EECS ELECTRICAL ENGINEERING AND COMPUTER SCIENCE
UNIVERSITY OF MICHIGAN

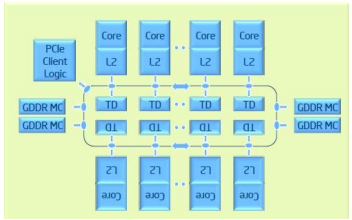
ECE 583 Paper Presentation

# COMP: Compiler Optimizations for Manycore Processors

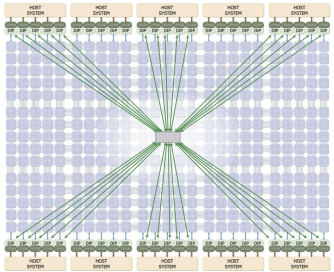
Group 25: Ruijie Gao, Barry Lyu

# Manycore Processors

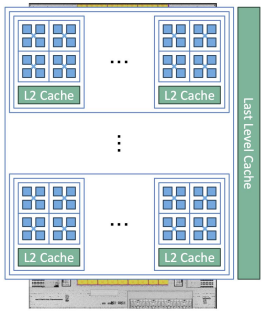
Modern manycore processors follow the MIMD (Multiple Instruction Multiple Data) architecture, connecting large amounts of independent cores through interconnects to offer massive thread-level parallelism.




Intel Xeon Phi
(62 Cores)



Tesla Dojo
(354 Cores)




Pezy SC4
(2048 Cores)

They excel at irregular workloads that require fine-grained synchronization.

- Graph Algorithms
- Sparse Linear Algebra
- Dynamic Programming
- High Performance Computing

# The Manycore Challenge

Achieving the peak performance is notoriously difficult.

**Data Transfer Overhead**

Offloading computation incurs huge overhead to transfer data from host to coprocessor.

**Irregular Memory Accesses**

Irregular memory accesses hurt cache locality, prevents vectorization, and stresses the on-chip interconnect.

**Pointer-Based Data Structures**

Pointer based data structures has small transfer granularity and incurs large number of page faults.

Relies on programmers to tune offloaded codes, requires high level of expertise, often involves:

- Rewriting loops in pipelined style
- Leveraging asynchronous communication
- Understanding and optimizing memory access patterns
- Choosing proper synchronization granularity.

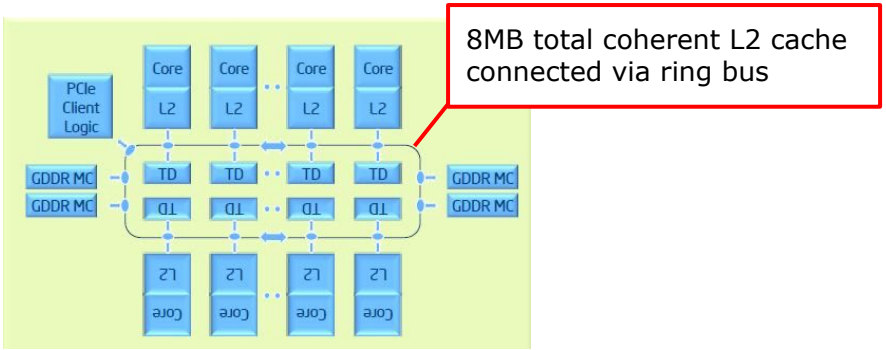
# Intel MIC Architecture

**Intel Many-Integrated Core (MIC) Architecture**

Thread-level Parallelism (TLP):
62 Cores, 1 reserved for OS

Data-level Parallelism (DLP):
512-bit SIMD operation per core

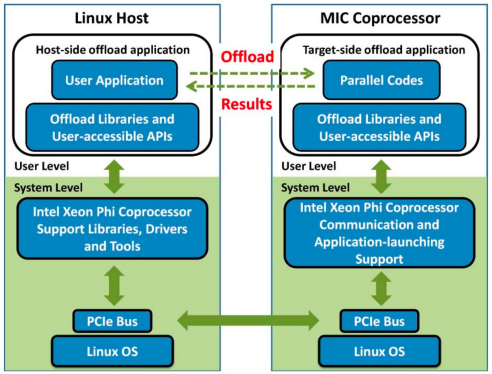
Simultaneous Multithreading (SMT):
4 simultaneous threads per core




**Offload Mode Execution**

Xeon-Phi cores are significantly slower than CPU in serial execution.

Serial code executed on modern CPU, parallel regions offloaded to MIC.



# Programming Model for Offloading Compute

**Implicit Data Transfer**

Shared data between CPU and manycore

```
int * _Cilk_shared v;
_Cilk_shared void foo() {
    for(int i = 0; i < 5; i++) {
        v[i] = i;
    }
}
int main() {
    int size = sizeof(int)*5;
    v = (int * _Cilk_shared)_Offload_shared_malloc(size);
   _Cilk_offload foo();
   return 0;
}
```

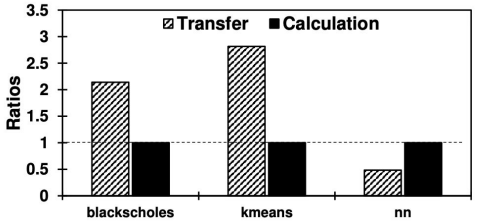
**Explicit Offload**

Specify input and output data to be transferred

```
pra  ma offload target(mic:0) \
 in(sptprice, ...: length(numOptions))\
 out(prices: len  th(numOptions))
 pra  ma omp parallel for private(i, price)
 for (i=0; i<numOptions; i++) {
    price = BlkSchlsEqEuroNoDiv(sptprice[i], .. );
    prices[i] = price;
  }
//offload: specify offload re  ion;
//target: specify which coprocessor to be used;
//in: input data for offload re  ion;
//out: results from offload re  ion.
```

# Optimization 1: Data Streaming

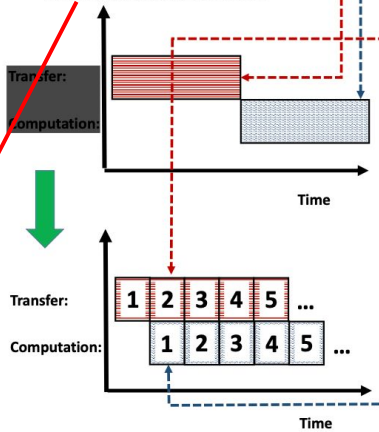
Data transfer is the bottleneck for a number of applications




1. Recv all input operands
2. Perform Computation
3. Send all output operands

```
pra ma offload target(mic:0) \
 in(sptprice, ...: length(numOptions))\
 out(prices: len th(numOptions))
pra ma omp parallel for private(i, price)
 for (i=0; i<numOptions; i++) {
   price = BlkSchlsEqEuroNoDiv(sptprice[i], ... );
   prices[i] = price;
 }
//offload: specify offload re ion;
//target: specify which coprocessor to be used;
//in: input data for offload re ion;
//out: results from offload re ion.
```

*(a) a LEO code example from benchmark blackscholes*




*(d) data streaming overview*

```
define ALLOC   alloc_if(1) free_if(0)
define REUSE   alloc_if(0) free_if(0)
define FREE    alloc_if(0) free_if(1)
...
#pragma offload target(mic:0) \
 nocopy(sptprice[0:numOptions]: ALLOC )\
 in(bsize, price)
 ... //allocate memory on the coprocessor
pra ma offload_transfer tar et(mic:0) \
 in(sptprice[0:bsize]: REUSE) \
 ...
 signal(sptprice)
 //asynchronous data transfer for 1st block
for(k=0; k<numOptions/bsize; k++) {
  if (k<numOptions/bsize-1) {
    start = (k+1) * bsize;
    pra ma offload_transfer tar et(mic:0) \
    in(sptprice[start:bsize]: REUSE)\
    ...
    signal(start + sptprice)
  } //asynchronous transfer (k+1)-th block
  start = k * bsize;
  #pragma offload target(mic:0) \
   wait(start+sptprice) \
   out(prices[start:bsize]: REUSE)
  pra ma omp parallel for private(i, price)
   for (i=start; i<start+bsize; i++) {
     price=BlkSchlsEqEuroNoDiv(...);
     prices[i] = price;
   }
   //perform the k-th computation block
}
#pragma offload target(mic:0) \
 nocopy(sptprice[0:numOptions]: FREE) \
 ... //free memory on the coprocessor
```

*(b) blackscholes after data streaming*

**Pipelined Data Transfer**

- Transfer 1st block async
- Transfer k+1-th block async
- Wait till k-th block is available
- Perform k-th block compute
- Send result back to processor

# Optimization 1: Data Streaming

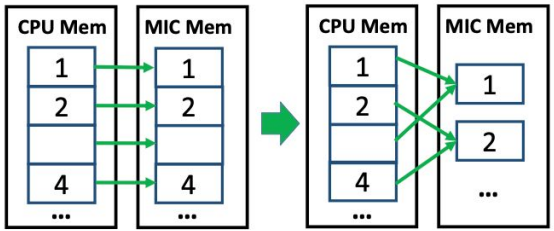
**Legality Check**

All array indices in the loop must be in form a*i+b for a loop to be transformable.

**Memory Allocation**

Preallocate memory to prevent excessive malloc calls.

Only need to allocate two blocks for double buffering, not the full input size.




**Offload Merging**

For large loops with multiple parallel inner loops, hoist loads and merge blocks.

**Deciding the Proper Block Size**

Find balance between launch overhead and initial transfer time.

D := total data transfer time
C := total compute time
K := Kernel launch overhead
N := Number of blocks

Choose N to minimize:

$$D/N + max\{C/N + K, D/N\} * (N - 1) + C/N + K$$

Best N is often between 10 and 40.

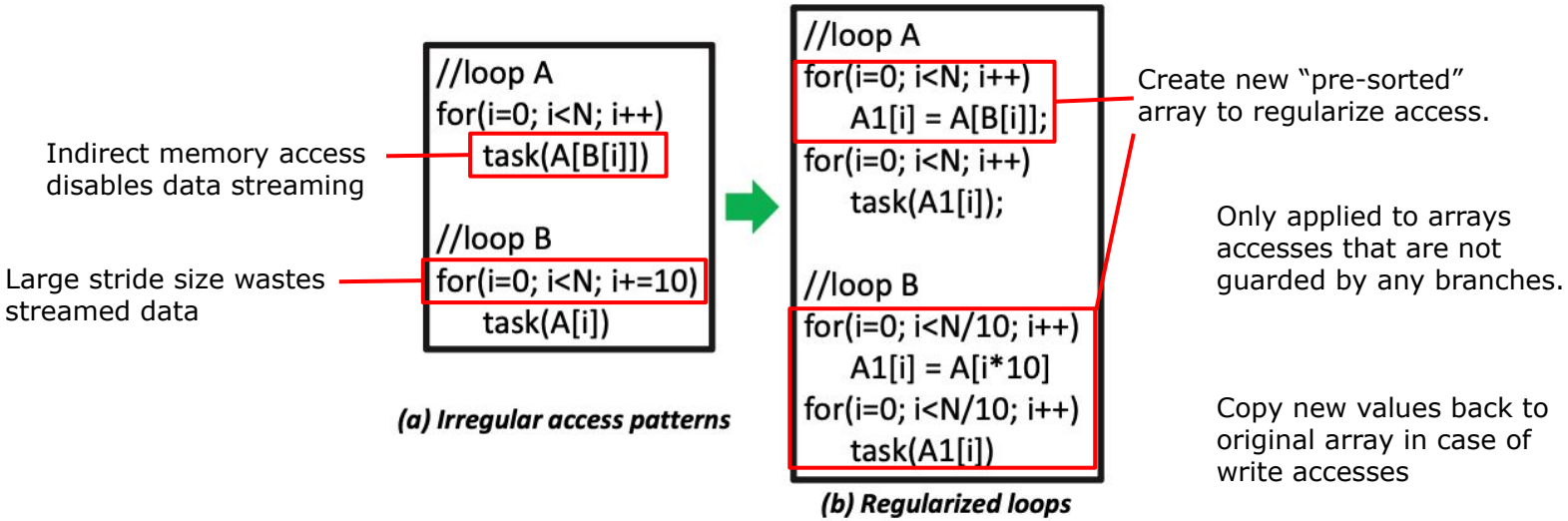
**Loop Transformation**

1. Perform legality check
2. Insert memory allocation code
3. Replace original loop with 2D loop, with the inner loop performing a compute block and the outer loop repeating over all blocks.
4. Merge offloads in separate inner loops
5. Insert data transfer and synchronization primitives in the outer loop body.
6. Duplicate outer loop for even and odd blocks, using double buffered inputs.

# Optimization 2: Regularization

**Irregular Accesses ::=**

Accesses that do not access elements continuously across iterations.



# Optimization 2: Regularization

**Loop Splitting**

Must be a parallel loop (no cross-iteration dependence), so irregular accesses can be safely split from the rest of the loop body.

**Array of Structures**

Regularize array of structures via static conversion to structure of arrays.

**Pipelining Regularization with Data Transfer**

Save overhead by pipelining and double buffering regularizations.

Regularize k+1-th block while k-th block is computing

**Loop Transformation**

1. Perform legality check
2. Identify irregular accesses
3. Insert regularization loop
4. Redirect irregular access to to regularized array
5. Insert write back to original array if necessary
6. Insert async signaling for pipelined streaming regularization.

# Optimization 3: Shared Memory

## Why standard MYO fails?

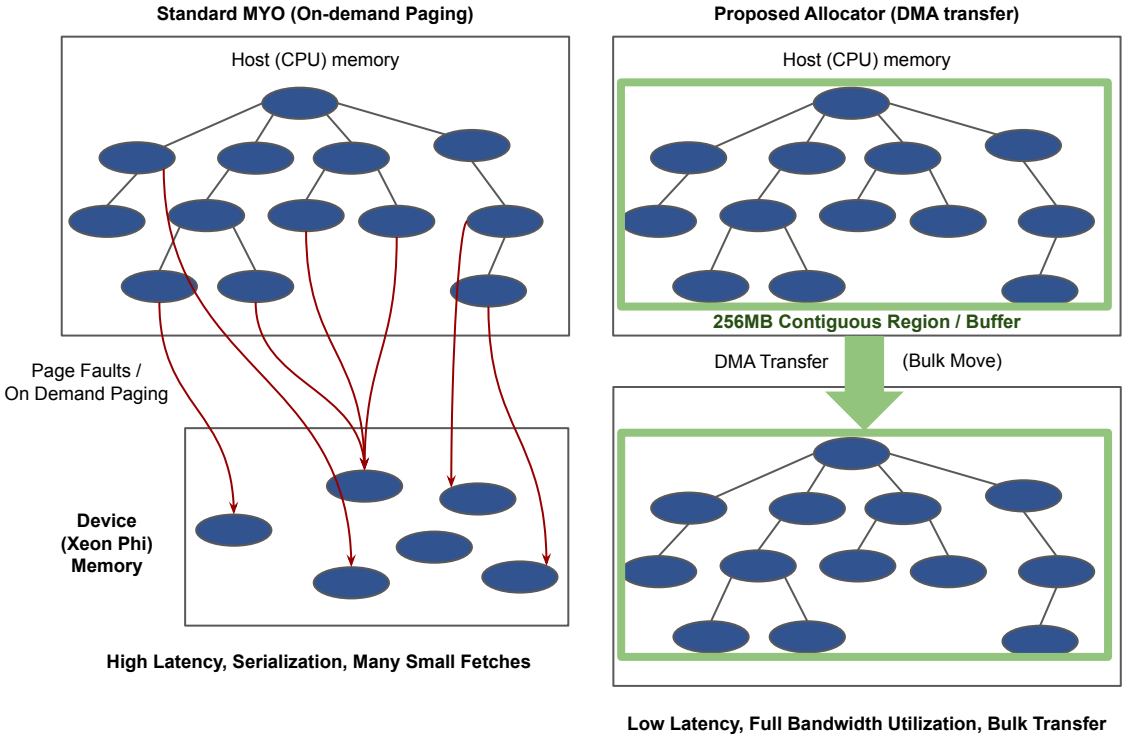
Intel's MYO (Mine Yours Ours) relies on on-demand paging, and is fundamentally mismatched to PCIe-attached manycore systems.

For pointer-heavy structures (like trees/graphs), this triggers a storm of page faults, serializing execution.

## Proposed Allocator

Instead of page-based transfer, we use **Region-based Allocation**.

- **Strategy**: Pre-allocate large buffers (e.g., 256MB chunks).
- **Dynamic Growth**: When a buffer fills, a new one is allocated
- **Transfer**: The entire buffer is copied via DMA, fully saturating PCIe bandwidth



# Shared Memory: Pointer Translation

**Host pointers (CPU Addresses) are invalid on the Device (MIC), and buffers are not contiguous.**

### The "BID" Mechanism

**Constraint**: For consistency, shared pointers always store CPU addresses, even on the MIC

**Challenge**: Because we allocate multiple non-contiguous regions (buffers), simply adding one offset doesn't work. Identifying which buffer a pointer belongs to requires `O(n)` range checks.

### Solution - Augmented Pointers

- We add a 1-bit **B**uffer **ID** to every shared pointer and object
- `ptr.bid` stores the ID of the buffer the pointer points to
- This allows O(1) identification of the target buffer

### Fast Translation (Delta Table)

**The Delta Table**: A lightweight lookup table created during transfer. Its size equals the number of buffers.

It stores the memory offset for each buffer:

```
delta[i] = BaseAddr_MIC[i] - BaseAddr_CPU[i]
```

**Dereferencing on MIC**:

When the code executes `*p` on the MIC, the compiler generates:

```
Real_Addr = p.value + delta[p.bid]
```

This simple addition is extremely fast compared to software address translation tables.

# Performance Evaluation

**Data Streaming**

Moderate Speedup, significant memory savings on 5 out of 12 benchmarks

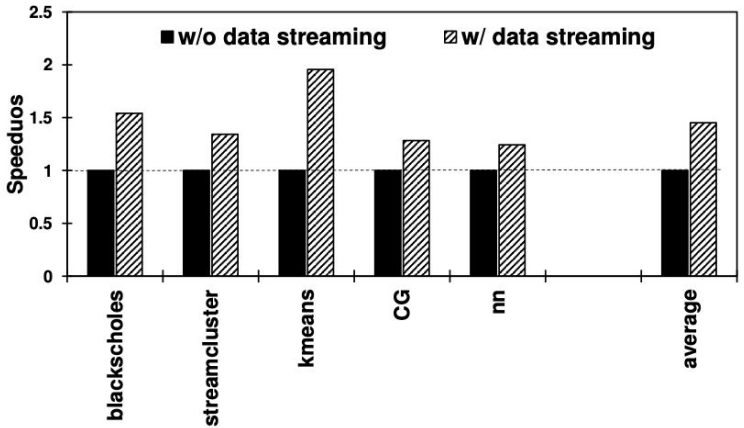



Figure 12: Performance gains by *data streaming*

~ 80% Avg Memory Saving

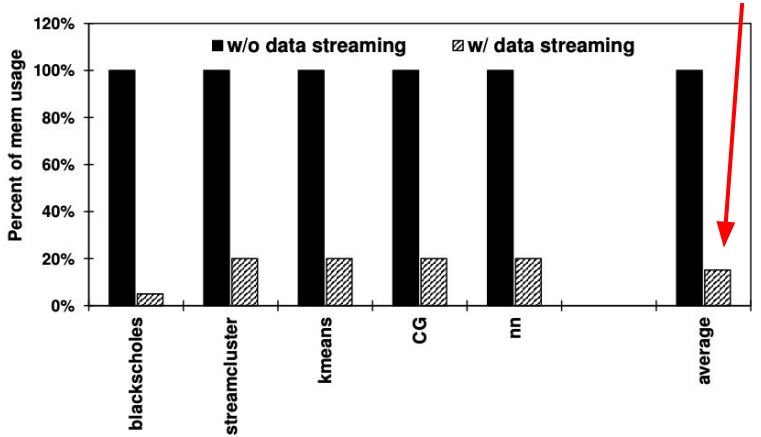



Figure 13: Memory usage after applying *data streaming*

# Performance Evaluation

**Offload Merging**

Significant Speedup

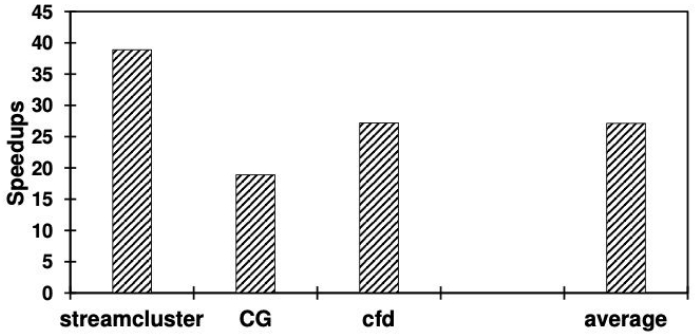



Figure 14: Performance gains by *offload merging*

**Regularization**

Speedup on 2 out of 12 benchmarks

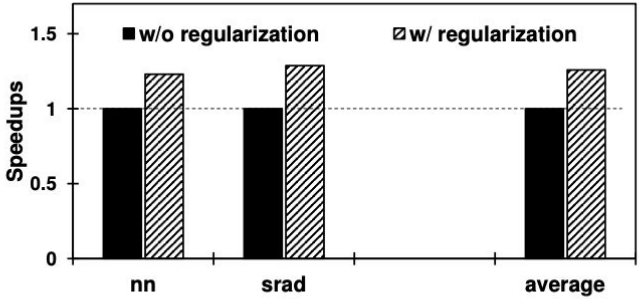
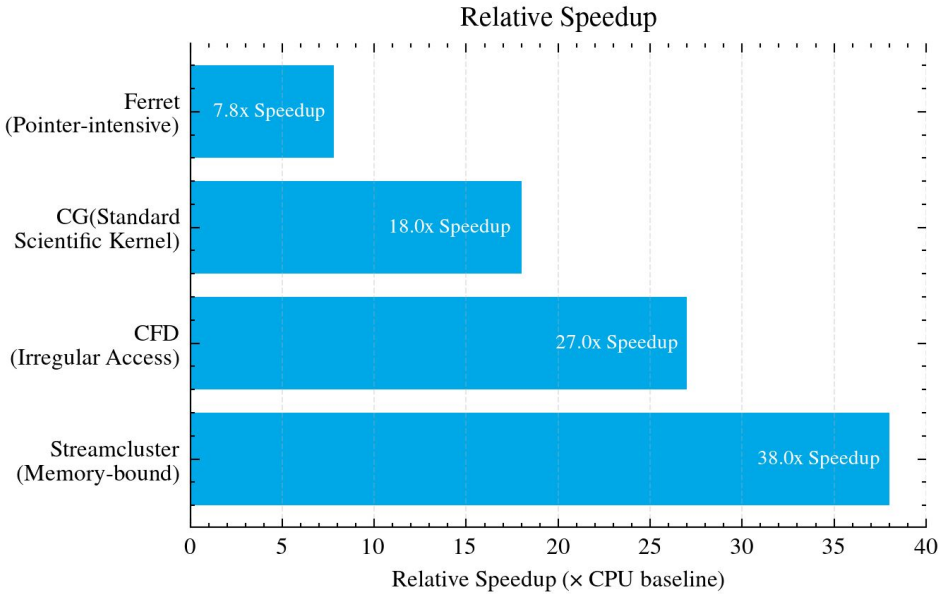



Figure 15: Performance gains by using *regularization*

# Performance Evaluation




**Performance of Representative Benchmarks**

## Key Takeaway:

- **Overall**: Optimizations benefit 9 out of 12 benchmarks
- **Data Streaming**: Crucial for memory-bound apps. Reduced memory usage by 80%.
- **Regularization**: Enable vectorization for `nn` and `srad`
- **Shared Memory**: Made `ferret` (pointer heavy) runnable, achieving 7.8x spread up over MYO

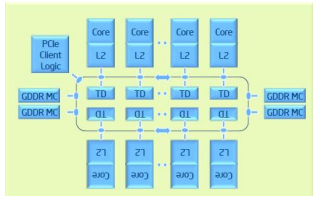
# Comments

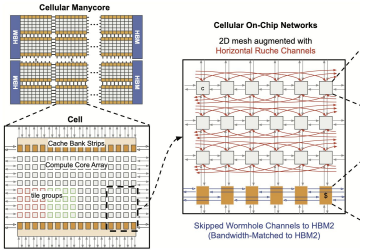
## Pros

- Concepts in this work are translatable to many offload-based compute that suffer from data transfer overheads, including GPUs and accelerators.

## Cons

- The work is a little old and targets the Intel Xeon Phi, which is not representative of the latest manycore architectures.
  - Mismatch in network topology
  - No complex hierarchy
- The work doesn't discuss in detail the synchronization and communication overheads within manycore execution.



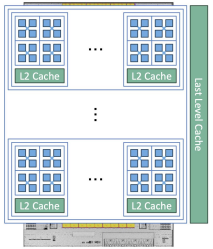

Intel Xeon Phi → HammerBlade Manycore [1]

62-Core, ring network, SIMD/core
2-level Memory Hierarchy

2048-Core, mesh-like network

PEZY SC4

2048-Core, mesh network, 4+ Level
Memory/Compute Hierarchy




[1] Jung et al., ISCA 24'