Efficient Programming Model & GPU Usage for Scientific Computing.

Master Research and Innovation Project

Autumn 2023

Author

Isaïe [redacted], *CEA*

Supervision

[redacted], *CEA*

[redacted], *Télécom Paris*

Contents

I. Introduction – Context & Purpose

I.A. The High Performance Computing Context

High Performance Computing (HPC) programming poses various challenges, some of which can be traced back to the hardware used in that domain. Besides traditional issues linked to parallelism, the programmer also has to deal with diverse, heterogeneous architectures. Ensuring portability of a HPC code across clusters is far from trivial as projects relying on GPU usage do so through platform-specific frameworks (e.g. CUDA for Nvidia GPUs). Consequently, these code bases require significant refactors when built for a different machine.

The question is the following: What can we do to ensure code portability across existing, and even future architectures? To take it a step further, how can we ensure not only compatibility, but also portability of performance?

Currently, one answer to those questions is the Kokkos programming model and library [1]. Kokkos is defined as a *C++ Performance Portability Programming Model*. The model approaches the problem by providing abstractions over two aspects of the code: parallel execution and data management.

I.B. Beyond the Current Solution

As mentioned above, Kokkos is both a programming model and an implementation of it. However, the implementation is exclusive to C++. This makes sense from a practical perspective, as the HPC ecosystem is far more advanced in C++ than in any other language. In our case though, we would like to work, at least partially, with Rust.

The Rust programming language provides very interesting features in the HPC context, most notably easy property checking and fearless concurrency. Additionally, Rust's defensive approach for memory handling – the ownership system [2] – seems compatible, if not complementary to the Kokkos programming model.

With this in mind, the questions we seek to answer are the following:

- Can the Kokkos programming model be implemented in Rust?
- If it can be, would this implementation be viable?

The answer to the first question is non-trivial as Rust comes with much stricter compilation rules than C++, and the current Kokkos implementation makes heavy use of C++-exclusive features (most notably templates). The second question is very much similar as it requires us to define *viability* and test for it.

II. Kokkos-rs – A Proof-of-Concept

II.A. Ported Features

While the project first started with the data management aspect as the main focus, it turned into a Proof-of-Concept (PoC) [3], with all basic elements needed to write a simple parallel computational kernel. The following features were implemented:

- `Views` – This is the main abstraction Kokkos provides in terms of data managment and was the first feature to be implemented.
- `parallel_for` statement – This is the most basic parallel routine implemented by Kokkos, which corresponds to the parallelization of a `for` loop with no returned value.
- Multiple CPU backends – Kokkos supports multiple parallelization backends to ensure versatility and portability, making the implementation of this feature an essential part for the Proof-of-Concept.

The main issue encountered, which incidentally is the main issue of Rust for HPC, is the lack of support for GPGPU in pure Rust. Beyond the fact that this gates the usage of a key component of most current architectures, this also creates issues with genericity if it were to be implemented: it would need to rely on interfacing or source-to-source translation.

Aside from this, the implementation was pretty straightforward. C++ templates were translated as generics implementing custom traits, function signatures were adjusted the same way backends were handled, through Cargo `features`, i.e. conditional compilation [4]. This results in a similar user-experience between the Rust and C++ libraries, although their inner-workings greatly differ.

The PoC currently supports three backends, including the fallback one:

- `rayon` – makes use of the `rayon` crate [5] for parallelization
- `threads` – makes use of standard library threads [6] for parallelization
- `serial` – executes the code using a regular serial statement. This is the fallback implementation.

For more details on implementations, refer to the PoC documentation [7]. A simple program implementation using both the PoC and the original library can be found in Section V.A.

II.B. Testing Methodology

In order to test the viability of the model in Rust, multiple kinds of benchmarks were written:

- BLAS kernels [8], written both using the PoC and the C++ implementation. Note that the C++ benchmarks were not meant to be optimized, but rather equivalent to their Rust counterparts.
- Abstraction cost evaluations: This includes benchmarking basic operations on `Views` as well as comparing hardcoded kernels to ones written using the library.
- A study of the model using the PoC on simple problems that Kokkos tackles.

Beyond plain performance, a *Kokkos-rs* crate would need to meet other criteria to be considered viable. This would include, for example, user-friendliness as well as problem coverage of similar level to the original Kokkos C++ implementation.

III. Results – Benchmarking & Analysis

III.A. Comparing Implementations

Using the BLAS kernels with fixed-size vectors and matrices, the following times were recorded:

Benchmark	Data size	Kokkos-rs	Kokkos (C++)	Hardcoded (Rust)
AXPY	2^{20}	*2.0349ms*	*0.182ms*	–
GEMV	2^{12}	*15.616ms*	*3.571ms*	–
GEMM	2^{10}	*212.97ms*	*169.8ms*	*173.50ms*

Times presented are the average values over 100 samples. They were measured on parallel execution over 8 threads (no hyperthreading), using `rayon` as the Kokkos-rs backend and OpenMP for the C++ version.

The first observation to make is that, although the original C++ library is faster, times are not magnitudes apart. Additionally, the difference seems less exacerbated for kernels with a heavier workload: Execution time is shorter tenfold for AXPY while it barely cuts down 25% for GEMM.

This could indicate a lack of optimization or performance difference between the dispatching & scheduling code of each implementation. `rayon` in particular isn't known for being fast at those tasks, which could be a side-effect of its work-stealing model. The granularity over parallel tasks probably also varies between the two models, but the importance of that also depends on the workload.

The hardcoded version of the GEMM kernel clearly indicates that similar performance can be achieved with Rust's parallelization, confirming the previous observations related to optimization.

One fundamental pitfall of the current PoC implementation is its usage of generic over specific code: In C++, templates can be used in order to generate specific code at compile time. Conditional compilation in Rust only covers basic use-cases (think `IFDEF`), which means that most of the code needs to use generic types implementing custom traits (think interfaces). This comes at a cost in performance that would be interesting to assess, whether for the PoC or for the language as a whole.

III.B. Layout Study

In order to assess the relevance of a Kokkos-rs implementation, we decided to realize benchmarks on a common case where the Kokkos library achieves performance through its approach. By measuring performance on the GEMM benchmark, using different memory layout setups, over a range of different data sizes, it was possible to highlight the phenomenon that Kokkos' abstractions mean to tackle.

The GEMM kernel corresponds to the following operation (α, β scalars, A, B, C matrices):

$$C = \alpha A.B + \beta C$$

Essentially, the bottleneck of the operation is the matrix-matrix product $A.B$, and its completion speed can be heavily dependent on the underlying memory layout of both matrices. Detailed definitions of both *layout* and *stride* can be found in Section V.B.

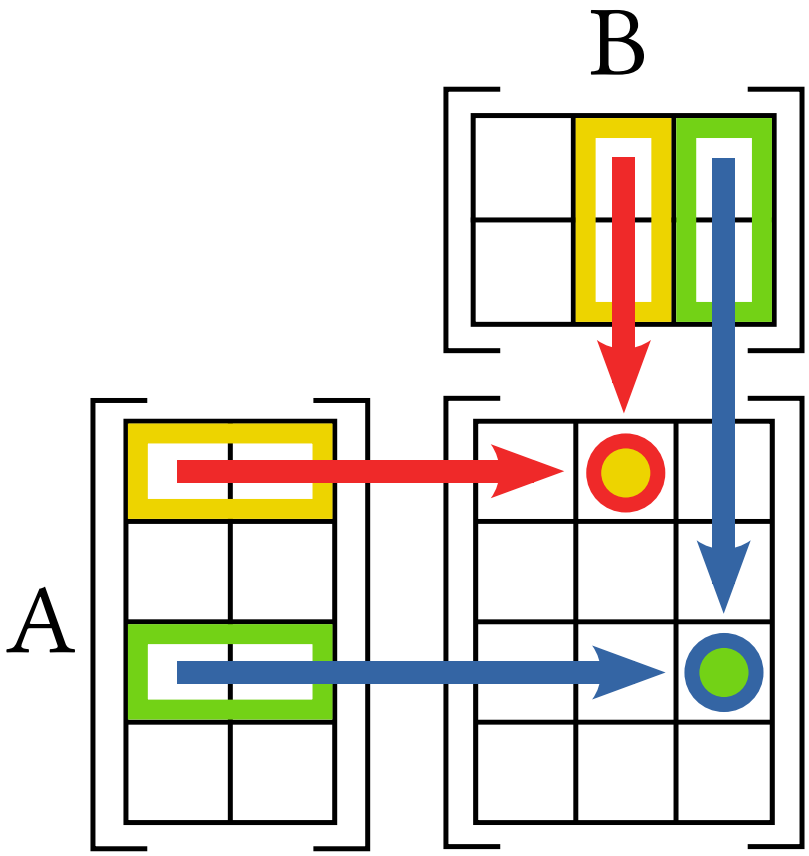

Figure 1: Matrix-matrix product [9].

In the case of CPU execution, we want elements of an A **row** to be stored contiguously, while we want elements of a B **column** to be stored contiguously. This is achieved by storing matrices A and B using respectively a `LayoutRight` and `LayoutLeft`.

The following performances were measured using 1024 by 1024 matrices of `f64`. Note that the hardware used to run these benchmarks is different from the hardware used in the previous section.

Setup	**A layout**	**B layout**	t_{average}
Ideal	`Layout::Right`	`Layout::Left`	*152.83ms*
Usual	`Layout::Right`	`Layout::Right`	*245.52ms*
Worst	`Layout::Left`	`Layout::Right`	*371.09ms*

Significant speedup is observed when using the ideal setup over the usual one. These effects, are observable on both CPU and GPU, but further exacerbated on GPU. We can also note as

a rule of thumb that the ideal setup on CPU is the worst on GPU, and vice versa, hence the importance of layout abstraction provided by Kokkos.

On CPU, this change in performance can be attributed to cache usage. The following graph shows the speed loss when using a regular setup compared to the ideal one, over a range of different matrix sizes:

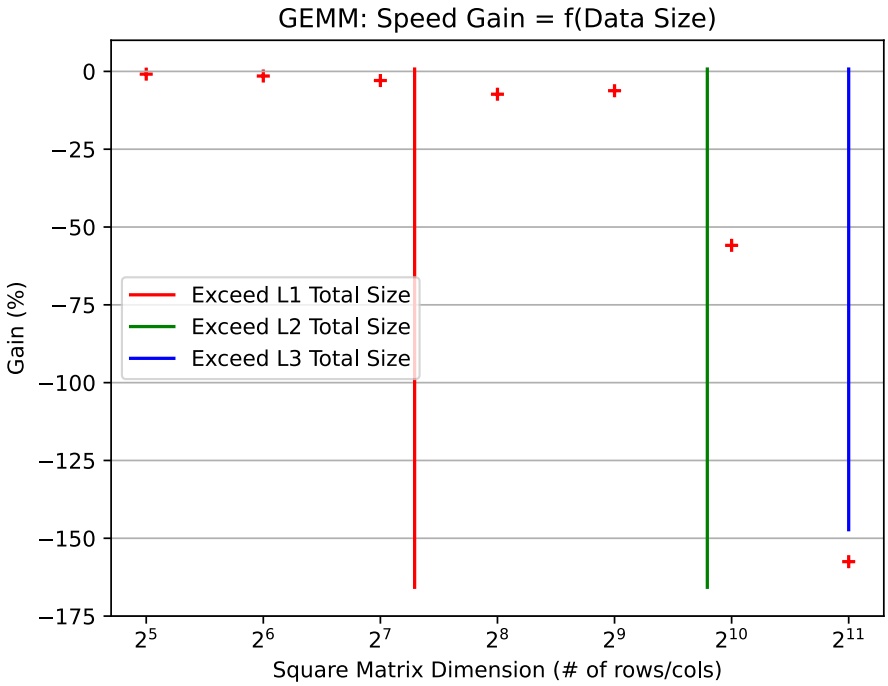

Figure 2: Speed variation between layouts.

We can observe a direct correlation between performance drops and thresholds where the size of matrix B exceeds total cache size. As a reference, those benchmarks were executed using an AMD Ryzen 5 7600x with the following characteristics:

- 6 Cores, 12 Threads
- Caches (NUMA system):
  - L1d: 32KB per core (192KB total)
  - L2: 1024KB per core (6MB total)
  - L3: 32MB total

We can compute B's size according to its dimensions. Note that benchmarks were done using matrices of `f64`, meaning each element has a size of 2^3 bytes. We ignore other elements as their size is negligible compared to the size of B. Still, note that they contribute to cache saturation when B's size equals a given cache's capacity.

N	2^4	2^5	2^6	2^7	2^8	2^9	2^{10}	2^{11}	2^{12}
# of elements	2^8	2^{10}	2^{12}	2^{14}	2^{16}	2^{18}	2^{20}	2^{22}	2^{24}
Size	*2KB*	*8KB*	*32KB*	*128KB*	*512KB*	*2MB*	*8MB*	*32MB*	*128MB*

The dimension thresholds are the following:

- $2^5/2^6$: Matrix B (8KB/32KB) equals L1 per-core capacity (32KB).
- $2^7/2^8$: Matrix B (128KB/512KB) exceeds L1 total capacity (192KB).
- $2^9/2^{10}$: Matrix B (2MB/8MB) exceeds L2 total capacity (6MB).
- $2^{10}/2^{11}$: Matrix B (8MB/32MB) equals L3 total capacity (32MB).

The first threshold does not appear on the speed variation measurements, but we can profile the GEMM kernel and fetch L1 cache miss rates using `perf`. Doing so over a range of matrix sizes yields the following results:

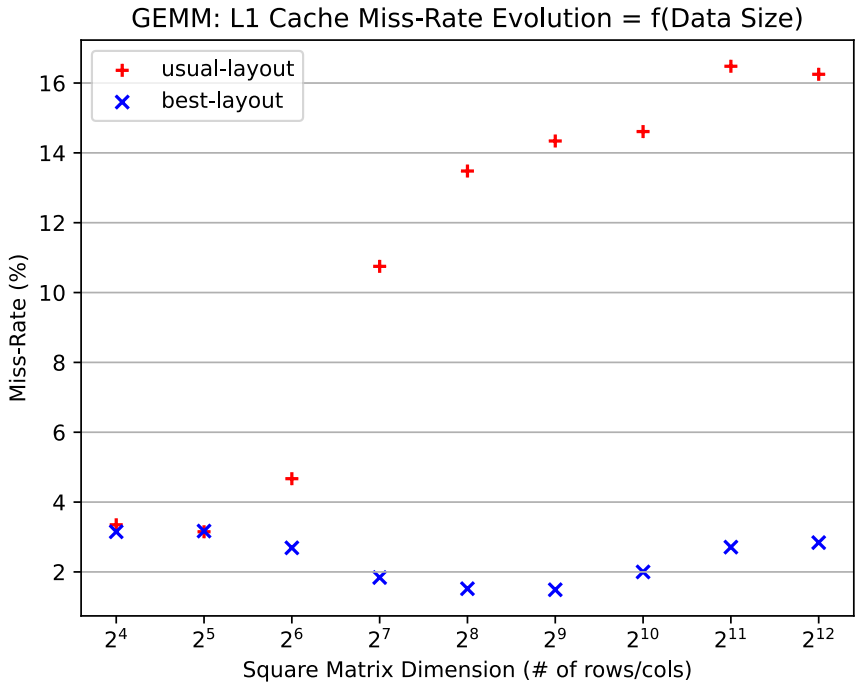

Figure 3: L1 cache miss rate.

This highlights the aforementioned threshold. Under it, the miss rate was essentially identical for both layouts. However, when B doesn't entirely fit in the L1 cache, the miss rates diverge as the usual layout generates much more misses. **This is due to the lack of locality of reference induced by a higher stride between two consecutive elements of a same column**.

Two additional observations can be made, but have not been explored:

- Miss rates seem to reach an upper bound past some point.
- The concave up curve made by the best layout's rates isn't due to noise or error in sampling, it is an actual tendency.

The equivalent measurements have been made for a sequential execution of the GEMM kernel and can be found in Section V.C.

IV. Conclusion

IV.A. Viability of Kokkos-rs

The following can be concluded from the implementation process as well as results from the testing phase:

- The Kokkos programming model is compatible with Rust: Backend abstractions as well as data management abstractions can be implemented using the language. Moreover, the model benefits from mutability checks and struct-like enums for type definition.
- Similar level of performance can probably be achieved by a Rust implementation, the main challenge being the usage of generic code (generic types implementing traits) over specific code (templates).
- The lack of GPU support for Rust code inherently limits the level of portability Kokkos-rs could offer at the moment.

However, multiple features being considered for implementation or stabilization in the language could prove useful for a Kokkos-rs crate. Most notably:

- Stabilization of the `Fn*` traits [10] would allow to define kernels as structures (functors), hence making the fallback process a type conversion implementation.
- Stabilization of `impl Trait` in type aliases [11] would greatly simplify the code and improve refactoring capabilities, hence make supporting additional backends easier.

IV.B. What Next ?

A number of possibilities emerge for any continuation of this Proof-of-Concept:

- Develop further the Proof-of-Concept by adding more existing Kokkos features. Most notably, tiling, team-based execution policies, or other parallel statements. Optimization could also be considered as the current implementation is closer to tinkering than anything else.
- Use the CXX crate [12] for interoperability between the Proof-of-Concept and the original C++ library. Such a usage would still be useful in the context of HPC programming as our main interest is property checking, not thread safety of kernels.
- Extend the Proof-of-Concept to support hybrid tools, e.g. `cudarc` [13]. This would allow GPU targeting, although this is partially dependent on feature stabilization and doesn't solve the fallback issue.

V. Appendices

V.A. Rust / C++ Simple Program

These code snippets correspond to an implementation of the third level of Basic Linear Algebra Subprograms (BLAS) functionality: *GEMM*, a general matrix multiplication [8].

```rust
// build the exec policy
let execp = ExecutionPolicy {
        space: ExecutionSpace::DeviceCPU,
        range: RangePolicy::RangePolicy(0..length),
        schedule: Schedule::Static,
    };

// C = alpha * A * B + beta * C
parallel_for(
    execp,
    |arg: KernelArgs<1>| match arg { // using an enum as kernel argument
        KernelArgs::Index1D(i) => {
            for j in 0..length {
                let ab_ij: f64 = (0..length)
                                    .map(|k| aa.get([i, k]) * bb.get([k, j]))
                                    .sum();
                let val: f64 = alpha * ab_ij + beta * cc.get([i, j]);
                cc.set([i, j], val);
            }
        }
        _ => unimplemented!(),
    }).unwrap();
```

```cpp
++
// C = alpha * A * B + beta * C
Kokkos::parallel_for(
    "GEMM kernel", // kernel name
    length,        // Kokkos defines a minimal signature through templating/overloading
    KOKKOS_LAMBDA(const uint64_t i) {
        for (uint64_t j = 0; j < length; j++) {
            double AB_ij = 0.0;
            for (uint64_t k = 0; k < length; k++) { AB_ij += A(i,k) * B(k,j); }
            // assign to C
            C(i, j) = alpha * AB_ij + beta * C(i, j);
        }
    });
```

V.B. Layout and Stride Definitions

What Kokkos calls layout refers *to "the mapping from a logical multidimensional index (i, j, k, ...) to a physical memory offset"*. The mapping process is characterized by the stride associated to each dimension/index.

The stride [14], in our context, refers to the number of memory units between the beginning of two successive elements along one index. For example, the space between

`Arr[i_fixed, j, k_fixed]` and `Arr[i_fixed, j + 1, k_fixed]` is the stride associated to index `j`; `Arr`'s layout is defined by three strides, one for each dimension.

Kokkos provides three main layouts, which strict definitions rely on the notion of stride:

- `LayoutLeft`: Lowest stride for the first index, increasing stride as index increases.
- `LayoutRight`: Highest stride for the first index, decreasing stride as index increases.
- `LayoutStride`: A custom layout that allows the user to control the exact way data is positioned in memory. Note that this allows creation of sparse data structures.

A more intuitive approach to understand the first two definitions is their 2D interpretations:

- `LayoutLeft`: "Column-major" layout where two successive column elements are stored contiguously in memory (ignoring alignment-related offset). This is how Fortran operates.
- `LayoutRight`: "Row-major" layout where two successive row elements are stored contiguously in memory (ignoring alignment-related offset). This is how C, C++, and Rust operate.

V.C. Serial L1 Cache Miss Rates

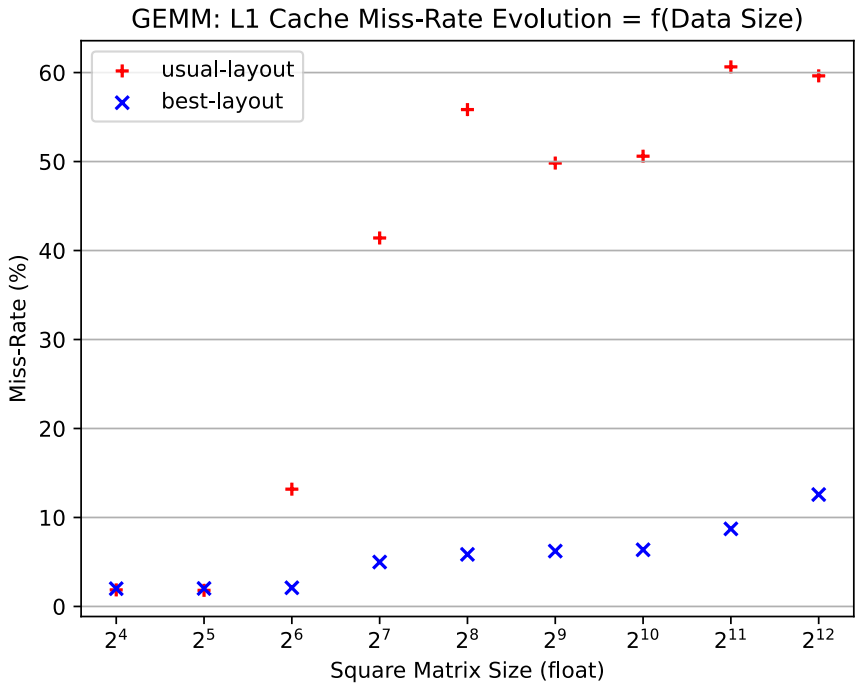

Figure 4: Serial L1 cache miss rate.

Bibliography

[1] "Kokkos: The Programming Model." Accessed: Sep. 21, 2023. [Online]. Available: https://kokkos.github.io/kokkos-core-wiki/index.html

[2] "What is Ownership? - The Rust Programming Language." Accessed: Dec. 07, 2023. [Online]. Available: https://doc.rust-lang.org/book/ch04-01-what-is-ownership.html

[3] "imrn99/poc-kokkos-rs: Kokkos-rs: A Proof of Concept." Accessed: Dec. 08, 2023. [Online]. Available: https://github.com/imrn99/poc-kokkos-rs

[4] "Features - The Cargo Book." Accessed: Dec. 07, 2023. [Online]. Available: https://doc.rust-lang.org/cargo/reference/features.html

[5] "rayon - Rust." Accessed: Dec. 08, 2023. [Online]. Available: https://docs.rs/rayon/latest/rayon/

[6] "std::thread - Rust." Accessed: Dec. 08, 2023. [Online]. Available: https://doc.rust-lang.org/std/thread/

[7] "Kokkos-rs: Github Deployment." Accessed: Dec. 08, 2023. [Online]. Available: https://imrn99.github.io/poc-kokkos-rs/

[8] "Basic Linear Algebra Subprograms." Accessed: Dec. 08, 2023. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Basic_Linear_Algebra_Subprograms&oldid=1184327524

[9] "Matrix Multiplication." [Online]. Available: https://en.wikipedia.org/wiki/Matrix_multiplication#/media/File:Matrix_multiplication_diagram_2.svg

[10] "Tracking issue for Fn traits (`unboxed_closures` & `fn_traits` feature) · Issue #29625 · rust-lang/rust." Accessed: Dec. 08, 2023. [Online]. Available: https://github.com/rust-lang/rust/issues/29625

[11] "Tracking issue for RFC 2515, "Permit impl Trait in type aliases" · Issue #63063 · rust-lang/rust." Accessed: Dec. 08, 2023. [Online]. Available: https://github.com/rust-lang/rust/issues/63063

[12] D. Tolnay, "CXX — safe FFI between Rust and C++." Accessed: Sep. 21, 2023. [Online]. Available: https://github.com/dtolnay/cxx

[13] C. Lowman, "cudarc: minimal and safe api over the cuda toolkit." Accessed: Dec. 09, 2023. [Online]. Available: https://github.com/coreylowman/cudarc

[14] "Stride of an array." Accessed: Dec. 08, 2023. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Stride_of_an_array&oldid=1171532311