ExaSGD Performance Profiling

Kalyan Perumalla Maksudul Alam

July 25, 2022

1 Introduction

Exascale Stochastic Grid Dynamics (ExaSGD) is one of the application projects under the US Department of Energy's Exascale Computing Project (ECP).

`HiOp` is an optimization solver developed at the Lawrence Livermore National Laboratory (LLNL) by Cosmin Petra.

2 Performance Profiling Strategy

2.1 Iterative Profiling and Performance Improvement

Algorithm 1: Iteration for Profiling-based Performance Improvement

```
Result: Accelerated code of whole program
initialization;
while any code units can be sped up do
    Obtain profile of current version of code;
    Identify most time consuming units in the profile;
    Accelerate the identified units;
end
```

2.2 Profiling strategy for Summit

2.3 Devised Profiling Strategy for Summit

We explored different options for profiling tools on the Summit supercomputer. We have identified at least four primary options for profiling: 1) whole program profiling using `HPCToolkit`, 2) whole program profiling with `nvprof`, 3) whole program profiling using `Exa-PAPI++`, and 4) targeted profiling using manual instrumentation.

1. Using `HPCToolkit`
2. Using `nvprof`
3. Using `Exa-PAPI++`
4. Using targeted profiling via manual instrumentation

2.3.1 Using `HPCToolkit`

Summit has a default `HPCToolkit` package installed that does not have GPU support. Therefore, we have obtained the latest `HPCToolkit` version from source and successfully compiled it ourselves on Summit by explicitly adding GPU support. We have been experimenting with this custom-built `HPCToolkit` on Summit and we find that it seems to be a good option to gain a general idea of the program profiles without having to undertake any modification to the source code. In fact, it is possible to use the same binaries as are used for normal execution. The binaries are automatically instrumented by `HPCToolkit` to generate runtime profile information that can be visualized using `HPCToolkit` graphical viewer. However, the GPU support for `HPCToolkit` is still under development by its authors, and we are continuing to experiment with it. As of this writing (June 9, 2020), it is our understanding that it might not be the best option to obtain other important performance metrics such as floating point operations per second (FLOPS) measurements using `HPCToolkit`.

2.3.2 Using `nvprof`

The next option for obtaining detailed performance profiles using unmodified source code is using NVIDIA's native `nvprof` profiling tool. `nvprof` can profile both CPU and GPU code executions along with detailed FLOP counts and many additional program counters. However, some of the profiling may require certain admin privileges. Also, the deep instruction-level profiling appears to incur a large amount of overhead. With an input matrix size of $16K$, collecting FLOP counts takes approximately 2 hours using `nvprof` for an iteration (out of approximately 15 iterations) of the Mixed Dense-Sparse (MDS) NLPs on `HiOp`. The same operation completes in a matter of seconds when executed normally (without `nvprof`). Therefore, collecting such information for the whole program using `nvprof` is not always feasible on Summit due to job time limitations.

In order to successfully use `nvprof`, an alternative is to restrict the program execution to only part of its initial portions. This way, a partial profile can be gathered, which can adequately represent the whole program. The partial profile can be obtained either by limiting the matrix size or by reducing the number of iterations.

The strength of `nvprof` is in easily gathering the FLOP counts on Summit. For this specific purpose, `nvprof` profiling is the suggested method to follow.

2.3.3 Using `Exa-PAPI++`

Another alternative is via collection of low-level (hardware-level) counters using the Performance Application Programming Interface (PAPI). The latest version intended to be used on exascale systems is called the `Exa-PAPI++` library[1]. `Exa-PAPI++` is designed to provide a generic and portable access to low-level performance counters found across the exascale machines.

An effective use of the `Exa-PAPI++` library requires us to instrument our code base manually to insert the correct invocations of the `Exa-PAPI++` routines. This instrumentation code is often hard to maintain because of the changes to the underlying `HiOp` source code. Therefore, the burden of maintaining the instrumentation code must be taken into account in planning the profiling tasks.

A major strength of `Exa-PAPI++` is its portability across a wide range of machines, as it will be helpful to measure performance when moving across platforms.

2.3.4 Targeted Profiling using Manual Instrumentation

The final option is the use of native support for timing and other critical measures of performance. CUDA routines can be inserted as manual instrumentation around the functions of interest. These function of interest can be shortlisted after gaining initial insights from tools such as `HPCToolkit` and `nvprof`. In general, perusal and study of the source code by an expert may also reveal some

[1]`https://icl.utk.edu/exa-papi/`

important candidate routines (or code fragments) that need to be instrumented for obtaining targeted performance details. This is achieved by inserting timing instrumentation inside the source code and compiling the application. Note that this requires inserting code fragments into the main application source code and hence needs to be done with extreme care (to not inadvertently introduce bugs or otherwise alter the functionality). Another important consideration is the difficulty in keeping the instrumentation up-to-date across software releases. As new versions of `HiOp` and other codes emerge, we need to be careful to update the profiling code also accordingly. One safe method is to create a separate code base for profiling the `HiOp` code, and maintaining it separately from the actual repository to ensure independent investigation.

2.4 Whole Program Profiles

2.4.1 `HiOp` Whole Program Profile

In this task, we performed a whole-program profiling without source code modification. We used `HPCToolkit` and `nvprof` for profiling a Mixed Dense-Sparse NLP problem with `HiOp`. The details of the experiments are given below.

2.4.2 Test Problem Under Consideration

In this experiment we used a Mixed Dense-Sparse NLP problem provided with `HiOp` called `nlpMDS_ex4`. Two important parameters of the program are: i) the size of the dense matrix and ii) the size of the sparse matrix. We measure the performance of MAGMA linear solver for different matrix sizes with `HiOp` application. For this experiment, we set both the matrix size to the same value, and vary them with the following sizes: $[4K, 8K, 15K]$.

2.4.3 Hardware and Software Setup

We used a local Summit proxy node Linux server with a GPU exactly identical to the Summit GPUs. The configurations of the system is given below.

- GPU: `Tesla V100-SXM2-16GB:s_7.0` (same as the Summit GPU)
- CPU: Intel(R) Xeon(R) Silver 4110 CPU
- OS: CentOS Linux 7 (Core)
- System Memory: 256GB
- GPU Memory: 16GB

We used the latest `HiOp` version committed on `Jun 22 15:19:14 2020`. This version includes a different solver method unlike the previous one we benchmarked (Section A.1.1).

2.4.4 Baseline: Normal Execution (without Profiling)

First we execute the test program without any instrumentation other than the CPU timers already in place of the code. There are two timers implemented in the original code base, 1) timer for per iteration of the `HiOp` solver, and 2) timer for the `MAGMA` GPU factorization using `magma_dsytrf()`. We use a matrix size of $8K$ for this experiment. Based on the output of the test program we found that:

- Per iteration takes 6.20 seconds on average.
- GPU factorization takes 2.19 seconds on average (including memory copy between CPU–GPU and computing factorization)

Therefore, the `MAGMA` factorization is responsible for approximately 35% of the time spent on per iteration of `HiOp` Solver. Note that the `MAGMA` GPU time includes the memory copy operation between CPU and GPU memory. In this version of the `HiOp` GPU solver, the underlying copy operations between CPU and GPU is being done by the `MAGMA` function `magma_dsytrf()` itself, so it is currently not possible to decompose time for memory copying and matrix computing without modifying `MAGMA` source code or using other profiler.

This timing from normal (un-instrumented) execution provides a baseline of GPU usage that can be analyzed further in the profiled execution reported in the next sections.

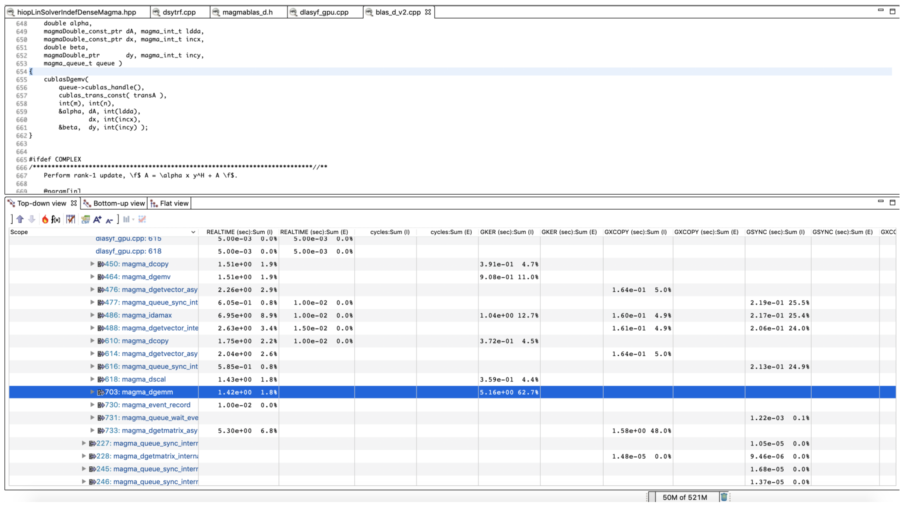
2.4.5 Graphical Profiling and Visualization using `HPCToolkit`

To study how the whole program is spending time in individual functions, we used `HPCToolkit`, a tool to measure performance and profiling for HPC applications. `HPCToolkit` can be used to identify which functions are taking the most amount of time and the call hierarchy using a visual interface. This tool can profile CPUs as well as GPUs for a basic set of metrics. We used a matrix size of $8K$ in this experiment with `HPCToolkit`. `HPCToolkit` offers two GUI tools to visualize the profiling results: 1) `HPCViewer`, and 2) `HPCTraceViewer`. We used both of them to analyze different aspects of the application.

`HPCViewer` presents a hierarchical call graph along with the source code from which the functions were called. It also provides an aggregated timing, processor cycles, and memory information for many common metrics. Due to the profiling overhead, the timing information is not entirely reliable for benchmark, rather it provides a qualitative information to compare the percentage of time being spent on different functions. Figure 12 presents the output of `HPCViewer` for this experiment. As the figure shows, out of the total time spent on the GPU:

- 9.2% (4.7% + 4.5%) is spent on `magma_dcopy()`,
- 11% is spent on `magma_dgemv()`,
- 12.7% is spent on `magma_idamax()`,
- 4.4% is spent on `magma_dscal()`, and
- 62.7% is spent on `magma_dgemm()`.

Therefore, most of the computation are being done with the `MAGMA` matrix multiplication as found here.



```
    double alpha,
    magmaDouble_const_ptr dA, magma_int_t ldda,
    magmaDouble_const_ptr dx, magma_int_t incx,
    double beta,
    magmaDouble_ptr       dy, magma_int_t incy,
    magma_queue_t queue )
{
    cublasDgemv(
        queue->cublas_handle(),
        cublas_trans_const( transA ),
        int(m), int(n),
        &alpha, dA, int(ldda),
                dx, int(incx),
        &beta,  dy, int(incy) );
}


#ifdef COMPLEX
/***************************************************************************//**
    Perform rank-1 update, \f$ A = \alpha x y^H + A \f$.

    @param[in]
```

Scope	REALTIME (sec):Sum (I)	REALTIME (sec):Sum (E)	cycles:Sum (I)	cycles:Sum (E)	GKER (sec):Sum (I)	GKER (sec):Sum (E)	GXCOPY (sec):Sum (I)	GXCOPY (sec):Sum (E)	GSYNC (sec):Sum (I)	GSYNC (sec):Sum (E)	GXCC
dlasyf_gpu.cpp: 615	5.00e-03 0.0%	5.00e-03 0.0%									
dlasyf_gpu.cpp: 618	5.00e-03 0.0%	5.00e-03 0.0%									
450: magma_dcopy	1.51e+00 1.9%				3.91e-01 4.7%						
464: magma_dgemv	1.51e+00 1.9%				9.08e-01 11.0%						
476: magma_dgetvector_asy	2.26e+00 2.9%						1.64e-01 5.0%				
477: magma_queue_sync_int	6.05e-01 0.8%	1.00e-02 0.0%							2.19e-01 25.5%		
486: magma_idamax	6.95e+00 8.9%	1.00e-02 0.0%			1.04e+00 12.7%		1.60e-01 4.9%		2.17e-01 25.4%		
488: magma_dgetvector_inte	2.63e+00 3.4%	1.50e-02 0.0%					1.61e-01 4.9%		2.06e-01 24.0%		
610: magma_dcopy	1.75e+00 2.2%	1.00e-02 0.0%			3.72e-01 4.5%						
614: magma_dgetvector_asy	2.04e+00 2.6%						1.64e-01 5.0%				
616: magma_queue_sync_int	5.85e-01 0.8%								2.13e-01 24.9%		
618: magma_dscal	1.43e+00 1.8%				3.59e-01 4.4%						
703: magma_dgemm	1.42e+00 1.8%				5.16e+00 62.7%						
730: magma_event_record	1.00e-02 0.0%										
731: magma_queue_wait_eve									1.22e-03 0.1%		
733: magma_dgetmatrix_asy	5.30e+00 6.8%						1.58e+00 48.0%				
227: magma_queue_sync_interr									1.05e-05 0.0%		
228: magma_dgetmatrix_interna							1.48e-05 0.0%		9.46e-06 0.0%		
245: magma_queue_sync_interr									1.68e-05 0.0%		
246: magma_queue_sync_interr									1.37e-05 0.0%		

Figure 1: `HPCToolkit` Visual Call-Graph Results via `HPCViewer`

HPCTraceViewer presents a timeline view of the application along with the call depth. The interface is shown in Figure 13. As shown in the figure, there is a main CPU thread, and multiple GPU streams on different time periods. When executing HiOp with GPU support, it uses MAGMA underneath. MAGMA creates a new stream every time a new BLAS function is executed. Therefore, it is showing multiple steams of GPU usage in this figure. Incidentally, the MAGMA function (magma_dsytrf()) is called once in each iteration of HiOp solver. Therefore, the number of streams also corresponds to the number of HiOp iterations.

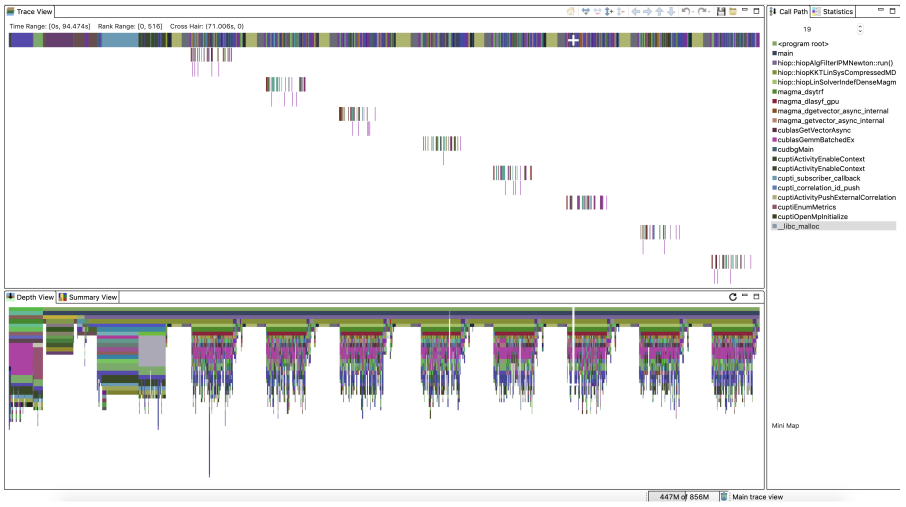

Trace View
Time Range: [0s, 94.474s]   Rank Range: [0, 516]   Cross Hair: (71.006s, 0)
Depth View
Summary View
Call Path
Statistics
19
<program root>
main
hiop::hiopAlgFilterIPMNewton::run()
hiop::hiopKKTLinSysCompressedMD
hiop::hiopLinSolverIndefDenseMagm
magma_dsytrf
magma_dlasyf_gpu
magma_dgetvector_async_internal
magma_getvector_async_internal
cublasGetVectorAsync
cublasGemmBatchedEx
cudbgMain
cuptiActivityEnableContext
cuptiActivityEnableContext
cupti_subscriber_callback
cupti_correlation_id_push
cuptiActivityPushExternalCorrelation
cuptiEnumMetrics
cuptiOpenMpInitialize
__libc_malloc
Mini Map
447M of 856M
Main trace view

Figure 2: HPCToolkit Results via HPCTraceViewer

One thing to note regarding the profiling of HPCToolkit for performance optimization is that the absolute timing might not be the actual run time. Therefore, we only take qualitative information from the HPCToolkit output, such as the percentage of time a function is taking with regard to the total run time. We do not rely on absolute time measures using this tool because it has significant run time overhead.

2.4.6 Profiling with nvprof and NVIDIA Visual Profiler

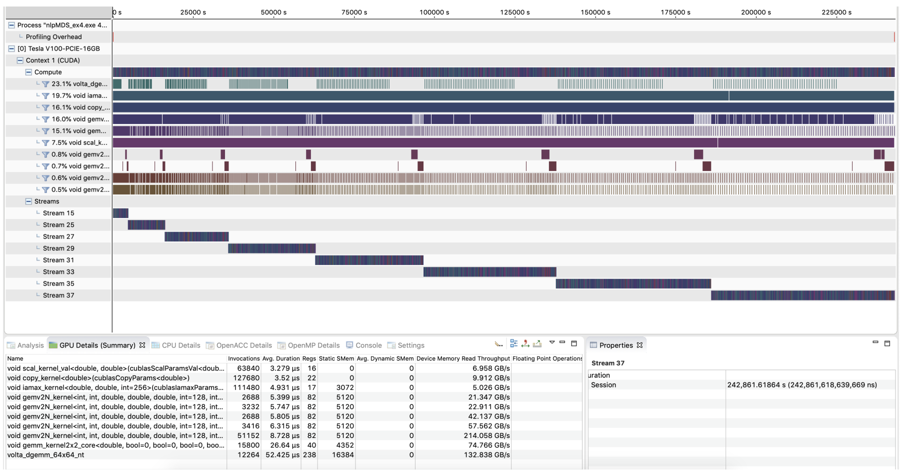
Although the HPCToolkit gives an idea of which functions takes the most of time, it fails to deliver some key metrics such as FLOP count and rates, DRAM throughput, achieved occupancy etc. To get those metrics, we use the native nvprof from NVIDIA. As profiling a compute intensive operation takes a long amount of time, we used a smaller matrix size of $4K$ in this experiment. nvprof took almost 70 hours to collect the desired metrics with only 8 iterations of HiOp solver. We used the command below to collect the profile information. The profiled data is visualized with NVIDIA Visual Profiler as shown in Figure 14.

```
#nvprof command to collect profiling metrics
nvprof --metrics flop_dp_efficiency,flop_count_dp,
    dram_read_transactions,dram_write_transactions
    nlpMDS_ex4.exe 4000 4000
```

As noted above, HiOp uses the MAGMA function magma_dsytrf() for computing the factorization on the GPU. Internally, this function uses 10 GPU kernels to compute the factorization as shown in Table 2.

Table 1: `nvprof` summarized metrics

Name	Device Memory Read Transactions	Device Memory Write Transactions	Device Memory Read Throughput (bytes/sec)	Device Memory Write Throughput (bytes/sec)	Achieved Occupancy	Floating Point Operations	Duration (ns)	FLOP Efficiency	TFLOP /Second
copy_kernel	-	-	9,976,345,713	15,678,824,305	0.00%	-	449512642	0.00	0.000
iamax_kernel	86,373,871	63,539,117	5,314,104,443	3,860,581,411	11.57%	257,117,040	549804765	0.00	0.000
scal_kernel_val	45,531,534	92,077,501	6,872,494,175	13,826,066,355	17.54%	256,158,000	209385186	0.01	0.001
gemv2N_kernel	9,682,216	9,485,604	21,241,064,649	20,529,848,816	6.20%	87,472,904	14513576	0.06	0.006
gemv2N_kernel	20,548,600	16,649,240	41,609,580,853	33,230,076,340	6.88%	197,257,464	15604941	0.13	0.013
gemv2N_kernel	2,986,718,025	1,471,900,154	196,218,593,312	98,490,468,488	11.21%	25,825,263,608	446489014	0.58	0.058
volta_dgemm_64x64_nt	2,669,009,636	1,732,499,537	132,133,738,203	85,900,388,113	5.25%	1,380,942,151,680	642948153	30.26	2.148
gemm_kernel2x2_core	983,471,632	1,182,388,381	69,774,480,330	85,642,244,738	17.99%	373,082,166,264	420922807	11.42	0.886
gemv2N_kernel	13,301,012	18,393,928	23,483,564,929	32,339,137,261	6.90%	159,905,480	18576704	0.08	0.009
gemv2N_kernel	38,805,779	34,946,831	60,250,954,921	53,301,753,535	6.20%	348,685,608	21572472	0.16	0.016



Name	Invocations	Avg. Duration	Regs	Static SMem	Avg. Dynamic SMem	Device Memory Read Throughput	Floating Point Operations
void scal_kernel_val<double, double>(cublasScalParamsVal<doub...	63840	3.279 µs	16	0	0	6.958 GB/s	
void copy_kernel<double>(cublasCopyParams<double>)	127680	3.52 µs	22	0	0	9.912 GB/s	
void iamax_kernel<double, double, int=256>(cublasIamaxParams...	111480	4.931 µs	17	3072	0	5.026 GB/s	
void gemv2N_kernel<int, int, double, double, double, int=128, int...	2688	5.399 µs	82	5120	0	21.347 GB/s	
void gemv2N_kernel<int, int, double, double, double, int=128, int...	3232	5.747 µs	82	5120	0	22.911 GB/s	
void gemv2N_kernel<int, int, double, double, double, int=128, int...	2688	5.805 µs	82	5120	0	42.137 GB/s	
void gemv2N_kernel<int, int, double, double, double, int=128, int...	3416	6.315 µs	82	5120	0	57.562 GB/s	
void gemv2N_kernel<int, int, double, double, double, int=128, int...	51152	8.728 µs	82	5120	0	214.058 GB/s	
void gemm_kernel2x2_core<double, bool=0, bool=0, bool=0, boo...	15800	26.64 µs	40	4352	0	74.766 GB/s	
volta_dgemm_64x64_nt	12264	52.425 µs	238	16384	0	132.838 GB/s	

Stream 37

uration	
Session	242,861.61864 s (242,861,618,639,669 ns)

Figure 3: `nvprof` profile visualization using NVIDIA Visual Profiler. Note that although the application time for each `HiOp` iteration is almost identical across iterations, the profiler takes more time per iteration to perform the profiling. Hence, the blue bars showing the time per stream (iteration) appear to be longer as the number of iterations increases.

2.5 Manual Instrumentation

* `nvtx` does not seem to provide programmatic interface to query FLOP metrics at run time.
  * We have a good coverage of the most computationally intensive portions of the linear algebra portions, including

3 Performance Units

FLOP rating at the linear algebra level (e.g., efficiency 70% of peak)
FLOP rating at the whole program level (e.g., efficiency 12% of peak)

4 Dense Solver Performance

4.0.1 Test Problem Under Consideration

In this experiment we used a Mixed Dense-Sparse NLP problem provided with `HiOp` called `nlpMDS_ex4`. Two important parameters of the program are: i) the size of the dense matrix and ii) the size of the sparse matrix. We measure the performance of MAGMA linear solver for increasing values of these matrix sizes with `HiOp` application. For this experiment, we set both the matrix size to the same value, and vary them with the following sizes: $[2K, 4K, 8K, 12K, 16K, 20K, 22K]$.

4.0.2 Hardware Setup

We used two systems to capture the performance of the linear solver performance. One is a Summit supercomputer node and the other is a local Linux server with a GPU exactly identical to the Summit GPUs. The configurations of the both systems are given below.

Summit
- GPU: `Tesla V100-SXM2-16GB:s_7.0`
- CPU: IBM POWER9

- OS: Red Hat Enterprise Linux 7
- Memory: 512GB per node

Local Summit Proxy Node Linux Server

- GPU: `Tesla V100-SXM2-16GB:s_7.0` (same as Summit GPU)
- CPU: Intel(R) Xeon(R) Silver 4110 CPU
- OS: CentOS Linux 7 (Core)
- Memory: 256GB

4.0.3 Results

We found that linear algebra so far constitutes 18% of total run time (per iteration) after the functionality has been ported to GPU using the `MAGMA` linear algebra solver (see Section D.1.1).

Figure 6 shows the GPU-time required to transfer data from CPU to GPU memory (and vice versa) and the GPU-time required to compute the matrix computation (factorization) for the linear solver on the two platforms. As seen in Figure 6, the GPU-time for computation (which is essentially the `dsytrs()` routine that is the most computationally intensive portion) increases with increasing input sizes. The complexity of the most intensive operation is $O(rN^2)$, where N is the size of the matrix and r is the number of right hand sides for the sparse matrix solution operation [2].

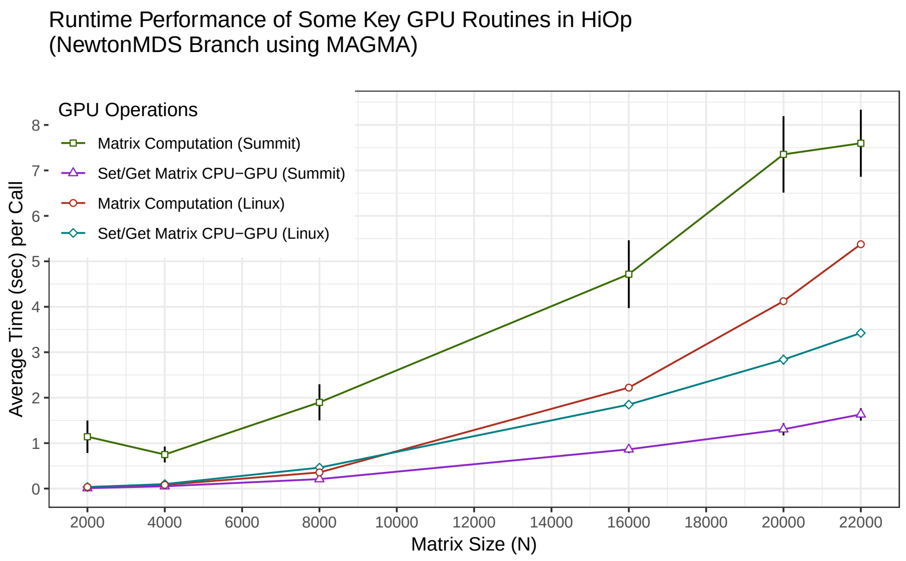


Figure 4: Average GPU-Time (seconds) per BLAS calls

The Summit runs exhibit more variability in the run time than observed on the Linux server. Also, the runtime is significantly larger on Summit than on the Linux server. On the other hand, Summit exhibits faster data transfer capabilities than the Linux server.

We also compared the run time of the computation with that of data transfer. This is shown in Figure 7. As noted previously, it is also seen in this plot that the computation time significantly dominates the communication time on Summit, which exhibits high computation time and significantly less communication time. As the matrix size is increased, the dominance is becomes less pronounced on Summit. In contrast, the same dominance actually increases on the Linux server. However, the computation time remains the most significant source of the time spent on the GPU.

[2] `https://www.nag.co.uk/numeric/fl/nagdoc_latest/html/f07/f07mef.html`

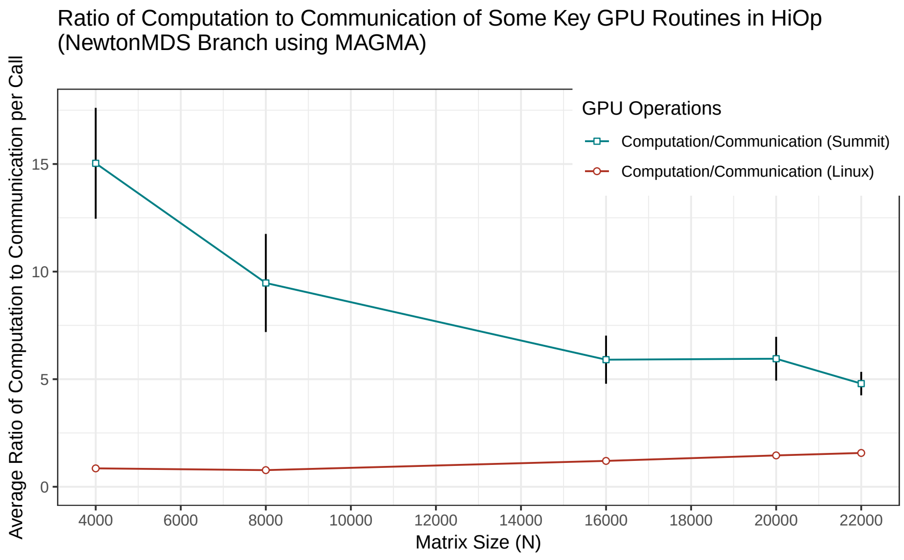


Figure 5: Ratio of Computation to Data Transfer

4.1 Comparison with CPU-based Solvers

Compare `MAGMA` performance gain on GPU relative to Intel `Intel MKL` and `Open BLAS` on CPU.

A JIRA ADSE22-145: Close-out Report

- **Points**: 25 pts
- **Sub-stories**: 3
- **JIRA Description**: Evaluate dense linear solvers on GPU

A.1 JIRA Substory 1: `MAGMA` Linear Solver Performance

- **Points**: 9 pts
- **Date Due**: May 31, 2020
- **Date Completed**: May 31, 2020
- **Authors**: Kalyan Perumalla and Maksudul Alam
- **JIRA Description**: Test `MAGMA` direct linear solver performance, with increasing matrix sizes with `HiOp` proxy apps.

A.1.1 Test Problem Under Consideration

In this experiment we used a Mixed Dense-Sparse NLP problem provided with `HiOp` called `nlpMDS_ex4`. Two important parameters of the program are: i) the size of the dense matrix and ii) the size of the sparse matrix. We measure the performance of MAGMA linear solver for increasing values of these matrix sizes with `HiOp` application. For this experiment, we set both the matrix size to the same value, and vary them with the following sizes: $[2K, 4K, 8K, 12K, 16K, 20K, 22K]$.

A.1.2 Hardware Setup

We used two systems to capture the performance of the linear solver performance. One is a Summit supercomputer node and the other is a local Linux server with a GPU exactly identical to the Summit GPUs. The configurations of the both systems are given below.

Summit
- GPU: `Tesla V100-SXM2-16GB:s_7.0`
- CPU: IBM POWER9
- OS: Red Hat Enterprise Linux 7
- Memory: 512GB per node

Local Summit Proxy Node Linux Server
- GPU: `Tesla V100-SXM2-16GB:s_7.0` (same as Summit GPU)
- CPU: Intel(R) Xeon(R) Silver 4110 CPU
- OS: CentOS Linux 7 (Core)
- Memory: 256GB

A.1.3 Results

We found that linear algebra so far constitutes 18% of total run time (per iteration) after the functionality has been ported to GPU using the `MAGMA` linear algebra solver (see Section D.1.1).

Figure 6 shows the GPU-time required to transfer data from CPU to GPU memory (and vice versa) and the GPU-time required to compute the matrix computation (factorization) for the linear solver on the two platforms. As seen in Figure 6, the GPU-time for computation (which is essentially the `dsytrs()` routine that is the most computationally intensive portion) increases with increasing input sizes. The complexity of the most intensive operation is $O(rN^2)$, where N is the size of the matrix and r is the number of right hand sides for the sparse matrix solution operation [3].

The Summit runs exhibit more variability in the run time than observed on the Linux server. Also, the runtime is significantly larger on Summit than on the Linux server. On the other hand, Summit exhibits faster data transfer capabilities than the Linux server.

We also compared the run time of the computation with that of data transfer. This is shown in Figure 7. As noted previously, it is also seen in this plot that the computation time significantly

[3] `https://www.nag.co.uk/numeric/fl/nagdoc_latest/html/f07/f07mef.html`

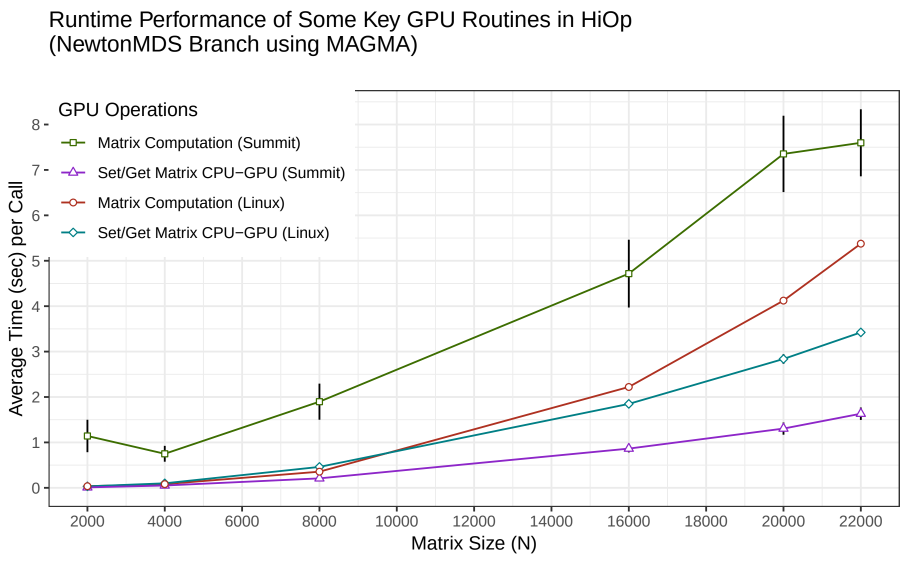


Figure 6: Average GPU-Time (seconds) per BLAS calls

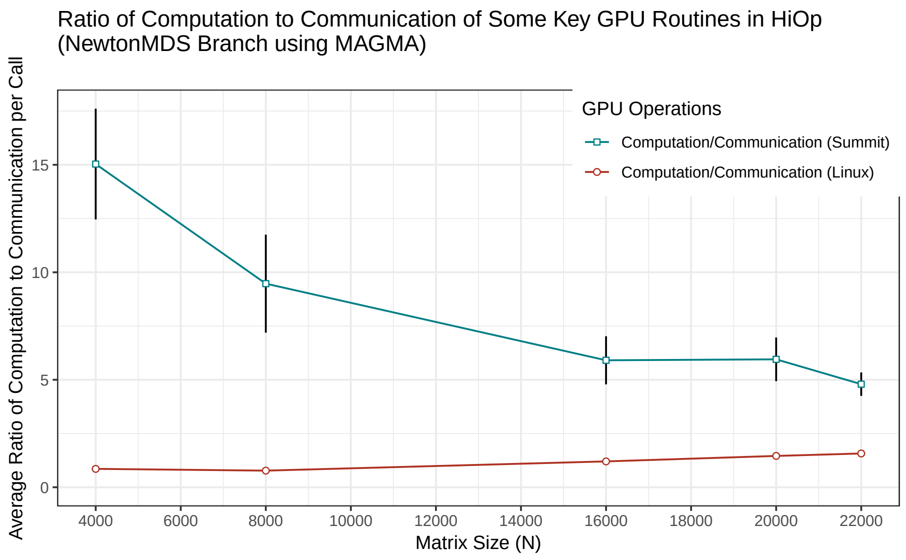


Figure 7: Ratio of Computation to Data Transfer

dominates the communication time on Summit, which exhibits high computation time and significantly less communication time. As the matrix size is increased, the dominance is becomes less pronounced on Summit. In contrast, the same dominance actually increases on the Linux server. However, the computation time remains the most significant source of the time spent on the GPU.

A.2 JIRA Substory 2: Memory Limit

- **Points**: 7 pts
- **Date Due**: July 15, 2020
- **Date Completed**: July 15, 2020
- **Authors**: Kalyan Perumalla and Maksudul Alam
- **JIRA Description**: Assess size limits and tradeoffs due to transfer time and memory capacities of GPU.

A.2.1 GPU Memory Usage

In this task, we performed the GPU memory requirement analysis of `HiOp` solver. Our analysis derives the maximum matrix size that can be accommodated within one GPU memory.

A.2.2 Hardware and Software Setup

We used our Summit proxy node, which is a Linux server with a GPU exactly identical to the Summit GPUs. The configurations of the system is given below.

- GPU: `Tesla V100-SXM2-16GB:s_7.0` (same as the Summit GPU)
- CPU: Intel(R) Xeon(R) Silver 4110 CPU
- OS: CentOS Linux 7 (Core)
- GPU Memory: 16GB
- CPU Memory: 256GB

A.2.3 GPU Memory Requirement Analysis

In this experiment we used a Mixed Dense-Sparse NLP problem provided with `HiOp` called `nlpMDS_ex4`. Two important parameters of the program are:

- The size of the dense matrix ($N_D \times N_D$) and
- The size of the sparse matrix ($N_S \times N_S$).

For any combination of N_D and N_S, `HiOp` uses an intermediate matrix size of $N \times N$, where $N = N_D + N_S$, denotes the aggregated matrix dimension. Therefore, the required amount of memory is primarily dependent on N.

The `HiOp` solver stores the matrix and a rhs vector in the device memory. Storing the matrix takes $N^2 \times d$ bytes where $d = 8$ is the size of data type `double`. Storing the rhs vector requires $N \times d$ bytes [4].

Further, in the `HiOp` solver, the core GPU computation is done by calling the function `magma_dsytrf()`. The function creates two matrices of the sizes $(N \times ldda)$ and $ldda \times (1 + nb)$, where $ldda$ is the aligned value of N by 32 and nb is computed from the function `magma_get_dsytrf_nb()` [5]. For the system under consideration, we have $nb = 96$. Note that those two matrices are being used in the functions `cublasDgemv()` and `cublasDgemm()` to compute the factorization. Therefore the theoretical amount of required memory (M) is defined by:

$$M = (N^2 + N + N \times ldda + (1 + nb) \times ldda) \times d.$$

[4] See class `hiopLinSolverIndefDenseMagmaDev` in file `src/LinAlg/hiopLinSolverIndefDenseMagma.hpp`

[5] See file `dsytrf.cpp` in MAGMA source folder

A.2.4 Theoretical versus Practical Limit

Based on our hardware setup, $M = 16\text{GB}$, $d = 8$, $nb = 96$, which gives us the maximum value of $N = 32736$. However, due to various software overheads, this limit is not achieved in practice. Nevertheless, we have been able to instantiate matrix sizes very close to that theoretical limit, reaching up to $N = 31800$ without memory overflow issues (which result in segmentation faults or other abnormal termination conditions).

The reason for not realizing the theoretical maximum is due to the inherent bookkeeping memory needed by `CUBLAS`. Also, memory alignment considerations also reduce the maximum matrix size. For instance, the internal `CUBLAS` blocks need to be aligned along memory blocks that are multiples of 32. Additional memory is also demarcated by `CUBLAS` and `MAGMA` for miscellaneous terms in their formula to determine the memory block for allocation.

A.3 JIRA Substory 3: MAGMA Performance Gain on GPU

- **Points**: 9 pts
- **Date Due**: September 30, 2020
- **Date Completed**: September 28, 2020
- **Authors**: Kalyan Perumalla and Maksudul Alam
- **JIRA Description**: Compare MAGMA performance gain on GPU relative to `Intel MKL` and `Open BLAS` on CPU.

A.3.1 Performance Summary of `HiOp` for Different BLAS Implementation

We conducted experiments to study which functions are taking the most amount of computational time during different BLAS implementation in `HiOp`. We used nlpMDS_ex4 to study the performance.

A.3.2 H/W Setup

We used Tesla V100 GPU with 16 GB RAM, Intel(R) Xeon(R) Silver 4110 CPU, with 256GB of RAM. We used CentOS Linux 7 (Core) as the operating system.

A.3.3 BLAS

We used the following BLAS implementations: 1. OpenBLAS (CPU) 2. Intel MKL (CPU) 3. MAGMA (GPU based, uses CUBLAS underlying)

A.3.4 Timing Info

We analyzed the code and put manual wall-clock timers on different parts of the functions to measure elapsed time for each function for each `HiOp` iteration. As each iteration of `HiOp` solver is almost the similar in behavior, we present the average timing info per iteration for each experiment conducted.

A.3.5 Solver Setup

We used nlpMDS_ex4 program to measure the performance. The program takes two inputs, 1) the size of the dense matrix, and 2) the size of the sparse matrix. For this experiment both size of the dense and sparse matrix is set as 8000.

A.3.6 Summary of Performance Results

As observed from the experiments, OpenBLAS takes 72.95 seconds, Intel MKL takes 18.56 seconds, and MAGMA takes 4.49 seconds per iteration. In essence MAGMA is 16X and 4X faster than OpenBLAS and Intel MKL respectively. The core of the operation depends on the computation of factorization of a symmetric matrix (BLAS function DSYTRF that computes the factorization of a real symmetric matrix A using the Bunch-Kaufman diagonal pivoting method).

Computing the factorization takes 67.92 seconds (93% of total time) for OpenBLAS, 13.61 seconds (73% of total time) for Intel MKL and 0.81 seconds (18% of total time) for MAGMA. Therefore, factorization seems to be quite faster on the GPU. The computation on the GPU basically consists of two parts, 1) copying data from CPU to GPU and back forth, and 2) actual computation of GPU factorization. Of the 0.81 seconds, about 0.45 is spent on memory copying, and the remaining 0.36 seconds are used for actual computation.

A.3.7 Detailed Hierarchical Timing Info for OpenBLAS

Below, we present the hierarchical call lists of major function per iteration. The most significant functions are highlighted in bold.

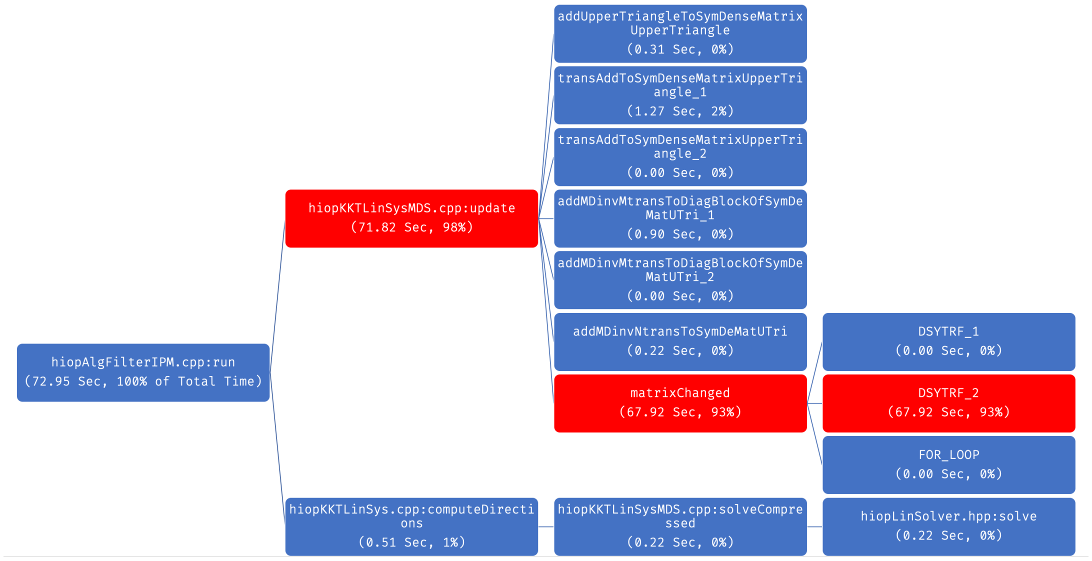

addUpperTriangleToSymDenseMatrixUpperTriangle (0.31 Sec, 0%)
transAddToSymDenseMatrixUpperTriangle_1 (1.27 Sec, 2%)
transAddToSymDenseMatrixUpperTriangle_2 (0.00 Sec, 0%)
hiopKKTLinSysMDS.cpp:update (71.82 Sec, 98%)
addMDinvMtransToDiagBlockOfSymDeMatUTri_1 (0.90 Sec, 0%)
addMDinvMtransToDiagBlockOfSymDeMatUTri_2 (0.00 Sec, 0%)
addMDinvNtransToSymDeMatUTri (0.22 Sec, 0%)
DSYTRF_1 (0.00 Sec, 0%)
hiopAlgFilterIPM.cpp:run (72.95 Sec, 100% of Total Time)
matrixChanged (67.92 Sec, 93%)
DSYTRF_2 (67.92 Sec, 93%)
FOR_LOOP (0.00 Sec, 0%)
hiopKKTLinSys.cpp:computeDirections (0.51 Sec, 1%)
hiopKKTLinSysMDS.cpp:solveCompressed (0.22 Sec, 0%)
hiopLinSolver.hpp:solve (0.22 Sec, 0%)

Figure 8: `HiOp` Exploration

A.3.8 Detailed Hierarchical Timing Info for Intel MKL

Below, we present the hierarchical call lists of major function per iteration. The most significant functions are highlighted in bold.

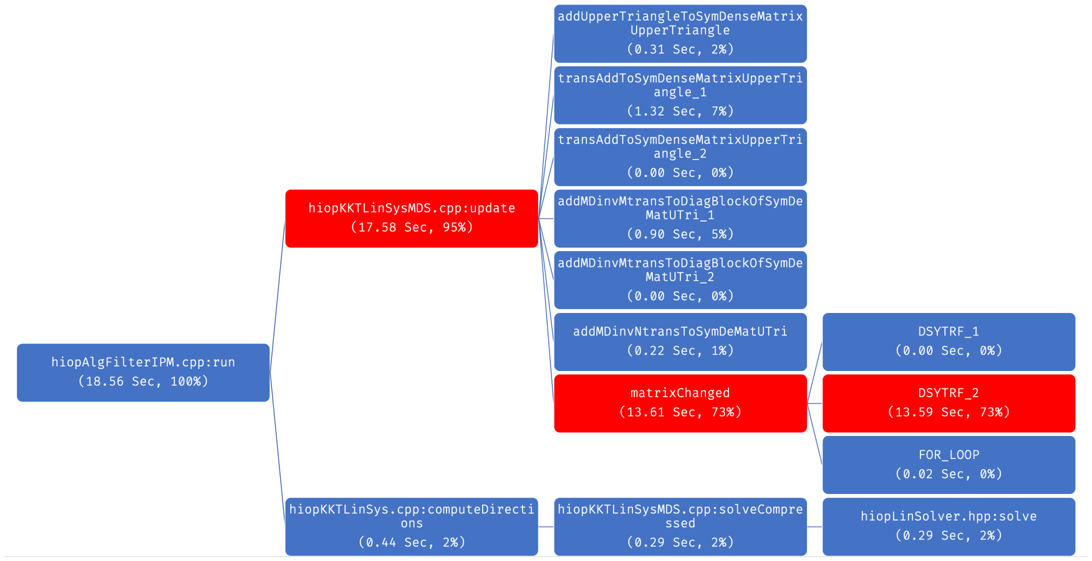

addUpperTriangleToSymDenseMatrixUpperTriangle (0.31 Sec, 2%)
transAddToSymDenseMatrixUpperTriangle_1 (1.32 Sec, 7%)
transAddToSymDenseMatrixUpperTriangle_2 (0.00 Sec, 0%)
hiopKKTLinSysMDS.cpp:update (17.58 Sec, 95%)
addMDinvMtransToDiagBlockOfSymDeMatUTri_1 (0.90 Sec, 5%)
addMDinvMtransToDiagBlockOfSymDeMatUTri_2 (0.00 Sec, 0%)
addMDinvNtransToSymDeMatUTri (0.22 Sec, 1%)
DSYTRF_1 (0.00 Sec, 0%)
hiopAlgFilterIPM.cpp:run (18.56 Sec, 100%)
matrixChanged (13.61 Sec, 73%)
DSYTRF_2 (13.59 Sec, 73%)
FOR_LOOP (0.02 Sec, 0%)
hiopKKTLinSys.cpp:computeDirections (0.44 Sec, 2%)
hiopKKTLinSysMDS.cpp:solveCompressed (0.29 Sec, 2%)
hiopLinSolver.hpp:solve (0.29 Sec, 2%)

Figure 9: `HiOp` Exploration

A.3.9 Detailed Hierarchical Timing Info for MAGMA

Below, we present the hierarchical call lists of major function per iteration. The most significant functions are highlighted in bold.

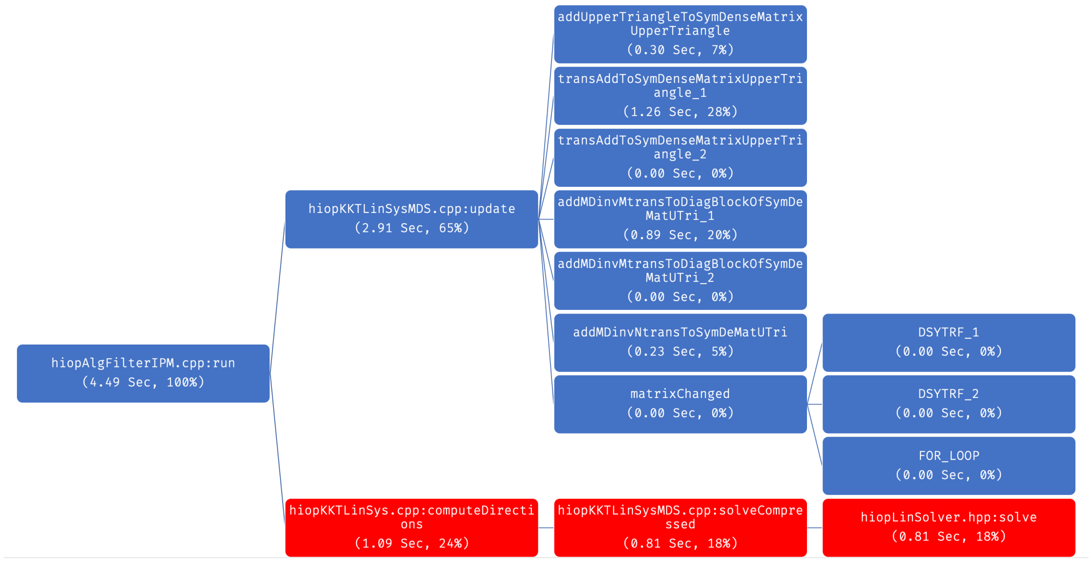

hiopAlgFilterIPM.cpp:run
(4.49 Sec, 100%)
hiopKKTLinSysMDS.cpp:update
(2.91 Sec, 65%)
hiopKKTLinSys.cpp:computeDirections
(1.09 Sec, 24%)
addUpperTriangleToSymDenseMatrixUpperTriangle
(0.30 Sec, 7%)
transAddToSymDenseMatrixUpperTriangle_1
(1.26 Sec, 28%)
transAddToSymDenseMatrixUpperTriangle_2
(0.00 Sec, 0%)
addMDinvMtransToDiagBlockOfSymDeMatUTri_1
(0.89 Sec, 20%)
addMDinvMtransToDiagBlockOfSymDeMatUTri_2
(0.00 Sec, 0%)
addMDinvNtransToSymDeMatUTri
(0.23 Sec, 5%)
matrixChanged
(0.00 Sec, 0%)
hiopKKTLinSysMDS.cpp:solveCompressed
(0.81 Sec, 18%)
DSYTRF_1
(0.00 Sec, 0%)
DSYTRF_2
(0.00 Sec, 0%)
FOR_LOOP
(0.00 Sec, 0%)
hiopLinSolver.hpp:solve
(0.81 Sec, 18%)

Figure 10: `HiOp` Exploration

B JIRA ADSE22-193: Close-out Report

- **Points**: 5 pts
- **Date Due**: June 15, 2020
- **Date Completed**: June 15, 2020
- **Authors**: Kalyan Perumalla and Maksudul Alam
- **JIRA Description**: Define profiling units

Units to be defined in linear solver, input data instantiation, and optimization solver.

- Identify major units of components, software.
- Identify units of measure (FLOPS, occupancy on GPU, roofline plots, etc.).
- Iterate as needed when profile information is available.
- Coordinate the activity with `HiOp`, `SCOPFlow` and relevant proxy app tasks.

B.1 Major Units of Components

The most computationally intensive operations performed in `HiOp` are the matrix operations. There are a variety of matrix operations used for various purposes, most of which are performed on the CPU. Among these operations, the most intensive ones are concerned with matrix factorization. It is these operations that are initially ported to the GPU. Therefore, the GPU port of the `HiOp` currently executes the factorization part on the GPU.

Based on the test problem (a mixed dense-sparse NLP called `nlpMDS_ex4`), we determined that the following matrix operations consume the most amount of computational time:

1. `addUpperTriangleToSymDenseMatrixUpperTriangle()`
2. `transAddToSymDenseMatrixUpperTriangle()`
3. `addMDinvMtransToDiagBlockOfSymDeMatUTri()`
4. `addMDinvNtransToSymDeMatUTri()`
5. Other routines that invoke `DSYTRF()` (a BLAS Routine)

Of these matrix routines, only the `DSYTRF()` routine has been ported with MAGMA on the GPU. In typical scenarios, `DSYTRF()` takes about 18% of the wall clock time per `HiOp` iteration. The time includes both the matrix computation as well as the time for data transfer to and from GPU. In the preceding list, matrix Operations other than `DSYTRF()` take about 65% of the time per `HiOp` iteration. The rest 17% are consumed by the rest of `HiOp` computation.

B.2 Profiling Metrics

We suggest the use of the following profiling metrics in evaluating the performance of `HiOp`:

1. GPU Occupancy
2. GPU Computational Efficiency and Memory Utilization
3. Achieved Host-GPU Memory Transfer Bandwidth
4. GPU Memory Throughput
5. FLOP Count and Rate
6. Roofline Analysis.

Metrics 1-5 are described in this section. The roofline analysis is described in the next section.

B.2.1 GPU Occupancy

In the GPU context, the occupancy denotes the ratio of active warps (a group of threads) on a Streaming Multiprocessor (SM) to the maximum number of active warps supported by the SM. As each warp begins and ends, occupancy can change and can be different for each SM. Typically, low occupancy indicates poor instruction issue efficiency, due to insufficient number of warps to run concurrently on the SMs. Also, if occupancy is beyond a sufficient level, the performance may take a hit due to the reduction in the amount of resources per thread. Therefore, a kernel performance analysis is suggested to check occupancy and observe the effects on kernel execution time at different occupancy levels.

There are two kinds of occupancy that can be measured for a GPU program. The *theoretical occupancy* of a kernel is an upper limit that can be determined as early as at compile time. This is obtained from the kernel's requirement of resources (such as registers) based on the known capabilities of the device SM. The *achieved occupancy* can be determined using profiling tools (such as `nvprof`) that measure the occupancy during the execution of the kernel.

B.2.2 GPU Computational Efficiency and Memory Utilization

The *GPU Computational Efficiency* (also called simply Utilization) measures the percentage of time one or more GPU kernels are running over a sample period. The sample period is between 1/6 seconds and 1 second depending on the card.

This metric can measure the efficiency of the GPU being used by a program. Typically the GPU utilization is probed via GPU driver (such as `nvidia-smi`). Additionally, special API's (such as `NVML`) are also available to access the GPU driver to collect this metric programmatically during profiling.

The *GPU Memory Utilization* measures the percentage of GPU memory being used. Similar to GPU Utilization, this is information is obtained by probing the GPU driver via the driver API.

B.2.3 Achieved Host-GPU Memory Transfer Bandwidth

The Host-GPU bandwidth measures the throughput of memory transfer between the host CPU and the GPU. As the GPU is connected to the system's motherboard via a bus (such as PCI Express), this bandwidth largely depends on how fast the bus (such as PCI bus) is and how many other hardware components are using it. There are also some overheads that are included in the measurements, such as function invocation time and array allocation time. The bandwidth can be measured using a profiler (such as `nvprof`). It can also be obtained by manually instrumenting the portions of the code that invoke memory transfer operations.

B.2.4 GPU Memory Throughput

Operations performed by the GPU involve computation, memory transfers (reads and writes), or combinations of both. Some operations do very little computation with each memory element. They are, therefore, dominated by the time taken to fetch/write the data from/to memory to read/write. Functions such as setting all elements to specific constants (such as ones, zeros, etc.) only write their output, whereas functions like transpose both read and write but do no computation. Even simple operators such as addition, deletion, multiplication, etc. perform such little computation per element that they may be bound only by the memory access speed. Typically, this measurement is available via native GPU profiler (`nvprof`).

For these reasons, the memory throughput achieved by the program becomes an important measure of runtime performance. For operations where the amount of computation performed per element read from or written to memory is low, the memory throughput is important.

B.2.5 FLOPS

For operations where the number of floating-point computations performed per element read from or written to memory is high, the memory speed is much less important. These operations are said to have high "arithmetic intensity". In this case the number of operations and speed of the floating-point units is the limiting factor. The measure of interest is called the number of floating point operations per second, abbreviated as FLOPS, or simply FLOP rate. We measure the FLOP rate and FLOP count as a measure of computational intensity of the program. Note that the FLOP rate can be measured over the entire duration of the program, or over specific segments of the program. In this case, we suggest measuring the FLOP rate using the the amount of time spent in corresponding GPU operations. We can determine FLOP rate in many ways, such as estimating theoretical FLOP operations, using direct observations from the profiler-extracted FLOP counts, or indirectly using the GPU efficiency measurements relative to the published FLOP ratings of peak efficiency. Since the FLOP counts and FLOP rates are often very large numbers, they are usually measured in millions. GFLOP is used as short notation for giga-FLOP, standing for billion floating point operations. Similarly, GFLOP/s is GFLOP per second.

B.3 Roofline Analysis

Additional insight into the key determinants of performance can be obtained via a technique called *roofline analysis* [6]. This type of analysis provides information to evaluate the efficiency of the arithmetic operations in programs such as `HiOp`, which in turn indicate opportunities available towards tuning the performance.

The roofline performance model presents an intuitive visual method for extracting key insightful computational characteristics of the applications and comparing them against the performance bounds of the underlying processor. It can abstract the complexity of non-uniform memory hierarchies to identify the most beneficial optimization technique.

In roofline analysis, we break down the application into core code fragments such as constituent loop nests or kernels. The performance of the core code fragments (such as computation, data movement, and run time) are plotted against the peak capabilities (such as GFLOP/s and bandwidth) of a given processor on an absolute scale. Typically, we plot the peak performance (GFLOP/s) against the arithmetic intensity, which is defined by the ratio of total floating-point operations (FLOPs) executed by the code fragment, to the total data movement (in bytes) required for those FLOPs.

Conceptually, we can measure the upper bound of the performance of the system under investigation using the vendor specifications. Although they provide some insight of the upper bounds, they do not adequately capture a realistic view of the execution environment. The peak performance of the realistic execution environment of the system can be measured by micro-kernels using a tool called Empirical Roofline Toolkit (ERT). This provides a baseline or tighter upper boundary of performance of the underlying system. The actual performance of the system can be measured using profiling tools to determine the arithmetic intensity and the peak GFLOP/s measures. When those measures are plotted as a roofline plot, the plot can provide helpful information regarding the essential nature of the computation (that is, whether it is memory-bound or compute-bound) and the level of performance compared to peak. Based on the plot we can decide whether to further optimize the code or not.

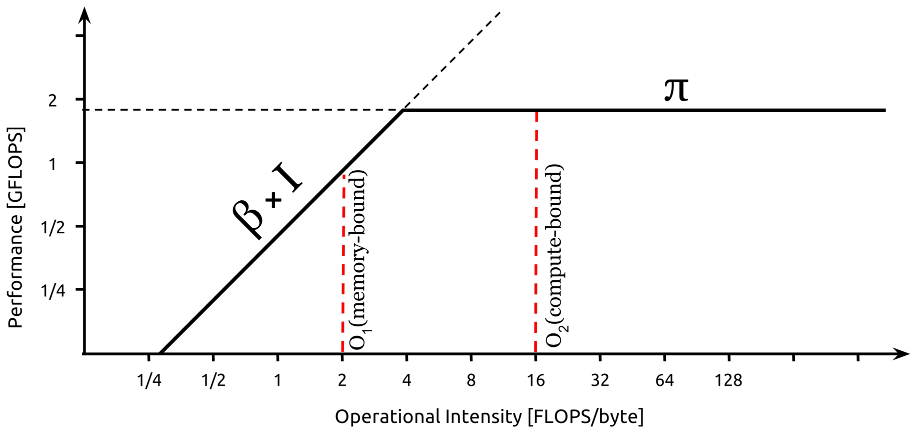


Figure 11: An example of a roofline plot.

[7]

An example of the roofline plot is shown in Figure 11. In this model, we use two parameters, namely, the peak computational throughput (defined as π) and the peak bandwidth (β) of the computing system under consideration. The peak computation throughput is expressed as GFLOP/s,

[6]S. Williams, A. Waterman, D. Patterson, "Roofline: an insightful visual performance model for multicore architectures", Communications of the ACM (CACM), April 2009.

[7]Taken from wikipedia `https://en.wikipedia.org/wiki/Roofline_model`

usually derived from system specification. The peak bandwidth is the DRAM bandwidth, which can be obtained via benchmarking. We next plot the attainable performance (defined as P) using the following equation:

$$P = \min \begin{cases} \pi \\ \beta \times I \end{cases}$$

where I is the arithmetic intensity. As noted previously, the arithmetic intensity is defined as the ratio of computational work to memory access. The point where the attainable performance saturates to the peak level π is called the ridge point. We can analyze the characteristics of a given kernel on the system by plotting a point (with its arithmetic intensity I on the x-axis and the achieved performance P on the y-axis) and drawing a vertical line that hits the roofline curve. If the vertical line is towards the left of the ridge point (line O_1 in Figure 11), the kernel is said to be memory-bound. On the other hand, if the vertical line is towards the right of the ridge point (line O_2 is Figure 11), the kernel is said to be compute-bound.

C JIRA ADSE22-194: Close-out Report

- **Points**: 5 pts
- **Date Due**: June 5, 2020
- **Date Completed**: June 5, 2020
- **Authors**: Kalyan Perumalla and Maksudul Alam
- **JIRA Description**: Profiling strategy for Summit
  This activity will identify what tools are available and which ones are applicable to the software stack and its components.
  1. Evaluate candidate tools (like HPCToolkit) and expand to complement to cover places where more information is needed.
  2. Generate preliminary profiling results from selected tools.
  3. Deliver documentation on what works (more emphasis on capabilities and adequacy, not necessarily focused on exact results for now).

C.1 Devised Profiling Strategy for Summit

We explored different options for profiling tools on the Summit supercomputer. We have identified at least four primary options for profiling: 1) whole program profiling using `HPCToolkit`, 2) whole program profiling with `nvprof`, 3) whole program profiling using `Exa-PAPI++`, and 4) targeted profiling using manual instrumentation.

1. Using `HPCToolkit`
2. Using `nvprof`
3. Using `Exa-PAPI++`
4. Using targeted profiling via manual instrumentation

C.1.1 Using `HPCToolkit`

Summit has a default `HPCToolkit` package installed that does not have GPU support. Therefore, we have obtained the latest `HPCToolkit` version from source and successfully compiled it ourselves on Summit by explicitly adding GPU support. We have been experimenting with this custom-built `HPCToolkit` on Summit and we find that it seems to be a good option to gain a general idea of the program profiles without having to undertake any modification to the source code. In fact, it is possible to use the same binaries as are used for normal execution. The binaries are automatically instrumented by `HPCToolkit` to generate runtime profile information that can be visualized using `HPCToolkit` graphical viewer. However, the GPU support for `HPCToolkit` is still under development by its authors, and we are continuing to experiment with it. As of this writing (June 9, 2020), it is our understanding that it might not be the best option to obtain other important performance metrics such as floating point operations per second (FLOPS) measurements using `HPCToolkit`.

C.1.2 Using `nvprof`

The next option for obtaining detailed performance profiles using unmodified source code is using NVIDIA's native `nvprof` profiling tool. `nvprof` can profile both CPU and GPU code executions along with detailed FLOP counts and many additional program counters. However, some of the profiling may require certain admin privileges. Also, the deep instruction-level profiling appears to incur a large amount of overhead. With an input matrix size of $16K$, collecting FLOP counts takes approximately 2 hours using `nvprof` for an iteration (out of approximately 15 iterations) of the

Mixed Dense-Sparse (MDS) NLPs on `HiOp`. The same operation completes in a matter of seconds when executed normally (without `nvprof`). Therefore, collecting such information for the whole program using `nvprof` is not always feasible on Summit due to job time limitations.

In order to successfully use `nvprof`, an alternative is to restrict the program execution to only part of its initial portions. This way, a partial profile can be gathered, which can adequately represent the whole program. The partial profile can be obtained either by limiting the matrix size or by reducing the number of iterations.

The strength of `nvprof` is in easily gathering the FLOP counts on Summit. For this specific purpose, `nvprof` profiling is the suggested method to follow.

C.1.3 Using `Exa-PAPI++`

Another alternative is via collection of low-level (hardware-level) counters using the Performance Application Programming Interface (PAPI). The latest version intended to be used on exascale systems is called the `Exa-PAPI++` library[8]. `Exa-PAPI++` is designed to provide a generic and portable access to low-level performance counters found across the exascale machines.

An effective use of the `Exa-PAPI++` library requires us to instrument our code base manually to insert the correct invocations of the `Exa-PAPI++` routines. This instrumentation code is often hard to maintain because of the changes to the underlying `HiOp` source code. Therefore, the burden of maintaining the instrumentation code must be taken into account in planning the profiling tasks.

A major strength of `Exa-PAPI++` is its portability across a wide range of machines, as it will be helpful to measure performance when moving across platforms.

C.1.4 Targeted Profiling using Manual Instrumentation

The final option is the use of native support for timing and other critical measures of performance. CUDA routines can be inserted as manual instrumentation around the functions of interest. These function of interest can be shortlisted after gaining initial insights from tools such as `HPCToolkit` and `nvprof`. In general, perusal and study of the source code by an expert may also reveal some important candidate routines (or code fragments) that need to be instrumented for obtaining targeted performance details. This is achieved by inserting timing instrumentation inside the source code and compiling the application. Note that this requires inserting code fragments into the main application source code and hence needs to be done with extreme care (to not inadvertently introduce bugs or otherwise alter the functionality). Another important consideration is the difficulty in keeping the instrumentation up-to-date across software releases. As new versions of `HiOp` and other codes emerge, we need to be careful to update the profiling code also accordingly. One safe method is to create a separate code base for profiling the `HiOp` code, and maintaining it separately from the actual repository to ensure independent investigation.

[8]`https://icl.utk.edu/exa-papi/`

D JIRA ADSE22-245: Close-out Report

- **Points**: 20 pts
- **Sub-stories**: 4
- **JIRA Description**: Based on profiling strategy, deliver preliminary profiling data for performance on Summit or equivalent architecture.

D.1 JIRA Substory 1: Preliminary profiling results for the software stack

- **Points**: 5 pts
- **Date Due**: June 30, 2020
- **Date Completed**: June 30, 2020
- **Authors**: Kalyan Perumalla and Maksudul Alam
- **JIRA Description**: `HiOp` whole program profile.

D.1.1 `HiOp` Whole Program Profile

In this task, we performed a whole-program profiling without source code modification. We used `HPCToolkit` and `nvprof` for profiling a Mixed Dense-Sparse NLP problem with `HiOp`. The details of the experiments are given below.

D.1.2 Test Problem Under Consideration

In this experiment we used a Mixed Dense-Sparse NLP problem provided with `HiOp` called `nlpMDS_ex4`. Two important parameters of the program are: i) the size of the dense matrix and ii) the size of the sparse matrix. We measure the performance of MAGMA linear solver for different matrix sizes with `HiOp` application. For this experiment, we set both the matrix size to the same value, and vary them with the following sizes: $[4K, 8K, 15K]$.

D.1.3 Hardware and Software Setup

We used a local Summit proxy node Linux server with a GPU exactly identical to the Summit GPUs. The configurations of the system is given below.

- GPU: `Tesla V100-SXM2-16GB:s_7.0` (same as the Summit GPU)
- CPU: Intel(R) Xeon(R) Silver 4110 CPU
- OS: CentOS Linux 7 (Core)
- System Memory: 256GB
- GPU Memory: 16GB

We used the latest `HiOp` version committed on `Jun 22 15:19:14 2020`. This version includes a different solver method unlike the previous one we benchmarked (Section A.1.1).

D.1.4 Baseline: Normal Execution (without Profiling)

First we execute the test program without any instrumentation other than the CPU timers already in place of the code. There are two timers implemented in the original code base, 1) timer for per iteration of the `HiOp` solver, and 2) timer for the `MAGMA` GPU factorization using `magma_dsytrf()`. We use a matrix size of $8K$ for this experiment. Based on the output of the test program we found that:

- Per iteration takes 6.20 seconds on average.
- GPU factorization takes 2.19 seconds on average (including memory copy between CPU–GPU and computing factorization)

Therefore, the `MAGMA` factorization is responsible for approximately 35% of the time spent on per iteration of `HiOp` Solver. Note that the `MAGMA` GPU time includes the memory copy operation between CPU and GPU memory. In this version of the `HiOp` GPU solver, the underlying copy operations between CPU and GPU is being done by the `MAGMA` function `magma_dsytrf()` itself, so it is currently not possible to decompose time for memory copying and matrix computing without modifying `MAGMA` source code or using other profiler.

This timing from normal (un-instrumented) execution provides a baseline of GPU usage that can be analyzed further in the profiled execution reported in the next sections.

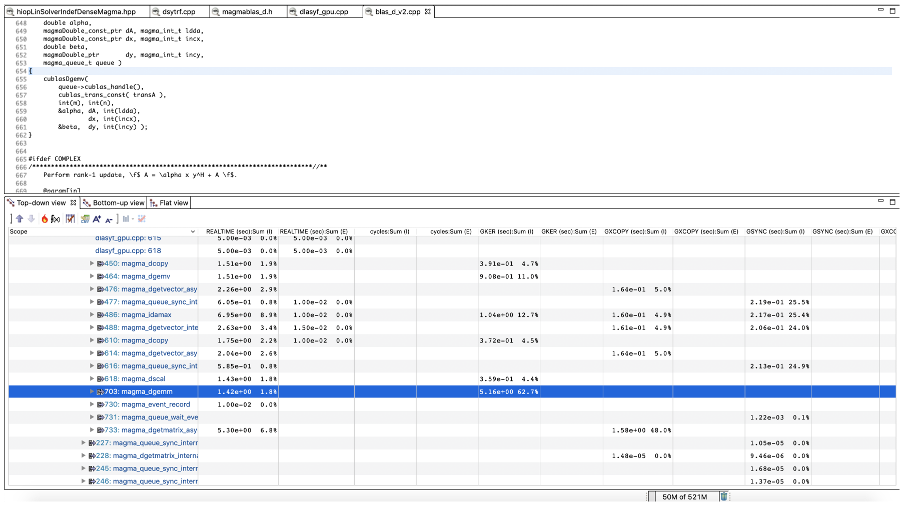
D.1.5 Graphical Profiling and Visualization using `HPCToolkit`

To study how the whole program is spending time in individual functions, we used `HPCToolkit`, a tool to measure performance and profiling for HPC applications. `HPCToolkit` can be used to identify which functions are taking the most amount of time and the call hierarchy using a visual interface. This tool can profile CPUs as well as GPUs for a basic set of metrics. We used a matrix size of $8K$ in this experiment with `HPCToolkit`. `HPCToolkit` offers two GUI tools to visualize the profiling results: 1) `HPCViewer`, and 2) `HPCTraceViewer`. We used both of them to analyze different aspects of the application.

`HPCViewer` presents a hierarchical call graph along with the source code from which the functions were called. It also provides an aggregated timing, processor cycles, and memory information for many common metrics. Due to the profiling overhead, the timing information is not entirely reliable for benchmark, rather it provides a qualitative information to compare the percentage of time being spent on different functions. Figure 12 presents the output of `HPCViewer` for this experiment. As the figure shows, out of the total time spent on the GPU:

- 9.2% (4.7% + 4.5%) is spent on `magma_dcopy()`,
- 11% is spent on `magma_dgemv()`,
- 12.7% is spent on `magma_idamax()`,
- 4.4% is spent on `magma_dscal()`, and
- 62.7% is spent on `magma_dgemm()`.

Therefore, most of the computation are being done with the `MAGMA` matrix multiplication as found here.



```
    double alpha,
    magmaDouble_const_ptr dA, magma_int_t ldda,
    magmaDouble_const_ptr dx, magma_int_t incx,
    double beta,
    magmaDouble_ptr       dy, magma_int_t incy,
    magma_queue_t queue )
{
    cublasDgemv(
        queue->cublas_handle(),
        cublas_trans_const( transA ),
        int(m), int(n),
        &alpha, dA, int(ldda),
                dx, int(incx),
        &beta,  dy, int(incy) );
}


#ifdef COMPLEX
/***************************************************************************//**
    Perform rank-1 update, \f$ A = \alpha x y^H + A \f$.

    @param[in]
```

Scope	REALTIME (sec):Sum (I)	REALTIME (sec):Sum (E)	cycles:Sum (I)	cycles:Sum (E)	GKER (sec):Sum (I)	GKER (sec):Sum (E)	GXCOPY (sec):Sum (I)	GXCOPY (sec):Sum (E)	GSYNC (sec):Sum (I)	GSYNC (sec):Sum (E)	GXC
dlasyf_gpu.cpp: 615	5.00e-03 0.0%	5.00e-03 0.0%									
dlasyf_gpu.cpp: 618	5.00e-03 0.0%	5.00e-03 0.0%									
450: magma_dcopy	1.51e+00 1.9%				3.91e-01 4.7%						
464: magma_dgemv	1.51e+00 1.9%				9.08e-01 11.0%						
476: magma_dgetvector_asy	2.26e+00 2.9%						1.64e-01 5.0%				
477: magma_queue_sync_int	6.05e-01 0.8%	1.00e-02 0.0%							2.19e-01 25.5%		
486: magma_idamax	6.95e+00 8.9%	1.00e-02 0.0%			1.04e+00 12.7%		1.60e-01 4.9%		2.17e-01 25.4%		
488: magma_dgetvector_inte	2.63e+00 3.4%	1.50e-02 0.0%					1.61e-01 4.9%		2.06e-01 24.0%		
610: magma_dcopy	1.75e+00 2.2%	1.00e-02 0.0%			3.72e-01 4.5%						
614: magma_dgetvector_asy	2.04e+00 2.6%						1.64e-01 5.0%				
616: magma_queue_sync_int	5.85e-01 0.8%								2.13e-01 24.9%		
618: magma_dscal	1.43e+00 1.8%				3.59e-01 4.4%						
703: magma_dgemm	1.42e+00 1.8%				5.16e+00 62.7%						
730: magma_event_record	1.00e-02 0.0%										
731: magma_queue_wait_eve									1.22e-03 0.1%		
733: magma_dgetmatrix_asy	5.30e+00 6.8%						1.58e+00 48.0%				
227: magma_queue_sync_interr									1.05e-05 0.0%		
228: magma_dgetmatrix_interna							1.48e-05 0.0%		9.46e-06 0.0%		
245: magma_queue_sync_interr									1.68e-05 0.0%		
246: magma_queue_sync_interr									1.37e-05 0.0%		

Figure 12: `HPCToolkit` Visual Call-Graph Results via `HPCViewer`

`HPCTraceViewer` presents a timeline view of the application along with the call depth. The interface is shown in Figure 13. As shown in the figure, there is a main CPU thread, and multiple GPU streams on different time periods. When executing `HiOp` with GPU support, it uses `MAGMA` underneath. `MAGMA` creates a new stream every time a new `BLAS` function is executed. Therefore, it is showing multiple steams of GPU usage in this figure. Incidentally, the `MAGMA` function (`magma_dsytrf()`) is called once in each iteration of `HiOp` solver. Therefore, the number of streams also corresponds to the number of `HiOp` iterations.

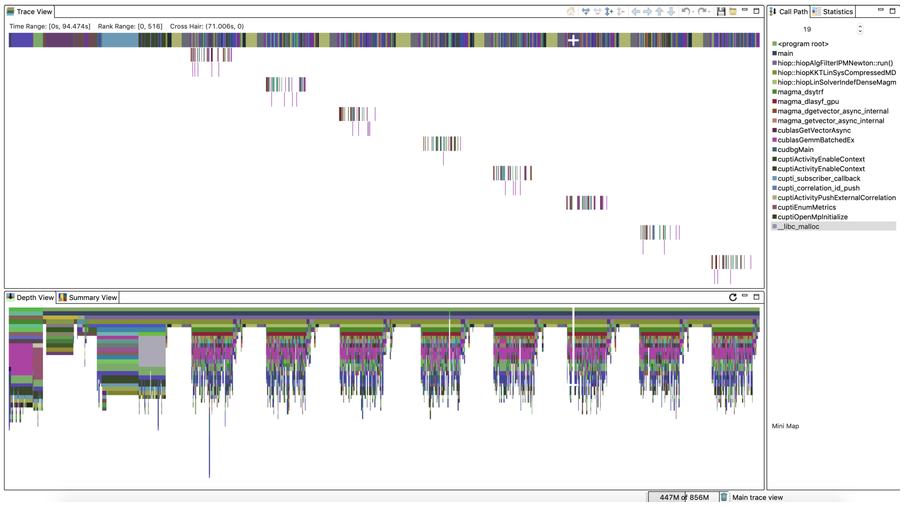

Trace View
Time Range: [0s, 94.474s]   Rank Range: [0, 516]   Cross Hair: (71.006s, 0)
Call Path
Statistics
19
<program root>
main
hiop::hiopAlgFilterIPMNewton::run()
hiop::hiopKKTLinSysCompressedMD
hiop::hiopLinSolverIndefDenseMagm
magma_dsytrf
magma_dlasyf_gpu
magma_dgetvector_async_internal
magma_getvector_async_internal
cublasGetVectorAsync
cublasGemmBatchedEx
cudbgMain
cuptiActivityEnableContext
cuptiActivityEnableContext
cupti_subscriber_callback
cupti_correlation_id_push
cuptiActivityPushExternalCorrelation
cuptiEnumMetrics
cuptiOpenMpInitialize
__libc_malloc
Depth View
Summary View
Mini Map
447M of 856M
Main trace view

Figure 13: `HPCToolkit` Results via `HPCTraceViewer`

One thing to note regarding the profiling of `HPCToolkit` for performance optimization is that the absolute timing might not be the actual run time. Therefore, we only take qualitative information from the `HPCToolkit` output, such as the percentage of time a function is taking with regard to the total run time. We do not rely on absolute time measures using this tool because it has significant run time overhead.

D.1.6 Profiling with `nvprof` and NVIDIA Visual Profiler

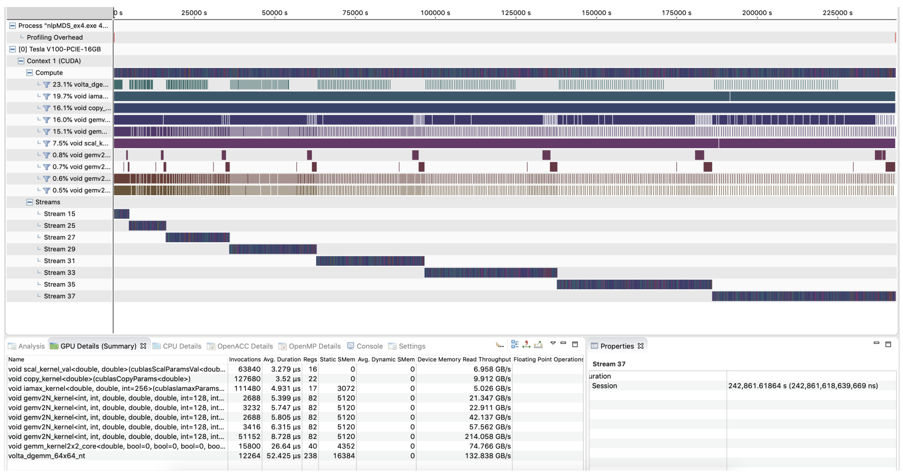
Although the `HPCToolkit` gives an idea of which functions takes the most of time, it fails to deliver some key metrics such as FLOP count and rates, DRAM throughput, achieved occupancy etc. To get those metrics, we use the native `nvprof` from NVIDIA. As profiling a compute intensive operation takes a long amount of time, we used a smaller matrix size of $4K$ in this experiment. `nvprof` took almost 70 hours to collect the desired metrics with only 8 iterations of `HiOp` solver. We used the command below to collect the profile information. The profiled data is visualized with NVIDIA Visual Profiler as shown in Figure 14.

```
#nvprof command to collect profiling metrics
nvprof --metrics flop_dp_efficiency,flop_count_dp,
    dram_read_transactions,dram_write_transactions
    nlpMDS_ex4.exe 4000 4000
```

As noted above, `HiOp` uses the MAGMA function `magma_dsytrf()` for computing the factorization on the GPU. Internally, this function uses 10 GPU kernels to compute the factorization as shown in Table 2.

Table 2: `nvprof` summarized metrics

Name	Device Memory Read Transactions	Device Memory Write Transactions	Device Memory Read Throughput (bytes/sec)	Device Memory Write Throughput (bytes/sec)	Achieved Occupancy	Floating Point Operations	Duration (ns)	FLOP Efficiency	TFLOP /Second
copy_kernel	-	-	9,976,345,713	15,678,824,305	0.00%	-	449512642	0.00	0.000
iamax_kernel	86,373,871	63,539,117	5,314,104,443	3,860,581,411	11.57%	257,117,040	549804765	0.00	0.000
scal_kernel_val	45,531,534	92,077,501	6,872,494,175	13,826,066,355	17.54%	256,158,000	209385186	0.01	0.001
gemv2N_kernel	9,682,216	9,485,604	21,241,064,649	20,529,848,816	6.20%	87,472,904	14513576	0.06	0.006
gemv2N_kernel	20,548,600	16,649,240	41,609,580,853	33,230,076,340	6.88%	197,257,464	15604941	0.13	0.013
gemv2N_kernel	2,986,718,025	1,471,900,154	196,218,593,312	98,490,468,488	11.21%	25,825,263,608	446489014	0.58	0.058
volta_dgemm_64x64_nt	2,669,009,636	1,732,499,537	132,133,738,203	85,900,388,113	5.25%	1,380,942,151,680	642948153	30.26	2.148
gemm_kernel2x2_core	983,471,632	1,182,388,381	69,774,480,330	85,642,244,738	17.99%	373,082,166,264	420922807	11.42	0.886
gemv2N_kernel	13,301,012	18,393,928	23,483,564,929	32,339,137,261	6.90%	159,905,480	18576704	0.08	0.009
gemv2N_kernel	38,805,779	34,946,831	60,250,954,921	53,301,753,535	6.20%	348,685,608	21572472	0.16	0.016



Name	Invocations	Avg. Duration	Regs	Static SMem	Avg. Dynamic SMem	Device Memory Read Throughput	Floating Point Operations
void scal_kernel_val<double, double>(cublasScalParamsVal<doub...	63840	3.279 µs	16	0	0	6.958 GB/s	
void copy_kernel<double>(cublasCopyParams<double>)	127680	3.52 µs	22	0	0	9.912 GB/s	
void iamax_kernel<double, double, int=256>(cublasIamaxParams...	111480	4.931 µs	17	3072	0	5.026 GB/s	
void gemv2N_kernel<int, int, double, double, double, int=128, int...	2688	5.399 µs	82	5120	0	21.347 GB/s	
void gemv2N_kernel<int, int, double, double, double, int=128, int...	3232	5.747 µs	82	5120	0	22.911 GB/s	
void gemv2N_kernel<int, int, double, double, double, int=128, int...	2688	5.805 µs	82	5120	0	42.137 GB/s	
void gemv2N_kernel<int, int, double, double, double, int=128, int...	3416	6.315 µs	82	5120	0	57.562 GB/s	
void gemv2N_kernel<int, int, double, double, double, int=128, int...	51152	8.728 µs	82	5120	0	214.058 GB/s	
void gemm_kernel2x2_core<double, bool=0, bool=0, bool=0, boo...	15800	26.64 µs	40	4352	0	74.766 GB/s	
volta_dgemm_64x64_nt	12264	52.425 µs	238	16384	0	132.838 GB/s	

Stream 37

uration	
Session	242,861.61864 s (242,861,618,639,669 ns)

Figure 14: `nvprof` profile visualization using NVIDIA Visual Profiler. Note that although the application time for each `HiOp` iteration is almost identical across iterations, the profiler takes more time per iteration to perform the profiling. Hence, the blue bars showing the time per stream (iteration) appear to be longer as the number of iterations increases.

D.2 Substory 2: Preliminary `HiOp` key functions profiles

- **Points**: 5 pts
- **Date Due**: July 31, 2020
- **Date Completed**: July 31, 2020
- **Authors**: Kalyan Perumalla and Maksudul Alam
- **JIRA Description**: `HiOp` key functions profiles

D.2.1 Performance Summary of HIOP for Different BLAS Implementation

We conducted experiments to study which functions are taking the most amount of computational time during different BLAS implementation in HIOP. We used nlpMDS_ex4 to study the performance.

D.2.2 H/W Setup

We used Tesla V100 GPU with 16 GB RAM, Intel(R) Xeon(R) Silver 4110 CPU, with 256GB of RAM. We used CentOS Linux 7 (Core) as the operating system.

D.2.3 Timing Info

We analyzed the code and introduced manual instrumentation of wall-clock timers in different parts of key functions to measure elapsed time for each function for each `HiOp` iteration. As each iteration of the `HiOp` solver is almost similar in run time behavior to other iterations, we present the average timing info per iteration for each experiment conducted.

D.2.4 Solver Setup

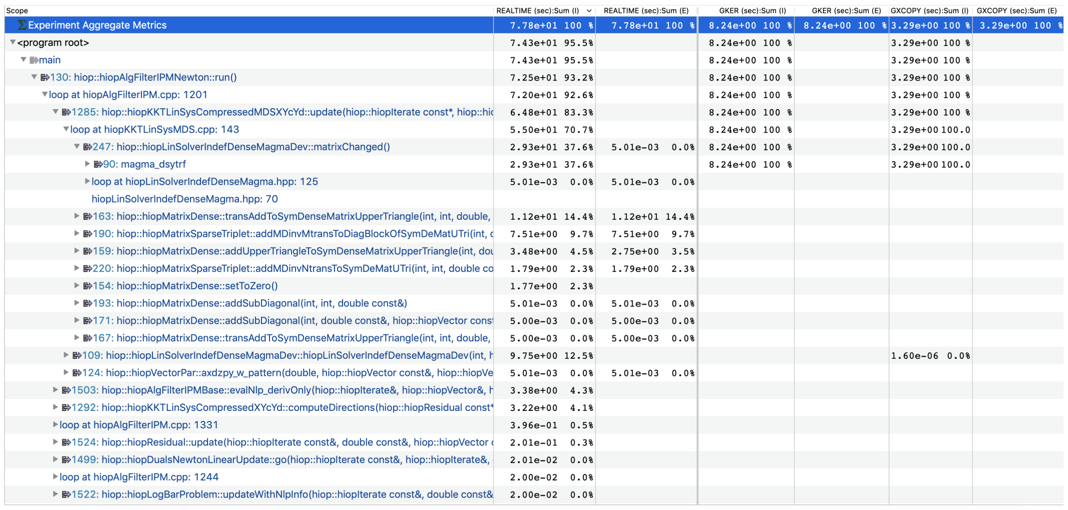
We used the `HiOp` test program `nlpMDS_ex4` to measure the performance. The program takes two inputs, 1) the size of the dense matrix, and 2) the size of the sparse matrix. For this experiment both sizes of the dense and sparse matrices is set to 15000.

D.2.5 Detailed Key Functions and Hierarchical Timing Info for MAGMA

To determine the key functions we first used `HPCToolkit` to collect the function names and the timing information without modifying the source code. The key function names can be seen from Figure 15, which is taken using `HPCViewer`. Although `HPCToolkit` does not provide with accurate timing information, it provides some insight into the functions to target. In this case the following functions seem to be of interest:

- `matrixChanged()`
- `transAddToSymDenseMatrixUpperTriangle()`
- `addMDinvMtransToDiagBlockOfSymDeMatUTri()`
- `addUpperTriangleToSymDenseMatrixUpperTriangle()`
- `magma_queue_create_internal()`
- `solveCompressed()`

Next, we investigated the source code and inserted timing instrumentation code around several additional functions of interest including the ones determined by `HPCToolkit`. The timing info collected this way more accurately measures the time needed by the functions compared to `HPCToolkit`.



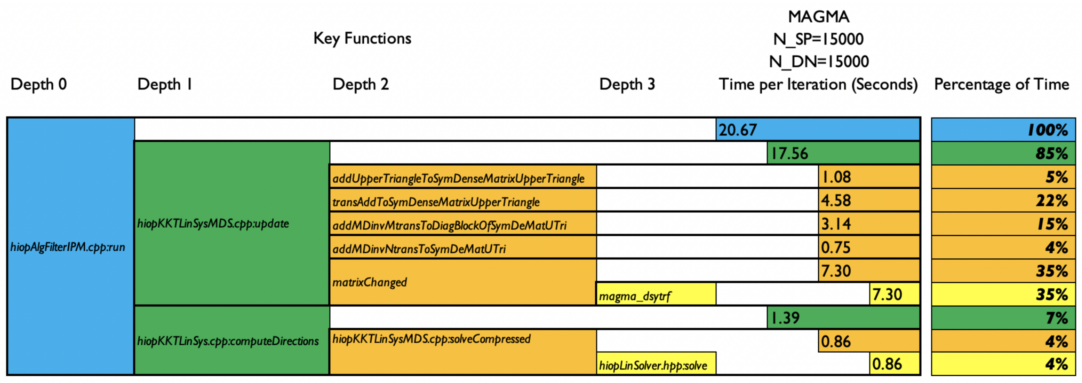
Scope	REALTIME (sec):Sum (I)	REALTIME (sec):Sum (E)	GKER (sec):Sum (I)	GKER (sec):Sum (E)	GXCOPY (sec):Sum (I)	GXCOPY (sec):Sum (E)
Experiment Aggregate Metrics	7.78e+01 100 %	7.78e+01 100 %	8.24e+00 100 %	8.24e+00 100 %	3.29e+00 100 %	3.29e+00 100 %
<program root>	7.43e+01 95.5%		8.24e+00 100 %		3.29e+00 100 %	
main	7.43e+01 95.5%		8.24e+00 100 %		3.29e+00 100 %	
130: hiop::hiopAlgFilterIPMNewton::run()	7.25e+01 93.2%		8.24e+00 100 %		3.29e+00 100 %	
loop at hiopAlgFilterIPM.cpp: 1201	7.20e+01 92.6%		8.24e+00 100 %		3.29e+00 100 %	
1285: hiop::hiopKKTLinSysCompressedMDSXYcYd::update(hiop::hiopIterate const*, hiop::hic	6.48e+01 83.3%		8.24e+00 100 %		3.29e+00 100 %	
loop at hiopKKTLinSysMDS.cpp: 143	5.50e+01 70.7%		8.24e+00 100 %		3.29e+00 100.0	
247: hiop::hiopLinSolverIndefDenseMagmaDev::matrixChanged()	2.93e+01 37.6%	5.01e-03 0.0%	8.24e+00 100 %		3.29e+00 100.0	
90: magma_dsytrf	2.93e+01 37.6%		8.24e+00 100 %		3.29e+00 100.0	
loop at hiopLinSolverIndefDenseMagma.hpp: 125	5.01e-03 0.0%	5.01e-03 0.0%				
hiopLinSolverIndefDenseMagma.hpp: 70						
163: hiop::hiopMatrixDense::transAddToSymDenseMatrixUpperTriangle(int, int, double,	1.12e+01 14.4%	1.12e+01 14.4%				
190: hiop::hiopMatrixSparseTriplet::addMDinvMtransToDiagBlockOfSymDeMatUTri(int, c	7.51e+00 9.7%	7.51e+00 9.7%				
159: hiop::hiopMatrixDense::addUpperTriangleToSymDenseMatrixUpperTriangle(int, dou	3.48e+00 4.5%	2.75e+00 3.5%				
220: hiop::hiopMatrixSparseTriplet::addMDinvNtransToSymDeMatUTri(int, int, double co	1.79e+00 2.3%	1.79e+00 2.3%				
154: hiop::hiopMatrixDense::setToZero()	1.77e+00 2.3%					
193: hiop::hiopMatrixDense::addSubDiagonal(int, int, double const&)	5.01e-03 0.0%	5.01e-03 0.0%				
171: hiop::hiopMatrixDense::addSubDiagonal(int, double const&, hiop::hiopVector const	5.00e-03 0.0%	5.00e-03 0.0%				
167: hiop::hiopMatrixDense::transAddToSymDenseMatrixUpperTriangle(int, int, double,	5.00e-03 0.0%	5.00e-03 0.0%				
109: hiop::hiopLinSolverIndefDenseMagmaDev::hiopLinSolverIndefDenseMagmaDev(int, h	9.75e+00 12.5%				1.60e-06 0.0%	
124: hiop::hiopVectorPar::axdzpy_w_pattern(double, hiop::hiopVector const&, hiop::hiopVe	5.01e-03 0.0%	5.01e-03 0.0%				
1503: hiop::hiopAlgFilterIPMBase::evalNlp_derivOnly(hiop::hiopIterate&, hiop::hiopVector&, h	3.38e+00 4.3%					
1292: hiop::hiopKKTLinSysCompressedXYcYd::computeDirections(hiop::hiopResidual const*	3.22e+00 4.1%					
loop at hiopAlgFilterIPM.cpp: 1331	3.96e-01 0.5%					
1524: hiop::hiopResidual::update(hiop::hiopIterate const&, double const&, hiop::hiopVector c	2.01e-01 0.3%					
1499: hiop::hiopDualsNewtonLinearUpdate::go(hiop::hiopIterate const&, hiop::hiopIterate&,	2.01e-02 0.0%					
loop at hiopAlgFilterIPM.cpp: 1244	2.00e-02 0.0%					
1522: hiop::hiopLogBarProblem::updateWithNlpInfo(hiop::hiopIterate const&, double const&	2.00e-02 0.0%					

Figure 15: Key functions collected from `HPCViewer`.

Figure 16 lists the key functions in a hierarchical manner up to 4-level depth. As observed from the figure, the function `magma_dsytrf()` is the most time consuming operation taking 35% of the total run time. The function computes the factorization of a symmetric matrix (MAGMA implementation of the BLAS function DSYTRF that computes the factorization of a real symmetric matrix A using the Bunch-Kaufman diagonal pivoting method). The function uses several `CUBLAS` routines to compute the factorization, specially using matrix multiplication operations on the GPU. Other key functions include:

- `transAddToSymDenseMatrixUpperTriangle()`,
- `addMDinvMtransToDiagBlockOfSymDeMatUTri()`, and
- `addUpperTriangleToSymDenseMatrixUpperTriangle()`.

These latter functions are sequential in nature and should be targeted for faster parallel implementation.



Key Functions				MAGMA N_SP=15000 N_DN=15000	
Depth 0	Depth 1	Depth 2	Depth 3	Time per Iteration (Seconds)	Percentage of Time
hiopAlgFilterIPM.cpp:run				20.67	100%
	hiopKKTLinSysMDS.cpp:update			17.56	85%
		addUpperTriangleToSymDenseMatrixUpperTriangle		1.08	5%
		transAddToSymDenseMatrixUpperTriangle		4.58	22%
		addMDinvMtransToDiagBlockOfSymDeMatUTri		3.14	15%
		addMDinvNtransToSymDeMatUTri		0.75	4%
		matrixChanged		7.30	35%
			magma_dsytrf	7.30	35%
	hiopKKTLinSys.cpp:computeDirections			1.39	7%
		hiopKKTLinSysMDS.cpp:solveCompressed		0.86	4%
			hiopLinSolver.hpp:solve	0.86	4%

Figure 16: Key functions timings collected from manual instrumentation.

D.3 JIRA Substory 3: Core dense solve (DGEMM) reproducibility and variance

- **Points**: 5 pts
- **Date Due**: Aug 31, 2020
- **Date Completed**: Aug 31, 2020
- **Authors**: Kalyan Perumalla and Maksudul Alam
- **JIRA Description**: Assess variance and reproducibility of run time of DGEMM on GPU.

D.3.1 H/W Setup

We used Tesla V100 GPU with 16 GB RAM, Intel(R) Xeon(R) Silver 4110 CPU, with 256GB of RAM. We used CentOS Linux 7 (Core) as the operating system.

D.3.2 Solver Setup and Timing Info

We used the `HiOp` `nlpMDS_ex4` program to measure the performance. The program takes two inputs, 1) the size of the dense matrix, and 2) the size of the sparse matrix. For this experiment both size of the dense and sparse matrix is varied from 2000 to 22000. We analyzed the code and placed manually inserted wall-clock timers on the core solver part performing the matrix multiplication operations.

D.3.3 Variance and Reproducibility of Dense Solver

When the execution of GPU-based linear solver was profiled for run time, it has been observed that the time taken per matrix-matrix operation varied significantly from one run to another. This affects the accounting of the fraction of time and speed distribution across the functions of the overall program. Further analyses were needed to get to the source of this run time discrepancies in the profiles.

To address this, we performed profiling on the Summit supercomputer as well as on a standalone GPU machine. Systematic experiments confirmed the following findings:

- On Summit, when two DGEMM operations are executed back-to-back, the former execution appears to complete faster than the latter execution.
- On the standalone GPU machine, the DGEMM operations executed back-to-back exhibit exactly the same run time for each execution, with little variation across iterations or across multiple program invocations.
- The discrepancy between execution times seems to get larger on Summit as the matrix size is increased.

The variance is obtained by running HiOp multiple times, one execution after another. On Summit supercomputer, this is achieved by invoking the same `HiOp` program multiple times within the same batch submission (i.e., within the same `bsub` script). The average time taken per matrix operation within one `HiOp` execution is recorded and these average times are plotted with error bars to indicate the variance. This experiment is repeated for increasing matrix sizes.

Figure 17 shows the average GPU time per iteration of each DGEMM call along with the 95% confidence intervals. As seen in the figure Summit exhibits more variances in per iteration time compared to the Summit proxy server. However the memory copy operation does not exhibit much variation compared to the DGEMM computation.

To better understand and compare the results we used relative standard deviation (standard deviation as a percentage of average) as shown in Figure 18. As seen from the figure, Summit exhibits a higher relative standard deviation for both DGEMM and memory copy operations compared to the

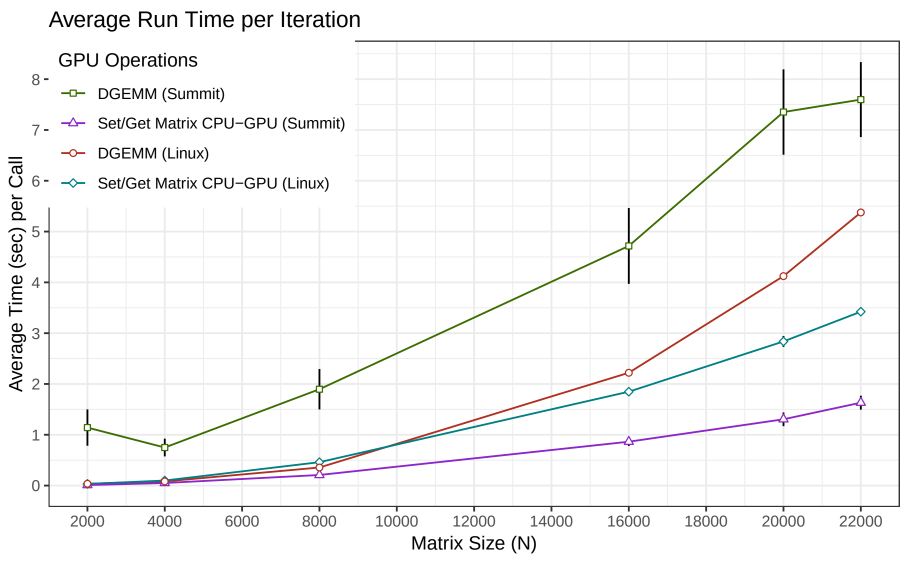


Figure 17: Average GPU Time (seconds) per DGEMM call

execution on our Summit proxy node. For larger matrix sizes, the computation time is much more consistent with Summit proxy (less than 5% relative standard deviation) than the actual Summit node (more than 20% relative standard deviation).

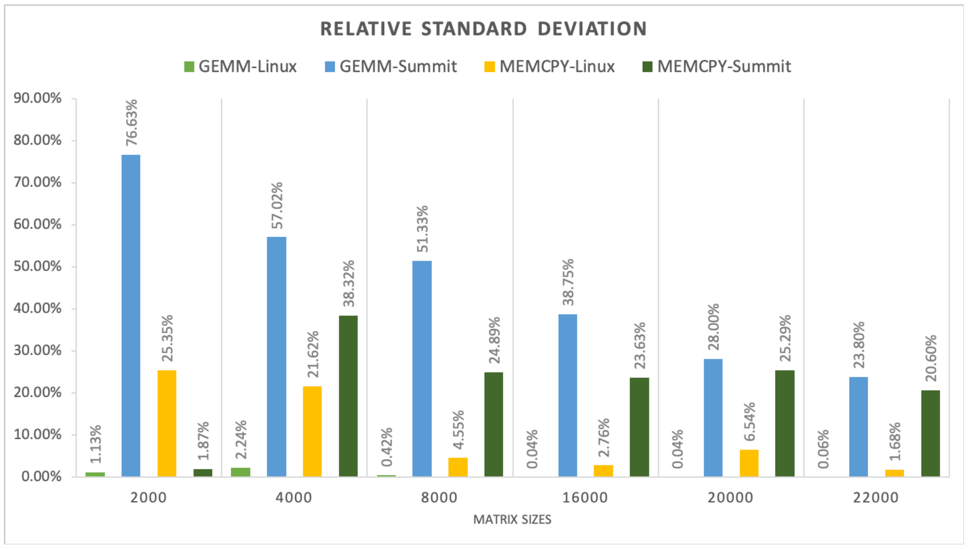


Figure 18: Relative Standard Deviation of DGEMM operations on Summit and Linux Proxy

D.4 JIRA Substory 4: SCOPFlow performance profile

- **Points**: 5 pts
- **Date Due**: Sep 30, 2020
- **Date Completed**: Sep 30, 2020
- **Authors**: Kalyan Perumalla and Maksudul Alam
- **JIRA Description**: Preliminary performance profile of SCOPFlow.

D.4.1 Building ExaGO Software Stack on Profiling Machine

The SCOPFlow program has been converted to a new ExaGO program. It is now enabled to use `HiOp` as one of the underlying solvers and can exploit the GPUs. The software dependencies are complex due to its use of all technologies used by `HiOp` in addition to its own. We have ported all these packages needed by ExaGO in order to prepare for the new profiling effort needed to understand its performance characteristics on GPU and CPU-based execution. To be able to profile ExaGO with support for `HiOp`-based solutions, we needed to port it to the standalone Linux-based Summit proxy server machine, which has different software modules and software stack than the Summit and Ascent machine architectures. The differences resulted in the need to rework and customize the build scripts, and also work around some of the CMake tool idiosyncrasies (or bugs). We have now successfully resolved all software dependencies, and developed the necessary build scripts to run it on the profiling proxy machine. Considerable effort went into determining the correct compiler and linker configurations and library specifications necessary for different modules such as netlib and MAGMA. The packages involved in this chain are listed in Table 3.

The following versions have been successfully compiled and linked. Execution of tests has just been started, and runtime errors are being resolved. Work is to be continued to resolve the runtime errors and then subject the entire software stack to the range of profiling techniques we have identified and developed as part of the projec so far.

Table 3: Packages Compiled and Linked as part of ExaGO

Package Name	**Version**	**Installation Source**
CUDA	11	NVIDIA
GCC	7.5.0	Ubuntu
Cmake	3.18.2	Spack
OpenMPI	3.1.6	Spack
OpenBLAS	0.3.10	Build From Source
MAGMA	master	Build From Source
mpfr	4.1.0	Build From Source
Suiteparse	5.8.1	Build From Source
SuiteSparse	0.11.0	Build From Source
Umpire	3.0.0	Build From Source
GMP	6.1.2	Spack
Metis	5.1.0	Spack
HiOp	raja-noineq-fix	Build From Source
PetSC	3.13.5	Build From Source
ExaGO	opflow-fix-unit-test-accuracy	Build From Source

E JIRA ADSE22-333: Close-out Report

- **Points**: 50 pts
- **Date Due**: Mar 31, 2021
- **Date Completed**: Mar 31, 2021
- **Authors**: Kalyan Perumalla and Maksudul Alam
- **JIRA Description**: Tulip One-node - Performance profiling of ExaGO and HiOp
    1. HiOP build/execute with profiling readiness
    2. HiOP-1node initial profile of routines
    3. Attempt HiOP-1node roofline or approximation
    4. ExaGO-1node initial profile of routines
    5. Attempt ExaGO-1node roofline or approximation

  Get preliminary profiling results for ExaGO and HiOp on Tulip (AMD/HIP) platform and compare with profiling results on Summit. Use same test cases – for HiOp Example 4 (a convex problem) and Texas 2000-bus grid for ExaGO. Identifying top time-consuming kernels and running roofline analysis for them is highly desirable. Share performance results with RAJA team.
  Also, evaluate profiling tools available on Tulip. Compare profiling capability on Tulip with that available on Summit.

E.1 HiOp build/execute with profiling readiness

We have successfully built and executed HiOP along with ExaGO and all their dependencies on other packages (including RAJA, Umpire, Metis and MAGMA) on Summit proxy machine (V100), Tulip (AMD pre-exascale readiness machine) and Ascent (Summit continuous integration machine).

We used the `spack` package management software to compile on all the platforms. The configurations are illustrated in Figure 19 and Figure 20.

```
exago@rebase-hip-porting-dev
    +hip+gpu+hiop+ipopt~mpi+petsc+raja build_type=Release amdgpu_target=gfx908
^exasgd.hiop@hip-porting-v0.3.99.0-dev
    +deepchecking+hip+gpu+kron~mpi+raja+shared~deepchecking
    build_type=Release amdgpu_target=gfx908
^builtin.petsc~mpi~hdf5~hypre
%clang@12.0.0-rocm
```

Figure 19: `spack` specification for building ExaGO on Tulip

E.2 HiOp 1-node initial profile of routines

Using different profilers on the different architectures, we have have been able to produce initial runtime profiles of all routines in the ExaGO and HiOP-based software repositories of ExaSGD. The NVIDIA nvprof and AMD ROCM profiler have been used to obtain detailed runtime information of the execution on two scenarios representing medium-scale and large-scale electric grid network sizes.

The profiles are included along with ExaGO profiles in Section E.4.

```
exago@0.99.2%gcc@7.5.0
    +cuda+gpu+hiop+ipopt+mpi+petsc+raja build_type=Release cuda_arch=70
^cuda@11.0
^cmake@3.18.2
^exasgd.hiop@0.3.99.2
    +cuda+gpu+kron+mpi+raja+shared~deepchecking build_type=Release cuda_arch=70
^magma@2.5.4 cuda_arch=70
^builtin.raja+cuda~examples~exercises cuda_arch=70
^builtin.umpire+cuda~examples+openmp
^ipopt+mumps
^python@3.7.0
```

Figure 20: `spack` specification for building ExaGO on Summit Proxy

E.3 Attempt HiOP 1-node roofline or approximation

A performance roofline based on floating point operations (FLOP) was possible to generate on the Summit supercomputing architecture because of the availability of NVIDIA V100's support for obtaining FLOP information from profiling on the device with native hardware support.

However, roofline based on FLOPs was not possible to be obtained directly on AMD pre-exascale machine (Tulip) because of lack of native support for instrumentation of execution for FLOP counts on the MI600 and MI100 cards. Nevertheless, a close approximation could be obtained from another natively supported metric, namely, the instruction counts, which are maintained and provided by the hardware.

Overall, we have been successful in collecting runtime data and generating detailed roofline plots for the full execution across key representative electric grid scenarios.

Since HiOP is exercised as part of the ExaSGD's ExaGO application, the rooflines for HiOP are included as part of the ExaGO roofline profiles shown later in Section E.5.

E.4 ExaGO 1-node initial profile of routines

E.4.1 Summit Proxy

All ExaGO profiles on the Summit Proxy machine were exercised on the largest available scenario of the application that would fit within one GPU node. This is the 10,000 node electric grid, which results in the underlying computational square matrix elements of size 23K along each dimension.

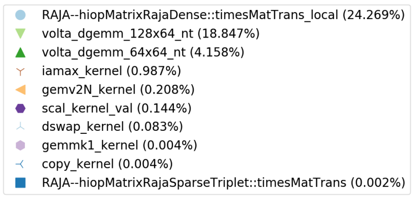
Dealing with Profiling Overheads The full profiling runs that collect all the runtime metrics such as FLOP counts, intensities, etc. consume an inordinately long amount of time to execute on the machine. Therefore, a detailed runtime profile of the entire program execution is nearly infeasible to obtain for large electric grid sizes. To deal with this problem, we adopted a two-pronged approach. First, we run the entire program from start to completion (of the global optimization problem) with basic profiling information that provides us the list of top-10 routines that are most relevant for detailed roofline analyses. Next, we run the application under detailed profiling specifications (including per-routine FLOP information) only as long as the wall clock time is a reasonable amount to wait for on a single run (in our experiments, we ran the application for up to 7 days continuously). The profiling runtime, fortunately, provided the ability to interrupt the program and still produce the detailed profile information, even though the run is a partial one.

We ran a basic profiling execution using `nvprof` for the full run. To collect the FLOP counts and memory transactions, we used a partial run that was executed up to 7 days. The partial run was quite slow to progress – it completed the initialization phase and entered the solving iterations but completed the first iteration partially by the time the 7-day wall clock time elapsed in profiling.

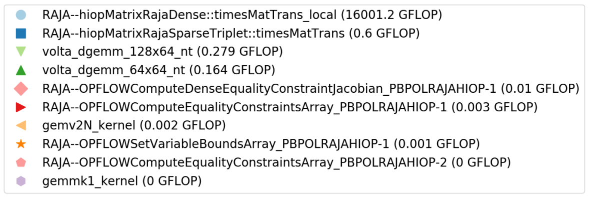
All profiles are obtained with no source modifications.

We collected top 10 of each GPU routines from the profiling information based on four categories. The top 10 routines based on each category are shown in Figure 21.

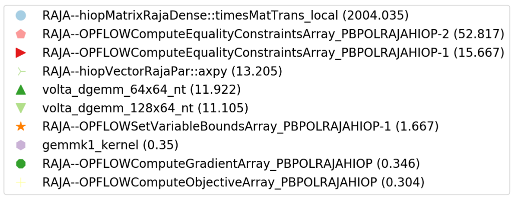
1. Total Duration of Full Run
2. Partial FLOP Count
3. Partial Arithmetic Intensity
4. Partial FLOP Rate



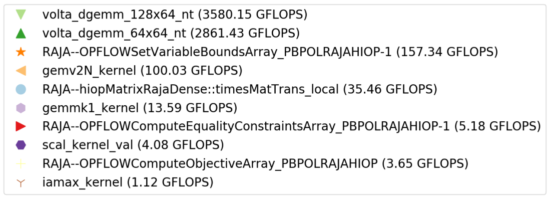
(a) Top 10 Sorted by Total Duration
(% time in full optimization run)



(b) Top 10 Sorted by FLOP
(GFLOP in partial profiling run)



(c) Top 10 Sorted by Arithmetic Intensity
(FLOP/byte in partial profiling run)



(d) Top 10 Sorted by FLOP Rate
(GFLOP/sec in partial profiling run)

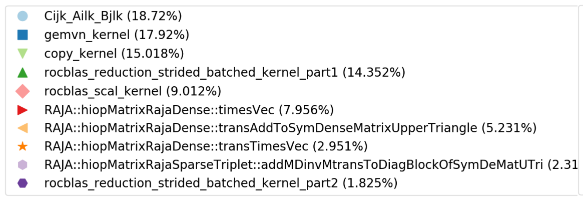
Figure 21: Top 10 Routines on VOLTA V100

E.4.2 Tulip

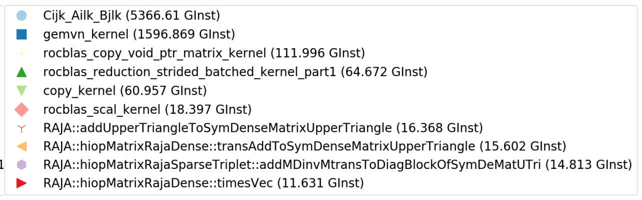
All ExaGO profiles on the Tulip machine were exercised on a medium-sized scenario (2K electric grid size = 4.8K matrix sizes). We ran profiling using `rocprof` for the full run. The profiling support was found to be limited on Tulip. As mentioned earlier, the FLOP operations are not possible to get at the moment using the device counters. Therefore, we used instruction counters. All profiles are obtained with no source modifications.

We collected the top 10 routines among all the GPU routines from the profiling information based on four categories. The top 10 routines based on each category are shown in Figure 22.

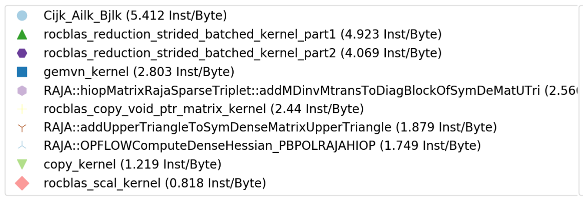
1. Total Duration of Full Run
2. Total Instruction Count
3. Arithmetic Intensity
4. Instruction Rate



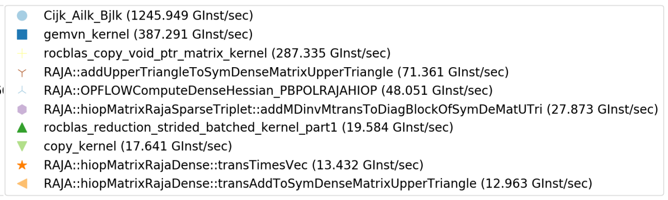
(a) Top 10 Sorted by Total Duration
(% time in full optimization run)



(b) Top 10 Sorted by Instructions
(GInst in full profiling run)



(c) Top 10 Sorted by Arithmetic Intensity
(GInst/byte in full profiling run)



(d) Top 10 Sorted by Instruction Rate
(GInst/sec in full profiling run)

Figure 22: Top 10 Routines on Tulip AMD MI100

E.5 Attempt ExaGO 1-node roofline or approximation

We generated a roofline plot for both Summit Proxy and Tulip machines using the data described previously. On the Summit Proxy, the actual FLOP counts were collected using `nvprof`. However, as the time required to collect the FLOP counts are quite large, it only have a partial count of the FLOP operations. The partial roofline plot is shown in Figure 23.

On the other hand, the Tulip machine does not provide FLOP counts, therefore, we used instruction counts instead of the FLOP counts for the roofline plot. The instruction based roofline plot is shown in Figure 24.

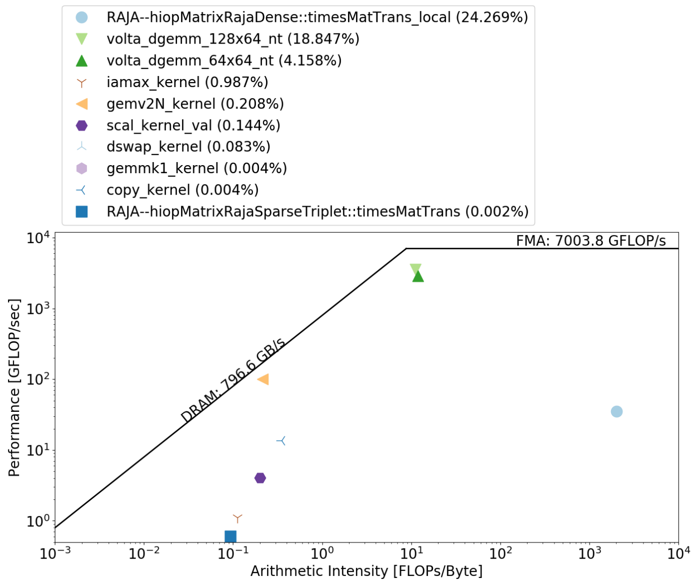


Figure 23: FLOP based Roofline for VOLTA V100

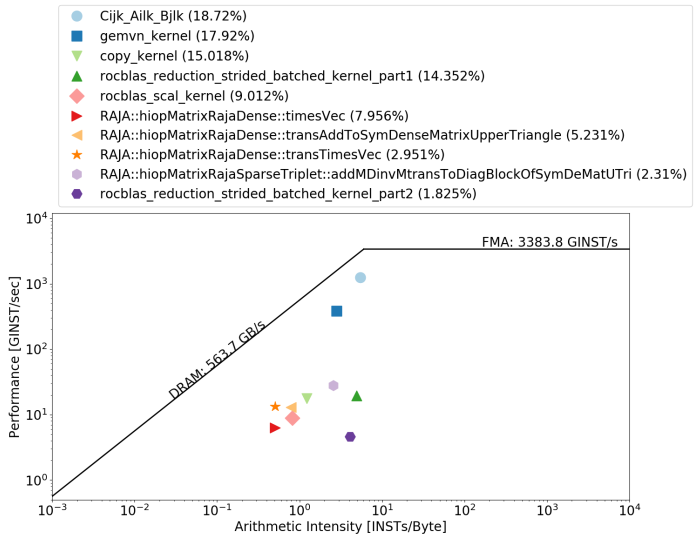


Figure 24: Instruction Count based Roofline for AMD MI100

E.6 Preliminary performance comparison across architectures

We have performed a preliminary performance comparison across the different architectures important for exascale applications. To provide a basic idea, we measured the performance on two network grid sizes, namely, 2K and 10K. The runtime is categorized into four bins corresponding to the four phases in the ExaGO solver:

- Main stage
- Reading data
- Set up, and
- Solve.

Across the architectures as well as across the network grid sizes, the solve phase was confirmed to consume the most amount of runtime, which was useful to eliminate the possibility of any bottlenecks in the preliminary phases. The solve phase takes the most amount of time because of the primary optimization operations performed in it.

It was also observed that there is a slight performance difference between Summit proxy and Tulip machines on the smaller grid size, but that difference becomes a relatively insignificant fraction when the network size is increased. This informs us that the code is fairly robust except for some constant overhead on small network sizes which is not a major cost.

- **Summit Proxy**: `HiOp 0.3.99.2`, `ExaGO 0.99.2`, `MAGMA 2.5.4`, `CUDA 11.0`
- **Tulip**: `HiOp hip-porting-v0.3.99.0-dev`, `ExaGO rebase-hip-porting-dev`, `MAGMA hipMAGMAv2.0.0`, `ROCM 4.0.0`
- **Ascent**: `HiOp 0.3.99.2`, `ExaGO 0.99.2`, `MAGMA 2.5.4`, `CUDA 10.2`

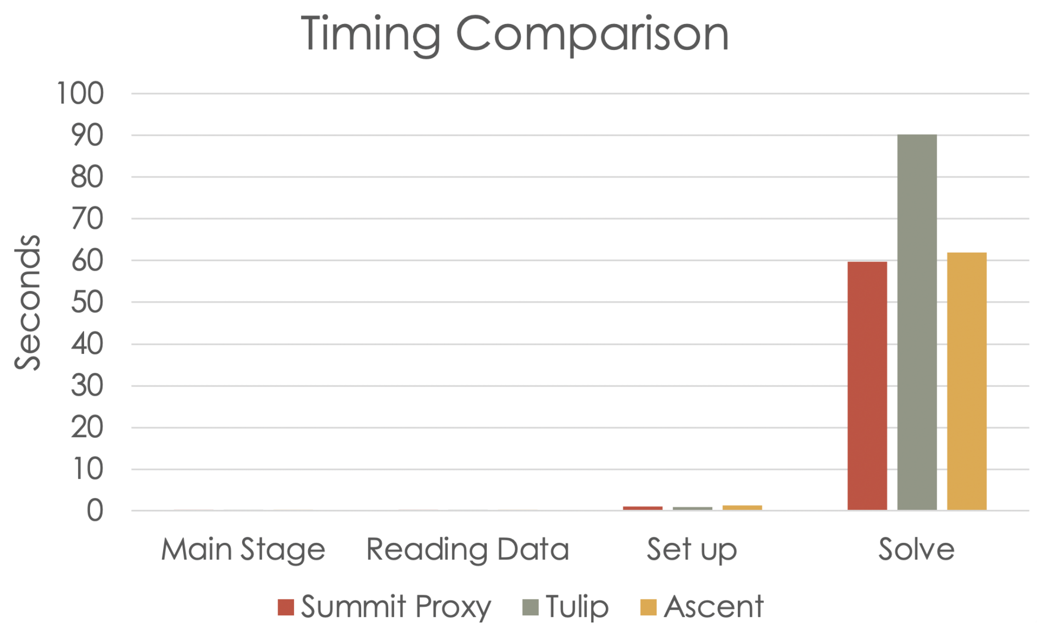


Figure 25: case_ACTIVSg2000.m
Grid Size 2K (Matrix Size 4.8K)
#Iterations=45

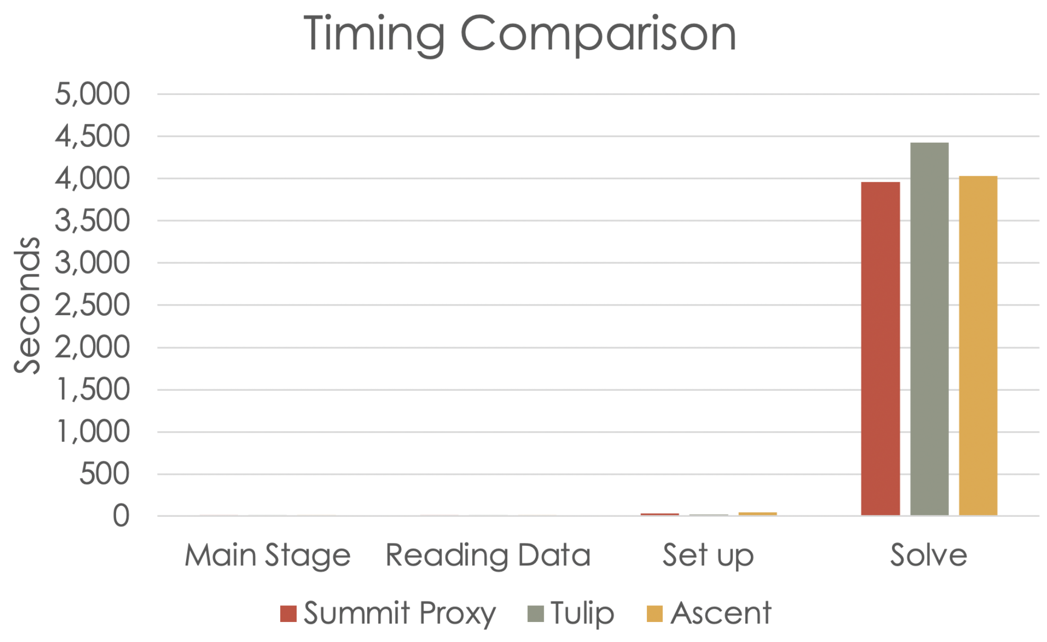


Figure 26: case_ACTIVSg10K.m
Grid Size 10K (Matrix Size 23K)
#Iterations=141

F JIRA ADSE22-432: Close-out Report

- **Points**: 60 pts
- **Date Due**: Dec 31, 2021
- **Date Completed**: Dec 31, 2021
- **Authors**: Kalyan Perumalla and Maksudul Alam
- **JIRA Description**: Profiling Activities
  1. HiOp 1-node initial profiling with TAU and Vampir
  2. HiOp multi-node initial profiling with TAU

  Get preliminary MPI-profiling results for HiOp on Ascent and Summit. Use same test cases – for HiOp Example 9 (Primal Decomposition). Identify top time-consuming MPI-Calls and analyze the variances.

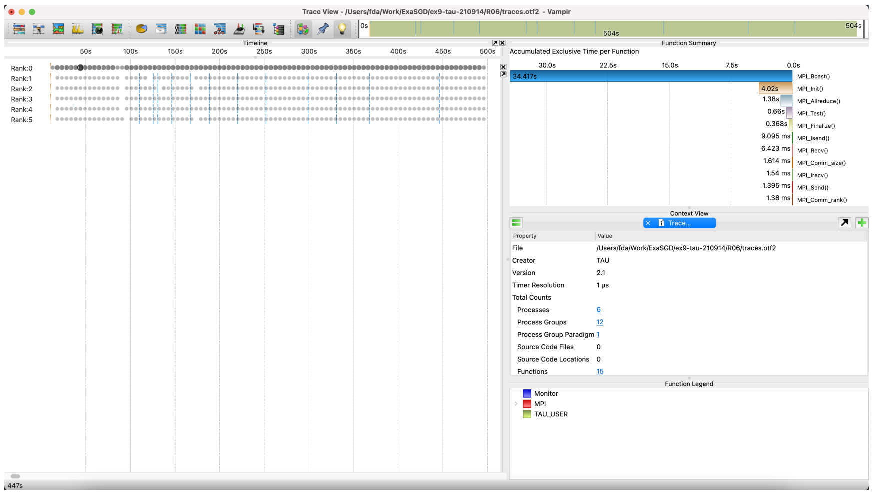
F.1 TAU Based MPI Profiling

F.1.1 Setup

We built the HiOp executables from the `github` repository and used the `nlpPriDec_ex9.exe` example to perform the MPI profiling. For the profiling, we used `TAU 2.30.1` on Summit. For the visualization we used `VAMPIR 9.11.1` (Local) and `9.10` (Summit). The experiments were conducted on 1, 2, 4, 20, and 128 computing nodes on Ascent for profiling with 1, 12, 24, 120, and 768 GPUs. We also used a $NX = 1000$ as the input for the `nlpPriDec_ex9.exe`. Each MPI process exactly one recurse problem on the GPU.

F.1.2 Visualization

We are able to successfully produce a visual profile using the VAMPIR tool. One of the visualization produced is shown in Figure 27. We can see the main timeline and a summary of the functions on the right. The summary shows total time taken by each type of MPI calls over the whole time period. We can zoom into finer level of details on the main timeline as shown in Figure 28.



34.417s	MPI_Bcast()
4.02s	MPI_Init()
1.38s	MPI_Allreduce()
0.66s	MPI_Test()
0.368s	MPI_Finalize()
9.095 ms	MPI_Isend()
6.423 ms	MPI_Recv()
1.614 ms	MPI_Comm_size()
1.54 ms	MPI_Irecv()
1.395 ms	MPI_Send()
1.38 ms	MPI_Comm_rank()

Property	Value
File	/Users/fda/Work/ExaSGD/ex9-tau-210914/R06/traces.otf2
Creator	TAU
Version	2.1
Timer Resolution	1 µs
Total Counts	
Processes	6
Process Groups	12
Process Group Paradigm	1
Source Code Files	0
Source Code Locations	0
Functions	15

🟦	Monitor
🟥	MPI
🟩	TAU_USER

Figure 27: VAMPIR Screenshots of MPI profiling with TAU with 6 GPUs

Using VAMPIR, a group of message exchanges in a small period of time called a message burst is observed as a big black circle. Zooming into the burst region provides insights into the pattern of

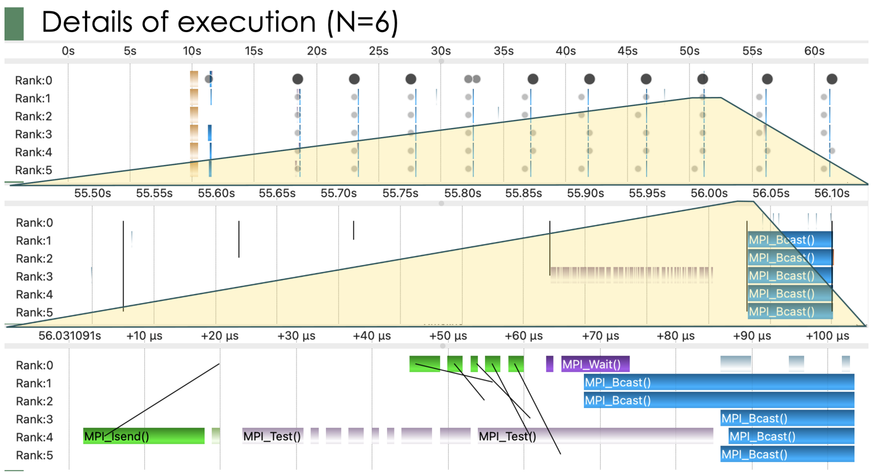


Figure 28: Zoom into the timeline for Details of MPI calls using VAMPIR

the message communications, including the origin and destination of the messages as a black arrows. The message burst are shown in Figure 29. Message bursts are important visualization tool that confirms the expected communication patterns.

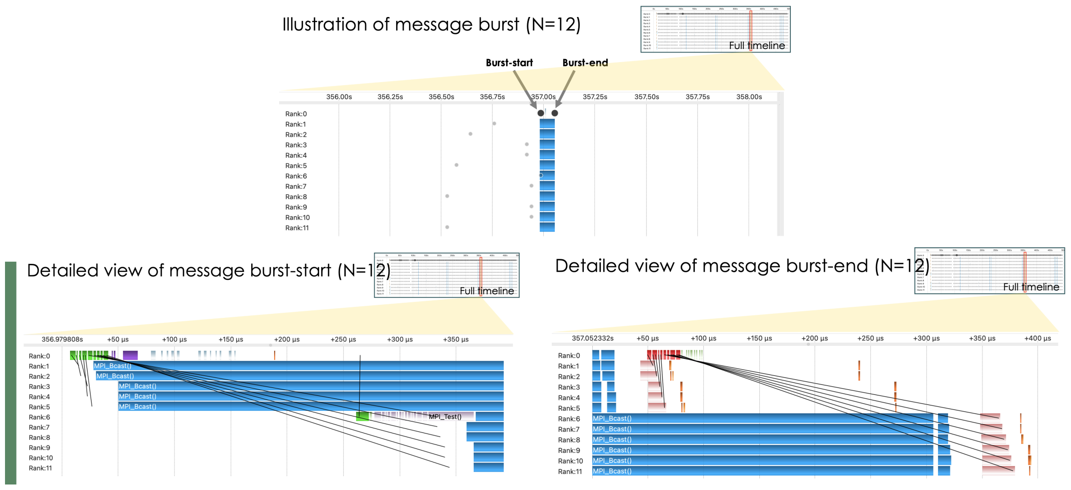


Illustration of message burst (N=12)
Full timeline
Burst-start
Burst-end
Detailed view of message burst-start (N=12)
Detailed view of message burst-end (N=12)
MPI_Bcast()
MPI_Test()

Figure 29: Message burst and communication pattern for $N = 12$

F.2 Extrapolation to Exa-Scale

We have used the detailed profiling data to arrive at rough estimates on the path to reaching exa-scale. In particular, the floating point computation rates along with specific weights of the top computationally-intensive routines have been factored to determine reasonable extrapolations. Although, the results are obtained from the Summit architecture, we were able to consider the increased speeds and capacities of the AMD GPUs based on publicly available information. Figure 30 shows the extrapolation of peak FLOP/sec with increasing number of GPUs. To obtain the baselines needed for the extrapolation, we used $N = 1, 6, 12, 24$ GPUs to get the peak FLOP/sec performance of the most computation intensive kernel (`volta_gemm_128x64`) for the executable `nlpPriDec_ex9.exe` with a large input size of 18K. The Peak GFLOP/sec of each run is used in the extrapolation to get the expected total GFLOP/sec. Extrapolation with $N = 1$ shows the highest GFLOP/sec performance.

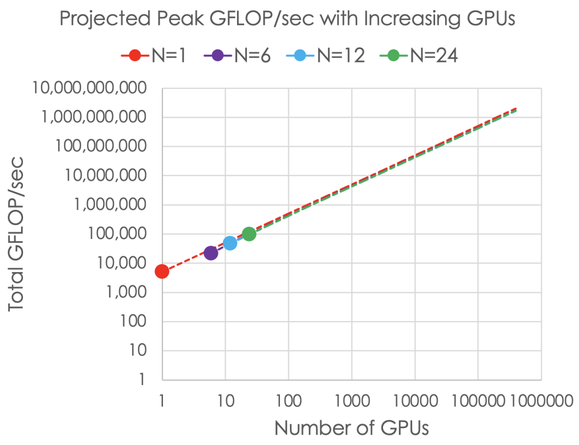


Figure 30: Projected HiOp Peak GFLOP/sec with Increasing number of GPUs

F.3 Summary

The detailed descriptions and results have been presented in the slides of our ExaSGD technical seminar (Nov 2, 2021) which is accessible on confluence.

- We faced and overcame multiple software configuration and execution issues with TAU and VAMPIR tools needed for our profile on Summit. The Summit upgrade introduced additional challenges which we overcame with close interaction with OLCF support team over multiple weeks.
- We have successfully developed visual profiling output for the HiOp solver part of the software stack and generated important insights for the run time behavior. The HiOp team was able to confirm certain dynamics and also found some additional dynamics to be investigated. For instance, our profiled visual output was able to to confirm the master GPU's idle time and the bursts of messages from master to workers. We also were able to visually determine the insignificant overhead of the broadcast operation relative to the solver time by all the workers.
- We were able to successfully exercise the visual profile effort to larger number of GPUs. However, we have encountered post-processing issues to handle the large trace output and are reaching the limits of the graphical interface. To overcome this in profiling larger runs we investigated alternative scalable approaches.
- The TAU on upgraded summit has introduced a new limitation that is unexpected. While previously only one thread was spawned per GPU, now four threads are spawned which quickly exhausts the thread limit of VAMPIR license. We are now investigating a command line approach to avoid visual limitations.

G JIRA ADSE22-436: Close-out Report

- **Points**: 60 pts
- **Date Due**: Mar 31, 2022
- **Date Completed**: Mar 31, 2022
- **Authors**: Kalyan Perumalla and Maksudul Alam
- **JIRA Description**: Profiling Activities
  1. Large Summit runs - OTF-based statistics: Get preliminary large-scale MPI-profiling results for HiOp on Summit. Use same test cases for HiOp (Example 9 Primal Decomposition). Identify top time-consuming MPI-Calls and analyze the variances using custom-developed scripts for processing Open Trace Format (OTF) files from profiled MPI runs.
  2. ExaGO latest S/W performance results: Measure performance of recent ExaGO version 1.3.0 relative to previous optimization algorithm.
  3. HiOp shallow profiles on Crusher: Preliminary porting and performance experiments on Crusher.

G.1 Hardware Platforms

G.1.1 Hardware Setup

We used multiple systems for the purpose of performance experiments. The configurations of the systems are given below.

Summit
- GPU: 6× `Tesla V100-SXM2-16GB:s_7.0`
- CPU: IBM POWER9
- OS: Red Hat Enterprise Linux
- Memory: 512GB per node

Local Summit Proxy Node Linux Server
- GPU: `Tesla V100-SXM2-16GB:s_7.0` (same as Summit GPU)
- CPU: Intel(R) Xeon(R) Silver 4110 CPU
- OS: UBUNTU 18.04.6 LTS
- Memory: 256GB

Newell
- GPU: 2× `Tesla V100-SXM2-16GB:s_7.0` (same as Summit GPU)
- CPU: Intel(R) Xeon(R) Silver 4110 CPU
- OS: Red Hat Enterprise Linux
- Memory: 1TB

Crusher
- GPU: 4× `AMD MI250X 64 GB Memory`
- CPU: 64-core AMD EPYC 7A53
- OS: Red Hat Enterprise Linux
- Memory: 512 GB DDR4

G.2 Large-Scale MPI Profiling with TAU

Previously, we conducted experiments with TAU-based MPI profiling of HiOp executables (`nlpPriDec_ex9.exe`) using up to 120 GPUs. We extended the experiment with large scale runs using 768 GPUs. Here we document the challenges we faced, and the results we obtained, and alternative solutions to work on the challenges.

G.2.1 Setup

We built the HiOp executables from the `github` repository and used the `nlpPriDec_ex9.exe` example to perform the MPI profiling. For the profiling, we used `TAU 2.30.1` on Summit. For the visualization, we used `VAMPIR 9.11.1` (Local) and `9.10` (Summit). The experiment was conducted on 128 computing nodes on Summit for profiling with 768 GPUs. We also used a value of $NX = 1000$ as the input for the `nlpPriDec_ex9.exe`. Each MPI process has exactly one recourse problem on the GPU.

G.2.2 Challenges

We faced the following challenges with large-scale MPI profiling with TAU.

Limitation of Visualization: One of the major problems with large-scale MPI profiling is the limitation of visualization tools such as VAMPIR that we used. When there are a large number of MPI processes, the visualization is sometimes difficult to interpret because the output is too unwieldy and cluttered to be sensible with many MPI ranks. As shown in Figure 31, with a modest parallel size of 120 GPUs, the window can only show a limited number of threads at a time, which is often difficult to interpret, especially regarding the inter rank message communications. Furthermore, VAMPIR displays all threads that are spawned from the MPI processes. The VAMPIR license currently limits the number of threads visualized to 1000. To further complicate the issue, the recent RHEL upgrade on Summit increased the number of threads by $\tilde{4}\times$. Due to these issues, we were unable to use VAMPIR as the primary profiling tool for large runs on Summit.

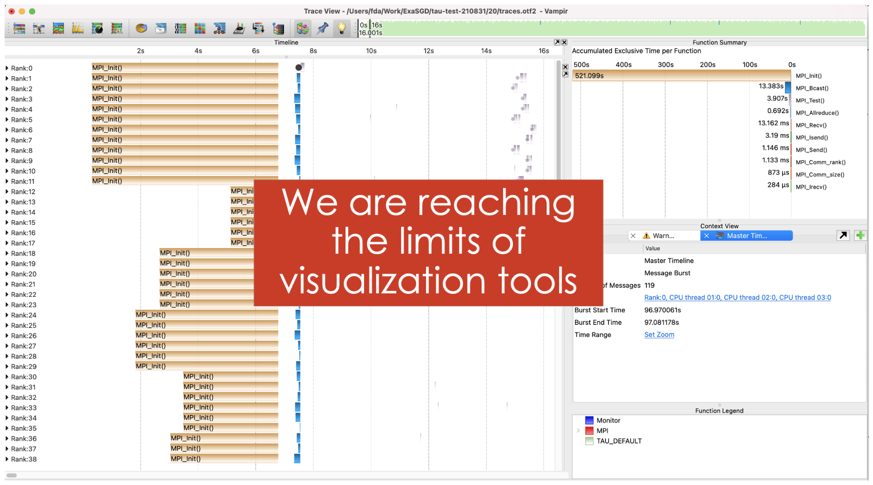


Figure 31: VAMPIR Screenshots of MPI profiling with TAU with 120 GPUs shows a limited number of threads could be visible in a window.

Limited Statistics: Furthermore, using VAMPIR, we can only use a preset collection of statistics that VAMPIR provides, which we found lacking for obtaining important statistics such as standard deviation measures. As our experiments show, those statistics are important to analyze the performance of our HiOp application, particularly timing into the rich set of MPI calls that it makes.

G.2.3 Our Solution

To get around the limitations of VAMPIR, we decided to generate and process the detailed trace files from TAU. Fortunately, TAU is capable of generating standard OTF-formatted trace files for our MPI execution.

We developed customized scripts to directly read the trace files generated by TAU. Additionally, we are able to generate customized statistics to better analyze and explain the behavior of the HiOp-based parallel runs (multi-node, multi-GPU).

Visualizations using the data generated by our scripts are shown in Figure 32 and 33.

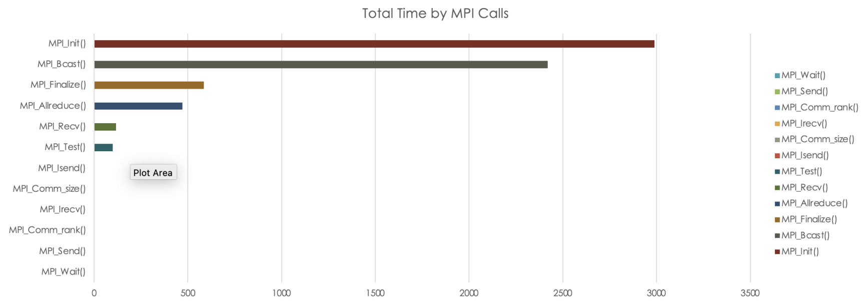


Figure 32: Total Time per MPI Calls using customized scripts.

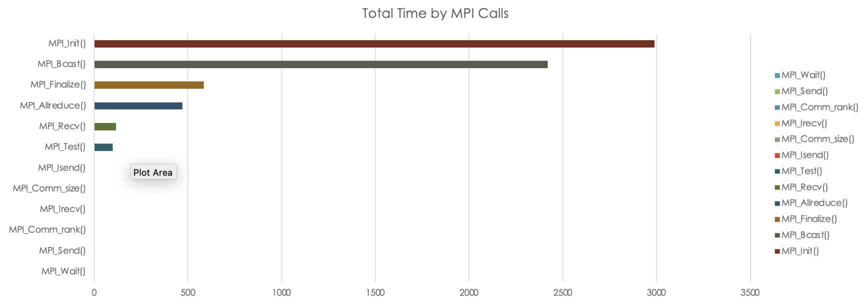


Figure 33: Avg. percentage of max MPI time spent by each rank in each MPI call.

G.3 Profiling ExaGO

All ExaGO profiles on the Summit Proxy machine were exercised on the largest scenario that fits within one GPU node. This scenario corresponds to the 10,000-node electric grid case, which results in the underlying computational square matrix elements to be of size 23K along each dimension.

We used the configuration shown in Figure 34 to build ExaGO on the Summit Proxy. The results were compared with a run with previous year's performance executed on March 2021.

As shown in Figure 35, the solving time is significantly reduced in the latest version of the ExaGO compared to last year. The most significant difference is that it takes fewer iterations to converge compared to last year (39 iterations latest version vs. 141 iteration last year).

```
exago@1.3.0%gcc@7.5.0
    +cuda+mpi+raja+ipopt+hiop cuda_arch=70 build_type=Release
^cuda@11.0
^cmake@3.21.3
^hiop@0.5.3
    +cuda+mpi+raja+sparse+deepchecking cuda_arch=70
^magma@2.6.1 +cuda cuda_arch=70
^raja@0.14.0+openmp+cuda cuda_arch=70
^umpire@6.0.0+openmp+cuda
^ipopt@3.12.10+coinhsl+metis~mumps
^openblas@0.3.17
^python@3.8.0
```

Figure 34: spack specification for building ExaGO on Summit Proxy

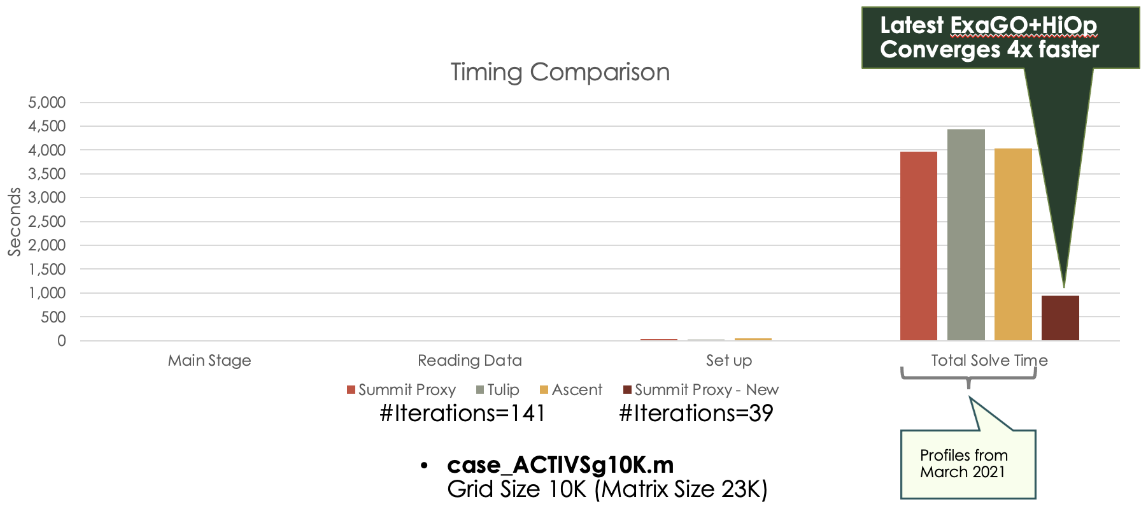
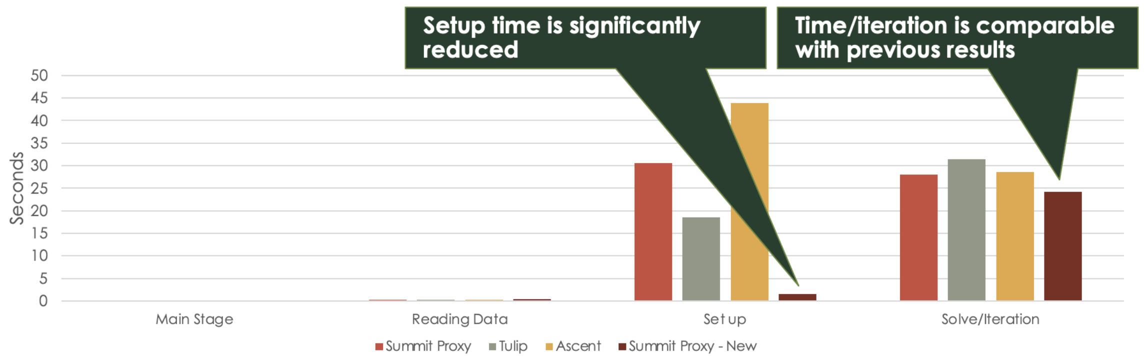


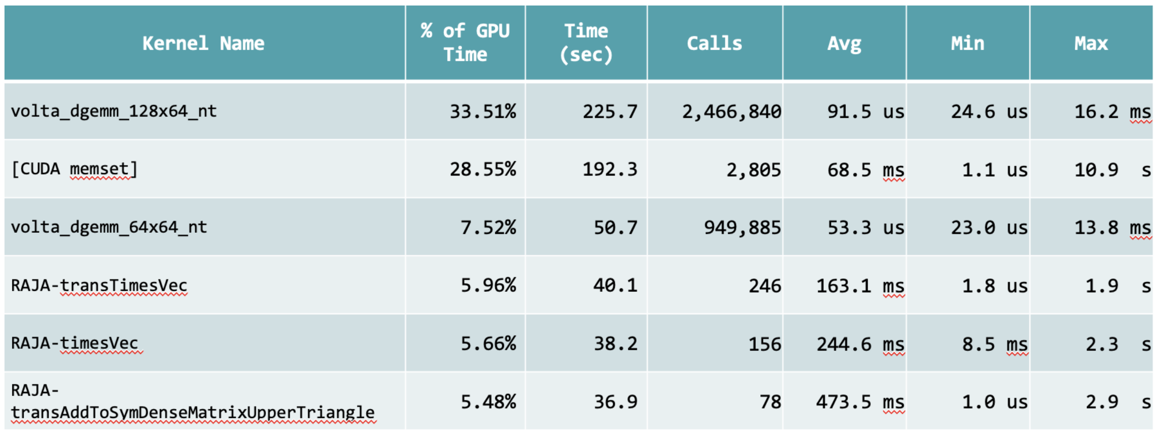
Figure 35: ExaGO Solver Execution Times on Platforms

When we take a deeper look into different stages of the execution shown in Figure 36, we note that the setup time is significantly reduced compared to last year's version. Also, there is a slight improvement of time per iteration in the current version of the ExaGO compared to last year's. On Summit-Proxy, we also performed a shallow profile of ExaGO and listed the ExaGO top computationally intensive kernels that consume at least 5% of total GPU time. They are listed in Figure 37. Note that these numbers correspond to the parallel dynamics by which ExaGO spends 73% of its time on GPU (673 seconds on GPU with a total run time of 920 seconds).



- **case_ACTIVSg10K.m**
  Grid Size 10K (Matrix Size 23K)

Figure 36: ExaGO Solver Execution Timing/Iteration on Platforms

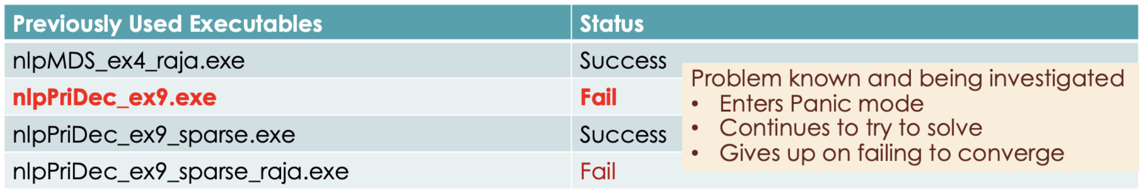


Kernel Name	% of GPU Time	Time (sec)	Calls	Avg	Min	Max
volta_dgemm_128x64_nt	33.51%	225.7	2,466,840	91.5 us	24.6 us	16.2 ms
[CUDA memset]	28.55%	192.3	2,805	68.5 ms	1.1 us	10.9 s
volta_dgemm_64x64_nt	7.52%	50.7	949,885	53.3 us	23.0 us	13.8 ms
RAJA-transTimesVec	5.96%	40.1	246	163.1 ms	1.8 us	1.9 s
RAJA-timesVec	5.66%	38.2	156	244.6 ms	8.5 ms	2.3 s
RAJA-transAddToSymDenseMatrixUpperTriangle	5.48%	36.9	78	473.5 ms	1.0 us	2.9 s

Figure 37: ExaGO Top Kernels (>= 5% of GPU Time) on Summit-Proxy [from shallow profiling]

G.4 Initial Profiling on Crusher (pre-Frontier) Machine

We successfully built HiOp using a pre-compiled set of binaries on Crusher. The build uses the recent HiOp version 0.5.3. However, on Crusher, not all executables are operational at the time of this writing. Figure 38 shows a list of the executables that are currently operational on Crusher.



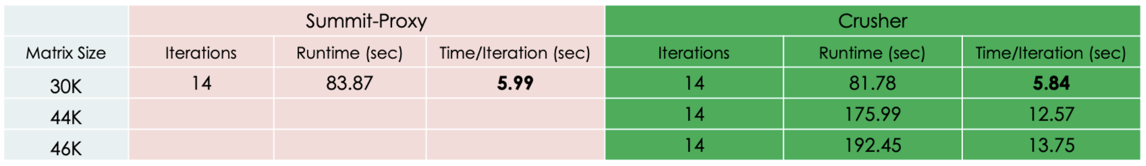
Previously Used Executables	Status
nlpMDS_ex4_raja.exe	Success
nlpPriDec_ex9.exe	**Fail**
nlpPriDec_ex9_sparse.exe	Success
nlpPriDec_ex9_sparse_raja.exe	Fail

Problem known and being investigated
- Enters Panic mode
- Continues to try to solve
- Gives up on failing to converge

Figure 38: List of HiOp examples

We used `nlpMDS_ex4_raja.exe` to test the performance of HiOp on Crusher vs. Summit Proxy. The findings are summarized next.

- While Crusher has 8× more memory, it does not translate to $\sqrt{8}\times$ larger matrix size. Instead, it holds a matrix that is only 1.5× larger. The timings from the scaled matrices are shown in Figure 39.
- At the moment, it is observed that the experimental performance is not yet reaching the advertised peak performance for the GPU. To empirically ascertain the peak performance, we used a set of publicly available micro-kernels to benchmark Crusher's performance and found the following advertised-peak vs. observed metrics:
  - Peak FP64 FLOP/sec is 19,075.42 (Observed) vs. 47,900 (Advertised) at 38% efficiency
  - Peak Bandwidth (GB/s) is 1,220.21 (Observed) vs. 3,276.8 (Advertised) at 37% efficiency
  - FLOP/Instruction is 1.97 Avg. FLOP per Instruction



	Summit-Proxy			Crusher		
Matrix Size	Iterations	Runtime (sec)	Time/Iteration (sec)	Iterations	Runtime (sec)	Time/Iteration (sec)
30K	14	83.87	**5.99**	14	81.78	**5.84**
44K				14	175.99	12.57
46K				14	192.45	13.75

Figure 39: HiOp Performance Comparison: Crusher vs. Summit Proxy

G.4.1 Profiling Challenges on Crusher

The following are some of the challenges we encountered in moving to Crusher.

- The `rocprof` AMD profiler output seems to miss key information
  - No native FLOP counts/efficiency metrics
  - Without native FLOP instrumentation on MI250X, we continue to rely on "Instruction-based Roofline Analysis"
- There is inconsistency in obtaining duration information for the profiled routines

  - The use of `rocprof` to obtain kernel durations and counters at the same time is problematic
  - Profiling involves running the same executable twice
    - First without collecting metrics (collecting only the run time duration of each routine)
    - Then executing again to obtain individual metrics

- The number of counters that can be collected in any single run is limited by the GPU hardware to small constant
  - Therefore, multiple runs need to be executed to collect all the desired metrics.

H JIRA ADSE22-440: Close-out Report

- **Points**: 60 pts
- **Date Due**: Jun 30, 2022
- **Date Completed**: Jun 30, 2022
- **Authors**: Kalyan Perumalla and Maksudul Alam
- **JIRA Description**: Profiling Activities
    1. Performance analyses on dense versus sparse solvers
    2. Performance profiles of latest HiOp versions
    3. Performance profiles of latest ExaGO versions
    4. Performance on Summit and Crusher (pre-Frontier) architectures
    5. First roofline results from Crusher (pre-Frontier) system

H.1 Computing Platforms

H.1.1 Hardware Setup

We used multiple systems for the purpose of running the performance profiling experiments. The configurations of the systems are given below.

Summit
- GPU: 6× `Tesla V100-SXM2-16GB:s_7.0`
- CPU: IBM POWER9
- OS: Red Hat Enterprise Linux
- Memory: 512GB per node

Local Summit Proxy Node Linux Server
- GPU: `Tesla V100-SXM2-16GB:s_7.0` (same as Summit GPU)
- CPU: Intel(R) Xeon(R) Silver 4110 CPU
- OS: UBUNTU 18.04.6 LTS
- Memory: 256GB

Newell
- GPU: 2× `Tesla V100-SXM2-16GB:s_7.0` (same as Summit GPU)
- CPU: Intel(R) Xeon(R) Silver 4110 CPU
- OS: Red Hat Enterprise Linux
- Memory: 1TB

Crusher
- GPU: 4× `AMD MI250X 64 GB Memory`
- CPU: 64-core AMD EPYC 7A53
- OS: Red Hat Enterprise Linux
- Memory: 512 GB DDR4

H.1.2 Software Setup

We built the ExaGO executables from the `git` repository and used the `opflow` and `scopflow` examples to perform the profiling. For the profiling we used three grid scenario datasets: `case_ACTIVSg200.m`, `case_ACTIVSg2000.m`, and `case_ACTIVSg10k.m`. To build ExaGO on the Crusher, we used the `spack`-based configuration shown in Figure 40. For Summit we used the `spack`-based configuration shown in Figure 41. Note that we also compiled another version of ExaGO where the specification of HiOp was changed from develop to `0.5.4` for each platform. On Crusher we used ROCM 4.5.0 and on Summit as well as Summit proxy we used CUDA 11.4.

```
exago@develop%clang@14.0.0+rocm+raja+mpi+hiop+ipopt~cuda amdgpu_target=gfx90a
^hiop@develop%clang@14.0.0+rocm+raja+mpi+shared+sparse~cuda amdgpu_target=gfx90a
^libiconv%gcc@7.5.0
^ipopt@3.14.5%clang@14.0.0+coinhsl~mumps
^openblas@0.3.20%clang@14.0.0
^petsc@3.16.6%clang@14.0.0~hdf5~hypre~superlu-dist
^raja@0.14.0%clang@14.0.0+rocm amdgpu_target=gfx90a
^umpire@6.0.0%clang@14.0.0+rocm amdgpu_target=gfx90a
^magma@2.6.2+rocm~cuda amdgpu_target=gfx90a
```

Figure 40: `spack` specification for building ExaGO on Crusher

```
exago@develop+cuda+hiop+ipopt+mpi+raja cuda_arch=70
^hiop@develop+cuda+cusolver+deepchecking+kron+mpi+raja+sparse cuda_arch=70
^openblas@0.3.20
^magma@2.6.2+cuda
^raja@0.14.0+cuda cuda_arch=70
^umpire@6.0.0+cuda cuda_arch=70
^ipopt@3.12.10+coinhsl+metis~mumps
^petsc@3.16.6
^openmpi@4.1.3
```

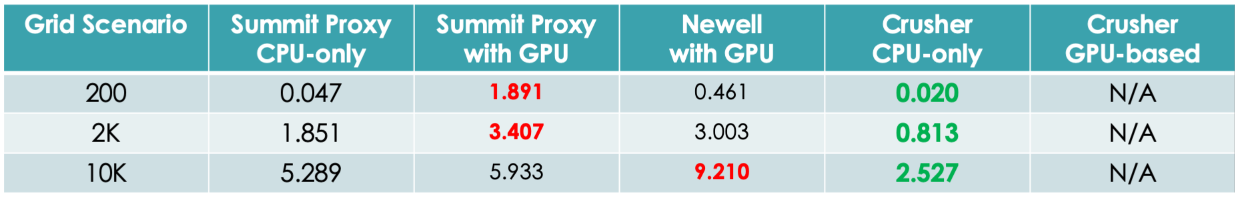
Figure 41: `spack` specification for building ExaGO on Summit Proxy

H.2 Performance Analysis of ExaGO with Sparse Solvers

We have measured the run time of ExaGO with sparse solvers using Summit-Proxy, Newell and Crusher systems and compared the results with those from using the dense GPU solvers.

Here, we first present the performance of sparse solvers across different machines and platforms. All the experiment were done with the most recent development version of ExaGO with the `opflow` example. We used both CPU and GPU-based (cusolver only on NVIDIA GPUs) sparse solvers in this experiment. The details of the results are shown in Figure 42. The results show that the Crusher CPU performance outperforms all CPU/GPU execution on other platforms. Note that the outputs of the CPU and GPU solvers are effectively the same. At the time of this writing, the corresponding sparse GPU solvers are not available for use with ExaGO for the AMD GPU architectures, and hence performance comparison is currently not possible to complete for the Crusher system.

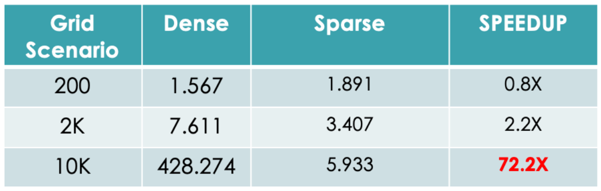
Next, we compare the performance of ExaGO using dense and sparse GPU solvers. Figure 43 shows the performance of ExaGO with dense and sparse GPU solvers on Summit-Proxy. As seen from the table, for larger problem sizes, sparse solver outperforms dense solvers significantly (in this case by a factor of 72×). Both solvers produce the same results when configured with identical



Grid Scenario	Summit Proxy CPU-only	Summit Proxy with GPU	Newell with GPU	Crusher CPU-only	Crusher GPU-based
200	0.047	**1.891**	0.461	**0.020**	N/A
2K	1.851	**3.407**	3.003	**0.813**	N/A
10K	5.289	5.933	**9.210**	**2.527**	N/A

Figure 42: ExaGO with Sparse solver on 1-GPU of Different Platforms

initialization settings and options. The high speed of the sparse solvers resulting in the same optimization results indicates that sparse solvers would be very useful for solving the grid optimization problem very efficiently on these systems.



Grid Scenario	Dense	Sparse	SPEEDUP
200	1.567	1.891	0.8X
2K	7.611	3.407	2.2X
10K	428.274	5.933	**72.2X**

Figure 43: ExaGO Dense vs. Sparse on 1 GPU of Summit Proxy

H.3 Performance Profiles of Latest HiOp Versions

We executed the latest HiOp version from the development branch `NlpPriDecEx2` on Summit proxy and compared its results against our previous results taken in August 2021. The comparison is shown in Figure 44. As seen from the figure, the latest runs are significantly faster due to early convergence. For the larger input size, the number of iterations significantly decreased from 199 to 26. Furthermore, there is a slight reduction in time per iteration. Combined with both improvements the application is $8 times$ faster on Summit, particularly for the large matrix size,.

H.4 Performance Profiles of Latest ExaGO versions

We profiled the latest version of ExaGO with the following new configurations:

- Deep profiling of ExaGO with Sparse GPU Solvers (`cusparse` on Summit-Proxy)
- Multi-Node Multi-GPU ExaGO on Summit.

H.4.1 Deep profiling of ExaGO with Sparse GPU Solvers on 1-GPU of Summit Proxy

We measured the FLOP performance of ExaGO with sparse solvers by deep profiling. We used a medium sized input and a large input to compute the respective change in performance.

For the medium input of 2K bus size we have the following findings:

- Peak FLOP Rate of a kernel
  - Kernel: `pegasus_csrmvDLL_kernel`
  - 156.38 GFLOP/sec (0.33% of total GPU Time)
  - **Only 2.25% of GPU Capacity**
- FLOP Rate of computationally intensive Kernels

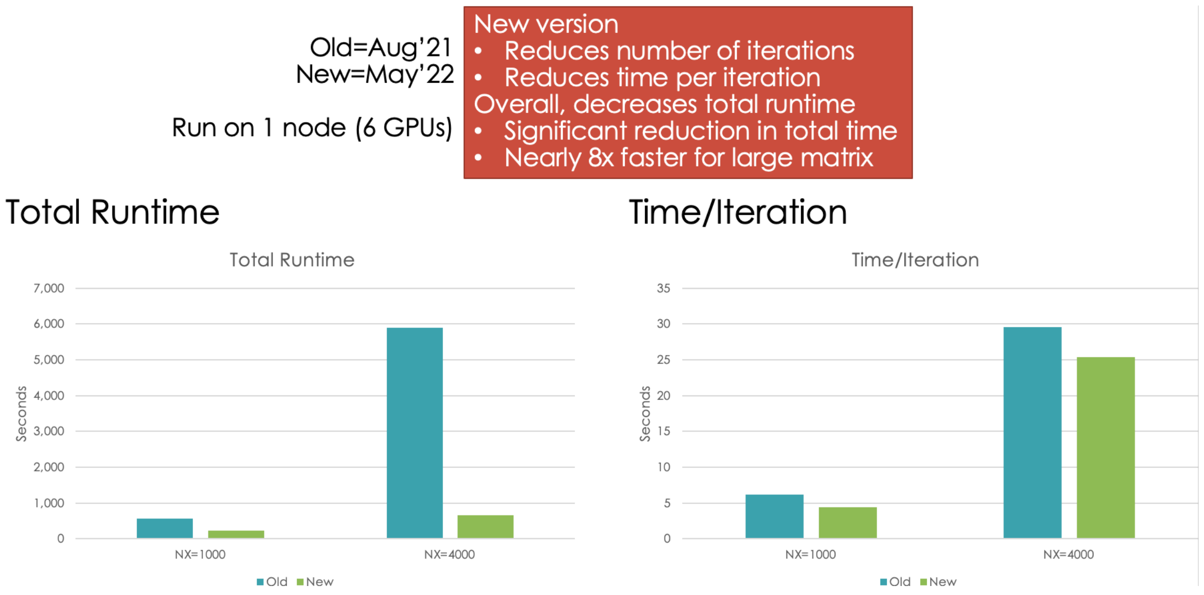


Figure 44: Timing of HiOp Parallel, Dense: Improvement on Summit

1. Kernel: `glu_lcsrsv2_solve_upper_nontranspose_kernel`
   – 6.37 GFLOP/sec (32.7% of total GPU Time)
2. Kernel: `glu_csrsv2_solve_upper_nontranspose_kernel`
   – 0.70 GFLOP/sec (32.51% of total GPU Time)

For the larger input of 10K bus size we have the following findings:

- Peak FLOP Rate of a kernel
  - Kernel: `pegasus_csrmvDLL_kernel`
  - 651.14 GFLOP/sec (0.22% of total GPU Time)
  - **Only 10% of GPU Capacity**
- FLOP Rate of computationally intensive Kernels
  1. Kernel: `glu_lcsrsv2_solve_upper_nontranspose_kernel`
     – 71.89 GFLOP/sec (17.57% of total GPU Time)
  2. Kernel: `glu_tile_schur_csr_csr_csr_kernel_v2`
     – 0.70 GFLOP/sec (14.63% of total GPU Time)

These performance results indicate that GPU utilization is relatively poor compared to the GPU capacity. However, as the problem size increases, the utilization gets better.

H.4.2 Multi-Node Multi-GPU ExaGO on Summit

We exercised a multi-node multi-gpu ExaGO execution using the `scopflow` application on Summit. Figure45 shows the weak scaling plot of the system. This is our first attempt running the Multi-GPU version of ExaGO. Additional detailed analyses of this important hardware-software combination is ongoing.

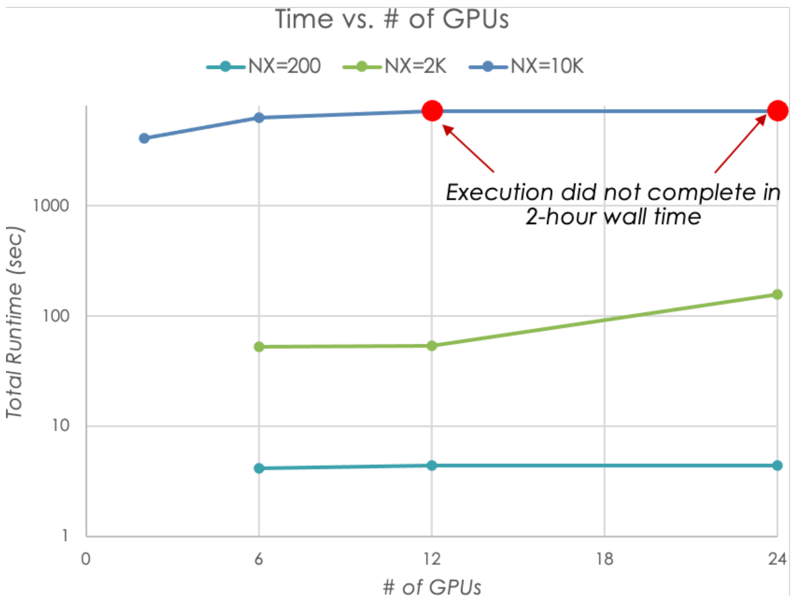


Figure 45: Multi-Node, Multi-GPU ExaGO (SCOPFlow) on Summit

H.5 Performance on Summit and Crusher (pre-Frontier) Architecture

We exercised a Multi-Node Multi-GPU execution of HiOp using the example `NlpPriDecEx2` on both Summit and Crusher. The results of the weak scaling is shown in Figure 46. As shown in the figure, the Crusher runs exhibit linear weak scaling whereas increasing the GPUs seems to affect the weak scaling performance of Summit.

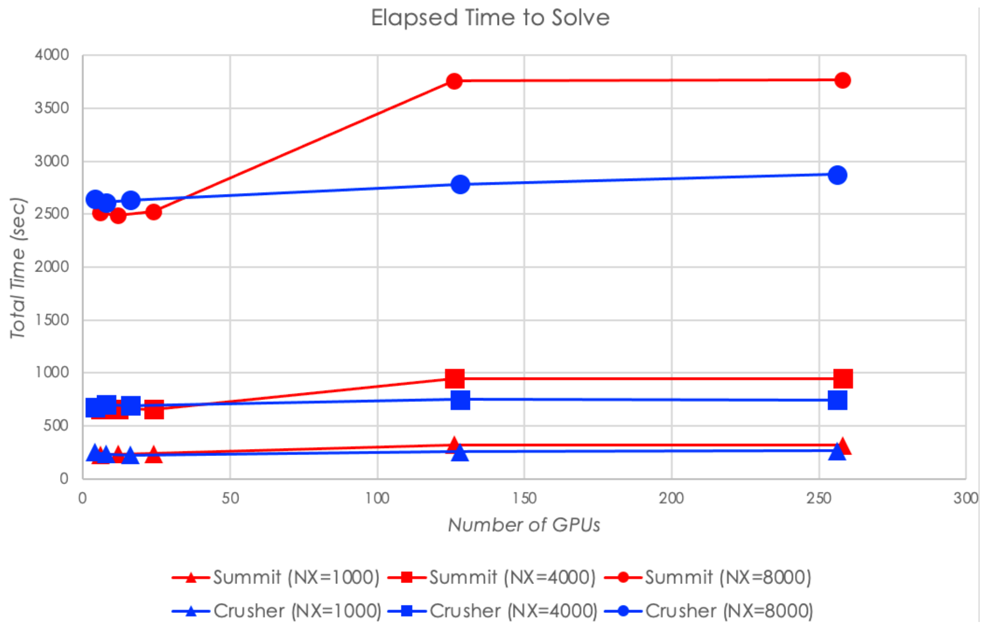


Figure 46: Parallel Dense HiOp (Pridec2) on Summit and Crusher

The results indicate an more predictable and well behaved scaling trend on the Crusher system, with excellent weak scaling exhibited as the number of GPUs is increased, spanning multiple nodes.

H.6 Preliminary Roofline Performance Results from Crusher (pre-Frontier) Machine

We used the HiOp (development branch) example `NlpPriDecEx2` on 4 GPUs in a single node on Crusher to execute the deep profiling. We used the parameter settings of `NX=100, S=3` for the experiment. For the profiling we used ROCM.4.5.0.

The key findings of profiling on Crusher are given below:

- It is observed that the currently observed performance has not yet reached the advertised peak performance for the GPU. Note that this lower efficiency is consistent with the observations in other applications as well. To empirically ascertain the peak performance, we used a set of publicly available micro-kernels to benchmark Crusher's performance and found the following advertised-peak vs. observed metrics:
  - Peak FP64 FLOP/sec is 18,421 (Observed) vs. 47,900 (Advertised) at 40% efficiency
  - Peak Bandwidth (GB/s) is 888.5 (Observed) vs. 3,276.8 (Advertised) at 27% efficiency
  - FLOP/Instruction is 1.97 Avg. FLOP per Instruction.

Due to lack of native FLOP counters, we used instruction-based roofline (as we previously reported on other pre-Frontier machines such as Tulip and Spock that are based on the AMD architectures). The instruction-based roofline measures the performance of the kernels as shown in Figure 47. The roofline indicates that our kernels do not appear to be running closer to capacity.

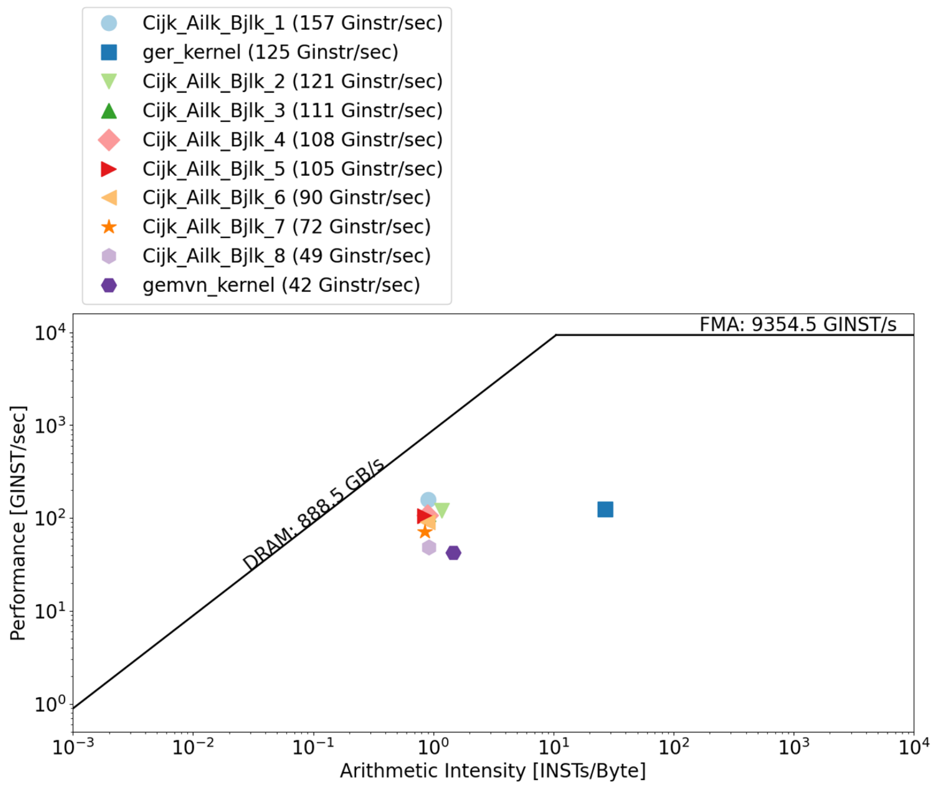


Figure 47: ExaGO Top 10 Kernels by GInstr/Sec per GPU