OpenACC and PGI Compiler

Seminar
„Compiler Techniques for Parallel Hardware Architectures“
Dimitri Blatner
Hybrid Architectures
Hybrid Architectures
Hybrid Architectures

Diagram showing the components of a hybrid architecture:
- CPU
- Main Memory
- Accelerator
- Acc. Memory
Hybrid Architectures

CPU
~ 150 GFLOPS

32 GB ECC
DDR3

~ 42 GB/s

GPU
~ 1.17 TFLOPS

5GB ECC
GDDR5

~ 200 GB/s

PCIe-2
8 GB/s
Hybrid Architectures

- **CPU**
  - 32 GB ECC DDR3
  - ~150 GFLOPS
  - ~42 GB/s

- **GPU**
  - 5 GB ECC GDDR5
  - ~1.17 TFLOPS
  - ~200 GB/s

- PCIe-2
  - 8 GB/s

- **less data transfer**
Hybrid Architectures

- CPU: 32 GB ECC DDR3, ~150 GFLOPS
- GPU: 5GB ECC GDDR5, ~1.17 TFLOPS

- Memory bandwidth:
  - ~42 GB/s
  - ~200 GB/s

- PCIe-2: 8 GB/s

- Features:
  - Less data transfer
  - Async. computing
Hybrid Architectures

- 32 GB ECC DDR3
- 5 GB ECC GDDR5
- CPU ~ 150 GFLOPS
- GPU ~ 1.17 TFLOPS
- PCIe-2 8 GB/s
- ~ 42 GB/s
- ~ 200 GB/s

- less data transfer
- async. computing
- keep data resident
Hybrid Architectures

- CPU
  - 32 GB ECC DDR3
  - ~ 150 GFLOPS
  - ~ 42 GB/s

- GPU
  - 5GB ECC GDDR5
  - ~ 1.17 TFLOPS
  - ~ 200 GB/s
  - PCIe-2
  - 8 GB/s

- less data transfer
- async. computing
- keep data resident
Programming
Programming

- CUDA
Programming

- CUDA
- OpenCL
Programming

• CUDA
• OpenCL

• Programming at a low-level
Programming

- CUDA
- OpenCL

- Programming at a low-level
- CUDA technology dependent
Programming

- CUDA
- OpenCL

- Programming at a low-level
- CUDA technology dependent
- Write kernels with specific language constructs
Programming

• CUDA
• OpenCL

• Programming at a low-level
• CUDA technology dependend
• Write kernels with specific language constructs
• Optimize for specific GPU
Programming

• CUDA
• OpenCL

• Programming at a low-level
• CUDA technology dependend
• Write kernels with specific language constructs
• Optimize for specific GPU
• Not easy to debug
Programming

- CUDA
- OpenCL

- Programming at a low-level
- CUDA technology dependend
- Write kernels with specific language constructs
- Optimize for specific GPU
- Not easy to debug
- Not easy to port to new accelerator
Programming

- CUDA
- OpenCL

Application

Programming Libraries
Compiler Directives
Language Constructs
Programming

- CUDA
- OpenCL
- Alternative approaches: hiCUDA, OpenMPC

Application

Programming Libraries

Compiler Directives

Language Constructs
Programming

• CUDA
• OpenCL

• Alternative approaches: hiCUDA, OpenMPC

Approach taken from OpenMP:
Compiler Directives for Accelerators, Libraries
OpenACC
OpenACC

- PGI accelerator directives, OpenHMPP, Cray CCE
OpenACC

- PGI accelerator directives, OpenHMPP, Cray CCE
- more and more merged into OpenACC
OpenACC

- PGI accelerator directives, OpenHMPP, Cray CCE
- more and more merged into OpenACC
  - Implicit data transfers
  - High-Level directives, but compiler must handle problems
- Loss in performance ~ 10-15% Ø
- C/C++ and Fortran supported
- Define regions & offload them to the accelerator device
OpenACC: Directives

```c
#pragma acc parallel [clause [,] clause] ...] new-line
C/C++ structure block
```

```fortran
!$acc parallel [clause [,] clause] ...] new-line
Fortran structure block
!$acc end parallel
```
OpenACC: Directives
OpenACC: Directives

The programmer also uses OpenACC directives to mark compute intensive code regions, which are then "offloaded" to the accelerator, i.e., the enclosed code is only run by one or more available accelerators. So just these regions will be accelerated.

OpenACC distinguishes between several different "regions," the most important are the parallel regions and kernels regions.

Typically, parallel regions contain work-sharing loops, where each iteration computes a discrete and mostly independent part of the whole loop. Kernels regions will execute the code region as a kernel, i.e., typically one or more nested loops that are divided into domains and can be executed by $N$ threads in any order in parallel. In CUDA or OpenCL a single but specific function is called a kernel. OpenACC does not specify how the compiler has to partition the loops, so at the end the compiler is the critical factor for the performance.

In fact, it is still important that the programmer knows the capabilities of the accelerator. If the host and the accelerator do not share the same memory address space, the movement of too much data compared to the computational effort leads to a loss in performance. Additionally, the limited accelerator memory can prevent the compiler from offloading regions to the accelerator.

5.1 Directives

The following definitions are taken from the OpenACC API v7 specification: Compiler directives in C and C11 are specified using the preprocessor keyword "#pragma," in Fortran the comment keyword "$!$" followed by "acc" and the directive name, so the compiler knows it is an OpenACC directive.

The syntax for C and C11 is defined as:

```
#pragma acc directive-name [clause [[, clause] ...] new-line
C/C++ structure block
```

The squared brackets `[]` intend that the argument is optional.

For Fortran the syntax is:

```
$r$
acc directive-name [clause [[, clause] ...] new-line
Fortran structure block
$r$
acc end parallel
```

There are directives sorted by their function, namely

- parallel
- kernels
- data
- host_data
- loop
- cache
- declare
- update
- wait

Every directive has a valid identity domain that is called a region and is usually a structured code block, e.g., a for loop. The most important construct is the "parallel" directive. The syntax therefore is:

```
#pragma acc parallel [clause [[, clause] ...] new-line
C/C++ structure block
```

Whenever a parallel region is reached, groups of workers (also called gangs) are created. Then, each worker (that is a thread or a group of threads) starts the execution of the structured block. During the execution of the parallel region, the amount of gangs and workers are fixed. No other parallel region or kernels region can be executed inside a parallel region. The optional clause can be one of:
OpenACC: Directives

#define acc parallel [clause [[,]] clause] ...] new-line
C/C++ structure block

#define acc kernels [clause [[,]] clause] ...] new-line structure block
OpenACC: Directives

#pragma acc parallel [clause [[,] clause] ...] new-line
C/C++ structure block

#pragma acc kernels [clause [[,] clause] ...] new-line
structure block

#pragma acc data/host_data [clause [[,] clause] ...] new-line
structure block
OpenACC: Directives

```
#pragma acc parallel [clause [[,] clause] ...] new-line
C/C++ structure block

#pragma acc kernels [clause [[,] clause] ...] new-line
structure block

#pragma acc data/host_data [clause [[,] clause] ...] new-line
structure block

#pragma acc loop [clause [[,] clause] ...] new-line
for loop
```
OpenACC: Directives

### OpenACC: Directives

- **#pragma acc parallel**
  
  ```
  pragma acc parallel [clause [[,] clause] ...] new-line
  ``

- **#pragma acc kernels**
  
  ```
  pragma acc kernels [clause [[,] clause] ...] new-line
  ``

- **#pragma acc data/host_data**
  
  ```
  pragma acc data/host_data [clause [[,] clause] ...] new-line
  ``

- **#pragma acc loop**
  
  ```
  pragma acc loop [clause [[,] clause] ...] new-line
  ``

- **#pragma acc cache**
  
  ```
  pragma acc cache( list ) new-line
  ```
OpenACC: Directives

```plaintext
#pragma acc parallel [clause [[,] clause] ...] new-line
C/C++ structure block

#pragma acc kernels [clause [[,] clause] ...] new-line
structure block

#pragma acc data/host_data [clause [[,] clause] ...] new-line
structure block

#pragma acc loop [clause [[,] clause] ...] new-line
for loop

#pragma acc cache( list ) new-line

#pragma acc declare declclause [[,] declclause]... new-line
```
OpenACC: Directives

```
#pragma acc parallel [clause [[,] clause] ...] new-line
C/C++ structure block

#pragma acc kernels [clause [[,] clause] ...] new-line
structure block

#pragma acc data/host_data [clause [[,] clause] ...] new-line
structure block

#pragma acc loop [clause [[,] clause] ...] new-line
for loop

#pragma acc cache( list ) new-line

#pragma acc declare declclause [[,] declclause]... new-line

#pragma acc update clause [[,] clause]... new-line
```
OpenACC: Directives

```c
#pragma acc parallel [clause [[,] clause] ...] new-line  
C/C++ structure block

#pragma acc kernels [clause [[,] clause] ...] new-line  
structure block

#pragma acc data/host_data [clause [[,] clause] ...] new-line  
structure block

#pragma acc loop [clause [[,] clause] ...] new-line  
for loop

#pragma acc cache( list ) new-line

#pragma acc declare declclause [[,] declclause]... new-line

#pragma acc update clause [[,] clause]... new-line

#pragma acc wait [(( scalar-integer-expression ))] new-line
```
OpenACC: Clauses
OpenACC: Clauses

if ( condition )
async [( scalar-integer-expression )]
num_gangs( scalar-integer-expression )
num_workers( scalar-integer-expression )
vector_length ( scalar-integer-expression )
reduction( operator:list )
copy( list )
copyin( list )
copyout( list )
create( list )
present( list )
present_or_copy( list )
present_or_copyin( list )
present_or_copyout( list )
present_or_create( list )
deviceptr( list )
private( list )
firstprivate ( list )
OpenACC: Clauses

if ( condition )
async [ ( scalar-integer-expression ) ]
num_gangs ( scalar-integer-expression )
num_workers ( scalar-integer-expression )
vector_length ( scalar-integer-expression )
reduction ( operator: list )
copy ( list )
copyin ( list )
copyout ( list )
create ( list )
present ( list )
present_or_copy ( list )
present_or_copyin ( list )
present_or_copyout ( list )
present_or_create ( list )
deviceptr ( list )
private ( list )
firstprivate ( list )
OpenACC: Example

Sample codes for a matrix transposition implemented using different approaches are illustrated in Fig. 1. The ACC directive version (b) differs from the serial version (a) only by the additional "acc region" and "acc do directives. The CUDA version, shown in (c) for kernel code and (d) for host code, is much more complex.

**a) Serial (CPU) code**
1. real(8)::a(m,n),b(n,m)
2. do i = 1,m
3.   do j = 1,n
4.     b(j,i) = a(i,j)
5.   end do
6. end do

**b) ACC directive code**
1. real(8)::a(m,n),b(n,m)
2. !$acc region
3. !$acc do
4.   do i = 1,m
5.     !$acc do
6.       do j = 1,n
7.         b(j,i) = a(i,j)
8.     end do
9.   end do
10. !$acc end region

**c) CUDA kernel**
1. attributes(global) subroutine &
2.   mt_kernel(m,n,a,b)
3. real(8)::a(m,n),b(n,m)
4. integer,parameter::bsize = 16
5. j = (blockidx%x-1)*bsize + threadIdx%x
6. i = (blockidx%y-1)*bsize + threadIdx%y
7. b(j,i) = a(i,j)
8. end subroutine mt_kernel

**d) CUDA host code**
1. real(8),device,allocatable,dimension(:,:) &
2.   :: a_dv,b_dv
3. integer,parameter::bsize = 16
4. type(dim3)::dgrid,dblock
5. allocate(a_dv(m,n),b_dv(n,m))
6. a_dv = a   !copy data to device
7. dblock = dim3(bsize,bsize,1)
8. dgrid = dim3(m/bsize,n/bsize,1)
9. call mt_kernel<<<dgrid,dblock>> &
10.   (m,n,a_dv,b_dv)
11. b = b_dv   !copy data back to host
12. deallocate(a_dv, b_dv)

Fig. 1. Sample codes for matrix transposition: a) serial version, b) ACC directive version, and c,d) CUDA kernel and host codes
PGI Optimizing Fortran, C and C++ Compilers & Tools

PGI Workstation™ and PGI Server™ for x64
PGI optimizing multi-core x64 compilers for Linux, MacOS & Windows with support for debugging and profiling of local MPI processes. A complete OpenMP/MPI SDK for high performance computing on the latest Intel and AMD CPUs. Try | Buy

PGI CUDA Fortran and CUDA-x86 Compilers
CUDA Fortran enables GPU acceleration of HPC applications using the NVIDIA CUDA parallel programming model in a native optimizing Fortran 2003 compiler. Use CUDA-x86 to compile your CUDA C or CUDA Fortran program to run on x64 targets. Both products are compatible and interoperable with NVIDIA's CUDA C. Try | Buy

PGI Accelerator™ with OpenACC
PGI Accelerator C99 & Fortran enable high level programming of GPUs for x64+accelerators using OpenACC compiler directives. Portable, and easy to use for application domain experts. Try | Buy

The PGI CDK® Cluster Development Kit®
The PGI CDK includes optimizing Fortran/C/C++ compilers configured to debug and profile MPI and hybrid MPI/OpenMP HPC applications for Linux or Windows Clusters using the major open source MPI implementations or MSMPI. Try | Buy

PGI Visual Fortran® for Microsoft Windows
PGI Visual Fortran brings optimizing multi-core x64 Fortran with integrated OpenMP/MPI debugging to scientists & engineers on Microsoft Windows within Microsoft Visual Studio. Try | Buy

PGI Compilers for Mobile and Embedded Applications

PGI OpenCL Compiler for Multi-core ARM
PGCL™ is PGI’s optimizing OpenCL™ compiler for the ST-Ericsson NovaThor integrated mobile platform. NovaThor is based on ARM® processing core with features that includes advanced graphics cores, powerful multimedia engines, and mobile broadband and connectivity technologies. Download
PGI Accelerator Compiler

- PGI Acceleration Model already directive-based
- OpenACC directives supersede PGIs one
- Implements all OpenACC directives
- Offloading of only marked code regions

- Techniques:
  - Data-level and task-level parallelism
  - Loop unrolling, parallelization, vectorization
  - Reduction detection
  - Cache management

Another OpenACC Compiler: accULL
PGI Accelerator Compiler

- Useful but missing:
  - within a loop no synchronization points can be specified
  - no user-defined reduction possible
  - non-inlined function calls within a accelerator region not possible
  - no detection or specification of critical/atomic regions possible
  - No asynchronous data copies
  - No automatic use of multiple GPUs attached to a host
Compiler Comparison

HotSpot Simulation Problem

```c
#pragma acc data copyin(_power[0:row*col],
                     _resultado[0:row*col]) copy(_temp[0:row*col])
{
  for (i = 0; i < num_iterations; i++) {
    #pragma acc kernels loop private(r)
    independent
    for (r = 0; r < row; r++) {
      #pragma acc loop private (c) independent
      for (c = 0; c < col; c++) {
        double delta;
        // Start computation
      }
    }
  }
}
```
Compiler Comparison

Abstract
— Using GPUs for general purpose programming is, nowadays, much easier than the previous years. In the very beginning were Brook-GPU or Close To Metal the approaches used for exploring the new possibilities of hardware accelerators. After that, CUDA and OpenCL were released. They had been adopted by many programmers due to theirs advantages but, however, both of them are quite complex for beginners. We need to find a way to leverage the programming effort otherwise, developers will spend most of their time focusing on device-specific code instead of implementing algorithmic enhancements.

The recent advent of the OpenACC standard [1] for heterogeneous computing represents an effort in this direction. Users only need to annotate the parallel regions to be offloaded into a GPU. We have developed our own version of the standard, accULL.

The main aim of this work has been to create a comparison between accULL and two different compilers that support the new standard: PGI and CAPS.
Memory Access

Access to global memory is very slow (400 cycles), so accesses should be coalesced if possible. Essentially this means that consecutive threads should access consecutive sections of memory.

**Coalesced accesses**

- Thread 0: Address 128
- Thread 1: Address 132
- Thread 2: Address 136
- Thread 3: Address 140
- Thread 4: Address 144
- Thread 5: Address 148
- Thread 6: Address 152
- Thread 7: Address 156
- Thread 8: Address 160
- Thread 9: Address 164
- Thread 10: Address 168
- Thread 11: Address 172
- Thread 12: Address 176
- Thread 13: Address 180
- Thread 14: Address 184
- Thread 15: Address 188

**Non-coalesced accesses**

- Thread 0: Address 128
- Thread 1: Address 132
- Thread 2: Address 136
- Thread 3: Address 140
- Thread 4: Address 144
- Thread 5: Address 148
- Thread 6: Address 152
- Thread 7: Address 156
- Thread 8: Address 160
- Thread 9: Address 164
- Thread 10: Address 168
- Thread 11: Address 172
- Thread 12: Address 176
- Thread 13: Address 180
- Thread 14: Address 184
- Thread 15: Address 188