COMPARING OPENACC AND OPENMP PERFORMANCE AND PROGRAMMABILITY

JEFF LARKIN, NVIDIA

GUIDO JUCKELAND, TECHNISCHE UNIVERSITÄT DRESDEN
AGENDA

- OpenACC & OpenMP Overview
- Case Studies
- SPECaccel Lessons Learned
- Final Thoughts
This talk is not intended to reveal that OpenX is strictly better than OpenY.

The purpose of this talk is to highlight differences between both specifications in relation to accelerators.
ALSO IMPORTANT

- We expected compilers supporting both OpenMP4 and OpenACC on the same device to make apples/apples comparisons, they were not available in time.
- Instead we are showing our best interpretation of how a compliant OpenMP compiler would build these kernels. Actual compiler performance will vary.
OPENACC & OPENMP OVERVIEW
OPENACC 2.0

- OpenACC is a specification for high-level, compiler directives for expressing parallelism for accelerators.
  - Abstract accelerator model
  - Performance Portability is primary concern
- 1.0: November 2011
- 2.0: June 2013
OPENMP 4.0

- OpenMP formed in 1997, focus on vendor-neutral Shared Memory Parallelism
- OpenMP 4.0: 2013
  - Expanded focus beyond shared memory parallel computers, including accelerators.
- The OpenMP 4.0 target construct provides the means to offload data and computation to accelerators.
CASE STUDY: DAXPY
The OpenACC parallel loop construct informs the compiler that all loop iterations are independent.

The compiler is free to parallelize as it sees fit for the hardware.

The PGI compiler will default to using blocks of 256 threads and enough blocks to complete N.

```c
!$acc parallel loop present(x,y)
do i=1,n
    y(i) = a*x(i) + y(i)
enddo
```
PARALLEL DO dictates the following:

- **A team of threads** is created
- The following for loop is distributed to those threads
- A static schedule is most common, with each thread getting \( \frac{N}{\text{NumThreads}} \) contiguous iterations

```c
!$omp parallel do
don i=1,n
    y(i) = a*x(i) + y(i)
enddo
```
OPENMP DAXPY: TARGET PARALLEL DO

TARGET PARALLEL DO dictates the following:

- Offload data and execution to the target device
- Use standard PARALLEL DO semantics once on the device
- Because threads can synchronize, the team must live within a thread block.
- Assumption: Static schedule with standard N/NTHREADS chunking

length = n / blockDim
start = (threadIdx%x - 1) * length + 1
finish = start + length - 1
do i = start,finish
  if ( i.le.n ) y(i) = a * x(i) + y(i)
enddo
OPENMP: TARGET PARALLEL DO INTERLEAVED

`length = n / blockDim%x`

do i = threadIdx%x,n,length
  if ( i.le.n ) y(i) = a * x(i) + y(i)
endo

- The standard static schedule used in the previous experiment results in poor memory coalescing.
- Interleaving iterations using a schedule(static,1) clause would correct this.
- The SIMD directive may be able to achieve the same thing.
- Still running in 1 thread block.
OPENMP: TARGET TEAMS DISTRIBUTE PARALLEL DO

```c
!$omp target teams distribute parallel do
do i=1,n
  y(i) = a*x(i) + y(i)
enddo
```

This directive instructs:
- Offload data and execution to the target device.
- Create a *league of teams*
- Distribute the loop across those teams
- Use PARALLEL DO to parallelize within the teams
- The number of teams to use and threads within those teams is implementation defined.
- This would probably work like the acc parallel loop
DAXPY TAKEAWAYS

- ACC PARALLEL LOOP expresses the parallelism and the compiler decides how to exploit it.
- TARGET PARALLEL DO is not sufficient for GPUs
  - In simple cases such as this, the compiler *might* detect the lack of synchronization and then *might* ignore worksharing rules if it believes it’s safe, this is not technically compliant though. (Does that matter?)
- TARGET TEAMS DISTRIBUTE PARALLEL DO (SIMD) is more portable
  - Using 1 team is both legal and equivalent to a simple PARALLEL DO
  - If the developer specifies the number of teams, threads, or simd length it becomes less portable.
CASE STUDY: ASYNCHRONOUS PROGRAMMING
OPENACC ASYNC/WAIT

- OpenACC handles asynchronicity between the device and host using ASYNC queues and WAIT directives.

```plaintext
#pragma acc parallel loop async(block)
...
#pragma acc update self(A[start:count]) async(block)
#pragma acc wait
```

- This technique maps simply to CUDA streams
OpenMP already had the TASK and TASKWAIT directives prior to 4.0 and 4.0 added task dependencies. In 4.0 these are used for asynchronous behavior.

- Task dependencies are more expressive than OpenACC async queues, but requires the CPU to resolve dependencies and start tasks.

```plaintext
#pragma omp task depend(inout:A)
{
    #pragma omp target teams distribute parallel for
}
#pragma omp task depend(in:A)
{
    #pragma omp target update host(A)
}
```

- As much as possible, back-to-back target directives should be fused into the same task to avoid involving the CPU in resolving dependencies.
Because resolving OpenMP 4.0 tasks requires the CPU, which could introduce unnecessary delays issuing work to the GPU, OpenMP 4.1 simplifies asynchronous target operations.

- TARGET constructs are now implicitly TASKS and accept DEPEND clauses.
- TARGET constructs are made asynchronous with a NOWAIT clause.

```c
#pragma omp target teams distribute \ parallel for nowait depend(inout:A)
#pragma omp target update host(A) nowait depend(in:A)
#pragma taskwait
```
WHY 4.1 NOWAIT IS BETTER THAN TASK

CPU must get involved to resolve each task before sending work to GPU.

CPU can enqueue work to GPU streams to squeeze out idle time.
ASYNCHRONOUS TAKEAWAYS

- OpenACC ASYNC/WAIT map nicely to CUDA streams
- OpenMP 4.0 TASK dependencies
  - More expressive than async queues
  - Require the CPU to resolve
- OpenMP 4.1 NOWAIT
  - Provides existing TASK dependencies
  - Removes requirements for CPU resolution

- Both models are compatible with OpenMP tasks
Comparing Apples and Oranges

Using SPECaccel as a Yardstick

Guido Juckeland (guido.juckeland@tu-dresden.de)
SPEC Accel provides a comparative performance measure of

- Hardware Accelerator devices (GPU, Co-processors, etc.)
- Supporting software tool chains (Compilers, Drivers, etc.)
- Host systems and accelerator interface (CPU, PCIe, etc.)

Computationally-intensive parallel High Performance Computing (HPC) applications, benchmarks, and mini-apps

Portable across multiple accelerators

Two distinct suites

- OpenACC v1.0
- OpenCL v1.1
<table>
<thead>
<tr>
<th>Benchmarks</th>
<th>Language</th>
<th>Origin</th>
<th>Application Domain</th>
</tr>
</thead>
<tbody>
<tr>
<td>303.ostencil</td>
<td>C</td>
<td>Parboil, University of Illinois</td>
<td>Thermodynamics</td>
</tr>
<tr>
<td>304.olbm</td>
<td>C</td>
<td>Parboil, University of Illinois, SPEC CPU2006</td>
<td>Computational Fluid Dynamics, Lattice Boltzmann</td>
</tr>
<tr>
<td>314.omriq</td>
<td>C</td>
<td>Rodinia, University of Virginia</td>
<td>Medicine</td>
</tr>
<tr>
<td>350.md</td>
<td>Fortran</td>
<td>Indiana University</td>
<td>Molecular Dynamics</td>
</tr>
<tr>
<td>351.palm</td>
<td>Fortran</td>
<td>Leibniz University of Hannover</td>
<td>Large-eddy simulation, atmospheric turbulence</td>
</tr>
<tr>
<td>352.ep</td>
<td>C</td>
<td>NAS Parallel Benchmarks (NPB)</td>
<td>Embarrassingly Parallel</td>
</tr>
<tr>
<td>353.clvrleaf</td>
<td>C, Fortran</td>
<td>Atomic Weapons Establishment (AWE)</td>
<td>Explicit Hydrodynamics</td>
</tr>
<tr>
<td>354.cg</td>
<td>C</td>
<td>NPB</td>
<td>Conjugate Gradient Solver</td>
</tr>
<tr>
<td>355.seismic</td>
<td>Fortran</td>
<td>GeoDynamics.org, University of Pau</td>
<td>Seismic Wave Modeling (PDE)</td>
</tr>
<tr>
<td>356.sp</td>
<td>Fortran</td>
<td>NPB</td>
<td>Scalar Penta-diagonal solver</td>
</tr>
<tr>
<td>357.csp</td>
<td>C</td>
<td>NPB</td>
<td>Scalar Penta-diagonal solver</td>
</tr>
<tr>
<td>359.miniGhost</td>
<td>C, Fortran</td>
<td>Sandia National Lab</td>
<td>Finite difference</td>
</tr>
<tr>
<td>360.ilbdc</td>
<td>Fortran</td>
<td>SPEC OMP2012</td>
<td>Fluid Mechanics</td>
</tr>
<tr>
<td>363.swim</td>
<td>Fortran</td>
<td>SPEC OMP2012</td>
<td>Weather</td>
</tr>
<tr>
<td>370.bt</td>
<td>C</td>
<td>NPB</td>
<td>Block Tridiagonal Solver for 3D PDE</td>
</tr>
</tbody>
</table>
# Used Hardware

<table>
<thead>
<tr>
<th></th>
<th>NVIDIA TESLA K40</th>
<th>Intel Xeon Phi</th>
<th>Radeon R9 290X</th>
<th>2x Intel Xeon X5680</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processing Units</td>
<td>2880</td>
<td>61*16 = 976</td>
<td>44<em>4</em>16 = 2816</td>
<td>12*4 = 48</td>
</tr>
<tr>
<td>Taktfrequenz</td>
<td>745 – 875 MHz</td>
<td>1,1 GHz</td>
<td>1,05 GHz</td>
<td>3,33 – 3,6 GHz</td>
</tr>
<tr>
<td>Speicher</td>
<td>12GB GDDR5</td>
<td>8GB GDDR5</td>
<td>4GB GDDR5</td>
<td>12GB DDR3</td>
</tr>
<tr>
<td>Bandbreite</td>
<td>288 GB/s</td>
<td>352 GB/s</td>
<td>346 GB/s</td>
<td>2*32 GB/s = 64 GB/s</td>
</tr>
<tr>
<td>GFLOPS (SP/DP)</td>
<td>4290 / 1430</td>
<td>2150 / 1075</td>
<td>5910 / 740</td>
<td>320 / 160</td>
</tr>
<tr>
<td>TDP</td>
<td>225 W</td>
<td>300 W</td>
<td>300 W</td>
<td>2*130 W = 260 W</td>
</tr>
</tbody>
</table>
OpenACC runtimes

SPEC Accel OpenACC base run

- Memory Limit
- Fixed in new PGI version.

Run time in seconds

- 303.ostencil
- 304.olbm
- 314.omriq
- 350.md
- 351.palm
- 352.ep
- 353.clvleaf
- 354.cg
- 355.seismic
- 356.sp
- 357.csp
- 359.miniGhost
- 360.ilbdc
- 363.swim
- 370.bt

- AMD Radeon R9 290X
- NVIDIA Tesla K40

Guido Juckeland
Converting the OpenACC suite to OpenMP
New with OpenMP 4.0

- target-directives = "offload" pragmas
- Basis: Host maybe with a "Device"
- Start on Host, directives for data- and control transfer
- Target-Directives orthogonal to parallel-directives
- similar to OpenACC
Challenges

Some OpenACC Directives/Clauses translate 1:1…

- acc parallel → omp target teams
- acc loop gang → omp distribute
- acc loop worker → omp parallel loop (!)
- acc loop vector → omp simd (!)
- acc declare → omp declare target
- acc data → omp target data
- acc update → omp target update
- copy/copy_in/copy_out → map(to/from/to/from:…) (!)
Challenges (2)

... some not!
- acc kernels
- acc loop
- omp parallel workshare

**Synchronization different**
- acc parallel: No Barrier between loops
- acc kernels: Implicit barrier between loops possible

**Scalars:**
- OpenACC: implicit „private“
Explicit conversions

```c
#pragma acc kernels
{
    #pragma acc loop worker
    for(i ...){
        tmp = ...;
        array[i] = tmp * ...;
    }
    #pragma acc loop vector
    for(i ...)
        array2[i] = ...;
}
```

```c
#pragma omp target
{
    #pragma omp parallel for private(tmp)
    for(i ...){
        tmp = ...;
        array[i] = tmp * ...;
    }
    #pragma omp simd
    for(i ...)
        array2[i] = ...;
}
```
#pragma acc parallel
{
    #pragma acc loop

    #pragma acc loop
    for (i ...){
        tmp = ...;
        array[i] = tmp * ...;
    }

    #pragma acc loop
    for (i ...)
        array2[i] = ...;
}

#pragma omp target
#pragma omp parallel
{
    #pragma omp for private(tmp) nowait
    for (i ...){
        tmp = ...;
        array[i] = tmp * ...;
    }

    #pragma omp for simd
    for (i ...)
        array2[i] = ...;
}
#pragma acc kernels
{
  for(i ...){
    tmp = ...;
    array[i] = tmp * ...;
  }

  for(i ...)
    array2[i] = ...;
}

#pragma omp target
#pragma omp parallel
{
  #pragma omp for private(tmp)
  for(i ...){
    tmp = ...;
    array[i] = tmp * ...;
  }

  #pragma omp for simd
  for(i ...)
    array2[i] = ...;
}
Copy vs. PCopy

```c
int x[10], y[10];

#pragma acc data copy(x) pcopy(y)
{
    ...
    #pragma acc kernels copy(x) pcopy(y)
    {
        // Accelerator Code
        ...
    }
    ...
}
```

```c
int x[10], y[10];

#pragma omp target data map(x,y)
{
    ...
    #pragma omp target update to(x)
    #pragma omp target map(y)
    {
        // Accelerator Code
        ...
    }
    ...
}
```
Map vs. Update

int foo, bar;
#pragma omp target data map(foo)
{
    // ...
#pragma omp target map(from: foo)
{
    bar = ...; foo = bar;
}
//foo != bar (!!!)
}
//foo == bar

#pragma omp declare target
int foo, bar;
#pragma omp end declare target

int main(…)
{
    // ...
#pragma omp target map(from: foo)
{
    bar = ...; foo = bar;
}
//foo != bar (!!!)
#pragma omp target update(from: foo)
//foo == bar
To Declare Target or not...

```c
#pragma omp declare target
int foo, bar;
#pragma omp end declare target

int main(...) {
    // ...
    #pragma omp target map
    {
        ...
    }
}
```

```c
int foo, bar;

int main(...) {
    // ...
    #pragma omp target
    {
        ...
    }
}
```
### Loop Ordering

<table>
<thead>
<tr>
<th>#pragma acc parallel</th>
<th>#pragma omp target</th>
<th>Differences in OMP: kji:</th>
</tr>
</thead>
<tbody>
<tr>
<td>#pragma acc loop collapse(3)</td>
<td>#pragma omp target</td>
<td>6s</td>
</tr>
<tr>
<td>for(k=0;k&lt;size;k++)</td>
<td>#pragma omp target</td>
<td>ljk: 0.2s</td>
</tr>
<tr>
<td>for(j=0;j&lt;size;j++)</td>
<td>#pragma omp parallel for collapse(2)</td>
<td>for(i=0;i&lt;size;i++)</td>
</tr>
<tr>
<td>for(i=0;i&lt;size;i++)</td>
<td>#pragma omp simd</td>
<td>for(j=0;j&lt;size;j++)</td>
</tr>
<tr>
<td>ar1[i][j][k]+=ar1[i][j][k]<em>eps</em>(ar2[i][j][k]/eps);</td>
<td>#pragma omp parallel for collapse(2)</td>
<td>for(i=0;i&lt;size;i++)</td>
</tr>
<tr>
<td></td>
<td>#pragma omp simd</td>
<td>for(j=0;j&lt;size;j++)</td>
</tr>
<tr>
<td></td>
<td>ar1[i][j][k]+=ar1[i][j][k]<em>eps</em>(ar2[i][j][k]/eps);</td>
<td>#pragma omp simd</td>
</tr>
<tr>
<td></td>
<td></td>
<td>for(k=0;k&lt;size;k++)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ar1[i][j][k]+=ar1[i][j][k]<em>eps</em>(ar2[i][j][k]/eps);</td>
</tr>
</tbody>
</table>
### Collapse

<table>
<thead>
<tr>
<th>loop simd</th>
<th>loop ... simd</th>
<th>loop</th>
</tr>
</thead>
<tbody>
<tr>
<td>collapse(3)</td>
<td>278</td>
<td>-</td>
</tr>
<tr>
<td>collapse(2)</td>
<td>52</td>
<td>55</td>
</tr>
<tr>
<td>collapse(1)</td>
<td>63</td>
<td>66</td>
</tr>
</tbody>
</table>
Comparing Various Fruit – Time to Solution

SPEC Accel OpenACC/OpenMP base run

Run time in seconds

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>AMD Radeon R9 290X</th>
<th>NVIDIA Tesla K40</th>
<th>Intel Xeon Phi</th>
</tr>
</thead>
<tbody>
<tr>
<td>303.ostencil</td>
<td>1125</td>
<td>2682</td>
<td>1125</td>
</tr>
<tr>
<td>304.olbm</td>
<td>303.ostencil</td>
<td>2682</td>
<td>1125</td>
</tr>
<tr>
<td>314.omriq</td>
<td>354.sp</td>
<td>356.sp</td>
<td>357.sp</td>
</tr>
<tr>
<td>350.md</td>
<td>352.ep</td>
<td>353.clrleaf</td>
<td>354.cg</td>
</tr>
<tr>
<td>352.ep</td>
<td>355.seismic</td>
<td>356.sp</td>
<td>357.sp</td>
</tr>
<tr>
<td>353.clrleaf</td>
<td>354.cg</td>
<td>356.sp</td>
<td>357.sp</td>
</tr>
<tr>
<td>354.cg</td>
<td>356.sp</td>
<td>357.sp</td>
<td>357.sp</td>
</tr>
<tr>
<td>355.seismic</td>
<td>356.sp</td>
<td>357.sp</td>
<td>357.sp</td>
</tr>
<tr>
<td>356.sp</td>
<td>357.sp</td>
<td>357.sp</td>
<td>357.sp</td>
</tr>
<tr>
<td>357.sp</td>
<td>357.sp</td>
<td>357.sp</td>
<td>357.sp</td>
</tr>
<tr>
<td>359.miniGhost</td>
<td>360.ibdc</td>
<td>360.ibdc</td>
<td>360.ibdc</td>
</tr>
<tr>
<td>360.ibdc</td>
<td>360.ibdc</td>
<td>360.ibdc</td>
<td>360.ibdc</td>
</tr>
<tr>
<td>363.swim</td>
<td>363.swim</td>
<td>363.swim</td>
<td>363.swim</td>
</tr>
<tr>
<td>370.bt</td>
<td>370.bt</td>
<td>370.bt</td>
<td>370.bt</td>
</tr>
</tbody>
</table>

Guido Juckeland
Comparing Various Fruit – Power Consumption

SPEC Accel OpenACC/OpenMP base run

Average Power Consumption

AMD Radeon R9 290X  NVIDIA Tesla K40  Intel Xeon Phi

Guido Juckeland
CONCLUSIONS

- OpenACC and OpenMP both provide features aimed at accelerators.
- The two are not equivalent and have their own strengths and weaknesses.
- Work parallelizing for one is transferable to the other.
- Soon compilers will exist to allow more apples/apples comparisons, but today the hardware may dictate the choice of directives.