Integrating GPGPU computations with CPU coroutines in C++

Pavel A. Lebedev
Department of Computer Security,
National Research University Higher School of Economics,
34 Tallinskaya Ulitsa, Moscow, Russia
E-mail: cygnus@michiru.ru

Despite various efforts on standardization, both open and proprietary, there is currently no universally available programming interface to access compute capabilities of heterogeneous systems. The two main competitors — NVIDIA CUDA and OpenCL — are being pushed by NVIDIA and AMD respectively, and none is universally and uniformly supported.

C++ is a proven language for high-performance computing, but it lags in standardization of recently rediscovered solution to elegant asynchronous processing, namely, coroutines. Coroutines have proven to be the primary means of reverting tangled and piecewise asynchronous code back to serial and readable form. While there are several concurrent proposals that involve resumable functions, coroutines for C++ already have efficient library-based solutions.

Both CUDA and OpenCL are capable of invoking callbacks as notifications for events. This allows us to integrate calls to GPGPU computation kernels and memory copies with existing ways of waiting for network I/O and general callbacks using asio and coroutines based on boost.context libraries. With these we can achieve good code readability of heterogeneously asynchronous code.

Our experience shows that for most tasks overhead of coroutines against pure callback-based code following reactor pattern is insignificant, but even straightforward integration of foreign APIs involves additional overhead of system calls of unpredictable latency.

We've tested CUDA in different context modes that determine the way CPU waits for GPUs. OpenCL was also tested as implemented by NVIDIA and AMD for their hardware. We produced experimental results for the common scenario of exchanging data with an accelerator for the smallest possible amount of work to better expose latencies.

For NVIDIA CUDA the overhead is about $50\mu s$ for the cases where no kernel synchronization is used, which is twice the running time without using our approach. While huge for this test case, computations that are worth offloading to the GPU take orders of magnitude more time, so in real applications it won't contribute noticeably. If you're using blocking synchronization, which is the case for low power devices, the overhead is half as much in absolute numbers and only 30% in relative.

The OpenCL implementation from NVIDIA is strangely a lot slower when coroutines are utilized, showing extra latencies of about $500\mu s$. Even then, this should be acceptable for long-running kernels. OpenCL for AMD GPUs is comparable with CUDA blocking case, being slightly worse at average of $95\mu s$ with extra $20\mu s$ of overhead for coroutine use.

This shows that existing libraries and APIs already allow unifying of asynchronous programming in heterogeneous environments with acceptable overheads and good readability.

Keywords: C++, CUDA, OpenCL, coroutines.