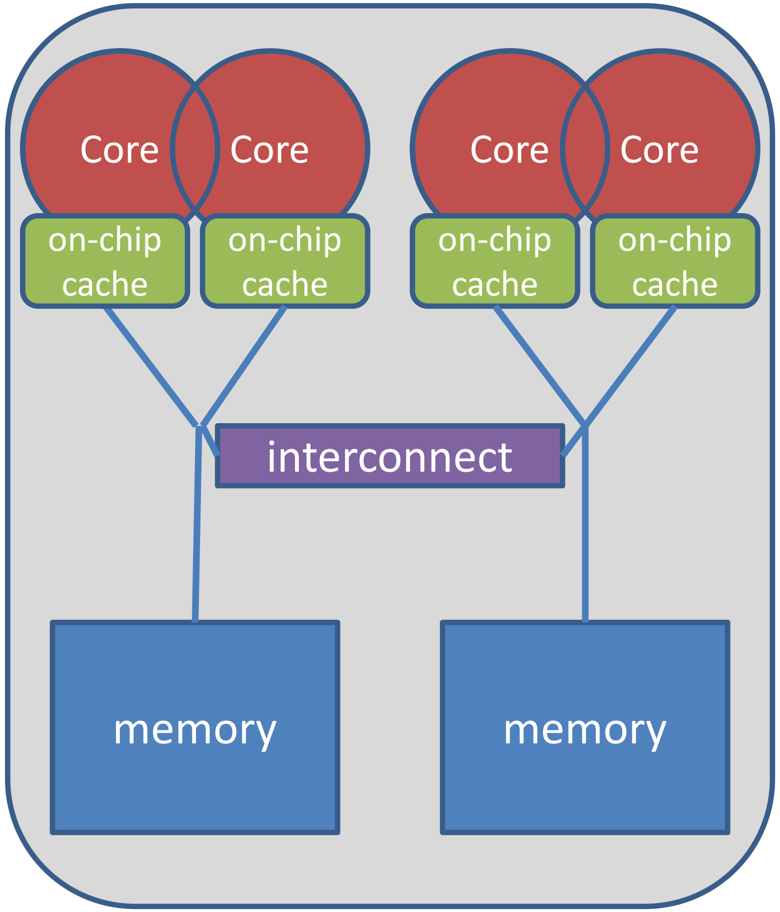
Progamming the OpenMP API

NUMA & Memory Access

Non-uniform Memory

How To Distribute The Data ?

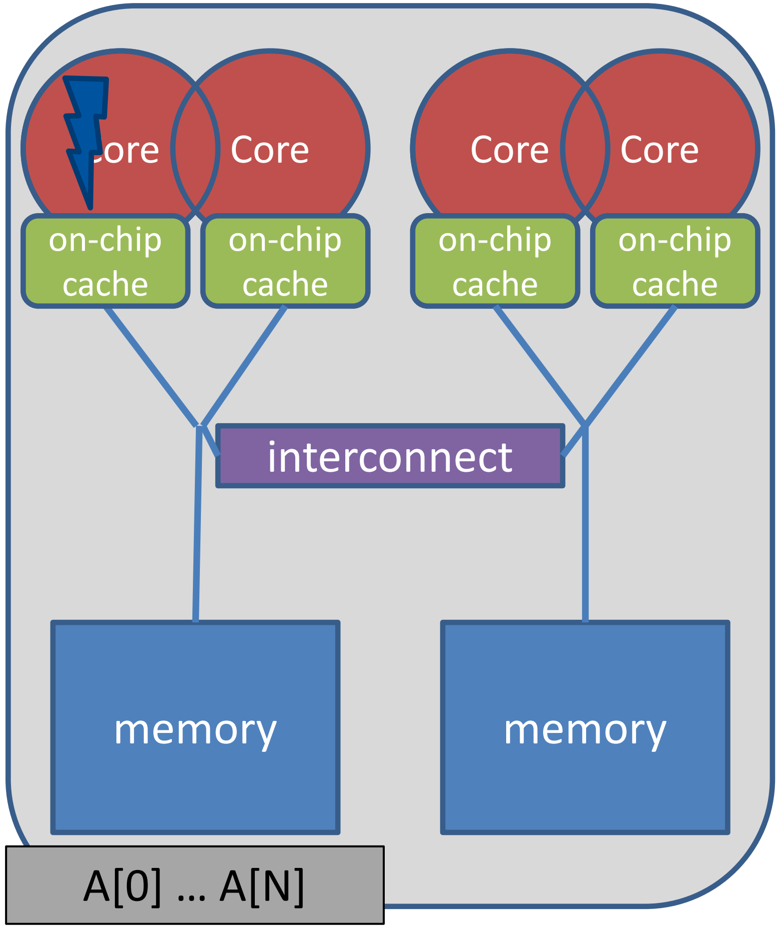
```
double* A;
A = (double*)
    malloc(N * sizeof(double));



for (int i = 0; i < N; i++) {
   A[i] = 0.0;
}
```



Non-uniform Memory

- Serial code: all array elements are allocated in the memory of the NUMA node closest to the core executing the initializer thread (first touch)

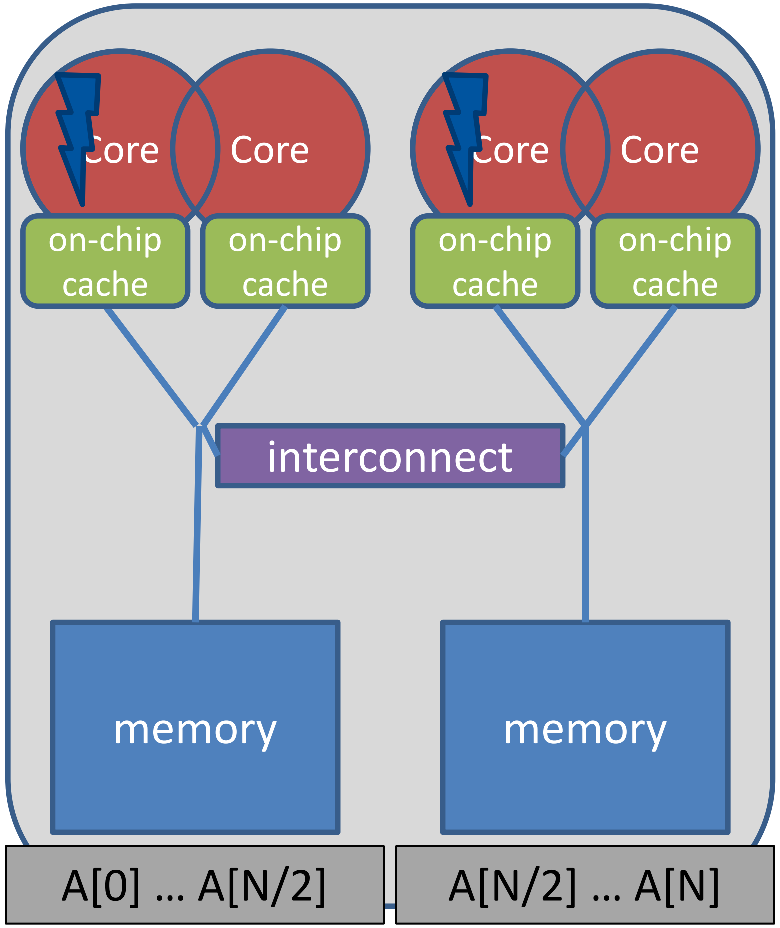
```
double* A;
A = (double*)
    malloc(N * sizeof(double));




for (int i = 0; i < N; i++) {
   A[i] = 0.0;
}
```



First Touch Memory Placement

- First Touch w/ parallel code: all array elements are allocated in the memory of the NUMA node that contains the core that executes the thread that initializes the partition

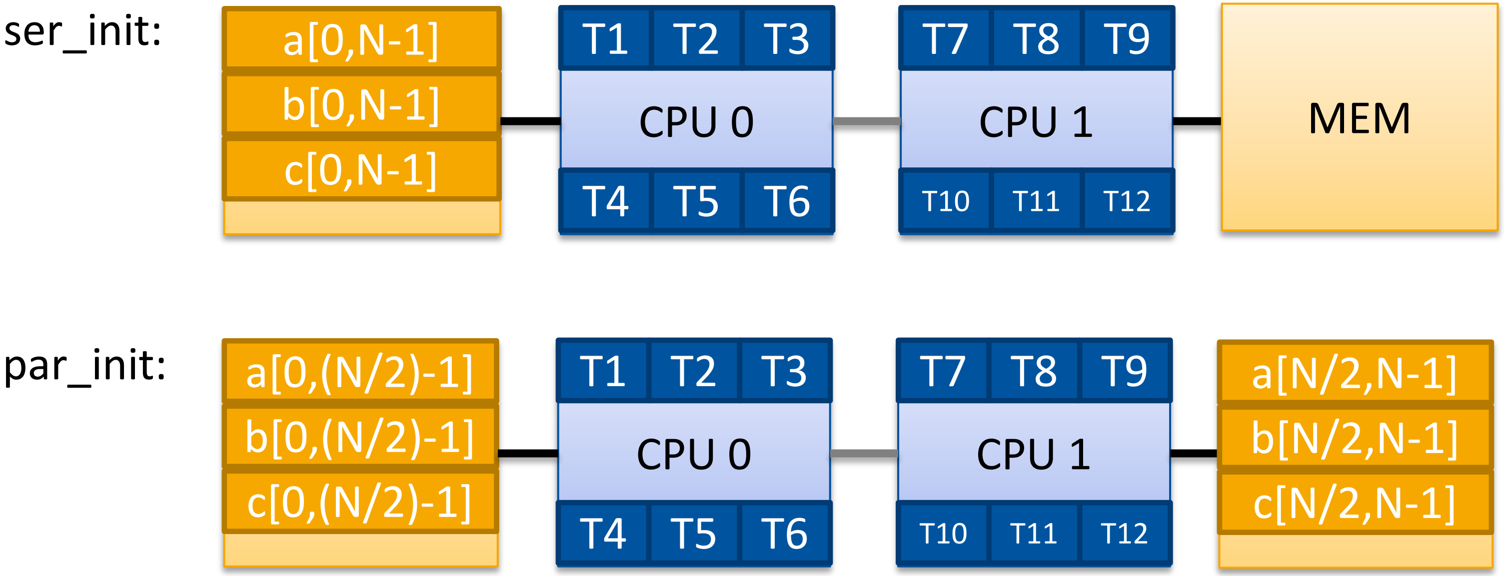
```
double* A;
A = (double*)
    malloc(N * sizeof(double));

omp_set_num_threads(2);

#pragma omp parallel for
for (int i = 0; i < N; i++) {
   A[i] = 0.0;
}
```



Serial vs. Parallel Initialization

- Stream example with and without parallel initialization.
  - → 2 socket sytem with Xeon X5675 processors, 12 OpenMP threads

	copy	scale	add	triad
ser_init	18.8 GB/s	18.5 GB/s	18.1 GB/s	18.2 GB/s
par_init	41.3 GB/s	39.3 GB/s	40.3 GB/s	40.4 GB/s



Thread Binding and Memory Placement

Get Info on the System Topology

- Before you design a strategy for thread binding, you should have a basic understanding of the system topology:
  - → Intel MPI‘s `cpuinfo` tool
    - → `module switch openmpi intelmpi`
    - → `cpuinfo`
    - → Delivers information about the number of sockets (= packages) and the mapping of processor IDs to CPU cores used by the OS
  - → hwlocs‘ `hwloc-ls` tool
    - → `hwloc-ls`
    - → Displays a graphical representation of the system topology, separated into NUMA nodes, along with the mapping of processor IDs to CPU cores used by the OS and additional information on caches

Decide for Binding Strategy

- Selecting the „right“ binding strategy depends not only on the topology, but also on the characteristics of your application.
  - → Putting threads far apart, i.e., on different sockets
    - →May improve the aggregated memory bandwidth available to your application
    - →May improve the combined cache size available to your application
    - →May decrease performance of synchronization constructs
  - → Putting threads close together, i.e., on two adjacent cores that possibly share some caches
    - →May improve performance of synchronization constructs
    - →May decrease the available memory bandwidth and cache size

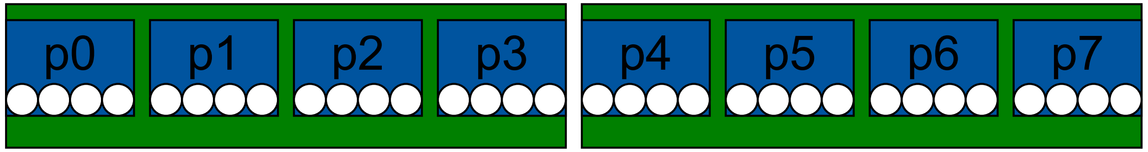
- If you are unsure, just try a few options and then select the best one.

Since OpenMP 4.0: Places + Policies

- Define OpenMP places
  - → set of OpenMP threads running on one or more processors
  - → can be defined by the user, i.e., `OMP_PLACES=cores`
- Define a set of OpenMP thread affinity policies
  - → SPREAD: spread OpenMP threads evenly among the places, partition the place list
  - → CLOSE: pack OpenMP threads near primary thread
  - → PRIMARY: collocate OpenMP thread with primary thread
- Goals
  - → user has a way to specify where to execute OpenMP threads for locality between OpenMP threads / less false sharing / memory bandwidth

OMP_PLACES env. variable

- Assume the following machine:



  - → 2 sockets, 4 cores per socket, 4 hyper-threads per core

- Abstract names for OMP_PLACES:
  - → threads: Each place corresponds to a single hardware thread.
  - → cores: Each place corresponds to a single core (having one or more hardware threads).
  - → sockets: Each place corresponds to a single socket (consisting of one or more cores).
  - → ll_caches (5.1): Each place corresponds to a set of cores that share the last level cache.
  - → numa_domains (5.1): Each places corresponds to a set of cores for which their closest memory is: the same memory; and at a similar distance from the cores.

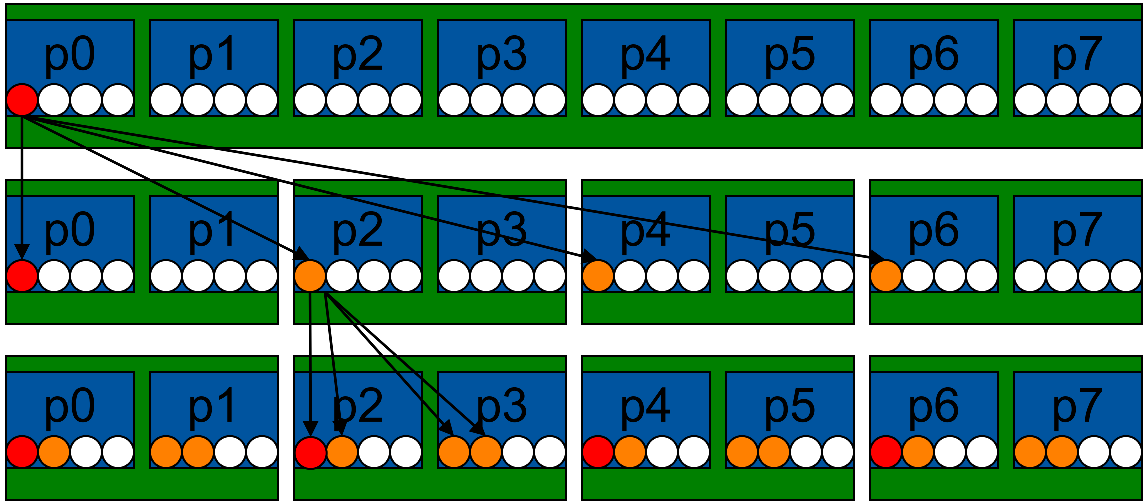
OpenMP 4.0: Places + Policies

- Example‘s Objective:
  - → separate cores for outer loop and near cores for inner loop
- Outer Parallel Region: proc_bind(spread), Inner: proc_bind(close)
  - → spread creates partition, compact binds threads within respective partition

`OMP_PLACES=(0,1,2,3), (4,5,6,7), ... = (0-4):4:8` **= cores**

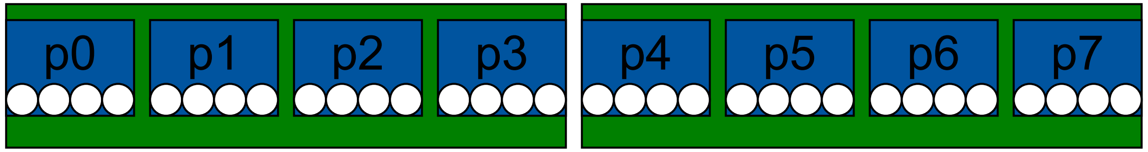
```
#pragma omp parallel proc_bind(spread) num_threads(4)
#pragma omp parallel proc_bind(close) num_threads(4)
```

- Example
  - → initial
  - → spread 4
  - → close 4



More Examples (1/3)

- Assume the following machine:

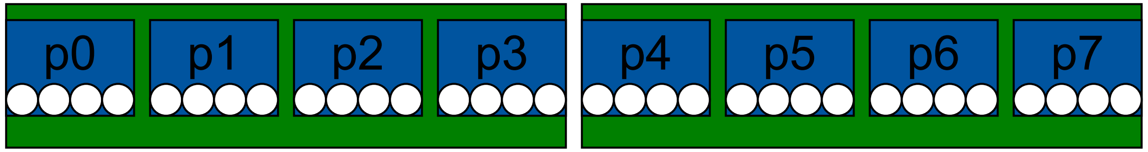


  → 2 sockets, 4 cores per socket, 4 hyper-threads per core

- Parallel Region with two threads, one per socket
  → `OMP_PLACES=sockets`
  → `#pragma omp parallel num_threads(2) proc_bind(spread)`

More Examples (2/3)

- Assume the following machine:



  → 2 sockets, 4 cores per socket, 4 hyper-threads per core

- Parallel Region with four threads, one per core, but only on the first socket

  → `OMP_PLACES=cores`

  → `#pragma omp parallel num_threads(4) proc_bind(close)`

More Examples (3/3)

- Spread a nested loop first across two sockets, then among the cores within each socket, only one thread per core
  - → `OMP_PLACES=cores`
  - → `#pragma omp parallel num_threads(2) proc_bind(spread)`
  - → `#pragma omp parallel num_threads(4) proc_bind(close)`

- Places API routines allow to
  - → query information about binding…
  - → query information about the place partition…

Places API: Example

- Simple routine printing the processor ids of the place the calling thread is bound to:

```
void print_binding_info() {
     int my_place = omp_get_place_num();
     int place_num_procs = omp_get_place_num_procs(my_place);

     printf("Place consists of %d processors: ", place_num_procs);

     int *place_processors = malloc(sizeof(int) * place_num_procs);
     omp_get_place_proc_ids(my_place, place_processors)

     for (int i = 0; i < place_num_procs - 1; i++) {
             printf("%d ", place_processors[i]);
     }
     printf("\n");

     free(place_processors);
}
```

OpenMP 5.x way to do this

- Set `OMP_DISPLAY_AFFINITY=TRUE`
  - →Instructs the runtime to display formatted affinity information
  - →Example output for two threads on two physical cores:
  - →Output c corresponding routine
  - →Formatted affinity information can be printed with `omp_display_affinity(const char* format)`

```
nesting_level=   1,   thread_num=   0,   thread_affinity=   0,1
nesting_level=   1,   thread_num=   1,   thread_affinity=   2,3
```

Affinity format specification

t	omp_get_team_num()
T	omp_get_num_teams()
L	omp_get_level()
n	omp_get_thread_num()
N	omp_get_num_threads()

a	omp_get_ancestor_thread_num() at level-1
H	hostname
P	process identifier
i	native thread identifier
A	thread affinity: list of processors (cores)

- Example:

```
OMP_AFFINITY_FORMAT=“Affinity: %0.3L %.8n %.15{A} %.12H‟
```

→Possible output:

```
Affinity: 001        0      0-1,16-17      host003
Affinity: 001        1      2-3,18-19      host003
```

Fine-grained control of Memory Affinity

- Explicit NUMA-aware memory allocation:
  - →By carefully touching data by the thread which later uses it
  - →By changing the default memory allocation strategy
    - →Linux: `numactl` command
  - →By explicit migration of memory pages
    - →Linux: `move_pages()`

- Example: using numactl to distribute pages round-robin:
  - →

```
numactl –interleave=all ./a.out
```

Memory Management

Different kinds of memory

- Traditional DDR-based memory
- High-bandwidth memory
- Non-volatile memory
- …

Cascade Lake (Leonide at INRIA)

```
CPU: Intel(R) Xeon(R) Gold 6230 CPU @ 2.10GHz
Freq Govenor: performance
---------------------
available: 4 nodes (0-3)
node 0 cpus: 0 2 4 6 8 10 12 14 16 18
             20 22 24 26 28 30 32 34 36 38
node 0 size: 191936 MB
node 0 free: 178709 MB
node 1 cpus: 1 3 5 7 9 11 13 15 17 19 21 23
             25 27 29 31 33 35 37 39
node 1 size: 192016 MB
node 1 free: 179268 MB
node 2 cpus:
node 2 size: 759808 MB
node 2 free: 759794 MB
node 3 cpus:
node 3 size: 761856 MB
node 3 free: 761851 MB
node distances:
node   0   1   2   3
  0:  10  21  17  28
  1:  21  10  28  17
  2:  17  28  10  28
  3:  28  17  28  10
```

DRAM + Optane

Memory Management

- Allocator := an OpenMP object that fulfills requests to allocate and deallocate storage for program variables

- OpenMP allocators are of type `omp_allocator_handle_t`

- Default allocator for host
  - →via `OMP_ALLOCATOR` env. var. or corresponding API

- OpenMP 5.0 supports
a set of memory allocators

OpenMP allocators

- Selection of a certain kind of memory

Allocator name	Storage selection intent
omp_default_mem_alloc	use default storage
omp_large_cap_mem_alloc	use storage with large capacity
omp_const_mem_alloc	use storage optimized for read-only variables
omp_high_bw_mem_alloc	use storage with high bandwidth
omp_low_lat_mem_alloc	use storage with low latency
omp_cgroup_mem_alloc	use storage close to all threads in the contention group of the thread requesting the allocation
omp_pteam_mem_alloc	use storage that is close to all threads in the same parallel region of the thread requesting the allocation
omp_thread_local_mem_alloc	use storage that is close to the thread requesting the allocation

Using OpenMP allocators

- New clause on all constructs with data sharing clauses:
  - → `allocate( [allocator:] list )`
- Allocation:
  - → `omp_alloc(size_t size, omp_allocator_handle_t allocator)`
- Deallocation:
  - → `omp_free(void *ptr, const omp_allocator_handle_t allocator)`

- `allocate` directive: standalone directive for allocation, or declaration of allocation stmt.

OpenMP allocator traits / 1

- Allocator traits control the behavior of the allocator

sync_hint	contended, uncontended, serialized, private default: contended
alignment	positive integer value that is a power of two default: 1 byte
access	all, cgroup, pteam, thread default: all
pool_size	positive integer value
fallback	default_mem_fb, null_fb, abort_fb, allocator_fb default: default_mem_fb
fb_data	an allocator handle
pinned	true, false default: false
partition	environment, nearest, blocked, interleaved default: environment

OpenMP allocator traits / 2

- `fallback`: describes the behavior if the allocation cannot be fulfilled
  - →`default_mem_fb`: return system's default memory
  - →Other options: null, abort, or use different allocator
- `pinned`: request pinned memory, i.e. for GPUs

OpenMP allocator traits / 3

- `partition`: partitioning of allocated memory of physical storage resources (think of NUMA)
  - →`environment`: use system’s default behavior
  - →`nearest`: most closest memory
  - →`blocked`: partitioning into approx. same size with at most one block per storage resource
  - →`interleaved`: partitioning in a round-robin fashion across the storage resources

Using OpenMP allocator traits

- Construction of allocators with traits via
  - → `omp_allocator_handle_t   omp_init_allocator( omp_memspace_handle_t memspace, int ntraits, const omp_alloctrait_t traits[]);`
  - → Selection of memory space mandatory
  - → Empty traits set: use defaults

- Allocators have to be destroyed with `*_destroy_*`

- Custom allocator can be made default with `omp_set_default_allocator(omp_allocator_handle_t allocator)`

Memory Management Status

- **LLVM OpenMP runtime internally already uses libmemkind (libnuma, numactl)**
  - → Support for various kinds of memory: DDR, HBW and Persistent Memory (Optane)
  - → Library loaded at initialization (checks for availability)
  - → If requested memory space for allocator is not available → fallback to DDR

- **Memory Management implementation in LLVM still not complete**
  - → Some allocator traits not implemented yet
  - → Some `partition` values not implemented yet (environment, interleaved, nearest, blocked)
  - → Semantics of `omp_high_bw_mem_space and omp_large_cap_mem_space` unclear. Which memory should be used?
    - → Explicitly target HBM → currently implemented in LLVM

- **LLVM has custom implementation of aligned memory allocation**
  - → Allocation covers → {Allocator Information + Requested Size + Buffer based on alignment}