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
• 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
  • 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
The Paper

• 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
    • All changes have been merged to master: Pull requests 6152, 6153, 6165
    • test/studies/prk
  • Analyzes Chapel’s intranode performance on two architectures including KNL
  • Suggests several optimizations in Chapel software stack
Outline

• Introduction & Motivation

• 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

```c
#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

```cpp
#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

```chapel
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

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

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

```c
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

```c
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();

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

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

• 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)
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();
```

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

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

Synchronization is already similar
Outline

• Introduction & Motivation

• Experimental Results
  • Environment, Implementation Caveats
  • Results

• Detailed Analysis
  • Memory Bandwidth Analysis on KNL
  • Idioms & Optimizations For Sparse
  • Optimizations for DGEMM

• Summary & Wrap Up
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
  - Better without hyperthreading
  - Peak performance is 114% of OpenMP
Sparse

• SpMV kernel
• Matrix is $2^{22} \times 2^{22}$ with 13 nonzeros 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 $2^{10} \times 2^{10}$ grid
- SINUSOIDAL, $k=1$, $m=1$

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

• Introduction & Motivation
• 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
Outline

• Introduction & Motivation
• Experimental Results
  • Environment, Implementation Caveats
  • Results
• Detailed Analysis
  • Memory Bandwidth Analysis on KNL
  • Idioms & Optimizations For Sparse
  • Optimizations for DGEMM
• Summary & Wrap Up
Different Sparse Idioms

- The naïve implementation
- Somewhat elusive race condition

```plaintext
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

```plaintext
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

```cpp
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

```c
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

```plaintext
const parentDom = {0..#N, 0..#N};
var matrixDom: sparse subdomain(parentDom)
    dmapped CSR(divideRows=false);
matrixDom += getIndexOfArray();
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

```plaintext
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

```c
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

```c
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

```c
data_t *fast_acc_ptr = NULL;
for(i = . . ) {
  for(j = . . ) {
    result_addr = this_ref(result, i);
    matrix_val = this_ref(matrix, i, j);
    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
Outline

• Introduction & Motivation
• Experimental Results
  • Environment, Implementation Caveats
  • Results

• Detailed Analysis
  • Memory Bandwidth Analysis on KNL
  • Idioms & Optimizations For Sparse
  • Optimizations for DGEMM
• Summary & Wrap Up
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

```chapel
var AA: [blockDom] real;
var AA = c_calloc(real, blockDom.size)
```
Chapel Array vs C Array in DGEMM

<table>
<thead>
<tr>
<th>Declaration/Initialization</th>
<th>Access</th>
</tr>
</thead>
<tbody>
<tr>
<td>var AA = c_calloc(real, blockDom.size)</td>
<td>AA[i*blockSize+j] = A[iB, jB];</td>
</tr>
</tbody>
</table>
Chapel Array vs C Array in DGEMM

<table>
<thead>
<tr>
<th></th>
<th>Declaration/Initialization</th>
<th>Access</th>
<th>Deallocation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chapel</td>
<td>var AA: [blockDom] real;</td>
<td>AA[i, j] = A[iB, jB]</td>
<td>N/A</td>
</tr>
<tr>
<td>C</td>
<td>var AA = c_calloc(real, blockDom.size)</td>
<td>AA[i*blockSize+j] = A[iB, jB];</td>
<td>c_free(AA);</td>
</tr>
</tbody>
</table>
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

<table>
<thead>
<tr>
<th></th>
<th>Xeon</th>
<th>KNL</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Base</td>
<td>Opt</td>
</tr>
<tr>
<td>Nstream</td>
<td>100%</td>
<td>-</td>
</tr>
<tr>
<td>Transpose</td>
<td>106%</td>
<td>-</td>
</tr>
<tr>
<td>DGEMM</td>
<td>56%</td>
<td>106%</td>
</tr>
<tr>
<td>Stencil</td>
<td>95%</td>
<td>-</td>
</tr>
<tr>
<td>Sparse</td>
<td>41%</td>
<td>73%</td>
</tr>
<tr>
<td>PIC</td>
<td>94%</td>
<td>-</td>
</tr>
</tbody>
</table>
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 cont.


