# DeepCuts: A Deep Learning Optimization Framework for Versatile GPU Workloads

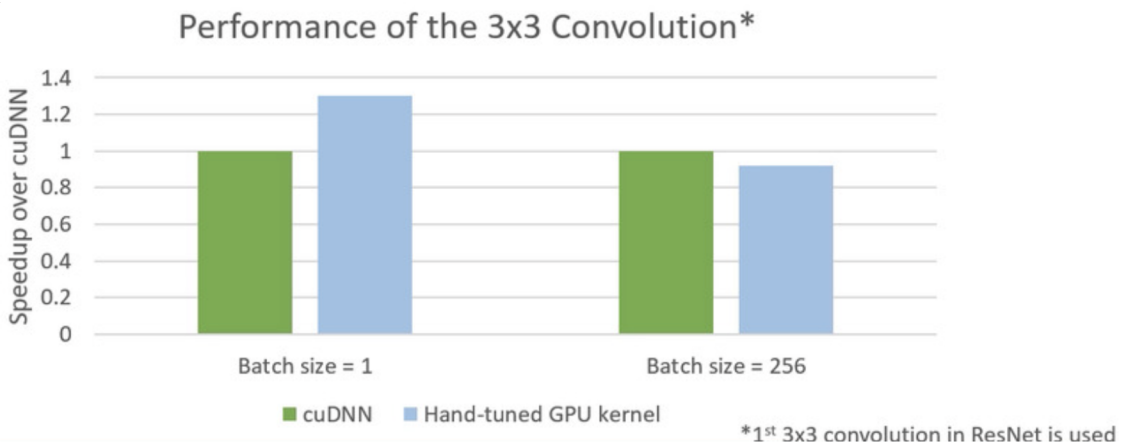
PLDI 2021

# Emergence of Deep Learning

- Deep learning (DL) becomes an essential technology for various applications
- GPUs are the de facto standard to run DL applications
- DL frameworks are widely used
  - E.g., PyTorch, Tensorflow, etc
  - built on top of primitive libraries provided by the GPU vender
  - map user-defined DL workloads to primitive library calls
  - NVIDIA cuDNN is a crucial component
    - Most of time-consuming DL operations are handled using cuDNN

# Limitation of DL Primitive Libraries

- Using cuDNN does not guarantee the best performance
  - cuDNN is mainly optimized for CNN inference and training with a large batch of inputs



# Limitation of DL Primitive Libraries

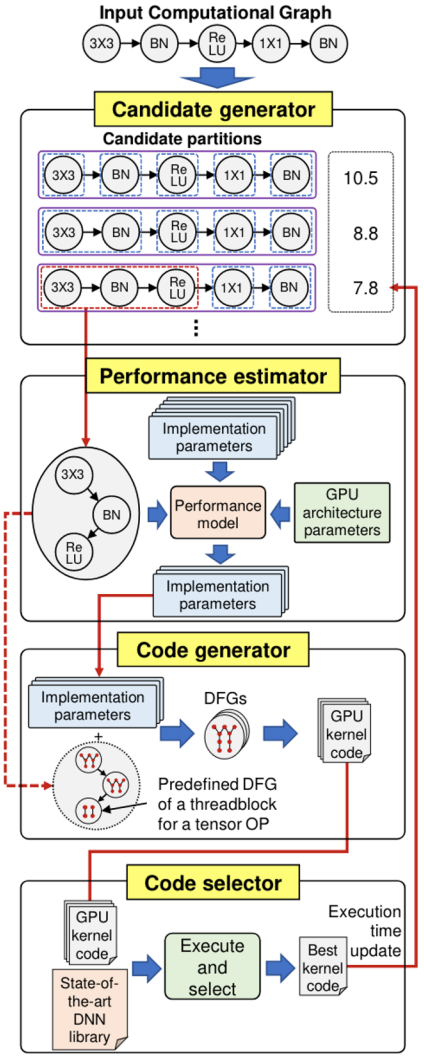
- Fixed, pre-implemented GPU kernels cannot handle the versatility of DL Workloads
  - Solution: Generate optimized GPU kernels for the given DL workloads
  - Several DL optimization frameworks have been proposed
    - TensorFlow XLA, TensorRT, TVM, Tensor Comprehensions
  - Two different categories
    - Frameworks that heavily rely on hand-tuned GPU kernels
    - Frameworks that use ML-based GPU kernel optimizer

# Limitation of DL Primitive Libraries

- Category 1: frameworks that heavily rely on hand-tuned GPU kernels
  - E.g., TensorFlow XLA and TensorRT
  - Problem 1: Low flexibility
    - TensorFlow XLA still heavily rely on cuDNN
      - when meets complex opertaions
    - TensorRT uses hand-tuned fused kernels
      - Shows good performance only for some specific cases
  - Problem 2: Closed-source
    - The detailed optimization techniques are hidden
- Category 2: frameworks that use ML-based GPU kernel optimizer
  - E.g., TVM and Tensor Comprehensions
  - Generate GPU kernels for the given DL workloads
    - Use ML-based performance estimation models
      - Do not directly use architectural information
  - Problem: poor performance
    - The performance is not competitive with cuDNN for many cases
      - E.g., large-batched CNN inferences

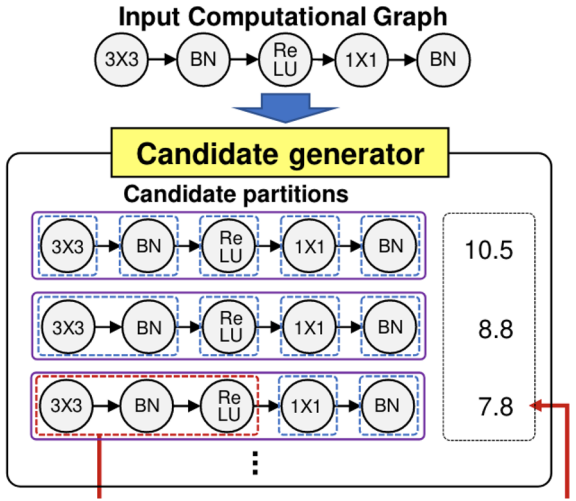
# DeepCuts Framework

- Performance model-based DL optimization framework
- Searches for the best-performing
  - Way of fusion
  - Parameters for the kernel generation
- Using
  - Architecture-aware performance estimation model
  - DFG-based flexible code generator
- Four components
  - candidate generator
  - performance estimator
  - code generator
  - code selector



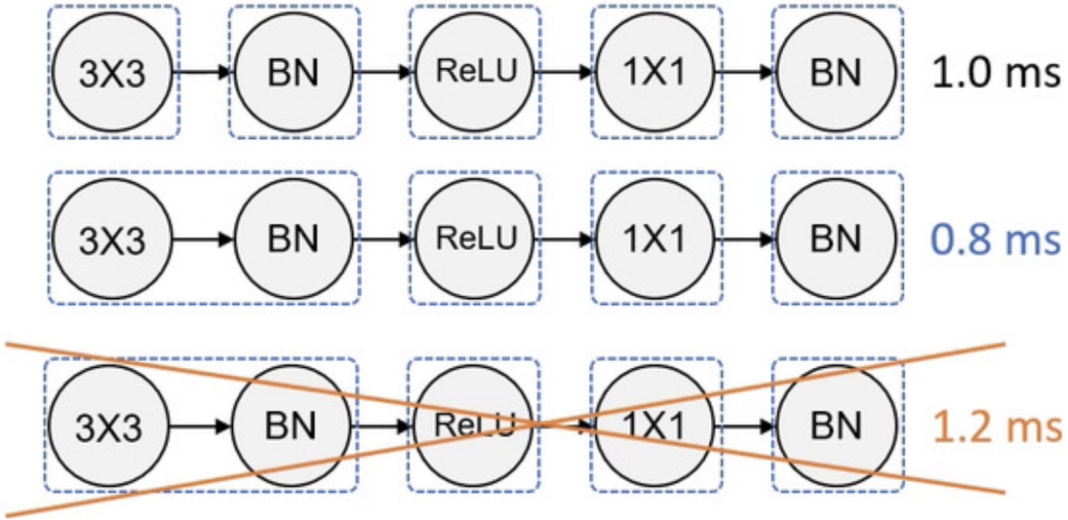
# Candidate Generator

- Generates candidate for the code generation
  - Partitions the set of tensor operations
  - Each partition becomes a candidate for the code generation
- Enumerates possible ways of kernel fusions



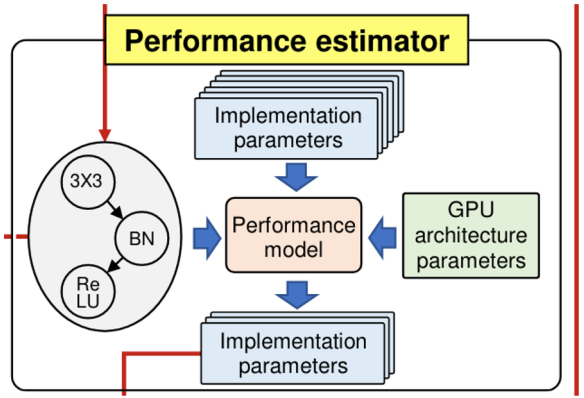
# Candidate Generator

- Problem: the search space is too huge
- Solution: greedy search
  - Try to fuse A + B + C only when fusing A + B or B + C gives speedup



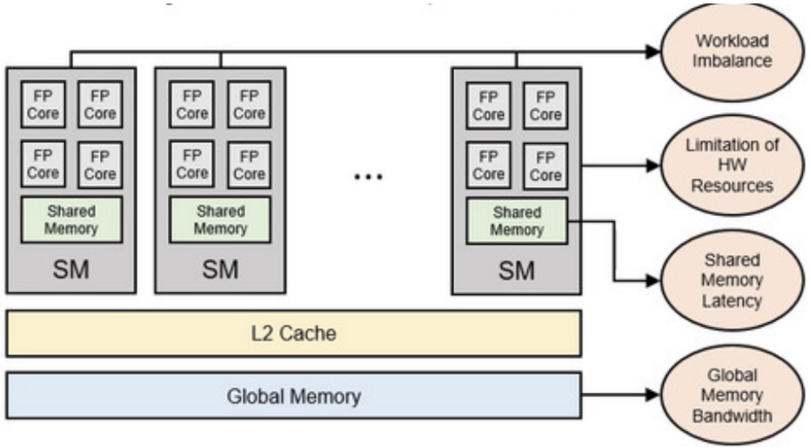
# Performance Estimator

- Enumerates possible set of kernel implementation parameters
  - E.g, tile size, thread block size, …
- Prunes definitely slow cases based on the performance model
  - The model estimate not the exact performance but the performance upper bound
  - Checks for four maior factors for performance limitations



# Performance Estimator

- Global memory bandwidth
- Shared memory latency
- Workload imbalance across SMs
- Limitation of hardware resources



# Kernel Implementation Parameters

**Table 3.** Kernel Implementation Parameters Obtained from Operation Definition

| Symbol | Parameters |
|---|---|
| $N, C, K, H, W$ | Batch size($N$), numbers of input/output channels($C$, $K$), height/width of the output feature map($H$, $W$) |
| $F_W$, $F_H$, $P_W$, $P_H$, $S_W$, $S_H$ | width-wise and height-wise sizes of the filter ($F$), padding ($P$), and stride ($S$) |

**Table 4.** Kernel Implementation Parameters to Search for

| Symbol | Parameters |
|---|---|
| $N_{block}$, $K_{block}$, $H_{block}$, $W_{block}$ | Number of output images in the batch ($N_{block}$), number of output channels ($K_{block}$), width ($W_{block}$) and height ($H_{block}$) of the output feature map that a thread block computes |
| $N_{thread}$, $K_{thread}$, $H_{thread}$, $W_{thread}$ | Number of output images in the batch ($N_{thread}$), number of output channels ($K_{thread}$), width ($W_{thread}$) and height ($H_{thread}$) of the output feature map that a thread computes |
| $C_{input}$ | Number of input channels processed by a thread block per iteration |
| $FORMAT_{shared}$ | Input data layout in shared memory |

- $N_{thread}$, $K_{thread}$, $H_{thread}$, and $W_{thread}$ are set to powers of two.
- $N_{block}$, $K_{block}$, $H_{block}$, and $W_{block}$ are multiples of $N_{thread}$, $K_{thread}$, $H_{thread}$, and $W_{thread}$, respectively.

# Performance Limiting Factors

**Table 6.** Amount of Computation per Thread Block

| Algorithm | Formula |
|---|---|
| Direct convolution | $2 \cdot N_{block} \cdot C \cdot K_{block} \cdot H_{block} \cdot W_{block} \cdot F_H \cdot F_W$ |
| Matrix multiplication | $2 \cdot N_{block} \cdot C_{block} \cdot K_{block}$ |
| Batch normalization | $3 \cdot N_{block} \cdot K_{block} \cdot H_{block} \cdot W_{block}$ |
| Winograd transform | $2.25 \cdot N_{block} \cdot K_{block} \cdot H_{block} \cdot W_{block}$ |
| Pooling | $N_{block} \cdot K_{block} \cdot H_{block} \cdot W_{block} \cdot F_H \cdot F_W$ |
| Element-wise operation | $N_{block} \cdot K_{block} \cdot H_{block} \cdot W_{block}$ |

**Table 8.** Global Memory Transactions per Thread Block

| Algorithm | Formula |
|---|---|
| Direct convolution | $N_{block} \cdot C \cdot (\lfloor (H_{block} + F_H - 2 \cdot P_H)/S_H \rfloor + 1) \cdot (\lceil (\lfloor (W_{block} + F_W - 2 \cdot P_W)/S_W \rfloor + 1)/TRANS_{global} \rceil) + \lceil (K_{block} \cdot C \cdot F_W \cdot F_H)/TRANS_{global} \rceil$ |
| Matrix multiplication | $(N_{block} \cdot \lceil C_{block}/TRANS_{global} \rceil + C_{block} \cdot \lceil K_{block}/TRANS_{global} \rceil$ |
| Batch normalization | $N_{block} \cdot K_{block} \cdot H_{block} \cdot (\lceil W_{block}/TRANS_{global} \rceil) + 3 \cdot \lceil (K_{block}/TRANS_{global}) \rceil$ |
| Winograd transform | $N_{block} \cdot K_{block} \cdot H_{block} \cdot (\lceil W_{block}/TRANS_{global} \rceil)$ |
| Pooling | $N_{block} \cdot K_{block} \cdot (\lfloor (H_{block} + F_H - 2 \cdot P_H)/S_H \rfloor + 1) \cdot \lceil (\lfloor (W_{block} + F_W - 2 \cdot P_W)/S_W \rfloor + 1)/TRANS_{global} \rceil$ |
| Element-wise unary op. | $N_{block} \cdot K_{block} \cdot H_{block} \cdot \lceil W_{block}/TRANS_{global} \rceil$ |
| Element-wise binary op. | $2 \cdot N_{block} \cdot K_{block} \cdot H_{block} \cdot \lceil W_{block}/TRANS_{global} \rceil$ |

the operational intensity of a thread block:

$$OI_{TB} = COMP_{block}/(SIZE_{element} \cdot TRANS_{global} \cdot NUM_{trans})$$

Table 6 Table 8

global memory bandwidth:

$$GMRatio = Min(1, OI_{TB}/(R_{peak}/BW_{global})$$

theoretical peak operational intensity

If $GMRatio$ = 1, the global memory bandwidth does not bound the performance.

# Performance Limiting Factors

**Table 6.** Amount of Computation per Thread Block

| Algorithm | Formula |
|---|---|
| Direct convolution | $2 \cdot N_{block} \cdot C \cdot K_{block} \cdot H_{block} \cdot W_{block} \cdot F_H \cdot F_W$ |
| Matrix multiplication | $2 \cdot N_{block} \cdot C_{block} \cdot K_{block}$ |
| Batch normalization | $3 \cdot N_{block} \cdot K_{block} \cdot H_{block} \cdot W_{block}$ |
| Winograd transform | $2.25 \cdot N_{block} \cdot K_{block} \cdot H_{block} \cdot W_{block}$ |
| Pooling | $N_{block} \cdot K_{block} \cdot H_{block} \cdot W_{block} \cdot F_H \cdot F_W$ |
| Element-wise operation | $N_{block} \cdot K_{block} \cdot H_{block} \cdot W_{block}$ |

**Table 7.** Shared Memory Load Operations per Thread

| Algorithm | Formula |
|---|---|
| Direct convolution | $N_{thread} \cdot C \cdot (\lfloor (H_{thread} + F_H - 2 \cdot P_H)/S_H \rfloor + 1) \cdot (\lfloor (W_{thread} + F_W - 2 \cdot P_W)/S_W \rfloor + 1) + (K_{thread} \cdot C \cdot F_W \cdot F_H)$ |
| Matrix multiplication | $N_{thread} \cdot C_{thread} + C_{thread} \cdot K_{thread}$ |
| Batch normalization | $N_{thread} \cdot K_{thread} \cdot H_{thread} \cdot W_{thread} + 3 \cdot K_{thread}$ |
| Winograd transform | $N_{thread} \cdot K_{thread} \cdot H_{thread} \cdot W_{thread}$ |
| Pooling | $N_{thread} \cdot K_{thread} \cdot H_{thread} \cdot W_{thread} \cdot F_H \cdot F_W$ |
| Element-wise unary op. | $N_{thread} \cdot K_{thread} \cdot H_{thread} \cdot W_{thread}$ |
| Element-wise binary op. | $2 \cdot N_{thread} \cdot K_{thread} \cdot H_{thread} \cdot W_{thread}$ |

shared memory latency:

$$SMRatio = Min(1, (COMP_{thread}/N_{load})/(LAT_{shared} \cdot COEF_{bc}))$$

Table 6 Table 7
(replace block with thread)

$SMRatio$ measures how much of the shared memory latency is hidden by other compute operations

For example, if a thread performs 10 floating-point operations per shared memory load, where the shared memory load latency is 20 cycles and there are no bank conflicts, then 10 cycles out of 20 cycles are hidden. In this case, $SMRatio$ = 0.5. If the number of operations is bigger than 20, $SMRatio$ becomes 1.

# Performance Limiting Factors

**Table 4.** Kernel Implementation Parameters to Search for

| Symbol | Parameters |
|---|---|
| $N_{block}$, $K_{block}$, $H_{block}$, $W_{block}$ | Number of output images in the batch ($N_{block}$), number of output channels ($K_{block}$), width ($W_{block}$) and height ($H_{block}$) of the output feature map that a thread block computes |
| $N_{thread}$, $K_{thread}$, $H_{thread}$, $W_{thread}$ | Number of output images in the batch ($N_{thread}$), number of output channels ($K_{thread}$), width ($W_{thread}$) and height ($H_{thread}$) of the output feature map that a thread computes |
| $C_{input}$ | Number of input channels processed by a thread block per iteration |
| $FORMAT_{shared}$ | Input data layout in shared memory |

**Table 5.** GPU Architecture Parameters

| Symbol | Parameters |
|---|---|
| $NUM_{SM}$ | Number of streaming multiprocessors. |
| $PERF_{peak}$ | Peak performance of single-precision floating-point operations |
| $BW_{global}$ | Bandwidth of the global memory |
| $TRANS_{global}$ | Maximum number of input elements in a global memory transaction |
| $LAT_{shared}$ | Latency of the shared memory |
| $MAX_{shared}$ | Maximum size of shared memory allocation for a thread block |
| $MAX_{thread}$ | Maximum number of threads in a thread block |

workload imbalance between SMs:

$$N_{TB} = (N/N_{block}) \cdot (K/K_{block}) \cdot (H/H_{block}) \cdot (W/W_{block})$$

$$WBRatio = 1 - ((N_{TB} \mod N_{SM})/N_{SM})/\lceil (N_{TB}/N_{SM}) \rceil$$

where $N_{TB}$ is the number of thread blocks and $N_{SM}$ is the number of SMs.

The higher $W\ BRatio$, the higher the expected performance.

# Performance Limiting Factors

**Table 4.** Kernel Implementation Parameters to Search for

| Symbol | Parameters |
|---|---|
| $N_{block}$, $K_{block}$, $H_{block}$, $W_{block}$ | Number of output images in the batch ($N_{block}$), number of output channels ($K_{block}$), width ($W_{block}$) and height ($H_{block}$) of the output feature map that a thread block computes |
| $N_{thread}$, $K_{thread}$, $H_{thread}$, $W_{thread}$ | Number of output images in the batch ($N_{thread}$), number of output channels ($K_{thread}$), width ($W_{thread}$) and height ($H_{thread}$) of the output feature map that a thread computes |
| $C_{input}$ | Number of input channels processed by a thread block per iteration |
| $FORMAT_{shared}$ | Input data layout in shared memory |

**Table 5.** GPU Architecture Parameters

| Symbol | Parameters |
|---|---|
| $NUM_{SM}$ | Number of streaming multiprocessors. |
| $PERF_{peak}$ | Peak performance of single-precision floating-point operations |
| $BW_{global}$ | Bandwidth of the global memory |
| $TRANS_{global}$ | Maximum number of input elements in a global memory transaction |
| $LAT_{shared}$ | Latency of the shared memory |
| $MAX_{shared}$ | Maximum size of shared memory allocation for a thread block |
| $MAX_{thread}$ | Maximum number of threads in a thread block |

limitation of hardware resources:

$$COEF_r = \begin{cases} 1 & NUM_{thread} < MAX_{thread} \\ & \text{and} \quad SIZE_{shared} < MAX_{shared} \\ 0 & otherwise \end{cases}$$

# Performance Estimator

$$PUL = GMRatio \cdot SMRatio \cdot WBRatio \cdot COEF_r$$

- $PUL$ represents the ratio of the expected maximum performance of the tensor operation to the peak theoretical performance of the target GPU.
  - If $PUL$ = 0.7, we cannot obtain more than 70% of the theoretical peak performance of the GPU with the given implementation parameters
- DeepCuts computes PUL of every possible implementation parameters
- Then pick the top 1% of the best performing combinations
  - They test on 3*3 convolution and do not find any case that the performance estimator prunes the best-performing set of parameters

# Shared-Memory-Level and Register-Level Fusion

**Table 2.** Types of Tensor Operations Handled by DeepCuts

| Category | Node type |
|---|---|
| Simple | Unary element-wise operations (*e.g.*, ReLU, tanh, sigmoid), Binary element-wise operations (*e.g.*, matrix addition, matrix subtraction), Winograd transformation, 2D FFT, Depthwise separable convolution, Pooling, Batch normalization, Layer normalization |
| Complex | Direct convolution, Matrix multiplication |

# Shared-Memory-Level and Register-Level Fusion

at the register level
1) simple + simple
2) complex + simple

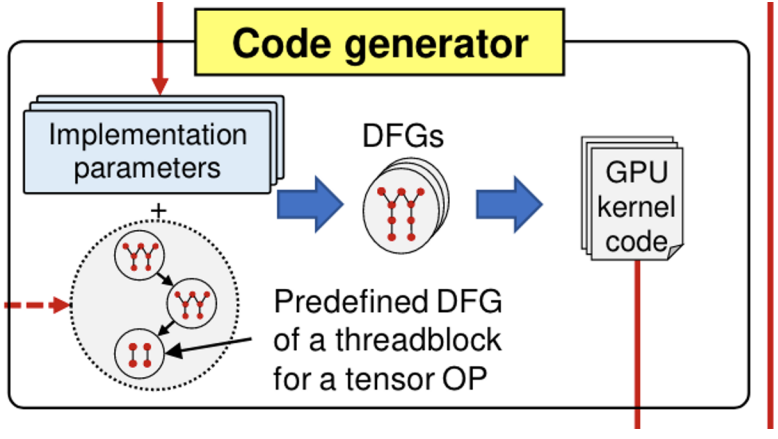
- $NUM_{trans}$ and $N_{load}$ of the fused operation is the same as those of the predecessor.
- $COMP_{block}$ and $COMP_{thread}$ of the fused operation is the sum of those of the predecessor and the successor.

at the shared memory level
1) complex + complex

- $NUM_{trans}$ for the successor is set to 0 (*i.e.*, $GMRatio$ is set to 1) because all input data to the successor that are generated by the predecessor are stored in the shared memory.
- The predecessor's output size per thread block should match the input size of the successor. The estimation model checks the implementation parameters, and discard unmatched cases.
- When the successor is a complex operation, its $C_{input}$ should be equal to $C$. This implies that all input data of the successor is assumed to be stored in the shared memory.

# Code Generator

- Generates GPU kernels based on the kernel implementation parameters
- Data Flow Graph (DFG) is used for the code generation
  - To support versatile types of DL operations and their fusions
- Step1: Baseline DFG generation
- Step2: DFG concatenation
- Step3: Subgraph extraction for a thread



# Step1: Baseline DFG generation

$$S = \{N = 1, C = 2, K = 2, W = 4, H = 4, F_W = 1, F_H = 1,$$
$$P_W = 0, P_H = 0, S_W = 1, S_H = 1,$$
$$N_{block} = 1, K_{block} = 1, W_{block} = 2, H_{block} = 2, C_{input} = 2,$$
$$N_{thread} = 1, K_{thread} = 1, W_{thread} = 2, H_{thread} = 1\}.$$

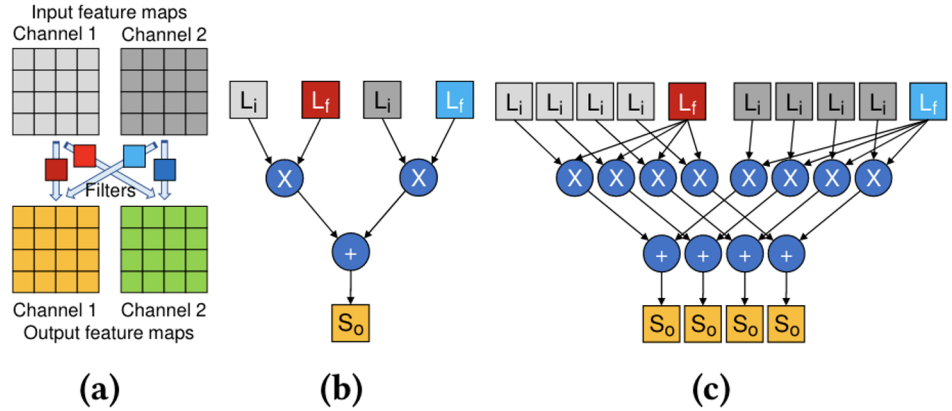



**Figure 3.** Baseline DFG construction. (a) $1 \times 1$ convolution operation with 2 input channels and 2 output channels. (b) The DFG of an output pixel when $C_{input} = 2$. (c) The baseline DFG for a thread block when $C_{input} = 2$.

# Step2: DFG concatenation

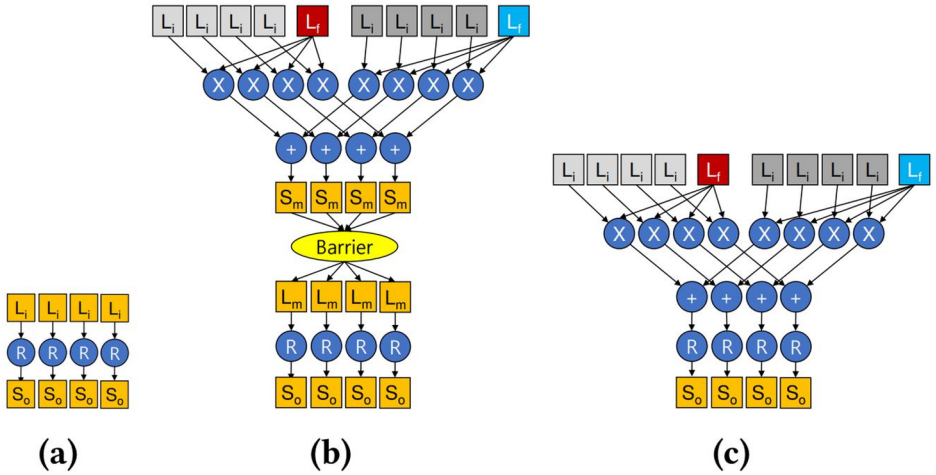



**Figure 4.** DFG concatenation. (a) Baseline DFG for ReLU operation. (b) Shared-memory-level fusion of a $1 \times 1$ convolution and a ReLU operation. (c) Register-level fusion of a $1 \times 1$ convolution and a ReLU operation.

# Step3: Subgraph extraction for a thread

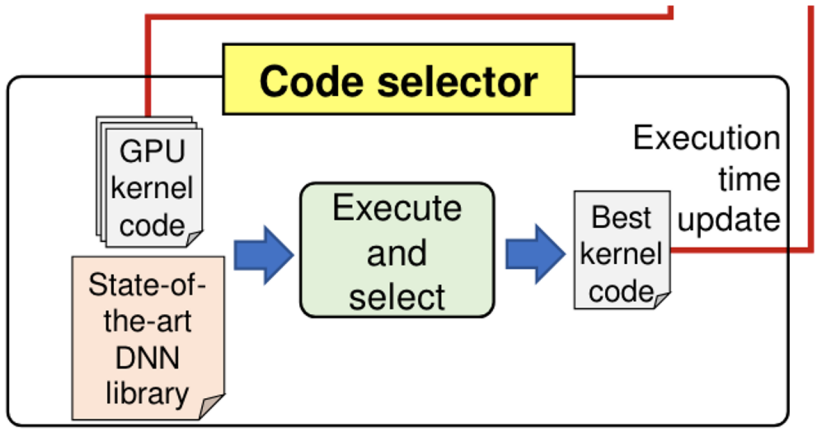
- CUDA kernel code describes the computation of a GPU thread
- DeepCuts partitions the last shared memory store operations into multiple equal-sized chunks
  - the size is defined by the implementation parameters ($Nthread$ , $Cthread$ , $Hthread$ , $Wthread$ )
  - each chunk is assigned to a different thread
- DeepCuts follows the edges in the reverse direction from the assigned store operations
  - marks all nodes visited in the traversal
  - after reaching the nodes with no predecessors, DeepCuts removes all unmarked nodes to extract the sub-DFG for a thread.

# Code Generator

- The code generator applies the following optimizations to the kernel
  - Coalesced access to GPU global memory
  - Exploiting vector I/O instructions
  - Bank-conflict-aware padding on shared memory
  - Prefetching data to hide global memory latency

# Code Selector

- Selects the best GPU kernel
  - Compares the performance with CuDNN and uses the better one



# Evaluation

**Table 1.** Comparison of DL Optimization Frameworks

| | | DeepCuts | TVM (autoTVM) | TensorFlow XLA | TensorRT |
|---|---|---|---|---|---|
| Models supported | CNNs | ✔ | ✔ | ✔ | ✔ |
| | RNNs | ✔ | | ✔ | ✔ |
| | TMLPs[a] | ✔ | | ✔ | ✔ |
| Workload types | Inference | ✔ | ✔ | ✔ | ✔ |
| | Training | ✔ | | ✔ | |
| Convolution algorithms | Winograd | ✔ | ✔ | ✔ | ✔ |
| | FFT | ✔ | | ✔ | ✔ |
| SOTA[b] performance | CNNs | ✔ | partial | partial | partial |
| | RNNs | ✔ | | ✔ | |
| | TMLPs | ✔ | | ✔ | ✔ |
| Key idea for GPU kernel optimization | | Architecture-aware performance model | ML-based performance model | Manually tuned library | Manually tuned library |

[a] TMLP: transformer-based MLP. [b] SOTA: state-of-the-art.

# Evaluation - setup

- 4 CNNs. 2 RNNs, and 2 transformer-based MLPs are used
  - For both inference and training workloads
  - Total 32 workloads
- A V100 GPU and an RTX 2080 GPU are used

**Table 9.** System Configuration & Software Versions

| | |
|---|---|
| CPU | 2 x Intel Xeon Gold 6130 CPU (16-core, 2.1GHz) |
| GPU | NVIDIA Tesla V100, Volta architecture (5120 CUDA cores, 16GB HBM2)<br>NVIDIA GeForce RTX 2080, Turing architecture (2944 CUDA cores, 8GB GDDR6) |
| Memory | 256GB (DDR4 2400MHz) |
| Software | CUDA 11.2, cuDNN 8.0.5, PyTorch 1.7.1, TVM 0.8.dev, TensorFlow 1.14.0 and 2.4.1, TensorRT 7.2.2 |

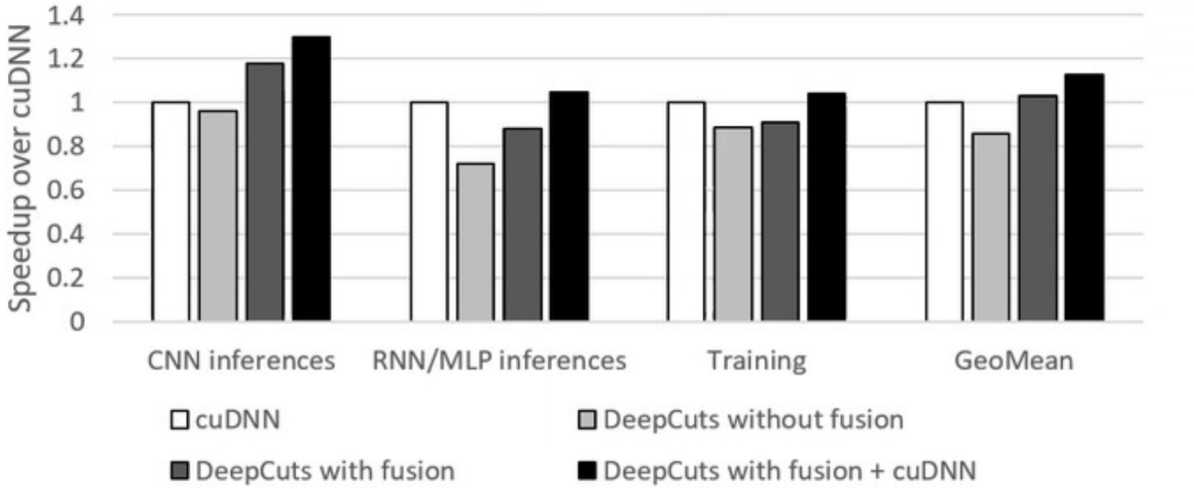
**Table 10.** Deep Learning Benchmark Applications

| | Architecture | Dataset |
|---|---|---|
| CNN | ResNet-50[24] | ImageNet Dataset[20] |
| | DenseNet-121[25] | |
| | Inception-v3[40] | |
| | MobileNetV2[38] | |
| RNN | DeepSpeech2[11] | AN4[4] |
| | Attention-RNN[30] | WMT'17[6] |
| MLP | Facebook-DLRM[31] | Kaggle Display Advertising Challenge Dataset [5] |
| | BERT[21] | SQUAD 1.1[37] |

# Evaluation - performance speedup

- Three versions of DeepCuts are used
  - Non-fused kernels only
  - With fusion
  - With fusion and cuDNN primitives

- 1.13X faster than cuDNN on average
  - 1.03X faster without using CuDNN primitives
- For CNN inferences
  - 1.3X faster than cuDNN for CNN inferences
  - 1.18X faster than cuDNN without using cuDNN primitives
- For non-CNN model inferences
  - 1.05X faster than cuDNN on average
- For training
  - 1.04X faster than cuDNN on average



# Evaluation - performance speedup

**Table 11.** # of Top-performing Workloads on V100

| Model Type | Number of top-performing workloads | | | | |
|---|---|---|---|---|---|
| | TensorFlow -XLA | TensorRT | TVM -default | TVM -autotvm | DeepCuts -mixed |
| CNN | 1 | 1 | 3 | 2 | 13 |
| RNN | 0 | 0 | 0 | 0 | 6 |
| MLP | 0 | 2 | 0 | 0 | 4 |
| Total | 1 | 3 | 3 | 2 | 23 |

**Table 12.** # of Top-performing Workloads on RTX 2080

| Model Type | Number of top-performing workloads | | | | |
|---|---|---|---|---|---|
| | TensorFlow -XLA | TensorRT | TVM -default | TVM -autotvm | DeepCuts -mixed |
| CNN | 1 | 5 | 1 | 1 | 12 |
| RNN | 0 | 0 | 0 | 0 | 6 |
| MLP | 0 | 2 | 0 | 0 | 4 |
| Total | 1 | 7 | 1 | 1 | 22 |

# Evaluation - code generation latency

13.53× faster than TVM-autotvm!

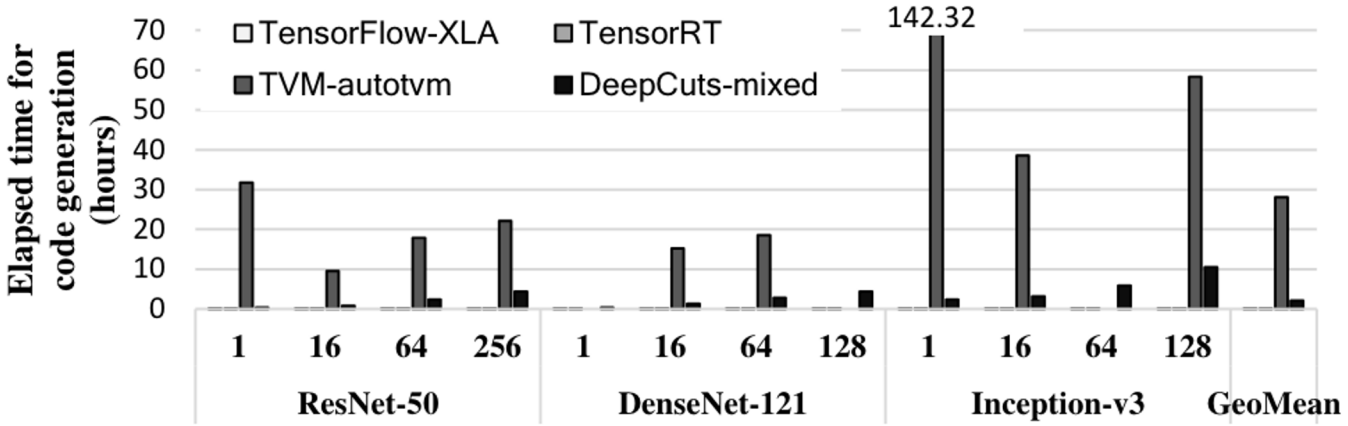



**Figure 7.** Elapsed time for code generation. The bars for the elapsed times of TensorFlow XLA and TensorRT are almost invisible because they take only a few seconds.

# Conclusion

- candidate generator (greedy search)
- performance estimator (upper bound estimation, choose top 1%)
- code generator (DFG concatenation and some optimizations)
- code selector (compared with cuDNN)

+ Transfer common tuning practice to the framework, such as the greedy search and parameter selection.
+ Design four clear and concise metrics for performance upper bound estimation.
- After pruning still need to execute one by one to find the best case, that’s why the code generation latency is high under large batch size.
- The kernel fusion approach is simple compared with DNNFusion.
- The provided metrics make the application scenarios limited since it only supports vertical concatenation and has requirements on the fused operations’ shapes.