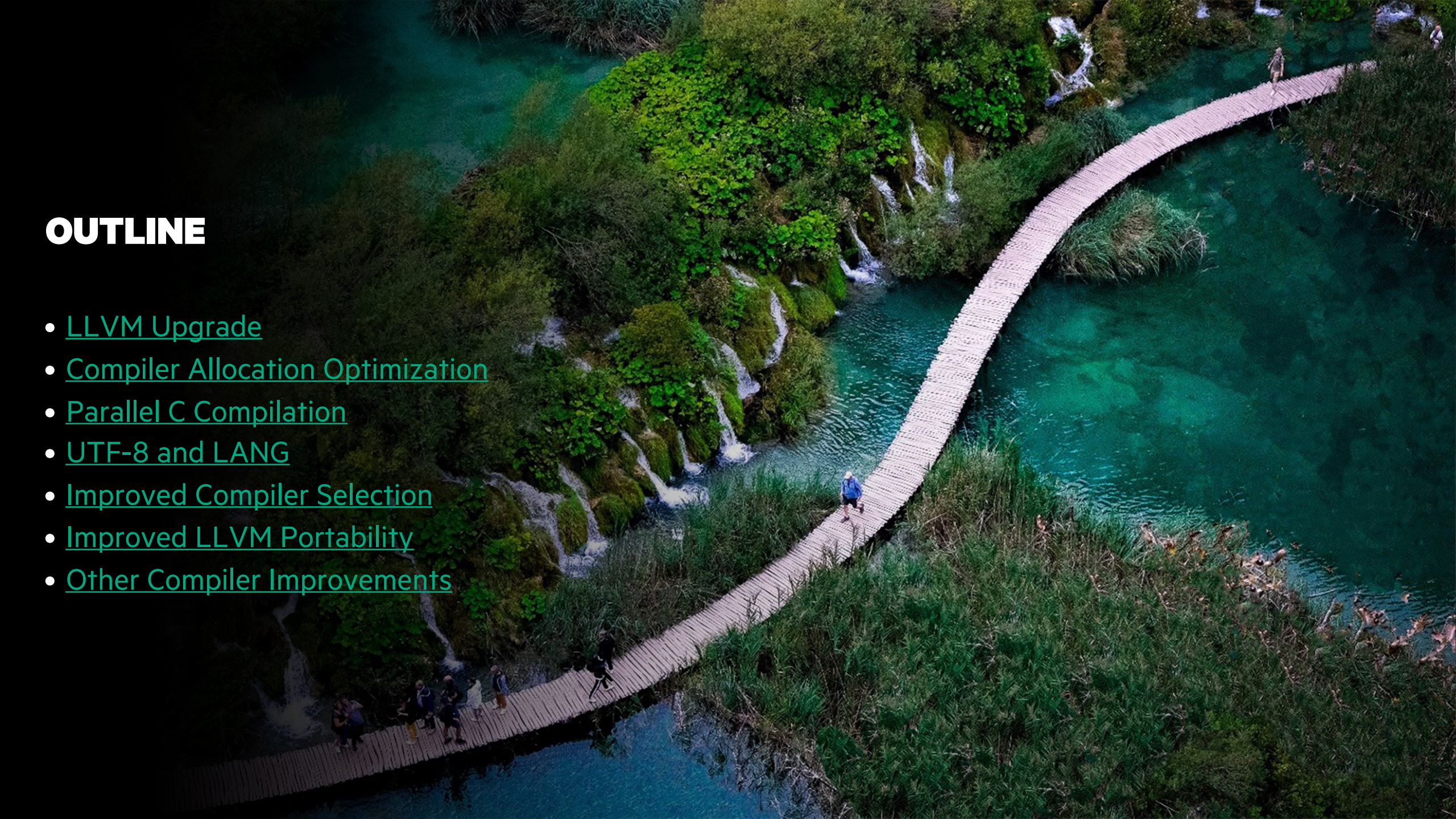
Hewlett Packard Enterprise

CHAPEL RELEASE NOTES, 1.25.1 / 1.26.0: COMPILER IMPROVEMENTS

Chapel Team

December 9, 2021 / March 31, 2022



OUTLINE

- LLVM Upgrade
- Compiler Allocation Optimization
- Parallel C Compilation
- UTF-8 and LANG
- Improved Compiler Selection
- Improved LLVM Portability
- Other Compiler Improvements

LLVM UPGRADE

LLVM UPGRADE

Background and This Effort

Background:

- The Chapel compiler started using LLVM as its default back-end in version 1.25
  - Supported LLVM version 11 (LLVM-11) only

This Effort:

- Upgraded the compiler to support multiple LLVM versions: 11, 12, or 13
  - Located and fixed problems due to API differences between major versions
  - When multiple supported system versions are available, the most recent is used (e.g., LLVM-13)
- Upgraded the bundled LLVM to version 13
- Updated nightly testing to begin using LLVM-13
  - Continued testing with 11 and 12 to a lesser extent
- Updated documentation and error messages to indicate supported LLVM versions

LLVM UPGRADE

Impact and Status

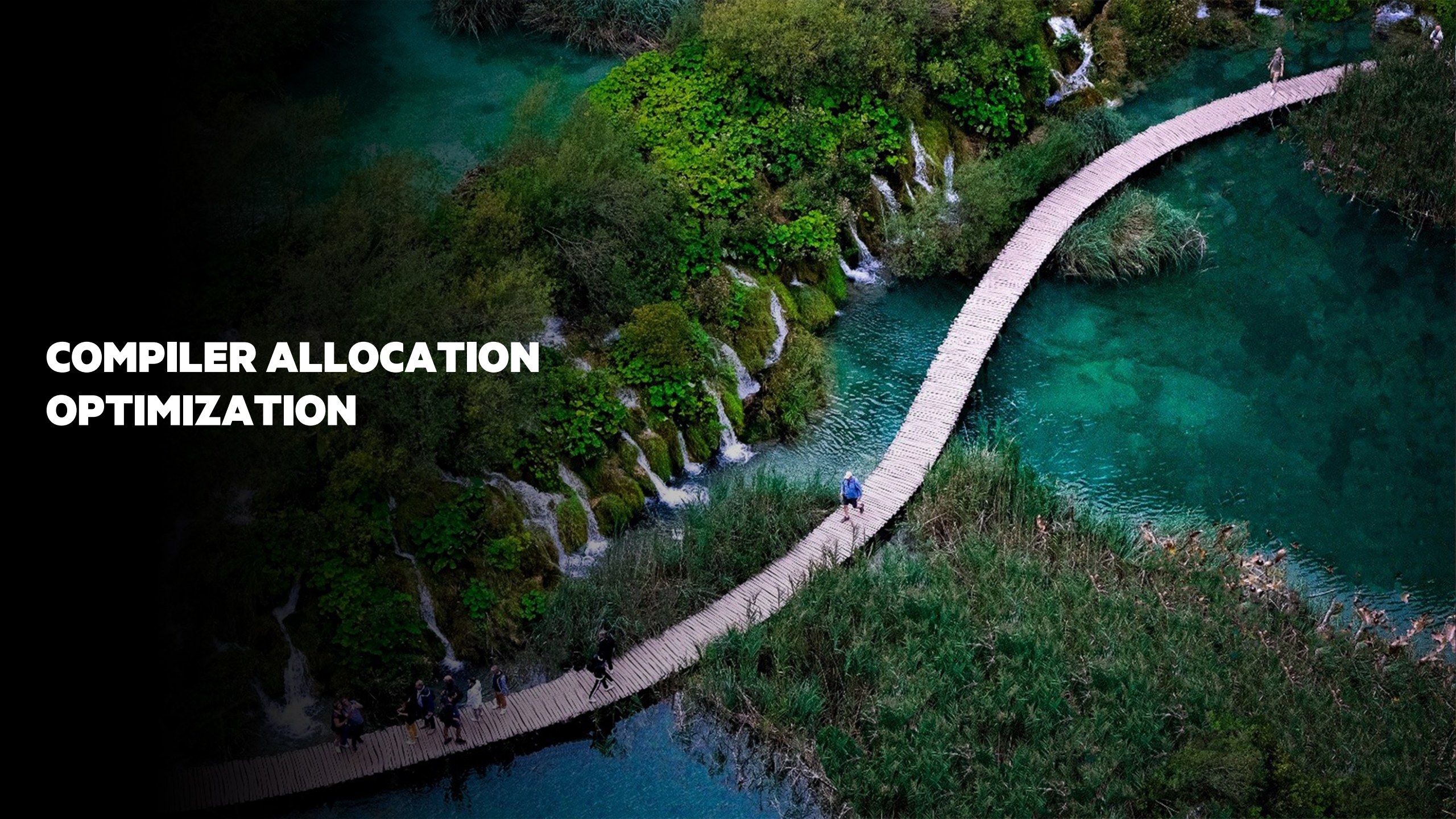
Impact:

- The Chapel compiler works with multiple LLVM versions
  - Users with system LLVM installations don't need to upgrade LLVM in lock-step with Chapel
  - Chapel can benefit from improvements to LLVM

Status:

- LLVM-13 is bundled with the Chapel release is the default for pre-built packages
- LLVM-13 is tested nightly against many tests and configurations
- LLVM-12 and LLVM-11 are still supported, but are tested less heavily

COMPILER ALLOCATION OPTIMIZATION



COMPILER ALLOCATION OPTIMIZATION

Background: The ‘chpl’ compiler allocates many objects

- ~7 million allocations for ‘chpl examples/hello.chpl’

This Effort: Added support for building ‘chpl’ with jemalloc

- jemalloc is used in the generated executable for its strength in parallel allocation
- Can also perform significantly better than the system allocator for some single-threaded allocation workloads

Status:

- The compiler can be built with jemalloc enabled using ‘CHPL_HOST_MEM=jemalloc’ before using ‘make’

Impact:

- 13% faster Hello World compile time (LLVM and Clang back-end)
- 16% faster Arkouda compile time with LLVM back-end (10% with Clang)

Next Steps: Enable as default

- Address any portability issues with overriding system allocator

SUPPORT FOR PARALLEL COMPILATION WITH C BACK-END

PARALLEL C COMPILATION

Background and This Effort

Background:

- 'chpl' supports both C-based and LLVM-based back-ends, where LLVM is now the default
- The CHAMPS team has suffered from growing compilation costs (time and memory) as their code has grown
  - they use the C back-end, in part because memory can be reduced by doing the C compile + link step as a separate step
  - yet, compilation overheads had become painful
    - e.g., icing model compilation was taking 19+ minutes, 35 GB
  - they could address some of this by tightening up the code, reducing reliance on 'use's, generics, etc.
    - e.g., improved icing model to 8-12 minutes, 8-9 GB
- Longer-term, our 'dyno' compiler rewrite is being designed to address such cases
  - but perhaps there's more we could do in the short-term?

This Effort:

- Enabled parallelization of the C compilation step

PARALLEL C COMPILATION

Background: the '--incremental' flag

- Chapel's C back-end generates a C file per Chapel module, plus a single header file with declarations
  - each internal and standard module also results in a C file
    - e.g., simple "hello, world" results in 76 '.c' files
  - all C files are '#include'd into a single logical '.c' file during C compilation
    - rationale: gives the C compiler full visibility of the generated code to maximize optimization opportunities
    - however, this monolithic approach can also require a lot of memory

- Chapel 1.14.0 added an experimental '--incremental' flag that enables separate compilation
  - with it, each generated user module's C file can be compiled separately
    - reduces memory requirements
  - all declarations are still generated within a single, shared header file, '#include'd by each '.c' file
    - as a result, compilation time tends to be higher since this (often large) header is re-parsed for each module

PARALLEL C COMPILATION

This Effort and Impact

This Effort:

- Extended '--incremental' to support standalone '.c' files for standard/internal modules as well as the user's
- Restructured compiler-generated Makefile for C compilation to enable 'make –j' after '--incremental'
- Added a new '-j'/'--parallel-make' developer flag that applies both '--incremental' and 'make –j'

Impact:

- Significantly reduced the time and memory requirements of compiling large applications

CHAMPS C compilation step	time	memory
prior to this effort	214 sec	6.9 G
with '--incremental' + 'make –j'	53 sec	0.6 G
improvement	**4.0x**	**11.5x**

Arkouda 'makeBinary' step	time
prior to this effort	190.9 sec
with 'chpl –j'	60.3 sec
improvement	**3.17x**

PARALLEL C COMPILATION

Next Steps

- Explore ways to improve the LLVM back-end as well
  - reduce memory requirements
  - enable parallel compilation across modules, similar to ‘make –j’
- Continue to reduce the memory/time requirements of the C back-end
  - remove ‘private’ declarations from the header file
  - break the monolithic header file into a ‘.h’ file per module and only ‘#include’ those that a given ‘.c’ file needs
- Explore ways to reduce the amount of code generated by the Chapel front-end
  - reduce code clones that result from generic routines (“type erasure”) [#10851]
  - look for ways to automate other transformations that the CHAMPS team performed manually
- Explore the impact of ‘-j’ / ‘--incremental’ on program performance
  - how much does enabling link-time optimizations in the C compiler help performance / impact compile-time?
- Continue work on ‘dyno’ compiler rework in support of separate/incremental Chapel compilation

UTF-8 AND LANG

UTF-8 AND LANG

Background: As of 1.25.1, Chapel required certain environment variables to function correctly

- Chapel documentation suggested specific settings for the C library locale variables to enable UTF-8 support:

```
LANG=en_US.UTF-8
LC_COLLATE=C
LC_ALL=""
```

- Observed strange bugs for users who did not have these variables set in a compatible way
- At the same time, the Chapel 'string' type always stores UTF-8 and the I/O code does not convert
  - i.e., trying to use these variables to control input format would not work

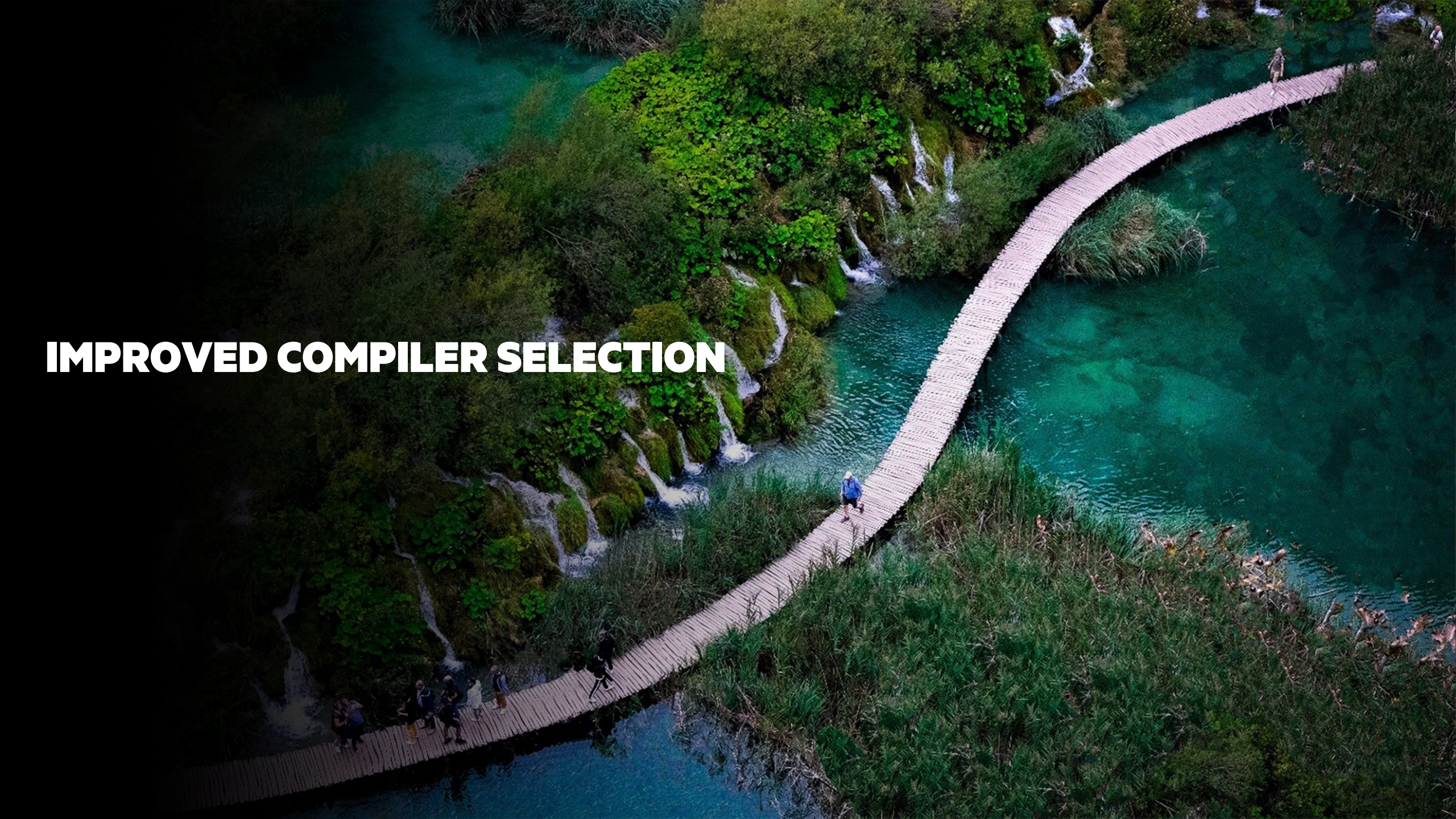
This Effort: Removed requirement to set these environment variables and updated the documentation

- Chapel runtime now configures the C library for UTF-8 support
  - Currently needed to determine a codepoint's width in columns

Next Steps:

- Allow the character set to be configured on a per-file basis when opening a file
  - Since 'string' is still UTF-8, I/O code will need to convert to/from Unicode

IMPROVED COMPILER SELECTION



COMPILER SELECTION
Background

- 'CHPL_{HOST,TARGET}_COMPILER' selects the compiler family used to build Chapel
  - HOST is used for 'chpl' compiler and launchers
  - TARGET is used for runtime and generated code
  - Can be set to a compiler family name (e.g., gnu, intel, llvm, ibm)

- 'CC', 'CXX', 'CHPL_{HOST,TARGET}_{CC,CXX}' were used to select specific compilers
  - Setting 'CC'/'CXX' impacts the defaults of other variables:

```
CC=/usr/bin/gcc
# led to these defaults:
CHPL_HOST_CC=/usr/bin/gcc
CHPL_TARGET_CC=/usr/bin/gcc
```

  - Setting 'CC'/'CXX' also interfered with LLVM-by-default:

```
CC=/usr/bin/gcc
# led to this default:
CHPL_TARGET_COMPILER=gnu  # potentially surprising with default LLVM code generation when CHPL_LLVM=system
```

COMPILER SELECTION

This Effort and Impact

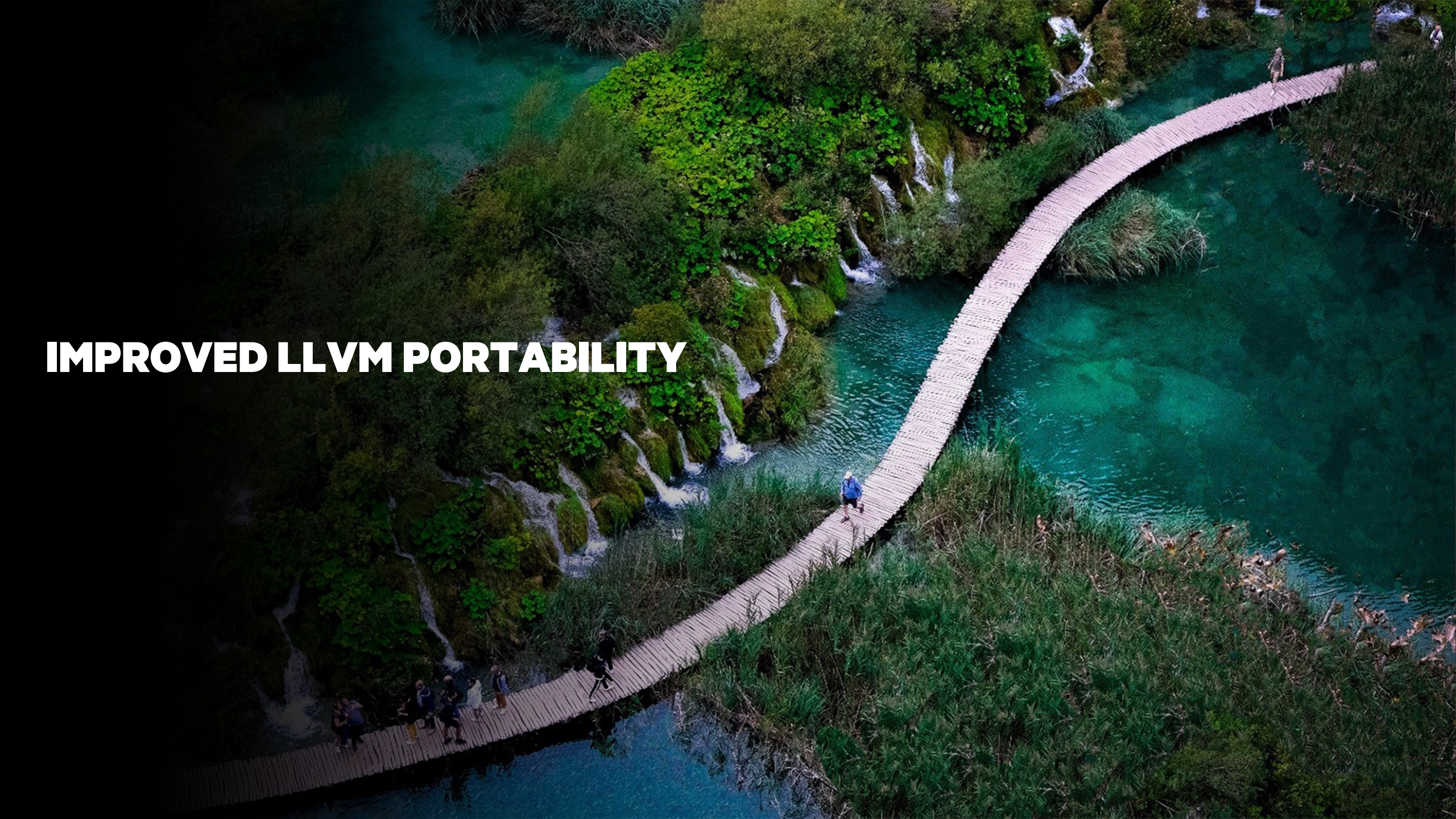
This Effort:

- Stop inferring CHPL_TARGET_COMPILER when using LLVM or the PrgEnv compilers

Impact:

- Chapel configuration is more predictable and flexible in 1.25.1

IMPROVED LLVM PORTABILITY



LLVM PORTABILITY
Background

Background:

- The LLVM back-end became the default in 1.25.0
- Soon after, began observing portability problems
  - On some systems, would try to link with a system RE2 library instead of the bundled one
  - Development packages for Clang on some Linux distributions did not include the static libraries used by Chapel
  - On Cray XC systems, the compiler was saving paths to C compiler resources that changed with C compiler upgrades

This Effort:

- Fixed the above issues in 1.25.1:
  - Carefully construct link and include search paths to list the bundled paths before system paths
  - On Linux systems, use the more common clang-cpp library instead of the Clang static libraries
  - On Cray XC systems, compute C compiler resources when 'chpl' is invoked to handle version changes in C compilers

OTHER COMPILER IMPROVEMENTS

OTHER COMPILER IMPROVEMENTS

For a more complete list of compiler changes and improvements in the 1.25.1 and 1.26.0 releases, refer to the following sections in the CHANGES.md file:

- ‘Packaging / Configuration Changes’
- ‘Tool Improvements’
- ‘Portability’
- ‘GPU Computing’
- ‘Compiler Improvements’
- ‘Error Messages / Semantic Checks’
- ’Bug Fixes’


THANK YOU
https://chapel-lang.org
@ChapelLanguage