Final Project Report
CS 839 SmartNIC Systems

Dian-Lun Lin
dlin57@wisc.edu

Cheng-Hsiang Chiu
chenghsiang.chiu@wisc.edu

1 Problem Statement

Many scientific problems, such as machine learning, electronic design automation, and compiler intermediate representation, can be formulated as a dynamic acyclic graph (DAG), in which vertices denote the functions and edges denote the functional dependencies. To efficient schedule the functions and meet the dependency constraints between functions is challenging.

DOCA provides a programming model, DOCA Graph, to facilitate running a DAG. DOCA Graph creates graph instances and submit the instances to the work queue for execution.

In this final project, we would like to investigate the runtime performance of DOCA Graph by running a bunch of micro benchmarks. Each micro benchmark will execute some simple mathematical computations. The micro benchmarks include 1) linear chain, 2) embarrassing parallelism, 3) binary tree, and 4) random DAG. Figure 1 illustrates the four patterns.

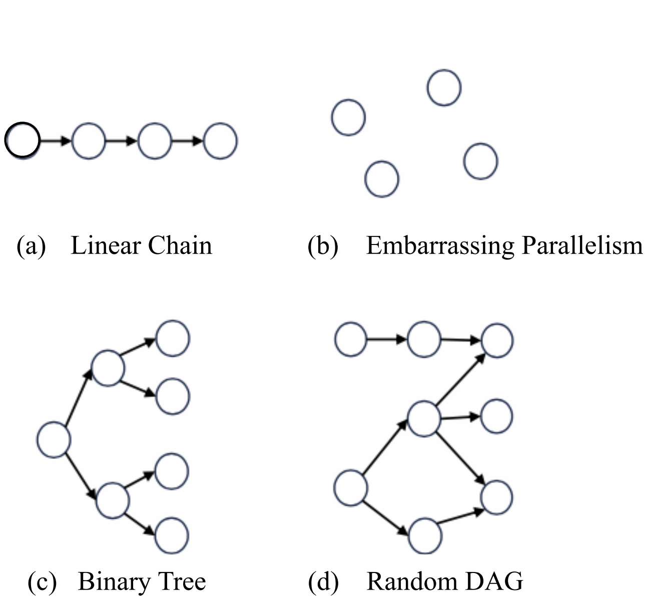


Figure 1: Illustration of four micro benchmarks.

2 Literature Review

There are some graph-related programming models developed for NVIDIA GPU. CUDA Graph is a new execution model that enables a series of CUDA kernels to be defined and encapsulated as a single unit, i.e., a task graph of operations, rather than a sequence of individually-launched operations. This organization allows launching multiple GPU operations through a single CPU operation and hence reduces the launching overheads, especially for kernels of short running time [3].

cudaFlow is an interface of Taskflow that manages a CUDA graph explicitly to execute dependent GPU operations in a single CPU call [2]. syclFlow is another graph exectuion model of Taskflow that allows users to offload a graph directly onto a SYCL device in a similar way to CUDA graph [1].

3 Our DOCA Graph Example

We demonstrated the use of DOCA Graph 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 2 illustrates the task graph.

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

1. Create the graph using `doca_graph_create` API. The

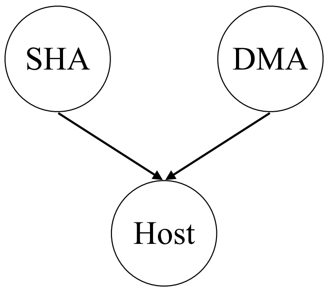


Figure 2: Illustration of the task graph. Circles denote the tasks and edges denote the dependencies.

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));
```

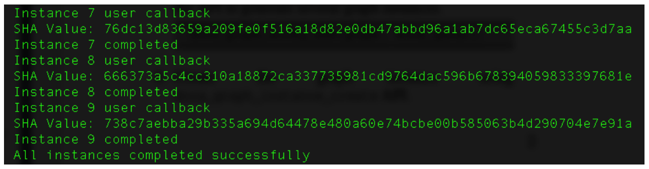
8. Submit the graph instance to the work queue using `doca_workq_graph_submit` API.

```
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 3 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: 666373a5c4cc310a18872ca337735981cd9764dac596b67839405983339 7681e
Instance 8 completed
Instance 9 user callback
SHA Value: 738c7aebba29b335a694d64478e480a60e74bcbe00b585063b4d290704e7e91a
Instance 9 completed
All instances completed successfully
```

Figure 3: Snapshot of the example. The example in total generated ten graph instances.

4 Evaluations

We evaluated the performance of DOCA Graph on a micro-benchmark. 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 at 128 GM RAM and a BlueField-2 DPU.

4.1 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(){
  for (int i = 0; i < instances; ++i) {
    pthread thread_sha, thread_dma;
    pthread_create(&thread_sha,NULL, run_sha, NULL);
    pthread_create(&thread_dma,NULL, run_dma, NULL);
    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.

4.2 Running the Experiment

Our source code includes the following five files:

- graph_main.c: The main function for DOCA Graph implementation.
- graph_sample.c: The function definitions for our DOCA Graph implementation.
- pthread_sample.c: The Pthread implementation.
- 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 follwing steps:

```
ssh ubuntu@192.168.100.2
cd /opt/mellanox/doca/samples/doca_common/
unzip final_project.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.

4.3 Runtime Comparison

Figure 4 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 performs the worst. For example, running with 1, 2, and 4 instance, DOCA Graph is 67.5×, 43×, and 21.3× slower than Pthread, respectively. The reason is that building a graph has some overheads. Running small numbers of instances with DOCA Graph does not give us any 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 13.8× 36.8×, and 55.66× 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 4 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.

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

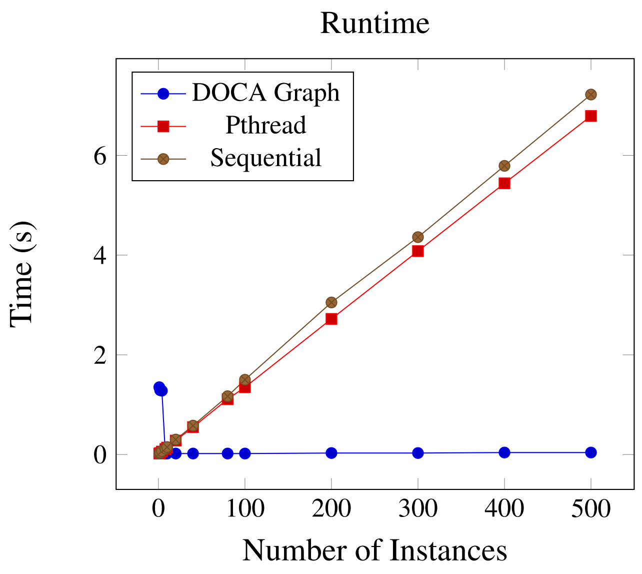


Figure 4: Runtime comparison between DOCA Graph, Pthread, and sequential.

provided two reasons to explain the runtime benefit of DOCA Graph over the baselines.

References

[1] 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).

[2] Lin, D.-L., and Huang, T.-W. Efficient GPU Computation Using Task Graph Parallelism. In *Euro-Par* (2021).

[3] NVIDIA. Cuda graph.