

Using Openmp* Effectively on Theta

Carlos Rosales-Fernandez & Ronald W Green

ron.green@intel.com

Intel Corporation

2019 ALCF Computational Performance Workshop

Intel Legal Disclaimer & Optimization Notice

Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products. For more complete information visit www.intel.com/benchmarks.

INFORMATION IN THIS DOCUMENT IS PROVIDED "AS IS". NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. INTEL ASSUMES NO LIABILITY WHATSOEVER AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO THIS INFORMATION INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT.

Copyright © 2019, Intel Corporation. All rights reserved. Intel, the Intel logo, Pentium, Xeon, Core, VTune, OpenVINO, Cilk, are trademarks of Intel Corporation or its subsidiaries in the U.S. and other countries.

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

Optimization Notice

Copyright © 2019, Intel Corporation. All rights reserved.

*Other names and brands may be claimed as the property of others.



Overview

This talk is not intended to teach OpenMP*, but rather focus on using OpenMP Effectively on Theta

- Brief introduction OpenMP
- OpenMP Affinity
- [Optional based on time] Using OpenMP SIMD instructions
- [Optional based on time] OpenMP tasking

What is OpenMP*?

OpenMP stands for Open Multi-Processing. It provides:

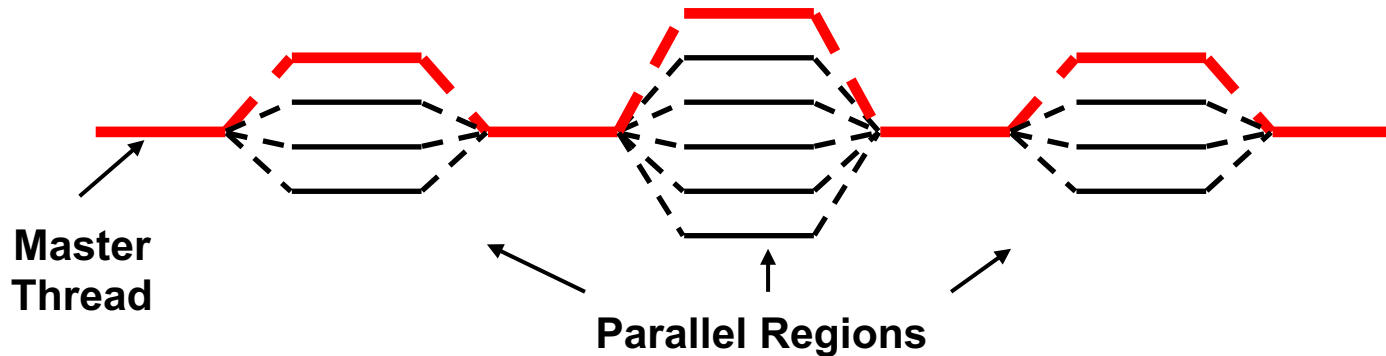
- Standardized directive-based multi-language high-level parallelism.
- Portable and Scalable model for shared-memory parallel programmers.
- Language support for C/C++/FORTRAN.
- Provides APIs and environment variables to control the execution of parallel regions.
- Latest specs and examples are available at <http://www.openmp.org/specifications/>.
- Supported by LLVM, Visual Studio Compiler, Intel Compiler, GNU GCC and others.

OpenMP* Programming Model

Real world applications are a mix of serial and inherently parallel regions.

OpenMP* provides **Fork-Join Parallelism** as a means to exploit inherent parallelism in an application within a **shared memory architecture**.

- Master thread executes in serial mode until a parallel construct is encountered.
- After the parallel region ends team threads synchronize and terminate, but master continues.



OpenMP Generalities

History of OpenMP starts in 1997/1998 with 1.0 Spec

Some very BROAD generalities

- 1.0 to 2.0 Parallel loops
- 3.0 added TASKS for more general parallel paradigms
- 4.0 added SIMD for controlling vectorization
- 4.0, 4.5, 5.0 OpenMP TARGET and MAP for accelerators

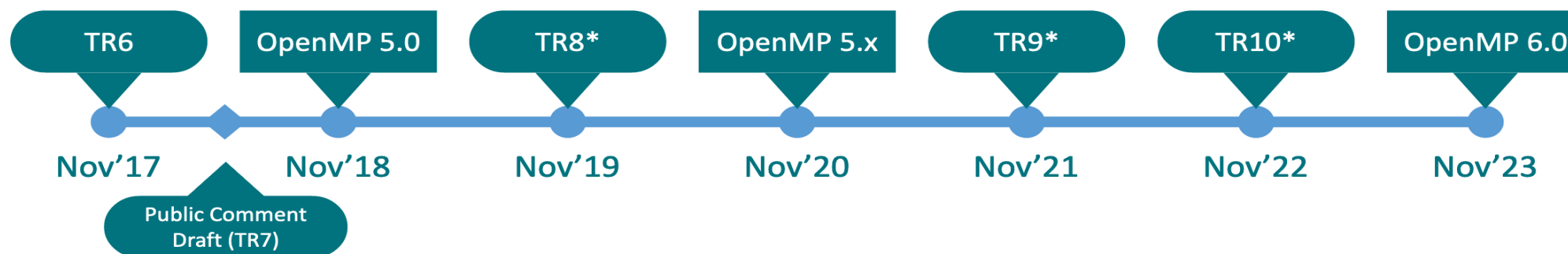
A true multi-tasker of a Specification

LOOPS/TASKS, VECTORIZATION, GPU Offload

OpenMP Roadmap

■ OpenMP has a well-defined roadmap:

- 5-year cadence for major releases
- One minor release in between
- (At least) one Technical Report (TR) with feature previews in every year



* Numbers assigned to TRs may change if additional TRs are released.

OpenMP* Constructs

Basic Components

Parallel - thread creation

- parallel

Work Sharing - work distribution among threads

- do, for, sections, single

Data Sharing - variable treatment in parallel regions and serial/parallel transitions

- shared, private

Synchronization - thread execution coordination

- critical, atomic, barrier

Advanced Functionality

- Tasking, SIMD, Affinity, Devices (offload)

Runtime functions and control

```
!$OMP PARALLEL
  !$OMP DO
    do i = 1, N
      a(i) = b(i) + c(i);
    end do
  !$OMP END PARALLEL
```

```
#pragma omp parallel
{
    #pragma omp for
    for(int i = 0; i < N; i++)
    {
        a[i] = b[i] + c[i];
    }
}
```


Coming SOON to Intel Compilers: OpenMP offload to Intel GPUs, FPGAs (and CPUs)

```
!$omp target teams distribute parallel do map(to: v1, v2) map(from: v3) device(1)
do i=1,N
    v2(i) = v2(i) * 2;    !..do we need the updated v2 values later? (from: v2, v3)
    v3(i) = v1(i) + v2(i);
end do
end subroutine vecadd
```

```
#pragma omp target data map (to: c[0:N], b[0:N]) map(tofrom: a[0:N])
#pragma omp target teams distribute parallel for
for (j=0; j<N; j++) {
    a[j] = b[j] + delta*c[j];
}
```



EFFICIENT USE OF THETA WITH OPENMP*

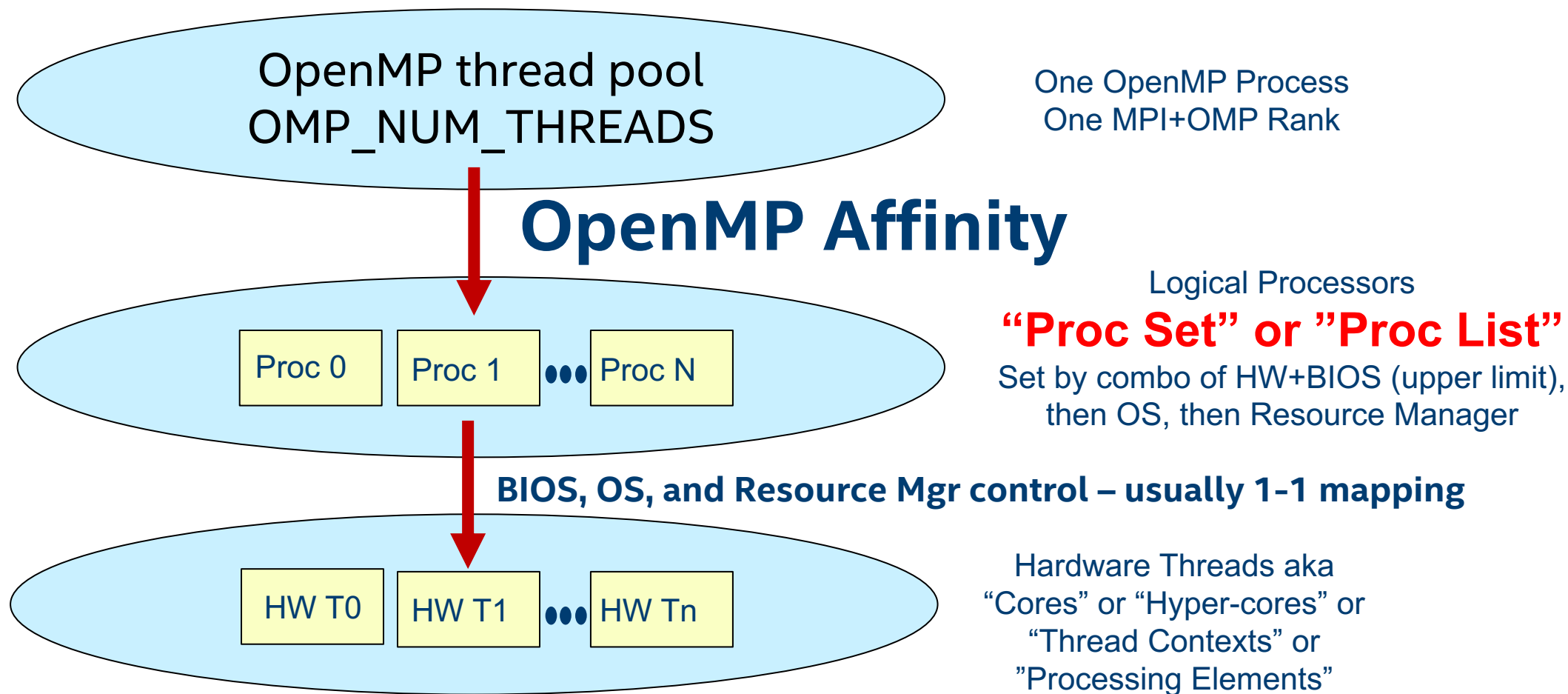
Affinity with OpenMP

Affinity – What is it, Why care?

- What: “Affinity” one of Processor affinity, memory affinity, cache affinity, even CPU Binding all describe the act of binding of processes or threads to a specific cpu or range of cpus to take advantage of hardware features: such as shared caches, close proximity to memory controllers, etc.
- Why?
 1. Binding – keep processes or threads from moving from 1 cpu to another
 - Cache example: data in L1 Dcache on a cpu: you lose it if thread moved
 - Allocate data on close mem controller, then thread gets migrated to another socket
 2. Ranks exchange boundary values with neighbors, want neighbor processes close (same tile, socket, node, or neighbor nodes on switch fabric)

OpenMP Binding

A mapping of OpenMP threads from "thread pool" to LOGICAL PROCESSORS



OpenMP PLACE: The Processor Set

- A "Node" has a number of "Packages or Sockets" each of which has processor chip.
 - Theta nodes have a single KNL / single-socket
- The processor chip has a fixed # of CORES
 - Theta has 64 KNL cores
- The CORE can be "hyper-threaded" or allow 1-N number of HW Threads
 - BIOS setting controls enable/disable
 - KNL cores support 4 HW threads = 256 HW Threads
- The OS maps 1 "Logical Processor" to each HW Thread
- Theta 256 Logical Processors
- Resource Manager carves out allocations for processes and can deliver a SUBSET of the available Proc List call "PROC SET" or "AFFINITY MASK"
 - Per node, max PROC SET is 256 logical processors
 - Actual AFFINITY MASK available to OMP is controlled by `aprun -cc, -j -d` arguments

Controlling and Binding to Proc Set

- Two Methods – pick one and stick with it.
 1. Use OpenMP Runtime to control placement of threads and binding
 - Advantage: Environment variables or API calls available on every vendor platform – portable (mostly – mappings change of course but concept and vars are the same.
 2. Use the vendor's Resource Manager and job launcher to control the PLACES List (Processor Set) and binding
 - Advantage: Sets limits that cannot be exceeded – allows sharing of nodes with multiple users. Keeps vendors and admins happy
 - Disadvantage: not portable, varies by resource manager and vendor – constantly have to re-learn new clusters.

Method 1: Use OpenMP to Control Affinity

For pure OpenMP* based codes the most effective way to set affinity is to disable affinity in aprun **-cc none** and then use OpenMP settings to bind threads.

Disabling affinity with aprun is simple:

```
$ aprun -n 1 -N 1 -cc none ./exe
```

Now threads can be pinned to specific hardware resources using the **OMP_PLACES** and **OMP_PROC_BIND** environmental variables.

Rich set of options with lots of flexibility and configuration granularity, but a few simple setups cover the vast majority of production cases.

Thread Affinity in OpenMP*

OpenMP* 4.0 introduces the concept of Places and Policies

- Set of threads running on one or more processors
- Places can be defined by the user
- Predefined places available: threads, cores, sockets
- Predefined policies : spread, close, master

And means to control these settings

- Environment variables `OMP_PLACES` and `OMP_PROC_BIND`
- Clause `proc_bind` for parallel regions

Optimal settings depend on application and workload

Pinning Step 1: OMP_PLACES

Two levels of granularity. You may specify a policy:

OMP_PLACES=<policy>

Where **policy** may be

- **sockets** : threads are allowed to float on sockets (multiple cores)
- **cores** : threads are allowed to float on cores (multiple logical processors)
- **threads** : threads are bound to specific logical processors

Or you may specify a list:

OMP_PLACES={lower_bound:length:stride}:repeat:increment

Pinning Step 2: OMP_PROC_BIND

`OMP_PROC_BIND=<policy>` Where `policy` must be chosen from:

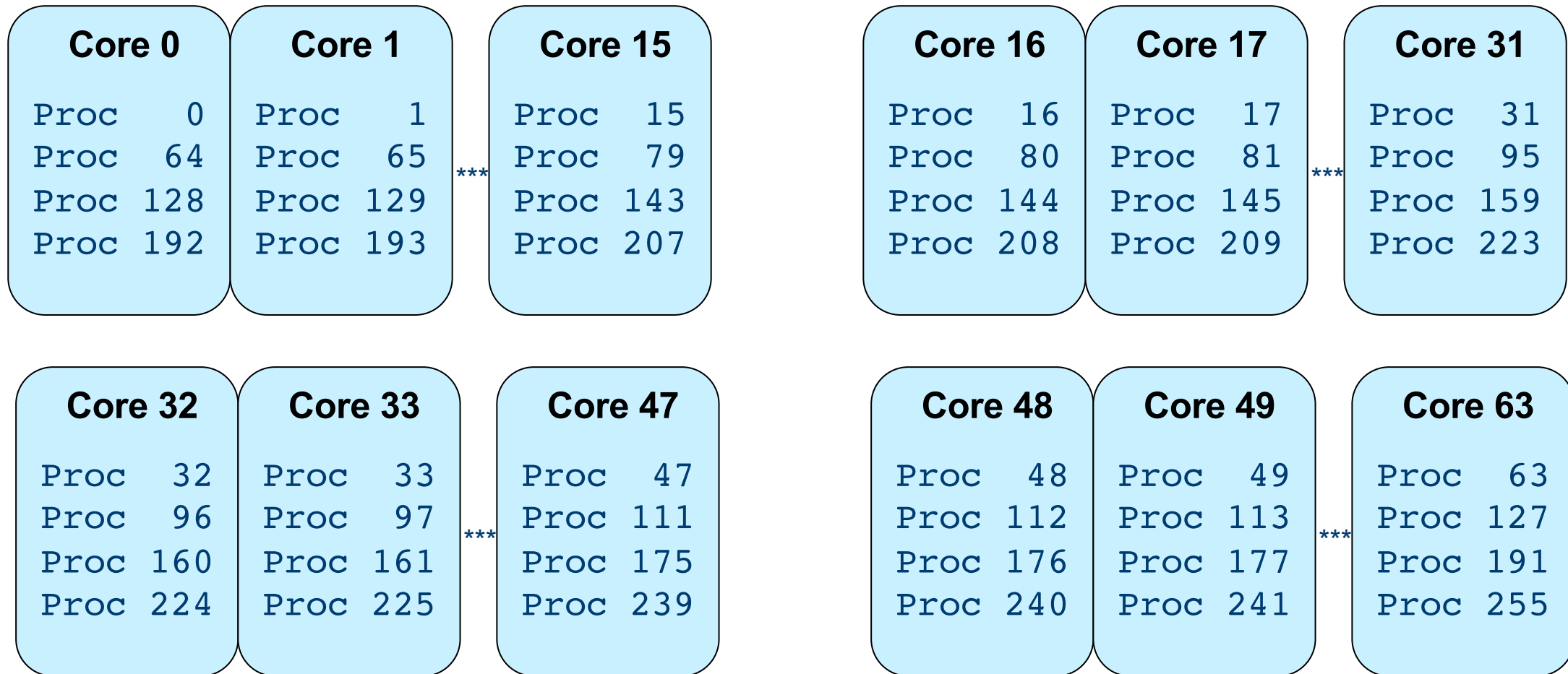
- **close** : threads placed consecutively, as near to the master place as possible
 - Best for possible cache reuse by nearby threads
- **spread** : threads spread equally on hardware to use most resources
 - Best for threads that do not benefit from shared caches
- **master** : threads placed on master place to enhance locality
 - Note: rarely if ever used, **master** can lead to heavy oversubscription of hardware resources, depending on the defined places.

CONFIRM WHAT YOU'RE GETTING!

`OMP_DISPLAY_AFFINITY=true` or (Intel) `KMP_AFFINITY=verbose`

`OMP_DISPLAY_ENV=true` #displays OMP related env vars

Logical Processor Mapping 64-core KNL Node



Some examples

```
OMP_NUM_THREADS=4; OMP_PLACES="{0:4:2}" #start, number, stride
```

Bound to [0] [2] [4] [6]

```
OMP_NUM_THREADS=4; OMP_PLACES=threads; OMP_PROC_BIND=close
```

Bound to [0] [64] [128] [192] – core 0 and its hyperthreads

```
OMP_NUM_THREADS=4; OMP_PLACES=threads; OMP_PROC_BIND=spread
```

Bound to [0] [16] [32] [48] – first core on each quadrant, no HT threads

```
OMP_NUM_THREADS=4; OMP_PLACES=cores; OMP_PROC_BIND=spread
```

Bound to [0,64,128,192] [16, 80, 144, 208] [32, 96, 160, 224] [48, 112, 176, 240]

Method 2: Use Theta APRUN to control thread binding and Proc Set. MPI example

```
aprun -n 4 -N 1 -cc depth -d 16 -j 1 -cc depth -e OMP_NUM_THREADS=16 <app> <app_args>
```

"-n 4" sets 4 MPI rank in total and "-N 1" places or 1 rank per node.

"-cc depth" use the depth arg and bind to that #HW threads

"-d 16" sets 16 hardware threads for each MPI rank

"-j 1" set 1 hardware thread per physical core.

"-e OMP_NUM_THREADS=16" sets the application to use 16 OpenMP threads per MPI rank

Want the Full Story on APRUN?

Attend our Hand—On coming very soon! Lots of examples
and of course ...

[https://www.alcf.anl.gov/user-guides/computational-systems#theta-\(xc40\)](https://www.alcf.anl.gov/user-guides/computational-systems#theta-(xc40))

but in PARTICULAR, this is one of the best pages on APRUN at any lab:

<https://www.alcf.anl.gov/user-guides/affinity-cray-xc40>

Using Hyperthreads – does it help or hurt?

Yes, well it depends

Hyperthreading – When it is Worthwhile?

- Where Hyperthreads **do not help**: if your code is FLOPS heavy, data demand is light and lots of cache reuse. Something like this:

```
for ( i=0, i< data-fits-in-cache ; i++ ){  
    a[i] = b[i] * func1(a[i]) - rhoPi*2.8754 ;  
    a[i] += b[i] / PI * (r*r) - (delta_t*accel);  
    b[i] = b[i] * MASS_DEN/4.00 + DELTA_RHO;  
} #just a few vars, stride 1, good # of flops in loop
```

**CPU
BOUND,
No HT**

- Where Hyperthreads **DO help** – latency on data, light flops/memory:

```
for ( i=0 , i< data-huge ; i++ ){  
    a[index[i]] = b[ index[i] ] + 1.0;  
} one flop, indirect memory access on A and B
```

**MEMORY
LATENCY
BOUND,
Use HT**



BUT the application is large, I inherited it, I don't know if it's CPU or memory bound! Or it varies by phase of the simulation or input. What do I do?



(is this your graduate advisor, PI, or sponsor? Hopefully NOT!)

Don't feel bad – it does vary by application; and by core counts, cpu architecture, memory architecture, and changes over time.

GENERALLY if you use memory indirection or pointers to mesh neighbor cells you PROBABLY have latency that can benefit from **MODERN** hypercores

Hyperthreading on Theta

- **Honest answer – WE HAVE NO CLUE if HT will benefit your code**
- Try binding one thread per core
- Try binding 2 threads per core, then 3, then 4
 - Side effects: you may change app behavior, as the threads usually determine the data decomposition
- GENERALLY on KNL systems like Theta
1 or 2 threads per core is USUALLY sufficient. UNLESS every memory access is indirect with little flops per memory fetch, low cache reuse, random memory access patterns – try 3 or 4

Can OpenMP Help?

- If threads stall on memory or other resources, what should the OpenMP Runtime system do?
1. Spin wait on CPU, assuming the stall is short (example, cache line not available immediately but fetch already in flight)
 - No HT, app CPU bound: Actively wait (spin) on the CPU until resource is freed (memory comes in, goes out, cache line is loaded, etc)
 - **OMP_WAIT_POLICY=ACTIVE**
 2. HT and long latency: Give up CPU, assume another thread has work and this stall will be long, like fetching `a[index[i]]`
 - **OMP_WAIT_POLICY=PASSIVE**

NUMA and Memory Considerations

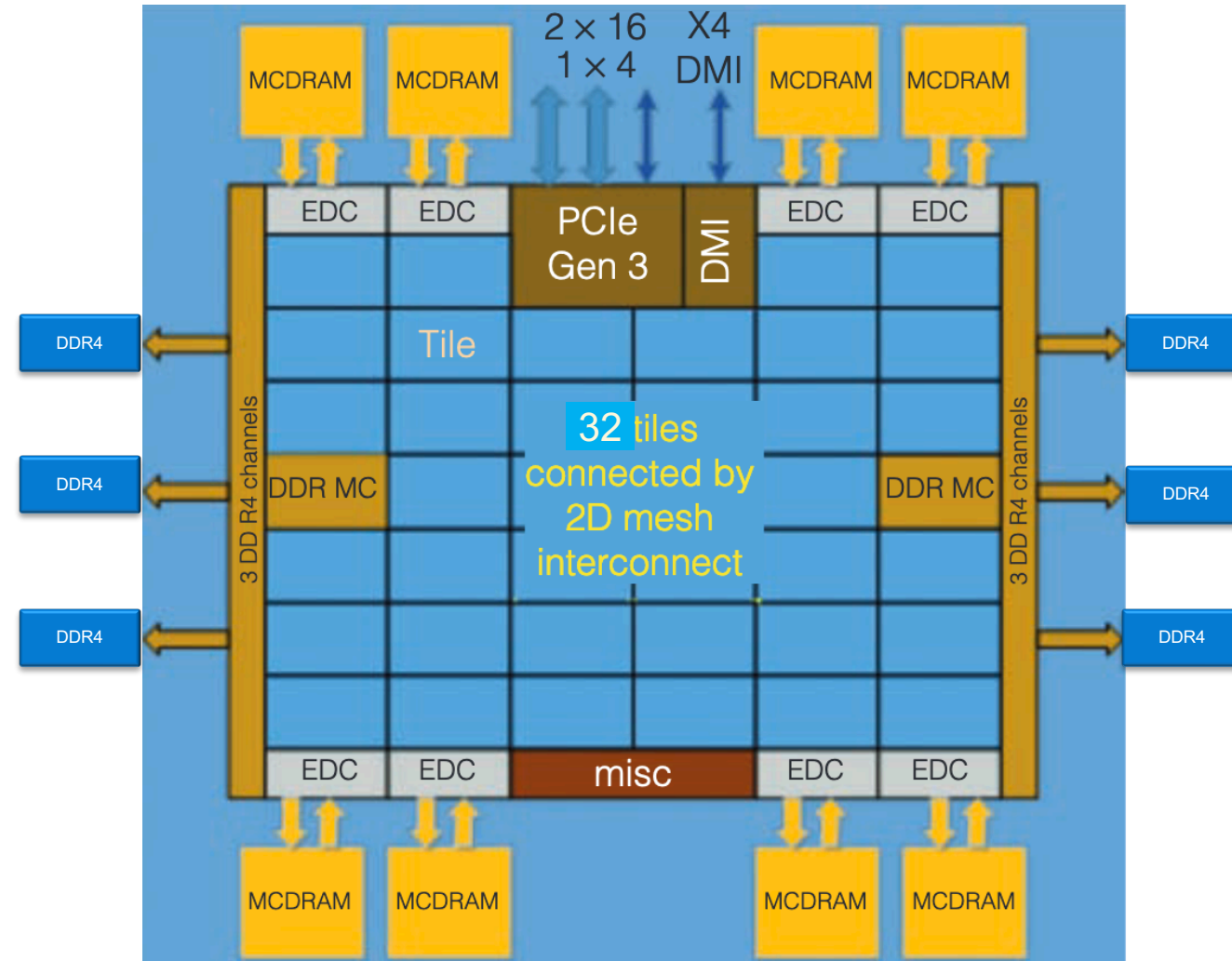
NUMA considerations

Locality – reduce memory latency.

- Use Linux first touch policy to your advantage by initializing data in an OpenMP* loop in the same way that it will be used later.
- C/C++ Do NOT use calloc() !!

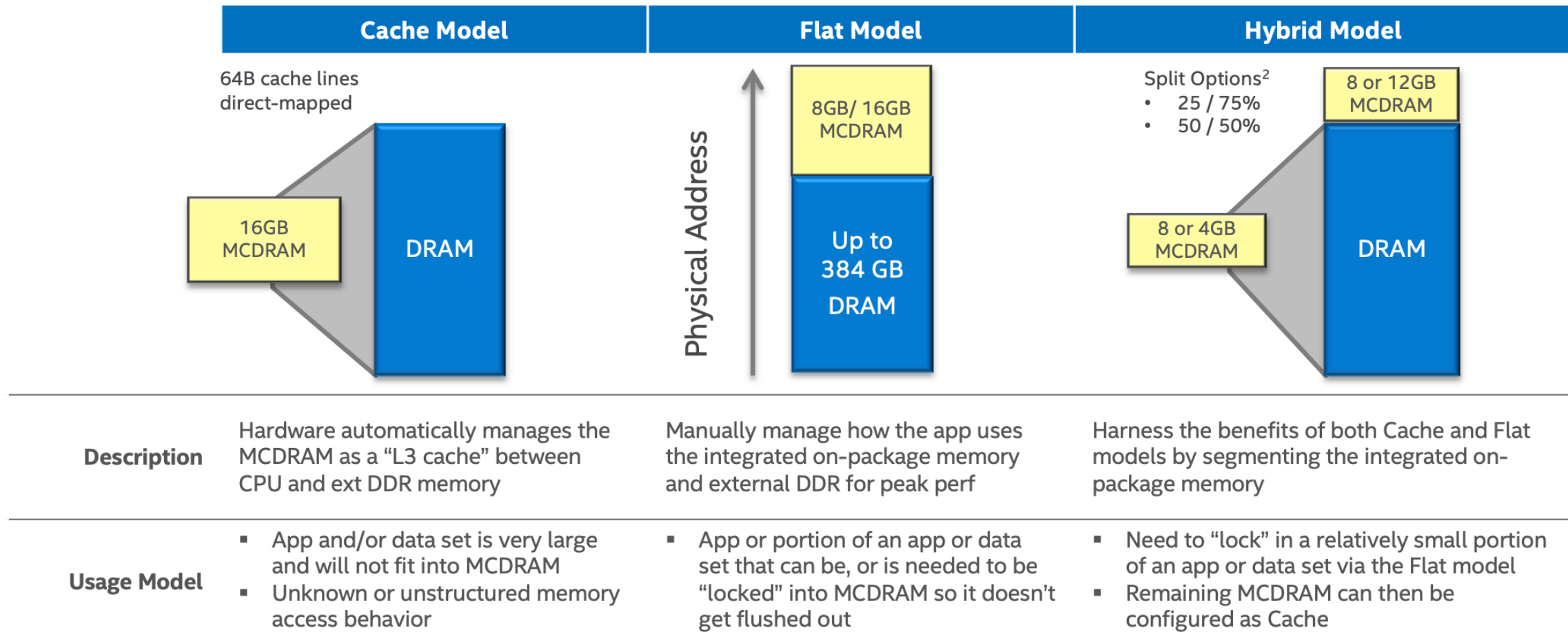
```
#pragma omp parallel for
  for(i=0; i<FLOPS_ARRAY_SIZE;
i++)
{
    fa[i] = (float)i + 0.1;
    fb[i] = (float)i + 0.2;
}
```

```
!$omp parallel do
do i = 1, FLOPS_ARRAY_SIZE
    fa(i) = i + 0.1
    fb(i) = i + 0.2
end do
!$omp end parallel do
```



Integrated On-Package Memory Usage Models

Model configurable at boot time and software exposed through NUMA¹



FLAT Model, Using MCDRAM the Easy Way

MCDRAM Provides higher bandwidth

- Important to make a conscious choice if running on flat mode

If running on flat mode you may use `numactl` to attach to the numa node 1 (MCDRAM) :

1. Allocations in MCDRAM first, spill into DRAM

```
aprun -n <ntot> -N <ppn> numactl --preferred=1 ./exe
```

2. Allocations in MCDRAM, abort if exceed space in MCDRAM

```
aprun -n <ntot> -N <ppn> numactl --membind=1 ./exe
```

3. API library "memkind" and "jemalloc" for more exact control

Using MCDRAM API

Flat MCDRAM SW Usage • Code Snippets

Allocate 1000 floats from DDR

```
float    *fv;  
fv = (float *)malloc(sizeof(float) * 1000);
```

Allocate 1000 floats from MCDRAM

```
float    *fv;  
fv = (float *)hbw_malloc(sizeof(float) * 1000);
```

Allocate arrays from MCDRAM & DDR in Intel FORTRAN

```
c    Declare arrays to be dynamic  
    REAL, ALLOCATABLE :: A(:), B(:), C(:)  
    !DIR$ ATTRIBUTES, FASTMEMORY :: A  
    NSIZE=1024  
  
c  
c    allocate array 'A' from MCDRAM  
c  
    ALLOCATE (A(1:NSIZE))  
  
c  
c    Allocate arrays that will come from DDR  
c  
    ALLOCATE (B(NSIZE), C(NSIZE))
```

Keeping the App Effort Level Low

Optimization Notice

Copyright © 2019, Intel Corporation. All rights reserved.
*Other names and brands may be claimed as the property of others.

High Bandwidth (HBW) malloc API

HBWMALLOC(3)

HBWMALLOC

HBWMALLOC(3)

NAME

`hbwmalloc` - The high bandwidth memory interface

SYNOPSIS

```
#include <hbwmalloc.h>
```

Link with `-ljemalloc -lnuma -lmemkind -lpthread`

```
int hbw_check_available(void);
void* hbw_malloc(size_t size);
void* hbw_calloc(size_t nmemb, size_t size);
void* hbw_realloc(void *ptr, size_t size);
void hbw_free(void *ptr);
int hbw_posix_memalign(void **memptr, size_t alignment, size_t size);
int hbw_posix_memalign_psize(void **memptr, size_t alignment, size_t size, int pagesize);
int hbw_get_policy(void);
void hbw_set_policy(int mode);
```



DATA ALIGNMENT

- 1) Force data alignment on cache boundaries
- 2) TELL the compiler the data is aligned at use site

Essential Step #1: Align Data, Fortran

Align the data at allocation ...

-align array64byte

compiler option to align all array types

Works for dynamic, automatic and static arrays (not in COMMON)

Align COMMON array on an “n”-byte boundary (n must be a power of 2)

!dir\$ attributes align:n :: array

n=64 MIC & AVX-512, n=32 for AVX AVX2, n=16 for older SSE
This equals L1 cache line size

Step: Fortran: Tell the compiler data is aligned

And tell the compiler WHERE you use the data ...

Prefer OpenMP SIMD directive:

```
!$omp SIMD aligned(list[:n])
```

Compiler can assume (list) arrays are aligned to n byte boundary

Old Intel-only directive. `!dir$ vector aligned`

Asks compiler to vectorize, overriding cost model, and assuming all array data accessed in loop are aligned for targeted processor

- Haswell/Broadwell and newer may only generate unaligned loads/stores
 - If data is aligned, unaligned load/store just as fast
 - NOT true on older Xeons

Align Data C++

`__declspec(align(n, [offset])) __attribute__((aligned(n)))`

Instructs the compiler to create the variable so that it is aligned on an “n”-byte boundary, with an “offset” (Default=0) in bytes from that boundary

```
struct S { short f[3]; } __attribute__((aligned (64)));
```

```
typedef int more_aligned_int __attribute__((aligned (64)));
```

`void* __mm_malloc (int size, int n)`

Replacement for malloc(), instructs the compiler to create a pointer to memory such that the pointer is aligned on an n-byte boundary

OR use a memory allocator that can control alignment (such as in TBB)

and tell the compiler...

`#pragma vector aligned | unaligned`

Vectorize using aligned or unaligned loads and stores for vector accesses, overriding compiler's cost model

`__assume_aligned(a,n)`

Instructs the compiler to assume that array a is aligned on an n-byte boundary

n=16 for SSE, n=32 for AVX, n=64 AVX512

MPI + OpenMP

Hybrid MPI + OpenMP*

When using hybrid applications aprun must be configured to create pinning ranges for each MPI task, and then OpenMP variables may be set to control thread pinning within each rank processor range. Example: 4 MPI tasks, 16 threads each, over 8 nodes

```
export OMP_NUM_THREADS=16
export OMP_PLACES=cores;
export OMP_PROC_BIND=spread
aprun -n 32 -N 4 -cc depth -d 64 -j 4 ./exe
```

MPI Rank	Thread 0	Thread 1	...	Thread 15
Rank 0	[0, 64, 128, 192]	[1, 65, 129, 193]	...	[15, 79, 143, 207]
Rank 1	[16, 80, 144, 208]	[17, 81, 145, 209]	...	[31, 95, 159, 223]
Rank 2	[32, 96, 160, 224]	[33, 97, 161, 225]	...	[47, 111, 175, 239]
Rank 3	[48, 112, 176, 240]	[49, 113, 177, 241]	...	[63, 127, 191, 255]

Recommended settings for Theta

The following setup is recommended for jobs using up to 4 threads per core

OMP_PLACES=cores

OMP_PROC_BIND=spread

aprun -n <totalTasks> -N <tasksPerNode> -cc depth -d 256/<tasksPerNode> -j 4

If using multiple threads per core you may want to test the effect of changing the default wait policy to passive:

OMP_WAIT_POLICY=passive

Cluster Mode Considerations

"Cluster Mode" describes how page tables and cache coherency managed.

Three on-die cluster modes supported:

- all-to-all: 1 master table)
- Quadrant (QUAD): Default mode, better performance, tables distributed over 4 tables with global coordination
- SNC-[4 | 2] (Sub-NUMA Clustering) 4 tables but no coordination – KNL looks like 4 nodes in a sub-cluster RARELY USED IN PRACTICE
- WHY DO YOU CARE?
 - On a QUAD node you get better performance with 1 MPI rank and threads spread evenly into each quadrant "quadrants" [0..15] [16..31] [32..47] [48..63]etc OR
 - 4 MPI ranks OR multiples of 4 placed in the quads



VECTORIZATION WITH OPENMP* SIMD

Vectorization and SIMD Overview

SIMD – Single Instruction Multiple Data

A class of parallel computers in [Flynn's taxonomy](#).

Systems with multiple processing elements (Pes) doing the same operation on multiple data points simultaneously.

A form of [data level parallelism](#): A single operation (instruction) directs the PEs to do the same operation on multiple data elements (vector or array) simultaneously.

Vectorization:

A form of SIMD parallel programming

Generally refers to transformation of a serial program or application to exploit SIMD hardware resources

Although often used in the context of compiler techniques to “auto-vectorize” serial code to exploit SIMD hardware, Vectorization is any programming method to exploit SIMD resources

Vectorization is Achieved through SIMD Instructions & Hardware

Intel® AVX-512 Announced in 2014

Vector size: 512bit

Data types: 32 and 64 bit floats

Other data types vary by part
(KNL 32 and 64bit Ints)

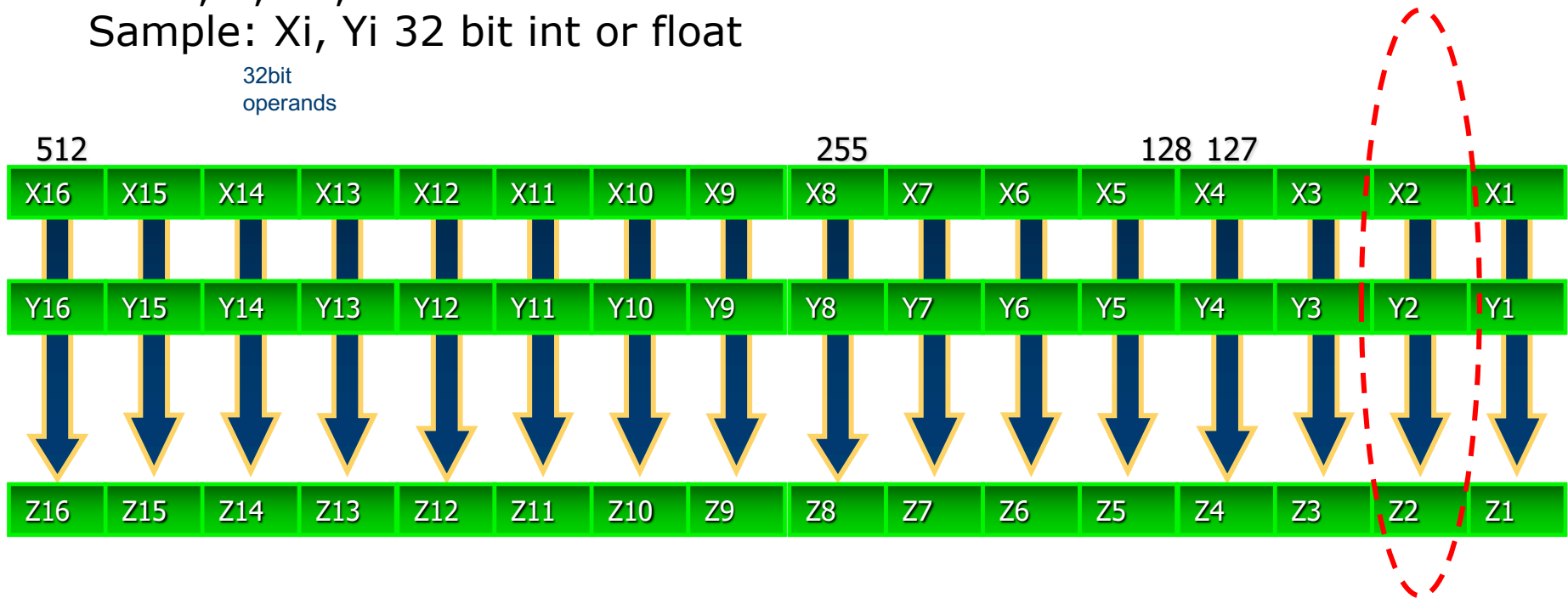
VL: 4, 8, 16, 32

Sample: Xi, Yi 32 bit int or float

Terminology:

SIMD “LANE” is a parallel slice of elements used in a vector operation .

Think of this as similar to a multi-lane freeway

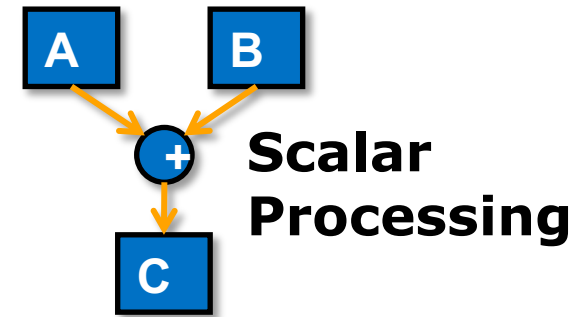


Terminology SIMD & Vectorization

Single Instruction Multiple Data (SIMD):

Hardware supported technique which allows an operation to be performed on multiple data points simultaneously.

Provides data level parallelism (DLP) which is more efficient than scalar processing



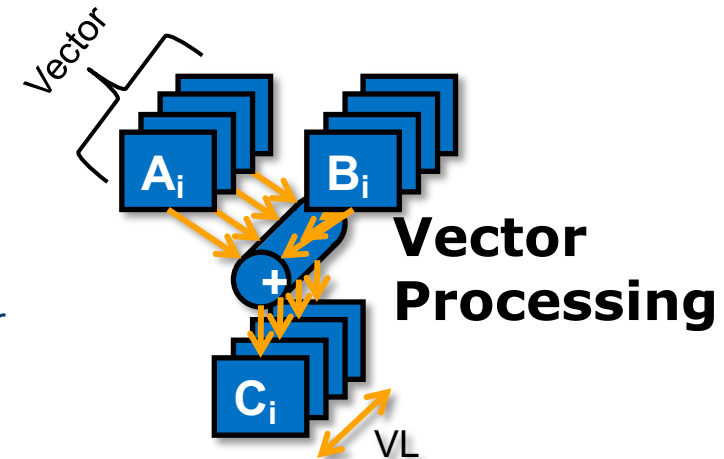
Vector:

Consists of more than one element

Elements are of same scalar data types
(e.g. floats, integers, ...)

Vector length (VL):

Number of elements of the vector which are processed together



Vectorization

Process which converts procedural loops that iterate over multiple pairs of data items and assigns a separate processing unit to each pair

Exploiting SIMD

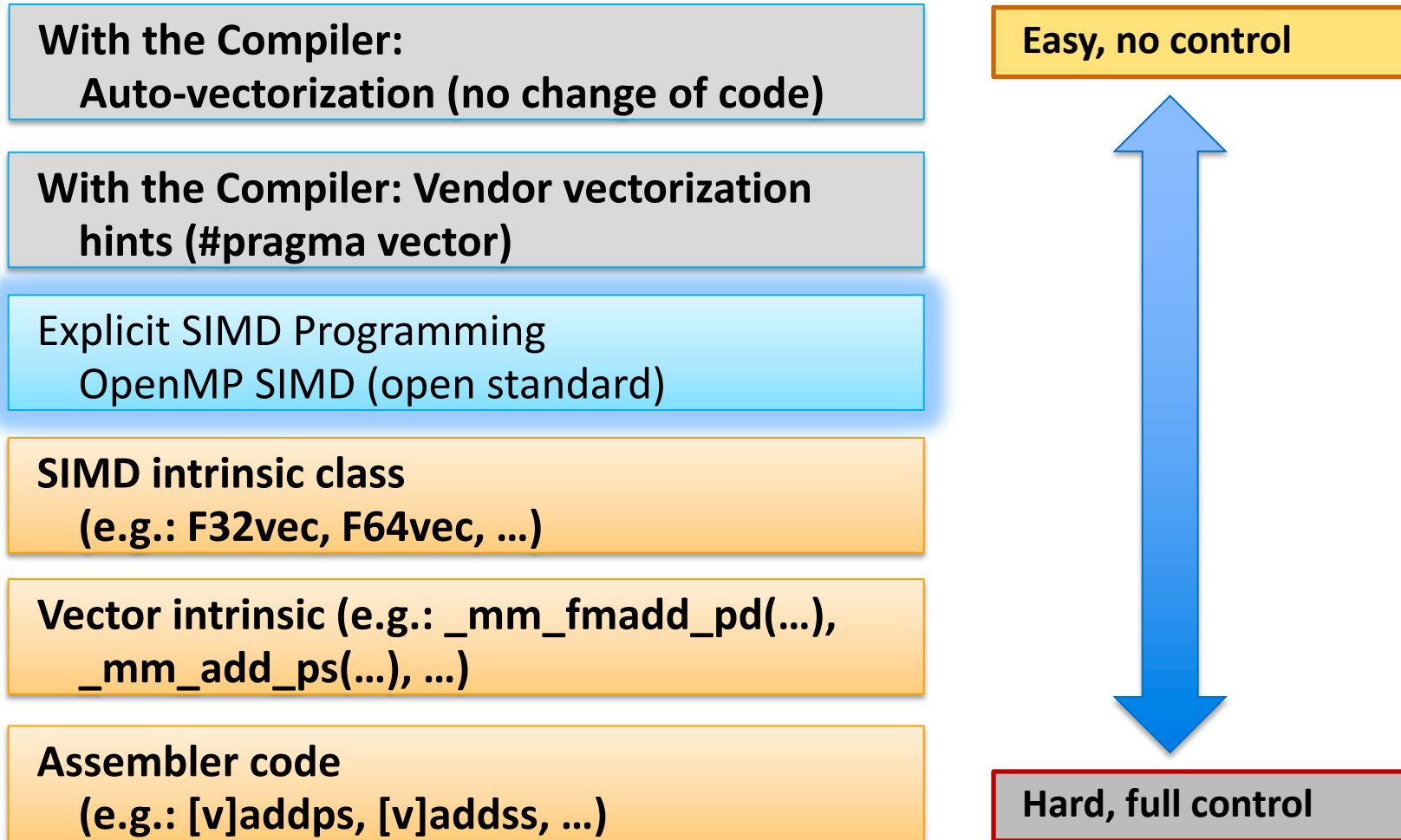
Optimization Notice

Copyright © 2019, Intel Corporation. All rights reserved.

*Other names and brands may be claimed as the property of others.



Ways to Vectorize Code



Auto-Vectorization

SIMD – Single Instruction Multiple Data

```
for (i=0; i<=MAX; i++)  
    c[i]=a[i]+b[i];
```

```
do i=1, MAX  
    c(i)=a(i)+b(i)  
end do
```

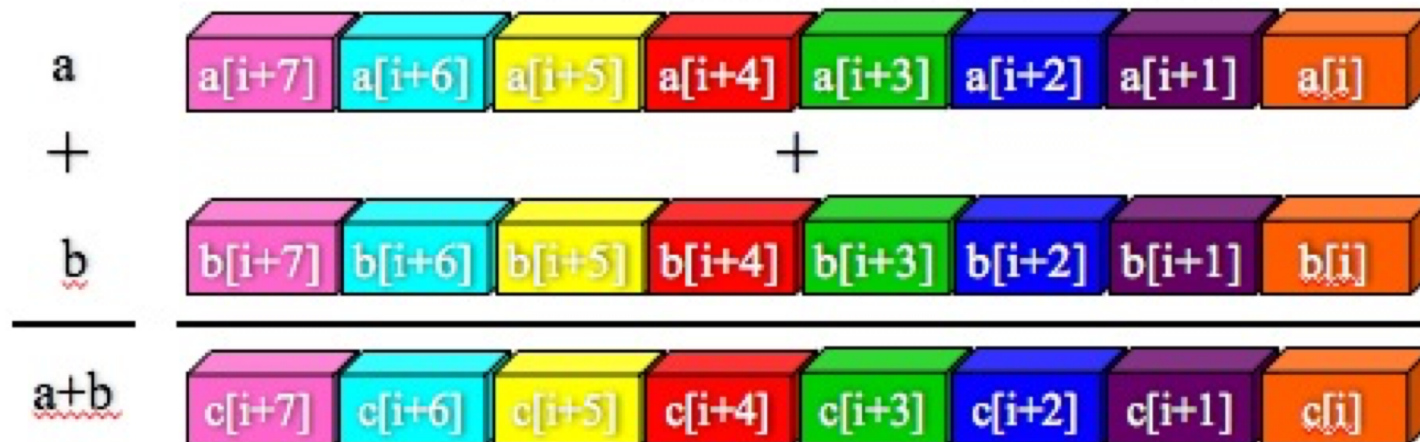
- **Scalar mode**

- one instruction produces one result



- **SIMD processing**

- with SSE or AVX instructions
- one instruction can produce multiple results



Auto-Vectorization

```
for (i=0; i<=MAX; i++)
```

```
    c[i] = a[i] + b[i];
```

...compiler rewrites as

```
for (i=0; i<=MAX; i+=8) {
```

```
    c[i] = a[i] + b[i];
```

```
    c[i+1]=a[i+1]+ b[i+1];
```

```
    c[i+2]=a[i+2]+ b[i+2];
```

```
    c[i+3]=a[i+3]+ b[i+3];
```

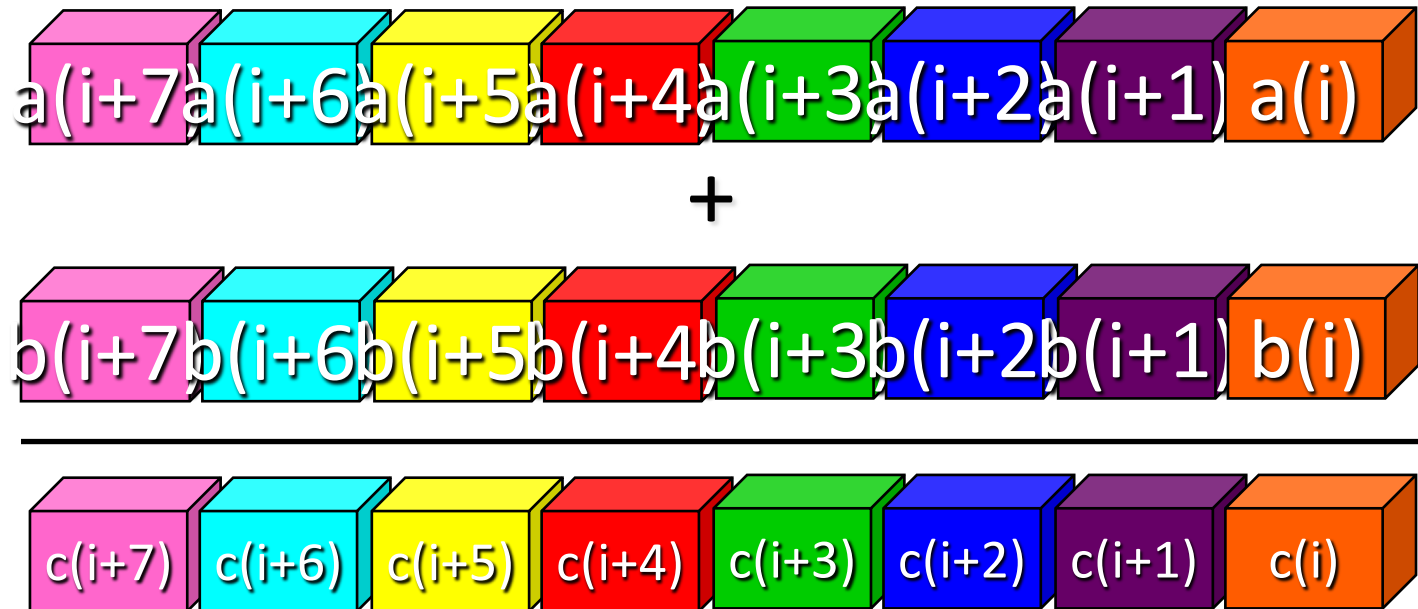
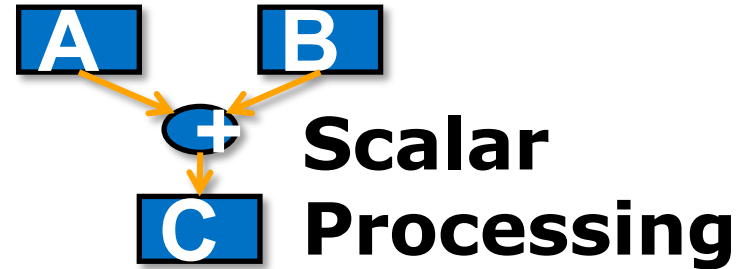
```
    c[i+4]=a[i+4]+ b[i+4];
```

```
    c[i+5]=a[i+5]+ b[i+5];
```

```
    c[i+6]=a[i+6]+ b[i+6];
```

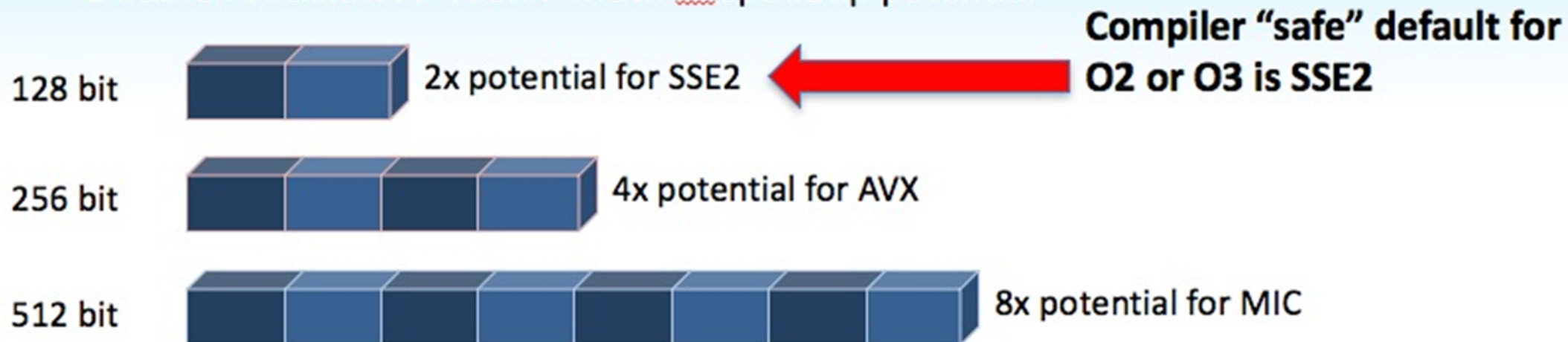
```
    c[i+7]=a[i+7]+ b[i+7];
```

```
}
```



Vectorization - Potential for Performance

Double Precision FP vector width vs speedup potential



Notes: Default at O2 or O3 is SSE3 – Compiler HAS TO ASSUME you may want to run this code on some ancient computer. But you can do better!

Wider vectors allow for higher potential performance gains
Gains of 4X and 8X within reach using vectorization capability
Gains of 8x to 16x in single precision floats

Legacy Processor-specific Compiler Switches

Intel processors only	Intel and non-Intel (-m also GCC)
-xsse2	-msse2 (default)
-xsse3	-msse3
-xssse3	-mssse3
-xsse4.1	-msse4.1
-xsse4.2	-msse4.2
-xavx -xcore-avx2	-mavx
-xmich-avx512	-march=knl
-xHost	-xHost (-march=native)
Intel cpuid check	No cpu id check
Runtime message if run on unsupported processor	Illegal instruction error if run on unsupported processor

Modern Processor-specific Compiler Switches

Intel processors only	Intel and non-Intel (-m also GCC)
-xcommon-avx512*	
-xcore-avx512*	
-xskylake*	-mskylake
-xskylake-avx512*	-mskylake-avx512
-xicelake-client	-micelake-client
-xicelake-server	-micelake-server
-xcascadelake	-mcascade-lake
-xHost	-xHost (-march=native)
Intel cpuid check	No cpu id check
Runtime message if run on unsupported processor	Illegal instruction error if run on unsupported processor

Easier Naming in Recent Compilers

- To simplify (maybe), we added codenames to `-x` and `-ax` options. Version 18.0 compilers and newer. `-x<name> -ax<name>` where `<name>`:
 - **Skylake and AVX512** options can be confusing.
 - “common-avx512” is the base set of AVX512 instructions.
 - “core-avx512” extends “common” with additional instruction sets for server grade processors.
 - “skylake” is client version of Skylake, doesn’t have AVX512
 - **For Skylake servers, answer is “-xskylake-avx512 -qopt-zmm-usage=high” for highest possible use of AVX512.**

BROADWELL

CANNONLAKE

HASWELL

ICELAKE-CLIENT (or ICELAKE)

ICELAKE-SERVER

IVYBRIDGE

KNL

KNM

SANDYBRIDGE

SILVERMONT

SKYLAKE

SKYLAKE-AVX512

MIC-AVX512

MIC-AVX512:

- May generate Intel(R) Advanced Vector Extensions 512 (Intel(R) AVX-512) Foundation instructions
- Intel(R) AVX-512 Conflict Detection instructions
- Intel(R) AVX-512 Exponential and Reciprocal instructions
- Intel(R) AVX-512 Prefetch instructions for Intel(R) processors
- and the instructions enabled with CORE-AVX2.

KEY CONCEPTS or TAKE-AWAY

- Any of `-x`, `-a`, `-march` are **SUGGESTIONS** to compiler on MAXIMUM instruction set it **MAY USE**
 - Compiler may use lesser vectorization if it could be more efficient
 - You can use either the Vector Extension name OR processor name
 - `-xsandybridge` is same as `-xavx`
 - `-xhaswell` is same as `-xcore-avx2`
 - `-xskylake-avx512` will often favor AVX2. For max AVX512 usage, use with:
 - **`-xskylake-avx512 -qopt-zmm-usage=high`**
 - `-xicelake-server` will favor AVX512 over AVX, no need for additional option like Skylake needs
 - “skylake” is for CLIENT processors which DO NOT have AVX512!

Auto-Vectorization Intel Options

{Linux &Mac} **-x<extension>** {Windows}: **/Qx<extension>**

Targeting Intel® processors - specific optimizations for Intel® processors

Compiler will try to make use of all instruction set extensions up to and including <extension>; for Intel® processors only !

Processor-check added to main-program

Application will not start (will display message), in case feature is not available

{L&M}: **-m<extension>** {Windows}: **/arch:<extension>**

No Intel processor check, gnu compatibility only

Does not perform Intel-specific optimizations and ***not as aggressive as -x or -ax***

Application is optimized for and will run on both Intel and non-Intel processors

Missing check can cause application to fail if instructions not supported

{L&M}: **-ax<extension>** {Windows}: **/Qax<extension>**

Multiple code paths – a 'baseline' and 'optimized, processor-specific' path(s)

Optimized code path for Intel® processors defined by <extension>

Baseline code path defaults to -msse2 (Windows: /arch:sse2)

- The baseline code path can be modified by -m or -x (/Qx or /arch) switches

-axavx,sse4.2 - paths for avx, sse4.2 and sse2 (implicit default)

-axmic-avx512 -xcore-avx2 (KNL path and default AVX2 path, no SSE2 path)

Use -x or -ax for best vectorization.

-axmic-avx512 -xavx

or

-axmic-avx512 -xcore-avx2

are good choices for KNL and modern Xeon blended code

or

-xhost

“host” cannot be used with -ax unfortunately

Non-Intel Compatible Processors

- Use `–march` compiler option for targeting
- No “codenames” for non-Intel processors. You have to know if it uses AVX2, SSE3, etc and use those codes OR the equivalent Intel names
- With 2019 (19.0) Update 4 and older compilers you can only target 1 architecture with `-march`
 - Currently no functionality like `–ax` option for non-Intel
 - With future compilers released after 2019 Update 4 we add a new option:

Windows syntax: `/Qauto-arch:<val>`

Linux syntax: `-mauto-arch=<val>`

‘val’ can be anything that is accepted by `/arch` on Windows or `–march` on Linux.

Similar as to `ax`, `Qax` this option tells the compiler to generate multiple, feature-specific auto-dispatch code paths for x86 architecture processors if there is a performance benefit. It also generates a baseline code path. Baseline code path used depends on OS and platform. See the `–ax` option for the default baseline codepath used.

Vectorization – More Switches and Directives

Disable vectorization

Globally via switch: {L&M}: **-no-vec** {W}: **/Qvec-**

For a single loop: directive **!DIR\$ novector**

Disabling vectorization means not using **packed SSE/AVX** instructions.

The compiler still may use SSE instruction set extensions

Enforcing vectorization for a loop - overwriting the compiler heuristics :

!dir\$ vector always

will enforce vectorization even if the compiler thinks it is not profitable to do so (e.g due to non-unit strides or alignment issues)

It's a SUGGESTION: Will not enforce vectorization if the compiler fails to recognize this as a semantically correct transformation – hint to compiler to relax

!dir\$ vector always assert

will print error message in case the loop cannot be vectorized and will abort compilation

Auto-vectorization

Terrific! Use the right compiler options and we're done, right?

Easy: Auto-Vectorization, Let the Compiler do the Vectorization

Auto-Vectorization relies on

- NO Dependencies in data between loop iterations
 - or dependencies that can be removed by simple methods
- Regular, predictable data patterns for all operands
 - No pointer chasing, indirect accesses
- Vector lengths that are large enough AND
- Data is aligned on natural boundaries (16, 32, or 64 bytes)
 - cache line matching boundary, can use cache as staging
- Streaming stores – store optimization

COST MODEL – is it worth it to vectorize???

Advanced optimizations help with some of these

Vectorizing DOES NOT IMPLY PERFORMANCE!
Vectorized code sometimes runs slower than non-vectorized code.
AVX2 can run faster than AVX512

Speed step (power), indirect addressing, indexed (strided) addressing, and pointer chasing are examples. Low trip counts with peel and remainder loops also

<https://software.intel.com/en-us/articles/requirements-for-vectorizable-loops/>

Requirements for Auto-Vectorization

Target: Innermost loops

Collapse to outermost IF perfectly-nested loop(s)

AVOID

Loop-carried dependencies

No early loop exits, no branches into loop (no throw/catch)

Data-dependent loop exit conditions

User Function/subroutine calls

Non-contiguous and poorly aligned data

Inefficient memory accesses often tagged 'not profitable'

POINTERS (Fortran too!) are pure evil, use sparingly

There are many, many reasons the compiler will refuse to vectorize a loop

Job #1 of a compiler is to create CORRECT ie SAFE code

Performance is secondary

Compilers will always assume the worse case scenario of your code

KNOW what your compiler is DOING,
and WHY – Optimization Reports!!!

When Does the Compiler Auto-Vectorize a Loop?

Intel ifort/icc/icpc

By default, at -O2 and above

Can disable by -no-vec

Get report with -qopt-report[0-5] (start with -qopt-report5)

GCC

At -O3

With -O2 -ftree-vectorize

Get report with -ftree-vectorizer-verbose=[0-7] (gcc ≤ 4.8) -fopt-info-vec (gcc ≥ 4.9)

Cray

-h vector[0-3] 0=only F90 array syntax, 1=conservative, no loop xforms
2=moderate: loop restructuring (Default for Cray -O2+)
3=aggressive

Get report with: ftn -rm, CC -hlist=m

Did my loop vectorize?

Simplest:

```
ifort -c -qopt-report5 ../../snb/addit.f90
```

```
addit.f90(17): (col. 2) remark: LOOP WAS VECTORIZED.
```

You can also recognize Intel® SSE registers and packed instructions in the assembly code: (...pd for doubles, ...ps for floats)

```
ifort -S addit.f90; grep addpd addit.s
```

```
addpd    (%rsi,%rax,8), %xmm0          (Intel® SSE)
vaddpd    (%rsi,%rax,8), %xmm0, %xmm1    (Intel® AVX -128)
vaddpd    (%rsi,%rax,8), %ymm0, %ymm1    (Intel AVX -256)
```

Or calls to the Short Vector Math Library (libsvml):

```
call  __svml_sin2    (Intel SSE)      (2 doubles per call)
call  __svml_sin4    (Intel AVX)      (4 doubles per call)
call  __svml_exp8     (Intel AVX)     (8 floats per call)
```

Did it Vectorize? Optimization Report

Applicable to Intel® Compiler version 15.0 and newer

- for C, C++ and Fortran
- for Windows*, Linux* and OS X*

(For readability, options may not be repeated for each OS where spellings are similar. Options apply to all three OS unless otherwise stated.)

Main options (there are a lot of qopt-report-* options):

-qopt-report[=N] (Linux and OS X)

/Qopt-report[:N] (Windows)

N = 1-5 for increasing levels of detail, (default N=2)

-qopt-report-phase=str[,str1,...]

str = loop, par, vec, openmp, ipo, pgo, cg, offload, tcollect, all

-qopt-report-file=[stdout | stderr | filename]

Vectorization – report levels

`[-q|/Q]opt-report-phase=vec [-q|/Q]opt-report=N`

N specifies the level of detail; default N=2 if N omitted

Level 0: No vectorization report

Level 1: Reports when vectorization has occurred.

Level 2: Adds diagnostics why vectorization did not occur.

Level 3: Adds vectorization loop summary diagnostics.

Level 4: Additional detail, e.g. on data alignment

Level 5: Adds detailed data dependency information

Report Output

Output goes to a text file by default

- File extension is .optrpt, root name same as object file's
- One report file per object file, in object directory
- created from scratch or overwritten (no appending)

`[-q | /Q]opt-report-file:stderr` gives to stderr

`:filename` to change default file name

`/Qopt-report-format:vs` format for Visual Studio* IDE

For debug builds, (`-g` on Linux* or OS X*, `/Zi` on Windows*),
assembly code and object files contain loop optimization info

- `/Qopt-report-embed` to enable this for non-debug builds



CONTROLLING VECTORIZATION

Guiding the compiler with directives

Controlling, Guiding Vectorization

Vendors provide a hodge-podge, vendor-specific pragmas and Fortran directives to guide vectorization

No standards, no cooperation between vendors (IVDEP is an exception)

OpenMP 4 Standardized a set of SIMD directives

Based on OMP Parallel directives

Intel Compiler Provided Directives

!DIR\$ directives

- IVDEP ignore vector dependency
- LOOP COUNT advise typical iteration count(s)
- UNROLL suggest loop unroll factor
- DISTRIBUTE POINT advise where to split loop
- VECTOR vectorization hints
 - Aligned assume data is aligned
 - Always override cost model
 - Nontemporal advise use of streaming stores
- NOVECTOR do not vectorize
- NOFUSION do not fuse loops
- INLINE/FORCEINLINE invite/require function inlining
- SIMD explicit vector programming (see later)

Use where needed to help the compiler.

Remember, these are Intel-compiler specific!

We recommend the newer OpenMP 4 SIMD directives for many of these

Explicit or Guided Vector Programming with OpenMP 4.0 and above

Modeled on OpenMP for threading (explicit parallel programming)

Enables reliable vectorization of complex loops that the compiler can't auto-vectorize

E.g. Possible dependency, inefficient, reduction variables

Directives are commands to the compiler, not hints

Programmer is responsible for correctness (like OpenMP threading)

E.g. PRIVATE and REDUCTION clauses

Incorporated in OpenMP 4.0 \Rightarrow portable

Intel `-qopenmp` or `-qopenmp-simd` to enable

GCC support in version 4.9.0+ (C/C++), gfortran 4.9.1

Needs `-fopenmp` or `-fopenmp-simd`

OpenMP Compiler Options

Intel

`-qopenmp`

Recognizes OpenMP Parallel and SIMD directives

`-qopenmp-parallel`

Recognizes OpenMP Parallel directives

Ignores OpenMP SIMD directives

`-qopenmp-simd`

Recognizes OpenMP SIMD directives

Ignores OpenMP Parallel directives

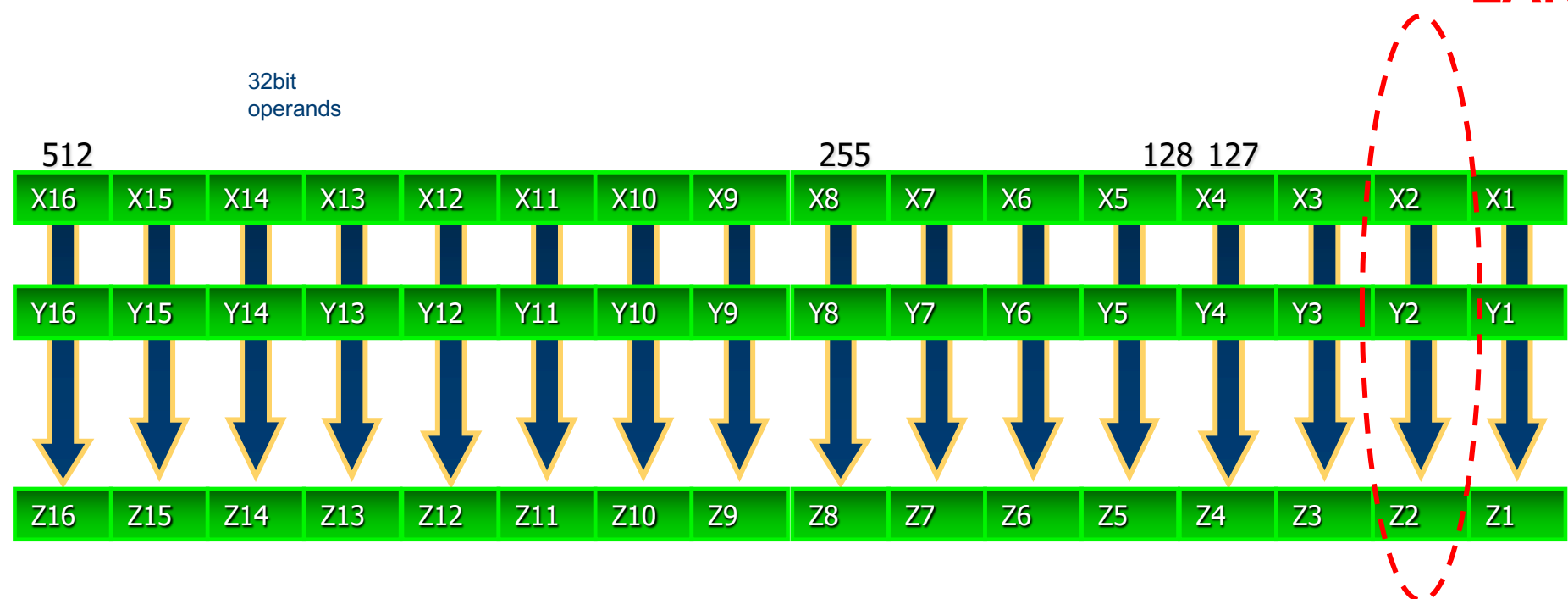
Similar for GNU `-fopenmp[-simd]-parallel]`

No compiler option? Ignores ALL OpenMP directives

Vectorization is Data Parallelism, Blocks of Vector Length

Terminology:
SIMD “LANE” is a parallel slice of elements used in a vector operation .
Think of this as similar to a multi-lane freeway

Vector Dependence:
Does **THIS LANE** need data or affect data from/in **another LANE?**



Clauses for OMP SIMD directives

The programmer (i.e. you!) is responsible for correctness

- Just like for race conditions in loops with OpenMP threading

Available clauses : `#pragma omp` or `!$omp simd` <clauses>

- PRIVATE
 - FIRSTPRIVATE
 - LASTPRIVATE
 - REDUCTION
- } like OpenMP for threading
- COLLAPSE (for nested loops)
 - LINEAR (additional induction variables)
 - SAFELEN (max iterations that can be executed concurrently)
 - ALIGNED (tells compiler about data alignment)

Data Sharing Clauses

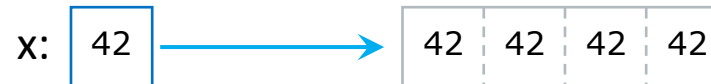
`private(var-list):`

Uninitialized vectors for variables in *var-list* – *allocate register for X, no initial value in each SIMD lane*



`firstprivate(var-list):`

Initialized vectors for variable(s), *each SIMD lane gets initial value*



`reduction(op:var-list):`

Create private variables for *var-list* and apply reduction operator *op* at the end of the construct



SIMD Loop Clauses

`aligned (list[:alignment])`

- Specifies that the list items have a given alignment
- Default is alignment for the architecture

`linear (list[:linear-step])`

- The variable's value is in relationship with the iteration number

$$x_i = x_{\text{orig}} + i * \text{linear-step}$$

`safelen (length)`

- Maximum number of iterations that can run concurrently without breaking a dependence
- in practice, set to the number of elements in the vector length

`collapse (n)`

- Collapse perfectly-nested loops into a single iteration space (loop) and vectorize

safelen(vector size) Example safelen(8)

What this expression inside a vectorized loop:

do/for $i=0, N-1$

$a(i) = a(i + Y) * \text{expr}$

$Y \geq 0$ is safe: $a(i) = a(\text{existing a value})$

$Y < 0$ dangerous: like $a(i) = a(i-1) * \text{expression}$

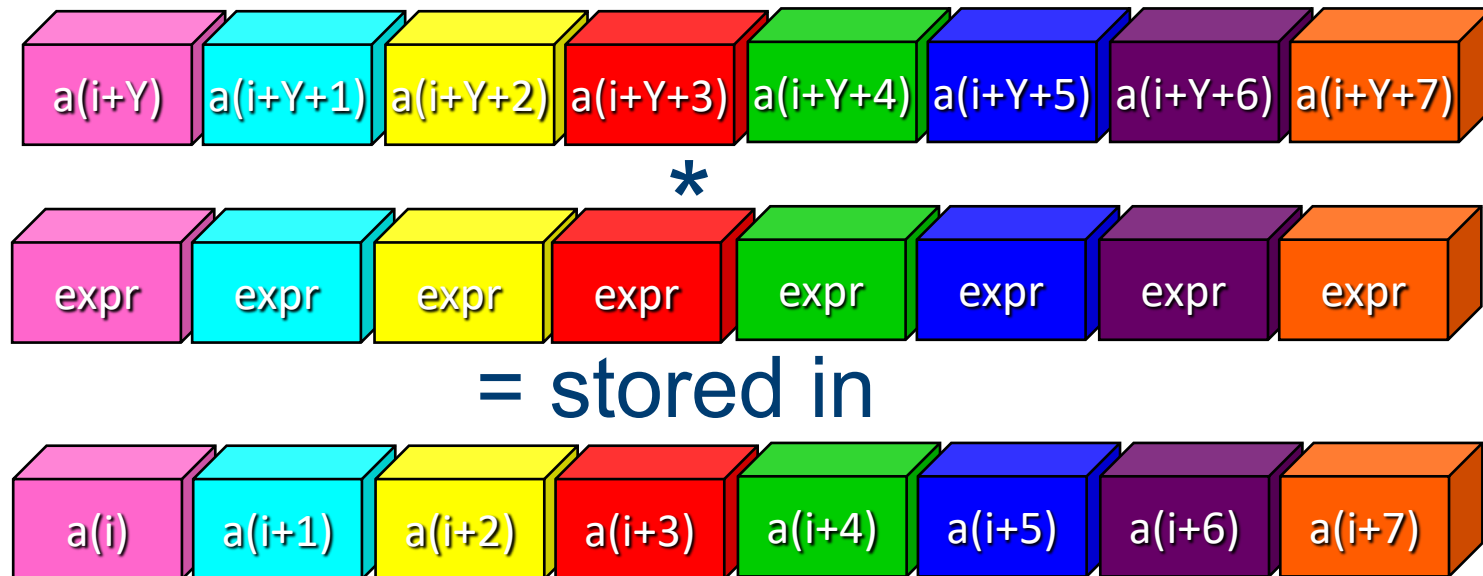
Order dependence: have to compute $a(i-1)$ before $a(i)$

BUT...

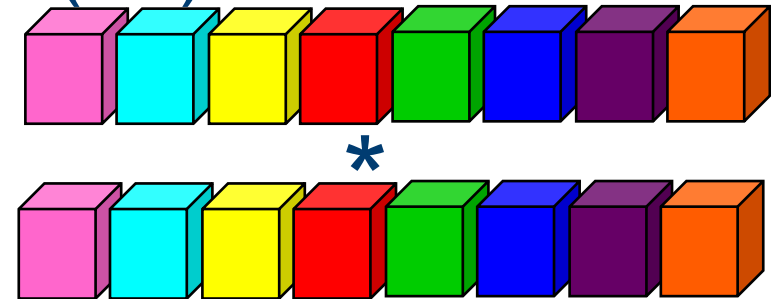
If $Y < 0$ and $Y \leq -8$ and vector len is 8 then

$a(i) = a(i-8) * \text{expression}$

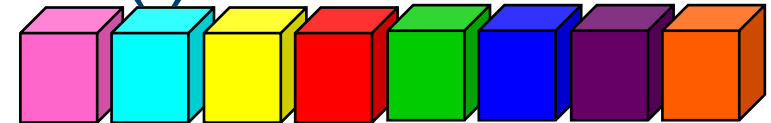
$a(i-8)$ was computed already in previous vector



$a(i-8)$ s last iteration



$a(i)$ s this iteration



SIMD loop example

```
subroutine mypi(count, pi)
    integer, intent(in) :: count
    real , intent(out) :: pi=0
    real :: t
    integer :: i

    !$omp simd private(t) reduction(+:pi)
    do i=1, count
        t = ((i+0.5)/count)
        pi = pi + 4.0/(1.0+t*t)
    end do
    pi = pi/count
end subroutine mypi
```

What About User Functions/Subroutines?

User function/subroutine calls inside loops break vectorization

Procedures may introduce loop carry dependencies

Procedures may have side effects, may need specific serial loop ordering

Compiler defaults to SAFE non-vector code

Exception: intrinsic functions, Intel Fortran provides vector-safe versions that are automatically substituted

- `-lib-inline` (default) `-nolib-inline`

Vectorized Math Intrinsics

- Calls to most mathematical functions in a loop are vectorized using vectorized replacement Library”:
 - Intel Short Vector Math Library (libsvml) provides vectorized implementations of different mathematical functions. Similar for Cray, GNU, etc.

acos	ceil	fabs	round
acosh	cos	floor	sin
asin	cosh	fmax	sinh
asinh	erf	fmin	sqrt
atan	erfc	log	tan
atan2	erfinv	log10	tanh
atanh	exp	log2	trunc
cbrt	exp2	pow	

Also float versions,
such as `sinf()` and Fortran
equivalents

Random Numbers

C/C++: `drand48()`

Fortran: `random_number()`

Concept of SIMD Functions

OMP 4: Allows use of scalar syntax to describe an operation on a single element

Applies operation to arrays in parallel, utilizing vector parallelism (process array elements in blocks of vector length)

The programmer:

Writes a standard procedure with a **scalar syntax**

Call the procedure with **scalar arguments** (no change in syntax for the call)

Annotates procedure at definition with

!\$omp declare SIMD <function, clauses>

At procedure call site, alert the compiler to look for a SIMD version of the procedure

!\$omp simd <clauses>

The compiler:

Generates a short vector version, and a serial version

Invokes the vector version in vectorizable regions, serial otherwise

SIMD procedures: Syntax

!\$omp declare simd (*function-or-procedure-name*) [*clauses*]

Instructs the compiler to

generate a SIMD-enabled version(s) of a given procedure (subroutine or function)

that a SIMD-enabled version procedure is available to use from a SIMD loop

SIMD functions: clauses

`simdlen(length)`

generate function to support a given vector length

`uniform(argument-list)`

argument has a constant value (invariant for all invocations of the procedure in the calling loop)

`inbranch`

function always called from inside an if statement

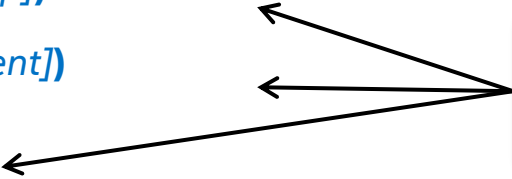
`notinbranch`

function never called from inside an if statement

`linear(argument-list[:linear-step])`

`aligned(argument-list[:alignment])`

`reduction(operator:list)`



Similar to annotations
used on loop vars, only
applied to arguments

Fortran Example

c We integrate the function:

c $f(x) = 4/(1+x^2)$

c between the limits $x=0$ and $x=1$. The result approximates the value of pi.

c The integration method is the n-point rectangle quadrature rule.

```
program computepi
  integer          n, i
  double precision sum, pi, x, h, f
  real start, finish
  external f

  n = 1000000000
  h = 1.0/n
  sum = 0.0

  call cpu_time(start)
  do 10 i = 1,n
    x = h*(i-0.5)
    sum = sum + f(x)
10  continue
  pi = h*sum
```

```
double precision function
f(x)
  double precision x
  f = (4/(1+x*x))
end
```

ifort -O2 -xhost -c fx.f

ifort -O2 -xhost -vec-report3 pi.f fx.o

pi.f(25): (col. 17) remark: routine skipped:
no vectorization candidates.

Example: VECTOR function

```
program computepi
  integer          n, i
  double precision sum, pi, x, h
  real start, finish

  interface
    double precision function f(x)
!$omp declare simd f
    double precision x
  end
end interface

  n = 1000000000
  h = 1.0/n
  sum = 0.0

  call cpu_time(start)
!$omp simd reduction(+:sum) private(x)
  do 10 i = 1,n
    x = h*(i-0.5)
    sum = sum + f(x)
10  continue
  pi = h*sum
```

```
!$omp declare simd f
    double precision function
f(x)
    double precision x
    f = (4/(1+x*x))
end
```

```
ifort -O2 -xhost -c fx2.f
ifort -O2 -xhost -vec-report3 pi2.f fx2.o
```

```
pi2.f(43): (col. 12) remark: SIMD LOOP
WAS VECTORIZED
```

Example: module function

```
program computepi
  use func_f
  integer          n, i
  double precision sum, pi, x, h
  real start, finish

  n = 1000000000
  h = 1.0/n
  sum = 0.0

  call cpu_time(start)
  !$omp simd reduction(+:sum) private(x)
  do 10 i = 1,n
    x = h*(i-0.5)
    sum = sum + f(x)
10  continue
  pi = h*sum
```

```
module func_f

contains
  !$omp declare simd f
  function f(x)
    real(8) :: f
    real(8) :: x
    f = (4/(1+x*x))
  end function
end module func_f
```

ifort -O2 -xhost -c fx3.f

ifort -O2 -xhost -vec-report3 pi3.f fx3.o

pi2.f(43): (col. 12) remark: SIMD LOOP
WAS VECTORIZED

module USE brings in the VECTOR attribute and definition of function 'f'

OpenMP parallel and SIMD

Yes you can nest OMP SIMD inside OMP Parallel directives

Parallel directives at same or outer loop level

SIMD inside parallel region

```
!$omp parallel DO simd ...  
do i=1 , MAX  
  
end do
```

```
!$omp parallel do <clauses>  
do cells=1, NUMCELLS  
    !$omp simd <clauses>  
    do parts=1, NUMPARTSCELL
```

Restrictions on Fortran SIMD procedures

proc-name must not be a generic name, procedure pointer or entry name.

Any **declare simd** directive must appear in a specification part of a subroutine subprogram, function subprogram or interface body to which it applies.

If a **declare simd** directive is specified in an interface block for a procedure, it must match a **declare simd** directive in the definition of the procedure.

If a procedure is declared via a procedure declaration statement, the procedure *proc-name* should appear in the same specification.



Thank You!

**Please Attend Our Hands-On
Lab Following Immediately**

BACKUP and OPTIONAL MATERIAL



OPENMP* TASKING CONCEPTS

Some Background

Prior to standard version 3.0, OpenMP* was focused exclusively on Data Parallelism, distributing work over threads executing the same code.

This work sharing model presented some limitations

- A need for a known loop count
- Very limited ability for dynamic scheduling
- Inconvenient for naturally task-parallel problems (dependencies, nesting)

Task parallelism constructs were introduced to complement the already existing set that supported data parallelism

Task parallelism is particularly useful in irregular computing

What is an OpenMP* Task?

From the standard document: “*specific instance of executable code and its data environment*”

- Explicit task: work generated by the **task** construct
- Implicit task: threads of a parallel region

In this section of the talk I will be only discussing explicit tasks.

By default tasks are deferrable, so the generating thread may execute it immediately or queue it

```
#pragma omp task  
myfunc() ;  
  
#pragma omp task  
for(int i = 0; i < N; i++){ ... }
```

Task Synchronization

Sibling tasks

The **taskwait** construct can be used to wait for deferred task completion at some point in the code

```
#pragma omp task
myfunc();

#pragma omp task
for(int i = 0; i < N; i++){ ... }

#pragma omp taskwait
```

Nested tasks

Synchronizing siblings and their descendants requires a **taskgroup**

```
#pragma omp taskgroup
{
    #pragma omp task
    myfunc();

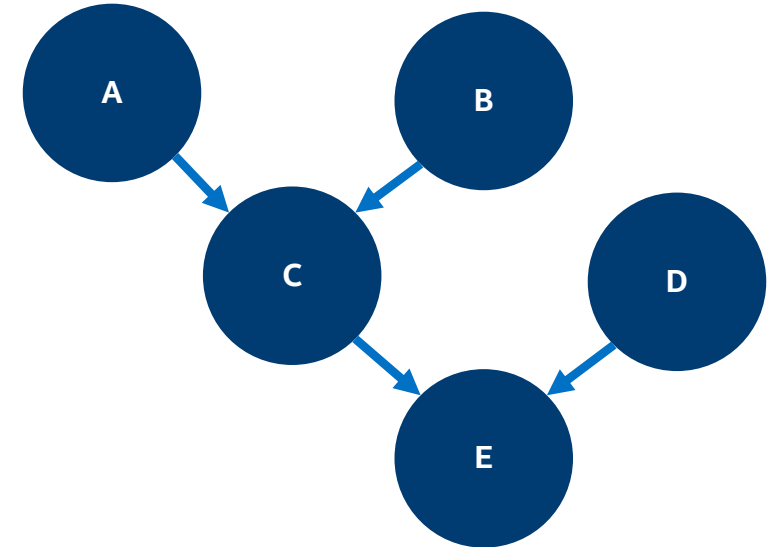
    #pragma omp task
    {
        for(int i = 0; i < N; i++){
            #pragma omp task
            nestedfunc();
        }
    }
}
```

Task Decomposition

Often an application can be decomposed into tasks which can execute simultaneously.

Following the Directed Acyclic Graph (DAG) shown on the right:

- Tasks A, B and C can start executing simultaneously.
- Task C can only be executed after A and B complete execution.
- Task E can only be executed after C and D complete execution.



```
a = A() ;  
b = B() ;  
c = C(a,b) ;  
d = D() ;  
printf( "%f\n", E(c,d) );
```

Parallel Execution of Tasks

```
#pragma omp parallel
{
    #pragma omp single
    {
        #pragma omp task
        a = A();
        #pragma omp task
        b = B();
        #pragma omp task
        d = D();
    }
    c = C(a, b);
    printf ( "%f\n", E(c,d) );
}
```

Start parallel region, forking N threads

Use a single thread to generate the tasks

Each independent code section may be defined as a task

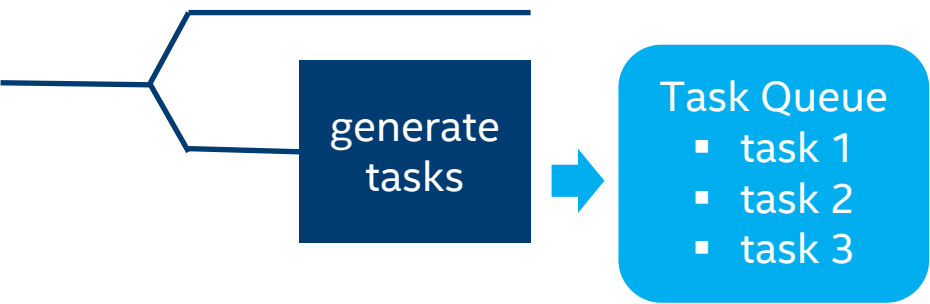
Once generated, each task may be performed by any available thread in the parallel region

Task Generation and Execution

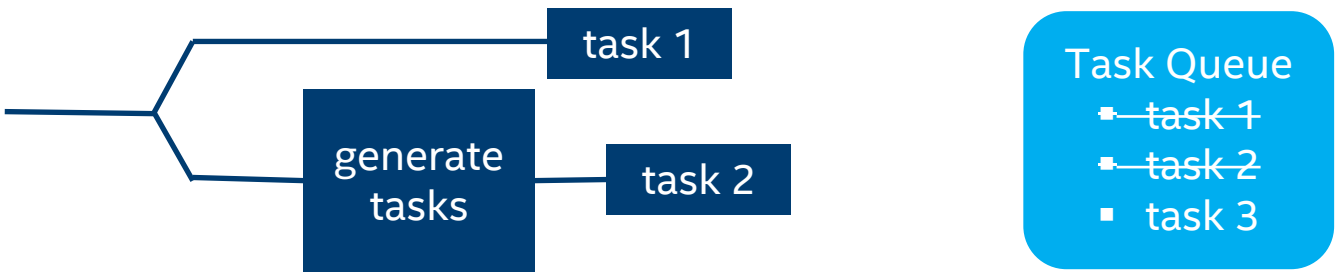
1. Threads are spawned from master



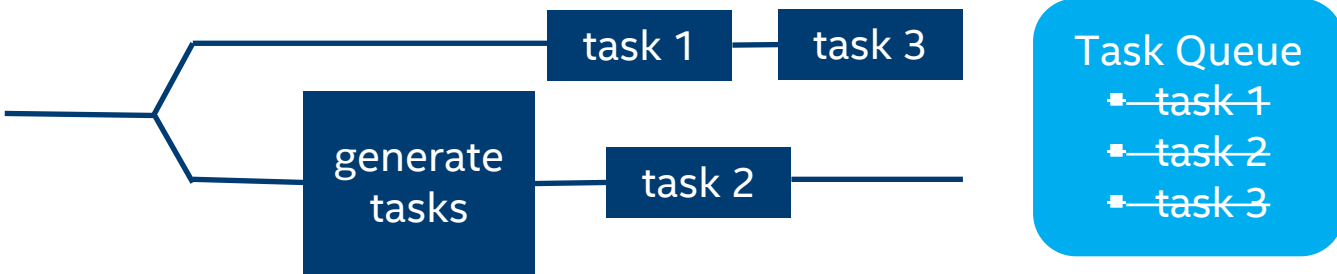
2. Work queue is generated by single thread



3. Tasks in queue are assigned to threads and executed



4. Process continues until queue is empty (or sync point)



Better Scheduling with Depend Clause

```
#pragma omp parallel
{
    #pragma omp single
    {
        #pragma omp task depend(out:a)
        a = A();
        #pragma omp task depend(out:b)
        b = B();
        #pragma omp task depend(out:d)
        d = D();
        #pragma omp task depend(in:a,b) depend(out:c)
        c = C(a, b);
        #pragma omp task depend(in:c,d)
        printf ( "%f\n", E(c,d) );
    }
}
```

depend clause allows to specify dependencies among tasks

depend (<in|out|inout>:<variables>)

Based on dependences C() can start executing once A() and B() are done.

Using the **depend** clause makes it possible to execute C() and D() simultaneously

Parallelize Recursions

```
void merge_sort_openmp(int a[], int tmp[], int first, int last)
{
    if (first < last) {
        int middle = (first + last + 1) / 2;
        if (last - first < 5000) {
            merge_sort(a, tmp, first, middle - 1);
            merge_sort(a, tmp, middle, last);
        } else {
            #pragma omp task
            merge_sort_openmp(a, tmp, first, middle - 1);
            #pragma omp task
            merge_sort_openmp(a, tmp, middle, last);
            #pragma omp taskwait
        }
        merge(a, tmp, first, middle, last);
    }
}
```

Merge sort is common recursive algorithm

- Its recursive nature used to pose a challenge in terms of expressing the parallelism.
- OpenMP* Tasking helps express the parallelism in recursive calls as shown below.
- Explicit taskwait synchronization forces a wait until all sibling tasks complete execution.
- Merging phase can't start until all the tasks spawned above have completed.

Other Interesting Tasking Tidbits

Tasks can be stopped and continued (at scheduling points). By default tasks are **tied** so they can only be continued by the same thread that started them (hot cache). This behavior can be overridden with the **untied** clause

```
#pragma omp task untied
```

You may introduce your own scheduling points using the **taskyield** directive

```
#pragma omp taskyield
```

The **taskloop** directive may be used to schedule loop iterations as independent tasks with a single generator (Intel® Compiler version 18+)

```
#pragma omp taskloop [[grainsize|numtask] [untied] [nogroups]  
[priority]]  
for( i = 0; i < N; i++){ ...}
```

Tasking Summary

Introduced to enable task-parallelism in shared memory architectures

Mostly used in irregular computing

Tasks are typically generated by a single thread

Dependencies can be specified to improve scheduling efficiency

Untied task generators can ensure progress

First-private is default data-sharing attribute

Shared variables remain shared