Heterogeneous Computing with OpenCL

We discuss the OpenCL specification and how it can be used to implement programs for heterogeneous platforms.

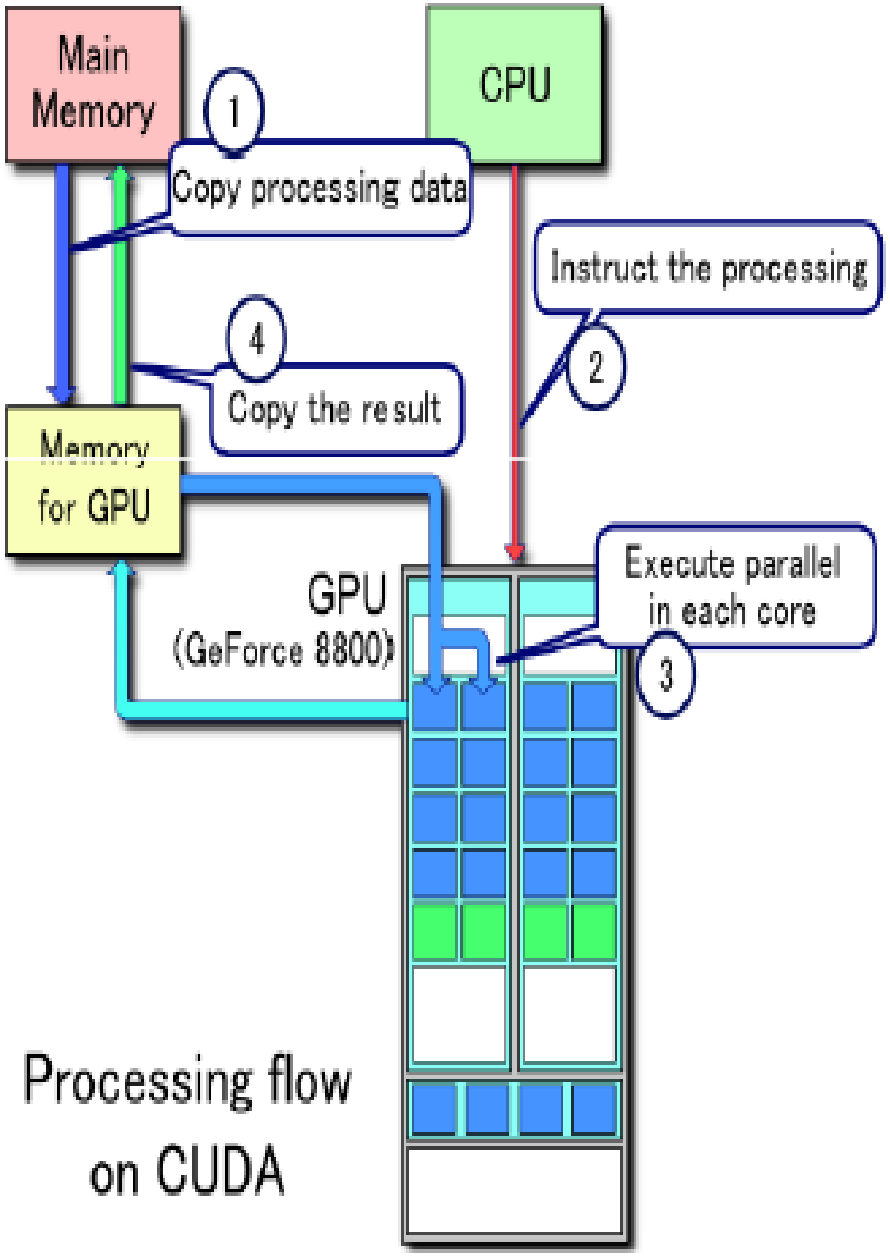
A = [1 2 3 4 5 6 7 8 9 10]

B = [9 8 7 6 5 4 3 2 1 0]

C = A + B
  = [10 10 10 10 10 10 10 10 10 10]

With MPI Can you write a pgm with two slaves?

- Copy data from main memory to GPU memory
- CPU instructs the process to GPU
- GPU executes the compute in parallel on each core
- Copy the result from GPU memory to main memory



What is Heterogeneous computing

- **Heterogeneous computing** systems refer to electronic systems…
- A computational unit could be a GPP, a special-purpose processor or GPU, a co-processor, or FPGA.
- Another term sometimes seen for this type of computing is “Hybrid computing“
- GPU programs are called *kernels.*

What is GPU Computing?

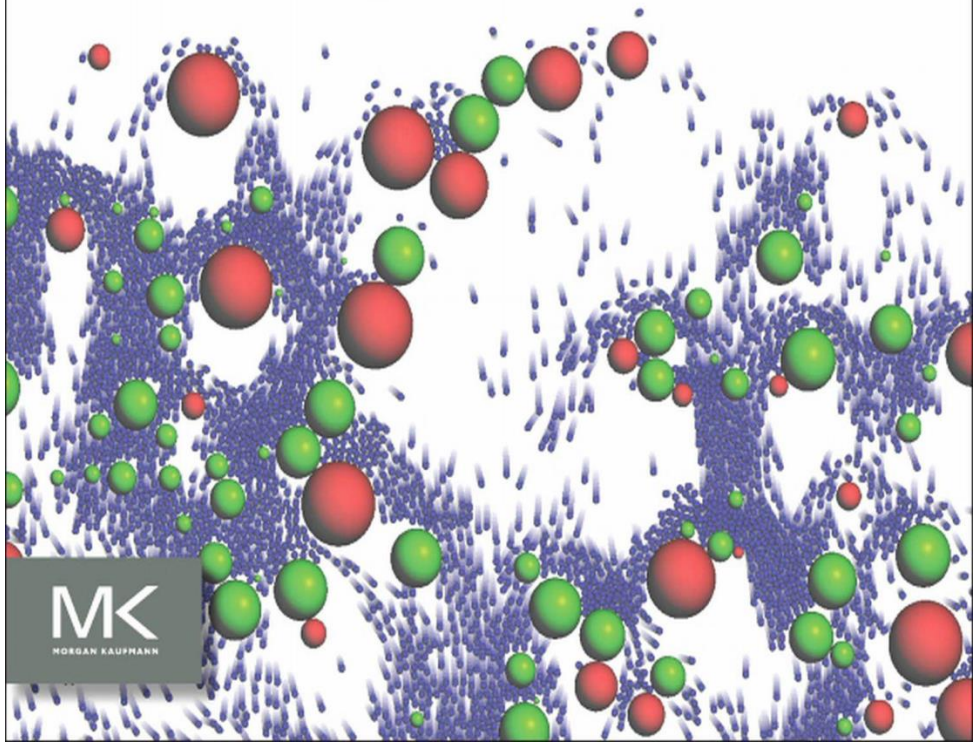
GPU computing…

Due to their massive-parallel architecture, GPU enables the computationally intensive assignments.

This is why GPU computing has enormous potential.

Benedict R. Gaster Lee Howes David Kaeli
Perhaad Mistry Dana Schaa

Heterogeneous Computing with OpenCL



Heterogeneous Computing with OpenCL

Benedict Gaster

Lee Howes

David R. Kaeli

Perhaad Mistry

Dana Schaa

ELSEVIER

AMSTERDAM • BOSTON • HEIDELBERG • LONDON
NEW YORK • OXFORD • PARIS • SAN DIEGO
SAN FRANCISCO • SINGAPORE • SYDNEY • TOKYO

Morgan Kaufmann is an imprint of Elsevier

The Open Computing Language (OpenCL) is a heterogeneous programming Framework.

OpenCL is a framework for developing applications.

It supports a wide range of levels of parallelism.

OpenCL provides parallel computing .

It currently supports CPUs and it has been adopted into graphics card drivers by both AMD and NVIDIA.

Support for OpenCL is rapidly expanding as a wide range of platform vendors have adopted OpenCL.

To name some of these vendors ARM, Imagination Technologies, the HPC vendors AMD, Intel, NVIDIA, and IBM.

The architectures supported include multi-core CPUs and vector processors such as GPUs.

OpenCL makes it an excellent programming model.

GPUs have evolved rapidly as very powerful, fully programmable, task and data-parallel architectures.

Combining CPUs and GPUs on a single die.

Compute-intensive and data-intensive portions of a given application, called kernels, may be offloaded to the GPU.

The degree of parallelism that can be achieved is dependent on the skill of the algorithm or software designer.

CH 2
Introduction to OpenCL

The OpenCL Standard

The Khronos group has developed API.

Using the core language and correctly following the specification.

The model set by OpenCL creates portable, vendor- and device-independent programs.

The OpenCL API is a C with a C++Wrapper API that is defined in terms of the C API.

OpenCL C is a restricted version of the C, with extensions appropriate for executing data-parallel code on a variety of heterogeneous devices.

The OpenCL Specification:

is defined in four parts :

1. *Platform model*: Specifies that there is one processor coordinating execution (the host) and one or more processors capable of executing OpenCL C code (the devices).

   It defines an abstract hardware model. The platform model defines this relationship between the host and device.

2. *Execution model*: Defines how the OpenCL environment is configured.

   This includes setting up an OpenCL context on the host.

   The host sets up a kernel for the GPU to run and initiates it with specified degree of parallelism.

The OpenCL Specification (Contd..):

3. *Memory model*: Defines the abstract memory hierarchy that kernels use, regardless of the actual memory architecture.

   The memory model closely resembles current CPU memory hierarchies.

   The data within the kernel is allocated by the programmer to specific parts of an abstract memory hierarchy.

   The runtime and driver will map these abstract memory spaces to the physical hierarchy.

4. *Programming model*: Defines how the concurrency model is mapped to physical hardware.

   Finally, hardware thread contexts that execute the kernel must be created and mapped to actual GPU hardware.

Kernels and the OpenCL Execution Model

Kernels are the parts of an OpenCL program.

The OpenCL API enables an application to create a context + describing the movement of data + the execution of kernel .

An OpenCL kernel is syntactically similar to a standard C function .

Kernels and the OpenCL Execution Model (Contd..)

A serial C implementation, a threaded C implementation, and an OpenCL implementation.

The code for a *serial C implementation* of the **vector addition** executes a loop with as many iterations as there are elements to compute.

// Perform an element-wise addition of A and B and store in C.
// There are N elements per array.

```
void vecadd (int *C, int* A, int *B, int N)
{
    for(int i = 0; i < N; i++)
    {
        C[i] = A[i] + B[i];
    }
}
```

The code for the multithreaded version is:

```
// Perform and element-wise addition of A and B and store in C.
// There are N elements per array and N_P CPU cores.
        void vecadd (int *C, int* A, int *B, int N, int N_P, int tid)
        {
          int ept = N/N_P; // elements per thread
          for (int i = tid*ept; i < (tid+1)*ept; i++)
            {
               C[i] = A[i] + B[i];
            }   }
```

The unit of concurrent execution in OpenCL is a work-item.

Each work-item executes the kernel function body.

We map a single iteration to a work-item.

We tell the OpenCL runtime to generate as many work-items as elements in the input and output arrays and allow the OpenCL to map those work-items to the underlying hardware, (CPU / GPU cores).

When an OpenCL provides intrinsic functions that allow a work-item to identify itself.

Below, the call to get_global_id(0) allows the programmer to make use of the position of the current work-item :

```
// Perform an element-wise addition of A and B and store in C
// N work-items will be created to execute this kernel.
_ _kernel void vecadd (_ _global int *C,                    _ _global
int* A, _ _global int *B)
                {
                  int tid = get_global_id(0);  //
                            OpenCL intrinsic function
                  C[tid] = A[tid] + B[tid];
```

Work-items behave independently, OpenCL allows the local workgroup size to be ignored by the programmer and generated automatically by the implementation; in this case, the developer will pass NULL instead.

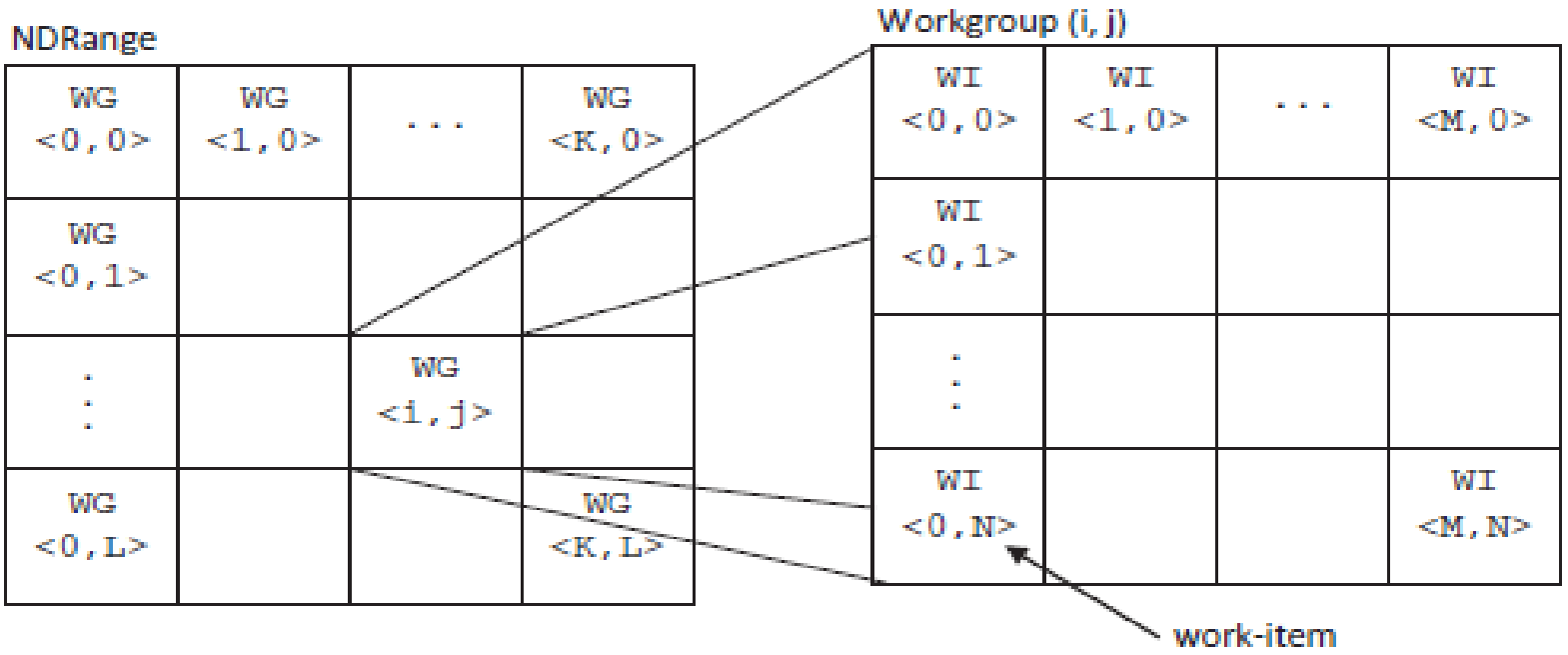


FIGURE 2.1

Work-items are created as an NDRange and grouped in workgroups.

Achieving scalability comes from dividing the work-items of an NDRange into smaller, equally sized workgroups.

Work items within a workgroup have a special relationship with one another.

Workgroup sizes are fixed.

If the total number of work-items per array is 1024, this results in creating 16 workgroups. [1024 work-items/(64 work-items per workgroup) = 16 workgroups.

Note that OpenCL requires that the index space sizes are evenly divisible by the workgroup sizes in each dimension.

Work-items behave independently, OpenCL allows the local workgroup size to be ignored by the programmer and generated automatically by the implementation; in this case, the developer will pass NULL instead.

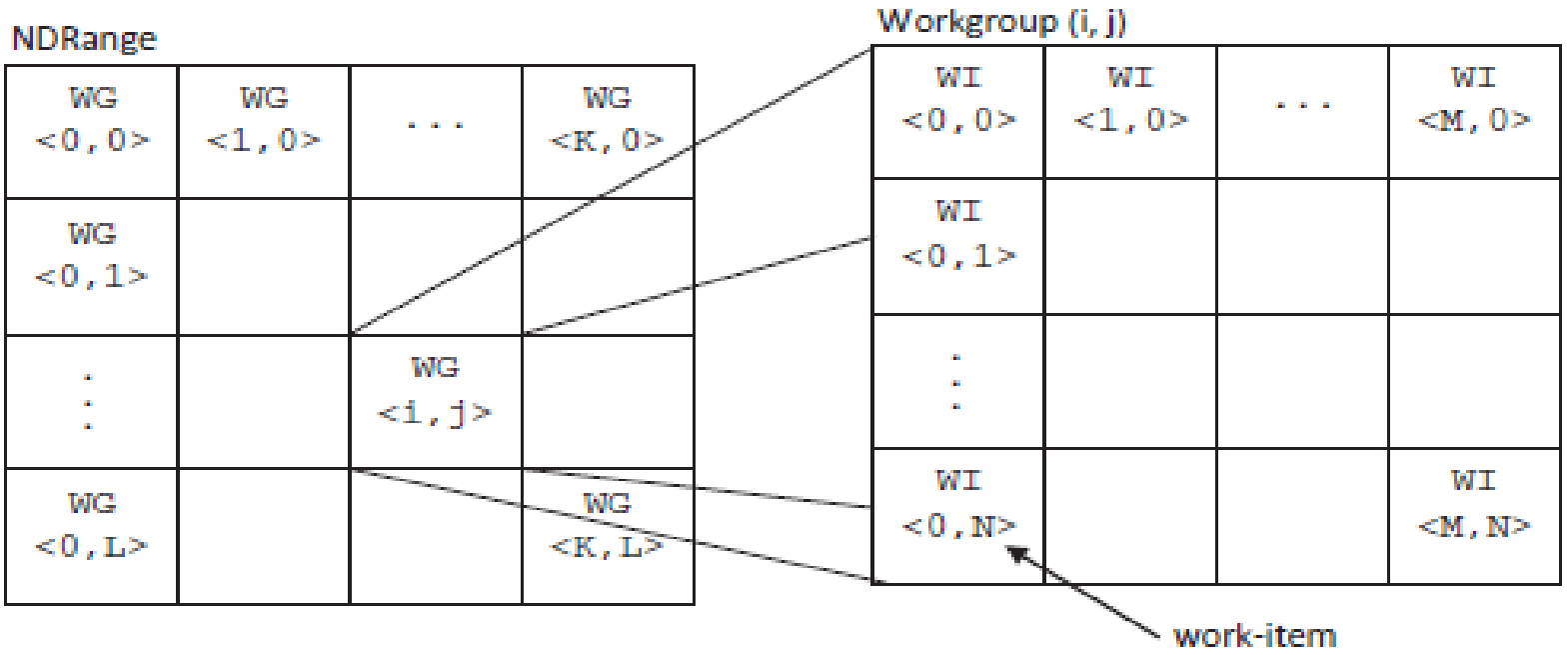


FIGURE 2.1

Work-items are created as an NDRange and grouped in workgroups.

In any Opencl Program

OpenCL Important Steps – Implementation

Step 1 : Discover and initialize the platforms
Step 2 : Discover and initialize the devices
Step 3 : Create context
Step 4 : Create a command queue
Step 5 : Create device buffers
Step 6 : Write host data device buffers
Step 7 : Create and compile the program
Step 8 : Create the kernel
Step 9 : Set the kernel arguments
Step 10 : Configure the work -items structure
Step 11 : Enqueue the kernel for execution
Step 12 : Read the output buffer back to the host
Step 13 : Release OpenCL resources

Now demo any openCL pgm

PLATFORM AND DEVICES

The OpenCL platform model defines the roles of the host and devices & provides an abstract hardware model.

Host–Device Interaction

A single host.

Platforms can be thought of as vendor-specific .

The devices that a platform can target are limited to those with which a vendor knows how to interact.

Vendors map this abstract architecture to the physical hardware.

Compute units are further divided into processing elements.

The platform device model corresponds to the hardware model of GPUs.

For example, the AMD Radeon 6970 graphics card.
Because each SIMD lane on the 6970 executes a four-way very long instruction word (VLIW) instruction, this OpenCL device allows us to schedule execution of up to 1536 instructions at a time. (i.e. 4 x 16 x 24=1536).

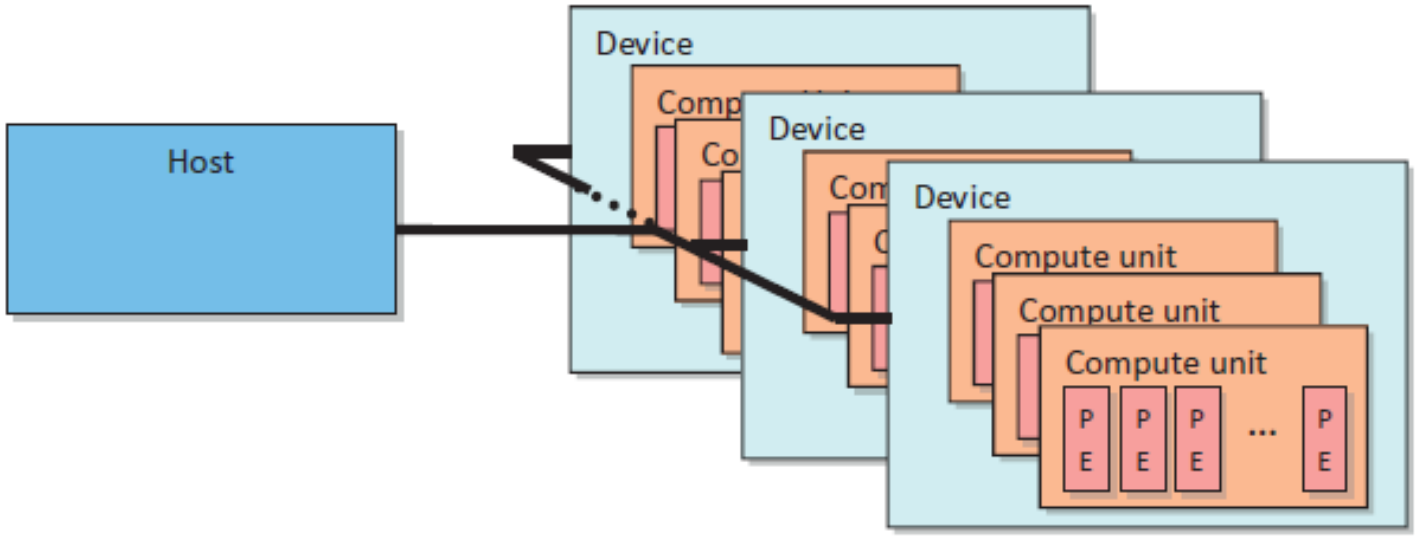
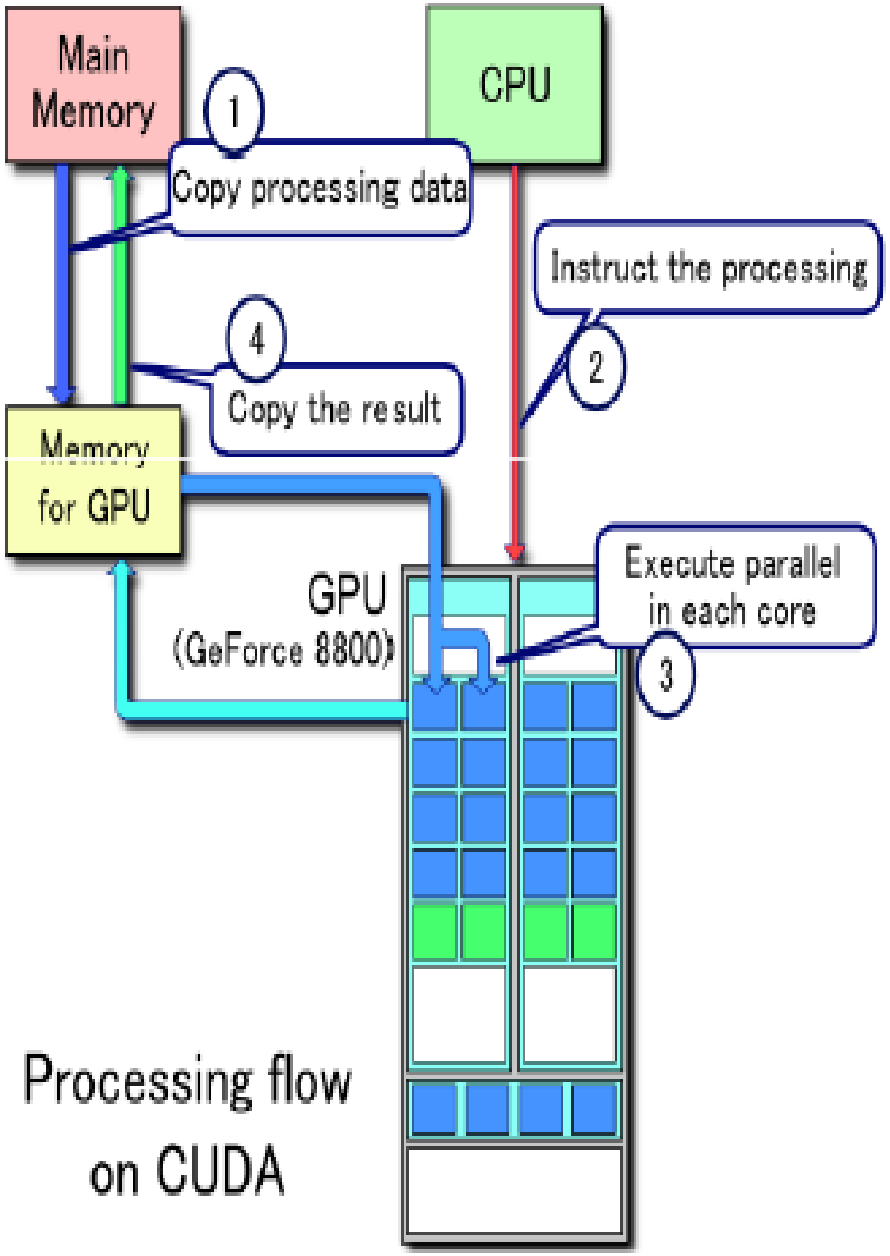


FIGURE 2.2

The platform model defines an abstract architecture for devices.

- Copy data from main memory to GPU memory
- CPU instructs the process to GPU
- GPU executes the compute in parallel on each core
- Copy the result from GPU memory to main memory



OpenCL Important Steps – Implementation

Step 1 : Discover and initialize the platforms

clGetPlatformIDs() is used to discover the set of available platforms.

General format:

```
cl_int
        clGetPlatformIDs(cl_uint num_entries,  cl_platform_id *platforms,
                                                cl_uint *num_platforms)
```

```
cl_int ret = clGetPlatformIDs(1, &platform_id, &ret_num_platforms);
```

OpenCL Important Steps – Implementation

Step 1 : Discover and initialize the platforms
Step 2 : Discover and initialize the devices

The clGetDeviceIDs() call works very similar to clGetPlatformIDs().

```
ret = clGetDeviceIDs( platform_id, CL_DEVICE_TYPE_ALL, 1,&device_id,
                                                    &ret_num_devices);
```

The device_type argument can be used to limit the devices to GPUs only (CL_DEVICE_TYPE_GPU), CPUs only (CL_DEVICE_TYPE_CPU), all devices (CL_DEVICE_TYPE_ALL).

As with platforms, clGetDeviceInfo() is called to retrieve such as name, type, and vendor from each device.

THE EXECUTION ENVIRONMENT

A context must be configured on the host.

Context

In OpenCL, a context is an abstract container that exists on the host.

A context coordinates the mechanisms for host–device interaction.

Context enables host to pass commands and data to the device.

The API function to create a context is clCreateContext().

```
cl_context context = clCreateContext( NULL, 1, &device_id, NULL, NULL, &ret);
```

After creating a context, the function clGetContextInfo() can be used to query information such as the number of devices present and the device structures.

In OpenCL, the process of discovering platforms and devices and setting up a context is tedious. However, after the code to perform these steps is written once, it can be reused for almost any project.

OpenCL Important Steps – Implementation

Step 1 : Discover and initialize the platforms
Step 2 : Discover and initialize the devices
Step 3 : Create context
Step 4 : Create a command queue

```
source_str = (char*)malloc(MAX_SOURCE_SIZE);
source_size = fread( source_str, 1, MAX_SOURCE_SIZE, fp);
fclose( fp );

// Get platform and device information
cl_platform_id platform_id = NULL;
cl_device_id device_id = NULL;
cl_uint ret_num_devices;
cl_uint ret_num_platforms;

cl_int ret = clGetPlatformIDs(1, &platform_id, &ret_num_platforms);
        // to retrieve the number of devices present
          ret = clGetDeviceIDs( platform_id, CL_DEVICE_TYPE_ALL,
                                        1,&device_id, &ret_num_devices);

// Create an OpenCL context
cl_context context = clCreateContext( NULL, 1, &device_id, NULL,
NULL, &ret);
```

```
// Create a command queue
cl_command_queue command_queue = clCreateCommandQueue
        (context, device_id, CL_QUEUE_PROFILING_ENABLE, &ret);

// Create memory buffers on the device for each vector
cl_mem a_mem_obj = clCreateBuffer(context, CL_MEM_READ_ONLY,
                                  LIST_SIZE * sizeof(int), NULL, &ret);
cl_mem b_mem_obj = clCreateBuffer(context, CL_MEM_READ_ONLY,
                                  LIST_SIZE * sizeof(int), NULL, &ret);
cl_mem c_mem_obj = clCreateBuffer(context, CL_MEM_WRITE_ONLY,
                                  LIST_SIZE * sizeof(int), NULL, &ret);

// Copy the lists A and B to their respective memory buffers
ret = clEnqueueWriteBuffer(command_queue, a_mem_obj, CL_TRUE, 0,LIST_SIZE * sizeof(int), A, 0, NULL, NULL);
ret = clEnqueueWriteBuffer(command_queue, b_mem_obj, CL_TRUE, 0,LIST_SIZE * sizeof(int), B, 0, NULL, NULL);

// Create a program from the kernel source
cl_program program = clCreateProgramWithSource(context, 1,(const char**)&source_str, (const size_t *)&source_size, &ret);
```

Command Queues: Communication with a device occurs by submitting commands to a command queue.

The command queue is the mechanism that the host uses to request action by the device.

Once the host decides which devices to work with and a context is created, one command queue needs to be created per device.

Whenever the host needs an action to be performed by a device, it will submit commands to the proper command queue.

The API clCreateCommandQueue() is used to create a command queue and associate it with a device:

```
cl_command_queue
clCreateCommandQueue(cl_context context,
                     cl_device_id device,
                     cl_command_queue_properties properties,
                     cl_int* errcode_ret)
```

```
cl_command_queue command_queue =
clCreateCommandQueue (context, device_id,
CL_QUEUE_PROFILING_ENABLE, &ret);
```

The properties parameter of clCreateCommandQueue() is a bit field that is used to enable profiling of commands (CL_QUEUE_PROFILING_ENABLE) and/or to allow out-of-order execution of commands (CL_QUEUE_OUT_OF_ORDER_EXEC_ MODE_ ENABLE).

With an in-order command queue (the default), commands are pulled from the queue in the order they were received.

Out-of-order queues allow the OpenCL implementation to search for commands that can possibly be rearranged to execute more efficiently.

If out-of-order queues are used, it is up to the user to specify dependencies that enforce a correct execution order.

```
cl_command_queue command_queue =
clCreateCommandQueue (context, device_id,
CL_QUEUE_PROFILING_ENABLE, &ret);
```

OpenCL Important Steps – Implementation

Step 1 : Discover and initialize the platforms
Step 2 : Discover and initialize the devices
Step 3 : Create context
Step 4 : Create a command queue
Step 5 : Create device buffers

Memory Objects

OpenCL applications work with large arrays or multidimensional matrices.

This data needs to be physically present on a device before execution can begin.

In order for data to be transferred to a device, it must first be encapsulated as a memory object.

OpenCL defines two types of memory objects:

i) buffers

ii) images.

Buffers are equivalent to arrays in C, created using malloc(), where data elements are stored contiguously in memory. Images, on the other hand, are designed as opaque objects, allowing for data padding and other optimizations that may improve performance on devices.

```
// Create a command queue
cl_command_queue command_queue = clCreateCommandQueue
                (context, device_id, CL_QUEUE_PROFILING_ENABLE, &ret);

// Create memory buffers on the device for each vector
cl_mem a_mem_obj = clCreateBuffer(context, CL_MEM_READ_ONLY,
                                  LIST_SIZE * sizeof(int), NULL, &ret);
cl_mem b_mem_obj = clCreateBuffer(context, CL_MEM_READ_ONLY,
                                  LIST_SIZE * sizeof(int), NULL, &ret);
cl_mem c_mem_obj = clCreateBuffer(context, CL_MEM_WRITE_ONLY,
                                  LIST_SIZE * sizeof(int), NULL, &ret);

// Copy the lists A and B to their respective memory buffers
ret = clEnqueueWriteBuffer(command_queue, a_mem_obj, CL_TRUE, 0,LIST_SIZE * sizeof(int), A, 0, NULL, NULL);
ret = clEnqueueWriteBuffer(command_queue, b_mem_obj, CL_TRUE, 0,LIST_SIZE * sizeof(int), B, 0, NULL, NULL);

// Create a program from the kernel source
cl_program program = clCreateProgramWithSource(context, 1,(const char**)&source_str, (const size_t *)&source_size, &ret);
```

Creating a buffer requires supplying the size of the buffer and a context in which the buffer will be allocated.
It is visible for all devices associated with the context.
Optionally, the caller can supply flags that specify that the data is read-only, write-only, or read-write.

```
cl_mem a_mem_obj = clCreateBuffer(context, CL_MEM_READ_ONLY, SIZE * sizeof(char), NULL, &ret);
```

OpenCL Important Steps – Implementation

Step 1 : Discover and initialize the platforms
Step 2 : Discover and initialize the devices
Step 3 : Create context
Step 4 : Create a command queue
Step 5 : Create device buffers
Step 6 : Write host data to device buffers

Data contained in host memory is transferred to and from an OpenCL buffer using the commands clEnqueueWriteBuffer() and clEnqueueReadBuffer(), respectively.
If a kernel that is dependent on such a buffer is executed on a GPU, the buffer may be transferred to the device. The buffer is linked to a context, not a device.

```
// Create a command queue
cl_command_queue command_queue = clCreateCommandQueue
                (context, device_id, CL_QUEUE_PROFILING_ENABLE, &ret);

// Create memory buffers on the device for each vector
cl_mem a_mem_obj = clCreateBuffer(context, CL_MEM_READ_ONLY,
                                  LIST_SIZE * sizeof(int), NULL, &ret);
cl_mem b_mem_obj = clCreateBuffer(context, CL_MEM_READ_ONLY,
                                  LIST_SIZE * sizeof(int), NULL, &ret);
cl_mem c_mem_obj = clCreateBuffer(context, CL_MEM_WRITE_ONLY,
                                  LIST_SIZE * sizeof(int), NULL, &ret);

// Copy the lists A and B to their respective memory buffers
ret = clEnqueueWriteBuffer(command_queue, a_mem_obj, CL_TRUE, 0,LIST_SIZE * sizeof(int), A, 0, NULL, NULL);
ret = clEnqueueWriteBuffer(command_queue, b_mem_obj, CL_TRUE, 0,LIST_SIZE * sizeof(int), B, 0, NULL, NULL);

// Create a program from the kernel source
cl_program program = clCreateProgramWithSource(context, 1,(const char**)&source_str, (const size_t *)&source_size, &ret);
```

Reading or writing a buffer requires a command queue to manage the execution schedule.

The enqueue function requires the buffer, the number of bytes to transfer, and an offset within the buffer.

The blocking_write option should be set to CL_TRUE if the transfer into an OpenCL buffer should complete before the function returns i.e., it will block until the operation has completed.

Setting blocking_write to CL_FALSE allows clEnqueueWriteBuffer() to return before the write operation has completed.

```
ret = clEnqueueWriteBuffer(command_queue, a_mem_obj, CL_TRUE, 0,LIST_SIZE * sizeof(int), A, 0, NULL, NULL);
```

OpenCL Important Steps – Implementation

Step 1 : Discover and initialize the platforms
Step 2 : Discover and initialize the devices
Step 3 : Create context
Step 4 : Create a command queue
Step 5 : Create device buffers
Step 6 : Write host data to device buffers
Step 7 : Create and compile the program

Creating an OpenCL Program Object

OpenCL C code is called a program.

```
// Create a command queue
cl_command_queue command_queue = clCreateCommandQueue
                (context, device_id, CL_QUEUE_PROFILING_ENABLE, &ret);

// Create memory buffers on the device for each vector
cl_mem a_mem_obj = clCreateBuffer(context, CL_MEM_READ_ONLY,
                                  LIST_SIZE * sizeof(int), NULL, &ret);
cl_mem b_mem_obj = clCreateBuffer(context, CL_MEM_READ_ONLY,
                                  LIST_SIZE * sizeof(int), NULL, &ret);
cl_mem c_mem_obj = clCreateBuffer(context, CL_MEM_WRITE_ONLY,
                                  LIST_SIZE * sizeof(int), NULL, &ret);

// Copy the lists A and B to their respective memory buffers
ret = clEnqueueWriteBuffer(command_queue, a_mem_obj, CL_TRUE, 0,LIST_SIZE * sizeof(int), A, 0, NULL, NULL);
ret = clEnqueueWriteBuffer(command_queue, b_mem_obj, CL_TRUE, 0,LIST_SIZE * sizeof(int), B, 0, NULL, NULL);

// Create a program from the kernel source
cl_program program = clCreateProgramWithSource(context, 1,(const char**)&source_str, (const size_t *)&source_size, &ret);
```

Creating an OpenCL Program Object (Contd..)

A program is a collection of functions called kernels, where kernels are units of execution that can be scheduled to run on a device.

OpenCL programs are compiled at runtime through a series of API calls.

This runtime compilation gives the system an opportunity to optimize for a specific device.

There is no need for an OpenCL application to have been prebuilt against the AMD, NVIDIA.

```
// Build the program
    ret = clBuildProgram(program, 1, &device_id, NULL, NULL, NULL);

    // Create the OpenCL kernel
cl_kernel kernel = clCreateKernel(program, "vector_add", &ret);
    // Set the arguments of the kernel
    ret = clSetKernelArg(kernel, 0, sizeof(cl_mem), (void *)&a_mem_obj);
    ret = clSetKernelArg(kernel, 1, sizeof(cl_mem), (void *)&b_mem_obj);
    ret = clSetKernelArg(kernel, 2, sizeof(cl_mem), (void *)&c_mem_obj);

   // Execute the OpenCL kernel on the list
    size_t global_item_size = SIZE; // Process the entire lists
    size_t local_item_size = 64; // Process in groups of 64

            cl_event event;
            ret = clEnqueueNDRangeKernel(command_queue, kernel, 1,
&global_item_size, &local_item_size, NULL, 0, NULL, &event);
```

OpenCL Important Steps – Implementation

Step 1 : Discover and initialize the platforms
Step 2 : Discover and initialize the devices
Step 3 : Create context
Step 4 : Create a command queue
Step 5 : Create device buffers
Step 6 : Write host data device buffers
Step 7 : Create and compile the program
Step 8 : Create the kernel

```
// Build the program
    ret = clBuildProgram(program, 1, &device_id, NULL, NULL, NULL);

    // Create the OpenCL kernel
cl_kernel kernel = clCreateKernel(program, "vector_add", &ret);
    // Set the arguments of the kernel
    ret = clSetKernelArg(kernel, 0, sizeof(cl_mem), (void *)&a_mem_obj);
    ret = clSetKernelArg(kernel, 1, sizeof(cl_mem), (void *)&b_mem_obj);
    ret = clSetKernelArg(kernel, 2, sizeof(cl_mem), (void *)&c_mem_obj);

   // Execute the OpenCL kernel on the list
    size_t global_item_size = SIZE; // Process the entire lists
    size_t local_item_size = 64; // Process in groups of 64

     cl_event event;
          ret = clEnqueueNDRangeKernel(command_queue, kernel, 1,
&global_item_size, &local_item_size, NULL, 0, NULL, &event);
```

The OpenCL Kernel

The final stage to obtain a cl_kernel object that can be used to execute kernels on a device is to extract the kernel from the cl_program.

Extracting a kernel from a program is similar to obtaining an exported function from a dynamic library.

The name of the kernel that the program exports is used to request it from the compiled program object.

The name of the kernel is passed to clCreateKernel(), along with the program object, and the kernel object will be returned if the program object was valid and the particular kernel is found.

```
cl_kernel kernel = clCreateKernel(program, "vector_add", &ret);
```

```
cl_kernel  clCreateKernel(
cl_program program,
 const char* kernel_name ,
cl_int * errcode_ret);
```

OpenCL Important Steps – Implementation

Step 1 : Discover and initialize the platforms
Step 2 : Discover and initialize the devices
Step 3 : Create context
Step 4 : Create a command queue
Step 5 : Create device buffers
Step 6 : Write host data device buffers
Step 7 : Create and compile the program
Step 8 : Create the kernel
Step 9 : Set the kernel arguments

```
// Build the program
    ret = clBuildProgram(program, 1, &device_id, NULL, NULL, NULL);

    // Create the OpenCL kernel
cl_kernel kernel = clCreateKernel(program, "vector_add", &ret);
    // Set the arguments of the kernel
    ret = clSetKernelArg(kernel, 0, sizeof(cl_mem), (void *)&a_mem_obj);
    ret = clSetKernelArg(kernel, 1, sizeof(cl_mem), (void *)&b_mem_obj);
    ret = clSetKernelArg(kernel, 2, sizeof(cl_mem), (void *)&c_mem_obj);

   // Execute the OpenCL kernel on the list
    size_t global_item_size = SIZE; // Process the entire lists
    size_t local_item_size = 64; // Process in groups of 64

            cl_event event;
            ret = clEnqueueNDRangeKernel(command_queue, kernel, 1,
&global_item_size, &local_item_size, NULL, 0, NULL, &event);
```

Some more steps needed before kernel is executed.

Unlike calling functions in regular C programs, we cannot simply call a kernel by providing a list of arguments.

Executing a kernel requires dispatching it through an enqueue function.

Due both to the syntax of the C language and to the fact that kernel arguments are persistent (and hence we need not repeatedly set them to construct the argument list for such a dispatch), we must specify each kernel argument individually using the function clSetKernelArg().

```
cl_int
clSetKernelArg(
cl_kernel kernel,
cl_uint arg_index ,
size_t  arg_size,
const void * arg_value);
```

```
ret = clSetKernelArg(kernel, 0, sizeof(cl_mem), (void *)&a_mem_obj);
```

- *Arg_index-Arguments to the kernel are referred by indices that go from 0 for the leftmost argument to n - 1, where n is the total number of arguments declared by a kernel.*

This function takes a kernel object, an index specifying the argument number, the size of the argument, and a pointer to the argument. When a kernel is executed, this information is used to transfer arguments to the device.

Now the kernel is ready to be executed. Requesting that a device begin executing a kernel is done with a call to clEnqueueNDRangeKernel():

OpenCL Important Steps – Implementation

Step 1 : Discover and initialize the platforms
Step 2 : Discover and initialize the devices
Step 3 : Create context
Step 4 : Create a command queue
Step 5 : Create device buffers
Step 6 : Write host data device buffers
Step 7 : Create and compile the program
Step 8 : Create the kernel
Step 9 : Set the kernel arguments

Step 11 : Enqueue the kernel for execution

```
// Build the program
    ret = clBuildProgram(program, 1, &device_id, NULL, NULL, NULL);

    // Create the OpenCL kernel
cl_kernel kernel = clCreateKernel(program, "vector_add", &ret);
    // Set the arguments of the kernel
    ret = clSetKernelArg(kernel, 0, sizeof(cl_mem), (void *)&a_mem_obj);
    ret = clSetKernelArg(kernel, 1, sizeof(cl_mem), (void *)&b_mem_obj);
    ret = clSetKernelArg(kernel, 2, sizeof(cl_mem), (void *)&c_mem_obj);

   // Execute the OpenCL kernel on the list
    size_t global_item_size = SIZE; // Process the entire lists
    size_t local_item_size = 64; // Process in groups of 64

            cl_event event;
            ret = clEnqueueNDRangeKernel(command_queue, kernel, 1,
&global_item_size, &local_item_size, NULL, 0, NULL, &event);
```

OpenCL Important Steps – Implementation

Step 1 : Discover and initialize the platforms
Step 2 : Discover and initialize the devices
Step 3 : Create context
Step 4 : Create a command queue
Step 5 : Create device buffers
Step 6 : Write host data device buffers
Step 7 : Create and compile the program
Step 8 : Create the kernel
Step 9 : Set the kernel arguments

Step 11 : Enqueue the kernel for execution
Step 12 : Read the output buffer back to the host

```
        ret = clFinish(command_queue);
        cl_ulong time_start, time_end;
        double total_time;

 clGetEventProfilingInfo (event, CL_PROFILING_COMMAND_START,
                                        sizeof(time_start), &time_start, NULL);

  clGetEventProfilingInfo(event, CL_PROFILING_COMMAND_END,
                                        sizeof(time_end), &time_end, NULL);
        total_time = double (time_end - time_start);

// Read the memory buffer C on the device to the local variable C
        ret = clEnqueueReadBuffer(command_queue, c_mem_obj,
CL_TRUE, 0,LIST_SIZE * sizeof(int), C, 0, NULL, NULL);

   // Display the result to the screen
    for(i = 0; i < LIST_SIZE; i++)
       printf("%d + %d = %d\n", A[i], B[i], C[i]);
```

OpenCL Important Steps – Implementation

Step 1 : Discover and initialize the platforms
Step 2 : Discover and initialize the devices
Step 3 : Create context
Step 4 : Create a command queue
Step 5 : Create device buffers
Step 6 : Write host data device buffers
Step 7 : Create and compile the program
Step 8 : Create the kernel
Step 9 : Set the kernel arguments

Step 11 : Enqueue the kernel for execution
Step 12 : Read the output buffer back to the host
Step 13 : Release OpenCL resources

```
// Clean up
    ret = clReleaseKernel(kernel);
    ret = clReleaseProgram(program);
    ret = clReleaseMemObject(a_mem_obj);
    ret = clReleaseMemObject(b_mem_obj);
    ret = clReleaseMemObject(c_mem_obj);
    ret = clReleaseCommandQueue(command_queue);
    ret = clReleaseContext(context);
```

Flush and Finish

The flush and finish commands are two different types of barrier operations for a command queue.

The clFinish() function blocks until all of the commands in a command queue have completed; its functionality is synonymous with a synchronization barrier.

The clFlush() function blocks until all of the commands in a command queue have been removed from the queue.

```
cl_int clFinish(cl_command_queue command_queue);

cl_int clFlush(cl_command_queue command_queue);
```

```
// Clean up
    ret = clFlush(command_queue);
    ret = clReleaseKernel(kernel);
    ret = clReleaseProgram(program);
    ret = clReleaseMemObject(a_mem_obj);
    ret = clReleaseMemObject(b_mem_obj);
    ret = clReleaseMemObject(c_mem_obj);
    ret = clReleaseCommandQueue(command_queue);
    ret = clReleaseContext(context);

    free(A);
    free(B);
    free(C);
```

OpenCL Important Steps – Implementation

Step 1 : Discover and initialize the platforms
Step 2 : Discover and initialize the devices
Step 3 : Create context
Step 4 : Create a command queue
Step 5 : Create device buffers
Step 6 : Write host data device buffers
Step 7 : Create and compile the program
Step 8 : Create the kernel
Step 9 : Set the kernel arguments
Step 10 : Configure the work -items structure
Step 11 : Enqueue the kernel for execution
Step 12 : Read the output buffer back to the host
Step 13 : Release OpenCL resources

```
// Build the program
    ret = clBuildProgram(program, 1, &device_id, NULL, NULL, NULL);

    // Create the OpenCL kernel
cl_kernel kernel = clCreateKernel(program, "vector_add", &ret);
    // Set the arguments of the kernel
    ret = clSetKernelArg(kernel, 0, sizeof(cl_mem), (void *)&a_mem_obj);
    ret = clSetKernelArg(kernel, 1, sizeof(cl_mem), (void *)&b_mem_obj);
    ret = clSetKernelArg(kernel, 2, sizeof(cl_mem), (void *)&c_mem_obj);

   // Execute the OpenCL kernel on the list
    size_t global_item_size = SIZE; // Process the entire lists
    size_t local_item_size = 64; // Process in groups of 64

// Read the memory buffer C on the device to the local variable C
            cl_event event;
            ret = clEnqueueNDRangeKernel(command_queue, kernel, 1,
&global_item_size, &local_item_size, NULL, 0, NULL, &event);
```

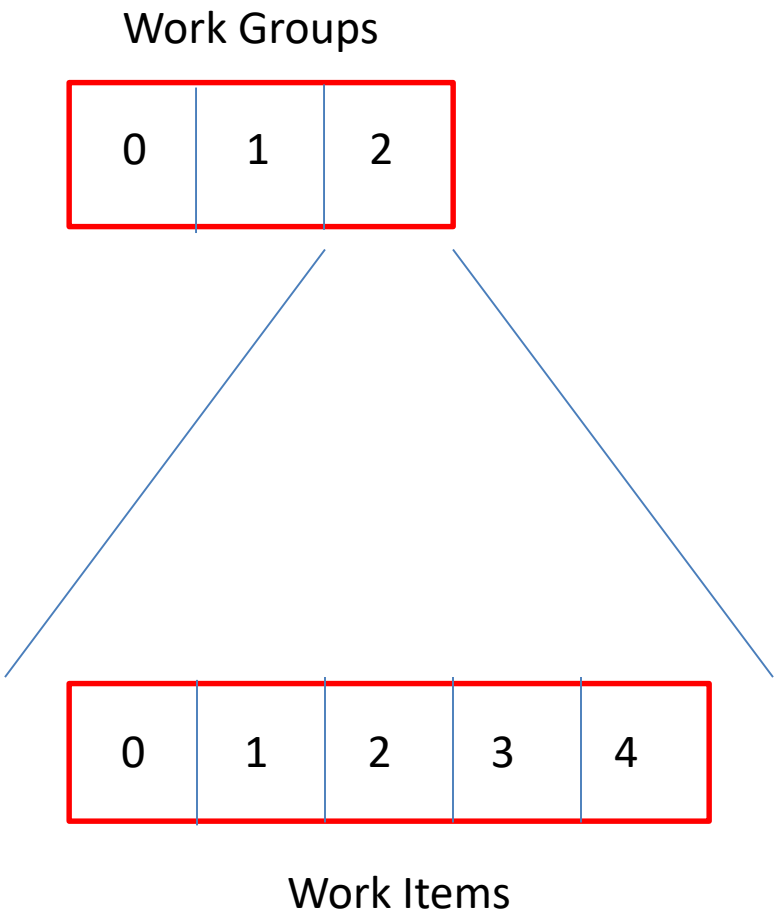
```
size_t global_item_size = SIZE; // Process the entire lists

size_t local_item_size = 5; // Process in groups
```

If the total number of work-items per array is SIZE, this results in creating SIZE/5 workgroups.

(SIZE work-items/(5 work-items per workgroup) = X no. of workgroups).

SIZE =15

Total WIs is 15



MEMORY MODEL

To support code portability, OpenCL's approach is to define an abstract memory model that programmers can target when writing code. The memory spaces defined by OpenCL are discussed here:

These memory spaces are relevant within OpenCL programs.

The keywords associated with each space can be used to specify where a variable should be created or where the data to reside.

Global memory is visible to all compute units on the device (similar to the main memory on a CPU-based host system).

Whenever data is transferred from the host to the device, the data will reside in global memory.

Any data that is to be transferred back from the device to the host must also reside in global memory.

The keyword _ _ global is added to a pointer declaration to specify that data referenced by the pointer resides in global memory.

For example, in the OpenCL C code _ _ global float* A, the data pointed to by A resides in global memory.

MEMORY MODEL

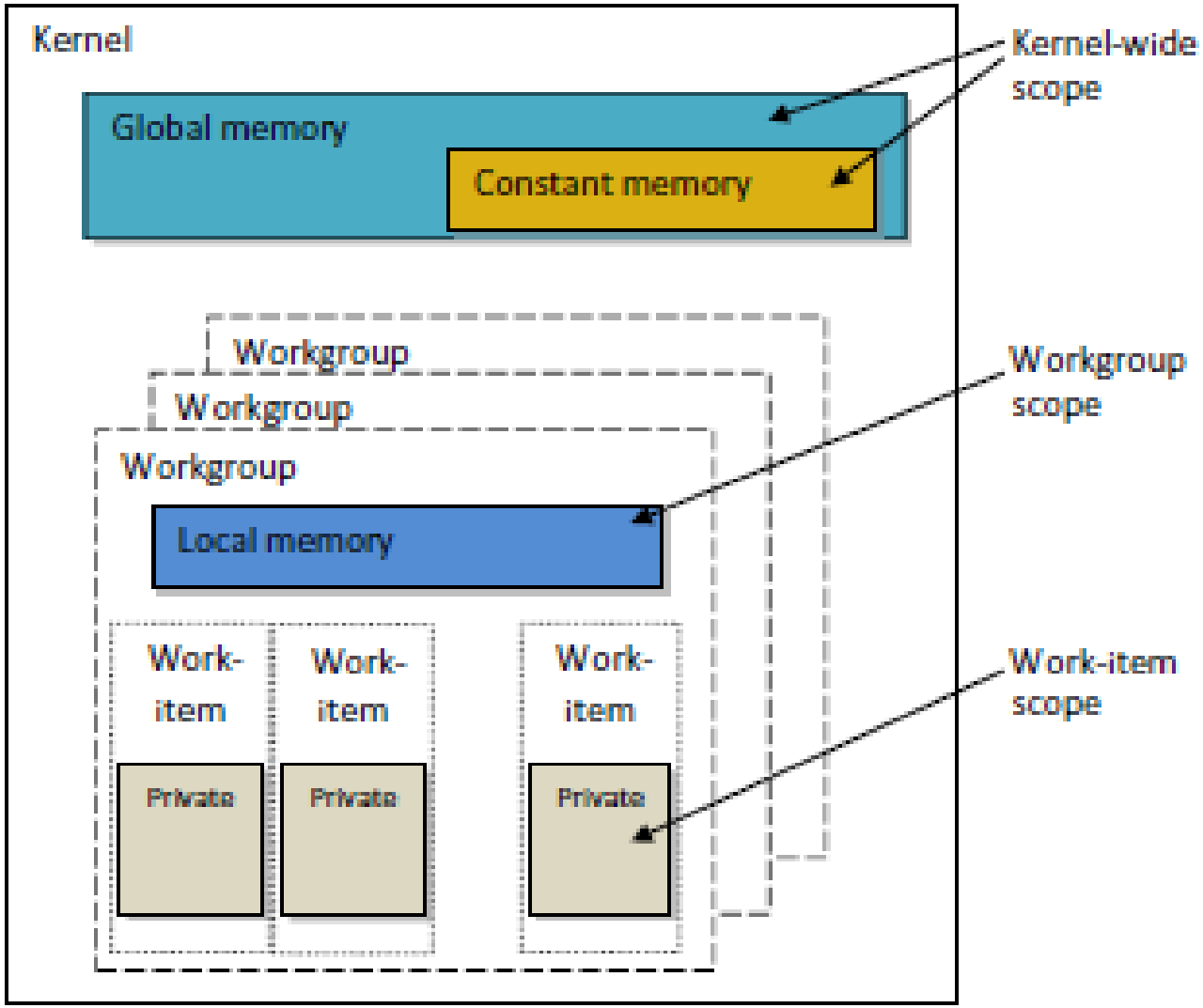


FIGURE 2.3

The abstract memory model defined by OpenCL.

Constant memory is for data where each element is accessed simultaneously by all work-items.

Variables whose values never change also fall into this category.

Constant memory is modeled as a part of global memory, so memory objects that are transferred to global memory can be specified as constant.

Data is mapped to constant memory by using the _ _constant keyword.

Local memory is a scratch pad memory whose address space is unique to each compute device. It is common for it to be implemented as on-chip memory.

Local memory is modeled as being shared by a workgroup.

Accesses may have much shorter latency and much higher bandwidth than global memory.

Calling clSetKernelArg() with a size, but no argument, allows local memory to be allocated at runtime, where a kernel parameter is defined as a _ _local pointer (e.g., _ _local float* sharedData).

Arrays can be statically declared in local memory by appending the keyword _ _local (e.g., _ _local float[64] sharedData).

Private memory is memory that is unique to an individual work-item.

Local variables and non-pointer kernel arguments are private by default.

The relationship between OpenCL memory spaces and those found on a GPU.

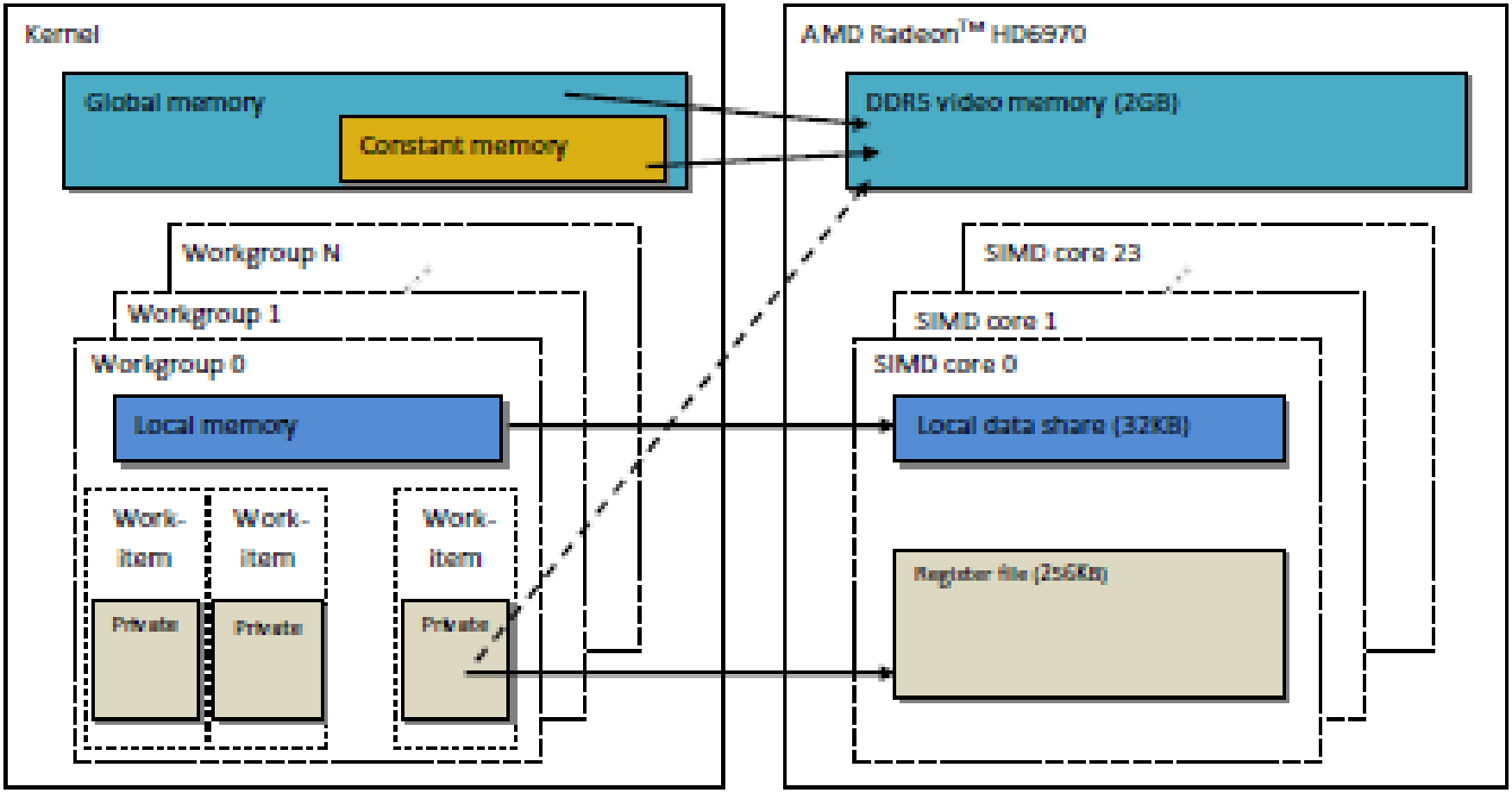


FIGURE 2.4

Mapping from the memory model defined by OpenCL to the architecture of an AMD Radeon 6970 GPU. Simple private memory will be stored in registers; complex addressing or excessive use will be stored in DRAM.

WRITING KERNELS

As previously described, OpenCL C kernels are similar to C functions and can be thought of as instances of a parallel map operation.

The function body will be executed once for every work-item created.

We utilize the code for the OpenCL kernel to illustrate how this mapping is accomplished.

Kernels begin with the keyword _ _kernel and must have a return type of void.

The argument list is as for a C function with the additional requirement that the address space of any pointer must be specified.

Buffers can be declared in global memory (__global) or constant memory (_ _constant).

Access qualifiers (__read_only, __write_only, and __read_write) can also be optionally specified because they may allow for compiler and hardware optimizations.

The __local qualifier is used to declare memory that is shared between all workitems in a workgroup.

When a local pointer is declared as a kernel parameter, such as __local float *sharedData, it is a pointer to an array shared by the entire workgroup.

Questions:

1. Study of OpenCL APIs.
2. Write an OpenCL program to initialise the GPU and test the initialization.
3. Write an OpenCL program to copy a string in the kernel.(using the kernel as a simple function)

   Hello program OR Hello print
4. Write an OpenCL program to convert an uppercase string to lowercase parallelly.
5. Write an OpenCL program for vector-vector addition.

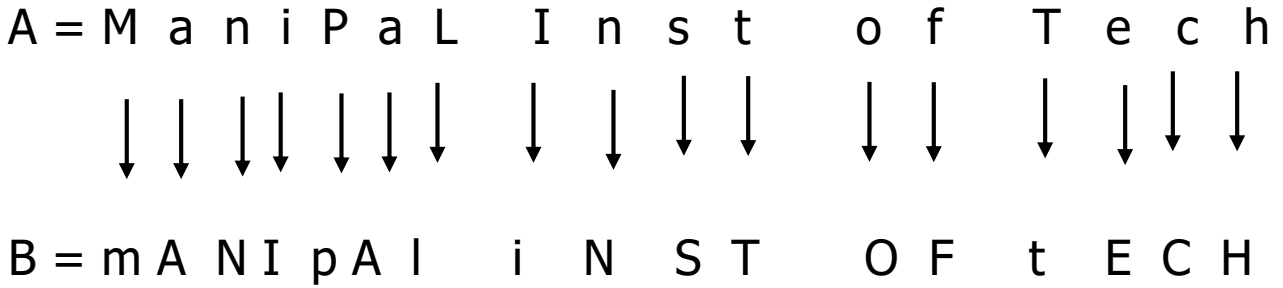
6. Write an OpenCL program to find the square of each element in a vector.
7. Write an OpenCL program to find the square of each element of an array and add the respective elements of the original array.
8. Write an OpenCL program to convert each element of an array in to its equivalent binary value.
9. Write an OpenCL program to copy a string n times parallelly.
10. Write an OpenCL program for matrix-matrix multiplication.

11. Write an OpenCL program to read n words of a string and reverse each word of it parallelly.
12. Write an OpenCL program to read a string and reverse it parallelly.
13. Write an OpenCL program for scalar-matrix multiplication.
14. Write an OpenCL program to sort N elements of a given array using selection sort method. Selection sort
15. Write an OpenCL program to read a string and sort it.

16. Write an OpenCL program to read *n* words of a string and sort the words parallelly.
17. Write an OpenCL program to calculate matrix-vector multiplication.
18. Write an OpenCL program to sum the row and column elements of a matrix.
19. Write an OpenCL program to compute the Transpose of given matrix.
20. Write an OpenCL program to calculate π-value.

OpenCL programs on Strings

OpenCL pgm which reads a string as input and it toggles every character of this string in parallel. Store the resultant string in another character array.



```
_ _kernel void str_chgcase (_ _global char *A, _ _global char
*B)
{

   int i = get_global_id(0);
  if(A[i] >='A' && A[i] <='Z')
     B[i] = A[i] + 32;
    else
      B[i] = A[i] - 32;
}


size_t global_item_size =?;
 size_t local_item_size = ?;
```

Work items and Work groups for Two Dimensional arrays:

For a two Dimensional array, global and local work size can be set as follows:

size_t global_item_size[2]={64,32};

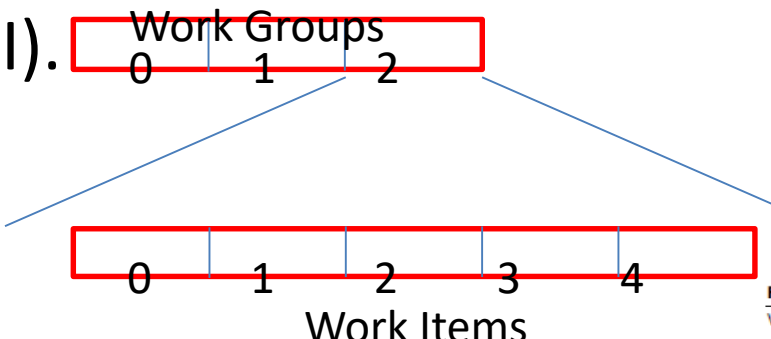
size_t local_item_size[2]={64,1};

For One Dimensional:

size_t global_item_size = 15; // Process the entire lists

size_t local_item_size = 5; // Process in groups

- A global size of 64 WI in dimension 0 and 32 WI in dimension 1, for a total of 64 * 32 = 2,048 total WI.
- It also specifies the local size to be 64 WI/WG in dimension 0 and 1 WI/WG in dimension 1.
- This results in 64/64 = 1 work-group in dimension 0 and 32/1 workgroups in dimension 1 for a total of 2,048 work-items (WI).



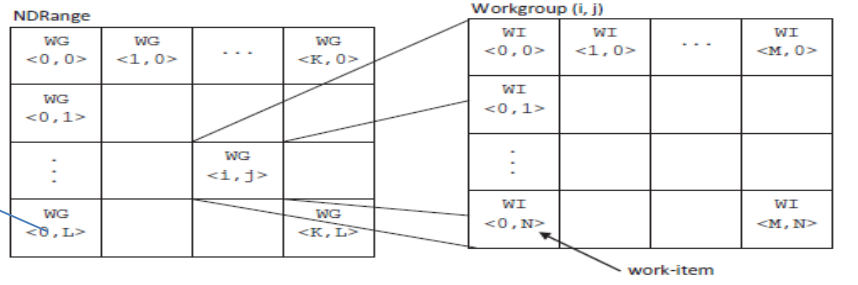


FIGURE 2.1

Work-items are created as an NDRange and grouped in workgroups.

size_t global_work_size[2]={64,32};
size_t local_work_size[2]={64,1};
1 work-group in dimension 0 and 32 workgroups in dimension 1
local size to be 64 WI/WG in dimension 0 and 1 WI/WG in dimension 1.

0,0
0,1
0,2
0,3
0,31

Work Groups

0,0	**1,0**	**2,0**	**3,0**					**63,0**

Work Items

Work Items

0,0	1,0	2,0					63,0
0, 1							
0, 2							
0,32							

Matrix Multiplication

Matrix Multiplication in OpenCL

Let us recall a serial C implementation:

The code iterates over three nested for loops.

The two outer loops are used to iterate over…..

The innermost loop will iterate over….

```
// Iterate over the rows of Matrix A
for(int i = 0; i < hA; i++)
{
   // Iterate over the columns of Matrix B
   for(int j = 0; j < wB; j++)
     {
        C[i][j] = 0;
   // Multiply and accumulate the values
         for(int k = 0; k < wA; k++)
          {
                C[i][j] += A[i][k] * B[k][j];
           }
      }
  }
```

To impl in OpenCL,
-two outer for-loops work independently of each other. i.e. a separate work-item can be created for each output element of the matrix.
The two outer for-loops are mapped to the two-dimensional range of work-items for the kernel.

```
// widthA = heightB for valid matrix multiplication
_ _kernel void simpleMultiply(_ _global float* inputA,  _ _global float* inputB, _ _global float*
                                                                                          outputC)
   {
     wA =10;
     wB = 10;
     int globalIdx = get_global_id(0);      //Get global position in X direction
     int globalIdy = get_global_id(1);     //Get global position in Y direction

    float sum = 0.0;

    //Calculate result of one element of Matrix C
    for (int i = 0; i < wA; i++)
        {
            float tempA = inputA[globalIdy *wA + i];
             float tempB = inputB[i*wB+ globalIdx];
            sum += tempA * tempB;
        }
    outputC[globalIdy *wB + globalIdx] = sum;
   }                    Note: Matrix to be read into array
```

size_t global_work_size[2]={16,16 };
size_t local_work_size[2]={4, 4};

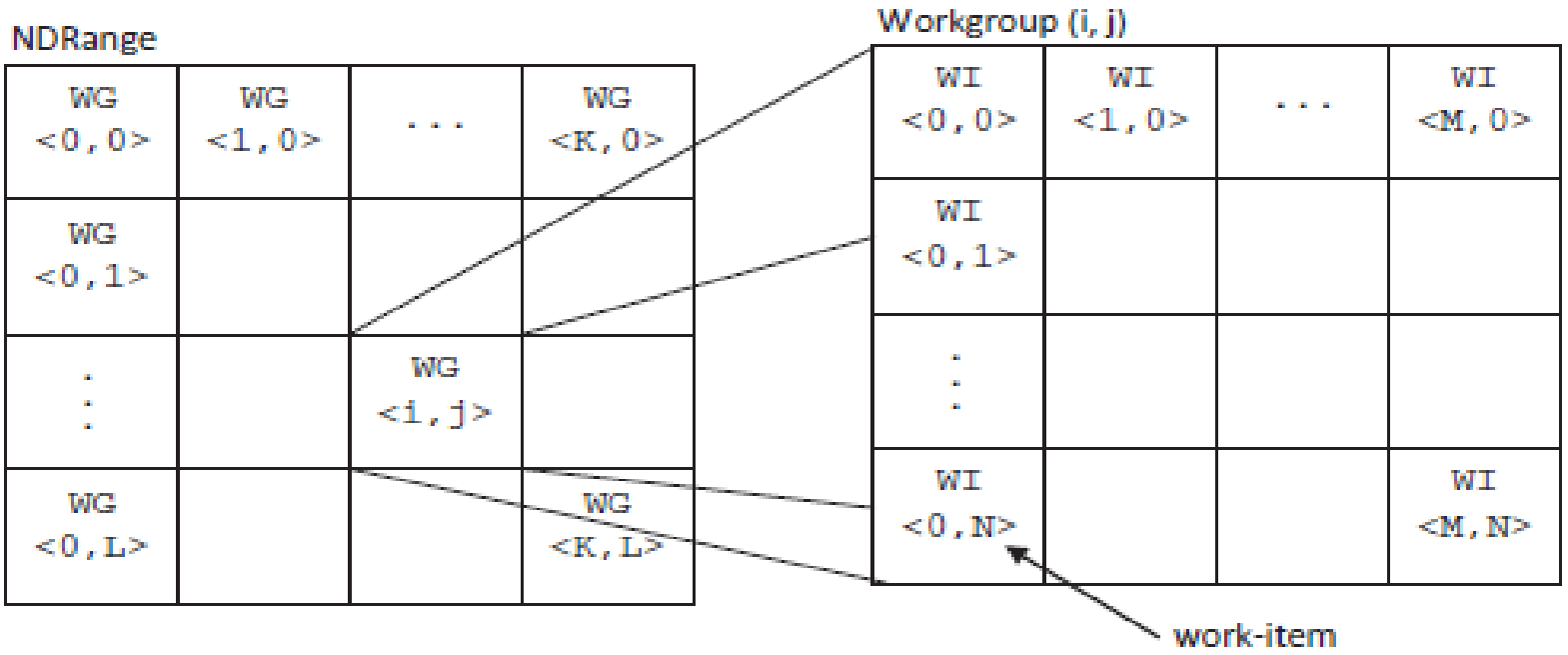


FIGURE 2.1

Work-items are created as an NDRange and grouped in workgroups.

Now ,

```
size_t global_work_size = 15;
size_t local_work_size = 5;
```

1

```
size_t global_work_size = 15;
size_t local_work_size = 1;
```

2

```
size_t global_work_size[2]={ 10, 10};
size_t local_work_size[2]={ 1, 1};
```

3

```
size_t global_work_size[2]={ 10, 10};
size_t local_work_size[2]={ 2, 2};
```

4