Preparing Ginkgo for AMD GPUs – A Testimonial on Porting CUDA Code to HIP

Yuhsiang M. Tsai[1], Terry Cojean[1], Tobias Ribizel[1], and Hartwig Anzt[1,2]

[1] Karlsruhe Institute of Technology, Karlsruhe, Germany
[2] University of Tennessee, Innovative Computing Lab, Knoxville, TN, USA
firstname.lastname@kit.edu

Abstract. With AMD reinforcing their ambition in the scientific high performance computing ecosystem, we extend the hardware scope of the Ginkgo linear algebra package to feature a HIP backend for AMD GPUs. In this paper, we report and discuss the porting effort from CUDA, the extension of the HIP framework to add missing features such as cooperative groups, the performance price of compiling HIP code for AMD architectures, and the design of a library providing native backends for NVIDIA and AMD GPUs while minimizing code duplication by using a shared code base.

Keywords: Portability; GPU; CUDA; HIP

1 Introduction

Over the last decade years, GPUs have been established as the main powerhouse in leadership supercomputers. GPUs have proven valuable components to accelerate computations not only for machine learning workloads, but also for numerical linear algebra libraries powering computational science. As of today, AMD and NVIDIA are considered the main GPU manufacturers. In the past, software efforts primarily focused on NVIDIA GPUs due to the comprehensive CUDA development environment and the common adoption in HPC centers. With the next leadership supercomputers deployed in the US National Laboratories being equipped with AMD GPUs, and the US Exascale Computing Project's mission to provide math library functionality on the leadership systems, we extend the scope of the Ginkgo library to feature an AMD GPU backend.

In this paper, we report and discuss the effort of porting a CUDA-focused library to the HIP ecosystem. We elaborate on the use of the perl-based script provided by AMD that aims at simplifying the transition process, its pitfalls and flaws. We also assess the performance HIP-based code achieves on NVIDIA architectures when compiled using NVIDIA's `nvcc` compiler.

Transitioning a code base from one architecture to another, and platform portability in general, is an important problem in the software technology ecosystem. In particular, the number of adopters and contributors of community software scales only in the presence of good platform portability. The effort of porting

a software stack to new architectures is, for example, described for molecular dynamics algorithm in [5], and for the solution of finite element problems in [10]. Concerning performance portability, the authors of [9] compare the algorithm performance for CUDA, HC++, HIP, and OpenCL backends.

Compared to previous work, we highlight that this work contains the following novel contributions:

- We discuss the porting of linear algebra kernels from CUDA to HIP.
- We add technology to the HIP ecosystem that is lacking but needed, e.g., a subwarp cooperative group concept with shuffle operations.
- We compare the performance of HIP and CUDA kernels coming from the same code base and providing the same functionality.
- Up to our knowledge, Ginkgo is the first open-source sparse linear algebra library supporting several matrix types (Coo, Csr, Sellp, Ell, Hybrid), solvers (CG, BiCG, GMRES, etc), preconditioner (block-jacobi) and factorization (ParILU and ParILUT) on AMD and NVIDIA GPUs.
- We ensure full result reproducibility by archiving all performance results.

Before providing more details about the porting effort in Section 3, we recall some background information about CUDA and HIP in Section 2. We present the results of the experiments of the same kernels being compiled by CUDA and HIP in Section 4. We conclude in Section 5 with a summary of this paper.

2 Background

2.1 Compute Unified Device Architecture - CUDA

NVIDIA developed the CUDA programming model and the corresponding `nvcc` compiler enabling developers to write GPU-parallel programs using the C or C++ programming language. Also, NVIDIA provides several math libraries, like cuBLAS, cuSPARSE, and cuSOLVER containing ready-to-use numerical algorithms and core functionalities allowing users to easily develop a parallel application without writing device kernel functions.

In Listing 1.1, CUDA uses `__global__` as the declaration specifier to tell the compiler this function runs on a GPU and uses execution configuration syntax (`<<< >>>`) to represent the configuration of grid and block dimensions, execution stream, and dynamically-sized shared memory. Moreover, developers can provide additional information at compile-time to optimize the execution performance like `__launch_bounds__` to limit the register usage.

```
template <int value>
__global__ void dummy_kernel(const int num, int *__restrict__ array) {
  // kernel_code
}
int main() {
  // allocation of memory and calculation of grid/block_size
  dummy_kernel<4> <<<dim3(grid_size), dim3(block_size)>>>(num, array);
  return 0;
}
```

Listing 1.1: CUDA kernel launch syntax.

2.2 C++ Heterogeneous-Compute Interface for Portability - HIP

As a counterpart to NVIDIA's CUDA ecosystem, AMD more recently developed the GPU compute programming language and library ecosystem "RadeonOpenCompute" (ROCm). ROCm is the first open-source HPC platform for GPU computing shipping with several math libraries, like rocBLAS, rocSPARSE, rocSOLVER, etc. This enables users to develop GPU-ready applications in ROCm like in the CUDA ecosystem.

Aside from ROCm, AMD also provides a HIP abstraction that can be seen as a higher layer on top of the ROCm ecosystem, enveloping also the CUDA ecosystem. The idea behind HIP is to increase platform portability of software by providing an interface through which functionality of both, ROCm and CUDA can be accessed. Obviously, this would remove the burden of converting or rewriting code for different hardware architectures, therewith also reducing the maintenance effort for libraries supporting several backends.

In Listing 1.2, HIP uses the same declaration specifier `__global__` like CUDA, but a different execution configuration syntax. HIP handles kernels featuring template parameters with the macro HIP_KERNELS_NAME. Although HIP also provides the `__launch_bounds__` flag for kernel optimization, the effect differs from the CUDA ecosystem due to the architectural differences between AMD and NVIDIA GPUs.

```
template <int value>
__global__ void dummy_kernel(const int num, int *__restrict__ array) {
  // kernel_code
}
int main() {
  // allocation of memory and calculation of grid/block_size
  hipLaunchKernelGGL(HIP_KERNEL_NAME(dummy_kernel<4>), dim3(grid_size),
                     dim3(block_size), 0, 0, num, array);
  return 0;
}
```

Listing 1.2: HIP kernel launch syntax.

2.3 Difference between AMD and NVIDIA GPUs

The primary technical difference between AMD and NVIDIA GPUs is the number of threads that are executed simultaneously in a wavefront/warp. In NVIDIA GPUs, a warp contains 32 threads, in AMD GPUs, a wavefront contains 64 threads. This difference potentially impacts all other parameter configurations and has to be taken into account when designing kernels and setting thread block size, shared memory and register usage, and compute grid size for valid parameter settings and optimal kernel performance.

Less relevant for the kernel design and parameter choice is that AMD and NVIDIA GPUs differ in the number of multiprocessors accumulated in a single device and in the memory bandwidth. While these are still relevant for kernel optimization, they rarely impact the correctness of a kernel design. We elaborate on the optimization of kernel parameters in Section 3.5.

As of today, AMD's ROCm ecosystem – and the HIP development ecosystem – still lacks some key functionality of the CUDA ecosystem. For example, HIP lacks a cooperative group interface that can be used for flexible thread programming inside a wavefront, see Section 3.3.

3 Porting CUDA functionality to the HIP ecosystem

Next, we report and discuss how we ported Ginkgo's GPU functionality available for CUDA backends to the HIP ecosystem. To understand the technical realization, it is however useful to first elaborate on Ginkgo's design.

3.1 Ginkgo design

A high-level overview of Ginkgo's software architecture is visualized in Figure 1. The library design collects all classes and generic algorithm skeletons in the "core" library which, however, is useless without the driver kernels available in the "omp", "cuda", and "reference" folders. We note that "reference" contains sequential CPU kernels used to validate the correctness of the algorithms and as reference implementation for the unit tests realized using the googletest[4] framework. The "include" folder contains the public interface. Extending Ginkgo's scope to AMD architectures, we add the "hip" folder containing the kernels in the HIP language, and the "common" folder for platform-portable kernels with the intention to reduce code duplication, see Section 3.2.

To reduce the effort of porting Ginkgo to AMD architectures, we use the same base components of Ginkgo like `config`, `binding`, `executor`, `types` and `operations`, which we only extend and adapt to support HIP.

- `config`: hardware-specific information like warp size, lane_mask_type, etc.;
- `binding`: the C++ style overloaded interface to vendors' BLAS and sparse BLAS library and the exception calls of the kernels not implemented;
- `executor`: the "handle" controlling the kernel execution and the ability to switch the execution space (hardware backend);
- `types`: the type of kernel variables and the conversion between library variables and kernel variables;
- `operations`: a class aggregating all the possible kernel implementations such as reference, omp, cuda and hip, which allows to switch between implementations at runtime.

Moreover, some components are not officially supported by vendors, e.g. complex number atomic_add[3] on CUDA and HIP, and warp-wide cooperative groups on HIP. For the functionality missing in both vendor ecosystems, we implement CUDA device functions providing the functionality and apply the work flow

[3] A complex atomic_add involves separate real and imaginary atomic_add and thus is not strictly an atomic operation, as no ordering between the individual components of multiple complex atomic operations is guaranteed.

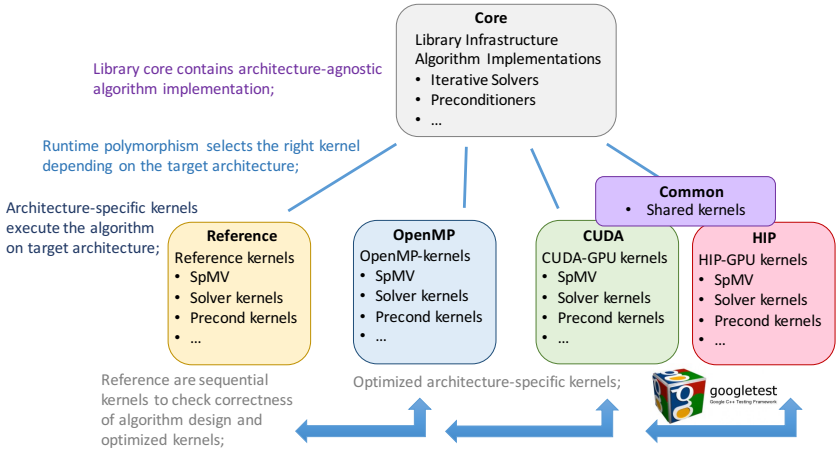


Fig. 1: The Ginkgo library design overview. The components added when extending the scope to AMD GPUs are the "HIP" and the "Common" modules.

listed in Algorithm 1 to generate corresponding HIP kernels. For components missing only in one vendor ecosystem, we implement kernels providing the same functionality in the other ecosystem. In particular, as the HIP ecosystem currently lacks the warp-wide cooperative groups we make heavy use of, we implement device functions that provides this functionality for AMD architectures, see Section 3.3.

3.2 Avoiding code duplication

Despite the fact that the HIP ecosystem allows to compile the kernels for both AMD and NVIDIA GPUs, we currently plan to still provide native support in the CUDA ecosystem. This choice is motivated by the wider adoption of CUDA in the high performance computing community on the one side, and the unclear future of this functionality remaining in the HIP ecosystem on the other side. A third reason is that preserving native CUDA support allows to utilize novel CUDA-specific technology, e.g., dynamic parallelism. Extending Ginkgo to AMD GPUs, a primary goal was to avoid a significant level of code duplication. For this purpose, we created the "common" folder containing all kernels and device functions that are identical or the CUDA and the HIP executor except for kernel configuration parameters (such as warp size or `launch_bounds`). These configuration parameters are not set in the kernel file contained in the "common" folder, but in the files located in "cuda" and "hip" that are interfacing these kernels. This way we can avoid code duplication while still configuring the parameters for optimal kernel performance on the distinct hardware backends.

3.3 Cooperative groups

CUDA 9 introduced cooperative groups for flexible thread programming. Cooperative groups provide an interface to handle thread block and warp groups and apply the shuffle operations that are used heavily in Ginkgo for optimizing sparse linear algebra kernels. HIP [1] only supports block and grid groups with `thread_rank()`, `size()` and `sync()`, but no subwarp-wide group operations like shuffles and vote operations.

For enabling full platform portability, a small codebase, and preserving the performance of the optimized CUDA kernels, we implement cooperative group functionality for the HIP ecosystem. Our implementation supports the calculation of size/rank and shuffle/vote operations inside subwarp groups. We acknowledge that our cooperative group implementation may not support all features of CUDA's cooperative group concept, but all functionality we use in Ginkgo.

The cross-platform cooperative group functionality we implement with shuffle and vote operations covers CUDA's native implementation. HIP only interfaces CUDA's warp operation without `_sync` suffix (which refers to deprecated functions), so we use CUDA's native warp operations to avoid compiler warning and complications on NVIDIA GPUs with compute capability 7.x or higher. We always use subwarps with contiguous threads, so we can use the block index to identify the threads' subwarp id and its index inside the subwarp. We define

$$\begin{aligned} \texttt{Size} &= \text{Given subwarp size} \\ \texttt{Rank} &= \texttt{tid \% Size} \\ \texttt{LaneOffset} &= \lfloor \texttt{tid \% warpsize / Size} \rfloor \times \texttt{Size} \\ \texttt{Mask} &= \sim 0 \texttt{ >> (warpsize - Size) << LaneOffset} \end{aligned}$$

where `tid` is local thread id in a thread block such that `Rank` gives the local id of this subwarp, and ~ 0 is a bitmask of 32/64 bits, same bits as `lane_mask_type`, filled with 1 bits according to CUDA/AMD architectures, respectively. Using this definition, we can realize the cooperative group interface, for example for the `shfl_xor`, `ballot`, `any`, and `all` functionality:

$$\begin{aligned} \texttt{subwarp.shfl\_xor(data, bitmask)} &= \texttt{\_\_shfl\_xor(data, bitmask, Size)} \\ \texttt{subwarp.ballot(predicate)} &= \texttt{(\_\_ballot(predicate) \& Mask) >> LaneOffset} \\ \texttt{subwarp.any(predicate)} &= \texttt{(\_\_ballot(predicate) \& Mask) != 0} \\ \texttt{subwarp.all(predicate)} &= \texttt{(\_\_ballot(predicate) \& Mask) == Mask} \end{aligned}$$

Note that we use the `ballot` operation to implement `any` and `all` operations. The original warp `ballot` returns the answer for the entire warp, so we need to shift and mask the bits to access the subwarp results. The `ballot` operation is often used in conjunction with bit operations like the population count (*popcount*), which are provided by C-style type-annotated intrinsics `__popc[ll]` in CUDA and HIP. To avoid any issues with the 64bit-wide lane masks on AMD GPUs, we provide a single function `popcnt` with overloads for 32 and 64 bit integers as well as an architecture-agnostic `lane_mask_type` that provides the correct (unsigned) integer type to represent a (sub)warp lane mask.

```
1 template <int Size, typename ValueType>
2 __global__ void reduce(ValueType *__restrict__ data, int inner_loops) {
3   auto local_data = data[threadIdx.x];
```

```
4     for (int i = 0; i < inner_loops; i++) {
5   +   auto group = group::tiled_partition<Size>(group::this_thread_block());
6       #pragma unroll
7   -   for (int bitmask = 1; bitmask < Size; bitmask <<= 1) {
8   +   for (int bitmask = 1; bitmask < group.size(); bitmask <<= 1) {
9   -     const auto remote_data = __shfl_xor(local_data, bitmask, Size);
10  +     const auto remote_data = group.shfl_xor(local_data, bitmask);
11        local_data = local_data + remote_data;
12      }
13    }
14    data[threadIdx.x] = local_data;
15  }
```

Listing 1.3: reduce kernel. Green part is cooperative group implementation, and red part is legacy implementation

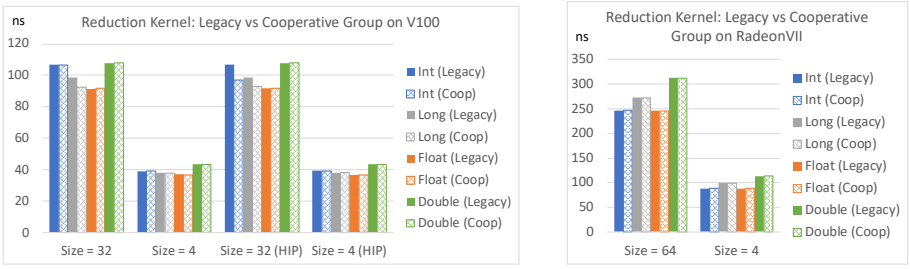


Fig. 2: Ginkgo's cooperative groups vs. legacy functions for different data types on V100 (left) and RadeonVII (right).

To assess the performance of our cross-platform cooperative group implementation, we use the local reduction kernel shown in Listing 1.3 that utilized either the vendor's legacy functionality (red) or Ginkgo's cross-platform cooperative group interface (green). In Figure 2, we report the runtime needed for 100 reduction operations (after a warm-up phase of 10 reductions) on NVIDIA's V100 GPU and AMD's RadeonVII GPU. To exclude the overhead of the kernel launch and memory operations, we run the kernel executing "inner_loops" reductions (line 4 of Listing 1.3) for "inner_loops = 1000" and "inner_loops = 2000" and report the runtime difference. This way, we can isolate the runtime needed for the warp-wide reduction by excluding the overhead of the kernel launch and memory operations. The results identifies Ginkgo's cross-platform cooperative group implementation as competitive to the vendor's native implementation. Both implementations use the same strategy for the reduction operation, and both implementations execute the reduction loop (line 7-12 of Listing 1.3) exactly $\log_2(Size)$ times. For the execution time for different values of $Size$, the theoretical performance ratios are $\frac{\log_2(4)}{\log_2(64)} = 0.333$ on the RadeonVII and $\frac{\log_2(4)}{\log_2(32)} = 0.4$ on the V100. In the experimental evaluation, we observe average

ratios $\frac{\text{runtime}(Size=4)}{\text{runtime}(Size=64)} = 0.360$ and $\frac{\text{runtime}(Size=4)}{\text{runtime}(Size=32)} = 0.394$ for the RadeonVII and the V100 GPUs, respectively.

3.4 Porting via the Cuda2Hip script

For easy conversion of CUDA code to the HIP language, we use a script based on the hipify-perl script provided by AMD with several modifications to meet our specific needs. First, the script generates the target filename including the path in the "hip" directory. Then AMD's hipify-perl script is invoked to translate the CUDA kernels to the HIP language, including the transformation of NVIDIA's proprietary library functions to AMD's library functions and the kernels launch syntax. Next, the script changes all CUDA-related header, namespace, type and function names to the corresponding HIP-related names. By default, the script hipify-perl fails to handle namespace definitions. For example, the hipify-perl script changes `namespace::kernel<<<...>>> (...)` to `namespace::hipLaunchKernelGGL(kernel, ...)`, while the correct output would be `hipLaunchKernelGGL(namespace::kernel, ...)`. Thus, the script ultimately needs to correct the namespaces generated by the hipify-perl script.

3.5 Porting workflow

In Algorithm 1, we sketch the workflow we use for porting Ginkgo's CUDA backend to HIP. Step 1 introduces a set of variables to represent the architecture-specific parameters such as the warp size (32 on CUDA devices, 64 on AMD devices) and optimization parameters. Step 2 moves the identical kernel codes into the "common" folder we introduced in Section 3.2. We include the code in the "common" folder after setting the configuration variables in Step 3 and Step 4. Step 5 runs the script Cuda2Hip script detailed in Section 3.4 to generate the corresponding hip files. Ultimately, we modify the hip "config" file in Step 6. After completion of these steps, the validity and correctness of the porting effort is tested. This is realized by invoking Ginkgo's unit test framework that employs googletest to check the correctness of the high performance kernels – in particular also the CUDA and HIP backends – against the reference kernels.

We note that Ginkgo's cross-platform cooperative group extension presented in Section 3.3 dramatically reduces backend-specific implementations and allows to use a shared kernel in "common" for both, the NVIDIA and the HIP backend.

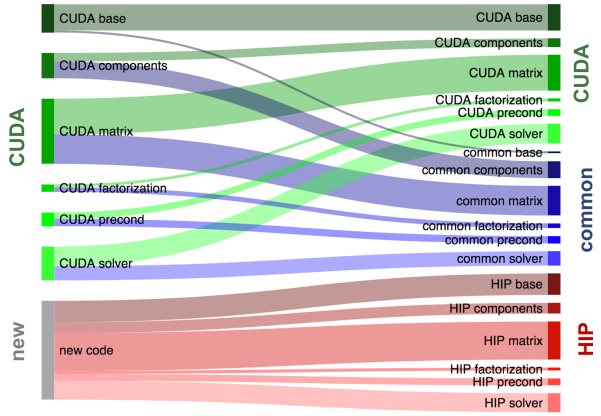
3.6 Porting statistics for Ginkgo

With the setup and tools described, extending the scope of Ginkgo to cover also AMD GPUs is a smooth process. We acknowledge that some kernels that are heavily tuned for performance needed additional attention, most notably the multiprecision block-Jacobi kernel [2]. Aside from this, the addition of the HIP ecosystem required slight modifications to the library architecture, most importantly the addition of the "common" module containing the kernels that

Algorithm 1: Ginkgo's porting workflow

1: Use a variable to represent the architecture-specific parameters
2: Move all shared code into a "common" file
3: Set the architecture-specific parameters before including a "common" file
4: Include the "common" file
5: Use the Cuda2Hip script for converting the code
6: Modify the hip file "config" to support different architectures

are identical up to parameter settings for the CUDA and the HIP ecosystems. In the left figure of Section 3.6, we visualize how existing code lines are relocated and new code lines are added when extending Ginkgo's scope to support also HIP. The exact number of code lines contained in the distinct modules of the extended Ginkgo library are listed in the right table of Section 3.6. We note that about one third of the code base is shared between the CUDA and the HIP executor, and that by creating the "common" folder we actually avoided duplicating 4,000 lines of code. The other modules each contain about 5,000 lines of code. While most submodules are comparable in size, the more significant differences for "base" and "component" stem from the differing comprehensiveness of the ecosystems and possibilities of architecture-specific optimization.



Module	common	cuda	hip
base	112	1435	1176
component	919	467	589
matrix	1617	1908	2048
factor	262	159	165
precond	395	356	375
solver	780	1071	1038

Fig. 3: Left: Reorganization of the Ginkgo library to provide a HIP backend for AMD GPUs. Right: (Physical) Lines of code in the "common", "cuda", and "hip" modules of the Ginkgo library, ignoring the unit tests.

4 Experiments

To assess how well the HIP ecosystem interfaces to the CUDA technology, we compare HIP code compiled for NVIDIA GPUs with native CUDA code. More

precisely, we apply the porting workflow we described in Section 3 to high performance sparse linear algebra kernels of Ginkgo's CUDA backend, and compare the performance of the generated HIP code when being compiled for NVIDIA GPUs with the original kernel performance. We run our experiments on NVIDIA's V100 (SXM2 16 GB) [6] with cuda 9.2.148 and hip 3.1.20044-3684ef8 (which is the latest version on Jan. 31 2020). We compare the Sellp, Coo, and cuSPARSE/hipSPARSE (Splib_Csr) SpMV kernels, and the Conjugate Gradient Solver employing the Sellp SpMV kernel for the Krylov subspace generation using either CUDA and HIP on the same device. For result reproducibility, we archive all performance results in a public repository [4]. We evaluate the performance of the Ginkgo SpMV for more than 2,800 matrices from the SuiteSparse Matrix Collection [8]. We run two iterations for warm-up and ten iterations to obtain average performance values.

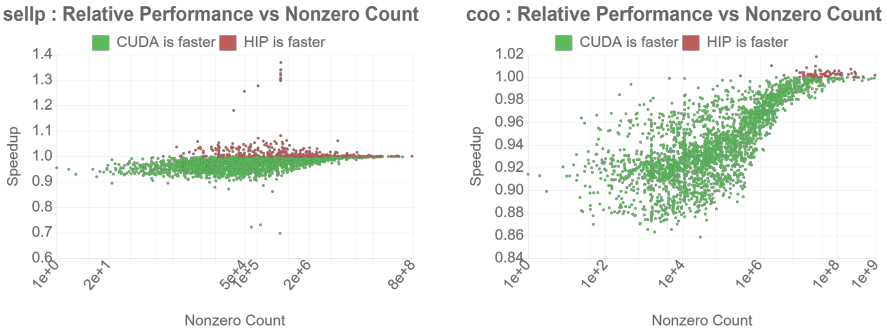


Fig. 4: Sellp SpMV (left) and Coo SpMV implemented in CUDA or HIP.

On the left-hand side of Figure 4, we evaluate the performance for Ginkgo's Sellp SpMV kernel, which does not use atomic operations. On the right-hand side of Figure 4, we do the same comparison for Ginkgo's Coo SpMV kernel which does rely on atomic operations. Running on NVIDIA's V100 GPU, one would expect to see small overhead of the HIP code interfacing CUDA code compared to native CUDA code. While this may prove true for most problems, we see some outliers where using the native CUDA implementation results in significant performance benefits. Surprisingly, for some test cases the HIP kernels achieve significantly better performance – even though HIP ultimately compiles with NVIDIA's `nvcc` compiler. The generated PTX code indicates that the differences may be attributed to slightly different types of `load` instructions being emitted, which in turn use different caches.

In Figure 5 we do the same experiment for the vendors' Csr SpMV (left-hand side) and 1,000 iterations of Ginkgo's Conjugate Gradient (CG) solver using Ginkgo's Sellp SpMV (right-hand side). For the vendors' Csr SpMV comparison on the left, the performance differences reflect only the overhead of the invocation of cuSPARSE by hipSPARSE. In the CG performance comparison on the right,

[4] https://github.com/ginkgo-project/ginkgo-data/tree/V100_cuda_hip

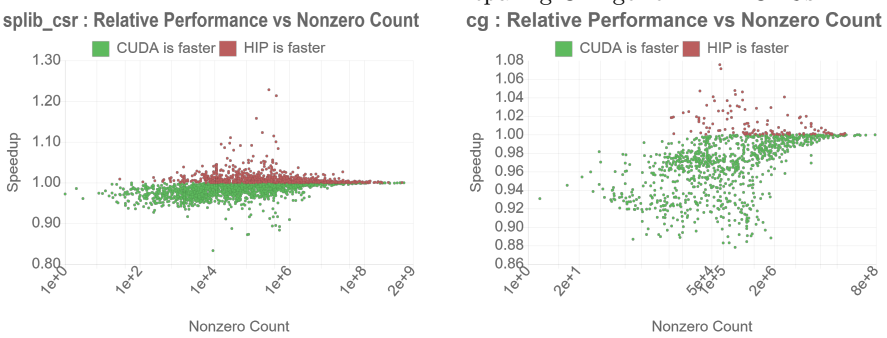


Fig. 5: Performance comparison for vendors' Csr SpMV (left) and 1,000 iterations of Ginkgo's CG solver (right).

we observe up to 15% performance degradation coming from the aforementioned differences in code generation. This is in accordance with Philip C. Roth [7] who compares the performance of CUDA and HIP for the scalable heterogeneous computing (SHOC) benchmark [3].

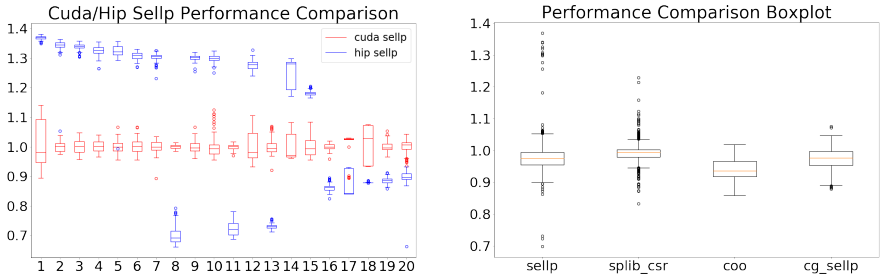


Fig. 6: Left: Performance variance for outliers in Sellp SpMV kernel analysis (Figure 4). All performance is normalized to the mean CUDA performance, CUDA performance in red, (relative) AMD performance in blue. Right: Performance statistics for all test cases and all kernels/algorithms.

As some of the performance differences in Figure 4 are significant, we investigate in Figure 6 (left) the mean and variance of the 20 most significant outliers in the Sellp SpMV analysis in Figure 4 (left). These statistics are collected from over 20 runs, each averaging the kernel characteristics over 100 invocations. Acknowledging the reproducibility of these outliers, we emphasize that they are still almost negligible when considering the complete test suite of more than 2,800 test matrices: The performance ratio statistics on the right-hand side of Figure 6 reveal that the performance means for all functionalities are just slightly below 1.0. Furthermore, 50% of the test cases show less than 3% performance difference, and 90% of the test cases show less than 10% performance difference. This reveals that HIP introduces only negligible overhead when comparing to CUDA-native code.

5 Conclusion

We elaborated how we extend the hardware scope of the Ginkgo linear algebra package to feature a HIP backend for AMD GPUs. We discussed the porting effort, and how the use of a shared code base reduces to minimize code duplication in a library providing native backends for NVIDIA and AMD GPUs. We also detailed the addition of functionality currently lacking in the HIP ecosystem and evaluated the performance price of compiling HIP code for NVIDIA architectures. We found that a significant portion of sparse linear algebra kernels allows for good platform portability. In future, we will create a Intel GPU backend and compare the porting process with the HIP backend integration.

Acknowledgment

This research was supported by the Exascale Computing Project (17-SC-20-SC) and the Helmholtz Impuls und Vernetzungsfond VH-NG-1241.

References

1. AMD: HIP. https://github.com/ROCm-Developer-Tools/HIP
2. Anzt, H., Dongarra, J., Flegar, G., Higham, N.J., Quintana-Ortí, E.S.: Adaptive precision in block-jacobi preconditioning for iterative sparse linear system solvers. Concurrency and Computation: Practice and Experience **31**(6), e4460 (2019)
3. Danalis, A., Marin, G., McCurdy, C., Meredith, J.S., Roth, P.C., Spafford, K., Tipparaju, V., Vetter, J.S.: The scalable heterogeneous computing (SHOC) benchmark suite. In: Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units. pp. 63–74. dl.acm.org (2010), https://dl.acm.org/doi/abs/10.1145/1735688.1735702
4. Google: https://github.com/google/googletest
5. Kuznetsov, E., Stegailov, V.: Porting CUDA-Based Molecular Dynamics Algorithms to AMD ROCm Platform Using HIP Framework: Performance Analysis. In: Supercomputing. pp. 121–130. Springer International Publishing (2019). https://doi.org/10.1007/978-3-030-36592-9_11
6. NVIDIA Corp.: Whitepaper: NVIDIA TESLA V100 GPU ARCHITECTURE (2017)
7. Roth, P.C.: Experiences with the Heterogeneouscompute Interface for Portability (HIP) on OLCF Summit (Oct 2019), https://www.olcf.ornl.gov/wp-content/uploads/2019/10/Roth-HIP-on-Summit-20191009.pdf
8. SuiteSparse: Matrix Collection. https://sparse.tamu.edu, Accessed 01/2020
9. Sun, Y., Mukherjee, S., Baruah, T., Dong, S., Gutierrez, J., Mohan, P., Kaeli, D.: Evaluating Performance Tradeoffs on the Radeon Open Compute Platform. In: 2018 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS). pp. 209–218 (Apr 2018). https://doi.org/10.1109/ISPASS.2018.00034
10. Zubair, M., Warner, J., Wagner, D.: Optimization of a Solver for Computational Materials and Structures Problems on NVIDIA Volta and AMD Instinct GPUs. In: 2019 IEEE/ACM 10th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems (ScalA). pp. 9–16 (Nov 2019). https://doi.org/10.1109/ScalA49573.2019.00007