Master's Thesis of Science in Engineering

# Microarchitecture-Aware Code Generation for Deep Learning on Single-ISA Heterogeneous Multi-Core Mobile Processors

## 단일 ISA 이기종 멀티코어 모바일 프로세서에서 딥러닝을 위한 마이크로 아키텍처를 고려한 코드생성

August 2020

Program in Intelligent Systems
Department of Transdisciplinary Studies
Graduate School of Convergence Science and Technology
Seoul National University

**Junmo Park**

# Microarchitecture-Aware Code Generation for Deep Learning on Single-ISA Heterogeneous Multi-Core Mobile Processors

지도 교수 전 동 석

이 논문을 공학석사 학위논문으로 제출함
2020 년 7 월

서울대학교 융합과학기술대학원
융합과학부 지능형융합시스템전공
박 준 모

박준모의 공학석사 학위논문을 인준함
2020 년 7 월

위 원 장 _____ 안 정 호 _____ (인)

부위원장 _____ 전 동 석 _____ (인)

위 원 _____ 박 재 흥 _____ (인)

# Abstract

While single-ISA heterogeneous multi-core processors are widely used in mobile computing, typical code generations optimize the code for a single target core, leaving it less suitable for the other cores in the processor. We present a microarchitecture-aware code generation methodology to mitigate this issue. We first suggest adopting Function-Multi-Versioning (FMV) to execute application codes utilizing a core at full capacity regardless of its microarchitecture. We also propose to add a simple but powerful backend optimization pass in the compiler to further boost the performance of Cortex-A55/75 cores. Based on these schemes, we developed an automated flow that analyzes the program and generates multiple versions of hot functions tailored to different microarchitectures. At runtime, the running core chooses an optimal version to maximize computation performance. Measurements confirm that the methodology improves the performance of Cortex-A55 and Cortex-A75 cores in Samsung's next-generation Exynos 9820 processor by 11.2% and 17.9% for CNN models, 10.9% and 4.5% for NLP models, respectively, while running TensorFlow Lite.

Keyword : Edge Computing, Function Multi-Versioning, Single-ISA Heterogeneous Multi-Core Processor

Student Number : 2018-24109

# Contents

# List of Tables

# List of Figures

# Chapter 1

# Introduction

## 1.1 Deep Learning on Single-ISA Heterogeneous Multi-Core Processors

The big.LITTLE architectures consisting of multiple cores with different microarchitectures are widely adopted in mobile environments where the tradeoff between power efficiency and performance is a critical issue. In order to fully exploit the benefit of those architectures, operating systems need to handle each program appropriately. On the Android platforms, tasks are divided into background, foreground and top-app. The type of each task determines where it should be executed; on most of big.LITTLE systems, top-app and foreground tasks can run on any core, whereas background tasks can run only on little core to save power consumption. For instance, if a task runs in the background, the scheduler will assign the task to the little core group. In addition,

many optimized schedulers such as an energy aware scheduler (EAS) have been applied to dynamically select an appropriate core in the assigned core group by scrutinizing the usage pattern and computational requirements [1]-[4]. The type of task may be changed from background to top-app or vice versa by user input, making it impossible to predict which core group the given task will be assigned to.

As deep learning algorithms have evolved in the last few years, developers have made many efforts to accelerate an inference process and improve energy efficiency in mobile environments. For instance, Google proposed multiple convolutional neural network (CNN) models optimized for real-time processing in mobile systems, accompanied by quantized models to further reduce overheads [5], [6].

The most performance-critical part of the inference using deep neural networks is a general matrix multiplication (GEMM). To accelerate GEMM operations, developers often apply hand-tuned assembly codes or intrinsic functions such as NEON in ARM architectures. For instance, two types of libraries are used for maximizing GEMM performance in the Tensorflow Lite infrastructure: 32-bit floating-point models use the Eigen library [7] based on ARM-NEON intrinsic functions, whereas 8-bit quantized models employ the gemmlowp library [8] consisting of hand-tuned assembly codes.

Compilers typically generate assembly codes with different patterns depending on the microarchitecture characteristics for maximal performance. Since the compiler-generated code pattern is optimized only for a target core, execution of the code may undergo performance degradation on the other cores in the processor with a big.LITTLE configuration.

The GCC compiler currently supports big.LITTLE -mcpu option in order to address this issue, which tries to optimize the code considering hardware characteristics of both big and little cores simultaneously. However, the resulting code still exhibits inferior performance to the codes solely optimized for each core due to architectural differences. For big.LITTLE architectures with big out-of-order and little in-order cores, traditional static compilers such as GCC generate assembly codes primarily focusing on little cores that do not have sufficient hardware resources such as a register renaming unit and a re-order buffer. As a result, significant performance degradation may be incurred when the code is executed in big cores.

## 1.2 Proposed approach

To solve the aforementioned code inefficiency problem, we propose a novel code generation methodology for single-ISA heterogeneous multi-core mobile processors aimed at deep

learning applications. We first suggest adopting Function-Multi-Versioning (FMV) to execute application codes utilizing a core at full capacity regardless of its microarchitecture. The concept of FMV was initially proposed several decades ago [9], but it has not been adopted broadly due to large code space overhead from generating multiple copies of the code. However, for mobile deep learning frameworks such as Tensorflow Lite, it is observed that a program is spending most of its time performing only few operations such as GEMM. Under this observation, we find that applying FMV to only a set of hot functions enhances performance noticeably while imposing very little code space overhead.

We also propose to add a simple but powerful backend optimization pass in the compiler to further boost the performance of smaller cores. By modifying the baseline AArch64 load-store optimization pass with the introduction of a load split pass, the code generation scheme achieves significant performance improvement for smaller cores running GEMM.

In order to apply these techniques to general systems, we also develop an automatic microarchitecture-aware code generation flow. It first analyzes a target program and selects candidate functions from the list of functions sorted by execution frequency based on the profiling result. Then the flow clones the target functions and inserts a runtime selector, resulting in target-specific assembly codes after compilation. At runtime, the selector checks

IDCODE in the Performance Monitoring Unit (PMU) register of the current core and chooses an optimal version among the clones.

The proposed code generation flow was tested on Samsung Exynos 8895 processor as well as next-generation Exynos 9820 processor. Measurements show a performance improvement of 12.7% for Exynos-M2 core in Exynos 8895, and performance boosts of 11.2% and 17.9% for Cortex-A55 and A75 cores in Exynos 9820, respectively, confirming that the proposed methodology is effective for processing deep learning models on single-ISA heterogeneous multi-core processors.

# Chapter 2

# Related works and Motivation

## 2.1 Function Multi Versioning (FMV)

FMV has been extensively studied in the last few years in order to maximize performance while suppressing code space increase. A program may run differently depending on the input data pattern due to unexpected control flows from conditional statements and varying call paths in a function. The conventional FMV technique profiles the program and generates multiple copies of the code, each optimized for specific input data pattern, through feedback-driven program optimization. This process may be repeated until an optimal point is reached [10], [11]. Some works also suggest applying the multi-versioning technique to smaller code regions such as a loop or set of basic blocks [12], [13].

Although conventional FMV schemes shorten the execution time of the program through adaptive code selection at runtime, deep learning algorithms usually exhibit very regular data patterns,

making input-dependent FMV less attractive. In addition, it still optimizes the code only for a single target core although each microarchitecture may have largely different hardware configurations.

The LLVM compiler [14], which is widely used in mobile environments, currently supports optimizations of target programs for a specific microarchitecture. If the compiler obtains target microarchitecture information using the *-mcpu* option, the compiler optimizes the code considering hardware resources of the given microarchitecture. For instance, the compiler can enforce different register allocation policies based on the information about the target microarchitecture. For in-order machines, the register numbers cannot be reused due to lack of register renaming unit in the microarchitecture. However, out-of-order machines usually have register renaming units, and hence the compiler can employ an improved instruction scheduling through reusing register numbers.

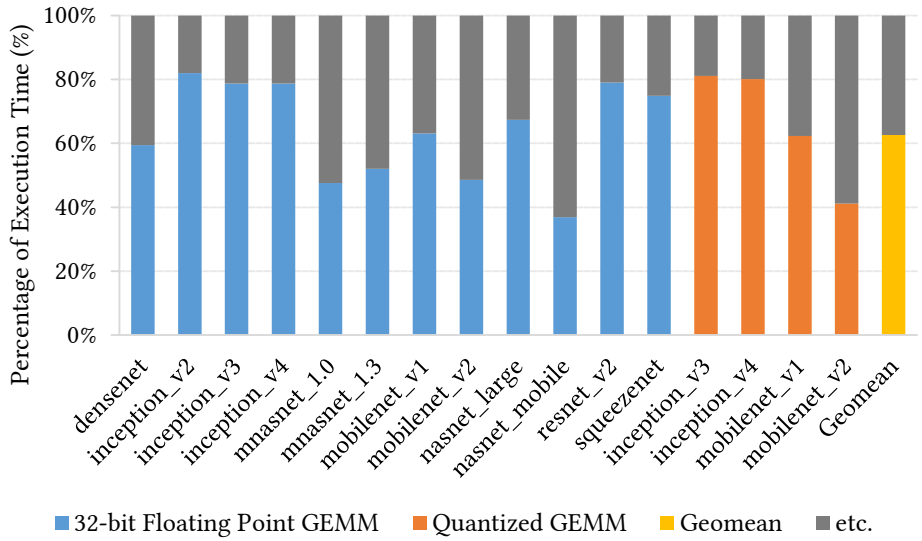


Figure 2.1 Proportion of GEMM libraries in execution time.

## 2.2 General Matrix Multiplication (GEMM)

GEMM is considered to be a key library in deep learning algorithms based on deep neural networks. For minimizing inference time on mobile devices, some prior works proposed to optimize the neural networks (e.g., Mobilenets [5] and Mnasnet [15]), whereas others try to reduce arithmetic calculation overhead by replacing costly 32-bit 5floating-point operations with 16-bit operations or even smaller fixed-point operations [6], [16].

In this work, for all analyses we use the Tensorflow Lite which is a lightweight version of TensorFlow aimed at running machine learning models on mobile and embedded devices. Tensorflow Lite supports both 32-bit floating-point and 8-bit quantized models.

| Model | | Top-1 Accuracy | Top-5 Accuracy |
|---|---|---|---|
| 32-bit Floating-point | densenet | 64.2% | 85.6% |
| | inception_resnet_v2 | 77.5% | 94.0% |
| | inception_v3 | 77.9% | 93.8% |
| | inception_v4 | 80.1% | 95.1% |
| | mnasnet_1.0 | 74.1% | 91.8% |
| | mnasnet_1.3 | 75.2% | 92.6% |
| | mobilenet_v1 | 71.0% | 89.9% |
| | mobilenet_v2 | 71.8% | 90.6% |
| | nasnet_large | 82.6% | 96.1% |
| | nasnet_mobile | 73.9% | 91.5% |
| | resnet_v2 | 76.8% | 93.6% |
| | squeezenet | 49.0% | 72.9% |
| 8-bit Quantized | inception_v3 | 77.5% | 93.7% |
| | inception_v4 | 79.5% | 93.9% |
| | mobilenet_v1 | 70.0% | 89.0% |
| | mobilenet_v2 | 70.8% | 89.9% |

Table 2. 1 Selected test models of TensorFlow Lite.

Figure 2.1 shows the percentage of GEMM libraries in execution time when different CNN models are processed using Tensorflow Lite on Exynos 9820 processor, which implies that the main bottleneck can be only few functions, and they consume most of the computation time. Specifically, the hottest function alone takes 64% and 61% of the runtime in 32-bit floating-point and 8-bit quantized models, respectively. This result suggests that if FMV is applied only to those functions, we could obtain significant performance improvement without code bloat. Note that some 32-bit floating-point models such as MobileNet [5] and MnasNet [15] have the second hottest function due to their specific neural network architectures that require depth-wise convolutions [5]. Table 2.1 displays representative models used for analyses throughout the paper, which are extracted from the hosted models

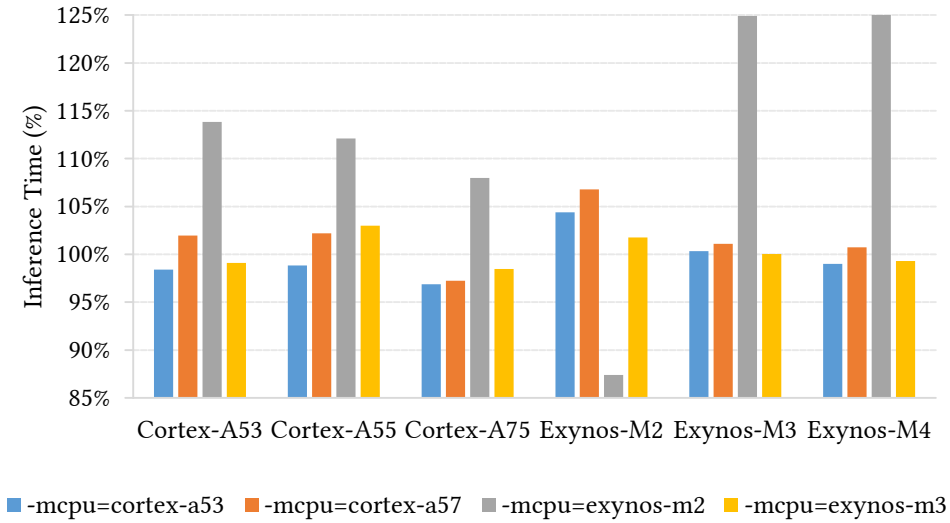


**Figure 2.2 Measured inference time on various cores with different -mcpu options used for compilation.**

and have varying neural network architectures and input sizes of 224×224×1 or larger. Other hosted models are mostly derivative of the selected models.

## 2.3 Function Multi Versioning (FMV)

Both GCC and LLVM currently support *-mcpu* option for microarchitecture-aware code generation. Without such an option, the compiler generates a generic code for generalized architecture such as AArch64 and ARMv7-a. This is equivalent to using *-mcpu=generic* option at compile time.

Figure 2.2 shows the measurement results when we compile Tensorflow Lite with different *-mcpu* options and run the models from Table 2.1 on various cores. Note that the baseline is -

*mcpu=generic* which is being used by most Android developers. Google provides a toolchain for building Android applications called Native Development Kit (NDK). Although NDK does support *-mcpu* option, in most cases developers use a generic option since there exists a wide range of SoCs with different microarchitectures. In Figure 2.2 , if we use *-mcpu=exynos-m2* option during compilation, a significant improvement of 12.6% is observed for Exynos-M2 core in Exynos 8895 processor. However, that option results in 13.8% performance degradation for Cortex-A53 core in the same Exynos 8895 processor, confirming that a simple microarchitecture-aware code generation is not effective in heterogeneous multi-core processors.

# Chapter 3

# Microarchitecture-Aware Code Generation

## 3.1 FMV Scheme for Heterogeneous Multi-Core Processors

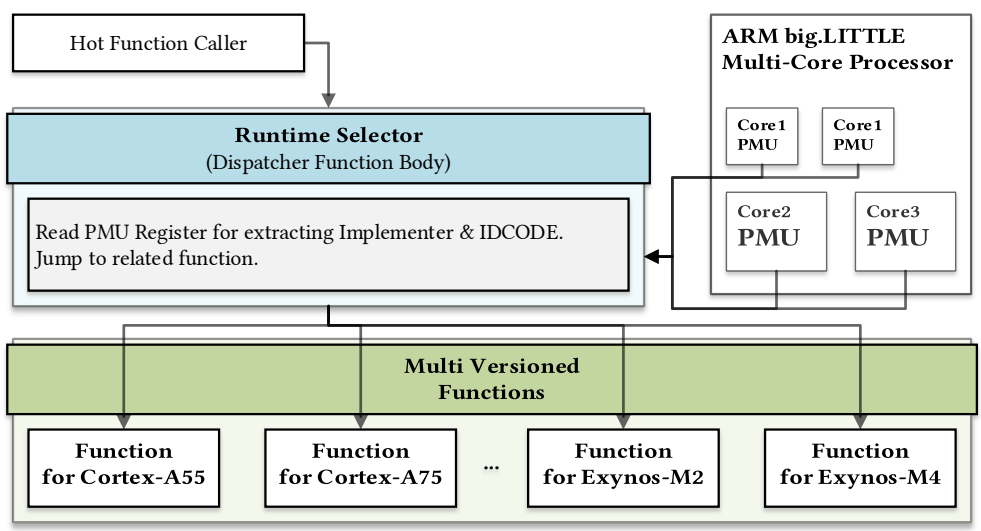


Figure 3.1 Proposed FMV scheme.

As discussed earlier, the conventional FMV scheme generates multiple versions of a given code but does not allow dynamic version change when a task migrates to a different core. In other

words, if a code is optimized for one core, the code may suffer from performance degradation on the other cores.

To resolve this issue, we propose a new FMV scheme which enables dynamic version change of target code regions at runtime so that each core can execute the version solely optimized for that core. Figure 3.1 depicts the proposed FMV scheme. Multiple versions of a target function are generated where each version is optimized differently by the compiler considering the target microarchitecture, and a runtime selector is inserted into the program.

For the proposed FMV scheme, the program must know which core it is currently located in at runtime to select a proper version. Hence, the program needs access to PMU to retrieve such information. PMU is an optional feature for some architectures such as ARMv8-A, but the feature is still strongly recommended by ARM [17] and most SoC manufacturers integrate the PMU block in the processor.

```
1.  function runtime_selector()
2.    pmu ← get_pmu_register()
3.    imp ← get_implementor_from_pmu_register(pmu)
4.    idcode ← get_idcode_from_pmu_register(pmu)
5.    case imp of
6.      ARM:
7.        case idcode of
8.          Cortex-A55:
9.            function_for_cortex-a55()
10.         Cortex-A75:
11.           function_for_cortex-a75()
12.       end case
13.     SAMSUNG:
14.       case idcode of
15.         Exynos-M2:
16.           function_for_exynos-m2()
17.         Exynos-M3:
18.           function_for_exynos-m3()
19.       end case
20.     Default:
21.       function_for_generic()
22.   end case
```

Code 3.1 Runtime selector example.

### 3.1.1 Runtime Selector

After a scheduler assigns a program to an appropriate core, the program must identify the running core for FMV. We insert a runtime selector in the program, so that the runtime selector dynamically chooses one of the versions that fits the current microarchitecture during execution. The selector can be realized with a few conditional statements followed by additional function calls. Conventional input-aware FMV schemes also employ a runtime selector in the flow, but the selector must analyze input data in detail to determine an optimal version, which translates to large processing overheads [18], [19]. However, our runtime selector chooses a version by simply looking at the implementer

(IMP) and identification (IDCODE) information retrieved from the PMU. Since the runtime selector always chooses the same version unless the program migrates to a different core, the pattern can be well predicted by the branch predictor in the core, and pipeline stalls due to branch prediction misses can be avoided. The only overhead of the runtime selector is reading PMU register and comparing it against predefined values, which does not impose noticeable performance degradation in real-world experiments as demonstrated in Section 5. The runtime selector is inserted as an inline assembly and Code 3.1 shows an example runtime selector.

### 3.1.2 Multi-Versioned Functions

In Tensorflow Lite, GEMM algorithm is implemented using two libraries: gemmlowp [8] and Eigen library [7]. Contrary to the gemmlowp library which is implemented in hand-tuned assembly, main functions of the Eigen library are written in ARM NEON intrinsics and hence can be optimized by the compiler. For those functions, the compiler selects NEON instructions and performs back-end optimizations including register allocation, instruction scheduling, and peephole optimizations such as load-store optimization. For FMV, target functions are cloned and additional function attributes stating target microarchitecture are inserted as shown in Code 3.2.

```
1.  __attribute__ ((target (“arch=cortex-a55”)))
2.  function function_for_cortex-a55 ( )
3.  __attribute__ ((target (“arch=exynos-m2”)))
4.  function function_for_exynos-m2 ( )
```

Code 3.2 Multi-versioned function example.

With the added attributes, the compiler can recognize the target microarchitecture of the function even if we use *-mcpu=generic* option or leave it empty. This information is also transferred to the compiler backend for applying target specific optimizations.

## 3.2 Load Split Optimization and LLVM-Based Unified Automatic Code Generation

While the FMV scheme allows conventional compilers to optimize functions for multiple target microarchitectures, in this section we propose an additional optimization technique for the backend of the LLVM compiler to further enhance performance. The *neon-gemm-kernel-benchmark* for the gemmlowp library provides an example hand-tuned assembly code for ARM Cortex-A55 core, where a single 128-bit load instruction is replaced with three separate instructions as shown in Code 3.3.

Cortex-A55 microarchitecture does not allow loading 128 bits at once. Hence, the core internally realizes the 128-bit load operation by loading 64 bits twice, which cannot be executed in the

same cycle although Cortex-A55 supports dual-issue. On the other hand, manually splitting load instructions as shown in Code 3.3 and placing other arithmetic instructions between them can mitigate pipeline stalls by processing a load and an arithmetic operation in the same cycle through dual-issue. We refer to this scheme as load split. Inspired by this observation, we propose a backend optimization pass for the LLVM compiler.

```
1.  “ldr  q0,    [x0]” // Split this as follows
2.  “ldr  d0,    [x0]”
3.  “ldr  x18, [x0, #8]”
4.  “ins  v0.d[1], x18”
```

Code 3.3 Load split assembly example.

### 3.2.1 LLVM Backend Passes

The LLVM compiler backend has more than 100 optimization passes, where the instruction selection and load-store optimization passes play an important role in boosting GEMM performance. Since GEMM operations require frequent matrix data load, the overall performance is largely affected by which load instructions are selected [20]. The load-store optimization pass attempts to combine instructions, searching for contiguous loads or stores that can be combined into a single instruction as depicted in Code 3.4.

```
1.    "ldr   q0, [x0]"
2.    "ldr   q1, [x0, #16]" // Combine these as follows
3.    "ldp   q0,   q1,   [x0]"
```

Code 3.4 Load Pairing Example.

Combining instructions reduces code size and lowers instruction fetch and decode overhead on the core. However, this optimization reverses the effect of load split and, therefore, this pass is turned off for Cortex-A55 and other similar microarchitectures in our flow.

### 3.2.2 Load Split Optimization Pass

We implement an additional optimization pass for load split after the load-store optimization pass. The load split optimization can be applied to any load instructions in a function, but we enforce the pass only for loops to minimize code size increase. A typical workflow is described in Algorithm 3.1.

This optimization provides maximal benefit when enough number of other independent arithmetic instructions can be placed between newly created load instructions. For example, the hottest function of Tensorflow Lite consists of 32 128-bit loads and 96 SIMD-FP-multiply-accumulate instructions, providing a good ratio between load and arithmetic operations. However, the 2nd hottest function consists of 12 128-bit loads, 4 SIMD-FP-multiply and 4 SIMD-FP-add, suggesting less performance boosting due to load split optimization.

```
1.  function load-split-pass(loop L ∈ function)
2.    for each instruction I ∈ basic blocks in L
3.      if I == 128-bit_load then
4.        Find available register r ∈ GPR64RegClass
5.        if exist(r) then
6.          Create I64-bit_load with Rsource = Rsource,I, Rdest = Rdest,I
7.          Create I64-bit_load with Rsource = Rsource,I + 8, Rdest = r
8.          Create I64-bit_mov with Rsource = r and Rdest = Rdest,I
9.          Remove I
10.       end if
11.     end if
12.   end for
```

Algorithm 3.1 Load Split Pass.

After the load split optimization pass, we need to run instruction scheduling again in order to move arithmetic instructions into the space between the split instructions. LLVM already has the PostRAScheduler feature that schedules again after register allocation. Currently, the feature is not activated for Cortex-A55/75 in LLVM 7.0, which is the most up-to-date version, and we reactivate this feature in the compilation step of the proposed flow.

### 3.2.3 Unified Automatic Microarchitecture-aware Code Generation Flow

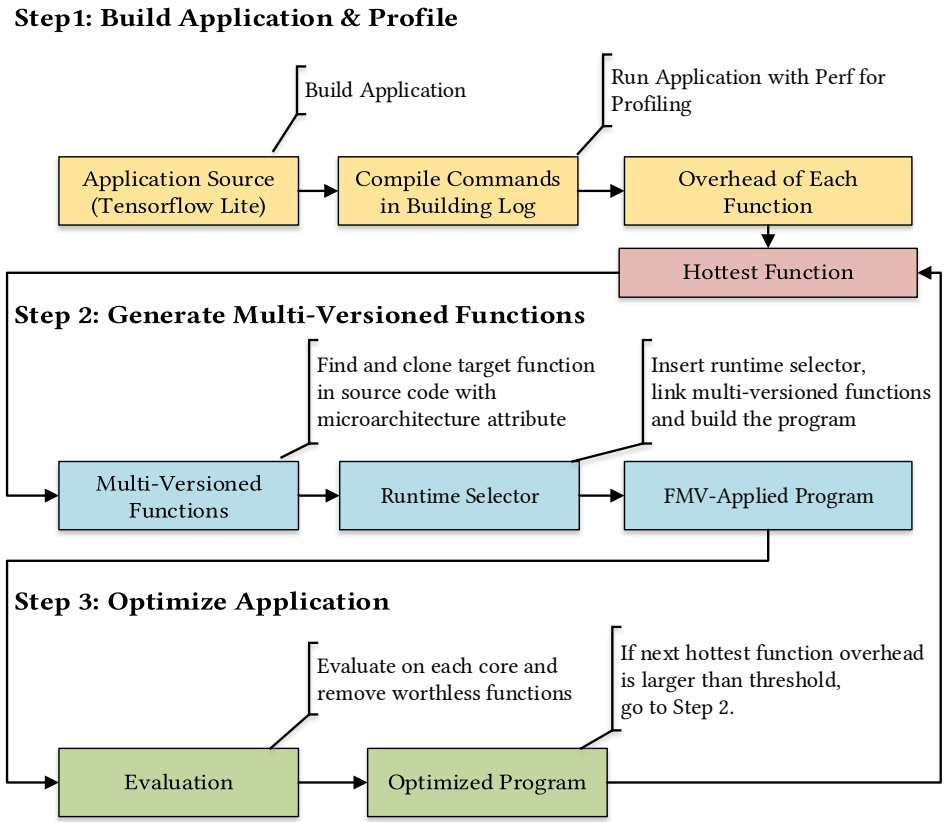


Figure 3.2 Proposed automatic code generation flow.

The proposed code generation methodology should be applied to only a subset of functions in order to suppress code size increase. We present a unified automatic code generation flow depicted in Figure 3.2. The flow first builds the target program while collecting build logs which are later used for tracking source files to be modified. Then the flow profiles the program using Linux Perf [21], which calculates the overhead of each function and sorts the

functions by their hotness. In the next step, the function at the top of the list is selected and cloned through FMV. Each clone is compiled with an optimal target microarchitecture directive as well as goes through the load split pass in the backend depending on the target microarchitecture. Finally, the resulting program is profiled on each core and the flow removes the versions that do not exhibit performance improvement. The flow continues onto the next function in the list if its share in runtime is higher than a threshold.

# Chapter 4

# Experimental Evaluation

## 4.1 Experimental Setup

In this work, we evaluate the flow using Tensorflow Lite framework which is a strong candidate for deep learning applications on mobile platforms. We run 12 floating point CNN models and 4 quantized CNN models which have different structures and use various sizes of memory. The flow was tested on two mobile processors: Samsung Exynos 8895 and next-generation Exynos 9820 processors. The Exynos 9820 features three processor clusters to realize a big.LITTLE architecture, each consisting of four Cortex-A55 cores, two Cortex-A75 cores and two Exynos-M4 cores, respectively. The Exynos 8895 has a typical ARM big.LITTLE microarchitecture and includes two clusters in total, each with four Cortex-A53 cores and four Exynos-M2 cores, respectively.

For evaluation, tasks are processed on a specific core using

*taskset* command. The frequencies of processors and memory interface are all fixed during experiments for reliable measurements. Since the two processors require different versions of Android OS, Ubuntu is used instead for experiments to align the experiment environments. We use *chroot* tool for running Ubuntu on each device and building Tensorflow Lite for Linux.

## 4.2 Floating-point CNN Models

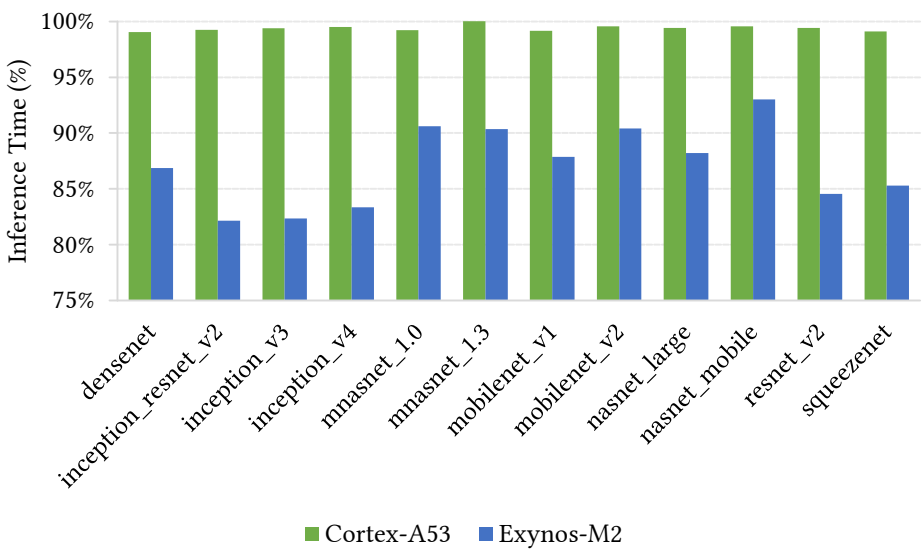


Figure 4.1 CNN performance improvement of cores in Exynos 8895 processor.

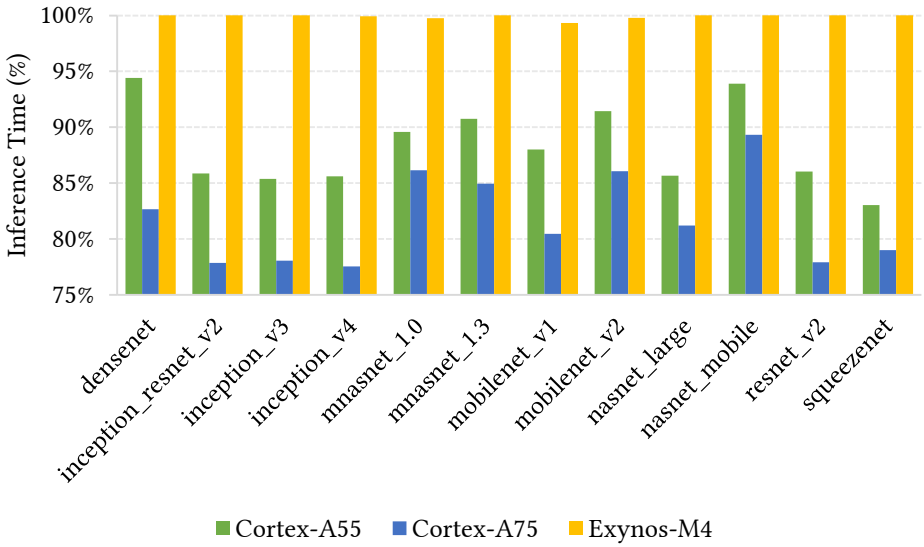


Figure 4.2 CNN performance improvement of cores in Exynos 9820 processor.

We first evaluate the proposed flow using the floating-point models in the Tensorflow Lite benchmark. Those models employ the Eigen library based on intrinsic functions. Figure 4.1 and 4.2 represent the relative runtime of each model using our code generation flow for Exynos 8895 and 9820 processors, respectively, compared to the baseline in which a generic option is used for compilation. In our flow, the FMV feature is used for all experiments in order to generate multiple versions solely optimized for each microarchitecture, and the backend optimization is enabled for Cortex-A55 and A75 cores in Exynos 9820.

For Exynos 8895, the big core (Exynos-M2) exhibits noticeable performance improvements across all models, with an average of 13.0%. For Exynos 9820, the little and middle cores

(Cortex-A55/A75) show apparent performance boosting of 11.2% and 17.9%, respectively, on average. Since the load split technique is enabled for those cores, the improvements are the combined effects of using an optimal compilation directive for each clone of a target function and applying additional backend optimization.

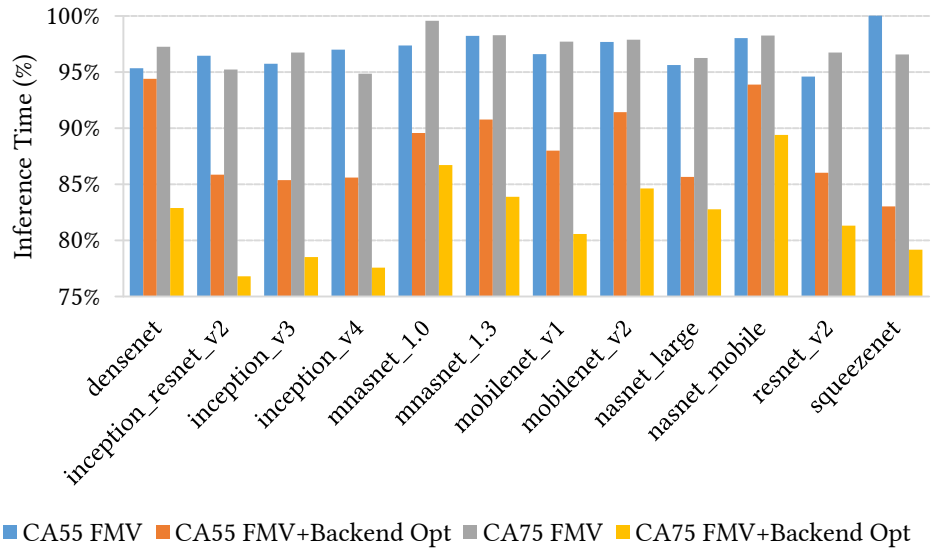


**Figure 4. 3 Effects of optimal compilation directive and load split optimization.**

Figure 4.3 displays the runtime reduction after applying compilation directives only, and after using both techniques. With compilation directives, the throughput of Cortex-A55 and A75 are improved by 3.0% and 2.9%, respectively, whereas the compiler backend optimization enhances the performance further by 8.2% and 15.0%, respectively.

In the aforementioned experiments, the flow creates 6 clones of

the target functions in order to support Cortex-A53/A55/A75 and Exynos-M2/M3/M4 cores altogether. However, the size of the generated binary increases by only 3.66%.

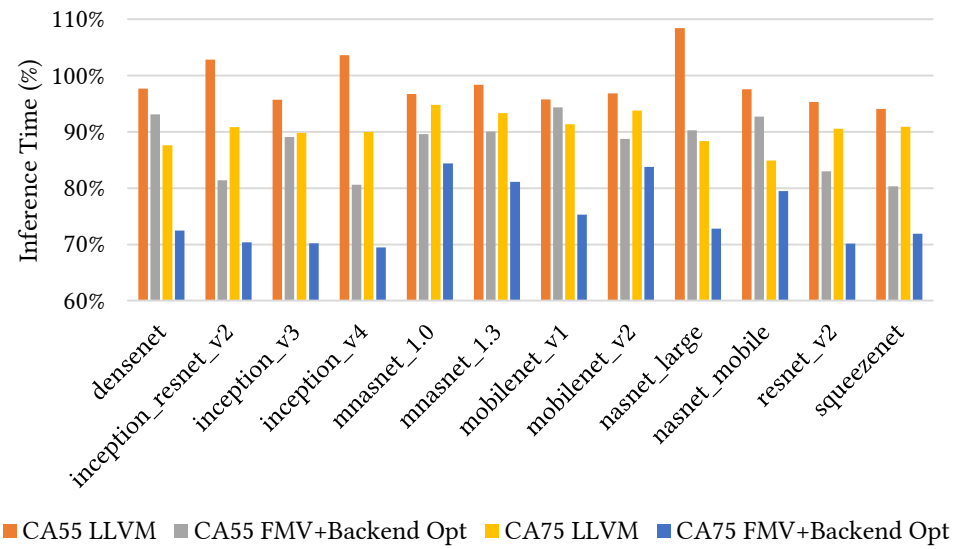


Figure 4.4 CNN performance improvements over GCC

We also compiled Tensorflow Lite using GCC 8.2 and measured the performance on Cortex-A55 and A75 cores in Exynos 9820 (Figure 4.4). On Cortex-A55, LLVM and LLVM with load-split optimization show 1.47% and 12.2% better performance than GCC, respectively, in average. On Cortex-A75, LLVM and LLVM with load-split optimization exhibit 9.49% and 24.88% better performance than GCC. We also tested GCC 7.2 for completeness, but it showed even lower performance than GCC 8.2. Google decided to deprecate GCC in their toolchain in October 2016 [22]

and hence most of the mobile applications are expected to be built by LLVM.

## 4.3 Quantized CNN Models

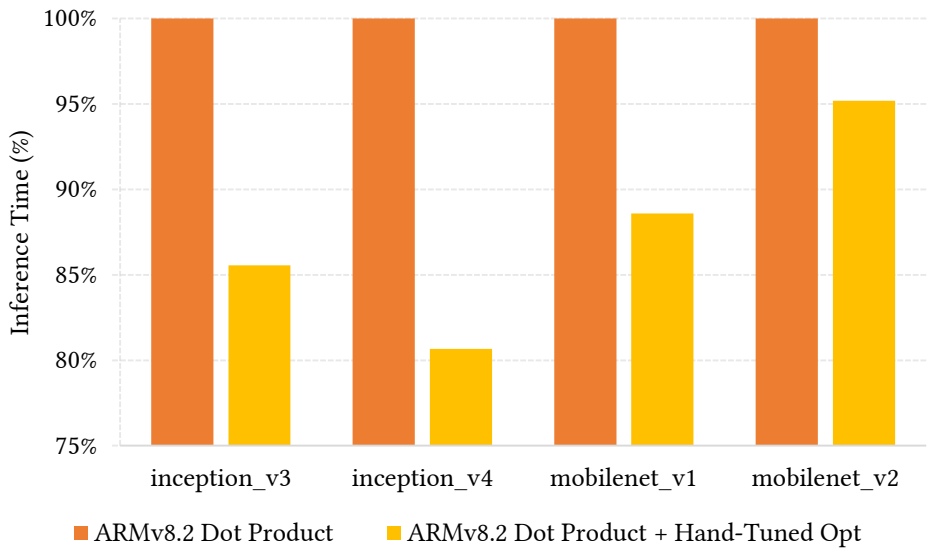


Figure 4.5 Assembly code optimization for gemmlowp library using load split technique.

The quantized models use the gemmlowp library, which relies on hand-tuned assembly codes. Therefore, the automated code generation flow cannot be directly applied. However, we can still manually adopt the proposed FMV and load split optimization techniques to improve computation performance. Figure 4.5 shows that the techniques raise the performance by 12.5% when applied to Cortex-A55 core. If the same code is used for Exynos-M4 accompanied in Exynos 9820, the performance drops by 3.5%, and this is remedied by applying the proposed FMV in the assembly,

confirming the effectiveness of our flow. Note that we modified the Tensorflow Lite build option to enable ARMv8.2 Dot Product feature.

## 4.4 Question Answering Models

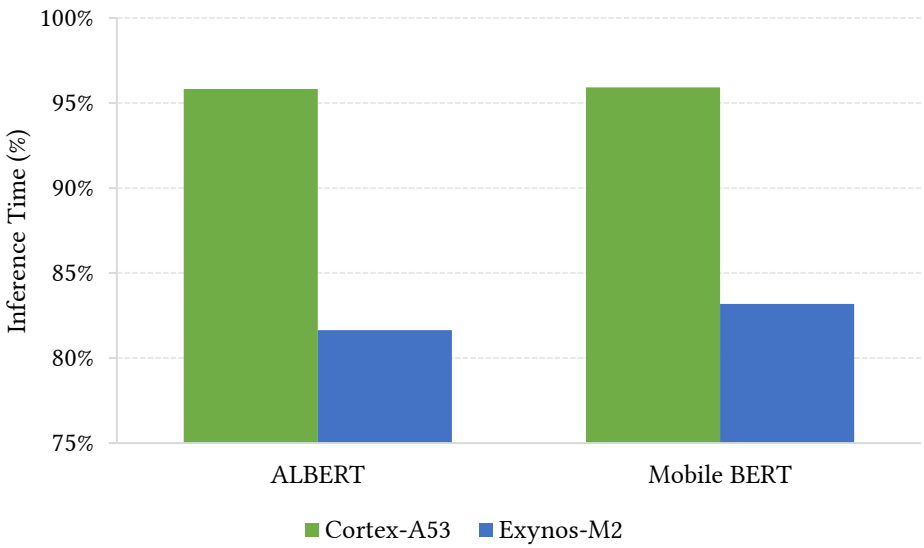


**Figure 4.6 NLP performance improvement of cores in Exynos 8895 processor.**

The question answering models can be used to build a system that can answer users' questions in natural language. It was created using a pre-trained BERT (Bidirectional Encoder Representations from Transformers) model which is fine-tuned on SQuAD (Stanford Question Answering Dataset) 1.1 [23], [24]. BERT is a method of pre-training language representations which obtains state-of-the-art results on a wide array of NLP (Natural Language Processing) tasks [24].

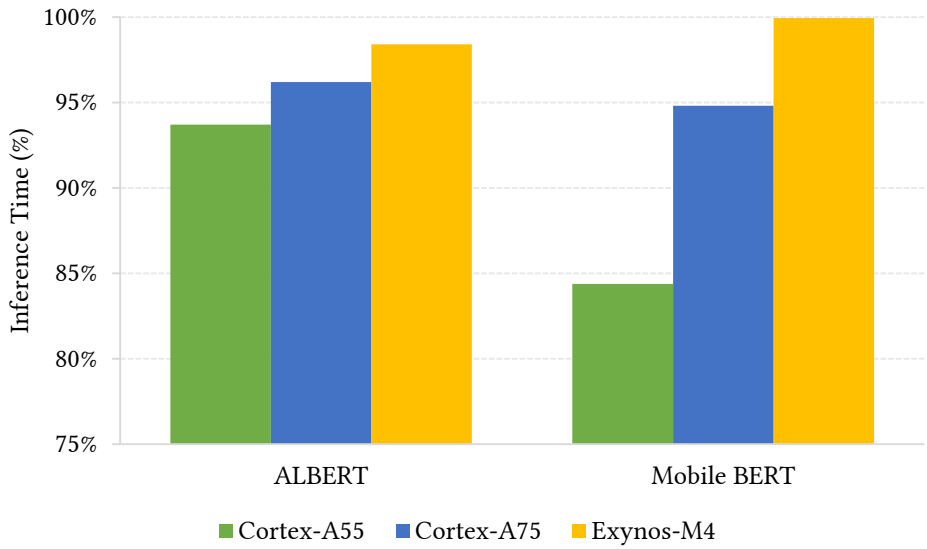


Figure 4.7 NLP performance improvement of cores in Exynos 9820 processor.

SQuAD is a reading comprehension dataset consisting of articles from Wikipedia and a set of question-answer pairs for each article. The model takes a passage and a question as input, then returns a segment of the passage that most likely answers the question [23].

Figure 4.6 and 4.7 represent the relative runtime of each model using our code generation flow for Exynos 8895 and 9820 processors, respectively, compared to the baseline in which a generic option is used for compilation. In our flow, the FMV feature is used for all experiments in order to generate multiple versions solely optimized for each microarchitecture, and the backend

optimization is enabled for Cortex-A55 and A75 cores in Exynos 9820.

For Exynos 8895, the big core (Exynos-M2) exhibits noticeable performance improvements across BERT models(A Lite version of BERT, Mobile BERT) with an average of 17.59%, whereas the little core (Cortex-A53) shows 4.13% performance improvement. For Exynos 9820, the little and middle cores (Cortex-A55/A75) show apparent performance boosting of 10.95% and 4.5%, respectively, on average.

# Chapter 5

# Conclusion

In this work, we present a microarchitecture-aware code generation technique for single-ISA heterogeneous multi-core processors. By applying FMV and introducing additional optimization pass in the compiler backend, both Exynos 8895 and the next-generation Exynos 9820 processors exhibit significant performance improvements for deep learning applications on all of 16 famous models. Although the flow was tested for two processors, the Cortex-A55 and A75 cores are expected to be used in most mobile SoCs in the near future. In order to adapt the flow in the current Android environments, it is essential to access PMU registers from user level. This is not allowed in the current OS version, but we hope this change is made in the next releases so that deep learning applications can be efficiently accelerated on mobile processors.

# Bibliography


[1] ARM and Linaro, “Energy Aware Scheduling (EAS).” [Online]. Available: https://developer.arm.com/open-source/energy-aware-scheduling.

[2] R. Kumar, D. M. Tullsen, P. Ranganathan, N. P. Jouppi, and K. I. Farkas, “Single-ISA Heterogeneous Multi-Core Architectures for Multithreaded Workload Performance,” in *ACM SIGARCH Computer Architecture News*, 2004, vol. 32, no. 2, p. 64.

[3] K. Van Craeynest, S. Akram, W. Heirman, A. Jaleel, and L. Eeckhout, “Fairness-aware scheduling on single-ISA heterogeneous multi-cores,” in *Parallel Architectures and Compilation Techniques - Conference Proceedings, PACT*, 2013, pp. 177–187.

[4] J. Corbet, “Scheduling for Android devices,” *Linux Plumbers Conference*, 2016. [Online]. Available: https://lwn.net/Articles/706374/.

[5] A. G. Howard *et al.*, “MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications,” *CoRR*, vol. abs/1704.0, 2017.

[6] B. Jacob *et al.*, “Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference,” *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2018. .

[7] G. Guennebaud, B. Jacob, and others, “Eigen v3,” 2010. [Online]. Available: http://eigen.tuxfamily.org.

[8] B. Jacob and Others, “gemmlowp: a small self-contained lowprecision GEMM library.,” 2018. [Online]. Available: https://github.com/google/gemmlowp.

[9] K. D. Cooper, M. W. Hall, and K. Kennedy, “Procedure

cloning," *Proc. 1992 Int. Conf. Comput. Lang.*, no. April 1992, pp. 96–105, 1992.
[10] D. Chen *et al.*, "Taming hardware event samples for FDO compilation," *Proc. 8th Annu. IEEE/ ACM Int. Symp. Code Gener. Optim. - CGO '10*, p. 42, 2010.
[11] D. Chen and D. X. Li, "AutoFDO : Automatic Feedback-Directed Optimization for Warehouse-Scale Applications," vol. 1, no. 212, pp. 12–23, 2016.
[12] X. Chen and S. Long, "Adaptive multi-versioning for OpenMP parallelization via machine learning," *Proc. Int. Conf. Parallel Distrib. Syst. - ICPADS*, vol. 12, pp. 907–912, 2009.
[13] K. Koukos, P. Ekemark, G. Zacharopoulos, V. Spiliopoulos, S. Kaxiras, and A. Jimborean, "Multiversioned decoupled access-execute: the key to energy-efficient compilation of general-purpose programs," *Proc. 25th Int. Conf. Compil. Constr. - CC 2016*, pp. 121–131, 2016.
[14] L. C and A. V, "Llvm-a-Compilation-Framework-for-Lifelong-Program-Analysis," in *Proceedings of the international symposium on Code generation and optimization-CGO*, 2004, no. c.
[15] M. Tan, B. Chen, R. Pang, V. Vasudevan, and Q. V. Le, "MnasNet: Platform-Aware Neural Architecture Search for Mobile," *CoRR*, vol. abs/1807.1, 2018.
[16] J. Wu, C. Leng, Y. Wang, Q. Hu, and J. Cheng, "Quantized Convolutional Neural Networks for Mobile Devices," in *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016, pp. 4820–4828.
[17] ARM, *ARM Architecture Reference Manual ARMv8, for ARMv8-A architecture profile*. 2018.
[18] K. Tian, Y. Jiang, E. Z. Zhang, and X. Shen, "An Input-centric Paradigm for Program Dynamic Optimizations," *OOPSLA '10Proceedings ACM Int. Conf. Object oriented Program. Syst. Lang. Appl.*, vol. 45, no. 10, pp. 125–139, 2010.
[19] P. Chuang, H. Chen, G. F. Hoflehner, D. M. Lavery, and W. Hsu, "Dynamic Profile Driven Code Version Selection," in *Interaction between Compilers and Computer Architecture*, 2007, pp. 74–81.
[20] X. Su, X. Liao, and J. Xue, "Automatic generation of fast BLAS3-GEMM: A portable compiler approach," *CGO 2017 - Proc. 2017 Int. Symp. Code Gener. Optim.*, pp. 122–133, 2017.
[21] "Linux Perf." [Online]. Available: https://perf.wiki.kernel.org/.
[22] Google, "NDK Revision History," 2019. [Online]. Available:

https://developer.android.com/ndk/downloads/revision_history.
[23] P. Rajpurkar, J. Zhang, K. Lopyrev, and P. Liang, “SQuAD: 100,000+ Questions for Machine Comprehension of Text,” Jun. 2016.
[24] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, “BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding,” *NAACL HLT 2019 - 2019 Conf. North Am. Chapter Assoc. Comput. Linguist. Hum. Lang. Technol. - Proc. Conf.*, vol. 1, pp. 4171–4186, Oct. 2018.

# 초 록

단일 ISA 이기종 멀티 코어 프로세서가 모바일 컴퓨팅에 널리 사용되는 반면, 코드 생성은 단일 대상 코어에 대한 최적화된 코드를 생성하기에 프로세서의 다른 코어에서는 적합하지 않을 수 있다. 본 논문에서는 이 문제를 완화하기 위해 마이크로 아키텍처를 인식할 수 있는 코드 생성 방법을 제시한다. 우선 마이크로 아키텍처와 상관없이 애플리케이션 코드를 실행하기 위해 모든 코어를 최대한 활용할 수 있도록 FMV (Function-Multi-Versioning)를 제안한다.

또한 Cortex-A55 / 75 코어의 성능을 더욱 향상시키기 위해 컴파일러에 간단하지만 강력한 백엔드 최적화 패스를 추가할 것을 제안한다. 이러한 기술을 기반으로 프로그램을 분석하고 서로 다른 마이크로 아키텍처에 맞게 여러 버전의 Function을 생성하는 Automated Flow를 개발하였다. 이를 통해 프로그램 실행 시, 실행 중인 코어는 연산 성능을 극대화하기 위해 최적 버전의 Function을 선택한다.

본 논문에서 제시한 방법론을 통해, TensorFlow Lite를 실행하는 동안 삼성의 Exynos 9820 프로세서의 Cortex-A55 및 Cortex-A75 코어에서 성능을 CNN 모델에서 11.2 % 및 17.9 %, NLP 모델에서 10.9 %, 4.5 % 향상시키는 것을 확인하였다.

**주요어 :** 엣지 컴퓨팅, 함수 멀티 버저닝, 단일 ISA 이기종 멀티 코어 프로세서

**학번 :** 2018-24109