Comparing Managed Memory and UVM with and without Prefetching on NVIDIA Volta GPUs

[*]An analysis of different memory allocation techniques available for NVIDIA GPUs

Rahulkumar Gayatri
NERSC
Lawrence Berkeley National Laboratory
Berkeley, USA
rgayatri@lbl.gov

Kevin Gott
NERSC
Lawrence Berkeley National Laboratory
Berkeley, USA
kngott@lbl.gov

Jack Deslippe
NERSC
Lawrence Berkeley National Laboratory
Berkeley, USA
jrdeslippe@lbl.gov

***Abstract*—One of the major differences in many-core versus multicore architectures is the presence of two different memory spaces: a host space and a device space. In the case of NVIDIA GPUs, the device is supplied with data from the host via one of the multiple memory management API calls provided by the CUDA framework, such as *CudaMallocManaged* and *CudaMemCpy*. Modern systems, such as the Summit supercomputer, have the capability to avoid the use of CUDA calls for memory management and access the same data on GPU and CPU. This is done via the Address Translation Services (ATS) technology that gives a unified virtual address space for data allocated with *malloc* and *new* if there is an NVLink connection between the two memory spaces. In this paper, we perform a deep analysis of the performance achieved when using two types of unified virtual memory addressing: UVM and managed memory.**

***Index Terms*—GPU, CUDA, managed memory, Unified Virtual Memory (UVM).**

I. Introduction

Unlike multicore architectures, accelerator architectures, such as NVIDIA GPUs, have two distinct memory spaces: one on the CPU and one on the GPU. For a kernel to run on a GPU, data needs to be transferred from CPU to GPU either via an explicit memory transfer made by the programmer or by making the data accessible to both memory spaces via a Unified Virtual Memory system. If the data is transferred by the explicit memory transfer calls provided in the CUDA API [2], such as *cudaMalloc* and *cudaMemcpy*, the data accessed by the GPU travels through the various memory hierarchies in the GPU memory space. These CUDA API calls are in addition to the regular host memory allocation, which adds an additional level of complexity to proper data management.

CUDA also has support for managed memory allocation which allows the user to access the same memory across both the memory spaces without the need for an explicit memory transfer. Data transfer is done by the Memory Management Unit at the granularity of a page. In order to use this feature the memory allocation has to be done via the CUDA API call *cudaMallocManaged*. This feature has been available in CUDA since version 6.0. For convenience, this paper will refer to this methodology as managed memory.

CUDA 9.2 further introduced support for Address Translation Services (ATS) [3] [4] for power platforms. This technology allows GPUs to access CPUs page tables directly. This functionality is supported on Volta GPUs by utilizing the NVLink connections. ATS allows GPUs to access memory allocated on the CPUs via the page tables. A miss in the GPU Memory Management Unit (MMU) will result in an Address Translation Request to the CPU. The CPU looks in its page tables for the virtual-to-physical mapping for that address and supplies the translation back to the GPU. ATS provides the GPU complete access to CPU memory, including memory allocated with the standard host-side functions *malloc* and *new* [3]. This functionality means that on such systems, a programmer does not need to use CUDA API calls to handle memory management between the device and host memory spaces. For convenience, this paper will refer to this methodology as Unified Virtual Memory (UVM). A brief introduction to the usage of UVM on Summit is shown here [7].

One important difference between managed memory and UVM is the granularity at which the data is brought into the GPU memory hierarchies. In the case of managed memory, a memory miss leads to the memory page with the requested data being copied between memory spaces, whereas UVM copies the data from host to device on the granularity of a cache line. UVM is cache coherent and hence would immediately update the CPU memory with the values updated on the GPU side. UVM can adjust the granularity of the data movement with the use of hints from the user such as memory prefetching. This requires the use of a CUDA API call, *cudaMemPrefetchAsync*.

Managed memory also allows the programmer to provide memory prefetch hints. But managed memory would transfer the data back to the host only when there is a request for the data on the CPU.

The goal of the paper is to evaluate the differences in performances of CPU and GPU kernels when UVM memory

NERSC, LBL, DOE

management is used compared to managed-memory implemented through the CUDA API. In order to evaluate these differences, we have designed a synthetic benchmark that allows us to parameterize the accesses on GPUs and CPUs. This benchmark is used to evaluate the performance of both the memory management systems with and without prefetching.

The rest of the paper is organized as follows: Section II will describe the DAXPY kernel that we have used for our benchmark and explore the possible access patterns on the GPU that might effect the performance of kernels on the GPU. In section III, we show the performance of UVM versus managed memory, and present the scenarios when one is better than the other. Our results also present the performance differences between managed and UVM, when both the memory managements have been provided with *cudaMemPrefetchAsync* hints. Finally in section IV, we present our analysis and suggestions on when to use one memory allocation technique over another.

II. DAXPY KERNEL

For this benchmark we use a daxpy kernel which is represented in the equation below :

$$y = y + a * x \tag{1}$$

In equation 1, y and x are two-dimenional vectors of type *double* and their dimensions are N and M respectively. This can be represented by the DAXPY kernel shown in Listing 1.

Listing 1: DAXPY kernel

```
//In .cu file, the daxpy kernel as distributed among the
    threadblocks and threads of a GPU.
__global__ daxpy_kernel (double *x, double *y)
{
 int i = 0, j = 0;
 for(i=0; i<N; i++)
  for(j=0.x; j<M; j++)
   y(i,j) += a*x(i,j);
}
```

Listing 1 shows the sequential implementation of our DAXPY kernel. The first loop iterates over the N rows of x and y vectors whereas the second loop iterates over the M columns within each row.

Listing 2: DAXPY kernel

```
//In .cu file, the daxpy kernel as distributed among the
    threadblocks and threads of a GPU.
__global__ daxpy_kernel (double *x, double *y)
{
 int i = 0, j = 0;
 for(i=blockIdx.x; i<N; i+=gridDim.x)
  for(j=threadIdx.x; j<M; j+=blockDim.x)
   y(i,j) += a*x(i,j);
}
```

In this study, the kernel is built to distribute the N dimension across the thread-blocks and the M across threads with each thread-block as shown in Listing 2. Listing 3 shows the memory setup for the benchmark.

Listing 3: Setup the memory

```
//The actual benchmark
void benchmark(double *x, double *y)
{
//If using managed memory allocate x and y vectors with
    cudaMallocManaged
#if managed_memory
  cudaMallocManaged(x,N*M*sizeof(double));
  cudaMallocManaged(y,N*M*sizeof(double));
//Else if using UVM allocate memory using malloc
#elif defined(UVM)
    x = malloc(N*M*sizeof(double))
    y = malloc(N*M*sizeof(double))
#endif
...   // Continued in Listing 3
```

The x and y vectors in benchmark are stored as $N*M$ sized *double*-type arrays. They are allocated by calling *cudaMallocManaged* when testing managed memory or *malloc* when testing UVM. The data is then initialized to a starting value and the kernel is launched as shown in Listing 4.

Listing 4: Launching DAXPY

```
...   // Continued from Listing 2
//outer - represents number of times the loop of GPU-CPU is
     iterated over
//inner - represents the number of times the given GPU
    kernel is launched
  dim3 grid(N,1,1);
  dim3 threads(32,1,1);
  for(outer){
    for(inner){
     daxpy_kernel<<<grid,threads>>>(x,y);
    } //end inner
    TouchOnCPU(y);
  } //end outer
} \\ End of function benchmark
```

The DAXPY kernel is launched with N thread-blocks, each with 32 threads. The M dimension is used to independently change the size of the data set being investigated. The loop structure in listing 4 has been designed to control data movement of the x and y arrays on CPUs and GPUs: the outer loop controls the number of times the memory is transferred between host and device while the inner loop controls the number of times the data set is consecutively accessed on the GPU, either on a page-level granularity when allocated via managed memory or on a cache-level granularity when allocated via UVM.

The data is returned to the CPU through the implementation of Listing 5.

Listing 5: Benchmark code

```
//The kernel touches data on CPU
void TouchOnCPU(double *y)
{
 int i = 0, j = 0;
 for(i=0; i<N; ++i)
  for(j=0; j<M; ++j)
   y(i,j) -= 0.5;
}
```

This is a simple calculation performed on every point in the y array to ensure the data has fully returned to the host before continuing.

III. RESULTS

The results presented in this section are collected on the Summit supercomputer [1] from Oak Ridge National Laboratory (ORNL). A Summit node is comprised of two sockets, each containing one IBM Power9 CPU [5] and three NVIDIA Volta GPUs (V100) [6]. The Power9 CPUs and NVIDIA GPUs

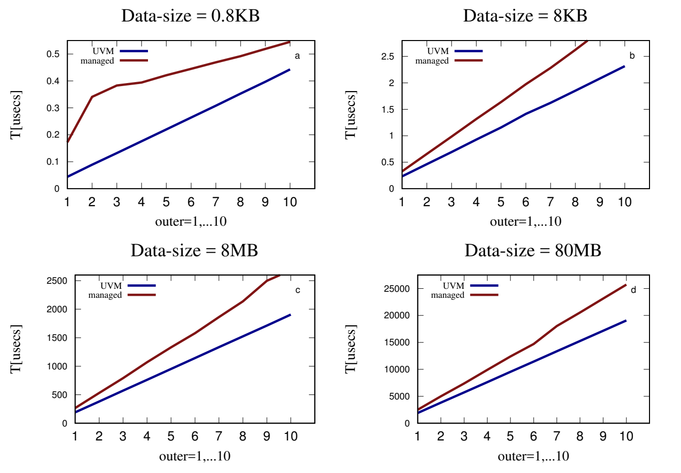

Figure 1: Varying the number of CPU memory touches (*outer*), using 2 consecutive kernel calls. (*inner*).

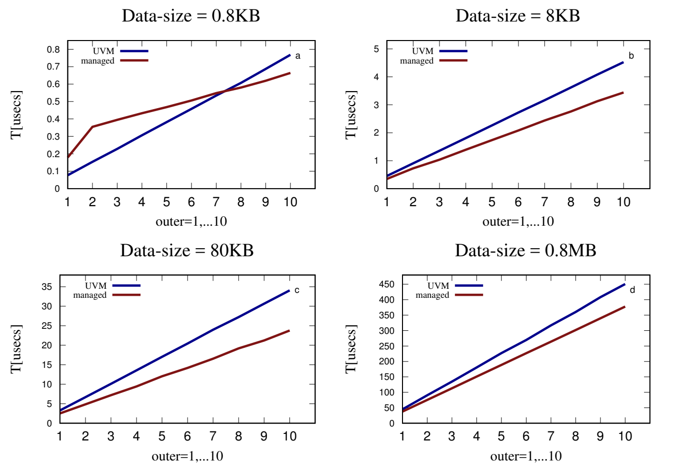

Figure 2: Varying the number of CPU memory touches (*outer*), using 4 consecutive kernel calls. (*inner*).

are connected via high speed NVLink. Each of the V100 GPUs has 16GB of high bandwidth memory and 6MB of L2 cache shared between its 80 Streaming Multiprocessors (SM). Each SM has a 128KB block of memory that is divided among the L1 cache and shared memory [8].

The study begins by exploring the conditions under which each memory space is preferable in III-A. Later we discuss the effects of prefetching in III-B.

A. UVM vs Managed

First, UVM and managed memory are examined for different memory movement strategies by changing the size of the *inner* and *outer* loops. The *inner* loop controls the the number of times a memory location is accessed on the device before being updated on the host. Meanwhile, the *outer* loop controls the number of times a memory location is transferred between host-and-device.

Each of these tests are performed for a variety of data sizes. In the results, the X-axis represents the number of *inner* or *outer* accesses whereas the Y-axis represents the time of the loop structure shown in Listing 3. Data size refers to the amount of data accessed by each of the thread-blocks, *data_size=M*sizeof(double)*.

1) Varying the number of host-to-device transfers: The effect of increasing the number of host-to-device data transfers was tested by fixing the *inner* loop to 2 and 4 and varying the size of the *outer* loop. The results are shown in Figures 1 and 2.

From figure 1 and 2 we can conclude that UVM consistently performs better when only 2 consecutive GPU kernel calls are performed before the data is accessed back on the CPU. However, when the GPU kernel is called for 4 consecutive times, managed memory is better than UVM for data sets that are bigger than a cache-line. This implies that managed memory is better for larger data sets and for applications which execute atleast 4 GPU kernels on a given block of memory before the updated memory locations are accessed back on the CPU.

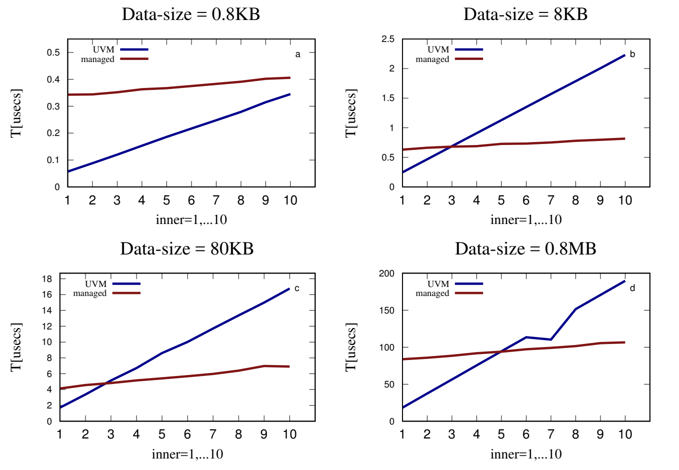

Figure 3: Varying the number of consecutive kernel calls (*inner*), using 2 CPU memory touches. (*outer*).

2) Varying the number of consecutive GPU kernel calls: In this section we explore the effects of increasing the number of consecutive GPU kernel calls by fixing the *outer* loop to 2 and varying the size of the inner loop. The results are given in Figure 3. The relative performance of managed memory improves as the number of consecutive kernel calls is increased. As shown in Figure 3a, when the data accessed by each thread-block is less than the cache line size, UVM is better than managed memory for a least the first 10 consecutive GPU kernel calls. But the figure also shows that managed memory will overtake UVM with respect to performance after approximately 12-13 consecutive GPU kernel calls. As the data size increases, the GPU kernel needs to be ran far fewer times for managed memory to overtake UVM. This is because managed memory works on the size of a memory page while UVM works on cache lines, resulting in far more data misses

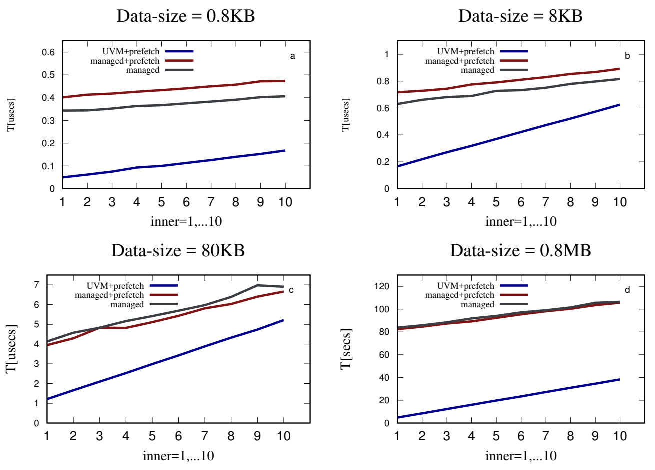

Figure 4: Prefetching effect on GPU Kernel performance with 2 outer calls.

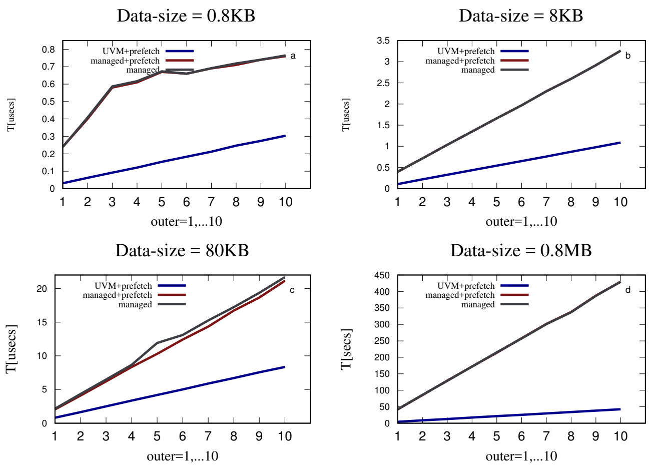

Figure 5: Prefetching with a varying number of CPU memory touches (*outer*), using 2 consecutive kernel calls (*inner*).

in UVM. A rough estimate of the number of data misses for each of the memory allocation scenarios is given by Equations 2 and 3.

$$UVM\_misses \approx data\_size/cacheline\_size \tag{2}$$

$$managed\_misses \approx data\_size/page\_size \tag{3}$$

Since a cache line size on a Volta is 128 bytes and the page size is 4KB, UVM is expected to have approximately 32 times more misses than managed memory, leading to poor performance for UVM that worsens as the size of the data set increases.

B. UVM vs managed when provided with prefetch hints

This section explores the effect of adding the prefetching hints to both UVM and managed memory. Prefetching informs the run-time that the data is available for asynchronous prefetching and may allow for additional optimizations on the number and timing of data transfers. In the case of this DAXPY benchmark, the prefetching informs the CUDA run-time that we will require the entire *x* and *y* arrays on the device, so the run-time is expected to copy the entire data structures to the device when the first element is accessed.

1) Prefetch Effect on Managed Memory and UVM: Figure 4 compares the performance of the GPU kernels when both UVM and managed memory allocation techniques are provided with prefetch hints. For this test, *outer* is fixed at 2. The performance comparison in Figure 4 shows that if both memory management techniques are provided with prefetch hints and the memory is not toggled more than two times between CPU-GPU, UVM+prefetch has the best performance for GPU kernels. It is also interesting to note that prefetching has only a small effect on managed memory. Another observation is that while the initial cost is higher in case of managed and managed+prefetch, after that the slope remains constant for all three techniques.

2) Vary the number of host-to-device transfers with prefetching: Figure 5 shows the effects of prefetch on the performance of UVM and managed memory for a varying number of host-to-device transfers.

From figure 4 and figure 5, we can observe that UVM with prefetching hints is consistently better for GPU kernels of all data sizes and not just for cases with a minimum number of CPU accesses. Additionally, the relative benefits of UVM with prefetch hints increase with the amount of data accessed by each thread-block. While the prefetching hints should benefit both managed and UVM memory allocation strategies, additional work is underway to understand the reason for the superior performance benefits observed by UVM compared to managed memory.

C. Effect of prefetching for the TouchOnCPU *performance*

Additional insights into the effects of prefetching can be gained by exploring the impact on accessing the data provided with the prefetch hints for GPU on the CPU. In this section, we explore the performance of the *TouchOnCPU* function using UVM, UVM+prefetching and managed+prefetching. The overall time of the benchmark, i.e., the sum of the GPU and the *TouchOnCPU* kernel times, is also examined.

Figure 6 shows the time taken by *TouchOnCPU* with 2 device-to-host data transfers and 10 consecutive GPU kernels. The *TouchOnCPU* performance does not change if the number of consecutive GPU kernels is varied, so the number of GPU calls was chosen to show useful results. The size of memory accessed inside each of the thread-blocks varies along the X-axis and the time taken by *TouchOnCPU* is shown on the Y-axis in microseconds.

Figure 6 shows us that there is a drastic increase in the time taken by the *TouchOnCPU* routine when we use prefetch hints with UVM and the difference increases with increasing data sizes. This effect becomes consistently worse as the number of *TouchonCPU* calls increases.

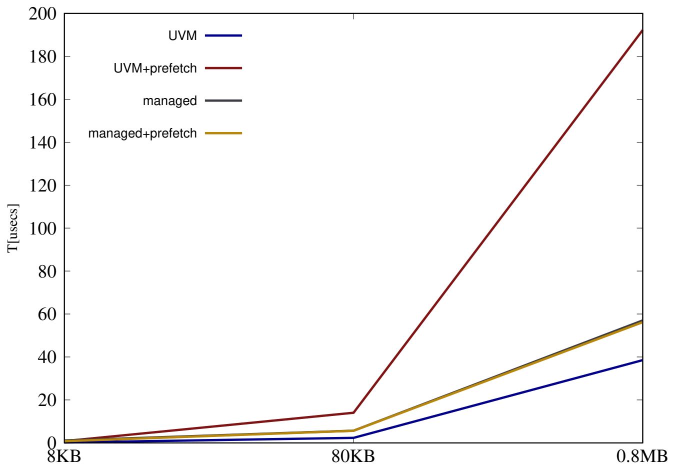

Figure 6: *TouchOnCPU* timing with 2 CPU data transfers *outer=2* and two consecutive GPU kernels (*inner=10*).

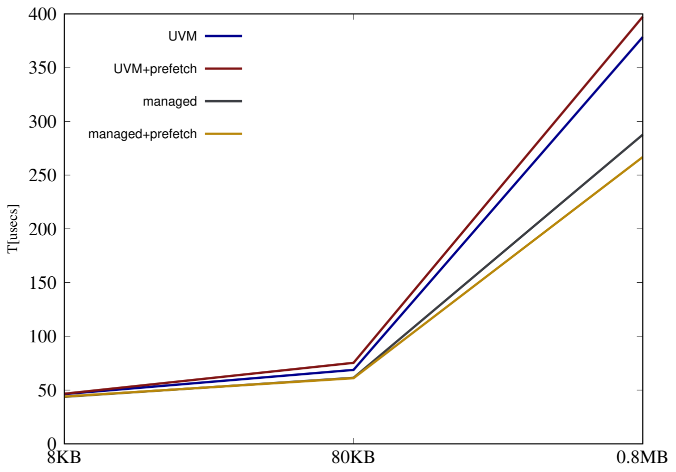

Figure 7: Total timing with 2 device-to-host data transfers (*outer=2*) and two consecutive GPU kernels (*inner=10*).

On the other hand, prefetching has very little effect on CPU performance when using managed memory. Comparing Figures 6 and 4 shows that managed memory has gained benefits in both CPU and GPU kernel run-times when provided with the prefetching hints, suggesting managed memory may be more performant on the code as a whole as the number of data transfers increases. This can be explored by plotting the total benchmark time, including both GPU and CPU data transfers.

Figure 7 shows the total timings of the benchmark when we compare UVM and managed memory, with and without the prefetching hints. The number of consecutive GPU kernel calls are set to 10 and the number of device-to-host data transfers set to two to be consistent with the previous plot.

Figure 7 shows that managed memory with prefetching yields the best performance whereas UVM with the same hints performs worst. Unfortunately, the improved performance of UVM+prefetch on the GPU kernel is overshadowed by the decreased performance in *TouchOnCPU*. Work is ongoing to determine the cause of this performance loss, but the current assumption is that when UVM is provided with prefetch hints, the data is left on the device and only moved back at cache line granularity when accessed in the *TouchOnCPU* function. Essentially, we suspect that when provided with prefetch hints, UVM performs exactly the opposite of what it does without the hint. The number of device-to-host transfers is also being explored to understand how the data transfers happen in each of the cases.

IV. Summary of Results

The performance of GPU kernels using UVM and managed memory with and without prefetching was assessed.

- UVM shows better performance when the data accessed on the GPU is closer to cache line size rather than page size.
- UVM without prefetching shows better performance when data is accessed on the CPU frequently and GPU reuse is minimial.
- Providing prefetching hints greatly improves the performance of UVM on GPU but show substantial penalties on the CPU.
- Prefetching hints are beneficial for managed memory on both CPU and GPU performance.

V. Conclusion

In this paper we devised a benchmark to study the performance of UVM provided through ATS versus managed memory allocated via *cudaMallocManaged* on an NVIDIA GPU DAXPY kernel. The benchmark was designed to test the performance of the GPU kernels when memory is transferred between host and device on the granularity of page size versus cache line size. It was observed that although managed memory allocation performed better in most cases, UVM is beneficial for data transfers on the order of a cache line or when data is returned to the CPU often. The benefits of prefetching were also explored. Managed memory shows consistent improvment when using prefetching. UVM shows substantial GPU improvement from prefetching, but CPU performance is substantially reduced. Overall, managed memory tends to be a better memory allocation strategy when the data transfers back to the CPU are also taken into consideration.

Acknowledgment

This research used resources of the National Energy Research Scientific Computing Center (NERSC), which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231. This research also used resources of the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725

References

[1] Sudharshan S. Vazhkudai, Bronis R. de Supinski, Arthur S. Bland, Al Geist, James Sexton, Jim Kahle, Christopher J. Zimmer, Scott Atchley, Sarp H. Oral, Don E. Maxwell, Veronica G. Vergara Larrea, Adam Bertsch, Robin Goldstone, Wayne Joubert, Chris Chambreau, David Appelhans, Robert Blackmore, Ben Casses, George Chochia, Gene Davison, Matthew A. Ezell, Tom Gooding, Elsa Gonsiorowski, Leopold Grinberg, Bill Hanson, Bill Hartner, Ian Karlin, Matthew L. Leininger, Dustin Leverman, Chris Marroquin, Adam Moody, Martin Ohmacht, Ramesh Pankajakshan, Fernando Pizzano, James H. Rogers, Bryan Rosenburg, Drew Schmidt, Mallikarjun Shankar, Feiyi Wang, Py Watson, Bob Walkup, Lance D. Weems, Junqi Yin, ”The Design, Deployment, and Evaluation of the CORAL Pre-Exascale Systems,” Proceedings of Supercomputing 2018 (SC18): 31th Int’l Conference on High Performance Computing, Networking, Storage and Analysis, Dallas, TX, November 2018.

[2] David Kirk. 2007. NVIDIA cuda software and gpu parallel computing architecture. In Proceedings of the 6th international symposium on Memory management (ISMM ’07). ACM, New York, NY, USA, 103-104. DOI: https://doi.org/10.1145/1296907.1296909

[3] ATS-GPU-BASE Real Time Signal Processing Software https://www.alazartech.com/Products/ATS-GPU-BASE_v4_0.pdf

[4] CUDA 9.2 UPDATE, OLCF User Group Call, 4/25/2018 https://www.olcf.ornl.gov/wp-content/uploads/2018/03/olcf_cuda9_2_update.pdf

[5] POWER9 Servers Overview https://www.ibm.com/downloads/cas/KDQRVQRR

[6] NVIDIA Tesla V100 GPU Architecture https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf

[7] Nikolay Sakharnykh, ”Everything you need to know about Unified Memory ” http://on-demand.gputechconf.com/gtc/2018/presentation/s8430-everything-you-need-to-know-about-unified-memory.pdf March 2018.

[8] Summit User Guide https://www.olcf.ornl.gov/for-users/system-user-guides/summit/summit-user-guide/#nvidia-v100-gpus