LAWRENCE
LIVERMORE
NATIONAL
LABORATORY

LLNL-TR-679873

Running Parallel Discrete Event Simulators on Sierra

P. D. Barnes, D. R. Jefferson

December 3, 2015

Disclaimer

This work performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344.

Running Parallel Discrete Event Simulators on Sierra

Peter Barnes, David Jefferson

July 10, 2015

Introduction

In this proposal we consider porting the ROSS/Charm++ simulator and the discrete event models that run under its control so that they run on the Sierra architecture and make efficient use of the Volta GPUs.

ROSS/Charm++ is itself a two-level piece of middleware for the support of dynamically-load-balanced optimistic parallel discrete event simulations. Charm++ is a runtime system that implements a model of parallel computation in which the application is (over)decomposed into small computational units called *chares* that are units of MPMD parallelism and also units of load migration. Charm++ also implements a one-sided asynchronous inter-chare messaging layer. Charm++ can be compared to MPI in that both are runtime systems that support models of parallelism and communication, but the chares of Charm++ are smaller than MPI tasks, and unlike tasks, are migratable between nodes of the underlying HPC platform.

ROSS is a platform for parallel discrete event simulations that runs over Charm++ and uses its services and abstractions in much the same way that many applications use MPI.

A discrete event model running over the ROSS/Charm++ simulator is a third layer of software. The relationship between the model code and the simulator is much like an application running over an operating system. ROSS/Charm++ generally provides services and abstractions to the model code, including: units of parallelism (logical processes, aka LPs), simulation time, timestamped event message communication, event scheduling, synchronization, error handling, load balancing, memory management, I/O, instrumentation, checkpoint/restart, normal and abnormal termination, etc.

ROSS, of course, is an *optimistic* simulator, which means that it must support *rollback* of events in the model code. Part of this work is done at compile time with a ROSE-based reverse code generator called Backstroke, and any port of model code to run Sierra will require support from Backstroke as well.

The complex of model code linked to ROSS/Charm++ is currently designed for generic multi-core, multiprocessor clusters and has been demonstrated to perform well up to the scale of Sequoia and beyond. However, the Sierra architecture with its GPUs is a radical departure, and refactoring the three-layer

Charm++/ROSS/model complex to take advantage of the new architecture is going to be very challenging.

There are several possible approaches to running the ROSS Complex on Sierra, which we will describe in the next section. How much of this will be possible will at least partly depend on features and details of the Volta architecture that we do not yet fully understand.

Approaches to running PDES on GPUs

We now describe five different approaches to "porting" ROSS/Charm++ to Sierra, presented in order of increasing ambition and complexity, and increasing requirements for the GPUs, their drivers, and system support for them.

1) Portions of model event code run on GPUs

We can run the Charm++ and ROSS layers exclusively on the CPU cores, and also by default run the model event code on the CPU cores as well, but allow an event to *optionally* launch kernels to run on the GPUs. In this case only those models that have event bodies with the appropriate kind of internal parallelism will run on the GPUs at all. It is likely that some models will make heavy use of this capability, some will make partial use in certain LPs, and others will make no use of the GPUs at all.

This is the simplest and most direct way for the ROSS complex to use the GPUs. It involves the least changes to the ROSS and Charm++ layers, but it still requires some.

- The ROSS layer must provide an API for the model code running on the CPU to use to launch GPU kernels.
- It is desirable for ROSS to be able to kill a kernel that is running on the GPU i.e. to preempt a running event,
- For debugging and validation we should provide the capability to run either a CPU-only model implementation or the GPU-enabled model implementation with no change to the code.
- We may need to make it possible for GPU code to make callbacks to ROSS/Charm++ running on the CPU in order to send event messages, read simulation time, or invoke other services. To the extent this isn't possible, we will have to provide standard workarounds to streamline interaction between the GPU kernel and the rest of the simulator.
- Runtime errors in a GPU kernel should trap (or signal, or throw) to code running in the simulator on the CPU. In optimistic simulation a runtime error in model code such as a zero divide or seg fault must not simply cause abnormal termination of the whole job. The error must be caught and handled by the simulator, because a rollback may occur to effectively undo the effects of the error.

- The Backstroke reverse code generator will have to be able to generate forward/reverse code for all parts of the event code that run on the GPUs, and it must integrate properly with forward/reverse code running in the same event but executing on the CPU.
- In particular, the fact that the CPU-based code and GPU-based code run asynchronously requires that the two instruction streams not modify the same state variables, including via aliasing effects. Alternatively, they will have to synchronize accesses to state variables shared between the CPU and GPU, or between two threads on the GPU.

2) Run full event bodies on the GPUs

Instead of having the event methods start on the CPU and initiate kernels to run on the GPU, it may be useful for the simulator itself to initiate an entire event body to run on the GPU.

This would make sense if, for example, all of the events for a particular LP were suitable for GPU execution and the *state* of the LP could reasonably be kept resident in fast GPU-local memory. It would also make sense, for example, if the large majority of all events in the simulation used the same event code, because then the code could be resident in GPU cache.

All of the issues faced in running some event code on the GPU, as described in the previous section, apply here as well. But running the full body of events on the GPU would in addition *definitely* require the ability to make callbacks to the simulator running on the CPU in order to send event messages or for other services..

3) Run some parts of the ROSS Simulator on the GPUs

So far we have considered only running model code on the GPU. But, depending on the architectural features of the GPU we could consider also running some of the asynchronous background activities of the simulator itself on the GPU, thereby offloading the CPU. Among the potential asynchronous simulator behaviors that might be made to run on the GPU are:

- priority queue management
- GVT calculation
- commitment actions, e.g. fossil collection (a form of storage management)
- load balancing policy calculations

4) Run the entire ROSS (or new `xpdes`) simulator on the GPU

Depending on how flexible the Volta GPU architecture turns out to be, it may be possible to run the entire ROSS simulator and Time Warp algorithm on the GPUs in addition to the model code. This would require the Volta GPU to be fully capable of acting logically like a multicore processor in almost all respects (except performance), with each warp in effect acting like a core.

In order for this to be possible the Volta GPU would have to support some rather unorthodox (for a GPU) programming constructs.

- It would need to allow the GPU to run some kernels (the simulator) that never terminate.
- Simulator code in one warp of the GPU would have to be able to dynamically launch additional kernels for GPU execution.
- Model kernels running on GPU warps would have to be able to call simulator functions which would run in a different kernel (i.e. the simulator)
- It would greatly benefit if the code on one warp could *interrupt* that on another (though polling could, with some loss of functionality and performance, be substituted).

5) Run Charm++ on the GPU

If it's possible to run the ROSS simulator layer on the GPU, it may be possible to run the Charm++ layer on the GPU as well. In addition to the capabilities required to run ROSS on the GPU, we would need:

- The capability of invoking operating system calls, for purposes such as message-sending and -receiving and reading the real time clock, directly from GPU code.

Work Plan

There is some limited prior work on running PDES on GPUs, but all of it is fairly old, predating even CUDA in most cases. Essentially all of the prior work considered only conservative PDES, not OPDES. We plan to revisit this prior work, and extend it to OPDES, in the light of anticipated Volta capabilities. We expect to conclude that some approaches are more or less straightforward development tasks, appropriate for iCOE funding, and some require significant new research, which will be more appropriate for an LDRD project.

The main objective for this work plan is to do enough development to distinguish these two cases. For iCOE development, we will produce a roadmap for further development work. Topics needing further research will become the meat of a future LDRD proposal.

Above we described approach #4 as "porting ROSS/Charm++." For a variety of reasons we have simultaneously been considering developing a new OPDES implementation to address a fairly long list of shortcomings in the ROSS code base. Since completing the earlier white paper we have learned that development of this new simulator, dubbed xpdes, will be funded starting in FY16. (In fact, we have already begun using existing FY15 dollars.) We believe it is definitely worthwhile doing the background and development work in this work plan even before we have

frozen the architecture of xpdes. What we develop in this first year will be illustrative working examples of how we can accomplish important OPDES operations on GPUs; the important insights will be independent of the specifics of the surrounding OPDES implementation. Further, these examples will inform the developing architecture of xpdes. (In fact xpdes will leverage much of the infrastructure of the ns-3 simulator, and several of the examples listed below are drawn from ns-3, so the implementation details will be nearly identical.)

Deliverables, FY16:

As an initial outcome of this first year effort:

- We will have the core xpdes team (3 people), the Backstroke developer, and our two lead application developers complete a short course (~12 days total, tutorial and practicum) on GPU programming, preparing them to contribute directly to developing OPDES to run efficiently on Sierra.

Specific deliverables for FY16 to accomplish Approach 1 are:

- We will develop the glue that allows event bodies to launch CUDA code (or OpenMP code) to run on a (current generation) GPU.
    - As a first step, we will port an existing GPU-implementation of all-pairs shortest path to the GOD routing protocol in ns-3. This will force us to address the glue code and interoperability issues for mixed CPU-GPU event implementations.
- We will modify Backstroke to support event code running on the GPU and/or CPU.
    - Specifically we will support the case where a new data structure is computed on the GPU, and then in the CPU moved to a model state variable.
- We will study the options for more ambitious use of the GPUs as outlined in Approaches 2–5. We will develop both the roadmap for iCOE-funded development, and the outline of an LDRD research proposal, for the appropriate topics.

Staffing

The core PDES team (those people with a hand in core simulator and/or application development) should be involved, if only for the training phase. The people are:

- Eddy Banks
- Peter Barnes
- David Jefferson
- Sergei Nikolaev
- Tomas Oppelstrup
- Markus Schordan
- Steve Smith
- Jae-Seung Yeom
- Postdoc

The core xpdes team, consisting of Peter Barnes, David Jefferson and Steve Smith, will spend 30% time for the year (0.9 FTE total). The Backstroke developer, Markus Schordan, and the application developers, Sergei Nikolaev, Tomas Oppelstrup and

Jae-Seung Yeom, will spend 30% time for three months, in several sprints, to test and critique the development code (0.3 FTE total). The total estimated effort is 1.2 FTE.

We also anticipate needing vendor support at ~30% FTE for most of FY16.

Hardware Access

In FY16 we anticipate using any available iCOE hardware at a modest level for exploratory development and benchmarking.