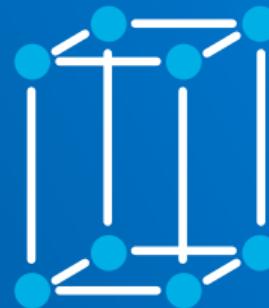


# Fundamentals of Parallelism on Intel® Architecture

Week 4  
Memory Traffic



# §1. Cheap FLOPs



# Loop Was Vectorized, Now What?

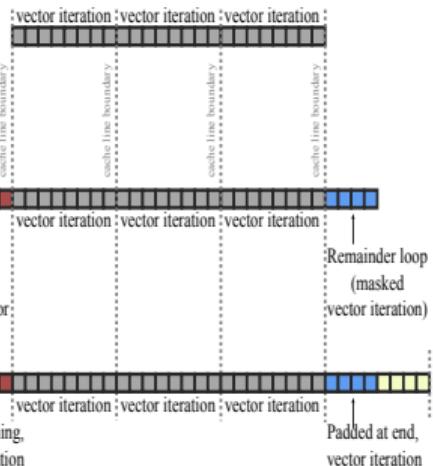
1. Unit-stride access
2. Data alignment
3. Container padding
4. Eliminate peel loops
5. Eliminate multiversioning
6. **Optimize data re-use in caches**

```
for (i = 0; i < n; i++) A[i] = ...
```

Code Path 1:  
data aligned from iteration 0,  
n is multiple of vector length

Code Path 2:  
data aligned from iteration 3,  
n is not a multiple of vector length

Optimization:  
padded loop count, aligned data  
to regularize vectorization pattern



# Loop Was Vectorized, Now What?

1. Unit-stride access
2. Data alignment
3. Container padding
4. Eliminate peel loops
5. Eliminate multiversioning
6. **Optimize data re-use in caches**

Vector Arithmetics is Cheap,  
Memory Access is Expensive

If you don't optimize cache  
usage, vectorization will not  
matter.

You will be bottlenecked by  
memory access.



# How Cheap are FLOPs?

Intel Xeon Phi processor 7250

$68 \text{ cores} \times 1.2 \text{ GHz} \times 8 \text{ vec.lanes} \times 2 \text{ FMA} \times 2 \text{ IPC} \approx 2.6 \text{ TFLOP/s}$

$2.6 \text{ TFLOP/s} \times 8 \text{ bytes} \approx 21 \text{ TB/s}$

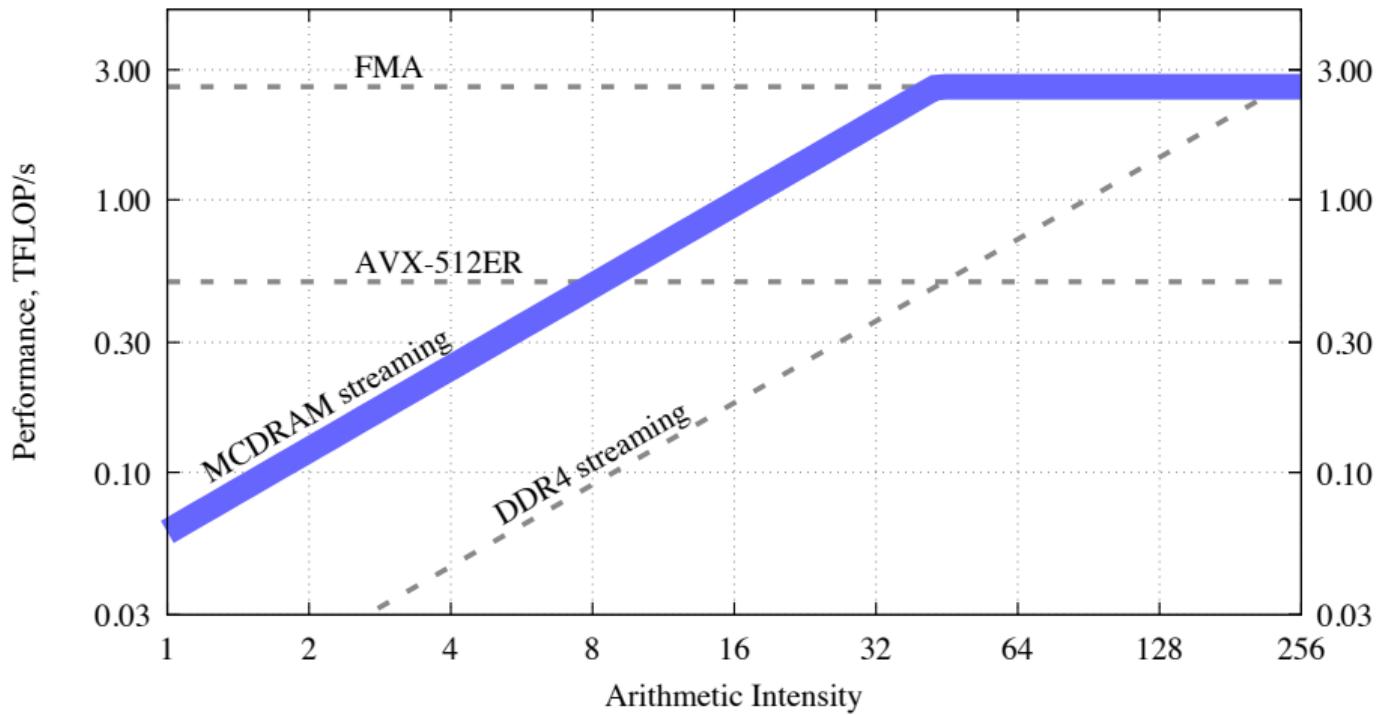
MCDRAM bandwidth  $\approx 0.48 \text{ TB/s}$

Ratio =  $21/0.48 \approx 43$  (FLOPs)/(Memory Access)

- ▷ > 50 FLOPs/Memory Access — Compute-bound Application
- ▷ < 50 FLOPs/Memory Access — Bandwidth-bound Application



# Arithmetic Intensity and Roofline Model



More on roofline model: Williams et al.

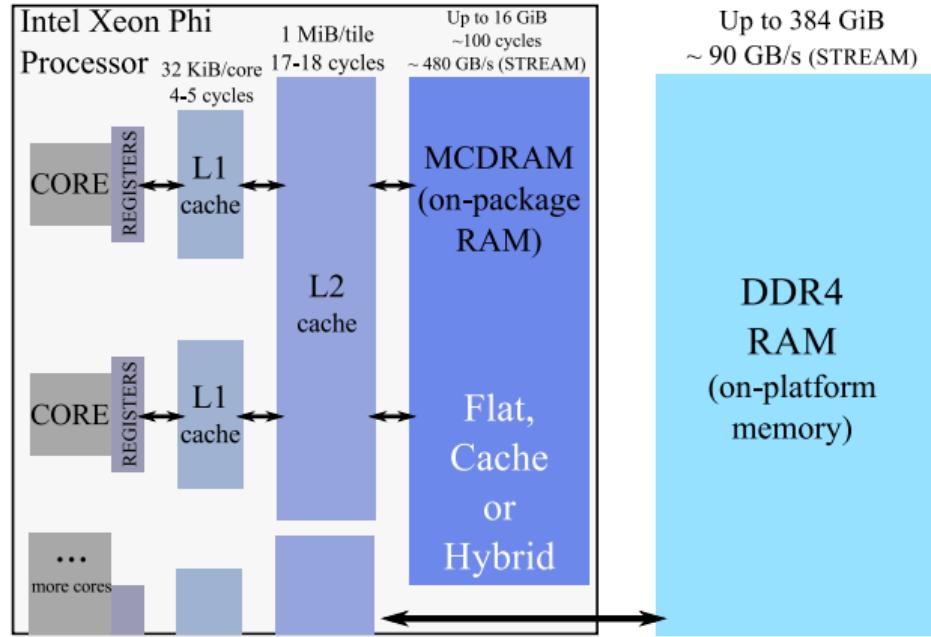


## §2. Memory Hierarchy



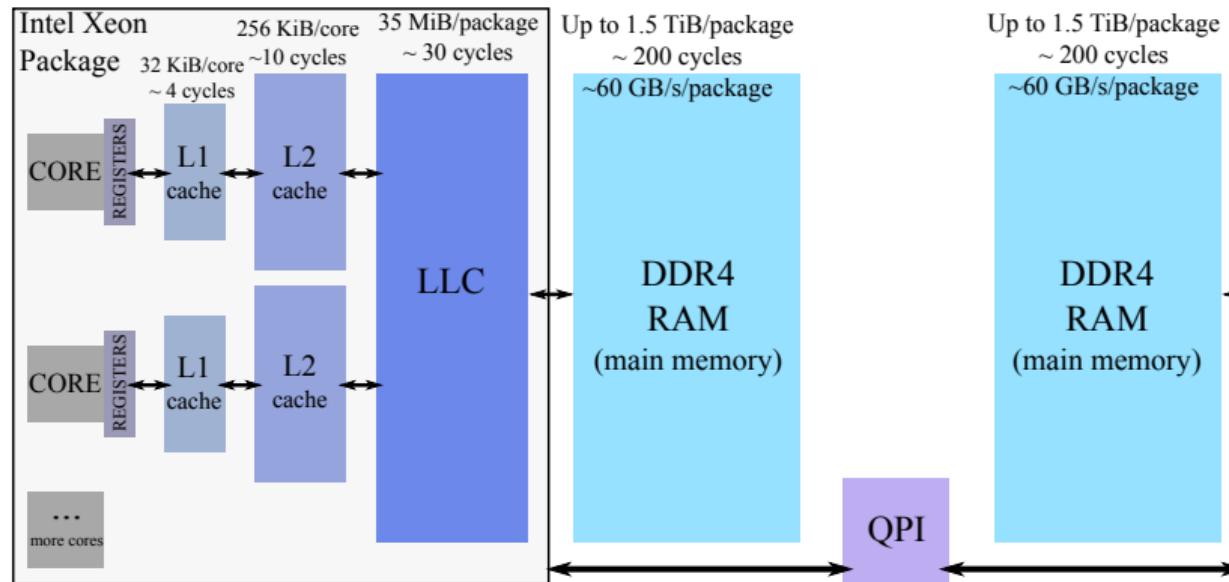
# KNL Memory Organization (bootable)

- ▶ On-package high-bandwidth memory (HBM) – MCDRAM
- ▶ Optimized for arithmetic performance and bandwidth (not latency)



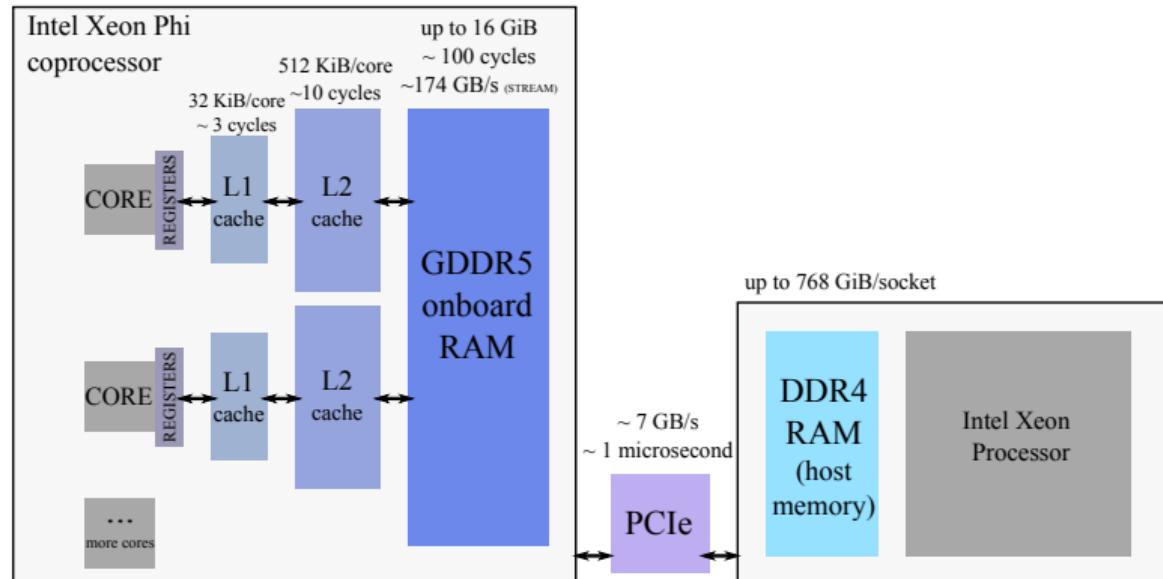
# Intel Xeon CPU: Memory Organization

- ▷ Hierarchical cache structure
- ▷ Two-way processors have NUMA architecture



# KNC Memory Organization

- ▶ Direct access to  $\leq 16$  GiB of cached GDDR5 memory on board
- ▶ No access to system DDR4, connected to host via PCIe



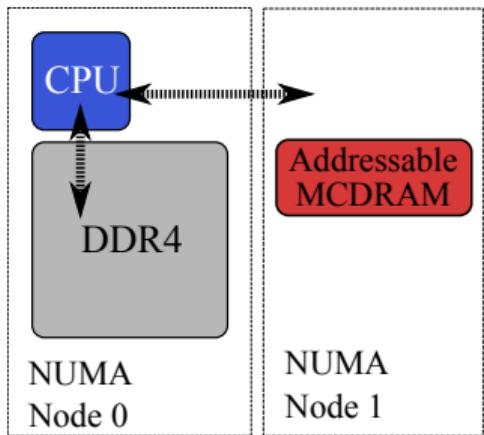
# §3. High-Bandwidth Memory



# High-Bandwidth Memory Modes

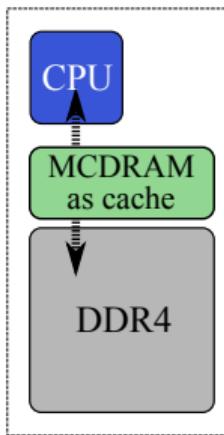
## Flat Mode

- ▷ MCDRAM treated as a NUMA node
- ▷ Users control what goes to MCDRAM



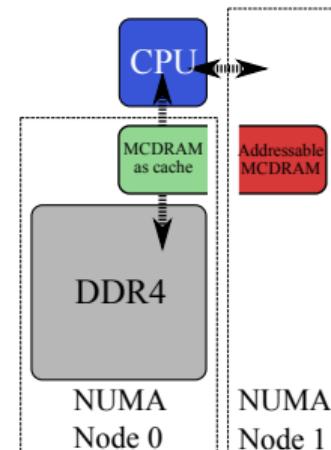
## Cache Mode

- ▷ MCDRAM treated as a Last Level Cache (LLC)
- ▷ MCDRAM is used automatically

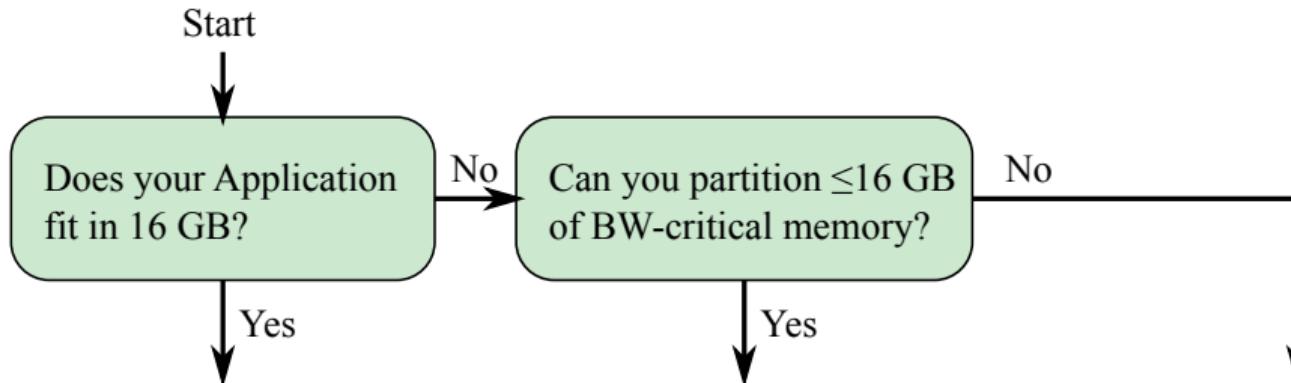


## Hybrid Mode

- ▷ Combination of Flat and Cache
- ▷ Ratio can be chosen in the BIOS



# Flow Chart for Bandwidth-Bound Applications



numactl	Memkind	Cache mode
<ul style="list-style-type: none"><li>▷ Simply run the whole program in MCDRAM</li><li>▷ No code modification required</li></ul>	<ul style="list-style-type: none"><li>▷ Manually allocate BW-critical memory to MCDRAM</li><li>▷ Memkind calls need to be added.</li></ul>	<ul style="list-style-type: none"><li>▷ Allow the chip to figure out how to use MCDRAM</li><li>▷ No code modification required</li></ul>



# Running Applications in HBM with numactl

- ▶ Finding information about the NUMA nodes in the system.

```
user@kn1% # In Flat mode of MCDRAM
user@kn1% numactl -H
available: 2 nodes (0-1)
node 0 cpus: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 ... 254 255
node 0 size: 98207 MB
node 1 cpus:
node 1 size: 16384 MB
```

- ▶ Binding the application to HBM (Flat/Hybrid)

```
user@kn1% icc myapp.c -o runme -xMIC_AVX512
user@kn1% numactl --membind 1 ./runme
// ... Application running in HBM ... //
```



# §4. Memory Allocation



# Allocation in HBM with Memkind Library

```
1 #include <hbwmalloc.h>
2 const int n = 1<<10;
3 // Allocation to MCDRAM
4 double* A = (double*) hbw_malloc(sizeof(double)*n);
5 // No replacement for _mm_malloc. Use posix_memalign
6 double* B;
7 int ret = hbw_posix_memalign((void**) &B, 64, sizeof(double)*n);
8 .....
9 // Free with hbw_free
10 hbw_free(A); hbw_free(B);
```



# Compilation with Memkind Library and hbwmalloc

To compile C/C++ applications:

```
user@knl% icpc -lmemkind foo.cc -o runme  
user@knl% g++ -lmemkind foo.cc -o runme
```

Open source distribution of Memkind library can be found at:  
[memkind.github.io/memkind](https://memkind.github.io/memkind)

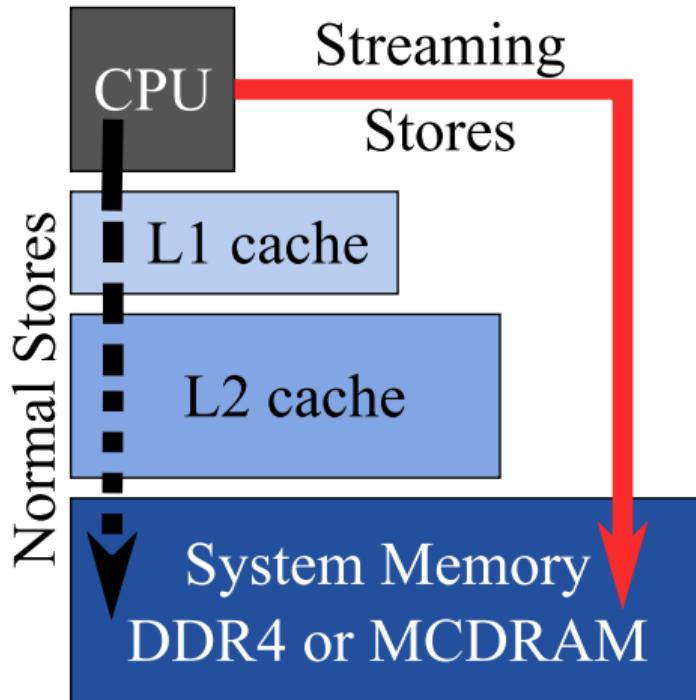
Learn more:  
[colfaxresearch.com/knl-mcdram](http://colfaxresearch.com/knl-mcdram)



# §5. Bypassing Caches



# Streaming Stores



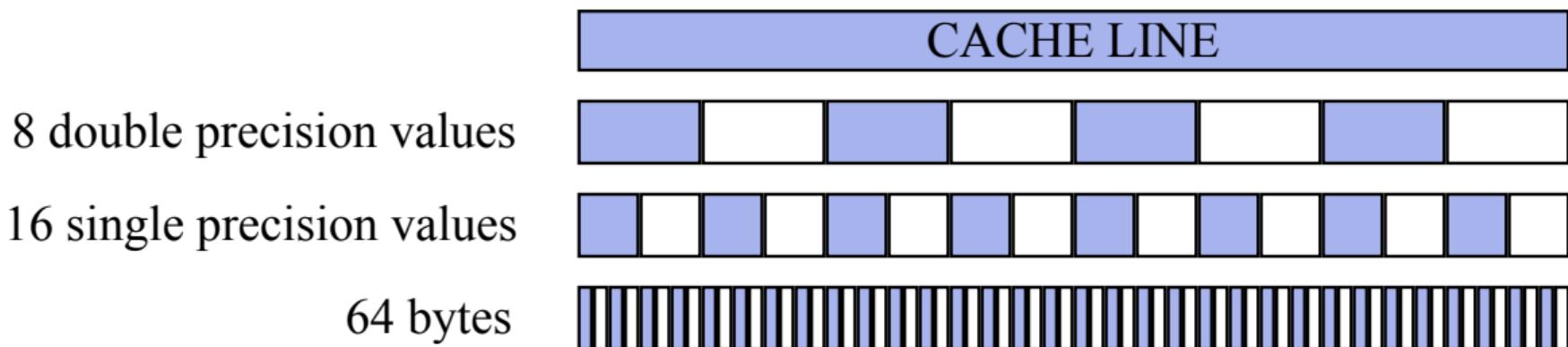
- ▷ Bypass cache, write to RAM
- ▷ Save cache for other data
- ▷ `#pragma vector nontemporal`
- ▷ `-qopt-streaming-stores=always`

# §6. Locality in Space



# Cache Lines

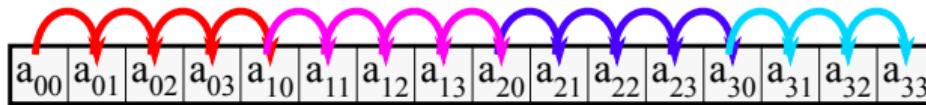
- ▷ Minimal block of data transferred between memory and cache
- ▷ 64 bytes long in Intel Architecture
- ▷ Aligned on 64-byte boundaries in memory



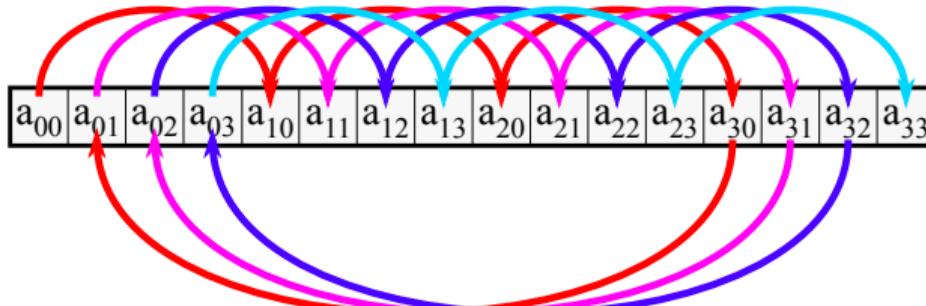
# Principle

Choose loop order to maintain unit-stride memory access

a <sub>00</sub>	a <sub>01</sub>	a <sub>02</sub>	a <sub>03</sub>
a <sub>10</sub>	a <sub>11</sub>	a <sub>12</sub>	a <sub>13</sub>
a <sub>20</sub>	a <sub>21</sub>	a <sub>22</sub>	a <sub>23</sub>
a <sub>30</sub>	a <sub>31</sub>	a <sub>32</sub>	a <sub>33</sub>



a <sub>00</sub>	a <sub>01</sub>	a <sub>02</sub>	a <sub>03</sub>
a <sub>10</sub>	a <sub>11</sub>	a <sub>12</sub>	a <sub>13</sub>
a <sub>20</sub>	a <sub>21</sub>	a <sub>22</sub>	a <sub>23</sub>
a <sub>30</sub>	a <sub>31</sub>	a <sub>32</sub>	a <sub>33</sub>



Compiler may or may not be able to automate loop permutation.

# Example: Over-simplified Matrix-Matrix Multiplication

$$C = AB \quad \Leftrightarrow \quad C_{ij} = \sum_{k=0}^{n-1} A_{ik}B_{kj}$$

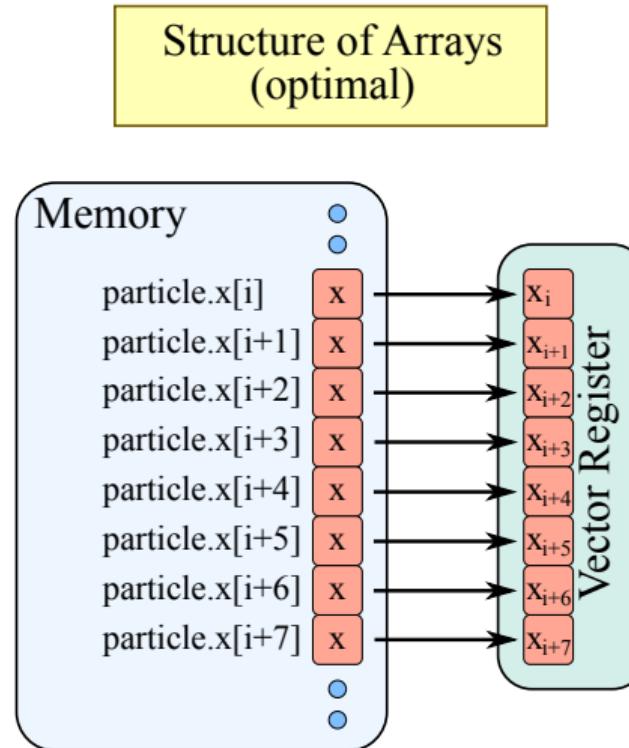
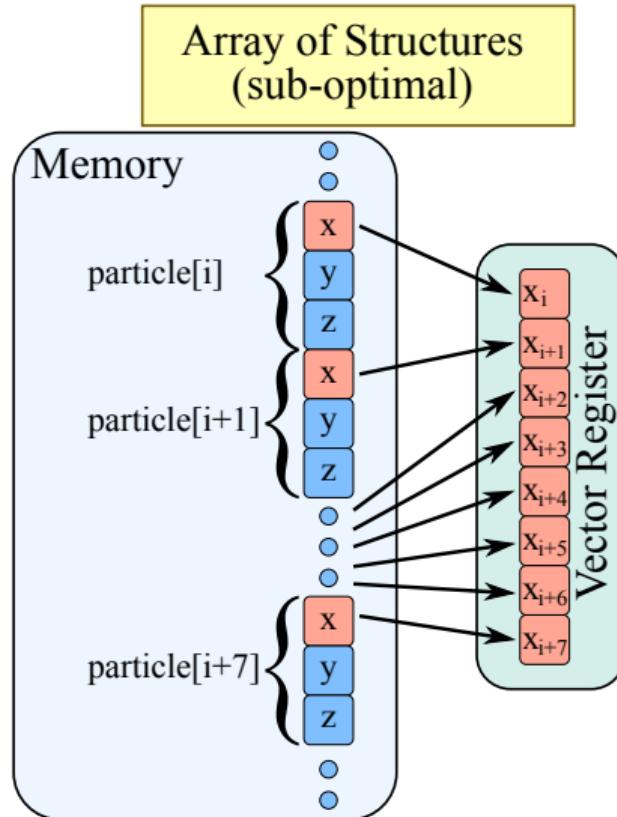
Before:

```
1 #pragma omp parallel for
2   for (int i = 0; i < n; i++)
3     for (int j = 0; j < n; j++)
4 #pragma vector aligned
5   for (int k = 0; k < n; k++)
6     C[i*n+j] += A[i*n+k]*B[k*n+j];
```

After:

```
1 #pragma omp parallel for
2   for (int i = 0; i < n; i++)
3     for (int k = 0; k < n; k++)
4 #pragma vector aligned
5   for (int j = 0; j < n; j++)
6     C[i*n+j] += A[i*n+k]*B[k*n+j];
```

# Why AoS to SoA Conversion Helps: Unit Stride



# Principle

- ▷ For best spatial locality, order loops to get unit-stride
- ▷ At -O2 and above, the compiler may interchange loops
- ▷ In complex cases, investigate loop interchange manually
- ▷ May need to re-design data containers to get unit stride



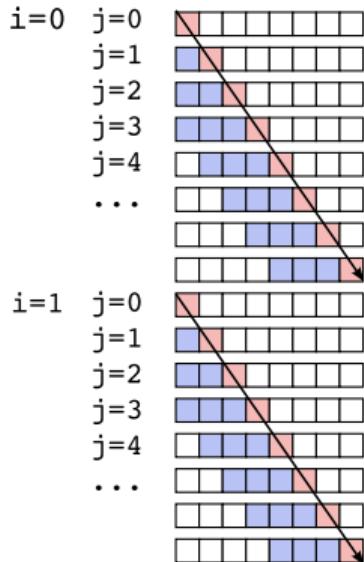
# §7. Locality in Time



# Loop Tiling: Cache Blocking

## Original:

```
for (i=0; i<m; i++)
    for (j=0; j<n; j++)
        ...=....*b[j];
```



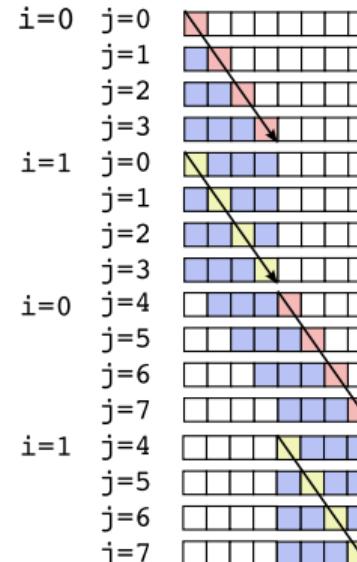
■ - cached, LRU eviction policy  
■ - cache miss (read from memory, slow)  
■ - cache hit (read from cache, fast)

Cache size: 4  
TILE=4  
(must be tuned to cache size)

Cache hit rate without tiling: 0%  
Cache hit rate with tiling: 50%

## Tiled:

```
for (jj=0; jj<n; jj+=TILE)
    for (i=0; i<m; i++)
        for (j=jj; j<jj+TILE; j++)
            ...=....*b[j];
```



# Loop Tiling (Cache Blocking) -- Procedure

```
1  for (int i = 0; i < m; i++) // Original code:  
2      for (int j = 0; j < n; j++)  
3          compute(a[i], b[j]); // Memory access is unit-stride in j
```

```
1  // Step 1: strip-mine inner loop  
2  for (int i = 0; i < m; i++)  
3      for (int jj = 0; jj < n; jj += TILE)  
4          for (int j = jj; j < jj + TILE; j++)  
5              compute(a[i], b[j]); // Same order of operation as original
```

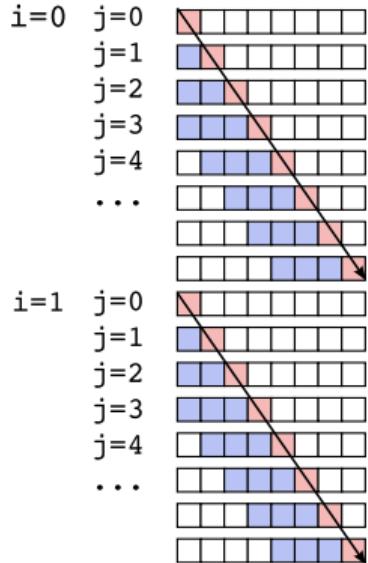
```
1  // Step 2: permute  
2  for (int jj = 0; jj < n; jj += TILE)  
3      for (int i = 0; i < m; i++)  
4          for (int j = jj; j < jj + TILE; j++)  
5              compute(a[i], b[j]); // Re-use to j=jj sooner
```



# Loop Tiling: Register Blocking

## Original:

```
for (i=0; i<m; i++)
    for (j=0; j<n; j++)
        ...=....*b[j];
```



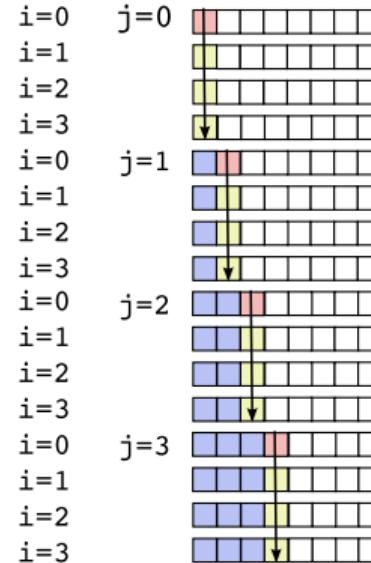
■ - cached, LRU eviction policy  
■ - cache miss (read from memory, slow)  
■ - cache hit (read from cache, fast)

Cache size: 4  
TILE=4  
(must be tuned to cache size)

Cache hit rate without tiling: 0%  
Cache hit rate with tiling: 50%

## Tiled:

```
for (ii=0; ii<m; ii+=TILE)
    for (j=0; j<n; j++)
        for (i=ii; i<ii+TILE; i++)
            ....=....*b[j];
```



# Loop Tiling (Unroll-and-Jam/Register Blocking)

```
1  for (int i = 0; i < m; i++) // Original code:  
2      for (int j = 0; j < n; j++)  
3          compute(a[i], b[j]); // Memory access is unit-stride in j
```

```
1  // Step 1: strip-mine outer loop  
2  for (int ii = 0; ii < m; ii += TILE)  
3      for (int i = ii; i < ii + TILE; i++)  
4          for (int j = 0; j < n; j++)  
5              compute(a[i], b[j]); // Same order of operation as original
```

```
1  // Step 2: permute and vectorize outer loop  
2  for (int ii = 0; ii < m; ii += TILE)  
3 #pragma simd  
4      for (int j = 0; j < n; j++)  
5          for (int i = ii; i < ii + TILE; i++)
```

# Loop Fusion Technique

Re-use data in cache by fusing loops in a data processing pipeline

```
1 MyData* data = new MyData(n);  
2  
3 for (int i = 0; i < n; i++)  
4     Initialize(data[i]);  
5  
6 for (int i = 0; i < n; i++)  
7     Stage1(data[i]);  
8  
9 for (int i = 0; i < n; i++)  
10    Stage2(data[i]);
```

```
1 MyData* data = new MyData(n);  
2  
3 for (int i = 0; i < n; i++) {  
4  
5     Initialize(data[i]);  
6  
7     Stage1(data[i]);  
8  
9     Stage2(data[i]);  
10 }
```

Potential positive side-effect: less data to carry between stages, reduced memory footprint, improved performance.



# §8. Example: Stencil Code



# Stencil Operators

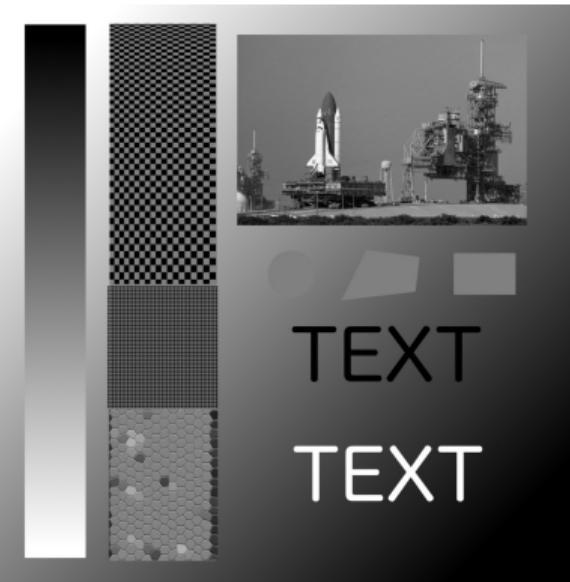
- ▷ Linear systems of equations
- ▷ Partial differential equations

$$Q_{x,y} = c_{00}P_{x-1,y-1} + c_{01}P_{x,y-1} + c_{02}P_{x+1,y-1} + \\ c_{10}P_{x-1,y} + c_{11}P_{x,y} + c_{12}P_{x+1,y} + \\ c_{20}P_{x-1,y+1} + c_{21}P_{x,y+1} + c_{22}P_{x+1,y+1}$$

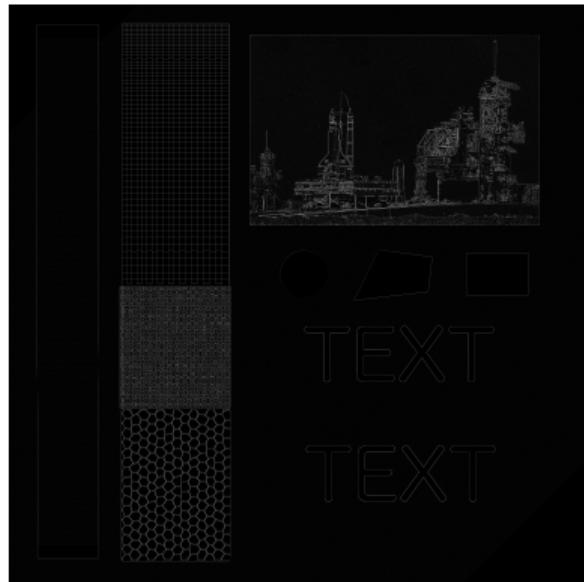
Fluid dynamics, heat transfer, image processing (convolution matrix),  
cellular automata.



# Edge Detection



$$\begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix} \rightarrow$$



# High-Bandwidth Memory (HBM)

## Option 1: numactl

```
user@vega% numactl -m 1 ./stencil test-image.png
```

## Option 2: Memkind

```
1 #include <hbwmalloc.h>
2 ...
3 hbw_posix_memalign((void**)&pixel, 64, sizeof(P)*width*height);
```

```
user@vega% icpc -o stencil *.o -lpng -lmemkind
```



# Streaming Stores

```
1 #pragma omp parallel for
2 for (int i = 1; i < height-1; i++)
3 #pragma omp simd
4 #pragma vector nontemporal
5 for (int j = 1; j < width-1; j++)
6     out[i*width + j] =
7         -in[(i-1)*width + j-1] - in[(i-1)*width + j] - in[(i-1)*width + j+1]
8         -in[(i   )*width + j-1] + 8*in[(i   )*width + j] - in[(i   )*width + j+1]
9         -in[(i+1)*width + j-1] - in[(i+1)*width + j] - in[(i+1)*width + j+1];
```



# Performance

