

# OpenMP 4.0 (and now 5.0)

John Urbanic

Parallel Computing Scientist  
Pittsburgh Supercomputing Center

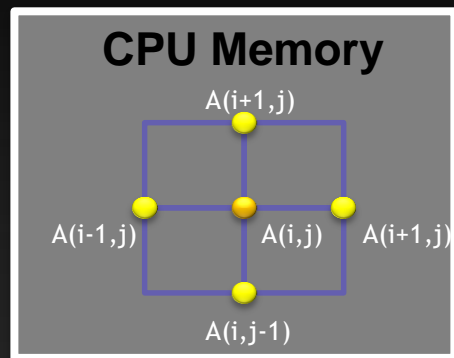
# Classic OpenMP

OpenMP was designed to replace low-level and tedious solutions like POSIX threads, or Pthreads.

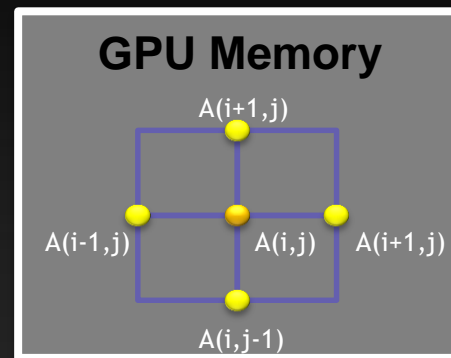
OpenMP was originally targeted towards controlling capable and completely independent processors, with shared memory. The most common such configurations today are the many multi-cored chips we all use. You might have dozens of threads, each of which takes some time to start or complete.

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.

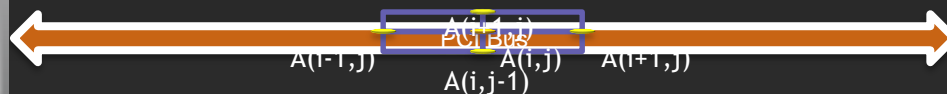
# Then Came This



**CPU**



**GPU**



# GPUs are not CPUs

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

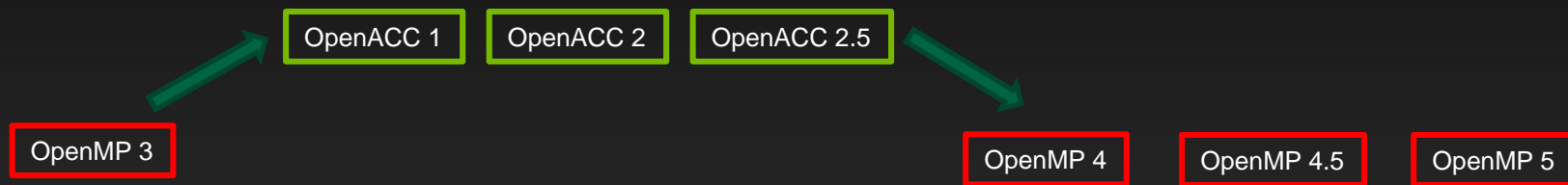
GPUs have thousands of cores.

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?”

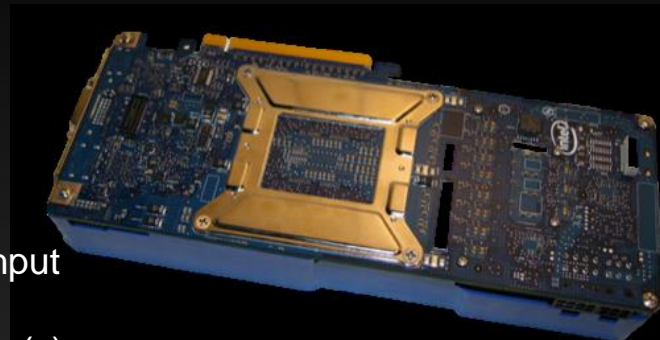
The Intel Many Integrated Core (MIC) architecture is about large die, simpler circuit, and much more parallelism, in the x86 line.



# What is 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



# Latest MIC Architecture

## Knights Landing

*Holistic Approach to Real Application Breakthroughs*



### Platform Memory

NEW

Up to **384 GB** DDR4 (6 ch)

### Compute

- Intel® Xeon® Processor Binary-Compatible
- **3+ TFLOPS<sup>1</sup>, 3X ST<sup>2</sup>** (single-thread) perf. vs KNC
- **2D Mesh** Architecture
- **Out-of-Order** Cores

### On-Package Memory

- Over **5x** STREAM vs. DDR4<sup>3</sup>
- Up to **16 GB** at launch

Over **60 Cores**

Integrated Intel® Omni-Path

Processor Package

**Omni-Path**  
(optional)

- **1<sup>st</sup>** Intel processor to integrate

I/O

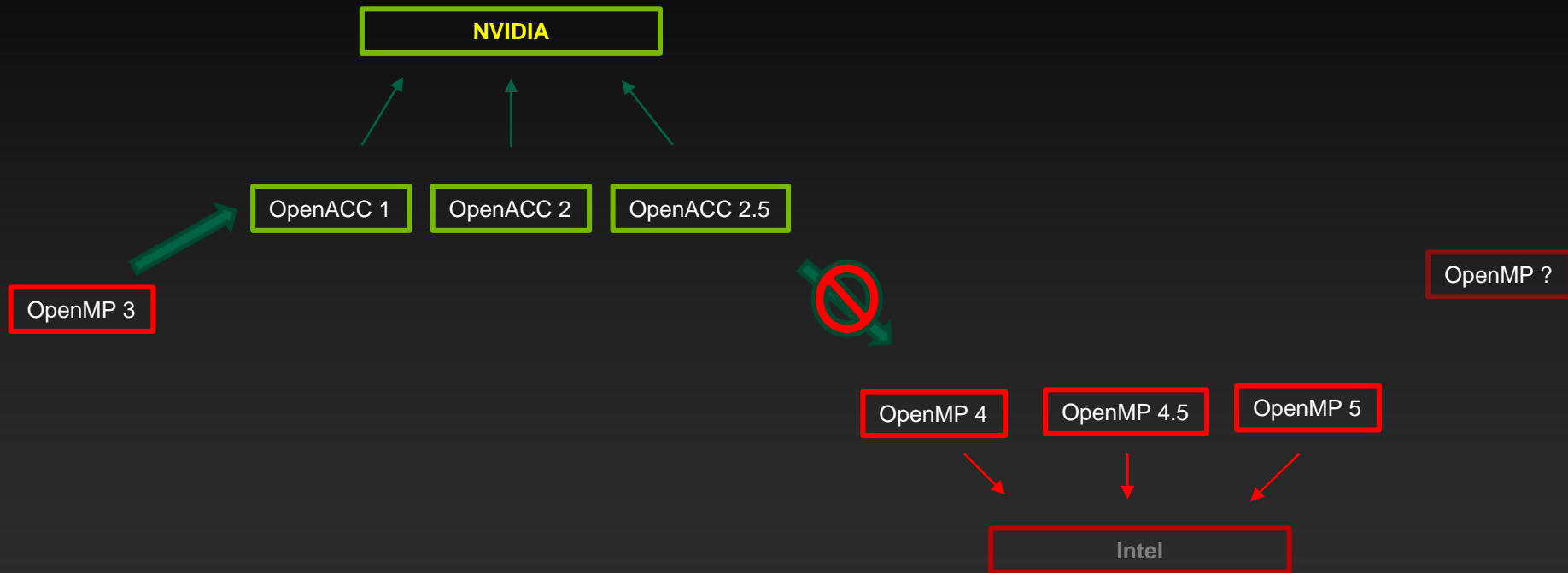
NEW

Up to **36 PCIe 3.0** lanes

- Ma
- L1
- Bi
- ne
- an



# Implications for the OpenMP/OpenACC Merge



Intel and NVIDIA have both influenced their favored approach 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(to:...)`
- `map(from:...)`
- `map(tofrom:...)`
- `map(alloc:...)`
- `map(release:...)`
- `map(delete:...)`

## OpenACC

- `copyin(...)`
- `copyout(...)`
- `copy(...)`
- `create(...)`
- `delete(...)`
- `delete(...) finalize`

# 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

```
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(int i = 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(int i = 0; i < n; ++i){
            y[i] = b*x[i] + y[i];
        }
    }
}
```



# 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]);
```

# OpenMP 4.0 Across Architectures

```
#if defined FORCPU
#pragma omp parallel for simd
#elif defined FORKNC
#pragma omp target teams distribute parallel for simd
#elif defined FORGPU
#pragma omp target teams distribute parallel for \
    schedule(static,1)
#elif defined FORKNL
#pragma omp parallel for simd schedule(dynamic)
#endif
for( int j = 0; j < n; ++j )
    x[j] += a*y[j];
```

# 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.
- If you are using NVIDIA GPUs, you are going to be using OpenACC.
- If you are using Knights Landing, you could also hope to treat it as a regular SMP and just use traditional OpenMP with cache mode. (Does this mean OpenMP 4.0 is a little moot?)

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.

# 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.

# OpenMP 5.0

## A New Emphasis

Technical Report 5: Memory Management Support for OpenMP 5.0 shows the real thrust of OpenMP in its hefty 78 pages. It's all about the memory hierarchy, and how to manage it:

**distance**  $\approx$  **near, far** Specifies the relative physical distance of the memory space with respect to the task the request binds to.

**bandwidth**  $\approx$  **highest, lowest** Specifies the relative bandwidth of the memory space with respect to other memories in the system

**latency**  $\approx$  **highest, lowest** Specifies the relative latency of the memory space with respect to other memories in the system.

**location** = see Table 2.6 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.

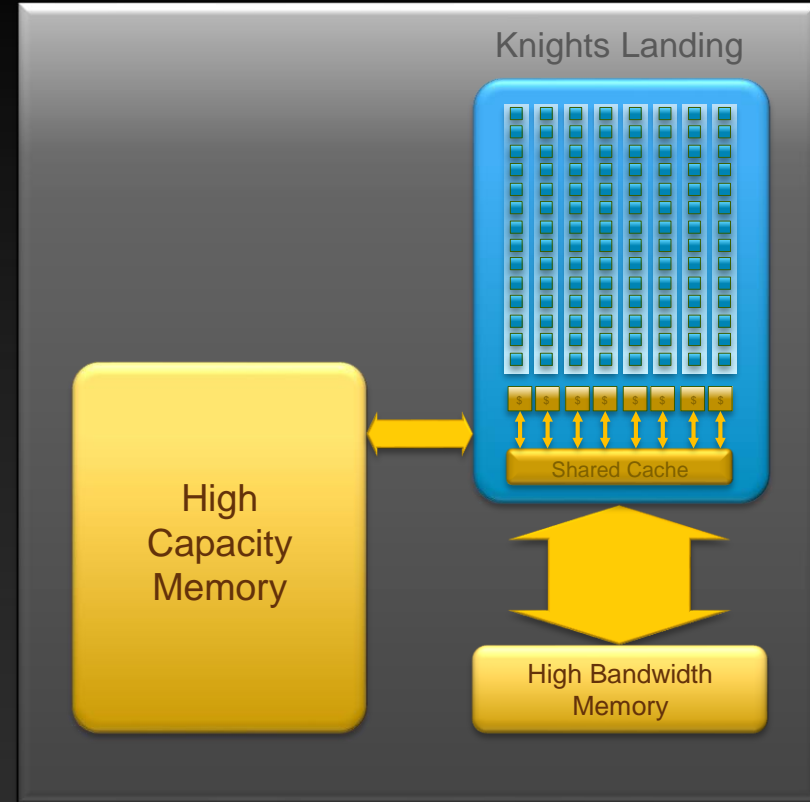
**capacity**  $\geq$  **positive integer** Specifies the physical capacity in bytes of the memory space. **available**  $\geq$  **positive integer** Specifies the current available capacity for new allocations in the memory space.

# 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.

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



# 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)
  - 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