ETH zürich

Rust programming language in the high-performance computing environment

Michal Sudwoj
msudwoj@student.ethz.ch

11th September 2020

Advisor: Dr. Roger Käppeli

D MATH

Abstract

Fortran and C++ have traditionally been the languages of choice for high-performance computing (HPC) applications. However, they are both over 35 years old, and do not offer much in terms of user-friendliness or memory safety. Rust is an emergent new systems language, aiming to be performant while offering such safety and usability, as well as bundling tools that a modern developer needs.

We compare multiple implementations of a finite difference stencil code, and show that idiomatically written Rust programs can be just as performant as their Fortran or C++ counterparts, while offering the above-mentioned advantages.

Contents

Chapter 1

Introduction

> *C makes it easy to shoot yourself in the foot; C++ makes it harder, but when you do it blows your whole leg off.*
> — Bjarne Stroustrup [46]

While Fortran and C++ are the langauges of choice for HPC applications, they offer little in terms of safety; by default, it is the developers responsibility to not access out-of-bound or invalid memory, to guarantee lack of data races and to make sure that pre- and post-conditions are upheld. Rust is an emergent systems programming language designed with safety in mind, while aiming to be just as performant; it shifts many of the aforementioned responsibilities from the programmer to the compiler, while allowing for manual override. In this thesis, we show how to use Rust on Piz Daint, a HPC system at Centro Svizzero di Calcolo Scientifico (CSCS), and compare the performance of 4th-order numerical diffusion codes written idiomatically in Fortran, C++ and Rust.

In chapter 2, we show that Rust is a viable alternative to Fortran and C++ with respect to its features. Chapter 3 introduces the benchmark and discusses its results. In chapter 4, we offer insight to making Rust more attractive for HPC development.

The author would like to thank Dr. Roger Käppeli for his guidance in writing this thesis; Dr. Anton Kozhevnikov, Teodor Nikolov, Simon Frash, and everyone else at CSCS for their help in using Piz Daint; and Dr. Oliver Fuhrer, for allowing the author to take code from his lecture *High Performance Computing for Weather and Climate* as a basis for implementation.

Chapter 2

Programming on a HPC system

> [*Most people*] *think FORTRAN's main contribution was to enable the programmer to write programs in algebraic formulas instead of machine language. But it isn't. What FORTRAN did primarily was to mechanize the organization of loops.*
> — John Backus [23]

2.1. Motivation

It is estimated that between 49% and 88% of bugs in software are caused by memory unsafety issues such as use-after-free, double-free, buffer over- and underflows, use of uninitialized memory, or data races [17]. Scientific software is not exempt from this problem: in a short questionnarie at the Scientific Software & Libraries (SSL) group at CSCS, over half of the participants indicated that memory safety issues are encountered on a regular basis (see fig. 2.1). While choice of programming language and related technologies is important to scientific software developers, this is usually dictated by support interfacing to legacy systems, or convenient usage of modern technologies, architectures, and developer familiarity. Some research has been done on performance and productivity of different programming languages, it assumes all contributors will have expert knowledge of the programming language they are working with [11]. Seeing as more and more scientific frameworks are being developed in the open-source model, we do not find that assumption to be reasonable: one cannot assume that all contributors will be experts. However, this should not deter the community from open-source development. Instead, we believe that user-friendly languages such as Rust have potential to gain traction in the scientific community, as we find them to be easier to teach and learn, and more inline with todays coding standards[1]

2.2. Background

Fortran (formerly FORTRAN) is the oldest of the programming languages we consider in this thesis. It was created in the 1950s at IBM by J. W. Backus et al. [4], and featured

[1] Rust RFCs often feature a "How do we teach this?" section dedicated to informing the community about the planned change, and providing outlining which beginner-friendly resources would need to be created, were the RFC accepted.

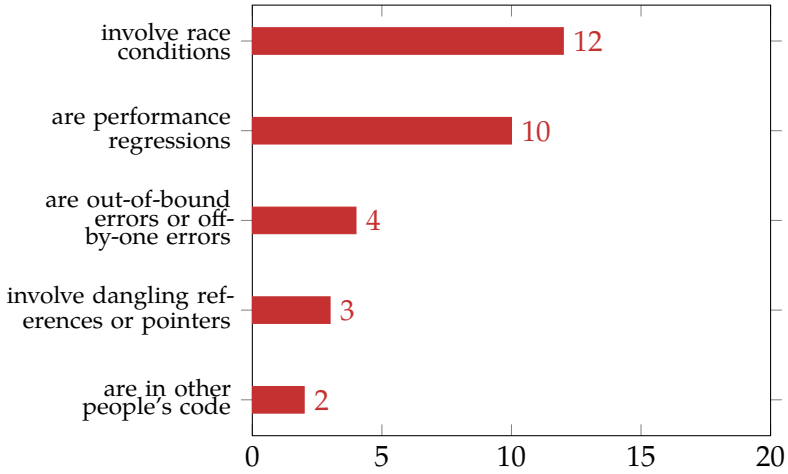

Figure 2.1.: Answers of SSL employees to the question "Most bugs I encounter..." ($N = 19$).

```
program main
  print *, "Hello, World!"
end program
```

Listing 1: A "Hello, World!" program in Fortran.

the first optimizing compiler [41]. The first Fortran standard was published by American National Standards Institute (ANSI) in 1966 [61], and revised in 1978 [62]. Further standards were published by International Organization for Standardization (ISO) in 1991 (Fortran 90) [33], 1997 (Fortran 95) [29], 2004 (Fortran 2003) [30], 2010 (Fortran 2008) [31] and 2018 [32], adding support for free-form input, modules, dynamic memory allocation (Fortran 90); object-oriented programming (Fortran 2003); and co-arrays (Fortran 2008).

Compared to C++ and Rust, Fortran's defining features are its built-in support for multi-dimesional arrays, array slicing, and co-arrays. However, it lacks support for conditional compilation[2], compile time-programming, inline assembly, and does not come with a standard library. Generic programming is supported, but not ergonomic. Compiler conformance varies strongly between vendors, with support for the newest standards being the least prevalent. It is not uncommon for compilers to provide their own extensions to alleviate percieved deficiencies of the standard.

C++ was developed in 1979 at AT&T Bell Laboratories (now Nokia Bell Laboratories), by Bjarne Stroustrup. It was not made public until 1985, when *The C++ Programming Language* appeared. ISO standardised the language in 1998 [24], and revised it in 2003 [25]. Since 2011, the ISO C++ Standard follows a three-year release schedule [26, 27, 28, 34].

[2]most compilers support preprocessing using a C-like preprocessor. This is however not part of the Fortran standard.

```cpp
#include <iostream>

int main() {
  std::cout << "Hello, World!" << std::endl;
}
```

Listing 2: A "Hello, World!" program in C++.

C++ features support for the C preprocessor as a means of conditional compilation. C++11 added support for compile-time programming using **`constexpr`**, with newer standards expanding the functionality. Templates provide support for generic functions and types, as well as allowing template meta-programming. Modern compilers adhere to the standard well, and conformance is tracked [9].

Rust is the youngest programming language of the three. Its development was started in 2006 at Mozilla Research by Graydon Hoare, with the version 1.0 being release in 2015. Since then, development has been taken over by the core team, with releases happening every 6 weeks.

Rust distinguishes itself from Fortran and C++ by bundling tooling with the compiler, allowing for easy dependency management, testing, and documentation generation. Much effort is put into making compiler messages clear and understandable: this includes providing suggestions for fixing code and path trimming [1]. Development of the language happens online, and is accessible to anyone interested, with the core team and community being open to contributions[3]. However, the Rust language and its reference compiler `rustc` are heavily intertwined. To date, no other compiler exists for Rust. Furthermore, Rust development is ongoing—it is by far not as mature of a language as Fortran or C++, as evidenced by the large amount of crucial features that are currently in development, such as constant functions, constant generics, specialization, generic associated types and support for GPU targets. On the other hand, being a young languge, Rust does away with a lot of the cruft of the past such as mutable-by-default variables, while allowing the programmer to opt-in to dangerous features when it is required—unsafe blocks limit such code to small small sections, whose contraints can easily be verified for soundness. Macros in Rust are hygienic, meaning that identifiers introduced in the macro will not "leak" or collide by chance with one declared by the user, as is the case in C and C++. The syntax of macros is also much more ergonomic: macros can be declared "by example", with subsitiution placeholders for tokens when called. When more flexibility is needed, procedural macros can be written, which are Rust functions called at compile-time that transform the token stream. The Rust team take great care when extending the language and reference compiler, making sure that there are no regressions of runtime performance or compile-times, and scanning the whole public ecosystem before introducing breaking changes so as to assess the impact

[3]Incidentaly, the author was able to successfully upstream a patch [50] to the compiler, with guidance from the community.

```rust
fn main() {
  println!("Hello, World!");
}
```

Listing 3: A "Hello, World!" program in Rust.

```
> curl --proto '=https' --tlsv1.2 -sSf https://sh.rustup.rs |
 ↪  sh
> rustup toolchain install stable
> rustup toolchain install nightly
> rustup toolchain install nightly-2020-05-01
> rustup target    install nvptx64-nvidia-cuda
```

Listing 4: Example of installing `rustup` as well as different toolchains.

(in any case, automatic migration tools are provided with such changes).

2.3. Rust as a HPC programming language

Installing Rust is as easy as following the instructions on `rustup.rs`. The `rustup` [45] installer provides automatically generated binaries of the Rust toolchain; in addition, it allows us to install specific versions of Rust, extra targets and tools such as `rustfmt` [44], the Rust code formatting utility.

Rust can also be installed from the operating system package manager (in case of Linux distributions), or from source using `spack` [16]. However, we do not reccomend these approaches, as we consider the `rustup` installer to be superior. In comparision to a system package manager, it allows for installation of multiple versions of Rust, as well as arbitrary historical versions, should those be needed eg. to reproduce results from past studies. In comparision to `spack`, the `rustup` supports installation of arbitrary nightly versions of the toolchain[4].

In terms of language and library features, Rust does not include support for complex numbers or multi-dimesional arrays out of the box. However, mature Rust libraries exist to support these features: `num-complex` for complex numbers; and `ndarray` and `nalgebra` for array and matrix support [12, 40, 56]. Bindings for BLAS and LAPACK are provided by the `blas`, `lapack`, `cblas`, `lapacke` crates [8]. A pure Rust implementation of general matrix multiplication is implemented in `matrixmultiply` [55]. MPI, HDF5 and netCDF are supported through the `mpi`, `hdf5` and `netcdf` crates, respectively, which are nearly feature-complete [**mpi**, **hdf5**, 38]. Bindings to other libraries can be generated in an automated manner using `bindgen`, while for the inverse case `cbindgen` can be used [5, 22]. `cxx` and `autocxx` provide additional features [3,

[4] until 2020-09-08, support for installing extra targets, such as the NVIDIA GPGPU toolchain was also lacking in `spack` [**SudwojRustaddednvptx**].

```
> export CARGO_TARGET_X86_64_UNKNOWN_LINUX_GNU_RUSTFLAGS="-C
 ↪  target_cpu=${CRAY_CPU_TARGET:-haswell} -C
 ↪  relocation-model=dynamic-no-pic"
> export CARGO_TARGET_NVPTX64_NVIDIA_CUDA_RUSTFLAGS="-C
 ↪  target-cpu=sm_60 -C target-feature=+sm_60,+ptx60 -C
 ↪  relocation-model=dynamic-no-pic"
> export MPICC=cc
> export CUDA_LIBRARY_PATH=${CUDATOOLKIT}/lib64
```

Listing 5: The environment flags used on Piz Daint.

58]. Multi-threading and shared-memory parallelism is supported by the `rayon` crate, while `crossbeam` and `parking_lot` provide optimized low-level synchronization primitives [13, 18, 43].

We find Rust, and its ecosystem, to lack support for GPU programming, however. Firstly, there is currently no compiler backend support for AMD graphic cards, only for NVIDIA ones. Secondly, this support is only preliminary, and not automatically tested; calling a GPU kernel function or accessing the thread index requires unstable features, and therefore `nightly` Rust. Thirdly, most GPU code written in Rust will be **`unsafe`** or even unsound, as the compiler cannot reason about the GPU memory model. Lastly, some performance-critical features, such as CUDA shared and constant memory, cannot be mapped to the current Rust memory model, and are therefore not implemented; we also doubt that they will be in the foreseeable future. Crates such as `rustacuda`, `ptx-builder`, `ptx-support` and `accel` do their best to provide as complete of a CUDA programming experience in Rust as possible [21, 57, 64, 65]. A good overview of the current Rust ecosystems for machine learning, scientific computing and GPU offloading can be found at `https://www.arewelearningyet.com` [39].

In order to fully make use of Rust on a HPC system, some environment variables might need to be set. For starters, certain crates might need help finding system libraries or tools, eg. `mpi` requires `MPICC` to be set to the MPI C compiler wrapper, while `rustacuda` requires `CUDA_LIBRARY_DIRECTORY` to locate the CUDA runtime libraries. Furthermore, code-generation options might need to be adjusted; if linking using the Cray toolchain on Piz Daint, we found that we had to set `RUSTFLAGS="-C relocation-model=dynamic-no-pic"`. Lastly, in order to optimize for the underlying architecture, the target CPU should be specified, eg. `RUSTFLAGS="-C target-cpu=haswell"`. The complete reccomended set of environment variables for Piz Daint can be found in listing 5.

Chapter 3

Performance

> *Programmers waste enormous amounts of time thinking about, or worrying about, the speed of noncritical parts of their programs, and these attempts at efficiency actually have a strong negative impact when debugging and maintenance are considered. We **should** forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil.*
>
> *Yet we should not pass up our opportunities in that critical 3%.*
> — Donald E. Knuth [35]

3.1. Previous work

We provide a short overview of the literature known to us that compares the performance of Rust to other languages.

- Sverdrup implemented matrix multiplication in Rust, being **23% slower than OpenBLAS** [53, 54]
- Wilkens compares C, Go and Rust implementations of Dijkstra's algorithm (finding the single-source shortest paths in a graph) in terms of productivity and performance [60]; finding the Rust implementation to be **26–50% faster than C**, and **7–17% faster than Go**, depending on the number of threads used.
- Perez compares C++ and Rust implementations of the algorithmic Lovász Local Lemma (determining the probability that none of a set of event will occur, where each event is nearly independent from all others) in terms of productivity and performance [42]; finding Rust to be **43–78% faster than C++ in serial code**[1].
- McKeogh compares Fortran and Rust implementations of shallow-water equations [37](a system of hyperbolic partial differential equations, being an approximation to the Navier-Stokes equation, when the depth of the fluid is much smaller than the horizontal domain); finding Rust to be between **59% slower** and **30% faster** than Fortran, depending on problem size.

[1] Rust was also **43–78% faster than C++ in parallel code**. However, the C++ parallel version was slower than the C++ serial version, with 60% of time being spent creating and joining threads. This author presumes a thread pool was not used.

- Lindgren compares C++ and Rust performance of a constrained version of the all-pairs shortest paths problem from graph theory [36]; finding the final Rust version to be **10% slower** than the final C++ version[2].
- Hahn compares Python, Go and Rust implementations of `diffimg`, a tool for comparing images [20]; finding Rust to be **35–58% faster than Go**, and **65–80% faster than Python**, depending on problem size.
- Blanco-Cuaresma and Bolmont compare C, Fortran, Go, Julia and Rust implementations of a simple N-body physics simulation [6, 7]; finding Rust to be **12% faster than C**, **22% faster than Fortran**, **36% faster than Julia**, and **39% faster than Go**.
- Atcheson compares C++ and Rust implementations of the generalized minimal residual method (an iterative method for finding the solution to a non-symmetric system of linear equations) [2].

The above-mentioned works show that Rust achieve performance comparable to other languages under a variety of conditions; however, we find all the above references to be either to specialized to a specific problem, or to not offer a fair comparison, due to optimization deficiencies in implementation. In this thesis, we would like to show that that is also the case for numerical codes, as could be written by a domain scientist. To this end, we implement a stencil code as detailed in sections 3.2 and 3.3 in Fortran, C++, and Rust.

3.2. Background

> *Finite difference approximations have a more complicated "physics" than the equations they are designed to simulate. This irony is no paradox, however, for finite differences are used not because the numbers they generate have simple properties, but because those numbers are simple to compute.*
> — Lloyd N. Trefethen [59]

As a basis for our language comparison, we choose the fourth-order numerical diffusion equation in the xy-plane,

$$\frac{\partial \phi}{\partial t} = -\alpha_4 \nabla^4 \phi = -\alpha_4 \Delta^2 \phi = -\alpha_4 \Delta(\Delta \phi), \tag{3.1}$$

where ϕ is a scalar-valued function of space and time. Such an equation could be eg. used as a noise filter in a climate or weather simulation [63]. More importantly, we discretize this equation using finite differences, which are abundantly used in scientific codes, and which leads us to believe that this is a suitable benchmark.

[2] During the optimization process, Rust was between **27% slower** and **18% faster** than the corresponding C++ version.

We solve eq. (3.1) on a cubic grid. Choosing a uniform spatial discretization in the horizontal plane and time, and an arbitrary discretization in the vertical direction, we define

$$\phi^n_{i,j,k} \overset{\text{def}}{\approx} \phi(x_i, y_j, z_k, t^n) \tag{3.2}$$

to be the approxiamte value of the function at the grid points

$$x_i \overset{\text{def}}{=} i \cdot \Delta x \qquad \text{where } 1 \le i \le n_x, \tag{3.3a}$$

$$y_j \overset{\text{def}}{=} j \cdot \Delta y \qquad \text{where } 1 \le j \le n_y, \tag{3.3b}$$

$$z_k \overset{\text{def}}{=} z(k) \qquad \text{where } 1 \le k \le n_z \text{ and} \tag{3.3c}$$

$$t^n \overset{\text{def}}{=} n \cdot \Delta t \qquad \text{where } 1 \le n \le T. \tag{3.3d}$$

We then approximate the laplacian using a second-order central finite difference,

$$\Delta\phi^n_{i,j,k} \approx \frac{-4\phi^n_{i,j,k} + \phi^n_{i-1,j,k} + \phi^n_{i+1,j,k} + \phi^n_{i,j-1,k} + \phi^n_{i,j+1,k}}{\Delta x \Delta y}. \tag{3.4}$$

For the time derivative, we use a first-order forward difference (explicit Euler method),

$$\frac{\partial \phi^n_{i,j,k}}{\partial t} \approx \frac{\phi^{n+1}_{i,j,k} - \phi^n_{i,j,k}}{\Delta t}. \tag{3.5}$$

Solving for $\phi^{n+1}_{i,j,k}$, we get the following finite difference scheme,

$$\psi^n_{i,j,k} = \frac{1}{\Delta x \Delta y}\left(-4\phi^n_{i,j,k} + \phi^n_{i-1,j,k} + \phi^n_{i+1,j,k} + \phi^n_{i,j-1,k} + \phi^n_{i,j+1,k}\right) \tag{3.6a}$$

$$\phi^{n+1}_{i,j,k} = \phi^n_{i,j,k} - \frac{\alpha_4 \Delta t}{\Delta x \Delta y}\Delta\left(-4\psi^n_{i,j,k} + \psi^n_{i-1,j,k} + \psi^n_{i+1,j,k} + \psi^n_{i,j-1,k} + \psi^n_{i,j+1,k}\right), \tag{3.6b}$$

which can be further simplified to

$$\begin{aligned}\phi^{n+1}_{i,j,k} = \phi^n_{i,j,k} - \frac{\alpha_4 \Delta t}{(\Delta x)^2 (\Delta y)^2}\Big(& -20\phi^n_{i,j,k} \\ & + 8\phi^n_{i-1,j,k} + 8\phi^n_{i+1,j,k} + 8\phi^n_{i,j-1,k} + 8\phi^n_{i,j+1,k} \\ & - 2\phi^n_{i-1,j-1,k} - 2\phi^n_{i-1,j+1,k} - 2\phi^n_{i+1,j-1,k} - 2\phi^n_{i+1,j+1,k} \\ & - \phi^n_{i-2,j,k} - \phi^n_{i+2,j,k} - \phi^n_{i,j-2,k} - \phi^n_{i,j+2,k}\Big)\end{aligned} \tag{3.7}$$

Equations (3.6a) and (3.6b) can be expressed as the stencil shown in fig. 3.1, while eq. (3.7) gives rise to the stencil in fig. 3.2.

For ease of implementation, we define $\alpha \overset{\text{def}}{=} \alpha_4 (\Delta t)(\Delta x)^{-2}(\Delta y)^{-2}$; this can be interpreted as $\Delta t = \Delta x = \Delta y = 1$. In order for the schemes to be stable, α must satisfy the CFL condition; we assume this is the case, and choose an appropriately small α for all our experiments.

Implementing eqs. (3.6a) and (3.6b) in code results in algorithm 1, which we name **`laplap`**. We note that this scheme requires a temporary array of size $n_x \cdot n_y$. In turn eq. (3.7) results in algorithm 2, which we name **`inline`**.

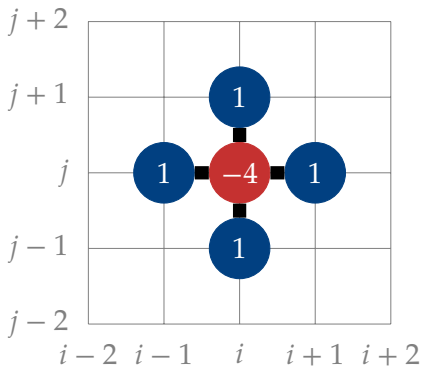

Figure 3.1.: Five-point stencil used in `laplap`.

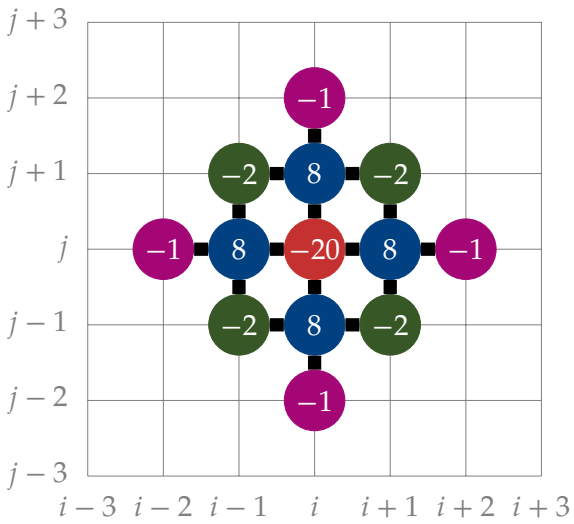

Figure 3.2.: Thirteen-point stencil used in `inline`.

```
for n ← 1 to T do
    for k ← 1 to n_z do
        for j ← 1 to n_y do
            for i ← 1 to n_x do
                tmp_{i,j} ← −4 · in_{i,j,k} + in_{i−1,j,k} + in_{i+1,j,k} + in_{i,j−1,k} + in_{i,j+1,k};
            end
        end
        for j ← 1 to n_y do
            for i ← 1 to n_x do
                laplap ← −4 · tmp_{i,j} + tmp_{i−1,j} + tmp_{i+1,j} + tmp_{i,j−1} + tmp_{i,j+1};
                if n = T then
                    out_{i,j,k} ← in_{i,j,k} − α · laplap;
                else
                    in_{i,j,k} ← in_{i,j,k} − α · laplap;
                end
            end
        end
    end
end
```

Algorithm 1: The `laplap` algorithm.

```
for n ← 1 to T do
    for k ← 1 to n_z do
        for j ← 1 to n_y do
            for i ← 1 to n_x do
                out_{i,j,k} ← (−20α+1)·in_{i,j,k}+8α·in_{i−1,j,k}+8α·in_{i+1,j,k}+8α·in_{i,j−1,k}+
                 8α · in_{i,j+1,k} − 2α · in_{i−1,j−1,k} − 2α · in_{i−1,j+1,k} − 2α · in_{i+1,j−1,k} − 2α ·
                 in_{i+1,j−1,k} − α · in_{i−2,j,k} − α · in_{i+2,j,k} − α · in_{i,j−2,k} − α · in_{i,j+2,k};
            end
        end
    end
    if n ≠ T then
        swap(in, out);
    end
end
```

Algorithm 2: The `inline` algorithm.

Technology	Cray (classic)	GNU	Intel	PGI
OpenMP	yes	yes	yes	yes
OpenMP offloading	no*	no†	no	yes
OpenACC	no*	no†	no	runtime error‡
CUDA	no	no	no	yes

* CUDA symbol not found
† compiler compiled without offloading support
‡ segmentation fault

Table 3.1.: Fortran compiler support for different technologies.

Technology	Cray (`clang`)	GNU	Intel	PGI
OpenMP	yes	yes	yes	yes
OpenMP offloading	no§	no†	no	yes
OpenACC	no	no†	no	runtime error¶
CUDA (`nvcc`)	runtime error‖	yes	runtime error‖	runtime error**

† compiler compiled without offloading support
‡ segmentation fault
§ CUDA invalid source
¶ Invalid handle
‖ CUDA invalid configuration
** `ldd` could not find symbol

Table 3.2.: C++ compiler support for different technologies.

3.3. Implementation

The author implemented algorithms 1 and 2 in Fortran, C++ and Rust. The author considers himself to be a novice Fortran programmer, an advanced C++ programmer and an average Rust programmer, having about 4 weeks, 5 years and 1 year of experience in each respectively.

In addition to the sequential version in each langauge, we implemented the following versions: in Fortran, using OpenMP, OpenMP offloading, OpenACC, and CUDA; in C++, using OpenMP, OpenMP offloading, OpenACC, and CUDA; and in Rust using `rayon`, `accel`, and CUDA. Complier support for these technologies varied, which we show in tables 3.1 to 3.3. We used optimization reports from compilers to confirm that code was being vectorized and offloaded where desired, and used flamegraphs [19] and CrayPat reports to check that out code was optimized; in cases where it was not, we implemented further optimizations by hand, as described in section 3.5.

Technology	`rustc`
`rayon`	yes
`accel`	yes[††]
CUDA	no[‡‡]

[††] using Rust version `nightly-2020-05-01` for device code
[‡‡] could not compile dependencies

Table 3.3.: Rust compiler support for different technologies.

3.4. Setup

We compiled the C++ and Fortran implementations using the four available toolchains on Piz Daint: Cray, GNU, Intel and PGI. The Rust implementation was compiled using the reference Rust compiler, `rustc`[3]. CUDA C++ versions used the NVIDIA CUDA compiler for device code compilation. We used appropriate optimization flags specific to each compiler to generate optimized code. Link-time optimization was not enabled. However, Fortran and C++ code was compiled with non-associative floating-point operations enabled (`-ffast-math` or equivalent), while Rust code was not; instead, we wrote a version in Rust using a special floating-point type that only enables such optimizations locally. Section 3.3 details the optmization process we undertook. All matrices were stored in column-major order, whose elements were single-precision floating-point numbers.

When parallelizing the code, care was taken to primarily split along the z-axis. While this might seem an artificial limitation for this simple code, it is more representative of a real-world scenario, where eg. in a climate code, many more operation and calculations would be performed for each vertical slice. Additionally, all artificial synchronization barriers were put at the end of each time iteration, as that is where inter-node communication would take place in a cluster-distributed simulation. This was left out due to time constraints, as it was deemed to be more dependent on the underlying vendor-provided communication library, rather than on any language itself.

The benchmark driver was written in Rust, and linked dynamically to each of the libraries containing the implementations. The initial conditions,

$$\phi(x,y,z,0) = \phi_0(x,y,z) = \begin{cases} 1 \text{ if } 0.25 \leq x,y,z \leq 0.75 \\ 0 \text{ otherwise} \end{cases}, \tag{3.8}$$

were set by the driver. A visualization thereof can be seen in fig. 3.3. For ease of implementation, we used a hardwall boundary condition, fixing the value at the boundary to be 0. Additionally, the driver calculated the l_∞-error from the baseline for each measurement. This baseline was chosen to be the numerical solution to the problem as calculated by the sequential Fortran `laplap` version compiled using the Cray toolchain. During testing and debugging, we noticed that the error correlates strongly with the choice of

[3]For exact versions, see appendix A.

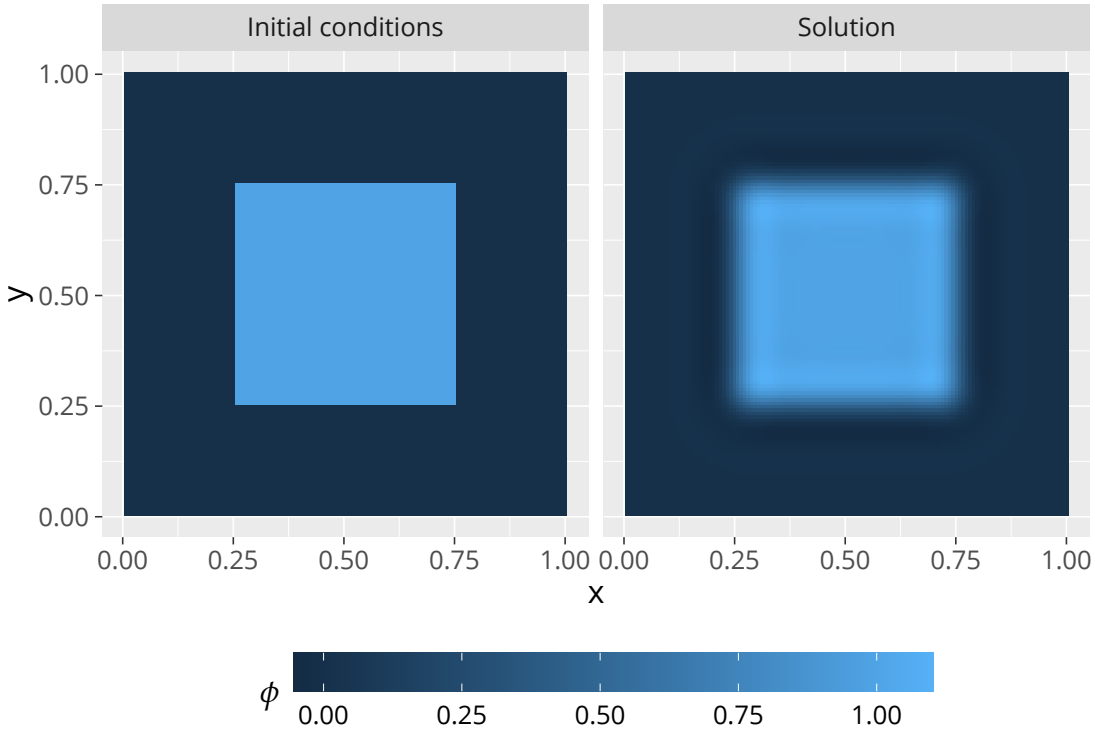

Figure 3.3.: Initial conditions and solution after 1024 iterations at $z = 0.5$.

algorithm and the number of iterations, being of the order of 1.5×10^{-7} per iteration. We therefore conjecture that the main cause of error are floating-point associativity caused by algorithmic differences. Furthermore, parallel versions exhibited a slightly higher error of about 2×10^{-7} per iteration, which is still in the precision range for single-precision floating-point arithmetic (6–9 decimal digits). We verified that solutions were reasonable in all cases using graphical comparision. In the end, we set the threshold for accepting a solution as correct to be 2.5×10^{-4} after 1024 iterations; all implementations fulfilled this criterion.

All code and measurement data are committed to to a version control system, and is available from the author upon request.

We ran the benchmark on Cray XC50 nodes of Piz Daint. Each of these consists of one Intel Xeon E5-2690v3 CPU and one Nvidia Tesla P100 GPU. α was set to 0.03125 and the number of iterations was fixed at 1024. Note, that this means, that for higher resolution, the simulation was run for a shorter amount of physical time, in order to satisfy the CFL condition. Due to time constraints, we only ran the benchmarks once per language-compiler-version combination, but are confident that our results are consistent, due to them exhibiting consistency during testing and debugging.

3.5. Results

In this section, we present our findings. Out of the 84 language-compiler-algorithm-version combinations, 52 could be successfully benchmarked, as detailed in section 3.3. This resulted in 524 measurements, as CPU-parallelized versions were run on 1, 2, 4, 8 and 12 cores, so that a scaling analysis could be done—these are detailed in appendix B. Next to a discussion of the runtime measurements, we choose to provide a subjective, developer-diary-like description of our experience in working with each language and technology. Figures 3.4, 3.6 and 3.7 show the benchmark runtimes as functions of the problem size, algorithm, implementation language, compiler and technology used. Immediately, we see that all these factors affect the runtime, including the choice of algorithm.

For the sequential case, as shown in fig. 3.4, we see that there is little difference between compilers for the majority of cases. Seemingly, only the Fortran PGI `laplap` version, the C++ PGI `inline`, and the Fortran GNU versions perform significantly worse. Our original Rust version, `seq`, also runs slower—this is unsurprising, as this direct translation from Fortran does an index bounds check on each array element access. The other Rust versions all perform comparatively fast; this includes using unchecked array element access (`seq_unchecked`), explicitly using fused-multiply-add intrinsics (`seq_fma`), non-associative floating-point operations (`seq_fast`), and iterators (`seq_zip`); this can be seen in fig. 3.5 in more detail. Seeing this, we concluded that these approaches were all similarly performant, and chose not to implement the `fma` and `fast` versions in parallelized or offloaded code.

In terms of implementation effort, programming in Fortran was the most simple, due to language support for multi-dimensional arrays; however, error handling, such as asserting preconditions, was significantly more difficult, due to lack of a standard library. In the C++ implementation, we chose to write a simple array wrapper class; this was a simple enough task. Some compilers required annotating loops with directives, so that the code would be appropriately vectorized. Our Rust code ended up being most complicated. We used the `ndarray` crate for array support, deciding against `nalgebra` due to its lack of support for arrays of more than 2 dimensions. Translating simple loops from Fortran resulted in bad performance, due to bound-checking being performed on each access. This was to be expected. Using unchecked accesses required using **`unsafe`** blocks, which we found to be bad practice. The iterator version (`seq_zip`) was simliarly fast; however required naming each summand in eqs. (3.6a), (3.6b) and (3.7) separately, which we found to not be very ergonomical. Additionally, in the `inline` version, the `ndarray` crate had to be patched in order support higher arity of the `zip!` macro. This also had to be done for the respective CPU versions.

In the parallelized CPU versions, we see even more differences between compilers. The Intel C++ compiler produces the fastest running code. Surprisingly, the Rust `laplap` code using `rayon` parallel iterators (`par_axis_zip`) comes in second place, only about 20% slower than the Intel version. Other compiler-language-algorithm combinations are significantly slower, by between 3.6x to 16.5x compared to the Rust version. Finally, the Intel Fortran compiler performs the worst, taking about 29 times as long to compute the solution, as the Intel C++ version. Due to time contraints, we were not able to fully

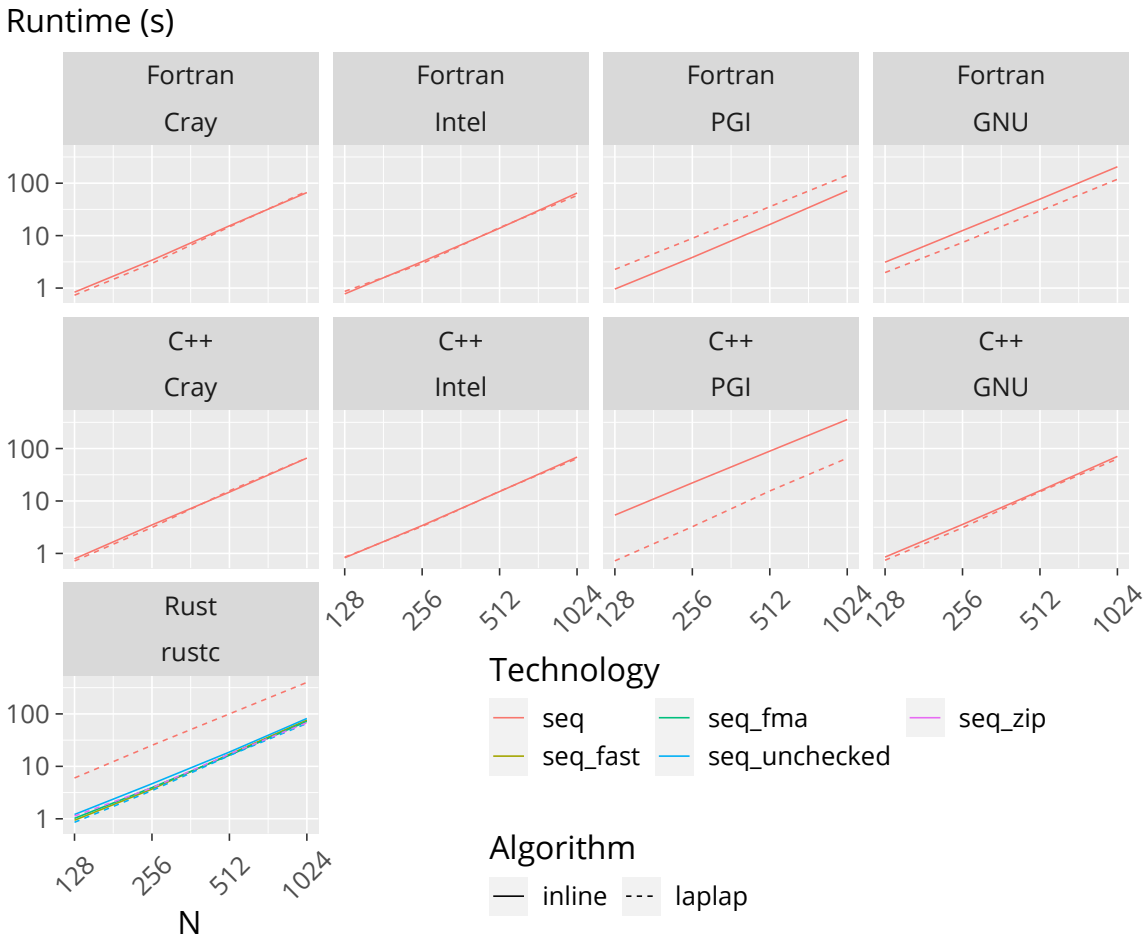

Figure 3.4.: Scaling by size of the sequential versions.

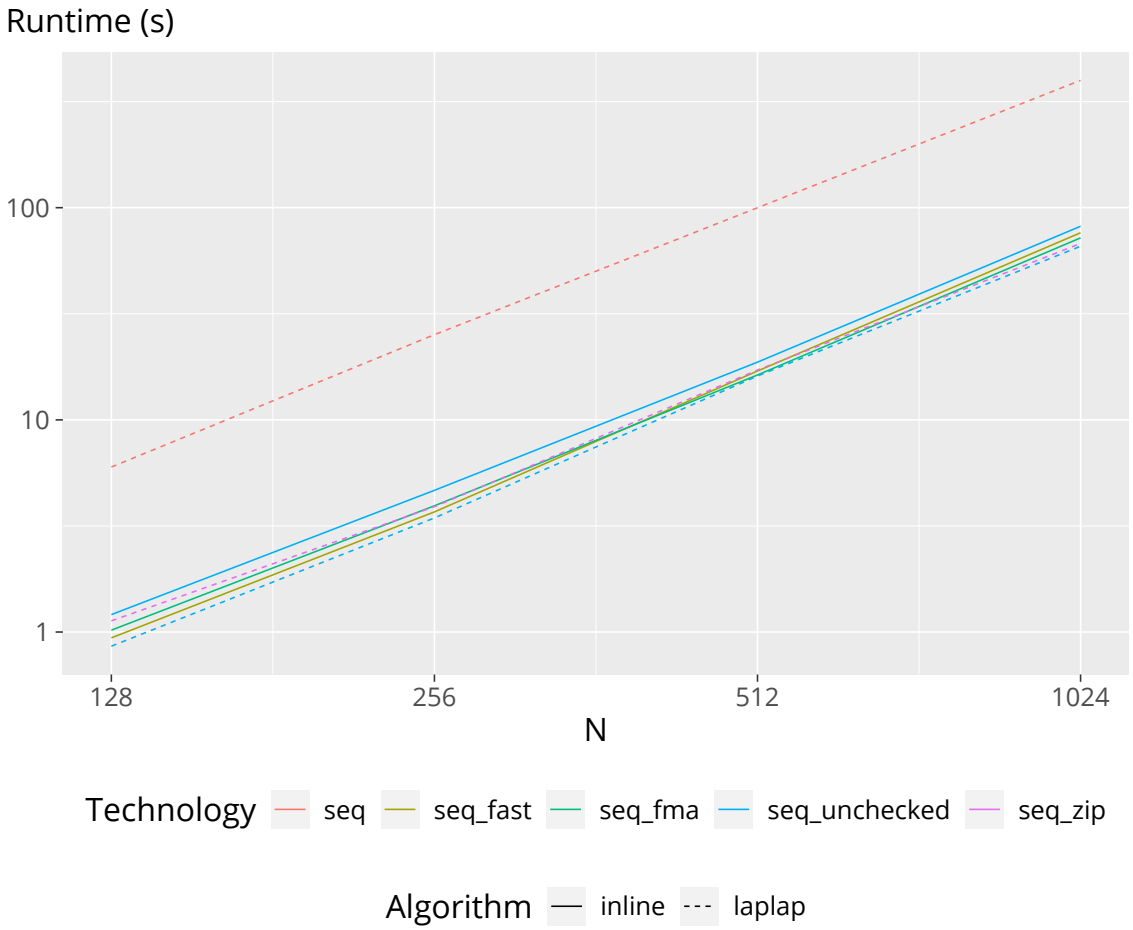

Figure 3.5.: Scaling by problem size of the sequential Rust versions (detailed).

analyze the issue, and are not sure how this discrepancy comes to be. We conjecture that given more time, we could significantly improve the time-to-solution of these versions, and bring them inline and closer to the others.

Implementation in Fortran and C++ was straightforward, being limited to the addition of OpenMP directives, barring minor non-adherence to the OpenMP standard in case of the GNU compiler; this was due to the version of the GNU toolchain that was available on the computing cluster—the issue has been resolved in GCC 9. On the Rust side, parallelization of code was similarly simple, by changing from a loop over all z-levels to using a parallel iterator over the z-axis.

All of our OpenMP implementations are parallelized only in the z-dimension; however, due to an error, our initial Rust implementation (labeled `par_zip`), may parallelize over all dimensions; this issue was fixed in the `par_axis_zip` version. This artificial limitation is more representative of how a real climate code might be structured—nevertheless, we decided to include the `par_zip` version in our results, because its scaling behaviour resembles that of the C++ versions compiled using the Intel toolchain. We hypothesize that the Intel C++ compiler might be optimizing our code more than we would want it to, as discusssed in section 3.4.

As mentioned in section 3.3, and indicated by tables 3.1 and 3.2, we implemented offloading to the GPU in Fortran and C++ using OpenMP, OpenACC and CUDA. Sadly, it turned out that compiler support for these technologies was highly lacking. The Intel compiler used did not support any offloading to GPUs, only to Intel Xeon Phi coprocessors. The Cray-provided GNU toolchain was not compiled with offloading support; at the time of benchmarking, installation of a offloading-capable GNU compiler using `spack` resulted in errors during its building, as did the LLVM toolchain. The newer Cray-`clang` toolchain does not yet support OpenACC; while using OpenMP `target` directives with it resulted in seemingly invalid PTX code. The Cray-classic toolchain used for Fortran generated OpenMP-offloaded code that failed at runtime. CUDA device code generated by the NVIDIA `nvcc` compiler resulted in runtime errors in all cases except when linking to GNU-generated host code. In the end, out of the 17 officially supported configurations with Fortran and C++, we could only successfully benchmark 4.

While using Rust to generate GPU device code, we could not get our pure Rust implementation to work; it failed to create correct shared libraries for some dependencies; generating Rust libraries (`rlib`s) by hand worked, was however not compatible with our benchmark driver. Therefore, we could only benchmark the Rust version implemented using `accel`, which hardcodes the device compiler to be the Rust nightly version from 2020-05-01.

We see that in the case of GPU offloaded code, the Rust version is about 2x slower than some of the OpenMP offloaded versions. The CUDA Fortran and C++ versions are also slower than the OpenMP versions; we suspect this is due to us not using shared memory to optimize GPU memory accesses in our CUDA codes. This was not done on the one hand due to time constraints, on the other hand, to offer a fair comparison with Rust, which supports neither constant nor shared CUDA memory spaces.

The OpenMP-offloaded versions necessitated only small adjustments from multi-threaded OpenMP implementations. The OpenACC versions were in turn similar to

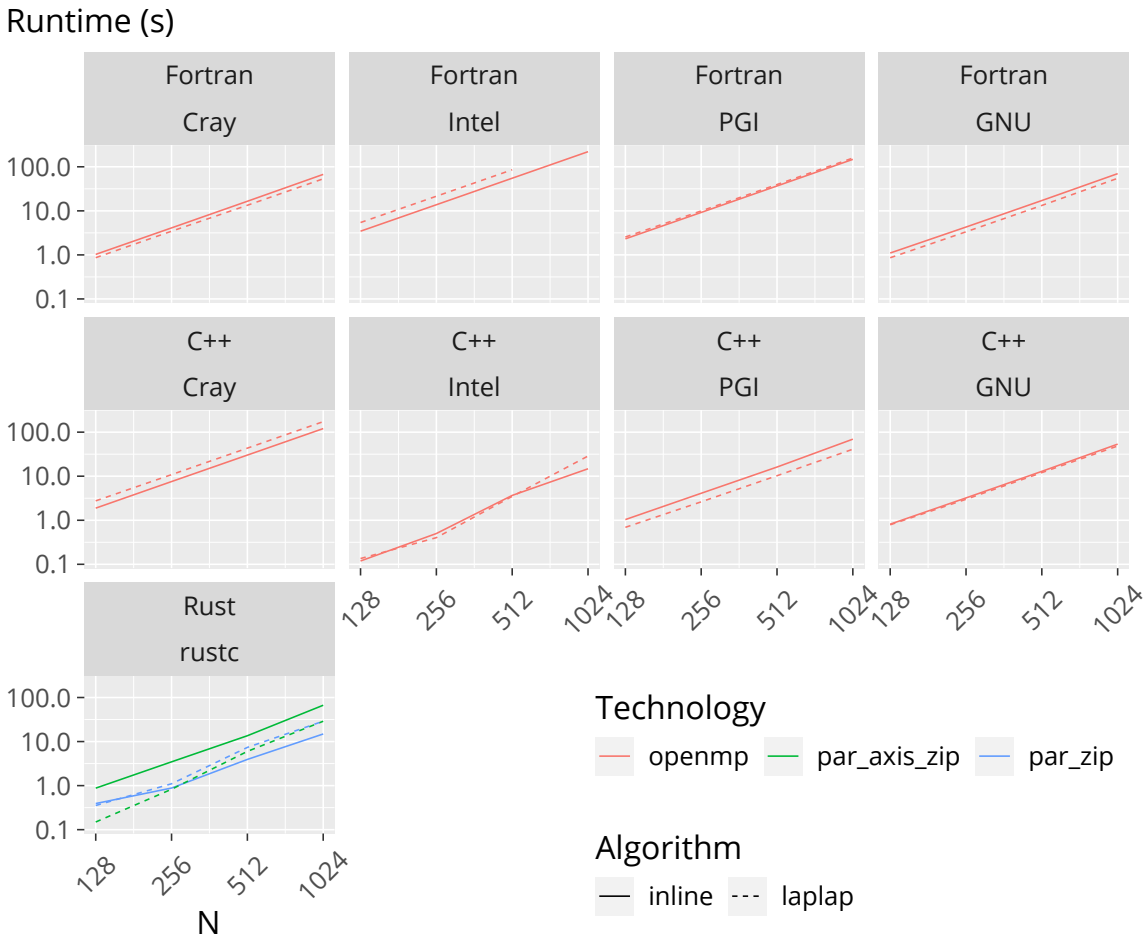

Figure 3.6.: Scaling by problem size of the parallelized versions (run on 12 cores).

these, due to the relatively small difference in the standards from end-user perspective. However, we were very dissappointed that none of the OpenACC versions could be successfully benchmarked. Furthermore, each toolchain implements such offloading in a different manner: the Cray-classic toolchain seems to create a shared library with only host code, loading and just-in-time-compiling device code at runtime; meanwhile, the PGI toolchain directly embeds the PTX and fatbin artefacts in the shared library. In all cases, we find the documentation to be very lacking in terms of instruction how to debug obscure errors seemingly related to the launching of OpenACC and OpenMP kernels; we are similarly dissapponted that these errors are not raised at compile-time. We had not expected that linking `nvcc`-compiled CUDA code with host code would be so cumbersome and error-ridden. However, we were pleasantly surprised at the ergonomics of CUDA Fortran—especially the use of custom attributes to designate host and device variables, and the implicit copying using the assignment operator between them.

Knowing that Rust support for GPU programming is in only experimentally supported, we did not have high expectations. We were mildly surprised in how `accel` provided a similar experience to CUDA C++; launching kernels from `accel` is a bit more explicit, however, as it required manual device selection and context creation. Due to Rust's safety guarantees, device code is inherently **`unsafe`**; the pointer arithmetic is similar to CUDA C++, but does not feel very *Rust-ic*. Our pure Rust implementation failed to fully compile (see table 3.3)—in our opinion this demonstrates the experimental state of GPU programming in Rust.

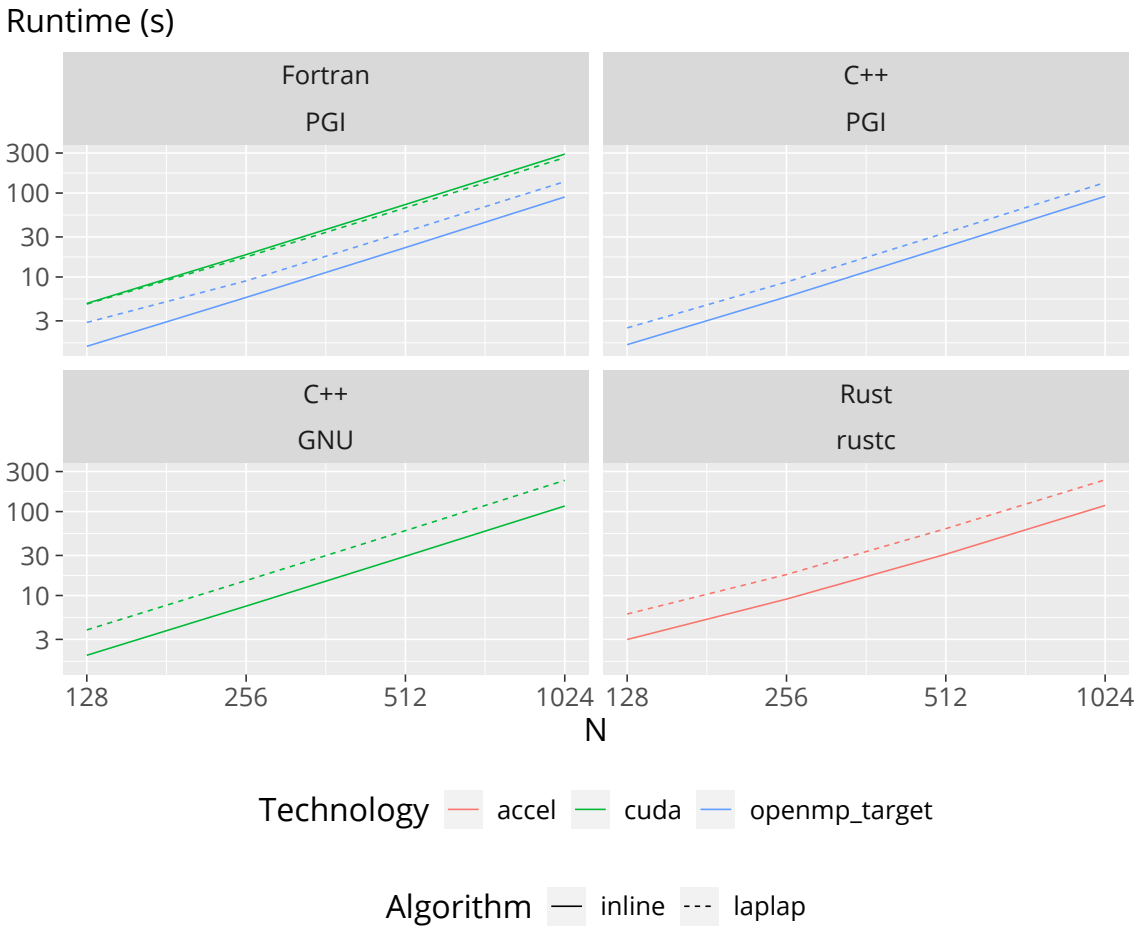

Figure 3.7.: Scaling by problem size of the offloaded versions.

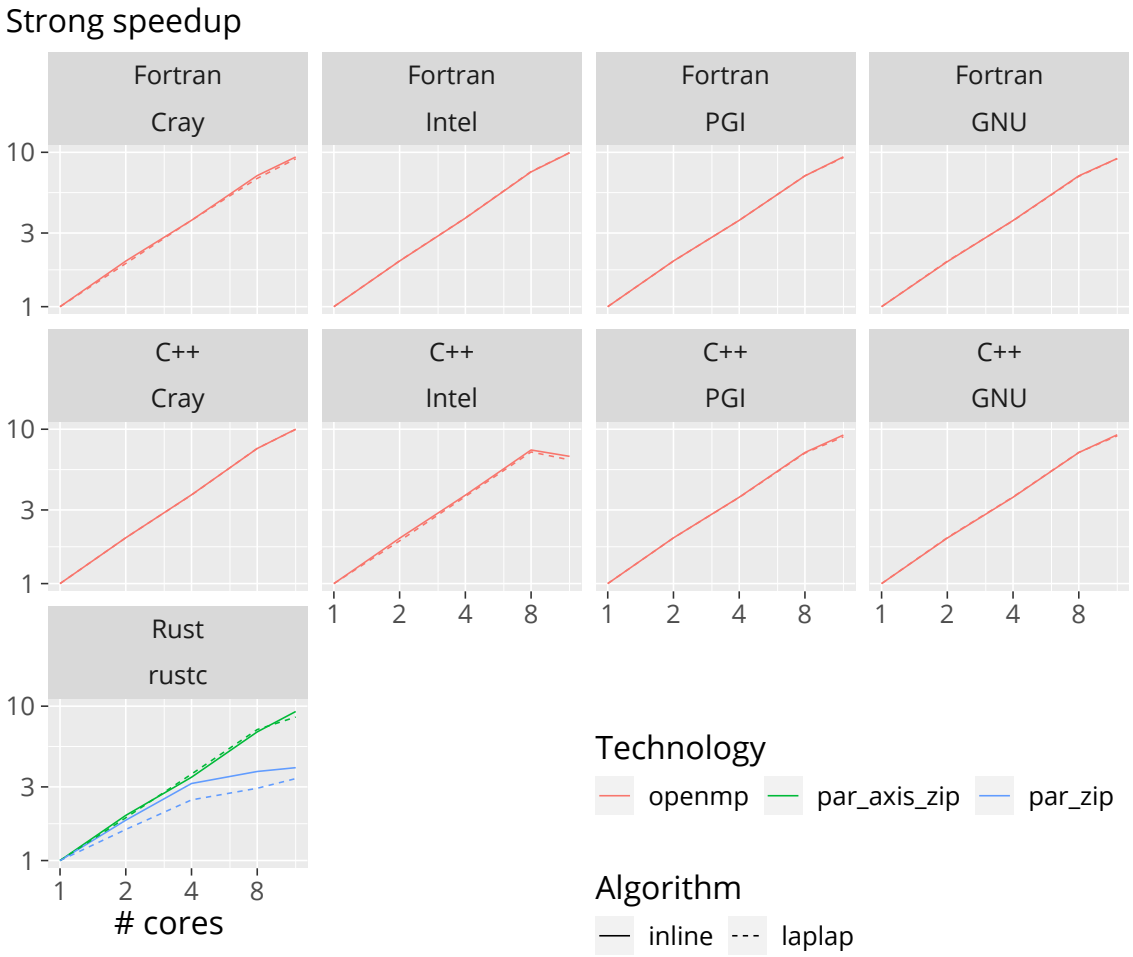

Figure 3.8.: Strong scaling of the parallelized versions for $N = 128$.

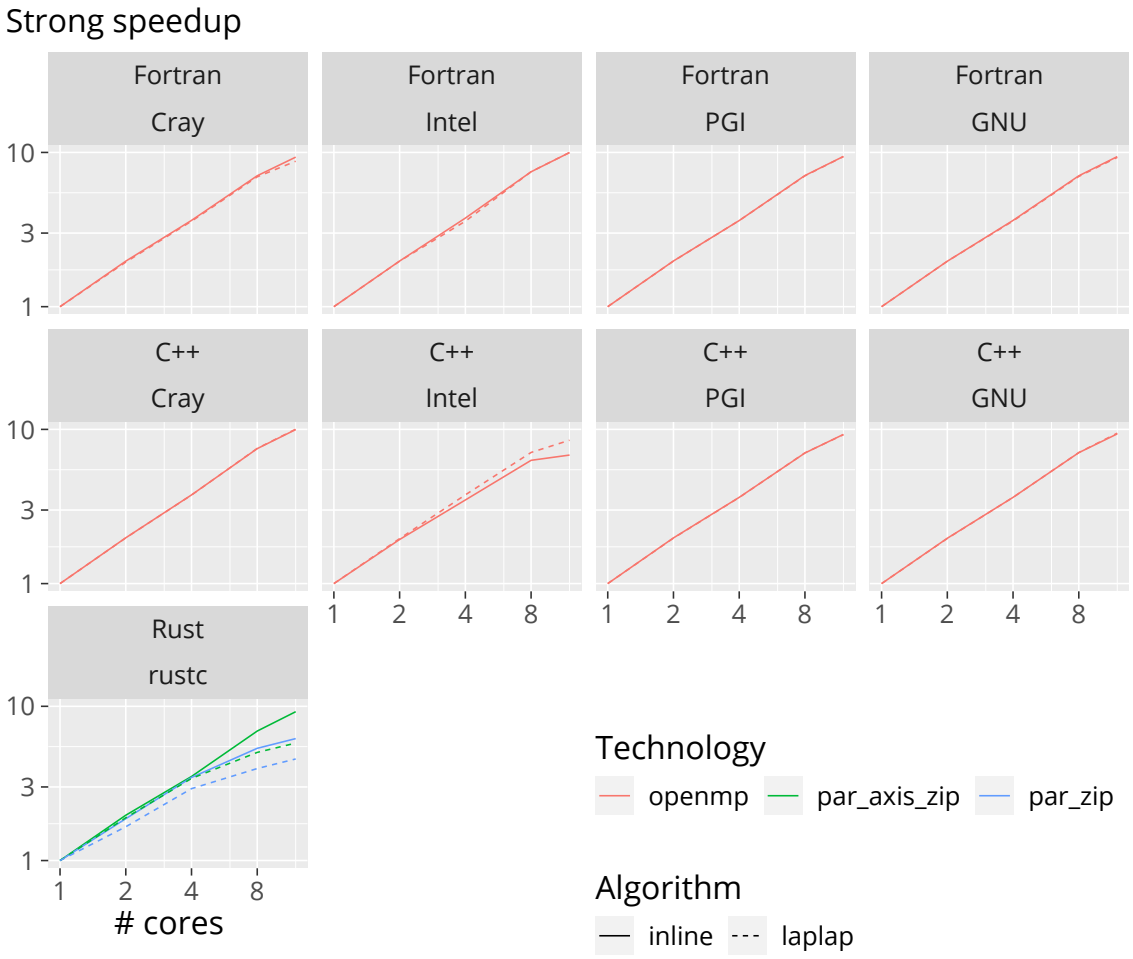

Figure 3.9.: Strong scaling of the parallelized versions for $N = 256$.

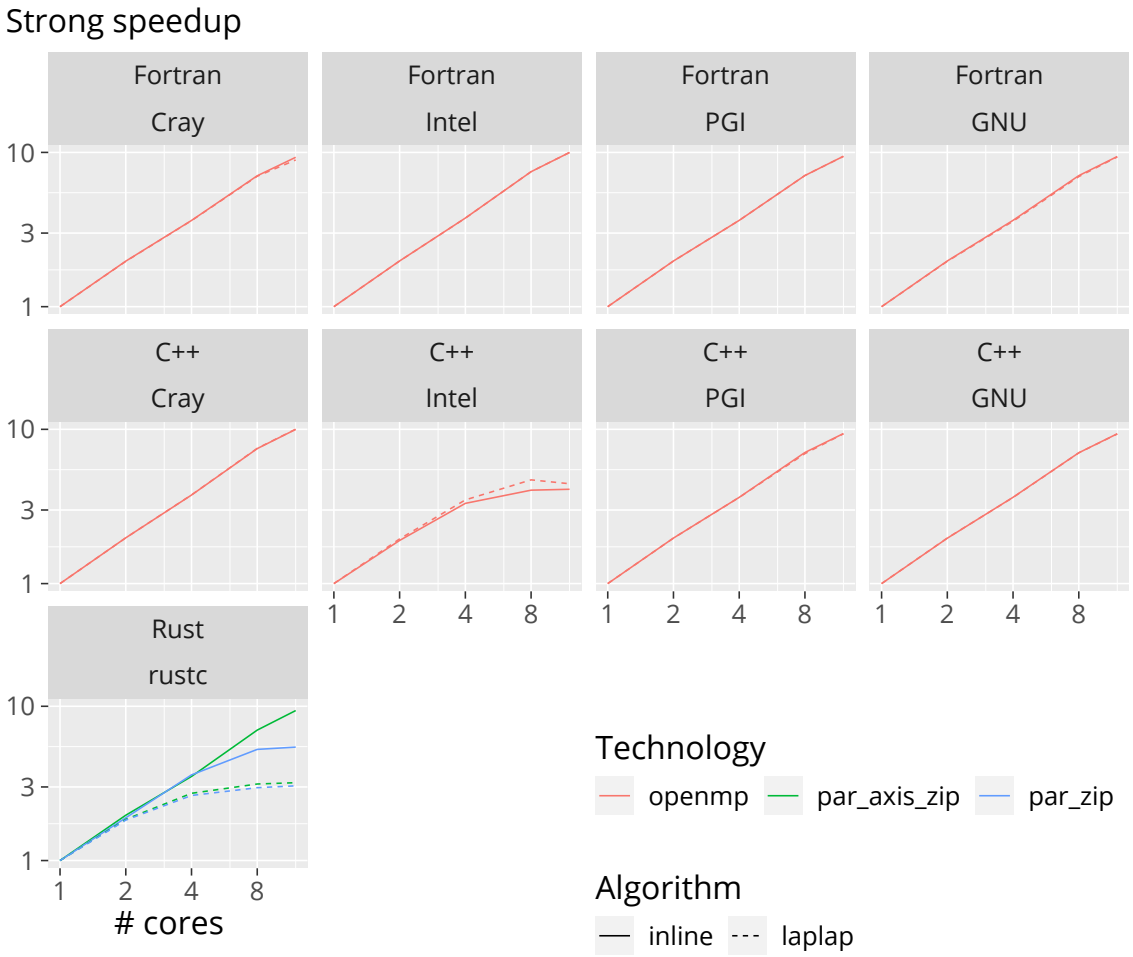

Figure 3.10.: Strong scaling of the parallelized versions for $N = 512$.

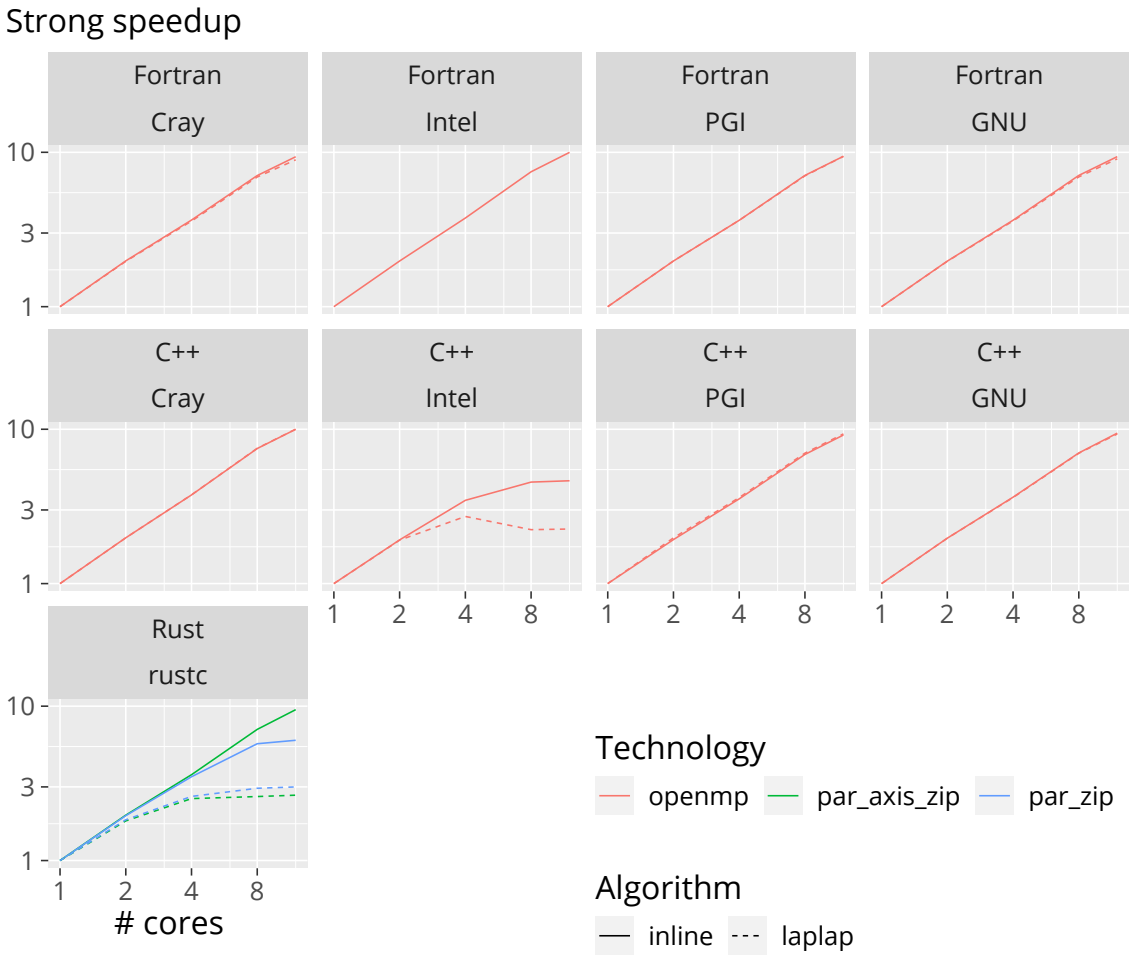

Figure 3.11.: Strong scaling of the parallelized versions for $N = 1024$.

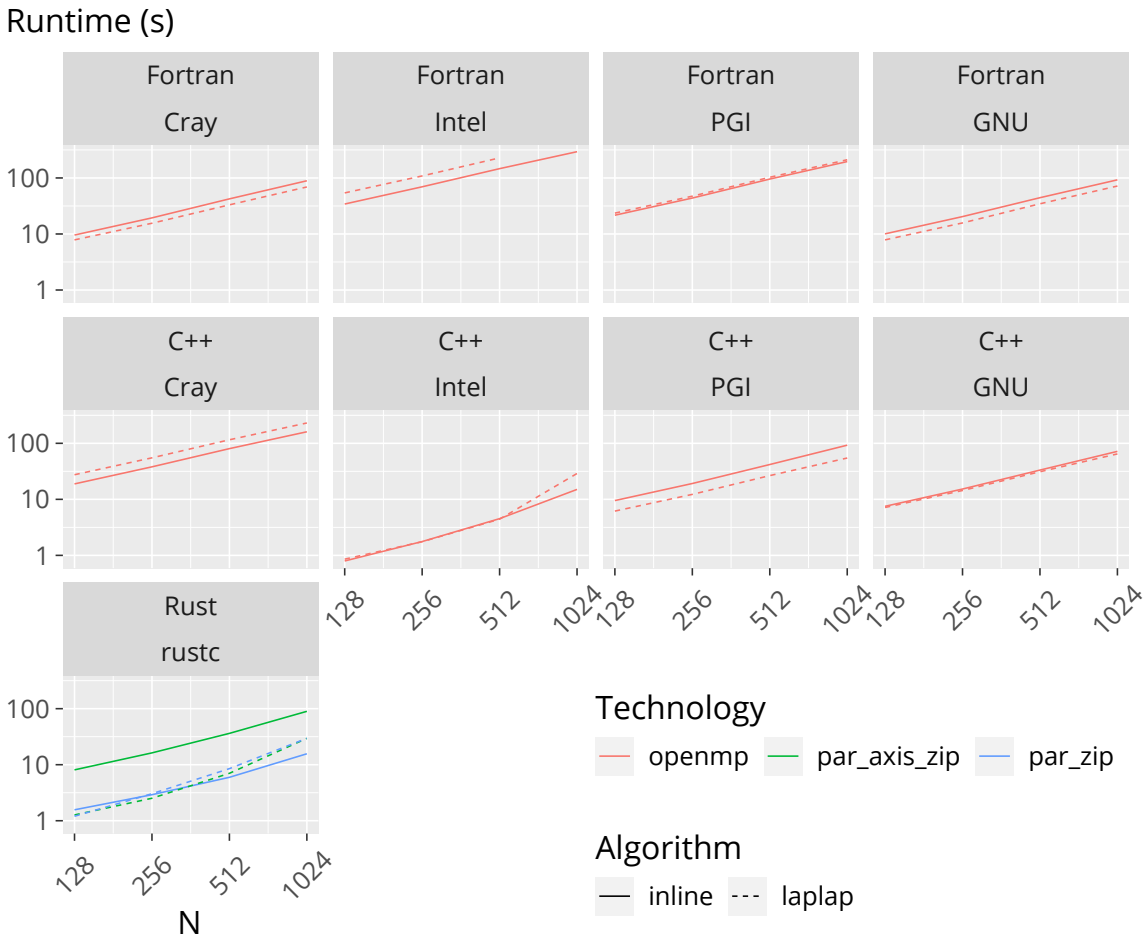

Figure 3.12.: Weak scaling of the parallelized versions.

Chapter 4

Conclusion

> *I don't know what the language of the year 2000 will look like, but I know it will be called Fortran.*
> — C.A.R. Hoare [15]

In this thesis, we have shown that Rust is a viable language choice for HPC applications, both in terms of its features, which match or surpass those of Fortran and C++, and in terms of its performance. We find the extra safety provided by the language to be very useful, and conjecture that these additional safety guarantees might enable more aggressive optimizations, and therefore contribute positively to its runtime performance.

However, we find that Rust support for GPU programming is not well developed. We have trouble envisioning how this will change in the future—while of general interest to the Rust community, it is our opinion that GPGPU programming will be difficult to achieve while maintaining Rust's safety guarantees. Nevertheless, Rust can be used for host-code, and we see no problems in calling GPU kernels written in other languages from Rust.

Next to clearer error messages, we see potential in Rust's hygienic macro systems to deliver safe and performant code. We could imagine that an OpenMP-like, directive-driven, parallelization model could be implemented in the current Rust version, using procedural macros or compiler plugins. However, the scientific community seems to currently not be aware of the riches that Rust has to offer, and therefore development of features geared towards scientific computing in Rust is rather slow. We encourage the community to try programming in Rust, show interest, and contribute to the Rust ecosystem and project, and hope that this thesis will be a stepping stone on that path.

Appendix A

Reproducibility

> *At the end of the day, reproducibility is what separates science from superstition.*
> — Sergey Fomel [10]

Here, we provide information on the compilers used for the benchmark. Compiler flags can be found in the project source code. Environment variables relevant to using Rust on the Piz Daint system can be found in listing 5.

Code is available from the author upon request.

```
> CC --version
Cray clang version 9.0.2
 ↪  (2a3a7003aaa5b93e2070bde59a5ee6b9682b67d7) (based on LLVM
 ↪  9.0.0svn)
Target: x86_64-unknown-linux-gnu
Thread model: posix
InstalledDir: /opt/cray/pe/cce/9.0.2/cce-clang/x86_64/bin
```

Listing 6: Output of `CC --version`.

```
> ftn --version
Cray Fortran : Version 9.0.2
```

Listing 7: Output of `ftn --version`.

```
> g++ --version
g++ (GCC) 8.3.0 20190222 (Cray Inc.)
Copyright (C) 2018 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.
 ↪  There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A
 ↪  PARTICULAR PURPOSE.
```

Listing 8: Output of `g++ --version`.

```
> gfortran --version
GNU Fortran (GCC) 8.3.0 20190222 (Cray Inc.)
Copyright (C) 2018 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.
 ↪  There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A
 ↪  PARTICULAR PURPOSE.
```

Listing 9: Output of `gfortran --version`.

```
> icpc --version
icpc (ICC) 19.0.1.144 20181018
Copyright (C) 1985-2018 Intel Corporation.  All rights
 ↪  reserved.
```

Listing 10: Output of `icpc --version`.

```
> ifort --version
ifort (IFORT) 19.0.1.144 20181018
Copyright (C) 1985-2018 Intel Corporation.  All rights
 ↪  reserved.
```

Listing 11: Output of `ifort --version`.

```
> pgc++ --version

pgc++ 19.7-0 LLVM 64-bit target on x86-64 Linux -tp haswell
PGI Compilers and Tools
Copyright (c) 2019, NVIDIA CORPORATION.  All rights reserved.
```

Listing 12: Output of `pgc++ --version`.

```
> pgfortran --version

pgfortran 19.7-0 LLVM 64-bit target on x86-64 Linux -tp
 ↪  haswell
PGI Compilers and Tools
Copyright (c) 2019, NVIDIA CORPORATION.  All rights reserved.
```

Listing 13: Output of `pgfortran --version`.

```
> nvcc --version
nvcc: NVIDIA (R) Cuda compiler driver
Copyright (c) 2005-2019 NVIDIA Corporation
Built on Fri_Feb__8_19:08:17_PST_2019
Cuda compilation tools, release 10.1, V10.1.105
```

Listing 14: Output of `nvcc --version`.

```
> rustc --version
rustc 1.47.0-nightly (6e87bacd3 2020-07-31)
```

Listing 15: Output of `rustc --version`.

Appendix B

Data

For ease of lookup, we provide raw benchmark results here— a machine-readable version of these is also available in the project repository. All measurements were done with $N = n_x = n_y$, $n_z = 64$, $\alpha = 0.3125$ and $T = 1024$ iterations. Runtimes are given in seconds, and errors are measured in the maximum norm l_∞, for which the baseline is the sequential Fortran version compiled using the Cray toolchain, as explained in section 3.4.

Type	Language	Compiler	Technology	Algorithm	N	# cpus	Runtime / s	Error
sequential	Fortran	Cray	`seq`	`inline`	128	1	0.834 320 612	0.000 166 893
sequential	Fortran	Cray	`seq`	`laplap`	128	1	0.730 569 095	0.000 000 000
sequential	Fortran	Intel	`seq`	`inline`	128	1	0.773 508 563	0.000 166 714
sequential	Fortran	Intel	`seq`	`laplap`	128	1	0.864 097 189	0.000 000 715
sequential	Fortran	PGI	`seq`	`inline`	128	1	0.958 177 879	0.000 167 370
sequential	Fortran	PGI	`seq`	`laplap`	128	1	2.271 993 505	0.000 000 596
sequential	Fortran	GNU	`seq`	`laplap`	128	1	1.976 091 950	0.000 000 954
sequential	Fortran	GNU	`seq`	`inline`	128	1	3.111 656 197	0.000 166 893
sequential	C++	Cray	`seq`	`inline`	128	1	0.788 973 350	0.000 167 191
sequential	C++	Cray	`seq`	`laplap`	128	1	0.718 197 252	0.000 000 596
sequential	C++	Intel	`seq`	`inline`	128	1	0.826 223 002	0.000 166 714
sequential	C++	Intel	`seq`	`laplap`	128	1	0.847 260 526	0.000 000 596
sequential	C++	PGI	`seq`	`laplap`	128	1	0.721 972 290	0.000 000 596
sequential	C++	PGI	`seq`	`inline`	128	1	5.381 884 588	0.000 167 370
sequential	C++	GNU	`seq`	`inline`	128	1	0.852 053 869	0.000 167 012
sequential	C++	GNU	`seq`	`laplap`	128	1	0.742 221 785	0.000 000 954
sequential	Rust	rustc	`seq`	`laplap`	128	1	5.994 463 596	0.000 000 596
sequential	Rust	rustc	`seq_fast`	`inline`	128	1	0.938 428 012	0.000 167 489
sequential	Rust	rustc	`seq_fma`	`inline`	128	1	1.020 469 138	0.000 167 131
sequential	Rust	rustc	`seq_unchecked`	`laplap`	128	1	0.857 512 664	0.000 000 596
sequential	Rust	rustc	`seq_unchecked`	`inline`	128	1	1.207 844 976	0.000 167 370
sequential	Rust	rustc	`seq_zip`	`laplap`	128	1	1.129 125 234	0.000 000 596
CPU	Fortran	Cray	`openmp`	`laplap`	128	1	7.847 057 922	0.000 000 000
CPU	Fortran	Cray	`openmp`	`inline`	128	1	9.574 371 178	0.000 167 012
CPU	Fortran	Cray	`openmp`	`laplap`	128	2	4.119 796 232	0.000 000 000
CPU	Fortran	Cray	`openmp`	`inline`	128	2	4.850 322 656	0.000 167 012
CPU	Fortran	Cray	`openmp`	`laplap`	128	4	2.162 870 340	0.000 000 000
CPU	Fortran	Cray	`openmp`	`inline`	128	4	2.640 960 400	0.000 167 012

Type	Language	Compiler	Technology	Algorithm	N	# cpus	Runtime / s	Error
CPU	Fortran	Cray	openmp	laplap	128	8	1.157 931 952	0.000 000 000
CPU	Fortran	Cray	openmp	inline	128	8	1.355 540 728	0.000 167 012
CPU	Fortran	Cray	openmp	inline	128	12	1.026 727 944	0.000 167 012
CPU	Fortran	Cray	openmp	laplap	128	12	0.864 520 219	0.000 000 000
CPU	Fortran	Intel	openmp	inline	128	1	34.261 413 820	0.000 166 714
CPU	Fortran	Intel	openmp	laplap	128	1	54.031 034 883	0.000 000 596
CPU	Fortran	Intel	openmp	inline	128	2	17.341 185 050	0.000 166 714
CPU	Fortran	Intel	openmp	laplap	128	2	27.281 257 785	0.000 000 596
CPU	Fortran	Intel	openmp	inline	128	4	9.148 581 223	0.000 166 714
CPU	Fortran	Intel	openmp	laplap	128	4	14.392 337 653	0.000 000 596
CPU	Fortran	Intel	openmp	inline	128	8	4.588 523 385	0.000 166 714
CPU	Fortran	Intel	openmp	laplap	128	8	7.202 359 007	0.000 000 596
CPU	Fortran	Intel	openmp	inline	128	12	3.453 800 380	0.000 166 714
CPU	Fortran	Intel	openmp	laplap	128	12	5.431 670 366	0.000 000 596
CPU	Fortran	PGI	openmp	inline	128	1	21.630 207 977	0.000 167 370
CPU	Fortran	PGI	openmp	laplap	128	1	23.557 755 916	0.000 000 596
CPU	Fortran	PGI	openmp	inline	128	2	10.945 492 106	0.000 167 370
CPU	Fortran	PGI	openmp	laplap	128	2	11.934 176 901	0.000 000 596
CPU	Fortran	PGI	openmp	laplap	128	4	6.495 220 855	0.000 000 596
CPU	Fortran	PGI	openmp	inline	128	4	5.969 200 886	0.000 167 370
CPU	Fortran	PGI	openmp	laplap	128	8	3.342 070 829	0.000 000 596
CPU	Fortran	PGI	openmp	inline	128	8	3.074 289 473	0.000 167 370
CPU	Fortran	PGI	openmp	laplap	128	12	2.546 244 565	0.000 000 596
CPU	Fortran	PGI	openmp	inline	128	12	2.313 911 807	0.000 167 370
CPU	Fortran	GNU	openmp	laplap	128	1	7.843 317 527	0.000 000 954
CPU	Fortran	GNU	openmp	inline	128	1	10.059 440 813	0.000 167 012
CPU	Fortran	GNU	openmp	laplap	128	2	3.981 160 550	0.000 000 954
CPU	Fortran	GNU	openmp	inline	128	2	5.140 539 120	0.000 167 012
CPU	Fortran	GNU	openmp	inline	128	4	2.783 790 742	0.000 167 012

Type	Language	Compiler	Technology	Algorithm	N	# cpus	Runtime / s	Error
CPU	Fortran	GNU	openmp	laplap	128	4	2.177 717 336	0.000 000 954
CPU	Fortran	GNU	openmp	inline	128	8	1.429 967 382	0.000 167 012
CPU	Fortran	GNU	openmp	laplap	128	8	1.122 432 254	0.000 000 954
CPU	Fortran	GNU	openmp	laplap	128	12	0.862 317 996	0.000 000 954
CPU	Fortran	GNU	openmp	inline	128	12	1.103 193 995	0.000 167 012
CPU	C++	Cray	openmp	inline	128	1	18.854 872 028	0.000 166 476
CPU	C++	Cray	openmp	laplap	128	1	27.484 511 349	0.000 000 596
CPU	C++	Cray	openmp	inline	128	2	9.517 673 656	0.000 166 476
CPU	C++	Cray	openmp	laplap	128	2	13.868 281 024	0.000 000 596
CPU	C++	Cray	openmp	inline	128	4	5.015 302 430	0.000 166 476
CPU	C++	Cray	openmp	laplap	128	4	7.328 090 536	0.000 000 596
CPU	C++	Cray	openmp	inline	128	8	2.512 877 222	0.000 166 476
CPU	C++	Cray	openmp	laplap	128	8	3.657 866 107	0.000 000 596
CPU	C++	Cray	openmp	laplap	128	12	2.755 623 380	0.000 000 596
CPU	C++	Cray	openmp	inline	128	12	1.883 594 063	0.000 166 476
CPU	C++	Intel	openmp	laplap	128	1	0.855 416 114	0.000 000 596
CPU	C++	Intel	openmp	inline	128	1	0.794 291 160	0.000 166 714
CPU	C++	Intel	openmp	inline	128	2	0.403 511 496	0.000 166 714
CPU	C++	Intel	openmp	laplap	128	2	0.452 124 722	0.000 000 715
CPU	C++	Intel	openmp	laplap	128	4	0.233 304 034	0.000 000 715
CPU	C++	Intel	openmp	inline	128	4	0.212 341 539	0.000 166 714
CPU	C++	Intel	openmp	inline	128	8	0.108 264 332	0.000 166 714
CPU	C++	Intel	openmp	laplap	128	8	0.120 062 291	0.000 000 715
CPU	C++	Intel	openmp	inline	128	12	0.119 007 820	0.000 166 714
CPU	C++	Intel	openmp	laplap	128	12	0.135 112 840	0.000 000 715
CPU	C++	PGI	openmp	laplap	128	1	6.172 303 816	0.000 000 596
CPU	C++	PGI	openmp	inline	128	1	9.504 483 977	0.000 167 370
CPU	C++	PGI	openmp	laplap	128	2	3.122 189 964	0.000 000 596
CPU	C++	PGI	openmp	inline	128	2	4.806 355 971	0.000 167 370

Type	Language	Compiler	Technology	Algorithm	N	# cpus	Runtime / s	Error
CPU	C++	PGI	`openmp`	`inline`	128	4	2.620 270 423	0.000 167 370
CPU	C++	PGI	`openmp`	`laplap`	128	4	1.706 457 829	0.000 000 596
CPU	C++	PGI	`openmp`	`laplap`	128	8	0.881 019 888	0.000 000 596
CPU	C++	PGI	`openmp`	`inline`	128	8	1.344 258 250	0.000 167 370
CPU	C++	PGI	`openmp`	`laplap`	128	12	0.690 600 415	0.000 000 596
CPU	C++	PGI	`openmp`	`inline`	128	12	1.036 689 164	0.000 167 370
CPU	C++	GNU	`openmp`	`inline`	128	1	7.509 525 729	0.000 167 012
CPU	C++	GNU	`openmp`	`laplap`	128	1	7.156 424 133	0.000 000 715
CPU	C++	GNU	`openmp`	`laplap`	128	2	3.604 634 325	0.000 000 715
CPU	C++	GNU	`openmp`	`inline`	128	2	3.806 000 910	0.000 167 012
CPU	C++	GNU	`openmp`	`laplap`	128	4	1.964 631 146	0.000 000 715
CPU	C++	GNU	`openmp`	`inline`	128	4	2.071 536 171	0.000 167 012
CPU	C++	GNU	`openmp`	`inline`	128	8	1.063 427 249	0.000 167 012
CPU	C++	GNU	`openmp`	`laplap`	128	8	1.010 685 044	0.000 000 715
CPU	C++	GNU	`openmp`	`inline`	128	12	0.815 336 882	0.000 167 012
CPU	C++	GNU	`openmp`	`laplap`	128	12	0.789 675 003	0.000 000 715
CPU	Rust	rustc	`par_axis_zip`	`laplap`	128	1	1.268 323 077	0.000 000 596
CPU	Rust	rustc	`par_axis_zip`	`inline`	128	1	8.085 041 451	0.000 167 370
CPU	Rust	rustc	`par_axis_zip`	`laplap`	128	2	0.670 306 545	0.000 000 596
CPU	Rust	rustc	`par_axis_zip`	`inline`	128	2	4.122 219 600	0.000 167 370
CPU	Rust	rustc	`par_axis_zip`	`laplap`	128	4	0.349 295 412	0.000 000 596
CPU	Rust	rustc	`par_axis_zip`	`inline`	128	4	2.333 304 155	0.000 167 370
CPU	Rust	rustc	`par_axis_zip`	`laplap`	128	8	0.179 801 593	0.000 000 596
CPU	Rust	rustc	`par_axis_zip`	`inline`	128	8	1.186 473 747	0.000 167 370
CPU	Rust	rustc	`par_axis_zip`	`laplap`	128	12	0.149 148 993	0.000 000 596
CPU	Rust	rustc	`par_axis_zip`	`inline`	128	12	0.875 683 578	0.000 167 370
CPU	Rust	rustc	`par_zip`	`laplap`	128	1	1.200 710 804	0.000 000 596
CPU	Rust	rustc	`par_zip`	`inline`	128	1	1.566 041 230	0.000 167 370
CPU	Rust	rustc	`par_zip`	`laplap`	128	2	0.755 483 126	0.000 000 596

Type	Language	Compiler	Technology	Algorithm	N	# cpus	Runtime / s	Error
CPU	Rust	rustc	`par_zip`	`inline`	128	2	0.857 979 483	0.000 167 370
CPU	Rust	rustc	`par_zip`	`laplap`	128	4	0.485 846 051	0.000 000 596
CPU	Rust	rustc	`par_zip`	`inline`	128	4	0.496 188 119	0.000 167 370
CPU	Rust	rustc	`par_zip`	`laplap`	128	8	0.408 654 278	0.000 000 596
CPU	Rust	rustc	`par_zip`	`inline`	128	8	0.414 851 546	0.000 167 370
CPU	Rust	rustc	`par_zip`	`laplap`	128	12	0.353 667 402	0.000 000 596
CPU	Rust	rustc	`par_zip`	`inline`	128	12	0.391 985 648	0.000 167 370
GPU	Fortran	PGI	`cuda`	`inline`	128	12	4.841 446 474	0.000 167 370
GPU	Fortran	PGI	`cuda`	`laplap`	128	12	4.757 428 835	0.000 000 596
GPU	Fortran	PGI	`openmp_target`	`inline`	128	12	1.491 191 697	0.000 167 370
GPU	Fortran	PGI	`openmp_target`	`laplap`	128	12	2.870 838 587	0.000 000 596
GPU	C++	PGI	`openmp_target`	`laplap`	128	12	2.475 156 022	0.000 000 596
GPU	C++	PGI	`openmp_target`	`inline`	128	12	1.561 888 769	0.000 167 370
GPU	C++	GNU	`cuda`	`inline`	128	12	1.939 123 306	0.000 167 191
GPU	C++	GNU	`cuda`	`laplap`	128	12	3.887 717 648	0.000 000 596
GPU	Rust	rustc	`accel`	`inline`	128	12	2.990 492 881	0.000 167 370
GPU	Rust	rustc	`accel`	`laplap`	128	12	6.014 267 068	0.000 000 596
sequential	Fortran	Cray	`seq`	`laplap`	256	1	2.946 258 008	0.000 000 000
sequential	Fortran	Cray	`seq`	`inline`	256	1	3.404 670 675	0.000 202 060
sequential	Fortran	Intel	`seq`	`inline`	256	1	3.191 272 063	0.000 202 000
sequential	Fortran	Intel	`seq`	`laplap`	256	1	2.931 468 021	0.000 001 311
sequential	Fortran	PGI	`seq`	`inline`	256	1	3.842 862 485	0.000 202 656
sequential	Fortran	PGI	`seq`	`laplap`	256	1	8.865 868 781	0.000 001 252
sequential	Fortran	GNU	`seq`	`laplap`	256	1	7.389 331 691	0.000 002 980
sequential	Fortran	GNU	`seq`	`inline`	256	1	12.474 772 310	0.000 202 060
sequential	C++	Cray	`seq`	`inline`	256	1	3.492 983 928	0.000 202 596
sequential	C++	Cray	`seq`	`laplap`	256	1	3.122 607 394	0.000 001 311
sequential	C++	Intel	`seq`	`laplap`	256	1	3.279 959 130	0.000 001 311
sequential	C++	Intel	`seq`	`inline`	256	1	3.400 640 907	0.000 202 000

Type	Language	Compiler	Technology	Algorithm	N	# cpus	Runtime / s	Error
sequential	C++	PGI	`seq`	`laplap`	256	1	3.230 457 313	0.000 001 311
sequential	C++	PGI	`seq`	`inline`	256	1	22.124 676 380	0.000 202 656
sequential	C++	GNU	`seq`	`laplap`	256	1	3.089 441 058	0.000 002 980
sequential	C++	GNU	`seq`	`inline`	256	1	3.568 529 159	0.000 202 537
sequential	Rust	rustc	`seq`	`laplap`	256	1	25.200 077 507	0.000 001 311
sequential	Rust	rustc	`seq_fast`	`inline`	256	1	3.683 168 117	0.000 202 894
sequential	Rust	rustc	`seq_fma`	`inline`	256	1	3.934 476 747	0.000 202 835
sequential	Rust	rustc	`seq_unchecked`	`laplap`	256	1	3.449 033 414	0.000 001 311
sequential	Rust	rustc	`seq_unchecked`	`inline`	256	1	4.654 520 614	0.000 202 656
sequential	Rust	rustc	`seq_zip`	`laplap`	256	1	3.895 527 788	0.000 001 311
CPU	Fortran	Cray	`openmp`	`laplap`	256	1	30.245 981 195	0.000 000 000
CPU	Fortran	Cray	`openmp`	`inline`	256	1	38.241 462 180	0.000 202 537
CPU	Fortran	Cray	`openmp`	`inline`	256	2	19.377 665 488	0.000 202 537
CPU	Fortran	Cray	`openmp`	`laplap`	256	2	15.550 603 920	0.000 000 000
CPU	Fortran	Cray	`openmp`	`laplap`	256	4	8.447 524 754	0.000 000 000
CPU	Fortran	Cray	`openmp`	`inline`	256	4	10.549 038 636	0.000 202 537
CPU	Fortran	Cray	`openmp`	`laplap`	256	8	4.353 291 626	0.000 000 000
CPU	Fortran	Cray	`openmp`	`inline`	256	8	5.416 081 697	0.000 202 537
CPU	Fortran	Cray	`openmp`	`laplap`	256	12	3.448 752 719	0.000 000 000
CPU	Fortran	Cray	`openmp`	`inline`	256	12	4.102 749 609	0.000 202 537
CPU	Fortran	Intel	`openmp`	`inline`	256	1	137.295 032 835	0.000 202 000
CPU	Fortran	Intel	`openmp`	`laplap`	256	1	214.383 874 065	0.000 001 311
CPU	Fortran	Intel	`openmp`	`inline`	256	2	69.468 640 682	0.000 202 000
CPU	Fortran	Intel	`openmp`	`laplap`	256	2	108.636 372 985	0.000 001 311
CPU	Fortran	Intel	`openmp`	`inline`	256	4	36.590 875 823	0.000 202 000
CPU	Fortran	Intel	`openmp`	`laplap`	256	4	59.871 217 135	0.000 001 311
CPU	Fortran	Intel	`openmp`	`inline`	256	8	18.324 022 881	0.000 202 000
CPU	Fortran	Intel	`openmp`	`laplap`	256	8	28.578 581 259	0.000 001 311
CPU	Fortran	Intel	`openmp`	`inline`	256	12	13.748 759 168	0.000 202 000

Type	Language	Compiler	Technology	Algorithm	N	# cpus	Runtime / s	Error
CPU	Fortran	Intel	`openmp`	`laplap`	256	12	21.464 541 064	0.000 001 311
CPU	Fortran	PGI	`openmp`	`inline`	256	1	86.593 510 786	0.000 202 656
CPU	Fortran	PGI	`openmp`	`laplap`	256	1	93.429 512 218	0.000 001 252
CPU	Fortran	PGI	`openmp`	`inline`	256	2	43.779 877 414	0.000 202 656
CPU	Fortran	PGI	`openmp`	`laplap`	256	2	47.302 263 532	0.000 001 252
CPU	Fortran	PGI	`openmp`	`laplap`	256	4	25.816 850 432	0.000 001 252
CPU	Fortran	PGI	`openmp`	`inline`	256	4	23.942 746 237	0.000 202 656
CPU	Fortran	PGI	`openmp`	`inline`	256	8	12.223 778 246	0.000 202 656
CPU	Fortran	PGI	`openmp`	`laplap`	256	8	13.253 094 582	0.000 001 252
CPU	Fortran	PGI	`openmp`	`inline`	256	12	9.194 586 701	0.000 202 656
CPU	Fortran	PGI	`openmp`	`laplap`	256	12	9.935 357 121	0.000 001 252
CPU	Fortran	GNU	`openmp`	`laplap`	256	1	31.063 809 055	0.000 002 980
CPU	Fortran	GNU	`openmp`	`inline`	256	1	40.276 274 911	0.000 202 537
CPU	Fortran	GNU	`openmp`	`inline`	256	2	20.432 591 988	0.000 202 537
CPU	Fortran	GNU	`openmp`	`laplap`	256	2	15.758 773 197	0.000 002 980
CPU	Fortran	GNU	`openmp`	`laplap`	256	4	8.641 554 217	0.000 002 980
CPU	Fortran	GNU	`openmp`	`inline`	256	4	11.113 152 903	0.000 202 537
CPU	Fortran	GNU	`openmp`	`inline`	256	8	5.710 768 492	0.000 202 537
CPU	Fortran	GNU	`openmp`	`laplap`	256	8	4.443 938 391	0.000 002 980
CPU	Fortran	GNU	`openmp`	`laplap`	256	12	3.343 548 260	0.000 002 980
CPU	Fortran	GNU	`openmp`	`inline`	256	12	4.285 778 820	0.000 202 537
CPU	C++	Cray	`openmp`	`inline`	256	1	75.387 448 596	0.000 202 537
CPU	C++	Cray	`openmp`	`laplap`	256	1	109.001 998 502	0.000 001 252
CPU	C++	Cray	`openmp`	`inline`	256	2	38.043 766 661	0.000 202 537
CPU	C++	Cray	`openmp`	`laplap`	256	2	55.079 457 278	0.000 001 252
CPU	C++	Cray	`openmp`	`inline`	256	4	20.064 275 603	0.000 202 537
CPU	C++	Cray	`openmp`	`laplap`	256	4	29.051 048 885	0.000 001 252
CPU	C++	Cray	`openmp`	`inline`	256	8	10.049 724 166	0.000 202 537
CPU	C++	Cray	`openmp`	`laplap`	256	8	14.484 797 713	0.000 001 252

Type	Language	Compiler	Technology	Algorithm	N	# cpus	Runtime / s	Error
CPU	C++	Cray	openmp	inline	256	12	7.561 154 319	0.000 202 537
CPU	C++	Cray	openmp	laplap	256	12	10.865 948 326	0.000 001 252
CPU	C++	Intel	openmp	inline	256	1	3.407 796 365	0.000 202 000
CPU	C++	Intel	openmp	laplap	256	1	3.426 748 766	0.000 001 311
CPU	C++	Intel	openmp	inline	256	2	1.763 955 135	0.000 202 000
CPU	C++	Intel	openmp	laplap	256	2	1.750 897 095	0.000 001 311
CPU	C++	Intel	openmp	inline	256	4	0.977 710 786	0.000 202 000
CPU	C++	Intel	openmp	laplap	256	4	0.910 849 358	0.000 001 311
CPU	C++	Intel	openmp	inline	256	8	0.540 961 138	0.000 202 000
CPU	C++	Intel	openmp	laplap	256	8	0.483 652 824	0.000 001 311
CPU	C++	Intel	openmp	laplap	256	12	0.402 965 419	0.000 001 311
CPU	C++	Intel	openmp	inline	256	12	0.500 043 699	0.000 202 000
CPU	C++	PGI	openmp	laplap	256	1	24.208 402 158	0.000 001 311
CPU	C++	PGI	openmp	inline	256	1	37.947 309 346	0.000 202 656
CPU	C++	PGI	openmp	laplap	256	2	12.270 726 607	0.000 001 311
CPU	C++	PGI	openmp	inline	256	2	19.181 676 664	0.000 202 656
CPU	C++	PGI	openmp	laplap	256	4	6.692 345 132	0.000 001 311
CPU	C++	PGI	openmp	inline	256	4	10.470 035 102	0.000 202 656
CPU	C++	PGI	openmp	laplap	256	8	3.437 439 997	0.000 001 311
CPU	C++	PGI	openmp	inline	256	8	5.399 604 209	0.000 202 656
CPU	C++	PGI	openmp	laplap	256	12	2.620 101 298	0.000 001 311
CPU	C++	PGI	openmp	inline	256	12	4.093 089 451	0.000 202 656
CPU	C++	GNU	openmp	inline	256	1	30.070 329 474	0.000 202 537
CPU	C++	GNU	openmp	laplap	256	1	28.247 504 742	0.000 002 801
CPU	C++	GNU	openmp	laplap	256	2	14.392 181 971	0.000 002 801
CPU	C++	GNU	openmp	inline	256	2	15.279 960 240	0.000 202 537
CPU	C++	GNU	openmp	laplap	256	4	7.796 336 546	0.000 002 801
CPU	C++	GNU	openmp	inline	256	4	8.298 743 850	0.000 202 537
CPU	C++	GNU	openmp	inline	256	8	4.258 013 944	0.000 202 537

Type	Language	Compiler	Technology	Algorithm	N	# cpus	Runtime / s	Error
CPU	C++	GNU	`openmp`	`laplap`	256	8	3.993 382 518	0.000 002 801
CPU	C++	GNU	`openmp`	`laplap`	256	12	2.990 103 232	0.000 002 801
CPU	C++	GNU	`openmp`	`inline`	256	12	3.216 413 361	0.000 202 537
CPU	Rust	rustc	`par_axis_zip`	`laplap`	256	1	4.771 166 870	0.000 001 311
CPU	Rust	rustc	`par_axis_zip`	`inline`	256	1	31.969 764 855	0.000 202 656
CPU	Rust	rustc	`par_axis_zip`	`laplap`	256	2	2.519 201 004	0.000 001 311
CPU	Rust	rustc	`par_axis_zip`	`inline`	256	2	16.254 688 368	0.000 202 656
CPU	Rust	rustc	`par_axis_zip`	`laplap`	256	4	1.400 080 186	0.000 001 311
CPU	Rust	rustc	`par_axis_zip`	`inline`	256	4	9.111 933 019	0.000 202 656
CPU	Rust	rustc	`par_axis_zip`	`laplap`	256	8	0.949 193 580	0.000 001 311
CPU	Rust	rustc	`par_axis_zip`	`inline`	256	8	4.624 925 040	0.000 202 656
CPU	Rust	rustc	`par_axis_zip`	`laplap`	256	12	0.826 375 653	0.000 001 311
CPU	Rust	rustc	`par_axis_zip`	`inline`	256	12	3.464 884 008	0.000 202 656
CPU	Rust	rustc	`par_zip`	`laplap`	256	1	5.018 743 652	0.000 001 311
CPU	Rust	rustc	`par_zip`	`inline`	256	1	5.438 857 435	0.000 202 656
CPU	Rust	rustc	`par_zip`	`laplap`	256	2	3.031 659 269	0.000 001 311
CPU	Rust	rustc	`par_zip`	`inline`	256	2	2.911 830 537	0.000 202 656
CPU	Rust	rustc	`par_zip`	`laplap`	256	4	1.715 940 160	0.000 001 311
CPU	Rust	rustc	`par_zip`	`inline`	256	4	1.573 115 345	0.000 202 656
CPU	Rust	rustc	`par_zip`	`inline`	256	8	1.018 059 593	0.000 202 656
CPU	Rust	rustc	`par_zip`	`laplap`	256	8	1.270 822 343	0.000 001 311
CPU	Rust	rustc	`par_zip`	`laplap`	256	12	1.101 276 648	0.000 001 311
CPU	Rust	rustc	`par_zip`	`inline`	256	12	0.880 914 964	0.000 202 656
GPU	Fortran	PGI	`cuda`	`laplap`	256	12	17.308 406 370	0.000 001 252
GPU	Fortran	PGI	`cuda`	`inline`	256	12	18.479 506 537	0.000 202 656
GPU	Fortran	PGI	`openmp_target`	`inline`	256	12	5.707 627 107	0.000 202 656
GPU	Fortran	PGI	`openmp_target`	`laplap`	256	12	8.992 560 937	0.000 001 252
GPU	C++	PGI	`openmp_target`	`inline`	256	12	5.802 441 647	0.000 202 656
GPU	C++	PGI	`openmp_target`	`laplap`	256	12	8.710 141 794	0.000 001 311

Type	Language	Compiler	Technology	Algorithm	N	# cpus	Runtime / s	Error
GPU	C++	GNU	`cuda`	`inline`	256	12	7.474 025 668	0.000 203 073
GPU	C++	GNU	`cuda`	`laplap`	256	12	15.061 924 979	0.000 001 311
GPU	Rust	rustc	`accel`	`inline`	256	12	9.061 606 428	0.000 202 656
GPU	Rust	rustc	`accel`	`laplap`	256	12	17.775 836 878	0.000 001 192
sequential	Fortran	Cray	`seq`	`laplap`	512	1	14.719 320 438	0.000 000 000
sequential	Fortran	Cray	`seq`	`inline`	512	1	15.379 167 639	0.000 202 596
sequential	Fortran	Intel	`seq`	`inline`	512	1	13.826 885 163	0.000 202 715
sequential	Fortran	Intel	`seq`	`laplap`	512	1	14.219 252 736	0.000 001 669
sequential	Fortran	PGI	`seq`	`inline`	512	1	16.299 587 816	0.000 202 537
sequential	Fortran	PGI	`seq`	`laplap`	512	1	35.584 118 638	0.000 001 431
sequential	Fortran	GNU	`seq`	`laplap`	512	1	29.666 141 715	0.000 002 980
sequential	Fortran	GNU	`seq`	`inline`	512	1	49.669 356 006	0.000 202 596
sequential	C++	Cray	`seq`	`inline`	512	1	14.739 980 037	0.000 203 192
sequential	C++	Cray	`seq`	`laplap`	512	1	15.626 475 614	0.000 001 788
sequential	C++	Intel	`seq`	`inline`	512	1	15.133 100 830	0.000 202 715
sequential	C++	Intel	`seq`	`laplap`	512	1	15.310 448 454	0.000 001 669
sequential	C++	PGI	`seq`	`laplap`	512	1	15.513 383 581	0.000 001 788
sequential	C++	PGI	`seq`	`inline`	512	1	89.288 411 140	0.000 202 954
sequential	C++	GNU	`seq`	`laplap`	512	1	15.044 901 353	0.000 002 980
sequential	C++	GNU	`seq`	`inline`	512	1	15.657 888 002	0.000 202 417
sequential	Rust	rustc	`seq`	`laplap`	512	1	99.935 973 450	0.000 001 788
sequential	Rust	rustc	`seq_fast`	`inline`	512	1	16.988 403 722	0.000 202 894
sequential	Rust	rustc	`seq_fma`	`inline`	512	1	16.272 931 134	0.000 202 715
sequential	Rust	rustc	`seq_unchecked`	`laplap`	512	1	16.094 893 388	0.000 001 788
sequential	Rust	rustc	`seq_unchecked`	`inline`	512	1	18.706 784 677	0.000 202 894
sequential	Rust	rustc	`seq_zip`	`laplap`	512	1	17.170 288 368	0.000 001 788
CPU	Fortran	Cray	`openmp`	`laplap`	512	1	119.584 261 617	0.000 000 000
CPU	Fortran	Cray	`openmp`	`inline`	512	1	153.375 804 438	0.000 202 537
CPU	Fortran	Cray	`openmp`	`laplap`	512	2	60.794 672 824	0.000 000 000

Type	Language	Compiler	Technology	Algorithm	N	# cpus	Runtime / s	Error
CPU	Fortran	Cray	`openmp`	`inline`	512	2	77.728 548 476	0.000 202 537
CPU	Fortran	Cray	`openmp`	`laplap`	512	4	33.104 969 087	0.000 000 000
CPU	Fortran	Cray	`openmp`	`inline`	512	4	42.345 910 414	0.000 202 537
CPU	Fortran	Cray	`openmp`	`laplap`	512	8	17.068 155 603	0.000 000 000
CPU	Fortran	Cray	`openmp`	`inline`	512	8	21.739 864 495	0.000 202 537
CPU	Fortran	Cray	`openmp`	`laplap`	512	12	13.387 751 743	0.000 000 000
CPU	Fortran	Cray	`openmp`	`inline`	512	12	16.497 406 902	0.000 202 537
CPU	Fortran	Intel	`openmp`	`inline`	512	1	549.709 539 745	0.000 202 715
CPU	Fortran	Intel	`openmp`	`laplap`	512	1	855.331 501 189	0.000 001 788
CPU	Fortran	Intel	`openmp`	`inline`	512	2	277.870 575 196	0.000 202 715
CPU	Fortran	Intel	`openmp`	`laplap`	512	2	432.597 003 126	0.000 001 788
CPU	Fortran	Intel	`openmp`	`inline`	512	4	146.480 966 987	0.000 202 715
CPU	Fortran	Intel	`openmp`	`laplap`	512	4	228.068 532 062	0.000 001 788
CPU	Fortran	Intel	`openmp`	`inline`	512	8	73.321 735 402	0.000 202 715
CPU	Fortran	Intel	`openmp`	`laplap`	512	8	114.044 836 768	0.000 001 788
CPU	Fortran	Intel	`openmp`	`inline`	512	12	55.016 608 721	0.000 202 715
CPU	Fortran	Intel	`openmp`	`laplap`	512	12	85.612 026 460	0.000 001 788
CPU	Fortran	PGI	`openmp`	`inline`	512	1	346.302 067 438	0.000 202 537
CPU	Fortran	PGI	`openmp`	`laplap`	512	1	372.621 738 507	0.000 001 431
CPU	Fortran	PGI	`openmp`	`inline`	512	2	175.203 242 713	0.000 202 537
CPU	Fortran	PGI	`openmp`	`laplap`	512	2	188.521 140 323	0.000 001 431
CPU	Fortran	PGI	`openmp`	`inline`	512	4	95.495 021 752	0.000 202 537
CPU	Fortran	PGI	`openmp`	`laplap`	512	4	102.741 244 472	0.000 001 431
CPU	Fortran	PGI	`openmp`	`inline`	512	8	48.886 498 957	0.000 202 537
CPU	Fortran	PGI	`openmp`	`laplap`	512	8	52.588 901 490	0.000 001 431
CPU	Fortran	PGI	`openmp`	`inline`	512	12	36.704 416 557	0.000 202 537
CPU	Fortran	PGI	`openmp`	`laplap`	512	12	39.530 824 699	0.000 001 431
CPU	Fortran	GNU	`openmp`	`laplap`	512	1	123.829 671 083	0.000 002 980
CPU	Fortran	GNU	`openmp`	`inline`	512	1	161.146 683 390	0.000 202 417

Type	Language	Compiler	Technology	Algorithm	N	# cpus	Runtime / s	Error
CPU	Fortran	GNU	openmp	laplap	512	2	62.879 083 440	0.000 002 980
CPU	Fortran	GNU	openmp	inline	512	2	81.439 904 460	0.000 202 417
CPU	Fortran	GNU	openmp	laplap	512	4	34.592 643 109	0.000 002 980
CPU	Fortran	GNU	openmp	inline	512	4	44.426 257 590	0.000 202 417
CPU	Fortran	GNU	openmp	laplap	512	8	17.771 845 344	0.000 002 980
CPU	Fortran	GNU	openmp	inline	512	8	22.799 873 256	0.000 202 417
CPU	Fortran	GNU	openmp	laplap	512	12	13.248 066 747	0.000 002 980
CPU	Fortran	GNU	openmp	inline	512	12	17.135 422 787	0.000 202 417
CPU	C++	Cray	openmp	inline	512	1	301.368 814 233	0.000 203 192
CPU	C++	Cray	openmp	laplap	512	1	434.518 563 830	0.000 001 431
CPU	C++	Cray	openmp	inline	512	2	152.156 724 093	0.000 203 192
CPU	C++	Cray	openmp	laplap	512	2	219.446 933 624	0.000 001 431
CPU	C++	Cray	openmp	inline	512	4	80.334 794 014	0.000 203 192
CPU	C++	Cray	openmp	laplap	512	4	115.593 076 252	0.000 001 431
CPU	C++	Cray	openmp	inline	512	8	40.171 023 364	0.000 203 192
CPU	C++	Cray	openmp	laplap	512	8	57.726 051 621	0.000 001 431
CPU	C++	Cray	openmp	inline	512	12	30.161 465 808	0.000 203 192
CPU	C++	Cray	openmp	laplap	512	12	43.339 529 226	0.000 001 431
CPU	C++	Intel	openmp	inline	512	1	14.982 047 209	0.000 202 715
CPU	C++	Intel	openmp	laplap	512	1	15.449 800 299	0.000 001 669
CPU	C++	Intel	openmp	inline	512	2	7.866 974 628	0.000 202 715
CPU	C++	Intel	openmp	laplap	512	2	7.929 776 365	0.000 001 669
CPU	C++	Intel	openmp	inline	512	4	4.522 208 614	0.000 202 715
CPU	C++	Intel	openmp	laplap	512	4	4.429 047 138	0.000 001 669
CPU	C++	Intel	openmp	inline	512	8	3.714 581 468	0.000 202 715
CPU	C++	Intel	openmp	laplap	512	8	3.284 246 329	0.000 001 669
CPU	C++	Intel	openmp	laplap	512	12	3.479 251 224	0.000 001 669
CPU	C++	Intel	openmp	inline	512	12	3.664 529 787	0.000 202 715
CPU	C++	PGI	openmp	laplap	512	1	95.972 135 673	0.000 001 788

Type	Language	Compiler	Technology	Algorithm	N	# cpus	Runtime / s	Error
CPU	C++	PGI	`openmp`	`inline`	512	1	151.585 063 669	0.000 202 954
CPU	C++	PGI	`openmp`	`laplap`	512	2	48.658 773 902	0.000 001 788
CPU	C++	PGI	`openmp`	`inline`	512	2	76.804 654 607	0.000 202 954
CPU	C++	PGI	`openmp`	`laplap`	512	4	26.507 040 779	0.000 001 788
CPU	C++	PGI	`openmp`	`inline`	512	4	41.838 093 066	0.000 202 954
CPU	C++	PGI	`openmp`	`laplap`	512	8	13.789 813 970	0.000 001 788
CPU	C++	PGI	`openmp`	`inline`	512	8	21.415 525 858	0.000 202 954
CPU	C++	PGI	`openmp`	`laplap`	512	12	10.258 350 718	0.000 001 788
CPU	C++	PGI	`openmp`	`inline`	512	12	16.147 286 146	0.000 202 954
CPU	C++	GNU	`openmp`	`laplap`	512	1	112.533 933 082	0.000 003 219
CPU	C++	GNU	`openmp`	`inline`	512	1	120.618 147 457	0.000 202 417
CPU	C++	GNU	`openmp`	`laplap`	512	2	57.004 988 909	0.000 003 219
CPU	C++	GNU	`openmp`	`inline`	512	2	61.248 811 464	0.000 202 417
CPU	C++	GNU	`openmp`	`laplap`	512	4	31.107 942 064	0.000 003 219
CPU	C++	GNU	`openmp`	`inline`	512	4	33.266 080 721	0.000 202 417
CPU	C++	GNU	`openmp`	`laplap`	512	8	15.980 376 578	0.000 003 219
CPU	C++	GNU	`openmp`	`inline`	512	8	17.116 232 189	0.000 202 417
CPU	C++	GNU	`openmp`	`inline`	512	12	12.887 018 673	0.000 202 417
CPU	C++	GNU	`openmp`	`laplap`	512	12	12.059 437 914	0.000 003 219
CPU	Rust	rustc	`par_axis_zip`	`laplap`	512	1	19.137 982 623	0.000 001 788
CPU	Rust	rustc	`par_axis_zip`	`inline`	512	1	127.190 944 195	0.000 202 894
CPU	Rust	rustc	`par_axis_zip`	`laplap`	512	2	10.342 595 944	0.000 001 788
CPU	Rust	rustc	`par_axis_zip`	`inline`	512	2	64.674 083 867	0.000 202 894
CPU	Rust	rustc	`par_axis_zip`	`laplap`	512	4	7.014 591 003	0.000 001 788
CPU	Rust	rustc	`par_axis_zip`	`inline`	512	4	36.249 744 327	0.000 202 894
CPU	Rust	rustc	`par_axis_zip`	`laplap`	512	8	6.104 738 306	0.000 001 788
CPU	Rust	rustc	`par_axis_zip`	`inline`	512	8	18.148 200 599	0.000 202 894
CPU	Rust	rustc	`par_axis_zip`	`laplap`	512	12	5.988 082 076	0.000 001 788
CPU	Rust	rustc	`par_axis_zip`	`inline`	512	12	13.561 551 683	0.000 202 894

Type	Language	Compiler	Technology	Algorithm	N	# cpus	Runtime / s	Error
CPU	Rust	rustc	`par_zip`	`inline`	512	1	21.387 776 169	0.000 202 894
CPU	Rust	rustc	`par_zip`	`laplap`	512	1	22.464 976 491	0.000 001 788
CPU	Rust	rustc	`par_zip`	`inline`	512	2	11.299 204 565	0.000 202 894
CPU	Rust	rustc	`par_zip`	`laplap`	512	2	12.289 275 146	0.000 001 788
CPU	Rust	rustc	`par_zip`	`laplap`	512	4	8.505 415 190	0.000 001 788
CPU	Rust	rustc	`par_zip`	`inline`	512	4	5.954 463 103	0.000 202 894
CPU	Rust	rustc	`par_zip`	`inline`	512	8	4.070 255 038	0.000 202 894
CPU	Rust	rustc	`par_zip`	`laplap`	512	8	7.574 286 692	0.000 001 788
CPU	Rust	rustc	`par_zip`	`laplap`	512	12	7.365 461 140	0.000 001 788
CPU	Rust	rustc	`par_zip`	`inline`	512	12	3.934 455 959	0.000 202 894
GPU	Fortran	PGI	`cuda`	`laplap`	512	12	66.680 631 226	0.000 001 431
GPU	Fortran	PGI	`cuda`	`inline`	512	12	73.032 562 831	0.000 202 954
GPU	Fortran	PGI	`openmp_target`	`inline`	512	12	22.405 046 436	0.000 202 537
GPU	Fortran	PGI	`openmp_target`	`laplap`	512	12	34.673 788 876	0.000 001 431
GPU	C++	PGI	`openmp_target`	`inline`	512	12	22.913 292 070	0.000 202 954
GPU	C++	PGI	`openmp_target`	`laplap`	512	12	33.634 764 148	0.000 001 788
GPU	C++	GNU	`cuda`	`inline`	512	12	29.365 935 965	0.000 203 013
GPU	C++	GNU	`cuda`	`laplap`	512	12	59.235 756 462	0.000 001 788
GPU	Rust	rustc	`accel`	`inline`	512	12	31.018 328 040	0.000 202 775
GPU	Rust	rustc	`accel`	`laplap`	512	12	62.594 685 755	0.000 001 788
sequential	Fortran	Cray	`seq`	`inline`	1024	1	66.237 637 588	0.000 202 537
sequential	Fortran	Cray	`seq`	`laplap`	1024	1	70.325 128 016	0.000 000 000
sequential	Fortran	Intel	`seq`	`laplap`	1024	1	58.239 505 197	0.000 001 788
sequential	Fortran	Intel	`seq`	`inline`	1024	1	64.950 393 936	0.000 202 835
sequential	Fortran	PGI	`seq`	`inline`	1024	1	71.589 372 425	0.000 202 656
sequential	Fortran	PGI	`seq`	`laplap`	1024	1	141.452 138 593	0.000 001 550
sequential	Fortran	GNU	`seq`	`laplap`	1024	1	118.688 713 958	0.000 003 219
sequential	Fortran	GNU	`seq`	`inline`	1024	1	205.715 177 009	0.000 202 537
sequential	C++	Cray	`seq`	`inline`	1024	1	65.752 372 372	0.000 203 013

Type	Language	Compiler	Technology	Algorithm	N	# cpus	Runtime / s	Error
sequential	C++	Cray	`seq`	`laplap`	1024	1	66.039 257 600	0.000 001 788
sequential	C++	Intel	`seq`	`laplap`	1024	1	64.402 517 990	0.000 001 788
sequential	C++	Intel	`seq`	`inline`	1024	1	68.714 996 201	0.000 202 835
sequential	C++	PGI	`seq`	`laplap`	1024	1	65.481 975 505	0.000 001 788
sequential	C++	PGI	`seq`	`inline`	1024	1	357.436 376 031	0.000 203 013
sequential	C++	GNU	`seq`	`laplap`	1024	1	63.489 961 235	0.000 003 219
sequential	C++	GNU	`seq`	`inline`	1024	1	70.844 715 034	0.000 202 537
sequential	Rust	rustc	`seq`	`laplap`	1024	1	398.239 593 129	0.000 002 027
sequential	Rust	rustc	`seq_fast`	`inline`	1024	1	76.184 909 134	0.000 202 715
sequential	Rust	rustc	`seq_fma`	`inline`	1024	1	72.057 852 046	0.000 202 656
sequential	Rust	rustc	`seq_unchecked`	`laplap`	1024	1	65.822 934 482	0.000 002 027
sequential	Rust	rustc	`seq_unchecked`	`inline`	1024	1	81.747 167 140	0.000 202 954
sequential	Rust	rustc	`seq_zip`	`laplap`	1024	1	67.831 611 418	0.000 002 027
CPU	Fortran	Cray	`openmp`	`laplap`	1024	1	477.547 237 670	0.000 000 000
CPU	Fortran	Cray	`openmp`	`inline`	1024	1	629.285 518 450	0.000 202 656
CPU	Fortran	Cray	`openmp`	`laplap`	1024	2	242.864 737 913	0.000 000 000
CPU	Fortran	Cray	`openmp`	`inline`	1024	2	317.780 303 740	0.000 202 656
CPU	Fortran	Cray	`openmp`	`laplap`	1024	4	133.240 820 146	0.000 000 000
CPU	Fortran	Cray	`openmp`	`inline`	1024	4	172.868 119 282	0.000 202 656
CPU	Fortran	Cray	`openmp`	`laplap`	1024	8	69.017 595 819	0.000 000 000
CPU	Fortran	Cray	`openmp`	`inline`	1024	8	89.007 179 073	0.000 202 656
CPU	Fortran	Cray	`openmp`	`laplap`	1024	12	53.417 788 306	0.000 000 000
CPU	Fortran	Cray	`openmp`	`inline`	1024	12	67.216 642 337	0.000 202 656
CPU	Fortran	Intel	`openmp`	`inline`	1024	1	2199.037 422 973	0.000 202 835
CPU	Fortran	Intel	`openmp`	`laplap`	1024	1	3416.980 869 262	0.000 001 788
CPU	Fortran	Intel	`openmp`	`inline`	1024	2	1113.058 842 875	0.000 202 835
CPU	Fortran	Intel	`openmp`	`inline`	1024	4	586.255 482 540	0.000 202 835
CPU	Fortran	Intel	`openmp`	`inline`	1024	8	294.027 128 936	0.000 202 835
CPU	Fortran	Intel	`openmp`	`inline`	1024	12	220.098 726 793	0.000 202 835

Type	Language	Compiler	Technology	Algorithm	N	# cpus	Runtime / s	Error
CPU	Fortran	PGI	openmp	inline	1024	1	1388.922 967 721	0.000 202 656
CPU	Fortran	PGI	openmp	laplap	1024	1	1487.341 300 792	0.000 001 550
CPU	Fortran	PGI	openmp	inline	1024	2	702.924 293 300	0.000 202 656
CPU	Fortran	PGI	openmp	laplap	1024	2	751.999 374 927	0.000 001 550
CPU	Fortran	PGI	openmp	inline	1024	4	382.976 203 331	0.000 202 656
CPU	Fortran	PGI	openmp	laplap	1024	4	409.973 681 490	0.000 001 550
CPU	Fortran	PGI	openmp	inline	1024	8	196.034 563 335	0.000 202 656
CPU	Fortran	PGI	openmp	laplap	1024	8	211.332 731 700	0.000 001 550
CPU	Fortran	PGI	openmp	inline	1024	12	147.396 034 229	0.000 202 656
CPU	Fortran	PGI	openmp	laplap	1024	12	158.026 986 116	0.000 001 550
CPU	Fortran	GNU	openmp	laplap	1024	1	497.141 256 026	0.000 003 219
CPU	Fortran	GNU	openmp	inline	1024	1	655.336 196 002	0.000 202 537
CPU	Fortran	GNU	openmp	laplap	1024	2	252.589 498 127	0.000 003 219
CPU	Fortran	GNU	openmp	inline	1024	2	332.231 287 506	0.000 202 537
CPU	Fortran	GNU	openmp	laplap	1024	4	138.445 009 839	0.000 003 219
CPU	Fortran	GNU	openmp	inline	1024	4	180.553 755 817	0.000 202 537
CPU	Fortran	GNU	openmp	laplap	1024	8	71.743 025 334	0.000 003 219
CPU	Fortran	GNU	openmp	inline	1024	8	92.520 770 199	0.000 202 537
CPU	Fortran	GNU	openmp	laplap	1024	12	54.807 739 957	0.000 003 219
CPU	Fortran	GNU	openmp	inline	1024	12	69.890 050 471	0.000 202 537
CPU	C++	Cray	openmp	inline	1024	1	1204.362 626 288	0.000 202 835
CPU	C++	Cray	openmp	laplap	1024	1	1734.766 162 548	0.000 001 550
CPU	C++	Cray	openmp	inline	1024	2	608.103 598 799	0.000 202 835
CPU	C++	Cray	openmp	laplap	1024	2	875.958 504 564	0.000 001 550
CPU	C++	Cray	openmp	inline	1024	4	320.888 766 346	0.000 202 835
CPU	C++	Cray	openmp	laplap	1024	4	461.308 876 094	0.000 001 550
CPU	C++	Cray	openmp	inline	1024	8	160.578 802 756	0.000 202 835
CPU	C++	Cray	openmp	laplap	1024	8	230.555 301 647	0.000 001 550
CPU	C++	Cray	openmp	inline	1024	12	120.554 467 401	0.000 202 835

Type	Language	Compiler	Technology	Algorithm	N	# cpus	Runtime / s	Error
CPU	C++	Cray	`openmp`	`laplap`	1024	12	173.260 970 363	0.000 001 550
CPU	C++	Intel	`openmp`	`laplap`	1024	1	64.728 860 214	0.000 001 788
CPU	C++	Intel	`openmp`	`inline`	1024	1	68.349 593 054	0.000 202 835
CPU	C++	Intel	`openmp`	`laplap`	1024	2	33.846 146 313	0.000 001 788
CPU	C++	Intel	`openmp`	`inline`	1024	2	35.616 234 886	0.000 202 835
CPU	C++	Intel	`openmp`	`inline`	1024	4	19.733 866 485	0.000 202 835
CPU	C++	Intel	`openmp`	`laplap`	1024	4	23.771 454 806	0.000 001 788
CPU	C++	Intel	`openmp`	`inline`	1024	8	15.036 968 414	0.000 202 835
CPU	C++	Intel	`openmp`	`laplap`	1024	8	28.949 695 321	0.000 001 788
CPU	C++	Intel	`openmp`	`inline`	1024	12	14.741 124 362	0.000 202 835
CPU	C++	Intel	`openmp`	`laplap`	1024	12	28.675 003 647	0.000 001 788
CPU	C++	PGI	`openmp`	`laplap`	1024	1	382.426 927 928	0.000 001 788
CPU	C++	PGI	`openmp`	`inline`	1024	1	636.152 004 599	0.000 203 013
CPU	C++	PGI	`openmp`	`laplap`	1024	2	193.374 542 952	0.000 001 788
CPU	C++	PGI	`openmp`	`inline`	1024	2	329.833 181 614	0.000 203 013
CPU	C++	PGI	`openmp`	`laplap`	1024	4	105.950 239 486	0.000 001 788
CPU	C++	PGI	`openmp`	`inline`	1024	4	179.718 979 231	0.000 203 013
CPU	C++	PGI	`openmp`	`laplap`	1024	8	54.631 636 527	0.000 001 788
CPU	C++	PGI	`openmp`	`inline`	1024	8	92.652 265 647	0.000 203 013
CPU	C++	PGI	`openmp`	`laplap`	1024	12	40.995 917 337	0.000 001 788
CPU	C++	PGI	`openmp`	`inline`	1024	12	69.332 644 205	0.000 203 013
CPU	C++	GNU	`openmp`	`laplap`	1024	1	451.733 794 494	0.000 002 980
CPU	C++	GNU	`openmp`	`inline`	1024	1	504.270 391 975	0.000 202 537
CPU	C++	GNU	`openmp`	`laplap`	1024	2	229.486 529 638	0.000 002 980
CPU	C++	GNU	`openmp`	`inline`	1024	2	256.048 825 448	0.000 202 537
CPU	C++	GNU	`openmp`	`laplap`	1024	4	125.113 964 482	0.000 002 980
CPU	C++	GNU	`openmp`	`inline`	1024	4	138.916 344 602	0.000 202 537
CPU	C++	GNU	`openmp`	`laplap`	1024	8	64.648 734 874	0.000 002 980
CPU	C++	GNU	`openmp`	`inline`	1024	8	71.837 846 336	0.000 202 537

Type	Language	Compiler	Technology	Algorithm	N	# cpus	Runtime / s	Error
CPU	C++	GNU	`openmp`	`laplap`	1024	12	48.478 260 799	0.000 002 980
CPU	C++	GNU	`openmp`	`inline`	1024	12	53.447 665 930	0.000 202 537
CPU	Rust	rustc	`par_axis_zip`	`laplap`	1024	1	76.699 016 579	0.000 002 027
CPU	Rust	rustc	`par_axis_zip`	`inline`	1024	1	633.732 555 769	0.000 202 954
CPU	Rust	rustc	`par_axis_zip`	`laplap`	1024	2	42.466 908 318	0.000 002 027
CPU	Rust	rustc	`par_axis_zip`	`inline`	1024	2	321.950 237 957	0.000 202 954
CPU	Rust	rustc	`par_axis_zip`	`laplap`	1024	4	30.443 616 884	0.000 002 027
CPU	Rust	rustc	`par_axis_zip`	`inline`	1024	4	176.085 380 628	0.000 202 954
CPU	Rust	rustc	`par_axis_zip`	`laplap`	1024	8	29.524 311 560	0.000 002 027
CPU	Rust	rustc	`par_axis_zip`	`inline`	1024	8	89.574 194 886	0.000 202 954
CPU	Rust	rustc	`par_axis_zip`	`laplap`	1024	12	28.929 096 535	0.000 002 027
CPU	Rust	rustc	`par_axis_zip`	`inline`	1024	12	66.737 820 836	0.000 202 954
CPU	Rust	rustc	`par_zip`	`laplap`	1024	1	86.949 532 571	0.000 002 027
CPU	Rust	rustc	`par_zip`	`inline`	1024	1	89.427 058 876	0.000 202 954
CPU	Rust	rustc	`par_zip`	`inline`	1024	2	45.803 981 588	0.000 202 954
CPU	Rust	rustc	`par_zip`	`laplap`	1024	2	47.337 513 854	0.000 002 027
CPU	Rust	rustc	`par_zip`	`inline`	1024	4	25.619 337 037	0.000 202 954
CPU	Rust	rustc	`par_zip`	`laplap`	1024	4	33.409 632 585	0.000 002 027
CPU	Rust	rustc	`par_zip`	`laplap`	1024	8	29.587 967 139	0.000 002 027
CPU	Rust	rustc	`par_zip`	`inline`	1024	8	15.659 154 580	0.000 202 954
CPU	Rust	rustc	`par_zip`	`inline`	1024	12	14.872 323 183	0.000 202 954
CPU	Rust	rustc	`par_zip`	`laplap`	1024	12	28.993 921 892	0.000 002 027
GPU	Fortran	PGI	`cuda`	`laplap`	1024	12	264.623 296 004	0.000 001 550
GPU	Fortran	PGI	`cuda`	`inline`	1024	12	291.018 365 879	0.000 203 013
GPU	Fortran	PGI	`openmp_target`	`inline`	1024	12	89.827 018 011	0.000 202 656
GPU	Fortran	PGI	`openmp_target`	`laplap`	1024	12	136.896 795 750	0.000 001 550
GPU	C++	PGI	`openmp_target`	`inline`	1024	12	91.394 230 858	0.000 203 013
GPU	C++	PGI	`openmp_target`	`laplap`	1024	12	133.446 966 663	0.000 001 788
GPU	C++	GNU	`cuda`	`inline`	1024	12	116.511 333 305	0.000 203 073

Type	Language	Compiler	Technology	Algorithm	N	# cpus	Runtime / s	Error
GPU	C++	GNU	`cuda`	`laplap`	1024	12	235.366 580 540	0.000 001 788
GPU	Rust	rustc	`accel`	`inline`	1024	12	118.267 676 443	0.000 203 848
GPU	Rust	rustc	`accel`	`laplap`	1024	12	239.227 172 981	0.000 002 027

Appendix C

Open-source contributions

During the course of this thesis, the author was able to make significant contributions to the following open-source projects: `rust` [50], by adding support for inline NVIDIA PTX assembly; `corrosion` [48, 51], by fixing compilation warnings, and adding support for paths with spaces; `spack` [52], by adding support for extra compilation targets; and `rust-llvm-proxy` [49], by fixing issues related to statically compiled LLVM libraries.

Appendix D

Bibliography

[1] Dan Aloni. *Path Trimming In Nightly Rust*. Sept. 4, 2020. URL: https://blog.aloni.org/posts/path-trimming-in-rust-nightly/ (visited on 2020-09-07).

[2] Reid Atcheson. *Rust and C++ on Floating-Point Intensive Code*. Oct. 19, 2019. URL: https://www.reidatcheson.com/hpc/architecture/performance/rust/c++/2019/10/19/measure-cache.html (visited on 2020-07-20).

[3] *autocxx*. Google. URL: https://crates.io/crates/autocxx (visited on 2020-09-11).

[4] J. W. Backus et al. "The FORTRAN Automatic Coding System". In: *Papers Presented at the February 26-28, 1957, Western Joint Computer Conference: Techniques for Reliability*. IRE-AIEE-ACM '57 (Western). Los Angeles, California: Association for Computing Machinery, Feb. 26, 1957, pp. 188–198. ISBN: 978-1-4503-7861-1. DOI: 10/bn86gx. URL: https://doi.org/10.1145/1455567.1455599 (visited on 2020-08-06).

[5] *bindgen*. URL: https://crates.io/crates/bindgen (visited on 2020-09-11).

[6] Sergi Blanco-Cuaresma. *marblestation/benchmark-leapfrog*. June 15, 2018. URL: https://github.com/marblestation/benchmark-leapfrog (visited on 2020-08-11).

[7] Sergi Blanco-Cuaresma and Emeline Bolmont. "What Can the Programming Language Rust Do for Astrophysics?" In: *Proceedings of the International Astronomical Union* 12.S325 (2016), pp. 341–344. DOI: 10/gg7gcn.

[8] *blas-lapack-rs Wiki Home*. URL: https://github.com/blas-lapack-rs/blas-lapack-rs.github.io/wiki (visited on 2020-09-11).

[9] *C++ Compiler Support*. URL: https://en.cppreference.com/w/cpp/compiler_support (visited on 2020-08-07).

[10] Satinder Chopra. "An Interview with Sergey Fomel". In: *CSEG Recorder* 35.1 (Jan. 2010), pp. 12–18. URL: https://www.csegrecorder.com/interviews/view/interview-with-sergey-fomel (visited on 2020-08-16).

[11] Iris Christadler, Giovanni Erbacci, and Alan D. Simpson. "Performance and Productivity of New Programming Languages". In: *Facing the Multicore - Challenge II: Aspects of New Paradigms and Technologies in Parallel Computing*. Ed. by Rainer Keller, David Kramer, and Jan-Philipp Weiss. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2012, pp. 24–35. ISBN: 978-3-642-30397-5. DOI: 10.1007/978-3-642-30397-5_3. URL: https://doi.org/10.1007/978-3-642-30397-5_3 (visited on 2020-08-06).

[12] Sébastien Crozet. *nalgebra*. URL: https://crates.io/crates/nalgebra (visited on 2020-09-11).

[13] Amanieu d'Antras. *parking_lot*. URL: https://crates.io/crates/parking_lot (visited on 2020-09-11).

[14] Oliver Fuhrer. *High Performance Computing for Weather and Climate*. June 16, 2020. URL: https://github.com/ofuhrer/HPC4WC (visited on 2020-08-06).

[15] *Future Of Programming Languages*. URL: https://wiki.c2.com/?FutureOfProgrammingLangu (visited on 2020-09-08).

[16] Todd Gamblin et al. "The Spack Package Manager: Bringing Order to HPC Software Chaos". In: *SC'15: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2015, pp. 1–12. DOI: 10/gf7tmd.

[17] Alex Gaynor and Geoffrey Thomas. "Writing Linux Kernel Modules in Safe Rust". Linux Security Summit 2019 (San Diego, California, USA). Aug. 20, 2019. URL: https://lssna19.sched.com/event/RHaT/writing-linux-kernel-modules-in-safe-rust-geoffrey-thomas-two-sigma-investments-alex-gaynor-alloy (visited on 2020-09-07).

[18] Stjepan Glavina. *crossbeam*. URL: https://crates.io/crates/crossbeam (visited on 2020-09-11).

[19] Brendan Gregg. "The Flame Graph". In: *Communications of the ACM* 59.6 (2016), pp. 48–57. DOI: 10/gg9zj6.

[20] Nicolas Hahn. *One Program Written in Python, Go, and Rust*. July 1, 2019. URL: http://nicolas-hahn.com/python/go/rust/programming/2019/07/01/program-in-python-go-rust/ (visited on 2020-08-06).

[21] Brook Heisler. *rustacuda*. URL: https://crates.io/crates/rustacuda (visited on 2020-09-11).

[22] Ryan Hunt. *cbindgen*. URL: https://crates.io/crates/cbindgen (visited on 2020-09-11).

[23] *IBM Archives: John Backus*. Jan. 23, 2003. URL: https://www.ibm.com/ibm/history/exhibits/builders/builders_backus.html (visited on 2020-08-07).

[24] *ISO/IEC 14882:1998*. Standard 14882:1998. ISO, Sept. 1998, p. 732. URL: https://www.iso.org/cms/render/live/en/sites/isoorg/contents/data/standard/02/58/25845.html (visited on 2020-08-07).

[25] *ISO/IEC 14882:2003*. Standard 14882:2003. ISO, Oct. 2003. URL: https://www.iso.org/cms/render/live/en/sites/isoorg/contents/data/standard/03/81/38110.html (visited on 2020-08-07).

[26] *ISO/IEC 14882:2011*. Standard 14882:2011. ISO, Sept. 2011. URL: https://www.iso.org/cms/render/live/en/sites/isoorg/contents/data/standard/05/03/50372.html (visited on 2020-08-07).

[27] *ISO/IEC 14882:2014*. 14882:2014. ISO, Dec. 2014. URL: https://www.iso.org/cms/render/live/en/sites/isoorg/contents/data/standard/06/40/64029.html (visited on 2020-08-07).

[28] *ISO/IEC 14882:2017*. Standard 14882:2017. ISO, Dec. 2017. URL: https://www.iso.org/cms/render/live/en/sites/isoorg/contents/data/standard/06/85/68564.html (visited on 2020-08-07).

[29] *ISO/IEC 1539-1:1997*. Standard 1539-1:1997. ISO, Dec. 1997. URL: https://www.iso.org/cms/render/live/en/sites/isoorg/contents/data/standard/02/69/26933.html (visited on 2020-08-07).

[30] *ISO/IEC 1539-1:2004*. Standard 1539-1:2004. ISO, Nov. 2004. URL: https://www.iso.org/cms/render/live/en/sites/isoorg/contents/data/standard/03/96/39691.html (visited on 2020-08-07).

[31] *ISO/IEC 1539-1:2010*. 1539-1:2010. ISO, Oct. 2010. URL: https://www.iso.org/cms/render/live/en/sites/isoorg/contents/data/standard/05/04/50459.html (visited on 2020-08-07).

[32] *ISO/IEC 1539-1:2018*. 1539-1:2018. ISO, Nov. 2018. URL: https://www.iso.org/cms/render/live/en/sites/isoorg/contents/data/standard/07/23/72320.html (visited on 2020-08-07).

[33] *ISO/IEC 1539:1991*. 1539:1991. ISO, July 1991. URL: https://www.iso.org/cms/render/live/en/sites/isoorg/contents/data/standard/00/61/6128.html (visited on 2020-08-07).

[34] *ISO/IEC DIS 14882*. Standard DIS 14882. ISO. URL: https://www.iso.org/cms/render/live/en/sites/isoorg/contents/data/standard/07/93/79358.html (visited on 2020-08-07).

[35] Donald E. Knuth. "Structured Programming with Go to Statements". In: *ACM Computing Surveys* 6.4 (Dec. 1, 1974), pp. 261–301. ISSN: 0360-0300. DOI: 10/dbkfbk. URL: https://doi.org/10.1145/356635.356640 (visited on 2020-08-16).

[36] Matias Lindgren. *Comparing Parallel Rust and C++*. Mar. 23, 2020. URL: https://parallel-rust-cpp.github.io/ (visited on 2020-07-22).

[37] Ferdia McKeogh. *Rewriting FORTRAN Software In Rust*. July 14, 2020. URL: https://mckeogh.tech/post/shallow-water/ (visited on 2020-07-22).

[38] *netcdf*. URL: https://crates.io/crates/netcdf (visited on 2020-09-11).

[39] Anthony Nowell. *Are We Learning Yet?* URL: https://www.arewelearningyet.com/ (visited on 2020-09-11).

[40] *num-complex*. URL: https://crates.io/crates/num-complex (visited on 2020-09-11).

[41] David Padua. "The FORTRAN I Compiler". In: *Computing in Science & Engineering* 2.1 (2000), pp. 70–75. DOI: 10/fc8qm5. URL: https://ieeexplore.ieee.org/document/814661/.

[42] Anthony Perez. "Rust and C++ Performance on the Algorithmic Lovasz Local Lemma". Project Report. Stanford: Stanford University, Dec. 2017. 9 pp. URL: https://cs242.stanford.edu/f17/assets/projects/2017/aperez8.pdf.

[43] *rayon*. URL: https://crates.io/crates/rayon (visited on 2020-09-11).

[44] *rust-lang/rustfmt*. The Rust Programming Language, Sept. 11, 2020. URL: https://github.com/rust-lang/rustfmt (visited on 2020-09-11).

[45] *rust-lang/rustup*. The Rust Programming Language, Sept. 11, 2020. URL: https://github.com/rust-lang/rustup (visited on 2020-09-11).

[46] Bjarne Stroustrup. *Stroustrup: FAQ*. URL: https://www.stroustrup.com/bs_faq.html#really-say-that (visited on 2020-08-05).

[47] Bjarne Stroustrup. *The C++ Programming Language*. 1st. Addison-Wesley, 1985. ISBN: 0-201-12078-X.

[48] Michal Sudwoj. *Fixed dyn trait object deprecation warnings*. June 20, 2020. URL: https://github.com/AndrewGaspar/corrosion/pull/21 (visited on 2020-09-11).

[49] Michal Sudwoj. *Fixed LLVM path search*. June 1, 2020. URL: https://github.com/denzp/rustc-llvm-proxy/pull/13 (visited on 2020-09-11).

[50] Michal Sudwoj. *NVPTX support for new asm!* May 30, 2020. URL: https://github.com/rust-lang/rust/pull/72439 (visited on 2020-08-07).

[51] Michal Sudwoj. *Properly quote paths*. June 20, 2020. URL: https://github.com/AndrewGaspar/corrosion/pull/22 (visited on 2020-09-11).

[52] Michal Sudwoj. *Rust: added nvptx variant*. Sept. 8, 2020. URL: https://github.com/spack/spack/pull/18209 (visited on 2020-09-11).

[53] Ulrik "bluss" Sverdrup. *bluss/matrixmultiply*. June 22, 2020. URL: https://github.com/bluss/matrixmultiply (visited on 2020-08-11).

[54] Ulrik "bluss" Sverdrup. *gemm: A Rabbit Hole*. Mar. 28, 2016. URL: https://bluss.github.io/rust/2016/03/28/a-gemmed-rabbit-hole/ (visited on 2020-08-11).

[55] Ulrik "bluss" Sverdrup. *matrixmulitply*. URL: https://crates.io/crates/matrixmultiply (visited on 2020-09-11).

[56] Ulrik "bluss" Sverdrup. *ndarray*. URL: https://crates.io/crates/ndarray (visited on 2020-09-11).

[57] Toshiki Teramura. *accel*. URL: https://crates.io/crates/accel (visited on 2020-09-11).

[58] David Tolnay. *cxx*. URL: https://crates.io/crates/cxx (visited on 2020-09-11).

[59] Lloyd N. Trefethen. *Finite Difference and Spectral Methods for Ordinary and Partial Differential Equations*. 1996. 325 pp. URL: http://people.maths.ox.ac.uk/trefethen/pdetext.html (visited on 2020-08-16).

[60] Florian Wilkens. "Evaluation of Performance and Productivity Metrics of Potential Programming Languages in the HPC Environment". Bachelor's thesis. Universiy of Hamburg, Apr. 28, 2015. 78 pp. URL: https://github.com/1wilkens/thesis-ba (visited on 2020-08-05).

[61] *X3.9-1966*. X3. 9-1966. ANSI, Mar. 7, 1966. URL: https://wg5-fortran.org/ARCHIVE/Fortran66.pdf.

[62] *X3.9-1978*. ANSI, 1978. URL: https://wg5-fortran.org/ARCHIVE/Fortran77.html.

[63] Ming Xue. "High-Order Monotonic Numerical Diffusion and Smoothing". In: *Monthly Weather Review* 128.8 (2000), pp. 2853–2864. DOI: 10/chzjb2.

[64] Denys Zariaiev. *denzp/rust-ptx-support*. Mar. 3, 2020. URL: https://github.com/denzp/rust-ptx-support (visited on 2020-09-11).

[65] Denys Zariaiev. *ptx-builder*. URL: https://crates.io/crates/ptx-builder (visited on 2020-09-11).

Appendix E

Glossary

ANSI American National Standards Institute. 5

CSCS Swiss National Supercomputing Centre. 4, 5

HPC high-performance computing. 1, 2, 4, 9, 29

ISO International Organization for Standardization. 5, 6

SSL Scientific Software & Libraries. 5, 6

ETH

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

The signed declaration of originality is a component of every semester paper, Bachelor's thesis, Master's thesis and any other degree paper undertaken during the course of studies, including the respective electronic versions.

Lecturers may also require a declaration of originality for other written papers compiled for their courses.

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

Title of work (in block letters):

Rust programming language in the high-performance computing environment

Authored by (in block letters):

For papers written by groups the names of all authors are required.

Name(s):	**First name(s):**
Sudwoj	Michał Grzegorz

With my signature I confirm that

- I have committed none of the forms of plagiarism described in the 'Citation etiquette' information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

Place, date	**Signature(s)**
Eglisau, 2020-09-11	Michał Sudwoj

For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.