Structured platform-aware programming

Francisco H. de Carvalho Junior¹, Allberson B. de Oliveira Dantas², Júlio H. Mendes³, Tiago Carneiro⁴, Claro H. S. Sales¹, Pedro A. F. de Sales¹

¹Pós-Graduação em Ciência da Computação (MDCC)
Universidade Federal do Ceará (UFC)
Fortaleza, Brazil

²Instituto de Engenharias e Desenvolvimento Sustentável
Universidade da Integração Internacional da Lusofonia Afro-Brasileira (Unilab)
Redenção, Brazil

³Arpeggeo Technologies
Rio de Janeiro, Brazil

⁴ExaScience: HPC software laboratory
Interuniversity Microelectronics Centre (IMEC)
Leuven, Belgium

heron@dc.ufc.br

Abstract. Platform-aware programming is a usual practice of HPC performance engineering programmers that is becoming more challenging due to the increasing heterogeneity of parallel computing platforms. In this paper, it is proposed a structured approach to platform-aware programming based on three concepts: platform typing, multiple dispatch, and feature detection. It has been implemented and evaluated through a proof-of-concept prototype built in Julia. It is evidenced that structured platform-aware programming provides better modularity and ease of maintenance with minor performance overhead.

1. Introduction

The unprecedented demand for high-performance computing (HPC) resources is currently motivated by applications in Big Data Analytics and Artificial Intelligence (AI). Parallel computing techniques have become widespread, and Heterogeneous Computing has become essential for supporting these applications.

Platform-aware programming is the performance engineering practice of coding by making assumptions about features of the execution platform, to face HPC requirements. It becomes more difficult by using unstructured means as parallel computing platforms become more heterogeneous. Designers of programming languages and frameworks attempt to find high-level abstractions and unified interfaces to circumvent platform-aware programming [Rocki et al. 2014].

There are reasons to encourage attempts to improve the practice of platform-aware programming for performance engineers. First, performance portability issues of high-level approaches will continue to be challenging to mitigate. Second, it encourages hardware designers to propose new accelerators, with new programming interfaces, to target specific computation patterns, such as deep learning (e.g., Google’s TPUs and NVIDIA’s...
tensor cores) and irregular applications [Carneiro et al. 2021]. Third, heterogeneous computing platforms tend to become more heterogeneous as the industry introduces increasingly specialized accelerators to meet the requirements of applications.

This paper proposes a structured approach to platform-aware programming based on platform typing, multiple dispatch, and feature detection. It is prototyped in Julia [Bezanson et al. 2017] as a package named PlatformAware.jl evaluated using two case studies. The first accelerates ImageQuilting.jl [Hoffmann et al. 2017], a solver from GeoStats.jl, a framework for geostatistics. The second accelerates a combinatorial search algorithm on several heterogeneous computing platform scenarios. The results evidence that structured platform-aware programming improves the ability to deal with a large number of assumptions about platform features with minor performance overhead.

This paper has five more sections. Section 2 discusses Related Work in the literature and programming language designs. Section 3 introduces platform-aware programming. Section 4 presents structured platform-aware programming and PlatformAware.jl, i.e., how it is implemented in Julia. Section 5 presents case studies to demonstrate PlatformAware.jl and evaluate structured platform-aware programming in terms of performance and productivity. Finally, Section 6 concludes the study and discusses future work.

2. Related Work

The literature on the implementation of algorithms exploring specific features of execution platforms is large. It became a major field of interest in the 2000s with the increasing interest of researchers to investigate how to use GPUs and FPGAs efficiently for their HPC applications [Hijma et al. 2022].

In Heterogeneous Computing, researchers propose high-level abstractions (e.g., [Nieplocha et al. 1996, De Wael et al. 2015]), unified programming interfaces (e.g., OpenCL, OpenACC, oneAPI), and auto-tuning tools [Park et al. 2022], hiding the heterogeneity of computational resources. Although successful in code portability, performance portability remains challenging [Rocki et al. 2014, Bertoni et al. 2020, Kwack et al. 2021]. This is why performance engineers still prefer architecture-oriented parallel programming interfaces (e.g. MPI and OpenMP), as well as proprietary APIs to program accelerators (e.g. CUDA and ROCm).

[Ernstsson and Kessler 2020] are the first that use the term platform-aware programming, to introduce multi-variant user functions to SkePU, a skeleton programming framework, which allows writing multiple versions of functions that exploit specific features of the execution platform. However, they do not exploit the idea of automatic feature detection nor the use of types and multiple dispatch to represent platform features and selection between function variants dynamically.

The function multi-versioning feature from the GNU C compiler (GCC) 4.8 uses label annotations to distinguish function versions that exploit special instruction sets of processors (e.g., SIMD). The compiler may optimize function versions according to the target instruction set, and they are selected at runtime. However, it only supports instruction sets as features and uses a priority list of labels to resolve dispatch ambiguities when multiple versions can be applied. In turn, this work resolves dispatch ambiguities through platform types, subtyping relations, and multiple dispatch semantics, but the use of platform assumptions to guide compiler optimizations is still a topic for future work.
Finally, Rust’s RFC 2045 (Request-for-Comments) was published in 2017 to introduce target features [The Rust RFC Book 2017], analogous to GCC’s function versions. It covers a smaller set of features than PlatformAware.jl, expresses concern about the feasibility of runtime feature detection, and discusses safety drawbacks and limitations that are addressed in PlatformAware.jl through platform typing and multiple dispatch.

3. Platform-aware programming

Ad hoc platform-aware programming is the practice of performance engineers to code by making assumptions about the features of the target execution platforms without the support of specific platform-aware programming language constructs or abstractions, to meet HPC requirements. It becomes more difficult as performance and heterogeneous computing requirements of parallel computing platforms become more critical.

Listing 1. The FFT kernel (fft) using ad hoc platform-aware programming

```julia
module MyFFT

import CUDA; const cu = CUDA; const cufft = cu.CUFFT
import FFTW; const fftw = FFTW
import OpenCL; const cl = OpenCL
import CLFFT; const clfft = CLFFT

fft_cpu(X) = fftw.fft(X)
fft_cuda(X) = cufft.fft(X)

function fft_opencl(X)
    _, ctx, queue = cl.create_compute_context()
    bufX = cl.Buffer(Complex64, ctx, :copy, hostbuf=X)
    p = clfft.Plan(Complex64, ctx, size(X))
    clfft.set_layout!(p, :interleaved, :interleaved)
    clfft.set_result!(p, :inplace)
    clfft.bake!(p, queue)
    clfft.enqueue_transform(p, :forward, 
                          [queue], bufX, nothing)
    result = cl.read(queue, bufX)
end

nvidia_cuda_ok() = all([cu.functional(), !isempty(cu.devices()), contains(cu.name(cu.device()), "NVIDIA")])
opencl_ok() = !isempty(cl.devices())

function selectKernel()
    nvidia_cuda_ok() && return fft_cuda
    opencl_ok() && return fft_opencl
    return fft_cpu
end

fft = selectKernel(); export fft
end
```

Listing 1 presents the code of a module in Julia that exports the `fft` function. If a GPU exists, the CUDA or the OpenCL method of `fft` is selected, depending on the GPU’s manufacturer ( NVIDIA or another, like AMD or Intel). Otherwise, a fallback CPU version based on FFTW.jl is selected. The selection is driven by a selection function (`selectKernel`) that checks if a GPU exists and whether it supports CUDA or OpenCL.

The above solution separates code variants and the selection code. However, for small-scale scenarios (e.g., few assumptions), using a single monolithic `fft` method whose code variants are scattered across multiple branches may work. Also, when functions have different assumptions, it is better to write a selection function for each kernel.

The selection of code variants may be static (compile time) or dynamic (execution time). This paper addresses the latter, in which the code can move between different platforms without static recompilation, achieving the code portability of high-level approaches without performance portability issues since the code is architecture-specific.
Ad hoc platform-aware programming may become rather complex in the cross-platform development of software that must run efficiently at an increasingly heterogeneous set of processors and accelerators. Furthermore, platform-aware coding decisions must be shared among programmers in a team because they may affect the overall design and implementation decisions. There are reasons to support research initiatives on new abstractions to improve platform-aware programming, as many performance engineering developers will continue to use architecture-specific programming interfaces.

4. Structured platform-aware programming

This paper proposes a structured approach to replace ad hoc platform-aware programming practice, based on tree concepts: platform typing; multiple dispatch; and feature detection.

As a proof-of-concept, structured platform-aware programming has been implemented in the Julia programming language [Bezanson et al. 2017], for the following reasons. First, it supports multiple dispatch combined with a rich type system that, together with its metaprogramming features, makes it possible its rapid prototyping without modifying either the compiler or Julia’s runtime system. Second, despite it being focused on productivity, like Python, it was originally designed to meet HPC requirements. Third, Julia offers a number of heterogeneous programming packages in the [JuliaParallel], [JuliaSIMD], and [JuliaGPU] Github organizations. Finally, Julia separates the concerns of developers and users of packages. While developers are concerned with HPC requirements of packages, users are concerned with using them to solve problems.

4.1. Platform typing

In programming languages, types specify the shape of data and the functions that can be applied to them [Pierce 1991]. The benefits of types for programming are safety, as they protect data from invalid operations, and performance, as they enable compilers to optimize functions according to how the data to which they are applied is represented.

To improve platform-aware programming, this paper proposes a platform abstraction along with data and function abstractions in type systems. Figure 1 depicts such an extension. A platform defines the environment where a function finds resources to process data efficiently. Not only hardware but also software resources, like the operating system, subroutine libraries, etc. Like a function that may be tuned according to assumptions about the type of data to which it will be applied, a function may be tuned according to the features of the platform that hosts its execution to accelerate execution. In fact, performance engineering programmers do this in ad hoc platform-aware programming.

Just as data types enable the implementation of different mechanisms for data abstraction, execution safety, compilation level optimization, etc., platform types serve dif-
This paper applies platform types to structured platform-aware programming, as a modular way to write different versions of functions with HPC requirements according to assumptions about the features of the target execution platforms.

A feature is a characteristic of the execution platform that may be exploited to accelerate the execution of a function. An assumption is a statement about a feature. Figure 2 shows a fragment of a hierarchy of platform types, with feature types at the top (SIMDExtensions and Microarchitecture). The other types are assumption types. For example, AVX512 is an assumption type to assert whether the processor supports the AVX512 extension for the feature related to the support of SIMD and vector instructions in CPUs (SIMDExtensions). Backward compatibility is a subtyping relation. So, Figure 2 asserts that processors with the AVX512 extension also support AVX2 and AVX.

For implementing platform typing, multiple dispatch [Muschevici et al. 2008] over platform types augmented with a subtyping relation is proposed. This is supported by Julia [Nardelli et al. 2018]. In a function prototype, in addition to regular parameters, there are platform parameters typed by feature types representing assumptions for the function implementation. Programmers may define function versions for different assumptions, by associating to each platform parameter an assumption type that is a proper subtype of the feature type of the parameter. During execution, the function version that best fits the actual features of the execution platform is selected through multiple dispatch over platform parameters. For that, the actual features of the execution platform must be automatically passed, as platform types, to platform parameters. These so-called platform arguments are determined by a feature detection mechanism, discussed in Section 4.6.

Listing 2. The FFT kernel function (fft) using PlatformAware.jl

```julia
module MyFFT
    import CUDA; const cu = CUDA; const cufft = cu.CUFFT
    import FFTW; const fftw = FFTW
    import OpenCL; const cl = OpenCL
    import CLFFT; const clfft = CLFFT
    using PlatformAware

    @platform default fft(X) = fftw.fft(X)
    @platform aware fft((accelerator_brand::NVIDIA, accelerator_api::(@api CUDA)), X) = cufft.fft(X)
    @platform aware function fft((accelerator_api::(@api OpenCL)), X)
        _, ctx, queue = cl.create_compute_context()
        bufX = cl.Buffer(Complex64, ctx, :copy, hostbuf=X)
        p = clfft.Plan(Complex64, ctx, size(X))
        clfft.set_layout!(p, :interleaved, :interleaved)
        clfft.set_result!(p, :inplace)
        clfft.bake!(p, queue)
        clfft.enqueue_transform(p, :forward, [queue], bufX, nothing)
        result = cl.read(queue, bufX)
    end

    export fft
end
```
4.2. Basic concepts and overview

A *kernel function* is a function with multiple versions, called *kernel methods*, for different assumptions about features of the execution platform, each represented as a platform parameter typed by an assumption type.

Listing 2 presents a structured version of the code in Listing 1. It uses the `@platform` macro to declare kernel methods with platform parameters enclosed in braces, just before regular parameters. The `default` modifier just after `@platform` distinguishes the *default kernel*, which does not declare platform parameters, acting as a fallback kernel.

The `@platform` macro derives kernel selection code through multiple dispatch based on feature detection, eliminating `selectKernel`. Feature detection is discussed in Section 4.6. Also, to apply platform types as platform arguments, `@platform` rewrites the type of each platform parameter from `T` to the power type of `T`, i.e., `Type{<:T}`.

The code in Listing 3 uses `MyFFT` to implement `fftconv`, an FFT-based convolution function. Note that the use of `PlatformAware.jl` does not affect the code that uses `MyFFT`.

```plaintext
using MyFFT

function fftconv(img, krn)
    padkrn = zeros(size(img))
    copyto!(padkrn, CartesianIndices(krn), krn, CartesianIndices(krn))
    fft(img) .* conj.(fft(padkrn))
end

Listing 3. The `fftconv` function
```

Platform types form a hierarchy with `PlatformType` at the root and `QualifierFeature` and `QuantifierFeature` as direct subtypes (quantifier and qualifier types).

4.3. Quantifier types (<: QuantifierFeature)

Platform types that specify numerically valued assumptions are called *quantifier types*, or simply *quantifiers*. For example, the choice of better scalable parallel algorithms depends on the number of processing resources (nodes, processors, accelerators, etc.), available memory, interconnection latency, bandwidth, etc. `PlatformAware.jl` supports powers of two quantifiers, which are common in the design of parallel algorithms and description of platform features. Thus, two finite sets of abstract types named `AtLeastN` and `AtMostN` are defined, where `N` is a label 0, for representing zero; `Inf`, a value that is greater than anyone, or `2^i` for `i ∈ {−30, −29, −28, . . . , 69}`. The label of a `2^i` value is a concatenation of two labels: *multiplier* and *magnitude*. The multipliers are 1, 2, 4, 8, 16, 32, 64, 128, 256, 512 and the magnitude labels are `n (2^−30)`, `u (2^−20)`, `m (2^−10)`, `K (2^10)`, `M (2^20)`, `G (2^30)`, `T (2^40)`, `P (2^50)`, and `E (2^60)`. For example, `AtLeast128G` represents `AtLeast{2^37}` (i.e., `128 × 2^30`), `AtLeast8` represents `AtLeast{2^3}`, and `AtMost16K` represents `AtLeast{2^14}`. `AtLeastInf` and `AtMost0` are the bottom at-least and at-most quantifiers, while `AtLeast0` and `AtMostInf` are the top ones, respectively, in subtyping.

Quantifier parameters are typed as `Type{<: Tuple{AtLeastM, AtMostN}}`. So, programmers work with interval assumptions, with values between `M` and `N`. As syntactic sugar, the macros presented in Table 1 may be used by programmers.
4.4. Qualifier types (<:: QualifierFeature)
Platform types that are not quantifiers are qualifier types, or qualifiers. They have been exemplified in Figure 2. Listing 4 shows a fragment of qualifiers of PlatformAware.jl for typing the accelerator_api parameter with assumption types representing CUDA versions. Programmers may use the macro @api api version, where api is the name of the API, and version is the API version. It was used in the example in Section 4.2.

4.5. Platform parameters
Table 2 presents the names and default types of the platform parameters currently supported by PlatformAware.jl, version 0.5.1. Default types are implicitly applied to platform parameters not referred to in the signature of kernel methods.

The @platform macro includes all the platform parameters by default for any kernel method. However, the programmer may explicitly select a list of parameters through @platform parameter declarations. For that, he/she may firstly clear the list using @platform parameter clear. Then, write a sequence of @platform parameter ⟨parameter name⟩ declarations to include the required ones.

4.6. Feature detection
Platform arguments are read from a feature description file called Platform.toml, automatically generated when PlatformAware.jl is installed by a call to PlatformAware.setup().

Table 1. Quantifier type macros

<table>
<thead>
<tr>
<th>name</th>
<th>default type</th>
</tr>
</thead>
<tbody>
<tr>
<td>node_count</td>
<td>@atleast 1</td>
</tr>
<tr>
<td>node_virtual</td>
<td>No</td>
</tr>
<tr>
<td>node_dedicated</td>
<td>No</td>
</tr>
<tr>
<td>node_maintainer</td>
<td>No</td>
</tr>
<tr>
<td>node_locality</td>
<td>No</td>
</tr>
<tr>
<td>node_machinefamily</td>
<td>No</td>
</tr>
<tr>
<td>node_machine</td>
<td>No</td>
</tr>
<tr>
<td>node_vcpus_count</td>
<td>@atleast 1</td>
</tr>
<tr>
<td>node_coworkers_count</td>
<td>@atleast 0</td>
</tr>
<tr>
<td>node_memory_size</td>
<td>@atleast 0</td>
</tr>
<tr>
<td>node_memory_latency</td>
<td>@atmost ∞</td>
</tr>
<tr>
<td>node_memory_bandwidth</td>
<td>@atleast 0</td>
</tr>
<tr>
<td>node_memory_type</td>
<td>@atmost ∞</td>
</tr>
<tr>
<td>processor_count</td>
<td>@atleast 1</td>
</tr>
<tr>
<td>processor_brand</td>
<td>@atleast 1</td>
</tr>
<tr>
<td>processor_architecture</td>
<td>@atleast 1</td>
</tr>
<tr>
<td>processor_isa</td>
<td>@atleast 1</td>
</tr>
<tr>
<td>processor_simd</td>
<td>@atleast 1</td>
</tr>
<tr>
<td>processor_power_efficiency</td>
<td>@atleast 1</td>
</tr>
<tr>
<td>processor_core_count</td>
<td>@atleast 1</td>
</tr>
<tr>
<td>processor_core_threads_count</td>
<td>@atleast 1</td>
</tr>
<tr>
<td>processor</td>
<td>@atleast 1</td>
</tr>
</tbody>
</table>

Table 2. platform parameters and their feature types in PlatformAware.jl (v0.5.1)

<table>
<thead>
<tr>
<th>name</th>
<th>default type</th>
</tr>
</thead>
<tbody>
<tr>
<td>node_count</td>
<td>@atleast 1</td>
</tr>
<tr>
<td>node_virtual</td>
<td>No</td>
</tr>
<tr>
<td>node_dedicated</td>
<td>No</td>
</tr>
<tr>
<td>node_maintainer</td>
<td>No</td>
</tr>
<tr>
<td>node_locality</td>
<td>No</td>
</tr>
<tr>
<td>node_machinefamily</td>
<td>No</td>
</tr>
<tr>
<td>node_machine</td>
<td>No</td>
</tr>
<tr>
<td>node_vcpus_count</td>
<td>@atleast 1</td>
</tr>
<tr>
<td>node_coworkers_count</td>
<td>@atleast 0</td>
</tr>
<tr>
<td>node_memory_size</td>
<td>@atleast 0</td>
</tr>
<tr>
<td>node_memory_latency</td>
<td>@atmost ∞</td>
</tr>
<tr>
<td>node_memory_bandwidth</td>
<td>@atleast 0</td>
</tr>
<tr>
<td>node_memory_type</td>
<td>MemoryType</td>
</tr>
<tr>
<td>processor_count</td>
<td>@atleast 1</td>
</tr>
<tr>
<td>processor_brand</td>
<td>Manufacturer</td>
</tr>
<tr>
<td>processor_architecture</td>
<td>ProcessorArchitecture</td>
</tr>
<tr>
<td>processor_isa</td>
<td>ProcessorISA</td>
</tr>
<tr>
<td>processor_simd</td>
<td>ProcessorSIMD</td>
</tr>
<tr>
<td>processor_power_efficiency</td>
<td>@unrestricted</td>
</tr>
<tr>
<td>processor_core_count</td>
<td>@atleast 0</td>
</tr>
<tr>
<td>processor_core_threads_count</td>
<td>@atleast 1</td>
</tr>
<tr>
<td>processor</td>
<td>Processor</td>
</tr>
<tr>
<td>storage_size</td>
<td>@atleast 0</td>
</tr>
<tr>
<td>storage_latency</td>
<td>@atmost ∞</td>
</tr>
<tr>
<td>storage_bandwidth</td>
<td>@atleast 0</td>
</tr>
<tr>
<td>storage_networkbandwidth</td>
<td>@atleast 0</td>
</tr>
<tr>
<td>storage_type</td>
<td>StorageType</td>
</tr>
<tr>
<td>storage_interface</td>
<td>StorageInterface</td>
</tr>
</tbody>
</table>

Listing 4. Qualifier types for CUDA versions less than or equal to 2.0

```
abstract type AcceleratorBackend <: QualifierFeature end
abstract type CUDA_API <: AcceleratorBackend end
abstract type CUDA_1_0 <: CUDA_API end; constant CUDA1 = CUDA_1_0
abstract type CUDA_1_3 <: CUDA_API end; constant CUDA1_3 = CUDA_1_3
abstract type CUDA_2_0 <: CUDA_API end; constant CUDA2 = CUDA_2_0
```
can also be called through Julia’s REPL. The setup function uses a set of system tools to detect features of the execution platform to determine platform arguments.

The user may edit Platform.toml manually to add, refine, and remove features. Cloud providers and HPC/supercomputing centers may provide Platform.toml files that accurately describe the features of their platforms. In fact, accurate automatic feature detection is challenging, still requiring a variety of tools, mostly outside the language environment, and often non-portable across operating systems. For this reason, PlatformAware.jl has a feature database of processors and accelerators from Intel, AMD, and NVIDIA, currently stored in the source code repository using CSV format.

5. Case studies and Performance Evaluation
Two case studies on the use of PlatformAware.jl are presented. In Section 5.1 it accelerates the imfilter kernel function of ImageQuilting.jl, a package for 3D image quilting simulation of a framework for high-performance geostatistics called GeoStats.jl [Hoffmann et al. 2017]. In Section 5.2 it packages a set of implementations of a combinatorial search algorithm that solves the N-Queens problem for distinct parallel computing platforms, originally coded in C and Chapel [Carneiro et al. 2021].

5.1. ImageQuilting.jl
The case study with ImageQuilting.jl shows a basic dispatch scenario targeting CUDA and OpenCL accelerators, with three methods of the kernel function imfilter. They are presented in Listing 5: kernel 1, a fallback CPU implementation, as a default kernel; kernel 2, that employs CUDA.jl code if an NVIDIA GPU is available; kernel 3, employing OpenCL.jl and CLFFT.jl for accelerators from other vendors.

Listing 5. Methods of the imfilter kernel function for ImageQuilting.jl

```julia
# kernel 1: fallback
@platform default imfilter(img, kern) = imfilter(img, centered(krn), Inner(), Algorithm.FFT())

# kernel 2: CUDA for NVIDIA GPUs
@platform aware function imfilter((accelerator_count::(@atleast 1 A), accelerator_brand::NVIDIA),img,kern)
where A
    imfilter_cuda(img, krn, A) # A is the number of accelerator GPUs
end

# kernel 3: OpenCL for GPUs of other vendors
@platform aware function imfilter((accelerator_count::(@atleast 1 A)),img,kern)
where A
    imfilter_opencl(img, krn, A) # A is the number of accelerator GPUs
end
```

Kernels 2 and 3 are selected if at least one accelerator is available, due to the accelerator_count assumption. Since CUDA is compatible only with NVIDIA devices, accelerator_brand is typed with the NVIDIA qualifier to guide the selection of the CUDA kernel. Anyone can introduce new imfilter kernels to exploit the features of other vendor-specific GPU programming APIs supported by Julia, such as AMDGPU.jl, oneAPI.jl, Metal.jl, and XLA.jl. More specialized imfilter kernels may also exist for specific API versions through accelerator_api, as well as GPU architectures (e.g., Kepler, Turing, Ampere, Hopper) through accelerator_architecture.

5.2. Combinatorial Search (NQueens)
This case study addresses the dispatch of parallel and accelerator-based methods of a kernel function implementing a backtracking algorithm that enumerates all complete and valid solutions to the N-Queens problem [Carneiro et al. 2021].
The Julia versions were encapsulated in the following functions: `nqueens_serial` (single core), `nqueens_mcore` (multicore), `nqueens_sgpu` (single GPU with CUDA support), `nqueens_mgpu` (multiple GPUs with CUDA support), `nqueens_mgpu_mcore` (multicore and multiple GPUs with CUDA support), and `nqueens_distributed` (cluster computing with multicore nodes). These functions are encapsulated in the methods of a kernel function called `nqueens` in a package called `PlatformAwareQueens.jl`. The code in Listing 6 shows the different platform assumptions of the `nqueens` kernels. Parameter `size` is the size $N$ of an $N \times N$ chess table.

```
Listing 6. Queens kernel methods

```@platform parameter clear
@platform parameter node_count, processor_count, processor_core_count, accelerator_count, accelerator_api
@platform default nqueens(size) = nqueens_serial(size)

# SINGLE GPU
@platform aware
function nqueens({node_count::@just(1), accelerator_count::(@just 1),
                    accelerator_api::(@api CUDA)}, size)
    nqueens_sgpu(size) # FALLBACK (single core)
end

# MULTI-GPUs
@platform aware
function nqueens({node_count::@just(1), accelerator_count::(@atleast 2),
                    accelerator_api::(@api CUDA)}, size)
    nqueens_mgpu(size) # MULTI-GPUs
end

# MULTICORE (multiple processors)
@platform aware
function nqueens({node_count::@just(1), processor_count::(@atleast 2),
                    processor_core_count::@just(0), accelerator_count::@just(0)}, size)
    nqueens_mcore(size) # MULTICORE (multiple processors)
end

# MULTICORE/MULTI-GPUs (multiple processors)
@platform aware
function nqueens({node_count::@just(1), processor_count::(@atleast 2),
                    processor_core_count::(@atleast 2),
                    accelerator_count::(@atleast 2), accelerator_api::(@api CUDA)}, size)
    nqueens_mgpu_mcore(size) # MULTICORE/MULTI-GPUs (multiple processors)
end

# MULTICORE/MULTI-GPUs (single processor with multiple cores)
@platform aware
function nqueens({node_count::@just(1), processor_count::(@atleast 2),
                    processor_core_count::(@atleast 2),
                    accelerator_count::(@atleast 2), accelerator_api::(@api CUDA)}, size)
    nqueens_mgpu_mcore(size) # MULTICORE/MULTI-GPUs (single processor with multiple cores)
end

# CLUSTER COMPUTING
@platform aware
function nqueens({node_count::(@atleast 2)}, size)
    nqueens_distributed(size) # CLUSTER COMPUTING
end
```

5.3. Evaluation objective

`PlatformAware.jl` is evaluated regarding performance to evidence that structured platform-aware programming does not lead to significant overheads compared to ad hoc platform-aware programming. For that, two versions of `ImageQuilting.jl` and `PlatformAwareQueens.jl` have been built: the structured versions, using `PlatformAware.jl`; and the ad hoc versions, where platform-aware programming is done through ad hoc means.

5.4. Results and Discussion

Table 4 shows the average execution times of 30 executions of each method of kernels `imfilter` and `nqueens`, for their structured and the ad hoc versions, across 3 workloads (`small`, `medium`, and `large`) defined in Table 3. To evaluate the JIT compilation overhead, the first run was separated from the next ones. The test-$t$ checks the statistical significance of the conclusions. Julia 1.8.5 was used. Table 3 also describes the testbed platforms.
Using the same protocol, Table 5 presents the average execution time for *iqsim*, which calls *imfilter* iteratively. In this experiment, the structured and the ad hoc versions of *ImageQuilting.jl* runs on a platform equipped with an NVIDIA GTX 1080 Ti GPU, leading to the selection of the CUDA kernel.

The results in Tables 4 and 5 evidence that the use of *PlatformAware.jl* for structured platform-aware programming does not cause significant execution or compilation overhead, as the vast majority of the *test-t* values are greater than α, where α = 0.05.

Table 6 shows the time spent in the first load of the ad hoc and structured versions of *ImageQuilting.jl* and *PlatformAwareQueens.jl* in the computer platform, when pre-compilation is performed. The additional costs of the structured versions are multiple dispatch over platform parameters and feature detection by reading the Platform.toml file. The overhead is 3.6s in *PlatformAwareQueens.jl* and 7.8s (8.5%) in *ImageQuilting.jl*.

### Table 3. Testbed (computation workloads and execution platforms)

<table>
<thead>
<tr>
<th>Case</th>
<th>Small</th>
<th>Medium</th>
<th>Large</th>
<th>Platform</th>
</tr>
</thead>
<tbody>
<tr>
<td>Queens</td>
<td>queens(15)</td>
<td>queens(16)</td>
<td>queens(17)</td>
<td>(1), (2)</td>
</tr>
<tr>
<td>imfilter</td>
<td>img = rand(64, 64, 64)</td>
<td>img = rand(128, 128, 128)</td>
<td>img = rand(256, 256, 256)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>km = rand(10, 10, 10)</td>
<td>km = rand(10, 10, 10)</td>
<td>km = rand(10, 10, 10)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>imfilter_kerneling, km</td>
<td>imfilter_kerneling, km</td>
<td>imfilter_kerneling, km</td>
<td></td>
</tr>
<tr>
<td>iqsimg</td>
<td>using Geostats, GeoStatsImage</td>
<td>using Geostats, GeoStatsImage</td>
<td>using Geostats, GeoStatsImage</td>
<td></td>
</tr>
<tr>
<td></td>
<td>T_k = geostatsImage(&quot;WalkerLake&quot;)</td>
<td>T_k = geostatsImage(&quot;StanfordV&quot;)</td>
<td>T_k = geostatsImage(&quot;Flumin&quot;)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>iqsimg(asarray(TIs, :facies), (30, 30))</td>
<td>iqsimg(asarray(TIs, :K), (30, 30, 30))</td>
<td>iqsimg(asarray(TIs, :K), (30, 30, 30))</td>
<td></td>
</tr>
</tbody>
</table>

(1) Two 16-core AMD EPYC 7351 CPUs sharing 128GiB RAM and four NVIDIA Tesla T4 15GiB GPUs.
(2) A cluster with four nodes, each comprising two 20-core Intel Xeon Gold 5218R CPUs sharing 96GiB RAM, interconnected through a 25Gb Ethernet network.
(3) Two 6-core Xeon E5-2650v4 CPUs sharing 128GiB RAM and two NVIDIA GTX 1080 Ti 11GiB GPUs.

---

### Table 4. Kernel execution time

<table>
<thead>
<tr>
<th>Package</th>
<th>Small</th>
<th>Medium</th>
<th>Large</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>direct</td>
<td>structured</td>
<td>Test-*</td>
</tr>
<tr>
<td>ImageQuilting.jl</td>
<td>CPU</td>
<td>15.42</td>
<td>15.38</td>
</tr>
<tr>
<td></td>
<td>CUDA</td>
<td>20852</td>
<td>20758</td>
</tr>
<tr>
<td></td>
<td>OpenCL</td>
<td>629.4</td>
<td>632.3</td>
</tr>
<tr>
<td>Queens.jl</td>
<td>1-CPU</td>
<td>32.34</td>
<td>32.79</td>
</tr>
<tr>
<td></td>
<td>M-CPU</td>
<td>3.15</td>
<td>3.17</td>
</tr>
<tr>
<td></td>
<td>1-GPU</td>
<td>25.37</td>
<td>25.13</td>
</tr>
<tr>
<td></td>
<td>N-CPU/GPU</td>
<td>23.90</td>
<td>23.82</td>
</tr>
<tr>
<td></td>
<td>distributed</td>
<td>2.20</td>
<td>2.22</td>
</tr>
<tr>
<td>ImageQuilting.jl</td>
<td>CPU</td>
<td>14.96</td>
<td>14.99</td>
</tr>
<tr>
<td></td>
<td>CUDA</td>
<td>1.001</td>
<td>0.999</td>
</tr>
<tr>
<td></td>
<td>OpenCL</td>
<td>13.25</td>
<td>13.40</td>
</tr>
<tr>
<td>Queens.jl</td>
<td>1-CPU</td>
<td>30.79</td>
<td>31.24</td>
</tr>
<tr>
<td></td>
<td>M-CPU</td>
<td>1.40</td>
<td>1.35</td>
</tr>
<tr>
<td></td>
<td>1-GPU</td>
<td>4.43</td>
<td>4.44</td>
</tr>
<tr>
<td></td>
<td>N-GPU</td>
<td>1.05</td>
<td>1.05</td>
</tr>
<tr>
<td></td>
<td>N-CPU/GPU</td>
<td>0.75</td>
<td>0.77</td>
</tr>
<tr>
<td></td>
<td>distributed</td>
<td>657</td>
<td>658</td>
</tr>
</tbody>
</table>

* The Test-*t* values less than 0.05 (≤ α) are in red/bold face, denoting a statistically significant difference with 95% confidence.
### 5.5. Productivity

Platform-aware programming impacts both the readability and maintainability of code. Regarding readability, the ad hoc approach makes it necessary to interpret the code of selection/kernel functions to understand platform assumptions, which becomes even more difficult as the number and complexity of kernels and assumptions increase than just reading platform parameters, as in the structured approach. Regarding maintainability, consider the problem of adding a new method to the kernel function. In the ad hoc approach, a new function is created and the kernel selection function is modified to test the new assumptions. In turn, in the structured approach, only a new kernel method is created based on the assumptions of the platform parameters. No existing code is modified. This is analogous to removing kernels and changing their assumptions, requiring only local adjustments in kernel methods. In collaborative development environments, better maintainability increases programming safety when multiple collaborators are adding, removing, and modifying kernels independently, an inherently error-prone task.

### 6. Conclusions

This work proposes a structured approach to platform-aware programming based on platform typing, multiple dispatch, and feature detection. It has been prototyped in Julia, as a package called PlatformAware.jl. The evaluation shows that structured platform-aware programming does not cause significant performance overhead compared with ad hoc approaches. Therefore, productivity and maintainability increase due to better modularity.

It is worth saying that platform-aware programming is not a replacement for high-level heterogeneous programming approaches, which may attend to the needs of most programmers to which performance portability issues are not critical. In fact, it complements them, by attending to critical needs in performance engineering tasks.

The following initiatives are proposed to continue this work. First, to maintain and evolve PlatformAware.jl to disseminate the use of structured platform-aware programming in the Julia community and stimulate the implementation of large case studies. Second, to implement structured platform-aware programming extensions for other programming languages, as well as to investigate alternative approaches. For example, by generalizing the target features of Rust and function multi-versioning of C++. Third, to investigate how Julia’s JIT compiler may benefit from platform awareness in code generation, taking into account platform assumptions, and possibly applying the results to other dynamic compilation languages. Fourth, to address the limitations of programming languages to implement dynamic feature detection.

### Table 5. Application execution time (ImageQuilting.jl)

<table>
<thead>
<tr>
<th>kernel</th>
<th>small 512 kernel calls</th>
<th>medium 96 kernel calls</th>
<th>large 800 kernel calls</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>ad hoc</td>
<td>structured</td>
<td>ad hoc</td>
</tr>
<tr>
<td>time (CUDA)</td>
<td>0.86</td>
<td>0.85</td>
<td>18.05</td>
</tr>
</tbody>
</table>

### Table 6. Package load time

<table>
<thead>
<tr>
<th>package</th>
<th>ad hoc</th>
<th>structured</th>
</tr>
</thead>
<tbody>
<tr>
<td>ImageQuilting.jl</td>
<td>84.7s</td>
<td>92.5s</td>
</tr>
<tr>
<td>PlatformAwareQueens.jl</td>
<td>8.1s</td>
<td>11.7s</td>
</tr>
</tbody>
</table>
References


