OpenCL - Heterogeneous Computing

Soumyajit Dey, Assistant Professor,
CSE, IIT Kharagpur

January 23, 2020

Course Organization

Topic	Week	Hours
Review of basic COA w.r.t. performance	1	2
Intro to GPU architectures	2	3
Intro to CUDA programming	3	2
Multi-dimensional data and synchronization	4	2
Warp Scheduling and Divergence	5	2
Memory Access Coalescing	6	2
Optimizing Reduction Kernels	7	3
Kernel Fusion, Thread and Block Coarsening	8	3
OpenCL - runtime system	9	3
OpenCL - heterogeneous computing	10	2
Efficient Neural Network Training/Inferencing	11-12	6

Recap

- Introduction to OpenCL
- OpenCL runtime system
- Synchronization in OpenCL

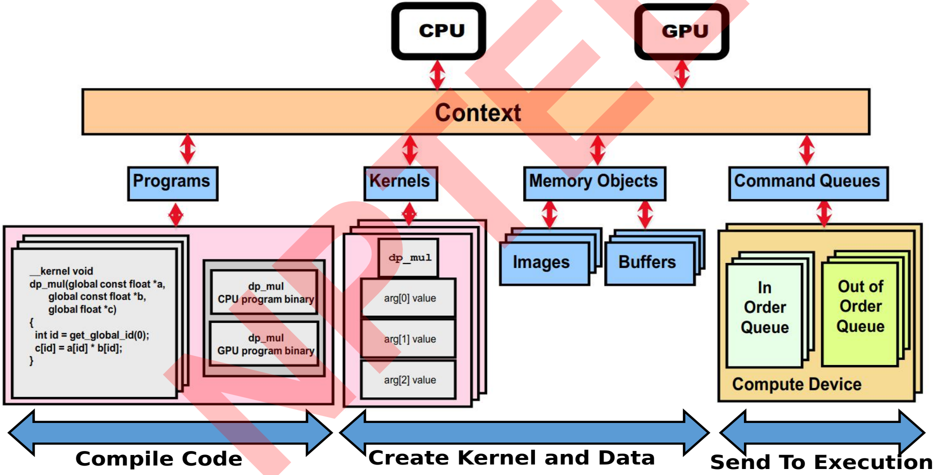
Heterogeneous Computing

- Computing platforms with more than one kind of devices (processor or cores) are called heterogeneous platform
- Heterogeneous Computing utilise this heterogeneity to gain performance or energy efficiency
- Concurrency is where applications execute functions concurrently on multiple processors
- OpenCL is an ideal programming language for heterogeneous computing implementation as it support programming across multiple computing devices, such as CPU, GPU, and FPGA from different vendors

Heterogeneous Computing

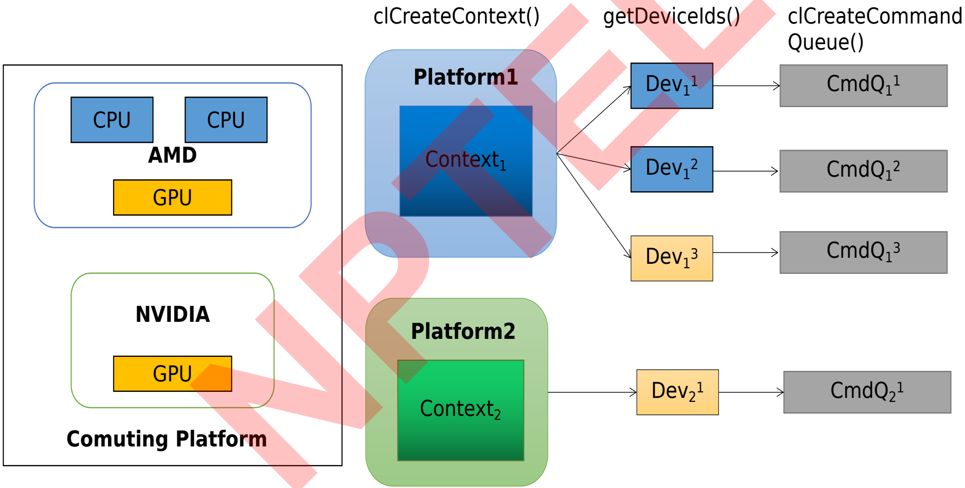
- Available processors in the system should be efficiently used in heterogeneous computing
- Challenge is to identify preffered task to device mapping, minimize overhead due to data transfer, synchronization etc. in such heterogeneous system
- To take full advantage of this heterogeneity to gain performance or energy efficiency, programmer need to handle these challenges efficiently

OpenCL Recap: Runtime System



(Figure from reference[2])

OpenCL Recap: Heterogeneous Computing



Command Queue and Device

- A command-queue can be associated with only one device
- Single device may be associated with single command-queue (already discussed)
- Single device may be associated with multiple command-queues with same context
- Single device may also be associated with multiple command queues associated with different contexts within the same platform
- Different devices may also be associated with multiple command queues associated with the same context
- Different devices may also be associated with multiple command queues associated with different contexts

Multiple Command Queues With Same Context In Single Device

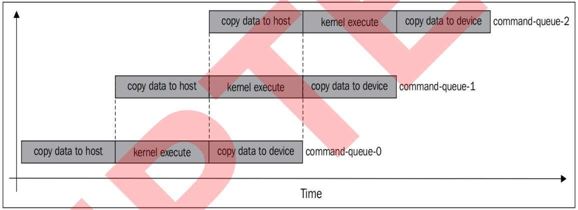
- Multiple command-queues can be mapped to the same device to overlap execution of different commands or overlap commands and host–device communication
- They execute commands independently
- They allows applications to queue multiple independent commands without requiring synchronization as long as these objects are not being shared

Multiple Command Queues With Same Context In Single Device Example

As an example of independent streams of computation we assume three independent tasks that need to execute on a device

- Three command queues (in-order execution only) with tasks enqueued in each of them
- A pipeline can be formed, such that the device executes the kernel code while I/O is being performed
- This achieve better utilization by not having the device sit idle waiting for data

Multiple Command Queues With Same Context In Single Device Example



(Figure from reference[3])

Multiple Command Queues With Different Contexts In Single Device

- Yes, we can create multiple contexts for the same device on the same application.
- Typically it has no benefit
- Useful when an application that use OpenCL for certain operations also use some third-party library that also happens to use OpenCL internally to accelerate some algorithms

Multiple Device Programming

Multiple device programming with OpenCL can be summarized with two execution models:

- Two or more devices work in a pipeline manner such that one device waits on the results of another
- A model in which multiple devices work concurrently, independent of each other

Multiple Command Queues With Same Context In Different Devices

- For multiple devices in a system (e.g., a CPU and a GPU or multiple GPUs), each device needs its own command queue
- Standard way to work with multiple devices on same platform is creating single context as-
  - Memory objects are globally visible to all devices within the same context
  - An event is only valid in a context where it was created

Multiple Command Queues With Same Context In Different Devices

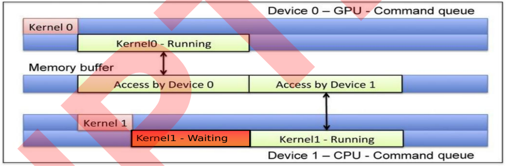
- Within same context, sharing of objects across multiple command-queues will require the application to perform appropriate synchronization
- Event objects visible to the host program can be used to define synchronization points between commands in multiple command queues
- If synchronization points are established , the programmer must assure that the command-queues progress concurrently and correctly establish existing dependencies

Multiple Command Queues Creation With Same Context In Different Devices Example

```
//One platform having one CPU device and one GPU device
cl_uint num_devices;
cl_device_id devices[2];
cl_context context;

//Obtain devices of both CPU and GPU types
err_code = clGetDeviceIDs(NULL,CL_DEVICE_TYPE_GPU,1,&devices[0],&num_devices);
err_code = clGetDeviceIDs(NULL,CL_DEVICE_TYPE_CPU,1,&devices[1],&num_devices);

//Create a context including two devices
context = clCreateContext(0, 2, devices, NULL, NULL, &err);
cl_command_queue queue_cpu, queue_gpu;

//Create queues to each device
queue_gpu = clCreateCommandQueue(context, devices[0], 0, &err);
queue_cpu = clCreateCommandQueue(context, devices[1], 0, &err);
```

Multiple Device Programming In Pipeline Manner Example

Multiple devices working in a cooperative manner on the same data such that the CPU queue will wait until the GPU kernel is finished.



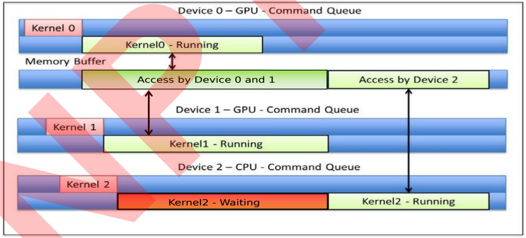
(Figure from reference[2])

Multiple Device Programming In Pipeline Manner Example

```
// A pipelined model of multidevice execution with single context
// The enqueued kernel on the GPU command queue waits for the kernel on the
    CPU command queue to finish executing
cl_event event_cpu, event_gpu;

// Starts as soon as enqueued
err = clEnqueueNDRangeKernel(queue_gpu,kernel0,2,NULL,global,local,0,NULL,&
    event_gpu);

// Starts after event_gpu is on CL_COMPLETE
err = clEnqueueNDRangeKernel(queue_cpu,kernel1,2,NULL,global,local,1,&
    event_gpu,&event_cpu);

//clFlush only guarantees that all queued commands to command_queue will
    eventually be submitted to the appropriate device. There is no guarantee
    that they will be complete after clFlush returns
clFlush(queue_cpu);
clFlush(queue_gpu);
```

Multiple Device Programming Work Concurrently and Pipeline Manner Example

- Multiple devices working in a parallel manner where both GPUs do not use the same buffers and will execute independently.
- The CPU queue will wait until both GPU devices are finished



(Figure from reference[2])

Multiple Device Programming In Concurrent Manner Example

```
// A concurrent and pipelined model of multidevice execution with single
    context on a single platform having 2 GPU and 1 CPU device
// Create 3 command queues, 2 queues for the 2 GPUs and 1 queue for the CPU
// The enqueued kernel on the CPU command queue waits for the kernels on the
    GPU command queues to finish
// Both the GPU devices can execute concurrently as soon as they have their
    respective data as they have no events in their waitlist

cl_event event_gpu[2];
err = clEnqueueNDRangeKernel(queue_gpu_0,kernel0,2,NULL,global,local,0,NULL,&
    event_gpu[0]);
err = clEnqueueNDRangeKernel(queue_gpu_1,kernel1,2,NULL,global,local,0,NULL,&
    event_gpu[1]);
// CPU will wait till both GPUs are done executing their kernels
err = clEnqueueNDRangeKernel(queue_cpu,kernel2,2,NULL,global,local,2,event_gpu
    ,NULL);

clFlush(queue_gpu_0);
clFlush(queue_gpu_1);
clFlush(queue_cpu);
```

Multiple Command Queues With Different Contexts In Different Devices

- Context is created with respect to a particular platform
- For different devices from different platform, we create multiple contexts
- For separate contexts created for different devices, synchronization using events would not be possible
- Only way to share data between devices would be to use clFinish and then explicitly copy data in and out of a given context and across contexts via host memory space

Creating Multiple Command Queues With Different Contexts In Different Devices Example

```
cl_uint numPlatforms;
cl_uint numDevices;
cl_device_id * deviceIDs;
size_t size;
clGetPlatformIDs(0, NULL, &numPlatforms);
cl_platform_id  platformIDs[numPlatforms];
cl_context contexts[numPlatforms][16]= {0};;
cl_command_queue commands[numPlatforms][16]= {0};;
clGetPlatformIDs(numPlatforms, platformIDs, NULL);
```

Creating Multiple Command Queues With Different Contexts In Different Devices Example

```
for(int p=0; p < numPlatforms; p++)
{
  errNum = clGetDeviceIDs(platformIDs[i],CL_DEVICE_TYPE_ALL,0,NULL,&numDevices
      );
  deviceIDs = (cl_device_id *)malloc(sizeof(cl_device_id)*numDevices);
  errNum = clGetDeviceIDs(platformIDs[i],CL_DEVICE_TYPE_ALL,numDevices,&
      deviceIDs,NULL);

  for(int d=0; d < numDevices; d++)
  {
    contexts[d] = clCreateContext(NULL, 1, deviceIDs[d], NULL, NULL, &err);
    commands[d] = clCreateCommandQueue(contexts[d],deviceIDs[d],0,0);
  }
}
```

Multiple Device Programming In Pipeline Manner Example

```
//Two contexts for two devices, each having its own command_queue
//There is a dependency between kernel 1 and kernel 2, output of kernel1 is
    used as input to kernel2
//Kernel1 is assigned to CPU and kernel 2 is assigned to GPU
...
clEnqueueWriteBuffer(queue_cpu,bufferA,CL_TRUE,0,10*sizeof(int),h_a,0,NULL,&
    writeEventA);
clEnqueueWriteBuffer(queue_cpu,bufferB,CL_TRUE,0,10*sizeof(int),h_b,0,NULL,&
    writeEventB);
kernelEvent[0]=writeEventA;
kernelEvent[1]=writeEventB;
clEnqueueNDRangeKernel(queue_cpu,kernel1,1,NULL,globalws,localws,2,eventList,&
    kernelEvent);
clEnqueueReadBuffer(queue_cpu,bufferC,CL_TRUE,0,10*sizeof(int),h_c,1,&
    kernelEvent,NULL);
clFinish(queue_cpu);
//Blocks until all previously queued OpenCL commands in a command-queue are
    issued to the associated device and have completed.
```

Multiple Device Programming In Pipeline Manner Example

```
...
clEnqueueWriteBuffer(queue_gpu,bufferD,CL_TRUE,0,10*sizeof(int),h_c,0,NULL,&
    writeEventA);
clEnqueueNDRangeKernel(queue_gpu,kernel2,1,NULL,globalws,localws,2,eventList,&
    kernelEvent);
clEnqueueReadBuffer(queue_gpu,bufferOut,CL_TRUE,0,10*sizeof(int),h_out,1,&
    kernelEvent,&readEvent);
clFinish(queue_gpu);
...
```

Multiple Device Programming In Concurrent Manner Example

```
//Two contexts for two devices, each having its own command_queue
// There is no dependency between the two kernels and can execute concurrently
...
err = clEnqueueNDRangeKernel(queue_gpu,kernel1,2,NULL,global,local,0,NULL,NULL
    );
err = clEnqueueNDRangeKernel(queue_cpu,kernel2,2,NULL,global,local,0,NULL,NULL
    );
...
```

Concurrent Kernel Execution

- Concurrency is property of a system in which a set of tasks in a system can remain active and make progress at the same time
- Programmers need to identify the concurrency in their problem and efficiently schedule in the host program
- The concurrent tasks can be running-
  - Different kernels from different independent applications
  - Different kernels without dependency between them from same application
  - Partitioned instances of same kernel that are SIMD in nature

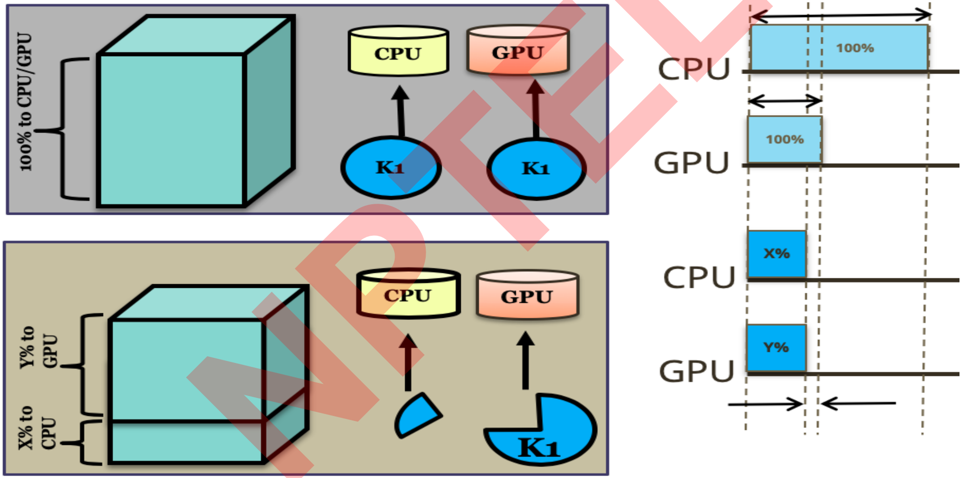
executing

- Multiple applications can executing concurrently across multiple devices
- Heterogeneous computing can efficiently exploit both CPU and GPU devices by invoking OpenCL's data transfer APIs, query memory objects, and data/work partitioning between the multiple devices
- Technique is used to partition the workload of a single kernel across multiple available OpenCL devices

Partitioning

- Some kernels performs better on either CPU or GPU devices
- While executing a kernel on a specific device the other available devices may remain idle
- Partitioning is a technique used to partition the data efficiently and distribute them across multiple OpenCL devices
- Then launch same kernel with partitioned data across multiple OpenCL devices
- The partitioned kernels can run concurrently on different devices reducing the total execution time

Partitioning



Manual Partitioning Using Example

- Vector addition of two vectors of size LENGTH
- Partition across 1 CPU device and 1 GPU on 2 seperate contexts
- Partitioning across CPU and GPU (20% to CPU and 80% to GPU)

Manual Partitioning Using Example

Initialisation and declaration host data

```
...
size_t dataSize = sizeof ( float ) * LENGTH ; //LENGTH is size of vector
float * h_a = ( float *) malloc ( dataSize ); // a vector
float * h_b = ( float *) malloc ( dataSize ); // b vector
float * h_c = ( float *) malloc ( dataSize ); // c vector ( result )
cl_int err ; // error code returned from OpenCL call
// Fill vectors a and b with random float values
int i = 0;
for ( int i = 0; i < LENGTH ; i ++) {
  h_a [i] = rand () / ( float ) RAND_MAX ;
  h_b [i] = rand () / ( float ) RAND_MAX ;
}
...
```

Manual Partitioning Using Example

Create buffer object for CPU

```
...
//Partitioning the vector across CPU and GPU
size_t dataSize_CPU = sizeof(float)*(LENGTH)*(20/100) ;
size_t dataSize_GPU = sizeof(float)*(LENGTH)*(80/100) ;
// Create array in CPU memory
cl_mem d_a_CPU  = clCreateBuffer(context_CPU,  CL_MEM_READ_ONLY |
    CL_MEM_COPY_HOST_PTR,  dataSize_CPU, h_a, &err);
cl_mem d_b_CPU  = clCreateBuffer(context_CPU,  CL_MEM_READ_ONLY |
    CL_MEM_COPY_HOST_PTR,  dataSize_CPU, h_b, &err);
cl_mem d_c_CPU  = clCreateBuffer(context_CPU,  CL_MEM_READ_WRITE,
    dataSize_CPU, NULL, &err);
```

Manual Partitioning Example

Create buffer object for GPU

```
// Create array in GPU memory
cl_mem d_a_GPU  = clCreateBuffer(context_GPU,  CL_MEM_READ_ONLY |
    CL_MEM_COPY_HOST_PTR,  dataSize_GPU, h_a, &err);
cl_mem d_b_GPU  = clCreateBuffer(context_GPU,  CL_MEM_READ_ONLY |
    CL_MEM_COPY_HOST_PTR,  dataSize_GPU, h_b, &err);
cl_mem d_c_GPU  = clCreateBuffer(context_GPU,  CL_MEM_READ_WRITE,
    dataSize_GPU, NULL, &err);
...
```

Manual Partitioning Example

Writing data from host to device

```
...
//Write the data from host to the CPU device
err = clEnqueueWriteBuffer( commands_CPU, d_a_CPU, CL_TRUE, 0, sizeof(float)
     * dataSize_CPU, h_a, 0, NULL, NULL );
err = clEnqueueWriteBuffer( commands_CPU, d_b_CPU, CL_TRUE, 0, sizeof(float)
     * dataSize_CPU, h_b, 0, NULL, NULL );

//Write the data from host to the GPU device
err = clEnqueueWriteBuffer( commands_GPU, d_a_GPU, CL_TRUE, 0, sizeof(float)
     * dataSize_GPU, h_a+dataSize_CPU, 0, NULL, NULL );
err = clEnqueueWriteBuffer( commands_GPU, d_b_GPU, CL_TRUE, dataSize_CPU,
    sizeof(float) * dataSize_GPU, h_b+dataSize_GPU, 0, NULL, NULL );
...
```

Manual Partitioning Example

Executing and reading output from device to host

```
...
size_t  global_work_size_CPU = dataSize_CPU;
size_t  global_work_size_GPU = dataSize_GPU;
size_t local_work_size=512;
err = clEnqueueNDRangeKernel(commands_CPU, ko_vadd, 1, NULL, &
    global_work_size_CPU , &local_work_size, 0, NULL, NULL);
err = clEnqueueNDRangeKernel(commands_GPU, ko_vadd, 1, NULL, &
    global_work_size_GPU , &local_work_size, 0, NULL, NULL);

 err = clEnqueueReadBuffer( commands_CPU, d_c_CPU, CL_TRUE, 0, sizeof(float)
      * dataSize_CPU, h_c, 0, NULL, NULL );
 err = clEnqueueReadBuffer( commands_GPU, d_c_GPU, CL_TRUE, dataSize_CPU,
     sizeof(float) * dataSize_GPU, h_c+dataSize_CPU, 0, NULL, NULL );
 ...
```

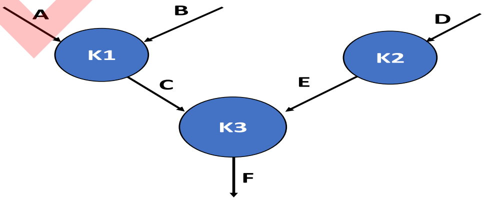
Manual Partitioning Example

Checking output

```
//Synchronize
clFlush(commands_CPU);
clFlush(commands_CPU);

// Test the results
int correct = 0;
for(int i = 0; i < count; i++)
  if(h[c]==h_a[i] + h_b[i])
    correct++;

printf("Vector add to find C = A+B:  %d out of %d results were correct.\n",
    correct, count);
...
```

Application: DAG Scheduling

- K1 performs matrix multiplication
- K2 squares each element of input vector
- K3 performs matrix-vector multiplication
- Target platform has one CPU and two GPU devices from same vendor



DAG Scheduling Example

- Create one context for the platform
- Create three devices each having its own command queue
- Enque write, ndRange and read for kernel K1 and K2 to two different devices(choose suitably) that will run concurrently
- Synchronise untill both kernels finish execution
- Partition the data across the two GPU devices and launch two kernels $K3_1$ and $K3_2$ concurrently
- Enque write, ndRange and read for kernel $K3_1$ and $K3_2$ for both of these two devices
- Synchronise again and check the result

Heterogeneous Computing: Factors To Consider

- Scheduling overhead
- Location of data: data currently resident on which device
- Granularity of workloads: How to divide the problem
- Execution performance relative to other devices

Device Fission

- The ability for some devices to be divided into smaller subdevices
- Device Fission is supported only on CPU-like devices
- It is possible to use Device Fission to build a portable and powerful threading application based on task parallelism

OpenCL Sub-Devices

We can create sub-devices partitioning an OpenCL device. The API used is **clCreateSubDevices**

- Creates an array of sub-devices; each referencing a non-intersecting set of compute units within the device, according to given partition scheme. Options:
  - CL_DEVICE_PARTITION_EQUALLY
  - CL_DEVICE_PARTITION_BY_COUNTS
  - CL_DEVICE_PARTITION_BY_AFFINITY_DOMAIN
- The output sub-devices may be used in every way that the parent device can be used, including creating contexts, building programs, further calls to clCreateSubDevices and creating command-queues.
- When a command-queue is created against a sub-device, the commands enqueued on the queue are executed only on the sub-device.

Creating OpenCL Sub-Devices Example

Creates an array of sub-devices, each referencing a non-intersecting set of compute units within given CPU device.

```
#define NUM_CUS 8
...
cl_device_partition_property subDeviceProperties[] = {
    CL_DEVICE_PARTITION_EQUALLY, NumOfCUperSubdevice, 0 };
clCreateSubDevices(device_id,subDeviceProperties,NUM_CUS/NumOfCUperSubdevice,&
    subDevices,NULL);
//cl_int clCreateSubDevices(cl_device_id in_device, const
    cl_device_partition_property *properties, cl_uint num_devices,
    cl_device_id *out_devices, cl_uint *num_devices_ret )
//Other partition properties are CL_DEVICE_PARTITION_BY_COUNTS and
    CL_DEVICE_PARTITION_BY_AFFINITY_DOMAIN
...
cl_command_queue commands[NUM_CUS/NumOfCUperSubdevice];
for (int i = 0; i < NUM_CUS/NumOfCUperSubdevice; i++)
 commands[i] = clCreateCommandQueue(context, subDevices[i], 0, &err);
...
```

Concurrency and CUDA

- Applications must execute functions concurrently on multiple processors so that available processors in the system can be efficiently used for heterogeneous computing
- CUDA Applications manage concurrency by executing asynchronous commands in streams, sequences of commands that execute in order
- Different streams may execute their commands concurrently or out of order with respect to each other.

CUDA Streams

- A CUDA stream refers to a sequence of asynchronous CUDA operations that execute on a device in the order issued by the host code.
- These operations can include host-device data transfer, kernel launches, and most other commands that are issued by the host but handled by the device.
- The execution of an operation in a stream is always asynchronous with respect to the host.
- It is the programmer's responsibility to use CUDA APIs to ensure an asynchronous operation has completed before using the result.

CUDA Streams

All CUDA operations (both kernels and data transfers) either explicitly or implicitly run in a stream. There are two types of streams:

- Implicitly declared stream (NULL stream)
- Explicitly declared stream (non-NULL stream)

The NULL stream is the default stream that kernel launches and data transfers use if you do not explicitly specify a stream. All CUDA examples discussed previously used the NULL or default stream.

Asynchronous API

```
// Basic stream operations
cudaStream_t stream;
cudaError_t cudaStreamCreate(cudaStream_t stream);
cudaError_t cudaStreamDestroy(cudaStream_t stream);
cudaError_t cudaStreamSynchronize(cudaStream_t stream);
//Blocks host until stream has completed all operations.
cudaError_t cudaStreamQuery(cudaStream_t stream);
// Queries an asynchronous stream for completion status.

// Asynchronous memory operations
cudaError_t cudaMemcpyAsync(void* dst, const void* src, size_t count,
    cudaMemcpyKind kind, cudaStream_t stream = 0);
// cudaMemcpyKind is an enum type with values: cudaMemcpyHostToHost,
    cudaMemcpyHostToDevice, cudaMemcpyDeviceToHost, cudaMemcpyDeviceToDevice,


//Pinning memory on the host is required for asynchronous data transfers
cudaError_t cudaMallocHost(void **ptr, size_t size);
// Launching Kernel
kernel_name<<<grid, block, sharedMemSize, stream>>>(argument list);
```

CUDA Streams: Basic Example

```
__global__ void kernel(float *g_data, float value)
{
  int idx = blockIdx.x * blockDim.x + threadIdx.x;
  g_data[idx] = g_data[idx] + value;
}
#define CHECK(call)
{
  const cudaError_t error = call;
  if (error != cudaSuccess)
  {
    fprintf(stderr, "Error: %s:%d, ", __FILE__, __LINE__);
    fprintf(stderr, "code: %d, reason: %s\n", error,
    cudaGetErrorString(error));
  }
}
```

CUDA Streams: Basic Example

```
int main(int argc, char *argv[])
{
  int devID = 0;
  cudaDeviceProp deviceProps;
  CHECK(cudaGetDeviceProperties(&deviceProps, devID));
  printf("> %s running on", argv[0]);
  printf(" CUDA device [%s]\n", deviceProps.name);
  int num = 1 << 24;
  int nbytes = num * sizeof(int);
  float value = 10.0f;
  // allocate host memory
  float *h_a = 0;
  CHECK(cudaMallocHost((void **)&h_a, nbytes));
  memset(h_a, 0, nbytes);
  // allocate device memory
  float *d_a = 0;
  CHECK(cudaMalloc((void **)&d_a, nbytes));
  CHECK(cudaMemset(d_a, 255, nbytes));
```

CUDA Streams: Basic Example

```
// set kernel launch configuration
  dim3 block = dim3(512);
  dim3 grid  = dim3((num + block.x - 1) / block.x);
  // create cuda event handles
  cudaEvent_t stop;
  CHECK(cudaEventCreate(&stop));
  // asynchronously issue work to the GPU (all to stream 0)
  CHECK(cudaMemcpyAsync(d_a, h_a, nbytes, cudaMemcpyHostToDevice));
  kernel<<<grid, block>>>(d_a, value);
  CHECK(cudaMemcpyAsync(h_a, d_a, nbytes, cudaMemcpyDeviceToHost));
  CHECK(cudaEventRecord(stop));
  // have CPU do some work while waiting for stage 1 to finish
  unsigned long int counter = 0;
  while (cudaEventQuery(stop) == cudaErrorNotReady) {
    counter++;
  }
```

CUDA Streams: Basic Example

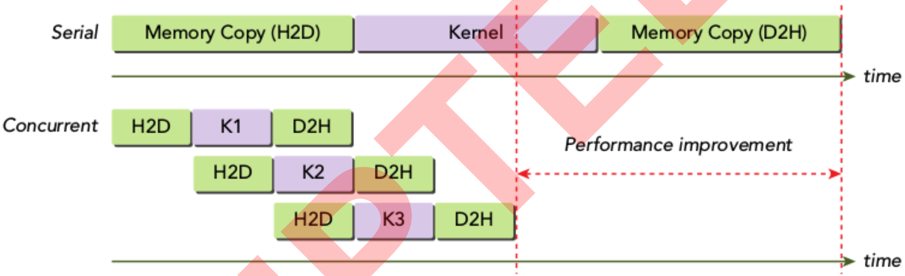
```
// print the cpu and gpu times
  printf("CPU executed %lu iterations while waiting for GPU to finish\n",
counter);
  // release resources
  CHECK(cudaEventDestroy(stop));
  CHECK(cudaFreeHost(h_a));
  CHECK(cudaFree(d_a));
  CHECK(cudaDeviceReset());
}
```

Exploiting Concurrency

Asynchronous, stream-based kernel launches and data transfers enable the following types of coarse-grain concurrency:

- Overlapped host computation and device computation
- Overlapped host computation and host-device data transfer
- Overlapped host-device data transfer and device computation
- Concurrent device computation

Concurrent Streams: Common Pattern



Data transfer operations are not executed concurrently, even though they are issued in separate streams. This contention is caused by a shared resource: the PCIe bus. Devices with a duplex PCIe bus can overlap two data transfers, but they must be in different streams and in different directions.

Synchronization

- Synchronizing the device `cudaDeviceSynchronize()` : wait until all streams finish
- Synchronizing a stream `cudaStreamSynchronize(stream)`
- Synchronizing an event in a stream `cudaStreamWaitEvent(stream, event)`
- Synchronizing across streams using an event `cudaEventSynchronize(event)`

Concurrency Example

```
int main(int argc, char **argv)
{
 printf("> %s Starting...\n", argv[0]);
 int dev = 0; cudaDeviceProp deviceProp;
 CHECK(cudaGetDeviceProperties(&deviceProp, dev));
 printf("> Using Device %d: %s\n", dev, deviceProp.name);
 CHECK(cudaSetDevice(dev));
 if(deviceProp.major < 3||(deviceProp.major==3&&deviceProp.minor<5)){
   if (deviceProp.concurrentKernels == 0)
     printf("> Concurrent Execution not supported. CUDA kernel runs will be
         serialized\n");
   else
   printf("> GPU does not support HyperQ. CUDA kernel runs will have limited
       concurrency\n\n");
 }
 printf("> Compute Capability %d.%d hardware with %d multi-processors\n",
 deviceProp.major, deviceProp.minor, deviceProp.multiProcessorCount);
```

Concurrency Example

```
// set up max connection
  char * iname = "CUDA_DEVICE_MAX_CONNECTIONS";
  setenv (iname, "1", 1);
  char *ivalue =  getenv (iname);
  printf ("> %s = %s\n", iname, ivalue);
  printf ("> with streams = %d\n", NSTREAM);
  // set up data size of vectors
  int nElem = 1 << 18;
  printf("> vector size = %d\n", nElem);
  size_t nBytes = nElem * sizeof(float);
  // malloc pinned host memory for async memcpy
  float *h_A, *h_B, *hostRef, *gpuRef;
  CHECK(cudaHostAlloc((void**)&h_A, nBytes, cudaHostAllocDefault));
  CHECK(cudaHostAlloc((void**)&h_B, nBytes, cudaHostAllocDefault));
  CHECK(cudaHostAlloc((void**)&gpuRef, nBytes, cudaHostAllocDefault));
  CHECK(cudaHostAlloc((void**)&hostRef, nBytes, cudaHostAllocDefault));
  // initialize data at host side
  initialData(h_A, nElem); initialData(h_B, nElem);
  memset(hostRef, 0, nBytes); memset(gpuRef,  0, nBytes);
```

Concurrency Example

```
// malloc device global memory
 float *d_A, *d_B, *d_C;
 CHECK(cudaMalloc((float**)&d_A, nBytes));
 CHECK(cudaMalloc((float**)&d_B, nBytes));
 CHECK(cudaMalloc((float**)&d_C, nBytes));
 cudaEvent_t start, stop;
 CHECK(cudaEventCreate(&start));
 CHECK(cudaEventCreate(&stop));
 // invoke kernel at host side
 dim3 block (BDIM);
 dim3 grid  ((nElem + block.x - 1) / block.x);
 printf("> grid (%d, %d) block (%d, %d)\n", grid.x, grid.y, block.x, block.y);
 // sequential operation
 CHECK(cudaEventRecord(start, 0));
 CHECK(cudaMemcpy(d_A, h_A, nBytes, cudaMemcpyHostToDevice));
 CHECK(cudaMemcpy(d_B, h_B, nBytes, cudaMemcpyHostToDevice));
 CHECK(cudaEventRecord(stop, 0));
 CHECK(cudaEventSynchronize(stop));
 float memcpy_h2d_time;
 CHECK(cudaEventElapsedTime(&memcpy_h2d_time, start, stop));
```

Concurrency Example

```
CHECK(cudaEventRecord(start, 0));
sumArrays<<<grid, block>>>(d_A, d_B, d_C, nElem);
CHECK(cudaEventRecord(stop, 0));  CHECK(cudaEventSynchronize(stop));
float kernel_time;
CHECK(cudaEventElapsedTime(&kernel_time, start, stop));
CHECK(cudaEventRecord(start, 0));
CHECK(cudaMemcpy(gpuRef, d_C, nBytes, cudaMemcpyDeviceToHost));
CHECK(cudaEventRecord(stop, 0));  CHECK(cudaEventSynchronize(stop));
float memcpy_d2h_time;
CHECK(cudaEventElapsedTime(&memcpy_d2h_time, start, stop));
float itotal = kernel_time + memcpy_h2d_time + memcpy_d2h_time;
printf("Measured timings (throughput):\n");
printf(" Memcpy host to device\t: %f ms (%f GB/s)\n",
memcpy_h2d_time, (nBytes * 1e-6) / memcpy_h2d_time);
printf(" Memcpy device to host\t: %f ms (%f GB/s)\n",
memcpy_d2h_time, (nBytes * 1e-6) / memcpy_d2h_time);
printf(" Kernel\t\t\t: %f ms (%f GB/s)\n",
kernel_time, (nBytes * 2e-6) / kernel_time);
printf(" Total\t\t\t: %f ms (%f GB/s)\n",
itotal, (nBytes * 2e-6) / itotal);
```

Concurrency Example

```
// grid parallel operation
 cudaStream_t stream[NSTREAM]; int iElem = nElem / NSTREAM;
 size_t iBytes = iElem * sizeof(float);
 grid.x = (iElem + block.x - 1) / block.x;
 for (int i = 0; i < NSTREAM; ++i)
   CHECK(cudaStreamCreate(&stream[i]));
 CHECK(cudaEventRecord(start, 0));
 // initiate all work on the device asynchronously in depth-first order
 for (int i = 0; i < NSTREAM; ++i){
   int ioffset = i * iElem;
   CHECK(cudaMemcpyAsync(&d_A[ioffset], &h_A[ioffset], iBytes,
       cudaMemcpyHostToDevice, stream[i]));
   CHECK(cudaMemcpyAsync(&d_B[ioffset], &h_B[ioffset], iBytes,
       cudaMemcpyHostToDevice, stream[i]));
   sumArrays<<<grid, block, 0, stream[i]>>>(&d_A[ioffset], &d_B[ioffset], &d_C
       [ioffset], iElem);
   CHECK(cudaMemcpyAsync(&gpuRef[ioffset], &d_C[ioffset], iBytes,
       cudaMemcpyDeviceToHost, stream[i]));
 }
 CHECK(cudaEventRecord(stop, 0));
```

Concurrency Example

```
CHECK(cudaEventSynchronize(stop)); float execution_time;
CHECK(cudaEventElapsedTime(&execution_time, start, stop));
printf("Actual results from overlapped data transfers:\n");
printf("overlap with %d streams : %f ms (%f GB/s)\n", NSTREAM, execution_time
    , (nBytes * 2e-6) / execution_time );
printf("speedup : %f \n", ((itotal - execution_time) * 100.0f) / itotal);
CHECK(cudaGetLastError());
// free device global memory
CHECK(cudaFree(d_A)); CHECK(cudaFree(d_B)); CHECK(cudaFree(d_C));
// free host memory
CHECK(cudaFreeHost(h_A)); CHECK(cudaFreeHost(h_B)); CHECK(cudaFreeHost(
    hostRef)); CHECK(cudaFreeHost(gpuRef));
// destroy events
CHECK(cudaEventDestroy(start)); CHECK(cudaEventDestroy(stop));
// destroy streams
for (int i = 0; i < NSTREAM; ++i)
 CHECK(cudaStreamDestroy(stream[i]));
CHECK(cudaDeviceReset()); return(0);
}
```

Concurrency Example

```
> ./simpleMultiAddDepth Starting...
> Using Device 0: Tesla K40m
> Compute Capability 3.5 hardware with 15 multi-processors
> CUDA_DEVICE_MAX_CONNECTIONS = 32
> with streams = 4
> vector size = 262144
> grid (2048, 1) block (128, 1)

Measured timings (throughput):
Memcpy host to device    : 0.397920 ms (2.635143 GB/s)
Memcpy device to host    : 0.180288 ms (5.816116 GB/s)
Kernel                   : 1595.653687 ms (0.001314 GB/s)
Total                    : 1596.231934 ms (0.001314 GB/s)

Actual results from overlapped data transfers:
overlap with 4 streams : 401.155762 ms (0.005228 GB/s)
speedup                : 74.868576
```

Breadth First Order

```
// initiate all asynchronous transfers to the device
for (int i = 0; i < NSTREAM; ++i){
  int ioffset = i * iElem;
  CHECK(cudaMemcpyAsync(&d_A[ioffset], &h_A[ioffset], iBytes,
      cudaMemcpyHostToDevice, stream[i]));
  CHECK(cudaMemcpyAsync(&d_B[ioffset], &h_B[ioffset], iBytes,
      cudaMemcpyHostToDevice, stream[i]));
}
// launch a kernel in each stream
for (int i = 0; i < NSTREAM; ++i){
  int ioffset = i * iElem;
   sumArrays<<<grid, block, 0, stream[i]>>>(&d_A[ioffset], &d_B[ioffset], &d_C
       [ioffset], iElem);
}
// enqueue asynchronous transfers from the device
for (int i = 0; i < NSTREAM; ++i){
  int ioffset = i * iElem;
  CHECK(cudaMemcpyAsync(&gpuRef[ioffset], &d_C[ioffset], iBytes,
      cudaMemcpyDeviceToHost, stream[i]));
}
```

Concurrency Example

```
> ./simpleMultiAddBreadth Starting...
> Using Device 0: Tesla K40m
> Compute Capability 3.5 hardware with 15 multi-processors
> CUDA_DEVICE_MAX_CONNECTIONS = 1
> with streams = 4
> vector size = 262144
> grid (2048, 1) block (128, 1)

Measured timings (throughput):
Memcpy host to device    : 0.383424 ms (2.734769 GB/s)
Memcpy device to host    : 0.182272 ms (5.752809 GB/s)
Kernel                   : 1605.997192 ms (0.001306 GB/s)
Total                    : 1606.562866 ms (0.001305 GB/s)

Actual results from overlapped data transfers:
overlap with 4 streams : 402.014435 ms (0.005217 GB/s)
speedup                : 74.976738
```

References

- Khronos OpenCL Working Group. *The OpenCL Specification Version-2.1*. 2018 February 13,
- Kaeli DR, Mistry P, Schaa D, Zhang DP. *Heterogeneous computing with OpenCL 2.0*. Morgan Kaufmann; 2015 Jun 18.
- John Cheng, Max Grossman, Ty Mckercher *Professional CUDA C Programming*