An Empirical View of SLATE Algorithms on Scalable Hybrid Systems

Asim YarKhan
Jakub Kurzak
Ahmad Abdelfattah
Jack Dongarra

Innovative Computing Laboratory

September 27, 2019

This material is based upon work supported by the National Science Foundation under Grant No. 1527706, "SHF: Small: Empirical Autotuning of Parallel Computation for Scalable Hybrid Systems".

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.

Revision	Notes
05-2019	first publication

```
@techreport{yarkhan2019empirical,
  author={YarKhan, Asim and Kurzak, Jakub and Abdelfattah, Ahmad and Dongarra, Jack},
  title={An Empirical View of {SLATE} Algorithms on Scalable Hybrid Systems},
  institution={Innovative Computing Laboratory, University of Tennessee},
  year={2019},
  month={September},
  number={ICL-UT-19-08},
  note={revision 09-2018}
}
```

Contents

1 Introduction

In the tuning and analysis of high performance software, it is crucial to be able to connect the performance achieved by the software to the capabilities of underlying hardware. If we are unable to determine to determine and describe the capabilities of the hardware, we will not be able to tell if the software is reaching the expected performance levels.

The Roofline model [1] is an intuitive visual performance model that can be used to put software performance in the context of the underlying hardware. The peak computation and communication capabilities of the hardware are used to bound the space of achievable performance. The performance of a software routine can be viewed within these bounds to determine if it is close to achieving the available performance.

This report looks at the implementation and performance of two mathematical routines from the SLATE linear algebra library project. The routines being examined are chosen to illustrate different aspects of the software and underlying hardware.

In this report we look at linear algebra routine in the hardware context of a current state-of-the-art high performance supercomputer Summit [1]. This class of machine combines CPUs with multiple GPUs and derives most of its computational capability from GPU acceleration. We will see if the Roofline Model is able to provide us with useful insight into the achieved performance on this class of machine. Understanding the behavior of high performance scientific software on Summit is essential along the path to Exascale computing.

1.1 Motivation

There have been very few systematic studies looking at mapping the performance of software libraries to the capabilities of the distributed hybrid hardware platforms such as Summit. Each node of Summit has 2 IBM Power9 processors and 6 NVidia V100 GPU's. The GPU's account for 97 percent of the double precision computational flops provided by the machine. Due to the presence of these powerful GPUs, the nodes are highly provisioned for compute capability. But the data movement within the nodes from CPUs-to-GPUs as well as the message passing between the nodes are a well-recognized bottleneck for obtaining scalable performance. We will attempt to use the Roofline model and other bounds models to visualize the performance of numerical software on this machine.

2 Background

2.1 The Roofline Model

The Roofline model [1] is a visual performance model which enables an intuitive understanding of the performance of a given computation kernel running on specific hardware architecture. The hardware architecture can be used to determine multiple bounds on performance based

[1] https://www.olcf.ornl.gov/olcf-resources/compute-systems/summit/

on characteristics such as the communication bandwidths (cache, memory, network, etc.) and on computational capabilities (from single-core, multi-core, accelerators, etc.).

The machine hardware characteristics are related to three principal components affecting performance of a software kernel: the computation it does, the communication required, and the locality of the data.

A basic Roofline model for an algorithm can be obtained by plotting floating-point performance (flop/sec) as a function of arithmetic intensity of the software, where arithmetic intensity is the number of flops/byte of data. If a computation has a high arithmetic intensity it is less likely to be limited by the communication bounds of the hardware; it is "compute-bound". In contrast, a computation which has a low arithmetic intensity is likely to be limited by the communication bandwidth of the hardware; it is "communication-bound". The basic Roofline model can be extended by adding more bounds based on other known hardware/software/-compiler limitations; for example we could have peak performance bounds with and without instruction-level-parallelism enabled. These can be used during kernel development to bound the maximum performance achievable by using different optimization techniques.

The Roofline model provides a readily understandable representation that puts the hardware bounds and the software kernel performance into context with each other, allowing a developer to see if the kernel's performance results meet expectations or if they can/should be improved.

2.2 The SLATE Linear Algebra Project

Software for Linear Algebra Targeting Exascale (SLATE) [2] [2] is being developed as part of the Exascale Computing Project (ECP) [3], which is a collaborative effort between two US Department of Energy (DOE) organizations, the Office of Science and the National Nuclear Security Administration (NNSA). SLATE will deliver fundamental dense linear algebra capabilities for current and upcoming distributed-memory systems, both GPU-accelerated and multicore-only. SLATE is designed to serve as a replacement for ScaLAPACK for the upcoming pre-exascale and exascale DOE machines. SLATE will accomplish this objective by leveraging recent progress in parallel programming models and by strongly focusing on supporting hardware accelerators.

The principles of the SLATE software framework were laid out in SLATE Working Note 3 [2]. SLATE's design relies on the following principles:

Tiled Matrix Layout: The matrix is represented as a set of individual tiles with no constraints on their locations in memory with respect to one another. Any tile can reside anywhere in memory and have any stride. Notably, a SLATE matrix can be created from a LAPACK matrix or a ScaLAPACK matrix without making a copy of the data.

Multi-level Scheduling: Node-level scheduling relies on two levels of nested OpenMP tasking, with the top level of tasking responsible for managing dynamic dataflow dependencies and the bottom level responsible for deploying large numbers of independent tasks to multi-core processors and accelerator devices.

[2] http://icl.utk.edu/slate/
[3] https://www.exascaleproject.org

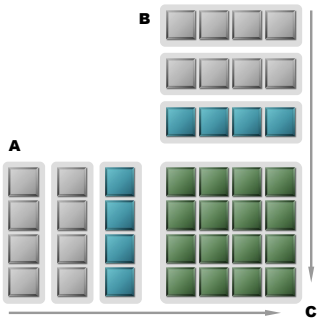

Figure 1: Implementation of *gemm* as a sequence of outer products.

Batch Execution: Batch BLAS is used extensively for obtaining node-level performance. Most routines spend the majority of their execution in the call to batch `gemm`.

Explicit Communication: The Message Passing Interface (MPI) is used for message passing with emphasis on group communication, with the majority of communication being structured as multi-casts to varying sets of processes that are identified dynamically.

2.3 Matrix-Matrix multiplication in SLATE

The implementation of SLATE matrix-matrix multiplication is described in detail in [3]. Figure 1 illustrates the implementation of the `gemm` operation $C \leftarrow \alpha AB + \beta C$ in SLATE as a sequence of outer products executed in an embarrassingly parallel manner, in the sense that each tile of the output matrix can be computed independently. This outer product `gemm` implementation enables a large number of independent operations that can be launched using batch execution on the GPUs. The batch execution can extract high performance from the GPU hardware even if each of the individual operations in the batch is relatively small.

The theoretical arithmetic intensity of the matrix-matrix multiplication operation is easily obtained. The double precision matrix-matrix multiplication from BLAS $dgemm(n)$ does $2n^3$ floating point operations while communicating $4n^2$ data. So it has an arithmetic intensity of $n/2$ flops/byte.

2.4 Matrix Norm in SLATE

The implementation of SLATE `norm` routines is described in detail in [3]. Figure 2 illustrates the most complicated case of the `matrix-norm`, where the matrix is spread across multiple distributed memory nodes and across multiple accelerators in each node. The figure shows the stages of computing the `one-norm`, i.e., finding the maximum column sum. This means computing the sum of all elements in each column and then finding the maximum sum. Figure 2 illustrates the case with four nodes in a 2D block cyclic arrangement and four devices per node, also in a 2D block cyclic arrangement. The process consists of four steps:

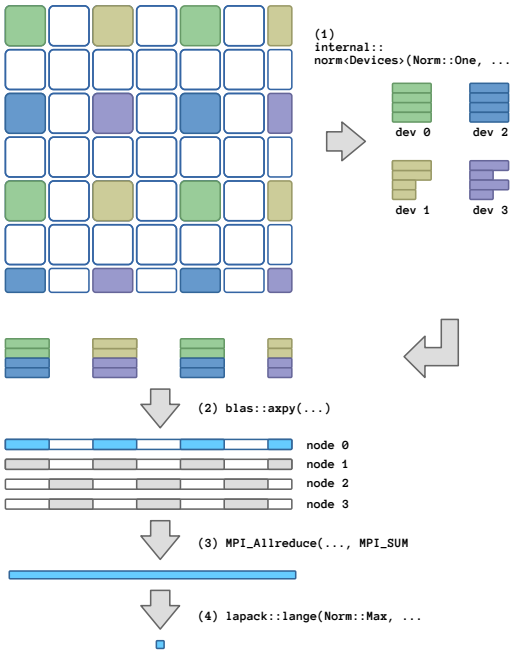

Figure 2: Stages of the one norm using four nodes in a 2D block cyclic arrangement and four devices per node in a 2D block cyclic arrangement.

(1) Within each node, each device computes column sums for each of its tiles, using a specialized device kernel.
(2) Within each node, contributions from all devices are summed up to a local vector of partial sums using `blas::axpy`.
(3) Partial sums from all the nodes are summed up using `MPI_Allreduce`.
(4) Within each node, the maximum sum is found using `lapack::lange`.

The `norm` implementations can run on CPU's or use device kernels implemented in CUDA for GPU acceleration. These `norm` routines are embarrassingly parallel and basically boil down to a sequence of reductions.

We use a theoretical flop count for the `norm` operation. For the `one-norm` on a $m \times n$ matrix we would need $(m-1) \times n$ floating point operations (additions) and would move $2 \times m \times n$ numbers. This means that the arithmetic intensity of the `norm` operation is a constant regardless of m or n. For this report, we present the performance of the `norm` operation on a bandwidth-against-matrixsize chart which will allow us to see how the `norm` implementation is constrained by the available bandwidth limits.

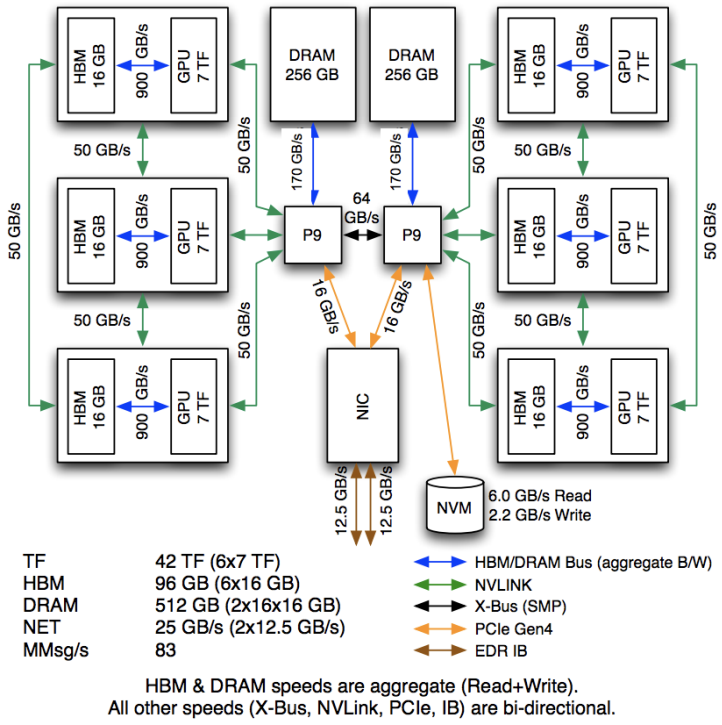

Figure 3: Summit node architecture (image from ORNL OLCF July 30 2018 Training Workshop by Judy Hill).

3 Experiments

3.1 Environmental Environment

Performance numbers were collected using the Summit system [4] at the Oak Ridge Leadership Computing Facility (OLCF). Summit is based on IBM POWER9 processors and NVIDIA V100 (Volta) accelerators.

Summit contains 4608 nodes, each with two sockets (see Figure 3 for the node architecture). Each socket has one POWER9 CPU and three NVIDIA V100 GPUs. Each two-socket node (2 POWER9s and 6 V100s) provides a double-precision peak-performance of 42 TFlops (teraflops). The three V100 GPUs in a socket are inter-connected with 50 GB/s NVLINK. The two sockets in a node are connected via a 64 GB/s X-Bus, which is a much smaller bandwidth than the inter-socket connectivity. The nodes are interconnected using via a 25 GB/s, dual-rail EDR Infiniband.

The software environment used for the experiments included GNU Compiler Collection (GCC) 6.4.0, CUDA 10.1.168, Engineering Scientific Subroutine Library (ESSL) 6.1.0, Spectrum MPI 10.3.0.1, Netlib LAPACK 3.8.0, and Netlib ScaLAPACK 2.0.2. All our results are for double precision data types (e.g. `dgemm`).

[4] https://www.olcf.ornl.gov/olcf-resources/compute-systems/summit/

3.2 Experiments with Matrix Multiplication

Our experiments examine if the SLATE gemm implementation is efficiently using and adapting to the available hardware and if we can relate the performance achieved to the Roofline model.

In order to explore the behavior of different bounds, we look at the performance of double precision gemm on (1) a single socket, (2) a single node, and (3) four nodes. On each of these three situations, we further test on two different execution targets (a) the Power9 CPUs, or (b) the V100 GPUs. These situations expose most of the communication limitations on the machine (Figure 3).

- Single socket CPU: Bandwidth is limited by the communication with main memory.
- Single socket GPU: Communication uses the high bandwidth memory (HBM) for data local to a single GPU and travels between GPUs and CPU over NVLink.
- Single node CPU: Now the communication is additionally limited by the relatively small bandwidth XBar connecting the two sockets.
- Single node GPU: Communication uses the HBM and NVLink but is limited by the XBar connecting the two sockets.
- Four nodes CPU: Communication between the nodes needs to go through the Infiniband network interface.
- Four nodes GPU: Communication goes through the Infiniband network interface, and on a node uses the XBar, NVLink, and HBM.

Our expectation is that the different communication limitations will be evident in the Roofline views of the executions.

Single socket only: Running the CPU gemm implementation on a single socket means that the bandwidth should be limited by DRAM and the maximum performance is limited by the Power9 CPUs. The GPU implementation of gemm on a single socket uses the NVLink interconnect between GPUs and the Power9, which provides 50 GB/s on each connection (GPU-GPU, GPU-CPU) leading to a large available aggregate bandwidth within a socket. The maximum performance of gemm on the 3 GPUs is bounded by the theoretical peak of $3 \times 7.8 = 23.4$ TFLOPS.

Figure 4 shows the roofline curves for double precision gemm on a single socket using the CPU and GPU implementations. The peak performances reached at the right side of the curve are close to the theoretical peaks for the CPUs and GPUs hardware. However, the bandwidth-bound part of the curves (on the left) are fairly distant from the theoretical bounds for communication.

Single node: Running on single node containing two sockets adds a new communication limitation to the gemm implementation. The two sockets are connected via an X-Bar switch that has a bandwidth of 64 GB/s. Both the CPU and GPU implementations of gemm need to communicate through the X-Bar to complete inter-socket communication.

Figure 5 displays a roofline graph that shows the peak performance constraints for the CPUs and GPUs, and the bandwidth dependent constraints from the DRAM, NVLINK and X-bar. In the bandwidth constrained part (left part) of the Roofline graph, both the CPU and the GPU

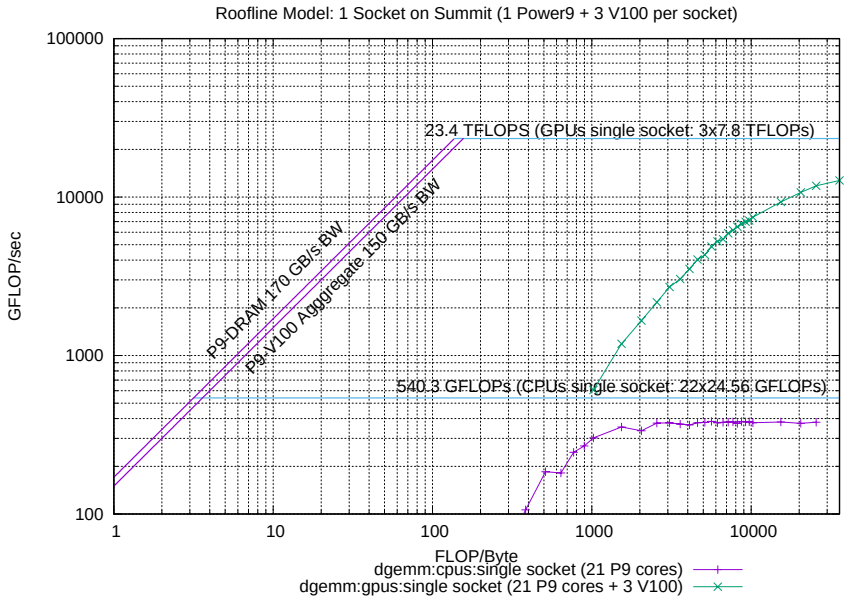

Figure 4: DGEMM performance on a one socket.

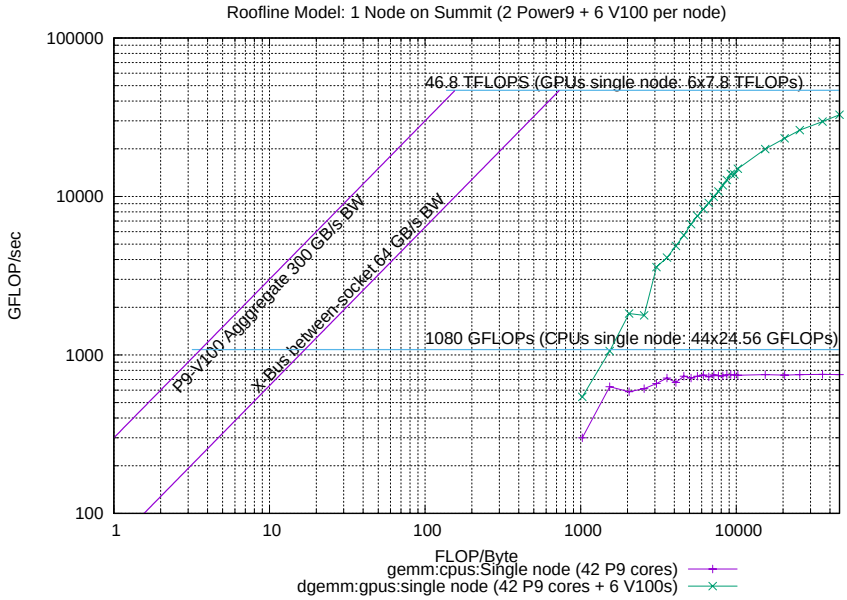

Figure 5: DGEMM performance on one node.

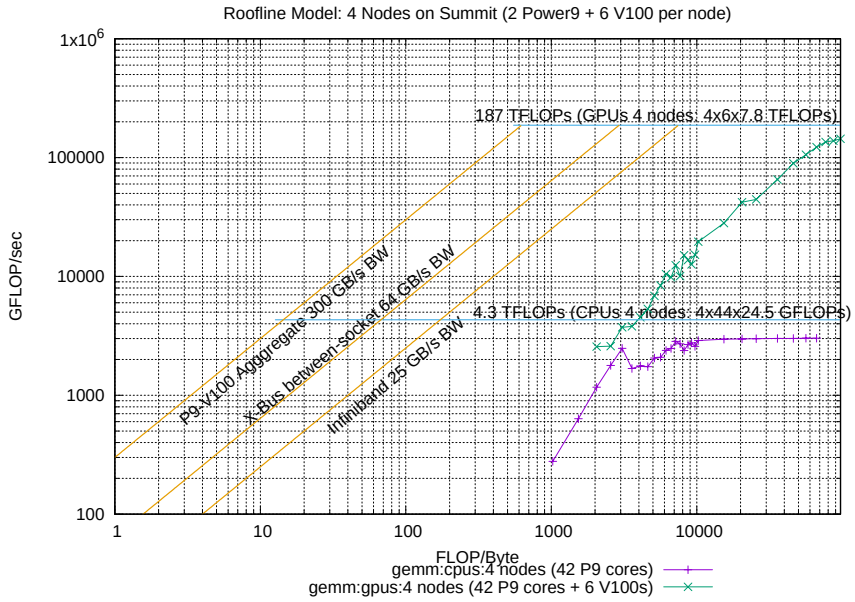

Figure 6: DGEMM performance on 4 nodes .

performance are far from the known constraints. This implies that it may be possible for the implementation to improve in that region. In the computation-bound region of the Roofline graph (right side), the performance of `gemm` on the CPU is approaching 750 GFlops which is fairly close to the theoretical peak performance of 1080 GFlops. For the GPU implementation of `gemm` the performance approaches 32 TFlops, which is reasonable fraction of the theoretical peak performance of 46 TFlops. Note, the GPU performance has not flattened out, but because of the size of the GPU memory we were unable to run larger problems.

Four nodes: We examine the behavior of the SLATE implementation of `gemm` in a distributed setting by running on 4 nodes. In a distributed setting the communication bottleneck is the bandwidth between the distributed nodes. For the nodes on summit, the bidirectional bandwidth provided the Infiniband network interface cards is 25 GB/sec each way (Figure 3).

Figure 6 displays a Roofline graph with the performance curves for `gemm` on the CPUs and on the GPUs. Once again, the communication-bound section of the Roofline curves (left side) are far from the hardware bounds, implying that there should be performance improvements available to the implementation. For the CPU, the computation-bound section of the curve (right side) achieves a maximum performance of 3.0 TFlops, which is a reasonable fraction of the maximum theoretical performance of 4.3 TFlops. The GPU performance for `gemm` reaches 143 TFlops which on 4 nodes (containing 24 V100 GPUs), where the theoretical peak performance is 187 TFlops.

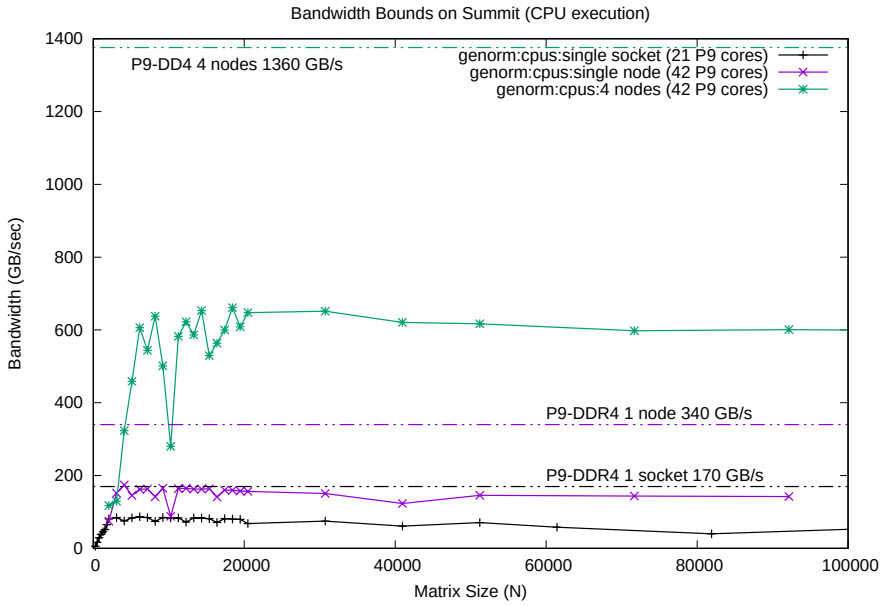

Figure 7: Norm bandwidth achieved using CPUs.

3.3 Experiments with the Matrix Norm

Since the matrix `norm` is bandwidth-limited, these results are measured and presented on bandwidth charts. The arithmetic intensity of the `norm` operation is constant regardless of the problem size, so using a standard Roofline graph would provide limited insights. For our results, we use SLATE to compute the *max norm* of the matrix. The bandwidth achieved for the `norm` operation is calculated by the bytes in the matrix ($n \times m$) divided by the time taken for computing the max `norm`. The performance of the `norm`-operation is measured on 1-socket, 1-node and 4-nodes. The results for the `norm` experiments are displayed grouped by CPU results and then grouped by GPU results.

Max-Norm performance on CPUs: Figure 7 shows the bandwidth achieved by SLATE's `norm` implementation on CPUs and the bandwidth bounds between the P9 CPUs and the DDR4 memory. On a single-socket the peak theoretical bandwidth bound is 170 GB/s and SLATE's CPU `max norm` implementation can achieve approximately 80 GB/s. So, on a single socket we can get 50% of the peak available bandwidth.

On a single-node, consisting of two sockets, the theoretical peak bandwidth from the CPUs to the memory is doubled to 340 GB/s since each socket is accessing local memory banks. This is reflected in the 160 GB/s bandwidth achieved during the `norm` implementation.

Finally, on four-nodes, the bandwidth achieved by the `norm` operation is about 600 GB/s. The `norm` implementation shows very good scaling from the single socket bandwidth (80 GB/s

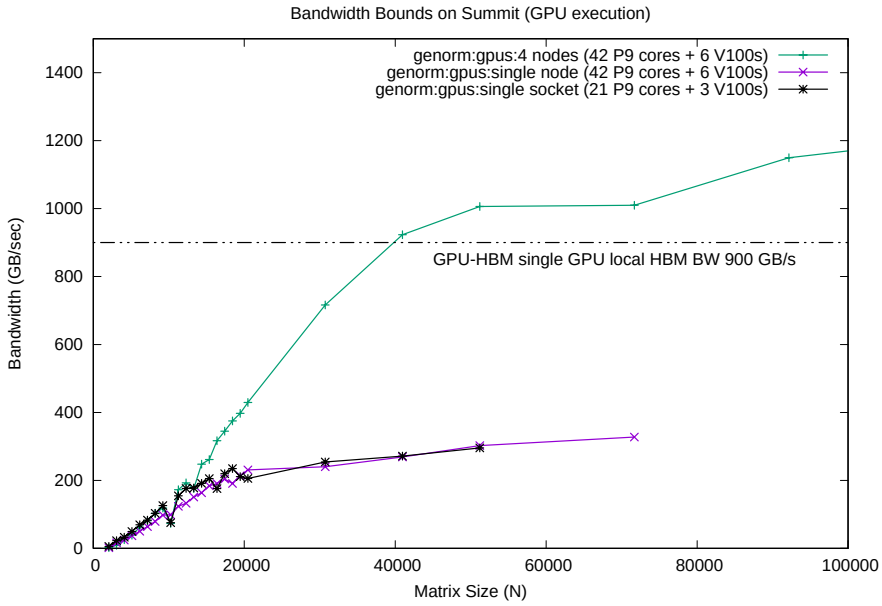

Figure 8: Norm bandwidth achieved using GPUs .

per-socket $\times$ 2 sockets $\times$ 4 nodes = 640 GB/s).

For all the CPU implementations of the `norm` operation, we see very good scalability in the bandwidth achieved, and we get a reasonable 50% of the theoretical peak bandwidth. This means that it is unlikely that the SLATE `norm` implementation on CPUs could be improved to obtain a larger fraction of the theoretical peak.

Max-Norm performance on GPUs: The SLATE implementation of the `norm` operation on reads data from local memory and exchanges a limited amount of information with other processes as needed. For the NVidia V100 GPUs, the bandwidth from the GPUs to local high-bandwidth memory (HBM) is a very fast 900 GB/s (Figure 3). So, relative to speed of accessing local HBM, even exchanging a limited amount of information with other processes can become a bottleneck in the computation.

In Figure 8 it can be seen that using a single socket with 3 V100 GPUs to compute the `max norm`,Z the bandwidth achieved is about 295 GB/s and is still slowly increasing. The bandwidth between a single V100's and the local HBM is 900 GB/s so the operation is only getting a fraction of that.

Using a single node, which consists of two sockets with a total of 6 V100 GPUs, there is no appreciable increase in the bandwidth achieved. This implies that there is some other bottleneck that is preventing the `norm` implementation from using the full available bandwidth.

Looking at the `norm` implementation on four-nodes it is seen that the bandwidth achieved has scaled very well, reaching approximately four times the bandwidth achieved on a single node.

This would seem to imply that the inter-node communication is not the immediate bottleneck for this implementation, however there should be improvements possible at a node level.

We did not display additional bandwidth bounds in the GPU `norm` Figure8, because the aggregated P9-HBM bandwidth for multiple GPUs would so high that the scale of the figure would be lost. It is clear that there are other bounds on the `norm` that are limiting performance.

4 Summary

We examined at the performance of the compute-bound `gemm` operation in the SLATE linear algebra library on a distributed memory GPU-intensive machine. We used Roofline graphs to indicate if the performance achieved matched the expectations and bounds that emerge from the underlying hardware. Our observation is that the `gemm` implementation in SLATE performs well in computation-bound regions for both CPUs and GPUs, however there is an unexplained performance lag in the bandwidth-bound regions. This implies that there may be opportunities for improving SLATE performance in bandwidth-bound regions.

We tested the memory-bound `norm` operation to see if the performance achieved indicates that the SLATE implementation is reaching the bandwidth bounds dictated by the hardware. Our observation is that SLATE's `norm` implementation on the CPU is unlikely to get much improvements, however the GPU implementation should be examined for possible performance improvements.

Roofline models and other hardware-bound models are difficult to use in the context of complex hardware systems because it is difficult to say which hardware bounds are relevant. For example, on a scalable, distributed memory, hybrid system such as Summit, when the operation under consideration is mostly local but requires some global communication, what would be the appropriate communication bottleneck? Our opinion is that hardware-bound models are best suited to simpler systems and small kernels and not well suited for larger operations on complex systems.

References

[1] Samuel Williams, Andrew Waterman, and David Patterson. Roofline: an insightful visual performance model for multicore architectures. *Communications of the ACM*, 52(4):65–76, 2009.

[2] Jakub Kurzak, Panruo Wu, Mark Gates, Ichitaro Yamazaki, Piotr Luszczek, Gerald Ragghianti, and Jack Dongarra. SLATE working note 3: Designing SLATE: Software for linear algebra targeting exascale. Technical Report ICL-UT-17-06, Innovative Computing Laboratory, University of Tennessee, September 2017. revision 09-2017.

[3] Jakub Kurzak, Mark Gates, Asim YarKhan, Ichitaro Yamazaki, Panruo Wu, Piotr Luszczek, Jamie Finney, and Jack Dongarra. SLATE working note 5: Parallel BLAS performance report. Technical Report ICL-UT-18-01, Innovative Computing Laboratory, University of Tennessee, March 2018. revision 04-2018.