Low-Latency BERT Inference for Heterogeneous Multi-Processor Edge Devices

Murray Kornelsen

Department of Electrical & Computer Engineering

McGill University

Montréal, Québec, Canada

April 15, 2023

A thesis presented for the degree of Masters of Science

Abstract

Popular transformer-based Deep Neural Networks (DNNs) such as Bidirectional Encoder Representations from Transformers (BERT) have achieved state-of-the-art accuracy in Natural Language Processing (NLP) tasks. However, these models often have high compute and memory requirements, resulting in high inference latency. This is especially problematic when targeting edge deployment, as edge devices have limited computational resources compared to cloud servers. Therefore, we aim to improve edge inference latency of BERT and other DNNs, which may enable improved user experience in many Artificial Intelligence (AI) enhanced applications.

In this work, we propose a heterogeneous mapping optimizer called DNN-Specific Mapping Optimizer (DNN-SMO), with the goal of utilizing multiple Processing Elements (PEs) in parallel to accelerate inference. This leads to the challenge of efficiently mapping DNN operations to heterogeneous processing elements. For a large neural network, the number of potential mappings of DNN operations to processing elements scales exponentially, necessitating the use of heuristic algorithms. DNN-SMO is based on Genetic

Algorithm (GA). Through intelligent initialization and a customized mutation operation, DNN-SMO is able to evaluate 20x fewer generations while finding superior CPU-GPU mapping configurations compared with a baseline GA. Using DNN-SMO, we find device placement configurations that achieve 15%, 24%, and 31% inference speed-up for BERT, SqueezeBERT, and InceptionV3, respectively compared to unpartitioned execution.

Furthermore, while DNN-SMO assumes 32-bit floating point data, we also propose a second optimizer which optimizes both the mapping to processing elements and quantization of DNN operations. Because quantization results in accuracy drops compared to fp32, this is a multi-objective optimization problem, resulting in an accuracy-latency Pareto optimal set of models, rather than a single optimal configuration. Using this method, we find that BERT can be accelerated by over 60% compared to fp32 inference on the CPU. Furthermore, we find up to 20% acceleration over int8 quantization while maintaining accuracy comparable to fp32.

Additionally, due to limitations in existing edge inference frameworks, especially with respect to quantization support, we developed a BERT implementation based on ARM Compute Library (ARMCL), which we call ARMCL BERT. Using ARMCL BERT, we are able to measure the performance and energy consumption of BERT models on different ARM based edge devices, on both the CPU and GPU, and at fp32, fp16, and int8 precisions. We also explore the performance of popular BERT variations, most importantly I-BERT. However, the main point of developing ARMCL BERT is to enable the multi-objective

optimization of both quantization and device assignment, which was not possible using existing frameworks.

Abrégé

Les réseaux de neurones profonds basés sur des transformateurs populaires tels que BERT ont atteint une précision de pointe dans les tâches de traitement du langage naturel (NLP). Cependant, ces modèles ont souvent des exigences de calcul et de mémoire élevées, ce qui entraîne une latence d'inférence élevée. Cela est particulièrement problématique lors du ciblage du déploiement en périphérie, car les appareils en périphérie ont des ressources de calcul limitées par rapport aux serveurs cloud. Par conséquent, nous visons à améliorer la latence d'inférence de bord de BERT et d'autres DNN, ce qui peut permettre une expérience utilisateur améliorée dans de nombreuses applications améliorées par l'IA.

Dans ce travail, nous proposons un optimiseur de cartographie hétérogène appelé DNN-SMO, dans le but d'utiliser plusieurs éléments de traitement (PE) en parallèle pour accélérer l'inférence. Cela conduit au défi de mapper efficacement les opérations DNN sur des éléments de traitement hétérogènes. Pour un grand réseau de neurones, le nombre de mappages potentiels d'opérations DNN sur des éléments de traitement évolue de manière exponentielle, ce qui nécessite l'utilisation d'algorithmes heuristiques. DNN-SMO est basé

sur des algorithmes génétiques (AG). Grâce à une initialisation intelligente et à une opération de mutation personnalisée, DNN-SMO est capable d'évaluer 20 fois moins de générations tout en trouvant des configurations de mappage CPU-GPU supérieures par rapport à un AG de base. À l'aide de DNN-SMO, nous trouvons des configurations de placement d'appareils qui atteignent une accélération de l'inférence de 15%, 24% et 31% pour BERT, SqueezeBERT et InceptionV3, respectivement.

De plus, alors que DNN-SMO suppose des données à virgule flottante 32 bits, nous proposons également un deuxième optimiseur qui optimise à la fois le mappage aux éléments de traitement et la quantification des opérations DNN. Étant donné que la quantification entraîne des baisses de précision par rapport à fp32, il s'agit d'un problème d'optimisation multi-objectifs, résultant en un ensemble optimal de modèles Pareto précision-latence, plutôt qu'une seule configuration optimale. En utilisant cette méthode, nous constatons que BERT peut être accéléré de plus de 60% par rapport à l'inférence en virgule flottante 32 bits sur le CPU. De plus, nous trouvons jusqu'à 20% d'accélération sur la quantification int8 tout en conservant une précision comparable à fp32.

De plus, en raison des limitations des cadres d'inférence de périphérie existants, en particulier en ce qui concerne la prise en charge de la quantification, nous avons développé une implémentation BERT basée sur ARM Compute Library (ARMCL), que nous appelons ARMCL BERT. En utilisant ARMCL BERT, nous sommes en mesure de mesurer les performances et la consommation d'énergie des modèles BERT sur différents appareils

périphériques basés sur ARM, à la fois sur le CPU et le GPU, et aux précisions fp32, fp16 et int8. Nous explorons également les performances des variantes BERT populaires, notamment I-BERT. Cependant, le point principal du développement d'ARMCL BERT est de permettre l'optimisation multi-objectifs de la quantification et de l'affectation des appareils, ce qui n'était pas possible avec les cadres existants.

Acknowledgements

First and foremost, I am grateful to my supervisor, Professor Warren Gross who accepted me into this program and whose ideas and knowledge are the basis for my research. I am thankful to you for giving me this opportunity and for believing in me. I am thankful for your guidance throughout this whole journey. I would like to express my gratitude to Professor Brett Meyer, Professor James Clark, and Dr. Seyyed Hasan Mozafari who have worked with me throughout my research, and have provided invaluable assistance both with technical work and with writing. I am deeply thankful for the time you have dedicated to helping me and your feedback and ideas have helped me tremendously.

I would also like to express my appreciation for my colleagues. I especially thank Hung-Yang Chang and Milad Ebrahimipour who discussed various ideas and approaches and gave me feedback in our weekly meetings. I am also grateful for the assistance of Lily Li and Negin Firouzian, who helped with setting up and operating the HiKey970 board which was crucial for my research. I also thank Charles Le, who discussed various ideas with me and inspired some of my ARM Compute Library developments.

I express my gratitude to my long-time friends Matthew McGregor and Andrew Mack, whose friendship outside of work helped keep me sane. Finally, I would like to give my sincere thanks to my family, who have always believed in me and supported me. It is only your support and love that has allowed me to complete this journey.

List of Figures

List of Tables

List of Acronyms

AI	Artificial Intelligence.
ARMCL	ARM Compute Library.
BERT	Bidirectional Encoder Representations from Transformers.
CNNs	Convolutional Neural Networks.
CPU	Central Processing Unit.
DNNs	Deep Neural Networks.
DNN-SMO	DNN-Specific Mapping Optimizer.
DSE	Design Space Exploration.
FC	Fully Connected.
FFN	Feed Forward Network.
FLOP	floating point operation.
FLOPS	floating point operations per second.
GA	Genetic Algorithm.
GELU	Gaussian Error Linear Unit.

GEMM	General Matrix-Matrix Multiply.
GPU	Graphical Processing Unit.
HMPSoCs	Heterogeneous multi-processor systems-on-chips.
IoTs	Internet of Things.
MHA	Multi-head Attention.
ML	Machine Learning.
NLP	Natural Language Processing.
NN	Neural Network.
PEs	Processing Elements.
POF	Pareto-optimal Front.
ReLU	Rectified linear unit.
SDPA	Scaled Dot-Product Attention.
SIMD	Single Input Multiple Data.
SoCs	Systems-On-Chips.
TFlite	Tensorflow Lite.

Chapter 1

Introduction

1.1 Influence of Deep Neural Networks on AI

Deep Neural Networks (DNNs) are the backbone of many Artificial Intelligence (AI) applications, and have been pushing the capabilities of AI in countless ways. For Natural Language Processing (NLP) in particular, the transformer architecture [2] has taken over for many real-world tasks. Bidirectional Encoder Representations from Transformers (BERT) models [1], which are based on the transformer architecture, offer a pre-trained generalizable language model which has been able to achieve state-of-the-art results on many downstream tasks such as question answering, natural language inference, information extraction, and semantic similarity. Moreover, BERT has been integrated into systems such as the Google Search engine to better understand queries and generate

higher-quality search results [6].

1.2 Importance of Edge Computing for AI

Currently, the number of edge devices is exploding, with the number of Internet of Things (IoTs) devices expected to reach 30 billion by 2025 [7]. In many cases, the processing these devices need to perform will be best handled by a DNN [8]. For example, voice assistants such as Amazon Alexa [9] and Google Nest [10] may require heavy language processing using DNNs. Currently, this processing would be performed by sending data to cloud servers [11]. However there are many benefits to performing the DNN computation directly on the device.

1.3 Benefits of Edge Computing

Some of the advantages that motivate research into edge computing, especially for DNNs, are the following:

1. Privacy:

   Because cloud computing requires that user data is sent to servers over the internet, users have to trust that the service provider is protecting their data and using it responsibly [12]. Edge computing, on the other hand, allows user data to remain completely local.

2. Network Dependency:

   When a device or application depends on cloud computing to function, it may stop working if a network goes down. This is of particular importance for safety critical applications, such as medical devices [13].

3. Lower Latency:

   Network operations can take a long time, depending on the amount of data sent, the speed of the network, and the distance between the server and device. In many cases, removing networking communication from a DNNs deployment can decrease latency, leading to a better experience for the end-user [14].

4. Lower Energy Use:

   Because edge devices often prioritize efficiency over computational power, moving DNNs inference to the edge may save a huge amount of energy [15]. Furthermore, sending data over the internet is another energy-intensive operation which may be reduced by edge computing.

1.4 Challenges of Edge Computing

While edge computing has clear advantages, there are also two main challenges with its implementation:

1. Memory Limitations:

Modern DNNs may require gigabytes of memory [16]. For example, the BERT-base model has 110 million parameters, requiring roughly 450 MB of memory [3].

Meanwhile, low-power Systems-On-Chips (SoCs) and modern embedded GPUs typically contain 8KB-512MB of RAM [17, 18]. Smartphones may have multiple GB of memory, but this memory may be shared between many applications running concurrently [19]. These factors significantly limit DNNs execution on edge devices.

2. Low Computational Power:

In addition to large memory requirements, modern DNNs also require a huge number of floating point operation (FLOP)s. For example, BERT-base has 110 million parameters [1] and requires 21 GFLOPs [20] for inference. Additionally, research is trending towards larger and computationally heavier networks [21]. In addition to larger parameter counts, the complexity of operations in deep neural networks (DNNs) is also growing, as operations such as multi-head attention [1, 2] are used in addition to the classic convolution and fully-connected layers. On cloud servers, DNNs can be massively accelerated by using GPUs, which are able to perform many FLOPs in parallel. However, on edge devices, the maximum speed of math operations will be much lower than a server processor, leading to much higher inference latency. For example, [3] reports a 1.69 second latency for BERT-base on a Pixel 3 smartphone. For many applications, a high inference latency will severely degrade the user experience.

1.5 Heterogeneous Edge Computing

In order to facilitate the deployment of such large complex machine learning models to edge devices, one approach is to utilize heterogeneous computing [22–26]. Modern edge devices are usually equipped with a multi-core ARM CPU and mobile GPU [19]. For high-end systems, an NPU (Neural Processing Unit) [19] may be included specifically for DNN acceleration. These heterogeneous processing elements (PEs), such as CPU, GPU, and NPU, may be used in parallel to accelerate DNN inference [27].

1.6 Challenge of Heterogeneous Mapping and Scheduling

Unfortunately, finding the optimal mapping of DNN operations to available hardware is a huge challenge, and current methods are extremely time consuming. To understand why, we look to the computational graph of BERT. Using the TVM library [28], the BERT-base graph contains ∼600 operations, where an operation usually represents a matrix multiplication or tensor reshape. Assuming each of these operations may be assigned to one of two PEs (e.g. CPU or GPU), there will be 2^{600} potential configurations. To find near-optimal configurations in such a huge design space, we need to use heuristic methods such as reinforcement learning [29], genetic algorithms [24], etc. [30]. The main drawback of these referenced methods is that they are extremely slow (e.g. requiring 20+ hours on a large cluster [29]).

1.7 Problem Statement

We aim to improve the latency of BERT inference on edge devices with heterogeneous CPU-GPU systems, which is critical for providing the best user experience in soft real-time applications. We also aim to perform this optimization as quickly as possible, enabling faster development and deployment of edge inference applications.

1.8 Past Work and Limitations

Recent research has employed two different strategies for optimizing DNN latency on heterogeneous embedded systems:

1.8.1 Mapping and Scheduling Optimization

Existing mapping and scheduling optimization methods can be divided into two main categories:

Pipeline partitioning [22–24, 31]: In pipelining methods, the computation graph of a DNN is partitioned into sub-graphs that are run sequentially on different PEs. Pipelining aims to optimize for DNN throughput rather than single-inference latency, and is not the focus of this work.

Operation-wise partitioning [14, 24, 25, 29]: In these approaches, DNN operations are partitioned such that heterogeneous PEs are used in parallel to process a single DNN input. This can be done either by processing each operation on all PEs [25] or by processing parallel branches of the computational graph on different PEs [24, 29]. These methods optimize for inference latency, which is critical for a smooth user experience in many applications. They also have the benefit of spreading a DNN across the memory of multiple processing elements, which can enable larger DNNs to be deployed on memory-constrained embedded systems. However, these methods are challenged by communication and synchronization costs between Processing Elements (PEs), as well as the previously discussed cost of searching the operation mapping massive design space.

1.8.2 Model Optimizations

There are many optimization techniques that aim to reduce model size while maintaining as much accuracy as possible. These methods can be broadly categorized into:

Knowledge Distillation [3, 32, 33]: These methods aim to transfer the knowledge of a large teacher network to a smaller student network which can have better inference latency.

Quantization [34–36] These methods utilize smaller datatypes, often integer types with 8 or fewer bits, instead of the 32 bit floating point type which is usually the default. This results in a huge memory saving and, depending on the hardware, sometimes lower latency.

There are multiple methods of converting full precision DNNs into quantized models, with varying performance in terms of accuracy and latency.

Pruning [37–40] These methods aim to remove certain weights and activations from the network which are deemed unnecessary for accurate inference. This results in sparse networks which can save both memory and computation.

1.9 Thesis Objective and Contributions

The objective of this thesis is to show how inference latency of BERT models can be improved through heterogeneous computing. We make the following contributions:

1. We develop and utilize latency models for DNNs on heterogeneous embedded systems, which allow us to quickly evaluate mapping/scheduling decisions. We use the HiKey970 [41] in this work, as it is representative of modern edge hardware.

2. Based on a genetic algorithm, we develop a DNN-specific mapping optimizer (DNN-SMO), which employs a customized initial population and mutation operation to (a) find lower-latency PE assignments, and (b) converge over 20x faster than an unmodified GA. While our DNN-Specific Mapping Optimizer (DNN-SMO) is optimized especially for BERT models, it can also be applied to other architectures.

3. We utilize DNN-SMO to, for the first time, apply heterogeneous execution to BERT

models on embedded hardware.

4. Using DNN-SMO, we find mappings with 15%, 24%, and 31% inference latency reductions for BERT, SqueezeBERT, and InceptionV3, respectively (compared to execution on a single processor). DNN-SMO finds these mappings using over 20x fewer iterations than a baseline GA.

5. In addition to DNN-SMO, which performs a single-objective optimization for minimum latency, we also build an NSGA-II based optimizer which incorporates quantization and performs a multi-objective optimization for both accuracy and latency.

6. We implement a high performance BERT model for ARM CPU and GPU using ARM Compute Library, which allows for performance measurement at fp32, fp16, and int8 precisions. We utilize this BERT implementation to generate performance measurements for the aforementioned multi-objective optimization.

1.10 Thesis Structure

This thesis is organized into seven chapters. Chapter 2 provides a comprehensive literature review of existing methodologies, their limitations, and potential directions we could include in future work. Chapter 3 reviews the BERT-base model and software frameworks we utilize, such as ARMCL and TVM. Chapter 4 describes the proposed methodology, and Chapter 5 details the experimental setup, both hardware and software. Finally, Chapter 6 illustrates

our experiments and results, and Chapter 7 closes with a conclusion and suggestions for future work.

Chapter 2

Literature Review

There exists a large number of works with the goal of improving DNNs latency, accuracy, and throughput. Past DNNs optimization work can be divided into two main categories: mapping and scheduling optimization, and DNNs optimization.

2.1 DNN Mapping and Scheduling Optimization

Mapping and scheduling optimization focuses on maximizing the utilization of available hardware resources. This can be accomplished in various ways, which we discuss below, but the overall goal is usually to assign tasks to different processing elements to be executed in parallel. Ideally, this enable a significant improvement in latency and/or throughput for a computational task, such as a neural network.

2.1.1 General

The problem of scheduling acyclic task graphs on heterogeneous systems has been studied for nearly two decades [42, 43]. For instance, [43] explores different mapping and scheduling alternatives and intelligently limits unfeasible solutions, while [42] ranks tasks based on their average execution time on all processing elements (PE), attempting to start the highest priority tasks as early as possible, using a greedy algorithm. To contrast, our method utilizes a genetic algorithm for optimization, as well as taking DNN-specific knowledge into account to improve the search process. More recent work [44] applies ideas very similar to ours for scheduling generic tasks on a heterogeneous system. Generally, they incorporate schedulability analysis and task latency information to build an improved genetic algorithm (ImGA). Like us, [44] uses domain-specific information to guide the genetic algorithm, however our work is the first to do so in the context of DNNs, leading to fundamentally different modifications to the GA.

2.1.2 Layer-wise & Pipeline Partitioning

There is a large body of work [14, 22–24, 31] which looks at pipelining of multiple DNN inferences: In these methods, a DNN computational graph is divided into sequential sub-graphs, each of which can be assigned to the (available) PE that executes it fastest. In some works [22, 23, 31], this is combined with pipelining to optimize processing element utilization, leading to higher DNN throughput. This method is particularly useful when

memory constraints prevent an entire graph from fitting on a single PE. As a representative example, [22] handles the common case of an ARM big.LITTLE SoC. They break Convolutional Neural Networks (CNNs) into consecutive segments, enabling multiple inferences to be pipelined between the two CPU clusters. As a result of reduced inter-cluster communications, they achieve significant throughput improvements. However, for single-inference latency (which is our focus in this work), the lack of parallelism between PEs means latency improvements will be limited, especially on embedded systems where CPU and GPU performance are similar [25].

2.1.3 Operation-wise Partitioning

There are also many works [24, 29, 45] which deal with single inference on heterogeneous systems: In these methods, the DNN operations are individually assigned to heterogeneous PEs, with the goal of parallelizing branches of the computational graph. Compared to the low-level operator partitioning methods, this may decrease synchronization overhead, as each processor is working on independent data. The work most similar to ours is [24], which schedules DNNs on heterogeneous embedded systems at the operation level. They use a genetic algorithm to optimize popular convolutional networks. However, their genetic algorithm is not provided with any DNN-specific information. While we also use a genetic algorithm for optimization, we demonstrate that search efficiency can be greatly improved by intelligently guiding the optimizer.

2.1.4 Application/Task-level Parallelism

Lastly, there are many works [24, 26, 46] that deal with higher level parallelism, which we call application-level or task-level: In these methods, the case of multiple DNN applications running on the same (heterogeneous) system is considered. In this case, it may seem simplest to assign each application to a single PE. However, if operations in the individual DNNs favor different processors, it may be possible to share all PEs between applications. This requires complex optimization methods, especially when strict timing deadlines are involved. For a specific example, [26] considers multiple DNNs on a CPU/GPU system. They perform layer-wise partitioning on all the networks, and then develop a method to assign layers to processing elements to maximize schedulability. This level of parallelism is orthogonal to our work.

2.2 DNNs Optimization

While mapping and scheduling optimization focuses on optimizing the utilization of hardware resources, another approach is to modify DNNs to minimize memory consumption and/or flops without a significant decrease in accuracy. These optimization techniques can be categorized into: knowledge distillation, quantization, pruning, and efficient neural architecture design. Here, we describe some previous works that have applied these techniques to BERT and NLP models, but they apply equally to CNNs.

2.2.1 Knowledge Distillation

Knowledge distillation is the process of transferring knowledge from a larger teacher network to a more efficient student network. In most works, the student network is similar in structure to the teacher network but with smaller parameter count. The goal is to create a student network that requires less memory and fewer flops while maintaining most of the teacher network's accuracy. As a result, the student network is able to have significantly lower latency than the teacher, while the distillation process is more efficient that training from scratch. Related works include SqueezeBERT [3], DistilBERT [32], MobileBERT [33], and DynaBERT [47].

2.2.2 Efficient Neural Architectures

Another approach to reducing memory and floating point operations per second (FLOPS) is to significantly modify the network's structure for lower complexity and higher computation efficiency. Some examples include SqueezeBERT [3] uses grouped operations to replace default BERT operations, MobileBERT [33] which uses a bottleneck architecture, and ALBERT [48] which uses parameter sharing.

2.2.3 Pruning

Pruning involves the introduction of sparsity into networks. This is accomplished by removing certain connections of weights or activations, often by setting a weight to 0. One

technique is to remove values from the network which are below a certain threshold, which results in a sparse model with fewer computations and less memory required to execute. Some example pruning works are [37–40]

2.2.4 Quantization

Quantization is a technique in which the parameters of a neural network are converted to a smaller data type. Currently, most DNNs utilize 32 bit floating point values during both training and inference. However, experiments have found that DNN parameters can be reduced in size, sometimes even to 1 bit, without losing much accuracy. By reducing the size of DNN parameters, the memory required for inference is massively reduced. Additionally, depending on hardware and software support, quantization can also significantly reduce latency. However, most GPUs are optimized to perform floating point math, which can limit quantization deployment. Example works include Q8BERT [49], I-BERT [34], and Qualcomm's Transformer Quantization [35]

2.3 Combining Mapping and Scheduling with DNN Optimization

LaLaRAND [50] combines quantization with layer-level mapping and scheduling optimization to improve the schedulability of real-time DNN tasks. LaLaRAND achieves on average 80%

higher schedulability ratio compared to vanilla PyTorch with GPU.

For single inference latency, one previous work is ulayer [27], which splits convolution computation between the CPU and GPU by having each PE process a disjoint set of the output channels. Additionally, they find that int8 is optimal for latency on the CPU while fp16 is optimal on the GPU, and quantize layers accordingly for minimum total latency. While this method avoids the problem of operation-level mapping optimization, it suffers from high communication and synchronization costs, especially for fully-connected layers. Additionally, by consuming both CPU and GPU for every operation, their method under-utilizes operator-level parallelism, which is more efficient as it requires less synchronization. For this reason, we focus on exploiting parallelism at the operator level.

Chapter 3

Background

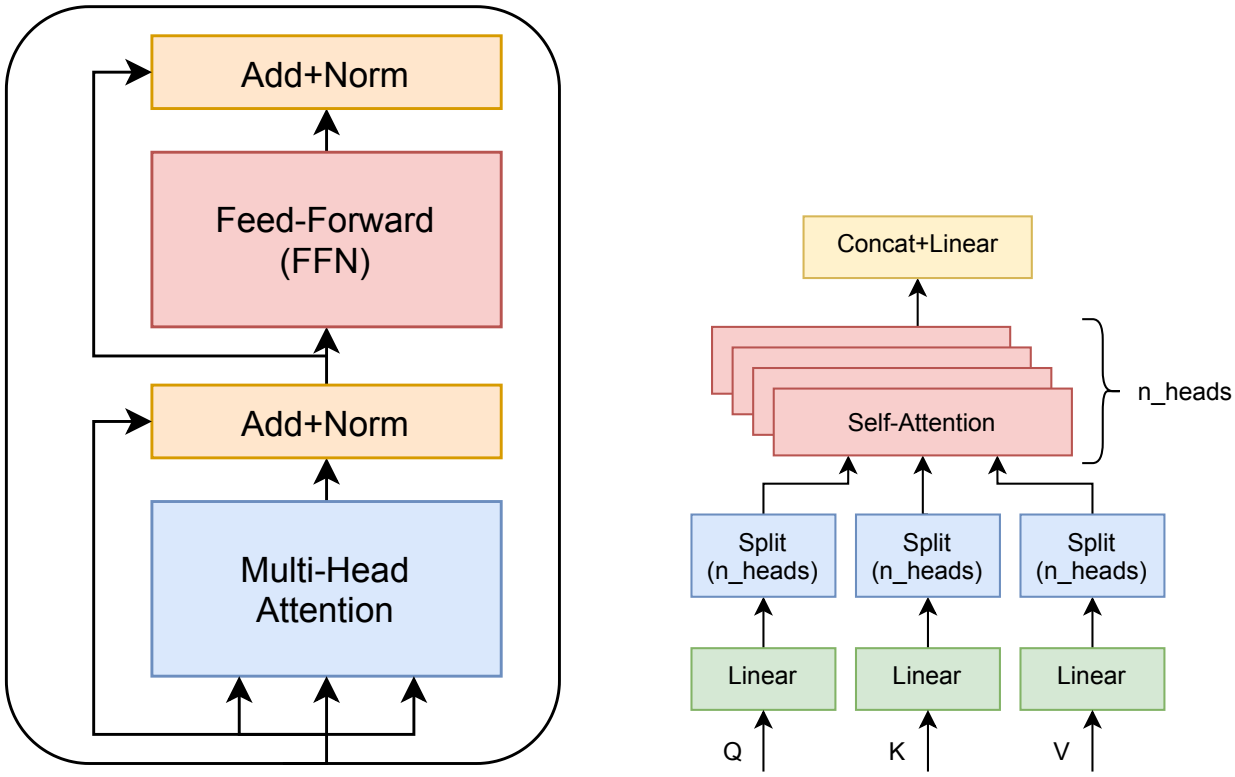
3.1 BERT

BERT [1] is a pre-trained language model, meaning that it provides a general model that can be quickly fine-tuned to perform specific NLP tasks. Essentially, by performing a vast and expensive pre-training process, a huge amount of time is saved when generating models for specific tasks. For BERT, the pre-training was done using BooksCorpus [51] and English Wikipedia text. From the raw text data, the authors created two unsupervised tasks designed to instill BERT with general language capabilities. They call these tasks masked language model (MLM) and next sentence prediction (NSP). In MLM, random words are masked out of an input sentence and the BERT model is trained to predict the missing word. In NSP, BERT is provided with two sentences and must determine whether the second sentence

followed the first. The exact methods of generating these training tasks are detailed in [1]. In theory, the MLM task taught BERT how words are related to each other within a sentence, while NSP taught it how two sentences could relate to each other. This combination resulted in a model able to handle a wide variety of NLP tasks, such as those in the GLUE benchmark (section 3.1.4).

To fine-tune the pre-trained BERT model, an output layer is appended which can be trained on a much smaller dataset for the specific task. For example, a common task is to determine if a sentence is linguistically acceptable or not (CoLA [52]). Since this task has two output classes (acceptable / not acceptable), a linear layer will be appended to the BERT model that brings its output down to a vector of length 2. This extra layer can then be trained using a standard loss function (often cross-entropy loss). As a result of BERT's pre-training process, [1] found that they were able to achieve state of the art accuracy with minimal fine-tuning time, even with small task-specific datasets.

BERT is an example of a transformer [2] architecture, consisting of an embedding layer followed by a stack of transformer encoders. Each encoder consists of two main operations: Multi-head Attention (MHA) and Feed Forward Network (FFN). The structure of these encoders is illustrated in Figure 3.1a. While the FFN section is a straightforward sequence of two linear layers, the MHA is more complicated and is illustrated in Figure 3.1b. Notably, the encoders used in BERT differ slightly from the original transformer work, replacing the Rectified linear unit (ReLU) activation function with Gaussian Error Linear Unit (GELU)



(a) Transformer Encoder **(b)** Multi-head Attention

Figure 3.1: BERT Architecture

[1, 53].

The common architectural parameters of BERT are as follows:

1. Sequence Length (S): The number of input samples that the model can process simultaneously.

2. Hidden Size (H): The width of token embeddings and the size of encoder inputs and outputs.

3. Intermediate Size (I): The width used between the two layers of the FFN.

4. Hidden Layers (L): The number of transformer blocks (encoders) in the model.

In BERT-base, there are 12 transformer encoders, each of which uses a hidden size of 768 and an intermediate size of 3072. While sequence length can be set depending on the application, we utilize a common length of 128 for consistency.

3.1.1 Input Embedding Layer

The functionality of BERT is designed to handle a wide variety of downstream tasks. To achieve this, its input representation is able to unambiguously represent both a single sentence and a pair of sentences (e.g., question and answer) in one token sequence. Figure 3.2 visualizes BERT's input embeddings, which are the sum of the token embeddings, the segment embeddings and the position embeddings.

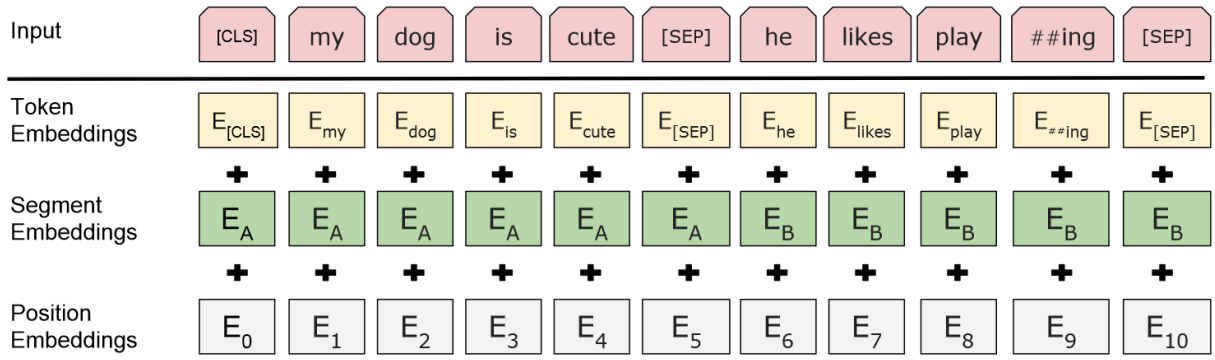


Figure 3.2: BERT input representation [1]

Firstly, the input sequence is padded to a fixed sequence length S, and each word in an input sequence is embedded into a token using WordPiece embeddings [54] with a 30k token vocabulary, wherein the embedding size is denoted as H. Secondly, BERT utilizes segment embedding to pack a single sentence or a pair of sentences (e.g., question, answer) together as one token sequence. Segment embedding is a learned embedding that indicates whether a token in the input sequence belongs to the first sentence or the second one. After segment embedding, positional embedding is applied to encode the position of the token in the input sequence.

The input to the encoder is thus the summation of the token, segment, and position embeddings. Note that the dimensions of the output of the embedding block depend on the embedding size H of a token and the number of tokens in sequence S.

3.1.2 Multi-Head Attention (MHA)

Within the Transformer architecture of [2], the main innovation is the use of MHA. Previously, sequence modeling problems had been dominated by recurrent neural networks. These had a few disadvantages, mainly that they could not be parallelized effectively and that relationships between distant tokens could be lost. By utilizing an attention function and eliminating recurrence, [2] addressed both of these issues and their work revolutionized NLP.

In general, an attention function defines a mapping between a query (Q) and a set of key-value (K-V) pairs to an output, all of these being vectors. By drawing Q, K, and V from the same input sequence, we create a special case called self-attention. The goal of self-attention is to relate different positions of a single sequence, which is critical for language comprehension tasks. [2] also extends simple attention to MHA, which computes multiple self-attention functions on the token sequence, each of which may encode a different relationship between tokens.

While many attention functions are possible, [2] introduces Scaled Dot-Product Attention (SDPA), which uses a dot-product to measure the similarity between query (Q) and key (K). MHA then utilizes multiple SDPA blocks, enabling different relationships between input tokens to be encoded. The number of SDPA blocks is called the number of attention heads.

In more detail, each attention head determines how strongly each pair of tokens in an input sequence are related to each other. This is determined by softmax-normalized dot

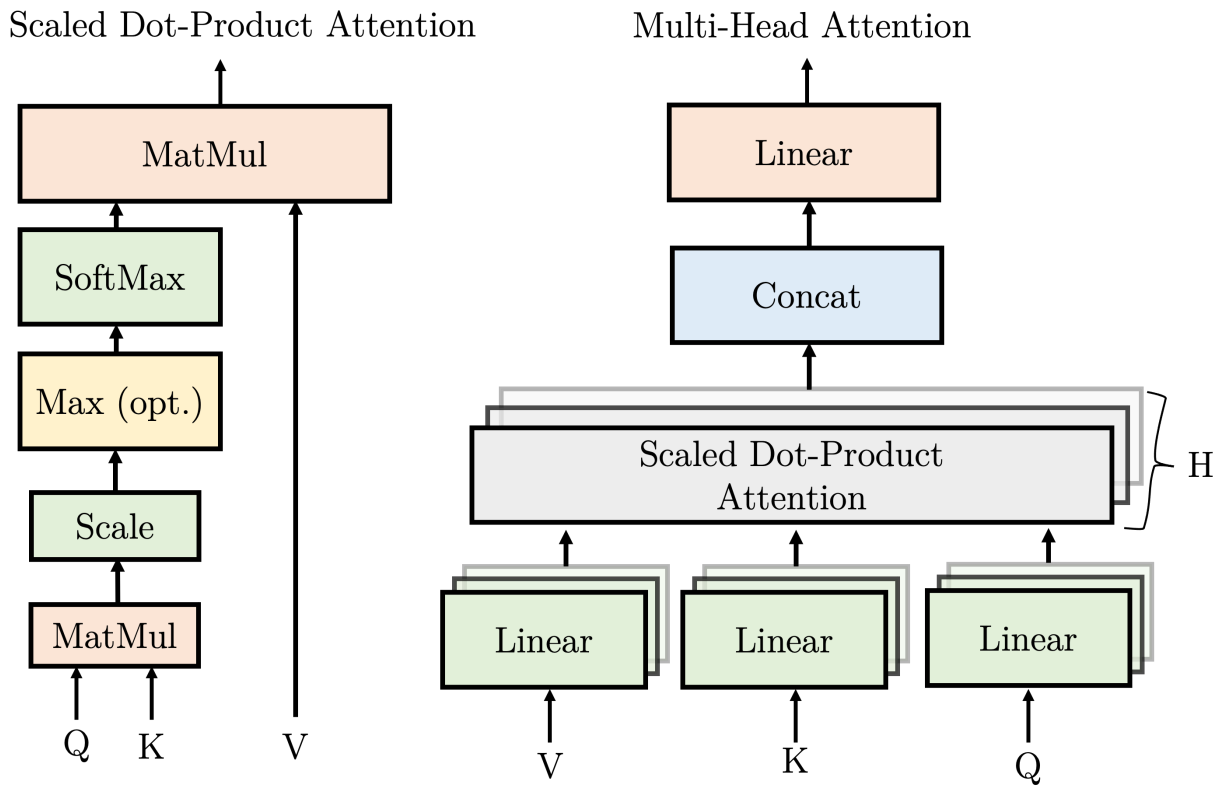


Figure 3.3: Scaled Dot-Product Attention and Multi-Head Attention, where Multi-Head Attention combines several SDPA operations. The number of SDPA operations is defined by H, the number of attention heads [2].

products between the Q and K vectors. The softmax output is referred to as attention weights, which represent how much a token attends to each token in the input sequence. The output of an attention head is the a weighted sum of the V vectors, using the previously computed attention weights. Attention can be expressed as follows:

$$Attention(Q, K, V) = \ softmax(\frac{QK^T}{\sqrt{d_k}})V \tag{3.1}$$

where d_k is the dimension of the keys and $\frac{1}{\sqrt{d_k}}$ acts as a scaling factor to counteract small gradients in softmax function when d_k is large.

While a single head is capable of generating the attention weights for each pair of tokens, the transformer architecture [2] extends this to MHA because it allows the model to handle information from different representation subspaces at different positions. MHA is defined by concatenating the attention heads as in Equation 3.2:

$$\begin{aligned} MultiHead(Q,K,V) = \ & Concat(head_1,...,head_h)W^O \\ where\ head_i = \ & Attention(QW_i^Q, KW_i^K, VW_i^V) \end{aligned} \tag{3.2}$$

$$where\ W_i^Q \in \mathbb{R}^{d_{model} \times d_q}, W_i^K \in \mathbb{R}^{d_{model} \times d_k}, W_i^V \in \mathbb{R}^{d_{model} \times d_v}, W^O \in \mathbb{R}^{d_{model} \times hd_v}$$

Note that there are four learnable weight matrices involved in MHA, denoted as $W_i^Q, W_i^K, W_i^V,\ and\ W_i^O$. Additionally, observe in Figure 3.3 that there is no data dependency between the linear and scaled dot-product attention layers of MHA. As such, these can be calculated in parallel across the H attention heads.

3.1.3 Feed-Forward Network

The FFN consists of two position-wise Fully Connected (FC) layers as follows:

$$FFN(x) = \ max(0, xW_1 + b_1)W_2 + b_2 \tag{3.3}$$

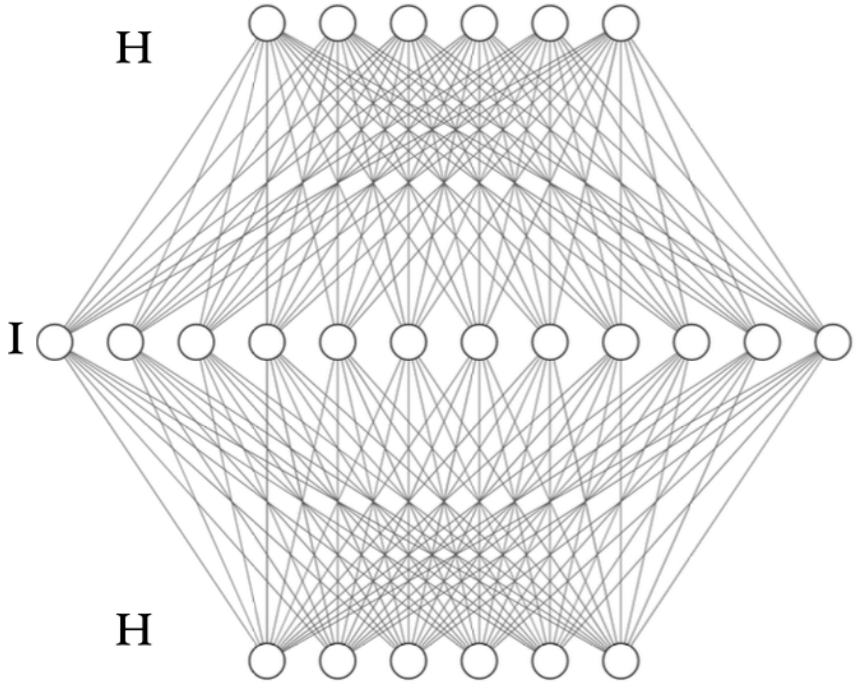


Figure 3.4: A two-layer FFN. Input with width H is projected to width I and then down-projected back to H

The first layer projects the input from the hidden size H to the intermediate size I, and the second layer down-projects it back to H as depicted in Figure 3.4. After the first projection, an activation function is applied. In Equation 3.3, the FFN is presented with the ReLU activation function. However, this choice can vary, with works such as BERT choosing to replace this with GELU for example.

To complete the transformer block, each of MHA and FFN has a residual connection around it followed by a layer normalization. Finally, the classifier block is added after the final transformer block and depends on the downstream NLP task. The pre-trained

contextual representations that BERT offers are available at the output of the last encoder which makes it relatively simple to train a classifier for a downstream (e.g. GLUE [52]) task.

3.1.4 Evaluation: GLUE Score

The GLUE benchmark [52] is an industry standard set of tests for natural language understanding. GLUE consists of nine tasks which either evaluate a single sentence or the relationship between a pair of sentences. For example, in a sentiment analysis task (SST-2), a single sentence is determined to be either positive or negative. Alternately, in a question answering task (QNLI), a pair of sentences is evaluated to determine whether the second sentence answers the question posed by the first. Each task results in a percentage score (0 to 100) which are often averaged to compute an overall "GLUE score". GLUE has been used to evaluate many models, with the original paper [52] presenting results for a few variants of BiLSTM [55] model. Importantly for our work, BERT and its subsequent variants [1, 3, 35, 48, 56, 57] are all evaluated using the GLUE benchmark. Notably, while the GLUE benchmark authors were only able to achieve an average score of 68.7 using BiLSTM, the BERT model achieved a much higher 82.1.

3.2 Inference Frameworks

One challenge in moving BERT inference onto edge devices is a lack of support in popular inference frameworks. Furthermore, in order to decrease latency and memory usage, fp16

and quantized int8 operations are preferable. In this regard, we develop an ARMCL BERT framework that runs on CPU and GPU and supports fp16 and int8 data types. To enable this implementation and optimize latency, we extend ARMCL with new quantizable GELU and LayerNorm implementations. ARMCL BERT will be detailed in chapter 4. However, in the development of this work, we also worked with the existing frameworks TensorFlow Lite and TVM.

3.2.1 TFLite

One of the most popular inference frameworks is Tensorflow Lite (TFlite) [58]. However, we choose not to use TFLite due to multiple limitations. Firstly, we were unable to use TFLite to run BERT models on an ARM GPU. This is due to a lack of operation support specifically on the GPU side within TFLite. Additionally, TFLite proved to be difficult to extract detailed profiling information. Finally, TFLite does not allow for low level control of which operations are assigned to the CPU or the GPU.

3.2.2 TVM

TVM is an end-to-end compiler stack that provides optimization at different levels of abstraction (e.g., at the graph- and tensor-level). The TVM front-end automatically compiles DNNs computational graph into minimum deployable modules in the format of high-level Relay intermediate representation (*Relay IR*). *Relay IR* is a purely functional

language to represent DNNs operations like add, multiply, or layer normalization as a data-dependency graph to balance efficient compilation, expressiveness, and portability. We refer readers to [59] for more detail. Because TVM is able to target both CPU and GPU processing elements, is is more generalizable that a framework such as TFLite. For this reason, we utilize TVM to perform fp32 profiling of BERT models for DNN-SMO. However, we were unable to profile lower precision BERT models using TVM. This task required a library with much lower level control, for which we choose ARM Compute Library (ARMCL).

3.2.3 ARM Compute Library

ARMCL [60] is a collection of highly optimized neural network operations targeting ARM CPUs and GPUs. Most importantly for BERT, it provides assembly optimized General Matrix-Matrix Multiply (GEMM) and GEMMLOWP (int8 GEMM) operations. However, as of ARMCL v22.05, there are still missing operations required for BERT, including GELU and an int8 quantized LayerNorm, for which we provide and verify our own implementations.

ARMCL has two main advantages over TVM and TFLite. Firstly, ARMCL provides superior performance to other frameworks, as each operation is hand optimized. Secondly, it is a much lower level library, which means it requires more effort to use but allows for much finer control over which operations are executed and on which processor. This also allows for optimal use of the shared memory between the CPU and GPU. Modern SoCs

include hardware such as ARM CoreLink [61], which maintains cache coherency between heterogeneous processors such as a CPU and GPU, enabling fast communication without memory copies. For these reasons, we utilize ARMCL to generate profiling data for int8 quantized and fp16 BERT models on both ARM CPUs and GPUs.

3.3 Heuristic Optimization Methods

There are many search problems for which the search space is far to large to search exhaustively. This includes games such as chess, the famous travelling salesman problem, and our problem of mapping tasks onto heterogeneous systems [30]. In order to handle problems of such magnitude, one approach is to utilize heuristic optimization methods. These algorithms trade completeness for speed, enabling the exploration of large search spaces with high effectiveness, though they may not find the global optimal solution. Two successful heuristic optimizers which we utilize are genetic algorithms [62] and the Grey Wolf algorithm [63].

3.3.1 Genetic Algorithm

Genetic algorithms are a form of evolutionary algorithm, meant to be analogous to biological evolution [62, 64]. In general, a genetic algorithm operates on a population that represents a set of solutions. Often, each solution consists of an array of numbers, which can be called genes. Each member of the population is evaluated by a fitness function, which is

analogous to the success of an organism in nature. At each iteration of a genetic algorithm, the population is updated and reevaluated. The update process usually consists of selection, crossover, and mutation, which each vary depending on the implementation and the problem being solved. In general, the selection process decides which members of the population are used to reproduce to create the next population. Commonly, the most fit members of the current population are selected while the least fit can be discarded. In the crossover step, selected solutions are mixed to create new solutions. Assuming each solution is an array of numbers, this can be done by selecting two solutions and picking a random split point, then concatenating the parts of different solutions to create two new ones. Finally, the mutation process allows the genetic algorithm to explore solutions that don't exist in the original population. This is done by randomly changing a certain proportion of genes at each iteration of the algorithm. Because the fitness function and gene structure used by a genetic algorithm are arbitrary, GAs have been applied to countless problems, such as engineering design and signal processing [65].

Multiobjective Genetic Algorithm: NSGA-II

While a standard genetic algorithm aims to optimize a single fitness function, multi-objective algorithms have also been developed. Where the standard genetic algorithm aims to find a single optimum in a search space, multi-objective GAs instead search for a Pareto-optimal front. One of the most successful is NSGA-II [66]. The adaptation from a

single objective to multi-objective algorithm is relatively straightforward. The standard genetic algorithm operations of crossover and mutation are still used but other pieces of the algorithm are modified to handle multiple fitness functions. The main issue with multi-objective optimization is the balancing of different fitness functions and maintaining diversity in the population. NSGA-II solves these problems by incorporating a crowding-distance computation, which measures the distance between members of the population (in terms of fitness values). This avoids the issue of requiring a manually defined sharing function and allows NSGA-II to maintain variance in the population. NSGA-II has been demonstrated to both converge better and find a superior Pareto front compared to PAES and SPEA (two other multi-objective optimizers) [66].

3.3.2 Grey Wolf Algorithm

The Grey Wolf algorithm [63] is a highly successful heuristic optimizer inspired by the hunting behavior of grey wolves. To do this, the algorithm considers a population of solutions to be wolves of varying rank. The best three solutions (called alpha, beta, and delta) guide the optimization process. Based on the positions of these top solutions in the search space, the position of the optimum is estimated and the other solutions are placed to surround that position, as wolves surround their prey. In order to balance exploration and exploitation of the design space, the randomization of wolf positions around the estimated optimum is carefully controlled. Over the course of the search, the wolves (solutions) transition from a

wide range of random positioning to being placed much nearer to the estimated optimum. In practice, the first half of the iterations of the algorithm are dedicated to exploration, covering as much of the search space as possible. Then, based on the results of the exploration, the remaining iterations aim to find the local optimum (prey). The Grey Wolf Algorithm was benchmarked on a large number of test functions and compared to other heuristic optimizers, including genetic algorithms. It compared favorably to other optimizers, which lead us to test it in our design space.

Chapter 4

Proposed Methodology

4.1 DNN-SMO

Figure 4.1 gives an overview of our proposed framework for optimizing DNNs (e.g. BERT) on heterogeneous processors. Our framework has two main components: a latency model and the DNN-specific mapping optimizer (DNN-SMO). In the first component, we capture the operation-level latency of a DNN, such as BERT, as well as communication costs on an embedded system (HiKey970). Using this data, we build a latency model for heterogeneous DNN execution. The latency model is used by DNN-SMO to optimize the PE mapping configuration. DNN-SMO is based on a genetic algorithm, which we augment to better fit the problem of heterogeneous DNN optimization. Note that while we build our optimization system around BERT on the Hikey970, and use BERT as an example to

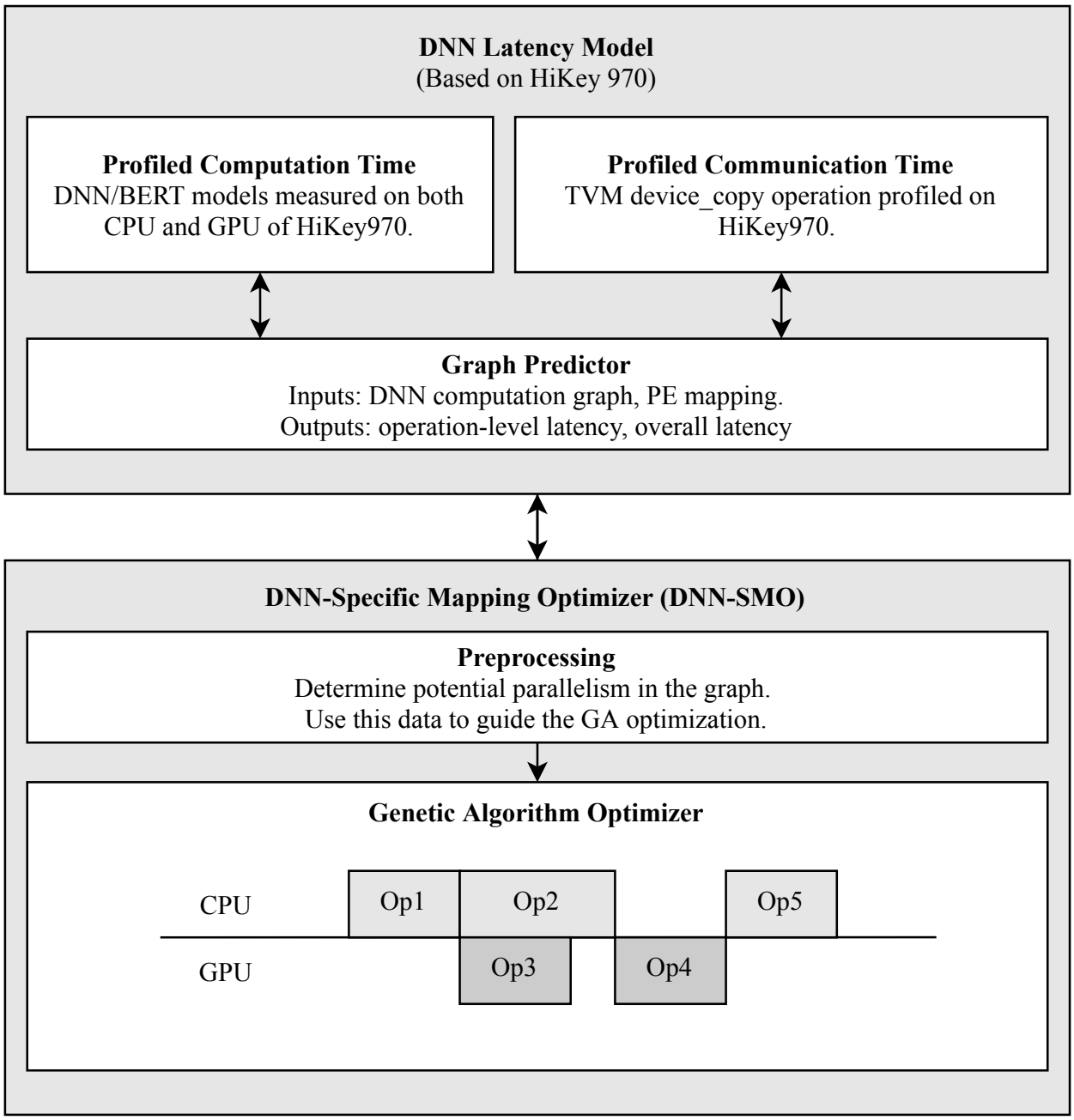


Figure 4.1: The proposed DNN mapping optimization framework.

explain our methodology, our system can be applied to other types of DNNs deployable to embedded CPU+GPU systems. We include a variety of BERT models and convolutional networks in our results.

4.2 Profiling

4.2.1 Computation Profiling

Hidden Size	128, 256, 512, 768
Attention Heads	2, 8
Hidden Layers	2, 4, 6, 8, 10, 12

Table 4.1: Profiled BERT model parameters

In order to build a latency model for BERT, we first need to understand the operator-level latency on both the CPU and GPU of an embedded system (HiKey970). We generate the range of BERT models shown in Table 4.1, which is based on the models tested in the original BERT paper [1]. We set the sequence length to 128 and the feed-forward size to 4x the hidden size, again based on the original work. Importantly, we include models that match the computational cost and parameter count of edge-targeted BERT models such as DistilBERT [32] and TinyBERT [67]. In order to ensure consistent latency, we set both the CPU and GPU of the HiKey970 to their highest frequencies and ensured that the board was properly cooled. Then, using TVM's debugger, we extract operation-level timestamps which we utilize to build our latency model.

4.2.2 Communication Profiling

To build a *heterogeneous* inference latency model, we also need to understand the cost of communication between the CPU and GPU. For this, we can again use TVM profiling, specifically TVM's *device_copy* operation, which uses OpenCL memory copy operations under the hood. In order to perform this profiling efficiently, we first identify the unique intermediate tensor shapes in our set of BERT models. We observed that the number of unique tensor sizes is relatively small (∼25 per BERT configuration). Additionally, many sizes are reused between BERT models. For example, adding more encoder layers to a BERT model does not add more unique intermediate tensors. Similarly, changing the number of attention heads will affect a few intermediate tensors, but many remain unchanged, especially in the feed-forward sections of the encoders. We perform measurements for both CPU to GPU and GPU to CPU transfers.

4.3 Latency Modeling

In order to feasibly search the massive design space for operation-to-PE mapping, we utilize a latency model. This allows us to save an enormous amount of time compared to profiling each configuration on real hardware. In this work, we consider only two PEs (CPU and GPU), though our model could extend to more, for example to include an NPU. We also assume that each PE is fully dedicated to one operation at a time: we maximize data-level

parallelism within each PE, while using task-level parallelism between the CPU and GPU.

To predict the latency of heterogeneous inference for a given BERT model and operation to PE mapping, we emulate execution of the computational graph: we keep track of the start time and completion time of each operation, with latency determined from our profiled data. When an operation is placed on a different PE from its data dependency, we insert delay based on profiled communication time. Once the graph is processed, we have both the total latency and timing information for each individual operation.

Latency Model Validation

Operation	CPU (ms)	GPU (ms)
1x1	2.34	1.67
3x3_1	2.45	1.06
3x3_2	3.48	1.21
3x3_3	3.86	1.38
5x5_1	1.95	0.73
5x5_2	6.30	3.56
pooling	4.24	1.47
concat	1.38	0.68
Total Latency	26.04	11.79

Table 4.2: Operation-level Latency of InceptionV3 Module on CPU & GPU using ARMCL

Instead of TVM, which we use to implement BERT on CPU and GPU individually, we utilize ARM Compute Library (ARMCL) [60] to validate our latency model. This is because TVM does not support heterogeneous inference. ARMCL is able to support heterogeneous computing, but it does not support BERT models. In order to validate our latency model, we need a network both complex enough to benefit from CPU-GPU parallelism and supported

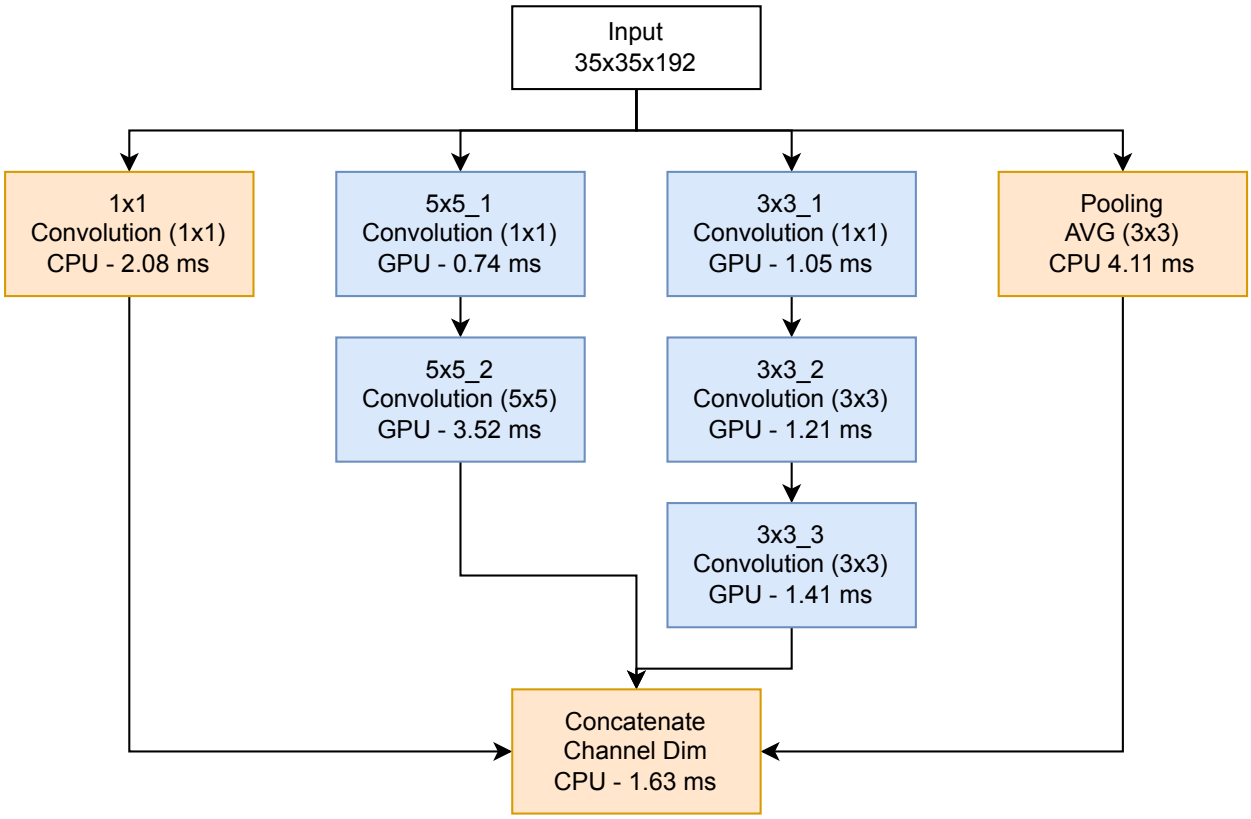


Figure 4.2: InceptionV3 Module Implemented in ARMCL for Latency Model Validation

by ARMCL. As such, we use InceptionV3 [4] as a test case. Specifically, we implement an Inception module that includes parallel 1x1, 3x3, and 5x5 convolutions, visualized in Figure 4.2. To validate our latency model, we manually partitioned operations between CPU and GPU. Figure 4.2 shows an example configuration where operations shown in orange are placed on the CPU and blue on the GPU.

Since our latency model is based on separate CPU and GPU profiling, we also implement single-device models in ARMCL, along with additional code that records the latency of individual operations (at the granularity of Figure 4.2). Table 4.2 contains our operation-

	Measured Latency	Predicted Latency	% Error
Test 1	9.96 ms	10.44 ms	4.52%
Test 2	18.90 ms	18.55 ms	1.88%
Test 3	12.58 ms	12.12 ms	3.70%
Test 4	14.90 ms	16.35 ms	9.73%
			Avg: 4.96 %

Table 4.3: Latency Model Prediction vs ARMCL Measurement for 4 Unique Device Mapping Configurations

level profiling results. In order to profile communication, we time the calls to ARMCL *Tensor.map()*, which is analogous to TVM's *device_copy*. For consistency, we run each model configuration (CPU-only, GPU-only, heterogeneous) 10 times with 2 warm-up runs, then take the average latency of the 10 runs. We repeated this experiment with a few unique device mapping configurations. The resulting measurements and latency model predictions are shown in Table 4.3 (Test 1 corresponds to the configuration shown in Figure 4.2). We observe that our latency model predictions are within 10% of the measured values for a variety of device mapping configurations, with an average error of 4.96%. As such, we are confident that our latency model is representative of real heterogeneous DNN execution.

4.4 DNN Mapping Optimization with GA

We optimize the assignment of DNN graph operations to heterogeneous resources using a genetic algorithm (GA). We implement our approach using the PyGAD library [68], which includes standard selection, crossover, and mutation operations. For an overview of genetic

algorithms, see [62].

4.4.1 Preliminaries

Genetic algorithms have a history of success in heterogeneous mapping problems [30, 44], as well as in DNN operation mapping [24, 45]. For some intuition on why they work for this problem, picture a set of random PE mappings. It is likely that some tasks will be better optimized on some PEs than others in the set. Through selection and crossover, a genetic algorithm is able to combine the best parts of the random configurations to generate superior configurations. Furthermore, mutation allows for the discovery of configurations not included in the initial population.

To adapt a genetic algorithm to the problem of DNN operation mapping, we define our genome as a set of 0s and 1s, representing the mapping of each DNN operation to the CPU or GPU respectively. Note that we could easily extend this to support more than two PEs by expanding the gene value space. We then define our fitness function as 1/latency, using the previously described latency model to evaluate each mapping configuration. This converts the problem of latency minimization to fitness maximization, which is the goal of GAs in general. For all GAs in this work, we use a population size of 50, as well as steady-state selection and single-point crossover operations.

4.4.2 Enhanced GA for DNN Mapping

In order to accelerate the mapping optimization process, we have enhanced the standard GA by tailoring it to DNNs in general and BERT models specifically. Our new system, which we call DNN-specific mapping optimizer (DNN-SMO), can be divided into two sections, as in Figure 4.1: a preprocessing stage, and an enhanced genetic algorithm optimizer. We have two overall goals in this approach. First, we focus the GA's search effort on more parallelizable regions of the computation graph, more quickly finding mappings that reduce total latency. Second, we allow branch-level mutations which modify groups of DNN graph nodes rather than one node at a time, resulting in mapping changes that benefit from reduced communication overhead. To achieve these goals, we modify both the initial population of the genetic algorithm and the search process itself.

Preprocessing

In order to implement our GA enhancements, we first add a preprocessing step, visualized in Figure 4.3. Our goal is to both identify branches of sequential nodes and determine how much latency can be reduced through parallelization. Preprocessing starts by selectively collapsing branches of the computational graph into single nodes. Then, for each group of parallel branches, we compute $1 - (longestBranch/sumOfBranches)$. This represents the percentage of latency within a group of branches that could be reduced if all branches were executed in parallel (as in Ahmdahl's Law). In Figure 4.3, we give a simple example of the

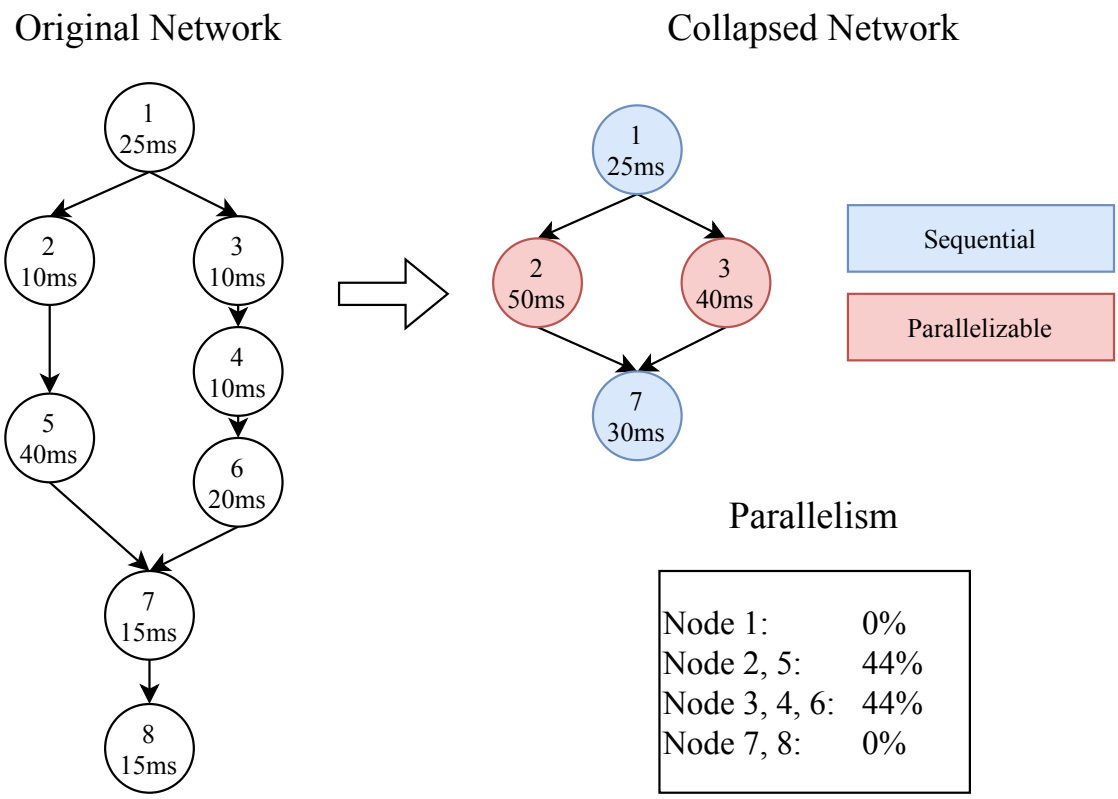


Figure 4.3: Visualization of preprocessing to determine parallelizable nodes.

branch collapse and parallelism computation. In this example, there is one group of parallel branches with a total computation time of 90ms. However, because the longest branch is 50ms, we can only cover 40ms or 44% of the computation time through parallelism. We record this parallelism value for every (original graph) node in the parallel branches. We will later use this value (0.44 in this example), to weight the search process of the GA.

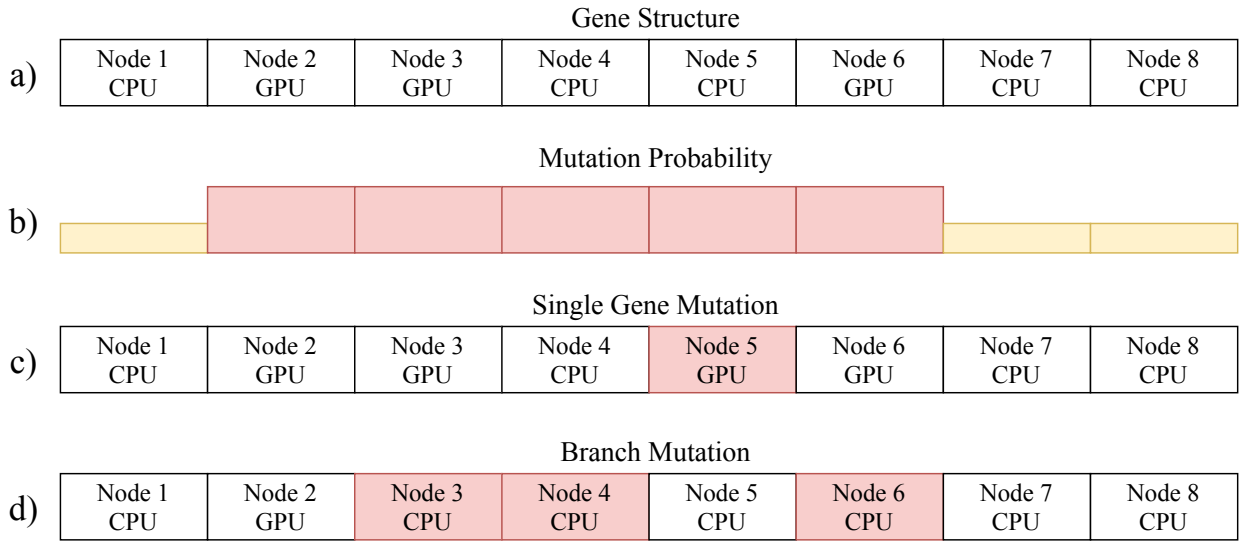


Figure 4.4: Visualization of custom mutation based on Figure 4.3. a) Gene structure. b) Weighted mutation probability. c) Example of single-gene mutation. d) Example of branch-level mutation.

DNN-specific Mapping Optimizer

Leveraging the metadata built during preprocessing, we implement multiple improvements, resulting in a DNN-specific mapping optimizer (DNN-SMO). First, we construct an initial population that deterministically maps sequential (unparallelizable) graph nodes to the fastest PE, minimizing communication costs on the critical path. We also improve the search process, especially for large BERT models, by implementing a custom mutation operation with three main features: gene reuse, weighted gene selection, and branch-level mutations. By combining these techniques with our custom initial population, we are able to massively decrease the number of generations required by the genetic algorithm.

Custom Initial Population: To generate a better initial population for the GA, our

main goal is to keep critical path tasks on the fastest processor. This addresses the problem with fully-random initialization, which results in poor latency and a poor starting point for optimization. We utilize metadata from preprocessing to create a selectively random initial population. For nodes that can be parallelized, mapping is randomized. However, nodes that cannot be parallelized are assigned to the PE that executes the full set of sequential nodes with the lowest latency. This can be seen in Figure 4.4a, in which nodes 1, 7, and 8 are placed on the fastest PE (CPU in this example), while nodes 2-6 are randomized. This corresponds to the structure of sequential and parallel nodes in the Figure 4.3 example. The logic behind this method is that sequential nodes represent the critical path of the computation graph. Mapping the sequential nodes to the fastest PE minimizes communication overheads, which is critical for latency minimization, especially on embedded systems.

Gene reuse means that a single gene value may correspond to multiple computation graph nodes. This is especially useful for BERT, which consists of a series of identical transformer encoders. Instead of optimizing all genes individually, we decrease the genome length and assign a single gene value for each set of repeated encoder operations. This significantly shrinks the search space, enabling much faster convergence. For non-BERT models or BERTs with varied encoder sizes [69], this feature can be disabled, though it may be useful for other networks with repetitive computational graph structures.

Weighted gene selection increases the probability that parallelizable nodes are selected for mutation. This guides the search process towards highly-parallel regions of the

computation graph. In the preprocessing stage, we determined the amount of parallelism available to each node. Now, we use that value to weight the mutation probabilities, visualized in Figure 4.4b. Specifically, we give all nodes in the graph a base weight of 0.2 (this is a hyperparameter). For parallelizable nodes, we add the parallelism value (usually 0.2-0.6) to this weight. We then convert this to a probability distribution by dividing each weight by the sum of weights (so that they add up to 1). This probability distribution is then passed to the mutation function and used to select genes to mutate. Combined with our custom initial population, we find that this allows the GA to find optimal configurations much faster.

Branch-level mutation adds structured multi-gene mutations to the genetic algorithm. By default, mutations only affect single nodes in the computational graph, as shown in figure 4.4c. However, we know that for a DNN graph, in order to minimize communication costs, we should keep nodes with data dependencies on the same PE. As such, we add a 50% chance of performing a branch-level mutation instead of single-node mutation. An example branch mutation is shown in figure 4.4d, in which the branch 3-4-6 is selected and placed on the CPU. This way, instead of depending on random mutations to coincidentally move a group of nodes at the same time, we are able to intentionally mutate towards better configurations.

4.5 ARMCL BERT

4.5.1 ARMCL Modifications

The first operation we added to ARMCL is the GELU activation function [53]:

$$GELU(x) = x * \frac{1}{2} * \left[1 + erf(\frac{x}{\sqrt{2}})\right]$$

For embedded GPUs, OpenCL provides erf as a standard math operation, so implementation was straightforward. On the CPU, code must be explicitly vectorized using ARM NEON instructions/intrinsics. While ARMCL already contains vectorized tanh and exp implementations, we found that using them to approximate GELU [53] was slow, as they are based on Taylor polynomials and require many instructions. In order to achieve low latency on CPUs, we therefore needed to write our own vectorized erf. We used the following approximation [70] due to its simplicity as well as its efficient mapping to multiply-accumulate instructions:

$$erf(x) = 1 - \frac{1}{(1 + a_1x + a_2x^2 + a_3x^3 + a_4x^4)^4}, x \geq 0$$

$$a_1 = .278393, a_2 = .230389, a_3 = .000972, a_4 = .078108$$

Using this erf implementation, the GELU implementation using NEON intrinsics is straightforward.

The second operation we added to ARMCL is int8 quantized LayerNorm:

$$LayerNorm(x) = \frac{x - Mean[x]}{\sqrt{Var[x] + \epsilon}} * \gamma + \beta$$

We first tested the existing fp32 LayerNorm, inserting dequantization and quantization operations as needed. However, this required extra intermediate tensors and additional DRAM accesses. To avoid this cost, we add a new LayerNorm that operates directly on int8 quantized tensors. Using ARM NEON intrinsics, we load vectors of 16x 8-bit values and accumulate them using low bit integer arithmetic. The computation of the mean and standard deviation, as well as the actual normalization, are performed using fp32. Finally, the normalized values are requantized back to int8 and stored as vectors.

4.5.2 BERT Implementation

Using our augmented ARMCL, we implement BERT-base with fp32, fp16, and int8 datatypes. The embedding and FFN operations are implemented using gather and GEMM/GEMMLOWP operations respectively.

The MHA operation is more complex, as it is usually implemented with transpose and reshape operations. Instead, we use ARMCL's SubTensor interface to divide the Q, K, and V

tensors into sections corresponding to attention heads. We found this to significantly reduce latency compared to ARMCL's reshaping functions, as many memory copy operations are eliminated.

For quantized operations, we apply asymmetric tensor quantization, meaning that each tensor is assigned a scale and zero point that maps 8-bit integers to floating point values. The scale and zero point are determined by first running the model in fp32 precision. For each tensor, the min and max are used to determine the quantization parameters.

4.6 Quantization & Multi-objective Optimization

While DNN-SMO is able to perform optimize latency by partitioning a DNN between CPU and GPU, it is limited to a single data type. Due to the rise of techniques such as quantization [35, 57, 71], we also investigated the potential of quantizing specific operations in a network. This necessitates a multi-objective optimization, as quantization is known to reduce accuracy compared to full fp32 precision. In order to include accuracy in our process, we needed to build an accuracy model for BERT which is able to handle models in which operations have different quantization settings. The latency model is completely reused, though profiling is performed using ARMCL BERT rather than TVM and extra parameters are added to contain the quantization information.

For the optimizer, we then replace the Genetic Algorithm of DNN-SMO with NSGA-II [66], which is a multi-objective variant. Since NSGA-II shares many

characteristics with a standard Genetic Algorithm (GA), we are able to incorporate much of the same domain-specific knowledge as before. In particular, the weighted mutation, branch mutation, and custom initialization are all reused. Regarding gene reuse, we found that sharing configurations across encoders was harmful to the optimization of quantization, as it limited the search space in ways that resulted in worse accuracy/latency trade offs. Overall, the structure of this new optimization system is extremely similar to that of DNN-SMO shown in Figure 4.1.

4.6.1 Accuracy Model

In order to build an accuracy model, we first needed to measure the accuracy of BERT models with various quantization configurations. To enable the measurement of a large number of configurations, we chose the AdaQuant method [72], which performs post-training quantization and requires relatively little calibration time. We also tested quantization-aware training methods but found them to be too slow to be feasible. Based on previous works [35], we choose the GLUE benchmark [52] as our dataset for accuracy measurement. We trained five BERT-base models for each GLUE task, varying the random seed used for training, as we found these to have significant impacts on accuracy. For each of these models, we then generated a large set of random quantization configurations and calibrate them using AdaQuant [72]. These configurations were generated by randomly setting the precision of each layer to fp32, fp16, or int8. We then measured the accuracy of each of these on the

GLUE tasks to create a large dataset.

For the accuracy modeling task, we utilize the Auto-Sklearn package [73], which is an AutoML [74] method. We also tried using a neural network to predict accuracy, but we found that Auto-Sklearn was more accurate when given enough training time, and much simpler to develop. Using Auto-Sklearn, we are able to easily split the dataset into train and test sets and the library then handles the training and hyperparameter optimization of various ML methods. We trained a separate Auto-Sklearn model for each of the GLUE tasks, as we observed different responses to quantization for each of them. The separate Auto-Sklearn model predictions can then be averaged to predict an overall GLUE score. We found that these predictions were extremely accurate, with less than 0.1% error.

Chapter 5

Experimental Setup

The goal of our experiment is to improve the latency of BERT on modern edge hardware. Due to practical limitations, we ended up utilizing different hardware and software environments for our experiments.

5.1 DNN-SMO

5.1.1 Hardware Platform: HiKey970

Our optimization framework is targeting heterogeneous embedded systems. The HiKey970 [41] is a heterogeneous system containing a Mali-G72 MP12 GPU, four high performance (big) A73 cores, and four lower performance (LITTLE) A53 cores. The architecture and performance of the HiKey970 is representative of common mobile

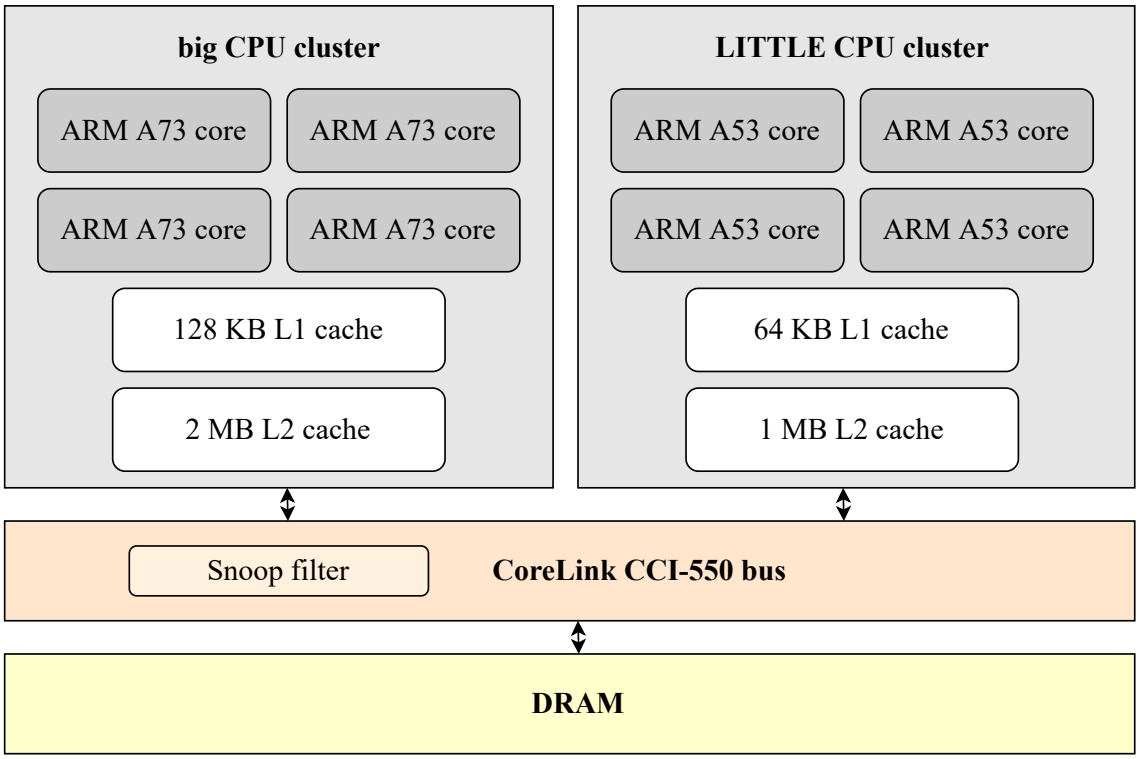


Figure 5.1: An abstract block diagram of *ARM big.LITTLE* heterogeneous CPU clusters in HiKey970 embedded platform. Note that we do not include GPU and NPU of HiKey970 in this diagram.

hardware [19] and is visualized in figure 5.1. As such, our framework is expected to be generalizable to a wide range of modern edge devices.

5.1.2 Software Platform: TVM

For the software implementation of BERT on the HiKey970, we use the TVM framework [28]. We chose TVM because, in our testing, it was the only framework that supported the

operations required by BERT on both CPU and GPU. Given a DNN computational graph, TVM automatically generates code for both CPU and GPU targets. It uses LLVM to generate CPU-targeted code and OpenCL for GPU-targeted code. Unfortunately, because the code generation needs to generalize to any hardware, it does not match the performance of a hand-optimized library. Fortunately, TVM provides auto-scheduling functionality [75], which is able to optimize the generated code to the level of manual optimization. In order to achieve strong performance on the HiKey970, we run TVM auto-scheduling until the latency stabilizes. Note that TVM does not utilize the four LITTLE cores of the HiKey970, as previous works [22] have shown that using only the big cores results in the lowest latency.

5.2 ARMCL BERT

Due to the versatility of ARMCL, we were able to perform measurements on an additional hardware platform for our ARMCL BERT implementation, as well as the HiKey970.

5.2.1 Hardware Platform: Galaxy A52

The Galaxy A52 has an ARMv8.2-A SoC with 2 Cortex-A77 high performance cores, 6 Cortex-A55 efficiency cores and an Adreno 619 GPU. Notably, the ARMv8.2 CPU includes fp16 compute operations absent in the v8 chip of the HiKey970. This enables additional measurements of fp16 operation latencies on the CPU which are not possible on the HiKey970.

5.2.2 Software Environments

In order to run our ARMCL BERT model on both the HiKey970 and the A52, we had to compile the library slightly differently for the two devices. Firstly, the operating systems differ, with the HiKey970 running a Linux OS and the A52 running Android, which is Linux based but creates some challenges. Secondly, due to the differing CPU capabilities mentioned above, the ARMCL compiled for the HiKey970 does not support the same set of operations. For both devices, we compile ARMCL in release mode with all supported compiler optimizations, as this makes a large difference in performance, especially for int8 operations.

Because ARMCL uses OpenCL to run GPU compute operations, it is important to note that they depend on vendor specific GPU drivers. This is relevant due to the fact that the HiKey970 has a Mali-G72 GPU while the A52 has a Qualcomm Adreno 619. As a result, the GPU drivers were written by different manufacturers and their performance may differ. In our experience, the Adreno driver seemed to have severe issues that were not present with the Mali driver. Specifically, we observed severe performance degradation with large fp32 matrices on the Adreno 619. Additionally, the Adreno driver was unable to compile certain compute kernels which were handled properly by the Mali driver.

5.3 Latency Measurements

In all cases, we apply standard techniques for measuring DNN execution time. For each measurement, the model is executed at least 10 times. We then take the average runtime excluding the first run (for warmup reasons). The inputs provided to the model are simply random values, as they do not matter for latency measurement. Note that we observed very high consistency in latency, meaning that the mean and median values were essentially identical.

The APIs used to measure runtime depend on the software platform of the model. For ARMCL measurements, we use the built in C++ time functions and for TVM, we use the equivalent Python APIs. In either case, runtime is simply measured by fetching the time before and after model execution and computing the difference.

5.4 Multi-objective Optimization

For the accuracy model used in the multi-objective version of our optimizer, we perform accuracy measurements on server hardware running PyTorch. For the latency model, we utilize our ARMCL BERT implementation on the HiKey970, as described above. Since the HiKey970 supports fp32 & fp16 on the GPU and fp32 & int8 on the CPU, we perform latency measurements for those configurations. Unfortunately, due to the aforementioned Adreno driver issues, the A52 was deemed nonviable for this experiment.

Chapter 6

Results

Here, we discuss the results of the methods described in chapter 4. First, we detail the performance of DNN-SMO, including comparisons to other optimization methods, an ablation study of our modifications, and a review of inference latency improvements for various models. Next, we discuss the performance of ARMCL BERT, with particularly interesting results regarding different quantization levels and hardware platforms. We also explore the impact of modifying the BERT model, again using ARMCL BERT for latency measurement. Finally, we show the potential improvement from joint optimization of quantization and heterogeneous computing using NSGA-II.

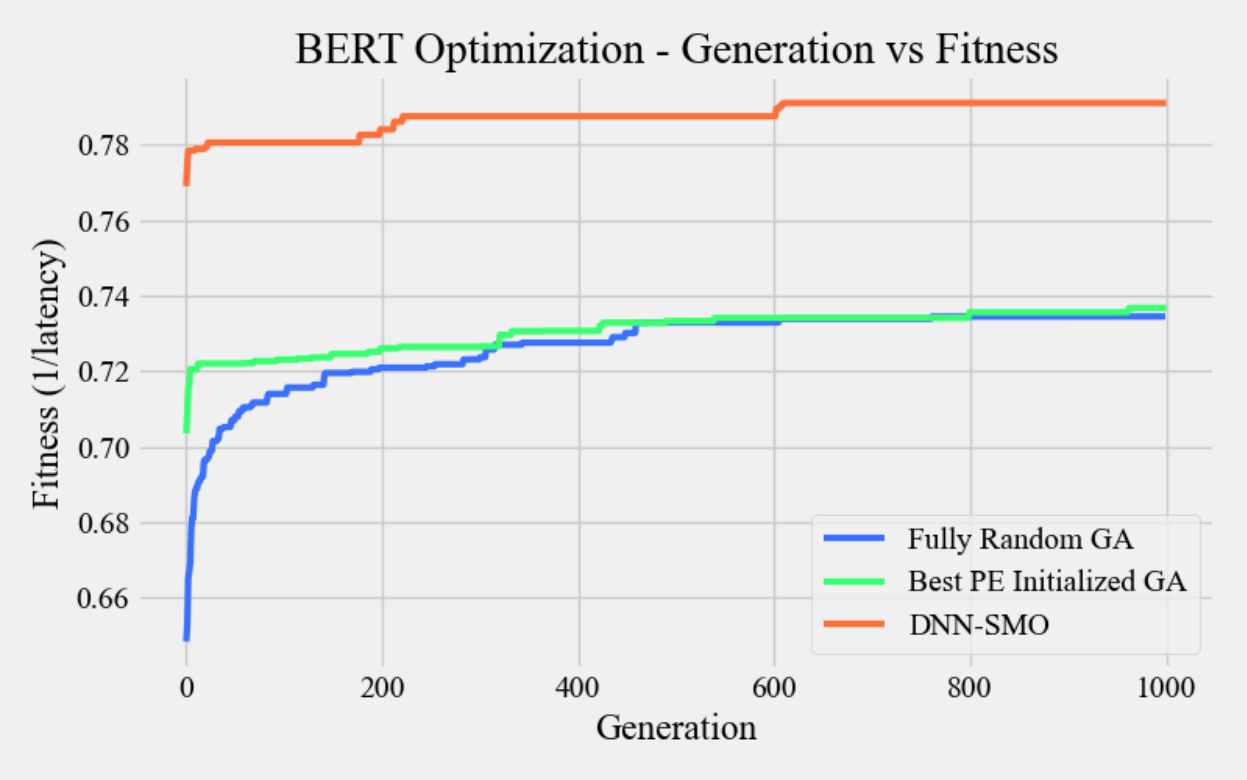


Figure 6.1: Chart of generation vs fitness showing improved convergence with DNN-SMO. In this example, the BERT configuration is hidden size 512, 2 attn. heads, 6 hidden layers.

6.1 DNN-SMO

6.1.1 Comparison to Baseline GA

Figure 6.1 shows the improvement in search time using our GA compared to baseline GAs. This chart was generated by averaging three runs of each GA on a BERT model with hidden size 512, 2 attn. heads, and 6 hidden layers. It shows the genetic algorithm fitness (defined in section 4.4) vs generation for three optimizers. Firstly, we have the unmodified (Fully Random) genetic algorithm, which uses a randomized initial population and no search

guidance, as in [24]. Secondly, we have a version which uses an initial population where all the nodes are placed on the overall fastest PE (i.e., best PE initialized). As we can see, this results in a stronger initial population, but the subsequent improvements are roughly identical. This makes sense, as a better initialization point alone does not help with avoiding local minima. Finally, we have our method (DNN-SMO), which uses the semi-random initial population as well as our custom mutation operation. As we can see, our method is able to achieve much stronger fitness than the other approaches. Through our custom population initialization, the GA is given an extremely strong starting point. Then, using the modified mutation operation, improvements are found rapidly.

In order to numerically compare our method to the unguided GAs, we ran our GA for 1000 generations and the randomized GA for 3000 generations. We repeat this experiment 20 times on the same BERT model (hidden size 512, 2 attn. heads, 6 layers). We focus on the (512, 2, 6) configuration as it is similar to DistilBERT [32] and therefore representative of a likely edge BERT deployment. However, similar results were found for both smaller (256, 2, 4) and larger (768, 8, 12) configurations. We find that on average, our GA was able to improve latency 2.05x as much as the random GA (std. dev = 0.2). For larger BERT models, this means our method results in ∼110ms latency reduction after 1000 generations, while the baseline GA only manages ∼55ms after 3000 generations.

To further illustrate the acceleration in mapping optimization achieved by our algorithm, we compare a baseline GA run for 20000 generations against DNN-SMO given

Model	Baseline GA 20k Generations	DNN-SMO 1k Generations
BERT (256, 2, 6)	40.58 ms	36.00 ms
BERT (512, 8, 6)	137.84 ms	130.32 ms
BERT (768, 2, 4)	184.18 ms	181.84 ms
BERT (768, 8, 12)	653.70 ms	627.70 ms
SqueezeBERT	271.59 ms	242.79 ms
InceptionV3	217.51 ms	216.40 ms

Table 6.1: Baseline GA vs DNN-SMO

1000 generations. The resulting latencies are shown in Table 6.1. For each model, DNN-SMO is able to achieve lower latency within 1000 generations compared to the baseline GA after 20000. The InceptionV3 test here is especially important, as gene reuse is not used for non-BERT architectures, so even with one optimization disabled, our algorithm is still far superior to the baseline GA. To emphasize the importance of this reduction in search time, consider performing this optimization with the baseline GA and without a latency model. Without a latency model, each configuration needs to be profiled, which in our experience takes ∼10s per configuration, as each model needs to be compiled and then executed multiple times. Running for 20000 generations, each of which requires 50 model evaluations, results in an optimization time of 115 days. Meanwhile, running our GA for 1000 generations with our latency model takes under an hour for the largest DNN models, and likely finds stronger mapping configurations.

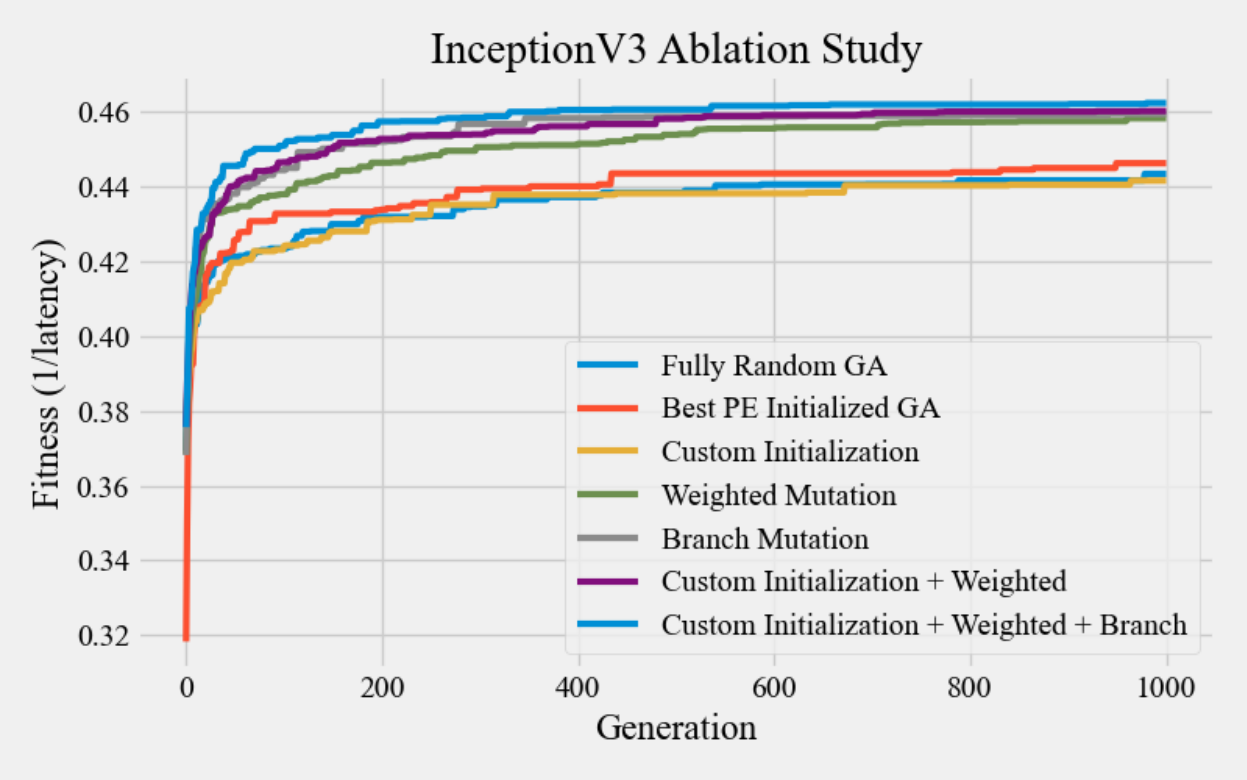


Figure 6.2: Ablation study showing the impact on GA convergence for the individual components of our optimizer: Custom Initialization, Weighted Mutation, and Branch Mutation. We use InceptionV3 in this test.

6.1.2 Ablation Study

In addition to the search acceleration demonstrated on BERT, we present an ablation study to show the contribution of each component of our optimization. To demonstrate the versatility of DNN-SMO, we use InceptionV3 for this test, and exclude the gene reuse component, as it is BERT-specific. We investigate the effect of custom initialization, weighted mutation, and branch-level mutations, both individually and in combination with each other. Again, we use the fully random and best PE GAs as baselines. The resulting

fitness vs generation chart is shown in Figure 6.2. For comparison, we run all methods for 1000 generations and average over three runs. As we can see, the fully random and best PE GAs perform similarly, as in the BERT experiment, resulting in 28.3% and 28.7% latency reduction respectively. Interestingly, we find that our custom initialization alone is unable to outperform these methods, with 28.0% latency reduction. Next, by using weighted mutation alone, we are able to find a significant improvement over the basic methods, with 30.6% latency reduction. Furthermore, we observe that combining our weighted mutation with custom initialization results in even better latency, with 30.9% reduction. This makes sense, as these components are meant to synergize, with the custom initialization placing the lower-weighted nodes near-optimally. Next, we find that branch-level mutations alone result in a significant improvement of 30.8%, nearly matching the combination of custom initial population and weighted mutation. Finally, as we would expect, the combination of all components results in the best latency reduction, with 31.2% as reported in Table 6.2.

To compare the speed of convergence, we can take the fitness value achieved by our best performing method at 200 generations and see how many additional generations were required by the other methods to match it. We find that the fully random, best PE, and custom initialization GAs are not able to achieve this value within 1000 generations, showing the importance of our modifications to the mutation operation. With weighted mutation only, it takes 849 generations to match the combined system 200 generation fitness. Branch-level mutation and the combined custom initialization + weighted mutation algorithms are

DNN	BERT															CNN		
Hidden Size	128		256			512				768								
Attn.Heads Hidden layers	2 2	2 12	2 2	8 2	8 12	2 2	2 4	2 12	8 2	2 2	2 4	2 6	2 12	8 2	8 12	SqueezeBERT [35]	InceptionV3 [31]	SqueezeNet [36]
CPU(ms)	3.66	21.3	13.0	14.4	86.1	45.7	97.7	285.6	47.2	99.7	201.6	307.8	625.5	110.1	742.8	318.0	314.5	45.8
GPU(ms)	11.98	68.1	25.1	23.3	137.5	57.8	118.0	342.3	65.7	115.7	235.9	350.2	687.9	129.8	762.7	347.0	402.4	56.5
Hetero(ms)	3.66	21.3	12.1	13.4	80.2	41.8	87.0	256.6	43.1	90.0	181.8	276.6	551.0	97.5	630.6	242.7	217.3	56.5
Improv. (%)	0%	0%	7%	7%	7%	9%	11%	10%	9%	10%	10%	10%	12%	11%	15%	24%	31%	7%

Table 6.2: Latency model results for DNN models using heterogeneous CPU-GPU execution. BERT configurations specified with hidden size, attention heads, number of hidden layers. All BERT models have sequence length 128 and feed-forward size = 4x hidden size. Also included are tests on popular CNN models SqueezeBERT [3], InceptionV3 [4], and SqueezeNet [5].

similar by this metric, with 346 and 480 generations respectively.

6.1.3 Comparison to Non-GA Optimizers

To further illustrate the performance of DNN-SMO, we also perform a comparison to other heuristic optimizers. The optimization methods we adapt are Grey Wolf (GWO) [63], Harris Hawks (HHO) [76], and reinforcement learning [29]. Like the genetic algorithm, GWO and HHO are nature-inspired optimization methods, but they use entirely different mathematical models. The reinforcement learning method we implement is based on [29], and uses an LSTM-based sequence-to-sequence model to learn to optimize device placements. The resulting comparison is visualized in Figure 6.3. In this chart, the each iteration corresponds to 50 evaluations of our latency model, so each method is given roughly the same amount of search time. As we can see, the other optimizers are unable to

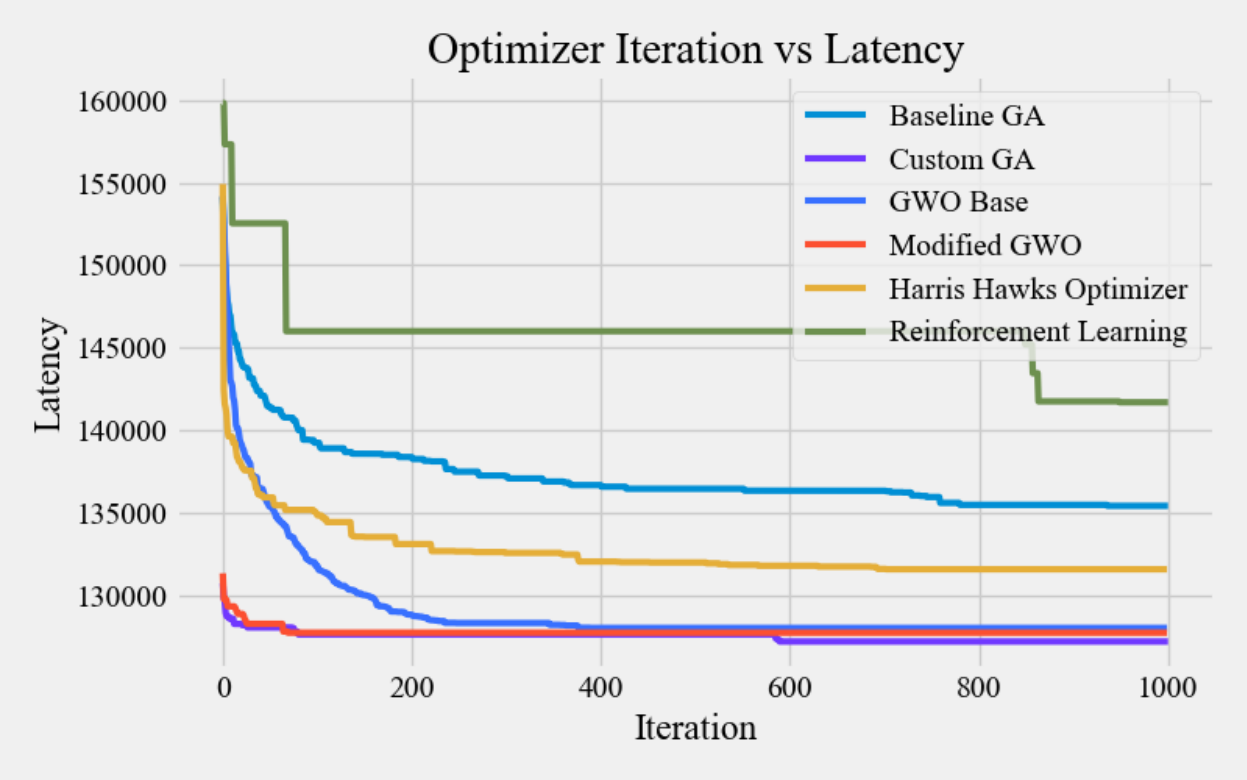


Figure 6.3: Chart comparing the convergence of alternative heuristic optimizers. In this example, the BERT configuration is hidden size 512, 2 attn. heads, 6 hidden layers.

match the performance of our customized genetic algorithm, even though both GWO and HHO significantly outperform the baseline GA. In our experience, the RL algorithm would learn to simply place operations on the fastest individual PE, rather than exploiting parallelism, resulting in the worst latency results. For a more competitive comparison, we also modified the GWO with both the custom initialization and gene reuse features. Even when initialized identically, our GA is able to explore configurations that the GWO cannot, resulting in a superior final latency. This also confirms the value of the branch-level mutation and weighted gene selection features. As the GWO uses a different mathematical

model, we were unable to replicate these features in the modified version.

6.1.4 Single-inference Latency Improvement

Table 6.2 shows our latency measurements for single-PE inference and the improvement predicted by our latency model for heterogeneous execution. Interestingly, we found CPU to be slightly faster than GPU for DNN execution on the HiKey970 when using TVM. Previous works, such as [25], have shown that mobile CPUs can outperform mobile GPUs for convolutional networks. This may be affected by many factors, especially the choice of inference framework (we use TVM), and the particular hardware (HiKey970). Our observation that the performance is similar between the CPU and GPU aligns with what we know about DNN performance on common embedded hardware [19].

In our testing, we first recognize that the model architecture has a large impact on how much parallelism can improve inference latency. In addition to BERT models, we also ran tests on InceptionV3 [4], SqueezeBERT [3], and SqueezeNet [5], which are CNNs that contain significant parallelism in their computational graphs. For BERT models, we are able to achieve up to 15% inference latency improvement. However, for InceptionV3, we are able to reach 31% improvement. Comparing the architectures of BERT and Inception, we are able to see why this is the case. In BERT, we find that there is a lot of potential parallelism in the multi-head attention layers, but not in the feed-forward layers. Because the feed-forward layers represent roughly 2/3 of the computational cost of the model, our

ability to improve performance through operator-level parallelism is limited. InceptionV3, on the other hand, contains parallel convolution operations throughout the network, with sequential sections only at the input and output [4]. SqueezeNet is an interesting case, as it contains similar parallelism to the Inception model [5]. However, we observe that the operations in SqueezeNet are quite small and the communication cost becomes significant, leading to only 7% improvement from heterogeneous execution. SqueezeBERT [3] replaces the fully-connected layers of BERT-base with convolutions. While a large amount of computation is still dedicated to sequential operations, we find that the multi-head attention layers constitute a larger portion of the total latency. As a result, we achieve better latency improvement with SqueezeBERT than with the standard BERT models, up to 24%.

For BERT models, we also observe some interesting trends relating the model size to the potential latency improvement from heterogeneous execution. For the smallest BERT models (hidden size 128), the CPU is significantly faster than the GPU. Previous works [77] have shown kernel launch overheads are large for DNNs on mobile GPUs, accounting for this performance gap. This means heterogeneous execution cannot improve the inference latency for the smallest BERT models. For hidden size 256, we observe a modest 7% improvement in latency. As the BERT model size increases, we receive larger latency improvement, with around 10% and 12% improvement for hidden sizes 512 and 768, respectively.

We also observe that the percentage of time that can be reduced is mostly constant as the

number of hidden layers is increased. For example, we see that for hidden size 768 and 2 attn. heads, the latency improvement is consistently 10-12% regardless of number of hidden layers. This makes sense, as transformer encoders in the BERT model are identical, meaning that the ratio of time saved through heterogeneous execution should also be consistent. Finally, we note that the number of attention heads does not significantly affect either the absolute latency or the latency decrease from heterogeneous execution. This is reasonable, as the structure of multi-head attention means that changing the number of attention heads does not significantly change the number of FLOPs [2]. Additionally, since multi-head attention accounts for roughly 1/3 of BERT execution time, the impact of changing the number of attention heads is further limited.

6.1.5 Summary of Results

Firstly, we show that DNN-SMO is vastly superior to a baseline GA, finding lower latency mappings within 1000 generations than a baseline GA given 20000 generations, across a variety of DNN models. We use an ablation study to show that each of our modifications (custom initialization, weighted mutation, branch-level mutation) contributes to this massive improvement in search speed. We then compare to three other optimization methods found in the literature: Grey Wolf [63], Harris Hawks [76], and reinforcement learning [29], finding that DNN-SMO results in the best mapping configuration given the same number of model evaluations. Finally, we show significant latency improvements of up to 15% for BERT, 24%

for SqueezeBERT, and 31% for InceptionV3.

6.2 Validation of ARMCL Operations

To validate our GELU implementation, we generate test tensors and compare to reference values computed using erf from the C++ stdlib. We fill tensors with random numbers in the range (-4, 4), the same range as tested in [57]. On the GPU side, we found that the OpenCL erf is near identical to the stdlib version, resulting in a maximum error of 2.3e-7. On the CPU side, we find a maximum error of 0.0006, which is small enough to be insignificant based on analysis from the I-BERT paper [57]. In I-BERT, a polynomial approximation of GELU with an error of up to 0.018 is used with no accuracy penalty. Since our implementation is well within this bound, it will not impact a BERT model's accuracy.

For LayerNorm, we use the existing fp32 LayerNorm and quantize its output to generate a baseline. Then, we quantize the input and apply the int8 LayerNorm. Dequantizing, we find a maximum error of $\sim$0.003 per unit range, which is within the expected quantization error [57].

6.3 ARMCL BERT Latency

We measure the latency of BERT on two hardware platforms, the HiKey970 and Samsung Galaxy A52. The HiKey970 has an ARMv8-A SoC with 4 Cortex-A73 big cores, 4 Cortex-

A53 LITTLE cores, and a Mali-G72 MP12 GPU. The A52 has an ARMv8.2-A SoC with 2 Cortex-A77 high performance cores, 6 Cortex-A55 efficiency cores and an Adreno 619 GPU. Notably, the v8.2 CPU includes fp16 compute operations absent in the v8 chip.

Platform	fp32	fp16	int8
A52 CPU	0.392 s	0.203 s	0.129 s
A52 GPU	132 s	0.791 s (*)	N/A
HiKey970 CPU	0.563 s	N/A	0.280 s
HiKey970 GPU	0.341 s	0.272 s	N/A

Table 6.3: BERT-base Latency Measurements

Table 6.3 shows our latency measurements for BERT-base across tested hardware platforms. Overall, our measurements are in line with previous works such as [27], which measured the latency of quantized CNNs on mobile SoCs, though they did not test any CPUs with fp16 support. Our first observation is that on the A52 with CPU fp16 computation, we find 49% latency improvement over fp32, while int8 achieves 67% speedup. The CPU in the HiKey970 does not support fp16, but we find that int8 improves latency by 50% over fp32. These measurements illustrate the potential benefits of lower-precision inference for DNNs in general and BERT in particular. It is also interesting to see that on the newer SoC, int8 achieves even greater speedup than on the older chip. This shows that significant improvements have been made in the integer SIMD units in newer devices. Meanwhile, our observation that fp16 halves latency makes this a strong option, as conversion from fp32 to fp16 is much simpler than quantization.

On the embedded GPU side, the HiKey970 gives very strong results, with 40% latency

reduction compared to the CPU in fp32. Converting from fp32 to fp16, a further 20% latency reduction is observed, resulting in very fast BERT inference. The A52 GPU performs poorly by comparison, taking over 2 minutes to execute in fp32 and suffering a kernel compilation error in fp16. We believe this is a driver issue, as other benchmarks [1] show that the Adreno 619 and Mali-G72 MP12 should perform similarly. We were able to find a workaround for the A52 GPU fp16 configuration, and report the resulting value, but it still performs poorly, with over 2x the latency of the A52 CPU fp32 configuration.

Overall, our results indicate that mobile GPUs have great potential for fast and efficient DNN inference, especially if fp16 compute is available. However, they are inconsistent, as our experience with the A52 shows. For older CPUs, int8 quantization is the only option and can achieve 50%+ speedup over fp32, while reducing memory usage by 4x. For newer CPUs, fp16 quantization may be preferred, as it may achieve higher accuracy than int8 but will only reduce memory usage by 2x. However, we do find that on the A52, int8 is still faster than fp16, making this the best option if accuracy can be sacrificed.

6.3.1 ARMCL BERT Variants

In addition to the standard BERT implementation, we also test a couple variations, with latency results shown in Table 6.4. The goal of these experiments is to understand more about the latency behavior of BERT and especially how integer quantization can be used to

[1] https://www.notebookcheck.net/Adreno-619-vs-Mali-G72-MP12_10584_8138.247598.0.html

accelerate it. Firstly, we test a simple variation related to MobileBERT [33], referred to as "Simplified". In particular, this variation replaces GELU with ReLU, replaces LayerNorm with NoNorm, and uses a simple approximation for SoftMax. We test this variant at both fp32 and int8 precisions on the CPU of both hardware platforms.

Secondly, we also implemented the operations of I-BERT [57]. This work utilizes polynomial approximations for the nonlinear operations of BERT, requiring special implementation in ARMCL. It is also important to note that I-BERT requires symmetric quantization, while other methods [35] allow for the specification of arbitrary zero-points for quantized values.

BERT Variant	Platform	fp32 Latency	int8 Latency
BERT-base	A52 CPU	0.396 s	0.137 s
	HiKey970 CPU	0.616 s	0.285 s
Simplified	A52 CPU	0.383 s	0.112 s
	HiKey970 CPU	0.578 s	0.264 s
I-BERT	A52 CPU	N/A	0.118 s
	HiKey970 CPU	N/A	0.270 s

Table 6.4: Latency of BERT Variants

This information is somewhat orthogonal to the rest of our work, but contains some very interesting results. Overall, we observe relatively small latency improvements comparing the Simplified BERT to BERT-base, with around 5% improvement in fp32. Due to integer math optimizations this implementation enables, the improvement is up to 18.5% in int8, but it is clear that most computation time of all BERT models is dedicated to matrix multiplications.

Secondly, we find that I-BERT [57] is able to achieve nearly the same latency as the

Simplified BERT. This is extremely important, as I-BERT utilizes very accurate polynomial approximations, while this Simplified BERT (and MobileBERT [33]), replace operations with entirely different ones. This means that I-BERT is more capable of maintaining the accuracy of the full size BERT model. This also demonstrates the efficiency of integer-only computation, as even complex nonlinear functions can be executed extremely quickly without much accuracy degradation.

6.4 Quantization Optimization Results

Figure 6.4 shows an example Pareto front generated by the multi-objective version of our optimizer. This result was generated using latency data from the HiKey970, and we present the baseline GPU fp32 latency and accuracy for comparison. The genetic algorithm is configured with a population of size 100, creating 20 offspring per iteration, and running for 50 generations. Overall, we find latency values in the range 0.227 s to 0.254 s, all of which compare favorably to single device inference on the HiKey970. Considering the best found latency of 0.227 s, this is a 60% reduction in latency compared to CPU fp32 inference and a 34% reduction compared to GPU fp32. Furthermore, it is also a reduction of 20% compared to full int8 quantization on the CPU. Additionally, we found that even this minimum latency configuration has higher accuracy than a fully int8 configuration, with an average GLUE score of 82.56% vs 82.39%. However, we note that the accuracy of all configurations fall within a narrow range due to the strong performance of the AdaQuant [72] method. As

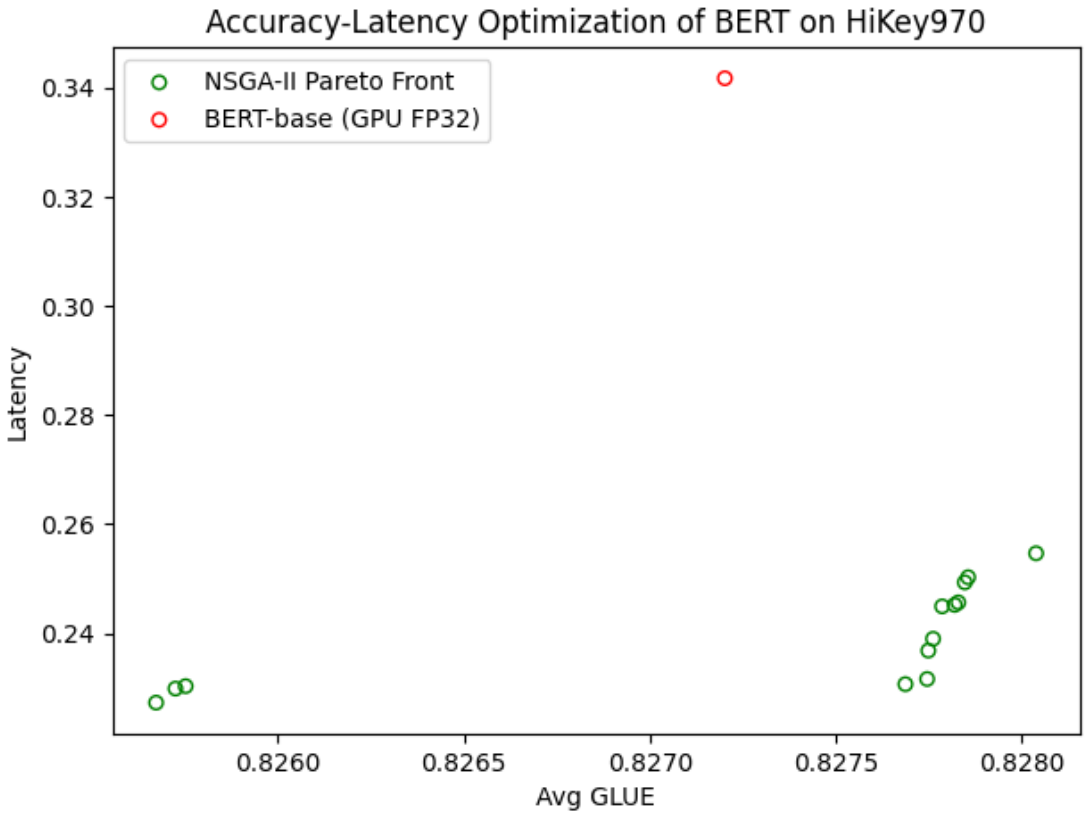


Figure 6.4: Example Pareto Front from NSGA-II Optimization

mentioned previously, the choice of this quantization method enabled us to generate a large number of quantized models in relatively little time. In general, we observed fairly low accuracy loss using this method, especially compared to simpler min-max quantization.

Chapter 7

Conclusion And Future Work

We introduced a genetic algorithm based DNN-specific mapping optimizer (DNN-SMO). DNN-SMO is able to find superior mappings of DNN computation graph nodes to a CPU+GPU heterogeneous system when compared with a standard genetic algorithm, while running 20x fewer generations. This is achieved through modifications to the GA: custom initialization, gene reuse, weighted mutation, and branch-level mutations. These enhancements are powered by a deterministic preprocessing step, which extracts metadata about parallelism available in the computation graph. From that information, our modifications accelerate the GA search process by both providing a stronger initial population and by generating mutations that are more likely to result in improvements. We evaluated DNN-SMO mapping BERT, SqueezeBERT, and InceptionV3 models to the HiKey970. We observed that our approach can improve inference latency by 15%, 24%,

and 31% for BERT, SqueezeBERT, and InceptionV3 respectively. We also evaluate the custom initial population, weighted mutation, and branch-level mutations individually through an ablation study, showing that they each contribute to improving search speed. Comparing DNN-SMO to other heuristic optimizers, we find that DNN-SMO finds the best mapping given the same number of model evaluations. In the future, we may extend this work to include other processing elements found in modern devices, such as DSP and NPU. We may also incorporate quantization, which may enable larger improvements to latency at the cost of model accuracy.

We extended ARMCL with low-latency GELU and LayerNorm operations used in BERT. We implement an ARMCL BERT framework that runs on mobile CPU and GPU, and supports fp32, fp16, and int8 data types. Measuring latency on the HiKey970 and Galaxy A52, we show 50% improvement using fp16 CPU operations, and over 50% using embedded GPU compared to CPU. We also show that int8 quantization can improve latency by up to 67% depending on the hardware platform. Additionally, we test a couple variants of the original BERT model, most notably I-BERT, finding that the integer arithmetic used in I-BERT enables a further 14% speedup over a standard int8 BERT.

Using our ARMCL BERT implementation for latency measurement, we were able to build a second optimizer that incorporates quantization with heterogeneous optimization. To account for the accuracy loss caused by quantization, we build an AutoML-based accuracy model, trained on data generated using AdaQuant [72]. To handle the new multi-objective

optimization problem, we utilize a genetic algorithm variant called NSGA-II, which is able to incorporate the optimizations of DNN-SMO. Using our NSGA-II based [66] optimizer, we find that we are able to improve latency by 20% compared to full int8 quantization while maintaining the accuracy of fp32 inference.

7.1 Future Work

Our work has shown encouraging results for acceleration of BERT inference using heterogeneous computing. However, there are multiple directions for future work. Firstly, we could include the usage of NPU hardware as well as the current CPU and GPU. This would be interesting due to the increase in energy efficiency and compute performance provided by NPUs. Secondly, we could investigate the usage of heterogeneous computing within operations. Currently, we are only able to parallelize at the DNN computation graph level. However, it may be possible to further improve latency if large operations such as matrix multiplication, and LayerNorm could be parallelized across multiple processors. Finally, due to time constraints, our experimentation with multi-objective optimization is somewhat limited. It would be ideal to test various other optimization methods and find new and better ways to enhance this implementation.

7.2 Acknowledgement

This work was supported in part by Huawei Technologies Canada Inc. through the McGill Edge Intelligence Lab.

Bibliography

[1] J. Devlin, M. Chang, K. Lee, and K. Toutanova, "BERT: pre-training of deep bidirectional transformers for language understanding," in *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2019, Minneapolis, MN, USA, June 2-7, 2019, Volume 1 (Long and Short Papers)* (J. Burstein, C. Doran, and T. Solorio, eds.), pp. 4171–4186, Association for Computational Linguistics, 2019.

[2] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," in *Advances in neural information processing systems*, pp. 5998–6008, 2017.

[3] F. N. Iandola, A. E. Shaw, R. Krishna, and K. Keutzer, "Squeezebert: What can computer vision teach NLP about efficient neural networks?," in *Proceedings of SustaiNLP: Workshop on Simple and Efficient Natural Language Processing, SustaiNLP@EMNLP 2020, Online, November 20, 2020* (N. S. Moosavi, A. Fan,

V. Shwartz, G. Glavas, S. R. Joty, A. Wang, and T. Wolf, eds.), pp. 124–135, Association for Computational Linguistics, 2020.

[4] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna, "Rethinking the inception architecture for computer vision," 2015.

[5] F. N. Iandola, S. Han, M. W. Moskewicz, K. Ashraf, W. J. Dally, and K. Keutzer, "Squeezenet: Alexnet-level accuracy with 50x fewer parameters and $<$ 0.5 mb model size," *arXiv preprint arXiv:1602.07360*, 2016.

[6] Google, "Google translate." `https://ai.googleblog.com/2020/06/recent-advances-in-google-translate.html` Accessed: 2022-10-20.

[7] Statista, "Number of internet of things (iot) connected devices worldwide from 2019 to 2021, with forecasts from 2022 to 2030," 2022. `https://www.statista.com/statistics/1183457/iot-connected-devices-worldwide/` Accessed: 2022-10-20.

[8] S. Dong, P. Wang, and K. Abbas, "A survey on deep learning and its applications," *Computer Science Review*, vol. 40, p. 100379, 2021.

[9] Amazon, "Amazon Alexa," `https://developer.amazon.com/en-US/alexa` Accessed: 2022-10-20.

[10] Google, "Google Home Nest," `https://store.google.com/product/nest_hub_2nd_gen?hl=en-GB` Accessed: 2022-10-20.

[11] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, *et al.*, "In-datacenter performance analysis of a tensor processing unit," in *Proceedings of the 44th annual international symposium on computer architecture*, pp. 1–12, 2017.

[12] Z. Xiao and Y. Xiao, "Security and privacy in cloud computing," *IEEE Communications Surveys Tutorials*, vol. 15, no. 2, pp. 843–859, 2013.

[13] D. Kollias, A. Tagaris, A. Stafylopatis, S. Kollias, and G. Tagaris, "Deep neural architectures for prediction in healthcare," *Complex & Intelligent Systems*, vol. 4, pp. 119–131, Jun 2018.

[14] N. D. Lane, S. Bhattacharya, P. Georgiev, C. Forlivesi, L. Jiao, L. Qendro, and F. Kawsar, "Deepx: A software accelerator for low-power deep learning inference on mobile devices," in *2016 15th ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN)*, pp. 1–12, 2016.

[15] T. Tambe, C. Hooper, L. Pentecost, E. Yang, M. Donato, V. Sanh, A. M. Rush, D. Brooks, and G. Wei, "Edgebert: Optimizing on-chip inference for multi-task NLP," *CoRR*, vol. abs/2011.14203, 2020.

[16] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 770–778, 2016.

[17] T. Instruments, "Msp430fr5994," 2018. `https://www.ti.com/product/MSP430FR5994` Accessed: 2022-10-20.

[18] Q. He, B. Segee, and V. Weaver, "Raspberry pi 2 b+ gpu power, performance, and energy implications," in *2016 International Conference on Computational Science and Computational Intelligence (CSCI)*, pp. 163–167, IEEE, 2016.

[19] A. Ignatov, R. Timofte, A. Kulik, S. Yang, K. Wang, F. Baum, M. Wu, L. Xu, and L. Van Gool, "Ai benchmark: All about deep learning on smartphones in 2019," in *2019 IEEE/CVF International Conference on Computer Vision Workshop (ICCVW)*, pp. 3617–3635, 2019.

[20] W. Liu, P. Zhou, Z. Zhao, Z. Wang, H. Deng, and Q. Ju, "Fastbert: a self-distilling BERT with adaptive inference time," 2020.

[21] J. Hestness, S. Narang, N. Ardalani, G. F. Diamos, H. Jun, H. Kianinejad, M. M. A. Patwary, Y. Yang, and Y. Zhou, "Deep learning scaling is predictable, empirically," 2017.

[22] S. Wang, G. Ananthanarayanan, Y. Zeng, N. Goel, A. Pathania, and T. Mitra, "High-throughput CNN inference on embedded ARM big.little multi-core processors," *CoRR*, vol. abs/1903.05898, 2019.

[23] J. Tarnawski, A. Phanishayee, N. R. Devanur, D. Mahajan, and F. N. Paravecino, "Efficient algorithms for device placement of DNN graph operators," *CoRR*, vol. abs/2006.16423, 2020.

[24] D. Kang, J. Oh, J. Choi, Y. Yi, and S. Ha, "Scheduling of deep learning applications onto heterogeneous processors in an embedded device," *IEEE Access*, vol. 8, pp. 43980–43991, 2020.

[25] Y. Kim, J. Kim, D. Chae, D. Kim, and J. Kim, "µlayer: Low latency on-device inference using cooperative single-layer acceleration and processor-friendly quantization," in *Proceedings of the Fourteenth EuroSys Conference 2019*, pp. 1–15, 2019.

[26] W. Kang, K. Lee, J. Lee, I. Shin, and H. S. Chwa, "Lalarand: Flexible layer-by-layer cpu/gpu scheduling for real-time dnn tasks," in *2021 IEEE Real-Time Systems Symposium (RTSS)*, pp. 329–341, 2021.

[27] Y. Kim, J. Kim, D. Chae, D. Kim, and J. Kim, "ulayer: Low latency on-device inference using cooperative single-layer acceleration and processor-friendly quantization," *EuroSys '19*, 2019.

[28] T. Chen, T. Moreau, Z. Jiang, H. Shen, E. Q. Yan, L. Wang, Y. Hu, L. Ceze, C. Guestrin, and A. Krishnamurthy, "TVM: end-to-end optimization stack for deep learning," *CoRR*, vol. abs/1802.04799, 2018.

[29] A. Mirhoseini, H. Pham, Q. V. Le, B. Steiner, R. Larsen, Y. Zhou, N. Kumar, M. Norouzi, S. Bengio, and J. Dean, "Device placement optimization with reinforcement learning," in *International Conference on Machine Learning*, pp. 2430–2439, PMLR, 2017.

[30] T. Braun *et al.*, "A comparison of eleven static heuristics for mapping a class of independent tasks onto heterogeneous distributed computing systems," *Elsevier JPDC*, vol. 61, no. 6, pp. 810–837, 2001.

[31] E. Aghapour, A. Pathania, and G. Ananthanarayanan, "Integrated arm big.little-mali pipeline for high-throughput cnn inference," *TechRxiv*, Jul 2021.

[32] V. Sanh, L. Debut, J. Chaumond, and T. Wolf, "DistilBERT, a distilled version of BERT: smaller, faster, cheaper and lighter," *arXiv preprint arXiv:1910.01108*, 2019.

[33] Z. Sun, H. Yu, X. Song, R. Liu, Y. Yang, and D. Zhou, "Mobilebert: a compact task-agnostic bert for resource-limited devices," in *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pp. 2158–2170, 2020.

[34] S. Kim, A. Gholami, Z. Yao, M. W. Mahoney, and K. Keutzer, "I-BERT: Integer-only BERT quantization," pp. 5506–5518, 2021.

[35] Y. Bondarenko, M. Nagel, and T. Blankevoort, "Understanding and overcoming the challenges of efficient transformer quantization," in *Proceedings of the 2021 Conference*

on Empirical Methods in Natural Language Processing, (Online and Punta Cana, Dominican Republic), pp. 7947–7969, Association for Computational Linguistics, Nov. 2021.

[36] A. Bhandare, V. Sripathi, D. Karkada, V. Menon, S. Choi, K. Datta, and V. Saletore, "Efficient 8-bit quantization of transformer neural machine language translation model," *arXiv preprint arXiv:1906.00532*, 2019.

[37] M. A. Gordon, K. Duh, and N. Andrews, "Compressing BERT: Studying the effects of weight pruning on transfer learning," 2020.

[38] V. Sanh, T. Wolf, and A. Rush, "Movement pruning: Adaptive sparsity by fine-tuning," *Advances in Neural Information Processing Systems*, vol. 33, pp. 20378–20389, 2020.

[39] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural network with pruning, trained quantization and huffman coding," in *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings* (Y. Bengio and Y. LeCun, eds.), 2016.

[40] T.-J. Yang, Y.-H. Chen, and V. Sze, "Designing energy-efficient convolutional neural networks using energy-aware pruning," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 5687–5695, 2017.

[41] Huawei, "Hikey970," 2018. `https://www.96boards.org/product/hikey970/` Accessed: 2022-10-20.

[42] H. Topcuoglu, S. Hariri, and M.-Y. Wu, "Performance-effective and low-complexity task scheduling for heterogeneous computing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 13, no. 3, pp. 260–274, 2002.

[43] F. Ferrandi, P. L. Lanzi, C. Pilato, D. Sciuto, and A. Tumeo, "Ant colony heuristic for mapping and scheduling tasks and communications on heterogeneous embedded systems," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 29, no. 6, pp. 911–924, 2010.

[44] R. Ayari, I. Hafnaoui, G. Beltrame, and G. Nicolescu, "Imga: An improved genetic algorithm for partitioned scheduling on heterogeneous multi-core systems," *DAES*, vol. 22, p. 183–197, jun 2018.

[45] A. Paliwal, F. Gimeno, V. Nair, Y. Li, M. Lubin, P. Kohli, and O. Vinyals, "Reinforced genetic algorithm learning for optimizing computation graphs," 2020.

[46] S.-h. Kang, D. Kang, H. Yang, and S. Ha, "Real-time co-scheduling of multiple dataflow graphs on multi-processor systems," in *2016 ACM/EDAC/IEEE DAC*, pp. 1–6, 2016.

[47] L. Hou, Z. Huang, L. Shang, X. Jiang, X. Chen, and Q. Liu, "Dynabert: Dynamic bert with adaptive width and depth," *Advances in Neural Information Processing Systems*, vol. 33, pp. 9782–9793, 2020.

[48] Z. Lan, M. Chen, S. Goodman, K. Gimpel, P. Sharma, and R. Soricut, "Albert: A lite bert for self-supervised learning of language representations," in *International Conference on Learning Representations*, 2020.

[49] O. Zafrir, G. Boudoukh, P. Izsak, and M. Wasserblat, "Q8BERT: Quantized 8bit BERT," pp. 36–39, 2019.

[50] W. Kang, K. Lee, J. Lee, I. Shin, and H. S. Chwa, "LaLaRAND: Flexible layer-by-layer cpu/gpu scheduling for real-time dnn tasks," in *2021 IEEE Real-Time Systems Symposium (RTSS)*, pp. 329–341, IEEE, 2021.

[51] Y. Zhu, R. Kiros, R. Zemel, R. Salakhutdinov, R. Urtasun, A. Torralba, and S. Fidler, "Aligning books and movies: Towards story-like visual explanations by watching movies and reading books," in *Proceedings of the IEEE international conference on computer vision*, pp. 19–27, 2015.

[52] A. Wang, A. Singh, J. Michael, F. Hill, O. Levy, and S. R. Bowman, "GLUE: A multi-task benchmark and analysis platform for natural language understanding," in *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*, OpenReview.net, 2019.

[53] D. Hendrycks and K. Gimpel, "Bridging nonlinearities and stochastic regularizers with gaussian error linear units," *CoRR*, vol. abs/1606.08415, 2016.

[54] Y. Wu, M. Schuster, Z. Chen, Q. V. Le, M. Norouzi, W. Macherey, M. Krikun, Y. Cao, Q. Gao, K. Macherey, *et al.*, "Google's neural machine translation system: Bridging the gap between human and machine translation," *arXiv preprint arXiv:1609.08144*, 2016.

[55] A. Conneau, D. Kiela, H. Schwenk, L. Barrault, and A. Bordes, "Supervised learning of universal sentence representations from natural language inference data," in *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing*, (Copenhagen, Denmark), pp. 670–680, Association for Computational Linguistics, Sept. 2017.

[56] O. Zafrir, G. Boudoukh, P. Izsak, and M. Wasserblat, "Q8bert: Quantized 8bit bert," in *2019 Fifth Workshop on Energy Efficient Machine Learning and Cognitive Computing - NeurIPS Edition (EMC2-NIPS)*, pp. 36–39, 2019.

[57] S. Kim, A. Gholami, Z. Yao, M. W. Mahoney, and K. Keutzer, "I-BERT: Integer-only BERT quantization," pp. 5506–5518, 2021.

[58] Tensorflow, "Tensorflow lite." `https://www.tensorflow.org/lite` Accessed: 2022-10-20.

[59] J. Roesch, S. Lyubomirsky, L. Weber, J. Pollock, M. Kirisame, T. Chen, and Z. Tatlock, "Relay: a new IR for machine learning frameworks," in *Proceedings of the 2nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, 2018.

[60] ARM, "Arm compute library," 2017. `https://www.arm.com/technologies/compute-library`.

[61] "Arm corelink cci-550." `https://developer.arm.com/Processors/CoreLink%20CCI-550`.

[62] S. Mirjalili, *Genetic Algorithm*, pp. 43–55. Cham: Springer International Publishing, 2019.

[63] S. Mirjalili, S. M. Mirjalili, and A. Lewis, "Grey wolf optimizer," 2014.

[64] J. H. Holland, "Genetic algorithms," *Scientific american*, vol. 267, no. 1, pp. 66–73, 1992.

[65] K. Tang, K. Man, S. Kwong, and Q. He, "Genetic algorithms and their applications," *IEEE Signal Processing Magazine*, vol. 13, no. 6, pp. 22–37, 1996.

[66] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan, "A fast and elitist multiobjective genetic algorithm: Nsga-ii," *IEEE transactions on evolutionary computation*, vol. 6, no. 2, pp. 182–197, 2002.

[67] X. Jiao, Y. Yin, L. Shang, X. Jiang, X. Chen, L. Li, F. Wang, and Q. Liu, "Tinybert: Distilling BERT for natural language understanding," 2019.

[68] A. Gad, "Pygad: An intuitive genetic algorithm python library," 2021.

[69] A. Khetan and Z. S. Karnin, "schubert: Optimizing elements of BERT," *CoRR*, vol. abs/2005.06628, 2020.

[70] M. Abramowitz, I. A. Stegun, and R. H. Romer, "Handbook of mathematical functions with formulas, graphs, and mathematical tables," 1988.

[71] D. Zhang, J. Yang, D. Ye, and G. Hua, "Lq-nets: Learned quantization for highly accurate and compact deep neural networks," in *Proceedings of the European conference on computer vision (ECCV)*, pp. 365–382, 2018.

[72] I. Hubara, Y. Nahshan, Y. Hanani, R. Banner, and D. Soudry, "Improving post training neural quantization: Layer-wise calibration and integer programming," *CoRR*, vol. abs/2006.10518, 2020.

[73] M. Feurer, A. Klein, K. Eggensperger, J. Springenberg, M. Blum, and F. Hutter, "Efficient and robust automated machine learning," *Advances in neural information processing systems*, vol. 28, 2015.

[74] F. Hutter, L. Kotthoff, and J. Vanschoren, eds., *Automated Machine Learning - Methods, Systems, Challenges*. Springer, 2019.

[75] L. Zheng and T. Chen, "Optimizing deep learning workloads on arm gpu with tvm," in *Proceedings of the 1st on Reproducible Quality-Efficient Systems Tournament on Co-Designing Pareto-Efficient Deep Learning*, ReQuEST '18, (New York, NY, USA), Association for Computing Machinery, 2018.

[76] A. A. Heidari, S. Mirjalili, H. Faris, I. Aljarah, M. Mafarja, and H. Chen, "Harris hawks optimization: Algorithm and applications," *Future Generation Computer Systems*, vol. 97, pp. 849–872, 2019.

[77] S. Kim, S. Oh, and Y. Yi, "Minimizing gpu kernel launch overhead in deep learning inference on mobile gpus," HotMobile '21, (New York, NY, USA), p. 57–63, Association for Computing Machinery, 2021.