Utilizing Unused Resources To Improve Checkpoint Performance

Ross Miller
Oak Ridge Leadership Computing Facility
Oak Ridge National Laboratory
Oak Ridge, Tennessee
Email: rgmiller@ornl.gov

Scott Atchley
Oak Ridge Leadership Computing Facility
Oak Ridge National Laboratory
Oak Ridge, Tennessee
Email: atchleyes@ornl.gov

***Abstract*—Titan, the Cray XK7 at Oak Ridge National Laboratory, has 18,688 compute nodes. Each node consists of a 16-core AMD CPU, a NVIDIA GPU, and 32GB DRAM. In addition, there is another 6GB of GDDR DRAM on each GPU card. Not all the applications that run on Titan make use of all a node's resources. For applications that are not otherwise using the GPU, this paper discusses a technique for using the GPU's RAM as a large write-back cache to improve the application's *perceived* file write performance.**

I. Introduction

Most HPC applications' I/O exhibit a very 'bursty' pattern. That is, there will be long periods with no file output while the application is in a compute phase, followed by a period of intense output and no computation when the application writes its checkpoint or output files. At ORNL, the filesystem is a center-wide resource shared with multiple systems including a Cray XK7, a Cray XC30, analysis clusters, and WAN-connected data mover nodes. Each of these competing systems place a varying load on the filesystem and thus the performance that individual users see can vary dramatically within a job and between jobs.

Having a sufficiently large, node-local, write-back cache could improve an application's performance in two ways. First, if the cache is large enough to hold an entire write and the individual writes are far enough apart, then the application never has to wait on the filesystem. It can write to the cache and then resume its computations while the cache drains to the filesystem. Secondly, write performance should be more predictable. From the application's point of view, write speed would only limited by how fast data can be copied into the cache, and since the cache is local to the node, it is not effected by outside influences.

Titan, the Cray XK7 at Oak Ridge National Laboratory, has 18,688 compute nodes. Each node consists of a 16-core AMD Interlagos CPU, an NVIDIA GPU and 32GB DRAM. In addition, there is another 6GB of GDDR DRAM on each GPU card. Not surprisingly, some applications do not use all the resources on a compute node.

To start with, during 2014, approximately 50% of the compute time on Titan was used on applications that did not use the GPU at all.[7] There may be good reasons for this. Utilizing the GPU requires changes to the application and it is possible the developers have not had the time or funding to do that. It is also possible that the type of computations the application performs simply do not lend themselves to GPU-based acceleration (i.e. not enough exposed parallelism). Regardless of why, it is clear that not all applications are benefiting from the GPU.

Secondly, the Interlagos has eight Bulldozer units, each of which has two full integer units and a shared floating-point unit. Some users with floating-point intensive applications run jobs using only eight processes per node (one process per Bulldozer) to avoid oversubscribing the floating-point units. This actually leaves eight integer cores available so long as they primarily execute integer instructions.

The unused GPU memory and integer cores can be used to implement a write-back cache without taking resources away from the main application. This paper demonstrates a technique for providing such a cache to applications. Runs were made on Titan using a synthetic benchmark to demonstrate performance both without such a cache and with a cache that utilizes DRAM on the GPU card. The results show definite performance benefits.

II. Design And Implementation

A. Overview

To test this idea, the authors created a synthetic benchmark application that consists of two executables: the main application which uses MPI and simulates the I/O patterns of a typical HPC application and a daemon process that is started on each compute node. The daemon has two tasks: manage the GPU memory and copy data from GPU memory out to a file. Note that the daemon does *not* copy the data into the GPU memory; that is handled by the individual ranks of the main application. There needs to be exactly one daemon process on each compute node regardless of how many MPI ranks are running on each node.[1]

With the daemon process running on each node, the application begins its main execution. In this case, since it is just a synthetic I/O benchmark, it performs no actual computation; it simply writes a specified amount of data and

[1]Due to the limitations of aprun, starting this daemon is somewhat involved. See section II-C.

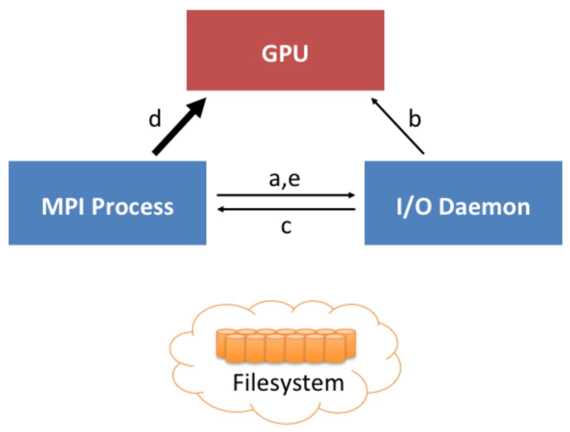

Figure 1. Buffering data to GPU memory

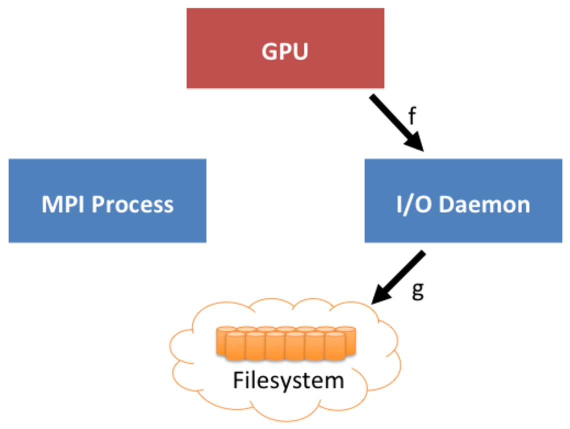

Figure 2. Draining data from GPU memory

then sleeps for a specified amount of time. For these tests, the sleep time was deliberately calculated to allow enough time for the data in GPU memory to drain.

B. Communication Between the Application and Daemon

The main application and daemon coordinate with each other using short messages provided by the Common Communication Interface (CCI).[4] Bulk data is copied to and from GPU memory using `cudaMemcpy()`.

Writing data from the application is a seven step process. When it is time to write, the application sends a short message to the daemon using CCI. This message is a request to write data, and contains the amount of data and the starting offset into the output file.[2] (Figure 1a)

When the daemon receives this message, it will attempt to allocate memory on the GPU. Assuming the allocation succeeds, the daemon will use `cudaIpcGetMemhandle()` to create a memory handle for the allocation. (Figure 1b)

The daemon will then reply to the application with the size of memory that was actually allocated and the memory handle that the application can use to access the GPU memory. Note that the size value may be less than what the application asked for. (Figure 1c)

Upon receiving the reply, the application opens the memory handle and converts it back to a standard pointer. The application then uses `cudaMemcpy()` to copy the specified amount of data up to the GPU memory. (Figure 1d)

When the copy has completed, the application closes the memory handle and sends a message back to the daemon saying that this particular write has completed. (Figure 1e) If the application has more data to write, it can repeat the previous steps, or else it can continue with its calculations.

Note that at this point the data has not made it out to the filesystem yet; it is just sitting in GPU memory.

The daemon maintains a pair of lists. One list is the blocks of GPU memory that have been assigned to the ranks of the main application, and which the ranks are presumably copying data into. The second list is for all the blocks of memory for which the 'write complete' message (step e) has been received. When the daemon thread that handles CCI messages receives a 'write done' message for a block, it moves that block from the first list to the second list. It then notifies the background write thread(s) that the block is now ready to be written to the filesystem.

The daemon maintains one or more background threads that are responsible for copying data out of GPU memory into normal system memory and then writing that data to the filesystem. These threads allow the writes to the filesystem to happen asynchronously while the main thread handles all the CCI messages. Since it is impossible to write directly from GPU memory to the filesystem, each thread will copy a block of data from GPU memory into a buffer in main memory. (Figure 2f) It will then write the data to the output file at the location specified in the original request message. (Figure 2g) Once data has been written to the filesystem, the write thread frees the GPU memory so that it can be used in another request.

Note that each write thread in the daemon has its own buffer in system memory. Each of these buffers is deliberately kept fairly small, 16MB in this test, in order to conserve resources. Testing done on Titan shows that 16MB block sizes are large enough for `cudaMemcpy()` to give good performance.[1] Larger buffers would simply use up

[2]For this benchmark, the daemon names output files based on the MPI rank of the requestor. For production code, something more flexible will obviously be needed.

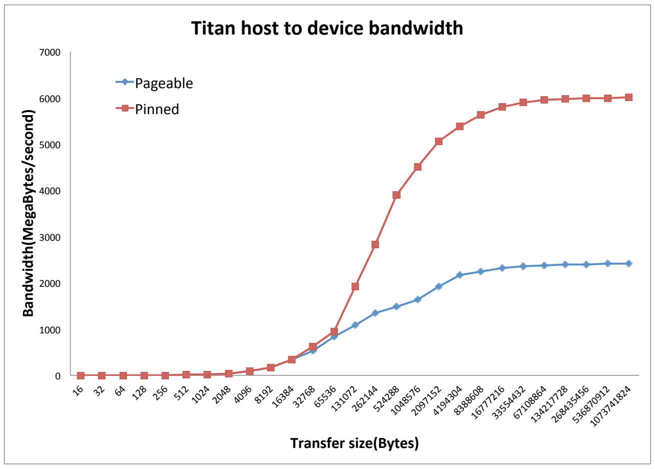

Figure 3. Bandwidth between system memory and GPU memory as a function of write size[1]

memory without improving performance noticeably. (See Figure 3.[3])

Also note that the daemon limits the amount of memory allocated to any single request to 16MB. This was done to provide some limited load-balancing. If an application wants to write 512MB, it will therefore have to make 32 separate requests. If other ranks on the node are trying to write at the same time, their requests will all be interleaved and hopefully no rank will be starved out.

C. *Cray-Specific Details*

There are a few peculiarities of the Cray environment that need to be taken in to account in order for this software to work properly. The first peculiarity has to do with limiting access to the GPU and associated hardware. The CUDA runtime can be configured to limit access to the GPU to a single thread, to multiple threads from a single process, or to multiple threads from multiple processes. On Titan, if no special options are given to qsub, the CUDA runtime will be configured for a single thread. Since software described in this paper requires access to the GPU from multiple processes, users must switch to the appropriate compute mode. This is done by passing the flag "-l feature=gpudefault" to qsub.[4]

The second peculiarity has to do with actually starting the daemon process. Normally, executables are started on the compute nodes by *aprun*, Cray's replacement for mpirun. By default, aprun is designed to start a single executable in single-program-multiple-data (SPMD) mode, not to start two different executables. Aprun does support multiple-program-multiple-data (MPMD) mode, but it will not start two different executables on the same node (i.e. each program executes on non-overlapping subsets of nodes). The solution is to have one rank on each node call `fork()` and then `execve()`. Since MPI purposely hides the compute topology from the application, deciding exactly which ranks call fork() takes a little more work. Specifically, all ranks will attempt to open a file using the `O_CREAT` and `O_EXCL` flags. The name of the file is taken from the compute node's host name. This combination of flags and filename ensures that exactly one rank on each node successfully opens the file. The ranks that succeed will then start the daemon processes.

III. Analysis

A. *Experiment Setup*

In order to evaluate this software, the authors ran two series of tests: one that ran the application with 8 ranks per node and another that ran with 16 ranks per node. Each series included tests of the daemon using 1, 2, and 4 background write threads. Both series also tested the write performance just using standard `write()` calls in order to get some baseline data for comparison.

For the first test series, the application was configured for 8 ranks per node. As mentioned in the introduction, 'real' applications will often run on Titan using only 8 ranks per node because the AMD processor in the compute nodes only has 8 floating point units. From the authors' perspective, this has the advantage of leaving 8 integer-only cores to run the daemon in multi-threaded mode. For this test series, aprun was configured to pin the application ranks to the even numbered cores and the daemon was configured to pin its threads to odd numbered cores. Furthermore, the daemon thread that processed the CCI messages was run in a mode where it continuously polled for new messages. This provided the lowest possible latency for message handling, but at the cost of effectively consuming one core.

For the second test series, the application was configured for 16 ranks per node. The daemon was again run with 1, 2, and 4 write threads. For this series, the daemon thread that handled the CCI messages was run in a mode where it would block waiting on a message. This added some latency to the message processing, but left that core free to perform useful work when there were no messages to process. Also for this series, no core pinning was used on the daemon.[5]

For both test series, the daemon was configured to write each rank's data to a separate file and the test measured the

[3]Figure 3 differentiates between 'pinned' memory and 'pageable' memory. Pageable memory is allocated using regular `malloc()` or `new` calls. Pinned memory is allocated using `cudaMallocHost()` or `cudaHostAlloc()`.[2] This work uses pinned memory exclusively.

[4]It may seem strange to have to explicitly switch to a mode called 'gpudefault'. The naming scheme comes from the CUDA documentation. On a standard CUDA install, the GPU would in fact default to allowing access from multiple processes. Titan is configured differently, however. Thus the need to explicitly tell qsub to switch the GPUs to the 'default' mode.

[5]In practical terms, a user would probably not want to use multiple write threads on the daemon if his/her application was running 16 ranks/node since it would oversubscribe the cores. The authors tested the daemon with 2 and 4 threads partially out of curiosity and also to keep the two test series as similar to each other as possible.

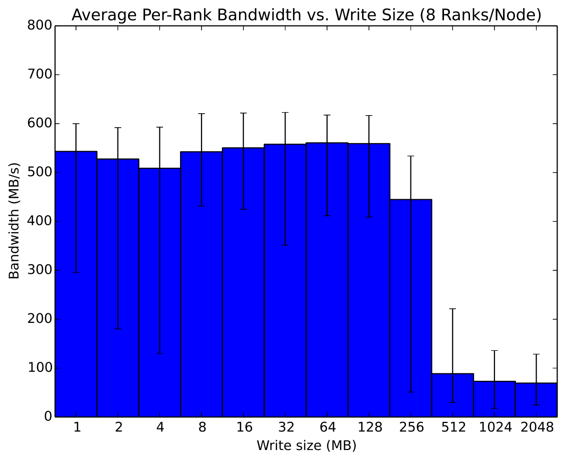

Figure 4. Average per-rank write bandwidth using standard `write()` calls. 8 ranks per node. No GPU memory cache.

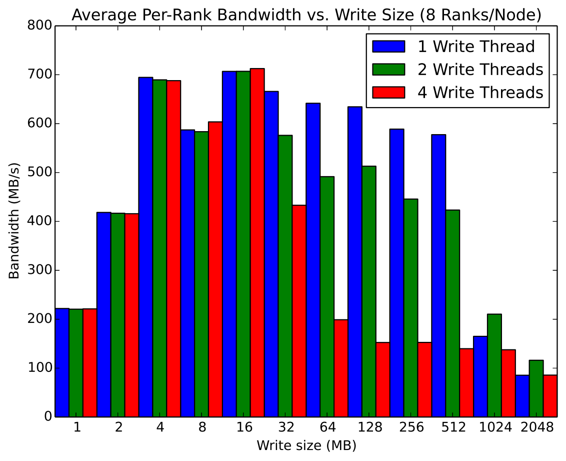

Figure 5. Average per-rank write bandwidth. 8 ranks per node.

application's perceived write performance as the write size increased. Note the word 'perceived'. What was actually measured was how long it took for each rank of the application to copy its data into GPU memory. The GPU cards in each of Titan's compute nodes have 6GB of RAM, though in practice only a little over 5GB is actually available to the user. This means that for the first test series, with 8 ranks per node, write sizes of up to 512 MB were small enough for all ranks' writes to fit into GPU memory. For the second test series, using 16 ranks per node, write sizes up to 256MB would fit. For both test series, once the write size exceeded the available GPU memory, the ranks would have to wait until the daemon was able to drain data out of GPU memory and into the filesystem page cache or over the network.

B. *Results*

Figure 4 shows the results of a baseline test. The colored bars show the average throughput for 128 nodes and the error bars display the minimum and maximum values. In this test, the application wrote to the filesystem with standard `write()` calls and the GPU memory was not used at all.[6] The results provide some baseline numbers that can be used for comparison with the tests using GPU memory.

Note the sharp drop in performance between 256MB and 512MB. On Titan, the Lustre client is configured to allow a maximum of 64MB per OST and to default to using 4 OSTs per file. Given the number of OSTs available, it is statistically likely that no two output files in this test used the same OSTs. In short, write sizes of 256MB or less were cached in system memory using the existing Lustre client cache and the performance of the 512MB, 1GB and 2GB sizes is dominated by the performance of the filesystem. In order for this work to be useful, it must obviously improve on that.

[6]The daemon process was not even started for this test.

Figure 5 shows the remaining results of the first test series. As stated earlier, this series was run with 8 ranks per node and with 1, 2, and 4 write threads in the daemon. Each column is the mean of 64 nodes' perceived per-process throughput. The graph shows a number of interesting features. The first and most obvious, is that multiple write threads decrease performance. Exactly why this is so is unclear, but it appears that having multiple threads read from GPU memory interfered with the individual ranks' ability to copy data into GPU memory.

Concentrating on the single thread performance, it is clear that the application benefits from using GPU memory out to the 512MB write size. This makes sense since eight ranks each writing 512MB is a total of 4GB and that will fit into the available GPU memory. Even at 1GB, the application sees somewhat improved performance because there is enough GPU memory to hold significant fraction of the data to be written. It is not until the 2GB write size that the write performance is dominated by the filesystem's throughput.

Also obvious from the graph is the fact that small write sizes are not particularly efficient. For write sizes less than 4MB, copying data to GPU memory is slower than writing to the Lustre cache. This is not surprising given the results shown in Figure 3.

Figures 6 and 7 show the results for the second series of tests. As noted above, this series used 16 ranks per node, plus the daemon's threads. This meant, of course, that the cores were oversubscribed. Notice that Figure 6 has the same basic shape as Figure 4. The only significant difference is that the reported speeds shown in Figure 6 are approximately half those shown in Figure 4. This is expected, since there are twice as many ranks writing. Again, the baseline test

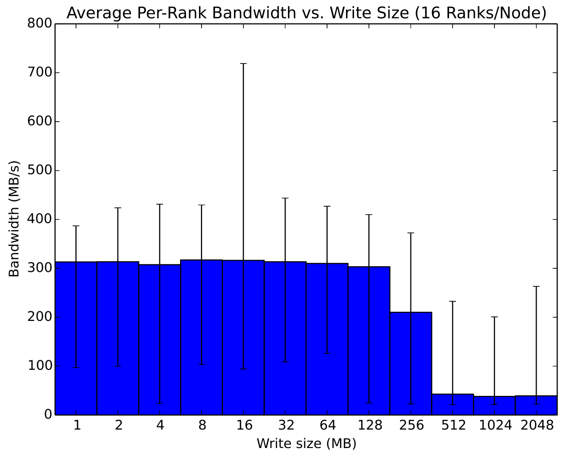

Figure 6. Average per-rank write bandwidth using standard `write()` calls. 16 ranks per node. No GPU memory cache.

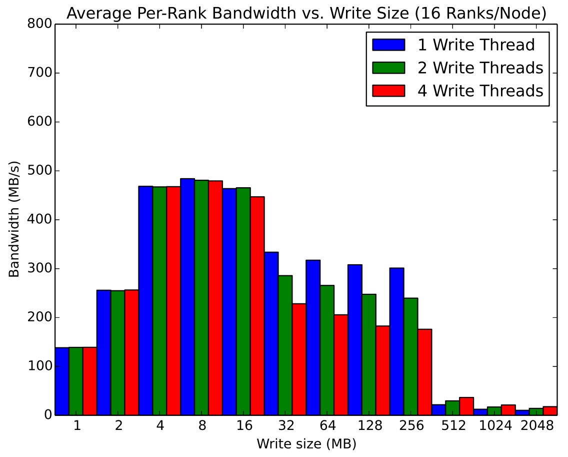

Figure 7. Average per-rank write bandwidth. 16 ranks per node.

uses 128 nodes while each of the daemon with 1, 2, and 4 write threads present the mean of 64 nodes each.

Figure 7 shows that again, the per-rank bandwidth is about half that shown in Figure 5 because there are twice as many ranks. It is also clear that the application ranks see good write performance up to the 256MB write size, which is the largest size that will entirely fit into GPU memory. It is also clear that, as in the first test series, writes must be at least 4MB in order to get reasonable performance copying the data into GPU memory.

Looking at figure 7, note the drop in performance between 16MB and 32MB and the slight downward trend from 32MB to 256MB. This pattern also appears in Figure 5, but is much more obvious in Figure 7. The authors hypothesize that this is related to the response time of the daemon. As mentioned earlier, the daemon allocates GPU memory in blocks of up to 16MB. Thus, if one of the ranks wants to write more than 16GB, it will need to go through more than one request/response message cycle. However, the daemon only has a single thread to handle these messages. When multiple ranks are sending message, they will necessarily have to wait while the daemon services any previous messages and this slows their overall performance. This is more noticeable in Figure 7 because there are more ranks making requests. In short, it appears that improving the response time of the message handling thread would be beneficial.

IV. Conclusion

For the first test series (8 ranks/node), the results show a clear performance improvement compared to writing directly to the filesystem. More data can be cached in GPU memory than in the Lustre client-side cache and the average write speed is, surprisingly, slightly higher. It is, however, necessary to use relatively large writes. Comparing Figure 4 and Figure 5, it can be seen that write sizes must be at least 4MB before transfers to GPU memory outperform writes to the Lustre cache.

Note that these tests were run with file striping left at its default value of 4. In theory, writes to the filesystem could be improved by increasing the number stripes for each file, but that requires more system memory, which necessarily leaves less for the application.

The results for the second test series (16 ranks/node) are less clear. As with the first test series, write sizes need to be at least 4MB Looking closely at Figure 6 and Figure 7, it appears that write sizes that can fit into GPU memory perform slightly better than just writing to disk. However, write sizes that exceed the capacity of the GPUs memory are actually a little slower than writing straight to the filesystem.

There are several other complications to consider, though. First, Since the test application did no actual computation, there were plenty of cores available for the daemon to execute on. In a real HPC application, this would not be the case and the application's performance would almost certainly be negatively affected by the context switches. On the other hand, an application that is running 16 ranks/node is unlikely to leave much system memory free for use by the Lustre client-side cache. So, the improvement from caching writes in GPU memory might offset the disruption caused by oversubscribing the cores. Whether or not this technique provides a net performance improvement in such a configuration will probably depend on the particular application.

An interesting compromise might be to have the application run 15 ranks/node instead of 16. That will leave 1 core available for the daemon. Both threads could be pinned to that core, with the message processing thread given higher priority. It would also be possible to use the core specialization option to aprun to schedule OS tasks on that core. Compared to an application running lots of MPI collectives, the daemon is not particularly sensitive to short interrupts like that. This configuration has not been tested and the authors are unaware of any application that routinely runs with 15 ranks/node.

In general, the results show that using the GPU memory to cache filesystem writes provides approximately the same performance as the Lustre client-side cache. The difference is that the Lustre cache uses system memory which may not actually be available. Even if it is, the user must ensure that writes are spread across a sufficient number of OSTs to make use of it. Neither of these conditions apply when using GPU memory as a cache.

V. Future Work

The authors work in a group that is not dedicated to pure research; part of the group's mandate is to deploy useful software in the production environment. As such, the most important piece of future work is to convert this demonstration code into a package that can actually be used by other applications running on Titan. In the introduction, the authors noted that one of the reasons some applications have not been converted to use the GPU for calculations is that the scientific applications' developers have not had time and/or funding. With that in mind, it is impractical to expect those same developers to make major changes in order to use this code.

The authors are considering a number of methods of packaging this code to make it easy for application developers to integrate into their own applications. One possibility is to intercept certain POSIX function calls, such as `open()` and `write()`, among others. While somewhat tricky to implement, this technique has been used with good success by the Mercury Posix project.[6] A second possibility is to modify one or more I/O libraries that are popular with applications, such as the NetCDF library, to make use of this code. This has the advantage of requiring zero code changes by the application developers. Properly maintaining the modified I/O libraries could be a real problem, though. A final possibility is to integrate this code into the Functional Partitioning project that the authors and their colleagues are working on.[5] Obviously, the authors could choose more than one of these alternatives.

Besides deploying this code in a production environment, there are a few features the authors would like to add. The simplest is to add a user-controlled option to limit the amount of GPU memory the daemon will try to allocate. As currently written, the daemon will allocate all the memory it can. That could cause problems if the application wants to use the GPU for calculations. By allowing the user to set a hard limit on the amount of memory the daemon will use, it becomes possible for this software to coexist with applications that use the GPU for calculations.

Another feature the authors would like to add is the ability to use regular system RAM in addition to the GPU RAM. It is well known that some applications do not use all the RAM on the compute node. It is debatable whether letting the daemon use the memory is better than simply letting the operating system use it as Lustre client cache or OS page cache. However, as was discussed in Section III and Section IV, the Lustre client-side cache has fairly low limits on the amount of dirty pages it will allow. (The Linux kernel page cache has similar low limits on dirty pages.) Some very basic initial testing implies that allowing the daemon to allocate memory may be more useful than letting the OS use the same memory for page cache, but further testing is required to be certain.

Lastly, the next big supercomputer at ORNL has already been announced. While some details have yet to be finalized, one thing that has been decided is that each node will have approximately 800GB of nonvolatile memory (NVM).[3] Exactly how applications will use this NVM memory is the subject of much research and debate. With only minor changes, this software could write to the NVM instead of (or in addition to) the GPU memory. Assuming the ease-of-use goals mentioned above are met, the authors feel this would allow application developers to quickly modify their applications to use the NVM as a large burst-buffer, even as those developers look for other, more effective ways to make use of the hardware.

Acknowledgment

This work was supported by the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory, which is managed by UT Battelle, LLC for the U.S. DOE (under the contract No. DE-AC05-00OR22725).

References

[1] "Accelerated computing guide," https://www.olcf.ornl.gov/support/system-user-guides/accelerated-computing-guide/#2981.

[2] "How to optimize data transfers in cuda c/c++," http://devblogs.nvidia.com/parallelforall/how-optimize-data-transfers-cuda-cc/.

[3] "Summit," https://www.olcf.ornl.gov/summit.

[4] S. Atchley, D. Dillow, G. Shipman, P. Geoffray, J. M. Squyres, G. Bosilca, and R. Minnich, "The common communication interface (CCI)," in *19th Annual IEEE Symposium on High-Performance Interconnects*, August 2011.

[5] M. Li, S. S. Vazhkudai, A. R. Butt, F. Meng, X. Ma, Y. Kim, C. Engelmann, and G. Shipman, "Functional partitioning to optimize end-to-end performance on many-core architectures," in *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 1–12. [Online]. Available: http://dx.doi.org/10.1109/SC.2010.28

[6] J. Soumagne and Q. Koziol, "Milestone 6.1: Posix function shipping demonstration," The HDF Group, Tech. Rep., 2013. [Online]. Available: https://wiki.hpdd.intel.com/download/attachments/12127153/M6.1_PosixFunctionShipping-Demo-v3.pdf

[7] J. C. White, "High performance computing facility operational assessment, 2014 oak ridge leadership computing facility," Oak Ridge National Laboratory (ORNL); Center for Computational Sciences, Tech. Rep., 2015. [Online]. Available: http://info.ornl.gov/sites/publications/Files/Pub54966.pdf