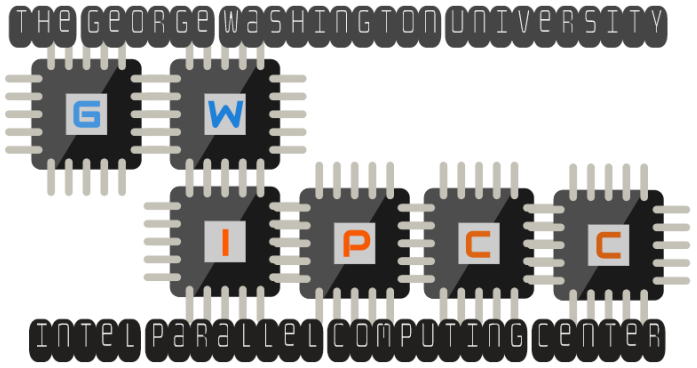


THE GEORGE WASHINGTON UNIVERSITY

WASHINGTON, DC

Comparative Performance and Optimization of Chapel in Modern Manycore Architectures*

Engin Kayraklioglu, Wo Chang, Tarek El-Ghazawi

*This work is partially funded through an Intel Parallel Computing Center gift.

Outline

- Introduction & Motivation
- Chapel Primer
  - Implementing Nstream-like Applications
  - More: Chapel Loops, Distributed Data
- Experimental Results
  - Environment, Implementation Caveats
  - Results
- Detailed Analysis
  - Memory Bandwidth Analysis on KNL
  - Idioms & Optimizations For Sparse
  - Optimizations for DGEMM
- Summary & Wrap Up

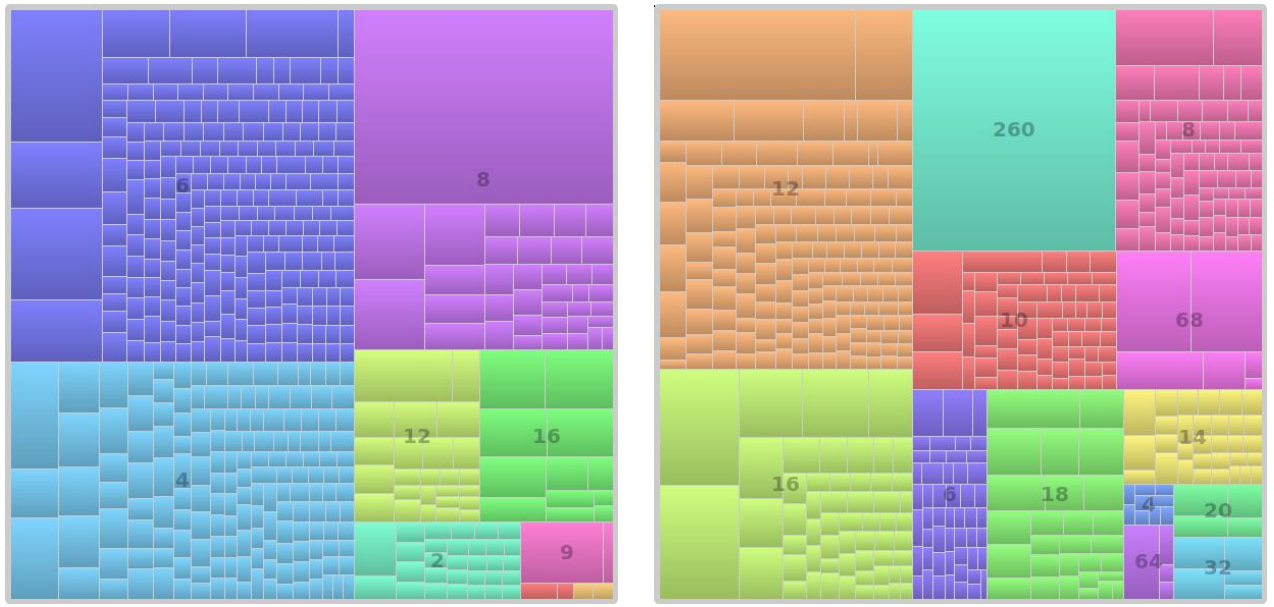
Outline

- Introduction & Motivation
- Chapel Primer
  - Implementing Nstream-like Applications
  - More: Chapel Loops, Distributed Data
- Experimental Results
  - Environment, Implementation Caveats
  - Results
- Detailed Analysis
  - Memory Bandwidth Analysis on KNL
  - Idioms & Optimizations For Sparse
  - Optimizations for DGEMM
- Summary & Wrap Up

HPC Trends

- Steady increase in core/socket in TOP500
- Deeper interconnection networks
- Deeper memory hierarchies

- More NUMA effects
- Need for newer programming paradigms

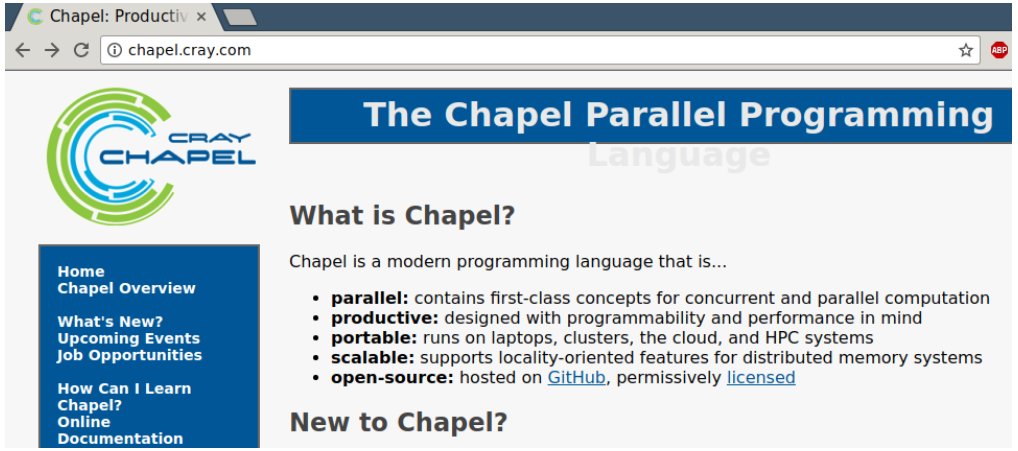


Core/socket Treemap for Top 500 systems of 2011 vs 2016

generated on top500.org

What is Chapel?

- Chapel is an upcoming parallel programming language
  - Parallel, productive, portable, scalable, open-source
- Designed from scratch, with independent syntax
- Partitioned Global Address Space (PGAS) memory
- General high-level programming language concepts
  - OOP, inheritance, generics, polymorphism..
- Parallel programming concepts
  - Locality-aware parallel loops, first-class data distribution objects, locality control



chapel.cray.com

The Paper

- To appear
  - in Chapel Users and Implementers Workshop at IPDPS17
- Compares Chapel's performance to OpenMP on multi- and many-core architectures
- Uses The Parallel Research Kernels for analysis
- Specific contributions:
  - Implements 4 new PRKs: DGEMM, PIC, Sparse, Nstream
    - Uses Stencil and Transpose from the Chapel upstream repo
  - Analyze Chapel's intranode performance on two architectures including KNL
  - Suggest several optimizations in Chapel software stack

PRK Source

- Currently hosted in
  - https://github.com/e-kayrakli/chapel_prk
- Planning to create a PR against Chapel upstream next week
- Once it goes in, create a PR against PRK repo

- Distributed versions are also implemented
  - And used in a recent SC17 submission

Outline

- Introduction & Motivation
- Chapel Primer
  - Implementing Nstream-like Applications
  - More: Chapel Loops, Distributed Data
- Experimental Results
  - Environment, Implementation Caveats
  - Results
- Detailed Analysis
  - Memory Bandwidth Analysis on KNL
  - Idioms & Optimizations For Sparse
  - Optimizations for DGEMM
- Summary & Wrap Up

Full Code For Vector Addition

```
config const N = 10;
var myDomain = {0..#N};

var A: [myDomain] real,
    B: [myDomain] real,
    C: [myDomain] real;

B = 1.5;
C = 2.0;

A = B + C;
```

Variable Declarations

```
config const N = 10;
var myDomain = {0..#N};

var A: [myDomain] real,
    B: [myDomain] real,
    C: [myDomain] real;

B = 1.5;
C = 2.0;

A = B + C;
```

Variables (non-reference ones) can be defined in three different ways:

```
var N = 10 // regular variable
const N = 10 // run-time constant
param N = 10 // compile-time constant
```

All of these can be prepended with `config`, which makes the variable a command line argument that can be set at run-/compile- time.

Type can be implicit or explicit

```
var N:int = 10 // explicit type
```

Domains

```
config const N = 10;
var myDomain = {0..#N};

var A: [myDomain] real,
    B: [myDomain] real,
    C: [myDomain] real;

B = 1.5;
C = 2.0;

A = B + C;
```

Domains are index sets that arrays can be defined on. They can be any-based, strided, sparse, associative and distributed.

Ranges can be defined in two ways:

lower_bound..upper_bound
lower_bound..#count

Arrays

```
config const N = 10;
var myDomain = {0..#N};

var A: [myDomain] real,
    B: [myDomain] real,
    C: [myDomain] real;

B = 1.5;
C = 2.0;

A = B + C;
```

Arrays are defined on domains. They retain the properties of the domain like stridedness, distribution, sparsity, associativity etc.

Arrays

```
config const N = 10;
var myDomain = {0..#N};

var A: [myDomain] real,
    B: [myDomain] real,
    C: [myDomain] real;

B = 1.5;
C = 2.0;

A = B + C;
```

Scalar operators like = are promoted to operate on vector data

Arrays

```
config const N = 10;
var myDomain = {0..#N};

var A: [myDomain] real,
    B: [myDomain] real,
    C: [myDomain] real;

B = 1.5;
C = 2.0;

A = B + C;
```

This single line gets translated into a parallel –and potentially distributed- loop

Explicit forall loop

```
config const N = 10;
var myDomain = {0..#N};

var A: [myDomain] real,
    B: [myDomain] real,
    C: [myDomain] real;

B = 1.5;
C = 2.0;

forall i in myDomain do
        A[i] = B[i] + C[i];
```

`forall` is a parallel loop that creates parallelism as defined by its iterator expression (`myDomain`)

Explicit forall loop with zippering

```
config const N = 10;
var myDomain = {0..#N};

var A: [myDomain] real,
    B: [myDomain] real,
    C: [myDomain] real;

B = 1.5;
C = 2.0;

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

Parallel iterators can be zippered

Explicit coforall

```
config const N = 10;
var myDomain = {0..#N};

var A: [myDomain] real,
    B: [myDomain] real,
    C: [myDomain] real;

B = 1.5;
C = 2.0;

coforall taskId in 0..#here.maxTaskPar {
  var myChunk = computeMyChunk(…);
  for i in myChunk do
    A[i] = B[i] + C[i];
}
```

coforall creates tasks for every iteration -> SPMD region

Distributing the Data

```
use BlockDist;

config const N = 10;

var myDomain = {0..#N} dmapped Block({0..#N});


var A: [myDomain] real,
    B: [myDomain] real,
    C: [myDomain] real;


B = 1.5;

C = 2.0;


A = B + C;
```

`dmapped` clause is used to change the memory layout of the domain

Currently you can define

- distributed
- sparse

domains with this clause.

Note that you do not need any change in the application

- This is also true for forall variants
- But not for coforall variant

Outline

- Introduction & Motivation
- Chapel Primer
  - Implementing Nstream-like Applications
  - More: Chapel Loops, Distributed Data
- **Experimental Results**
  - **Environment, Implementation Caveats**
  - **Results**
- Detailed Analysis
  - Memory Bandwidth Analysis on KNL
  - Idioms & Optimizations For Sparse
  - Optimizations for DGEMM
- Summary & Wrap Up

Test Environment

- Xeon
  - Dual-socket Intel Xeon E5-2630L v2 @2.4GHz
  - 6 core/socket, 15MB LLC/socket
  - 51.2 GB/s memory bandwidth, 32 GB total memory
  - CentOS 6.5, Intel C/C++ compiler 16.0.2
- KNL
  - Intel Xeon Phi 7210 processor
  - 64 cores, 4 thread/core
  - 32MB shared L2 cache
  - 102 GB/s memory bandwidth, 112 GB total memory
  - Memory mode: cache, cluster mode: quadrant
  - CentOS 7.2.1511, Intel C/C++ compiler 17.0.0

Test Environment

- Chapel
  - `6fce63a`
    - between versions 1.14 and 1.15
  - Default settings
    - CHPL_COMM=none, CHPL_TASKS=qthreads, CHPL_LOCALE=flat
  - Intel Compilers
    - Building the Chapel compiler and the runtime system
    - Backend C compiler for the generated code
  - Compilation Flags
    - `fast` – Enables compiler optimizations
    - `replace-array-accesses-with-ref-vars` – replace repeated array accesses with reference variables
- OpenMP
  - All tests are run with environment variable `KMP_AFFINITY=scatter,granularity=fine`
- Data size
  - All benchmarks use ~1GB input data

Caveat: Parallelism in OpenMP vs Chapel

```
#pragma omp parallel
{
  for(iter = 0 ; iter<niter; iter++) {
    if(iter == 1) start_time();
    #pragma omp for
    for(…) {} //application loop
  }
  stop_time();
}
```

- Parallelism introduced early in the flow
- This is how PRK are implemented in OpenMP

Caveat: Parallelism in OpenMP vs Chapel

```
#pragma omp parallel
{
  for(iter = 0 ; iter<niter; iter++) {
    if(iter == 1) start_time();
    #pragma omp for
    for(…) {} //application loop
  }
  stop_time();
}
```

- Parallelism introduced early in the flow
- This is how PRK are implemented in OpenMP

```
coforall t in 0..#numTasks
{
  for iter in 0..#niter {
    if iter == 1 then start_time();
    for … {} //application loop
  }
  stop_time();
}
```

- Corresponding Chapel code
- Feels more “unnatural” in Chapel
- `coforall` loops are (sort of) low-level loops that introduce SPMD regions

Caveat: Parallelism in OpenMP vs Chapel

```
for(iter = 0 ; iter<niter; iter++) {
  if(iter == 1) start_time();
  #pragma omp parallel for
  for(…) {} //application loop
}
stop_time();
```

- Parallelism introduced late in the flow
- Cost of creating parallel regions is accounted for

Caveat: Parallelism in OpenMP vs Chapel

```
for(iter = 0 ; iter<niter; iter++) {
  if(iter == 1) start_time();
  #pragma omp parallel for
  for(…) {} //application loop
}
stop_time();
```

- Parallelism introduced late in the flow
- Cost of creating parallel regions is accounted for

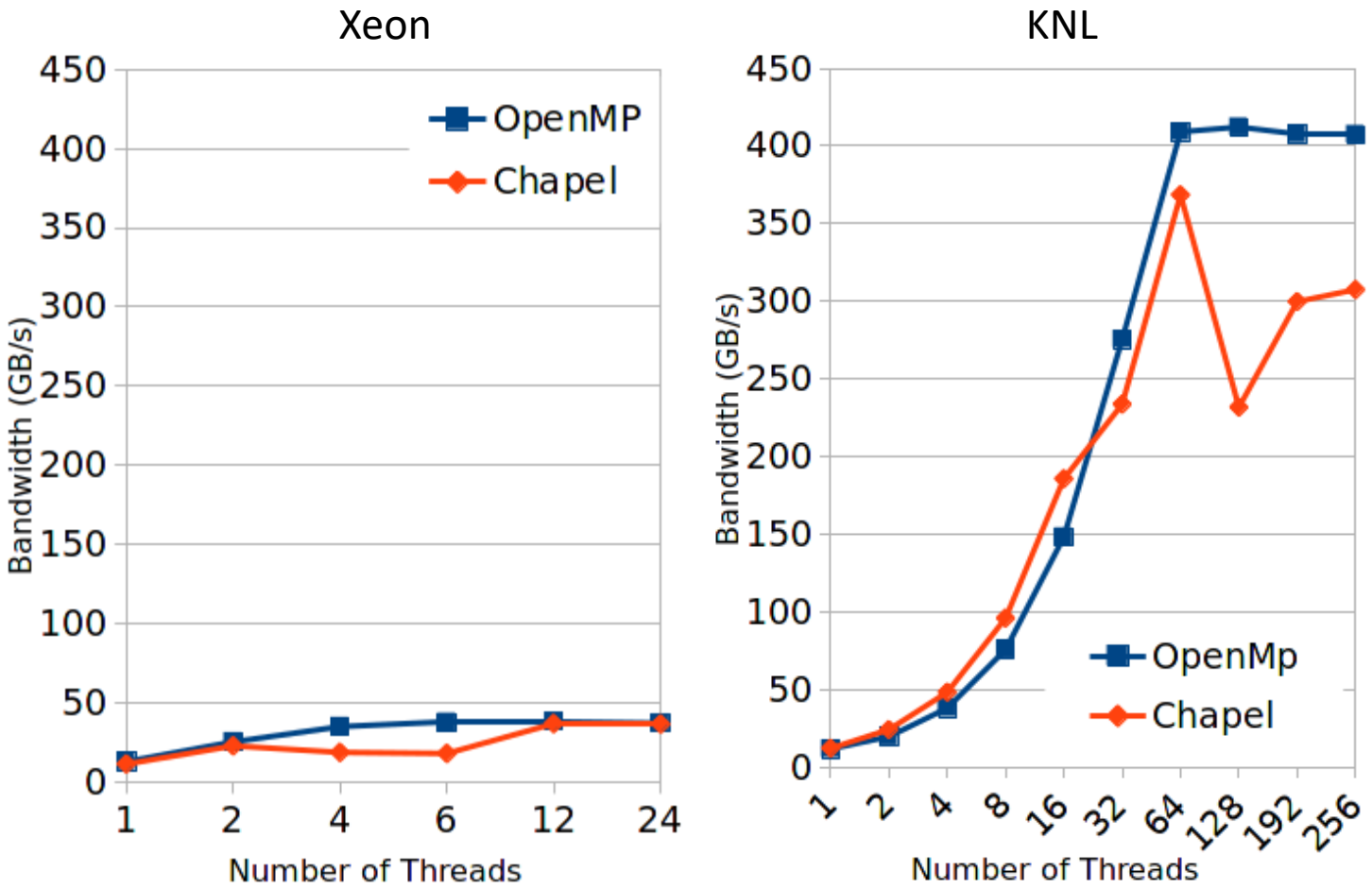
```
for iter in 0..#niter {
  if iter == 1 then start_time();
  forall .. {} //application loop

}
stop_time();
```

- Corresponding Chapel code
- Feels more “natural” in Chapel
- Parallelism is introduced in a data-driven manner by the forall loop
- This is how Chapel PRK are implemented, for now. (Except for blocked DGEMM)

Nstream

- DAXPY kernel based on HPCC-STREAM Triad
- Vectors of 43M doubles

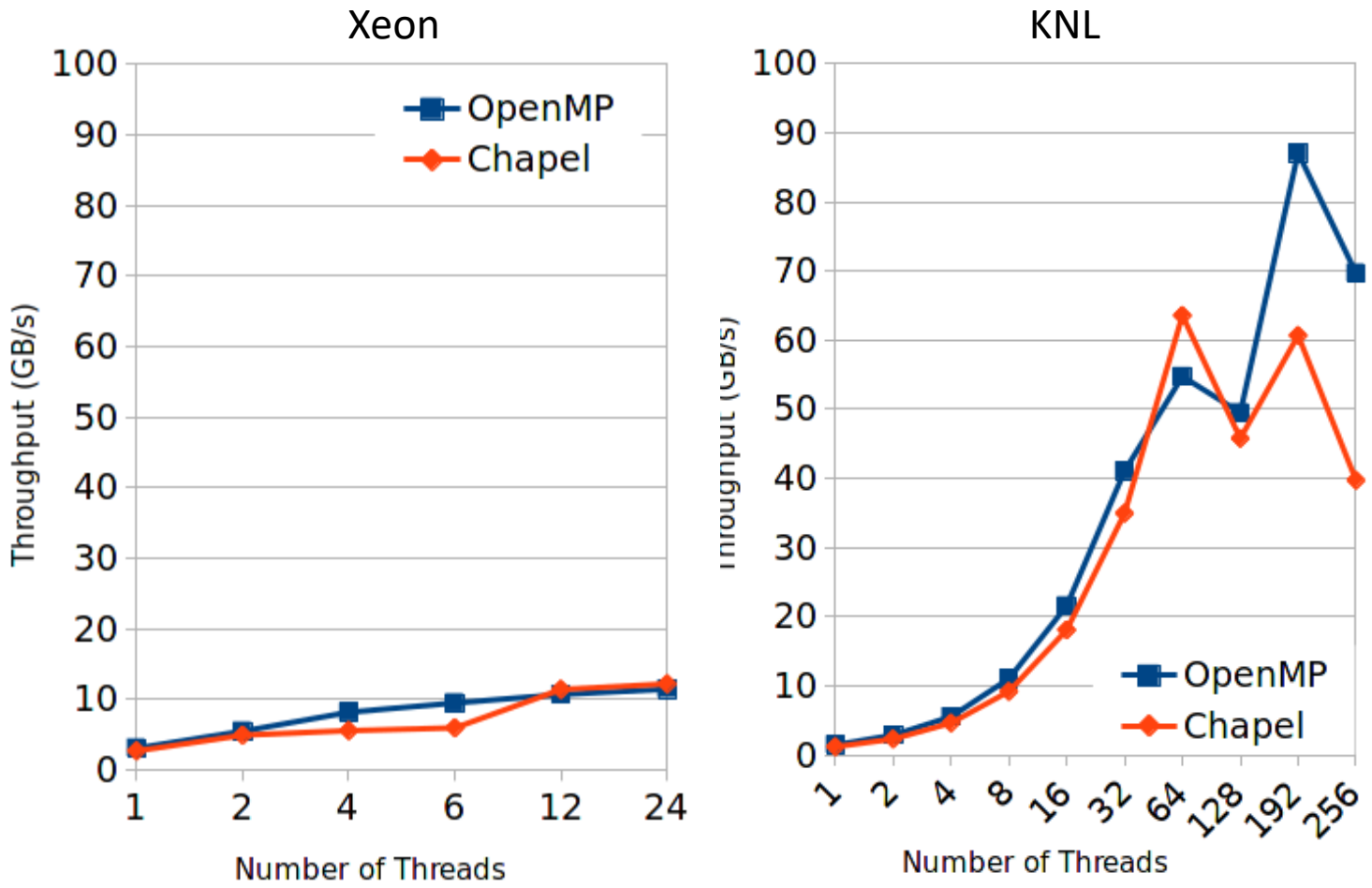
- On Xeon
  - both reach ~40GB/s
- On KNL
  - Chapel reaches 370GB/s
  - OpenMP reaches 410GB/s



Transpose

- Tiled matrix transpose
- Matrices of 8k*8k doubles, tile size is 8

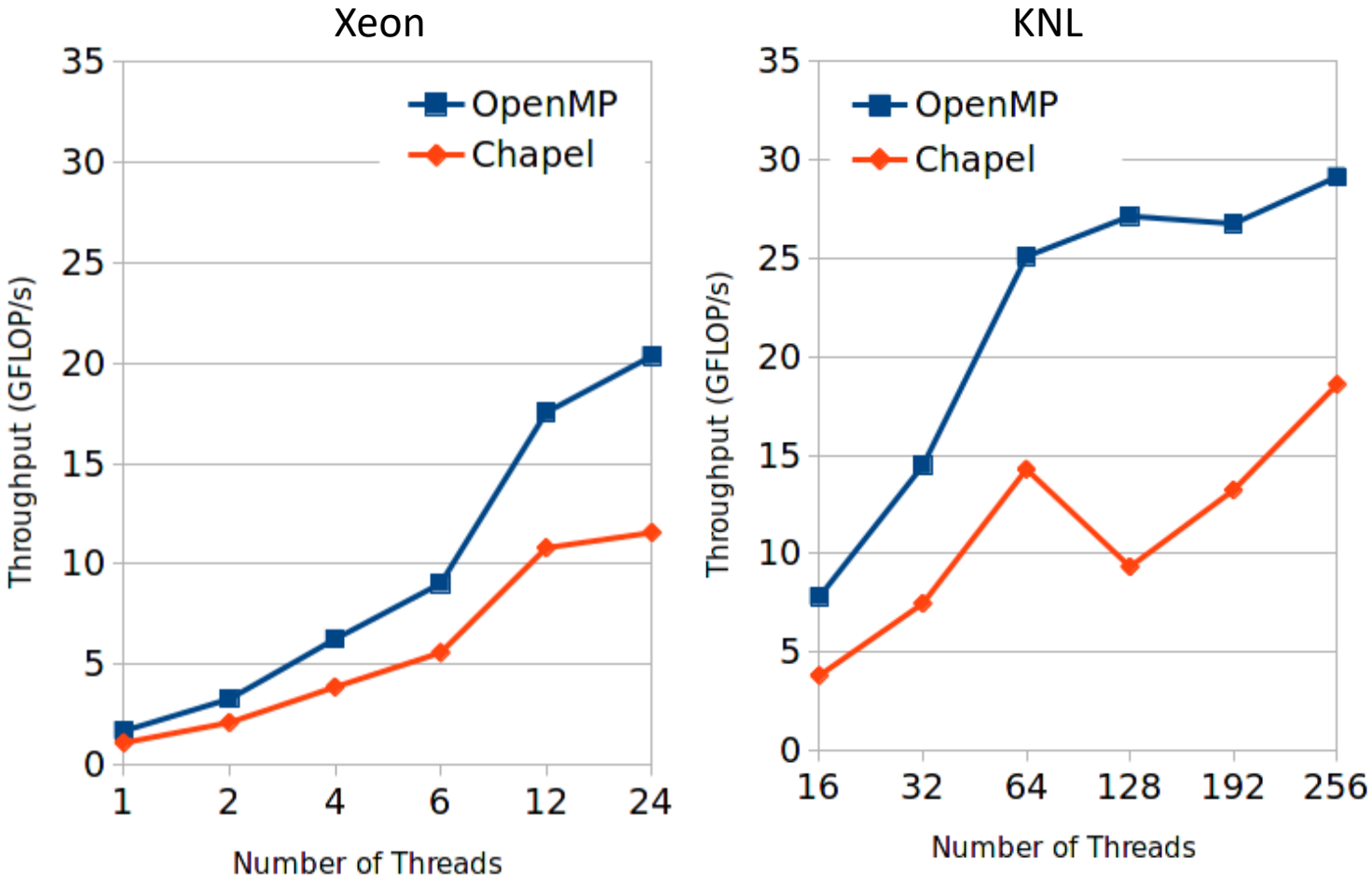
- On Xeon
  - both reach ~10GB/s
- On KNL
  - Chapel reaches 65GB/s
  - OpenMP reaches 85GB/s
  - Chapel struggles more with hyperthreading



DGEMM

- Tiled matrix multiplication
- Matrices of 6530*6530 doubles, tile size is 32

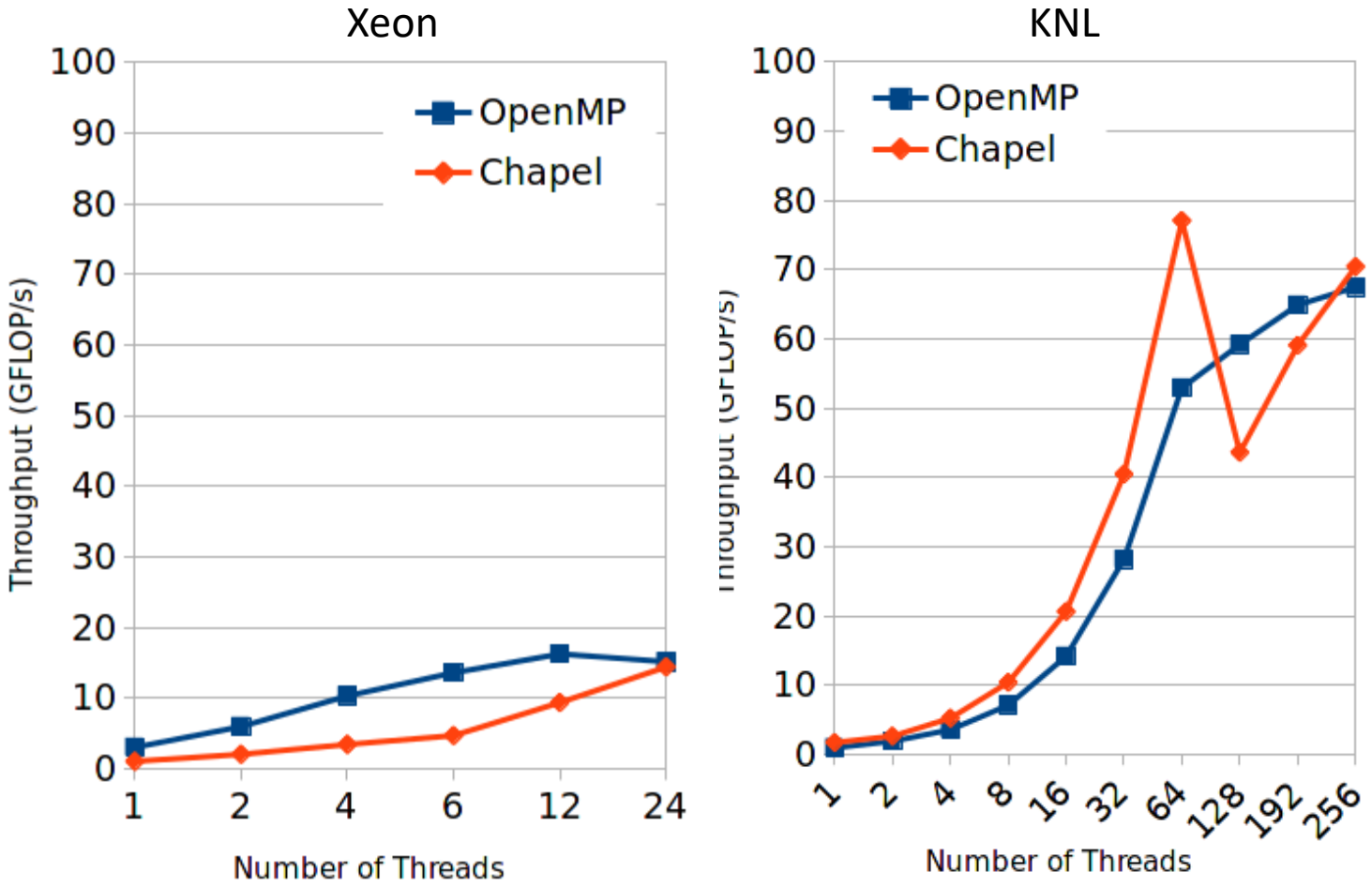
- Chapel reaches ~60% of OpenMP performance on both
- Hyperthreading on KNL is slightly better
- We propose an optimization that brings DGEMM performance much closer to OpenMP



Stencil

- Stencil application on square grid
- Grid is 8000x8000, stencil is star-shaped with radius 2
- OpenMP version is built with `LOOPGEN` and `PARALLELFOR`

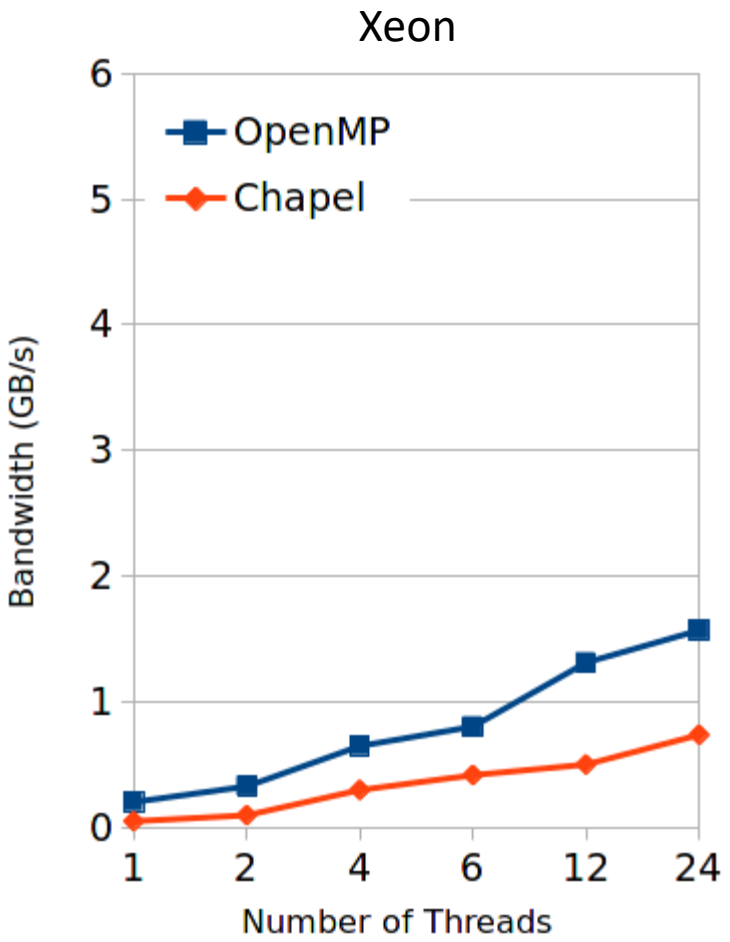
- On Xeon
  - Chapel did not scale well with low number of threads, but reaches 95% of OpenMP
- On KNL
  - Chapel performed better without hyperthreading
  - Relative peak performance is 114% of OpenMP

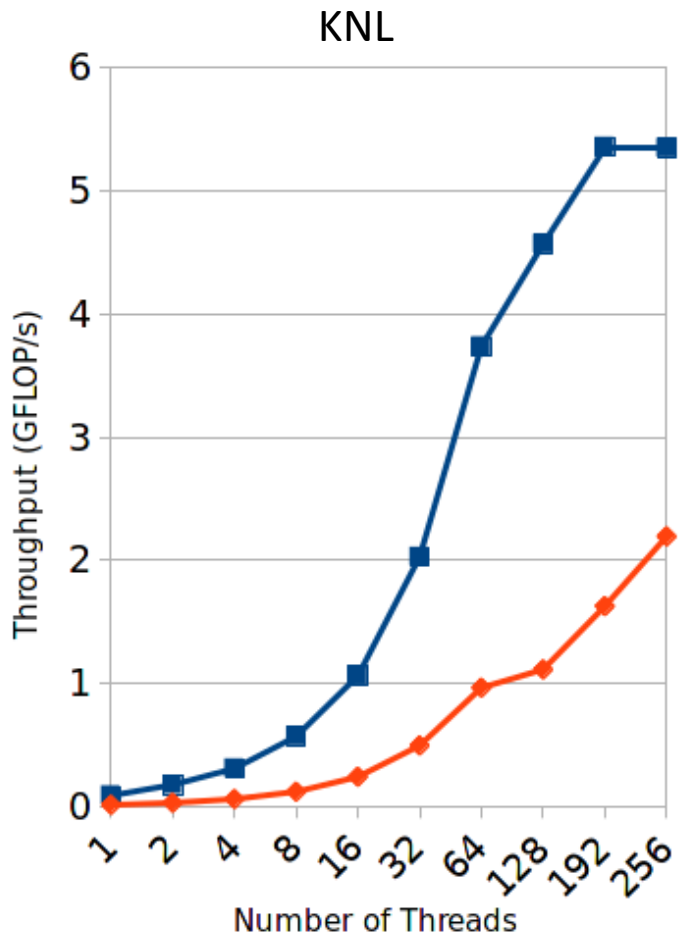


Sparse

- SpMV kernel
- Matrix is $2^{22}x2^{22}$ with 13 nonzeroes per row. Indices are scrambled
- Chapel implementation uses default CSR representation
- OpenMP implementation is vanilla CSR implementation – implemented in application level

- On both architectures, Chapel reached <50% of OpenMP
- We provide detailed analysis of different idioms for Sparse
- Also some optimizations

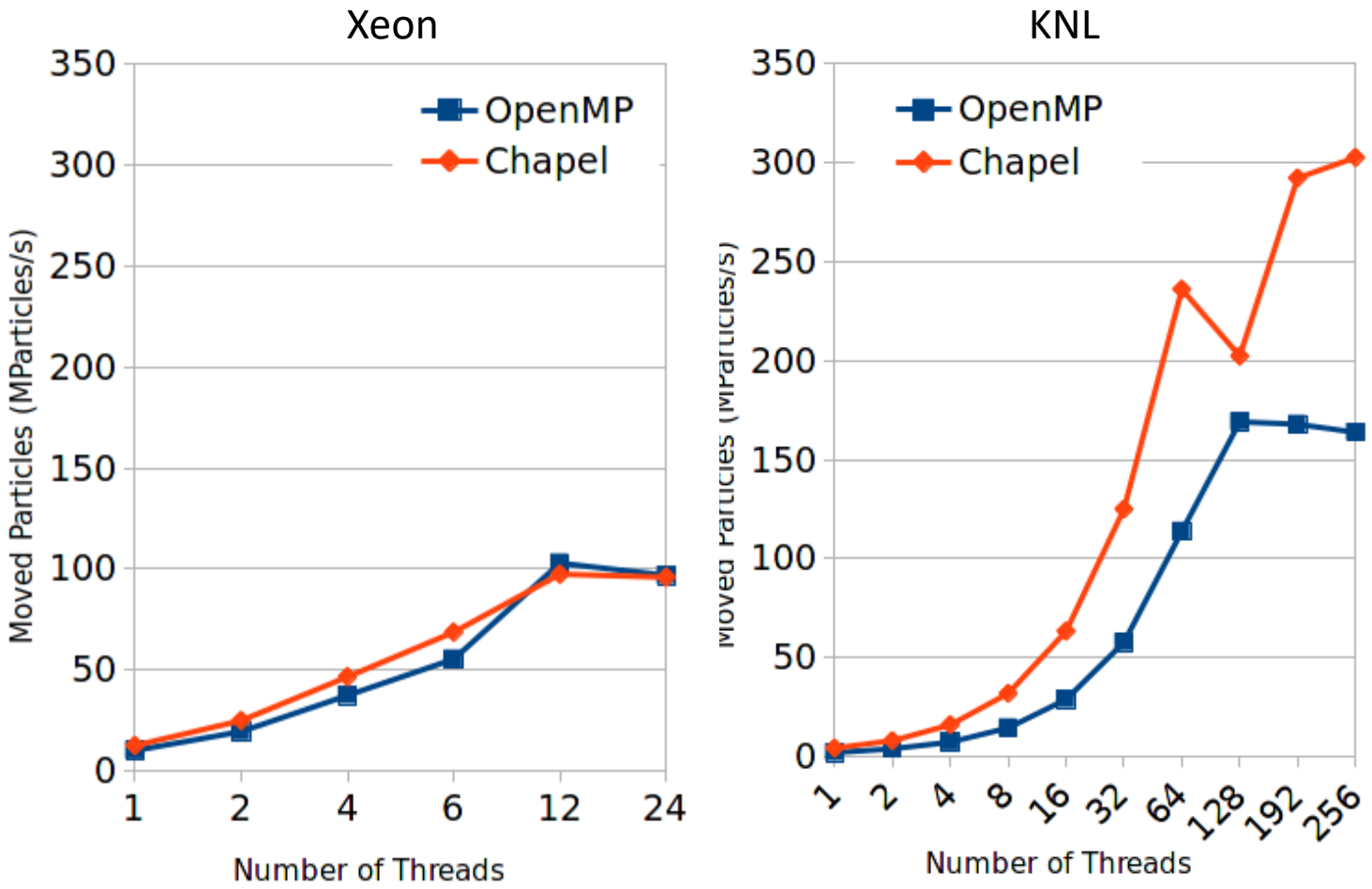




PIC

- Particle-in-cell
- 141M particles requested in a 2^{10}x2^{10} grid
- `SINUSOIDAL, k=1, m=1`

- On Xeon
  - They perform similarly
- On KNL
  - Chapel outperforms OpenMP reaching 184% at peak performance



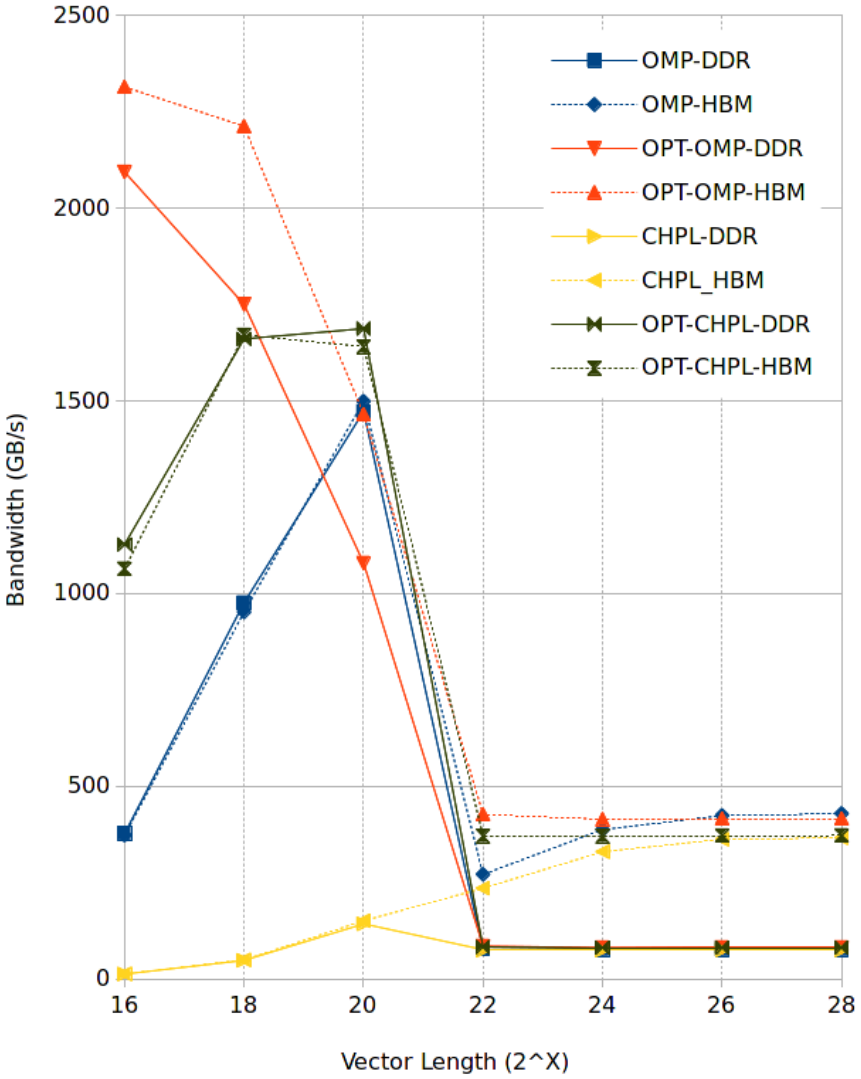
Outline

- Introduction & Motivation
- Chapel Primer
  - Implementing Nstream-like Applications
  - More: Chapel Loops, Distributed Data
- Experimental Results
  - Environment, Implementation Caveats
  - Results
- **Detailed Analysis**
  - **Memory Bandwidth Analysis on KNL**
  - **Idioms & Optimizations For Sparse**
  - **Optimizations for DGEMM**
- Summary & Wrap Up

Memory Bandwidth on KNL

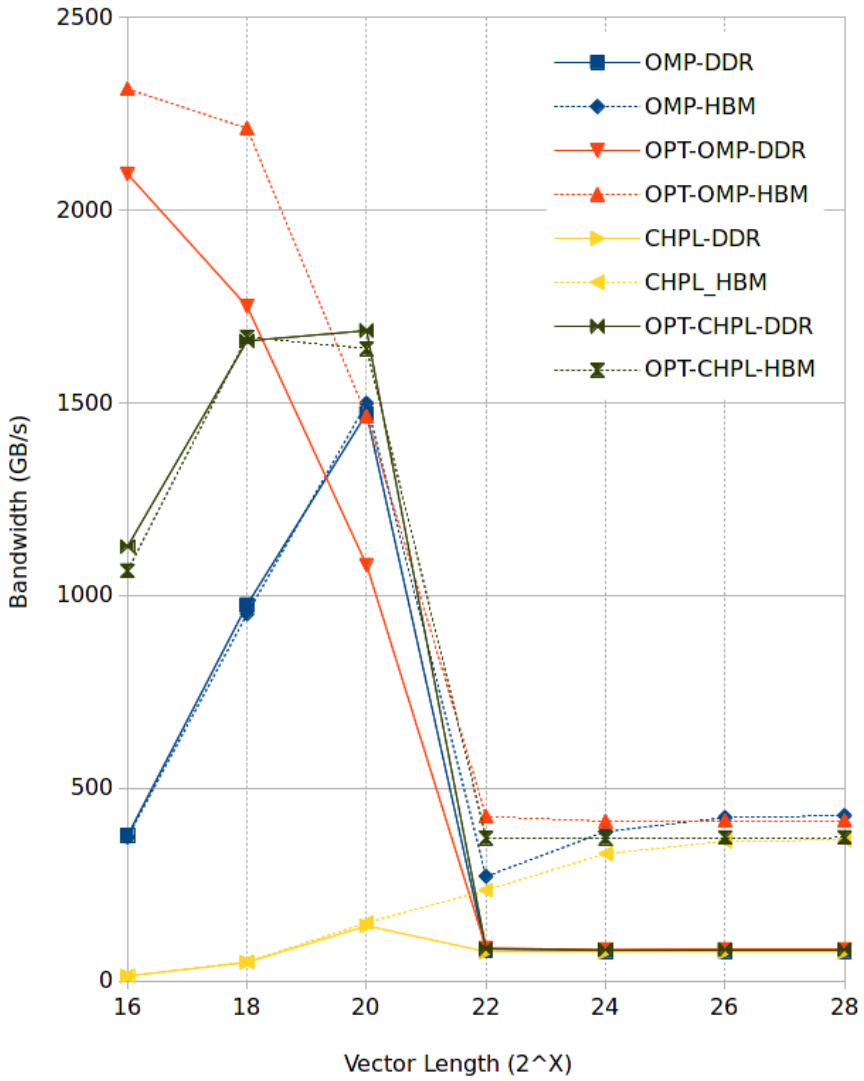
- Varying vector size on Nstream
- Flat memory mode + numactl to control memory mapping

- Versions:
  - CHPL : Nstream with scalar promotion (equivalent to forall)
  - OPT-CHPL : Nstream with coforall
  - OMP : Base Nstream
  - OPT-OMP : Nstream + nowait on the stream loop
  - DDR : numactl -m0
  - HBM : numactl -m1



Memory Bandwidth on KNL

- Different behavior when data size <LLC vs >LLC
- Chapel;
  - forall version is considerably bad with small data
  - coforall version is ~10x times faster – no parallelism cost
- OpenMP;
  - Without nowait, outperformed by coforall version
  - With nowait, outperforms Chapel in smaller data sizes, but not 2^{20}
- When data size is >LLC
  - They both perform similarly on DDR -> ~75 GB/s
  - OpenMP slightly outperforms Chapel -> ~366 GB/s vs ~372 GB/s



Sparse Domains/Arrays in Chapel

```
var denseDom = {1..10, 1..10};
var sparseDom: sparse subdomain(denseDom);
var denseArr: [denseDom] real;
var sparseArr: [sparseDom] real;
sparseDom += (3,5);
sparseArr[3,5] = 5.0;
```

Sparse Domains/Arrays in Chapel

```
var denseDom = {1..10, 1..10};
var sparseDom: sparse subdomain(denseDom);
var denseArr: [denseDom] real;
var sparseArr: [sparseDom] real;
sparseDom += (3,5);
sparseArr[3,5] = 5.0;
```

Sparse domains are declared as "subdomains" of dense ones

Sparse Domains/Arrays in Chapel

```
var denseDom = {1..10, 1..10};
var sparseDom: sparse subdomain(denseDom);
var denseArr: [denseDom] real;
var sparseArr: [sparseDom] real;
sparseDom += (3,5);
sparseArr[3,5] = 5.0;
```

Sparse array declaration is identical

Sparse Domains/Arrays in Chapel

```
var denseDom = {1..10, 1..10};
var sparseDom: sparse subdomain(denseDom);
var denseArr: [denseDom] real;
var sparseArr: [sparseDom] real;
sparseDom += (3,5);
sparseArr[3,5] = 5.0;
```

Indices can be added to sparse domains after creation

Sparse Domains/Arrays in Chapel

```
var denseDom = {1..10, 1..10};
var sparseDom: sparse subdomain(denseDom);
var denseArr: [denseDom] real;
var sparseArr: [sparseDom] real;
sparseDom += (3,5);
sparseArr[3,5] = 5.0;
```

Sparse arrays can be accessed directly with indices

Different Sparse Idioms

- The naïve implementation
- “Hidden” race condition

```
const parentDom = {0..#N, 0..#N};
var matrixDom: sparse subdomain(parentDom)
                    dmapped CSR();
matrixDom += getIndexArray();
var matrix: [matrixDom] real;
forall (i,j) in matrix.domain do
  result[i] += matrix[i,j], vector[j];
```

Different Sparse Idioms

- Parallelism in rows only
- Use dimIter library function
- No race condition

```
const parentDom = {0..#N, 0..#N};
var matrixDom: sparse subdomain(parentDom)
                    dmapped CSR();
matrixDom += getIndexArray();
var matrix: [matrixDom] real;
forall i in matrix.domain.dim(1) do
  for j in matrix.domain.dimIter(2, i) do
    result[i] += matrix[i,j], vector[j];
```

Different Sparse Idioms

- Reduce intents
- Not a good idea
  - The whole vector is a reduction variable
  - But in most common cases race condition would occur in small amount of data
  - Whole vector is copied to tasks and reduced in the end

```
const parentDom = {0..#N, 0..#N};
var matrixDom: sparse subdomain(parentDom)
                    dmapped CSR();
matrixDom += getIndexArray();
var matrix: [matrixDom] real;
forall (i,j) in matrix.domain
                with (+ reduce result) do
  result[i]+=matrix[i,j] * vector[j];
```

Different Sparse Idioms

- Introducing: row distributed sparse iterators
- A compile time flag when defining a sparse domain
- Minor modification in the iterator
  - Chunks are adjusted to avoid dividing rows
  - divideRows is a param, ie compile time constant
  - No branching at runtime
- Not a performance improvement

```
const parentDom = {0..#N, 0..#N};
var matrixDom: sparse subdomain(parentDom)
                dmapped CSR(divideRows=false);
matrixDom += getIndexArray();
var matrix: [matrixDom] real;
forall (i,j) in matrix.domain do
  result[i] += matrix[i,j] * vector[j];
```

Different Sparse Idioms

- Suggested by Brad Chamberlain
- Zip the domain and array so as to avoid the binary search to sparse array
- Still requires row-distributed iterators to avoid the race condition

```
const parentDom = {0..#N, 0..#N};
var matrixDom: sparse subdomain(parentDom)
                dmapped CSR(divideRows=false);
matrixDom += getIndexArray();
var matrix: [matrixDom] real;
forall (elem,(i,j)) in
        zip(matrix, matrix.domain) do
  result[i] += elem * vector[j];
```

Compiler-Injected Fast Access Pointers

- Access to an index of a CSR array requires a binary search
- Simplest sparse kernel

```
forall (i,j) in matrix.domain do
    result[i] += matrix[i,j], vector[j];
```

- Observations
  - Loop iterator is the domain of `matrix`
  - Loop index is the same as the index used to access `matrix`
- Then, within a task, it is guaranteed that elements of matrix is accessed consecutively

Compiler-Injected Fast Access Pointers

No optimization

```
for(i = . . ) {
  for(j = . . ) {
    result_addr = this_ref(result, i);
    matrix_val = this_val(matrix, i, j);
    vector_val = this_val(vector, j);
    *result_addr = *result_addr +
                   matrix_val *
                   vector_val;
  }
}
```

Compiler-Injected Fast Access Pointers

No optimization

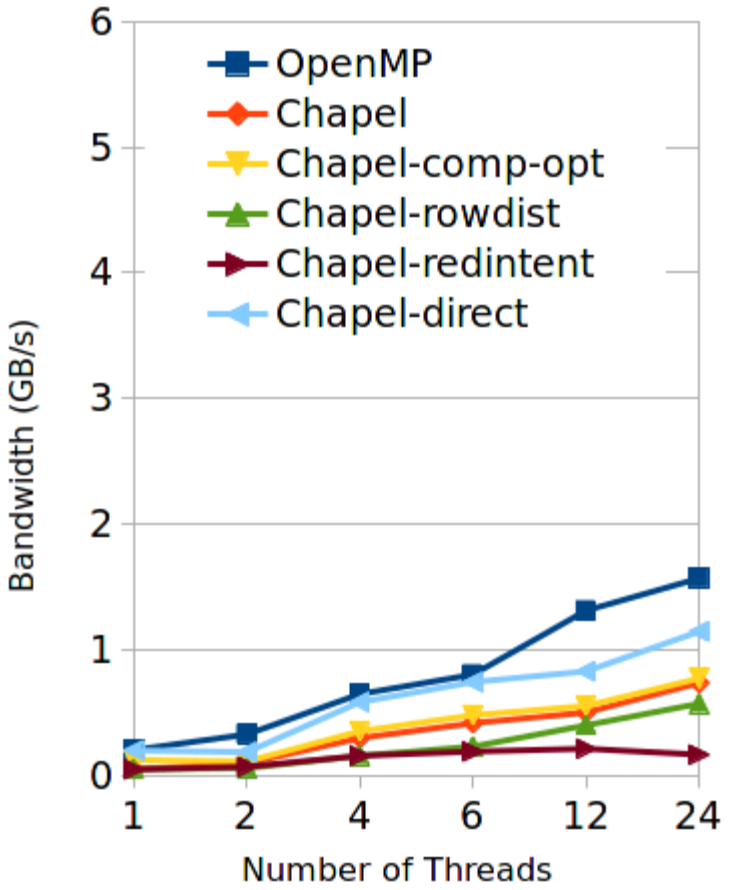
```
for(i = . . ) {
  for(j = . . ) {
    result_addr = this_ref(result, i);
    matrix_val = this_val(matrix, i, j);
    vector_val = this_val(vector, j);
    *result_addr = *result_addr +
                   matrix_val *
                   vector_val;
  }
}
```

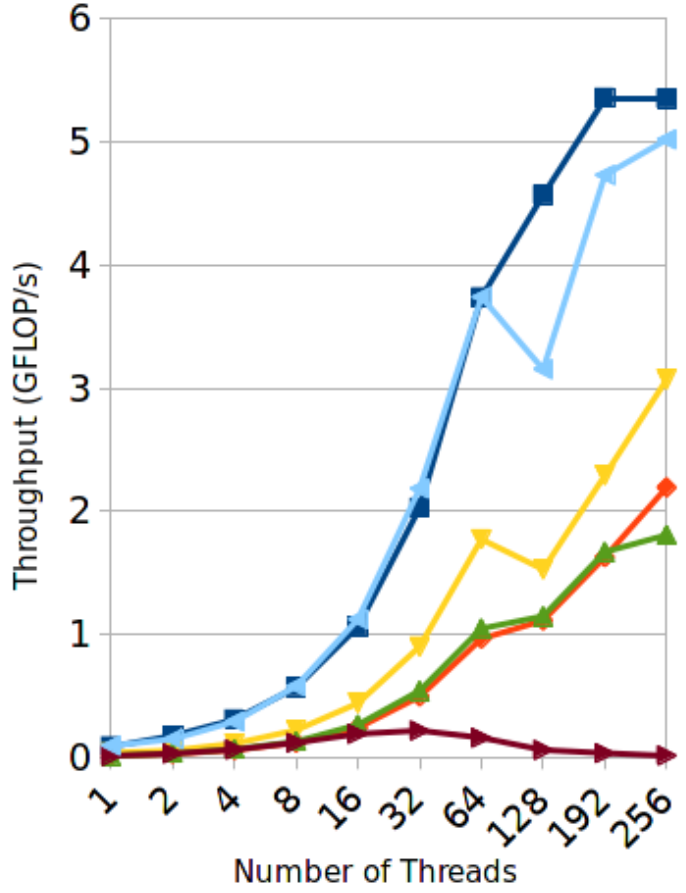
Optimization

```
data_t *fast_acc_ptr = NULL;
for(i = . . ) {
  for(j = . . ) {
    result_addr = this_ref(result, i);
    if(fast_acc_ptr)
      fast_acc_ptr += 1;
    else
      fast_acc_ptr = this_ref(matrix, i, j);
    matrix_val = *fast_acc_ptr;
    vector_val = this_val(vector, j);
    *result_addr = *result_addr +
                   matrix_val *
                   vector_val;
  }
}
```

Detailed Sparse Performance

- Reduce intent performance is abysmal – not surprising
- Row distributed iterators perform similarly to the base
- Compiler optimization is especially good in KNL
  - Possibly due to less/regular memory access by avoiding binary search
- Direct access to the internal CSR arrays is the best
  - Fair: close to what OpenMP implementation is doing
  - Unfair: advanced knowledge/questionable code maintainability





C Arrays For Tiling

- Blocked DGEMM uses Arrays within deeply nested loops
- Generated C code showed some bookkeeping for Chapel arrays not being hoisted to the outer loops
- Use C arrays instead of Chapel arrays
  - More lightweight, less functionality
  - Shouldn’t be a general approach but scope of “tile” arrays is relatively small

Chapel Array vs C Array in DGEMM

Declaration/Initialization

```
var AA: [blockDom] real;
```

```
var AA = c_calloc(real, blockDom.size)
```

Chapel Array vs C Array in DGEMM

Declaration/Initialization

```
var AA: [blockDom] real;
```

```
var AA = c_calloc(real, blockDom.size)
```

Access

```
AA[i,j] = A[iB,jB]
```

```
AA[i*blockSize+j] = A[iB,jB];
```

Chapel Array vs C Array in DGEMM

Declaration/Initialization

```
var AA: [blockDom] real;
```

```
var AA = c_calloc(real, blockDom.size)
```

Access

```
AA[i,j] = A[iB,jB]
```

```
AA[i*blockSize+j] = A[iB,jB];
```

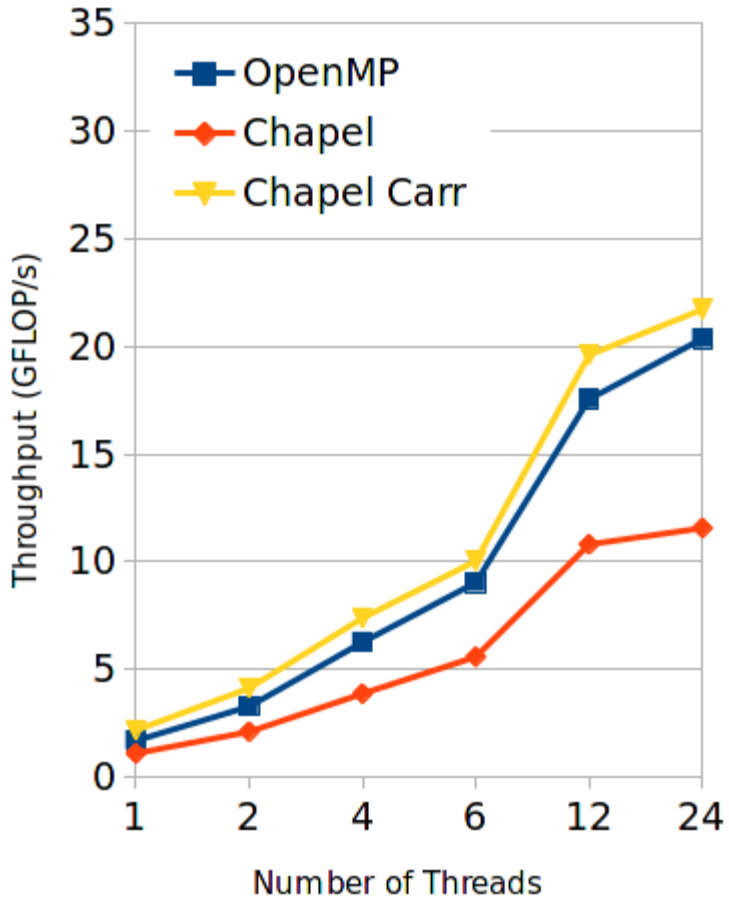
Deallocation

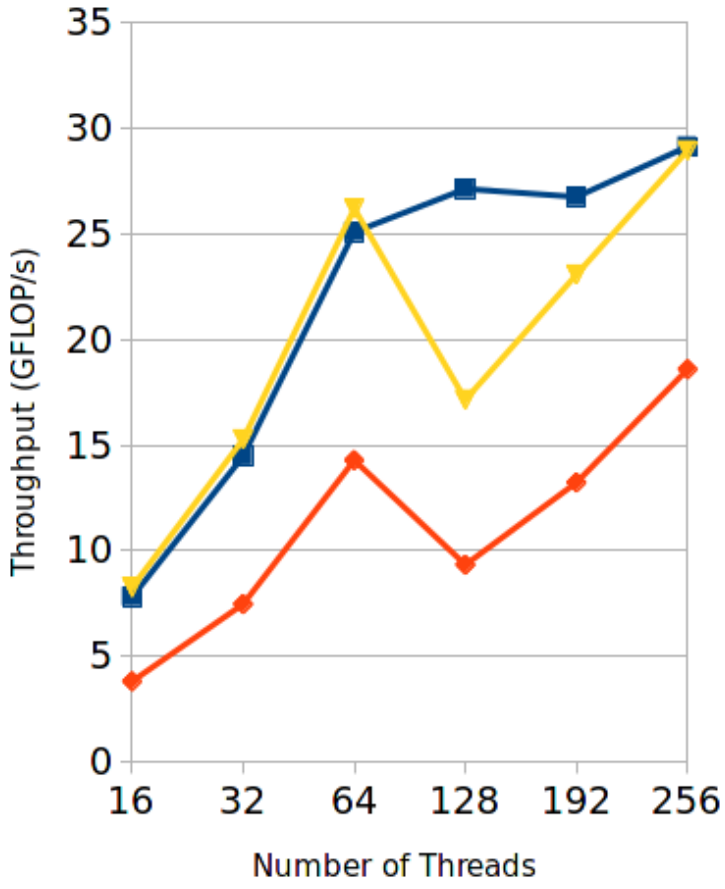
N/A

```
c_free(AA);
```

Detailed DGEMM Performance

- Optimized version perform slightly better than OpenMP
  - Except for 2-3 threads/core on KNL
- Performance improvement is 2x on Xeon and 1.6x on KNL





Outline

- Introduction & Motivation
- Chapel Primer
  - Implementing Nstream-like Applications
  - More: Chapel Loops, Distributed Data
- Experimental Results
  - Environment, Implementation Caveats
  - Results
- Detailed Analysis
  - Memory Bandwidth Analysis on KNL
  - Idioms & Optimizations For Sparse
  - Optimizations for DGEMM
- **Summary & Wrap Up**

Summary & Wrap Up

- Except for Transpose relative Chapel performance is better on KNL
  - Transpose: No computation, memory bound, mix of sequential and strided accesses
- Stencil and PIC
  - Chapel outperforms OpenMP on KNL
- Optimizations
  - Up to 2x performance improvement
  - DGEMM performance is similar to OpenMP
  - Sparse performance gap is smaller

	Xeon		KNL	
	Base	Opt	Base	Opt
Nstream	100%	-	100%	-
Transpose	106%	-	72%	-
DGEMM	56%	106%	63%	99%
Stencil	95%	-	114%	-
Sparse	41%	73%	47%	93%
PIC	94%	-	184%	-

Summary & Wrap Up

- Chapel is an emerging parallel programming language
- Manycore is an emerging architectural concept
- We analyzed Chapel's intranode performance
  - In comparison to widely-accepted OpenMP
  - Using the Parallel Research Kernels
- In some PRK workloads Chapel outperforms OpenMP out-of-the-box
- In some we offered user-level and/or language level improvements
  - Brings performance mostly on par with OpenMP

Acknowledgement

The authors would like to thank Rob F. Van der Wijngaart and Jeff R. Hammond for many useful discussions and insights that contributed to the quality of this paper.

Thank You

Full Paper References

[1] B. Chamberlain, D. Callahan, and H. Zima, “Parallel Programmability and the Chapel Language,” Int. J. High Perform. Comput. Appl., vol. 21, no. 3, pp. 291–312, Aug. 2007.

[2] T. El-Ghazawi, W. Carlson, T. Sterling, and K. Yelick, UPC: Distributed Shared-Memory Programming. Wiley-Interscience, 2003.

[3] A. Shterenlikht, L. Margetts, L. Cebamanos, and D. Henty, “Fortran 2008 coarrays,” SIGPLAN Fortran Forum, vol. 34, no. 1, pp. 10–30, Apr. 2015.

[4] K. Ebcioglu, V. Saraswat, and V. Sarkar, “X10: Programming for hierarchical parallelism and non-uniform data access,” in Proceedings of the International Workshop on Language Runtimes, OOPSLA, vol. 30. Citeseer, 2004.

[5] B. Chapman, T. Curtis, S. Pophale, S. Poole, J. Kuehn, C. Koelbel, and L. Smith, “Introducing OpenSHMEM: SHMEM for the PGAS Community,” in Proceedings of the Fourth Conference on Partitioned Global Address Space Programming Model. New York, NY, USA: ACM, 2010, pp. 2:1–2:3.

[6] Y. Zheng, A. Kamil, M. Driscoll, H. Shan, and K. Yelick, “UPC++: A PGAS Extension for C++,” in Parallel and Distributed Processing Symposium, 2014 IEEE 28th International, May 2014, pp. 1105–1114.

[7] K. Fürlinger, T. Fuchs, and R. Kowalewski, “DASH: A C++ PGAS Library for Distributed Data Structures and Parallel Algorithms,” in Proceedings of the 18th IEEE International Conference on High Performance Computing and Communications (HPCC 2016), Sydney, Australia, 2016.

[8] R. Diaconescu and H. Zima, “An Approach To Data Distributions in Chapel,” International Journal of High Performance Computing Applications, vol. 21, no. 3, pp. 313–335, Aug. 2007.

[9] B. L. Chamberlain, S.-e. Choi, S. J. Deitz, D. Iten, and V. Litvinov, “Authoring user-defined domain maps in chapel,” in Proceedings of Cray Users Group, 2011.

[10] “TOP500 Supercomputer Sites,” http://top500.org, [Online; accessed 17-Jan-2017].

[11] A. Anbar, O. Serres, E. Kayraklioglu, A.-H. A. Badawy, and T. El-Ghazawi, “Exploiting Hierarchical Locality in Deep Parallel Architectures,” ACM Transactions on Architecture and Code Optimization (TACO), vol. 13, no. 2, p. 16, 2016.

[12] R. F. V. d. Wijngaart and T. G. Mattson, “The Parallel Research Kernels,” in 2014 IEEE High Performance Extreme Computing Conference (HPEC), Sep. 2014, pp. 1–6.

[13] L. V. Kale and S. Krishnan, CHARM++: a portable concurrent object oriented system based on C++. ACM, 1993, vol. 28, no. 10.

[14] Z. DeVito, N. Joubert, F. Palacios, S. Oakley, M. Medina, M. Barrientos, E. Elsen, F. Ham, A. Aiken, K. Duraisamy, E. Darve, J. Alonso, and P. Hanrahan, “Liszt: A Domain Specific Language for Building Portable Mesh-based PDE Solvers,” in Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis, ser. SC ’11. New York, NY, USA: ACM, 2011, pp. 9:1–9:12.

[15] E. A. Luke and T. George, “Loci: A Rule-based Framework for Parallel Multi-disciplinary Simulation Synthesis,” Journal of Functional Programming, vol. 15, no. 3, pp. 477–502, May 2005.

[16] I. Karlin, A. Bhatele, J. Keasler, B. L. Chamberlain, J. Cohen, Z. DeVito, R. Haque, D. Laney, E. Luke, F. Wang, and others, “Exploring traditional and emerging parallel programming models using a proxy application,” in Parallel & Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on. IEEE, 2013, pp. 919–932.

Full Paper References cont.

[17] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou, “Cilk: An Efficient Multithreaded Runtime System,” in Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. New York, NY, USA: ACM, 1995, pp. 207–216.

[18] “The Go Programming Language,” http://golang.org, [Online; accessed 16-Jan-2017].

[19] J. Reinders, Intel Threading Building Blocks, 1st ed. Sebastopol, CA, USA: O’Reilly & Associates, Inc., 2007.

[20] S. Nanz, S. West, K. S. d. Silveira, and B. Meyer, “Benchmarking Usability and Performance of Multicore Languages,” in 2013 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, Oct. 2013, pp. 183–192.

[21] R. B. Johnson and J. Hollingsworth, “Optimizing Chapel for Single-Node Environments,” in 2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), May 2016, pp. 1558–1567.

[22] E. Kayraklioglu and T. El-Ghazawi, “Assessing Memory Access Performance of Chapel through Synthetic Benchmarks,” in 2015 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid), May 2015, pp. 1147–1150.

[23] E. Kayraklioglu, O. Serres, A. Anbar, H. Elezabi, and T. El-Ghazawi, “PGAS Access Overhead Characterization in Chapel,” in 2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW). IEEE, 2016, pp. 1568–1577.

[24] J. Nelson, B. Holt, B. Myers, P. Briggs, L. Ceze, S. Kahan, and M. Oskin, “Grappa: A latency-tolerant runtime for large-scale irregular applications,” in International Workshop on Rack-Scale Computing (WRSC w/EuroSys), 2014.

[25] R. F. V. d. Wijngaart, A. Kayi, J. R. Hammond, G. Jost, T. S. John, S. Sridharan, T. G. Mattson, J. Abercrombie, and J. Nelson, “Comparing Runtime Systems with Exascale Ambitions Using the Parallel Research Kernels,” in High Performance Computing, ser. Lecture Notes in Computer Science. Springer International Publishing, Jun. 2016, pp. 321–339.

[26] R. F. V. d. Wijngaart, S. Sridharan, A. Kayi, G. Jost, J. R. Hammond, T. G. Mattson, and J. E. Nelson, “Using the Parallel Research Kernels to Study PGAS Models,” in 2015 9th International Conference on Partitioned Global Address Space Programming Models, Sep. 2015, pp. 76–81.

[27] D. Doerfler, J. Deslippe, S. Williams, L. Oliker, B. Cook, T. Kurth, M. Lobet, T. Malas, J.-L. Vay, and H. Vincenti, “Applying the Roofline Performance Model to the Intel Xeon Phi Knights Landing Processor,” in High Performance Computing, ser. Lecture Notes in Computer Science. Springer International Publishing, 2016, pp. 339–353.

[28] I. Surmin, S. Bastrakov, Z. Matveev, E. Efimenko, A. Gonoskov, and I. Meyerov, “Co-design of a Particle-in-Cell Plasma Simulation Code for Intel Xeon Phi: A First Look at Knights Landing,” in Algorithms and Architectures for Parallel Processing, ser. Lecture Notes in Computer Science. Springer International Publishing, 2016, pp. 319–329.

[29] A. Heinecke, A. Breuer, M. Bader, and P. Dubey, “High Order Seismic Simulations on the Intel Xeon Phi Processor (Knights Landing),” in High Performance Computing, ser. Lecture Notes in Computer Science, J. M. Kunkel, P. Balaji, and J. Dongarra, Eds. Springer International Publishing, 2016, pp. 343–362.

Full Paper References cont.

[30] “Intel xeon phi processor high performance programming (second edition),” J. Jeffers, J. Reinders, and A. Sodani, Eds. Boston: Morgan Kaufmann, 2016.

[31] B. L. Chamberlain, S. J. Deitz, D. Iten, and S.-E. Choi, “User-defined distributions and layouts in chapel: Philosophy and framework,” in Proceedings of the 2Nd USENIX Conference on Hot Topics in Parallelism, HotPar’10. Berkeley, CA, USA: USENIX Association, 2010, pp. 12–12.

[32] B. L. Chamberlain, S.-e. Choi, S. J. Deitz, and A. Navarro, “User-Defined Parallel Zippered Iterators in Chapel,” in Proceedings of the Fifth Conference on Partitioned Global Address Space Programming Model, 2011.

[33] “Reduce Intents - Chapel Documentation 1.14,” http://chapel.cray.com/docs/1.14/technotes/reduceIntents.htmlxh, [Online; accessed 24-Jan-2017].