Random notes about OpenCL 2.0

Versions

Comment	Author
Initial Version	Konstantin Burlachenko, burlachenkok@gmail.com

1. About ARM Mali GPU

1.1 Introduction about ARM Mali GPU

Arm produces families of Mali GPUs. Bifrost and Valhall are two of the Mali GPU architectures.
Mali GPUs run data processing tasks in parallel that contain relatively little control code. Mali GPUs typically contain many more processing units then application processors. This enables Mali GPUs to compute at a higher rate than application processors without using more power.

Mali GPUs can contain many identical shader cores.
Each shader core supports **hundreds** of concurrently executing threads.

Each shader core contains:
• One to three arithmetic pipelines or Execution Engines.
• One load-store pipeline
• One texture pipeline (used for reading image data types)

In the execution engines in Mali Bifrost and Valhall GPUs, scalar instructions are executed in parallel so the GPU operates on multiple data elements simultaneously.

Due to ARM MALI GPU developer guide – engineer is not required to perform vectorization of his/her code to do this.

1.2 About ARM Mali-G76 MP10 from OpenCL compute point of view

ARM MALI GPUs consist of a small number of shader cores (GPU "ARM Mali-G76 MP10" have 10 shader cores working).

Important things:

- Each shader core has 3 execution unit. Each execution unit can run 8(or 4) threads (depend on device).
- Inter core task management supports managing workloads across the cores.
- GPU threading, in general, is hardware controlled rather than exposed to the operating system.
- An embedded design allow share the same global memory with embedded CPUs, reducing the need to copy data across memory spaces.

- GPU design is throughput oriented, and rely very heavily on thread-level parallelism to utilize their large numbers of vector processing units.

The arithmetic pipes in ARM Mali Bifrost and ARM Mali Valhall GPUs are based on quad-style vectorization. Scalar instructions are executed in parallel so the GPU operates on multiple data elements simultaneously. Mali-G71 and Mali-G72, a quad 4-wide SIMD unit, with each lane possessing separate FMA and ADD/SF pipes.

The width of a wave front at the ISA-level for these parts has also been just 4 instructions, meaning all of the threads within a wave front are issued in a single cycle. 4-wide design was a notably narrow choice relative to most other graphics architectures. By going with a narrower wave front, a group of threads is less likely to diverge. Divergences are easy enough to handle (just follow both paths), but as usual the split hurts performance. Arm doesn't officially disclose the size of a quad's register file, they have confirmed that there are 64 registers per lane for G76's register file.

1.3 Example of ARM GPUs and ARM CPUs

CPU, ISA	**CPU**
ARMv8-A	Cortex-A73
ARMv8.2-A	Cortex-A76
ARMv8-A	Cortex-A73 Cortex-A53
ARMv8-A	Cortex-A73 Cortex-A53
ARMv8.2-A	Cortex-A76 Cortex-A55
pu Arch	**GPU**
Bifrost	Mali-G51 MP4
Bifrost	Mali-G52 MP6
Bifrost	Mali-G71 MP8
Bifost	Mali-G72 MP12
Bifrost	Mali-G76 MP10

1.4 Exact number of threads in a wave-front for ARM GPUs

GPU	Adjacent threads
Mali-G71	4
Mali-G72	4
Mali-G51	4
Mali-G31	4
Mali-G52	8
Mali-G76	8
Mali-G77	16

2. About OpenCL in General

2.1 OpenCL Terminology

Context	Term	Meaning or examples
Setup infrastructure	Device	CPU, GPU
Setup infrastructure	Context	Collection of devices
Setup infrastructure	Queue	Submit work to device. Each queue is attached to specific device.
Work with memory	Buffer	Block of raw memory
Work with memory	Images	Block of raw memory to 2D/3D formatted images
Execution of work	Program	Collection of kernels
Execution of work	Kernel	Execution instances

2.2 Some OpenCL and CUDA terminology

OpenCL	CUDA	Comment
Work-item	Thread	In fact it performs sequence of SIMD Lane operations
Wave-front	Warp	Thread of SIMD Instructions which worked in parallel.
WorkItems grouped into Wavefront and Wavefronts are grouped into WorkGroup	Threads are grouped into Warps and Warps are grouped into ThreadBlock	None
Can not sync for different WG	Can not sync between different TB	The same principle for scaling algorithms to more powerful

		devices is used in CUDA and OpenCL
Host	Host	Your computer with code which is typically running in CPU
Compute Device (CPU, GPU)	Device	Compute Device
Compute Unit (hardware device concept) Executes Workgroup (groups of parallel threads which is software concept)	Thread Block Execution, CTA (hardware device concept) Executes Blocls (group is parallel threads which is software concept)	
Processing Element	Scalar Core	Place where single thread is executed. Also it can be called a ***Lane*** both in OpenCL and CUDA.
NDRange	Grid	Spatial description of computation task

2.3 Tools to assist OpenCL development for ARM

Arm Development Studio – several debug tools from ARM in one. Includes emulator for ARM CPU chips.
https://developer.arm.com/tools-and-software/embedded/arm-development-studio

ARM Mali offline compiler to produce statistics for your kernels and check the ratio between arithmetic instructions and loads.
https://developer.arm.com/tools-and-software/graphics-and-gaming/arm-mobile-studio/components/mali-offline-compiler

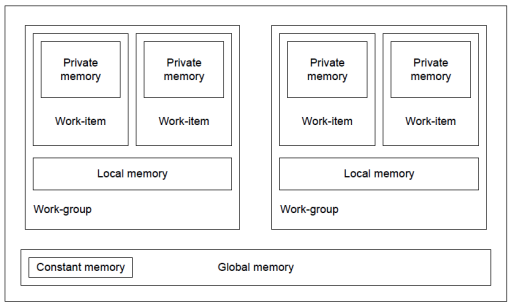
AMD CodeXL: https://gpuopen.com/compute-product/codexl/

AMD Mali Driver Development Kit:

https://developer.arm.com/ip-products/graphics-and-multimedia/mali-gpus/mali-driver-development-kit

3. OpenCL technical details

3.1 OpenCL memory model



The default address space for function arguments and local variables within a function or block is **private**.
Casting from one address space to another is not legal. Image arguments always live in the **global** address space.

The actual meaning of each memory space in terms of a hardware mapping is very much implementation dependent and should be checked for each specific hardware device.

1. Within a work-item, memory operations are ordered predictably. The rule: **no reorder happened in work-item**.
2. But between work-items within a specific work-group, memory is guaranteed to be consistent only after synchronization using an atomic operation, memory fence or barrier.
3. **Work-items from different work-groups cannot synchronize using a barrier.**

Within a work-group, the programmer may require all work-items in the work-group to synchronize at a barrier using a call to *work_group_barrier()*. The flags parameter to *work_group_barrier()* is used to specify which types of accesses must be visible after the barrier completes.

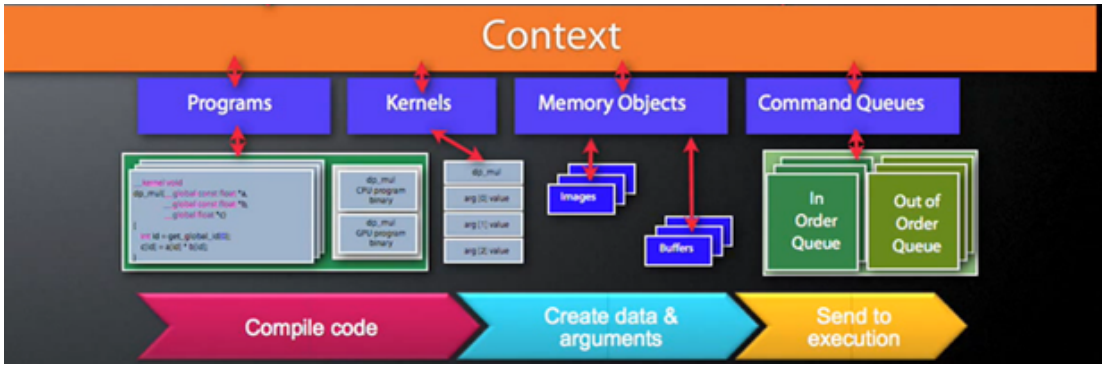
3.2 OpenCL platform and execution

Host connected to one or more OpenCL devices. Each OpenCL device is a collection of *compute units.* Each compute unit – contains several *Processing Elements*. Each Processing Element execute code.

Relative to CPU multiple core CPU is considered as a single device in terms of OpenCL.

OpenCL execute in data-parallel manner. All communication (e.g. submission of work) with devices are mostly happened with queue.

OpenCL platform has various objects and all objects are grouped by Context.



3.3 Kernel development relative terminology

Kernels have various built-in functions to work with several things.

Term	Meaning
Global ID	Every work-item has a unique global ID that identifies it within the index space
Work-group ID	Each work-group has a unique work-group ID
Local ID	Within each work-group, each work-item has a unique local ID
Private memory	Private memory is specific to a work-item. It is not visible to other work-items
Local memory	Local memory is local to a work-group. It is accessible by the work-items in the work-group. Special keyword **__local** is devoted to be a qualifier for this kind of memory.
Global memory	**Global memory** is accessible to all work-items executing in a context. Consistent across work-items in a single work-group. It is accessed with the **__global** keyword. There is no guarantee of memory consistency between different work-groups.
Kernel	All kernel functions must be identified in the application source with the **__kernel** qualifier

3.4 Kernel development requirements

Kernels are writing in OpenCL C language, which is derived from C99 with some difference from it:

1. No standard C99 headers
2. No function pointers, not recursion, no variable length arrays
3. Addition functions to identify work based on work-items and work-groups ideology.
   a. get_global_size
   b. get_work_dim
   c. get_local_size

d. get_num_groups
e. get_group_id
f. get_local_id
g. get_global_id

4. Built-in vector types
5. Synchronization primitives
6. Address space qualifiers
7. Optimized image access
8. Other built-in functions

3.5 Role of synchronization between work-items within a kernels

Work-items execute in an undefined order within work-group.

This means you cannot guarantee the order that work-items write data in output memory.

If you want a work-item to read data that are written by another work-item, you must use a barrier to ensure that work-items executed in the correct order.

barrier(CLK_LOCAL_MEM_FENCE); // Wait for all work-items in this work-group

After the synchronization will be completed, all writes to shared buffers are guaranteed to have been completed.

It is then safe for work-items within workgroup to read data written by other work-items, but which are within the same work-group.

3.6 How obtain meta-information with OpenCL API

Function	Description
clGetPlatformIDs()	Discover the set of available OpenCL platforms for a given system.
clGetPlatformInfo()	Determine by which implementation (vendor) the platform was defined.
clGetDeviceIDs()	Query the devices available to that platform
clGetDeviceInfo()	Retrieve information such as name, type, and vendor from each device
clGetSupportedImageFormats()	The list of supported image formats

3.7 Workgroups in OpenCL

1. Work-items within a work-group have a special relationship with another work-items. They can perform barrier operations to synchronize and they have access to a shared memory address space.

2. For programs such as vector addition in which work-items behave independently (even within a work-group) OpenCL allows the work-group size to be ignored by the programmer altogether and to be generated automatically by the implementation; in this case, the developer can pass NULL when defining the work-group size.

4. Steps to work with OpenCL

4.0. Check that algorithm at least can be look like can be parallelized in OpenCL execution paradigm

An application that involves global communication across its execution space is usually inefficient to parallelize with OpenCL.

4.1. Create Context

In OpenCL, a *context* is an abstract environment within which coordination and memory management for kernel execution. Use **clCreateContext()** to create it.

4.2. Create Command Queue

After creating your OpenCL context, use **clCreateCommandQueue()** or **clCreateCommandQueueWithProperties()** to create a command queue.

- OpenCL does not support the automatic distribution of work to devices.
- If you want to share work between devices, or have dependencies between operations enqueued on devices, then you must create the command queues in the same OpenCL context.

Actions specified by commands includes:

1. Executing kernels
2. Performing data transfers
3. Performing synchronization
4. For a device to send certain commands to itself

4.3. Create an OpenCL program object

clCreateProgramWithBinary() or **clCreateProgramWithSource()**

Program in OpenCL is analog to dynamic library.

Build OpenCL program **clBuildProgram()**. During building it's possible to pass compilation/build erros.

4.4. Create kernels

clCreateKernel() *or* ***clCreateKernelsInProgram()***

The final step of obtaining a cl_kernel object is similar to obtaining an exported function from a dynamic library.

4.5. Creating memory objects

To create buffer objects, use the **clCreateBuffer()** function. Buffers in some sense are equivalent to arrays in C created using *malloc()*.Buffer objects are one-dimensional arrays in the traditional CPU sense.

By standard - it is visible for all devices associated with the context.

Buffers can contain any scalar data type, vector data type, or user-defined structure.

To read buffer use: **clEnqueueReadBuffer()**.

It's Runtime that determinates the precise time when the data should be moved, it's not Software Engineer job.

To create image objects with specified format use the **clCreateImage()** function. Image formats are a combination of a channel order and a channel type.

Compared with buffers, the image read and write functions take additional parameters and are specific to the image's data type via *read_imagef()*.

Image structures are completely *opaque* not only to the developer, but also to the kernel code. Images are accessible only through specialized access. To read from host image use **clEnqueueReadImage()**

4.6. Executing the kernel

1. Determinate data dimension

2. Determining work-group and work-item sizes

3. Enqueuer execution via **clEnqueueNDRangeKernel()**

Launch Kernel Example

```
size_t globalWorkSize[1] = { ARRAY_SIZE };  // number of global work-items

size_t localWorkSize[1] = { 4 };                    // number of work-items that make up a work-group

errNum = clEnqueueNDRangeKernel(commandQueue, kernel, 1, NULL,

globalWorkSize, localWorkSize, 0, NULL, NULL);
```

Two forms of "waiting" result in Host

API call	Description
clFinish()	Blocks execution of the host thread until all of the commands in a command-queue **completed** execution.
clFlush()	Blocks execution until all of the commands in a command-queue **have been removed** from the queue.

	This means at this point of time commands will definitely have been submitted to the device. This in fact does not mean that commands have completed execution. Of course maybe it's true, but really it is not required.

4.7. Meta information obtaining

clGetKernelWorkGroupInfo(CL_KERNEL_WORK_GROUP_SIZE)

- maximum work-group size for a specific kernel

clGetDeviceInfo(CL_DEVICE_MAX_WORK_ITEM_SIZES)

- maximum sizes for the simplest kernel, and dimensions might be lower for more complex kernels

4.8. Reading results back

First way:

void* local_buffer = clEnqueueMapBuffer(queue, buffer, CL_NON_BLOCKING, CL_MAP_READ, 0, data_size, num_deps, &deps[0], NULL, &err);

Second way:

Or call clEnqueueWriteBuffer() and clEnqueueReadBuffer()

clFinish() must be called to make the buffer available or CL_BLOCKING should be specified during clEnqueueMapBuffer call

4.9. Releasing resources

The OpenCL context should be released last since all OpenCL objects such as buffers and command-queues are bound to a context.

4.10 Check status of submitted request/commands

Various asynchronous command if you will request will return event.

Via using event status you can monitor status of specific request.

1. **Queued:** The command has been placed into a command-queue.
2. **Submitted:** The command has been removed from the command-queue and has been submitted.
3. **Ready:** The command is ready for execution on the device.

4. **Running:** Execution of the command has started on the device.
5. **Ended:** Execution of the command has finished on the device.
6. **Complete:** The command and all of its child commands have finished.

5. About OpenCL in context of ARM Mali GPU

5.1 How abstract concepts from OpenCL are instantiated in ARM Mali GPU

1. ARM Mali GPUs have a unified memory system with the ARM CPU processor
2. In fact ARM Mali GPUs use global memory backed with caches in place of local or private memories
3. If you allocate local or private memory, it is allocated in global memory. Moving data from global to local or private memory typically does not improve performance
4. Copying data is not required, provided it is allocated by OpenCL in the correct manner
5. Each compute device, that is, shader-core, has its own data caches.
6. Use the OpenCL API to allocate memory buffers.

5.2 ARM Mali GPU specific type things

1. Mali GPUs can contain many identical **shader cores**

2. Shader Core

   a. Each shader core supports hundreds of concurrently executing threads.
   b. It has one to three arithmetic pipelines or execution engines.
   c. One load-store pipeline and one texture pipeline.

3. Scalar instructions are executed in parallel so the GPU operates on multiple data elements simultaneously. You are not required to vectorize your code to do this.

4. If you are targeting Mali GPUs, the global and local OpenCL address spaces are mapped to the same physical memory and access is accelerated by L1 and L2 caches. Therefore there are no performance advantage using local or private memories in OpenCL code for Mali GPUs.

5. This means that you are not required to use explicit data copies or implement the associated barrier synchronization

6. Before create OpenCL code for Mali GPUs, you must first remove all types of optimizations to create a non device-specific reference implementation.

5.3 Speedup technics and non-technics for Arm Mali GPUs architecture

1. If you are targeting Mali GPUs, the global and local OpenCL address spaces are mapped to the same physical memory and are accelerated by L1 and L2 caches. You are not required to use explicit data copies or implement the associated barrier synchronization.

2. The OpenCL local and private memories are mapped into main memory. There is therefore no performance advantage using local or private memories in OpenCL code for Mali GPUs.

3. Unfortunately using local or private memories can reduce performance in OpenCL on Mali GPUs

4. Mali GPUs have a cache line size of 64-bytes. It's useful to inspect data - structures and algorithms for data processing in subject of Cache Line size.

5. For high performance, do as many computations per memory access as possible.

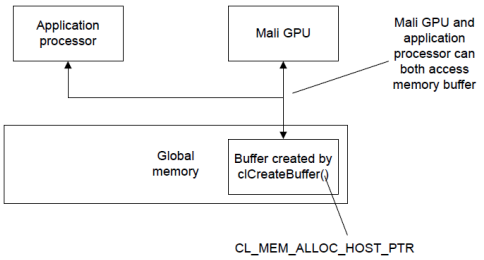
6. ARM recommends avoid *clFinish()* if possible because it serializes execution.

7. Calls to clFinish() introduce delays because the control thread must wait until all of the jobs in the queue to complete execution. The control thread is idle while it is waiting for this process to complete. Instead of clFinish() it's better to use clWaitForEvents() or callbacks.

8. Avoid making the copies, use the OpenCL API to allocate memory buffers and mapping operations.

9. Avoid use CL_MEM_USE_HOST_PTR, instead **use CL_MEM_ALLOC_HOST_PTR**.

The reason is that the Mali GPU can access the memory buffers created by clCreateBuffer(CL_MEM_ALLOC_HOST_PTR). It ensures that the memory pages are always mapped into physical memory.



10. Do not allocate memory buffers created with malloc() for OpenCL applications. Unfortuantely the Mali GPU cannot access the memory buffers created by malloc() because they are not mapped into the address space of the Mali GPU.

11. Fully coherent systems with Mali Bifrost or Valhall GPUs support fine-grained shared virtual memory in OpenCL 2.0. With full system coherency, application processors and GPUs can access memory without requiring cache clean. So it's better to use *OpenCL 2.0* standard for better performance.

12. Ensure that the threads within warp all take the same branch direction in if-statements and loops

13. If a thread requires more than 64 registers, the compiler might start storing register data in memory.

14. Using of shared virtual memory can give speedup

15. For Bifrost and Valhall GPUs, you manually enable kernel auto-vectorizer and a kernel unroller via: *fkernel-vectorizer, fkernel-unroller* kernel compiling flags.

6. About OpenCL code optimization and undefined behavior

6.1 Performance relative tricks about OpenCL kernels and under the hood understanding

1. Applications with pre-built program objects are not portable across platforms and driver version

2. The global work size is the total number of work-items required for all dimensions combined

3. You can change the global work size by the following trick - processing multiple data items in a single work-item

4. In case of processing several items in thread - the new global work size is then the original global work size divided by the number of data items processed by each work-item.

5. To get the maximum work-group size for a specific kernel, call clGetKernelWorkGroupInfo() with CL_KERNEL_WORK_GROUP_SIZE. If the maximum work-group size for a kernel is lower than 128, performance is reduced. If this is the case, try simplifying the kernel.

6. Queuing the kernel for execution does not mean that it executes immediately. The kernel execution is put into the *command queue* so the device can process it later

7. Kernels that are enqueued to an *in-order queue* automatically wait for kernels that were previously enqueued on the same queue

8. Ensure the reference counts for all OpenCL objects reach zero when your application no longer requires them. You can obtain the reference count by querying the object. For example, by calling *clGetMemObjectInfo()*.

9. Decrease reference counters occurred via using *clRelease*()*

10. If parallelizing the code appears to be impossible, **this only means that a particular code implementation cannot be parallelized, not algorithm**.

11. If tasks have few dependencies, it might be possible to run them in parallel. Dependencies between tasks prevent parallelization because they force tasks to be performed sequentially.

12. If the loop only processes a relatively small number of elements, it might not be appropriate for data parallel processing via OpenCL. It might be better to parallelize these sorts of loops with task parallelism on one or more application processors.

13. If the loop is part of a series of nested loops and the total number of iterations is large, this loop is probably appropriate for parallel processing.

14. Perfect loops - process thousands of items. Have no dependencies on previous iterations. Access data independently in each iteration.

15. If the loop contains dependencies that you cannot remove, investigate alternative methods of performing the computation and which might be parallelizable.

16. Parallel processing techniques in OpenCL
    a. Different ways of computing values
    b. Removing dependencies
    c. Software pipelining
    d. Task parallelism

17. To split dependencies we can use two buffers for read and write even for example in sequential program you could use one buffer.

18. Parallel processing with non-parallelizable code
    a. Use parallel versions of your data structures and algorithms
    b. Solve the problem in a different way

19. Work-items can write to the same address really only in one case. And here this case: “If it is guaranteed that both work - items write the same value to the element”

6.2 Implementation defined and undefined behavior

1. The actual subtle behavior associated with memory is in fact is implementation defined, and even more can depend on device.

2. The result of reading from a memory object while another kernel is modifying it is undefined.

3. Undefined behavior occurs if and memory object that is currently mapped for reading by the host is written to by a device.

4. Undefined behavior occurs if an object that is currently mapped for *writing* by the host is read by a device in the same time.

4. Calling *clSVMFree()* and then accessing a buffer can therefore result in a segmentation fault as can happen in a normal C program.

6.3 Various Tips and Tricks

1. To have ability to generate both the final binary format and various intermediate representations and serialize them you could call **clGetProgramInfo() with CL_PROGRAM_BINARIES**

2. For discrete GPUs when **clEnqueueWriteBuffer()** or **clEnqueueReadBuffer()** commands is executing actual transfer across **PCI-Express** is happened.

3. OpenCL provides a command, **clEnqueueMigrateMemObjects()**, to *migrate* data from its current location (wherever that may be) to a device for which command queue have been created. Command Queue is one of the parameters of this API.

4. Providing the option CL_MEM_ALLOC_HOST_PTR to flags in **clCreateBuffer()** tells the runtime to allocate new space for the object in "host-accessible" memory, and CL_MEM_USE_HOST_PTR tells the runtime to use the space pointed to by host_ptr directly.

5. **clEnqueueReadBuffer()** would always result in a copy of the data. Mapping via **clEnqueueMapBuffer()** does not necessarily imply creating a copy. The command to unmap is the same for all memory objects: **clEnqueueUnmapMemObject()**

6. Three ways to wait for result

- Waiting for the completion of a specific OpenCL event via **clWaitForEvents()**
- **clFinish()** call that blocks the host's execution until an entire queue completes execution.
- Execution of a blocking memory operation

7. Subtle things with errors.

OpenCL API calls cannot simply return error conditions or profiling data that relate to the execution due to it's asynchronous design.

The API calls only return error conditioning on relating information known at enqueue time

(e.g., validity of parameters).

clGetEventInfo with *param_name = CL_EVENT_COMMAND_EXECUTION_STATUS*

Unsuccessful completion results in abnormal termination of the command is indicated by setting the event status to a negative value.

8. Wait for events and syncronization

clEnqueueBarrierWithWaitList() // marker does not block the execution of subsequent commands

clEnqueueMarkerWithWaitList()

clWaitForEvents()

clSetEventCallback()

Synchronization using OpenCL events can be done only for commands within the same context.

9. Alternative to out-of-order queues

OpenCL allows multiple command queues from a context to be mapped to the same device. This is potentially useful to overlap execution of independent command so overlap commands and host device communication, and is an alternative to out-of-order queues.

6.4 OpenCL scaling

If you want to leverage in several devices during execution

1. Pipelined execution. Two or more devices work in a pipeline manner such that one device waits on the results of another.

2. Independent execution. A scenario in which multiple devices work independently of each other

Within each work-group, some degree of communication is allowed.

The OpenCL specification defines that an entire work-group can run concurrently on an element of the device known as a Compute Unit.

Because of this SIMD execution, it is often noted that for a given device, an OpenCL work-group's size should be an even multiple of that device's SIMD width.

This value can be queried from the runtime as the parameter **CL_KERNEL_PREFERRED_WORK_GROUP_SIZE_MULTIPLE** to the **clGetKernelWorkGroupInfo()** function

7. Features and benefits of OpenCL 2.0

7.1 Device-side queuing

A kernel executing on a device in OpenCL 2.0 has the ability to *enqueue* another kernel into a device-side-command-queue. In this scenario, the kernel currently executing on a device is referred to as the ***parent kernel***, and the kernel that is enqueued is known as the ***child kernel***.

Parent kernel is not registered as **completed** until all its child kernels have completed.

7.2 Arbitrarily Work-group size

In previous versions of the **OpenCL specification**, the index space dimensions would have to be rounded up to be a multiple of the work-group dimensions. OpenCL 2.0 specification allows each dimension of the index space that is not evenly divisible by the work-group size to be divided into two regions.

get_local_size() and get_enqueued_local_size() can be used to get need info.

Also *OpenCL 2.0* has also introduced built-in functions for linear indexing that simplifies a common calculation that programmers had to code by hand.

size_t get_global_linear_id() Returns a one-dimensional global ID for the work-item.

size_t get_local_linear_id() Returns a one-dimensional local ID for the work-item.

7.3 Possibility write/read into image from the kernel

In previous versions of the OpenCL standard, a kernel was **not allowed** to both read from and write to a single image. However, OpenCL 2.0 has relaxed this restriction by providing synchronization operations that let programmers safely read and write a single image within a kernel.
Images types: image1d_t, image2d_t, image3d_t

7.4 Generic address space

The generic address space was added in OpenCL 2.0 supports conversion of pointers to and from private, local, and global address spaces, and hence lets a programmer write a single function that at compile time can take arguments from any of the three named address spaces.

Starting in *OpenCL 2.0*, pointers to a named address spaces can be implicitly converted to the generic address space

```
void doDoubleLocal (__local float * data, int index ) {
 data [index] *= 2;
}
void doDoubleGlobal (__global float * data, int index ) {
 data [index] *= 2;
}
// =>

void doDouble ( float * data , int index ) {
 data[index] *= 2;
}
```

7.5 Shared virtual memory (SVM)

One of the most significant updates to **OpenCL 2.0** is the support of *SVM* (**Shared Virtual Memory**).

SVM extends global memory into the host's memory region, allowing virtual addresses to be shared between the host and all devices in a context.

Instead using memory buffers clCreateBuffer() – use clSVMAlloc() (similar to host malloc() by meaning). To free you should call - clSVMFree() or clEnqueueSVMFree().

7.6 Atomics

The atomics defined in OpenCL 2.0 are based on C/C++11 atomics and are used to provide atomicity and synchronization.

7.7 Built-in reduce, scan and predicates in kernel level

OpenCL supports two built-in parallel primitive functions **reduce and scan**.

This operation are performed for all work-items within work-group.

E.g.: *float max = work_group_reduce_max(local_data[get_local_id(0)]);*

float prefix_sum_val = work_group_scan_inclusive_add(local_data[get_local_id(0)]);

Predicates:

int work_group_any(int predicate)

int work_group_all(int predicate)

This operations are performed for all work-items within work-group.

7.8 Improve global memory mechanism

An important addition to the OpenCL 2.0 specification is optional support of consistency guarantees. Any movement of data in and out of OpenCL memory objects from a CPU pointer must be performed through application programming interface (API) functions.

It is important to note that OpenCL's memory objects are defined within a context and not on a device. It is the job of the runtime to ensure that data is in the correct place at the correct time.

7.9 Nested parallelism

OpenCL 2.0 has lifted this restriction by defining device-side command-queues, which allow a child kernel to be enqueued directly from a kernel executing on a device (referred to as the parent kernel).

The main benefit of a device-side command-queue is that it enables nested parallelism—a parallel programming paradigm where a thread executing a parallel task can spawn additional threads to execute additional tasks. Nested parallelism benefits applications with an irregular or data-driven loop structure.

A common data-driven algorithm is the breadth-first search (BFS) graph algorithm.

Previously, algorithms containing recursion, irregular loop structures, or other constructs that do not fit a uniform single level of parallelism had to be redesigned for OpenCL.

To launch kernels you have to use OpenCL C built-in function *enqueue_kernel()*. This function require to use command-queue. In fact command queue should be created in *host – side with*

CL_QUEUE_ON_DEVICE | CL_QUEUE_OUT_OF_ORDER_EXEC_MODE_ENABLE flags.

7.10 Memory pipes

OpenCL 2.0 provides a new type of memory object called a pipe. A pipe memory object is an ordered sequence of data items (referred to as *packets*) that are stored on the basis of a first in, first out (FIFO) method. Any device that can support pipes must at least have the ability to implement atomic operations on data shared between kernels, and must have a memory consistency model that supports acquire and release semantics. Pipes is a nice abstraction that enables producer-consumer parallelism.

Relative API: *clCreatePipe(),clGetPipeInfo()*

Kernel declaration example: *kernel void foo(read_only pipe int pipe0, write_only pipe float4 pipe1)*

Rules for working with pipes:

1. At any time, only one kernel may write into a pipe, and only one kernel may read from a pipe.
2. The same kernel may not be both the writer and the reader for a pipe.
3. Work with pipe is going via read_pipe() and write_pipe()
4. When creating a pipe using the OpenCL API call clCreatePipe(), one must supply the packet size along with the maximum number of entries in the pipe

8. References

[1] Heterogeneous Computing with OpenCL 2.0 , David R. Kaeli (Author), Perhaad Mistry (Author), Dana Schaa (Author), Dong Ping Zhang (Author)