Other abstraction layers

Lars Pastewka

Table of contents

Throughout this class we use Kokkos as our hardware abstraction layer. Kokkos is a good fit for our use case – performance-portable C++ code for scientific simulations – but it is far from the only option. The ecosystem of tools for programming accelerators is large and growing rapidly, driven by both the HPC community and the machine learning community.

This chapter surveys the most important alternatives. Understanding the landscape helps you choose the right tool for a given problem and recognize the common ideas that underlie all of them.

The baseline: Kokkos

As a point of reference, here is a simple vector addition written in Kokkos (see the Kokkos lecture for details). All frameworks below are illustrated with the same operation:

```
Kokkos::View<double*> a("a", N), b("b", N), c("c", N);

Kokkos::parallel_for("vector_add", N, KOKKOS_LAMBDA(const int i) {
    c(i) = a(i) + b(i);
});
```

The `Views` live in the default memory space (GPU memory when a device backend is enabled), and the `parallel_for` dispatches one iteration per GPU thread.

Compiler-based approaches

OpenMP target offloading

OpenMP is a directive-based programming model that has been part of the HPC ecosystem since 1997. Originally designed for shared-memory parallelism on CPUs (using `#pragma omp parallel for`), OpenMP added GPU offloading capabilities in version 4.0 (2013) through `target` directives.

The key idea is that the programmer annotates existing code with compiler directives (pragmas), and the compiler generates the appropriate GPU code:

```
#pragma omp target teams distribute parallel for map(to: a[0:N], b[0:N]) map(from: c[0:N])
for (int i = 0; i < N; i++) {
    c[i] = a[i] + b[i];
}
```

The `target` directive moves execution to the GPU. The `map` clauses specify data transfers: `map(to:...)` copies data to the device, `map(from:...)` copies results back. The `teams distribute parallel for` clauses control the thread hierarchy (teams map to thread blocks, threads within a team map to GPU threads).

Advantages:

- Incremental adoption: existing CPU code can be offloaded by adding pragmas, without rewriting.
- Standardized: supported by GCC (12+), Clang/LLVM, NVIDIA's `nvc++`, and AMD's `amdclang++`.
- Single source: the same code compiles for CPU (ignoring pragmas) and GPU.

Limitations:

- Performance portability is harder to achieve than with Kokkos. The programmer must manually manage data movement and thread mapping, and optimal pragma configurations differ between vendors.
- Compiler support for GPU offloading is still maturing; not all OpenMP features are supported on all backends.
- Debugging and profiling are more difficult than with vendor-native tools.

SYCL

SYCL is a C++ programming model standardized by the Khronos Group (the same organization behind OpenGL and Vulkan). It provides a single-source C++ approach to programming heterogeneous systems.

```
sycl::queue q;
sycl::buffer<double> buf_a(a.data(), N);
sycl::buffer<double> buf_b(b.data(), N);
sycl::buffer<double> buf_c(c.data(), N);

q.submit([&](sycl::handler& h) {
    auto acc_a = buf_a.get_access<sycl::access::mode::read>(h);
    auto acc_b = buf_b.get_access<sycl::access::mode::read>(h);
    auto acc_c = buf_c.get_access<sycl::access::mode::write>(h);
    h.parallel_for(N, [=](sycl::id<1> i) {
        acc_c[i] = acc_a[i] + acc_b[i];
    });
});
```

SYCL uses *accessors* to manage data dependencies and *queues* for task submission. Intel's oneAPI implementation (compile with `icpx -fsycl`; the separate `dpcpp` driver has been retired) is the most mature, targeting Intel GPUs, CPUs, and FPGAs. There are also open-source implementations that target NVIDIA and AMD GPUs (e.g. AdaptiveCpp, formerly hipSYCL).

SYCL is conceptually similar to Kokkos but uses standard C++ constructs rather than macros like `KOKKOS_LAMBDA`, making it more familiar to modern C++ programmers. However, its ecosystem is smaller and less battle-tested in the HPC community than Kokkos or OpenMP.

C++ abstraction libraries

RAJA

RAJA is a C++ abstraction layer developed at Lawrence Livermore National Laboratory (LLNL). Like Kokkos, it provides execution policies and portable parallel loop abstractions:

```
RAJA::forall<RAJA::cuda_exec<256>>(
    RAJA::RangeSegment(0, N),
    [=] RAJA_DEVICE(int i) {
        c[i] = a[i] + b[i];
    });
```

The template parameter `RAJA::cuda_exec<256>` specifies the execution backend (CUDA with 256 threads per block). Switching to `RAJA::omp_parallel_for_exec` runs the same loop on the CPU with OpenMP. RAJA also provides portable reductions, scans, and atomic operations.

RAJA and Kokkos share the same fundamental design: parallel dispatch through execution policies and data management through abstraction layers. RAJA differs in that it does not manage memory layout or provide its own multi-dimensional array type – it focuses on loop execution and leaves data management to the user or to companion libraries like CHAI (for managed arrays) and Umpire (for memory allocation). Kokkos bundles all of these concerns into a single framework (`View`, memory spaces, execution spaces).

RAJA is widely used in LLNL production codes and is a strong alternative to Kokkos, particularly when integrating with existing codebases that already manage their own data structures.

Python frameworks

The machine learning revolution has produced a set of Python frameworks that provide GPU acceleration through a very different programming model than the C++ approaches above. Instead of explicit kernel launches and memory management, these frameworks operate on *tensors* (multi-dimensional arrays) and use *just-in-time (JIT) compilation* to generate GPU code from high-level Python expressions.

JAX

JAX is a numerical computing library developed by Google that provides a NumPy-compatible API with automatic GPU acceleration, JIT compilation, and automatic differentiation.

```
import jax
import jax.numpy as jnp

@jax.jit
def vector_add(a, b):
    return a + b

a = jnp.ones(1_000_000)
b = jnp.ones(1_000_000)
c = vector_add(a, b)  # Runs on GPU if available
```

The `@jax.jit` decorator triggers JIT compilation via XLA (Accelerated Linear Algebra), a compiler originally developed for TensorFlow. XLA compiles the Python function into optimized GPU (or CPU/TPU) code at the first call, and subsequent calls reuse the compiled kernel.

JAX's key features:

- **Automatic differentiation**: `jax.grad(f)` returns a function that computes the gradient of `f`. This works through arbitrary Python control flow and supports forward-mode, reverse-mode, and higher-order derivatives.
- **Vectorization**: `jax.vmap(f)` automatically vectorizes a function over a batch dimension, eliminating the need for manual batching.
- **Parallelization**: computations are distributed across multiple GPUs or TPU cores by combining `jax.jit` with sharding annotations, or with `shard_map` for explicit per-device programming. (The older `jax.pmap` transformation is deprecated.)
- **Functional programming model**: JAX functions must be pure (no side effects), which enables aggressive compiler optimizations.

JAX is particularly popular in scientific machine learning, molecular dynamics (e.g. JAX-MD), and differentiable physics simulations.

PyTorch

PyTorch is the dominant framework for deep learning research, developed by Meta. While its primary focus is neural network training, its tensor operations and GPU acceleration are general enough for scientific computing.

```
import torch

a = torch.ones(1_000_000, device='cuda')
b = torch.ones(1_000_000, device='cuda')
c = a + b  # Runs on GPU
```

PyTorch uses an *eager execution* model by default: operations execute immediately as Python evaluates them, making debugging straightforward. For performance, `torch.compile` (introduced in PyTorch 2.0) traces and compiles Python code into optimized kernels via the Triton compiler or the TorchInductor backend.

PyTorch provides automatic differentiation through its `autograd` engine, which records operations on tensors and can compute gradients via backpropagation:

```
x = torch.tensor([2.0], requires_grad=True, device='cuda')
y = x**2 + 3*x
y.backward()
print(x.grad)  # tensor([7.]) = 2*x + 3 evaluated at x=2
```

PyTorch targets NVIDIA GPUs via CUDA and AMD GPUs via ROCm. Intel GPU support is available through the `intel-extension-for-pytorch` package.

Taichi

Taichi is a domain-specific language embedded in Python, designed for high-performance spatial computing – simulations involving particles, grids, and meshes. It is particularly popular in computer graphics and computational physics.

```
import taichi as ti

ti.init(arch=ti.gpu)

N = 1_000_000

a = ti.field(dtype=ti.f64, shape=N)
b = ti.field(dtype=ti.f64, shape=N)
c = ti.field(dtype=ti.f64, shape=N)

@ti.kernel
def vector_add():
    for i in range(N):
        c[i] = a[i] + b[i]

vector_add()  # Compiled and executed on GPU
```

The `@ti.kernel` decorator triggers compilation of the Python function into GPU code. Taichi uses its own compiler infrastructure built on top of LLVM (see below) to generate CUDA, Vulkan, Metal, or OpenGL code from the Python source.

Taichi's distinguishing features are its *sparse data structures* (for simulations with spatially varying resolution) and its *automatic differentiation* support, which makes it suitable for differentiable

physics simulations. It also supports complex data layouts (struct-of-arrays, array-of-structs) through its `ti.field` abstraction.

Julia

Julia takes a fundamentally different approach: rather than providing a library or framework within an existing language, Julia is a *language designed from the ground up* for high-performance numerical computing. It compiles to native code via LLVM and achieves performance comparable to C/C++ while providing a high-level, dynamic syntax similar to Python or MATLAB.

GPU programming in Julia is provided through packages:

- CUDA.jl: native Julia interface to NVIDIA GPUs
- AMDGPU.jl: AMD GPU support via ROCm
- KernelAbstractions.jl: a hardware abstraction layer conceptually similar to Kokkos

```julia
using KernelAbstractions, CUDA

@kernel function vector_add_kernel!(c, a, b)
    i = @index(Global)
    c[i] = a[i] + b[i]
end

a = CuArray(ones(Float64, 1_000_000))
b = CuArray(ones(Float64, 1_000_000))
c = similar(a)

vector_add_kernel!(CUDABackend())(c, a, b; ndrange=length(a))
```

`KernelAbstractions.jl` provides the same portability as Kokkos: the kernel is written once and dispatched to CUDA, ROCm, or CPU by changing the backend. Julia's JIT compilation through LLVM means that GPU kernels are generated at runtime from the same source code, without a separate compilation step.

Julia also has strong support for automatic differentiation through packages like Enzyme.jl (which operates at the LLVM level and can differentiate GPU kernels) and Zygote.jl (source-to-source AD).

The role of LLVM

Several of the frameworks above – Taichi, Julia, and parts of JAX and PyTorch – rely on LLVM as their compilation infrastructure. Understanding LLVM's role clarifies why so many different languages can target the same GPU hardware.

LLVM is a compiler framework that provides a language-independent *intermediate representation* (IR). The compilation pipeline works in stages, as shown in Figure 1.

The front end (language-specific) translates source code into LLVM IR. The middle end applies optimizations (constant folding, loop unrolling, vectorization) that are independent of both the source language and the target hardware. The back end translates the optimized IR into machine code for a specific target.

For GPU targets, LLVM includes back ends for:

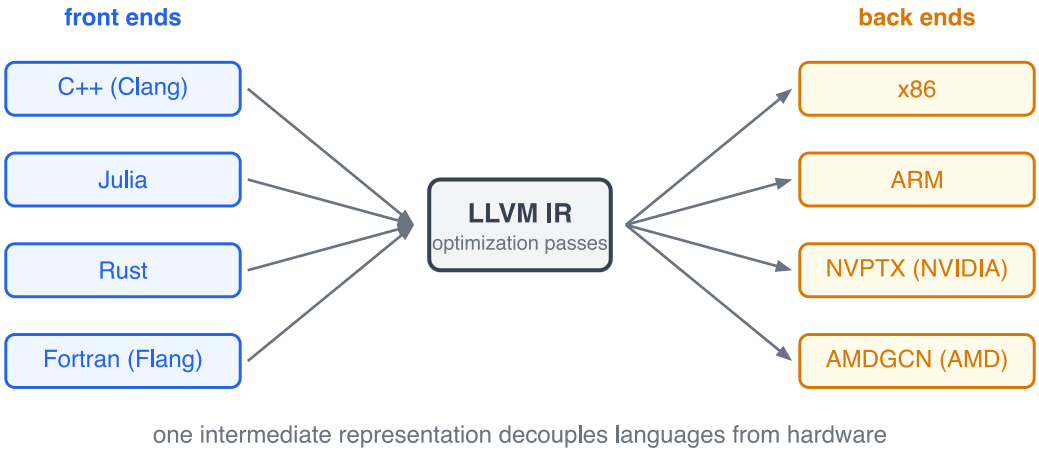

Figure 1: The LLVM hourglass: language-specific front ends translate many source languages into one common intermediate representation, where optimizations happen once, before target-specific back ends emit machine code for x86, ARM, NVIDIA PTX, AMD GCN and other hardware.

- **NVIDIA GPUs**: generates PTX (Parallel Thread Execution) assembly, which is then compiled to machine code by NVIDIA's `ptxas` assembler.
- **AMD GPUs**: generates GCN/RDNA machine code directly.

This architecture is what allows Julia to compile GPU kernels from a dynamic language, Taichi to generate CUDA code from Python, and Clang/LLVM to support OpenMP target offloading – they all share the same LLVM back end for GPU code generation.

NVIDIA's own compiler `nvcc` uses a different pipeline (based on the Edison Design Group C++ front end and NVIDIA's proprietary `ptxas`), but the trend in the ecosystem is toward LLVM-based toolchains. AMD's ROCm stack is entirely LLVM-based, and Intel's oneAPI/SYCL compiler is also built on LLVM.

Automatic differentiation

Several of the frameworks above (JAX, PyTorch, Taichi, Julia) provide *automatic differentiation* (AD) as a core feature. AD is the ability to compute exact derivatives of programs with respect to their inputs, without deriving and implementing the derivatives by hand.

Why AD matters for GPU computing

AD is not GPU-specific – it is a general technique that works on any computation. However, the combination of AD with GPU acceleration is transformative for two fields:

- **Machine learning**: training neural networks requires computing gradients of a loss function with respect to millions of parameters. AD (specifically reverse-mode AD, also called backpropagation) makes this tractable, and GPUs make it fast.
- **Differentiable physics**: simulations where gradients of the output with respect to input parameters are needed – for optimization, inverse problems, or sensitivity analysis. For example, optimizing the shape of an airfoil by differentiating through a fluid dynamics solver. Applied to this course's project, differentiating through the Lattice Boltzmann

solver would enable gradient-based design – for instance, optimizing an obstacle shape or a boundary condition directly from the simulated flow field.

How AD works

There are two modes:

- **Forward mode**: propagates derivatives alongside the computation. For a function $f : \mathbb{R}^n \to \mathbb{R}^m$, one forward pass computes $f(x)$ and $\partial f/\partial x_i$ for a single input direction i. Efficient when n is small (few inputs, many outputs).
- **Reverse mode**: records the computation in a *tape* (computation graph), then traverses it backward to accumulate gradients. One backward pass computes $\partial f/\partial x_i$ for *all* inputs i simultaneously. Efficient when m is small (many inputs, few outputs) – which is exactly the case in machine learning (millions of parameters, one scalar loss).

On GPUs, both the forward computation and the backward gradient computation can be parallelized. Frameworks like JAX and PyTorch handle this transparently: the user writes the forward computation, and the framework generates and executes the backward pass on the GPU automatically.

AD at the compiler level

A notable recent development is Enzyme, an LLVM plugin that performs AD at the LLVM IR level. Because it operates on the compiler's intermediate representation rather than on source code, Enzyme can differentiate programs written in *any* LLVM-based language (C, C++, Fortran, Julia, Rust) and can differentiate through GPU kernels (CUDA, ROCm). This is a fundamentally different approach from the source-level or operator-overloading AD used by PyTorch and JAX, and it can achieve better performance because the compiler can optimize the forward and backward passes jointly.

Comparison

The following table summarizes the key characteristics of each framework:

Framework	Language	GPU backends	Memory management	AD	Compilation
Kokkos	C++	CUDA, HIP, SYCL	Views + memory spaces	No	Ahead-of-time
RAJA	C++	CUDA, HIP, OpenMP	External (CHAI/Umpire)	No	Ahead-of-time
OpenMP	C/C++/Fortran	CUDA, HIP, oneAPI	Pragmas (`map`)	No	Ahead-of-time
SYCL	C++	Intel, CUDA, HIP	Buffers + accessors	No	Ahead-of-time
JAX	Python	CUDA, ROCm, TPU	Automatic (XLA)	Yes	JIT (XLA)

Framework	Language	GPU backends	Memory management	AD	Compilation
PyTorch	Python	CUDA, ROCm	Automatic (tensors)	Yes	Eager + JIT
Taichi	Python	CUDA, Vulkan, Metal	Fields	Yes	JIT (LLVM)
Julia	Julia	CUDA, ROCm	Arrays	Yes (Enzyme)	JIT (LLVM)

The C++ frameworks (Kokkos, RAJA, OpenMP, SYCL) prioritize performance control and are dominant in traditional HPC. The Python frameworks (JAX, PyTorch, Taichi) prioritize developer productivity and automatic differentiation, and are dominant in machine learning and increasingly used in scientific computing. Julia occupies a middle ground, offering both high-level productivity and low-level performance control.

For this class, Kokkos is the right choice because it provides explicit control over memory layout and execution policy – exactly the concepts covered in the GPU architecture and memory performance lectures. However, for projects where development speed, automatic differentiation, or integration with machine learning workflows is more important than fine-grained hardware control, the Python frameworks or Julia may be more appropriate.