R Benjamin Clay HW 5 11/11/09
rbclay@ncsu.edu CSC 548

Overview

Since the last report was written, we have made a few important architectural changes in our proposed solution, as well as hit upon some unforeseen problems with our implementation. We were originally overambitious in establishing a timeline, and that has been adjusted as well based on our experiences. Parallelization of NAS PB IntegerSort is definitely feasible using CUDA, but these issues must be overcome before significant speedups are realized. We have revised our timeline to support these new developments.

Updates to Proposed Solution

Previously, the core of our proposed speedup was the running of multiple iterations of the rank() function simultaneously. While we still intend to investigate this, from our initial work splitting and porting the rank() function to CUDA, we are seriously concerned about data dependence between successive iterations of the rank() function, as well as the logistics of the MPI-based communication required.

Initially, we misinterpreted the functionality provided by the cudaMPI library. cudaMPI allows the host to perform MPI calls based on device memory locations, essentially handling DMA operations for the programmer in a transparent fashion. We had initially believed cudaMPI allowed the devices themselves to place MPI calls, which was erroneous. The outcome of this is that the original host-run rank() function has been split into 3 separate CUDA kernels, with MPI calls between them. Between rank1 and rank2 cudaMPI is used, and between rank2 and rank3 normal MPI calls are used with manual DMA. This, coupled with the logistics of managing different datasets for each iteration, significantly complicates the original proposal to perform rank() iterations in parallel (cross-iteration). While we still intend to investigate this concept, it is no longer the core focus of our speedup efforts.

Instead, we intend to focus on parallelizing the individual rankX() kernels themselves, to complete the execution of the rank() iterations as fast as possible. rank1() is the best candidate, with multiple loops iterating over the largest arrays used in IS, but rank2() and rank3() could benefit from parallelization as well. This is a more reasonable target, given the data dependence issue with cross-iteration changes, and even if cross-iteration speedup is feasible, speeding the rankX() executions would be tangential anyway. We feel this is a better target for initial speedup, with cross-iteration investigations performed later as time allows.

Implementation Issues

In attempting to parallelize the rankX() functions, we have hit on two significant roadblocks. The first and most troubling is data corruption in global memory, due to contention across blocks. The other is kernel crashing due to use of shared memory within blocks, which is probably due to implementation specifics.

When we encountered this data corruption issue, we were attempting to parallelize the larger loops inside rank1(). These loops iterate 2 million times for class A, and many more times for the larger classes, so they are the central part of what we want to speed up. The large size of these arrays mean that we must keep them in global memory, which is slower than shared memory. The core of these loops is sorting the array of random keys into buckets, which involves using the random keys as indices into other temporary arrays. The result of this is that for a given CUDA thread, we can partition the input space (key arrays) such that each thread works independently, but we don't know what values in the intermediate array will be altered, meaning cross-thread dependence could exist. This results in a contention problem, because two random keys being handled by two threads may result in writes to the same intermediate array position. This is complicated by the fact that most of the writes are in fact increments, which mean both a read and a write for each operation.

When we first parallelized these loops, we were able to run them for very small numbers of threads (2-8 total), but above this number we started seeing data corruption, which IntegerSort detects during its partial and full verification phases. To fix this, we decided to use the atomic operations that CUDA provides, which guarantee atomic transactions on both shared and global memory. Using these atomic operations, we are able to scale our loops to 1 block with 512 threads, which is a significant increase over 8 threads. However, in the best cases, we are unable to move to more than 1 block, presumably because of reliance on the __syncthreads() function. __syncthreads() acts as a barrier to all threads within a block, but is local to that block, so when multiple blocks are operating in parallel, one could move on before global memory changes have taken effect. The __threadfence() function ensures memory visibility across blocks, but fails to help our issue.

To sync across blocks, we will have to further split the rank1 kernel into several. Parallelizing rank1() from 1 to 512 CUDA threads resulted in a speedup of 40 seconds -> 18 seconds for the overall IS class A execution, but given that it executes in 4 seconds on the host, this is still not acceptable.

The second issue we encountered was during an attempt to move some of the smaller intermediate arrays used in rank1() into shared memory, which should speed up all operations, including those that are atomic, by reducing memory latency significantly. Shared memory introduces a new problem of cross-block synchronizing, but since we're currently limited to 512 threads we chose to ignore that issue for now. Shared memory is limited to 16k per block, so the arrays targeted needed to be aggregately less than this value. Most of them are around 1024 in length, and composed of integers, so 1024*4bytes = 4096 bytes should be significantly less than the maximum. We should also be able to 2 or even 3 of these arrays per block, but we chose to just use a single shared array at first.

The problem that resulted from use of shared arrays was that subsequent kernel instantiations resulted in system crashes. We assume this is due to the shared array overwriting some valuable block context information, or other memory that should not be touched. This occurred both when we used an explicitly sized shared array, as well as one without a size, as described by the CUDA programming guide (extern __shared__ myarray[];). We found this very puzzling, because we got no errors during the kernel execution that uses the shared memory, but instead the following kernel execution. We have never used shared arrays of this size with CUDA before, but as demonstrated above we should be well below the system's maximum. We will continue to investigate this issue, as shared memory offers good speedup, but since CUDA's atomic operations support both global and shared memory, this conversion is desired but not required.

Timeline – Updated

Subject to change dependent on final project due date.

Day 4
- Split rank() function into rank1(), rank2(), and rank3() performing MPI calls in main()

Day 6
- Switch from using global memory to memory pointers for rankX() functions

Day 10
- Port rank() functions to CUDA kernel code
- Replace MPI calls with corresponding cudaMPI calls

Day 15
- Evaluate rankX() parallelization potential, begin experimental changes
- Write intermediate report

Day 25
- Complete rank1() parallelization to large/arbitrary number of threads
- Evaluate feasibility of using shared memory

Day 28
- Complete rank2() parallelization to large/arbitrary number of threads

Day 30
- Complete rank3() parallelization to large/arbitrary number of threads

Day 35
- Examine feasibility of performing rank() iterations in parallel
- Implement rank() iterations in parallel if possible

Day 40
- Compare performance to original results
- Document all coding efforts