www.bsc.es

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

OpenACC and Other Approaches to GPU Computing

Antonio J. Peña

Based on material from NVIDIA's GPU Teaching Kit

Barcelona, July 4-6 2016

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

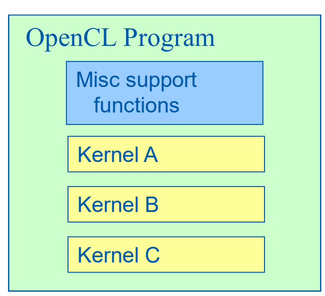
OPENCL

Background

- OpenCL was initiated by Apple and maintained by the Khronos Group (also home of OpenGL) as an industry standard API
  - For cross-platform parallel programming in CPUs, GPUs, DSPs, FPGAs,…
- OpenCL draws heavily on CUDA
  - Easy to learn for CUDA programmers
- OpenCL host code is much more complex and tedious due to desire to maximize portability and to minimize burden on vendors

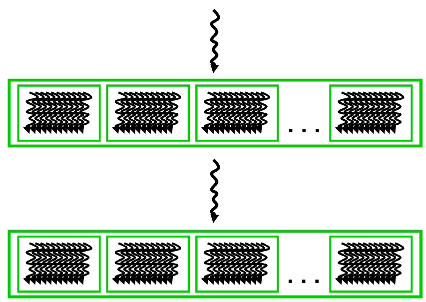
OpenCL Programs

- An OpenCL "program" is a C program that contains one or more "kernels" and any supporting routines that run on a target device
- An OpenCL kernel is the basic unit of parallel code that can be executed on a target device



OpenCL Execution Model

- Integrated host+device app C program
  - Serial or modestly parallel parts in host C code
  - Highly parallel parts in device SPMD kernel C code



Mapping Between OpenCL and CUDA Data Parallelism Model concepts

OpenCL Parallelism Concept	CUDA Equivalent
host	host
device	device
kernel	kernel
host program	host program
NDRange (index space)	grid
work item	thread
work group	block

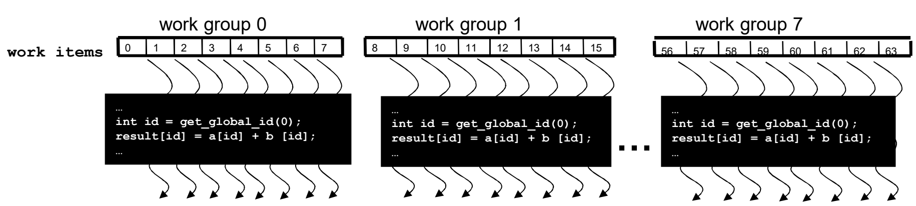
OpenCL Kernels

- Code that executes on target devices
- Kernel body is instantiated once for each work item
  - An OpenCL work item is equivalent to a CUDA thread
- Each OpenCL work item gets a unique index

```
__kernel void  vadd(__global const float *a,
                    __global const float *b,
                    __global float *result)
{
    int id = get_global_id(0);
    result[id] = a[id] + b[id];
}
```


Array of Work Items

- An OpenCL kernel is executed by an array of work items
  - All work items run the same code (SPMD)
  - Each work item can call get_global_id() to get its index for computing memory addresses and make control decisions



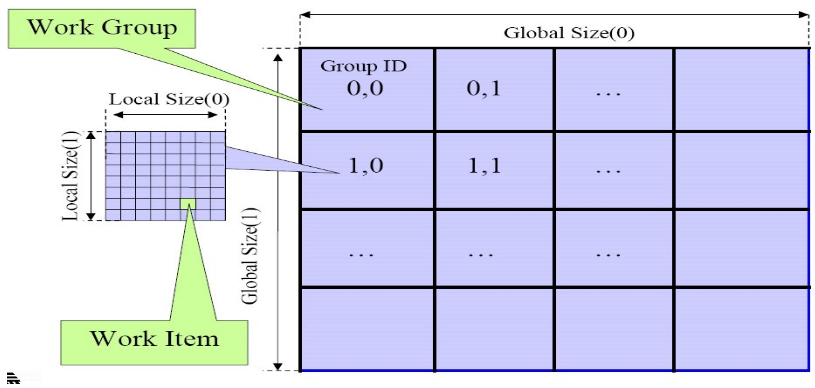
Work Groups: Scalable Cooperation

- Divide monolithic work item array into work groups
  - Work items within a work group cooperate via shared memory and barrier synchronization
  - Work items in different work groups cannot cooperate
- OpenCL counterpart of CUDA Thread Blocks

OpenCL Dimensions and Indices

OpenCL API Call	Explanation	CUDA Equivalent
get_global_id(0);	global index of the work item in the x dimension	blockIdx.x*blockDim.x +threadIdx.x
get_local_id(0)	local index of the work item within the work group in the x dimension	threadIdx.x
get_global_size(0);	size of NDRange in the x dimension	gridDim.x*blockDim.x
get_local_size(0);	Size of each work group in the x dimension	blockDim.x

Multidimensional Work Indexing

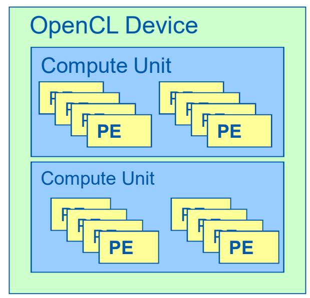


OpenCL Data Parallel Model Summary

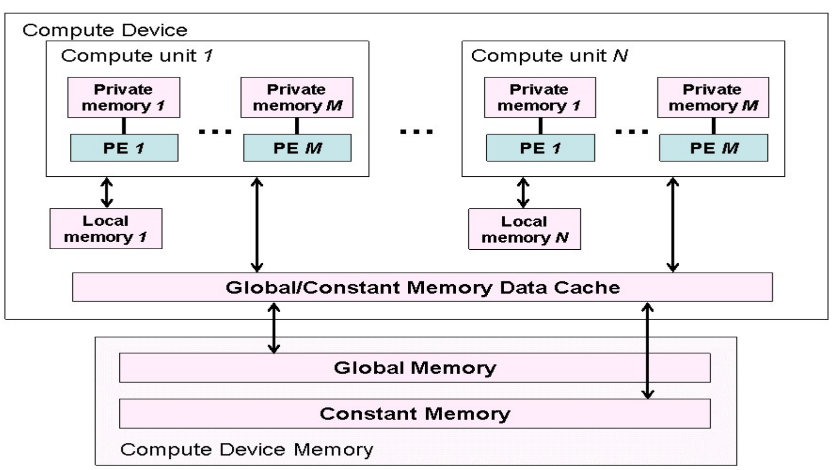
- Parallel work is submitted to devices by launching kernels
- Kernels run over global dimension index ranges (NDRange), broken up into “work groups”, and “work items”
- Work items executing within the same work group can synchronize with each other with barriers or memory fences
- Work items in different work groups can’t sync with each other, except by terminating the kernel

OpenCL Hardware Abstraction

- OpenCL exposes CPUs, GPUs, and other Accelerators as “devices”
- Each device contains one or more “compute units”, i.e. cores, Streaming Multicprocessors, etc...
- Each compute unit contains one or more SIMD “processing elements”, (i.e. SP in CUDA)



OpenCL Device Architecture



OpenCL Device Memory Types

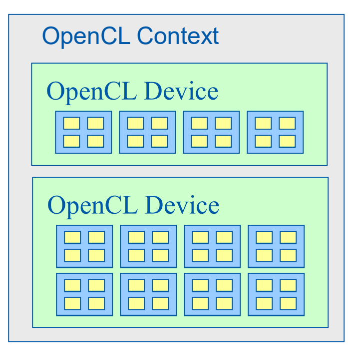
Memory Type	Host access	Device access	CUDA Equivalent
global memory	Dynamic allocation; Read/write access	No allocation; Read/write access by all work items in all work groups, large and slow but may be cached in some devices.	global memory
constant memory	Dynamic allocation; read/write access	Static allocation; read-only access by all work items.	constant memory
local memory	Dynamic allocation; no access	Static allocation; shared read-write access by all work items in a work group.	shared memory
private memory	No allocation; no access	Static allocation; Read/write access by a single work item.	registers and local memory

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

59

OpenCL Context

– Contains one or more devices
– OpenCL device memory objects are associated with a context, not a specific device



Barcelona Supercomputing Center
Centro Nacional de Supercomputación

60

OpenCL Context

- Contains one or more devices
- OpenCL memory objects are associated with a context, not a specific device
- clCreateBuffer() is the main data object allocation function
  - error if an allocation is too large for any device in the context
- Each device needs its own work queue(s)
- Memory copy transfers are associated with a command queue (thus a specific device)

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

61

OpenCL Context Setup Code (simple)

```
cl_int clerr = CL_SUCCESS;
cl_context clctx = clCreateContextFromType(0, CL_DEVICE_TYPE_ALL,
NULL, NULL, &clerr);

size_t parmsz;
clerr = clGetContextInfo(clctx, CL_CONTEXT_DEVICES, 0, NULL, &parmsz);

cl_device_id* cldevs = (cl_device_id *) malloc(parmsz);
clerr = clGetContextInfo(clctx, CL_CONTEXT_DEVICES, parmsz, cldevs,
NULL);

cl_command_queue clcmdq = clCreateCommandQueue(clctx, cldevs[0], 0,
&clerr);
```

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

62

OpenCL Kernel Compilation: vadd

OpenCL kernel source code as a big string

```
const char* vaddsrc =
   "__kernel void vadd(__global float *d_A, __global float *d_B,
__global float *d_C, int N) { \n"   [...etc and so forth...]
```

Gives raw source code string(s) to OpenCL

```
cl_program clpgm;
clpgm = clCreateProgramWithSource(clctx, 1, &vaddsrc, NULL,
&clerr);
```

Set compiler flags, compile source, and retrieve a handle to the "vadd" kernel

```
char clcompileflags[4096];
sprintf(clcompileflags, "-cl-mad-enable");
clerr = clBuildProgram(clpgm, 0, NULL, clcompileflags, NULL,
NULL);
cl_kernel clkern = clCreateKernel(clpgm, "vadd", &clerr);
```

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

OpenCL Device Memory Allocation

- clCreateBuffer();
  - Allocates object in the device Global Memory
  - Returns a pointer to the object
  - Requires five parameters
    - OpenCL context pointer
    - Flags for access type by device (read/write, etc.)
    - Size of allocated object
    - Host memory pointer, if used in copy-from-host mode
    - Error code
- clReleaseMemObject()
  - Frees object
    - Pointer to freed object

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

OpenCL Device Memory Allocation (cont.)

- Code example:
  - Allocate a 1,024 single precision float array
  - Attach the allocated storage to *d_a*
  - "*d_*" is often used to indicate a device data structure

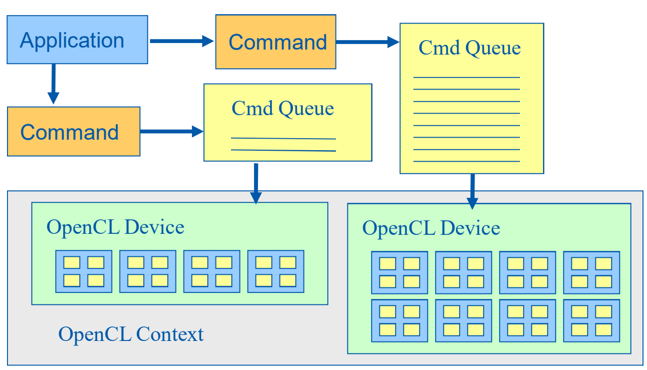
```
VECTOR_SIZE = 1024;
cl_mem d_a;
int size = VECTOR_SIZE* sizeof(float);

d_a = clCreateBuffer(clctx,
    CL_MEM_READ_ONLY, size, NULL, NULL);
…
clReleaseMemObject(d_a);
```

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

65

OpenCL Device Command Execution



Barcelona Supercomputing Center
Centro Nacional de Supercomputación

66

OpenCL Host-to-Device Data Transfer

- `clEnqueueWriteBuffer();`
  - Memory data transfer to device
  - Requires nine parameters
    - OpenCL command queue pointer
    - Destination OpenCL memory buffer
    - Blocking flag
    - Offset in bytes
    - Size (in bytes) of written data
    - Source host memory pointer
    - Number of events to be completed before execution of this command
    - List of events to be completed before execution of this command
    - Event object tied to this command

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

OpenCL Device-to-Host Data Transfer

- `clEnqueueReadBuffer();`
  - Memory data transfer to host
  - requires nine parameters
    - OpenCL command queue pointer
    - Source OpenCL memory buffer
    - Blocking flag
    - Offset in bytes
    - Size of bytes of read data
    - Destination host memory pointer
    - Number of events to be completed before execution of this command
    - List of events to be completed before execution of this command
    - Event object tied to this command

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

OpenCL Host-Device Data Transfer (cont.)

- Code example:
  - Transfer a *mem_size* single precision float array
  - *a* is in host memory and *d_a* is in device memory

```
clEnqueueWriteBuffer(clcmdq, d_a, CL_FALSE, 0,
            mem_size, (const void *)a, 0, 0, NULL);

clEnqueueReadBuffer(clcmdq, d_result, CL_FALSE, 0,
            mem_size, (void *) host_result, 0, 0, NULL);
```

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

OpenCL Host-Device Data Transfer (cont.)

- `clCreateBuffer` and `clEnqueueWriteBuffer` can be combined into a single command using special flags.
- Eg:

```
d_A=clCreateBuffer(clctxt, CL_MEM_READ_ONLY |
      CL_MEM_COPY_HOST_PTR,  mem_size, h_A, NULL);
```

- Combination of 2 flags here. `CL_MEM_COPY_HOST_PTR` to be used only if a valid host pointer is specified.
- This creates a memory buffer on the device, and copies data from h_A into d_A.
- Includes an implicit `clEnqueueWriteBuffer` operation, for all devices/command queues tied to the context clctxt.

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Device Memory Allocation and Data Transfer for *vadd*

```
float *h_A = …,   *h_B = …;
// allocate device (GPU) memory
cl_mem d_A, d_B, d_C;
d_A = clCreateBuffer(clctx, CL_MEM_READ_ONLY |
       CL_MEM_COPY_HOST_PTR, N *sizeof(float), h_A, NULL);
d_B = clCreateBuffer(clctx, CL_MEM_READ_ONLY |
       CL_MEM_COPY_HOST_PTR, N *sizeof(float), h_B, NULL);
d_C = clCreateBuffer(clctx, CL_MEM_WRITE_ONLY, N *sizeof(float),
       NULL, NULL);
```

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

71

Device Kernel Configuration Setting for *vadd*

```
clkern = clCreateKernel(clpgm, "vadd", NULL);
…
clerr = clSetKernelArg(clkern, 0, sizeof(cl_mem),(void *)&d_A);
clerr = clSetKernelArg(clkern, 1, sizeof(cl_mem),(void *)&d_B);
clerr = clSetKernelArg(clkern, 2, sizeof(cl_mem),(void *)&d_C);
clerr = clSetKernelArg(clkern, 3, sizeof(int), &N);
```

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

72

Device Kernel Launch and Remaining Code for *vadd*

```
cl_event event = NULL;
llerr = clEnqueueNDRangeKernel(clcmdq, clkern, 2, NULL,
      Gsz, Bsz, 0, NULL, &event);
clerr = clWaitForEvents(1, &event);
clEnqueueReadBuffer(clcmdq, d_C, CL_TRUE, 0,
      N*sizeof(float), h_C, 0, NULL, NULL);
clReleaseMemObject(d_A);
clReleaseMemObject(d_B);
clReleaseMemObject(d_C);
}
```

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

73

Question 1

- In comparing OpenCL and CUDA, which of the following is not a valid comparison?

  a) A compute unit in OpenCL is like a streaming processor in CUDA
  b) An NDRange in OpenCL is like a grid in CUDA
  c) A work-item in OpenCL is like a thread in CUDA
  d) A work-group in OpenCL is like a thread block in CUDA

Question 2

- In comparing OpenCL and CUDA, which of the following is not a valid comparison?

  a) get_local_id(0) in OpenCL is like threadIdx.x in CUDA
  b) get_local_id(1) in OpenCL is like threadIdx.y in CUDA
  c) get_local_size(0) in OpenCL is like blockDim.x in CUDA
  d) get_global_size(0) in OpenCL is like gridDim.x in CUDA

Question 3

- In comparing OpenCL and CUDA, which of the following is not a valid comparison?

  a) clCreateBuffer(…) in OpenCL is like cudaMalloc(…) in CUDA
  b) clEnqueueReadBuffer() in OpenCL is like cudaMemcpy(…) in CUDA
  c) clEnqueueWriteBuffer(…) in OpenCL is like cudaMemset(…) in CUDA
  d) clReleaseMemObject(…) in OpenCL is like cudaFree(…) in CUDA

Question 4

- Which of the following statements about OpenCL is not true?

  a) Whenever an OpenCL buffer is created with clCreateBuff(), it is created in all devices in the specified context.
  b) Input arguments to an OpenCL kernel must be passed in the clEnqueueKernel() call.
  c) OpenCL kernels are compiled with the clBuildProgram() call.
  d) OpenCL kernels are declared with the __kernel keyword.

OpenACC

- The OpenACC Application Programming Interface provides a set of
  - compiler directives (pragmas)
  - library routines and
  - environment variables

  that can be used to write data parallel Fortran, C and C++ programs that run on accelerator devices including GPUs and CPUs

OpenACC Pragmas

- In C and C++, the #pragma directive is the method to provide to the compiler information that is not specified in the standard language.
  - These pragmas extend the base language

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Vector Addition in OpenACC

```
void VecAdd(float * __restrict__ output, const float * input1, const float * input 2, int inputLength)
{
 #pragma acc parallel loop copyin(input1[0:inputLength],input2[0:inputLength]),
       copyout(output[0:inputLength])
   for(i = 0; i < inputLength; ++i) {
      output[i] = input1[i] + input2[i];
   }
}
```

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Simple Matrix-Matrix Multiplication in OpenACC

```
1. void computeAcc(float *P, const float *M, const float *N, int Mh, int Mw, int Nw)
2. {
3.  #pragma acc parallel loop copyin(M[0:Mh*Mw]) copyin(N[0:Mw*Nw]) copyout(P[0:Mh*Nw])
4.  for (int i=0; i<Mh; i++) {
5.    #pragma acc loop
6.     for (int j=0; j<Nw; j++) {
7.        float sum = 0;
8.        for (int k=0; k<Mw; k++) {
9.            float a = M[i*Mw+k];
10.           float b = N[k*Nw+j];
11.           sum += a*b;
12.       }
13.       P[i*Nw+j] = sum;
14.    }
15.  }
16. }
```


Some Observations (1)

```
1. void computeAcc(float *P, const float *M, const float *N, int Mh, int Mw, int Nw)
2. {
3.  #pragma acc parallel loop copyin(M[0:Mh*Mw]) copyin(N[0:Mw*Nw]) copyout(P[0:Mh*Nw])
4.  for (int i=0; i<Mh; i++) {
5.    #pragma acc loop
6.     for (int j=0; j<Nw; j++) {
7.        float sum = 0;
8.        for (int k=0; k<Mw; k++) {
9.            float a = M[i*Mw+k];
10.            float b = N[k*Nw+j];
11.            sum += a*b;
12.        }
13.        P[i*Nw+j] = sum;
14.     }
15.  }
16. }
```

The code is almost identical to the sequential version, except for the two lines with #pragma at line 3 and line 5.

Some Observations (2)

```
1. void computeAcc(float *P, const float *M, const float *N, int Mh, int Mw, int Nw)
2. {
3.  #pragma acc parallel loop copyin(M[0:Mh*Mw]) copyin(N[0:Mw*Nw]) copyout(P[0:Mh*Nw])
4.  for (int i=0; i<Mh; i++) {
5.    #pragma acc loop
6.    for (int j=0; j<Nw; j++) {
7.      float sum = 0;
8.      for (int k=0; k<Mw; k++) {
9.        float a = M[i*Mw+k];
10.       float b = N[k*Nw+j];
11.       sum += a*b;
12.     }
13.     P[i*Nw+j] = sum;
14.   }
15. }
16. }
```

The #pragma at line 3 tells the compiler to generate code for the 'i' loop at line 4 through 15 so that the loop iterations are executed at the first level of parallelism on the accelerator.

Some Observations (3)

```
1. void computeAcc(float *P, const float *M, const float *N, int Mh, int Mw, int Nw)
2. {
3.  #pragma acc parallel loop copyin(M[0:Mh*Mw]) copyin(N[0:Mw*Nw]) copyout(P[0:Mh*Nw])
4.  for (int i=0; i<Mh; i++) {
5.    #pragma acc loop
6.    for (int j=0; j<Nw; j++) {
7.      float sum = 0;
8.      for (int k=0; k<Mw; k++) {
9.        float a = M[i*Mw+k];
10.       float b = N[k*Nw+j];
11.       sum += a*b;
12.     }
13.     P[i*Nw+j] = sum;
14.   }
15. }
16. }
```

The copyin() clause and the copyout() clause specify how the compiler should arrange for the matrix data to be transferred between the host and the accelerator.

Some Observations (4)

```
1. void computeAcc(float *P, const float *M, const float *N, int Mh, int Mw, int Nw)
2. {
3.  #pragma acc parallel loop copyin(M[0:Mh*Mw]) copyin(N[0:Mw*Nw]) copyout(P[0:Mh*Nw])
4.  for (int i=0; i<Mh; i++) {
5.    #pragma acc loop
6.    for (int j=0; j<Nw; j++) {
7.      float sum = 0;
8.      for (int k=0; k<Mw; k++) {
9.        float a = M[i*Mw+k];
10.       float b = N[k*Nw+j];
11.       sum += a*b;
12.     }
13.     P[i*Nw+j] = sum;
14.   }
15. }
16. }
```

The #pragma at line 5 instructs the compiler to map the inner 'j' loop to the second level of parallelism on the accelerator.

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Motivation

– OpenACC programmers can often start with writing a sequential version and then annotate their sequential program with OpenACC directives.
  – leave most of the details in generating a kernel, memory allocation, and data transfers to the OpenACC compiler.

– OpenACC code can be compiled by non-OpenACC compilers by ignoring the pragmas.

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

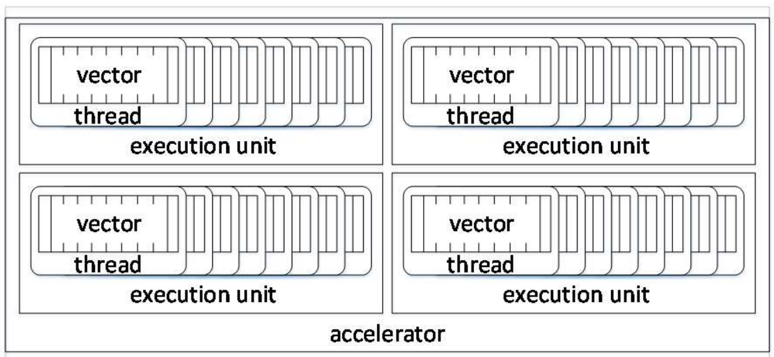
Frequently Encountered Issues

- Some OpenACC pragmas are hints to the OpenACC compiler, which may or may not be able to act accordingly
  - The performance of an OpenACC program depends heavily on the quality of the compiler.
  - It may be hard to figure out why the compiler cannot act according to your hints
  - The uncertainty is much less so for CUDA or OpenCL programs

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

93

OpenACC Device Model

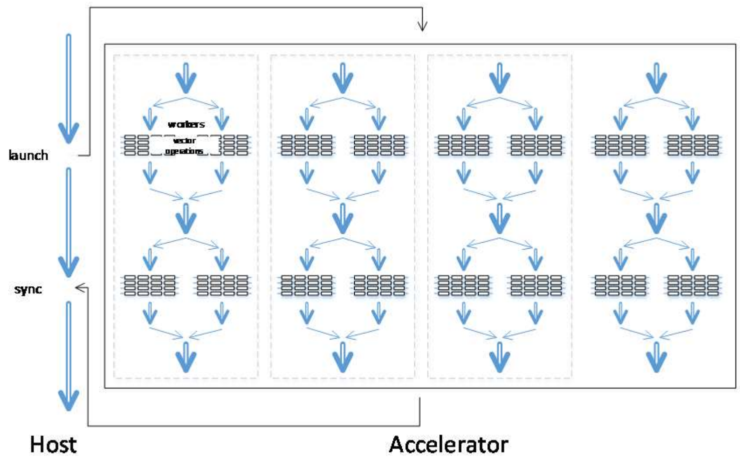


Currently OpenACC does not expose synchronization across threads to the programmers.

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

94

OpenACC Execution Model



Parallel vs. Loop Constructs

```
#pragma acc parallel loop copyin(M[0:Mh*Mw])
copyin(N[0:Mw*Nw]) copyout(P[0:Mh*Nw])
for (int i=0; i<Mh; i++) {
…
}
```

is equivalent to:

```
#pragma acc parallel copyin(M[0:Mh*Mw]) copyin(N[0:Mw*Nw])
copyout(P[0:Mh*Nw])
{
    #pragma acc loop
    for (int i=0; i<Mh; i++) {
        …
    }
}
```

(a parallel region that consists of a single loop)

More on Parallel Construct

```
#pragma acc parallel copyout(a) num_gangs(1024) num_workers(32)
{
    a = 23;
}
```

1024*32 workers will be created. a=23 will be executed redundantly by all 1024 gang leads

- A parallel construct is executed on an accelerator
- One can specify the number of gangs and number of workers in each gang
  - Equivalent to CUDA blocks and threads

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

97

What Does Each “Gang Loop” Do?

```
#pragma acc parallel num_gangs(1024)
{
    for (int i=0; i<2048; i++) {
        …
    }
}
```

```
#pragma acc parallel num_gangs(1024)
{
#pragma acc loop gang
    for (int i=0; i<2048; i++) {
        …
    }
}
```

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

98

Worker Loop

```
#pragma acc parallel num_gangs(1024) num_workers(32)
{
    #pragma acc loop gang
    for (int i=0; i<2048; i++) {
        #pragma acc loop worker
        for (int j=0; j<512; j++) {
            foo(i,j);
        }
    }
}
```

1024*32=32K workers will be created, each executing 1M/32K = 32 instance of foo()

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

99

A More Substantial Example

– Statements 1, 3, 5, 6 are redundantly executed by 32 gangs

```
#pragma acc parallel num_gangs(32)
{
    Statement 1;
    #pragma acc loop gang
    for (int i=0; i<n; i++) {
        Statement 2;
    }
    Statement 3;
    #pragma acc loop gang
    for (int i=0; i<m; i++) {
        Statement 4;
    }
    Statement 5;
    if (condition) Statement 6;
}
```

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

100

A More Substantial Example

- The iterations of the n and m for-loop iterations are distributed to 32 gangs
- Each gang could further distribute the iterations to its workers
  - The number of workers in each gang will be determined by the compiler/runtime

```
#pragma acc parallel num_gangs(32)
{
   Statement 1;
   #pragma acc loop gang
   for (int i=0; i<n; i++) {
      Statement 2;
   }
   Statement 3;
   #pragma acc loop gang
   for (int i=0; i<m; i++) {
      Statement 4;
   }
   Statement 5;
   if (condition) Statement 6;
}
```

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Avoiding Redundant Execution

- Statements 1, 3, 5, 6 will be executed only once
- Iterations of the n and m loops will be distributed to 32 workers

```
#pragma acc parallel
num_gangs(1) num_workers(32)
{
   Statement 1;
   #pragma acc loop worker
   for (int i=0; i<n; i++) {
      Statement 2;
   }
   Statement 3;
   #pragma acc loop worker
   for (int i=0; i<m; i++) {
      Statement 4;
  }
  Statement 5;
   if (condition)  Statement 6;
}
```

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Kernel Regions

- Kernel constructs are descriptive of programmer intentions
  - The compiler has a lot of flexibility in its use of the information
- This is in contrast with Parallel, which is prescriptive of the action for the compile follow

```
#pragma acc kernels
{
    #pragma acc loop gang(1024)
    for (int i=0; i<2048; i++) {
        a[i] = b[i];
    }
    #pragma acc loop gang(512)
    for (int j=0; j<2048; j++) {
        c[j] = a[j]*2;
    }
    for (int k=0; k<2048; k++) {
        d[k] = c[k];
    }
}
```

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

103

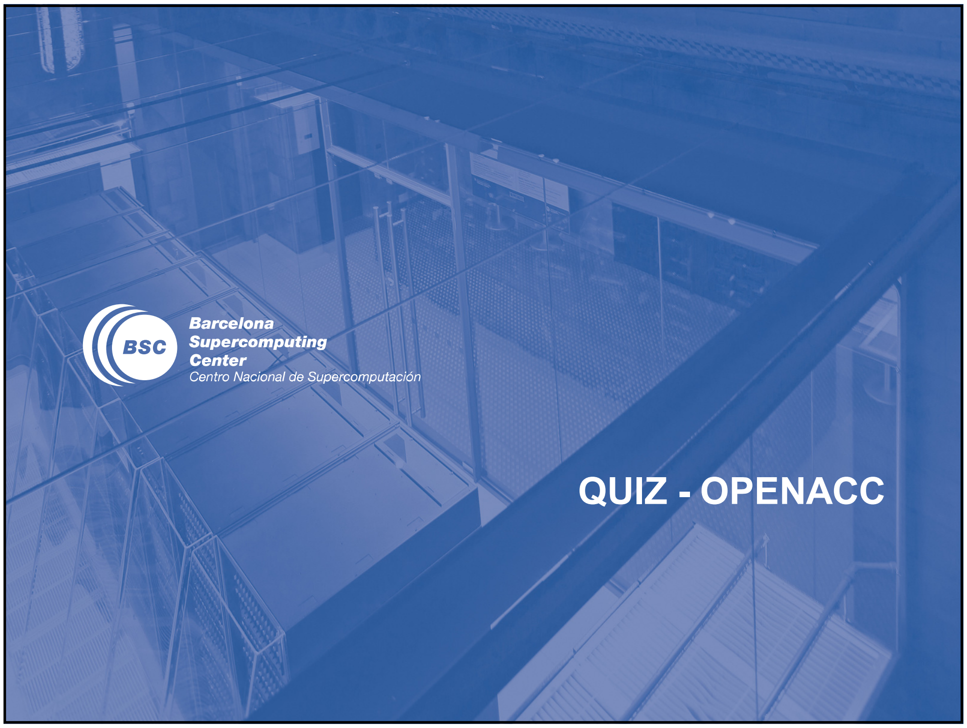
Kernel Regions

- Code in a kernel region can be broken into multiple CUDA/OpenCL kernels
- The i, j, k loops can each become a kernel
  - The k-loop may even remain as host code
- Each kernel can have a different gang/worker configuration

```
#pragma acc kernels
{
    #pragma acc loop gang(1024)
    for (int i=0; i<2048; i++) {
        a[i] = b[i];
    }
    #pragma acc loop gang(512)
    for (int j=0; j<2048; j++) {
        c[j] = a[j]*2;
    }
    for (int k=0; k<2048; k++) {
        d[k] = c[k];
    }
}
```

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

104



QUIZ - OPENACC

Question 1

- OpenACC can be used to parallelize

  a) C and C++ only
  b) C, C++, and C#
  c) C, C++, and Java
  d) C, C++, and FORTRAN

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

106

Question 2

- In comparing OpenACC and CUDA, which of the following is not a valid comparison?

  a) copyin(…) in OpenACC is like cudaMemcpy(…) in CUDA
  b) #barrier in OpenACC is like __syncthreads() in CUDA
  c) Gangs in OpenACC are like thread blocks in CUDA
  d) Workers in OpenACC are like threads in CUDA

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

108

Question 3

- How many times will foo() and bar() be executed?

```
#pragma acc parallel num_gangs(64)
{
  foo();
  #pragma acc loop gang
  for (int i=0; i<n; i++) bar(i);
}
```

  a) foo() once, bar() n times
  b) foo() 64 times, bar() n times
  c) foo() once, bar() 64*n times
  d) foo() 64 times, bar() 64*n times

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

110

www.bsc.es

Barcelona
Supercomputing
Center
Centro Nacional de Supercomputación

Thank you!

For further information please contact
antonio.pena@bsc.es