EasyChair Preprint

№ 6531

An Experimental Study of SYCL Task Graph Parallelism for Large-Scale Machine Learning Workloads

Cheng-Hsiang Chiu, Dian-Lun Lin and Tsung-Wei Huang

September 1, 2021

An Experimental Study of SYCL Task Graph Parallelism for Large-Scale Machine Learning Workloads

Cheng-Hsiang Chiu, Dian-Lun Lin and Tsung-Wei Huang

University of Utah, Salt Lake City, UT, USA
cheng-hsiang.chiu@utah.edu
dian-lun.lin@utah.edu
tsung-wei.huang@utah.edu

Abstract. Task graph parallelism has emerged as an important tool to efficiently execute large machine learning workloads on GPUs. Users describe a GPU workload in a *task dependency graph* rather than aggregated GPU operations and dependencies, allowing the runtime to run whole-graph scheduling optimization to significantly improve the performance. While the new *CUDA graph* execution model has demonstrated significant success on this front, the counterpart for SYCL, a general-purpose heterogeneous programming model using standard C++, remains nascent. Unlike CUDA graph, the SYCL runtime leverages out-of-order queues to implicitly create a task execution graph induced by data dependencies. For explicit task dependencies, users are responsible for creating SYCL events and synchronizing them at a non-negligible cost. Furthermore, there is no specialized graph execution model that allows users to offload a task graph directly onto a SYCL device in a similar way to CUDA graph. This paper conducts an experimental study of SYCL's default task graph parallelism by comparing it with CUDA graph on large-scale machine learning workloads in the recent HPEC Graph Challenge. Our result highlights the need for a new SYCL graph execution model in the standard.

1 Introduction

Modern GPUs are fast and, in many scenarios, the time taken by each GPU operation (e.g., kernel or memory copy) is now measured in microseconds. The overheads associated with the submission of each operation to the GPU, also at the microsecond scale, are becoming significant and can dominate the performance of a GPU algorithm. For instance, inferencing a large neural network launches many dependent kernels on partitioned data and models. If each of these operations is launched to the GPU separately and repetitively, the overheads can combine to form a significant overall degradation to performance.

To overcome the overheads of kernel calls, CUDA has recently introduced a new graph programming model, namely *CUDA graph* [10], that allows users to describe a large GPU workload in a single task graph and offload the task

graph directly onto a GPU using a single CPU call. This new execution model opens several exciting opportunities for further accelerating the performance of large-scale machine learning workloads that compose thousands of GPU operations (i.e., kernels and memory copies). For instance, the recent research at 2021 Nvidia GTC has shown over 3× performance improvement in TensorFlow by replacing stream-based execution with CUDA graph [23]. In the same line, our research of CUDA graph has achieved 2× speed-up over existing stream-based solutions in completing the inference workloads of large sparse deep neural networks (DNN) that compose more than 46K GPU operations and 69K dependencies [21].

In addition to CUDA, SYCL [8] has emerged as a promising alternative to GPU programming using completely standard C++. As more ML systems start leveraging SYCL to design their back-ends (e.g., Intel oneAPI [7]), enabling direct task graph parallelism on a SYCL device is a high priority for efficiently executing large-scale machine learning workloads that define thousands of GPU operations and dependencies. The default SYCL runtime counts on an *out-of-order* queue to dynamically construct a task execution graph for submitted kernels described in *command group function objects*. Task dependencies are implicitly inferred from data dependencies extracted from *accessor* objects. In a unified shared memory (USM) [5, 12] environment where accessors are not required, users must explicitly construct dependencies between submitted tasks and synchronize their events. This organization adds burdens to developers and can be error-prone because of tedious event management.

Consequently, this paper introduces a high-level programming interface called *syclFlow* [11] to express task graph parallelism with SYCL. We leverage the out-of-order property of the SYCL queue to design a simple and efficient scheduling algorithm using topological sort. We compare the performance of syclFlow with CUDA graph on a large-scale machine learning workload from the HPEC Sparse DNN Inference Challenge [6]. The largest DNN model spans 1920 layers each of 65536 neurons and composes over 46K GPU operations to complete the inference loop. Under the same kernel algorithm, SYCL can be up to 5× slower than CUDA graph as a result of execution overheads (e.g., submission calls, event synchronizations). The experiment results highlight the need for a new SYCL graph execution model that allows explicit task graph parallelism on a SYCL device.

2 The Proposed SYCL Task Graph Programming Model

Our SYCL task graph programming model, *syclFlow*, enables users to describe workloads in a *task dependency graph*. Once a task graph is given, we schedule and submit dependent tasks to the SYCL runtime using out-of-order queue and event synchronization.

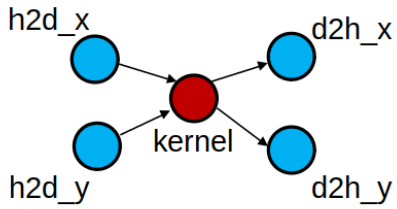

Fig. 1: An example of GPU task graph. There are one kernel, denoted in a red circle, and four memory copies, denoted in blue circles.

2.1 Task Graph Construction in syclFlow

syclFlow allows users to construct a task dependency graph using standard C++17 and SYCL20 based on USM. Figure 1 illustrates a GPU task graph of five tasks (one kernel, `kernel`, two host-to-device memory copies, `h2d_x` and `h2d_y`, and two device-to-host memory copies, `d2h_x`, and `d2h_y`) and four dependencies. Listing 1.1 implements Figure 1 using the proposed syclFlow programming model. We create a syclFlow object (`sf`), use `parallel_for` and `copy` to construct five task graph nodes, and relate the dependencies between nodes using `precede` and `succeed`. The code explains itself, inspired by our Taskflow project [18].

Listing 1.1: Example code of Figure 1 using syclFlow.

```
syclFlow sf;
syclTask h2d_x = sf.copy(dx, hx, size);
syclTask h2d_y = sf.copy(dy, hy, size);
syclTask kernel = sf.parallel_for(dx, dy);
syclTask d2h_x = sf.copy(hx, dx, size);
syclTask d2h_y = sf.copy(hy, dy, size);
kernel.succeed(h2d_x, h2d_y);
kernel.precede(d2h_x, d2h_y);
```

2.2 Task Graph Scheduling in syclFlow

Since syclFlow uses SYCL's out-of-order queue in which the SYCL runtime may not schedule tasks in the same order of their submissions, we have to schedule a user's task dependency graph before submitting tasks to SYCL. Algorithm 1 presents our scheduler. In Line 1, we apply topological sort algorithm to sort a user's task dependency graph and get the sorted graph in `T`. In Lines 2-4, we submit each task to the queue and get an event back. In Line 5, we synchronize the whole execution by `queue.wait`. This scheduling is the notable overhead that syclFlow has on top of SYCL. Please refer to Section 3 for detailed runtime breakdown. Algorithm 2 briefs the `submit` function. In Line 1, we declare a

command group function object for a task. In Lines 2-4, we use `depends_on` to specify the dependencies between the task and its dependents and encapsulate the dependencies together with the task in the command group function object. In Line 5, we return the command group function object as an event.

Algorithm 1: syclFlow's scheduler

Input: G: syclFlow task dependency graph defined by users
Input: *queue*: a SYCL queue associated with a SYCL device
1 $T \leftarrow$ topological_sort(G)
2 **for** $task \in T$ **do**
3 | $task.event = queue.submit(task)$
4 **end**
5 $queue.wait()$

Algorithm 2: queue.submit

Input: *task*: a user's task
1 $cgf \leftarrow$ create_command_group_function_object($task$)
2 **for** $p \in task.dependents$ **do**
3 | $cgf.depends\_on(p.event)$
4 **end**
5 return_event(cgf)

Listing 1.2: Example code of syclFlow::on to directly create a SYCL task.

```
syclFlow sf;
syclTask task = sf.on(
  [=](sycl::handler& handler) {
    handler.require(accessor);
    handler.single_task([=](){
      data[0] = 1;
    });
  }
);
```

Figure 2 demonstrates how syclFlow offloads a task dependency graph to a SYCL device and interacts with the SYCL runtime using an out-of-order queue and the `depends_on` method to schedule dependent tasks. The SYCL runtime schedules the submitted command group function objects and implicitly constructs its task graph based on submitted events.

syclFlow also allows users to exploit the full functionality of SYCL using `on` method to directly create a SYCL task from a command group object. Listing 1.2

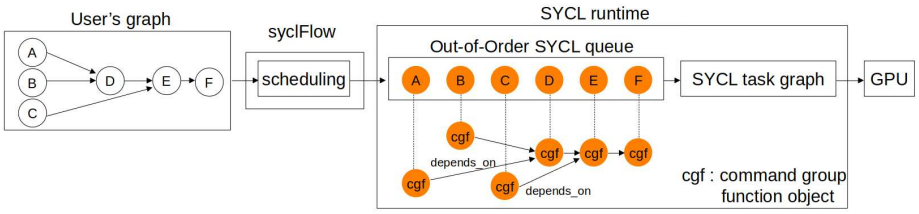

Fig. 2: syclFlow offloads a user-specified task dependency graph to a SYCL device. The SYCL runtime schedules command group function objects from out-of-order queue and constructs a task graph based on submitted events. Every arrow between two cgfs denotes a dependency using `depends_on` method.

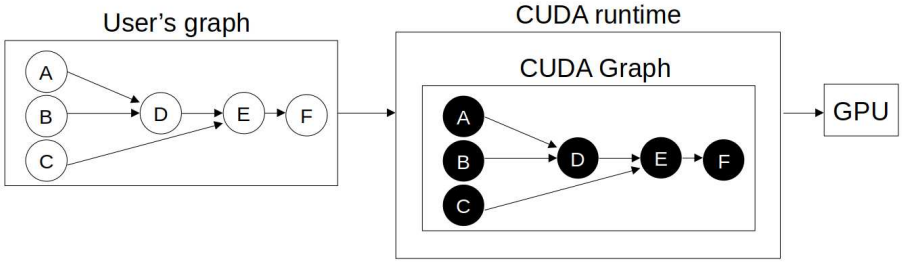

Fig. 3: Transformation of a user's task dependency graph to CUDA graph.

demonstrates the usage of `on` which takes a command group object to perform a task assigning a constant value, 1, to the element in a data array.

It is worth noting that the CUDA graph execution model is very different from the SYCL runtime, as shown in Figure 3. The CUDA runtime directly transforms a given CUDA graph into an executable graph in no need of additional user-level scheduling. This organization also allows the CUDA runtime to perform whole-graph optimization to significantly improve the performance. The synchronization overhead is minimized because CUDA graph does not synchronize tasks but the whole graph at once. That is, the synchronization overhead is limited to the number of CUDA graph submissions rather than the size of the graph.

3 Experimental Results

We demonstrate the significance of GPU task graph parallelism on large-scale machine learning workloads. The goal of this experiment is to highlight the need for a new SYCL graph execution model that does not require additional scheduling at the user level. We base our experiment on IEEE HPEC Sparse Deep Neural Network Inference Challenge [6]. The challenge is to speed up the computation of inference on extremely large DNNs. Table 1 shows the statistics of the benchmarks. We leverage the award-winning algorithm [21] to design our SYCL kernels and task graph parallelism with syclFlow. Figure 4 shows a partial task graph of our algorithm. We run the experiments on a Ubuntu Linux 20.04.2 LTS (Focal Fossa) x86 64-bit machine with Intel(R) Core(TM) i7-9700K Processor at 3.6 GHz, one GeForce RTX 2080 GPU with 8 GB memory, and 32 GB RAM. All programs are compiled by using Nvidia CUDA nvcc 11.1 on a host compiler of DPC++ clang [4] with C++17 standards.

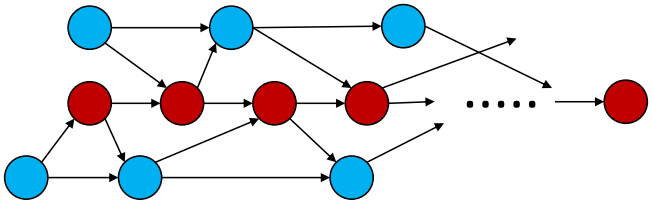

Fig. 4: Schematic view of a partial task graph in the inference workload based on our algorithm in [21]. A blue node represents a memory copy, and a red node denotes a kernel. The entire task graph on the largest DNN composes 3K tasks and 5K dependencies.

Performance comparison. Table 1 compares the elapsed runtime (in seconds) between syclFlow and CUDA graph on executing 12 DNN models using one

Table 1: Comparison of the total execution time between syclFlow and cudaGraph for completing 12 DNN models.

Model					syclFlow	cudaGraph
#Neurons	#Layers	#Tasks	#Dependencies	Size	Time	Time
1024	120	246	364	1.25 GB	0.55 s	0.47 s
	480	966	1444		1.86 s	1.53 s
	1920	3846	5764		6.98 s	5.79 s
4096	120	246	364	5.40 GB	1.96 s	1.48 s
	480	966	1444		6.85 s	5.11 s
	1920	3846	5764		26.32 s	19.66 s
16384	120	246	364	22.70 GB	8.94 s	4.36 s
	480	966	1444		30.51 s	14.82 s
	1920	3846	5764		146.82 s	57.23 s
65536	120	246	364	94.70 GB	80.08 s	17.29 s
	480	966	1444		273.29 s	51.92 s
	1920	3846	5764		> 600 s	162.20 s

GPU. The execution time of syclFlow is longer than CUDA graph across all models. For example, in the DNN model of 4096 neurons and 1920 layers, it takes 26.32 seconds for syclFlow to complete, whereas CUDA graph can finish in 19.66 seconds. In addition, the gap between syclFlow and CUDA graph keeps increasing as we enlarge the size of the DNN models. For instance, in the largest model of 65536 neurons and 480 layers, syclFlow is more than 5× slower than CUDA graph. Figure 5 visualizes the trend.

Synchronization overhead. Figure 6 lists the number of event synchronizations of syclFlow and CUDA graph on completing the inference of the DNN models with 1024 neurons. The default number of batch iterations in the inference loop is 12 [21]. Since CUDA graph constructs the graph only once when users submit a task dependency graph to the CUDA runtime, the number of event synchronizations is equal to the number of submissions. However, syclFlow requires frequent synchronizations between the user-level scheduler and the SYCL runtime. The number of synchronized events grows as we increase the number of layers in the DNN model, which in turn increases the size of the syclFlow graph. Figure 7 details the runtime breakdown of syclFlow and CUDA graph. We can easily see that in syclFlow event synchronizations consume 41%, which is as much as the kernel activities. While in CUDA graph, synchronization only costs 1.33%.

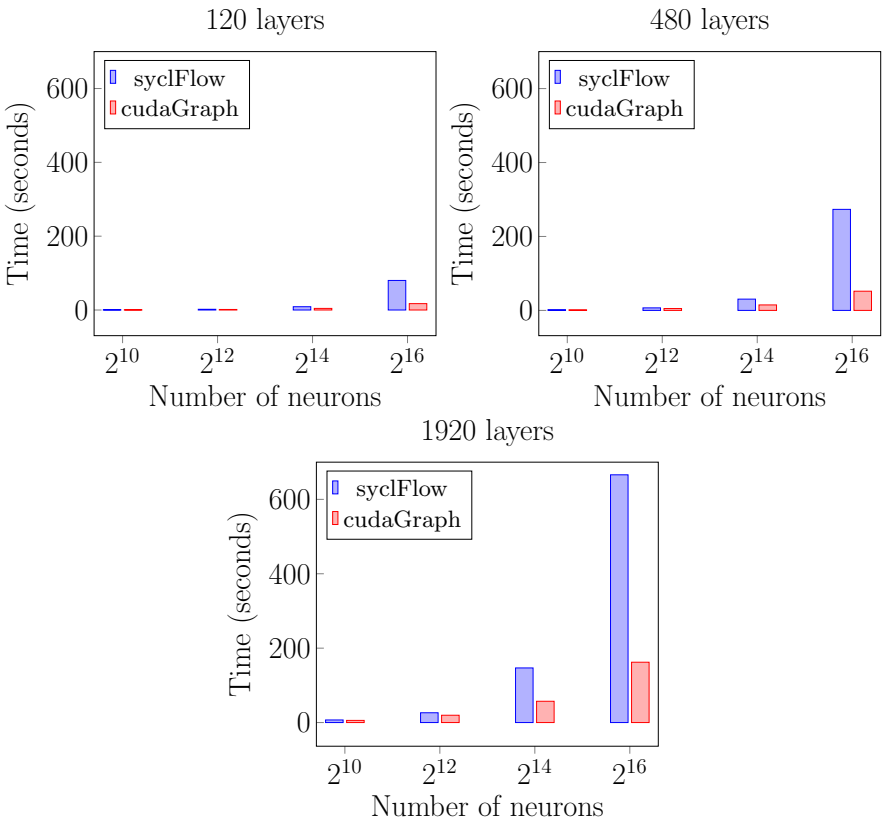

Fig. 5: Comparison of execution time between syclFlow and CUDA graph on different DNN models. The performance gap between syclFlow and CUDA graph increases as we enlarge the DNN model sizes.

Need for a New SYCL Graph Model. While we can devise a way to program and execute task graphs using the current SYCL standards (e.g., out-of-order queue, event synchronizations), this experiment highlights a critical need for a new SYCL graph execution model which allows us to *directly* offload task graph parallelism onto a SYCL device. This is especially important for accelerating large-scale machine learning workloads. Specifically, modern GPUs are very fast and the overhead of kernel calls and user-level scheduling have become very expensive in many machine learning task graphs that compose thousands of dependent GPU operators. These task graphs normally do not change once the neural network architecture is decided, and there is no need to repetitively offload the same task graph using expensive host function calls and scheduling methods.

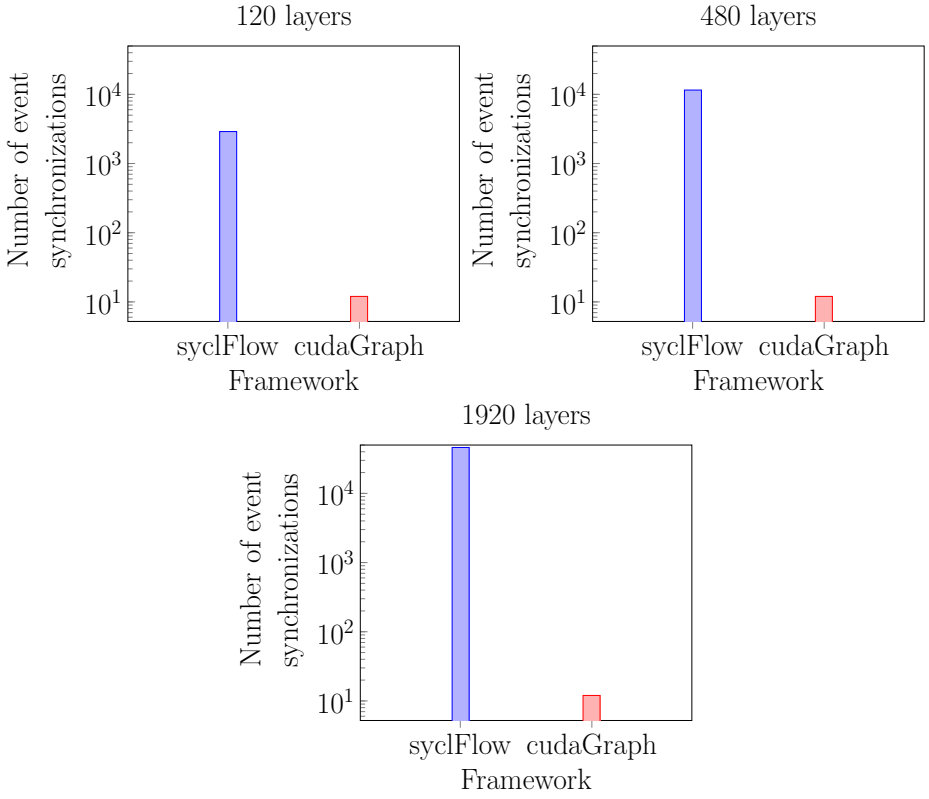

Fig. 6: Comparison of the number of event synchronizations between syclFlow and CUDA graph on DNN models with 1024 neurons. As the models increment, the synchronization overhead keeps the same for CUDA graph, whereas syclFlow suffers seriously from the overhead.

4 Related Work

Task graph-based programming models have received much attention over the last few years. Taskflow [18] develops a simple and powerful task programming model, which enables efficient implementations of heterogeneous decomposition strategies and leverages both static and dynamic task graph constructions to incorporate computational patterns. PaRSEC [14] expresses applications as DAG of tasks with labeled edges designating data dependencies. It provides a generic framework for architecture-aware scheduling and management of micro-tasks on distributed many-core heterogeneous architectures. Kokkos's functional approaches [15] provide task graph constructions. It allows applications to achieve performance portability on diverse many-core architectures. Legion [13] describes a runtime system that dynamically extracts parallelism from Legion programs, using a distributed, parallel scheduling algorithm that identifies both indepen-

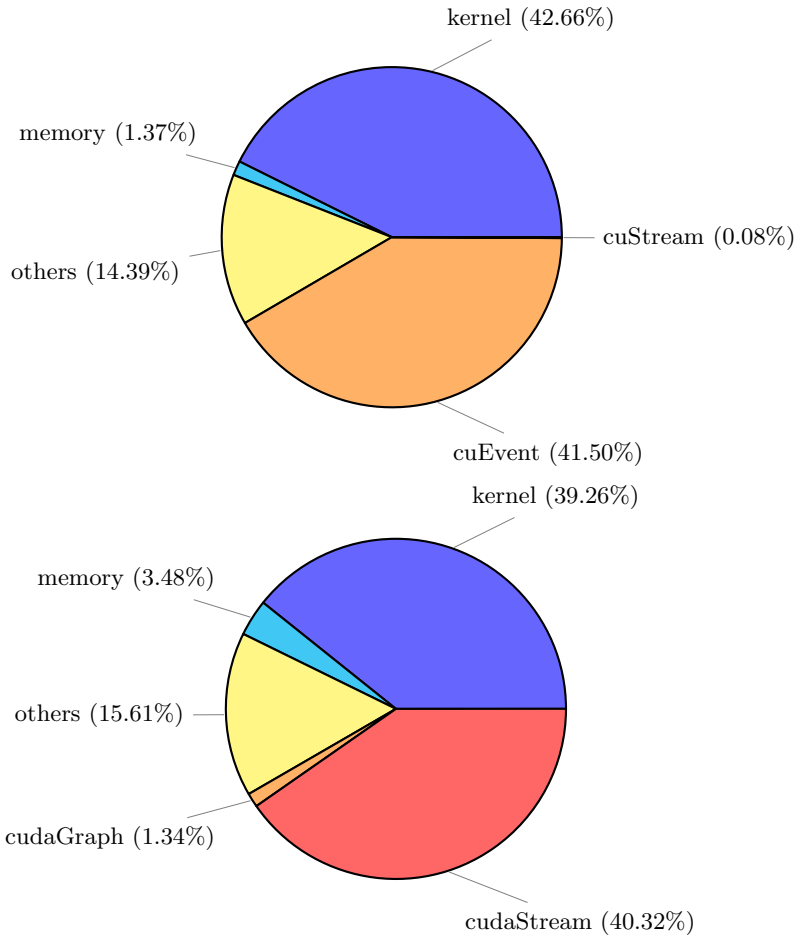

Fig. 7: Runtime breakdown of syclFlow (on the top, 1320.72 seconds in total) and CUDA graph (on the bottom, 1097.44 seconds in total) on a model with 120 layers and 1024 neurons. Kernel refers to the kernel activities, memory includes memory copy and memory set, cuEvent presents event-related APIs, cuStream contains stream-related APIs, cudaGraph denotes all of the APIs associated with cudaGraph, and cudaStream covers APIs about cudaStream.

dent tasks and nested parallelism. While these frameworks offer means to describe heterogeneous workloads in different forms of task graphs, they do not target direct task graph parallelism on a GPU.

CUDA graph is one of the early programming models that allow users to program task graph directly on a GPU. There are two ways to program a CUDA graph, explicit graph construction and implicit stream capturing. Explicit CUDA graph construction is often the most efficient, but it requires all the parameters

known upfront, which is impossible for many high-performance third-party libraries, such as cuSparse [3], cuBLAS [2], and cuDNN [9]. The second option is implicit graph construction, which captures a CUDA graph using existing stream-based application programming interfaces (APIs). Implicit CUDA graph construction is more flexible and general, allowing users to manually allocate and control streams. However, it requires users to wrangle with concurrency details through events and streams that are known difficult to program correctly. To simplify CUDA graph programming, Lin and Huang propose a unified interface coupled with an optimization method to program CUDA graph in both explicit and implicit modes [22].

SYCL is a programming model that allows users to write C++ single-source heterogeneous code. Users submit tasks to an out-of-order queue that is associated with a SYCL device (e.g., CPU, GPU, FPGA). The SYCL runtime schedules tasks from the out-of-order queue and constructs their dependencies based on user-specified events and/or data dependencies from buffer accessor objects. This type of task parallelism is different from CUDA graph, which takes the whole graph to schedule and performs whole-graph optimization to reduce overheads of synchronization and kernel calls.

5 Conclusion

In this paper, we have introduced syclFlow to enable efficient task graph programming using standard C++ and SYCL. We have compared the performance of syclFlow using the default task graph parallelism of the SYCL runtime with the new CUDA graph programming model on large-scale machine learning workloads. The experiments have shown that offloading task graph parallelism directly on a GPU can have a significant impact on the performance. For example, at the largest model of over 46K dependent GPU operations, CUDA graph can outperform syclFlow more than 5× faster. This paper signals a need for a new SYCL graph execution model that allows us to offload task graph parallelism directly on a SYCL device. At the time of this writing, we are actively collaborating with Codeplay [1] to design a new SYCL Graph standard through Khronos [8].

Our future work plans to design a source-code translation algorithm that automatically translates a written syclFlow code into a CUDA graph equivalent. We also plan to measure the performance difference between SYCL and CUDA graph in large-scale simulation problems [16, 17, 19, 20].

References

1. Codeplay Software Ltd https://codeplay.com
2. cuBLAS https://docs.nvidia.com/cuda/cublas/index.html
3. cuSPARSE https://docs.nvidia.com/cuda/cusparse/index.html
4. DPC++ Compiler https://intel.github.io/llvm-docs/GetStartedGuide.html
5. DPC++ Reference: Unified Shared Memory https://reurl.cc/5oYVOz

6. HPEC Sparse Deep Neural Network Inference Challenge https://graphchallenge.mit.edu/challenges
7. Intel oneAPI https://reurl.cc/2bpNzX
8. Khronos SYCL group https://www.khronos.org/sycl/
9. NVIDIA cdDNN https://developer.nvidia.com/cudnn
10. NVIDIA CUDA Graph https://developer.nvidia.com/blog/cuda-graphs/
11. syclFlow https://taskflow.github.io/taskflow/GPUTaskingsyclFlow.html
12. Unified Shared Memory https://reurl.cc/WElyRx
13. Bauer, M., Treichler, S., Slaughter, E., Aiken, A.: Legion: Expressing locality and independence with logical regions. In: SC'12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis. pp. 1–11. IEEE (2012)
14. Bosilca, G., Bouteiller, A., Danalis, A., Herault, T., Lemariner, P., Dongarra, J.: DAGuE: A Generic Distributed DAG Engine for High Performance Computing. pp. 1151–1158. IEEE, Anchorage, Alaska, USA (2011-00 2011)
15. Edwards, H.C., Trott, C.R., Sunderland, D.: Kokkos: Enabling manycore performance portability through polymorphic memory access patterns. Journal of Parallel and Distributed Computing **74**(12), 3202 – 3216 (2014)
16. Guo, G., Huang, T.W., Lin, Y., Wong, M.: GPU-accelerated Path-based Timing Analysis. In: ACM/IEEE Design Automation Conference (DAC) (2021)
17. Guo, Z., Huang, T.W., Lin, Y.: GPU-accelerated Static Timing Analysis. In: IEEE/ACM International Conference on Computer-aided Design (ICCAD). pp. 1–8 (2020)
18. Huang, T.W., Lin, C.X., Guo, G., Wong, M.: Cpp-taskflow: Fast task-based parallel programming using modern c++. In: 2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS). pp. 974–983. IEEE (2019)
19. Huang, T.W., Lin, C.X., Wong, M.D.F.: OpenTimer v2: A Parallel Incremental Timing Analysis Engine. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD) **40**(4), 776–789 (2021)
20. Huang, T.W., Wong, M.: OpenTimer: A high-performance timing analysis tool. In: IEEE/ACM International Conference on Computer-aided Design (ICCAD). pp. 895–902 (2015)
21. Lin, D.L., Huang, T.W.: A Novel Inference Algorithm for Large Sparse Neural Network using Task Graph Parallelism. In: 2020 IEEE High Performance Extreme Computing Conference (HPEC). pp. 1–7. IEEE (2020)
22. Lin, D.L., Huang, T.W.: Efficient GPU Computation using Task Graph Parallelism. In: 2021 IEEE/ACM European Conference on Parallel and Distributed Computing. IEEE (2021)
23. Yao, J., Li, C.: CUDA Graph in TensorFlow. Nvidia GPU Technology Conference (GTC)