Subject: PARALLEL COMPUTER ARCHITECTURE AND PROGRAMMING

Subject Code: CSE 3252

The main functions used in CUDA programs are as follows:

cudaMemcpy:

__host__cudaError_t cudaMemcpy (void* dst, const void* src, size_t count, cudaMemcpyKind kind)

Copies data between host and device.

Parameters: dst- Destination memory address src- Source memory address

count- Size in bytes to copy kind- Type of transfer

Returns: cudaSuccess, cudaErrorInvalidValue, cudaErrorInvalidMemcpyDirection

Description: Copies count bytes from the memory area pointed to by src to the memory area pointed to by dst, where kind specifies the direction of the copy, and must be one of cudaMemcpyHostToHost, cudaMemcpyHostToDevice, cudaMemcpyDeviceToHost, cudaMemcpyDeviceToDevice, or cudaMemcpyDefault. Passing cudaMemcpyDefault is recommended, in which case the type of transfer is inferred from the pointer values. However, cudaMemcpyDefault is only allowed on systems that support unified virtual addressing. Calling cudaMemcpy() with dst and src pointers that do not match the direction of the copy results in an undefined behaviour.

cudaFree:

__host____device__cudaError_t cudaFree (void* devPtr)

Frees memory on the device.

Parameters: devPtr - Device pointer to memory to free

Returns: cudaSuccess, cudaErrorInvalidValue

Description: Frees the memory space pointed to by devPtr, which must have been returned by a previous call to cudaMalloc(). Otherwise, or if cudaFree(devPtr) has already been called before, an error is returned. If devPtr is 0, no operation is performed. cudaFree() returns cudaErrorValue in case of failure.

cudaMalloc:

__host____device__cudaError_t cudaMalloc (void** devPtr, size_t size)

Allocate memory on the device.

Parameters: devPtr - Pointer to allocated device memory size - Requested allocation size in bytes

Returns: cudaSuccess, cudaErrorInvalidValue, cudaErrorMemoryAllocation

Description: Allocates size bytes of linear memory on the device and returns in *devPtr a pointer to the allocated memory. cudaMalloc() returns cudaErrorMemoryAllocation in case of failure.

Declaring variables in Constant Memory:

To declare an *M* array in constant memory, the host code declares it as follows: This is a global variable declaration and should be outside of any function in the source file.

#define MAX_MASK_WIDTH 10 __constant__ float M[MAX_MASK_WIDTH];

cudaMemcpyToSymbol:

This is a special memory copy function that informs the CUDA runtime that the data being copied into the constant memory will not be changed during kernel execution.

cudaError_t cudaMemcpyToSymbol(const char * symbol, const void * src, size_t count, size_t offset = 0, enum cudaMemcpyKind kind = cudaMemcpyHostToDevice)

Parameters: symbol - Symbol destination on device src - Source memory address

count - Size in bytes to copy offset - Offset from start of symbol in bytes

kind - Type of transfer

Returns: cudaSuccess, cudaErrorInvalidValue, cudaErrorInvalidSymbol, cudaErrorInvalidDevicePointer, cudaErrorInvalidMemcpyDirection

Description: Copies count bytes from the memory area pointed to by src to the memory area pointed to by offset bytes from the start of symbol symbol. The memory areas may not overlap. symbol can either be a variable that resides in global or constant memory space, or it can be a character string, naming a variable that resides in global or constant memory space. kind can be either cudaMemcpyHostToDevice or cudaMemcpyDeviceToDevice.

Querying Device Properties:

cudaGetDeviceCount:

__host____device__cudaError_t cudaGetDeviceCount (int* count)

Returns the number of compute-capable devices.

Parameters: count- Returns the number of devices

Returns: cudaSuccess

Description: Returns in *count the number of devices with compute capability greater or equal to 2.0 that are available for execution.

cudaGetDeviceProperties:

__host__cudaError_t cudaGetDeviceProperties (cudaDeviceProp* prop, int device)

Returns information about the compute-device.

Parameters:

prop- Properties for the specified device device - Device number to get properties for

Returns: cudaSuccess, cudaErrorInvalidDevice

Description: Returns in *prop the properties of device dev. The cudaDeviceProp structure has many fields like size_t sharedMemPerBlock; int regsPerBlock etc. to obtain information about the compute device.

Functions used in the calculation of time taken by the kernel function:

cudaEventElapsedTime:

__host__cudaError_t cudaEventElapsedTime (float* ms, cudaEvent_t start, cudaEvent_t end)

Computes the elapsed time between events.

Parameters: Ms - Time between start and end in ms start- Starting event end - Ending event

Returns:cudaSuccess, cudaErrorNotReady, cudaErrorInvalidValue, cudaErrorInvalidResourceHandle, cudaErrorLaunchFailure

Description: Computes the elapsed time between two events (in milliseconds with a resolution of around 0.5 microseconds).

cudaEventCreate:

__host__cudaError_t cudaEventCreate (cudaEvent_t* event)

Creates an event object.

Parameters: event- **Newly created event**

Returns: cudaSuccess, cudaErrorInvalidValue, cudaErrorLaunchFailure, cudaErrorMemoryAllocation

Description: Creates an event object for the current device using cudaEventDefault

cudaEventRecord:

__host____device__cudaError_t cudaEventRecord (cudaEvent_t event, cudaStream_t stream = 0)

Records an event.

Parameters : event - Newly created event

Returns: cudaSuccess, cudaErrorInvalidValue, cudaErrorLaunchFailure, cudaErrorMemoryAllocation

Description:Creates an event object for the current device using cudaEventDefault.

Parameters : event- Event to record stream- Stream in which to record event

Returns:cudaSuccess, cudaErrorInvalidValue, cudaErrorInvalidResourceHandle, cudaErrorLaunchFailure

Description: Captures in event the contents of stream at the time of this call. event and stream must be on the same device. Calls such as cudaEventQuery() or cudaStreamWaitEvent() will then examine or wait for completion of the work that was captured. Uses of stream after this call do not modify event. See note on default stream behavior for what is captured in the default case.

cudaEventSynchronize:

__host__cudaError_t cudaEventSynchronize (cudaEvent_t event)

Waits for an event to complete.

Parameters: event- Event to wait for

Returns: cudaSuccess, cudaErrorInvalidValue, cudaErrorInvalidResourceHandle,

cudaErrorLaunchFailure

Description: Waits until the completion of all work currently captured in event.

Error Handling in CUDA programs:

cudaGetLastError:

__host____device__cudaError_t cudaGetLastError (void)

Returns the last error from a runtime call.

Returns

cudaSuccess, cudaErrorMissingConfiguration, cudaErrorMemoryAllocation, cudaErrorInitializationError, cudaErrorLaunchFailure, cudaErrorLaunchTimeout, cudaErrorLaunchOutOfResources, cudaErrorInvalidDeviceFunction, cudaErrorInvalidConfiguration, cudaErrorInvalidDevice, cudaErrorInvalidValue, cudaErrorInvalidPitchValue, cudaErrorInvalidSymbol, cudaErrorUnmapBufferObjectFailed, cudaErrorInvalidDevicePointer, cudaErrorInvalidTexture, cudaErrorInvalidTextureBinding, cudaErrorInvalidChannelDescriptor, cudaErrorInvalidMemcpyDirection, cudaErrorInvalidFilterSetting, cudaErrorInvalidNormSetting, cudaErrorUnknown, cudaErrorInvalidResourceHandle, cudaErrorInsufficientDriver, cudaErrorNoDevice, cudaErrorSetOnActiveProcess, cudaErrorStartupFailure, cudaErrorInvalidPtx, cudaErrorUnsupportedPtxVersion, cudaErrorNoKernelImageForDevice, cudaErrorJitCompilerNotFound, cudaErrorJitCompilationDisabled

Description

Returns the last error that has been produced by any of the runtime calls in the same host thread and resets it to cudaSuccess.

OpenCL APIs

Memory objects

Create device buffers

cl_mem clCreateBuffer (cl_context context, cl_mem_flags flags, size_t size, void *host_ptr,

cl_int *errcode_ret);

Parameters

- context is a valid OpenCL context used to create the buffer object.
- flags is a bit-field that is used to specify allocation and usage information such as the memory arena that should be used to allocate the buffer object and how it will be used. The Memory Flags table describes the possible values for flags. If the value specified for flags is 0, the default is used which is CL_MEM_READ_WRITE.
- size is the size in bytes of the buffer memory object to be allocated.

- host_ptr is a pointer to the buffer data that may already be allocated by the application. The size of the buffer that host_ptr points to must be ≥ size bytes.

cl_program clCreateProgramWithSource(cl_context context, cl_uint count, const char** strings, const size_t* lengths, cl_int* errcode_ret);

Parameters

- context must be a valid OpenCL context.
- strings is an array of count pointers to optionally null-terminated character strings that make up the source code.
- lengths argument is an array with the number of chars in each string (the string length). If an element in lengths is zero, its accompanying string is null-terminated. If lengths is NULL, all strings in the strings argument are considered null-terminated. Any length value passed in that is greater than zero excludes the null terminator in its count.
- errcode_ret will return an appropriate error code. If errcode_ret is NULL, no error code is returned.

Discover & initialize Devices

cl_GetDeviceIDs() call works very similar to cl_GetPlatformIDs().

cl_int clGetDeviceIDs (cl_platform_id platform, cl_device_type device_type,

cl_uint num_entries, cl_device_id *devices, cl_uint *num_devices);

Parameters

- platform refers to the platform ID returned by clGetPlatformIDs or can be NULL. If platform is NULL, the behavior is implementation-defined.
- device_type is a bitfield that identifies the type of OpenCL device. The device_type can be used to query specific OpenCL devices or all OpenCL devices available. The valid values for device_type are specified in the Device Categories table.
- num_entries is the number of cl_device_id entries that can be added to devices. If devices is not NULL, the num_entries must be greater than zero.
- devices returns a list of OpenCL devices found. The cl_device_id values returned in devices can be used to identify a specific OpenCL device. If devices is NULL, this argument is ignored. The number of OpenCL devices returned is the minimum of the value specified by

num_entries or the number of OpenCL devices whose type matches device_type.

- num_devices returns the number of OpenCL devices available that match device_type. If num_devices is NULL, this argument is ignored.

cl_int clBuildProgram(cl_program program, cl_uint num_devices, const cl_device_id* device_list, const char* options, void (CL_CALLBACK* pfn_notify)(cl_program program, void* user_data), void* user_data);

Parameters

- program is the program object.
- device_list is a pointer to a list of devices associated with program. If device_list is a NULL value, the program executable is built for all devices associated with program for which a source or binary has been loaded. If device_list is a non-NULL value, the program executable is built for devices specified in this list for which a source or binary has been loaded.
- num_devices is the number of devices listed in device_list.
- options is a pointer to a null-terminated string of characters that describes the build options to be used for building the program executable. The list of supported options is described in Compiler Options. If the program was created using clCreateProgramWithBinary and options is a NULL pointer, the program will be built as if options were the same as when the program binary was originally built. If the program was created using clCreateProgramWithBinary and options string contains anything other than the same options in the same order (whitespace ignored) as when the program binary was originally built, then the behavior is implementation defined. Otherwise, if options is a NULL pointer then it will have the same result as the empty string.
- pfn_notify is a function pointer to a notification routine. The notification routine is a callback function that an application can register and which will be called when the program executable has been built (successfully or unsuccessfully).

Create command queue

- clCreateCommandQueue: Communication with a device occurs by submitting commands to a command queue.

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

Parameters:

- context must be a valid OpenCL context.
- device must be a device or sub-device associated with context. It can either be in the list of devices and sub-devices specified when context is created using clCreateContext or be a root device with the same device type as specified when context is created using clCreateContextFromType.

• properties specifies a list of properties for the command-queue. This is a bit-field and the supported properties are described in the table below. Only command-queue properties specified in this table can be used, otherwise the value specified in properties is considered to be not valid. properties can be 0 in which case the default values for supported command-queue properties will be used.

Discover & initialize platforms

• CLGetPlatformIDs() – It is used to discover the set of available platforms for a given system.

cl_int clGetPlatformIDs (cl_uint num_entries, cl_platform_id *platforms,

cl_uint *num_platforms);

Parameters:

• num_entries is the number of cl_platform_id entries that can be added to platforms. If platforms is not NULL, the num_entries must be greater than zero.

• platforms returns a list of OpenCL platforms found. The cl_platform_id values returned in platforms can be used to identify a specific OpenCL platform. If platforms is NULL, this argument is ignored. The number of OpenCL platforms returned is the minimum of the value specified by num_entries or the number of OpenCL platforms available.

• num_platforms returns the number of OpenCL platforms available. If num_platforms is NULL, this argument is ignored.

Create the Kernel

• clCreateKernel ():

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

Parameters: kernel_name - A function name in the program declared with the __kernel qualifier

Set the kernel arguments

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

• 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.

Set/get global & local work size

• uint get_work_dim(): Returns the number of dimensions in use. This is the value given to the work_dim argument specified in clEnqueueNDRangeKernel.

- size_t get_global_size(uint dimindx): Returns the number of global work-items specified for dimension identified by dimindx. This value is given by the global_work_size argument to clEnqueueNDRangeKernel.

- size_t get_global_id(uint dimindx): Returns the unique global work-item ID value for dimension identified by dimindx. The global work-item ID specifies the work-item ID based on the number of global work-items specified to execute the kernel.

- size_t get_local_size(uint dimindx): Returns the number of local work-items specified in dimension identified by dimindx. This value is at most the value given by the local_work_size argument to clEnqueueNDRangeKernel.

- size_t get_local_id(uint dimindx): Returns the unique local work-item ID, i.e. a work-item within a specific work-group for dimension identified by dimindx.

Write host data to device buffers

- clEnqueueWriteBuffer():

cl_int clEnqueueReadBuffer(cl_command_queue command_queue, cl_mem buffer,cl_bool blocking_read,size_t offset,size_t size,void* ptr,cl_uint num_events_in_wait_list,const cl_event* event_wait_list,cl_event* event);

cl_int clEnqueueWriteBuffer (cl_command_queue command_queue, cl_mem buffer,

cl_bool blocking_write, size_t offset, size_t cb, const void

*ptr, cl_uint num_events_in_wait_list, const cl_event

*event_wait_list, cl_event *event);

Parameters

- command_queue is a valid host command-queue in which the read / write command will be queued. command_queue and buffer must be created with the same OpenCL context.

- buffer refers to a valid buffer object.

- blocking_read and blocking_write indicate if the read and write operations are blocking or non-blocking (see below).

- offset is the offset in bytes in the buffer object to read from or write to.

- size is the size in bytes of data being read or written.

- ptr is the pointer to buffer in host memory where data is to be read into or to be written from.

- event_wait_list and num_events_in_wait_list specify events that need to complete before this particular command can be executed. If event_wait_list is NULL, then this particular command does not wait on any event to complete. If event_wait_list is NULL,

num_events_in_wait_list must be 0. If event_wait_list is not NULL, the list of events pointed to by event_wait_list must be valid and num_events_in_wait_list must be greater than 0. The events specified in event_wait_list act as synchronization points. The context associated with events in event_wait_list and command_queue must be the same. The memory associated with event_wait_list can be reused or freed after the function returns.

- event returns an event object that identifies this particular read / write command and can be used to query or queue a wait for this particular command to complete. event can be NULL in which case it will not be possible for the application to query the status of this command or queue a wait for this command to complete. If the event_wait_list and the event arguments are not NULL, the event argument should not refer to an element of the event_wait_list array.

Barrier Operations for a command queue

cl_int clFinish (cl_command_queue cmdQueue);

cl_int clFlush (cl_command_queue cmdQueue);

APIs used to find the time taken by kernel:

- clGetEventProfilingInfo():

cl_int clGetEventProfilingInfo (cl_event event, cl_profiling_info param_namesize_t param_value_size, void *param_value, size_t *param_value_size_ret);

Parameters:

- event specifies the event object.

- param_name specifies the profiling data to query. The list of supported param_name types and the information returned in param_value by clGetEventProfilingInfo is described in the Event Profiling Queries table.

- param_value is a pointer to memory where the appropriate result being queried is returned. If param_value is NULL, it is ignored.

- param_value_size is used to specify the size in bytes of memory pointed to by param_value. This size must be ≥ size of return type as described in the Event Profiling Queries table.

- param_value_size_ret returns the actual size in bytes of data being queried by param_name. If param_value_size_ret is NULL, it is ignored.

Execution Environment- Context

- clCreateContext () : Context is an abstract container that exists on the host.

cl_context clCreateContext (const cl_context_properties *properties,

cl_uint num_devices, const cl_device_id *devices, void

(CL_CALLBACK *pfn_notify)(const char *errinfo, const void *private_info, size_t cb, void *user_data),void *user_data, cl_int *errcode_ret);

Parameters

- properties specifies a list of context property names and their corresponding values. Each property name is immediately followed by the corresponding desired value. The list is terminated with 0. The list of supported properties is described in the Context Properties table. properties can be NULL in which case the platform that is selected is implementation-defined.
- num_devices is the number of devices specified in the devices argument.
- devices is a pointer to a list of unique devices9 returned by clGetDeviceIDs or sub-devices created by clCreateSubDevices for a platform.
- pfn_notify is a callback function that can be registered by the application. This callback function will be used by the OpenCL implementation to report information on errors during context creation as well as errors that occur at runtime in this context. This callback function may be called asynchronously by the OpenCL implementation. It is the applications responsibility to ensure that the callback function is thread-safe. If pfn_notify is NULL, no callback function is registered.
- user_data will be passed as the user_data argument when pfn_notify is called. user_data can be NULL.
- errcode_ret will return an appropriate error code. If errcode_ret is NULL, no error code is returned.

Enqueue the kernel for execution:

cl_int clEnqueueNDRangeKernel(cl_command_queue command_queue, cl_kernel kernel, cl_uint work_dim, const size_t *global_work_offset, const size_t *global_work_size, const size_t *local_work_size, cl_uint num_events_in_wait_list, const cl_event *event_wait_list, cl_event *event);

4 fields are related to work-item creation:

- work_dim: specifies the number of dimensions in which work-item will be created
- global_work_size: specifies the number of work items in each dimension of the ND range
- local_work_size: specifies the number of work items in each dimension of the workgroup

- global_work_offset: used to provide global IDs to the work items that do not start from zero

Read the output buffer back to the host

```
cl_int  clEnqueueReadBuffer ( cl_command_queue command_queue, cl_mem buffer,
                              cl_bool blocking_write, size_t offset, size_t cb, const
                              void *ptr, cl_uint num_events_in_wait_list, const
                              cl_event *event_wait_list, cl_event *event);
```

Parameters

- command_queue is a valid host command-queue in which the read / write command will be queued. command_queue and buffer must be created with the same OpenCL context.
- buffer refers to a valid buffer object.
- blocking_read and blocking_write indicate if the read and write operations are blocking or non-blocking (see below).
- offset is the offset in bytes in the buffer object to read from or write to.
- size is the size in bytes of data being read or written.
- ptr is the pointer to buffer in host memory where data is to be read into or to be written from.
- event_wait_list and num_events_in_wait_list specify events that need to complete before this particular command can be executed. If event_wait_list is NULL, then this particular command does not wait on any event to complete. If event_wait_list is NULL, num_events_in_wait_list must be 0. If event_wait_list is not NULL, the list of events pointed to by event_wait_list must be valid and num_events_in_wait_list must be greater than 0. The events specified in event_wait_list act as synchronization points. The context associated with events in event_wait_list and command_queue must be the same. The memory associated with event_wait_list can be reused or freed after the function returns.
- event returns an event object that identifies this particular read / write command and can be used to query or queue a wait for this particular command to complete. event can be NULL in which case it will not be possible for the application to query the status of this command or queue a wait for this command to complete. If the event_wait_list and the event arguments are not NULL, the event argument should not refer to an element of the event_wait_list array.