GPU-to-CPU Callbacks

Jeff A. Stuart[1], Michael Cox[2], and John D. Owens[1]

[1] University of California, Davis
[2] NVIDIA Corporation

Abstract. We present GPU-to-CPU callbacks, a new mechanism and abstraction for GPUs that offers them more independence in a heterogeneous computing environment. Specifically, we provide a method for GPUs to issue callback requests to the CPU. These requests serve as a tool for ease-of-use, future proofing of code, and new functionality. We classify the types of these requests into three categories: System calls (e.g. network and file I/O), device/host memory transfers, and CPU compute, and provide motivation as to why all are important. We show how to implement such a mechanism in CUDA using pinned system memory and discuss possible GPU-driver features to alleviate the need for polling, thus making callbacks more efficient with CPU usage and power consumption. We implement several examples demonstrating the use of callbacks for file I/O, network I/O, memory allocation, and debugging.

1 Introduction

Microprocessor architectures are becoming ever more complicated. Even now, the world is moving in the direction of System-on-a-Chip (SoC), wherein the CPU and many specialized cores reside on one common die. With an SoC, systems will inevitably contain more cores than can be concurrently powered. The heat and wattage demands will be such that only a fraction of the cores can run at a time. There is no reason the CPU should not be able to idle or power off while other cores stay functional. Right now, this is impossible because the CPU controls many functions of the machine and is the only core to do so.

There is no insurmountable challenge to having another core, such as the GPU, be capable of managing aspects of a machine. In fact, we contend that the GPU should be able to access hard drives, communicate with network controllers, make other various system calls, and even wake up a sleeping CPU. As of right now, the graphics driver does not allow this, and would require a significant amount of work from driver architects to properly implement.

For now, the concept of a machine controlled by the GPU is attractive, if for no other reason than a CPU often need only sit idle while the GPU is working. In fact, users often split long GPU kernels into multiple smaller kernels simply because CPU intervention is necessary for one reason or another, typically to make a system call or transfer memory from the CPU to the GPU.

In this paper, we detail a new mechanism that allows the GPU to execute system calls and control the CPU, and we provide a library that wraps this functionality via callbacks. By using GPU-to-CPU callbacks, our library exposes a mechanism that allows the GPU to request computation from the CPU, and that allows these tasks to execute concurrently with GPU kernels. This mechanism demonstrates the new model of GPU programming that allows the GPU to execute system calls and control the machine.

In our current implementation, callbacks most often do *not* yield performance improvements, nor are they meant to. We implemented callbacks for two reasons: to make code future-proof (as drivers and systems advance, update the callback library and code runs more efficiently) and to make certain tasks easier, especially those that use either concurrent compute and memory transfers, or concurrent CPU and GPU compute. We predict that performance improvements will come as the GPU driver and system architecture evolve.

2 Existing Methods

Callbacks can used in two ways: the GPU requests some data/computation from the CPU, or the GPU feeds debugging/checkpointing information to the CPU. The GPU currently does not request work from the CPU because there are few mechanisms by which to do so. Instead, most kernels take the data they need, perform their computation, and leave results in GPU memory. Even though there are times when logically it makes more sense for a GPU to request work or data from a CPU, programmers do not write applications this way due to the restrictions of the GPU.

The second general way in which callbacks can be used is debugging. Many GPU-debugging tools have been announced recently, including a fully capable GDB port from NVIDIA called CUDA-GDB [3]. However, CUDA-GDB does not address the problems that arise through dataflow, when a bug in one kernel results in erroneous data used by other kernels, which eventually causes a later kernel to perform incorrectly or even crash. In 2009, Hou et al. used dataflow recording to debug GPU stream programs [2]. This was a solid start to debugging large applications, but it required users to be able to mark their data, fit all debugging information on the GPU along with the data, and interpret a visualization to understand exactly where a problem occurred.

To further address the deficiencies in debugging, NVIDIA ported *printf* functionality to CUDA as *cuPrintf* [4], which lets developers use console output. This eased the burden of debugging for many but had the drawback that results were only printed to *stdout* upon kernel completion. With long-running kernels and kernels that crash the GPU, *cuPrintf* might not function properly.

Stuart and Owens [5] created a mechanism to let the GPU execute system calls using their library DCGN. DCGN relied on concurrent, asynchronous memory copies from the GPU to the CPU, which are not possible on compute-1.0 architectures, are not supported in OpenCL, and are never guaranteed to execute asynchronously. Furthermore, all calls to the CPU in this work are hard-coded, precluding implementions of new system-call requests without heavily modifying the underlying library.

3 Implementation

We implemented callbacks with 0-copy memory and a polling mechanism. Polling goes against our motivation of having idle CPUs, but we expect driver innovations to mitigate this. When issuing a callback, the GPU puts a callback request in 0-copy memory and sets a flag to alert the CPU of said request. The kernel then either continues with its work, busy-waits on the callback, or finishes execution and examines the return value of the callback with a future kernel. On the CPU, our library handles callbacks via polling (modern GPU drivers do not expose any interrupt/signal capabilities). Once a kernel is invoked, the user calls one of two functions. Figure 1 shows a basic overview of how the CPU and GPU interact via callbacks. Calling *callbackSynchronize* ceases all CPU execution except for handling callbacks until both the kernel has finished executing and all callbacks have completed. The other function, *callbackQuery(cudaStream_t)*, polls the status of the kernel and any active/pending callbacks, but does not block. A successful return value from *callbackQuery* implies all callbacks completed and all kernels finished executing. We provide more details below.

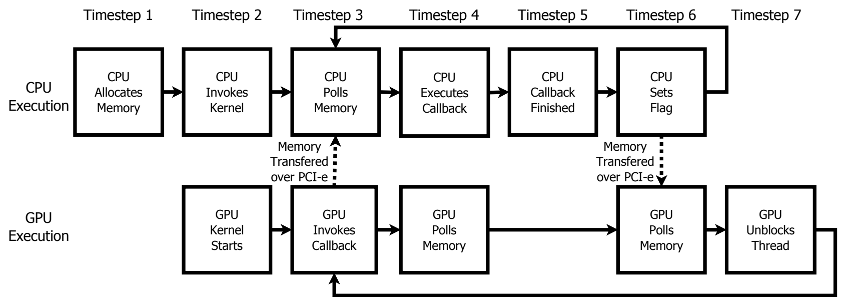

Fig. 1. An overview of callbacks and the CPU-GPU interactions they create. The CPU initializes the callback library and allocates the pool of 0-copy memory. It then invokes a kernel and executes *callbackSynchronize*, and the GPU starts the kernel. Next, the GPU invokes a synchronous callback and the CPU polls 0-copy memory. Then, the GPU halts and the CPU executes the callback. Next, the callback finishes. Now, the CPU sets a 0-copy memory flag to tell the GPU that it handled the callback request. Finally, the GPU unblocks the calling thread and continues its execution.

3.1 CPU Implementation

Our library provides an API to register callbacks and issue them within a kernel, as well as a backend that handles callbacks. The user initializes the library by specifying how many concurrent callback requests the GPU may issue, the maximum number of parameters for any callback, and the number of CPU threads to handle callback requests. The library creates a CPU-thread pool and allocates enough 0-copy memory to hold all concurrent callback requests.

Once the library finishes initialization, it is ready to register callbacks. To do so, a user first creates a data structure that contains information about the callback (parameter type(s), return type, and function address), then registers the callback with the library and receives a unique handle for use on the GPU (which can be passed as a kernel argument). After launching all kernels, the user can either sleep wait or periodically check on the kernels. Both of these must be done using library-supplied functions and not the functions provided by CUDA.

When the CPU detects a callback request, it passes the parameters from 0-copy memory to the user-specified callback function. Once completed, the callback returns a value which the library gives to the GPU, and sets a flag to tell the GPU the callback completed. This method works well, even with many callbacks (on the order of hundreds) in flight.

3.2 GPU Implementation

The user may invoke any number of callbacks in a kernel (though the number of concurrent callbacks is limited). To do so, any GPU thread invokes *callbackExecute* (synchronous callback request) or *callbackExecuteAsync* (asynchronous callback request), passing the callback handle and callback parameters. Every thread that invokes either of these two functions will trigger a callback, so a conditional is needed when the user only wants specific GPU threads to execute a callback. The parameters are loosely typed; the compiler will not issue any warnings when the user passes an incorrect number of arguments, nor when the parameter types require strong coercion (e.g. *int* to *float*), which is handled implicitly by the library.

When a GPU thread executes a synchronous callback, the GPU thread spins until the request completes. For performance, we suggest using the asynchronous version of callbacks whenever possible. If the user executes an asynchronous callback, the invokation function returns a handle the user, in their GPU code, may query or block upon in a manner similar to *cudaStreamQuery* and *cudaStreamSynchronize* (again, these function are host functions, though the functions we provide are device functions). This allows the CPU and GPU to overlap work without stalling the GPU.

When a user invokes a callback, the library coerces the parameters to their correct types and moves them to 0-copy memory. The function identifier is also set in 0-copy memory, and then a *threadfence()* is executed to ensure that the writes to 0-copy memory are flushed. The library returns a handle for querying or waiting. When the user executes a synchronous callback, the function implicitly waits on the handle until the callback is complete. Blocking works by spinning on a region of 0-copy memory until the CPU signals that the specific callback is complete.

To guarantee an upper bound on the number of concurrent callbacks, the library simply uses a queue of freely available "slots." We borrow the notion of slots from DCGN, but instead of explicitly requiring a slot identifier, the library uses an atomicly guarded queue to obtain the next available slot and mark previously unavailable slots as available again. When no slots are available, the library will spin wait. We experimented with a version of callbacks that required explicit slot identifiers (similar to DCGN) but found that slots make sense for communication patterns in DCGN but are often cumbersome and unnecessary for general-purpose callbacks.

4 Tests

We categorize callbacks into three types of combinable requests: system calls, overlapping CPU and GPU compute, and overlapping compute and memory transfers. The most interesting to us is system calls because it shows how programmers will leverage the GPU when it gains such an ability and no longer requires a user-level library to mimic that ability. System calls also allow for command line I/O such as *printf*, a very useful tool during the development and debugging cycle of any application.

A useful callback library must have three characteristics: an ability to scale, an acceptable response time, and be easy to use. Scalability is important because a GPU has many thread processors and the situation may arise that each thread processor has an outstanding asynchronous (or synchronous) callback. As such, it is important for the library to work well when dealing with one callback request or tens-to-hundreds of callback requests. An acceptable response time is important because each millisecond spent on a callback represents numerous FLOPs of wasted compute. And like any library, ease of use is important.

We wrote several applications to demonstrate callbacks, our library, and a proper implementation of each of the above-mentioned requirements, and touch on three of them: a TCP/IP client and server, a CPU-side memory manager for the GPU, and a command-line debugging tool that uses *printf*. All applications use callbacks and are straightforward to write.

For our tests, we used a GTX 280 running in a machine with a 2.5 GHz Intel Core 2 Quad and 4 GB of RAM. The machine is running the 2.6.18 Linux kernel, GCC 4.1.2, the 2.3 CUDA Toolkit, and the NVIDIA 190.53 GPU driver.

4.1 TCP/IP

We wrote a TCP/IP client and server to demonstrate the response time of our library. Both the client and a server operate in the same kernel[3]. The configuration uses two blocks with one thread each. This guarantees each thread is run in a different warp and thus does not deadlock from warp-divergence stalls. The first thread uses callbacks to create a server socket, accept an incoming connection, receive a string, and close both the server socket and the receiving socket. The second thread uses callbacks to create a socket by connecting to the server,

[3] On our test GPU, this was necessary as the GPU only runs a single kernel at a time. This test also is beneficial as it shows one way to perform interblock communication, though obviously one would avoid network I/O for such a thing in a real application.

send a string, and close the socket. For reference, the device code for the server is shown in Figure 2. This code is representative of most callback code, so for brevity we only show the code for this test and not others.

```
__device__ void serverKernel(callbackData_t * cdata, callbackFunction_t * funcs)
{
  int serverSocket  = callbackExecute<int>(cdata, funcs[START_SERVER_CALLBACK], SERVER_PORT);
  int recvSocket    = callbackExecute<int>(cdata, funcs[ACCEPT_CALLBACK]);
  int bytesRead     = callbackExecute<int>(cdata, funcs[RECV_CALLBACK],  recvSocket, 0, 1024, 0);
                      callbackExecute<int>(cdata, funcs[CLOSE_CALLBACK], serverSocket);
                      callbackExecute<int>(cdata, funcs[CLOSE_CALLBACK], recvSocket);
}
```

Fig. 2. The GPU callback code to create a TCP server. *callbackExecute* requires a template parameter for the callback return type, a pointer to a *callbackData_t* structure that contains information about all callbacks registered before kernel invokation, the handle for the callback to be invoked, and the parameters to the callback.

This test is important because it shows callback performance when there is little contention, at most two callbacks run concurrently. The GPU version of this test executes in approximately 2.5 ms, while the CPU version executes in approximately 1 ms. The slowdown can be attributed to several things, but the primary reasons are thread-safety mechanisms in the callback library and the overhead of polling 0-copy memory with active callbacks.

4.2 Memory Manager

We implemented a primitive memory manager and matrix multiplication to showcase the effectiveness of our library at scale. We make a certain number of "pages" available to any number of blocks. Each block uses a callback to request an adequately-sized page to hold three large, square matrices from the CPU. Once the CPU responds, the block generates two random matrices, stores the product of the two, then issues a callback to free the page (the result is essentially ignored). It is possible that every block is stalled waiting for the CPU to return a page, thus providing details on our library's performance with many concurrent callbacks. While this test is trivial and the GPU often would not work on random data, the test is important because it grants a wish for many GPU programmers: dynamic memory allocation from the GPU during runtime[4]. Because a kernel can request new memory while running, and the request goes to the CPU, users are free to implement a heap as complicated as necessary to accomplish their tasks. This is prohibitively difficult without callbacks.

[4] CUDA does not allow any memory management while a kernel or asynchronous memory copy is running. Many writers of GPU renderers have stated that being able to use more complex data structures and dynamic memory on the GPU would be beneficial.

We varied the total number of blocks from 30 to 20000 and the maximum number of concurrently scheduled blocks from 30 to 120 (we did this by using various sizes of shared memory). Our *allocatePage* and *deallocatePage* callbacks each required the use of a lock to either get the next available page or free a page. The locks affected performance in the worst case, but only slightly. Our average time to service a callback was under 4 ms, the minimum was under 2 ms, and the maximum was approximately 10 ms.

4.3 Debugging

The last test we wrote was one for debugging, something important with either a long-running kernel producing erroneous results, or a kernel that crashes. In either case, the programmer absolutely must find the problem(s) and fix their code. The current methods are 1) allocate GPU memory for debug information that the CPU consumes and displays (Hou et al. did this for stream dataflow debugging), which is very similar to 2) breaking up a kernel into many smaller kernels and inspect certain GPU variables after each kernel completes, which is also similar to 3) removing pieces of a kernel, from last line to first line, until the last executed step is found to be the offending step, 4) use cuda-gdb, which often times does not behave properly when a GPU crashes, or 5) use *cuPrintf*, which only prints to the console upon kernel completion and offers no guarantee of proper functionality if a kernel crashes before completion or if the user executes too many *cuPrintf* statements.

All these options have drawbacks; they are time-consuming to implement, not guaranteed to work, or require significant refactoring of code. Callbacks offer a simple method: the user simply inserts synchronous callbacks so the CPU code will immediately print to the console. The GPU warp halts until the CPU outputs to the console (and if desired, a global barrier can halt the entire GPU), guaranteeing the proper execution of *printf* before the GPU crashes. In fact, even if the GPU crashes, as long as the callback buffers remain uncorrupted, the library will still issue the callback, even after it detects a GPU crash.

We would like to point out some notable tradeoffs in using a *synchronous* callback to execute *printf*, and using *cuPrintf*. *cuPrintf* will execute quickly, because it only writes straight to GPU memory, whereas, on a discrete GPU, the CPU must issue many PCI-e transactions to get the callback's payload. However, we contend that this is fine because of the other tradeoff, *cuPrintf* is delayed until the end of kernel execution, whereas even *synchronous* callbacks execute concurrently with the kernel.

To demonstrate this, we wrote a test wherein each block has a preassigned memory space on the GPU. The block generates two random matrices and multiplies them, then invokes a callback to the print out the determinant of the result. This kernel, when ran with sufficiently many blocks, takes several minutes to complete. During execution, we can see the progress of the GPU as each printed determinant tells us that another block has finished execution.

5 Conclusion

As we have shown, callbacks offers several advantages to users. Perhaps the most important advantage is that code that uses our library is now future-proof (all one must do is use an updated version of the library that takes advantage of new GPU advancements). As drivers (software) and system architectures (hardware) progress, kernels that are split or use modified algorithms to work around deficiencies in the GPU hardware and driver require modification. Code that takes advantage of callbacks simply requires an update to the callback library to use the new features of the driver and system. This is both in terms of hardware advancements and software advancements.

We see several avenues for advancement. For hardware, the important advancement is that of a more tightly-integrated and powerful (not in of FLOPs but in system control) GPU. AMD is on this path with their Fusion [1] processors; putting a CPU and GPU on the same die. This is great as it lowers memory latency (especially 0-copy) and paves the way for more advancements. It is in these advancements that we hope to see a GPU that can control the machine and put CPU cores to sleep. In terms of software advancement, a mechanism that allows the GPU to send signals or interrupts to the CPU would most benefit callbacks. This is useful as it allows the CPU to be put to sleep and woken up when either the GPU issues a callback or completes a kernel.

Acknowledgements

We would like to say thank you to Dominik Göddeke, the SciDAC Institute for Ultrascale Visualization and the National Science Foundation (Award 0541448), and to NVIDIA for equipment donations.

References

1. Advanced Micro Devices: Coming soon: The AMD fusion family of APUs, `http://sites.amd.com/us/fusion/APU/Pages/fusion.aspx`
2. Hou, Q., Zhou, K., Guo, B.: Debugging GPU stream programs through automatic dataflow recording and visualization. ACM Transactions on Graphics 28(5), 153:1–153:11 (Dec 2009)
3. NVIDIA Corporation: CUDA-GDB: The NVIDIA CUDA debugger (2008), `http://developer.download.nvidia.com/compute/cuda/2_1/cudagdb/CUDA_GDB_User_Manual.pdf`
4. NVIDIA Corporation: NVIDIA CUDA compute unified device architecture programming guide (Feb 2010), `http://developer.nvidia.com/cuda`
5. Stuart, J.A., Owens, J.D.: Message passing on data-parallel architectures. In: Proceedings of the 23rd IEEE International Parallel and Distributed Processing Symposium (May 2009), `http://www.idav.ucdavis.edu/publications/print_pub?pub_id=959`