CUDA TOOLKIT 4.0 READINESS FOR CUDA APPLICATIONS

March 3, 2011

Technical Brief

TECHNICAL BRIEF

INTRODUCTION

In NVIDIA® CUDA™ Toolkit version 4.0, a major emphasis has been placed on improving the programmability of multi-threaded and multi-GPU applications and on improving the ease of porting existing code to CUDA C/C++. This document describes the key API changes and improvements that have been made toward that end, particularly where they have the potential to impact existing applications.

This document also highlights a few of the improvements that have been made to the libraries bundled with the CUDA Toolkit.

For a complete listing of new features, please refer to the CUDA Toolkit Release Notes; for complete documentation of CUDA APIs, please refer to the CUDA Toolkit Reference Manual, the CUDA C Programming Guide, and the CUDA library documentation.

MULTI-GPU PROGRAMMING

In CUDA Toolkit 3.2 and earlier, there were two basic approaches available to execute CUDA kernels on multiple GPUs (CUDA "devices") concurrently from a single host application:

- Use one host thread per device, since any given host thread can call cudaSetDevice() at most one time.
- Use the push/pop context functions provided by the CUDA Driver API.

For applications that do not require tight coupling of the various CUDA devices within a system (e.g., when the devices are processing independent data sets with little need to communicate or synchronize with each other), these approaches were often sufficient.

However, for applications that need a tighter coupling of the execution of work across devices, or where the use of multiple host threads is otherwise inconvenient, it would sometimes be better if a single host thread could easily launch work onto any devices it needed.

CUDA Runtime API

The CUDA Runtime now provides a native means to accomplish this: a host thread can simply call cudaSetDevice() at any time (rather than just once at the beginning of the program) to change the currently active device. This has the following consequences:

- Kernel launches will be executed on the currently selected device.
- Memory allocations will be made on the currently selected device.
- Streams and events created will be associated with the currently selected device.[1]

For example:

```
cudaSetDevice(0);       // start on device 0
cudaMalloc(&p0, size); // allocate memory for p0 on device 0
K0<<<1,1>>>(p0);        // launch kernel K0 on device 0

cudaSetDevice(1);       // switch to device 1 (Not legal in CUDA 3.x!)
cudaMalloc(&p1, size); // allocate memory for p1 on device 1
K1<<<1,1>>>(p1);        // launch kernel K1 on device 1
```

Also, note that since a host thread can now control more than one device, the `cudaThread*()` functions (e.g., `cudaThreadSynchronize()`) no longer have names that match their functionality. For this reason, these functions have been replaced with `cudaDevice*()` functions of similar names and functionality (e.g., `cudaThreadSynchronize()` is now `cudaDeviceSynchronize()`). The old function names remain but are now deprecated; code using the old names will trigger a compile-time warning.

[1] This restriction might be removed in a future version of the CUDA Toolkit.

CUDA Driver API

While the CUDA Driver API prior to version 4.0 already provided a way to access multiple devices from within a single host thread (namely `cuCtxPushCurrent()` and `cuCtxPopCurrent()`), feedback from CUDA application developers showed that a set/get context management interface would in some cases be more convenient than the push/pop stack-based paradigm.

Consider the case where a given host thread needs to cycle through a set of devices. With the stack-based interface, operating on device 0 and then device 1 requires two pairs of API calls: `push(ctx0); pop(ctx0); push(ctx1); pop(ctx1).` With a set/get interface, this would become simply `set(ctx0); set(ctx1).`

With this in mind, `cuCtxSetCurrent()` and `cuCtxGetCurrent()` have been added to version 4.0 of the CUDA Driver API in addition to the existing `cuCtxPushCurrent()` and `cuCtxPopCurrent()` functions.

Unified Virtual Addressing and GPUDirect™ v2.0

Additional features have been added to CUDA Toolkit version 4.0 to ease programmability of multi-GPU environments for NVIDIA Fermi GPUs running in 64-bit mode on Linux or on Windows in TCC mode.

Unified Virtual Addressing (UVA) allows the system memory and the one or more device memories in a system to share a single virtual address space. This allows the CUDA Driver to determine the physical memory space to which a particular pointer refers by inspection, which simplifies the APIs of functions such as `cudaMemcpy()`, since the application need no longer keep track of which pointers refer to which memory.

Built on top of UVA, GPUDirect v2.0 provides for direct peer-to-peer communication among the multiple devices in a system.

Inter-Device Coupling

The `cudaStreamWaitEvent()` function now allows synchronizing across streams from *different* contexts, including streams for different devices. In conjunction with the features listed above, this allows for much simpler inter-device coupling.

For example, consider an iterative application that executes kernels on two or more devices, synchronizes the devices, exchanges the boundary (halo) data among the devices, synchronizes again, and then repeats. Prior to version 4.0, this would have required separate host threads (one per device), and it would have required those host threads to somehow synchronize with each other as well as to arrange the inter-context data transfer. With version 4.0, however, a single host thread can accomplish this quite easily with a simple `for()` loop:

```
for (int i=0; i<iterations; i++)
{
    // Launch kernels on each device and record events to let
    // us know when the kernels launches complete
    cudaSetDevice(0);
    kernel<<<grid, block, 0, stream0>>>(cells0, halo0, halo1dst);
    cudaEventRecord(event0, stream0);
    cudaSetDevice(1);
    kernel<<<grid, block, 0, stream1>>>(cells1, halo1, halo0dst);
    cudaEventRecord(event1, stream1);

    // Synchronize the devices with each other
    cudaStreamWaitEvent(stream0, event1);
    cudaStreamWaitEvent(stream1, event0);

    // Set up asynchronous copies to transfer the boundary (halo)
    // data from each device to the other, and again record events
    // to signal completion.  Note: the use of cudaMemcpyAsync() in
    // this way instead of cudaMemcpyPeerAsync() assumes UVA support.
    cudaMemcpyAsync(halo1dst, halo1, size, cudaMemcpyDefault, stream0);
    cudaMemcpyAsync(halo0dst, halo0, size, cudaMemcpyDefault, stream1);
    cudaEventRecord(event0, stream0);
    cudaEventRecord(event1, stream1);

    // Synchronize the devices with each other
    cudaStreamWaitEvent(stream0, event1);
    cudaStreamWaitEvent(stream1, event0);
}
```

MULTI-THREADED PROGRAMMING

Another common scenario is when multiple host threads in a multi-threaded application want to share access to a single device. In version 4.0, major changes have gone in to both the CUDA Runtime and the CUDA Driver to improve this:

CUDA Runtime API

Prior to version 4.0 of the CUDA Runtime, each host thread accessing a particular device would get its own context to (view of) that device. As a consequence, it was not possible to share memory objects, events, and so forth across host threads, even when they were referring to the same device.

In version 4.0, host threads within a given process that access a particular device automatically share a single context to that device, rather than each having its own context. In other words, the new model for runtime applications is *one context per device per process*.

As an example, consider the following code listing:

Host Thread A	Host Thread B
`cudaSetDevice(0);`	`cudaSetDevice(0);`
`cudaMalloc(&p0, size);`	
`pthread_barrier_wait(b);` 	`pthread_barrier_wait(b);`
	`K0<<<1,1>>>(p0);`
	`cudaFree(p0);`

This example is an invalid program with version 3.2 and earlier, because host threads `A` and `B` would have separate contexts and consequently separate address spaces, so `p0` would be an invalid pointer in thread `B`. This is a valid program in version 4.0, however, because host threads A and B will use the same context.

This has several important ramifications for multi-threaded processes:

- Host threads can now share device memory allocations, streams, events, or any other per-context objects (as seen above).
- Concurrent kernel execution on devices of compute capability 2.x is now possible across host threads, rather than just within a single host thread. Note that this requires the use of separate streams; unless streams are specified, the kernels will be executed sequentially on the device in the order they were launched. In all cases, kernel launch via the `<<<>>>` notation is a thread-safe operation.
- `cudaGetLastError()` is per-host-thread: it returns the last error returned by an API call in that host thread, even if other host threads are concurrently accessing the same device.
- **IMPORTANT:** Host threads now share __global__, __constant__, and other file-scoped variables within `.cu` files. **If an application previously depended on each host thread getting its own instance of these variables, *this assumption is now broken*.** (As a workaround, such applications could explicitly create separate contexts for each host thread by calling `cuCtxCreate()` at the beginning of each host thread.)
- **IMPORTANT: If one host thread calls `cudaThreadExit()`, the context will be torn down immediately, *even if other host threads are still using it*.** It is the application's responsibility to ensure that all host threads are finished with the device before calling this function. If any CUDA Runtime API calls are made subsequent to `cudaThreadExit()`, a new context will be created, and any references to objects created in the earlier context will be invalid. To make it more clear that this is the expected behavior of this function, `cudaThreadExit()` has been renamed to `cudaDeviceReset()` (the older function name still exists but is now deprecated). This issue is of particular importance because in prior versions of the CUDA Toolkit, the recommended best practice was for an application to explicitly call `cudaThreadExit()` when cleaning itself up prior to exit. The general idea remains valid: an application should, as a best practice, call either `cudaDeviceSynchronize()` or `cudaDeviceReset()` prior to terminating (allowing the flushing of profiler buffers and other graceful

cleanups to occur); however, if `cudaDeviceReset()` is used for this, only *one* host thread per device should call it.

CUDA Driver API

While the CUDA Driver API prior to version 4.0 provided a way to hand off a context from one host thread to another, this still did not allow for *concurrent* access to that context from multiple host threads, in part because the CUDA Driver's interface for launching kernel grids was stateful and therefore not thread-safe.

In version 4.0, it is now legal for multiple host threads to set a particular context current simultaneously using either `cuCtxSetCurrent()` or `cuCtxPushCurrent()`. This has several important ramifications for multi-threaded processes:

- Host threads can now share device memory allocations, streams, events, or any other per-context objects (as seen above).
- Concurrent kernel execution devices of compute capability 2.x is now possible across host threads, rather than just within a single host thread. Note that this requires the use of separate streams; unless streams are specified, the kernels will be executed sequentially on the device in the order they were launched.
- **IMPORTANT:** For thread-safety, host threads launching kernels in the same context concurrently **must** use the new thread-safe *stateless launch* API function `cuLaunchKernel()`, which takes the place of the more verbose earlier API (i.e., `cuParamSet*()` + `cuFuncSetBlockShape()` + `cuFuncSetSharedSize()` + `cuLaunchGrid()`). Note that with this new API, kernel grid launches in the CUDA Driver API more closely resemble kernel launches via the <<<>>> syntax of the CUDA Runtime API.

Compute-exclusive mode for multi-threaded applications

The "compute-exclusive mode" for a device (settable via `nvidia-smi`) previously allowed no more than one context used by a single thread of one process to access the device at any given time. This mode has been renamed to "compute-exclusive-thread mode" and retains the same functionality. A new mode named "compute-exclusive-process mode" has been added wherein the threads of a single *process* (sharing one context) can access the device at any particular time.

PORTING CODE TO CUDA C/C++

Following are a few of the key features that have been added to CUDA Toolkit 4.0 that can help improve the porting of existing applications to CUDA C or CUDA C++:

- CUDA C++ now supports `operator new` and `operator delete` on devices of compute capability 2.x.
- Classes in CUDA C++ now support virtual functions on devices of compute capability 2.x.
- CUDA C/C++ now support the inlining of PTX assembly into device code using the `asm()` construct.
- The CUDA Driver and CUDA Runtime APIs now support no-copy, after-the-fact pinning of arbitrary buffers in system memory. In other words, the decision to pin the memory using the CUDA APIs no longer needs to be made at the time the buffer is allocated (using `cudaHostAlloc()`). This decoupling of allocation from pinning should be helpful in applications where memory allocation is done well ahead of the point at which the decision to copy the contents of that memory to the device is made.
- CUDA C/C++ now support *layered textures*, which are a type of texture that holds a group of some other type of texture. Note that the various layers of the layered texture must be textures of the same type, size and format and that filtering is done only *within* each layer and not *across* layers.
- Kernel launches using the CUDA Runtime's <<<>>> and the new stateless launch API in the CUDA Driver now both support three-dimensional grids of thread blocks on Fermi GPUs, which should simplify the porting of algorithms that naturally map to 3D domains to CUDA C/C++.

CUDA TOOLKIT LIBRARIES

- The CUBLAS library now supports a new API that is thread-safe and allows the application to more easily take advantage of parallelism using streams, especially for functions with scalar return parameters. This new API allows CUBLAS to work cleanly with applications using the new multi-threading features of CUDA Runtime 4.0. The legacy CUBLAS API is still supported, but it is not thread-safe and does not offer as many opportunities for parallelism with streams as the new API.
- **IMPORTANT:** The CUFFT library is *not* yet thread-safe and therefore cannot safely access the same device context from multiple host threads concurrently. This restriction will be removed in a future release of the CUDA Toolkit.
- The CURAND library now supports double precision Sobol, scrambled Sobol, log-normal distributions, and a faster setup technique for XORWOW.

- The CUFFT and CUBLAS library APIs now include functions that will report the library's version number.
- The CUSPARSE library now provides a solver for triangular sparse linear systems via the `cusparse*csrsv_analysis()` and `cusparse*csrsv_solve()` API functions.
- The Thrust template library and the NPP image processing library are now bundled with the CUDA Toolkit, with no additional download required.
- Some API functions in the NPP library were changed to pass results via device pointer instead of via host pointer for consistency with all of the rest of the NPP API.

Notice

Trademarks

Copyright