NAS Parallel Benchmark: Integer Sort Using NVIDIA CUDA Kernels

R. Benjamin Clay
rbclay@ncsu.edu

Table of Contents

Overview **3**
Project 3
Evaluation 3
Outcome 3

Implementation **4**
Algorithm Analysis 4
As Implemented 5
Parallel Loops 5
Serial Loops 5
Communication and DMA 6

Results and Analysis **7**
Methods 7
Timers 7
Systems 7
Interference 8
Results 9
Analysis 11
Communication 11
DMA 11
Computation 11

Issues **12**
Parallelization 12
Algorithm-Related Limitations 12
Rank3 Design Problems 12
Contention and Atomic Operations 13
Memory Architecture 13
Global vs Shared Memory 13
MPI and DMA Operations 14

Future Work **15**
References **15**

Overview

Project

In this project, we have attempted to parallelize the NAS Parallel Benchmark: Integer Sort (IS)[1] using NVIDIA CUDA [2] kernels. We chose to focus on the rank() function within IS, which was previously shown to be the source of slowdowns unrelated to specific system implementation, and especially as test class and thus input size grew. To take advantage of the unique parallelization capabilities of the CUDA platform, we utilized three major programming constructs: parallelization of loops, use of shared memory, and selective DMA operations to ensure data consistency between hosts and CUDA devices.

Evaluation

We evaluated our implementation (CUDA) against the original IS code on 4 and 8 processors within the os40-os56 machines provided for the course. We timed total execution, communication vs computation, as well as the specific components of rank() execution in the CUDA implementation. The same machines were used for all runs, which were performed as close together as possible, to ensure environmental unknowns affected all tests equally.

Outcome

Ultimately, we were unable to significantly speed the operation of the IS benchmark, primarily due to its dependence on very expensive collective communication routines which resulted in computation representing only a very small portion of total execution time. However, even when only considering computation, we achieved very minor speedups and slowdowns, with all results obtained well within the margin of error.

In addition to communication dependence, the memory architecture of CUDA does not lend itself to use of such large arrays, resulting in heavy reliance on global memory, which is the slowest of CUDA memory types. This is exacerbated by the fact that the random inputs to the IS function result in unpredictable and potentially conflicting array accesses across multiple threads executing simultaneously, which necessitates the use of CUDA's Atomic operations to preserve data integrity. These Atomic operations incur additional slowdown, especially when used on global memory.

Finally, the unavailability of all-to-all MPI functions in the cudaMPI [3] library require these operations be performed on-host, which means additional DMA penalties in transferring the required arrays to and from the host before and after the calls. This specific problem has led us to choose to perform the rank() operations after the all-to-all communication on the host, and forgo optimization of these loops.

These issues and how they affected our results are discussed below in the Issues section.

Implementation

We focused exclusively on the rank() function for our CUDA optimizations. We divide the rank function into 3 sections, demarcated by MPI calls, for convenience:

- rank1 – before the MPI_Allreduce function
- rank2 – between the MPI_Allreduce function and the MPI_Alltoall function
- rank3 – after the MPI_Alltoallv function

Algorithm Analysis

Table 1 lists all the major loops in rank() and their iteration counts for relevant numbers of processors (for processor count discussion, see the Results and Analysis - Methods section).

- Green loops are ideal candidates for parallelization, based on iteration count and algorithm construction.
- Orange loops cannot be effectively parallelized due to algorithm construction.
- Grey loops can be parallelized but are too small to make any difference.

				NPROCS = 4		NPROCS = 8	
Location	**Nickname**	**Parallelization Possible**	**Parallelization Implemented**	**Class B Iterations**	**Class C Iterations**	**Class B Iterations**	**Class C Iterations**
rank1	init	Yes	Yes	1024	1024	1024	1024
rank1	countkeys	Yes	Yes	8388608	33554432	4194304	16777216
rank1	bucketsizes	No	-	1024	1024	1024	1024
rank1	sort	Yes	Yes	8388608	33554432	4194304	16777216
rank2	-	No	-	1024	1024	1024	1024
rank3	clearwork	Yes	No	837631	3350527	694271	2777087
rank3	lesserkeys	Yes	No	408*rank	408*rank	338*rank	338*rank
rank3	localkeys	Yes	No	408	408	338	338
rank3	indivkeypop	Yes	No	8399306	33581466	4238226	16945007
rank3	keyrank	No	-	837631	3350527	694271	2777087

Table 1: Major loops in rank()

As Implemented

Seen in Table 1 above, we implemented the rank1 and rank2 major loops, including both the parallelizable and serial loops. We were unable to implement the rank3 loops due to problems discussed in the Issues section.

Parallel Loops

For the trivial parallelizable loop, rank1 - init, we used 2 blocks of 512 threads each, and used each thread to initialize a single element in several global arrays. Since each thread initializes its own elements without contention, this implementation was simple.

For the two parallelizable loops, rank1 - countkeys and rank1 - sort, we faced data contention issues. Unfortunately, both use random values to index output arrays, which lends itself very poorly to parallelization. Because of the random indices, we are unable to partition the output space of each thread, and could only partition the input space. Additionally, the operations performed on these output arrays are not simple assignments but instead increments, which require the original value to be read before setting the new value.

The result of these two concepts, inability to partition output space and heavy use of increments, is that contention was a serious threat, which we indeed experienced in the form of data corruption and even kernel panics as we attempted to scale up the number of threads used. Ultimately, to leverage the power of parallelization with a large number of threads, we were forced to turn to CUDA's Atomic operations, which insure data integrity with absolution at the cost of performance. Using Atomic operations, we were able to correctly perform increments and updates to global memory, and scaled both kernels up to 2048 blocks of 512 threads.

Serial Loops

We implemented two serial loops as CUDA kernels to avoid DMA overhead.

rank1 - bucketsizes is considered serial because it calculates array elements in a pipeline fashion - each is dependent on the one before it. This unfortunately cannot be parallelized, at least as far as computation. We did however use a shared memory approach with this kernel, by calculating these values in a shared memory array and then using additional threads to copy it into global memory. It is run with 512 threads so all threads can access the same shared array. This is highly inefficient, as all 511 threads wait while 1 performs computation, but given that this kernel was unavoidable we included it.

rank2 is considered serial, not because it has strict position dependence as described above, but because its use of counters is too complex to perform on more than one thread. Shared memory with Atomic operations was considered to preserve the integrity of these counters , but ultimately discarded because of the logic required to prevent multiple threads from responding when a counter reached a target value. We opted to go with a simpler shared memory approach, similar to rank1 - bucketsizes above. Calculations are performed in shared memory by a single thread, and at the end all threads copy the shared arrays to global memory. This kernel is run with 128 threads instead of 512 because of array sizes.

Communication and DMA

The communication structure of our implementation mirrors that of the original, except for the replacement of MPI_Allreduce with cudaMPI_AllReduce. This allows us to keep all arrays in device memory for rank2, and satisfies a requirement of this project. Unfortunately cudaMPI does not offer versions of MPI_Alltoall or MPI_Alltoallv, so we were forced to DMA all arrays back into host memory before performing these calls between rank2 and rank3.

DMA is performed before rank1 functions and after rank2. Had we chosen to perform rank3 on the GPGPU, we would have needed to perform another pair of DMA transfers to first place the arrays in device memory before rank3, and then again to pull it back out afterwards. We opted not to do this due to problems outlined in the Issues section below, which greatly reduced DMA overhead (which was high - see Results) and implementation complexity.

Results and Analysis

Methods

Timers

We took these results using the timers built into the original Integer Sort algorithm, as well as CUDA timers to time the kernel and DMA operations.

IS contains a more detailed timer mode that separates communication from computation time, which we used extensively. When adding CUDA kernel calls, we were careful to preserve the placement of timer manipulations, and included DMA and other CUDA-related operations in the computation timer. The exception to this was the single cudaMPI call, which was timed with the communication timer.

All IS timers are maximized across the participating MPI nodes, to ensure that stragglers are included in the totals. We mimicked this behavior with our CUDA timers, so all times listed are the maximum experienced by all participating nodes.

Systems

We were limited to a maximum of 8 working CUDA machines for testing - 16 were potentially available, but CUDA runtime and host problems as well as other users limited us to fewer than 16, so we opted for 8. To test both the original and modified (CUDA-enabled) benchmarks side by side, we ran them using the same MPI machinefile, listed below. In addition, for a given class and number of processors, we were careful to run the original and the modified benchmarks as close together as possible, to ensure network and contention issues affected both tests.

We performed the tests on the os40-os56 machines provided, and limited ourselves to the following machines using an MPI machinefile. Prior to testing, we verified that the CUDA systems were up and functional on these systems using the sample matrixMul program provided with the CUDA SDK. Our machinefile:

```
os52.csc.ncsu.edu
os53.csc.ncsu.edu
os54.csc.ncsu.edu
os55.csc.ncsu.edu
os41.csc.ncsu.edu
os42.csc.ncsu.edu
os44.csc.ncsu.edu
os46.csc.ncsu.edu
```

Interference

Unfortunately, these machines are shared for use within the class, and although we used the bash `finger` tool to ensure they were not being used before beginning, network traffic and potentially other MPI runs were unavoidable. These systems do not have `bsub` and do not allow batch jobs to be submitted, and we were unable to ascertain any other mechanism to ensure exclusive execution.

We did repeat the tests multiple times and observed some variance, typically in the order of a second or two but as high as 5 or even 10 seconds in one case. We considered sampling an average of several runs for each value, but given the high administrative overhead of ensuring the systems were not in use, and with no way to sample network traffic, we opted instead to run tests as close together as possible to allow interference to affect but the original and the CUDA run.

Results

CUDA refers to the modified Integer Sort algorithm discussed in this report. Original refers to the original NAS Parallel Benchmark IS code. All values listed are in seconds, except for noted percentages.

It is important to note that these values do not directly add up - Communication + Computation != Total. This is because, as described above in Methods, the IS timers are maximized across all nodes. We kept this approach because we felt it best included the times of stragglers, which are especially relevant here because of heavy reliance on collective communication.

Class=B, NPROCS=4

	Total	Communication	Computation Totals		Computation - CUDA breakdown		
			Total	Total w/o DMA	DMA	Kernel	Host
CUDA	80.43	76.30	4.67	2.79	1.88	2.20	0.59
Original	77.40	75.97	2.72	2.72	-	-	-
% Slowdown	3.91%	0.43%	71.69%	2.57%	-	-	-

Class=B, NPROCS=8

	Total	Communication	Computation Totals		Computation - CUDA breakdown		
			Total	Total w/o DMA	DMA	Kernel	Host
CUDA	59.93	57.66	2.44	1.49	0.95	1.11	0.38
Original	60.56	59.82	1.40	1.40	-	-	-
% Slowdown	-1.04%	-3.61%	74.29%	6.43%	-	-	-

Class=C, NPROCS=4

	Total	Communication	Computation Totals		Computation - CUDA breakdown		
			Total	Total w/o DMA	DMA	Kernel	Host
CUDA	302.52	286.11	18.53	11.21	7.32	8.87	2.34
Original	307.08	300.77	11.83	11.83	-	-	-
% Slowdown	-1.48%	-4.87%	56.64%	-5.24%	-	-	-

Class=C, NPROCS=8

	Total	Communication	Computation Totals		Computation - CUDA breakdown		
			Total	Total w/o DMA	DMA	Kernel	Host
CUDA	262.62	254.41	9.55	5.88	3.67	4.41	1.47
Original	253.10	250.03	5.29	5.29	-	-	-
% Slowdown	3.76%	1.75%	80.53%	11.15%	-	-	-

Analysis

Communication

It is obvious from these results that the vast majority of execution time is spent on communication, both in the original implementation as well as our CUDA-enabled code. Although only 3 MPI calls are made, these are all expensive collective communication routines - MPI_Allreduce (here as cudaMPI_Allreduce), MPI_Alltoall and MPI_Alltoallv. We discuss this further in our Issues section below, but the bottom line is that any gains/losses from the CUDA implementation are minor in the face of relatively huge communication costs.

In theory, the communication costs should be identical across the two implementations, with the exception of any DMA actions managed by the cudaMPI routines. However, we did not see this in practice - in 2 out of the 4 cases, the CUDA code actually performed communication faster, and in the other two cases the difference was very small. We attribute this to network interference, and not any significant difference in the two implementations.

DMA

When examining computation costs, we were first struck by the large slowdown from the CUDA code. We knew some of the issues discussed below (atomic operations, serial loops) could result in inefficiency, but it was only when we began timing DMA separately that the picture became clear. When DMA is removed from the computation time, the speedup difference between computation times for the original and CUDA implementations drops from 55-80% to 2-12%, which is quite significant.

This shows that while the CUDA kernels may not be significantly faster than the original algorithm, at least the kernels themselves are not responsible for this sizeable slowdown. DMA is clearly not a trivial investment, and indeed is equivalent to 80-85% of the time spent performing computations in the CUDA kernel.

Using only 4 processors incurred exactly double DMA costs of 8 processors, which was due to the twice-as-large arrays needed. This shows that DMA costs scale linearly, which was known but is interesting to observe.

Computation

Ignoring DMA costs, the time of our computation algorithms are slightly higher than the original. We found this puzzling, but feel this is probably due to the problems discussed below in the Issues section. Of the kernel functions we optimized, only two were ideal for parallelization, while the other two were serial and were implemented as kernels to avoid additional DMA overhead. Any benefit we gained from parallelization was most likely lost to either kernel management overhead (startup, shutdown, synchronization) or slowdown in the serial kernels. Since the GPGPUs operate at a lower clock speed than the hosts, slowdown is assured when operating in serial.

Issues

We believe these issues collectively prevented us from achieving considerable speedup with our implementation. The algorithm-related limitations are the most damaging, but all the below problems are inter-related.

Parallelization

Algorithm-Related Limitations

As mentioned in the Results - Analysis section above, overall IS performance, at least on the network between the CUDA machines we needed to use, is highly dependent on communication. Computation represents only a very small portion of overall execution time, which makes IS a poor target for CUDA optimization in the first place. A different system setup with a much faster network that is free from contention would have been ideal, but we cannot definitively say if this would have significantly improved our results.

In addition, IS's rank function is definitely not embarrassingly parallel, with 3 out of 10 loops impossible to parallelize at all (aside from shared memory usage), and 3 out of 4 large parallelizable loops suffering from the contention problem discussed below. Ultimately, sorting random numbers instead of a predetermined input set, combined with the use of bucket sort instead of an algorithm with a more predictable output set such as merge sort, hamstrung our efforts by causing threads running concurrently to affect each other in a significant fashion. Indeed, only the initialization loops fit this embarrassingly parallel optimum, leading us to believe that IS's rank() was a poor target.

Finally, and significantly, cross-dependence between different iterations of the rank() function terminated our efforts to perform these iterations in parallel, as we'd first discussed. If possible, this would have provided significant speedup, but the complex nature of inter-node MPI communication coupled with data dependence between executions of rank(), especially with regard to verification, made this impossible. Had IS instances only communicated at the end of all their rank() executions, instead of multiple times within rank(), this may have been possible, but our original assumption of feasibility was based on a lack of familiarity with the IS architecture.

Rank3 Design Problems

The rank3 section of rank contained two loops that we would have liked to parallelize, and one of these, clearwork, doesn't suffer from the contention problems discussed below, making it very attractive. However, we choose not to perform this parallelization for two reasons – implementation difficulty with some pointer math performed in rank3, and high DMA costs we would have incurred due to the all-to-all communication functions required between rank2 and rank3, which could not be performed using device memory.

Rank3 performs some pointer math, which is used internally in all iterations and passed outside to the verification functions on the final iteration. We struggled considerably with porting this behavior to CUDA, with no clear cause. Because CUDA memory addresses don't map directly to host memory, we were forced to pass out an offset instead of the adjusted pointer itself. Confusingly, we were unable to successfully achieve this – although the offset should allow us to re-calculate the desired pointer outside of the CUDA kernel, we kept getting unsuccessful verifications when using it.

Secondly, and more importantly, performing rank3 in a CUDA kernel instead of the host would have incurred a second DMA transfer cost. Since rank3 is implemented on the host, the Computation - CUDA breakdown: Host column of the Result tables roughly represents rank3 execution time (along with timer management), and in all cases is 30-50% of the time spent on DMAing. Thus, an additional DMA transfer in and out would in no way compensate for any speedup gained by porting these functions to CUDA kernels.

Luckily, our troubles with pointer math led us to put this section aside temporarily, and before we returned to it we took timer measurements and realized the extent of DMA overhead. However, if the MPI related issues, discussed below, that force us to perform an additional DMA transfer were to be solved, this section of rank would likely offer at least some speedup.

Contention and Atomic Operations

In the two major loops that we both wanted to and were able to parallelize, we hit problems with race conditions which ultimately necessitated the use of atomic operations, which we discuss above. This introduced significant performance costs, which limited the effectiveness of parallelization. As we scaled up the thread count, this performance cost became readily apparent: between 1024 and 2048 blocks of 512 threads each, we saw no change in performance.

We believe this high penalty is due to a higher degree of contention and thus delay from atomic operations offsets the smaller number of operations performed by each thread. Whatever the cause, the necessity for Atomic operations was a serious blow to our parallelization efforts, especially because they occurred in the most promising hotspots. Had we parallelized the rank3 loops, one of the two parallelizable loops, indivkeypop, would have suffered from this problem as well.

Memory Architecture

Global vs Shared Memory

Ideally, as many operations as possible would be performed in CUDA's shared memory, which is much faster than global memory. However, two primary issues limited our usage of shared memory, discussed below.

The first issue is that shared memory is limited to 16k per thread block, including local variables, which is much too small for many of our applications. In the best cases we could use it for a few arrays of 1024 integers, but nowhere could we use it to store the large arrays that would highly benefit from it. The speed of shared memory, especially when using atomic operations as discussed below, is highly attractive, but with input arrays of the size required by IS class B and C, this was outside the realm of possibility in almost all cases.

The second issue is that shared memory is only visible within a thread block. In the majority of situations, we would wish to first copy global arrays into shared arrays, then perform necessary operations on them, and finally copy them back out to global memory. This would be a significant issue when scaling to multiple thread blocks, in the simple case of needing to aggregate multiple shared arrays at the end of the kernel execution, as well as the much more complex case of needing to aggregate every loop iteration due to dependence on the sum. Because of shared memory limits, we did not have too much trouble because of this issue, but had we been able to use large shared memory arrays, this most certainly would have been a significant design hurdle.

In the two cases we could use shared memory, we were able to provide small speed benefits to serial operations on the GPU, primarily by using many threads to perform global to shared and shared to global memory movements, while using a single thread to perform the actual computation. This is highly inefficient, as the additional 511 threads wait at a barrier while the 1st thread performs all the computation, but this method was used as a stopgap to avoid additional DMA penalties incurred by performing the computation on the host.

MPI and DMA Operations

MPI's collective functions are known to be performance-poor, and we definitely saw that here. This is a general tenant but seems to especially be the case on the network between the test systems we used. Unfortunately, the algorithm would need to be rewritten to remove these calls entirely.

The cudaMPI library does not provide corresponding GPU functions for MPI_Alltoall and MPI_Alltoallv. The result of this is that we must DMA from the device any associated arrays before performing these functions on the host, and then DMA back to the device upon finishing. Because of this penalty, and as discussed in the Rank3 Design Problems section above, we opted not to implement rank3 in the CUDA kernel. We are not confident that implementing all rank functions as CUDA kernels would have offered significant performance benefits, but we would have liked the opportunity to evaluate this path using execution timers.

One solution to this would have been to replace the all-to-all routines with a collection of equivalent cudaMPI calls, such as cudaMPI_Isend and cudaMPI_Irecv. However, we decided against this approach, because of the complexity of implementation as well as a lack of knowledge about the overhead of running many cudaMPI calls independently instead of as a single packaged function. Given more time, we may have considered appending to the cudaMPI library with collective routines of our own.

Future Work

Given the problems discussed in the above Issues section, we find it hard to recommend any further optimization be performed on the NAS PB Integer Sort benchmark.

However, if such an approach is desirable, the first step would be to port the required all-to-all MPI communication routines into the cudaMPI library, so they can be used with GPU memory.

Once this is complete, rank3's pointer math issue would need to be resolved, but given more time we do not see this being a problem.

Finally, and most importantly, test systems connected to a faster network would need to be used to effectively realize larger speedups.

References

[1] NAS Parallel Benchmark. http://www.nas.nasa.gov/Resources/Software/npb.html

[2] NVIDIA CUDA. http://www.nvidia.com/object/cuda_home.html

[3] cudaMPI library. http://www.cs.uaf.edu/sw/cudaMPI/