Reproduced Computational Results Report for "Ginkgo: A Modern Linear Operator Algebra Framework for High Performance Computing"

CODY J. BALOS, Lawrence Livermore National Laboratory

The article titled "Ginkgo: A Modern Linear Operator Algebra Framework for High Performance Computing" by Anzt et al. presents a modern, linear operator centric, C++ library for sparse linear algebra. Experimental results in the article demonstrate that Ginkgo is a flexible and user-friendly framework capable of achieving high-performance on state-of-the-art GPU architectures.

In this report, the Ginkgo library is installed and a subset of the experimental results are reproduced. Specifically, the experiment that shows the achieved memory bandwidth of the Ginkgo Krylov linear solvers on NVIDIA A100 and AMD MI100 GPUs is redone and the results are compared to what presented in the published article. Upon completion of the comparison, the published results are deemed reproducible.

CCS Concepts: • **Software and its engineering** → **Software creation and management**; • **Mathematics of computing** → *Mathematical software*; • **Computing methodologies** → *Massively parallel algorithms*

Additional Key Words and Phrases: High performance computing, healthy software lifecycle, multicore and manycore architectures

ACM Reference format:
Cody J. Balos. 2022. Reproduced Computational Results Report for "Ginkgo: A Modern Linear Operator Algebra Framework for High Performance Computing". *ACM Trans. Math. Softw.* 48, 1, Article 3 (February 2022), 7 pages.
https://doi.org/10.1145/3480936

1 INTRODUCTION

In [2], Anzt et al. introduce a modern C++ library for sparse linear operator algebra, Ginkgo. To demonstrate flexibility and performance, the authors conduct several experiments on GPU architectures relevant to today's supercomputers and offer performance comparisons between the Ginkgo implementation of certain algorithms and GPU vendor implementations.

In this report, we focus on reproducing a subset of the results presented by Anzt et al. in [2]; specifically, we reproduce the solver memory bandwidth data and plots given in Figures 8 and 9. The results portrayed by Figure 8 of [2] are generated using an NVIDIA A100 GPU with 40 GB of

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under contract DE-AC52-07NA27344. Lawrence Livermore National Security, LLC. LLNL-JRNL-823784.
This research used resources of the Oak Ridge Leadership Computing Facility, which is a DOE Office of Science User Facility supported under Contract DE-AC05-00OR22725.
Authors' address: Cody J. Balos, Lawrence Livermore National Laboratory, 7000 East Ave, Livermore, CA 94550; email: balos1@llnl.gov.
Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2022 Association for Computing Machinery.
0098-3500/2022/02-ART3 $15.00
https://doi.org/10.1145/3480936

memory and the CUDA programming model [5], while the results portrayed by Figure 9 of [2] are generated using an AMD MI100 GPU with 32 GB of memory and the HIP programming model [1]. The general steps we use to reproduce the experimental results are the same for both GPUs:

(1) Install the ssget tool and prefetch test matrices from the SuiteSparse collection.
(2) Download and build the Ginkgo library.
(3) Generate the scripts needed to execute the experiments.
(4) Upload the experiments to a git repository and use the Ginkgo Performance Explorer interactive website to generate the figures.

The published manuscript [2] utilized the Tulip early access system for the Frontier supercomputer to benchmark Ginkgo, but in this report, we utilize the Spock early access system sited at Oak Ridge National Laboratory [7]. The Spock system has AMD MI100 GPUs, like Tulip [2], so the results we generate will be comparable to the original. For reproducing the NVIDIA A100 results, we utilize the Guyot system hosted by the **Innovative Computing Laboratory** (**ICL**) at the University of Tennessee, Knoxville. While Guyot has NVIDIA A100 GPUs, they are slightly different from the A100 GPUs used in the published article. In particular, the Guyot A100 GPUs have 80 GB of memory instead of 40 GB and 2039 GB/s of memory bandwidth vs 1555 GB/s [6]. All of the other specifications of the A100 GPUs in Guyot match the A100 GPUs used to generate the published experimental data. Therefore, we should still be able to reproduce the results within a reasonable margin, and we only have one variable, the increased memory bandwidth, which we must account for in our analysis.

2 REPRODUCTION OF EXPERIMENTAL RESULTS

Anzt et al. provide detailed steps to reproduce Figures 8 and 9 and their relevant data in Annex A of the published article [2]. We follow the instructions exactly (including using the same software versions), except where noted, and as such, we only summarize most of the steps below. Steps that differ for the A100 and MI100 experiments are also noted.

2.1 Preparing to Execute the Experiments

2.1.1 Clone the ssget Tool. The ssget tool is cloned from the git repository https://github.com/ginkgo-project/ssget to `$HOME/TOMS-gko-reproduce`. The `ARCHIVE_LOCATION` variable on line 39 of the `ssget.sh` script is set to be `$HOME/TOMS-gko-reproduce` on Guyot. On Spock, the variable is set to a directory, `/gpfs/alpine/<projid>/scratch/<username>`, that resides on the Spock parallel file system. We add the location of the ssget script to our `$PATH` variable.

2.1.2 Use ssget to Download the SuiteSparse Matrices Needed for the Experiments. Following the author-provided instructions, we create a file with a list of 10 relevant matrix IDs from the SuiteSparse Matrix Collection [3]. The ssget script is then used to fetch each of the matrices.

2.1.3 Clone and Build Ginkgo. On Guyot, we follow the author-provided instructions to clone Ginkgo from https://github.com/ginkgo-project/ginkgo.git and build it. We change line 3 of Listing 13 in the published article to load the equivalent modules on Guyot. We also omit line 15 of Listing 13, since Guyot does not require us to submit a job through a job scheduler. The available CUDA version on Guyot is a slightly newer version than used by Anzt et al. (CUDA 11.0.221 instead of 11.0.194). With these modifications, we successfully built Ginkgo on Guyot and all unit tests run by `make test` passed.

On Spock, first, we modified Listing 13 for HIP use as the authors instructed. The exact script we use is given in Listing 1. On the Spock system, the ROCM version used by Anzt et al., 4.0.20496, is not available. Instead, we utilize ROCM 4.1.0, which requires us to modify line 157 of the CMake-

Lists.txt in the root directory of the Ginkgo source code. Specifically, we change the line to read `if (GINKGO_HIP_PLATFORM MATCHES "hcc|amd")`. After these steps were completed, running `make -j10` successfully compiled Ginkgo, and all tests run by `make test` passed as well.

```
ginkgo_source=$HOME/TOMS-gko-reproduce/ginkgo
ginkgo_build=/gpfs/alpine/<projid>/scratch/<username>/TOMS-gko-reproduce/
    ginkgo-build
module load craype-accel-amd-gfx908 rocm cmake git

# For every new session, the previous setup is required
# git clone https://github.com/ginkgo-project/ginkgo.git ${ginkgo_source} --
    branch master
mkdir -p ${ginkgo_build} && cd ${ginkgo_build}
cmake -DGINKGO_BUILD_CUDA=off -DGINKGO_BUILD_HIP=ON -DGINKGO_BUILD_OMP=off -
    DGINKGO_BUILD_EXAMPLES=off -DGINKGO_BUILD_TESTS=on -DGINKGO_DEVEL_TOOLS=
    off -DCMAKE_CXX_COMPILER=$(which hipcc) ${ginkgo_source}

# Compilation can happen either directly or through a job depending on the
# system policies.
srun -N 1 -A <projid> -p <partition> --gres=gpu:1 --time=3:00:00 --export=ALL
     make -j10
make -j10 # afterwards, ensure everything is compiled
make test
# Everything should run without failure. If cuda tests fail logging
# in again might solve some issue, this could be due to the hardware
# restrictions on summit after 4 hours of login time
```

Listing 1. A script to download and build the Ginkgo library to reproduce the MI100 experiments on Spock with HIP.

2.1.4 Clone and Build BabelStream. Since the NVIDIA A100 GPUs on Guyot have a higher theoretical memory bandwidth than the A100 GPUs utilized by Anzt et al. in [2], we need to run the BabelStream [4] TRIAD benchmark on Guyot. We clone the BabelStream git repository (https://github.com/UoB-HPC/BabelStream.git) to `$HOME/TOMS-gko-reproduce/babelstream` and then follow the documentation to build it with CUDA enabled. After building, we end up with the cuda-stream script in the `$HOME/TOMS-gko-reproduce/babelstream/build` directory.

2.2 Executing the Experiments

To execute the A100 experiments on Guyot, we do not need to use a SLURM batch script like the authors instructed in the article (so we skip the steps in Listings 14 and 15 of [2]). Instead, we execute the benchmarks directly by navigating to the directory `${ginkgo_build}/benchmark`, where `${ginkgo_build}` is the directory where Ginkgo was built earlier, and then running the command `sh run_all_benchmarks.sh`. As in the published article, the experiments employ a single NVIDIA A100 GPU.

For the MI100 experiments that we run on Spock, a SLURM batch script is required. We utilize Listing 14 of [2] as a guide to create the batch script. Due to the queue limitations, we cannot run all of the experiments as a single batch job. As such, we split the experiments into two jobs by splitting the matrices list file from Section 2.1.2 into two files, each with five matrix IDs. Our exact batch script is given in Listing 2. Note that each batch job uses a single AMD MI100 GPU for the experiments.

In addition to the Ginkgo experiments, we also run the BabelStream TRIAD benchmark on Guyot. To do this, we navigate to `$HOME/TOMS-gko-reproduce/babelstream/build`, execute the

command `./cuda-stream --triad-only`, and then note the output bandwidth (1721 GB/s) for later use.

```
#!/bin/bash
#SBATCH --nodes=1
#SBATCH --ntasks=1
#SBATCH --exclusive
#SBATCH --gres=gpu:1
#SBATCH --time=3:00:00
#SBATCH --account=<account>
#SBATCH --partition=<partition>
#SBATCH --export=ALL

cd ${ginkgo_build}/benchmark
make -j10
chmod +x run_all_benchmarks.sh
# choose which part of the experiment to run in this job
export MATRIX_LIST_FILE=$HOME/TOMS-gko-reproduce/matrices.list.part1
#export MATRIX_LIST_FILE=$HOME/TOMS-gko-reproduce/matrices.list.part2
export SOLVERS_PRECISION=1e-200
export SYSTEM_NAME=MI100_solvers
export EXECUTOR=hip
export BENCHMARK=solver
export FORMATS="coo"
exec ./run_all_benchmarks.sh
```

Listing 2. The SLURM batch script we utilize to launch the MI100/HIP experiments on Spock.

2.3 Evaluating the Reproduced Results

The result of the experiment batch jobs is a set of JSON files, one for each test matrix, containing timing information for five of the different Krylov linear solver methods available in Ginkgo: BiCGSTAB, CG, CGS, FCG, and GMRES. To evaluate the reproduced results and recreate Figures 8 and 9 of [2], we utilize the **Ginkgo Performance Explorer** (**GPE**) plotting tool (https://ginkgo-project.github.io/gpe/) as discussed by Anzt et al. in Annex A.4 of their article.

Continuing to follow the authors' instructions, we first fork the ginkgo-data (https://github.com/ginkgo-project/ginkgo-data) git repository. To publish the results, we use Listing 16 from [2] with one change: For the A100 results from Guyot, we publish the files to the TOMS-interface branch of our ginkgo-data fork (https://github.com/balos1/ginkgo-data), but for the MI100 results from Spock, we publish the files to a branch named TOMS-interface-hip. After publishing, we use the GPE to inspect our reproduced results. Again, we simply follow the instructions of Anzt et al. to load the reproduced results into the GPE.

In order to recreate the plot with the MI100 results, we modify the plotting script `plots/solver-bandwidth.plot.jsonata` in our fork of ginkgo-data on the TOMS-interface-hip branch so that MI100 results are plotted instead of the A100 results. We provide the exact changes needed as a diff in Listing 3.

Upon inspection of the results generated from Guyot, we note that the achieved memory bandwidth for the thermal2, atmosmodj, and StocF1465 matrices are comparable to the results generated by Anzt et al [2]. However, for the other matrices, we note that our reproduced results achieve a memory bandwidth that is approximately 15% higher than original results. As we pointed out in Section 1, the NVIDIA A100 GPU available on Guyot has a higher peak memory bandwidth than the NVIDIA A100 GPU utilized by Anzt et al. Indeed, our run of the BabelStream TRIAD benchmark indicated that the Guyot A100 could achieve a 19% higher memory bandwidth than the other

A100 (1,732 GB/s vs 1,399 GB/s). Furthermore, the vectors associated with the first three matrices are small enough to fit into the cache of the Guyot A100, which is the same size as the cache of the A100 used in the published article (so the increased memory bandwidth has a limited effect) [2]. Therefore, the discrepancy with the seven larger matrices is not surprising, and we look at performance as a percentage of the STREAM bandwidth instead. When we do this comparison, we see that the published results and newly generated results are similar as both achieve between 65% and 85% of the STREAM bandwidth recorded for their respective version of the A100.

```
diff --git a/plots/solver-bandwidth.plot.jsonata b/plots/solver-bandwidth.
    plot.jsonata
index 09af1c0580..bdc349de70 100644
--- a/plots/solver-bandwidth.plot.jsonata
+++ b/plots/solver-bandwidth.plot.jsonata
@@ -2,7 +2,7 @@
 $data_hip := content[dataset.executor="hip" and dataset.system="
    MI100_solvers"];
 $data_cuda := content[dataset.executor="cuda" and dataset.system="
    A100_solvers"];

-$solvers := $data_cuda.solver~>$keys();
+$solvers := $data_hip.solver~>$keys();

 $matrices := [
 "thermal2",
@@ -46,7 +46,7 @@ $memops := function($name, $n, $nnz, $it, $k, $vtype,
    $itype) {

 $plot := $solvers~>$map(function($v, $i) {{
 "label": $v,
- "data": $data_cuda.{
+ "data": $data_hip.{
 "x": problem.name,
 "y": $memops($v, problem.rows, problem.nonzeros, (solver~>$lookup($v)).apply
    .iterations, 100, 8, 4) / ((solver~>$lookup($v)).apply.time/1000000000)
 },
@@ -58,7 +58,7 @@ $plot := $solvers~>$map(function($v, $i) {{
 $bw := $matrices~>$map(function($v, $i) {
- $get_bandwidth("cuda")
+ $get_bandwidth("hip")
 });

 {
@@ -82,7 +82,7 @@ $bw := $matrices~>$map(function($v, $i) {
 },
 "title": {
 "display": true,
- "text": "Bandwidth of selected problems for Ginkgo solvers on A100(CUDA)",
+ "text": "Bandwidth of selected problems for Ginkgo solvers on MI100(Hip)",
 "fontSize": 20
 },
 "tooltips": {
```

Listing 3. A diff showing the changes made to the ginkgo-data plotting script in order to plot the MI100/HIP results.

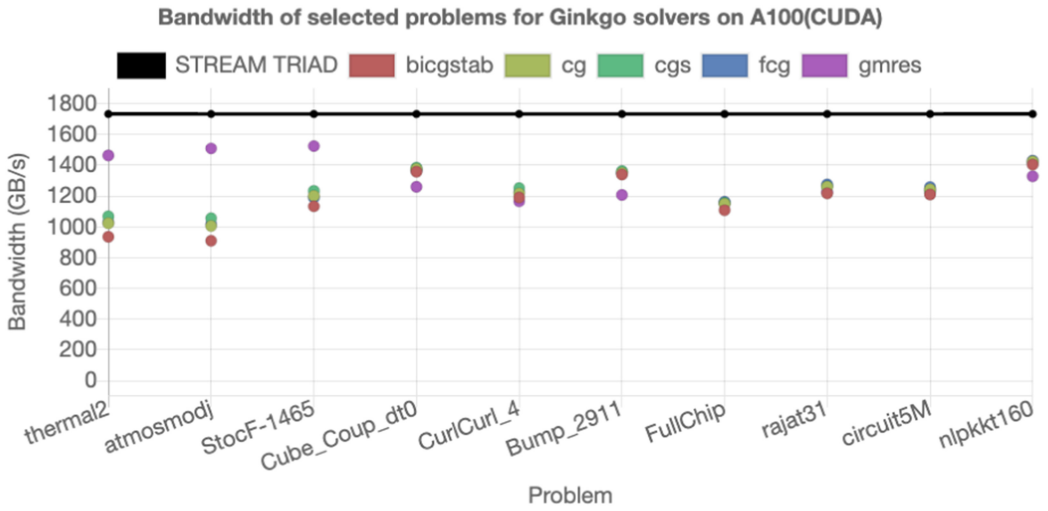

Fig. 1. Reproduction of Figure 8 from Anzt et al. [2] using the Guyot system. The plot shows the memory efficiency of Ginkgo's Krylov solvers on an NVIDIA A100 GPU. The matrices are sorted by number of rows. When compared to the published results, the new ones are similar once the increased memory bandwidth of the Guyot A100 GPU is accounted for.

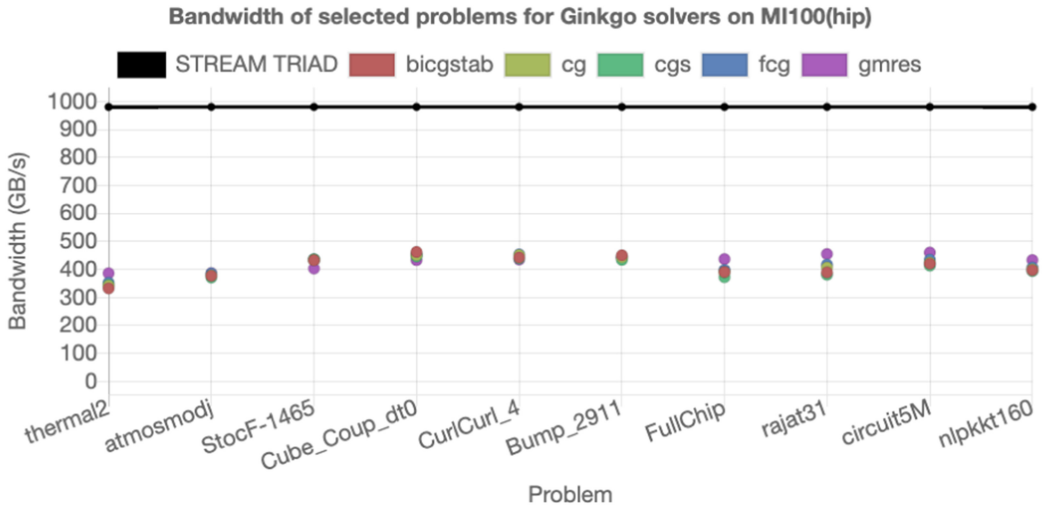

Fig. 2. Reproduction of Figure 9 from Anzt et al. [2] using the Spock system. The plot shows the memory efficiency of Ginkgo's Krylov solvers on the AMD MI100 GPU. The matrices are sorted by number of rows. These newly reproduced results are nearly identical to the published results.

Examining the reproduced results we generated with an AMD MI100 GPU on Spock, we see that they are nearly identical to the memory bandwidth values presented by Anzt et al. in [2]. For all of the matrices, roughly 50% of the STREAM bandwidth is achieved.

Figures 1 and 2 show our reproductions of Figures 8 and 9 of [2]. Since both the results generated with the NVIDIA A100 GPU and AMD MI100 GPU are comparable to the published results, we consider the published results successfully reproduced.

3 CONCLUSION

Anzt et al. present Ginkgo, a modern C++ sparse linear operator algebra framework that targets GPU accelerators, in [2]. Using the instructions provided by the authors in Annex A of [2], we were able to install all of the software needed on two different computers: Guyot and Spock. We utilized Guyot for experiments that require CUDA and an NVIDIA A100 GPU and Spock for experiments that required HIP and an AMD MI100 GPU. All of the software was freely and openly available. Once the software was installed, we were able to continue using the instructions provided by the authors, with only a few deviations, to reproduce the experimental data. Finally, we utilized the Ginkgo Performance Explorer plotting tool to recreate the plots in Figures 8 and 9 of the published article [2]. After analyzing the newly generated results and comparing them to the published results, the reviewer deems the published results to be reproducible.

ACKNOWLEDGMENTS

This document was prepared as an account of work sponsored by an agency of the United States government. Neither the United States government nor Lawrence Livermore National Security, LLC, nor any of their employees makes any warranty, expressed or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States government or Lawrence Livermore National Security, LLC. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States government or Lawrence Livermore National Security, LLC, and shall not be used for advertising or product endorsement purposes.

REFERENCES

[1] AMD ROCm. 2021. HIP Programming Guide, v4.1.0. Retrieved June 18, 2021 from https://rocmdocs.amd.com/en/latest/Programming_Guides/Programming-Guides.html.

[2] Hartwig Antz, Terry Cojean, Goran Flegar, Fritz Göbel, Thomas Grützmacher, Pratik Nayak, Tobias Ribizel, Yushiang Mike Tsai, and Enrique S. Quintana-Ortí. [n.d.]. Ginkgo: A modern linear operator algebra framework for high performance computing. *ACM Transactions on Mathematical Software* ([n. d.]). In Press.

[3] Timothy A. Davis and Yifan Hu. 2011. The University of Florida sparse matrix collection. *ACM Transactions on Mathematical Software.* 38, 1, Article 1 (Dec. 2011), 25 pages. DOI: https://doi.org/10.1145/2049662.2049663

[4] Tom Deakin, James Price, Matt Martineau, and Simon McIntosh-Smith. 2018. Evaluating attainable memory bandwidth of parallel programming models via BabelStream. *International Journal of Computational Science and Engineering* 17, 3 (2018), 247–262.

[5] NVIDIA, Péter Vingelmann, and Frank H. P. Fitzek. 2020. CUDA, release: 11.0. Retrieved June 18, 2021 from https://docs.nvidia.com/cuda/archive/11.0/index.html.

[6] NVIDIA Corporation. 2021. NVIDIA A100. Retrieved June 18, 2021 from https://www.nvidia.com/en-us/data-center/a100/.

[7] Oak Ridge National Laboratory. 2021. Spock Quick-Start Guide. Retrieved June 18, 2021 from https://docs.olcf.ornl.gov/systems/spock_quick_start_guide.html.

Received June 2021; accepted August 2021