Case for Compiled Languages for HPC

Chapel Story

Engin Kayraklioglu, HPE

HPSFCon 2026, Chapel Project Meeting

March 18, 2026

What is Chapel?

Chapel is a programming language that is:

- **parallel & scalable:**
  - users benefit from multicore parallelism, GPUs, supercomputers…
- **portable:**
  - runs on Linux, MacOS, WSL, Cloud, InfiniBand, Slingshot, Raspberry Pi…
- **general-purpose:**
  - used in fields like aerodynamics, data and graph analytics, various earth sciences…
- **modern:**
  - object-orientation, memory and type safety, error handling, tool/IDE support…
- **open-source:**
  - a member of High-Performance Software Foundation & Linux Foundation…

Expressiveness Helps the Programmer *and* the Compiler

```
forall i in DistArr.domain do
      ... DistArr[i] ...
```

Can take a fast path if this is local, but we need to check that first.
incurs: per-access dynamic check

```
forall i in DistArr.domain do
     .... DistArr[f(i)] ...
```

We may need to send data across the network on each iteration.
incurs: per-access communication

```
DistArr1[x..y] = DistArr2[a..b];
```

Bulk data copy should be fast, but first, we need to create slice descriptors
incurs: a noticeable constant cost

Common problem:
High-level abstractions can incur execution time costs

The point I will make:
High-level abstractions can help the language stack optimize away those costs away and more

Expressiveness Helps the Programmer *and* the Compiler

```
forall idx in DistArr.domain do
      ... DistArr[i] ...
```

Can take a fast path if this is local, but we need to check that first.
incurs: per-access dynamic check

```
forall idx in DistArr.domain do
      .... DistArr[f(i)] ...
```

We may need to send data across the network on each iteration.
incurs: per-access communication

```
DistArr1[x..y] = DistArr2[a..b];
```

Bulk data copy should be fast, but first, we need to create slice descriptors
incurs: a noticeable constant cost

Common problem:
High-level languages can incur execution time costs

The point I will make:
High-level languages can help the language stack optimize away those costs away and more

Automatic Local Access

Before This Optimization

Three common idioms for implementing STREAM Triad in Chapel

```
var D = newBlockDom(1..n);
var A, B, C: [D] int;
```

Idiom 1

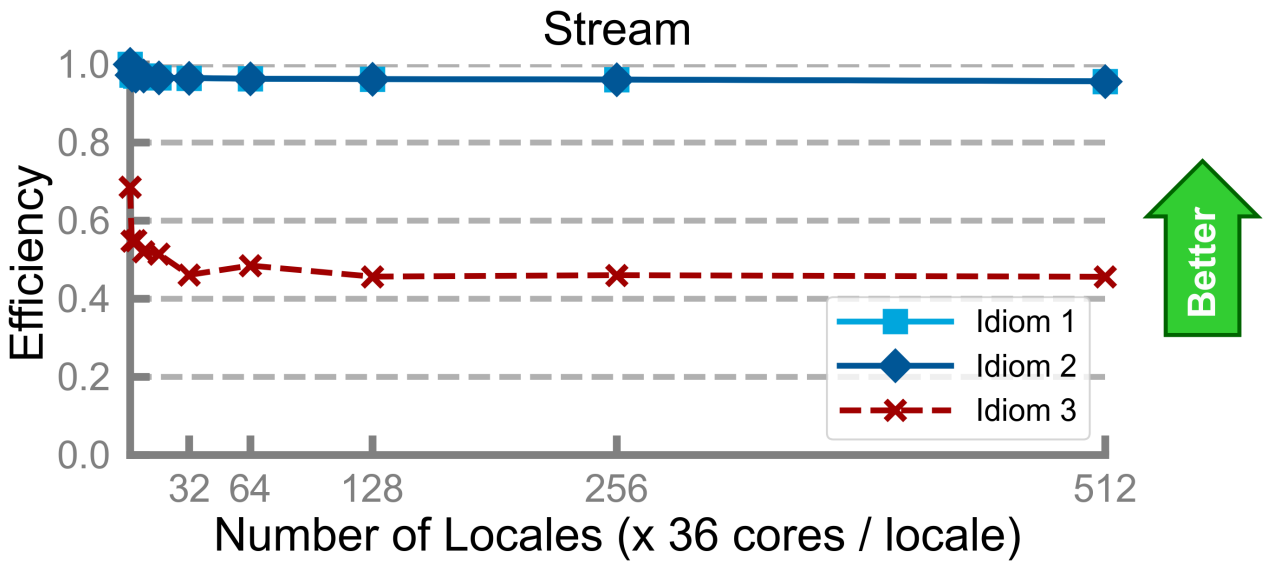
```
A = B + alpha * C;
```

Idiom 2

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

Idiom 3

```
forall i in D do
  A[i] = B[i] + alpha * C[i];
```



Automatic Local Access

Locality Check Overhead

```
var D = newBlockDom(1..n);
var A, B, C: [D] int;
```

```
proc array.access(idx: int) {
  if isLocalIndex(idx) then
    localAccess(idx);
  else
    nonLocalAccess(idx);
}
```

Idiom 3

```
forall i in D do
  A[i] = B[i] + alpha * C[i];
```

A per-access check is the source of overhead

- This check can be avoided for all 3 accesses
- Because;
  - The ‘forall’ distribution is aligned with A’s
    … because the loop is over A.domain
  - The loop index is the same as the access index
  - B and C’s distribution is aligned with A’s
    … because they all share the same domain

Automatic Local Access

After This Optimization: STREAM Triad

```
var D = newBlockDom(1..n);
var A, B, C: [D] int;
```

Idiom 1

```
A = B + alpha * C;
```

Idiom 2

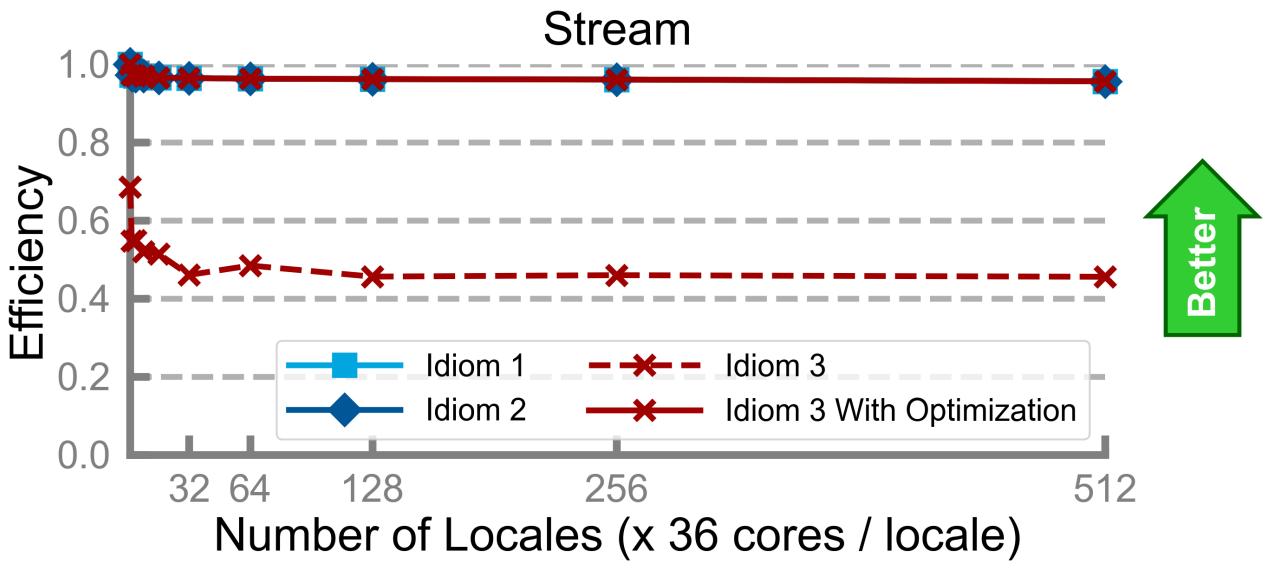
```
forall (a,b,c) in zip(A,B,C) do
  a = b + alpha * c;
```

Idiom 3

```
forall i in D do
  A[i] = B[i] + alpha * C[i];
```

Reaches 96% efficiency at-scale

All idioms perform similarly



Automatic Local Access

After This Optimization: NAS Parallel Benchmarks - FT

Explicit 'localAccess' calls are no longer needed in NPB-FT

Kernel with 'localAccess' calls

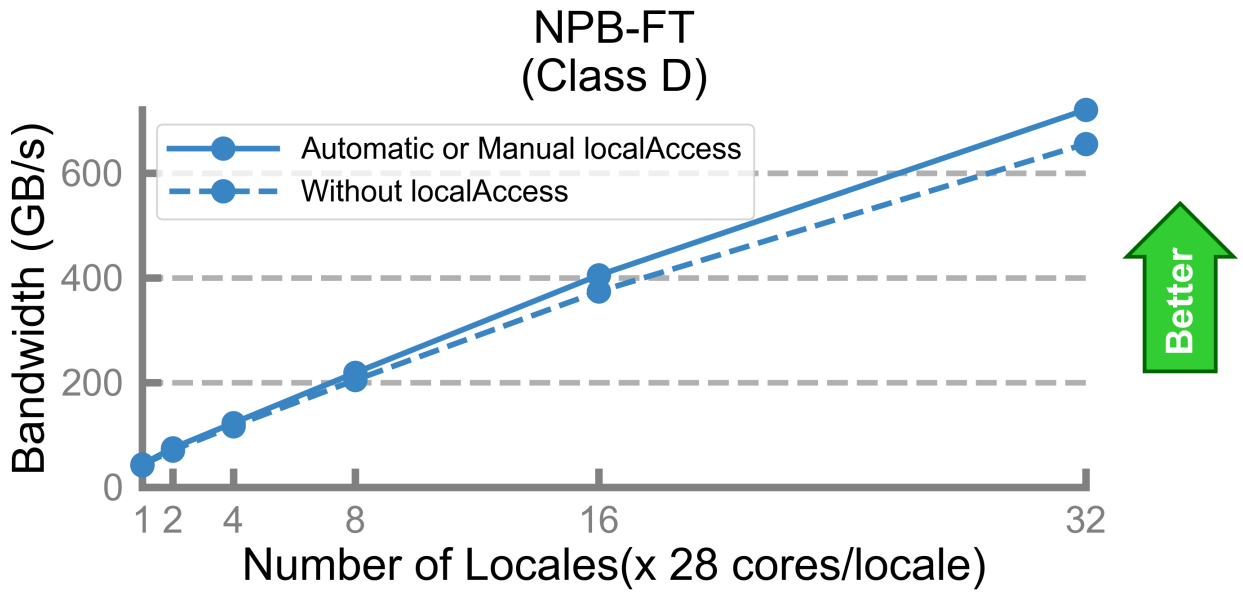
```
forall ijk in DomT {
  const elt = V.localAccess[ijk] *
              T.localAccess[ijk];

  V.localAccess[ijk] = elt;
  Wt.localAccess[ijk] = elt;
}
```

Kernel without 'localAccess' calls

```
forall ijk in DomT {
  const elt = V[ijk] *
              T[ijk];

  V[ijk] = elt;
  Wt[ijk] = elt;
}
```



Automatic Local Access

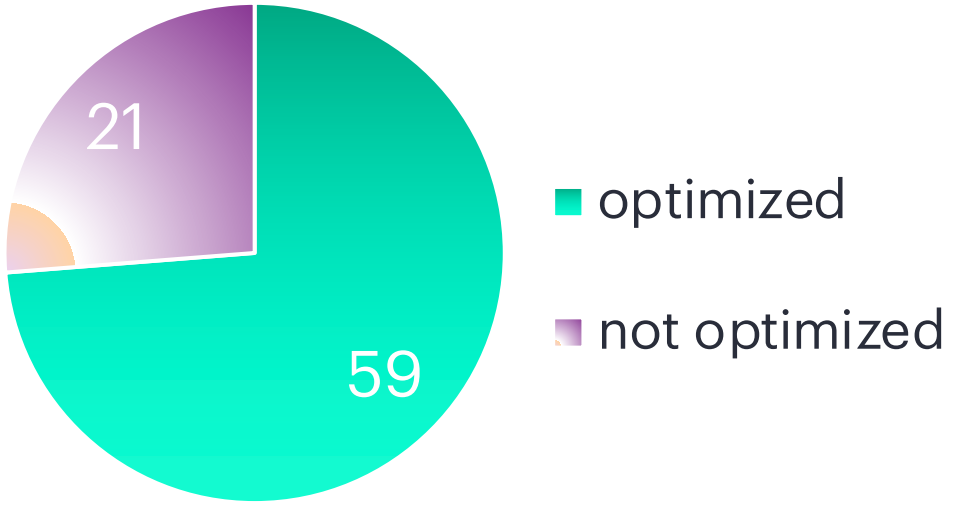
After This Optimization: chplUltra

chplUltra[1] is an Ultralight Dark Matter simulator written in Chapel

We removed all explicit calls to *localAccess*

— 80 places in total
- 59 are optimized automatically
- 21 were not optimized

— The patterns where the optimization does not fire
- 10 locality hard to detect due to complex alignments
- 7 array access indices are not loop indices
- 4 is not inside forall loops



[1] Nikhil Padmanabhan et al. “Simulating Ultralight Dark Matter in Chapel”. 2020 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW). May 2020,

Expressiveness Helps the Programmer *and* the Compiler

```
forall idx in DistArr.domain do
        ... DistArr[i] ...
```

Can take a fast path if this is local, but we need to check that first.
incurs: per-access dynamic check

```
forall idx in DistArr.domain do
      .... DistArr[f(i)] ...
```

We may need to send data across the network on each iteration.
incurs: per-access communication

```
DistArr1[x..y] = DistArr2[a..b];
```

Bulk data copy should be fast, but first, we need to create slice descriptors
incurs: a noticeable constant cost

Common problem:
High-level languages can incur execution time costs

The point I will make:
High-level languages can help the language stack optimize away those costs away and more

Automatic Aggregation

Before This Optimization

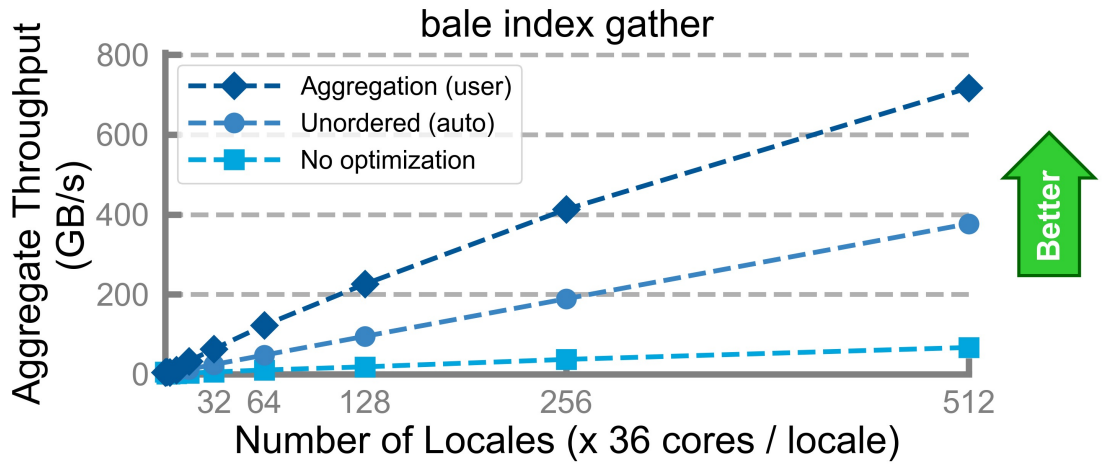
The *indexgather* benchmark from the bale[2] study tests random access performance

```
var D = newBlockDom(1..n);
var Src, Dst, Inds: [D] int;
```

Straightforward *indexgather*

```
forall (d, i) in zip(Dst, Inds) do
  d = Src[i];
```

Communication can be done in any order
The Chapel compiler already had "unordered forall" optimization



A Manual Approach for Data Aggregation

```
forall (d, i) in zip(Dst, Inds) with (var agg = new SrcAggregator(int)) do
  agg.copy(d, Src[i]);
```

[3] https://github.com/Bears-R-Us/arkouda

Automatic Aggregation

Connecting the Dots for Automatic Aggregation

Straightforward *indexgather*

```
forall (d, i) in zip(Dst, Inds) do
```

What does it take to make this transition automatically?

Q: Is it safe to complete communication in any order?

A: unordered forall optimization does that analysis

Q: Is exactly one side of the operation local?

A: automatic local access optimization does that analysis

Q: Do you have means to aggregate the data?

A: aggregator objects can do that

A Manual Approach for Data Aggregation

```
forall (d, i) in zip(Dst, Inds) with (var agg = new SrcAggregator(int)) do
  agg.copy(d, Src[i]);
```

Automatic Aggregation

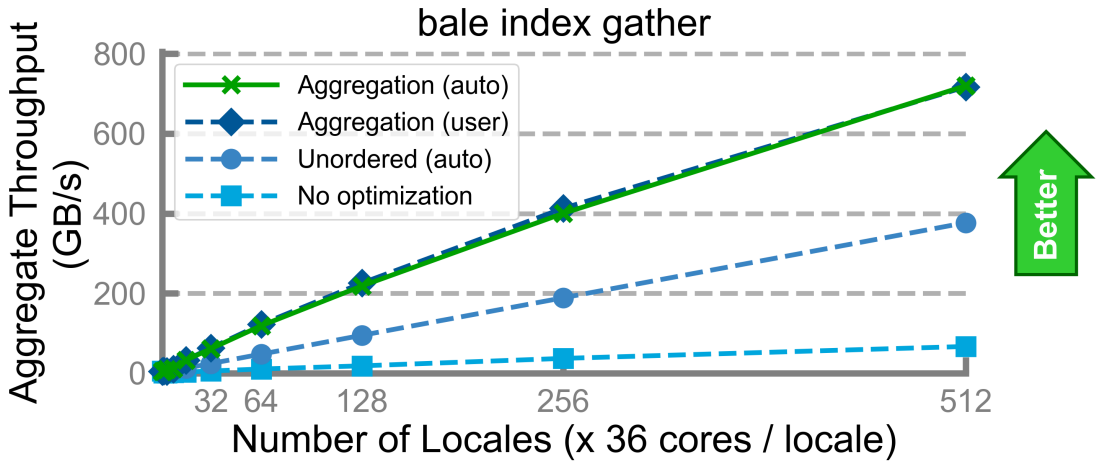
After This Optimization: *indexgather*

The *indexgather* benchmark from the bale study tests random access performance

```
var D = newBlockDom(1..n);
var Src, Dst, Inds: [D] int;
```

Straightforward *indexgather*

```
forall (d, i) in zip(Dst, Inds) do
  d = Src[i];
```



Straightforward version performs identical to the manually-aggregated version

A Manual Approach for Data Aggregation

```
forall (d, i) in zip(Dst, Inds) with (var agg = new SrcAggregator(int)) do
  agg.copy(d, Src[i]);
```

Automatic Aggregation

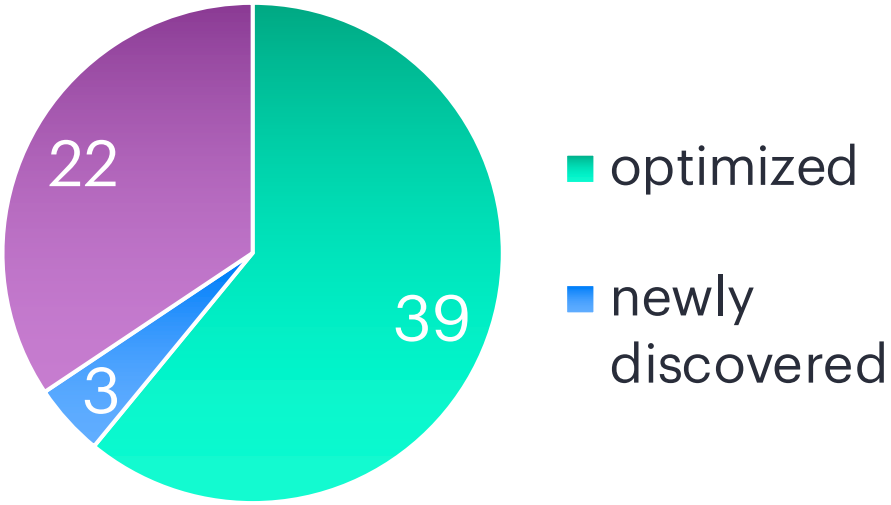
After This Optimization: Arkouda

Arkouda[3] is a data analytics tool that has a Python client and a server implemented in Chapel

We removed all the manual aggregation from the source

— 61 places in total
  - 39 are optimized automatically
  - 22 are not optimized

— 3 cases that were not using aggregators are now optimized



— The patterns where the aggregation does not fire:
  - 9: aggregation is not based on ‘forall’ loops
  - 6: compiler cannot prove that unordered operation is safe
  - 3: locality is hard to detect
  - 2: aggregated copy is not in the last statement of the body
  - 1: one side of the assignment is defined within the loop body
  - 1: needs further investigation

[3] https://github.com/Bears-R-Us/arkouda

Expressiveness Helps the Programmer *and* the Compiler

```
forall idx in DistArr.domain do
      ... DistArr[i] ...
```

Can take a fast path if this is local, but we need to check that first.
incurs: per-access dynamic check

```
forall idx in DistArr.domain do
     .... DistArr[f(i)] ...
```

We may need to send data across the network on each iteration.
incurs: per-access communication

```
DistArr1[x..y] = DistArr2[a..b];
```

Bulk data copy should be fast, but first, we need to create slice descriptors
incurs: a noticeable constant cost

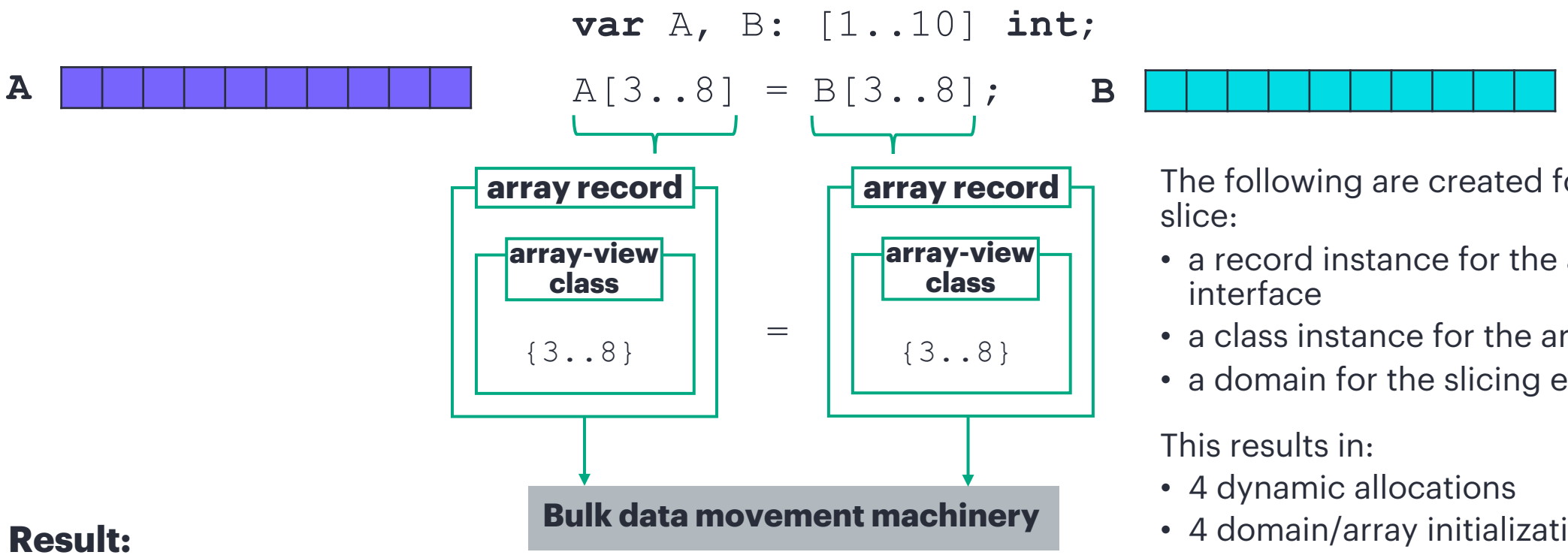
Common problem:
High-level languages can incur execution time costs

The point I will make:
High-level languages can help the language stack optimize away those costs away and more

Array View Elision

Before This Optimization

The common pattern of copying between two slices had a lot of overhead



The following are created for each slice:

- a record instance for the array interface
- a class instance for the array view
- a domain for the slicing expression

This results in:

- 4 dynamic allocations
- 4 domain/array initializations

These costs can impact performance with small transfers

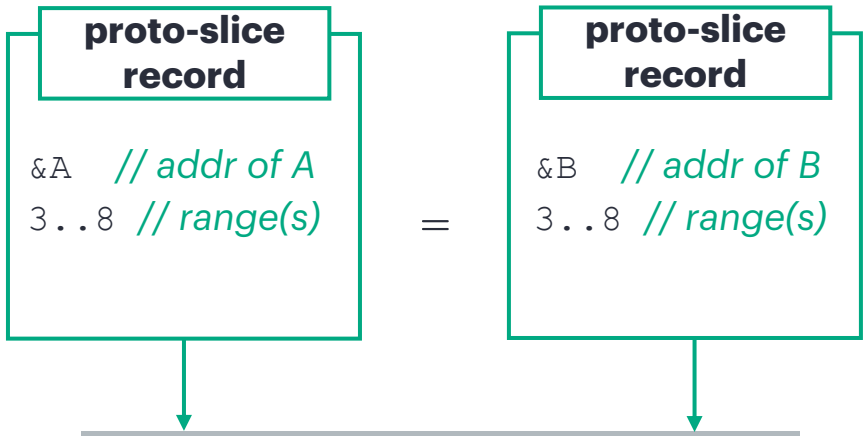
Array View Elision

Reducing Costs of High-Level Representation

With Chapel 2.2, the compiler detects this common pattern and optimizes it:

```
var A, B: [1..10] int;

A[3..8] = B[3..8];
```



Bulk data movement machinery
(now capable of handling proto-slices)

Result:

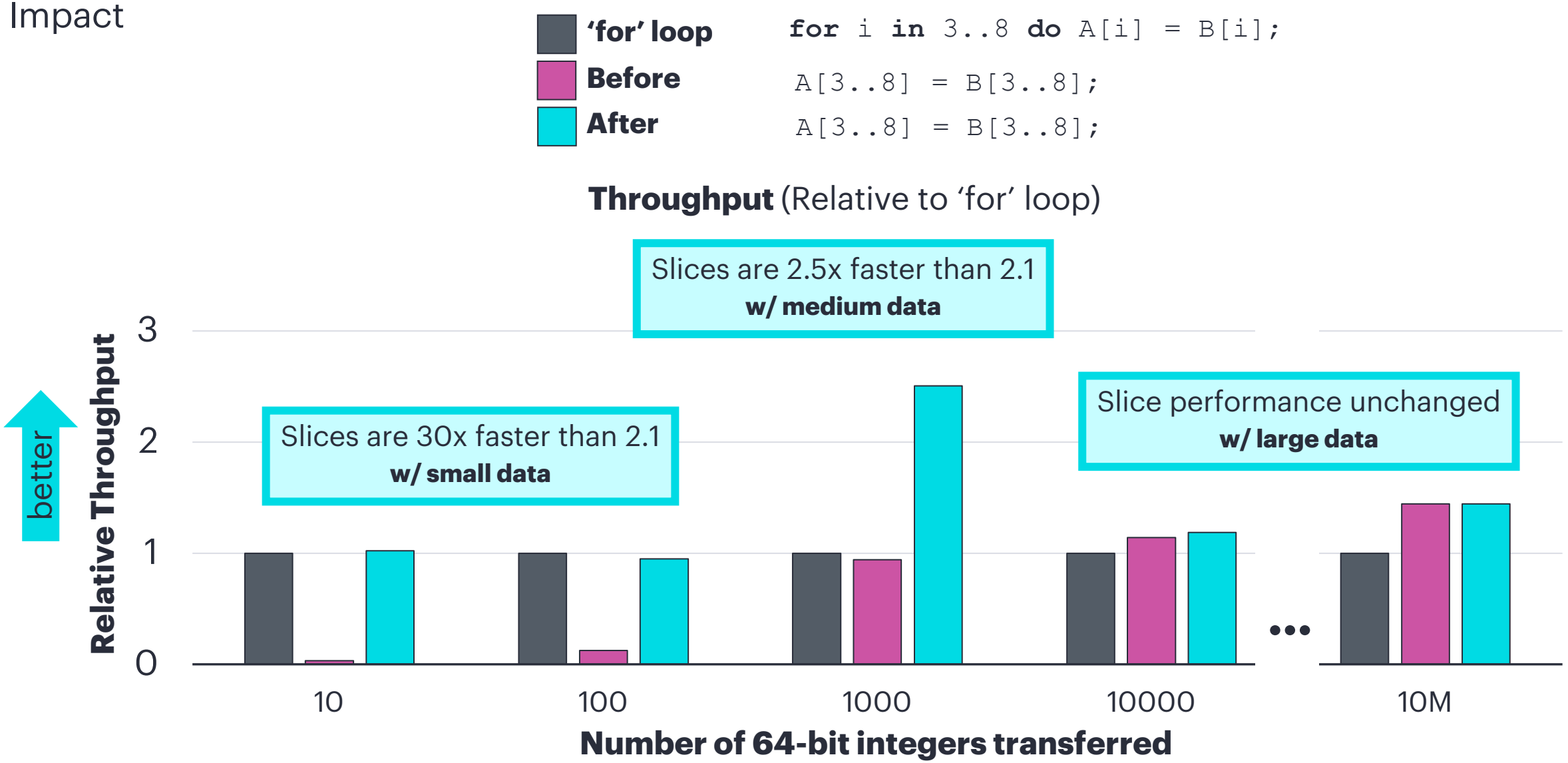
A "proto-slice" record stores:
- the original array's address
- a (tuple of) range(s)

This results in:
- no dynamic allocations
- much faster initialization

Significantly reduced overhead for a common operation

Array View Elision

Impact



Summary

- Chapel is a high-level programming language
  - Abstractions make programmer's life easier
  - However, they can also incur costs

- High-level abstractions can also enable the compiler and the runtime system to optimize execution
  - The primary reason is that they don't prescribe what the runtime system should do
  - Rather, they express what the programmer wants to achieve
  - This enables the language stack to "implement" the operation in an optimized (and portable!) way

The optimizations we covered today make programmer's life easier without sacrificing performance

Thank you!