Introduction to OpenACC

Paul Springer

Aachen Institute for Advanced Study in
Computational Engineering Science

Aachen, 06.06.13

Outline

Outline

Why Coprocessors?

- Power efficiency
- Massive compute power
  - More than 1 TFLOPS/s DP per coprocessor

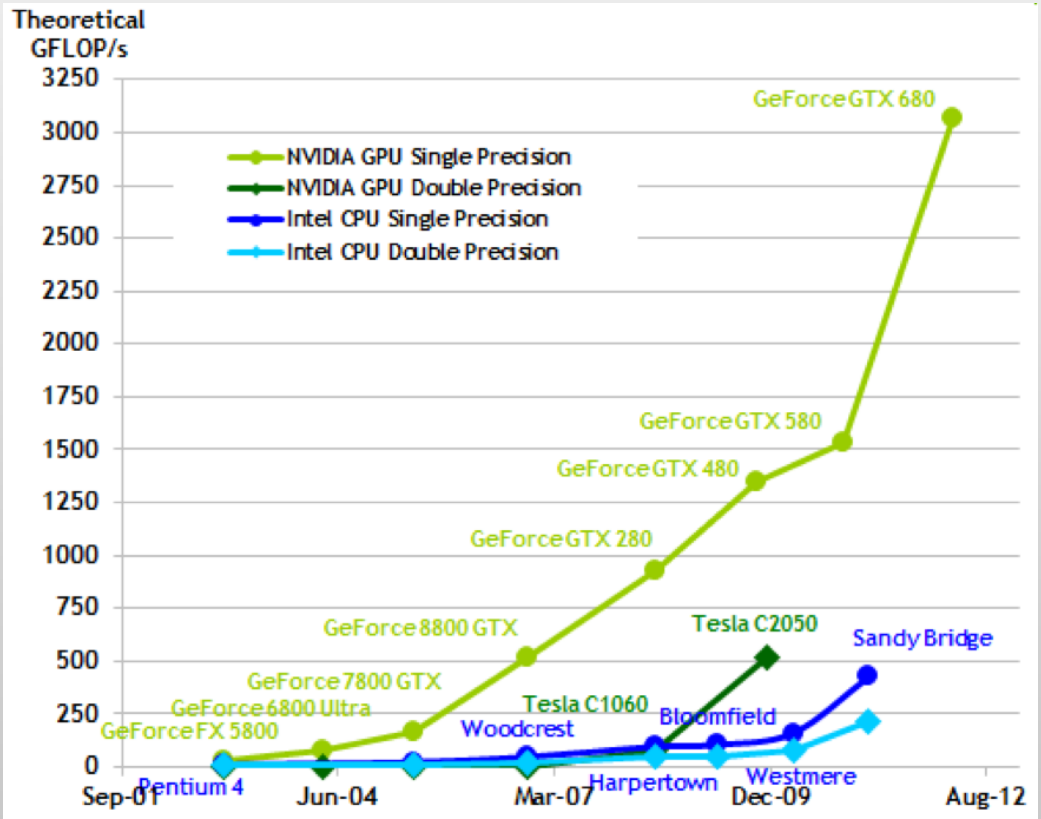


Figure: Theoretical peak performance. Taken from [1].

Why OpenACC?

- Directive-based parallel programming
  - Incremental parallelization
  - High productivity
  - Increased portability
- Growing diversity of coprocessors
  - NVIDIA GPUs
  - AMD GPUs
  - Intel's Xeon Phi
  - Digital Signal Processors (DSPs)
  - Field Programmable Gate Array (FPGAs)

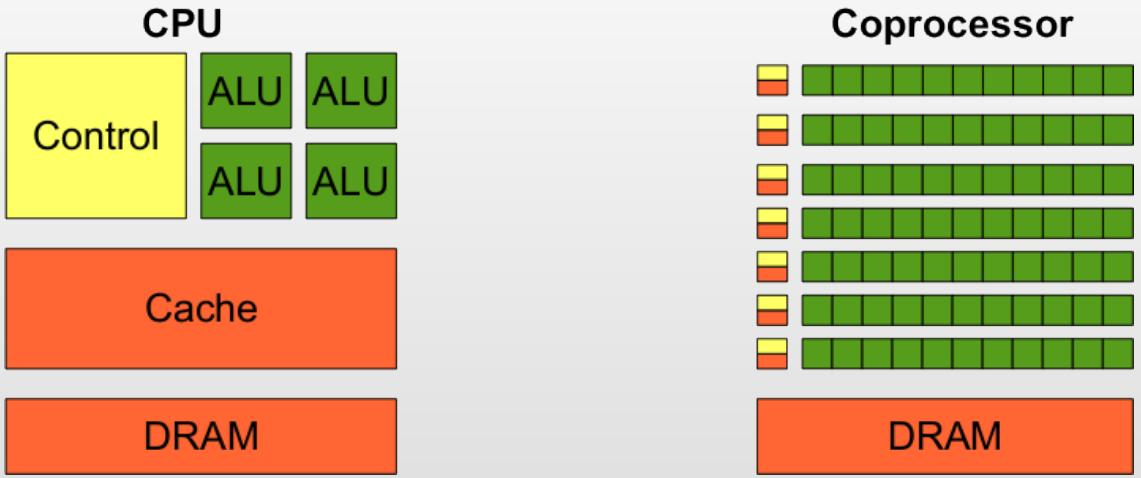
Why OpenACC?

- Top500 List Nov 2012
  - Contains 62 heterogeneous systems
  - Titan @ Oak Ridge National Lab (17.6 PFLOPS/s)
    - NVIDIA K20X GPU accelerator
  - Stampede @ Texas Advanced Computing Center (2.7 PFLOPS/s)
    - Intel Xeon Phi

Why OpenACC?

- Top500 List Nov 2012
  - Contains 62 heterogeneous systems
  - Titan @ Oak Ridge National Lab (17.6 PFLOPS/s)
    - NVIDIA K20X GPU accelerator
  - Stampede @ Texas Advanced Computing Center (2.7 PFLOPS/s)
    - Intel Xeon Phi
- Top500 List Jun 2013 (Leaked information)
  - Tianhe-2 ($\approx$ 30+ PFLOPS/s)
    - Intel Xeon Phi

Common features of coprocessors



- Massively parallel
- Asynchronous execution to the host (i.e. CPU)
- Separate memory space from the host

Outline

- Introduced in November 2011
  - PGI, CAPS, NVIDIA and CRAY
- Directive-based approach
- Offloads work to a coprocessor
- More comprehensible way to program the GPU
- Available for
  - C/C++ and Fortran
  - NVIDIA and AMD GPUs

Syntax

```
#pragma acc directive-name [clause, ...]
{  structured block  }
```

ACC Kernels

```
#pragma acc kernels [clause, ...]
{  structured block  }
```

- Compiler responsible for finding parallelism
- Can generate multiple kernels
- Synchronization between kernels

ACC Parallel

```
#pragma acc parallel [clause, ...]
{  structured block  }
```

- User responsible for finding parallelism
- Generates a single kernel
- No synchronization between loops within kernel

ACC Kernels

```
#pragma acc kernels [clause, ...]
{  structured block  }
```

- Compiler responsible for finding parallelism
- Can generate multiple kernels
- Synchronization between kernels

ACC Parallel

```
#pragma acc parallel [clause, ...]
{  structured block  }
```

- User responsible for finding parallelism
- Generates a single kernel
- No synchronization between loops within kernel

```
For i=1:N DO
    ...
END DO

For i=1:N DO
    ...
END DO
```

ACC Kernels

```
#pragma acc kernels [clause, ...]
{  structured block  }
```

- Compiler responsible for finding parallelism
- Can generate multiple kernels
- Synchronization between kernels

```
For i=1:N DO
    ...
END DO
```

```
For i=1:N DO
    ...
END DO
```

ACC Parallel

```
#pragma acc parallel [clause, ...]
{  structured block  }
```

- User responsible for finding parallelism
- Generates a single kernel
- No synchronization between loops within kernel

ACC Kernels

```
#pragma acc kernels [clause, ...]
{  structured block  }
```

- Compiler responsible for finding parallelism
- Can generate multiple kernels
- Synchronization between kernels

ACC Parallel

```
#pragma acc parallel [clause, ...]
{  structured block  }
```

- User responsible for finding parallelism
- Generates a single kernel
- No synchronization between loops within kernel

```
For i=1:N DO
    ...
END DO

For i=1:N DO
    ...
END DO
```

Live Demo

Vector-Vector Multiplication

Live Demo

- PGI Compiler Flags:
  - Minfo=accel
  - -ta=nvidia,cc20
- Restict keyword
  - double * **restrict** x;
- Compiler **can** automatically detect
  - Parallelism
  - Data transfers
  - Reductions
- PGI_ACC_TIME=1
  - Timings

- Execution Units
  - **Gang** similar to CUDA threadblock
  - **Worker** similar to CUDA warp
  - **Vector** similar to CUDA threads
- No synchronization between gangs (same as CUDA)

ACC Loop

```
#pragma acc loop [clause, ...]
{  structured block  }
```

- Work-sharing directive
- Specify scheduling policy
  - gang([integer])
  - worker([integer])
  - vector([integer])
  - seq

ACC Data

```
#pragma acc data [clause, ...]
{  structured block  }
```

- Data region
- Can save redundant data transfers
- Explicitly state data transfers
  - Allocate: create(list)
  - Copy HtoD: copyin(list)
  - Copy DtoH: copyout(list)
  - Copy HtoD and DtoH: copy(list)
  - Data is present on device: present(list)

Live Demo

Matrix-Vector Multiplication

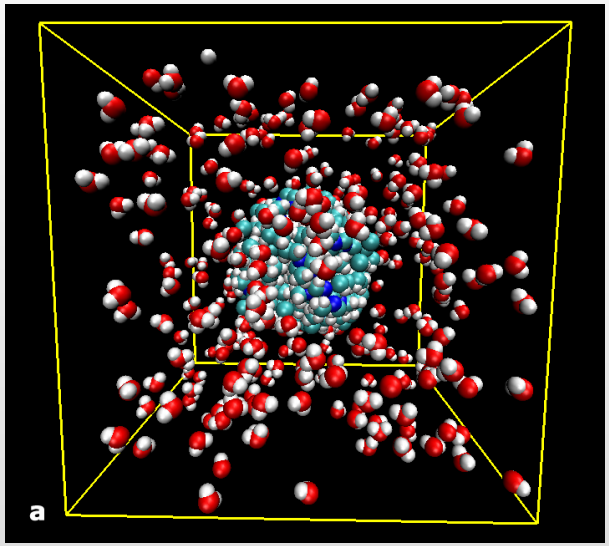
Advanced Features

- Multi GPU support
- Asynchronous execution and data transfers
- CUDA interoperability

Outline

Molecular Dynamics Simulation

- System of N interacting particles
  - E.g.: Atoms, molecules, planets
- Simulate their motion
- Detect chemical reactions
- Forces of particle i



© Amir Niazi, AICES

$$\vec{f_i} = m_i \vec{a_i} = -\nabla_i U(t) \tag{1}$$

- Potential

$$U(t) = \frac{1}{2} \sum_{i=1}^{N} \sum_{\substack{j=1 \\ j \neq i}}^{N} U_{i,j}(\|\vec{r}_{i,j}\|) \tag{2}$$

Algorithm 1 Overview of the main Molecular Dynamics routine.

1: **for** $i = 1$ to M **do**
2: $t \leftarrow t + dt$
3: compute_forces($\vec{r}, \vec{f}$)
4: integrate($\vec{r}, \vec{f}, \vec{v}, dt$)
5: //Do something with the data
6: **end for**

- *compute_forces* has a complexity of $\mathcal{O}(N^2)$

Algorithm 2 Compute_forces routine.

1: **for** $i = 1$ to N **do**
2: $\quad \vec{f_i} \leftarrow 0$
3: $\quad$ **for** $j = 1$ to N **do**
4: $\quad\quad \vec{r}_{i,j} \leftarrow \vec{r_j} - \vec{r_i}$
5: $\quad\quad f_{i,j} \leftarrow \text{compute\_force}(\|\vec{r}_{i,j}\|)$
6: $\quad\quad \vec{f_i} \leftarrow \vec{f_i} + f_{i,j}\ \vec{r}_{i,j}$
7: $\quad$ **end for**
8: **end for**

Algorithm 3 Naive OpenACC compute_forces routine.

```
1: #pragma acc kernels
2: for i = 1 to N do
3:     f_i ← 0
4:     for j = 1 to N do
5:         r_{i,j} ← r_j − r_i
6:         f_{i,j} ← compute_force(‖r_{i,j}‖)
7:         f_i ← f_i + f_{i,j} r_{i,j}
8:     end for
9: end for
```

- Inner loop can not be parallelized
  - Loop-carried dependencies

Algorithm 4 Improved compute_forces routine.

```
1:  #pragma acc kernels
2:  for i = 1 to N do
3:      f_i ← 0
4:      #pragma acc loop reduction(+:f_i)
5:      for j = 1 to N do
6:          r_{i,j} ← r_j − r_i
7:          f_{i,j} ← compute_force(‖r_{i,j}‖)
8:          f_i ← f_i + f_{i,j} r_{i,j}
9:      end for
10: end for
```

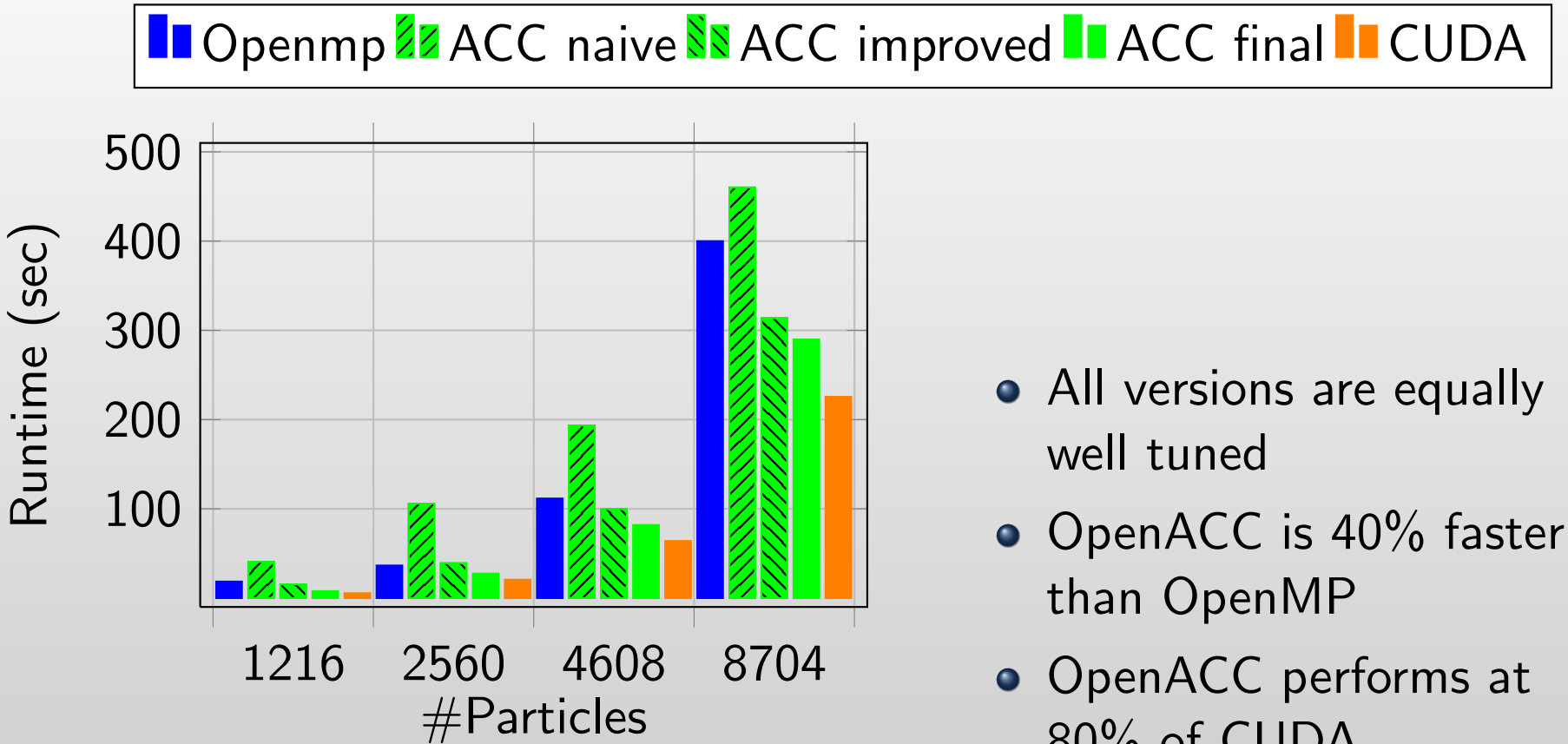
- Good: Inner loop can be parallelized
- Bad: Arrays are reallocated in every iteration

Algorithm 5 Overview of the main Molecular Dynamics routine with an OpenACC data region.

1: #pragma acc data create ($\vec{r}$[0:N],$\vec{f}$[0:N])
2: **for** $i = 1$ to M **do**
3: $t \leftarrow t + dt$
4: compute_forces($\vec{r}, \vec{f}$)
5: integrate($\vec{r}, \vec{f}, \vec{v}, dt$)
6: //Do something with the data
7: **end for**

Algorithm 6 Final compute_forces routine.

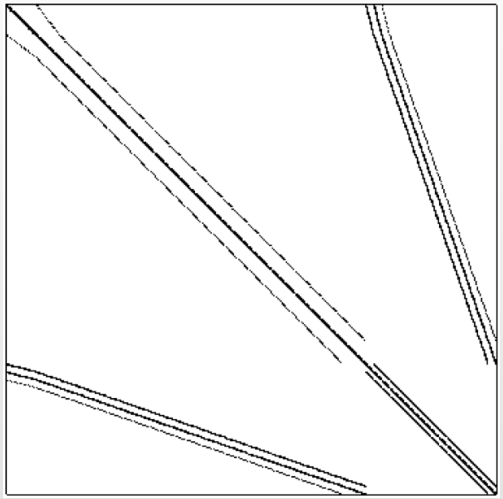
```
1:  #pragma acc update device(r[0:N])
2:  #pragma acc kernels present(r[0:N], f[0:N])
3:  for i = 1 to N do
4:      f_i ← 0
5:      #pragma acc loop reduction(+:f_i)
6:      for j = 1 to N do
7:          r_{i,j} ← r_j − r_i
8:          f_{i,j} ← compute_force(‖r_{i,j}‖)
9:          f_i ← f_i + f_{i,j} r_{i,j}
10:     end for
11: end for
12: #pragma acc update host(f[0:N])
```



- All versions are equally well tuned
- OpenACC is 40% faster than OpenMP
- OpenACC performs at 80% of CUDA

Figure: Runtime of a Molecular Dynamics (MD) Simulation for different problem sizes over 10.000 iterations. All calculations are run in double precision. OpenMP: 16 core SMP node. OpenACC/Cuda: Nvidia Quadro 6000 GPU.

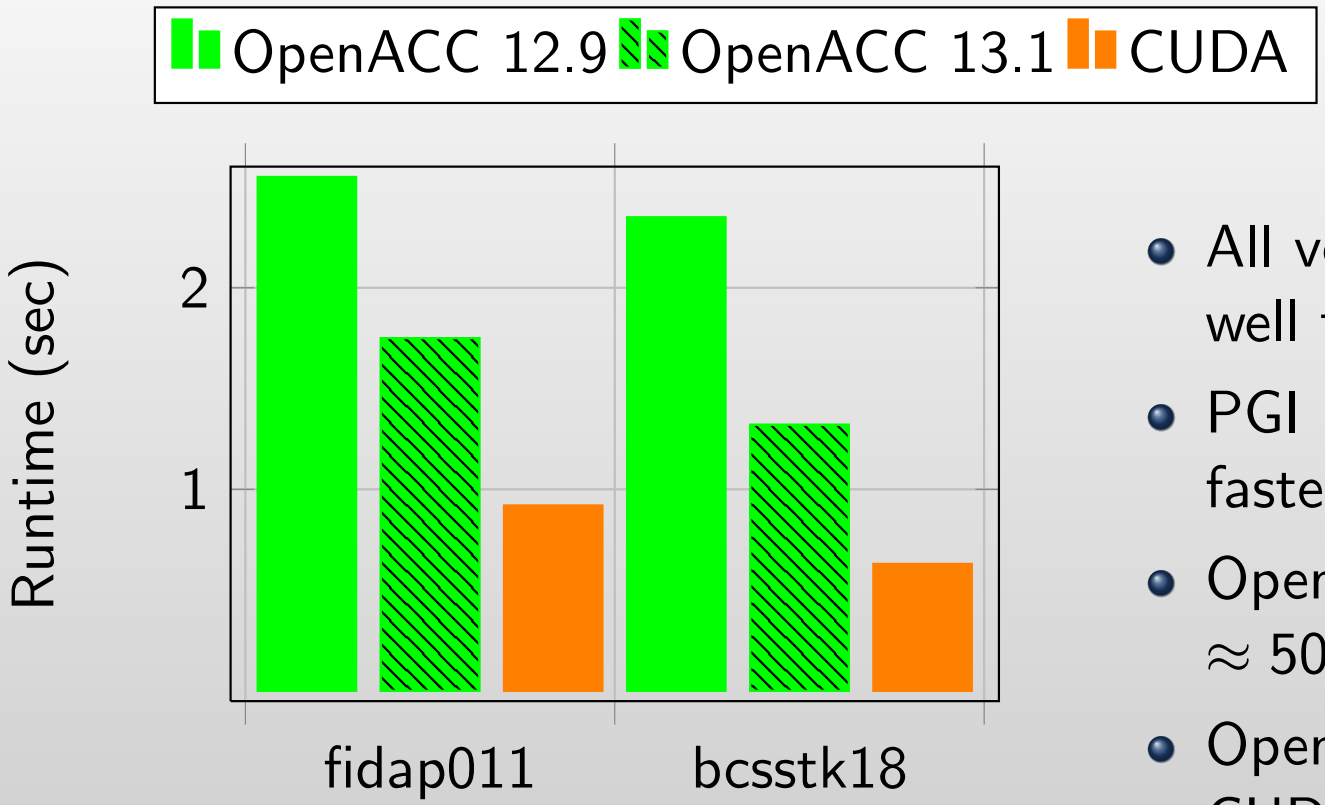
Conjugate Gradient Method



- Iterative solver
- Solve a large sparse linear system

$$Ax = b \tag{3}$$

- Frequently arise from *partial differential equations* in physics
- Runtime dominated by *Sparse Matrix-Vector Multiplication* SPMV



- All versions are equally well tuned
- PGI 13.1 50%/80% faster than PGI 12.9
- OpenACC performs at $\approx$ 50% of CUDA
- OpenMP outperforms CUDA

Figure: Runtime of a Conjugate Gradient (CG) Method for two sparse matrices. All calculations are run in double precision. OpenMP: 16 core SMP node. OpenACC/Cuda: Nvidia Quadro 6000 GPU.

Productivity

Productivity - Contra

- Function calls require inlining
- PGI compiler does not support C++
- Limited debugging support for PGI compiler
  - Revert to debugging the logic of your application

	OpenMP	OpenACC	CUDA
MD	23	16	92
CG	8	16	156

Table: Number of added and modified lines of source code for each case study and paradigm with respect to the serial version.

- Few added/modified lines of source code
- Data transfers are straight forward
- Reductions require almost no additional effort
- No need to worry about “boundary conditions”
- Compiler is able to tune for a specific coprocessor

Outline

Future of OpenACC

- OpenACC 2.0
  - Announced at SC 12 (November)
  - Additional features
    - Multi dimensional tiling
    - Nested parallelism
    - Function calls within accelerator regions
  - PGI support expected for mid 2013
- PGI announced AMD and Xeon PHI support by mid 2013
- OpenMP 4.0
  - OpenACC and OpenMP is likely to merge in the future

Outline

Conclusion

- High productivity (if you don't run into compiler bugs)
- Decent performance
- Limited debugging support for PGI compiler
- Makes coprocessor programming more straightforward
  - C code $\rightarrow$ OpenACC code $\rightarrow$ CUDA code

- OpenACC
  - webinars
    - developer.nvidia.com/cuda/gpu-computing-webinars
    - www.pgroup.com/resources/articles.htm#webinars
  - www.openacc-standard.org

Thank you for your attention.

References I

NVIDIA.
CUDA C Best Practices Guide, August 2012.
Version 4.2.

S. Wienke, P. Springer, C. Terboven, and D. an Mey.
OpenACC-First Experiences with Real-World Applications.
Euro-Par 2012 Parallel Processing, pages 859–870, 2012.