Task Graph-based Parallelism using DOCA Graph

Dian-Lun Lin
dlin57@wisc.edu

Cheng-Hsiang Chiu
chenghsiang.chiu@wisc.edu

1 Introduction

Task graph programming systems (TGPSs) play a crucial role in many scientific computing applications, such as machine learning, graph algorithms, and simulation. Unlike loop-based models that execute tasks across independent iterations, TGPS encapsulates function calls and their dependencies in a top-down *task graph*. This task graph model enables applications to efficiently implement irregular parallel decomposition strategies and scale dependent tasks to a large number of processors. As a result, the parallel computing community has yielded many TGPS solutions for different application domains, such as Taskflow [11], cudaFlow [14] oneTBB FlowGraph [4], Kokkos-DAG [10], HPX [12], StarPU [6], TPL [13], Legion [7], PaRSEC [8], and Fastflow [5].

Recently, TGPSs has been adopted in accelerators such as GPU [9, 14]. 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 address this issue, CUDA has recently introduced a new CUDA graph programming model to enable efficient launch and execution of GPU work. CUDA graph enables a define-once-run-repeatedly execution flow that reduces the overhead of kernel launching. Users describe dependent GPU operations in a task graph rather than aggregated single operations. The CUDA runtime can perform whole-graph optimization and launch the entire graph in a single CPU operation to reduce overhead.

DOCA also provides a programming model, DOCA Graph, to facilitate running a DAG. DOCA Graph creates graph instances and submits the instances to the work queue for execution. In this final project, we dive into DOCA Graph's programming model to run an application. Compared to sequential and Pthread-based solutions, DOCA Graph achieves up to 101× and 94.1× speedup, respectively.

2 Our DOCA Graph Example

We demonstrated the use of DOCA Graph programming model [3] with an example. In the example, we have three tasks: 1) DPU calculates a SHA value (denoted as `SHA`) 2) DMA copies a string from a source buffer to a destination buffer (denoted as `DMA`), and 3) Host prints out the SHA value and compares the match between the source buffer and the destination buffer (denoted as `Host`). As the `SHA` task and the `DMA` task are independent to each other and both must finish before the `Host` task starts. we can describe the three tasks and the dependencies as a task graph using DOCA Graph programming model. Figure 1 illustrates the task graph.

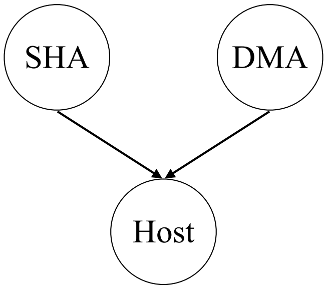


Figure 1: Illustration of the task graph. Circles denote the tasks and edges denote the dependencies.

To implement the task graph using DOCA Graph programming model, there are nine steps:

1. Create the graph using `doca_graph_create` API. The API create a doca graph object `graph`.

```
doca_graph_create(graph);
```

2. Create the graph nodes. To create a context node using `doca_graph_ctx_node_create` API and an user node using `doca_graph_user_node_create`. For example, we create a context node `sha_node` with the job callable `SHA_JOB` in `graph`.

```
// Create a context node for SHA job
doca_graph_ctx_node_create(
  graph, SHA_JOB, sha_node);
// Create a context node for DMA job
doca_graph_ctx_node_create(
  graph, DMA_JOB, dma_node);
// Create an user node for Host job
doca_graph_user_node_create(
  graph, user_node_callback, user_node);
```

3. Define dependencies using `doca_graph_add_dependency` API. For example, we add the dependency from `sha_node` to `user_node` in `graph`.

```
// Add dependency from sha_node to user_node
doca_graph_add_dependency(
  graph, sha_node, user_node);
// Add dependency from dma_node to user_node
doca_graph_add_dependency(
  graph, dma_node, user_node);
```

4. Start the graph using `doca_graph_start` API.

```
doca_graph_start(graph);
```

5. Add the graph to a work queue using `doca_graph_workq_add` API if necessary. We created the same task graph several times and thus used the API to generate several graph instances.

```
doca_graph_workq_add(graph, work_queue);
```

6. Create the graph instance using `doca_graph_instance_create` API.

```
create_graph_instance(graph_instance);
```

7. Set the nodes data (e.g., `doca_graph_instance_set_ctx_node_data` API for context nodes). This step is to initialize the nodes.

```
doca_graph_instance_set_ctx_node_data(
  graph_instance, dma_node, dma_job,
  dma_job_event));
doca_graph_instance_set_ctx_node_data(
  graph_instance, sha_node, sha_job,
  sha_job_event));
```

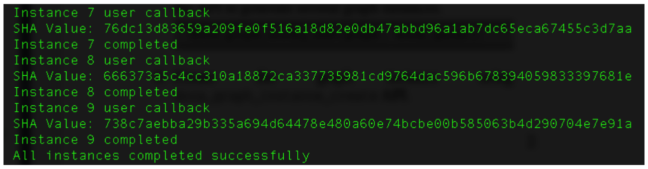
8. Submit the graph instance to the work queue using `doca_workq_graph_submit` API.

```
doca_workq_graph_submit(
  work_queue, graph_instance);
```

9. Call `doca_workq_progress_retrieve` until it returns `DOCA_SUCCESS`. We keep pooling the status of the work queue. When one graph instance finishes, we get `DOCA_SUCCESS` and increment the number of completed instances `completed_inst` by one. When all graph instances finish (`completed_inst` $\leq$ `ALL_INST`), we stop the program.

```
while (completed_inst < ALL_INST) {
  while (doca_workq_progress_retrieve(
  work_queue))!= DOCA_SUCCESS) {}
  completed_inst++;
}
```

By following the nine steps, we are able to create tasks and specify dependencies in a task graph, create multiple graph instances, and submit the graph instances to a work queue to execute. Figure 2 shows a snapshot of the execution of this example.



```
Instance 7 user callback
SHA Value: 76dc13d83659a209fe0f516a18d82e0db47abbd96a1ab7dc65eca67455c3d7aa
Instance 7 completed
Instance 8 user callback
SHA Value: 666373a5c4cc310a18872ca337735981cd9764dac596b678394059833397681e
Instance 8 completed
Instance 9 user callback
SHA Value: 738c7aebba29b335a694d64478e480a60e74bcbe00b585063b4d290704e7e91a
Instance 9 completed
All instances completed successfully
```

Figure 2: Snapshot of the example. The example in total generated ten graph instances.

3 Evaluations

We evaluated the performance of DOCA Graph on a microbenchmark. We studied the runtime performance. We compiled all programs using gcc 11.4. We ran all the experiments on a Ubuntu Linux 22.04 host with a AMD EPYC 7302 16-Core CPU at 128 GB RAM and a BlueField-2 DPU. We modified the `SHA` and `DMA` code presented in the DOCA application overview page [2].

3.1 Baseline

To evaluate the performance of DOCA Graph programming model, we chose the Pthread implementation as the baseline. In the Pthread implementation, we spawned one thread for the `SHA` task and one for the `DMA` task. We did not spawn another one thread for the `Host` task because we integrated the `Host` task to `SHA` and `DMA`. Listing 1 shows the Pthread implementation to run multiple instances. One instance includes the `SHA` task and the `DMA` task. We created two threads. One thread ran the `run_sha` function. The other thread ran the `run_dma` function. These two threads ran in parallel. We joined the two threads to end the instance.

```
int main(){
  pthread thread_sha, thread_dma;
  for (int i = 0; i < instances; ++i) {
    // Run SHA and DMA in parallel
    pthread_create(&thread_sha,NULL, run_sha, NULL);
    pthread_create(&thread_dma,NULL, run_dma, NULL);

    // Explicitly join the two threads
    pthread_join(thread_sha, NULL);
    pthread_join(thread_dma, NULL);
  }
}
```

Listing 1: Pthread implementation of running the SHA and the DMA task in parallel.

Moreover, we implemented a sequential program to justify the parallel execution of our Pthread implementation. In the sequential implementation, we ran the SHA task and the DMA task in sequence. Listing 2 shows the sequential implementation.

```
int main(){
  pthread thread;

  for (int i = 0; i < instances; ++i) {
    // Run SHA first
    pthread_create(&thread,NULL, run_sha, NULL);
    pthread_join(thread, NULL);

    // Run DMA later
    pthread_create(&thread,NULL, run_dma, NULL);
    pthread_join(thread, NULL);
  }
}
```

Listing 2: Sequential implementation of running the SHA and the DMA task in order.

3.2 Running the Experiment

Our source code includes the following five files:

- /src/graph_main.c: The main function for DOCA Graph implementation.
- /src/graph_sample.c: The function definitions for our DOCA Graph implementation.
- /src/pthread_sample.c: The Pthread implementation.
- /src/sequential_sample.c: The sequential implementation.
- run.sh: The script to compile and run the experiment.

In our source code, we have provided a script run.sh to compile and execute the code. To run the experiment, please follow the following steps:

```
ssh ubuntu@192.168.100.2
cd /opt/mellanox/doca/samples/doca_common/
unzip CS839-SmartNIC-main.zip
cd final_project
./run.sh
```

In the experiment, every implementation runs up to 500 instances. We used */usr/bin/time* to measure the runtime. The source code is available at the repository [1].

3.3 Runtime Comparison

Figure 3 compares the runtime performance between DOCA Graph and Pthread with up to 500 instances running. When running small numbers of instances, we find out that DOCA Graph implementation is not better than Pthread. For example, running with 1, 2, and 4 instance, DOCA Graph is less than or equal to Pthread. The reason is that building a graph has some overheads. Running small numbers of instances with DOCA Graph does not give us much runtime benefit. However, when running with more instances, the runtime improvement of DOCA Graph over Pthread is very obvious. For example, the speedup of DOCA Graph over Pthread is 23.6× 66.5×, and 94.1× when running with 100, 300, and 500 instances, respectively.

The runtime improvement of DOCA Graph over Pthread comes from two reasons. Firstly, Pthread needs to explicitly call pthread.join to synchronize between the two tasks. We did not know how DOCA Graph resolves the dependencies to synchronize between the tasks. Based on our experience, resolving the dependencies in a task graph could be done using lightweight atomic counter. That is, when we specify one dependency between the SHA and the Host task, Host node would have atomic counter one denoting one dependency. When SHA finishes, SHA atomically decrements Host's atomic counter by one. Once that counter reaches zero, the runtime schedules Host task. Therefore, the implementation using atomic counter is much lightweight than pthread.join.

Secondly, DOCA Graph builds the graph once and repetitively submits the same graph up to the number of instances for execution. From our experience, building the graph is very time-consuming. DOCA Graph can largely reduce the graph building overhead. Our Pthread implementation did not build the graph. However, we used explicit synchronization pthread.join to represent the dependencies in the task graph. When finishing one instance, our Pthread implementation needed to synchronize once. The number of total synchronizations is equal to the number of instances, which causes much overhead than DOCA Graph's one time graph building.

In Figure 3 we also show the runtime of sequential implementation to justify the parallel execution of our Pthread implementation. We can find that the sequential implementation is consistently slower than the Pthread implementation.

4 Work Distribution

The work distribution of the implementation is listed below.

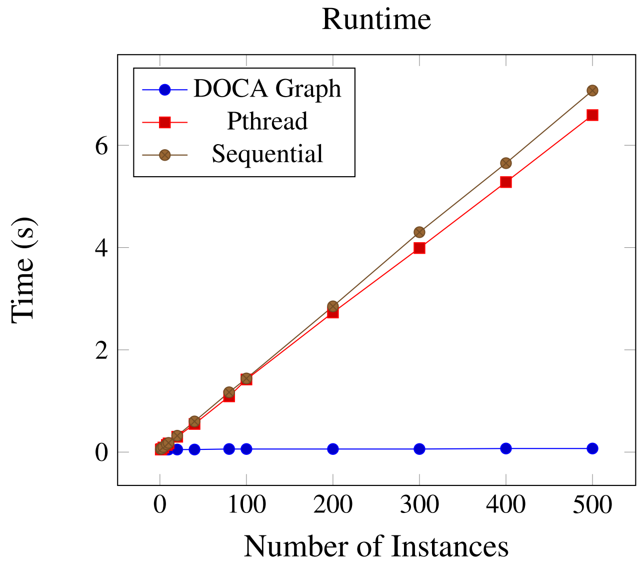


Figure 3: Runtime comparison between DOCA Graph, Pthread, and sequential.

- Dian-Lun: Implemented the DOCA Graph implementation.
- Cheng-Hsiang: Implemented the Pthread and sequential implementations.

For the presentation slide and report, Dian-Lun and Cheng-Hsiang finished them together.

5 Conclusion

In the final project, we have mentioned our motivation. We have presented how to implement a program using DOCA Graph programming model and presented one sample, in which three tasks `SHA`, `DMA`, and `Host` executed up to multiple instances. We have compared the runtime performance of DOCA Graph with a Pthread implementation, and a sequential implementation. Based on our experiences, we have provided two reasons to explain the runtime benefit of DOCA Graph over the baselines.

References

[1] Github repository. https://github.com/cheng-hsiang-chiu/CS839-SmartNIC.

[2] Nvidia DOCA Application Overview. https://docs.nvidia.com/doca/sdk/applications-overview/index.html.

[3] Nvidia DOCA Programming Guide. https://docs.nvidia.com/doca/sdk/doca-core-programming-guide/index.html.

[4] Intel onetbb.

[5] ALDINUCCI, M., DANELUTTO, M., KILPATRICK, P., AND TORQUATI, M. Fastflow: High-level and efficient streaming on multicore. *Programming multi-core and many-core computing systems* (2017).

[6] AUGONNET, C., THIBAULT, S., NAMYST, R., AND WACRENIER, P.-A. Starpu: a unified platform for task scheduling on heterogeneous multicore architectures. In *Euro-Par: Parallel Processing: 15th International Euro-Par Conference* (2009).

[7] BAUER, M., TREICHLER, S., SLAUGHTER, E., AND 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* (2012), pp. 1–11.

[8] BOSILCA, G., BOUTEILLER, A., DANALIS, A., FAVERGE, M., HERAULT, T., AND DONGARRA, J. J. Parsec: Exploiting heterogeneity to enhance scalability. *Computing in Science & Engineering* (2013), 36–45.

[9] CHIU, C.-H., LIN, D.-L., AND HUANG, T.-W. An Experimental Study of SYCL Task Graph Parallelism for Large-Scale Machine Learning Workloads. In *Euro-Par Workshop* (2022).

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

[11] HUANG, T.-W., LIN, D.-L., LIN, C.-X., AND LIN, Y. Taskflow: A lightweight parallel and heterogeneous task graph computing system. *IEEE Transactions on Parallel and Distributed Systems* (2022), 1303–1320.

[12] KAISER, H., HELLER, T., ADELSTEIN-LELBACH, B., SERIO, A., AND FEY, D. Hpx: A task based programming model in a global address space. In *Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models* (2014).

[13] LEIJEN, D., SCHULTE, W., AND BURCKHARDT, S. The design of a task parallel library. *SIGPLAN Not.* (2009), 227–242.

[14] LIN, D.-L., AND HUANG, T.-W. Efficient gpu computation using task graph parallelism. In *Euro-Par: Parallel Processing: 27th International Conference on Parallel and Distributed Computing, Proceedings* (2021), p. 435–450.