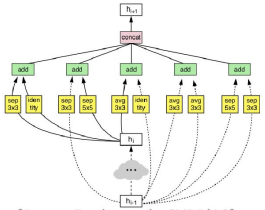
Rammer: Enabling Holistic Deep Learning Compiler Optimizations with rTasks

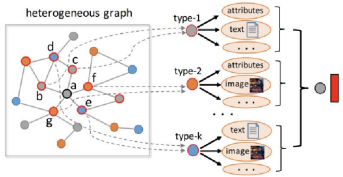
OSDI 2020

DL Frameworks Bridge the Gap of Models and Hardware

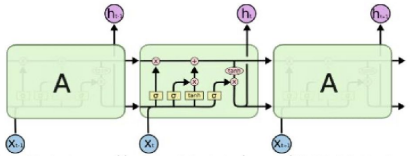
DNN Models



[Barret Zoph, et.al., CVPR'18]



[Chuxu Zhang, et.al., KDD'19]



[Christopher Olah, https://colah.github.io/posts/2015-08-Understanding-LSTMs/]

julia mxnet K DEEP LEARNING theano Microsoft CNTK Caffe2 Chainer

Modern Accelerators

NVIDIA GPU

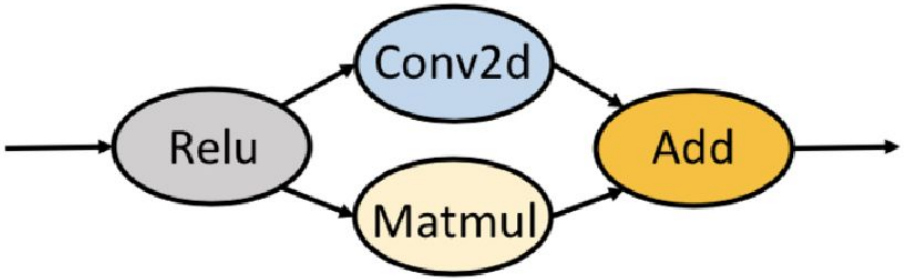
AMD GPU

TPU

Graphcore IPU

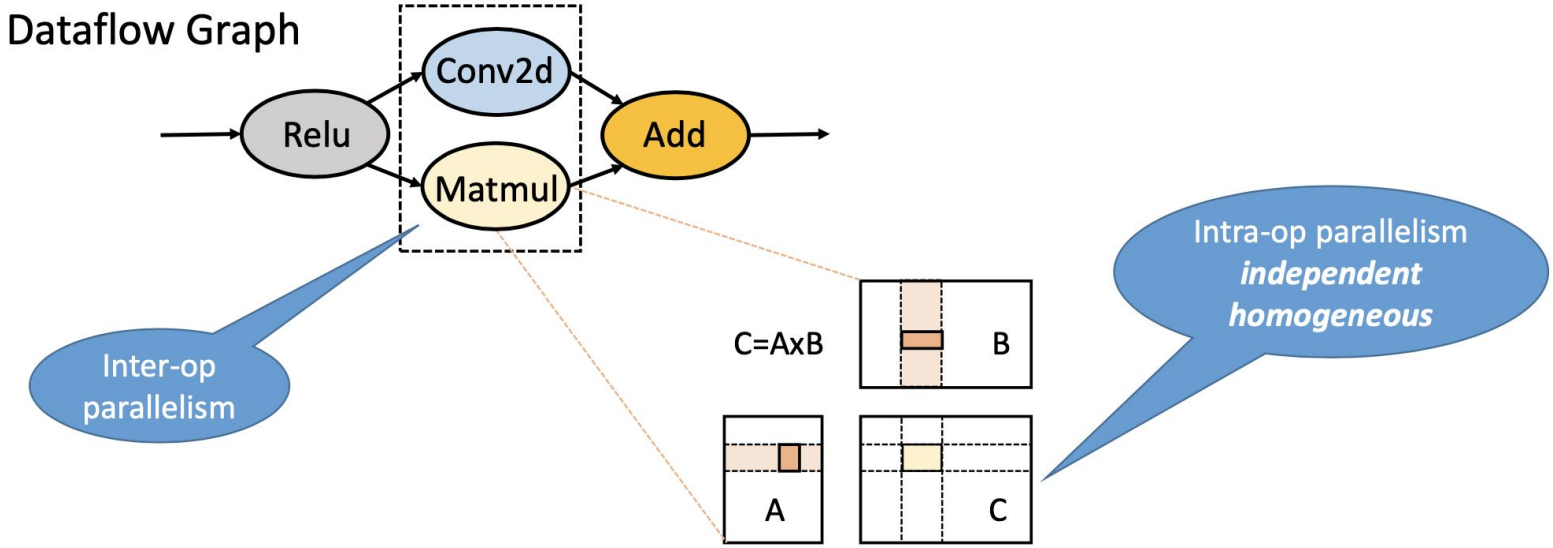
Existing DL frameworks

- DNN is usually modeled as a dataflow graph (DFG)
  - DFG is a graph which represents a data dependancies between a number of operations



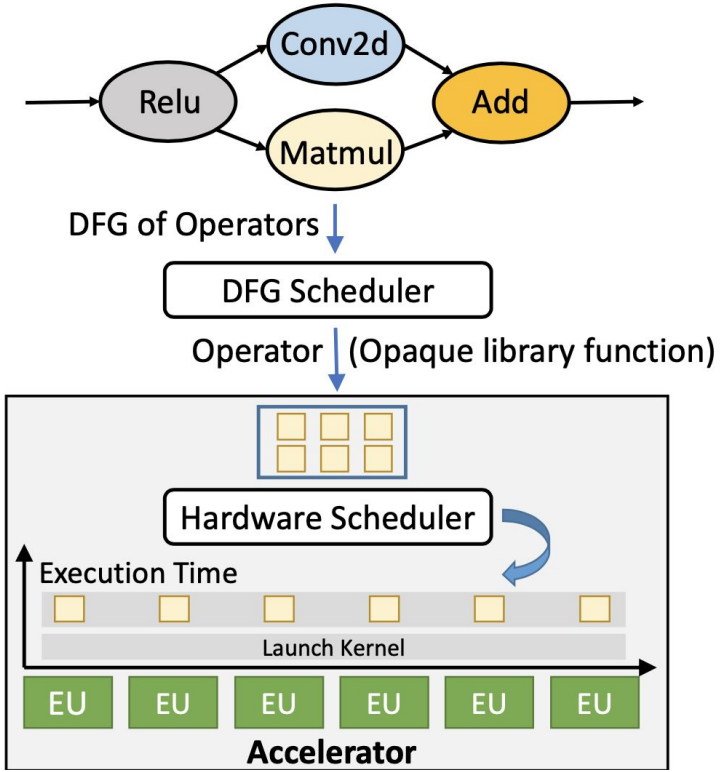
Existing DL frameworks

- DNN is usually modeled as a dataflow graph (DFG)
- DFG naturally contains two levels of parallelism



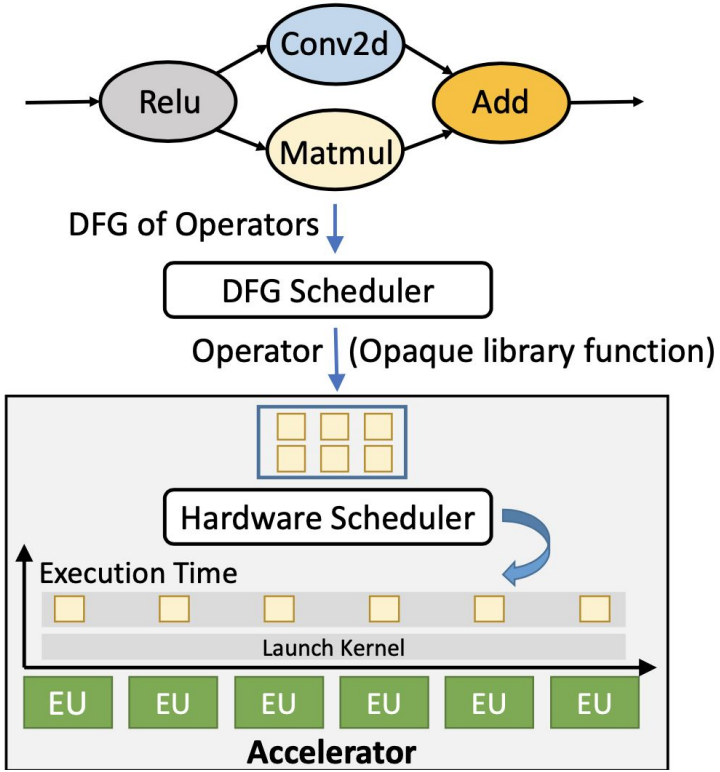
Existing Approach: Two-Layer Architecture

- ***DFG scheduler*** exploits inter-op parallelism
  - Emit operators that are ready for execution
  - Operators are treated as **opaque library functions**
- ***Hardware scheduler*** exploits intra-op parallelism
  - Map intra-op computation to parallel execution units (EUs)



Limitations of Existing Two-Layer Architecture

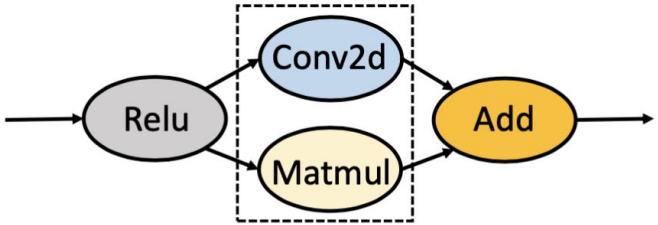
- Two-layer architecture works well when:
  - Schedule overhead is negligible
  - Intra-op parallelism can saturate all EUs
- However, this is often not the case in practice
  - Accelerators are becoming more and more powerful
    - P100 (9.3 Tflops)-> RTX 3090 (35.6 Tflops)
- Low GPU utilization
  - 2% ~ 62% utilization
- High operator scheduling overheads
  - 38% ~ 65% non-kernel time

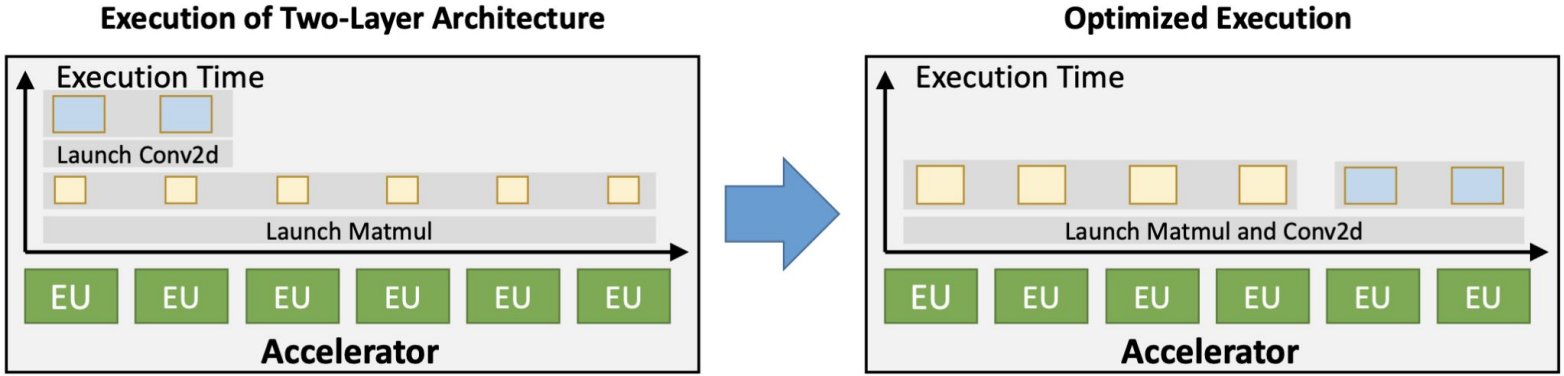


All data are reported on the inference task (BS=1) of 6 models on a V100 GPU

Limitations of Existing Two-Layer Architecture

- Overlook the subtle interplay of inter- and intra- op parallelism





Their proposal: Rammer

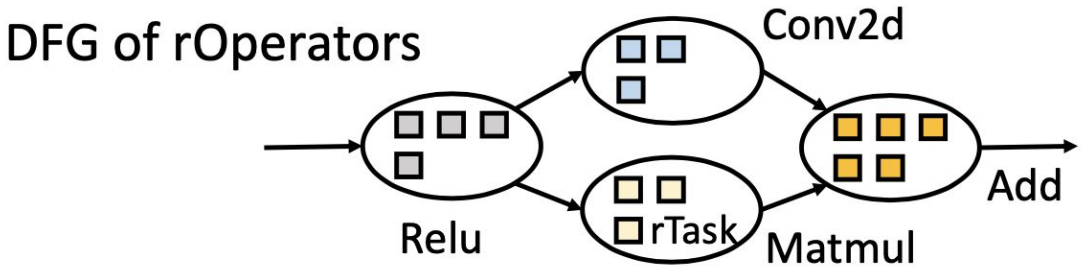
- A DNN compiler that optimizes the execution of DNN workloads on massively parallel accelerators
- Key idea
  - holistically exploiting parallelism through inter- and intra- operator co-scheduling
    - No longer low GPU utilization!
  - generates a static spatio-temporal schedule for a DNN at compile time
    - No longer high scheduling overhead!

Rammer design

Challenge 1: operators are opaque functions and do not expose fine-grained intra-op parallelism

Solution: rTask-Operator (rOperator) abstraction

- Expose fine-grained intra-op parallelism
- A group of *independent*, *homogeneous* rTasks
- rTask is the *minimum computation unit* on an EU

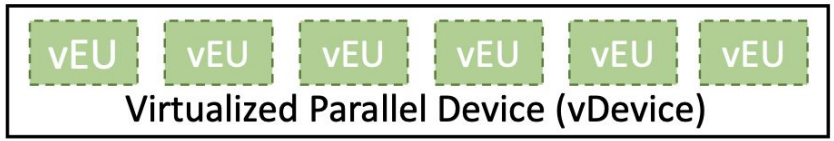


Rammer design

Challenge 2: accelerators (e.g., GPU) do not expose interfaces for intra-op scheduling

Solution: virtualized parallel device (vDevice) abstraction

- Expose hardwares' fine-grained scheduling capability
- Decouple scheduling from hardware devices
- Bypass the hardware scheduler

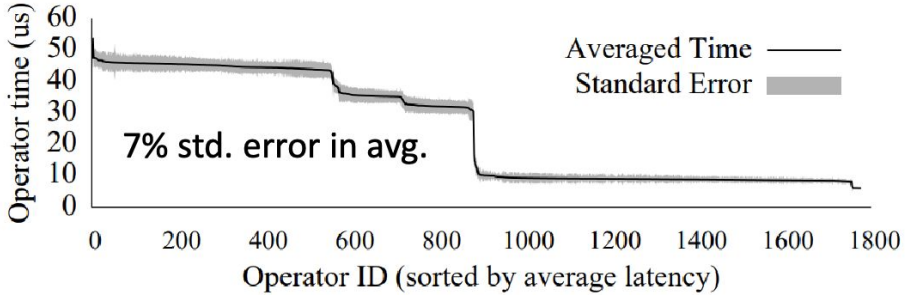


Rammer design

Challenge 3: fine-grained scheduling could incur even more scheduling overheads

Observation: predictability of DNN computation

- Most DNN's DFG is available at the *compile time*
- Operators exhibit *deterministic* performance



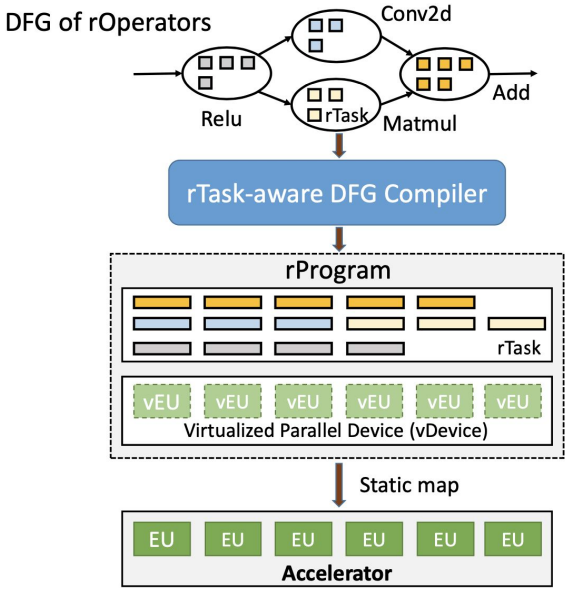
The profiled kernel time of all the operators in ResNeXt model. Each data point ran 1,000 times.

Rammer design

Challenge 3: fine-grained scheduling could incur even more scheduling overheads

Solution: generate execution plan (rProgram) at ***compile time***

- **Mechanism**: scheduling interfaces & profiler
- **Policy**: wavefront scheduling policy



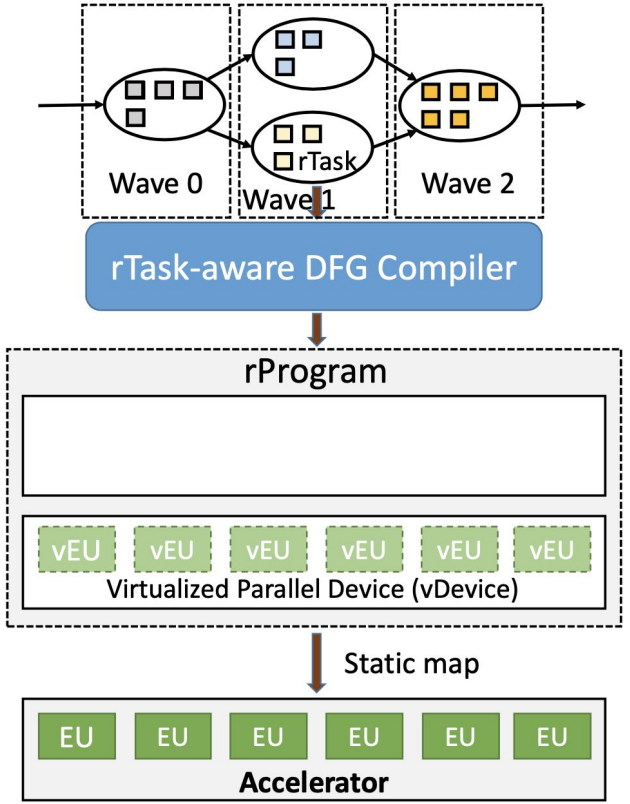
Wavefront scheduling policy

- Each rOperator has different kernel implementations

- Partition DFG into waves by BFS

- Select fastest kernels if current wave does not saturate all EUs

- Select resource-efficient kernels for inter-/intra-op interplay if current wave saturates all EUs



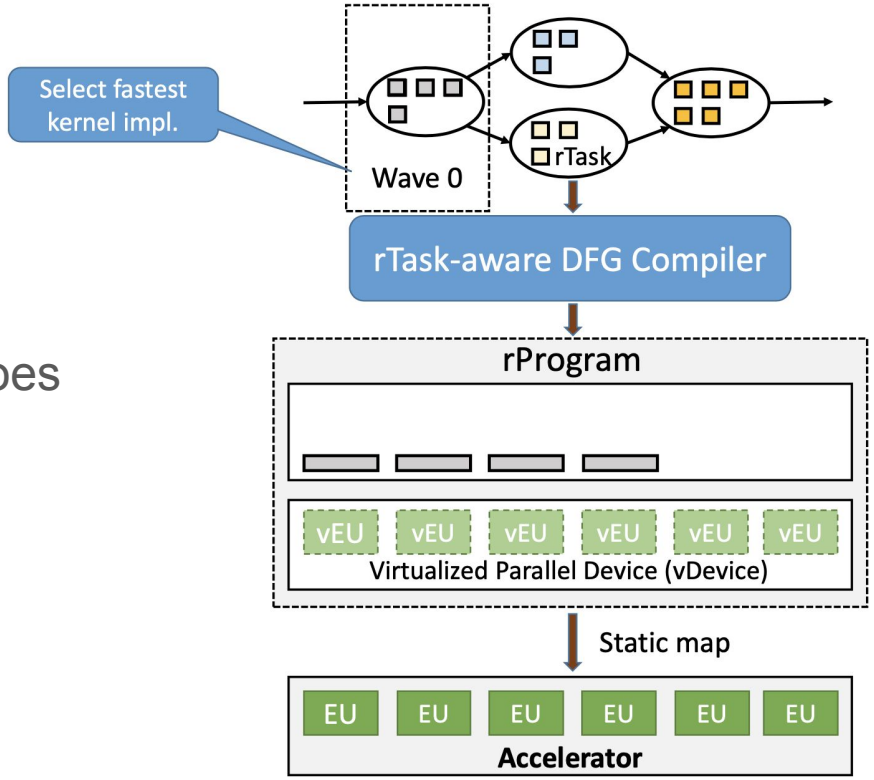
Wavefront scheduling policy

- Each rOperator has different kernel implementations

- Partition DFG into waves by BFS

- Select fastest kernels if current wave does not saturate all EUs

- Select resource-efficient kernels for inter-/intra-op interplay if current wave saturates all EUs



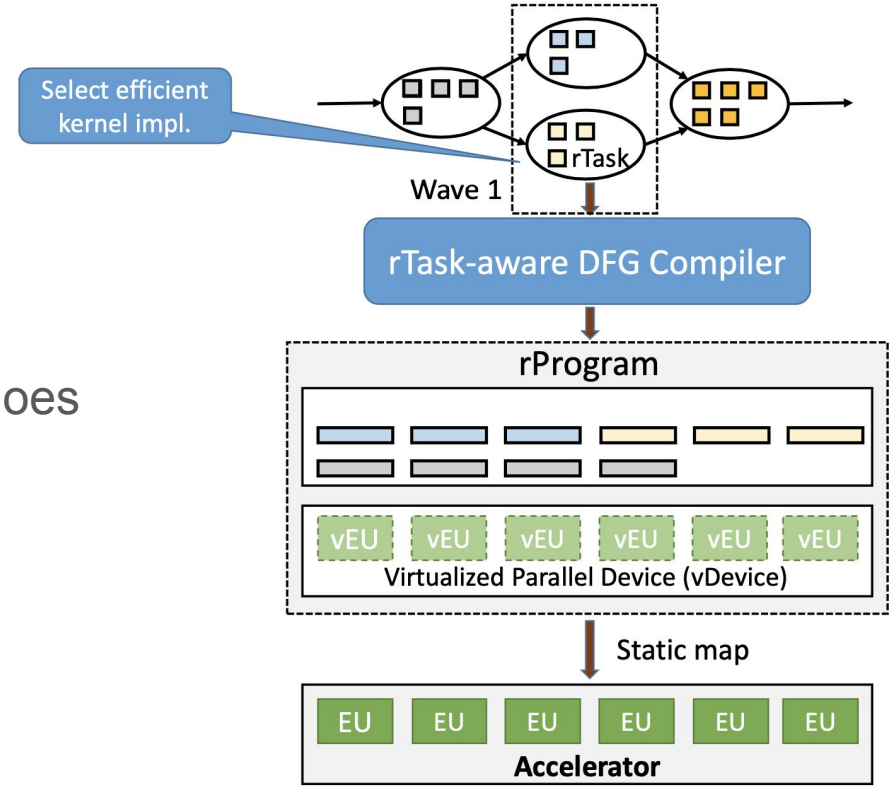
Wavefront scheduling policy

- Each rOperator has different kernel implementations

- Partition DFG into waves by BFS

- Select fastest kernels if current wave does not saturate all EUs

- Select resource-efficient kernels for inter-/intra-op interplay if current wave saturates all EUs



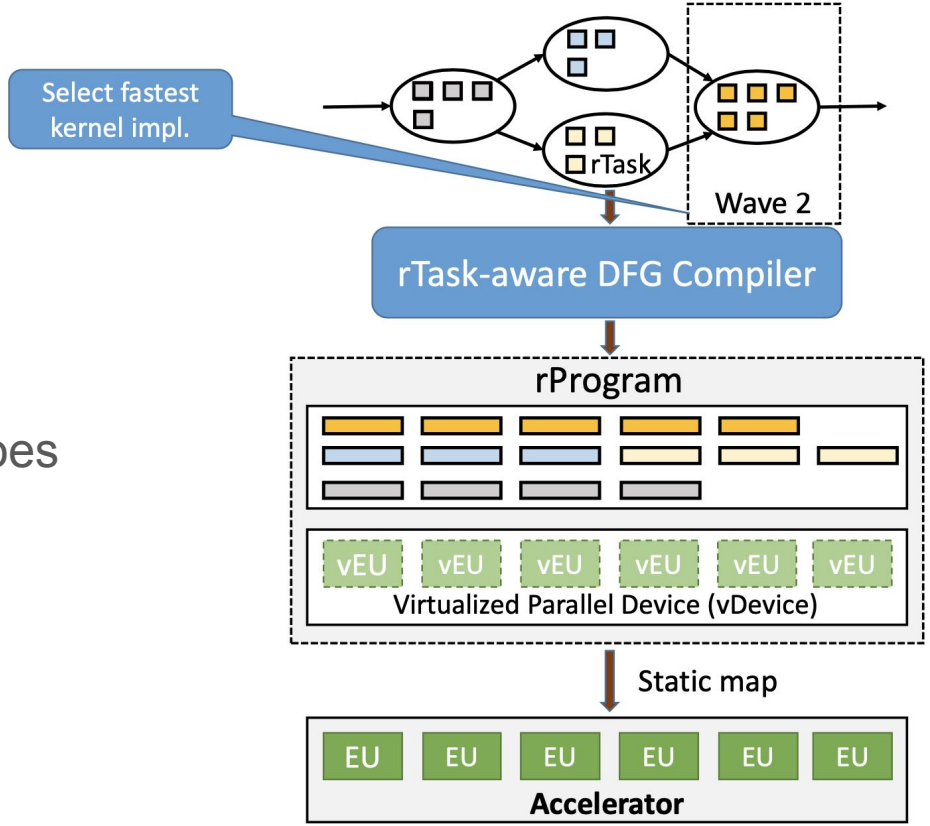
Wavefront scheduling policy

- Each rOperator has different kernel implementations

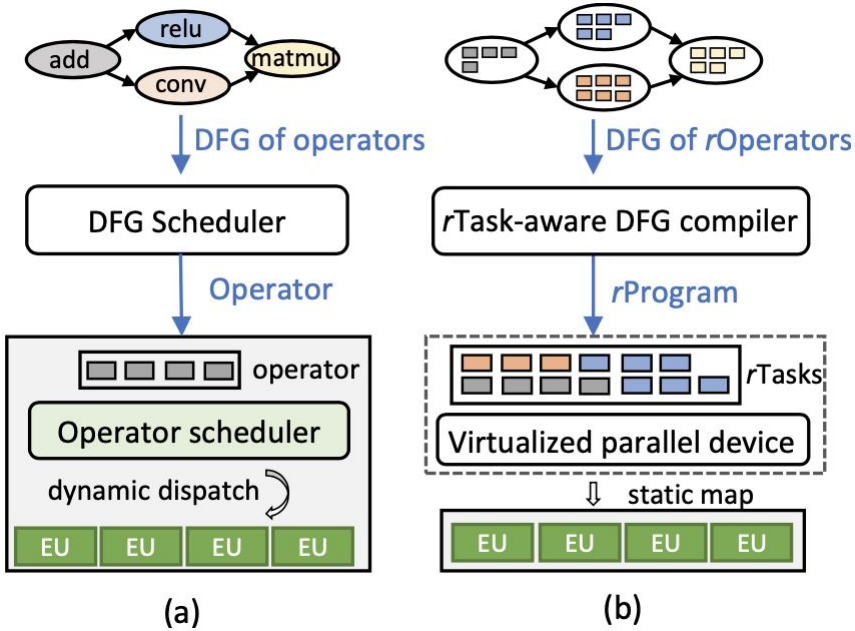
- Partition DFG into waves by BFS

- Select fastest kernels if current wave does not saturate all EUs

- Select resource-efficient kernels for inter-/intra-op interplay if current wave saturates all EUs

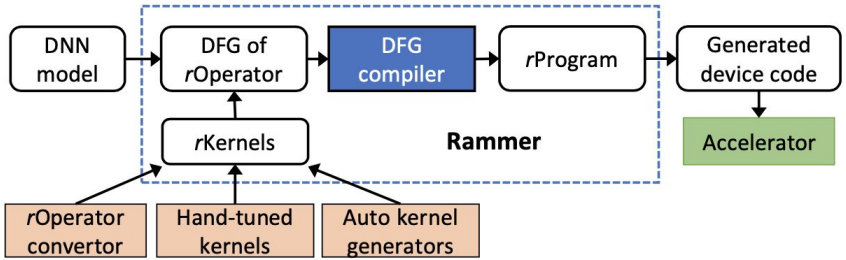


Recall: Rammer design



Existing DNN frameworks

RAMMER



Evaluation

Case Study: LSTM-TC-BS4

End-to-end Performance on CUDA GPU

GPU Utilization

Scheduling Overhead

End-to-end Performance on AMD GPU

End-to-end Performance on GraphCore IPU

Evaluation

Case Study: LSTM-TC-BS4

End-to-end Performance on CUDA GPU

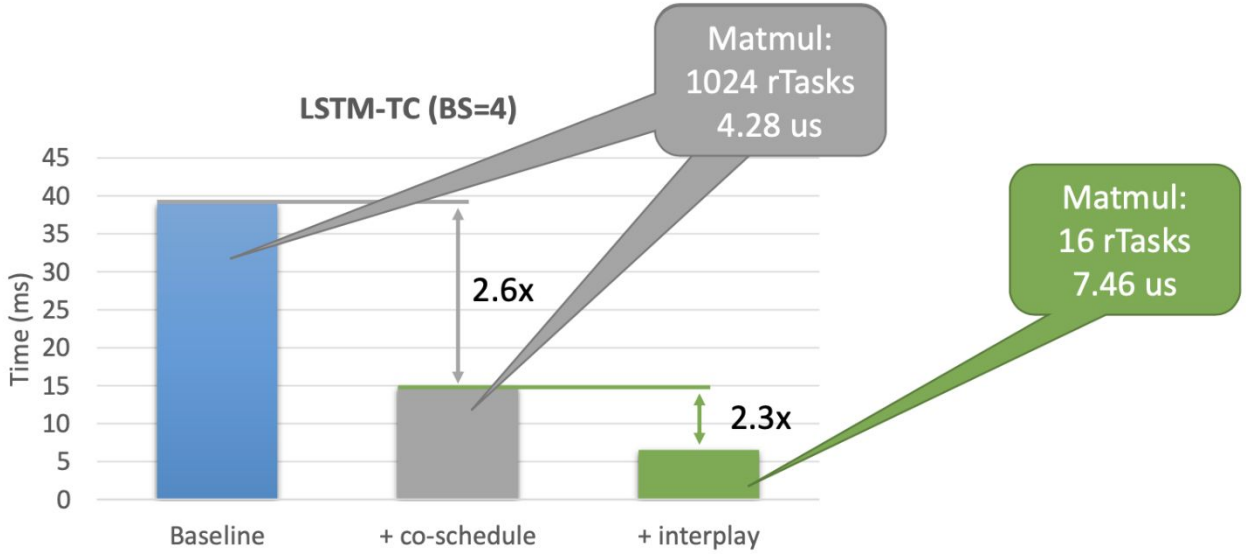
GPU Utilization

Scheduling Overhead

End-to-end Performance on AMD GPU

End-to-end Performance on GraphCore IPU

Case Study: LSTM-TC-BS4

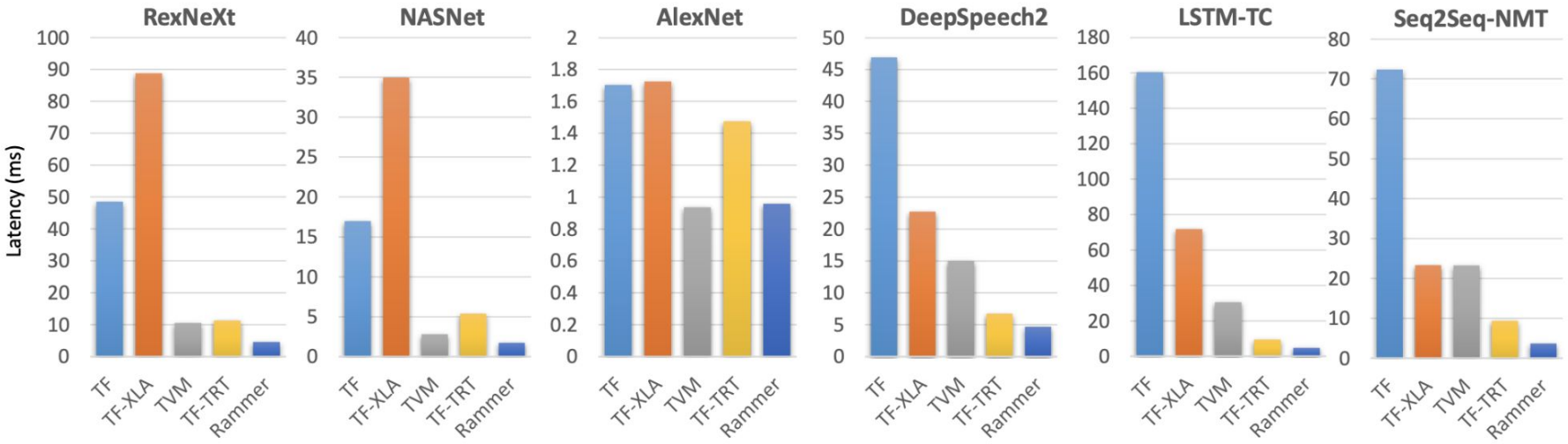


Baseline: two-layer architecture with compiler optimizations (e.g., kernel fusion, kernel tuning)

+ co-schedule (fastest kernels): operator co-scheduling on fastest kernels (same kernels as Baseline)

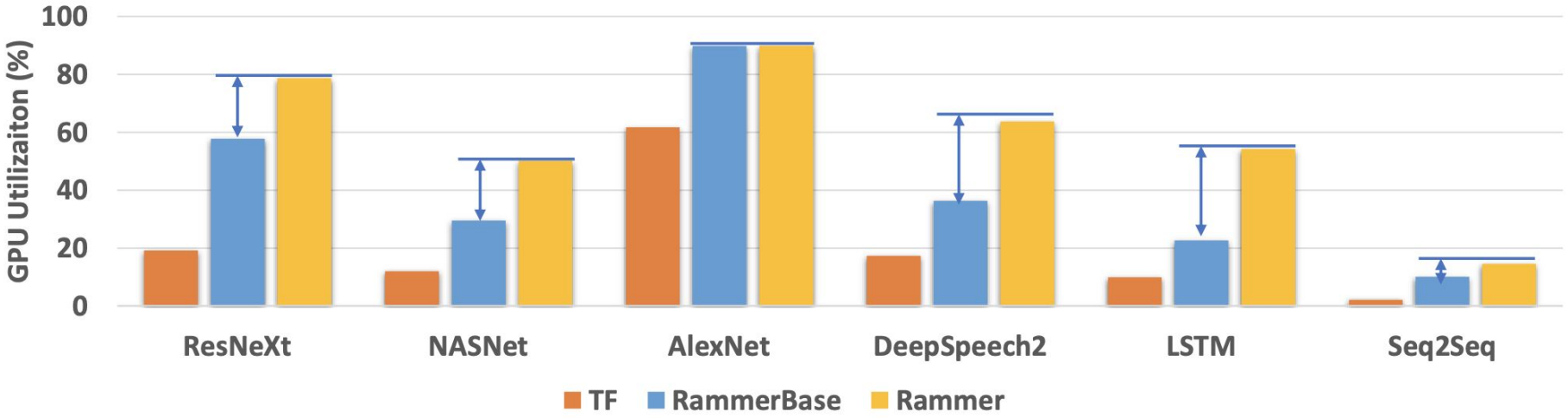
+ interplay: operator co-scheduling with interplay of inter-/intra- operator parallelism

End-to-end Performance on CUDA GPU



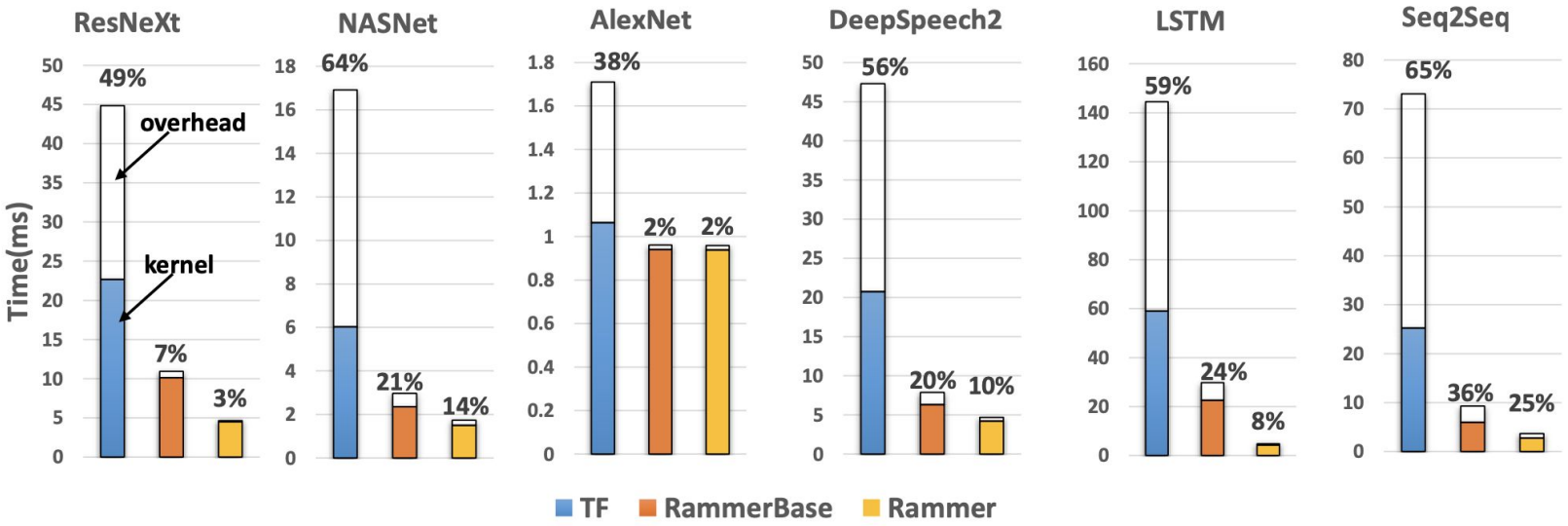
- up to 33.94x speedup over TensorFlow-1.15.2 (SOTA DL framework)
- up to 20.12x speedup over TensorFlow-XLA-1.15.2 (SOTA DL compiler)
- up to 6.46x speedup over TVM-0.7 (with AutoTVM) (SOTA DL compiler)
- up to 3.09x speedup over TensorRT-7.0 (SOTA vendor optimized proprietary library)

GPU Utilization



- The average GPU utilization of TensorFlow is only 20.3%
- Rammer can improve the average GPU utilization by 4.32x
- Compared to RammerBase, Rammer's scheduling by itself can improve the utilization by 1.61x

Scheduling Overhead



- RammerBase reduces avg overhead from 32.29 ms to 2.27 ms over TensorFlow
- Rammer can further reduce avg overhead to 0.37 ms over RammerBase

Conclusion

- Rammer: holistic approach to manage the parallelism in DNN for scheduling
- Hardware neutral solution
  - **rTask-Operator Abstraction**: expose fine-grained intra-operator parallelism
  - **Virtualized Parallel Device**: expose hardwares fine-grained scheduling capability

strengths:

- a holistic design to acheive really good performance improvement

weaknesses:

- DNN execution is determistic, can not support dynamic models

Reference

1. Rammer's OSDI presentation

Implemenation details

- Given that an rTask is logically identical to a parallel task, RAMMER relies on external tools to partition an rOperator into rTasks (e.g., TVM [23]).

```
1  interface Operator { void compute(); };
2  interface rOperator {
3    void compute_rtask(size_t rtask_id);
4    size_t get_total_rtask_num();
5  };
```

Figure 6: The execution interfaces of traditional operator and *r*Operator. More details in §4.

RAMMER profiler provides the following three types of information:
1) individual rTask execution time on a vEU;
2) resource usage of an rTask such as the local memory or registers used;
3) the overall execution time of an rProgram.

Several novel abstractions

- rTask
  - the minimum schedulable unit runs on a single execution unit of an accelerator (e.g., a streaming multiprocessor SM in a GPU)
- rTask-operator (rOperator)
  - An rOperator consists of multiple independent, homogeneous rTasks
  - Treat DNN as a data flow graph of rOperator nodes
- virtualized execution units (vEU)
- virtualized parallel device (vDevice)
  - Abstracts a hardware accelerator and contains multiple vEUs
  - Treat a hardware accelerator as a vDevice
  - Allows to run several rTasks on a specified vEU in a desired order
  - Maps a vEU to one of the physical execution units in an accelerator to perform the actual computation
- rKernel
  - The implementation of an rOperator
  - One rOperator might have multiple versions of rKernels based on different tiling strategies
- rPrograms
  - Two dimensional array of rTask: prog[vEU_id][order]
  - Assign rTasks to vEUs