RWTH AACHEN UNIVERSITY

OpenACC Basics

Directive-based GPGPU Programming

Sandra Wienke, M.Sc.
wienke@rz.rwth-aachen.de
Center for Computing and Communication
RWTH Aachen University

PPCES, March 2013

Agenda

- **Motivation & Overview**
- **Execution & Memory model**
- **Directives (in examples)**
  - → Offload regions (kernels, parallel & loops)
  - → Data movement (data region, data clauses, array shaping, update data)
- **Runtime Library Routines**
- **Development Tips**
- **Summary**

Example SAXPY – CPU

```
void saxpyCPU(int n, float a, float *x, float *y) {
    for (int i = 0; i < n; ++i)
        y[i] = a*x[i] + y[i];
}

int main(int argc, const char* argv[]) {
  int n = 10240; float a = 2.0f;
  float *x = (float*) malloc(n * sizeof(float));
  float *y = (float*) malloc(n * sizeof(float));

  // Initialize x, y
  for(int i=0; i<n; ++i){
    x[i]=i;
    y[i]=5.0*i-1.0;
  }

  // Invoke serial SAXPY kernel
  saxpyCPU(n, a, x, y);

  free(x); free(y);
  return 0;
}
```

SAXPY = Single-precision real Alpha X Plus Y

$$\vec{y} = \alpha \cdot \vec{x} + \vec{y}$$

Example SAXPY – OpenACC

```
void saxpyOpenACC(int n, float a, float *x, float *y) {
#pragma acc parallel loop vector_length(256)
    for (int i = 0; i < n; ++i)
        y[i] = a*x[i] + y[i];
}

int main(int argc, const char* argv[]) {
  int n = 10240; float a = 2.0f;
  float *x = (float*) malloc(n * sizeof(float));
  float *y = (float*) malloc(n * sizeof(float));

  // Initialize x, y
  for(int i=0; i<n; ++i){
    x[i]=i;
    y[i]=5.0*i-1.0;
  }

  // Invoke serial SAXPY kernel
  saxpyOpenACC(n, a, x, y);

  free(x); free(y);
  return 0;
}
```

Example SAXPY – CUDA

```
__global__ void saxpy_parallel(int n,
   float a, float *x, float *y) {
    int i = blockIdx.x * blockDim.x +
    threadIdx.x;
    if (i < n){
        y[i] = a*x[i] + y[i];
    }
}

int main(int argc, char* argv[]) {
 int n = 10240; float a = 2.0f;
 float* h_x,*h_y; // Pointer to CPU memory
 h_x = (float*) malloc(n* sizeof(float));
 h_y = (float*) malloc(n* sizeof(float));
 // Initialize h_x, h_y
 for(int i=0; i<n; ++i){
   h_x[i]=i;
   h_y[i]=5.0*i-1.0;
 }
 float                                  emory
 cudaM
 cudaM
```

1. Allocate data on GPU + transfer data to CPU

```
cudaMemcpy(d_x, h_x, n * sizeof(float),
  cudaMemcpyHostToDevice);
cudaMemcpy(d_y, h_y, n * sizeof(float),
  cudaMemcpyHostToDevice);

// Invoke parallel SAXPY kernel
dim3 threadsPerBlock(128);
dim3 b                              .x);
saxpy_parallel<<<blocksPerGrid,
  threadsPerBlock>>>(n, 2.0, d_x, d_y);

cudaM                                  t),
  cuda

cudaFree(d_x); cudaFree(d_y);
free(h_x); free(h_y);
return 0;
}
```

2. Launch kernel

3. Transfer data to CPU + free data on GPU

Introduction

- **Nowadays: low-level GPU APIs (like CUDA, OpenCL) often used**
  - → Unproductive development process
- **Directive-based programming model delegates responsibility for low-level GPU programming tasks to compiler**
  - → Data movement
  - → Kernel execution
  - → “Awareness” of particular GPU type
  - → …

→ **OpenACC**

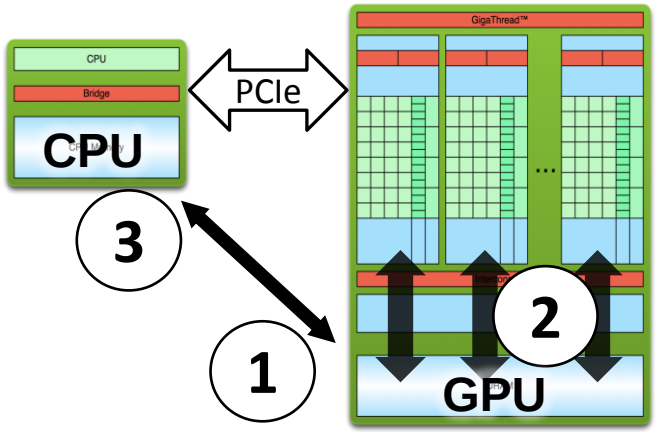
Directive-based model to offload compute-intensive loops to attached accelerator

Overview *OpenACC*

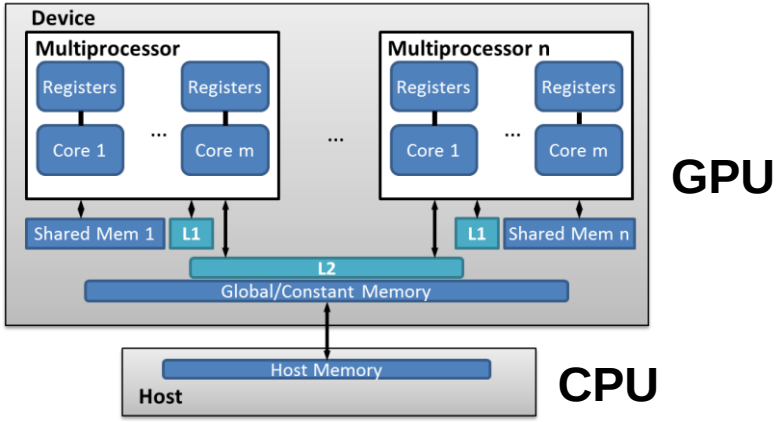
- **Open industry standard**
  - → Portability
- **Introduced by CAPS, Cray, NVIDIA, PGI (Nov. 2011)**
- **Support**
  - → C,C++ and Fortran
  - → NVIDIA GPUs, AMD GPUs & Intel MIC (now/near future)
- **Timeline**
  - → Nov'11: Specification 1.0
  - → Q1/Q2'12: Cray, PGI & CAPS compiler understands parts of OpenACC API
  - → Nov'12: Proposed additions for OpenACC 2.0
    More supporters, e.g. TUD, RogueWave

Execution & Memory Model (Recap)

- **Host-directed execution model**



- **Separate host and device memories**



GPU

CPU

→ Many tasks can be done by compiler/ runtime
→ User-directed programming

Directives

- **Syntax**

C

```
#pragma acc directive-name [clauses]
```

Fortran

```
!$acc directive-name [clauses]
```

- **Iterative development process**
- **Compiler feedback helpful**
  - → Whether an accelerator kernel could be generated
  - → Which loop schedule is used
  - → Where/which data is copied

Directives (in examples): SAXPY serial (host)

```
int main(int argc, const char* argv[]) {
  int n = 10240; float a = 2.0f; float b = 3.0f;
  float *x = (float*) malloc(n * sizeof(float));
  float *y = (float*) malloc(n * sizeof(float));
  // Initialize x, y

  // Run SAXPY TWICE









  for (int i = 0; i < n; ++i){
        y[i] = a*x[i] + y[i];
  }




  for (int i = 0; i < n; ++i){
        y[i] = b*x[i] + y[i];
  }

  free(x); free(y); return 0;
}
```

Directives (in examples): SAXPY
acc kernels

RWTH AACHEN UNIVERSITY

```
int main(int argc, const char* argv[]) {
  int n = 10240; float a = 2.0f; float b = 3.0f;
  float *x = (float*) malloc(n * sizeof(float));
  float *y = (float*) malloc(n * sizeof(float));
  // Initialize x, y

  // Run SAXPY TWICE


#pragma acc kernels
{
  for (int i = 0; i < n; ++i){
        y[i] = a*x[i] + y[i];
  }


  for (int i = 0; i < n; ++i){
        y[i] = b*x[i] + y[i];
  }
}
  free(x); free(y); return 0;
}
```

Kernel 1

Kernel 2

PGI Compiler Feedback

```
main:
29,Generating copyin(x[:10240])
   Generating copy(y[:10240])
   Generating compute capability 2.0
   binary
31,Loop is parallelizable
   Accelerator kernel generated
   31, #pragma acc loop gang, vector(32)
37,Loop is parallelizable
   Accelerator kernel generated
   37, #pragma acc loop gang, vector(32)
```

Directives (in examples): SAXPY
acc parallel, acc loop, data clauses

```
int main(int argc, const char* argv[]) {
  int n = 10240; float a = 2.0f; float b = 3.0f;
  float *x = (float*) malloc(n * sizeof(float));
  float *y = (float*) malloc(n * sizeof(float));
  // Initialize x, y

  // Run SAXPY TWICE



#pragma acc parallel copy(y[0:n]) copyin(x[0:n])
#pragma acc loop
  for (int i = 0; i < n; ++i){
        y[i] = a*x[i] + y[i];
  }

#pragma acc parallel copy(y[0:n]) copyin(x[0:n])
#pragma acc loop
  for (int i = 0; i < n; ++i){
        y[i] = b*x[i] + y[i];
  }

  free(x); free(y); return 0;
}
```

Directives (in examples): SAXPY
acc data

```
int main(int argc, const char* argv[]) {
  int n = 10240; float a = 2.0f; float b = 3.0f;
  float *x = (float*) malloc(n * sizeof(float));
  float *y = (float*) malloc(n * sizeof(float));
  // Initialize x, y

  // Run SAXPY TWICE
#pragma acc data copyin(x[0:n])
{

#pragma acc parallel copy(y[0:n]) present(x[0:n])
#pragma acc loop
  for (int i = 0; i < n; ++i){
        y[i] = a*x[i] + y[i];
  }

#pragma acc parallel copy(y[0:n]) present(x[0:n])
#pragma acc loop
  for (int i = 0; i < n; ++i){
        y[i] = b*x[i] + y[i];
  }
}
  free(x); free(y); return 0;
}
```

OpenACC PGI Compiler

RWTH AACHEN UNIVERSITY

- **OpenACC compiler in RWTH Cluster**

  Environment: PGI Compiler

grey = background information

```
module switch intel pgi          # switch default Intel
                                 # compiler to PGI compiler
```

- **Compiling**

```
$CC -acc -ta=nvidia,cc20,5.0 -Minfo=accel saxpy.c
```

→ `$CC`	If PGI module is loaded, this maps to `pgcc`
→ `-acc`	Tells compiler to recognize OpenACC directives
→ `-ta=nvidia`	Specifies the target architecture → here: NVIDIA GPUs
→ `cc20`	Optional. Specifies target compute capability 2.0
→ `5.0`	Optional. Uses CUDA Toolkit 5.0 for code generation
→ `-Minfo=accel`	Optional. Compiler feedback for accelerator code

Directives: Offload regions

RWTH AACHEN UNIVERSITY

- **Offload region**
  - → Region maps to a CUDA kernel function

C/C++
```
#pragma acc parallel [clauses]
```
Fortran
```
!$acc parallel [clauses]

!$acc end parallel
```

- → User responsible for finding parallelism (loops)
- → **`acc loop`** needed for work-sharing
- → No automatic sync between several loops

C/C++
```
#pragma acc kernels[clauses]
```
Fortran
```
!$acc kernels[clauses]

!$acc end kernels
```

- → Compiler responsible for finding parallelism (loops)
- → **`acc loop`** directive only for tuning needed
- → Automatic sync between loops within kernels region

Directives: Offload regions

- **Clauses for compute constructs (`parallel`, `kernels`)**

	C/C++, Fortran
→ If *condition* true, acc version is executed	`if(condition)`
→ Executes async, see Tuning slides	`async [(int-expr)]`
→ Define number of parallel gangs (parallel only)	`num_gangs(int-expr)`
→ Define number of workers within gang (parallel only)	`num_workers(int-expr)`
→ Define length for vector operations (parallel only)	`vector_length(int-expr)`
→ Reduction with *op* at end of region (parallel only)	`reduction(op:list)`
→ H2D-copy at region start + D2H at region end	`copy(list)`
→ Only H2D-copy at region start	`copyin(list)`
→ Only D2H-copy at region end	`copyout(list)`
→ Allocates data on device, no copy to/from host	`create(list)`
→ Data is already on device	`present(list)`
→ Test whether data on device. If not, transfer	`present_or_*(list)`
→ See Tuning slides	`deviceptr(list)`
→ Copy of each *list*-item for each parallel gang (parallel only)	`private(list)`
→ As `private` + copy initialization from host (parallel only).	`firstprivate(list)`

Directives: Loops

- **Share work of loops**
  - → Loop work gets distributed among threads on GPU (in certain schedule)

```
C/C++
#pragma acc loop[clauses]

Fortran
!$acc loop[clauses]
```

`kernels` `loop` defines loop schedule by int-expr in `gang`, `worker` or `vector` (instead of num_gangs etc with `parallel`)

- **Loop clauses**
  - → Distributes work into thread blocks ……..
  - → Distributes work into warps ……
  - → Distributes work into threads within warp/ thread block ……
  - → Executes loop sequentially on the device……
  - → Collapse *n* tightly nested loops……
  - → Says independent loop iterations(kernels loop only)….
  - → Reduction with *op*……
  - → Private copy for each loop iteration……

```
C/C++, Fortran
gang
worker
vector
seq
collapse(n)
independent
reduction(op:list)
private(list)
```

Loop schedule

Directives: Data regions & clauses

RWTH AACHEN UNIVERSITY

- **Data region**
  - → Decouples data movement from offload regions

C/C++
```
#pragma acc data [clauses]
```

Fortran
```
!$acc data [clauses]

!$acc end data
```

- **Data clauses**
  - → Triggers data movement of denoted arrays
  - → If *cond* true, move data to accelerator……
  - → H2D-copy at region start + D2H at region end ……
  - → Only H2D-copy at region start ……
  - → Only D2H-copy at region end ……
  - → Allocates data on device, no copy to/from host ……
  - → Data is already on device ……
  - → Test whether data on device. If not, transfer……
  - → See Tuning slides……

C/C++, Fortran
```
if(cond)
copy(list)
copyin(list)
copyout(list)
create(list)
present(list)
present_or_*(list)
deviceptr(list)
```

Misc on Loops

RWTH AACHEN UNIVERSITY

- **Combined directives**

```
#pragma acc kernels loop
for (int i=0; i<n; ++i) { /*...*/}

#pragma acc parallel loop
for (int i=0; i<n; ++i) { /*...*/}
```

- **Reductions**

```
#pragma acc parallel
#pragma acc loop reduction(+:sum)
for (int i=0; i<n; ++i) {
  sum += i;
}
```

PGI compiler can often recognize reductions on its own. See compiler feedback, e.g.:
`Sum reduction generated for var`

Data Movement

- **Data clauses can be used on `data`, `kernels` or `parallel`**
  - → `copy, copyin, copyout, present, present_or_copy, create, deviceptr`

- **Array shaping**
  - → Compiler sometimes cannot determine size of arrays
  - → Specify explicitly using data clauses and array “shape”

[lower bound: size]

C/C++

```
#pragma acc data copyin(a[0:length]) copyout(b[s/2:s/2])
```

Fortran

```
!$acc data copyin(a(0:length-1)) copyout(b(s/2:s))
```

[lower bound: upper bound]

Directives (in examples)
acc update

RWTH AACHEN UNIVERSITY

Rank 0

```
for (t=0; t<T; t++){
  // Compute x

  MPI_SENDRECV(x, …)

  // Adjust x
}
```

Rank 1

```
#pragma acc data copy(x[0:n])
{
  for (t=0; t<T; t++){
    // Compute x on device
    #pragma acc update host(x[0:n])
    MPI_SENDRECV(x, …)
    #pragma acc update device(x[0:n])
    // Adjust x on device
  }
}
```

Directives: Update

RWTH AACHEN UNIVERSITY

- **Update executable directive**
  - → Move data from GPU to host, or host to GPU
  - → Used to update existing data after it has changed in its corresponding copy

C/C++
```
#pragma acc update host|device [clauses]
```

Fortran
```
!$acc update host|device [clauses]
```

  - → Data movement can be conditional or asynchronous

- **Update clauses**
  - → *list* variables are copied from acc to host……
  - → *list* variables are copied from host to acc……
  - → If *cond* true, move data to accelerator……
  - → Executes async, see Tuning slides……

C/C++, Fortran
```
host(list)
device(list)
if(cond)
async[(int-expr)]
```

Runtime library routines

- **Interface to runtime routines & data types**

C/C++
```
#include “openacc.h”
```

Fortran
```
use openacc
```

- **Initialization of device**

→ E.g. to exclude initialization time from computation time

C/C++, Fortran
```
acc_init(devicetype)
```

- **Specify the device to run on**
- **…**

Development Tips

- **Favorable for parallelization: (nested) loops**
  - → Large loop counts to compensate (at least) data movement overhead
  - → Independent loop iterations
- **Think about data availability on host/ GPU**
  - → Use data regions to avoid unnecessary data transfers
  - → Specify data array shaping (may adjust for alignment)
- **Inline function calls in directives regions**
  - → PGI compiler flag: `-Minline` or `-Minline=levels:<N>`
  - → Call support for OpenACC 2.0 intended
- **Conditional compilation for OpenACC**
  - → Macro `_OPENACC`

Development Tips

- **Using pointers (C/C++)**
  - → Problem: compiler can not determine whether loop iterations that access pointers are independent → no parallelization possible
  - → Solution (if independence is actually there)
    - →Use restrict keyword: `float *restrict ptr;`
    - →Use compiler flag: `-ansi-alias`
    - →Use OpenACC loop clause: `independent`
- **Verifying execution on GPU**
  - → See PGI compiler feedback. Term "`Accelerator kernel generated`" needed.
  - → See information on runtime (more details in Tools slides):

    ```
    export ACC_NOTIFY=1
    ```

Summary

- **Constructs for basic parallelization on GPU**

➔ **Often productive development process, but knowledge about GPU programming concepts and hardware still needed**

- **Productivity may come at the cost of performance**
  - → But, potentially good ratio of effort to performance

- **Important step towards (OpenMP-) standardization**
  - → *OpenMP for Accelerators* technical report in Nov. 2012
  - → *OpenMP 4.0* specification intended for ISC 2013
- ***OpenMP for Accelerators* supported by many vendors/ architectures**
  - → NVIDIA and AMD GPUs
  - → Intel Many Integrated Core (MIC) architecture
  - → …