TORSTEN HOEFLER, SABELA RAMOS, TAL BEN-NUN, AND SPCL’S DAPP TEAM

HPC Performance Optimization Advances at Extreme Scale
Capability Models for Manycore Memory Systems: A Case-Study with Xeon Phi KNL
Microarchitectures are becoming more and more complex
How to optimize codes for these complex architectures?

- **Performance engineering**: “encompasses the set of roles, skills, activities, practices, tools, and deliverables applied at every phase of the systems development life cycle which ensures that a solution will be designed, implemented, and operationally supported to meet the non-functional requirements for performance (such as throughput, latency, or memory usage).”

- **Manually profile codes and tune them to the given architecture**
  - Requires highly-skilled performance engineers
  - Need familiarity with
    - NUMA (topology, bandwidths etc.)
    - Caches (associativity, sizes etc.)
    - Microarchitecture (number of outstanding loads etc.)
An engineering example – Tacoma Narrows Bridge
Scientific Performance Engineering

1) Observe

2) Model

3) Understand

4) Build
Scientific Performance Engineering

Application Model

\[ f(p) = \sum_{k=1}^{n} c_k \times p^i_k \times \log_2^j(p) \]
- Automatic application modeling for optimization and co-design

Calotoiu et al., SC’14

Compute Node Model
- PCIe model
- Used for multi-GPU system

Martinasso et al., SC’16

Network Model
- Close-to-optimal network topology design

Besta et al., SC’14

CPU Model
- Detailed memory architecture models for development

Ramos et al., IPDPS’17
Modeling by example: KNL Architecture (mesh)
KNL Architecture (memory: Flat & Cache)
KNL Architecture (all to all mode)
KNL Architecture (Quadrant or Hemisphere)
KNL Architecture (SNC-4 or SNC-2)
KNL Architecture

How much does this all matter?
What is the real cost of accessing cache?
What is the cost of accessing memory?
Step 1: Understand core-to-core transfers – MESIF cache coherence

<table>
<thead>
<tr>
<th>Latency [ns] (Copy/BenchIT)</th>
<th>Local (L1)</th>
<th>Tile (L2)</th>
<th>Remote</th>
</tr>
</thead>
<tbody>
<tr>
<td>SNC4</td>
<td>3.8</td>
<td>34 (M)</td>
<td>107-122 (M)</td>
</tr>
<tr>
<td>SNC2</td>
<td>3.8</td>
<td>34 (M)</td>
<td>111-125 (M)</td>
</tr>
<tr>
<td></td>
<td>17 (E)</td>
<td>18 (E)</td>
<td>98-114 (E)</td>
</tr>
<tr>
<td></td>
<td>14 (S,F)</td>
<td>14 (S,F)</td>
<td>96-118 (S,F)</td>
</tr>
</tbody>
</table>

| Write back overhead |

<table>
<thead>
<tr>
<th>Bandwidth [GB/s] (Copy)</th>
<th>Tile</th>
<th>Remote</th>
</tr>
</thead>
<tbody>
<tr>
<td>SNC4</td>
<td>6.7  (M)</td>
<td>7.7</td>
</tr>
<tr>
<td>SNC2</td>
<td>6.7  (M)</td>
<td>6.7</td>
</tr>
<tr>
<td></td>
<td>7.6  (E)</td>
<td>6.7</td>
</tr>
<tr>
<td></td>
<td>7.7</td>
<td>7.7</td>
</tr>
</tbody>
</table>

| Location: only 5-15% difference |

<table>
<thead>
<tr>
<th>Contention [ns] (1:N copy)</th>
<th>α</th>
<th>200</th>
<th>200</th>
<th>200</th>
<th>200</th>
<th>200</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linear, $\mathcal{T}_C(N) = \alpha + \beta \cdot N$</td>
<td>β</td>
<td>34</td>
<td>34</td>
<td>34</td>
<td>34</td>
<td>34</td>
</tr>
</tbody>
</table>

That is curious!

All values are medians within 10% of the 95% nonparametric CI, cf. TH, RB: “Scientific Benchmarking of Parallel Computing Systems”, SC16
### Step 2: Understand core-to-memory transfers – DRAM and MCDRAM

<table>
<thead>
<tr>
<th></th>
<th>Software NUMA</th>
<th></th>
<th>Software UMA</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>SNC4</td>
<td>SNC2</td>
<td>QUAD</td>
<td>HEM</td>
</tr>
<tr>
<td>Latency [ns] (BenchIT)</td>
<td>DRAM 130-140, 160-175</td>
<td>MCDRAM 134-146, 160-170</td>
<td>DRAM 140, 167</td>
<td>MCDRAM 140, 167</td>
</tr>
<tr>
<td>Bandwidth [GB/s] (Copy NT / STREAM Copy)</td>
<td>DRAM 69 / 77, 342 / 418</td>
<td>MCDRAM 69 / 77, 333 / 388</td>
<td>DRAM 70 / 77, 333 / 415</td>
<td>MCDRAM 71 / 77, 315 / 372</td>
</tr>
<tr>
<td>Bandwidth [GB/s] (Read)</td>
<td>DRAM 71, 243</td>
<td>MCDRAM 71, 288</td>
<td>DRAM 77, 314</td>
<td>MCDRAM 77, 314</td>
</tr>
<tr>
<td>Bandwidth [GB/s] (Write)</td>
<td>DRAM 33, 147</td>
<td>MCDRAM 34, 163</td>
<td>DRAM 36, 171</td>
<td>MCDRAM 36, 165</td>
</tr>
<tr>
<td>Bandwidth [GB/s] (Triad NT / STREAM Triad)</td>
<td>DRAM 71 / 82, 371 / 448</td>
<td>MCDRAM 71 / 82, 347 / 441</td>
<td>DRAM 74 / 82, 340 / 441</td>
<td>MCDRAM 73 / 82, 332 / 434</td>
</tr>
<tr>
<td>Latency [ns] (BenchIT)</td>
<td>DRAM 158-178, 161-171</td>
<td>MCDRAM 166, 168</td>
<td>DRAM 166, 168</td>
<td>MCDRAM 172</td>
</tr>
<tr>
<td>Bandwidth [GB/s] (Copy NT / STREAM Copy)</td>
<td>150 / 252, 130 / 252</td>
<td>175 / 255, 134 / 237, 132 / 233</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bandwidth [GB/s] (Read)</td>
<td>87, 56</td>
<td>95, 56</td>
<td>124, 72</td>
<td>128, 72</td>
</tr>
<tr>
<td>Bandwidth [GB/s] (Write)</td>
<td>87, 56</td>
<td>95, 56</td>
<td>124, 72</td>
<td>128, 72</td>
</tr>
<tr>
<td>Bandwidth [GB/s] (Triad NT / STREAM Triad)</td>
<td>296 / 292, 246 / 294, 296 / 309</td>
<td>273 / 274, 264 / 269</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

MCDRAM 20% slower!

MCDRAM 4-6x faster!

Need to read and write for full bandwidth

Cache mode >20% slower

Bandwidth suffers a bit

All values are medians within 10% of the 95% nonparametric CI, cf. TH, RB: “Scientific Benchmarking of Parallel Computing Systems”, SC16
Performance engineers optimize your code!

Are you kidding me?

- **Table:**
  - **Software NUMA**
    - SNC4
    - SNC2
  - **Software UMA**
    - QUAD
    - HEM
    - A2A
  - **Latency [ns]**
    - Local (L1)
      - Copy/BenchIT: 3.8
      - Tile (L2): 34 (M)
        - 17 (E)
          - 14 (S,F)
            - Remote: 107-122 (M)
              - 96-114 (E)
            - 116 (E)
        - 16 (E)
          - 14 (S,F)
            - Remote: 107-117 (S,F)
              - 109-117 (S,F)
  - **Bandwidth [GB/s] (Read)**
    - Tile: 7.5 (M)
      - Remote: 7.5 (M)
  - **Bandwidth [GB/s] (Copy)**
    - Tile: 7.5 (M)
      - Remote: 7.5 (M)
  - **Contention [ns] (1:N copy)**
    - Linear, $T_C(N) = \alpha + \beta \cdot N$
      - $\alpha$: 200
      - $\beta$: 34
  - **Congestion (P2P pairs)**: None

- **Diagram:**
  - (a) All-to-all mode.
  - (b) Quadrant mode.
  - (c) SNC4 mode.

- **Text:**
  - Are you kidding me?
A principled approach to designing cache-to-cache broadcast algorithms

Multi-ary tree example

- Tree depth:
  \[ T_{tree} = \sum_{i=1}^{d} T_C(k_i) = \sum_{i=1}^{d} (c \cdot k_i + b) \]
  \[ = \sum_{i=1}^{d} (R_R + R_L + c \cdot (k_i - 1)) \]

- Tree cost:
  \[ T_{sbcost} = \min_{d,k_i} \left( T_{fw} + \sum_{i=1}^{d} (c \cdot k_i + b) + \sum_{i=1}^{d} T_{nb}(k_i + 1) \right) \]

- Reached threads:
  \[ N \leq 1 + \sum_{i=1}^{d} \prod_{j=1}^{i} k_j, \quad \forall i < j, k_i \leq k_j \]

S. Ramos, TH: “Cache line aware optimizations for ccNUMA systems (IEEE TPDS’17).
Model-driven performance engineering for broadcast

\[ T_{sbcast} = \min_{d,k_i} \left( T_{fw} + \sum_{i=1}^{d} (c \cdot k_i + b) + \sum_{i=1}^{d} T_{nb}(k_i + 1) \right) \]

\[ N \leq 1 + \sum_{i=1}^{d} \prod_{j=1}^{i} k_j, \quad \forall i < j, k_i \leq k_j \]

Binary or binomial trees?

Oh! But sure I can do that.
Model-driven performance engineering for broadcast

Binary or binomial trees?

13x faster, lower variance

Oh! But sure I can do that.

(a) Filling Tiles.

(b) Scatter.
Easy to generalize to similar algorithms

Barrier (7x faster than OpenMP)

Reduce (5x faster than OpenMP)
Sorting in complex memories

- Triad Cache Mode: SNC4
- Triad Flat Mode: SNC4

Check! Let’s do something “Big Data”!

We’ll need a parallel sort on all cores, right?
Memory model: Bitonic Mergesort

- Slices of 16 elements go through a bitonic network.
- Communication: CPU₀ accesses data from local and remote caches.
- Synchronization: CPU₀ waits for CPU₁.
- Memory accesses: latency vs. bandwidth.
Modeling Bitonic Sorting

\[ C_{L1}(n) = [\log_2(n) - 1]2n \cdot \text{cost}_{L1} + 2n \cdot \text{cost}_{mem} \]

\[ C_{L2}(n) = \frac{n}{n_{L1}} C_{L1}(n_{L1}) + \]
\[ + [\log_2(n) - \log_2(n_{L1})]2n \cdot \text{cost}_{L2} \]

\[ C_{mem}(n) = \frac{n}{n_{L2}} C_{L2}(n_{L2}) + \]
\[ + [\log_2(n) - \log_2(n_{L2})]2n \cdot \text{cost}_{mem} \]
Bitonic Sort of 1 kiB

(a) Sorting 1 KB of integers.

- Synchronization outweights memory costs for small data!
- So don’t parallelize too much!
Bitonic Sort of 4 MiB

Latency (seconds) vs. Number of threads

- Mem. model Lat.
- Mem. model BW
- Full model Lat.
- Full model BW
- Measured

“parallelization boundary”

model including synchronization cost
model (just) memory costs
Bitonic Sort of 1 GiB

Always use all cores!

synchronization negligible
The most surprising result last ...

The model (and practical measurements) indicate that it does not matter.

Thesis: the higher bandwidth of MCDRAM did not help due to the higher latency ($\log^2 n$ depth).

Disclaimer: this is NOT the best sorting algorithm for Xeon Phi KNL. It is the best we found with limited effort. We suspect that a combination of algorithms will perform best.
But what now? Is that it?

Not really, we need to enable large-scale applications!
DAPPy – Data-centric Parallel Programming for Python

- Memory access decoupled from computation

- Programs are composed of Tasklets and Memlets
  - Tasklets wrapped by simple primitives: Map, Iterate, Reduce
  - Hide communication, caching and data-movement

- Easy-to-integrate Python programming interface

- Graph-based compilation pipeline

```python
@dapp.program
def gemm(A, B, C):
    # local definitions
    @dapp.map(_[0:M, 0:K, 0:N])
    def multiplication(i, j, k):
        in_A << A[i,k]
        in_B << B[k,j]
        out >> tmp[i,j,k]
        out = in_A * in_B

    @dapp.reduce(tmp, C, axis=2)
    def sum(a,b):
        return a+b
```
DAPPy Compilation Infrastructure

**Code**

@dapp.program
def program(A, B):
    @dapp.map([0:N,0:M])
def transpose(i, j):
        a << A[i,j]
b >> B[j,i]
    ...

dappy Program

**Specialization**

DAPP Framework

Subgraph Matching

Performance Models

- Custom Patterns
- Custom Models

Stateful Dataflow Graph (SDFG)

**Runtime**

- CPU Library
- GPU Library
- FPGA Modules
- System Probe
- Microbenchmarks

**Partitioning, Scheduling**

**Domain Programmer**

**Performance Engineer**

**Architecture Expert**
Performance

Naive
Performance

SDFG

MapReduceFusion
Performance

![Diagram of SDFG with operations and performance graph]

The diagram represents the SDFG (Structured Data Flow Graph) with operations such as Map, Reduce, and UnmapReduceSum. The performance graph shows the comparison between LoopReorder, MapReduceFusion, and Naive methods across different problem sizes.

The x-axis represents the problem size, and the y-axis represents performance in GFlop/s (GigaFlops per second). The graph illustrates the performance improvement achieved by LoopReorder and MapReduceFusion compared to the Naive method.
Performance

SDFG

BlockTiling
LoopReorder
MapReduceFusion
Performance

SDFG

RegisterTiling
BlockTiling
LoopReorder
MapReduceFusion

Naive
Performance

- LocalStorage
- RegisterTiling
- BlockTiling
- LoopReorder
- MapReduceFusion

Naive

Performance [GFlop/s]

Problem Size
Performance

![Graph showing performance comparison between Intel MKL, DAPP, and OpenBLAS across different problem sizes. The graph indicates a 25% difference between Intel MKL and DAPP.]
void _program_gemm(int sym_0, int sym_1, int sym_2, double * __restrict__ A, double * __restrict__ B, double * __restrict__ C) {
    // State s0
    for (int tile_k = 0; tile_k < sym_2; tile_k += 128) {
        #pragma omp parallel for
        for (int tile_i = 0; tile_i < sym_0; tile_i += 64) {
            for (int tile_j = 0; tile_j < sym_1; tile_j += 240) {
                for (int regtile_j = 0; regtile_j < (min(240, sym_1 - tile_j)); regtile_j += 12) {
                    vec<double, 4> local_B_s0_0[128 * 3];
                    Global2Stack_2D_FixedWidth<double, 4, 3>(&B[tile_k*sym_1 + (regtile_j + tile_j)], sym_1,
                        local_B_s0_0, min(sym_2 - tile_k, 128));

                    for (int regtile_i = 0; regtile_i < (min(64, sym_0 - tile_i)); regtile_i += 4) {
                        vec<double, 4> regtile_C_s0_1[4 * 3];
                        for (int i = 0; i < 4; i += 1) {
                            for (int j = 0; j < 3; j += 1) {
                                double in_A = A[(i + regtile_i + tile_i)*sym_2 + tile_k];
                                vec<double, 4> in_B = local_B_s0_0[0*3 + j];
                                // Tasklet code (mult)
                                auto out = (in_A * in_B);
                                regtile_C_s0_1[i*3 + j] = out;
                            }
                        }
                    }
                }
            }
        }
    }
    // ...
}
Scientific performance engineering for complex memory systems

Questions/Discussions?