GPU TECHNOLOGY CONFERENCE
April 4-7, 2016 | Silicon Valley

S6410 - Comparing OpenACC 2.5 and OpenMP 4.5

James Beyer, NVIDIA
Jeff Larkin, NVIDIA

GTC16 – April 7, 2016

PRESENTED BY
NVIDIA.

AGENDA

History of OpenMP & OpenACC

Philosophical Differences

Technical Differences

Portability Challenges

Conclusions

A Tale of Two Specs.

A Brief History of OpenMP

1996 - Architecture Review Board (ARB) formed by several vendors implementing their own directives for Shared Memory Parallelism (SMP).

1997 - 1.0 was released for C/C++ and Fortran with support for parallelizing loops across threads.

2000, 2002 – Version 2.0 of Fortran, C/C++ specifications released.

2005 – Version 2.5 released, combining both specs into one.

2008 – Version 3.0 released, added support for tasking

2011 – Version 3.1 release, improved support for tasking

2013 – Version 4.0 released, added support for offloading (and more)

2015 – Version 4.5 released, improved support for offloading targets (and more)

A Brief History of OpenACC

2010 – OpenACC founded by CAPS, Cray, PGI, and NVIDIA, to unify directives for accelerators being developed by CAPS, Cray, and PGI independently

2011 – OpenACC 1.0 released

2013 – OpenACC 2.0 released, adding support for unstructured data management and clarifying specification language

2015 – OpenACC 2.5 released, contains primarily clarifications with some additional features.

Philosophical Differences

OpenMP: Compilers are dumb, users are smart. Restructuring non-parallel code is optional.

OpenACC: Compilers can be smart and smarter with the user's help. Non-parallel code must be made parallel.

Philosophical Differences

OpenMP:

The OpenMP API covers only user-directed parallelization, wherein ***the programmer explicitly specifies*** the actions to be taken by the compiler and runtime system in order to execute the program in parallel.

The OpenMP API ***does not cover*** compiler-generated automatic parallelization and directives to the compiler to assist such parallelization.

OpenACC:

The programming model allows the programmer to ***augment information available to the compilers***, including specification of data local to an accelerator, ***guidance on mapping of loops*** onto an accelerator, and similar performance-related details.

Philosophical Trade-offs

OpenMP

Consistent, predictable behavior between implementations

Users can parallelize non-parallel code and protect data races explicitly

Some optimizations are off the table

Substantially different architectures require substantially different directives.

OpenACC

Quality of implementation will greatly affect performance

Users must restructure their code to be parallel and free of data races

Compiler has more freedom and information to optimize

High level parallel directives can be applied to different architectures by the compiler

Technical Differences

Parallel: Similar, but Different

OMP Parallel

Creates a *team* of *threads*

Very well-defined how the number of threads is chosen.

May synchronize within the team

Data races are the user's responsibility

ACC Parallel

Creates 1 or more *gangs* of *workers*

Compiler free to choose number of gangs, workers, vector length

May not synchronize between gangs

Data races not allowed

OMP Teams vs. ACC Parallel

OMP Teams

Creates a *league* of 1 or more *thread teams*

Compiler free to choose number of teams, threads, and simd lanes.

May not synchronize between teams

Only available within target regions

ACC Parallel

Creates 1 or more *gangs* of *workers*

Compiler free to choose number of gangs, workers, vector length

May not synchronize between gangs

May be used anywhere

Compiler-Driven Mode

OpenMP

Fully user-driven (no analogue)

Some compilers choose to go above and beyond after applying OpenMP, but not guaranteed

OpenACC

`Kernels` directive declares desire to parallelize a region of code, but places the burden of analysis on the compiler

Compiler required to be able to do analysis and make decisions.

Loop: Similar but Different

OMP Loop (For/Do)

Splits (“*Workshares*”) the iterations of the next loop to threads in the team, guarantees the user has managed any data races

Loop will be run over threads and scheduling of loop iterations may restrict the compiler

ACC Loop

Declares the loop iterations as independent & race free (parallel) or interesting & should be analyzed (kernels)

User able to declare independence w/o declaring scheduling

Compiler free to schedule with gangs/workers/vector, unless overridden by user

Distribute vs. Loop

OMP Distribute

Must live in a `TEAMS` region

Distributes loop iterations over 1 or more thread teams

Only master thread of each team runs iterations, until `PARALLEL` is encountered

Loop iterations are implicitly independent, but some compiler optimizations still restricted

ACC Loop

Declares the loop iterations as independent & race free (parallel) or interesting & should be analyzed (kernels)

Compiler free to schedule with gangs/workers/vector, unless overridden by user

Distribute Example

```
#pragma omp target teams
{
#pragma omp distribute
  for(i=0; i<n; i++)
    for(j=0;j<m;j++)
      for(k=0;k<p;k++)
}
```

```
#pragma acc parallel
{
#pragma acc loop
  for(i=0; i<n; i++)
#pragma acc loop
    for(j=0;j<m;j++)
#pragma acc loop
      for(k=0;k<p;k++)
}
```

Distribute Example

```
#pragma omp target teams
{
#pragma omp distribute
  for(i=0; i<n; i++)
    for(j=0;j<m;j++)
      for(k=0;k<p;k++)
}
```

Generate a 1 or more thread teams

Distribute “i” over teams.

No information about “j” or “k” loops

```
#pragma acc parallel

#pragma acc loop
=0; i<n; i++)
#pragma acc loop
(j=0;j<m;j++)
#pragma acc loop
      for(k=0;k<p;k++)
}
```

Distribute Example

```
#pragma omp target teams
{
#pragma omp distribute
  for(i=0; i<n; i
    for(j=0;j<m;j
      for(k=0;k<p;k++)
}
```

Generate a 1 or more gangs

These loops are independent, do the *right thing*

```
#pragma acc parallel
{
#pragma acc loop
  for(i=0; i<n; i++)
#pragma acc loop
    for(j=0;j<m;j++)
#pragma acc loop
      for(k=0;k<p;k++)
}
```

Distribute Example

```
#pragma omp target teams
{
#pragma omp distribute
  for(i=0; i<n; i++)
    for(j=0;j<m;j++)
      for(k=0;k<p;k++)
}
```

What's the *right thing?*

Interchange? Distribute? Workshare?
Vectorize? Stripmine? Ignore? …

```
#pragma acc parallel
{
#pragma acc loop
  for(i=0; i<n; i++)
#pragma acc loop
    for(j=0;j<m;j++)
#pragma acc loop
      for(k=0;k<p;k++)
}
```

Synchronization

OpenMP

Users may use barriers, critical regions, and/or locks to protect data races

It's possible to parallelize non-parallel code

OpenACC

Users expected to refactor code to remove data races.

Code should be made truly parallel and scalable

Synchronization Example

```
#pragma omp parallel private(p)
{
  funcA(p);
#pragma omp barrier
  funcB(p);
}
```

```
function funcA(p[N]){
  #pragma acc parallel
}
function funcB(p[N]){
  #pragma acc parallel
}
```

Synchronization Example

```
#pragma omp parallel for
for (i=0; i<N; i++)
{
#pragma omp critical
  A[i] = rand();
  A[i] *= 2;
}
```

```
parallelRand(A);
#pragma acc parallel loop
for (i=0; i<N; i++)
{
  A[i] *= 2;
}
```

Portability Challenges

How to Write Portable Code (OMP)

```
#ifdef GPU
#pragma omp target omp teams distribute parallel for reduction(max:error) \
            collapse(2) schedule(static,1)
#elif defined(CPU)
#pragma omp parallel for reduction(max:error)
#elif defined(SOMETHING_ELSE)
#pragma omp …
endif
        for( int j = 1; j < n-1; j++)
        {
#if defined(CPU) && defined(USE_SIMD)
#pragma omp simd
#endif
            for( int i = 1; i < m-1; i++ )
            {
                Anew[j][i] = 0.25 * ( A[j][i+1] + A[j][i-1]
                                    + A[j-1][i] + A[j+1][i]);
                error = fmax( error, fabs(Anew[j][i] - A[j][i]));
            }
        }
```

Ifdefs can be used to choose particular directives per device at compile-time

How to Write Portable Code (OMP)

```
#pragma omp \
#ifdef GPU
target teams distribute \
#endif
parallel for reduction(max:error) \
#ifdef GPU
collapse(2) schedule(static,1)
endif
        for( int j = 1; j < n-1; j++)
        {
            for( int i = 1; i < m-1; i++ )
            {
                Anew[j][i] = 0.25 * ( A[j][i+1] + A[j][i-1]
                                    + A[j-1][i] + A[j+1][i]);
                error = fmax( error, fabs(Anew[j][i] - A[j][i]));
            }
        }
```

Creative ifdefs might clean up the code, but still one target at a time.

How to Write Portable Code (OMP)

```
usegpu = 1;
#pragma omp target teams distribute parallel for reduction(max:error) \
#ifdef GPU
collapse(2) schedule(static,1) \
endif
if(target:usegpu)
        for( int j = 1; j < n-1; j++)
        {
            for( int i = 1; i < m-1; i++ )
            {
                Anew[j][i] = 0.25 * ( A[j][i+1] + A[j][i-1]
                                    + A[j-1][i] + A[j+1][i]);
                error = fmax( error, fabs(Anew[j][i] - A[j][i]));
            }
        }
```

The OpenMP if clause can help some too (4.5 improves this).

Note: This example assumes that a compiler will choose to generate 1 team when not in a target, making it the same as a standard “parallel for.”

How to Write Portable Code (ACC)

```
#pragma acc kernels
{
        for( int j = 1; j < n-1; j++)
        {
            for( int i = 1; i < m-1; i++ )
            {
                Anew[j][i] = 0.25 * ( A[j][i+1] + A[j][i-1]
                                    + A[j-1][i] + A[j+1][i]);
                error = fmax( error, fabs(Anew[j][i] - A[j][i]));
            }
        }
}
```

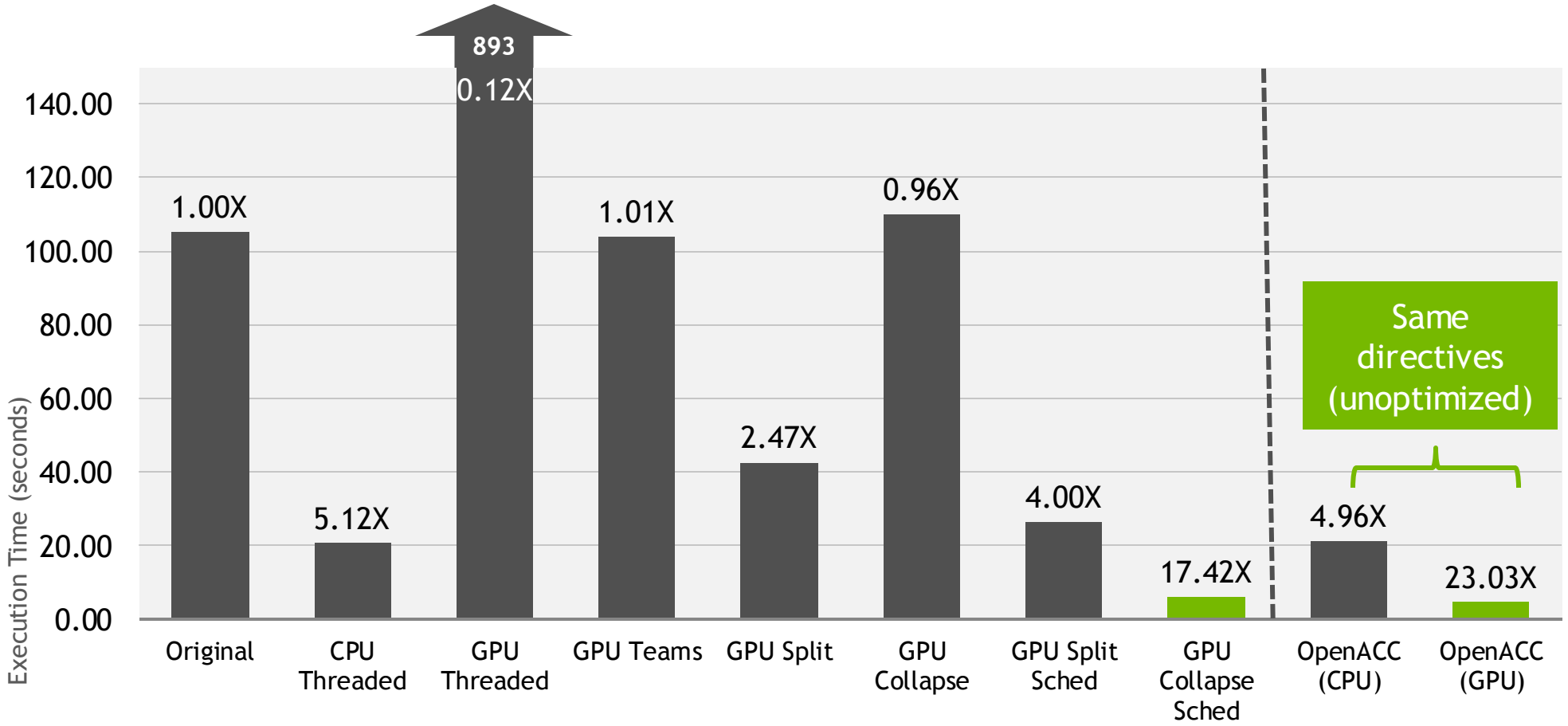
Developer presents the desire to parallelize to the compiler, compiler handles the rest.

How to Write Portable Code (ACC)

```
#pragma acc parallel loop reduction(max:error)
{
        for( int j = 1; j < n-1; j++)
        {
#pragma acc loop reduction(max:error)
            for( int i = 1; i < m-1; i++ )
            {
                Anew[j][i] = 0.25 * ( A[j][i+1] + A[j][i-1]
                                    + A[j-1][i] + A[j+1][i]);
                error = fmax( error, fabs(Anew[j][i] - A[j][i]));
            }
        }
}
```


Developer asserts the parallelism of the loops to the compiler, compiler makes decision about scheduling.

Execution Time (Smaller is Better)



NVIDIA Tesla K40, Intel Xeon E5-2690 v2 @ 3.00GHz - See GTC16 S6510 for additional information

Compiler Portability (CPU)

OpenMP

Numerous well-tested implementations

- PGI, IBM, Intel, GCC, Cray, …

OpenACC

CPU implementations beginning to emerge

- X86: PGI
- ARM: PathScale
- Power: Coming soon

Compiler Portability (Offload)

OpenMP

Few mature implementations

- Intel (Phi)
- Cray (GPU, *Phi?*)
- GCC (Phi, GPUs in development)
- Clang (Multiple targets in development)

OpenACC

Multiple mature implementations

- PGI (NVIDIA & AMD)
- PathScale (NVIDIA & AMD)
- Cray (NVIDIA)
- GCC (in development)

Conclusions

Conclusions

OpenMP & OpenACC, while similar, are still quite different in their approach

Each approach has clear tradeoffs with no clear *"winner"*

It should be possible to translate between the two, but the process may not be automatic