GPU TECHNOLOGY CONFERENCE

COMPARING OPENACC AND OPENMP PERFORMANCE AND PROGRAMMABILITY

JEFF LARKIN, NVIDIA

GUIDO JUCKELAND, *TECHNISCHE UNIVERSITÄT DRESDEN*

GPU TECHNOLOGY CONFERENCE

AGENDA

- OpenACC & OpenMP Overview
- Case Studies
- SPECaccel Lessons Learned
- Final Thoughts

IMPORTANT

- 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

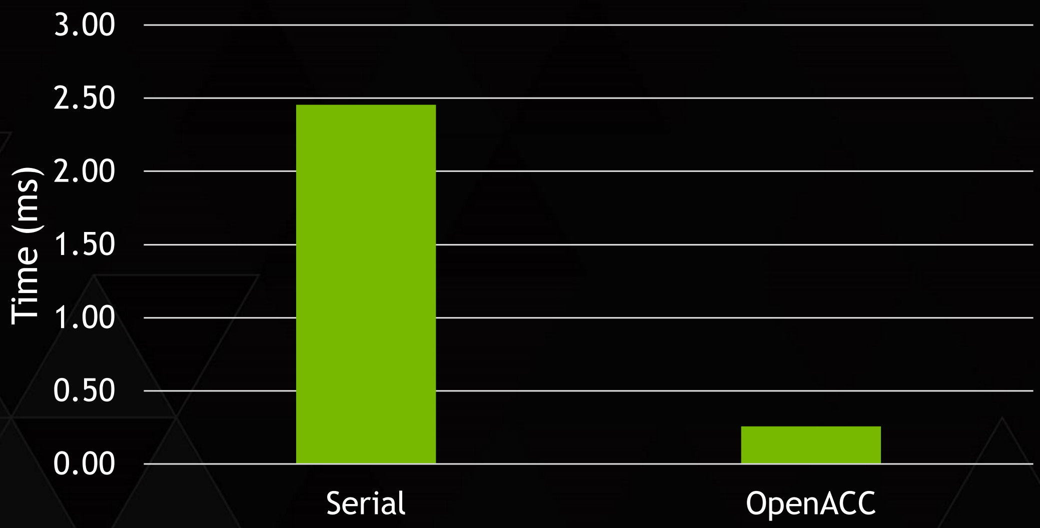
- OpenMP formed in 1997, focus on vendor-neutral Shared Memory Parallelism
- OpenMP 4.0: 2013
  - Expanded focus beyond shared memory parallel computers, including accelerators.
- The OpenMP 4.0 **`target`** construct provides the means to offload data and computation to accelerators.

CASE STUDY: DAXPY

GPU TECHNOLOGY CONFERENCE

OPENACC DAXPY

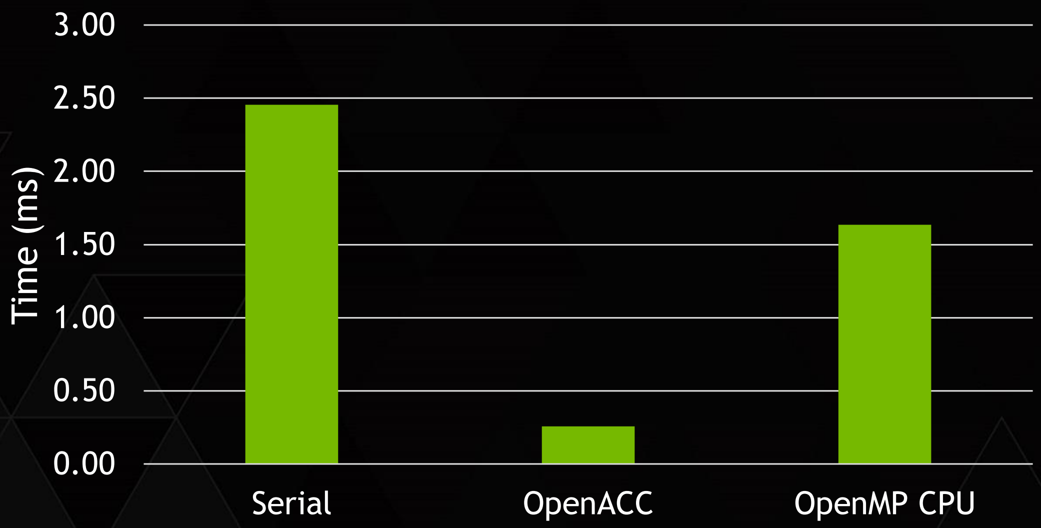
```
!$acc parallel loop present(x,y)
do i=1,n
  y(i) = a*x(i) + y(i)
enddo
```



- 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

OPENMP DAXPY: PARALLEL DO

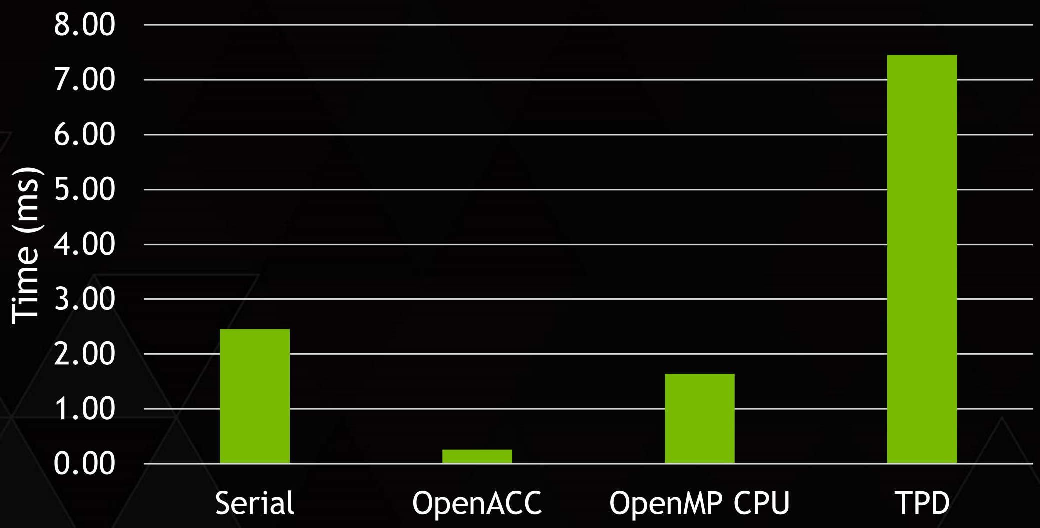
```
!$omp parallel do
do i=1,n
  y(i) = a*x(i) + y(i)
enddo
```



- 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 N/NumThreads contiguous iterations

OPENMP DAXPY: TARGET PARALLEL DO

```
length = n / blockDim%x
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
```

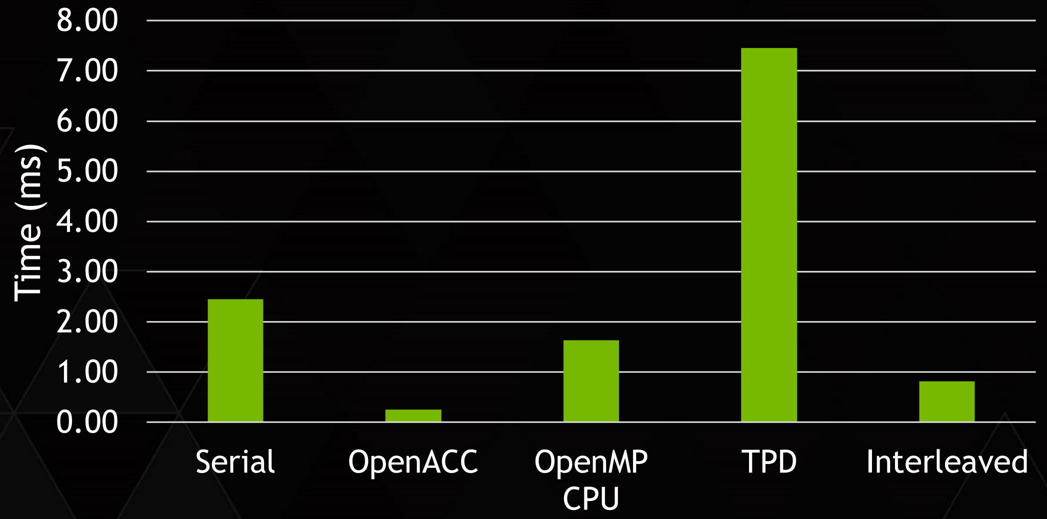


- TARGET PARALLEL DO dictates the following:
  - Offload data and execution to the target device
  - Use standard PARALLEL DO semantics once on the device
- Because threads can synchronize, the team must live within a thread block.
- Assumption: Static schedule with standard N/NTHREADS chunking

GPU TECHNOLOGY CONFERENCE

OPENMP: TARGET PARALLEL DO INTERLEAVED

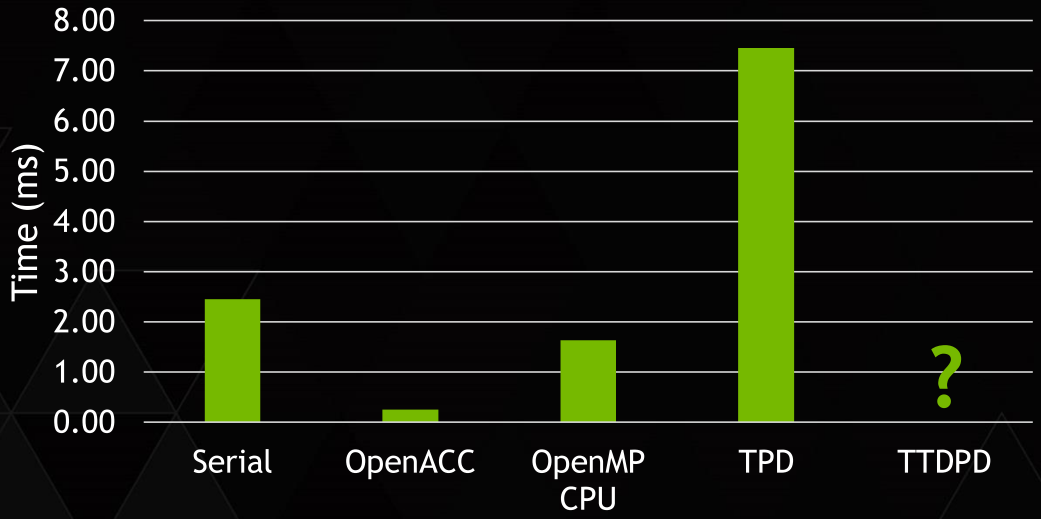
```
length = n / blockDim%x

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



- 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

```
!$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.

```
#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 4.0 TASKING

- 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.

```
#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)
}
```

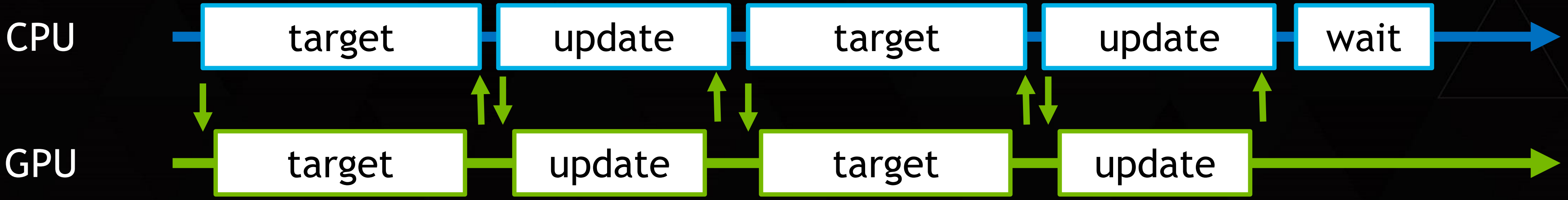
- As much as possible, back-to-back target directives should be fused into the same task to avoid involving the CPU in resolving dependencies.

OPENMP 4.1 TARGET DEPENDENCIES

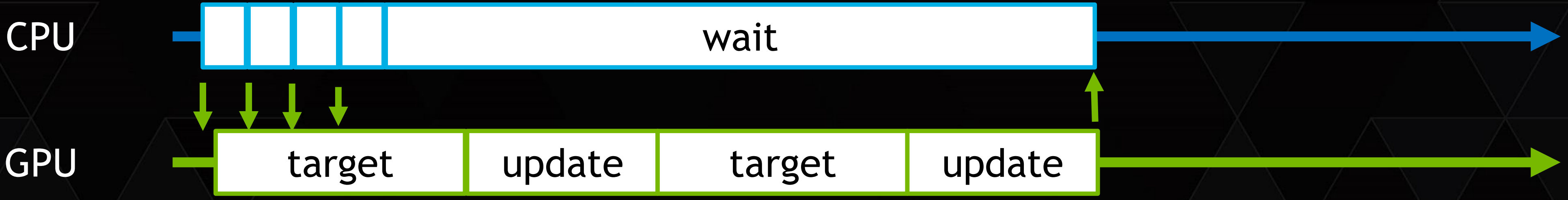
- Because resolving OpenMP 4.0 tasks requires the CPU, which could introduce unnecessary delays issuing work to the GPU, OpenMP4.1 simplifies asynchronous target operations.
- TARGET constructs are now implicitly TASKS and accept DEPEND clauses.
- TARGET constructs are made asynchronous with a NOWAIT clause

```
#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

TECHNISCHE
UNIVERSITÄT
DRESDEN

Center for Information Services and High Performance Computing (ZIH)

Comparing Apples and Oranges

Using SPECaccel as a Yardstick

Guido Juckeland (guido.juckeland@tu-dresden.de)

CUDA™
CENTER OF
EXCELLENCE

ZIH
Center for Information Services &
High Performance Computing

SPEC ACCEL

- 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

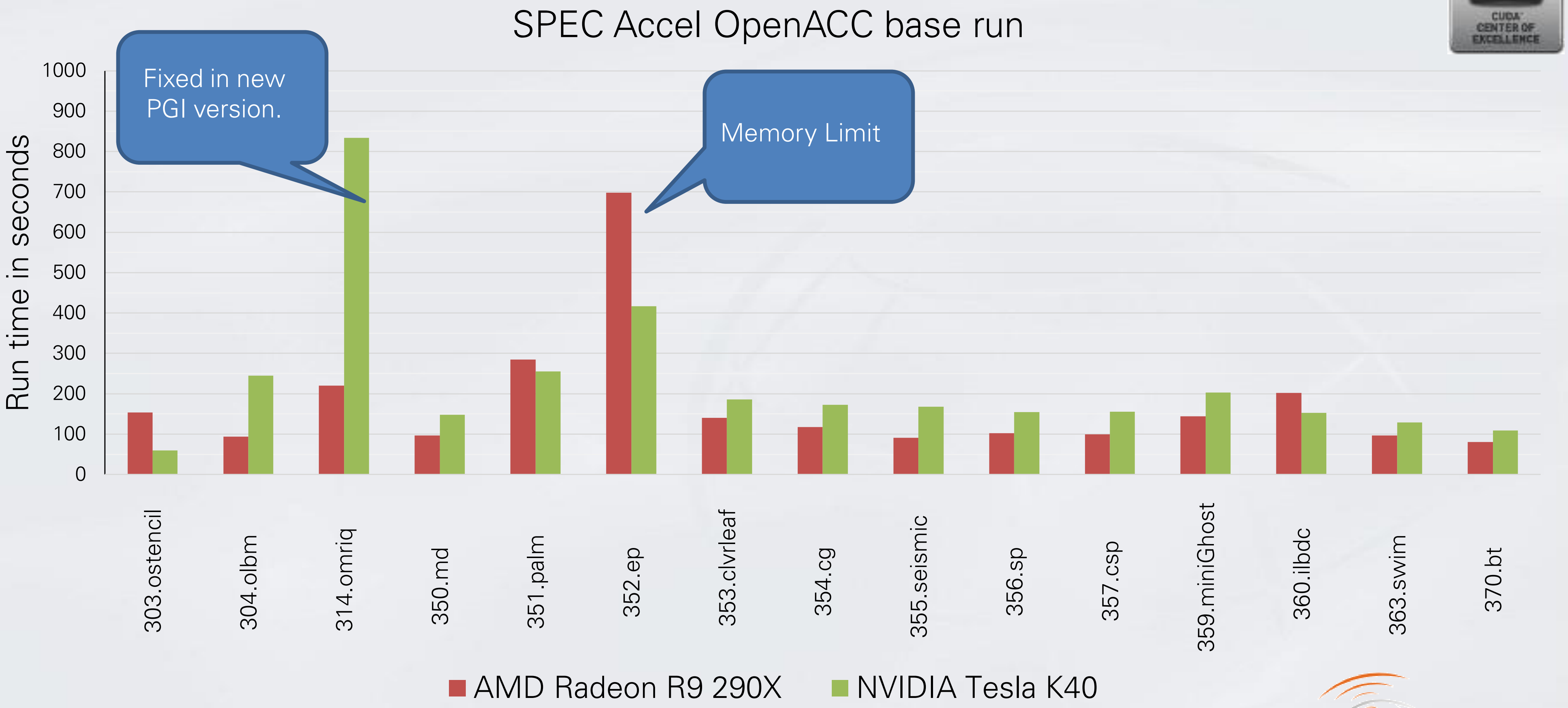
OpenACC Suite

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

Used Hardware

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

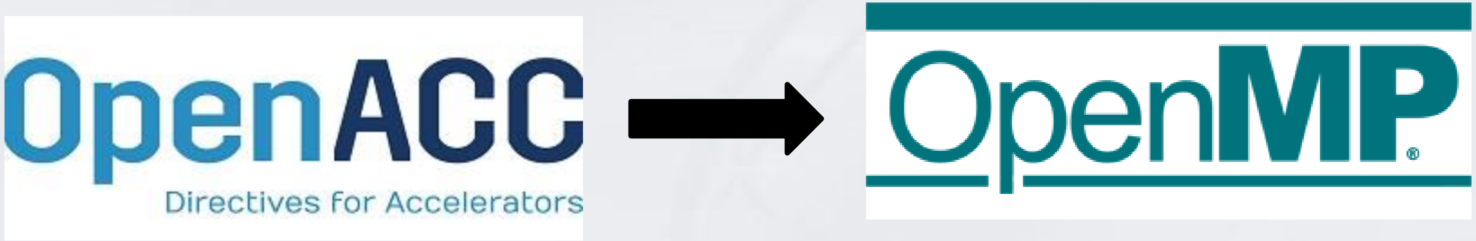
OpenACC runtimes



Fixed in new PGI version.

Memory Limit

Converting the 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(tofrom/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

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

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

ACC parallel

```
#pragma acc parallel
{
 #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] = …;
}
```

ACC Kernels

```
#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

```
int x[10],y[10];
#pragma acc data copy(x) pcopy(y)
{
 ...
 #pragma acc kernels copy(x) pcopy(y)

 {
  // Accelerator Code
  ...
 }
 ...
}
```

```
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...

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

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

}
```

```
int foo, bar;


int main(...)

 {

 // ...

 #pragma omp target

 {

  ...

 }

}
```

Loop Ordering

```
#pragma acc parallel
 #pragma acc loop collapse(3)
 for(k=0;k<size;k++)
 for(j=0;j<size;j++)
   for(i=0;i<size;i++)
    ar1[i][j][k]+=ar1[i][j][k]*eps*(ar2[i][j][k]/eps);
```

```
    #pragma omp target
     #pragma omp simd
     for(k=0;k<size;k++)
  #pragma omp parallel for collapse(2)
      for(j=0;j<size;j++)
   for(i=0;i<size;i++)
    ar1[i][j][k]+=ar1[i][j][k]*eps*(ar2[i][j][k]/eps);
```

Differences in OMP: kji:

6s

Ijk: 0,2s

```
#pragma omp target
 #pragma omp parallel for collapse(2)
 for(i=0;i<size;i++)
  for(j=0;j<size;j++)
   #pragma omp simd
   for(k=0;k<size;k++)
    ar1[i][j][k]+=ar1[i][j][k]*eps*(ar2[i][j][k]/eps);
```

Collapse

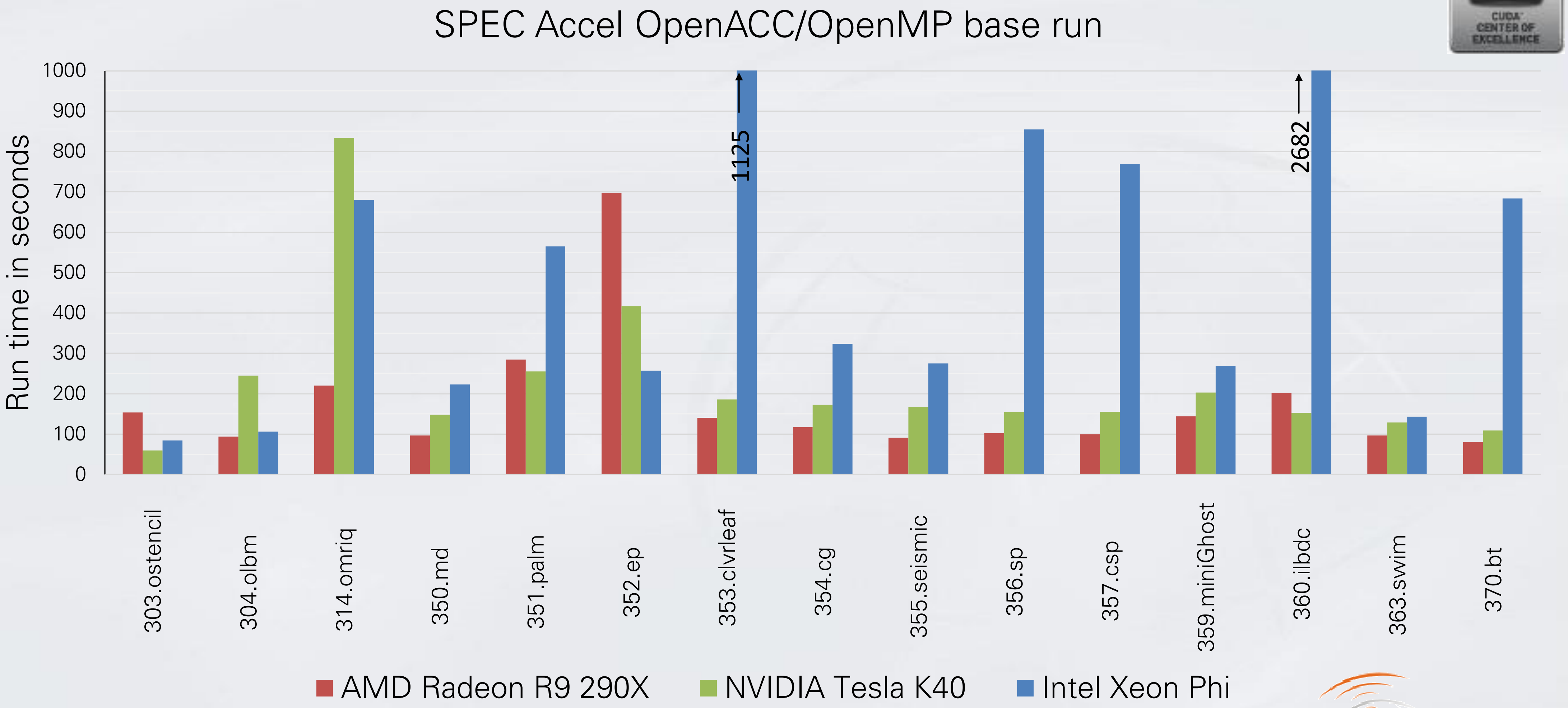
```
#pragma omp target
 #pragma omp parallel for simd collapse(n)
 for(i=0;i<size;i++)
  for(j=0;j<size;j++)
  for(k=0;k<size;k++)
    ar1[i][j][k]+=ar1[i][j][k]*eps*(ar2[i][j][k]/eps);
```

```
#pragma omp target
 #pragma omp parallel for collapse(n)
 for(i=0;i<size;i++)
  for(j=0;j<size;j++)
   #pragma omp simd
   for(k=0;k<size;k++)
    ar1[i][j][k]+=ar1[i][j][k]*eps*(ar2[i][j][k]/eps);
```

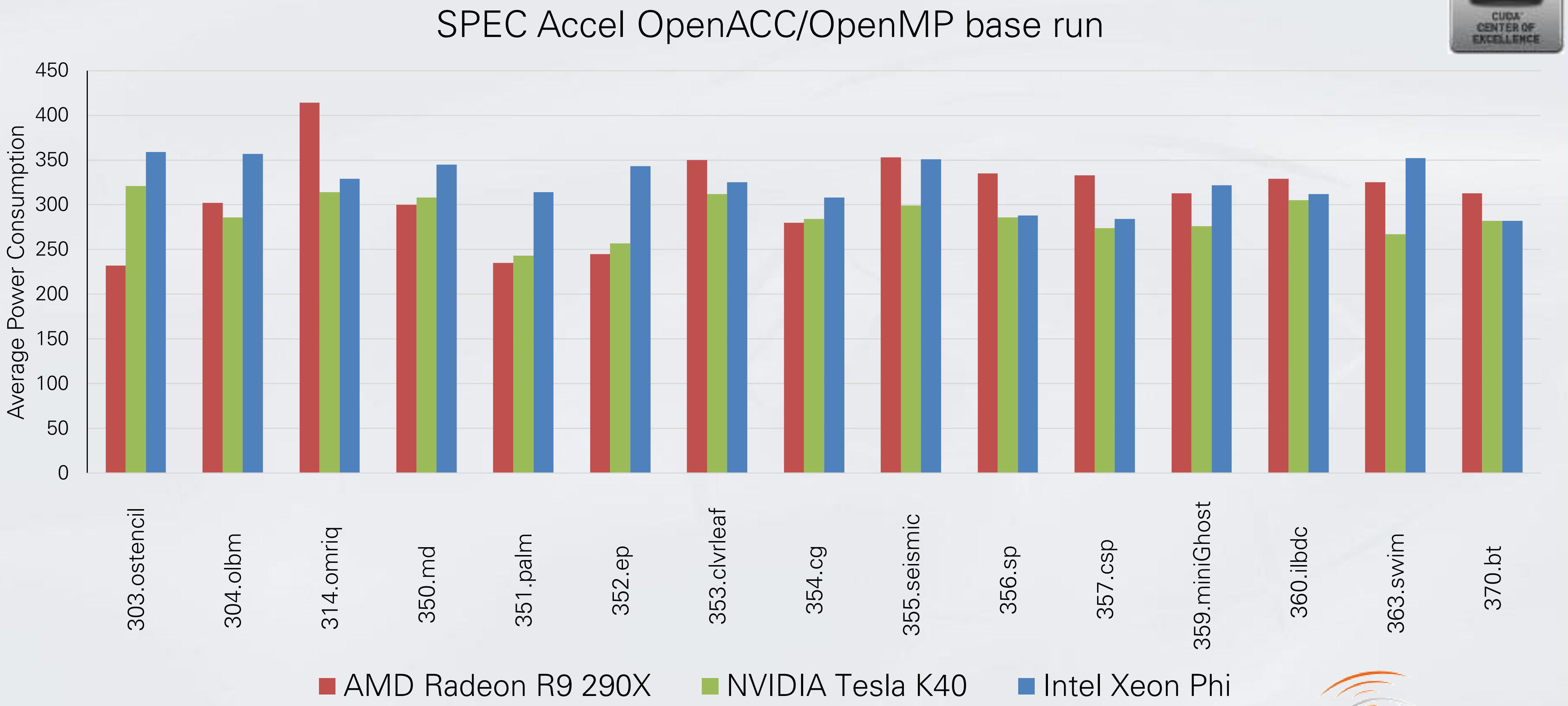
	loop simd	loop … simd	loop
collapse(3)	278	-	279
collapse(2)	52	55	65
collapse(1)	63	66	63

```
#pragma omp target
 #pragma omp parallel for collapse(n)
 for(i=0;i<size;i++)
  for(j=0;j<size;j++)
  for(k=0;k<size;k++)
    ar1[i][j][k]+=ar1[i][j][k]*eps*(ar2[i][j][k]/eps);
```

Comparing Various Fruit – Time to Solution



Comparing Various Fruit – Power Consumption



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.