Grape: Practical and Efficient Graph-based Executions for Dynamic Deep Neural Networks on GPUs

Bojian Zheng[1, 2, 3], Cody Hao Yu[4], Jie Wang[4], Yaoyao Ding[1, 2, 3], Yizhi Liu[4], Yida Wang[4], Gennady Pekhimenko[1, 2, 3]

[1]CentML [2]University of Toronto [3]Vector Institute [4]AWS

https://github.com/UofT-EcoSystem/Grape-MICRO56-Artifact

1. Background: CUDA Graphs

Ubiquitous CPU overheads in machine learning systems:

```
foo.py
C = A + B
```

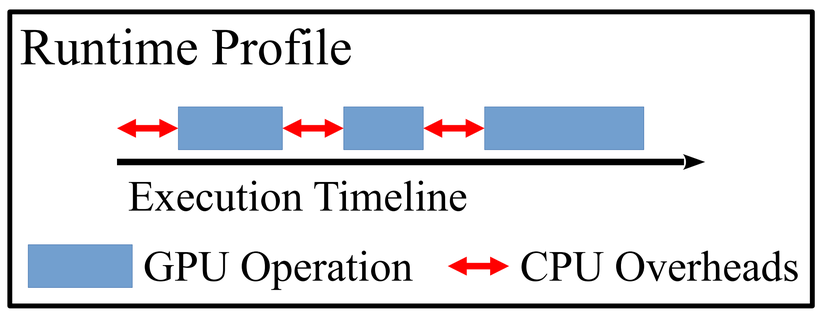
E.g.,
- Python invokes C APIs.
- Framework checks input shapes.
- CUDA launches compute kernels on GPUs.
- …

leads to

PyTorch

```
CHECK(A.shape == B.shape);
cudaAddKernel<<<…>>>(…);
```

GPU

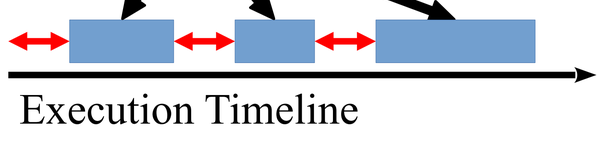


CUDA graphs remove CPU overheads by capturing effective GPU operations and then replaying them:

Capture

```
A = Tensor()
graph_ctx = CUDAGraph()
with graph_ctx:
  MyDNN(A)
```

Record

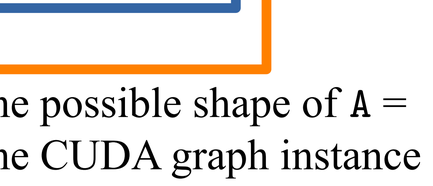


Execution Timeline

Replay

```
graph_ctx.replay()
```

Replay



Execution Timeline

No CPU overheads!

2. Challenges posed by CUDA Graphs

CUDA graphs request computations to be **frozen**.

Capture

```
ph_A = Tensor()
graph_ctx = CUDAGraph()
with graph_ctx:
  MyDNN(ph_A)
```

Placeholders are used at capture time.

One possible shape of `A` = One CUDA graph instance

\+ Every instance consumes GPU memory.

`MyDNN` cannot have data-dependent control flows (e.g., `if`, `break`, and `continue` statements).

Replay

```
A = Tensor()
ph_A.copyFrom(A)
graph_ctx.replay()
```

Placeholders are populated with real input values at runtime.

❶ Extra data movements into placeholders incur runtime overheads (up to 13%).

❷ Huge GPU memory consumption to efficiently support dynamic-shape workloads (20-100 GB).

❸ No support for data-dependent control flows.

3. Grape's Key Ideas

① Alias Prediction

```
foo.py
42 A = Tensor()
```

☞ `ph_A.copyFrom(A)`

`A`, generated at `foo.py+42`, is an alias of `ph_A`.

Next Iteration

```
foo.py
42 A = Tensor()
```

Predict `A` to be an alias, and directly use `ph_A`'s memory region for `A`.

~~`ph_A.copyFrom(A)`~~

② Metadata Compression

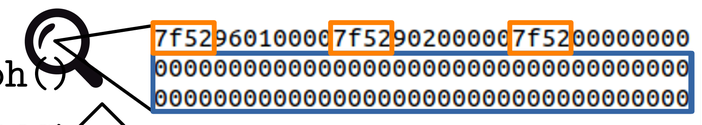
```
graph_ctx = CUDAGraph()
```

describes

```
__global__ void cudaKernel(
  const float* input,
  float* output
)
```

Pointer values

Underutilize the reserved function argument spaces (4 KB in CUDA).



High **sparsity** and **value redundancy** ⟹ Up to **36.4**× compression ratio.

③ Predication Contexts

```
with Predicate(x):
```

GPU operations within are **nullified** if `x` is `False`.

```
if x:
  …
else:
  …
```

≡

```
with Predicate(x):
  …
with Predicate(!x):
  …
```

```
for …:
  if x:
    continue
  …
```

≡

```
for …:
  with Predicate(!x):
    …
```

```
for …:
  if x:
    break
  …
```

≡

```
flag = CUDATensor(False)
for …:
  with Predicate(!flag):
    flag = x
  with Predicate(!flag):
    …
```

CUDA graph-optimizable ⟹ **1.79**× speedup in GPT-2 beam search module.

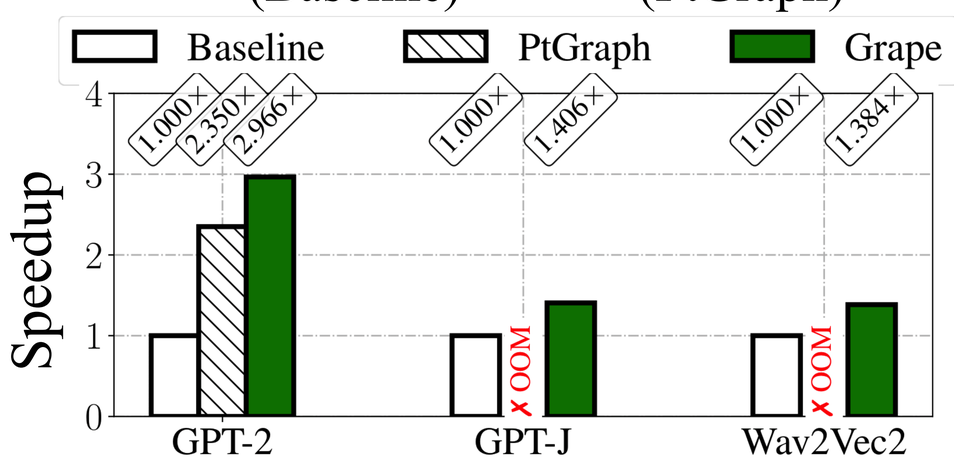
4. Evaluation

Infrastructure: NVIDIA RTX 3090 and A100 with PyTorch ver. 1.12 and NVIDIA CUDA ver. 12.0.

Applications: GPT-2[137M], GPT-J[6B], and Wav2Vec2[Base] from 🤗 Transformers.

Compared with PyTorch and CUDA Graphs + PyTorch.

(Baseline) (PtGraph)



- On GPT-2, **2.97**×/**1.26**× better than Baseline/PtGraph.
- On GPT-J and Wav2Vec2, up to **1.41**× better than Baseline while PtGraph goes OOM.

MICRO 2023 56th IEEE/ACM International Symposium on Microarchitecture®