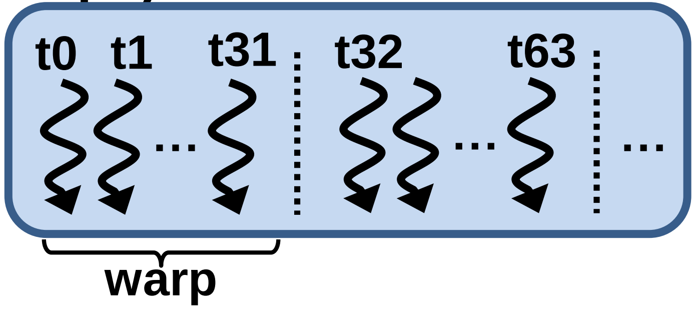
Outline

Control flow

- Threads execute as groups of 32 threads (warps)
  - → Threads in warp share same program counter
- → **SIMT architecture**



- **Diverging threads within a warp**
  - → Threads within a single warp take different paths
  - → Different execution paths within warp are serialized

```
if (threadIdx.x > 2)
{...} else {...}
```

- → **Same code path for threads within warp**
  - → Threads execute synchronously
- → **Granularity of branches: multiple of warp size**

```
if (threadIdx.x / WARP_SIZE > 2) {...} else {...}
```

Different code for different warps: no impact on performance

Launch configuration

- Kernel executes grid of blocks of threads
  → Launch configuration

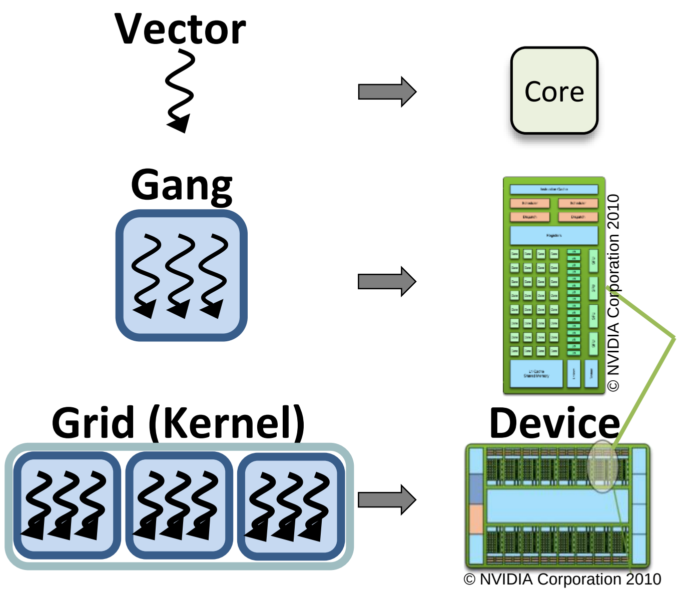
```
myKernel<<<numBlocks, numThreads>>>(…)
```

```
#pragma acc kernels loop gang(numBlocks) vector(numThreads)
```

```
#pragma acc parallel num_gangs(numBlocks) vector_length(numThreads)
```

- Possible mapping to CUDA terminology (GPUs) *compiler dependent*
  → gang = block
  → worker = warp
  → vector = threads
    → Within block (if omitting worker)
    → Within warp (if specifying worker)

- Execution Model



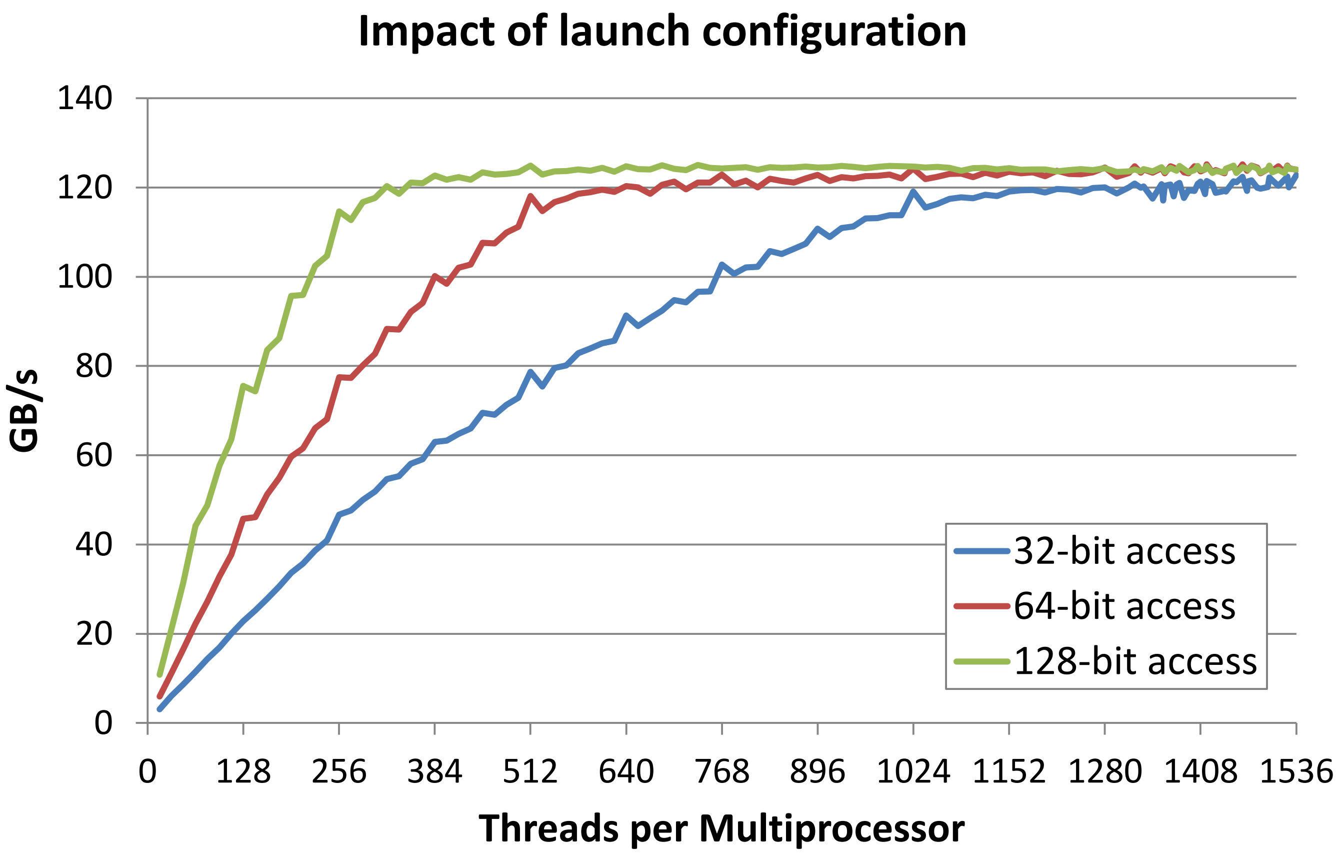
Launch configuration

- **How many threads/blocks to launch?**
  - → OpenACC runtime tries to find good values automatically
    - → Performance portability across different device types possible
    - → Own tuning may deliver better performance on a certain hardware
  - → In CUDA codes, the developer has to specify both sizes
- **Hardware operation**
  - → Instructions are issued in order
  - → Thread execution stalls when one of the operands isn't ready
  - → Latency is hidden by switching threads
    - → GMEM latency: 400-800 cycles
    - → Arithmetic latency: 18-22 cycles

➔ **Need enough threads to hide latency**

Launch configuration

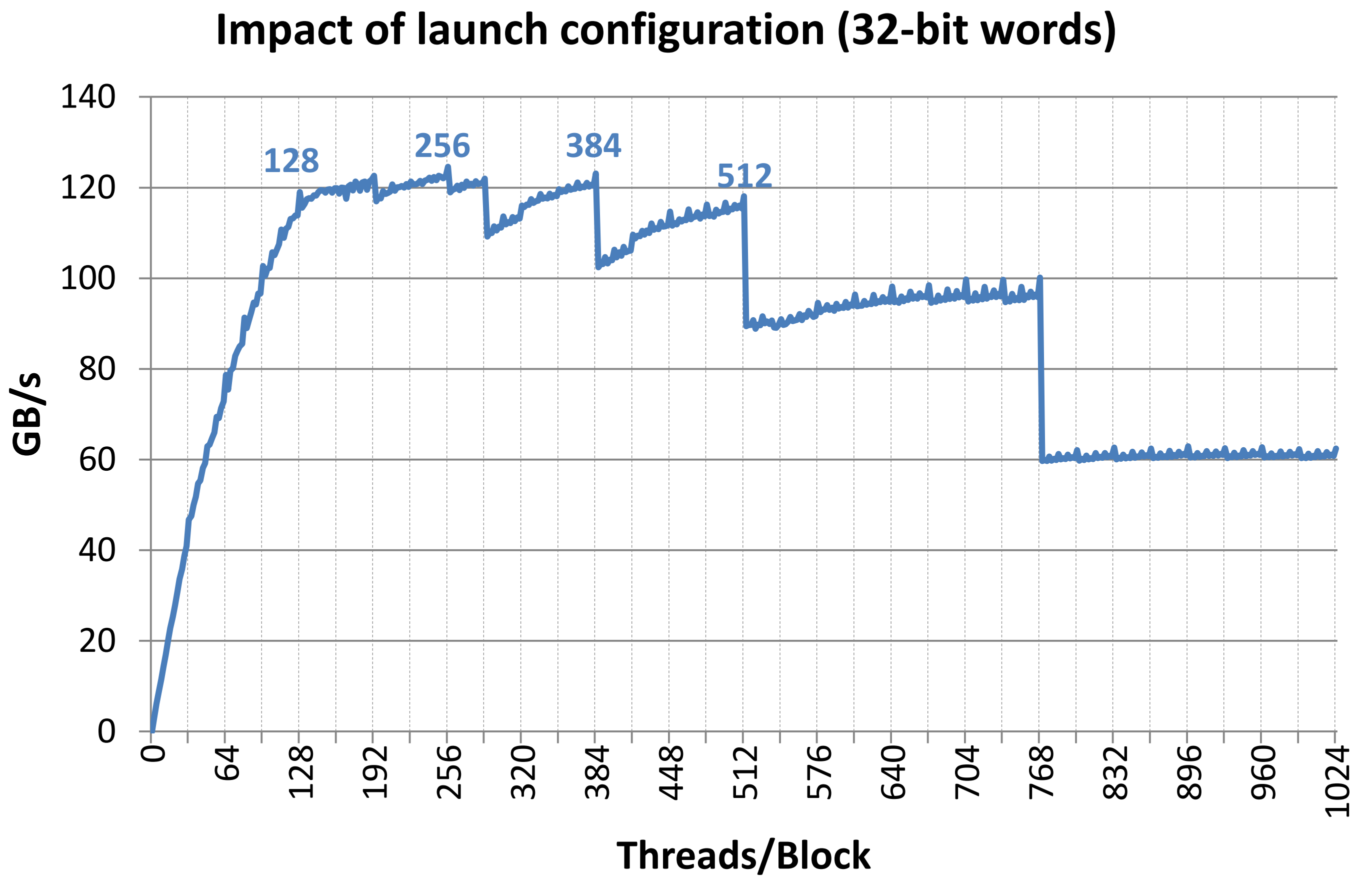
- **Maximizing global memory throughput**
  - → Example program: increment an array of ~67M elements



- NVIDIA Tesla C2050 (Fermi)
- ECC off
- Bandwidth: 144 GB/s

Launch configuration

- **Maximizing global memory throughput**
  - → Example program: increment an array of ~67M elements



- NVIDIA Tesla C2050 (Fermi)
- ECC off
- Bandwidth: 144 GB/s

Launch configuration

- **Threads per block: Multiple of warp size (32)**
  - → Threads are always spanned in warps onto resources
  - → Not used threads are marked as inactive
  - → Starting point: 128-256 threads/block
- **Blocks per grid heuristics**
  - → `#blocks > #SM`
    - → MPs have at least one block to execute
  - → `#blocks / #SM > 2`
    - → MP can concurrently execute up to 8 blocks
    - → Blocks that aren’t waiting in barrier keep hardware busy
    - → Subject to resource availability (registers, smem)
  - → `#blocks > 100` to scale to future devices
  - → Most obvious: `#blocks * #threads = #problem-size`
    - → Multiple elements per thread may amortize setup costs of simple kernels:

      `#blocks * #threads < #problem-size`

→ **Launch MANY threads to keep GPU busy!**

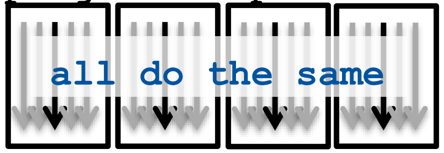
Occupancy

- **Main concept: hiding latencies by switching threads**
  - → Strive for high occupancy
- **Occupancy (per SM) =** $\dfrac{\text{active warps}}{\text{max. supported active warps}}$
- **Some reasons for low occupancy**
  - → Too few blocks launched on GPU (e.g., problem too small)
  - → Unbalanced workloads
  - → Device capabilities (warps, blocks)
  - → Limit of shared resources (registers, shared memory)

→ **Poor instruction issue efficiency (not able to hide latencies)**
  - → BUT: does not <u>necessarily</u> mean low performance
  - → E.g., several simultaneous kernel launches can increase utilization

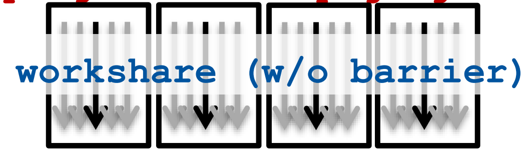
Loop schedule *(in examples)*

```
int main(int argc, const char* argv[]) {
  int n = 256; int blocks = 4; int bsize = 32;
  float *x = (float*) malloc(n * sizeof(float));
  float *y = (float*) malloc(n * sizeof(float));
  // Define scalars a, b & initialize x, y
  // Run SAXPY TWICE
#pragma acc data copyin(x[0:n])
{
#pragma acc parallel copy(y[0:n]) present(x[0:n]) \
                                            num_gangs(blocks) vector_length(bsize)
```

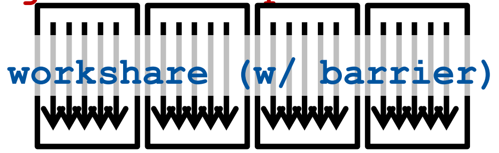


e.g. int tmp = 5;

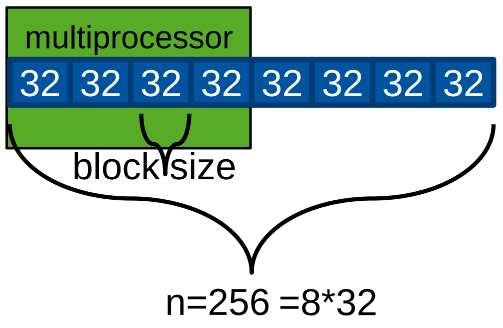
```
#pragma acc loop gang
```



```
#pragma acc loop vector
```



```
  for (int i = 0; i < n; ++i){
          y[i] = a*x[i] + y[i];
  } // Change y on host & do second SAXPY on device
}
  free(x); free(y); return 0; }
```



Note: Loop schedules are implementation dependent. This representation is one (common) example.

Loop schedule *(in OpenACC examples)*

```
#pragma acc parallel vector_length(256)
#pragma acc loop gang vector
  for (int i = 0; i < n; ++i){
      // do something
  }
```

Distributes loop to **n** threads on GPU.
→256 threads per thread block
→usually `ceil(n/256)` blocks in grid

```
#pragma acc parallel vector_length(256)
#pragma acc loop gang vector
  for (int i = 0; i < n; ++i){
    for (int j = 0; j < m; ++j){
         // do something
  }  }
```

Distributes outer loop to **n** threads on GPU. Each thread executes inner loop sequentially.
→256 threads per thread block
→usually `ceil(n/256)` blocks in grid

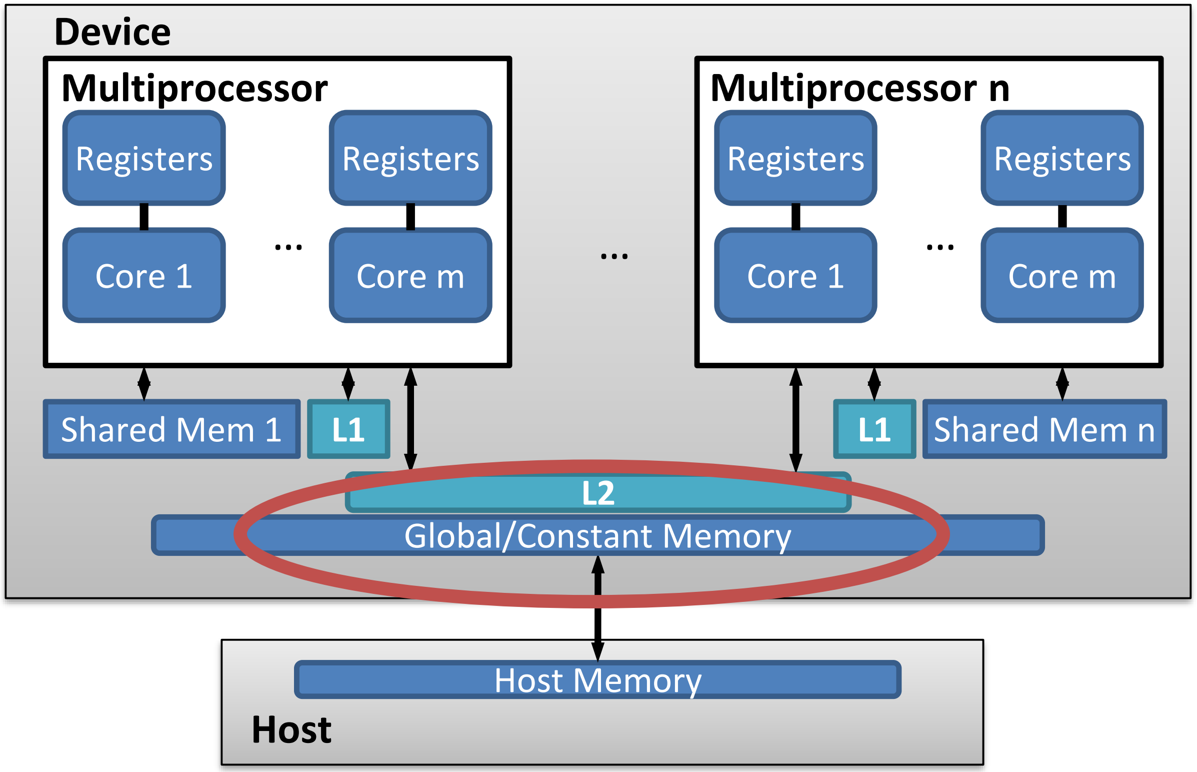
```
#pragma acc parallel vector_length(256)
#pragma acc loop gang
  for (int i = 0; i < n; ++i){
#pragma acc loop vector
    for (int j = 0; j < m; ++j){
         // do something
  }  }
```

Distributes outer loop to GPU multiprocessors (block-wise). Distributes inner loop to threads within thread blocks.
→256 threads per thread block
→usually **n** blocks in grid

Loop schedule principles also apply for CUDA, but must be manually implemented.

Global memory throughput

- **Local memory/ Registers**
- **Shared memory/ L1**
  - → Very low latency (~100x than gmem)
  - → Bandwidth (aggregate): 1+ TB/s (Fermi), 2.5 TB/s (Kepler)
- **L2**
- **Global memory**
  - → High latency (400-800 cycles)
  - → Bandwidth: 144 GB/s (Fermi), 250 GB/s (Kepler)



gmem access

- **Stores:** Invalidate L1, write-back for L2
- **Loads:**

	Caching	Non-caching
Enabling by	default (Fermi)	Compiler-option CUDA: `-Xptxas-dlcm=cg` OpenACC PGI: `-Mx,180,8`
Attempt to hit:	L1 → L2 → gmem	L2 → gmem (no L1: invalidate line if it's already in L1)
Load granularity	128-byte line	32-byte line

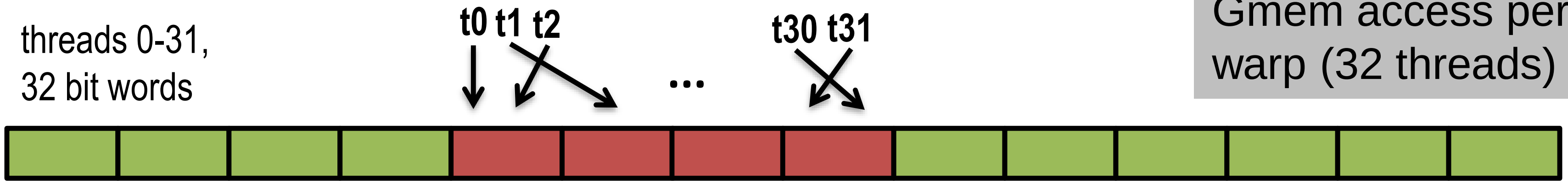
Kepler uses L1 cache only for thread-private data → L2 cache with 128-byte cache line.

→ Threads in a warp provide memory addresses

→ Determine which lines/segments are needed

→ Request the needed lines/segments

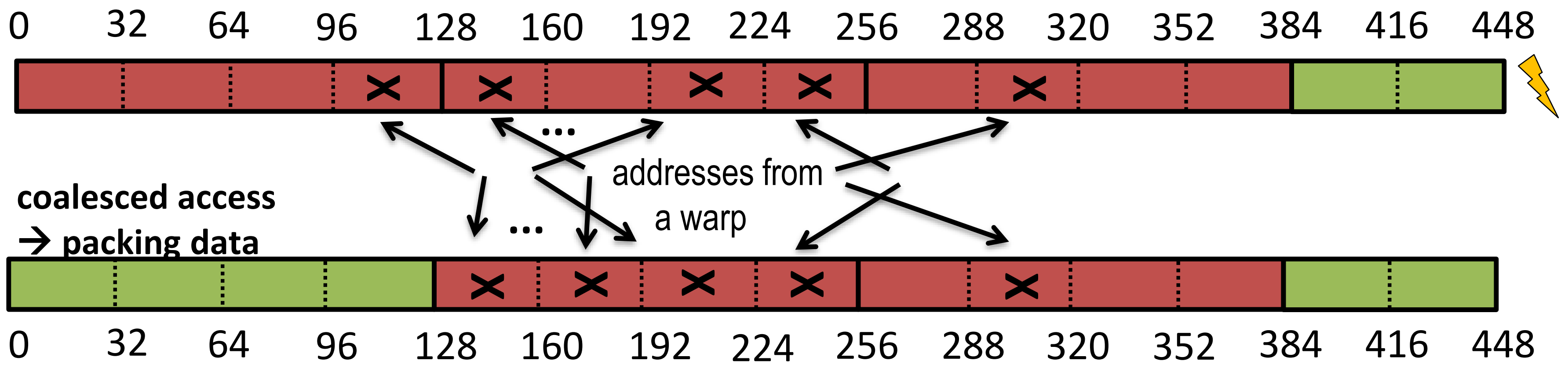
Gmem access per warp (32 threads)



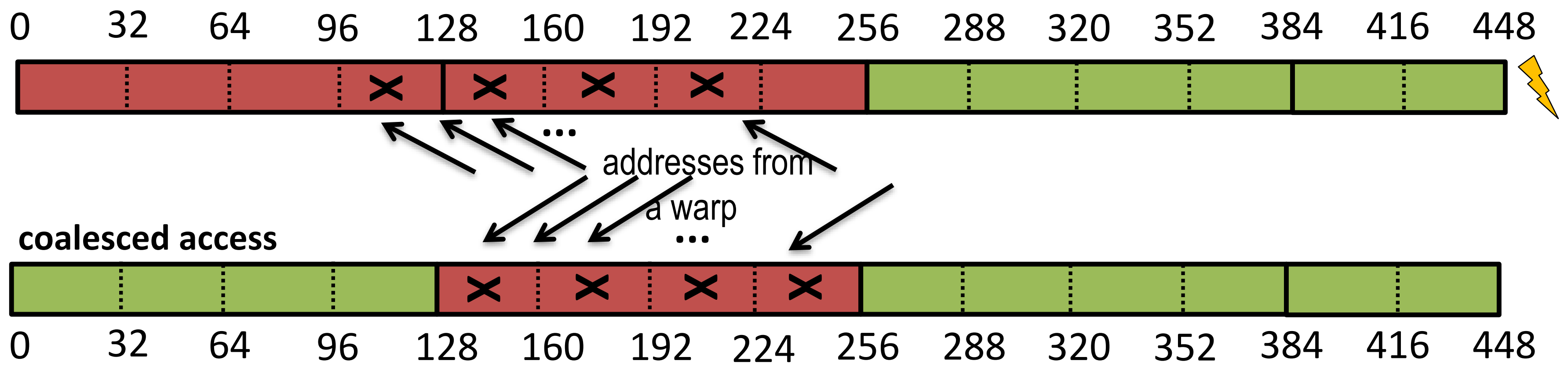
gmem throughput – coalescing

caching loads

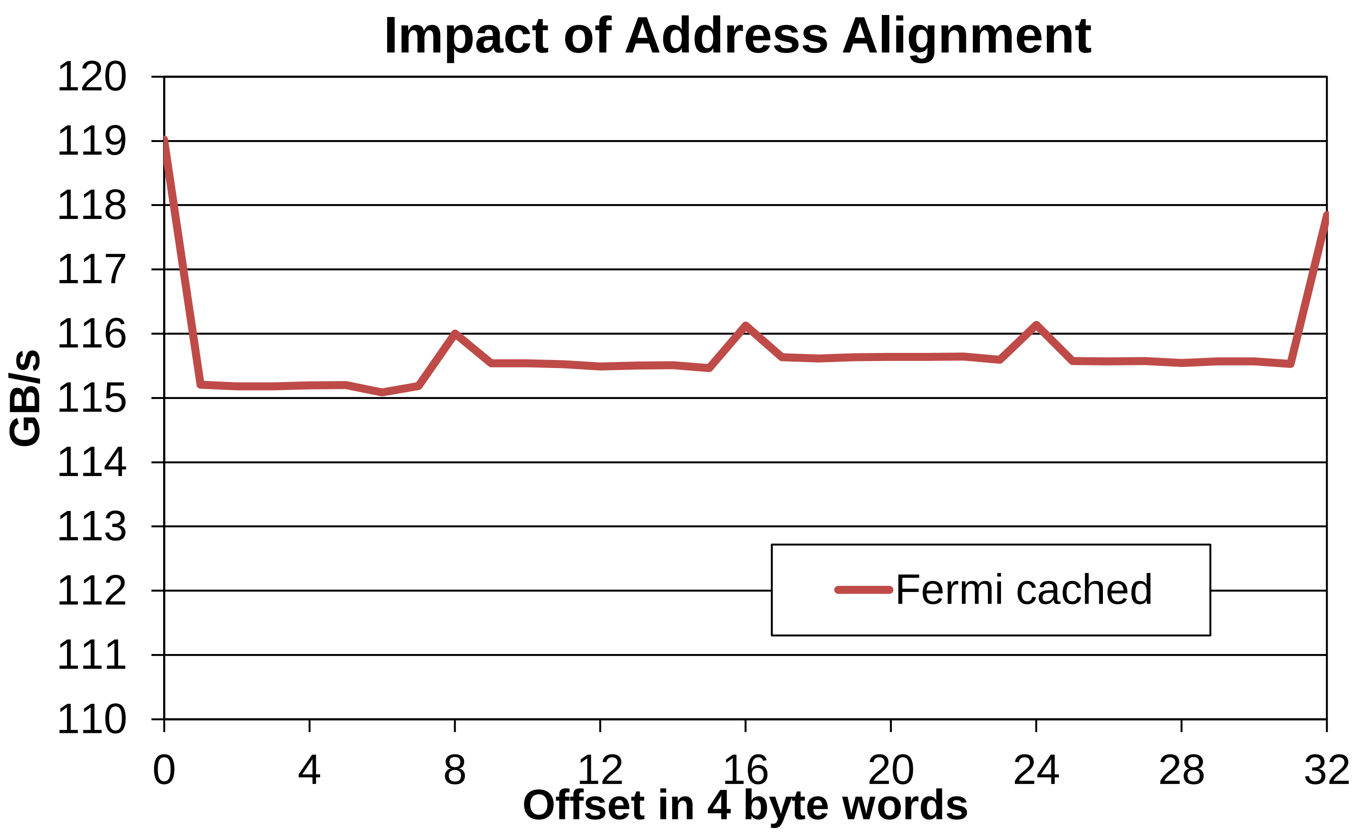
- **Range of accesses**



- **Address alignment**



Impact of address alignment



Example program:

- Copy ~67 MB of floats

- NVIDIA Tesla C2050 (Fermi)
- ECC off
- 256 threads/block

→ **Misaligned accesses can drop memory throughput**

Global memory throughput - Data layout matters (improve coalescing)

- **Example: AoS vs. SoA**

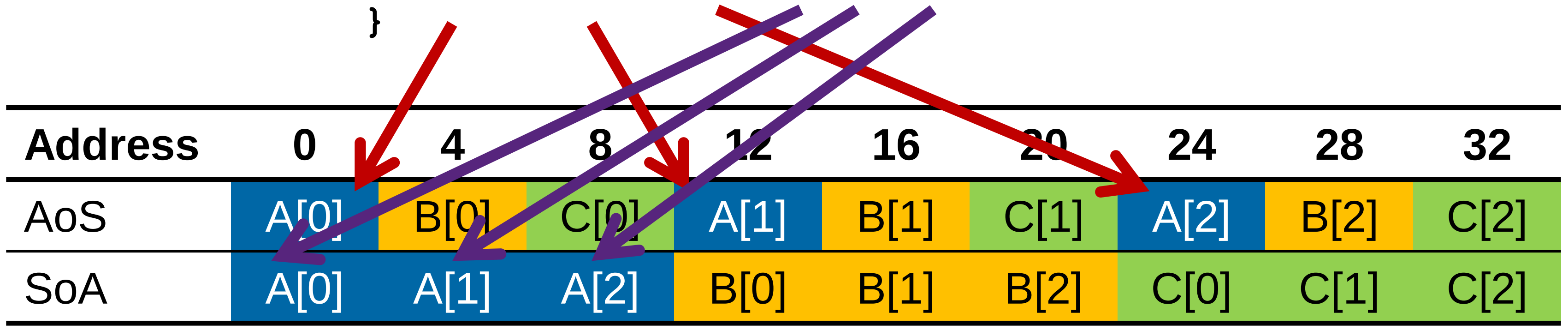
Array of Structures (AoS)

```
struct myStruct_t {
   float a;
   float b;
   int c;
}
myStruct_t myData[];
```

Structure of Arrays (SoA)

```
struct {
   float a[];
   float b[];
   int c[];
}
```

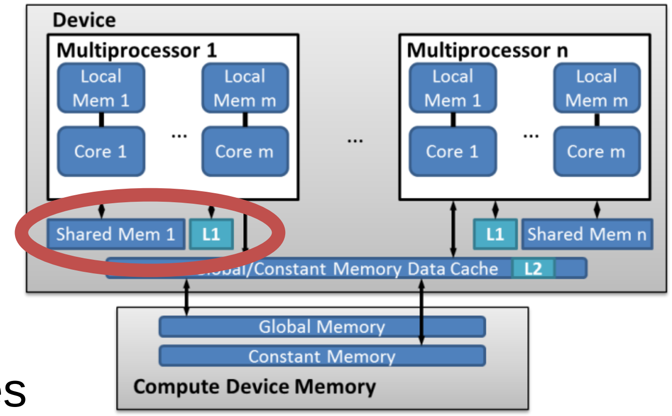
```
#pragma acc kernels
for (int i=0; i<n; i++) {
   … myData[i].a / myData.a[i] …
}
```



Address	0	4	8	12	16	20	24	28	32
AoS	A[0]	B[0]	C[0]	A[1]	B[1]	C[1]	A[2]	B[2]	C[2]
SoA	A[0]	A[1]	A[2]	B[0]	B[1]	B[2]	C[0]	C[1]	C[2]

Shared memory

- **Smem per MP (10s of KB)**
  - → Inter-thread communication within a block
  - → Synchronization to avoid race conditions
- **Low-latency, high-throughput memory**
  - → Cache data in smem to reduce gmem accesses



CUDA

```
__global__ void myKernel(…) {
 // Compute thread IDs x, y &
 // mapping to row, col
 __shared__ double as[BS][BS];
 as[x][y] = a[row][col];
 __syncthreads();
 if (/* valid */) {
  b[row][col] = (as[x][y-1] +
   as[x][y+1] + as[x-1][y] +
   as[x+1][1]) / 4;
}}
```

OpenACC

```
#pragma acc kernels
#pragma acc loop gang vector
  for (int i = 1; i < N-1; ++i){
#pragma acc loop gang vector
   for (int j = 1; j < N-1; ++j){
#pragma acc cache(a[i-1:3][j-1:3])
      b[i][j] = (a[i][j-1] +
       a[i][j+1] +  a[i-1][j] +
       a[i+1][j]) / 4;
  }    }
```

CUDA – Shared Memory

- **Per thread block**
- **Static shared memory**
  - → Kernel code: `__shared__ int myarray[64];`
- **Dynamic shared memory**
  - → Host code (specify shared memory size as third argument):

```
myKernel<<<1, n, n*sizeof(int)>>>(...);
```

  - → Kernel code: `extern __shared__ int myarray[];`
- **Synchronize threads (within thread block)**
  - → `__syncthreads();`

background CUDA

Directives: Caching & Strip-mining

- **Cache construct**
  - → Prioritizes data for placement in the highest level of data cache on GPU

C/C++
```
#pragma acc cache(list)
```

Fortran
```
!$acc cache(list)
```

Sometimes the PGI compiler ignores the `cache` construct.
→ See compiler feedback

  - → Use at the beginning of the loop or in front of the loop

- **Tile clause**
  - → Strip-mines each loop in a loop nest according to the values in *expr-list*

C/C++
```
#pragma acc loop [<schedule>] tile(expr-list)
```

Fortran
```
!$acc loop [<schedule>] tile(expr-list)
```

1. entry applies to innermost loop

background OpenACC

What you have learnt

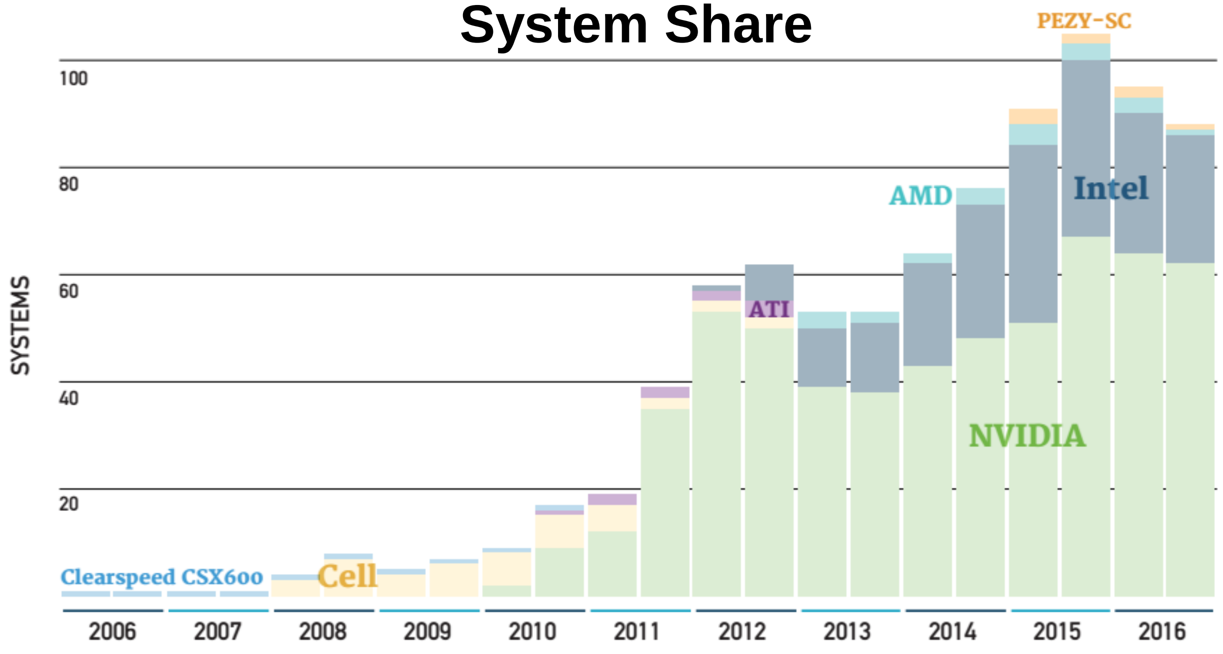
- **Which impact does the PCIe have?**
- **What is branch divergence?**
  - → Which performance impact does it have?
- **What can be a good launch configuration and why?**
  - → How can the launch configuration be specified?
- **What can be done to saturate the bus?**
  - → What is coalescing?
    - → How can it be achieved?
    - → What impact does AoS and SoA have?

Outline

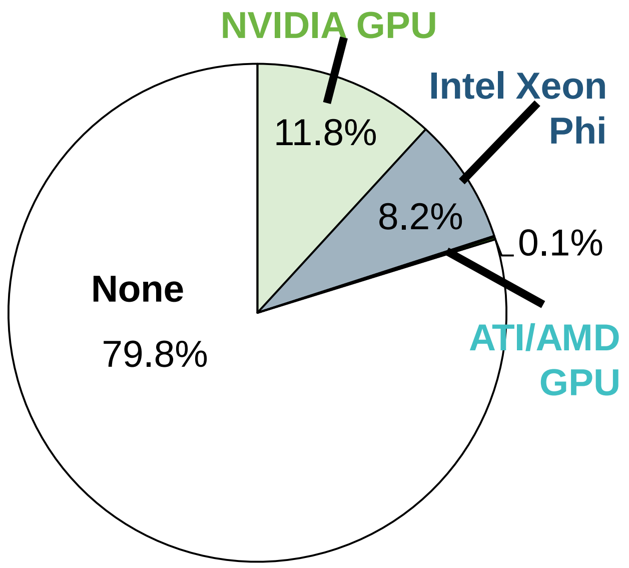
Motivation

- **Accelerators/ co-processors**
  - → GPGPUs (e.g. NVIDIA, AMD)
  - → Intel Many Integrated Core (MIC) Arch. (Intel Xeon Phi)
  - → FPGAs (e.g. Convey), …
- **Heterogeneous Arch.**
  - → Combination of commodity processors & accelerators

System Share



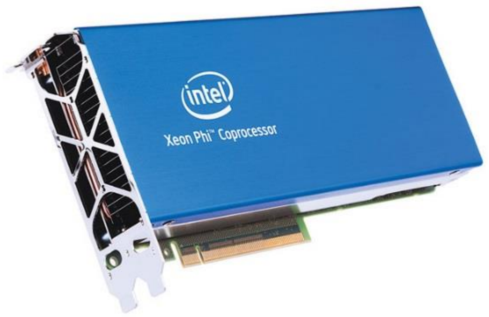
Performance Share

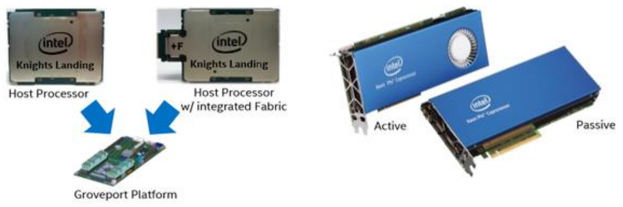


Source: Top500, 11/2016

Intel Xeon Phi

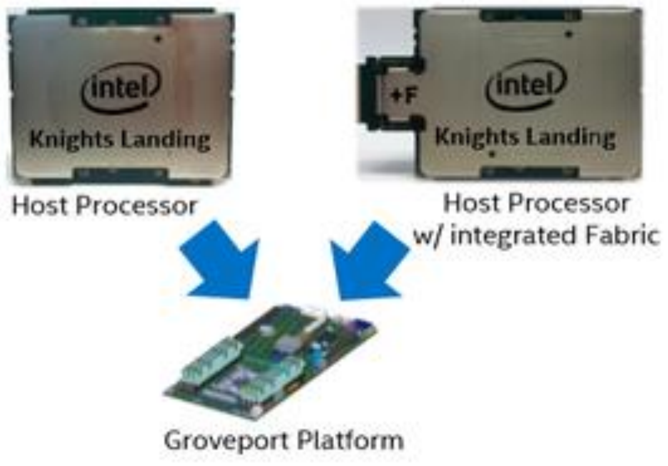
- **Knight's Ferry (engineering sample) (2010)**
  - → 32 x86-based cores, 128 threads
  - → 1.2 GHz, 512-bit SIMD
  - → 1-2 GB GDDR5
- **Knight's Corner (2013)**
  - → 61 x86-based cores, 240 threads
  - → 1.2 GHz, 512-bit SIMD, ~ 1 TFLOPS peak
  - → 8 or 16 GB GDDR5
- **Knight's Landing (2016)**
  - → 72 x86-based cores, 288 threads
  - → 1.5 GHz, 2x 512-bit SIMD, ~ 3.5 TFLOPS peak
  - → 16 GB MCDRAM + 384 GB DDR4





Knights Landing

- **Xeon Phi Processor**
  - → binary compatible with x86-based Servers
  - → use MPI, OpenMP, ...
  - → no need for Offloading
  - → no PCI bottleneck
  - → OPA Fabric on-package available

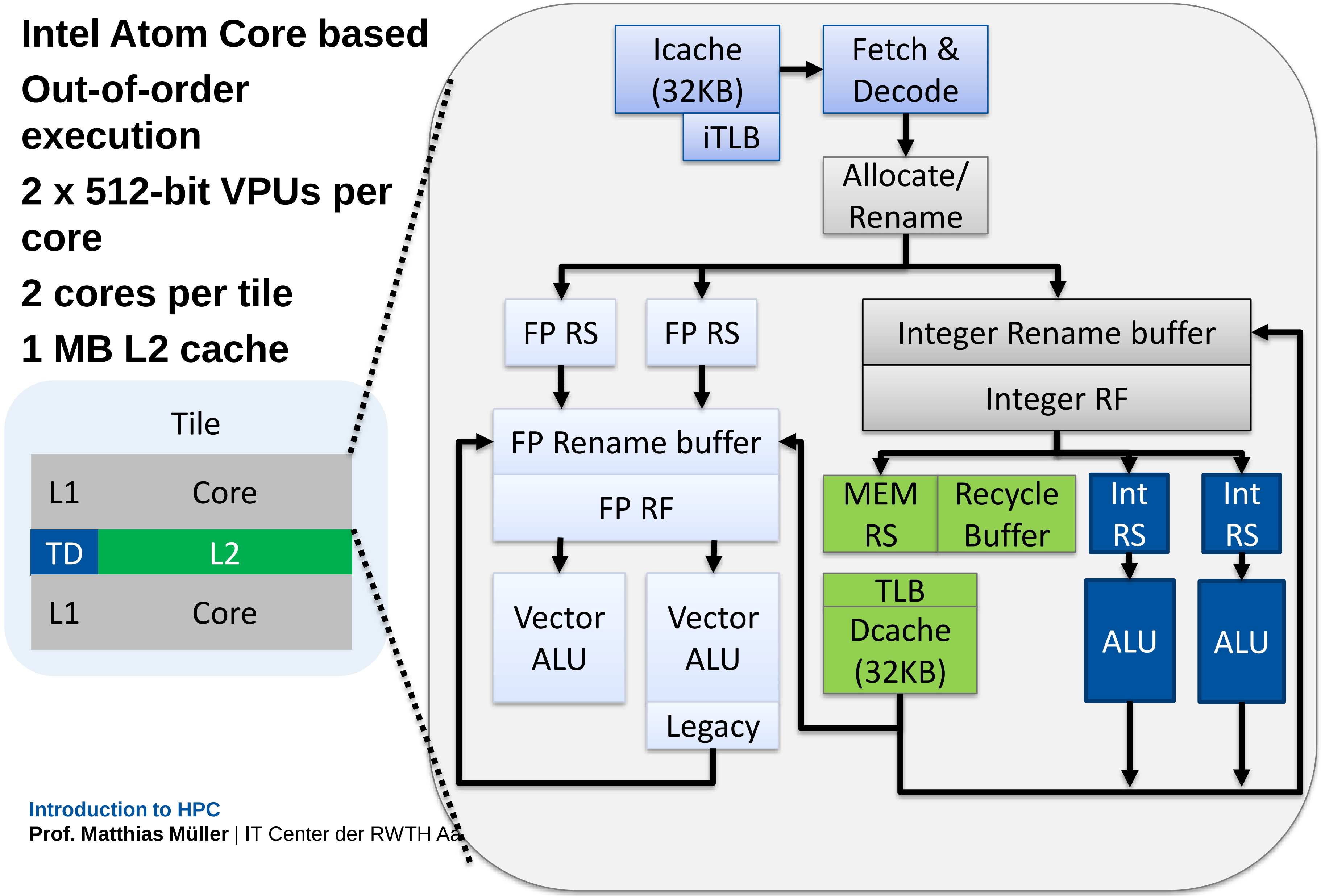


- **Important for applications**
  - → to use SIMD operations
  - → to express a lot of parallelism
  - → make efficient use of the MCDRAM

Outline

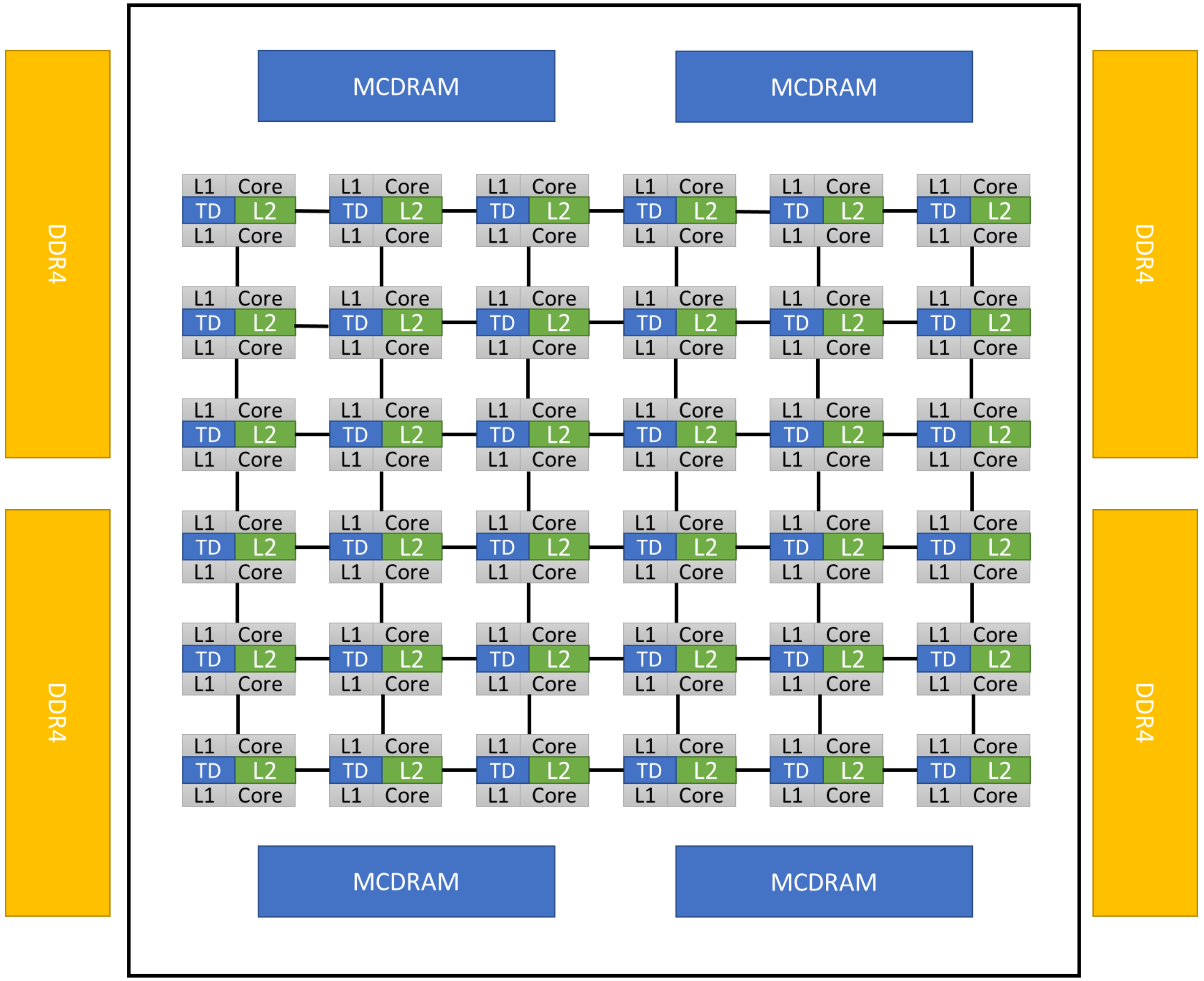
Knights Landing Tile

- Intel Atom Core based
- Out-of-order execution
- 2 x 512-bit VPUs per core
- 2 cores per tile
- 1 MB L2 cache

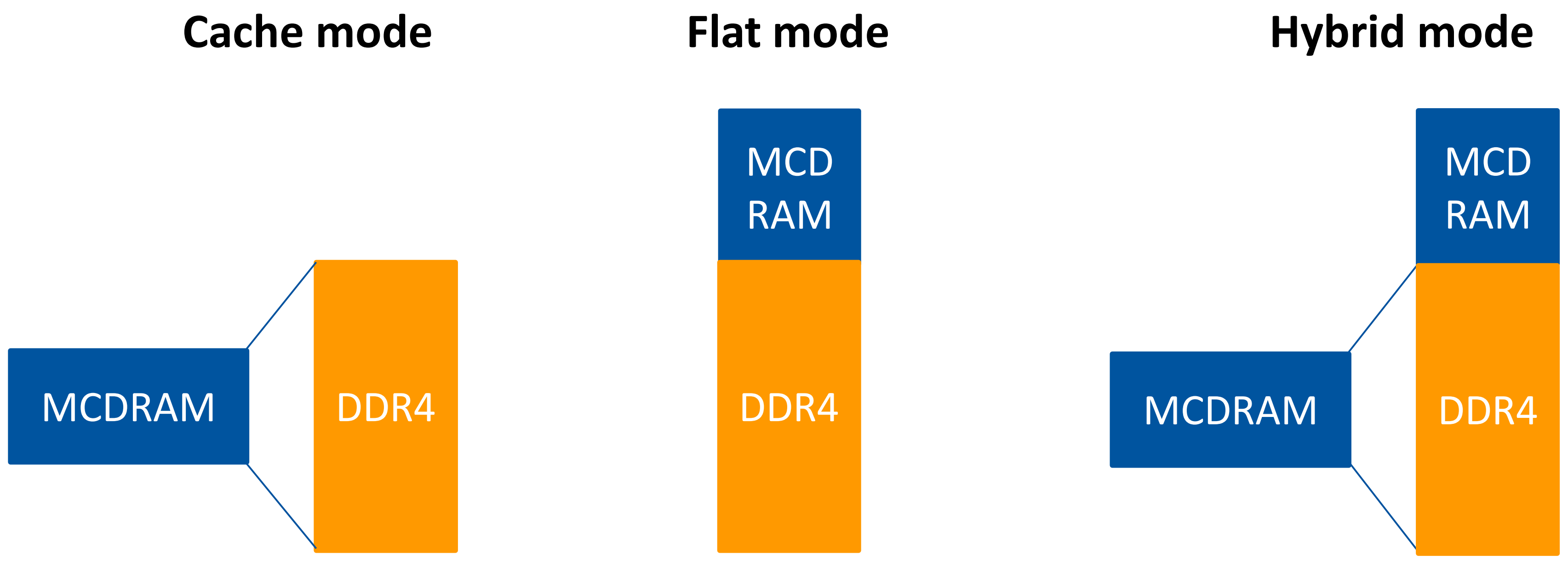


Knights Landing Overview

- up to 36 tiles
- 2D mesh interconnect
- 4 MCDRAM on-chip
- DDR4 off-chip



MCDRAM



- automatic caching to MCDRAM
- easy to use
- everything is cached
- caching overhead

- MCDRAM shown as NUMA node
- explicit allocation needed
- user decides which data is in MCDRAM

- mixture of both modes
- 8/8 GB or 4/12 GB splitting

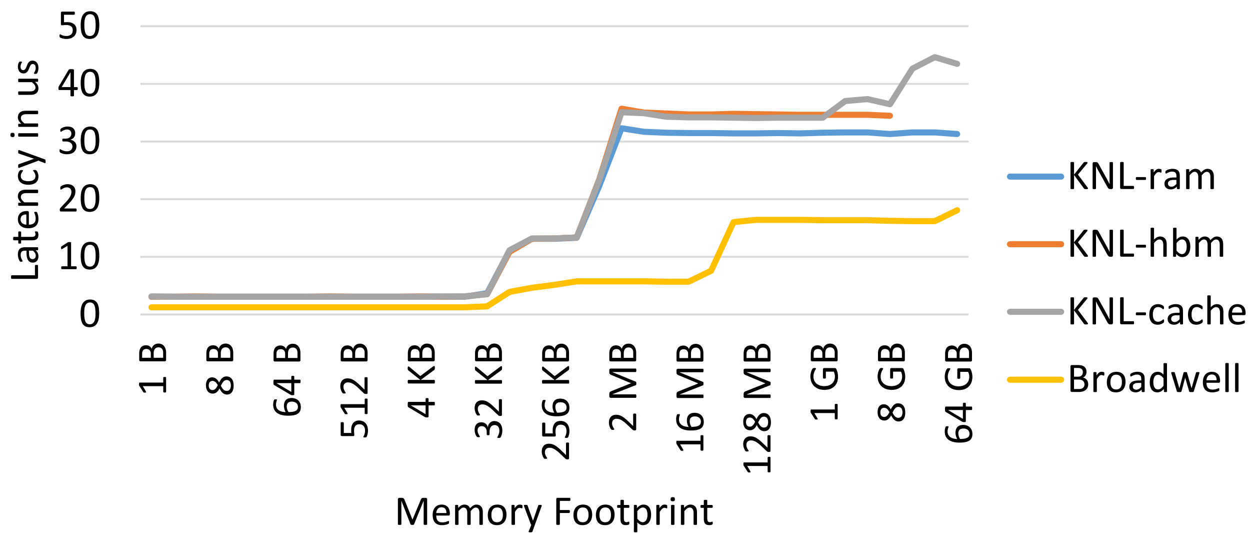
Investigating NUMA topologies

numactl - command line tool to investigate NUMA topologies

- **$ numactl --hardware - prints information on NUMA nodes in the system**
- **$ numactl --show - prints information on available recourses for the process**

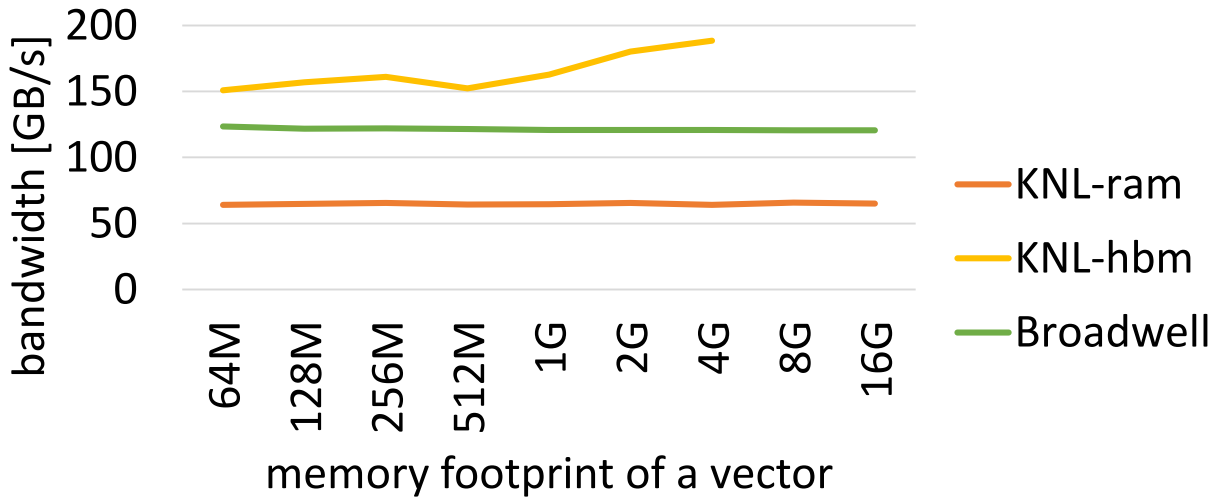
```
$ numactl --hardware
available: 2 nodes (0-1)
node 0 cpus: 0 ..  255
node 0 size: 98178 MB
node 0 free: 94138 MB
node 1 cpus:
node 1 size: 16384 MB
node 1 free: 15932 MB
node distances:
node   0   1
  0:  10  31
  1:  31  10
```

MCDRAM



- **latency of MCDRAM is slightly higher than DDR4**

- **bandwidth of MCDRAM higher than DDR4**
- **close to 200 GB/s**



Controlling NUMAness

numactl - command line tool to investigate and handle NUMA under Linux

- `$ numactl --cpunodebind 0,1,2 ./a.out`

  → only use cores of NUMA node 0-2 to execute a.out

- `$ numactl --physcpubind 0-17 ./a.out`

  → only use cores 0-17 to execute a.out

- `$ numactl --membind 0,3 ./a.out`

  → only use memory of NUMA node 0 and 3 to execute a.out

Controlling NUMAness

libnuma - library for NUMA control

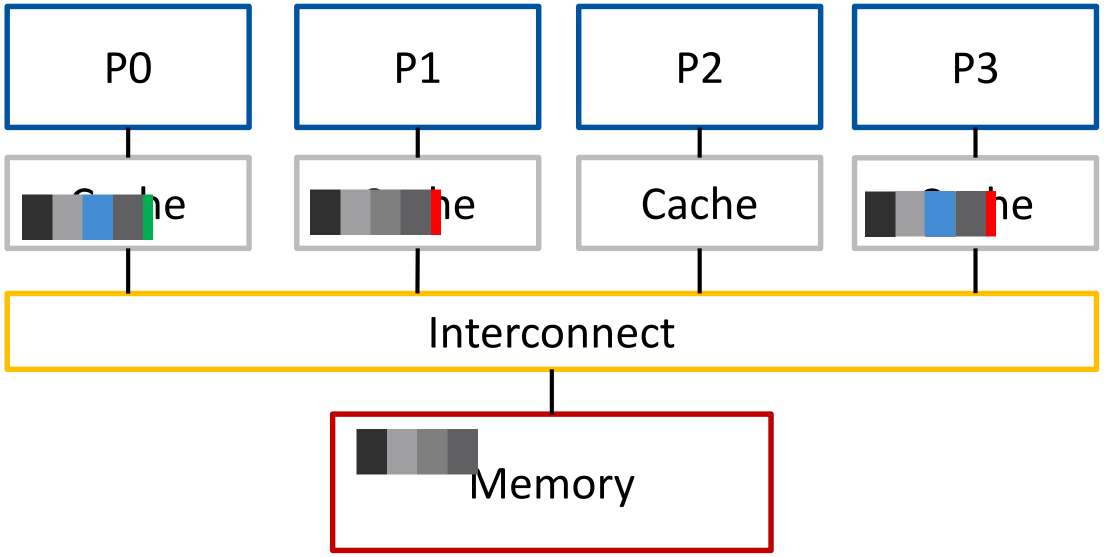
(include numa.h and link -lnuma)

- `void *numa_alloc_local(size_t size);`
  → allocate memory on the local NUMA node
- `void *numa_alloc_onnode(size_t size, int node);`
  → allocate memory on NUMA node node
- `int numa_move_pages(int pid, unsigned long count, void **pages, const int *nodes, int *status, int flags);`
  → migrate memory pages at runtime to different NUMA nodes

Cache coherency (simplified)

1. **Cache-line in Memory**
2. **read by P1 -> copy CL in Cache of P1**
3. **read by P3 -> copy CL in Cache of P3**
4. **read by P0 -> copy CL in Cache of P0**
5. **write by P0 -> P0 changes local CL; other CLs marked invalid**
6. **read by P3 -> local line is invalid, get copy from P0**

how is this done

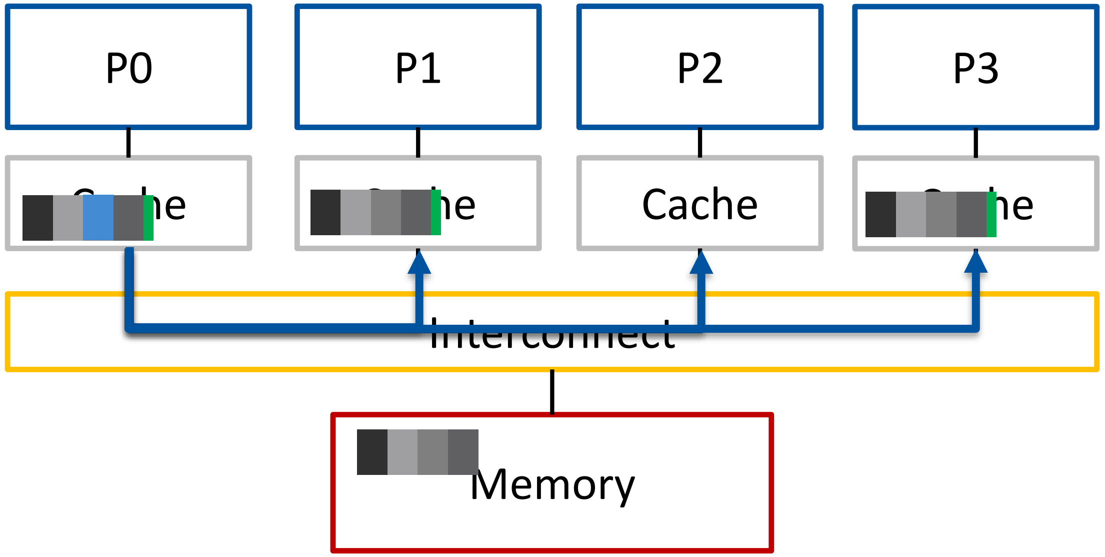


Cache coherency (simplified)

Task: Inform every cache holding the cache line to invalidate it

Solution 1: Snooping

- **send write information to processes**
- **always check for write information and check the local cache to invalidate cache lines**



Pro: Fast, all messages at once

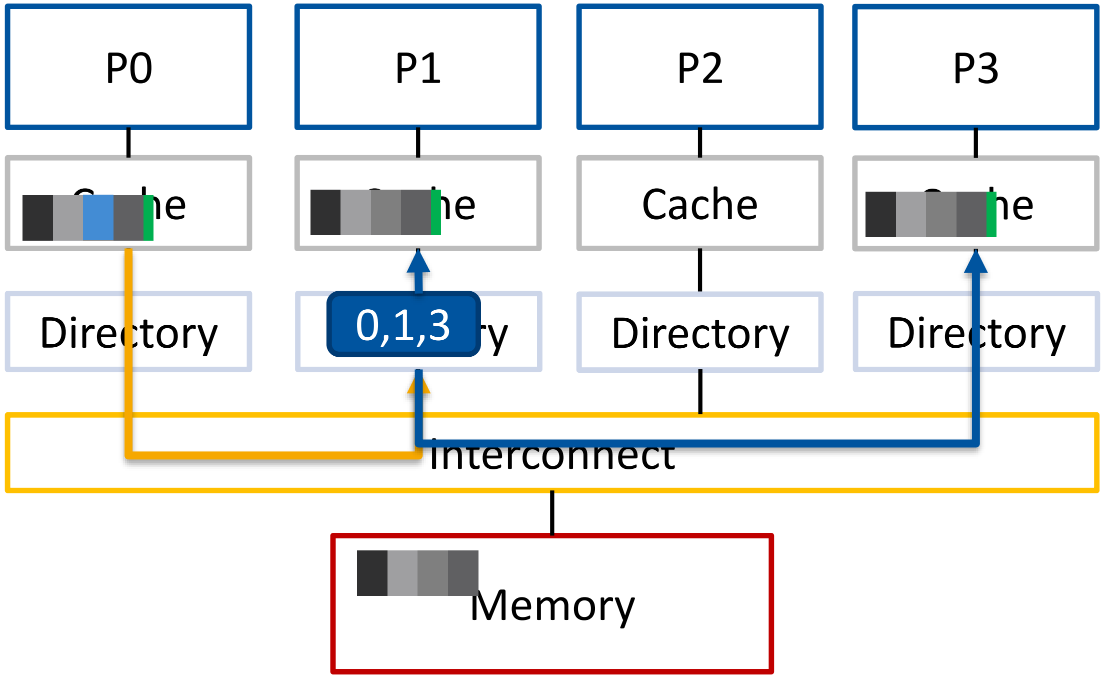
Con: Lots of traffic for many processes

Cache coherency (simplified)

Task: Inform every cache holding the cache line to invalidate it

Solution 2: Directories

- **every cache line belongs to a home directory**
- **store all Ps with a copy**
- **an a write inform the home agent**
- **home agent informs only Ps with a CL**



Pro: reduces traffic on large machines

Con: Slower, one more hop

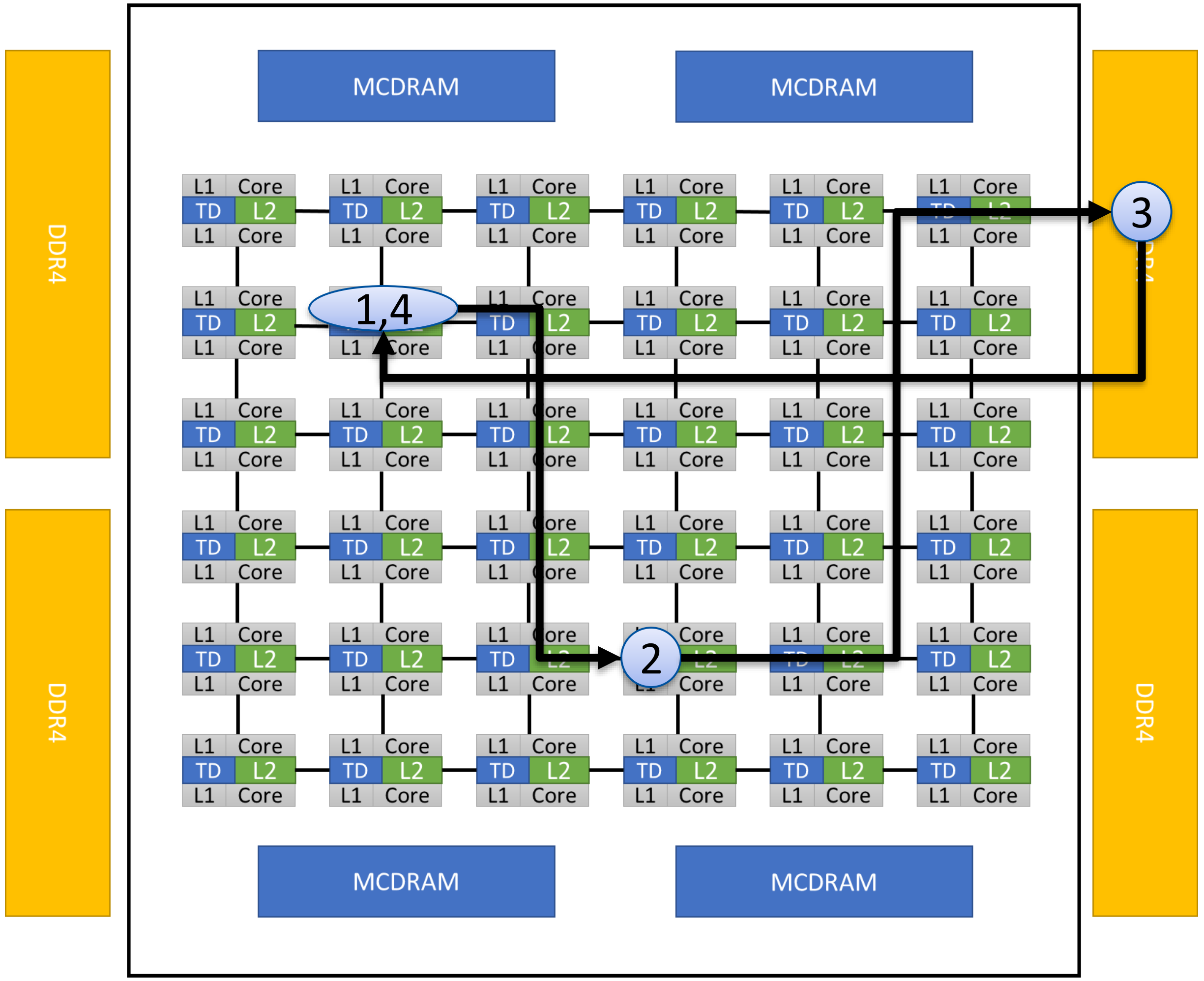
Sub NUMA Clustering

All-to-all mode

1. **miss in local L2$**
2. **send request to TD**
3. **forward request to memory**
4. **send cache-line to requesting core**

TD: random distribution

DDR: random distribution



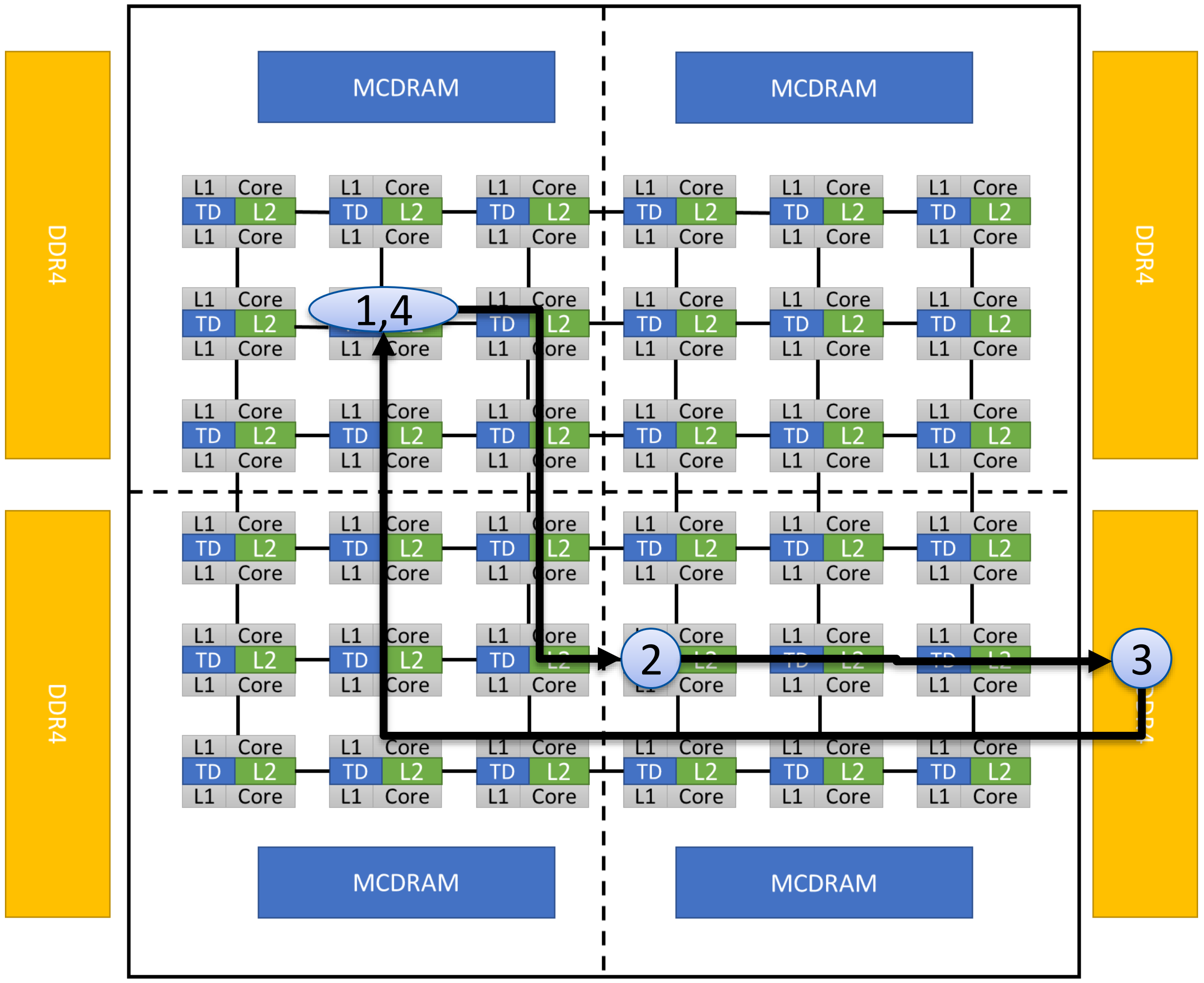
Sub NUMA Clustering

Quadrant mode

1. miss in local L2$
2. send request to TD
3. forward request to memory
4. send cache-line to requesting core

TD: random distribution

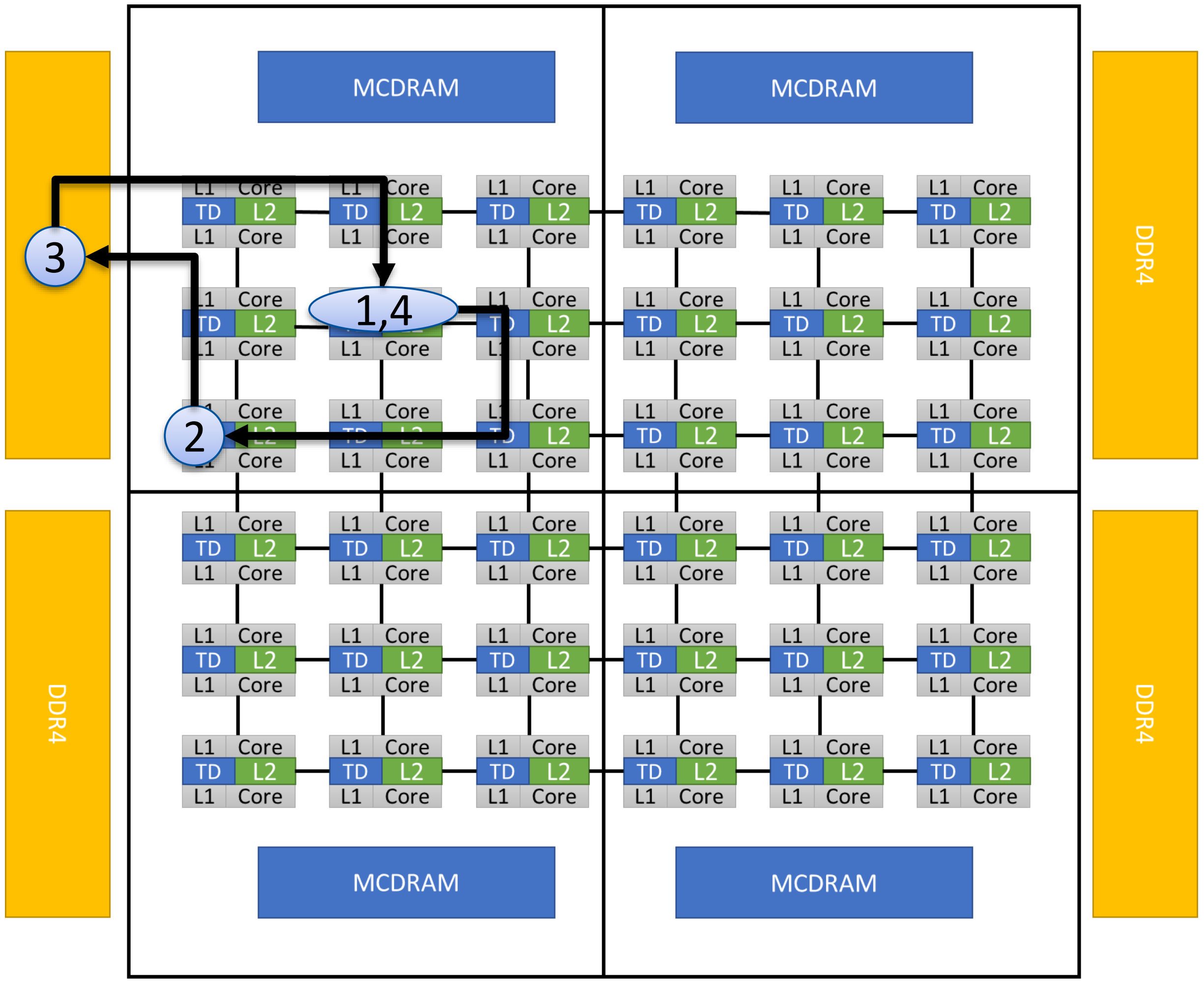
DDR: same quadrant as TD



Sub NUMA Clustering

Sub NUMA Clusters:

1. **miss in local L2$**
2. **send request to TD**
3. **forward request to memory**
4. **send cache-line to requesting core**

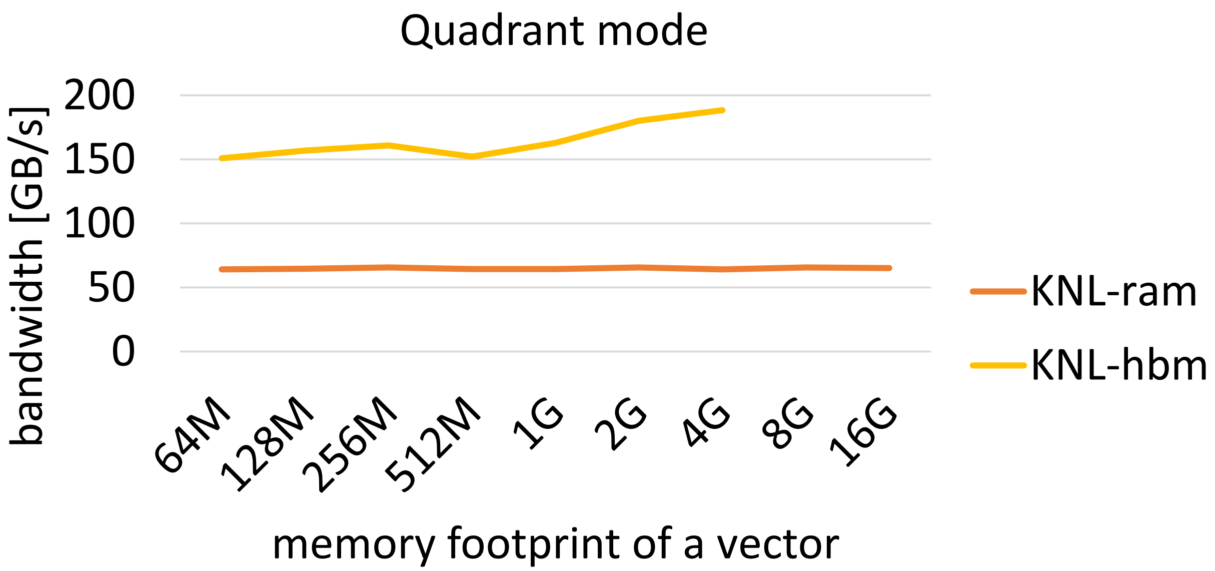


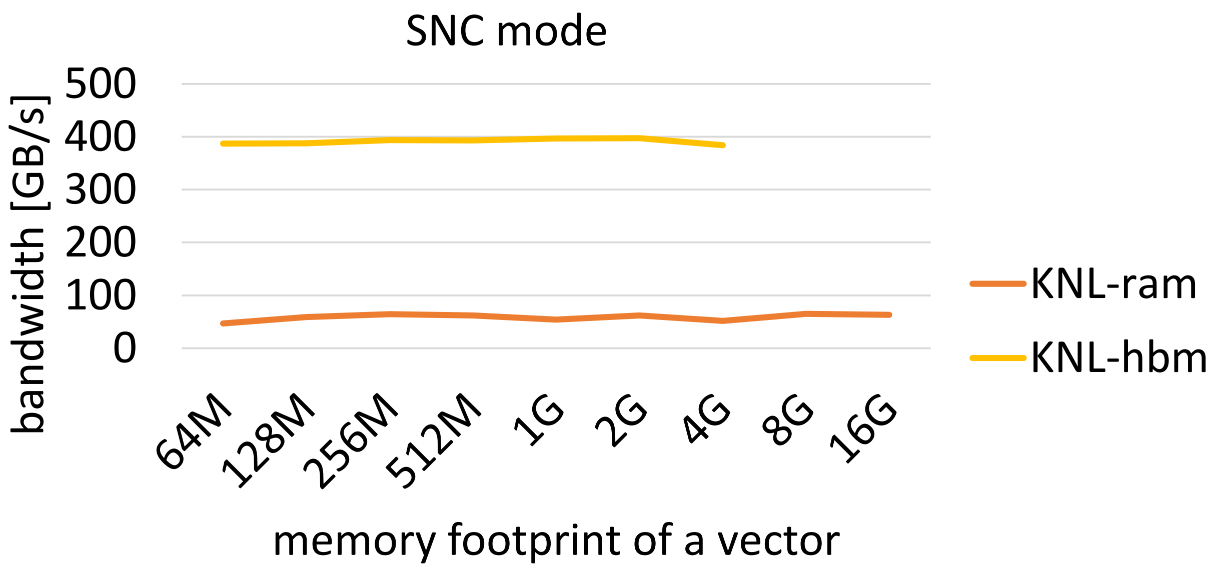
TD: quadrants shown as NUMA-nodes

DDR: same quadrant as TD

Sub NUMA Clustering

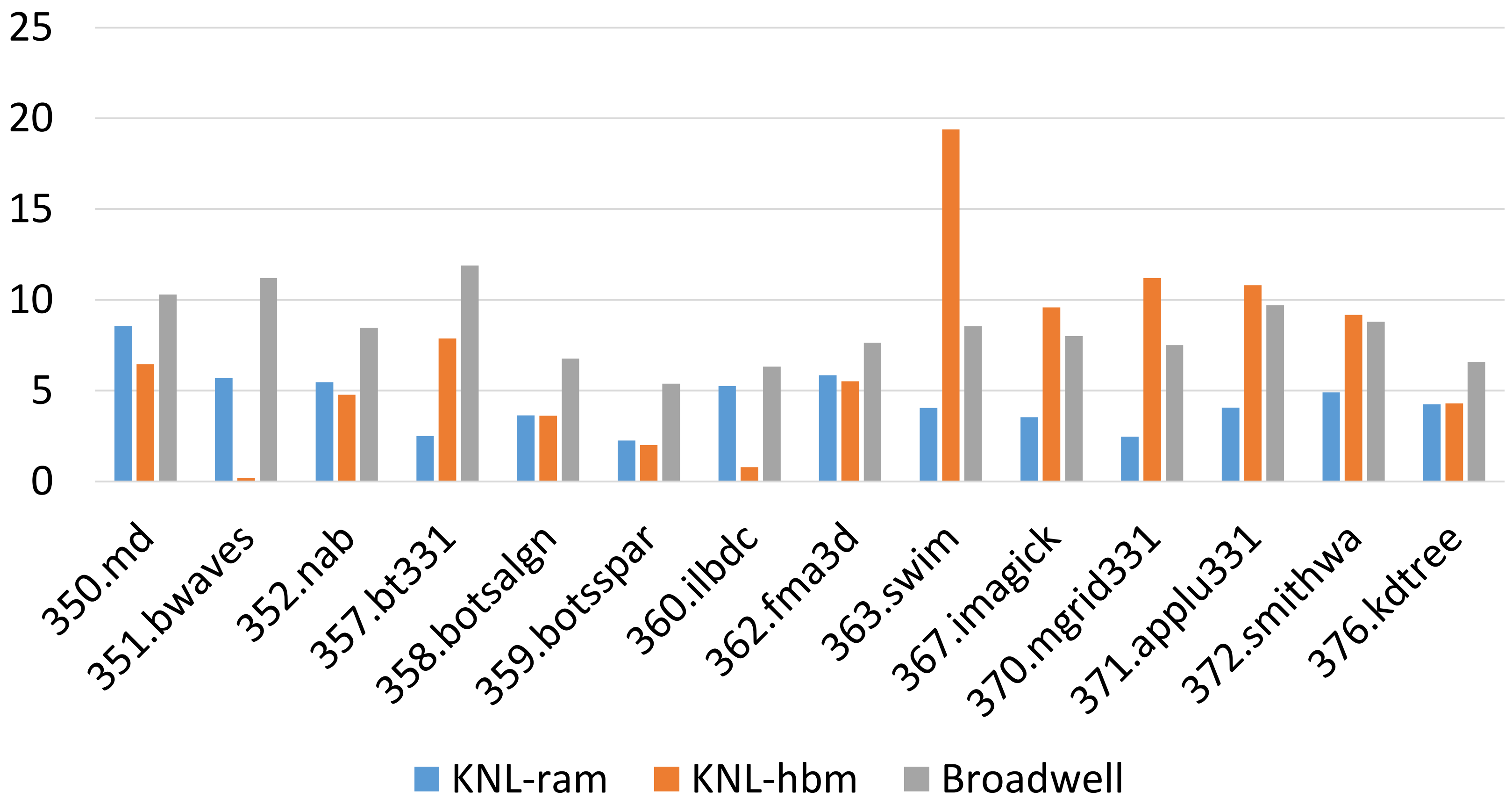
```
$ numactl --hardware
available: 8 nodes (0-7)
node 0 cpus: 0-15,64-79, 128-143, 192-207
node 0 size: 24450 MB
node 0 free: 21956 MB
node 1 cpus: 16-31, 80-95, 144-159, 208-223
node 1 size: 24576 MB
node 1 free: 23266 MB
node 2 cpus: 32-47, 96-111, 160-175, 224-239
node 2 size: 24576 MB
node 2 free: 23306 MB
node 3 cpus: 48-63, 112-127, 176-191, 240-255
node 3 size: 24576 MB
node 3 free: 23590 MB
node 4 cpus:
node 4 size: 4096 MB
node 4 free: 3981 MB
node 5 cpus:
node 5 size: 4096 MB
node 5 free: 3984 MB
node 6 cpus:
node 6 size: 4096 MB
node 6 free: 3981 MB
node 7 cpus:
node 7 size: 4096 MB
node 7 free: 3984 MB
node distances:
node   0   1   2   3   4   5   6   7
  0:  10  21  21  21  31  41  41  41
  1:  21  10  21  21  41  31  41  41
  2:  21  21  10  21  41  41  31  41
  3:  21  21  21  10  41  41  41  31
  4:  31  41  41  41  10  41  41  41
  5:  41  31  41  41  41  10  41  41
  6:  41  41  31  41  41  41  10  41
  7:  41  41  41  31  41  41  41  10
```





SPEC Benchmarks

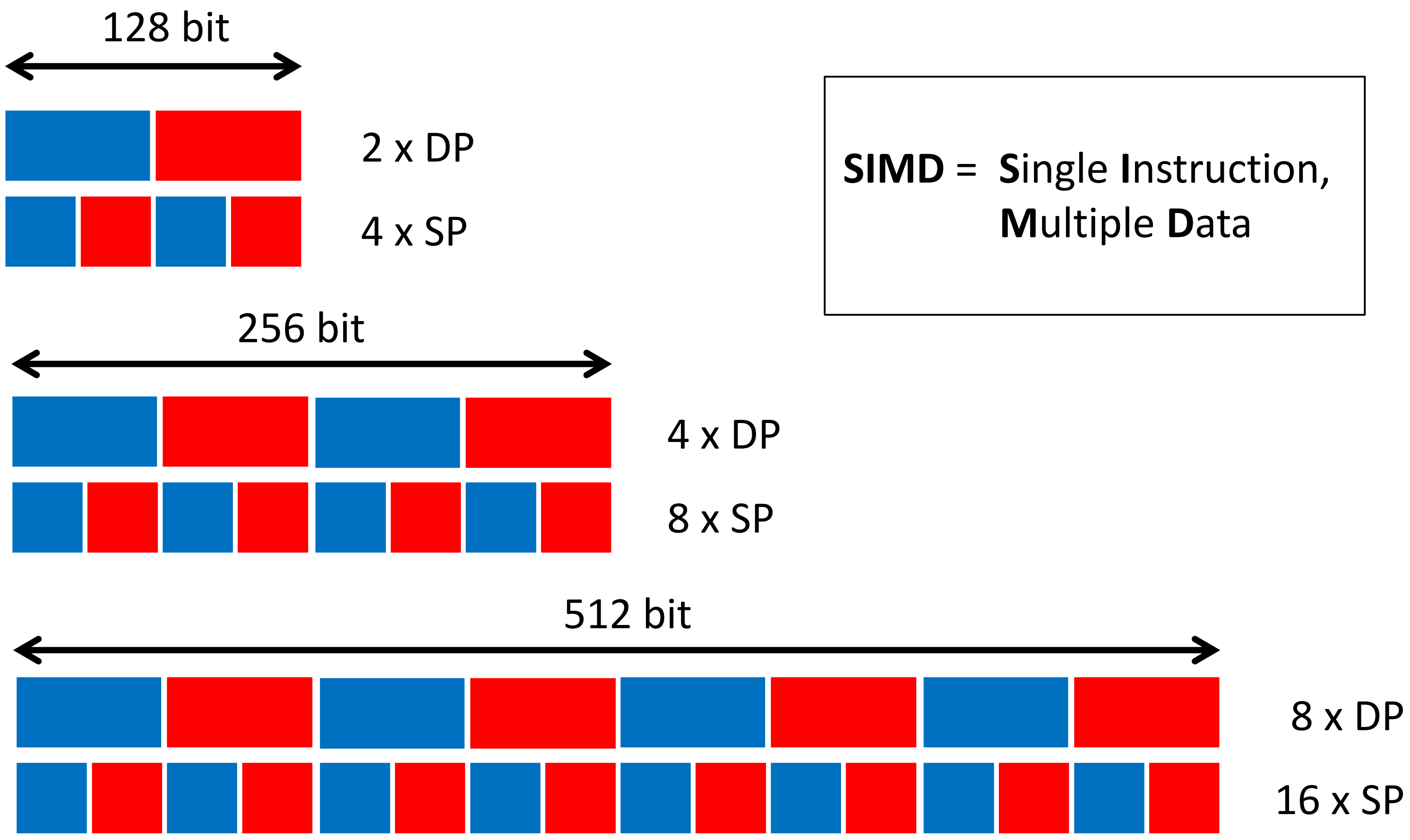
- SPEC-OMP benchmarks on KNL and Broadwell
- no code optimization done so far



Outline

Vectorization

- **SIMD Vector Capabilities**



SSE - Streaming SIMD Extension

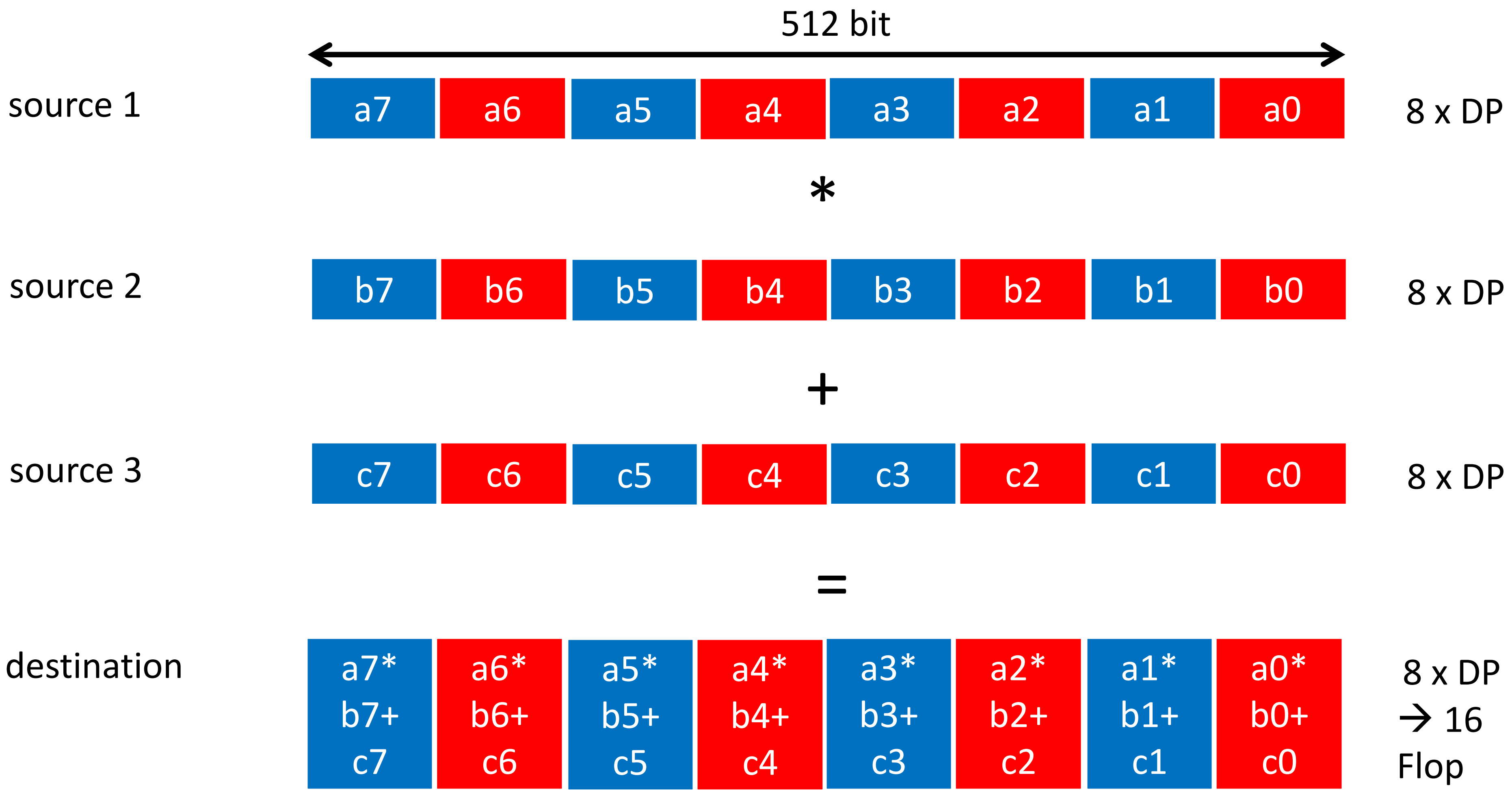
AVX – Advanced Vector Extension

MIC – Many Integrated Core Architecture

SIMD = **S**ingle **I**nstruction, **M**ultiple **D**ata

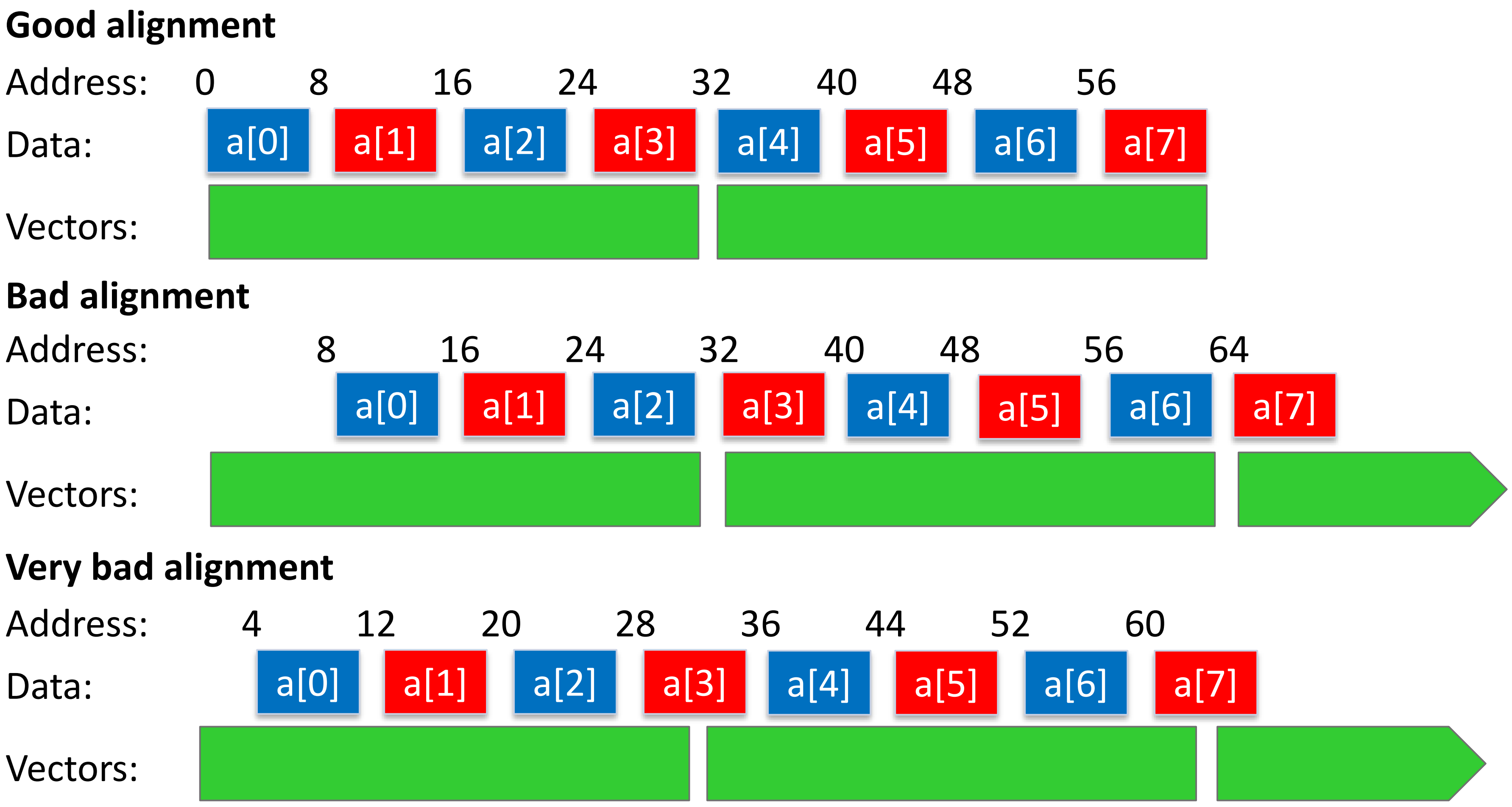
Vectorization

- **SIMD Fused Multiply Add**



Vectorization + Data Alignment

- Vectorization works best on aligned data structures.



Good alignment
Address: 0 8 16 24 32 40 48 56
Data: a[0] a[1] a[2] a[3] a[4] a[5] a[6] a[7]
Vectors:
Bad alignment
Address: 8 16 24 32 40 48 56 64
Data: a[0] a[1] a[2] a[3] a[4] a[5] a[6] a[7]
Vectors:
Very bad alignment
Address: 4 12 20 28 36 44 52 60
Data: a[0] a[1] a[2] a[3] a[4] a[5] a[6] a[7]
Vectors:

Compiler Support for SIMD Vectorization

- **Intel auto-vectorizer**
  - → Combination of loop unrolling and SIMD instructions to get vectorized loops
  - → No guarantee given, compiler might need some hints
- **C/C++ aliasing:** use `restricted` keyword
- **Compiler feedback**
  - → Use `-vec-report [n]` to control the diagnostic information of the vectorizer
  - → concentrate on hotspots for optimization

```
Example: -vec-report=3

driver.c(96): (col. 3) remark: loop was not vectorized: not inner loop
driver.c(64): (col. 5) remark: loop was not vectorized: existence of vector dependence
driver.c(68): (col. 7) remark: vector dependence: assumed FLOW dependence between f line 68 and dx line 65
driver.c(65): (col. 7) remark: vector dependence: assumed ANTI dependence between dx line 65 and f line 68
driver.c(68): (col. 7) remark: vector dependence: assumed FLOW dependence between f line 68 and dy line 66
```

OpenMP SIMD Loop Construct

- **Vectorize a loop nest**
  - → Cut loop into chunks that fit a SIMD vector register
  - → No parallelization of the loop body

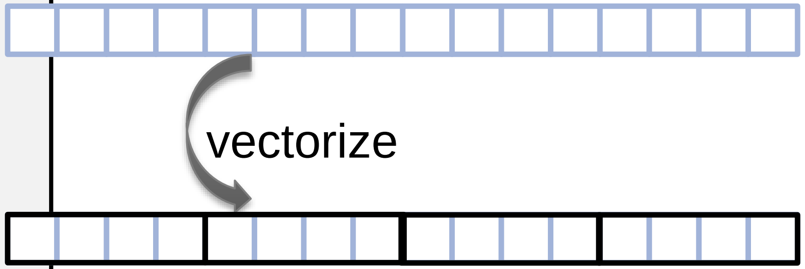
C/C++

```
#pragma omp simd [clause[[,] clause],…]
for-loops
```

Fortran

```
!$omp simd [clause[[,] clause],…]
do-loops
```

```
void sprod(float *a, float *b, int n)
{
  float sum = 0.0f;
#pragma omp simd reduction(+:sum)
  for (int k=0; k<n; k++)
    sum += a[k] * b[k];
  return sum;
}
```



Source: Michael Klemm, Intel

OpenMP SIMD Loop Construct

- **Parallelize & vectorize loop nest**

```
C/C++
#pragma omp for simd [clause[[,] clause],…]
for-loops
```

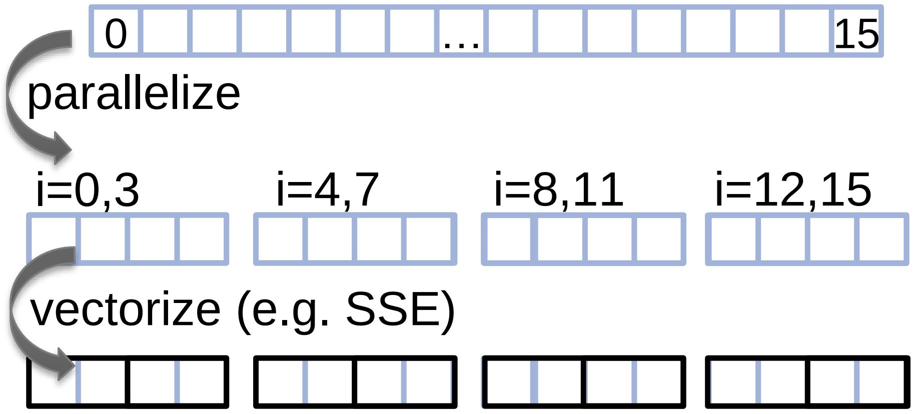
1. Parallelization

   →Distribution of iterations across implicit tasks of Parallel Region

2. Vectorization

   →Convertion of chunks of iterations to SIMD loops

```
void vecAdd(double *a, double *b,
                   double *c, int n)
{
#pragma omp for simd
  for (int i=0; i<n; i++)
    c[i] = a[i] * b[i];
}
```

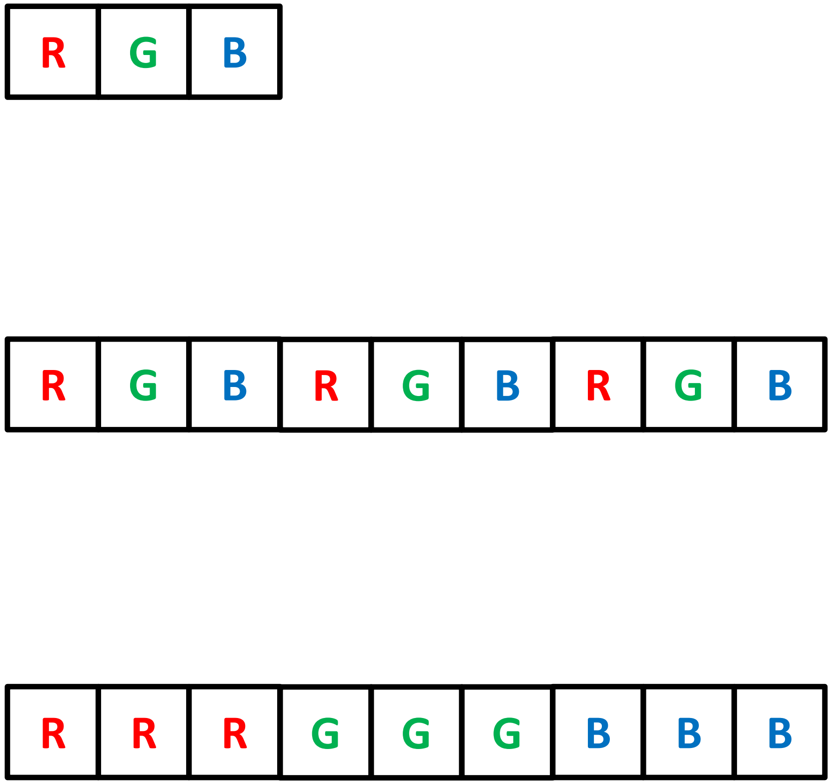


SoA vs. AoS

- **Use Structure of Arrays (SoA) instead of Array of Structures (AoS)**
  - → Color structure

```
struct Color{ //AoS
    float r;
    float g;
    float b;
}
Color* c;
```

```
struct Colors{ //SoA
    float* r;
    float* g;
    float* b;
}
```



- **Detailed information: Intel Vectorization Guide**
  http://software.intel.com/en-us/articles/a-guide-to-auto-vectorization-with-intel-c-compilers/

What you have learnt

- **How does a Knights Landing look like?**
  - → How does the memory layout look like?
  - → In which modes can it be operated?
  - → How many threads/ vector-widths are available?
- **Which optimization strategies should be applied?**
  - → Which impact does vectorization have?
  - → How can vectorization be achieved?

Outline

General GPU Links

- **GPGPU Community:** http://gpgpu.org/
- **GPU Computing Community:** http://gpucomputing.net/
- **GPU Science:** http://gpuscience.com/

- **GPU Technology Conference (GTC) On-Demand**
  http://www.gputechconf.com/gtcnew/on-demand-GTC.php

  Collection of presentations given in the context of different GPU conferences & workshops

- **Webinars**

  Videos on different GPU topics

  https://developer.nvidia.com/gpu-computing-webinars

CUDA Links

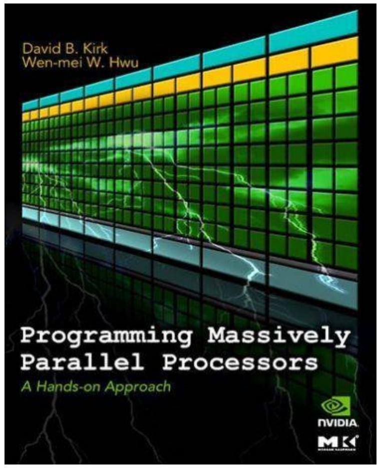
- **Nvidia CUDA Zone (Toolkit, Profiler, SDK, documentation,…): http://www.nvidia.com/object/cuda_home_new.html**
  - → CUDA Programming Guide
  - → CUDA Best Practice Guide

- **PGI's CUDA Fortran: http://www.pgroup.com/resources/cudafortran.htm**
- **PGI's CUDA-x86: http://www.pgroup.com/resources/cuda-x86.htm**

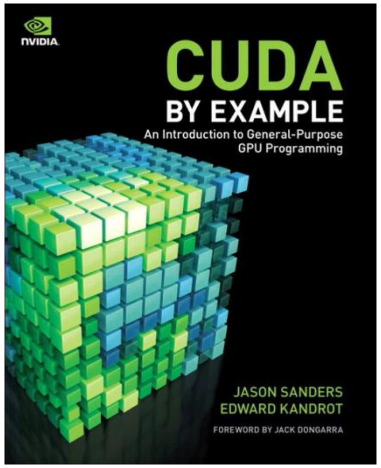
- **Volkov2010: http://www.cs.berkeley.edu/~volkov/volkov10-GTC.pdf**

CUDA Books

- David Kirk und Wen-Mei W. Hwu: *Programming Massively Parallel Processors – A Hands-on Approach* (2010)

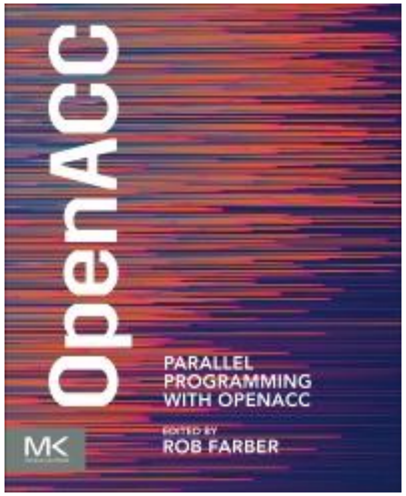


- Jason Sanders und Edward Kandrot: *CUDA by Example: An Introduction to General-Purpose GPU Programming* (2010)


NVIDIA
CUDA
BY EXAMPLE
An Introduction to General-Purpose GPU Programming
JASON SANDERS
EDWARD KANDROT
FOREWORD BY JACK DONGARRA

OpenACC Book/ Links

- **Rob Farber: Parallel Programming with OpenACC, 1st Edition, Morgan Kaufmann (2016)**



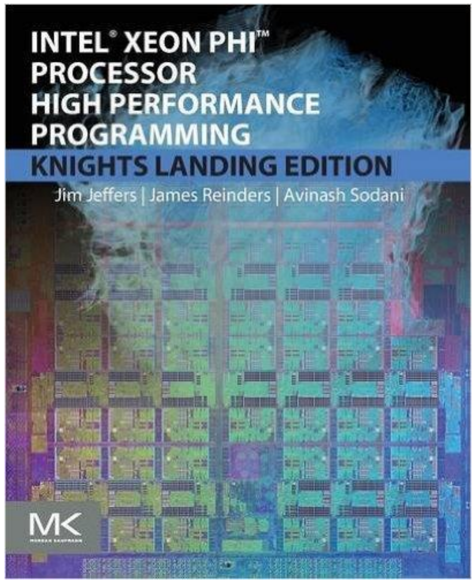
- **OpenACC Home, Specification, Quick Reference Card:** www.openacc.org/
- **OpenACC@PGI:** http://www.pgroup.com/resources/accel.htm
  - → Tutorial presentations and articles, e.g. “OpenACC Features in the PGI C Compiler”
  - → Tutorial videos and webinars
  - → All kind of articles: http://www.pgroup.com/resources/articles.htm
- **OpenACC@NVIDIA:** https://developer.nvidia.com/openacc
  - → Blog & Examples: https://developer.nvidia.com/blog/tags/4621

OpenACC Links

- **Dr. Dobb’s**
  - → http://www.drdobbs.com/parallel/easy-gpu-parallelism-with-openacc/240001776
  - → http://www.drdobbs.com/parallel/the-openacc-execution-model/240006334
- **Videos/ Webinars**
  - → http://developer.nvidia.com/gpu-computing-webinars
    - → OpenACC and the new PGI Accelerator
    - → Getting the most of OpenACC directives – Optimization with PGI Accelerator
  - → http://www.openacc.org/video
- **Code examples**
  - → Case Studies: http://www.openacc.org/Downloads

Intel Xeon Phi Links

- **Intel**
  - → Developer: http://software.intel.com/en-us/mic-developer
- **Book**
  - → J. Jeffers, J. Reinders, A. Sodani: Intel Xeon Phi Processor High Performance Programming (March 2016)


INTEL® XEON PHI™ PROCESSOR HIGH PERFORMANCE PROGRAMMING
KNIGHTS LANDING EDITION
Jim Jeffers | James Reinders | Avinash Sodani

RWTH Environment

- **GPU cluster**
  - → 28 nodes with each

    2 NVIDIA Quadro 6000 GPUs (Fermi)

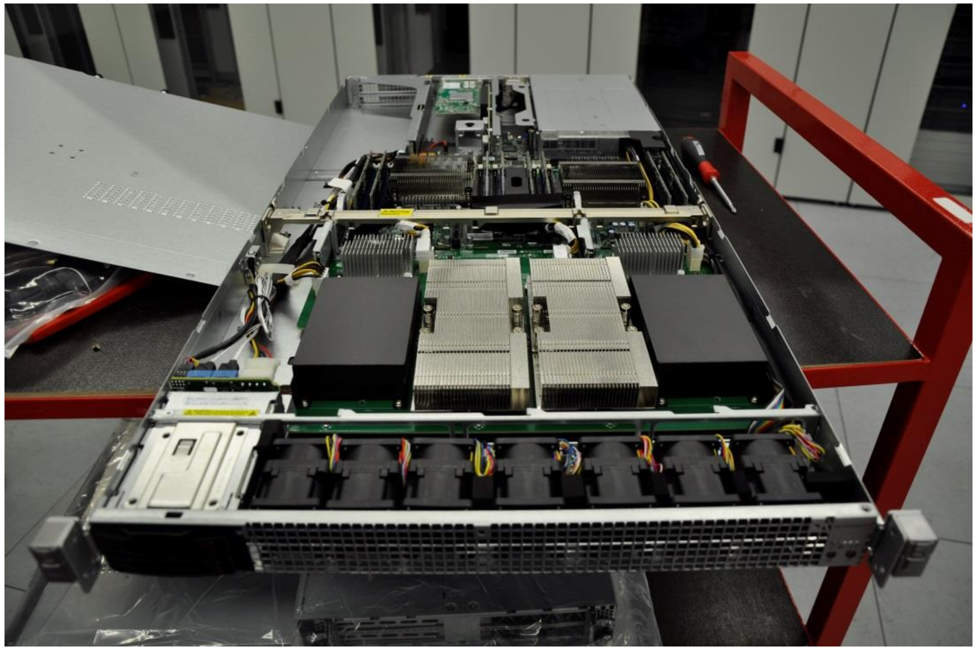
    → VR + HPC
  - → 2 nodes with each

    2 NVIDIA K20X GPUs (Kepler)
  - → 10 nodes with each

    2 NVIDIA P100 SXM2 (Pascal)

    with 16GB

aixCAVE, VR, RWTH Aachen, since June 2012



GPU Cluster - Hardware stack (Fermi)

		4 dialogue nodes	24 rendering nodes	1 head node
Name		linuxgpud[1-4]	linuxgpus[01-24]	linuxgpum1
Devices	#	2		1
	details/GPU	**NVIDIA Quadro 6000 (Fermi)** 448 cores 1.15 GHz 6 GB RAM ECC on max. GFlops: 1030.4 (SP), 515.2 (DP)		
Host	processor	**2 x Intel Xeon X5650 EP** (Westmere) (12-core CPU) @ 2.67GHz		
	RAM	24 GB		48 GB
Network		QDR InfiniBand		

GPU Cluster - Software stack

- **Environment as on compute cluster (modules,…)**

- **CUDA Toolkit: 7.5**
  - → CUDA
  - → OpenCL

```
module load cuda
```
→ directory: `$CUDA_ROOT`

- **PGI Compiler**
  - → CUDA Fortran
  - → PGI OpenACC

```
module load pgi
(module switch intel pgi)
```

- **CUDA Debugging**
  - → TotalView
  - → DDT
  - → Eclipse

```
module load totalview
module load ddt
```

GPU Cluster - How to use?

- **Doc: https://doc.itc.rwth-aachen.de/display/CC/GPU+cluster**
- **Interactive mode**
  - → Short runs/tests only, debugging
  - → 2 dialogue nodes (**`linuxgpud1, linuxnvc01`**): 24/7
  - → 2 dialogue nodes (**`linuxgpud[2,3]`**): Mon – Fri, 8am – 8pm
- **Batch mode**
  - → No interaction, commands are queued + scheduled
  - → For performance tests, long runs
  - → 24 rendering nodes
    4 dialogue nodes } Mon – Fri, 8pm – 8am; Sat + Son, whole day
  - → 2 dialogue nodes (**`linuxgpud4, linuxnvc02`**): Mon – Fri, 8am – 8pm
    → for short test runs during daytime
- **Note: reboot at switch from interactive to batch mode**

GPU Cluster - How to use: Interactive mode

- **Jump from frontend cluster node to GPU dialogue node:**
  `ssh –Y linuxgpud[1-3]`

- **GPUs are set to “exclusive mode” (per process)**
  - → Only one person can access GPU
  - → If occupied, e.g. message “all CUDA-capable devices are busy or unavailable”
  - → If not set a certain device in (CUDA) code, automatically scheduled to other GPU within node (if available)
- **Debugging**
  - → Be aware: debugger run usually on GPU with ID 0 (fails if GPU is occupied)
  - → Use CUDA_VISIBLE_DEVICES=1 to use other GPU

GPU Cluster - See what is running: nvidia-smi

`nvidia-smi -q`
Lists GPU details

```
linuxgpud1$> nvidia-smi
Mon Oct 17 12:41:01 2011
+------------------------------------------------------+
| NVIDIA-SMI 2.285.05   Driver Version: 285.05.09      |
|-------------------------------+-----------------------+----------------------+
| Nb.  Name                     | Bus Id                | Volatile ECC SB / DB |
| Fan   Temp   Power Usage /Cap | Memory Usage          | GPU Util. Compute M. |
|===============================+=======================+======================|
| 0.   Quadro 6000              | 0000:02:00.0  Off     |          0         0 |
|  30%   80 C  P0    Off /  Off |   4%   208MB / 5375MB |   99%     E. Process |
|-------------------------------+-----------------------+----------------------|
| 1.   Quadro 6000              | 0000:85:00.0  On      |          0         0 |
|  36%   84 C  P8    Off /  Off |   0%    22MB / 5375MB |    0%     E. Process |
|-------------------------------+-----------------------+----------------------|
| Compute processes:                                                           |
|  GPU  PID      Process name                                          Usage  |
|==============================================================================|
|  0.  30234    nbody                                                  196MB  |
+------------------------------------------------------------------------------+
```

ECC (SB: single bit, DB: double bit)

GPU ID + type

display mode

compute mode: 1 person (1 process)

process running on GPU

GPU Cluster - How to use: Batch mode

- **Create batch compute job for LSF**
- **Select appropriate queue to get scheduled on GPU-cluster**
  `-q gpu` (over night), `-a gpu` (over day)
- **Exclusive nodes**
  → Nodes are allocated exclusively → at least 2 GPUs for one job
  → Please use resources reasonably!
- **Submit your job**
  `bsub < myGPUScript.sh`
  → Starts running as soon as: batch mode starts and job is scheduled
- **Display pending reason: `bjobs -p`**
  → During daytime: **`Dispatch windows closed`**
- **More documentation**
  → RWTH Compute Cluster User‘s Guide: https://doc.itc.rwth-aachen.de/display/CC/Using+the+batch+system