Structured platform-aware programming for Rust

Francisco Heron de Carvalho-Junior
Departamento de Computação
Universidade Federal do Ceará
Fortaleza, Brazil
heron@dc.ufc.br

José Mykael Alves Nogueira
Campus de Quixadá
Universidade Federal do Ceará
Quixadá, Brazil
mykaelalves@alu.ufc.br

João Marcelo Uchôa de Alencar
Campus de Quixadá
Universidade Federal do Ceará
Quixadá, Brazil
joao.marcelo@ufc.br

ABSTRACT

Structured platform-aware programming provides performance engineers with the ability to develop optimized function variants that exploit specific features of target execution platforms in a modular way. In this paper, we introduce and evaluate an approach for structured platform-aware programming with Rust, a systems programming language that has gained attention in HPC.

KEYWORDS

platform-aware programming, programming languages, Rust, high-performance computing

1 Introduction

The broad interest in applications of artificial intelligence (AI) and Big Data analysis in industry and academia has pushed forward computer architecture technology towards an unprecedented level of heterogeneity and complexity, leading to the advent of processors with instruction subsets for addressing specific purposes, deep memory hierarchies, and accelerator devices whose architecture challenges the premises of the Von-Neumann architecture [14].

The need to harness the full potential of these computer systems presents challenges to both performance engineers and programming language designers. However, we observe a contradiction. While programming language designers seek new high-level programming abstractions to hide complexity and heterogeneity and promote code portability, performance engineers seek ways to explore specific features from a potentially wide range of target platforms, often giving preference to platform-specific programming interfaces, resulting in code that is highly complex and difficult to maintain and evolve due to modularity anomalies [3].

We define platform-aware programming as the performance engineering activity in which the programmer writes code that makes assumptions about the specific characteristics of the target execution platform using a high-level programming language [3, 4]. We emphasize the importance of programming language designers and researchers proposing platform-aware programming languages or language extensions that incorporate abstractions explicitly addressing the heterogeneity inherent in execution platforms, rather than attempting to conceal such heterogeneity behind generic abstractions. As an example of such an effort, PlatformAware.jl[1] is an experimental package we developed for structured platform-aware programming using the Julia programming language [4]. Some Julia features have simplified the implementation, including support for meta-programming, just-in-time (JIT) compilation, dynamic multiple dispatch, and a rich type system with subtyping.

[1]https://github.com/PlatformAwareProgramming/PlatformAware.jl

In this work, we extend the Rust programming language with structured platform-aware programming through the platform-aware crate. Rust is a statically typed programming language that has gained popularity as a systems programming language. Rust is also considered a promising alternative in high-performance computing (HPC) due to its performance compared to C and Fortran, while introducing memory safety features that can prevent execution errors in long-running computations. However, it lacks the Julia features listed in the last paragraph that make structured platform-aware programming easier to implement without modifying the programming language compiler and/or runtime system. For these reasons, Rust is another effective exercise in introducing structured platform-aware programming into existing programming languages, complementing the work done with Julia.

We present a proof-of-concept case study using the CG kernel of NPB-Rust[2], an unofficial multithreading implementation of the NAS Parallel Benchmarks (NPB) [2] implemented by GMAP/PUCRS using Rust. It demonstrates how the platform-aware crate makes it possible to select between multicore and CUDA-based GPU execution [10] at the beginning of the execution. Additionally, the CUDA code, if selected, may vary depending on the compute capability of the underlying GPU and the version of the CUDA driver. Finally, a performance study has been performed to motivate a discussion on the impact of using platform-aware on CG's performance.

The remainder of the paper consists of four additional sections. Section 2 provides background and related works on platform-aware programming and the Rust programming language. Then, Section 3 shows how structured platform-aware programming has been implemented in Rust through the platform-aware crate. In Section 4, the proof-of-concept and performance study using NPB-Rust/CG is presented. Finally, Section 5 concludes the paper by summarizing its contributions and presenting lines for further work.

2 Background and related works

In this section, we introduce structured platform-aware programming, present the Rust programming language, and provide an overview of related works on the structured support of platform-aware programming in programming languages.

2.1 Platform-aware programming

Platform-aware programming is the performance engineering practice of coding by making assumptions about features of the execution platform to face HPC and system programming requirements [3]. For example, programmers may implement functions through low-level code that makes optimized use of vector registers

[2]https://github.com/GMAP/NPB-Rust

through SIMD instruction set extensions, or they can tie the code of a function to a specific GPU backend (e.g., CUDA, oneAPI, and ROCm) or backend version to access particular features of a GPU of a given model that is available in the execution environment. Also, at a higher level, parallel programmers may write code that selects between alternative parallel algorithms that fit the scalability features of the underlying parallel computing platform [6]. This is particularly useful in distributed-memory parallel computing platforms, such as clusters and Massive Parallel Processors (MPPs).

Platform-aware programming may be *static* or *dynamic*. In the former, programmers use directives and macros to guide compilers in generating platform-specific code, requiring recompilation if platform features change. In the latter, the binary code is portable across platforms, and runtime checks are used to switch between platform-specific code. However, the dynamic discovery of platform features is still challenging. Dynamic platform-aware programming is *ad-hoc* when programmers lack support from programming language abstractions to discover platform features and write different function versions to take advantage of them.

Attempts to perform platform-aware programming through *ad hoc* means become even more challenging as computer architectures become increasingly heterogeneous. For this reason, programming language designers attempt to unify programming interfaces through high-level abstractions, thereby circumventing platform-aware programming and avoiding its complexity when designing programming languages [11]. Conversely, we are interested in non-ad hoc practices of dynamic platform-aware programming.

Structured platform-aware programming has been introduced and implemented in the Julia programming language [4] through the PlatformAware.jl package [3]. Using this package, Julia programmers can implement different methods for a so-called kernel function for various feature combinations of the underlying execution platform. The method to be applied in a kernel function invocation is selected through Julia's multiple dispatch mechanism, which operates over platform types embedded in Julia's type system. In this paper, we bring structured platform-aware programming to Rust, a language with markedly different characteristics from Julia.

2.2 The Rust programming language

The history of Rust [9] dates back to 2006, when Graydon Hoare, a software engineer for the Mozilla Foundation, developed it as a personal project. The memory safety features of Rust attracted the attention of Mozilla's executives, who started sponsoring Rust in 2009, allocating a team of engineers led by Hoare to evolve and implement it.

The first stable release of Rust was launched in 2015, consolidating its main features, the most prominent of which is memory safety without garbage collectors by implementing ownership types. Such a design achieves high execution performance compared to C and provides memory safety comparable to statically typed functional languages. Addressing these two requirements simultaneously was treated as contradictory in the design of earlier programming languages. Ownership types also bring *fearless concurrency* to Rust, preventing data race conditions and other concurrency issues.

[3]https://github.com/PlatformAwareProgramming/PlatformAware.jl

The syntax of Rust draws inspiration from C++ and functional programming languages, such as OCaml, incorporating characteristics of object-oriented languages through *structs*, *enums*, *traits*, and *methods*. Since 2014, it has been evolving through a Request for Comments (RFC) process to introduce new features, accepting contributions from the open-source community worldwide[4].

The popularity of Rust is increasing rapidly in the developer community. Since its initial release, it has been applied in several large projects, including those of major companies. This has motivated a set of companies, including Amazon Web Services (AWS), Google, Huawei, and Microsoft, to join Mozilla as sponsors in the creation of The Rust Foundation[5] in 2021.

2.2.1 *In system programming.*

Due to its zero-cost abstractions and memory safety, Rust has been introduced as a systems programming language. For instance, it is the first language besides C to be officially supported in the Linux kernel. Also, Microsoft has written several parts of the Windows kernel in Rust. Together with HPC, systems programming is another potential application of platform-aware programming. It leverages portability, since system programs commonly interface with hardware elements.

2.2.2 *In HPC.*

The reconciliation of high-performance and high-level abstractions makes Rust a natural choice for meeting the modern requirements of HPC applications. While its performance in serial execution is compared to C and Fortran, Rust offers modern programming abstractions to handle complex and large-scale code more effectively than these languages. Memory safety is also a significant concern, particularly in long-running computations, where execution errors can be costly for users.

Today, however, the use of Rust in HPC faces a bottleneck in the still incipient and immature availability of packages aimed at typical HPC applications in its ecosystem, including packages for accessing accelerator features (e.g., GPUs) from the leading manufacturers, as well as packages for AI applications, whose interest in HPC platforms has grown considerably. However, this context is changing rapidly due to the growing interest in using Rust to leverage traditional AI and HPC applications.

2.3 Related works

The programming languages research community has given limited attention to explicit support for platform-aware programming through abstractions and programming language constructs. Among the initiatives reported in the literature, we highlight our previous work with PlatformAware.jl [4], for the support of structured platform-aware programming in Julia, which has a direct relation to this work. Also, the *multivariant user functions* of SkePU, a skeleton programming framework, deserve attention [5].

In language designing, two initiatives deserve mention: the *function multi-versioning*[6] feature from the GNU C compiler (GCC) 4.8 and the *target features* RFC for Rust [13]. Both attempt to provide support for different versions of functions according to specific-purpose instruction sets supported by the target architecture during compilation, such as SIMD instruction sets (e.g., SSE1-4, AVX2,

[4]https://rust-lang.github.io/rfcs
[5]https://rustfoundation.org
[6]https://gcc.gnu.org/onlinedocs/gcc/Function-Multiversioning.html

AVX512). These are fully static approaches, in which the binary code generated by the compiler is not platform-aware. In contrast, the approach presented in this paper is dynamic, selecting function variants at program startup. Furthermore, it can be applied to a broader range of features beyond SIMD instruction sets.

Works on configurable systems and feature models on software product lines also deserve mention [1]. Although they are generally agnostic about programming languages, tuning software dynamically according to the target execution platform's features is of interest in this area [12]. This includes selecting between function implementation variants, the approach addressed in this paper. We highlight works on performance models for the selection of alternative configurations, which can be applied dynamically fed by information about platform features, where, in recent years, techniques of machine learning have also been introduced [7, 16].

3 Platform-aware programming in Rust

This paper introduces the support for structured platform-aware programming in Rust through the platform-aware crate[7]. The rest of this section shows how Rust programmers can use it, how it has been implemented, and how it can be extended.

In platform-aware programming, a program is composed of a set of *kernel components* with coexisting implementations that differ in their assumptions about the features of the underlying computing platform. A kernel component may be any composition element, such as functions, subroutines, modules, or classes, depending on the underlying programming language's paradigm and features.

Listing 1: A simple example of kernel module

```
#[platformaware(my_kernel_function)]
mod my_kernel_module {
   #[kernelversion] // fallback version (no platform arguments)
   fn my_kernel_function(x:i64, y:i64, z:i64) { ... } // v0

   #[kernelversion(cpu_simd=AVX512, cpu_core_count=AtMost{val=4})]
   fn my_kernel_function(x:i64, y:i64, z:i64) { ... } // v1

   #[kernelversion(cpu_simd=AVX512,
                   cpu_core_count=AtLeast{val=4})]
   fn my_kernel_function(x:i64, y:i64, z:i64) { ... } // v2

   #[kernelversion(acc_model=NVIDIA_H100)]
   fn my_kernel_function(x:i64, y:i64, z:i64) { ... } // v3

   #[kernelversion(acc_count=AtLeast{val=2}, acc_backend=CUDA,
                   acc_cudacc=AtLeast{val=90})]
   fn my_kernel_function(x:i64, y:i64, z:i64) { ... } // v4

   #[kernelversion(acc_backend=CUDA, acc_cudacc=AtLeast{val=80})]
   fn my_kernel_function(x:i64, y:i64, z:i64) { ... } // v5

   #[kernelversion(acc_count=2, acc_model=AMD_MI325X)]
   fn my_kernel_function(x:i64, y:i64, z:i64) { ... } // v6
}
```

For Rust, kernel components are functions, referred to as *kernel functions* or *k-functions*. Listing 1 outlines the code of a simple k-function named `my_kernel_function` with six versions (`v0`, `v1`, `v2`, `v3`, `v4`, `v5`, and `v6`). A k-function must be implemented in a module, referred to as a *kernel module* (*k-module*), which is introduced by the `platformaware` macro, listing the names of the k-functions declared in its scope. In the kernel module, each version of the k-function is introduced by the `kernelversion` macro, where each argument defines an application of a *platform feature* as a *platform argument* to a *platform parameter* identified by a name. For example, in `v2`, `AVX512` and AtLeast{val=4} are platform arguments applied to the platform parameters cpu_simd and cpu_core_count, respectively. The set of platform arguments defines the assumption for implementing the k-function version. The first one declared is the *fallback* version, which does not declare any assumptions because it must be executed on any computing platform.

[7]https://github.com/PlatformAwareProgramming/platform-aware

main processor		
cpu_count	<: AtLeast{val=1}	quantifier
cpu_core_count	<: AtLeast{val=1}	quantifier
cpu_threads_count	<: AtLeast{val=1}	quantifier
cpu_model	CPUModel	qualifier
cpu_microarchitecture	CPUMicroarchitecture	qualifier
cpu_isa	CPUISA	qualifier
cpu_simd	CPUSIMD	qualifier
cpu_L1_size	<: AtLeast{val=0}	quantifier
cpu_L1_latency	<: AtMost{val=∞}	quantifier
cpu_L2_size	<: AtLeast{val=0}	quantifier
cpu_L2_latency	<: AtMost{val=∞}	quantifier
cpu_L3_size	<: AtLeast{val=0}	quantifier
cpu_L3_latency	<: AtMost{val=∞}	quantifier
accelerator		
acc_count	<: AtLeast{val=0}	quantifier
acc_type	<: ACCType	qualifier
acc_backend	<: ACCBackend	qualifier
acc_manufacturer	<: ACCManufacturer	qualifier
acc_model	<: ACCModel	qualifier
acc_architecture	<: ACCArchitecture	quantifier
acc_memorysize	<: AtLeast{val=0}	quantifier
acc_cudacc	<: AtLeast{val=10}	quantifier
acc_cudatoolkit	<: AtLeast{val=10}	quantifier
acc_cudadriver	<: AtLeast{val=100}	quantifier
memory		
mem_type	MEMType	qualifier
mem_latency	AtMost{val=∞}	quantifier
mem_bandwidth	AtLeast{val=0}	quantifier
mem_size	AtLeast{val=0}	quantifier

Table 1: Platform parameters

There is no coding restriction for implementing k-function versions. It is regular Rust programming. So, the programmer can utilize any allowed means to access specific platform features. Indeed, as in the case study of Section 4, it can make calls to lower-level code in other languages, such as C, C++, and Fortran, through the Foreign Function Interface (FFI).

At the beginning of execution, each kernel module runs a *resolution algorithm* over the assumptions of all k-function versions declared in its scope to select a k-function whose assumption is compatible with the features of the underlying computing platform, known as the *actual platform feature set*. For example, suppose an actual platform feature set for a computing platform with two Intel Xeon processors, each with 20 cores, 128 GB of memory, and four NVIDIA H100 GPUs. For this computer, the k-functions `v0`, `v2`, `v3`, `v4`, and `v5` are compatible. Section 3.3 explains how the resolution algorithm works and why `v4` is selected.

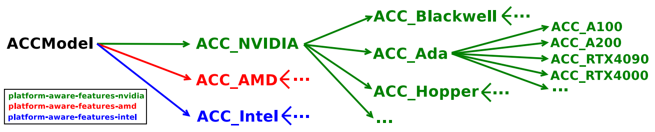

Figure 1: Feature hierarchy for `acc_model`

3.1 Platform features

Table 1 lists the names of the platform parameters that can be used in applications of the `kernelversion` macro, along with their *typing restrictions*. Each platform parameter denotes a hierarchy of platform features that can be applied to it as a platform argument, as illustrated in Figure 1 for acc_model, focusing NVIDIA devices.

Platform features may be viewed as types, with a subtyping relation that specifies specialization/generalization relations between them, where the typing restriction associated with the platform parameter in Table 1 is the *top type*, i.e., the type that is a supertype of any platform feature applied (as a platform argument) to the platform parameter. The set of platform arguments of a k-function version denotes its *assumption type*, henceforth referred to as a *platform assumption*.

A platform parameter may receive quantifier or qualifier features as platform arguments, as pointed out in Figure 1. In Rust, qualifiers are platform features implemented as unit-like structs implementing the `Feature` trait. Examples are `AVX512` and `CUDA`, arguments of `cpu_simd` and `acc_backend` in `v2` and `v4`, respectively. In turn, quantifier features may be integers (e.g., `acc_count=2` in `v6`), or `AtLeast`/`AtMost` structs (e.g., `acc_count=AtLeast{val=2}` in `v4`, `cpu_core_count=AtMost{val=4}` in `v1`, and `acc_cudacc=AtLeast{val=80}` in `v5`), with a subtyping relation inspired in *refinement types* [15].

The currently supported qualifiers are declared in pre-existing crates platform-aware-intel, platform-aware-amd, and platform-aware-nvidia, for three processor and accelerator manufacturers, respectively, Intel, AMD, and NVIDIA. Qualifiers for other manufacturers, or according to some different criteria, may be added in third-party crates, as explained in Section 3.5.

Implementing the feature hierarchy (subtyping). Since Rust does not support nominal subtyping between `structs`, we have implemented the feature hierarchy through the Feature trait in the `platform-aware-features` crate. Each `struct` that denotes a feature must implement `Feature` and supply an implementation for the `supertype` function, pointing to the feature that is its direct supertype. The `Feature` trait has a default implementation for a `subtype` function, which defines the subtyping relation for qualifiers, using the `supertype` function, and quantifiers. It is a key component of the resolution algorithm.

3.2 Full quantifiers

The quantifier platform features enable the specification of assumptions for numerically valued platform parameters.

Let n be a value of `i32` type. A quantifier may be an integer n, denoting a singleton set including n, `AtLeast{val:`n`}`, denoting all integers greater or equal to n, or `AtMost{val:`n`}`, denoting all integers less or equal to n. The following rules specify the subtyping relation (`<:`) for quantifiers:

- `AtLeast{val:`m`}` <: `AtLeast{val:`n`}` iif $m \geq n$;
- `AtMost{val:`m`}` <: `AtMost{val:`n`}` iif $m \leq n$;
- m <: `AtLeast{val:`n`}` iif $m \geq n$;
- m <: `AtMost{val:`n`}` iif $m \leq n$;

This proposal for implementing quantifiers in Rust addresses a limitation of PlatformAware.jl, as it supports *full quantifiers*, allowing for the direct use of numeric values in quantifiers. In Julia, numeric values in quantifiers are implemented as a combination of two qualifiers, one denoting a power of two quantity $(2, 4, 8, \ldots, 512)$ and the other denoting a magnitude multiplier (`n`, `u`, `m`, `K`, `M`, `G`, `T`, `P`, and `E`), so that `AtLeast{val=`2^8`}` represents `AtLeast256M` in Rust.

3.3 The resolution algorithm

In Julia, platform types are implemented as Julia types to take advantage of dynamic multiple dispatch, which selects one of the versions of a kernel function, thereby implementing the so-called *resolution algorithm*. The subtyping and 'types as values' features of Julia also motivated this approach. However, it leads to limitations in expressiveness. For example, full quantifiers cannot be supported, since the Julia type system does not support dependent and refinement types. For Rust, such a limitation does not exist because platform types are implemented as values, since Rust does not support dynamic multiple dispatch. Such an approach makes full quantifiers possible but requires implementing the resolution algorithm from scratch.

The `resolve` function is implemented in the `resolve.rs` module of the platform-aware-features crate. It takes as input the actual platform feature set and an array of assumptions (feature sets) for each kernel function version. The algorithm first looks for the first kernel version whose assumption K_0 is a supertype of the actual platform feature set P in inverse declaration order. If one is found, it continues to check the subsequent kernel versions to improve the selection by building a chain of n kernel versions with assumptions such that $P <: K_n <: \ldots <: K_1 <: K_0$, where <: denotes the subtype relation. The kernel function version with the K_n assumption is the one selected. Note that K_n makes the strongest assumption compatible with the actual platform feature set.

In the example presented in the introduction to this section, `v5` is the first k-function version selected, as H100 GPUs support compute capability 10.0. So, it supports compute capability 8.0 by backward compatibility. However, the assumption of `v4` is not only compatible with the actual platform feature set but also a subtype of the assumption of `v5`, since compute capability 9.0 is closer to the compute capability of H100 GPUs than 8.0 (`AtLeast{val=90} <: AtLeast{val=80}`). Additionally, since the actual platform feature set declares four GPUs, it satisfies `v4`, which requires at least two. Therefore, the code of `v4` is certainly prepared to take advantage of multiple GPUs, while that of `v5` may not be. `v3` is also compatible with the actual platform feature set. It refers to the exact GPU model, but its assumption is not a subtype of `v4`'s assumption. `v2` is also compatible, since it has more than eight processor cores, but it is not a subtype of the assumption of `v4` either. So, `v4` is selected.

This algorithm has been designed to give programmers control of which kernel version has priority when the assumptions of one or more kernel versions that are compatible with the actual platform feature set overlap. In the Julia approach, overlapping assumptions cause ambiguity errors due to the semantics of dynamic multiple dispatch. For Rust, programmers are encouraged to order kernel versions from the most general to the most specific assumptions, so that if assumptions of kernel versions overlap, the one declared later is selected, avoiding ambiguity issues.

3.4 Platform feature detection

Automatically detecting the actual platform feature set is still a challenge. It lacks a standard, portable way to communicate with hardware elements of different purposes from different manufacturers to collect their features of interest for platform-aware programming. Currently, this can be done using various tools that strongly depend on operating system support, making portability difficult. We argue that this is the main threat to the explicit support of platform-aware programming in programming languages.

In Rust, we adopt the same static approach implemented in Julia, employing a *platform description file* specifying the actual platform feature set. Indeed, the same configuration file format, i.e., TOML[8], is used for the platform file, named Platform.toml by default.

Despite being a static approach, the `Platform.toml` file offers flexibility, allowing it to be human-readable and editable, so that users of platform-aware programs can easily create and modify it on their platforms. However, nothing prevents the development of tools to assist these users in creating platform description files.

Finally, since Rust is statically compiled, the compiler could consider the `Platform.toml` to generate code specific to a particular platform or class of platforms through a compilation flag.

3.5 Extending features

The platform-aware-features crate offers the basic functionality for implementing feature sets to be used in structured platform-aware programs in Rust, like those provided by the pre-defined crates platform-aware-features-intel, platform-aware-features-amd, and platform-aware-features-nvidia. Therefore, anyone can create similar crates that offer feature sets according to specific criteria, or extend the feature set of an existing one. Platform-aware developers may import such third-party crates into their code to benefit from new or extended feature sets, probably needing to write feature detection code to help their users create Platform.toml files.

4 Case study

For a proof-of-concept evaluation of the platform-aware crate, the serial and parallel implementations of the CG kernel of NPB-Rust [8], henceforth referred to as NPB-Rust/CG, have been employed. NPB-Rust is a Rust implementation for the NAS Parallel Benchmarks (NPB) developed by the GMAP[9] research group of PUC-RS. The parallel implementation exploits multi-threaded data-parallelism through the rayon crate.

The NAS Parallel Benchmarks (NPB). NPB is a benchmark suite to evaluate the performance of parallel computers in CFD applications, first proposed at the beginning of the 1990s by the NASA Advanced Supercomputing (NAS) Division [2]. It was initially implemented in C and Fortran, comprising five kernels and three simulated applications. It gained high popularity not only for evaluating parallel computing platforms but also for assessing the performance of programming languages designed for scientific computing, especially their abstractions, constructs, and libraries that support parallelism. Consequently, new official and unofficial implementations in other languages have been developed and publicized over the years, along with new kernels and standard instance classes to meet the new requirements of emerging parallel computing platforms. CG is an NPB kernel that implements the conjugate gradient method to exploit irregular memory access and communication in benchmarking parallel computers and languages.

alloc_a_d alloc_colidx_d alloc_rowstr_d	allocates the input sparse matrix (a), using the CRS format, in the device
move_a_to_device	move a from the host to the device
alloc_x init_x update_x	allocate the vector x and returns its opaque pointer initialize the vector x to $(1, 1, \dots, 1)$ update x after each *conjgrad* iteration
alloc_p	allocates the vector p and returns its opaque pointer
alloc_q	allocates the vector q and returns its opaque pointer
alloc_r	allocates the vector r and returns its opaque pointer
alloc_z	allocates the vector z and returns its opaque pointer
freevectors	deallocates all the vectors from host or device memory
init_conj_grad	initializes the vectors for a *conjgrad* routine

Table 2: Auxiliary kernel functions

In general terms, CG is an iterative code that performs linear algebra operations over a collection of input and auxiliary vectors (x, z, p, q, and r) and a sparse matrix (a) in the CSR (Compressed Sparse Row) format. The platform-aware crate has been used to implement these operations by making assumptions on the features of the underlying computing platform. In particular, regarding the support for multi-threading parallelism in the presence of multiple processing cores and acceleration by offloading kernels to GPUs that support CUDA. For this purpose, the original NPB-Rust code, which follows the same structure as the reference implementation in Fortran developed by NAS, has been refactored to encapsulate linear algebra operations in functions, as they were inlined in the original code. These functions constitute the set of k-functions for platform-aware programming purposes. They are:

- matvecmul, which multiplies a sparse matrix a in CRS format and a vector x and put the result in a vector y;
- vecvecmul, which performs the dot product of two vectors x and y, so returning a scalar;
- scalarvecmul1, which takes a scalar α and two vector x and y to perform $y = x + \alpha \times y$;
- scalarvecmul2, which takes a scalar α and two vector x and y to perform $y = y + \alpha \times x$;
- norm, which returns the norm of vectors x and y ($\sqrt{x^2 - y^2}$).

In NPB-Rust, CG utilizes the rayon data-parallelism library to implement the kernels, leveraging multiple processor cores. We add the offloading of kernel functions to CUDA GPUs whenever they are available and the problem class fits the GPU's memory space.

To support accelerator devices, such as GPUs, efficiently, all vectors and matrices must be initially allocated and initialized on the device, thereby minimizing data copies between the host and the device during computation. This is the reason to execute all the linear algebra kernels on the acceleration device, despite almost all

[8]TOM's Obvious Minimal Language, https://toml.io
[9]https://gmap.pucrs.br

serial (default)	`#[kernelversion]`
multithreading	`#[kernelversion(cpu_core_count=(AtLeast{val:2}))]`
CUDA (ver. 0)	`#[kernelversion(acc_count=(AtLeast{val:1}),` `acc_backend=CUDA,` `acc_cudacc=(AtLeast{val:13}),` `acc_cudatoolkit=(AtLeast{val:20}),` `acc_cudadriver=(AtLeast{val:17713}))]`

Table 3: Common versions of computation kernel functions

ver.	cc	CUDA toolkit	CUDA driver	highlights
0	1.3	2.0	177.13	• all NVIDIA GPUs that suppport FP64 arithmetics.
1	3.5	5.0	418.39	• optimize global memory access using `__ldg()`; • prefetching vector x using `#pragma unroll`.
2	6.0	10.1	384.81	• uses the cuSPARSE library.
3	7.0	9.0	319.37	• one warp per row; • optimize global memory access using `__ldg()`; • uses a warp level primitive: `__shfl_down_sync`.

Table 4: Kernel version for matvecmul

class	platform	execution time★ L	S	M	B	speedup L	S	M	B
A	***serial***	1460	410	441	550	1.0	1.0	1.0	1.0
	multicore	440	220	158	190	3.3	1.9	2.8	2.9
	CUDA 0	7980	220	50	50	0.2	1.9	8.9	11.3
	CUDA 1	–	250	48	50	–	1.6	9.3	11.9
	CUDA 2	–	160	48	50	–	2.5	9.2	12.1
	CUDA 3	–	140	38	40	–	3.0	11.7	15.0
B	***serial***	98.3	21.8	22.2	25.3	1.0	1.0	1.0	1.0
	multicore	21.5	14.9	8.7	13.3	4.6	1.5	2.5	1.9
	CUDA 0	55.0	8.6	3.0	1.1	1.8	2.5	7.3	23.4
	CUDA 1	–	10.0	3.0	1.0	–	2.2	7.4	25.2
	CUDA 2	–	5.4	1.2	0.6	–	4.0	18.2	42.9
	CUDA 3	–	4.5	1.1	0.6	–	4.8	20.4	41.2
C	***serial***	279.1	72.0	58.7	62.9	1.0	1.0	1.0	1.0
	multicore	57.6	35.4	24.7	36.9	4.8	2.0	2.4	1.7
	CUDA 0	85.5	43.8	7.8	2.4	3.3	1.6	7.5	26.6
	CUDA 1	–	45.0	7.8	2.3	–	1.6	7.5	27.8
	CUDA 2	–	22.5	2.9	1.3	–	3.2	20.0	49.5
	CUDA 3	–	24.9	2.6	1.3	–	2.9	22.6	49.3
D	***serial***	17710	5522	1990	2224	1.0	1.0	1.0	1.0
	multicore	3308	2028	637	490	5.4	2.7	3.1	4.4
	CUDA 0	–	–	215	51	–	–	9.2	43.6
	CUDA 1	–	–	207	50	–	–	9.6	44.3
	CUDA 2	–	–	72	28	–	–	27.8	80.8
	CUDA 3	–	–	63	26	–	–	31.6	84.7

★ in *milliseconds* (ms) for class A, and *seconds* (s) for the other classes.

Table 5: CG experimental cases

the execution time being spent on matvecmul. The only exception is the sparse matrix a, which is generated in the host and then copied to the device. For this purpose, the code that creates and initializes each vector is encapsulated in auxiliary kernel functions, with default versions that allocate them on the host and specific versions that allocate them on each type of accelerator device. Table 2 presents the list of these auxiliary kernel functions.

In the current implementation, while the auxiliary kernel functions have two versions — the default and GPU ones — each computation kernel function has three versions in common, as depicted in Table 3. Therefore, the CUDA kernel version is compiled with CUDA Toolkit 2.0 and can be executed on an older GPU that supports compute capability 1.3, with a minimum CUDA driver version 177.13. Such an old GPU would hardly still be in use. However, the purpose of this case study is to explore the limits of platform-aware's ability to express execution restrictions for different versions of a kernel for NVIDIA GPUs, the most popular.

Since it is the kernel with the most significant impact on CG's execution time, matvecmul supports three additional kernel versions, depicted in Table 4, optimized for more recent GPUs. While versions 0, 1, and 3 are directly programmed in CUDA, version 2 employs the cuSPARSE library offered by NVIDIA. In Appendix A, there is a fragment of cg.rs including the declarations of the matvecmul versions. They are declared in the order of their performance achieved in the experimental evaluation discussed in Section 4.1, from the lower to the higher performance. Such an order coincides with ordering by the minimal compute capability required by these kernels, which defines the set of features supported by the GPU. The code in Appendix B presents the code of the C function launch_matvecmul_cuda, called in line 30 of Appendix A. It launches the CUDA kernel version 0 for matvecmul on the GPU.

4.1 Testbed environment

An experiment has been performed to study the performance of a platform-aware program (NPB/CG) on a testbed environment comprising four computing platforms identified by **legacy** (**L**), **small** (**S**), **medium** (**M**), and **big** (**B**), whose platform features are described in their respective Platform.toml files. Their main features are summarized below:

- **L** is a workstation with an Intel Xeon CPU E5-2620 v2 (6 cores), a NVIDIA GeForce GTX 760 legacy GPU (2GB GDDR3), and 32 GB RAM, commissioned in 2013;
- **S** is a Dell Inspiron 14 5440 laptop equipped with a Core 7-150U processor (12 cores), a NVIDIA GeForce MX570A GPU (2GB GDDR6), and 32GB RAM;
- **M** is a workstation equipped with an AMD Ryzen 7 5700X processor (8 cores), an NVIDIA Quadro RTX 4000 Ada Generation (20GB GDDR6), and 128GB RAM;
- **B** is a node of the Apollo cluster at the CENAPADUFC[10] infrastructure, equipped with two AMD EPYC 7513 processors (32 cores), an NVIDIA A100 GPU (80 GB HBM2e), and 512GB RAM.

The computing platforms have been deliberately selected to cover different application scenarios. Indeed, **L** is an old workstation with a GPU launched more than a decade ago. In turn, **S**, **M**, and **B** are modern GPU-based computing platforms of different classes and purposes: a simple mobile computer, a personal workstation aimed at graphical and technical computing applications, and a node of a high-end cluster aimed at computational sciences and artificial intelligence applications, respectively.

4.2 Results and discussion

For the computing platforms **L**, **S**, **M**, and **B**, Table 5 presents performance measures for problem classes from **A** to **D** of the platform-aware version of NPB-Rust/CG for its possible execution contexts based on the contents of their platform description files: ***serial***, for acc_count = 0 and cpu_core_count = 1; ***multicore***, for acc_count = 0 and cpu_core_count > 1; and ***CUDA*** i, where $i \in \{0, 1, 2, 3\}$, for acc_count > 0. Note that problem class D does not fit the GPU's memory size of **L** and **S** computing platforms.

CUDA 0, ***CUDA*** 1, ***CUDA*** 2, and ***CUDA*** 3 are the four execution contexts using a single GPU for the matvecmul kernel specified in

[10]https://cenapad.ufc.br

class	serial (s)	32 cores (s)	speedup	efficiency	
				OpenMP	*rayon**
A	0.55	0.03	18.4	58%	9%
B	24.9	1.0	24.8	77%	6%
C	65.9	3.1	21.4	67%	5%
D	2345.1	102.5	22.9	71%	14%

*using the speedups for the Rust/rayon version from Table 5.

Table 6: Multicore efficiency for NPB/CG on the B computing platform (FORTRAN/OpenMP)

Table 4. In the experiment, the platform description file for each platform was modified to execute all possible scenarios for the platform, demonstrating the performance gains derived from the platform-aware approach. Except for the legacy platform, whose GPU is compatible only with ***CUDA*** 0 due to its low compute capability (acc_cudacc = 30), the other platforms, with modern GPUs, are compatible with all CUDA versions, since acc_cudacc > 70 for all of them.

The performance study has also been essential to justify the order in which the second and third kernel versions are declared for matvecmul. Both are state-of-the-art implementations, but their requirements, based on compute capabilities and versions of the CUDA driver and toolkit, conflict; i.e., neither is a subtype (specialization) of the other. As the third version presents better performance for all testbed GPUs, it is declared after the second version.

The results demonstrate the performance variability of NPB-Rust/CG, a program with HPC requirements, according to the assumptions of its implementation regarding the features of the underlying computing platform. The speedup obtained by using platform-aware optimized kernel functions increases with the size of the problem class, which is evidence of scalability. Moreover, as expected, the speedup is higher for computing platforms equipped with GPUs, which present higher processing capacity, reaching 84.7 for **B** on an NVIDIA A100 in class D. In contrast, **M** achieves a 31.6 speedup on an NVIDIA RTX 4000 Ada for the same class.

The case study shows how NPB-Rust/CG uses the platform-aware crate to include multiple versions of performance-critical kernel functions in a single executable, without sacrificing source code modularity despite the platform-aware expression problem [3]. The code in Appendix A illustrates this. This facilitates the addition or removal of kernel versions as the application evolves.

In addition, the platform-aware version of NPB-Rust/CG can adapt, without recompilation, to changes in the computing platform (e.g., add, remove, or update accelerators) that modify the features on which the program makes assumptions to implement the k-function versions. To do so, it is only necessary to reflect those changes in the platform description file.

The fact that **M** has outperformed **B** in serial execution may come as a surprise at first glance, but it is a known fact that the AMD Ryzen 7 5700X processor of **M** achieves higher performance in single-core execution than the AMD EPYC 7513 processor of **B**[11]. Also, the overhead of multithreading execution has been greater on the **B** platform, where rayon achieved the shortest execution time using all the 32 cores only for problem class D, while more than two threads led to a decrease in execution time for the other classes. On the other hand, for all classes, the shortest multithreading execution time for **M** was achieved using all eight cores, leading to higher speedup compared to **B**, although marginally.

[11]https://www.cpu-monkey.com/en/compare_cpu-amd_ryzen_7_5700x-vs-amd_epyc_7513

It is worth discussing the low efficiency achieved with rayon compared to the efficiency gained using OpenMP for the reference implementation in FORTRAN. This is depicted in Figure 6 for the **B** execution platform. The efficiency metric is calculated by dividing the speedup by the number of cores of **B** for which the reference FORTRAN/OpenMP implementation achieves the minimum execution time (*baseline*), to isolate parallelism overheads due to fine-grained data parallelism in rayon. In this way, we have a more realistic comparison between the speedup obtained using the higher-level approach of rayon and the lower-level approach of OpenMP. In fact, while FORTRAN's serial times are similar to those obtained with Rust, the multithreading times using OpenMP are significantly lower than for rayon.

This scenario demonstrates the overhead of high-level parallelism approaches. In contexts like HPC, where efficient use of computational resources is critical, such approaches — e.g., rayon — often prove inadequate. As a result, performance engineers resort to low-level or ad hoc platform-aware solutions, despite their complexity. This underscores the need for structured platform-aware programming to support better software engineering practices in performance-critical code.

5 Conclusions

To meet the growing demand from Big Data analytics and AI applications, the computer industry has addressed conflicting goals of achieving both peak performance and energy efficiency. This has led to increasingly complex and heterogeneous high-end systems that challenge, or even surpass, the Von Neumann architecture. Such complexity presents difficulties for programming language designers, who aim to abstract it through high-level constructs and compilation techniques. In contrast, performance engineers often rely on platform-aware programming, a strategy that exploits low-level, platform-specific features of the target system.

Platform-aware programming remains largely ad hoc, with minimal support in existing languages. This paper presents an implementation of structured platform-aware programming in Rust, extending prior work done in Julia. Unlike Julia, Rust features static compilation and a strong, subtype-free type system designed for memory safety. Given Rust's growing role in systems and high-performance computing, domains where platform-aware programming is essential, developers may benefit from the platform-aware crate, the main artifact introduced in this paper.

In a proof-of-concept case study using the NPB-Rust CG kernel, we demonstrated how the platform-aware crate enables structured platform-aware programming. The case involved selecting between basic and state-of-the-art CUDA kernels for accelerating CG computations - specifically, the sparse matrix-vector multiplication in CSR format - based on the GPU's compute capability and the installed CUDA version. The study also highlighted the overhead of using Rust's high-level multithreading library (rayon) compared to OpenMP in the original Fortran implementation.

In the ongoing development of the platform-aware crate, we intend to add new functionalities and platform features. For example, programmers could encode domain-specific assumptions in k-functions, like problem classes in NPB-Rust/CG, so that GPU kernels are applied only when memory requirements are met. This will also allow modular crates like platform-aware-intel, platform-aware-nvidia, and platform-aware-amd to define platform-specific parameters. For example, NVIDIA-specific parameters such as acc_cudacc, acc_cudadriver, and acc_cudatoolkit are still declared in the main platform-aware crate instead of platform-aware-nvidia.

REFERENCES

[1] S. Apel, D. Batory, C. Kstner, and G. Saake. 2013. *Feature-Oriented Software Product Lines: Concepts and Implementation.* Springer Publishing Company, Incorporated.

[2] D. H. Bailey and et al. 1991. The NAS Parallel Benchmarks. *International Journal of Supercomputing Applications* 5, 3 (1991), 63–73.

[3] F. H. de Carvalho Junior. 2024. The Expression Problem in Platform-Aware Programming. In *XXVIII Brazilian Symposium on Programming Languages (SBLP'2024)* (Curitiba, PR). SBC, Porto Alegre, Brazil.

[4] F. H. de Carvalho Junior, A. B. Dantas, J. M. Hoffiman, T. Carneiro, C. S. Sales, and P. A. S. Sales. 2023. Structured Platform-Aware Programming. In *XXIV Simpósio em Sistemas Computacionais de Alto Desempenho (SSCAD'2023)* (Porto Alegre, RS). SBC, Porto Alegre, Brazil, 301–312.

[5] A. Ernstsson and C. Kessler. 2020. *Parallel Computing: Technology Trends.* Advances in Parallel Computing, Vol. 36. IOS Press, Amsterdam, Chapter Multi-Variant User Functions for Platform-Aware Skeleton Programming, 475–484. doi:10.3233/APC200074

[6] A. Grama, A. Gupta, J. Karypis, and V. Kumar. 2003. *Introduction to Parallel Computing.* Addison-Wesley. 256 pages.

[7] H. Ha and H. Zhang. 2019. DeepPerf: Performance Prediction for Configurable Software with Deep Sparse Neural Network. In *IEEE/ACM 41st International Conference on Software Engineering (ICSE).* 1095–1106. doi:10.1109/ICSE.2019.00113

[8] E. M. Martins, L. G. Faé, R. B. Hoffmann, L. S. Bianchessi, and D. Griebler. 2025. NPB-Rust: NAS Parallel Benchmarks in Rust. arXiv:2502.15536 [cs.DC] https://arxiv.org/abs/2502.15536

[9] N. D. Matsakis and F. S. Klock II. 2014. The Rust Language. In *ACM SIGAda Ada Letters*, Vol. 34. ACM, 103–104. doi:10.1145/2692956.2663188

[10] NVIDIA Corporation. 2024. *CUDA C Programming Guide.* NVIDIA. https://docs.nvidia.com/cuda/cuda-c-programming-guide/.

[11] K. Rocki, M. Burtscher, and R. Suda. 2014. The Future of Accelerator Programming: Abstraction, Performance or Can We Have Both?. In *29th Annual ACM Symposium on Applied Computing* (Gyeongju, Korea). ACM, New York, NY, USA, 886––895.

[12] N. Siegmund, A. Grebhahn, S. Apel, and C. Kästner. 2015. Performance-influence models for highly configurable systems. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering* (Bergamo, Italy) *(ESEC/FSE 2015).* Association for Computing Machinery, New York, NY, USA, 284–294. doi:10.1145/2786805.2786845

[13] The Rust RFC Book. 2017. *RFC 2045 - Target Features.* https://rust-lang.github.io/rfcs/2045-target-feature.html

[14] N. C. Thompson and S. Spanuth. 2021. The decline of computers as a general purpose technology. *Commun. ACM* 64, 3 (Feb. 2021), 64–72. doi:10.1145/3430936

[15] N. Vazou, E. L. Seidel, R. Jhala, D. Vytiniotis, and S. Peyton Jones. 2014. Refinement types for Haskell. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming* (Gothenburg, Sweden) *(ICFP'14).* ACM, New York, NY, USA, 269–282. doi:10.1145/2628136.2628161

[16] M. Velez, P. Jamshidi, N. Siegmund, S. Apel, and C. Kästner. 2021. White-Box Analysis over Machine Learning: Modeling Performance of Configurable Systems. In *Proceedings of the 43rd International Conference on Software Engineering* (Madrid, Spain) *(ICSE'21).* IEEE Press, 1072–1084. doi:10.1109/ICSE43902.2021.00100

A The matvecmul kernel function

```
#[platformaware(init_x, update_x, init_conj_grad, alloc_a_d, alloc_colidx_d, alloc_rowstr_d,
                move_a_to_device, alloc_x, alloc_p, alloc_q, alloc_r, alloc_z, freevectors,
                matvecmul, vecvecmul, scalarvecmul1, scalarvecmul2, norm)]
mod cg {
    ...
    #[kernelversion]  // written by GMAP/PUCRS team.
    fn matvecmul(colidx: &[i32],  rowstr: &[i32], a: &[f64], x: &[f64], y: &mut[f64]) {
        (&rowstr[0..NA as usize]).into_iter()
        .zip(&rowstr[1..NA as usize + 1]).zip(&mut y[0..(LASTCOL - FIRSTCOL + 1) as usize])
        .for_each(|((j, j1), y)| {
            *y = (&a[*j as usize..*j1 as usize]).into_iter()
                .zip(&colidx[*j as usize..*j1 as usize])
                .map(|(a, colidx)| a * x[*colidx as usize]).sum();
        });
    }

    #[kernelversion(cpu_core_count=(AtLeast{val:2}))]  // written by GMAP/PUCRS team.
    fn matvecmul(colidx: &[i32],  rowstr: &[i32], a: &[f64], x: &[f64], y: &mut[f64]) {
        ( &rowstr[0..NA as usize], &rowstr[1..NA as usize + 1],
          &mut y[0..(LASTCOL - FIRSTCOL + 1) as usize]).into_par_iter()
              .for_each(|(j, j1, y)| {
                                *y = (&a[*j as usize..*j1 as usize]).into_iter()
                                    .zip(&colidx[*j as usize..*j1 as usize])
                                    .map(|(a, colidx)| a * x[*colidx as usize]).sum();
              }); // rayon implementation
    }

    #[kernelversion(acc_count=(AtLeast{val:1}), acc_backend=CUDA,
                    acc_cudatoolkit=(AtLeast{val:20}), acc_cudacc=(AtLeast{val:13}),
                    acc_cudadriver=(AtLeast{val:17713}))]  // CUDA version 0
    fn matvecmul(colidx: &[i32],  rowstr: &[i32], a: &[f64], x: &[f64], y: &mut[f64]) {
        let nnz = a.len() as i32;
        let num_rows = y.len() as i32;
        let x_len = x.len() as i32;
        unsafe { launch_matvecmul_cuda(a.as_ptr(), colidx.as_ptr(), rowstr.as_ptr(),
                                       x.as_ptr(), y.as_mut_ptr(), nnz, num_rows, x_len); }
    }

    #[kernelversion(acc_count=(AtLeast{val:1}), acc_backend=CUDA,
                    acc_cudatoolkit=(AtLeast{val:50}), acc_cudacc=(AtLeast{val:35}),
                    acc_cudadriver=(AtLeast{val:31937}))]  // CUDA version 1
    fn matvecmul(colidx: &[i32],  rowstr: &[i32], a: &[f64], x: &[f64], y: &mut[f64]) {
        let nnz = a.len() as i32;
        let num_rows = y.len() as i32;
        let x_len = x.len() as i32;
        unsafe { launch_matvecmul_CC35(a.as_ptr(), colidx.as_ptr(), rowstr.as_ptr(),
                                       x.as_ptr(), y.as_mut_ptr(), nnz, num_rows, x_len); }
    }

    #[kernelversion(acc_count=(AtLeast{val:1}), acc_backend=CUDA,
                    acc_cudatoolkit=(AtLeast{val:101}), acc_cudacc=(AtLeast{val:60}),
                    acc_cudadriver=(AtLeast{val:41839}))]  // CUDA version 2
    fn matvecmul(colidx: &[i32],  rowstr: &[i32], a: &[f64], x: &[f64], y: &mut[f64]) {
        let nnz = a.len() as i32;
        let num_rows = y.len() as i32;
        let x_len = x.len() as i32;
        unsafe { launch_matvecmul_CC60(a.as_ptr(), colidx.as_ptr(), rowstr.as_ptr(),
                                       x.as_ptr(), y.as_mut_ptr(), nnz, num_rows, x_len); }
    }

    #[kernelversion(acc_count=(AtLeast{val:1}), acc_backend=CUDA,
                    acc_cudatoolkit=(AtLeast{val:90}), acc_cudacc=(AtLeast{val:70}),
                    acc_cudadriver=(AtLeast{val:38481}))]  // CUDA version 3
    fn matvecmul(colidx: &[i32],  rowstr: &[i32], a: &[f64], x: &[f64], y: &mut[f64]) {
        let nnz = a.len() as i32;
        let num_rows = y.len() as i32;
        let x_len = x.len() as i32;
        unsafe { launch_matvecmul_CC70(a.as_ptr(), colidx.as_ptr(), rowstr.as_ptr(),
                                       x.as_ptr(), y.as_mut_ptr(), nnz, num_rows, x_len); }
    }
}
```

B The launch_matvecmul_cuda CUDA kernel

```
extern "C" {
    __global__ void matvecmul_cuda(
                        const double* a, const int* colidx, const int* rowstr,
                        const double* x, double* y, int num_rows)
    {
        int row = blockIdx.x * blockDim.x + threadIdx.x;
        if (row < num_rows) {
            double sum = 0.0;
            int start = rowstr[row], end = rowstr[row + 1];
            for (int i = start; i < end; ++i) sum += a[i] * x[colidx[i]];
            y[row] = sum;
        }
    }

    void launch_matvecmul_cuda (
         const double* d_aa,  const int* d_colidx, const int* d_rowstr,
         const double* d_xx, double* d_yy, int nnz, int num_rows, int x_len)
    {
        int blockSize = BLOCK_SIZE, gridSize = (num_rows+blockSize-1)/blockSize;
        matvecmul_cuda<<<gridSize, blockSize>>>(d_aa, d_colidx, d_rowstr,
                                                d_xx, d_yy, num_rows);
        cudaDeviceSynchronize();
     }
}
```