Integrating GPGPU computations with CPU coroutines in C++

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

Abstract. We present results on integration of two major GPGPU APIs with reactor-based event processing model in C++ that utilizes coroutines. With current lack of universally usable GPGPU programming interface that gives optimal performance and debates about the style of implementing asynchronous computing in C++, we present a working implementation that allows a uniform and seamless approach to writing C++ code with continuations that allow processing on CPUs or CUDA/OpenCL accelerators. Performance results are provided that show, if corner cases are avoided, this approach has negligible performance cost on latency.

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

1. Introduction
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 [1] and OpenCL [2] — are being pushed by NVIDIA and AMD respectively. While CUDA is only supported on NVIDIA devices, OpenCL support is more widespread, but its implementations frequently have inconsistent behavior or performance. We will discover an example of such difference later in this paper. Automatic offloading tools are also in their infancy, which also contributes to the need to manually target these APIs to exploit full capabilities of the hardware.

C++ is a proven language for high-performance computing, but it lags in standardization of concurrent and parallel processing facilities. This includes recently rediscovered solution to elegant asynchronous processing in the form of coroutines. Coroutines are a generalization of subroutines ("functions" in C++) that may suspend execution and return control to the caller to be resumed later [3]. Different types of coroutines are useful to clearly express generators, state machines, cooperating pipeline stages and other types of control flow.

Traditional asynchronous code in C++ contains lots of relatively small functions that specify each other as a point at which the algorithm should continue once an invoked asynchronous call completes. This quickly complicates program structure in comparison with blocking synchronous code that doesn’t need to split algorithms with multiple asynchronous calls into pieces. Coroutines can be used as the primary means of reverting tangled and piecewise asynchronous code back to serial and readable form.

Currently there are several proposals that cover coroutines, which differ in the type of coroutines supported (stackless or stackful) and, implicit or explicit suspension syntax and
memory requirements and allocation strategies. A comparison of these approaches is given in [4]. While none of these approaches have been accepted into C++ language standard so far, many of them have working prototypes.

The first proposal to add stackful coroutine support to the language was a purely library-based solution based on Boost.Context library [5]. On the other hand, resumable functions [6] have been implemented by Microsoft as part of the recent release of Visual C++ compiler and are an example of leveraging compiler support to reduce memory requirements for saving stackless coroutine state. There are also hybrid proposals, for example [7]. Obviously, solutions based on language extensions have potential to be more efficient than pure library-based implementations.

CUDA and OpenCL both provide blocking and nonblocking API calls. Blocking calls are not usable in scalable code because they force to dedicate a separate operating system thread to every independent logical flow that works in parallel. This results in losses due to excessive context switch overhead and is known in networking applications as C10K problem [8]. Asynchronous APIs allow to avoid it, but the program’s readability suffers, as we’ve explained earlier.

In this work we’ll discuss how to seamlessly integrate computations on CPU and I/O with GPU operations that allow to specify any combinations of asynchronous operations in a unified form. While we’re waiting for coroutines to become an integral part of the C++ language, we may already obtain an estimate of abstraction penalty that coroutines impose in comparison with traditional code that uses GPU APIs.

2. Test implementation
In addition to using GPUs, programs perform computations on the CPU and input/output with storage, network and other devices. Computations directly consume CPU time, but waiting for I/O doesn’t. According to this scheme, operations on GPUs should be treated as another type of I/O as they consume no CPU time while waiting for results. This includes both data exchange with GPU and computations that run on it.

Operating system threads that are waiting for I/O requests to complete can be reused to perform other computations meanwhile. The reactor-based completion event handling [9] allows to exploit these opportunities, but decomposes algorithms to chains of continuations that are more difficult to read and reason about than traditional blocking code.

Coroutines allow to combine efficiency and scalability of this asynchronous approach while maintaining readability of the implementation. In this case coroutines suspend immediately after an asynchronous call and specify a small thunk that resumes them as their continuation. This construct integrates properly with existing dispatch and scheduling facility that is already present in asynchronous code.

While C++ already has future/promise as a way to perform asynchronous computations, a more rich set of facilities is required to specify policies for execution. Again, there are several competing proposals that tackle this problem. In this work we’ll be using a reference implementation of executors and asynchronous operations [10] that underpins current C++ Netowrking Library proposal, that is itself a derivative of asio library [11], a proven solution for writing scalable network I/O software in C++. The coroutines will be provided by the aforementioned Boost.Context library [12]. We choose these libraries because they are readily available, do not depend on particular compiler or operating system, and represent the least efficient library-based approach. This allows us to treat our performance as a guaranteed lower bound, which can only be improved on.

Reactor-based event processors allow easy integration with foreign event notification methods. Integration with asynchronous programming framework can allow GPU operations to have the same invocation syntax as other asynchronous operations. Even more important is that this will allow these operations to be composable with asynchronous operations of other types: different GPU API, network or disk I/O etc. This greatly simplifies design of complex and general
applications in heterogeneous environments. In fact, current design of C++ futures is not
generally useful exactly because it has poor composition support, but this is being worked on
[13]. It is important to note that both GPU APIs support multiple command queues, therefore
composition may be required even in scenarios that work with a single GPU.

Fortunately, both CUDA and OpenCL are capable of invoking callbacks as notifications for
events. The relevant functions are cuStreamAddCallback (CUDA) and clSetEventCallback
(OpenCL). Both memory copy and code execution operations can be integrated in this way.
Thanks to the use of coroutines and code execution operations can be integrated in this way.
An example of how the presented solution influences code design is given below. Consider a
scenario of receiving a fixed width data block from network into host memory that is mapped
into GPU address space, processing it in place and sending the result back. The following code
assumes different GPU APIs have already been wrapped for use with asio reactor and employs
generic programming to take advantage of this fact.

```cpp
template<typename Stream,typename Kernel>
void sync_code(Stream& stream,array_view view,Kernel& kernel)
{
    while(read(stream,view)==view.size()){
        kernel.invoke(view.data(),view.size());
        write(stream,view);
    }
}

template<typename Stream,typename Kernel,typename CompletionToken>
void async_code(Stream& stream,array_view view,Kernel& kernel,CompletionToken&& done)
{
    struct io_op
    {
        struct data
        {
            Stream& stream;
            array_view view;
            Kernel& kernel;
            CompletionToken&& done;
        }

        shared_ptr<data> d;

        io_op(Stream& stream,array_view view,Kernel& kernel,CompletionToken&& done)
            : d(make_shared<data>(stream,view,kernel,std::forward<CompletionToken>(done)))
        {
            write_done(0,{});
        }

        void write_done(size_t bytes_written,error_code ec)
        {
            async_read(d->stream,d->view,bind(&io_op::read_done,*this));
        }

        void read_done(size_t bytes_read,error_code ec)
        {
            if(bytes_read==d->view.size())
```


```cpp
kernel.async_invoke(d->view.data(),d->view.size(),
    bind(&io_op::compute_done,*this));
else
    std::forward<CompletionToken>(d->done)();
}
void compute_done(error_code ec)
{
    async_write(d->stream,d->view,bind(&io_op::write_done,*this));
}
}
op(stream,view,kernel,std::forward<CompletionToken>(done));
}
template<typename Stream,typename Kernel>
void async_coro_code(Stream& stream,array_view view,Kernel& kernel)
{
    while(async_read(stream,view,yield)==view.size()){
        kernel.async_invoke(view.data(),view.size(),yield);
        async_write(stream,view,yield);
    }
}
```

The `sync_code` function represents the most simple blocking solution: it is concise, but consumes an OS thread for its execution. It returns when there is no more data to process.

The `async_code` function shows the complexity of traditional asynchronous approach that requires splitting the logical loop into multiple intermediate operation completion handlers. These examples do not contain any error checking for clarity of presentation, but in real code its presence would complicate code even further. This function also contains shared ownership memory allocation to satisfy completion handler technical requirements. As is customary in asynchronous code, this function returns after initiating its operation and calls the provided continuation when its job is done.

The last function `async_coro_code` is the result of approach presented in this work. It mixes GPU operations with I/O on an arbitrary stream abstraction (network, disk, etc) and is as simple as the blocking code while retaining benefits of being asynchronous. `yield` is a global function object that suspends the coroutine when it is passed as continuation parameter using boost.coroutine. It also changes the invoked operation’s semantics back to synchronous case, where it returns the result of the operation directly. The implementation of this object is based on the one provided by asio library, but our version adds additional features, such as support for coroutine-specific local storage, that extends the use of thread-specific local storage to coroutines.

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 with unknown latency. Both APIs do not document the exact set of threads that user callbacks may be invoked on. Since we follow the executors model with strict guarantees for this kind of behavior, and given the restriction of operations that may be invoked in callbacks, we have to repost the notification to the proper thread from the callback. This involves at least one context switch and is another source of latencies.

3. Performance comparison
We have tested our implementation on two machines with different operating systems, both equipped with NVIDIA and AMD GPUs:

(i) Core i7-920 CPU, NVIDIA GTX 580 GPU, AMD Radeon HD 7570 GPU, Linux OS.
(ii) Core 2 Quad Q6600 CPU, NVIDIA GTX 460 CPU, AMD Radeon R7 265 GPU, Windows OS.

Our goal is to measure relative overhead of using coroutines with GPUs. In our tests we measure execution time for the common scenario of sending data to the GPU, performing computations there and sending results back to CPU. To make coroutine overhead more visible, we use the least possible amount of data (1 machine word) and trivial computation (single GPU thread with a single increment instruction) to process it. For this test GPU performance difference between all 4 GPUs is irrelevant, instead we compare performance of different APIs on different operating systems.

We’ve tested CUDA in different context modes that determine the way CPU waits for GPUs (see flags of `cuCtxCreate`). Spin mode corresponds to a busy loop waiting for completion. Yield mode also uses a wait loop, but it gives up its thread time slice on every iteration. In all our scenarios and different system loads we did not find any measurable difference between spin and yield options, so they are presented as a single entry. Block mode uses unspecified thread synchronization primitive to wait for GPU completion without consuming any CPU time.

First we consider results for Linux OS presented in table 1. For NVIDIA CUDA the overhead is about 50 µs for the cases where no kernel synchronization is used. 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. For example, this consumes 0.3% of time allotted to a single frame of a real-time 60 frames per second visualization. When using blocking synchronization, the overhead is unmeasurable. Blocking synchronization is preferable for long-running computations, on low power devices and when trying to utilize all computation power of a system for high-performance computations.

The OpenCL implementation from NVIDIA is strangely a lot slower when coroutines are utilized. Even then, this should be acceptable for long-running kernels. We believe this to be a driver or runtime quality of implementation issue. OpenCL for AMD GPUs is slightly slower for the non-coroutine baseline, but the overhead is comparable to CUDA blocking case when coroutines are used. Unlike CUDA, OpenCL has no standard or other way to specify whether to use synchronization primitives, spin loops or other ways to detect GPU events. Since performance of AMD OpenCL implementation is more close to the blocking case, we also tested a manual spin loop implementation ("poll" entry in the table). Surprisingly, this allows for better latency, which seems to confirm that AMD implementation does use blocking. We consider it another case in support of OpenCL not fulfilling its role of the universal API.

Unfortunately, we’ve discovered incompatibility between implementation of CPU context switching in Boost.Context library on Windows and NVIDIA runtime. This issue is known and is a result of lack of documentation of Windows OS internals. In anticipation of such problems Boost.Context on Windows can use existing Fiber API to achieve compatibility at the price of
Table 2. Performance results for Windows OS (preliminary).

| API     | Variant  | Synchronous code | Coroutines |
|---------|----------|------------------|------------|
| CUDA    | spin/yield block | 120 ± 20 230 ± 20 | 250 ± 25 260 ± 25 |
| OpenCL  | NVIDIA   | 160 ± 20         | *          |
|         | AMD      | 200 ± 30         | 205 ± 30   |
|         | AMD(poll)| 155 ± 15         | N/A        |

performance. While it allowed us to run tests, it doesn’t work reliably, therefore we consider results for the Windows OS to be preliminary.

Most of the results follow the same pattern as on Linux. The AMD OpenCL implementation on Windows is slower than NVIDIA, but remains competitive. The main problem is the test of coroutines with NVIDIA OpenCL implementation, that is unacceptably slow (about 0.036 seconds, not included in the table). In some runs this test hanged and failed to complete at all. We believe this to be another incompatibility with the use of fibers. While we’ve run into several problems on Windows, we believe these can be overcome with future updates to drivers and libraries. When the C++ language evolves to include coroutine support in the standard with compiler-assisted optimizations, our solution can be adapted to use them, which will allow us to eschew complete CPU context switches and corresponding problems.

This shows that existing libraries and APIs already allow unifying of asynchronous programming in heterogeneous environments with acceptable overheads and good readability. Our results serve as a worst-case estimate that is already acceptable barring corner cases and will be improved in the future.

References
[1] NVIDIA CUDA
   [http://www.nvidia.com/object/cuda_home_new.html](http://www.nvidia.com/object/cuda_home_new.html)
[2] Khronos OpenCL
   [https://www.khronos.org/opencl/](https://www.khronos.org/opencl/)
[3] Knuth D 1997 The Art of Computer Programming 3rd ed, vol 1 (Amsterdam: Addison-Wesley) pp. 193-200
[4] Riegel T 2015 P0073R0: On unifying the coroutines and resumable functions proposals
   [http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/p0073r0.pdf](http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/p0073r0.pdf)
[5] Kowalke O 2015 N4397: A low-level API for stackful coroutines
   [http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4397.pdf](http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4397.pdf)
[6] Nishanov G and Vandevoorde D 2015 N4499: Draft wording for Coroutines (Revision 2)
   [http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4499.pdf](http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4499.pdf)
[7] Kohlhoff C 2015 P0114R0: Resumable expressions (revision 1)
   [http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/p0114r0.pdf](http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/p0114r0.pdf)
[8] Kegel D The C10K Problem
   [http://www.kegel.com/c10k.html](http://www.kegel.com/c10k.html)
[9] Schmidt D et al. 2000 Pattern-Oriented Software Architecture vol 2 (New York: Wiley)
[10] Kohloff C 2015 P0113R0: Executors and Asynchronous Operations, Revision 2
    [http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/p0113r0.html](http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/p0113r0.html)
[11] asio C++ library
    [http://think-async.com](http://think-async.com)
[12] Boost.Context
    [http://www.boost.org/doc/libs/develop/libs/context/doc/html/index.html](http://www.boost.org/doc/libs/develop/libs/context/doc/html/index.html)
[13] N4399: Working Draft, Technical Specification for C++ Extensions for Concurrency 2015 ed Laksberg A
    [http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4399.html](http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4399.html)