OpenMP and GPUs

John Urbanic
Parallel Computing Scientist
Pittsburgh Supercomputing Center

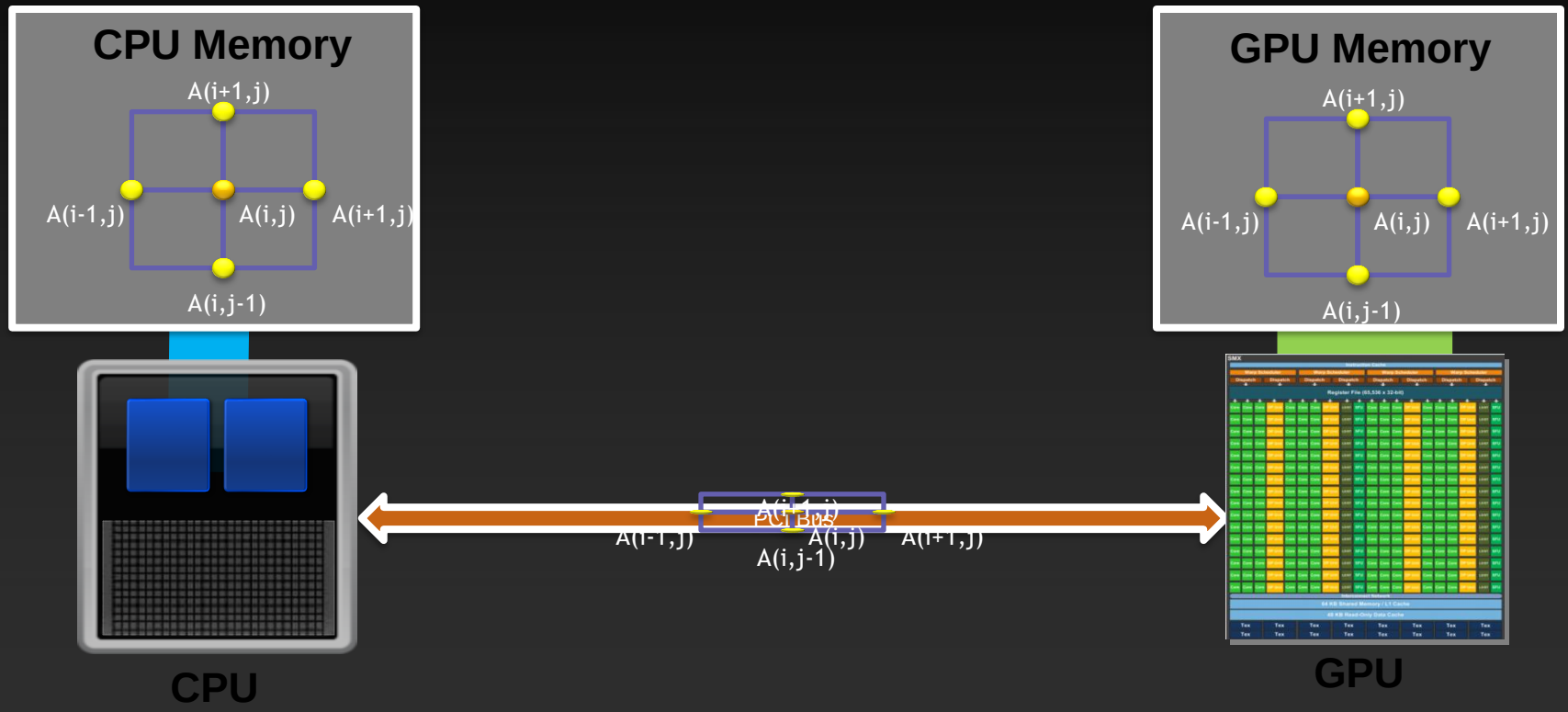
Classic OpenMP

OpenMP was designed to replace low-level and tedious multi-threaded programming 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



GPUs are not CPUs

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

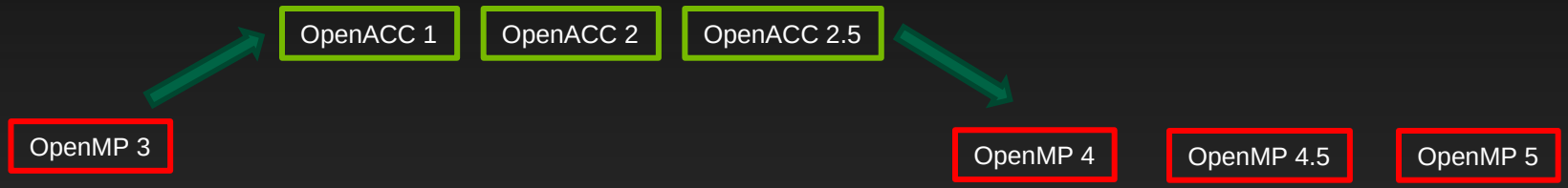
GPUs have thousands of cores.

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



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

Meanwhile...

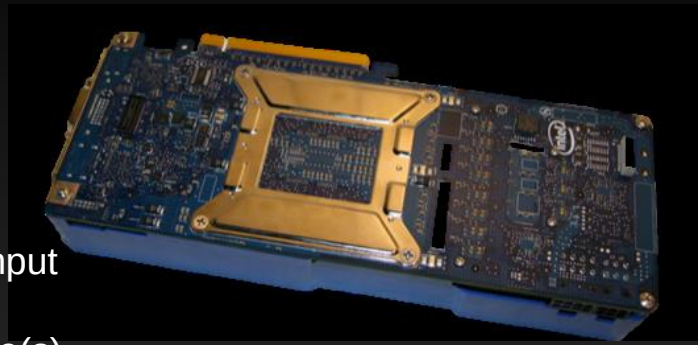
Since the days of RISC vs. CISC, Intel has mastered the art of figuring out what is important about a new processing technology and saying “why can’t we do this in x86?”

The Intel Many Integrated Core (MIC) architecture is about large die, simpler circuit, and much more parallelism, in the x86 line.

Courtesy Dan Stanzione, TACC

What ~~is~~ was MIC?

Basic Design Ideas:

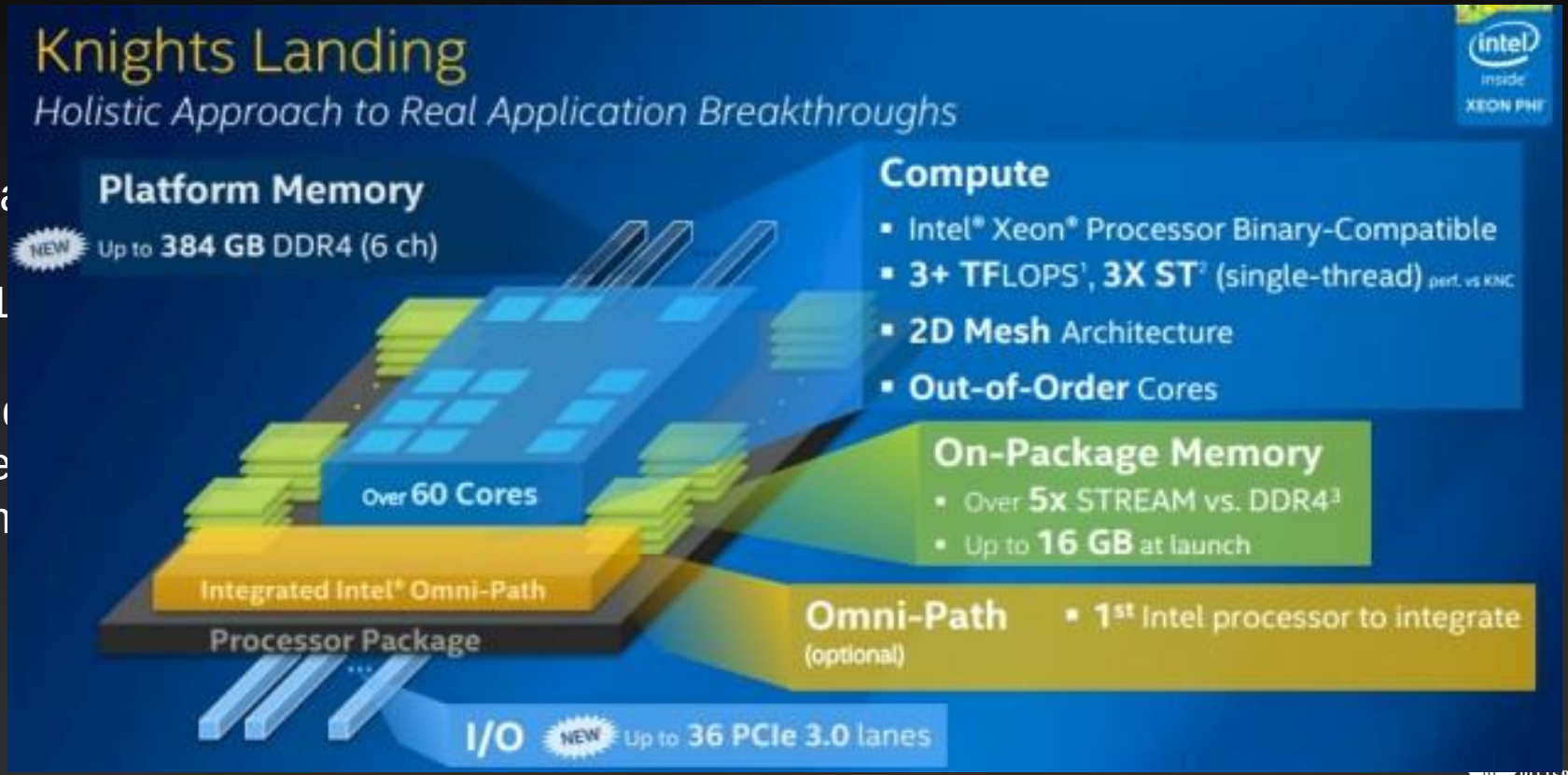


- Leverage x86 architecture (a CPU with many cores)
- Use x86 cores that are simpler, but allow for more compute throughput
- Leverage existing x86 programming models
- Dedicate much of the silicon to floating point ops., keep some cache(s)
- Keep cache-coherency protocol
- Increase floating-point throughput 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

Courtesy Dan Stanzione, TACC

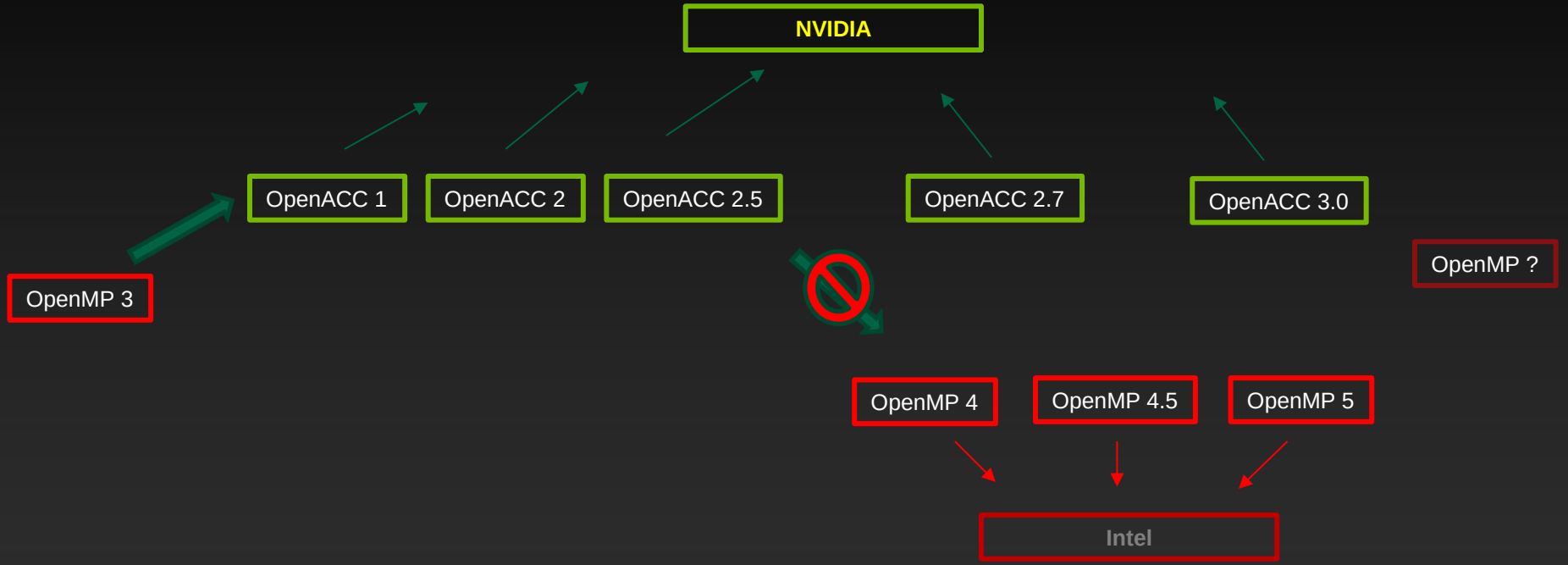
~~Latest~~ last MIC Architecture

- Ma
- L1
- Bi
  ne
  an



Courtesy Dan Stanzione, TACC

Implications for the OpenMP/OpenACC Merge



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

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

- map(
- map(
- map(
- map(
- map(
- map(

OpenACC

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	OpenACC
target	parallel / kernels
teams	parallel / kernels
distribute	loop gang
parallel	parallel / kernels
for / do	loop worker or loop gang
simd	loop vector
is_device_ptr(...)	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

```
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];
     }
}
}
```

Courtesy Christian Terboven

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]);
```

First two examples
Courtesy Christian Terboven

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)
```

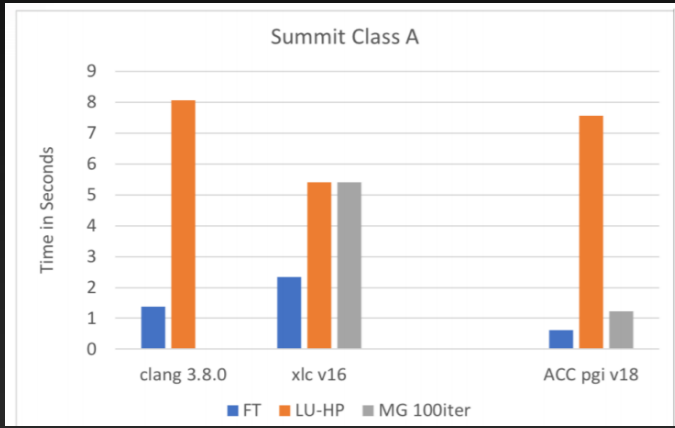
GCC C Compiler (v6.1))

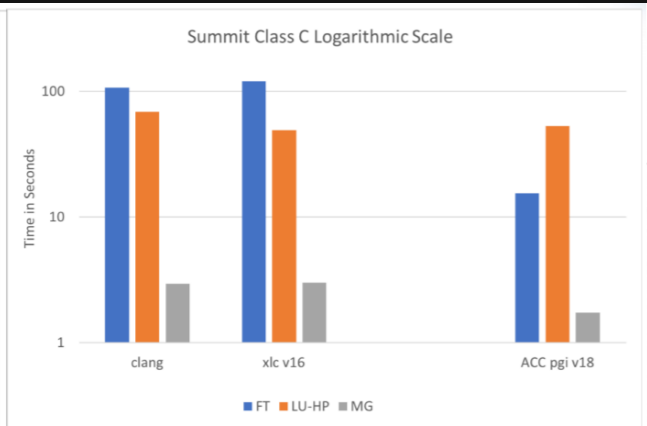
```
#pragma omp target teams distribute \
        parallel for
for(int ii = 0; ii < y; ++ii)
```

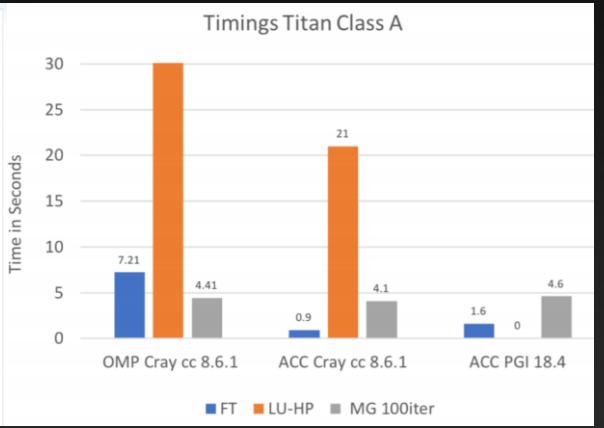
Subtle and confusing? You bet.
For a nice discussion of these examples visit their authors at
https://www.openmp.org/wp-content/uploads/Matt_openmp-booth-talk.pdf

Recent 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 the very obsolete Phi Knights Corner or Knights Landing, you are probably going to be using the Intel OpenMP 4+ release.

- If you are using NVIDIA GPUs, you should probably 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.

The national labs have decided to deal with this by adding an additional layer of abstraction that will translate to the most usable lower level API with frameworks such as oneAPI (Includes DPC++ and extends SYCL) or Kokkos or Raja. Lampson's Law at work!