OpenMP 4.0 (now 4.5) for Accelerators

John Urbanic
Parallel Computing Scientist
Pittsburgh Supercomputing Center

Copyright 2017
What is Heterogeneous Programming?

Application Code

Compute-Intensive Functions
20% of Code
Most of compute time.

Rest of Sequential
CPU Code

CPU

Accelerator
Have to Move Data

CPU Memory

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

GPU Memory

A(i, j)
A(i, j-1)
A(i-1, j)
A(i+1, j)
OpenMP Thread Control Philosophy

OpenMP was originally targeted towards controlling completely 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. This also partly reflects 1990’s vs. 2010’s compiler expectations.

OpenMP assumes that every thread has its 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 such as no `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.
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

Courtesy Dan Stanzione, TACC
Latest MIC Architecture

- Many cores on the die
- L1 and L2 cache
- Bidirectional ring network for L2 Memory and PCIe connection

Courtesy Dan Stanzione, TACC
Still have a bottleneck to manage.

Or maybe not?

Does cache mode take the pain away?

Small and fast, or large and slow, we still have to pay attention to data locality.
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)
```
<table>
<thead>
<tr>
<th>OpenMP</th>
<th>OpenACC</th>
</tr>
</thead>
<tbody>
<tr>
<td>target data</td>
<td>data</td>
</tr>
<tr>
<td>target enter data</td>
<td>enter data</td>
</tr>
<tr>
<td>target exit data</td>
<td>exit data</td>
</tr>
<tr>
<td>target update</td>
<td>update</td>
</tr>
<tr>
<td>declare target</td>
<td>declare</td>
</tr>
</tbody>
</table>
# OpenMP vs. OpenACC Data Clauses

<table>
<thead>
<tr>
<th>OpenMP</th>
<th>OpenACC</th>
</tr>
</thead>
<tbody>
<tr>
<td>map(to:...)</td>
<td>copyin(...)</td>
</tr>
<tr>
<td>map(from:...)</td>
<td>copyout(...)</td>
</tr>
<tr>
<td>map(tofrom:...)</td>
<td>copy(...)</td>
</tr>
<tr>
<td>map(alloc:...)</td>
<td>create(...)</td>
</tr>
<tr>
<td>map(release:...)</td>
<td>delete(...)</td>
</tr>
<tr>
<td>map(delete:...)</td>
<td>delete(...) finalize</td>
</tr>
</tbody>
</table>
## OpenMP vs. OpenACC Compute Constructs

<table>
<thead>
<tr>
<th>OpenMP</th>
<th>OpenACC</th>
</tr>
</thead>
<tbody>
<tr>
<td>target</td>
<td>parallel / kernels</td>
</tr>
<tr>
<td>teams</td>
<td>parallel / kernels</td>
</tr>
<tr>
<td>distribute</td>
<td>loop gang</td>
</tr>
<tr>
<td>parallel</td>
<td>parallel / kernels</td>
</tr>
<tr>
<td>for / do</td>
<td>loop worker or loop gang</td>
</tr>
<tr>
<td>simd</td>
<td>loop vector</td>
</tr>
<tr>
<td>is_device_ptr(...)</td>
<td>deviceptr(...)</td>
</tr>
</tbody>
</table>
## OpenMP vs. OpenACC Differences

<table>
<thead>
<tr>
<th>OpenMP</th>
<th>OpenACC</th>
</tr>
</thead>
<tbody>
<tr>
<td>device(n)</td>
<td>---</td>
</tr>
<tr>
<td>depend(to:a)</td>
<td>async(n)</td>
</tr>
<tr>
<td>depend(from:b)</td>
<td>async(n)</td>
</tr>
<tr>
<td>nowait</td>
<td>async</td>
</tr>
<tr>
<td>loops, tasks, sections</td>
<td>loops</td>
</tr>
<tr>
<td>atomic</td>
<td>atomic</td>
</tr>
<tr>
<td>master, single, critical, barrier, lock, ordered, flush, cancel</td>
<td>---</td>
</tr>
</tbody>
</table>
int main(int argc, const char* argv[]) {
    int n = 10240; float a = 2.0f; float b = 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];
        }
    }
}

Courtesy Christian Terboven
Comparing OpenACC with OpenMP 4.0 on NVIDIA & Phi

**OpenMP 4.0 for Intel Xeon Phi**

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

First two examples

Courtesy Christian Terboven

**OpenMP 4.0 for NVIDIA GPU**

```c
#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**

```c
#pragma acc kernels
for (i=0; i<N; ++i)
  B[i] += sin(B[i]);
```
#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];
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 and with Knights Landing, maybe even longer. 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 I haven’t tested. 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 Knights Corner, 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 Phi Knights Landing, you may 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.

```c
#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++) {
        }
    }
}
```

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.
2) Indicate that a function is vectorizable.

```c
#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.
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