A Performance Portability Study Using Tensor Contraction Benchmarks

(Invited Paper)

M. Emin Ozturk, Omid Asudeh*, Gerald Sabin†, P. Sadayappan* Aravind Sukumaran-Rajam‡*,*
**Kahlert School of Computing, University of Utah, Salt Lake City, Utah*
†RNET Technologies, Dayton, Ohio
‡Meta Platforms , Menlo Park, California

***Abstract*—Driven by the end of Moore's law, heterogeneous architectures, particularly GPUs, are experiencing a surge in demand and utilization. While these platforms hold the potential for achieving high performance, their programming remains challenging and requires extensive hardware knowledge. This complexity is further exacerbated by the different proprietary languages utilized by various vendors. In this paper, we conduct a performance-portability study on two portable languages, SYCL and Kokkos. Specifically, we focus on the case study of tensor contractions and employ COGENT, a DSL compiler for tensor contractions, to generate CUDA code for the 48 different tensor contractions in the TCCG benchmark suite. We extend COGENT to produce Kokkos code, and use Hipify and SycloMatic, which are tools that convert CUDA code to HIP and SYCL. Our analysis involves a comparison of the performance of each framework on both Nvidia and AMD GPUs. Our experiments show that identically tiled tensor contraction kernels in Kokkos and SYCL can exhibit significant performance differences compared to the corresponding CUDA/HIP program, respectively on Nvidia/AMD GPUs. The main reason for the performance differences arise from differences in register usage and the management of register spills to thread-private stack memory, affecting overall degree of thread-level concurrency and the volume of data movement to/from GPU DRAM.**

***Keywords*-Performance portability; GPUs; Tensor contractions; Kokkos; SYCL**

I. Introduction

GPUs have become the primary hardware platform for high-performance machine learning and also for many applications in scientific computing. However, a significant software challenge has emerged with multiple proprietary GPU programming languages, e.g., CUDA for Nvidia GPUs, HIP for AMD GPUs, and a different programming model like OpenMP for CPUs. Maintaining multiple versions of an application code for different hardware platforms is highly undesirable. Portable programming models like Kokkos [11]and SYCL [5] seek to address this software challenge. A number of prior efforts have performed evaluations on the effectiveness of such portable programming models. In this paper, we undertake a performance portability study of Kokkos and SYCL using 48 tensor contraction benchmarks from the TCCG benchmark suite [17].

To achieve performance portability, the programming model must provide an abstract way of expressing concurrency and parallelism, which can be mapped to the parallel features of the underlying hardware by the implementation. The range of implementation options for both expressing parallelism and mapping to the hardware can impact the degree to which performance portability is achieved.

SYCL is a portable parallel programming model that offers a range of abstraction levels for developing parallel applications, including data-parallel kernels, hierarchical parallelism, SIMT-style NDRange kernels, and high-level task-based concurrency. Several implementations of the SYCL specification exist, including hipSYCL and Intel oneAPI DPC++.

Kokkos is a portable programming model that provides a range of abstraction levels for developing parallel applications, including data-parallel kernels, hierarchical parallelism, and task parallelism. Kokkos is designed to support the development of performance portable applications that can run efficiently on different architectures, including CPUs, GPUs, and accelerators. It accomplishes this by providing a single-source programming model that can be compiled and executed on different architectures without the need for significant modifications. Multiple backends are available for the Kokkos programming model, including those for CUDA, HIP, OpenMP, and SYCL.

This paper aims to evaluate the effectiveness of Kokkos and SYCL in enabling performance portability for high-performance tensor computations and to identify potential opportunities for improvement. While a number of prior efforts have performed studies to evaluate performance-portable frameworks [9], [10], [12], [13], they have generally focused on one or a very small number of kernels. In contrast, this study uses a large number of GPU kernels for tensor contractions, featuring tensors and iteration spaces of different dimensionality. Further, most prior performance portability

979-8-3503-1199-0/23/$31.00 ©2023 IEEE
DOI 10.1109/IPDPSW59300.2023.00102

studies [9], [10], [12], [13] appear to have focused mainly on bandwidth-limited GPU kernels/applications where observed performance differences across portable frameworks like Kokkos/SYCL and native GPU programming models like CUDA/HIP were not very significant.

Our experiments with the tensor contraction benchmarks showed much greater variability across frameworks and target platforms than previous performance portability studies. Some high-level observations from our study are listed below.

- Unlike with bandwidth-limited GPU kernels studied in prior studies in the literature, optimized GPU kernels for tensor contraction use fairly large register tiles for data locality optimization. The management of thread-local data (scalar intermediates and the variables associated with register-tiles) in the generated code by different frameworks results in significant performance differences across frameworks and hardware platforms.
- Register usage with Kokkos tends to be higher than both CUDA and HIP, resulting in lower thread occupancy and lower performance. On Nvidia GPUs, the use of launch bounds directives is very helpful in increasing Kokkos performance, raising performance to even higher levels than CUDA for many of the benchmarks. On AMD GPUs, launch bounds did not improve performance of Kokkos kernels.
- Achieved thread occupancy on Nvidia GPUs was quite consistently higher for SYCL than Kokkos but SYCL performance was often worse than kokkos performance due to significantly higher global-memory data writes due to register spilling into the thread's stack memory.
- SYCL performance relative to the native programming model (CUDA/HIP) differed quite significantly on the Nvidia GPU compared to the AMD GPU, being considerably worse than CUDA on the former and overall better than HIP on the latter.

II. Programming Models and Performance Portability

Performance portability is an important concern in high-performance computing, where applications need to be able to run efficiently on different hardware architectures. To address this challenge, various performance portability tools have been developed in recent years. In the following we go through some of the state of the art performance portability frameworks/interfaces.

1) Kokkos: Kokkos[11] is a widely used programming framework created at the U.S. Department of Energy's Sandia National Laboratories. It enables the development of performance-portable applications that can run efficiently on various computer architectures such as CPUs, GPUs, and other accelerators. Kokkos uses a parallel programming model based on C++ templates and provides a high-level programming interface that is independent of hardware. It is widely used in scientific computing, engineering, and high-performance computing applications, and is incorporated into more than 100 software components and applications, achieving performance portability on at least 5 of the top 10 supercomputers. Users write their applications using Kokkos abstractions, and performance portability is automatically enabled across different hardware platforms, including multi-core CPUs and GPUs.

The Kokkos programming model provides a range of features for developing high-performance scientific applications across a variety of hardware platforms. One of the key features of recent Kokkos versions is the capability of describing multi-dimensional iteration spaces using the MDRangePolicy, which is an execution policy that defines the iteration space and thread mapping for parallel constructs like parallel_for and parallel_reduce [18]. The MDRange (Multi-Dimensional Range) policy enables users to specify the iteration space by defining the lower and upper bounds for each dimension, as well as the degree of parallelism or the number of threads used to execute the parallel construct. Kokkos maps threads to the iteration space, dividing the space into smaller tiles and assigning one thread to each element in the tile. The number of threads used for parallel execution can be specified by providing a third argument to the policy constructor, called the tileSize [18]. This approach helps improve performance by leveraging the underlying hardware's parallelism and memory hierarchy. The MDRange policy plays a vital role in managing parallel execution in Kokkos, allowing users to exploit the full potential of modern high-performance computing architectures.This flexibility allows developers to write efficient and portable code that can be optimized for specific hardware architectures. Furthermore, Kokkos provides a range of tools and libraries to help developers optimize their code for performance, such as the Kokkos Profiling Interface (KPI) and the Kokkos Core library.

2) SYCL: SYCL[5] is a C++ abstraction layer for developing portable heterogeneous applications that can run on multiple hardware platforms, including CPUs, GPUs, FPGAs, and other accelerators. It is a standard

programming model developed by the Khronos Group, and it provides a unified programming interface for developers to write parallel code that can be executed on different hardware architectures.

SYCL is based on a single-source programming model, where a single C++ source code can be written for both host and device. SYCL programs are written using standard C++ and can be compiled using standard C++ compilers. SYCL provides an interface for expressing data parallelism and task parallelism using the concept of command groups and kernels, respectively. SYCL is supported by various vendors, including Intel, AMD, and Xilinx, and it is integrated with various programming frameworks, such as TensorFlow, PyTorch, and OpenCV. The features in SYCL for specifying multi-level parallelism have been heavily influenced by Kokkos and are conceptualy very similar to those discussed in this Section for Kokkos.

SYCLomatic[8] is a code conversion tool designed to help developers convert code written in different programming languages to SYCL. While the tool can assist with code migration, the final verification and editing process is still manual and must be performed by the developer. To make the migration process as simple as possible, developers can use the ”c2s” command to migrate existing CUDA codebases to SYCL. Once code is migrated to SYCL, it can be compiled and executed using any compiler that implements the SYCL specification.

3) HIP: HIP (Heterogeneous-Compute Interface for Portability)[2] is a C++-based language developed by AMD that provides an interface for writing code that can run on different heterogeneous compute systems, including both CPUs and GPUs [2]. The language is designed to be familiar to developers who have experience with CUDA. With HIP, developers can write code once and target different hardware architectures without significant changes to the code.

HIP provides a set of tools for compiling, debugging, and profiling HIP code, including the HIP runtime library, which allows for easy runtime deployment of HIP code, and the ROCm software stack, which is designed to provide optimized GPU acceleration for machine learning and scientific computing applications.

Hipify is an open-source tool provided by AMD's ROCm platform that allows developers to automatically convert CUDA code to HIP code, making it easier to port their applications to AMD GPUs. Hipify works by parsing the input CUDA code and transforming it into equivalent HIP code. The tool can handle most common CUDA constructs, such as kernels, device functions, and CUDA libraries. It also provides a set of command-line options that allow developers to customize the transformation process, including choosing the target architecture, enabling/disabling specific transformations, and specifying custom header files.

4) CUDA: The CUDA programming model, developed by NVIDIA, is a proprietary technology that has been widely used. CUDA, short for Compute Unified Device Architecture, is designed for massively parallel computing on NVIDIA GPUs [1].

With CUDA, developers can take advantage of the processing power of NVIDIA GPUs to accelerate computationally intensive tasks. This is achieved by utilizing the thousands of cores available on a GPU, which are optimized for data-parallel operations.

III. Code generation for tensor contractions

This section provides a brief background on tensor contractions and outlines the process of generating contraction code in various programming languages.

A. Tensor Contractions

Tensor contractions can be viewed as a generalization of matrix multiplication to higher dimensions. Consider the following Einstein notation for contracting two 4D tensors A and B to produce another 4D tensor C which represents $\sum_{e,f} A[a,e,b,f] * B[d,f,c,e]$:

$$C[a,b,c,d] = A[a,e,b,f] * B[d,f,c,e] \qquad \text{(III.1)}$$

The indices e and f are the contraction indices (internal indices), while a, b, c, and d are the external indices. In the Einstein index convention, summation (contraction) is implied for indices not appearing in the left-hand side tensor.

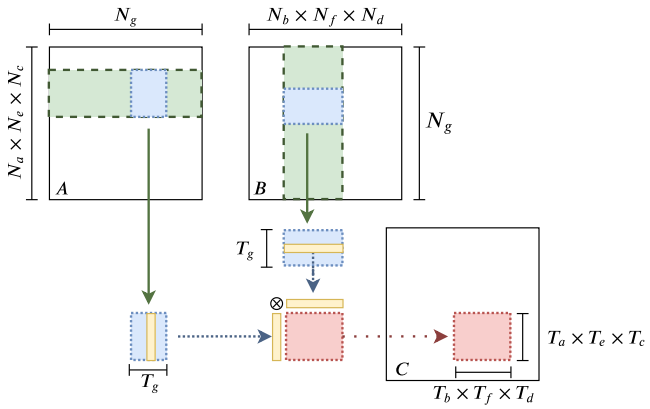

Figure 1: Illustration of Tiled-Execution of Tensor Contraction

We utilize COGENT [15], a CUDA code generator for tensor contractions based on the direct contraction-based approach. COGENT's contraction scheme involves streaming input tensors using shared memory and registers, while the output stays stationary in registers. Initially, a multidimensional slice of each input tensor is brought into shared memory. Next, the data in shared memory is sliced, and a column slice of input tensor A and a row slice of tensor B are loaded into registers from shared memory. These slices are subjected to outer product, and the partial results are stored in registers. All remaining input slices required to form the output are similarly streamed via shared memory and registers. Once all outer products are completed, the final output is written back to shared memory. Figure 1 provides a high-level overview of COGENT's contraction scheme.

In this work, we extend COGENT to generate HIP code and Kokkos code. We also tried using hipify to generate hip code from CUDA. We used SYCLomatic to convert CUDA code to SYCL.

B. Kokkos code generation

To generate Kokkos code, we utilized a similar approach to the original COGENT, but we adapted the code generator to utilize Kokkos constructs. Kokkos provides two primary mechanisms for specifying parallel execution patterns: TeamPolicy and MDRange. We selected TeamPolicy due to its ability to enable control over shared memory usage. Specifically, we used TeamPolicy to regulate the league_size (i.e., grid size) and team_size (analogous to grid and thread blocks in CUDA terminology). A team is similar to a CUDA block and is a collection of Kokkos threads that can synchronize and exchange data through a shared scratch pad (akin to shared memory in CUDA terminology) [3]. To represent the tensors (which were previously represented by raw pointers in COGENT's CUDA code), we utilized Kokkos::View. We launched a kernel using the 'Kokkos::parallel_for' function, and the Team_rank() function enabled us to obtain the thread identifier in Kokkos, which corresponds to threadIdx.y in CUDA.

C. SYCL code generation

To facilitate the generation of SYCL code, we employed the SYCLomatic tool, which automatically translates CUDA code to SYCL. However, we encountered difficulties with certain benchmarks that utilized constant memory, as SYCLomatic was unable to properly handle this code and generated runtime errors. To address this issue, we manually modified the translated code by converting the constant memory to regular global memory. This modification enabled the benchmarks to execute successfully in the SYCL programming model.

D. HIP code generation

Given the syntactical similarities between HIP and CUDA, it is relatively straightforward to migrate code between the two platforms. For instance, code elements such as cudaMemcpy, __global__, cudaStream_t can be replaced with their equivalent counterparts in HIP, namely hipMemcpy, __global__ __host__, and hipStream_t, respectively. Therefore, we adapted COGENT to directly generate HIP code, while also investigating the use of the hipify code converter to generate HIP code from existing CUDA code. Our experimental results show that the performance characteristics of the code generated via both approaches are very similar.

IV. Experimental Evaluation

In this section, we present an in-depth evaluation of the performance characteristics of Kokkos, SYCL, HIP, and CUDA on an Nvidia and AMD platform. We benchmarked various frameworks using 48 distinct tensor contractions from the TCCG suite [17], which represent contractions arising in practical applications. The corresponding contraction equations are listed in Table I:

- 1 to 8: eight tensor contractions representing tensor computations in machine learning domain.
- 9 to 11: three tensor contractions used to transform a set of two-electron integrals from an atomic orbital basis to a molecular orbital basis.
- 12 to 30: nineteen tensor contractions from the CCSD method.
- 31 to 48: eighteen tensor contractions from the CCSD(T) method.

Table I provides the dimensions of each tensor, the permutation of indices, and the total number of operations involved. The 'Tensor Contraction' column shows the Einstein representation of the contraction – the first group of letters corresponds to the indices of the output tensor, the second and third groups of letters correspond to the indices of the two tensors being contracted. For instance, in TCCG-2 (#2), two tensors $[d, c, a]$ and $[b, d]$ are contracted to generate the output matrix $[a, b, c]$, where d is the contraction index.

Performance evaluation was conducted on two systems: (1) Nvidia RTX 3060 (12 GB) GPU paired with an AMD Ryzen 7 3700x CPU, running Ubuntu 20.04 OS with CUDA version 11.6 and ROCm 5.3, and (2) AMD W6800 GPU (32 GB) paired with 10th Gen Intel(R) Core i9-10980X, running Ubuntu 20.04 OS with ROCm

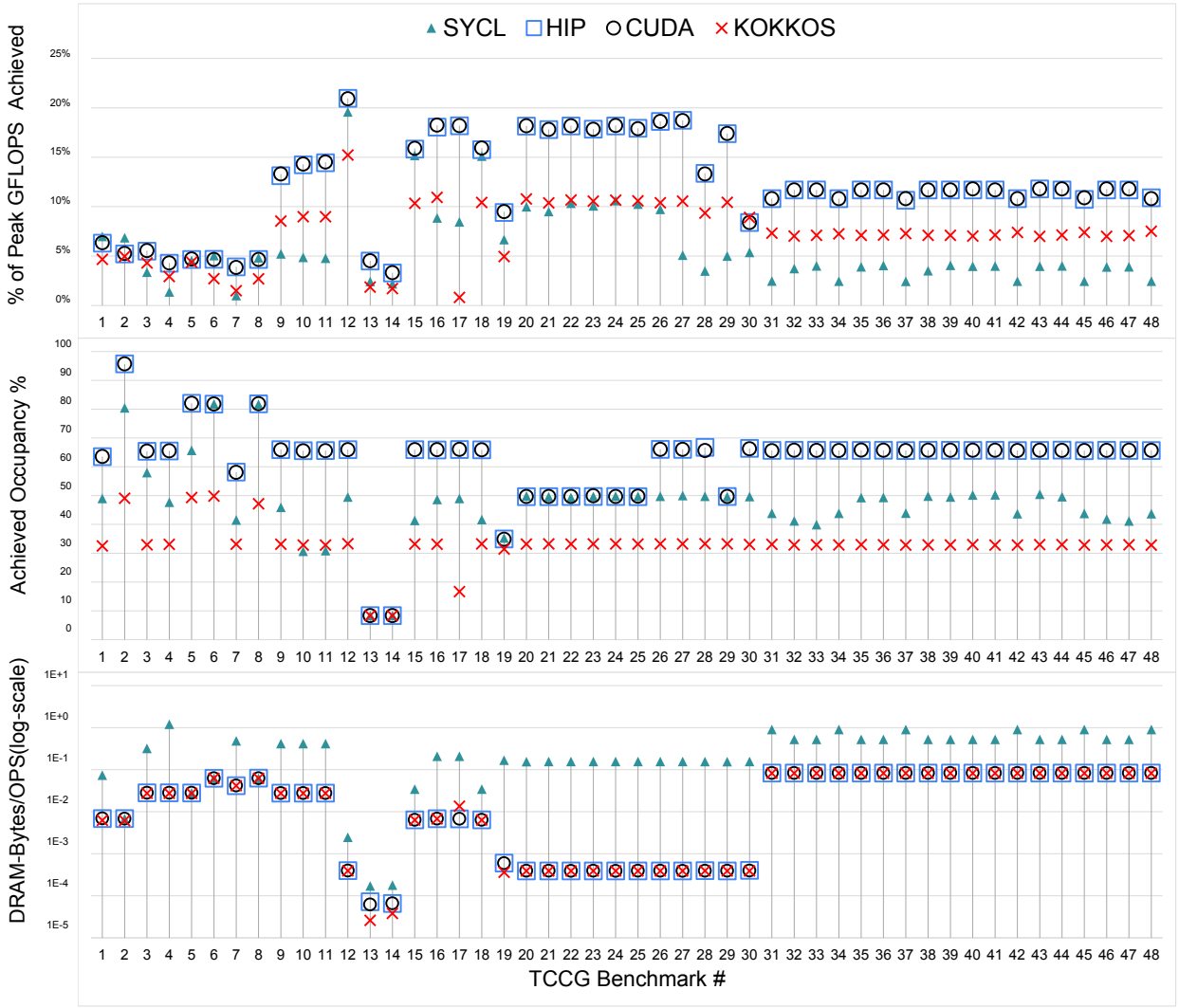

Figure 2: Percentage of peak GFLOPS achieved, Achieved Occupancy, and DRAM Bytes Write Per Operations on NVIDIA.

5.4 (for HIP and Kokkos) and ROCm 4.5.2 (SYCL). We had to use two versions of ROCm for W6800 as Kokkos was not compatible with ROCm 4.5.2 and SYCL was not compatible with ROCm 5.4.

We used DPC++ CUDA plugin from CodePlay [7] to run SYCL implementation of Intel oneAPI DPC++ (latest 2023.0.0 version) and DPC++ HIP plugin from CodePlay (2023.0.0 Beta) [6] on AMD machine. We used KOKKOS v3.7 [4]. Nsight was used to extract kernel time and collect performance metrics on the Nvidia platform, and ROC-profiler (rocprof 4.5.2) on the AMD platform. However, we could only obtain limited metrics on the AMD platform due to ROC-profiler's limited support on consumer cards W6800 with rocm-4.5.

Figure 2 presents a comparison of the achieved GFLOPS by different frameworks, relative to the peak GFLOPS, along with the achieved occupancy and the dram-bytes written per arithmetic operation (inverse of arithmetic intensity). CUDA and HIP achieves similar performance on the Nvidia platform. In general, KOKKOS outperforms SYCL, but performs less efficiently than CUDA/HIP. Our adapted COGENT code generator produces HIP code with similar performance characteristics to the HIP code generated by the hipfy tool for converting CUDA code. Therefore, we will fo-

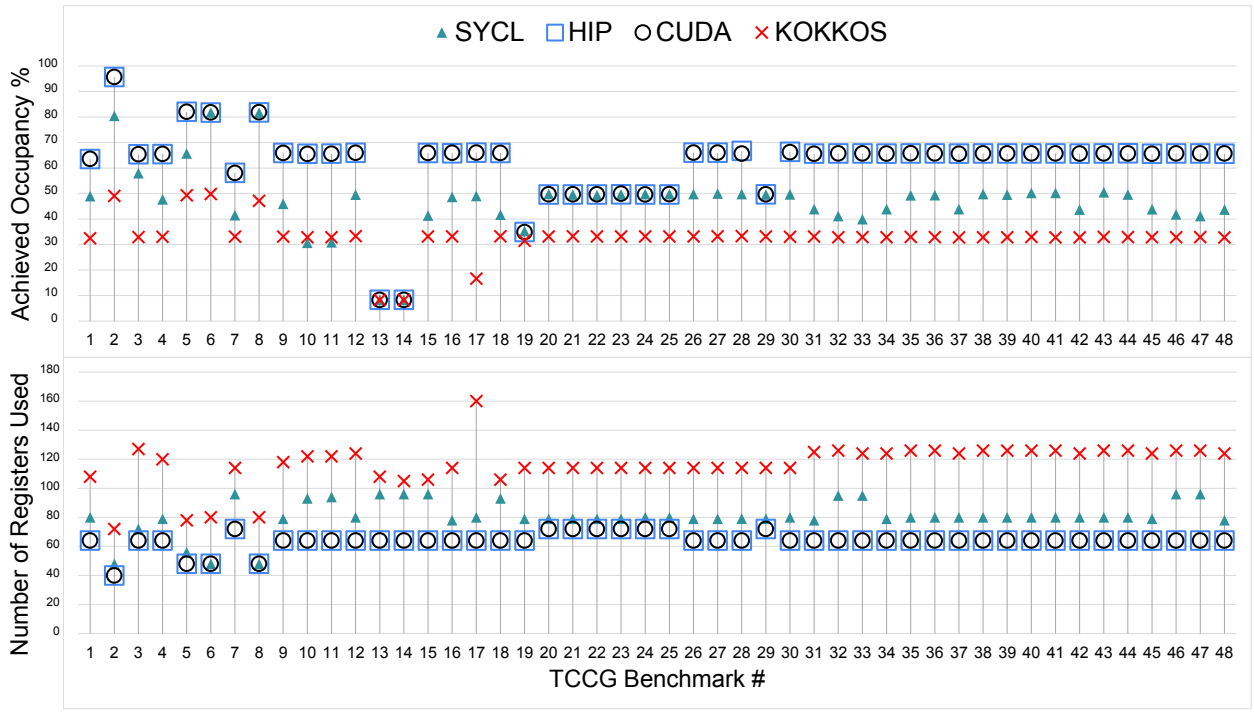

Figure 3: Achieved Occupancy versus Total Number of Registers Used on NVIDIA RTX3060

cus our studies on the COGENT-generated HIP code.To gain a better understanding of the performance differences, we conducted an analysis of the achieved occupancy and global memory (DRAM) data movement. Occupancy, defined as the ratio of the maximum number of active threads to the number of active threads, is an important metric that reflects the degree to which a GPU's computational resources are being utilized. A higher occupancy indicates that more threads are active, which can help in hiding memory latency and improving performance. In general, higher occupancy correlates with higher performance, except for SYCL, which has higher occupancy than KOKKOS but poorer performance due to the high data movement. This higher data movement is attributable to the use of local memory transactions in SYCL.

The occupancy of COGENT-generated code is mainly determined by register usage and shared memory usage. As shown in Figure 3, occupancy is well-correlated with register usage. We observed that KOKKOS typically uses a much higher number of registers than other frameworks. To better tune resource usage based on the launch configuration, we used launch_bounds. The "__launch_bounds__" primitive in the CUDA programming language enables us to inform the compiler of the launch configuration. The "__launch_bounds__(Max_Threads_Per_Block, Min_Blocks_Per_SM)" construct takes two parameters: maximum number of threads per thread block and minimum number of blocks per SM. KOKKOS exposes launch bound control using the Kokkos::LaunchBounds construct. We evaluated several {Max_Threads_Per_Block, Min_Blocks_Per_SM} configurations of the form {128, x} where x $\in$ {2, 4, 6, 8, 10, 12}, {256, y} where y $\in$ {2, 3, 4, 5, 6}, and {512, z} where z $\in$ {2, 3, 4}. This tuning process aims to identify the optimal values for these parameters that result in higher occupancy, which can lead to improved performance for each specific TCCG cases. Figure 4 shows the impact of launch bound tuning. The geomean speedup of KOKKOS tuned over the base version is $\sim 1.34x$ and for CUDA is $\sim 1.02x$. We observed that for certain case like 32 the performance gain was %120. As seen on Fig.4, we gain up to 2x speed up by tuning Kokkos implementation over all TCCG benchmark. For instance, TCCG benchmark case 15 and case 36, which represent a problem in CCSD and CCSD-T methods in chemistry, we get

Table I: TCCG Benchmark Tensor Contraction

#	Tensor Contraction	Extents	OPS
1	abc-bda-dc	312 312 24 312	1.46e+9
2	abc-dca-bd	312 24 296 312	1.38e+9
3	abcd-dbea-ec	72 72 24 72 72	1.29e+9
4	abcd-deca-be	72 24 72 72 72	1.29e+9
5	abcd-ebad-ce	72 72 24 72 72	1.29e+9
6	abcde-efbad-cf	48 32 24 32 48 32	3.62e+9
7	abcde-ecbfa-fd	48 32 32 24 48 48	5.44e+9
8	abcde-efcad-bf	48 24 32 32 48 32	3.62e+9
9	abcd-ea-ebcd	72 72 72 72 72	3.87e+9
10	abcd-eb-aecd	72 72 72 72 72	3.87e+9
11	abcd-ec-abed	72 72 72 72 72	3.87e+9
12	ab-ac-cb	5136 5120 5136	2.70e+11
13	ab-acd-dbc	312 296 296 312	1.71e+10
14	ab-cad-dcb	312 296 312 312	1.80e+10
15	abc-acd-db	312 296 296 312	1.71e+10
16	abc-ad-bdc	312 312 296 296	1.71e+10
17	abc-adc-bd	312 312 296 296	1.71e+10
18	abc-adc-db	312 296 296 312	1.71e+10
19	abc-adec-ebd	72 72 72 72 72	3.87e+9
20	abcd-aebf-dfce	72 72 72 72 72 72	2.79e+11
21	abcd-aebf-fdec	72 72 72 72 72 72	2.79e+11
22	abcd-aecf-bfde	72 72 72 72 72 72	2.79e+11
23	abcd-aecf-fbed	72 72 72 72 72 72	2.79e+11
24	abcd-aedf-bfce	72 72 72 72 72 72	2.79e+11
25	abcd-aedf-fbec	72 72 72 72 72 72	2.79e+11
26	abcd-aefb-fdce	72 72 72 72 72 72	2.79e+11
27	abcd-aefc-fbed	72 72 72 72 72 72	2.79e+11
28	abcd-eafb-fdec	72 72 72 72 72 72	2.79e+11
29	abcd-eafc-bfde	72 72 72 72 72 72	2.79e+11
30	abcd-eafd-fbec	72 72 72 72 72 72	2.79e+11
31	abcdef-dega-gfbc	24 16 16 24 16 16 24	1.81e+9
32	abcdef-degb-gfac	24 16 16 24 16 16 24	1.81e+9
33	abcdef-degc-gfab	24 16 16 24 16 16 24	1.81e+9
34	abcdef-dfga-gebc	24 16 16 24 16 16 24	1.81e+9
35	abcdef-dfgb-geac	24 16 16 24 16 16 24	1.81e+9
36	abcdef-dfgc-geab	24 16 16 24 16 16 24	1.81e+9
37	abcdef-efga-gdbc	24 16 16 16 24 16 24	1.81e+9
38	abcdef-efgb-gdac	24 16 16 16 24 16 24	1.81e+9
39	abcdef-efgc-gdab	24 16 16 16 24 16 24	1.81e+9
40	abcdef-gdab-efgc	24 16 16 16 24 16 24	1.81e+9
41	abcdef-gdac-efgb	24 16 16 16 24 16 24	1.81e+9
42	abcdef-gdbc-efga	24 16 16 16 24 16 24	1.81e+9
43	abcdef-geab-dfgc	24 16 16 24 16 16 24	1.81e+9
44	abcdef-geac-dfgb	24 16 16 24 16 16 24	1.81e+9
45	abcdef-gebc-dfga	24 16 16 24 16 16 24	1.81e+9
46	abcdef-gfab-degc	24 16 16 24 16 16 24	1.81e+9
47	abcdef-gfac-degb	24 16 16 24 16 16 24	1.81e+9
48	abcdef=gfbc-dega	24 16 16 24 16 16 24	1.81e+9

%120 and %50 improvement respectively. The original Kokkos implementation for Case-15 used 106 registers, resulting in an occupancy of 33% only. However, when the maximum number of threads per block was set to 128 and minimum number of active blocks per streaming multiprocessor (SM) was set to 8, the register usage decreased to 64 and the occupancy increased to 66%, significantly boosted the performance from 1316 to 2859 GFLOPS.

Despite having similar register usage compared to several other benchmarks, all frameworks demonstrated lower performance on case 13 and 14. The reason for this can be attributed to a smaller grid size (number of thread blocks) leading to low occupancy and underutilization of resources.

Figure 5 shows the percentage of peak GFLOPS achieved by SYCL, HIP, and Kokkos on AMD W6800. In many cases, SYCL outperforms Kokkos and even HIP on this platform. Unfortunately, we could not reliably collect performance metrics using ROC-prof on AMD W6800.

Figures 6 and 7 provide a comprehensive overview of SYCL and Kokkos performance on different platforms. It is worth noting that Figure 6 did not utilize launch bounds tuning due to the unavailability of an interface to tune it in SYCL, whereas Figure 7 presents results after launch bounds tuning. Overall, the performance of SYCL is better on the AMD W6800 platform, and in most cases, it performs comparably or better than HIP. However, SYCL's performance on the Nvidia RTX3060 platform is significantly lower than that of CUDA. Conversely, Kokkos demonstrates the opposite trend, where it performs relatively well on the Nvidia RTX3060 platform compared to the AMD W6800.

V. Related Work

Dufek et al. [10] conducted an extensive case study investigating the effectiveness of Kokkos and SYCL as performance-portable frameworks for the Milc-Dslash benchmark, focusing on NVIDIA, AMD, and Intel GPUs. The authors primarily developed a GPU parallel implementation for the Milc-Dslash application and assessed its performance portability using the widely recognized Kokkos and SYCL frameworks. By testing the implementations on GPUs from three major vendors—NVIDIA, AMD, and Intel— Dufek et al. [10] aimed to highlight the advantages of using performance-portable frameworks for achieving cross-platform compatibility and comparable performance levels. Their work provides valuable insights into the potential of Kokkos and SYCL for simplifying the development process and minimizing the need for platform-specific optimizations, while maintaining high-performance computing capabilities across various GPU architectures.

Several studies have been conducted to evaluate the performance of HPC-style SYCL applications in comparison to traditional programming models like OpenMP, CUDA, and OpenCL. McIntosh-Smith et. al. [9] examine the performance of three SYCL applications (BabelStream, Heat and CloverLeaf) from the high-performance computing (HPC) domain across various CPU and GPU architectures. Similarly, Milthorpe et. al. [14] evaluates 38 different kernels on SYCL with several SYCL implementation including DPC++,

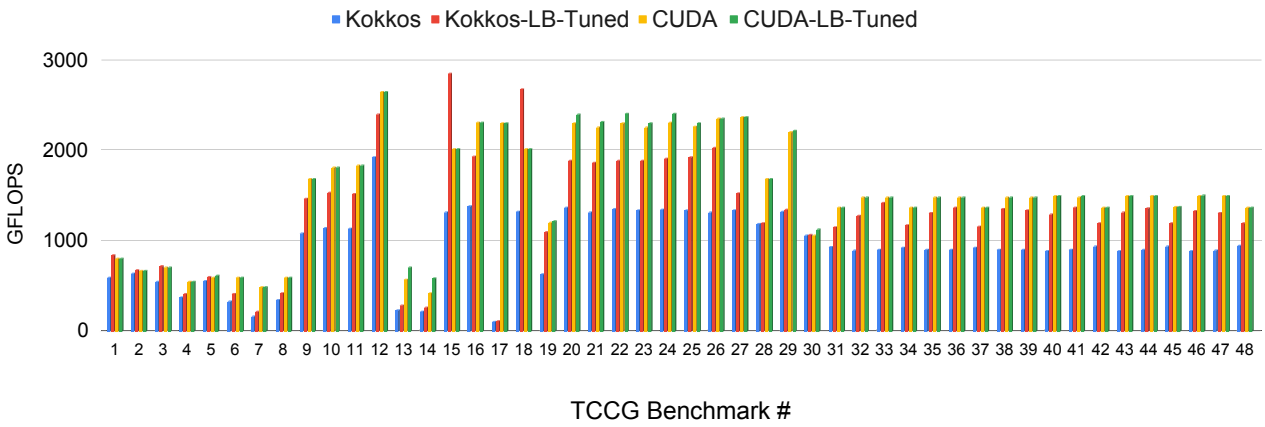

Figure 4: Performance impact of tuning launch bounds on Nvidia RTX 3060

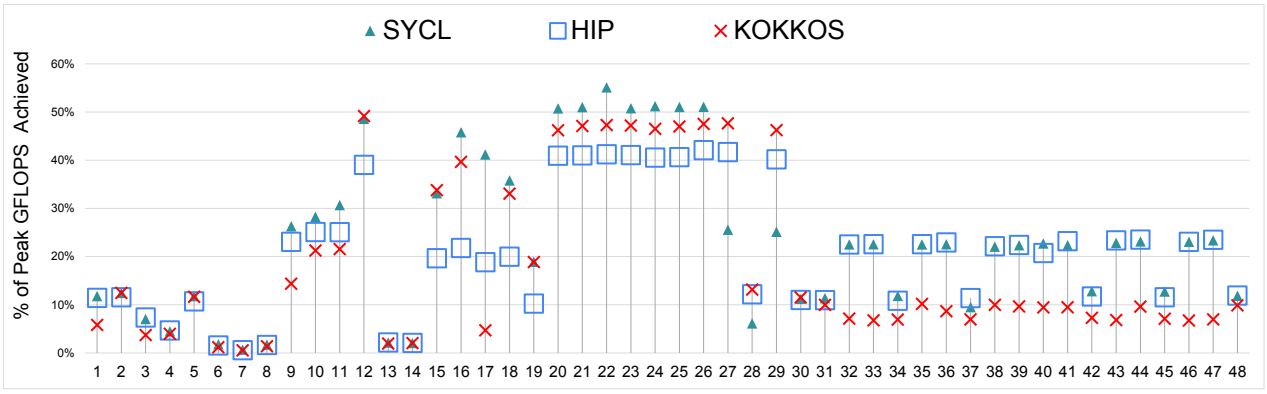

Figure 5: Percentage of peak GFLOPS achieved by SYCL, HIP, and Kokkos on AMD W6800

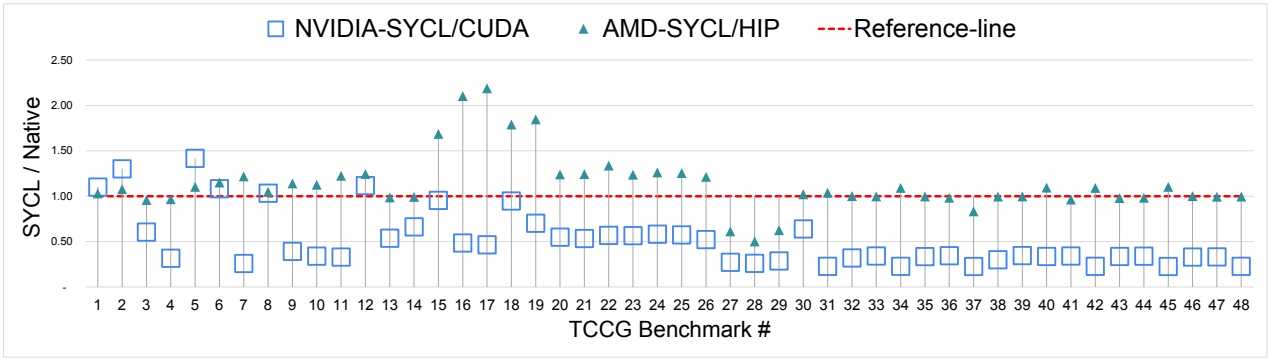

Figure 6: Correlation of NVIDIA-SYCL versus CUDA, and AMD-SYCL versus HIP

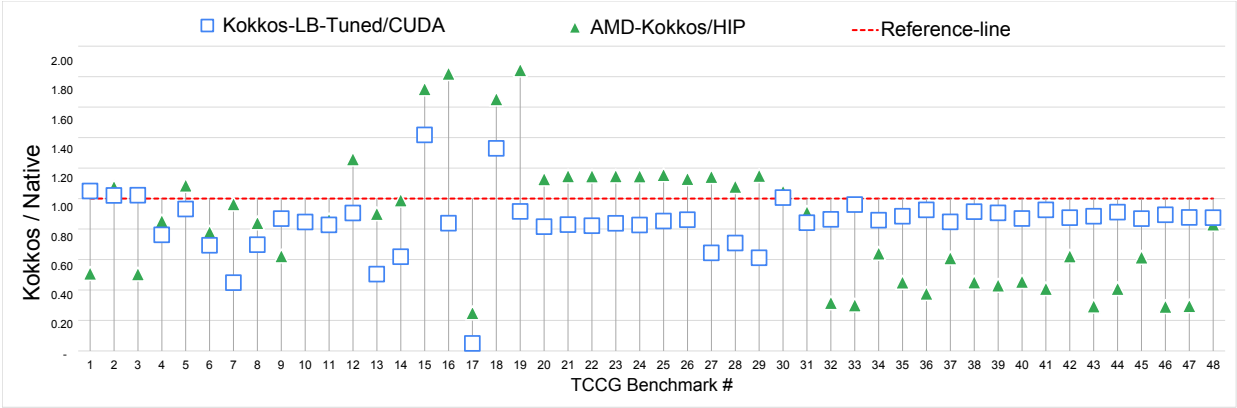

Figure 7: Correlation of Kokkos with MAX-Performing launch-bound versus CUDA, and AMD-Kokkos versus HIP

ComputeCPP, OpenSYCL(HipSYCL) and triSYCL on several architectures.

In recent works, several researchers have explored the migration of CUDA implementations to SYCL and oneAPI programming models for better performance portability on GPUs.

Jin et al. [12] discuss the conversion of a CUDA implementation of a high-order epistasis detection algorithm to SYCL and the evaluation of its performance on an NVIDIA V100 GPU. The paper covers various aspects of the migration process, including memory management, intrinsic and arithmetic functions, atomic functions, and kernel execution. The authors note that while the pointer-based memory management interfaces in SYCL make integration with existing CUDA programs easier, loop unrolling must be applied manually for comparable performance in SYCL. Additionally, the performance of the reduction operation within a group in the SYCL library depends on the problem size and work-group size. The study concludes that the highest performance is achieved when parallelized with four OpenMP threads and GPU processing concurrently.

In another work [13], Jin et al. explore the performance portability of SYCL kernels on a GPU by migrating representative kernels from CUDA to SYCL in bioinformatics applications, evaluating their performance on an NVIDIA GPU, and analyzing performance gaps through profiling and analyses. The findings provide suggestions for improving performance portability, such as optimizing address generation for shared local memory and evaluating the performance of a SYCL library interface. While the experimental results are based on a limited number of bioinformatics applications, they may be representative of kernels using optimized memory footprints and global memory accesses, shared local memory, loop(s) for iterating over attributes and elements, fast math library, or vendor-specific libraries for productivity and performance.

A study by Sakiotis et al.[16] discuss porting of optimized CUDA implementations to oneAPI, focusing on numerical integration use cases. Challenges included differences in registers, compiler optimizations, and mapping of CUDA library calls to oneAPI equivalents. After addressing the challenges, the performance of oneAPI implementations was found to be comparable to CUDA versions, at most 10% slower. The paper discusses the need for performant multi-platform execution, the development of portable programming models such as RAJA, Kokkos, and oneAPI, and the challenges faced during the porting process. The authors conclude that oneAPI is a viable platform for attaining performance on Nvidia GPUs.

VI. Conclusion

We compared the performance of tensor contractions implemented using various programming languages such as SYCL, Kokkos, HIP, and CUDA on Nvidia and AMD platforms. Our findings indicate that vendor languages (CUDA, HIP) outperformed SYCL and Kokkos. Our detailed analysis identified occupancy and data movement arising from the use of local memory as the primary factors contributing to this gap. Tuning launch bounds helped to reduce the performance gap between Kokkos and the native implementations. Our observations hint that a domain specific code generator which can directly emit portable code or a domain specific code generator paired with code convertors together with a tuning framework can strike the right balance between, programmability, productivity and performance.

ACKNOWLEDGMENT

This work was supported in part by the U.S. National Science Foundation through awards 1946752 and 2018016.

REFERENCES

[1] Cuda programming guide. https://docs.nvidia.com/cuda/cuda-c-programming-guide/. Accessed: 2023-03-15.

[2] Hip programming guide. https://docs.amd.com/bundle/HIP-Programming-Guide-v5.3. Accessed: 2023-02-27.

[3] Kokkos-wiki. https://kokkos.github.io/kokkos-core-wiki/. Accessed: 2023-03-15.

[4] Kokkosgithub. https://github.com/kokkos/kokkos/tree/release-candidate-3.7.02. Accessed: 2023-02-27.

[5] Sycl. https://www.khronos.org/sycl/. Accessed: 2023-02-27.

[6] Sycl-amd. https://developer.codeplay.com/products/oneapi/amd/home. Accessed: 2023-02-27.

[7] Sycl-nvidia. https://developer.codeplay.com/products/oneapi/nvidia/2023.0.0/guides/get-started-guide-nvidia. Accessed: 2023-02-27.

[8] Syclomatic. https://oneapi-src.github.io/SYCLomatic/index.html. Accessed: 2023-02-27.

[9] DEAKIN, T., AND MCINTOSH-SMITH, S. Evaluating the performance of hpc-style sycl applications. In *Proceedings of the International Workshop on OpenCL* (2020), pp. 1–11.

[10] DUFEK, A. S., GAYATRI, R., MEHTA, N., DOERFLER, D., COOK, B., GHADAR, Y., AND DETAR, C. Case study of using kokkos and sycl as performance-portable frameworks for milc-dslash benchmark on nvidia, amd and intel gpus. In *2021 International Workshop on Performance, Portability and Productivity in HPC (P3HPC)* (2021), IEEE, pp. 57–67.

[11] EDWARDS, H. C., TROTT, C. R., AND SUNDERLAND, D. Kokkos: Enabling manycore performance portability through polymorphic memory access patterns. *Journal of parallel and distributed computing 74*, 12 (2014), 3202–3216.

[12] JIN, Z., AND VETTER, J. S. Performance portability study of epistasis detection using sycl on nvidia gpu. In *Proceedings of the 13th ACM International Conference on Bioinformatics, Computational Biology and Health Informatics* (2022), pp. 1–8.

[13] JIN, Z., AND VETTER, J. S. Understanding performance portability of bioinformatics applications in sycl on an nvidia gpu. In *2022 IEEE International Conference on Bioinformatics and Biomedicine (BIBM)* (2022), IEEE, pp. 2190–2195.

[14] JOHNSTON, B., VETTER, J. S., AND MILTHORPE, J. Evaluating the performance and portability of contemporary sycl implementations. In *2020 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)* (2020), IEEE, pp. 45–56.

[15] KIM, J., SUKUMARAN-RAJAM, A., THUMMA, V., KRISHNAMOORTHY, S., PANYALA, A., POUCHET, L.-N., ROUNTEV, A., AND SADAYAPPAN, P. A code generator for high-performance tensor contractions on gpus. In *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)* (2019), IEEE, pp. 85–95.

[16] SAKIOTIS, I., ARUMUGAM, K., PATERNO, M., RANJAN, D., TERZIC, B., AND ZUBAIR, M. Porting numerical integration codes from cuda to oneapi: a case study. *arXiv preprint arXiv:2302.05730* (2023).

[17] SPRINGER, P., AND BIENTINESI, P. Tensor contraction benchmark v0.1. https://github.com/HPAC/tccg/.

[18] TROTT, C. R., LEBRUN-GRANDIE, D., ARNDT, D., CIESKO, J., DANG, V., ELLINGWOOD, N., GAYATRI, R., HARVEY, E., HOLLMAN, D. S., IBANEZ, D., ET AL. Kokkos 3: Programming model extensions for the exascale era. *IEEE Transactions on Parallel and Distributed Systems 33*, 4 (2021), 805–817.