Efficient GPU Computation Using Task Graph Parallelism

Dian-Lun Lin(✉) and Tsung-Wei Huang

University of Utah, Utah, USA
{dian-lun.lin,tsung-wei.huang}@utah.edu

Abstract. Recently, CUDA introduces a new task graph programming model, *CUDA graph*, to enable efficient launch and execution of GPU work. Users describe a GPU workload in a task graph rather than aggregated GPU operations, allowing the CUDA runtime to perform whole-graph optimization and significantly reduce the kernel call overheads. However, programming CUDA graphs is extremely challenging. Users need to explicitly construct a graph with verbose parameter settings or implicitly capture a graph that requires complex dependency and concurrency managements using streams and events. To overcome this challenge, we introduce a lightweight task graph programming framework to enable efficient GPU computation using CUDA graph. Users can focus on high-level development of dependent GPU operations, while leaving all the intricate managements of stream concurrency and event dependency to our optimization algorithm. We have evaluated our framework and demonstrated its promising performance on both micro-benchmarks and a large-scale machine learning workload. The result also shows that our optimization algorithm achieves very comparable performance to an optimally-constructed graph and consumes much less GPU resource.

1 Introduction

The performance of GPU architectures continues to increase with every new generation. 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

L. Sousa et al. (Eds.): Euro-Par 2021, LNCS 12820, pp. 435–450, 2021.
https://doi.org/10.1007/978-3-030-85665-6_27

whole-graph optimization and launch the entire graph in a single CPU operation to reduce overheads [4,5].

However, programming CUDA graphs is extremely challenging. First, users can *explicitly* construct a CUDA graph that maps each vertex to a GPU operation and each edge to a dependency between two GPU operations. 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, cuBLAS, and cuDNN. Also, the CUDA runtime maximally parallelizes the given CUDA graph without limiting the stream usage. In large graphs, the GPU memory can explode. 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.

Consequently, we propose in this paper a lightweight task graph programming framework to enable efficient GPU computation using CUDA graph. Our framework introduces an *expressive* GPU task graph programming model for users to focus on high-level development of dependent GPU operations with relatively ease of programming. A written task graph is then cast to a native CUDA graph through our transformation algorithm optimized for kernel concurrency and graph size. The process is *transparent*. Users need not to handle any intricate concurrency details and dependency controls using streams and events. More importantly, we identify a research problem of optimizing CUDA graphs through stream capturers. The proposed research can assist CUDA developers in improving the performance of existing GPU applications through new CUDA graph parallelism.

2 The Proposed GPU Task Graph Programming Model

Our GPU task graph programming model consists of two parts, *cudaFlow* and *cudaFlowCapturer*, to handle explicit and implicit graph constructions in different use cases.

2.1 cudaFlow: Explicit CUDA Graph Construction

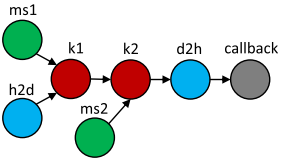


Fig. 1. An example of GPU task graph.

cudaFlow provides methods for users to explicitly construct a GPU task graph that presents a one-to-one mapping to a native CUDA graph. Each node in the task graph represents a GPU operation (copy, kernel, etc.), and each edge represents a dependency between two operations. Figure 1 shows a GPU task graph of seven nodes (two kernels, `k1` and `k2`, two typed copies, `h2d` and `d2h`, two untyped copies, `ms1` and `ms2`, and one host callback, `callback`) and six dependencies

(e.g., `k1`→`k2`). Listing 1.1 gives the implementation of Fig. 1 using our model. We create a cudaFlow object (`cf`) and use the four methods, `kernel`, `memset`, `copy`, and `host`, to create the seven task graph nodes and use the two methods, `succeed` and `precede`, to relate dependencies between nodes. The code *explains itself* through an expressive graph description language in just 12 lines of code. The same example but written in the plain CUDA graph model is partially shown in Listing 1.2, which requires more than 150 lines of code.

Listing 1.1. Example code of Fig. 1 using cudaFlow.

```
cudaFlow cf;
cudaTask h2d = cf.copy(inputVec_d, inputVec_h, inputSize);
cudaTask ms1 = cf.memset(outputVec_d, 0, input_size);
cudaTask ms2 = cf.memset(result_d, 0, 1);
cudaTask k1 = cf.kernel(reduce, inputVec_d, outputVec_d, inputSize);
cudaTask k2 = cf.kernel(reduce_final, outputVec_d, result_d);
cudaTask d2h = cf.copy(result_h, result_d, 1);
cudaTask callback = cf.host(fn, &hostFnData);
k1.succeed(h2d, ms1);
k2.succeed(k1, ms2);
k2.precede(d2h);
d2h.precede(callback);
```

Listing 1.2. Example code of Fig. 1 using the plain CUDA graph.

```
cudaStream_t streamForGraph;
cudaGraph_t graph;
std::vector<cudaGraphNode_t> nodeDependencies;
cudaGraphNode_t memcpyNode, kernelNode, memsetNode;
checkCudaErrors(cudaStreamCreate(&streamForGraph));
cudaKernelNodeParams kernelNodeParams = {0};
cudaMemcpy3DParms memcpyParams = {0};
cudaMemsetParams memsetParams = {0};
memcpyParams.srcArray = NULL;
memcpyParams.srcPos = make_cudaPos(0, 0, 0);
memcpyParams.srcPtr =
    make_cudaPitchedPtr(inputVec_h, sizeof(float) * inputSize, inputSize, 1);
memcpyParams.dstArray = NULL;
memcpyParams.dstPos = make_cudaPos(0, 0, 0);
memcpyParams.dstPtr =
    make_cudaPitchedPtr(inputVec_d, sizeof(float) * inputSize, inputSize, 1);
memcpyParams.extent = make_cudaExtent(sizeof(float) * inputSize, 1, 1);
memcpyParams.kind = cudaMemcpyHostToDevice;
checkCudaErrors(cudaGraphCreate(&graph, 0));
checkCudaErrors(
  cudaGraphAddMemcpyNode(&memcpyNode, graph, NULL, 0, &memcpyParams
));
//... more than 100 lines of code to follow
```

2.2 cudaFlowCapturer: Implicit CUDA Graph Construction

cudaFlow allows users to explicitly construct a CUDA graph, but it requires all execution parameters known in advance. This property restricts users from using commercial CUDA libraries, such as cuDNN and cuBLAS, that do not provide details for launching kernels but a public stream-based API. To overcome this restriction, we introduce cudaFlowCapturer with a stream-based method to capture GPU kernels and transform the given task graph into a native CUDA graph using our graph transformation algorithm. Listing 1.3 shows the cudaFlowCapturer code of Fig. 1, assuming the two kernels, `k1` and `k2`, are only invokable through a stream-based API. The cudaFlowCapture provides a method, `on`, that passes a stream created by our optimizer to the callable for users to capturer kernels or other asynchronous GPU operations.

Listing 1.3. Example code of Fig. 1 using cudaFlowCapturer.

```
cudaFlowCapturer cap;
cudaTask h2d = cap.copy(inputVec_d, inputVec_h, inputSize);
cudaTask ms1 = cap.memset(outputVec_d, 0, input_size);
cudaTask ms2 = cap.memset(result_d, 0, 1);
cudaTask k1 = cap.on([&](cudaStream_t stream){
  cublas_gemm(stream, my_paremeters...);
});
cudaTask k2 = cap.on([&](cudaStream_t stream){
  cublas_gemv(stream, my_paremeters...);
});
cudaTask d2h = cap.copy(result_h, result_d, 1);
cudaTask callback = cf.host(fn, &hostFnData);
k1.succeed(h2d, ms1);
k2.succeed(k1, ms2);
k2.precede(d2h);
d2h.precede(callback);
```

3 Transform a cudaFlowCapturer to a CUDA Graph

By default, we translate a cudaFlow directly into a native CUDA graph and use a single CPU call to offload the graph. To launch a cudaFlowCapturer, we need to transform the task graph defined in the cudaFlowCapturer into a native CUDA graph using stream capturer.

3.1 Problem Formulation

We describe the transformation problem as follows: Given a task graph G_t and the number of streams ($num\_streams$), discover an order to construct dependencies between nodes, i.e., assign each node $n \in G_t$ to a stream and decide an event for each node such that the execution order of tasks ("transformed CUDA graph") imposed by the streams and events is topologically identical to the original task

graph. The objective is to balance the load of each stream and minimize the transformed graph size. For example, using two streams, the task graph in Fig. 2(a) can be transformed into two different CUDA graphs, (b) and (c), both resulting in different critical paths and graph sizes. CUDA stream is in-order. Placing two dependent nodes at two different streams may require creating an event to build a dependency in the CUDA graph, as shown in the red points. Since the optimal number of streams is highly dependent on application level, we leave *num_streams* to users to tune the number of streams based on their applications.

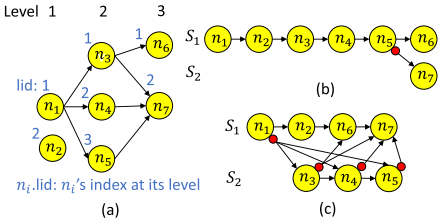


Fig. 2. Transformation of a task graph to a CUDA graph using two streams. (Color figure online)

This transformation problem has two challenges: Firstly, CUDA stream capture is stateful [1]. We can only construct a dependency in *one direction* from an assigned node to the node that is being enqueued to a stream. That is, optimizing the event count and, hence, the graph size, through back-and-forth traversal is not possible. Second, graph size matters. The same task graph can have many feasible transformations (see Fig. 2). Different transformations result in different execution efficiencies.

3.2 Our Algorithm: Round Robin with Dependency Pruning

At a high level, our algorithm assigns each node to a stream in a round-robin fashion and applies a dependency pruning to reduce redundant dependencies. We use Fig. 3 to illustrate our algorithm transforming the task graph of Fig. 2(a) to a CUDA graph using two streams. First, we levelize the task graph, G_t, to a 2D level list. Based on the 2D level list, we assign each node n_i to indicate the index of the topological ordering of G_t, and $n_i.lid$ to indicate the index of its level (see Fig. 2(a)). We assign each node to a stream of id equal to $(n_i.lid + 1)\%num\_streams + 1$ as a result of the round-robin. For example, n_4 is assigned to stream s_2 (i.e., $(2+1)\%2+1$). Assigning nodes in a round-robin manner at each level facilitates load balancing because nodes are evenly distributed across streams. The motivation of levelization is to implicitly capture dependencies between levels using the same stream. For instance, the dependency between n_1 and n_3 is implicitly captured by s_1.

We iterate each node level by level to perform three steps: *construct dependencies*, *assign stream*, and *decide an event*. At the first level, since n_1 does not have predecessors, we assign it to s_1 (Fig. 3(a)). We then check if any of n_1's successors (n_3, n_4, n_5) will be assigned to the different stream, s_2. Since n_4 will be assigned to s_2, we need to create an event for n_1 so that the later iteration can wait on it to create a dependency edge (Fig. 3(b)). At the second level, since n_3 is assigned to the same stream as its predecessor, n_1, we do not create a dependency from n_3 to n_1; but, we create an event for n_3 because its successor n_7 will be assigned to s_2, as shown in Fig. 3(c). Figure 3(d) and (e) show the process of n_4. Since n_4 is assigned to the different stream from it's predecessor,

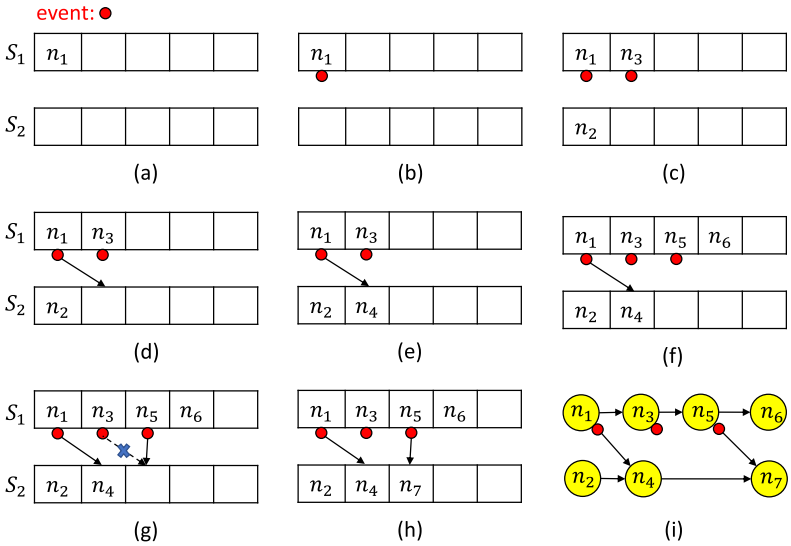


Fig. 3. Illustration of our algorithm on Fig. 2 using two streams.

n_1, we need to create a dependency before assigning n_4 to s_2 by waiting on n_1's event. The same procedure continues until we iterate all nodes. Our dependency pruning happens at assigning n_7 to s_2 (Fig. 3(g) and (h)). n_7's predecessors, n_3 and n_5, are both assigned to s_1. We only construct a dependency from n_5 to n_7 since n_5 is guaranteed to be executed after n_3 in the same stream, s_1. This pruning reduces redundant dependencies. The transformed CUDA graph from this assignment is shown in Fig. 3(i).

Algorithm 1 presents the details of our algorithm. We iterate all nodes at each level to perform the three tasks: *construct dependencies*, *assign stream*, and *decide an event*. For simplicity, $n.idx$ represents n_i's index, i.

Construct Dependencies (Lines 6–19): We construct dependencies from n's predecessor, *pred*, to n. Since n's predecessors may be assigned to the same stream that implicitly capturers sequential order of enqueued nodes, we only need to construct a dependency from the last assigned predecessor, *last_assign*, to n and prune the other dependencies starting from n's predecessors that is assigned to the same stream.

Assign Stream (Line 20) & Decide an Event (Lines 21–30): We assign n to s_{sid}, where *sid* is the id of the stream assigned to n. We decide an event by checking whether n is assigned to a different stream from one of its successors, *suc*. If true, we create and record an event for n so that *suc* can construct a dependency from n to *suc* at the later iteration. We further assign *suc.sm* to s_{sid} for dependency pruning that happened in the later *construct dependencies* stage.

Algorithm 1: Round Robin with Dependency Pruning.

```
   Input: num_streams: number of streams
   Input: graph: task graph defined by users
   /* create streams...                                              */
 1 levelized ← levelize(graph)
 2 for each_level_graph in levelized do
 3     for n in each_level_graph do
 4         sid ← (n.lid + 1)%num_streams + 1
 5         last_assign ← null
 6         for pred in n.predecessors do
 7             psid ← (pred.lid + 1)%num_streams + 1
 8             if s_psid == n.sm then
 9                 if last_assign == null or last_assign.idx < pred.idx then
10                     last_assign = pred
11                 end
12             end
13             else if s_psid != s_sid then
14                 cudaStreamWaitEvent(s_sid, pred.event)
15             end
16         end
17         if last_assign != null then
18             cudaStreamWaitEvent(s_sid, last_assign.event)
19         end
20         n.assign(s_sid)
21         for suc in n.successors do
22             ssid = (suc.lid + 1)%num_streams + 1
23             if s_ssid != s_sid then
24                 if n.event == null then
25                     cudaCreateEvent(n.event)
26                     cudaEventRecord(n.event, s_sid)
27                 end
28                 suc.sm ← s_sid
29             end
30         end
31     end
32 end
```

4 Experimental Results

We evaluate the performance of cudaFlow and cudaFlowCapturer on (1) five micro-benchmarks[1] that are representative for many GPU algorithm patterns, and (2) a large-scale machine learning workload directly derived from the 2020 champion of the HPEC Sparse Deep Neural Network (DNN) Inference Challenge [24]. Both cudaFlow and cudaFlowCapturer have different use cases that

[1] Source code: https://github.com/dian-lun-lin/cudaFlow-benchmarks.

complement each other. The purpose of our experiment is not to demonstrate which one outperforms another but to highlight that our transformation algorithm can achieve comparable performance (or even better) to the optimally-constructed CUDA graph when explicit graph construction is not possible. By default, we transform a cudaFlow into a CUDA graph of the same topology because all kernel execution parameters are known up-front. In cudaFlowCapturer, we use RR1, RR2, RR4, and RR8 to represent our algorithm using 1, 2, 4, and 8 streams in the round-robin loop, respectively. To demonstrate the effectiveness of our dependency pruning, $RR4^-$ and $RR8^-$ represent our algorithm without dependency pruning under 4 and 8 streams. We do not report $RR1^-$ and $RR2^-$ because redundant dependencies only occur between nodes that are assigned to different streams. Using one or two streams creates few redundant dependencies. All experiments ran on a Ubuntu Linux 5.0.0-21-generic x86 64-bit machine with 40 Intel Xeon Gold 6138 CPU cores at 2.00 GHz, one GeForce RTX 2080 Ti GPU with 11 GB memory, and 256 GB RAM. We compiled all programs using Nvidia CUDA nvcc 11.1 on a host compiler of GNU GCC-9.2.1 with C++17 standards and optimization flags `-O2` enabled. All data is an average of ten runs.

4.1 Micro-benchmarks

We consider five common GPU task graphs as our micro-benchmarks: linear chain (LC), embarrassing parallelism (EP), map-reduce (MR), divide and conquer (DC), and random DAG. LC task graph defines a sequence of sequentially dependent nodes. EP task graph defines only independent nodes. MR task graph defines several iterations each of 16 mappers and one reducer. DC task graph defines a complete binary tree. Random DAG defines a more generalized task graph; we randomly generate up to 50 nodes at each level and create at most five edges per node between successive levels. For all benchmarks, each node contains three sequential GPU operations: host-to-device (H2D) copy, reduction kernel, and device-to-host (D2H) copy. H2D operation first copies 2^{20} integers from CPU to GPU, the reduction kernel performs parallel sum reduction on all elements, and D2H operation copies the reduced sum from GPU to CPU. We focus on large GPU work where the effect of task graph parallelism is significant.

Performance Comparison. Table 1 compares the native CUDA graph size (#nodes+#edges) of each benchmark among cudaFlow and cudaFlowCapturer of different stream counts. Apparently, all methods have the same CUDA graph size in the LC task graph. cudaFlowCapturer has a larger CUDA graph size than cudaFlow in the EP task graph, since our algorithm assigns independent nodes to streams that implicitly capture the sequential execution order of enqueued nodes. The same situation happens in the DC task graph, where the number of independent nodes grows exponentially over levels. The CUDA graph size of DC, MR, and random DAG task graphs using cudaFlowCapturer become larger as we increase the number of streams. In our algorithm, more streams can have higher concurrency. However, it may result in more events to implicitly capture

Table 1. Comparison of CUDA graph sizes (#nodes+#edges) on linear chain, embarrassing parallelism, divide and conquer, map-reduce, and random DAG task graphs between cudaFlow and cudaFlowCapturer under different stream numbers 1 (RR1), 2 (RR2), 4 (RR4), and 8 (RR8). RR4$^-$ and RR8$^-$ represent our algorithm without the dependency pruning.

Task graph	cudaFlow	cudaFlowCapturer					
		RR1	RR2	RR4	RR4$^-$	RR8	RR8$^-$
Linear chain (65536 nodes)	393215	393215	393215	393215	393215	393215	393215
Embarrassing parallelism (65536 nodes)	327680	393215	393214	393212	393212	393208	393208
Divide and conquer (16 levels)	393209	393209	425975	442356	442356	450543	450543
Map-reduce (1024 iterations)	119813	104453	113668	125954	129026	132094	133118
Random DAG (512 levels)	103316	77893	86981	99217	104084	107822	112182
Random DAG (1024 levels)	207552	155437	169574	201875	214088	217454	226009
Random DAG (2048 levels)	410639	311453	347290	403291	423119	437859	447629
Random DAG (4096 levels)	832298	628715	693860	808276	857334	859342	892507

Table 2. Comparison of the number of streams issued by the CUDA runtime to run each task graph between cudaFlow and cudaFlowCapturer.

Task graph	cudaFlow	cudaFlowCapturer			
		RR1	RR2	RR4	RR8
Linear chain (65536 nodes)	12	12	13	15	19
Embarrassing parallelism (65536 nodes)	65547	12	14	18	26
Divide and conquer (16 levels)	32779	12	14	18	26
Map-reduce (1024 iterations)	15372	12	14	18	26
Random DAG (512 levels)	5318	12	80	244	559
Random DAG (1024 levels)	10547	12	150	440	1106
Random DAG (2048 levels)	21116	12	263	888	2213
Random DAG (4096 levels)	42545	12	522	1757	4348

the dependencies of the original task graph. Our dependency pruning shows a significant effect on reducing the CUDA graph size in MR and random DAG task graphs. For example, the CUDA graph size on Random DAG with 4096 levels using RR8 is 5.7% smaller than RR8$^-$. This is because MR and random DAG task graphs contain nodes that have more dependencies than others.

Table 2 compares the number of streams issued by the CUDA runtime to run each task graph. cudaFlow consumes much larger numbers of streams than cudaFlowCapturer on all benchmarks except the LC graph. By default, cudaFlow keeps a one-to-one mapping between the task graph and the CUDA graph. The CUDA runtime will issue as many streams as possible to maximize the task concurrency, whereas cudaFlowCapturer transforms the task graph into CUDA graph with a limited number of streams.

Figure 4 shows the execution time (including CUDA graph construction time) of each benchmark. Since LC task graph contains only sequential nodes, all

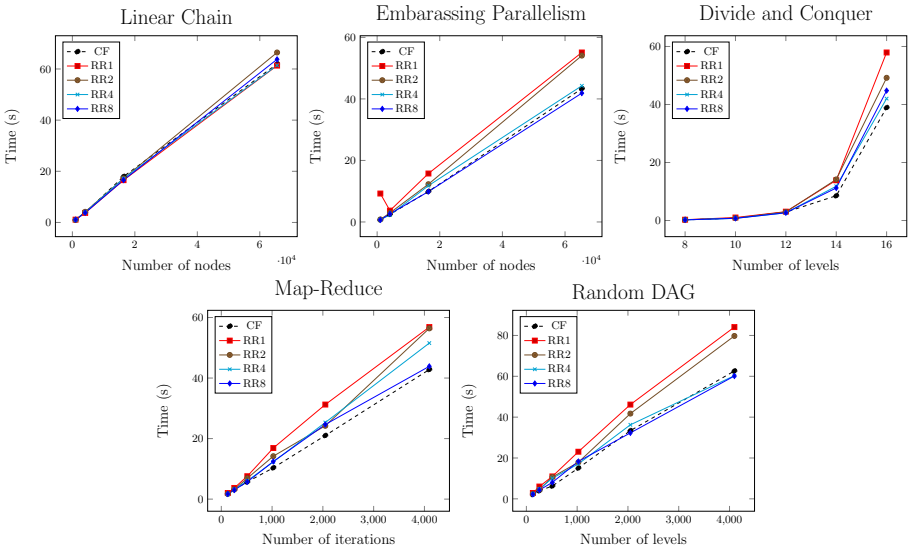


Fig. 4. Execution time of each task graph at different task graph sizes running on cudaFlow and cudaFlowCapturer of RR1, RR2, RR4, and RR8.

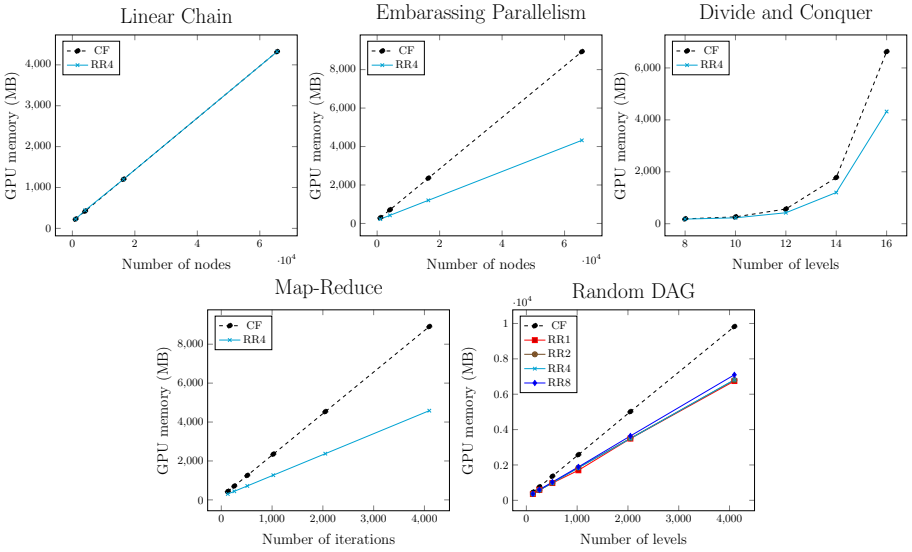
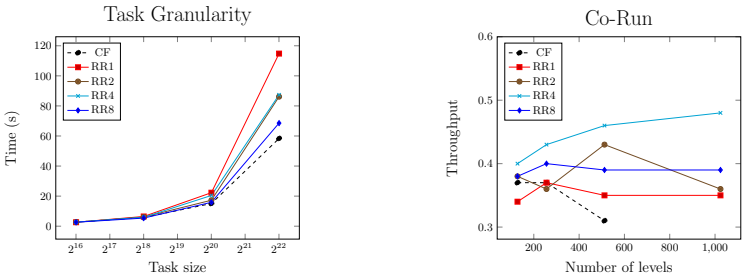


Fig. 5. Comparison of peak GPU memory usage of each task graph at different task graph sizes between cudaFlow and cudaFlowCapturer.

methods have almost the same execution time. RR4, RR8, and cudaFlow are faster than RR1 and RR2 in all other task graphs, because more streams have higher concurrency that leads to faster execution time. Figure 5 compares the



(a) Execution time of random DAG with 1024 levels at different task sizes.

(b) Throughput of corunning random DAG at different task graph sizes.

Fig. 6. (a) Task granularity and (b) co-run of random DAG running on cudaFlow and cudaFlowCapturer.

peak GPU memory usage of each benchmark at different task graph size running on cudaFlow and cudaFlowCapturer. We only compare cudaFlow with RR4 in LC, EP, DC, and MR task graphs since RR1, RR2, RR4, and RR8 have almost the same GPU memory usage in these task graphs. The GPU memory usage of cudaFlow is much higher than cudaFlowCapturer on all benchmarks except the LC task graph. In EP task graph, cudaFlow consumes 2.1× more GPU memory than cudaFlowCapturer. This is because the CUDA runtime does not limit the number of streams to run CUDA graphs. Figure 6(a) compares the execution time under different task sizes. Task size is the number of elements computed at each node. cudaFlow and RR8 become faster than the others when the task size grows. Compared to lightweight tasks with the same stream count, heavy tasks benefit more from higher kernel concurrency.

Next, we study the throughput of co-running multiple GPU graphs. The motivation is to emulate a server-like environment where multiple client GPU programs run concurrently on the same machine. We consider four co-run processes each executing one random DAG with the same number of levels. The throughput is defined as the execution time of running one process over the execution time of running four processes concurrently [16]. A throughput of 1 implies that the co-run's throughput is the same as if the processes were run consecutively. Figure 6(b) compares the throughput of each method. RR4 produces the highest throughput than others, whereas cudaFlow runs out of GPU memory due to unlimited streams.

4.2 Machine Learning: Large Sparse Neural Network Inference

The second experiment compares the performance of our transformation algorithm with an optimally-constructed CUDA graph (i.e., cudaFlow) using a large-scale machine learning workload from the IEEE HPEC Graph Challenge 2020. The challenge is to inference extremely large sparse DNN models. We rearchitect the CUDA graph-based champion solution in [24] using cudaFlow and

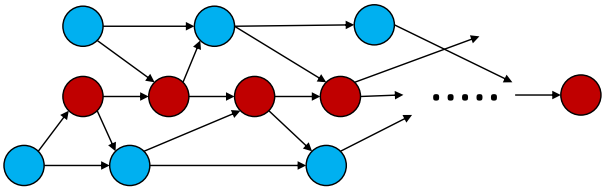


Fig. 7. [24] describes the inference workload in a task graph. A blue node represents a memory copy, and a red node represents a kernel. (Color figure online)

cudaFlowCapturer. We run the experiment on six DNN models composed of different neurons and layers. The statistics of each DNN and its modeled task graph size are summarized in Table 3. Figure 7 shows a partial task graph of the inference workload.

Table 3. The modeled task graph size (#nodes+#edges) and the statistics of each DNN benchmark (model size and image nonzeros).

Neurons/layers	120	480	1920	Model size	Image nonzeros
4096	599	2399	9599	5.40 GB	25,019,051
65536	599	2399	9599	94.70 GB	392,191,985

Table 4. Comparison of the execution time between cudaFlow and cudaFlowCapturer for completing six DNN models.

#Neurons	#Layers	cudaFlow	cudaFlowCapturer			
			RR1	RR2	RR4	RR8
4096	120	1.61	1.34	1.19	1.20	1.19
	480	4.70	4.74	4.19	4.19	4.20
	1920	17.41	19.14	17.08	17.14	17.15
65536	120	14.78	15.99	14.06	14.06	14.05
	480	43.00	50.59	42.92	42.81	42.90
	1920	162.20	193.11	162.12	162.35	162.30

Performance Comparison. Table 4 compares the execution time (in seconds) of each benchmark using cudaFlow and cudaFlowCapturer at different stream numbers. All methods except RR1 have similar execution time across all DNNs. We observe cudaFlowCapturer of two streams finishes the inference workload with comparable performance of cudaFlow. Using four or eight streams does not decrease the runtime. Table 5 compares the number of streams issued by the CUDA runtime. cudaFlow consumes a similar number of streams to cudaFlowCapturer. This is because the maximum degree of concurrency in this particular

Table 5. Comparison of number of streams issued by the CUDA runtime between cudaFlow and cudaFlowCapturer for completing six DNN models.

#Neurons	#Layers	cudaFlow	cudaFlowCapturer			
			RR1	RR2	RR4	RR8
4096	120	35	23	36	38	42
	480	35	23	36	38	42
	1920	35	23	36	38	42
65536	120	35	23	36	38	42
	480	35	23	36	38	42
	1920	35	23	36	38	42

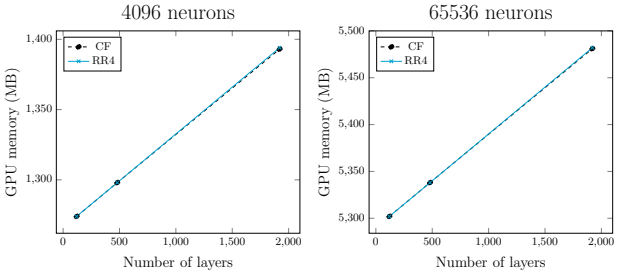


Fig. 8. Comparison of peak GPU memory usage at different number of layers between cudaFlow and cudaFlowCapturer (RR4).

task graph is around two, and the CUDA runtime will not consume too many streams to maximize the parallelism. Figure 8 compares the peak GPU memory usage at different numbers of layers. Both methods have almost the same peak GPU memory usage due to similar stream usage. This experiment demonstrates the efficiency of our transformation algorithm.

5 Related Work

[2,28] presents a compiler transformation method that translates OpenMP code into CUDA graphs. However, their transformation method only considers explicit graph construction. Our work offers users both explicit graph construction APIs (cudaFlow) and implicit graph construction APIs (cudaFlowCapturer) using our scheduling algorithm. [25] proposes a compiler-based approach that combines CUDA graph with an image processing DSL and a source-to-source compiler called Hipacc. Their kernel pipelining approach optimizes the schedule specifically for the scattering-pattern applications. [9] presents the Hybrid Task Graph Scheduler (HTGS) to aid in building hybrid workflows for high performance image processing. This architecture is different from our model that can handle and schedule arbitrary GPU task graphs.

Graph-based model is extensively studied on CPU-parallel architectures. Just name a few: Cpp-Taskflow [15,16,19,20,22,23] develops a simple and powerful task programming model enabling efficient implementations of heterogeneous decomposition strategies. PaRSEC [10] 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 [11] uses functional approaches to offer task graph constructions. It enables applications to achieve performance portability on diverse many-core architectures. Legion [8] describes a runtime system that dynamically extracts parallelism from Legion programs, using a distributed, parallel scheduling algorithm that identifies both independent tasks and nested parallelism. These models have their own pros and cons, but they do not target GPU graph parallelism.

Another line of related work to our transformation algorithm is the removal of redundant dependencies in DAGs. A common category is *transitive reduction* in graph theory. Alfred V. Aho et al. [6] propose algorithms for transitive reduction based on matrix multiplication. Other work [7,14,26,27,29,30] focuses on DAG traversal that processes each node separately. However, it is unknown how these algorithms can apply to our problem domain, in which the stateful property of CUDA stream constrains the order of dependency construction.

6 Conclusion

In this paper, we have introduced a lightweight task graph programming framework, cudaFlow and cudaFlowCapturer, to enable efficient GPU computation using CUDA graph in different scenarios. In five micro-benchmarks and a real machine learning workload, our transformation algorithm achieves comparable performance to the optimally-constructed CUDA graph and consumes much less GPU resource. The source of our programming model is available in [3].

7 Future Work

Future work includes applying reinforcement learning to find an optimal (near-optimal) scheduling solution and choose the optimal number of streams. An optimal scheduling solution varies due to not only different applications, but also different hardware specifications (e.g., GPUs) and different software (e.g., CUDA runtime). We plan to deploy a learning-based algorithm to learn from user environments and find optimal (near-optimal) scheduling solutions. Another line of future work is to extend our work to multiple GPUs. For example, we plan to introduce new stream management algorithms for multiple CUDA graphs that can run in parallel. On the application sides, we plan to use the proposed cudaFlow to solve large-scale simulation workloads in VLSI designs [12,13,17,18,21] and machine learning [24].

References

1. NVIDIA CUDA graph example. https://github.com/NVIDIA/cuda-samples/blob/master/Samples/simpleCudaGraphs/simpleCudaGraphs.cu
2. OpenMP. https://www.openmp.org
3. Taskflow. https://taskflow.github.io
4. Cuda graph in tensorflow. In: NVIDIA GPU Technology Conference (GTC) (2021). https://www.nvidia.com/en-us/on-demand/session/gtcspring21-s31312/
5. Effortless CUDA graphs. In: NVIDIA GPU Technology Conference (GTC) (2021). https://www.nvidia.com/en-us/on-demand/session/gtcspring21-s32082/
6. Aho, A.V., Garey, M.R., Ullman, J.D.: The transitive reduction of a directed graph. SIAM J. Comput. **1**(2), 131–137 (1972)
7. Augonnet, C., Thibault, S., Namyst, R., Wacrenier, P.A.: StarPU: a unified platform for task scheduling on heterogeneous multicore architectures. Concurr. Comput. Pract. Exp. **23**(2), 187–198 (2011)
8. Bauer, M., Treichler, S., Slaughter, E., Aiken, A.: Legion: expressing locality and independence with logical regions. In: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, SC 2012, pp. 1–11. IEEE (2012)
9. Blattner, T., Keyrouz, W., Bhattacharyya, S.S., Halem, M., Brady, M.: A hybrid task graph scheduler for high performance image processing workflows. J. Sig. Process. Syst. **89**(3), 457–467 (2017)
10. Bosilca, G., Bouteiller, A., Danalis, A., Herault, T., Lemariner, P., Dongarra, J.: DAGuE: a generic distributed DAG engine for high performance computing. In: 2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum, Anchorage, Alaska, USA, pp. 1151–1158. IEEE (2011)
11. Edwards, H.C., Trott, C.R., Sunderland, D.: Kokkos: enabling manycore performance portability through polymorphic memory access patterns. J. Parallel Distrib. Comput. **74**(12), 3202–3216 (2014). Domain-Specific Languages and High-Level Frameworks for High-Performance Computing. https://doi.org/10.1016/j.jpdc.2014.07.003. http://www.sciencedirect.com/science/article/pii/S0743731514001257
12. Guo, G., Huang, T.W., Lin, Y., Wong, M.: GPU-accelerated Pash-based timing analysis. In: ACM/IEEE Design Automation Conference (DAC) (2021)
13. 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)
14. Habib, M., Morvan, M., Rampon, J.X.: On the calculation of transitive reduction-closure of orders. Discret. Math. **111**(1–3), 289–303 (1993)
15. Huang, T.W.: A general-purpose parallel and heterogeneous task programming system for VLSI CAD. In: IEEE/ACM International Conference on Computer-aided Design (ICCAD) (2020)
16. 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)
17. Huang, T.W., Lin, C.X., Wong, M.D.F.: OpenTimer v2: a parallel incremental timing analysis engine. IEEE Trans. Comput. Aided Des. Integr. Circ. Syst. (TCAD) **40**(4), 776–789 (2021)
18. Huang, T.W., Lin, C.X., Wong, M.D.F.: OpenTimer v2: a parallel incremental timing analysis engine. IEEE Des. Test **38**(2), 62–68 (2021)

19. Huang, T.W., Lin, D.L., Lin, Y., Lin, C.X.: Taskflow: a general-purpose parallel and heterogeneous task programming system. IEEE Trans. Comput. Aided Des. Integr. Circ. Syst. (TCAD) (2021)
20. Huang, T.W., Lin, Y., Lin, C.X., Guo, G., Wong, M.D.F.: Cpp-Taskflow: a general-purpose parallel task programming system at scale. IEEE Trans. Comput. Aided Des. Integr. Circ. Syst. (TCAD) **40**, 1687–1700 (2021)
21. 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)
22. Lin, C.X., Huang, T.W., Guo, G., Wong, M.: An efficient and composable parallel task programming library. In: IEEE High Performance Extreme Computing (HPEC), pp. 1–7 (2019)
23. Lin, C.X., Huang, T.W., Guo, G., Wong, M.D.F.: A modern c++ parallel task programming library. In: ACM Multimedia Conference, pp. 2284–2287 (2019)
24. Lin, D.L., Huang, T.W.: A novel inference algorithm for large sparse neural network using task graph parallelism. In: IEEE High Performance Extreme Computing Conference (HPEC), pp. 1–7. IEEE (2020)
25. Qiao, B., Akif Özkan, M., Teich, J., Hannig, F.: The best of both worlds: combining CUDA graph with an image processing DSL. In: 2020 57th ACM/IEEE Design Automation Conference (DAC), pp. 1–6 (2020). https://doi.org/10.1109/DAC18072.2020.9218531
26. Simon, K.: An improved algorithm for transitive closure on acyclic digraphs. Theoret. Comput. Sci. **58**(1–3), 325–346 (1988)
27. Valdes, J., Tarjan, R.E., Lawler, E.L.: The recognition of series parallel digraphs. In: Proceedings of the 11th Annual ACM Symposium on Theory of Computing, pp. 1–12 (1979)
28. Yu, C., Royuela, S., Quiñones, E.: OpenMP to CUDA graphs: a compiler-based transformation to enhance the programmability of NVIDIA devices.. In: Proceedings of the 23th International Workshop on Software and Compilers for Embedded Systems, SCOPES 2020, New York, NY, USA, pp. 42–47. Association for Computing Machinery (2020). https://doi.org/10.1145/3378678.3391881
29. Zhou, J., Yu, J.X., Li, N., Wei, H., Chen, Z., Tang, X.: Accelerating reachability query processing based on DAG reduction. VLDB J. **27**(2), 271–296 (2018)
30. Zhou, J., Zhou, S., Yu, J.X., Wei, H., Chen, Z., Tang, X.: DAG reduction: fast answering reachability queries. In: Proceedings of the 2017 ACM International Conference on Management of Data, pp. 375–390 (2017)