Vectorization of Chapel Code

Elliot Ronaghan, Cray Inc.

CHIUW @PLDI

June 13, 2015

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.

Vectorization: Background

- **Vectorization is crucial for achieving peak performance**
  - true for commodity and HPC systems
  - becoming increasingly important, particularly in HPC
    - AVX-512 (Xeon and Xeon Phi)
    - NEON (ARM)

- **Chapel relies on back-end compiler to auto-vectorize**
  - Chapel's primary back-end generates C code
  - C compilers are frequently thwarted by memory aliasing
    - must make conservative assumptions that inhibit auto-vectorization

Vectorization: Background (continued)

- **Chapel is well-suited for vectorization**
  - limited aliasing
  - support for array programing

    ```
    A = B + C;
    ```

  - parallelism is a first class citizen

    ```
    forall i in 1..10 do …
    ```

- **Need to convey Chapel semantics to back-end**
  - do not want to generate explicit vectorization
    - rather, convey when vectorization is legal
    - leverage back-end compilers' sophisticated and refined cost models

Vectorization: Background (continued)

- **Several recent efforts to help the back-end vectorize:**
  - Generate Chapel for-loops as C for-loops
  - Optimize anonymous range iteration
  - Annotate data parallel loops with vectorization pragmas
  - Currently exploring manual marking of vectorizable loops

C for-loops: Background

- **Chapel for-loops and C for-loops are different**
  - Chapel for-loops invoke iterators or iterate over data structures:

    ```
    for i in 1..10 do …
    ```

  - C for-loops are a specialized while-loop with init and incr clauses:

    ```
    for (i=1; i<=10; i+=1) …
    ```

  - Chapel for-loops are more powerful:

    ```
    for a in myArray do …
    for (a, j) in zip(myArray, 1..10) do …
    ```

- **Want Chapel for-loops to be generated as C for-loops**
  - this is the form back-end compilers are designed to optimize
  - required for attaching vectorization annotations
  - will result in clean and readable generated code

C for-loops: Background (continued)

- **Most for-loops are driven by ranges**
  - they either directly iterate over range
  - or a structure whose iterator forwards to a range iterator
    - e.g. arrays, distributions

```
for a in myArray do …              // iterate over an array
```

…is implemeted in terms of…

```
array.these() {
  for i in myDomain do             // array iterates over its domain
    yield dsiAccess(i);
}
```

…which is implemented in terms of…

```
domain.these() {
  for i in myRange do              // domain iterates over its range(s)
    yield i;
}
```

C for-loops: Background (continued)

- **Range iterators traditionally generated C while-loops**

```
for i in 1..10 do …                 // range iteration
```

generated:

```
i = first;
end = last + 1;
cont = (i != end);
while(cont) {                       // generated while loop
  tmp = (i+1);
  i = tmp;
  cont = (tmp != end);              // != relational operator
}
```

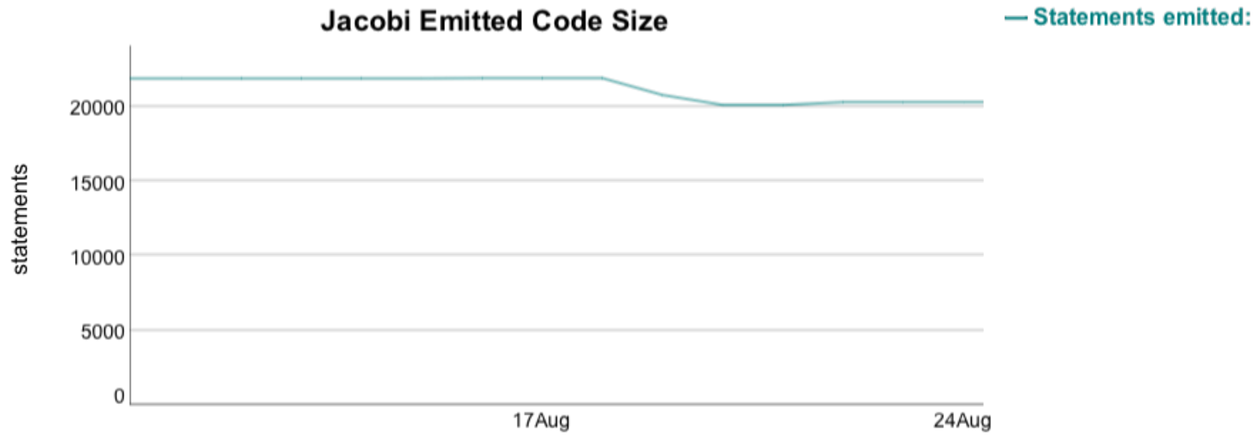
- not a loop that back-end compilers are designed to optimize
- not amenable to auto-vectorization or vectorization pragmas

C for-loops: This Effort

- **To generate most Chapel for-loops as C for-loops we now:**
  - generate range iterators using C for-loops
    - results in iterators that forward to ranges being generated as C for-loops
  - generate zippered iterators as C for-loops
    - because iterator inlining/lowering process is different for zippered iterators

C for-loops: Impact

- **Generated code improvements**
  - Decreased generated code size: ~22,000 => ~20,500 for Jacobi



  - Improved readability of generated code

```
for i in 1..10 do …
```

generates:

```
for (i = start; (i <= end); i += INT64(1))
```

C for-loops: Impact (continued)

- **Generated code for range iteration**

```
for i in 1..10 do …
```

previously:

```
i = start;
end = last + 1;
cont = (i != end);
while (cont) {
  tmp = (i+1);
  i = tmp;
  conttmp = (tmp != end);
  cont = conttmp;
}
```

now:

```
for (i = start; (i <= end); i += INT64(1))
```

C for-loops: Impact (continued)

- **Generated code for zippered array iteration**

```
for (a, b) in zip(A, B)
```

previously:

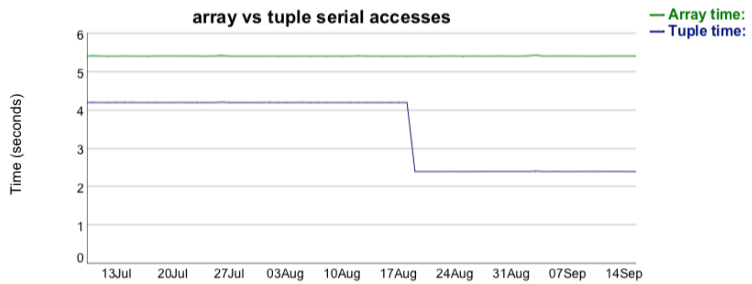
```
 for (;_cond;) {
    _ref_tmp_5 = &_ic__F6_i;
    *(_ref_tmp_5) += _ic__F4_step;
    tmp31 = (_ic__F6_i != _ic__F5_last);
    if (tmp31)
      _ic__more = INT64(1);
     else
      _ic__more = INT64(0);
    _cond = (_ic__more != INT64(0));
    _ref_tmp_6 = &_ic__F6_i2;
    *(_ref_tmp_6) += _ic__F4_step2;
}
```

now:

```
for (_ic_i = start1, _ic_i2 = start2;
(_ic_i <= _ic_last); ic_i += _ic_step, _ic_i2 += _ic_step2)
```

C for-loops: Impact (continued)

- **Performance Improvements**
  - not many changes in our nightly performance testing
    - couple cases like serial tuple accesses



  - however, we believe there are real world performance improvements
    - our nightly testing uses gcc 4.7
    - it seems to lack some optimizations that benefit from C for-loops
  - performed manual testing using gcc 4.9, intel, and cray compilers
    - used stream and simple vector addition
    - showed > 25% performance improvement in some cases
    - back-end compiler tools indicate more auto-vectorization occurring

C for-loops: Summary

- **Most Chapel for-loops now generate clean C for-loops**
  - However, previous generated code excluded range construction

```
for i in 1..10 do …
```

actually generates:

```
build_range(INT64(1), INT64(10), range);
start = range->low;
end = range->high;
for (i = start; i <= end; i += INT64(1))
```

- Want to eliminate construction when possible
  - such as when iterating over anonymous ranges

Anonymous Range Opt: Background

- **Anonymous ranges: those not stored in a named variable**
  - cannot be referenced elsewhere
  - commonly used directly in a loop

```
for i in 1..10 do
for i in lo..hi do
```

- **Ranges are implemented as records**
  - as a result, each range literal constructs a record
  - anonymous ranges are not captured and cannot be used again
    - so why waste time constructing them?

Anonymous Range Opt: This Effort

- **Eliminate construction for common anonymous ranges**
  - provide an optimized iterator when stride is known at compile time
  - eliminate cost of construction
  - allow back-end compiler to better optimize and auto-vectorize

- **This optimization occurs at parse time**
  - for-loop builder recognizes certain range patterns
  - replaces those with a direct range iterator
    - iterator takes low, high, stride as arguments
    - e.g., compiler replaces:

```
for i in 1..10 do
```

with:

```
for i in chpl_direct_range_iter(1, 10, 1) do
```

Anonymous Range Opt: Impact

- **Eliminates range construction for many common cases**

```
for i in 1..10 do writeln(i);
```

previously:

```
build_range(INT64(1), INT64(10), range);
start = range->low;
end = range->high;
for (i = start; i <= end; i += INT64(1))
  writeln(i);
```

now:

```
for (i = INT64(1); i <= INT64(10); i += INT64(1))
  writeln(i);
```

Anonymous Range Opt: Impact (continued)

- **Optimized iteration for strides known at compile time**

```
for i in 1..10 by 2 do writeln(i);
```

previously:

```
// function call to build range
// function call to apply 'by' operator to range
// function call and conditional check to see if range is ambiguous
// function call to compute the starting value
// conditional check to see if range is empty (e.g. 2..1)
// function call to compute the ending value
for (i = start; i != end; i += str)  // finally iterate, but using !=
  writeln(i);
```

71 SLOC

now:

```
for (i = INT64(1); i <= INT64(10); i += INT64(2))
  writeln(i);
```

Anonymous Range Opt: Impact (continued)

- **Better back-end optimization and auto-vectorization**
  - range construction and other checks obfuscate iteration pattern
  - we now propagate range literals directly to the C for loop
    - helps create cleaner vectorized code (eliminates some loop peeling)
    - allows compiler to better select unrolling factor and trip count

- **No major changes seen in nightly performance graphs**
  - not terribly surprising
    - most time spent in loop body, not prelude
    - not many benchmarks iterate over nested anonymous ranges
    - still lacked performance testing with modern vectorizing back-end compilers
      - have since started testing with the newest versions of Cray, GNU, Intel, and PGI

Anonymous Range Opt: Status

- **Cases that are currently handled**

```
for i in 1..10 do                              // works for simple ranges
for i in 1..10+1 do                            // works with expressions in ranges
var lo=1, hi=10; for i in lo..hi do            // works for variables
for i in 1..10 by 2 do                         // works for strided ranges
for (i, j) in zip(1..10, 1..10) do             // works for zippered iters
for (i, j) in zip(A, 1..10) do                 // following non-ranges also works
coforall i in 1..10 by 2 do                    // works for coforalls as well
```

- **Cases that are not handled**

```
for i in (1..) do                   // doesn't handle unbounded ranges
for i in 1..10 by 2 by 2 do         // doesn't handle more than 1 'by' operator
for i in 1..10 align 2 do           // doesn't handle 'align' operator
for i in 1..#10 do                  // doesn't handle 'count' operator
var r = 1..10; for i in r do        // not an anonymous range
forall i in 1..10 do                // does not get applied to foralls
```

Anonymous Range Opt: Next Steps

- **Handle additional cases**

  ```
  for i in 1..#10 // used frequently in leader and standalone iterators
  ```

- **Move optimization from parse-time to after resolution**
  - requires that resolution is moved before normalization
  - would allow us to handle more cases
    - …and not be so careful about preserving user errors
  - would allow us to anonymize named ranges used only for iteration

  ```
  var r = 1..10;
  if debugParam then writeln(r);    // common in our iterators
  for i in r do yield i;

  var r = 1..10;
  for i in r do A[i] = i;
  for i in r do A[i] = A[i%10+1]; // common in benchmarks & user code
  ```

Anonymous Range Opt: Summary

- **Most Chapel for-loops generate "ideal" C for-loop equivalent**

- **Can now focus on conveying Chapel semantics to back-end**
  - Remember that Chapel is well-suited for vectorization because
    - limited aliasing
    - support for array programing

      ```
      A = B + C;
      ```

    - parallelism, and especially data parallelism is a first class concept

      ```
      forall i in 1..10 do …
      ```

Data-par vectorization: Background

- **Data-parallel operations are vectorizable**
  - user asserts there are no data dependencies or ordering constraints

```
A = B + C;
forall i in 1..n do A[i] = B[i] + C[i];
forall (a, b, c) in zip(A, B, C) do a = b + c;
```

- **Data-parallelism implemented in terms of task-parallelism**
  - leader iterators create parallelism and assign work to followers
  - follower iterators serially do the chunk of work assigned by the leader
    - work assigned to followers should have no vector dependencies

Data-par vectorization: This Effort

- **Mark follower loops with '#pragma ivdep' in C code**
  - 'ivdep' tells the back-end compiler to ignore vector dependencies
    - each compiler has slightly different semantics for the pragma
- **'ivdep' permits back-end to ignore assumed dependencies**
  - iteration dependence, memory aliasing, etc.
  - back-end may unconditionally vectorize loops with potential aliases
    - instead of two loops with a runtime check to see if the vector version is safe
  - back-end can vectorize loops that it assumed were illegal before

Data-par vectorization: This Effort (continued)

- **Compiler approach for marking follower loops with ivdep**
  - mark yielding follower loops as order-independent during resolution
    - these are the loops that will execute the body of a forall loop
    - (others may do bookkeeping unrelated to the loop's forall semantics)
  - propagate order-independence during iterator lowering/inlining
    - loops that cannot be inlined are not order-independent
      - advance() function cannot be vectorized
    - a zippered iterator is order-independent iff all iterands are & they are inlined
  - if vectorization is enabled, annotate these order-independent loops
    - generate CHPL_PRAGMA_IVDEP, defined in the runtime for each compiler

- **Added extensive test suite**
  - uses a reporting mechanism to ensure correct loops are annotated
    - and other loops are not mistakenly annotated

Data-par vectorization: Impact

- **Many serial follower loops are annotated**

```
forall i in 1..10 do A[i] = i;
```

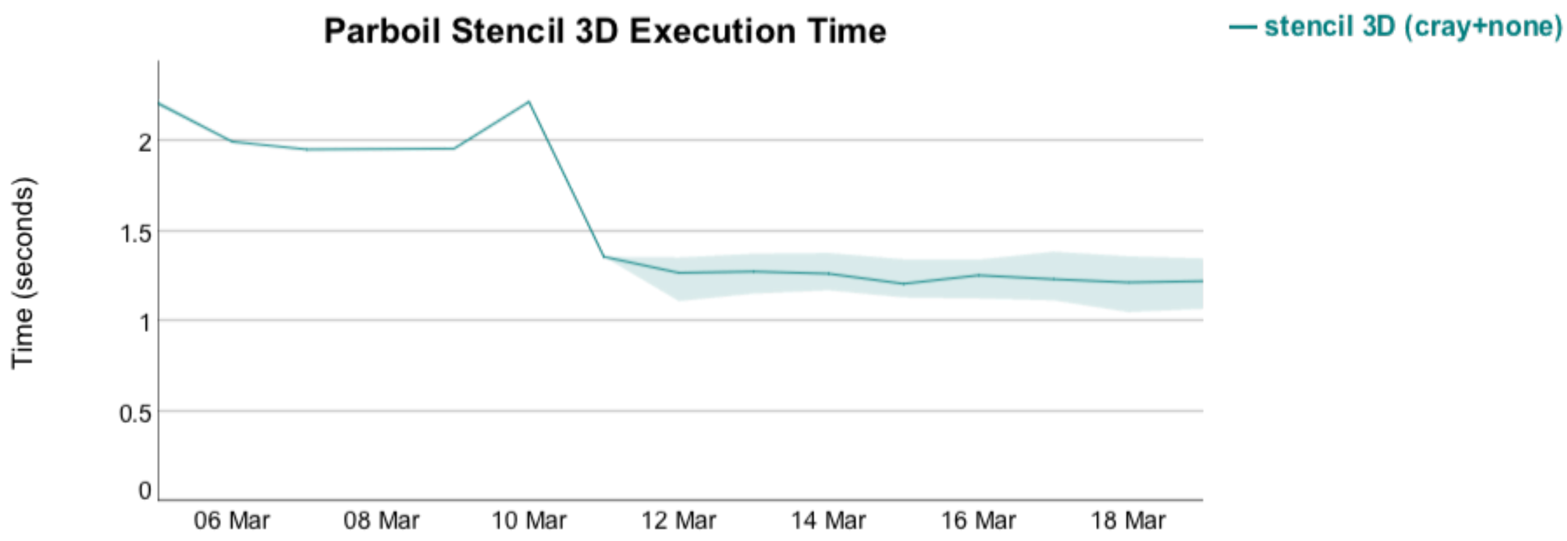
generates:

```
//~15 lines of follower setup
CHPL_PRAGMA_IVDEP
for (i = low; i <= high; i += INT64(1)) {
  call_tmp = (shiftedData + i);
  *(call_tmp) = i;
}
```

- **Improves vectorization of loops**
  - determined via back-end vectorization reporting output
    - fewer conditional checks at runtime
    - some previously non-vectorizable loops are now being vectorized

Data-par vectorization: Impact (continued)

- **Performance improvements**
  - 20% performance improvement of stream-ep on Intel KNC
    - runtime checks were more expensive on KNC vs. Xeon
  - improvements for benchmarks with complex array access patterns



Data-par vectorization: Status

- **Vectorization is enabled with the --vectorize flag**
  - automatically enabled with --fast
  - controls whether order-independent loops are marked with ivdep
    - will control more settings in the future (hence generic name)

- **Ran into issues with Cray as the back-end compiler**
  - ‘ivdep’ has slightly different semantics compared to other compilers
    - discovered late in release cycle
    - conservatively stopped annotating with ‘ivdep’ for Cray
    - additional work required to re-enable in appropriate cases

Data-par vectorization: Next Steps

- **Add more loop and vectorization benchmarks**
  - Livermore Compiler Analysis Loop Suite (LCALS)
    - (formerly Livermore Loops)
- **Add tests to inspect back-end vectorization reports**
  - to detect which loops are actually being vectorized
- **Start nightly performance testing on Xeon Phi**
- **Explore options with Cray compiler**
  - see what additional analysis we need to attach 'ivdep'

Data-par vectorization: Summary

- Many serial follower loops are annotated
- Improved vectorization of loops

Vectorization: Combined Impact

- Combined, these efforts greatly improve vectorization of Chapel code

```
forall i in 1..10 do …
```

1.9:

```
// ~75 lines of follower setup …
_build_range(fol.low, fol.high, &folRange);
// 4 fn calls to un-densify follower
_build_range(undenLow, undenHigh, &undenRange);
// 2 fn calls and conditional to compute start/end
while (test) {…}
```

102 SLOC

now:

```
// ~15 lines of follower setup …
low = fol.low, high = fol.high;
CHPL_PRAGMA_IVDEP
for (i = low; i <= high; i += INT64(1)) {…}
```

Vectorization: Next Steps

- **Let users provide vectorization hints on serial loops**
  - currently being worked on

```
for i in vectorizedIter(1..10) do …
```

- **Align memory allocations and generate alignment hints**
  - eliminate loop peeling, cleaner vectorization
- **Mark non-aliasing pointers with ‘restrict’ keyword**
  - perform alias analysis at Chapel level and annotate restricted pointers
    - Chapel has limited aliasing, this helps convey that to the back-end
    - should help with vectorization and other performance optimizations
- **Continue exploring other languages vectorization stories**
  - Does anyone have a good story?
    - Fortran? Julia? Intel’s ISPC?

Vectorization: Potential Next Steps

- **Investigate potential generated code improvements**
  - engage back-end compiler developers for recommendations
- **Explore what we can do with LLVM**
  - we may become constrained by what we can express in C
  - might be able to convey more Chapel semantics to LLVM back-end
- **Explore users need for more explicit vectorization support**
  - do we need to provide explicitly vectorized data structures & libraries?

Vectorization: Closing Thoughts

- **Vectorization has greatly improved with recent releases**
  - with no user code changes required
- **That said, we still plenty of work to do**
  - with several improvements already in the pipeline
- **We are extremely interested in any user feedback**
  - about our current and future vectorization roadmap
  - and about other programming models with good vectorization stories

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, URIKA, and YARCDATA. 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 CHAPEL

CRAY
THE SUPERCOMPUTER COMPANY

http://chapel.cray.com chapel_info@cray.com http://sourceforge.net/projects/chapel/