OpenMP 4.0 (now 4.5) for Accelerators

John Urbanic
Parallel Computing Scientist
Pittsburgh Supercomputing Center

OpenACC vs. OpenMP

- OpenMP has a very similar directive philosophy. This is no surprise as OpenACC was started by OpenMP members as an “accelerator development branch” with the idea of merging it back in.
- OpenMP assume(d) that memory movement isn’t an issue, but that thread startup overhead is. The available directives reflect that.
- OpenACC assumes threads are very lightweight, but that data movement onto and off of the accelerator are significant. The directives reflect that.
- But, they are both similar in approach and assume that you, the programmer, are responsible for designating parallelizable loops.
- They are also complementary and can be used together very well.

OpenMP Thread Control Philosophy

OpenMP was traditionally oriented towards controlling fully independent processors. In return for the flexibility to use those processors to their fullest extent, OpenMP assumes that you know what you are doing and does not recognize data dependencies in the same way as OpenACC.

While you override detected data dependencies in OpenACC (with the `independent` clause), there is no such thing in OpenMP. Everything is ***assumed*** to be independent. You must be the paranoid one, not the compiler.

OpenMP assumes that every thread has it's own synchronization control (barriers, locks). 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.

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 very, very lightweight threads.

Intel's MIC Approach

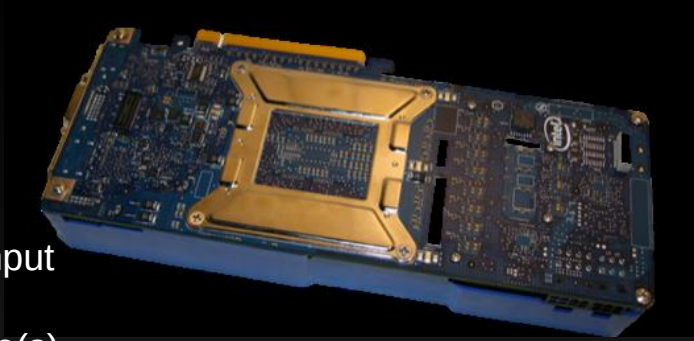
Since the days of RISC vs. CISC, Intel has mastered the art of figuring out what is important about a new processing technology, and saying "why can't we do this in x86?"

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

Courtesy Dan Stanzione, TACC

What is MIC?

Basic Design Ideas:

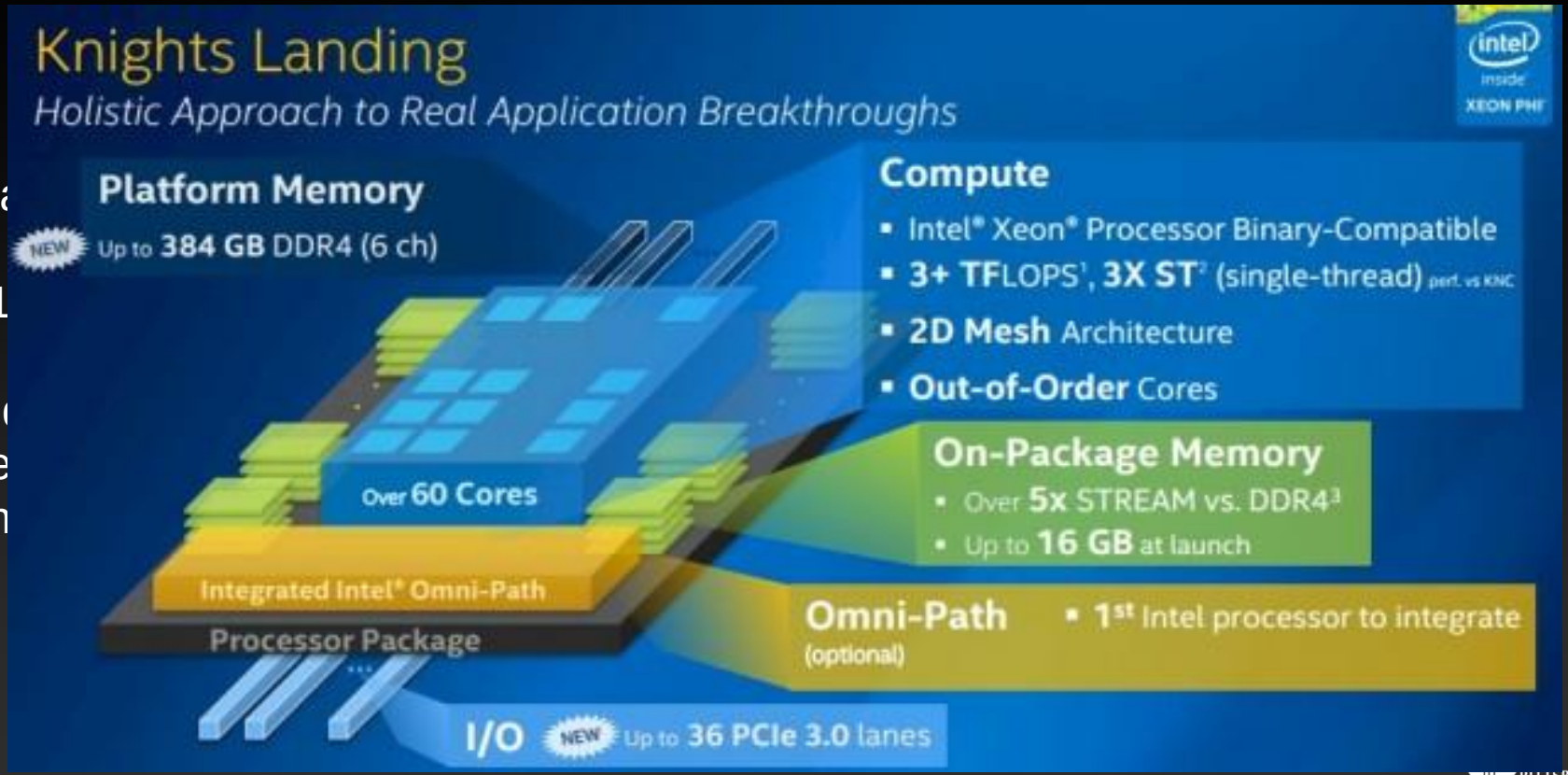


- Leverage x86 architecture (a CPU with many cores)
- Use x86 cores that are simpler, but allow for more compute throughput
- Leverage existing x86 programming models
- Dedicate much of the silicon to floating point ops., keep some cache(s)
- Keep cache-coherency protocol
- Increase floating-point throughput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

Courtesy Dan Stanzione, TACC

MIC Architecture

- Ma
- L1
- Bi
  ne
  an



Courtesy Dan Stanzione, TACC

OpenMP 4.0 Data Migration

OpenMP comes from an SMP multi-core background. The original idea was to avoid the pain of using Unix/Posix pthreads directly. As SMPs have no concept of different memory spaces, OpenMP has not been concerned with that until now. With OpenMP 4.0, that changes. We now have data migration control and related capability like data shaping.

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

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

```
#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];
```

Courtesy Michael Wolfe

Which way to go?

While this might be an interesting discussion of the finer distinctions between these two standards and the future merging thereof, it is not. At the moment, there is a simpler reality:

- OpenMP 4.0 was ratified in July 2013, and it will be a while before it has the widespread support of OpenMP 3. It is currently fully implemented only on Intel compilers for Xeon Phi and partially now in GCC 5.x and better in GCC 6.1. LLVM Clang seems to be on way.

- OpenACC supports Phi with the CAPS compiler, but via an OpenCL back end. PGI has had something "coming" for a while. You would really have to have a good reason to not use the native Intel compiler OpenMP 4.0 at this time.

So, at this time…

- If you are using Phi, you are probably going to be using the Intel OpenMP release.
- If you are using NVIDIA GPU's, 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.

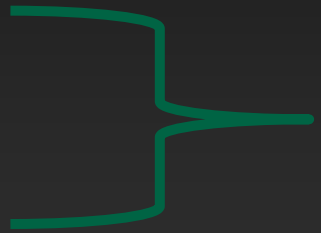
Going Hostless

Both Intel and NVIDIA are converging towards a hostless future.

- Intel
  - Plug a bunch of MICs (Knights Landing) into backplanes
  - Programming model doesn't really change
- NVIDIA
  - Expanding MIMD capability of hardware with each generation
  - CUDA evolving towards remote data access with each version
  - Adding CPU on board

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)
- DirectCompute (Microsoft)
  - Not HPC oriented
- C++ AMP (MS/AMD)
- TBB (Intel C++ template library)
- Cilk (Intel, now in a gcc branch)



Very C++ for threads