ANL-19/41

Evaluation of PETSc on a Heterogeneous Architecture the OLCF Summit System Part I: Vector Node Performance

Mathematics and Computer Science Division

ANL-19/41

Evaluation of PETSc on a Heterogeneous Architecture the OLCF Summit System Part I: Vector Node Performance

Prepared by
Hannah Morgan, Richard Tran Mills, and Barry Smith
Mathematics and Computer Science Division, Argonne National Laboratory

April 2020
This work was supported by the Office of Advanced Scientific Computing Research, Office of Science, U.S. Department of Energy, under Contract DE-AC02-06CH11357.

Evaluation of PETSc on a Heterogeneous Architecture the OLCF Summit System Part I: Vector Node Performance

Hannah Morgan, Richard Tran Mills, Barry Smith
Mathematics and Computer Science Division
Argonne National Laboratory

Abstract

Our goal is to report on the basic performance of PETSc vector operations on a single node of the Oak Ridge Leadership Computing Facility system Summit. We describe the Summit system and present data collected from several vector operations. Limited analysis is also presented.

1 Introduction

We report on the performance of the Portable, Extensible Toolkit for Scientific Computation (PETSc) [2, 3] vector operations on a single node of the IBM/NVIDIA Summit computing system [1] at the Oak Ridge Leadership Computing Facility (OLCF). PETSc provides NVIDIA GPU support for vector and sparse matrices based on CUDA and the cuBLAS and cuSPARSE libraries [9]. Using the organization of the PETSc library, many PETSc solvers and preconditioners are able to run with GPU vector and matrix implementations. This report summarizes the basic design of a Summit node and the performance of PETSc vector operations on that node and provides a limited analysis of the results. The planned United States Department of Energy exascale computing systems [10] have designs similar to that of Summit. Thus, it is important to have a well-developed understanding of Summit in preparation for these systems. This document is not intended to provide a strict benchmarking of the Summit system; rather it is to develop an understanding of systems similar to Summit, in order to guide PETSc development.

2 The Summit System and Experimental Setup

Each node on Summit is equipped with six NVIDIA Volta V100 GPU accelerators and two IBM POWER9 processors, each with 21 cores available to users, for a total of 42 cores (Figure 1, left). A GPU on Summit can be utilized by more than one MPI rank simultaneously via the NVIDIA Multi-Process Service; we refer to this as *virtualization*. For example, a single physical GPU may be treated as four virtual GPUs by four MPI ranks running on four CPU cores. Virtualization on Summit is enabled by submitting jobs with the bsub option `-alloc_flags gpumps`.

To obtain high performance on a Summit node for benchmarking (that is, by varying the number of CPU cores and GPUs used to understand their performance) one must select the optimal physical cores and associate them with appropriate GPUs. The two POWER9 processors are each connected directly to three of the GPUs, so cores associated with each process should, for best performance, utilize one of the three GPUs connected to the same socket. The `jsrun` job launcher on Summit supports the concept of *resource sets* for controlling how the resources on a node are partitioned between tasks in a job. When utilizing a single resource set consisting of only the CPUs (for example, in order to compare GPU and CPU performance), the Summit CPU cores are assigned from the first socket until it is full before using cores from the second socket; benchmarking numbers that utilize these defaults are not appropriate. To utilize the CPUs effectively one should use two resource sets, each with 21 associated cores, in order to take advantage of the combined memory bandwidth of both sockets of the Summit node (red and yellow in Figure 1, right). We accomplish this with the `jsrun` options `--nrs 2 --cpu_per_rs 21`. Furthermore, since adjacent Summit CPU cores

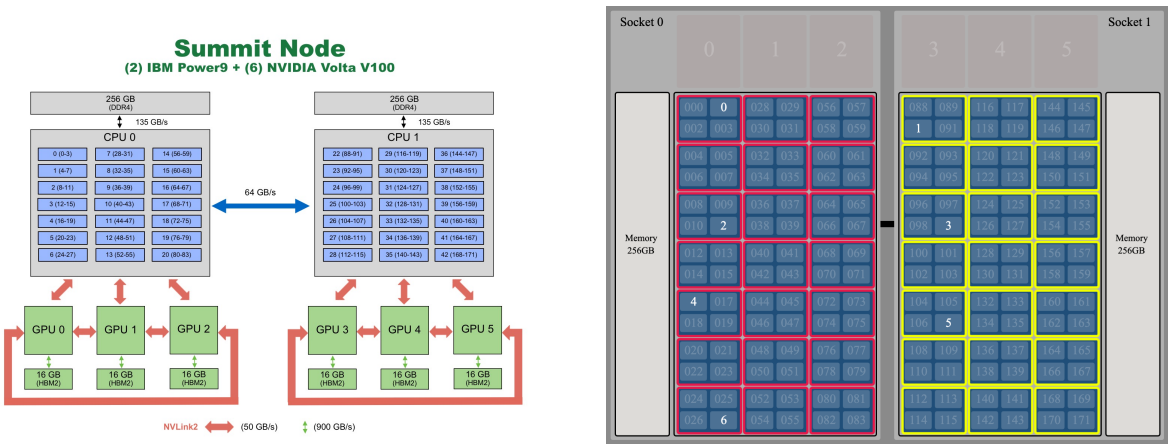


Figure 1: Diagram of one node of Summit (left) and CPU allocation of 7 MPI ranks (right) from https://www.olcf.ornl.gov and https://jsrunvisualizer.olcf.ornl.gov/.

share L2 and L3 cache we use the option `--bind packed:2` to bind each MPI rank to two CPU cores so that each MPI rank has dedicated L2 and L3 cache. We also launch all jobs using the `--launch_distribution cyclic` option so that MPI ranks are assigned to resource sets in a circular fashion. These choices are shown with seven MPI ranks in Figure 1, right.

The Summit CPU operating system uses traditional Unix memory management with memory pages. In order to obtain high performance in copies between the CPU and the GPU, the CPU memory must be "pinned" (i.e., page-locked). This is done by allocating the memory with a special routine `cudaMallocHost` instead of the system `malloc`. Our benchmarks are computed with pinned CPU memory.

Collecting useful and appropriate timings on a GPU-based system is not always straightforward. One can collect timings on the NVIDIA GPUs using `cudaEventRecord` or on the CPU cores using traditional Unix timers. Which timers to use depends on the goals of the analysis. In addition, care must be taken in timing asynchronous calls to the GPU. These can be subtle; for example, small (fewer than 64 kilobytes) cudaMemcpy() calls to the GPU are asynchronous, so it is easy to get misleading timings from the CPU on these transfers without care. In this report, since it treats PETSc in the traditional host-device model where the outer code is MPI parallel and we are interested in the higher-level details of the performance, we always use the MPI rank timers (using PETSc logging). This approach may result in overtiming some operations since it includes the time to synchronize back to the CPU, though that in actual optimized runs that synchronization may not take place.

For these results PETSc 3.12 was built with CUDA version 10.1.168 and PGI compilers version 19.4. The codes used for the measurements in this report are available in the PETSc repository. They are in src/vec/vec/examples/tutorials/performance.c. This file also indicates where one can obtain the scripts for producing the graphs. Peak performance numbers for Summit are from [7]. When comparing GPU to CPU performance we use all 42 available cores of the CPU. This may not be the optimal value for obtaining CPU performance since the memory bandwidth is shared between cores. Since our goal is not to benchmark Summit's CPU scalability, we have not collected statistics with a range of CPU cores.

3 PETSc Vector Operations

We present and analyze results from runs of PETSc vector operations on one node of Summit. Our benchmark code creates vectors with random entries, copies them to the appropriate memory, and then performs vector operations. We report the flop rate in Mflops/s for various vector sizes utilizing either GPUs or CPUs. We also report memory throughput (in 8 Mbytes/s for easy comparison with the floating pointing results). These computations are essentially the same as the STREAMS benchmark [8] but measured on the PETSc vector operations instead of raw C or CUDA code. We use the term "throughput" to mean the amount of data moved divided by the time to move it. This is a combination of latency and bandwidth, introduced in Section 5. In all cases the vectors are already in the memory of the device where the performance is being

measured. All asynchronous operations are synchronized with `CudaDeviceSynchronize()` so the true time of the operation as realized on the CPU is used. The caches are flushed by performing vector operations on other vectors to remove any effects of the data being in the cache. For parallel benchmarks, an MPI_Barrier() is used immediately before the operation so that all MPI ranks start together.

The specific vector operations are the following.

- $x = ax + y$, known as AXPY and implemented with PETSc's VecAXPY operation (which utilizes BLAS on the CPU and cuBLAS on the GPUs).
- $d = x^T y$, known as the dot product and implemented with PETSc's VecDot operation (this also utilizes BLAS and cuBLAS).
- Memory copy, implemented with VecCopy (which utilizes memcpy on the CPU and cudaMemcpy on the GPUs). In addition we use cudaMemcpy for copies between the CPU and GPU.
- Setting all entries in a vector to zero, implemented with PETSc's VecSet function (which utilizes memset on the CPU and cudaMemset on the GPUs).

A single vector operation is timed for each operation. All vector sizes refer to the global size of the vector, which may be spread among multiple computational units. There are two floating-point operations per entry for the VecAXPY and VecDot operations. There are three memory accesses per vector entry for VecAXPY, two for VecDot, two for VecCopy, and one for VecSet. Thus, computation of flop rates and memory throughput involves appropriate scaling by these values.

4 Experimental Results

Figure 2 presents a high-level view of the performance of the Summit nodes. Details are provided below. Note the use of the log scale and that, toward the right, the GPUs are performing significantly better than the 42 CPU cores, while towards the left the CPUs are faster. Figure 3 presents an alternative view of the same data, known as a static scaling or work-time spectrum plot [4, 5]. It has the advantage that both the asymptotic bandwidth and the latency of the operations can be directly read from the figure.

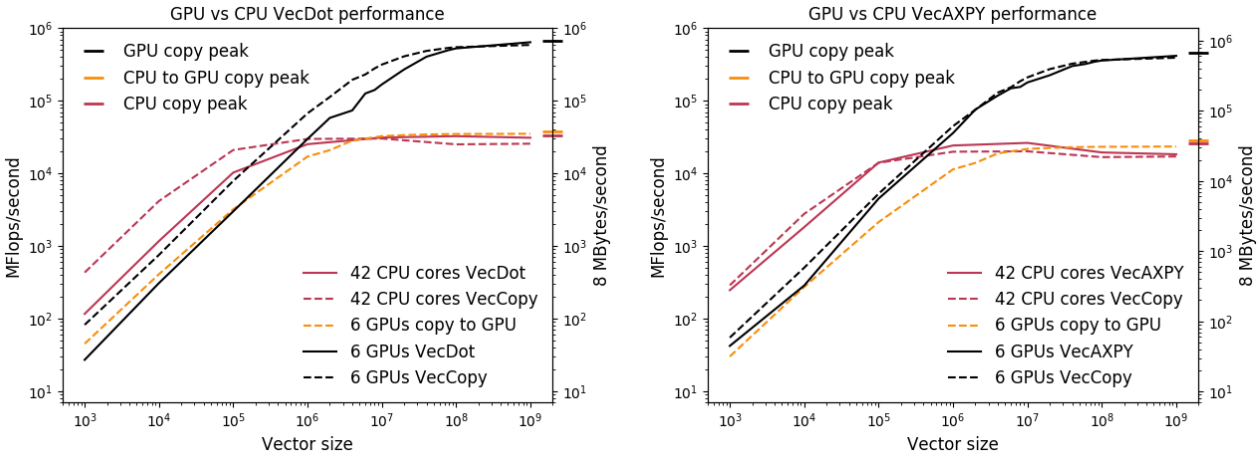


Figure 2: Effect of vector size on vector performance and memory throughput (one MPI rank per GPU). Note the log scale.

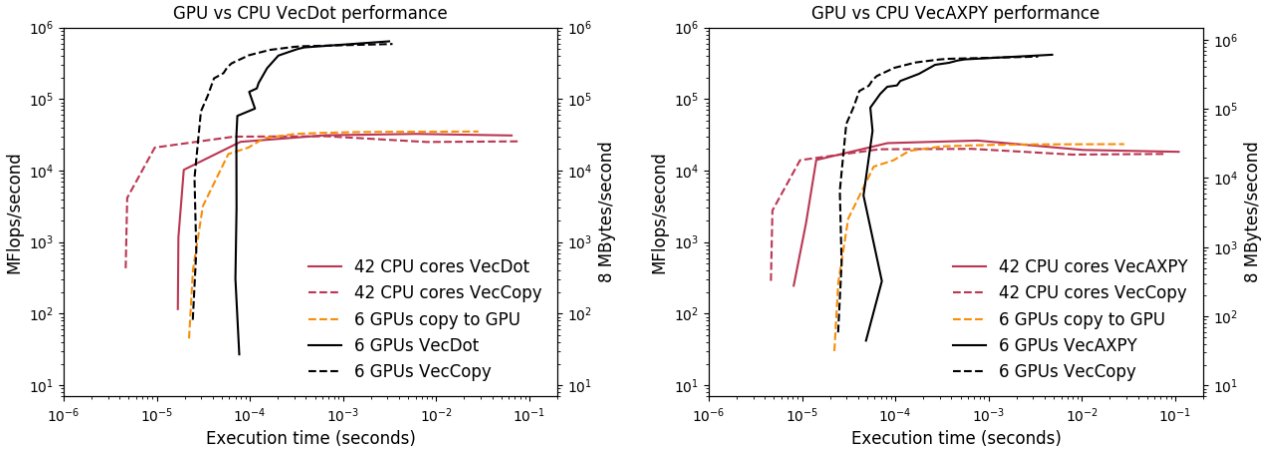


Figure 3: Effect of execution time on vector performance and memory throughput (one MPI rank per GPU). Note the log scale.

Figures 4 and 5 compare the performance of the GPUs and CPUs for increasing vector sizes. For shorter vectors, fewer than around 10^6 entries, the CPU performance for vector operations, including copies, is far superior to that of the GPU. For larger sizes, 10^9 vector entries, on the other hand, the GPU throughput is much higher. Performance on the CPU is relatively independent of vector size, but on the GPUs is low except for large vectors. The drop in performance of the CPU at 22 cores occurs because the remaining cores share L2 and L3 caches with previously utilized cores and share the available memory bandwidth.

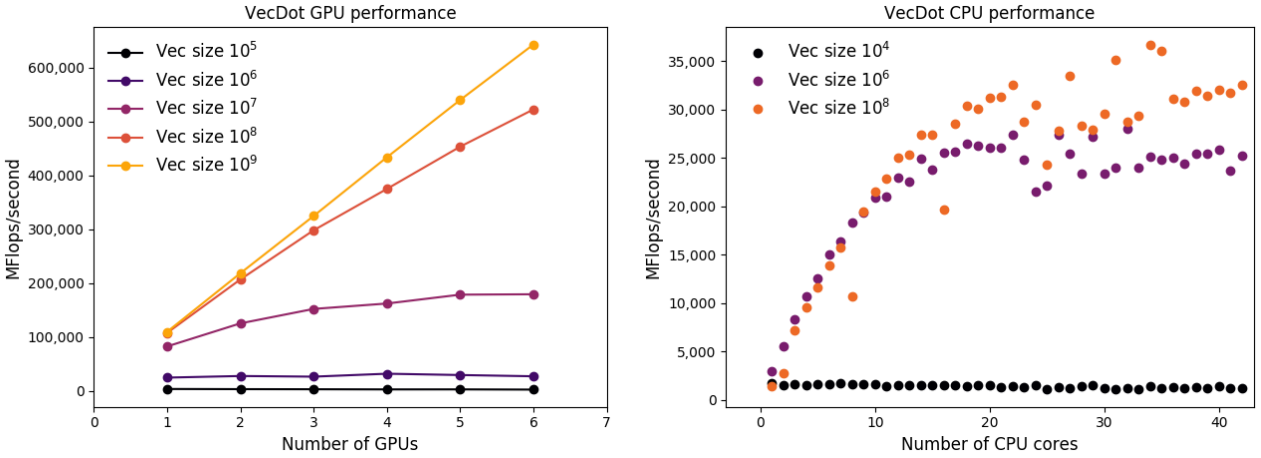


Figure 4: VecDot flop rate, with one MPI rank per GPU (left) and CPUs (right).

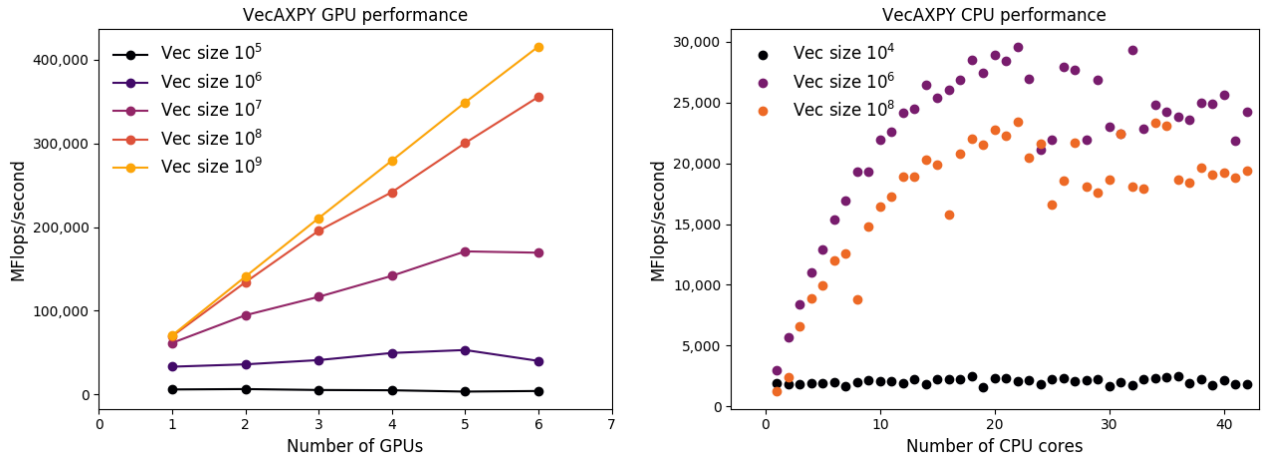


Figure 5: VecAXPY flop rate, with one MPI rank per GPU (left) and CPUs (right).

Figures 6, 7, and 8 explore the performance effects of GPU virtualization. Performance of the GPUs is similar even when divided into up to eight virtual GPUs for smaller vectors. For larger vectors the performance is up to about 20% worse. Note that in Figure 6 the vectors on six GPUs are 1/6 the size they are on one GPU. This means the throughput each is achieving is lower than may have been expected, since the six GPUs no longer have long enough vectors for full performance.

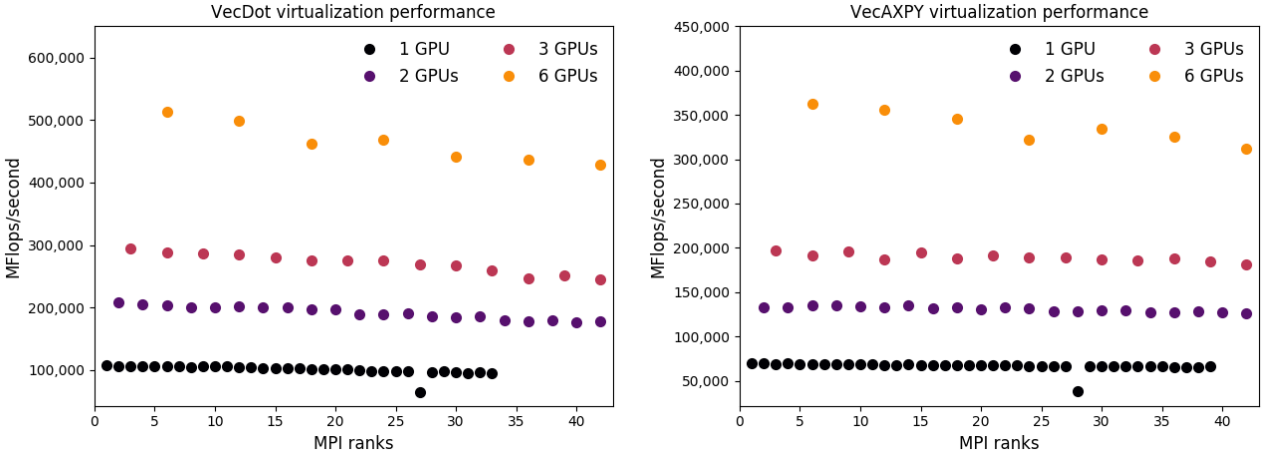
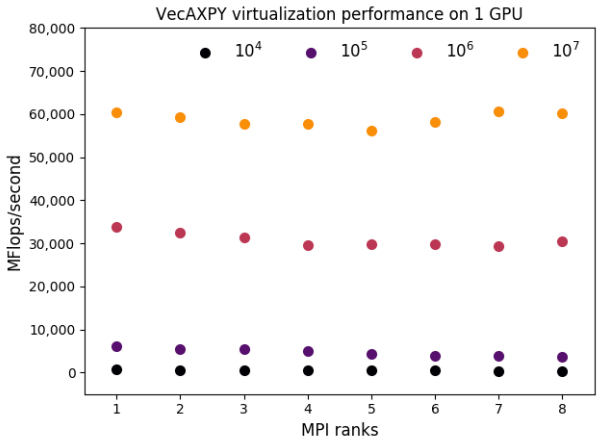


Figure 6: Performance of GPU virtualization for VecDot operations (left) and VecAXPY (right) for vectors of length 10^8.



MPI ranks	latency	bandwidth
1	31	99,000
2	33	99,000
3	34	96,000
4	38	96,000
5	40	96,000
6	43	99,000
7	45	105,000
8	46	105,000

Figure 7: VecAXPY virtualization performance for small vectors ($10^5 - 10^7$) on 1 GPU with latency (10^{-6} seconds) and bandwidth (8 Mbytes/second) as defined in Section 5.

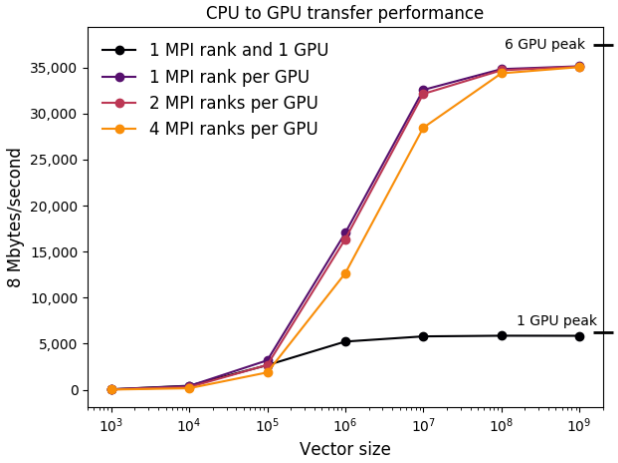


Figure 8: Effect of vector size on CPU to GPU copy throughput with 6 GPUs (multiple MPI ranks per GPU) and 1 GPU with 1 MPI rank. Pinned memory is always used.

Figure 8 shows the performance of CPU to GPU copies. For large vectors, memory transfers nearly reach the hardware peak.

Figures 9 and 10 show the performance of the vector operations as a function of vector size for different number of GPUs and CPU cores. Since the GPUs are independent entities, the performance scales essentially perfectly for more GPUs. The CPU cores are not independent, since they share a common memory, and hence performance improvement decreases as more cores are utilized.

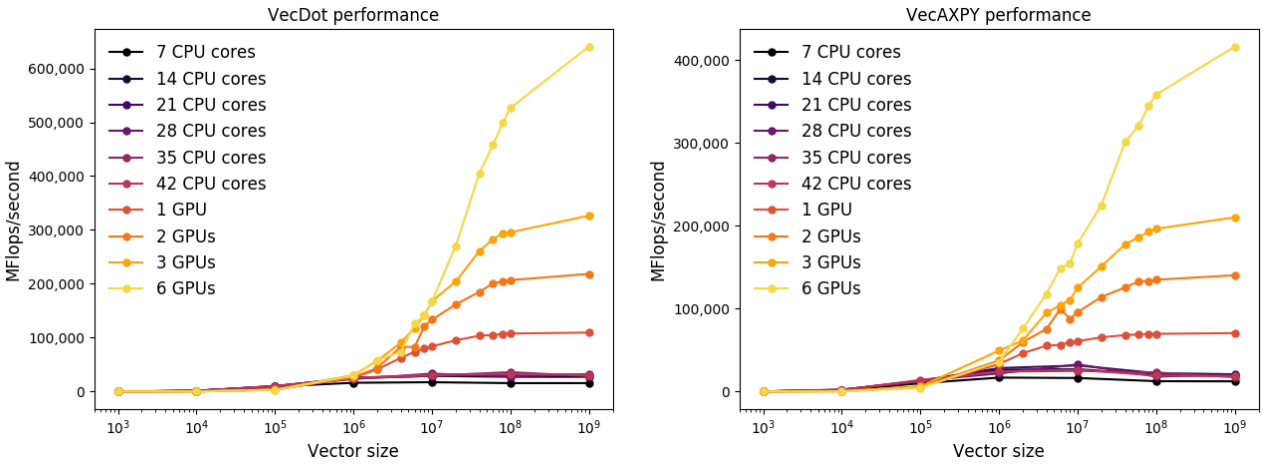


Figure 9: Effect of vector size on flop rate for VecDot (left) and VecAXPY (right) on CPU and GPUs with one MPI rank per GPU.

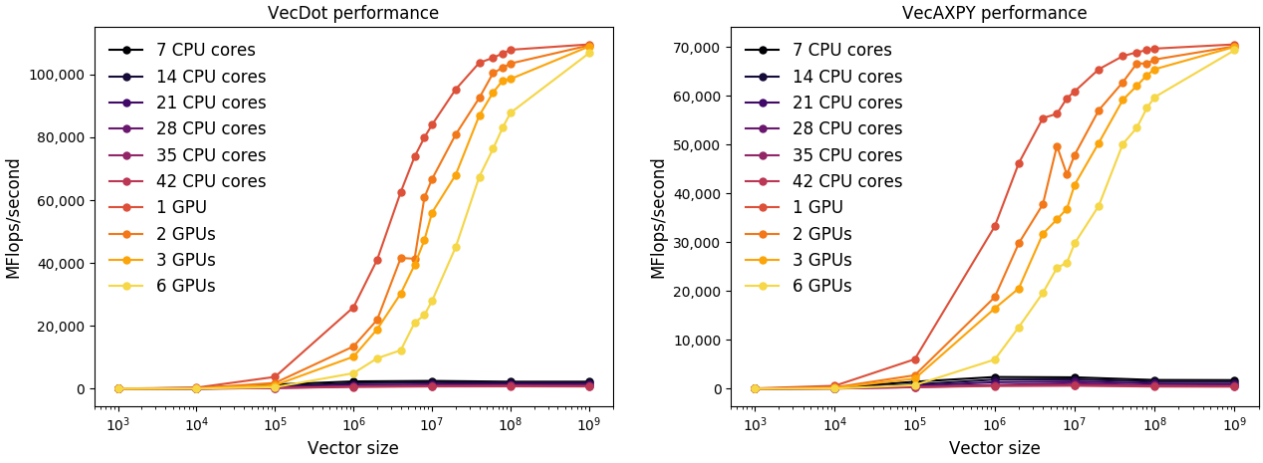


Figure 10: Effect of vector size on flop rate for VecDot (left) and VecAXPY (right), on CPU and GPUs with one MPI rank per GPU, scaled by the number of CPU cores or GPUs.

5 Discussion

The performance of the basic vector operations as a function of vector size n is complicated on modern CPU and GPU systems. The traditional linear model is given by

$$t = latency + \frac{n}{bandwidth} = l + \frac{n}{b}.$$

In Table 1 we show the bandwidth (computed via least squares) and latency (computed via least squares or obtained directly from the data for small vector sizes) pairs for each operation for 6 GPUs with 1 MPI rank per GPU. We found that operations on the GPUs follow this linear latency/bandwidth model for large vectors. The behavior for small vectors is depicted in Figure 11. For small vectors, the performance is determined completely by latency. Counterintuitively, working with slightly larger vectors takes the same amount of time as shorter vectors. This odd behavior is because the GPU has thousands of computational units, and until they are all occupied, no additional time is needed for additional computations that now occupy some of the previously unoccupied units. This behavior is not captured in the linear model. Table

2 shows the latencies and bandwidths for the CPU. Operations on the CPU do not closely match a linear model for any range of vector sizes; thus the latencies are not computed via least squares.

Table 3 contains the latency and bandwidth values for copies from the CPU to the GPU. We found that for non-trivial size vectors the performance is well modeled by the linear model.

Vec size	10^3-10^5		10^5-10^7		10^7-10^8	
Operation	latency	bandwidth	latency	bandwidth	latency	bandwidth
VecDot	93	-	87	567,000	89	667,000
VecAXPY	69	-	59	375,000	89	627,000
VecSet	24	-	25	609,000	26	667,000
VecCopy	29	-	31	559,000	32	593,000

Table 1: Latency (10^{-6} seconds) and bandwidth (8 Mbytes/second) for vector operations on 6 GPUs.

Vec size	10^3 - 10^5		10^5 - 10^7		10^7 - 10^8	
Operation	latency	bandwidth	latency	bandwidth	latency	bandwidth
VecDot	17	35,000	-	32,000	-	33,000
VecAXPY	9	48,000	-	40,000	-	28,000
VecSet	3	27,000	-	22,000	-	18,000
VecCopy	4	36,000	-	32,000	-	24,000

Table 2: Latency (10^{-6} seconds) and bandwidth (8 Mbytes/second) for vector operations on 42 CPU cores.

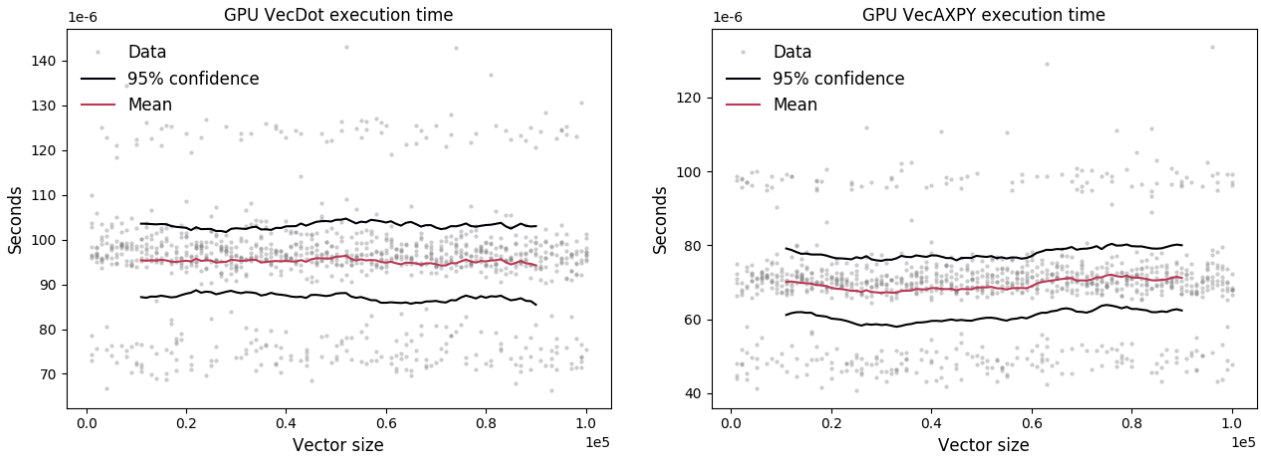


Figure 11: GPU execution time on small vectors.

Vec size	10^3-10^5		10^5-10^7		10^7-10^8	
Operation	latency	bandwidth	latency	bandwidth	latency	bandwidth
Copy to GPU	48	42,000	36	34,000	43	35,000

Table 3: Latency (10^{-6} seconds) and bandwidth (8 Mbytes/second) for copies from the CPU to 6 GPUs.

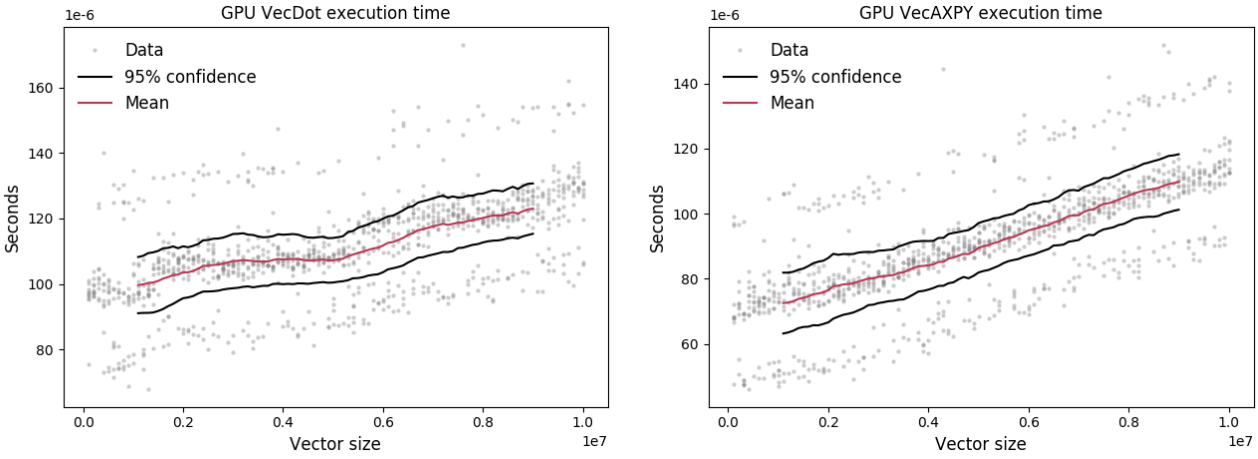


Figure 12: GPU execution time on medium sized vectors.

Even though the linear model does not capture the full behavior of the system, it is still useful in understanding and comparing different systems behavior. Given the linear model above, the throughput can be written as

$$T(n) = \frac{n}{l + \frac{n}{b}} = \frac{n \times b}{l \times b + n}.$$

The characteristic shape of this curve is reflected in the GPU values in Figure 2 as well as Figures 8, 9, and 10.

The throughput, as a function of time, as depicted in Figure 3, can be modeled with[1]

$$T(t) = \frac{b(t-l)}{t} = b(1 - \frac{l}{t}).$$

The bandwidth provides the "height" of the curve, and the latency is the "start" of the curve. One can understand the shape of the curve from a simple asymptotic analysis. For t at the latency

$$\frac{d\,T}{dt}|_{t=l} = \frac{b}{l},$$

hence the steep initial slope. For extremely large t, the throughput approaches the bandwidth. In the discussion below we will use the machine characteristics of Summit with the VecAXPY operation to provide examples for the models; $l_G = 89 \times 10^{-6}$, $b_G = 627 \times 10^9$, $l_C = 9 \times 10^{-6}$, and $b_C = 28 \times 10^9$. Since latencies for large vector sizes are not available on the CPU, we will use the value for the small vectors.

In the following we assume perfect scalability; that is, adding more nodes does not increase or change the timings of the portions of the calculations. Of course this is certainly not true, but it can still produce useful information about the potential of the computations. From the model, one can trivially derive formulas for the time and vector sizes needed to achieve any fraction, $\frac{\beta}{\beta+1}$, of the peak performance using

$$T(t_\beta) = b(1 - \frac{l}{t_\beta}) = \frac{\beta}{\beta+1} \times b.$$

Thus $t_\beta = (\beta+1)l$. Similarly $n_\beta = \beta \times l \times b$. So, for example, to obtain 90% of peak on the GPUs, one needs a vector length of $9l_G \times b_G$; on Summit this is 5.022×10^8. To achieve 99% of peak one needs a vector length of $99l_G \times b_G$. The combination of latency and bandwidth provides the following.

- An easy way to determine, for a given size problem, the **fraction of peak one will achieve**.

$$\beta = \frac{n}{l \times b}.$$

[1] For $t > l$.

- A way to compare two systems.
  - In order to achieve the **same** β on both systems, the ratio of the time is the ratio of the latencies while the ratio of the needed problem sizes is the ratio of each system's $l \times b$. For example, on Summit, since $l_G/l_C = 10$, the GPUs will take 10 times longer to reach the goal. Similarly, since $b_G/b_C = 22$, the problem size must be 220 times larger on the GPU.
  - In order to solve the **same size problem**, n on both systems, the ratio of the times is

$$\frac{t_C}{t_G} = \frac{l_C + \frac{n}{b_C}}{l_G + \frac{n}{b_G}}.$$

    On Summit, if we select the size that is needed to achieve 90% of peak on the GPUs, the time required on the CPUs will be 20.2 times as long.
  - To achieve the **same runtime** on both systems by increasing the number of the CPUs used, let n_C be the vector size on a single CPU system. Then

$$l_C + \frac{n_C}{b_C} = l_G + \frac{n}{b_G},$$

    resulting in

$$n_C = b_C(\frac{n}{b_G} + l_G - l_C).$$

    Again, if $n = 5.022 \times 10^8$, that is, achieving 90% of peak on the GPU, then $n_C = 2.467 \times 10^7$, and the number of CPU units needed is 21. Note that if we ignore latencies, then the ratio of the times and the number of nodes needed is the ratio of the bandwidths.
  - In order to achieve the **runtime dictated by the latency of the CPU system**, rather than the GPU system (assuming 90% utilization of both), the number of CPU nodes needed is

$$\frac{n}{n_C} = \frac{l_G \times b_G}{l_C \times b_C}.$$

    For Summit this would require 222 nodes, but note that the runtime would be 9.9 times faster.

- A way to **select the number of nodes to use for a given size problem to come close to minimizing the runtime**. First note that the latency provides an absolute lower bound on the compute time, regardless of the problem size and the number of nodes available. Assume one is willing to accept a lower efficiency, β_s; in return for a faster compute time with more units, then $t_s = \frac{\beta_s+1}{\beta+1}t$. The new number of processors is given by $\frac{\beta}{\beta_s}$. Thus if one desires to cut the time by $\frac{1}{k}$, one must use $\beta_s = \frac{\beta+1}{k} - 1$, and the number of processors needed is $\frac{k\times\beta}{\beta+1-k}$. If β is 9 (90% efficiency) and k is 2 (half the runtime) then the number of nodes needed increases by a factor of 2.2. If instead one desires to run 4 times faster, then the number of processors increases by 6. A speedup of 8 would require 36 times as many nodes. Note that the bandwidth of the system plays no role in the analysis.

Table 4 provides a summary of the Summit system and PETSc's vector performance on one node. This is followed by notes and observations on the data.

		Vec size	CPU	interconnect	GPU
Sockets			2		2
Cores/GPUs			42		6
Latency					
	VecDot				
		small	17	-	93
		large	-	-	89
	VecAXPY				
		small	9	-	69
		large	-	-	89
	VecCopy				
		small	4	48	-
		large	-	43	32
Bandwidth					
	VecDot				
		medium	32		567
		large	33		667
	VecAXPY				
		medium	40		375
		large	28		627
	VecCopy				
		small	36	42	-
		medium	32	-	559
		large	24	35	593
Launch time of null kernel				10	
CPU-GPU synchronization after launch				11	
CPU-GPU synchronization when free				6	
Size for 90% of peak	VecAXPY		2.268×10^6		5.022×10^8
Nodes for the time of $10l_G$			20.2		1
Nodes for the time of $10l_C$			222		-

Table 4: Summary of PETSc vector performance on Summit. Latency is in 10^{-6} seconds, bandwidth in 8,000 Mbytes/second.

We note the following.

- When performing scaling studies on a node (that is, increasing CPU processor or GPU counts), correctly choosing resource sets is crucial; otherwise the results can be misleading.
- GPU virtualization has a small negative impact (up to 20% on vectors of length 10^8) on the performance of the GPU vector computations (Figures 6 and 7). For codes that contain a significant portion of CPU computations, it makes sense to use many or all of the CPU cores as MPI ranks and have them share the virtualized GPUs. For pure GPU computation, there is no benefit to virtualization and possibly a performance loss.
- Using pinned memory is crucial to achieve high throughput in copying between the CPU and GPU. We found a 4.5 times speedup of pinned over nonpinned memory for large vectors and smaller speedups for small vectors.
- Timing results can be collected on the GPU, on the CPU, or some combination of the two. Understanding the goal of the data collection is important in deciding which timings to collect.
- We do not measure the throughput directly between GPUs since PETSc currently has no mechanism to utilize this hardware.
- The performance of the vector operations on the GPU is affected by a combination of latencies. These include the following:

– CPU to GPU launch time[2]. From Table 4 it is 10×10^{-6}. This means, for example, that only around 100,000 kernel launches are possible in one second. The item labeled "CPU Synchronization after launch" is the time the CPU waits for the synchronization from the GPUs after the launch subroutine has returned; this is the time at which the CPU knows the null kernel is complete.
– GPU kernel synchronization (wait) time with the CPU[3] (Table 4.)
– Main GPU memory access latency.
– Occupancy of all of the thousands of compute units.
– For routines that return values to the CPU, such as VecDot, the additional latency of the data movement back to the CPU.

The performance of the PETSc VecAXPY operation on a Summit node can be summarized as follows: much higher bandwidth on the GPUs (22 times higher), and significantly lower latencies on the CPU (10 times lower). Much larger vectors are required on the GPU to achieve high performance (over 220 times larger, the product of the ratios of the bandwidth and latency).

Acknowledgments

We thank Junchao Zhang for several crucial technical corrections. We thank Jed Brown for introducing us to the work-time spectrum plot. We thank Todd Munson and Karl Rupp for their useful comments to improve this report.

This material was based on the work supported by the U.S. Department of Energy, Office of Science, Advanced Scientific Computing Research under Contract DE-AC02-06CH11357.

This research was supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration.

This research 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] Summit User Guide Website Authors. Summit User Guide. https://www.olcf.ornl.gov/for-users/system-user-guides/summit/summit-user-guide/. Accessed: 2019-08.

[2] Satish Balay, Shrirang Abhyankar, Mark F. Adams, Jed Brown, Peter Brune, Kris Buschelman, Lisandro Dalcin, Alp Dener, Victor Eijkhout, William D. Gropp, Dinesh Kaushik, Matthew G. Knepley, Dave A. May, Lois Curfman McInnes, Richard Tran Mills, Todd Munson, Karl Rupp, Patrick Sanan, Barry F. Smith, Stefano Zampini, Hong Zhang, and Hong Zhang. PETSc users manual: Revision 3.13. Technical Report ANL-95/11 - Rev 3.13, Argonne National Laboratory, 2020.

[3] Satish Balay, Shrirang Abhyankar, Mark F. Adams, Jed Brown, Peter Brune, Kris Buschelman, Lisandro Dalcin, Alp Dener, Victor Eijkhout, William D. Gropp, Dinesh Kaushik, Matthew G. Knepley, Dave A. May, Lois Curfman McInnes, Richard Tran Mills, Todd Munson, Karl Rupp, Patrick Sanan, Barry F. Smith, Stefano Zampini, Hong Zhang, and Hong Zhang. PETSc Web page. http://www.mcs.anl.gov/petsc, 2019.

[2] Determined by a simple program that runs on six MPI ranks and launches six GPU kernels simultaneously. It measures the time for a no-op kernel; the time for $tkernel <<< 2000, 32 >>> ()$; with $\_global\_\_voidtkernel(){}$ [6]

[3] Time waiting on the CPU for the result *cudaDeviceSynchronize()*.

[4] J. Chang, K.B. Nakshatrala, M.G. Knepley, and L. Johnsson. A performance spectrum for parallel computational frameworks that solve PDEs. *Concurrency and Computation: Practice and Experience*, 30(11):e4401, 2018.

[5] Justin Chang, Maurice S. Fabien, Matthew G. Knepley, and Richard T. Mills. Comparative study of finite element methods using the time-accuracy-size(tas) spectrum analysis. *SIAM Journal on Scientific Computing*, 40(6):C779–C802, 2018.

[6] Robert Crovella. Reduce overhead of launching a new thread block. https://devtalk.nvidia.com/default/topic/1029890/reduce-overhead-of-launching-a-new-thread-block, 2018.

[7] Judy Hill. Summit at the Oak Ridge Leadership Computing Facility. https://press3.mcs.anl.gov/atpesc/files/2018/08/ATPESC_2018_Track-1_6_7-30_130pm_Hill-Summit_at_ORNL.pdf. Accessed: 2019-11.

[8] John D McCalpin et al. Memory bandwidth and machine balance in current high performance computers. *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, 2(19–25), 1995.

[9] Victor Minden, Barry Smith, and Matthew G Knepley. Preliminary implementation of PETSc using GPUs. In *GPU Solutions to Multi-scale Problems in Science and Engineering*, pages 131–140. Springer, 2013.

[10] Exascale Support Team. Exascale web page. https://exascaleproject.org/, 2019.

Mathematics and Computer Science Division
Argonne National Laboratory
9700 South Cass Avenue, Bldg. 240
Argonne, IL 60439

www.anl.gov

Argonne National Laboratory is a U.S. Department of Energy
laboratory managed by UChicago Argonne, LLC