Chameleon: Adaptive Code Optimization for Expedited Deep Neural Network Compilation

Byung Hoon Ahn, Prannoy Pilligundla, Amir Yazdanbakhsh, Hadi Esmaeilzadeh

ICLR 2020

University of California, San Diego ,Google Research

The general life-cycle of deep learning models

1st

designing &training model

A research scientist

The highest accuracy

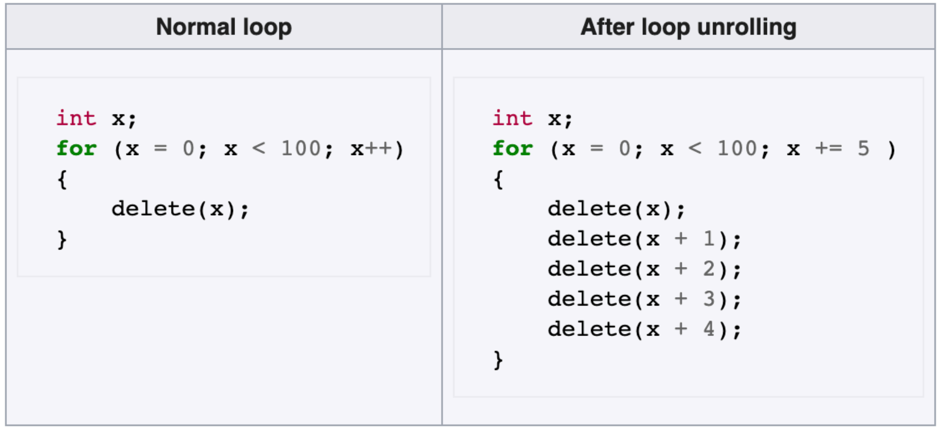
2nd

deploy to target

A deployment engineer

The inference speed

Compilation Workflow For Deep Neural Network



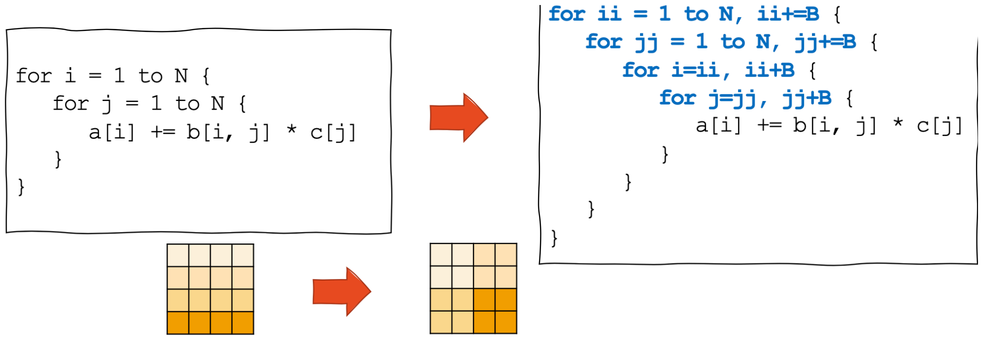
Normal loop

```
int x;
for (x = 0; x < 100; x++)
{
    delete(x);
}
```

After loop unrolling

```
int x;
for (x = 0; x < 100; x += 5 )
{
    delete(x);
    delete(x + 1);
    delete(x + 2);
    delete(x + 3);
    delete(x + 4);
}
```

Loop Unrolling



Loop Tiling

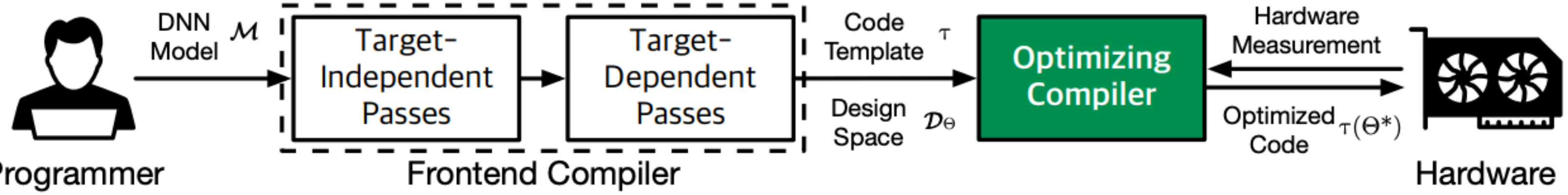


Figure 1: Overview of our model compilation workflow, and highlighted is the scope of this work.

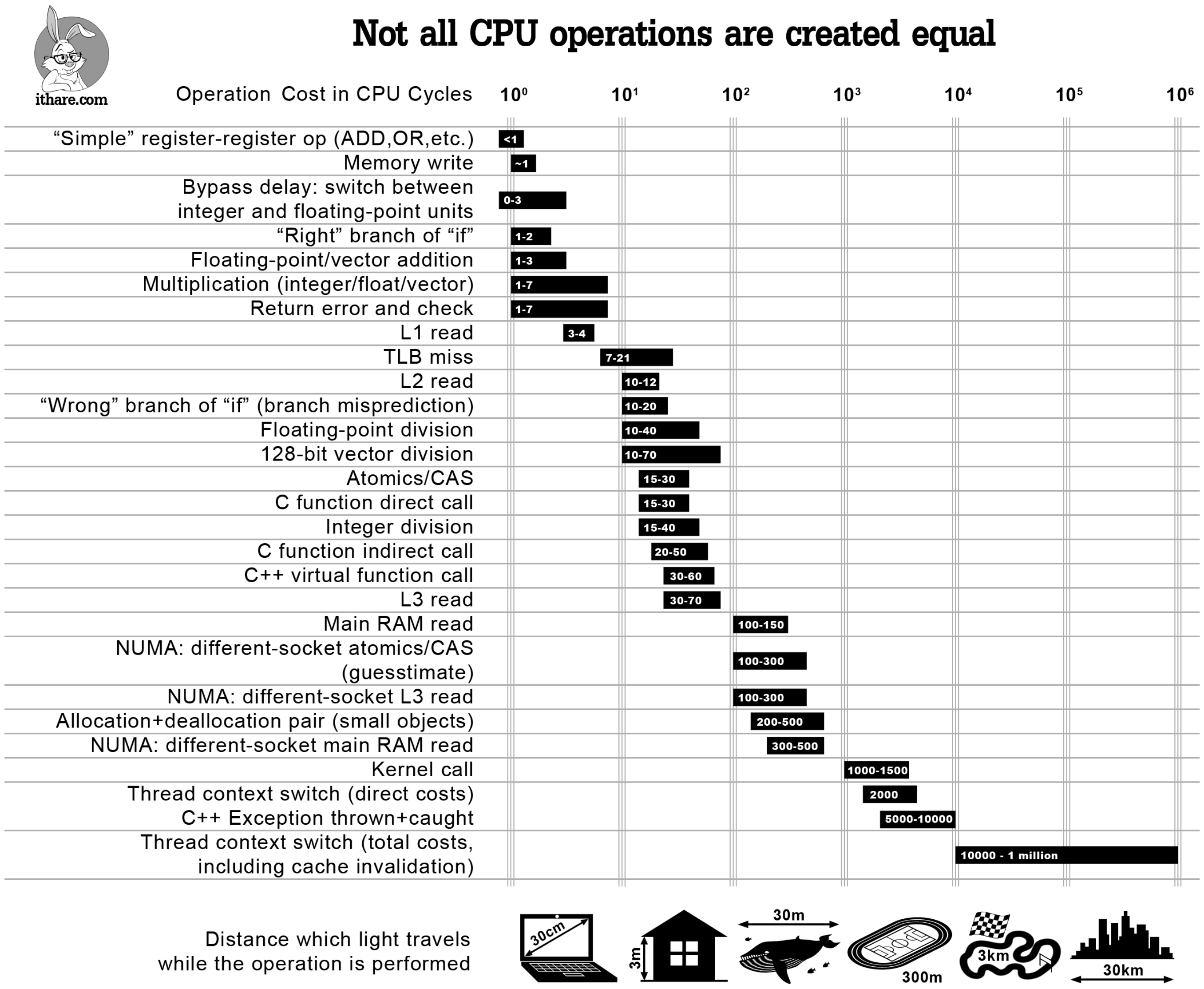
Compilation Workflow For Deep Neural Network

Normal loop

```
int x;
for (x = 0; x < 100; x++)
{
    delete(x);
}
```

```
B {
j+=B {
{
+B {
[i, j] * c[j]
```

Loop Unrol



DI
Mc

Programmer

are

Figure 1: Overview of this work.

http://ithare.com/infographics-operation-costs-in-cpu-clock-cycles/

Optimizing Complier For Deep Neural Networks

$$\Theta^* = \underset{\Theta}{\operatorname{argmax}} f(\tau(\Theta)), \qquad \text{for } \Theta \in \mathcal{D}_\Theta. \qquad \text{knobs } \theta \quad \Theta = (\theta_1, \theta_2, ..., \theta_n)$$

e compute expression

```
A = t.placeholder((1024, 1024))
B = t.placeholder((1024, 1024))
k = t.reduce_axis((0, 1024))
C = t.compute((1024, 1024),
    lambda y, x:
    t.sum(A[k, y] * B[k, x], axis=k))
```

x_0 default code

```
for y in range(1024):
  for x in range(1024):
    C[y][x] = 0
    for k in range(1024):
      C[y][x] += A[k][y] * B[k][x]
```

s_1 loop tiling

```
yo, xo, yi, xi = s[C].title(y, x, ty, tx)
s[C].reorder(yo, xo, k, yi, xi)
```

$x_1 = g(e, s_1)$

```
for yo in range(1024 / ty):
  for xo in range(1024 / tx):
    C[yo*ty:yo*ty+ty][xo*tx:xo*tx+tx] = 0
    for k in range(1024):
      for yi in range(ty):
        for xi in range(tx):
          C[yo*ty+yi][xo*tx+xi] +=
            A[k][yo*ty+yi] * B[k][xo*tx+xi]
```

- An optimizing compiler input τ(task e.g. conv2d,dense..) for each layer
- Use and learn of a search algorithm f
- Knob Θ is such thing like compile option(e.g. tiling factor..)
- Target is the best configuration Θ^* ∈ DΘ in real hardware

Optimizing Complier For Deep Neural Networks

$$\Theta^* = \operatorname*{argmax}_{\Theta} f(\tau(\Theta)), \qquad \text{for } \Theta \in \mathcal{D}_\Theta. \qquad \text{knobs } \theta \quad \Theta = (\theta_1, \theta_2, ..., \theta_n)$$

e compute expression

```
A = t.placeholder((1024, 1024))
B = t.placeholder((1024, 1024))
k = t.reduce_axis((0, 1024))
C = t.compute((1024, 1024),
   lambda y, x:
   t.sum(A[k, y] * B[k, x], axis=k))
```

x_0 default code

```
for y in range(1024):
  for x in range(1024):
    C[y][x] = 0
    for k in range(1024):
      C[y][x] += A[k][y] * B[k][x]
```

s_1 loop tiling

```
yo, xo, yi, xi = s[C].title(y, x, ty, tx)
s[C].reorder(yo, xo, k, yi, xi)
```

$x_1 = g(e, s_1)$

```
for yo in range(1024 / ty):
  for xo in range(1024 / tx):
    C[yo*ty:yo*ty+ty][xo*tx:xo*tx+tx] = 0
    for k in range(1024):
      for yi in range(ty):
        for xi in range(tx):
          C[yo*ty+yi][xo*tx+xi] +=
            A[k][yo*ty+yi] * B[k][xo*tx+xi]
```

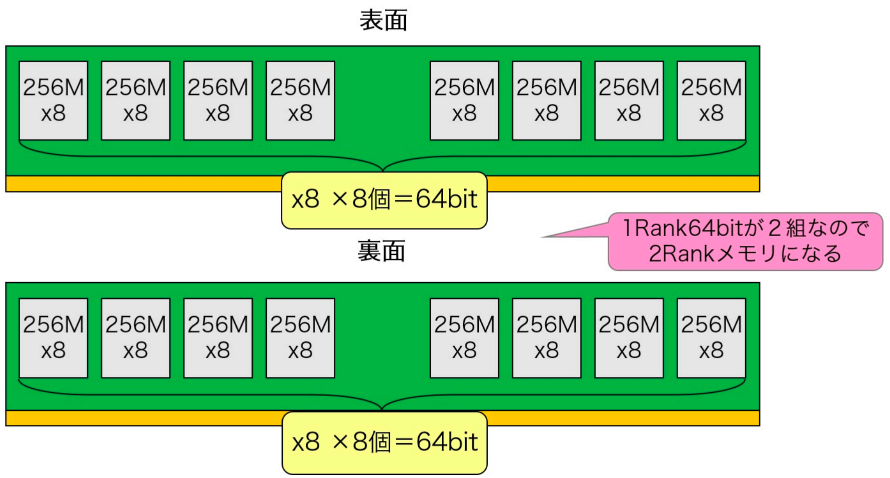
- For the effectiveness of the optimizing compiler
  - A large and diverse enough design space => variety of transformations opportunity
  - An effective search algorithm => adequately navigate this space
  - A mechanism to cut down the number of hardware measurements => reduce hardware measure overhead

Optimizing Complier For Deep Neural Networks

- The knobs optimize various aspects of the execution
  - maximizes data reuse
  - uses the shared memory wisely
  - minimizes bank conflicts
- This work has a design space with 10^10 possibilities.

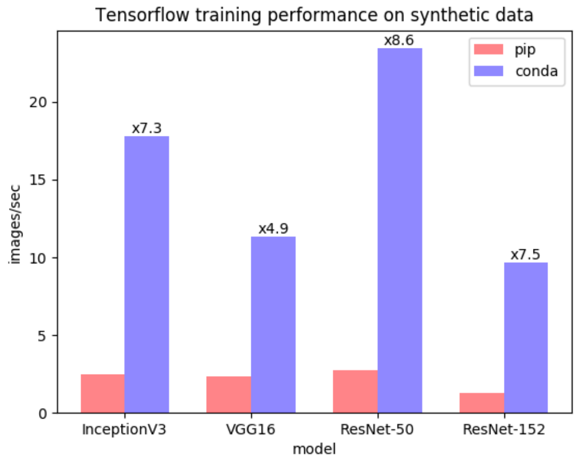
KNOBS	DEFINITION
tile_f, tile_y, tile_x	Factors for tiling and binding # of filters height, and width of feature maps.
tile_rc, tile_ry, tile_rx	Factors for tiling reduction axis such as # of channels, height, and width of filters.
auto_unroll_max_step	Threshold of number of steps in the loop to be automatically unrolled.
unroll_explicit	Explicitly unroll loop, this may let code generator to generate pragma unroll hint.

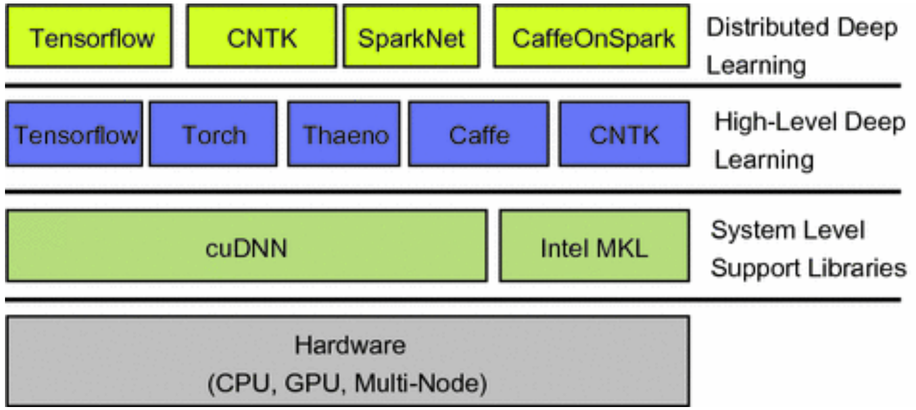
Table 1: Knobs in the design space to optimize convolution.



https://gigglehd.com/gg/hard/933610

Introduction



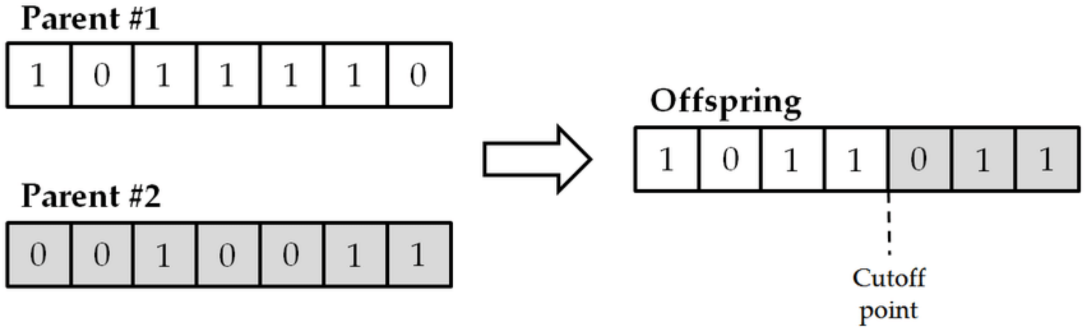


- The enormous computational intensity of DNNs
- Hand-optimized kernels, such as NVIDIA cuDNN or Intel MKL that serve as backend for a of programming environment such as TensorFlow and PyTorch.

https://towardsdatascience.com/stop-installing-tensorflow-using-pip-for-performance-sake-5854f9d9eb0c
https://link.springer.com/chapter/10.1007/978-981-13-1165-9_40

Previous works

- TensorComprehensions are based on random or genetic algorithms to search the space of optimized code for neural networks.
- Each new candidate is bred through three parent uniform cross-over and also undergoes mutation with a low probability



Previous works

- AutoTVM builds on top of TVM and leverage boosted trees as part of the search cost model.

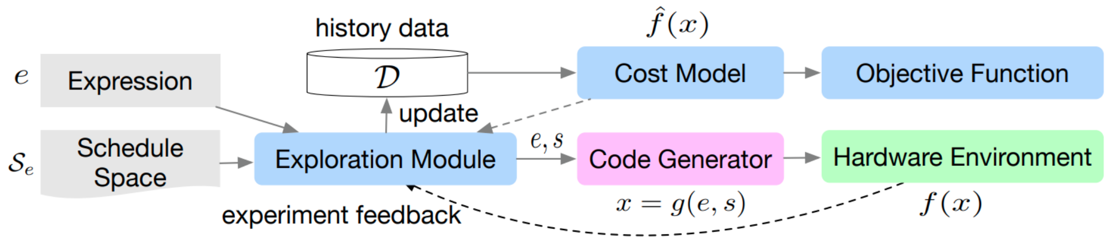
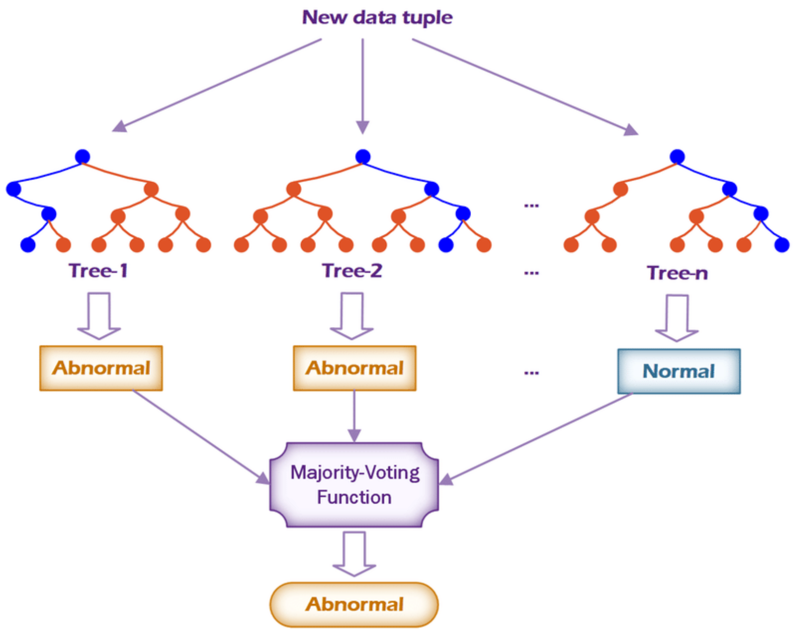
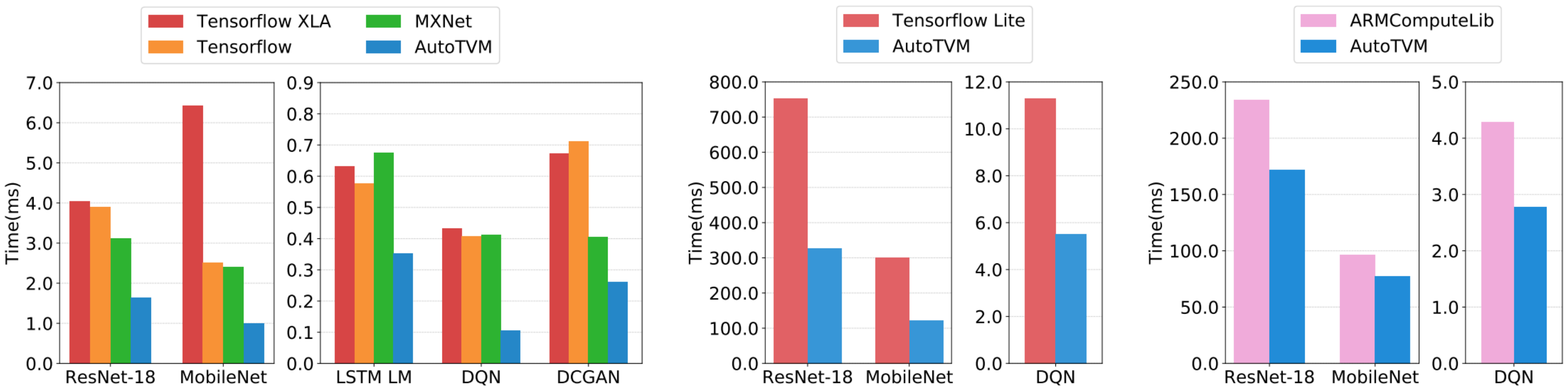


Figure 2: Framework for learning to optimize tensor programs.



xgboost

Previous works



(a) NVIDIA TITAN X End2End (b) ARM Cortex-A53 End2End (c) ARM Mali-T860 End2End

Figure 11: End-to-end performance across back-ends. [2]AutoTVM outperforms the baseline methods.

Problems

- Time can be around 10 hours for ResNet-18,
- Long compilation time hinders innovation.
- The current approaches are oblivious to the patterns in the design space of schedules that are available for exploitation, and causes inefficient search.
- The current solutions that rely on greedy sampling lead to significant fractions of the candidate configurations being redundant over iterations, compiler are prone to invalid configurations

Contributions

1. Devising an **Adaptive Exploration** module that utilizes *reinforcement learning* to adapt to *unseen design space* of new networks to *reduce search time* yet achieve better performance.

2. Proposing an **Adaptive Sampling algorithm** that utilizes *clustering to adaptively reduce the number* of costly hardware measurements, and devising a domain-knowledge inspired **Sample Synthesis** to find configurations that would potentially yield better performance.

Overall design and compilation overview

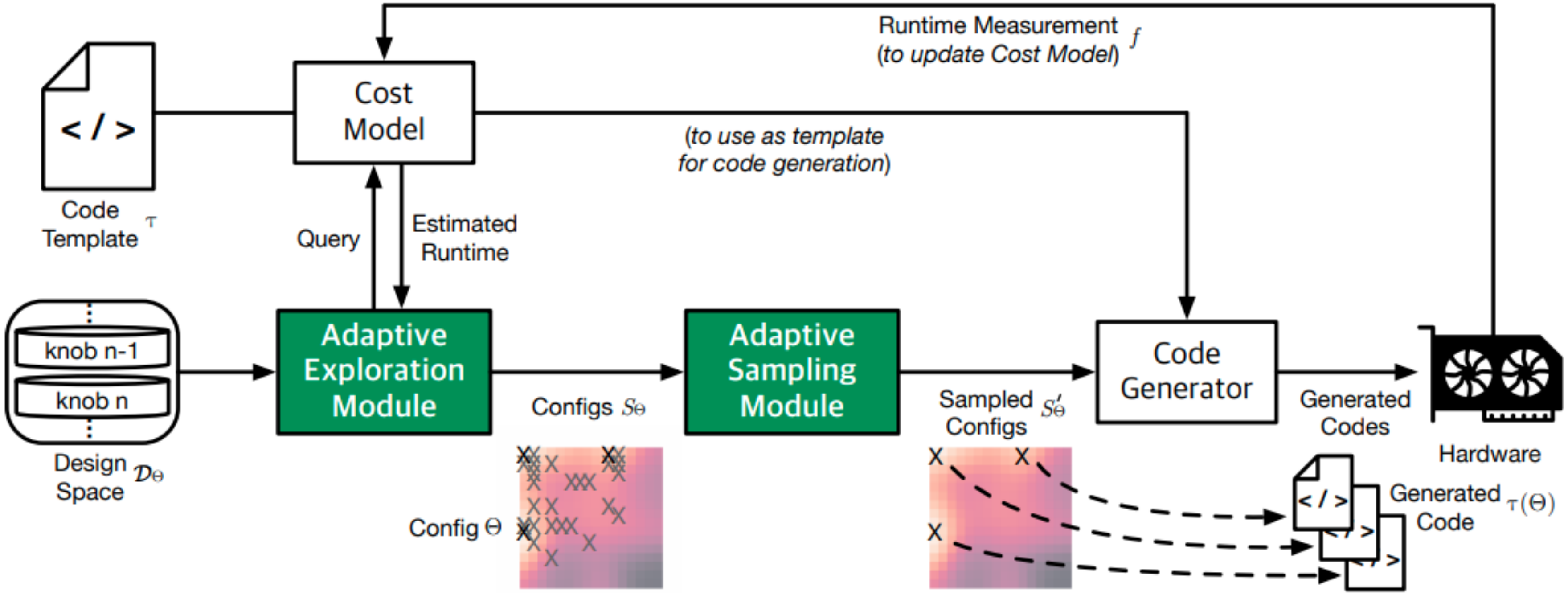


Figure 3: Overall design and compilation overview of the **CHAMELEON**.

Overall design and compilation overview

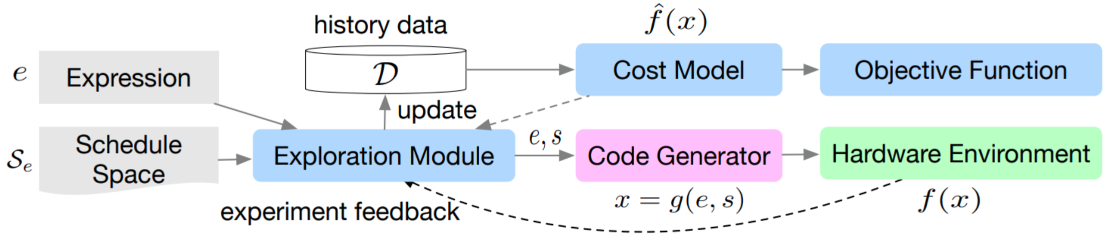


Figure 2: Framework for learning to optimize tensor programs.

Autotvm

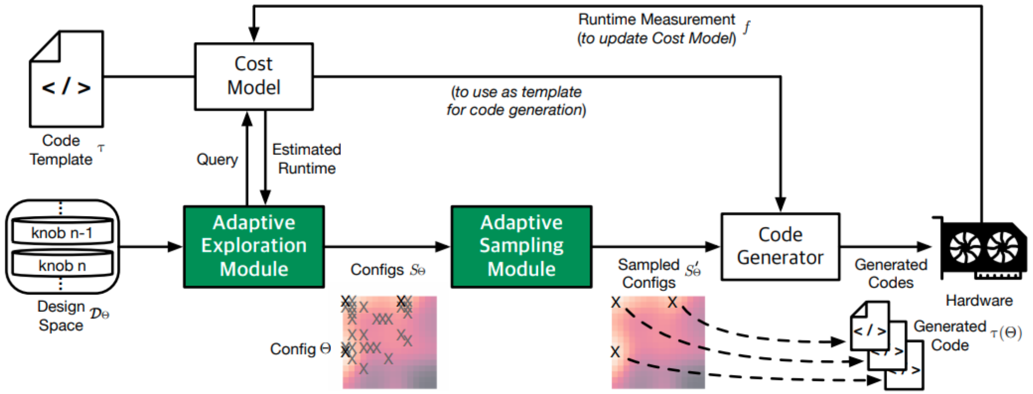


Figure 3: Overall design and compilation overview of the **CHAMELEON**.

Chameleon

Adaption Exploration

- The current approach requires numerous iterations of exploration to converge to a reasonable solution.

- Adaptive Exploration by leveraging Reinforcement Learning (RL), which is concerned with learning to maximize reward given an environment by *making good exploration and exploitation tradeoffs*

Reinforcement learning formulation

- Adaptive Exploration module uses an actor-critic style RL(PPO), where policy network learns to emit a set of directions (vector of increment/decrement/stay) for each knob in the design space.
- These networks not only learn the dependencies among the *different knobs of the design space* but also lean *the potential gains of the modifications* to the configurations

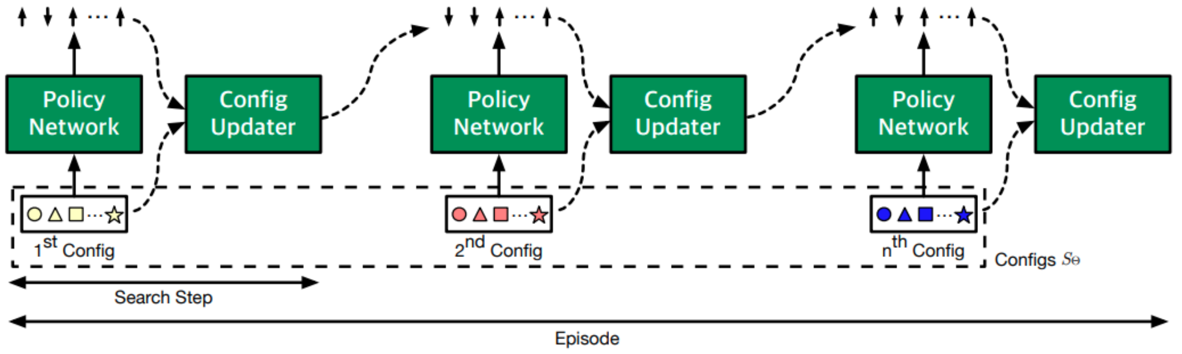
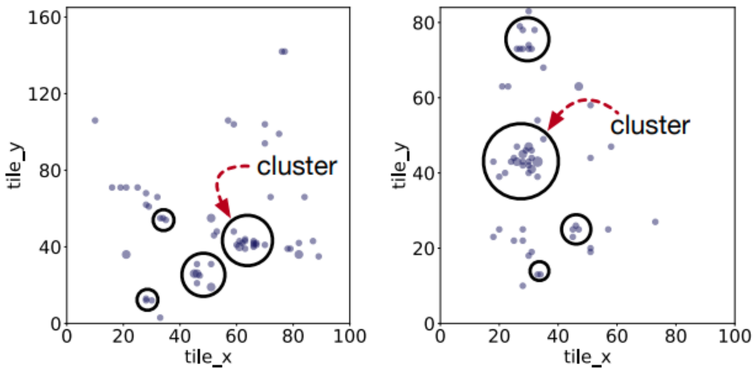


Figure 4: Adaptive Exploration Module of **CHAMELEON** in action.

Adaptive Sampling

- The candidate configurations are clustered in subregions of the design space and these clusters are non-uniformly distributed
- A large fraction of configurations within each **cluster achieve similar runtime**



(a) VGG-16 4th layer (b) ResNet-18 11th layer

Figure 5: Clusters of candidate configurations.

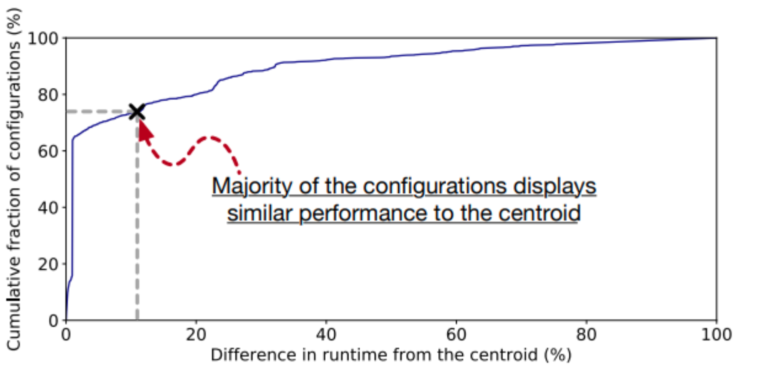


Figure 6: Cumulative Distribution Function (CDF) of the difference in runtime among the configurations in the cluster.

Sample Synthesis

- The configurations are discarded due to the greedy sampling which entirely depends on the cost model for its selections of the configurations.
- The proposed method analyzes the candidate samples to determine the most probable (most frequent = mode function) non-invalid choice for each knob to come up with a new configuration.
- This statistical combination of the most frequent knob settings yield configurations that combine the strengths of different knobs to converge to a better overall solution.

Adaptive Sampling & Sample Synthesis

Algorithm 1 Adaptive Sampling and Sample Synthesis

procedure ADAPTIVESAMPLING(s_Θ, v_Θ) ▷ s_Θ: candidate configs, v_Θ: visited configs
    new_candidates $\leftarrow \emptyset$, previous_loss $\leftarrow \infty$
    for k **in** range(8, 64) **do**
        new_candidates, clusters, L2_loss $\leftarrow$ K-means.run(s_Θ, k)
        if Threshold $\times$ L2_loss $\geq$ previous_loss **then** break ▷ Exit loop at *knee* of loss curve
        previous_loss $\leftarrow$ L2_loss
    end for
    for candidate **in** new_candidates **do** ▷ Replace visited config with new config
        if candidate **in** v_Θ **then** new_candidates.replace(candidate, mode(s_Θ))
    end for
    return new_candidates ▷ Feed to *Code Generator* to make measurements on hardware
end procedure

Experiment setup

Table 4: Details of the DNN models used in evaluating CHAMELEON.

NETWORK	DATASET	NUMBER OF TASKS
AlexNet	ImageNet	5
VGG-16	ImageNet	9
ResNet-18	ImageNet	12

Table 5: Details of the layers used in evaluating CHAMELEON.

NAME	MODEL	LAYER TYPE	TASK INDEX
L1	AlexNet	convolution	1
L2	AlexNet	convolution	4
L3	VGG-16	convolution	1
L4	VGG-16	convolution	2
L5	VGG-16	convolution	4
L6	ResNet-18	convolution	6
L7	ResNet-18	convolution	9
L8	ResNet-18	convolution	11

Table 6: Details of the hardware used for evaluation of CHAMELEON.

SPECIFICATIONS	DETAILS
GPU	Titan Xp
Host CPU	3.4G Hz Intel Core i7
Main Memory	32GB 2400 MHz DDR3

Table 7: Hyper-parameters uses in CHAMELEON.

HYPERPARAMETER	VALUE	DESCRIPTION
$iteration_{opt}$	16	number of iterations for optimization process (equivalent to 1000 hardware measurements)
$mode_{GBT}$	xgb-reg	type of loss used for cost model
b_{GBT}	64	maximum batch size of planning in GBT (Chen & Guestrin, 2016) cost model per iteration of optimization process
$episode_{rl}$	128	number of episodes for reinforcement learning
$step_{rl}$	500	maximum steps of one reinforcement learning episode
$threshold_{meta}$	2.5	threshold used for meta-search in sampling

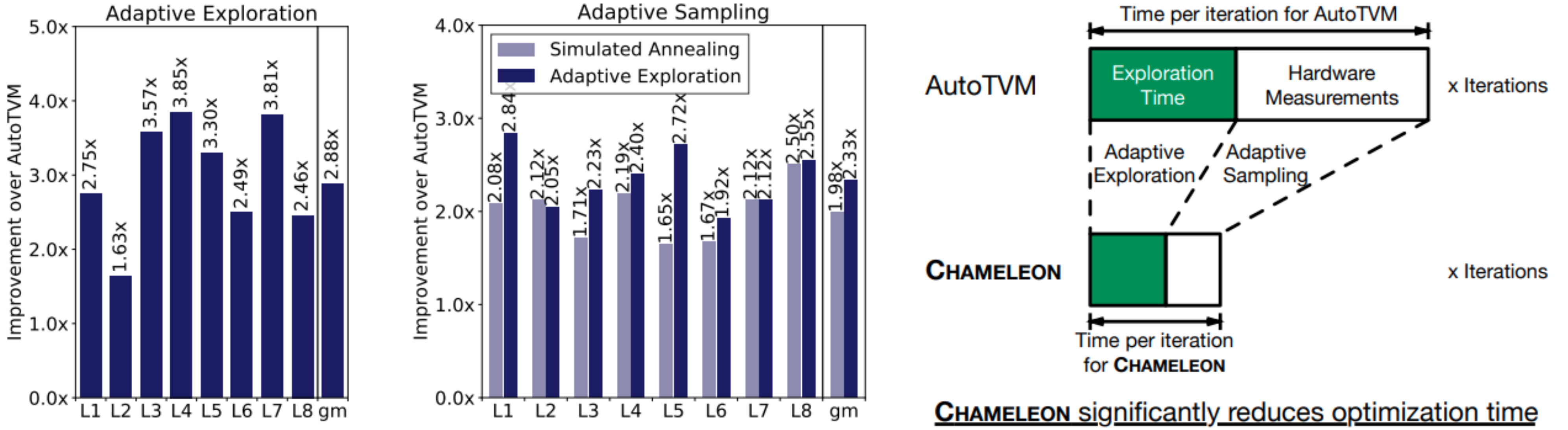
Table 8: Hyper-parameters uses in AutoTVM (Chen et al., 2018b).

HYPERPARAMETER	VALUE	DESCRIPTION
$\Sigma(b_{GBT})$	1000	total number of hardware measurements
$mode_{GBT}$	xgb-reg	type of loss used for cost model
b_{GBT}	64	batch size of planning in GBT (Chen & Guestrin, 2016)
n_{sa}	128	number of Markov chains in parallel simulated annealing
$step_{sa}$	500	maximum steps of one simulated annealing run

Table 9: Hyper-parameters used in CHAMELEON's PPO (Schulman et al., 2017) search agent.

HYPERPARAMETER	VALUE
Adam Step Size	1×10^{-3}
Discount Factor	0.9
GAE Parameter	0.99
Number of Epochs	3
Clipping Parameter	0.3
Value Coefficient	1.0
Entropy Coefficient	0.1

Evaluation



(a) Reduction in number of steps for convergence.

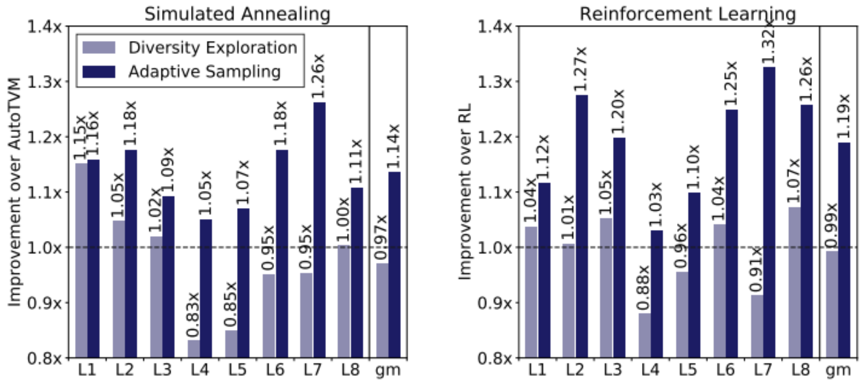
(b) Reduction in number of hardware measurements.

(c) Illustration of how the each component of CHAMELEON reduces the optimization time.

Figure 7: Component evaluation of CHAMELEON.

The Adaptive Sampling algorithm reduces the number of measurements by 1.98×when used with simulated annealing and 2.33×with reinforcement learning

Evaluation



(a) Simulated Annealing. (b) Reinforcement Learning.

Figure 8: Comparison to AutoTVM's diversity exploration.

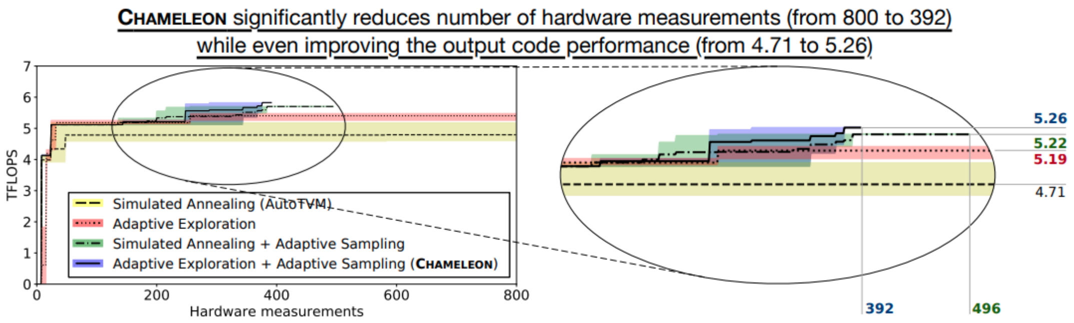
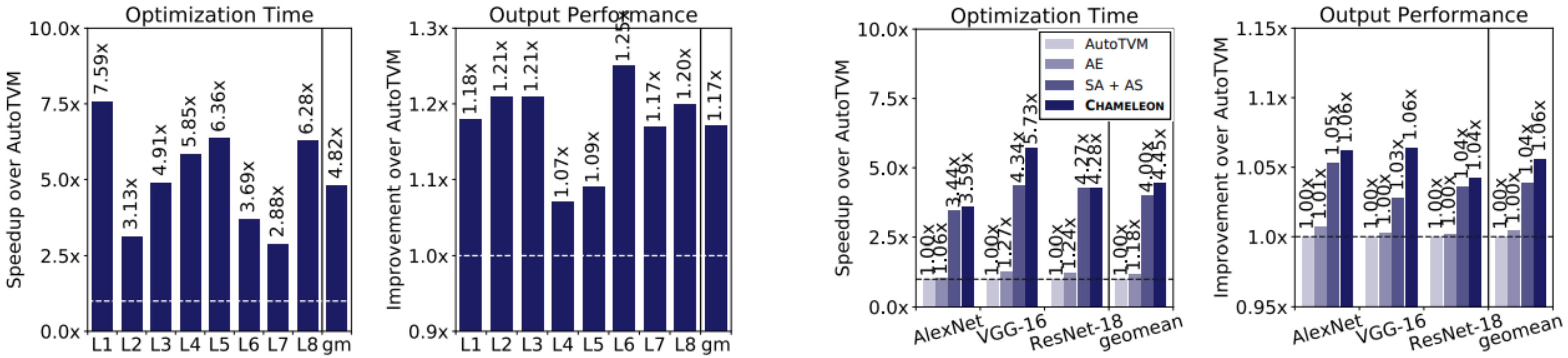


Figure 9: Layer evaluation of output performance for ResNet-18's 11th layer.

End-to-end evaluation



(a) Layer evaluation.

(b) End-to-end evaluation.

End-to-end evaluation

NETWORK	SA (AutoTVM)	AE	SA + AS	AE + AS (CHAMELEON)
AlexNet	4.31 Hours	4.06 Hours	1.25 Hours	**1.20 Hours**
VGG-16	11.18 Hours	8.82 Hours	2.57 Hours	**1.95 Hours**
ResNet-18	9.13 Hours	7.39 Hours	2.14 Hours	**2.13 Hours**

Table 2: End-to-end evaluation of the optimization time for deep networks.

NETWORK	SA (AutoTVM)	AE	SA + AS	AE + AS (CHAMELEON)
AlexNet	1.0277 ms	1.0207 ms	0.9762 ms	**0.9673 ms**
VGG-16	3.9829 ms	3.9710 ms	3.8733 ms	**3.8458 ms**
ResNet-18	1.0258 ms	0.9897 ms	0.9897 ms	**0.9831 ms**

Table 3: End-to-end evaluation of the output performance for deep networks.

Thanks