XaaS Containers: Performance-Portable Representation With Source and IR Containers

Marcin Copik
ETH Zürich
Zürich, Switzerland
mcopik@gmail.com

Eiman Alnuaimi
ETH Zürich
Zürich, Switzerland
ealnuaimi@student.ethz.ch

Alok Kamatar
University of Chicago
Chicago, USA
alokvk2@uchicago.edu

Valerie Hayot-Sasson
University of Chicago
Chicago, USA
valerie.hayot-sasson@etsmtl.ca

Alberto Madonna
ETH Zürich
Lugano, Switzerland
Swiss National Supercomputing Centre (CSCS)
Lugano, Switzerland
alberto.madonna@cscs.ch

Todd Gamblin
Lawrence Livermore National Laboratory (LLNL)
Livermore, USA
tgamblin@llnl.gov

Kyle Chard
University of Chicago
Chicago, USA
Argonne National Laboratory (ANL)
Chicago, USA
chard@uchicago.edu

Ian Foster
University of Chicago
Chicago, USA
Argonne National Laboratory (ANL)
Chicago, USA
foster@uchicago.edu

Torsten Hoefler
ETH Zürich
Zürich, Switzerland
Swiss National Supercomputing Centre (CSCS)
Zürich, Switzerland
htor@inf.ethz.ch

Abstract

High-performance computing (HPC) systems and cloud data centers are converging, and containers are becoming the default method of portable software deployment. Yet, while containers simplify software management, they face significant performance challenges in HPC environments as they must sacrifice hardware-specific optimizations to achieve portability. Although HPC containers can use runtime hooks to access optimized MPI libraries and GPU devices, they are limited by application binary interface (ABI) compatibility and cannot overcome the effects of early-stage compilation decisions. Acceleration as a Service (XaaS) proposes a vision of *performance-portable* containers, where a containerized application should achieve peak performance across all HPC systems. We present a practical realization of this vision through Source and Intermediate Representation (IR) containers, where we delay performance-critical decisions until the target system specification is known. We analyze specialization mechanisms in HPC software and propose a new LLM-assisted method for automatic discovery of specializations. By examining the compilation pipeline, we develop a methodology to build containers optimized for target architectures at deployment time. Our prototype demonstrates that new XaaS containers combine the convenience of containerization with the performance benefits of system-specialized builds.

SC '25, St Louis, MO, USA

ACM ISBN 979-8-4007-1466-5/2025/11
https://doi.org/10.1145/3712285.3759868

CCS Concepts

• **Computer systems organization** → **Cloud computing**; • **General and reference** → *Performance*; • **Software and its engineering** → **Software performance**; *System administration.*

Keywords

Containers, Intermediate Representation, Performance Portability

ACM Reference Format:
Marcin Copik, Eiman Alnuaimi, Alok Kamatar, Valerie Hayot-Sasson, Alberto Madonna, Todd Gamblin, Kyle Chard, Ian Foster, and Torsten Hoefler. 2025. XaaS Containers: Performance-Portable Representation With Source and IR Containers. In *The International Conference for High Performance Computing, Networking, Storage and Analysis (SC '25), November 16–21, 2025, St Louis, MO, USA.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3712285.3759868

XaaS Implementation: https://github.com/spcl/xaas-containers
XaaS Artifact: https://doi.org/10.5281/zenodo.17115960

1 Introduction

High-performance computing (HPC) systems and cloud data centers have been developed to address different goals—HPC aimed at peak performance, while cloud platforms focused on usability. Recent years have brought a shift towards the convergence of HPC and cloud systems: the architecture of supercomputers is becoming more commoditized [43, 54], and the popularity of HPC workloads is growing, fueled by the massive demand for machine learning (ML) training. Cloud introduces new user types and software development philosophies such as containers. Containers are built on the fundamental assumption that an application is shipped with all its dependencies configured and compiled, regardless of the type of

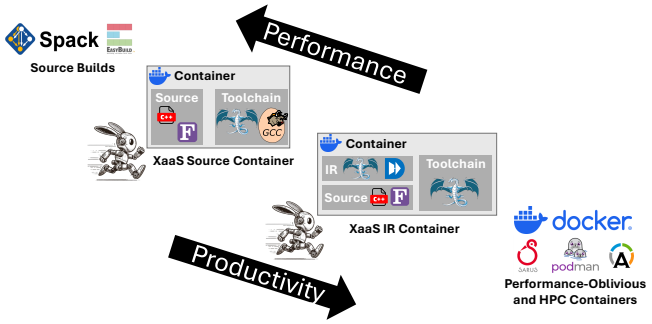

Figure 1: Continuum of software deployment in HPC: performance-portable XaaS containers provide better productivity than optimized builds, while avoiding limitations of traditional containers.

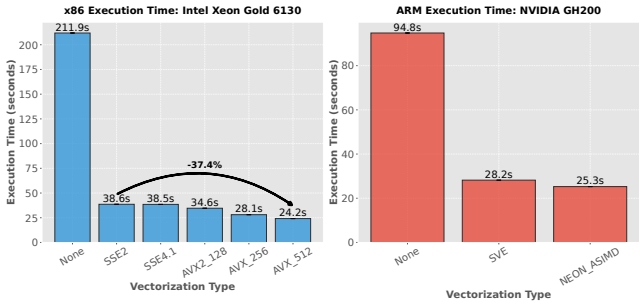

Figure 2: The impact of vectorization in GROMACS (16 threads, 100 timesteps, I/O time excluded): enabling newer features can improve performance, at the cost of creating a non-portable deployment.

system on which it will be deployed. Thus, containers offer workload portability, enabling execution at different cloud providers with minimal deployment complexity. For example, they can efficiently support rapid deployment to spot virtual machines, which are offered at a discount but might not provide the exact hardware specification expected by the user. Thus, users can decrease computation costs by scaling up their applications on a mix of virtual machines with different hardware configurations that cannot always be predicted. In HPC, containers can help with seamless deployment across systems with heterogeneous hardware configurations [3]. However, the adoption of containers in HPC is limited by the lack of **performance portability**, i.e., the ability to *"achieve excellent performance on a variety of architectures"* [42].

While containers simplify deployment, they do not necessarily benefit HPC system providers. Traditional HPC systems use modules containing carefully tailored applications and libraries that allow operators to *nudge* users toward performant solutions. However, this adds a major human cost of managing HPC software with complex dependencies [41]. However, when users can simply run any container image, neither they nor administrators will optimize the build, leading to poor performance as users bring ready-to-use but unoptimized software. To encourage performant solutions, we need a different approach to containers in HPC.

The adoption of containers is also limited by the many parameter configurations that need to be supported, with a combinatorial explosion of containers specialized toward different systems, runtimes, compilers, and parallelism approaches [31]. HPC containers can be (re)specialized at runtime with hooks defined by the Open Container Initiative (OCI) standard. These hooks replace libraries inside the container with system-specific versions. A common example is replacing MPI libraries [13], which requires implementations with compatible Application Binary Interface (ABI) [39]. The Libfabric installation can be replaced to access proprietary and custom network providers, which accelerates communication without changes to MPI [51]. However, solutions using Libfabric are not fully portable due to differences in the capabilities of network providers (Section 2.2). Furthermore, Fortran applications are known to lack ABI compatibility, which prevents straightforward runtime replacement of libraries such as BLAS or LAPACK. Finally, it is often too late to overcome the consequences of prior decisions that affect the generated code, like GPU acceleration and vectorization.

We propose that containers should be agnostic of selected configurations and target platforms, distributing software packages almost ready for installation while deferring performance decisions until the target system is known. However, this problem is difficult because HPC build systems are sophisticated, often Turing-complete tools that hardcode performance-critical decisions early in the build process, making complete analysis of these systems not only a massive engineering undertaking but potentially intractable.

Acceleration as a Service (XaaS) [42] introduced a new vision of HPC, where *performance-portable* containers can offer the convenience of containers with the performance of specialized builds. We realize this vision with the concept of **Source** and **Intermediate Representation (IR)** containers. We aim to strike a balance between traditional HPC practices, where builds are conducted entirely on the destination system, and limited container optimizations at runtime (Figure 1). We propose to deploy applications in a portable manner and provide the benefits of traditional containers: smaller size, faster deployment, and the ability to hide the application's source code. Both representations enable configuration for a specific system, allowing us to improve performance by tuning parameters such as vectorization, which enables hardware features unavailable on all platforms (Figure 2). Instead of distributing *multi-arch* Docker containers targeting different Instruction Set Architectures (ISAs), we distribute *multi-arch-IR* containers to support different compilers and toolchains, as long as they offer an intermediate representation to the end user.

We first analyze the broad world of HPC software to determine their specialization mechanisms (Section 2). Based on those results, we examine the compilation pipeline and analyze at which levels performance-critical decisions are made and how they can be delayed until the final hardware specification is known (Section 3). We configure many instances of the same project with different parameters. We isolate a shared core of IR files where compilation is identical across configurations or is unaffected by the change in performance-critical parameters (Section 4). We build a container image that is fully optimized and lowered to the target architecture only during *deployment* on the selected HPC system (Section 5). We demonstrate a prototype of IR containers with LLVM IR [47].

In this paper, we make the following contributions:

- We study specialization points in HPC applications and propose reorienting container deployment around them.
- We analyze the multi-layer HPC compilation stack, and design two solutions for performance-portable containers.
- We introduce IR containers that combine the convenience of generic containers with performance of specialized builds.

Table 1: Most important specialization points of selected HPC applications and benchmarks: analysis of representative applications shows a wide diversity of specializations for accelerated and high-performance computing.

Domain	Name	Architecture Spec.	GPU Acceleration	Parallelism	Vectorization	Performance Libraries
Molecular Dynamics	GROMACS [76]	Architecture-specific FFT	OpenCL, CUDA, SYCL, HIP	OpenMP, MPI	Automatic, many ISAs	BLAS/LAPACK, FFT (many)
Hydrodynamics	LULESH [1]	-	-	OpenMP, MPI	-	-
Electronic Structure	Quantum Espresso [35]	Compiler adaptations	CUDA, OpenACC	OpenMP, MPI	-	BLAS/LAPACK, ELPA [7], ScaLAPACK [19], FFT (many)
Lattice QCD	MILC [14]	Compiler adaptations	CUDA, HIP, SYCL	OpenMP, MPI	Compiler flags, many ISAs (Intel, AMD, PowerPC)	LAPACK, PRIMME [72], FFTW [29], QUDA [2, 20]
	OpenQCD [52]	Optimized for x86 CPUs	-	OpenMP, MPI	Assembly (SSE, AVX, FMA3)	-
Particle-in-Cell	VPIC [16], VPIC 2.0 [15]	Kokkos portability	CUDA	OpenMP, MPI	OpenMP and V4 library (many ISAs)	-
Cloud Physics	CloudSC [55]	System-specific toolchains	CUDA, SYCL, HIP, OpenACC	OpenMP, MPI	-	Atlas [25]
Weather & Climate	ICON [82]	System-specific toolchains	CUDA, HIP, OpenACC	OpenMP, MPI	System-specific compiler flags	BLAS/LAPACK
LLM Inference	llama.cpp [33]	Optimization flags	Eight, including CUDA, HIP, SYCL	OpenMP, pthreads	Intrinsics (AVX, AVX2, AVX512, AMX, NEON, …)	BLAS (OpenBLAS, MKL, BLIS [77])

Table 2: Levels of code portability and their implementations. Libraries like MPI can be shipped with a container and replaced dynamically at runtime or fully mounted inside the container during execution, as long as ABI compatibility is provided.

Level	Technology	Description	Portability Approach	Dependency Integration
Building	Spack [31], EasyBuild [44]	From-source package manager	Parameterized package compilation	Automatic, dependency resolver
Linking	Sarus [13], Apptainer [46]	HPC container runtime	Runtime binding, OCI hooks	Manual, CLI option, and host bind
Lowering	Linux Popcorn [9]	Multi-ISA binary system	Heterogeneous-OS containers	No direct integration
	H-containers [8, 79]	ISA-agnostic container with IRs	Container + recompilation	No direct integration
	NVIDIA PTX	Runtime JIT compilation	Virtual GPU architecture	No direct integration
Emulation	Wi4MPI [48], mpixlate [23]	MPI compatibility layer	Runtime emulation of MPI ABIs	No direct integration

2 State of HPC Software

To understand the challenge of performance portability in HPC, we begin with identifying configuration parameters that affect the performance by *specializing* to the target machine (Section 2.1). Then, we analyze the existing approaches for code portability, focusing on *when* the optimizations are applied (Section 2.2). This process helps us decide which build steps should be conducted before distributing software to the end user (Section 3).

2.1 Specialization Points

HPC applications are highly configurable since they aim to run on heterogeneous systems, with many built on custom and specialized hardware (Table 1). We define **specialization points** as *application parameters that must be known at the configuration and build time, stay constant throughout the entire application's lifetime, and whose values affect the final performance and portability.* In particular, we focus on parameters that dictate which specific hardware and software solutions should be employed by the final application. These options are not always mutually exclusive, as the application can support multiple GPU backends that are only selected at runtime. We consider the following categories of specialization points.

- Network fabric and communication library like MPI.
- Acceleration, such as NVIDIA, AMD, or Intel GPUs.
- CPU-specific optimizations such as vectorization.
- Libraries like BLAS, LAPACK, and FFT.

2.2 Portability Layers

We classify portability solutions into four categories based on the fraction of the build that is conducted on the target system (Table 2).

Building performs a full compilation of the application on the destination system. This approach provides the highest performance portability, at the cost of increased complexity - each user builds their copy manually or with the help of a package manager.

In **linking**, the dynamic dependencies of an existing application are replaced at runtime with an optimized and systems-specific implementation, e.g., through OCI hooks for containers. The main constraint here is the requirement of ABI compatibility, which prevents such replacements for BLAS/LAPACK libraries. For example, Libfabric allows the implementation of network communication with a standardized API and dynamic selection of network providers at runtime [62]. In practice, it still requires manual and specialized implementations because network providers differ in the support of libfabric features (Table 3). Furthermore, while libfabric replacement can accelerate a containerized MPI runtime [51], it might require additional plugins to support intra-node messaging [69]. Thus, relinking the libfabric installation is not a general method for performance specialization of an already compiled application.

Lowering replaces the intermediate representation with the final binary product at the target system. These solutions support multiple ISAs, even when the hardware popularity changes over time. HPC applications cannot be limited to x86 deployments, with primary examples of contenders being PowerPC in the past and ARM today, e.g., Fugaku's A64FX [67], Graviton CPUs [80] in AWS cloud, and NVIDIA's Grace Hopper superchip [70]. Similarly, this approach provides compatibility with different NVIDIA GPU architectures by deploying Parallel Thread Execution (PTX), an ISA for virtual GPU architectures in CUDA. PTX is JIT-compiled to a binary code, providing portability across many GPU generations [59].

```
#if not defined(HAVE_ANY_BLAS)
void transpose(double* A,
 double* B, int rows, int cols) {
 for (int i; i < rows; i++) {
  for (int j; j < cols; j++) {
   B[j * rows + i] =
    A[i * cols + j];
  }}
}
#endif
```

(a) Manual Implementation

```
#if defined(HAVE_OPENBLAS)
void transpose(double* A,
 double* B, int rows, int cols) {
 cblas_domatcopy(
  CblasRowMajor, CblasTrans,
  rows, cols, 1.0, A,
  cols, B, rows
 );
}
#endif
```

(b) OpenBLAS Implementation

```
#if defined(HAVE_MKL)
void transpose(double* A,
 double* B, int rows,
 int cols) {
 mkl_domatcopy(
  'R', 'T', rows, cols,
  1.0, A, cols, B, rows
 );
}
#endif
```

(c) Intel MKL Implementation

```
#if defined(HAVE_CUBLAS)
void transpose(double* A, double* B,
    int rows, int cols) {
 double alpha = 1.0, beta = 0.0;
 cublasDgeam(handle, CUBLAS_OP_T,
  CUBLAS_OP_N, rows, cols, &alpha, A,
  cols, &beta, nullptr, rows, B, rows
 );
}
#endif
```

(d) cuBLAS Implementation

Figure 3: Matrix transposition: a simple linear algebra kernel that has not been standardized, requiring a custom solution chosen at build time. While manual implementation is a safe choice that will work everywhere, it will prevent achieving the highest performance.

Table 3: Feature availability in libfabric 2.0 [62] providers (P - partial support, N/A - not used, ? - unknown). Libfabric offers a portable API, but implementations must still specialize to the hardware.

Feature	TCP (tcp)	IB (verbs)	Slingshot (cxi)	EFA (efa)	Omni-Path (opx)
Message	✔	✔	✘	✘	✘
Reliable Datagram	✔	P	✔	✔	✔
Datagram	✘	✔	✘	P	✘
Tagged Message	✔	P	✔	✔	✔
Directed Receive	✔	✘	✔	✔	✔
Multi Receive	✔	✘	✔	✔	✔
Atomic Operations	✘	P	✔	P	✔
Memory Registration	N/A	Basic	Scalable	Local	Scalable
Manual Progress	✘	✘	✔	✔	✔
Auto Progress	✔	✔	✘	✘	P
Wait Objects	✔	P	✔	✘	?
Completion Events	✔	✘	✔	✘	✘
Resource Management	✔	P	✔	P	✔
Scalable Endpoints	✘	✘	✘	✘	✔
Trigger Operations	✘	✘	✔	✘	✘

Finally, **emulation** attempts to patch incompatibilities at runtime without code modifications. An example is replacing MPI runtimes when the application has been built against MPI that is not ABI compatible with the host implementation [23, 48, 68].

3 HPC Specialization in XaaS

To fundamentally change the way we distribute HPC software, we first need to understand how *specialization points* affect the build and installation process (Section 3.1). Since discovering specialization points is complex due to the lack of standardization in build systems, we apply semi-automatic detection with the help of artificial intelligence (Section 3.2). By detecting specialization points, we can design performance-portable containers that delay the impact of specialization until we know the final specification (Section 4).

3.1 HPC Specialization

The build process of an application can be split into three major parts: **configuration** that resolves dependencies and decides what should be built, and how; **compilation and linking**, responsible for turning source files into libraries and executables; and **installation**, which places headers, binaries, and project resources in a selected destination. To create a transparent and seamless experience for HPC users, any solution must support all three steps.

During **configuration**, source modules and files are enabled or disabled depending on the selected specialization. Compiler flags are adjusted, and the build system adds compile-time definitions embedded into the application. Paths to dependencies are resolved, and additional packages can be fetched into the build directory.

Once source files are **compiled**, headers of chosen libraries will be introduced, preprocessing directives are applied, and compile-time definitions like C++ templates are resolved. After that stage, we can no longer switch between libraries that are not ABI-compatible since the application has been introduced to types with different representations and functions with incompatible signatures. Furthermore, preprocessing directives can potentially exclude certain code paths and already decide which kernels will be generated, as shown in the example of matrix transpose in BLAS libraries (Figure 3). Since this operation is not standardized, different implementations are needed, but they can only be enabled if the selected library is present in the system.

Once the source files are translated into the intermediate representation and optimized, the ISA is chosen, processor-specific decisions are made, and the final code is emitted. At this point, the code is no longer portable between different systems. Furthermore, it is no longer feasible to change vectorization settings or apply optimizations valid only on specific types of CPUs.

At the **linking** stage, applications are relinked to a specific implementation of a dependency. This decision can be changed later, as long as the library is linked dynamically and its implementations are ABI compatible. For example, an application compiled against MPICH can be relinked to use Cray's specialized MPICH implementation. While future MPI implementations will be ABI compatible [39], this method is currently limited since MPI types can have different implementations. After that point, the only possible performance adjustments are runtime options, such as switching network providers in applications built on top of libfabric.

Finally, the application is **installed**, which includes copying the contents of the package. Specialization affects the generation of project-specific headers and the installation of libraries, since the inclusion of specific dependencies is affected by user decisions.

3.2 Specialization Discovery

To generate the list of specialization points an application supports, we need to parse build scripts and understand what dependencies and optimizations can be selected during configuration. Unfortunately, this process is not standardized in common HPC programming languages, like C++ and Fortran. In addition to supporting different build systems such as autotools, handwritten Makefiles, CMake, Bazel, or custom scripts, there is often no single way of determining dependencies within one ecosystem. For example, third-party libraries can be located in CMake using standard CMake calls such as `find_package`, with custom find modules for libraries not supported by CMake, by using `pkg-config`, or with a manual

```
{
  "gpu_build": {"value": true, "build_flag": "-DGMX_GPU"},
  "gpu_backends": {
    "CUDA": {"min_version": "12.1", "flag": "-DGMX_GPU=CUDA"},
    "HIP": {"min_version": "5.4.3", "flag": "-DGMX_GPU=HIP"}
  },
  "vectorization": {
    "None": {"flag": "-DGMX_SIMD=None"},
    "SSE4.1": {"flag": "-DGMX_SIMD=SSE4.1"},
    "AVX2_256": {"flag": "-DGMX_SIMD=AVX2_256"},
    "AVX_512": {"flag": "-DGMX_SIMD=AVX_512"},
    "ARM_NEON_ASIMD": {"flag": "-DGMX_SIMD=ARM_NEON_ASIMD"}
}}
```


(a) Specialization Points


```
{
 "CPU Info": {
   "Architecture": "x86_64",
   "Vectorization": [
    "avx512f", "avx", "avx2", "sse4_1"
   ]
 },
 "GPU Backends": {
   "CUDA": { "version": "12.1",
     "lib": ["/lib/libcuda.so.1"],
   }
 }
}
```


(b) System Features


```
{
  "common_specialization": {
    "vectorization_flags": {
      "SSE4.1": "-DGMX_SIMD=SSE4.1",
      "AVX_512": "-DGMX_SIMD=AVX_512",
      "AVX2_256": "-DGMX_SIMD=AVX2_256"
    },
    "gpu_backends": {
      "CUDA": { "version": "12.1",
        "flag": "-DGMX_GPU=CUDA",
      }}
}
```


(c) Common Specialization Points

Figure 4: Simplified example of GROMACS' specialization points and how the checker determines the intersection of specialization points.

search for specific headers and libraries. Moreover, large projects often define custom routines for locating packages.

Analyzing configuration files to identify specialization points is difficult to automate due to the many diverse and unique patterns. At the same time, it is a task that humans can handle easily. Thus, we employ a Large Language Model (LLM) to help users identify specialization points by processing the project configuration files with a structured prompt. We apply *in-context learning* by including in the prompt examples of specialization options, build flags, and CMake commands, helping the LLM to extract specialization options accurately and capture all relevant choices in the build file. The model outputs a JSON file containing the detected specialization points. To enforce consistency and facilitate automated processing, we supply a predefined JSON schema, guiding the model to adhere to a structured format and minimizing anomalies. As the accuracy and correctness of LLM systems vary heavily, the results of LLM extraction still serve mainly as a guideline for the developer to prepare the final specification (Section 6.2).

On the target system, we collect information on system features and available specialization points. Then, we intersect these results with the specialization discovery of the application. At this point, we exclude the non-supported configuration options and present the user with a list of options for each specialization point. Figure 4 illustrates a subset of GROMACS's specialization points alongside the system features of our test environment. GROMACS supports OpenCL, SYCL, HIP, and CUDA as GPU backends, whereas the system is limited to CUDA and OpenCL. The automatic checker identifies the intersection of supported GPU backends, and allows the user to manually select the final specialization points.

4 XaaS Containers

In XaaS, we aim to resolve the two major limitations of existing containers—lack of performance portability and a combinatorial explosion of the number of final representations. First, we deploy **source containers** that bring the application and its environment to the final system (Section 4.1). The source container images contain the HPC application with development tools (Figure 5), and are only built for the target system once hardware configuration and all dependencies are known. Then, we propose that *intermediate representation* becomes a new mode for distributing software (Section 4.2). Intuitively, we distribute a container image where build steps are conducted until we cannot progress further without making performance-critical decisions (Section 4.3).

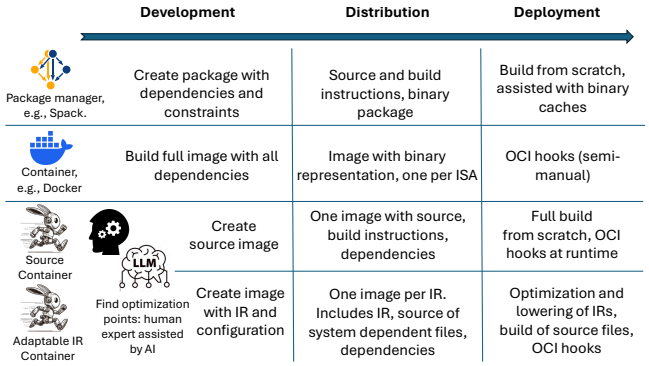

Figure 5: Performance-portable XaaS containers provide better productivity while avoiding limitations of traditional containers.

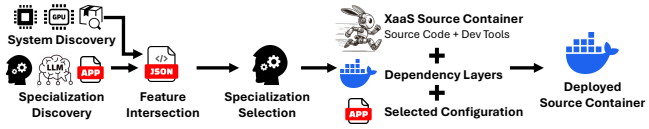

Figure 6: Deployment of source containers on HPC system.

The new types of containers require a new **deployment step**, when specialization points are matched against the system specification and user preferences. The remaining source files are compiled, architecture-specific optimizations are applied, and the entire application is lowered to the selected ISA. As a result, we obtain a new image different from the one provided in the registry, which allows for specialization of the application to the selected HPC system.

4.1 Source Containers

Source containers deliver the application source code, an open-source MPI implementation, and the build toolchain to the HPC system. This solution can support HPC applications and systems that benefit from specialized and vendor-provided compilers, which often do not expose their intermediate representation explicitly. Since no build steps are conducted before the deployment, this approach does not suffer from the large number of combinations: only one image is needed per toolchain and architecture.

The **deployment** begins by automatically detecting CPU features, accelerators, and the development environment (Figure 6). This step must be conducted on a compute node, and in an environment with all standard modules loaded. We augment the results with knowledge of standard HPC environments. For example, when a ROCm or CUDA installation is discovered, we assume the availability of rocFFT and cuFFT, respectively, even if they are not explicitly

detected. The discovery can be enhanced with solutions for labeling microarchitectural features, e.g., archspec [24], and strengthened with explicit system specification provided by system operators.

Then, we perform the automatic intersection of specializations (Section 3.2), and the user selects the best fit from the available options. After that, we generate a Dockerfile to create a new image that inherits from the source container and builds the application with selected options. We implement support for a subset of popular dependencies, inheriting dependency versions from the system environment when possible, and provide them as Docker layers or build steps. Other dependencies could be supported by employing package managers like Spack. Furthermore, we allow switching base images at deployment times to use optimized and recommended images for a specific platform, e.g., oneAPI images in Aurora (Section 6.3).

The new container is no longer portable and can often only be executed on that specific system. However, images derived from source containers should achieve near-native performance since we enable specializations available for bare metal applications, and the performance losses can only come from the container runtime itself. By providing the infrastructure for building and storing a single deployed container, we avoid the situation where users manually build multiple copies of the same application. From the user's point of view, the entire process is still convenient and relatively automatic—only a *cold pull* takes longer than a traditional container build since the very first user of a container on a system will have to wait for the build to finish. Users are only expected to select the values for discovered specialization points. However, this step could also be accelerated by allowing system operators to supply preferred configurations, e.g., preferring MKL on Intel systems over other BLAS/FFT libraries, relying on third-party configuration like in Spack [31], or providing the AI system with the application documentation to suggest the best option for the target platform.

4.2 IR Containers

IR containers are close to the original idea of containers, with the main goal of *build once and run anywhere*. However, the original build is augmented with the deployment step, responsible for the final optimizations and lowering to the target architecture. The application is distributed in the compiler's intermediate representation, and the image should not contain any object code that depends on the final architecture, as this would be neither portable nor performance-portable. In addition to selecting the architecture of the container image, we specify the *IR*, e.g., LLVM IR.

IR containers can suffer the same problem of combinatorial explosion that affects performance-oblivious containers. With multiple possible choices for parallelization, acceleration, hardware specialization, and communication, the number of *build configurations* grows combinatorially. The cost of building containers that include all combinations would be too high for many applications, and their size would be a major deployment problem. To make IR deployments practical, we need to deduplicate build configurations and build only the *unique* intermediate representation files:

Hypothesis 1: Let $P_1, P_2, \ldots, P_N$ be N different configurations of the same HPC application. Each configuration P_i compiles T_i different IR files. Let T' be the total number of **distinct** IR files produced in all N configurations. Then, $T' < \sum_i T_i$.

The set of unique results of compilation is not immediately detectable, as different compilation settings will obscure the analysis while not affecting the result. Many build configurations apply compilation flags globally to targets, e.g., C/C++ include flags or enabling OpenMP. To resolve the problem of too many build configurations, we apply a *behavioral* approach. Due to the complexity and intractability of the problem, we do not attempt to understand what build systems do but examine the compilation instructions of each target created in build configurations. We identify the differences between configurations and build only the delta when selecting a specific option. When two different configurations produce different targets from the same input, we build two different IR files for that target. Instead of storing all results of many different builds, we use one common set of IR files shared across all configurations, and a set of deltas applied only to selected configuration.

4.2.1 System Dependency. Before assembling the IR container, we define the necessary conditions for deploying the application's source files in that form.

Definition 1: A system-independent source file (*SI*) can be passed through the configuration and compilation stages without specifying the final software and hardware configuration.

A typical HPC example of such a source file would be numerical computations. Computations can be parallelized with OpenMP since the file can be compiled twice to IR, once with and once without OpenMP. However, MPI dependencies are not permitted for this category due to the lack of ABI compatibility in current runtimes. In practice, such files can be compiled with MPICH to be deployed with a widely accepted binary interface [56]. CUDA's PTX can be included in this category, while the binary representation of a compiled kernel (cubin) cannot.

Definition 2: A system-dependent source file (*SD*) cannot be compiled to a shared IR without sacrificing portability.

This category includes files with functionality conditionally enabled only for some configurations, and files requiring a dedicated compiler that does not expose its intermediate representation.

Hypothesis 2: HPC applications can be decomposed into two sets of source files: system-independent (*SI*) and system-dependent (*SD*). Most importantly, for most practical applications, $|SI| >> |SD|$.

The corollary of the last part of the hypothesis is critical: the effort of building a specialized pipeline makes sense only if the majority of the source code can be processed without knowing the final system; otherwise, source containers are a better solution.

4.3 IR Containers Pipeline

We create a modular container build pipeline (Figure 7) that solves multiple problems to determine the unique set of IR files:

(1) Combinatorial explosion of build configurations on projects with many specialization points.
(2) Code modules that can be excluded during the project's configuration, depending on specialization points.
(3) C/C++ preprocessor that can encode the effects of specialization points.
(4) Compilation flags that do not affect the result.

In particular, we implement optimizations that analyze the effects of OpenMP and vectorization flags.

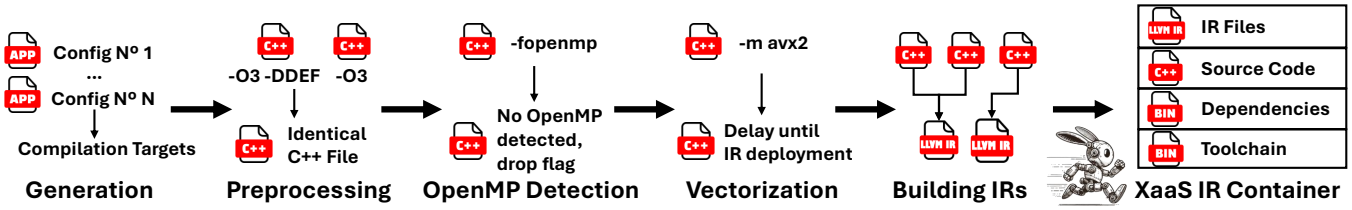

Figure 7: XaaS IR container. The modular pipeline reduces the build cost by detecting IR files shared by different configurations.

Configuration While we try to constrain the cost of building a container (Problem 1), we need to ensure that we do not prematurely exclude code modules that might become necessary during the deployment step (Problem 2). First, we generate a specialized build configuration for each combination of provided specialization points, e.g., LULESH [1] with two specialization points—MPI and OpenMP—will produce four different configurations. Each build configuration is created in a containerized environment, where the build directory is always mounted under the same path. This helps remove the effects of different locations on the generated compilation flags. The container is assembled from layers that provide the toolchain and dependencies of the selected application.

For each build configuration, we obtain the list of all compilation steps and associated compilation flags, e.g., by examining the compile commands database generated by CMake, which can be obtained without analyzing the internal structure of each build system. Other build systems can be supported by extracting compilation flags with third-party tools and compiler wrappers. We identify *compilation targets* and not source files, since a single source file can be mapped to multiple targets but with different compilation flags. Then, we compare the results of each build profile to identify the common denominator, a shared core of files that are always built in the same manner. In the case of LULESH, where each build consists of five source files, we obtain 20 IR files.

Preprocessing The configuration step is followed by a preprocessor evaluation to determine if different compile-time definitions produce a semantically different source file. Thus, we create preprocessed C/C++ files, hash them, and look for identical files. In LULESH, this step does not change the result since enabling MPI changes the source files, and the OpenMP compilation flag is attached to all files. Since not all files will use OpenMP, we apply a Clang AST analysis pass to detect if the processed file contains any OpenMP constructs. If two compilation targets from different build configurations have the same hash but differ only in the OpenMP flag, then we can treat them as the same. After that step, LULESH has been reduced from 20 to 14 different IR files.

Vectorization is another example of divergence across many builds of the same application. For example, GROMACS [76] supports nine different configurations for vectorization on x86 CPUs. Since LLVM vectorizers work at the IR level, we can ignore the architecture-specific flags when comparing different configurations of the same file. Instead, the vectorization will be applied during deployment once the final ISA is known. Our experiments show that LLVM optimizations need to be delayed as well, as otherwise the code will not be re-vectorized efficiently once the target is known.

Compilation During compilation, applications become aware of types that might introduce incompatibilities between different implementations of the same library. Thus, we need to isolate them

Figure 8: Deploying XaaS IR container: user selects one configuration that will be optimized and lowered to the architecture.

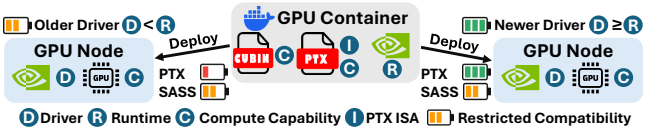

Figure 9: CUDA compatibility is determined by six parameters: two on host (driver and device capability), and four in container (runtime, PTX version, compute capability of PTX and device binary cubin).

from the rest of the codebase: the "default" partition (*SI*) can continue compilation as previously, while the new partition dependent on the ABI-problematic library (*SD*) will not be compiled at all until the final deployment for the target system.

MPI applications are the most important source of ABI compatibility, and we compile against MPICH to provide wide portability. Since we expect future MPI runtimes to be ABI compatible [39], we do not focus on this problem. To support Open MPI, XaaS containers could detect all files dependent on MPI and compile them multiple times against different ABIs, or employ portability layers [38].

Container Build: We generate a container with the LLVM, all build directories, IR files, and the source repository. The latter is necessary to support system-dependent files and perform the final installation. For each build configuration, we generate a specific installation file with instructions for compiling IR files and placing them in their respective locations. We do not include all image layers, e.g., GPU runtimes, as these will be reassembled at deployment.

GPU Compatibility: GPU runtimes target multiple architectures by generating either direct device code or portable PTX ISA for later JIT compilation (Figure 9). Portable containers can use the oldest supported CUDA runtime to ensure backward compatibility, while newer runtimes offer updated libraries and support for new hardware features at the cost of additional compatibility steps. We provide compatibility across CUDA minor versions, e.g., CUDA 12.x. First, we search for any use of compile-time definition indicating CUDA runtime version, which is a pessimistic check if the application might depend conditionally on API features unavailable in older drivers. Once we decide if the newest CUDA runtime can be used, we emit device binaries for all architectures and a PTX for the latest compute capability to support newer devices.

4.3.1 Deployment. The user selects specialization points from the list of parameters and their values chosen at configuration time. Then, we create a new container by assembling dependencies explicitly defined for that specialization. We select a subset of IRs for

that configuration, optimize and compile them, and let the build system finish linking (Figure 8). Image tag includes specialization points to support the coexistence of many builds.

Code Generation: We lower all IR files of a selected build configuration to the target architecture. This step can be much faster than a complete compilation of a C/C++ application. We also apply vectorization at this stage if it is detected during container build.

Linking: Once the binary code is generated, we can use the existing project configuration to link them together into final libraries and executables. Alternatively, linking flags for each target can be inferred from the build system, e.g., CMake exposes them explicitly. For runtime replacement system-optimized libraries, we can rely on the capabilities of existing HPC containers.

5 XaaS Containers in Practice

We implement a prototype of XaaS containers that can create source and IR images and then deploy them on selected HPC systems. For common choices of specialization points, like CUDA or Intel oneAPI, we provide an extensible fleet of Docker containers and manual installation steps. In source containers, we build a toolset for matching system specifications and specialization points, implement application-specific patching and integration, and provide two base source images, one for x64 and one for ARM64. The prototype of IR containers is built on top of Clang 19 and CMake, and includes a collection of Docker images with common runtimes and application dependencies. Users provide application-specific parameters and build steps, from which we generate all build configurations.

The new types of containers proposed in this work differ fundamentally from existing approaches, which raises new challenges in handling different applications and systems (Section 5.1). We transition from multi-arch container images to multi-IR images, highlighting that performance portability requires a change in container structure. Source and IR containers are vessels for delivering the correct environment and application, and they need to be transformed during the deployment step. XaaS containers will need new approaches to integrate into container ecosystems (Section 5.2).

5.1 Challenges

Can an IR container be cross-platform? XaaS needs to create one IR container per architecture, e.g., *IR, x86* and *IR, AArch64*. While the LLVM intermediate representation can be independent of the target system, this condition does not hold for practical compilation of C/C++ applications.[1] The IR is affected by the compilation platform, e.g., through type sizes, definitions included in system headers, inlined assembly, and intrinsics [45].

How to handle custom targets? Applications can use custom targets to fetch dependencies or generate source files. For example, when no FFT implementation is selected for GROMACS, it will build a custom implementation, but this does not happen at configuration time - only at build time. We assume that the user specifies all such targets, and we execute them before analyzing build configurations.

Which IRs are available? The IR container requires a toolchain that can export the intermediate representation and import it in subsequent compilation steps. Here, LLVM IR is the primary example. While GNU Compilers export the program representation in GIMPLE, this format cannot be imported later and lowered to the target architecture. On GPUs, the intermediate representation can be provided through PTX on NVIDIA architectures, and SPIR-V for applications using SYCL and OpenCL. However, at this moment, the intermediate representations of Intel DPC++/C++ Compiler and Cray Compiling Environment are unavailable to end users. When partial compilation to IR is impossible, source containers offer the fallback option. XaaS can also use high-level intermediate representations suitable for HPC optimizations, e.g., DaCe SDFG [11].

[1] C/C++ code *cannot* be compiled to a platform-independent LLVM IR [50].

5.2 Compatibility with OCI Containers

Our deployment model fundamentally differs from traditional containers: XaaS completely breaks the relationship between the image in the registry and the image on the system. This can raise the question of OCI compliance, since the container standard requires that changes to image layers are recorded in the manifest, leading to a new hash value and a new immutable identifier of the image [61]. However, XaaS publishes standard container images, pulls them from container registries, and produces specialized images in the same OCI-compliant format that can be consumed later by general-purpose and HPC-focused container runtimes. We introduce a new deployment tool customized for HPC specialization, but all other steps of container management - building, publishing, pulling, and running - are conducted with standard and existing container tools. Furthermore, virtually none of the current HPC container solutions preserve OCI compliance: images are generally flattened [13, 34, 65] (destroying the original OCI layers), converted to SquashFS, or use the custom Singularity Image Format (SIF) [46].

Image Architecture and Annotations: In XaaS containers, we propose that the source and IR formats become a new identifying feature of the container image. This would require that the OCI specification recognizes LLVM IR as a valid architecture. The current specification allows an image to have an architecture and a variant of the architecture [61]. Additionally, it reserves a list of *features* which can be used to encode deployment format.

OCI images use annotations for additional metadata in various media types (indexes, manifests, image configurations), with the latter consumed directly by container runtimes. In XaaS, annotations could embed specialization points of the HPC application. We propose that future versions could include specialization points as image annotations, allowing XaaS tools to query them before pulling and building the final image. Furthermore, it would simplify image tags and allow for the easy location of specialized images.

6 Evaluation

In the evaluation, we focus on the following research questions:

- Can LLM systems process large C++ project configurations?
- Can source containers provide performance portability?
- Can IR containers perform better than portable containers?
- Can IR containers optimize deployments?

6.1 Benchmarking Setup

We demonstrate the portability of our containers on three systems:

Table 4: Performance and cost of LLMs parsing GROMACS configuration. Token counts, latency, and estimated cost are averaged from 10 runs executed from Zürich, Switzerland. F1, precision, and recall metrics are aggregated across all runs and reported as Min/Median/Max per model.

Model Version	Tokens	Tokens Out	Time (s)	Cost ($)	$F1_{min}$	$F1_{med}$	$F1_{max}$	P_{min}	P_{med}	P_{max}	R_{min}	R_{med}	R_{max}
gemini-flash-1.5-exp	15803 ± 0	2333.5 ± 147.6	16.40 ± 1.00	0.002	0.863	0.902	0.942	0.838	0.898	0.948	0.868	0.909	0.936
gemini-flash-2-exp	15803 ± 0	2610.8 ± 189.4	11.96 ± 0.86	0.003	0.873	0.978	0.994	0.873	0.981	1.000	0.873	0.981	0.988
claude-3-5-haiku-20241022	17841 ± 0	1568.9 ± 174.2	20.09 ± 1.96	0.021	0.628	0.672	0.702	0.807	0.863	0.894	0.5	0.539	0.622
claude-3-5-sonnet-20241022	17841 ± 0	1528.7 ± 39.2	126.18 ± 335.31	0.077	0.661	0.672	0.692	0.875	0.878	0.882	0.539	0.544	0.57
claude-3-7-sonnet-20250219	17841 ± 0	3122.7 ± 155.1	50.29 ± 21.67	0.100	0.878	0.883	0.911	0.857	0.868	0.923	0.9	0.9	0.9
o3-mini-2025-01-31	13538 ± 0	8003.9 ± 1160.8	108.40 ± 40.02	0.050	0.559	0.924	0.968	0.549	0.909	0.987	0.57	0.923	1.000
gpt-4o-2024-08-06	13539 ± 0	1540.0 ± 146.1	26.06 ± 6.96	0.049	0.547	0.774	0.887	0.508	0.892	0.979	0.53	0.675	0.859

- **CSCS Ault** Heterogenous system with Sarus [13] containers. We use Ault23 (Intel 6130 CPU, V100 GPU), Ault25 (AMD EPYC 7742, A100 GPU), and Ault01-04 (Intel 6154 CPU).
- **CSCS Alps.Clariden** Cray system with the GH200 superchip, Cray Slingshot network, and Podman [73] containers.
- **Aurora** Cray system with Intel Xeon CPU Max CPUs, Intel Data Center GPU Max, and Apptainer [46] containers.

The deployment images for Clariden are built on the compute nodes, and we use a local development machine with Docker for Ault23 and Aurora, as neither system supports container building.

We consider two high-performance applications as case studies. **GROMACS 2025.0** [76] supports many vectorization options, with automatic detection of the best option on the system. In addition to MPI and OpenMP, it contains mutually exclusive GPU backends. **llama.cpp** [33] The C++ LLM inference engine achieves good portability by separating the implementation into multiple backends, which can be loaded dynamically at runtime.

6.2 Specialization Discovery

We propose automating the process of discovering specialization points with the help of LLMs. Given LLM's tendency to hallucinate and produce incorrect output, this raises the question: can these results be trusted? Thus, we evaluate the analysis of GROMACS with models from OpenAI, Anthropic, and Google. For each model, we apply *in-context learning* by providing examples from CMake configuration options, internal commands, and flags for GROMACS, Quantum Espresso, and Kokkos. We normalize the structure of specialization points, and compare specializations found in the ground truth and LLM result, counting true/false positives and negatives. We repeat the prompt 10 times and evaluate the characterization of the build system, vectorization, FFT and linear algebra libraries, parallel computing solutions, and GPU backends. Results in Table 4 show that the correctness and performance vary widely between models. Both GPT models produce inconsistent results across repetitions (F-score 0.55–0.97). Gemini models perfom best, which can be explained by the large context window. Examples of failures include returning only a subset of options (Claude 3.5, GPT-4o) and mixing FFT and linear algebra libraries (GPT-4o, Gemini 1.5).

The basic CMake configuration contains 13299 tokens, while all CMake scripts add 154,946 more tokens. Recommended configurations are described in the documentation that adds almost 1M tokens (without plots), which might be out of reach for many current models [49]. Documentation can be processed iteratively, but this would significantly increase processing time and further increase the uncertainty of the final result. Thus, while in-context learning

Figure 10: Performance portability of GROMACS between systems.

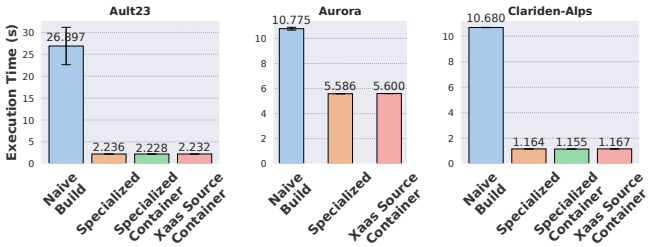

Figure 11: Performance portability of llama.cpp between systems.

can enable LLM models to perform well in the automatic analysis of specialization points, the processing pipeline requires tight human supervision and corrections, and performance preferences might be provided by system operators or application developers.

Generalization We evaluated llama.cpp for which we provide no prompt examples. We pass CMake configurations of llama.cpp and its main subproject *ggml* (2544 and 4574 tokens, respectively). Best performing models are Sonnet 3.7 (F1 0.55–0.62) and GPT-o3 (F1 0.62–0.7). Models often underperform due to minor discrepancies (inconsistent hyphen/underscore, missing *-D* prefix). Normalization improves performance: Sonnet 3.7 (F1 0.63–0.74), Gemini Flash 2 (F1 0.53–0.79), and GPT-o3 (F1 0.73–0.79). Additionally, ggml contains over 20 optimization flags; including them in evaluation decreases overall performance while improving Sonnet's best result to 0.78.

6.3 Performance Portability

We evaluate the performance portability of XaaS source containers by comparing them against local builds, specialized containers, and Spack packages or modules - when available. We deploy different source images based on system discovery and user input results.

6.3.1 GROMACS. We execute test cases A and B from UEABS [64] 30 times, and subtract the I/O overhead reported by GROMACS from timings (Figure 10). We use default GCC 11.4 in XaaS source images, and Spack-installed GCC 11.5 for other test cases. Naive build uses the default CMake command from the documentation,

which results with lack of GPU acceleration even when the CUDA module is loaded. Both naive and native builds pick up MKL from the HPC modules environment. On Ault23, we compare Spack installation with MPI and CUDA, and a second configuration with explicit selection of MKL, which achieves performance close to the XaaS source container. According to GROMACS logs, the default Spack installation performs worse in the CPU part of the application, indicating possible issues with multithreading or the automatically selected OpenBLAS. Spack cases uses the latest available GROMACS 2024.4, and a subsequent reevaluation of 2025.0 with test B on Ault23 demonstrated an average improvement of 1-2 seconds.

We use two baselines on Aurora: a hand-written specialized container and a module version of GROMACS 2024.5 since it cannot be installed by Spack. For XaaS and specialized containers, we use the oneAPI image recommended by system operators. The module version uses MPI, while other benchmarks use the internal Threads-MPI due to MPI compatibility issues in containers on Aurora (Section 6.5). However, the default source container uses only CPUs because the build is incompatible with Intel Max GPUs. There, GROMACS uses a compile-time definition specialized only for this device, found in the documentation but not the build configuration. For the *manual* fix, source containers need an additional source of knowledge, such as documentation parsing (Section 6.2) or specialization parameters provided by developers (Section 4.1).

Portable Container. A GPU-capable container is possible only with the SYCL backend. This *exotic* configuration [37] uses the vkFFT library [75], which can be compiled for one hardware backend only. Instead, we used the recently added oneMath library [28], which supports MKL and cuFFT simultaneously. We built GROMACS with the CUDA plugin for SYCL [21] in the oneAPI image, and compared it against our source containers on V100 (Ault23) and A100 (Ault25). The SYCL container is 11%-20% slower on test A, and fails to run test B. Portability is further limited by two factors: GPU compatibility still requires compile-time definitions, and the CUDA plugin generates code for one GPU architecture at a time.

6.3.2 llama.cpp. We use the internal llama.cpp benchmark, running 40 repetitions of prompt processing (512) and text generation (128), with a 4-bit quantized LLama-2-13B-chat model. We configure source containers to compiler and library versions close to those available on the system. On Aurora, we have to manually patch the source code to compile with the Intel icpx compiler, as the SYCL backend cannot be compiled with Clang 21. For XaaS source containers, we switch to the official oneAPI image at deployment. In this benchmark, the specialized build performs comparably to XaaS, while the naive default build does not enable GPU (Figure 11).

6.4 IR Containers

We evaluate the performance portability of IR containers with GROMACS on CPU, with OpenMP, and tuned against five different CPU architectures, from SSE4.1 to AVX-512. This requires discovering CPU tuning flags and delaying optimizations until final deployment (Section 4.3). For each variant, we build a separate container layer hosting `fftw3` library tuned for that architecture. Additionally, we build IR containers with CUDA 12.8 by generating IR files with embedded device code, which we later lower to the target platform. We compare against portable and specialized containers built with

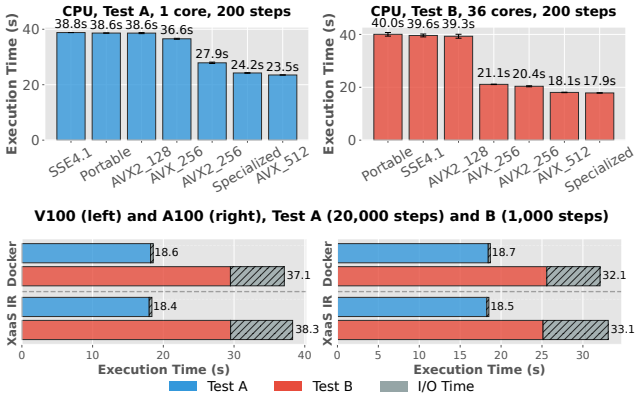

Figure 12: IR containers on CPU (top, Ault01-04) and GPU (bottom).

Clang 19 for SSE4.1 and AVX_512 (CPU), and CUDA containers (GPU). Figure 12 shows the runtime of GROMACS with I/O time excluded, demonstrating that a specialization of the IR container can improve performance by up to 2x when compared to a performance-oblivious container. We also evaluate a separate deployment of IR containers configured against CUDA and two vectorization levels for Ault23 and Ault25 nodes. On the GPU, we provide performance comparable to a specialized container, with a slight increase in I/O time for test case B.

Configurability and System Dependency This experiment validates positively Hypotheses 1 and 2 on GROMACS (Section 4.2). To deploy five GROMACS containers tuned to different ISAs, developers must build 8710 translation units (TU). In our container, we build only 2695 IRs (69% reduction). The reduction would not be possible without the XaaS pipeline, as 96% of compiled source files have incompatible build flags across projects; the primary reason is the inclusion of header files in the build directory. Preprocessing determines that only 14.3% of additional TUs require separate IR compilation. However, 95% of identical targets have different CPU tuning, which is resolved by the vectorization pass of our pipeline.

Four build configurations with two vectorization settings and CUDA require 7052 TUs, which we reduce to 2694 IRs (76% reduction). The build combinations obtained with enabling OpenMP and/or MPI require compiling and lowering 6976 TUs. Thanks to preprocessing and determining when the OpenMP flag has no effect, we build only 2333 IRs (66.4% reduction).

6.5 Network Performance

We focused on single-node deployments to demonstrate specialization to the available hardware and software, and did not evaluate distributed execution due to technical limitations on our systems. On Aurora, Apptainer containers did not function with MPI, and we had to use Thread-MPI instead. On Clariden, co-location of MPI ranks is needed to utilize the four GH200 chips on each node. However, the intra-node MPI communication is implemented separately from *cxi*, the Slingshot provider in Libfabric [69]. Thus, containerized MPIs can access the high-speed network through Libfabric replacement, but cannot use shared memory. While bare-metal Cray-MPICH achieves up to 64 GB/s on the same socket, co-located containers reach only up to 23.5 GB/s (OpenMPI). LinkX [66] is a Libfabric provider that combines remote and local communication,

and provides up to 64 (MPICH) and 70 (OpenMPI) GB/s of intra-node bandwidth. However, the provider is still experimental, e.g., it does not function well on all benchmarks and hardware.

7 Related Work

Building: Languages common in HPC, like C++ and Fortran, are notably missing in commonly used package managers. EasyBuild [44] builds HPC applications from source using specific toolchains, supporting hierarchical module creation [32]. Spack [31] is a package manager that parameterizes builds with constraints and versioned dependencies. Resolving dependencies can be reduced with declarative programming [30] or machine learning [57]. E4S provides curated HPC software stacks, including hardware-specific containers [41, 78]. Binary distribution is possible: Spack uses binary caches [71], and EESSI distributes EasyBuild stacks via network filesystems [27]. XaaS complements these tools by addressing the trade-off between container portability and performance.

Portable and HPC Containers: Injecting or replacing container libraries with host counterparts can be achieved with many container runtimes, but it can require expert knowledge of the system. Apptainer [46] supports semi-manual mounting of host MPI [6]. Charliecloud [65] uses heuristics to copy resource-specific files (NVIDIA, libfabric) into images, permanently modifying them. Sarus [13, 51] and Podman-HPC [73] use OCI hooks to inject host MPI and GPU libraries. XaaS can use the same hooks, but source containers can be compiled to use the version of the specialized library compatible with the one available on the system. Vendor container registries offer optimized but platform-specific images [5, 60].

Containers can already contain intermediate representation as Python and Java bytecode [83]. Popcorn Linux [9] enables cross-ISA live migration with a custom compiler and kernel that transform LLVM IR into multi-ISA binaries with compatible data layouts [40]. H-Containers [8, 79] achieve migration by decompiling to LLVM IR and recompiling to different ISAs. To the best of our knowledge, this is the only known use of IR for container deployment. However, it differs fundamentally from XaaS: we use IR-based representations to access customized performance features of each system.

Performance Portability: Performance portability often involves rewriting applications using models like OpenMP, OpenACC, or SYCL [17, 63]. Frameworks provide new abstractions for memory access (Kokkos [18]), loop parallelism (Raja [10]), and data-centric programming (DaCe [84]). Compilers can translate programming idioms to specialized libraries [36] and accelerators [53], and upgrade applications to use newer and specialized implementations of linear algebra libraries [22]. XaaS focuses on portable representations of existing applications without rewriting or requiring single-source code. We do not require applications to be single-source or use the same set of source files across all systems and devices.

Emulation, Translation, and JIT: Cross-ISA emulation, such as Docker with QEMU [26], is unsuitable for HPC due to performance overheads. Runtime MPI ABI translation layers like Wi4MPI [48] can incur performance overhead. Other tools include mpixlate [23] (compatibility with Cray MPI), MPITrampoline [68], Mukautuva [38], and MPI-Adapter2 [74]. JIT compilation, used in CUDA PTX, OpenCL, SYCL IR [4], allows for specialization of the final implementation by compiling the code dynamically.

8 Discussion

We demonstrate XaaS containers with representative HPC applications. However, modern HPC workloads are often not limited to a single application [12]. Large HPC workflows like MOFA [81] are built from several different tasks, each with its own requirements for CPU and GPU computation. Performance-portable containers could create a seamless deployment of a heterogeneous workflow across different HPC hardware. To transition the IR format to large and complex applications, we need to support dependency management (Section 8.1) and software installation (Section 8.2).

8.1 Dependency Management

When building different versions of an application, we should not repeat the entire build step for all dependencies. Instead, dependencies should be composed into a final package, as is already the case for package managers such as Spack [31]. Standard containers distribute dependencies as binary images assembled for the selected specialization. In IR containers, each dependency should be distributed in the IR form. However, they must be located during the configuration phase of IR containers, and build parameters are affected by compilation and linking flags of dependencies. This leads to a conflict: we want to deploy partially compiled applications, but still provide installation configuration to compose IR containers.

To support composability, future work can include creating *fat* binaries with embedded IR, similarly to existing approaches for deploying GPU device code in CUDA and SYCL [58]. This approach will generate a full installation target, allowing for seamless operation of build systems, while providing the necessary metadata and IRs to optimize and regenerate the dependency for the target. Furthermore, extended dependency management could support version constraints, similarly to existing solutions in package managers. This feature will restrict the matching process of specialization points, and prevent build failures caused by incompatibilities between the containerized application and its dependencies.

8.2 Installation

IR containers include the application's source code, even if the entire application is compiled to an intermediate representation. This is not a strict requirement of our method but a limitation of existing build systems. To finalize the application build, we need to perform linking and installation. However, these steps can include many user-defined and customized instructions, such as generating custom headers, and they cannot be easily extracted from the build configuration. Consequently, the IR image embeds all build configurations. Automating installation would reduce the complexity of the container and solve the problem of IR container composability. Furthermore, we can deduplicate installation targets as currently done with IR files. Then, the IR container will only need to contain a shared installation core and a delta for each build configuration.

9 Conclusions

XaaS Source and Intermediate Representation (IR) containers bring a new methodology for software management in HPC. We show

that changing the software distribution allows for delaying performance critical decisions until the deployment, avoiding performance limitations of traditional containers. Our prototype demonstrates that software deployments based on LLVM IR can significantly reduce the number of files that need to be generated without sacrificing performance.

Acknowledgments

This project received support by the SwissTwins project (funded by the Swiss State Secretariat for Education, Research and Innovation) and the ERC PSAP project (Grant Agreement No. 101002047). This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under contract DE-AC52-07NA27344 (LLNL-CONF-2010482), Lawrence Livermore National Security, LLC, and by Argonne National Laboratory under Contract DE-AC02-06CH11357. We thank the Swiss National Supercomputing Centre (CSCS) and Argonne Leadership Computing Facility (ALCF) for providing computational resources and technical support that facilitated this project. The authors leveraged Claude to assist with light editing of the manuscript. Copilot and Claude Code were used during code development. All content and ideas remain the authors' original work.

References

[1] 2011. *Hydrodynamics Challenge Problem, Lawrence Livermore N ational Laboratory.* Technical Report LLNL-TR-490254. 1–17 pages.

[2] 2021. QUDA. https://github.com/lattice/quda Accessed: 2025-08-25.

[3] Dong H. Ahn, Jim Garlick, Mark Grondona, Don Lipari, Becky Springmeyer, and Martin Schulz. 2014. Flux: A Next-Generation Resource Management Framework for Large HPC Centers. In *2014 43rd International Conference on Parallel Processing Workshops*. 9–17. doi:10.1109/ICPPW.2014.15

[4] Aksel Alpay and Vincent Heuveline. 2023. One Pass to Bind Them: The First Single-Pass SYCL Compiler with Unified Code Representation Across Backends. In *International Workshop on OpenCL*. ACM, Cambridge United Kingdom, 1–12. doi:10.1145/3585341.3585351

[5] AMD. 2025. AMD Infinity Hub. https://www.amd.com/en/developer/resources/infinity-hub.html Accessed: 2025-01-04.

[6] Apptainer. 2025. Apptainer and MPI applications. https://apptainer.org/docs/user/latest/mpi.html Accessed: 2025-01-04.

[7] T. Auckenthaler, V. Blum, H.-J. Bungartz, T. Huckle, R. Johanni, L. Krämer, B. Lang, H. Lederer, and P.R. Willems. 2011. Parallel solution of partial symmetric eigenvalue problems from electronic structure calculations. *Parallel Comput.* 37, 12 (2011), 783–794. doi:10.1016/j.parco.2011.05.002 6th International Workshop on Parallel Matrix Algorithms and Applications (PMAA'10).

[8] Antonio Barbalace, Mohamed L. Karaoui, Wei Wang, Tong Xing, Pierre Olivier, and Binoy Ravindran. 2020. Edge computing: the case for heterogeneous-ISA container migration. In *Proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments* (Lausanne, Switzerland) *(VEE '20)*. Association for Computing Machinery, New York, NY, USA, 73–87. doi:10.1145/3381052.3381321

[9] Antonio Barbalace, Robert Lyerly, Christopher Jelesnianski, Anthony Carno, Ho-Ren Chuang, Vincent Legout, and Binoy Ravindran. 2017. Breaking the Boundaries in Heterogeneous-ISA Datacenters. *SIGARCH Comput. Archit. News* 45, 1 (April 2017), 645–659. doi:10.1145/3093337.3037738

[10] David A. Beckingsale, Jason Burmark, Rich Hornung, Holger Jones, William Killian, Adam J. Kunen, Olga Pearce, Peter Robinson, Brian S. Ryujin, and Thomas RW Scogland. 2019. RAJA: Portable Performance for Large-Scale Scientific Applications. In *2019 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*. 71–81. doi:10.1109/P3HPC49587.2019.00012

[11] Tal Ben-Nun, Johannes de Fine Licht, Alexandros N. Ziogas, Timo Schneider, and Torsten Hoefler. 2019. Stateful dataflow multigraphs: a data-centric model for performance portability on heterogeneous architectures. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Denver, Colorado) *(SC '19)*. Association for Computing Machinery, New York, NY, USA, Article 81, 14 pages. doi:10.1145/3295500.3356173

[12] Tal Ben-Nun, Todd Gamblin, D. S. Hollman, Hari Krishnan, and Chris J. Newburn. 2020. Workflows are the New Applications: Challenges in Performance, Portability, and Productivity. In *2020 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*. 57–69. doi:10.1109/P3HPC51967.2020.00011

[13] Lucas Benedicic, Felipe A. Cruz, Alberto Madonna, and Kean Mariotti. 2019. Sarus: Highly Scalable Docker Containers for HPC Systems. In *High Performance Computing*, Michèle Weiland, Guido Juckeland, Sadaf Alam, and Heike Jagode (Eds.). Springer International Publishing, Cham, 46–60.

[14] Claude Bernard, Michael C. Ogilvie, Thomas A. DeGrand, Carleton E. DeTar, Steven A. Gottlieb, A. Krasnitz, R.L. Sugar, and D. Toussaint. 1991. Studying Quarks and Gluons On Mimd Parallel Computers. *The International Journal of Supercomputing Applications* 5, 4 (1991), 61–70. doi:10.1177/109434209100500406 arXiv:https://doi.org/10.1177/109434209100500406

[15] Robert Bird, Nigel Tan, Scott V. Luedtke, Stephen Lien Harrell, Michela Taufer, and Brian Albright. 2022. VPIC 2.0: Next Generation Particle-in-Cell Simulations . *IEEE Transactions on Parallel & Distributed Systems* 33, 04 (April 2022), 952–963. doi:10.1109/TPDS.2021.3084795

[16] K. J. Bowers, B. J. Albright, L. Yin, B. Bergen, and T. J. T. Kwan. 2008. Ultrahigh performance three-dimensional electromagnetic relativistic kinetic plasma simulationa). *Physics of Plasmas* 15, 5 (03 2008), 055703. doi:10.1063/1.2840133 arXiv:https://pubs.aip.org/aip/pop/article-pdf/doi/10.1063/1.2840133/14083352/055703 _1_online.pdf

[17] Marcel Breyer, Alexander Van Craen, and Dirk Pflüger. 2022. A Comparison of SYCL, OpenCL, CUDA, and OpenMP for Massively Parallel Support Vector Machine Classification on Multi-Vendor Hardware. In *Proceedings of the 10th International Workshop on OpenCL* (Bristol, United Kingdom, United Kingdom) *(IWOCL '22)*. Association for Computing Machinery, New York, NY, USA, Article 2, 12 pages. doi:10.1145/3529538.3529980

[18] H. Carter Edwards, Christian R. Trott, and Daniel Sunderland. 2014. Kokkos: Enabling manycore performance portability through polymorphic memory access patterns. *J. Parallel and Distrib. Comput.* 74, 12 (2014), 3202–3216. doi:10.1016/j.jpdc.2014.07.003 Domain-Specific Languages and High-Level Frameworks for High-Performance Computing.

[19] J. Choi, J.J. Dongarra, R. Pozo, and D.W. Walker. 1992. ScaLAPACK: a scalable linear algebra library for distributed memory concurrent computers . In *The Fourth Symposium on the Frontiers of Massively Parallel Computation*. IEEE Computer Society, Los Alamitos, CA, USA, 120,121,122,123,124,125,126,127. doi:10.1109/FMPC.1992.234898

[20] M.A. Clark, R. Babich, K. Barros, R.C. Brower, and C. Rebbi. 2010. Solving lattice QCD systems of equations using mixed precision solvers on GPUs. *Computer Physics Communications* 181, 9 (2010), 1517–1528. doi:10.1016/j.cpc.2010.05.002

[21] Codeplay. 2024. Install oneAPI for NVIDIA GPUs. https://developer.codeplay.com/products/oneapi/nvidia/2024.2.0/guides/get-started-guide-nvidia Accessed: 2025-08-25.

[22] Bruce Collie, Philip Ginsbach, and Michael F.P. O'Boyle. 2019. Type-Directed Program Synthesis and Constraint Generation for Library Portability . In *2019 28th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. IEEE Computer Society, Los Alamitos, CA, USA, 55–67. doi:10.1109/PACT.2019.00013

[23] Cray. 2025. mpixlate. https://cpe.ext.hpe.com/docs/24.03/mpt/mpixlate/mpixlate.html Accessed: 2025-01-04.

[24] Massimiliano Culpo, Gregory Becker, Carlos Eduardo Arango Gutierrez, Kenneth Hoste, and Todd Gamblin. 2020. archspec: A library for detecting, labeling, and reasoning about microarchitectures. In *2020 2nd International Workshop on Containers and New Orchestration Paradigms for Isolated Environments in HPC (CANOPIE-HPC)*. 45–52. doi:10.1109/CANOPIEHPC51917.2020.00011

[25] Willem Deconinck, Peter Bauer, Michail Diamantakis, Mats Hamrud, Christian Kühnlein, Pedro Maciel, Gianmarco Mengaldo, Tiago Quintino, Baudouin Raoult, Piotr K. Smolarkiewicz, and Nils P. Wedi. 2017. Atlas : A library for numerical weather prediction and climate modelling. *Computer Physics Communications* 220 (2017), 188–204. doi:10.1016/j.cpc.2017.07.006

[26] Docker. 2025. Multi-platform builds. https://docs.docker.com/build/building/multi-platform/ Accessed: 2025-01-04.

[27] Bob Dröge, Victor Holanda Rusu, Kenneth Hoste, Caspar van Leeuwen, Alan O'Cais, and Thomas Röblitz. 2023. EESSI: A cross-platform ready-to-use optimised scientific software stack. *Software: Practice and Experience* 53, 1 (2023), 176–210. doi:10.1002/spe.3075 arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/spe.3075

[28] Unified Acceleration (UXL) Foundation. 2025. oneAPI Math Library (oneMath). https://github.com/uxlfoundation/oneMath/tree/v0.8 Accessed: 2025-08-25.

[29] M. Frigo and S.G. Johnson. 2005. The Design and Implementation of FFTW3. *Proc. IEEE* 93, 2 (2005), 216–231. doi:10.1109/JPROC.2004.840301

[30] Todd Gamblin, Massimiliano Culpo, Gregory Becker, and Sergei Shudler. 2022. Using Answer Set Programming for HPC Dependency Solving. In *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–15. doi:10.1109/SC41404.2022.00040

[31] Todd Gamblin, Matthew LeGendre, Michael R. Collette, Gregory L. Lee, Adam Moody, Bronis R. de Supinski, and Scott Futral. 2015. The Spack package manager: bringing order to HPC software chaos. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Austin, Texas) *(SC '15)*. Association for Computing Machinery, New York, NY, USA, Article 40, 12 pages. doi:10.1145/2807591.2807623

[32] Markus Geimer, Kenneth Hoste, and Robert McLay. 2014. Modern Scientific Software Management Using EasyBuild and Lmod. In *2014 First International Workshop on HPC User Support Tools*. 41–51. doi:10.1109/HUST.2014.8

[33] Georgi Gerganov. 2025. llama.cpp. https://github.com/ggml-org/llama.cpp Accessed: 2025-01-04.

[34] Lisa Gerhardt, Wahid Bhimji, Shane Canon, Markus Fasel, Doug Jacobsen, Mustafa Mustafa, Jeff Porter, and Vakho Tsulaia. 2017. Shifter: Containers for HPC. *Journal of Physics: Conference Series* 898, 8 (oct 2017), 082021. doi:10.1088/1742-6596/898/8/082021

[35] Paolo Giannozzi, Stefano Baroni, Nicola Bonini, Matteo Calandra, Roberto Car, Carlo Cavazzoni, Davide Ceresoli, Guido L Chiarotti, Matteo Cococcioni, Ismaila Dabo, Andrea Dal Corso, Stefano de Gironcoli, Stefano Fabris, Guido Fratesi, Ralph Gebauer, Uwe Gerstmann, Christos Gougoussis, Anton Kokalj, Michele Lazzeri, Layla Martin-Samos, Nicola Marzari, Francesco Mauri, Riccardo Mazzarello, Stefano Paolini, Alfredo Pasquarello, Lorenzo Paulatto, Carlo Sbraccia, Sandro Scandolo, Gabriele Sclauzero, Ari P Seitsonen, Alexander Smogunov, Paolo Umari, and Renata M Wentzcovitch. 2009. QUANTUM ESPRESSO: a modular and open-source software project for quantum simulations of materials. *Journal of Physics: Condensed Matter* 21, 39 (sep 2009), 395502. doi:10.1088/0953-8984/21/39/395502

[36] Philip Ginsbach, Toomas Remmelg, Michel Steuwer, Bruno Bodin, Christophe Dubach, and Michael F. P. O'Boyle. 2018. Automatic Matching of Legacy Code to Heterogeneous APIs: An Idiomatic Approach. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems* (Williamsburg, VA, USA) *(ASPLOS '18)*. Association for Computing Machinery, New York, NY, USA, 139–153. doi:10.1145/3173162.3173182

[37] GROMACS. 2025. Installation guide for exotic configurations. https://manual.gromacs.org/2025.2/install-guide/exotic.html Accessed: 2025-08-25.

[38] Jeff Hammond. 2025. Mukautuva. https://github.com/jeffhammond/mukautuva Accessed: 2025-01-04.

[39] Jeff Hammond, Lisandro Dalcin, Erik Schnetter, Marc Pé Rache, Jean-Baptiste Besnard, Jed Brown, Gonzalo Brito Gadeschi, Simon Byrne, Joseph Schuchart, and Hui Zhou. 2023. MPI Application Binary Interface Standardization. In *Proceedings of the 30th European MPI Users' Group Meeting* (Bristol, United Kingdom) *(EuroMPI '23)*. Association for Computing Machinery, New York, NY, USA, Article 1, 12 pages. doi:10.1145/3615318.3615319

[40] Balvansh Heerekar, Cesar Philippidis, Ho-Ren Chuang, Pierre Olivier, Antonio Barbalace, and Binoy Ravindran. 2024. Offloading Datacenter Jobs to RISC-V Hardware for Improved Performance and Power Efficiency. In *Proceedings of the 17th ACM International Systems and Storage Conference* (Virtual, Israel) *(SYSTOR '24)*. Association for Computing Machinery, New York, NY, USA, 39–52. doi:10.1145/3688351.3689152

[41] M Heroux, J Willenbring, S Shende, C Coti, W Spear, J Peyralans, J Skutnik, and E Keever. 2020. E4S: Extreme-scale scientific software stack. LLVM 2011 European User Group Meeting. https://collegeville.github.io/CW20/WorkshopResources/WhitePapers/heroux-willenbring-shende-coti-spear-et-al-E4S.pdf Accessed: 2025-01-04.

[42] Torsten Hoefler, Marcin Copik, Pete Beckman, Andrew Jones, Ian Foster, Manish Parashar, Daniel Reed, Matthias Troyer, Thomas Schulthess, Daniel Ernst, and Jack Dongarra. 2024. XaaS: Acceleration as a Service to Enable Productive High-Performance Cloud Computing. *Computing in Science & Engineering* 26, 3 (2024), 40–51. doi:10.1109/MCSE.2024.3382154

[43] Torsten Hoefler, Ariel Hendel, and Duncan Roweth. 2022. The Convergence of Hyperscale Data Center and High-Performance Computing Networks. *Computer* 55, 7 (2022), 29–37. doi:10.1109/MC.2022.3158437

[44] Kenneth Hoste, Jens Timmerman, Andy Georges, and Stijn De Weirdt. 2012. EasyBuild: Building Software with Ease. In *2012 SC Companion: High Performance Computing, Networking Storage and Analysis*. 572–582. doi:10.1109/SC.Companion.2012.81

[45] Jin-Gu Kang. 2011. More Target Independent LLVM Bitcode. LLVM 2011 European User Group Meeting. https://llvm.org/devmtg/2011-09-16/ Accessed: 2025-01-04.

[46] Gregory M. Kurtzer, Vanessa Sochat, and Michael W. Bauer. 2017. Singularity: Scientific containers for mobility of compute. *PLOS ONE* 12, 5 (05 2017), 1–20. doi:10.1371/journal.pone.0177459

[47] C. Lattner and V. Adve. 2004. LLVM: a compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* 75–86. doi:10.1109/CGO.2004.1281665

[48] Edgar A. León, Marc Joos, Nathan Hanford, Adrien Cotte, Tony Delforge, François Diakhaté, Vincent Ducrot, Ian Karlin, and Marc Pérache. 2021. On-the-Fly, Robust Translation of MPI Libraries. In *2021 IEEE International Conference on Cluster Computing (CLUSTER)*. 504–515. doi:10.1109/Cluster48925.2021.00026

[49] Nelson F. Liu, Kevin Lin, John Hewitt, Ashwin Paranjape, Michele Bevilacqua, Fabio Petroni, and Percy Liang. 2024. Lost in the Middle: How Language Models Use Long Contexts. *Transactions of the Association for Computational Linguistics* 12 (2024), 157–173. doi:10.1162/tacl_a_00638

[50] LLVM. 2025. "Can I compile C or C++ code to platform-independent LLVM bitcode?". https://llvm.org/docs/FAQ.html#can-i-compile-c-or-c-code-to-platform-independent-llvm-bitcode Accessed: 2025-01-04.

[51] Alberto Madonna and Tomas Aliaga. 2022. Libfabric-based Injection Solutions for Portable Containerized MPI Applications. In *2022 IEEE/ACM 4th International Workshop on Containers and New Orchestration Paradigms for Isolated Environments in HPC (CANOPIE-HPC)*. 45–56. doi:10.1109/CANOPIE-HPC56864.2022.00010

[52] Stefan Schaefer Martin Lüscher. 2024. openQCD. https://luscher.web.cern.ch/luscher/openQCD/ Accessed: 2025-01-04.

[53] Pablo Antonio Martínez, Jackson Woodruff, Jordi Armengol-Estapé, Gregorio Bernabé, José Manuel García, and Michael F. P. O'Boyle. 2023. Matching Linear Algebra and Tensor Code to Specialized Hardware Accelerators. In *Proceedings of the 32nd ACM SIGPLAN International Conference on Compiler Construction* (Montréal, QC, Canada) *(CC 2023)*. Association for Computing Machinery, New York, NY, USA, 85–97. doi:10.1145/3578360.3580262

[54] Satoshi Matsuoka, Jens Domke, Mohamed Wahib, Aleksandr Drozd, and Torsten Hoefler. 2023. Myths and legends in high-performance computing. *The International Journal of High Performance Computing Applications* 37, 3-4 (2023), 245–259. doi:10.1177/10943420231166608 arXiv:https://doi.org/10.1177/10943420231166608

[55] Balthasar Reuter Michael Lange, Willem Deconinck. 2025. CLOUDSC. https://github.com/ecmwf-ifs/dwarf-p-cloudsc Accessed: 2025-01-04.

[56] MPICH. 2013. MPICH ABI Compatibility Initiative. https://www.mpich.org/abi/ Accessed: 2025-01-04.

[57] Daniel Nichols, Harshitha Menon, Todd Gamblin, and Abhinav Bhatele. 2024. A Probabilistic Approach To Selecting Build Configurations in Package Managers. In *SC24: International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–13. doi:10.1109/SC41406.2024.00090

[58] NVIDIA. 2025. The CUDA Compilation Trajectory. https://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc/#the-cuda-compilation-trajectory Accessed: 2025-08-25.

[59] NVIDIA. 2025. CUDA: Virtual Architectures. https://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc/index.html#virtual-architectures Accessed: 2025-01-04.

[60] NVIDIA. 2025. NVIDIA NGC Containers. https://www.nvidia.com/en-us/gpu-cloud/ Accessed: 2025-01-04.

[61] Open Containers Initiative (OCI). 2024. The OpenContainers Image Spec. https://specs.opencontainers.org/image-spec/ Accessed: 2025-01-04.

[62] OFIWG. 2024. Open Fabric Interfaces. https://github.com/ofiwg/libfabric/tree/v2.0.0 Accessed: 2025-01-04.

[63] S. John Pennycook, Jason D. Sewall, Douglas W. Jacobsen, Tom Deakin, and Simon McIntosh-Smith. 2021. Navigating Performance, Portability, and Productivity. *Computing in Science & Engineering* 23, 5 (2021), 28–38. doi:10.1109/MCSE.2021.3097276

[64] PRACE. 2024. Unified European Applications Benchmark Suite. https://repository.prace-ri.eu/git/UEABS/ueabs Accessed: 2025-01-04.

[65] Reid Priedhorsky and Tim Randles. 2017. Charliecloud: unprivileged containers for user-defined software stacks in HPC. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Denver, Colorado) *(SC '17)*. Association for Computing Machinery, New York, NY, USA, Article 36, 10 pages. doi:10.1145/3126908.3126925

[66] Howard P Pritchard, Thomas Naughton III, Amir Shehata, and David Bernholdt. 2023. Open MPI for HPE Cray EX Systems. In *Proceedings of the Cray User Group (CUG) Conference* (Helsinki, Finland). https://cug.org/proceedings/cug2023_proceedings/includes/files/pap140s2-file1.pdf

[67] Mitsuhisa Sato, Yutaka Ishikawa, Hirofumi Tomita, Yuetsu Kodama, Tetsuya Odajima, Miwako Tsuji, Hisashi Yashiro, Masaki Aoki, Naoyuki Shida, Ikuo Miyoshi, Kouichi Hirai, Atsushi Furuya, Akira Asato, Kuniki Morita, and Toshiyuki Shimizu. 2020. Co-Design for A64FX Manycore Processor and "Fugaku". In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–15. doi:10.1109/SC41405.2020.00051

[68] Erik Schnetter. 2022. *MPItrampoline*. doi:10.5281/zenodo.6174409

[69] Amir Shehata, Thomas Naughton, David E. Bernholdt, and Howard Pritchard. 2024. Bringing HPE Slingshot 11 support to Open MPI. *Concurrency and Computation: Practice and Experience* 36, 22 (2024), e8203. doi:10.1002/cpe.8203 arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/cpe.8203

[70] Nikolay A. Simakov, Matthew D. Jones, Thomas R. Furlani, Eva Siegmann, and Robert J. Harrison. 2024. First Impressions of the NVIDIA Grace CPU Superchip and NVIDIA Grace Hopper Superchip for Scientific Workloads. In *Proceedings of the International Conference on High Performance Computing in Asia-Pacific Region Workshops* (Nagoya, Japan) *(HPCAsia '24 Workshops)*. Association for Computing Machinery, New York, NY, USA, 36–44. doi:10.1145/3636480.3637097

[71] Spack. 2022. Announcing public binaries for Spack. https://spack.io/spack-binary-packages/ Accessed: 2025-01-04.

[72] Andreas Stathopoulos and James R. McCombs. 2010. PRIMME: preconditioned iterative multimethod eigensolver—methods and software description. *ACM Trans. Math. Softw.* 37, 2, Article 21 (April 2010), 30 pages. doi:10.1145/1731022.1731031

[73] Laurie Stephey, Shane Canon, Aditi Gaur, Daniel Fulton, and Andrew J. Younge. 2022. Scaling Podman on Perlmutter: Embracing a community-supported container ecosystem. In *2022 IEEE/ACM 4th International Workshop on Containers and New Orchestration Paradigms for Isolated Environments in HPC (CANOPIE-HPC)*. 25–35. doi:10.1109/CANOPIE-HPC56864.2022.00008

[74] Shinji Sumimoto, Toshihiro Hanawa, and Kengo Nakajima. 2024. MPI-Adapter2: An Automatic ABI Translation Library Builder for MPI Application Binary Portability. In *Proceedings of the International Conference on High Performance Computing in Asia-Pacific Region Workshops* (Nagoya, Japan) *(HPCAsia '24 Workshops)*. Association for Computing Machinery, New York, NY, USA, 63–68. doi:10.1145/3636480.3637219

[75] Dmitrii Tolmachev. 2023. VkFFT-A Performant, Cross-Platform and Open-Source GPU FFT Library. *IEEE Access* 11 (2023), 12039–12058. doi:10.1109/ACCESS.2023.3242240

[76] David Van Der Spoel, Erik Lindahl, Berk Hess, Gerrit Groenhof, Alan E. Mark, and Herman J. C. Berendsen. 2005. GROMACS: Fast, flexible, and free. *Journal of Computational Chemistry* 26, 16 (2005), 1701–1718. doi:10.1002/jcc.20291 arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/jcc.20291

[77] Field G. Van Zee and Robert A. van de Geijn. 2015. BLIS: A Framework for Rapidly Instantiating BLAS Functionality. *ACM Trans. Math. Software* 41, 3 (June 2015), 14:1–14:33. https://doi.acm.org/10.1145/2764454

[78] James M. Willenbring, Sameer S. Shende, and Todd Gamblin. 2024. Providing a Flexible and Comprehensive Software Stack Via Spack, an Extreme-Scale Scientific Software Stack, and Software Development Kits. *Computing in Science & Engineering* 26, 1 (2024), 20–30. doi:10.1109/MCSE.2024.3395016

[79] Tong Xing, Antonio Barbalace, Pierre Olivier, Mohamed L. Karaoui, Wei Wang, and Binoy Ravindran. 2022. H-Container: Enabling Heterogeneous-ISA Container Migration in Edge Computing. *ACM Trans. Comput. Syst.* 39, 1–4, Article 5 (July 2022), 36 pages. doi:10.1145/3524452

[80] Shulei Xu, Aamir Shafi, Hari Subramoni, and Dhabaleswar K. Panda. 2022. Arm meets Cloud: A Case Study of MPI Library Performance on AWS Arm-based HPC Cloud with Elastic Fabric Adapter. In *2022 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 449–456. doi:10.1109/IPDPSW55747.2022.00083

[81] Xiaoli Yan, Nathaniel Hudson, Hyun Park, Daniel Grzenda, J. Gregory Pauloski, Marcus Schwarting, Haochen Pan, Hassan Harb, Samuel Foreman, Chris Knight, Tom Gibbs, Kyle Chard, Santanu Chaudhuri, Emad Tajkhorshid, Ian Foster, Mohamad Moosavi, Logan Ward, and E. A. Huerta. 2025. MOFA: Discovering Materials for Carbon Capture with a GenAI- and Simulation-Based Workflow. arXiv:2501.10651 [cs.DC] https://arxiv.org/abs/2501.10651

[82] Günther Zängl, Daniel Reinert, Pilar Rípodas, and Michael Baldauf. 2015. The ICON (ICOsahedral Non-hydrostatic) Modelling Framework of DWD and MPI-M: Description of the Non-Hydrostatic Dynamical Core. *Quarterly Journal of the Royal Meteorological Society* 141, 687 (2015), 563–579. doi:10.1002/qj.2378 arXiv:https://rmets.onlinelibrary.wiley.com/doi/pdf/10.1002/qj.2378

[83] Nannan Zhao, Vasily Tarasov, Hadeel Albahar, Ali Anwar, Lukas Rupprecht, Dimitrios Skourtis, Arnab K. Paul, Keren Chen, and Ali R. Butt. 2021. Large-Scale Analysis of Docker Images and Performance Implications for Container Storage Systems. *IEEE Transactions on Parallel and Distributed Systems* 32, 4 (2021), 918–930. doi:10.1109/TPDS.2020.3034517

[84] Alexandros Nikolaos Ziogas, Timo Schneider, Tal Ben-Nun, Alexandru Calotoiu, Tiziano De Matteis, Johannes de Fine Licht, Luca Lavarini, and Torsten Hoefler. 2021. Productivity, portability, performance: data-centric Python. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (St. Louis, Missouri) *(SC '21)*. Association for Computing Machinery, New York, NY, USA, Article 95, 13 pages. doi:10.1145/3458817.3476176

A LLM Prompt to Discover Specialization Points

I will share a build file, and I would like you to identify all the specialization points for an HPC program and the associated build flags used to enable those features during the build process. Please pay close attention to:

- Comments and messages within the build file, as they often reveal the necessary flags.
- Functions like `gmx_option_multichoice`, which specify build flags and options for libraries.
- Ensure libraries are correctly matched to their corresponding build flags based on these functions.
- Option Commands: In some projects, build flags are provided in `option` commands. Look at these commands to extract the build flags correctly.
- Full Build Flags Extraction: Ensure that the full build flags are extracted, not just partial representations. For instance, if a flag is defined as `-DQE_ENABLE_CUDA=ON`, extract the entire flag with its value.
- Distinguish Between Build Flags and Preprocessor Definitions: Do not confuse preprocessor definitions (e.g., `__CUDA`, `__MPI`) with actual build flags (e.g., `-DQE_ENABLE_CUDA`, `-DQE_ENABLE_MPI`). Extract only the build flags that are explicitly defined in the build configuration.
- Portability Frameworks: Some build systems use portability frameworks like Kokkos. Pay attention to build flags like `-DKokkos_ENABLE_OPENMP`, `-DKokkos_ENABLE_PTHREAD`, and `-DKokkos_ENABLE_CUDA`.
- Vectorization Libraries: Some projects use external vectorization libraries like V4. Look for build flags such as `-DUSE_V4_ALTIVEC`, `-DUSE_V4_PORTABLE`, and `-DUSE_V4_SSE`.

Key Instructions:

1. Analyze Functions for Build Flags:

- Look for functions such as `gmx_option_multichoice`, `gmx_dependent_option`, and `option` commands that define build flags and their corresponding options.
- For example, the flag `-DGMX_FFT_LIBRARY` has options like `fftw3`, `mkl`, and `fftpack[built-in]`.
- Another example is `-DGMX_GPU_FFT_LIBRARY` with options like `cuFFT`, `VkFFT`, `clFFT`, `rocFFT`, and `MKL`. Match the library names with the build flags from these function calls.
- Additionally, the flag `-DGMX_GPU` has options like `CUDA`, `OpenCL`, `SYCL`, and `HIP`. Ensure these GPU backends are matched correctly to their corresponding flags.
- For Kokkos, look for flags like `-DKokkos_ENABLE_OPENMP`, `-DKokkos_ENABLE_PTHREAD`, and `-DKokkos_ENABLE_CUDA`.

2. Match Libraries to Flags:

- Libraries should be matched to their respective build flags based on these function definitions.
- For example:
  - If `GMX_FFT_LIBRARY` is set to `fftw3`, the build flag is `-DGMX_FFT_LIBRARY=fftw3`.
  - If `GMX_GPU_FFT_LIBRARY` is set to `cuFFT`, the build flag is `-DGMX_GPU_FFT_LIBRARY=cuFFT`.
  - For vectorization, look for flags like `-DUSE_V4_ALTIVEC`, `-DUSE_V4_PORTABLE`, and `-DUSE_V4_SSE`.

3. Match GPU Backends to `GMX_GPU`:

- Ensure that GPU backends (CUDA, OpenCL, SYCL, HIP, METAL) are matched to the `GMX_GPU` flag based on the `gmx_option_multichoice` function.
- For example:
  - If `GMX_GPU` is set to `CUDA`, the build flag is `-DGMX_GPU=CUDA`.
  - If `GMX_GPU` is set to `SYCL`, the build flag is `-DGMX_GPU=SYCL`.

- For Quantum ESPRESSO: Ensure that GPU backends like CUDA are matched to their corresponding build flags, such as `-DQE_ENABLE_CUDA`, instead of preprocessor definitions like `__CUDA`.

4. Consider Default Values and Dependencies:

- Identify the default libraries and how they are conditionally set. For example:
  - `GMX_FFT_LIBRARY_DEFAULT` is `mkl` if `GMX_INTEL_LLVM` is set, otherwise `fftw3`.
  - The GPU FFT library defaults vary based on the GPU backend (e.g., `cuFFT` for CUDA, `VkFFT` for OpenCL).

5. Special Attention to FFT Libraries:

- Look for all flags related to FFT libraries like:
  - `-DGMX_FFT_LIBRARY`
  - `-DGMX_FFT_LIBRARY_DEFAULT`
  - `-DGMX_GPU_FFT_LIBRARY`
- Extract not only the flag but also the corresponding library it enables (e.g., `fftw3`, `mkl`, `cuFFT`).

6. Include Relevant Build Flags:

- Do not include preprocessor definitions generated internally. Only include build flags explicitly defined in the file.
- Ensure that each build flag is extracted with its full definition, including any assigned values.

Specifically, identify the following:

- Does the build system support GPU builds? (For example, the presence of a flag like `BUILD_GPU` indicates GPU support.)
- What GPU backends does it support (e.g. CUDA, HIP, SYCL, OpenCL)? Are these backends enabled or disabled by default? What is their minimum version, if specified?
- What parallel programming libraries (e.g. MPI, OpenMP, Pthread, thread-MPI, OpenACC) are supported, and are they enabled or disabled by default? What is their minimum version, if specified?
- What linear algebra libraries (e.g. BLAS, LAPACK, SCALAPACK, MKL/oneMKL) does the build system use, and under which conditions? What are the default libraries used in the build process?
- What Fast Fourier Transform libraries (e.g. FFTW, fftpack, MKL/oneMKL, cuFFT, vkFFT, clFFT, rocFFT) does the build system use? What library is built-in? Are there specific dependencies for the library to be used (for example, they must be used with a certain GPU backend or parallel library)? Are they enabled or disabled by default? For the build-flags, look for flags defined via `gmx_option_multichoice` such as `-DGMX_FFT_LIBRARY`, `-DGMX_FFT_LIBRARY_DEFAULT`, `-DGMX_GPU_FFT_LIBRARY`.
- What other external libraries are used, what versions are specified, and what are their dependencies? List all external libraries and the conditions for their use.
- What other compiler flags are supported?
- Are there build flags used to optimize the performance of the program? (e.g., auto-tuning, team reduction, hierarchical parallelism, accumulators, qunatization, batch size, force use of custom matrix multiplications)
- Which compilers are supported, and what are the minimum versions required?
- What architectures does the system support?
- Does it support SIMD vectorization, and what vectorization levels are supported? find the build flag for each supported vectorization level.
- What is the minimum version required for the build system? Is it a CMake or Make build system?
- Are there any libraries that require internal builds? If so, name them and provide the build flags (e.g. `-DGMX_BUILD_OWN_FFTW`, `DBUILD_INTERNAL_KOKKOS`).

The answer should be provided as a JSON structure adhering to the specified schema, with keys including `gpu_build`, `gpu_backends`, `parallel_programming_libraries`, `linear_algebra_libraries`, `fft_libraries`, `other_external_libraries`, `optimization_build_flags`, `compiler_flags`, `compilers`, `architectures`, `simd_vectorization`, and `build_system`, `internal_build`. The `build_flag` value for each feature should be the flag itself (e.g., `-DGMX_SIMD`, `-DGMX_GPU`, `-DQE_ENABLE_CUDA`, `-DQE_ENABLE_MPI`, `-DKokkos_ENABLE_OPENMP`, `-DUSE_V4_ALTIVEC`) without any surrounding text. Do not include any preprocessor definitions that are generated internally. The response must be a valid JSON structure; do not include any introductory or explanatory text.

Here is the build file: `{file_content}`

JSON output schema. Use this JSON schema to format your response but do not include it in the output: `{schema}`

B JSON Schema for Specialization Points

```
{
    "$schema": "http://json-schema.org/draft-07/schema#",
    "type": "object",
    "properties": {
      "gpu_build": {
        "type": "object",
        "properties": {
          "value": {
            "type": "boolean"
          },
          "build_flag": {
            "type": ["string", "null"]
          }
        },
        "required": ["value", "build_flag"]
      },
      "gpu_backends": {
        "type": "object",
        "additionalProperties": {
          "type": "object",
          "properties": {
            "used_as_default": {
              "type": "boolean"
            },
            "build_flag": {
              "type": ["string", "null"]
            },
            "minimum_version": {
              "type": ["string", "null"]
            }
          },
          "required": ["used_as_default", "build_flag",
          ↪ "minimum_version"]
        }
      },
      "parallel_programming_libraries": {
```

```
      "type": "object",
      "additionalProperties": {
        "type": "object",
        "properties": {
          "used_as_default": {
            "type": "boolean"
          },
          "build_flag": {
            "type": ["string", "null"]
          },
          "minimum_version": {
            "type": ["string", "null"]
          }
        },
        "required": ["used_as_default", "build_flag",
        ↪ "minimum_version"]
      }
    },
    "linear_algebra_libraries": {
      "type": "object",
      "additionalProperties": {
        "type": "object",
        "properties": {
          "used_as_default": {
            "type": "boolean"
          },
          "build_flag": {
            "type": ["string", "null"]
          },
          "condition": {
            "type": ["string", "null"]
          }
        },
        "required": ["used_as_default", "build_flag",
        ↪ "condition"]
      }
    },
    "FFT_libraries": {
      "type": "object",
      "additionalProperties": {
        "type": "object",
        "properties": {
          "built-in": {
            "type": "boolean"
          },
          "used_as_default": {
            "type": "boolean"
          },
          "dependencies": {
            "type": ["string", "null"]
          },
          "build_flag": {
            "type": ["string", "null"]
          }
        },
        "required": ["used_as_default", "condition",
        ↪ "build_flag"]
      }
    },
    "other_external_libraries": {
      "type": "object",
      "additionalProperties": {
        "type": "object",
        "properties": {
          "version": {
            "type": "string"
          },
          "used_as_default": {
            "type": "boolean"
          },
          "conditions": {
            "type": "string"
          },
          "build_flag": {
            "type": ["string", "null"]
          }
        },
        "required": ["version", "used_as_default",
        ↪ "conditions", "build_flag"]
      }
    },
    "compiler_flags": {
      "type": "array",
      "items": {
        "type": "string"
      }
    },
    "optimization_build_flags": {
      "type": "array",
      "items": {
        "type": "string"
      }
    },
    "compilers": {
      "type": "object",
      "additionalProperties": {
        "type": "object",
        "properties": {
          "minimum_version": {
            "type": "string"
          }
        },
        "required": ["minimum_version"]
      }
    },
    "architectures": {
      "type": "array",
      "items": {
        "type": "string"
      }
    },
    "simd_vectorization": {
      "type": "object",
      "additionalProperties": {
        "type": "object",
        "properties": {
          "build_flag": {
            "type": ["string", "null"]
          },
          "default": {
            "type": "boolean"
          }
        },
        "required": ["build_flag", "default"]
      }
    },
      "build_system": {
        "type": "object",
        "properties": {
          "type": {
            "type": "string",
            "enum": ["cmake", "make", "undetermined"]
          },
          "minimum_version": {
            "type": "string"
          }
        },
        "required": ["type", "minimum_version"]
      },
      "internal_build": {
      "type": "object",
      "properties": {
        "library_name": {
          "type": "string"
        },
        "build_flag": {
          "type": ["string", "null"]
        }
      },
      "required": ["library_name", "build_flag"]
  }
  },
  "required": [
    "gpu_build",
    "gpu_backends",
```

```
      "parallel_programming_libraries",
      "linear_algebra_libraries",
      "FFT_libraries",
      "other_external_libraries",
      "compiler_flags",
      "optimization_build_flags",
      "compilers",
      "architectures",
      "simd_vectorization",
       "build_system",
       "internal_build"
    ],
    "additionalProperties": false
}
```