Performance Study of GPU applications using SYCL and CUDA on Tesla V100 GPU

Goutham Kalikrishna Reddy Kuncham*
NextGen R&D
Tata Consultancy Services
India
email: kali.goutham.krishna@gmail.com

Rahul Vaidya*
NextGen R&D
Tata Consultancy Services
India
email: rahulv9196@gmail.com

Mahesh Barve
HPC Center Of Excellence
Tata Consultancy Services
India

* *Authors contributed equally*

2021 IEEE High Performance Extreme Computing Conference (HPEC) | 978-1-6654-2369-4/21/$31.00 ©2021 IEEE | DOI: 10.1109/HPEC49654.2021.9622813

***Abstract*—SYCL standard enables single-source programs to run on heterogeneous platforms consisting of CPUs, GPUs, FPGAs across different hardware vendors. SYCL combines modern C++ features along with OpenCL's portability. SYCL runtime is also capable of targeting the CUDA backend directly on NVIDIA GPUs. This approach can potentially improve the performance of SYCL on NVIDIA devices. Although NVIDIA GPUs can be targeted via OpenCL backend, their features and capabilities are limited, and the performance is inadequate.**

In this study, we compare the performance of the Nvidia V100 GPU using SYCL and CUDA. For performance evaluation, we selected three GPU applications: BabelStream, Mixbench, and Tiled Matrix-Multiplication. We conducted extensive tests to understand the performance in terms of DRAM bandwidth, kernel execution time, compilation time, and throughput. As per our study, the performance of SYCL and CUDA were found to be similar. However, in some cases, CUDA outperformed SYCL.

***Index Terms*—SYCL, Heterogeneous Computing, CUDA, GPU Benchmarks**

I. Introduction

Heterogeneous computing has become more popular in recent years. This is due to the fact that certain parts of an application are better suited to specific hardware. Scalar tasks, for example, could be more efficient on a CPU. GPU would be an excellent solution for vector tasks. For spatial workloads, FPGAs are preferable. In 2009, OpenCL [15] was the first framework that enabled support for heterogeneous platforms across multiple hardware vendors. Before OpenCL, developers had to learn vendor-specific programming models (CUDA) to run vendor's hardware. Since OpenCL is only a standard, hardware vendors are required to develop their flavor of OpenCL. One problem with OpenCL is that it is too low-level and inconvenient for the developers. SYCL [16] is a recently introduced, open-standard programming model that allows codes to run on heterogeneous systems across various hardware vendors, as shown in figure 1. SYCL programming model combines the portability of OpenCL along with the latest C++ constructs. SYCL runtime takes care of the low-level details such as data management and synchronization implicitly, thereby improving developers' productivity. As per SYCL specification [17], SYCL integrates OpenCL devices with modern C++. SYCL uses SMCP (single source multiple compiler-passes) approach to compile host and device codes, generating a fat executable file that can run on the host and targeted device(s).

No previous studies have compared the performance of LLVM-SYCL(Intel) and CUDA on NVIDIA GPUs. We investigated the performance of SYCL and CUDA on the NVIDIA V100 in this article. We use bar graphs/line plots to illustrate our findings for DRAM bandwidth, throughput, kernel execution time, compilation time, host to device, and device to host bandwidth.

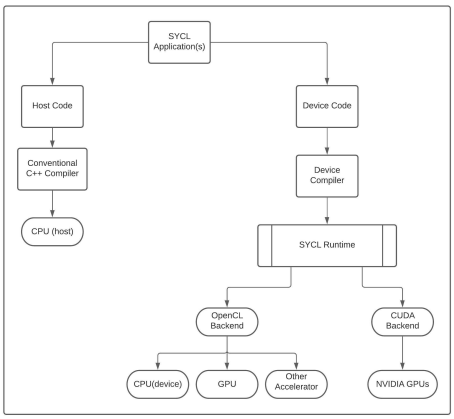

Fig. 1. SYCL Programming Model

Recently, LLVM based compiler for SYCL (open source effort led by Intel) [18] provided support for NVIDIA GPUs via CUDA backend as described in figure 1, since its performance was inadequate on OpenCL backend.

II. Literature Review

Benchmarking has been used to characterize a variety of systems and programming models. Rodinia benchmark suite was proposed by Che et al. [1] to evaluate new platforms such as GPUs. Furthermore, Che et al. [2] extended their work, characterized the Rodinia suite and compared it to coexisting CMP workloads. Burtscher et al. [3] proposed two metrics named control flow irregularity and memory-access irregularity and studied how irregular GPU kernels

978-1-6654-2369-4/21/$31.00 ©2021 IEEE
Authorized licensed use limited to: Universidad Industrial de Santander. Downloaded on October 11,2023 at 17:42:43 UTC from IEEE Xplore. Restrictions apply.

differ from regular kernels in terms of these metrics. Danalis et al. [4] designed the Scalable Heterogeneous Computing Benchmark Suite (SHOC) to compare the OpenCL and CUDA programming frameworks.

Reguly's research examines the performance of a single application over various devices, and types, including SYCL [5]. The study focuses on the portability of each kernel's implementation across numerous devices and programming models. Similarly, Joo et al. [6] investigate the performance of a single kernel implemented in SYCL and Kokkos. They are primarily concerned with GPU performance, with only a few CPU statistics included in the reference. In addition, Silva et al. compare the performance of two kernels written in SYCL, OpenCL, and OpenMP on CPU devices [7]. GPUs were not taken into account in their research. Hammond et al. [8] compared the SYCL and Kokkos programming models in semantics and parallelism. However, they did not explicitly offer performance data. Various parallel programming paradigms for GPUs and other accelerators have been compared in multiple papers [9], [10]. [13] looks at the performance of HPC Style applications across a range of devices in various programming models, including SYCL.

Previously, SYCL implementations targeted NVIDIA GPUs via the OpenCL backend. However, the problem with such an approach was that NVIDIA's OpenCL drivers offered limited support in terms of features to its GPUs. Consequently, its performance also was not up to the mark as observed in [14].

III. SYCL Applications

We use three different applications; two of these are High-Performance Computing benchmarks are available for both SYCL and CUDA. The applications selected contain all the characteristics of HPC codes. Mixbench and Babelstream have inbuilt models that gauge the performance parameters such as memory bandwidth, runtime, etc. We have also developed tiled matrix multiplication using shared/local memory in 2 different approaches.

A. BabelStream

Many real-world scientific applications are memory bandwidth bound and thus are commonly evaluated against the popular STREAM benchmark. The BabelStream benchmark is an implementation of this well-known STREAM benchmark in multiple parallel programming paradigms. BabelStream is mainly used to measure memory transfer rate to/from capacity memory. It incorporates a wide variety of = programming models which includes SYCL, CUDA [21]. BabelStream implements the STREAM benchmark's four major kernels. Along with these four kernels, it also implements the Dot product to evaluate reduction performance. It measures the time take for the kernels when applied to three large arrays.

1) Copy: c[i] = a[i]
2) Multiply: b[i] = αc[i]
3) Add: c[i] = a[i] + b[i]
4) Triad: a[i] = b[i] + αc[i]
5) Dot: sum = sum + a[i] $*$ b[i]

where a, b, c = large arrays; α = scalar constant

The main intention of this analysis is to estimate the realistic performance expectations of the portable programming models. It is assumed that if these simple kernels do not perform well, it is unlikely that an extensive scientific application with a similar codebase written will perform well. Since all the kernels in Babelstream are memory bandwidth bound, we used memory bandwidth and kernel run-time values as a metric to evaluate both implementation performances.

B. Mixbench

The motivation behind the Mixbench benchmark suite is to assess the execution limits of GPUs for mixed operational intensity kernels [22]. In general, GPU kernels can either be memory-bound or compute-bound. In this suite, the kernels are tuned based on a variety of operational intensity settings, which helps identify the critical performance factor of a program with regard to its operational intensity. Here the operational intensity is the compute to memory transfer ratio, typically measured in flops/byte.

This micro-benchmark suite allows us to examine the compute (GFLOPS) and memory (GB/sec) performance of various GPUs using a configurable operation intensity, i.e., this microbenchmark is run on a wide range of operation intensity values, allowing us to investigate the behavior of various GPUs on a range of different operation intensity values. Generally, the highest performance gained with multiply-add operations is usually provided by GPU suppliers. These multiplication and addition operations are combined into a single instruction. These instructions are usually designed to run in a single shader cycle. Vendors' theoretical peaks are based on a perfectly balanced stream of floating-point additions and multiplications. Performance suffers if the stream of executed instructions is not precisely balanced. Mixbench kernels involve both compute and memory traffic in a customizable proportion to generate a mixed-type workload. As a result, the behavior of various GPUs in mixed kinds of instruction streams may be studied. Multiply-additions kernels are evaluated on single-precision, double-precision, and integer operations:

C. Tiled Matrix Multiplication

Matrix multiplication is one of the most fundamental and essential computations performed across multiple domains. We use the tiled version of the matrix multiplication using shared memory (CUDA) and local memory (SYCL). Shared/local memory is expected to be much faster than global memory as per the memory hierarchy. Shared/local memory is a memory that is shared across all the threads/work items in a thread block (CUDA) and workgroup (SYCL). They are mainly used to minimize global memory accesses, thereby improving their performance significantly.

In this implementation, each workgroup/thread block computes a sub-matrix of the output matrix C. The sub-matrix of C is equal to the product of two rectangular matrices, A and B. These two rectangular matrices are divided into as many tiles as specified by the tile size. Then we can compute

978-1-6654-2369-4/21/$31.00 ©2021 IEEE
Authorized licensed use limited to: Universidad Industrial de Santander. Downloaded on October 11,2023 at 17:42:43 UTC from IEEE Xplore. Restrictions apply.

the submatrix of C by taking the sum of the product of these matrix tiles. We can store these tiles in shared/local memory, which is available across a thread block/workgroup, and perform the computation.

Terminology: A and B are Input matrices, C is the output matrix. N = size of a square matrix, tileSize = Size of the tile

We use two different approaches for tiled matrix multiplication, as discussed below. We fill the input matrices A and B with sin and cosine values, respectively. The output is stored in Matrix C. For performance evaluation, we have considered square matrices of varying input sizes and tile sizes.

In the CUDA version, we launch the matrix multiplication kernel with a grid size of (N/tileSize, N/tileSize) and thread block size (tileSize, tileSize). Correspondingly, in SYCL, we launch a kernel with a global size of (N, N) and a local size (tileSize, tileSize).

1) Approach 1: We have used a struct data type (Matrix) to store the input and output matrices in this approach. This approach is similar to the article given in [20].

```
typedef struct {
    int width;
    int height;
    int stride;
    float* elements;
    } Matrix;
```

The below structure stores Matrix attributes such as length, width, stride, and float pointer to store matrix elements. With the help of stride, the values stored at a given row and column can be accessed. Stride length is the distance between consecutive rows of a matrix. The height attribute to the number of rows of a matrix, and the width attribute stores the number of columns in a matrix.

Here we utilize the Matrix structure (struct) to create sub matrices inside the kernel. Each thread-block/workgroup computes a sub matrix of C of size tileSize x tileSize. The computation is blocked with the help of barrier function (SYCL) and sync threads function (CUDA) until all the tiles have been loaded with input matrix data. Then we perform computation on the loaded tile data stored in local/shared memory and write the results back to the global memory after computation.

2) Approach 2: In this approach, we first fill all the elements in every tile from input matrices A and B. Then we wait until all the threads have finished execution. Once the data elements get loaded in the tiles, we perform computation on it and write the results back to the global memory. Unlike the previous approach, we do not store the matrices in struct format.

IV. Results

Environment details:

- SYCL Compiler: LLVM-SYCL (an open-source development led by Intel)
- CUDA Compiler: Nvidia CUDA Toolkit 11.1
- GPU Device: Nvidia Tesla V100 PCIE (16GB)
- HostOS: Ubuntu 20.04

A. BabelStream

This study presents the results of five BabelStream kernels, i.e., Copy, Multiply, Add, Triad, and Dot for both SYCL and CUDA. All kernels contain a simple 1D cl::sycl::parallel_for where each work item works on its respective buffer without sharing. We aim to evaluate the performance of SYCL and CUDA programming models with the metrics Memory Bandwidth and Run time using the BabeStream benchmark suite.

1) Bandwidth Analysis:

Copy: From the figure 2, we can see that CUDA and SYCL achieve similar bandwidth values. CUDA was able to reach (809.9GB/s), which is approximately 89.9% of the theoretical bandwidth of V100. At the same time, SYCL was able to get (800.6GB/s), which is about 88.9% of the theoretical bandwidth with double precision. We also presented the values with a single-precision float which exhibits the identical result.

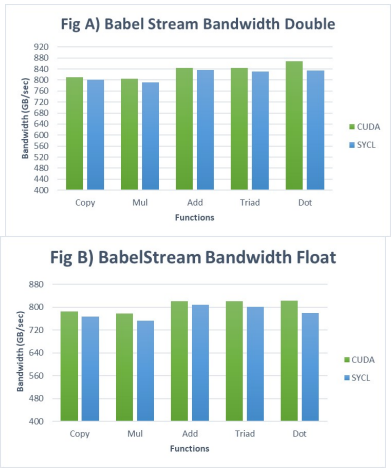

Fig. 2. BabelStream Bandwidth

Multiply: We can observe from the figure 2 that CUDA and SYCL have similar bandwidth values. CUDA achieved a bandwidth of 805.1 GB/s, which is around 89.4% of V100's theoretical capacity. SYCL was able to obtain 792.1GB/s at the same moment, which is roughly 88% of the potential bandwidth. These values were also presented using a single-precision float, which produced CUDA(86.3%) and SYCL(83.7%) bandwidths.

Add: With double precision, CUDA and SYCL versions were able to achieve 93.7% and 92.8% of the theoretical bandwidth, respectively.

Triad: From figure 3 we observe that CUDA implementation of Triad kernel achieved 93.7% of theoretical bandwidth, whereas SYCL implementation gained 92.4% of max achievable bandwidth.

DOT: From figure 2, we can observe that with the Dot product, both CUDA and SYCL implementations were able to get the highest bandwidth, i.e., 96.4% and 92.6%, respectively, with double precision.

2) Runtime Analysis: To compare the best runs of CUDA and SYCL, we ran the kernel

978-1-6654-2369-4/21/$31.00 ©2021 IEEE
Authorized licensed use limited to: Universidad Industrial de Santander. Downloaded on October 11,2023 at 17:42:43 UTC from IEEE Xplore. Restrictions apply.

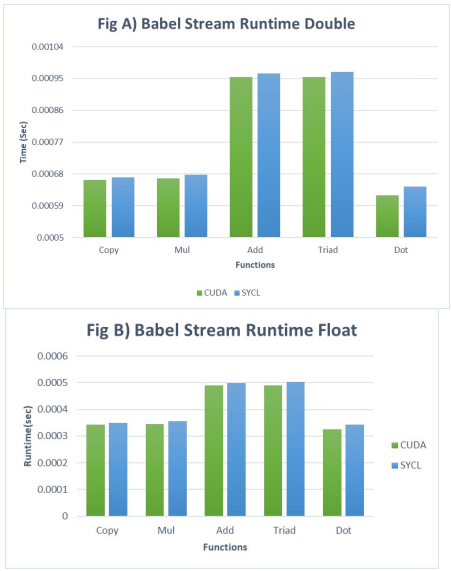

Fig. 3. BabelStream Runtime

100 times. When comparing CUDA with SYCL, from figure 3 we can find that CUDA performs somewhat better. CUDA is around 1.1 percent faster than SYCL. Despite the SYCL and CUDA implementations being equivalent, we observe CUDA performing slightly better than SYCL.

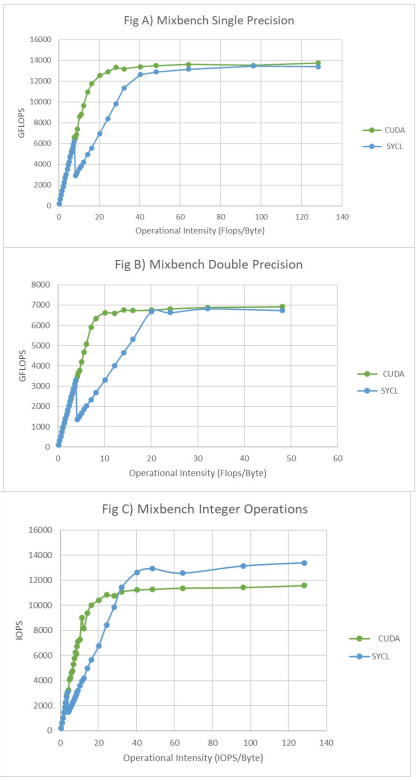

Fig. 4. Mixbench Roofline Analysis

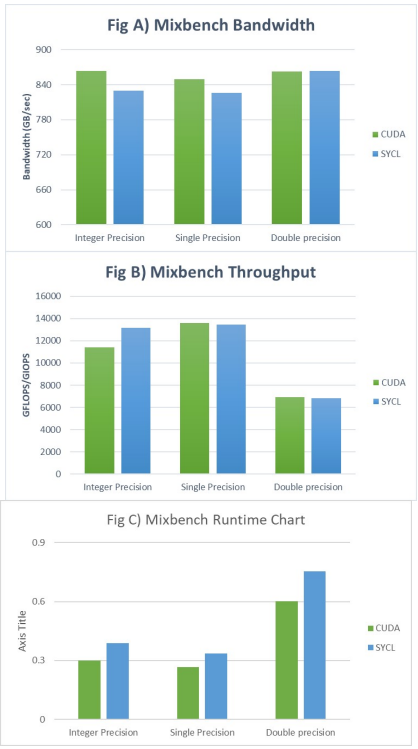

Fig. 5. Illustration of Bandwidth, Throughput and Runtime analysis of Mixbench Benchmark.

B. Mixbench

As we focus on GPU kernels, we use MixBench micro-benchmark suit in order to study the behavior of GPU on various operational intensities. In this study, we evaluate the performance of SYCL and CUDA implementations using the MixBench benchmark suite.

1) Roofline Analysis: Figure 4.A: (Single precision) shows single-precision analysis. The green line represents the performance of CUDA implementation of the suite, whereas the blue line represents the performance of SYCL implementation. In an ideal case, as per the theoretical peak performance specified by the GPU specifications, the GFLOPS should continuously increase with an increase in operational intensity till a certain threshold, and after that, the graphs flattens out(GFLOPS). From fig 4., A (single precision), the observed performance of CUDA follows a similar pattern to the theoretical one. Whereas, with respect to SYCL, although the initial graph tends to increase, there is a sudden drop in GLOPS at an operational intensity of 8.25 Flops/byte. However, after that, the GFLOPS starts growing and catches up with CUDA.

Figure 4.B (Double precision) represents the study of double-precision operations. Similar to single precision, double precision exhibits identical behavior with CUDA and SYCL implementation of Mixbench.

The analysis of Integer operations is depicted in Figure 4.C (Integer Ops). With integer operations, the performance of CUDA follows a similar trend to the theoretical one. However, SYCL performance presents some interesting behavior that is

978-1-6654-2369-4/21/$31.00 ©2021 IEEE
Authorized licensed use limited to: Universidad Industrial de Santander. Downloaded on October 11,2023 at 17:42:43 UTC from IEEE Xplore. Restrictions apply.

not seen with other precision operations. Although initially, the SYCL graphs follow a similar pattern to other precisions, after the operational intensity of 4.25 IOPS/byte, the GIOPS increase sharply and perform better when compared to CUDA implementation.

2) Bandwidth Analysis: We ran the benchmark with varying operational intensity.

Figure 5.A: [Bandwdith chart] illustrates the maximum attained bandwidth for Integer, Single precision, and double-precision operations. With CUDA implementation for Integer operations, the maximum bandwidth of 95.9% of theoretical bandwidth is reached at 3.25 IOPS/byte. At the same time, single-precision float operations achieved maximum bandwidth of 94.3% at 7.75 Flops/byte. Double-precision operations were able to attain 95.8% of theoretical bandwidth at 0.875 Flops/byte. With SYCL implementation, At 1.75 IOPS/byte, the maximum bandwidth of 92.2% of theoretical bandwidth is obtained for integer operations. On the other hand, single-precision float operations attained a maximum bandwidth of 91.7 percent at 0.75 Flops/byte. At 2.125 Flops/byte, double-precision operations were able to achieve 95.9% of theoretical bandwidth.

3) Throughput Analysis:

Figure 5.B:[Throughput chart], represents the maximum attained throughput(GFLOPS/GIOPS) for Integer, single-precision, and double-precision instructions. At 96.25 IOPS/byte, CUDA implementation for integer operations achieves a maximum throughput of 11413 GIOPS. Single-precision float operations attained a maximum throughput of 13606.87 GFLOPS at 64.25 Flops/byte at the same time. At 48.125 FLOPS/byte, double-precision operations were able to achieve 6917.2GFLOPS. The SYCL implementation for integer computations reaches a maximum performance of 13137.77GIOPS at 96.25 IOPS/byte. At 96.25 Flops/byte, single-precision float operations achieved a maximum throughput of 13445.73 GFLOPS. Double-precision operations were able to produce 6817.73GFLOPS at 32.125 FLOPS/byte. With SYCL implementation for Integer operations, there is an interesting observation that we can note, i.e., the throughput of SYCL implementation with integer operations is slightly greater than CUDA implementation.

4) Runtime Analysis:

We have studied the average of all varying operational intensity runs to evaluate the runtime of kernels for SYCL and CUDA implementations. Figure 5.C [Runtime Chart] depicts the average runtime values for Integer, single-precision, and double-precision operations. From the figure, we can observe that CUDA is slightly faster than SYCL.

C. Tiled Matrix Multiplication

We present the results of the tiled matrix multiplication (approach 1 and 2) on both SYCL and CUDA evaluated across input sizes ranging from 512 to 32768 (only considering the powers of 2 from 512 till 32768) and tile sizes ranging from 8 to 32 (only considering the powers of 2 from 8 till 32) using integer, float and double data types. We ran every combination of input matrix size and tile size for 50 times.

To measure compilation time, we used "time" command (Linux package). We measured Host to device bandwidth, device to host bandwidth, kernel execution times with the help cudaEvents in CUDA and sycl event in SYCL.

1) Compilation time: Although, compilation times are usually trivial, we analyzed the time taken to compile the program on SYCL and CUDA. The compilation time taken for a CUDA program is approximately 4 times faster than the SYCL program. We have compiled tiled matrix multiplication versions of CUDA and SYCL programs more than 100 times. On an average, CUDA programs took 1.043 seconds to compile, whereas SYCL programs took 4.094 seconds to compile

2) Host to device bandwidth: Host to device bandwidth is the amount of data sent from host to the device in one second. It is usually measured in number of Gigabytes (GB) of data travelled per second. For the SYCL version, the maximum host to device bandwidth attained is around 4.28 GB/s when the input matrix size is 8192 x 8192. Whereas on CUDA, the maximum bandwidth is around 7.5 GB/s when the input matrix size is 4096 x 4096.

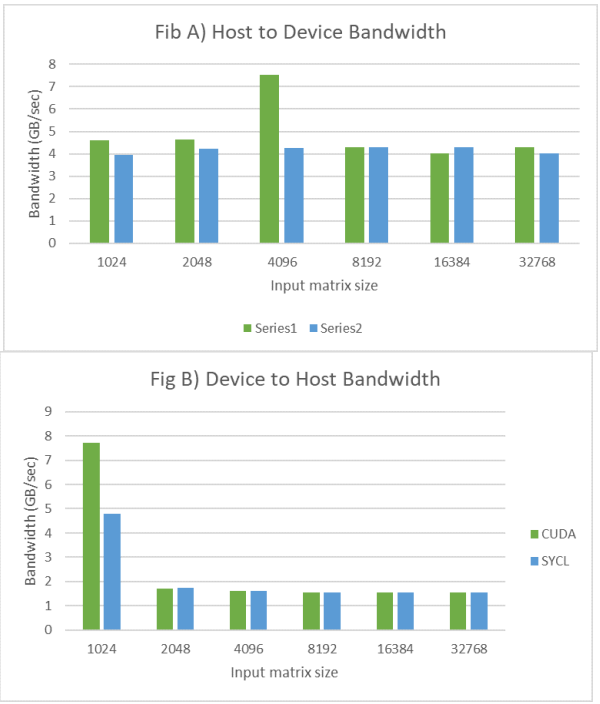

Fig. 6. Illustration of Host to Device Bandwidth and Device to Host Bandwidth

In figure 6.A, we plot the maximum host to device bandwidth attained for multiple matrix input sizes. The maximum transfer rates are pretty much the same across multiple input sizes except for 4096 x 4096 matrix where CUDA's transfer rate was significantly higher.

3) Device to host bandwidth: Device to host bandwidth is the amount of data sent from device to host in one sec. It is measured in GB/second. For the CUDA version, the

978-1-6654-2369-4/21/$31.00 ©2021 IEEE
Authorized licensed use limited to: Universidad Industrial de Santander. Downloaded on October 11,2023 at 17:42:43 UTC from IEEE Xplore. Restrictions apply.

maximum device to host bandwidth attained is around 7.7 GB/sec when the matrix size is 1024 x 1024. Whereas on SYCL, the maximum bandwidth attained is around 4.8 GB/sec with a matrix size of 1024 x 1024.

In figure 6.B, we plot the maximum device to host bandwidth attained for multiple matrix input sizes. As seen in the plot, the maximum transfer rates are almost similar for both CUDA and SYCL expect for 1024 x 1024 matrix where CUDA's transfer rate was considerably better.

4) Kernel execution time: We have evaluated the kernel execution times of SGEMM on 32768 x 32768 input matrices with 8, 16, 32 tiles sizes using two different approaches. For computing DGEMM,we have considered input matrices of size 16384 x 16384 because of memory constraints. Here are the results:

Approach 1:

From figure 7.A, we observe that the performance of approach 1 to be significantly faster on CUDA when compared to SYCL. For instance, to compute SGEMM (Single precision matrix-matrix multiplication) of 32768 x 32768 square matrix with a tile size of 16, the CUDA implementation only took 25.02 seconds, whereas on the other hand, SYCL took 120.49 seconds. The performance on SYCL further degrades with a tile size of 32 where SYCL implementation took 367.2 seconds and on the other hand CUDA implementation just took 18.4 secs. GEMM with integer data type performs exactly like SGEMM, in terms of execution time.

From figure 7.B, we observe that CUDA leads the charts with lower execution times even on 16384 x 16384 DGEMM. For a tile size of 32, CUDA took 4.2 seconds to execute where as SYCL's execution time was close to 46 seconds. SYCL's poor performance with this algorithm compelled us to come up with a different approach to solve this problem.

Here are the performance results with our second approach:

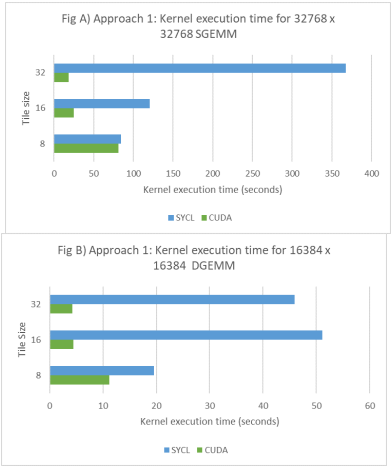

Fig. 7. Kernel execution time of SGEMM and DGEMM for approach 1

Approach 2:

We observed a significant performance gain on SYCL as seen in figure 8.A. In fact, SYCL has outperformed CUDA when computing GEMM/SGEMM on 32768 x 32768 input matrices. SYCL took 44.48 seconds with a tile size of 32. Whereas, CUDA took close to 154 seconds.

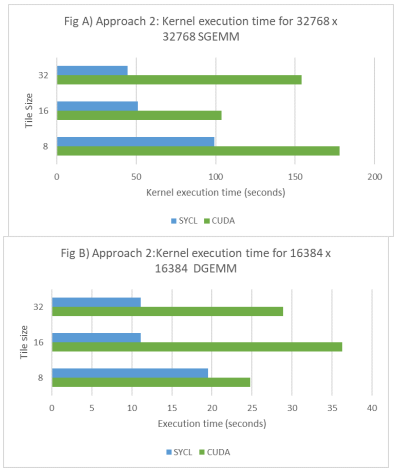

Fig. 8. Kernel execution time of SGEMM and DGEMM for approach 2

16384 x 16384 DGEMM performance was also found to be better on SYCL when compared to CUDA as per the figure 8.B

5) Observations: Approach 1 performed better on CUDA, whereas approach 2 was preferred by SYCL. Our hypothesis is that, since approach 1 creates additional sub-matrices within the kernel, the resulting overhead is significantly high in case of SYCL. Also, decreasing the tile size led to slight improvement in SYCL's performance when using approach 1, which indeed proves that performance drop can be attributed to additional memory creation. CUDA on the other hand seems to efficiently manage additional memory creation.

V. Conclusion and Future Work

As observed, SYCL's performance has matched CUDA in various aspects with respect to the above benchmarks, with DGEMM being an exception, where CUDA (approach 1) was approximately 2 times better SYCL (approach 2). When using additional memories within the kernel (like we demonstrated in SGEMM approach 1), SYCL's performance reduced drastically. For such scenarios, CUDA seems to perform better. When it comes to portability, SYCL applications can be run across multiple hardware platforms, whereas, CUDA is confined to NVIDIA GPUs. One limitation of SYCL is that, currently, SYCL runtime can only target a single backend. It is not possible to mix multiple backends. For instance, if we wish to target both OpenCL as well as CUDA backend for 2 different hardware platforms simultaneously in a single source code, that would not be possible.

For future work, we intend to compare and analyze the performance of HIP SYCL [19], LLVM SYCL and CUDA. HIP SYCL also targets NVIDIA GPUs via CUDA backend, hence it would be interesting to observe its performance when compared to the other two.

978-1-6654-2369-4/21/$31.00 ©2021 IEEE
Authorized licensed use limited to: Universidad Industrial de Santander. Downloaded on October 11,2023 at 17:42:43 UTC from IEEE Xplore. Restrictions apply.

REFERENCES

[1] Che, S., Boyer, M., Meng, J., Tarjan, D., Sheaffer, J., Lee, S.H., Skadron, K.: Rodinia: A Benchmark Suite for Heterogeneous Computing. In: Proceedings of the IEEE International Symposium on Workload Characterization, IISWC (2009)

[2] Che, S., Sheaffer, J.W., Boyer, M., Szafaryn, L.G., Wang, L., Skadron, K.: A Characterization of the Rodinia Benchmark Suite with Comparison to Contemporary CMP Workloads. In: Proceedings of the IEEE International Symposium on Workload Characterization, IISWC (2010)

[3] Burtscher, M., Nasre, R., Pingali, K.: A Quantitative Study of Irregular Programs on GPUs. In: Proc. IEEE, IISWC (2012)

[4] Danalis, A., Marin, G., McCurdy, C., Meredith, J.S., Roth, P.C., Spafford, K., Tipparaju, V., Vetter, J.S.: The Scalable Heterogeneous Computing (SHO

[5] Istvan Z. Reguly. 2019. Performance Portability of Multi-Material Kernels. In 2019 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC). IEEE, 26–35. https://doi.org/10.1109/P3HPC49587.2019.00008

[6] Balint Joo, Thorsten Kurth, M A Clark, Jeongnim Kim, Christian Robert Trott, Dan Ibanez, Daniel Sunderland, and Jack Deslippe. 2019. Performance Portability of a Wilson Dslash Stencil Operator Mini-App Using Kokkos and SYCL. In 2019 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC). IEEE, 14–25. https://doi.org/10.1109/P3HPC49587.2019.00007

[7] Hercules Cardoso Da Silva, Flavia Pisani, and Edson Borin. 2017. A comparative study of SYCL, OpenCL, and OpenMP. Proceedings - 28th IEEE International Symposium on Computer Architecture and High Performance Computing Workshops, SBAC-PADW 2016 (2017), 61–66. https://doi.org/10.1109/SBAC-PADW.2016.19

[8] Jeff R. Hammond, Michael Kinsner, and James Brodman. 2019. A comparative analysis of Kokkos and SYCL as heterogeneous, parallel programming models for C++ applications. ACM International Conference Proceeding Series (2019). https://doi.org/10.1145/3318170.3318193

[9] Memeti S et al. 2017. In Proceedings of the 2017 Workshop on Adaptive Resource Management and Scheduling for Cloud Computing 1-6

[10] Hoshino T, Maruyama N, Matsuoka S, Takaki R. 2013. In 2013 13th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing. 136-143

[11] Guo X, Wu J, Wu Z, Huang B. 2016. IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing. 9(4):1653-62

[12] Mikhail Khalilov and Alexey Timoveev 2021 J. Phys.: Conf. Ser. 1740 012056

[13] T. Deakin and S. McIntosh-Smith, "Evaluating the performance of HPC-style SYCL applications", Proceedings of the International Workshop on OpenCL, pp. 1-11, apr 2020.

[14] S. Lal, A. Alpay, P. Salzmann, B. Cosenza, A. Hirsch, N. Stawinoga, et al., "SYCL-Bench: A Versatile Cross-Platform Benchmark Suite for Heterogeneous Computing", International European Conference on Parallel and Distributed Computing (Euro-Par), 2020.

[15] https://www.khronos.org/opencl/

[16] https://www.khronos.org/sycl/

[17] https://www.khronos.org/registry/SYCL/specs/sycl-1.2.1.pdf

[18] https://github.com/intel/llvm/blob/sycl/sycl/doc/GetStartedGuide.md

[19] https://github.com/illuhad/hipSYCL

[20] https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.htmlshared-memory

[21] https://github.com/UoB-HPC/BabelStream

[22] https://github.com/ekondis/mixbench

978-1-6654-2369-4/21/$31.00 ©2021 IEEE
Authorized licensed use limited to: Universidad Industrial de Santander. Downloaded on October 11,2023 at 17:42:43 UTC from IEEE Xplore. Restrictions apply.