



Denglin HammingTM V2

dICU Programming Guide

DL-DG/SW-032B-01

2023-6-09

Copyright©苏州登临科技有限公司，2019 - 2025，版权所有。

未经苏州登临科技有限公司事先书面同意，不得以任何形式或方式复制或传播本文件的任何部分。

商标和许可



和其它苏州登临科技有限公司的其它登临科技的图标为苏州登临科技有限公司的商标。本手册中提及的所有其他商标均为其各自所有者的财产。

通知

所购买的产品、服务和特性由苏州登临科技有限公司与客户签订的合同规定。本文件中描述的所有或部分产品、服务和特性可能不在采购范围或使用范围内。除非合同中另有规定，本文件中的所有声明、信息和建议均按“原样”提供，无任何明示或暗示的保证或陈述。

本手册中的信息如有更改，恕不另行通知。本文件在编制过程中已尽一切努力确保内容的准确性，本文件中的所有声明、信息和建议不构成任何明示或暗示的保证。

苏州登临科技有限公司

苏州工业园区扬富路11号南岸新地一期商务楼5号1101室，江苏，中国

<http://www.denglin.ai>

Email : support@denglin.ai

Change History

Version	Change description
02	Added notes in chapter <i>Heterogeneous Programming</i> .
01	Initial version.

Table of Contents

1 Introduction

2 Programming Model

2.1 Kernels

2.2 Thread Hierarchy

2.3 Memory Hierarchy

2.4 Heterogeneous Programming

3 Programming Interface

3.1 Compilation with dlcc

3.2 dICU C Runtime

3.2.1 Initialization

3.2.2 Device Memory

3.2.3 Shared Memory

3.2.4 Asynchronous Concurrent Execution

3.2.4.1 Concurrent Execution between Host and Device

3.2.4.2 Overlap of Data Transfer and Kernel Execution

3.2.4.3 Streams

3.2.4.3.1 Creation and Destruction

3.2.4.3.2 Default Stream

3.2.4.3.3 Explicit Synchronization

3.2.4.3.4 Implicit Synchronization

3.2.4.3.5 Callbacks

3.2.4.4 Graphs

3.2.4.4.1 Graph Structure

3.2.4.4.2 Creating a Graph Using Graph APIs

3.2.4.4.3 Creating a Graph Using Stream Capture

3.2.4.4.4 Cross-stream Dependencies and Events

3.2.4.4.5 Prohibited and Unhandled Operations

3.2.4.4.6 Invalidation

3.2.4.4.7 Using Graph APIs

3.2.4.5 Events

3.2.4.5.1 Creation and Destruction

3.2.4.5.2 Elapsed Time

3.2.4.6 Synchronous Calls

3.2.5 Surface Memory

3.2.5.1 Surface Object API

3.2.5.2 CUDA Arrays

4 dICU Driver API Reference

Appendix A. C++ Language Extensions

A.1 Surface Functions

A.1.1 Surface Object API

surf1Dread()

surf1Dwrite()

surf2Dread()

surf2Dwrite()

surf1DLayeredread()

surf1DLayeredwrite()

surf2DLayeredread()

surf2DLayeredwrite()

A.2 Printf Functions

Format Specifiers

[Limitations](#)

[Examples](#)

1 Introduction

Driven by the insatiable market demand for real-time, high-definition 3D graphics, the programmable Graphic Processor Unit or GPU has evolved into a highly parallel, multithreaded, manycore processor with tremendous computational horsepower and very high memory bandwidth.

The reason behind the discrepancy in floating-point capability between the CPU and the GPU is that the GPU is specialized for compute-intensive, highly parallel computation - exactly what graphics rendering is about - and therefore designed such that more transistors are devoted to data processing rather than data caching and flow control.

The GPU is especially well-suited to address problems that can be expressed as data-parallel computations - the same program is executed on many data elements in parallel - with high arithmetic intensity - the ratio of arithmetic operations to memory operations. Because the same program is executed for each data element, there is a lower requirement for sophisticated flow control, and because it is executed on many data elements and has high arithmetic intensity, the memory access latency can be hidden with calculations instead of big data caches.

Data-parallel processing maps data elements to parallel processing threads. Many applications that process large data sets can use a data-parallel programming model to speed up the computations. In 3D rendering, large sets of pixels and vertices are mapped to parallel threads. Similarly, image and media processing applications such as post-processing of rendered images, video encoding and decoding, image scaling, stereo vision, and pattern recognition can map image blocks and pixels to parallel processing threads. In fact, many algorithms outside the field of image rendering and processing are accelerated by data-parallel processing, from general signal processing or physics simulation to computational finance or computational biology.

dICU comes with a software environment that allows developers to use C as a high-level programming language.

2 Programming Model

This chapter introduces the main concepts behind the dICU programming model by outlining how they are exposed in C. An extensive description of dICU C is given in [Programming Interface](#).

Full code for the vector addition example used in this chapter and the next can be found in the vectorAdd dICU sample.

2.1 Kernels

dICU C extends C by allowing the programmer to define C functions, called kernels, that, when called, are executed N times in parallel by N different dICU threads, as opposed to only once like regular C functions.

A kernel is defined using the `__global__` declaration specifier and the number of dICU threads that execute that kernel for a given kernel call is specified using a new `<<<...>>>` execution configuration syntax. Each thread that executes the kernel is given a unique thread ID that is accessible within the kernel through the built-in `threadIdx` variable.

As an illustration, the following sample code adds two vectors A and B of size N and stores the result into vector C:

```
// kernel definition
__global__ void vecAdd(float* A, float* B, float* C)
{
    int i = threadIdx.x;
    C[i] = A[i] + B[i];
}

int main()
{
    ...
    // kernel invocation with N threads
    vecAdd<<<1, N>>>(A, B, C);
    ...
}
```

Here, each of the N threads that execute `VecAdd()` performs one pair-wise addition.

2.2 Thread Hierarchy

For convenience, `threadIdx` is a 3-component vector, so that threads can be identified using a one-dimensional, two-dimensional, or three-dimensional thread index, forming a one-dimensional, two-dimensional, or three-dimensional block of threads, called a thread block. This provides a natural way to invoke computation across the elements in a domain such as a vector, matrix, or volume.

The index of a thread and its thread ID relate to each other in a straightforward way:

- For a one-dimensional block, they are the same.
- For a two-dimensional block of size (Dx, Dy), the thread ID of a thread of index (x, y) is (x + y Dx).
- For a three-dimensional block of size (Dx, Dy, Dz), the thread ID of a thread of index (x, y, z) is (x + y Dx + z Dx Dy).

As an example, the following code adds two matrices A and B of size NxN and stores the result into matrix C:

```
// kernel definition
__global__ void MatAdd(float A[N][N], float B[N][N],
                      float C[N][N])
{
    int i = threadIdx.x;
    int j = threadIdx.y;
    C[i][j] = A[i][j] + B[i][j];
}

int main()
{
    ...
    // kernel invocation with one block of N * N * 1 threads
    int numBlocks = 1;
    dim3 threadsPerBlock(N, N);
    MatAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
    ...
}
```

There is a limit to the number of threads per block, since all threads of a block are expected to reside on the same processor core and must share the limited memory resources of that core. On current GPUs, a thread block may contain up to 1024 threads.

However, a kernel can be executed by multiple equally-shaped thread blocks, so that the total number of threads is equal to the number of threads per block times the number of blocks.

The number of threads per block and the number of blocks per grid specified in the `<<<...>>>` syntax can be of type `int` or `dim3`. Two-dimensional blocks or grids can be specified as in the example above.

Each block within the grid can be identified by a one-dimensional, two-dimensional, or three-dimensional index accessible within the kernel through the built-in `blockIdx` variable. The dimension of the thread block is accessible within the kernel through the built-in `blockDim` variable.

Extending the previous `MatAdd()` example to handle multiple blocks, the code becomes as follows.

```
// kernel definition
__global__ void MatAdd(float A[N][N], float B[N][N],
                      float C[N][N])
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int j = blockIdx.y * blockDim.y + threadIdx.y;
    if (i < N && j < N)
        C[i][j] = A[i][j] + B[i][j];
}

int main()
{
    ...
    // kernel invocation
    dim3 threadsPerBlock(16, 16);
    dim3 numBlocks(N / threadsPerBlock.x, N / threadsPerBlock.y);
    MatAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
    ...
}
```



```
}

```

A thread block size of 16x16 (256 threads), although arbitrary in this case, is a common choice. The grid is created with enough blocks to have one thread per matrix element as before. For simplicity, this example assumes that the number of threads per grid in each dimension is evenly divisible by the number of threads per block in that dimension, although that need not be the case.

Thread blocks are required to execute independently: It must be possible to execute them in any order, in parallel or in series.

Threads within a block can cooperate by sharing data through some shared memory and by synchronizing their execution to coordinate memory accesses. More precisely, one can specify synchronization points in the kernel by calling the `__syncthreads()` intrinsic function; `__syncthreads()` acts as a barrier at which all threads in the block must wait before any is allowed to proceed. [Shared Memory](#) gives an example of using shared memory.

For efficient cooperation, the shared memory is expected to be a low-latency memory near each processor core (much like an L1 cache) and `__syncthreads()` is expected to be lightweight.

2.3 Memory Hierarchy

dICU threads may access data from multiple memory spaces during their execution. Each thread block has shared memory visible to all threads of the block and with the same lifetime as the block. All threads have access to the same global memory.

2.4 Heterogeneous Programming

The dICU programming model assumes that the dICU threads execute on a physically separate device that operates as a coprocessor to the host running the C program.

The dICU programming model also assumes that both the host and the device maintain their own separate memory spaces in DRAM, referred to as host memory and device memory, respectively. Therefore, a program manages the global memory spaces visible to kernels through calls to the dICU runtime (described in [Programming Interface](#)).

This includes device memory allocation and deallocation as well as data transfer between host and device memory.

Note :

When developing applications based on Hamming SDK, the CUDA header files used must be these located in the directory where the SDK is installed.

If the CUDA header file used is not those specified, it may cause unknown problems.

3 Programming Interface

dICU provides a simple path for users familiar with the C programming language to easily write programs for execution by the device.

It consists of a minimal set of extensions to the C language and a runtime library.

The core language extensions have been introduced in [Programming Model](#). They allow programmers to define a kernel as a C function and use some new syntax to specify the grid and block dimension each time the function is called. A complete description of all extensions can be found in [Appendix A. C++ Language Extensions](#). Any source file that contains some of these extensions must be compiled with dlcc as outlined in [Compilation with dlcc](#).

The runtime provides C functions that execute on the host to allocate and deallocate device memory, transfer data between host memory and device memory, manage systems with multiple devices, and so on. A complete description of the runtime can be found in document *dICU-API.pdf*.

The runtime is built on top of a lower-level C API, the dICU driver API, which is also accessible by the application. The driver API provides an additional level of control by exposing lower-level concepts such as dICU contexts - the analogue of host processes for the device - and dICU modules - the analogue of dynamically loaded libraries for the device. Most applications do not use the driver API as they do not need this additional level of control and when using the runtime, context and module management are implicit, resulting in more concise code. The driver API is introduced in document *dICU-API.pdf* and fully described in the reference manual.

3.1 Compilation with dlcc

```
dlcc -o target_file_name.bc source_file_name.cu
```

For details on how to use dlcc, see document *DengLin-Compute-Compiler-User-Guide.pdf*.

3.2 dICU C Runtime

The runtime is implemented in the `curt` library. All its entry points are prefixed with `cuda`.

3.2.1 Initialization

There is no explicit initialization function for the runtime; it initializes the first time a runtime function is called (more specifically any function other than functions from the device and version management sections of the reference manual). One needs to keep this in mind when timing runtime function calls and when interpreting the error code from the first call into the runtime.

During initialization, the runtime creates a dICU context for each device in the system. This context is the primary context for this device and it is shared among all the host threads of the application. As part of this context creation, the device code is just-in-time compiled if necessary and loaded into device memory. This all happens under the hood and the runtime does not expose the primary context to the application.

When a host thread calls `cudaDeviceReset()`, this destroys the primary context of the device the host thread currently operates on (for example, the current device as defined in Device Selection). The next runtime function call made by any host thread that has this device as current will create a new primary context for this device.

3.2.2 Device Memory

As mentioned in [Heterogeneous Programming](#), the dICU programming model assumes a system composed of a host and a device, each with their own separate memory. Kernels operate out of device memory, so the runtime provides functions to allocate, deallocate, and copy device memory, as well as transferring data between host memory and device memory.

Device memory can be allocated as linear memory.

Linear memory exists on the device in a 40-bit address space, so separately allocated entities can reference one another via pointers, for example, in a binary tree.

Linear memory is typically allocated using `cudaMalloc()` and freed using `cudaFree()`, and data transfer between host memory and device memory are typically done using `cudaMemcpy()`. In the vector addition code sample of Kernels, the vectors need to be copied from host memory to device memory:

```
// Device code
__global__ void VecAdd(float* A, float* B, float* C, int N)
{
    int i = blockDim.x * blockIdx.x + threadIdx.x;
    if (i < N)
        C[i] = A[i] + B[i];
}

// Host code
int main()
{
    int N = ...;
    size_t size = N * sizeof(float);

    // Allocate input vectors h_A and h_B in host memory
    float* h_A = (float*)malloc(size);
    float* h_B = (float*)malloc(size);

    // Initialize input vectors
    ...

    // Allocate vectors in device memory
    float* d_A;
    cudaMalloc(&d_A, size);
    float* d_B;
    cudaMalloc(&d_B, size);
    float* d_C;
    cudaMalloc(&d_C, size);

    // Copy vectors from host memory to device memory
    cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);

    // Invoke kernel
    int threadsPerBlock = 256;
    int blocksPerGrid =
        (N + threadsPerBlock - 1) / threadsPerBlock;
```

```

    VecAdd<<<blocksPerGrid, threadsPerBlock>>>(d_A, d_B, d_C, N);

    // Copy result from device memory to host memory
    // h_C contains the result in host memory
    cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);

    // Free device memory
    cudaFree(d_A);
    cudaFree(d_B);
    cudaFree(d_C);

    // Free host memory
    ...
}

```

3.2.3 Shared Memory

Shared memory is allocated using the `__shared__` memory space specifier. Shared memory is expected to be much faster than global memory as mentioned in [Thread Hierarchy](#) and detailed in [Shared Memory](#). Any opportunity to replace global memory accesses by shared memory accesses should therefore be exploited as illustrated by the following matrix multiplication example.

The following code sample is an implementation of matrix multiplication that does take advantage of shared memory.

By blocking the computation this way, we take advantage of fast shared memory and save a lot of global memory bandwidth since A is only read (B.width / block_size) times from global memory and B is read (A.height / block_size) times.

The Matrix type from the previous code sample is augmented with a stride field, so that sub-matrices can be efficiently represented with the same type. `__device__` functions are used to get and set elements and build any sub-matrix from a matrix.

```

// Matrices are stored in row-major order:
// M(row, col) = *(M.elements + row * M.stride + col)
typedef struct {
    int width;
    int height;
    int stride;
    float* elements;
} Matrix;

// Get a matrix element
__device__ float GetElement(const Matrix A, int row, int col)
{
    return A.elements[row * A.stride + col];
}

// Set a matrix element
__device__ void SetElement(Matrix A, int row, int col,
                           float value)
{
    A.elements[row * A.stride + col] = value;
}

```

```

}

// Get the BLOCK_SIZExBLOCK_SIZE sub-matrix Asub of A that is
// located col sub-matrices to the right and row sub-matrices down
// from the upper-left corner of A
__device__ Matrix GetSubMatrix(Matrix A, int row, int col)
{
    Matrix Asub;
    Asub.width    = BLOCK_SIZE;
    Asub.height   = BLOCK_SIZE;
    Asub.stride   = A.stride;
    Asub.elements = &A.elements[A.stride * BLOCK_SIZE * row
                                + BLOCK_SIZE * col];

    return Asub;
}

// Thread block size
#define BLOCK_SIZE 16

// Forward declaration of the matrix multiplication kernel
__global__ void MatMulKernel(const Matrix, const Matrix, Matrix);

// Matrix multiplication - Host code
// Matrix dimensions are assumed to be multiples of BLOCK_SIZE
void MatMul(const Matrix A, const Matrix B, Matrix C)
{
    // Load A and B to device memory
    Matrix d_A;
    d_A.width = d_A.stride = A.width; d_A.height = A.height;
    size_t size = A.width * A.height * sizeof(float);
    cudaMalloc(&d_A.elements, size);
    cudaMemcpy(d_A.elements, A.elements, size,
               cudaMemcpyHostToDevice);

    Matrix d_B;
    d_B.width = d_B.stride = B.width; d_B.height = B.height;
    size = B.width * B.height * sizeof(float);

    cudaMalloc(&d_B.elements, size);
    cudaMemcpy(d_B.elements, B.elements, size,
               cudaMemcpyHostToDevice);

    // Allocate C in device memory
    Matrix d_C;
    d_C.width = d_C.stride = C.width; d_C.height = C.height;
    size = C.width * C.height * sizeof(float);
    cudaMalloc(&d_C.elements, size);

    // Invoke kernel
    dim3 dimBlock(BLOCK_SIZE, BLOCK_SIZE);
    dim3 dimGrid(B.width / dimBlock.x, A.height / dimBlock.y);
    MatMulKernel<<<dimGrid, dimBlock>>>>(d_A, d_B, d_C);

    // Read C from device memory
    cudaMemcpy(C.elements, d_C.elements, size,

```

```

        cudaMemcpyDeviceToHost);

    // Free device memory
    cudaFree(d_A.elements);
    cudaFree(d_B.elements);
    cudaFree(d_C.elements);
}

// Matrix multiplication kernel called by MatMul()
__global__ void MatMulKernel(Matrix A, Matrix B, Matrix C)
{
    // Block row and column
    int blockRow = blockIdx.y;
    int blockCol = blockIdx.x;

    // Each thread block computes one sub-matrix Csub of C
    Matrix Csub = GetSubMatrix(C, blockRow, blockCol);

    // Each thread computes one element of Csub
    // by accumulating results into Cvalue
    float Cvalue = 0;

    // Thread row and column within Csub
    int row = threadIdx.y;
    int col = threadIdx.x;

    // Loop over all the sub-matrices of A and B that are
    // required to compute Csub
    // Multiply each pair of sub-matrices together
    // and accumulate the results
    for (int m = 0; m < (A.width / BLOCK_SIZE); ++m) {

        // Get sub-matrix Asub of A
        Matrix Asub = GetSubMatrix(A, blockRow, m);

        // Get sub-matrix Bsub of B
        Matrix Bsub = GetSubMatrix(B, m, blockCol);

        // Shared memory used to store Asub and Bsub respectively
        __shared__ float As[BLOCK_SIZE][BLOCK_SIZE];
        __shared__ float Bs[BLOCK_SIZE][BLOCK_SIZE];

        // Load Asub and Bsub from device memory to shared memory
        // Each thread loads one element of each sub-matrix
        As[row][col] = GetElement(Asub, row, col);
        Bs[row][col] = GetElement(Bsub, row, col);

        // Synchronize to make sure the sub-matrices are loaded
        // before starting the computation
        __syncthreads();

        // Multiply Asub and Bsub together
        for (int e = 0; e < BLOCK_SIZE; ++e)
            Cvalue += As[row][e] * Bs[e][col];
    }
}

```

```

        // Synchronize to make sure that the preceding
        // computation is done before loading two new
        // sub-matrices of A and B in the next iteration
        __syncthreads();
    }

    // Write Csub to device memory
    // Each thread writes one element
    SetElement(Csub, row, col, Cvalue);
}

```

3.2.4 Asynchronous Concurrent Execution

dICU exposes the following operations as independent tasks that can operate concurrently with one another:

- Computation on the host
- Computation on the device
- Memory transfers from the host to the device/device to the host
- Memory transfers within the memory of a given device

3.2.4.1 Concurrent Execution between Host and Device

Concurrent host execution is facilitated through asynchronous library functions that return control to the host thread before the device completes the requested task. Using asynchronous calls, many device operations can be queued up together to be executed by the dICU driver when appropriate device resources are available. This relieves the host thread of much of the responsibility to manage the device, leaving it free for other tasks. The following device operations are asynchronous with respect to the host:

- Kernel launches
- Memory copies within a single device's memory
- Memory copies performed by functions that are suffixed with Async

3.2.4.2 Overlap of Data Transfer and Kernel Execution

It is possible to perform an intra-device copy simultaneously with kernel execution and/or with copies to or from the device. Intra-device copies are initiated using the standard memory copy functions with destination and source addresses residing on the same device.

3.2.4.3 Streams

Applications manage the concurrent operations described above through streams. A stream is a sequence of commands (possibly issued by different host threads) that execute in order. Different streams, on the other hand, may execute their commands out of order with respect to one another or concurrently; this behavior is not guaranteed and should therefore not be relied upon for correctness (for example, inter-kernel communication is undefined).

3.2.4.3.1 Creation and Destruction

A stream is defined by creating a stream object and specifying it as the stream parameter to a sequence of kernel launches and host <-> device memory copies. The following code sample creates two streams and allocates an array hostPtr of float.

```
cudaStream_t stream[2];
for (int i = 0; i < 2; ++i)
    cudaStreamCreate(&stream[i]);
float* hostPtr;
cudaMallocHost(&hostPtr, 2 * size);
```

Each of these streams is defined by the following code sample as a sequence of one memory copy from host to device, one kernel launch, and one memory copy from device to host:

```
for (int i = 0; i < 2; ++i) {
    cudaMemcpyAsync(inputDevPtr + i * size, hostPtr + i * size,
                    size, cudaMemcpyHostToDevice, stream[i]);
    MyKernel <<<100, 512, 0, stream[i]>>>
        (outputDevPtr + i * size, inputDevPtr + i * size, size);
    cudaMemcpyAsync(hostPtr + i * size, outputDevPtr + i * size,
                    size, cudaMemcpyDeviceToHost, stream[i]);
}
```

Streams are released by calling cudaStreamDestroy().

```
for (int i = 0; i < 2; ++i)
    cudaStreamDestroy(stream[i]);
```

3.2.4.3.2 Default Stream

Kernel launches and host <-> device memory copies that do not specify any stream parameter, or equivalently that set the stream parameter to zero, are issued to the default stream. They are therefore executed in order.

For code that is compiled using the --default-stream per-thread compilation flag (or that defines the CUDA_API_PER_THREAD_DEFAULT_STREAM macro before including CUDA headers (cuda.h and cuda_runtime.h)), the default stream is a regular stream and each host thread has its own default stream.

For code that is compiled using the --default-stream legacy compilation flag, the default stream is a special stream called the NULL stream and each device has a single NULL stream used for all host threads. The NULL stream is special as it causes implicit synchronization as described in Implicit Synchronization.

For code that is compiled without specifying a --default-stream compilation flag, --default-stream legacy is assumed as the default.

3.2.4.3.3 Explicit Synchronization

There are various ways to explicitly synchronize streams with each other.

cudaDeviceSynchronize() waits until all preceding commands in all streams of all host threads have completed.

`cudaStreamSynchronize()` takes a stream as a parameter and waits until all preceding commands in the given stream have completed. It can be used to synchronize the host with a specific stream, allowing other streams to continue executing on the device.

`cudaStreamWaitEvent()` takes a stream and an event as parameters (see [Events](#) for a description of events) and makes all the commands added to the given stream after the call to `cudaStreamWaitEvent()` delay their execution until the given event has completed. The stream can be 0, in which case all the commands added to any stream after the call to `cudaStreamWaitEvent()` wait on the event.

`cudaStreamQuery()` provides applications with a way to know if all preceding commands in a stream have completed.

To avoid unnecessary slowdowns, all these synchronization functions are usually best used for timing purposes or to isolate a launch or memory copy that is failing.

3.2.4.3.4 Implicit Synchronization

Two commands from different streams cannot run concurrently if any one of the following operations is issued in-between them by the host thread:

- a device memory allocation
- any command to the NULL stream

3.2.4.3.5 Callbacks

The runtime provides a way to insert a callback at any point into a stream via `cudaStreamAddCallback()`. A callback is a function that is executed on the host once all commands issued to the stream before the callback have completed. Callbacks in stream 0 are executed once all preceding tasks and commands issued in all streams before the callback have completed.

The following code sample adds the callback function `MyCallback` to each of two streams after issuing a host-to-device memory copy, a kernel launch and a device-to-host memory copy into each stream. The callback will begin execution on the host after each of the device-to-host memory copies completes.

```
void CUDART_CB MyCallback(cudaStream_t stream, cudaError_t status, void *data){
    printf("Inside callback %d\n", (size_t)data);
}
...
for (size_t i = 0; i < 2; ++i) {
    cudaMemcpyAsync(devPtrIn[i], hostPtr[i], size, cudaMemcpyHostToDevice, stream[i]);
    MyKernel<<<100, 512, 0, stream[i]>>>(devPtrOut[i], devPtrIn[i], size);
    cudaMemcpyAsync(hostPtr[i], devPtrOut[i], size, cudaMemcpyDeviceToHost, stream[i]);
    cudaStreamAddCallback(stream[i], MyCallback, (void*)i, 0);
}
```

The commands that are issued in a stream (or all commands issued to any stream if the callback is issued to stream 0) after a callback do not start executing before the callback has completed. The last parameter of `cudaStreamAddCallback()` is reserved for future use.

A callback must not make dICU API calls (directly or indirectly), as it might end up waiting on itself if it makes such a call leading to a deadlock.

3.2.4.4 Graphs

Graphs present a new model for work submission in dICU. A graph is a series of operations, such as kernel launches, connected by dependencies, which is defined separately from its execution. This allows a graph to be defined once and then launched repeatedly. Separating out the definition of a graph from its execution enables a number of optimizations: first, CPU launch costs are reduced compared to streams, because much of the setup is done in advance; second, presenting the whole workflow to dICU enables optimizations which might not be possible with the piecewise work submission mechanism of streams.

To see the optimizations possible with graphs, consider what happens in a stream: when you place a kernel into a stream, the host driver performs a sequence of operations in preparation for the execution of the kernel on the GPU. These operations, necessary for setting up and launching the kernel, are an overhead cost which must be paid for each kernel that is issued. For a GPU kernel with a short execution time, this overhead cost can be a significant fraction of the overall end-to-end execution time.

Work submission using graphs is separated into three distinct stages: definition, instantiation, and execution.

- During the definition phase, a program creates a description of the operations in the graph along with the dependencies between them.
- Instantiation takes a snapshot of the graph template, validates it, and performs much of the setup and initialization of work with the aim of minimizing what needs to be done at launch. The resulting instance is known as an executable graph.
- An executable graph may be launched into a stream, similar to any other dICU work. It may be launched any number of times without repeating the instantiation.

3.2.4.4.1 Graph Structure

An operation forms a node in a graph. The dependencies between the operations are the edges. These dependencies constrain the execution sequence of the operations.

An operation may be scheduled at any time once the nodes on which it depends are complete. Scheduling is left up to the dICU system.

Node Types

A graph node can be one of:

- Kernel
- CPU function call
- memory copy
- memset
- empty node
- child graph: To execute a separate nested graph.

3.2.4.4.2 Creating a Graph Using Graph APIs

Graphs can be created via two mechanisms: explicit API and stream capture. The following is an example of creating and executing the following graph.

```
// Create the graph - it starts out empty
cudaGraphCreate(&graph, 0);

// For the purpose of this example, we'll create
// the nodes separately from the dependencies to
```

```
// demonstrate that it can be done in two stages.
// Note that dependencies can also be specified
// at node creation.
cudaGraphAddKernelNode(&a, graph, NULL, 0, &nodeParams);
cudaGraphAddKernelNode(&b, graph, NULL, 0, &nodeParams);
cudaGraphAddKernelNode(&c, graph, NULL, 0, &nodeParams);
cudaGraphAddKernelNode(&d, graph, NULL, 0, &nodeParams);

// Now set up dependencies on each node
cudaGraphAddDependencies(graph, &a, &b, 1);    // A->B
cudaGraphAddDependencies(graph, &a, &c, 1);    // A->C
cudaGraphAddDependencies(graph, &b, &d, 1);    // B->D
cudaGraphAddDependencies(graph, &c, &d, 1);    // C->D
```

3.2.4.4.3 Creating a Graph Using Stream Capture

Stream capture provides a mechanism to create a graph from existing stream-based APIs. A section of code which launches work into streams, including existing code, can be bracketed with calls to `cudaStreamBeginCapture()` and `cudaStreamEndCapture()`. See below.

```
cudaGraph_t graph;

cudaStreamBeginCapture(stream);

kernel_A<<< ..., stream >>>(...);
kernel_B<<< ..., stream >>>(...);
libraryCall(stream);
kernel_C<<< ..., stream >>>(...);

cudaStreamEndCapture(stream, &graph);
```

A call to `cudaStreamBeginCapture()` places a stream in capture mode. When a stream is being captured, work launched into the stream is not enqueued for execution. It is instead appended to an internal graph that is progressively being built up. This graph is then returned by calling `cudaStreamEndCapture()`, which also ends capture mode for the stream. A graph which is actively being constructed by stream capture is referred to as a capture graph.

Stream capture can be used on any dICU stream except `cudaStreamLegacy` (the “NULL stream”). Note that it can be used on `cudaStreamPerThread`. If a program is using the legacy stream, it may be possible to redefine stream 0 to be the per-thread stream with no functional change. See [Default Stream](#).

Whether a stream is being captured can be queried with `cudaStreamIsCapturing()`.

3.2.4.4.4 Cross-stream Dependencies and Events

Stream capture can handle cross-stream dependencies expressed with `cudaEventRecord()` and `cudaStreamWaitEvent()`, if the event being waited upon was recorded into the same capture graph.

When an event is recorded in a stream that is in capture mode, it results in a captured event. A captured event represents a set of nodes in a capture graph.

When a captured event is waited on by a stream, it places the stream in capture mode if it is not yet, and the next item in the stream will have additional dependencies on the nodes in the captured event. The two streams are then being captured to the same capture graph.

When cross-stream dependencies are present in stream capture, `cudaStreamEndCapture()` must still be called in the same stream where `cudaStreamBeginCapture()` was called; this is the origin stream. Any other streams which are being captured to the same capture graph, due to event-based dependencies, must also be joined back to the origin stream. The code is shown below. All streams being captured to the same capture graph are taken out of capture mode upon `cudaStreamEndCapture()`. Failure to rejoin to the origin stream will result in failure of the overall capture operation.

```
// stream1 is the origin stream
cudaStreamBeginCapture(stream1);

kernel_A<<< ..., stream1 >>>(...);

// Fork into stream2
cudaEventRecord(event1, stream1);
cudaStreamWaitEvent(stream2, event1);

kernel_B<<< ..., stream1 >>>(...);
kernel_C<<< ..., stream2 >>>(...);

// Join stream2 back to origin stream (stream1)
cudaEventRecord(event2, stream2);
cudaStreamWaitEvent(stream1, event2);

kernel_D<<< ..., stream1 >>>(...);

// End capture in the origin stream
cudaStreamEndCapture(stream1, &graph);

// stream1 and stream2 no longer in capture mode
```

Note: When a stream is taken out of capture mode, the next non-captured item in the stream (if any) will still have a dependency on the most recent prior non-captured item, despite intermediate items having been removed.

3.2.4.4.5 Prohibited and Unhandled Operations

It is invalid to synchronize or query the execution status of a stream which is being captured or a captured event, because they do not represent items scheduled for execution. It is also invalid to query the execution status or synchronize a broader handle which encompasses an active stream capture, such as a device or context handle when any associated stream is in capture mode.

When any stream in the same context is being captured, and it was not created with `cudaStreamNonBlocking`, any attempted use of the legacy stream is invalid. This is because the legacy stream handle at all times encompasses these other streams; enqueueing to the legacy stream would create a dependency on the streams being captured, and querying it or synchronizing it would query or synchronize the streams being captured.

It is therefore also invalid to call synchronous APIs in this case. Synchronous APIs, such as `cudaMemcpy()`, enqueue work to the legacy stream and synchronize it before returning.

If the kernel param's memory is created by another context, the operation to capture any stream in the current context is invalid. It is therefore also invalid to use graph API with different context.

Note: As a general rule, when a dependency relation would connect something that is captured with something that was not captured and instead enqueued for execution, dICU prefers to return an error rather than ignore the dependency. An exception is made for placing a stream into or out of capture mode; this severs a dependency relation between items added to the stream immediately before and after the mode transition.

It is invalid to merge two separate capture graphs by waiting on a captured event from a stream which is being captured and is associated with a different capture graph than the event. It is invalid to wait on a non-captured event from a stream which is being captured.

3.2.4.4.6 Invalidation

When an invalid operation is attempted during stream capture, any associated capture graphs are invalidated. When a capture graph is invalidated, further use of any streams which are being captured or captured events associated with the graph is invalid and will return an error, until stream capture is ended with `cudaStreamEndCapture()`. This call will take the associated streams out of capture mode, but will also return an error value and a NULL graph.

3.2.4.4.7 Using Graph APIs

`cudaGraph_t` objects are not thread-safe. It is the responsibility of the user to ensure that multiple threads do not concurrently access the same `cudaGraph_t`.

A `cudaGraphExec_t` cannot run concurrently with itself. A launch of a `cudaGraphExec_t` will be ordered after previous launches of the same executable graph.

Graph execution is done in streams for ordering with other asynchronous work. However, the stream is for ordering only; it does not constrain the internal parallelism of the graph, nor does it affect where graph nodes execute.

3.2.4.5 Events

The runtime also provides a way to closely monitor the device's progress, as well as perform accurate timing, by letting the application asynchronously record events at any point in the program and query when these events are completed. An event has completed when all tasks - or optionally, all commands in a given stream - preceding the event have completed. Events in stream zero are completed after all preceding tasks and commands in all streams are completed.

3.2.4.5.1 Creation and Destruction

The following code sample creates and destroys two events:

```
cudaEvent_t start, stop;
cudaEventCreate(&start);
cudaEventCreate(&stop);

cudaEventDestroy(start);
cudaEventDestroy(stop);
```

3.2.4.5.2 Elapsed Time

The events created in [Creation and Destruction](#) can be used to time the code sample of [Creation and Destruction](#) in the following way:

```
cudaEventRecord(start, 0);
for (int i = 0; i < 2; ++i) {
    cudaMemcpyAsync(inputDev + i * size, inputHost + i * size,
                    size, cudaMemcpyHostToDevice, stream[i]);
    MyKernel<<<100, 512, 0, stream[i]>>>
        (outputDev + i * size, inputDev + i * size, size);
    cudaMemcpyAsync(outputHost + i * size, outputDev + i * size,
                    size, cudaMemcpyDeviceToHost, stream[i]);
}
cudaEventRecord(stop, 0);
cudaEventSynchronize(stop);
float elapsedTime;
cudaEventElapsedTime(&elapsedTime, start, stop);
```

3.2.4.6 Synchronous Calls

When a synchronous function is called, control is not returned to the host thread before the device has completed the requested task.

3.2.5 Surface Memory

A CUDA array, created with the `cudaArraySurfaceLoadStore` flag, can be read and written via a surface object or surface reference using the functions described in [A.1 Surface Functions](#). Both the maximum surface width and height are 65535.

3.2.5.1 Surface Object API

A surface object is created using `cudaCreateSurfaceObject()` from a resource description of type struct `cudaResourceDesc`.

The following code sample applies simple transformation kernel to a texture.

```
// Simple copy kernel
__global__ void copyKernel(cudaSurfaceObject_t inputSurfObj,
                           cudaSurfaceObject_t outputSurfObj, int width, int height)
{
    // Calculate surface coordinates
    unsigned int x = blockIdx.x * blockDim.x + threadIdx.x;
    unsigned int y = blockIdx.y * blockDim.y + threadIdx.y;
    if (x < width && y < height) {
        uchar4 data;
        // Read from input surface
        surf2Dread(&data, inputSurfObj, x, y);
        // Write to output surface
        surf2Dwrite(data, outputSurfObj, x, y);
    }
}

// Host code
```

```

int main()
{
    // Allocate CUDA arrays in device memory
    cudaChannelFormatDesc channelDesc =
        cudaCreateChannelDesc(8, 8, 8, 8, cudaChannelFormatKindUnsigned);
    cudaArray* cuInputArray;
    cudaMallocArray(&cuInputArray, &channelDesc, width, height,
        cudaArraySurfaceLoadStore);
    cudaArray* cuOutputArray;
    cudaMallocArray(&cuOutputArray, &channelDesc, width, height,
        cudaArraySurfaceLoadStore);
    // Copy to device memory some data located at address h_data
    // in host memory
    cudaMemcpyToArray(cuInputArray, 0, 0, h_data, size, cudaMemcpyHostToDevice);
    // Specify surface struct cudaResourceDesc resDesc;
    memset(&resDesc, 0, sizeof(resDesc)); resDesc.resType = cudaResourceTypeArray;
    // Create the surface objects
    resDesc.res.array.array = cuInputArray;
    cudaSurfaceObject_t inputSurfObj = 0;
    cudaCreateSurfaceObject(&inputSurfObj, &resDesc);
    resDesc.res.array.array = cuOutputArray;
    cudaSurfaceObject_t outputSurfObj = 0;
    cudaCreateSurfaceObject(&outputSurfObj, &resDesc);
    // Invoke kernel
    dim3 dimBlock(16, 16);
    dim3 dimGrid((width + dimBlock.x - 1) / dimBlock.x,
        (height + dimBlock.y - 1) / dimBlock.y);
    copyKernel<<<dimGrid, dimBlock>>>(inputSurfObj, outputSurfObj, width, height);
    // Destroy surface objects
    cudaDestroySurfaceObject(inputSurfObj);
    cudaDestroySurfaceObject(outputSurfObj);
    // Free device memory
    cudaFreeArray(cuInputArray);
    cudaFreeArray(cuOutputArray);
    return 0;
}

```

3.2.5.2 CUDA Arrays

CUDA arrays are opaque memory layouts optimized for image pixel fetching. They are one dimensional, two dimensional, or three-dimensional (**not supported now**) and composed of elements, each of which has 1, 2 or 4 components that may be signed or unsigned 8-bit, 16-bit, or 32-bit integers, 16-bit floats, or 32-bit floats. CUDA arrays are only accessible by kernels through surface reading and writing as described in [A.1 Surface Functions](#).

4 dICU Driver API Reference

Refer to document *dICU-API.pdf*.

登临科技保密材料

Appendix A. C++ Language Extensions

A.1 Surface Functions

Surface objects are described in [Surface Object API](#).

In the sections below, `boundaryMode` specifies the boundary mode, that is how out-of-range surface coordinates are handled; it is equal to either `cudaBoundaryModeClamp` in which case out-of-range coordinates are clamped to the valid range, or `cudaBoundaryModeZero`, in which case out-of-range reads return zero and out-of-range writes are ignored, or `cudaBoundaryModeTrap`, in which case out-of-range accesses cause the kernel execution to fail.

Notice:

- T only supports the data types whose lengths are 1, 2, or 4 bytes.
- Surface memory uses coordinates to address a pixel. This means that the X coordinate of the surface element accessed through the surface function **does not have to be multiplied by the byte size** of the element in order to access the same element through the surface function.

For example, the element at surface coordinates x and y of a two-dimensional floating-point CUDA array bound to a surface reference `surfRef` is accessed by `surf2Dread(surfRef, x, y)` via `surfRef`.

A.1.1 Surface Object API

`surf1Dread()`

```
template<class T>
T surf1Dread(cudaSurfaceObject_t surfobj, int x, boundaryMode = cudaBoundaryModeTrap);
```

reads the CUDA array specified by the one-dimensional surface object `surfobj` using coordinate x.

`surf1Dwrite()`

```
template<class T>
void surf1Dwrite(T data, cudaSurfaceObject_t surfobj, int x, boundaryMode =
cudaBoundaryModeTrap);
```

writes value data to the CUDA array specified by the one-dimensional surface object `surfobj` at coordinate x.

`surf2Dread()`

```
template<class T>
T surf2Dread(cudaSurfaceObject_t surfobj, int x, int y, boundaryMode =
cudaBoundaryModeTrap);
template<class T>
void surf2Dread(T* data, cudaSurfaceObject_t surfobj, int x, int y, boundaryMode =
cudaBoundaryModeTrap);
```

reads the CUDA array specified by the two-dimensional surface object `surfobj` using coordinates x and y.

surf2Dwrite()

```
template<class T>
void surf2Dwrite(T data, cudaSurfaceObject_t surfObj, int x, int y, boundaryMode =
cudaBoundaryModeTrap);
```

writes value data to the CUDA array specified by the two-dimensional surface object `surfObj` at coordinates `x` and `y`.

surf1DLayeredread()

```
template<class T>
T surf1DLayeredread(cudaSurfaceObject_t surfObj, int x, int layer, boundaryMode =
cudaBoundaryModeTrap);
template<class T>
void surf1DLayeredread(T data, cudaSurfaceObject_t surfObj, int x, int layer, boundaryMode
= cudaBoundaryModeTrap);
```

reads the CUDA array specified by the one-dimensional layered surface object `surfObj` using coordinate `x` and the index `layer`.

surf1DLayeredwrite()

```
template<class Type>
void surf1DLayeredwrite(T data, cudaSurfaceObject_t surfObj, int x, int layer, boundaryMode
= cudaBoundaryModeTrap);
```

writes value data to the CUDA array specified by the two-dimensional layered surface object `surfObj` at coordinate `x` and the index `layer`.

surf2DLayeredread()

```
template<class T>
T surf2DLayeredread(cudaSurfaceObject_t surfObj, int x, int y, int layer, boundaryMode =
cudaBoundaryModeTrap);
template<class T>
void surf2DLayeredread(T data, cudaSurfaceObject_t surfObj, int x, int y, int layer,
boundaryMode = cudaBoundaryModeTrap);
```

reads the CUDA array specified by the two-dimensional layered surface object `surfObj` using coordinates `x` and `y`, and the index `layer`.

surf2DLayeredwrite()

```
template<class T>
void surf2DLayeredwrite(T data, cudaSurfaceObject_t surfObj, int x, int y, int layer,
boundaryMode = cudaBoundaryModeTrap);
```

writes value data to the CUDA array specified by the one-dimensional layered surface object `surfObj` at coordinates `x` and `y`, and the index `layer`.

A.2 Printf Functions

Format Specifiers

As for standard `printf()`, format specifiers take the form: `%[flags][width][.precision][size]type`

The following fields are supported (see widely-available documentation for a complete description of all behaviors):

- *Flags:* `# ' 0 ' + ' - '`
- *Width:* `* ' 0-9'`
- *Precision:* `' 0-9'`
- *Size:* `h ' l ' ll'`
- *Type:* `' %cdiouxXpeEfgGaA'`

Note that `printf()` will accept any combination of flag, width, precision, size and type, whether or not overall they form a valid format specifier. In other words, `"%hd"` will be accepted and `printf` will expect a double-precision variable in the corresponding location in the argument list.

Limitations

Final formatting of the `printf()` output takes place on the host system. This means that the format string must be understood by the host-system's compiler and C library. Every effort has been made to ensure that the format specifiers supported by `printf` function form a universal subset from the most common host compilers, but exact behavior will be host-OS-dependent.

The `printf()` cannot support string format specifier. Which means the `%s` cannot working and it will report compiling error when build the program.

As described in [Format Specifiers](#), `printf()` will accept *all* combinations of valid flags and types. This is because it cannot determine what will and will not be valid on the host system where the final output is formatted. The effect of this is that output may be undefined if the program emits a format string which contains invalid combinations.

The `printf()` command can accept at most 32 arguments in addition to the format string. Additional arguments beyond this will be ignored, and the format specifier output as-is.

Owing to the differing size of the long type on 64-bit Windows platforms (four bytes on 64-bit Windows platforms, eight bytes on other 64-bit platforms), a kernel which is compiled on a non-Windows 64-bit machine but then run on a win64 machine will see corrupted output for all format strings which include `"%ld"`. It is recommended that the compilation platform matches the execution platform to ensure safety.

The output buffer for `printf()` is set to a fixed size of 2MB. It is circular and if more output is produced during kernel execution than can fit in the buffer, older output is overwritten. It is flushed only when one of these actions is performed:

- Kernel launch via `<<<<>>>` or `cuLaunchKernel()` (at the start of the launch, and if the `CUDA_LAUNCH_BLOCKING` environment variable is set to 1, at the end of the launch as well).
- Synchronization via `cudaDeviceSynchronize()`, `cuCtxSynchronize()`, `cudaStreamSynchronize()`, `cuStreamSynchronize()`, `cudaEventSynchronize()`, or `cuEventSynchronize()`.
- Memory copies via any blocking version of `cudaMemcpy()` or `cuMemcpy()`.
- Context destruction via `cudaDeviceReset()` or `cuCtxDestroy()`.

Internally `printf()` uses a shared data structure and so it is possible that calling `printf()` might change the order of execution of threads.

Examples

The following code sample:

```
#include <stdio.h>
__global__ void hello(float f)
{
    printf("Hello thread %d, f=%f\n", threadIdx.x, f);
}
int main()
{
    hello<<<1, 5>>>>(1.2345f);
    cudaDeviceSynchronize();
    return 0;
}
```

will output:

```
Hello thread 2, f=1.2345
Hello thread 1, f=1.2345
Hello thread 4, f=1.2345
Hello thread 0, f=1.2345
Hello thread 3, f=1.2345
```

Notice how each thread encounters the `printf()` command, so there are as many lines of output as there were threads launched in the grid. As expected, global values (i.e., float `f`) are common between all threads, and local values (i.e., `threadIdx.x`) are distinct per-thread.