Compiler / Implementation

Chapel Team, Cray Inc.

Chapel version 1.16

October 5, 2017

Safe Harbor Statement

This presentation may contain forward-looking statements that are based on our current expectations. Forward looking statements may include statements about our financial guidance and expected operating results, our opportunities and future potential, our product development and new product introduction plans, our ability to expand and penetrate our addressable markets and other statements that are not historical facts. These statements are only predictions and actual results may materially vary from those projected. Please refer to Cray's documents filed with the SEC from time to time concerning factors that could affect the Company and these forward-looking statements.

Outline

- LLVM Back-end Improvements
- Partially Generic Resolution Changes
- Iterator Inlining Improvements
- Dynamic Cast Optimization
- Other Compiler Improvements
- Error message improvements
- Bug Fixes

LLVM Back-end Improvements

LLVM: Background

- **We'd like to use LLVM more with Chapel…**

  …but have not been able to do so yet due to:
  - performance problems
  - insufficient testing

- **What is LLVM?**
  - Open source, modular, reusable compiler technologies

- **How does Chapel work with LLVM?**
  - Since 1.6, one can build Chapel with CHPL_LLVM=llvm
  - Then, the --llvm compilation flag activates the LLVM back-end
  - --llvm is an alternative to generating C and running a C compiler

LLVM: Background

- **Why do we want to work more with the LLVM back-end?**
  - only have to wrestle with portability/performance of a single back-end
    - instead of multiple versions of various C compilers
  - possible to create Chapel-specific low-level optimizations
  - Chapel compiler can choose which low-level optimizations run
  - possibility to migrate some Chapel optimizations to LLVM
    - LLVM is a strong, documented framework
    - more compiler developers are proficient with LLVM than with Chapel AST

LLVM: This Effort

- **Prepare LLVM back-end for production use**
  - Testing improvements
    - Nightly testing now includes full --llvm testing instead of small subset
  - Performance improvements
    - Upgraded from LLVM 3.7 to 4.0
    - Google Summer of Code improvements
  - Developer-focused changes
    - Chapel sources are ready to build with LLVM 5
    - New command-line option --mllvm to set LLVM optimization flags
    - Perform more LLVM optimizations in 'chpl' rather than at link-time

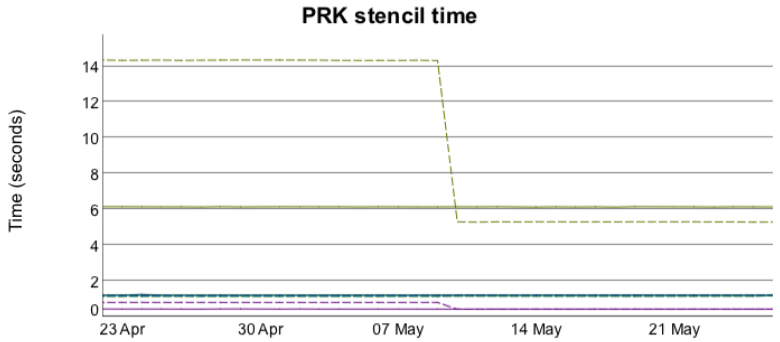
LLVM: Google Summer of Code

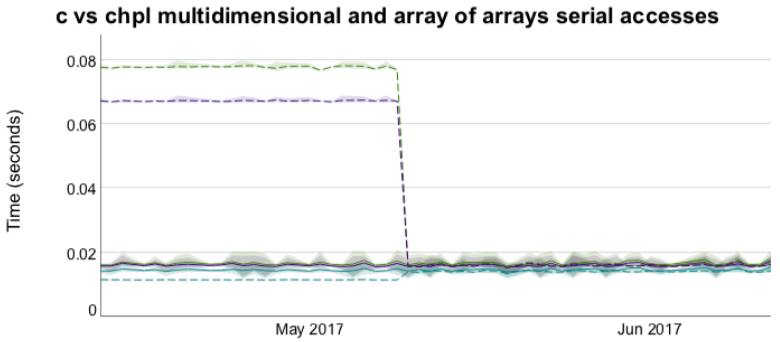
LLVM GSoC: This Effort

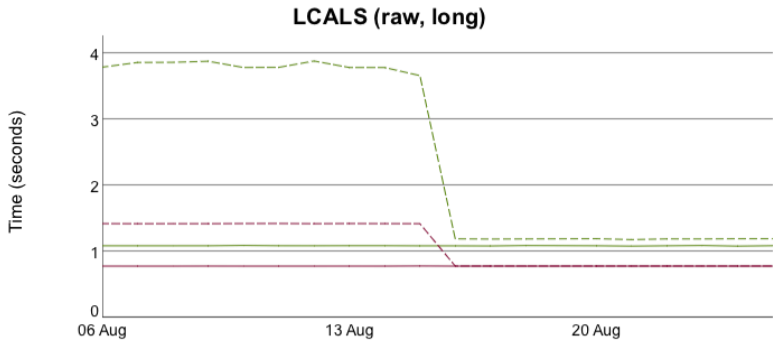
- **Przemysław Leśniak contributed many improvements:**
  - mark signed integer arithmetic with 'nsw' to improve loop optimization
  - command-line flags to emit LLVM IR at particular points in compilation
  - new tests that use LLVM tool FileCheck to verify emitted LLVM IR
  - mark order-independent loops with llvm.parallel_loop_access metadata
  - mark const variables with llvm.invariant.start
  - enable LLVM floating point optimization when --no-ieee-float is used
  - add nonnull attribute to ref arguments to functions
  - use clang built-ins to improve performance of arithmetic on complex numbers

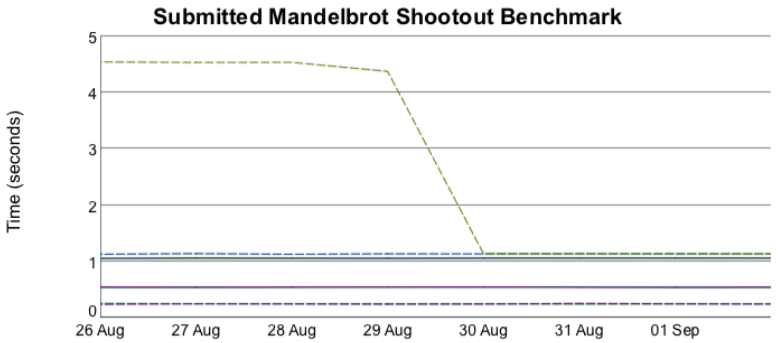
LLVM GSoC: Impact

- **Performance improvements for a variety of benchmarks**
  - Notably, many array-focused benchmarks now on par with C back-end







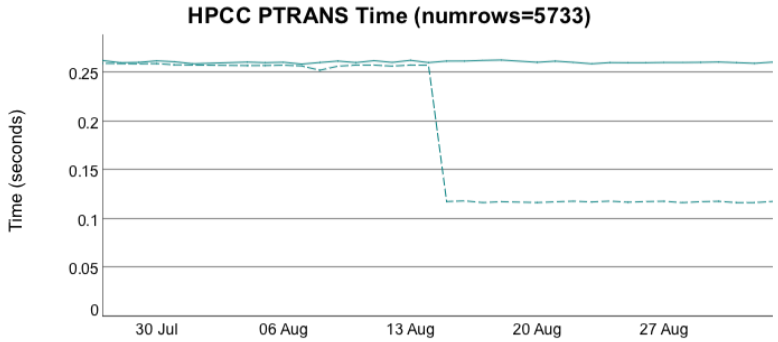


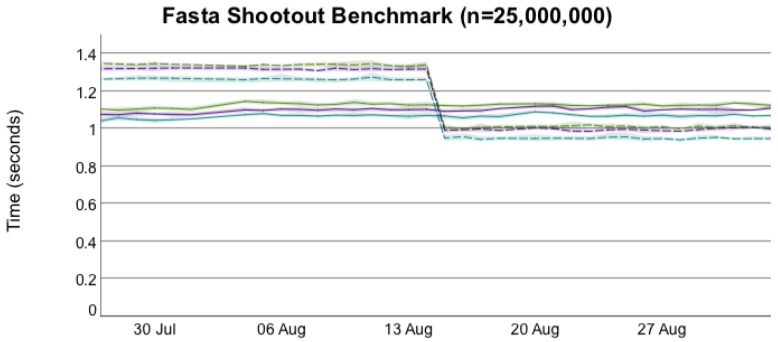
LLVM 4 Upgrade

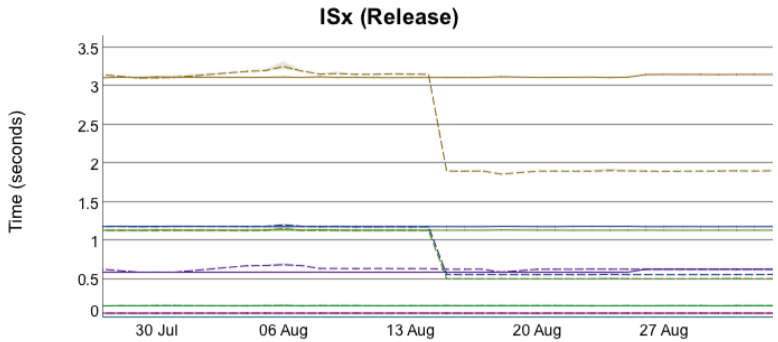
LLVM 4 Upgrade: This Effort and Impact

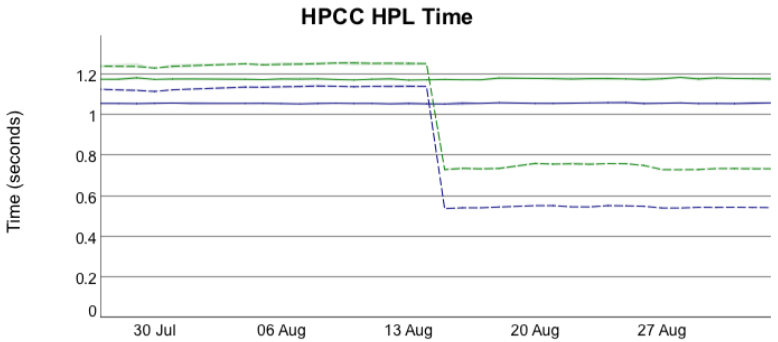
This Effort: Upgraded from llvm 3.7 to 4.0

Impact: Performance improvements for many benchmarks





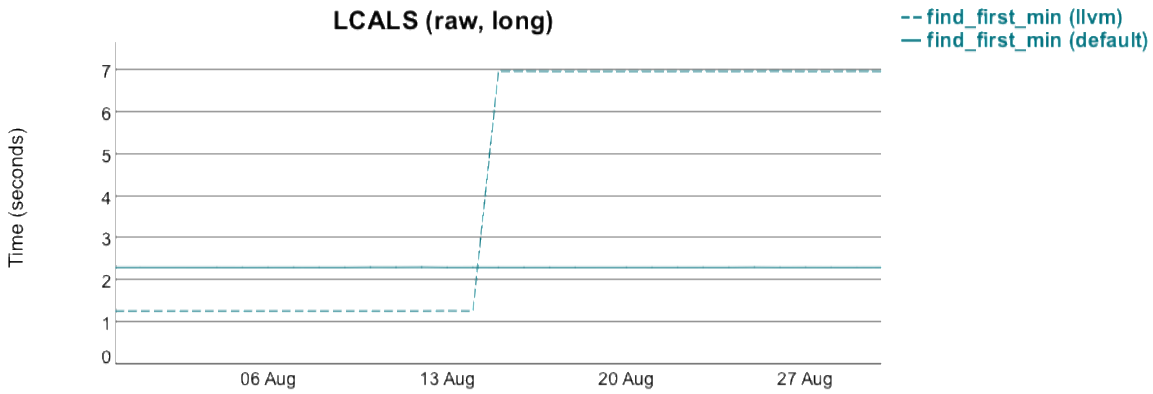




LLVM 4 Upgrade: Impact Cont.

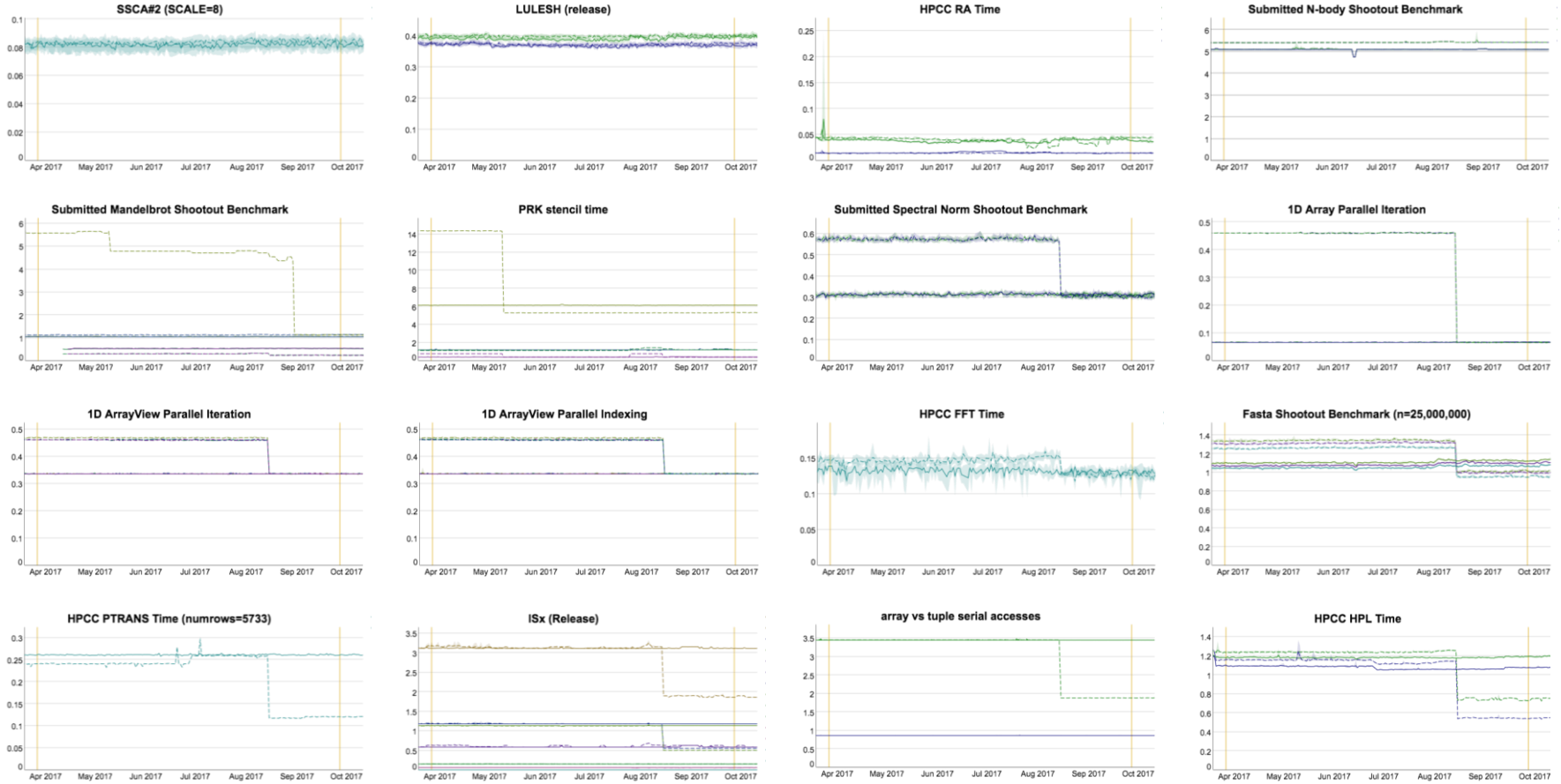
Impact: One surprising performance regression

- lcals find_first_min 6x slower (still need to investigate why)



LLVM: Summary

- Most benchmarks are now competitive with C back-end



LLVM: Status & Next Steps

Status: Ready to rely on LLVM more in Chapel project

- --llvm performance has generally improved
  - very competitive with the C back-end
  - significantly faster in some cases
- Chapel won't need internal clang headers with future versions of clang

Next Steps:

- Make --llvm the default back-end
  - Address any remaining performance differences first
- Remove uses of private clang headers
  - clang API has improved making these unnecessary in some cases
  - contributed a patch for clang 6 to enable us to remove the rest
- Complete ongoing efforts
  - Type-Based Alias Analysis Metadata improvements
  - Enabling --vectorize by default with --llvm

Partially Generic Resolution Changes

Partially Generic: Background

- Function resolution prefers 'where' clauses

```
proc f(x) where isIntegralType(x.type){
  writeln("f(x) where isIntegralType");
}
proc f(x) {
  writeln("f(x) generic)");
}
f(1);
// prints out “f(x) where isIntegralType”
```

Partially Generic: Challenge

- **What about this example?**

```
proc t1(A:[ ]) where true {
  writeln("t1(A:[ ]");
}
proc t1(A:[ ] int) {
  writeln("t1(A:[ ] int)");
}
var A:[1..100] int;
t1(A);
```

- **Before 1.16, produced an ambiguity error**
  - compiler added a 'where' clause to implement the ':[ ] int' argument
  - resulting in 2 fully generic overloads with where clauses
  - causing an ambiguity error in accordance with language specification

Partially Generic: This Effort

This Effort: Improved resolution rules

- Now prefer the ‘:[ ] int’ version
- More intuitive and in line with other resolution rules, namely:
  - existence of a 'where' clause is a last resort tie-breaker
  - arguments specifying a concrete type are preferred over generic arguments

Status: Implemented and documented in specification

Iterator Inlining Improvements

Iterator Inlining

Background: Could only inline iterators with a single yield

- Iterators that aren't inlined suffer significant performance penalty

This Effort: Permit iterators with multiple yields to be inlined

- For non-zippered iterators only
- Defaults to 10 yields, can be changed with --inline-iterators-yield-limit

Impact: More iterators can now be inlined

- Significantly improved performance of a user application
- No performance changes in our nightly suite
  - internal iterators have already been highly optimized

Next steps: Improve iterator inlining for zippered iterators

Dynamic Cast Optimization

Dynamic Casts

Background: Dynamic cast is a runtime check

- Check if a class instance has type that is a subtype of another type

```
class Parent { ... }
class Child : Parent { ... }
...
var p:Parent = ...;
var c = p:Child; // casts the object or returns nil if is not of that type
```

- Was implemented with a series of conditionals…

…that checked against every subtype

…so had O(# classes) code size, runtime complexity

This Effort: Improved dynamic cast to be constant time

Impact: Reduced generated code size in some cases

Other Compiler Improvements

Other Compiler Improvements

- Added support for #-based comments in '-f' configuration files
- Added --print-unused-functions to identify unused routines
- Improved CHPL_UNWIND output to include more functions
- Improved --print-callgraph output to include more functions
- Stopped heap-promoting local variables in on-clauses for 'qthreads'
- Removed support for the deprecated array alias '=>' operator
- Removed the --conditional-dynamic-dispatch-limit flag and feature
  - rely on existing dynamic-dispatch table instead

Error Message Improvements

Error Message Improvements

- improved 'const' checking
- extended --div-by-zero-checks to also check for modulus (%) 0 operations
- added an error message for exported functions with generic arguments
- improved error messages for illegal 'delete' statements
- removed checks that iterators must contain 'yield' statements
- added an error for records that try to subtype another type
- added a number of error messages for poorly formed (or unsupported) init()s
- improved the error message generated when closing a file before its channels
- added an error for returning a tuple of the wrong size
- improved an error message for bad forwarding calls to parallel iterators
- improved an error message about type mismatches between fields

Bug Fixes

Bug Fixes

- fixed a number of bugs related to initializers
- fixed a number of bugs related to error-handling
- fixed several bugs in the 'forwarding' feature for object fields
- fixed bugs in counting tasks and creating the right number of new tasks
- fixed bugs for several forall intent cases
- fixed a bug in which a qualified module reference was incorrectly shadowed
- fixed a bug in isAlpha() for characters between upper- and lowercase letters
- fixed a bug in bulk assignment for rank-change slices
- fixed a bug in variable deinitialization order
- fixed a bug in which 'use' statements were not considered in program order
- fixed a bug in which 'rmTree' would not remove directories with hidden files

More Bug Fixes

- fixed some bugs in loop invariant code motion (LICM)
- fixed a portability bug in padding years in the DateTime module
- fixed a bug in dead code elimination relating to local record types
- fixed a bug comparing floating point expressions on linux32
- fixed a bug in complicated type aliases
- fixed a bug in denormalization for '~' for small integers
- fixed a bug in which remote-value forwarding didn't handle dereferences well
- fixed a bug relating to scoped accesses to internal modules
- fixed bugs with parallel iteration over domains with non-natural alignment
- fixed a bug in the implementation of the &= operator for associative domains
- fixed a bug in applying 'reindex()' to an empty domain/array

Even More Bug Fixes

- fixed a bug in modules with just one non-initialization function declaration
- fixed a bug in dead code elimination for do-while loops
- fixed a bug in which isRecord*() returned 'true' for sync/single types
- fixed a bug related to task counters not being stored in task-local storage
- fixed a bug for ambiguous 'param' methods
- fixed a bug in 'fifo' tasking in which not enough threads were created

Legal Disclaimer

Information in this document is provided in connection with Cray Inc. products. No license, express or implied, to any intellectual property rights is granted by this document.

Cray Inc. may make changes to specifications and product descriptions at any time, without notice.

All products, dates and figures specified are preliminary based on current expectations, and are subject to change without notice.

Cray hardware and software products may contain design defects or errors known as errata, which may cause the product to deviate from published specifications. Current characterized errata are available on request.

Cray uses codenames internally to identify products that are in development and not yet publically announced for release. Customers and other third parties are not authorized by Cray Inc. to use codenames in advertising, promotion or marketing and any use of Cray Inc. internal codenames is at the sole risk of the user.

Performance tests and ratings are measured using specific systems and/or components and reflect the approximate performance of Cray Inc. products as measured by those tests. Any difference in system hardware or software design or configuration may affect actual performance.

The following are trademarks of Cray Inc. and are registered in the United States and other countries: CRAY and design, SONEXION, and URIKA. The following are trademarks of Cray Inc.: ACE, APPRENTICE2, CHAPEL, CLUSTER CONNECT, CRAYPAT, CRAYPORT, ECOPHLEX, LIBSCI, NODEKARE, THREADSTORM. The following system family marks, and associated model number marks, are trademarks of Cray Inc.: CS, CX, XC, XE, XK, XMT, and XT. The registered trademark LINUX is used pursuant to a sublicense from LMI, the exclusive licensee of Linus Torvalds, owner of the mark on a worldwide basis. Other trademarks used in this document are the property of their respective owners.

CRAY

THE SUPERCOMPUTER COMPANY