Chameleon: Adaptive Code Optimization for Expedited Deep Neural Network Compilation

(arXiv: 2001.08743)

Taehee Jeong

Chameleon is an animal that is capable of Adapting to their environments which helps them survive.

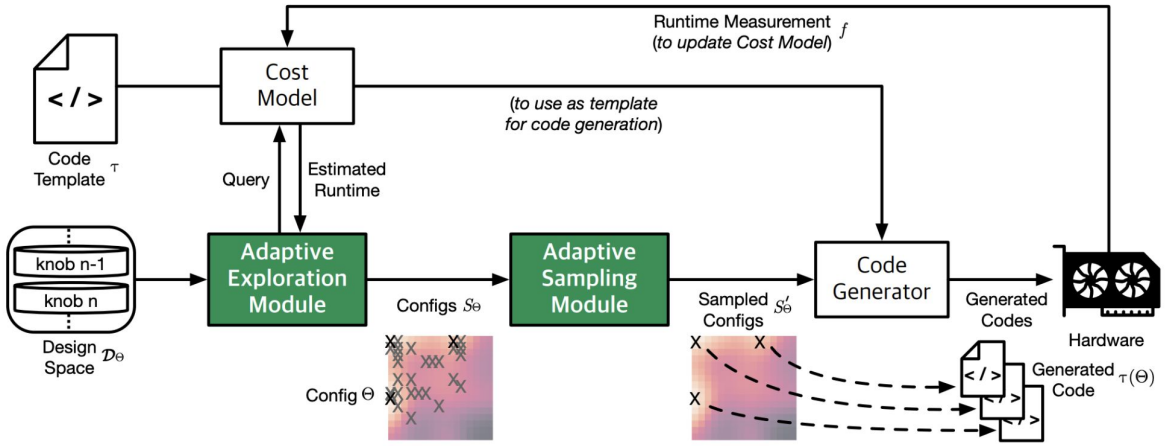


Figure 3: Overall design and compilation overview of the **CHAMELEON**.

Chameleon is an animal that is capable of Adapting to their environments which helps them survive. In our work, **CHAMELEON** is an entity that Adapts to the **variations in the design space** and the distribution of the candidate configurations, enabling expedited deep neural network compilation.

(arXiv: 2001.08743)

Model Compilation Workflow

2.1 Compilation Workflow for Deep Neural Networks

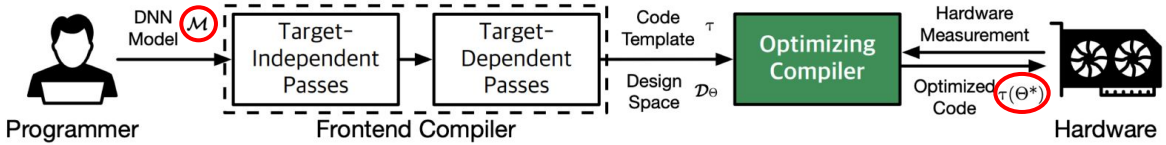


Figure 1: Overview of our model compilation workflow, and highlighted is the scope of this work.

(arXiv: 2001.08743)

Model Compilation Workflow

2.1 Compilation Workflow for Deep Neural Networks

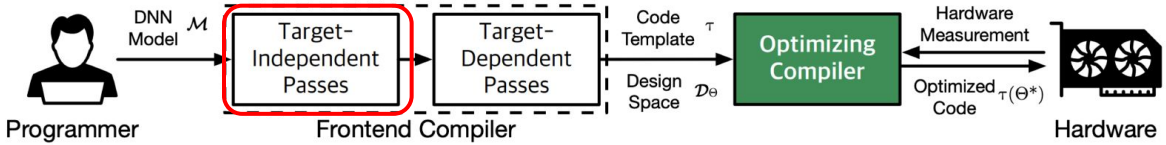
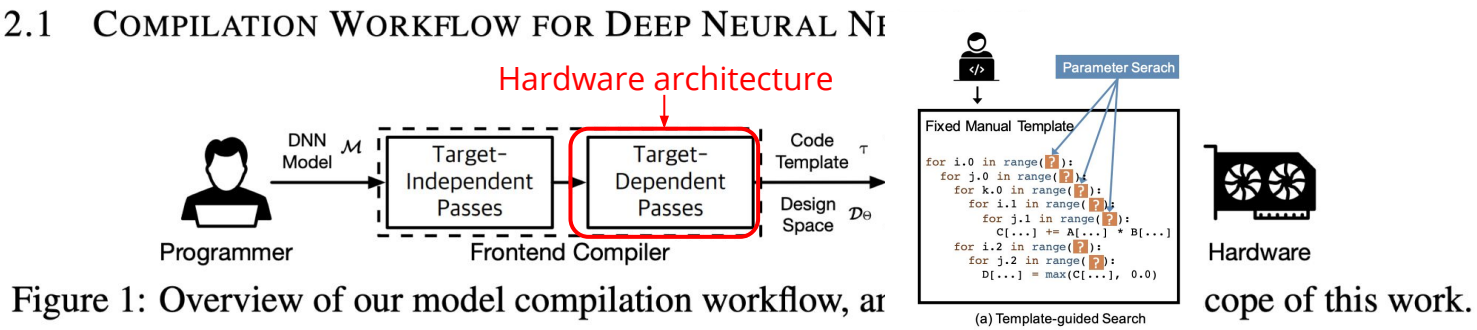


Figure 1: Overview of our model compilation workflow, and highlighted is the scope of this work.

- Target-independent pass
  - Optimizations in code model semantics + traditional compiler frontend optimizations
  - Operator fusion, data layout transformation
  - Dead code elimination, constant folding, ...

(arXiv: 2001.08743, 1805.08166)

Model Compilation Workflow



- Target-dependent pass
  - Template code(τ) generation based on target hardware architecture
  - Traditional compiler backend optimizations
  - Cost of execution time is not taken into account yet

(arXiv: 2001.08743, 1805.08166)

Model Compilation Workflow

2.1 Compilation Workflow for Deep Neural Networks

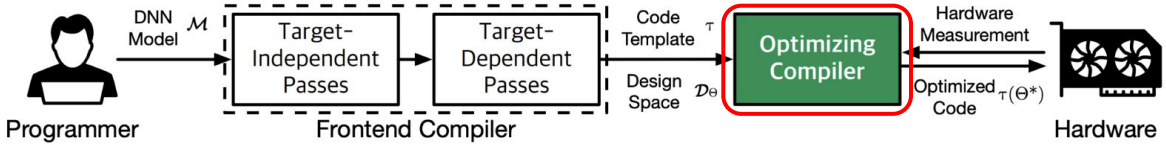


Figure 1: Overview of our model compilation workflow, and highlighted is the scope of this work.

- "Black-box" optimization pass: Optimizing Compiler
  - Generate optimal code, given design space D_Θ, find $\Theta^* \in D_\Theta$ that minimizes runtime.
  - $\tau(\Theta^*)$: optimal code given template and configurations
  - Leverages actual runtime measurements from HW to get data points

(arXiv: 2001.08743)

Optimizing Compiler: **Components and Requirements**

Configuration Exploration

- Configuration (“Knobs”): optimization parameters (tiling, loop unrolling)
- Search space should be **diverse** enough. (10^{10}? design space)
- Algorithm should be **effective** at navigating large design space
- **Reduce** costly hardware measurements

Cost Estimation

- Estimate runtime (fitness) based on code. (Same as AutoTVM)
- Avoid overfitting to specific set of configurations

Optimizing Compiler: **Previous approaches**

Configuration Exploration

- Hand-optimized libraries, compiler heuristics
- Genetic algorithm (TensorComprehension)
- Stochastic methods (Simulated annealing - AutoTVM)
- Diversity-aware exploration (AutoTVM)

Cost Estimation

- Actual HW measurements
- ML-based approach
  - XGBoost(GBT)/TreeGRU- AutoTVM, **Chameleon**
  - GNN + LSTM/Transformer - XLA

Optimizing Compiler: **Existing problems**

Configuration Exploration

- Takes too much iterations and time
- Similar configurations -> redundant calculations -> waste of computing resource
- Invalid randomly sampled configuration -> hardware failure -> reset time needed

Cost Estimation

- Greedy sampling based on cost model -> hard to explore untried configs

Chameleon: **Key contributions** in Configuration Exploration

Adaptive Exploration

- **RL** based approach - novel method in this domain
- Adapt to unseen design space

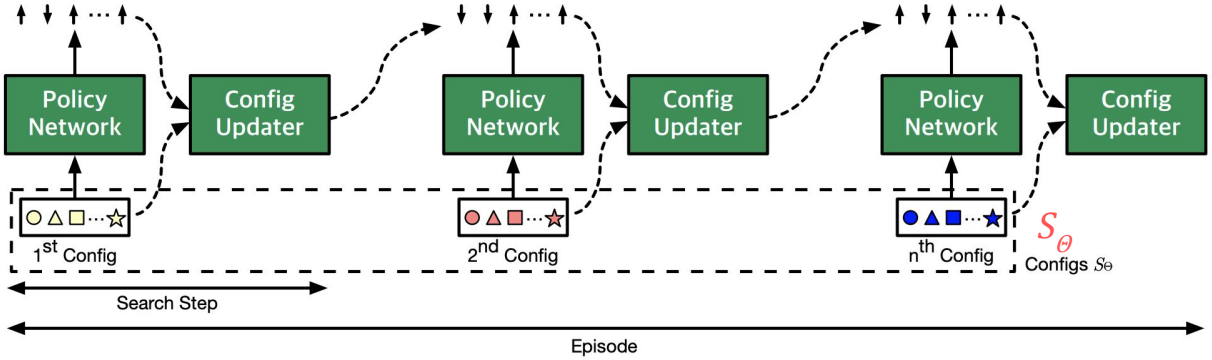
Adaptive Sampling

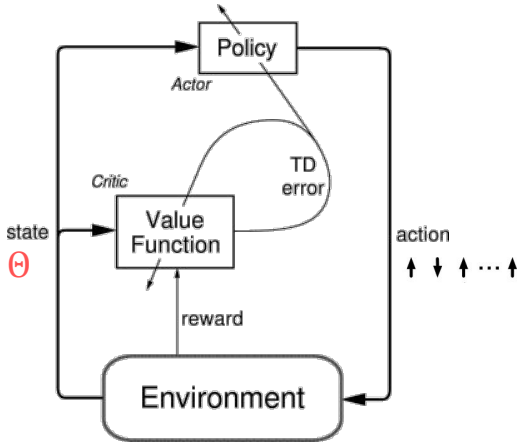
- **Clustering** of similar configurations: reduce redundant calculations
- Sample synthesis w/ **domain knowledge**: generate novel configurations that are **non-invalid**, by taking **“safe”** choices

Result:

- **4.45x** improvement in **optimizing compilation** time
- **5.56%** improvement in inference time (vs. AutoTVM)

Adaptive Exploration





Cost model

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

Actor-critic style RL

- *"actor"* **proposes config updates** (turn each "knob" up/down/stay)
- *"critic"* gets **fitness** (predicted execution time) as **reward**, and then gives "actor" a feedback.
  - Critic **learns design space** while getting reward and giving feedback

(arXiv: 2001.08743), *Reinforcement Learning: An Introduction*

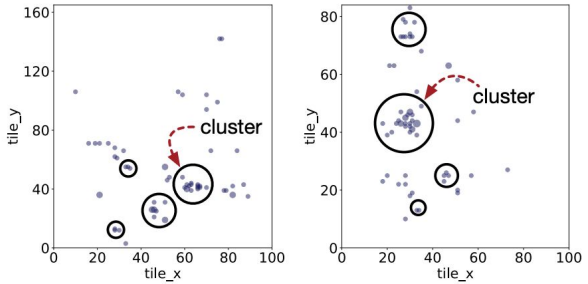
Adaptive Exploration

- **Policy networ**k suggests **good configurations** as the iteration goes
- **Value network** assists this by learning value function: *design space -> fitness*
- Used **PPO**(Proximal policy optimization) algorithm
- Performed **architecture search** to find tradeoff between performance vs network latency

Output

- S_θ: set of **all configurations** emitted from policy network each step.

Adaptive Sampling



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

Figure 5: Clusters of candidate configurations.

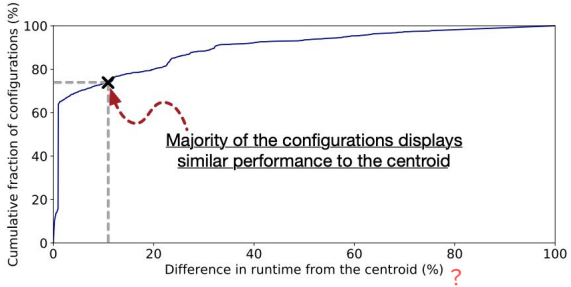


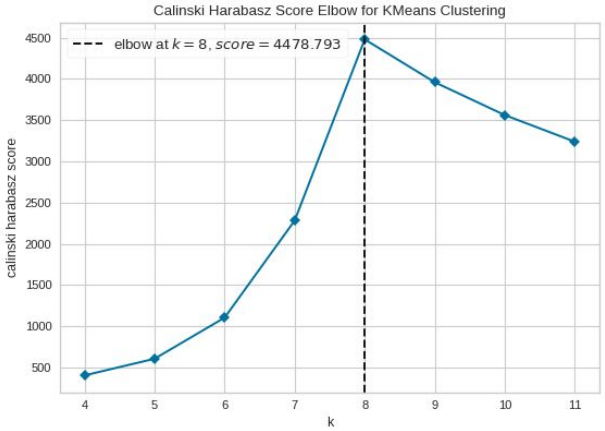
Figure 6: Cumulative Distribution Function (CDF) of the difference in runtime among the configurations in the cluster.

1. Explored configurations are **clustered**
2. For each cluster, most of configurations has **runtime really close to the centroid** configuration (80% of configs has 20% diff - Pareto?!)

(arXiv: 2001.08743)

Adaptive Sampling: **K-means clustering**

- *Threshold-based Swift Meta-Search*
  - `L2_loss * threshold > prev_loss? break`
- Determine optimal K by getting **“elbow point”** from L2 loss.
  - Choose “just enough” amount of hardware measurements



(https://www.scikit-yb.org/en/latest/api/cluster/elbow.html)

Adaptive Sampling: **Sample Synthesis**

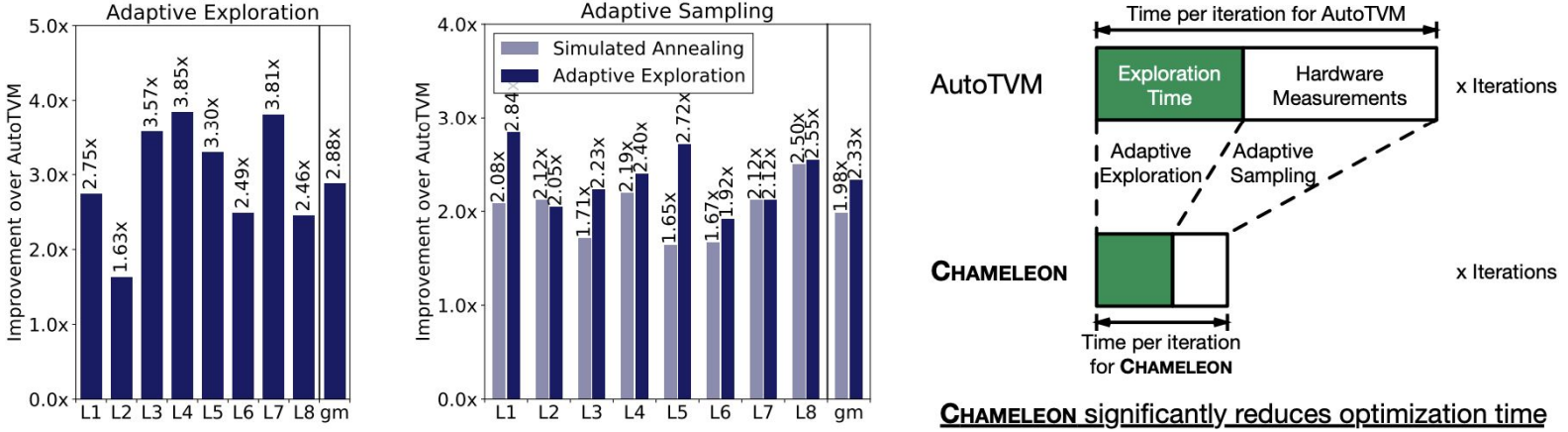
- K-means remove redundancy **within iteration**
- Overfit cost model preferes already seen configurations -> redundancy **among iterations** when greedily sampling best configurations.
  - Unvisited regions will usually have lower fitness, and thus ignored
- Randomly generated new samples would be often **invalid**. ex) too large tile -> memory error. -> **H/W reset** required
- When a config is **already explored** - replace with **mode(최빈값)** of proposed configurations(S_θ)

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 ← ∅, previous_loss ← ∞
    for k **in** range(8, 64) **do**
        new_candidates, clusters, L2_loss ← K-means.run(s_Θ, k)
        if Threshold × L2_loss ≥ previous_loss **then** break ▷ Exit loop at *knee* of loss curve
        previous_loss ← 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

Results



(a) Reduction in number of steps for convergence.

(b) Reduction in number of hardware measurements.

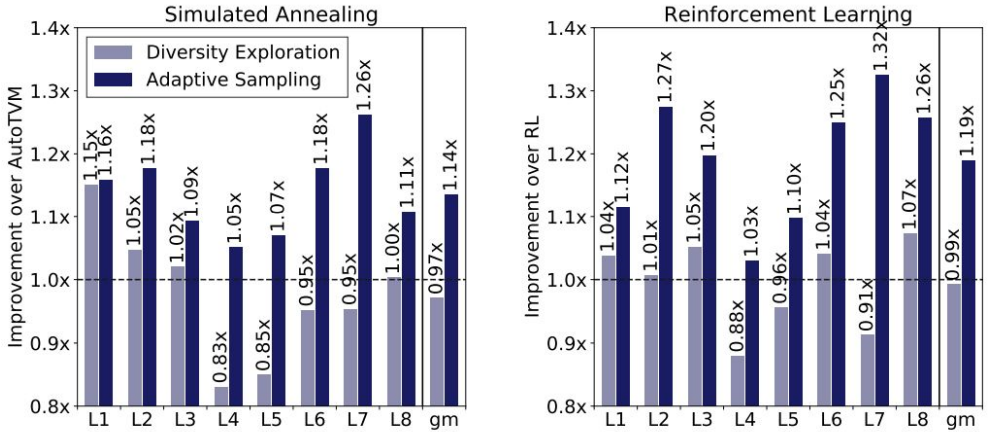
(c) Illustration of how the each component of **CHAMELEON** reduces the optimization time.

Figure 7: Component evaluation of **CHAMELEON**.

=> Adaptive sampling (obviously) reduces H/W measurements. (better w/ RL)

(arXiv: 2001.08743)

Results



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

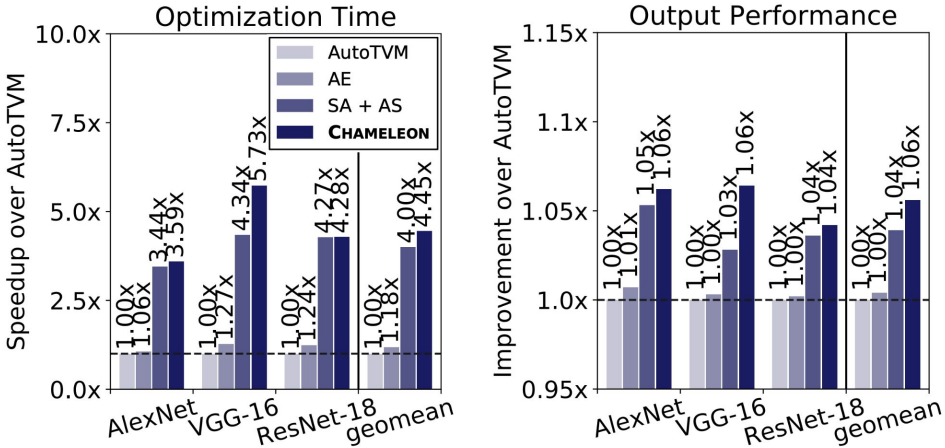
Figure 8: Comparison to AutoTVM's diversity exploration.

Compared with **AutoTVM's diversity exploration**.

Adaptive sampling works better in all cases, possibly due to domain knowledge(S_Θ)

(arXiv: 2001.08743)

Results



(b) End-to-end evaluation.

Adaptive sampling is the more effective improvement

RL also have notable impact, but less impressive than adaptive sampling.

(arXiv: 2001.08743)

Chameleon: **Key contributions** in Configuration Exploration

Adaptive Exploration

- **RL** based approach - novel method in this domain
- Adapt to unseen design space (not enough evidence?)

Adaptive Sampling

- **Clustering** of similar configurations: reduce redundant calculations
- Sample synthesis w/ **domain knowledge**: generate novel configurations that are **non-invalid**, by taking **“safe”** choices

Result:

- **4.45x** improvement in **optimizing compilation** time
- **5.56%** improvement in inference time (vs. AutoTVM)

Limitations (My opinion)

- RL seems noble, but not so much improvement (1.18x) compared to significant reduction in number of steps (2.88x)
- Runtime performance improvement is also not impressive with RL only. Value network can’t generalize its value function beyond what it has seen.
- Limited result in pretty simple convolution models. Not sure if it can be generalized to more complex models (e.g. NASNet, Inception)
- Only convolution performance is considered - maybe MatMul is already highly optimized?

THANKS!