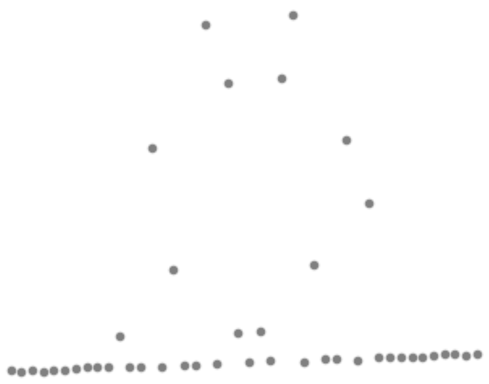
Report ANL-19/41

Evaluation of PETSc on a Heterogeneous Architecture the OLCF Summit System

Part 1: Vector Performance

Mathematics and Computer Science Division



The image above is the throughput for vectors of nearby sizes performing a dot product on the IBM POWER9. This demonstrates how the complexity of the hardware design can produce different performance results in surprising ways.

*ANL-19/41

Evaluation of PETSc on a Heterogeneous Architecture the OLCF Summit System

Part 1: Vector Performance

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

October 9, 2019

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

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

1 Introduction

We report on basic performance of the Portable, Extensible Toolkit for Scientific Computation (PETSc) [?, ?] on the IBM/NVIDIA Summit computing system [?] 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 [?]. 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 Summit, the performance of PETSc vector operations, and provides a limited analysis of the causes of the results. The planned exascale computing systems have a similar design to Summit thus it is important to have a well developed understanding of Summit in preparation for these systems.

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. A GPU on Summit can be utilized by more than one MPI rank simultaneously, the term NVIDIA uses is Multi-Process Service (MPS), we use the term virtualization. For example a single physical GPU may be treated as 4 virtual GPUs by four MPI ranks running on four CPU cores. Virtualization is enabled by submitting jobs with the bsub option `-alloc_flags gpumps`.

To obtain high performance on Summit's node it is important to select the optimal physical cores and associate them with appropriate GPUs. The two POWER9 processors are each connected directly to three of the GPUs (Figure 1, left) so cores associated with each process should, for best performance, utilize one of the three GPUs connected to the same socket. By default when only utilizing the CPUs (for example, to compare GPU and CPU performance) the CPU cores are assigned from the first socket until it is full before using cores from the second socket; thus benchmarking numbers that utilize these defaults are not appropriate. To utilize the CPUs effectively we advocate for the use of two resource sets each with 21 associated cores to take advantage of both sockets of the Summit node (red and yellow in Figure 1, right). Furthermore, adjacent CPU cores share L2 and L3 cache so we use the jsrun option `--bind packed:2` to bind each rank to two CPU cores so that each process 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, which we deem appropriate for most high performance computing (HPC) algorithms. These choices are shown with seven MPI ranks in Figure 1, right. Since the Summit system has 6 GPUs and 42 POWER9 cores we will, when comparing performance of the GPU to the CPU, often compare the performance of one GPU to seven CPU cores.

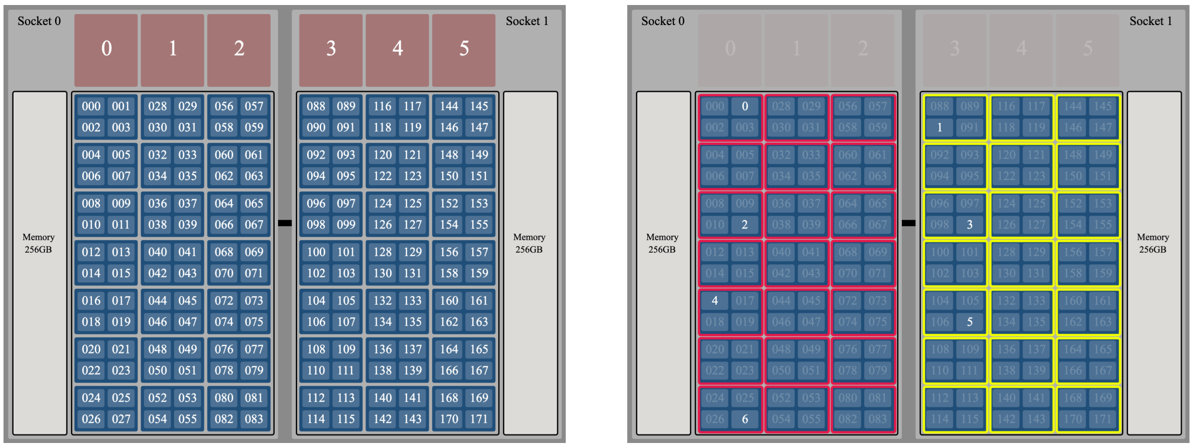


Figure 1: One node of Summit (left) and CPU allocation of 7 MPI ranks (right) from https://jsrunvisualizer.olcf.ornl.gov/.

2.1 PETSc Vector Operations

We present results from runs of PETSc vector operations on one node of Summit. The program creates vectors with random entries, copies them to the appropriate memory and then performs operations. We report the flop rate in Mflops/s for various vector sizes utilizing both GPUs and CPUs. We also report memory throughput (in 8 Mbytes/s for easy comparison with the floating pointing results) for vector copy and set operations. We use the term "throughput" to mean the amount of data moved in a time interval. This is a combination of latency and bandwidth. In all cases the vectors are already in the memory of the device where the performance is being measured. All operations are synchronized with `CudaDeviceSynchronize()` so the true time of the operation is used. The caches are flushed by performing vector operations on other vectors to remove any effects of the data, by happenstance, 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

- $x = a * x + y$, 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 cudaMemcopy on the GPUs). In addition we use cudaMemcpy for copies between the CPU and GPU.
- setting a vector to a single value, implemented with VecSet function (which utilizes cudaMemset to set a vector to zero 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.

3 Experimental results

Figure 2 presents a high level view of the performance of the Summit nodes. Details are discussed below. Note the use of the log scale and that towards the right the GPU is performing significantly better than the 42 CPU cores, while towards the left the CPU is faster.

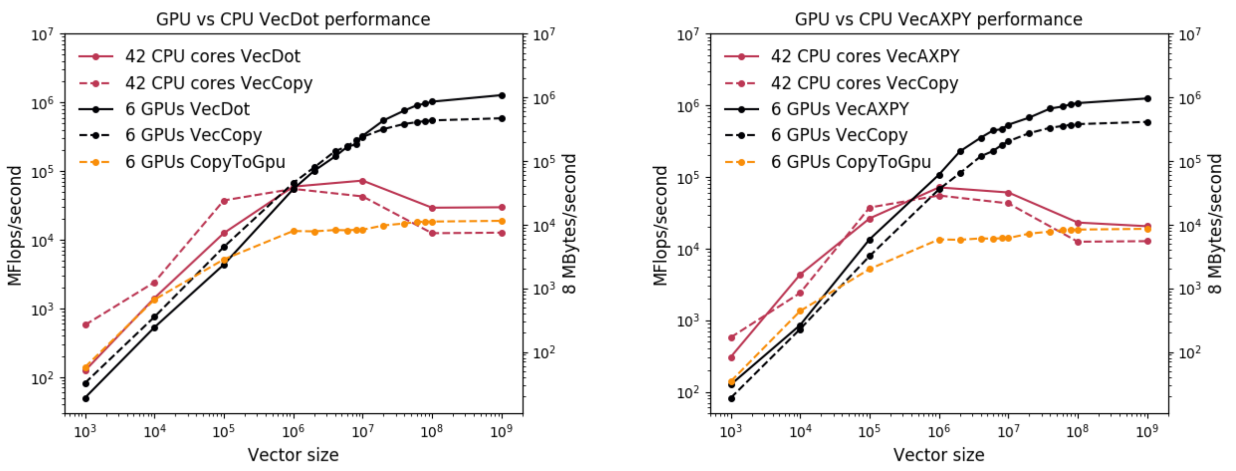


Figure 2: Effect of vector size on vector operations compared with memory throughput (one MPI rank per GPU). Note the log scale.

Figures 3 and 4 compare the performance of the GPUs and CPUs for increasing vector sizes. Note the cache effects of the CPU cores for moderate sized vectors. For shorter vectors, less than around 10^6 entries, the CPU performance for vector operations including copies, is far superior to the GPU. For larger sizes, on the other hand, the GPU can be roughly 20 times faster. This means that 97 percent of the achievable vector performance for large vectors is on the GPUs. Performance on the CPU is relatively independent of vector size (outside the cache effect size), but for the GPUs is low except for large vectors. The drop in performance of the CPU at 21 cores is because the remaining cores share L2 and L3 caches with previously utilized cores and must also share the memory bandwidth.

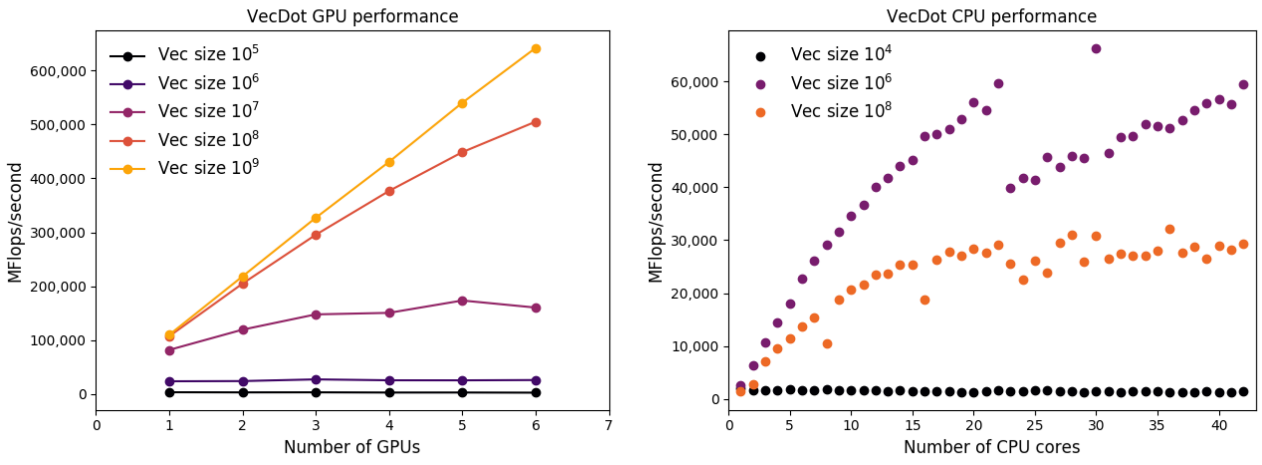


Figure 3: VecDot flop rate on GPUs, with one MPI rank per GPU, (left) and CPUs (right).

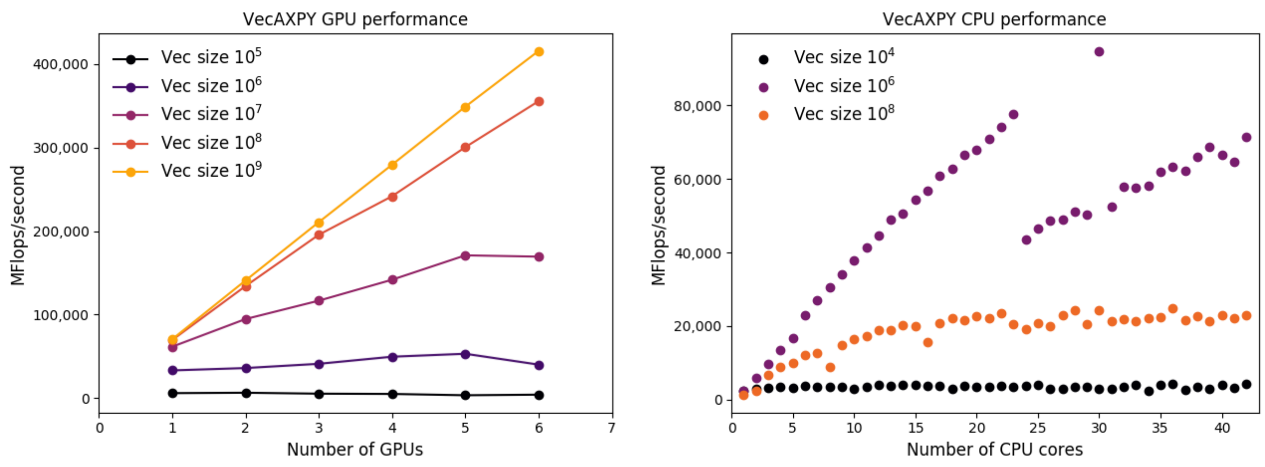


Figure 4: VecAXPY flop rate on GPUs, with one MPI rank per GPU, (left) and CPUs (right).

Figures 5, 6, and 7 explore the performance effects of GPU virtualization. Performance of the GPUs is essentially the same even when divided into up to eight virtual GPUs for smaller vectors. For larger vectors the performance is worse. Note that in Figure 5 the vectors on six GPUs are 1/6 the size they are on one GPU, this means the effective throughput numbers each are utilizing are lower than may have been expected, since the six GPUs no longer have a long enough vector for full performance. Figure 7 shows the clear benefits in CPU to GPU copies with GPU virtualization. Using 2 MPI ranks per GPU almost doubles the performance, after this the performance is roughly the same, presumably because the CPU memory bandwidth has become saturated. This indicates that if an application has many CPU to GPU copies it may benefit from two or more MPI ranks per GPU.

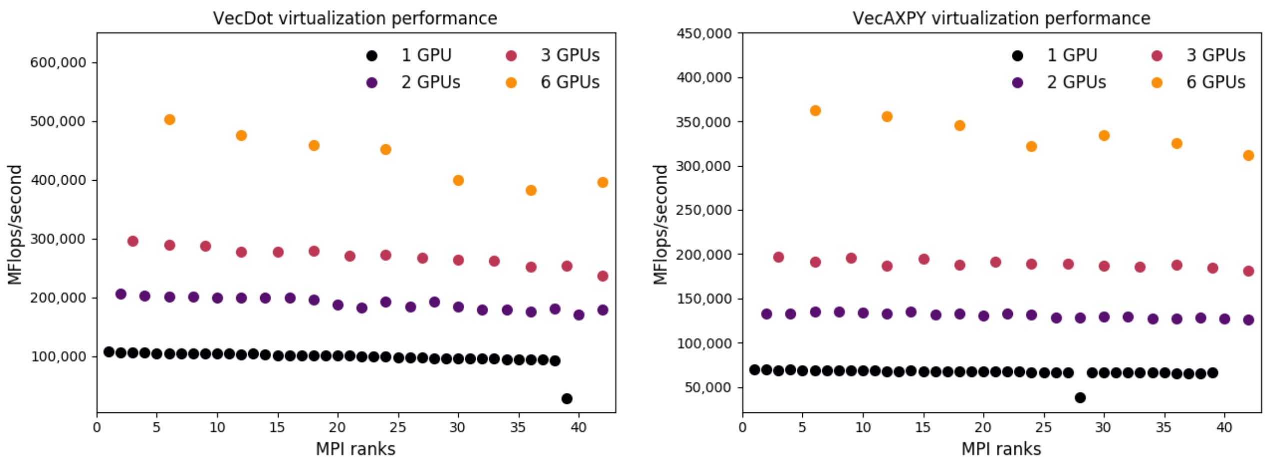
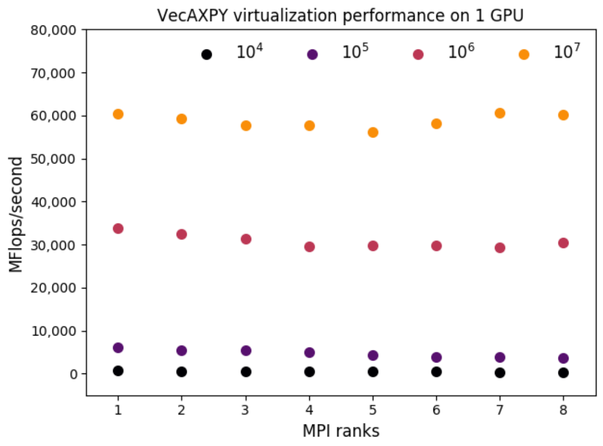


Figure 5: Performance of GPU virtualization for VecDot operations (left) and VecAXPY (right) for vectors of length 10^8.



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

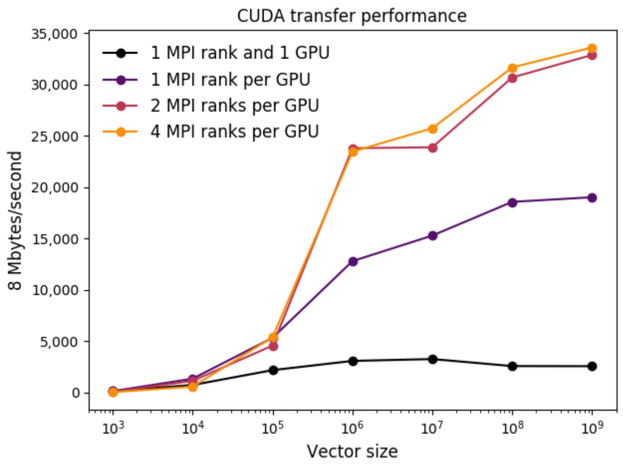


Figure 7: Effect of vector size on CPU to GPU copy throughput with 6 GPUs (multiple MPI ranks per GPU) and 1 GPU with 1 MPI rank.

Figures 8, 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, as they share a common memory, and hence performance improvement decreases as more cores are utilized.

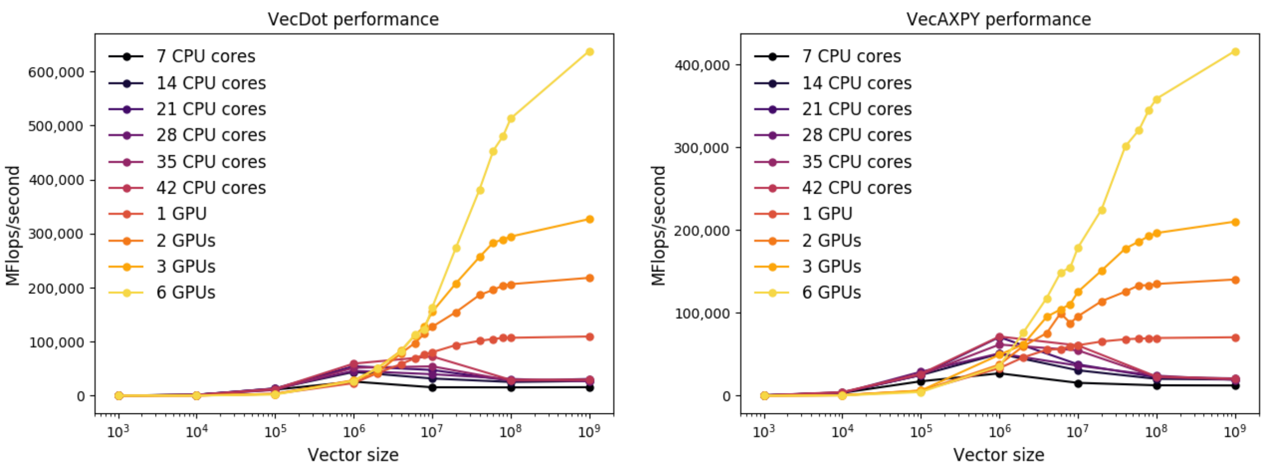


Figure 8: Effect of vector size on flop rate for VecDot (left) and VecAXPY (right) on the CPU and GPUs with one MPI rank per GPU.

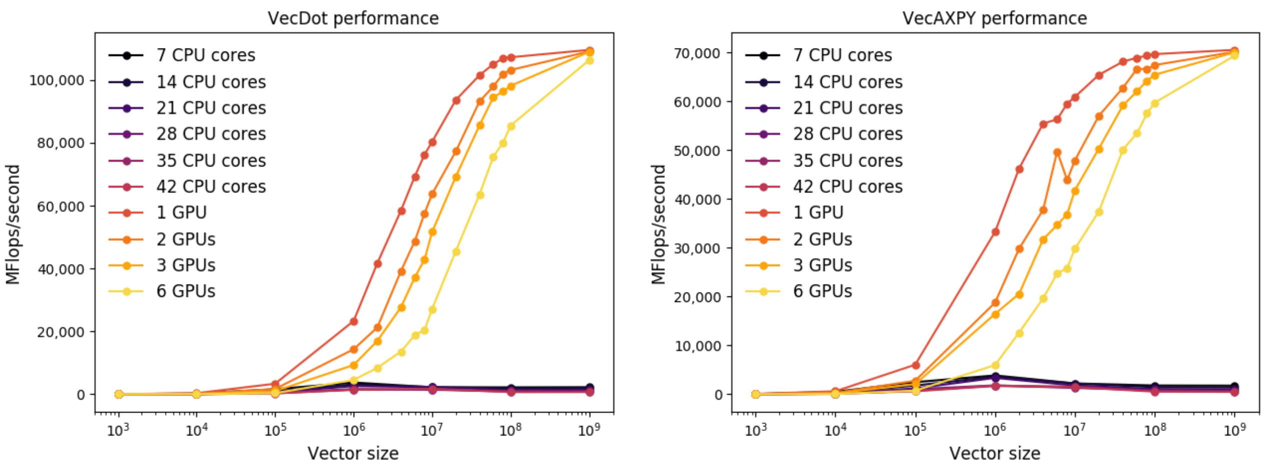
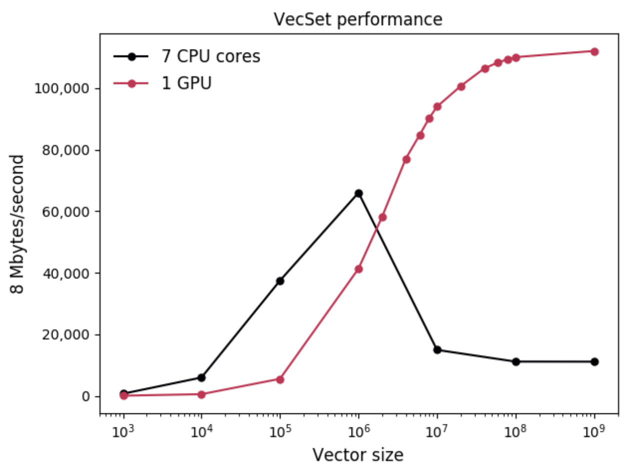


Figure 9: Effect of vector size on flop rate for VecDot (left) and VecAXPY (right), on the CPU and GPUs with one MPI rank per GPU, scaled by the number of CPU cores or GPUs.

Figure 10 compares the achieved throughput for setting a vector to a constant value and copying vectors, including copying from CPU to GPU.



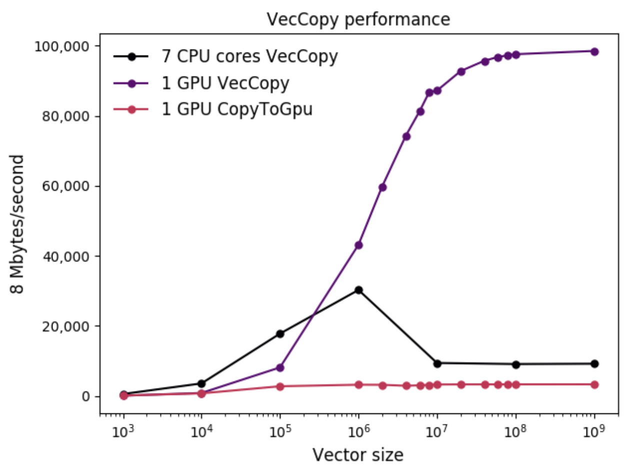


Figure 10: Effect of vector size on VecSet (left) and vector copy (right) throughput.

4 Discussion

The performance of the basic vector operations can be modeled with

$$t = latency + (1/bandwidth) * size.$$

In Table 1 and Table 2 we show the computed (via least squares) latency and bandwidth pairs for each operation for a single GPU and one CPU core. We find that operations on a single GPU follow this simple linear latency/bandwidth model for large vectors, the behavior for small vectors is depicted in Figure 11. For small vectors performance is completely determined by latency. Counter intuitively, working with slightly larger vectors takes (slightly) less time than 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 simply occupy some of the unoccupied units.

Vec size	10^3 - 10^5		10^5 - 10^7		10^7 - 10^8	
Operation	latency	bandwidth	latency	bandwidth	latency	bandwidth
VecDot	90	-	64	110,000	66	111,000
VecAXPY	47	-	29	100,000	44	106,000
VecSet	-	-	15	106,000	22	113,000
VecCopy	-	-	18	88,000	36	99,000
Copy to GPU	19	3,100	86	3,300	269	2,600

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

Vec size	10^3 - 10^5		10^5 - 10^7		10^7 - 10^8	
Operation	latency	bandwidth	latency	bandwidth	latency	bandwidth
VecDot	4	3,900	-119	2,500	-356	2,400
VecAXPY	2	5,600	79	3,500	-525	3,400
VecSet	-1	8,000	-316	2,800	-2,508	2,400
VecCopy	0	7,200	-233	3,300	-1,093	3,200

Table 2: Latency (10^{-6} seconds) and bandwidth (8 Mbytes/second) for vector operations on 1 CPU core.

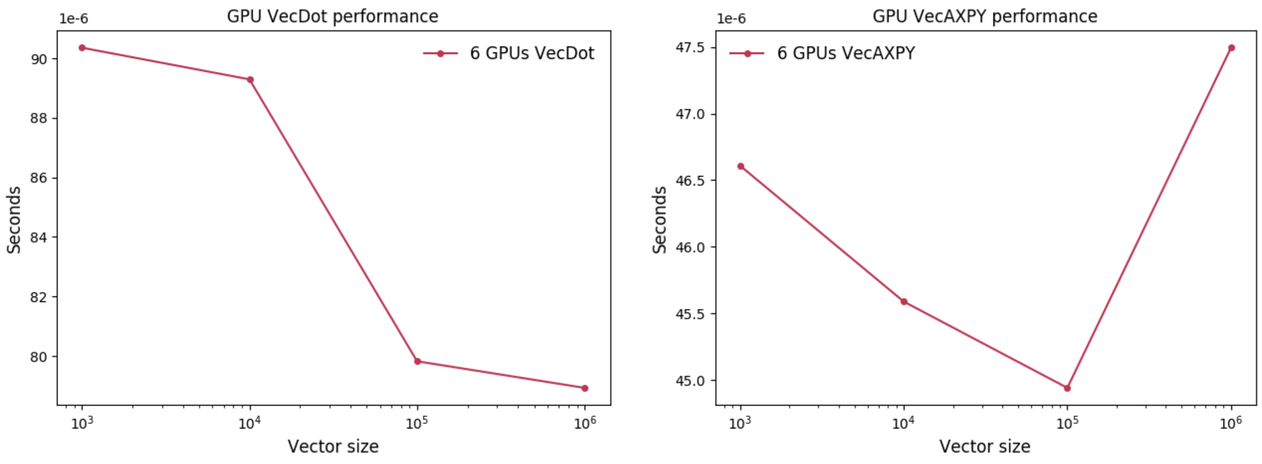


Figure 11: GPU performance on small vectors 10^3 to 10^6.

Operations on one CPU core do not follow one linear model for all vector sizes. We fit the latency and bandwidth to different regimes in Table 2. Some of the latencies and bandwidths computed are actually negative, this showing the limitations of the simple linear model. But even those values provide some useful information. Note that the latency and bandwidth computations for the CPU cores are determined for a single core (not seven as in the earlier plots), this is to make getting the timings simpler and more accurate since it is not possible to launch the seven cores computation simultaneously. Note that one cannot just compare the bandwidth numbers in two tables directly; one must multiple by six for the total GPUs and around nine (this is the point at which the bandwidth on the CPUs are saturated) for the CPU cores

For a given size vector the *throughput* can be computed as

$$throughput(size) = size/time = \frac{size}{latency + (1/bandwidth) * size.}$$

This model can explain the shape of the curves in Figure 2 and many of the other figures. The higher the latency the more the curve shifts to the right. A negative latency indicates the curve is less than linear. This, for example, occurs on the CPU cores as the vectors become larger than the cache size, the throughput is dropping with increased vector sizes.

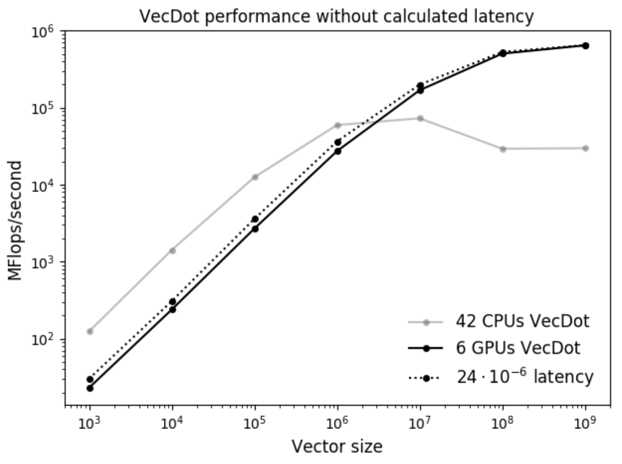
For large vector sizes, the total flop rate over six GPUs for VecACPY is about 20 times faster than for 42 POWER9 cores. See Figure 4 with a vector size of 10^8.

The performance of the vector operations on the GPU are affected by a variety of latencies. These include

- CPU to GPU launch latency – 8 microseconds[1]
- GPU kernel launch time – 16 microseconds[1]
- Main GPU memory access latency
- Occupancy of all of the thousands of compute units

From Figure 11 and Table 1 the VecDot latencies are 90 and then 66 microseconds, for VecAXPY they are 46 and 44 microseconds. (We don't understand why the latencies are higher for the dot product). The kernel launch latency gets a great deal of blame for poor GPU performance but from these numbers it is clearly not the only cause of the low performance for small vectors. Figure 12 shows the data provided initially in this report (Figure 2) but with the launch latencies subtracted out from the time (resulting in higher throughputs). This figure demonstrates that though these latencies are problematic, removing them does not drastically improve the performance.

[1]Determined by a simple program that measures the latency for a no-op kernel



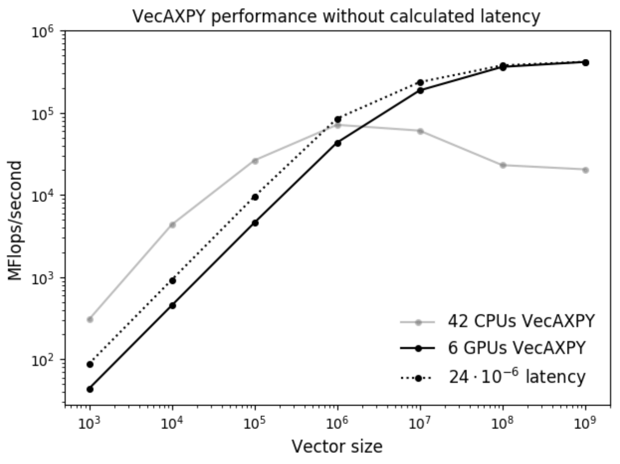


Figure 12: Projected performance without 24 microsecond latency.

Notes and observations:

- From Table 1 and the numbers above one sees the latency for small vectors on the CPU is significantly smaller than for the GPU. About 20 times lower.
- GPU virtualization has little impact on performance of the GPU computations. For codes that contain a significant portion of CPU computations it then makes sense to use many or all of the CPU cores as MPI ranks and have them share the virtualized GPUs. Since the achieved throughput for copies from the CPU to GPU increases rapidly when using two MPI ranks per GPU instead of one, for codes that require significant communication between the CPUs and GPUs, virtualization also makes sense.
- For these results PETSc 3.11 was with built with CUDA version 10.1.168 and PGI compilers version 19.4.
- The bandwidth numbers for communication between the CPUs and the GPUs are significantly below the hardware peak and those reported by OLCF (using custom code). We have no explanation for this.
- We do not measure the throughput directly between GPUs since PETSc currently has no mechanism to utilize this hardware.
- 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