_t* d_B = (bench_t*) malloc(mem_size_B);
60                for (int i=0; i<arguments_parameters->size; i++){
61                        #ifdef INT
62                        A[i] = rand() % (NUMBER_BASE * 100);

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

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

GPU4S Bench Identified Issues

- Out of bounds accesses in kernel execution due to incorrectly specified sizes
- Code runs without a crash or output verification in other platforms under RTEMS and Linux!

```
void execute_kernel(GraficObject *device_object, unsigned int n, unsigned int m,unsigned int w, unsigned int kernel_size)
{
        // Start compute timer
    struct timespec start, end;
    clock_gettime(CLOCK_MONOTONIC_RAW, &start);
        const unsigned int kernel_rad = kernel_size / 2;
        const unsigned int output_size = n + kernel_size - 1;

        for(unsigned int i = 0; i < output_size; ++i)
        {
                for (unsigned int j = 0; j < kernel_size; ++j)
                {
                        if (i +(j - kernel_size + 1) >= 0 && i +(j - kernel_size +1)<  n)
                {
                        device_object->d_B[i] += device_object->kernel[kernel_size - j - 1] * device_object->d_A[i +(j - kernel_size + 1) ];
```

size_B = n + kernel_size-1

```
execute_kernel(fir_bench, size_B, arguments_parameters->size, size_B, arguments_parameters->kernel_size);
```

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

A Programming Pattern for Buffer Overflow Detection

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

Should we forget to check for the upper boundary of our vectors, 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)

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               (int + float version)`
- `fast_fourier_transform_bench (float only  version)`

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

For several of them, we also reached bronze adoption level.
Gradually we are going to increase the adoption level.

Conclusions and Contributions

- We found a way to run the Ada-SPARK prover on Ada code targeting CUDA
- We developed examples showcasing SPARK's viability on kernel code
- We constructed a pattern for buffer overflow detection across host and device code
- We ported GPU4s benchmark suite to AdaCore’s CUDA backend
  - applying our developed methodologies
  - achieving at minimum stone level SPARK adoption level
  - Latent errors found in C/CUDA version do not exist in the Ada SPARK version
  - All our developments are released as open source [1][2]

[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

BSC
Barcelona
Supercomputing
Center
Centro Nacional de Supercomputación
EXCELENCIA
SEVERO
OCHOA
esa
Thank you!

Backup Slides

Extending Semantics to Better Reflect the CUDA Model

Emulating Kernel Invocation Semantics

We lack information that could be useful in specification verification.

Our kernel code is going to be run multiple times according to the grid and block dimensions.

The use of Ada-SPARK's *ghost* constructs helps us emulate the missing semantics.

```
procedure VectorMax
  (A, B, C : Vector_Device_Access)
is
   -- Mirror wrapper's precondition semantics with assumptions
   ( ... )
   procedure Max_Apply_All (A, B : Vector; C : in out Vector) with Ghost,
     Pre  ⇒
      (A'First = 0 and B'First = 0 and C'First = 0)
      and then (A'Last = B'Last and A'Last = C'Last),
     Post ⇒ (for all I in C'Range ⇒ C (I) ⩾ A (I) and C (I) ⩾ B (I))
   is
   begin
      for I in C'Range loop
         C (I) := Integer'Max (A (I), B (I));
         pragma Loop_Invariant (for all Idx in C'First .. I ⇒
                                C (Idx) = Integer'Max (A (Idx), B (Idx)));
      end loop;
   end Max_Apply_All;
   C_Ghost : Vector := C.all with Ghost;
begin
   if X ⩽ A'Last then
      C (X) := Integer'Max (A (X), B (X));
      pragma Assert (C (X) ⩾ A (X) and C (X) ⩾ B (X));
   end if;

   Max_Apply_All (A.all, B.all, C_Ghost);
   pragma Assert
     (for all X in C_Ghost'Range ⇒ C_Ghost (X) ⩾ A (X) and C_Ghost (X) ⩾ B (X));
end VectorMax;
```

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Kernel Patterns Enjoying Proper Verification

Safe Kernel Patterns

The last missing information the prover lacks for proper verification are the asynchronous thread execution model semantics.

We identify two common kernel patterns that are inherently safe from data races and synchronization problems:

- A kernel that takes arbitrary read-only inputs, one write-only output vector, and within each thread writes a single and unique output cell.
- A kernel that takes arbitrary read-only inputs, and has either one write-only scalar, or one write-only non-scalar output that gets updated exclusively through atomic operations.

In both cases, the kernel must not make use of the GPU's shared memory.