GPU-to-CPU Callbacks

Jeff Stuart and John D. Owens
University of California, Davis

Michael Cox
NVIDIA Corp.

Motivation & Background

System-on-a-Chip Architecture
Microprocessor architectures are becoming ever-more complex. Even now, the trend is towards System-on-a-Chip [1], wherein many components of a standard computer (e.g. CPU, GPU, NIC) are integrated onto a single die. However, the CPU remains as the sole controller of the machine as a whole.

Intuitive Workflow
The GPU has a very specific method of programming. Users create kernels, CPU code launches those kernels. But sometimes, code would be more intuitive to write if the GPU had (pseudo-)access to system calls e.g. file I/O.

Debugging
Debugging is a hard problem on the GPU. Hou et al. [2] used a visual method for debugging, while NVIDIA Corp. [3] produced a GDB interface to the GPU. However, neither of these completely satisfy people in need of familiar debugging techniques.

Previous Work
DCGN [4] used asynchronous memory copies to allow the GPU to request work from the CPU. However, the work to be requested was hard coded, and the request method was error prone.

1. Next Generation NVIDIA Tegra: http://www.nvidia.com/object/tegra_250.html
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. 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://graphics.cs.ucdavis.edu/publications/print pub?pub id=959

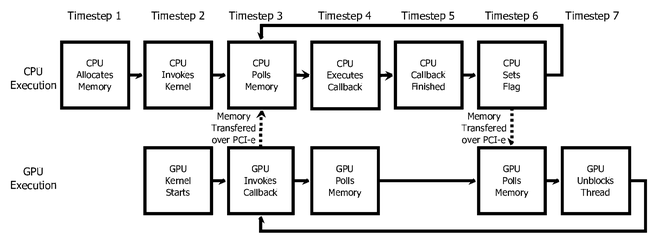
Callbacks in a Nutshell

A callback is a mean for a GPU to explicitly request work from a CPU. This is analogous to a CUDA-kernel invocation.

We implemented callbacks using pools of 0-copy memory and polling. When a GPU thread issues a callback, it copies function parameters into 0-copy memory and then sets a flag, also in 0-copy memory. The CPU, meanwhile, is polling the same area of 0-copy memory. When it notices a callback, it executes the appropriate CPU function and unsets the flag on the GPU, thus returning control to the GPU thread. A diagram of this behavior is shown below. Note that we assume when GPU vendors expose signals/interrupts to developers, we will be able to implement a sleep-based mechanism for handling callbacks.

Diagram of a Callback

The following diagram shows how the CPU and GPU behave when the GPU issues a callback. Note the overhead due to polling. We hope this will be one more reason for GPU vendors to expose interrupt/signal capabilities to developers.



The Look of Callbacks

Callbacks not only mitigate the problems mentioned in the motivation and background, but they're also very easy to use. Below is a small code fragment demonstrating callbacks. This particular example has the GPU read file data directly via callbacks. The master thread executes the callback, the CPU picks up the request, executes the user-defined read function, and then returns control to the GPU. Once the GPU detects the callback is complete, it begins its computations.

On CPU

```
callbackHandle_t readFileHandle;
createCallback(&readFileHandle, readFileFunction, CB_TYPE_INT,
               numberOfParameters, parameterTypesArray);
callbackData_t * gpuCallbackData = callbackGetGPUData();
kernel<<<gridSize, blockSize, 0, stream>>>(gpuCallbackData,
                                           readFileHandle,
                                           gpuMemory,
elementsToProcess);
callbackSynchronize(stream);
```

On GPU

```
__global__ kernel(callbackData_t * cbd, callbackHandle_t cbh,
                  int * gpuMem, int elementCount)
{
  if (threadIdx.x + blockIdx.x == 0)
  {
    callbackAsyncRequest_t req;
    req = callbackExecuteAsync<int>(cbd, cbh, gpuMem, elementCount);
  }
  while (!*ready)
  {
    if (threadIdx.x == 0 && blockIdx.x == 0)
    {
      if (callbackAsyncPoll(req) == CALLBACK_FINISHED)
      {
        *(volatile int * )ready = ready;
        __threadfence();
      }
    }
    doSomeProcessing();
  }
  doSomeMoreProcessingThatDependsOnGPUMem();
}
```