

CHAPEL 1.23 RELEASE NOTES: ONGOING EFFORTS

Chapel Team
October 15, 2020

OUTLINE

- Chapel Stabilization
- Ongoing Performance Efforts
- Ongoing Portability Efforts
- Targeting GPUs From Chapel

CHAPEL STABILIZATION (CHAPEL 2.0)

- [Overview](#)
- [Implicit ‘sync’ Reads/Writes](#)
- [‘in’, ‘out’, and ‘inout’ Intents](#)
- [Stabilizing Standard Libraries](#)

CHAPEL STABILIZATION OVERVIEW



CHAPEL STABILIZATION OVERVIEW

- **Chapel 2.0:** A forthcoming release in which the core language is stabilized to avoid breaking user code
- Chapel 1.23 contained very few breaking changes compared to other recent releases
 - addressed main issues noted in Chapel 1.22 release notes: POI, array initialization, init/deinit order, ...
 - main upgrade issue has been the need to ‘use’/‘import’ standard modules that had incorrectly been visible
 - contrast with:
 - Chapel 1.17: shifted from constructors to initializers
 - Chapel 1.18: switched classes to managed memory
 - Chapel 1.19: changed throw/catch to use ‘owned’ errors
 - Chapel 1.20: made classes non-nilable by default
 - Chapel 1.21: added support for split initialization and copy elision
 - Chapel 1.22: switched from 1-based implicit indices to 0-based
- **Next steps:**
 - take stock of where we are with language stability
 - focus increasingly on library stability



IMPLICIT SYNC READS/WRITES



IMPLICIT SYNC READS/WRITES

Background

Background:

- Since Chapel's inception, sync/single variables have supported implicit accesses:

```
var count$: sync int;  
count$ = count$ + 1; // equivalent to: 'count$.writeEF(count$.readFO + 1);'
```

- Rationale:
 - more convenient than requiring methods for every read/write
 - followed the precedent set by the Tera MTA / Cray XMT programming model
- However, this has also been a source of long-term concern:
 - unwitting reads/writes to such variables can cause deadlock
 - this led to the convention of naming sync/single variables with a '\$' to alert programmers to their presence
 - yet, it's arguably a red flag when a language depends so heavily on a naming convention like this for clarity
 - has also resulted in some asymmetries in the language, e.g.:

```
var x = y;           // in most cases, x.type == y.type  
var z = count$;    // here, z.type == int
```
- In considering aspects of the core language we might regret freezing as-is, this stood out

IMPLICIT SYNC READS/WRITES

This Effort, Status, and Next Steps

This Effort:

- Explored the impact of disabling implicit accesses
 - on our code base
 - on key user applications
- Generally, increased our confidence that removing implicit sync accesses would be a positive change
 - makes programmer's intent more explicit
 - removes special cases

Status:

- Failed to deprecate implicit sync accesses in time for Chapel 1.23.0
 - updating Chapel code base to avoid implicit reads/writes was straightforward, though laborious
 - compiler's reliance on implicit reads/writes was more complicated
 - particularly in the context of compiler-generated initializers

Next Steps: Barring significant objections, deprecate these accesses for the next Chapel release



IN, OUT, AND INOUT INTENTS



IN, OUT, INOUT INTENTS

Motivation

- This section considers how ‘in’, ‘out’, and ‘inout’ arguments might interact with ‘init=’
- Why is this an interesting topic?
 - These intents...
 - ...currently support implicit conversions in some cases but not others
 - ...have become more like initializing and returning variables over time, yet behave differently in mixed-type situations
 - These inconsistencies point to a potential language design problem
- There are several use cases that would benefit from user-defined implicit conversions:
 - types that wrap integers, e.g. ‘byteIndex’ and ‘codepointIndex’
 - converting from ‘strideable=false’ to ‘strideable=true’ ranges
- ‘init=’ already supports many patterns that seem similar to implicit conversion
 - yet it does not enable implicit conversions in argument passing
 - can we rationalize the current behavior, or is there something missing?



IN, OUT, INOUT INTENTS

Background

- A record can support initialization and assignment from another type with ‘init=’ and ‘=’:

```
record R {  
    var x: int;  
}  
  
proc R.init=(rhs: int) { this.x = rhs; }  
proc =(ref lhs: R, const ref rhs: int) { lhs.x = rhs; }  
  
var x: R = 1; // OK: runs R.init=(1)  
x = 2;          // OK: runs the '=' overload above
```

- Note that split initialization changes some cases that might look like ‘=’ calls into ‘R.init=’ calls:

```
var y: R; // y is not initialized here  
y = 2;    // y is split-initialized here with R.init=(int)
```

- The remainder of this discussion assumes the above definition of R



IN, OUT, INOUT INTENTS

'in' Intent: Background

- 'in' formal arguments are intended to be symmetric with variable initialization
- The compiler rewrites each 'in' intent as a variable initialization at the call site:

```
proc fIn(in arg: R) { }
```

fIn(something)

translates into

```
var intTemp = something;
fIn'(ref intTemp); // fIn' accepts argument by 'ref' and
// takes ownership of it
```

- 'in' formal arguments currently allow implicit conversions for built-in types:

```
proc fIntIn(in arg: int) { }
```

```
var myInt8: int(8) = 1;
fIntIn(myInt8); // currently allowed: performs implicit conversion from int(8) to int
```

```
...myInt8 + myInt64... // this '+' expression also relies on coercion due to its 'in' intent
```

IN, OUT, INOUT INTENTS

'in' Intent: Question

- Since this is allowed ...

```
var x: R = 1; // uses R.init=(int)
```

... should this also be allowed? [[issue 16576](#)]

```
proc fIn(in arg: R) {}  
  
fIn(1); // should it use R.init=(int) ?
```

- This call is not allowed today
- However, allowing it would:
 - make 'in' intent more similar to variable initialization
 - let 'init=' across types create an implicit conversion
- Note that the C++ problem solved by 'explicit' is avoided by having distinct routines for 'init' vs. 'init='



IN, OUT, INOUT INTENTS

'out' Intent: Background

- 'out' formal arguments are intended to behave similarly to returning
 - If we have ...

```
proc fReturn() {           proc fOut(out arg: R) {  
    return new R(1);       arg = new R(1);  
}  
}
```

... then the following 3 variable declarations behave similarly in terms of initialization:

```
var a = fReturn(); //no record copy  
var b: R;  
b = fReturn(); //no record copy due to split-init  
var c: R;  
fOut(c); //no record copy due to split-init via out intent
```



IN, OUT, INOUT INTENTS

'out' intent: Background

- We can think of the 'out' intent as creating a temporary variable at the call site and then assigning:

```
proc fOut(out arg: R) { ... }
```

fOut(c)

translates into

```
var outTmp: R;  
fOut' (ref outTmp);  
c = outTmp;
```

- Note that the 'out' intent uses '=' in some cases and 'init=' in others due to split init
- What if the 'out' formal has a different type from the actual argument?

```
proc int8Out(out arg: int(8)) { }  
var myInt: int = 1;  
int8Out(myInt); //should this call resolve?
```

- This call is not allowed today
- However, if it were allowed, it would have a clear interpretation:

int8Out(myInt)

translates into

```
var outTmp: int(8);  
int8Out' (ref outTmp);  
myInt = outTmp; //OK: setting `int` from `int(8)` is allowed
```

IN, OUT, INOUT INTENTS

'out' intent: Question

- If we have ...

```
proc iReturn() {  
    return 1;  
}  
  
proc iOut(out arg: int) {  
    arg = 1;  
}
```

... then the following are allowed

```
var a: R = iReturn(); // initializes 'a' with R.init=(int)  
  
var b: R;  
b = iReturn(); // split-init of 'b' through call to R.init=(int)
```

- Should the corresponding 'out' intent example also be allowed? [[issue 16582](#)]

```
var c: R;  
iOut(c); // Should this initialize c with R.init=(int) ?
```

- What about the 'int8Out(myInt)' example on the previous slide?
- The compiler could use the following reasoning when considering if 'iOut' is a candidate for the call:
 - observe that 'arg' is producing an 'int' value
 - check for availability of 'init=' and/or '=' to set the actual 'c' of type 'R' from an 'int' [see also [issue 15838](#)]

IN, OUT, INOUT INTENTS

'inout' Intent: Background

- 'inout' is intended to represent a combination of 'in' and 'out'

```
proc fInout(inout arg: R) { }
```

```
fInout(something);
```

translates into

```
var inTemp = something; // copy-initialize
fInout'(ref inTemp); // fInout' accepts by 'ref' and updates
something = inTemp; // write-back to original value
```

- What if the formal argument 'arg' and the actual argument 'something' have different types?
- Such calls are generally not allowed—but they are allowed for array slices
- If allowed, there would be at a minimum these two requirements:
 - Need to be able to copy-initialize to the formal type from the actual
 - so need something like 'proc FormalType.init=(arg: ActualType)'
 - Need to be able to assign to the actual from the formal
 - so need something like 'proc =(ref lhs: ActualType, rhs: FormalType)'

IN, OUT, INOUT INTENTS

'inout' Intent: Question

- If the record provides 'init=' and '=' to meet the two minimum requirements...

```
proc R.init=(rhs: int) { ... }

proc =(ref lhs: int, const ref rhs: R) { ... }
// and perhaps int.init=(R) - see issue #16582
```

... then should the following be allowed? [[issue 16554](#)]

```
proc fInout(inout arg: R) { }

var i = 1;
fInout(i);
```

- If allowed, it would have a clear interpretation:

```
fInout(i);
```



```
var inTemp: R = i; // copy-initialize using R.init(int)
fInout'(ref inTemp);
i = inTemp;          // write-back to i using =(int, R)
```

IN, OUT, INOUT INTENTS

Summary

- Should mixed-type ‘init=’ overloads indicate when implicit conversions are allowed?
 - Would allow the ‘init=’ to do the conversion for an ‘in’ intent argument
- Should implicit conversions be allowed for ‘out’ arguments?
 - when implicit conversion exists from the formal type to the actual
- Should implicit conversions be allowed for ‘inout’ arguments?
 - when an implicit conversion exists from the actual to the formal and back again



STABILIZING STANDARD LIBRARIES



STABILIZING STANDARD LIBRARIES

Background and This Effort

Background:

- Recent releases have focused on stabilizing the core language for a forthcoming Chapel 2.0 release
- However, breaking changes to the standard library can be equally frustrating for existing programs
 - e.g., having ‘writeln’ change its interface could be as impactful as changing from 1- to 0-based indexing
- For this reason, Chapel 2.0 should also commit to stabilizing a core set of the standard libraries
 - also need to review library-like features on built-in types
 - e.g., methods defined on ranges, domains, and arrays

This Effort:

- Enumerated a set of core standard libraries to prioritize for review
- Discussed approach and factors to consider in reviewing libraries
 - names of modules, routines, types, variables, arguments, ...
 - behaviors of features
 - etc.
- Identified an initial set of libraries to review as a team; and owners to lead discussions



STABILIZING STANDARD LIBRARIES

Status and Next Steps

Status:

- Started this review process toward the end of this release cycle
 - reviewed the ‘Time’ module as a team
 - shoring up core language features continued to consume most of our Chapel 2.0 cycles for this release
- ‘Time’ module confirmed our suspicions: even a modest-sized library can take a fair amount of time
 - though hopefully we’ll get more efficient as we go

Next Steps:

- Resolve open questions and complete action items identified in reviewing ‘Time’
- Start iterating through other modules regularly



ONGOING PERFORMANCE EFFORTS

- Aggregation Improvements
- Remote Cache Improvements

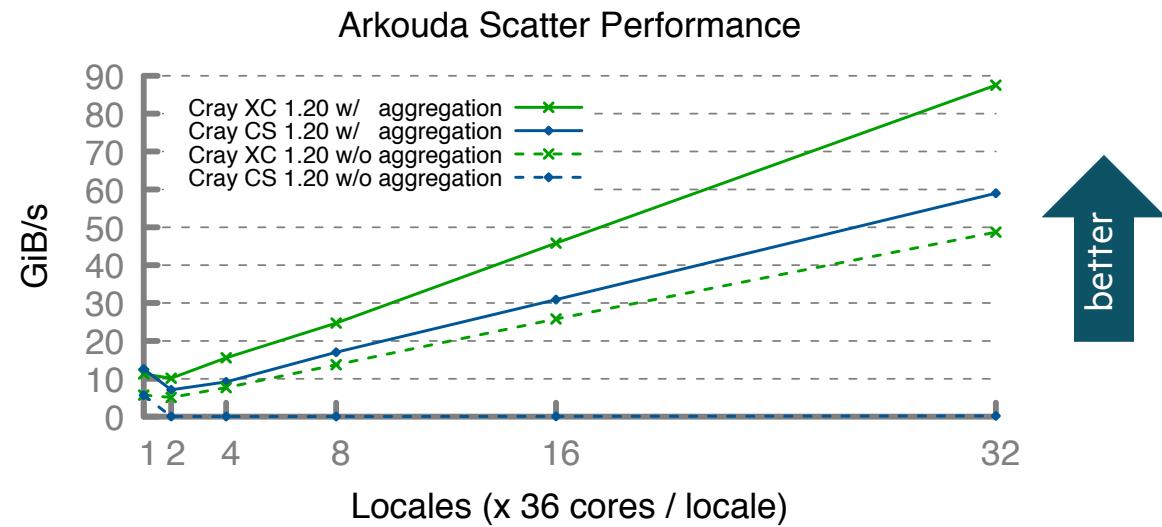
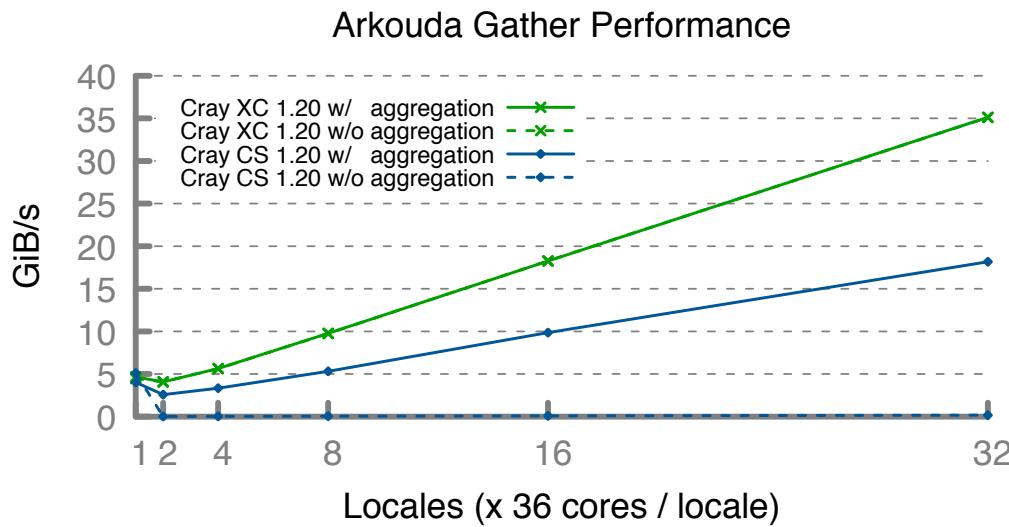
AGGREGATION IMPROVEMENTS



AGGREGATION IMPROVEMENTS

Arkouda Background

- Copy aggregators were added to Arkouda in the 1.20 timeframe
 - Significantly improved the performance of fine-grained operations
 - For 32-node Cray CS (FDR InfiniBand) and 32-node Cray XC (Aries):
 - 100x speedup for index gathers (remote GETs) on CS, slower on XC so not used
 - 300x speedup for index scatters (remote PUTs) on CS, 80% speedup on XC



AGGREGATION IMPROVEMENTS

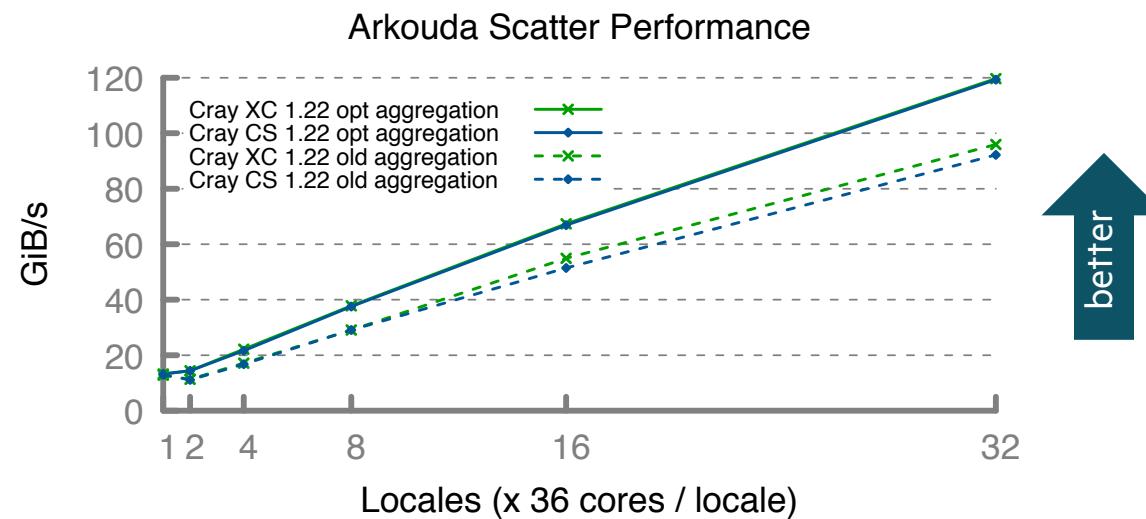
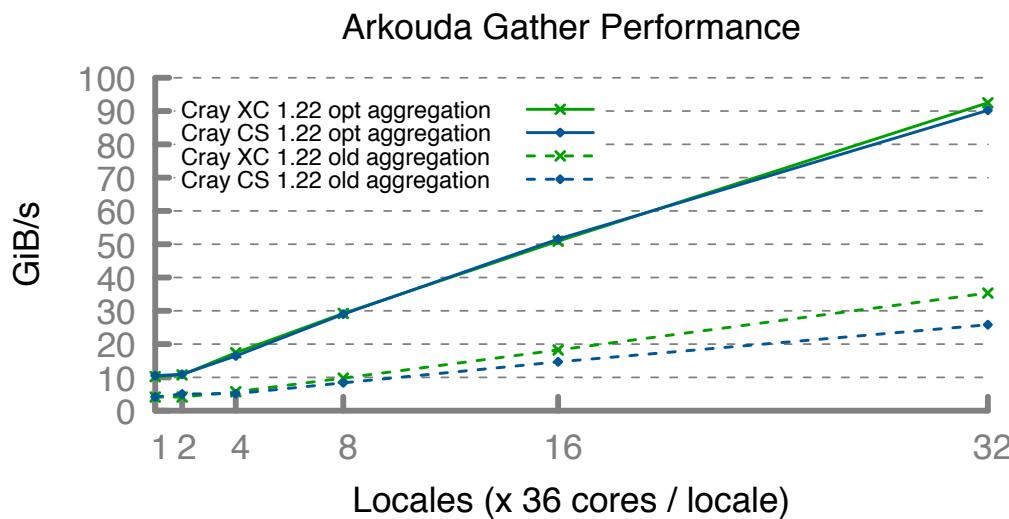
Arkouda Effort and Impact

This Effort: Further tuned aggregator performance, enabled GET aggregator for XC

- Cached remote allocations, optimized bulk transfers, and improved yield frequency

Impact: Significant performance improvements, performance parity between CS and XC

- 3.5x speedup for index gathers on CS, 2.5x speedup on XC
- 30% speedup for index scatters on CS, 25% speedup on XC



AGGREGATION IMPROVEMENTS

Arkouda Status

- Believe we're approaching performance limit for this aggregation implementation
 - However, hard to gauge without a reference implementation to compare against
 - To that end we wanted to compare aggregation performance against Bale



AGGREGATION IMPROVEMENTS

Bale Background

- Bale is a collection of mini-applications written in UPC/SHMEM
 - Tests various communication idioms and patterns
 - Histogram (stresses remote atomics)
 - Indexgather (stresses remote GETs)
- Bale also contains aggregated communication libraries
 - Exstack – synchronous single-hop aggregators
 - Conveyors – supports multiple implementations, including asynchronous multi-hop aggregators
- Previously ported and optimized elegant Chapel versions of Histogram and Indexgather
 - But had not looked at aggregated versions



AGGREGATION IMPROVEMENTS

Bale Effort

- Wrote aggregated version of Indexgather
 - Copy aggregators require minor code changes but provide significant performance speedups

// Intuitive indexgather

```
forall (t, r) in zip (tmp, Rindex) do
    t = A[r];
```

// Aggregated indexgather

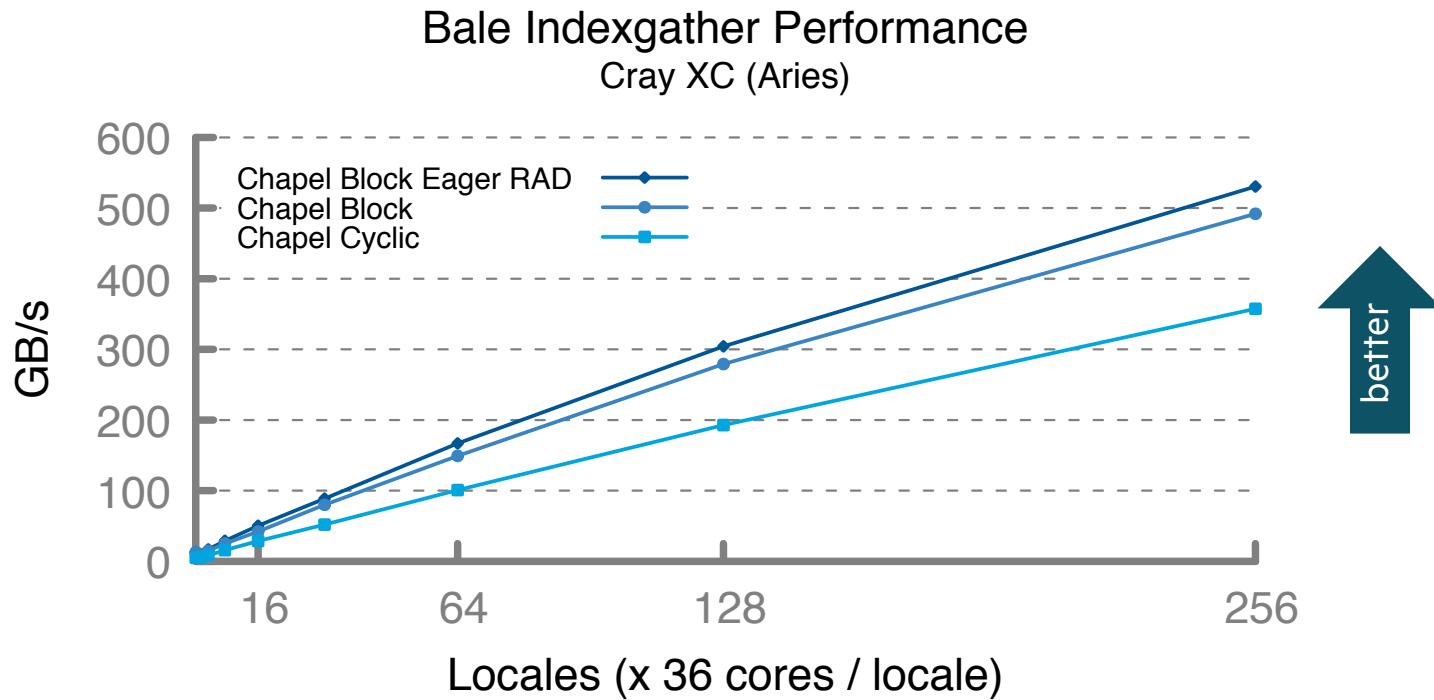
```
forall (t, r) in zip (tmp, Rindex) with (var agg = new SrcAggregator(int)) do
    agg.copy(t, A[r]);
```



AGGREGATION IMPROVEMENTS

Bale Status

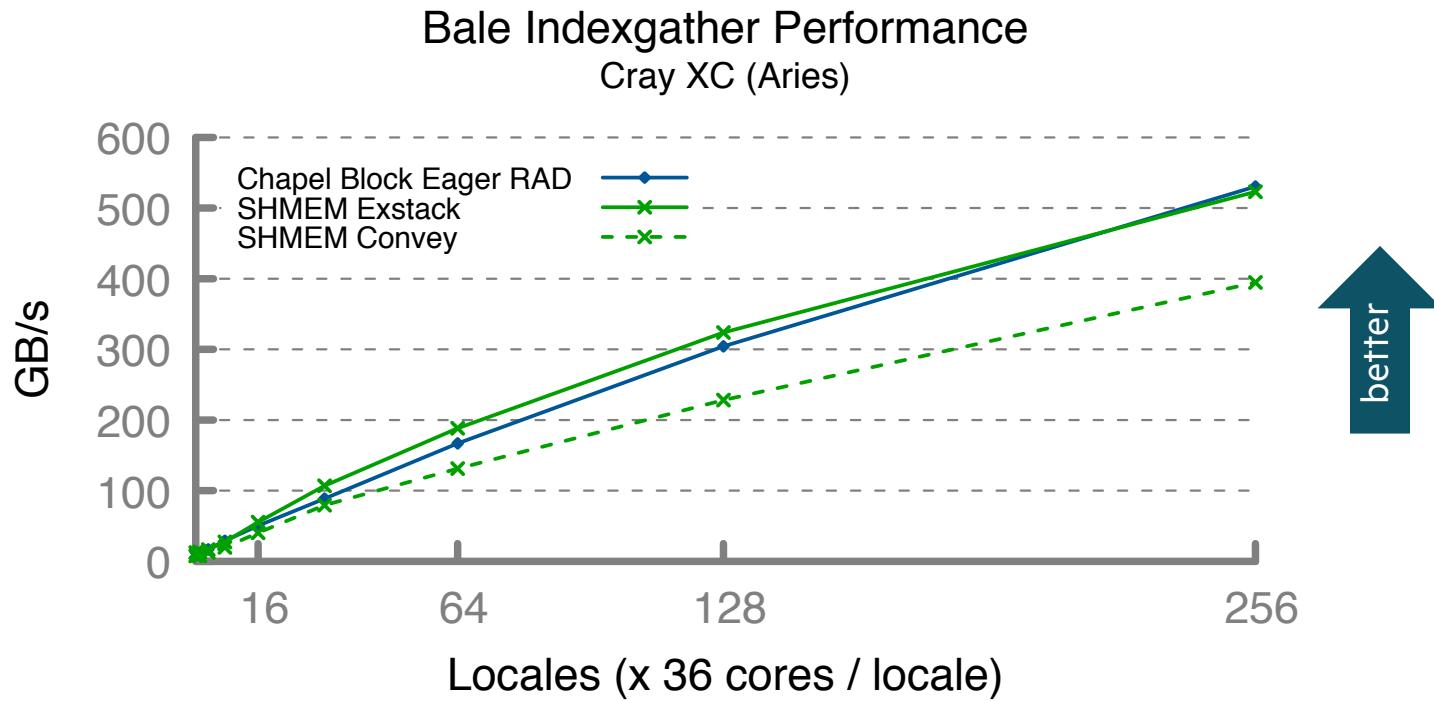
- Identified local overheads in distributed array indexing
 - Cyclic overhead is most dramatic, but Block has overhead too
 - Can be mitigated with eager remote-access-data (RAD) optimization, at the expense of slower array creation time



AGGREGATION IMPROVEMENTS

Bale Status

- Performance is competitive with reference SHMEM
 - On par with Exstack with less memory overhead
 - Ahead of Conveyors, though we expect to fall behind for non-trivial applications and at higher scale



AGGREGATION IMPROVEMENTS

Next Steps

- Continue to tune/improve performance
 - Improve local overheads for indexing into distributed arrays
 - Optimize for additional networks: newer InfiniBand, Ethernet, Amazon EFA
 - Explore auto-tuning to choose aggregation buffer sizes and other parameters for specific hardware/networks
- Migrate aggregation to Chapel's standard library
 - Support arbitrary aggregators, not just copy aggregators
 - Draw upon previous work with external [Chapel Aggregation Library](#)
- Use Bale as a mechanism to help explore aggregation
 - Provides non-trivial use cases, and a reference implementation with which to compare performance
- Add a compiler optimization to automatically use aggregation



REMOTE CACHE IMPROVEMENTS



REMOTE CACHE IMPROVEMENTS

Background and This Effort

Background:

- Chapel has a cache for remote data that can be enabled with --cache-remote
- Since 1.22, it has been stable enough to recommend to users
- In a few benchmarks, --cache-remote was adding significant overhead

This Effort: Reduced --cache-remote overhead in the worst cases

- Fixed comm/compute overlap under ‘ugni’ to address performance problems with SSCA2
- Evaluated performance and memory overheads at scale (512 nodes on an XC)
 - RA-on, RA-atomics, ISX, stream, and stencil showed no performance issues
 - RA-rmo had a 15% performance hit
- Developed a prototype that reduces data structure overhead at scale
 - RA-rmo has only a 9% performance hit with the prototype



REMOTE CACHE IMPROVEMENTS

Status, and Next Steps

Status: Close to enabling --cache-remote by default

- Some benchmarks have significantly better performance with the cache
 - 4x improvement for HPL
 - 3x improvement for miniMD
 - 20x improvement for PTRANS
- Most benchmarks have about the same performance when enabling the cache
- The ~10% overhead of RA-rmo represents the worst case we've seen
 - not particularly surprising: random access is the worst cache access pattern; fastest network is most sensitive to overhead
- Changes addressing RA-rmo performance at scale did not make it into the 1.23 release

Next Steps: Enable --cache-remote by default

- Finalize and merge change reducing data structure overhead to benefit RA-rmo at scale
- Continue to track and resolve performance issues
- Enable non-blocking comms in ugni
 - expect this to provide roughly 30% performance improvement for RA-rmo based on experiments with GASNet

ONGOING PORTABILITY EFFORTS

- 'ofi' Communication Layer
- Chapel on HPE Cray EX Systems



'OFI' COMMUNICATION LAYER

Background and This Effort

Background:

- 'ofi' comm layer had passed functional testing on a wide variety of system networks
- Neither stability nor performance were where we wanted them to be

This Effort:

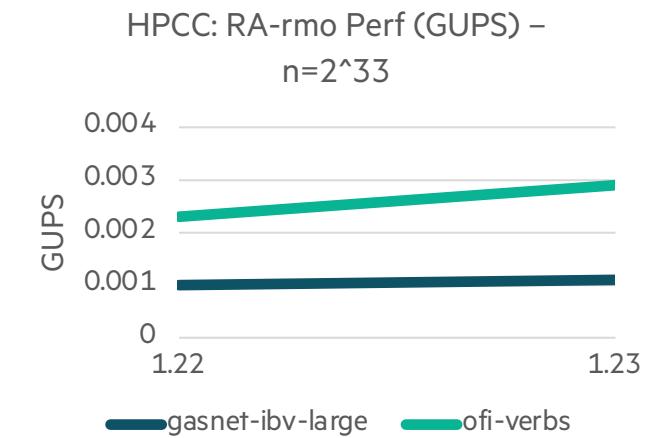
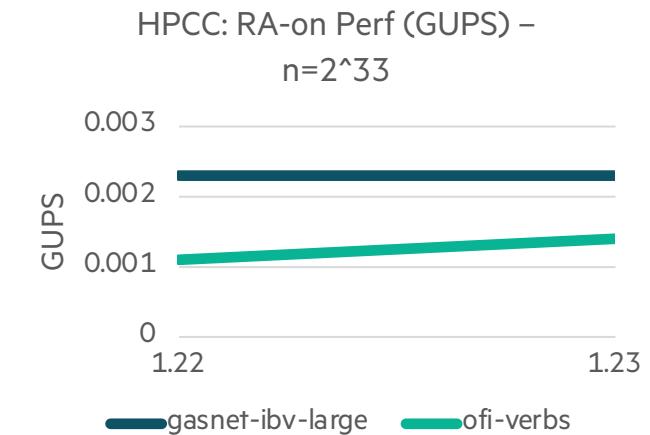
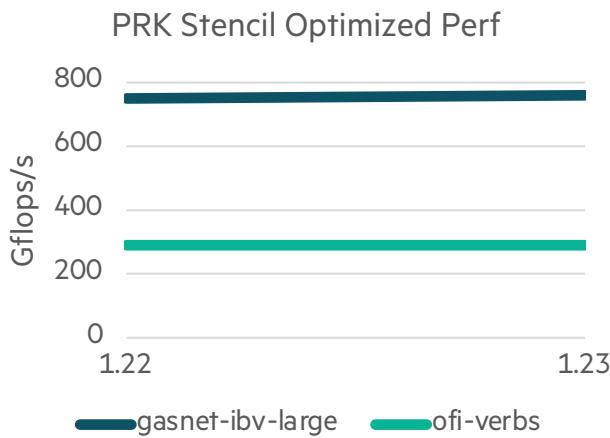
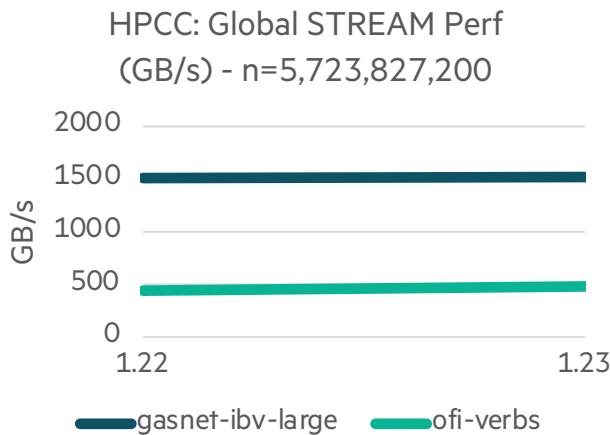
- Made conformance with the Chapel Memory Consistency Model (MCM) more principled
- Adjusted provider selection to more often find the highest-performing one that can achieve correctness
- Improved portability
 - EX system with dual Rome-64 processors and InfiniBand (found libfabric verbs scalability problem, fixed upstream)
 - AWS, with the high-performance EFA network
- Added a bundled third-party libfabric for systems without a system version
- Improved performance



OFI COMM

Performance

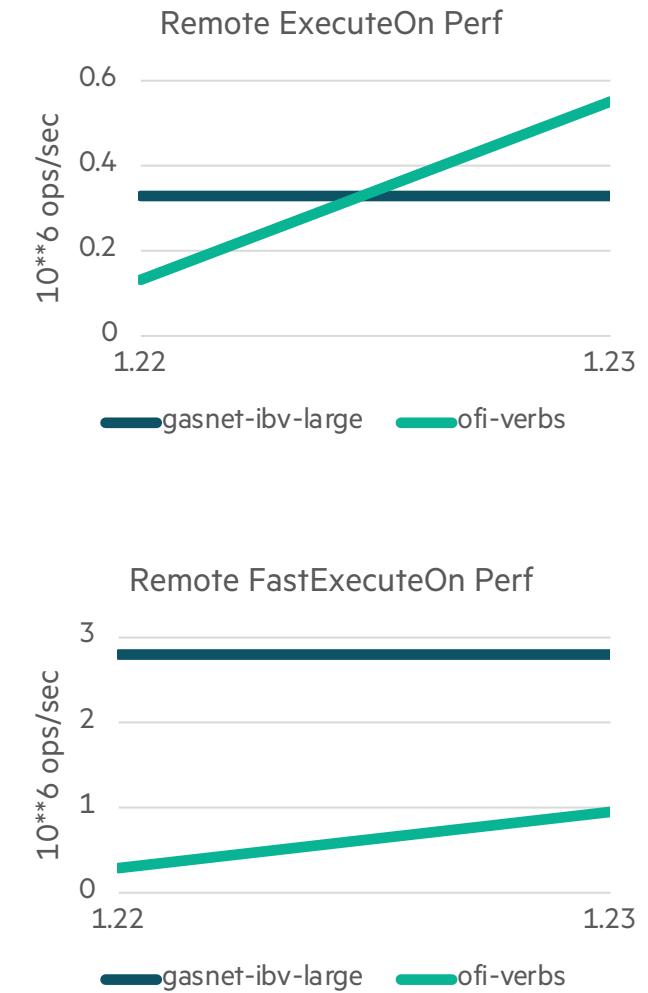
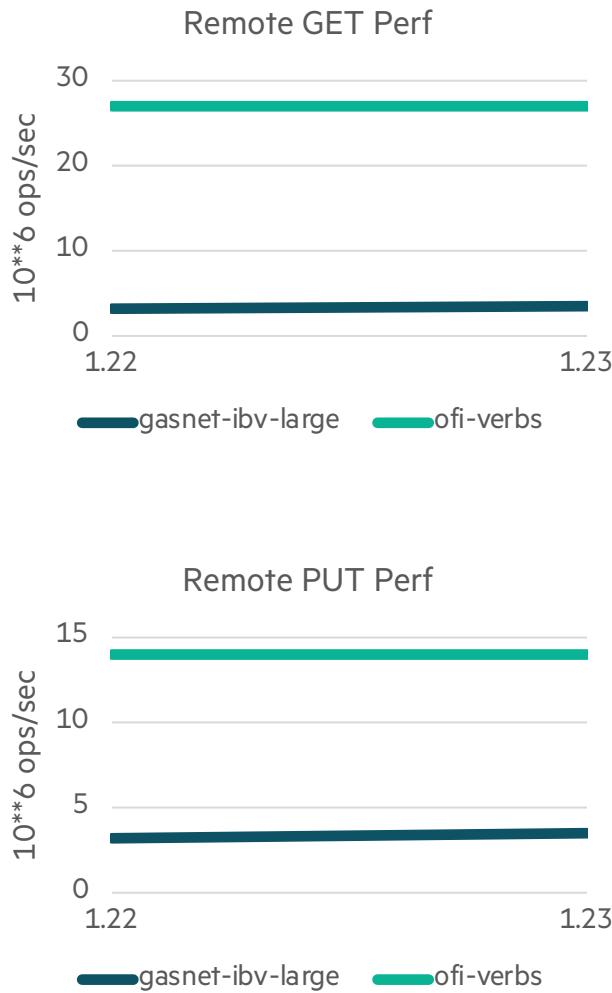
- 16-node Cray CS, InfiniBand network



OFI COMM

Performance

- 16-node Cray CS, InfiniBand network



Status:

- Stable, doesn't lack any core functionality
- Performance has improved but still needs work

Next Steps:

- Productization, so we can stop saying "work in progress"
- Add testing on more systems and networks, beyond just IB systems using the verbs provider
- Continue to improve performance





CHAPEL ON HPE CRAY EX SYSTEMS

CHAPEL ON HPE CRAY EX SYSTEMS

Background:

- Have ensured Chapel continues to work on EX as the systems evolve

This Effort:

- Rebranded: Shasta (informal name) → HPE Cray EX (official product name)
- Made changes to track COS (Cray Operating System for EX) and module system changes
- Expanded module configurations nearly to parity with XC, except lacking comm=gasnet

Status:

- Chapel continues to improve on EX, becoming more product-like

Next Steps:

- Add comm=gasnet configurations
- Adjust module file(s) to integrate with new Lua-based module system
- Continue tracking EX product evolution





TARGETING GPUS FROM CHAPEL

TARGETING GPUS FROM CHAPEL

- We are interested in supporting GPU programming from Chapel
 - GPUs are very common, yet challenging to program
 - GPU support is frequently asked about at Chapel presentations
 - it would improve upon Chapel's "any parallel algorithm on any parallel hardware" theme
- There have been several efforts relating to GPU support in Chapel
 - [early work](#) at UIUC
 - collaboration with AMD [\[1\]](#) [\[2\]](#) [\[3\]](#)
 - recent work from Georgia Tech and ANU, featured at [CHIUW 2019](#) and [CHIUW 2020](#)
 - meanwhile, users have run on GPUs via Chapel interoperability features (e.g., Inria Lille's branch & bound work)
- Our team's efforts started in earnest during this release cycle (yet are still at an early stage)
 - getting experience wrapping GPU libraries
 - considering how to compile Chapel to GPUs



CHAPEL INTERFACE TO CUBLAS



CHAPEL INTERFACE TO CUBLAS

Background and This Effort

Background:

- cuBLAS is a library of basic linear algebra subroutines (BLAS) optimized for GPUs
- Chapel's C interoperability simplifies interfacing with the cuBLAS C API and CUDA C runtime

This Effort:

- Initial effort to use GPUs with Chapel
- Wrap a part of the CUDA runtime and cuBLAS library with Chapel



CHAPEL INTERFACE TO CUBLAS

Example: SAXPY ($y[i] = a*x[i] + y[i]$)

- Initialize variables normally

```
use cuBLAS;

var N = 10: int(32);
var a = 2: real(32);

var X: [1..N] real = 3.0;
var Y: [1..N] real = 5.0;
```

- Function to copy data from CPU to GPU

```
var gpu_ptr_X = cpu_to_gpu(c_ptrTo(X), c_sizeof(real)*N:size_t);
var gpu_ptr_Y = cpu_to_gpu(c_ptrTo(Y), c_sizeof(real)*N:size_t);
```

- cuBLAS wrapper

```
cu_saxpy(cublas_handle, N, gpu_ptr_X, gpu_ptr_Y, a);
```

- Function to copy data from GPU to CPU

```
gpu_to_cpu(c_ptrTo(Y), gpu_ptr_Y, c_sizeof(real)*N:size_t);
```

-Note: Function names (e.g. `cpu_to_gpu` and `cu_saxpy`) and cuBLAS module name may change in the future.

CHAPEL INTERFACE TO CUBLAS

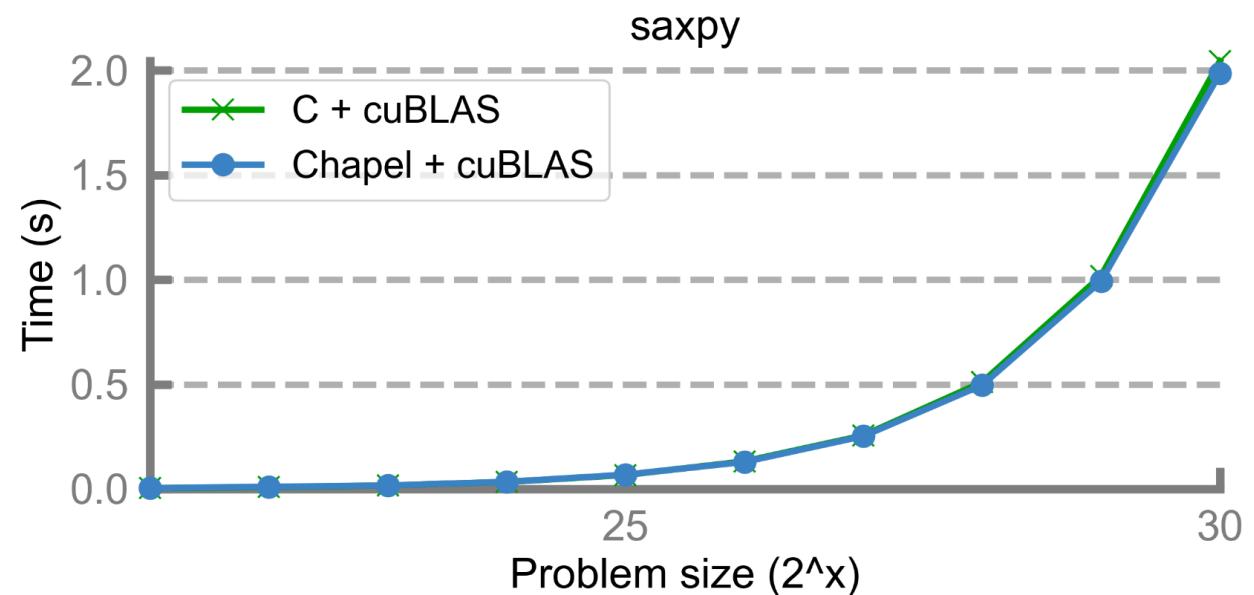
Status and Next Steps

Status:

- No runtime performance difference when using cuBLAS from C vs. Chapel
- Currently level-1 cuBLAS wrappers
(scalar-vector operations) are implemented

Next Steps:

- Implement cuBLAS wrappers for
level-2 (matrix-vector) and
level-3 (matrix-matrix) operations



A wide-angle photograph of a mountainous landscape. In the foreground, a calm lake reflects the surrounding environment. A cluster of trees with vibrant yellow autumn leaves stands on a small island or peninsula in the center-left of the lake. To the right, a dense forest of green and yellow trees stretches across the middle ground. In the background, a range of mountains is visible, with the most prominent peak being Mount Moran, which is heavily covered in snow. The sky is filled with dramatic, dark clouds.

COMPILING FOR GPUS (AND VECTORIZATION)

COMPILING FORGPUS (AND VECTORIZATION)

Introduction

- Vectorization and compiling for GPUs have many overlapping concerns
- This section will discuss how the Chapel language might be updated to support GPUs and vectorization
 - It makes 7 proposals that combine to improve the situation





VECTORIZATION IMPLEMENTATION UPDATE

VECTORIZATION UPDATE

Background:

- LLVM's compilation strategy supports detailed hinting about vectorizability
 - in the form of loop and memory access hints
- Experimental integration with the Region Vectorizer (RV) provides outer-loop vectorization
 - providing significant performance improvements, including 4–6x for a Clenshaw benchmark
 - However, correctness improvements to vectorization in 1.19 prevented RV vectorization for the Clenshaw benchmark

This Effort:

- Revisit LLVM vectorization hinting strategy to vectorize in more cases
 - Encountered problems with order-independence of RandomStream follower loops (see later section)

Impact:

- Vectorization hinting to the LLVM Loop Vectorizer is more likely
- Clenshaw benchmark once again vectorizes with RV

Next Steps: Resolve language design questions brought up by this effort





COMBINING GPU SUPPORT AND VECTORIZATION

COMBINING GPUS AND VECTORIZATION

Types of Vectorization

There are 2 different types of vectorization that will be supported by the Chapel compiler:

1. Automatic vectorization for any loop where the compiler can prove it will not impact correctness
 - Already reasonably well-supported by C compilers and by the stock LLVM loop optimization pass
 - Applies to inner loops, including those that come about from iterator inlining
 - Most reductions will disable vectorization (only a few are recognized, gives up for others)
 2. Vectorization that occurs with programmer input
 - Applies to parallel constructs like ‘forall’ / ‘vectorizeOnly’ including as outer loops
 - Requires features like ‘reduce’, ‘in’ and ‘var’ intents to create per-vector-lane operations
-
- We expect to continue to provide (1)
 - This section is focused on (2)



COMBINING GPUS AND VECTORIZATION

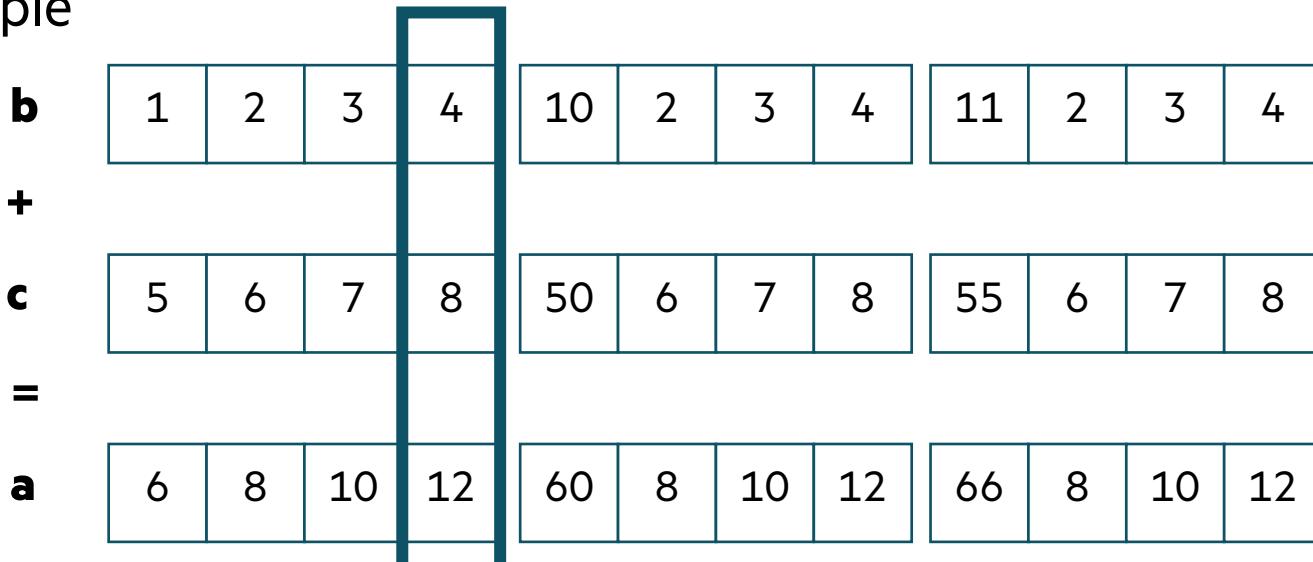
SIMD vs SIMT

- A GPU has some similarities with a CPU with an advanced vector unit
 - A SIMT platform like a GPU is not so far off from a SIMD-style instruction set supporting
 - masking out vector lanes (to handle branches in the vectorized code)
 - gather and scatter (to handle loads and stores in the vectorized code)
- In SIMT-style programming, each “task” has a separate thread of control
- SIMD-style programming only has one thread of control
- It is possible to start from a SIMT-style programming model and then convert it to SIMD
 - SIMT is arguably a more platform-independent strategy
 - tools exist that do this, e.g. Intel’s Implicit SPMD Program Compiler or an OpenCL compiler targeting CPUs



COMBINING GPUS AND VECTORIZATION

SIMD vs SIMT: Example



SIMD

```
for i in 0..12 by 4 {  
    var a, b, c: vec4real;           "vector lane"  
    b = simd_load_4real(B[i]);  
    c = simd_load_4real(C[i]);  
    simd_add_4_real(a, b, c);  
    simd_store_4real(A[i], a);  
}
```

SIMT

```
proc kernel(ref A, B, C) {  
    var i = getMySimtIndex();  
    A[i] = B[i] + C[i];  
}  
  
run_kernel(kernel,  
          0..12,  
          workgroupSize=4,  
          A, B, C);
```

COMBINING GPUS AND VECTORIZATION

Unifying

- **Proposal 1:** Enable a unified programming model for vectorization and for GPUs
- Depending on the target, the compiler will translate the same code into
 - vectorized code for execution on a CPU
 - a GPU kernel for execution on a GPU
- The programming model for these two will be substantially similar
- (Platform-specific Chapel code will still be needed in some cases for performance reasons)



COMBINING GPUS AND VECTORIZATION

Global-view

- **Proposal 2:** Enable vectorization/GPU execution for global-view programs
- Global-view programming is important to Chapel's productivity

$A = B + \alpha * C$

- Global-view programming includes
 - promoted operations over arrays
 - forall-intents to create “task-local” variables: ‘in’ and ‘var’ intents
 - forall ‘reduce’ intent
- Chapel programmers need to be able to use these features without disabling vectorization/GPU support



COMBINING GPUS AND VECTORIZATION

SIMT

- **Proposal 3:** Support SIMT-style programming to keep flexibility
- SIMT-style programming allows low-level code to be added within a global-view context
 - expecting that a ‘forall’ will become a GPU kernel
 - GPU-specific SIMT code can be added within the ‘forall’ in the SIMT model
 - compare with the SIMD vector code example where the loop invoking the SIMD code needed to be modified
 - this property would also help with distributed memory programming [[issue 14405](#)]
- SIMT-style also allows one to use GPU features like ‘shared memory’ within a block (CUDA terms)
 - a.k.a. ‘local data store’ within a workgroup (OpenCL terms)
- Being able to write SIMT code within a larger parallel loop allows one to fine-tune performance
 - imagine a science application written with global-view programming
 - a performance engineer can update portions of it but should not need to restructure everything



TASK IDS



TASK IDS

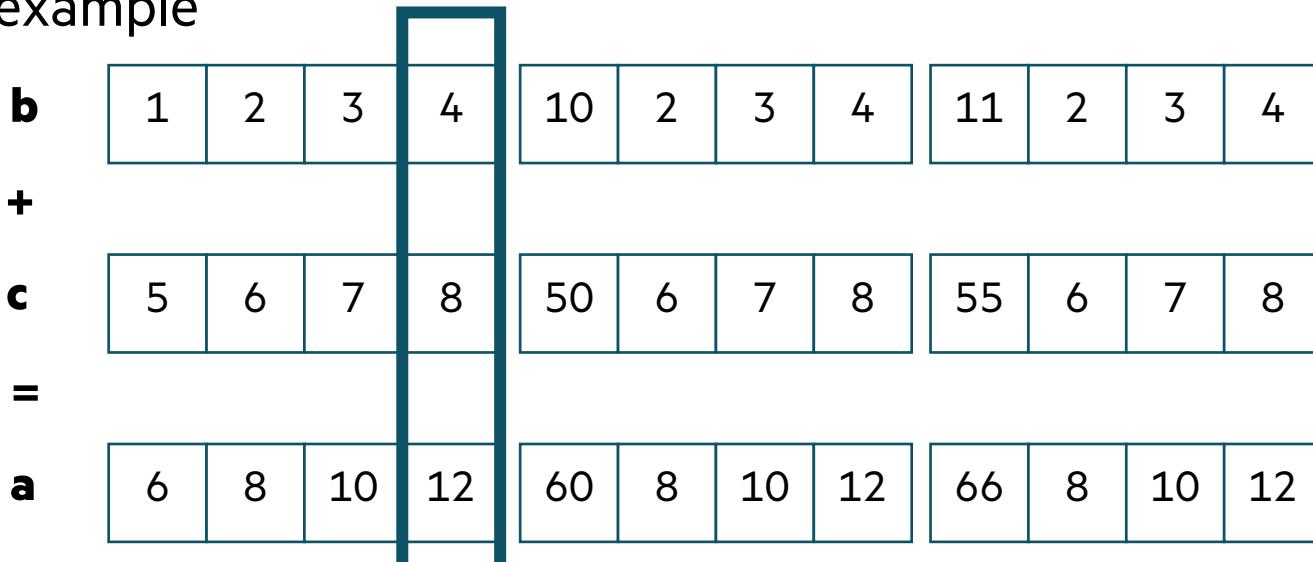
Motivation

- One of the key elements of SIMD programming is that there is a way to get some form of thread index
 - Along with this, GPU programming models also allow one to query the space being iterated
-
- Helps with smoothing boundary between forall and coforall+on in distributed programs [[issue 14405](#)]
 - Important for GPU features like ‘shared memory’ within a block (CUDA terms) [[chip 17](#)]
 - a.k.a. ‘local data store’ within a workgroup (OpenCL terms)



TASK IDS

Example from SIMD example

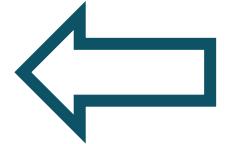


SIMD

```
for i in 0..12 by 4 {  
    var a, b, c: vec4real;           "vector lane"  
    b = simd_load_4real(B[i]);  
    c = simd_load_4real(C[i]);  
    simd_add_4_real(a, b, c);  
    simd_store_4real(A[i], a);  
}
```

SIMT

```
proc kernel(ref A, B, C) {  
    var i = getMySimtIndex();  
    A[i] = B[i] + C[i];  
}  
  
run_kernel(kernel,  
          0..12,  
          workgroupSize=4,  
          A, B, C);
```



TASK IDS

Distributed Parallel Iterators

- A typical distributed parallel iterator looks like this:

```
iter myiter(... tag=standalone...) {
    coforall loc in targetLocales do on loc {
        const numChunks = _computeNumChunks(...);
        coforall t in 1..numChunks {
            const part = computePartforChunk(...);
            order-independent-for i in part {
                yield f(i);
            }
        }
    }
}
```

- This iterator has 3 portions:

- Divide work among locales ('coforall loc ... on loc')
- Divide work among tasks ('coforall t in 1..numChunks')
- Indicate the work per task ('order-independent-for i in part')



TASK IDS

Forall loops and vectorization

- **Proposal 4:** Provide ways for code executing within a forall loop to discover task & vector division
- What information is needed?
 - query current vector lane and number of vector lanes
 - i.e., for GPU programming, the thread number among threads in the block and the block size
 - query the launched task number and the number of tasks that were launched
 - (potentially, similar information for number of locales used)
- It is important for some use cases that the task division be repeatable
 - i.e. two different ‘forall’ loops create the same number of tasks - see [\[issue 14405\]](#)
- It is also easier in many cases if the vector lanes can be multidimensional
 - so that e.g. ‘Forall.getCurrentVectorLane()’ returns a tuple rather than an integer



TASK IDS

Sketch

- Here is a sketch of what that might look like:

- First, for a tiled matrix transpose (for GPU)

```
forall (i, j) in Dom {  
    const (idx, idY) = Forall.getCurrentVectorLane();  
    const (nX, nY) = Forall.getNumberOfVectorLanes();  
    ...;
```

- Second, for a distributed matrix transpose

```
forall (i, j) in Dom {  
    const (idx, idY) = Forall.getTask();  
    const (nX, nY) = Forall.getNumberOfTasks();  
    ...
```



BARRIERS



BARRIERS

- **Proposal 5:** Allow barriers among vector lanes within ‘forall’ and ‘foreach’ loops
- Barriers are important in GPU programming
 - shared memory in GPU programming only exists during the kernel & execution of a thread block
 - as a result, having different parts of a computation use shared memory requires a barrier
 - because separate ‘forall’ loops would create different kernels
 - a potential alternative is to create a separate notion of the GPU tasks—different from the ‘forall’
 - and then to ensure certain adjacent ‘forall’ loops use the same tasks
 - this seems more fraught
- The barrier will wait for all vector lanes to reach that point
- It might look like something along these lines

```
Forall.barrierVectorLanes();
```



GPUS AND LOCALES



GPUS AND LOCALES

- **Proposal 6:** Opt into GPU execution with ‘on’ statements
- We will not want all code to run on the GPU
 - GPUs are only effective for data-parallel computations
- Use a GPU sublocale and ‘on’ statements to request GPU execution
 - Since GPUs can’t run all code, ‘on’ a GPU means to create affinity with a GPU, where
 - memory is allocated in GPU memory
 - data-parallel loops are run as GPU kernels (‘order-independent-for’ in the examples)
- Using sublocales here allows things like a Block-distributed array over GPUs
 - even for programming, something like a cluster of nodes where each node contains 4 GPUs
- Thinking is that memory transfers will be optimized similarly to PGAS / distributed memory
- **Implication:** leader iterators may, but won’t necessarily, request GPU execution, with ‘on’ statements





**REQUESTING VECTOR WIDTH OR
NUMBER OF TASKS**

LOOP REQUESTS

- **Proposal 7:** Provide a way to request the GPU block size on a given ‘forall’ loop
- Proposal 4 suggests that this information can be discovered within a ‘forall’ loop
- Still need a way to set the information for a ‘forall’ loop
- A sketch of a solution:
 - all iterators need to accept a ‘configuration: loopConfiguration’ argument
 - ‘loopConfiguration’ stores properties normally communicated to the iterator from the calling loop
 - GPU block size request / requested number of vector lanes
 - requested number of qthreads-style tasks to create
 - it could also contain the shape of the overall iteration space when it is known



GPU EXAMPLES



GPU EXAMPLES

- Stream
- Reduction
- Transpose



GPU EXAMPLES

Stream

- Intent is for existing promoted stream to continue to be effective

```
A = B + alpha*C //Supposing that A, B, and C are (say) Block-distributed to GPU memory
```

- Let's consider a lower-level view

```
forall (a, b, c) in zip(A, B, C) {  
    a = b + alpha*c;  
}
```

- compiler translates into

```
coforall ... do on ... {  
    coforall ... {  
        order-independent-for i in followThis {  
            ref a = A[i], b = B[i], c = C[i]; //from follower iterators  
            a = b + alpha*c;  
        }  
    }  
}
```

GPU EXAMPLES

Stream: Memory Access Order

- Since the follower loop is order-independent, it can run vectorized or on a GPU
- Since we earlier had an ‘on’ statement to a GPU sublocale, it will run on GPU as a kernel

```
order-independent-for i in followThis {  
    ref a = A[i]; ref b = B[i]; ref c = C[i]; // from follower iterators  
    a = b + alpha*c;  
}
```

- It is important to make sure that the default loop ordering provides a reasonable memory access order
- For the GPU, we need GPU threads to work with adjacent memory for best efficiency
 - GPU has hardware coalescing support
- For vectorization, there is a similar requirement
 - common for a vector ‘load’ instruction to require adjacent memory—cheaper than an arbitrary *gather*
- So, supposing 8 GPU threads / vector lanes, the per-lane ‘foreach’ translates into

```
8-way-vector-parallel { for i in curVectorLane+followThis by 8 { ... } }
```

GPU EXAMPLES

Reduction

- A reduction might look like this

```
var sum = 0;  
forall a in A with (+ reduce sum) {  
    sum += a;  
}
```

- The compiler will translate this into something like:

```
c forall ... do on ... {  
    c forall ... {  
        8-way-vector-parallel {  
            var locSum = 0; // a different accumulator per vector lane / GPU task  
            for i in curVectorLane+followThis by 8 {  
                ref a = A[i]; // from follower iterators  
                locSum += a;  
            }  
            // accumulate the 8 locSum values into the task total  
        }    }    }
```



GPU EXAMPLES

Transpose

- Simplest matrix transpose

```
forall (i, j) in Dom {
    Output[j, i] = Input[i, j];
}
```

- Tiled matrix transpose

```
forall (i, j) in {0..<n by t, 0..<n by t} {
    var tile: [0..<t, 0..<t] int;
    // copy to local tile while transposing
    for (ti, tj) in tile.domain {
        tile[tj, ti] = Input[i*t + ti, j*t + tj];
    }
    // copy the already-transposed local tile to output
    for (ti, tj) in tile.domain {
        Output[j*t + ti, i*t + tj] = tile[ti, tj];
    }
}
```

```
// matrix transpose setup
const Dom = {0..<n, 0..<n};
var Input: [Dom] int;
var Output: [Dom] int;
param t = 32; // tile size dimension
```

GPU EXAMPLES

Transpose: using shared memory for more GPU performance

- Tiled matrix transpose #2

```
forall (i,j) in {0..<n by t, 0..<n by t}.these(
    configuration=new loopConfiguration(numVectorLanes=(t,8)) { //Proposal 7
        on GPUSharedMemory var tile: [0..<t, 0..<t] int;
        var x = i, y = j;
        var threadIdxX = Forall.getCurrentVectorLane()(0); //Proposal 4
        var threadIdxY = Forall.getCurrentVectorLane()(1); //assuming it returns a tuple for the (x,y) dimensions
        //copy to local tile while transposing
        for j in 0..t by 8 {
            tile[threadIdxY+j, threadIdxX] = Input[y+j, x];
        }
        Forall.vectorBarrier(); //Proposal 5 -- equivalent to __syncthreads()
        x = y - threadIdxY + threadIdxX;
        y = x - threadIdxX + threadIdxY;
        for j in 0..t by 8{
            Output[y+j, x] = tile[threadIdxX, threadIdxY + j];
        }
    }
}
```

GPU EXAMPLES

Transpose: using shared memory for more GPU performance

- Tiled matrix transpose #2

```
forall (i,j) in {0..<n by t, 0..<n by t}.these(
    configuration=new loopConfiguration(numVectorLanes=(t,8)) {
        on GPUSharedMemory var tile: [0..<t, 0..<t] int;
        var x = i, y = j;
        var threadIdxX = Forall.getCurrentVectorLane()(0);
        var threadIdxY = Forall.getCurrentVectorLane()(1);
        //copy to local tile while transposing
        for j in 0..t by 8 {
            tile[threadIdxY+j, threadIdxX] = Input[y+j, x];
        }
        Forall.vectorBarrier();
        x = y - threadIdxY + threadIdxX;
        y = x - threadIdxX + threadIdxY;
        for j in 0..t by 8{
            Output[y+j, x] = tile[threadIdxX, threadIdxY + j];
        }
    }
}
```

- Q: Is it possible to have the same level of performance with a kernel written within a ‘forall’ loop over ‘Dom’ rather than tiled?

TARGETING GPUS: SUMMARY



TARGETING GPUS: SUMMARY

- This discussion explored GPU support and Vectorization within Chapel
- It includes 7 proposals in support of GPU programming:
 - **Proposal 1:** Enable a unified programming model for vectorization and for GPUs
 - **Proposal 2:** Enable vectorization/GPU execution for global-view programs
 - **Proposal 3:** Support SIMD-style programming to keep flexibility
 - **Proposal 4:** Provide ways for code executing within a forall loop to discover task & vector division
 - **Proposal 5:** Allow barriers among vector lanes within ‘forall’ and ‘foreach’ loops
 - **Proposal 6:** Opt-in to GPU execution with ‘on’ statements
 - **Proposal 7:** Provide a way to request the GPU block size on a given ‘forall’ loop





THANK YOU

<https://chapel-lang.org>
@ChapelLanguage

