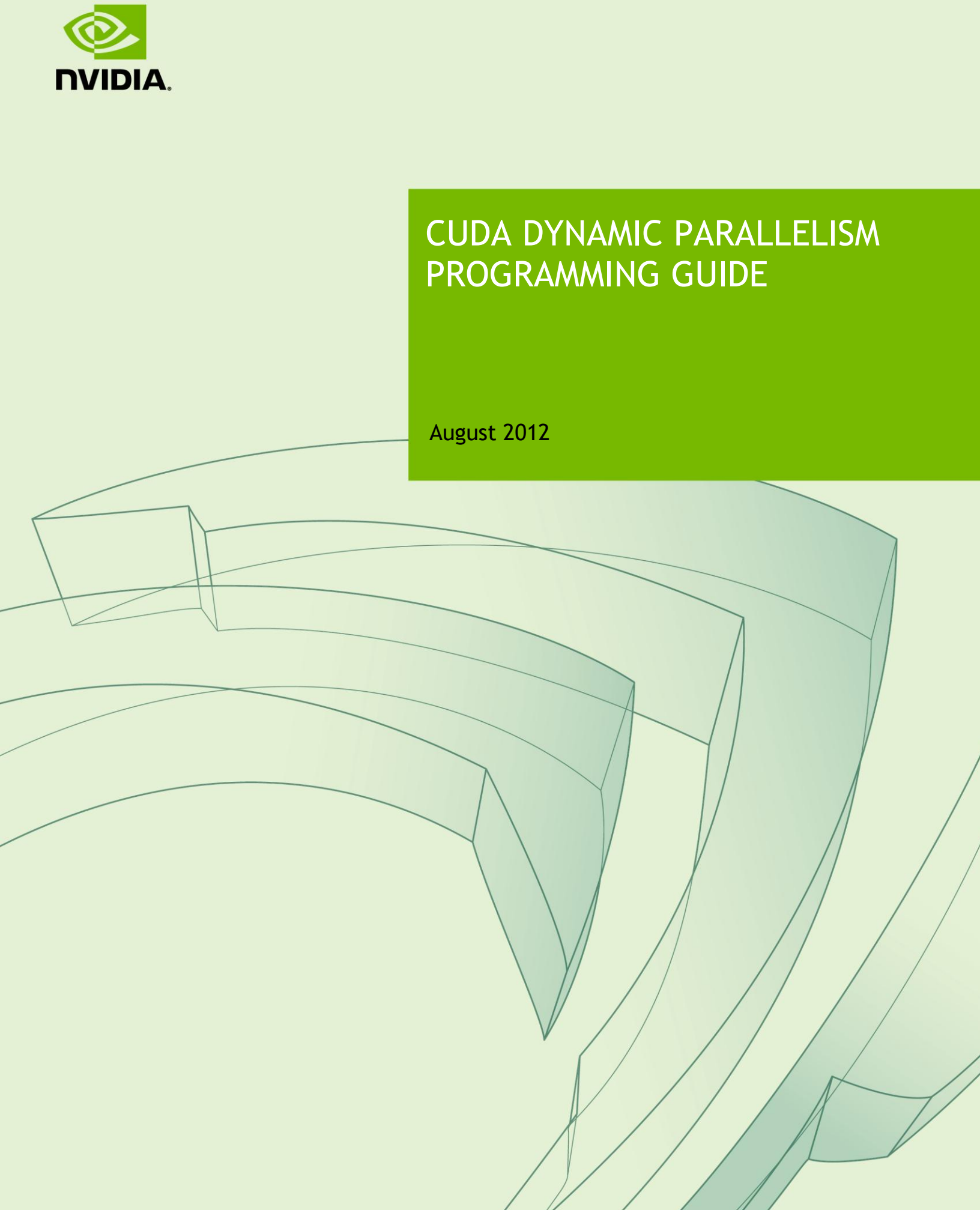


# CUDA DYNAMIC PARALLELISM PROGRAMMING GUIDE

August 2012

# TABLE OF CONTENTS

# INTRODUCTION

This document provides guidance on how to design and develop software that takes advantage of the new Dynamic Parallelism capabilities introduced with CUDA 5.0.

## OVERVIEW

Dynamic Parallelism is an extension to the CUDA programming model enabling a CUDA kernel to create and synchronize with new work directly on the GPU. The creation of parallelism dynamically at whichever point in a program that it is needed offers exciting new capabilities.

The ability to create work directly from the GPU can reduce the need to transfer execution control and data between host and device, as launch configuration decisions can now be made at runtime by threads executing on the device. Additionally, data-dependent parallel work can be generated inline within a kernel at run-time, taking advantage of the GPU's hardware schedulers and load balancers dynamically and adapting in response to data-driven decisions or workloads. Algorithms and programming patterns that had previously required modifications to eliminate recursion, irregular loop structure, or other constructs that do not fit a flat, single-level of parallelism may more transparently be expressed.

This document describes the extended capabilities of CUDA which enable Dynamic Parallelism, including the modifications and additions to the CUDA programming model necessary to take advantage of these, as well as guidelines and best practices for exploiting this added capacity.

Dynamic Parallelism is only supported by devices of compute capability 3.5 and higher.

# GLOSSARY

Definitions for terms used in this guide.

## Grid

A Grid is a collection of Threads. Threads in a Grid execute a Kernel Function and are divided into Thread Blocks.

## Thread Block

A Thread Block is a group of threads which execute on the same multiprocessor (SMX). Threads within a Thread Block have access to shared memory and can be explicitly synchronized.

## Kernel Function

A Kernel Function is an implicitly parallel subroutine that executes under the CUDA execution and memory model for every Thread in a Grid.

## Host

The Host refers to the execution environment that initially invoked CUDA, typically the thread running on a system's CPU processor.

## Parent

A Parent Thread, Thread Block, or Grid is one that has launched new grid(s), the Child Grid(s). The Parent is not considered completed until all of its launched Child Grids have also completed.

## Child

A Child thread, block, or grid is one that has been launched by a Parent grid. A Child grid must complete before the Parent Thread, Thread Block, or Grid are considered complete.

## Thread Block Scope

Objects with Thread Block Scope have the lifetime of a single Thread Block. They only have defined behavior when operated on by Threads in the Thread Block that created the object and are destroyed when the Thread Block that created them is complete.

## Device Runtime

The Device Runtime refers to the runtime system and APIs available to enable Kernel Functions to use Dynamic Parallelism.

# EXECUTION ENVIRONMENT AND MEMORY MODEL

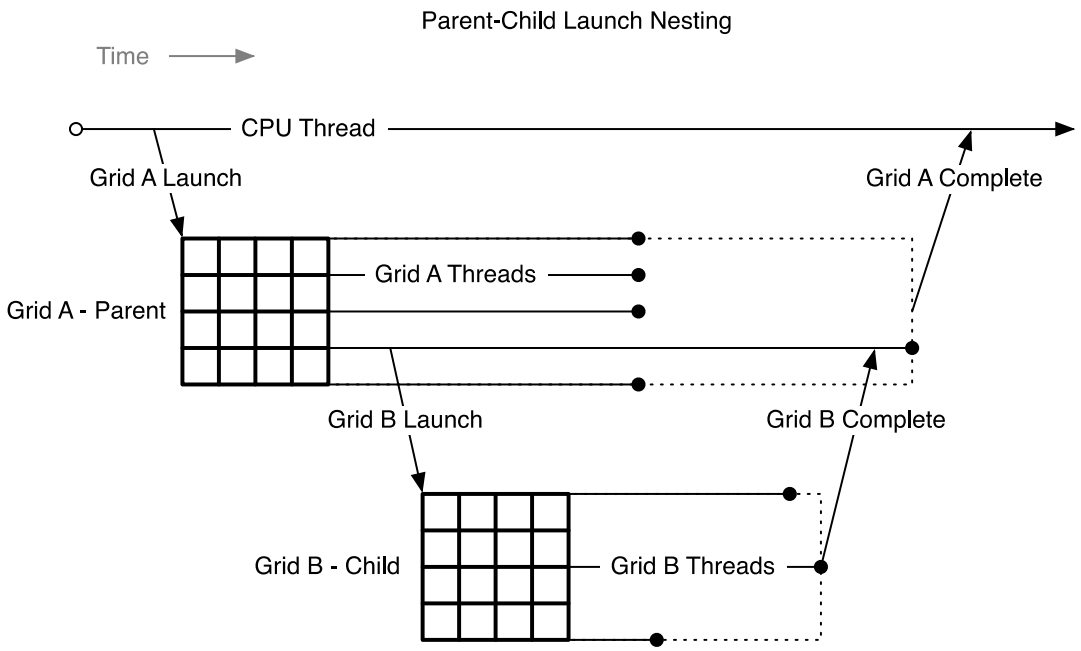
## EXECUTION ENVIRONMENT

The CUDA execution model is based on primitives of threads, thread blocks, and grids, with kernel functions defining the program executed by individual threads within a thread block and grid. When a kernel function is invoked the grid's properties are described by an execution configuration, which has a special syntax in CUDA. Support for dynamic parallelism in CUDA extends the ability to configure, launch, and synchronize upon new grids to threads that are running on the device.

### Parent and Child Grids

A device thread that configures and launches a new grid belongs to the parent grid, and the grid created by the invocation is a child grid.

The invocation and completion of child grids is properly nested, meaning that the parent grid is not considered complete until all child grids created by its threads have completed. Even if the invoking threads do not explicitly synchronize on the child grids launched, the runtime guarantees an implicit synchronization between the parent and child.




## Scope of CUDA Primitives

On both host and device, the CUDA runtime offers an API for launching kernels, for waiting for launched work to complete, and for tracking dependencies between launches via streams and events. On the host system, the state of launches and the CUDA primitives referencing streams and events are shared by all threads within a process; however processes execute independently and may not share CUDA objects.

A similar hierarchy exists on the device: launched kernels and CUDA objects are visible to all threads in a thread block, but are independent between thread blocks. This means for example that a stream may be created by one thread and used by any other thread in the same thread block, but may not be shared with threads in any other thread block.

## Synchronization

CUDA runtime operations from any thread, including kernel launches, are visible across a thread block. This means that an invoking thread in the parent grid may perform synchronization on the grids launched by that thread, by other threads in the thread block, or on streams created within the same thread block. Execution of a thread block is not considered complete until all launches by all threads in the block have completed. If all threads in a block exit before all child launches have completed, a synchronization operation will automatically be triggered.

## Streams & Events

CUDA Streams and Events allow control over dependencies between grid launches: grids launched into the same stream execute in-order, and events may be used to create dependencies between streams. Streams and events created on the device serve this exact same purpose.

Streams and events created within a grid exist within thread block scope but have undefined behavior when used outside of the thread block where they were created. As described above, all work launched by a thread block is implicitly synchronized when the block exits; work launched into streams is included in this, with all dependencies resolved appropriately. The behavior of operations on a stream that has been modified outside of thread block scope is undefined.

Streams and events created on the host have undefined behavior when used within any kernel, just as streams and events created by a parent grid have undefined behavior if used within a child grid.

## Ordering and Concurrency

The ordering of kernel launches from the device runtime follows CUDA Stream ordering semantics. Within a thread block, all kernel launches into the same stream are executed in-order. With multiple threads in the same thread block launching into the same stream, the ordering within the stream is dependent on the thread scheduling within the block, which may be controlled with synchronization primitives such as *__syncthreads()*.

Note that because streams are shared by all threads within a thread block, the implicit 'NULL' stream is also shared. If multiple threads in a thread block launch into the implicit stream, then these launches will be executed in-order. If concurrency is desired, explicit named streams should be used.

Dynamic Parallelism enables concurrency to be expressed more easily within a program; however, the device runtime introduces no new concurrency guarantees within the CUDA execution model. There is no guarantee of concurrent execution between any number of different thread blocks on a device.

The lack of concurrency guarantee extends to parent thread blocks and their child grids. When a parent thread block launches a child grid, the child is not guaranteed to begin execution until the parent thread block reaches an explicit synchronization point (e.g. *cudaDeviceSynchronize()*).

While concurrency will often easily be achieved, it may vary as a function of device configuration, application workload, and runtime scheduling. It is therefore unsafe to depend upon any concurrency between different thread blocks.

## Device Management

There is no multi-GPU support from the device runtime; the device runtime is only capable of operating on the device upon which it is currently executing. It is permitted, however, to query properties for any CUDA capable device in the system.

# MEMORY MODEL

Parent and child grids share the same global and constant memory storage, but have distinct local and shared memory.

## Coherence and Consistency

### Global Memory

Parent and child grids have coherent access to global memory, with weak consistency guarantees between child and parent. There are two points in the execution of a child grid when its view of memory is fully consistent with the parent thread: when the child grid is invoked by the parent, and when the child grid completes as signaled by a synchronization API invocation in the parent thread.

All global memory operations in the parent thread prior to the child grid's invocation are visible to the child grid. All memory operations of the child grid are visible to the parent after the parent has synchronized on the child grid's completion.

In the following example, the child grid executing `child_launch` is only guaranteed to see the modifications to `data` made before the child grid was launched. Since thread 0 of the parent is performing the launch, the child will be consistent with the memory seen by thread 0 of the parent. Due to the first *__syncthreads()* call, the child will see `data[0]=0, data[1]=1, ..., data[255]=255` (without the *__syncthreads()* call, only `data[0]` would be guaranteed to be seen by the child). When the child grid returns, thread 0 is guaranteed to see modifications made by the threads in its child grid. Those

modifications become available to the other threads of the parent grid only after the second *__syncthreads()* call:

```
__global__ void child_launch(int *data) {
     data[threadIdx.x] = data[threadIdx.x]+1;
}

__global__ void parent_launch(int *data) {
     data[threadIdx.x] = threadIdx.x;

     __syncthreads();

     If (threadIdx.x == 0) {
          child_launch<<< 1, 256 >>>(data);
          cudaDeviceSynchronize();
     }

     __syncthreads();
}

void host_launch(int *data) {
     parent_launch<<< 1, 256 >>>(data);
}
```

## Zero-Copy Memory

Zero-copy system memory has identical coherence and consistency guarantees to global memory, and follows the semantics detailed above. A kernel may not allocate or free zero-copy memory, but may use pointers to zero-copy passed in from the host program.

## Constant Memory

Constants are immutable and may not be modified from the device, even between parent and child launches. That is to say, the value of all __constant__ variables must be set from the host prior to launch. Constant memory is inherited automatically by all child kernels from their respective parents.

Taking the address of a constant memory object from within a kernel thread has the same semantics as for all CUDA programs, and passing that pointer from parent to child or from a child to parent is naturally supported.

## Shared and Local Memory

Shared and Local memory is private to a thread block or thread, respectively, and is not visible or coherent between parent and child. Behavior is undefined when an object in

one of these locations is referenced outside of the scope within which it belongs, and may cause an error.

The NVIDIA compiler will attempt to warn if it can detect that a pointer to local or shared memory is being passed as an argument to a kernel launch. At runtime, the programmer may use the *__isGlobal()* intrinsic to determine whether a pointer references global memory and so may safely be passed to a child launch.

Note that calls to *cudaMemcpy*Async()* or *cudaMemset*Async()* may invoke new child kernels on the device in order to preserve stream semantics. As such, passing shared or local memory pointers to these APIs is illegal and will return an error.

### Local Memory

Local memory is private storage for an executing thread, and is not visible outside of that thread. It is illegal to pass a pointer to local memory as a launch argument when launching a child kernel. The result of dereferencing such a local memory pointer from a child will be undefined.

For example the following is illegal, with undefined behavior if `x_array` is accessed by `child_launch`:

```
int x_array[10];        // Creates x_array in parent's local memory
child_launch<<< 1, 1 >>>(x_array);
```

It is sometimes difficult for a programmer to be aware of when a variable is placed into local memory by the compiler. As a general rule, all storage passed to a child kernel should be allocated explicitly from the global-memory heap, either with *cudaMalloc(), new()* or by declaring __device__ storage at global scope.

For example:

```
__device__ int value;
__device__ void x() {
    value = 5;
    child<<< 1, 1 >>>(&value);
}
```

Correct - "value" is global storage

```
__device__ void y() {
    int value = 5;
    child<<< 1, 1 >>>(&value);
}
```

Invalid - "value" is local storage

## Texture Memory

Writes to the global memory region over which a texture is mapped are incoherent with respect to texture accesses. Coherence for texture memory is enforced at the invocation of a child grid and when a child grid completes. This means that writes to memory prior to a child kernel launch are reflected in texture memory accesses of the child. Similarly,

writes to memory by a child will be reflected in the texture memory accesses by a parent, but only after the parent synchronizes on the child's completion. Concurrent accesses by parent and child may result in inconsistent data.

# PROGRAMMING INTERFACE

## CUDA C/C++ REFERENCE

This section describes changes and additions to the CUDA C/C++ language extensions for supporting Dynamic Parallelism.

The language interface and API available to CUDA kernels using CUDA C/C++ for Dynamic Parallelism, referred to as the Device Runtime, is substantially like that of the CUDA Runtime API available on the host. Where possible the syntax and semantics of the CUDA Runtime API have been retained in order to facilitate ease of code reuse for routines that may run in either the host or device environments.

As with all code in CUDA C/C++, the APIs and code outlined here is per-thread code. This enables each thread to make unique, dynamic decisions regarding what kernel or operation to execute next. There are no synchronization requirements between threads within a block to execute any of the provided device runtime APIs, which enables the device runtime API functions to be called in arbitrarily divergent kernel code without deadlock.

### Device-Side Kernel Launch

Kernels may be launched from the device using the standard CUDA <<< >>> syntax:

```
kernel_name<<< Dg, Db, Ns, S >>>([kernel arguments]);
```

- `Dg` is of type `dim3` and specifies the dimensions and size of the grid
- `Db` is of type `dim3` and specifies the dimensions and size of each thread block

- `Ns` is of type `size_t` and specifies the number of bytes of shared memory that is dynamically allocated per thread block for this call and addition to statically allocated memory. `Ns` is an optional argument that defaults to 0.
- `S` is of type `cudaStream_t` and specifies the stream associated with this call. The stream must have been allocated in the same thread block where the call is being made. `S` is an optional argument that defaults to 0.

## Launches are Asynchronous

Identical to host-side launches, all device-side kernel launches are asynchronous with respect to the launching thread. That is to say, the <<<>>> launch command will return immediately and the launching thread will continue to execute until it hits an explicit launch-synchronization point such as *cudaDeviceSynchronize()*. The grid launch is posted to the device and will execute independently of the parent thread. The child grid may begin execution at any time after launch, but is not guaranteed to begin execution until the launching thread reaches an explicit launch-synchronization point.

## Launch Environment Configuration

All global device configuration settings (e.g. shared memory & L1 cache size as returned from *cudaDeviceGetCacheConfig()*, and device limits returned from *cudaDeviceGetLimit()*) will be inherited from the parent. That is to say if, when the parent is launched, execution is configured globally for 16k of shared memory and 48k of L1 cache, then the child's execution state will be configured identically. Likewise, device limits such as stack size will remain as-configured.

For host-launched kernels, per-kernel configurations set from the host will take precedence over the global setting. These configurations will be used when the kernel is launched from the device as well. It is not possible to reconfigure a kernel's environment from the device.

## Launch From __host__ __device__ Functions

Although the device runtime enables kernel launches from either the host or device, kernel launches from __host__ __device__ functions are unsupported. The compiler will fail to compile if a __host__ device__ function is used to launch a kernel.

# Streams

Both named and unnamed (NULL) streams are available from the device runtime. Named streams may be used by any thread within a thread-block, but stream handles may not be passed to other blocks or child/parent kernels. In other words, a stream should be treated as private to the block in which it is created. Stream handles are not guaranteed to be unique between blocks, so using a stream handle within a block that did not allocate it will result in undefined behavior.

Similar to host-side launch, work launched into separate streams may run concurrently, but actual concurrency is not guaranteed. Programs that depend upon concurrency between child kernels are not supported by the CUDA programming model and will have undefined behavior.

The host-side NULL stream's cross-stream barrier semantic is not supported on the device (see below for details). In order to retain semantic compatibility with the host runtime, all device streams must be created using the *cudaStreamCreateWithFlags()* API, passing the *cudaStreamNonBlocking* flag. The *cudaStreamCreate()* call is a host-runtime-only API and will fail to compile for the device.

As *cudaStreamSynchronize()* and *cudaStreamQuery()* are unsupported by the device runtime, *cudaDeviceSynchronize()* should be used instead when the application needs to know that stream-launched child kernels have completed.

## The Implicit (NULL) Stream

Within a host program, the unnamed (NULL) stream has additional barrier synchronization semantics with other streams (see the CUDA Programming Guide for details). The device runtime offers a single implicit, unnamed stream shared between all threads in a block, but as all named streams must be created with the *cudaStreamNonBlocking* flag, work launched into the NULL stream will not insert an implicit dependency on pending work in any other streams.

# Events

Only the inter-stream synchronization capabilities of CUDA events are supported. This means that *cudaStreamWaitEvent()* is supported, but *cudaEventSynchronize()* *cudaEventElapsedTime()*, and *cudaEventQuery()* are not. As *cudaEventElapsedTime()* is not supported, cudaEvents must be created via *cudaEventCreateWithFlags()*, passing the *cudaEventDisableTiming* flag.

As for all device runtime objects, event objects may be shared between all threads within the thread-block which created them but are local to that block and may not be passed to other kernels, or between blocks within the same kernel. Event handles are not guaranteed to be unique between blocks, so using an event handle within a block that did not create it will result in undefined behavior.

# Synchronization

The *cudaDeviceSynchronize()* function will synchronize on all work launched by any thread in the thread-block up to the point where *cudaDeviceSynchronize()* was called. Note that *cudaDeviceSynchronize()* may be called from within divergent code (see below).

It is up to the program to perform sufficient additional inter-thread synchronization, for example via a call to *__syncthreads()*, if the calling thread is intended to synchronize with child grids invoked from other threads.

## Block-Wide Synchronization

The *cudaDeviceSynchronize()* function does not imply intra-block synchronization. In particular, without explicit synchronization via a *__syncthreads()* directive the calling thread can make no assumptions about what work has been launched by any thread other than itself. For example if multiple threads within a block are each launching work and synchronization is desired for all this work at once (perhaps because of event-based dependencies), it is up to the program to guarantee that this work is submitted by all threads before calling *cudaDeviceSynchronize()*.

Because the implementation is permitted to synchronize on launches from any thread in the block, it is quite possible that simultaneous calls to *cudaDeviceSynchronize()* by multiple threads will drain all work in the first call and then have no effect for the later calls.

# Device Management

Only the device on which a kernel is running will be controllable from that kernel. This means that device APIs such as *cudaSetDevice()* are not supported by the device runtime. The active device as seen from the GPU (returned from *cudaGetDevice*()) will have the same device number as seen from the host system. The *cudaGetDeviceProperty()* call may request information about another device as this API allows specification of a device ID

as a parameter of the call. Note that the catch-all *cudaGetDeviceProperties()* API is not offered by the device runtime – properties must be queried individually.

# Memory Declarations

## Device and Constant Memory

Memory declared at file scope with __device__ or __constant__ qualifiers behave identically when using the device runtime. All kernels may read or write __device__ variables, whether the kernel was initially launched by the host or device runtime. Equivalently, all kernels will have the same view of __constant__s as declared at the module scope.

## Textures & Surfaces

CUDA supports dynamically created texture and surface objects[1], where a texture reference may be created on the host, passed to a kernel, used by that kernel, and then destroyed from the host. The device runtime does not allow creation or destruction of texture or surface objects from within device code, but texture and surface objects created from the host may be used and passed around freely on the device. Regardless of where they are created, dynamically created texture objects are always valid and may be passed to child kernels from a parent.

**NOTE: The device runtime does not support legacy module-scope (i.e. Fermi-style) textures and surfaces within a kernel launched from the device.** Module-scope (legacy) textures may be created from the host and used in device code as for any kernel, but may only be used by a top-level kernel (i.e. the one which is launched from the host).

## Shared Memory Variable Declarations

In CUDA C/C++ shared memory can be declared either as a statically sized file-scope or function-scoped variable, or as an extern variable with the size determined at runtime by the kernel's caller via a launch configuration argument. Both types of declarations are valid under the device runtime.

---

[1] Dynamically created texture and surface objects are an addition to the CUDA memory model introduced with CUDA 5.0. Please see the CUDA Programming Guide for details.

```
__global__ void permute(int n, int *data) {
      extern __shared__ int smem[];
      if (n <= 1)
            return;

      smem[threadIdx.x] = data[threadIdx.x];
      __syncthreads();

      permute_data(smem, n);
      __syncthreads();

      // Write back to GMEM since we can't pass SMEM to children.
      data[threadIdx.x] = smem[threadIdx.x];
      __syncthreads();

      if (threadIdx.x == 0) {
            permute<<< 1, 256, n/2*sizeof(int) >>>(n/2, data);
            permute<<< 1, 256, n/2*sizeof(int) >>>(n/2, data+n/2);
      }
}

void host_launch(int *data) {
      permute<<< 1, 256, 256*sizeof(int) >>>(256, data);
}
```

### Symbol Addresses

Device-side symbols (i.e. those marked __device__) may be referenced from within a kernel simply via the "&" operator, as all global-scope device variables are in the kernel's visible address space. This also applies to __constant__ symbols, although in this case the pointer will reference read-only data.

Given that device-side symbols can be referenced directly, those CUDA runtime APIs which reference symbols (e.g. *cudaMemcpyToSymbol()* or *cudaGetSymbolAddress()*) are redundant and hence not supported by the device runtime. Note this implies that __constant__ data cannot be altered from within a running kernel, even ahead of a child kernel launch, as references to __constant__ space are read-only.

## API Errors and Launch Failures

As usual for the CUDA runtime, any function may return an error code. The last error code returned is recorded and may be retrieved via the *cudaGetLastError()* call. Errors are recorded per-thread, so that each thread can identify the most recent error that it has generated. The error code is of type *cudaError_t.*

Similar to a host-side launch, device-side launches may fail for many reasons (invalid arguments, etc). The user must call *cudaGetLastError()* to determine if a launch generated

an error, however lack of an error after launch does not imply the child kernel completed successfully.

For device-side exceptions, e.g., access to an invalid address, an error in a child grid will be returned to the host instead of being returned by the parent's call to *cudaDeviceSynchronize()*.

## Launch Setup APIs

Kernel launch is a system-level mechanism exposed through the device runtime library, and as such is available directly from PTX via the underlying *cudaGetParameterBuffer()* and *cudaLaunchDevice()* APIs. It is permitted for a CUDA application to call these APIs itself, with the same requirements as for PTX. In both cases, the user is then responsible for correctly populating all necessary data structures in the correct format according to specification. Backwards compatibility is guaranteed in these data structures.

As with host-side launch, the device-side operator <<<>>> maps to underlying kernel launch APIs. This is so that users targeting PTX will be able to enact a launch, and so that the compiler front-end can translate <<<>>> into these calls.

| Runtime API Launch Functions | Description of Difference From Host Runtime Behaviour (behaviour is identical if no description) |
|---|---|
| **cudaGetParameterBuffer** | Generated automatically from <<<>>>. Note different API to host equivalent. |
| **cudaLaunchDevice** | Generated automatically from <<<>>>. Note different API to host equivalent. |

New Device-only launch implementation functions

The APIs for these launch functions are different to those of the CUDA Runtime API, and are defined as follows:

```
extern __device__ cudaError_t cudaGetParameterBuffer(void **params);
extern __device__ cudaError_t cudaLaunchDevice(void *kernel,
                                               void* params, dim3 gridDim,
                                               dim3 blockDim,
                                               unsigned int sharedMemSize = 0,
                                               cudaStream_t stream = 0);
```

# API Reference

The portions of the CUDA Runtime API supported in the device runtime are detailed here. Host and device runtime APIs have identical syntax; semantics are the same except where indicated. The table below provides an overview of the API relative to the version available from the host.

| Runtime API Functions | Details |
|---|---|
| **cudaDeviceSynchronize** | Synchronizes on work launched from thread's own block only |
| **cudaDeviceGetCacheConfig** | |
| **cudaDeviceGetLimit** | |
| **cudaGetLastError** | Last error is per-thread state, not per-block state |
| **cudaPeekAtLastError** | |
| **cudaGetErrorString** | |
| **cudaGetDeviceCount** | |
| **cudaGetDeviceProperty** | Will return properties for any device |
| **cudaGetDevice** | Always returns current device ID as would be seen from host |
| **cudaStreamCreateWithFlags** | Must pass *cudaStreamNonBlocking* flag |
| **cudaStreamDestroy** | |
| **cudaStreamWaitEvent** | |
| **cudaEventCreateWithFlags** | Must pass *cudaEventDisableTiming* flag |
| **cudaEventRecord** | |
| **cudaEventDestroy** | |
| **cudaFuncGetAttributes** | |
| **cudaMemcpyAsync** | Notes about all *memcpy*/*memset* functions:<br>Only async memcpy/set functions are supported<br>Only device-to-device memcpy is permitted<br>May not pass in local or shared memory pointers |
| **cudaMemcpy2DAsync** | |
| **cudaMemcpy3DAsync** | |
| **cudaMemsetAsync** | |
| **cudaMemset2DAsync** | |
| **cudaMemset3DAsync** | |
| **cudaRuntimeGetVersion** | |
| **cudaMalloc** | May not call cudaFree on the device on a pointer created on the host, and vice-versa |
| **cudaFree** | |

Supported API functions

# DEVICE-SIDE LAUNCH FROM PTX

This section is for the programming language and compiler implementers who target *Parallel Thread Execution* (PTX) and plan to support Dynamic Parallelism in their language. It provides the low-level details related to supporting kernel launches at the PTX level.

## Kernel Launch APIs

Device-side kernel launches can be implemented using the following two APIs accessible from PTX: *cudaLaunchDevice()* and *cudaGetParameterBuffer()*. *cudaLaunchDevice()* launches the specified kernel with the parameter buffer that is obtained by calling *cudaGetParameterBuffer()* and filled with the parameters to the launched kernel. The parameter buffer can be NULL, i.e., no need to invoke *cudaGetParameterBuffer()*, if the launched kernel does not take any parameters.

### cudaLaunchDevice

At the PTX level, *cudaLaunchDevice()* needs to be declared in one of the two forms shown below before it is used.

```
// When .address_size is 64
.extern .func(.param .b32 func_retval0) cudaLaunchDevice
(
  .param .b64 func,
  .param .b64 parameterBuffer,
  .param .align 4 .b8 gridDimension[12],
  .param .align 4 .b8 blockDimension[12],
  .param .b32 sharedMemSize,
  .param .b64 stream
)
;
```

PTX-level declaration of *cudaLaunchDevice()* when .address_size is 64

```
// When .address_size is 32
.extern .func(.param .b32 func_retval0) cudaLaunchDevice
(
  .param .b32 func,
  .param .b32 parameterBuffer,
  .param .align 4 .b8 gridDimension[12],
  .param .align 4 .b8 blockDimension[12],
  .param .b32 sharedMemSize,
```

```
  .param .b32 stream
)
;
```

PTX-level declaration of *cudaLaunchDevice()* when .address_size is 32

The CUDA-level declaration below is mapped to one of the aforementioned PTX-level declarations and is found in the system header file *cuda_device_runtime_api.h*. The function is defined in the *cudadevrt* system library, which must be linked with a program in order to use device-side kernel launch functionality.

```
extern "C" __device__
cudaError_t cudaLaunchDevice(void *func, void *parameterBuffer,
                             dim3 gridDimension, dim3 blockDimension,
                             unsigned int sharedMemSize,
                             cudaStream_t stream);
```

CUDA-level declaration of *cudaLaunchDevice()*

The first parameter is a pointer to the kernel to be is launched, and the second parameter is the parameter buffer that holds the actual parameters to the launched kernel. The layout of the parameter buffer is explained in "Parameter Buffer Layout", below. Other parameters specify the launch configuration, i.e., as grid dimension, block dimension, shared memory size, and the stream associated with the launch (please refer to the CUDA Programming Guide for the detailed description of launch configuration, and of *cudaLaunchDevice()* specifically).

## cudaGetParameterBuffer

*cudaGetParameterBuffer()* needs to be declared at the PTX level before it's used. The PTX-level declaration must be in one of the two forms given below, depending on address size:

```
// When .address_size is 64
.extern .func(.param .b64 func_retval0) cudaGetParameterBuffer
(
  .param .b64 alignment,
  .param .b64 size
)
;
```

PTX-level declaration of *cudaGetParameterBuffer()* when .address_size is 64

```
.extern .func(.param .b32 func_retval0) cudaGetParameterBuffer
(
  .param .b32 alignment,
  .param .b32 size
)
;
```

PTX-level declaration of *cudaGetParameterBuffer()* when .address_size is 32

The following CUDA-level declaration of cudaGetParameterBuffer() is mapped to the aforementioned PTX-level declaration:

```
extern "C" __device__
void *cudaGetParameterBuffer(size_t alignment, size_t size);
```

CUDA-level declaration of *cudaGetParameterBuffer()*

The first parameter specifies the alignment requirement of the parameter buffer and the second parameter the size requirement in bytes. In the current implementation, the parameter buffer returned by *cudaGetParameterBuffer()* is always guaranteed to be 64-byte aligned, and the alignment requirement parameter is ignored. However, it is recommended to pass the correct alignment requirement value – which is the largest alignment of any parameter to be placed in the parameter buffer – to *cudaGetParameterBuffer()* to ensure portability in the future.

## Parameter Buffer Layout

Parameter reordering in the parameter buffer is prohibited, and each individual parameter placed in the parameter buffer is required to be aligned. That is, each parameter must be placed at the $n^{th}$ byte in the parameter buffer, where $n$ is the smallest multiple of the parameter size that is greater than the offset of the last byte taken by the preceding parameter. The maximum size of the parameter buffer is 4KB.

For a more detailed description of PTX code generated by the CUDA compiler, please refer to the PTX-3.5 specification.

# TOOLKIT SUPPORT FOR DYNAMIC PARALLELISM

## Including device runtime API in CUDA code

Similar to the host-side runtime API, prototypes for the CUDA device runtime API are included automatically during program compilation. There is no need to include *cuda_device_runtime_api.h* explicitly.

## Compiling and Linking

CUDA programs are automatically linked with the host runtime library when compiled with *nvcc*, but the device runtime is shipped as a static library which must explicitly be linked with a program which wishes to use it.

The device runtime is offered as a static library (*cudadevrt.lib* on Windows, *libcudadevrt.a* under Linux and MacOS), against which a GPU application that uses the device runtime must be linked. Linking of device libraries can be accomplished through *nvcc* and/or *nvlink*. Two simple examples are shown below.

A device runtime program may be compiled and linked in a single step, if all required source files can be specified from the command line:

```
$ nvcc -arch=sm_35 -rdc=true hello_world.cu -o hello -lcudadevrt
```

It is also possible to compile CUDA .cu source files first to object files, and then link these together in a two-stage process:

```
$ nvcc -arch=sm_35 -dc hello_world.cu -o hello_world.o
$ nvcc -arch=sm_35 -rdc=true hello_world.o -o hello -lcudadevrt
```

Please see the "Using Separate Compilation" section of "The CUDA Driver Compiler NVCC" guide for more details.

# PROGRAMMING GUIDELINES

## BASICS

The device runtime is a functional subset of the host runtime. API level device management, kernel launching, device memcpy, stream management, and event management are exposed from the device runtime.

Programming for the device runtime should be familiar to someone who already has experience with CUDA. Device runtime syntax and semantics are largely the same as that of the host API, with any exceptions detailed earlier in this document.

The following example shows a simple “Hello World” program incorporating dynamic parallelism:

```
#include <stdio.h>

__global__ void childKernel()
{
      printf("Hello ");
}

__global__ void parentKernel()
{
      // launch child
      childKernel<<<1,1>>>();
      if (cudaSuccess != cudaGetLastError()) {
            return;
      }

      // wait for child to complete
      if (cudaSuccess != cudaDeviceSynchronize()) {
            return;
      }

      printf("World!\n");
}

int main(int argc, char *argv[])
{
      // launch parent
      parentKernel<<<1,1>>>();
      if (cudaSuccess != cudaGetLastError()) {
            return 1;
      }

      // wait for parent to complete
      if (cudaSuccess != cudaDeviceSynchronize()) {
            return 2;
      }

      return 0;
}
```

This program may be built in a single step from the command line as follows:

```
$ nvcc -arch=sm_35 -rdc=true hello_world.cu -o hello -lcudadevrt
```

# PERFORMANCE

## Synchronization

Synchronization by one thread may impact the performance of other threads in the same Thread Block, even when those other threads do not call *cudaDeviceSynchronize()* themselves. This impact will depend upon the underlying implementation.

## Dynamic-parallelism-enabled Kernel overhead

System software which is active when controlling dynamic launches may impose an overhead on any kernel which is running at the time, whether or not it invokes kernel launches of its own. This overhead arises from the device runtime's execution tracking and management software and may result in decreased performance for e.g. library calls when made from the device compared to from the host side. This overhead is, in general, incurred for applications that link against the device runtime library.

# IMPLEMENTATION RESTRICTIONS & LIMITATIONS

Dynamic Parallelism guarantees all semantics described in this document, however, certain hardware and software resources are implementation-dependent and limit the scale, performance and other properties of a program which uses the device runtime.

## Runtime

### Memory Footprint

The device runtime system software reserves memory for various management purposes, in particular one reservation which is used for saving parent-grid state during synchronization, and a second reservation for tracking pending grid launches. Configuration controls are available to reduce the size of these reservations in exchange for certain launch limitations. See *Configuration Options*, below, for details.

The majority of reserved memory is allocated as backing-store for parent kernel state, for use when synchronizing on a child launch. Conservatively, this memory must support storing of state for the maximum number of live threads possible on the device. This means that each parent generation at which *cudaDeviceSynchronize()* is callable may

require up to 150MB of device memory, depending on the device configuration, which will be unavailable for program use even if it is not all consumed.

## Nesting & Synchronization Depth

Using the device runtime, one kernel may launch another kernel, and that kernel may launch another, and so on. Each subordinate launch is considered a new "nesting level", and the total number of levels is the "nesting depth" of the program. The "synchronization depth" is defined as the deepest level at which the program will explicitly synchronize on a child launch. Typically this is one less than the nesting depth of the program, but if the program does not need to call *cudaDeviceSynchronize()* at all levels then the synchronization depth might be substantially different to the nesting depth.

The overall maximum nesting depth is limited to 24, but practically speaking the real limit will be the amount of memory required by the system for each new level (see *Memory Footprint* above). Any launch which would result in a kernel at a deeper level than the maximum will fail. Note that this may also apply to *cudaMemcpyAsync()*, which might itself generate a kernel launch. See *Configuration Options*, below, for details.

By default, sufficient storage is reserved for two levels of synchronization. This maximum synchronization depth (and hence reserved storage) may be controlled by calling *cudaDeviceSetLimit()* and specifying *cudaLimitDevRuntimeSyncDepth*. The number of levels to be supported must be configured before the top-level kernel is launched from the host, in order to guarantee successful execution of a nested program. Calling *cudaDeviceSynchronize()* at a depth greater than the specified maximum synchronization depth will return an error.

An optimization is permitted where the system detects that it need not reserve space for the parent's state in cases where the parent kernel never calls *cudaDeviceSynchronize()*. In this case, because explicit parent/child synchronization never occurs, the memory footprint required for a program will be much less than the conservative maximum. Such a program could specify a shallower maximum synchronization depth to avoid over-allocation of backing store.

## Pending Kernel Launches

When a kernel is launched, all associated configuration and parameter data is tracked until the kernel completes. This data is stored within a system-managed launch pool. The size of the launch pool is configurable by calling *cudaDeviceSetLimit()* from the host and specifying *cudaLimitDevRuntimePendingLaunchCount*.

## Configuration Options

Resource allocation for the device runtime system software is controlled via the *cudaDeviceSetLimit()* API from the host program. Limits must be set before any kernel is launched, and may not be changed while the GPU is actively running programs.

The following named limits may be set:

| Limit | Behaviour |
|---|---|
| **cudaLimitDevRuntimeSyncDepth** | Sets the maximum depth at which cudaDeviceSynchronize() may be called. Launches may be performed deeper than this, but explicit synchronization deeper than this limit will return the *cudaErrorLaunchMaxDepthExceeded*. The default maximum sync depth is 2. |
| **cudaLimitDevRuntimePendingLaunchCount** | Controls the amount of memory set aside for buffering kernel launches which have not yet begun to execute, due either to unresolved dependencies or lack of execution resources. When the buffer is full, launches will set the thread's last error to *cudaErrorLaunchPendingCountExceeded*. The default pending launch count is 2048 launches. |

## Memory Allocation and Lifetime

*cudaMalloc()* and *cudaFree()* have distinct semantics between the host and device environments. When invoked from the host, *cudaMalloc()* allocates a new region from unused device memory. When invoked from the device runtime these functions map to device-side *malloc()* and *free()*. This implies that within the device environment the total allocatable memory is limited to the device *malloc()* heap size, which may be smaller than the available unused device memory. Also, it is an error to invoke *cudaFree()* from the host program on a pointer which was allocated by *cudaMalloc()* on the device or vice-versa.

| | **cudaMalloc() on Host** | **cudaMalloc() on Device** |
|---|---|---|
| **cudaFree() on Host** | Supported | Not Supported |
| **cudaFree() on Device** | Not Supported | Supported |
| **Allocation limit** | Free device memory | cudaLimitMallocHeapSize |

## SM Id and Warp Id

Note that in PTX %smid and %warpid are defined as volatile values. The device runtime may reschedule thread blocks onto different SMs in order to more efficiently manage resources. As such, it is unsafe to rely upon %smid or %warpid remaining unchanged across the lifetime of a thread or thread block.

## ECC Errors

No notification of ECC errors is available to code within a CUDA kernel. ECC errors are reported at the host side once the entire launch tree has completed. Any ECC errors which arise during execution of a nested program will either generate an exception or continue execution (depending upon error and configuration).