OpenCilk: A Modular and Extensible Software Infrastructure for Fast Task-Parallel Code

Tao B. Schardl
NUWEST
January 18, 2024
Albuquerque, NM

Parallel programming in different languages

Different languages introduce different constructs for parallelism.

Example Julia code

```
Z .= a .* X .+ Y
```

Implicitly parallel operators

Example Julia @threads code

```
function saxpy_threads(Z, X, Y, a)
  Tasks.@threads for I in eachindex(Z, Y, X)
    Z[I] = a*X[I] + Y[I]
  end
  Z
end
```

Parallel loop of Julia tasks

Example Halide code

```
C.update(0)
  .fuse(i0, j0, tile)
  .parallel(tile);
```

Parallel scheduling command

Example CUDA.jl code

```
function saxpy_cuda(Z, a, X, Y)
  i = (blockIdx().x - 1) * blockDim().x + threadIdx().x
  if i <= length(Z)
    Z[i] = a * X[i] + Y[i]
  end
  return nothing
end
```

GPU kernel

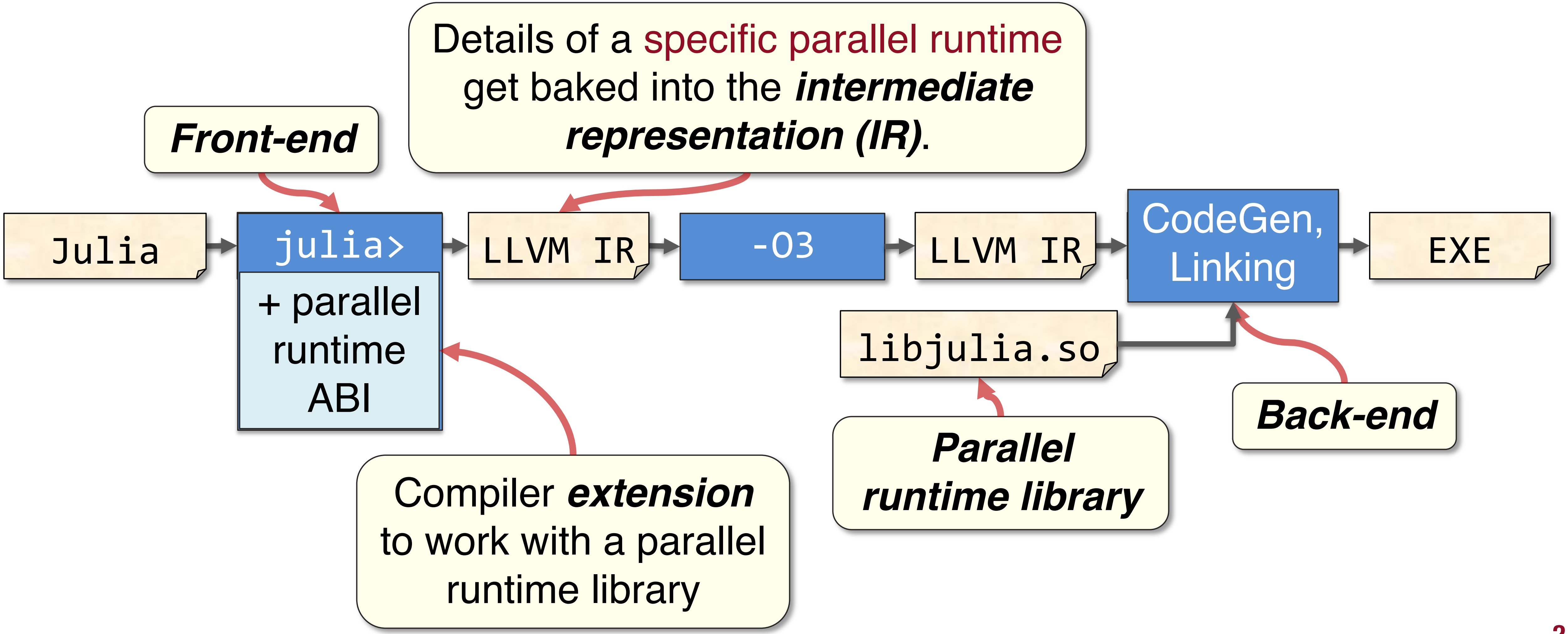
Parallel-loop library routine

Example Kokkos code

```
View<ForceType *> atomForces("atomForces", numberOfAtoms);
View<AtomDataType> data("data", size);
Kokkos::parallel_for(numberOfAtoms,
    KOKKOS_LAMBDA(const size_t atomIndex) {
      atomForces(atomIndex) = calculateForce(data);
});
```

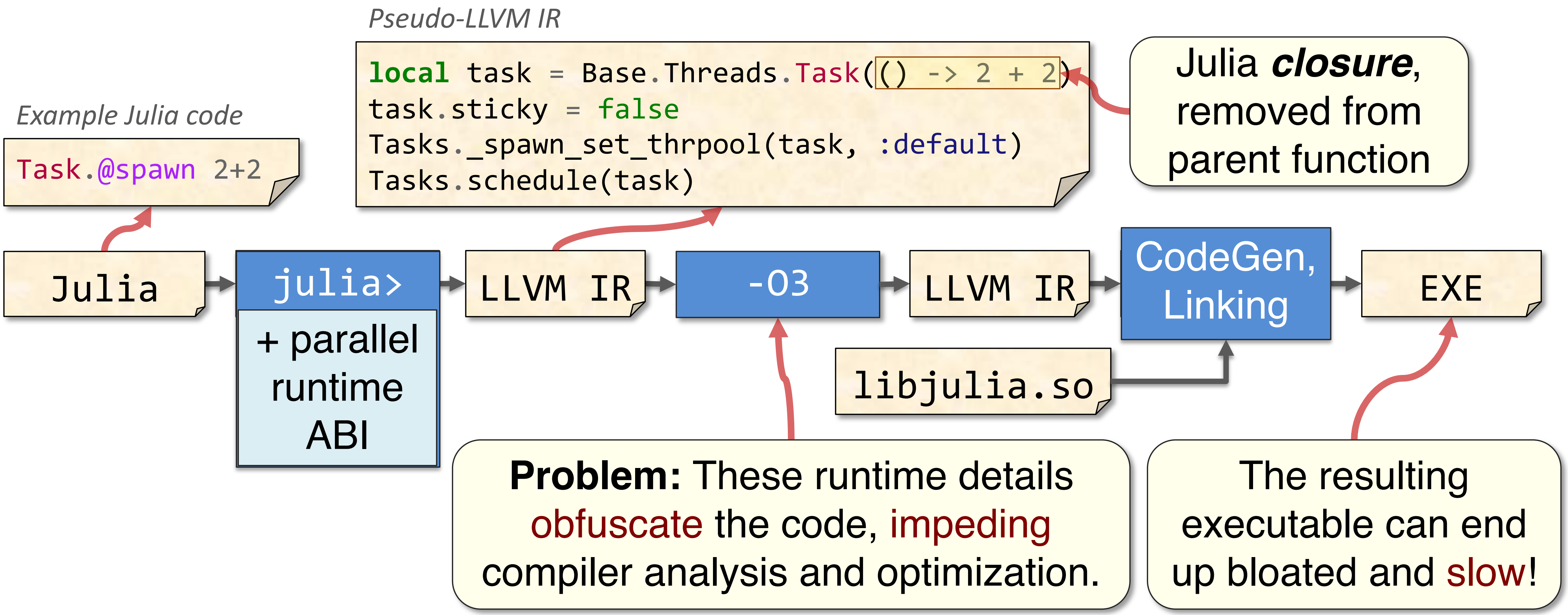
Traditional compiler design for parallelism

Traditionally, compiler internals assume a sequential, flat-memory machine and lack a deep understanding of parallelism.



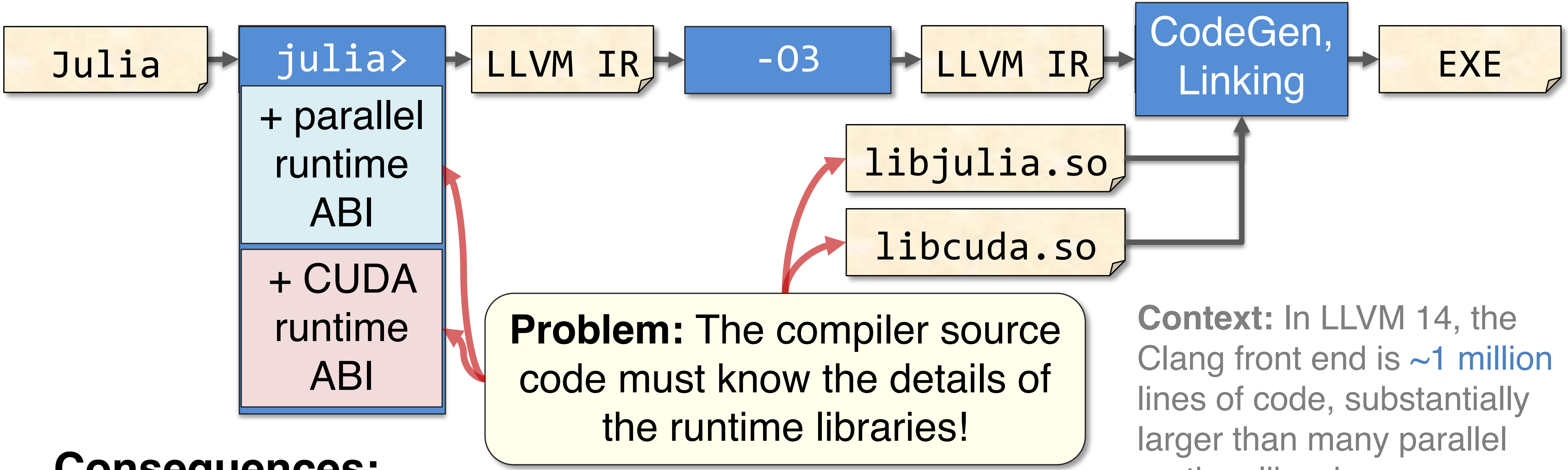
Problem: Parallel performance

This approach to compiling parallel code is bad for performance.



Problem: Modifying and extending runtimes

Modifying or extending a parallel runtime ABI requires substantial engineering effort to modify both the compiler and runtime library.



Context: In LLVM 14, the Clang front end is ~1 million lines of code, substantially larger than many parallel runtime libraries.

Consequences:

- It is hard to modify runtimes to make them compose.
- It is hard to add support for new parallel hardware.

Previous work: Tapir [SML17]

Previously, we developed ***Tapir***, a compiler intermediate representation that allows the compiler to understand task parallelism.

Tapir/LLVM compiler

Tapir adds three instructions to LLVM IR that encode recursive fork-join parallelism.

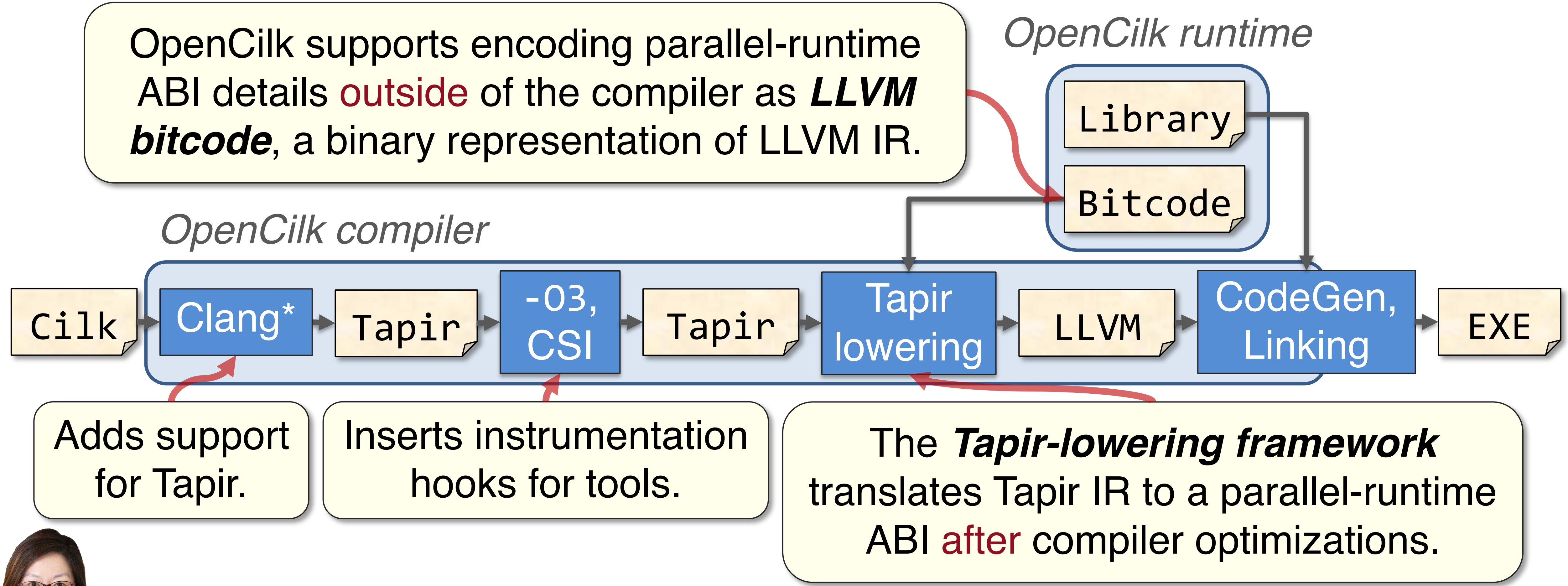
With only minor changes, LLVM's existing optimizations and analyses work on parallel code.

Compiling with Tapir significantly improves the performance of task-parallel programs.

[SML17] Schardl, Moses, Leiserson. Tapir: Embedding Fork-Join Parallelism into LLVM's Intermediate Representation. In *PPoPP*, 2017.

OpenCilk system architecture [SL23]

OpenCilk uses LLVM and Tapir to make it easy to modify and extend the compiler and runtime to different parallel programming platforms.

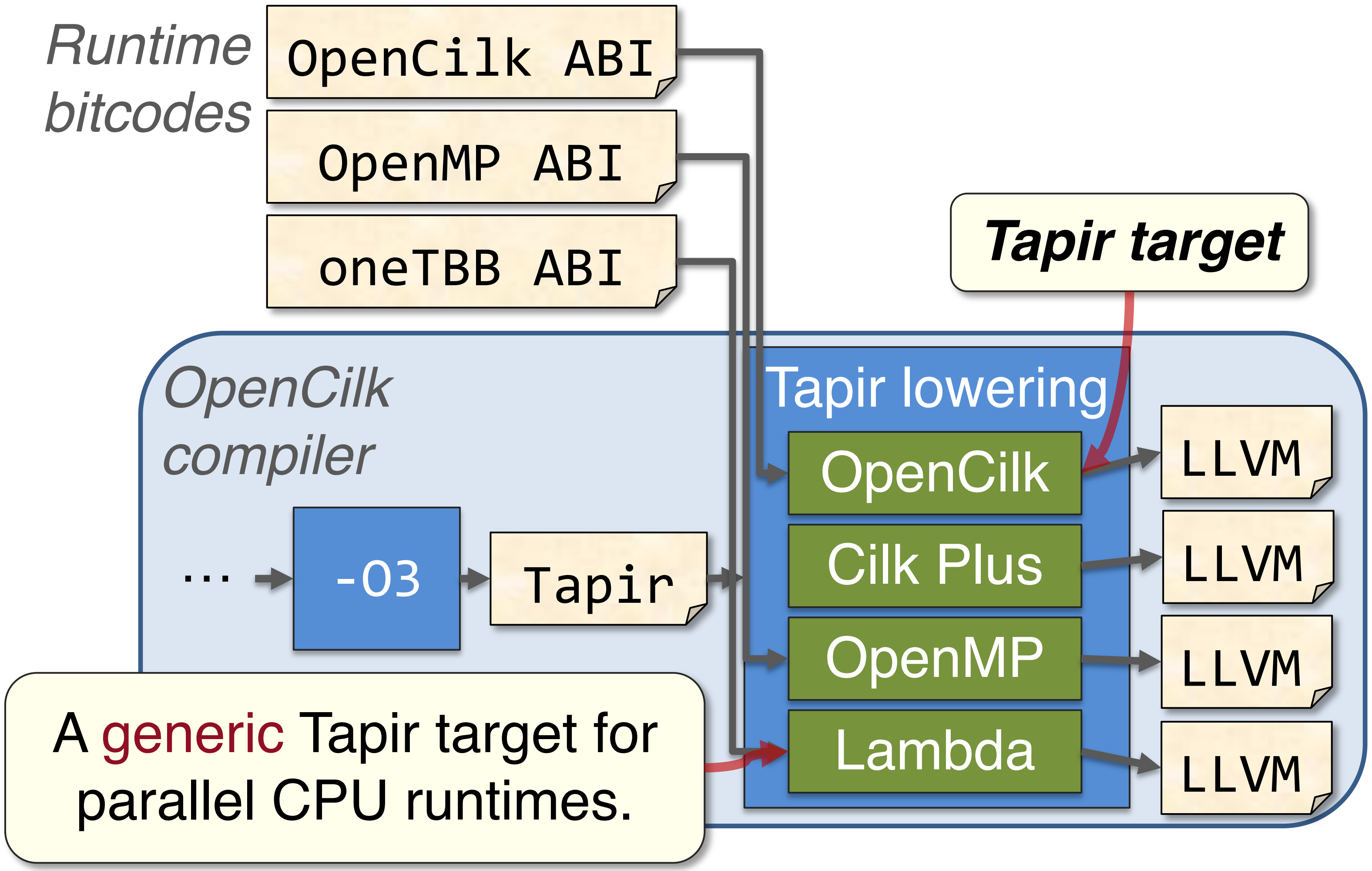


Collaborative work with Prof. I-Ting Angelina Lee.

[SL23] Schardl, Lee. OpenCilk: A Modular and Extensible Software Infrastructure for Fast Task-Parallel Code. PPoPP, 2023.

Adding new parallel-runtime backends

We extended OpenCilk to compile Cilk programs to different parallel runtime systems, including Cilk Plus, OpenMP tasks, and oneTBB.



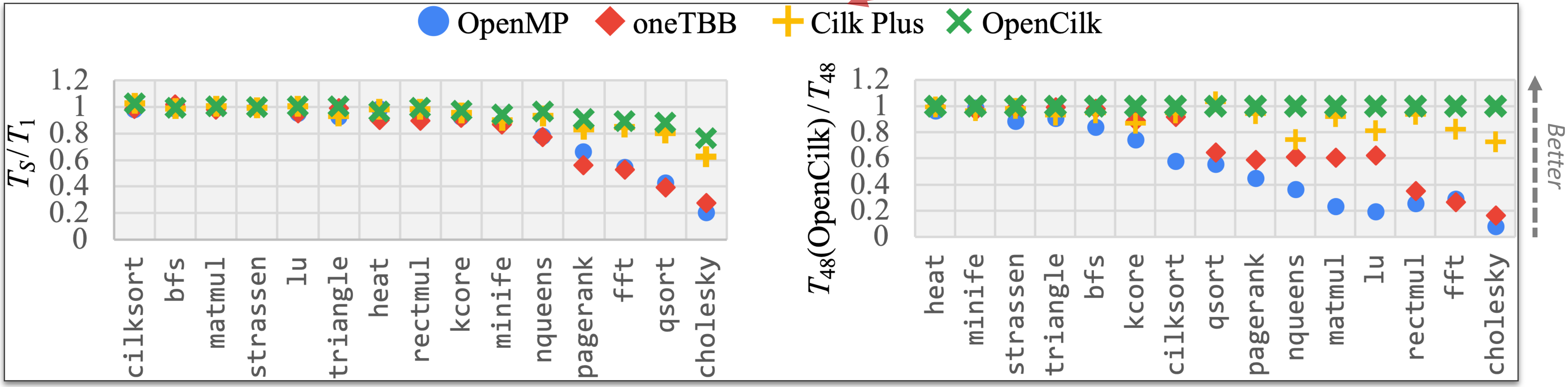
Runtime back end	Approx. new lines of code
OpenCilk	1,680
Cilk Plus	1,900
OpenMP tasks	850
oneTBB	780

Each new parallel runtime requires fewer than 2000 new lines of code.

Performance of OpenCilk

OpenCilk produces fast code.

Comparable to the original Tapir/LLVM compiler.



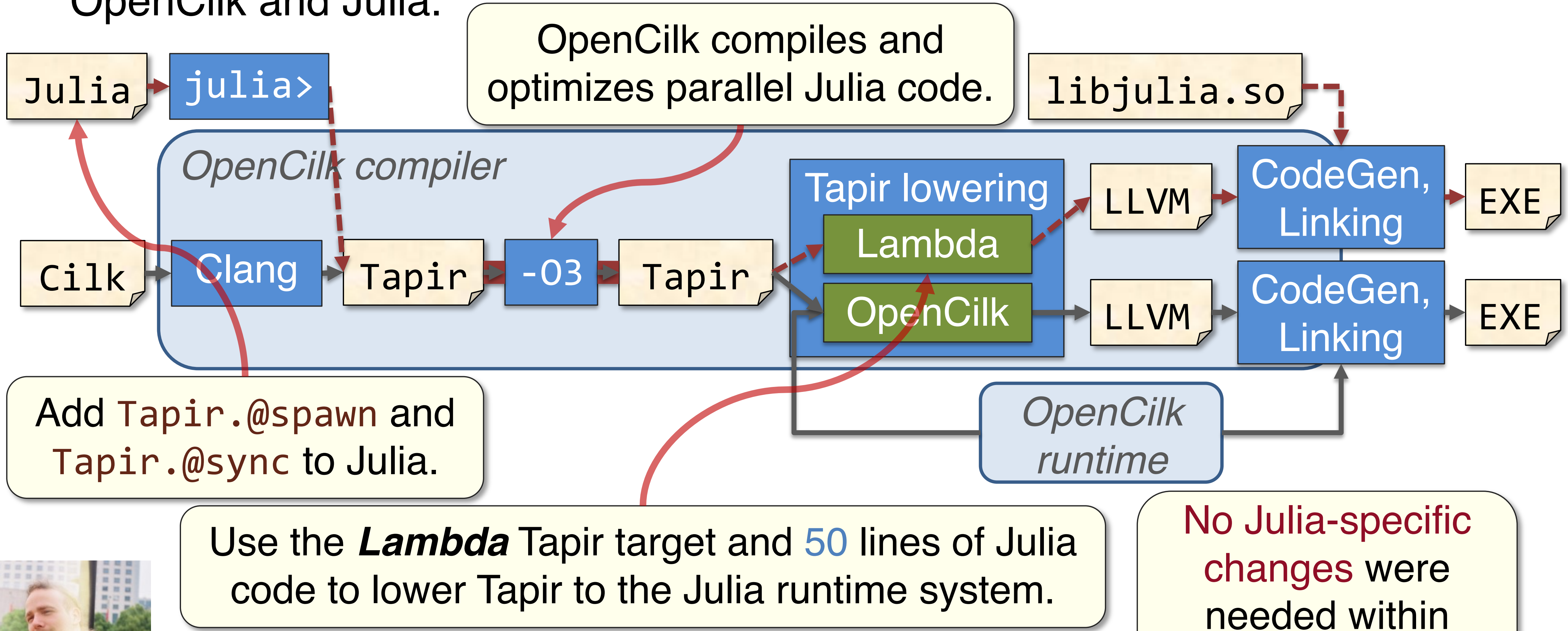
OpenCilk achieves high ***work efficiency***.

OpenCilk **scales well** on parallel processors.

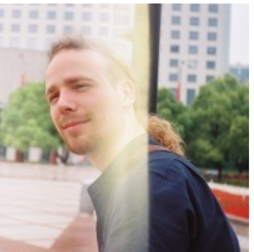
OpenCilk's design using a bitcode ABI makes it easy to engineer the runtime system to improve performance.

Integrating Julia and OpenCilk

Recently, we used these latest developments in OpenCilk to integrate OpenCilk and Julia.



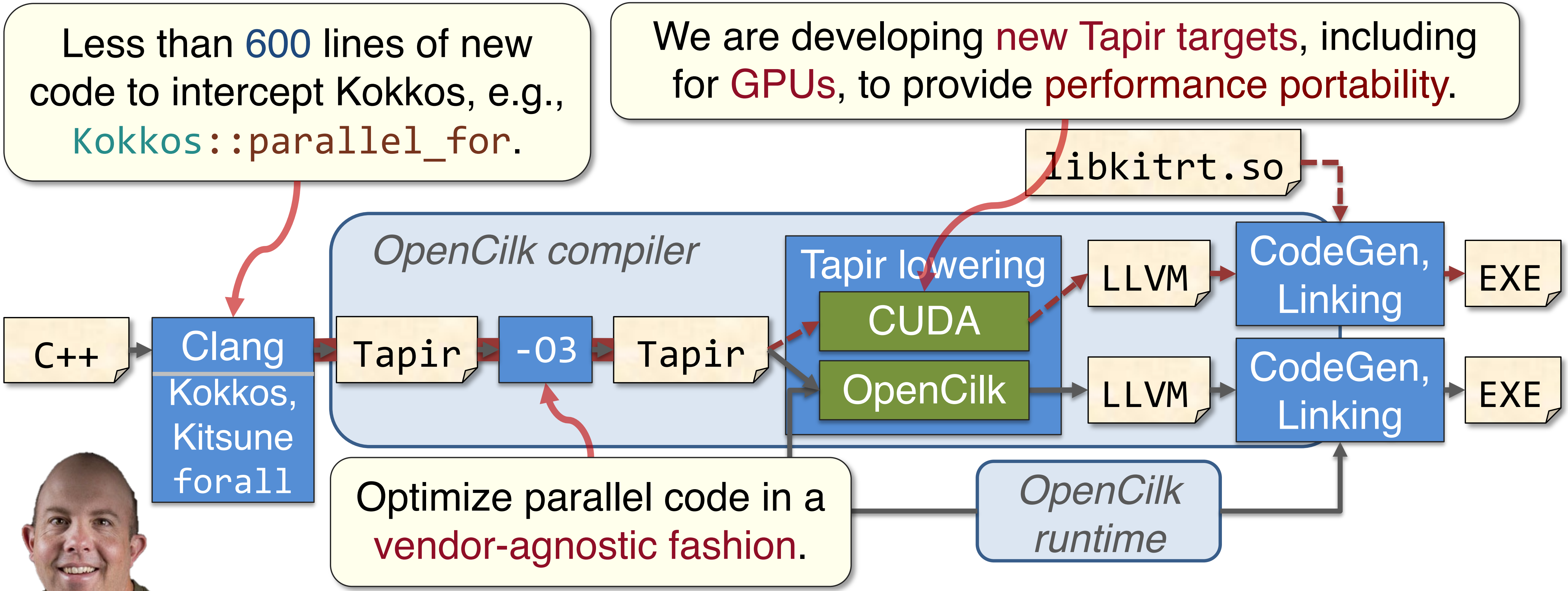
No Julia-specific changes were needed within OpenCilk!



Collaborative work with Valentin Churavy

Performance and portability with Kitsune

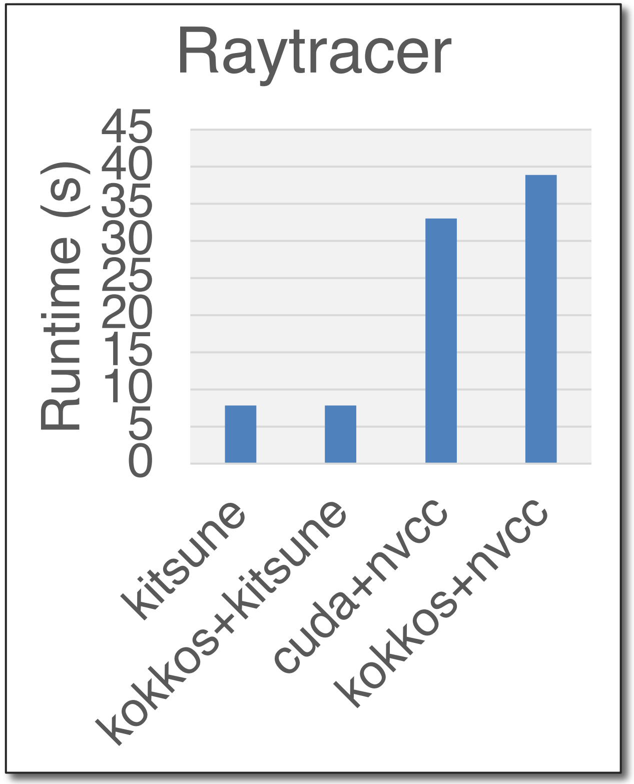
Kitsune is a parallel-aware compiler toolchain, built using OpenCilk, to compile and optimize Kokkos and other DOE software.



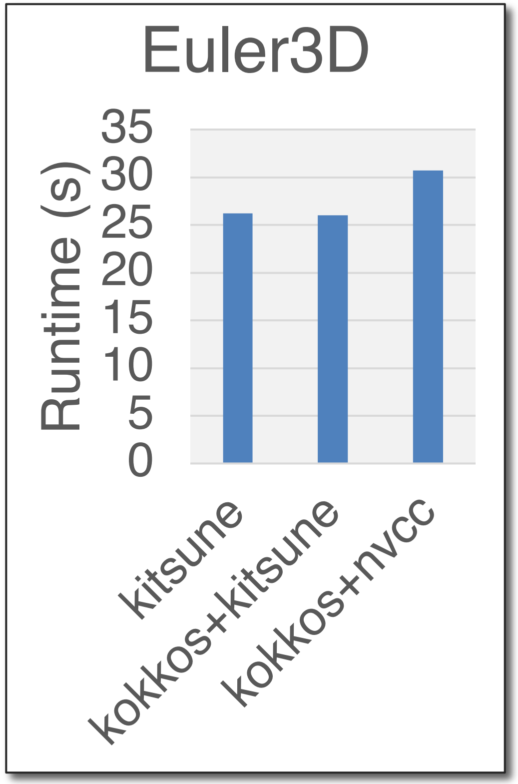
Collaborative work with Dr. Patrick McCormick and his team at LANL

Performance results, NVIDIA H100, CUDA 12.2

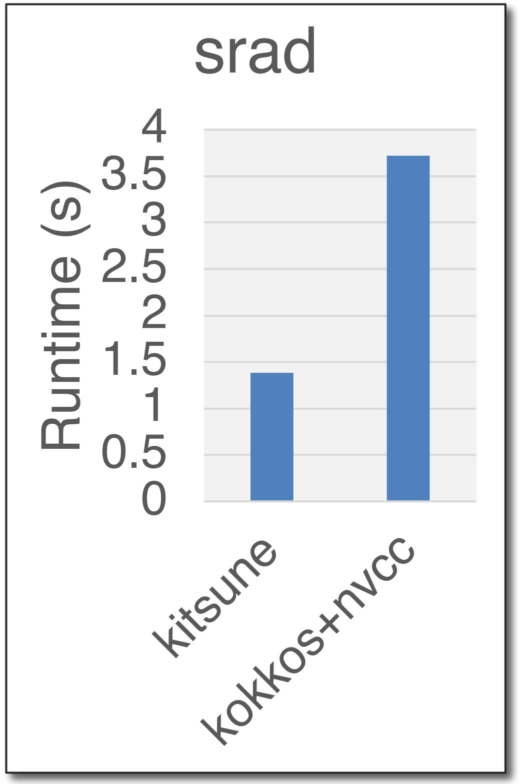
Kitsune's vendor-agnostic, parallel-aware compilation strategy improves the performance of several benchmark programs on GPUs.



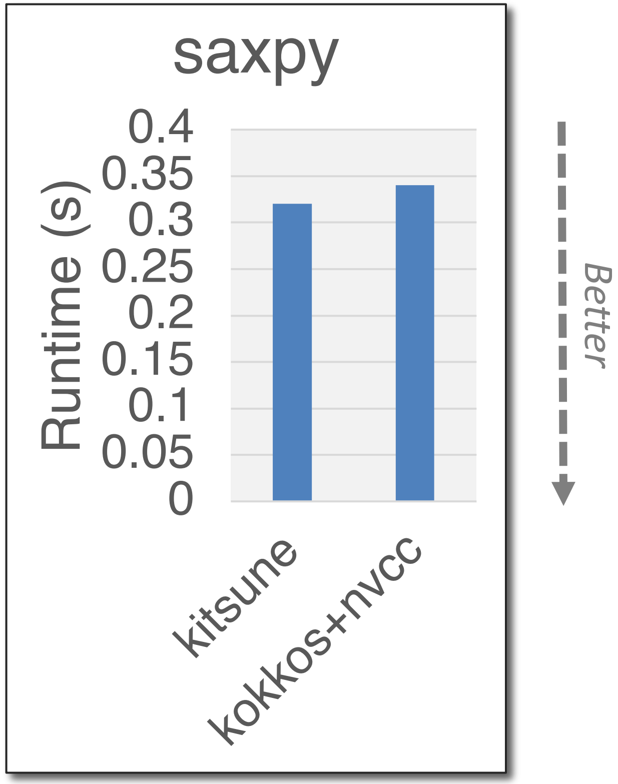
4.2x faster



1.17x faster



2.7x faster



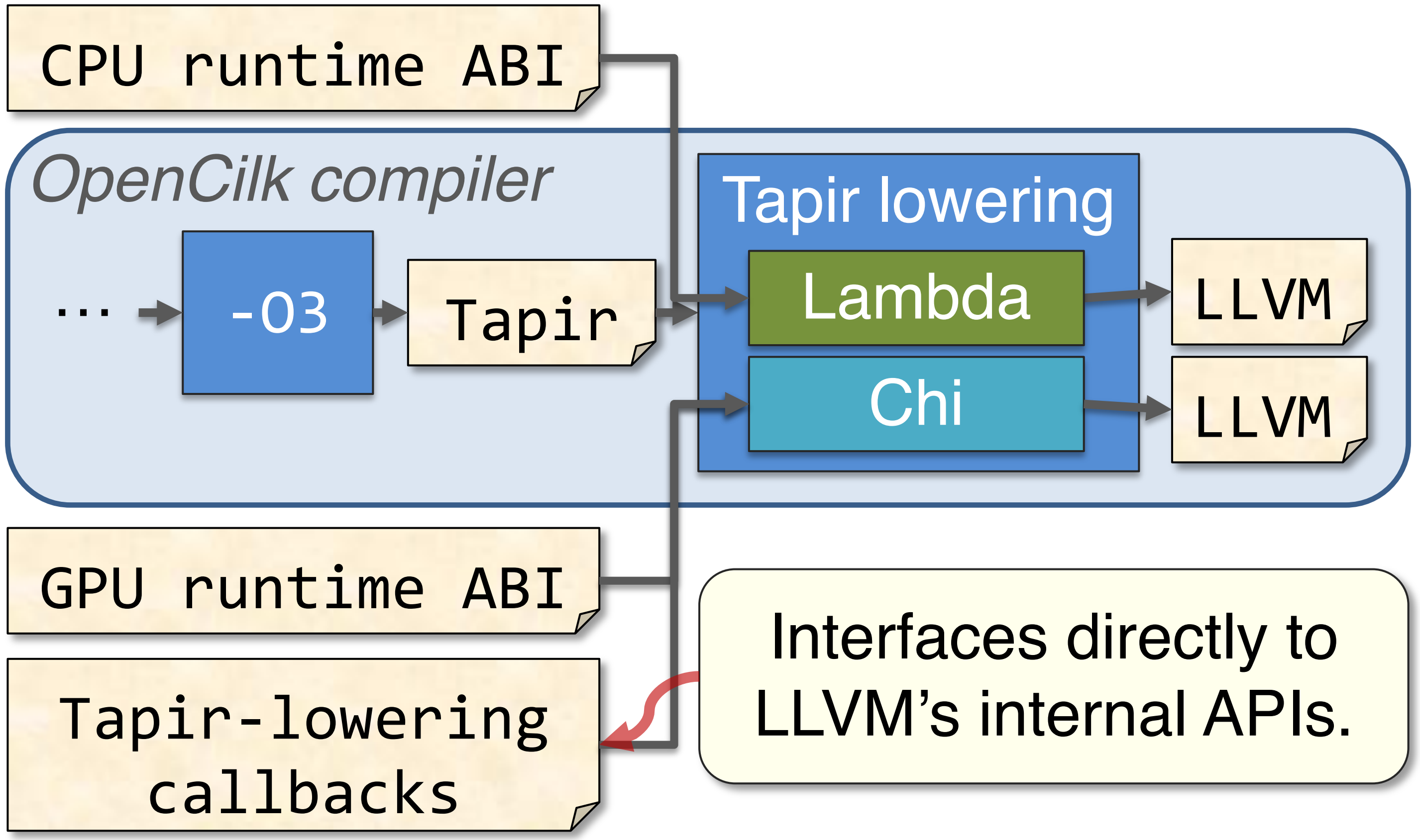
1.06x faster

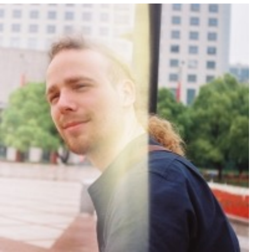
Chi: A flexible Tapir target for accelerators

We recently prototyped a new Tapir target, called ***Chi***, that aims to generalize GPU Tapir targets to move runtime details out of the compiler.

- Chi borrows many ideas and insights from Kitsune's GPU targets.
- Chi provides hooks and callbacks to specify ABI details of a language or GPU runtime ABI.

Callbacks are needed to handle the diverse ways GPU kernels are encoded and launched.





Collaborative work with Valentin Churavy

From Julia to OpenCilk to GPUs

With Chi, we were able to rapidly prototype a new Julia parallel-loop construct that compiles to a GPU kernel using OpenCilk.

```julia
function saxpy(Z, X, Y, a)
    Tapir.foreach(eachindex(Z, Y, X)) do I
        @inbounds Z[I] = a*X[I] + Y[I]
    end
    Z
end
```

Julia

New parallel-loop function that compiles to Tapir and eventually lowers to a GPU kernel.

Preliminary results using CUDA

Mechanism	Running time (us)
Tapir, Chi	75.97
GPUArrays	71.41
CUDA.jl	95.39

Performance on par with other CUDA-programming solutions in Julia.

Using Chi to support Julia's CUDA offloading required:

- ~50 lines of C++ in callbacks.
- ~150 lines of Julia code, to process and launch the GPU kernel within the Julia runtime.

For More About OpenCilk

Check out OpenCilk yourself!

- Website: https://www.opencilk.org
- GitHub: https://github.com/OpenCilk/

Use SpeedCode to try out OpenCilk online: http://speedcode.org

Special thanks to the OpenCilk team — Tim Kaler, Alexandros-Stavros Iliopoulos, John Carr, Dorothy Curtis, Bruce Hoppe, and Charles E. Leiserson — and everyone who has contributed to and supported OpenCilk.

Code-Along Preview

Come to the code-along, *Writing Fast Task-Parallel Code Using OpenCilk*, where we'll use OpenCilk to do some software performance engineering of a C/C++ matrix-multiplication code.

Implementation	Running time (s)	Relative speedup	Absolute Speedup	GFLOPS	Percent of peak
C	971.185	1.00	1	0.142	0.003
+ interchange loops	185.530	5.23	5	0.741	0.016
+ optimization flags	52.091	3.56	19	2.638	0.057
Parallel loops	1.418	36.74	685	96.925	2.103
Parallel divide-and-conquer	0.547	2.59	1,775	251.260	5.453
+ compiler vectorization, AVX2	0.245	2.23	3,964	560.975	12.174
+ compiler vectorization, AVX512	0.178	1.38	5.456	772.129	16.756
+ hand vectorization	0.052	3.42	18,677	2,643.057	57.358
oneMKL with OpenMP	0.056	0.93	17,343	2,454.267	53.261

Problem: 4k-by-4k matrix multiply **Machine:** AWS c5.metal, Intel Xeon Platinum 8275CL