UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Facultat d'Informàtica de Barcelona

FIB

EVALUATION OF THE ADA-SPARK LANGUAGE EFFECTIVENESS IN GRAPHICS PROCESSING UNITS FOR SAFETY CRITICAL SYSTEMS

DIMITRIOS ASPETAKIS

Thesis supervisor: LEONIDAS KOSMIDIS (Department of Computer Architecture)

Degree: Bachelor Degree in Informatics Engineering

Specialisation: Computer Engineering

Thesis report

Facultat d'Informàtica de Barcelona (FIB)

Universitat Politècnica de Catalunya (UPC) - BarcelonaTech

15/05/2023

Abstract

Modern safety critical systems require high levels of performance for the implementation of advanced functionalities, which are not possible with the simple conventional architectures currently used in them. Embedded General Purpose Graphics Processing Units (GPGPUs) are among the hardware technologies which can provide the high performance required in these domains. However, their massively parallel nature complicates the verification of their software and increases its cost because it usually involves code coverage through extensive human-driven testing.

The Ada SPARK language has traditionally been used in highly-critical environments for its formal verification capabilities and powerful type system. The use of such tools, especially those being backed up by theorem provers, has significantly lowered the amount of effort needed to validate functionality of safety-critical systems.

In this work, we utilize AdaCore's CUDA backend for Ada – currently in closed beta – in conjunction with the SPARK language subset to assess the state of static verification for GPU kernels. We show how common programming mistakes in GPU kernels can be prevented, formulate a pattern for buffer overflow detection, and close with a few GPU case studies.

Acknowledgements

This work was performed within the European Space Agency (ESA)-funded project "Formal Methods for GPU Software Development and Verification" (ESA STAR AO 2-1856/22/NL/GLC/ov) and the European Commission's METASAT Horizon Europe project (grant agreement 101082622).

This work was also supported by AdaCore, in the form of license donation for the Ada CUDA backend for GPUs which is currently under development and available in closed beta, as well as GNAT Prove and technical support.

Finally, this Bachelor's thesis was also partially supported by the Spanish Ministry of Economy and Competitiveness (MINECO) under the grant IJC2020-045931-I.

Contents

1 Introduction

Since the early days of civilization, humans have been creating machines to aid them in life. The kinds of inventions vary, with some being mere toys for entertainment, others providing aid in real-world tasks previously carried out by humans, and —some of them— drastically changing how we live our lives. While most are considered harmless in use, there are some (see cars, airplanes and medical equipment) that should they somehow fail during operation, the consequences are severe — possibly even deadly. All those inventions have come to be characterized as **safety critical systems**.

The information technology era has enabled unprecedented rates of innovation. There are many, and possibly world-changing ideas stirring up in R&D laboratories all over the world. While early stages of those systems might be less safe and secure than later iterations, humans have always striven to ensure critical failure rates are kept extremely low before their eventual mass adoption. Consequently, this makes it much harder for institutions and companies to develop and roll out their safety-critical products.

While traditional safety-critical systems were mechanical, modern systems are heavily based on computing systems, implementing functionalities "X-by-wire" i.e. drive or fly-by-wire. In particular, current cars or plane steering systems do not rely on hydraulic systems which transfer driver's/pilot's movement to forces that control mechanical parts, but instead consist of sensors, actuators and computing systems which control the relevant components by software. Modern low-end cars currently contain more than 100 million lines of code, creating a major challenge for their verification [35].

To combat this safety verification bottleneck, automated verification tools were developed to aid the resource-heavy manual testing and code reviews. Those verification tools can be divided into two subcategories: dynamic and static. Dynamic tools are usually easy to apply, but their effectiveness relies on the tests' coverage exhaustiveness. Test coverage is not an easy problem to solve, and even if done properly, it might end up needing more computation time than what is acceptable. Static verification tools on the other hand, operate on the semantics of systems, trying to prove properties that hold for all possible inputs.

1.1 Problem Formulation

Modern safety critical systems require high performance processing power for the implementation of advanced functionalities such as advanced driver assistance systems (ADAS) and upcoming fully autonomous systems. Some of these features, such as automatic emergency breaking, are mandatory for all vehicles sold in the European Union starting from 2022 [42]. These func-

tionalities cannot be implemented with traditional processing elements used in safety critical systems, which are relying on older and simpler processing technologies. As an indication, a modern car contains more than 100 Electronic Control Units (ECUs), while the advanced functionalities require more complex technologies [35].

Among the candidate processing technologies, Embedded General Purpose Graphics Processing Units (GPUs) are the most promising due to the fact that they can offer high performance, low power consumption and easier programmability than other parallel systems. For this reason, they are considered among all types of safety critical systems such as in the automotive [48], avionics [34, 39] and space [38].

While conventional software systems running on CPUs have long enjoyed the advantages of the static verification approach [1, 26], GPUs have not really dived deep into this area. While some research tools do exist as we describe in the upcoming section, each of them targets a specific problem area, and are not closely coupled with the underlying programming language. A unified programming environment would greatly benefit the adoption of GPGPUs on safety-critical systems, but such a product is surprisingly still absent.

The best current solution is writing GPU kernels in a non-safe language like CUDA, and using those tools – possibly aiding them with annotations – to get the necessary verification results. Due to the tools being targeted at GPU kernels, interaction with the surrounding CPU code is limited, if not non-existent. So, not only we do need multiple different tools to achieve a provably bug-free system, we also have to make additional verification for the interactions between CPU and GPU code [40].

1.2 Contributions

The contributions of this Bachelor's thesis are the following:

- First, we evaluate the possibility of programming GPUs for safety critical systems using the Ada SPARK language subset.
- Next, we examine the strengths and limitation of Ada SPARK's formal methods for preventing GPU software errors. In particular, we examine various categories of GPU programming errors and explore whether Ada SPARK and the current version of its associated tools are able to find them and prove the code correctness.
- We develop a methodology for writing GPU software in Ada SPARK, which facilitates the identification of some GPU programming errors which are not able to be identified simply by using the Ada SPARK tools.

- We compile a set of Ada SPARK GPU examples with injected GPU software errors, demonstrating how they can be detected. Our implementations are released as open source [45], and contribute to the limited number of publicly available Ada SPARK resources in the literature.
- We demonstrate the effectiveness of Ada SPARK in an open source benchmarking suite of a safety critical domain, GPU4S Bench (GPU for Space) [31]. In particular, we port its benchmarks in the Ada SPARK benchmarks reaching at least stone level of SPARK adoption.

  Our implementations are released as open source [46].

1.3 Thesis Organisation

The rest of this Bachelor's thesis is organised as follows. Section 2 introduces the necessary background and concepts required to understand the contribution of this thesis, and positions our work with respect to other related works in this area. Sections 3 and 4 contain the main outcomes of this thesis. Section 3 discusses how the use of Ada SPARK for the development of GPU code for safety critical systems helps in avoiding several programming mistakes. Section 4 showcases (through two small case studies) some stronger Ada SPARK constructs that aid us in verifying kernels, and mentions our results with a GPU4S benchmarking suite port. Section 5 lays out two common kernel patterns that (in conjunction with strategies we developed and showcase in previous sections) ensure verification guarantees on-par with conventional SPARK verification of CPU programs. Finally, Section 6 provides the conclusions of this thesis, and Section 7 our plans for future work.

2 Background and Related Work

2.1 General Purpose GPUs

GPUs (graphics processing units) are specialized hardware devices introduced to accelerate the (highly date-parallel) graphics in our computing systems. Slowly, their SIMT (single instruction, multiple threads) model got the attention of the HPC (high-performance computing) industry, leading to the introduction of general purpose GPU programming. Nowadays, GPUs hold a significant role in HPC and the industry, being the most appropriate devices to handle massively parallel workloads. This is evident in the most recent edition of the Top500 Supercomputing List (November 2022), in which the majority of the supercomputers, are based on GPUs, including the number 1 and the 7 supercomputers in the top 10.

With the introduction of embedded and mobile GPUs, this type of computing devices has dominated also consumer devices. Moreover, as all safety critical domains require high performance for the implementation of advanced functionalities, GPUs are considered also for use in these domains.

GPUs are programmed using heterogeneous programming models such as CUDA and OpenCL, both of which are based on the C programming language. GPU are accelerators, which means that are not standalone devices. Therefore they require the presence of a host processor, ie. a Central Processing Unit (CPU). The CPU and the GPU have distinct address spaces, even in the case where both systems share the same main memory, as it is the case in embedded GPUs.

GPU programming entails writing two different types of programs, one targeting the CPU and one targeting the GPU. The CPU code is in charge of performing the main interaction with the system as well as regular computations. Whenever a heavy computation is required, the CPU can offload the computation to the GPU. This happens in the form of *kernels*, which are functions sent to GPU for the execution. GPU kernels are written in a kernel programming language, such as CUDA – used in NVIDIA devices – or OpenCL – a Khronos standard supporting multiple vendors – which is a subset of the C language.

The programmer needs to specify the kernel *configuration* which describes how many threads will be used for the kernel execution, as well as how these threads are organised in *blocks* within the *Grid* of threads, as shown in Figure 1. Threads within the same block are executed in the same hardware unit in the GPU, known as streaming multiprocessor (SM) in NVIDIA's terminology. Moreover, they can communicate through a fast on-chip memory known as *shared memory* and synchronise their execution using *barriers*. Threads from different blocks can only communicate through *Global Memory*, which resides in DRAM. Finally, GPU threads are executed

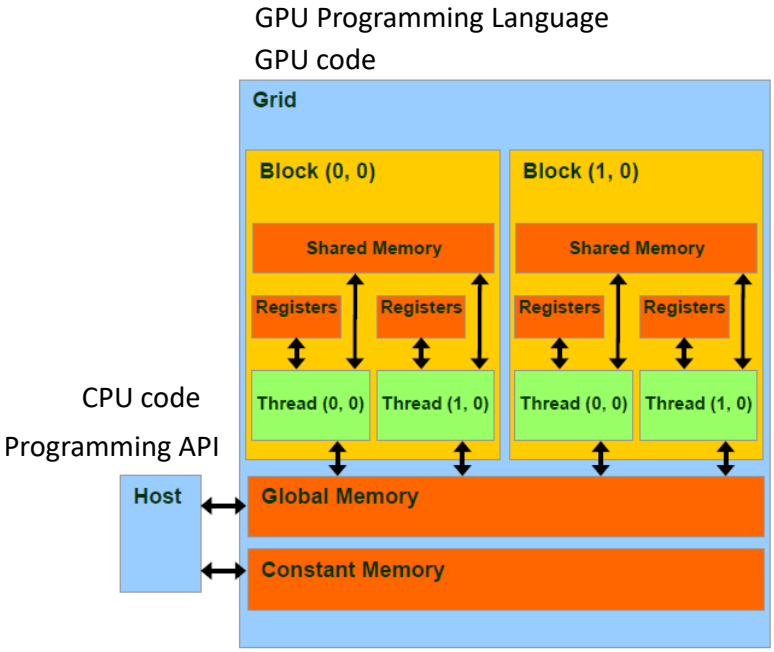

Figure 1: GPU Programmer's view. Image credit Wen-mei W. Hwu (UIUC) and David Kirk (NVIDIA).

in lockstep groups of 32 threads, known as *warps*.

Since the CPU and the GPU have different address spaces, it is the programmer's responsibility to manually perform memory transfers between them, before and after a kernel is executed.

2.2 Safety Critical Systems

Contrary to most systems, safety-critical systems value above all their unhindered and correct functionality conforming to their predefined specifications. Such systems include satellites, spacecrafts (both on-earth and outer space ones), cars, and in general, most systems that interact with the real world and are either vital to something or dangerous. These systems need to be developed and demonstrate compliance with *functional safety standards*, which consist of a set of rules than need to be followed during their development and their verification. Different safety critical domains have different functional safety standards, however all of them are very similar, and therefore a system developed for a critical domain, can be adopted for use in another one. In the automotive domain, ISO 26262 [27] is used, while avionics follows the DO-178C [12] and the aerospace domain uses the ECSS (European Cooperation for Space Standardization) set of standards [7].

Common points among all safety critical standards for software development include the limited use of pointers and dynamic memory allocation, as well as extensive code coverage of software during testing, in the form of Modified Condition/Decision Coverage (MC/DC) [40].

Lately, safety critical systems are in need for very high performance, to enable the implementation of advanced functionalities. For this reason, the adoption of massively parallel architectures seems very promising. However, as discussed in the previous subsection, GPUs are programmed with low-level programming languages based in C. The C language although it is portable, very efficient and its use is widespread to almost any type of system, places an important burden to the programmer in order to write correct code, as well as to the system verification.

Previous analyses for the use of GPUs in safety critical systems have shown that existing GPU programming languages violate several of the guidelines of functional safety standards for software development [28][39]. In particular, the fact that GPU programming requires extensive use of pointers, dynamic memory allocations and explicit memory transfers between the CPU and GPU memory spaces, violate these rules, and complicates their verification.

In order to design and ship a safety-critical system in the market, one has to validate that it respects its specifications, as it is required from the different functional safety standards. Two options have been prevalent in achieving this: extensive testing and formal methods. Extensive testing, complimented by strict work and test methodologies such as MC/DC coverage [40], is still quite prevalent in the industry.

As the systems get increasingly more complicated though, this approach fails to scale along with them. This has attracted companies and the research community to turn their attention towards the aforementioned formal methods. In fact, the avionics domain has a specific variant of DO-178C, which explicitly deals with the application of formal methods, the DO-333 [13].

2.3 Formal Methods

Formal methods are techniques based on mathematical principles, that aid in the specification, design and verification of systems. The results of such techniques can be trusted, and no additional proofs should be needed for their promises, given that the design and implementation of the program of the formal method itself is correct.

The advantage of such methods is that the extensive testing is transferred from multiple target programs to the formal method implementation. Once the formal method program can be trusted, target programs designed and/or verified by it can also be trusted, achieving much higher trustworthiness levels that we could ever hope to reach through sheer extensive testing.

Traditionally formal methods have been very difficult to use, since they required deep mathematical knowledge related to the fundamentals of computer science. In particular, this involves knowledge of Hoare and separation logic, as well as the use of niche programming languages and proof assistants like Coq [19]. In this way, program properties needed to be described mathematically, in order to prove either that the program adheres to its specification, or that the generated executable code is equivalent to what the programmer specified [11, 8].

This rare expertise has prevented the widespread use of formal methods despite their benefits. However, recent advances in formal methods and programming languages, such as the ones provided by Ada SPARK and Frama-C [16], allow their use by non-experts without prior knowledge of advanced logic methods.

2.4 Ada SPARK

Ada is a programming language (first released in 1980), that at first mainly targeted embedded and real time systems. With revisions though, it got higher-level features like object-oriented programming and dynamic dispatch. Ada has been used a lot by the safety-critical sector, since it supports both an extensive pool of safety checks at runtime, and the Design-by-Contract methodology. SPARK is a formally defined subset of Ada. As quoted from the Ada Information Clearinghouse (AdaIC) website [49], a service of the Ada Resource Association, SPARK provides many advantages, specifically important for high-integrity, safety critical systems:

> The formal, unambiguous, definition of SPARK allows and encourages a variety of static analysis techniques to be applied to SPARK programs. These include information flow analysis, proof of absence of run-time exceptions, proof of functional correctness, and proof of safety and security properties. Proof of termination is now also possible using loop variant contracts.

Ada SPARK has been used in highly critical systems such as the Ship / Helicopter Operational Limits Instrumentation System (UK Interim Defence Standard 00-55), and the Lockheed C130J Mission Computer (DO-178B Level A) [4]. Although initially Ada SPARK was proprietary technology of Praxis, currently a GPL licensed version for open source projects is offered by AdaCore, SPARK Community version, while a commercial license is also provided.

Ada SPARK consists both of an executable subset of the Ada language, as well as a subset of its specification language. The former means that Ada SPARK is similar to the MISRA-C [26] subset used for the development of safety critical systems using the C language. The latter means that

SPARK allows the introduction of specification statements about the software behaviour such as preconditions, postconditions and invariants. These statements are not executable, but are converted internally to logic statements known as Verification Conditions (VC) which are used in order to detect errors or prove the correctness of the code. In fact, the AdaCore compiler generates additional Verification Conditions from the executable code, which is combined with the specification statements. These Verification Conditions are passed to an *automatic theorem prover system* which either proves that all these verification conditions hold, or find a counter example in which this is not satisfied. In this case, a user friendly message is provided to the programmer, in order to solve the issue, or to provide additional information in the specification, in order to make it hold.

However, many times it is not possible to prove all verification conditions. In this case, SPARK is using testing for these verification conditions. In fact, specification statements are also converted to dynamic checks, similar to assertions, which are used for this purpose. However, whenever these assertions are proven to hold in all cases, these dynamic checks are disabled, in order to avoid their runtime cost.

SPARK defines 5 adoption levels: Stone, Bronze, Silver, Gold and Platinum [44]. Stone level is achieved when the code is converted to the executable subset of SPARK. Bronze level ensures that initialisation and correct control flow is guaranteed. This is achieved by introducing specifications related to the use of global data in the code. In Silver level, the absence of runtime errors is proven. In Gold level key integrity properties of the software are proven correct. Finally, in Platinum level there is a full functional proof of requirements, ie. that the software meets its specification.

Each level of adoption after Stone level requires the introduction of SPARK specification statements and can be achieved gradually. However, achieving the next level requires additional cost and effort.

As mentioned earlier, Ada SPARK has been used for several decades for the development of safety critical systems for conventional CPU systems. Currently, AdaCore is developing a CUDA backend for its Ada compiler (GNAT), which can generate code for NVIDIA GPUs. This software is at the time of the writing of this document in a closed beta phase. This means that some of the CUDA features are not yet implemented. In this Bachelor's thesis we were provided access to this experimental toolchain in order to evaluate its benefits and current limitations for the development of GPU code for safety critical systems, as discussed in detail in Section 3.

2.5 Ada SPARK Resources

Unfortunately there are very limited available resources for learning Ada SPARK, all of which are only focused on CPU programming. In particular,

the Ada SPARK books from the SPARK inventor John Barnes [3][5][6] which mainly cover earlier versions of the language (SPARK 83, 95 and 2014) and are currently out of print, the Ada SPARK book [22] from McCormick and Chapin which covers the latest version of the language, Jakub Jedryszek's Master's thesis [21] at KSU, the SPARK by example tutorial [30] from Léo Creuse et al and very recently the AdaCore's SPARK website [43].

Publicly available source code related to SPARK is also very rare. This is typical in safety critical and security critical systems, since their development cost is very high and companies developing such systems want to keep their competitive advantage. Moreover, in certain domains such as aerospace, software is subject to export control restrictions which prevent their distribution [38].

Altran, the original company commercialising SPARK before joining forces with AdaCore, has developed an open source, highly secure biometric software case study called Tokeneer under contract from NSA [10][9]. Moreover, Codelabs GmbH is developing an open source, formally proven Separation Kernel for highly critical systems and advanced national security platforms [20][47].

In order to bridge this gap, this thesis produced examples to show how Ada SPARK can be applied for GPU code, as well as a set of GPU case studies in Ada SPARK, which are available as open source [45][46], including a port of the open source GPU4S Bench benchmarking suite [33].

2.6 Related Work

In this subsection, we examine briefly some related works and their relevance to our work, as well as how our work compares with them.

Formal Methods in GPUs

As previously mentioned, it is not the first time that static verification has been applied to GPUs. There exists a handful of research tools developed for detecting potential coding errors in GPU kernels.

The `GPU_Verify` [15] and `GKLEE` [17] tools can be used to find synchronization errors. `GPU_Verify`, along with `VerCors` [14] can also detect data races. `ESBMC-GPU` [24] and `CIVL` [23] can mitigate index-out-of-bounds errors. For functional correctness guarantees, one can use `VerCors` and `Vericuda` [25]. Finally, `CIVL` can also be used for equivalence checking among GPU kernels and their serial implementations.

It is worth noting, that to the best of our knowledge, these tools are aiming to mitigate existing classes of problems, but none of them provides a complete verification strategy, one living up to the expectations of safety-critical software. As an example, the MISRA-C specification [26] and the

Ada SPARK toolchain [1] are able to guarantee absence of integer overflow, unreachable code and division-by-zero errors. However, these methods have been only applied so far to CPU systems. To our knowledge, this is the first first work that evaluates the applicability of Ada SPARK in GPU code.

GPUs in Safety Critical Systems

GPUs have been started only recently to be considered for use in safety critical systems. Trompouki and Kosmidis [28] analysed for the first time that all GPU programming models like CUDA and OpenCL, violate several of the guidelines for software developments found in functional safety standards. This is because they rely on pointers, dynamic memory allocation and low level memory operations such as memory transfers. For this reason, they proposed Brook Auto [28], a high-level, open source programming language which prevents some of the issues found in GPU programming. Brook Auto follows the approach taken by MISRA C in the CPU domain, offering a safe subset of the language to prevent mistakes by restricting error prone language features. However, Brook Auto cannot prove the absence of programming errors, as we do in this thesis in which we employ Ada SPARK's formal features.

BRASIL [32] is an extension of Brook Auto. While Brook Auto [28] defined a safe GPU language subset and proved that GPU code can be certified for automotive use using the ISO 26262 safety standard, it did not cover its tool qualification aspect. All tools used in the development of safety critical systems, need to be *qualified* according to the corresponding functional safety standard, in order to prove that it is safe to be used. BRASIL has studied which modifications were required in the Brook Auto compiler, in order to achieve tool qualification for the development of the highest criticality automotive GPU software according to ISO 26262. To our knowledge, BRASIL has been the first qualifiable GPU programming toolchain.

AdaCore's tools are qualified for use in many safety critical environments, specifically Ada SPARK ones. While the experimental CUDA backend we use in this work is not qualified yet, we believe that it will be qualified once its development is completed, following AdaCore's long tradition in all safety critical domains.

In [39], the authors analysed existing and upcoming methods (i.e. Vulkan SC) for GPU programming in the avionics domain and discussed their certification aspects according to DO-178C. They concluded that the use of high level abstractions need to be used to ease programmability and certification. We believe that Ada SPARK offers this high level abstraction, and its formal features can assist certification as it has been demonstrated in the past for CPU software used in this language.

In [40], the authors discussed the application of MC/DC testing coverage in GPU code in avionics, and identified some aspects like the consistency between host and GPU code as very important for the verification of safety critical GPU code. Moreover, they identified that static analysis and formal methods can complement GPU software testing. The use of Ada SPARK which is examined in our work for GPU programming can provide these checks. In particular, we develop a specific programming pattern that allows the Ada SPARK tools to check consistency between the CPU and GPU code. To our knowledge, this is the first work in the literature, including the GPU formal methods examined in the previous subsection, which offer this feature. In addition, Ada SPARK formal tools even in CPUs, take into account all possible values taken by variables, and therefore can complement MC/DC coverage and help finding errors that might be impossible or very costly to find with testing.

In addition to the aforementioned scientific literature, some prior Bachelor's and Master's theses have explored various aspects of the application of GPUs in safety critical systems.

Marc Benito [29][34] ported an avionics GPU software case study provided by Airbus Defence and Space in Brook Auto/BRASIL and OpenGL SC 2.0 and evaluated the programmability and performance obtained by programming in these GPU languages. Similarly, in this work we use the GPU4S Bench [31] benchmarking suite, which is representative of aerospace software accelerated by GPUs, which we port in the Ada SPARK language.

Alvaro Jover Alvarez [37][36] ported GPU4S Bench in OpenMP and evaluated the performance of embedded CPUs and GPUs, showing that embedded GPUs can provide higher performance than multicore CPUs for computationally intensive workloads found in safety critical systems.

Cristina Peralta [41][48] has evaluated the performance and programmability of two high level programming models for GPUs, OpenACC and SYCL, for use in the development of safety critical GPU software. For this, she ported GPU4S Bench as well as a pedestrian detection application, showing that high level programming models like SYCL, can offer a good trade-off between programmability and performance, and even in some cases achieve the same performance with hand written GPU code.

Despite that these works ported safety critical-relevant GPU code in new languages and programming models, they were mainly focused on the evaluation of performance and programmability for safety critical GPU code. On the other hand, in this work we are focused mainly on software verification, and making sure that we can detect GPU programming mistakes.

3 Avoiding Common Mistakes in GPU Programming using Ada SPARK

In this section, we describe how the use of Ada SPARK helps to eliminate common programming mistakes. We begin by explaining how Ada can be used to describe a GPU kernel, using the CUDA backend of the AdaCore compiler. Next, we examine a series of possible programming mistakes and we show how the SPARK subset of Ada and its formal methods can help detecting or preventing their them altogether.

This section is written in the form of a tutorial, providing a step by step guide of how a GPU program in Ada SPARK can be developed and its functionality can be formally verified. In particular, we start with small program examples in which we intentionally inject the type of errors we want to detect and avoid, and then we show how the Ada SPARK tools can be used to achieve this goal. All the code examples used in this section are included in our open source repository [45].

3.1 Writing a Simple GPU Kernel in Ada

The first step of avoiding several programming mistakes in GPU code starts by using Ada, which has certain advantages over C and C-based languages like CUDA. Without exaggerating, as we discuss in the subsequent subsections, some of these mistakes are impossible to make in Ada, so even the fact of using this language for code development can provide a first level of protection.

Writing a simple GPU kernel in Ada is not that different from writing the equivalent CUDA kernel. We will showcase both the device and host portions of the code (recall Figure 1), since the interactions between them are important and are a common source of programming mistakes. The complete code from the following example can be found in `test00` of our Ada SPARK examples' repository [45].

Let's say we want to write a vector addition kernel, which is the simplest GPU kernel. First, as we do with every Ada procedure, we need to declare the types we are working with, and then provide the specification of the kernel, which is equivalent to a function prototype in C-based languages, but Ada is much more descriptive. This information is provided in a *specification* file with extension `.ads` (Ada specification), similar to the header files used in C-based languages. However, in Ada these files are not included in the form of a preprocessor, which prevents several problems like the inclusion of the same file multiple times.

Ada's pointers equivalent are *access* types. They are more powerful, carrying additional information about the underlying objects, like the size of an array, known as *range* in Ada.

Because the CUDA GPU memory model might differ from the CPU memory model, we need to specify `CUDA.Storage_Models.Model` as the designated storage model for every argument we want to pass to the kernel. We must add the `Cuda_Global` aspect in the kernel specification to tell the compiler that this procedure can be called from both host (CPU) and device (GPU) code. There is also a `Cuda_Device` aspect intended for procedures called only from device code.

```
1 type Vector is array (Natural range <>) of Integer;
2
3 type Vector_Device_Access is access Vector with
4  Designated_Storage_Model => CUDA.Storage_Models.Model;
5
6 procedure VectorAdd
7  (A : Vector_Device_Access; B : Vector_Device_Access;
8   C : Vector_Device_Access) with
9  Cuda_Global;
```

Next, we have to write the body of our kernel, in an Ada implementation file with extension `.adb`. There is nothing special about this procedure compared to a conventional Ada procedure, except that it gets run on multiple GPU threads asynchronously, and has access to Ada's CUDA Runtime API procedures and functions. We use them just like we do in CUDA kernels, to acquire an index for our kernel thread (line 5).

Note that in line 7, we need to check that the index is within the size of the arrays, in order to make sure that there are no out of bounds accesses. As we mentioned in Section 2.1, GPU threads are executed in groups of threads, called blocks. Whenever the programmer launches a kernel, based on the provided grid configuration, the appropriate number of threads is created. Since this number is always multiple of the number of threads used in a block, if the size of the data to be processed is not an exact multiple of the block size, some thread identifiers can go out of bounds.

In a regular CUDA kernel this information needs to be passed as an additional kernel argument, however in Ada, as we mentioned earlier, access types carry also the size of the array, which is retried using the `'Last` attribute.

```
1  procedure VectorAdd
2    (A : Vector_Device_Access; B : Vector_Device_Access;
3     C : Vector_Device_Access)
4  is
5     X : Integer := Integer (Block_Dim.X * Block_Idx.X + Thread_Idx.X);
6  begin
7     if X <= A'Last then
8        C (X) := A (X) + B (X);
9     end if;
10 end VectorAdd;
```

Moving over to the host side, we have some preliminaries too. Since CPU and GPUs have different address spaces, we need to have data structures holding the data for each of these address spaces. We need a separate access type for the host-allocated vectors, and we also need to construct de-allocation procedures for both the host and device access types.

```
1  type Vector_Host_Access is access Vector;
2
3  procedure Free is new Ada.Unchecked_Deallocation
4    (Vector, Vector_Host_Access);
5
6  procedure Free is new Ada.Unchecked_Deallocation
7    (Vector, Vector_Device_Access);
```

Now we can proceed with writing the host code. Just like CUDA, we have to define CUDA's block and grid dimensions of our kernel call (lines 6–8). The call itself is a built-in compiler pragma, where we give it the kernel with its actual parameters alongside the aforementioned dimensions. The other notable thing here is the absence of synchronization before the use of `H_C` in line 24. In conventional CUDA, the data transfers through `cudaMemcpy()` are non-blocking. Line 23 is indeed mapped to a `cudaMemcpy()` call, but is presumably calling `cudaDeviceSynchronize()` too, since documentation tells us that dependent transfers are actually blocking host execution until the kernel completes.

```
1  procedure Main is
2     Vector_Size : Integer := 1_024;
3     H_A, H_B, H_C : Vector_Host_Access :=
4       new Vector (0 .. Vector_Size - 1);
5     D_A, D_B, D_C : Vector_Device_Access :=
6       new Vector (0 .. Vector_Size - 1);
7
8     Threads_Per_Block : Dim3 := (256, 1, 1);
9     Blocks_Per_Grid   : Dim3 :=
10      ((unsigned (Vector_Size) + Threads_Per_Block.X - 1) /
11       Threads_Per_Block.X, 1, 1);
12 begin
13    -- Initialize host vectors
14    Generate_Vector (H_A.all);
15    Generate_Vector (H_B.all);
16    -- Initialize device vectors by transferring the contents
17    -- of the host vectors
18    D_A.all := H_A.all;
19    D_B.all := H_B.all;
20
21    -- Call the kernel
22    pragma Cuda_Execute
23      (VectorAdd (D_A, D_B, D_C), Threads_Per_Block, Blocks_Per_Grid);
24
25    -- Move the device output vector to the host vector and print it
26    H_C.all := D_C.all;
27    Print_Vector (H_C);
28    -- Free all host and device vectors.
29    (...)
30 end Main;
```

As mentioned in Section 2.2, prior work in the use of GPUs in safety critical systems such as Trompouki and Kosmidis [28] have identified that explicit memory transfers between the host and the GPU, and vice versa are a common source of GPU programming mistakes. In particular, explicit low-level memory copies and initializations through `cudaMemcpy()` and `cudaMemset()` are very prone to errors due to the fact that the allocation or memory copy needs to be specified in number of bytes instead of the number of elements, it has to be consistent with the size of the memory structures and access is performed using pointers. Note that the use of Ada prevents all these mistakes, since the access data types make sure that the allocations or memory transfers take only up to the number of elements available in each data structure. Moreover, if due to a programming mistake, the size of the two arrays that are assigned to one another does not match, the program will not compile.

The example presented in this Section runs smoothly and gives correct results. Nevertheless, if we add the `SPARK_Mode` aspect on the specification and bodies of our procedures and run the code through `gnatprove` (the tool responsible for SPARK-compliance), it results in many errors and —because of some of them— it ends up not analysing the actual kernel code.

Enabling the `SPARK_Mode` aspect is the first step towards achieving the stone level of adoption in SPARK, as described in Section 2.4.

3.2 Extending our Kernel with SPARK Verification

The CUDA backend has been an on-going project for AdaCore, and it is apparent from its state and the limited documentation available that they have not considered integrating it with SPARK yet. As such, we are forced to ignore certain warnings and carefully avoid analysing certain procedures to circumvent unavoidable errors. Once the CUDA backend implementation is complete and it is fully integrated with the SPARK formal tools, the work-arounds that we devise and provide in this section, will not be needed anymore.

In this section, we will showcase a programming pattern that allows us to check for correctness across the CPU and GPU code. This is an element which is very important for the validation of GPU software as indicated by [40], but it is not addressed by any related work in the literature.

We will use as an example the vector addition kernel introduced in the previous subsection, but we will focus on the CPU code, and the transition to the GPU. As we will see in Section 3.3, even a simple integer addition cannot always pass SPARK verification.

One of the problems we cannot work around is that SPARK does not allow access types with storage pools. Our best option here is to comment out the `Designated_Storage_Model` attribute entirely from our device ac-

cess types every time we wish to run the prover on our code:

```
type Vector_Device_Access is access Vector;
  --with Designated_Storage_Model => CUDA.Storage_Models.Model;
```

The same action can be applied to omit the `Cuda_Global` aspect from the specification of our kernel, but since it is reflected as a simple warning and does not hinder analysis, we leave it as is in our complete repository examples.

Moving on, the other thing we have to fix in our kernel (that is not part of its data computation), is the construction of the index in the CUDA code. Because the `Block_Dim`, `Block_Idx` and `Thread_Idx` API functions return an unsigned integer from Ada's C interface, the conversion to our vectors' index type (that is, Ada's `Natural` type) is unsafe, let alone the arithmetic overflows that might arise, as the prover indicates:

```
kernel.adb:15:57: medium: range check might fail, cannot prove lower bound for Block_Dim.X * Block_Idx.X
+ Thread_Idx.X
   15 |      X : Integer := Integer (Block_Dim.X * Block_Idx.X + Thread_Idx.X);
      |                              ~~~~~~~~~~~~~~~~~~~~~~~~~~^~~~~~~~~~~~~
  reason for check: result of addition must be convertible to the target type of the conversion
```

We make the reasonable assumption that our kernel's call dimensions will not exceed the upper bound of Ada's `Natural` type, and hence we use a function that will not get analyzed for the conversion:

```
1 function Cuda_Index
2   (Block_Dim, Block_Idx, Thread_Idx : unsigned) return Natural with
3   SPARK_Mode => Off
4 is
5 begin
6    return Natural (Block_Dim * Block_Idx + Thread_Idx);
7 end Cuda_Index;
```

The last thing we must attend to, is assuring that the constructed index never points outside our vectors' bounds. A possible solution would basically alleviate all buffer overflow errors inside our code. Should we try to invoke the prover at this stage, we would get errors like the following:

```
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
```

To combat this issue, we propose the following three-stage pattern:

1. Construct a wrapper for the CUDA kernel invocation and the data transfers before and after it. Importantly, the wrapper's parameters include both the input/output vectors and the desired CUDA block and grid dimensions. The body of the wrapper **will not get analyzed** for SPARK verification.

2. Add preconditions in the wrapper's specification that dictate invariants among the vectors' ranges and the given CUDA block and grid dimensions. The wrapper's specification **will get analyzed** for SPARK verification.

3. In the declaration part of our kernel's body, reflect the wrapper's preconditions with Ada assumptions to properly inform the prover. Here is where we also construct the CUDA index with the predefined `Cuda_Index()` function. Due to the underlying GPU architecture, we might get more threads than we expect. Specifically, the dimensions of our CUDA index will probably get rounded up to a multiple of the architecture's warp size, and hence we need one more Ada assumption for this. Both the specification and body of our kernel **will get analyzed** for SPARK verification.

The best way to apply this pattern mechanically in practice, is to make any assumptions in the declaration part of the kernel's body you need to achieve verification, and then reflect them at the wrapper's specification with preconditions. Here is an example showcasing that pattern:

```
1  procedure VectorAddWrapper
2    (Threads_Per_Block, Blocks_Per_Grid : Pos3;
3     A, B : Vector; C : out Vector) with
4    Pre =>
5     Threads_Per_Block.X * Blocks_Per_Grid.X in Positive'Range and then
6     (A'First = 0 and B'First = 0 and C'First = 0 and
7      A'Last = (Threads_Per_Block.X * Blocks_Per_Grid.X) - 1 and
8      B'Last = (Threads_Per_Block.X * Blocks_Per_Grid.X) - 1 and
9      C'Last = (Threads_Per_Block.X * Blocks_Per_Grid.X) - 1),
10    SPARK_Mode => On;
```

Listing 1: wrapper specification

```
procedure VectorAddWrapper
  (Threads_Per_Block : Pos3; Blocks_Per_Grid : Pos3;
   A, B : Vector; C : out Vector) with
  SPARK_Mode => Off
is
   procedure Free is new Ada.Unchecked_Deallocation
     (Vector, Vector_Device_Access);

   D_A : Vector_Device_Access := new Vector (A'Range);
   D_B : Vector_Device_Access := new Vector (B'Range);
   D_C : Vector_Device_Access := new Vector (C'Range);
begin
   D_A.all := A;
   D_B.all := B;

   pragma Cuda_Execute
     (VectorAdd (D_A, D_B, D_C),
      (Threads_Per_Block.X, Threads_Per_Block.Y, Threads_Per_Block.Z),
      (Blocks_Per_Grid.X, Blocks_Per_Grid.Y, Blocks_Per_Grid.Z));

   C := D_C.all;

   Free (D_A);
   Free (D_B);
   Free (D_C);
end VectorAddWrapper;
```

Listing 2: wrapper body

```
procedure VectorAdd (A, B, C : not null Vector_Device_Access) with
  Pre => A /= null and then B /= null and then C /= null,
  SPARK_Mode => On,
  Cuda_Global;
```

Listing 3: kernel specification

```
procedure VectorAdd (A, B, C : not null Vector_Device_Access) with
  SPARK_Mode => On
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
      -- Kernel data computation
      (...)
   end if;
end VectorAdd;
```

Listing 4: kernel body

It is worth mentioning how both the `Cuda_Index` function and the wrapper's body have a seemingly mechanical implementation. That is important, since those are the two entities that do not get analyzed for SPARK verification. This allows to replace this manual task in the future with automatic code generation.

A complete example showcasing the aforementioned pattern can be found in `test01` of our Ada SPARK examples' repository [45]. With it, we are able to minimize the errors reported by the prover to just 1:

```
main.adb:29:04: medium: "memory accessed through objects of access type" might not be initialized after
elaboration of main program "Main"
   29 |   VectorAddWrapper (Threads_Per_Block, Blocks_Per_Grid, A, B, C);
      |   ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

We have not found a way to get rid of this error. It seems to come from an inability to deduct that the vectors we pass to `VectorAddWrapper()` are initialized, even though we initialize them with random values. We tried to get rid of it by initializing them in our `Main`'s elaboration stage, but the error persisted. We concluded that because it does not hinder further analysis of our host and device code, it was insignificant enough to ignore.

The safety of this pattern is pretty robust. Should you make a mistake in the kernel's specification preconditions, leaving room for buffer overflow errors, any potential buffer overflow will get reported on the code written inside the kernel's body. Should you make a mistake in the host code, like giving incorrect dimensions to the wrapper, the error will get reported on the wrapper's preconditions. The only way to get a buffer overflow on our vectors is to either deviate from the given `Cuda_Index` function, make a mistake in the body of the wrapper, or to improperly reflect the wrapper's precondition semantics in the kernel's assumptions section.

Unfortunately, even though we are currently passing a SPARK verification stage that would guarantee us lack of runtime errors, it is important to note this is not the case here. The prover lacks the semantics of a CUDA kernel launch, namely, the multiple asynchronous executions of the kernel instructed by the invocation's dimensions. Those semantics are important for detection of data races and synchronization errors. We should note here that the current version of the AdaCore's CUDA backend does not support writing Ada kernels using features that introduce data races, like shared memory and synchronisation among threads. Those features will be available in future versions of the tools.

Regardless the missing support for those features, there are certain kernel patterns, as we'll see in Section 5, that can guarantee freedom of those errors, too. Those patterns enable a holistic verification strategy.

3.3 Integer Underflow/Overflow

One might notice that we restrained from applying the simple addition inside the body of our `VectorAdd()` kernel in the previous section. That was on purpose; two of the possible runtime errors that SPARK verification can prove their absence are **integer overflows and underflows**.

The addition of two elements of the `Integer` type will non always fit inside an `Integer`. Letting such an addition slip, the prover understandably complains:

```
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
```

We have two options to alleviate this hindrance:

- Add assertions after the generation of the input vectors that constrain them to a range that can be multiplied by 2 and not exceed Ada's `Integer` range. Those assertions would unfortunately have to get reflected as assumptions in the beginning of our kernel's body.
- Make use of Ada's powerful type system, and create two new `Integer` subtypes for our input and output vectors respectively. The input vectors' subtype should be able to get multiplied by 2 and still be inside the output vector's subtype range.

Here is an example approach following the second option, which we find more appropriate:

```
1  type Int_10 is new Integer range -10 .. 10;
2  type Int_20 is new Integer range -20 .. 20;
3
4  type Vector is array (Natural range <>) of Int_10;
5  type Fat_Vector is array (Natural range <>) of Int_20;
6
7  type Vector_Device_Access is access Vector with
8    Designated_Storage_Model => CUDA.Storage_Models.Model;
9
10 type Vector_Device_Constant_Access is access constant Vector;
11
12 type Fat_Vector_Device_Access is access Fat_Vector with
13   Designated_Storage_Model => CUDA.Storage_Models.Model;
14
15 procedure VectorAdd
16   (A, B : not null Vector_Device_Constant_Access;
17    C    : not null Fat_Vector_Device_Access) with
18   Pre => A /= null and then B /= null and then C /= null, Cuda_Global;
```

Listing 5: kernel specification package

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

Listing 6: computation inside the kernel's body

A showcase of the above strategy can be found in `test02` of our Ada SPARK examples' repository [45].

3.4 Division by Zero

Divisions are commonplace in GPU kernels, but the possibility of dividing with zero is frequently left unchecked. However, in a safety critical system this might result in a hazard if the outcome of such undefined operation is used. SPARK's silver level guarantees freedom of all runtime errors, including this. Lets stay with the previous kernel structure, where we have two input vectors and one output vector. The output will include the division of the elements from the first vector with the second one:

```
procedure VectorDiv (A, B, C : not null Vector_Device_Access) with
  SPARK_Mode => On
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) := A (X) / B (X);
   end if;
end VectorDiv;
```

Listing 7: kernel body

The above code snippet would generate the following errors:

```
kernels.adb:40:25: medium: divide by zero might fail
   40 |         C (X) := A (X) / B (X);
      |                  ~~~~~~^~~~~

kernels.adb:40:25: medium: overflow check might fail, cannot prove lower bound for A (X) / B (X)
   40 |         C (X) := A (X) / B (X);
      |                  ~~~~~~^~~~~
  reason for check: result of division must fit in a 32-bits machine integer
```

We can see that the prover detects the possibility of a 0-value divisor. To circumvent that, we can make sure such a division never happens in any of our possible control flow paths, so that the prover's flow analysis pass can detect this property.

The second error we get is a familiar scalar overflow. The erroneous case here is when we have `Integer'First / -1`, where because of the underlying architecture (that is, conventional signed integers), its result would be `Integer'Last + 1`. The solution here, just like in Section 3.3, is to create a new type for our vectors' items.

Putting it all together, we no longer get any errors reported for our kernel. Here is an example of the aforementioned mitigations:

```
type Safe_Div_Int is
  new Integer range Integer'First + 1 .. Integer'Last;

type Vector is array (Natural range <>) of Safe_Div_Int;

type Vector_Device_Access is access Vector with
  Designated_Storage_Model => CUDA.Storage_Models.Model;

procedure VectorDiv
  (A, B, C : not null Vector_Device_Access) with
  Pre => A /= null and then B /= null and then C /= null, Cuda_Global;
```

Listing 8: kernel specification package

```
procedure VectorDiv (A, B, C : not null Vector_Device_Access) with
  SPARK_Mode => On
is
   -- Mirror wrapper's precondition semantics with assumptions
   (...)
begin
   if X <= A'Last and then B (X) /= 0 then
      C (X) := A (X) / B (X);
   elsif X <= A'Last then
      case A (X) is
         when 0 =>
            C (X) := 0;
         when 1 .. Safe_Div_Int'Last =>
            C (X) := Safe_Div_Int'Last;
         when Safe_Div_Int'First .. -1 =>
            C (X) := Safe_Div_Int'First;
       end case;
   end if;
end VectorDiv;
```

Listing 9: kernel body

A complete example showcasing the division-by-zero error can be found in `test03` of our Ada SPARK examples' repository [45].

3.5 Use of Uninitialized Variables

Uninitialized variables, can lead to runtime errors when they are used. Most programmers that have suffered a NULL-pointer deference understand that very well. Here, SPARK guarantees that for every spot in our code, every variable that gets used has valid content — that is, we cannot ever use an uninitialized variable.

If, for example, we forget to generate random variables for one of our input kernels, say vector A, the error we would get looks like the following one:

```
main.adb:44:58: medium: "A" might not be initialized
   44 |   VectorAddWrapper (Threads_Per_Block, Blocks_Per_Grid, A, B, C, Vector_Size);
      |                                                         ^ here
```

3.6 Ineffectual Statements

Even though ineffectual statements (statements that do not have an effect on our program) might not classify as a runtime error, in clean codebases (like presumably those of safety critical systems) their detection can hint at probable logical errors. Ada SPARK's prover can assist here too. Even though it will not guarantee their total absence, it can detect a good amount of them.

As an example, let's say that in the vector addition kernel, we want to make the addition for the first 100 elements of our output vector, and we want to set the rest to 0, but we make a mistake inside the kernel:

```
1 if X <= A'Last and X < 100 then
2    C (X) := Int_20 (A (X) + B (X));
3 elsif X <= A'Last then
4    X := 0;                -- We ment to write C (X) instead of X.
5 end if;
```

Listing 10: kernel body

We get back a nice error directing us towards the unfortunate typo:

```
kernels.adb:40:07: warning: statement has no effect
   40 |      elsif X <= A'Last then
      |      ^~~~

kernels.adb:41:12: warning: unused assignment
   41 |         X := 0;    -- We ment to write C (X) instead of X.
      |         ~~^~~
```

A showcase of the errors that can be prevented here and on Section 3.5 can be found in `test04` of our Ada SPARK examples' repository [45].

3.7 Fixed-Point Arithmetic

On February 25, 1991, during the Gulf War, there was a failed attempt from an American Patriot missile to intercept and exterminate the threat of an incoming Iraqi Scud missile, resulting in the death of 28 soldiers. In a U.S. Government Accountability Office report published in 1992 [2], the cause of the failure was attributed to a flawed time measurement technique that involved an error accumulation coming from the use of floating point numbers, and their inexact representation.

As we saw, floating point arithmetic can be a source of critical errors due to its inexact representation of numbers like 0.1. As a language targeted at safety critical systems, Ada provides support for fixed-point types too. Those types can represent only exact values. The underlying arithmetic operations' implementation uses conventional integer operations with scaling, and so we should be able to run them on GPU hardware.

As a demonstration of Ada's capabilities to run fixed-point arithmetic operations on CUDA GPUs, we constructed a kernel (`test05` in our Ada SPARK examples' repository [45]) with two input vectors (`A` & `B`) and one output vector (`C`), where `C (I) = (A (I) + B (I)) / 2`:

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

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

type Grade_Vector_Device_Access is access Grade_Vector
  with Designated_Storage_Model => CUDA.Storage_Models.Model;

procedure AvgGrades
  (A, B, C : Grade_Vector_Device_Access) with
  Pre =>
   A /= null and then B /= null and then C /= null,
  Cuda_Global;
```

Listing 11: kernel specification package

```
procedure AvgGrades (A, B, C : Grade_Vector_Device_Access) with
  SPARK_Mode => On
is
   -- Mirror wrapper's precondition semantics with assumptions
   (...)

   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 GradesAvg;
```

Listing 12: kernel body

Type `Grade` is declared to have a delta precision of 0.1, and a range of 0.0–10.0. This means that the `Grade` type can represent the exact values 0.0, 0.1, 0.2, ..., 10.0. As you can see, we need to declare another type, `Grade_20`, with double the range. It is needed for the intermediate states of our computation, since the `A (X) + B (X)` addition cannot fit inside the `Grade` type. The prover can deduct that after the division, the `tmp` variable contains a value inside the range of the `Grade` type. If we add 0.1 to it before its division, we will get this error:

```
kernel.adb:45:23: medium: range check might fail, cannot prove upper bound for tmp + 0.1
   45 |         tmp    := tmp + 0.1;
      |                   ~~~~^~~~
  reason for check: result of fixed-point addition must fit in the target type of the assignment

kernel.adb:47:26: medium: range check might fail, cannot prove upper bound for tmp
   47 |         C (X) := Grade (tmp);
      |                          ^~~
  reason for check: value must be convertible to the target type of the conversion
```

4 Case Studies

In this section we will present two small case studies showcasing more advanced capabilities with the SPARK verification prover and our three-stage pattern for buffer overflows. After those, in Section 4.3, we briefly report our results from the porting of the GPU4S benchmark suite [33], a suite targeted at safety critical GPU kernels for space. The code of the first two case studies can be found in our Ada SPARK examples repository [45], while the ports can be found in our GPU4S Ada benchmarks repository [46].

4.1 Histogram

In this case study, we will demonstrate the power and generality of our three-stage pattern proposed in Section 3.2. The desired result is to have a kernel that takes an input vector of arbitrary size, and counts how many times each of the possible values its elements occurs inside it. The output of this vector will be another vector of the same range as the type of the input vector's elements. Therefore, each of the output vector's cells will represent how many times a certain value occurs inside the input array.

To achieve this, we utilize both Ada's strong type system, and – as mentioned earlier – the generality of Section 3.2's pattern. Firstly, its quite intuitive to use specialized types for the input and output vectors:

```
type Int_1000 is new Integer range 0 .. 1_000;

type Vector is array (Natural range <>) of Int_1000;

type Counter_Array is array (Int_1000'Range) of Natural;

type Vector_Device_Access is access Vector
  with Designated_Storage_Model => CUDA.Storage_Models.Model;

type Counter_Array_Device_Access is access Counter_Array;
  with Designated_Storage_Model => CUDA.Storage_Models.Model;

procedure VectorCount
  (A : not null Vector_Device_Access;
   B : not null Counter_Array_Device_Access) with
  Pre => A /= null and then B /= null, Cuda_Global;
```

Listing 13: kernel specification package

The other important place to check here is the wrapper's specification:

```
procedure VectorCountWrapper
  (Threads_Per_Block : Pos3; Blocks_Per_Grid : Pos3;
   A : Vector; B : out Counter_Array; 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 A'Last = Vector_Size - 1);
```

Listing 14: wrapper specification

We see that, unlike the example in Section 3.2, we do not need to specify all of the vectors' `Range` attributes. B's range is statically defined in its type. Nevertheless, we assert that this range starts at 0, since we know that we are going to index our output vector with the result of the `Cuda_Index` function. It is important to remind here, that even if we miss this or any other semantics for the vectors' range relations, the prover will report errors if we end up going over or under the vectors' ranges inside the kernel.

Moving to the body of our kernel, things get a little bit more complicated:

```
1  procedure VectorCount
2    (A : not null Vector_Device_Access;
3     B : not null Counter_Array_Device_Access)
4  is
5     -- Mirror wrapper's precondition semantics with assumptions
6     (...)
7     Idx : Int_1000;
8     Sum : Natural := 0;
9  begin
10    if X <= Integer (B'Last) then
11       Idx := Int_1000 (X);
12       for I in A'Range loop
13          pragma Loop_Invariant (Sum <= Sum'Loop_Entry + I);
14          if A (I) = Idx then
15             Sum := Sum + 1;
16          end if;
17       end loop;
18       B (Idx) := Sum;
19    end if;
20 end VectorCount;
```

Listing 15: kernel body

We need a valid type to index `B` in line 23, and hence we declare `Idx`, and cast `X` in line 12 to it. This cast is safe because we are inside a condition that implicitly limits `X` to `Int_1000`'s true range (remember, from the vectors' type declarations: `Int_1000'Last = B'Last`).

The last unfamiliar thing here is the loop invariant pragma in line 15. Its purpose is to aid the prover in deducting absence of a possible integer overflow in line 17. The invariant can be proved inductively, since at the loop entry `Sum = 0`. Each iteration we add at most 1 to it, and hence the invariant can inductively be proven. With this invariant information, the prover can deduct that `Sum` has basically an upper bound equal to `A'Range`'s upper bound. Now we can understand why in the preconditions of the wrapper we assured `A'Last <= Natural'Last -1`. Should we assure `A'Last <= Natural'Last` instead, we will get an error like this:

```
   30 |   VectorCountWrapper (Threads_Per_Block, Blocks_Per_Grid, A, B, Vector_Size);
      |   ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

kernels.adb:45:27: medium: overflow check might fail, cannot prove upper bound for Sum + 1
   45 |             Sum := Sum + 1;
      |                    ~~~~^~~
  reason for check: result of addition must fit in a 32-bits machine integer
```

4.2 Max Value

This kernel case study emphasizes another powerful verification strategy pattern specifically targeted at GPU code. Until now, we could only verify functional correctness in our kernel for one cell at a time. That is, we could not assert properties on the whole range of the output vector. To achieve such a thing, we will need to use another one of Ada SPARK's powerful features: *ghost procedures* and *ghost variables*. Constructs with the *ghost* aspect in Ada have the property that do not get compiled to run on the executable, but are taken into account for verification proofs. More importantly, their specification disallows them to have an effect on non-ghost constructs.

The kernel we demonstrate with this pattern is simple. We have the familiar two-input one-output vector kernel, with all vectors having the same range. We want each cell of the output vector to contain the biggest value from the respective input cells, and we want to assert that this holds with a statically-proven assertion at the end of our kernel:

```
pragma Assert
  (for all X in C'Range =>
     C (X) >= A (X) and C (X) >= B (X));
```

Everything new to us is inside the kernel's body:

```
1  procedure VectorMax
2    (A : Vector_Device_Access; B : Vector_Device_Access;
3     C : Vector_Device_Access)
4  is
5     -- Mirror wrapper's precondition semantics with assumptions
6     (...)
7
8     procedure Max_Apply_All (A, B : Vector; C : in out Vector) with
9       Ghost,
10      Pre  =>
11       (A'First = 0 and B'First = 0 and C'First = 0)
12       and then (A'Last = B'Last and A'Last = C'Last),
13      Post => (for all I in C'Range =>
14                 C (I) >= A (I) and C (I) >= B (I))
15    is
16    begin
17       for I in C'Range loop
18          C (I) := Integer'Max (A (I), B (I));
19          pragma Loop_Invariant
20            (for all Idx in C'First .. I =>
21               C (Idx) = Integer'Max (A (Idx), B (Idx)));
22       end loop;
23    end Max_Apply_All;
24
25    C_Ghost : Vector := C.all with Ghost;
26 begin
27
28    if X <= A'Last then
29       C (X) := Integer'Max (A (X), B (X));
30       pragma Assert (C (X) >= A (X) and C (X) >= B (X));
31    end if;
32
33    Max_Apply_All (A.all, B.all, C_Ghost);
34    pragma Assert
35      (for all X in C_Ghost'Range =>
36         C_Ghost (X) >= A (X) and C_Ghost (X) >= B (X));
37
38 end VectorMax;
```

Listing 16: kernel body

We can see the conventional assertion for one cell in lines 28–31. To achieve full-range assertion on the output vector we must somehow inform the prover with semantics that the kernel's computation will get executed as many times as the number of our threads, and with the respective indices.

The pattern we have developed is not as mechanical as the previous one from Section 3.2, but nonetheless it is equally effective. We use the ghost procedure `Max_Apply_All()` to simulate the effect of line 29, and use a loop invariant pragma to aid the verification of its postcondition. After we call it in line 33, it is trivial for the prover to match the ghost function's postcondition with the assertion in lines 34–36.

The ghost variable `C_Ghost` is necessary since, as we said earlier, ghost constructs cannot have an effect on non-ghost ones (in this case, ghost procedure `Max_Apply_All` cannot effect `C`).

4.3 GPU4S Benchmarks

Aside from the examples and the two case studies we presented so far and are included in our Ada SPARK examples repository [45], we also ported the GPU4S Bench benchmarking suite [31]. GPU4S Bench has been developed within the GPU4S (GPU for Space) ESA funded project, and it is representative of computationally intensive algorithmic building blocks used in multiple domains of the aerospace sector. Our implementation is also released as open source, in the GPU4S Ada SPARK repository [46].

In particular, we ported the benchmarks to Ada SPARK, achieving Stone level SPARK adoption. Moreover, we extended the integer implementation of `matrix_multiplication_bench`, achieving bronze level of SPARK verification. The kernels ported are the following ones:

- `matrix_multiplication_bench (int + float implementations)`
- `convolution_2D_bench (int + float implementations)`
- `max_pooling_bench (int + float implementations)`
- `relu_bench (int + float implementations)`
- `softmax_bench (int + float implementations)`
- `correlation_2D (int + float implementations)`
- `fast_fourier_transform (float implementation)`

As expected, all ported benchmarks work properly and provide identical results with their CUDA counterparts. Our work, demonstrates that porting actual GPU programming software from CUDA to Ada SPARK's GPU backend is possible, and opens the door of gradually increasing the confidence of its correctness.

5 Kernel Patterns that can be Fully Verified

As we previously mentioned, because the prover lacks the semantics of the kernel's true execution path, SPARK verification, at least in the current version of the tools, it cannot guarantee use freedom of data races and synchronization errors. The work of this thesis focuses on trying to formally verify GPU code through Ada SPARK, and hence we find it useful to mention a few common kernel patterns that, in conjunction with the two patterns we created for buffer overflow detection and all-thread output verification, give us verification guarantees on-par with conventional CPU SPARK verification. We argue such guarantees for two specific (and quite common) kernel patterns:

1. A kernel that takes arbitrary constant inputs, one write-only output vector, and within each thread writes a single and unique output cell. The kernel must not make use of the GPU's shared memory.

2. A kernel that takes arbitrary constant inputs, and has either one write-only scalar, or one write-only non-scalar output that gets updated exclusively through atomic operations. The kernel must not make use of the GPU's shared memory.

Take note that all of our example kernels from Section 3 fit into the first pattern, as does our `Max_Value` and `Histogram` case studies.

Synchronization problems arise when we either have kernels with phases that need to get synchronized at the block level, and data race issues arise when different threads can write at the same output location without some sort of synchronization. The former case is covered by the write-only nature of our output arrays. The single and unique-cell writes on the first pattern, and the atomics on the second guarantee us freedom of data races.

Even though these restrictive patterns discourage use of advanced GPU features that provide great speedups, it is worth keeping in mind that a safety-critical system's first goal is safety. Furthermore, we should keep in mind that even without advanced techniques, the use of a GPGPU can yield significant performance gains over a conventional CPU implementation.

6 Conclusions

In this Bachelor's thesis, we have evaluated the effectiveness of the Ada SPARK language and its formal methods for the development of General Purpose GPU software for safety critical systems. For this work, we have relied on an experimental CUDA backend for Ada SPARK which is currently under development at AdaCore.

We have shown how CUDA kernels can be programmed in the Ada SPARK subset and we have examined several types of common programming mistakes arising in GPU programming, and how these can be prevented. We have noticed that GPU programming in Ada is not very different from CUDA, although its syntax is more verbose. We observed and discussed that even the use of Ada alone can prevent some of the common GPU programming mistakes, and we have shown that GPU programming in the SPARK subset of the language is also possible and further helps preventing more GPU programming mistakes.

Moreover, we have explored some of the formal verification capabilities offered by the specification aspects of the SPARK language subset, in order to prove the absence of certain errors in GPU code. However, we have noticed that since the CUDA backend is currently under development and has loose integration with the SPARK formal tools, the detection of certain types of errors is not possible at this point. This includes the use of incorrect use of shared memory, data races and wrong thread synchronisation within a kernel, as well as consistency checks between the CPU and GPU code.

Besides these limitations, we come up with a programming pattern which can allow the use of SPARK's formal checks to verify the consistency between CPU and GPU code, and detect any possible buffer overflow errors. We also proposed two common kernel patterns that guarantee us freedom of both data races and synchronization errors. Following these two contributions, one can have verification guarantees for his GPU code at a level on-par with conventional SPARK verification on the CPU.

As part of this work, we have ported a space-relevant open source GPU benchmarking suite, GPU4S Bench [31], achieving bronze-level SPARK verification in one of the benchmarks, and stone level in the rest of them.

All the Ada SPARK code developments of this thesis [45][46] are released as open source, contributing to the wider adoption of Ada SPARK and specifically for GPU development and verification.

7 Future Work

This Bachelor's thesis was performed within the "Formal Methods for GPU Software Development and Verification" ESA-funded activity, which explores the use of formal methods in GPU software. Our work will be extended in this project in order to cover cases which we have identified that can be automated. This includes the implementation of the programming pattern we proposed in Section 3.2 in a GPU source code translator, which can relieve the programmer from this mechanical task, and ensure that it is implemented without introducing any programming mistake in the process.

Moreover, as new features will become available by AdaCore, such as support for shared memory or synchronization in CUDA kernels, we will explore the effectiveness of formal SPARK tools to find them.

In addition, this thesis contributed to the METASAT Horizon Europe project, funded by the European Commission. METASAT is focused on model-based design software development for high performance space systems, based on multi-cores and GPUs. Existing model-based design tools used in the development of space systems like the open source TASTE [18] framework have the capability of generating Ada SPARK code for CPUs, therefore the possibility to generate Ada SPARK code for GPUs will be explored, as well as the application of our methodologies.

These extensions, are planned to be performed as part of a Master's and a PhD thesis.

Bibliography

[1] Bernard Carré and Jonathan Garnsworthy. "SPARK—an Annotated Ada Subset for Safety-Critical Programming". In: *Proceedings of the Conference on TRI-ADA '90*. TRI-Ada '90. Baltimore, Maryland, USA: Association for Computing Machinery, 1990, pp. 392–402. ISBN: 0897914090. DOI: 10.1145/255471.255563. URL: https://doi.org/10.1145/255471.255563.

[2] U.S. Government Accountability Office. *Patriot Missile Defense: Software Problem Led to System Failure at Dhahran, Saudi Arabia*. Report GAO/IMTEC-92-26. 1992.

[3] John Barnes. *High Integrity Ada: The Spark Approach*. Addison-Wesley, 1997. ISBN: 978-0201175172.

[4] Roderick Chapman. "Industrial Experience with SPARK". In: *IEEE Spectrum* (2000).

[5] John Barnes. *High Integrity Software: The SPARK Approach to Safety and Security*. Addison-Wesley, 2003. ISBN: 0321136160.

[6] John Barnes. *Spark: The Proven Approach to High Integrity Software*. Altran Praxis, 2003. ISBN: 0957290500.

[7] European Cooperation for Space Standardization. *Engineering, Management, Product Assurance and Sustainability Space Standards*. 2006.

[8] Xavier Leroy. "Formal verification of an optimizing compiler". In: *2007 5th IEEE/ACM International Conference on Formal Methods and Models for Codesign (MEMOCODE 2007)*. 2007, pp. 25–25. DOI: 10.1109/MEMCOD.2007.371254.

[9] Altran. *Tokeneer SPARK 2014 Source Code*. https://github.com/AdaCore/spark2014/tree/master/testsuite/gnatprove/tests/tokeneer. 2008.

[10] David Cooper and Janet Barnes. *Tokeneer ID Station, EAL5 Demonstrator: Summary Report*. Tech. rep. S.P1229.81.1. https://www.adacore.com/uploads/downloads/Tokeneer_Report.pdf. Altran, 2008.

[11] Xavier Leroy. “Formal Verification of a Realistic Compiler”. In: *Commun. ACM* 52.7 (July 2009), pp. 107–115. ISSN: 0001-0782. DOI: 10.1145/1538788.1538814. URL: https://doi.org/10.1145/1538788.1538814.

[12] RTCA and EUROCAE. *DO-178C / ED-12C, Software Considerations in Airborne Systems and Equipment Certification*. 2011.

[13] RTCA and EUROCAE. *DO-333C / ED-216, Formal Methods Supplement to DO-178C and DO-278A*. 2011.

[14] Afshin Amighi et al. “The VerCors Project: Setting up Basecamp”. In: *Proceedings of the Sixth Workshop on Programming Languages Meets Program Verification*. PLPV ’12. Philadelphia, Pennsylvania, USA: Association for Computing Machinery, 2012, pp. 71–82. ISBN: 9781450311250. DOI: 10.1145/2103776.2103785. URL: https://doi.org/10.1145/2103776.2103785.

[15] Adam Betts et al. “GPUVerify: A Verifier for GPU Kernels”. In: *SIGPLAN Not.* 47.10 (Oct. 2012), pp. 113–132. ISSN: 0362-1340. DOI: 10.1145/2398857.2384625. URL: https://doi.org/10.1145/2398857.2384625.

[16] Pascal Cuoq et al. “Frama-C”. In: *International Conference on Software Engineering and Formal Methods*. 2012.

[17] Guodong Li et al. “GKLEE: Concolic Verification and Test Generation for GPUs”. In: *SIGPLAN Not.* 47.8 (Feb. 2012), pp. 215–224. ISSN: 0362-1340. DOI: 10.1145/2370036.2145844. URL: https://doi.org/10.1145/2370036.2145844.

[18] Maxime Perrotin et al. “TASTE: An Open-source Tool-chain for Embedded System and Software Development”. In: *Embedded Real Time Software and Systems (ERTS)*. 2012. URL: https://hal.science/hal-02191871.

[19] Andrew W. Appel et al. *Program Logics for Certified Compilers*. USA: Cambridge University Press, 2014. ISBN: 110704801X.

[20] Robert Dorn, Adrian-Ken Rueegsegger, and Reto Buerki. “The Muen Separation Kernel”. In: *High Integrity Software Conference*. 2014.

[21] Jakub Jedryszek. “A model-driven development and verification approach for medical devices”. https://krex.k-state.edu/handle/2097/18222. Master’s Thesis. Kansas State University, Department of Computing and Information Sciences College of Engineering, 2014.

[22] John W. McCormick and Peter C. Chapin. *Building High Integrity Applications with SPARK*. Cambridge University Press, 2015. ISBN: 0321136160.

[23] Stephen F. Siegel et al. “CIVL: The Concurrency Intermediate Verification Language”. In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. SC ’15. Austin, Texas: Association for Computing Machinery, 2015. ISBN: 9781450337236. DOI: 10.1145/2807591.2807635. URL: https://doi.org/10.1145/2807591.2807635.

[24] Phillipe Pereira et al. “Verifying CUDA Programs Using SMT-Based Context-Bounded Model Checking”. In: *Proceedings of the 31st Annual ACM Symposium on Applied Computing*. SAC ’16. Pisa, Italy: Association for Computing Machinery, 2016, pp. 1648–1653. ISBN: 9781450337397. DOI: 10.1145/2851613.2851830. URL: https://doi.org/10.1145/2851613.2851830.

[25] Kensuke Kojima and Atsushi Igarashi. “A Hoare Logic for GPU Kernels”. In: *ACM Trans. Comput. Logic* 18.1 (Feb. 2017). ISSN: 1529-3785. DOI: 10.1145/3001834. URL: https://doi.org/10.1145/3001834.

[26] Roberto Bagnara, Abramo Bagnara, and Patricia M. Hill. *The MISRA C Coding Standard and its Role in the Development and Analysis of Safety- and Security-Critical Embedded Software*. 2018. arXiv: 1809.00821 [cs.PL].

[27] International Organization for Standardization. *ISO 26262. Road Vehicles – Functional Safety*. 2018.

[28] Matina Maria Trompouki and Leonidas Kosmidis. “Brook Auto: High-Level Certification-Friendly Programming for GPU-powered Automotive Systems”. In: *55th ACM/ESDA/IEEE Design Automation Conference (DAC)*. 2018.

[29] Marc Benito. “Analysis and Evaluation of Embedded Graphics Solutions for Critical Systems”. https://upcommons.upc.edu/handle/2117/175735. Bachelor’s Thesis. Universitat Politècnica de Catalunya, 2019.

[30] Léo Creuse et al. “SPARK by Example: An Introduction to Formal Verification through the Standard C++ Library”. In: *Ada Lett.* 38.2 (Dec. 2019), pp. 89–96. ISSN: 1094-3641. DOI: 10.1145/3375408.3375415. URL: https://doi.org/10.1145/3375408.3375415.

[31] I. Rodriguez et al. *GPU4S Bench: Design and Implementation of an Open GPU Benchmarking Suite for Space On-board Processing*. Tech. rep. UPC-DAC-RR-CAP-2019-1. https://www.ac.upc.edu/app/research-reports/public/html/research_center_index-CAP-2019,en.html. Universitat Politecnica de Catalunya, 2019.

[32] Matina Maria Trompouki and Leonidas Kosmidis. "BRASIL: A High-Integrity GPGPU Toolchain for Automotive Systems". In: *2019 IEEE 37th International Conference on Computer Design (ICCD)*. 2019, pp. 660–663. DOI: 10.1109/ICCD46524.2019.00094.

[33] Leonidas Kosmidis et al. "GPU4S: Embedded GPUs in space - Latest project updates". In: *Microprocessors and Microsystems* 77 (2020), p. 103143. ISSN: 0141-9331. DOI: https://doi.org/10.1016/j.micpro.2020.103143. URL: https://www.sciencedirect.com/science/article/pii/S0141933120303100.

[34] Marc Benito et al. "Comparison of GPU Computing Methodologies for Safety-Critical Systems: An Avionics Case Study". In: *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. 2021, pp. 717–718. DOI: 10.23919/DATE51398.2021.9474060.

[35] Robert N. Charette. "How Software Is Eating the Car". In: *IEEE Spectrum* 58.6 (2021).

[36] Á. Jover-Alvarez et al. "The UP2DATE Baseline Research Platforms". In: *Design, Automation & Test in Europe Conference (DATE)*. 2021.

[37] Álvaro Jover-Alvarez. "Evaluation of the Parallel Computational Capabilities of Embedded Platforms for Critical Systems". https://upcommons.upc.edu/handle/2117/361408. Master's Thesis. Universitat Polytècnica de Catalunya, Oct. 2021.

[38] Leonidas Kosmidis et al. "GPU4S: Major Project Outcomes, Lessons Learnt and Way Forward". In: *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. 2021, pp. 1314–1319. DOI: 10.23919/DATE51398.2021.9474123.

[39] Matina Maria Trompouki and Leonidas Kosmidis. "DO-178C Certification of General-Purpose GPU Software: Review of Existing Methods and Future Directions". In: *2021 IEEE/AIAA 40th Digital Avionics Systems Conference (DASC)*. 2021, pp. 1–9. DOI: 10.1109/DASC52595.2021.9594412.

[40] Jaime Luis Martin Aleman et al. "On the MC/DC Code Coverage of Vulkan SC GPU Code". In: *2022 IEEE/AIAA 41st Digital Avionics Systems Conference (DASC)*. 2022, pp. 1–9. DOI: 10.1109/DASC55683.2022.9925766.

[41] Cristina Peralta Quesada. "Evaluation of High-Level Programming Models for High-Performance Critical Systems". https://upcommons.upc.edu/handle/2117/380697. Master's Thesis. Universitat Polytècnica de Catalunya, Oct. 2022.

[42] Philip E. Ross. "Brakes that Slam Themselves: Automatic emergency braking will become standard in Europe". In: *IEEE Spectrum* 59.1 (2022).

[43] AdaCore. *Introduction To SPARK*. https://learn.adacore.com/courses/intro-to-spark/index.html. 2023.

[44] AdaCore and Thales. *Implementation Guidance for the Adoption of SPARK*. https://www.adacore.com/uploads/books/pdf/Spark-Guidance-1.2-web.pdf. 2023.

[45] Dimitris Aspetakis. *Ada SPARK GPU Code Examples*. https://gitlab.bsc.es/dimitris_aspetakis/ada-spark-gpu. 2023.

[46] Dimitris Aspetakis. *GPU4S Bench implementations in Ada SPARK*. https://gitlab.bsc.es/dimitris_aspetakis/gpu4s-bench-ada. 2023.

[47] Codelabs GmbH. *Muen Separartion Kernel Code Repository*. https://git.codelabs.ch/?p=muen.git. 2023.

[48] Cristina Quesada Peralta, Matina Maria Trompouki, and Leonidas Kosmidis. "Evaluation of SYCL's Suitability for High-Performance Critical Systems". In: *Proceedings of the 2023 International Workshop on OpenCL*. IWOCL '23. Cambridge, United Kingdom: Association for Computing Machinery, 2023. DOI: 10.1145/3585341.3585378. URL: https://doi.org/10.1145/3585341.3585378.

[49] *Adacore website*. www.adacore.com/about-spark. Accessed: 2023-04-14.