OpenMP 4 (and now 5.0)

John Urbanic
Parallel Computing Scientist
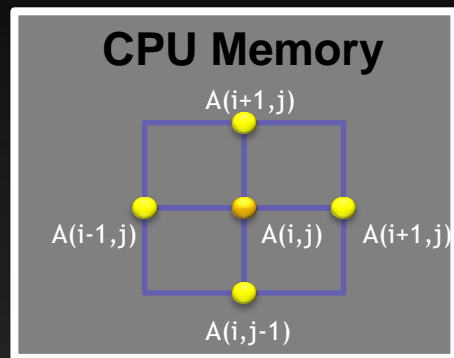*Pittsburgh Supercomputing Center*

Classic OpenMP

OpenMP was designed to replace low-level and tedious solutions like POSIX threads, or Pthreads.
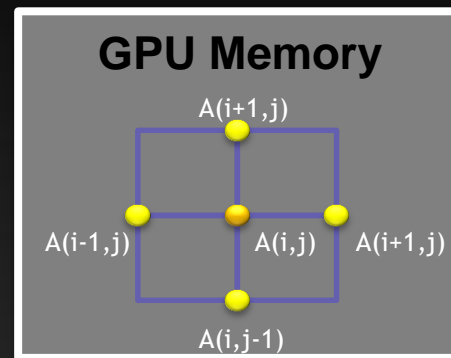
OpenMP was originally targeted towards controlling capable and completely independent processors, with shared memory. The most common such configurations today are the many multi-cored chips we all use. You might have dozens of threads, each of which takes some time to start or complete.

In return for the flexibility to use those processors to their fullest extent, OpenMP assumes that you know what you are doing. You prescribe what how you want the threads to behave and the compiler faithfully carries it out.
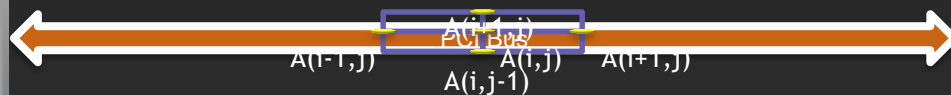
Then Came This

CPU Memory

A(i+1,j)

A(i-1,j)   A(i,j)   A(i+1,j)

A(i,j-1)

GPU Memory

A(i+1,j)

A(i-1,j)   A(i,j)   A(i+1,j)

A(i,j-1)

CPU

GPU

PCI Bus

A(i+1,j)

A(i-1,j)   A(i,j)   A(i+1,j)

A(i,j-1)

GPUs are not CPUs

GPU require memory management. We do not simply have a single shared dataspace.

GPUs have thousands of cores.

But they aren't independent.

And they can launch very lightweight threads.

But it seems like the OpenMP approach provides a good starting point to get away from the low-level and tedious CUDA API...

Original Intention

OpenACC 1   OpenACC 2   OpenACC 2.5

OpenMP 3   OpenMP 4   OpenMP 4.5   OpenMP 5
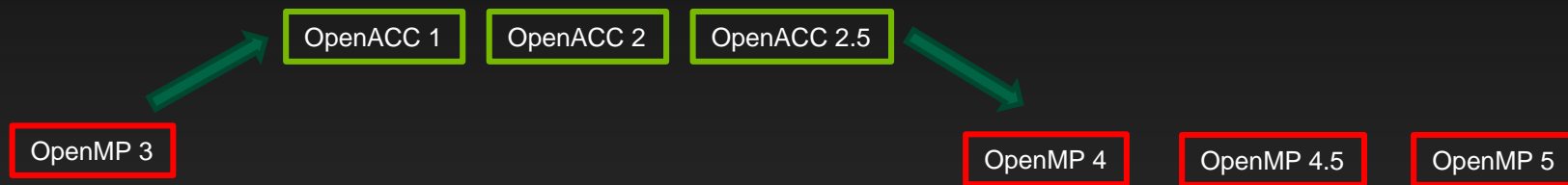
Let OpenACC evolve rapidly without disturbing the mature OpenMP standard.
They can merge somewhere around version 4.0.

Meanwhile...

Since the days of RISC vs. CISC, Intel has mastered the art of figuring out what is important about a new processing technology and saying "why can't we do this in x86?"
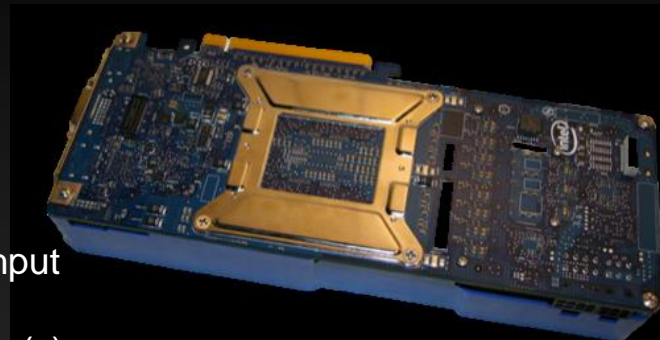
The Intel Many Integrated Core (MIC) architecture is about large die, simpler circuit, and much more parallelism, in the x86 line.

What ~~is~~ was MIC?

Basic Design Ideas:

- Leverage x86 architecture (a CPU with many cores)
- Use x86 cores that are simpler, but allow for more compute throughput
- Leverage existing x86 programming models
- Dedicate much of the silicon to floating point ops., keep some cache(s)
- Keep cache-coherency protocol
- Increase floating-point throughput per core
- Implement as a separate device
- Strip expensive features (out-of-order execution, branch prediction, etc.)
- Widened SIMD registers for more throughput (512 bit)
- Fast (GDDR5) memory on card

PITTSBURGH
SUPERCOMPUTING
CENTER

~~Latest~~ last MIC Architecture

- Ma

- L1

- Bi
  ne
  an

Courtesy Dan Stanzione, TACC

Implications for the OpenMP/OpenACC Merge

NVIDIA

OpenACC 1   OpenACC 2   OpenACC 2.5

OpenMP 3

OpenMP ?

OpenMP 4   OpenMP 4.5   OpenMP 5
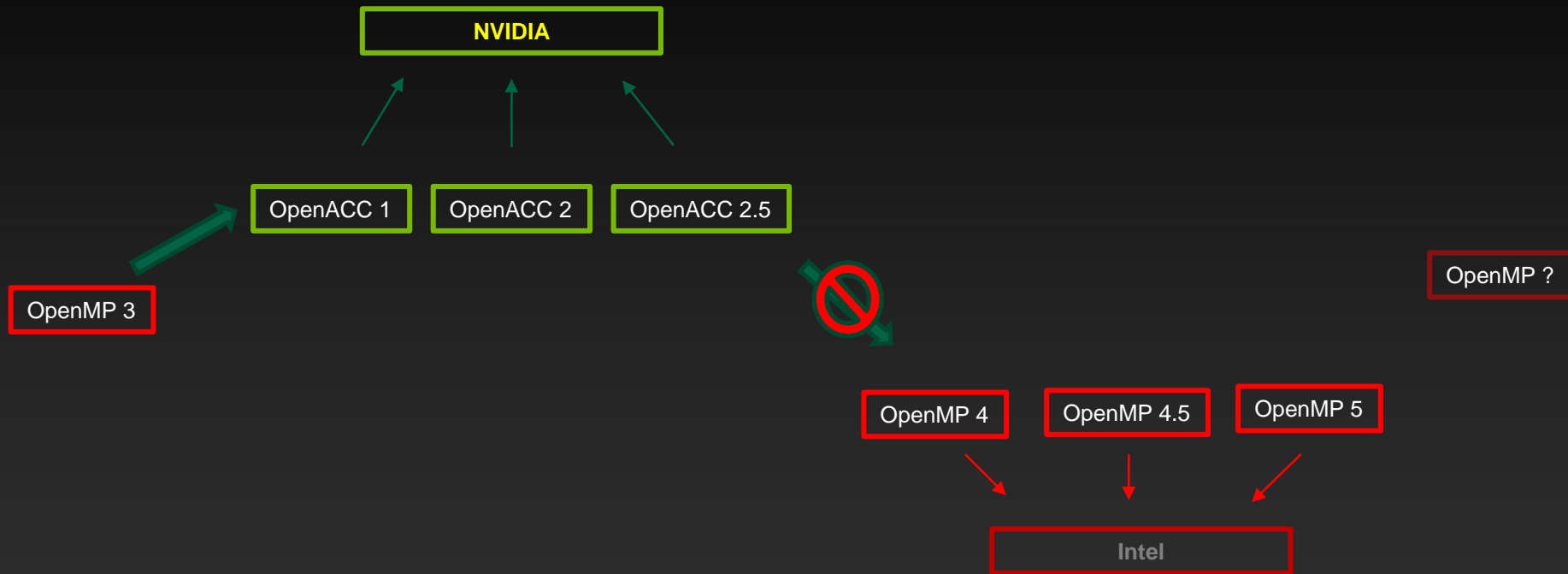
Intel

Intel and NVIDIA have both influenced their favored approached to make them more amenable to their own devices.

PITTSBURGH
SUPERCOMPUTING
CENTER

OpenMP 4.0

The OpenMP 4.0 standard did incorporate the features needed for accelerators, with an emphasis on Intel-like devices. We are left with some differences.

OpenMP takes its traditional prescriptive approach ("*this is what I want you to do*"), while OpenACC could afford to start with a more modern (compilers are smarter) descriptive approach ("*here is some parallelizable stuff*, *do something smart*"). This is practically visible in such things as OpenMP's insistence that you identify loop dependencies, versus OpenACC's *kernel* directive, and its ability to spot them for you.

OpenMP assumes that every thread has its own synchronization control (`barriers`, `locks`), because real processors can do whatever they want, whenever. GPUs do not have that at all levels. For example, NVIDIA GPUs have synchronization at the warp level, but not the thread block level. There are implications regarding this difference such as no OpenACC `async`/`wait` in parallel regions or kernels.

In general, you might observe that OpenMP was built when threads were limited and start up overhead was considerable (as it still is on CPUs). The design reflects the need to control for this. OpenACC starts with devices built around thousands of very, very lightweight threads.

They are also complementary and can be used together very well.

OpenMP 4.0 Data Migration

The most obvious improvements for accelerators are the data migration commands. These look very similar to OpenACC.

```
#pragma omp target device(0) map(tofrom:B)
```

OpenMP vs. OpenACC Data Constructs

OpenMP

- target data
- target enter data
- target exit data
- target update
- declare target

OpenACC

- data
- enter data
- exit data
- update
- declare

OpenMP vs. OpenACC Data Clauses

OpenMP

OpenACC

- map
- map
- map
- map
- map
- map

OpenMP 5 has also embraced the NVIDIA "unified shared memory" paradigm

```
#pragma omp requires unified_shared_memory

complex_deep_data * cdp = create_array_of_data();

#pragma omp target    //Notice no mapping clauses!
operate_on_data( cdp );
```

Just like with NVIDIA Unified Memory, this is hopelessly naïve and is not used in production code. It is often recommended for a "first pass" (but I find that counter-productive).

The closely related deep copy directives (*declare mapper*) can be useful to aid in moving pointer-based data. As can the *allocate* clauses.

OpenMP vs. OpenACC Compute Constructs

OpenMP

- target
- teams
- distribute
- parallel
- for / do
- simd
- is_device_ptr(…)

OpenACC

- parallel / kernels
- parallel / kernels
- loop gang
- parallel / kernels
- loop worker or loop gang
- loop vector
- deviceptr(…)

OpenMP vs. OpenACC Differences

OpenMP

- device(n)
- depend(to:a)
- depend(from:b)
- nowait
- loops, tasks, sections
- atomic
- master, single, critical, barrier, locks, ordered, flush, cancel

OpenACC

- ---
- async(n)
- async(n)
- async
- loops
- atomic
- ---

SAXPY in OpenMP 4.0 on NVIDIA

```c
int main(int argc, const char* argv[]) {
    int n = 10240; floata = 2.0f; floatb = 3.0f;
    float*x = (float*) malloc(n * sizeof(float));
    float*y = (float*) malloc(n * sizeof(float));

// Run SAXPY TWICE inside data region
#pragma omp target data map(to:x)
{
#pragma omp target map(tofrom:y)
#pragma omp teams
#pragma omp distribute
#pragma omp parallel for
    for(inti = 0; i < n; ++i){
        y[i] = a*x[i] + y[i];
    }
#pragma omp target map(tofrom:y)
#pragma omp teams
#pragma omp distribute
#pragma omp parallel for
    for(inti = 0; i < n; ++i){
        y[i] = b*x[i] + y[i];
    }
}
}
```

PITTSBURGH
SUPERCOMPUTING
CENTER

Comparing OpenACC with OpenMP 4.0 on NVIDIA & Phi

OpenMP 4.0 for Intel Xeon Phi

```
#pragma omp target device(0) map(tofrom:B)
#pragma omp parallel for
for (i=0; i<N; i++)
    B[i] += sin(B[i]);
```

OpenMP 4.0 for NVIDIA GPU

```
#pragma omp target device(0) map(tofrom:B)
#pragma omp teams num_teams(num_blocks) num_threads(bsize)
#pragma omp distribute
for (i=0; i<N; i += num_blocks)
        #pragma omp parallel for
        for (b = i; b < i+num_blocks; b++)
            B[b] += sin(B[b]);
```

OpenACC for NVIDIA GPU

```
#pragma acc kernels
for (i=0; i<N; ++i)
    B[i] += sin(B[i]);
```

PITTSBURGH
SUPERCOMPUTING
CENTER

OpenMP 4.0 Across Architectures

OpenMP now has a number of OpenACC-like metadirectives to help cope with this confusion:

```
#pragma omp target map(to:a,b) map(from:c)
#pragma omp metadrictive when (device={arch(nvptx)}: teams loop) default (parallel loop)
for (i = 1; i<n; i++)
    c[i] = a[i] * b[i]
```

And also variant functions to substitute code for different targets.

```
#pragma omp declare target
int some_routine(int a){
    //do things in serial way
}

#pragma omp declare variant ( int some_routine(int a) ) match(context={target} \
                                               device={arch(nvptx)} )

int some_routine_gpu(int a){
    //gpu optimized code
}
```

OpenMP 4.0 Across Compilers

Cray C Compiler (v8.5)

```
#pragma omp target teams distribute
for(int ii = 0; ii < y; ++ii)
```

Clang Compiler (alpha)

```
#pragma omp target teams distribute \
        parallel for chedule(static,1)
for(int ii = 0; ii < y; ++ii)
```

Intel C Compiler (v16.0)

```
#pragma omp target
#pragma omp parallel
for for(int ii = 0; ii < y; ++ii)
```

GCC C Compiler (v6.1))

```
#pragma omp target teams distribute \
        parallel for
for(int ii = 0; ii < y; ++ii)
```

Subtle and confusing? You bet.
For a nice discussion of these examples visit their authors at
https://www.openmp.org/wp-content/uploads/Matt_openmp-booth-talk.pdf

PITTSBURGH
SUPERCOMPUTING
CENTER

Latest Data.

From the excellent paper *Is OpenMP 4.5 Target Off-load Ready for Real Life? A Case Study of Three Benchmark Kernels (Diaz, Jost, Chandrasekaran, Pino)* we have some recent data:
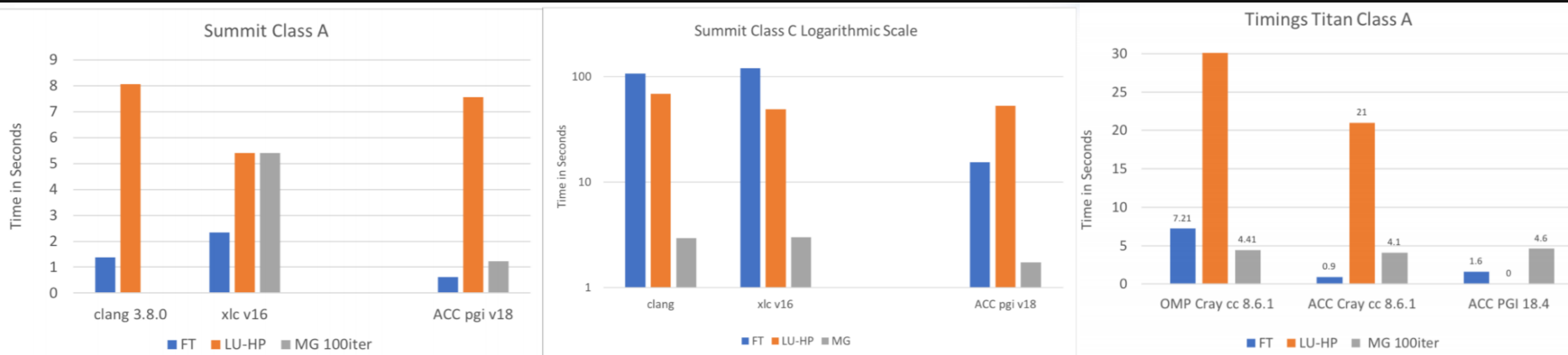
In summary, using NPB benchmarks (FFT, Gauss Seidel, Multi-Grid) on leadership class platforms (Titan and Summit) using multiple compilers (clang, gcc, PGI, Cray, IBM), OpenMP is not yet competitive with OpenACC on GPUs.

A very interesting side-note is that OpenACC *kernels* and *loop* performed the same.

So, at this time…

- If you are using Phi Knights Corner or Knights Landing, you are probably going to be using the Intel OpenMP 4+ release. Unless you use it in cache mode, and then this is moot (more later).

- If you are using NVIDIA GPUs, you are going to be using OpenACC.

Of course, there are other ways of programming both of these devices. You might treat Phi as MPI cores and use CUDA on NVIDIA , for example. But if the directive based approach is for you, then your path is clear. I don't attempt to discuss the many other types of accelerators here (AMD, DSPs, FPGAs, ARM), but these techniques apply there as well.

And as you should now suspect, even if it takes a while for these to merge as a standard, it is not a big jump for you to move between them.
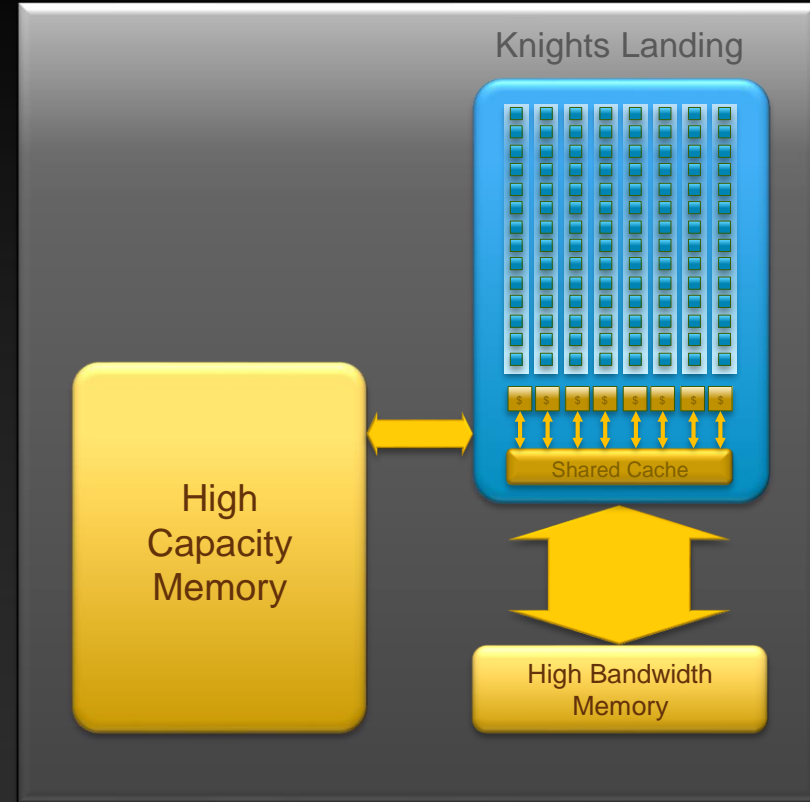
Is OpenMP 4 moot?*

OpenMP 5 and the Knights Landing architecture seem to be suggesting we should use cache mode and forget all about the data management.

Developments thus far show this to be a popular, if inefficient, approach.

With the deprecation of the Phi family, perhaps all of these target-oriented extensions are moot...

Unless this takes over from OpenACC as the dominant way of programming NVIDIA devices.

*Note we are just talking about data management here. OpenMP 4.0 has some other nifty features.

Knights Landing

Shared Cache

High Capacity Memory

High Bandwidth Memory

PITTSBURGH SUPERCOMPUTING CENTER

OpenMP 4.0 SIMD Extension

Much earlier I mentioned that vector instructions fall into the realm of "things you hope the compiler addresses". However as they have become so critical achieving available performance on newer devices, the OpenMP 4.0 standard has included a simd directive to help you help the compiler. There are two main calls for it.

1) Indicate a simple loop that should be vectorized. It may be an inner loop on a parallel for, or it could be standalone.

```
#pragma omp parallel
{
  #pragma omp for
  for (int i=0; i<N; i++) {
    #pragma omp simd safelen(18)
    for (int j=18; j<N-18; j++) {
      A[i][j] = A[i][j-18] + sinf(B[i][j]);
      B[i][j] = B[i][j+18] + cosf(A[i][j]);
    }
  }
}
```

There is dependency that prohibits vectorization. However, the code can be vectorized for any given vector length for array B and for vectors shorter than 18 elements for array A.

OpenMP 4.0 SIMD Extension

2) Indicate that a function is vectorizable.

```
#pragma omp declare simd
float some_func(float x) {
        ...
        ...
}

#pragma omp declare simd
extern float some_func(float);

void other_func(float *restrict a, float *restrict x, int n) {
  for (int i=0; i<n; i++)  a[i] = some_func(x[i]);
}
```

There are a ton of clauses (private, reduction, etc.) that help you to assure safe conditions for vectorization. They won't get our attention today.

We won't hype these any further. Suffice it to say that if the compiler report indicates that you are missing vectorization opportunities, this adds a portable tool.

PITTSBURGH
SUPERCOMPUTING
CENTER

OpenMP 5.0
A New Emphasis

OpenMP 5.0 shows the real new focus of OpenMP. It is a recognition of the increasing importance of the memory hierarchies found in real devices, and how to manage it. Just on a single node (our concern for OpenMP) we have:

- Registers (including vector registers)
- Caches (multiple levels)
- RAM (processor local or NUMA memory)
- HBM?
- Accelerators?
- NVM?

These are being accessed in various patterns by:

- Loops (hopefully vectorized)
- Threads
- Processes
- Cores
- Processors

ORNL Cray XC30 Node
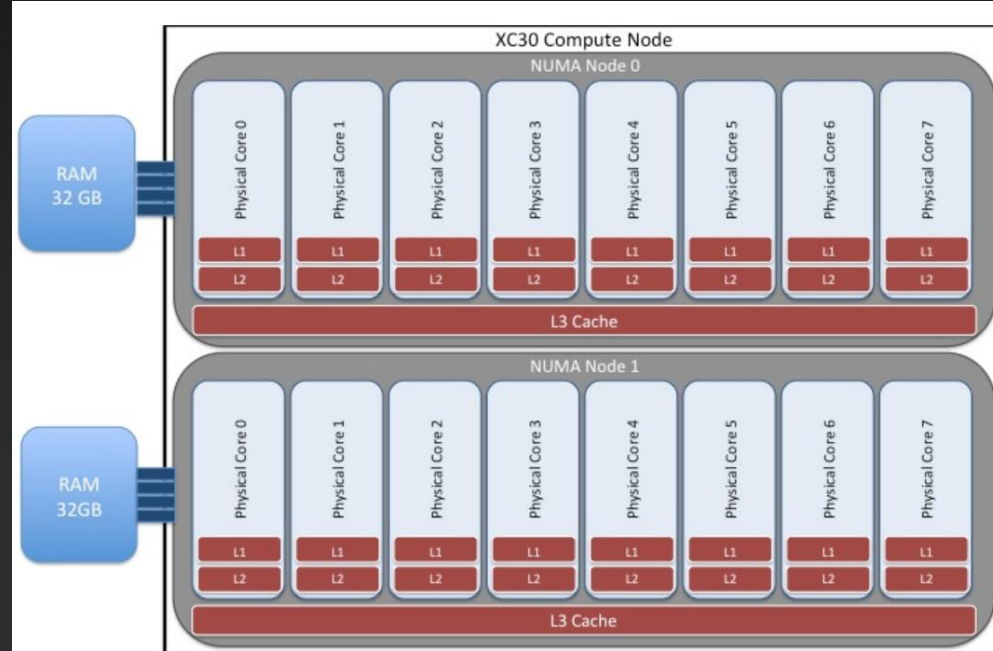
OpenMP 5.0

The specifiers in the new spec give you some idea of how many ways we can characterize this.

distance ≈ near, far   Specifies the relative physical distance of the memory space with respect to the task the request binds to.

bandwidth ≈ highest, lowest   Specifies the relative bandwidth of the memory space with respect to other memories in the system

latency ≈ highest, lowest   Specifies the relative latency of the memory space with respect to other memories in the system.

location = Specifies the physical location of the memory space.

optimized = bandwidth, latency, capacity, none   Specifies if the memory space underlying technology is optimized to maximize a certain characteristic. The exact mapping of these values to actual technologies is implementation defined.

pagesize = positive integer   Specifies the size of the pages used by the memory space.

permission = r, w, rw   Specifies if read operations (r), write operations (w) or both (rw) are supported by the memory space.

capacity ≥ positive integer   Specifies the physical capacity in bytes of the memory space. available ≥ positive integer Specifies the current available capacity for new allocations in the memory space.

Data Affinity

These new specifiers are used in multiple ways throughout the spec. They will show up quite naturally in the data management clauses for the accelerators that we have just discussed.

They also have more accessible, and universal uses. One topic that we haven't had time for today, but is frequently of interest when optimizing multi-threaded codes, is data affinity. Many common multi-core systems present the main RAM memory as a unified space. In reality the some sections of the memory can be closer to certain cores than other sections. And certainly the cache is closer to particular cores. It is nice if our threads can run nearest the data they need.

The reason we haven't had time for this today is that the placement of threads, or their migration by the operating system, is typically very platform dependent and the means of controlling this behavior is often a number of overlapping bits of trivia; sometimes environment variables, other times wrappers to run the code, or kernel settings.

OpenMP brings some standardization to this.

Easy Data Affinity

Here is a good example of how easy it can be to request data/thread affinity for a couple of tasks that we know share data.

```
void related_tasks( float* A, int n ){

    float* B;

    #pragma omp task shared(B) depend(out:B) affinity(A[0:n])
    {
        B = compute_B(A,n);
    }
    #pragma omp task firstprivate(B) depend(in:B) affinity (A[0:n])
    {
        update_B(B);
    }
    #pragma omp taskwait
}
```

Some things we did not mention

- **OpenCL (Khronos Group)**
    - Everyone supports, but not as a primary focus
    - Intel – OpenMP
    - NVIDIA – CUDA, OpenACC
    - AMD – now HSA (hUMA/APU oriented)
- **Fortran 2008+ threads (sophisticated but not consistently implemented)**
- **C++11 threads are basic (no loops) but better than POSIX**
- **Python threads are fake (due to Global Interpreter Lock)** But *you* can now fix this!
- **DirectCompute (Microsoft) is not HPC oriented**
- **C++ AMP (MS/AMD)**
- **TBB (Intel C++ template library)**
- **Cilk (Intel, now in a gcc branch)**

Very C++ for threads