State Explosion: An Obvious Limitation To Strong Scaling

Gregory Diamos
School of Electrical and Computer Engineering
Georgia Institute of Technology
gregory.diamos@gatech.edu

October 17, 2009

1 Introduction

The scaling of future computing systems is currently limited to a single available dimension : parallelism. Applications that claim to be scalable today must be both efficient on modern architectures with limited degrees of parallelism and future architectures with extreme degrees of parallelism. This implies that the specification of an application must include a large number of parallel operations, threads, as well as an efficient mechanism of fusing threads together for architectures with a relatively small degree of parallelism.

This dynamic range of parallelism in target architectures poses several fundamental problems that are currently not addressed by modern programming models. Most significantly, many programming models will experience a state explosion when highly parallel applications designed to scale to future systems are mapped onto state of the art serial architectures.

2 Discussion

Consider a program that is very highly parallel, say that it can be partitioned into N independent threads each requiring a maximum of M units of asociated memory state. For an N-way parallel system with independent processing and memory resources for each thread, each processor will at most require M units of memory state. However, if this program is run on an architectiture with fewer processors, say K processors for example, an average of N/K threads will be mapped onto each processor. If threads must be all alive at the same time, i.e. if they are allowed to share data via producer/consumer relationships, then each processor must be able to store $N/K * M$ units of memory. In the limit where $K- > 1$ and $N- >$ inf, the amount of memory required per processor explodes towards infinity.

This memory state explosion prevents the execution of applications with a large number of threads on smaller scale systems. Conversely, applications that can execute on smaller systems are artificially limited from scaling to future parallel systems because they cannot explicitly express enough parallelism.

Taking a more concrete example. The NVIDIA CUDA parallel programming model allows threads to be grouped together in units of 512, with the maximum number of thread groups being 65536x65536 – the total number of threads in a CUDA program is 2^{41}. This is a significantly larger amount of parallelism than is available in hardware. In fact, even if each thread required only a single 64-bit program counter of state, it would require 16TB of memory to store only the program counters for each thread, if all threads were alive at the same time. We refer to this as state explosion.

Several CUDA programs, though, actually do use all 2^{41} threads. CUDA gets around this state requirement by disallowing threads that exist in distinct groups from exchanging data. This means that only a subset of groups need to be allocated state and alive at any time; in other words, it means that a program can contain an unlimited number of thread blocks without experiencing a state explosion.

All other parallel programming models that I am aware of experience this problem, including pthreads, OpenMP, and MPI.

3 Scalable Programs with Unlimited Threads

The primary cause of state explosion in parallel programming models is a contention between the ability to launch a large number of threads, and have threads communicate via global synchronization. In this section I discuss several alternatives to global synchronization.

The CUDA programming model makes a significant contribution towards the design of a scalable parallel programming model. It avoids a state explosion by disallowing thread blocks to synchronize. In order to synchronize, it is necessary to launch a kernel multiple times, where there is an implicit barrier between kernel launches. This approach works because all threads that were alive in the first kernel are implicitly killed at the end of the kernel and their local state is discarded.

However, it also means that there is no low latency mechanism for global synchronization. This lack of global synchronization has been the source of much debate to the extent that some developers devote significant efforts to artificially enabling global synchronization via mutexes using atomic operations. These implementations are only correct if the number of thread blocks launched can all be executed concurrently on a particular GPU. Anything more and they again run into the state explosion problem, where the resources required to keep all threads alive is greater than those available in hardware.

4 Destructive Global Synchronization

As a solution to this state explosion problem, I introduce the concept of a destructive global barrier and globally persistent shared state. A destructive global barrier is a barrier across all threads in a program such that no thread local state is persistent across the barrier. This means that threads cannot maintain registers, program counters, local memory, or thread-group local memory across a destructive global barrier. They can only maintain a single program counter, from which all threads resume execution upon completion of the barrier. They can also maintain a fixed-size persistent shared memory that is shared across all threads in the program and made consistent across global barriers.

I believe that these two features, coupled with the thread hierarchy of CUDA including sub-groups of threads which can communicate, will allow programs to be written with an arbitrary number of threads in a way that avoids state explosion on modest systems. Thread groups can be either executed in parallel or fused together into sequential loops on serial architectures. Global synchronization can be supported in hardware with low latency via destructive global barriers where the only state that can be shared across these barriers is fixed size persistent memory. Persistent memory can either be implemented in low latency on-chip memory, or off-chip DRAM. Eventually, as processors contain more cores and the cost of core-to-core communication increases, I believe that the benefits from shared on-chip memory will diminish.

5 Suggestions

I believe that future scalable parallel programming models will have the following characteristics:

- **Stateless global synchronization only**: state that is replicated for every thread must not be persistent across global barrier boundaries.
- Local domains of synchronization are allowed to have persistent state.
- Serial sections performed by one thread and parallel sections performed by all threads should be explicitly identified.
- Explicitly managed state that is persistent across global barriers.

6 Looking Forward

Looking forward, a two-level hierarchy that disginguishes between global and local synchronization is lacking as the difference in cost of synchronization at each level is likely to continue to increase. A more robust programming model might generalize the problem into an arbitrarily leveled hierarchy of domains of synchronization. In this case, all levels would need to define persistent shared state and destructive barriers to remain scalable.

7 Thanks

Thanks to Tim Murray from NVIDIA for helping me identify specifically state explosion as the limiting factor in fusing threads together on modestly parallel architectures and understanding the design of the CUDA programming model.