RWTH AACHEN UNIVERSITY

OpenACC Programming on GPUs

Directive-based GPGPU Programming

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

HiPerCH, April 2013

Agenda

- **Motivation & Overview**
- **Execution & Memory Model**
- **Offload Regions**
  - → kernels, parallel & loop
  - → Loop Schedule
- **Data Movement**
- **Optimizing Memory Accesses**
  - → Global Memory Throughput
  - → Shared Memory
- **Heterogeneous Computing**
  - → Asynchronous Operations
  - → Multiple GPUs (& 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
 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
saxpy_parallel<<<blocksPerGrid,
  threadsPerBlock>>>(n, 2.0, d_x, d_y);

cudaM
  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

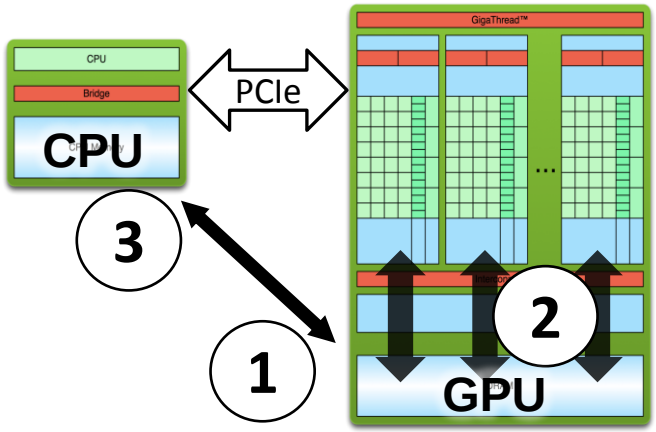
- **Open industry standard**
  - → Portability
- **Introduced by CAPS, Cray, NVIDIA, PGI (Nov. 2011)**
- **Support**
  - → C,C++ and Fortran
  - → NVIDIA GPUs, AMD GPUs & Intel MIC (now/near future)
- **Timeline**
  - → Nov'11: Specification 1.0
  - → Q1/Q2'12: Cray, PGI & CAPS compiler understands parts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

Agenda

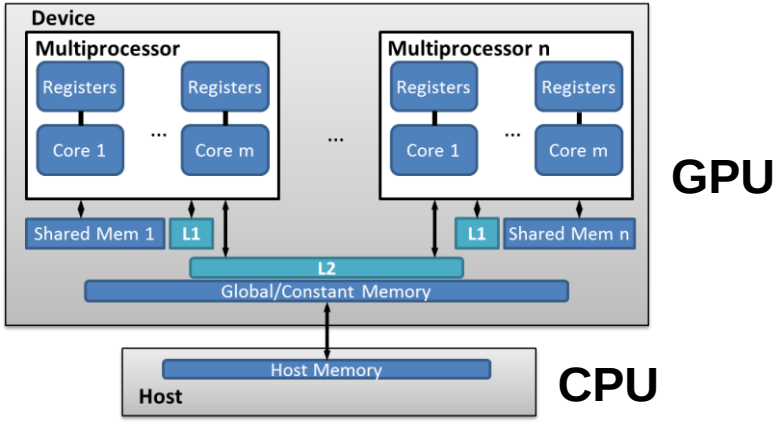
- Motivation & Overview
- **Execution & Memory Model**
- Offload Regions
  - → kernels, parallel & loop
  - → Loop Schedule
- Data Movement
- Optimizing Memory Accesses
  - → Global Memory Throughput
  - → Shared Memory
- Heterogeneous Computing
  - → Asynchronous Operations
  - → Multiple GPUs (& Runtime Library Routines)
- Development Tips
- Summary

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

Compiling (PGI)

```
pgcc -acc -ta=nvidia,cc30,5.0 -Minfo=accel saxpy.c
```

- → **pgcc** C PGI compiler (**pgf90** for Fortran)
- → **-acc** Tells compiler to recognize OpenACC directives
- → **-ta=nvidia** Specifies the target architecture → here: NVIDIA GPUs
- → **cc30** Optional. Specifies target compute capability 3.0
- → **5.0** Optional. Uses CUDA Toolkit 5.0 for code generation
- → **-Minfo=accel** Optional. Compiler feedback for accelerator code

We focus on PGI's OpenACC compiler here. For other compiler, compilation and feedback will look differently.

Agenda

- Motivation & Overview
- Execution & Memory Model
- **Offload Regions**
  - → kernels, parallel & loop
  - → Loop Schedule
- Data Movement
- Optimizing Memory Accesses
  - → Global Memory Throughput
  - → Shared Memory
- Heterogeneous Computing
  - → Asynchronous Operations
  - → Multiple GPUs (& Runtime Library Routines)
- Development Tips
- Summary

Offload Regions (in examples) 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;
}
```

Offload Regions (in examples)
acc kernels

```
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)
```

Offload Regions (in examples)
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: Offload regions

RWTH AACHEN UNIVERSITY

- **Offload region**

  → Region maps to a CUDA kernel function

grey = background information

```
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

Loop schedule:
- `gang`
- `worker`
- `vector`
- `seq`

- `collapse(n)`
- `independent`
- `reduction(op:list)`
- `private(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`

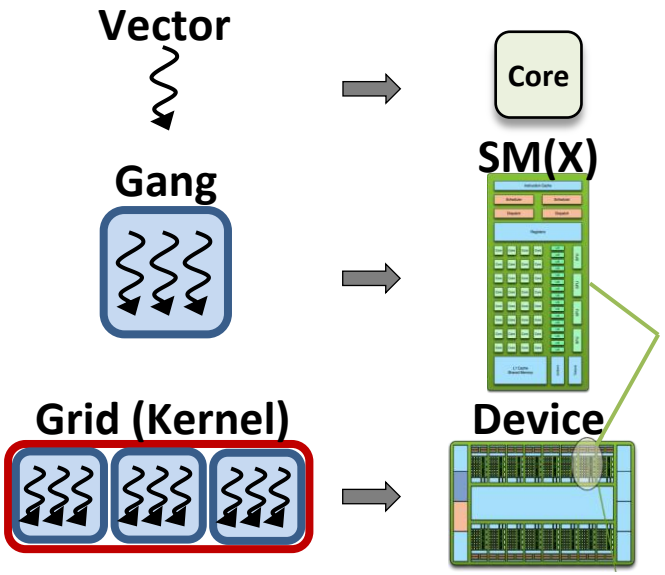
Agenda

- Motivation & Overview
- Execution & Memory Model
- **Offload Regions**
  - → kernels, parallel & loop
  - → Loop Schedule
- Data Movement
- Optimizing Memory Accesses
  - → Global Memory Throughput
  - → Shared Memory
- Heterogeneous Computing
  - → Asynchronous Operations
  - → Multiple GPUs (& Runtime Library Routines)
- Development Tips
- Summary

Mapping OpenACC on CUDA model

- **Possible mapping to CUDA terminology (GPUs)** *compiler dependent*
  - → gang = block
  - → worker = warp
  - → vector = threads
    - →Within block (if omitting worker)
    - →Within warp (if specifying worker)

- **Execution Model**



Loop schedule *(in examples)*

```
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];
  }                                   } Kernel 1



  for (int i = 0; i < n; ++i){
        y[i] = b*x[i] + y[i];
  }                                   } Kernel 2
}
  free(x); free(y); return 0;
}
```

Loop schedule *(in examples)*

```
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 copy(y[0:n]) copyin(x[0:n])
#pragma acc loop gang vector(256)
  for (int i = 0; i < n; ++i){
        y[i] = a*x[i] + y[i];
  }

#pragma acc kernels copy(y[0:n]) copyin(x[
#pragma acc loop gang(64) vector
  for (int i = 0; i < n; ++i){
        y[i] = b*x[i] + y[i];
  }

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

Without loop schedule

```
33, Loop is parallelizable
    Accelerator kernel generated
    33, #pragma acc loop gang,
    vector(128)
39, Loop is parallelizable
    Accelerator kernel generated
    39, #pragma acc loop gang,
    vector(128)
```

With loop schedule

```
33, Loop is parallelizable
    Accelerator kernel generated
    33, #pragma acc loop gang,
    vector(256)
39, Loop is parallelizable
    Accelerator kernel generated
    39, #pragma acc loop
    gang(64), vector(128)
```

Sometimes the compiler feedback is buggy. Check once with ACC_NOTIFY.

Loop schedule *(in examples)*

```
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])  vector_length(256)
#pragma acc loop gang vector
  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])  num_gangs(64)
#pragma acc loop gang vector
  for (int i = 0; i < n; ++i){
        y[i] = b*x[i] + y[i];
  }

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

`vector_length`: Optional: Specifies number of threads in thread block.
`num_gangs`: Optional. Specifies number of thread blocks in grid.

Loop schedule *(in examples)*

```
#pragma acc parallel vector_length(256)
#pragma acc loop gang vector
  for (int i = 0; i < n; ++i){
       // do something
  }
```

Distributes loop to **n** threads on GPU.
→256 threads per thread block
→usually `ceil(n/256)` blocks in grid

```
#pragma acc parallel vector_length(256)
#pragma acc loop gang vector
  for (int i = 0; i < n; ++i){
     for (int j = 0; j < m; ++j){
          // do something
  }  }
```

Distributes outer loop to **n** threads on GPU.
Each thread executes inner loop sequentially.
→256 threads per thread block
→usually `ceil(n/256)` blocks in grid

```
#pragma acc parallel vector_length(256) num_gangs(16)
#pragma acc loop gang vector
  for (int i = 0; i < n; ++i){
       // do something
  }
```

Distributes loop to threads on GPU (see above). If 16*256 < n, each thread gets multiple elements.
→256 threads per thread block
→16 blocks in grid

Loop schedule *(in examples)*

```
#pragma acc parallel vector_length(256)
#pragma acc loop gang
  for (int i = 0; i < n; ++i){
#pragma acc loop vector
     for (int j = 0; j < m; ++j){
          // do something
  }  }
```

Distributes outer loop to GPU multiprocessors (block-wise). Distributes inner loop to threads within thread blocks.

→256 threads per thread block

→usually **n** blocks in grid

```
#pragma acc kernels
#pragma acc loop gang(100) vector(8)
  for (int i = 0; i < n; ++i){
#pragma acc loop gang(200) vector(32)
     for (int j = 0; j < m; ++j){
          // do something
  }  }
```

Multidimensional portioning not yet possible with **parallel** construct.
→ See proposed **tile** clause in v2.0

With nested loops, specification of multidimensional blocks and grids possible: use same resource for outer and inner loop.

→100 blocks in Y-direction (rows); 200 blocks in X-direction (columns)

→8 threads in Y-dimension of one block; 21 threads in X-dimension of one block

→Total: 8*32=256 threads per thread block; 100*200=20,000 blocks in grid

Agenda

- Motivation & Overview
- Execution & Memory Model
- Offload Regions
  - → kernels, parallel & loop
  - → Loop Schedule
- **Data Movement**
- Optimizing Memory Accesses
  - → Global Memory Throughput
  - → Shared Memory
- Heterogeneous Computing
  - → Asynchronous Operations
  - → Multiple GPUs (& Runtime Library Routines)
- Development Tips
- Summary

Data Movement (in examples)

```
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;
}
```

Data Movement (in examples)
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;
}
```

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)
```

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”

C/C++

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

Fortran

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

[lower bound: size]

[lower bound: upper bound]

Data Movement (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)]
```

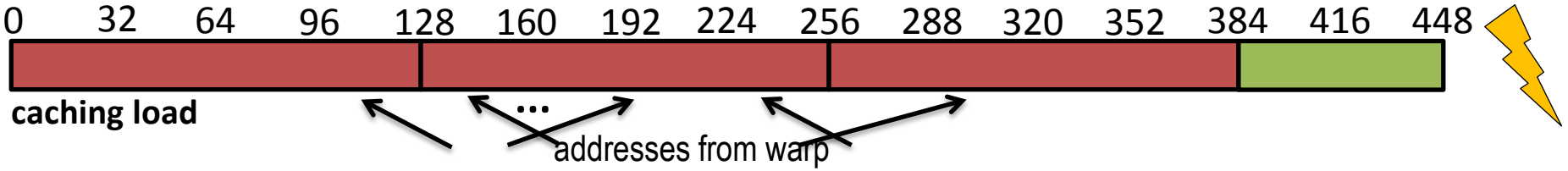
Agenda

- Motivation & Overview
- Execution & Memory Model
- Offload Regions
  - → kernels, parallel & loop
  - → Loop Schedule
- Data Movement
- **Optimizing Memory Accesses**
  - → Global Memory Throughput
  - → Shared Memory
- Heterogeneous Computing
  - → Asynchronous Operations
  - → Multiple GPUs (& Runtime Library Routines)
- Development Tips
- Summary

Global Memory Throughput (Recap)

- **Strive for perfect coalescing**
  - → Warp should access within contiguous region
  - → Align starting address (may require padding)

→ **Data layout matters (e.g. SoA over AoS)**



- **Have enough concurrent accesses to saturate the bus**
  - → Process several elements per thread
  - → Launch enough threads to cover access latency
- **Try caching or non-caching memory loads**
  - → PGI: caching is default;

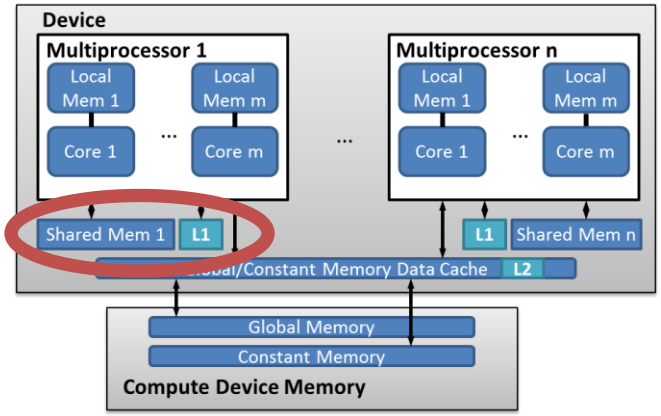
    non-caching with experimental compiler option: `-Mx,180,8`

Agenda

- Motivation & Overview
- Execution & Memory Model
- Offload Regions
  - → kernels, parallel & loop
  - → Loop Schedule
- Data Movement
- **Optimizing Memory Accesses**
  - → Global Memory Throughput
  - → Shared Memory
- Heterogeneous Computing
  - → Asynchronous Operations
  - → Multiple GPUs (& Runtime Library Routines)
- Development Tips
- Summary

Shared Memory

- **Smem per SM(X) (10s of KB)**
  - → Inter-thread communication within a block
  - → Need synchronization to avoid RAW / WAR / WAW hazards
- **Low-latency, high-throughput memory**
  - → Cache data in smem to reduce gmem accesses




```
#pragma acc kernels copyout(b[0:N][0:N]) copyin(a[0:N][0:N])
#pragma acc loop gang vector
  for (int i = 1; i < N-1; ++i){
#pragma acc loop gang vector
   for (int j = 1; j < N-1; ++j){
#pragma acc cache(a[i-1:i+1][j-1:j+1])
      b[i][j] = (a[i][j-1] + a[i][j+1] +
                 a[i-1][j] + a[i+1][j]) / 4;
  }    }
```

```
Accelerator kernel generated
65, #pragma acc loop gang, vector(2)
    Cached references to size
    [(y+2)x(x+2)] block of 'a'
67, #pragma acc loop gang, vector(128)
```

Directives: Caching

RWTH AACHEN UNIVERSITY

- **Cache construct**

  → Prioritizes data for placement in the highest level of data cache on GPU

C/C++
```
#pragma acc cache(list)
```

Fortran
```
!$acc cache(list)
```

Sometimes the PGI compiler ignores the `cache` construct.
→ See compiler feedback

  → Use at the beginning of the loop or in front of the loop

Agenda

- Motivation & Overview
- Execution & Memory Model
- Offload Regions
  - → kernels, parallel & loop
  - → Loop Schedule
- Data Movement
- Optimizing Memory Accesses
  - → Global Memory Throughput
  - → Shared Memory
- **Heterogeneous Computing**
  - → Asynchronous Operations
  - → Multiple GPUs (& Runtime Library Routines)
- Development Tips
- Summary

Asynchronous Operations

- **Definition**
  - → Synchronous: Control does not return until accelerator action is complete
  - → Asynchronous: Control returns immediately
    - →Allows heterogeneous computing (CPU + GPU)
- **Asynchronous operations with `async` clause**
  - → Kernel execution: `kernels`, `parallel`
  - → Data movement: `update`, PGI only: `data` construct

```
#pragma acc kernels async
for (int i=0; i<n; i++) {…}

// do work on host
```

Directives: async clause

RWTH AACHEN UNIVERSITY

- **Async clause**
  - → Executes **`parallel`** | **`kernels`** | **`update`** operation asynchronously while host process continues with code

C/C++

```
#pragma acc parallel|kernels|update async[(scalar-int-expr)]
```

Fortran

```
!$acc parallel|kernels|update async[(scalar-int-expr)]
```

  - → Integer argument (optional) can be seen as CUDA stream number
  - → Integer argument can be used in a wait directive
  - → Async activities with same argument: executed in order
  - → Async activities with diff. argument: executed in any order relative to each other

Streams

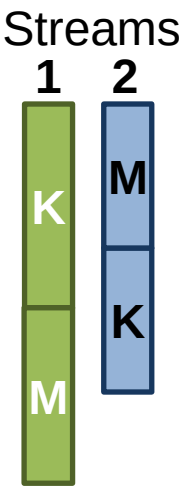
- **Streams = CUDA concept**
- **Stream = sequence of operations that execute in issue-order on GPU**
  - → Same int-expr in async clause: one stream
  - → Different streams/ int-expr: any order relative to each other
- **Tips for debugging - disable async (PGI only)**
  - → Use int-expr “-1” in async clause (PGI 13.x)
  - → Set ACC_SYNCHRONOUS=1

Wait on Asynchronous Operations

- **Synchronize async operations → wait directive**
  - → Wait for completion of an asynchronous activity (all or certain stream)

```
#pragma acc kernels loop async
for(int i=0; i<N/2; i++) {/* do work on device: x[0:N/2] */}
// do work on host: x[N/2:N]
#pragma acc update host(x[0:N/2]) async
#pragma acc wait
```

```
#pragma acc kernels loop async(1)
for(int i=0; i<N; i++) {/* do work on device: x[0:N] */}
#pragma acc update device(y[0:M]) async(2)
#pragma acc kernels loop async(2)
for(int i=0; i<M; i++) {/* do work on device: y[0:M] */}
#prama acc update host(x[0:N]) async(1)
// do work on host: z[0:L]
#pragma acc wait(1)
// do work on host: x[0:N]
#pragma acc wait(2)
```



Directives: wait

RWTH AACHEN UNIVERSITY

- **Wait directive**
  - → Host thread waits for completion of an asynchronous activity
  - → If integer expression specified, waits for all async activities with the same value
  - → Otherwise, host waits until all async activities have completed

C/C++
```
#pragma acc wait[(scalar-int-expr)]
```

Fortran
```
!$acc wait[(scalar-int-expr)]
```

- **Runtime routines**
  - → **`int acc_async_test(int)`**: tests for completion of all associated async activities; returns nonzero value/.true. if all have completed
  - → **`int acc_asnyc_test_all()`**: tests for completion of all async activities
  - → **`int acc_async_wait(int)`**: waits for completion of all associated async activites; routine will not return until the latest async activity has completed
  - → **`int acc_async_wait_all()`**: waits for completion of all asnyc activities

Agenda

- Motivation & Overview
- Execution & Memory Model
- Offload Regions
  - → kernels, parallel & loop
  - → Loop Schedule
- Data Movement
- Optimizing Memory Accesses
  - → Global Memory Throughput
  - → Shared Memory
- **Heterogeneous Computing**
  - → Asynchronous Operations
  - → Multiple GPUs (& Runtime Library Routines)
- Development Tips
- Summary

Multiple GPUs

- **Several GPUs within one compute node**
  - → Can use within one program (or use MPI)
  - → E.g. assign one device to each CPU thread/ process

```
#include <openacc.h>

acc_set_device_num(0, acc_device_nvidia);
#pragma acc kernels async

acc_set_device_num(1, acc_device_nvidia);
#pragma acc kernels async
```

Multiple GPUs – OpenMP + MPI

OpenMP

```
#include <openacc.h>
#include <omp.h>

#pragma omp parallel num_threads(12)
{
  int numdev = acc_get_num_devices(acc_device_nvidia);
  int devID = omp_get_thread_num() % numdev;
  acc_set_device_num(devID, acc_device_nvidia);
}
```

MPI

```
#include <openacc.h>
#include <mpi.h>

int myrank;
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
int numdev = acc_get_num_devices(acc_device_nvidia);
int devID = myrank % numdev;
acc_set_device_num(devID, acc_device_nvidia);
```

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)
```

- **…**

Agenda

- Motivation & Overview
- Execution & Memory Model
- Offload Regions
  - → kernels, parallel & loop
  - → Loop Schedule
- Data Movement
- Optimizing Memory Accesses
  - → Global Memory Throughput
  - → Shared Memory
- Heterogeneous Computing
  - → Asynchronous Operations
  - → Multiple GPUs (& Runtime Library Routines)
- **Development Tips**
- Summary

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: `-alias=ansi`
    - →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`

Agenda

- Motivation & Overview
- Execution & Memory Model
- Offload Regions
  - → kernels, parallel & loop
  - → Loop Schedule
- Data Movement
- Optimizing Memory Accesses
  - → Global Memory Throughput
  - → Shared Memory
- Heterogeneous Computing
  - → Asynchronous Operations
  - → Multiple GPUs (& Runtime Library Routines)
- Development Tips
- **Summary**

Summary

- **Constructs for basic parallelization on GPU**
  - → `kernels`, `parallel`, `loop`, `data`
- **Optimization possible**
  - → Loop Schedules (`gang`, `worker`, `vector`)
  - → Software caching (`cache`)
  - → Asynchronous Operations (`async`, `wait`)
  - → Interoperability with low-level kernels & libraries (not done here)
- **Proposal on OpenACC 2.0: new features**

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

- **Productivity may come at the cost of performance**

- **Important step towards (OpenMP-) standardization**
  - → *OpenMP 4.0 Release Candidate includes accelerator directives*

German Heterogeneous Computing Group (GHCG)

- **Usergroup Meeting 23.5.-24.5.2013 (2 half days)**
- **Workshop on programming accelerators 22.5.-23.5.2013 (2 half days)**