# GPU Computing and Accelerators: Part III

# Compute Unified Device Architecture (CUDA)

## Compute Capabilities

<table>
<tr><th>Feature Support</th><th colspan="7">Compute Capability</th></tr>
<tr><th>(Features differently supported)</th><th>1.0</th><th>1.1</th><th>1.2</th><th>1.3</th><th>2.x</th><th>3.0</th><th>3.5</th></tr>
<tr><td>Atomic functions on 32-bit integer values in global memory</td><td>No</td><td colspan="6">Yes</td></tr>
<tr><td>atomicExch() on 32-bit floating point values in global memory</td><td colspan="2">No</td><td colspan="5">Yes</td></tr>
<tr><td>Atomic functions on 32-bit integer values in shared memory</td><td colspan="2">No</td><td colspan="5">Yes</td></tr>
<tr><td>atomicExch() on 32-bit floating point values in shared memory</td><td colspan="2">No</td><td colspan="5">Yes</td></tr>
<tr><td>Atomic functions on 64-bit integer values in global memory</td><td colspan="2">No</td><td colspan="5">Yes</td></tr>
<tr><td>Warp vote functions</td><td colspan="2">No</td><td colspan="5">Yes</td></tr>
<tr><td>Double-precision floating-point numbers</td><td colspan="3">No</td><td colspan="4">Yes</td></tr>
<tr><td>Atomic functions operating on 64-bit integer values in shared memory</td><td colspan="4">No</td><td colspan="3">Yes</td></tr>
<tr><td>Atomic addition operating on 32-bit floating point values in global and shared memory</td><td colspan="4">No</td><td colspan="3">Yes</td></tr>
<tr><td>__ballot() (Warp Vote Functions)</td><td colspan="4">No</td><td colspan="3">Yes</td></tr>
<tr><td>__threadfence_system()</td><td colspan="4">No</td><td colspan="3">Yes</td></tr>
<tr><td>__syncthreads_count()</td><td colspan="4">No</td><td colspan="3">Yes</td></tr>
<tr><td>__syncthreads_and()</td><td colspan="4">No</td><td colspan="3">Yes</td></tr>
<tr><td>__syncthreads_or()</td><td colspan="4">No</td><td colspan="3">Yes</td></tr>
<tr><td>Surface functions</td><td colspan="4">No</td><td colspan="3">Yes</td></tr>
<tr><td>3D grid of thread blocks</td><td colspan="4">No</td><td colspan="3">Yes</td></tr>
<tr><td>Funnel shift (see reference manual)</td><td colspan="6">No</td><td>Yes</td></tr>
</table>

Table: Compute Capabilities: Features by Compute Capability Version (from CUDA C Programming Guide version 5.0)

| | Compute Capability | | | | | | |
|---|---|---|---|---|---|---|---|
| Technical Specifications | 1.0 | 1.1 | 1.2 | 1.3 | 2.x | 3.0 | 3.5 |
| Maximum dimensionality of grid of thread blocks | 2 | | | | 3 | | |
| Maximum x-dimension of a grid of thread blocks | $65535 = 2^{16} - 1$ | | | | | $2^{31} - 1$ | |
| Maximum y- or z-dimension of a grid of thread blocks | 65535 | | | | | | |
| Maximum dimensionality of thread block | 3 | | | | | | |
| Maximum x- or y-dimension of a block | 512 | | | | 1024 | | |
| Maximum z-dimension of a block | 64 | | | | | | |
| Maximum number of threads per block | 512 | | | | 1024 | | |
| Warp size | 32 | | | | | | |
| Maximum number of resident blocks per multiprocessor | 8 | | | | | 16 | |
| Maximum number of resident warps per multiprocessor | 24 | | 32 | | 48 | 64 | |
| Maximum number of resident threads per multiprocessor | 768 | | 1024 | | 1536 | 2048 | |
| Number of 32-bit registers per multiprocessor | 8 K | | 16 K | | 32 K | 64 K | |
| Maximum number of 32-bit registers per thread | 128 | | | | 63 | | 255 |
| Maximum amount of shared memory per multiprocessor | 16 KB | | | | 48 KB | | |
| Number of shared memory banks | 16 | | | | 32 | | |
| Amount of local memory per thread | 16 KB | | | | 512 KB | | |
| Constant memory size | 64 KB | | | | | | |
| Cache working set per multiprocessor for constant memory | 8 KB | | | | | | |
| Cache working set per multiprocessor for texture memory | Device dependent, between 6 KB and 8 KB | | | | | | |
| Maximum number of instructions per kernel | 2 million | | | | 512 million | | |

Table: Compute Capabilities: Selected Technical Specifications (from CUDA C Programming Guide version 5.0)

## Compute capabilities 1.3

We have learned from Table 3 that double precision floating point numbers have been added in Version 1.3 of the CUDA compute capabilities. It additionally provides a fused multiply add operation merging multiplication and addition to be faster and more accurate, but non IEEE 754 compliant.

## Compute Capabilities 2.0 and above

Compute capabilities 2.0 introduces IEEE 754 compliance for most parts of the standard as the default. The compiler switches `-ftz=false|true`, `-prec-div=true|false`, `-prec-sqrt=true|false` influence IEEE compliance of the computation. If the second option is used everywhere one switches to fast mode. The first options are the default though.

## IEEE 754 Rounding Modes

IEEE 754 defines four rounding modes

- round to nearest,
- round towards zero,
- round towards $+\infty$,
- round towards $-\infty$,

all of which are supported by CUDA. However in contrast to x86 CPUs where they can be dynamically switched, CUDA uses them statically.

Compiler intrinsics can be used to change the mode for individual operations, though.

## Main Differences to x86 CPUs

- no dynamical control of rounding modes

## Main Differences to x86 CPUs

- no dynamical control of rounding modes
- floating point exceptions not handled (especially all `NaN`s are silent)

### Main Differences to x86 CPUs

- no dynamical control of rounding modes
- floating point exceptions not handled (especially all `NaN`s are silent)
- no status flags indicating the exceptions exist

## Local versus Remote memory

Viewing from the host perspective, the device memory is remote memory that can only be accessed via the comparably slow system bus.

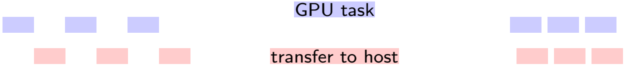
Looking at things from the device perspective the same hold for the hosts memory. Going even further, already the device memory may be considered slow from the view of the streaming multiprocessors. The local memory of the multiprocessors should be used to implement a user controlled cache.

## Consequences for CUDA Programs

- Keep data movements between device and host as little as possible

## Consequences for CUDA Programs

- Keep data movements between device and host as little as possible
- If they are necessary, try to overlap communication and computations




(a) bad pattern causing waiting times due to communication.

(b) good pattern.

Figure: Execution patterns for CUDA programs

### Consequences for CUDA Programs

- Keep data movements between device and host as little as possible
- If they are necessary, try to overlap communication and computations
- Make use of multiprocessors local shared memory to cache buffer kernel operations and avoid frequent access to global device memory

# Compute Unified Device Architecture (CUDA)

## Data Communication Issues

### Example

`../Material/CUDAbyExample/chapter05/dot.cu`

**Note:**

- automatic scaling of `blocksPerGrid`
- usage of local shared buffer `cache`
- synchronization in reduction block

# Compute Unified Device Architecture (CUDA)

## The CUDA Application Programmers Interface

We ave seen some elements of the CUDA API in the examples before:

- qualifiers: `__global__`, `__device__`, `__host__`, `__shared__`, `__constant__`

# Compute Unified Device Architecture (CUDA)

## The CUDA Application Programmers Interface

We ave seen some elements of the CUDA API in the examples before:

- qualifiers: `__global__`, `__device__`, `__host__`, `__shared__`, `__constant__`
- launch size specifiers `<<<grid, block size>>>`

# Compute Unified Device Architecture (CUDA)

## The CUDA Application Programmers Interface

We ave seen some elements of the CUDA API in the examples before:

- qualifiers: `__global__`, `__device__`, `__host__`, `__shared__`, `__constant__`
- launch size specifiers `<<<grid, block size>>>`
- type `dim3`

# Compute Unified Device Architecture (CUDA)

## The CUDA Application Programmers Interface

We ave seen some elements of the CUDA API in the examples before:

- qualifiers: `__global__`, `__device__`, `__host__`, `__shared__`, `__constant__`
- launch size specifiers `<<<grid, block size>>>`
- type `dim3`
- predefined variables: `threadIdx.x`, `blockIdx.x`, `blockDim.x`, `gridDim.x`

# Compute Unified Device Architecture (CUDA)

## The CUDA Application Programmers Interface

We ave seen some elements of the CUDA API in the examples before:

- qualifiers: `__global__`, `__device__`, `__host__`, `__shared__`, `__constant__`
- launch size specifiers `<<<grid, block size>>>`
- type `dim3`
- predefined variables: `threadIdx.x`, `blockIdx.x`, `blockDim.x`, `gridDim.x`
- memory functions: `cudaMalloc()`, `cudaFree()`, `cudaMemcpy()`

We ave seen some elements of the CUDA API in the examples before:

- qualifiers: `__global__`, `__device__`, `__host__`, `__shared__`, `__constant__`
- launch size specifiers `<<<grid, block size>>>`
- type `dim3`
- predefined variables: `threadIdx.x`, `blockIdx.x`, `blockDim.x`, `gridDim.x`
- memory functions: `cudaMalloc()`, `cudaFree()`, `cudaMemcpy()`
- thread synchronization mechanism: `__syncthreads();`

We ave seen some elements of the CUDA API in the examples before:

- qualifiers: `__global__`, `__device__`, `__host__`, `__shared__`, `__constant__`
- launch size specifiers `<<<grid, block size>>>`
- type `dim3`
- predefined variables: `threadIdx.x`, `blockIdx.x`, `blockDim.x`, `gridDim.x`
- memory functions: `cudaMalloc()`, `cudaFree()`, `cudaMemcpy()`
- thread synchronization mechanism: `__syncthreads();`

Some have been introduced earlier. For the others and a few more we will go into some more detail now.

- ```
  cudaError_t cudaFree ( void* devPtr )
  ```
  Frees the memory on the device that is refered to by `devPtr`.
  ```

# Compute Unified Device Architecture (CUDA)

## The CUDA Application Programmers Interface: Important Memory Operations

- ```
  cudaError_t cudaFree ( void* devPtr )
  ```
  Frees the memory on the device that is refered to by `devPtr`.

- ```
  cudaError_t cudaMalloc ( void** devPtr, size_t size )
  ```
  Allocate an amount corresponding to `size` of memory on the device and associate it to `devPtr`.

# Compute Unified Device Architecture (CUDA)

## The CUDA Application Programmers Interface: Important Memory Operations

- ```
  cudaError_t cudaFree ( void* devPtr )
  ```
  Frees the memory on the device that is refered to by `devPtr`.

- ```
  cudaError_t cudaMalloc ( void** devPtr, size_t size )
  ```
  Allocate an amount corresponding to `size` of memory on the device and associate it to `devPtr`.

- ```
  cudaError_t cudaMemcpy ( void* dst, const void* src, size_t count,
       cudaMemcpyKind kind )
  ```
  Copy data between host and device. `src` and `dst` represent the source and destination memory locations. The direction of operation is specified by `kind` and can be either `cudaMemcpyHostToDevice`, or `cudaMemcpyDeviceToHost`. The `count` argument is used to specify the number of data items to be copied.
  ```

# Compute Unified Device Architecture (CUDA)

## The CUDA Application Programmers Interface: Device Management Basics

- ```
  cudaError_t cudaGetDeviceCount ( int* count )
  ```
  Returns the number of compute-capable devices available in the system.
  ```

# Compute Unified Device Architecture (CUDA)

## The CUDA Application Programmers Interface: Device Management Basics

- ```
  cudaError_t cudaGetDeviceCount ( int* count )
  ```
  Returns the number of compute-capable devices available in the system.

- ```
  cudaError_t cudaChooseDevice ( int* device, const cudaDeviceProp*
       prop )
  ```
  Select compute-device which best matches criteria specified in `prop`. These can, e.g., be `int major`, `int minor` version numbers of the compute capabilities, or whether the chip is `int integrated` in the chipset or a plugged in device, but also simply the `char name[256]` of the device, and many more.

- ```
  cudaError_t cudaGetDevice ( int* device )
  ```
  Returns which device is currently being used.
  ```

# Compute Unified Device Architecture (CUDA)

## The CUDA Application Programmers Interface: Device Management Basics

- ```
  cudaError_t cudaGetDevice ( int* device )
  ```
  Returns which device is currently being used.
- ```
  cudaError_t cudaSetDevice ( int  device )
  ```
  Set device to be used for GPU executions

- ```
  cudaError_t cudaGetDevice ( int* device )
  ```
  Returns which device is currently being used.
- ```
  cudaError_t cudaSetDevice ( int  device )
  ```
  Set device to be used for GPU executions
- ```
  cudaError_t cudaDeviceSynchronize ( void )
  ```
  Wait for compute device to finish. If for the current device the synchronization flag `cudaDeviceScheduleBlockingSync` was set, the host thread will block until the device has finished its work.
  ```

- ```
  const __cudart_builtin__ char* cudaGetErrorString ( cudaError_t
      error )
  ```
  Returns the message string from the error code given in `error`.
  ```

# Compute Unified Device Architecture (CUDA)

## The CUDA Application Programmers Interface: Error Handling

- ```
  const __cudart_builtin__ char* cudaGetErrorString ( cudaError_t
      error )
  ```
  Returns the message string from the error code given in `error`.

- ```
  cudaError_t cudaGetLastError ( void )
  ```
  Returns the last error that has been produced by any of the runtime calls in the same host thread and resets it to `cudaSuccess`.

# Compute Unified Device Architecture (CUDA)

## The CUDA Application Programmers Interface: Error Handling

- ```
  const __cudart_builtin__ char* cudaGetErrorString ( cudaError_t
      error )
  ```
  Returns the message string from the error code given in `error`.

- ```
  cudaError_t cudaGetLastError ( void )
  ```
  Returns the last error that has been produced by any of the runtime calls in the same host thread and resets it to `cudaSuccess`.

- ```
  cudaError_t cudaPeekAtLastError ( void )
  ```
  As above but doe not reset the error code.
  ```

- ```
  cudaError_t cudaEventCreate ( cudaEvent_t* event )
  ```
  Creates, i.e., initializes the event object `event`.
  ```

- ```
  cudaError_t cudaEventCreate ( cudaEvent_t* event )
  ```
  Creates, i.e., initializes the event object `event`.

- ```
  cudaError_t cudaEventRecord ( cudaEvent_t event, cudaStream_t
       stream = 0 )
  ```
  Record `event`. The record may take some time so before evaluation it is recommended to use `cudaEventSynchronize()` to make sure it has terminated.

- ```
  cudaError_t cudaEventCreate ( cudaEvent_t* event )
  ```
  Creates, i.e., initializes the event object `event`.

- ```
  cudaError_t cudaEventRecord ( cudaEvent_t event, cudaStream_t
       stream = 0 )
  ```
  Record `event`. The record may take some time so before evaluation it is recommended to use `cudaEventSynchronize()` to make sure it has terminated.

- ```
  cudaError_t cudaEventSynchronize ( cudaEvent_t event )
  ```
  Wait until `event` has completed operations.
  ```

- ```
  cudaError_t cudaEventCreate ( cudaEvent_t* event )
  ```
  Creates, i.e., initializes the event object `event`.

- ```
  cudaError_t cudaEventRecord ( cudaEvent_t event, cudaStream_t
      stream = 0 )
  ```
  Record `event`. The record may take some time so before evaluation it is recommended to use `cudaEventSynchronize()` to make sure it has terminated.

- ```
  cudaError_t cudaEventSynchronize ( cudaEvent_t event )
  ```
  Wait until `event` has completed operations.

- ```
  cudaError_t cudaEventElapsedTime ( float* ms, cudaEvent_t start,
      cudaEvent_t end )
  ```
  Computes the elapsed time between two events (in milliseconds with a resolution of around 0.5 microseconds).

# Compute Unified Device Architecture (CUDA)

**The CUDA Application Programmers Interface: Events and Performance Measures**

## Example

A minimal performance measurement configuration:

```
cudaEvent_t start, stop;
cudaEventCreate(start);
cudaEventCreate(stop);
cudaEventRecord(start, 0);

// complete some tasks

cudaEventRecord(stop, 0);
cudaEventSynchronize(stop);

float etime;
cudaEventElapsedTime( &etime, start, stop);
```