ORNL/TM-2024/3463

Oak Ridge National Laboratory

Zheming Jin

July 2024

OAK RIDGE National Laboratory

ORNL IS MANAGED BY UT-BATTELLE LLC FOR THE US DEPARTMENT OF ENERGY

DOCUMENT AVAILABILITY

Reports produced after January 1, 1996, are generally available free via OSTI.GOV.

Website www.osti.gov

Reports produced before January 1, 1996, may be purchased by members of the public from the following source:

National Technical Information Service
5285 Port Royal Road
Springfield, VA 22161
Telephone 703-605-6000 (1-800-553-6847)
TDD 703-487-4639
Fax 703-605-6900
E-mail info@ntis.gov
Website http://classic.ntis.gov/

Reports are available to US Department of Energy (DOE) employees, DOE contractors, Energy Technology Data Exchange representatives, and International Nuclear Information System representatives from the following source:

Office of Scientific and Technical Information
PO Box 62
Oak Ridge, TN 37831
Telephone 865-576-8401
Fax 865-576-5728
E-mail reports@osti.gov
Website https://www.osti.gov/

This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor any agency thereof, nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof.

ORNL/TM-2024/3463

Computer Science and Mathematics Division

EVALUATING OPERATORS IN DEEP NEURAL NETWORKS FOR IMPROVING PERFORMANCE PORTABILITY OF SYCL

Zheming Jin

July 2024

Prepared by
OAK RIDGE NATIONAL LABORATORY
Oak Ridge, TN 37831
managed by
UT-BATTELLE LLC
for the
US DEPARTMENT OF ENERGY
under contract DE-AC05-00OR22725

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

Zheming Jin

Oak Ridge National Laboratory
Oak Ridge, TN, 37830, USA
jinz@ornl.gov

Abstract. SYCL is a portable programming model that strives to be performance-portable across heterogeneous computing devices. Towards understanding and improving performance portability of SYCL for machine learning workloads, benchmarks for basic operators in deep neural networks (DNNs) are developed for performance evaluation. These operators could be offloaded to heterogeneous computing devices such as graphics processing units (GPUs) to speed up computation. In this work, we introduce the benchmarks, evaluate the performance of the operators on GPU-based systems, and describe the potential causes of the performance gap between the SYCL and Compute Unified Device Architecture (CUDA) kernels. We find that the gaps are related to the utilization of the texture cache for read-only data, optimization of the memory accesses with strength reduction, shared local memory accesses, and register usage per thread. We hope that the benchmarks for studying performance portability will facilitate 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 languages and libraries have been addressing many of these differences. For example, CUDA [4] and Heterogeneous Computing Interface for Portability (HIP) [2] are advanced programming models for NVIDIA and AMD 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 promising programming model for various hardware accelerators. It is a royalty-free, cross-platform abstraction layer with an open and evolving specification for heterogeneous computing [8].

Previous studies evaluate SYCL by comparing the performance of applications and benchmarks 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 programs and how they are optimized by developers and compilers. For example, migrating an optimized bioinformatics workload 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 others see considerable overhead, varying from 25% to 190%, due to their execution of more GPU instructions and/or underutilization of GPU resources [12]. Vendor-specific programming models are mature in their toolchains and libraries for application development [15], but portable programming models could improve programming productivity with abstractions across vendors' computing devices. Hence, it is worthwhile to study and improve performance and portability of portable programming models such as SYCL.

On HPC systems with GPUs, obtaining reasonable performance portability requires nontrivial efforts for deep neural networks (DNNs) in SYCL. 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 [16]. Towards improving performance portability of SYCL for machine learning workloads, we have been collecting and developing benchmarks for operators used in DNNs. DNNs are typically expressed as computation graphs in which nodes represent basic operations. These operators may be offloaded to GPU accelerators to speed up computation [17]. 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. We hope that our work will facilitate discussion and interactions within the community.

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 hardware accelerators [18, 19]. However, writing an OpenCL program tends to be error-prone [20, 21]. Based 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 for 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]. We choose USM for data management because it is a pointer-based approach that is close to data management in CUDA.

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 could 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

The benchmarks for the operators are based upon open-source machine learning frameworks and applications. We will expand the benchmarks to represent more operations from DNN. This section is a summary of the benchmarks listed in alphabetic order. All the benchmarks are implemented in both CUDA and SYCL. These benchmarks are available at https://github.com/zjin-lcf/HeCBench.

Accuracy. The benchmark implements a function for computing prediction accuracy [23]. The kernel reads a label index from an "index" array, and then queries the value of a predicted label (*p*) from a "label" array with the index. The value *p* is compared against each predicted label in the array. A counter is incremented when the label's value 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, 25]. For each parameter in a timestep, the kernel computes a scaled gradient, updates a biased first moment and a biased second raw moment estimate, computes a bias-corrected first moment estimate and a second raw moment estimate, and finally adjusts the parameter based on the learning rate and corrected moments.

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 commonly used in different domains [27]. The benchmark contains three compute kernels that are executed consecutively. 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. The result is a vector of n dimensions. Then, it is processed with a softmax function. Finally, a dot product of the normalized vector with each column of the $n \times d$ "value" matrix produces the weighted sums of vectors.

ChannelShuffle. The benchmark implements the channel-shuffle function that divides a four-dimensional (4D) tensor into groups and rearranges 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 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 long short-term memory (LSTM) 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]. The network comprises a hidden layer and an output layer. The hidden layer consists of an input gate, a forget gate, a cell gate, an output gate, a cell node, and a hidden node [31]. The benchmark implements inference of a one-layer and five-node LSTM network. At each time step, the network reads an input, updates the values of all gates and nodes, and then generates an output based on the hidden node's value.

Concat. The benchmark concatenates two tensors into a new tensor [32]. The one tensor is of shape (*batch_size*, *beam_size*, *num_head*, *sequence1*, *hidden_dimension*) and 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 (the negative log-likelihood) in the backward propagation phase of training a neural network. The benchmark supports the data types of half-, single-, and double-precision floating-point formats. The performance can be measured with the bandwidth metric [33].

DenseEmbedding. The benchmark adds values from a dense embedding table and values from an input array and stores the sums in an output array [23]. The input and output arrays are each accessed with a base address and a stride. The base address is read from an "offset" array indexed by a "batch" number while the stride equals the embedding dimension. The access range is determined by the difference of two consecutive offset values in the "offset" array.

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 *m* filters [23]. *m* determines how many filters are applied to one input channel (i.e., the number of output channels generated per input channel). The height and width of each filter are 1, 3, or 5. The stride, padding, and dilation for both dimensions are one.

Expdist. The benchmark implements a simplified Bhattacharyya distance function between two sets of points [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 with logistic regression [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 for sparse features. 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 the first dimension, a sequence length is the second dimension, and a batch size is the third dimension. The bias array is a 1D array. The two arrays are stored in memory using the half-precision floating-point format for reduced memory footprint. The array elements are converted to single-precision floating-point numbers for the arithmetic operations.

Glu. The benchmark applies a gated linear unit function over a tensor. The gating mechanism is useful for language models 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 a log probability (a logarithm of a probability) of each token in a batch of sequences [23]. There are two compute kernels in the benchmark. The first kernel applies the log-softmax function [40] on the output values from the "logits" layer. The second kernel accumulates the probabilities along the sequence dimension in a batch of sequences.

Mask. The benchmark applies mask operations over a batch of regions [23]. The mask types include a sequence mask, a window mask, masks of upper and lower parts of a matrix. Each mask operation is implemented as a compute kernel. When a mask is applied, the output value is set with a predefined value. 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 filter is equal to the stride of the filter. The horizontal and vertical strides are the same.

Meanshift. The benchmark is an implementation of the mean shift clustering algorithm [42]. The algorithm computes the weights of nearby points by applying a Gaussian function on the squared distance to the current estimate for re-estimation of the mean. The process is repeated until the mean shift converges. There are two implementations in the benchmark. The optimized 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 finds the location in a multinomial probability distribution in which the value of a sample falls. When the values of the distribution are weights, they will be normalized to probabilities. A prefix sum (scan) operation is performed on the normalized probabilities before a sample is compared with the scan results to find the location. 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 GPU-based computing platforms in the Experimental Computing Lab (ExCL) at Oak Ridge National Laboratory. The ExCL provides diverse computational resources in terms of chips, memories, and storage. On the first platform (P1), the host has an AMD EPYC 7513 32-core processor, and the device is an NVIDIA Tesla V100 GPU with 32 GB memory. On the second platform (P2), the host has an Intel Core i7-9700 processor, and the device is an NVIDIA GeForce RTX 2080 GPU with 8 GB memory. On the third platform (P3), the host has an AMD EPYC 7513 32-core processor, and the device is an NVIDIA Tesla A100 GPU with 80 GB device memory. On the fourth platform (P4), the host has an AMD Ryzen Threadripper 3970X processor and the device is an NVIDIA GeForce RTX 3090 GPU with 24 GB memory. On the fifth platform (P5), the host has an AMD EPYC 9454 48-core processor, and the device is an NVIDIA H100 GPU with 94 GB memory. The SYCL programs are compiled with the Intel oneAPI toolkit and the oneAPI for NVIDIA GPUs plugin. The plugin adds a CUDA backend to the SYCL compiler [43]. The version of the toolkit is 2024.2.0. The CUDA versions from the NVIDIA system management interface are

12.2 and up. The versions of the CUDA compilation tools are 12.3 and 12.4. The offloading GPU architectures are “sm_70”, “sm_75”, “sm_80”, “sm_86” and “sm_90”.

There is a warmup run for each benchmark and each run executes GPU kernels for at least 100 iterations. When a benchmark produces multiple timing results, they are added together. For performance evaluation, we average the kernel execution time measured with the C++ chrono library. The kernel time includes the time of launching all kernels from a host processor, kernel execution time on a device, and the time of waiting for all kernels to complete.

3.2 Experimental Results

Figure 1 shows the ratios of the SYCL kernel time to the CUDA kernel time on the GPUs. When the ratio is over one, it means that the SYCL kernel time is longer than the CUDA kernel time. We find that the “glu” and “flip” benchmarks in SYCL cause a “CUDA out-of-memory” runtime error on P2, so their ratios are not available in the figure. The geometric means across all benchmarks on the four platforms are 1.021, 1.042, 1.053, 1.059 and 1.030, respectively. While most SYCL benchmarks achieve reasonable performance portability, there exist performance gaps for certain benchmarks across the platforms.

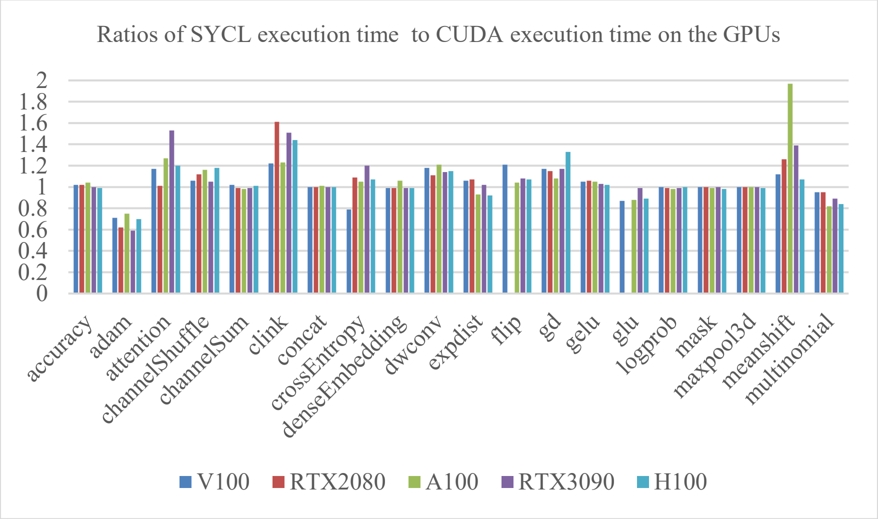

Fig. 1. Comparison of the SYCL and CUDA kernel time on the four computing platforms with NVIDIA GPUs

Understanding the Performance Gap. Compiler optimizations are often evasive for application developers and domain scientists, so the causes of the gap might be better understood by analyzing 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 does not assume 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 its 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 built-in function based on the type of the data the pointer points to. For example, in the "clink" benchmark, the input and output read-only weights and biases in the network could be cached to improve the kernel performance and performance portability.

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

Listing 1. 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 accumulates 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

```
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

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

Listing 3.a. Unrolling the loop by a factor of 4

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

Listing 3.b. Strength reduction

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.

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 may emit 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 manually in the source code.

Listing 4.a shows the assembly codes generated by the CUDA compiler for the code snippets in Listing 2. Analyzing the codes indicates that the compiler can apply the

```
ld.global.nc.f32      %f12, [%rd29];
ld.global.nc.f32      %f13, [%rd28];
fma.rn.f32            %f14, %f13, %f12, %f29;
add.s64               %rd20, %rd29, %rd4;
ld.global.nc.f32      %f15, [%rd20];
ld.global.nc.f32      %f16, [%rd28+4];
fma.rn.f32            %f17, %f16, %f15, %f14;
add.s64               %rd21, %rd20, %rd4;
ld.global.nc.f32      %f18, [%rd21];
ld.global.nc.f32      %f19, [%rd28+8];
fma.rn.f32            %f20, %f19, %f18, %f17;
add.s64               %rd22, %rd21, %rd4;
add.s64               %rd29, %rd22, %rd4;
ld.global.nc.f32      %f21, [%rd22];
ld.global.nc.f32      %f22, [%rd28+12];
fma.rn.f32            %f29, %f22, %f21, %f20;
add.s32               %r22, %r22, 4;
add.s64               %rd28, %rd28, 16;-
```

Listing 4.a. The GPU assembly codes generated by the CUDA compiler for the code snippets in Listing 2

```
cvta.global.u64     %rd15, %rd14;
ld.global.nc.f32    %f6, [%rd25];
ld.global.nc.f32    %f7, [%rd15];
fma.rn.ftz.f32      %f8, %f7, %f6, %f18;
add.s64             %rd17, %rd14, %rd16;
cvta.global.u64     %rd18, %rd17;
ld.global.nc.f32    %f9, [%rd25+4];
ld.global.nc.f32    %f10, [%rd18];
fma.rn.ftz.f32      %f11, %f10, %f9, %f8;
add.s64             %rd19, %rd17, %rd16;
cvta.global.u64     %rd20, %rd19;
ld.global.nc.f32    %f12, [%rd25+8];
ld.global.nc.f32    %f13, [%rd20];
fma.rn.ftz.f32      %f14, %f13, %f12, %f11;
add.s64             %rd21, %rd20, %rd16;
ld.global.nc.f32    %f15, [%rd25+12];
ld.global.nc.f32    %f16, [%rd21];
fma.rn.ftz.f32      %f18, %f16, %f15, %f14;
add.s32             %r14, %r14, 4;
add.s64             %rd25, %rd25, 16;
```

Listing 4.b. The GPU assembly codes generated by the SYCL compiler after applying the strength reduction manually

```
.shared .align 4 .b8  local_data[768]
.shared .align 4 .b8  valid_data[256]
 mov.u32          %r14, %ntid.x;
 mov.u32          %r15, %ctaid.x;
 mov.u32          %r1, %tid.x;
 mad.lo.s32       %r2, %r15, %r14, %r1;
 mul.lo.s32       %r16, %r2, 3;
 shl.b32          %r17, %r1, 2;
 mov.u32          %r18, valid_data;
 add.s32          %r3, %r18, %r17;
 mov.u32          %r19, local_data;
 mad.lo.s32       %r4, %r1, 12, %r19;
 … …
 st.shared.f32   [%r4], %f48;
 st.shared.f32   [%r4+4], %f49
 st.shared.f32   [%r4+8], %f79;
 st.shared.f32   [%r3], %f80;
```

Listing 5.a. The assembly code snippets generated by the CUDA compiler for the optimized "meanshift" kernel

```
.extern .shared .align 4 .b8  shared_mem[]
 ld.param.s32    %rd22, [param_0];
 ld.param.s32    %rd23, [param_1];
 mov.u64         %rd24,  shared_mem;
 add.s64         %rd25, %rd24, %rd23;
 add.s64         %rd26, %rd24, %rd22;
 mov.u32         %r1, %ntid.x;
 cvt.u64.u32     %rd27, %r1;
 mov.u32         %r2, %ctaid.x;
 mov.u32         %r3, %tid.x;
 cvt.u32.u64     %r4, %rd31;
 mul.lo.s32      %r5, %r3, 3;
 mul.wide.s32    %rd32, %r3, 4;
 add.s64         %rd1, %rd25, %rd32;
 mul.wide.u32    %rd33, %r5, 4;
 add.s64         %rd2, %rd26, %rd33;
 add.s32         %r6, %r5, 1;
 mul.wide.u32    %rd34, %r6, 4;
 add.s64         %rd3, %rd26, %rd34;
 add.s32         %r7, %r5, 2;
 mul.wide.u32    %rd35, %r7, 4;
 add.s64         %rd4, %rd26, %rd35;
 add.s64         %rd7, %rd25, 4;
 add.s64         %rd8, %rd26, 12;
 … …
 st.shared.f32   [%rd2], %f72;
 st.shared.f32   [%rd3], %f73;
 st.shared.f32   [%rd4], %f74;
 st.shared.f32   [%rd1], %f75;
```

Listing 5.b. The assembly code snippets generated by the SYCL compiler for the optimized "meanshift" kernel

optimization of strength reduction (L4, L8, L12) 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.

*Shared Local Memory Accesses***.** We find that the SYCL compiler may generate more instructions for addressing shared local memory in a kernel. Listing 5.a shows the assembly code snippets generated by the CUDA compiler for the optimized version of the "meanshift" kernel. In the optimized version, two single-precision floating-point arrays, "local_data" and "valid_data", are statically allocated in the CUDA shared memory space (L1 and L2). The 32-bit base addresses to the arrays are stored in registers using two "move" instructions (L9 and L11). In contrast, the SYCL local accessors allocate device local memory for the two arrays and then pass the underlying pointers to the kernel function. The assembly code snippets in Listing 5.b indicates that only a single array will be dynamically allocated (L1). The 64-bit base address to this array is stored in a register (L4). Then, the load instructions load 32-bit address variables from two kernel function parameters for computing the 64-bit base addresses to the two "logical" arrays.

In the optimized kernel, each thread stores three consecutive elements to "local_data" and one element to "valid_data". In Listing 5.a, the CUDA compiler generates three "store" instructions with a base register and constant offsets to store the register contents to the shared local memory whose addresses are computed by the sum of the base register and the offsets. On the other hand, the SYCL compiler generates three "store" instructions that use a unique register each for addressing the local array. The content of each register needs to be computed with a multiply instruction that multiplies two 32-bit numbers to produce a 64-bit result and a 64-bit add instruction. The two instructions may be combined to a single multiply-and-add instruction. Comparing the two addressing modes shows that the CUDA compiler could generate fewer instructions with more efficient register allocation for accessing shared local memory.

*Register Usage Per Thread***.** Occupancy is the ratio of the number of active warps per multiprocessor (MP) 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 are 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 MP is decreased, thereby lowering the occupancy of the MP. When the SYCL compiler causes high register utilization of a SYCL kernel, programmers may explore the impact of the register usage upon the kernel performance by adjusting the maximum number of registers per thread manually (e.g., `-Xcuda-ptxas -maxrregcount=32`) at compile-time.

4 Related Work

Previous studies described other compiler optimizations that could improve performance portability of SYCL in scientific domains. In [47], the authors find that the SYCL compiler did not unroll a nested loop automatically in the epistasis detection kernel while the CUDA compiler fully unrolls the loop. Unrolling the loop manually with a compiler pragma significantly improves 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 a math function from the SYCL math library in device code can lead to 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 CUDA informs the compiler of the launch configuration. The compiler could adjust resource usage based on the configuration. In a molecular docking case study [50], comparing the performance of the CUDA and SYCL applications shows 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.

5 Conclusion

SYCL is a cross-platform programming model with an open and evolving specification for heterogeneous computing. As a portable programming model, obtaining reasonable performance portability is important for both 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, the optimization of the memory accesses with strength reduction, the accesses of shared local memory, and the register usage per thread contribute to the performance gap between the SYCL and CUDA kernels on NVIDIA GPUs.

Our future work will evaluate performance portability of the SYCL implementation on other vendors' devices. We hope that the efforts of studying performance portability of SYCL with the development of benchmarks in multiple programming languages will promote discussion, interactions, and feedback within the community.

Acknowledgment

The reviewers' comments and suggestions are much appreciated. This research used resources of the Experimental Computing Laboratory at the Oak Ridge National Laboratory. This manuscript has been authored by UT-Battelle LLC under contract no. DE-AC05-00OR22725 with the US Department of Energy.

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. Willemsen, F.J., van Nieuwpoort, R. and van Werkhoven, B., 2021, November. Bayesian Optimization for auto-tuning GPU kernels. In 2021 International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS) (pp. 106-117). IEEE.
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., 2023, May. Understanding Performance Portability of SYCL Kernels: A Case Study with the All-Pairs Distance Calculation in Bioinformatics on GPUs. In 2023 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW) (pp. 366-372). 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. Experiences Migrating CUDA to SYCL: A Molecular Docking Case Study. In Proceedings of the 2023 International Workshop on OpenCL (IWOCL '23). Association 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 International Workshop on OpenCL (IWOCL '23). Association for Computing Machinery, New York, NY, USA, Article 24, 1–12.