Benchmarking Operators in Deep Neural Networks for Improving Performance Portability of SYCL

Abstract. SYCL is a portable programming model for heterogeneous computing, so it is important to obtain reasonable performance portability of SYCL. Towards the goal of better understanding and improving performance portability of SYCL for machine learning workloads, we have been developing benchmarks for basic operators in deep neural networks (DNNs). These operators could be offloaded to accelerators such as graphics processing units (GPUs) to speed up computation. In this paper, we introduce the benchmarks, evaluate the performance of the operators on GPU-based systems, and describe the causes of the performance gap between the SYCL and CUDA kernels. We find that the causes are related to the utilization of the texture cache for read-only data, optimization of the memory accesses with strength reduction, use of local memory, and register usage per thread. We hope that the efforts of developing benchmarks for studying performance portability will stimulate discussion and interactions within the community.

Keywords: Performance Portability, Benchmarks, DNN operators.

1 Introduction

Computing platforms upon which workloads are evaluated differ in the details of the hardware accelerators and software stacks [1, 2, 3]. Vendor-specific programming libraries and languages have been addressing many of these differences. For example, Compute Unified Device Architecture (CUDA) [4] and Heterogeneous Computing Interface for Portability (HIP) [2] are common programming models for NVIDIA and AMD graphics processing units (GPUs), respectively. However, commonalities among these programming models exist and several portable programming methods allow for writing a program targeting multiple platforms [5, 6, 7]. A portable programming model, which facilitates the execution of a program across multiple computing platforms, could improve programming productivity and exploit performance potentials of programmable accelerators. In turn, the programming model may be improved in functionality and performance with the evaluation of applications and benchmarks. SYCL

is a royalty-free, cross-platform abstraction C++ programming model with an open and evolving specification for heterogeneous computing [8].

Previous studies evaluate SYCL by comparing the performance of benchmarks and applications in high-performance computing (HPC) on GPUs with vendors' programming models. In [9, 10, 11, 12, 13], the results show that whether the performance of running SYCL is competitive with using a vendor-specific programming model depends on the applications and how they are optimized by developers and compilers. For example, migrating the optimized bioinformatics workloads in CUDA required significant code changes, and the SYCL implementation was about 2X slower [10]. After converting the Rodinia benchmark suite [14] to a variant of SYCL, the researchers observe that some SYCL kernels achieve performance portability and other kernels see considerable overhead, varying from 25% to 190%, due to their execution of more GPU instructions and/or underutilization of GPU resources [12]. While vendor-specific programming models are recommended for their maturity in development environment and libraries for facilitating application development [15], it is important to study and improve performance and portability for the growth of portable programming models such as SYCL for heterogeneous computing.

However, obtaining reasonable performance portability requires nontrivial performance optimization for deep neural networks (DNNs) in SYCL [16]. The performance of a neural network implemented in SYCL without optimization is almost 50% slower, and the optimization could reach 90% of the performance of the CUDA DNN library. Towards the goal of better understanding and improving performance portability of SYCL for machine learning workloads, we have been developing a set of benchmarks for operators used in DNNs. DNNs are usually expressed as computation graphs in which nodes represent basic operations. These operators may be offloaded to accelerators to speed up computation [17]. In this paper, we focus on an introduction to the benchmarks and evaluating and understanding performance portability of SYCL using the benchmarks. In the following sections, we will give a summary of each benchmark, evaluate the benchmarks in SYCL and CUDA on NVIDIA GPUs, explain the performance gaps, discuss related work, and conclude the paper with future work.

2 Background

2.1 Brief Introduction to SYCL

Open Computing Language (OpenCL), a standard maintained by the Khronos group, has facilitated the development of parallel computing programs for execution on accelerators [18, 19]. However, writing an OpenCL program tends to be error-prone [20, 21]. Built on the underlying concepts, portability, and efficiency of OpenCL and ease of use and flexibility of single-source C++ [22], SYCL combines a host program and a device program for the simplicity of writing a single program like CUDA, and a compiler to statically type-check the correctness of the program. The SYCL buffer and

unified shared memory (USM) are two abstractions for data management [8]. USM is a pointer-based approach that is similar to the CUDA programming approach.

A routine, which is sent by an application to a graphics device for execution, is often called a "kernel" in GPU computing. In contrast to CUDA, a SYCL program requires a programmer to explicitly specify a queue to which kernels are submitted for execution on a device. A SYCL queue is either in-order or out-of-order. For an in-order queue, kernels are executed in the order they were submitted to the queue. For an out-of-order queue, kernels can be executed in an arbitrary order subject to the dependency constraints among them. Because CUDA kernels are executed in the order they were launched, we choose an in-order queue for the SYCL benchmarks for consistency.

2.2 Summary of the Benchmarks for DNN Operators

We have been developing a set of SYCL benchmarks for the operators based on open-source machine learning frameworks and applications. We will expand the set to represent more operations from DNN. This section is a summary of the benchmarks listed in alphabet order. Each benchmark is written with CUDA and SYCL.

Accuracy. The benchmark implements a function for computing prediction accuracy [23]. The kernel reads a label index from an "index" array, and then accesses the value of a predicted label (p) from a "label" array with the index. p is compared against each predicted label in the array. A counter is incremented when the label is larger than p. When two labels are equal, the counter is incremented based on the comparison of the labels' indices. After the comparison of all labels, the accuracy rate is incremented when the counter's value is below a threshold.

Adam. The benchmark computes individual adaptive learning rates for parameters from estimates of first and second moments of the gradients [24]. In the implementation of the benchmark [25], the kernel reads the scaled gradient, updates biased first moment and biased second raw moment estimate, computes bias-corrected first moment estimate and second raw moment estimate vector, and finally updates the parameter in each time step. The size of each parameter and the number of time steps are user-defined.

Attention. The benchmark implements a mechanism that pays attention to what is relevant to the currently processed information through content-based similarity search [26]. The mechanism is used in different domains of neural network [27]. The implementation contains three compute kernels. For a "query" vector with d dimensions and a "key" matrix with n vectors where each vector has d dimensions, the attention mechanism first computes a similarity score by a dot product of the "query" vector with each "key" vector. After this process, a vector of n dimensions is obtained. The vector is then processed with a function that normalize it into probabilities. Finally, the normalized vector is used as a weight to compute the weighted sum of vectors from the $n \times d$ value matrix.

ChannelShuffle. The benchmark implements the channel-shuffle function that divides a four-dimensional (4D) tensor into groups and rearranges (shuffle) them while maintaining the shape of the original tensor [28]. The benchmark evaluates the shuffling performance for the channel-first and channel-last orderings of a tensor.

ChannelSum. The benchmark implements the channel sum function that computes the per-channel sums of a 4D tensor [23]. The benchmark evaluates the sum performance for the channel-first and channel-last orderings of a 4D tensor. For the channel-first ordering, a 2D GPU thread block is assigned to compute the sums. For the channel-last ordering, a 1D GPU thread block is assigned to compute the sums. The sum reduction in a thread block is implemented using a library call for reusable software components such as CUB [29].

Clink. The benchmark performs inference of a one-hidden-layer *N*-node long short-term memory (LSTM) network. The network is a type of recurrent neural networks that can be used for temporal signal prediction tasks, such as handwriting recognition and speech recognition [30]. A typical LSTM network comprises a hidden layer and an output layer. The hidden layer consists of an input gate, a forget gate, a cell gate and an output gate, a cell node, and a hidden node [31]. At each time step, the network reads the input, updates the values of all the gates and the nodes, and then generates output based on the hidden node value. The original implementation was developed for energy-efficient signal processing on neurofeedback devices. The benchmark requires an input data file for evaluation.

concat. The benchmark concatenates two tensors into a new tensor [32]. The dimension of one tensor is *batch_size* × *beam_size* × *num_head* × *sequence1* × *hidden_dimension* and the dimension of the other tensor *batch_size* × *beam_size* × *num_head* × *sequence2* × *hidden_dimension*. The two tensors have the same shape except in the concatenating dimension.

CrossEntropy. The benchmark computes a loss function in the backward propagation phase. The implementation of the benchmark supports data types of half-, single-, and double-precision floating-point. The performance can be measured with the bandwidth metric [33].

DenseEmbedding. The benchmark computes the sum of input and dense values and stores the result in an output array [23]. The dimension of the dense array is *batch_size* × *embedding dimension*. The input and output arrays are accessed with a base address and an offset. The base address is read from an “offset” array indexed by the batch number while the offset falls within a range computed by the difference of two consecutive offset values.

Dwconv. The benchmark applies a 2D depth-wise convolution [34] over an input signal composed of several input planes. Each input channel is convolved with its own set of filters of size *K* [23]. In the implementation of the benchmark, *K* ranges from 1 to 4. The kernel sizes for the height and width dimension are 1, 3, or 5. The stride, padding, and dilation for both dimensions are one.

Expdist. The benchmark reduces the cross terms computed by a Gaussian transform [35]. In the implementation of the benchmark [36], the first kernel produces cross terms for two sets of points using a Gaussian kernel. The second kernel reduces these terms to a final sum in parallel.

Flip. The benchmark reverses the order of elements over axes of a tensor [23]. After the flip, the elements are reordered, but the shape of the array is preserved. The benchmark assumes that the order of elements over all axes of a tensor will be reversed, and the sizes of all dimensions are the same.

Gd. The benchmark implements gradient descent to solve a binary classification problem for sparse features [37]. The benchmark requires an input file for evaluation. The data are read from the file and stored in memory as a compressed sparse matrix. The number of iterations for training is 100 by default to reduce the training time.

Gelu. The benchmark applies the Gaussian error linear unit function [38] over the sum of a source array and a bias array. The approximate algorithm is "tanh" [23]. The source array is organized as a 3D tensor where a hidden dimension is dimension zero, a sequence length is dimension one, and a batch size is dimension two of the tensor. The bias array is a 1D array in hidden dimension. The two arrays are stored in memory using the half-precision floating-point format for reduced memory footprint.

Glu. The benchmark applies a gated linear unit function over a tensor. The gating mechanism is useful for language mdel as it allows a model to select which words or features are relevant for predicting the next work [39]. In the implementation of the benchmark, the split dimension must be divisible by two. It is assumed that the sizes of all dimensions of a tensor are the same.

Logprob. The benchmark computes the log probability of each token in a batch of sequences [23]. In the implementation of the benchmark, the first kernel applies the softmax function [40] to an input array of vocabulary size. The second kernel accumulates the probabilities along the sequence dimension in a batch of sequences.

Mask. The benchmark applies mask operations over a region [23]. The mask types are a sequence mask, a window mask, masks of upper and lower parts of a matrix. Each mask operation is implemented as a kernel. When a mask is applied, the output value

is set with a predefined value (e.g., -1). When not masked, the output value is equal to the input value.

Maxpool3d. The benchmark applies 3D maxpooling, a form of filtering commonly used in convolutional neural network, over an image set [41]. The size of the window to take a maximum over is two, and the stride of the window is equal to the size of the window.

Meanshift. The benchmark is an implementation of the mean shift clustering algorithm [42]. There are two implementations of the algorithms. One implementation takes a tiling approach by using shared local memory available in GPUs while the other does not utilize any shared memory.

Multinomial. The benchmark returns a tensor where each row contains an index (i.e., one sample) sampled from the multinomial probability distribution located in the corresponding row of the input tensor. When the input values are weights instead of probabilities, the weights will be normalized to probabilities. The largest index where the distribution is non-zero will be selected from the distribution [23].

3 Evaluation

3.1 Experimental Setup

We evaluate the benchmarks on four GPU-based computing nodes. On the first node (P1), the host is an Intel Xeon E5-2698 v4 processor and the device an NVIDIA V100 DGXS GPU with 32 GB device memory. On the second node (P2), the host is an AMD Ryzen Threadripper 3970X processor and the device an NVIDIA GeForce RTX 3090 GPU with 24 GB memory. On the third node (P3), the host is an Intel Xeon E5-2683 v4 processor and the device an NVIDIA P100 GPU with 12 GB memory. On the fourth node (P4), the host is an Intel Core i7-9700 processor and the device an NVIDIA GeForce RTX 2080 GPU with 8 GB memory. We compile the SYCL programs targeting NVIDIA GPUs with the prebuilt CUDA plugin from Codeplay [43] and the Intel oneAPI toolkit, version 2023.2.0. The versions of the CUDA runtimes in the NVIDIA HPC software development kits installed on the nodes are 11.8 and up. The optimization option is "-O3" and the offloading GPU architectures are "sm_60", "sm_70", "sm_75", and "sm_86" for both compilers. We run each benchmark four times, and each run executes the kernel(s) for at least 100 iterations. For each benchmark, we select the minimum average kernel execution time for performance evaluation.

3.2 Experimental Results

Figure 1 shows the ratios of the SYCL kernel time to the CUDA kernel time on the four nodes. When the ratio is over one, it means that the SYCL kernel time is longer than

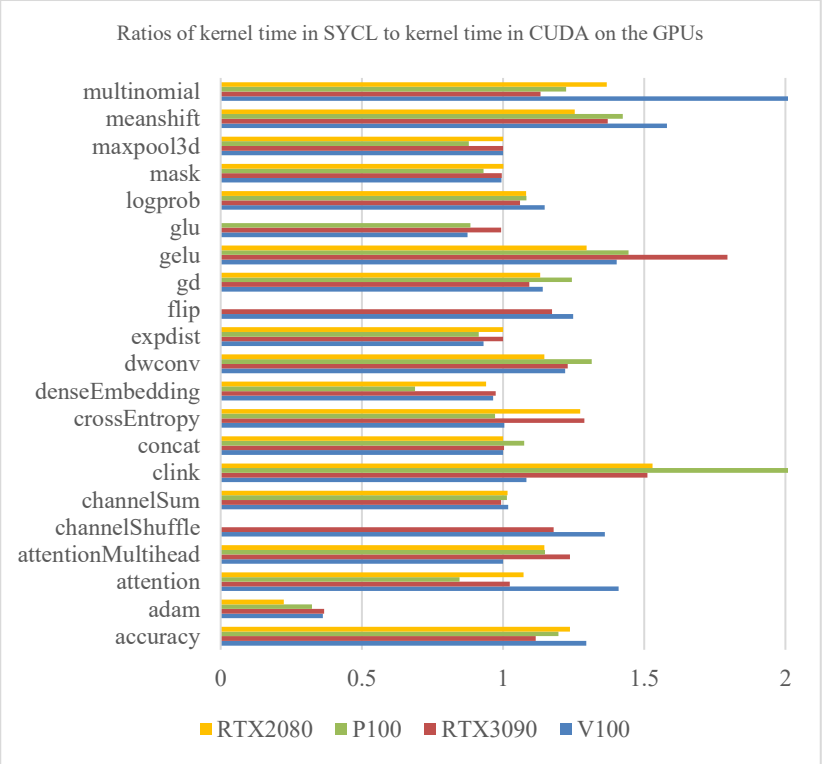

Fig. 1. Comparison of the SYCL and CUDA kernel time on the four GPU-based computing nodes

the CUDA kernel time. Because of the limited sizes of the GPU device memories on P3 and P4, the timing results of the "channelShuffle" and "flip" benchmarks in SYCL and CUDA are not available. In addition, the result of the "glu" benchmark is not available on P4. For each node, we compute the average ratio across all benchmarks that produce valid timing results to measure performance portability of the SYCL kernels. The average ratios on the four nodes are 1.148, 1.121, 1.099, and 1.095, respectively. Hence, there exist a performance gap between the CUDA and SYCL kernels on the NVIDIA GPUs.

Understanding the Performance Gap. Compiler optimizations are often evasive for application developers, so we attempt to find the causes of the performance gaps based on the analyses of the GPU assembly codes generated by the compilers and the results of profiling the kernels using the vendor's performance profiler.

Utilization of the Texture Cache for Read-only Data. On NVIDIA GPU architectures, the texture cache often has higher bandwidth and longer latency than the global memory cache, so it may offer higher performance for an application with sufficient parallelism

to cover the longer latency. On the other hand, the cache can only be used for data that is read-only for the lifetime of the kernel. The CUDA compiler is not sure that a pointer in a CUDA kernel references read-only data unless the pointer is marked with both "const" and "__restrict__" [44].

The SYCL compiler has implemented an experimental extension to the CUDA backend to allow read-only data to be cached in the texture cache. However, the SYCL compiler needs to know which data will be cached explicitly from a programmer. To enable the feature in the SYCL compiler, a SYCL kernel must call the specific function, as shown in Listing 1, in the device code to cache data. The function will call the appropriate low-level builtin function based on the type of the data the pointer points to. For example, in the "clink" benchmark, we can cache input and output read-only weights and biases in the LSTM network to improve the kernel performance and obtain performance portability.

```
namespace sycl::ext::oneapi::experimental::cuda {
  template<typename T>
  T ldg(const T* ptr);
} // namespace sycl::ext::oneapi::experimental::cuda
```

Listing. 2. The SYCL templated function allows users to load a register variable to the non-coherent read-only texture cache [45].

Straight-Line Strength Reduction. Programs, which access arrays for matrix multiply or dot product, usually have unrolled loops (either unrolled automatically by a compiler or manually by a programmer) that iterate over an array with a fixed access pattern. The expressions that compute the indices or pointer addresses of these accesses may be partially redundant [46]. For example, in the "attention" benchmark, the last kernel performs dot product by accumulating over the product of the score and value elements. The relevant code snippets of the kernel are shown in Listing 2. This computation order does not eliminate the partial redundancy between `(i+1)*d` and `(i+2)*d`. However, `(i+2)*d` could be replaced with `(i+1)*d+d` that takes only one extra add operation.

```
float sum = 0;
for (int i = 0; i < n; i++)
  sum += score[i] * value[i * d + j];
```

Listing. 2. The code snippets for describing the straight-line strength reduction

We find that both compilers can automatically unroll the loop in Listing 2 to increase instruction-level parallelism. In Listing 3.a, the loop is manually unrolled by a factor of four to illustrate the effect. However, the SYCL compiler emits inefficient code in terms of addressing the "value" array by following the source code. In contrast, the CUDA compiler optimizes the addressing of the strided elements of the "value" array with a constant offset for the add operations in each loop iteration. Listing 3.b shows the optimization applied in the source code.

```
1 for (int i = 0; i < n; i += 4) {
2   sum += score[i] * value[i*d + j]
3   sum += score[i+1] * value[(i+1)*d + j]
4   sum += score[i+2] * value[(i+2)*d + j]
5   sum += score[i+3] * value[(i+3)*d + j]
6 }
```

Listing 3.a. After loop unrolling

```
1   for (int i = 0; i < n; i++) {
2     p0 = &value[i*d+j]
3     sum += score[i] * (*p0)
4     p1 = p0 + d
5     sum += score[i+1] * (*p1)
6     p2 = p1 + d
7     sum += score[i+2] * (*p2)
8     p3 = p2 + d
9     sum += score[i+3] * (*p3)
10  }
```

Listing 3.b. After strength reduction

```
1  ld.global.nc.f32     %f12, [%rd29];
2  ld.global.nc.f32     %f13, [%rd28];
3  fma.rn.f32           %f14, %f13, %f12, %f29;
4  add.s64              %rd20, %rd29, %rd4;
5  ld.global.nc.f32     %f15, [%rd20];
6  ld.global.nc.f32     %f16, [%rd28+4];
7  fma.rn.f32           %f17, %f16, %f15, %f14;
8  add.s64              %rd21, %rd20, %rd4;
9  ld.global.nc.f32     %f18, [%rd21];
10 ld.global.nc.f32     %f19, [%rd28+8];
11 fma.rn.f32           %f20, %f19, %f18, %f17;
12 add.s64              %rd22, %rd21, %rd4;
13 add.s64              %rd29, %rd22, %rd4;
14 ld.global.nc.f32     %f21, [%rd22];
15 ld.global.nc.f32     %f22, [%rd28+12];
16 fma.rn.f32           %f29, %f22, %f21, %f20;
17 add.s32              %r22, %r22, 4;
18 add.s64              %rd28, %rd28, 16;
```

Listing 4.a. Assembly codes generated by the CUDA compiler for the code snippets in Listing 2

```
1  cvta.global.u64     %rd15, %rd14;
2  ld.global.nc.f32    %f6, [%rd25];
3  ld.global.nc.f32    %f7, [%rd15];
4  fma.rn.ftz.f32      %f8, %f7, %f6, %f18;
5  add.s64             %rd17, %rd14, %rd16;
6  cvta.global.u64     %rd18, %rd17;
7  ld.global.nc.f32    %f9, [%rd25+4];
8  ld.global.nc.f32    %f10, [%rd18];
9  fma.rn.ftz.f32      %f11, %f10, %f9, %f8;
10 add.s64             %rd19, %rd17, %rd16;
11 cvta.global.u64     %rd20, %rd19;
12 ld.global.nc.f32    %f12, [%rd25+8];
13 ld.global.nc.f32    %f13, [%rd20];
14 fma.rn.ftz.f32      %f14, %f13, %f12, %f11;
15 add.s64             %rd21, %rd20, %rd16;
16 ld.global.nc.f32    %f15, [%rd25+12];
17 ld.global.nc.f32    %f16, [%rd21];
18 fma.rn.ftz.f32      %f18, %f16, %f15, %f14;
19 add.s32             %r14, %r14, 4;
20 add.s64             %rd25, %rd25, 16;
```

Listing 4.b. Assembly codes generated by the SYCL compiler after applying the strength reduction manually

Listing 4.a shows the assembly codes generated by the CUDA compiler for the code snippets in Listing 2. Analyzing the codes (L4, L8, L12) indicates that the compiler can apply the optimization of strength reduction automatically. Listing 4.b shows the codes generated by the SYCL compiler after applying the optimization manually. We observe that the type conversion instructions (L1, L6, L11) are generated by the SYCL compiler to convert 64-bit signed numbers to 64-bit unsigned numbers. These instructions may be optimized away when 64-bit signed numbers are considered valid memory addresses for the load instructions.

```
1     .local .align 8 .b8     __local_depot0[16];
2   .reg .b64                 %SP;
3   .reg .b64                 %SPL;
4    mov.u64                  %SPL, __local_depot0;
5    cvta.local.u64           %SP, %SPL;
6    mov.u32                  %r11, 0;
7    st.u32                   [%SP+8], %r11;
8    mov.u64                  %rd3, 0;
9    st.u64                   [%SP+0], %rd3;
```

Listing 5 Assembly codes generated by the SYCL compiler for local memory allocation and writes

*Local Memory Allocation***.** The local space is one of the state spaces defined in CUDA [43]. The local state space (.local) is private memory for each thread to keep its own data. We observe that the SYCL compiler allocates such space and stores the contents of registers in it. Listing 5 lists the assembly codes for memory allocation and writes generated from the "gelu" benchmark.

Local memory is usually allocated by the CUDA compiler when each thread needs to keep a local array in a kernel. However, such per-thread array does not exist in the benchmark kernels. Hence, optimizing away local memory in the SYCL compiler will reduce the number of issued and executed instructions, improving the throughput of memory accesses and the raw performance of the kernels.

*Register Usage Per Thread***.** Occupancy is the ratio of the number of active warps per multiprocessor to the maximum number of possible active warps on NVIDIA GPUs. Alternatively, it is the percentage of the hardware's ability to process warps that is active. While higher occupancy does not always equate to higher performance, low occupancy always affects the hardware's ability to hide memory latency, resulting in performance degradation. Register availability is an important factor to determine occupancy. Register storage allows threads to store variables in registers for fast accesses. However, the register resource must be shared among all threads resident on a multiprocessor. Registers are allocated to an entire thread block. When each thread block uses too many registers, the number of warps that can be resident on a multiprocessor is decreased, thereby lowering the occupancy of the multiprocessor.

Profiling the execution of the SYCL and CUDA kernels shows that a SYCL kernel may require more registers to store all variables specified in a kernel. For example, the numbers of registers used by each thread is 28 for the CUDA kernel and 48 for the SYCL kernel in the "multinomial" benchmarks on the V100 GPU. Hence, the theoretical occupancy of the SYCL kernel is only 50%. To reduce the register utilization of the SYCL kernel and improve the raw performance of the SYCL kernel, we can set the maximum number of registers per thread manually (i.e., -Xcuda-ptxas -maxrregcount) at compile-time.

4 Related Work

In addition to the descriptions of the performance gap between the SYCL and CUDA kernels in this paper, previous studies find other compiler optimizations that could improve performance portability of SYCL in scientific domains. In [47], the authors find that the SYCL compiler did not automatically unroll a nested loop in the epistasis detection kernel while the CUDA compiler fully unrolls the loop. Unrolling the loop manually with a compiler pragma can significantly improve the kernel performance. After evaluating a set of bioinformatics kernels in SYCL and CUDA, the authors find that the use of an out-of-order SYCL queue in a host program and the choices of the math function from the SYCL math library in device code can lead to the performance gaps on an NVIDIA GPU [11]. In addition, evaluating the CUDA and SYCL kernels for all-pairs distance calculation shows that the sizes of memory addresses, widths of memory

accesses, and sub-word accesses contribute to the performance gaps on an NVIDIA GPU [48]. In [49], the authors conduct a performance portability study of tensor contraction using SYCL. They find that one of the major performance differences compared to the CUDA programs arise from differences in register usage. The "__launch_bounds__" primitive in the CUDA programming language informs the CUDA compiler of the launch configuration. Then, the compiler will adjust resource usage based on the configuration. In a molecular docking case study [50], comparing the performance of the CUDA and SYCL applications show that 2X higher register pressure in SYCL causes 2X lower kernel occupancy on an NVIDIA GPU. In [51], the authors show that a newer version of the SYCL compiler reduces the number of divergent branches and instructions for atomic operations, but the CUDA compiler utilizes fewer registers, reducing the number of memory transfers involving shared memory and between global memory and the Level-1 cache. While the support of the launch configuration in SYCL is not complete, these studies indicate that optimizing register utilization of a SYCL kernel in the compiler is critical regardless of the specification of launch configuration by a programmer.

5 Conclusion

SYCL is a cross-platform programming model for heterogeneous computing. As a portable programming model, obtaining reasonable performance portability is important for application and compiler developers. In this paper, we introduce the benchmarks for DNN operators written in CUDA and SYCL, evaluate the performance of the kernels in the benchmarks on the four GPU-based computing platforms, and describe the causes of the performance gap by analyzing the assembly codes and profiling results from the toolchains. We find that the utilization of the texture cache for read-only data, optimization of the memory accesses with strength reduction, the use of local memory, and the register usage per thread contribute to the performance gap between the SYCL and CUDA kernels on the GPUs.

Currently, the implementation of the CUDA plugin in the SYCL compiler is more mature than that of the HIP plugin for AMD GPUs. Our future work will evaluate performance portability of SYCL on AMD GPUs with the development of the compiler from the community. We hope that our efforts of studying performance portability of SYCL with the development of benchmarks in multiple programming models will promote discussion, interactions, and feedback within the community.

References

1. Lindholm, E., Nickolls, J., Oberman, S. and Montrym, J., 2008. NVIDIA Tesla: A unified graphics and computing architecture. IEEE Micro, 28(2), pp.39-55.
2. Gutierrez, A., Beckmann, B.M., Dutu, A., Gross, J., LeBeane, M., Kalama-tianos, J., Kayiran, O., Poremba, M., Potter, B., Puthoor, S. and Sinclair, M.D., 2018, February. Lost in abstraction: Pitfalls of analyzing GPUs at the intermediate language level. In 2018 IEEE

International Symposium on High Performance Computer Architecture (HPCA) (pp. 608-619). IEEE.
3. Blythe, D., 2020, August. The Xe GPU Architecture. In 2020 IEEE Hot Chips 32 Symposium (HCS) (pp. 1-27). IEEE Computer Society.
4. Garland, M., Le Grand, S., Nickolls, J., Anderson, J., Hardwick, J., Morton, S., Phillips, E., Zhang, Y. and Volkov, V., 2008. Parallel computing experiences with CUDA. IEEE MICRO, 28(4), pp.13-27.
5. Portability Across DOE Office of Science HPC Facilities. [online] Available: https://performanceportability.org/
6. Trott, C.R., Lebrun-Grandié, D., Arndt, D., Ciesko, J., Dang, V., Elling-wood, N., Gayatri, R., Harvey, E., Hollman, D.S., Ibanez, D. and Liber, N., 2021. Kokkos 3: Programming Model Extensions for the Exascale Era. IEEE Transactions on Parallel and Distributed Systems, 33(4), pp.805-817.
7. Dagum, L. and Menon, R., 1998. OpenMP: an industry standard API for shared-memory programming. IEEE computational science and engineering, 5(1), pp.46-55.
8. SYCL 2020 Specification (revision 5) [online] https://www.khronos.org/registry/SYCL/specs/sycl-2020/html/sycl-2020.html
9. Homerding, B. and Tramm, J., 2020, April. Evaluating the Performance of the hipSYCL Toolchain for HPC Kernels on NVIDIA V100 GPUs. In Proceedings of the International Workshop on OpenCL (pp. 1-7).
10. Haseeb, M., Ding, N., Deslippe, J. and Awan, M., 2021, November. Evaluating Performance and Portability of a core bioinformatics kernel on multiple vendor GPUs. In 2021 International Workshop on Performance, Portability and Productivity in HPC (P3HPC) (pp. 68-78). IEEE
11. Jin, Z. and Vetter, J.S., 2022, December. Understanding performance portability of bioinformatics applications in SYCL on an NVIDIA GPU. In 2022 IEEE International Conference on Bioinformatics and Biomedicine (BIBM) (pp. 2190-2195). IEEE.
12. Castaño, G., Faqir-Rhazoui, Y., García, C. and Prieto-Matías, M., 2022. Evaluation of Intel's DPC++ Compatibility Tool in heterogeneous computing. Journal of Parallel and Distributed Computing, 165, pp.120-129.
13. Hardy, D.J., Choi, J., Jiang, W. and Tajkhorshid, E., 2022, May. Experiences Porting NAMD to the Data Parallel C++ Programming Model. In International Workshop on OpenCL (pp. 1-5).
14. Che, S., Boyer, M., Meng, J., Tarjan, D., Sheaffer, J.W., Lee, S.H. and Skadron, K., 2009, October. Rodinia: A benchmark suite for heterogeneous computing. In 2009 IEEE International S0ymposium on Workload Characterization (IISWC) (pp. 44-54). IEEE.
15. Marcel Breyer, Alexander Van Craen, and Dirk Pflüger. 2022. A Comparison of SYCL, OpenCL, CUDA, and OpenMP for Massively Parallel Support Vector Machine Classification on Multi-Vendor Hardware. In International Workshop on OpenCL (IWOCL'22). Association for Computing Machinery, New York, NY, USA, Article 2, 1–12. https://doi.org/10.1145/3529538.3529980
16. Tanvir, M., Narasimhan, K., Goli, M., El Farouki, O., Georgiev, S. and Ault, I., 2022, May. Towards performance portability of AI models using SYCL-DNN. In International Workshop on OpenCL (pp. 1-3).
17. Li, J., Cao, W., Dong, X., Li, G., Wang, X., Zhao, P., Liu, L. and Feng, X., 2021. Compiler-assisted Operator Template Library for DNN Accelerators. International Journal of Parallel Programming, 49, pp.628-645.
18. Munshi, A., Gaster, B., Mattson, T.G. and Ginsburg, D., 2011. OpenCL programming guide. Pearson Education.

19. Kaeli, D., Mistry, P., Schaa, D. and Zhang, D.P., 2015. Heterogeneous computing with OpenCL 2.0. Morgan Kaufmann.
20. Li, P., Brunet, E., Trahay, F., Parrot, C., Thomas, G. and Namyst, R., 2015, September. Automatic OpenCL code generation for multi-device heterogeneous architectures. In 2015 44th International Conference on Parallel Processing (pp. 959-968). IEEE.
21. Steuwer, M. and Gorlatch, S., 2014. SkelCL: a high-level extension of OpenCL for multi-GPU systems. The Journal of Supercomputing, 69(1), pp.25-33.
22. Stroustrup, B., 2013. The C++ Programming Language. Pearson Education.
23. Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L. and Desmaison, A., 2019. Pytorch: An imperative style, high-performance deep learning library. Advances in neural information processing systems, 32.
24. Kingma, D. P. & Ba, J. Adam: a method for stochastic optimization. In Proc. 3rd International Conference on Learning Representations (ICLR) (ICLR, 2015).
25. Li, S., Fang, J., Bian, Z., Liu, H., Liu, Y., Huang, H., Wang, B. and You, Y., 2021. Colossal-AI: A unified deep learning system for large-scale parallel training. arXiv preprint arXiv:2110.14883.
26. Ham, T.J., Jung, S.J., Kim, S., Oh, Y.H., Park, Y., Song, Y., Park, J.H., Lee, S., Park, K., Lee, J.W. and Jeong, D.K., 2020, February. A^ 3: Accelerating attention mechanisms in neural networks with approximation. In 2020 IEEE International Symposium on High Performance Computer Architecture (HPCA) (pp. 328-341). IEEE.
27. A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, “Attention is all you need,” in International Conference on Neural Information Processing Systems, NIPS, 2017.
28. Zhang, X., Zhou, X., Lin, M. and Sun, J., 2018. Shufflenet: An extremely efficient convolutional neural network for mobile devices. In Proceedings of the IEEE conference on computer vision and pattern recognition (pp. 6848-6856).
29. The NVIDIA CUB library, https://docs.nvidia.com/cuda/cub/index.html
30. Chen, Z., Howe, A., Blair, H.T. and Cong, J., 2018, July. CLINK: Compact LSTM inference kernel for energy efficient neurofeedback devices. In Proceedings of the International Symposium on Low Power Electronics and Design (pp. 1-6).
31. Hochreiter, S. and Schmidhuber, J., 1997. Long short-term memory. Neural computation, 9(8), pp.1735-1780.
32. Xiaohui Wang, Ying Xiong, Yang Wei, Mingxuan Wang, and Lei Li. 2021. LightSeq: A High Performance Inference Library for Transformers. In Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies: Industry Papers, pages 113–120
33. The Intel LLVM Github repository, https://github.com/intel/llvm/issues/5969
34. Howard, A.G., Zhu, M., Chen, B., Kalenichenko, D., Wang, W., Weyand, T., Andreetto, M. and Adam, H., 2017. Mobilenets: Efficient convolutional neural networks for mobile vision applications. arXiv preprint arXiv:1704.04861.
35. Schoonhoven, R., van Werkhoven, B. and Batenburg, K.J., 2022. Bench-marking optimization algorithms for auto-tuning GPU kernels. IEEE Transactions on Evolutionary Computation.
36. A software development tool for the creation of highly-optimized and tuned GPU applications, https://github.com/benvanwerkhoven/kernel_tuner
37. C++ implementation of Gradient Descent, Stochastic Gradient Descent for Sparse Data, https://github.com/CGudapati/BinaryClassification
38. Hendrycks, D. and Gimpel, K., 2016. Gaussian error linear units (GELUs). arXiv preprint arXiv:1606.08415.

39. Dauphin, Y.N., Fan, A., Auli, M. and Grangier, D., 2017, July. Language modeling with gated convolutional networks. In International conference on machine learning (pp. 933-941). PMLR.
40. Bengio, Y., Goodfellow, I. and Courville, A., 2017. Deep learning (Vol. 1). Cambridge, MA, USA: MIT press.
41. OpenCL Labs for PAPAA Summer School 2016 Edition, https://github.com/nachiket/papaa-opencl
42. Implementations of Mean Shift Clustering, https://github.com/w00zie/mean_shift
43. Reyes, R., Brown, G. and Burns, R., 2020, April. Bringing performant support for NVIDIA hardware to SYCL. In Proceedings of the International Workshop on OpenCL (pp. 1-1).
44. The CUDA programming guide. https://docs.nvidia.com/cuda/parallel-thread-execution/index.htm
45. The SYCL extensions implemented in the Intel LLVM compiler. https://github.com/intel/llvm/blob/sycl/sycl/doc/extensions/experimental/sycl_ext_oneapi_cuda_tex_cache_read.asciidoc
46. Wu, J., Belevich, A., Bendersky, E., Heffernan, M., Leary, C., Pienaar, J., Roune, B., Springer, R., Weng, X. and Hundt, R., 2016, February. gpucc: an open-source GPGPU compiler. In Proceedings of the 2016 International Symposium on Code Generation and Optimization (pp. 105-116).
47. Jin, Z. and Vetter, J.S., 2022, August. 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 (pp. 1-8).
48. Jin, Z. and Vetter, J.S., 2022, December. Understanding Performance Portability of Bioinformatics Applications in SYCL on an NVIDIA GPU. In 2022 IEEE International Conference on Bioinformatics and Biomedicine (BIBM) (pp. 2190-2195). IEEE.
49. Ozturk, M.E., Asudeh, O., Sabin, G., Sadayappan, P. and Sukumaran-Rajam, A., 2023, May. A Performance Portability Study Using Tensor Con-traction Benchmarks. In 2023 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW) (pp. 591-600). IEEE.
50. Leonardo Solis-Vasquez, Edward Mascarenhas, and Andreas Koch. 2023. Ex-periences Migrating CUDA to SYCL: A Molecular Docking Case Study. In Pro-ceedings of the 2023 International Workshop on OpenCL (IWOCL '23). Associ-ation for Computing Machinery, New York, NY, USA, Article 15, 1–11.
51. Marcel Breyer, Alexander Van Craen, and Dirk Pflüger. 2023. Performance Evolution of Different SYCL Implementations based on the Parallel Least Squares Support Vector Machine Library. In Proceedings of the 2023 Interna-tional Workshop on OpenCL (IWOCL '23). Association for Computing Machinery, New York, NY, USA, Article 24, 1–12.