• Generated Assembly
• Reducing Compilation Time
• Array Creation Optimizations
• Parallel Array Deinitialization
• Docker Changes
• LLVM Status
• Portability and Prerequisites
• Documentation Improvements
SHOWING THE GENERATED ASSEMBLY
SHOWING THE GENERATED ASSEMBLY

**Background:** Performance-minded users have requested a way to view a procedure’s generated assembly
- Useful for checking compiler optimizations and for evaluating different ways to write something in Chapel

**This Effort:** Enabled showing an assembly dump for a specific function
- For example, we might like to know if the procedure below uses a vectorized ‘sqrt()’
- The command on the right can be used to answer this question
  ```chapel
  config const n = 16;
  var A: [1..n] real(32);
  proc foo() {
    foreach i in 1..n {
      A[i] = sqrt(i: real(32));
    }
  }
  foo();
  ```

**Status:** The new flag currently only works when using the LLVM backend

```bash
$ chpl program.chpl --fast \
  --llvm-print-ir foo \ 
  --llvm-print-ir-stage asm

# Disassembling symbol foo_chpl
... output showing vsqrtss instruction ...
```
REDUCING COMPILATION TIME
Reduced Polynomial Overhead in Compiler

**Background:** Chapel users and developers are understandably annoyed by slow compilation times
- Long-term, ‘dyno’ is being designed and engineered to help reduce compilation times
- In the meantime, large applications like Arkouda are suffering

**This Effort:** Eliminated one source of polynomial overhead in the compiler
- For each routine returning ‘true’/’false’/’void’, the compiler looked at all occurrences of that value in the program
  - This included a huge number of occurrences internal to the compiler

**Impact:** 20% reduction in Arkouda build time

**Next Steps:** Continue speeding up the compiler
- Look for similar sources of overhead in production
- Continue improving ‘dyno’s resolution capabilities
  - Goal: make it the production resolver
REDUCING COMPILATION TIME
Building Compiler with ‘jemalloc’

**Background:** The compiler allocates many objects
- ~7 million allocations for ‘chpl examples/hello.chpl’
- Previous releases added the option to build the compiler with ‘jemalloc’, which improves allocation performance
- Users had to opt-in to using ‘jemalloc’ to benefit from improvements

**This Effort:** Made ‘jemalloc’ the default for building the compiler whenever possible

**Impact:** 25% reduction in Arkouda build time
ARRAY CREATION OPTIMIZATIONS
**Background and This Effort**

**Background:** Chapel uses *privatization* to replicate distributed domain and array metadata to all locales

- Privatization increases creation time, but speeds up later uses
- Creation time is not a bottleneck for many codes
  - Tends to be outside timed kernels for most benchmarks
  - HPC applications tend to create arrays once and heavily reuse them
- Unlike most HPC codes, Arkouda frequently creates new arrays
  - A recent operation to display a summary of a DataFrame (DF) creates dozens of small arrays
  - This motivated trying to improve array creation speed

**This Effort:** Optimized distributed domain and array privatization

- Improved communication strategy used to broadcast metadata
- Eliminated re-privatization when creating rectangular domains
ARRAY CREATION OPTIMIZATIONS

Impact

- Improved performance for distributed domain and array creation
  - Non-trivial speedup for many Arkouda operations, especially when combined with ‘SymEntry’ optimizations

### 16-node Block Domain/Array Creation

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Before</th>
<th>After</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>DF Display</td>
<td>0.8 s</td>
<td>0.4 s</td>
<td>2x</td>
</tr>
<tr>
<td>Stream</td>
<td>465 GiB/s</td>
<td>600 GiB/s</td>
<td>1.3x</td>
</tr>
<tr>
<td>Scan</td>
<td>580 GiB/s</td>
<td>1010 GiB/s</td>
<td>1.7x</td>
</tr>
</tbody>
</table>

16-node Arkouda

- Chapel 1.28
- Chapel 1.29
ARRAY CREATION OPTIMIZATIONS

Next Steps

• Further optimize domain and array creation
  • Implement minor communication and allocation reductions for ‘BlockDist’
  • Reset task placement to improve cache reuse between domain and array creation
  • Explore replacing eager privatization with on-demand forwarding
PARALLEL ARRAY DEINITIALIZATION
Background and This Effort

**Background:** Array elements are initialized in parallel, but were historically deinitialized serially
- Parallel init is important for first-touch and speeding up memory fault-in for all types
- Many types do not require deinit
  - Only complex types like domains/arrays and records/classes with ‘deinit( )’ methods
- Historically, trying to parallelize deinit resulted in large regressions for array-of-arrays
  - Caused by contention on a lock used to implement domain reference counting and array tracking
  - These overheads have been reduced in recent releases, but not eliminated
- Recently-added Arkouda ‘bigint’ arrays were impacted by slow serial deinitialization
  - Motivated revisiting parallel deinitialization

**This Effort:** Parallelized array deinitialization for all types
- Uses the same size heuristics as parallel initialization
PARALLEL ARRAY DEINITIALIZATION

Impact

- Faster array deinitialization for many types that require deallocation, including Arkouda ‘bigint’ arrays

- Slower array-of-arrays deinitialization, though still faster than initialization
PARALLEL ARRAY DEINITIALIZATION

Next Steps

• Reduce overheads for initializing and deinitializing array-of-arrays
  • Reduce need for locking by using atomic counter for reference counting
  • Do bulk reference counting for array-of-arrays
  • Explore eliding reference counting if compiler can prove lifetimes
**DOCKER CHANGES**

**Background:**
- Previous Dockerfile fetched latest release’s source tarball from GitHub and built that release’s image
  - Only provided pre-built LLVM backend

**This Effort:**
- Modified Dockerfile to build from its containing Chapel source tree and build the C backend as well

**Impact:**
- Enabled building and using Chapel Docker images from any version of Chapel source code
  - Can build images from specific commits
  - More in line with general practice for Dockerfiles
  - Removes the necessity of fetching the latest release
  - Allows creation of a CI job to test building Docker image from latest source
- C backend can be used to reduce time or memory overheads when compiling Chapel programs
LLVM STATUS

Background:
- LLVM is Chapel's recommended backend
  - Versions 11–14 are supported and tested nightly
  - Version 15 removed support for typed pointers, which the Chapel compiler has relied upon

This Effort:
- Started adjusting the LLVM backend to stop using typed pointers
  - Manually tracking types for LLVM pointers where needed

Status:
- More work remains before Chapel can support LLVM 15

Next Steps:
- Make LLVM 15 the default
  - Continue adjusting the backend to use opaque pointers
PORTABILITY AND PREREQUISITES
PORTABILITY

Background: Have been gradually improving portability of Chapel on a variety of Unix systems

This Effort: Performed ad hoc testing with many current operating systems

Status: Verified portability to 12 OS distributions and 32 versions:

- ‘make’ and ‘make check’ work with or without the system LLVM package on the following systems:
  - Alma Linux 8, 9.0, 9.1
  - Alpine Linux 3.15, 3.17
  - Amazon Linux 2
  - Arch linux (March 2023 version)
  - CentOS Stream 8, 9
  - Debian 10, 11, 12
  - Fedora 34, 35, 36
  - FreeBSD 12.2, 12.4, 13.1
  - Mac OS X (with Homebrew)
  - OpenSuse Leap 15.3, 15.4
  - Rocky Linux 8, 9.0, 9.1
  - Ubuntu 20.04, 22.04, 22.10
  - Ubuntu 22.04 with Homebrew

- ‘make’ and ‘make check’ work with ‘quickstart’, but the system LLVM package cannot be used
  - Amazon Linux 2023
  - CentOS 7 with Devtoolset 11
  - Fedora 37, 38

Next Steps: Automate this portability testing to run it more frequently
**PREREQUISITES DOCUMENTATION**

**Background:** Chapel requires some tools to be pre-installed in order to build correctly

**This Effort:** Wrote scripts to automatically generate platform-specific prerequisite docs
- lists commands for installing required packages based on portability testing results

**Impact:** Users with tested distributions can easily find commands to install prerequisites
DOCUMENTATION IMPROVEMENTS
DOCUMENTATION IMPROVEMENTS

Background and This Effort

Background:
• For 2.0, beyond keeping documentation up-to-date, we’ve also been improving descriptions of existing features
• Recent releases have particularly focused on the “Built-in Types and Functions” section of the docs
  – These were topics that were technically part of the language, yet whose documentation was generated by ‘chpldoc’

This Effort:
• Folded the remaining “Built-in Types and Functions” topics from Chapel 1.28 into the language specification
• Clarified the language specification with respect to several features:
  – abstract argument intents
  – storage of records with array fields
  – ‘out’ arguments and split initialization
  – ‘yield’ semantics
  – re-exporting symbols
  – non-promoted arguments in promoted expressions
  – definitions of subroutine bodies
• Also improved documentation for several standard modules
DOCUMENTATION IMPROVEMENTS

Impact and Next Steps

Impact:

• The “Built-in Types and Functions” section of the sidebar no longer exists:

• Chapel’s documentation continues to reflect the language better and more accurately going into Chapel 2.0

Next Steps: Continue improving docs as we approach Chapel 2.0
OTHER IMPLEMENTATION / PACKAGING IMPROVEMENTS
For a more complete list of implementation and packaging changes and improvements in the 1.29.0 and 1.30.0 releases, refer to the following sections in the CHANGES.md file:

- ‘Configuration / Build / Packaging Changes’
- ‘Tool Improvements’
- Compilation-Time / Generated Code Improvements’
- ‘Performance Optimizations / Improvements’
- ‘Language Specification Improvements’ and ‘Other Documentation Improvements’
- ‘Portability / Platform-specific Improvements’
- ‘Compiler Improvements’ and ‘Compiler Flags’
- ‘Error Messages / Semantic Checks’
- ‘Bug Fixes’
- ‘Third-Party Software Changes’
- ‘Developer-oriented changes: ...’