Comparing Managed Memory and ATS 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, including *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* via the 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: ATS and managed memory.**

***Index Terms*—GPU, CUDA, managed memory, ATS, Unified Virtual Memory.**

I. Introduction

Unlike multicore architectures, accelerator architectures 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 data on both memory spaces without the need for an explicit memory transfer. Data transfer is done by the Memory Management Unit (MMU) 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 introduced support for Address Translation Services (ATS) [3] [4] for power platforms. This technology allows GPUs to access CPU page tables directly and is supported on Volta GPUs through NVLink connections. A miss in the 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] [7]. 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 ATS.

One important difference between managed memory and ATS 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. ATS copies the data between host and device on the granularity of a cache line in a cache coherent manner (the CPU can cache GPU memory), immediately updating the CPU memory with the new values from the GPU.

ATS can adjust the granularity of the data movement with the use of optimization 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 optimization, which allows preemptive data transfers without the need to page fault.

The goal of the paper is to evaluate the differences in performances of CPU and GPU kernels between ATS and managed memory implemented through the CUDA API. In order to evaluate these differences, a benchmark has been designed that allows accesses on the CPUs and GPUs to be parameterized. This benchmark is used to evaluate the performance of both memory management systems with and

NERSC, LBL, DOE

978-1-7281-5977-5/19/$31.00 ©2019 IEEE
DOI 10.1109/PMBS49563.2019.00010

without prefetching.

This study is organized as follows: Section II will describe the DAXPY kernel benchmark developed to study ATS and managed memory. In Section III, the performance results of ATS and managed memory are presented. The results also present the performance differences between managed and ATS, when prefetching has been utilized. Finally, Section IV summarizes the observed behavior and describes when one memory management technique should be preferred over the other.

II. DAXPY KERNEL

For this benchmark we use a DAXPY kernel which is represented by the equation below:

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

In equation 1, y and x are two-dimensional vectors of type *double* and their dimensions are N and M respectively. This can be solved sequentially using the DAXPY kernel shown in Listing 1:

Listing 1: Sequential DAXPY kernel

```
//Standard CPU implementation of DAXPY.
void daxpy_kernel (double *x, double *y)
{
 int i = 0, j = 0;
 for(i=0; i<N; i++)
  for(j=0; j<M; j++)
   y(i,j) += a*x(i,j);
}
```

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.

Given each iteration is independent, this algorithm is well-suited for GPUs. A variety of parallel techniques could be applied to this algorithm, but a simple and direct threading technique was chosen for this study, as shown in Listing 2:

Listing 2: GPU DAXPY kernel

```
//Threaded GPU implementation of DAXPY.
__global__ void 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);
}
```

The kernel is built to distribute the N dimension across the thread-blocks and the M across threads with each thread-block.

Listing 3 shows the memory setup for the benchmark:

Listing 3: Setup the memory

```
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));
//Otherwise, allocate x and y with malloc to test ATS
#elif defined(ATS)
    x = malloc(N*M*sizeof(double))
    y = malloc(N*M*sizeof(double))
#endif
...   // Continued in Listing 4
```

The x and y vectors are stored as $N*M$ sized *double*-type arrays. They are allocated by calling *cudaMallocManaged* when testing managed memory or *malloc* when testing ATS.

Then, the data is initialized and prepared as shown in Listing 4:

Listing 4: Preparing DAXPY

```
...   // Continued from Listing 3
//One threadblock per outer loop (N).
//Each threadblock is launched with 1 warp (32 threads).
daxpy_kernel<<<N, 32>>>(x,y);
daxpy_kernel<<<N, 32>>>(x,y);

TouchOnCPU(y);
...   // Continued in Listing 5
```

Before the benchmark begins, the *DAXPY* kernel is launched a few times to eliminate any first-time effects and the y array is returned to the CPU for the beginning of the benchmark. Note that the value of the x array is unchanged, as it is not accessed on the CPU, which should be consistent with most real implementations.

Next, the benchmark is performed as shown in Listing 5:

Listing 5: Launching DAXPY

```
...   // Continued from Listing 4
//outer - times the data is brought back to the CPU
//inner - times the given GPU kernel is consecutively
     launched
  for(outer){
    for(inner){
     daxpy_kernel<<<N, 32>>>(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 5 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 *malloc*.

The data is returned to the CPU through the implementation of Listing 6:

Listing 6: Returning data to the CPU

```
//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 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. The cache line size is 128 bytes [8], the CPU page size is 4KB and the GPU page size is up to 2MB.

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. In III-C, we discuss the effects of the various memory management strategies on the performance of *TouchOnCPU* and on the total time taken for the experiment.

A. ATS vs Managed

First, ATS 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 and the Y-axis represents the time taken by the *DAXPY* GPU kernel in micorseconds. 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 and varying the size of the *outer* loop with different data sizes. The results for *inner=4* and *inner=7* are shown in Figures 1 and 2.

Figure 1 shows that, for this DAXPY benchmark, if there are 4 consecutive GPU kernel calls ATS is always better than managed memory. This remains true if the number of GPU calls is less than 4. Additionally, ATS has a smaller slope, which implies that as the number of CPU touches increases ATS would be more beneficial. From Figure 2, we see that as the data size increases managed memory overtakes ATS. At larger data sizes, managed memory is always better than ATS.

These results imply that managed memory should be chosen whenever data is substantially larger than a cache line and can be reused on the GPU regularly before being returned to the CPU. While there is some effect from the frequency of CPU accesses, the number of consecutive GPU accesses is more important to memory technique selection.

2) Varying the number of consecutive GPU kernel calls: Next, the effects of increasing the number of consecutive GPU kernel calls is explored by fixing the *outer* loop to 4.

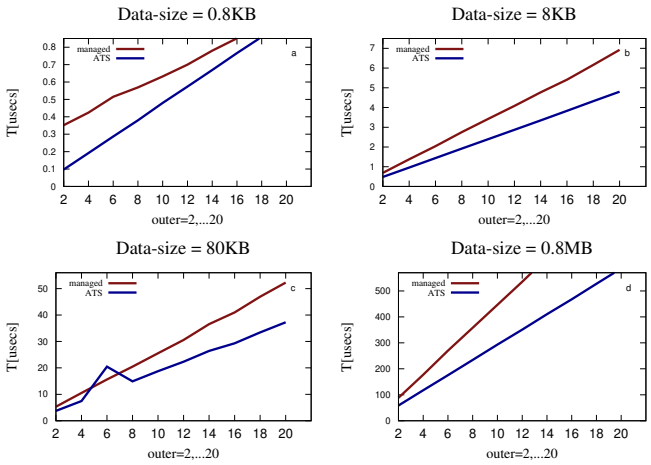

Figure 1: Varying the number of CPU memory touches (*outer*), using 4 consecutive kernel calls. (*inner*).

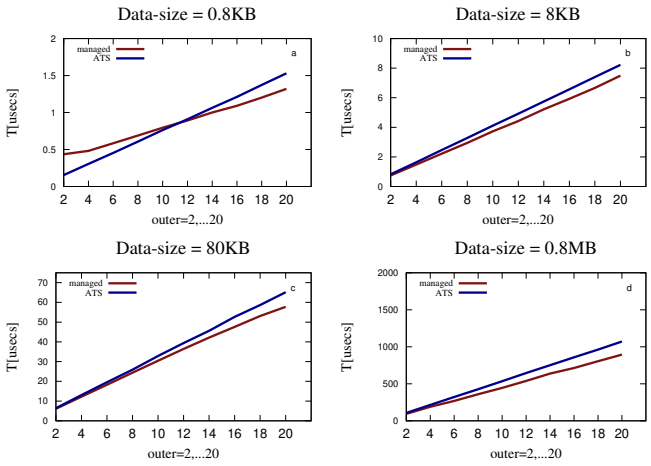

Figure 2: Varying the number of CPU memory touches (*outer*), using 7 consecutive kernel calls. (*inner*).

The results are given in Figure 3. As expected, the relative performance of managed memory improves as the number of consecutive GPU kernel calls is increased. The effect of granularity can be observed in Figure 3. When the data size is substantially larger than a cache line, the *DAXPY* GPU kernel has to be run substantially fewer times in a row for managed memory to outperform ATS. For Figures 3b, 3c and 3d, ATS overtakes managed memory after only 6-to-8 consecutive GPU kernel calls. This is due to the difference in the granularity at which the data is moved between the host and device among the two memory management techniques.

The major benefit of managed memory in cases with large data-set sizes and many kernel invocations is that after paying the expense of migrating the pages over to GPU memory for the first kernel execution, subsequent kernels calls can reuse the data out of high-bandwidth GPU memory, whereas, with ATS, the data still resides on the CPU and must be accessed across the CPU-GPU bus in each kernel invocation. This ATS

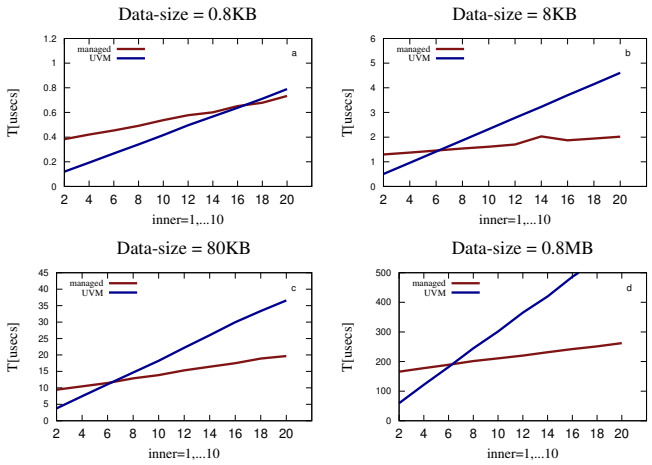

Figure 3: Varying the number of consecutive kernel calls (*inner*), using 4 CPU memory touches. (*outer*).

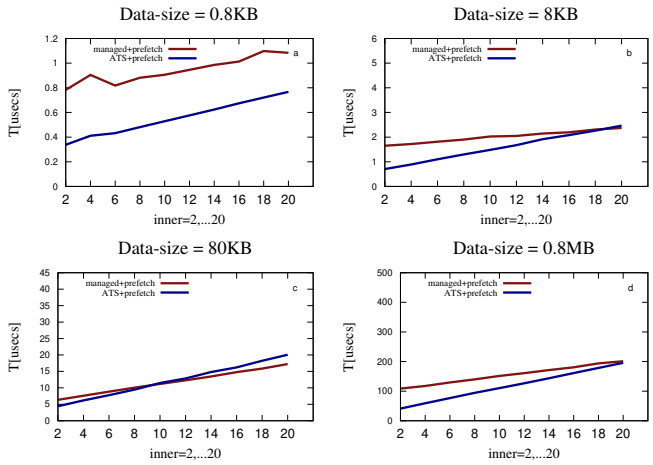

Figure 4: Prefetching effect on varying number of consecutive kernel calls *(inner)* using 4 CPU memory touches *(outer)*.

behavior can be changed by implementing prefetching.

B. ATS vs managed when provided with prefetch optimization

This section explores the effect of adding the prefetching optimization to both ATS 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 either on the host or the device, so the run-time is expected to copy the entire data structure to the corresponding memory space when the CUDA runtime has available resources. The implementation of the bnechmark with prefetching is shown in Listing 7:

Listing 7: Prefetching during DAXPY

```
// Benchmark with prefetching
  for(outer){
    //Prefetch y to GPU before the kernel launches.
    cudaMemPrefetchAsync (y, N*M*sizeof(double),
         gpuDeviceId, 0 );

    for(inner){
     daxpy_kernel<<<N, 32>>>(x,y);
    }

    //Prefetch y back to CPU before accessing it.
    cudaMemPrefetchAsync (d_Y, N*M*sizeof(double),
         cudaCpuDeviceId, 0 );

    TouchOnCPU(y);
  }
```

1) Vary the number of consecutive GPU kernels with prefetching: Figure 4 compares the performance of the GPU kernels when both ATS and managed memory allocation techniques are provided with prefetch optimization. For this test, the number of CPU accesses, *outer*, is fixed at 4 and the y-axis is kept identical to Figure 3 to allow direct comparisons.

Comparing Figures 4 and 3 shows that both ATS and managed memory improve performance when using prefetching for large data sizes. Prefetching substantially increases the number of consecutive GPU kernels required for managed to overtake ATS, making ATS more useful when proper prefetching optimizations can be applied.

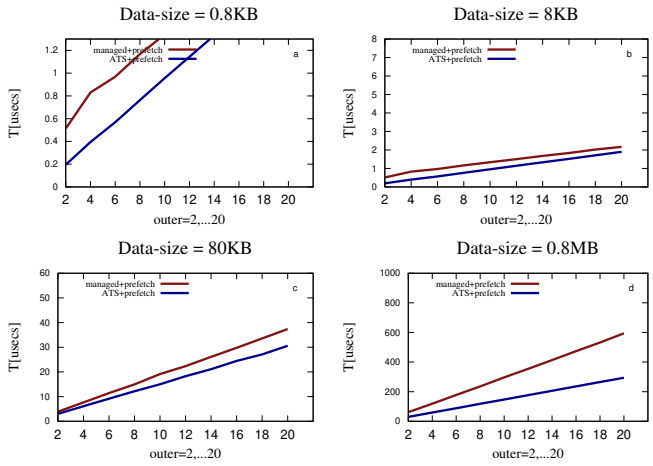

Figure 5: Prefetching effect on varying number of CPU memory touches (*outer*), using 4 consecutive kernel calls (*inner*).

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

From Figure 1 and Figure 5, it is observed that ATS with prefetching optimization is consistently better for GPU kernels of large data sizes (with 4 inner GPU kernel invocations). Additionally, the relative benefits of ATS with prefetch optimization increase with the amount of data accessed by each thread-block. While the prefetching optimization benefits both managed and ATS memory allocation strategies, ATS has a higher benefit. The prefetch optimization cause ATS data to be migrated to the GPU, which is the more optimal behavior in this implementation and results in bigger performance improvements.

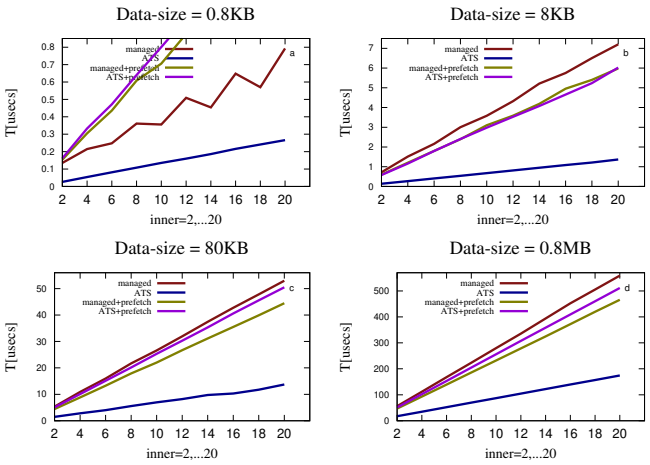

Figure 6: *TouchOnCPU* timing with 4 consecutive GPU kernel calls *inner=4*).

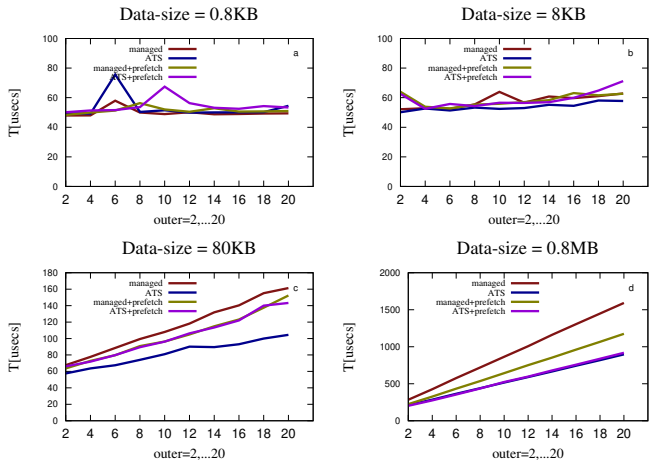

Figure 7: Total timing with 4 consecutive GPU kernel calls *inner=4*).

In this section it has been shown that both ATS and managed memory gain benefits in the performance of the GPU *DAXPY* kernel when provided with prefetching optimization. When prefetching is possible, ATS should be chosen for cases that do not substantially reuse the data on the GPU, while managed is still preferred whenever data will be substantially reused on the device.

C. Effect of prefetching for CPU accesses and overall runtime

This section explores the performance of other features of the benchmark, including the *TouchOnCPU* function and the total algorithm time, with all 4 memory management techniques: ATS, managed, ATS+prefetching and managed+prefetching. The *TouchOnCPU* performance does not change if we vary the number of consecutive GPU kernels *inner*, so *inner* is fixed and the number of CPU accesses is changed. Figure 6 shows the time taken by *TouchOnCPU* with 4 consecutive GPU kernels. Figure 6 shows that there is a drastic increase in the time taken by *TouchOnCPU* when the memory handled via ATS is provided with the prefetch optimization. This is because when ATS is prefetched, the data is cached on the GPU and must be returned during *TouchOnCPU*, whereas ATS without prefetching leaves the data on the CPU.

On the other hand, prefetching improves the managed memory performance in all cases. When using data sizes on the order of pages or larger, managed+prefetch is the fastest method to move the data to the CPU in almost all configurations.

Figure 7 shows the total performance of the experiment when we fix the number of consecutive GPU kernels to 4 and vary the number of times we toggle the data between host-to-device. The timings shown in 7 include the time taken by the initial memory allocation of x and y shown in listing 3, initialization of x, warm up *DAXPY* and *TouchOnCPU* kernel executions shown in 4 and the experiment shown in listing 5

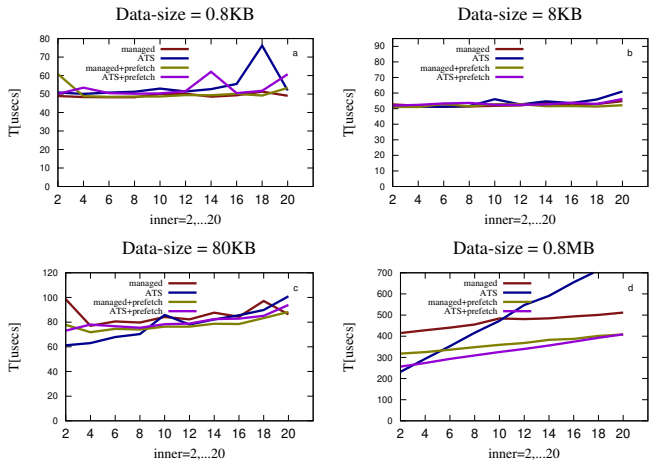

Figure 8: Total timing with 4 CPU memory accesses (*outer*=4).

From Figure 7, we can observe that while for the smaller problem size all 4 memory management techniques give similar performance. As the data size increases ATS and ATS+prefetch perform better compared to managed and managed+prefetch. When taking the entire time into account, ATS yields better performance without prefetching and managed memory is faster with prefetching. These results suggest for a fixed number of CPU accesses, ATS can be the superior choice, regardless of other kernel parameters.

Figure 8 shows the total time taken by the experiment when we fix the number of times data is transferred between CPU-to-GPU, i.e., *outer* while varying the number of consecutive GPU kernels, i.e., *inner*. Again, for smaller data sizes the memory management techniques show similar results, regardless of prefetching. However, for larger data sizes, the patterns seen in the kernel timings begin to emerge: ATS is dominant when minimal GPU data reuse occurs, ATS+prefetch is the superior for a small number of reuses and managed+prefetch eventually becomes dominant when GPU reuse is substantial.

The managed strategies show the smallest increases in time as the number of kernel calls increase, which is consistent with previous results.

IV. Summary of Results

The performance of GPU kernels when using memory allocated via *malloc* and handled via ATS versus managed memory allocated via *cudaMallocManaged* with and without prefetching was assessed.

- ATS with and without prefetching shows better performance when data is accessed on the CPU frequently and GPU reuse is minimial.
- Providing optimizations such as prefetch greatly improves the performance of both memory handled via ATS and managed memory, but slows down CPU accesses for ATS on the CPU.
- Managed memory with prefetching is superior when a large amount of GPU data reuse can be utilized.
- Without prefetching, the optimal method depends on data movement patterns and should be carefully explored for each application.

V. Conclusion

In this paper we devised a benchmark to study the performance of two different UVM implementations on NVIDIA GPUs: 1) Memory allocated via *malloc* and *new* is handled via ATS versus 2) 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 in these two models. It was observed that although managed memory allocation performed better in most cases when there is significant GPU data reuse, ATS is beneficial for when data is returned to the CPU often. The benefits of prefetching were also explored. Managed memory shows consistent improvement when using prefetching. ATS shows substantial GPU improvement from prefetching, but CPU performance is substantially reduced which is important for cases which require consistent access on the CPU.

In general, for cases with significant data reuse on the GPU, ATS can benefit significantly from prefetch optimization. However, one may point out that this is similar in spirit to manually managing memory via explicit CUDA API calls - diminishing the usability and automatic management value of ATS.

Overall, managed memory tends to be a better memory allocation strategy when data-set sizes on the order of MB or higher and data reuse on the GPU is frequent.

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