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]

Karlsruhe Institute of Technology, Karlsruhe, Germany
[2] Innovative Computing Lab, University of Tennessee, Knoxville, TN, USA
yu-hsiang.tsai terry.cojean tobias.ribizel hartwig.anzt}@kit.edu

Abstract With AMD reinforcing their ambition in the scientific high performance computing ecosystem, we extend the hardware scope of the INKGO linear algebra package to feature a HIP backend for AMD GPUs. In this paper, we report and discuss the porting e ort 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

Introduction

Over the last decade, GPUs have been established as the main powerhouse in leadership supercomputers [1]. GPUs have proven valuable components to accelerate computations not only for machine learning workloads, but also for numerical linear algebra libraries powering computational science [2]. As of today, AMD and NVIDIA are considered the main GPU manufacturers. In the past, software e orts 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 [2], and the US Exascale Computing Project's mission to provide math functionality on the leadership systems, we extend the scope of the INKGO library to feature an AMD GPU backend.

In this paper, we report and discuss the e ort 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 e ort of porting a software stack to new architectures is, for example, described for molecular dynamics algorithm in [7], and for the solution of finite element problems in [12]. Concerning performance portability, the authors of [11] 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, INKGO 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 e ort in Sect. 3, we recall some background information about CUDA and HIP in Sect. 2. We present the results of the experiments of the same kernels being compiled by CUDA and HIP in Sect. 4. We conclude in Sect. 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 kernels for GPU architectures 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 "RadeonOpen-Compute" (ROCm). ROCm is the first open-source HPC platform for GPU computing shipping with several math libraries, like rocBLAS, rocSPARSE, roc-SOLVER, 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 di erent hardware architectures, therewith also reducing the maintenance e ort for libraries supporting several backends.

In Listing 1.2, HIP uses the same declaration specifier `__global__` like CUDA, but a di erent 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 e ect di ers from the CUDA ecosystem due to the architectural di erences 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 Di erence Between AMD and NVIDIA GPUs

The primary technical di erence 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 di erence 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 GPUs di er 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 Sect. 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 Sect. 3.3.

3 Porting CUDA Functionality to the HIP Ecosystem

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

3.1 Ginkgo Design

A high-level overview of INKGO's software architecture is visualized in Fig. 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 [6] framework. The "include" folder contains the public interface. Extending INKGO'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 Sect. 3.2.

To reduce the e ort of porting INKGO to AMD architectures, we use the same base components of INKGO 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;

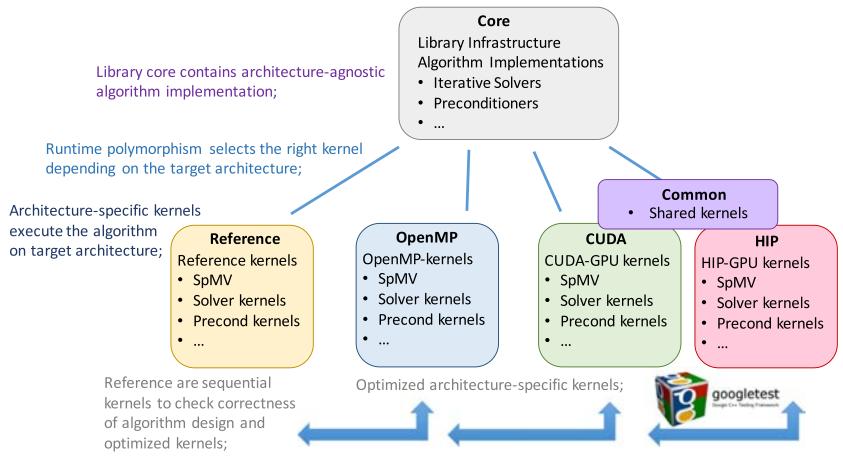

Core
Library Infrastructure
Algorithm Implementations
• Iterative Solvers
• Preconditioners
• ...
Library core contains architecture-agnostic algorithm implementation;
Runtime polymorphism selects the right kernel depending on the target architecture;
Architecture-specific kernels execute the algorithm on target architecture;
Common
• Shared kernels
Reference
Reference kernels
• SpMV
• Solver kernels
• Precond kernels
• ...
OpenMP
OpenMP-kernels
• SpMV
• Solver kernels
• Precond kernels
• ...
CUDA
CUDA-GPU kernels
• SpMV
• Solver kernels
• Precond kernels
• ...
HIP
HIP-GPU kernels
• SpMV
• Solver kernels
• Precond kernels
• ...
Reference are sequential kernels to check correctness of algorithm design and optimized kernels;
Optimized architecture-specific kernels;
googletest

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

– `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[1] 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 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 Sect. 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

[1] 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.

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 INKGO for optimizing sparse linear algebra kernels. HIP [3] 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 INKGO.

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

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

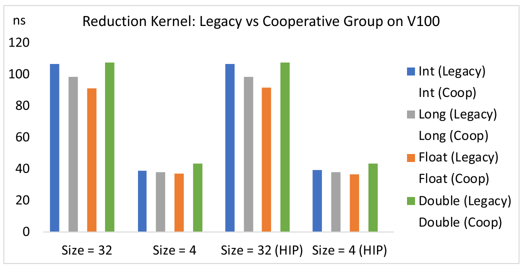
where `tid` is local thread id in a thread block such that `Rank` gives the local id of this subwarp, and 0 is a bitmask of 32/64 bits, same bits as `lane_mask_type`, filled with 1 bits according to CUDA/AMD architectures, respectively. Using this definition, we can realize the cooperative group interface, 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 (*pop count*), 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 = tiled_partition<Size>(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



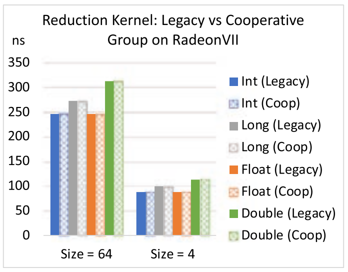


Fig 2 INKGO’s cooperative groups vs. legacy functions for di erent data types on V100 (left) and RadeonVII (right). (Color figure online)

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 INKGO’s cross-platform cooperative group interface (green). In Fig. 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”