LLVM Optimizations for PGAS Programs

-Case Study: LLVM Wide Pointer Optimizations in Chapel-

CHIUW2014 (co-located with IPDPS 2014),
Phoenix, Arizona

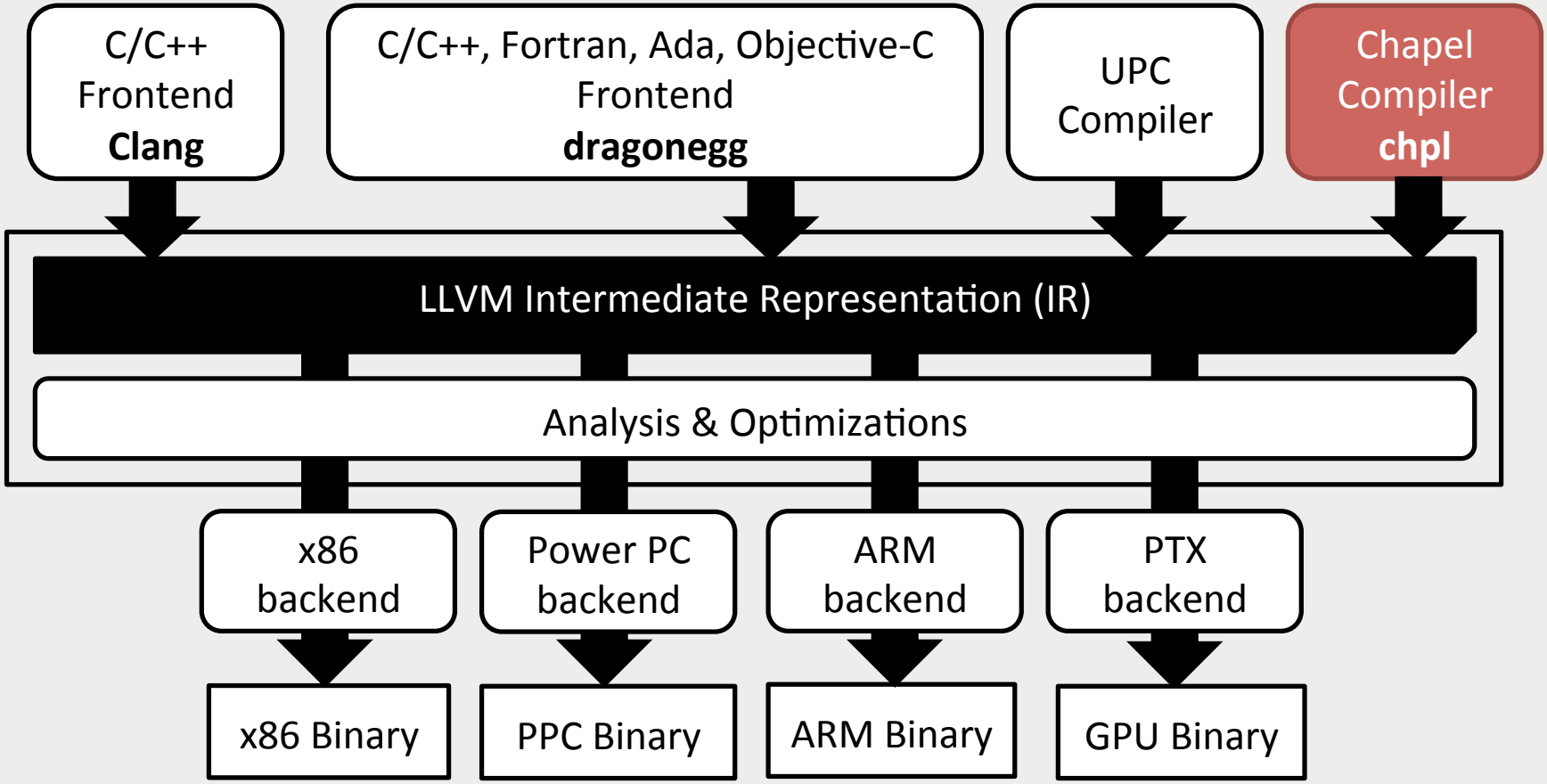
Akihiro Hayashi, Rishi Surendran,
Jisheng Zhao, Vivek Sarkar
(Rice University),
Michael Ferguson
(Laboratory for Telecommunication Sciences)

Background: Programming Model for Large-scale Systems

- Message Passing Interface (MPI) is a ubiquitous programming model
  - but introduces non-trivial complexity due to message passing semantics
- PGAS languages such as Chapel, X10, Habanero-C and Co-array Fortran provide high-productivity features:
  - Task parallelism
  - Data Distribution
  - Synchronization

RICE

Motivation: Chapel Support for LLVM



- Widely used and easy to Extend

RICE

A Big Picture

©Oak Ridge National Lab.

© Argonne National Lab.

© RIKEN AICS

CRAY CHAPEL

BERKELEY UPC UNIFIED PARALLEL C

LLVM COMPILER INFRASTRUCTURE

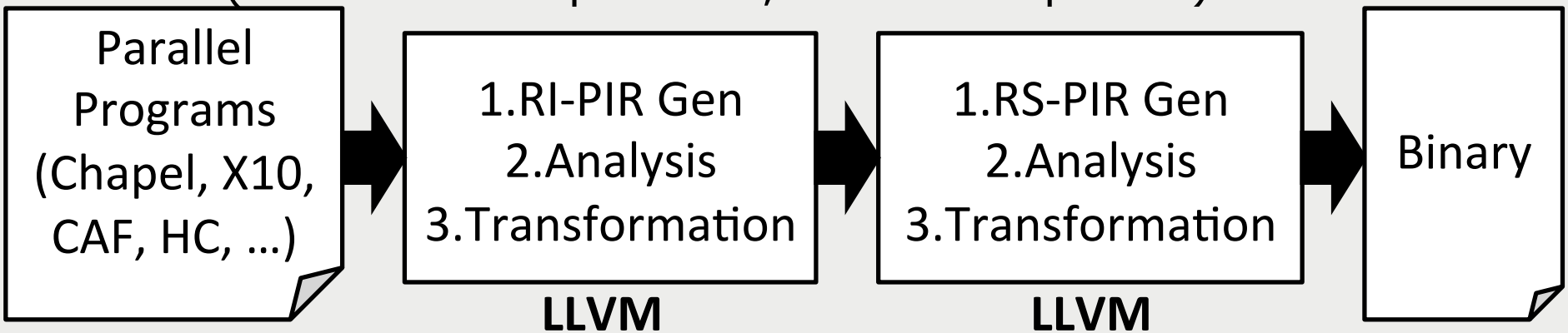
Habanero-C, ...

RICE

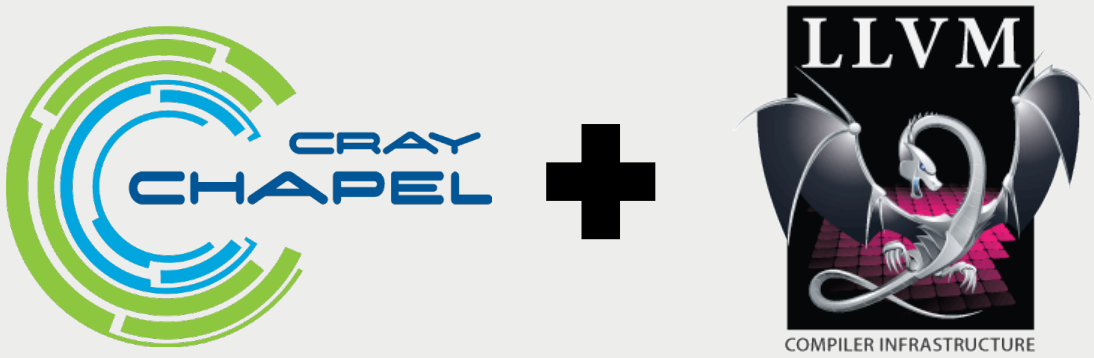
Pictures borrowed from http://chapel.cray.com/logo.html, http://llvm.org/Logo.html, http://upc.lbl.gov/, http://commons.wikimedia.org/, https://www.olcf.ornl.gov/titan/

Our ultimate goal: A compiler that can uniformly optimize PGAS Programs

- Extend LLVM IR to support parallel programs with PGAS and explicit task parallelism
  - Two parallel intermediate representations(PIR) as extensions to LLVM IR (Runtime-Independent, Runtime-Specific)



The first step: LLVM-based Chapel compiler



- Chapel compiler supports LLVM IR generation
- This talk discusses the pros and cons of LLVM-based communication optimizations for Chapel
  - *Wide pointer optimization*
- Preliminary Performance evaluation & analysis using three regular applications

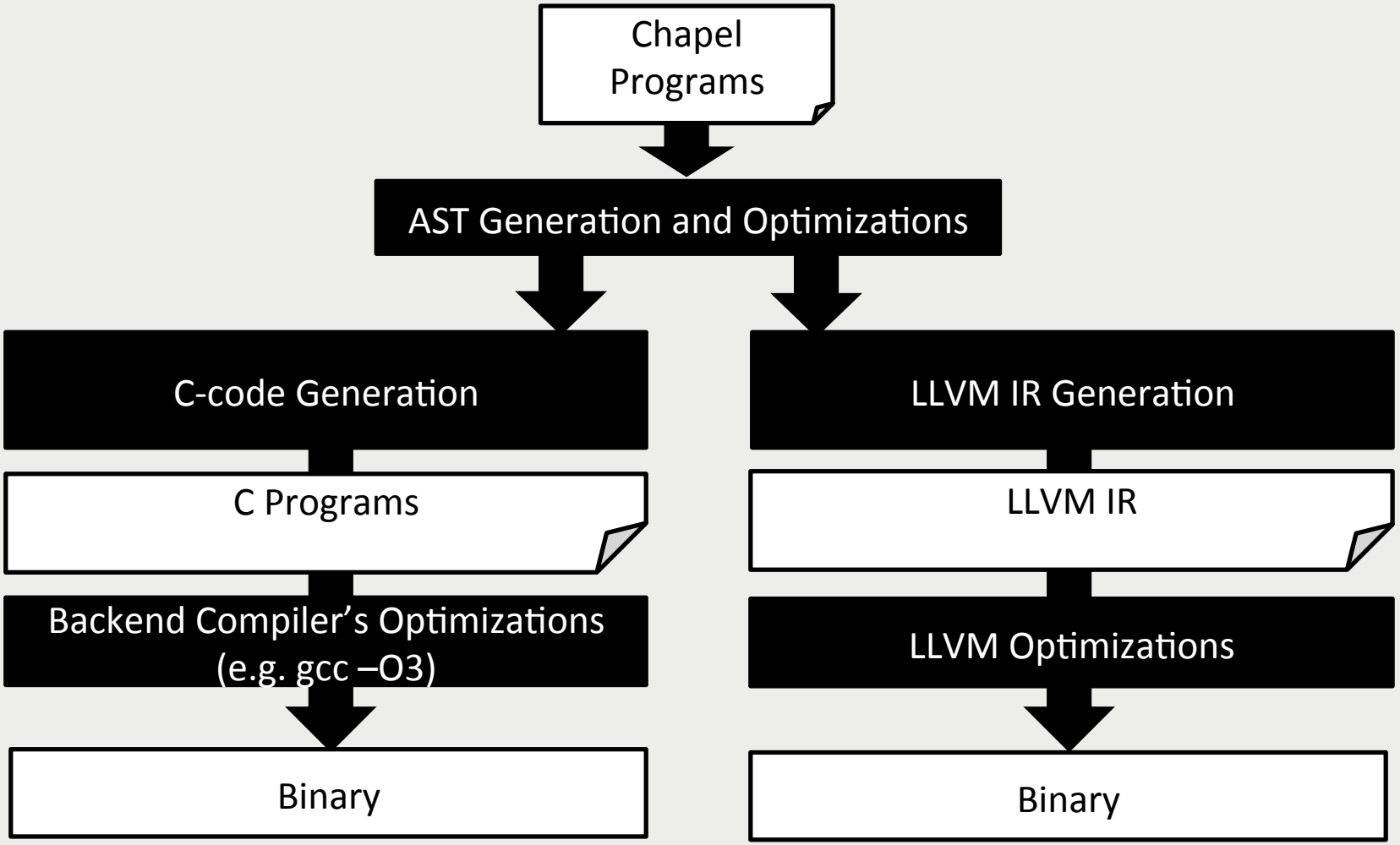
RICE

Pictures borrowed from 1) http://chapel.cray.com/logo.html
2) http://llvm.org/Logo.html

Chapel language

- An object-oriented PGAS language developed by Cray Inc.
  - Part of DARPA HPCS program
- Key features
  - Array Operators: *zip, replicate, remap,...*
  - Explicit Task Parallelism: *begin, cobegin*
  - Locality Control: *Locales*
  - Data-Distribution: *domain maps*
  - Synchronizations: *sync*

Compilation Flow



RICE

The Pros and Cons of using LLVM for Chapel

- **Pro:** Using address space feature of LLVM offers more opportunities for **communication optimization** than C gen

```
// Chapel
x = remoteData;
```

```
// C-Code generation
chpl_comm_get(&x, …);
```

Backend Compiler's Optimizations
(e.g. gcc –O3)

Few chances of optimization because remote accesses are lowered to chapel Comm APIs

```
// LLVM IR
%x = load i64 addrspace(100)* %xptr
```

LLVM Optimizations
(e.g. LICM, scalar replacement)

1. the existing LLVM passes can be used for communication optimizations
2. Lowered to chapel Comm APIs after optimizations

Address Space 100 generation in Chapel

- Address space 100 = possibly-remote (our convention)
- Constructs which generate address space 100
  - **Array Load/Store** (Except Local constructs)
  - **Distributed Array**
    - var d = {1..128} dmapped Block(boundingBox={1..128});
    - var A: [d] int;
  - **Object and Field Load/ Store**
    - class circle { var radius: real; … }
    - var c1 = new circle(radius=1.0);
  - **On statement**
    - var loc0: int;
    - on Locales[1] { loc0 = …; }
  - **Ref intent**
    - proc habanero(ref v: int): void { v = …; }

Except remote value forwarding optimization

RICE

Motivating Example of address space 100

LICM by LLVM

```
(Pseudo-Code: Before LICM)
for i in 1..N {
    // REMOTE GET
    %x = load i64 addrspace(100)* %xptr
    A(i) = %x;
}
```

```
(Pseudo-Code: After LICM)
// REMOTE GET
%x = load i64 addrspace(100)* %xptr
for i in 1..N {
    A(i) = %x;
}
```

RICE

LICM = Loop Invariant Code Motion

The Pros and Cons of using LLVM for Chapel (Cont'd)

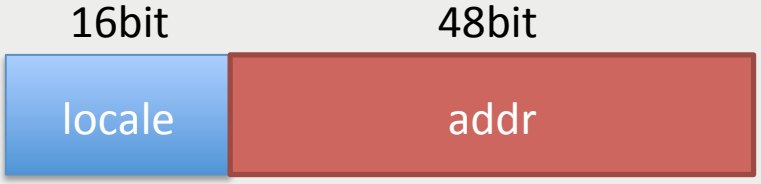
- **Drawback:** Using LLVM may lose opportunity for optimizations and may add overhead at runtime

For C Code Generation :
128bit struct pointer
CHPL_WIDE_POINTERS=struct

```
typedef struct wide_ptr_s {
  chpl_localeID_t locale;
  void* addr;
} wide_ptr_t;
```

```
wide.locale;
wide.addr;
```

For LLVM Code Generation :
64bit packed pointer
CHPL_WIDE_POINTERS=node16




```
wide >> 48
wide | 48BITS_MASK;
```

1. Needs more instructions
2. Lose opportunities for Alias analysis

- In LLVM 3.3, many optimizations assume that the pointer size is the same across all address spaces

RICE

Performance Evaluations: Experimental Methodologies

- We tested execution in the following modes
  - 1.C-Struct (--fast)
    - C code generation + struct pointer + gcc
    - Conventional Code generation in Chapel
  - 2.LLVM without wide optimization (--fast --llvm)
    - LLVM IR generation + packed pointer
    - **Does not** use address space feature
  - 3.LLVM with wide optimization (--fast --llvm --llvm-wide-opt)
    - LLVM IR generation + packed pointer
    - Use address space feature and apply the existing LLVM optimizations

RICE

Performance Evaluations: Platform

- Intel Xeon-based Cluster
  - Per Node information
    - Intel Xeon CPU X5660@2.80GHz x 12 cores
    - 48GB of RAM
- Interconnect
  - Quad-data rated Infiniband
  - Mellanox FCA support

RICE

Performance Evaluations: Details of Compiler & Runtime

- Compiler:
  Chapel version 1.9.0.23154 (Apr. 2014)
  - Built with
    - CHPL_LLVM=llvm
    - CHPL_WIDE_POINTERS=node16 or struct
    - CHPL_COMM=**gasnet** CHPL_COMM_SUBSTRATE=**ibv**
    - CHPL_TASK=**qthread**
  - Backend compiler: gcc-4.4.7, LLVM 3.3
- Runtime:
  - GASNet-1.22.0 (ibv-conduit, mpi-spawner)
  - qthreads-1.10
    (2 Shepherds, 6 worker per shepherd)

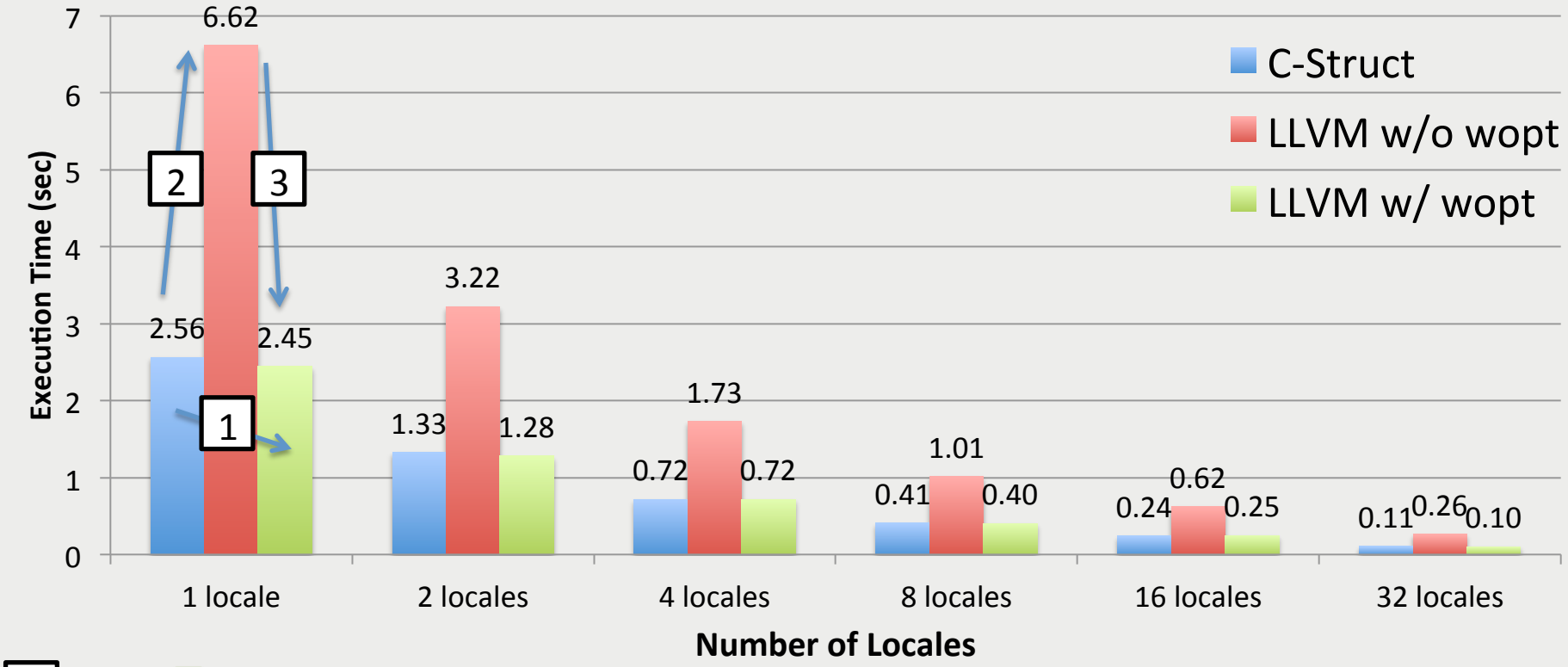
Stream-EP

```
coforall loc in Locales do on loc {
    // per each locale
    var A, B, C: [D] real(64);
    forall (a, b, c) in zip(A, B, C) do
        a = b + alpha * c;
}
```

- From HPCC benchmark
- Array Size: 2^30

Stream-EP Result

Lower is better



1 vs. LLVM+wide opt is faster than the conventional C-Struct (1.1x)

2 vs. Overhead of introducing LLVM + packed pointer (2.6x slower)

3 vs. Performance improvement by LLVM opt (2.7x faster)

RICE

Stream-EP Analysis

Dynamic number of Chapel PUT/GET APIs actually executed (16 Locales):

C-Struct	LLVM w/o wopt	LLVM w/ wopt
1.39E+11	1.40E+11	5.46E+10

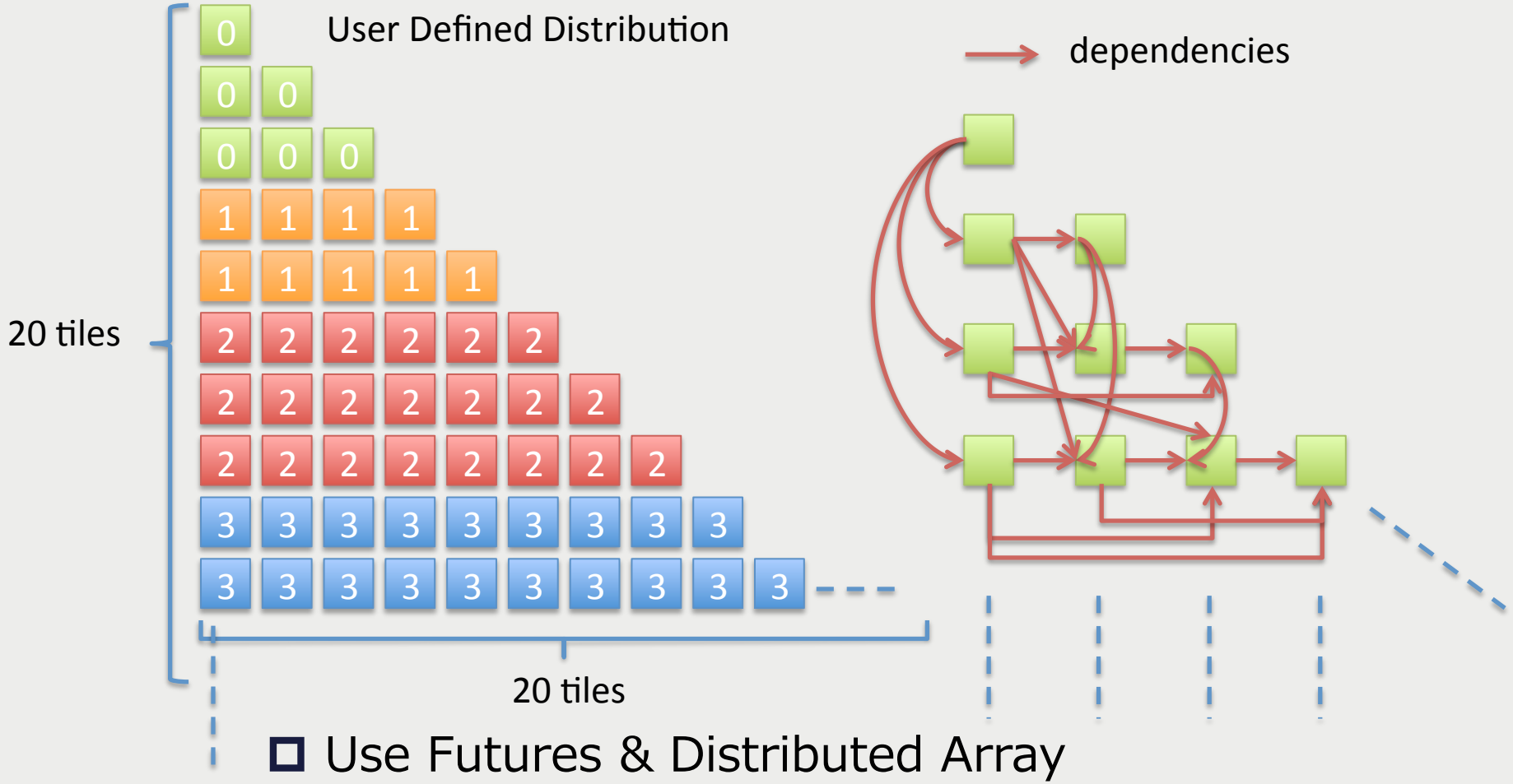
```
// C-Struct, LLVM w/o wopt
forall (a, b, c) in zip(A, B, C) do
    8GETS / 1PUT
```

```
// LLVM w/ wopt
6GETS (Get Array Head, offs)
forall (a, b, c) in zip(A, B, C) do
    2GETS / 1PUT
```

LICM by LLVM

RICE

Cholesky Decomposition

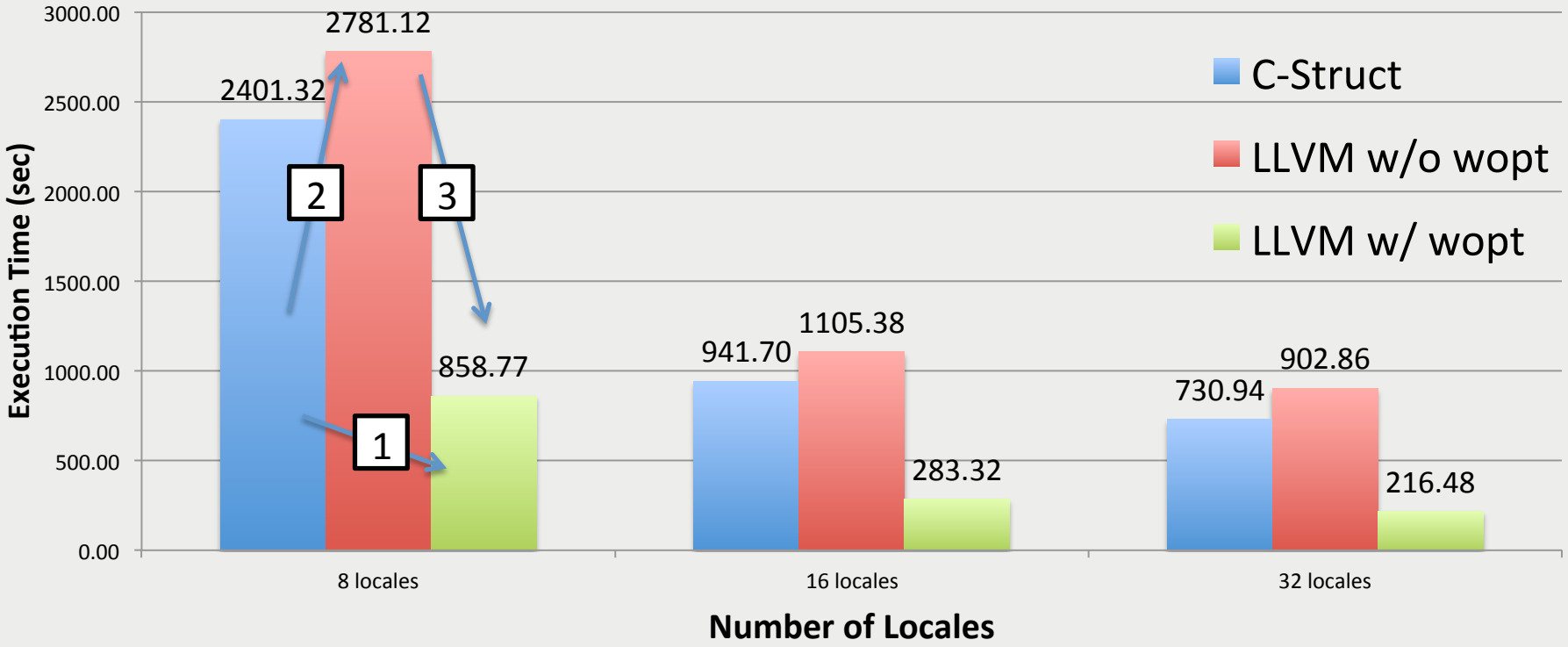


- Use Futures & Distributed Array
- Input Size: 10,000x10,000
  - Tile Size: 500x500

RICE

Cholesky Result

Lower is better



1 vs. LLVM+wide opt is faster than the conventional C-Struct (3.4x)

2 vs. Overhead of introducing LLVM + packed pointer (1.2x slower)

3 vs. Performance improvement by LLVM opt (4.2x faster)

RICE

Cholesky Analysis

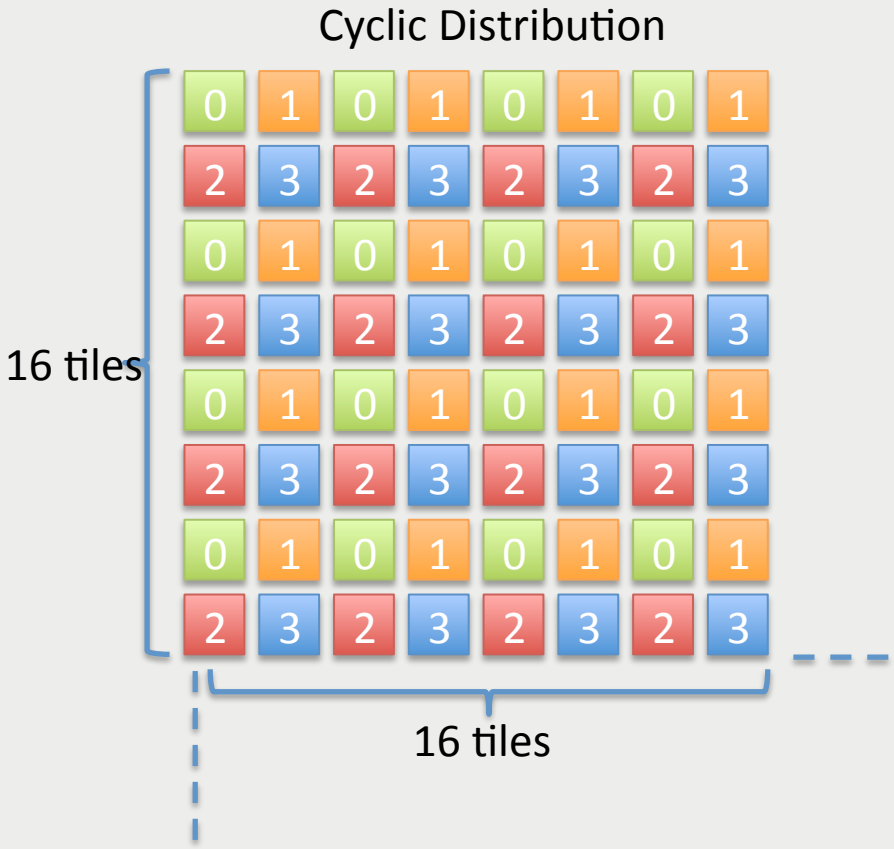
Dynamic number of Chapel PUT/GET APIs actually executed (2 Locales):
Obtained with 1,000 x 1,000 input (100x100 tile size)

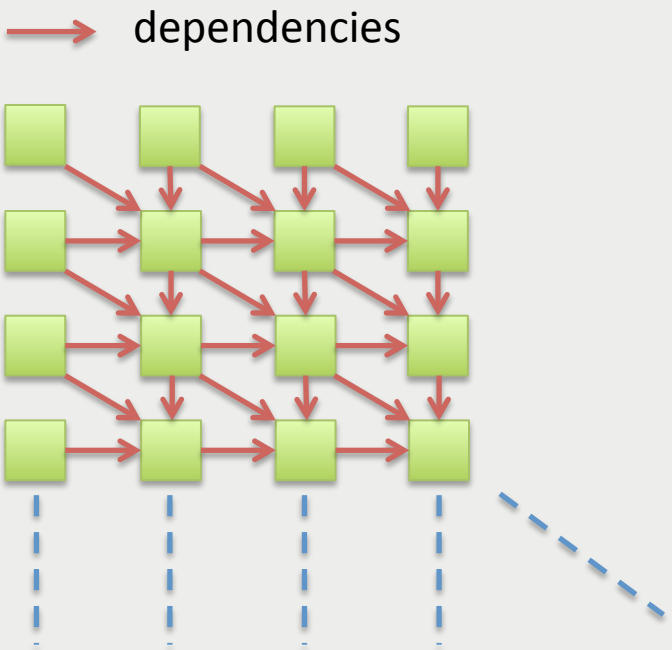
C-Struct	LLVM w/o wopt	LLVM w/ wopt
1.78E+09	1.97E+09	5.89E+08

```
// C-Struct, LLVM w/o wopt
for jB in zero..tileSize-1 do {
  for kB in zero..tileSize-1 do {
    4GETS
    for iB in zero..tileSize-1 do {
      8GETS (+1 GETS w/ LLVM)
      1PUT
    }}}
```

```
// LLVM w/ wopt
for jB in zero..tileSize-1 do {
  1GET
  for kB in zero..tileSize-1 do {
    3GETS
    for iB in zero..tileSize-1 do {
      2GETS
      1PUT
    }}}
```

Smithwaterman



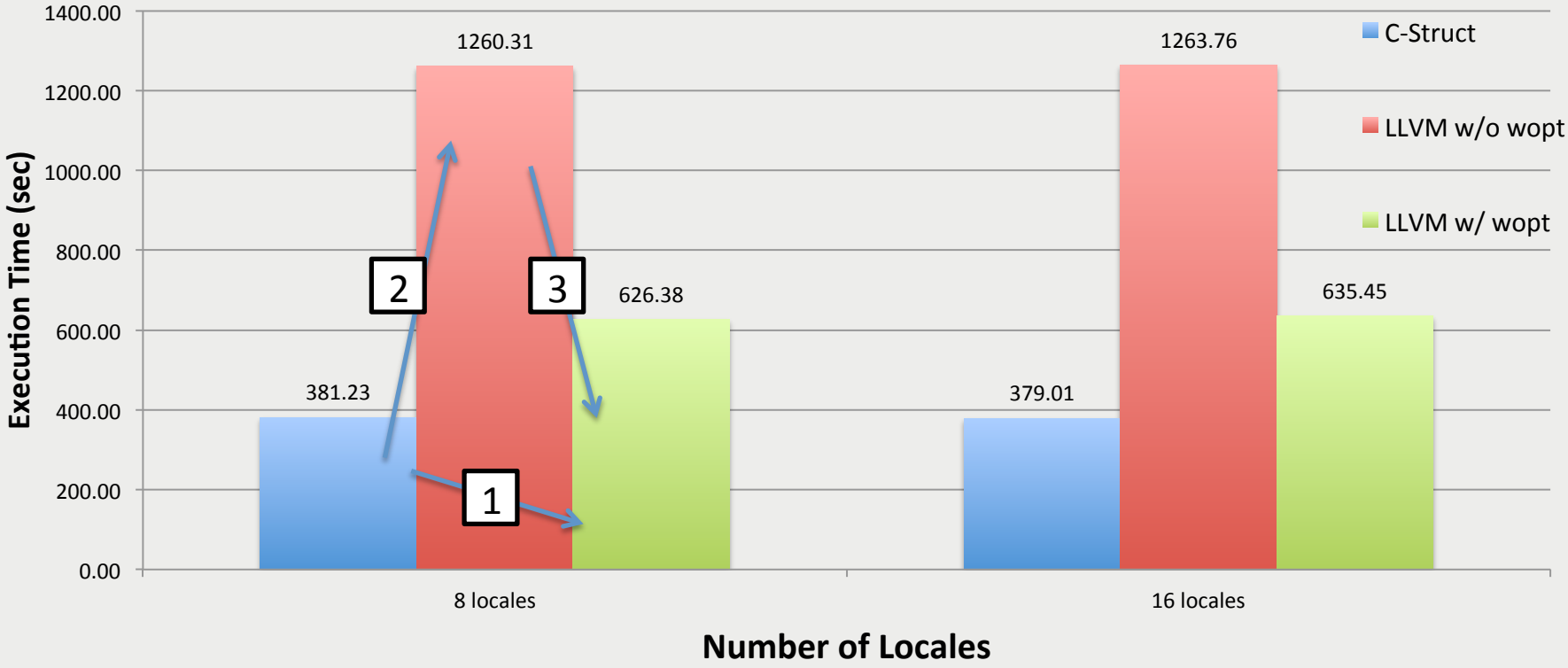


- Use Futures & Distributed Array
- Input Size: 185,500x192,000
  - Tile Size: 11,600x12,000

RICE

Smithwaterman Result

Lower is better



1 vs. LLVM+wide opt is **slower** than the conventional C-Struct (0.6x)

2 vs. Overhead of introducing LLVM + packed pointer (3.3x slower)

3 vs. Performance improvement by LLVM opt (2.0x faster)

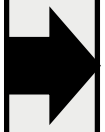
RICE

Smithwaterman Analysis

Dynamic number of Chapel PUT/GET APIs actually executed (1 Locale):
Obtained with 1,856 x 1,920 input (232x240 tile size)

C-Struct	LLVM w/o wopt	LLVM w/ wopt
1.41E+08	1.41E+08	5.26E+07

```
// C-Struct, LLVM w/o wopt
for (ii, jj) in tile_1_2d_domain
{
   33 GETS
   1 PUTS
}
```



```
// LLVM w/ wopt
for (ii, jj) in tile_1_2d_domain
{
   12 GETS
   1 PUTS
}
```

No LICM though there are opportunities

Key Insights

- Using address space 100 offers finer-grain optimization opportunities (e.g. Chapel Array)

```
for i in {1..N} {
  data = A(i);
}
```

```
for i in 1..N {
    head = GET(pointer to array head)
    offset1 = GET(offset)
    data = GET(head+i*offset1)
}
```

Opportunities for
1.LICM
2.Aggregation

Conclusions

- The first performance evaluation and analysis of LLVM-based Chapel compiler
  - Capable of utilizing the existing optimizations passes even for remote data (e.g. LICM)
    - Removes significant number of Comm APIs
  - LLVM w/ opt is always better than LLVM w/o opt
  - Stream-EP, Cholesky
    - LLVM-based code generation is faster than C-based code generation (1.04x, 3.4x)
  - Smithwaterman
    - LLVM-based code generation is slower than C-based code generation due to constraints of address space feature in LLVM
    - No LICM though there are opportunities
    - Significant overhead of Packed Wide pointer

Future Work

- Evaluate other applications
  - Regular applications
  - Irregular applications
- Possibly-Remote to Definitely-Local transformation by compiler

```
on Locales[1] { // hint by programmer
   var A: [D] int; // Definitely Local
```

```
local { A(i) = … } // hint by programmmer
… = A(i); // Definitely Local
```

- PIR in LLVM

Acknowledgements

- Special thanks to
  - Brad Chamberlain (Cray)
  - Rafael Larrosa Jimenez (UMA)
  - Rafael Asenjo Plaza (UMA)
  - Shams Imam (Rice)
  - Sagnak Tasirlar (Rice)
  - Jun Shirako (Rice)

RICE